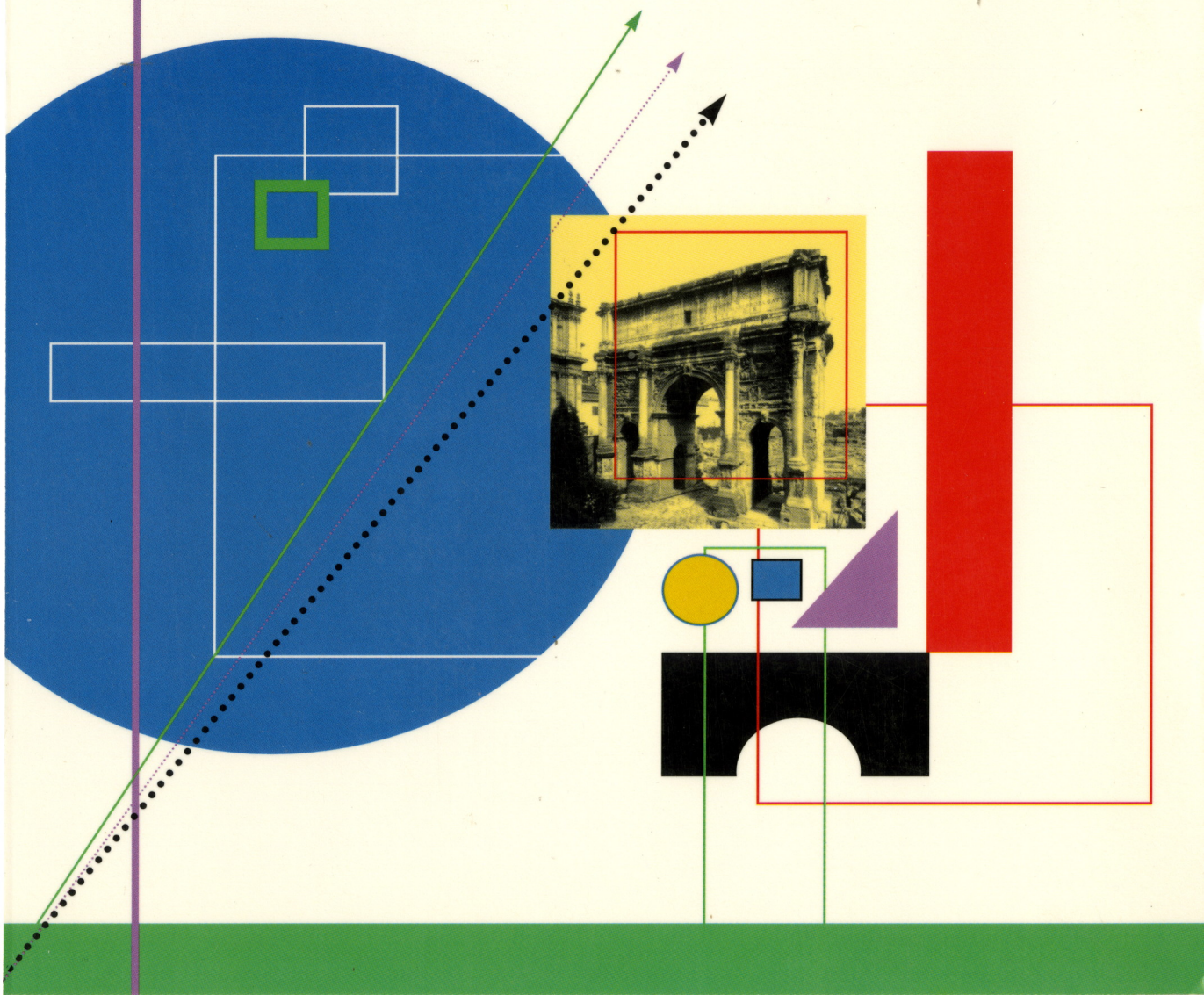
Designing High-Powered OS/2® Warp Applications

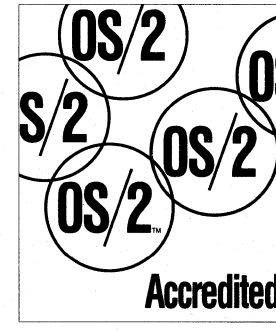
The Anatomy of Multithreaded Programs

David E. Reich



DESIGNING HIGH-POWERED OS/2® WARP APPLICATIONS

The Anatomy of Multithreaded Programs



David E. Reich

John Wiley & Sons, Inc.

New York • Chichester • Brisbane • Toronto • Singapore

Publisher: Katherine Schowalter

Editor: Theresa Hudson

Managing Editor: Maureen B. Drexel

Text Design & Composition: Integre Technical Publishing Co., Inc.

ISBN 0-471-11586-X

Printed in the United States of America

10 9 8 7 6 5 4 3 2 1

TRADEMARKS

IBM is a registered trademark of International Business Machines Corporation.

Operating System/2, OS/2, Presentation Manager, Workplace Shell, Information Presentation Facility, Systems Application Architecture (SAA), Common User Access (CUA), PS/2, VisualAge, and Hyperwise are trademarks or registered trademarks of International Business Machines Corporation.

CIL is a trademark of Component Integration Laboratories.

PowerPC is a trademark of IBM Corporation.

Mach3 is a trademark of Carnegie Mellon University.

Intel is a registered trademark of Intel Corporation.

Lotus is a trademark of Lotus Development Corporation.

Aldus is a trademark of Aldus Corporation.

PageMaker is a registered trademark of Aldus Corporation.

KASE:PM VIP is a trademark of Kase Systems, Inc.

SmallTalk is a registered trademark of Digitalk Inc.

GPF is a trademark of Microformatic Inc.

Microsoft is a registered trademark of Microsoft Corporation.

Mirrors and SMART are trademarks of One Up Corporation.

VX/Rexx is a registered trademark of Watcom, Inc.

Vis/Pro REXX is a trademark of Hockware, Inc.

Windows is a trademark of Microsoft Corporation.

To Stacy, whose constant love and support keep me going and for Liora, who will always be my light.

FOREWORD

In April 1992, IBM opened a whole new world of computing with the introduction of 32-bit OS/2. OS/2 is the integrating platform. For the first time, computer users can run virtually any application written for DOS, Windows, or OS/2 and run them simultaneously. Moreover, OS/2 provides the facilities for you, the application designers and developers, to create faster, more powerful, more flexible applications than ever before. With the introduction of OS/2 Warp, this power has been updated and enhanced to run faster, use less memory, and provide vastly more functions than ever before.

All this power lies waiting for you to utilize in your full-fledged 32-bit applications. You are free of 64k segments and 640k boundaries, memory extenders, and other add-ons loaded on top of each other. You want memory? You got it. You have the facilities of powerful graphical user and programming interfaces at your fingertips. One of the less tangible yet most powerful features of OS/2 is its preemptive, prioritized multithreading model. Other systems simulate multitasking through simple time slicing. OS/2 brings program execution to the thread, allowing you to write different parts of your application to run in parallel, creating better throughput for the application and higher productivity for the end user through more efficient use of the computer hardware. These, along with the other technological advances of OS/2, are wrapped in a state-of-the-art, object-oriented user shell.

In working with developers all over the world, David has heard the questions, seen the hurdles, and helped overcome them. Through his

work in the OS/2 development organization, as well as his work with developers, he has gained insight into what is needed to design and develop high-performance applications that exploit all that OS/2 has to offer.

This book contains that insight and answers the questions of how to structure and design your applications to take the best advantage of OS/2's features. You don't have to use everything that is in OS/2 just because it is there. Throughout this book you will be shown how to effectively break tasks into threads and which functions are more appropriate in different situations. Once you decide on a task, you are shown how to optimize your usage of system functions and resources to create the fastest, most robust applications possible.

We at IBM are committed to OS/2 and proud of our technical accomplishments with OS/2 Warp. We also know that we need you and your applications to make our system really shine. David has the ability to explain the most technically involved topics in ways that everyone can understand. Use his experience, and this book, as you advisor in creating the 32-bit applications of the future.

Lee Reiswig
President, IBM Personal Software Products Division

ABOUT THE AUTHOR

David E. Reich is currently an OS/2 Development Manager with IBM in Boca Raton, Florida and has been with the OS/2 development team since 1987. In that time, he has worked on many parts of the operating system, supported customers and application developers, and has traveled the world teaching classes and helping developers write OS/2 applications. Mr. Reich has a Master of Science degree in computer science from the State University of New York at Albany, and is a contributing editor to *OS/2 Magazine* and *OS/2 Developer* magazine.

ACKNOWLEDGMENTS

First and foremost, thanks to all the designers, developers, testers, writers, and everyone who is part of the OS/2 development team. You are the most talented, dedicated, and craziest bunch of people I've ever had the pleasure of working with. Due to your hard work, long hours, and sleepless nights, OS/2 is the step into the future the PC world needs. OS/2 turned from a software product into a labor of love, and you made it happen.

Special thanks go to the following:

Darren Miclette, a good friend and an amazing talent. We will all miss you, but your work lives on as does the memory of all the good times.

Kelvin Lawrence, for whom there are no words. All I can say to the man who believes "sleep is for the weak" is that you have to be the most knowledgeable person on OS/2 there is. Thank you for your friendship and for being there to help with those questions too numerous, obscure, and involved for anyone else to understand what I was talking about, and for your immeasurable contribution to the product.

James Taylor and Brett Adams, for being good friends and for being there.

Dan Kehn, for introducing me to the architecture of SOM and the Workplace Shell during that great seafood lunch.

Peter Magid, for being my sounding board on all of my questions and crazy ideas on the shell and what it can really do.

Ari Erev, for all those questions you always ask me that make me take a look deep into how the functions work; that educates me as well.

Glen Brew, Paul Devriendt, and Edd Doutre, for helping with all those nuances of memory management and how OS/2 starts and manages the millions of things going on in the system every second.

David Kerr, whose expertise never ceases to amaze me. Thank you for your advice on techniques and all those tips that are so often overlooked when putting code together.

Michelle Vaghari and Laura Sanders, for giving me a chance and your support of all my activities, including my writing work.

Joe Tano and Ken Christopher for your perseverence and faith in my abilities.

Kevin Maier, for all kindsa stuff.

David Moskowitz, for always keeping me on my toes.

Of course, I can't forget Mike Santivenere and Ed Potts.

To TEAMOS2, application developers, IBMers, and OS/2 supporters alike. Without your tireless efforts, OS/2 would not have achieved the levels of success it has in so short a time. You are visionaries, and the world will look back and ask you how you knew what the future would be before it happened.

Finally, and most importantly, to all the families of the entire OS/2 team for sacrificing the time apart from your loved ones to make the OS/2 dreams reality. Without your support, none of us would have made it through with what little sanity we have left intact.

CONTENTS

Introduction xxiii

About This Book xxv
Overall Design xxvi
Features xxvii
Maintainability xxviii
Tools xix
Testing and Code Change xxx
Install Programs xxx

Section I Why OS/2? 1

Chapter 1 OS/2 as an End-User Platform 3

Multiprocessing 4
Using Several Applications at Once 4
Sharing and Communicating Data 5
Multithreading 6
Dividing Applications into Parallel Pieces 6
Increased User Productivity 7
Better Performance 7
32-Bit Memory Management 8
Flat Memory Model 8
Independence from Physical Memory 9
Up to 512 Meg per Program 9
Paging versus Swapping 10

Better Application Performance 10
Better System Performance 11
Intuitive User Interface 11
Users Can View Many Applications at Once 12
Workplace Shell 13
Consistent Behavior 13
Lower Learning Curve 13
Object-Oriented 13
Works the Way People Work 14
Contextual Help 15
Learning Is Easy 15
Device Independence 16
Applications See Generic Output Space 16
New Devices Only Need New Drivers, Not New Applications 16
IBM Compatibility to Future Releases 17
Symmetric Multiprocessing 17
Portability to PowerPC 18
Summary 19

Chapter 2 Why Program for OS/2? 21

Powerful and Flexible API 21
Function Call Interface to All System Services 23
Consistent, Easy Coding 23
Portability, Flexibility, Expandability 24
Easy to Modularize and Maintain 25
System Coding Conventions 26
Fast Prototyping 27
OS/2 Development: The Next Generation 27
Summary 28

Chapter 3 OS/2 as a Development Platform 29

Multitasking for Development 30
Debugger Support 30
Crash Protection 32
Summary 32

Section II Overall Application Design 33

Chapter 4 Good Programs Have Good Up-Front Design 37

Understanding the Target Environment 38
60 Percent Design, 30 Percent Code, 10 Percent Test 38
60 Percent Design 39
30 Percent Code 40
10 Percent Test 41
Summary 42

Chapter 5 OS/2 Kernel Architecture 43

Overview of the Kernel 44
Structure 44
Scheduler/Dispatcher 47
Scheduling on SMP 48
Loader 49
System Services Flow 50
Digging Deeper 53
Protection Mechanism 54
Process/Thread Model 55
Priority 57
Thread Management 59
Memory Management 60
File System 68
Device Drivers 69
DLL Mechanism 72
Base Subsystems 73
Summary 74

Chapter 6 Presentation Manager, Graphics, and the User Interface 75

Presenting Data 76
Presentation and Translation Flow 78
Device Context 78
Presentation Space 80
Tracing a Drawing Call 80

Graphics Engine 81
Presentation Drivers 84
Brute-Force versus Full-Function Drivers 84
Printer Drivers 85
Screen Drivers 85
Window Manager 86
Input 86
Workplace Shell 89
Summary 93

Chapter 7 OS/2 Warp for the PowerPC 95

What Is Workplace? 96
Comparing and Contrasting with Intel OS/2 97
Microkernel 100
Personality-Neutral Services 102
Out-of-Kernel Device Drivers 104
Personalities 105
The OS/2 Personality 106
OS/2 Client Library 106
Presentation Manager and Other Libraries 107
The OS/2 Server 108
Single-Source Applications 109
Summary 110

Chapter 8 Features for Your Application 113

What Is the Main Function or Objective of the Application? 114
How Will the Application Be Presented? 115
Presentation Manager Graphics/Text 115
AVIO Windows 118
Text-Based 119
Workplace Shell Objects and Templates 121
Choosing Features to Include 123
Data Communications with Other Programs
(Open Application) 124

Dynamic Data Exchange 124
Clipboard 126
Pipes and Queues 127
Data Interchange Formats and Filters 129
REXX Hooks 130
Printing 131
Fonts and WYSIWYG 131
Summary 132

Chapter 9 Application Structure 133

Isolate from Underlying Hardware 134
Use OS/2's Device Independence 135
Stick with Portable Languages and Tools 136
Modular Design and Threads 138
Using Multiple Processes 141
Using DLLs and Code Sharing 142
Resource Sharing and Synchronization 143
Keep Upgradability, Portability, and Serviceability in Mind 145
Summary 145

Section III Use Building Blocks or Your App Will Crumble 147

Chapter 10 Block Design and Architecture 149

Taking the Black-Box Approach 149
Letting the Operating System Do It for You 152
Summary 153

Chapter 11 Designing the User Interface 155

Window Design 155
CUA 156
Presenting Data 157
Window Controls 158
Dialogs 159

Application Defaults 163
Application Development Tools for OS/2 164
Object Technologies 170
Workplace Shell 172
Objects and Templates 173
Step 1—Simple Drag-and-Drop 173
Step 2—Associations 174
Step 3—Writing an Object 176
Application Launching 176
Single-Process Model Considerations 178
Interprocess Objects and Agents 179
Subset Function 180
Object-Oriented Printing 180
To Write an Object, or Not to Write an Object 183
Summary 183

Chapter 12 Where's the Beef? 185

Designing the Core 186
Modularizing the "Worker" Code 187
Memory Management 188
Device Drivers and Device Independence 192
File Layout 195
HPFS and FAT Features 196
`.EXE`s and `.DLL`s 197
`INI` Files 199
Multiprocess (Multiprogram) Applications 200
Summary 202

Section IV Making It Happen 203

Chapter 13 The Development Environment 205

Source Code Control 206
Platform 206
Function 207
Problem Tracking 208

Tree Structures 209
Tools 210
Summary 211

Chapter 14 Prototyping the User Interface 213

Paint Your Windows 214
Multithreading Considerations 218
Handling Long Jobs 221
Using PM's Thread Handling 222
Object Windows 223
User-Defined Window Messages 225
WinPostQueueMsg 226
Keeping the User Interface Thread Responsive 228
Summary 229

Chapter 15 Building the Core Function 231

Memory Manager Package 233
Memory Suballocation 234
Guard Pages and Exceptions 237
16- and 32-Bit Techniques and Coexistence 238
Multithreading 240
Synchronizing Threads 241
Semaphores 242
Critical Sections 246
Should You Even Use Another Thread? 247
IPC 248
Queues 248
Pipes 249
Shared Memory 250
File Functions 251
Internal File Formats 251
Taking the LAN into Account 252
Using File System Structures 253
Summary 254

Chapter 16 Using Advanced Functions 257

Clipboard 258
Text Data 259
Metafile Data 259
Application-Defined Data 260
Dynamic Data Exchange (DDE) 260
Printing 262
Print Destinations 264
Fonts 266
Help Facilities 267
Help and Multimedia 269
Summary 270

Chapter 17 Non-English-Language Support 273

Flexibility in Your Code 273
Message Files 274
Windows and Dialogs 275
Stringtables 275
Building Windows on-the-Fly 276
Using Lengths and Proportions 277
Resources in DLLs 279
Structuring Your Development 280
Summary 282

Section V Performance 285

Chapter 18 Base Tuning 287

Memory Tuning 287
Code and Data Working Sets 288
Locality of Reference and Data Positioning 288
Code Size versus Path 289
Dynamic Link Considerations 290
Messages and Other Resources 294

Throughput Using Threads 295
Thread Priority 295
Packing the Executable 298
DLL Placement 298
Summary 300

Chapter 19 Visual Tuning 301

Window Tuning Tips 301
Keeping Windows Around 302
Filling Windows Invisibly 302
Letting PM Manage Work 303
Your Own Multipurpose Classes 304
Summary 306

Section VI Testing and Code Change 307

Chapter 20 Testing Methodology 309

Scaffolding 310
Testing Units 310
Testing Modules and Components 311
Testing the System 311
Summary 313

Chapter 21 Code Change 315

Code Change According to Design 316
When There Are Too Many for Comfort 317
Summary 317

Section VII Installation Programs 319

Chapter 22 Designing the Installation Program 321

Just Another (Small) Application 321
User Interface 322

Multithreading 323
Multiple Installations 324
Media Considerations 325
Packing Your Code 326
"Series" Applications 326
Summary 326

Summary and Conclusion **329**

Bibliography **331**

Index **333**

INTRODUCTION

OS/2 is the first true multitasking, virtual memory, multiprocess operating system for IBM PCs, PS/2s, and compatibles. OS/2 provides threads, huge memory objects, and myriad functions and structures to create applications that were never even thought of before. That means that all you were able to make a computer do before, you can now make it do faster and more efficiently.

OS/2 will run virtually any program written for DOS or Windows. OS/2 obviously, will run any program written for the current release of OS/2 or any prior version. So really, OS/2 will run any application written for the PC.

This is a big advantage for developers, because users can continue to run the current levels of your applications while you build the more advanced OS/2 versions. OS/2, with its interprocess communication features, will let all of these applications talk to each other even if they are a combination of DOS, Windows, and OS/2 applications, allowing users to migrate to the more powerful applications easily.

While OS/2 for Intel systems runs your applications on Intel-based computers, OS/2 Warp for the PowerPC will run them on PowerPC-based computers. The underlying goal of OS/2 is to isolate users *and* applications from hardware by providing device independence for applications, and consistent interfaces for users.

Not only is OS/2 an excellent platform to run end-user applications, but it is an excellent development platform. How many times have you grown impatient waiting for your computer to finish a compile, knowing all the while you could have been doing something, anything, else with it, such as working on another part of the project or even sending electronic mail? Well, OS/2 solves all that.

OS/2 provides a platform for users to run many applications at once. These applications can be a combination of DOS, Windows, or OS/2 programs. However, OS/2 provides not only the facilities for running many varied programs at once, it also provides system services to native OS/2 programs unlike any other before it, or even those currently trying to mimic its function.

Programs that are written specifically for OS/2 have access to large, flat memory objects. This smashes the long-imposed 64K segment limit introduced with the original PC and DOS, and which still hampers development in these environments. Among a variety of other services, OS/2 also offers true protection between programs, interprocess communication services, multiple threads of execution within a process, built-in spooling functions, and a powerful graphical subsystem, all wrapped up in an easy-to-use, object-oriented user shell. Best of all, it is a full 32-bit system with a consistent 32-bit programming interface, with the 32-bit and object technologies underneath supporting it all.

Writing programs is a generally straightforward task. Once you decide what your program will do, you generally sit down and write the code. Of course, there are a variety of algorithms you may choose to accomplish all of the tasks the program will perform, but coding is mostly a matter of learning a language such as C, and writing the code. However, as programs become more powerful and complex, they also grow. As they grow, it becomes more and more important to structure them so that they are efficient, expandable, maintainable, and make the most efficient use of all their operating system has to offer. This has to be the most important aspect of writing software today.

Proper design of an application (or series of applications) means code that is easily enhanced, maintained, and ported to other environments and code that can interface with other applications easily. This

translates into productivity gains for developers and cheaper and faster development cycles. Case in point is OS/2 Warp for the PowerPC, which as you will see in Chapter 7 is a virtually no-work-required port assuming you've designed the application well.

Failure to properly design an application means code that has to be frequently rewritten, that is hard to read, and difficult to maintain. This translates into long development cycles and programmer frustrations.

One of the reasons that OS/2 Warp for the PowerPC has been developed in such a relatively short period of time is because the subsystems of OS/2 such as the Workplace Shell and graphics subsystems were designed well, and written in high-level languages. Although they are system-level code, moving them over to the PowerPC has been much easier and faster than if they were written in assembler, or coded to specifics of the Intel processor.

I have seen my share of poorly designed (or hacked-together) code. I've also witnessed code that was so well designed that it was ported to the PowerPC in a day. In many cases, however, I have seen applications where the designers tried to do it right, but fell somewhat short. The main cause for this is that they were not aware of all the system had to offer, they were not aware of theories that help performance and maintainability and in general, they just lack the experience needed to make the most of their application running under OS/2.

This book will help you do it right—the first time.

ABOUT THIS BOOK

This is a book about designing applications. OS/2 is full of functions and features, and with them go decisions about which to use and when. Once you have decided on the functions you want to provide to your users, there are considerations about the implementation of each.

By the time you have finished this book, you will have walked through OS/2 application design, A to Z. Note that this is not a programming book. Programming books tell you how to write functions and how to

make things work. This book shows you the functions that are available in the system and how they work, providing examples and scenarios describing which is best when. That is why there is virtually no code in this book.

The design of an application entails when to put functions on separate threads, how to manage those threads, how to structure memory, and which communications mechanisms to use. You may know that you need to use a huge multidimensional matrix for a function. How should you allocate memory for it to maximize performance in all situations? This book will help you make that decision. You know you need to communicate between application windows. Should you send messages? Post messages? What is the difference? When communicating between processes, should you use Dynamic Data Exchange, pipe, queues, or the clipboard? What types of semaphores are there to coordinate access to shared data structures? This book will answer these questions and more.

Programming without design is doomed. By following the topics and examples in this book, you will see how you can take advantage of the features OS/2 offers while leaving the implementation decisions such as which windows should be subclassed or which API to use in a specific situation, to the programming stage of application development, where it belongs. In doing so, you create code upon which you can build rather than have to figure out how to add to or modify it.

This book will take you through application design from the time you get an idea for an application all the way through the finishing touches such as media packaging, non-English-language support, and the installation program.

OVERALL DESIGN

The design of an application does not only mean to evaluate and decide on various methods, algorithms, and features your programs will use and provide; it also means to do all of this *and* other work before any

coding begins. This includes understanding all of the functions provided by the host environment, how they relate to what you want to provide to your end users and how you can make the most of each function.

Not only does proper design include quantitative functionality, but this design also includes structuring your modules to make the best of the environment. This includes modularizing your code, designing your thread and process models, prototyping, performance, code size, and flexibility.

Now we also all know that requirements of a software project change during the development cycle. If you set down your functional requirements first, then heads-down code it to completion, it will likely be obsolete before you ever release it. If you do not take this fact into account, you may wind up writing your application forever, never releasing it. However, if you do your design right from the start, new requirements during development should be a trivial thing you can easily handle.

This book will show you techniques to structure your applications to make them as flexible and updatable as possible.

FEATURES

As mentioned before, OS/2 provides features to applications never before seen in the personal computing arena, and some not previously in *any* computing arena. You need to understand the features and functions available to you, the application designer and programmer, in order to make intelligent choices as to which ones you wish to take advantage of and how to make the most efficient use of each. Sections III and IV will show you what the functions are, which ones fit better in different situations and how you can make the most of each in your designs.

The most important thing you will do in your application design is break tasks into subtasks as you would with any good program, but with OS/2, you will be doing it with parallel processing in mind as well. By subdividing your tasks appropriately, understanding concurrency and control issues, your application will run reliably on Intel-based, SMP, as

well as PowerPC-based systems. The theory is the same. Simply design according to the guidelines given herein, and you're all set.

Some of these features include how to manage your program's memory and input/output services. Other topics are interprocess communications methods such as Dynamic Data Exchange (DDE), Clipboard, queues, pipes, and shared memory. You see, OS/2 protects applications from one another, so unlike DOS or Windows, programs cannot inadvertently (or on purpose in the case of some viruses) step on one another. It is very useful, however, for programs to be able to communicate with each other such as in the case of multiple-process applications or a series of related applications. Which methods are more appropriate and efficient in different situations is very important not only in terms of performance, but also in terms of application usability.

Reading a list or programming guide for OS/2 to determine which features are available in different situations is not a very efficient way of figuring out what to incorporate into your programs.

It takes someone with experience to analyze which features you would like to incorporate into your programs and to recommend the best way to implement them given the operating environment.

This book will serve as your experienced advisor when designing your applications.

MAINTAINABILITY

Another important aspect of a good application, as mentioned earlier, is maintainability. No one wants to pick up spaghetti code and have to fix a bug or add a new feature.

Creating maintainable code is more than just modularizing your functions or using meaningful variable names. It is more than just mapping out flow diagrams or control blocks.

Under OS/2, many of the features available to make code perform better and more efficiently can also be used to make your code maintainable, updatable, and serviceable. You will see how to use various

compiler options, tree structures, and module structures such as DLLs to make it less expensive to maintain.

TOOLS

An often neglected piece of application design is a good set of design and programming tools. One of your first choices should not even really be a choice, and that is which operating environment you should target.

Obviously, since you are reading this book, you are looking at OS/2 mode. But remember, OS/2 supports 16- as well as 32-bit programs. I'm sure one of the primary reasons you are writing for OS/2 is the ability to use flat, 32-bit memory model programming and never worry about segmentation ever again. So that step is already decided and taken.

The next step in your choice of tools should be which of the variety of tools you would like to use to aid your design and prototype phases of your product's development. Now, granted, there is not a large number of CASE tools available for OS/2 at the time of this writing, but the several that are out there are worth looking at. None of the ones discussed in this book target optimizing for 32-bit OS/2, but they are very good at helping you structure your program and prototype functions quickly. The other things you will need to effectively decide how to build your whole application will be described in this book.

In addition to CASE tools, you will inevitably need a compiler, which leads to another set of choices. First, which programming language you should use is a good place to start. Once you decide on that, you need to pick a compiler and set of tools that complement this choice. As will be discussed later on, the C language is the natural choice for OS/2.

Along with the language compiler, you'll need to pick a set of related tools such as MAKE, and a source code tree structure to make code updates and revisions as easy as possible. You will see techniques to help you develop versions for your code to work in languages other than English with only one source code tree.

TESTING AND CODE CHANGE

Throughout this book, I will be stressing the importance of design over testing. However, testing is, nonetheless, a vital part of the development process. Your code should, for the most part, undergo the most stringent testing during the design phase. Once the code is done, however, you must ensure that it behaves according to the design you set out for it.

If you think about this up front, the majority of your testing will be to ensure that the code works the way you intended. Many times I have seen applications that had quality code, performing to the specification to which it was designed. Unfortunately, the design did not take many things into account and as a result, the code appeared buggy.

Another item of importance is code change. As with anything else, there is a right way to do it, a wrong way to do it, and a decision point for when to cut your losses and rewrite the module in question.

We will discuss different ways to change code so as to avoid breaking the design and adding more bugs each time you fix one. You will also see suggestions on when it's time to just trash the bug-prone module and rewrite it, taking into account all you've learned from the failed module. It's important to throw away virtually all of the failed module when you do this. We will elaborate on this topic later.

INSTALL PROGRAMS

An often overlooked piece of an application is the installation program. These programs are a user's first impression of your product and with just a little bit of work can add snap to your product, not to mention aid in allowing users to upgrade versions or install fixes without either messing up their system or having to learn some messy set of instructions.

Also, a well-structured installation program will allow you to create a common look and feel for all your applications. Then, each program you

write will be easier for your users to install than the one before, because they all look alike. Large data center or administrative managers like this as well because their employees do not have to learn a new interface to use your latest technological breakthroughs.

The installation program for your application should be flexible enough to accommodate any of your user's needs such as installing on various drives or over a LAN.

Aesthetically, the installation program should resemble the application. A simple, graphical interface should do the trick, but you want to keep portability, expandability and code reuse in mind.

Use this book as your reference and advisor in writing your OS/2 applications. This book will be your guide to designing effective, efficient applications that are easy to code, test, and maintain. Many of the concepts here are applicable to *any* environment, not only OS/2—so, use the concepts in any software project you work on.

SECTION 

Why OS/2?

OS/2 is a powerful system, but the power of the system can be demonstrated only by powerful applications. OS/2 provides application developers with the tools and resources to build powerful, flexible applications. Users can see this power in the applications.

OS/2 by itself does not show its power. By running the variety of applications on the market, OS/2 gives developers the freedom to develop these powerful applications. It provides this freedom in several ways. First and foremost, developers are not precluded from selling their existing applications for DOS and Windows systems while developing the OS/2 versions, because OS/2 runs the vast majority of DOS and Windows code.

In OS/2 Warp, the user interface is the most advanced anywhere, the performance enhanced to run in lower-memory systems, making it available to a wider audience and hence, a larger market for you! That, in addition to the tools and utilities in the BonusPak, gives you even more features and functions for you to use in your applications.

Next, OS/2 gives developers the structure and flexibility to write virtually any kind of software. Developers can access hardware through a standard interface, or they can write their own. Through multi-

threading, applications can gain performance benefits and give the user throughput never before possible in the single-tasking DOS and Windows world. Using the threading information and theories described in this text, you can be assured of exceptional performance on single-processor systems and SMP machines with no changes to your code. The SMP information herein will enable you to write code to safely exploit SMP, and will make your code running on uniprocessors more robust as well.

All of this means that users can do things with their computers and in their businesses that they could not do before. All they need are the applications. This capability opens a whole new world of software development and, with that, a whole new market for productivity-enhancing applications.

CHAPTER 1

OS/2 as an End-User Platform

OS/2 is the first fully preemptive, prioritized, multitasking, multithreaded operating system for the personal computer. This means that users can now run many programs at the same time on a single machine, all with better performance and more efficient use of the hardware than ever before. OS/2 also allows users to use more memory than is physically in the computer. A graphical, window-based subsystem is provided to present multiple applications on the same screen, or desktop, at one time. This alone enables users to be more productive than with other systems. Another feature is *device independence*, which will be discussed in depth later on. For users, this means that, in contrast to DOS, when a new piece of hardware is installed into the computer (a display, for instance), the application software need not be reconfigured or reinstalled. All of this and more is wrapped up in a state-of-the-art, object-oriented user shell called the *Workplace Shell*. OS/2 Warp has further enhanced this interface based on more than two

years of user feedback and in-depth usability studies, as well as advances in technology.

In this chapter, end-user benefits of OS/2 will be explored. If you are only considering programming for OS/2, this chapter alone will help you make your decision.

MULTIPROCESSING

The discussion of the advantages of OS/2 to the end user will begin at the top level and work downward into the core of the system. At this top level is the concept of multiprocessing.

Using Several Applications at Once

Multiprocessing is the ability to run many programs, or processes, at the same time. Each program that has been started is allocated its own *virtual console*. This means that each program has its own keyboard, mouse, memory, files, and display. Herein lies the entire foundation for OS/2's complexity and power.

This type of operation is fraught with danger for applications. OS/2 goes to great lengths to protect applications from interfering with each other, either intentionally or by chance. When running under DOS, a program had no problem simply using any part of the computer; it could accidentally step on anything that it was not using. With many programs running around the system at once, all of the services of the computer must be controlled, or chaos will ensue. OS/2 manages all of this so that all the currently executing applications can peacefully coexist in a single machine. Along with this management, many features for applications have been built into the system.

Other systems may claim the ability to run multiple applications at once, but when you look closely at how they accomplish it, you will see problems in certain areas. For example, some systems will slow considerably when printing in the background and cannot hold high-speed communications lines while performing multiple tasks. OS/2 multitask-

ing is robust and makes efficient use of the processor to avoid these problems.

Not only can users run all of their applications at once, but with the graphical interface they can see them all at once as well. To top that off, they can switch between them simply with the click of a mouse. No longer do users have to spend time loading and unloading applications. All they need to do is click, switch to another program, click, and switch back, and thus become more productive instantly.

Sharing and Communicating Data

Another prominent feature that OS/2 affords to applications is the ability to communicate data. OS/2 provides many different ways for applications to talk to each other. Some are provided by the kernel services in rudimentary structures such as queues, pipes, and semaphores. Other higher-level and more powerful mechanisms are the Clipboard and Dynamic Data Exchange (DDE).

Users have different preferences in what they want to see in the programs they use. Application designers also have different preferences of which functions are provided and how these functions are presented. Also, as technology advances and as programmers come up with new functions, new applications are created.

Not all users will want the same suite of applications to get their job done. As such, it is vital to the success of any of your applications that it be able to share data with others.

From the user's standpoint, the primary concern is how easy it is to transfer the data. In many of the current DOS applications, users must use a variety of file formats to save a file and then import it, sometimes using a filter, into the target application. This is a cumbersome process. With the mechanisms provided by OS/2, you can implement public data sharing through the clipboard or a published DDE protocol, or you can implement a private mechanism for use by your series of applications only. The choices are up to you.

Another little-known feature of OS/2 interprocess communications is the ability for DOS applications to talk to OS/2 applications. This

is accomplished through *named pipes*. DOS applications cannot create named pipes, but if a named pipe is created by an OS/2 program, it can be used by a DOS program. Each of these features is available to the applications; how they are used is up to the programmers.

The main point of sharing data is that the user should only have to know how to ask other applications (1) if they share data and (2) to place the data into the target application.

Multithreading

Not only is OS/2 a multiprocessing system, it is also a *multithreading* system. More precisely, OS/2 accomplishes multiprocessing through the use of multithreading.

The *thread* is the unit of execution within OS/2. Every program that runs has at least one thread of execution. The process, or program, owns resources such as files and memory. The thread executes instructions that manipulate those resources of the process. Each process can have many of these threads.

What does this mean to users? Users can't see or touch threads, and most likely they don't know that threads are there. DOS programs don't use threads as such, but every DOS program conceptually has one thread, that being the only thread in DOS; hence the restriction that only one DOS program may run at a time under DOS. Under OS/2, each DOS program is treated as one thread so that many can run at the same time.

Dividing Applications into Parallel Pieces

What this really means to the user is not only that OS/2 uses threads to allow many programs to run at once, but also that users gain even more productivity because developers can divide their applications into parallel pieces that function independently. For instance, imagine an application that has significant overhead loading into memory, setting itself up, reading configuration files, and so on. Now picture that ap-

plication having the ability to allow the user to begin working with it before it has even finished initializing. Now imagine a large spreadsheet with complicated calculations that takes a significant amount of time to reformat and recalculate. Picture the user of this spreadsheet working on another part of the spreadsheet, or even an entirely different one within the same application, rather than sitting and waiting for the recalculation to finish.

Increased User Productivity

Using threads properly, you can write pieces of your applications to run in parallel. Of course, there are some things you can't do, such as allowing the user to print a document on one thread and change it on another. However, with proper design, you can subdivide your applications into separate pieces that can run in parallel, providing users with productivity that under DOS could be provided only by using multiple computers, which is not a very cost-effective solution.

Better Performance

More efficient and streamlined application throughput is only one advantage of threads. Remember that every process has at least one thread; some have many. OS/2 manages execution and CPU time with threads.

The OS/2 scheduler coordinates CPU time at the thread level. Each thread has a *priority* and a *context*, which is its state of execution. The scheduler dispatches threads according to priority. These priorities are modified based on many factors that will be discussed in detail later on. The result is that OS/2 makes more efficient use of the CPU than does DOS or Windows; this efficiency translates into productivity gains for the user. While one application may be waiting for a disk access to complete, another may be running at the same time that can use the CPU to reformat a document and yet another can be printing. As such, the overall system throughput is enhanced through the use of multithreading.

This function presents interesting challenges for programmers that are thoroughly explored in Sections IV and V. If threads are being

switched into and out of the processor very quickly, and being preempted at any moment, coordinating access to code and data can get tricky. This can be even further complicated by SMP machines where threads can be physically executing at the same instant in time. By using these features well, you can create applications that outperform any others.

Users can't see multiple threads, but they benefit from them in numerous ways. The more the applications take advantage of this, the more benefit the users derive.

32-BIT MEMORY MANAGEMENT

A feature of OS/2 is 32-bit memory management. Previous versions of OS/2 were 16-bit systems; that is, all processing was done with 16-bit words. Now OS/2 uses 32-bit words, freeing applications from the segmented memory model.

OS/2 can use memory in chunks of any size up to 512 megabytes. Previously, the largest memory object you could use was 64K bytes, unless you went through some contortions to simulate larger objects. OS/2 removes this longtime constraint.

With other hardware platforms coming into play, such as PowerPC, it is important to not make assumptions about available memory or even page sizes; later on in Sections IV and V, you will see how to design your applications to simply read the operating system and tune on-the-fly.

Flat Memory Model

DOS and previous versions of OS/2 used a *segmented* memory model in which all memory was segmented into 64K chunks; memory was addressed with a segment (or paragraph) address and an offset within that segment. OS/2 now presents memory to applications in a large, flat address space, using a *flat*, 32-bit address which, to applications, is just an offset somewhere between the beginning of memory and 512 megabytes.

Independence from Physical Memory

Most personal computers do not have 512 megabytes of memory. Most do not even have 16 megabytes. However, OS/2 allows applications and users to access much more memory than is physically in the machine.

Many times, under DOS, applications have run up against the end of memory, or applications needed to be written to perform unnatural acts to squeeze into the 640K limit. Programs running under OS/2 simply need to ask the computer for more memory whenever it is needed. OS/2 has the responsibility of providing that memory.

Users of DOS systems have often encountered the message stating that there is not enough memory to run an application. With OS/2 memory management and its independence from physical memory, the user will rarely, if ever, be told that he or she cannot run an application because of insufficient memory.

Up to 512 Meg per Program

Using virtual memory, each application can have an address space of up to 512 megabytes. The limit is restricted only by the amount of hard disk space on the swap drive.

Swapping is the mechanism by which OS/2 accomplishes the feat of virtualizing memory. When a memory request cannot be filled because there are no free memory blocks of the requested size, OS/2 examines memory and selects a piece to be written to the hard disk and taken out of its memory, which is now free to be allocated to the requestor. If the program that owns the chunk just swapped out subsequently needs to use it, OS/2 moves another chunk of memory out to the hard disk to make room to bring that chunk back in.

Swapping (and paging) take place completely transparently to applications. That is the beauty of allowing OS/2 to manage things and having applications request services. OS/2 lets programs think they have the same addresses throughout their execution, although, under the covers, OS/2 manages the pointers and whether the information is in memory or currently on disk.

Paging versus Swapping

Swapping is the technique whereby the operating system will take pieces of memory not currently in use and write them to the disk in a *swap file* when another memory request needs to be filled. This was introduced in OS/2 version 1.0.

Beginning with OS/2 version 2.0, OS/2 took swapping a step further. Since OS/2 became a 32-bit-only operating system and no longer needed to restrict functions to a common denominator of the 80286 chip, *paging* could be used.

Swapping in 16-bit versions of OS/2 worked on chunks of memory in any size increments up to the segment size of 64K. This was very inefficient. Paging is more or less the same as swapping, except that the chunks are the same size: 4K. Additionally, memory is managed a bit differently in a paged system. (This will be discussed in more detail in Chapter 5.)

What all of this means to the user is better performance.

Better Application Performance

Applications can use about as much memory as they like under OS/2. This means larger spreadsheets, bigger documents, and more graphics, with the same performance as the smaller ones. You don't need to code overlays; if you need memory, you use it, and OS/2 does the rest. This means consistency for coders and performance for users.

Part of the performance gain stems from the fact that segments are a thing of the past. Because OS/2 uses the flat memory model, segment registers are no longer needed. This was one of the biggest performance inhibitors of previous releases. In the OS/2 protection model, whenever a segment register was loaded, permission checks were done to ensure that a violation was not taking place. For example, if an application loaded the code segment register with a data segment pointer, a protection exception occurred; likewise for a segment number that the application did not own.

This permission-checking overhead made it expensive to keep loading segment registers, but it had to be done for all intersegment jumps,

such as when a program called system services. With the flat memory model, there is one segment value, which represents the entire flat memory object of system memory. The end of segment register loads equals better performance.

Better System Performance

Along with better application performance due to 32-bit flat memory comes better overall system performance. The biggest reason for this is that OS/2 is using fixed-size, 4K pages rather than variable size segments. Because OS/2 is using 4K pages, there no longer needs to be memory compaction due to holes in memory of inadequate size. Every space left vacant in memory is 4K in size, and every memory allocation is 4K in size. This translates into better memory performance due to low overhead.

This arrangement does create a challenge for the programmer, however. It used to be convenient and economical to allocate memory in small chunks and only when needed. With this 32-bit, 4K page scheme, the small-allocation approach will waste memory. Every allocation is 4K in size, so if you need 10 bytes and ask for an allocation of 10 bytes, you'll actually get 4K. As you will see later on, this is not necessarily a bad thing. Used correctly, it can actually help you in managing memory within your application.

The flat memory model, along with the 4K fixed-page-size paging algorithms, helps applications perform better than they do on other systems on the market and helps the system itself perform better.

INTUITIVE USER INTERFACE

One of the first things a user sees after booting up an OS/2 system is a screen with pictures. These pictures are representative of all the actions you can take with your computer. You can open objects such as folders and documents; move objects from place to place, such as from folder

to folder or to a printer; and access various parts of these objects in a consistent way.

For example, all OS/2 objects bring up a *context menu* when the user clicks button 2 on the object. Double-clicking button 1 on the object will open it. It is this type of consistency, combined with the graphical representation of these objects, that makes the interface easy to learn and use.

OS/2 Warp introduced enhancements to this user interface, including a launch pad for starting the most frequently used applications with a single button click. An open folder now shows not only an in-use emphasis, but changes to a picture of an open folder too. In addition, many of the formerly private programming interfaces to the shell are exposed for your use.

Users Can View Many Applications at Once

The biggest user benefit of a graphical user interface (GUI) is the ability to view all of the currently running applications at the same time.

With a pure character-based system, an application consumes the screen. For instance, in OS/2 version 1.0, although many applications could be running at the same time, only one was visible at a time. With a GUI, each application is given its own real estate on the screen, called a window.

Applications present their data within their windows. Users can change the size and position of each window or make them invisible. Even applications that are text-based can run within a window. OS/2 just maps their I/O calls to the screen to a window, so users can see all applications at once, not just the ones written to the GUI. What this means to users is that they can see what is going on with all their applications all the time, or for as much of the time as they wish.

OS/2 did not invent the GUI, but it has refined it into a more efficient, integrated environment that permeates and works tightly with the operating system itself.

Workplace Shell

The OS/2 graphical subsystem is called the Presentation Manager. The GUI, however, is the Workplace Shell. The *shell* is what wraps up all the functions of the interface into an integrated package, providing easy, consistent access to all system services. This includes managing files, configuring the system, and manipulating objects, as well as the primary function of a shell: launching programs. The Workplace Shell is built on top of the Presentation Manager, which provides the graphics functions, screen management, and communications facilities such as the Clipboard and DDE.

Consistent Behavior

As stated earlier, one of the biggest advantages of the Workplace Shell is that all objects behave in a consistent manner. Every object behaves in generally the same manner given a set of inputs. Of course, objects can be created and modified by programmers, but the extent of these modifications should be no more than to perform a function specific to that object.

Lower Learning Curve

Consistent behavior means that once users learn how a basic object behaves, they know how all objects behave. Once they learn how to drag one folder to another place, they know how to drag any object to any other object. Once users learn the paradigm of drag-and-drop, they know how to start programs, open objects, and configure the system.

Consistency in the entire operating system means shorter learning times and more productive people. Not only that, but the graphical interface actually becomes fun to use.

Object-Oriented

The Workplace Shell is an *object-oriented* interface. Everything is represented by an object. There are printer objects, data file objects, program

objects, a font palette object, and even a shredder object. All actions are performed on objects.

Each object represents something real in your computer. A data file object represents a data file, and a program object represents a program. Users need only learn the basic behaviors of objects and then manipulate them to do their jobs. For example, a printer object is represented by an icon of a printer and a folder is represented by an icon of a folder. Once users understand how the basic objects behave and how to manipulate them, they know how to use any of the applications.

Object technology in OS/2 has not stood still, either. OS/2 Warp includes System Object Model (SOM) release 2 which is CORBA-compliant and forms the foundation of object technology in many other (not only IBM) products.

The SOM object engine underneath the Workplace Shell provides the extensibility for not only programmers, but also users to create new instances of classes and cause behaviors to be inherited through a class hierarchy.

Works the Way People Work

The object-oriented paradigm is fairly awkward to explain on paper. The best way to describe why object-oriented systems are needed for productivity is that they work the way people work. People open folders to look at and use things in them. People open documents to create, read, modify, and print them. An object-oriented interface mimics the way people use everyday objects, representing them on a computer where the objects are stored in a convenient fashion.

For example, to print a document, a user can open a folder, drag the document in the folder to the printer, and drop it there. To change the color of an object such as a folder, users need only open the color palette, drag a color to the folder, and drop it there.

No longer do users need to start an application, feed it a data file, ask it to manipulate the data, and ask it to spit the results back out into a file. That process requires a lot of knowledge on the users' part, such as where the document resides on the disk, how to open a file within the application, and where to put the file being saved, among other things.

Using the object-oriented Workplace Shell, users now need only open a document. The shell will invoke the application and feed it the data file, thus presenting the document in the way the user is accustomed to looking at it—in the application's window. The shell handles knowing where the file actually resides on the disk and how to present it.

Of course, this depends on you, the application designer and programmer. The function is there for you to use and is straightforward to implement. Using it will make your applications much easier to use. We will discuss ways to implement these functions in Chapter 7.

Contextual Help

Another feature of OS/2 that is of great benefit to users is contextual help. OS/2 has the Information Presentation Facility (IPF), which is a fancy name for a subsystem that allows you to code help panels and even online books for OS/2.

Help is important in any computer system. OS/2 provides help for virtually any action through several mechanisms. The easiest way to find help on any item is to press the F1 key. OS/2 also includes several online help *documents*, such as the Master Help Index. All of these objects have a consistent interface, because they all use the IPF to present panels and provide links between topics.

Learning Is Easy

With the online books such as the Master Help Index and Command Reference at their fingertips, along with the contextual help obtainable at any time with the F1 key, users can learn how to use applications and the operating system with ease.

These facilities for learning are provided for you, the application designer and programmer, to enable fast, easy learning for your users. With just a few simple function calls, you can display help panels on exactly the topic you want. As a matter of fact, you can even choose to create an application consisting solely of online books. All this is possible using the tools provided by OS/2.

DEVICE INDEPENDENCE

OS/2 has enormous graphics capabilities. Anyone designing applications should take a lesson from the OS/2 graphical subsystem, because it provides a modular design, isolating the application from the underlying hardware. What this means to applications is that they don't have to know about the underlying hardware to take full advantage of the graphical capabilities of the display or printer. What this means to end users is that when they upgrade their computers, they only need to tell OS/2 what the new hardware is. Under DOS, applications had to be reconfigured or reinstalled; sometimes users had to go so far as to purchase upgraded versions of their favorite applications to take advantage of the new hardware. With OS/2, applications don't have to be changed at all. As hardware technology advances, or even as users just upgrade their systems, applications stay the same.

OS/2 Warp contains more drivers for more devices than ever before. What this means to you is that your programs are accessible to more users on more of the latest hardware without any more work on your part.

Applications See Generic Output Space

As you will see later, applications write to a device-independent Presentation Space. The OS/2 graphics subsystem takes care of all device translation. Under the Presentation Space is the Device Context. This is really where the mapping to the physical device takes place. Users don't have to touch the applications.

Device Contexts exist for any Presentation Manager output device. The two devices most often changed by users are screens and printers, which share a common presentation driver architecture.

New Devices Only Need New Drivers, Not New Applications

This type of architecture means that users need only to install new display or printer drivers to take advantage of new hardware. Under DOS,

there was no standardization of access to devices such as displays, so every application had its own device drivers and its own unique way of communicating with them. Therefore, applications were dependent on their own device drivers to take advantage of hardware, and they had to keep up with all new hardware.

Under OS/2, users have only one driver for the screen, for example. Using the consistent function call interface, all applications access it the same way. New hardware requires a new driver, and all applications immediately take advantage of it with no changes to them on the part of the user.

IBM COMPATIBILITY TO FUTURE RELEASES

The other prominent advantage to coding for OS/2 is IBM's consistent direction when it comes to the desktop platform, and IBM's commitment to compatibility. Any program that runs on OS/2 will run on any subsequent release of OS/2, unchanged, which guarantees that what you write will be viable for as long as you want it to be. This means that users will not have to purchase new versions of applications just because a new version of OS/2 is available. Of course, as you add new features and functions, they will purchase new versions of your applications because they want to, not because they have to. This makes for more satisfied users.

Symmetric Multiprocessing

Symmetric Multiprocessing (SMP) machines have come onto the scene sporting two or more independent, but tightly coupled processors. While many systems have dedicated processors to handle, say, video operations, SMP machines have several *main* processors that handle the general-purpose computing. OS/2 for SMP is written to provide the scalability and performance gains that adding these processors provides.

No specific action needs to be taken to ensure portability to OS/2 SMP systems, but to realize the full benefits of SMP scalability, you need to carefully follow the discussions on threads and scheduling. There is more one can get away with on uniprocessor systems that can cause significant problems on SMP systems. The discussions on threading will help you avoid these problems while realizing the full benefits of SMP.

Portability to PowerPC

The next generation of RISC processor chip, the PowerPC, is a vastly different architecture from the Intel chips. Object code compiled for Intel chips will not run on most PowerPC RISC chips. Additionally, OS/2 for the PowerPC has a different architecture from that of its current Intel counterpart. You might start to think that you have to rewrite for OS/2 for the PowerPC if you want your applications to run. Not so.

IBM is working to make application porting as easy as possible. The OS/2 API set is structured so that the underlying implementation is of very little concern to application developers. Of course there are those such as device driver writers who will need to do some more work to port to PowerPC, but the vast majority of OS/2 application vendors will have to do little more than run the same 32-bit source code through the cross-compiler to generate 32-bit OS/2 for PowerPC programs. Herein lies the beauty of good design. By designing OS/2 with a set of standard interfaces, IBM has been able to insulate you from underlying changes in the operating system.

On the other side, with your good design and modular, high-level source code, you need only to recompile your code for the new target processor. This provides your users with their favorite applications on whatever platform they choose, because of the consistency of the systems. In contrast to others, with sets and subsets of APIs, this gives your users the consistency without you having to rewrite code to exploit the set or subset of function depending on which version of the operating system you may be on.

SUMMARY

OS/2 provides many advantages for end users. Better performance, consistency, the ability to run multiple applications, ease of use and learning, device independence, and maintainability are only some of the benefits OS/2 offers users.

All this translates into better productivity: Users can do more things at once; users do not have to reconfigure applications every time they upgrade their systems; users do not have to purchase or install new revisions of software just to upgrade operating system software; and the system itself works more efficiently.

By choosing to write your applications for OS/2, you are choosing a platform that will allow your users and your applications to do more than ever before. Users will want the power of OS/2. If you have the applications that use that power, yours will be the applications they'll buy.

CHAPTER 2

Why Program for OS/2?

As you have just seen, OS/2 has many advantages for users. OS/2 also offers many advantages to programmers and designers. OS/2 was designed to provide productivity gains not only to users, but to programmers as well.

OS/2 is designed to make it easy to access system services and manipulate resources. All of the constructs you will use lend themselves well to portable, modularized code. Well-designed OS/2 code can be expanded and maintained with ease and will run on all future versions of OS/2.

POWERFUL AND FLEXIBLE API

API stands for Application Programming Interface. This is the way applications request all system services under OS/2. Under DOS, applications would load some of the registers in the processor and then call an

interrupt, such as `INT 21`. The interrupt service routine would read the registers and perform the requested operation. Each interrupt routine had a specific purpose that could involve several subfunctions selected by loading different values into some of the registers.

As time went on, and high-level language compilers were developed for DOS, more and more of the system services could be accessed by simple high-level function calls. For example, a C compiler has a function called `read()` that could read from a file or a port. When the C code was compiled, it would translate down to the interrupt-level interface, where the actual function was implemented. Programs became more portable, as long as the same particular language compiler was available on many platforms. This made programming easier but, in many cases, less flexible. Designers and programmers were restricted to whatever the language compiler designers wished to put in as far as how flexible the `read()` was.

In general, OS/2 language compilers encapsulate the OS/2 API. For example, a C language call to `read()` translates down to a call to DosRead. Sometimes you are given more power through these high-level language functions, but sometimes not. You should use a mixture of the OS/2 API calls and the high-level language functions to suit your purposes. OS/2 API functions are readable but less portable to other environments such as UNIX. However, since OS/2 provides functions that almost no other systems do (such as threading), you can choose to code the functions unique to the OS/2 API directly and code the portable pieces with your chosen language's runtime functions. Of course, you could always write the interrupt-level code, too, but this is cumbersome and difficult to maintain.

One thing I have always liked is readable code. Coding `INT 15` and `INT 21` is not all that readable. In that type of code you must rely on comments to understand quickly what the code does. This can lead to inconsistency, because not everyone comments their code, and some people comment differently than others.

As you can see, there are many drawbacks and inconsistencies in coding for DOS. Many tools have been created to isolate the low-level constructs from programmers, but again, these are add-ons that them-

selves lead to inconsistencies due to the wide variety of interfaces and functions. By contrast, OS/2's API is consistent, powerful, flexible, expandable, and readable.

Function Call Interface to All System Services

As discussed earlier, the DOS programming interface with its interrupts and subfunctions is both clunky and difficult to read. Language compilers such as C have been written to provide a function call interface whereby a programmer can call `malloc()` to allocate memory and `read()` to read a file.

OS/2 has taken this high-level concept and applied it to the basic operating system programming interface. Where DOS has interrupts, OS/2 has functions. All system services must be requested through OS/2, as discussed in the previous chapter. These system services are accessed through function calls in much the same way as the C language implements functions.

When an OS/2 application needs to read a file, it will call `DosRead`. When a window needs to be created, `WinCreateWindow` is called. The first advantage of this is obvious: readability. It is much easier to pick up some code and see what it is doing if it calls `GpiPlayMetafile` or if it calls `INT18`. (Note: `INT18` *not* a "playmetafile" call in DOS.)

This kind of convention increases programmer productivity by making code easier to write, allowing programmers to leave their reference manuals on the shelf longer because they don't have to look up each function they need.

It also increases programmer productivity by making code easier to maintain and enhance, especially if the person working on the code is not the one who originally wrote it. Learning code written by someone else is more difficult than writing the code from scratch.

Consistent, Easy Coding

The function call interface makes coding easy, not only in terms of readability, but also in terms of being able to "know" what function to use.

Once you get used to the naming conventions, it is highly likely that you will be able to know what function to use even if you have never used it before.

Programming OS/2 applications is not trivial. To the uninitiated, it can be overwhelming. However, once you get used to the naming conventions, such as `WinQueryXXX` or `DosQXXX` (to query windows or kernel functions respectively), or `WinDestroyXXX` to remove allocated window management structures, you will be able to figure out quickly what functions you need to perform the tasks you wish.

There are hundreds of function calls in the OS/2 API, but consistency is the key that keeps it manageable. Even through the releases, IBM has kept a consistent interface and set of names for the functions, making coding that much easier.

Portability, Flexibility, Expandability

Another important aspect of the OS/2 API is the expandability built into it. Because of the ever-increasing function of OS/2 from release to release, new functions have been added. Some APIs, however, have simply had their functions enhanced. In some cases, new parameters need to be added.

Now you will agree that by adding new parameters to a function, you destroy its compatibility with existing code. After all, if you pass five parameters to a function that now requires six, your code will no longer work. I agree. OS/2 has designed a way around this from the beginning.

Many APIs in OS/2 may look annoying at first glance. I thought so at first, too. I mean, why pass as parameters a pointer to a buffer, the length of the buffer, and an indicator of what *info level* is being queried or set?

After a short time (about five minutes) I understood. By using this structure for parameter passing, OS/2 has built virtually limitless expandability into each single API. For example, querying file information is easy with `DosQFileInfo`. With the pointer to a `FileInfoBuf` and the info-level parameter, one can get any set or subset of information about a file. When new functions are added, such as the High-Performance File

System added in version 1.2, new information is easy to build into the function: OS/2 simply added a new info level and a new `FileInfoBuf`. Inside the function `DosQFileInfo`, the info level is examined to tell the function what the structure of the `FileInfoBuf` is like. By adding the new info level in a new OS/2 version, OS/2 has expanded the function without having to change the format of the API, ensuring upward compatibility.

Of course, this does not ensure backward compatibility. For example, if info level 6 was added to an API in OS/2 version 2.0, an application calling the API with info level 6 would not work under version 1.3. This is a pitfall of upward compatibility, but there are ways to take this into account, as we will discuss later.

Along with the flexible and expandable functions, OS/2's function call interface is portable and not hardware-dependent. As previously mentioned with OS/2 for PowerPC, and OS/2 for SMP, and with the OS/2 API, porting to other hardware platforms is little more than a recompile.

You'll notice, from the few APIs mentioned here as well as all the other OS/2 APIs, that application source code is not at all hardware-dependent. Of course, some parts of OS/2, and some applications such as device drivers are by nature hardware-dependent, but the majority of application code is not hardware-specific. This means that most of the code you will be writing would need only be recompiled to run on another platform.

The OS/2 function call interface is one of the easiest in the industry to code for. Many developers I have spoken with echo this sentiment. There is a single base API that is of course always being expanded, but subsets of one working on one system and other possibly disjoint subsets targeted for another version of OS/2 as is seen on some other systems, is not in IBM's vocabulary.

Easy to Modularize and Maintain

Along with readability, the function call interface of OS/2 lends itself well to modularized code. With proper design, code written for OS/2 can be subdivided and modularized to make maintenance almost trivial.

The function call interface works along the C language paradigm of functions and so works well in modularized designs. As opposed to a system like DOS, where system services need to be called via an interrupt mechanism, OS/2 code is easily modularized. The interrupt mechanism goes hand-in-hand with an assembly language way of programming that is more or less top-down, not modularized. Extra work is required to modularize assembler code in the way you think of modularized C code.

In addition to the function call mechanism, Dynamic Link Libraries (DLLs) also lend themselves to modularized code. By isolating common functions to save memory and placing them in DLLs, you are also modularizing your code. As a matter of fact, most of the OS/2 APIs are implemented in DLLs.

DLLs are themselves just libraries of functions. If you code each function to be used by potentially many different routines in your applications, your code becomes well tested and efficient, as well as memory-saving. If a function has to be modified, you need only modify that one function in that one DLL and replace that one DLL on the machine.

In Chapter 7, you will see how this concept of modularity makes OS/2 for the PowerPC portable and expandable, taking the concept of DLLs further, using the Workplace architecture to make it not only maintainable, but also scalable so that users need to purchase only those parts of the system they want.

System Coding Conventions

The OS/2 system coding conventions are yet another way to make you more productive. This goes beyond the function call interface.

OS/2 has implemented a set of coding conventions used throughout the system. These are evident in the OS/2 toolkit header files, as well as the OS/2 programming reference documentation and the online reference files. In fact, OS/2 itself is developed using these coding conventions. You can see these conventions, and how they can help you, in the OS/2 programming toolkit sample programs.

These conventions are not limited to OS/2; they can be used anywhere, on any system, by anyone. However, OS/2 code is built around these conventions, and by adhering to them you will quickly become

more productive. Again, you do not have to follow them, but by doing it you are helping yourself immensely.

Without getting into too much detail here, the conventions are simply a way of naming variables and identifying structures in a consistent way so that you can look at the name `ulcbNameList` and know that it is an unsigned long variable, representing the number of bytes in a name list (`ul` for unsigned long, `cb` for count of bytes, and `NameList` is self-explanatory). This convention is used throughout the OS/2 system and will make your code more consistent, readable, and maintainable than ever before.

Fast Prototyping

Another important result of all these productivity gains is in prototyping interfaces faster. By using the functions provided along with the other advantages mentioned here, you can develop prototypes of your code much faster than ever before.

Once you get a couple of functions working, you can build on them very quickly. With the OS/2 programming model, once you get the basic skeleton of a program done, you can quickly add functions and try out new things. This leads to fast prototyping for evaluation—even after your design is done and you simply want to compare various algorithms for a particular task.

As OS/2's momentum has picked up, many tool vendors have written software development packages that do everything from paint dialogs to generate source code to generating an entire application using objects and parts.

OS/2 Development: The Next Generation

The future of software development lies with objects, reuse, and program development environments. The environments for OS/2 can assist you in prototyping, coding, and even testing. As with most development environments, code generation tools, and such, you have a trade-off between speed of development and fine control of the code for performance and working set tuning.

In Section II, you will see what these environments and tools can give you and what you give up when you use them. Some will give you speed of development while increasing application working set size through runtime libraries. Others can generate or even port source code from other platforms for you. There are many such tools now available and for whatever reason, they seem to be appearing on OS/2 first, before many other systems.

With the advances of SOM, its CORBA compliance and acceptance by CIL and vendors, along with the advances and acceptance of OpenDoc and its OS/2 implementation, OS/2 is more on the technological forefront than ever before.

The real key to all of this is IBM's plan to keep not only the user, but also the application developer insensitive to the underlying impementation, whether it be the IBM Microkernel at the core, or the Intel-based architecture, or an SMP machine. By using the consistency of the API and the fundamentals you will see here, you can write code that runs on just about anything running OS/2, with a minimum of work.

SUMMARY

All this leads to the fact that OS/2 is a highly desirable platform for which to write programs. OS/2 was designed not only as a system for end users, but programmers were also in mind when OS/2 was designed and are still in mind when the system is enhanced.

The OS/2 Application Programming Interface is neither trivial nor simple. However, the OS/2 development team takes pains to make functions easy for end users to perform while keeping them flexible and powerful for application developers. The high-level language function call interface, along with functions that are expandable and flexible and a consistent programming interface, all lead to a system that is powerful, flexible, and expandable.

With the new tools that generate all or parts of applications for you, OS/2 development has become even easier and more convenient.

CHAPTER 3

OS/2 as a Development Platform

As you have seen, OS/2 provides flexibility and expandability to users and will run virtually all applications written for the PC. You have also seen that OS/2 is one of the best systems in the industry for which to write, with its flexible, expandable function call API as well as modularized system services second to none.

But, not only is OS/2 an ideal environment for users and an excellent system to code for, it is also an excellent development platform—that is, the system you will write, test, and debug on.

First, OS/2 offers developers the same productivity it offers other users in being able to run many productivity applications at once. E-Mail, editors, LAN, and utilities can all be running simultaneously. Not only can you run the everyday productivity applications, but all of your development tools can be run simultaneously as well.

MULTITASKING FOR DEVELOPMENT

OS/2's multitasking extends beyond spreadsheet recalculations or database sorts and searches. Since OS/2 runs all of your programs at the same time, you can be editing one file while another is being compiled in the background, while you run some GREPs in yet another window.

OS/2 increases programmer productivity immensely. In the days before OS/2, I remember having two or three computers to get work done. Of course, there was one to do the editing and programming work. Another was for debugging and testing. The third was just there to run compiles. Each machine consumed desk space, power, and most importantly, the cost of each machine. Multiply this by the number of developers, and you get ridiculous overhead on your projects.

With OS/2 you can develop, compile, and do all your daily work on the same machine. This is cost- and space-effective—not to mention the logistics of having to configure environments on each machine and having to maintain them. This translates into savings across the entire project in terms of time, work, money, and even space.

DEBUGGER SUPPORT

Multitasking extends even beyond the obvious advantages of running compiles in the background while you do other work. Application debugging takes on a whole new look under OS/2.

One of the problems of debugging programs on DOS-based machines was the fact that when an application crashed or hung, you usually had to reboot the machine. Sometimes the crash was so bad that the application ate part of DOS and you actually had to power off the computer.

OS/2 relieves the problem of having to reboot your machine during debug sessions. Since these programs also run in separate sessions from

any others, such as your editors or compilers, you can debug with impunity and not have to worry about trashing your system. At the worst, you'll hang the debug session and have to restart it.

There are some exceptions to this rule, however. OS/2 applications generally run at ring 3 or user mode, the least privileged level. As such, nothing running at this level can touch anything else, so the only thing it can mess with is its own session. However, developers may need to write device drivers to support specific hardware, such as scanners. OS/2 device drivers run at ring 0 on Intel, and, as such, are just as privileged as the OS/2 kernel. These programs can inadvertently step on any part of the system and possibly force a reboot. This is a pitfall of developing device drivers.

However, in general, most developers are writing user applications and don't suffer the problems associated with writing ring 0 code. Writing user level code is easy and convenient to debug. In fact, you could go so far as to write several variations in your code to test out function, fixes, or performance and debug them side by side in different sessions. OS/2 allows this because of the protection and virtualization in the system.

Not only does OS/2 provide applications the ability for smooth debugging of applications, but there is also a set of APIs specifically geared to support debuggers. Debugger authors have the tools to start programs "underneath" them so that they can manage execution, modify registers, and even restart the program being debugged upon command.

These functions provide you with complex application development support while allowing the system to continue running even if the application hangs its session. Debuggers become an integral part of the operating system with the debugging APIs as well. All this helps you efficiently debug your code.

Recent functional enhancements in OS/2 include not only system dump support, which has been there all along, but now also process dump support for those instances where you may encounter a timing problem and real-time debugging can mask the problem. You can tell the system when a certain error occurs to take a process dump for later analysis.

CRASH PROTECTION

The other prominent feature of OS/2 that appeals to developers was just mentioned in the preceding paragraphs: crash protection. OS/2 has this unique feature—an advantage over any other PC operating environment.

Of course, this has the advantage for users in that an application that has bugs undiscovered during test will not hang the user's system. This protection extends to the developer's desk to prevent system crashes during program development and testing. This translates into fewer reboots and increased productivity.

SUMMARY

Just because OS/2 provides a wealth of features to end users does not mean that developers are excluded. OS/2 affords everyone equal protection.

Protection, along with multitasking and debugger APIs, makes OS/2 an ideal development platform. And, you are not restricted to developing OS/2 applications on OS/2; OS/2 runs all of your Windows and DOS applications. This includes being able to debug these applications under development.

Of course, OS/2 applications can take advantage of all of the features of OS/2, thus making them more powerful than the other classes of applications although OS/2 will support them all. In fact, I have heard of many developers who are using OS/2 to develop their DOS and Windows as well as their OS/2 versions.

OS/2 will do it for you as well as your users.

SECTION II

Overall Application Design

One thing that I stated previously, and will continue to state throughout this book, is that most applications are not designed well. Please don't get offended at this comment—there are many factors that hold all of us back from designing our applications well.

One of the biggest inhibitors of good, thorough design is something all programmers hate to hear: schedules. Often good design plans get tossed aside for the sake of schedules. Companies are in business to make money, and timing is quite often the most critical factor in developing a product. The one thing all project managers must understand is that although they can say that the code must be done on a date that leaves little time for design, that lack of design will come back to haunt them later. A favorite saying of mine is, "There's never enough time to do it right the first time, but there's always time to do it over."

Sadly enough, this is true of many software projects. People will forsake quality in design for expediency. This never works.

Another item that looms large in the excuse for poor or lack of design is its overhead. It is limiting. It hinders a programmer's creative freedom. It locks us down and binds us. Nothing is further from the truth.

Contrary to these common claims and misconceptions, good design allows a great deal of freedom. With a good design, you have time to do all those things you really want to do. Because you have set up your objectives and how you will divide and conquer the tasks, you have the freedom to decide how you want to accomplish those tasks.

Good design frees you from figuring things out as you go along. You will have your tasks set out before you and will have the interfaces between tasks defined. This allows you the freedom to optimize these interfaces before you get to the point of having to do so out of necessity.

Good design does not lock you down, either. Obviously, you have to have a set of objectives fairly well set when you design your applications, but those objectives can change. With good design, additional functions and modified objectives are trivial. One of the main design points of software is that anything you do must remain flexible. Functionality should be able to be added and modified with little impact on the rest of the system.

Notice that all these points are real design. This is not modularized code design. This is not meaningful variable names. This is application-level and functional design.

No task is too large if you tackle it properly. Football players can tackle others much larger than themselves with a good overall plan. Trying to overpower something too large is futile. However, like a football player wanting to tackle a much larger player, knowing that you want to grab this problem by the ankles is the first place to start. How you will get to the ankles, where and how you will take hold, and how you plan to hold on are just details. The point is that you've developed an overall strategy.

You have assimilated the facts about the environment. You understand the objectives. You understand the tools available to you and your limitations. You begin to develop a strategy about what functions you will exploit and how they will help you accomplish the functions you

want to provide in your application. Like the smaller football player, you see which cleats to wear, if the field is muddy, and how best to exploit your physical abilities—your tools.

This section will cover overall design and strategy. The advantages and philosophies of good design will be discussed, along with the environment and the functions provided for you to use.

The design of an application requires that you not only understand the concepts of application design, but also know the environment. OS/2 is not well understood yet, simply due to the relatively short time it has been around compared to some other systems. I will show you OS/2's architecture to give you a better understanding of how it works and how to make your application part of it.

We will also discuss the features OS/2 provides applications, which features work better in certain situations, and how you can choose which fits best for you. OS/2 provides several ways to accomplish functions—some easier to code, some more difficult, some more fundamental, some more powerful—each with its own set of advantages and disadvantages. We will discuss how to choose the best ones for you.

In later sections, we will delve deeper into the design process and work on the function-level design. Just like the football player, we'll work on the game strategy, learn the field and stadium conditions, and then work on the play selection.

CHAPTER 4

Good Programs Have Good Up-Front Design

If you couldn't tell from the introduction to this section, I am a big proponent of doing things right the first time rather than fixing or doing them over later on.

By getting your applications designed before coding begins, you run a much better chance of catching any mistakes or omissions before they become more expensive to fix. Once coding starts, any changes to the design become more expensive, both in terms of having to recode and in terms of subsequently introducing more design holes and bugs.

If you maintain the patience and discipline to write your application in design first rather than code, you'll reap the rewards in the long run. Your application will be more flexible, maintainable, and, more than likely, easier to fix should there be problems in the code.

UNDERSTANDING THE TARGET ENVIRONMENT

The first part of designing an application is to understand the environment. This is even more important than what the application itself will do. You must know the environment before you think about what your application will do.

Different environments are better suited to different types of programs, user interfaces, file mechanisms, and features provided to the users. By understanding the environment, you make the best choices for the application, given the parameters it has to work with.

You must understand the environment for beginning any application design. The worst thing you will see is someone spending time designing an interface or function and finding out that the environment supports it elegantly and easily, negating the work just done and forcing other interfaces to be reworked. Understanding the environment is important for any applications design, not just under OS/2.

60 PERCENT DESIGN, 30 PERCENT CODE, 10 PERCENT TEST

60, 30, 10 is the formula for software project success. All too many software projects use this formula, but in reverse—10, 30, 60. Quite simply, if software is done right, most of your time will be spent designing. You'll then code it and spend a little time at the end testing.

Some people balk at this and ask how anyone can put out a piece of software in good conscience with only 10 percent of the time on the project spent testing. The answer is quite simple.

First, 60, 30, 10 are not hard and fast numbers. I did not derive them from any scientific formula nor from empirical evidence. They are there

only to demonstrate a point. That point is: By spending the bulk of your time designing, using coding to implement the design, and testing to test the code's implementation, you will have a much more productive project than if you just start slinging code and wring out problems as you test.

By following the 60-30-10 formula, you have already gone through every possible scenario that a test program could, and you've nipped the potential problems even before the code was written. By catching these things up front rather than later, you run less risk of introducing more problems just by fixing one problem after the code is written.

No matter how good you are, if you have to fix problems after you've coded, you are likely to introduce more problems than you are fixing. The reason for this too is simple. When you are fixing the problems in the code, you see that piece of code alone, and without looking at the entire picture while you are doing the fix (which is unlikely), you are likely to introduce some other problems. This takes the 10-percent testing to, say, 50 percent.

You may notice that this turns your project into 140 percent. That's why most projects neglect the important items up front and wind up doing them over later. Remember, "There's never time to do it right, but there's always time to do it over."

To avoid spending 140 percent of your time on a project, you must have the discipline to design the entire project before you start coding. Changes later on can be handled easily if the design is done properly, and they do not have to affect the rest of the project.

Many times we start thrashing, getting nothing done because requirements and new function requests keep rolling in with no end in sight. As soon as you think you are close, more things are heaped on top. This does not have to be.

60 Percent Design

Stick with the 60-30-10 formula. By spending that 60 percent of the time first, you make later changes easy. The best way to do this is to design in your original requirements. Keep in mind the ones gathering up in the

background. Once the original design is done, you can work on adding in the new requirements.

Don't get me wrong. This does not mean to schedule 60 percent of the allocated time for a project and working on the initial requirements just to say you are spending 60 percent of the time on design. Be sure to leave time for those late-breaking functions and requirements we all know will be there.

Another important fact to remember is not to take all requirements and add them just because they are there. At some point, you will need to make a cut and say, "Okay, that's it. We need to get this project done. We can address the rest in the next release."

If you don't do this, you'll be forever adding functions to the design until you run out of money, because you won't have a product to sell. It will still be on the drawing board.

I will reiterate: This takes discipline and the cooperation of your sales and management force. It is up to you, as the project designer, to keep everything in perspective to make this release and all future releases as easy to produce as possible.

30 Percent Code

This sounds pretty self-explanatory, but there is more to it than just reading the design and turning it into code. There are many things that can happen during the coding phase of the project, and again, discipline is the key in making the best product possible.

The most common problem encountered in the coding phase of an application is the point when you sit back and look at what you are doing and realize you are heading down the wrong path. This can be the realization of design errors or coding errors. In any case, you again have a hard decision to make.

The decision is whether to continue, knowing that you are saving time in the short term but are only exacerbating the problem in the process, or to bite the bullet and fix the root cause of the problem.

It's fairly obvious which is the right thing to do. Of course, business decisions may dictate otherwise, but the best choice is to fix the errors

in design or rewrite the code in question, rather than building on an admittedly problematic base.

I can only sum it up this way: You've spent so much time working to wring out the design and do it in the smartest way possible. You understand what it will be like trying to enhance and maintain this code later on. Also, who knows how much time you will waste in the testing phase of the project if you are having these problems now? Why throw it all away if you can catch the errors early?

By catching the problems early and really doing something about them as soon as you find them, you save time and money not only for this project, but also for anything built on top of this project.

10 Percent Test

Testing is usually the most focused part of a project. Actually, the test phase of a project should not consist of much more than a *design test*, a *unit function test*, and an *overall system test*.

The main part of the test really tests the design. If there are functions missing or some do not work, the design needs to be examined. If the application was designed properly, the only things you should need to look for are mistakes in the code. With proper design, all holes should be filled before coding starts.

All too often I see applications that did not take this into account or did not expect users to try to use a function in a slightly different way than it was implemented. With foresight, you can avoid this and not have to worry about it in test. This foresight comes from spending that extra time in design and from not trying to get it into test before it is ready.

The testing process should test code paths for both normal operation and error conditions. Remember, the application is already designed. Any problems caught in test should be simple code mistakes rather than holes in the product. Should any of these holes be found, that subsystem should go back to the design step.

The testing of an application should just test to ensure the code meets the design specifications. If the application was designed in the design

phase and not the code phase, this should be only about 10 percent of the whole project schedule.

SUMMARY

The importance of doing most of your work up front cannot be overstated. This includes defining the requirements for your application and working on them. Any new requirements should be kept in mind, but not added into the design until the core or original design is ready for them.

At some point, you need to stop taking new requirements and start to code. This should be fairly straightforward, because all of your interfaces and functional aspects are designed. You can then let the developers loose to implement these functions as they wish, making sure to stick to the published design specs.

Once the code is about done, you can run through your test suite. The testing process should not have to be too involved, since you've wrung out all the problems and holes in the design so that all you are really testing is the code quality. Keep in mind that if you run into too many errors in a certain component, you may want to reevaluate the design of that component, or possibly just recode it if the design still meets the requirements.

Sticking to the 60-30-10 formula does not guarantee success. However, it will make your project more manageable and your results more maintainable.

CHAPTER 5

OS/2 Kernel Architecture

Understanding an operating environment means understanding how it works, the functions it performs, and the services it provides. OS/2 is a complex system. It has to be to offer all of the functionality it does. At the heart of OS/2 is the kernel.

The OS/2 kernel provides all of the core, or kernel, functions such as protection, the file system, scheduler/dispatcher, base device drivers, and processor management. The term *kernel* refers to all of the base OS/2 systems, such as memory management and the scheduler, along with the device drivers, such as the driver for the disk drive and the driver for the parallel ports. Everything in OS/2 goes through the kernel to get work done.

In a pure sense, device drivers are not part of the kernel but, conceptually they become extensions to the kernel when they are loaded. These are examples of device drivers in the traditional sense: low-level routines that manage the interaction between the system software and the hardware.

The *base subsystems*, such as the video and keyboard subsystems, are not device drivers or part of the kernel, but they will be discussed here because they are part of the low-level code that makes up the core of OS/2.

OVERVIEW OF THE KERNEL

The kernel provides the basic functions needed for operating system function. Included in the kernel are the base file system, the scheduler/dispatcher, and all of the protection features, such as the exception handler and memory management.

Structure

Figure 5.1 shows the basic architecture of OS/2. You will notice that this diagram shows the Control Program, which includes the kernel as well as surrounding subsystems. Although the kernel is a separate physical entity, it encompasses functions of memory management, tasking/scheduling, and exception management, among others. The Control Program is really the core of the operating system built around the kernel.

It all begins, however, with the hardware. The only software that interfaces with the hardware is the kernel and the device drivers. All the rest of the operating system goes through the kernel and device drivers to get CPU time, read and write files, communicate to output ports, and write data to the screen. This restricted hardware access is how OS/2 accomplishes a large part of its protection and coordination of the operating system and the hardware.

The kernel is constructed from a set of core services upon which the other kernel and Control Program services are built. Let's start at the beginning—when the system boots up. Once you understand how the system loads, you will better understand how the parts of OS/2 interact. Remember, though, that this is more of an overview discussion; each item discussed has many parts, and some items are discussed at a

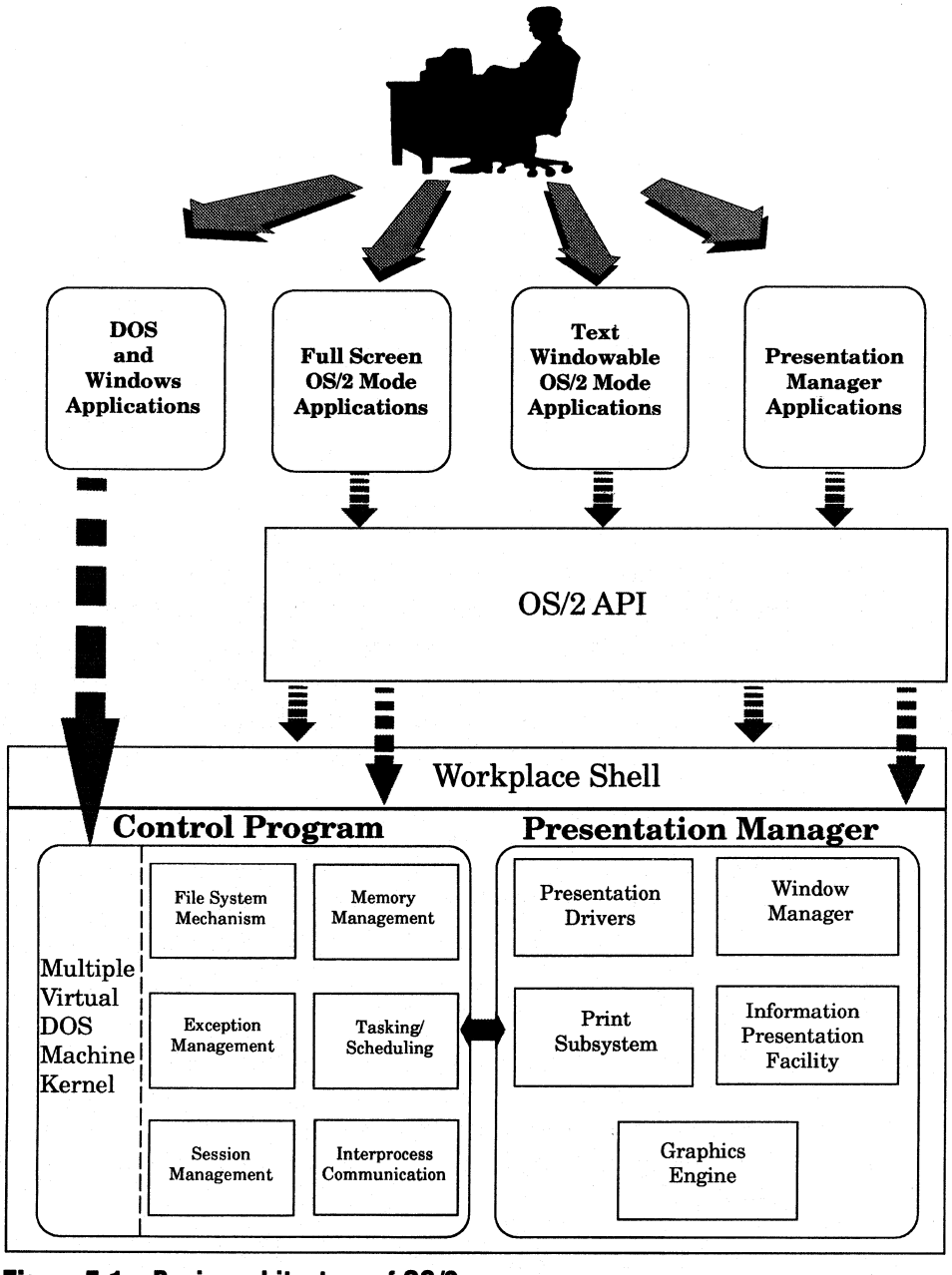


Figure 5.1 Basic architecture of OS/2.

superficial level. This is not an OS/2 architecture book. This discussion is intended to aid you in designing your OS/2 applications by helping you to understand how OS/2 works.

After a PC completes its power-on self-test, it begins reading from the boot sector of the hard disk. The OS/2 boot sector loads a program called `OS2LDR`—the loader that loads the kernel.

Next, `CONFIG.SYS` is read, and the statements in it are processed. For example, one of the first statements processed is the `IFS` to load any file system drivers. Other statements, needed to build control blocks, are also read very early. These statements, such as the `THREADS`, `MAXWAIT`, and `MEMMAN`, instruct the system kernel how to configure itself and allocate control blocks for the work it needs to perform, such as scheduling. File system control structures are also built at this time, as well as all of the tables and pointers to manage system memory.

Once these statements are processed and the systems structures set up, a large part of the kernel is initialized, and the next step is to load the device drivers by processing the `DEVICE` statements. Each OS/2 device driver is specified to the system in a `DEVICE` statement. Each device driver is loaded and called at its initialization routine. Some device drivers will display a message at this point; others will not. Each device driver is loaded in turn and becomes, conceptually if not physically, part of the kernel.

The device drivers are the way the system communicates with peripheral devices, such as displays, disks, and printers. The kernel basically controls the CPU and memory accesses, while the device drivers manage access to the other devices. For example, the file system is part of the kernel, but it goes through the disk device driver to talk to the physical device.

On top of the kernel layer sit the base subsystems, which manage the *virtualization* of all console services. Virtualization is what separates each session's input and output from all others. Virtualization makes each session a separate console with its own virtual keyboard, mouse, and display.

The base subsystems consist of device drivers and sets of common routines, usually contained in DLLs, such as `BVHSVGA.DLL` (for the base

video subsystem). First, the device drivers are loaded; then the DLLs are loaded.

The DLLs that make up the support routines for the base subsystems and the Presentation Manager are loaded last. This code is more like an application that uses the system services of the kernel and base subsystems. These pieces of OS/2 will be discussed in depth in the next chapter.

Scheduler/Dispatcher

The OS/2 scheduler/dispatcher component is responsible for managing all the multitasking of the operating system. Some may think that OS/2 is simply a round-robin time slicer. That may seem true on the surface; as a matter of fact, any single-processor operating system is something of a time slicer. The OS/2 scheduler/dispatcher, though, is much more.

The OS/2 scheduler is about the only component that actually loops. Under this multitasking system, everything needs to be event-driven. If any program loops, it will burn CPU time, regardless of whether it needs the CPU and take computing time away from those programs ready to run that do need it. The scheduler, however, does need to loop.

The scheduler has a fairly straightforward job. It runs a dispatch loop and determines which thread within the system should get dispatched next. (Of course, this, too, is oversimplifying the underlying algorithms.) The scheduler is really the focal point of the kernel, as all threads must be made ready by the scheduler/dispatcher and subsequently dispatched.

In a classical queueing, scheduling system, there is a list of tasks, called the *ready list*. Depending on the type of system, these tasks can be programs or, in the case of OS/2, threads. Each thread has a *slot number*, identifying it distinctly from all other threads in the system.

Each of these threads has a block of information associated with it. This block of information includes the thread's *priority*, which the scheduler uses to determine which task occupies the CPU. Each thread also has a *state*, such as ready or blocked. These different states will be discussed shortly.

In general, the highest-priority thread in the ready list that is also ready to run is the one that is using the CPU. This is the one thread that is listed as running if you were able to look at the ready list.

The scheduler operates on the principle that any thread that becomes ready to run and that has a priority higher than that of the thread currently running will be dispatched as soon as possible. When this happens, the currently running task will be *preempted*—that is, taken out of the CPU and placed back on the ready list. This is called *preemptive multitasking*. Any thread that becomes ready and is of higher priority than the currently running thread will cause the running thread to be preempted. This is how OS/2 maintains the integrity of time-critical applications, such as communications programs.

Part of the scheduler/dispatcher's job is to manage the priorities of threads so that no one gets starved out of the system. For example, if there were high-priority threads that were CPU-bound, they could conceivably run forever, starving out all of the other, lower-priority threads. The scheduler/dispatcher takes care of this as well. As a thread sits on the ready list, its priority is dynamically modified by OS/2, based on how much time has elapsed since that thread had CPU time and whether the thread is part of the foreground process, among others. We will discuss this in more depth shortly.

The scheduler and threads are really the core of OS/2, but there are many other facets to the system and understanding how it works. Threads execute code, which makes up the other OS/2 systems.

Scheduling on SMP

Scheduling threads on symmetric multiprocessor machines is conceptually not much different than uniprocessors. There are many extra things that the operating system must account for, such as if a processor goes down, or if an application or device driver disables interrupts (should interrupts for the system or only that processor be disabled?), among other things. For you, however, not much is different.

On uniprocessor machines, you know that not more than one thread is physically executing at any moment in time. You should not, but many

developers do, make assumptions based on this fact. Most assumptions made are on operations being atomic and some being executed in a specific order. In an SMP environment, multiple instructions may be executing simultaneously, invalidating that assumption.

By writing your code assuming nothing about the order of execution of functions or timing, you can be assured of being "SMP-safe."

The SMP scheduler schedules tasks in a similar fashion to the uniprocessor scheduler, except that it does this *n* times, where *n* is the number of processors in the system. This scheduling is dynamic, with standard priority modification as on a uniprocessor system. Rather than the highest priority thread being the one that is running, however, the *n* highest priority threads that are ready to run will be actively executing in a processor.

The other details of how this works are not all that relevant to application writers, but more to device driver writers. Suffice it to say that the scalability you might see on other SMP-enabled operating systems is there on OS/2 as well. That scalability is based purely on how well threaded your application is. If you have only one thread, users of your application will not experience the same benefits on an SMP machine as if your application has several threads. Don't get the idea of adding lots of threads to run better on SMP machines; rather, you should thread appropriately based on the function and application, knowing that the more efficiently you do it, the better you will perform on SMP as well as uniprocessor boxes. Application threading concepts and structure will be discussed in detail in Sections IV and V.

Loader

The loader is the part of OS/2 that gets everything moving. As you have already seen, it is the first piece of OS/2 code that runs when the system boots. Once the system is up and running, the loader is responsible for initiating each instance of a program. When a user requests that a program be loaded, the loader is invoked and takes charge. The OS/2 program loader is responsible for allocating memory, control blocks, and other structures for a program or process to run. The loader reads

in the executable file header, and sets up these structures for that specific program. Memory is allocated, and code is loaded into the memory areas. The structures are set up for the main thread of the program, and control is transferred to the point of the program indicated by the executable header.

The loader is also involved whenever code needs to be accessed that has not been read into memory yet. When a program is first loaded, only the first part of the code is loaded unless the executable header indicates otherwise. When a subsequent code segment (or page, as is said now in the 32-bit world) needs to be loaded, the program loader takes care of it. The application need not have to know where the code is or which piece needs to be loaded.

As you will see in the discussion of memory management, code is discarded and reread from the executable file on the disk when memory is overcommitted. Only data is paged out to disk. So, as long as the program remains in memory, the loader may be called upon to load or reload pieces of the code.

Once the loader has set up the control, memory, and thread structures, control is handed to the thread, which begins executing instructions. Application threads do most of the work within OS/2. The operating system code simply coordinates execution.

System Services Flow

Very few threads actually belong to the kernel of OS/2. Every executable program (`.EXE`) has at least one thread, but the kernel itself has very few. Most of the OS/2 threads are part of executables, such as the spooler and the shell. Except for the scheduler/dispatcher, the OS/2 kernel really does not "run" *per se*; it uses the application or user threads to accomplish its work.

Let's trace a system call from the application, down through the kernel and device drivers, all the way back up to the application. Refer to Figure 5.1 during this discussion.

Let's assume the user has selected to save a file. This will generate a call to `DosWrite` at some point. Even if the programmer of this application

has used the C runtime function `write()`, the call will ultimately be made to `DosWrite`.

`DosWrite` has several parameters, including the buffer to be written and a handle to the file (obtained previously by a call to `DosOpen`). Inside the program's code, there is a call to a function in a DLL called `DOSCALL1`. This function is the actual code for `DosWrite`. Recall that this system code resides in DLLs to eliminate multiple copies of the code in the system, and to provide maintainability. (Note that beginning with OS/2 Warp, many of these functions have been merged for system performance into a DLL called `PMMERGE.DLL`, but the external references to the functions are still where they were, such as `PMWIN` and `PMGRE.DLL`.)

The address resolution for this call was done at load time by the OS/2 program loader, which sets up the set of addresses to system calls. In the executable file header is a list of all DLLs that are referenced by the code being loaded. When the loader sets up the code at load time, these references are fixed up to the actual addresses where the code in those DLLs lives. This process is known informally as *fixing up*, and the actual code referred to as *fixups*, which will be elaborated on in later discussions of performance.

Looking back at the example function call, you will see that the thread belonging to the process executing the `DosWrite` jumped from executing code from the `EXE` into executing code in the DLL. At some point, there is code in the DLL to make a raw file system call, which is not published. It does not need to be. The interface to write a file in OS/2 is `DosWrite`. The subsystem in `DOSCALL1.DLL` translates the file handle into raw file system information and calls the file system functions down in the kernel. This layer of abstraction relieves the applications of the burden of knowing what the underlying file system is; if any modifications need to be done to the low-level code, applications need not be aware of it.

The raw file system call transfers the thread from executing code in the DLL to code in the kernel—the file system, to be exact. There may be several calls and returns between the DLL and the file system; that is not important. The point here is that the application's thread is still doing all this work.

At some point, the file system will use the services of the kernel and call into the physical disk device driver. The file system knows about sectors, disk directory structures, and so on. The physical disk device driver is used to tell the disk to move the head, write the bytes at this spot on the disk, and so on. Again, the application's thread is doing all this work.

When the request goes to the physical device driver, the thread executes code that tells the disk controller hardware to perform the specified action. When this occurs, the thread blocks. There is nothing for it to do until the hardware signals with a hardware interrupt that it is done. At this time, another thread can be dispatched to utilize the processor efficiently. There is no reason the processor should sit idle waiting for the disk to get done if there are other threads waiting to run.

When the physical device signals that it is done, the waiting thread is put back on the ready list with a somewhat higher priority, since it was in the processor last and gave it up voluntarily. The scheduler/dispatcher will dispatch this thread, which will return up the function call chain down which it came, finally ending up at the application with a return code.

The device driver has some return indicator, which is given to its caller, the file system. The file system interprets the return code and formulates one for its caller, the `DosWrite`. The code in the DLL for `DosWrite` takes this information and formats it in the structure or return code that is returned from that API.

Notice that through all of this, no kernel or OS/2 system thread was involved. You can start to see why you will want to make use of multiple threads to perform tasks that take time, while allowing your users to perform other actions.

All OS/2 system services follow this same flow of execution. This is a very simplified example—many APIs cause threads to be created and destroyed on behalf of the application—but understanding the basic flow is vital. By looking at the flow of this function and multiplying it by the number of threads that can be running at a given moment, you can see how the subsystems and device drivers handle overlapped I/O. In the example just given, assume another thread comes into the device driver

while the first is either still doing work in there or blocked waiting on the device. This other thread will simply execute the same code in another context, block waiting on the device, and will be dispatched when it is next on the ready list. It is a simple flow multiplied many times.

Because of this flow of execution, it is really the application's threads that perform all of the system coordination and multitasking resource synchronization. Since it is the application threads executing the code in the file system and device drivers, for example, any coordination is done by those threads.

In the code for the file system, video subsystem, keyboard subsystem, and all parts of OS/2, there are semaphores and other control structures that will serialize access to the various system resources. For example, if two applications (actually, two threads) try to gain access to a single device, such as an output port, the first thread will grab a semaphore when inside the subsystem code. When the second thread comes along to try to execute the same function, it will try to grab the same semaphore when inside the subsystem code. Since the first thread is not done, the semaphore will not be available, so the second thread will have to wait until the first is done.

This is the premise behind OS/2's multitasking and common subsystems. The threads of the applications will coordinate themselves, not because of the application code, but because all the threads execute a common set of code in the OS/2 subsystems. When you think about it, the subsystems coordinate the execution, but the subsystems don't really run, since they don't have any threads. The application threads coordinate execution, because they are the parts of the system that run.

Digging Deeper

Now that you have an overview of the architecture of OS/2 and understand how the function call mechanism and control flow work, let's dig a bit deeper into the subsystems to see what makes them work. By understanding them, you will have a much better handle on how to design your applications to work with them in the most efficient way possible.

Protection Mechanism

Protection is one of the primary goals of OS/2. In a multitasking system, all programs must be protected from each other, for obvious reasons. OS/2 uses the features of the Intel 80386 and 80486 processors along with protection mechanisms of its own to ensure that no program can interfere with another.

The ring protection mechanism of the processor is the center of this protection. The ring mechanism is fairly simple. As shown in Figure 5.2, the rings are concentric circles. The center is called ring 0, and the outer ring is ring 3. The kernel and device drivers run at ring 0, which is the most privileged code in the system. Code at ring 0 can access any memory or any piece of hardware. Applications generally run at ring 3. Most of the code you will be writing will be ring 3 code, unless you have a need to write a device driver. We will discuss device drivers later on.

Applications running at ring 3 are the least trusted code in the system and must request system services for everything. The operating system code is what runs at ring 0 and controls and coordinates the execution. As you have just seen, these services are the way OS/2 uses the application's threads to manage system resources.

As applications request system services, transitions are made between the protection rings depending on the code being accessed. Each

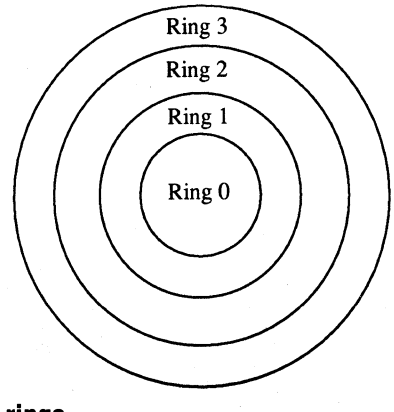


Figure 5.2 Protection rings.

ring has its own set of functions it is permitted, and the code running at each ring has a set of functions it is responsible for. Ring transitions are coded into the subsystems and will be discussed in more detail when we discuss memory management.

Protection is accomplished through a set of routines that are exercised when memory, or any hardware for that matter, is accessed. These algorithms, too, will be discussed when we cover memory management, since that is where they are used the most.

Since threads are the structures that execute code, it is the threads that access memory or other hardware, and protection occurs at the thread level on every hardware access. Although this protection is done on the thread level, any termination due to protection violations occurs at the process level.

Let's now explore the relationships between processes and threads, and the various interactions between threads of the same process and between different processes.

Process/Thread Model

The process is how OS/2 manages all the system resources. Recall the discussion of how the system boots up. Basically, that is how every program or process in OS/2 is born. When a user requests a program to be run, the loader brings that program into memory and sets it up to be run. As has already been stated, that is a very simple description of a very involved process.

When the loader sets up a program to run, it creates one thread to be the main thread of the application. This thread executes code and, in that code, manipulates system resources. One of the most powerful features of OS/2 is that a thread can create other threads or even processes and sessions.

Other threads, processes, and sessions are created through system APIs, such as `DosCreateThread` and `DosExecPgm`. Recall that the operating system has very few threads of its own and uses application threads to do much of the work.

In our discussion of threads and processes so far, the definitions have been fairly obvious. Let's take a moment to refine and define the role of the process and thread in the scheme of things.

The *process* is how OS/2 manages all system resources. The process is the unit of ownership, whereas the thread is the unit of execution. A process does not run; a process owns things, such as memory, files, semaphores, and other system resources. A process also owns threads.

The *thread* is the unit of execution in OS/2. Each thread has its own context in which it runs. A thread context is a set of registers and what is called an *execution instance*. The context is a complete environment in which the thread owning the context can run. A thread switch by the scheduler is really just a context switch.

Along with the registers and other information specific to that thread, the context indicates a *thread state*. These states include *running*, *blocked*, and *frozen*. This is the part of the thread context that is always accessed by the scheduler/dispatcher to maintain the ready list and determine which thread is the next to run.

Since the process is the owning entity, everything in that process has the ability to access anything owned by it. This means that all threads within a process can access all of that process's memory, files, and control structures. OS/2 protects applications or processes from interfering with each other. However, all threads within a single process can access anything within that process. This leads to some interesting dilemmas and design considerations. We will get into these topics in depth in Chapter 12.

OS/2's protection model revolves around the process. Recall, however, that a process does not run—it owns, whereas a thread runs. When a thread is about to issue an instruction that would cause an access violation, the thread and its owning process are terminated. When the process is terminated, everything the process owns is released, and all threads belonging to that process are terminated.

It is vital that all threads of a process be terminated, since each thread has access to all of the process's data. If one thread causes a violation, that means something the process owns has gone wrong, whether it be a pointer or attempted access to the hardware. The reason for ter-

minating the entire process, and all of its threads along with it, is that if one thread of the process has gone wrong, then all integrity of the process is lost. You can't count on anything being correct anymore.

If the other threads of the process tried to use a return code from the errant thread or to use some common data structure, data could be corrupted or lost. By terminating the entire process immediately, the system preserves the integrity of the data on the disk associated with that process, along with the integrity of all other processes in the system.

Priority

One of the most important features of OS/2's multitasking and its management of multiple tasks is priority of threads. Every thread in OS/2 has a priority. The priority of a thread is a number relative to the priority number of the other threads in the system. This is how the scheduler knows what threads to schedule next or whether any thread preemption needs to be done.

There are four priority classes in OS/2, each having 31 sublevels. The classes, in order of highest priority to lowest, are Time Critical; Fixed High, which is also sometimes called the Server class; the Regular class; and the Idle Time class.

There is also a line in the `CONFIG.SYS` indicating whether priority is to be dynamic or absolute. Usually, the `PRIORITY` line in `CONFIG.SYS` indicates dynamic priority. This is the default as installed, but it can be changed by the user to absolute. If priority is dynamic, the scheduler/dispatcher component of OS/2 will dynamically modify priorities of threads based on which is in the foreground, on whether the thread has been starved for a certain period of time, and other criteria. In general, every thread created is intialized with a default priority somewhere in the Regular class. This priority can be subsequently modified, either by the thread itself or by the scheduler/dispatcher.

If a thread is to modify its own priority, it can do so by a call to `DosSetPriority`. The parameters to this function are simply the new priority for the thread, along with the *scope*. The scope indicates whether the new priority is for that thread alone or for all threads in the process.

The Time Critical class is for threads that perform, as you might guess, time-critical operations. Examples would be threads communicating through a serial communications port, where two-way response is critical to process function, or threads gathering data from a peripheral device. It is rare for a Time Critical thread to be preempted, as it can only be preempted by a higher-priority Time Critical thread.

The Fixed High class is for those threads that need a somewhat higher priority than the Regular class, but are not really time-critical. An example may be a thread that reads from a tape drive asynchronously. You would want a thread such as this to be of higher priority than the threads reading the buffers being filled.

Let's look at this more carefully, to see why the thread reading from a slower device needs to be at a higher priority than the one that is CPU-bound and reading those buffers. Since the thread reading the tape is I/O-bound, every time the request to read the tape goes to the device, it will block and the CPU-bound thread reading the buffers will be dispatched. Since it is CPU-bound, it will run to the end of its time slice unless it is preempted by another thread. If left at the same priorities, the CPU-bound thread will starve the I/O-bound thread. By bumping the I/O-bound thread's priority slightly higher than the CPU-bound thread's (not necessarily higher than every thread in the system), you ensure that the I/O thread is dispatched whenever it is ready to run.

Regular class is where every thread starts out. When a thread is created, either by the loader when starting a process or when another thread calls `DosCreateThread`, its priority is in the Regular class. In this class, the scheduler dynamically modifies priorities.

Every thread in the Regular class is eligible to have its priority modified by the scheduler/dispatcher. Some examples of how priorities are modified include raising the priority of threads belonging to the foreground process and boosting up priorities of threads that have been waiting for processor time longer than others. This dynamic modification occurs only within the Regular class. Don't use `DosSetPriority` to change the priority of a thread within the Regular class, because it will likely be changed by the scheduler/dispatcher soon afterward. In all other classes, any priority set by the application will stay unless

reset by the application. There is one other fact to mention: The scheduler/dispatcher will never modify the priority of a thread to take it out of the Regular class. They will all stay Regular unless the application changes it to another class.

Finally, the lowest class in the hierarchy is the Idle class. This is like Time Critical and Fixed High in that the scheduler does not modify priorities of threads in this class, and will not lower a thread's priority from the Regular class into Idle class. Idle class is for threads that can wait until the processor has nothing else to do. This is useful for such tasks as a background file-mirroring program. Any type of background task is a candidate for the Idle class. You don't want such a task preempting "real" work, so the Idle class is the place for it.

Thread priorities and when to change them will be discussed in depth in Chapter 18.

Thread Management

Threads are managed though various control and synchronization structures. The management of the threads and the use of these control structures is completely up to the designer and programmer. Of course, there is the control built into the OS/2 system services as discussed previously, but most of the control of threads, the amount of parallel activity within an application, and how that activity is managed is up to the developers.

The basic control structures are semaphores, queues, and pipes. Actually, the only control structure in that list is the semaphore. There is another structure, although not used as often, called a *critical section*.

A *semaphore* is a structure that when implemented properly will ensure that only one thread in a process can manipulate a resource. A semaphore is like a flag, that only one thread can have captured at any moment. If another thread requests the flag while the first one still has it, this second thread will be blocked until the first completes what it is doing and clears the semaphore.

You could implement this simply by using a software flag, and indeed, that is one type of semaphore available to you. However, by letting OS/2 manage the semaphore control, you ensure consistency in all your

code and between your code and the other code that is in the system. It is important that you use semaphores for any data that can be accessed by multiple threads, whether it be in a disk file or in memory.

OS/2 also provides several flavors of semaphores. Each type of semaphore serves a specific purpose. For example, aside from the standard synchronization semaphore, there is the `MUTEX` (mutual exclusion) semaphore that is used to control access to a shared data structure. A way to control access to a structure would be to put all access to the structure in one function that requests a `MUTEX` semaphore on entry and releases it on exit. This way, if one thread is accessing the structure, others will be made to wait on the semaphore.

Another type of semaphore is the `EVENT` semaphore. Whereas the `MUTEX` semaphore controls access to structures, the `EVENT` semaphore is used to manage order of execution among threads. The `EVENT` semaphore signals an event to all waiting threads when posted. `EVENT` semaphores could actually be used as `MUTEX` semaphores as well, given the proper coding around the structure, but OS/2 provides both.

Yet another type of semaphore is the `MUXWAIT` semaphore. This is a semaphore that allows a thread to wait on a list of events, not just one at a time.

The OS/2 semaphore mechanism allows programmers the flexibility to manage threads at a variety of levels and synchronize them for any situation.

Memory Management

Memory management is one of the most interesting features of OS/2. The memory management system in OS/2 not only protects applications from accessing memory of others, but it also allows applications to access more memory than is physically in the system, and it has constructs that allow applications to share memory.

Memory management has changed dramatically from 16- to 32-bit OS/2. In 16-bit OS/2, memory was addressed with 16 bits, and words were 16 bits. 32-bit OS/2 supports both 16- and 32-bit programs. The underlying architecture is the same; however, there is an added layer for 16-bit programs running under OS/2.

First, 16-bit memory management will be outlined (to give you a frame of reference), and then the discussion will expand to the memory management implemented in 32-bit OS/2, exploring full "32-bitness" and 16-bit compatibility. You should also note that OS/2 for PowerPC is a 32-bit system and code written with the 32-bit API for OS/2 will port almost seamlessly to the PowerPC artchitecture. Therein lies the power of a consistent 32-bit API.

Figure 5.3 shows how memory management works in 16-bit OS/2. This structure does not change for 16-bit applications running under 32-bit OS/2.

All memory references are similar to that in DOS, where the reference is to a *segment*, or paragraph, in memory and an *offset* into the segment. This is a 16:16 address: 16 bits for the segment, and 16 bits for the offset. Under DOS, this value pointed to physical memory. Since only one program runs at any time in DOS, applications could use memory without regard for any other application, since there was no other application running at the time.

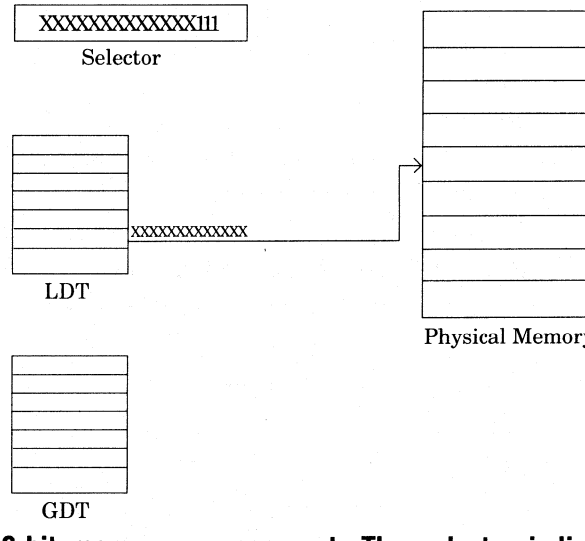


Figure 5.3 16-bit memory management. The selector indicates a memory access at ring 3, using the LDT for the process. The "XXXXXXXXXXXX" is the offset in the LDT where the descriptor points to the physical memory.

OS/2 adds a new twist to this. Rather than a segment and offset (although they are still sometimes called that), the address consists of a *selector* and an offset. The address is still 16:16, but the segment portion is now a selector.

If you look at Figure 5.3, you will see that a selector is similar to a segment value, but rather than pointing to a physical segment of memory, it points into a table, called a *descriptor table.* (This is a construct supported by the 80286 and later processors.)

There are two types of descriptor tables in the system. The first is the Global Descriptor Table (GDT). The system contains one of these, which maps the system address space. The other type of descriptor table is the Local Descriptor Table (LDT). Each process in the system has an LDT that maps the local, or process address space. The contents of each type of descriptor table have the same form. Each entry in the descriptor tables is a descriptor that maps a virtual address (a selector) to a physical address.

The selector indicates several things to the memory management code. Figure 5.4 shows the structure of a selector. Although it looks just like a real segment address, it is really composed of a privilege level, an indicator of which table (the GDT or the LDT for that process) to look in, and an offset into the table.

At the offset into the table is a descriptor that contains all the information needed about the physical segment of memory. The descriptor is defined by the Intel processor and is fully documented in the Intel manuals. Basically, the descriptor describes the segment of memory.

A descriptor contains, among other things, the privilege, or ring, level required to access this particular segment, an indicator of whether

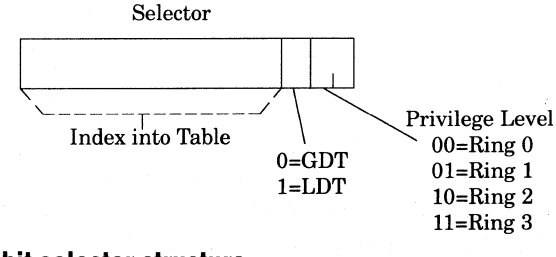


Figure 5.4 16-bit selector structure.

the segment is physically in memory; and if so, what the physical address is. This level of indirection is necessary to allow OS/2 to manage segment swapping and movement with a minimal impact to the application.

Once memory is allocated, the application is given a selector it can use to access that memory. That selector can always be used as long as the application wants. This gives the appearance to the application that the memory is always at the same location, but OS/2 knows that it really may be swapped out, or that it may have moved since last accessed.

Because the applications do not know about what is in the descriptors and access memory only through selectors, OS/2 can move memory or swap it out, completely transparent to the application, by simply manipulating the descriptors. When the memory is needed and is accessed, the OS/2 memory manager causes the swapped segment to be brought back in and then fixes up the descriptor. This indirection allows OS/2 to provide applications with more memory than is physically in the computer and to make efficient use of the memory that is there.

32-bit memory management adds in a new twist and better performance than 16-bit memory management. 16-bit constructs still exist for 16-bit application compatibility, but they do not hamper pure 32-bit operation.

32-bit memory management uses a feature of the 80386SX and later processors called *paging*. Paging is very similar in concept to 16-bit swapping except that with swapping, each segment can be of any size up to 64K bytes. Paging uses fixed-size, 4K pages.

Since each page is the same size, you can see immediately that segment movement is no longer needed, since all "holes" in physical memory are the same size. Each page fits perfectly. That is only the tip of the iceberg in performance gains with 32-bit memory management.

Figure 5.5 shows how 32-bit memory management works. The first thing you will notice is that 32-bit memory addresses are not 16:16 addresses; rather, for all practical purposes, they are simply 32-bit linear addresses in a flat address space. The linear address, however, is really an ordered tuplet that represents offsets into memory management structures.

Bits 22 to 31 in a linear address specify an offset in the *page table directory*, which is a system structure. The page table directory contains

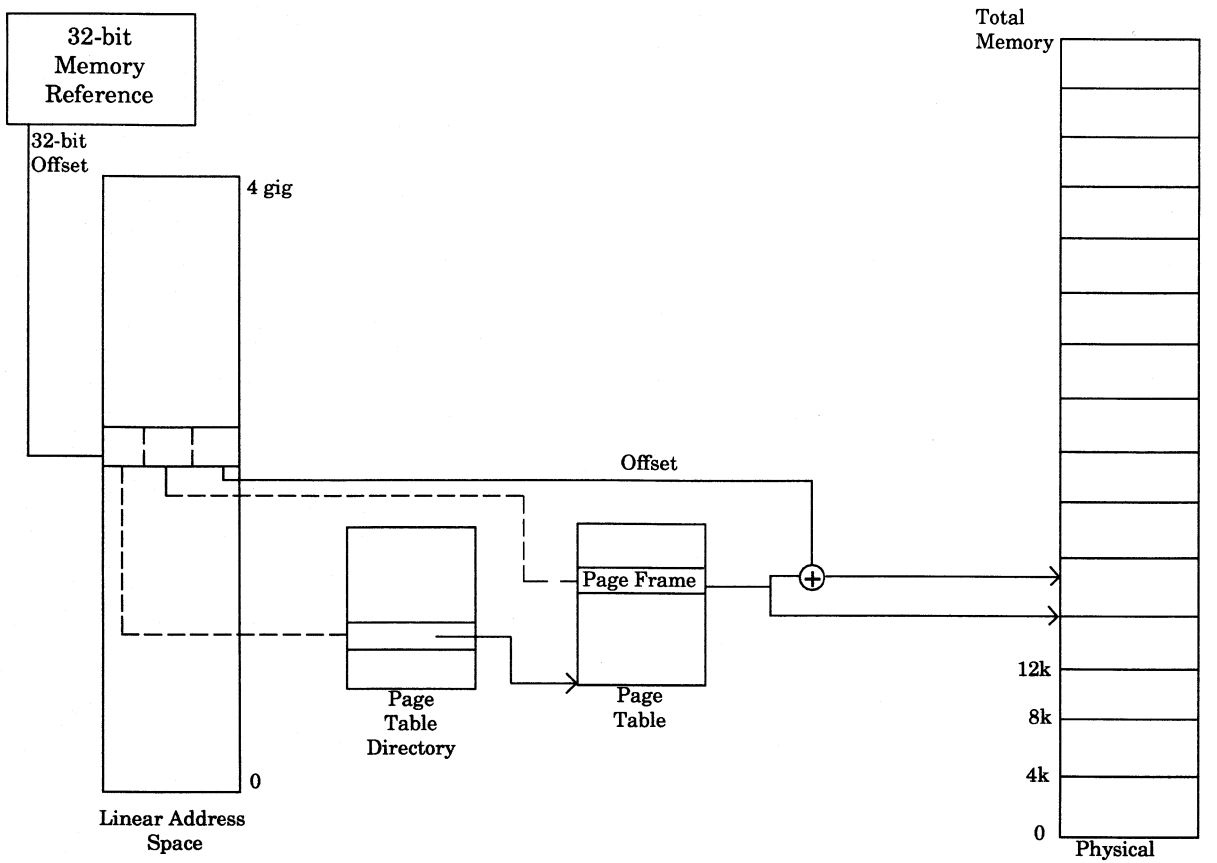


Figure 5.5 32-bit memory management.

pointers to *page tables*. Each page table consists of *page frames*, which are analogous to descriptors in 16-bit memory management.

Bits 12 through 21 of the linear address are used as an offset in the page table to indicate the page frame in which the memory reference is contained. Like a descriptor, not only does this page frame entry contain the physical address of the frame (analogous to a segment)—assuming the frame is in physical memory—but it also contains other indicators to indicate attributes, including whether the page is read/write, if it is present, and whether it is accessed.

Bits 0 through 11 of the linear address are used as the offset into the page of the actual byte being referenced.

This is how 32-bit memory accesses are accomplished. The linear address, which is the virtual address in 32-bit memory management, is an offset into the page table directory to obtain a page table; an offset into the page table to obtain the page frame; and an offset that is added to get the byte within the frame in memory.

Because of this scheme, memory accesses are even faster. No longer do segment registers need to be loaded. Since addresses are effectively 0:32, memory references do not cause segment registers to be reloaded. That is a very expensive hardware operation, since each time a segment register is loaded, all access permissions are checked based on the value being loaded. This is why addresses are really 16:32. The operating system has one segment, into which all 32-bit addresses are offsets. So even though 32-bit addresses are viewed as 0:32, they are really 16:32.

16-bit applications are supported very elegantly even when the underlying architecture is 32-bit. As you can see in Figure 5.6, it is elegant yet simple.

16-bit applications still require the use of 16:16 addresses, because that is all they know. Quite simply, what comes out of the descriptor table is not a 16-bit descriptor but a 32-bit linear address, which gets translated through the normal 32-bit address translation. In a 32-bit application, memory references are 32-bit addresses; in 16-bit applications, the addresses are 16:16, which go through the normal 16-bit address translation and *then* through the 32-bit translation to get the physical memory.

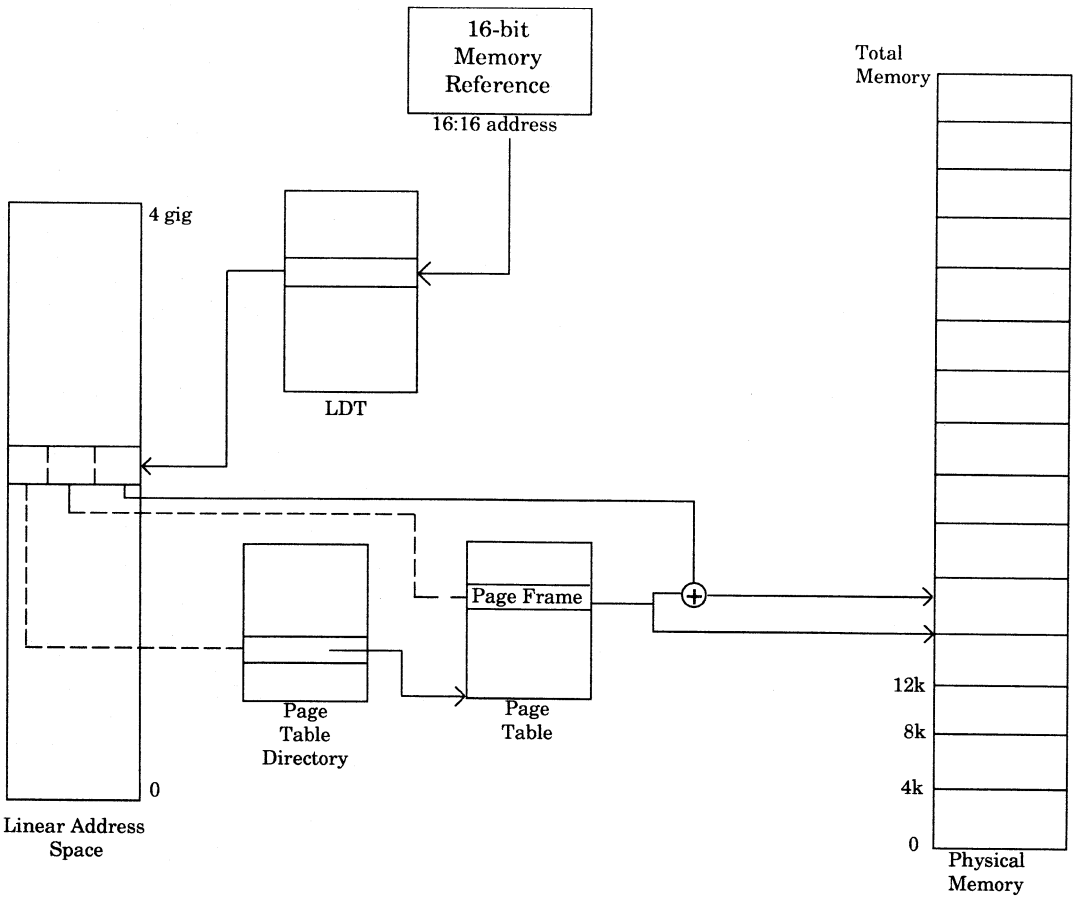


Figure 5.6 16-bit memory access in 32-bit OS/2.

Another interesting and efficient feature of OS/2 32-bit memory management is the concept of *sparse memory allocation*, which gives applications the freedom to allocate memory without actually committing physical memory or swap space. In addition, there are other structures, such as *guard pages*, which are used to notify an application how physical memory is reacting to its operations.

Each page of memory has a set of attributes. With respect to sparse allocation, memory can have the attributes of committed or guard pages. There are other attributes, but these are the ones that deal with sparse allocation. When pages are allocated by the loader, for example, the application has no direct control. These pages are the standard code and data pages for the application, and the loader will mark a page read, read/write, execute, and so on. The application controls when memory is allocated via `DosAllocMem` or other memory management APIs.

An allocated but not committed page simply has addresses reserved in the requesting process's linear address space. An allocation does not necessarily cause the physical space to be used. If the allocation request specifies the flag to commit the memory as well, then the physical "backing store" is allocated as well as the address space. The other way the page can be committed is by calling `DosSetMem` to commit a page for a previously allocated but uncommitted page.

Guard pages have a special purpose. They are initially uncommitted. When an address in a guard page is touched, a guard page exception is raised. If the application has registered an exception handler, it will be called. The default system exception handler for a guard page exception is to change the page to a committed page. However, the exception is raised nonetheless and is a valuable notification mechanism for applications.

A good example of sparse memory allocation is in the use of large, complex data structures, such as multidimensional matrices or hash tables. Such structures have the property that they need to be large, but they have many holes, with slots filled sparsely throughout the structure. By using sparse allocation and these notifications, you can allocate, say, a 10-megabyte linear memory object without committing it. You can set the guard page attribute on the pages in this object and register a

guard page exception handler. Inside this exception handler, you will use `DosSetMem` to commit each page as it is needed. In this way, you can manage the memory object without unnecessarily eating real memory, and/or swap file space.

There are many considerations when designing how your applications will use memory, ranging from how you will structure the code and data to how you will allocate and use memory to the most efficient way to access pages. We will discuss these in more detail in Chapter 12.

File System

The OS/2 file system has many parts. The standard file system, based on the File Allocation Table (FAT) mechanism first introduced in DOS, is always part of OS/2 and is built into the OS/2 kernel. OS/2 supports installable file systems, which become extensions to the kernel and are definable by the user. OS/2 supplies an installable file system with the OS/2 product, called the High Performance File System (HPFS). Other file systems are available, such as one for the LAN and another to make IBM AS/400 disk drives appear local to OS/2.

Figure 5.7 shows the "black-box" architecture of the OS/2 file system mechanism. Whenever an application or any part of the operating system, such as the swapper component of the memory manager, requests a file system operation to be performed, the request goes into a part of the kernel called the File System Request Router. This is simply a redirector that determines, from the file handle being accessed, which drive is being accessed. Depending on the drive being accessed, the request router will direct the request to the proper file system.

For example, if the request was to read from a file on a FAT drive, the request router will forward the file system read request to the FAT component of the kernel, which will honor the request and return to the caller. The file system is the subsystem that interacts with the physical device driver to satisfy requests, as has been outlined earlier.

Another example is if the request is for a drive using the file system for the AS/400. This is a product separate from OS/2 that is available. In this example, the request router will take the file system request and,

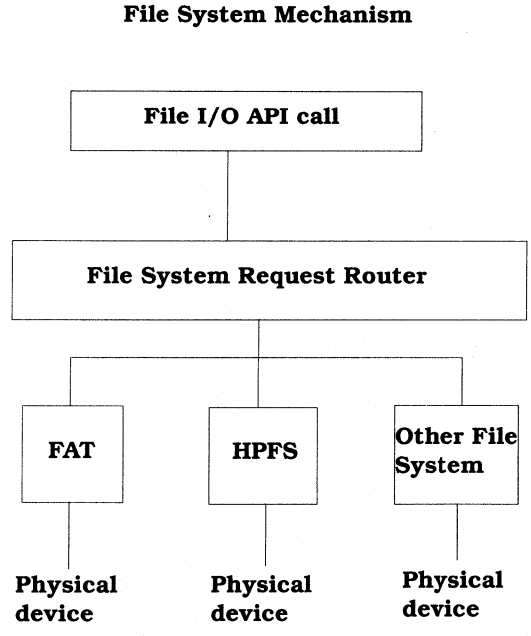


Figure 5.7 The OS/2 file system mechanism.

based on the file handle, will resolve it to be the drive on the AS/400, needing the AS/400 installable file system component.

Once the request router forwards the file system request to the AS/400 file system, it is routed over the token ring connection to the AS/400, where there is accompanying code that talks to the physical device. In this case, the OS/2 file system being used is not talking directly to a physical device or device driver. It is sending a request over a communications link to another machine that has the physical device.

As you can see, each file system can access its device in a different way, and OS/2 can have several different installable file systems in the systems at one time, so that users can access many different file systems in the same machine.

Device Drivers

Device drivers are the way physical devices are managed in OS/2. The device driver mechanism is more structured and defined than was the

case in the DOS world in which each application came with its own device driver, and device drivers were not generally usable by more than one application at a time. This was okay, though, as DOS could run only one application at a time.

OS/2 has a well-designed device driver mechanism and has standard, published interfaces to these device drivers and a special API to access their functions. Device drivers must be able to handle many threads requesting services concurrently as shown in Figure 5.8.

Device drivers are loaded via the `DEVICE=` statement in the file `CONFIG.SYS`. At this time, they are called at an entry point called the *initialization routine*. In this initialization routine, the device driver will

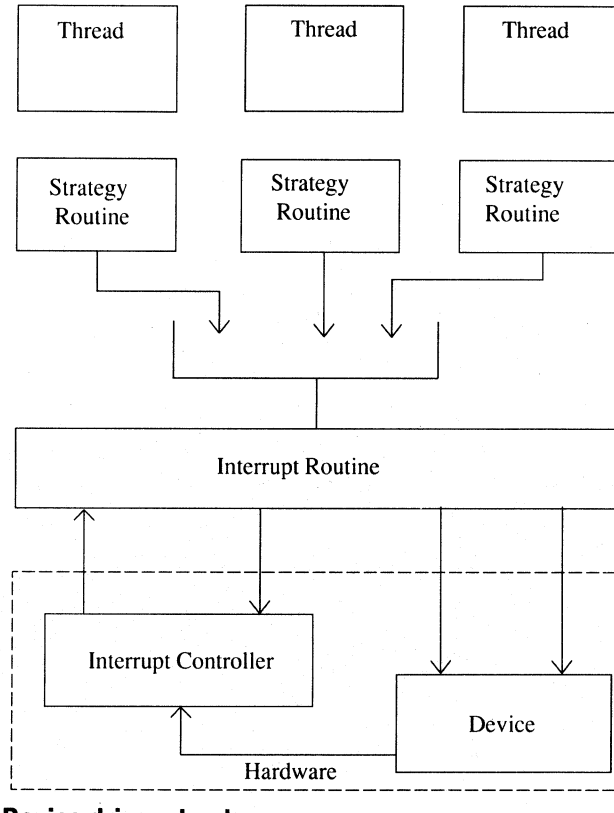


Figure 5.8 Device driver structure.

initialize the device, set up any control or data structures it needs, and then signal to the kernel that it is done.

Once the initialization is done, the device driver remains dormant until its services are requested. As you have seen earlier, a device driver's services are requested through an application calling into one of the OS/2 subsystems. The subsystem will at some point need either to tell the device to do something or to return some information. This is the job of the device driver.

Device drivers are told what to do via an I/O Control (`IOCTL`) packet. This is a packet of data, unique to each device driver, that tells the device driver what to do. Every device driver has a set of functions and subfunctions that it knows about, and that are specific to that device. For example, the disk device driver can be called to tell whether a drive has removable or fixed media, but that function is not applicable to the keyboard.

Subsystems use these `IOCTL` packets to communicate with device drivers. Applications too, can communicate directly with device drivers by using the API `DosDevIOCTL`. There is a drawback to using this from an application—that is, you are tying the application to specific OS/2 device drivers. If there is an API to perform the function you wish, it is better to use the API than `DosDevIOCTL`.

Once the `IOCTL` packet is sent, the device driver is called at its *strategy* routine. This routine is a router to the correct part of the device driver to service the request. The part of the device driver that services requests is the *interrupt routine*.

The interrupt routine is the part of the device driver that actually executes `IN` and `OUT` instructions to the port associated with its device. That is why device drivers need to run at ring 0: They need direct hardware access. The interrupt routine sends the request off to the device, blocks the requesting thread, or provides the information to the requestor itself, depending on the circumstances. Recall the discussion on subsystems and threads.

When a thread calls `DosDevIOCTL`, for example, it causes a jump into a DLL, which passes the request and packet into the device driver. This passing of the request packet is performed by the requesting thread. The

device driver uses the requestor's thread to do the work in the interrupt routine. When the request is sent to the device and it will take time before the request completes, as in the case of a disk access, the device driver executes code to block the thread. This is, again, the requestor's thread.

Another thread can come in with another request while the disk is physically processing the first. There is no reason the device driver cannot begin processing the request while the first is working, so the thread from the first request remains blocked while the thread from the second request is used to process it up to the point where the device driver needs to serialize access to the device and therefore blocks until the first request completes.

When the disk is finished with the first request, the device driver works with the scheduler to make that thread ready to run, while it also releases the semaphore for the second thread to grab. This wakes up the second requestor's thread. The device driver will use this second thread to communicate the request to the device and block it waiting on the physical device, just as with the first request.

The physical device drivers are used in this manner to serialize access to the ports, such as the disk, display, printer, and serial communications.

DLL Mechanism

Dynamic link libraries (DLLs) are used for performance and maintainability. Since there are many common routines used during system operation, having each one bound to each executable creates many copies of the same function in memory at the same time. Additionally, if a routine changes due to enhancements or, dare I say it, bugs, any program using this routine would have to be rebuilt and reshipped. This is not a desirable situation. By using DLLs to share many common functions, memory savings and maintainability are gained.

Under DOS, or any statically linked system, application code is compiled and linked with static link libraries, which binds the runtime code to the application code. When the application calls the runtime rou-

tines, they are in the application's own code segment, so a simple call is done. The problems are those just listed, however: maintainability and memory size.

The DLL mechanism is fairly simple, yet powerful. DLLs use the same paradigm of building the applications, but the libraries used are somewhat different. A static link library has the runtime code in it. A DLL's library file has an external reference record rather than the actual code. The library (`.LIB`) file has the references, while the DLL has the real code.

When the application is linked, the references to the runtime code are replaced with these external reference records, which are fixed up when the application is loaded and run. Fixing up the reference means loading the DLL and resolving the address.

When an application is built with DLL references, not only are the external reference records there, but all DLL references are listed in the `.EXE` file header. When the application is loaded, the `.EXE` file is read, as we have seen before, and all DLLs referenced have references resolved or are loaded. Then the external references are resolved into real addresses. The application can then call the DLL's routines using `CALL` instructions.

As applications resolve to functions in DLLs, the system keeps a usage count of how many applications are using the DLL. Once that use count goes to zero, the system will unload the DLL from memory.

This is the idea behind the OS/2 subsystems. Since all applications have to call a set of common subsystems to request system services, it is only natural that these system services be implemented in DLLs.

Base Subsystems

The OS/2 base subsystems are what virtualize the base system services, such as the keyboard, mouse, and display, giving the user a separate, virtual console, or computer, in each session. As you have seen, device drivers handle a large portion of the work of serializing access to physical devices. The subsystems are higher-level routines that communicate with the device drivers to gain information and instruct the de-

vices to perform tasks. The subsystems provide a consistent, hardware-independent, high-function interface to the system functions.

OS/2 for PowerPC shows that the OS/2 API is not processor-dependent and the underlying subsystems can be ported to a different processor architecture, or in the case of the "Workplace" architecture of OS/2 for PowerPC, a different operating system architecture as well. The key is that to you, the application developer, nothing is different.

SUMMARY

As you have seen, OS/2 is a very complex, yet easy to understand system. The kernel is the center of the system, and all of the low-level services are built closely around it. Higher-level services are then built on those and so on, until you get to the services accessible to applications. This isolation is important to allow you to keep your code separate from the underlying hardware and to allow the operating system to be changed under it without affecting application code.

So far, we have just covered the base system. Built on top of these services are more powerful functions, such as graphics, window management, the print subsystem, and the shell. The architecture of these pieces of OS/2 will be covered in the next chapter.

CHAPTER 6

Presentation Manager, Graphics, and the User Interface

Now that you have an understanding of the kernel and base subsystems of OS/2, we can move on to the higher functions: the Presentation Manager, graphics subsystems, printing subsystem, and the Workplace Shell.

In a pure sense, these items are really just applications, all using the services of the base subsystems. Actually, when you think about it, the only services in OS/2 that are not applications are the kernel and device drivers.

OS/2 is built in layers, each providing more isolation from the hardware than the one just below it. Of course, when you get too deep in this layering, performance begins to suffer. OS/2 is optimized to take advantage of the isolation as well as improving performance. A good balance has been struck between functionality and performance.

You have already seen the lowest layer—the kernel and base subsystems. The next layer up contains the graphical subsystems and the shell. These components of OS/2 use the base subsystems extensively and provide more powerful features to applications.

Of course, your applications can use any of these functions, but the higher-level functions are more powerful and efficient than if you were to code the same function using only the base subsystems. Performance does not suffer; to the contrary, performance is in many cases better than if you were to code the functions yourself, because they have been refined and tested by the OS/2 developers and testers.

As you design and write code, I think you'll actually find it fun to use these higher-level functions, especially when you see how much you can do with just a few lines of code. Windows can be created and graphics drawn with relatively little work, leaving you free to concentrate on adding and tuning powerful computing features of your applications.

PRESENTING DATA

Figure 6.1 shows a different look at the "black-box" architecture of OS/2 than you saw in Chapter 5. Of course, the kernel and base subsystems are still there, but now you can see more detail of some of the higher-level subsystems.

This subsystem has a name of its own: Presentation Manager (PM). Presentation Manager is a set of system services that provide the graphics-based presentation functions to applications, other subsystems, and the user shell.

Presentation Manager consists of a set of DLLs that provide isolation not only from the virtualization of hardware, but also from the device altogether, while allowing applications to take full advantage of the various devices present, such as displays and printers.

Starting with OS/2 Warp, many of the system services DLLs, including the Presentation Manager DLLs, have been merged into one larger DLL, `PMMERGE.DLL`. In addition, the remaining parts of PM that were still implemented as 16-bit functions are now 32-bit functions. The en-

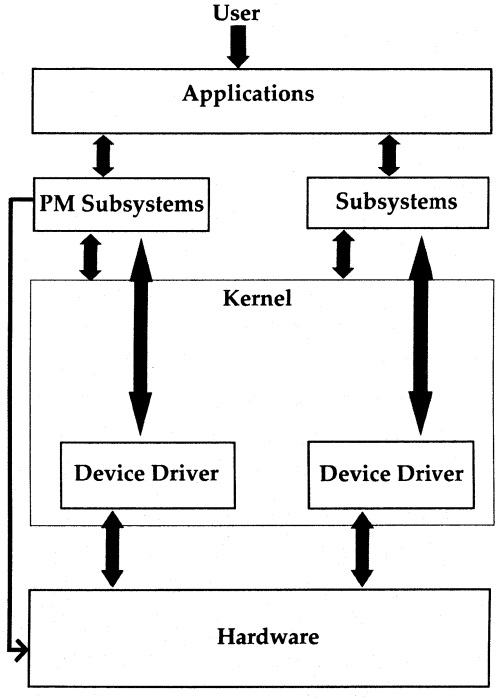


Figure 6.1 Black-box view of OS/2 architecture.

try points can still accommodate 16- and 32-bit callers, but the underlying implementation of PM is fully 32-bit.

By using the functions offered by PM, applications can take advantage of the devices present without having to know what the capabilities of the device are. The same code can be run on systems with differing hardware with no code changes at all, while still utilizing all of the features of the device.

This seemingly magical feat is accomplished through another function call mechanism with a very specific purpose. By setting up a few simple data structures and calling the graphics (`GPI`) and window management (`WIN`) APIs, applications can ignore such issues as whether the screen is color or monochrome; whether the printer is dot matrix or laser; or what the resolution of any of the devices is.

The flow through the graphics engine and presentation drivers is the key to accomplishing device independence.

PRESENTATION AND TRANSLATION FLOW

Everything that is displayed on the Presentation Manager desktop goes through the graphics engine. It is this engine, in conjunction with the presentation drivers for the screen, printer, and other devices, that performs the translation from a device-independent function call in the application to the specific commands to optimize and take advantage of the output device.

If you refer to Figure 6.1, you will see that the subsystems are broken up into the base subsystems and the PM subsystems. Figure 6.2 shows an even more detailed view of the PM API-to-display translation flow.

Of course, anything to be displayed on the PM desktop begins with an application request. This can be a user application such as a spreadsheet, a system application such as one of the utilities included with the system, or it can even be the Workplace Shell. All requests for data to be displayed on the screen come in the same way.

Throughout the next few pages, the discussion may seem to jump around a bit. The graphics-rendering methodology is quite complex, and in some cases there are circular references among data structures, graphics objects, and parts of the rendering algorithms. As the discussion progresses, the pieces will begin to fit together.

Device Context

A *device context* (DC) is a structure that links the device-independent data structures to the physical device. More precisely, the DC connects to the Presentation Driver for the target output device. The Presentation Driver is the workhorse of the translation process and works very closely with the graphics engine to map the device-independent graphics orders from the applications and turn them into a device-specific data stream. Each DC is linked to one device.

The device-independent graphics orders come to the DC from a *presentation space*.

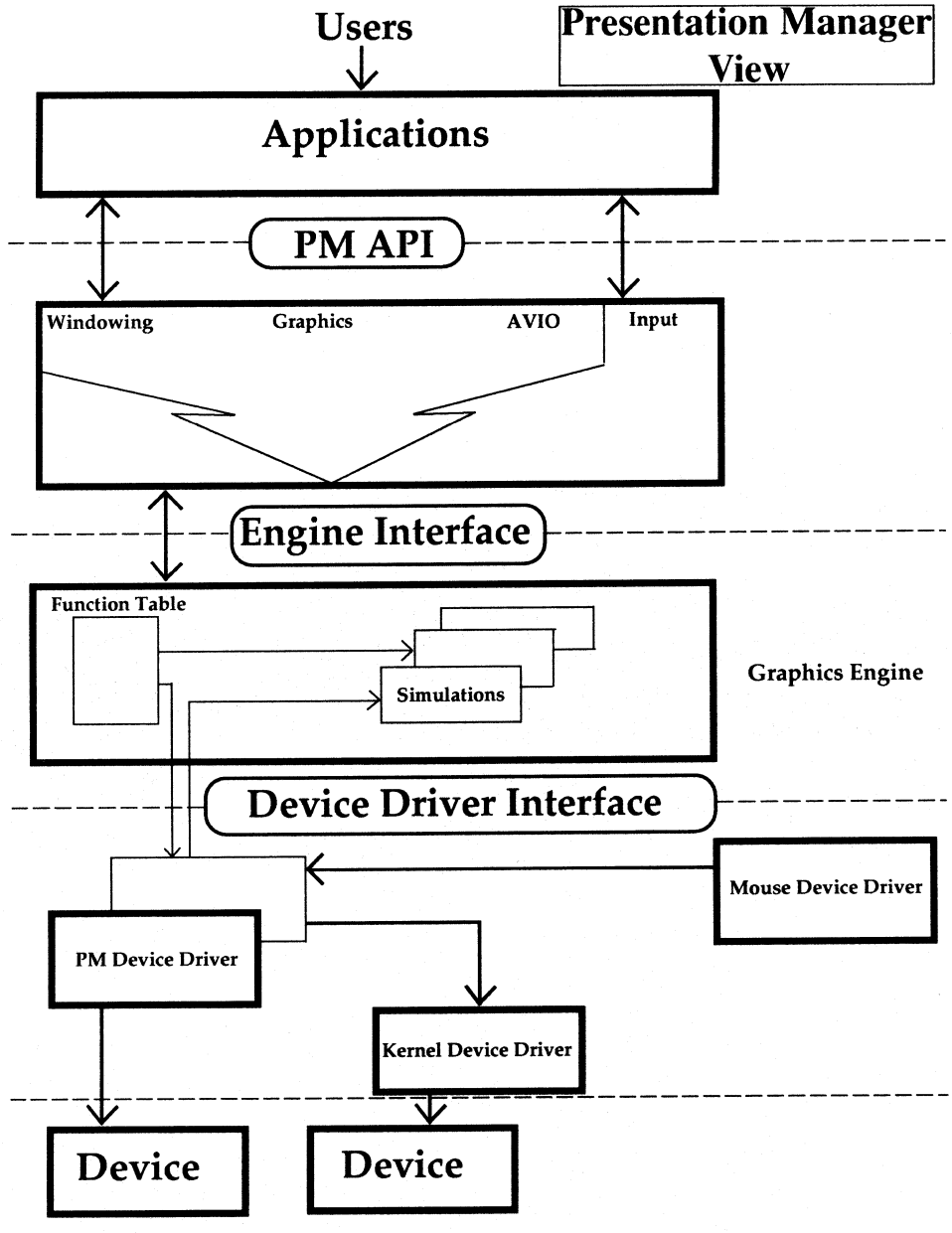


Figure 6.2 Detailed view of OS/2 Presentation Manager architecture.

Presentation Space

A presentation space (PS) is a device-independent entity used for drawing. All drawing is done "into" a PS. There are several types of PS, each with its own properties of persistence. The varying degrees of persistence describe how each type of PS maintains its state across function calls along with its performance.

Before any drawing can be done anywhere through Presentation Manager, a PS must be created. However, in order to draw into a PS, you must first have a device context (DC).

Figure 6.2 depicts the relationship between the PS and the DC. Neither is a very tangible object, and the internal structure is irrelevant. The important point is that they are needed and they represent a significant link in the application to device translation. The presentation space is maintained at the graphics-engine level, and the device context is at the presentation-driver level, according to Figure 6.2.

Whenever an application draws, it draws into a PS. Every PS is associated with at most one DC. The DC is associated with the device.

Tracing a Drawing Call

Any drawing request is a function call, just like all system services. Let's assume for this example that the user has requested the application to draw a box. Notice that I did not specify on what device this box is to be drawn. As you will soon see, drawing on a printer is virtually the same as drawing on the screen. The function for this operation is `GpiBox`.

`GpiBox`, like all the other functions that cause drawing to take place, requires (among other information) one key parameter: a handle to a presentation space. The PS is the key to all drawing in PM.

The PS has to be associated with a DC in order for the call to cause something to be drawn on a device. The call to `GpiBox` the code to jump into `PMGPI.DLL`, which does some work and, using the PS and DC link, makes calls into the graphics engine, which in turn calls the Presentation Driver.

These two entities will call each other for various functions and pieces of information until there is enough information available for the

Presentation Driver to send the device-dependent commands through the physical device driver to the device.

The graphics engine and the Presentation Drivers for the various devices are the most involved and complex pieces of this story. These two units complement each other to support the variety of devices, and they enable PM applications to output onto them with a minimum of effort.

Graphics Engine

The graphics engine contains a function table containing all of the graphics orders available along with simulation routines. It acts as a function resolver more than a function generator or engine, as the name implies. Figure 6.3 shows the high-level architecture of the graphics engine.

Let's go back to looking at the flow of a graphics function call. As you have already seen, the call originates at the application and goes into the PM subsystem layer for initial processing. In the case of graphics calls, this is `PMGPI.DLL(PMMERGE.DLL)`. Next, using the PS and DC connections, the request is passed into the graphics engine layer.

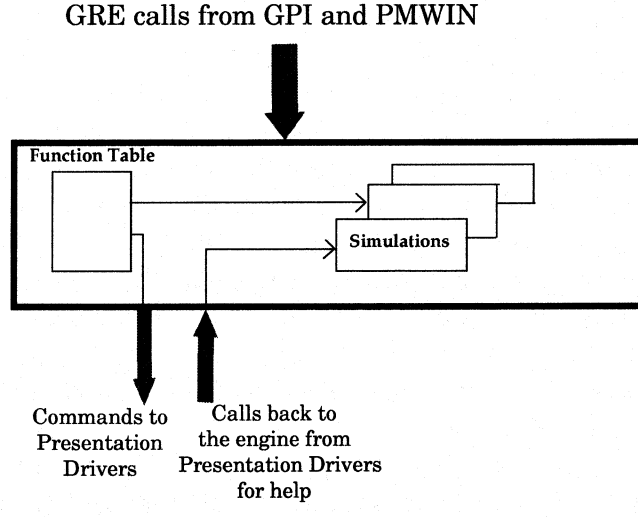


Figure 6.3 OS/2 graphics engine structure.

At the engine layer, the call is routed to a function table. This is a table that contains a pointer to the routines representing every graphics order available. As you will soon see when we discuss Presentation Drivers, there is a copy of the function table for each device and Presentation Driver. That is where the DC comes into play: to differentiate between the various devices and route the graphics functions to the correct place for the device.

The GPI layer calls the engine, using the DC to get the correct copy of the function table for the specified output device. The function table, specific to the device, points to the routines that resolve the device translation and optimization. These routines include functions in the Presentation Driver along with simulation routines in the engine itself.

Using the functions of both the Presentation Driver and the engine, calls are made to turn the complex graphics call into rudimentary functions the device can understand. This is where all the mapping from device independence to device specificity takes place.

The engine layer will use the function table to make calls specific to the request. The calls may be to its own simulation routines or to the Presentation Driver's routines, depending on how the function table for that device is set up. Calls are made back and forth between the engine and the Presentation Driver until the high-level request is a set of low-level, device-specific calls.

Let's now go back to the `GpiBox` function call. As you have seen, the call to `GpiBox` contains, along with the information about the box to be drawn, a handle to a PS. The function resolves into the GPI. At the GPI subsystem layer, the PS is examined to determine the DC with which it is associated. The GPI layer also determines what calls are to be made to the engine layer to cause the drawing to take place. In some cases, such as a line, this may result in only one call from the GPI to the engine. In others, it may result in many calls. This first layer is still device-independent, but the GPI is optimizing the call and simplifying it for the engine.

Once the engine layer is entered, each function in turn is called through the function table. Which copy of the function table is used is dependent on the DC, which is determined by the GPI layer from the

PS specified on the API call. The function table is used to process the call. In this `GpiBox` example, the function table would be called at the box entry points.

Let's now assume that this particular device is very smart and has a box function built into the hardware. The function table for that device would resolve the box function into the Presentation Driver, which would in turn know that the device supported box functions and, therefore, would simply generate a box request specific to the device and send that to the physical device driver.

Many devices, such as displays, do not have a box function. In this case, the Presentation Driver for the display would know that, and its function table would point into the engine for the box function. The engine, not knowing too much about the device, would break the box function down into a series of line requests and send these requests through the function table for the device.

Assuming that the device has a line primitive, the Presentation Driver would take those line requests and send them in a device-specific data stream to the device through the kernel device driver. The device would then render the box on the screen by following these requests for one line at a time.

Now let's go a step further and assume that the display does not have a line primitive that it can draw. In that case, the Presentation Driver for the display would call once again into the engine to break the lines down further.

The engine would not, however, break the lines down into a set of functions to turn pixels on and off. These pixel functions go to the function table for the display, which points to code in the Presentation Driver, which will send the pixel commands to the device. I can say this part with assurance because there is a common set of functions that all devices and, subsequently, all Presentation Drivers must support. A pixel on/off function is one of them.

As you can now see, the mechanism to translate drawing commands from an application to a specific device is straightforward, yet very powerful. Using this mechanism, any application can issue commands to draw in this generic fashion, and it is up to the Presentation Driver, along with the engine, to make the best use of the available hardware.

Presentation Drivers

The Presentation Driver is also sometimes called a Presentation Manager Device Driver (PMDD). There are Presentation Drivers for every device that is supported under PM. Recall that everything that shows up on the PM desktop goes through the graphics engine. Now you can see that for OS/2 to take advantage of any device in PM, there must be a Presentation Driver for it. Note that these are not device drivers with interrupt and strategy routines, as discussed in the last chapter—these are drivers, specific to PM, that understand the capabilities of the associated device. The main function of a kernel device driver is to coordinate access to a device. The Presentation Driver handles the PM device-independent translation to that device.

There are Presentation Drivers that ship with the OS/2 product, and there are others that have been written by various hardware and software vendors. As such, each is subject to many factors.

Brute-Force versus Full-Function Drivers

As you have already seen, the graphics engine simulations and the Presentation Driver functions complement each other. This is a very flexible architecture that benefits developers, in that there are only six functions that all Presentation Drivers must support in order to function. All others can be handled by the engine.

This is an advantage, because one can write a driver that supports only the basic six and, as time goes on, add more functions. With some drivers, however, the driver designer intended to support only the six root functions and let the engine handle the rest. This is what is called a *brute-force* driver.

A brute-force driver is one that does not take advantage of the device it supports at all. The driver functions just enough to allow the device to work with Presentation Manager. This type of driver is the slowest and takes the least advantage of the device. Since the engine is doing all of the work, and there are many more calls to render an object if the

driver does virtually nothing, a brute-force driver will be slower than a driver that is more intelligent. A full-function driver, on the other hand, will provide the best resolution and performance.

The trade-off is speed of development versus performance of the driver. Although it is not very desirable to release a completely brute-force driver that does nothing on its own, it is often useful to be able to release a driver that supports many, but not all, functions just to be able to get it out in the marketplace sooner than waiting until it supports all functions. The only drawback would be performance. However, for someone who has business riding on getting the product into the market sooner, the flexibility is extremely convenient.

The most common types of Presentation Drivers are for printers and displays, although others, such as for FAX hardware, are not unheard of.

Printer Drivers

OS/2 printer drivers are really just Presentation Drivers for printers. It is important to note that printers know only data streams, so whatever comes out of the printer port must be specific to the device. OS/2 is designed so that the applications need not know about what device is installed, so the best way to implement this function is with a Presentation Driver for the printer.

When a user adds or changes a printer, all the user need do is change the printer driver; all applications will be able to take advantage of the device and to output data to it as they would any other device. In contrast, DOS systems required the user to reconfigure each application to take advantage of the printer.

Screen Drivers

A screen driver is really a display Presentation Driver. Before the 32-bit graphics engine was put into OS/2, the display driver had a dual purpose: It acted as a rasterizer for printer functions as well as the translator for the screen. In the 16-bit OS/2 world, the printer driver or

the engine would regularly call the display Presentation Driver for help with printing functions. In 32-bit OS/2, the screen drivers are dedicated solely to display functions.

While there may be many printer drivers installed in the system at any given time, there is usually only one display driver.

Window Manager

The OS/2 window manager is `PMWIN.DLL`. If your user is running OS/2 Warp or later, the functions are now in `PMMERGE.DLL` but, for the purposes of programming, they are still exported from `PMWIN`. This subsystem is responsible for creating, manipulating, and destroying windows. `PMWIN` understands how and when windows overlap, cover, and uncover areas. It has the job of notifying applications when they need to paint their windows. It also manages all input in the PM session.

As with anything else in OS/2, all input and output in windows on the PM desktop must be requested from the system—in this case, the window manager.

Input

All input in the PM session is coordinated by `PMWIN`. Recall that each session is a virtual console so that each application has its own keyboard, mouse, and display. PM is a single session, so a mechanism had to be devised to allow the variety of programs that run at the same time, and are visible at the same time under PM, to have their own keyboard, mouse, and display.

Figure 6.4 shows the input structure of PM. All mouse and keyboard input in the PM session comes in through a queue. This queue keeps all user input time-ordered so that all input gets to the window it is intended for and type-ahead is preserved. From this queue, the raw input goes into a router that makes decisions about what to do with the input. For mouse input, for example, the router will determine what window the mouse was over when the mouse button was depressed or released.

The router also knows which window had the *focus* when a key or combination of keys was pressed. The focus window is defined as the

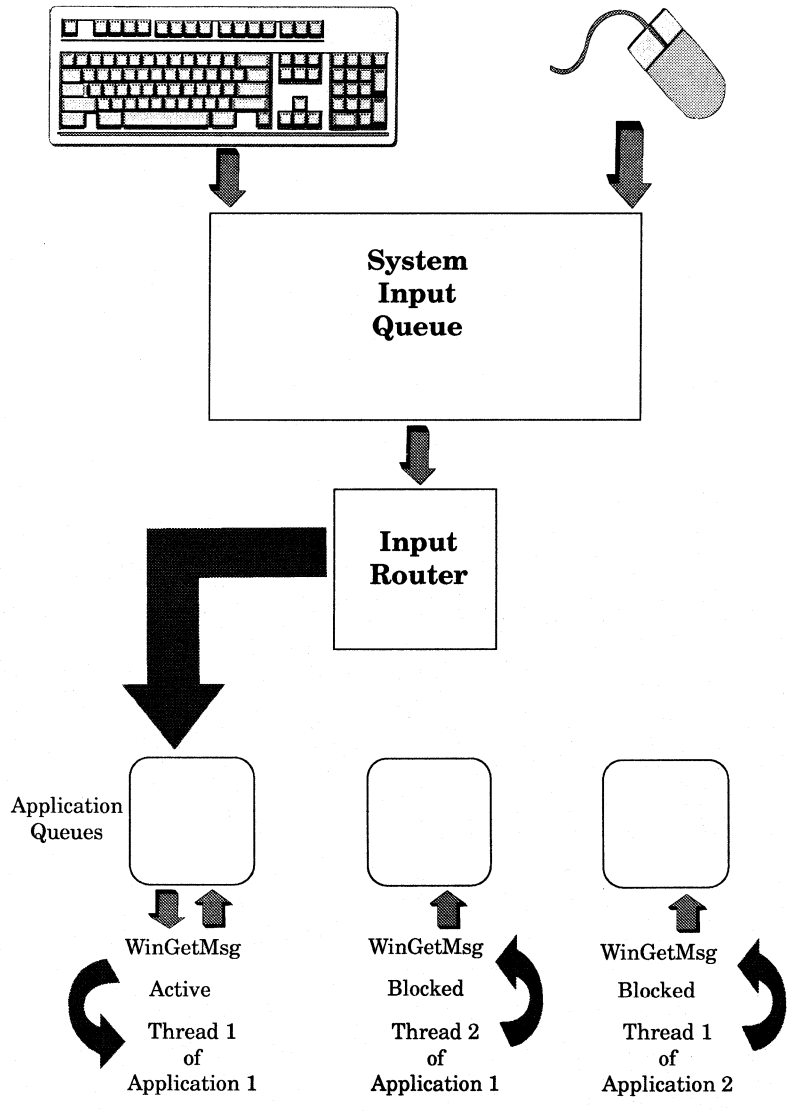


Figure 6.4 OS/2 Presentation Manager input architecture.

window to receive keyboard input. While mouse input simply goes to whatever window the mouse is over, keyboard input is directed to one window, which is called the *keyboard focus*. Focus can change from window to window, but there is only one focus window at any given moment. Once the router determines the window to which the input message is to be sent, it places the message on a message queue associated with that window.

This message architecture of PM demonstrates how the system is event-driven. Applications simply wait for messages that are provided by the input router. When a message is received, the application awakens, processes the message, and checks for more messages. If there are no more, the application goes back to sleep.

This can be accomplished by applications using semaphores, but `PMWIN` has another interesting feature built into it called the *message loop*. Yes, there is a loop recommended in an OS/2 application. It is what goes on inside this loop that makes it unique.

As the loop runs, it checks to see if there is a message on its queue by using `WinGetMsg`, which is an API in `PMWIN`. `WinGetMsg` will retrieve a message off the application's queue if there is one available. If there are no messages on the queue, `WinGetMsg` causes the thread to block; the thread will remain blocked until a new message arrives on the queue. At that time, the thread is awakened and processes the message. Then the thread will come back to the top of the loop to execute `WinGetMsg` again. The process repeats until the user requests the application be closed.

Aside from the obvious efficiency advantages of this scheme, you can use this mechanism to help you manage multiple threads of your applications without having to create and use semaphores and other structures yourself. This will be discussed in depth in Chapter 14.

The other important role that `PMWIN` plays is to provide event-driven function to OS/2. The PM subsystem needs to understand and keep tabs on which windows are owned by which other windows, which windows are where, and how movement and visibility change. `PMWIN` has the job of notifying windows when they need to repaint themselves.

Repaint notifications come by way of `WM_PAINT` messages arriving directly to the window procedure of the window. Under DOS, applications must always maintain their visual integrity and know what is going on

in the system. OS/2 Presentation Manager applications need respond only to system notifications of when actions need to be taken.

When a window is uncovered, or unhidden, it is PMWIN that tells it what needs to be repainted. When an application requests that its window be hidden, or the user clicks a button that causes the application to be sized, it is PMWIN that notifies the application.

PMWIN is very much like an application when it comes to painting and other window management. As a matter of fact, most of PMWIN runs at ring 3. PMWIN uses the services of the GPI and engine to display everything. As you have recently seen, everything going to the PM screen goes through the engine. PMWIN is a tool that provides OS/2 applications with a coordinated way of managing the windows on the PM desktop.

As you can see, OS/2's layering provides more and more powerful functions as you move outward from the hardware. OS/2 provides functions and subsystems for coordination and resource management on top of the basic multitasking of the threads and CPU.

WORKPLACE SHELL

The highest layer of function in the system is the user shell. OS/2 provides an object-oriented user shell that not only offers object-oriented features, such as drag-and-drop, to users, but also offers these features to applications. The OS/2 shell is called the Workplace Shell. It derives its name from the fact that it works the way people work: with objects. The architecture of the shell is just like the rest of OS/2: simple, yet powerful.

When OS/2 boots and processes CONFIG.SYS, the PROTSHELL statement is processed. The executable program specified in this statement is what starts shell initialization. The Workplace Shell is structured such that PMSHELL.EXE, the program specified in the PROTSHELL, is a *monitor* process. It sets up control structures and processes the RUNWORKPLACE statement. What is specified in the RUNWORKPLACE statement is treated as the workplace process. If for some reason the workplace process dies, PMSHELL will restart it automatically.

The Workplace Shell is fundamentally just an application, albeit a very powerful one that not only allows users to manipulate objects, but also externalizes its functions to allow applications to manipulate objects.

The primary function of any shell is a program launcher. The OS/2 Workplace Shell provides a set of functions on top of the basic program-launching function. It provides all of its interfaces as a set of objects, each with its own properties, following the concepts of object-oriented technology.

The OS/2 Workplace Shell is the first implementation of a technology called the System Object Model (SOM). SOM is an object-oriented tool and language for creating object-oriented (OO) systems. SOM has its own language and "compiler" that will take SOM language "programs" and convert them into standard language programs.

SOM basically consists of the compiler, a small class hierarchy with which to build a class library, an object manager, and a message resolver. It is a set of tools upon which the OS/2 Workplace Shell is based. These tools are supplied with the IBM OS/2 Programmer's Toolkit and are also found on the OS/2 Developer's Connection or the SOMObjects toolkit. Any application can use SOM to create object-oriented applications.

The Workplace Shell could have been implemented with C, C++, or any other language, but SOM was chosen for its portability and function. As you will see in the next chapter on OS/2 for the PowerPC, high-level programming tools such as SOM, C, and C++ make porting to other platforms very easy.

The object hierarchy of the Workplace Shell is defined by the shell itself. SOM is simply the tool with which it was implemented. SOM can be used for applications, to write Workplace objects, or even to write your own shell to replace the Workplace Shell. Some important clarifications need to be understood at this point. SOM is not the Workplace Shell, and the Workplace Shell is not SOM. SOM is simply the object-oriented programming tool used to implement the shell.

This goes into the other important distinction: A SOM object is not necessarily a Workplace Shell object. All Workplace objects are SOM objects, however. As you will soon see, SOM objects cannot (always)

access Workplace objects, even though Workplace objects are SOM objects. The distinction is in how the SOM engine is implemented and how the Workplace Shell is architected. This is related directly to the single-process model just described. The SOM engine can have many client processes, only one of which is the Workplace process.

All Workplace objects are descended from a base class called `WPObject`. The three primary subclasses are `WPAbstract`, `WPFileSystem`, and `WPTransient`. `WPTransient` objects are the easiest to understand, because they are not persistent. These objects go away between system reboots. An example of a transient object is the Task List. It is created when the user either hits Ctrl-Esc or presses both mouse buttons down at once (a *chord*). The Task List appears and, upon any subsequent mouse action, disappears.

The other two classes of objects are persistent objects. The main difference is how they store their persistence. Any object that is a subclass of `WPAbstract` stores its persistence information, such as color, state, and icon, in the `OS2.INI` file. Examples of abstract objects are the color and font palettes.

File system objects are subclasses of `WPFileSystem` and store their information in the file system. You may have wondered what the empty directories on your boot drive are. These directories are placeholders for *extended attributes* (EAs). EAs are a file system construct that allows data to be associated with a file or directory (the extended attributes of the file). All file system objects store their persistence information in extended attributes. The directory entries are there for no purpose other than to have something to associate EAs with.

Each directory entry represents one file system object. If you manually traverse the directory structure using command-line instructions, you will see all your folders and other file system objects represented there.

The OS/2 Workplace Shell has a two-process architecture. As you just saw, the first invocation of `PMSHELL.EXE` is the monitor process and the second (as specified in the `RUN WORKPLACE` statement) is the real Workplace Shell. All workplace objects run in the context of the shell process. For example, if you were to build a class that is a derivative of say,

`WPDataFile`, you would be building a DLL that is called by the Workplace process and becomes part of the shell.

This is one downside to the design. A problem arises when developers write their own objects and the object has some coding error that causes a protection violation. Recall that if any thread of a process traps, it takes the whole process down. Well, if a Workplace object traps, it takes the shell down. Although the Workplace Shell does have a built-in mechanism that will restart it in this case, it is inconvenient for users to have to watch the shell go down and come back up again. While this does not affect processes started by the shell (after all, 90% of what you are running will be in separate processes and will not come down if the shell does), it is not fun to have an object that brings the shell down.

OS/2 2.x uses the first version of the System Object Model (SOM), which is restricted to a single process. That is, for one SOM object to use SOM and talk to another, both objects must be in the same process. While the SOM engine can be used by many client processes, in order for one object to invoke methods on another, both must be in the same process.

Beginning with OS/2 Warp, the SOM engine shipped with OS/2 is the workstation-enabled DSOM. This means that the SOM engine shipped with OS/2 allows objects not in the same process to invoke methods upon each other. While the Workplace Shell is still implemented as a single process, it is enabled to use this new level of SOM engine, so no longer are you required to put derivative classes of the Workplace Shell object classes in the Workplace process.

This leads to an important set of design considerations: How much of your application should be implemented as objects? In general, you want to write custom objects to only start up the application and view data; any other functions of the application should be in a separate process and communicate with the object via DDE, the Clipboard, or some other interprocess communication mechanism.

Now that you have the capabilities of DSOM, you are not restricted to writing your object to run only in the Workplace process, but you also need to understand the performance implications in writing objects to run in the Workplace process, those that will run in other processes,

and how much code to put in separate processes. Interprocess comunication becomes more and more important as you write more fully functional stand-alone and client-server code. These topics will be explored in depth in Chapters 8 and 11.

SUMMARY

Designed properly, and with the knowledge of the underlying subsystems, applications can make the most of all OS/2 has to offer. The Presentation Manager provides device independence and isolation so that applications using PM services need not care about the capabilities of the hardware in the computer.

Each layer in PM has a set of functions it performs, ranging from the primitive functions in the engine all the way through to the powerful APIs in the shell and `PMWIN`. There are trade-offs in terms of power versus detail of control in each of the APIs at each layer of PM. By understanding how these subsystems interact and complement each other, you will be able to make better decisions as to which to use in different circumstances.

CHAPTER 7

OS/2 Warp for the PowerPC

A great deal has been said, written, and speculated about this thing called Workplace OS. It has been called a replacement for OS/2, an extension of OS/2, and, in some places, the "Sybil" operating system. Much of this stems from the newness of the architecture and, in some cases, a plain lack of understanding into where IBM is going with its operating systems technology. Even the term "Workplace" has undergone many meaning changes, and by the time you read this, it may have metamorphosed, yet again.

This chapter will not go into all the details and inner workings of Workplace, but I will show you what it is, where it is going, how it works, how it compares to OS/2 on the Intel platform (which is what you're used to) and most importantly, what you need to know to ensure your applications run there (of course exploiting the advanced functionality) with a minimum of work on your part.

WHAT IS WORKPLACE?

Workplace in this context refers to an architectural model for operating systems. Yes, the OS/2 user shell is also called the Workplace Shell, but the Workplace (or Workplace OS) architecture describes the foundation for the future of OS/2. The primary reason for this change in architecture, other than that it is better to build on, enhance, and maintain, is that it is much easier to port. The first implemenation of the Workplace architecture is in OS/2 Warp for the PowerPC.

Users buy computers to run applications. The applications are closely linked to the operating system they were written for. If that system is available only on a single hardware platform, the application market is also limited to that platform. In porting the operating system, users have more choice in the hardware they can select to work more efficiently. Their favorite operating system and applications will be there, and the only thing different is the circuitry.

In a nutshell, Workplace takes the monolithic architecture of OS/2 and breaks it up into modular pieces, providing a more robust and more portable platform for operating systems. It takes the kernel, which as you saw in Chapter 5 contains the tasking, memory management, file system, loader, exception handling, and other critical operating system functions and breaks it into pieces. Around the OS/2 Intel kernel are the subsystems; below it as an extension are the device drivers and at the top of course, the shell and applications. Workplace moves most of the kernel functions and device drivers out of the kernel and into user tasks.

At the core of this is the IBM Microkernel. On top of that are personality-neutral services, often called *common* and *shared* services, which are *servers* and *shared libraries*. As the names imply, these are personality-independent. What is a personality, though?

A personality is fundamentally the API frameworks. So, you can have an OS/2 personality, a DOS (also called the MVM) personality, an AIX personality, and so on. One of the personalities is called the dominant personality, which provides the user shell and manages critical system events such as initialization and shutdown. Since a Workplace OS ar-

chitected system can have several personalities, you can see how it got the "Sybil" nickname.

Before we get into some of the details behind this, you're probably wondering what this all means. In short, the Intel-architected, monolithic OS/2 you know has had its main plumbing replaced. The interfaces to the user and applications remain the same, but how it accomplishes its functions down in the core of the system is completely different. Your applications can remain the same and run just fine if you follow the information and recommendations in this book. If you try coding to hardware specifics, you lose the ability to port your applications along with the system.

So, what the user sees is OS/2. What your applications see is OS/2. Underneath is Workplace, and that gives you and your users the freedom to choose.

COMPARING AND CONTRASTING WITH INTEL OS/2

You've already seen the description of Workplace OS, or OS/2 for the PowerPC as OS/2 with the plumbing replaced. The point at which the pipes were cut and replaced are just below the API layer. That is, the APIs are the applications' interface to the operating system. The framework and API set is there, and it provides the same interface to system services, but where and how the system services are implemented are changed.

Figure 7.1 shows the current Intel OS/2 architecture. Now, look at Figure 7.2, showing the same level of detail of Workplace architecture.

The most glaring difference is the number of components in each. You'll notice that the Workplace Architecture takes some of the larger components and breaks them up into well-defined parts. This is fundamental to the Workplace methodology, which compartmentalizes functions that formerly were in a monolithic system.

You can see in Figure 7.1 that there are three essential layers in the Intel version of OS/2 today: the kernel/device driver layer, the OS/2 subsystem layer, and the application layer, as was outlined in the previous

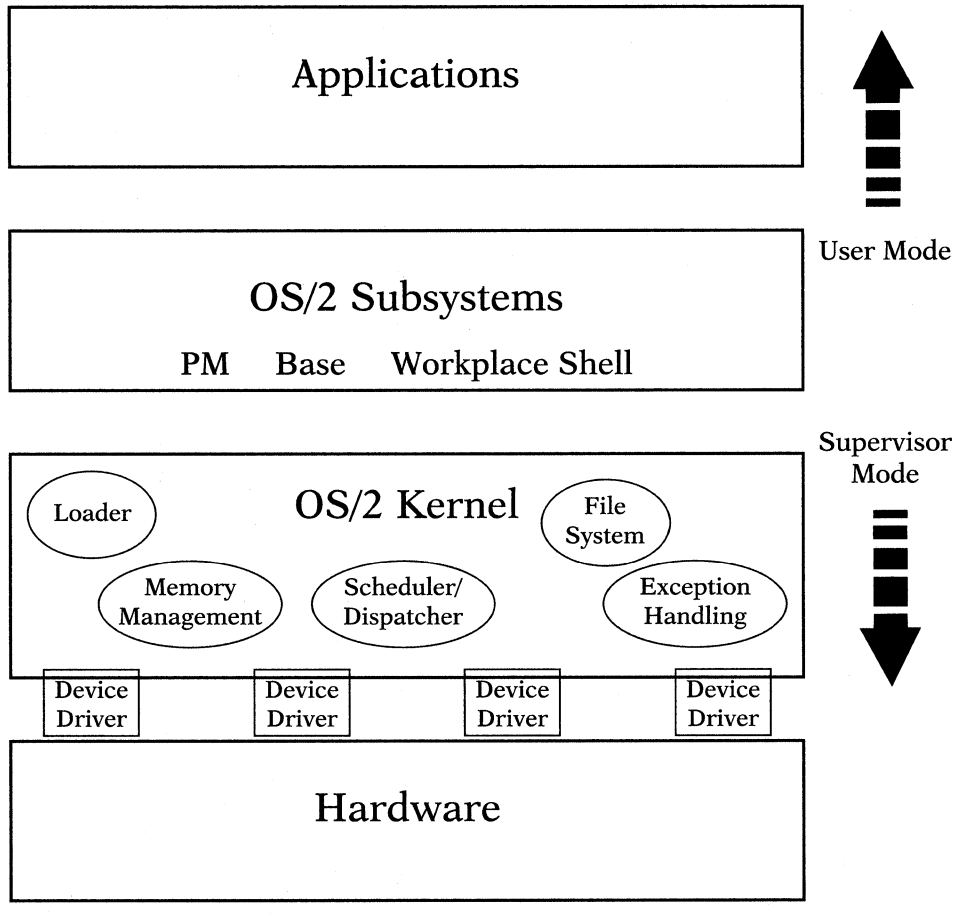


Figure 7.1 Intel OS/2 Architecture.

two chapters. These layers loosely map to the Intel protection rings, where the kernel and device drivers are the most privileged code, and the application layer up to it is the least trusted code.

The OS/2 kernel is the single largest part of the system. It contains all of the vital system functions including the system initialization, scheduler, memory management, program loader, process man-

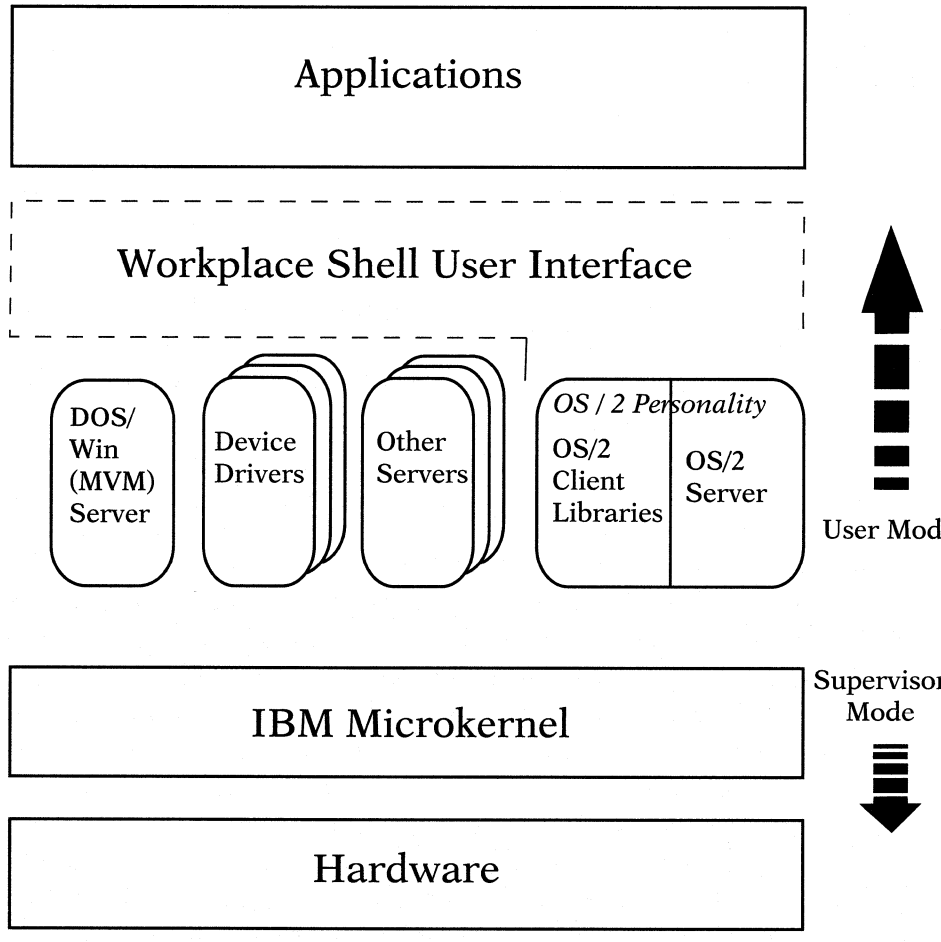


Figure 7.2 "Workplace" OS/2 Warp for the Power PC Architecture.

agement, timer, semaphores, interrupt manager, interprocess communications, and the MVDM kernel. The subsystem layer contains all of the shared libraries including SOM, window management, graphics subsystem, WINOS2, the spooler, and the DOS API functions, which interface to the kernel and device drivers.

As you can see from the diagram in Figure 7.1, the kernel and device drivers are tightly coupled to the hardware and, since they are required to run in supervisor mode, are inherently difficult to port. This makes the implementation of the Intel version of OS/2 on other platforms a challenge.

The Workplace architecture, as shown in Figure 7.2, moves much of the function from the OS/2 kernel out into user tasks, similar to how the subsytems and shared libraries run. The architecture is based on the IBM Microkernel. The microkernel provides a minimum of operating system functions. Most of the other functions found in the Intel OS/2 kernel such as device drivers, MVDM kernel, semaphore handling, and memory management, are moved out into user-level tasks, as servers or shared libraries in Workplace.

These functions that were in the monolithic kernel are separated into a number of modules. Each provides a defined set of functions and cooperates with other modules using interprocess communications (IPC). Only one small module, the IBM Microkernel, runs at supervisor level and as such, is the only code that is hardware-dependent.

There really are only two layers to the Workplace architecture. Outside of the microkernel are all of the system services as well as applications running at user level. This protects the system from code written in applications, system services, as well as system extensions. The most important facet of this, however, is that it is all portable. By doing the work to port the microkernel to a new hardware platform, and providing the same microkernel services on that new platform, the rest of the system, including device drivers, and most importantly for you, applications, port with virtually no work other than a recompile.

MICROKERNEL

The microkernel in OS/2 is the IBM Microkernel. This is a derivative of the Carnegie Mellon University Mach 3 microkernel. In contrast to the Intel OS/2 kernel, a microkernel provides only a small set of functions

that must be performed in supervisor mode, and other operating system functions are delegated to one or more user-level tasks.

The IBM Microkernel performs the tasks of Interprocess Communication, first-level interrupt handling (the remainder of interrupt handling is delegated to a user-level task), thread and task management, basic I/O services, and management of the processor (see Figure 7.3).

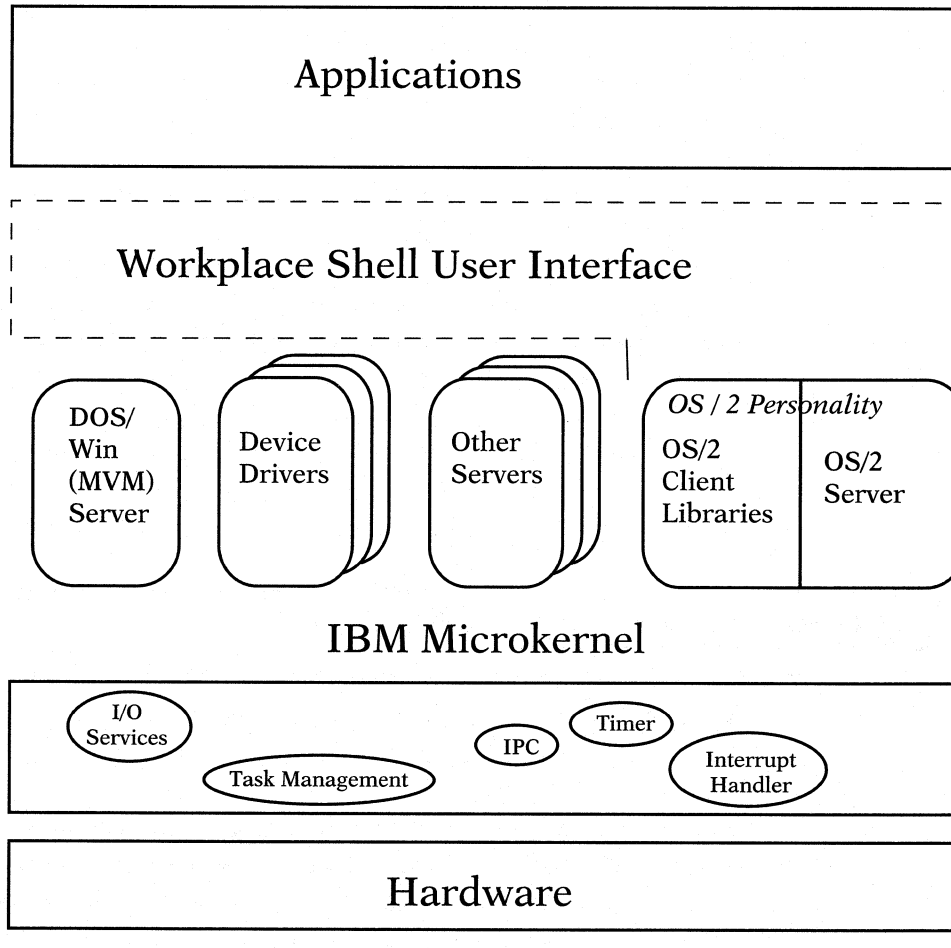


Figure 7.3 OS/2 Warp for the PowerPC with detailed Microkernel view.

The most important function the microkernel provides is its communications facilities. The modules that provide all of the other traditional kernel services that are now in separate modules communicate with the microkernel and other user-level tasks through microkernel messages. Each of these modules provides a defined set of functions and since they are user-level tasks (think of Ring 3 in Intel terminology) they can only communicate via an architected IPC mechanism. In this case, it is microkernel messages.

The IBM Microkernel controls communications through ports, and each port has a set of access rights. These access rights (such as read privilege and write privilege) are granted to tasks for their communication jobs. Notice how this is moving towards an object-oriented system (in addition to just the o-o shell) by having server tasks communicate through messages, where direct manipulation of data is only performed by the task that owns that data.

Looking at Figure 7.2, you can see that the IBM Microkernel manages the hardware, and the remainder of the operating system is a client-server architecture. The operating system functions are performed by a set of server tasks, and the applications use client libraries to request system services from these servers to perform functions.

Using this architecture, the microkernel is truly micro, and the bulk of system services are performed by the client-server architecture using the microkernel to communicate and manage several key services.

PERSONALITY-NEUTRAL SERVICES

As you've probably noticed, many of the names you see in the Workplace architecture seem confusing because many things refer to the same structures.

What you have been seeing as the server tasks can also be referred to as *personality-neutral* services. In this client-server model, the server runs as a separate task, and the client libraries run in the application's task space. The analogy of these client libraries are the subsystems de-

scribed in the Intel architecture. Applications that require these system services make calls into the OS/2 subsystems—in other words, the client libraries. This code runs in the application address space.

Requests for operating system services that do not require data or function known only to a server task can be satisfied directly by the client libraries. More complex requests and those that require data known only to a server task must be satisfied by the server. This request is communicated to the server in the form of a microkernel message. Note that since the server is a separate task, requests must be made through messages, while requests that will be performed by the libraries can be executed directly. (Note that since all requests from your applications go through the API layer into the client libraries, there is generally no need for your application to code to the microkernel message interface.) It is also possible that a single request from an application will cause multiple requests to be made of several server libraries or that a single function may be split between client and server libraries. Through all of this, the application does not need to know what is going on underneath; it just calls system sevice APIs.

This client server model increases system stability by encapsulating data for which protection is required into the server task. The model also provides the ability to run clients and servers remotely, leaving the architecture and system open to future expansion, and replacement of servers (either for maintenance or upgrades) independent of the rest of the system.

The servers operate to provide services, regardless of the personality, or the face shown to the user and applications. Yet another term for these personality-neutral services are *common and shared* services. While they are not bound to any personality, allowing them to operate in the absence of a specific personality, they also cannot use the services of any personality.

One example of a personality-neutral server is the file server. It provides the critical file system functions of opening, closing, reading from, and writing to files. The file server provides these services to any operating system personality and because it is behind (or underneath)

the personality and APIs, this reengineering of the operating system is transparent to your applications.

Some other examples of personality-neutral servers are:

- Master Server
- Event/Session Manager
- Name Server
- Registry
- Default Pager
- LAN Protocol Stacks

OUT-OF-KERNEL DEVICE DRIVERS

Another example of moving traditional kernel services out of supervisor mode and into user address spaces is *device drivers*. One of the biggest inhibitors to new operating system acceptance in the marketplace is the initial lack of device drivers. They are traditionally written in assembler, directed at the target hardware platform and as such, are very expensive to write and maintain, and reuse is almost nonexistent.

By taking device drivers out of supervisor mode, and making them personality-neutral, device driver authors are now not only free from tying their code to specific hardware platforms, but the device driver can also be used across personalities. For example, a single device driver can be used by OS/2, DOS, Windows, AIX, or any other applications running on a Workplace-architected system. In addition, because the device driver is not written to run in supervisor mode and is written to talk to hardware only through the microkernel, a port of the microkernel to another hardware platform means that the device driver can run on that platform as well (recompiled for the new platform, of course).

The IBM Microkernel grants rights to specific memory resources to these new user-level device drivers. A small piece of the device driver is put into the microkernel as an interrupt handler, which performs a very restricted set of functions. The majority of the function of the device

driver in a Workplace system runs as a separate task, so the rest of the system is protected, and everyone gets the benefits of a portable device driver. What a concept.

PERSONALITIES

Fundamentally, a personality is the inteface to the user and the applications. The Workplace architecture provides for multiple personalities, with one being the dominant personality. You can think of the dominant personality as the one the user sees. You can also think of different personalities as different API sets, supporting different applications. For example, the OS/2 personality provides the ability to run OS/2 programs. The MVM personality provides `INT21` functions and the ability to run DOS and Windows programs.

Aside from the API set, it is up to the individual personalities to provide the policies that govern resource allocation and management. For example, the OS/2 personality determines the scheduling policy for OS/2 tasks, how resource such as memory addressability and handles are managed and inherited for child tasks as well as semaphore coordination, and other OS/2-specific function. Structurally, a personality is a set of servers and shared, or client libraries. Some servers as you have just seen are personality-neutral, and others are specific and are the implementation of a personality.

In OS/2 for the PowerPC, the OS/2 personality is dominant, and the Workplace Shell is the user interface. This dominant personality, aside from running OS/2 applications, handles the system functions of startup and shutdown, as well as the policy for system exceptions and recoveries.

The Workplace architecture provides the ability to run multiple personalities concurrently, so the user can have OS/2, DOS, Windows, and other programs all running side by side on the same machine. All you need to know is that if the appropriate personality is present, your application will run, and not know, nor need to know the specifics about what is underneath the APIs it is calling.

THE OS/2 PERSONALITY

Now that you've seen the overall architectural differences between Intel OS/2 and Workplace, let's look into the actual OS/2 personality, which is the first implementation of the Workplace Architecture.

One set of assumptions you can make already is that the IBM Microkernel is there, as are some sets of personality-neutral services, such as a master file server, name server, device drivers, event/session manager (note that this is a service common to all personalities now, in contrast to Presentation Manager managing it through the session switching and singe queue), default pager for memory management and the registry, among others.

The OS/2 personality consists of the Workplace Shell (since that is its user interface and in OS/2 for the PowerPC, the OS/2 personality is dominant), and the OS/2 system services frameworks (in other words, the OS/2 scheduling and session policies, the critical system functions the dominant personality performs such as system initialization and the OS/2 API).

As you might imagine, after seeing the previous client/server discussion, the OS/2 personality is comprised of an OS/2 Personality Server and a set of client libraries. These client libraries are analogous to the OS/2 system services DLLs from the Intel version. For example, there is the OS/2 client library (the `DOSxxx` API calls), the Presentation Manager client library (including the `Winxxx` and `GPIxxx` APIs), and others such as SOM. These client libraries remain essentially unchanged in OS/2 for the PowerPC.

OS/2 Client Library

The OS/2 Server provides a client library that exports the functions of the `DOSxxx` APIs. The majority of the work in these APIs cannot be handled in the client library alone and thus the work is accomplished

by communicating with various servers in the system. All of the 32-bit APIs are provided by the client library, so applications are source code compatible.

Some examples of functions provided by the OS/2 Client Library and how they are accomplished include:

- Memory Management—Large memory object allocation is done by placing requests with the OS/2 server. Memory object suballocation, however, is done right within the client library.
- Tasking and Scheduling—These functions are handled by the OS/2 Server at the request of the client library.
- File functions—File I/O is performed by the File Server. Since the OS/2 Server needs to keep track of file handles being used by OS/2 tasks, this work is accomplished by a mutual cooperation and coordination between the File Server and the OS/2 Server.
- Device I/O—This work is handled by the device drivers, but since they are detached from any specific personality, the device I/O requests are communicated to the device drivers by the OS/2 server at the request of the functions in the client library.

As you can see, much of the work of the OS/2 client library is setting up requests to be performed by the OS/2 Server, which may in turn need to work with other servers. You may say that this is a performance problem, but given the speed of the hardware and the efficiency of microkernel communications, this is not a problem. The layers provide the independence from the hardware you need to elegantly and easily port applications and the operating system.

Presentation Manager and Other Libraries

Much of the work in the PM libraries is done in the library itself. For example, window management and drawing calls (`WIN` and `GPI`) are handled in the client library, and hence, right in the user process since the client libraries run in the application's address space. Critical data is

protected in the server tasks, but since much of the client code runs right in the application task space, these functions can be much faster.

Some of the PM functions are accomplished by client calls into the various server tasks. An example of this is the PM input mechanism. Changing this would break applications. But, since all system input is handled by the event/session manager, PM input is accomplished by requesting the services of the event/session server task. This is how the architectural problem of a single input queue in Intel OS/2 is overcome in Workplace without breaking applications.

An example of another library in the OS/2 personality is SOM. Much of the work done in the SOM client library (read DLL) is done in the library itself. These libraries together along with the OS/2 server make up the OS/2 Personality.

The OS/2 Server

The OS/2 Server is the cornerstone of the OS/2 Personality and is the primary server used by the OS/2 client library (and the other OS/2 personality libraries). The server runs as a separate user-level task and is started as part of the system startup process. In some cases, the OS/2 Server performs function. In other cases, however, the OS/2 Server does not perform function, but sets policy for the microkernel and other servers to implement and enforce. A good example of this is *task scheduling*. The OS/2 Server sets the task scheduling policy according to the OS/2 scheduling policy, but the job of task scheduling and management is a microkernel function. The OS/2 Server sets task priorities and the mode in which the scheduler runs, and the microkernel implements this.

There is a fine line between what the server does and what the microkernel does. Some examples are task and thread management, which you just saw. The OS/2 Server sets the policy, and handles the creation, suspension, and termination of tasks and threads in OS/2 for the PowerPC, but the actual scheduling and switching is done by the code that manages the processor(s), the IBM Microkernel.

Another example is session switching. The OS/2 Server controls the creation of sessions and screen groups but it does so by calling the Event/Session Manager. Memory is another fine example. The OS/2 Server defines how the address space is set up, but memory is allocated from the system end reserved by asking the microkernel to do it.

This is not a trivial architecture nor is the function. However, you must remember that to you, the application programmer and designer, this is transparent. All you need to know is to use the high-level OS/2 API and the system does the rest. You need not have to know where the function is performed, whether it is in a server task or a client library.

Another good demonstration of where you need not know about the underpinnings of the system is for DOS applications and the DOS Personality. The DOS Personality is designed to be only an alternate personality (it can't be dominant) and it provides all of the DOS emulation functions to enable DOS applications to run unchanged. The only reason you need to recompile 32-bit OS/2 programs for OS/2 for the PowerPC is because the chips have a real-mode Intel instruction set execution capability. Note that all you have to do is *recompile* your 32-bit OS/2 programs and you have OS/2 for the PowerPC native programs.

SINGLE-SOURCE APPLICATIONS

As has been inferred several times thus far, your applications for Workplace-architected OS/2 systems such as OS/2 for the PowerPC are source code compatible with the Intel-based OS/2. The API set is consistent, and like any operating system you may have enhancement from release to release adding new APIs, but as long as you stick to that API set, your applications can be moved to new hardware platforms with not much more than a recompile (and maybe some `MAKE` file changes for different compilers and tools).

This is in sharp contrast to other operating systems in which there are varying flavors of the system API, requiring more than just IFDEFing

code for different execution environments—in many cases, rearchitecture. For example, if you look at the Microsoft Windows API, you will see `WIN32`, `WIN32C`, `WIN32S`, among others. Some of the subsets of functions are close enough that you can code around, but why should you have to? Others, such as the difference between `WIN32S` and the others have fundamental differences in architecture that require a full redesign in order to properly exploit the functionality offered in each. The case in this point is the lack of threads in `WIN32S`.

Using the consistent OS/2 API that is implemented and supported across the variety of OS/2 platforms, you can generate huge markets for your 32-bit applications with only the investment of good design.

Design and code your applications according to the theories and recommendations you see in this text, and you will find yourself portable to any OS/2 platform, and with some API mapping, even to the `WIN32x` flavors.

SUMMARY

OS/2 for the PowerPC will run applications written for the 32-bit OS/2 API. The system itself has been designed to be portable by changing how the underlying functions work, but not what they do with respect to applications.

The Workplace architecture is an advancement in operating system technology, allowing a powerful system that exploits the features of hardware to be more easily ported to other platforms, as well as enhanced, scaled, and maintained.

From an OS/2 32-bit application's view, there is nothing different between an Intel-based OS/2 native application and one written (more precisely, just recompiled) for the OS/2 Personality on a Workplace-architected system. The fact that the plumbing is different is hidden and it is that new plumbing that allows the portability of the system.

With the IBM Microkernel and the surrounding servers and user-level tasks such as device drivers, your applications can now run on a wider variety of hardware with a minimum amount of work from you. You can concentrate on making your applications fast, flexible, and powerful, and let IBM worry about where the operating system will run. You know that with not much more than a recompile, your application market will grow as IBM moves this system to new hardware platforms.

CHAPTER 8

Features for Your Application

An often neglected step of application design is to decide the purpose and features of the application at the outset. I have often seen designers who had an idea of what they wanted their application to do, but did not bother to decide what features would be in it, how it would communicate with other applications, or how it would present itself.

OS/2 provides many features to applications, as the previous chapters have shown. Before any coding or prototyping can begin, and even before any kind of design can begin, the features of the application must be laid out.

Many times someone will set out to write a spreadsheet, word processor, or communications program and not fully set down the most fundamental ideas. They will begin designing or coding and, later on, will discover things about the system they would like to utilize. Unfortunately, they have already done so much work that they would have to

trash a large amount of it to build in this new function. Just a little work beforehand would have avoided this.

The first order of business is to decide on the main objective of the application. I have seen many applications that looked like they started out to be one thing but had so many convoluted twists and turns that you could not tell their real purpose.

WHAT IS THE MAIN FUNCTION OR OBJECTIVE OF THE APPLICATION?

Applications sometimes get lost in the myriad functions they are surrounded with. I have seen text editors that had so many extraneous functions that the main text-editing function was terrible. The program sure was fun to play with, but as a text editor it was terrible.

Throughout the design and coding of your applications, it is vital always to keep the main objective of the application in mind and not hide it behind a load of snazzy toys and utilities. The surrounding utilities and other functions must complement the application—not obscure it.

Deciding what the application will be is not quite as simple as it sounds. Before embarking on an application, you need to analyze your target environment, not only in the types of applications it supports, but also its size and complexity, along with the market the environment represents. If the application requires extensive processing and long-running tasks, a multithreaded, multiprocessing environment is very useful. If the program is a text editor, for example, you will want to ensure that you can write the code so that it can run on as many varied platforms as possible.

These factors will weigh in your decisions of what features to include and how to structure the code. You may not wish to provide a graphical, WYSIWYG (what you see is what you get) interface if you intend to support this program under native DOS; you may wish to provide a user-tailorable user interface in which the user can choose a text-based interface even for OS/2, with an option for a graphical one.

Throughout this entire process, you must be sure to keep focused on what services the application will provide and how it will be presented to take the most advantage of available features, yet remain as portable as possible. The other key is to remember to keep the tools and advanced features in perspective and not to overpower the application with them.

HOW WILL THE APPLICATION BE PRESENTED?

OS/2 provides many ways to present applications. In some cases, the functions the application must provide will dictate the presentation. In most cases, however, you have the choice of several ways.

There is the obvious—the full-blown Workplace Shell-aware, Presentation Manager graphical application. There's also the full-screen text-based Video Input/Output (VIO) application. You also have the option of creating a hybrid, running your program in a text window under PM, getting some of the features of both worlds.

Aside from determining how the application will look, the presentation method determines functions you may use. There is a set of functions that are available in PM or text-based programs. There are some functions that may *not* be used in the PM session at all, such as device monitors. There are still others that can be used *only* in PM programs. These functions are also a determining factor in how you will present your applications.

Presentation Manager Graphics/Text

The most favored method of presenting applications is via the Presentation Manager graphical interface. This method provides applications the most advanced, easy-to-use interface. The PM interface provides the device independence for display text and graphics as well as printed text and graphics.

Presentation Manager alone lends itself well to object-oriented applications. It is a complicated task to manage objects and other aspects of a graphical or visual interface in a text-based session.

Along with the PM interface and API is the Workplace Shell. The Workplace Shell adds a new dimension to graphical applications. It is the first system to seamlessly integrate object-oriented technologies into the shell as well as supply these objects for use by all applications. This leads to easy-to-learn, intuitive interfaces for your applications. The system-defined classes, along with your derivative subclasses, provide consistency between applications alongside the flexibility to create your own custom objects that interface with the rest of the shell.

The PM application is presented in a PM window, usually with a menu or action bar and title bar, and sometimes with scroll bars to allow users to scroll the visible area. Figure 8.1 shows a typical PM window.

Although the PM window has a standard appearance, you can modify it as you wish, removing some controls or windows and adding oth-

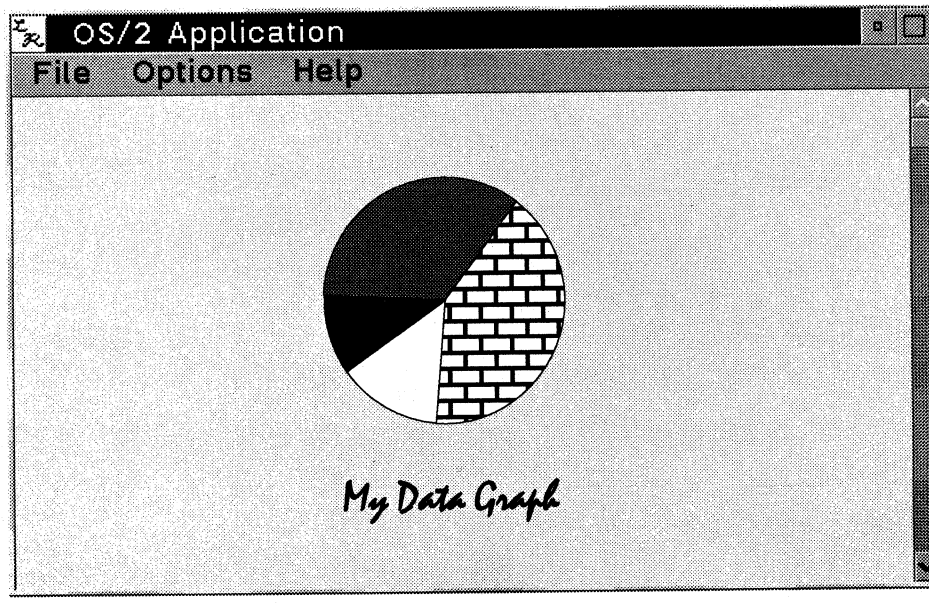


Figure 8.1 Presentation Manager window.

ers as you see fit. The main area, however, called the *client area* of the window, is where the action happens.

The client area is where you present your data, whether it be graphical in nature, text, or a combination of the two. PM is the ideal environment for combining text and graphics in the same presentation area of the screen.

The main reason for this is that everything in these PM windows is going through the graphics engine. As such, text in PM windows is simply treated as a set of graphic objects, painted just as a circle in the window would be painted.

This does present one drawback, however. Despite the flexibility and power to render text in a variety of fonts along with graphics easily in the windows, the drawback is performance. Although the performance of fonts in PM windows is quite good, it is definitely going to be somewhat slower than those applications that use hardware text modes. Characters have to be rendered as graphical objects, which is inherently inferior to using the hardware character modes in terms of speed.

What you lose in speed—which, I must reiterate, is not much—you more than make up for in flexibility. Since these characters are graphical objects, they can be manipulated in the same way as other graphical objects. Characters can be skewed, rotated, filled, hollowed, or even broken apart into their component vectors. Hardware character mode use is very restricted.

Should you present your application on a PM window versus another presentation method? The answer depends on how you intend to present data and interface with the user. If you feel the functions of your application are enhanced by graphics, fonts, and easy interaction with another program, then the PM window is for you.

You may ask, "Why use anything else if I get all this power with PM windows?" The reason is that PM provides some functions that other modes cannot, but other base functions, such as interprocess communications, are available in any OS/2 program, regardless of the presentation method.

There are really only two drawbacks to using the PM interface. The first is that programming the PM interface is radically different from

programming text-mode programs. There are many tools, which will be discussed later, that aid in coding the PM interface of your programs so that drawback is negligible.

The other drawback is that there are some programs that are just not well suited to the PM interface. Such an application may be a terminal emulator. First, there is really not much use in having a terminal emulator that can use scalable fonts or WYSIWYG graphics. Next, since speed of data is of the essence in data communications, especially over a timing-sensitive modem, any delay in rendering the data on the screen can cause data to be corrupted or lost. It does not seem very practical to write a terminal emulator to allow users to communicate on a BBS with a 30-point script font. Programs such as these are better suited to using the hardware text modes of the display. Fortunately, you do not have to abandon the power of PM just because you want to write an application that does not take advantage of the graphical nature of PM windows. There is another type of window called an AVIO (Advanced VIO) window.

AVIO Windows

An AVIO window is so called because, rather than a PM presentation space, it uses something called an AVIO presentation space. An AVIO window is a cross between a PM window and a text window. The main difference between the PM window and the text-based window is that a program running in a text window does not know it is running in a window. For all it cares, it is running in a full-screen session, using standard I/O calls for input and output. OS/2 has a layer called the VIO shield that makes that possible.

AVIO windows, on the other hand, look like text windows but are really PM windows. To be more precise, the real difference between the PM window and the AVIO is simply the presentation space used. A PM window, obviously, uses a PM presentation space and is afforded the functionality of all the PM calls, such as graphics, fonts, and text.

An AVIO window uses an AVIO presentation space. This means that the user interface of the application is very similar to the PM window

in that it has a menu or action bar, scroll bars, and so forth, and the application can manipulate all of these controls. The difference is how the data is presented in the client area of the window. In an AVIO presentation space, the application uses the standard text display output (VIO) calls, such as `VioWrtTTY`, to display data on the screen or, more precisely, in the window.

AVIO windows have all the functionality of PM windows in terms of data exchange and window manipulation. The only things they cannot do are PM graphics and font functions. They are also limited to fixed, bit-mapped, nonscalable fonts. The main advantage to an AVIO window is that you can take advantage of the hardware text modes' speed while not losing the power of PM's event-driven architecture.

AVIO windows also use the PM programming model; as such, the PM-based CASE tools will work for AVIO windows just as well as for full-blown PM windows. The AVIO window combines the best of both worlds for a great deal of flexibility in your choices of how to best use the PM environment to fill your needs.

Text-Based

Text-based applications are usually run full-screen. While you are able to run most programs designed for full-screen sessions in a text window, they were generally designed to use a text-based character mode interface similar to that of DOS.

Such programs do not enjoy the graphical functions of PM. Additionally, users must switch out of the Presentation Manager desktop to use these programs. The device independence and the ability of viewing several programs at once are also not available running in a full-screen session.

Why would one want to code for text mode? There is the obvious advantage that you do not have to recode a new user interface. By sticking with pure text mode, you can use the same user interface you used in DOS, while enjoying the multithreading, large memory, and data exchange features OS/2 provides. This allows you to get your application to market quicker, while still taking advantage of some of OS/2's most powerful features.

In this instance, text mode is just a stepping stone. In other cases, text mode may be just what you want. For example, if you are writing a command interpreter or programming utilities or any other program that is not long-term user-interactive, text mode may be better suited to your purposes.

Most text-mode programs can also be run in text windows (called VIO windows). Programs that use a set of restricted functions, such as those looking to gain access to the physical video buffer or those that register device monitors, will not run in VIO windows. The reason is that these few functions cannot be supported while the Presentation Manager is in the foreground. Most of the text mode programs can be run in VIO windows, however.

These VIO windows, as the name implies, run in the Presentation Manager session, using the VIO shield to act as though they are running in a full-screen session, although they are in a window. The VIO shield handles all the window management such as sizing and movement, so the application need not even be aware it is running in a window.

In contrast to the AVIO window, the program running in a VIO window has no knowledge of PM or its controls and cannot make use of PM functions such as scroll bars and DDE. The text window does, however, serve a very useful function.

A perfect example of the utility of a text window is in the programming environment. It is very often useful to allow the user or programmer in this case to watch all phases of a project going on. In one text window a compile could be going on, while the programmer can be editing other files in another window and running some source code search programs in another. All these programs do not need the function of fonts and graphics, yet it is extremely useful to see the output of all of them at once.

The other major advantage of the text-mode window is to speed the release of an OS/2 application. In fact, the first release of OS/2 itself did not have the Presentation Manager. It did support virtual memory, threads, and so on, as you can do with the early versions of your applications if you so desire.

Workplace Shell Objects and Templates

As mentioned previously, the Workplace Shell is one of the most interesting and powerful features of OS/2. There are many things to consider when deciding how your applications will interact with the shell.

Let's first consider the shell and the functions it serves and provides. A shell is fundamentally a program launcher. The original version of DOS did not have a shell *per se*, but it had the command prompt, which was used to launch programs. In the most rudimentary sense, DOS's shell was a simple command prompt, which was cumbersome and had limited function.

Software vendors then designed shells, starting from the menu-driven, such as in later versions of DOS, to some graphical shells, such as Microsoft Windows, developed later on. As time progressed, shells became more and more powerful. Usually, however, applications did not have to do much to work with the shell. These shells began to be packaged with utility programs to manage the shell and configure the system and the programs the shell managed.

OS/2's Workplace Shell (WPS) is different. Of course, any applications that OS/2 runs can be launched by the Workplace Shell. The shell comes with a set of utilities that allow the user to configure both the shell and the operating system. The biggest difference between the Workplace Shell and other shells for PCs is that the Workplace Shell works with the applications and your applications become part of the shell and the integrated OS/2 environment.

The Workplace Shell is an object-oriented system that provides functions to applications so that applications become part of the computer system and are not just programs. The WPS works with objects, such as programs, documents, devices, and so on. The WPS presents objects to users the way they are used to seeing them. It works the way people work.

For example, a user can take a Lotus 1-2-3 spreadsheet object, open it up, and have it presented just the way the user expects: within a Lotus 1-2-3 window. There are several ways to have your applications work with the shell.

There are advantages and drawbacks to each method of shell interaction. Usually, the trade-off is how much work you want the program to do versus how much work you want the end users to do in setting up their system. Of course, you want your application to be easy to configure, but the flexibility was designed into the shell to allow users to continue using their old applications yet still take advantage of the object-oriented features of the system.

At its most basic functional level, the shell supports object associations in which each document, or data file, from your application is represented by a workplace object. This object can have associations so that when the object is opened (when the user double-clicks on it), the application in the association will be started, and the data file the object represents, loaded.

The only requirement here is that the application must be able to accept a command-line parameter. The shell accomplishes this function by starting the application listed as the default association, passing the data file name as a parameter. This allows users to set up associations and use their existing applications in an object-oriented manner.

The other way to work with the shell is to have a Workplace Shell aware application. This is an application that uses the functions provided by the shell to create templates and new objects. Applications have the option of registering instances of new objects, and creating their own object classes. These classes are used to allow the user to interact with the application via the objects. No longer will users open a program, feed it a data file, and at some later time ask the application to spit out the file. With application-defined workplace objects, a user simply needs to open the object. The object is responsible for initializing the program.

Another interesting feature of using workplace objects in your applications is to allow a user to drag a data file to the printer. Without custom workplace objects, the only way to allow a user to print a file via drag/drop is either to have the file be in a printer-specific format or to have the file be a plain text file.

Many applications store their data files in a format, specific to the application, that is neither printer-specific nor plain text. This is often

necessary for application performance, but it should not have to affect the user's ability to print files in an object-oriented manner.

By defining the workplace object to represent the data file, for example, a user can drag the file to the printer and drop it there. Since the object defines the behavior, this custom workplace object can launch the program it belongs to and ask the program to print it.

It is not difficult to write an object. As a matter of fact, it is quite easy. Using object-oriented constructs such as inheritance, all you really need to do is subclass the base class you want and supply your own deviations to it. In this manner, you pick up all of the basic behaviors for the object and simply add or override the methods you want to be unique.

This only scratches the surface of the power you hold in your hands. Of course, the choice of whether to write objects to go along with your application is entirely up to you. Writing a custom object is your choice. You can gain some function out of the shell without it, but the more you add, the easier your application is to use.

Presenting your application using the Workplace Shell adds new dimensions in function and usability. You can get away with using the base function and knowledge of the parameter-passing function if you choose to not write your own object, or you can choose to go all the way and use the shell to its full potential.

CHOOSING FEATURES TO INCLUDE

Once you have decided how to present your application, the next decisions are which features of the many provided by OS/2 to use. The choice of features is dependent on the presentation method of the application. For example, it is pretty useless to try to use the clipboard if your application is full-screen.

OS/2 provides several methods of data communications and other functions. You may ask why are there so many redundancies. The reason is that there are several ways to present applications, and the OS/2 designers wanted to afford all applications the same function. Additionally, each method has certain advantages and disadvantages, and the

choice allows you to choose and balance out the function and overhead to get an optimal result in your application.

The most important choices are how your application will communicate with others, how it will print data, and, along those lines, how it will interact with the shell.

Data Communications with Other Programs (Open Application)

Multiple threads and virtual memory are features of OS/2 that are more structural than optional features. Data communications is one of the optional features you can add to your applications to add utility to whatever the main function of the program is.

Many programs perform a function. It is often useful to have a program that performs a specific function rather then being an all-purpose program. These specific programs are written to make the most out of one type of work, such as graphics. It is important that these programs be able to share data with others.

For example, let's look at a desktop publishing system. The main functions needed are graphics composition, scanning, text processing, and page layout. You may be able to find one program that does all of this, but it seems better to find several pieces of software that do each job better. It is like buying stereo components rather than an integrated system. The key is that the components need to be able to interface with each other.

Under DOS, "data communications" meant to export a file from one application, stop that application, start another, and import the file. OS/2 provides real-time data transfer while the various programs are running simultaneously.

Dynamic Data Exchange

Dynamic Data Exchange (DDE) is about the most powerful of the data communications methods. DDE is available to PM applications in two flavors: The first is a one-time exchange of data, the other is a hot link. Each is requested or initiated by the user and each requires both the

receiving (client) and sending (server) application to know about the data protocol.

Usually, the user will mark an area in the server application and request a DDE exchange. The difference between a hot link and a one-time exchange is that on the one-time exchange, the data is transferred, and that is the end of that communication session. The hot link, on the other hand, is a continuous exchange of data and notifications. If the area in the server changes, it is communicated to the client with no further action required on the part of the user.

As you can see, this can be used to link applications so that any change in one will cause a change in the other. An example of this would be a spreadsheet linking to a graphing program. The graphing program could be showing the data in, say, a pie chart. As the user changes data in the spreadsheet, the pie chart will reflect it.

There are several pieces of work that an application has to do in order to support DDE. The most important one is that the interface must be published. DDE is not only a set of APIs, but a protocol of messages and data. Applications wishing to do DDE with another send out a broadcast message asking if there are any applications wishing to do DDE on a specific topic.

A recipient of this broadcast must respond properly, or the exchange cannot take place. Because there is no universal standard for the data contained in the messages, applications must be specifically designed to accept certain data in certain formats. If you are going to design your application to do DDE with others, it is important to publish your DDE protocol, and it is equally important to obtain DDE protocols of applications you wish to interface with.

DDE is usually user-initiated, so it is also important to make it as easy as possible for the user to initiate the exchange. One method is to provide menu choices indicating DDE, and possibly a submenu indicating a hot link versus a one-time exchange. You could set up an object-oriented exchange in which the user marks an area and then drags the marked area to the other application. A dialog box could pop up as a result, asking the user if it is to be a hot link or a static exchange. A variation on that could simply be a key held down with the button for a hot link and a different key with the drag for the static exchange.

Another example of the power of OS/2 DDE function is that OS/2 applications can communicate with applications running under WINOS2. You may decide to have your program communicate with WINOS2 applications. In this case, you will need to follow the established protocol for those applications. Each of these decisions needs to be made before you begin work on the application.

So, DDE is a user-initiated data exchange that has the ability to be a one-time exchange or an ongoing conversation, with a predefined communication protocol and synchronization and free-form data format. You also have the ability to communicate with WINOS2 programs. DDE function is restricted to graphical applications only. Keep these attributes in mind when looking at the next few types of IPC. By looking at all of your choices in this manner, you will be in an excellent position to make a call on which ones fit best in your environment.

As you can see, DDE is very powerful. However, it is not a simple function to program, and there are many considerations, such as how many other applications know how to DDE and, more importantly, how many publish their DDE interfaces (data formats).

Clipboard

The OS/2 clipboard is exactly that. It is a virtual area in the system where applications can place data (clip it into the clipboard). Applications can also retrieve data from the clipboard. There are several operations available with the clipboard: cut, copy, and paste.

The first two place data into the clipboard in two different ways. A *cut* will place the data in the clipboard and remove it from the source application. A *copy* will place the data in the clipboard, but leave it in the source application intact.

A *paste* operation takes the data from the clipboard and places it into the target application. There are several standard data types in the OS/2 clipboard.

There is the text data type, the metafile data type used to exchange graphical data, and then the clipboard has a construct to allow an application to place an application-defined data type into it. The application-

defined type is available to let applications put nonstandard data into the clipboard.

As with DDE, any application that will pull data from the clipboard must understand the data type resident in the clipboard. However, unlike DDE, there are several data types that are predefined, and the extraction from the clipboard is a single API, not a message-based protocol.

Of course, the clipboard is only a static exchange method and is not quite as powerful as DDE, but it is easier to use and, at the time of this writing, is more widely supported by applications. The clipboard is also only available for PM applications, however a user can share clipboard data with applications running under WINOS2 as well.

You may notice that it is possible to use the clipboard with VIO windows too. The reason is that VIO windows are managed by the VIO shield, and one of the functions provided by the shield is rudimentary support for the clipboard. Text applications running in VIO windows cannot use the clipboard on their own, but the VIO shield provides enough support that you can copy and paste data from/to the clipboard.

Looking at the overall attributes of the clipboard now, you can see that it is a user-initiated, one-time exchange of data, with some predefined data formats (but you can choose your own as well) and no real communication protocol needed. You can also exchange data with WINOS2 programs. Like DDE, the clipboard is restricted to graphical applications, although you can do some work through the VIO shield. Now let's look at the rawest form of IPC in OS/2—pipes and queues.

Pipes and Queues

Pipes and queues are available to all applications, not just graphical applications. These are base system constructs that allow applications to exchange data between processes in a raw form. They are exactly what they sound like: pipes and queues.

Just as water flows through a pipe, so does data through an OS/2 pipe. A queue can be defined by the applications as they wish. Both pipes and queues are first-in-first-out structures. Both need to have access synchronized with some construct, such as a semaphore.

The other major consideration in the use of pipes and queues is that both the client and server processes must know about the data in the pipe or queue. There are no data format standards in using pipes and queues, so, just as in DDE, the applications must know about the data being retrieved. Also as in DDE, pipes and queues can be used to transfer data on a one-time basis, or a type of hot link.

Unlike DDE, however, there is no specific protocol and access to the structure is generally unrestricted, controlled only by synchronization semaphores. Of course, the writers and readers of the pipes and queues must know what the data and the structure are. It is the transfer of the data into the pipe or queue that is free-form.

Pipes and queues afford almost all of the functionality of the clipboard for DDE, but they are available to all applications, not just PM applications.

Pipes and queues are not usually directly user-initiated (some user action may cause you to open a queue or a pipe, but I've yet to see an application that has an "Open Pipe" menu option), the exchange is as you wish in terms of static or ongoing, the data is completely free-form and all coordination and synchronization is up to the programmer.

Looking at these key facts in the data communications services available to you, you might see that you want flexibility in free-form data, but you don't want to have to deal with semaphores. In this case, DDE would be a good choice. As with any of these IPC services, you can start DDE with or without explicit user action, but the choice is there.

Let's look at a concrete example. Say you are designing a program for a medical facility, where the task is to capture and display heart-rate information for analysis. (You will see this example used in several places in this book). For reasons that will be explained later, this application is implemented in several processes. One will gather the data from the medical equipment. Another will display that data. How should the data get passed from one to the other?

I think you'll agree that the clipboard is not a good choice. You'll probably want an ongoing data exchange and the clipboard can't do that. So the clipboard is out. How about DDE? It has free-form data, ongoing communication ability, a way for users to initiate and terminate it (via menu choices that you provide), and a defined communication

protocol with synchronization. Both processes need to know about the data format, but that would be the case with any IPC mechanism you use. Why would you want to use DDE as opposed to a pipe or queue?

DDE, pipes, and queues provide many of these same features, but I would choose DDE in this case for two reasons. First, DDE provides the communications protocol for me, as well as the synchronization in that protocol. Call me lazy, but if the system provides it for me, and I don't need anything else that it does not, I'd rather use it than start having to deal with semaphores, full and empty pipes and queues, and so on. There are plenty of reasons to use pipes and queues over DDE, but this is not one of them.

One might be if you have variable-sized data, or are passing different types of data through the same structure to multiple processes. In this example, I'd choose DDE. You can see how you need to analyze the functions that OS/2 offers to make the appropriate decisions for your applications, not only to avoid confusing your users, but also to make your application development faster and easier. Complicated is not always better.

Data Interchange Formats and Filters

Another aspect of communicating with other applications is not at runtime. It is often necessary to be able to work with data from other applications in their own file format. In these cases it is useful to be able to import and export data in other applications' formats.

There are several universal formats, such as encapsulated PostScript and TIFF, in addition to the formats specific to applications. In order to use the universal formats, you will sometimes need to write the file yourself. Such an example is the TIFF or CGM file format. For others, such as encapsulated PostScript (EPS), you may be able to take advantage of functions within OS/2. For example, you may be able to utilize the PostScript printer driver to have it generate a PostScript file that you can place in a file.

When it comes to application file formats, however, you are required to understand the format of the other applications and write filters for them. It is unreasonable to ask you to write filters for all applications.

However, if it is applicable for your type of program, you should probably consider at least writing an interface to a set of filters so that you can add filters as you feel necessary and as new applications hit the market.

Another possibility to note, if you are planning on importing files from other applications, is converting the file directly into your own internal file format as you read it in. This will give you much better performance, as you will not have to work with the file in some foreign format. Additionally, if the user has imported the file, chances are that the user will want to save it in your application's native format. This is a simple trick that will give you better performance.

REXX Hooks

Another often neglected feature of OS/2 is REXX, which is an interpreted language based on the IBM mainframe implementation. OS/2 comes with a REXX interpreter built in. This gives users the ability to write complex programs that are much more powerful than the batch language inherited from DOS.

REXX has constructs and programming structures that allow users and programmers alike to write powerful utilities with a simple text editor. REXX requires no compiler to work. When a .CMD file has a C-style comment as its first line (`/* This is a comment */`), it is not interpreted by the standard command interpreter when executed, but rather it is passed to the REXX interpreter.

Where does this fit in application design, however? It does not fit in every type of application, but some applications can either "shell out" to the system to perform some operations or actually run batch or `.CMD` files in order to do some work. A good example is a text editor or word processor that has a command line to perform tasks. The function of the editor is controlled by the menus and action bar, but there is a command line to execute commands. Some of these commands are internal to the editor. It is possible to call the shell and/or the REXX interpreter if the command is not internal to the program.

REXX can add a new dimension to your applications if the program lends itself to REXX's function. Of course, a graphical program such as a page layout or drawing program may not lend itself well to REXX hooks.

However, for those that do, REXX can add more power and flexibility to your programs.

Printing

I have seen many applications that perform their primary function superbly, but fall miserably short when it comes to printing. OS/2 provides a powerful print subsystem, but if misused it makes an application look sloppy.

There are several ways to print data. You can support drag-and-drop from the data file object, you can support drag-and-drop from within the application, and you can support printing from the menu or action bar.

In a general sense, you will want to have your application work with the OS/2 spooler. There are some applications that have been written in such a way that they cannot work if the spooler is running. These programs are usually written using restricted function calls, but in any event, they restrict the user's flexibility in what they can do with the computer.

Regardless of the method of printing, you should be aware of how to provide the users with the greatest amount of flexibility, balancing function, power, ease of use, and ease of coding. Print destinations are really the primary focus.

A print destination should be a queue, obtained via a set of APIs. You can print the data using the same code you used to display on the screen. The only difference is that you will be using a printer PS as opposed to a screen PS. There are also some extra functions you must execute, such as the `DevEsc` calls to begin the document, to separate pages, and to end the document, but the drawing of the text and graphics is the same.

The details of printing will be discussed further in Chapter 16.

Fonts and WYSIWYG

Font usage is another subjective topic. Many applications do not need the advanced capabilities of fonts, and others cannot afford the overhead.

Recall that the advanced font functions are available only in a Presentation Manager PS. If you are not using a PM PS, then your decisions become much easier. However, you can still utilize the font functions of the hardware in an AVIO PS.

If your application is using an AVIO PS, you may wish to query the display device for the fonts it has so that you can provide whatever fonts are available to the user.

If you are using a PM PS, then you have the choice of any font installed in the system. The fonts are stored on disk and are described in the `.INI` file and in the font files themselves. A font query function will return the information in the `.INI` file, and a subsequent call to get a particular font's information will get the data from the font file itself.

The decision to use PM, AVIO, or any fonts for that matter, depends on the purpose of your application. How you use the fonts is what matters. Font usage will be discussed in depth in Chapter 16.

SUMMARY

Too often, people will "overcode" and provide features simply because they are there, not because they add value to the program. All this does is confuse the user and make your application bigger and more difficult to use.

Choosing the right features is the first step in your design and choosing the most appropriate functions is paramount. Always think back to that medical application example and how to analyze the functions that OS/2 offers to see which is best for your needs. Don't get "drugged" by the fancy features of the operating environment—offer what will enhance your application. Don't complicate it by adding too much.

CHAPTER 9

Application Structure

The structure of your application determines performance, maintainability, and flexibility. It determines how well your applications work under OS/2, how easily they can be ported to other environments, and how well they take advantage of their native environments.

OS/2 is designed to allow applications to be free from having to know specifics about the computer hardware, to use *parallelism* and multitasking, and to share code and data. You need to decide how you will use these features to your advantage without compromising maintainability or defeating their purpose. You don't want to read the programming manuals and blindly begin coding functions.

If you do not set up your structure properly, you may wind up writing code that will cut off options later on. For example, if you start writing your code and then later decide to multithread that code, you will most likely have to rewrite the code. It is difficult to restructure existing code to add multithreading without introducing deadlock potential. Additionally, you will not be able achieve the full potential and power of multithreading due to the lack of parallel function design.

This chapter will discuss how to structure your code to take advantage of OS/2's features while remaining flexible. It is more than just writing modular code—it is a whole way of thinking about tasks within your applications.

ISOLATE FROM UNDERLYING HARDWARE

As computer hardware becomes more diversified and more vendors release new products, it becomes more and more important to isolate the application code from the hardware. It should be the job of the operating system to manage the hardware and provide services to application programs. OS/2 does just that.

OS/2 does provide the flexibility for those applications desiring to work with specific hardware to do so, but, except for special instances, you want to keep your code as isolated from the hardware as possible.

Every device in a computer system on which OS/2 is running has a device driver. As you saw in Chapter 5, the device driver has the job of synchronizing access to a physical device. More importantly, however, it completely manages the device. It accomplishes this through a set of functions or entry points callable through `IOCTL` packets.

Manipulating hardware through `IOCTL`s is effective; however, it makes your applications directly hardware-dependent. Code written to send `IOCTL` packets to device drivers will not be portable to other systems and, in some cases, will not be supported on some hardware. You see, not all device drivers support all functions or features.

The basic device drivers in OS/2 were written by IBM and, in some cases, by vendors with IBM guidance. As time goes on, however, more and more device drivers will be written by independent vendors. There is a core set of functions all device drivers must support, but many of the more advanced functions may not be supported. As such, if you write your code to interact with the device driver directly, you cannot be assured of all advanced functions being present.

You could, on the other hand, write to the lowest common denominator in terms of which functions you use. The drawback there is that

you limit your code to the slowest, most limited function, even though users may have sophisticated hardware in their system.

You will be happy to hear, however, that OS/2 for the PowerPC has been written to support the IOCTLs from the Intel code base. This applies, however, only to the device drivers supplied with OS/2. While others supplied by other vendors may work, this is not guaranteed. This is where IBM has done the work to preserve your investment, but you should not rely on it, because IBM does not write all the device drivers that are out in the marketplace. The key is to use the high-level APIs as much as possible and to let the operating system handle the "dirty work."

Use OS/2's Device Independence

Not only does OS/2 provide interfaces to device drivers directly, but these functions are also wrapped in API function calls. You can accomplish virtually any function you need through the OS/2 API.

The OS/2 API serves several purposes. First, it provides a higher-level interface to system functions so that applications need not know about the hardware present. Second, it provides a layer of function that, in addition to the synchronization of threads into devices provided by the device driver, aids in coordination and optimization of threads in each function.

Another role the API layer plays is to provide a level of portability. By coding to the API rather than to specific hardware through IOCTLs, the application is portable to any system OS/2 runs on, including OS/2 for the PowerPC.

One other point about the portability of the APIs is that there are other systems that use a similar set of functions and, although there may not be a one-to-one mapping between systems, there is such a similarity that you can convert your application easily. Many of the porting tools written to port applications to OS/2 function that way. For the most part, the OS/2 API is a superset of most other systems in terms of function, so porting to OS/2 using APIs is straightforward.

Device independence under OS/2 takes on another flavor as well. Recall the discussion on presentation spaces and PM device interaction with the Presentation Drivers and the graphics engine. Imagine that you are writing a graphics design or CAD program to run on a variety of hardware. Don't even think about the display at this point. Just think about printing the user's output, such as blueprints or electrical designs.

There are a variety of plotters and printers that are in use for such applications of CAD work (or any graphics work, for that matter). Imagine writing code that has to understand the capabilities and language of every plotter and printer. Not a pleasant prospect. You don't want hardware dependence here either.

By using the device independence of OS/2 and PM, you can address the various display and printer hardware without even differentiating between them. Through the OS/2 API you will query the system for the name and type of display attached and for the printer name the user has selected. The only hardware dependence you need to be concerned with is the page size of the device (which is queried through a simple API). The rest is handled through the engine.

Of course, as with any other function of OS/2, the program is able to generate its own device-specific data streams for control the Presentation Drivers may not provide. Again, the drawback with this type of code is that it is neither portable nor flexible. As new devices become available and data streams change and improve, the application would have to change to accommodate the new function. By using the Presentation Driver interface, applications do not have to take hardware into account. The generic presentation space architecture takes care of that low-level function.

By sticking with the high-level interfaces and structures, you can keep your code running on any hardware OS/2 runs on with little or no change.

Stick with Portable Languages and Tools

An important thing to consider in the development of an application is the tool and language selection. Not only does OS/2 support writing

code to isolate the application from the hardware, but it also supports high-level languages such as C, C++, SmallTalk, and, yes, even COBOL.

Of course, you can always choose to write in assembler for performance and control. Many people choose assembler because they just don't like the optimization or performance these high-level language compilers provide. The trade-off is portability, ease of coding, and the lack of high-level constructs.

You may have noticed that most of the examples I have been using and will continue to use will lean toward the C language. The reason is that although there are many companies with language compilers, C lends itself well to the demands placed on a language, such as recursive coding. Such languages as COBOL have their place in OS/2, but they usually need other tools to generate and manage windows. C does not have that problem. Although COBOL is a commonly known language, it has its limitations under OS/2.

By using a language such as C, you gain portability to other environments (just think of what it would take to port your assembler application to a PowerPC machine) not only for code you write for OS/2, but also for code you have written for other environments that you may want to port over to OS/2. Compiler optimization has improved dramatically over the last few years, and what little you may lose in performance and optimization, you will gain in programmer productivity and code reuse.

In addition to the portability of C, OS/2 is written mostly in C and its primary language bindings are C language bindings. The function-calling interface and parameter-passing conventions are based on C as well. Since many of the current programming languages base themselves on interfaces and parameter-passing conventions defined by C, C is the logical choice to use.

Many of the most popular languages and tools have a common ancestry in C. Many programming tools, such as `AWK`, `GREP`, and `MAKE`, also have histories in C. In addition to having common ancestry, these tools and utilities are widely used on many varying platforms.

By using the high-level languages and tools, along with such tools as `MAKE`, `GREP`, and so on, you can keep ports as simple as possible. OS/2 was written using these types of tools, so applications written with them will

be using the same calling conventions as the operating system, giving that much more performance where some may be lost as compared to, say, assembler.

MODULAR DESIGN AND THREADS

Of course, the most basic of concepts in programming is "divide and conquer." Ever since your first programming course you've probably been told to modularize your code and separate as much as possible into functions. You'll not be told any different here.

As a matter of fact, in this multithreaded environment, it is vital to write modularized code and to make these functions as reentrant as possible, so as to maximize a single set of code accessible to many threads. Modularized code is only the first step, however.

Separating code into functions is not a difficult task. What is crucial in determining how well your application takes advantage of multithreading and parallel processing is how well you modularize and structure your code to run as many tasks as possible in parallel.

The way to start is to isolate all tasks your application will provide. No task is too small, but don't subdivide too finely or you will have each line of code on its own thread. Task division should be at a level higher than the function but lower than "we create graphics." You will likely make mistakes and never get it exactly right, especially the first time. Just as in driving a car, every turn and every application is unique and there is no pat answer or method.

A good method of separating tasks for parallel processing is to look at the main purpose of the application and subdivide from there. Just as though you were making a flowchart, begin to subdivide your application from the time it is invoked to the time it terminates. Let's look at an example.

Most applications follow the same three basic tasks: Initialize, Process, Clean Up. These tasks can all be further subdivided to provide internal parallelism. There is even some degree of parallelism you can provide between these functions. Let's begin by looking at initialization.

You can look at initialization in several parts. Obviously, you will need to set up data structures and control blocks as well as to initialize any utility threads, such as an internal memory manager. There's really no way to multitask these items, simply because they must be set up in order for the application to be able to accomplish anything. Once the basic structures are set up, you can begin to see how much of the remainder of the initialization can be multithreaded.

Many applications need to load DLLs and resolve entry points to functions. Some allocate memory, read the environment (such as available printers or other relevant hardware) from the system, and set up menus or other user controls. This set of functions is ideally suited to allowing the user to execute actions while the rest of the program is still loading and initializing.

For example, when an application is launched, the first operation is usually to load in a data file. There's no reason this cannot happen while the printer information is being read and the related structures set up. Obviously, the user will not ask to have something printed if it is not yet read in, so why should you restrict the users by making them wait until the printer information is initialized before allowing them to open a data file? Once the code to read the file is loaded, the user should be allowed to execute that action.

You could, for example, build into the code that reads the printer data a function that activates the PRINT menu item. This way the menu item can be there from the start, just not active until the print code is loaded. Actually, in this case you are really overlapping the "process" part of the three-step application with the initialization piece.

In general, you should have your user interface thread up and running as soon as possible, even if initialization is not yet complete. Using the menu messages of PM, you can enable menu items as their support code is initialized.

Another application of multithreading structure is when working with multiple data files. For example, time-consuming processes, such as printing, should not inhibit the user from working with other parts of the application. Of course, while a document is printing, you should not allow the user to change it, but there is no reason the user cannot

load another data file or switch to a previously loaded one and begin working with it.

The real rule of thumb in deciding which pieces of the application to multitask is common sense. The way you do it, however, depends on the structure of your code. You will need to design how you will multithread the code at the beginning of your module design. Actually, a more precise term is *functional design*, because it is functions you will be designing to be run in parallel both within and between functions.

This topic goes directly back to the 60-30-10 rule. If you decide later on to add in multiple threads, be prepared to go back and redesign, otherwise you will be one of those who spend 140 percent of the time on the project.

Another important point of modularizing code, although not as involved as multithreading concerns, is modularizing for subsystem replacement. Even applications should be modeled after the subsystem approach to allow you the freedom to replace modules or subsystems as necessary.

As time goes on and technologies advance or, perish the thought, you find an error-prone component, you may want to rewrite a piece of your application. With cleanly defined interfaces and well-structured subsystems, you can replace parts of your application transparently to the end user through an upgrade package, a fix package, or even automatic bulletin board updates.

Modularized code is easier to maintain than code that just works. By modularizing your code you are forced to think about the multitasking and subsystem topics before you begin coding. You have seen how important it is to subdivide the tasks the program will perform logically to make the most use of multiple threads.

There are techniques to use these threads that will be covered later, but it is important to remember that you must draw a line on how much to multitask, because while threads do work in parallel, they also have overhead, so too much of a good thing is not necessarily better. The same goes for modularizing code. If you go too far, each line of code would be its own function.

Look at your tasks, break them into functional groups, taking into account which of these functions in the larger scheme of things will potentially run in parallel, but more importantly, which ones CANNOT run in parallel. Then take these functions and build them up into functional groupings. The reason you are doing this is to establish the sets of functions that can and cannot be executed concurrently at the outset, so you can begin to structure the code to avoid conflicts and deadlocks.If you follow these steps, you'll have code that, although it will likely not be perfect on the first go-around, will be flexible enough that you can make changes without breaking your design.

USING MULTIPLE PROCESSES

Multiple-process programs are not too common. The main reason is that OS/2's protection is at the process level, and each process represents a program. Applications wishing to share data and especially code within pieces of themselves are more likely to use threads. However, you may want to have some redundant subsystems in the case of critical applications, where the backup systems need to be protected from any problems in the primary ones.

A case in point is the medical example from the previous chapter. If you have one process that gathers data from the medical equipment and another to handle the graphical display of that data, if the display routine encounters problems and traps, it will not take down the process that is gathering the data and possibly storing it in a data base. If this were a single-process application, a graphical display problem would also terminate the gathering of the critical data. This is a perfect example of where the overhead of multiple processes is worth it.

Since ownership of resources—thus, sharing of resources—is done at the process level, no special action needs to be taken to share data among threads of the same application or process. However, when using multiple processes to perform an application's work, you will want to understand and decide early in the design process which interprocess communication (IPC) mechanisms you are going to use.

Writing a multiprocess single application gives you a little more flexibility in which IPC mechanisms you choose, because you don't have the extra burden of understanding the IPC mechanisms for which other vendors are using or publishing interfaces. This part of IPC is internal to the application, so you can balance power, flexibility, and ease of coding without that extra consideration weighing on the decisions.

USING DLLS AND CODE SHARING

Now that you have seen the considerations in structuring your code logically, you can begin to see how the physical structure can also play a big role in how efficient your application is in terms of memory usage, disk space, and load time.

Some programs use no DLLs at all. They usually take a long time to load and initialize and have huge `.EXE` files. There are some at the other end of the spectrum that have very small `.EXE` files and a large number of very small `.DLL` files. Neither is a good approach. Just like anything else in life, moderation is the key. There will be special cases, but, in general you should balance the code between the `.EXE` files and the `.DLL`(s), according to (here is the key) how you use the functions.

There is no guideline on how much or how little code should be in DLLs. The factors you need to take into account are how often the code is used, how long it takes to load the code, and if the functions will be externalized to other processes. In 16-bit OS/2, there was another consideration in that all DLL calls were `FAR` calls, requiring a segment register load, and thus were somewhat expensive. Under 32-bit OS/2, this is no longer a concern.

One of the primary uses of DLLs is to share code between processes. If your application is stand-alone and single-process, there is no compelling reason to put much code in DLLs. The only real reason you would want to do this for the stand-alone, single-process application is to delay loading of code that is used very seldom and for short periods. An example is "help" code, which could be loaded as an asynchronous

task, whereas if it were part of the main executable file, it could be loaded only when called.

If the user of your application never requests help, there is no reason the code should ever get loaded. This is a very good implementation of a DLL, because if you can delay loading of the code until it is needed—in this case, help code is usually needed for only a short time—you will save on the amount of memory your application will require.

Another very good reason to use DLLs is if you have more than one application you work on and sell. It is often useful to write code such as your print routines or application configuration code in a DLL so that you can reuse the functions between the applications. This structure is also advantageous in that any fixes you may apply to one of the products will be picked up by the other. This set of core functions can be kept in DLLs quite easily.

Recall, however, that DLLs are used to share code between processes. There is no reason to place code in DLLs only to be shared among threads in the same process. Since all threads within a process share the same resources, they share the same code and data. As just discussed, if you have a single-process application, the main reason to use a DLL is for performance, by delaying the loading of code usually unneeded, such as configuration utilities.

RESOURCE SHARING AND SYNCHRONIZATION

The idea of resource sharing between threads is enticing, but it too comes with some important design and structural considerations.

When sharing resources, you must ensure synchronization so that only one thread at a time can have access to the resource. This leads to interesting design decisions. Go back to the discussion on how to structure your code for multiple threads and look at how you need to break the application into tasks that can run in parallel. Now look at the fact that you will need to serialize access to shared resources such as buffers, tables, and so on. You cannot do one without thinking about the other.

Sometimes you will have no choice but to refrain from setting up some functions to run in parallel, because of how they need to access resources within the program. In other cases, you can just ignore that fact in the structure and let the semaphores and other control structures you set up handle the concurrency issues. After all, there is no reason not to use multiple threads simply because one thread has to wait for another to finish using a buffer. You can "semaphore-protect" the buffer and let the synchronization fall where it may. This is the most important set of decisions you will make in designing your application. If critical resources are not adequately protected, your application will behave unpredictably. If the cost of protecting it is too high, such as if it is just very complex in terms of how you are forced to accomplish it, it may just be wiser to not multithread that section of code.

The structures you use are also important. The main synchronization structures are the *semaphore* and the *critical section*.

Semaphores were discussed in Chapter 5. They are very simple yet powerful constructs to manage concurrent execution. Critical sections are even more powerful, but they are somewhat more expensive in terms of performance and they have an effect on the entire process—not just the threads requesting a common resource.

When an application calls `DosEnterCritSec`, the effect on the process is that *all* threads except the thread that called the API are immediately frozen, and the only dispatchable thread in the process is that one. The threads are frozen until `DosExitCritSec` is called. In terms of performance, this is an expensive operation. It is also usually overkill except in specific situations. As a matter of fact, I have seen many applications use critical sections, but none that I have seen could not have easily been replaced by a (set of) semaphore(s).

There are no hard and fast rules for structuring your code for semaphores and multiple threads. The point to be made here is—as with everything else said in this book—to think about these considerations and make those decisions before you start coding. Under DOS and Windows, you could get away with less planning, because there were no threads. In this multithreaded environment, either do it right and do it up front, or don't do it.

This is not meant to sound like an ultimatum, but you need to decide whether you will be using the threads. Just remember either to stick to the decision or to rearchitect and redesign if you change your mind. You'll be saving yourself many a sleepless night by following this advice.

KEEP UPGRADABILITY, PORTABILITY, AND SERVICEABILITY IN MIND

Above all, you should keep in mind that the software project you are working on is not done when you ship the product. The work you are doing can be ported and enhanced if you structure it right. You don't want to have to rewrite functions just to add a new feature to the application or rewrite a function just to fix a tiny bug.

By sticking to the subsystem model and modularizing function, you can make working on this project fun and exciting. After all, you'll probably enjoy figuring out new things to add—and how to add them—more than spending the same time figuring out which piece of code an improvement might fit into or how you can squeeze it in without breaking anything else.

SUMMARY

Application structure is more than just how the application will be presented and which features flow into each other. It is about how the code itself is structured to make the best use of OS/2's features, especially multithreading, while remaining as portable and upgradable as possible.

The main points to remember about structuring your code are to isolate your code from the hardware as much as possible and to think about making the best use of threads, balancing it with the variety of tasks your application will perform.

By breaking your code into DLLs intelligently, you will not wind up on either extreme, and if you understand why you want to use DLLs in

some cases and avoid them in others, your applications can be stellar performers without sacrificing function.

Most of all, it is paramount to keep in mind all of these considerations when architecting the structure of the code, because just as in a house, if you want to go back and replace a piece of the foundation without having the support to do that, it will fall and be worse than when it started.

SECTION III

Use Building Blocks or Your App Will Crumble

The importance of using the divide-and-conquer approach to OS/2 programming cannot be overstressed. A block architecture is necessary for OS/2 or any system programming to ensure maintainability and flexibility, not to mention that dividing any task into several subtasks and tackling them one at a time is more productive than jumping in headfirst.

Case in point is the Workplace architecture, as described in Chapter 7. Look at the modularity of the system, where each block has a specific function, and can be updated without the inconvenience of being intertwined with others. They use each other's functions, but in a cleanly architected manner.

Recall the three basic pieces of any application: initialization, process, and clean-up. By subdividing these tasks you can simplify each and keep each piece relatively separate. Using a "black-box" approach, where each of these magic boxes performs a task, allows you to design an application without regard to the details.

At some future time you will indeed have to deal with the details, but too often an application is designed by coding logical functions and then plugging them together. By separating tasks into black boxes that perform an ideal function, your initial design will be devoid of any conventional limitations.

When you break apart those black boxes later in the design phase, you will look at those limitations and turn the black boxes into subsystems and functions. You'll break the *process* task into the user interface, the core or worker code, and their surrounding functions, for example.

This all may seem very elementary and, to some, useless. However, you will find that by following this simple process you will discover most of the mistakes you will make and holes you will leave. It will force you to make decisions up front rather than delaying them—which only delays the inevitable while holding up other pieces of your application waiting for that decision.

CHAPTER 10

Block Design and Architecture

Once you understand the environment and choose the features you will incorporate into your programs, the next step is to map out the high-level architecture by looking at the primary features the application will offer. These features should then be subdivided into the foundation for the entire application.

TAKING THE BLACK-BOX APPROACH

In taking the black-box approach, you will systematically break down your application into functional groups and subdivide those groups into smaller and smaller groups until you reach the level of a single function. This is a long and painstaking process that will lead to frustration and dead ends. It is important to get these kinks out before you get to the low-level design.

This chapter will deal only with the processing part of the application; the initialization and clean-up pieces are fairly straightforward. Let's look at how an application processes information.

First, there is the consideration of a user interface. The user interface of an application is how data gets into the system. You may also have other types of input, such as data collection devices, but, in the majority of cases, user interaction with the application is needed. This should be the first point of attack, as this is the part of an application that usually raises the most questions and concerns.

The user interface is a box. Inputs into this box are user requests and commands, and the output is a coordinated set of functions directed at the worker code. Another input into this user interface box is the information given to it by the worker code, either as a direct result of some user action or by way of the worker code needing to notify the user of an event. Figure 10.1 shows the beginning of the user interface box.

Now that you have this box, you need to make some initial decisions. Actually, most of them should have been made already, such as what mode this application will run in (PM, AVIO, VIO, and so forth). Without that decision at least, don't go any further.

Now that the user interface box is set up, let's move on to the processing box, which has a set of inputs and outputs as well. The inputs are the commands from the user interface box and any other devices or events that can trigger actions by the processing code, such as data collection equipment or modems.

The outputs are many: information to be given to the user interface to present to the user; outputs to memory to simply process and store data for later use; and outputs for printing and data file storage. You may also have outputs for or from utilities, such as data export filters.

These are the beginnings of your foundation block design. As you can see from the simple descriptions of the inputs and outputs of the two major subsystems, the interface and processing subsystems, breaking the application into black boxes is a fairly straightforward task.

OS/2 adds a twist, though. Recall from the discussion on threads that some tasks are easily run in parallel, while others need to be synchronized. The first place to start looking for the parallel opportunities is

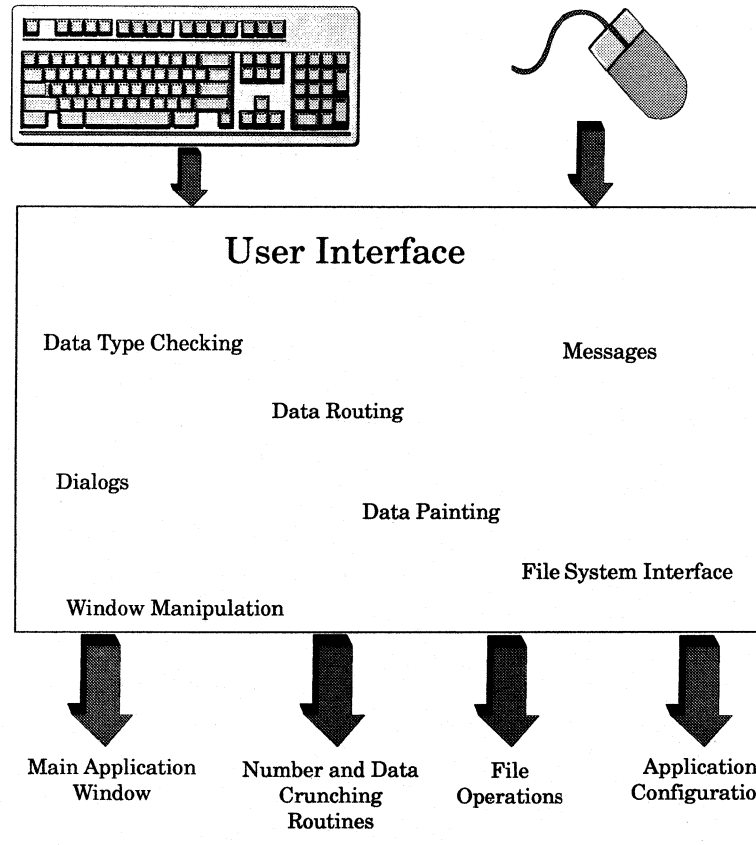


Figure 10.1 User interface black box structure.

this initial black-box foundation. As you break apart these boxes further, as will be done in the next two chapters, you will begin to see how this seemingly simple task will surely lead to better-performing and more readable, understandable, and maintainable code.

In looking at writing multithreaded programs, it is easy to look for tasks that you can run in parallel. In fact, I used to say that you should always be looking for what else you could be doing when performing functions. You should actually be breaking down your tasks into functions as just described, but rather than look for what else could be done at that time, look for the things that you *can't* be doing while that func-

tion is executing. You'll create groups of functions that cannot be run in parallel, and you will put them on the same thread of execution. Then you'll take another group of functions that cannot be run in parallel, but can be run concurrently with the first group. You will put these on another thread. In this fashion, you will create pools of multipurpose threads to efficiently thread your application at the design stage, and not later on when it is most difficult to code and more importantly, to debug and maintain.

LETTING THE OPERATING SYSTEM DO IT FOR YOU

When working on the foundation and the black boxes, keep in mind that the operating system can do a large part of your work. For example, when you work on the user interface subsystem of your application, keep in mind that the Presentation Manager user input subsystem delivers all user input to your application. This way, you don't have to be concerned with reading keystrokes or the mouse movement. All user actions are given to the application in the form of PM messages.

When dividing and conquering the print job, keep in mind that the operating system provides independence from device specifics, so much of the work of drawing the output on the printer is already done if it is on the screen. By understanding this, you can see that it is efficient to have a box that draws the current data. This box can be used to draw on the screen, the printer, or any other output device.

Since OS/2 provides the presentation space interface for drawing, all the application needs to do is to have a PS for the screen, a PS for the printer, and one function to draw. This can be taken even a step further by understanding how user input functions and how you can write the drawing function on a separate thread.

For example, drawing is usually a time-consuming operation. Recall how the system processes user input through the single queue. If the user requests something to be drawn and the application is single-threaded, all user input is held up until the drawing is done. By placing all drawing functions (or any time-consuming function, for that mat-

ter) on a separate thread, you can ship the task off to be processed asynchronously while the main thread continues to process user input. Notice how this holds true for screen drawing and printing so that you can reuse the drawing code. Also understand however, that in the current Intel implementation of OS/2 this architecture will hold up all user input in the system. Under the Workplace architecture as outlined in Chapter 7, however, the event/session manager controls user input, so other applications not requiring the single queue (along with session switching) will continue.

This only scratches the surface of how to begin building the foundation of your application. The concepts are simple; the most important thing to remember is to go though this phase of design slowly. Because it is so straightforward, it is tempting to draw a couple of boxes and completely skip the important issues.

SUMMARY

Create your box foundation and define all your inputs and outputs. Begin to subdivide the main subsystems one by one, but all the time looking at the others to see where you can share and reuse.

In the following chapters in this section you will see how to break these subsystems apart into components, how to look for opportunities to let the system work for you rather than reinventing the wheel, and ways to keep your functions separate while mixing functions and threads.

CHAPTER 11

Designing the User Interface

The first part of your application that will be running is the user interface. In general, you will code some sort of interface to the main worker code to begin testing both. Why not make the user interface as close to your final product as possible?

The OS/2 Presentation Manager is a window-oriented presentation medium. The first part of the interface needs to be what the windows will look like and how they will interact with the user and each other. Windows display data, act upon commands from the user, and solicit data from the user when the situation dictates.

WINDOW DESIGN

The window design of your application defines how easy it is to use the application. The interaction between the user and the application is directly affected by how well you work all of the functions of the application into windows.

There are things you should keep in mind when working out how your windows will show data and receive input, and there are tools to help you do it. For example, when should you prompt the user for information through a dialog window versus having the user take the initiative to set options? How should the menu bar be structured? Should you use submenus from the menu bar or actions directly? Will there be draggable objects, representing data, within the windows? How can you get the skeleton code of the user interface running as quickly as possible? Are there guidelines to help you decide?

The answers to these questions are mostly up to you, the designer. There are guidelines and tools to help you make your applications look and feel consistent with other OS/2 applications. These tools will also help you generate the code for the foundation of your user interface.

CUA

CUA (Common User Access) is a piece of IBM's Systems Application Architecture (SAA). CUA defines a set of guidelines for how applications should present data and interact with users. CUA has undergone several changes since its introduction in 1987. The specifications and guidelines have been developed and updated based on studies of usability and productivity.

An example of CUA is the guideline on how a menu should behave when selected. An action bar item, when clicked upon, should not cause execution of an action but should only display a pulldown menu. According to CUA, all action bar items should display a pulldown.

Another example is the behavior of menu items. Menu items within a pulldown should cause an action to occur when selected. CUA also says that any menu item that causes a dialog window to be displayed should be followed by an ellipsis (...). Figure 11.1 shows an example of such a menu, where the "Open" item has an ellipsis indicating that a dialog will appear when this item is selected. These guidelines help users avoid surprises when using their software and lower the learning curve of applications conforming to CUA.

CUA is a slowly moving target. You can aim at and hit it, but with ever-increasing studies and knowledge CUA will likely change some-

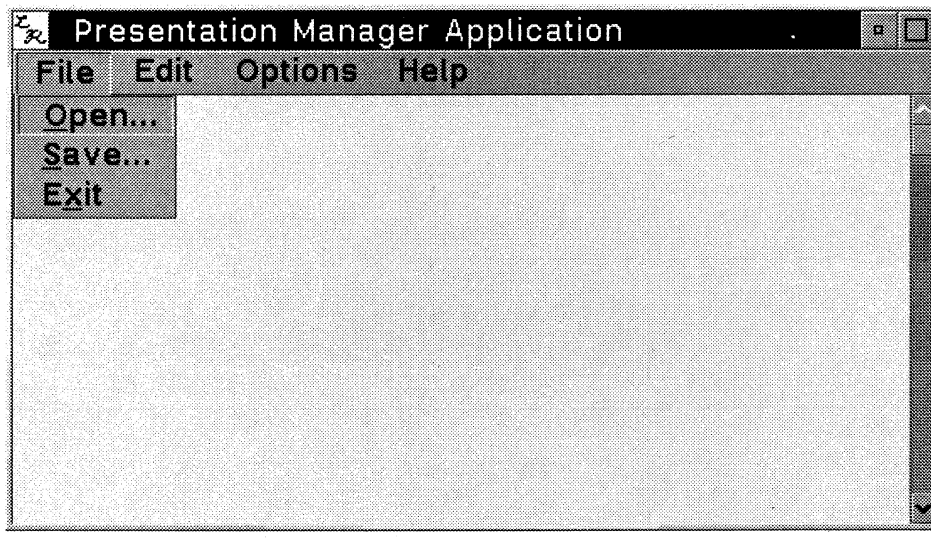


Figure 11.1 Menu bar with ellipsis.

what within a few years. The CUA guidelines are just that: guidelines. No law says that you must follow all of them. The biggest advantage of sticking close to CUA is to benefit from the studies and other work done by IBM on user interfaces. Another important thing to consider is to keep the learning curve of your applications low by making them consistent with the operating system and other applications.

CUA should not limit your creativity and your efforts to make your programs flashier and more interesting than those of your competitors. However, by staying somewhat in line with CUA, you get the best of both.

Presenting Data

A window is defined as a rectangular area of the screen upon which data is displayed and input from. After that, it's up to you. This is one area that will differentiate your application from others.

What you sometimes think of as a window, which is called the *standard* window for convenience, is really a complex network of windows. Figure 11.2 shows the parts of a standard window. Each of the pieces

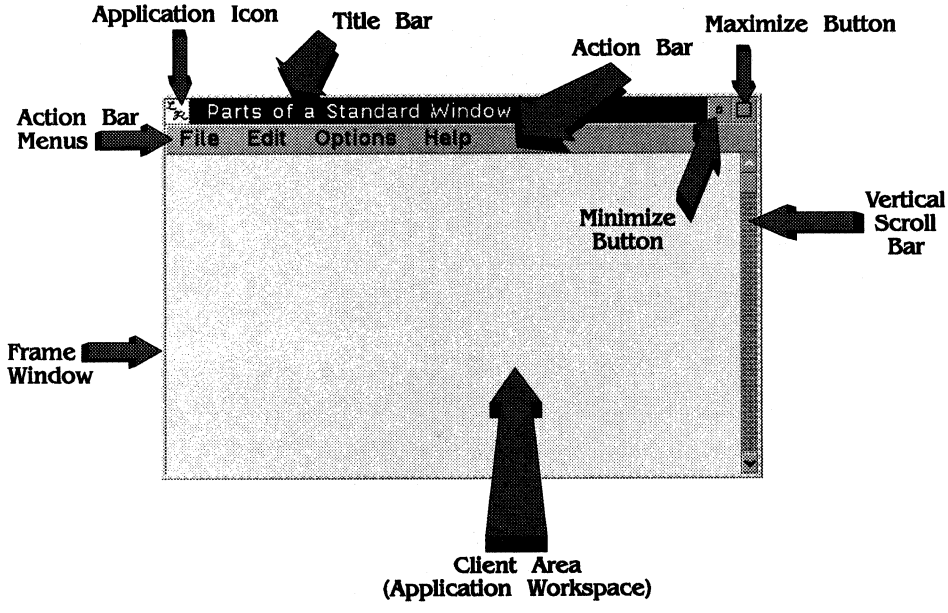


Figure 11.2 Parts of an OS/2 standard window.

of a standard window is a window itself. The area where the data is presented is called the *client* area, or client window.

The first order of business in window design is to decide how you are going to present data in the client area and how you want to allow the user to control it. There are many types of applications, from CAD and graphical drawing programs, to spreadsheets and word processors, to database and statistical analysis programs. There are types of programs not even thought of yet.

The way you display data in the client area is completely up to you, as is how you control it. OS/2 provides control windows to help you manipulate how the client area is shown, and there are some commonly accepted practices. Of course, you can use these ideas as you see fit.

Window Controls

Some considerations when making your initial decisions include whether the data will be scalable or fixed-size within the window or whether

you plan to implement drag-and-drop functions for the data. These items help determine how your main window will be structured.

If, for example, you plan to have the data displayed in the window resized when the window is resized, scroll bars are not necessary for the client window. This depends on the application type, of course, but for word processing or typesetting programs, for example, you will likely want to keep the data at the same size no matter what the size of the window. You can provide the user with zooming functions, but in general WYSIWYG applications the data is scrolled using scroll bars rather than always sized to fit its window's current size.

Graphics-oriented programs, on the other hand, are not as particular when it comes to sizes, and the data can be scaled to fit the window. You can provide scroll bars and possibly an application option to turn off dynamic sizing of data and give the user the choice of scrolling or sizing. Of course, you can do this for the other types of applications previously mentioned as well.

Drag-and-drop functions on data do not affect the client area's controls as such, but it is important to decide very early whether you will be implementing this type of function when you are putting together the window design. The main reason has to do with multiple documents (files, graphs, or what have you) and moving data between them.

Wouldn't it be useful for a circuit designer to map out a design and be able to drag and drop it onto any circuit boards being worked on? How about simply grabbing a section of a spreadsheet and dragging it to a letter or report? You may notice that these functions can be accomplished with SOM, DSOM, the PM drag-and-drop APIs or even by writing the mouse and data handling code yourself. The considerations of using the various methods of data interchange will be discussed in depth in Chapter 16.

Dialogs

The implementation of dialogs in OS/2 has gotten easier from release to release. Dialogs are special-purpose windows with a set of controls designed to solicit data from the user (carry on a *dialog* with the user).

Unlike standard window controls, dialog window controls display and can receive data controllable by the application and the user. Standard window controls (usually called *frame controls*) are those that simply send notifications to their owner so that the owner can perform an action based on the user's interaction with the control. An example is the Minimize button. When the user clicks on the Minimize button, it sends a notification to its owner (the frame of the standard window) to tell the frame that the Minimize button has been selected. The frame, as a response to this notification, minimizes the standard window. The Minimize button by itself does not perform a complex function—it simply does as any control does: It notifies its owner of the action.

A dialog control, such as an entry field, is designed to present and/or receive data. Other such controls are listboxes, multiline entry fields, and radio buttons.

Early in OS/2's evolution, only the ability to present dialogs was available. It was up to the designers and programmers of applications to define dialogs for such tasks as opening files, selecting printer destinations, and selecting fonts. This led to a large amount of "reinventing the wheel," as each application had to come up with its own variation on the File Open dialog, for example. As OS/2 matured and continues to mature, standard dialogs are being added to the base operating system for use by applications.

Two such dialogs are the File Open/Save dialog and the Font Selection dialog. These are predefined by the system (according to CUA, of course) and are available to applications through a simple API. You can customize some parts of the dialog by manipulating the data structure you pass to the API.

Dialogs are an important part of the application. They are a primary method of getting data into the application, and how your dialogs look and feel determine in large part how your application's usability is rated.

Dialogs should be clean and efficient, and the application should not have so many layers of dialogs that it begins to look cluttered. Many people ignore the basic precept of dialogs: A dialog is not a lingering conversation. A dialog is meant to be interacted with and then dismissed. All too often I have seen applications in which the dialog needs to be

displayed during a large part of the application's operation. Have the conversation with the user, then end it to have another later on.

Stick to this simple principle: The user will select an action. If you can, perform it. If not, put up a dialog to get the details, then get it out of there.

Screen real estate is valuable, especially in a system in which you will have several applications going at once. The last thing the user wants to do is to have to keep moving dialog windows out of the way to see what he or she is doing.

Now that the purpose of a dialog window is clear, you need to decide the most efficient way to get the information from the user. A dialog should flow much like a printed magazine advertisement. The main idea and primary pieces of information should be near the top of the dialog, with secondary information below it. It should flow smoothly from left to right (keep in mind considerations for countries that have written language that flows from right to left) and down the window.

Along with the flow of the dialog is the choice of controls for the different pieces of information you need to solicit/present. For example, you could simply use the entry field control for each piece of information. It is easy to code, but not very efficient or intuitive. OS/2 provides a variety of controls for different uses.

For example, to ask the user to select from a fixed list of choices, a listbox is appropriate. If the list is very short, and you might like to show the user a preview of his or her action should the dialog be closed with that item selected, you might choose a set of radio buttons. If the list is dynamic, or you want the user to be able to type in a selection should the desired one not be in the list, a listbox or combo-box, respectively, would fit the bill nicely.

All of the controls in OS/2 are described in detail in the OS/2 programming reference books. The important thing to remember is to keep it simple! All too often, programmers want to add flash and intricate-looking function to parts of their code. In the case of dialogs, it is counterproductive. The dialog needs to be smooth, easy, and intuitive. Save your flash energy for the data presentation or computational features. Make the user do as little work as possible.

One suggestion is to prefill in some fields for the user, based on system defaults or the action the user is currently performing. Don't limit their choices and options to change your assumptions; the whole point is to make the user more productive—not require answers to more questions.

One other fine point about dialogs is the issue of *modality*: how accessible other windows are while the dialog is processing. A dialog window may be modeless or modal. A modeless dialog does not inhibit interaction with any window while the dialog is being processed. A modal dialog, on the other hand, inhibits the user from interacting with other windows while it is up.

There are two types of modal windows: system modal and application modal. A system modal window is one that inhibits user interaction with all other windows in the system while that dialog with the user is going on. Application modal windows, as the name implies, inhibit only user interaction with other windows within the application. More precisely, such a dialog inhibits interaction with the owner of the dialog and all children of that owner.

This leads you to make careful decisions with your owner and parent hierarchy, especially with application modal dialogs. If you look carefully at the definition of an application modal dialog, you can see that if the owner and parent of the dialog window is the main window of the application, you will inhibit not only all children of the main window, but also the dialog itself, since it is a child of the main window too! You will have a hang situation in your application.

Properly managed, modal dialogs can be a powerful tool. For example, you could use an application modal dialog to change settings in the application without worrying that the user may be changing something else within the window while the dialog is going on. Applications usually don't have much use for system modal dialogs, but application modal dialogs can make your applications work more intuitively.

How dialogs are built is another important aspect of dialog manipulation. Dialogs can be built using the dialog editor to build a resource file to be bound to your application, and in fact, that is the generally accepted method of doing it. This dialog template in the resource file is

built in memory when you load it using `WinLoadDialog`. Dialogs built in this fashion are relatively static.

You have several other, more dynamic options when you build dialogs. One, for example, is to build that dialog template in memory. You would execute its function the same way as if you loaded it with `WinLoadDialog`, but in this case, you've already loaded it, since you have built the template in memory.

Another way is simply to create a window with the appropriate controls (push buttons, combo-boxes, listboxes, and so on) to function like a dialog with the attributes you desire. The trade-offs here are that as you move down this list of options, you have to add more code at each step.

A dialog in the resource file requires the smallest amount of code to build and operate. Building the template in memory is a bit more involved, and of course, creating all the windows and controls on-the-fly is the most involved. In all cases you will need a window procedure to govern the behavior of the dialog, but each step is more complicated in how it is created. The added benefit is the flexibility. So if you need a dynamically built dialog, you know what you're getting into at each step.

Application Defaults

Keeping with the simplicity theme, you should have the ability to prefill dialog fields with information from the overall system configuration or individual application configuration options. By using application defaults, you can avoid asking the user questions for the most common tasks. An example would be a default printer for the application.

By having the user specify a default printer when the application is installed (of course you will provide a menu option for the user to change *any* application default setting anytime), you can avoid asking the user for the printer every time a job is to be printed. The user could ask to have a job printed, and a simple confirmation dialog would appear. If the user has no changes from the application defaults, it is a matter of one mouse click or one keystroke to complete the job. I have seen some applications that provide a default printer, but they just query the default

printer for the system every time they start up. By saving the application defaults, you preserve user choices across application invocations, not just within the same one.

Somewhere on the action bar you should have a menu so that the user can change any of the application defaults anytime. Again, as with dialogs, keep it simple. In any type of dialog you can overdo it. Don't force the user to fill in every option, especially the obscure ones. Pick the most common functions, such as where to pull data files from or where to print data to, and make those available as application defaults.

To sum up dialogs: Keep them simple. The faster, the better. Dialogs are an important part of your application. Just as with the action bar, they determine how easy or hard it is to work with the data in the application. The features and functions are only half the battle. If it is too hard to use the functions, users will go elsewhere.

Dialogs determine how elegant or clunky your application is and how it is perceived in the marketplace. Try to avoid nesting dialogs wherever possible. Just as with nested menus, it becomes annoying to finish with one dialog only to have it bring up yet another for another interrogation session. If you have involved information to gather, see how you can simplify it. You may even want to have an option for different detail levels for different types of users.

Try to anticipate what the user is asking, and prefill fields. If you have many pieces of information you need, move as many as possible to application defaults. One other consideration is that you are technical. Your users may not be. Some choices that you might want, users may never even care about. I don't suggest removing large functions, but keep in mind who your target audience is.

The easier the dialogs and menus are, the more productive your users are, and the more they will rave about how easy your application is to use.

Application Development Tools for OS/2

The complicated interactions between windows in your applications can seem very intimidating. Recall that each control is a window and

each window interacts with at least one other or the user. Some windows own many others and have to coordinate behavior. Defining efficient and useful dialogs means that you will be creating and destroying hundreds of windows throughout the execution of your application. Just writing this user interface code can keep you occupied just by its volume.

Traditionally, user interface code tends to be as complex as (or even more complex than) the code that does the real work in the application. This does not mean that you have to spend a large part of your development budget on user interface code. Computer-aided software engineering (CASE) tools have been on the programming scene for years. For some systems, they can be used to develop an entire application; this is especially true of database systems. CASE tools can come in a variety of flavors. Some take the form of interface builders and code generators, whereas others are full-blown object-oriented languages with their own runtime and object libraries. As with anything, each has advantages and drawbacks. For simplicity's sake, they will all be referred to as CASE tools here, since they are all tools for computer-aided software engineering.

By using entity-relationship diagrams, among other methods, CASE tools can be used to "write" entire applications. Although there are some complete object-oriented languages, such as SmallTalk, you can also find full application development environments for OS/2 that will go from helping you write user interface code, to helping you port existing application source code, all the way to actually being the application for you.

The main function of a CASE tool is to help the programmer write a program without writing a lot of code. Depending on the tool, it will generate objects, executable code, executable code that requires the CASE tool's runtime library, or even stand-alone source code that can subsequently be modified. In any case, these tools are designed to make some of the more routine tasks of programming faster and more efficient, leaving the programmer free to tailor the design and optimize the application.

Since OS/2 2.0, there have been huge advances in the number and power of CASE tools available for OS/2. Since this book is not designed

to be a product review, they will not be compared and contrasted; rather, I will simply present my own personal view on what I prefer in the tools and what they provide. There are advantages and drawbacks to the different types of tools, not just in different vendors' implementation of the tools.

In the early days of OS/2 development, application development tools were limited to GUI generators and high-level languages. There was even an OS/2 version of SmallTalk. Since then, however, the number of choices has grown significantly to application porting tools, API translators and rapid application development environments in addition to more function in those GUI generators and languages. We'll start out with the GUI generation tools, then move on to the API layering and application porting tools, and finally on to the development environments.

GUI generation tools were the first to arrive on the OS/2 scene. Most of these tools allow you to "paint" windows, specifying colors, sizes, attributes (such as sizability and visibility), window controls, text, and other behavioral aspects. Such tools include GPF from Microformatic, and Kase:PM VIP from Kase Systems. This type of tool usually takes the painted windows and generates code for your use.

This type of tool is very good for those who have not done an extensive amount of PM programming, as it takes a lot of the guesswork out of owner and parent window relationships, programming dialog flow and window control and communication. The development process with these tools is usually iterative, where you will paint some windows, generate the code and add in your own functions (since GUI generation tools give you only skeleton window code), compile the code, observe the results, and then do it all over again until you get the desired results.

The code generated by these tools is usually pretty good, and usually well commented. It takes a great deal of the burden of writing routine window management code out of your hands so that you can concentrate on data display, graphics routines, printing, threading, and data manipulation. In most cases the regeneration of the application code by the tool after inserting your own code will not affect that new code, so you can add windows later and have the tool regenerate for you, without fear of destroying custom pieces you've written.

One drawback, however, is that you are locked into the programming preferences of the tool developers. For example, if you want to send a message to a window owned by another, you could call `WinWindowFromID` to get the handle of the target of the message, then call `WinSendMsg` (or `WinPostMsg`) to get the message to the window. Alternatively, you could call `WinSendDlgItemMsg` which in essence, sends a message to the window with the ID specified that is owned by the target of `WinSendDlgItemMsg`, effectively combining the two calls into one API. While this example is simply a matter of programming style, there are many such cases that could affect application size and performance.

Some tools, such as Kase:PM VIP, have replaceable modules that allow you to rewrite some of these preferences (called the Knowledge Base) and add others.

The advantages of this type of tool are that it is fast, you get native source code, there are no runtime libraries to add overhead and you have the flexibility to modify any of the code given to you, since it is all source code. Some of the drawbacks are that you can do only the things the tool allows you in terms of painting your windows, you either have to live with or work to modify the code given to you if you are not happy with it, and many of the tools integrate your code with "user-exit" type functions rather than allowing you to add functions "in-line."

If you are writing code from scratch (i.e., not porting an application), have little PM programming experience, and want to get your user interface up and running quickly, this is the type of tool for you. The code is highly tunable (after all, it is source code) and you can, at some point, make a break from the tool if you want. One thing to remember, though, is that these tools generate interfaces. There are some that can do much more for you.

Another type of tool is the porting tool. One tool that has been around for some time now is an API mapping layer called Mirrors. This will take a program written to the Microsoft Windows API and translate those calls, on-the-fly, to OS/2 API calls. This technology is becoming dated and while effective, is not a good strategic alternative.

Another one of these tools that operates on source code is SMART. This is from One Up Corporation and stands for Source Migration Analysis Reporting Tool. Quite simply, it takes source code written for a

platform such as Windows, and it ports it to OS/2 source code for you. For functions it can port, it will. For those it cannot either because there is no direct mapping, or because it does not understand what you were doing, it will provide recommendations and even estimate how long it will take for the remainder of your porting effort.

SMART is very effective in porting your source code from Windows to OS/2, and it is possible for this tool to be extended to give you common source code bases across platforms, by taking code for other platforms and giving you OS/2 source code or vice versa. This provides you a great deal of flexibility since you have the source code, the benefit of the SMART programmers giving you advice on how to convert the code it could not, and the opportunity to tune the code as you wish.

The caveats to this approach are simply those of any port. Since OS/2 is the first operating system in the PC arena to support the programming of threads within processes, the architecture of your resultant ported OS/2 code is restricted to that of the original environment. Recall the discussion of adding threads to an application after coding is done. While SMART is excellent for getting you a native OS/2 application, it should be viewed as a stepping-stone to fully exploiting all that OS/2 offers.

Applications ported in this manner will not utilize threads, the full-featured memory management subsystem, or other functions of OS/2 that make it a more flexible system to write for. Tools like SMART give you the ability to have a native application quickly, upon which you should learn and build. Of course if you're writing an application from scratch, you should look into the first group of tools discussed, or the next group, the application development environments.

Another option open to you is not a new concept, but there are new tools in this area coming on the market all the time. This is the library product providing libraries you can write to. These are usually positioned as extensions to the OS/2 API, and the one I am thinking of in this instance is the User Interface Class Library (UICL) from IBM. This product is a set of classes that you can use in your applications and derive new classes from, or just instantiate yourself. The idea here is that the OS/2 API, and no API for that matter, is ever as complete as

programmers want it to be. The market is still open for these tools, and many excellent ones such as the UICL are available today.

Application development environments come in many flavors. One thing they usually share in common is their reliance on runtime libraries. The trade-off in using these tools over some of the previously described ones is that application development environments are used for their speed of application development over their performance tunability. Let's look at the SmallTalk language for example.

A good introduction into the world of object-oriented systems is SmallTalk. This is a product that is available on many platforms, including OS/2. SmallTalk is a language and a development environment that uses object definitions based on a class library it has built into it and allows you to construct on top of that. SmallTalk for OS/2 is consistent in use with the versions on other platforms, which makes its learning curve lower than those of other tools. SmallTalk generates executable code that requires the SmallTalk runtime and class libraries and uses internal SmallTalk routines.

SmallTalk is itself an object-oriented language, and its graphical implementation is similar across all the platforms on which it is supported. This is an advantage in and of itself, since once you have used SmallTalk on one platform, it is virtually the same on the others. It incorporates true object-oriented functions and abstracts your applications at such a high level that you will likely not have to deal with much source code (depending on how much you consider the SmallTalk definitions source code). SmallTalk's language uses its existing complex hierarchy of classes to allow you to build your own classes and hierarchies to complement its own. It is also neatly packaged and can be installed on any system with a minimum of work. IBM has developed and sells an implementation of SmallTalk in addition to the one put out by DigiTalk.

Taking the SmallTalk concept a level higher is the VisualAge application development environment from IBM. This is not a GUI generator, or a programming language or object manipulator like SmallTalk. VisualAge is an environment that allows developers to rapidly develop full-functioned applications in an object-oriented fashion without writing source code.

One might argue that this is the end of programming as we know it. Not quite. While VisualAge is an excellent development environment, it requres a significant amount of overhead to run not only the environment, but also the resultant applications. You see, environments such as SmallTalk and VisualAge are powerful and fast, but they require you to use their runtime libraries. One of these applications is only as good as the support code underneath it.

In contrast to the other two types of tools that give you source code, this type of application development tool gives you an application—not code. As such, you lack the ability to fine-tune the application depending on the environment. If your goal is to get a database transaction application online quickly, and you have workstations that are not small in terms of resources (memory and disk) then one of these tools is for you. In fact, VisualAge uses the IBM SmallTalk as its base.

Other such tools that do not require huge run time libraries are VX REXX from Watcom and VisPro/REXX from Hockware. These are application development environments that are object-oriented, and will give you a fully functional application, but they use the OS/2-supplied REXX feature as their runtime base, thus reducing the amount of overhead such as application development environment might use.

You can see that in only a short time, a wide variety of development environments have come on the scene and, by the time you read this, there will probably be more.

Object Technologies

Still other application development tools lean toward the object-oriented technologies, assisting you in adding exremely advanced function to your applications, in addition to making the new technologies such as SOM and OpenDoc more accessible and easier to use.

As you've seen, and will be elaborated on a bit later, SOM is the core of the Workplace Shell. SOM has also been adopted by CIL and CORBA, and is fast becoming an object technology standard. While SOM is not terribly difficult to write code for, it is not much fun and it can be very slow.

Just like any other technology, the tools took some time but, now compiler vendors are working to put SOM directly into their compilers. This technology is being referred to as Direct-to-SOM (DTS). DTS is analagous to putting database action verbs into your COBOL programs. DTS is an extension to the programming language you are using (mostly C) that allows you to invoke methods and define behaviors right inside your C code. This makes the development of SOM objects easier and faster.

Another object technology that is coming on the scene is OpenDoc, which is an object-linking and -embedding technology that can be used by applications to allow users to easily compose compound documents from parts in various applications. For example, a user could take part of a spreadsheet, combine it with a document from a word processor, and add in a graph from a statistical package. Of course you could do this with the clipboard, or even with DDE, but OpenDoc provides an architecture that is being adopted by many companies to make the programmer's job easier by providing a framework rather than requiring everyone to figure out how each specific application communicates. OpenDoc adds a new dimension to your applications without adding new dimensions to your coding job.

OpenDoc should be viewed as a tool for your applciation to extend itself and interoperate with others, rather than a feature for your application in a stand-alone environment. It's a way for users to take parts (and they are called OpenDoc "Parts") of applications and combine them to create online compound documents. In fact, many OpenDoc-based documents have "live" objects that are always updating, and are not always suitable for printing, but they are suitable for electronic sharing and viewing, showing the power of the technology.

When selecting an application development tool, you need to look not only at the application, but also whether you are porting code, whether you care more about speed and size versus speed of development, and how much code you are willing to write. You may find that you will use a combination of methodologies. You could use SMART to migrate code from one area of an application being ported, while using Kase:PM VIP to write the GUI from scratch. Whatever you choose,

look at what your development environment is, and look at the intended execution environment. These will guide you to the most appropriate tools.

WORKPLACE SHELL

It is not only important to have a smooth flow within the windows of your application; it is equally important to make your application look like part of the computer system. This goes along with the integrated environment concept. The Workplace Shell (WPS) is an application launcher, as is any shell. However, the WPS provides much more to applications.

It is just as important to use only the features of the shell that you need as it is to not overpower your application's function with fancy controls and unnecessary menus. The WPS is very powerful. You can create objects, new classes, and templates and implement many functions not previously available in the PC arena. You need to understand the architecture of the WPS and not overdo it with function; otherwise, just as if you had too many cluttered dialog windows, your application can appear clunky and hard to use.

You should also avoid the replacement of parts of the shell on a user's system, as this directly affects the goal of intuitive use, especially if the user goes to another system that does not have your application installed and the behavior of the whole system is different. The WPS allows you to replace parts, but that feature is there for specific-purpose applications rather than the general computer user.

The main purpose of the shell is to start programs and manipulate data in a consistent manner. As you already know, the Workplace Shell is an object-oriented shell that allows the users to use a computer and work the way they do without a computer: with objects.

A program is an object, as is a data file or a printer. To look at a data file, you must somehow tell the computer what that file is. Generally, it is a data file created by some application, such as a word processor or spreadsheet. The shell works with associations between objects. That is how it knows how to open and present objects. When an object is opened

(with a double-click), the object is opened with its default association. You will shortly see how these associations are built.

Let's start with the basics of how the shell operates. Everything in the computer can be represented with a Workplace object. This object can be created by an application via `WinCreateObject`, or the user can create an object through templates, copying another object, creating a shadow of an existing object, and so on. Once the object is created, it can be manipulated by the user. How the object behaves is defined by the class it belongs to.

Objects and Templates

As you have seen in Chapter 6, there are three general types of objects. There is the transient object, which has no information stored across reboots, and then there are the abstract and file system objects, which store their persistence in the `.INI` file and file system, respectively. How do these objects work?

In general, you can think of a program reference object as an entry in a program starter list. You can create a program reference that points to an executable file. When this object is opened, the program is started, and you can open data files or print, as you have in any other system. This is the application-launching function present in any good user shell. However, the WPS is more than this, and by intelligently implementing functions provided by the shell, you can change people's impressions of applications. Applications, data files, and other objects will become part of the computer, part of the user's work environment.

Do not think of your applications any longer as programs that need to be fed data. Think of documents, spreadsheets, or any other object that the user wishes to work with. Now that you have that firmly set in your mind, let's move to the choices you have in presenting these documents to your users.

Step 1—Simple Drag-and-Drop

The first step most users will take in trying out the drag-and-drop functions of the Workplace Shell is to drag a data file to a program reference

object and drop it, hoping that the application will start and load the data file. Once there, subsequent files would have to be loaded the old way, by asking the application to open them. This is a good first step and a nice introduction into the object-oriented world of the WPS.

Let's look at how to make this work. The WPS cannot do all the work; the application must do some. In order for this function to work, the application must be able to take a *command-line* parameter. That is simply how the shell does it. When an object is dropped on any other object—a program reference in this case—the shell will start the program (`DosExecPgm`) and pass the name of the dropped object as a parameter. Assuming the application knows how to use this information, the application will start and will do whatever it will with the file dropped.

A good example of an application that just about always takes such a parameter is a text editor. Usually, you can type `edit filename.ext` (assuming the editor's name is `edit`) and it will start up, loading `filename.ext` as the file being edited. Some of the more complex applications that have been written do not accept such parameters and will not work within this scheme.

As you will see, there are several ways to cause your application to present documents or data through the functions of the WPS. This is the simplest. My recommendation is to support this, even if you are using other methods as well. The reason behind this is that it takes a trivial amount of code to support this and gives your users more flexibility in how they view objects. The worst thing you can do is tell the users, "That's the way it is. It is good for you and you'll do it *this* way." Keep things flexible. If you can do it with a minimum of work, go for it.

Any application that can handle a command-line parameter can make use of the WPS drag-and-drop function this way.

Step 2—Associations

The next more powerful function you can use is *associations*. An association is exactly what it sounds like: It associates an executable with type(s) of objects, either by object class or by filename extension. Associations can be created by the user or by the application.

User-created associations are not relevant here, since we are dealing with keeping work away from the users and letting the applications do it. Application-created associations are straightforward.

When an application is built, it can have a resource known as an *association table* ("assoctable"). The table is built using a resource definition with the `ASSOCTABLE` statement. When the resources are compiled into the executable program, the assoctable is built. The first time the shell wakes up the object representing this executable, it looks to see if there is an assoctable. If so, it builds the associations for the application.

For example, let's say you are writing a word processor called "My Word Processor," and all of its documents will have the extension of `.MWP`. Listing 11.1 shows what the `ASSOCTABLE` statement might look like in the resource file.

When the program object is awakened for the first time by the shell, an association for the file type `.MWP` will be built, and a new template will be created in the Templates folder. Once this occurs, the user can then simply grab a template for an `.MWP` document and place it anywhere. When the user opens the object, the program will be started and the particular document will be loaded. Of course, this still relies on the command-line parameter mechanism mentioned previously. The parameter-passing mechanism is the cornerstone of this function, and you can add to the ease of use further by adding the assoctable to the application.

At this point, your users can drag a document to your program object and drop it there, or, if they wish, they can simply open the document by double-clicking on it, and it will be loaded into the program. This additional function of opening the data file object and the system starting

```
ASSOCTABLE
 BEGIN
  "MyApp Document", "*.MWP", AF_DEFAULTOWNER,MYAPP.ICO
  "MyApp Backup Document", "*.MBK"
END
```

Listing 11.1 Sample association table in a resource definition file.

the application and loading in the data file was made possible by the association.

Step 3—Writing an Object

The next step is to write an object. This is not as difficult as it sounds, and it adds a new dimension to your application. There are some things you need to understand and be aware of when writing objects, but the power and flexibility you gain far outweigh the downsides.

If you have been programming for the WPS for a little while, you will also notice that the WPS methods have been enhanced in OS/2 Warp, showing the ongoing maturation of the architecture and the robustness of the function. Many formerly undocumented methods are now exposed for your use and of those that were there, many have been reimplemented to provide more consistent and intuitive function, making your programming task easier. I highly recommend obtaining the latest programming manuals from IBM either in hard copy form or from the Developer's Connection CD-ROM. After reading the improved WPS programming documentation, you'll find your job easier than ever before.

Application Launching

As you have seen, the main purpose of the shell is to launch applications. The Workplace Shell provides functions to allow these applications to communicate with other objects in the system—most importantly, the data file objects they will be manipulating. At first glance, it seems intuitive to write your application as an object. After all, when a file gets dropped or an object is opened, it is started. You would think that you would want to simply implement your program as an object.

This is not a good idea. As you will see through the next few pages of discussion, there are better ways of implementing object functions in your application than writing it as an object. In actuality, the best way to write this function is to write an object that represents your *data file(s)*. Each type of data file you will manipulate should be a separate type of

object (unless of course they all behave the same way, in which case you need only one class of object).

Remember that users deal with objects. Your job as an application designer is to hide the fact that they are dealing with programs at all. Users should be working with objects, such as documents and graphs. By writing objects that represent real-world objects, you allow users to learn your applications easily.

Objects that you will write (in the most general case; there will obviously be many special uses and exceptions to this) are subclasses of the data file object. When you create a class with `WinRegisterObject Class`, you create a new template in the `Templates` folder. When the user "rips off" a new object from the appropriate template, you create a new object belonging to this class.

The main method you will subclass in this new class of yours is `wpOpen`. When `wpOpen` is invoked in your object, you should call `DosExecPgm` to invoke the executable that is your application. You can also pass these command-line parameters to tell the application what is going on and why it is being started. In the most general case, the user is simply trying to work with the document, and the application's job is to provide it. As you will see shortly, this mechanism is useful for other object functions as well.

Now you have some decisions to make. First, what do you do if the user opens another document? Should you start another copy of the application code? It is easy enough to detect whether a copy is already running, but you need to define an object interface to the application code to tell it to load another file. You could choose to use shared memory, system semaphores, Dynamic Data Exchange, or any other IPC mechanism you like.

There is no set answer as to what to do in these situations. It depends on your application and the functions you wish to provide to the user. Of course, you will most likely not want to start a separate copy of the application for each object opened, but you need to decide how you want to implement the communication between the object and the application.

Recall the principles of object-oriented systems. The most important ones here are inheritance and the fact that all objects belonging to a

class behave the same way. Once you decide what you want to do, you have to write it only once. Each instance of the class behaves the same way.

Single-Process Model Considerations

Recall from Chapter 6 the single-process model of the Workplace Shell. Workplace objects run in the Workplace process. If you were to write your application as an object, several things would happen.

First, if your object has a bug in its code (that will never happen, we all write great code the first time, right?), it can take down the shell process. This is a drawback of the design of the shell, but for now, it is a situation that must be dealt with. As discussed in Chapter 6, the shell process will be restarted and objects will be restored to their previous state. However, the more code you write to run in the shell process, the more risk you are taking.

The other more important consideration to note in writing your objects is that if the shell does need to be restarted, it is the whole process that needs to be restarted, since it is the process that is brought down. Other processes are not affected, but anything running within the Workplace process has to be restarted. If your application is part of, and is running in, the Workplace process, all updates to the object in the application will be lost. By using `DosExecPgm` to start the program when your object is called at `wpOpen`, the application is running in a separate process and is thus unaffected if the Workplace process has a problem.

While the addition of DSOM to the function of the shell is important, the bottom line is that you should write only what you have to as an object and not code the whole application as a DLL to run under the Workplace process. This also gives you the added advantage of inheriting many of the methods from the parent class (most often the data file object class) and only subclassing (or overriding) the methods you need to start and communicate with the application executable.

Interprocess Objects and Agents

It is often useful to communicate with Workplace objects. An example is when a user takes a data file object, drags it, and drops it onto a running instance of the application. You need to know what you want to do in that case. Since your application is not a Workplace object, it cannot invoke the object's `wpOpen` method. One way to do this is to have a special Workplace object class whose sole purpose is to talk via IPC with the running instance of the application and invoke WPS methods at its direction. This is an example of an *agent*.

An agent is a piece of code whose sole purpose is to transfer messages between dissimilar communicators. In this case, the agent is a Workplace object, so it can invoke methods on other objects, and is also designed to talk to other processes—in this case an application executable—via IPC. Workplace objects can tell the agent things that may need to be passed to the executable, and the application can in effect invoke methods on WPS objects, even though it cannot do so directly.

Another feature introduced in OS/2 Warp is the inclusion of DSOM. Whereas in SOM release 1 method invocations were restricted to objects in the same process, DSOM allows you to write objects in different processes to invoke methods on each other directly through the SOM engine. The SOM runtime included in OS/2 Warp and beyond has the workstation version of the SOM runtime, which means that this will work between processes on a single machine. There is also a version of SOM that will allow this same function across networked computers, although that version is not included with OS/2 (it can be licensed separately).

Using DSOM, you can accomplish the same functions as with agents, but without having to write a dedicated communications object class. You could put subclasses of `WPDataFile` in your application, and manipulate those directly along with other data file objects from the application running as a SOM client. This way you get the benefit of the application being in a process separate from the Workplace process for protection, but still have access to the power of subclassing the WPS class hierarchy.

Structurally, however, you should still keep the majority of your code out of the Workplace process for stability, and use objects to represent things the user will operate on, and the application is that operator.

Subset Function

Other features you may wish to add to your application are something I call *subset functions*. Subset functions are those that need the application (as the operator or manipulator of data represented by objects) to perform, but the user may not wish to view the object or manipulate the data.

Object-Oriented Printing

An example of a subset function is a printing function. When the object—say, a document—is dragged and dropped on a print object, the document object is called at its `wpPrintObject` method. The default or inherited method is that of a plain text or printer-specific file, since the superclasses of the objects are made very generic. You will likely have data files in some format specific to the application, so a plain text or printer-specific file print would not work.

Another factor is that by using the superclass's `wpPrintObject` inherited method, you are bound to the defaults set up for that printer. No customization is possible.

When you write your document object class, you should override the `wpPrintObject` method and have some interface to the application that starts only a subset of the application. This is the reason I call this a subset function. An example of a subset function is the object-oriented printing you have just seen.

An example of how to design this is that if your document object is called at `wpPrintObject`, you call `DosExecPgm` on the executable, passing in the filename to be printed, and some sort of flag to indicate that the user does not want to do anything with the data other than print it. As a response, the application will start and load the specified data file but will not show it in its main window as if the user wanted to update the

data. Rather, this flag will signal to the application to put up its print dialog.

You may not want to go even that far and instead just have the application read in the data file and print it to the printer specified, using the application defaults. My recommendation would be to put up a print window, however. If the user wants to accept the application defaults, one more keystroke or mouse click will not make a difference, and you will be providing the flexibility to allow the user to change anything he or she wishes.

Once the data has been completely spooled, the application terminates, where the user does not even know it ever started. The user simply dragged a file onto a printer, was asked to make sure this is the way it should be printed, and off it went. No more do users have to start the program, bring in the file, select to print the file, and then close the application. Now you're letting the users work with objects represented in the computer the same way they do with tangible objects.

You might say that starting the application is overkill. I agree with you. That is one way to do it, and I showed it to demonstrate an important point. That point is using DLLs to share code between processes. DLLs are not used just to write a series of applications like an office suite. How about if the code needs to be called from another process. A system process. Say, the shell process?

If you externalize your printing function to even the application and put it in a DLL, that print function can be called from anyone importing it. You can streamline the printing of an object with a custom data file format by putting the print code into a DLL. Let's look at how this might work.

Your print function is called MyPrint. This function resides in a DLL. When the user of the application asks it to print the current file, MyPrint is called from the application. It does not matter that it is in a DLL, the application linked with its `.LIB` file to be able to call the function. MyPrint has a set of parameters and when called from the application, an indicator in the parameters tells it so. It knows it does not need to open or load a file, it just takes the information from the parameters. Maybe it puts up the Print dialog box with printer properties to allow

the user to change any of the defaults. When the user selects the OK button, the printing begins.

Now look at it from the shell's side of things. A user has dragged an object of this application's data file class onto a printer object. It is invoked at `wpPrintObject`. Since it is an application-specific file format and not plain text or printer-specific, `wpPrintObject` is overridden in this class. The override will take some data from the WPS, such as the name of the file the object represents and the printer information supplied when `wpPrintObject` was invoked, and it will pass these things, along with a flag identifying the function call is coming from the shell, to MyPrint. MyPrint will do things similar to when called from the application, but notice that no additional process was started. The Workplace object class (yours) simply called MyPrint, because when you built the object class, you linked it with the `.LIB` file for the DLL that contains MyPrint.

Taking this one step further, recall that long jobs should be done on dedicated, or separate threads. When printing in an application, you may call `DosCreateThread` to start a printing thread. You could do that in your object class as well, because if you just called MyPrint from the object, you would be executing your printing function on one of the shell's threads.

Another way you could do this, which would only have you coding an additional thread once is to have the print thread created in MyPrint. This way, regardless of who calls MyPrint, whether it be the object class or inside the application, it will start up the print thread and return, where all of the dialog display, printing code and clean-up execute on this new thread. Now you've only coded the thread start up in one place, and you get the benefit in both. Of course when called from the WPS the thread runs in the context of the Workplace process, but you'd be doing that regardless of whether it was on a separate thread. If you are very concerned about the protection aspect of this, just use the first method of starting another process with `DosExecPgm`.

You can see how the extensibility of the system gives you the options, depending on how you want or need to reduce code size, code path, and general overhead and to protect from errant code. This is an excellent example of using DLLs to share code among processes.

To Write an Object, or Not to Write an Object

The real question is how far to go when writing an object, or if you should even write an object at all. If your application is something very straightforward, such as a text editor, the choice is simple. By using an assoctable and taking command-line parameters, you can accomplish all the things you need. Printing will work just fine, because you are using ASCII text files or in some cases, files with embedded printer control codes. If you are writing such utility programs as these, the shell already gives you most of the functions you need. That is the beauty of the object-oriented system. Inherit what you need; if you need nothing else, you're done.

If you are going to be storing files in some format other than plain ASCII or require more complex function, then a custom object, representing data files, is probably the right way to go. This gives you the flexibility of drag-and-drop functions and data communications, as well as everything the previous method gives you. Again, by using the object-oriented principle of inheritance, simply override the methods you need, such as `wpPrintObject`, and let the superclass's methods handle the rest.

I cannot stress enough that you should not implement your entire application as an object. There may be some cases where this is applicable, but for the majority of programs that will be written, this is undesirable both from an application standpoint and an overall system standpoint. Applications are really not objects. Documents, printers, FAX machines, and graphs are all objects and should be manipulated by "behind-the-scenes" application code. Computing is moving from an application-oriented environment to an object-oriented environment. Use the application as the operator and the object as the operand(s).

SUMMARY

By now you should have a good idea of how to design and map out the user interface. The first things the user sees are the objects and templates you create, either through assoctables or explicit shell calls

to create new ones. The user has several ways in which to interact with these objects.

Once the user opens an object, the application code comes into play. The window design should be smooth, flowing, and intuitive. Dialogs should not look cluttered, and you should ask the user as few questions as possible. Application defaults help you maintain user preferences and ask fewer questions per execution of a function.

The main point behind this discussion is: Keep it simple. Don't overwhelm the user with flash and glitzy interfaces; this only confuses the user. Use that time and energy to add the flash and glitz to the data you are displaying. Better to offer a 3-D view of data than to add in three more dialog windows of options on window colors.

As much as you may hate to admit it, it is not always function that makes or breaks an application in the market. The user interface—its power and ease of use—is what sells programs.

Hide the program from the user. The users do not want to see your application. They want to see their data in the way they are accustomed to looking at things. Give them new and exciting ways to view and manipulate their information to make them more productive. The user interface is a tool to communicate. Don't overshadow the function of your applications with a complicated user interface.

By using the power of the Workplace Shell you can create a powerful yet easy to use interface that is consistent with the rest of the operating system, making your programs look as if they are part of the computer system.

CHAPTER 12

Where's the Beef?

Now that the user interface of the application is mapped out, the next task is to plug in the holes with real worker code. The idea of mapping out the user interface first is to decide on how things will be presented. Now you must decide what the work inside will be to generate the data to be displayed.

Many decisions lie ahead here. For example, you must decide whether you will support only a specific version of the operating system because of available functions. You must decide whether you need custom device drivers and how tightly you will tie the application to device drivers (custom or not). You need to work out your applications file layout and whether you will be exploiting functions of a particular file system. How you will store application information is another important decision, as is the decision to use multiple processes or stay with one process.

This chapter will discuss the up- and downsides of the alternatives you have with respect to structuring the code.

DESIGNING THE CORE

In general, the user interface code has a large impact on how the rest of your code will be structured. Actions the user takes are communicated to the program through the user interface, which calls the worker code. This worker code can be structured in many ways.

You could decide to have the user interface be a process unto itself that ships work off to another process. This scheme can be useful if you want to isolate data manipulation from the user interface so that if the user interface code gets hung up somewhere, the data is intact because they are on separate processes. This idea has some merit, but the overhead required to do it is not worth the benefits in most cases. This may have limited usefulness, but in general this is not a wise scheme. One example where this is useful is the medical application example used earlier.

Another technique is to have another thread that acts as a work router. This would be a large piece of code that simply takes instructions from the user interface code and ships the work off to different pieces of code. With this scheme, you would have one large piece of code which is the user interface and another which is the worker code that crunches numbers and manipulates data. This scheme also has merit and is better than the previous one for general-purpose use. There are some drawbacks to this one as well.

As with any program, you want to provide the most function, but again, keep the code simple. By creating a large piece of code that performs many complex functions, you make it difficult to test and maintain. The other drawback is that it can act as a bottleneck and, to some extent, inhibit parallel operation within the application. You may say that you can design the code around this, but why? There is an even simpler way to structure your code.

The user interface code, and Presentation Manager in general, has modularized functions built in. Each user action generates an event to be delivered to the application. When this event is delivered, just act on it. There is no reason to funnel these actions any further. Presentation

Manager window procedures are structured such that each event delivered causes execution of some specific code. It is this code that can ship off the work to the code that crunches the data.

This scheme makes your code simpler, easier to test and maintain, and gives you maximum flexibility in structuring the modules, the control flow, and even the file layout such as DLLs or modules within the `.EXE` file.

Modularizing the "Worker" Code

The worker code is actually very simple to structure. For every user action there is an event. The response to this event is a call to some function. It is counterproductive to try to route the requests to another large piece of code only to have it broken out to the function that will ultimately do the work. The structure of the window procedure has already done that. All that needs to be done from the window procedure is to call the function(s) that does the work.

The events that must take place as a result of user actions are not always very atomic. That is, there is often setup that must occur, such as memory allocation, file opens or reads, or painting into a window. The event can be a request for a simple function such as a paint, but the complicated functions such as reading from a file are nothing to be concerned about with this scheme.

Recall that the OS/2 API function calls are really just coordinated sets of requests for kernel, CP, or subsystem functions and services. The same is true for these events. For example, let's look at a file open request.

The user selects to open a file. This is communicated to the application by a `WM_COMMAND` message, indicating that the menu item representing a file open has been selected. What does the application need? First, it needs to know what file to open. Then, it needs to read the file into memory and display the data. This is nothing more than a few somewhat more atomic requests. To satisfy this user-driven event, the application should first put up a dialog to ask the user which file to open. Once that is known, the application calls the OS/2 subsystems to give it some

memory into which to read the file. Then the calls to `DosOpen` and `DosRead` are made to read in the file, and finally the data is displayed.

This is a good example of how atomic you should get with the worker code. As a result of the `WM_COMMAND` message, the window procedure should call a "`FileOpen`" function. This function calls the `WinFileDlg` API to get the dialog for the file, calls `DosAllocMem` for the memory, calls `DosOpen` and `DosRead` for the file functions, and calls a data-painting or -drawing routine within the application code. The reason for the last step, rather than drawing the data right in the same code, is that the code that draws in the client area will likely be used every time the data changes. That is a common function that should live in only one place in your code. This makes maintenance easier and also reduces the size of your code files.

When working out where to place all of the various functions, such as the drawing function just mentioned, keep in mind that you want to maximize parallel processing in the application as much as possible. After all, you don't want the user to have to wait while you draw the screen. You want them to be able to execute other functions. A good candidate for a separate thread is the drawing function. The main thing to keep in mind is the same as with any reusable code in OS/2, and that is that it be reentrant. You could have several threads trying to get into the same code. This in and of itself is not a problem, but you need to ensure synchronous access to the data structures (presentation spaces, buffers, and so on). Some of this reentrancy will need to be synchronized or time-ordered for integrity's sake.

These ideas should not be news to you. This section (so far) is only meant to reinforce your understanding of how modularized OS/2 code is and how, although you will be dealing with parallel threads and other OS/2-specific functions, the basic rules of good programming still apply.

Memory Management

Even though OS/2 provides virtual memory, and applications can use much more memory than is in the system, it is still important to use

memory judiciously and efficiently. The more memory each application wastes, the slower the system becomes. You don't want to be known as the memory-hogging application.

Memory management can be an involved issue with OS/2. You can simplify it in several ways. The most common thing to do with memory is to allocate it as you need it, in blocks of the size you need. Under DOS, this was acceptable. Even under OS/2 1.x, where segments were anywhere from 1 byte up to 64K, this worked well.

32-bit OS/2 memory management is a different animal. All memory is allocated in pages of 4K bytes, whether you request 1 byte, 4K bytes, or 200K bytes. As you can quickly see, allocating small blocks of memory as you need them is not very efficient. The most efficient solution is to write a small memory management package for your application.

By having separate memory management for the application, you can allocate and use memory efficiently according to the needs of the routines requesting memory. Now this does not mean to go out when the application loads and allocate and commit 5 megabytes of memory just so it is available to you, because this will quickly lead to system thrashing, not to mention long application load time. This *does* mean to have a small set of routines that manage the pages of memory within the application. Allocating 5 megabytes at the outset may be fine if you choose to use sparse memory allocation with allocated but uncommitted memory.

With respect to code, the operating system (more specifically, the loader) manages the allocation of memory for code and its subsequent paging. You can do some work to optimize this, as you will see when tuning is discussed in Chapter 18. For data, however, that is completely up to you from start to finish.

Since all memory is dealt with in 4K pages, you should map out the memory usage of the entire application and localize memory references. What this means is that you should start out by mapping out the memory needed by the application. Block it logically into groups from the most used structures down to the least used.

For example, buffers that are used to read and write the data files are usually the least used. I say "usually" because there are some appli-

cations that are specifically designed to gather data and write it to a file on a real-time basis. In this case, the buffer is some of the most often accessed memory. Let's assume, however, that this is a word processor we are dealing with. In this case, the read/write buffers are the least used blocks of application memory. The work areas, however, where the data being manipulated is kept, are the most often used application memory.

Once you have this memory mapped out, decide (or at least estimate) what the optimal size of these various structures needs to be. Now you have an application memory map with usage and size mapped out. The next step is to categorize and *rate* the blocks from most often down to least often used.

Once this is complete, you can analyze the memory and allocate it efficiently, taking into account the size of the blocks (remembering that any allocation is at least 4K and is in 4K increments) and how often they are used. The goal is to keep the most often used blocks in the same 4K pages or grouped into several 4K pages and keep the least often used code in their own pages.

This will optimize your application for swapping/paging as well as allocation. By grouping into blocks you can optimize each 4K page, wasting as little memory as possible. You may even find that you have a couple of hundred bytes per page to give yourself some room to play later on.

The other more important benefit of this mapping and structuring is that by separating out the least often used blocks of memory, they can be paged out sooner since they are referenced less often. If these logical blocks were in the same pages as the most often used blocks of memory, they would be taking up real memory while not being used. This is wasteful within the system and your application. Of course, if you use so little memory, or you use most of the memory blocks equally as often, you may as well keep them together. After all, if you only need say 200 bytes for this least used memory and you have 1K left over in your most often used pages, then it makes more sense to keep them together than allocate a separate 4K page just for those 200 bytes.

Now this is a simple scenario, with localized data access. Many memory structures are implemented with potentially huge memory objects.

In older systems, all the memory needed to be allocated and committed in order to make this mechanism work. This left a great deal of memory consumed but not used. Using sparse allocation, you have even more freedom to manage memory within the application.

Sparse allocation is not only the attributes on the pages of allocated memory, but the other part is the guard page mechanism along with the per-thread exception handling. The fundamental part of sparse allocation is the allocation of memory without the commitment (physical backing) of it. `DosAllocMem` allows allocation without commitment. A subsequent call to `DosSetMem` can change the attributes on a memory page, including changing it to a committed page.

An interesting technique that was introduced in Chapter 5, is to allocate enough memory for the application but not commit it. The memory can all be marked with the guard page attribute. In this way, when the algorithms for address mapping points to and tries to touch a guard page, an exception is generated. By having an application-defined exception handler, the memory can simply be changed to a committed page when it is accessed. In this way, you can allocate megabytes of memory up front without hurting application or system performance.

OS/2 uses a similar mechanism to grow stacks. You can create a variation on this to monitor memory usage and growth within the application. For example, you can create an array or other dynamic structure at the "end" of a page on purpose. If, when the structure grows past the end of the page, the next page is a guard page, you can be notified by this exception handler. You can implement a "protection" mechanism within the application using this.

In any event, you must do some work within the application to manage memory efficiently. The benefit is that you have access to huge amounts of memory, better performance, and memory usage directly proportional to what the user does with your application. A good example is a spreadsheet program. You can have spreadsheets almost limitless in size. If the user wants to use cells down in row number 123456 and column number 4321, no problem. However, the user using spreadsheet with say, 200 by 300 columns will use less memory (unless of course the former puts information in cells sparsely rather than filling in all the

cells, thus demonstrating the further gains in sparse allocation). The only drawback is that your application must do some more work than just allocating memory and using it. A very good trade-off if you ask me.

Device Drivers and Device Independence

Quite often I have been asked, "Should I write a device driver?" There are only a few specific reasons to write a device driver for OS/2:

1. If you have a specific piece of hardware you wish to support (such as scanner, sound, or MIDI card)
2. If you must do work that can be done *only* at ring 0 (some direct hardware access) and there is *no* OS/2 API to accomplish the desired function

The reason why these are the only real cases to write code to run at ring 0 (which is where all device drivers on Intel run) is that you open the entire system up to any possible bugs in your code. You are also tying yourself to the hardware. Of course, if you are writing code to make your company's MIDI card work on OS/2, this is a perfectly acceptable thing to do. However, there are some people out there who want to write for ring 0 thinking that it will just give them better performance. Wrong.

It is true that ring transitions are somewhat expensive operations, but OS/2 is written to give applications excellent performance. By writing code to run at ring 0, you are exposing the entire system to anything you have in your code. Since ring 0 is the most privileged code, your code would be able to access any area of the system. Ring 0 is reserved for only that code that absolutely needs direct hardware access.

Reason 1, in the preceding list, is really about the only reason you would ever need to write a device driver. Reason 2 is there as a catchall for those of you who want to write special-purpose code such as debuggers, hardware extensions, or even extensions to the operating system, such as installable file systems.

In general, keep your code as isolated from the hardware as possible. If you do have to write hardware-specific code, keep it separate from

the other code. This will help you port your code to other platforms should you wish to do so. The discussion in Chapter 7 on the Workplace architecture underscores this point very well. Device drivers are for managing hardware. In the PowerPC there is no Ring 0. If you write a device driver thinking you will be more powerful or faster in Ring 0 you will not only be incorrect, but you will have to write a device driver task under Workplace just to look the same to applications!

If you are going to write an OS/2 device driver, first study the device drivers included with OS/2, such as the communications (COM) device driver. You'll not be able to see the source code for this driver, but you can look in the IBM technical reference books to see examples of how its functions are laid out along with its interface and calling conventions. Keeping in line with the system's conventions will not only make things more consistent for your applications that will interface with the device driver but will also help others to write code for it easily. After all, if you are writing a device driver to support your hardware device, you are just as interested in selling your hardware as your application, so the more developers writing applications for your hardware, the better off you'll be. You win in either case.

As for the application aspect of hardware-dependent code, keep these functions separate from the rest of your code; to really do it right, you should write your own APIs to access the hardware. Let's take this from the bottom up.

You will be writing a device driver to support some hardware. (Regardless of whether you actually will, let's just assume you are for a starting point. If not, the device driver described here is simply one of the OS/2-supplied drivers.) This device driver has a set of interfaces: the `IOCTL` packets described in Chapter 5. Each function has a specific purpose and tells the driver to perform an action and return a result code. This interface should be consistent with the OS/2-supplied drivers.

Now you need to interface with the device driver. For the OS/2-supplied drivers, there is most likely some API to perform the function you need. You see, the APIs are really just a coordinated way to send commands to the device drivers to act on the hardware on behalf of the application. You could simply write `DosDevIOCTLs` to instruct your device

driver to do the things you need. This gets a little messy and makes things more complicated and harder to follow. The right way to do this is to write functions to do the `IOCTL` work. For each high-level function, you should write a function or your own API.

Of course, many of the OS/2 API functions, such as `DosOpen` or `DosRead`, will work on many device drivers. There will undoubtedly be functions for which OS/2 has no API, or you may want to write some custom functions to maintain finer control over the device. If you keep these functions separate from the rest of the code, you are really now defining a new subsystem.

Now, instead of your application calling `DosDevIOCTL` all over the place, you can call functions much in the same way you would call standard OS/2 APIs. This gives you many advantages.

The first advantage is that the code you write that interfaces with the device can be placed in DLLs. Now, this common code can be accessed by many applications. You can publish the API interfaces to allow other applications to use your hardware and interface libraries. This helps you sell more hardware, because anyone can have access to your hardware through your high-level interface.

The next advantage is that your application will be easier to read, enhance, and maintain, since all you are doing is calling high-level functions from within the application code and keeping the hardware interface separate. This makes testing easier as well as debugging. If you know that the hardware interface routines are okay, you can concentrate on the application; or, if you want to update the hardware interface routines, you need only replace the DLL, not the whole application. This is also advantageous if you release different revisions of the hardware, or derivative products. With this interface library, you don't have to continually change the whole application (nor do others writing to your hardware).

The other, and in my opinion most important, advantage is portable code. By keeping the hardware-specific code in this separate package, you can write basically platform-independent application code. Of course, it is device-specific in the respect that it is written to use a certain hardware device, but the device is being accessed by function calls to

some intermediary routines. OS/2 provides hardware access via IOCTLs. Other platforms do it differently. If you want to write your code to run on different platforms, it is much easier to rewrite the internals of the interface library package than to find and change all of the hardware-specific code inside the application. Depending on the environment, it may be as simple as a recompilation of the application along with a new interface library.

By keeping the application as isolated from the hardware as possible, you ensure maximum flexibility and keep all your options open for expandability and portability. This means for you not only to write and use interface libraries for custom hardware, but also to use the OS/2 API wherever possible and, if there is some function you need that does not have an API, write your own but keep it separate. Who knows—maybe you'll come up with enough high-level functions to release your own "high-level language" interface package for OS/2.

FILE LAYOUT

Keeping modularity of function is important, for all of the reasons discussed previously, but just as important is the structure of your files. This does not necessarily mean your source code files (although they will have an impact too), but, rather how you will package your application. For example, should you use extended attributes? How about long filenames? Which code should be in DLLs versus in the main .EXE file? What about using the system's .INI files? Should you use your own .INI file? Should the program span processes or even sessions? These are big questions. The answers are very straightforward. I know that that word has been used a great deal throughout this book, but when you really look hard at OS/2, it is powerful and complex, yet very straightforward.

Once again, you see the phrase, "Keep it simple." You should be sure of why you are doing something before you do it. When choosing whether to support long filenames for HPFS drives, be sure you don't create incompatibilities for non-HPFS systems. When choosing how you

will store application defaults, think about the ramifications of using different types of files. If it does not make good sense, or if you can think of downsides without good reasons to incur those hardships, don't do it. The more complicated it is, the harder it is to install, use, and service. Even the simplest decisions, such as directory structures, have important impacts.

HPFS and FAT Features

HPFS files and FAT files are interchangeable. You can copy files between the file systems (actually, any file system within OS/2, as long as the user sticks to the OS/2 interfaces such as the `COPY` command), and several file systems can coexist on the same system. The main difference between them, aside from the disk layout (which incidentally is isolated from applications) and performance, is that HPFS supports long filenames—up to 256 characters, including some filename characters that FAT does not like.

When writing applications, choosing to use FAT vs. HPFS is really a nonissue. Some of the decisions about how you will accomplish other functions may rely on this decision, however. For example, since filenames in HPFS can be 256 characters long, you need to do some work to figure out how you want to save a file to a FAT drive that was read in with a long name from an HPFS drive. You also do not want to use only long filenames, because then you are excluding all FAT users.

The long filename issue is really not a big one at all. A more important issue would be whether you will use extended attributes (EAs) for your files. Recall that the shell uses EAs all over the place. HPFS handles EAs much better than FAT, because EAs were designed into HPFS from the start. EAs on FAT were an afterthought, since FAT has been around as long as DOS has—since the PC was introduced in 1981. The only real difference is that EA performance is slightly better on HPFS than on FAT.

One thing that the shell does that you might want to do yourself is that when a file with a long name is copied to a FAT drive by dragging the file between folders, the shell will truncate the filename to an 8.3

name and store the long version of the name in a .LONGNAME extended attribute. If your user asks you to save to a FAT drive a file that already has a long name, you might consider mimicking this behavior to be consistent with the system behavior.

Other than the EA issue and dealing with long filenames, there is really nothing much to the FAT vs. HPFS decision from a programming perspective. This has been a very common misconception in the programming community. Even if you are going to write file system utilities, the only difference is the absolute disk layout, which is generally protected from applications anyway. Standard file system calls or, for these utility programs, IOCTLs are about all you have to work with, so the interface is consistent.

The real questions come when you are going to lay out the files for your application. The .EXE(s), .DLL(s), data, and initialization (.INI, or system defaults) files, and any utility programs are the real focus.

.EXEs and .DLLs

In most cases, your application will have one main .EXE file. This is the main part of the application and the control point. It is from this .EXE that the first thread of the application gets created by the program loader and where execution begins. From there, it's all up to you.

Let's first look at why you might want to put functions in DLLs versus just writing one large executable file. In the discussion on hardware dependence, you have just seen one very good reason for writing some functions in a DLL. If you see you are getting too deep into device- or even operating system-specific code, such as DosDevIOCTL, you will want to look into making the code in the .EXE more generic and writing an interface package in a DLL. You could just write the interface or isolation functions and put them in the .EXE file, but why? The point of this exercise is to isolate the application code. You'd only be doing half the job if you left the interface routines in the .EXE.

Another reason for using DLLs arises if you intend to share code between applications or if your product is itself a set of library routines to some hardware, other software, or even just a set of runtime routines.

Just as with the DOS world, OS/2 is not perfect, and it does not provide every API that any programmer might like to use. It is perfectly rational to write a set of more powerful APIs to do more intricate function than the base OS/2 APIs do. As a matter of fact, that is what Presentation Manager really is: a set of routines that live on top of the kernel and the base subsystems and that provide the function of a graphical user interface. PM is really a bit more than that, mind you, but conceptually that is all it is.

Code can be shared among many types of applications. As you have already seen, there is the issue of specific hardware interfaces. As another example, if you intend to release a series of applications, there is no reason they cannot share code for common functions such as printing, installation, and utilities. Lotus Development Corporation has done a very good job of this with its spreadsheet and presentation graphics packages. If you have fixes to make, you make them once, and all of your applications benefit. This cuts development and maintenance costs as well as distribution costs.

Look back to the discussion in the previous chapter on subset functions and calling application code from the shell. This is an excellent example of putting code in a DLL to be shared among processes in which you are not writing your own application suite.

Another example of code sharing is new PM controls. You can register these new window classes from the OS2.INI file, and with the window procedures in a DLL these new controls are shared and available to any PM application. You can create a consistent look and feel for functions based on these new controls.

Moving code to DLLs is not a panacea, however; there are some drawbacks. The most obvious one is that there are more files on each user's disk to maintain. Also, DLLs are simply pieces of code sitting in memory waiting to be executed. They are not part of the `.EXE` file. Therefore, it stands to reason that they are in different pages. This increases the memory required by the application, and bringing this code in means more potential page faults.

This is not as expensive as it was in 16-bit OS/2, where every DLL reference was in a segment different from the `.EXE` and required a segment register load, but it is still more expensive than having the code

in the same page. You can use the same techniques as with .EXE files to pack code into pages (which will be discussed in Chapter 17) to make this somewhat more efficient.

If you only have one or two functions that meet the criteria for going into a DLL, it is probably not worth the effort. However, if you have several functions that you think you should isolate from the application .EXE file, a DLL is the way to go. Some programmers think that moving code from the .EXE file into a DLL will give them performance gains or let them use less memory. Although you can use runtime dynamic linking to delay code loading until it is needed, the OS/2 application loader does the same thing. With code that is properly structured (which will be covered in Chapter 18), you can enjoy the same benefits with much less work. If that is your primary reason for moving code into a DLL, read Chapter 18 before finalizing that decision.

INI Files

INI files are used by systems and applications to store important information such as user preferences or the last known state of the application. OS/2 uses two such files: OS2.INI and OS2 SYS.INI. These two files are used to store information about the last known state of the system, the system configuration, abstract objects, and other system data. There are also areas in these files set up for application use. Using a set of APIs, applications can write their own information into the INI files. Many application installation programs do just this. They not only copy their files to the hard disk, but they can also create an object for themselves in a folder or, in terms of 16-bit OS/2, create a program entry in a group.

Although applications can use the system INI files, I personally recommend against using them. Of course, I have several reasons for recommending this. The first is that the system INI files are intended for system use, not application use. Despite the fact that an application can gain access to the files, I think they should be reserved for system use.

The next and more important reason is that the user may wish to reset some system configuration information and choose to do so by reinstalling the original INI files (from when OS/2 was first installed) or

some other prior level of INI files. This effectively wipes out your application's information. Another reason is that if you do something wrong while writing the INI file, you can affect proper system operation. Still another reason is that the system INI files have a specific format and are in binary. This means that you are bound to the system INI files' formats and users cannot manipulate them easily. That is one of the reasons for making the system INI files binary: so users cannot inadvertently change them.

In the Workplace architecture, INI files cease to exist as a structure. All of the work handled by the INI files is now in the system registry. While all of your interfaces to the registry operate as if there were an INI file, the INI file construct is no longer there. This is an example of advancing the function while maintaining compatibility for prior-version applications. While these functions and interfaces to INI files will remain, you can see why you should use your own INI files rather than the system constructs.

It is perfectly acceptable, and I recommend it, to maintain an application INI file. This file can be any format you wish—even plain text. This is where your application defaults should be stored, along with any other information you see fit. This keeps you isolated from anything the users do with their system INI files; it limits the system INI files strictly to system information; and gives you the freedom to do whatever you'd like in there. It also is another place where you can isolate the application from the operating system.

The application INI file should be kept in the same place as the .EXE file and can be updated whenever you wish, such as when the application is closed or when the user asks that application defaults be changed.

MULTIPROCESS (MULTIPROGRAM) APPLICATIONS

A feature of OS/2 is Interprocess Communications (IPC). IPC is very powerful but can be somewhat expensive in terms of performance. Multiprocess applications are fairly rare, especially in 32-bit OS/2. When writing multiprocess applications, you need to evaluate why you want to use multiple processes.

In 16-bit OS/2, there were more reasons to write multiprocess applications. The segmented architecture and the 64K segment limit imposed limits on processes. There were limited numbers of file handles, threads, and other resources per process. If you wanted more, you had to go to another process and write code to allow them to talk to each other.

Under 32-bit OS/2, virtually all of the resources' per-process limits are the same as the system-wide limits, so if you run out of resources, you have a lot more things to think about than going to another process. Multiprocess applications do have a place and supply an important function.

Recall that protection is done on a per-process basis. Any thread within a process can access any of that process's resources. Therefore, any protection between threads of an application must be done by the application itself. There is also the problem of a thread taking down the process. If any thread in a process goes down with a protection violation, for example, the whole process is killed by OS/2. For some applications, this must not happen. Think back to our medical application example.

To solve this problem, you can implement another process. That is exactly what was done for the Workplace Shell. Any bad object can take down the shell; however, the shell is implemented such that there is another process that sits idle, watching over it. This process consumes no resources until the shell comes down for some reason. This process (actually, the thread within the process) wakes up and restarts the Workplace process. Since the Workplace is a separate process, this "watcher" process is not affected if the shell process goes down.

If you have functions that you want to protect from other parts of the application, first look at protecting them with semaphores or other coordinating tools provided by OS/2. If those are still not sufficient for the security or consistency you desire, then another process is the logical choice.

You should try to avoid multiple processes for the simple reason of overhead. It is also more expensive to talk through IPC rather than just have the threads share data or other resources. You can see the benefits of using multiprocess applications in those situations where you cannot afford to have some piece of the application go down even if another part does.

SUMMARY

The big decisions are now made. Along with your user interface, you now have the main "block" structure of your application. You have seen how to make efficient use of memory, device drivers, files, and processes. You've also seen how to isolate your application from the underlying hardware and even the operating system, while retaining the power and flexibility you need, by designing your own interface routines.

Your "black box" design is complete. Now it's time to get into the real low-level design and development. In the following section you will see how to structure your development environment (which is much more important than it sounds on the surface), prototype the designs you have been working on, and get the code written and tested, while leaving room for change and future enhancements.

SECTION IV

Making It Happen

In this section we will discuss writing the code. Until now only the architecture and black-box functions have been examined. At this point the framework for a powerful, flexible application has been constructed. Now comes the job of setting up the development environment, writing some code, working with advanced functions, tuning the code, and tailoring it to your specific needs.

The first thing to do is define the basics of the development environment. This does not sound like something you should even have to think about, but it is. Many days have been lost due to inappropriate development environments. I'm not talking about compilers or tools, but the basics, such as the tree structure and source code control.

Once the development environment is set you can begin to write your code. By following the modular designs outlined previously you can begin to establish the user interface as well as the core code. Unit tests on all these pieces can occur in parallel until you are ready to start putting some of them together.

This section also discusses code change and how it relates to the design of an application. No matter how much you design before you

code, you will inevitably find holes in your design. You can choose to ignore them (not the wisest move), fix them on-the-fly, or, depending on their severity and scope, take the flawed part of the system back to the drawing board.

This section will help you turn your designs into reality.

CHAPTER 13

The Development Environment

The development environment is just as important to developer productivity as the compiler and other tools and just as important as how you have modularized your code, isolated it from the hardware, and kept it portable. Every programmer has his or her own favorite way of developing and testing code. Your job is to design not only the application but also the environment in which it will be developed to make the most of the programmers' talents without restricting their freedom. It is also extremely important to understand and acknowledge the future of your product. You must define how you intend to service it, release fixes, and release updates and future versions. These factors also affect how you will develop the code and control changes throughout the life of the product.

The first thing to keep in mind is that you want to be able to control the source code so that you can maintain integrity throughout the development and testing cycles. You also don't want to impose so much control that you inhibit a developer's ability to do his or her job.

SOURCE CODE CONTROL

Source code control has always been a topic of interest among developers. Most developers hate having to deal with it but love it when they have to roll back some changes or need to maintain multiple levels of a file.

There are many source code control systems available. Some run on the PC platform under OS/2, DOS, or even UNIX. Some control systems run on minis or mainframe systems. Your choice of a system should be based on what it does for you. Don't choose a system simply because it is the most popular or the least expensive. The most popular system may not have a key feature you need, and as far as the expense of a system is concerned, you usually get what you pay for. Evaluate what you need, and find what fits.

Platform

The platform you choose is not as important as what the source code control system does for you. There is nothing wrong with developing PC software with a mainframe-based source code control system if such a setup provides the control function you need. The only real consideration in your choice of platform is that it must somehow interface to your build environment. If you use a PC-based system the interface should be simple. A LAN would be sufficient. However, if you use a non-PC-based control system you have several choices.

The first option is to use a build environment equivalent to the control environment. For example, if you need to use a mainframe-based control system you may wish to go with a mainframe-based compiler. Of course, mainframe-based PC compilers are not quite as plentiful as PC-based ones, and they don't boast all of the features. Another option is to set up an interface between the mainframe source control system and a PC build environment. This solution adds a degree of complexity, however, since all current source and changes have to be shadowed to the PC environment, but the advantage is that your users (developers) will

be able to build and test code in a consistent environment. The other part to this equation is that you should ensure you have a PC-based interface to the source control system.

Developer productivity depends on an easy-to-use environment. The source code control system should be simple and unobtrusive. Once it becomes work, it's too much.

Function

The determining factor for which source code control system you should use must be the features and functions it provides. You need to examine how you intend to develop, maintain, test, and release your code. Not only do you need to consider this for when you develop the product, but you also need to look at how you plan to work on future versions and provide fixes to current versions.

All source code control systems perform the basic function of co-ordinating changes in the code during the development of a product. The first thing you need to look at is how the system accomplishes this function. The simplest systems simply ensure that a single file is checked out to only one user at any given time. When build time rolls around the development tree is simply frozen, snapshot, and then built.

With the more primitive systems, once a change is made to a file, that's it. If a change needs to be rolled back, it must be done manually by editing out the changed lines. More complex control systems, however, work on revisions of files and can roll back some changes. With these more powerful systems, all you need to do is specify a revision level of a file, and the system will reconstruct it for you.

An often neglected aspect of source code control systems is the future of the product. The vast majority of source control systems deal with a single release very well. However, if you wish to develop a derivative release or add-on to the product while maintaining the original code, you'll have to clone the original code and create a whole new system.

An important consideration in your choice of control systems is the ability to handle multiple releases of the software. The ideal way to handle this situation is for the system to maintain multiple levels of

the same file for different releases. Not only would the system handle revision levels of the file, but you'd also be able to pull the latest files for, say, release 2.0 of the product while someone else (say, someone doing service) could pull the latest code from version 1.0. The source library would be the same, and the control system would take care of handling the change control within and between versions.

Problem Tracking

Another important (but not critical) aspect of developing a software product is the problem-tracking system. Some projects have such a complex tracking system that the developers spend more time trying to work through the system and track problems than they do writing code. I won't try to tell you what functions you need in a tracking system. I've actually seen products for which the level of problem tracking involved nothing more than slips of paper. Other products use a tracking system in which you can generate reports of how many lines of code were changed or how many lines of code a particular developer touched on a particular day.

My view on problem-tracking systems is that they should (1) be able to track a problem from opening to closure, (2) provide basic reporting on how many problems are open, closed, and so on, and (3) be an integrated, almost natural part of the development environment.

The problem-tracking system should be well integrated into the source code control system. Of course, this is not a hard and fast rule, but just as if you had a control system on a mainframe and the build environment on a PC, there will be more work to do at each step of development and more interfaces to be built between them.

The real point here is to be sure that you consider all your functional requirements first. It may be worthwhile to build an interface between a mainframe and the PC to be able to "version" files the way your software strategy requires. Ease of use is a major requirement, as is the platform. Of course, you won't buy a mainframe simply because a great software control system runs on it, but you must keep everything in perspective and not limit the developers; however, don't let the code get out of control, or else the whole project could fall apart.

TREE STRUCTURES

Another important yet often overlooked part of a development environment is the tree structure of the code. Of course, this is completely your decision, but there are some things you can do to make life easier in specific areas. The first thing to do is look at how (or if) you plan to create your product in several languages. OS/2 has several tools to make this easier. The main tool is the message file; the other is the resource file.

First, get used to the idea of not using string literals in your code. You should move *all* translatable items out of the code source files. They should reside in string tables in the resource files or in the message files. This way, all you need to do is translate these files—you won't have to touch the source code. You simply build these translated resources into the executable code for each language you plan to support.

You may wonder what this has to do with tree structures. The details of translated versions will be covered in more detail in Chapter 17. The tree structure can help you work with translated versions more efficiently. You can use environment variables (assuming a PC-based build system) to point to the tree with your translatable resources.

Figure 13.1 shows an example tree that you can use. To accomplish the task you would set up an environment variable called `RESOURCES`. Inside your `MAKE` files and build system, you would use this environment variable to point to the resources. One key is to treat the U.S. version just

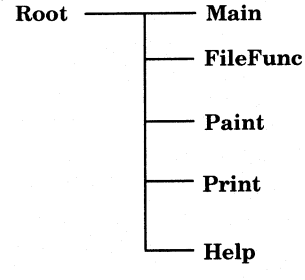


Figure 13.1 Sample source code tree.

like any other version. Don't make the U.S. version your primary one with the versions for other countries optional. By treating the United States on equal footing with the other countries, you will make translation easier and faster. For each language version, all you need to do is change the RESOURCES environment variable to point to the resources for the appropriate language. Other than that, your tree should basically look like the program. You should separate components into their own directories as well. A good rule of thumb is to have a separate directory for each loadable module: DLL, EXE, or other file such as a font file. This helps keep the source in order and close to the files that it creates.

TOOLS

The tools you use are, of course, up to you. Your choice of tools should take into account their ease of use and the flexibility they provide. The most important feature, however, is the code that is finally produced. The only way to write totally optimized code for any platform is to write it in assembler. Of course, doing so means losing all the control, readability, and portability of a higher-level language, such as C or C++. The compiler and tools you choose should generate optimized code.

Compiler technology is continually advancing, and the object code that new compilers produce is continually improving. You need to choose a compiler vendor carefully because you are betting your product and your future on your decision. Take into account the optimizers, code checking (ANSI, for example), and speed of the compiler itself. The surrounding tools are also important. Some of the more basic programming tools such as GREP, AWK, and MAKE should definitely be considered. They add to programmer productivity and make code easier to write. You may even want to write some source generators for AWK, SED, and GREP. For example, you can keep tables that are pumped into SED and AWK scripts to generate some of your code, especially MAKE files to build only the necessary code.

SUMMARY

Although the source code, tree structures, tools, and development environment are usually by-products of research and development, they are important aspects of your application design that should be planned out.

You won't ruin your application if you don't plan out the development environment, but with just a little work and forethought you can cut weeks or even months off your development cycle.

CHAPTER 14

Prototyping the User Interface

Chapter 11 discussed the design of the user interface in depth. Everything—from which controls you will use in your main window to the dialog structure and layout, along with Workplace Shell interaction and PM Interprocess Communication—should be laid out at this point. Now comes the time to write the code.

The first step in code development should be to get the user interface up and running. The reasoning behind this is that you can get the application running from a global standpoint more quickly than if you simply wrote the worker code and left the user interface for last.

It goes without saying that you want to write the user interface code at the same time that you write the code that does the real number crunching, but you should not place a higher priority on either. By quickly putting into operation the user interface you can begin the real testing of not only your code (to verify that it matches your design) but also the design itself.

It is very easy to put a design on paper based on reports, studies, and analysis. However, until someone can actually sit down and use it, your "good" design is just theory. You know that the code that does the real work under the user interface will do what you intend, and you also know that as long as it does its job the user won't care how it looks or works. However, since the user interface dictates what goes on with the worker code, it is important to make the design and function of this interface stable first. Changes in the user interface can cause you to make changes in the worker code underneath. By getting the user interface prototyped and into live usability testing early, you can avoid major rewrites later on.

In addition to getting usability testing underway, implementing the user interface early gives you the advantage of testing the application as a whole (system testing the app) earlier than if you wrote the code from the inside out. In doing this, you will be able not only to see what the overall product will look like but also to fill in any holes in the set of functions you are providing; you may also see something that would add to the application. By viewing it from a user's perspective early in the development phase, you will come out with a better overall product in terms of functionality and usability.

Using one of the CASE tools discussed in Chapter 11 will help you get the prototype up and running quickly.

PAINT YOUR WINDOWS

The first step in generating the user interface is to create and paint the main window of the application. This is the standard window with which the user will interact. The first task is to get the window created as you need it and then get the menu going. Remember the discussion on language support. Don't hurt yourself by hard-coding strings into the code, even just to get it running. Set up your string tables, message files, and other text-based resources early. You can either do it right now or do it over later.

The basic PM skeleton code is put in at this time. This is the code that calls `WinInitialize`, creates the main message queue for the application, and creates the initial window. The message-processing loop for the application also takes place at this step. Don't bother with any other initialization code yet. At this point you should be concerned simply with getting the building blocks set up. Just get the window created.

The other part of this initial step is to set up the window procedure for this main window. Listing 14.1 shows the most fundamental window procedure possible. From there, all you need to do is to `CASE` the functions and message you are interested in. Don't worry about threads yet; they're in your design and, with the techniques that will be presented shortly you can add them to the code afterward. Since you've already blocked out the functions that need to be performed to accomplish the application's job, and you've already looked at what can and cannot be run in parallel, you're all set to put them on threads at the right time. At this step you're just going to get some fundamental building blocks in place, and you'll add in the important threaded code in a moment.

```
MRESULT APIENTRY MyWindowProc(hWnd, msgid, mparam 1, mparam 2)
HWND     hWnd;
ULONG    msgid;
MPARAM   mparam 1;
MPARAM   mparam 2;
{

switch(msgid)
{
default:
return(WinDefWindowProc(hWnd,msgid,mparam 1,mparam 2));
break;
} /* end switch */

}
```

Listing 14.1 Skeleton window procedure.

Listing 14.2 shows a window procedure that handles some of the more basic messages such as `WM_PAINT` or `WM_COMMAND`. As you can see, this window procedure does nothing as a result of these messages. For each message you will likely call a function to perform the work for that event. As you will see later in this chapter and in further detail in Chapter 19, the work can easily be moved to another thread.

It must be stressed, however, that moving functions to other threads is *not* simply finding functions and moving them to other threads. The job of each thread and which code will be run in parallel are already defined. The mechanism to accomplish the parallelism is discussed here. First, just get the foundation of the startup, message processing, and clean-up code done.

Begin by putting enough painting code in your window procedure so that all the menus show up as well as the background of the client window. Add in the code that creates and processes dialog windows. This is not as simple as it sounds. You should put in not only the code that creates and shows the windows but also the code that paints the controls.

Now that the windows and their "subwindows" and controls are in place, put in the `CASE` statements to respond to messages in the window procedures. A technique I like to use to verify that everything is working as intended is to write a function called `NotHereYet`, which takes a string as a parameter. The function's sole purpose is to display a message box that tells the user that the action on that message has not been implemented yet. The string parameter is simply a short description of the user action that caused the message.

As you write the worker code, the calls to `NotHereYet` will be replaced with the real calls to the worker code. Again, don't concern yourself just yet with the multithreading aspect of the code. All you've really done so far is to put in the initialization code, the message-processing loop, and the window procedures to manage the behavior of the windows and controls.

At this point your windows show themselves and all of their controls. All of the `CASE` statements to respond to controls such as pushbuttons and

```
MRESULT APIENTRY MyWindowProc(hWnd, msgid, mparam 1, mparam 2)
HWND     hWnd;
ULONG    msgid;
MPARAM   mparam 1;
MPARAM   mparam 2;
{

switch(msgid)

}

case WM_BUTTON 1 DOWN:
 DosBeep(500,700);
 return(TRUE);
break;

case WM_CHAR:
 Look at the flags in the parameters and act on any
 keystrokes you are interested in
break;

case WM_COMMAND:
 Look at the parameters in the message to see which menu
 item was selected and do something as a result
break;

case WM_CONTROL:
 Look at the parameters in the message to see which control
 is sending you this message and act accordingly.
break;

default:
 return(WinDefWindowProc(hWnd,msgid,mparam 1,mparam 2));
break;
} /* end switch */

}
```

Listing 14.2 Basic window procedure.

listbox selections are in place, but they simply display a message stating that they haven't been finished yet.

So far, the data manipulation and display have not been considered, nor have the function of and interaction with the shell or application initialization file. This is the work that you will move to other threads.

MULTITHREADING CONSIDERATIONS

Let's step back for a moment and look at the role of threads in executing window procedures. Recall the discussion in Chapter 5 with respect to the relationship between processes and threads. The process is the owning entity. The thread executes. The window procedure can be viewed almost the same way as a DLL. It is a piece of code that sits there and is called upon only when needed: when a message is sent to the window procedure. A message is sent to the main window procedure every time the user takes an action on the window, such as a button click, keystroke, or even a mouse movement.

Which thread is used to execute in the window procedure's code depends on how the message is sent. You saw in Chapter 5 that the input router places user input messages on the application's message queue. What happens after that is up to the application. The user input message is posted. The difference between posting a message and sending it depends primarily on whether the action should be asynchronous. Sending and posting messages also indirectly describes which thread executes the code in the window procedure. Let's trace a user input message by looking at the threads involved.

When the user executes an action, an event is placed on the system message queue. It goes through the router and ends up on the application message queue. At some point, when the application's main thread (the user input thread) gets CPU time, the message is removed from the queue. The message dispatch loop is entered, and `WinDispatchMsg` called. `WinDispatchMsg` has a twofold function. First, it takes the QMSG structure taken from the queue and breaks it into the four items that make

up a PM message. Coincidentally, these are the four parameters that are passed to a window procedure.

Once it has separated the message components, `WinDispatchMsg` (actually, the thread executing in `WinDispatchMsg`) calls `WinSendMsg`. `WinSendMsg` causes the thread to make a call into the target window procedure. The thread executes inside the window procedure; when the window procedure encounters the "return" statement, the thread returns back into the message-processing loop. If there is another message on the queue the process repeats. This continues until either there are no more messages on the queue, at which time the thread gets blocked, or the application receives a `WM_QUIT` message, which causes it to drop out of the message loop and terminate. This is shown in Figure 14.1.

As you can see, user input messages are processed on the thread of the application. However, there are more ways to get messages to windows than having them come from the user. Although PM is not completely object-oriented if you look at it from a purist view, it is object-based. Each window is an object, and its associated window procedure is the class definition with each `CASE`, the methods. As such, the only way to get windows to do anything is to send messages to them (as with any object).

Applications, or windows, can send messages to other windows or even to themselves. Now you can begin to see how complex the message passing within PM can get and how you need to understand which actions generate others which generate others and so on. With respect to threads, you have already seen how a user input message is posted and subsequently acted upon by the application's own thread. Any posted message is processed by the target window's thread.

Messages can be posted by the system, as in the case of user input messages, but applications can also call `WinPostMsg`. These different means of transmitting messages demonstrate the difference between synchronous and asynchronous messages. Messages are synchronous or asynchronous with respect to the originator of the message. When a message is posted, the originating application, when it calls `WinPostMsg`, places a message on the recipient's application queue and continues on.

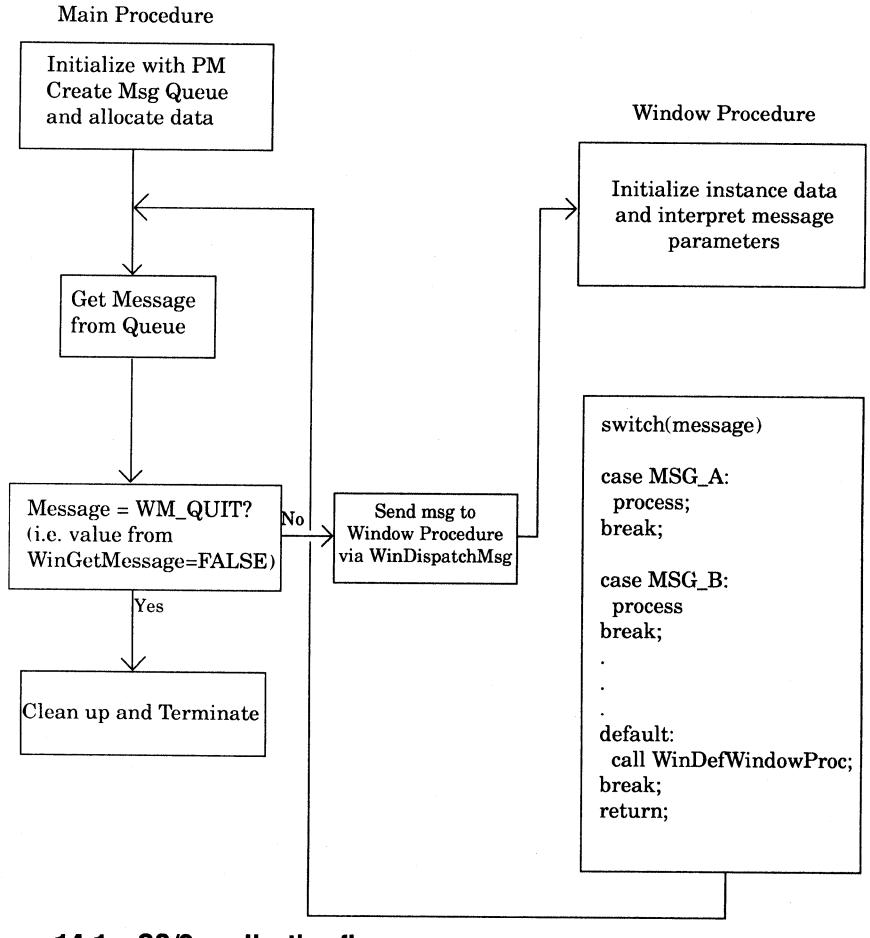


Figure 14.1 OS/2 application flow.

Posting messages is an asynchronous event. Sending a message, on the other hand, is synchronous with respect to the sender. When a piece of code calls `WinSendMsg`, the thread executing that code makes a call into the window procedure for that window. You can begin to see the difference. When the thread calls `WinSendMsg`, it is jumping into the code for another window procedure. This can be any window in the system.

Now take this a step further. As a result of a message, some windows will send a series of messages to themselves or other windows. Some

will call some APIs. However, the `WinXXXXX` APIs really just represent a system-defined, coordinated way of sending messages to windows. This thread, just like any other that calls `WinSendMsg` (even the one that calls it via `WinDispatchMsg`) will jump all over the system. Think about it in terms of your application. Your thread, the one by which you are simply sending a message to another window, is, as a result, executing all over the system. You must decide if you can post the message or send it. Remember, if you send a message as a result of receiving one, your application's thread will be waiting for the target window procedure to complete the appropriate action in response to your message instead of processing your user's input.

Handling Long Jobs

Sent messages are executed or responded to on the thread of the sender, whereas posted messages execute on the thread of the target window's application. After careful consideration you can see why applications must adhere to the one-tenth-of-a-second rule, which says that all processing in a window procedure (that receives user input) must be completed within one-tenth of a second. Of course, there is no real way to measure this, so I usually just say, "Do it quickly." If it cannot be done quickly, it should be on a separate thread.

If it is a function that cannot be "paralleled" with other functions, then you have two options. The first option is to change the pointer to the clock (`SPTR_WAIT`) while the processing is taking place. This tells the user that he or she cannot do anything else while your window procedure is busy, but the user is also locked out from doing anything else in the system for the moment. Another problem with this approach is that if there is a problem with your program you will lock out all user input. Multitasking can still continue, but the user cannot interact with the system.

The other, more desirable option if you cannot perform the task on a separate thread is to disable certain functions within the application and run the task on a separate thread anyway. For example, you can easily store application defaults in a file on a separate thread without inhibiting

any function of the application. However, you can't manipulate a data file until you have finished reading and interpreting it. In this case you would disable certain menus until the operation is complete. The code that handles your user interaction should always run on its own thread.

For example, the first thing that the thread that reads the file should do is disable the proper menus. It should then perform the operation and finally reenable the menus. You will find that there will be only a few functions you can execute completely without worrying about some other function the user could choose to request. This method will be what you use in most of your threads.

Using PM's Thread Handling

You've seen how thread interaction affects not only your application but also all of PM and all applications running in the system. So far you've been told to use separate threads for longer work while the main thread remains dedicated to servicing user input. This section will explain how to do this. There are many ways to use threads in PM applications.

To start with, the threads must be created. This can be done at either the start of the application or dynamically when the threads are needed. Since thread creation is not the cheapest of operations it is advisable to create the threads at the earliest point possible. This will of course slow down the application load time, but if the time has to be spent somewhere, it's better to have a longer load time than to force the user to wait while executing an operation. You could also use the menu-disabling technique to enable functions one by one as the application and all of its threads come to life.

You also have some choices with respect to managing the threads. You don't want to write code that loops while waiting for work to do. The obvious answer is to write code that uses semaphores to execute code and to block when there is no work to do. It is the responsibility of your other code to clear the semaphore and inform the thread of the work it has to do. This coordination must all be originated by the window procedure. After all, this is where all events go, and everything in the application is triggered by an event.

Object Windows

Think back to the message-processing loop described in Chapter 6 and analyzed at the thread level earlier in this chapter. You can use this mechanism to manage work for your entire application. An important fact about windows is that they do not have to be visible. As a matter of fact, the construct called the object window is invisible. The object window and any of its children are invisible. Because it is a window, it has a window procedure, as do its children. Using this fact, you can combine your knowledge of the message-processing loop with how it interacts with threads and events to manage all of your work when it comes to semaphores and threads. Your application will have visible windows that interact with the user and other windows to get and display data. When it comes time to send work to other threads, why not just let invisible windows do the work? This is best explained with an example.

Aside from the windows that show data, you can create one or more windows that are children of the object window. (The owner is `HWND_OBJECT`) Because these children of the object window are never visible, they will never receive `WM_PAINT` messages. They also will not receive the standard messages from user input such as `WM_COMMAND` or `WM_CHAR`. Why, then, create this object window? The reason is that it has a message-processing loop. This loop is a natural manager of work and threads. This particular message loop will be on a thread separate from the user-input message loop's thread. Messages can be sent or posted because there is a queue to get the message. If there are no messages on the queue, the thread managing that queue is blocked. If a message is sent to the window it executes on the thread of the sender.

For the work that you want to execute in parallel to other work, or to let the main thread of the application be free to respond to user input, just post a message to this other (object) window. The posting thread (the user input thread) will be able to continue on in parallel, since it posted (not sent) the message, and the thread managing the object window's message queue will get it and process it on its own thread. The message can be a PM-defined message, but it is more common to use specific user-defined messages for this kind of work.

There is a value defined in the header files called WM_USER. All values below this are reserved for system use. WM_USER above are for application-specific messages. By simply posting, say, WM_USER+1 with the parameters specific to your message, you can activate this object window thread. Since you have posted it, the main thread can come back to the user while this object window thread is off doing the work.

When the object window is done doing the work, it can post a message back to the main window to signal that the work is finished. Of course, you could do all of this with a common data structure for information and a semaphore, but PM can do it for you. This way you don't have to worry about the state of the semaphore or who else can touch it or any deadlock problems.

The object window does not really behave as most windows do. It does not receive any user input. You can read files, paint the main window (which is usually the most time-consuming operation), or even perform long calculations on these separate windows. Additionally, each function, or black box, in your design does not have to have its own window. Because of the SWITCH structure of the window procedure, you can pack many tasks into one object window's procedure. The number of window procedures (and object windows) you need depends on the thread design you devised earlier. This is where you will take the list of functions that cannot be run in parallel and put them into one thread, or window procedure in this instance. You can see how this naturally leads to coordination, since this object window is never sent messages; it only processes posted messages on its own thread. Therefore the window procedure will have only one thread executing in it at any moment and forces the fact that none of the functions in the CASE statements within it can run concurrently.

An important decision when using these kinds of threads is to decide which ones will have message queues of their own. Some PM APIs require a message queue for the thread that calls the function. If a thread is going to process messages posted by another, it must have a message queue. However, it is vital that if a thread has a message queue but does not process user input it must call WinCancelShutdown. The effect of this is to make sure the system does not send the thread a

WM_QUIT at the time the system is shut down. This is important because you want to be able to control the termination of your own threads when the application shuts down; you don't want the thread to drop out of its message loop when the system begins its shutdown. This could cause disastrous results if the thread was in the middle of a critical operation.

A good example of using these threads and ensuring that you call WinCancelShutdown is when dealing with object windows, which are always invisible and should obey the ideas first outlined.

User-Defined Window Messages

The messages sent to windows by the system are predefined. Earlier in this chapter you saw the WM_USER value. This value marks the end of the system-defined messages and begins, as the name implies, user values. All of the message identifiers are numeric values represented by such names as LM_INSERTITEM and WM_CHAR. These are defined in the OS/2 header files, as is WM_USER. These names make reading and writing code easier. Just as you would not want to be looking at code with such messages as 0x0051, you don't want to look at such messages as WM_USER+1 or WM_USER+5.

In your application's header files you can set up defines for your own messages. You don't want to conflict with OS/2's predefined messages, but you can define your own based on WM_USER+x. Listing 14.3 gives an example of defining your own messages in a header file.

A convention for using such messages is to take the WM_USER values and simply give them names of their own. Of course, you don't want to risk conflicting with system-defined messages in your own code, so

```
#define UM_OPENFILE             WM_USER+1
#define UM_READFILE             WM_USER+2
#define UM_PAINTMAINWINDOW      WM_USER+3
#define ON_OPENFILEDONE         WM_USER+4
```

Listing 14.3 Defining user messages.

I suggest using UM_ for your own messages (where UM stands for user message).

You can also use these values for the notifications that object windows post back to the main window to indicate that work is completed. You can stick with the UM of messages for everything, or you can use the example in Listing 14.3, which shows an ON (object notification) message as another naming example.

WinPostQueueMsg

Another technique similar to that of the object window uses the message processing and coordination of a PM message queue, but does not incur the overhead of an object window. The same premise applies.

For your other worker threads, you will call WinInitialize and WinCreateMsgQueue, just as you would for an object window. However, instead of calling WinCreateWindow to create a child of the object window and then going into a WinGetMsg/WinDispatchMsg loop, you will call WinGetMsg and immediately drop into a SWITCH statement on qmsg.msg (the message piece of the qmsg structure). What comes into WinGetMsg when it gets the message from the queue is a QMSG structure. As you have seen, WinDispatchMsg breaks up that structure into the four widow procedure parameters and then calls WinSendMsg. You will break up the four parts yourself and operate on them directly.

Really, what you're doing is putting the SWITCH statement that would have been in the window procedure right into your WinGetMsg Loop. Listing 14.4 shows an example of a thread set up to use WinPostQueueMsg.

By using WinPostQueueMsg, you don't incur the overhead associated with object windows (even though the limits on window handles is much higher than ever before) and your code path is shorter and the code smaller, since you are not managing a window, but you are using PM's message coordination to free you from having to deal with event coordination semaphores and shared data areas needing to be protected.

Both of these options allow you to group the set of functions for one thread in that thread, with each being performed in a different CASE statement in the switch. Now you are starting to see how you can take

```
void  MySecondThread()
{

HAB hAB;
HMQ hMQ;
BOOL rc;
PQMSG pqmsg;
QMSG qmsg;
HWMD hwndSender;

hAB = WinInitialize(NULL);

hMQ = WinCreateMsgQueue(hAB, LONG(0));
pqmsg = &qmsg;

/* Don't forget that since this is a message queue that should not receive */
/* user input and we don't want the system sending us WM_QUIT messages     */
/* on shutdown, we need to call WinCancelShutdown                          */
rc = WinCancelShutdown(hMQ,TRUE);

while WinGetMsg(hAB,pqmsg,NULL,NULL,NULL)
    switch(qmsg.msg)

    {
      CASE UM_OPENFILE:

        open the file
        WinPostMsg(hwndSender,UM_FILEOPENDONE,NULL,NULL);
        break;

      CASE UM_OTHEROPERATION:

        do other operation
        WinPostMsg(hwndSender,UM_OTHEROPDONE,NULL,NULL);
        break;

      default:
        do some default thing in case I got a message I don't know
        what do do with, but I never should....

        break;

       .
       .
       .

    /* End Switch */
     }

  DosExit(0);  /* If I got here, I got a WM_QUIT, so end this thread */}
```

Listing 14.4 WinPostQueueMsg thread example.

one set of functions that cannot be run in parallel and group them in one thread, another set that cannot be run in parallel with respect to each other but can be with respect to the first group in another thread and so on.

You will use the main window procedure as a work dispatcher, and ship off work to these other threads by posting messages, and have them post a message back when they are done. You can now see how this is all falling nicely into place.

Keeping the User Interface Thread Responsive

The biggest issue in multithreading your PM application is to keep the user input thread responsive. A single-threaded application may be a bit slower than a multithreaded counterpart, but another major concern is that a single-threaded version will hold up user input throughout the system. A multithreaded PM application will also hold up user input if not implemented correctly.

As you have seen, if the user input thread is also used to do all of the work within the application, it will clog up the pipeline that represents the system input queue and input router. (Note that in OS/2 Warp for the PowerPC this will not hold up the system but will hold up PM applications, because the event/session manager manages the system, but for compatibility with applications, the OS/2 personality operates with a single queue once it gets the input from the event/session manager.) Many applications that have been ported from other platforms, such as Windows, will likely be single-threaded, because they (like Windows) have no concept of threads. However, you are reading this book because you want to design good applications and do it right the first time.

The entire key here is to use the main window procedure, the one that gets user input, as a router, and do all of the work on the other threads you've just set up. Although we're only designing the user interface at this stage, you can see how, when we get into the real multithreading of the code worker code in the next chapter, you will manage these threads easily and with a minimum of code.

By using the message-passing mechanism and thread management of PM you can make your job and your code much simpler and more efficient. After all, this PM mechanism is tested and easy to use, and best of all it's already there.

SUMMARY

The user interface should be the easiest part of the code to get running. My opinion is that it is also the most fun part of an application to code. By getting the user interface going early you can test the interface and your designs on "real people." You can quickly see if the windows you laid out are aesthetically pleasing or if they look awkward.

The first part of the job is to get the skeleton code going and the main window with all of its menus visible. Then add the menu functions, and as a natural progression, the secondary and dialog windows. In each case, within the window procedures you simply call a placeholder function, to be sure you are getting all the messages you think you should be. Once these features are operative you can begin to experiment with the threading aspects of the application. You have many choices when you implement your threads, from the most basic (in which you manage all of the coordination and data passing) to the more "automated" (in which you let such Presentation Manager constructs as the window procedure and message-processing loop take care of a large part of the work).

Understanding the difference between posting and sending messages is the most important factor in putting the code together efficiently. The different results of these two actions are important, but their respective effects on the threads of the applications that transmit messages are more important. If a message is sent, then whatever code sends the message cannot process any more work until the thread returns. A posted message, on the other hand, can come back immediately, but the target of the message must be able to process it asynchronously from the poster.

By using object windows and `WinPostQueueMsg` you can make this a reality with very little work. You can thus concentrate more on determining which functions can run in parallel with each other and less on how to write the code. PM is not a fully object-oriented (OO) system, but it has its basis in OO technology; it will allow you to let its objects handle a large part of the work of an application. By using this mechanism and extending the PM message constructs and names to fit your task, you can make your code easy to read and write and still remain consistent with the rest of the system.

CHAPTER 15

Building the Core Function

Although the user interface is the most visible part of the application, it would be useless without something behind it to create the data that it shows. The core code is usually the easiest to port from other environments. This core code consists of the file-reading, number-crunching, database-searching, and memory management functions. The user interface is responsible for everything the user sees and does. The core code does the work and should not show anything to the user.

If you discover something that the user interface does not account for when it comes time to work on the core code, you will need to decide how to incorporate the addition. The solution may be as simple as providing messages to notify the user of a nonexistent file or an invalid calculation. In these cases, you can define an interface within the user interface so that the core code can signal the user interface to display a message. The core code should not have any display responsibility.

Data validation should also be coded into the user interface for two reasons. First, there is no reason to pass invalid data within the appli-

cation. The invalid data should be stopped as the user is attempting to enter it wherever possible. Users should get immediate feedback regarding invalid data wherever you can write it into the code. For instance, when a user tries to enter a letter into a date or currency field, a beep should sound as the letter is entered; the letter should not be accepted or displayed in the field, and possibly a popup should be displayed to tell the user what is going on. This is much more desirable from the user's perspective than receiving an error message after entering the whole field.

The other reason for data validation at the user interface level is so that the core code can simply assume the data passed to it conforms to what is expected. This makes the core code independent of the user interface and thus more portable. This independence goes back to a basic principle discussed at the beginning of this book: modularization with clean interfaces.

The OS/2 Presentation Manager delivers events to the user interface. The user interface should deliver data with integrity to the core code. Of course, there is no way for the user interface to know the data should be "David" if the user types "Stacy" but the user interface can at least do some type checking to ensure the data format is correct.

It is possible, however, to go overboard by putting too much type checking in the user interface. The type checking in the user interface code should be kept basic, such as determining whether the user types a character when a number is needed. Checking, say, whether the user asked to update a record that is nonexistent should be left up to the core code. Again, this is your call, but the optimal design is to keep the code as modular and independent as possible. In lieu of true object-oriented languages, this approach is as close to code reuse as you will get, so you may as well take advantage of it.

In this chapter you will see how to structure the core of the code, such as thread and memory management. The basic parallel tasks have already been outlined and have been implemented at a high level in the user interface. You know how you plan to manage the coordination of the work using PM message queues and the semaphore functions within `WinGetMsg` to relieve you of that added work. Now it is time to really get

down and dirty and see how finely you can break down these tasks into smaller, parallel units of work. At this stage the concepts of packing code, data pages, and page usage are put into practice. By the end of this chapter, you'll have completed the foundation of the application. Then comes the job of adding such advanced features as data interchange and international support.

MEMORY MANAGER PACKAGE

The memory manager package of your application does not have to be as complicated as if you were writing for DOS, because OS/2 manages memory itself throughout the system. You cannot inadvertently wander off into system memory. The system gives you a linear address, and once you have it, you can do with it what you wish. The only thing your memory management package needs to be concerned with is the allocation and subdivision of the memory in your application. Since you will be allocating memory pages and subdividing them, you should have some code that manages that work. As explained previously, if you allocate memory in only the sizes you need, there will be a great deal of waste because pages are allocated in 4K blocks.

The memory manager package of your application should be the only place within the application that the memory management APIs are called. You will write this code as its own subsystem with internal APIs. You may define such functions as `GetMem` and `FreeMem` to satisfy your storage requests. The functions within the memory manager package will simply decide if another page is needed or if the storage request can be satisfied by giving the requester some memory in an already allocated (or committed) page. Pages can also be freed when enough calls to `FreeMem` that map to the same page are made, and there is no memory in that page being used.

By structuring the code such that all memory management comes through `GetMem` and `FreeMem` you have the ability to control exactly how memory is allocated, suballocated, and freed in the entire application. This is critical in this virtual memory system in which memory leaks can

eventually bring the system to its knees. By centralizing the the memory management functions, you can avoid memory leaks, and should you see things appear that cause you concern, you know exactly where to look. In these functions will be all the assumptions on memory allocation for the application and if you ever need to change them for another version of the application, if the assumptions change or for any other reason, you have a clean, modular design. So, what should you do inside `GetMem` and `FreeMem`?

To begin with, decide if you want to allocate all of the memory the application will need up front, and subsequently suballocate it or if you want to allocate new blocks as needed. You may even want to combine both approaches. You may decide it is better to preallocate some of the larger and more often used data structures' memory and leave the smaller allocations you can't always assume to be done at runtime, and let your memory management functions handle those later. It's your application, with specific needs, and only you can make those decisions. By using this structure, you can tailor it to your needs and still leave room for change.

Memory Suballocation

You will keep a mapping of all of the physical memory that is allocated (and also which pages are committed) for your application. How this map is implemented is completely up to you. This map will be used by the memory management code inside your application to see if there is a free block within the currently committed memory (so you can suballocate that page) or if you will need to allocate and/or commit one or more new pages. (You may have some pages that are allocated but not yet committed.) If you need to commit new pages, you have several choices available to you.

You first need to decide if you want to allocate and commit just enough to satisfy the memory allocation request, or if you want to allocate extra for the next set of requests. This is where you can make more assumptions about the memory usage within the application that the operating system cannot.

In some cases, you may want to allocate more memory than you need for a single request, because you know that the specific request usually precedes more requests based on how the code functions. This preallocation can help you manage memory more efficiently if that fits the way you want to use memory. The advantage to doing all of this in your memory manager package is that you can tell it (in a parameter passed to `GetMem`, for example) where in the application the request is coming from, so it knows the characteristics of the memory needs at the time the request is made.

You may have decided to allocate the major data structures when the application first starts, and as such you know that all other memory allocation requests are usually small and dynamic. In this case, you would just allocate individual (or groups of few) pages as needed, possibly even allocating them and not committing them until needed, subdividing the pages as required. If you need a new page and have one that is not yet committed, a `DosSetMem` call inside `GetMem` takes care of that quickly. If you have no pages available, then of course, you will be calling `DosGetMem` to obtain new pages. You can see how the flexibility of the OS/2 memory management model and APIs help you do your work, while not forcing you to manage protection or other aspects of memory allocation you might otherwise have to. All you do is allocate pages where you need, and suballocate if you don't need new pages.

If you don't need to allocate more physical pages at the time of the request, you will suballocate existing committed pages.

There are varying degrees of detail with which you can manage the suballocated pages. The simplest way is to use the first-fit algorithm. Your memory manager receives a request for memory. With first-fit it will make a simple sequential search of its internal allocation tables to see if there is enough free space in any of the already-allocated pages. If so the request is filled with that memory.

The next scheme is best-fit. Using best-fit, the memory management package works in a similar fashion to first-fit, but rather than stopping at the first free block of memory within the already allocated pages, it will continue and keep track of *all* blocks of memory that will satisfy

the request. Once the entire mapping of pages is traversed the block of memory that best fits the requested block is used.

This scheme has advantages and disadvantages. An advantage is that you may waste less memory throughout the application. A disadvantage is that your performance may suffer a bit. First, it takes longer to traverse the whole list than it does to stop at the first free block. Of course, if a new page has to be allocated this difference is nonexistent. The other disadvantage is not even necessarily a disadvantage. That is that you will leave smaller and smaller blocks free within each page, so that as memory is allocated and freed within your pages you may wind up with blocks so small that they are not realistically usable. This may not, however, be a disadvantage because you may end up wasting less per page than with first-fit; furthermore, you will know approximately how big your memory allocations will be. An operating system does not know this information, but you will be writing your own special-purpose code and can make more assumptions than the operating system does.

There is yet another scheme called worst-fit. Some schoools of thought subscribe to the theory that by suballocating the largest block available within a page, you will end up wasting less memory in the application than the other methods. The properties of this method are the same as best-fit, since the entire application memory map must be scanned before a suballocation block is chosen. Whichever suballocation method you choose, you are making efficient use of system memory, and your performance will benefit.

There are some other assumptions you should think about in designing the list of parameters for `GetMem`. Since your application is given a set of linear addresses when a page is allocated and you give that set to a routine during the suballocation, you cannot move memory blocks around to compact pages. Doing so would place an unacceptable overhead burden on the application. However, since you know the characteristics of the memory requests of your application you can build in special checks to group the most used blocks in the same pages and the least used blocks in other pages. As outlined earlier, this will help in your paging performance. Of course, you could also choose to keep blocks of memory that live a long life in the same pages and other, shorter lived

or transient blocks together in other pages. Actually, the best scheme is a balance between the two. Since you are the application author, you must decide what best suits your specific needs.

Another thing to consider is the deallocation of pages (which you can assume, since you already know your specific allocation requirements). You may want to deallocate pages after all *client* routines of the page no longer need it, or you might want to keep them around since you already have them. Another possibility is to keep some number of free pages immediately available at all times and free any pages over that number. You could also have an idle-time thread that watches the memory map and frees extra pages when appropriate. In many cases this is overkill, but it is yet another option available to you since there will be some application programs that should make use of it. These are decisions to design in, making the assumptions about how your code will function to provide the best performance to your users.

You can overmanage memory if you aren't careful. It's easy to get caught up in trying to optimize down to the byte, but OS/2's memory management is very efficient by itself. All you want to do is minimize waste by strategically placing code and data where it is most efficient. You can actually harm performance by overmanaging memory. By combining the techniques of this chapter with those of sparse memory allocation introduced earlier you can create a very frugal application. In Chapter 18 you will see ways to use these techniques in specific situations to performance-tune the application once you have your application-defined memory management routines done.

Guard Pages and Exceptions

In Chapter 12 you saw how guard pages can be used to manage sparse allocations for large data structures. If you have large, multidimensional data structures where you are unsure of how many pages will be touched, you will allocate them as guard pages, with your own guard page exception handler to take care of things when uncommitted guard pages are touched. You can also use guard pages to manage the smaller memory requests in `GetMem` as described previously. When a new page

is needed, all you have to do is commit the next page. Again, another option for you. Let's look at how this exception handler should work.

When an application attempts to access an uncommitted page of memory, a page fault is usually generated. If, however, the page has the Guard attribute on it, a guard page exception is raised. If you are using this in your application, you do not want the system default guard page exception handler handling this, you want your own guard page exception handler doing the work for you.

Exception handlers are registered and operate on a per-thread basis. Since you have your memory management routines in a small package, and it is likely you will not want the different functions to run in parallel (after all, you don't want to free a piece of memory as you are allocating it, or have two threads manipulating the same piece of the memory map, and semaphore-protecting it is the same as synchronizing it) so you will take your memory manager package and put it on its own thread. That is the thread you will register the exception handler for, since it will be the first one that tries to touch a piece of uncommitted guard-page memory. Do you see how this is all starting to fall into place?

Back to the exception handler. All you need to do when you are called at this exception handler is call `DosSetMem` to commit the page. As shown in Chapter 12 you can also log these page hits to gauge the effectiveness of the memory management schemes you've chosen. Since you've put the routines in this package, you can always change it later. You've now created a flexible, efficient memory management package.

16- and 32-Bit Techniques and Coexistence

Depending on your application and business needs, you may need to restrict your application to 16-bit for a time. In this case you will not have the page allocation APIs and 32-bit functions available to you. You can choose to write your code for 32-bit, 16-bit, or a combination of the two. Even if you write 16-bit code to run under 32-bit OS/2, simply understanding that all memory is allocated in 4K pages is almost all that is necessary.

Note also that if you structure your memory management as just described, using your own `GetMem` and `FreeMem` APIs, you can replace the 16-bit memory management routines with 32-bit `DosGetMem/DosSetMem/DosFreeMem` calls and you have just gone to 32-bit memory management. So modular structuring helps here.

There are, however, some extra layers of translation that go along with 16-bit memory management that may affect your decision making. When a memory request is made from 16-bit code via `DosAllocSeg`, at least one 4K page is allocated. However, since a selector is allocated for the memory and a selector maps up to 64K, 64K of linear address space is allocated. This does not mean that 64K of real memory is reserved, just 64K of linear addresses. When the `MEMMAN` statement in `CONFIG.SYS` specifies the `COMMIT` flag, and memory is allocated via 16-bit `DosAllocSeg`, all of the memory for the allocation is immediately committed as well.

The most significant difference between optimizing 16-bit allocations versus 32-bit allocations is that since you have reserved 64K in your application address space for any allocation and all memory from `DosAllocSeg` is immediately committed, you may as well allocate several pages and just subdivide them. You can optimize the size based on your knowledge of the application's characteristics.

Another temptation when using 16-bit code is to allocate in one-page increments. This gives you a small advantage in that you allocate and commit only one page at a time, but each time you are reserving 64K of linear address space, of which you have only 512 megabytes for the whole application. You need to keep that number in mind because there is nothing worse than running out of linear address space in an application when you have plenty of real or virtual memory in the system.

Memory management routines that you write can help you manage the memory that OS/2 allocates to you more efficiently and can also add some portability to your code. Since you are calling internal APIs for each memory allocation, which resolve into your own small, compact memory management code, all it takes is a rewrite of the memory management package for another platform. Such an approach is better than having operating-system-specific memory management calls scattered throughout the code.

The next step is to examine how to structure and coordinate the threads in the application.

MULTITHREADING

Some techniques for multithreading were discussed in Chapter 14, but they only scratched the surface. Multithreading is more than just paralleling tasks; it can go to any level you desire. You could theoretically write your code to run on just one thread, but doing so would ignore one of the major advantages of writing programs for OS/2.

You have already taken the first steps. Such main tasks as initializing the application, reading the files, and performing lengthy calculations have been separated into parallel groupings. The tasks that can be run in parallel are now in groups and (if you have followed this book step by step) are set up to be called from object window procedures, or via `WinPostQueueMsg` using the pool of threads. The main task invocation and thread activation as a result of user input are already in place through the PM message-passing mechanism. The next step in multithreading is to break the tasks into parallel pieces. You need to balance the usefulness of moving part of a task to another thread against the overhead of managing the timing and coordination of the threads doing the work.

Just as you broke the applications into tasks that could be run in parallel, you must break up those tasks. The application is a task. It must be separated into smaller tasks that in turn will be broken further into subtasks. It's a basic divide-and-conquer strategy. You took the app and divided it into conquerable and manageable tasks. Now those tasks should be further divided.

The management of these threads can be handled in several ways. The simplest way is to create threads as they are needed and let them die when the subtask is complete. This is the easiest to code but the most expensive in terms of overhead; you may not have to worry about several jobs being requested of the same thread at once, but creating threads on-the-fly is an expensive operation.

Just as with the major application tasks, you can use threads as a multipurpose tool. Threads can be created at the outset of the application and set to sleep until needed. In this manner, the threads can be multipurpose; they won't use CPU when idle, and they won't have to be created for every task required. The overhead here is the coordination within the application.

I recommend the latter for thread creation. My preference is to have all of the worker threads managed by their own message queue, utilizing `WinPostMsg` to an object window or `WinPostQueueMsg`. You should set up the threads in your code to execute the sets of functions as you've structured in groupings of tasks that *cannot* be run in parallel. You should code your application as the pseudocode in Listing 15.1.

By kicking off the initialization of all of your threads before you create your standard window, you will be multitasking that operation, and by the time you start dispatching work to them, they will be ready. Upon application termination (by falling out of the `WinGetMsg` loop, which means you've received a `WM_QUIT`) you will post `WM_QUIT` messages to the message queues of all of the threads you have created. Recall you've called `WinCancelShutdown` in these threads, so to get them to go away, you will need to send them `WM_QUIT`s yourself.

Once you have done that, you will clean up the main thread's resources, message queue, call `WinTerminate` and terminate the main thread. Using these techniques will give you an easy way to multithread your application and keep it flexible and fast.

Synchronizing Threads

A primary concern in the use of threads is deciding when the shipment of a small task to another thread is worth the overhead. By overdoing the multithreading of the application you can end up wasting a great deal of time.

One of the biggest pitfalls of using threads within applications, though, is deadlock, a situation in which two threads each control a resource that the other thread is requesting. There is no easy way to recover from this situation. The problem of deadlock, however, can be

```
Initialize critical data structures

WinInitialize

DosCreateThread
DosCreateThread
    .
    .
    .
DosCreateThread

WinCreateMsgQueue

WinCreateStdWindow

while WinGetMsg
  WinDispatchMsg


WinPostMsg(WM_QUIT) to all threads

Clean Up
Terminate
```

Listing 15.1 Initialization and thread creation for WinPostQueueMsg design.

isolated and handled. There are many control structures that can be used to manage threads, coordinate execution, and avoid deadlocks. These structures were touched upon earlier and will be discussed in more detail here. They are the control structures such as semaphores, pipes, queues, shared memory, and critical sections.

Semaphores

A semaphore, as you will recall, is a structure that has an owner and two states: set and clear. When a thread issues a call to wait on a

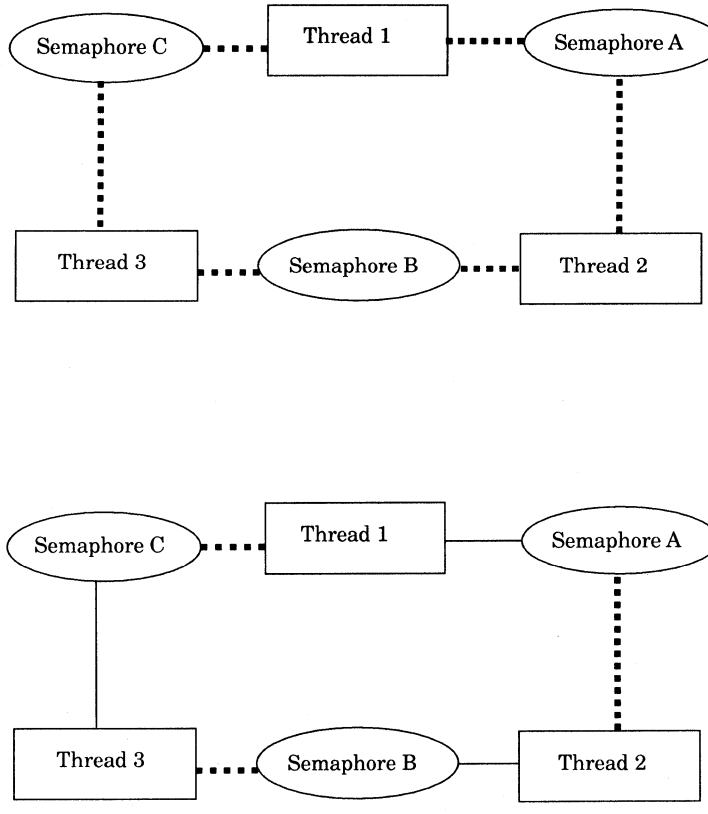
semaphore, the state of the semaphore is tested. If it is set the requesting thread will wait until the semaphore is cleared, at which time the scheduler/dispatcher will wake the thread and grant it ownership of the semaphore, and it will change back to the set state. If the semaphore is clear at the time of the request, its state is changed to set, and the thread continues. Any other thread requesting the semaphore before the "setter" thread clears it will wait until the semaphore is cleared.

As you can see, when you deal with several threads and several semaphores you can end up with deadlocks. There is no set formula to ensure that no deadlocks will occur, nor is there any easy way to recover from deadlocks. The best approach involves cautious programming and analysis. There is a way to recover from deadlocks, but it takes a little bit of trickery.

The analysis needed to prevent deadlocks involves a simple principle that is not difficult to apply: You must realize that virtually no operation is atomic. Your thread can be preempted at just about any moment—in the middle of an API, during a runtime function, or during a simple addition or multiplication. You will need to analyze not only the tasks and how you break them into concurrent subtasks but also how these subtasks interact and share resources. If two or more threads need to synchronize with a single semaphore you need to look at their other requirements and resources to see where these same threads cross paths again. By minimizing these multiple interactions you can begin to limit the chances of deadlock.

Another thing you can do is build a pictorial network of the thread interactions. Figure 15.1 illustrates this approach. You can see there are three threads. Threads 1 and 2 make use of semaphore A. Threads 2 and 3 make use of semaphore B, and threads 1 and 3 make use of semaphore C. No two threads make use of two of the same resources, but you can see that if thread 1 has semaphore A, thread 2 has semaphore B, and thread 3 has semaphore C, any semaphore request by any of the three will cause a deadlock.

Classical concurrent programming theory and principles, such as diagramming with Petri nets, and the theories behind such classical



Dotted lines indicate potential ownership

In the lower figure, solid lines indicate real ownership and deadlock if any thread requests any of semaphores A, B or C

Figure 15.1 Thread deadlock.

problems as the dining philosophers and bounded buffer can also be used to understand the implications of concurrent programming.

Once a deadlock occurs it is very difficult to detect. This poses another classical computational problem: the program that runs to completion. Who can know for certain if a program or task will ever complete? In reality, you never know whether a task will run to completion

until it actually does. If, however, you have a means of detecting that a deadlock has occurred, you have a chance to recover. First, though, consider the following: Even if you have a way to determine that a deadlock has occurred, it is not necessarily worthwhile to risk a deadlock to save a few milliseconds. In some cases it may be worth the risk to save 20 or 30 seconds on a single task. It is of course up to you to weigh the risks and benefits.

In general, you will be breaking up your application from the top down, so you are dividing tasks into subtasks and analyzing the attributes of which can and cannot be run in parallel, increasing the granularity at each step. Only you can know when you have broken it down to the appropriate level. Understand these subjects here, and "you'll just know" when it is broken down far enough.

If you do detect a deadlock in software, you can terminate a thread and resolve the deadlock by using an API introduced in OS/2 2.0 called `DosKillThread`, a feature unavailable in 16-bit OS/2. This solution has numerous implications, however. Once one of the threads in deadlock is gone the remaining threads are in an indeterminate state. The resources can no longer be relied upon to be correct. Depending on the task being performed, you can restart the task, but in some cases you are better off telling the user that something has gone wrong and that the application needs to be restarted. In some cases, `DosKillThread` will not be able to kill a thread, such as when the thread is blocked inside a device driver. In general, a thread must be in user mode (ring 3 in Intel terms) for `DosKillThread` to be effective. In any event, the more reliable method is to terminate the process, taking into account that one or more threads may not terminate gracefully.

In many cases, deadlock will occur because of unforeseen circumstances, but perhaps most often *race conditions* will be the cause of multithreading problems. Race conditions occur when two or more threads race for a resource; depending on the system the same result will not be obtained consistently. The reason for this inconsistency is that if a thread has used too much CPU time or if a time-critical or higher-priority thread is running, the threads racing for the resource will not "arrive" at the resource at the same time. As such, you may

be relying on synchronization and characteristics that may not always hold true. If you plan on finely slicing up a task, using threads to do the work, be sure to balance the advantages of parallel work with the effects of subdividing your tasks too much. There is also a mechanism by which you can ensure that threads do not race for a resource: the *critical section*.

Critical Sections

A critical section is a construct in OS/2 that ensures that only one thread within a process can run at any one time. It is a very expensive operation that should be used with great care. I have seen applications that use it as a simple synchronization tool, which is not a wise move. It is usually more expensive than not using a second thread at all.

A critical section is entered when a thread calls `DosEnterCritSec`. Inside this API, the scheduler is told to immediately block *all* other threads within the process. The process stays in this state until `DosExitCritSec` is called, at which time the threads are returned to their prior state. As you can see, the sequence of events isn't complicated.

The biggest issue with critical sections is their performance cost; furthermore, when you enter a critical section all of the application threads except for the one that has called `DosEnterCritSec` are frozen. If you feel a drastic tool such as a critical section is necessary, you should take a closer look at why the task is on another thread and/or why you need to synchronize the threads to such a large degree. At this point you may want to restructure the tasks. Critical sections are the "easy way out" of a thread coordination situation. I've never seen a good reason to use critical sections to resolve a resource conflict that could not be done with an intelligent use of semaphores. Since critical sections affect all threads in the process and not just the ones vying for a resource, there is a more widespread effect on the application and the possibility of causing more problems than you are solving. What I'm saying is try to use semaphores wherever possible for coordination, and use critical sections only if you must have a task on another thread and cannot find a way to do what you need with semaphores.

Critical sections and semaphores are the main control structures. You can also devise any home-grown method you wish using your own data structures in your own memory. After all, all threads have access to their process's structures, so whether you are synchronizing with a semaphore or a structure of your own there is no real difference. Semaphores are system-defined and are there for you, and unless there is something you need a semaphore to do that it cannot, there is no reason to reinvent what is provided for you.

Should You Even Use Another Thread?

The question of whether another thread should even be used for a task often arises. As with other aspects of the application, only you can make this determination. You have seen the pros and cons, advantages, and implications of many of the decisions you will have to make. For example, critical sections should usually be avoided. However, there are times within the application when you want to ensure that only one of its threads is running. The decision to even use multiple threads is also up to you. As you have already seen, it makes sense to have at least two threads within the application. One thread will service user input, and the other will do the work. Beyond that, however, it is a very subjective decision.

Common sense should be your guide. If you will get a performance gain by multithreading or further breaking down and subdividing a task, then you should consider it. However, if the control and coordination between that thread and any other is too fragile or risky, you must weigh what you are gaining against the potential problems. Gaining a few milliseconds by moving the work to another thread may not be worth the overhead if the possibility of synchronization problems or the cost of synchronization is too high.

In general, you should have threads set up at the beginning of the program waiting to do work. Threads should also be multipurpose, otherwise there is no real reason to have them waiting to do work—they could just be created on-the-fly. The mechanism discussed in the last chapter involving the use of an object window and window procedure,

or using `WinPostQueueMsg` can be employed throughout the system. There is no reason you should be limited to using object windows and window procedures to split out tasks from the main application window's procedure. You can use this construct to manage all the work within your application if you so desire.

There are many uses for threads. Just be sure you don't make things more complicated without making worthwhile gains.

IPC

Interprocess communication mechanisms are generally used only when you need to communicate across programs or processes. The data structures used for IPC can be used within a process as well. Be sure you don't set up any of these data structures for interprocess use if you intend to use them in only a single process. There is additional overhead in setting up the structures for interprocess use.

Queues

A queue is a standard data structure. An interprocess queue is somewhat more unique. You can set up queues for use between programs so that you can pass data between them. When you set up any queue, be it inter- or intraprocess, you need to synchronize access to the data. You need to ensure that there is data in the queue when you try to execute a read and that the queue is not full when trying to execute a write. You also want to be sure the same queue element is not being manipulated by two threads at once, hence the need for a semaphore.

Interprocess queues, as you will see with many of the interprocess data structures, use pseudodirectories in the file system to communicate. For example, interprocess queues are created in the `QUEUES` pseudodirectory. Although this directory does not appear on your hard disk, it is in the file system. A queue within a process can just be some memory.

With the interprocess queue, you will use the API `DosCreateQueue` with a name that will be created in `QUEUES`. This queue can then be accessed by other programs. It can be manipulated any way you like.

Remember, however, that these accessing processes are separate and distinct. They must both (or all) access the queue and manipulate it the same way. All queue accesses must be serialized, and the data added to and removed from it must be done the same way. This means either that there must be an established protocol documented by the application that others must follow or that the separate processes must be part of the same application. In any event, all accessing processes must access the structure consistently. This rule applies to any of the interprocess communications methods.

As you will see in the next chapter, the OS/2 Clipboard and DDE are the only IPC tools with predefined protocols that every application will know.

The data placement into and removal from the queue must not only be done consistently by all accessing processes, but these operations must also be coordinated through the use of a system semaphore. The system semaphore of choice for this type of coordination is a *mutex* semaphore. By requesting the mutex semaphore for the queue before executing a read or write on the queue, you can be sure you're not reading something while another process is writing potentially the same place.

Event semaphores are useful for triggering events, but they do not work well for coordinating access to structures. You should note that for pipes as well as queues, mutex semaphores are required to ensure data integrity. You will request the mutex semaphore before the access is attempted and free it once your access to the structure is complete. You must be sure the process does not own resources the other may request so as to avoid deadlock.

These considerations apply for pipes as well as queues, or any other shared structure you may build yourself (using shared memory). The DDE and clipboard functions you saw in Chapter 8 and will see more of in the next chapter have their synchronization built in.

Pipes

Pipes are another data structure that doesn't have much use within a single-process application but is very efficient for interprocess use. A

pipe has the function of a subset of a queue. That is, it functions exactly like a pipe in your house. Data goes in one end and comes out the other side in the same order. Just as with the queue, the reader and the writer of a pipe must understand the protocol, or the format, of the data in the structure. Since pipes are used mostly in interprocess communications, you will need to document the format of the data and the communications protocol if you intend to communicate with other applications and not just another process of your own.

Pipes are created in the `PIPES` pseudodirectory. Just as with the other interprocess data structures, access to the pipe must be synchronized and controlled.

Shared Memory

Shared memory is the easiest IPC structure to work with. It is just a chunk of memory that can be accessed by more than one process. Shared memory comes in several flavors. There is *shared memory* and *named shared memory*.

The main difference between named shared memory and what is called *anonymous* shared memory is that in named shared memory any application that knows the name of the memory can use it, whereas with anonymous shared memory, one application must explicitly give the memory while another application must call an API to get access to it. Using shared memory, you can implement any type of data structures you wish.

Once again, it is important to make prudent use of semaphores to coordinate access to any shared data structure, even free-form shared memory.

IPC is most useful for writing client-server type applications. Most of your everyday word processors or other standard applications don't have much use for IPC. However, if you are writing client-server applications for which coordination between programs is important, IPC may be just for you. Other than that, though, IPC is usually overkill.

This feature is a testament to OS/2's flexibility and scalability. You can write code that is very isolated and separate, or you can write code that communicates and works with others, depending on your needs.

FILE FUNCTIONS

File functions are often overlooked pieces of an application. OS/2 provides many features to applications based on the file system structures and mechanisms it supports. Since this is a book for designing applications I will not discuss writing installable file systems or device drivers. However, it is important to know the ramifications of using various features and structures of file systems.

Just as it is important to isolate your application from the underlying hardware, you want to weigh the benefits of using features specific to certain file systems. Some features are common to all file systems, but some are available only on certain ones. There are ways to support the features of one without precluding the use of others.

Internal File Formats

The first decision you have is to choose your application's internal data file format. There are really no operating system considerations that should affect your choice of data file structures. The main consideration is the same for any environment: data compression. Regardless of the operating environment, disk access is the most time-consuming operation. There are ways, however, to make this as fast as possible given the speed of the hardware. The main way is to compress the data such that it is optimized for your application.

A good example of an application that would make good use of a binary format is a graphics program. A word processor, on the other hand, can use a standard ASCII file. This ASCII file could have the word processor's control information (such as bold or italic) imbedded in the ASCII text, or you could even store that information in an extended attribute (EA) for the file. The EA method could make the file usable by other applications such as plain text editors or other programs that read ASCII files. This is a way to make your application's data files more open. Of course, some applications such as graphics do not lend themselves well to this "openness," but with a little imagination you can make your word processor, database, and spreadsheet files open. Then

again, by using EAs within the program to store parts of the data file, you introduce a level of incompatibility with other systems that do not understand EAs. Each choice has some kind of drawback; you must weigh your options.

Of course, don't compromise your own performance for this openness. It's not worth it if it makes your application need five seconds per kilobyte longer to read the file. Look at how often your users will use the function, and *then* determine the benefit or loss.

Another example of a file system function is the long filename support built into the High-Performance File System (HPFS). It is useful to use the `NEWFILES` keyword in your module definitions file to allow your application to use long filenames. However, it is not a very good idea to make the default data filenames long (that is, longer than the 8.3 FAT naming convention). Doing so would preclude the use of your applications on FAT systems. However, enabling your applications for long names gives you flexibility in that they will work just fine on FAT (8.3) systems while allowing you to elegantly handle situations in which the user has HPFS and chooses to give his or her data files long names.

The file system does not affect the choice of internal file formats in terms of how quickly the file is read from or written to. Some file systems are faster, and some are slower, but it is the user's choice as to which should be used. You should make your application flexible enough to handle anything the users may wish to do without locking them into anything or precluding the use of any of the file system (or any operating system) features.

Taking the LAN into Account

Many users will want to run their applications over the LAN. There are some considerations when writing LAN-capable applications. Aside from the file system considerations you need to realize that you will potentially have many users running the same code, which presents a good case for using instance data as opposed to global data structures. The application initialization file must also be considered here. If you have only one, then everyone will have to share it. You may want to

build some code in the application to allow users to choose the path of the application initialization file.

The main thing to understand with the LAN is that the connection can go down anytime. Therefore, you may wish to specify the attributes of your code and data a bit differently. Code is usually discarded and reread from the EXE or DLL when needed. This is not good if the LAN connection goes down. In such a case you may wish to specify that the code is to be preloaded and should be swapped as opposed to discarded. This will increase your swapper file size but will give you a bit more security. The performance difference will be negligible, because whether you reread code from the swap file or the original executable, you are still reading from the disk. In fact, it may even be slightly faster swapping the code since the swap file is local, whereas rereading from the executable would require a somewhat slower LAN read.

The next consideration in whether to LAN-enable your application is the way you structure the data. In a LAN environment many users will use the code at one time. Any shared files or even IPC structures need to be examined to ensure that there is no confusion. For example, you should lock data files when reading them, and named IPC structures such as queues and pipes should have names that uniquely identify the workstation. Since these structures can be used across processes and across the LAN the names cannot be generic. For example, if your application is called MyApp, you don't want to create a queue called `QUEUES\MyAppQ1`. You should use the workstation name in any of your named IPC mechanisms to ensure data integrity.

The final consideration in using the LAN for the application holds for any other file system function—namely, don't rely on the application always being run on a LAN or, more specifically, on any particular type of LAN. By staying generic you will be more portable not only in terms of your coding, but also in the flexibility of your user's environment.

Using File System Structures

Let's extend this discussion of file system-dependent features to the structures available in file systems. By sticking with the OS/2 API you

can rest assured that your application will run on any OS/2 file system. However, you must decide whether you wish to use such structures as EAs or to rely on specific control blocks. In general, you want to stay away from the control block-level functions. Although you can accomplish a great deal with this information and `DosDevIOCTL`, you start to make yourself vulnerable to users' whims when it comes to installable file systems and new technology.

When evaluating EAs or other file system structures you need to examine and stick to the lowest common denominator in terms of function. If not, you will need to require users to configure their systems the way you prefer, which can limit your application's acceptance. In addition to using only the lowest common denominator functions, you should conduct your performance testing with the slowest of the bunch. This gives you two pieces of vital information. First, it gives you a good indication of whether you want to use this function. If it was a toss-up to begin with, this may influence your decision one way or the other. Second, you have a benchmark to see the slowest speed at which your system will run.

Using these pieces of information you can make informed decisions as to which functions to use or leave out, and whether you need or want to require a specific file system configuration. You may have optimized as much as possible and still find that the slowest system is too slow. In this case you will either have to find something else to do, find something to leave out, or specify a specific configuration.

SUMMARY

Now the main application design is complete, so you can start to look at the advanced features you want to add. Since you have gone through the pros and cons of the user interface, core function, thread, IPC, and file system features, you have a much better idea of which advanced functions (such as DDE or clipboard) you want to use.

As you have seen so far, the application has been divided many times—first by splitting the job of the application into the user interface

and base functions. Now the user interface and base functions are well defined and designed to get the job done. The next phase of the design is to put in the "bells and whistles." Note that these must be incorporated into the design as well. Don't get lazy here. Every piece of code you put in the application should be designed with an understanding of everything you may encounter. You have the bulk of the work done. Now is the time to look at the advanced features, some further tuning tips, and the finishing touches to make your application a well-designed package.

CHAPTER 16

Using Advanced Functions

The advanced functions in your application are something that you looked at, evaluated, and decided upon very early on in your design. You figured out that you wanted your program to have such features as data communications, graphics, and help. You also looked at how these functions related to your specific application. For example, when you looked at data communications you looked at the main function of the application. Did you use multiple processes? Did you need IPC private and specific to the application? Did you want to publish an interface to your application or just have it use a generic, preestablished interface? How do you want to print? Is the application graphics- or text-based?

All of these are decisions you made when you mapped out the major functions of the application. Now is the time to choose the correct mechanism and tools to implement these functions. Throughout this chapter the advanced features you can use for your programs as well as which ones are best suited to your tasks will be explored. Until now our

discussion has been limited to a description of the features. Now is the time to incorporate them into the design.

Recall how the user interface was put together and how you will add in code step by step and section (or function) by section, simply filling in the functions that are called from your main window procedure's CASE statements. These functions are no different. You will probably just want to save them for the end of the design and coding phases of your project, if only to be sure you have your most important functions—the data manipulations—working properly. Once these functions are working and your data is displaying the way you want it to in all the various flavors you plan to offer to the users, you should begin to look into the advanced features. The other reason to wait until now is that during the design and prototyping of the other parts of the application you may want to change some of the user interface or data display routines. If you have already implemented the advanced features such as DDE you are faced with the choice of either throwing that work away or not changing.

By holding off on the advanced features until you are sure of the other parts of the application, you have the most flexibility if you want to change. Of course, you should always keep all of these items in mind when designing any part of the application or changing anything, but hold off on committing yourself until you need to.

CLIPBOARD

The most commonly used functions for data exchange are DDE and the clipboard. What they are and how they are used has already been discussed, so let's just jump into putting them into use. The clipboard is the easiest function to design and code and is usually the easiest for the user, for the simple reason of its consistency. The clipboard protocol is predefined and common to all applications, and there is not really too much that applications can do differently from one another.

The main function of the clipboard is to allow a user to mark an area of data and choose to cut or copy it to the clipboard. Furthermore, an application can allow the user to take the data in the clipboard and

paste it in wherever the user chooses. This is usually initiated with menu functions. The use of the clipboard depends on the function of the application. If you are writing a text-windowed application the clipboard is only of limited use, and you cannot easily manipulate clipbard data anyway. The VIO shield layer introduced in Chapter 6 showed the clipboard interaction for programs running in a text window.

There are three data formats supported by the OS/2 clipboard in PM graphics, or AVIO windows: text, metafile, or application-defined. In general, the text and metafile formats are the most commonly used, because they are predefined and available to any application. The user-defined data format is versatile, but the originator and the recipient of the data must both know about the data format, otherwise it cannot be used. This is a small limitation, but the function is useful for cooperating programs.

Text Data

Text data is quite simple. When writing the data to the clipboard the format specified by the program is `CF_TEXT`. The text data is placed into a spot in memory, and the memory is given to the clipboard. The data in there is simply plain text. When an application comes along to get the data from the clipboard (such as when a user selects `PASTE` from a menu) it queries the clipboard as to what type of data if any is in there, and then requests that data.

Sometimes there may be data in the clipboard that is unidentifiable by the application. It may be that this application is a text processor but that the data in the clipboard is `CF_METAFILE`. The application looking to get the data from the clipboard has several options. It can monitor the clipboard and keep the `PASTE` item grayed until a recognizable piece of data is placed in, or it can tell the user what is going on when the attempt is made to actually paste.

Metafile Data

Metafile data is just another type of data that can be placed in the clipboard. It is a standard OS/2 metafile, which is a defined graphical data

exchange format. To place this data into the clipboard you just need to specify CF_METAFILE in your API call along with the address of the data that has the metafile in it.

Application-Defined Data

Application-defined data is simply data that the application has defined. Most other applications will not know about the data format. It is sometimes useful to hide data from other applications, in which case you may wish to use DDE. The only real drawback to DDE is that the recipient needs to know the data format. Of course, the same holds for CF_USER data in the clipboard. You may want to go to DDE just for application-specific data.

DYNAMIC DATA EXCHANGE (DDE)

Dynamic Data Exchange (DDE) is both a set of PM messages and a protocol for exchanging data. As introduced in Chapter 8, there are two types of DDE communications: the one-time exchange and the hot link. DDE is a powerful tool that most applications do not take full advantage of.

Although the clipboard is used to paste a block of information from one window to another, in many instances the data is not directly interpretable by the target application without some work. For example, if you import a graphic into a drawing program it usually comes in a metafile format. In order for the target program to manipulate it (other than positioning it), the target program must translate the graphic and change it to its own internal data format. Of course, some data such as text is easily used by the target application, but in general DDE is used to move data between applications as opposed to the clipboard, which usually moves a representation of data.

DDE is somewhat of a PM pipe. An exchange of data takes place through a coordinated set of messages in a conversation initiated by a WM_DDE_INITIATE message broadcast by the source application. Usually,

DDE conversations are activated by menu choices selected by the user. The user also selects which application is to be the server and which the client (actually, it can be a peer-to-peer conversation as well).

When the user initiates the DDE conversation and your program broadcasts a `WM_DDE_INITIATE`, another application can respond with a `WM_DDE_INITIATEACK`. Messages are then passed back and forth to verify that they want to "talk" about the same topic and, if so, to set up and maintain the data transfer between programs (`WM_DDE_ADVISE` and `WM_DDE_UNADVISE`, which establish and terminate the hot link). The messages defined by DDE provide the messaging protocol, and the applications simply provide pointers to the data.

A restriction of DDE as opposed to pipes is that DDE cannot be run across a network. Named pipes, however, are more appropriate if the communication is to take place across machines.

The DDE messages are defined by OS/2 and despite minor differences can be used between native Presentation Manager applications and applications running in WINOS2 (the Microsoft Windows code built into OS/2) sessions. The messages define the communications protocol and contain pointers to data that is application-specific.

When you are deciding on which IPC tool to use, keep in mind that if simple data is to be transferred and manipulated by another application, DDE is a very good choice. PM manages the messages being passed as well as defines the protocol. Unless you are moving data between processes on different machines DDE is usually the best choice.

A one-time exchange of data is more often accomplished through the clipboard than through DDE because of the setup overhead required by DDE, unless of course the data is to be manipulated by the target. The hot link is the feature of DDE that is used more often. The reason that the hot link is so much more powerful than, say, pipes or queues is that it is a two-way conversation between applications. Sure, you could use multiple pipes or semaphores to accomplish some of the same functions as DDE, but they are much more cumbersome, and the overhead and synchronization can be a nightmare. DDE provides efficient two-way communication. Additionally, the data can be of any type you choose (as is also the case with queues and pipes).

A large part of your decision of which IPC tool to use lies in personal preference. However, for intermachine communications you should stick with name pipes. For one-time data placement, such as importing a graphic into a document or moving text between programs, the clipboard is the best choice. If, however, you want to hold a conversation between applications on the same machine where data and other information is passed back and forth, DDE is the way to go.

PRINTING

Printing is a hot topic in OS/2. The OS/2 print subsystem is very complex, flexible, and powerful, yet it is also very straightforward. It can seem very confusing to the uninitiated, but guess what? You're going to be initiated right now.

Printing will be discussed here as it relates to PM, since PM is the primary interface for OS/2 applications and also provides the greatest level of power and flexibility. PM printing is device-independent. That is, the application does not need to care if the printer is a plotter, laser printer, or a 9-pin dot-matrix printer from 1982. There are only a few pieces of information the application needs. OS/2 does the rest.

Let's first go through the print subsystem. Figure 16.1 shows the PM side of printing. It works with printer drivers, which are special presentation drivers discussed back in Chapters 5 and 6. Recall also how all applications output to a presentation space, which is associated with a device context. The device context in this case is the printer driver.

In general, a print job will go from the application to the spool queue. The files that are placed in the spool queue are usually metafiles. Once in the queue the jobs are removed in order and sent through the printer driver a second time to be translated from the metafile to the printer-specific data.

As you can see from Figure 16.1, the printer driver is used the first time by the engine and the application to create the metafile. The job description file is also created and is placed in the spool queue along with the job file. At the proper time, the queue processor will come along

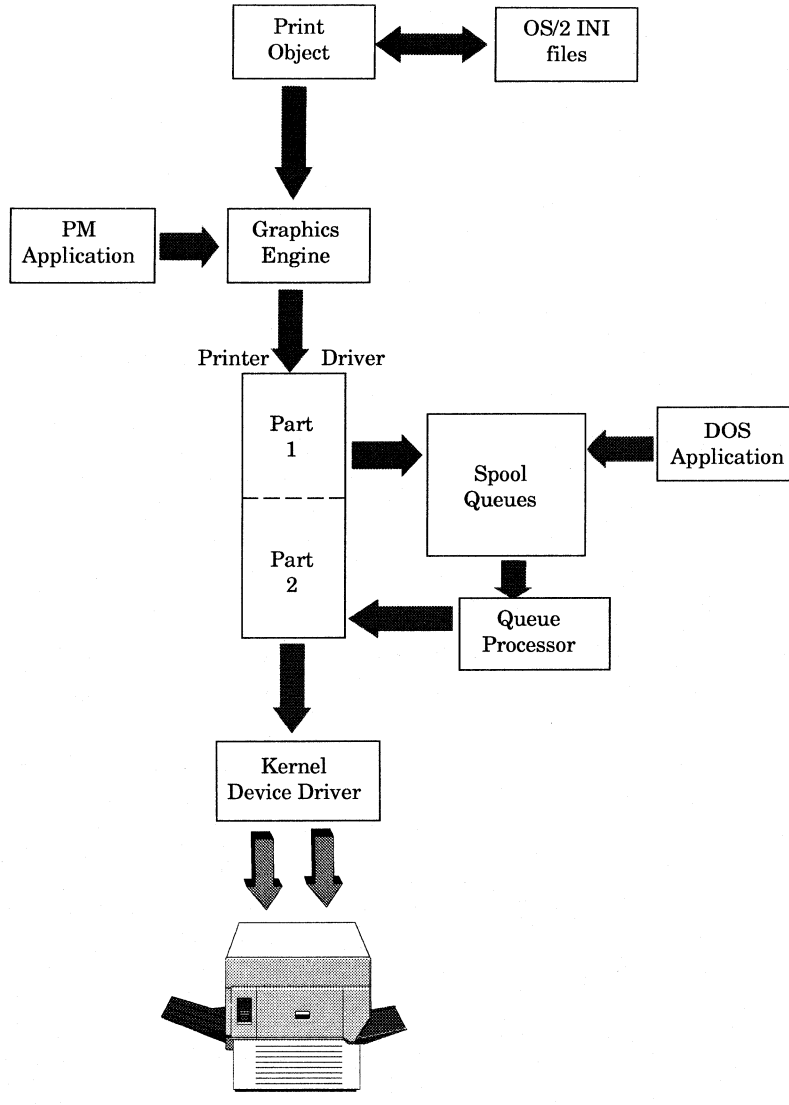


Figure 16.1 OS/2 print architecture.

and pull the job from the queue. The queue processor will create a PS and DC, and then invoke the second pass of the printer driver to do the actual translation to the specific device. There are some printer drivers, such as the PostScript driver, that do not use metafiles; rather, they just place the raw data in the queue.

PostScript is unique because PostScript is a programming language, so it is more efficient to just generate the PostScript program in one pass rather than generate the metafile first. The reason for the metafile for the queued print files is that it is a compact data form, so the jobs in the spool queue will be smaller. It is also a generic data format and can be easily manipulated and viewed.

There are paths that the data can take based on choices both by the user and by the programmer. When an application wants to print data it can do so in either standard or raw mode. Raw mode is rarely used because it bypasses the device-independence of the system. In raw mode (`PM_Q_RAW`) the application is responsible for creating the printer-specific data stream. When an application opens a printer DC with `PM_Q_RAW` the printer driver is simply told not to create a metafile from the print job. The printer-specific data is expected from the application, so whatever the application outputs is placed into the spool file.

When an application prints using `PM_Q_STD` (standard printing) the first pass through the printer driver is used to create the metafile. This is indicated to the printer driver by the `PM_Q_STD` in the `DevOpenDC` call. The spool queue processor then comes along to pull the job out of the queue and send it to the printer, just as in raw printing. However, in this case the queue processor sees the `PM_Q_STD` in the job description file and treats the job file differently. In `PM_Q_STD` printing the queue processor has to issue a `GpiPlayMetafile` call to play the metafile on the PS and DC. When the job is raw the queue processor knows that the job is not a metafile; it simply tells the printer driver that the data is already translated and should just pass through.

Print Destinations

Now that you see how the print subsystem works at an architectural level, let's discuss the issue of designing printing in your applications.

The first and most obvious item of concern to an application is where to print. OS/2 offers the concept of the print destination. A print destination is neither a printer, nor a queue, nor a port. As far as the user is concerned, a print object is what is asked for. Programmers know that behind this print object is a queue with a printer driver that is associated with a port (refer to Figure 16.1).

The user has two options in printing a document. The first, which has already been discussed, is when the user drags a document to a print object and drops it. The document object is called at the `wpPrintObject` entry point or method. As a result, the application should be invoked in "subset" mode to print the document or the function in the application DLL should be called to perform the printing.

The other way to print is to ask that the data be printed when the user goes through the application menu. Unlike the scenario in which the user drops an object on a printer, the destination is not known. When the user drops an object on a printer the destination is already known by the application because it has been passed by the object. When the user asks the application to print a document via a menu selection the application must query the user for the print destination. The first step for the application is to query all of the available print destinations by calling `SplEnumPrinter`. It should then present a list of destinations and ask the user which one is desired. You may have an application default print destination stored in the application defaults file, which should appear highlighted in the list. The user then has the choice of either changing the selection or accepting the default.

Once the destination is determined, the two scenarios are almost identical. The next item of concern is to decide on the print job properties. Each printer and queue has default properties. The selected destination should be queried for its job properties, which should be presented to the user along with an opportunity to change those properties. This information must be determined before the printer DC is opened.

Once this information is available and the printer DC is opened, the print job can be sent. This sequence of steps is important to keep the application flexible and able to use all of the operating system's features. For example, there can be multiple queues for printers, each with different properties. By querying the user for the printer name and

verifying the properties for the print job the application provides the user with maximum flexibility and allows the user to take advantage of all that OS/2 offers.

The main issue in printing is to ask the user where to print. Once that is determined, look for the job properties in the queue or destination selected (they are generally the same, since each queue associated with a printer driver is another print object or printer name). Next, display those properties to the user, providing the opportunity to change them. Once the properties are to the user's liking, use them to create the printer DC.

Using this approach the application maintains flexibility and consistency with the system configuration. Please see the references in the back of this book for information on actually coding printing in applications.

FONTS

Font use in programs goes hand in hand with printing and works in a similar fashion. As a matter of fact, this concept of querying the system and presenting the user a list of choices as a result applies to the entire system, because each system is unique. Fonts installed in one may not be in another, and printers with varying capabilities are on some systems and not on others. By querying the system, you don't bind your application to a specific font or printer.

Querying the fonts can be a tough job. Fonts have names built into the font file, and different fonts from different vendors can cause naming conflicts. As a result, there was no easy way to programmatically present a list of all available fonts to the user. OS/2 introduces a standard font dialog. The font dialog is one of a set of standard controls such as the file open dialog or even the push button or MLE that are available to programmers. By simply filling in a structure and calling the `WinFontDlg` API, a dialog box can be displayed to show the user all of the available fonts. Because fonts are device-dependent one of the parameters to `WinFontDlg` is a handle to a PS. This PS must be associated

with a DC. This way `WinFontDlg` knows which device's fonts to query. In general, the fonts are either screen or printer fonts.

Depending on how the structure is filled in by the application (which is usually a direct result of user preferences either in the application defaults or from some dialog) the list of fonts, font styles (strikeout, bold, and so on), and point sizes are displayed for the user to select. Along with the font names and styles there is a preview area of the dialog to show the user what the font looks like. This sample string is also determined by the application and passed in the `FONTDLG` structure passed to `WinFontDlg`.

Once the user selects a font through the dialog, the `FONTDLG` structure is filled in with the user's choice and returned to the application. The application can then use the font; it will know the font is available since an unavailable font couldn't be a choice in the `WinFontDlg` dialog window. That is the main reason for querying the system as to what is available before trying to use anything. If an application attempts to use a font or font metrics without knowing whether the font is installed in this particular system, the engine will just give the closest match. Many times this will not be what the user wants. By querying the system first, however, the user can see what is available. If the desired font appearance is not there the user can install new fonts or choose the closest to what is desired.

`WinFontDlg` is a simple way to display all of the fonts in the system that the user is operating, and it returns the user's choice to the application. Once the font is selected you can proceed to use the font in the application. The key is that the application has the flexibility to use this function.

HELP FACILITIES

Another major feature of any application is a help facility. OS/2 has a help facility built in, with a set of APIs to access it. OS/2's help facility is called the Information Presentation Facility (IPF), which is a set of APIs

and code in the OS/2 subsystems that presents a CUA-conforming set of help windows as simple or complex as the application designer wishes.

The help text for the IPF is something that anyone familiar with IBM systems should know. The "language" or "mark-up language" for the IPF is a set of tags compatible with IBM's Bookmaster. It is a syntax and a set of tags that can be added to standard text that have significance to a processing program. On the mainframes the program is the Bookmaster processor. On OS/2, the processor is the Information Presentation Facility Compiler (IPFC). The text files that compose the help text are input to the IPFC, and what comes out are help files. The tags specify heading levels, bold or italic fonts, and more advanced functions such as hypertext and hypergraphics, which are linking facilities built into the tags, the IPFC, and the IPF code. By using special tags both text and graphics can be linked to other help windows or topics. All the user need do is double-click on the highlighted text or the graphic to cause a jump to another window. This facility is helpful in providing further detail on a particular topic.

One important item to note is a tool from IBM called HyperWise, which builds IPF tags into documents and in a WYSIWYG fashion, builds IPF files for use with the compiler to create help files and help panels. The help file is structured into panels that are really windows. Each window is a help panel with a unique panel number. The help code in the application works hand in hand with these help files produced by the IPFC. The APIs in the code link these help files and panels with the windows.

Based on how deeply you wish to imbed help code, you can use a simple help menu, context-sensitive help, or a combination. OS/2 has a default behavior (recommended by CUA, of course) that whenever the user presses F1 a help message is sent to the application's window. As a result the application has the choice of displaying a main help menu or taking advantage of the context-sensitivity. The context-sensitivity is obviously more complicated, but the advantages to the user are immeasurable. Whenever the user has a question about the current operation the F1 key will offer help. Quite simply, the system will identify the context of the help being requested. If you wish your application to take

advantage of the information, no problem. Ignoring the details does not preclude your use of a help menu for general help or overall application help, but an intelligent use of the two will make your application even easier for users.

The cross-references should be conducted through the use of hypertext and common panels. For example, when the user is about to print data and presses F1, a help panel on printing and print options for the application should be shown. However, when the user selects the "Help Index" from the menu and wants to see how to print, the same help panel should be shown. All you are doing is creating a flexible interface to the same help information. It's a little redundant, but it's absolutely necessary.

Your main goal for the user interface is ease of use. However, help is inevitably needed, whether for simple or complex operations. By using the IPFC and the IPF API (such as `WinCreateHelpInstance`), you can create the complete solution.

There is one other little-known fact with the OS/2 help facility: There is a switch for the IPFC that causes it to generate an `.INF` file. An `.INF` file is an online book that is readable by the `VIEW.EXE` program supplied with OS/2. This is how the OS/2 online documentation is prepared, and there is no reason that you cannot generate your application's documentation the same way. The input to the IPFC for such a book is the same as a help file, without the panel numbers or names. The only difference in compiling a help panel or an online book is the `/INF` switch when the file is compiled with the IPFC. By using the help facility and IPFC, you can create a consistent set of books, both online with the IPFC and printed with Bookmaster, or by printing the online book along with the help panels for the application.

HELP AND MULTIMEDIA

One often overlooked aspect of OS/2 is the application implications of the built-in multimedia functions. Of course, you can write games,

video players, and other traditional multimedia-based applications, but this feature of the operating system opens up new horizons for the IPF help subsystem and multimedia.

Picture an encyclopaedia written in IPF for hypertext linking, along with hypergraphics links linking pictures, image captures, and video clips—all written knowing you need to ship only a set of `INF` files, because all of the multmedia functions are built in.

The OS/2 IPF facility has the ability to hyperlink graphics with a simple `:link.` tag. This allows you to build graphics into your help files (either help panels in applications or in `INF` files) by embedding graphics. IPF also has the ability to launch programs. Why not design your help windows and `INF` files to launch the `MMPM/2` movie player to show how things are done?

The functions built into the operating system such as `MMPM`, can add a new dimension to what you can do with your applications with no new source code. You may even want to just write new publications in IPF, embedding graphics, audio and video clips for documents, newspapers, magazines, help files for your applications or even training applications for people. This should generally remain separate from the application program, or could even be the product itself. These features further assist you in defining modular designs that you can expand, enhance, and replace when needed.

SUMMARY

The IPF is just one of a large number of extendable subsystems provided by OS/2 as well as by others. IPF was implemented by IBM, but there are others being developed all the time, and you may be the designer of one of your own. As you can see, the advanced features of OS/2 are no real mystery; many of them can be implemented outside of the mainline development. For example, adding font support is really no big deal. All you need to do is to add in some font calls to obtain a font from the user and use it. You should probably look at your internal file format

to preserve this font information, but other than that there is not much that needs to be done.

The help code can also be developed offline from your main functions and added in later. Recall our earlier discussions on language dependencies. These will be expanded in the next chapter, since they are important in developing any code that displays messages.

This chapter cannot begin to cover all of the features in OS/2 that can make your applications the most powerful in the market, especially because new features are being implemented by IBM and others all the time. This chapter was designed to give you information on some of the most common features at the time of this writing as well as an insight into how to incorporate them into your code.

Just remember, this advanced function code can be written outside of the core development, given the modularity you are designing, but you should have a thorough idea of what you want and how it should look before considering your design complete.

CHAPTER 17

Non-English-Language Support

Non-English-language support in applications is something that is usually overlooked but can add huge numbers to your application's sales. Quite often this is realized after the English version of the application is complete. By structuring your code and your development with non-English support in mind the option is open whether or not you plan on it from the outset.

FLEXIBILITY IN YOUR CODE

The key to enabling your code for non-English support (from now on national language support, or NLS for short will be used) is to keep it flexible. This means not coding specific strings or string lengths or even putting them in the same places in your source code tree. By taking out all of the strings, or words that can be translated, you make the job of

building NLS versions easier. You won't have to change a line of code in order to build the NLS versions.

The main idea behind this separation is to keep the code files separate from the readable text. When it comes time in the build process to link in the messages and other readable resources, the variables in the makefiles will point to the appropriate language files. This is best described by examples, which will be given in the remainder of this chapter.

There will always be places where you will have readable text that will not change, such as filenames read from the disk or names of programs or types of objects in your applications that have a universal meaning. However, for the translatable text these techniques will help you save development costs while enabling you to develop NLS versions of your code by simply translating the text.

Message Files

The first order of business is your messages. These are text messages you might see if the application cannot start or has a problem reading a file. Of course you can always use PM message boxes for such messages, but the point of this section is that message files are available. In general, message files are used for text-based applications. Otherwise, since the application is PM the message should be a PM-based message such as a message box. The message file is exactly that—a file with only messages.

There is a set of APIs and utilities to build message files. The source for a message file is a plain text file. The text file contains message numbers along with the text of each message. By using the tool `MKMSGF` supplied with the IBM OS/2 Programmer's Toolkit you can turn this message text file into a binary `.MSG` file. The `.MSG` file is analogous to an `.OBJ` file. Rather than using `LINK` to bind the `.OBJ` to the `.EXE` there is another tool, `MSGBIND` that, binds the `.MSG` file to the `.EXE`. As you can begin to see, the message numbers are what the application is interested in, and they are universal.

By using only message numbers in the executable code any message file can be linked in with the `.EXE` as long as the message numbers that

the code expects exist in the `.MSG` file. The message numbers and the message file are accessed by the message APIs such as `DosGetMessage`. These APIs will allow the application to retrieve a message from the file based on the message number.

In summary, the message file is all that needs to change. You keep the same message numbers, but by rebuilding the `.EXE` with a different message file for each language you can make your code independent of the language.

Windows and Dialogs

Dialogs represent a challenge in coding for NLS because a dialog is a window that interacts with the user and contains such controls as radio and push buttons that are defined with text. For example, a push button has text defined to be displayed inside the button. This makes translation a bit difficult because in order to define the button in the code you also have to define the text. There are ways to get around this. There are some programming techniques along with constructs provided by OS/2 to separate these types of text as well as with message files. Some of the tools available are stringtables and string constants. Another way to separate the translatable text is with the resource definition (RC) file. The RC file is usually treated as a translatable entity anyway. Some of your code may require that you generate the window and its controls as the user runs the program. Others are more static and can be defined completely in the RC file.

Stringtables

A tool that can be used whether the window is built at runtime or is predefined is the stringtable. A stringtable is a type of resource that can be defined in the RC file. Since the RC file is a translatable file that requires a different version for each language, putting strings in it is no big deal. The stringtable is PM's equivalent to the message file. Using `WinLoadString`, you can pull strings from the table and treat them as though they are string literals in the code.

The RC file is built into the .RES file by the resource compiler and subsequently linked into the .EXE in much the same way as a message file. Just like the .MSG file the RC file can be different without changing the executable code and simply relinked to make the complete PM executable file for each NLS version.

Building Windows on-the-Fly

So far you have seen some methods for static windows. These windows are defined and have all of the same elements all the time. There are circumstances in which you may need to create windows or dialogs based on either the current state of the application or user actions. These windows are a bit more involved to code, but they are not very difficult.

On the surface, you might look to subclassing the controls in order to customize their text and initial values such as in a listbox. There is an API that you might not think to use in coding these windows. Most controls have text either associated with or displayed inside them, such as the push-button example cited earlier. By using WinSetWindowText, you can set the text of any of your controls on-the-fly. Another aspect of this approach, which will be elaborated in Chapter 19, is to create the control as invisible, set the text with WinSetWindowText, and then make it visible.

WinSetWindowText takes a string and a window handle and sends a message to the window. This string can come from anywhere. It can be a string literal or a string loaded from one of the stringtables or other resources. So even though you may need to build the window at run time as opposed to having it defined in the RC file, the strings can be separate from the code in the RC file. You will load the string from the resources and use WinSetWindowText to put it in the control. Then, when the control becomes visible it shows up with the text just as if the control were defined in the RC file completely.

As you can see, you can create any combination of windows and text, whether you define them at compile time or build them dynamically. Regardless of how you do it you need to keep the executable code

separate from the text. The last piece to this puzzle is to size the text, buffers, and windows flexibly as well.

Using Lengths and Proportions

In addition to separating the text from the executable code you need to keep measurements of window sizes, text buffers, and other pieces of data independent of the string or message size. In fact, this is a good practice to follow in all of your code. Since you will be dealing with text in differing lengths in the same windows to convey the same information, you cannot hard-code the sizes of the windows or their controls. The way to size your windows is to use the OS/2 API functions as well as others such as the C function `sizeof()` to determine the screen space the text will take up. Then use that value to calculate the size of the windows.

The simplest way to calculate the size of a string is to use the `sizeof()` function to determine the number of characters in a string. Of course, this assumes the size of the buffer intended to hold the string is large enough. Only you can determine the proper size of the buffers based on the message. You can simply look at all the versions of the message and make a buffer that is big enough to hold the largest one.

However, knowing the length of the string is not enough. The size of the font is also important. You need to take into account the fact that users can choose one of many fonts and sizes for the windows (such as title bar or button text). There are generally two ways to accomplish this. The first is to query the font for the window (or trap the paint message for the window by subclassing it), determine the average width of a character, and multiply by the number of characters in the string. Although this is the simplest method, there are some drawbacks. The average character width is just that, an average. With the proportional font spacing in PM you can easily see that the character "box" around the letter *i* is much narrower than, say, the letter *w*. If the string you are displaying is a string of ten *w*'s you will need a wider window than if it was just the word "Hello." The average character width is not always a good indicator of how wide a window you will need.

The second way requires a bit more code, but it is much more precise. You can specify a parameter in a call to `WinDrawText` to tell it not to draw the text and instead return the size of the rectangle needed to draw the text. This is done by specifying `DT_QUERYEXTENT` in the `WinDrawText` call.

One of the parameters of `WinDrawText` is a pointer to a rectangle structure. When you want to draw the text you pass a pointer to the rectangle into which you want to draw. If the flags passed into the call contain `DT_QUERYEXTENT`, `WinDrawText` will ignore the rectangle structure you passed and calculate, from the font size for that PS and string passed, the rectangle needed to draw the text. The rectangle structure you passed to `WinDrawText` will be filled in with this calculated rectangle, so you can use it on a subsequent `WinDrawText` (or `WinSetWindowText`) call to actually draw the text.

You can also use this `DT_QUERYEXTENT` flag to just get a bounding rectangle for any text drawing. For example, if you have a window upon which you want to call `WinSetWindowText` you can query the size of the window and then use a `DT_QUERYEXTENT WinDrawText` call to figure out how large the window whose text is to be set needs to be.

Another technique is to use `WinDrawText` with the `DT_QUERYEXTENT` call to determine the size of the rectangle that will hold such text as a window's title bar or system menu. The reason you may want to do this is to ensure that the window you create is of the right size to hold the title bar text, action bar, and so on.

Since the strings are not always known beforehand (due to NLS considerations) you cannot hard-code window sizes and positions. By querying the string you are loading (in addition to the font it will be drawn in) to determine how much space is needed, you can set up proportions between the windows themselves and the windows and the screen to give your application a consistent look no matter what the language.

In general, it is a good idea to use screen proportions. As alluded to earlier in the discussions on device independence, you should query the size of the screen in whatever units of measurement you are interested in, then figure out proportions of the screen to deal with. Use those coordinates and measurements rather than hard-coding numbers. By

using the proportions you can use coordinates based on the device and let the presentation drivers optimize the line and curve drawing. This way, not only are you generating sizes for objects at runtime to compensate for different-sized strings in different languages but you are also compensating for different resolution devices. Note that this does not mean that you are becoming device-dependent; on the contrary, you are simply making your windows the same relative size to each of the different displays users can have. The other nice side effect is that by using coordinates (which you must pass to drawing and sizing calls) based on the proportions obtained by querying the device you can generate parallel-looking lines and concentric circles as well as line up text without too much work.

Of course, this is all more work than just hard-coding coordinates or string sizes and letting the user size windows to see all of the text, be it all in English or any other language, but it gives your application a clean and consistent look. You don't want to spend so much time on function and classy data manipulation only to have a sloppy-looking window. By starting all of your development with proportions and readable text separation in mind, your applications can have this clean, consistent look.

You can see how difficult it can be to add this function later on in the development. However, at the beginning you can design a few small functions that calculate all of your string sizes and proportional fonts so that you only have to code it once; then you can just sprinkle calls to these functions throughout the rest of the code, making the process painless while gaining the full benefit.

Resources in DLLs

Another useful fact about DLLs is that you can store resources in them. This feature serves a twofold function. First, of course, is that you can share resources, such as pointer, bitmaps, or even stringtables or window templates between processes by putting them in DLLs. The other thing you can do is—yes, I'm saying it—make your executable file smaller.

By placing your resources in a DLL, you will not be binding them to your executable file and by doing so, you can make your executable file completely language independent. If you put all of the resources, including strings in stringtables, in a DLL, all you need to do is ship a different DLL to ship an international version of the application. This will help you truly separate your code from the translatable resources such that you can even separate the building of the resources from the executable while making packaging, distribution, and maintenance easier. This means more flexibility at a lower cost. But it is important that you structure and design the code like this from the outset. Once again, adding it in later is more expensive.

Structuring Your Development

Now that you have seen how and why to separate the readable text from the executable code, let's take a look at how you can structure your development to make this as easy as possible.

The code and text separation have already been taken care of. You have separate files for the code and the messages, such as in RC or .MSG files, regardless of whether you've chosen to put them in DLLs. How can they be combined and managed as easily as possible? The answer is to use a combination of `MAKE` files and environment variables. You can structure your source code tree in such a way that you can have your code in one place and the messages or resources in another. Use the `MAKE` files and environment variables to point to the resources and messages for the language you are building.

Let's look at an example. Figure 17.1 depicts a sample source code tree. Root is whatever the root of your development tree is. It can be the root directory of the build drive or any subdirectory. It is the highest directory common to your entire development tree. Under this Root directory are directories structured as discussed in Chapter 13. Each executable or loadable module in the case of DLLs, has its own subdirectory. There is a common directory for include or header files as well. For each component, or loadable module, there can be a set of text or translatable resources. As you can see, for each component's directory

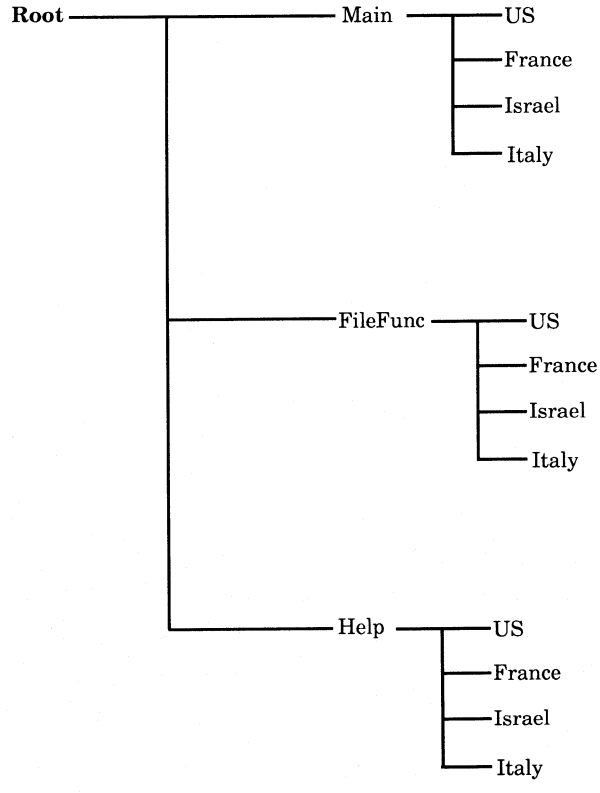


Figure 17.1 NLS source code tree.

there is a set of subdirectories, each representing a different language. This is where you will put the RC files, `.MSG` file, or other files with translatable items.

Now that you have these messages separated both in the code and in the files in your build tree you need to bring them together and make building the language versions as easy as possible. The way you can do this is with the `MAKE` utility and `MAKE` files.

After doing so much work to keep your code independent of the runtime environment, one thing that you don't want to do is to make it dependent on the build environment. You can specify path and directory information in your `.DEF` and code files, but you can do it more easily

```
# Sample makefile excerpt for NLS applications
# This sets the variable LANGUAGE to be the language being
# built.
# You can also set environment variables and pick those up
# if you wish.

LANGUAGE=US

demo.res:  .#(LANGUAGE)#demo.rc demo.h
    rc -r .#(LANGUAGE)#demo.rc
```

Listing 17.1 Makefile excerpt for NLS resource building.

in the MAKE file. In MAKE files you can set local environment variables and let them control where files are pulled from.

Listing 17.1 shows a sample makefile. As you can see, there is an entry in the path to pick up text message files for the resource compiler and the message file building from the environment string. In this way you can simply set an environment variable such as

```
SET LANGUAGE=FRANCE
```

which will change the environment string for the system, and since the makefile picks up this environment variable the text files will be picked up from the FRANCE directory. The build process stays the same, with the same make files just a different environment variable.

SUMMARY

Non-English versions of applications are usually not thought of early enough in a product cycle to enable easy development. By thinking about it and designing the code with the flexibility to incorporate the languages seamlessly, you add new dimension to your product's features. After all, English is not the only useful language for software, so designing in the flexibility at the outset keeps your options open for any translation

you may want to do. The real key to making this notion successful is to assume language neutrality. Refer to Figure 17.1. English should be treated just as any other language; by doing so, you are not likely to miss much when separating the text from the code.

Another important task, with any part of OS/2, is to query the system for what the application needs, as opposed to anticipating what fonts may be available or how big a window needs to be. This gives your programs flexibility and your users the freedom to use the hardware they choose, not to mention just taking advantage of a major feature of OS/2. Once you do the initial work, you will be free from any new hardware that may come out or any crazy font for which the user may have a fetish. You can use whatever your users want and maintain that crisp look to your application.

Another part of the job is to make the build environment flexible and work hand in hand with the separated files. By using environment variables all you need to do is to set the language environment variable and set the `MAKE` in motion. Since English is being treated the same as the other languages the build process is exactly the same for all languages except for the name of the language you type at the outset. Using this process, one set of code, one makefile per component, and one process make your overall development easier and faster; most importantly, you can develop your applications for different languages without much more than it takes to translate the text.

SECTION V

Performance

Now that you have seen the ways to put the code together, the functions to add, and how to structure the system, the time has come to look at performance issues. Performance tuning with respect to your application is generally a two-part process. The first part involves the base function. This encompasses the overall thread and memory management with respect to the tasks that must be performed to manipulate the data within the application. The other part involves tuning the windows. This encompasses the management of the windows and their visibility.

Painting can cause windows to flicker depending on the speed of computer and the speed of the display along with the amount of change in the window. An example is filling a listbox with data. The listbox is not available until all the items are inserted. Also, due to its nature (and its window procedure), a listbox flickers when items are inserted anywhere within the visible area due to item movement. Why not just create the listbox as invisible, fill it with data, and then make it visible? This is just one of the techniques you will see in this section on performance.

CHAPTER 18

Base Tuning

Base tuning refers to all of the core functions of the application such as memory management, resource allocation, load time, and thread management. Back in Chapter 15 we discussed breaking tasks into threads and how best to parallel tasks. The OS/2 memory management package was explored along with some ways to structure your memory allocations. This chapter will dig deeper into structuring your memory allocations and managing your threads. You will see how to manage your threads' priorities based on their job in relation to the other threads in the system and the application.

MEMORY TUNING

Memory tuning was touched on in Chapters 12 and 15. Recall that in 32-bit OS/2, all memory is allocated in 4K pages. Even a 2-byte allocation will give you a 4K page. By analyzing your code as you develop it you can determine working sets, locality of reference, and positioning of reference.

Code and Data Working Sets

The first thing to look at is how your code works within itself. You do this by looking at both the logical structure of the code (how functions call others) and the LINK map. By looking at how the functions call each other you can determine how the code logically flows. By looking at the LINK map you can see the exact sizes of the functions and how the linker is working with your function "order" and is allocating code pages.

The goal of your analysis is to minimize or even eliminate waste within the pages of code. The data can be managed using the techniques described in Chapter 15. You control the memory allocations for data. The real analysis takes place with the code.

Locality of Reference and Data Positioning

Locality of reference refers to the real location of functions with respect to each other. Of course, you as a programmer cannot control where the pages are placed within physical memory, but you can control within which pages your code resides. The first step is to determine which functions reference which others and how often. Then, use the link map of your build to see how big each of these functions are. After that, the process is relatively simple. Take the functions that reference each other, and draw a hierarchy map. In this hierarchy map, map out which functions reference each other, and write down each function's size. Now, understanding that everything is allocated in 4K pages you can use the `.DEF` file and LINK statements to group functions together that reference each other.

The ideal goal is to keep the functions that reference each other most often on the same physical page of code if possible. Of course, this may not be possible due to the size of these functions or the number of functions that reference each other. Another possibility is to move some of the data structures associated with some of the functions to global data rather than instance data. This way the function will be smaller. By

doing this you can also control the locality of the data with your internal memory manager. Of course, there will be times when you must have instance data automatically allocated with a function invocation such as with reentrant functions, but this approach can still be of value.

In moving functions around to keep interfunction references within pages you can also minimize waste. When looking at the functions to keep together you should also look at their sizes. There really is no optimal way to figure out whether to move a function that is referenced more often to another page so that you can fit another two functions into the first page. You need to make those determinations based on the size, the function usage frequency, and your knowledge of how the code works. Chances are that if you are splitting hairs you won't generally be wrong. At that level of detail one choice versus another may mean nothing more than a few milliseconds.

You can organize the functions on a strictly mathematical basis (how many references to a function versus how much is wasted moving to another code page), but the real key is knowing how the code is referenced. For example, if a particular feature of the application is used more often than others but the others would save space if moved to a lesser-used function's location, you need to see how much savings you get in either case. If the bottom line is nothing more than wasting a few K of memory throughout the application, the locality of reference of the functions is more important. The few K is not a big deal, but if the application has to keep bouncing the same few pages in and out of memory in a constrained system, your performance will be hampered.

There is no formula for making these decisions. The function sizings and references are your guides along with your intimate knowledge of your application.

Code Size versus Path

One of the things that was uncovered during the performance analysis of the operating system during the development of OS/2 Warp was the trade-off between code size versus code path length.

What I will say you should also realize follows the trend of the computer industry. Over time, processor speed has increased dramatically, while disk and memory storage have remained more of a premium. As such, it is vastly more efficient and economical for your users if you optimize your application to be as small as possible. You usually have spare processor cycles with which to unpack code or set things up that will help you save space.

I'm sure you also realize that not optimizing for code path will also cause an increase in size because of a greater number of instructions but there are ways you can do both. A perfect example has been mentioned several times in passing thus far, and that is code packing.

By packing the code using the `PACK` options on the linker and the `-p` option on the resource compiler you can make the code smaller, but not add instructions to the application, and the spare cycles in the processor are used by the loader to unpack the code on-the-fly.

Another technique in this area from which you can benefit is in the storage of data. If, for example, you use many graphical files, you may want to compress data in the files to read less from the disk and unpack it only when needed.

The real key is that memory is usually at a premium in a computer, especially in a system such as OS/2 where the user is running many programs at once. Disk space, while cheaper than memory, is also one of the slowest devices in the computer. Keep transfers small, and code and data compact.

DYNAMIC LINK CONSIDERATIONS

The first thing to understand with dynamic linking by import (as opposed to `DosLoadModule/DosQueryProcAddr` runtime dynamic linking) is that every function call requires *fixups*. Fixups are records that point to the real functions in DLLs. As outlined in Chapter 5, DLLs contain external reference records that point to functions in the DLLs. Each of these function calls must have some kind of fixup at runtime to gain addressability to the actual function.

There is a series of tables associated with executables as well as with DLLs that cross-reference the functions in DLLs that are called by the application programs. These tables are stored in the extended executable file header and must be interpreted and resolved.

The fixups are calculated at load time and are kept in swappable application memory. When the call is actually made to the function, the fixup tables, if not present, are brought back into memory, and the function call is resolved and executed. In general, this is not bad but, by understanding this, you can see ways to improve performance with some simple changes to your code. The way to do this is what I call "API aliasing." What this means is to have only one function in the application that calls a particular DLL function. Whenever you want the services of that API you call your own function (the alias). You will create a function for each API function call you make in the application and give it a name such as `MyDosAllocMem` or `MyDosCreateThread`. It would take the same parameters as the real API and return information the same way the API does.

The advantage of this is that there is only one fixup per API function used in the code, as opposed to having fixups all over your code for each API function call. The former approach would just add size to the fixup tables, take time to resolve, and add size to the application working set (recall that the tables remain in the application's swappable memory).

An important decision in aliasing the API functions is how many of them you actually call. If you use a particular API only a relatively few times (say, five or less) throughout the application, then the overhead of writing another function to alias the API is probably not worthwhile. However, for functions that are called on a regular basis an API alias will help your performance by reducing the size of the tables and the number of fixups required. In writing the API aliases you create a bunch of near, direct references to your own function throughout your code. This keeps each (indirect) invocation of the API out of the fixup tables since there is only one place in the application where the API is called. This makes your working set smaller, the executable file smaller, and the load time faster.

Another technique you can use to reduce the working set of your application is to look for functions in your DLLs that you reference infrequently. DLL references to system code are really of no concern since most of that code is resident during system operation anyway.

When structuring where code resides in your DLL files, apply the same locality rules you do for the executable files. For example, group lesser-accessed functions in the same pages. Pack the pages efficiently, with functions that reference each other, just as you would for the executable file and for the memory being used for application data.

Your own DLL references can be set up to optimize locality of reference for functions that call others within the same DLL. Earlier we discussed deciding which functions to put in DLLs. Performance also plays a part in this decision. If you have a function that can fit in a page close to its caller, you may be better off in keeping that function in the `.EXE` rather than in the DLL. This is a moot point if the function needs to be shared between processes (which is the major reason for putting a function in a DLL anyway). The other half of this DLL optimization is to delay the loading of a DLL until it is needed.

By loading a DLL when you call a function in it by name you take the performance hit at load time, which may be more desirable, but you'll also increase the working set size of your application because that DLL will be in use and loaded all the time. It can be swapped out in a constrained situation, but look at the net result. You'll load the DLL at the outset to take the hit at load time. However, if the code in the DLL is hardly ever used (or not used for a long time) it is likely to be paged out. If it is paged out when the actual call is made into it, it will need to be paged in again. Now you have loaded the page twice to use it once.

By delaying the load by using `DosLoadModule/DosQueryProcAddr` you load the DLL only when you need it. Since you'll take the performance hit if the code has to be paged in at the time a jump into it is made or if it is being loaded for the first time when the jump is made, at least you won't load it in at application load time as well. There may be times, such as with frequently used code, when you may want to place a function in a DLL despite this fact, but now you know the ups and downs of both approaches.

Another important consideration is the life span of a DLL. That is when you reference a DLL, how long after the function is used should the DLL be kept around? If you are calling the functions by import or name you have no choice. The DLL is loaded for the life of the application. Of course, it (or parts of it) may be paged out, but that still consumes system resources. When it comes to DLLs loaded via `DosLoadModule` the choice is completely up to you. A good example is a "help" DLL.

Help code is usually used for a short time and then released. However, if the user is in some sort of learning mode, such as when an application is first being learned and used, help is requested more often. You don't want to blindly free the DLL that contains the help code after each help function call, because if you make several calls close together you'll spend all the system resources loading and unloading that DLL. You also have to consider times when an experienced user may need to find only one piece of information and will not request help again during the application's execution. In that case you don't want the DLL hanging around. You could put in a switch for something such as an "expert mode" whereby the DLL would indeed be freed after each help call since an expert may not need it more than once or twice. Users who keep the expert mode switch off will keep the DLL loaded because they are likely to request help again.

A help function is only one example. In general you should look at how and where your DLLs are used and how long they should remain loaded. Keep in mind that it takes resource to load and unload DLLs; you must weigh this against the size of the file. If you have a small DLL, for example, it may be feasible to keep the DLL loaded since it will take up only a small amount of swap space, whereas if the DLL is large you will need to make a trade-off decision. This may also lead you to make a set of smaller DLLs rather than one large one.

All these facts must be considered when setting up your DLLs. It is easy to forget, though, that the main reason for putting code in a DLL is to share it between processes. If the code does not need to be shared you should simply keep it in the main executable file and manage where in the file it lives (via the `.DEF` and LINK statements) as outlined earlier.

The same method of managing the code in the executable file can be applied to the DLL as well.

The biggest thing you should remember about DLLs is that you need to use them for a purpose. Sharing code among processes, such as print function code that will be called from the application as well as the Workplace Shell (for the subset function described in Chapter 11) is a good example. Since we are talking about performance here, I will reiterate that putting code in a DLL to make the executable file smaller does not help and in more cases than not, will also hurt you.

Delayed code loading; sharing code among processes; and controlling the in-memory life of the code are all uses for DLLs. Use them properly and you will help the performance of your application and the system immensely.

MESSAGES AND OTHER RESOURCES

Yet another consideration in memory management is how your resources are linked into the executable. Important items of note are message files. If you use a message file without statically binding the messages to the executable with `MSGBIND` but still use the message APIs you will get a message "segment." This message segment is about 44 bytes and will eat up a full 4K page. This is a good reason to use `MSGBIND` if you plan to use `DosGetMessage`.

With respect to resources you can use an option on the resource compiler to pack the resources together. By using the `-p` option when compiling the RC file into the `.RES` file you will pack the resources so that they will be a bit smaller on the disk. Since application load time is usually gated by the speed of the disk, the smaller the resources that need to be loaded off the disk (resources are usually large), the faster the application will load. The resources can be unpacked in memory fairly efficiently. You should be using the API `DosGetResource2` for packed resources.

Another tip for using resources applies if you are using icons or bitmaps. Each icon or bitmap file can contain many versions of the

resource, such as VGA, XGA, CGA, and device-independent. In general, you need the device-independent version to cover the range of display devices you'll be dealing with. This can save you about 6 to 7K per icon or bitmap you have in your set of resources. Of course, you won't have all of the fancy colors or high-resolution bitmaps as if you stored all of the formats, but if you are concerned about memory and disk space, this idea offers another way you can economize.

The next step is to look at managing priorities of the threads within the application.

THROUGHPUT USING THREADS

The major tasks and subtasks have already been analyzed and separated into threads. Let's now take that a step further and discuss the priorities of these threads and how they interact and block on system events.

Thread Priority

Thread priorities are not as complicated as they look. Recall that the scheduler is a simple mechanism. At any moment in time the highest-priority thread in the system that is in the ready state is the one that is running. It runs either until the end of a time slice or until a thread of higher priority becomes ready. At the end of a time slice the priorities of all threads are examined, and the highest priority thread is context-switched in to run. If the thread that just finished is still the highest in the system it is put in for another time slice. Throughout this process other threads' priorities are modified based on whether starvation is occurring, whether a process or focus switch is made to move another thread into the foreground, or whether one of the other criteria for priority boosts (such as returning from I/O blocks) is met.

Threads in the regular class have their priorities modified by the system as just outlined. However, threads in any of the other classes (idle, fixed-high, and time-critical) are set. When any thread is first created it lives in the regular class. Once `DosSetPriority` is called to move

a thread to one of the other three classes, the system will not mess with the thread's priority unless another call to `DosSetPriority` subsequently sets it to the regular class again.

Here is where you want to work with the priorities. Just as with any other part of the application, there is no real formula for determining what the priorities should be. There are some general guidelines, however. The first thread to look at is your main user interface thread, the one that lives in the `WinGetMsg` loop and dispatches messages to the window procedure. This thread should remain in the regular class. If the application is in the foreground, the system will boost the priority of this thread enough to remain very responsive to the user. Don't make it higher than regular; otherwise, when this application is put in the background any messages posted to it will get higher priority than even the foreground threads of other applications. Obviously, you don't want to move it lower than regular class either.

The next threads you want to look at are I/O threads. Usually these threads voluntarily block, for reasons that go back to the discussion on device drivers. When a thread requests some I/O, a series of calls is made, ending up at the device driver. When the request is passed from the device driver to the device, the device driver blocks the thread until the physical I/O is complete. The thread is then made ready when the interrupt comes back from the device. Because these threads will block often and I/O is usually a slow operation, you will likely want to set the priority of your I/O-bound threads higher. You will not use up CPU time needlessly because these threads spend more time blocked than most others. If you leave I/O-bound threads in the regular class you will have to rely on the system to bump the priority on an I/O boost, which may or may not be higher than other threads running. By setting the thread someplace low within the fixed-high class, you keep the I/O threads ready to run whenever they come out of being blocked, but not so high that they will preempt more time-critical operations.

The next type of thread is the CPU-bound thread. This is one that has no or very few dependencies on an I/O device. Such a thread might be one that recalculates spreadsheets or empties buffers. These threads do not usually block until their task is finished and they wait for another.

If you bump this thread above the regular class it will always run to the end of its time slice unless another even higher-priority thread comes along. You should leave these threads in the regular class. If you do find reason to move them higher, keep them low within the fixed-high class.

The higher priorities such as those in the time-critical class should be reserved for communications threads and others for which data will be lost if they are starved when ready. If you move your recalc threads into the time-critical class just to improve performance you will hurt other applications in the system. Communications threads, however, should be in the time-critical class. If this type of thread does not get serviced when it is in the ready state, the communications line to whatever is on the other end could be dropped. Applications of these threads include standard modem communications packages, data acquisition equipment, or even pipes between processes.

When deciding on priorities to use for threads, keep in mind not only how the thread will interact with other processes in the system but also how they will interact with the other threads in your application. For example, consider one thread that reads or writes files and another that fills and empties buffers for the I/O. If both threads are of the same priority (outside the regular class, otherwise the system would modify the priorities) you can end up with a situation in which the CPU-bound thread starves the I/O-bound thread. If the buffers are filled when the CPU thread comes in, it will run to the end of its time slice unless the buffers become empty. If this thread is of the same priority as the I/O thread, the CPU thread will be switched in every time the I/O thread goes to read the disk (which is all it does). The I/O thread, coming back from a device driver I/O request, then has to wait for the CPU thread to complete before it can do its work. Of course, if the buffers empty before the CPU thread is done with its time slice, it will need to block (you need to do this through your thread synchronization). In this state you will not be using both threads at peak efficiency. The I/O thread will send off a read request and block. The CPU thread will be dispatched and will empty whatever is in the buffers until either the buffers are empty or the I/O thread is ready to run at the end of the CPU thread's time slice. Now the buffers get filled with the full contents of the read buffer, and another

I/O request is sent (unless, of course, the I/O thread's time slice ends, in which case the CPU thread comes back in). Now, when the I/O thread blocks again, the CPU thread empties the buffers, which will usually not be full. You can see how this thrashing can hurt performance.

It is essential to look carefully at the scenarios and interactions between your threads carefully to be sure you neither starve your own application nor give yourself a higher priority than threads that truly need it.

One other note about threads: When using idle-class threads, it is important to set the priority back to the regular class before calling `DosExit`. The reason for this is that if you call `DosExit` in the thread in the idle class, it will wait for the system to be idle before calling its exit list routine. This will prevent the application from terminating. Therefore, before calling `DosExit` in any idle-class threads, change those threads' priority to regular class.

PACKING THE EXECUTABLE

Another item that is often overlooked is the executable file size. There are options on the linker that allow you to pack the code, which gives you the advantage of a smaller executable. This may seem trivial, but smaller executables on the disk means fewer diskettes for your application, which means lower production costs. More important than the cost of a diskette is the hard-disk savings for the end user. The real cost is that the code has to be unpacked at load time. As mentioned just before, loading code is usually an I/O-bound operation, and there are spare CPU cycles to unpack the code. In the long run, you'll save by packing all of your executables.

DLL PLACEMENT

There is one other subtle point about DLLs that does not affect the code so much as installing and invoking the application. It involves the

LIBPATH statement in the CONFIG.SYS file. The LIBPATH is a string read from CONFIG.SYS and set in memory during system initialization. This environment variable tells the system in which directories DLLs live. Usually, the OS/2 DLLs live in the OS2 DLL directory, but some applications install their own DLLs there or create directories of their own, which in many cases can hurt system performance.

When code importing functions need to load the DLL via the API DosLoadModule, the full path and filename of the DLL is supplied. However, when a DLL is loaded by code importing functions by name (such as just calling DosAllocMem), the directories in LIBPATH environment variable are searched until the DLL is found. Your users can encounter performance problems if all of their applications add entries to the LIBPATH statement. Look at the application that adds its DLL path to the end. Each and every directory in the LIBPATH would have to be searched before these DLLs are found, which leads to applications trying to add their paths to the front of the LIBPATH. This can present other problems, since the first information added can wind up at the end (if every application did it), even if it is the most frequently used code, which would cause the performance to deteriorate.

The OS/2 system installation program offers help for this problem in that it puts a dot (.) as the first entry in the LIBPATH. This dot indicates that the first place searched for DLLs should be whatever the current directory is. (The file system representation for the current directory is the "." character.) This feature is a tool for applications to use.

Quite simply, when installing the application you should create a program reference object to start the program. In that program reference object there is a field that represents the working directory for the application. By using this to point to the directory that contains the application and its DLLs, and starting the application from the program reference object, you implicitly put the directory for your DLLs in the LIBPATH for only that application, without affecting any other. When the program reference object is opened, the current directory is set. Because there is the "." in the LIBPATH, the first place checked for DLLs is the current directory in that session. Any DLLs specific to the application that contain functions imported by name are searched for in that

directory first. Of course, the system DLLs are found in their standard place (which, by the way, is usually at the front of the LIBPATH right after the "." and is usually loaded already anyway).

By using this technique you will help your application's performance without cluttering up the user's LIBPATH and potentially hurting other applications or even the system's performance. As the user adds more and more applications to the system this concept becomes more and more important.

SUMMARY

So far, you have seen base tuning, which includes the threads, memory management, and working sets in terms of code and data, as well as other small yet important functions. By managing priorities of threads you can further enhance the performance of the application and increase throughput. By watching what they do you can make them perform efficiently without taking system resources away from background applications. In minimizing working sets and optimizing locality of reference of functions you keep your application running efficiently even in low memory conditions, and you don't waste space, which keeps your own code streamlined.

Another technique discussed in this chapter to improve your application's performance without hurting anything else involves proper DLL placement. There are not too many things in life that offer benefits with no real downside. This is one of them.

In the next chapter we will look at how to make your windows appear crisper and run more efficiently.

CHAPTER 19

Visual Tuning

Throughput in the application is only half the battle in creating a fast application. In addition to having it be fast under the covers, you need to make it look fast. It doesn't make much sense to have code that can crunch data at blinding speeds if the visual update can't keep up. Displays have limited speeds, and of course there is code that must be executed to refresh the display. You also have the overhead of the device-independence of the presentation drivers and the engine. There are ways you can optimize your code, and for the cases in which operations are just plain slow there are some tricks to make the windows at least appear faster.

WINDOW TUNING TIPS

Along with ways to speed up the code there are things you can do with your windows such as prefilling listboxes or creating invisible windows. This section will show you some ways to keep your windows flashy as opposed to making the user wait around for visual refresh.

Keeping Windows Around

The first thing to look at is how often a window is used or displayed—not a particular button but a standard window. Creating a standard window is fast, but it does take time. If you have a window that is used often you should create it the first time it is needed and, rather than destroy it, hide it and reshow it later.

This invisible window concept provides the basis for making PM run as quickly as possible. Screen updates are inherently slow, especially when graphics are involved. Since everything in PM is graphics-based the best idea is to build the graphic in an offscreen area and then show it. The `WS_VISIBLE` bit is turned off by default when a window is created. You can turn it on either by specifically specifying `WS_VISIBLE` in the creation flags or by using `WinShowWindow` (or `WinSetWindowPos`) after the window is created. `WinShowWindow` and `WinSetWindowPos` make an existing window visible or invisible by simply changing one parameter. When a window is invisible it will not receive any posted user input messages (since there is no way for the user to give it the focus), and it will never get any mouse messages. However, the resources for the window are allocated and/or loaded the whole time. If the window is created and just invisible, showing it is much faster than creating it each time you need it.

There is a downside to this approach: Windows consume resources, and there are limits for the system. Don't create every window you may ever need and hide them until needed. This technique should be used for just some of your most commonly used windows. Additionally, if the windows are different in some respect, such as the items in a listbox within the window, you probably don't want those windows to hang around when not needed. The reason is that you'll have to build at least some part of it every time you need it anyway, so there is not much reason to use the overhead involved in keeping it around.

Filling Windows Invisibly

Taking this approach a step further, you should do the time-consuming visual work invisibly as well. This prevents windows from flickering.

It also gives a more intuitive look to the windows. When you update windows, data usually needs to be redrawn. In some cases, such as when you insert data into a listbox, more than just the data has to be moved. For instance, when you insert data into a listbox, data must usually be moved to preserve order in the control. Every time you insert an item you potentially have to move all of the data down in the listbox, and in any event the listbox will need to redraw itself. This will cause the window to flicker, creating an unpleasant look to the window.

By creating the window as invisible, filling it, and then showing it, you eliminate the flicker since all items were inserted while the window was invisible. When you subsequently make the window visible it will look exactly as it would if you had filled it visibly. The other nice effect this has is to let the users know when the list is filled with data and is available to be used. The reason behind this is simple: If the window is not visible until it is filled, the users can't mistake the window for being ready until they can see it. A listbox that is filling may or may not be flickering depending on the size of the listbox, positioning of the data, and so on. The user may not know whether the listbox is ready. By not showing the window until it is ready the user won't become frustrated clicking on a listbox that won't respond.

LETTING PM MANAGE WORK

You should always let PM handle as much work as possible. The PM code has been (and will continue to be) optimized and improved upon over time. In addition, each of the system-defined classes has been optimized to work with each other, and it is the operating system developer's responsibility to ensure that they continue to work well together.

Now that the PM subsystem is fully 32-bit code, the window management code is faster than before and many of the limits on window handles and other common data areas (the PM heap) are far larger than in versions prior to OS/2 Warp. Of course you should still ensure that your code runs well on earlier versions of the operating system, but you can now take advantage of more of the native OS/2 functionality than

before, such as larger memory objects and increased system limits. This gives you even more incentive to be lazy and use PM's services wherever available.

I have seen code in which a frame that gets a WM_SIZE, calculates the new sizes of and sends messages to the title bar, minimizes and maximizes buttons, and so on. This is a waste of resources and is unpleasant to code. The reason the standard window exists is to handle this function. The reason controls tell their owners about certain events is to allow the system to change, say, the highlighting of a button.

The basic premise of Presentation Manager programming, just as with object-oriented programming, is to handle only the functions you are interested in and to let the default window procedures (super- or metaclasses) do the rest of the work.

It is often tempting to subclass windows and objects to control everything going on or to send messages yourself as opposed to using the established APIs. However, in the long run these things will cause you more problems than they are worth. By sticking to the established APIs you insulate yourself from any message protocol changes in a future version of the operating system. You also take advantage of the optimizations built into these APIs. For example, there are some APIs that start other threads under the covers to accomplish work. There is no way that you as a programmer should know that. Quite simply, unless you need to modify an existing class, don't subclass. Unless you need to send a single message or control the flow of messages for a window, use an API if there is one to suit your need.

YOUR OWN MULTIPURPOSE CLASSES

Another technique to help the PM part of your application perform better is to design a multipurpose window procedure wherever possible. For example, if you have a window or set of windows that need to behave similarly, why not define only one class rather than several? Let's say these windows behave almost the same, just like all instances of the WC_BUTTON. With WC_BUTTON, all instances (such as radio and push

buttons) behave virtually identically. They have text, they show highlight emphasis when they have the focus, and they all send a WM_CONTROL to their owner with their window handle and their ID when either the Enter key is pressed or they are clicked on. There is no reason you can't do the same thing with your window procedures and private classes. You can implement a single class with styles. The styles, which you will define as well, can have their identifier stored in the QWL_USER of the window words. Inside the window procedure you should just query the window words of the window receiving the message to see which style it is. This is what happens in the WC_BUTTON window procedure.

You can even use OS/2's predefined window classes to form the base of your own custom classes. Let's say, for example, you want to create a set of windows that all behave similarly to the system-defined entry field window class. You could set a style for the entry field windows, but let's say that you want these windows to be formatted for phone numbers. There is no window style defined by OS/2 for that function.

You want the full function of the entry field but only want this one change. You could write a whole new window procedure to do this, but who knows which functions of the system-defined entry field you'd miss? You would need to understand every single function of the entry field and then reimplement it. There is an easier way.

You could use the technique of subclassing windows to provide this function. Subclassing windows is how a program can replace the window procedure for a single window and point the window to the program's own function. This function can deal with the aspect of the function that departs from the default behavior and pass all other data onto the default procedure. You can use this to inhibit data or even modify data before it gets to the default window procedure. This is effective for one window, but let's say you have many of these windows you need to create. You don't want to have to subclass each instance of the window. There is a better way.

You can create your own class that is a subclass of a system-defined class. To do this, simply call WinQueryClassInfo for the class in question (such as WC_ENTRYFIELD or WC_LISTBOX) to obtain the address of the window procedure for the class. Next, create a window procedure that is the

same as if you would be subclassing only one instance of the window. At the end of this procedure, call the function address you obtained in the previous step rather than `WinDefWindowProc`, which you would call in other, more generic window procedures. Now register a new class name with this function as the window procedure associated with it.

Now, when you want these custom windows, call `WinCreateWindow` with this new class name. What you have effectively done is to create a new subclass of a system class for your use.

By using this technique you will have less code to write and maintain, as well as a smaller code footprint in memory and a smaller executable size on the disk. You are letting OS/2 and PM do most of the work for you.

SUMMARY

There aren't many techniques for optimizing the window management side of an application, and the ones that exist are relatively simple. Don't be sloppy with the little details. Use the techniques of this chapter to put the finishing touches on your application. You know it processes data as quickly as possible, so here is where you can show it to the user.

SECTION VI

Testing and Code Change

Testing and changing code are not new concepts. Everyone who has ever written a program has had to test whether it worked as intended and then go back and change the code. Recall our discussion at the outset of this book in which the 60-30-10 rule was introduced. Testing methodologies are simple and precise. You can never test every permutation of what may be encountered in the real world, but with proper modularized testing you can feel comfortable with your application's quality.

Changing the code is a vital part of the development process. You will inevitably have to change the code as a result of testing. Intelligent code change will eliminate headaches, whereas applying quick fixes to the code will cause you nothing but trouble.

You need to keep a proper perspective when changing code. Any change should be in accordance with the design. This is easier said than done, especially with product schedules and costs, but it is better to "pay me now than pay me later."

CHAPTER 20

Testing Methodology

Testing methodologies have remained fairly consistent since the introduction of computer software. Early on in the computer age, testing was simply a matter of finding out whether the program did what it was told. Later on, as programs became more complex, test programs had to change accordingly. Today, programs are generally still procedural, following the divide-and-conquer approach whereby a large task is broken into smaller, more manageable tasks. We have already discussed the process of designing tasks based on the functions you wish to support in the applications. Testing is the reverse of this process.

The first thing that must be tested is the smallest unit, usually the function. If you have done your design completely you know what each function does and what its possible inputs and outputs are. These are all documented (right?), so your test group can read the design and create test cases to verify that all units function as specified. Once the functions of a component are complete you will put them together in their module, generate a testable object, and again, referring to your designs, try all possible inputs. The final phase is to test the entire system or application.

Here's where things can get complicated. Users will do things with your application that you never expected. You must test not only the expected operations but also things that make no sense at all. When looking at the code, the functions may work, but when putting them together with others and then introducing multiple threads with them, you can wind up with interactions that you did not expect. If you spent the proper time on the design, however, none of these things should surprise you; you should be able to deal with them swiftly and efficiently.

SCAFFOLDING

Scaffolding is a technique by which you can test functions for which the functions that call them are not complete or tested. This is accomplished by "hacking" together code that calls the function under test with the inputs it expects. Just like a real scaffold, this code is a temporary structure to hold up other functions and test the units as early as possible. It should be thrown away immediately after the testing is complete. Many times programmers forget and leave the scaffold in the build. However, as soon as the testing is complete or the function it represents is written, it should be discarded.

TESTING UNITS

Early on you wrote down the main functions of your application and broke them down into several large tasks. Those tasks were broken down further until you ended up with small, manageable functions. Each of these functions has a fixed set of inputs and outputs. In general, they were not the functions that deal with the users, but rather the ones that manipulate the data within the application. As such, treating the functions as miniprograms and bombarding them with the permutations of inputs is a relatively simple task. The easiest way to do this is to replace the function name with `main()` and make the parameters passed into it come in from the command line when the program is invoked. Alter-

natively, you can use the scaffolding technique outlined earlier. Some people prefer to use a debugger to just test the code and trace execution. This method can become tedious, however, and can take more time than is actually needed.

Function or unit testing should be done during development and should adhere strictly to the design specifications.

TESTING MODULES AND COMPONENTS

Once the units of a component are verified, the next step is to build the component and test it in the same way as the units. At this point you still have a good handle on virtually anything that can be input to these components. However, there is beginning to be some degree of uncertainty in what can happen, such as in the order of events or unexpected things the user can do. With respect to the user, there is not much that is beyond your control at this level. The components generally interact with each other. Only the user interface component deals with the user.

In general, the things you want to look for in the component test phase are the interactions between the functions within the component. This includes the data and function calls being passed between the functions as well as any multithreading within the component. Since you have already tested the functions for their inputs and outputs before getting to this phase, you really just need to verify that any multithreading within the component adheres to your designs. You will likely want to scaffold code here not only to test all expected inputs but to simulate error paths as well.

TESTING THE SYSTEM

Once the components are verified, the final test phase is to put it together as a system. You've already verified that all of the functions and the components to which they belong work as expected. At the sys-

tem test phase you should really need to look only for stress situations and overall application thread interactions. You should "gorilla test" the application—that is, take any kind of input (rational or not) and throw it at the system. This approach ranges from the simple case of using invalid file names to possibly writing some simulation code that pumps user input at the application in rapid fire.

You need to see how the application handles user input and manages the threads. You want to carefully analyze your thread design in real user situations as well as anything else a user may throw at it. You want to look for race-for-resources conditions between threads as well as determine how well you have prioritized the threads.

As you can see, the functionality of the application can almost be taken for granted here, because if your design was correct and the testing passed the design specifications the functions will work. Of course, you should verify the functions in the system test phase, but this should be a minor part of the test.

In the final application or system test you want to verify that the threads behave as you intended and don't fight for resources. You also need to see how the threads interact in terms of priority and to determine if any of your threads are starving others. You want to look at how you have structured your code loading and memory management in general as well as in low-resource conditions. If you have the type of application that can work on multiple documents you want to load in the maximum number of documents. If you have variable-sized documents such as in a spreadsheet you want to look at the maximum-sized document; you should do this in low-resource conditions as well to verify that you are not thrashing, either in your own code or in the system in general.

One other item to examine with respect to data allocation is leakage. If you allocate memory and then don't need some of it after awhile (as in a multiple-document scenario in which one document is closed) you should free memory. Otherwise, you will only cause the swap file to grow and you will degrade system performance.

The final test is to look at performance and see how you are doing overall, especially in low-memory situations. This is where the perfor-

mance work you did in the last few chapters really pays off. Not only will you be a good "computing citizen" by being frugal with system resources, but you will also perform better in constrained systems.

SUMMARY

Notice that testing of the application is geared toward verifying that the code performs according to the designs. Of course, you can never test every possible user action or every environmental consideration, but by taking the users into account in the design and verifying your code correctness in a test, you can feel comfortable with the quality of your product.

There are those who would disagree and say that testing should be the most intensive part of product development. I agree that testing is important, but by doing the work up front you end up with code that is cleaner and more maintainable without having to spend a huge amount of time and resource testing.

CHAPTER 21

Code Change

Code change refers to anything that goes into your code other than exactly what the design calls for. Code change is anything you have to do to the code after it is initially written and submitted for test. Sometimes you will find a hole in the design during test. Maybe it is something the user may do or want to do that you did not anticipate. Maybe it is an ambiguous area of the application where you intend for the program to work one way but it makes just as much sense for it to work another.

You should not only have the design completed before you code, but you should have it reviewed and approved as well. Your testers will be good judges of what, if anything, has been left out, is ambiguous, or is lacking in the application.

CODE CHANGE ACCORDING TO DESIGN

When it becomes necessary to add in changes you must make sure you don't compromise the design to fix inconsistencies. Each design point has been evaluated, analyzed, and approved based on the design of the rest of the application. It is a link in the chain. By changing the shape or function of one link the entire chain can fall apart. You must review each change that needs to be made and analyze how it can affect other parts of the system.

If, for example, you are getting bad output from one function in one instance, you can't just change the code inside the function and assume that fixing the internals of one function won't affect anything else. It may seem trivial, but you need to examine the needed change to determine whether something is lacking in the design or whether you just have a coding bug. A simple coding error is not as much cause for worry as a flaw in the design. If the design has been documented properly, a code change within a function should do nothing more than adhere to the inputs and outputs you have laid down for it.

Complications can arise when you test the components or the entire system. Usually the most complex problems are exposed here. However, if you have broken down the tasks early in the design phase you should have a fairly easy time of finding which function is failing. Although the scenarios during component and system testing are more complex than those of unit testing, you can go back to the scaffold code if you need to try to isolate the suspected functions. The biggest job is to analyze why the problem is there and to fix it within the design, not just hack code in there so that the code works in the failing scenario.

If you designed properly at the outset, you can rewrite a component or subsystem of the application without affecting the rest of the application. Just view it as maintenance and subsystem replacement before it gets to the field, rather than later on.

You may encounter a situation in which one or more pieces of your application must be changed more than others. Each time you make

a change you take the risk of injecting more defects than you remove. You need to control all changes and try to find where and why you are having problems.

WHEN THERE ARE TOO MANY FOR COMFORT

One of the main things you are looking for is to see if you have error-prone components or places where the design is too loose or fuzzy. In these cases you have a choice. You can change the design on-the-fly (or just continue fixing the problems), or you can go back to the drawing board with that component. This is a drastic measure and a decision not to be made lightly. If you have done the right design work, the only real reason to go back to the drawing board is for an error-prone component. If that is the case you may just need to throw out the current module and recode it. This does not mean that the design has to change. It is possible that the design is fine but the way in which the module was coded is too slow, too inflexible, or just too "buggy."

There is no fixed formula that says that for X defects per Y lines of code you should rewrite a module, but it is usually evident to the testers and programmers which modules need to be redone. Don't ignore this information. No one wants to throw away work and start over, but retaining faulty modules can only exacerbate problems. After all, even if you have only one error-prone component, it has to interface with one or more other components. If you can't rely on what one component is doing you may end up trying to fix nonexistent problems in others.

SUMMARY

This very short chapter is in this book to stress discipline—the discipline to know when a component needs replacement. If the design was done right and the interfaces between components are clean, component re-

placement should be nothing more than writing a new component. This discipline involves being able to make such a difficult decision despite the possible schedule slips.

Component replacement is not restricted to situations in which one fails to make the grade. As technologies advance and new ways of accomplishing tasks are developed you may even want to replace a component in a subsequent release. This does not always mean redesign.

You need to maintain your perspective with respect to the overall picture in your application's life cycle. If you have one week to go in your schedule and one component that is marginal with respect to being replaced, your decision should be obvious. However, if you have a component that is causing nothing but headaches, you should take a long, hard look and possibly even push back your schedules.

Perspective and discipline.

SECTION VII

Installation Programs

The finishing touch to the program (actually, the first impression if you are the user) is the installation program. This is the program that will install and configure your application. Install programs can be as simple as command (.CMD) files, or they can be complete PM programs in their own right. These programs can be used not only to install the application but also to install upgrades and bug fixes, or even reconfigure the whole application.

Installation programs are often neglected, yet they can add an excellent dimension to your application, and they don't necessarily have to be difficult to code. You can use facilities of the application or even use code from the application itself.

The installation program is the user's first impression of the application. It should be the simplest and most intuitive part of your application. The user should almost be able to run the install program without reading the manuals. By making the installation as simple as possible, yet flexible enough so that you can use the same interface for upgrades and service, you lower the learning curve for the application and make users more willing to buy your software.

CHAPTER 22

Designing the Installation Program

The installation program needs to be designed just like any other application. The fact that it will be used only once (or very infrequently) is no reason to just slap it together. You must outline the requirements of the program and what features you plan to provide. You must decide if it will be an all-purpose configuration program or a simple command file. You must determine what medium the distribution will be on and how (or if) the file(s) will be packed. You also must determine the installation program invocation.

Although the installation program is the first visible part of the application, it puts the finishing touches on the application.

JUST ANOTHER (SMALL) APPLICATION

The installation program is another application, just on a smaller scale than your word processors and spreadsheets. As such, you should go

through some level of design. You don't need to go in as deeply as you would for the "real" application, but you need a user interface, worker code, possible multithread considerations, and (as with any program) a level of testing and code change.

You may decide that you don't want to put much effort into the installation program. Maybe your decision is that you just want to copy the files to the hard disk (or *any* disk, for that matter) and have all configuration in the application itself. Maybe your application is so complex that you have a large number of DLLs and other code that require changes to the `CONFIG.SYS` file.

This application must have its purpose designed. If you plan to use it to just install the program once, then stick with that plan. The same rule applies if you plan to use it to install the application and future upgrades, or if you have a series of related applications for which you wish to use the same basic installation code. The bottom line is, don't change after you're done.

USER INTERFACE

The user interface of the installation program is just as important as the user interface of the main application. If the users cannot figure out how to use the installation program, they won't be able to install your software, will be disgruntled even if they can get it installed, and will be likely to return it. This does not even touch on the fact that your support organization will be bombarded with support calls from customers just trying to install the program, which adds to your product cost.

Your choice of user interface depends a bit on your initial decision concerning the scope and future of the install program. If you don't plan any future use for the program and choose to use a `.CMD` file, the user interface is simple. They can type `INSTALL` and let your `.CMD` file do the rest. Even if you need modifications to the `CONFIG.SYS` file, you can write the function into the `.CMD` file, or you can even ask the user to make modifications based on another file you provide. The more work you ask the user to do, the greater the risk of mistakes, however.

If you have grand plans for the installation code (or even if you don't, but you want to provide a consistent look to all of your supplied code), you may choose to write a full-blown PM interface. A PM interface does not have to be as involved as the main application's. It can be as simple as having a window with a menu to set some installation options, such as whether a WPS object should be created or whether the `CONFIG.SYS` file should be automatically updated. It can also be as complex as controlling which files or features get installed and where they all will go.

There is not nearly as much to the user interface in the installation program, and not nearly as many dialogs. Something you may wish to consider is using the application code for part of the install program's user interface. It adds consistency and improves the aesthetics to have your installation program's interface look like the application. Of course, the menu will be different, but the window style, icons, and general look and feel can be the same. You can even share some of the same source code to make this job easier.

There are really no rules for the installation program's user interface. The only suggestion I can give is to consider the overall picture and plan for the program before making any of the user interface decisions.

MULTITHREADING

One subtle yet effective advantage you can gain over other systems and in fact other programs is using multithreading even in the installation program. Just as with the main application, you should look at the installation process from a tasking viewpoint. One task could be copying files. Another task could be unpacking files (if they are indeed packed on the media). Yet another task could be asking the user for the application defaults or writing the changes to the `CONFIG.SYS` file.

Again, as with any application design, you must look at the tasks to see what you can multitask. The most time-consuming task is copying the files and subsequently unpacking. It does not make much sense to multitask just the copying of files. Of course, you might say that copying

two files at once is faster than one at a time. On a fast access data device that supports overlapped I/O this may be true. However, most software is distributed on diskette or CD-ROM, which are relatively slow devices. If you try copying files on multiple threads all you will succeed in doing is making the head or laser beam move furiously across the disk without accomplishing much. However, if you have packed files, it is often useful to copy a file to the hard disk on one thread and then signal the name of the file to another that unpacks the file. In the meantime, the first thread can copy other files. In fact, OS/2's own installation works on a principle similar to this.

Some of the other functions of the installation program are somewhat easier to identify for multiple threads. Writing changes to the `INI` files, creating Workplace objects, and updating the `CONFIG.SYS` file are all such tasks. The only real consideration is whether the installation abnormally terminates before it is complete. You'll either have to figure out a way to undo those changes or delay them until anything else that could go wrong is completed. The biggest thing that can go wrong is if the user does not have enough hard disk space. You can query this before you start, but who knows what else the user is doing while installing your software? Remember that any system query is a static picture of the system at one instant in time. This could change a millisecond later. The disk space you thought you had could be gone by the time you are copying the last files.

Keep these considerations in mind when designing the structure of the program. You may think these situations will never arise, but take it from experience—they inevitably will.

MULTIPLE INSTALLATIONS

Inevitably, the user will accidentally erase some files or "blast" the application configuration in some other way. The user may also try to reinstall after a failed attempt. Users may run into hard disk problems. Then there are some users who just don't like the way they configured the application and decide that they will just reinstall over the old copy to erase their changes.

Consider what you want to do when the user installs over an existing copy. You don't want to create WPS objects all over the place or dump redundant data into the INI files or CONFIG.SYS. You should put a flag somewhere in the system to indicate that the application has already been installed. This can be done in almost any way, and the user can probably erase each of the flags, so you should build mechanisms into the individual "subsystems" in the installation program to detect prior installations.

For example, in the code that created the Workplace object, query the system to see if that object class or template already exists. Before adding information to CONFIG.SYS, scan the file to see if the information is already there. By looking at each step, you can eliminate the need for the flags and not have to worry if the user tries to remove the application by hand but forgets some pieces.

One other consideration is in deinstallation of the program. You should give the users a clean way to deinstall the program if they wish. Of course, no one will want to remove your code from their system, right? Well, it is inevitable, so you should be as clean as possible.

Deinstalling the application should clean up as much as possible. This includes removing INI file entries, Workplace objects, and changes to CONFIG.SYS. It may seem trivial, but it is important to the user. Better to be known as the nice guy than as the application that wouldn't leave.

MEDIA CONSIDERATIONS

Media considerations really depend on the type of application you are writing. For example, if your application is multimedia-based for which CD-ROMs are commonplace, then both diskette and CD distribution are equally viable. The only considerations are cost of manufacturing and the speed of the device for installation.

In general, software is distributed on diskette, which presents speed and size considerations. CD-ROM disks can hold hundreds of megabytes of data, so packing of files is not really needed unless it could improve installation speed. Diskettes, however, are much smaller. You will likely need to pack the files on the diskette and/or use multiple diskettes.

Packing Your Code

Packing files is not necessarily a bad thing. In many cases it will speed the installation since a smaller number of bytes need to be read from the diskette, which is about the slowest device except for some tape drives. The faster hard disk would then be used to decompress the files. Your decision depends largely on the algorithm you use to pack the code. There are established programs that can be used as shareware or licensed at a low cost. It is much cheaper to take that approach than it is to develop new algorithms that may be less efficient at best and buggy at worst. The established algorithms have been improved over time and field testing. Why try to reinvent the wheel yourself? By using the established packages you can save time and money. What more could you want?

"SERIES" APPLICATIONS

When you design the installation program take into account whether you have a series of applications for which you want a common installation program. If you don't have a common program, at least try to have a common-looking program.

As with anything else, you can take this idea as far as you like. If you wish you can design a separate application as being a generic application installer whereby the customers, who are developers, will supply some kind of script file. You can interpret this file and run the program creating dialogs and resources on-the-fly based on the script.

SUMMARY

Installation programs are no different from applications. They are just on a smaller scale. The same considerations apply, with a few twists. There is not much to say about installation programs other than what has been discussed briefly in this chapter, which is intended merely to

give you some insight into the considerations of writing your installation programs. You really need to evaluate your intentions for the program. Will it just install the application by copying files, or will it be a general application configuration tool?

Installation programs can gain the same benefits from the use of multiple threads as any other program, especially if the code is packed. Make your installation programs consistent with the application. First impressions are lasting ones.

Summary and Conclusion

There are many parts to designing OS/2 applications. No book can cover everything. As stated at the outset, only experience will enable you to write robust, powerful applications. Since you can't get an injection of OS/2 experience, this book is the next best thing. Throughout this book I have drawn on many of my experiences with application vendors to explain various design points of applications.

Having supported OS/2 in many capacities in the last few years, I have seen customers who thought that there were flaws in an application or in OS/2. Upon examination, the user's problems occurred not because of a bug in either OS/2 or the application but because the application did not support certain features of the system. These are the kinds of things that programming books can't teach.

We have taken a journey all the way through OS/2 at a level deep enough to understand how things work within the system, how each function and subsystem relates to the others, and how to make applications best take advantage of them all.

We started out discussing why anyone should even write applications for OS/2. There are many compelling reasons to write for OS/2, ranging from the productivity gains you can give users to the productivity gains programmers can get as well. From there the discussion moved into the architecture of the system at a functional level. OS/2 is not a mystery. On the contrary, if you understand the basics and use common sense and simple logic you can figure out how just about every piece of the system works.

The next step was to get into the design of your application. The concepts presented here can apply to software development in general, but they were geared toward specific OS/2 examples since OS/2 is the target environment. The steps can be tedious and take a little longer than you'd like, but the patience is worth it.

By having that patience and discipline to do it right the first time, rather than just doing what works quickly, you will reap the benefits of portability to PowerPC, the ease and efficiency with which you can maintain and enhance your code (providing benefits on many platforms by doing the work on only one) and the speed and all of the function offered by OS/2.

There is nothing wrong with prototyping functions from day one, but the real code should not be written until the basic design is complete. Stick with the fundamentals, the building blocks, and the function breakdown. Don't forget to document everything, including why decisions were made. Your design document should be not just a bunch of interface definitions but rather a working document that is more like a cookbook, with summaries of the alternatives considered and a list of the reasons for your choices. This way there will be no ambiguity later on when you develop, test, and fix the code.

I hope you have gained insight into not only how OS/2 works but how to design applications in general. This book is intended to be an "experience guide" for many of the decisions that you must make when developing OS/2 code. I'm sure you've noticed that there is very little code in this book. The reason is to allow you to concentrate on the design issues. Coding is a process that can be learned by looking at the programming technical references. Algorithms are the programmer's canvas, and the APIs are the paint. The design stage is where the critical decisions are made. Programming decisions are also important, but they are the implementation of the designs. By working from real-world examples you will be able to make maximum use of all that OS/2 has to offer.

Best of luck to you and sell millions of applications!!!!!!!!

David E. Reich

Bibliography

IBM Corporation, OS/2 2.0 Technical Library, Application Design Guide, 1992.

IBM Corporation, OS/2 Technical Library, Control Program Programming Reference, 1994.

IBM Corporation, OS/2 Technical Library, Control Program Programming Guide, 1994.

IBM Corporation, OS/2 Technical Library, Presentation Manager Programming Reference, Volume 1, 1994.

IBM Corporation, OS/2 Technical Library, Presentation Manager Programming Reference, Volume 2, 1994.

IBM Corporation, OS/2 Technical Library, Presentation Manager Programming Guide, Advanced Topics.

IBM Corporation, OS/2 Technical Library, Graphics Programming Interface Programming Guide, 1994.

IBM Corporation, OS/2 Technical Library, Graphics Programming Interface Programming Reference, 1994.

IBM Corporation, OS/2 Technical Library, Information Presentation Facility Programming Guide, 1994.

IBM Corporation, OS/2 Technical Library, System Object Model Guide and Reference, 1992.

IBM Corporation, OS/2 Technical Library, Workplace Shell Programming Reference, 1994.

Kogan, M. Dietel, "The Design of OS/2," 1992.

Reich, D., and Cheatham, Robinson, "OS/2 Presentation Manager Programming," January 1990.

Borgendale, K., and Bramnick, Holland, "Workplace OS: What is the OS/2 Personality?" 1994

Index

A

Agents, 179
Allocated pages, 67
API, 19–27, 30, 49, 67, 68, 71, 75, 76, 81, 94, 103, 108, 113, 114, 139, 158, 165, 166, 171, 192, 203, 208, 218, 252, 253, 266
API aliasing, 291, 292
Application defaults, 163
ASSOCTABLE, 175
Average character width, 277, 278
AVIO, 118, 119, 259

C

CASE, 164–166
Client area, 117, 158
Client Library, 106, 107
Clipboard, 5, 13, 92, 126, 249
Combo box, 168
Committed pages, 67, 68
Common services, 95, 103
Communications:
  background, 4
  data, 5, 6, 124
  interprocess, 5, 96, 111, 119, 172, 173, 177, 179, 200, 201, 213, 248
Context menu, 12
Control Program, 44
Controls, 159, 160
CORBA, 128, 170
Crash protection, 32
Critical section, 144, 246
CUA, 157, 157, 160, 268

D

Data validation, 231, 232
Deadlock, 243, 145
Debugging, 28, 29, 30, 190
Derivative classes, 304
Descriptor, 60–67
Device context, 16, 78–85
Device driver, 16, 23, 31, 43–53, 69–72, 81, 82, 104, 192
DevOpenDC, 264
Dialog window, 159–163
Direct-To-SOM, 171
DosAllocMem, 67, 160, 163, 260
DosCreateQueue, 248
DosCreateThread, 55, 58
DosDevIOCTL, 71, 194
DosEnterCritSec, 144, 246
DosExecPgm, 174, 180
DosExitCritSec, 144, 246
DosGetMessage, 294
DosGetProcAddr, 290, 292, 299

DosKillThread, 245
DosLoadModule, 290, 292, 299
DosQFileInfo, 24, 25
DosSetMem, 67, 68
DosSetPriority, 57–59
DT_QUERYEXTENT, 278
Dynamic Data Exchange, 5, 13, 92, 120, 124, 125, 249
Dynamic Link Library, 24, 26, 47, 51, 72–73, 106, 142, 143, 178, 181, 194, 197, 198, 291
  .LIB, 73

E

EVENT semaphore, 60
Event/Session Manager, 109, 153
Exception handler, 191, 237, 238
Extended attributes, 196, 197, 251, 254

F

File System Request Router, 68, 69
FileInfoBuf, 25
Fixups, 51, 73, 290

G

Global Descriptor Table, 62
GpiBox, 80, 82
GpiPlayMetafile, 23, 264
Graphics engine, 75–89
  function table, 81–83
Guard page, 67, 68, 237, 238

H

Help:
  contextual, 14, 268, 269
  hypergraphics, 268
  hypertext, 268
  IPFC, 268, 269
HPFS, 25, 196, 252
HyperWise, 268

I

INF files, 269
Inheritance, 177–178
INI files, 91, 195, 200
Initialization, 139
Input router, 86
Installable File System, 46
IOCTL, 71, 72, 134, 193
IPF, 14, 15, 267, 269, 270

K

KASE:PM VIP, 164–166
Kernel, 5, 29, 43, 44, 45, 47, 50, 187
Keyboard focus, 86, 88

L

LAN, 206, 252
LIBPATH, 291, 292
Linear Address, 63–67
Loader, 44, 46–48, 52, 67
Local Descriptor Table, 62
Locality of reference, 287–290

M

Memory compaction, 10
Memory leakage, 293
Memory management, 44, 45, 50, 60, 61–65, 233–237
  640 K limit, 8
  best-fit, 235
  first-fit, 235
  flat, 8, 9, 11, 64, 66
  linear, 64, 66, 68
  overcommitment, 47, 190
  segmented, 8, 9, 59–61

sparse allocation, 67, 68
virtual, 9, 63
Message queue, 88
Microkernel, 28, 95, 100, 101, 104, 108
Microkernel Messages, 103
Modality, 162, 163
Multiprocessing, 200, 201
Multitasking, 3, 4, 30, 48, 50, 140
Multithreading, 1, 3, 6, 20, 98, 133, 139, 151, 152, 218, 221, 222, 224, 228, 240, 247, 295
MUTEX semaphore, 60
MUXWAIT semaphore, 60
MVM, 95, 105

N

Named shared memory, 250
National Language Support, 273, 274

O

Object:
associations, 174
Workplace, 11, 13, 88, 89, 90–92
Object-oriented, 13, 14, 15, 90, 94, 99, 101, 143, 144, 148, 153, 191, 266
printing, 180
Object window, 223
OpenDoc, 28, 170, 171
OS/2 Server, 108, 109

P

Page frame, 63, 65
Page table, 63, 65
Page table directory, 63, 65
Paging, 9, 10, 11, 63, 67
Parallelism, 133
Path Length, 289
Personality Neutral, 102
Pipes, 5, 127, 249
PM_Q_RAW, 264
PM_Q_STD, 264
Portability, 22
Positioning of reference, 249–251
PowerPC, 18, 25, 26, 74, 95, 97, 105, 106, 108, 135, 137, 193, 228
Preemptive, 3, 48, 208
Presentation driver, 77–86, 136
brute force, 84, 85
Presentation Manager, 12, 47, 75–89, 93, 94–98, 107, 108, 116, 155, 187, 198
Presentation space, 16, 78–85, 96, 97, 136, 152
Print destination, 264
Protection violation, 56, 57

Q

Queues, 127, 248
QWL_USER, 305

R

Resource sharing, 143
REXX, 130, 170
Ring:
ring 0, 31, 54, 71, 192
ring 3, 31, 54
transition, 54, 192
RISC, 18

S

SAA, 156
Scaffold, 310
Scheduler, 7, 47–58, 69, 108, 243
Selector, 62
Semaphore, 50, 59, 60, 69, 88, 144, 242, 243, 247
Shared Services, 95, 103, 105
Smalltalk, 137
SMP, 2, 8, 17, 18, 25, 28, 48, 49
SMP-safe, 49

SOM, 14, 28, 90–92, 108, 159, 170, 171
SplEnumPrinter, 265
Spooler, 140
Standard window, 157, 160
Stringtable, 275
Subset function, 180
Swap file, 9, 60, 65, 217
Swapping, 9, 60, 65, 161
System testing, 41, 303, 306, 310

T

Templates, 121, 172, 173
Thread, 6, 7, 42–59, 71–72, 182
  context, 7, 57–59
  priority, 7, 47, 57–59, 295
  absolute, 57, 58
  classes, 57–59
  dynamic, 57–59
  foreground boost, 58

U

Unit testing, 41, 3102
UNIX, 22, 206

V

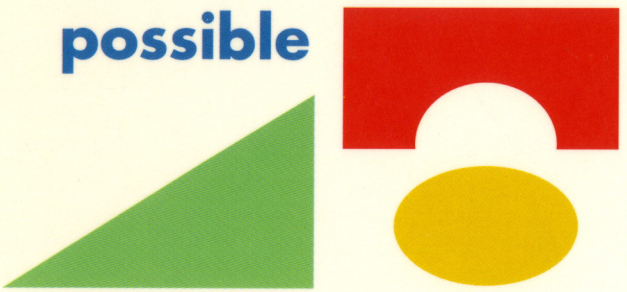
VIO, 115, 120, 259
virtual console, 4, 44
virtualization, 46

W

WC_BUTTON, 304
WC_ENTRYFIELD, 305
WC_LISTBOX, 305
WinCancelShutdown, 224, 241
WinCreateClass, 305
WinCreateHelpInstance, 269
WinCreateWindow, 23
WinDefWindowProc, 215
WinDispatchMsg, 218, 226
Window procedure, 215–218
WinDrawText, 278
WinFontDlg, 266
WinGetMsg, 88, 224, 226
WinInitialize, 215
WinPostMsg, 167, 219
WinPostQueueMsg, 226, 240, 247
WinQueryClassInfo, 305
WinSendDlgItemMsg, 167
WinSendMsg, 167, 219
WinSetWindowPos, 302
WinSetWindowText, 276
WinShowWindow, 302
WinWindowFromID, 167
WM_CHAR, 217
WM_COMMAND, 187, 188, 216, 217, 223
WM_CONTROL, 217
WM_PAINT, 216, 223
WM_QUIT, 219, 241
WM_SIZE, 304
WM_USER, 224–226
Working set, 287
Workplace Shell, 3, 12–15, 75, 78, 89–93, 96, 105, 106, 116, 121, 170, 172–176, 178, 201, 213, 294
Workplace Architecture, 26, 74, 95, 96, 147, 153, 200
WPAbstract, 91
WPDataFile, 92, 179
WPFileSystem, 91
WPObject, 91
wpOpen, 179
wpPrintObject, 180, 182, 183, 265
WPTransient, 91
WS_VISIBLE, 304
WYSIWYG, 114, 118, 131, 159, 268

$34.95

A total A-to-Z guide to designing the most powerful and efficient OS/2® applications possible



Do you know how to design and write applications to achieve maximum throughput using threads?

Do you know which interprocess communication tools are available to you and which ones to use when?

How do you coordinate execution among threads?

What about techniques to efficiently use and structure your application's use of memory?

Did you know that with the right design, you can write applications to run on single or symmetric multiprocessor Intel machines as well as PowerPC machines with a single set of source code?

What are the ways to tune your applications for maximum performance?

Get the answers to these and many other crucial design and programming questions in *Designing High-Powered OS/2 Warp Applications.*

Written by internationally renowned OS/2 guru David Reich, this book is a gold mine of insider tips and techniques for designing powerful, efficient applications that are easy to code, test, and maintain. Reich introduces you to *all* the features available in OS/2 Warp, tells you how they work and, with the help of numerous real-life examples and scenarios, shows you how to make optimal use of them. He covers the intricacies of:

- **The Workplace Shell and object-oriented programming in OS/2**
- **Client/server, multithreading, memory management, HELP facilities, and running Windows applications in OS/2**
- **Compilers, tree structures, and module structures that make your applications easier and less expensive to maintain**
- **CASE tools that help you structure your program and prototype functions quickly**
- **Developing versions of your code that work in languages other than English, using only one source code tree**

DAVID E. REICH is a Development Manager with IBM's OS/2 development team. He is also a regular columnist and contributing editor for *OS/2 Developer*. A well-known speaker at international OS/2 user groups and conferences, he has traveled the world, teaching classes and helping developers write high-quality OS/2 applications. His previous books include *Designing OS/2 Applications* and *OS/2 Presentation Manager Programming*, both available from Wiley.

Cover Design: Adrienne Weiss

John Wiley & Sons, Inc.
Professional, Reference and Trade Group
605 Third Avenue, New York, N.Y. 10158-0012
New York • Chichester • Brisbane • Toronto • Singapore

ISBN 0-471-11586-X