Towards More Usable Systems:

The LSRAD Report

Large Systems Requirements for Application Development

December 1979

SHARE Inc. Chicago, Illinois 60601

Copyright SHARE Inc., 1979. All Rights Reserved.

Published by SHARE Inc., a non-profit Delaware membership corporation. Permission to reprint in whole or in part may be granted for educational and scientific purposes upon written application to the Secretary, One Illinois Center, 111 E. Wacker Dr., Chicago, Illinois 60601. Permission is hereby granted to members of SHARE Inc. to reproduce this publication in whole or in part solely for internal distribution within the member's organization and provided the copyright notice printed above is set forth in full text on the title page of each item reproduced. Printed in the United States of America.

Preface

This is a report of the SHARE Large Systems Requirements for Application Development (LSRAD) task force. This report proposes an evolutionary plan for MVS and VM/370 that will lead to simpler, more efficient and more usable operating systems. The report is intended to address two audiences: the users of IBM's large operating systems and the developers of those systems.

The views expressed in this report represent those of the individuals participating in the task force rather than those of the sponsoring corporations.

The participants were:

Dana M. Becker Jay A. Jonekait Julie A. Lovinger Bruce S. Marshall Jay Michlin William Pashwa Marc Sewell Carol A. Schneier Richard W. Suko Albert B. Wilson, Jr. Bell Laboratories; Naperville, Ill.
Tymshare; Cupertino, Calif.
General Motors Research; Warren, Mich.
Perkin-Elmer Corp; Danbury, Conn.
Exxon Corp; Florham Park, N.J.
Bell Laboratories; Piscataway, N.J.
Boeing Computer Services; Seattle, Wash.
IBM, DSD; Poughkeepsie, N.Y.
Boeing Computer Services; Seattle, Wash.

Acknowledgements

The LSRAD task force would like to thank our respective employers for the constant support they have given us in the form of resources and encouragement. We further thank the individuals, both within and outside SHARE Inc., who reviewed the various drafts of this report. We would like to acknowledge the contribution of the technical editors, Ruth Ashman, Jeanine Figur, and Ruth Oldfield, and also of the clerical assistants, Jane Lovelette and Barbara Simpson.

Two computer systems proved invaluable for producing this report. Draft copies were edited on the Tymshare VM system. The final report was produced on the IBM Yorktown Heights experimental printer using the Yorktown Formatting Language under VM/CMS.



Contents

| Fragmentation The LSRAD Approach System Synergism Synergistic Computer Systems The Whole System A New Direction Programmer Productivity Benefits art II: Technical Proposals. Data in Memory Named Spaces Sharing in Virtual Memory Extended Addressing Device Independent I/O. Hierarchical Storage Manager Enhanced Programming Environment Integrated Command System. Data Access Control Subsystem Protection Mechanism Execution and Testing Environment Related Technical Material Evolution Versus Revolution. Compatibility and Migration Considerations. On Limits | Executive | Summary | ••• |
|--|------------|----------------------------------|-----|
| Task Force History History of Operating Systems The Problem Today Fragmentation The LSRAD Approach System Synergism Synergistic Computer Systems The Whole System A New Direction Programmer Productivity Benefits art II: Technical Proposals Data in Memory Named Spaces Sharing in Virtual Memory Extended Addressing Device Independent I/O Hierarchical Storage Manager Enhanced Programming Environment Integrated Command System Data Access Control Subsystem Protection Mechanism Excution and Testing Environment Related Technical Material Evolution Versus Revolution Compatibility and Migration Considerations. On Options On Limits. | Part I: Co | nceptual Framework | ••• |
| Task Force History History of Operating Systems The Problem Today Fragmentation The LSRAD Approach System Synergism Synergistic Computer Systems The Whole System A New Direction Programmer Productivity Benefits art II: Technical Proposals Data in Memory Named Spaces Sharing in Virtual Memory Extended Addressing Device Independent I/O Hierarchical Storage Manager Enhanced Programming Environment Integrated Command System Data Access Control Subsystem Protection Mechanism Excution and Testing Environment Related Technical Material Evolution Versus Revolution Compatibility and Migration Considerations. On Options On Limits. | | | |
| History of Operating Systems The Problem Today Fragmentation The LSRAD Approach System Synergism Synergistic Computer Systems The Whole System A New Direction Programmer Productivity Benefits art II: Technical Proposals Data in Memory Named Spaces Sharing in Virtual Memory Extended Addressing Device Independent I/O. Hierarchical Storage Manager Enhanced Programming Environment Integrated Command System Data Access Control Subsystem Protection Mechanism Execution and Testing Environment Related Technical Material Evolution Versus Revolution Compatibility and Migration Considerations. On Options On Limits | Defir | ing the Problem | |
| The Problem Today Fragmentation . The LSRAD Approach System Synergistic Computer Systems The Whole System A New Direction Programmer Productivity Benefits art II: Technical Proposals . Data in Memory Named Spaces . Sharing in Virtual Memory Extended Addressing Device Independent I/O. Hierarchical Storage Manager . Enhanced Programming Environment Integrated Command System Data Access Control Subsystem Protection Mechanism Execution and Testing Environment Related Technical Material Evolution Versus Revolution Compatibility and Migration Considerations On Options Data Migraphy | | | |
| Fragmentation The LSRAD Approach System Synergism Synergistic Computer Systems The Whole System A New Direction Programmer Productivity Benefits art II: Technical Proposals. Data in Memory Named Spaces Sharing in Virtual Memory Extended Addressing Device Independent I/O. Hierarchical Storage Manager Enhanced Programming Environment Integrated Command System. Data Access Control Subsystem Protection Mechanism Execution and Testing Environment Related Technical Material Evolution Versus Revolution. Compatibility and Migration Considerations. On Options. On Limits | | | |
| The LSRAD Approach System Synergism Synergistic Computer Systems The Whole System A New Direction Programmer Productivity Benefits art II: Technical Proposals. Data in Memory Named Spaces Sharing in Virtual Memory Extended Addressing Device Independent I/O. Hierarchical Storage Manager Enhanced Programming Environment Integrated Command System. Data Access Control Subsystem Protection Mechanism Execution and Testing Environment Related Technical Material Evolution Versus Revolution. Compatibility and Migration Considerations. On Options. On Limits | | | |
| System Synergistic Synergistic Computer Systems The Whole System A New Direction Programmer Productivity Benefits art II: Technical Proposals. Data in Memory Named Spaces Sharing in Virtual Memory Extended Addressing Device Independent I/O. Hierarchical Storage Manager Enhanced Programming Environment Integrated Command System. Data Access Control Subsystem Protection Mechanism Execution and Testing Environment Related Technical Material Evolution Versus Revolution. Compatibility and Migration Considerations. On Options On Limits | • | ragmentation | • |
| System Synergistic Synergistic Computer Systems The Whole System A New Direction Programmer Productivity Benefits art II: Technical Proposals. Data in Memory Named Spaces Sharing in Virtual Memory Extended Addressing Device Independent I/O. Hierarchical Storage Manager Enhanced Programming Environment Integrated Command System. Data Access Control Subsystem Protection Mechanism Execution and Testing Environment Related Technical Material Evolution Versus Revolution. Compatibility and Migration Considerations. On Options On Limits | The l | SRAD Approach | |
| Synergistic Computer Systems The Whole System A New Direction Programmer Productivity Benefits art II: Technical Proposals. Data in Memory Named Spaces Sharing in Virtual Memory Extended Addressing Device Independent I/O. Hierarchical Storage Manager Enhanced Programming Environment Integrated Command System. Data Access Control Subsystem Protection Mechanism Execution and Testing Environment Related Technical Material Evolution Versus Revolution. Compatibility and Migration Considerations. On Options On Limits | | | |
| A New Direction Programmer Productivity Benefits art II: Technical Proposals. Data in Memory Named Spaces Sharing in Virtual Memory Extended Addressing . Device Independent I/O. Hierarchical Storage Manager Enhanced Programming Environment Integrated Command System. Data Access Control Subsystem Protection Mechanism Execution and Testing Environment Related Technical Material Evolution Versus Revolution. Compatibility and Migration Considerations. On Options. On Limits . | | | |
| A New Direction Programmer Productivity Benefits art II: Technical Proposals. Data in Memory Named Spaces Sharing in Virtual Memory Extended Addressing . Device Independent I/O. Hierarchical Storage Manager Enhanced Programming Environment Integrated Command System. Data Access Control Subsystem Protection Mechanism Execution and Testing Environment Related Technical Material Evolution Versus Revolution. Compatibility and Migration Considerations. On Options. On Limits . | The V | Whole System | |
| Programmer Productivity Benefits art II: Technical Proposals. Data in Memory Named Spaces Sharing in Virtual Memory Extended Addressing Device Independent I/O. Hierarchical Storage Manager Enhanced Programming Environment Integrated Command System Data Access Control Subsystem Protection Mechanism Execution and Testing Environment Related Technical Material Evolution Versus Revolution. Compatibility and Migration Considerations. On Options On Limits | | | |
| art II: Technical Proposals. Data in Memory Named Spaces Sharing in Virtual Memory Extended Addressing Device Independent I/O. Hierarchical Storage Manager Enhanced Programming Environment Integrated Command System Data Access Control Subsystem Protection Mechanism Execution and Testing Environment Related Technical Material Evolution Versus Revolution. Compatibility and Migration Considerations. On Options. On Limits | | | |
| Data in Memory Named Spaces Sharing in Virtual Memory Extended Addressing Device Independent I/O Hierarchical Storage Manager Enhanced Programming Environment Integrated Command System Data Access Control Subsystem Protection Mechanism Execution and Testing Environment Related Technical Material Evolution Versus Revolution Compatibility and Migration Considerations On Options On Limits | | Togrammer Floductivity Benefits | • • |
| Named Spaces Sharing in Virtual Memory Extended Addressing Device Independent I/O. Hierarchical Storage Manager Enhanced Programming Environment Integrated Command System. Data Access Control Subsystem Protection Mechanism Execution and Testing Environment Related Technical Material Evolution Versus Revolution. Compatibility and Migration Considerations. On Options. On Limits | rt II: To | chnical Proposals | • |
| Named Spaces Sharing in Virtual Memory Extended Addressing Device Independent I/O. Hierarchical Storage Manager Enhanced Programming Environment Integrated Command System. Data Access Control Subsystem Protection Mechanism Execution and Testing Environment Related Technical Material Evolution Versus Revolution. Compatibility and Migration Considerations. On Options. On Limits | Data | n Memory | , |
| Sharing in Virtual Memory Extended Addressing Device Independent I/O. Hierarchical Storage Manager Enhanced Programming Environment Integrated Command System. Data Access Control Subsystem Protection Mechanism Execution and Testing Environment Related Technical Material Evolution Versus Revolution. Compatibility and Migration Considerations. On Options On Limits | | • | |
| Extended Addressing . Device Independent I/O. Hierarchical Storage Manager Enhanced Programming Environment Integrated Command System. Data Access Control Subsystem Protection Mechanism Execution and Testing Environment Related Technical Material Evolution Versus Revolution. Compatibility and Migration Considerations. On Options On Limits | | | |
| Device Independent I/O. Hierarchical Storage Manager Enhanced Programming Environment Integrated Command System. Data Access Control Subsystem Protection Mechanism Execution and Testing Environment Related Technical Material Evolution Versus Revolution. Compatibility and Migration Considerations. On Options. On Limits. | | | |
| Hierarchical Storage Manager Enhanced Programming Environment Integrated Command System. Data Access Control Subsystem Protection Mechanism Execution and Testing Environment Related Technical Material Evolution Versus Revolution. Compatibility and Migration Considerations. On Options. On Limits | | | |
| Enhanced Programming Environment Integrated Command System. Data Access Control Subsystem Protection Mechanism Execution and Testing Environment Related Technical Material Evolution Versus Revolution. Compatibility and Migration Considerations. On Options On Limits | | | |
| Integrated Command System. Data Access Control Subsystem Protection Mechanism Execution and Testing Environment Related Technical Material Evolution Versus Revolution. Compatibility and Migration Considerations. On Options. On Limits. | | | |
| Data Access Control Subsystem Protection Mechanism Subsystem Protection Mechanism Execution and Testing Environment Related Technical Material Evolution Versus Revolution. Compatibility and Migration Considerations. On Options On Options On Limits | Enha | ced Programming Environment | 4 |
| Subsystem Protection Mechanism Execution and Testing Environment Related Technical Material Evolution Versus Revolution. Compatibility and Migration Considerations. On Options. On Limits | J | ntegrated Command System | . (|
| Execution and Testing Environment |] | Jata Access Control | . 6 |
| Related Technical Material Evolution Versus Revolution. Compatibility and Migration Considerations. On Options. On Limits bliography. | | ubsystem Protection Mechanism | . (|
| Evolution Versus Revolution. Compatibility and Migration Considerations. On Options. On Limits. bliography. On Contemport | 1 | xecution and Testing Environment | . ' |
| Evolution Versus Revolution. Compatibility and Migration Considerations. On Options. On Limits. bliography. On Contemport | Relat | ed Technical Material | |
| Compatibility and Migration Considerations. | | | |
| On Options | | | |
| On Limits | | | |
| bliography | | | |
| | | ni Linnis | |
| nondiv: Survey Results | bliograpl | y | . 9 |
| | mondi | Survey Desults | |



EXECUTIVE SUMMARY

The Large Systems Requirements for Application Development (LSRAD) task force was organized by the IBM users' group, SHARE, Inc., to:

1

- 1. Identify the problems that confront the users of large IBM computer systems
- 2. Propose evolutionary operating system enhancements to improve the usability of both MVS and VM/370

The task force findings and recommendations are summarized in this Executive Summary. The summary makes no attempt to ustify the findings or recommendations or to present detailed technical proposals. The deta is are covered in the body of the report.

The impetus behind the formation of the task force was the realization that the approaching bottleneck in the path of expanding data processing usage is programmer productivity. There is a critical shortage of qualified programmers to address the growing applications backlog in most companies today. Therefore, the task force was directed to propose a way to make these programmers more productive.

THE PROBLEM

MVS and VM/370 were developed as a collection of subsystems, access methods, and numerous programming tools. Each of these facilities was designed to meet a specific set of needs. This segmented approach has led to an increasing set of complex and inconsistent interfaces which the programmers must understand to use the systems. Consequently, MVS and VM/370 users are encountering an environment which can best be described as *fragmented*.

Due to the early origins of the operating systems, programmers must understand a great deal about the computer hardware and software in order to move their data from external storage devices to internal storage, manipulate it, and then return it to the external devices for permanent storage. It is the management of internal (virtual) storage and external (file) storage in the current systems that shapes the programming environment. The early origins are also reflected in the complex and inconsistent utility and command languages provided today. Major problems face the data processing industry:

- 1. Operating systems have become increasingly complex and difficult to use.
- 2. Programmers are encountering significant conversion problems for current applications with each new hardware device or release of an operating system.
- 3. Software maintenance costs are taking an ever-increasing portion of the data processing budget.
- 4. Programmer training and retraining time and costs are continuing to rise.

APPROACH

The task force decided that the best place to introduce staged, evolutionary change which benefits the most users is in the foundation of the operating systems. Historically, most changes have been made as additions to the systems, such as in the languages or in the subsystems. This has led to fragmentation. A benefit provided by a change to one subsystem or language is not benificial to the rest of the system. The LSRAD task force is suggesting change at the foundation of the operating systems, where the leverage is greatest. Whatever is done in the operating systems can benefit all subsystems and all languages while being integrated into a consistent programming environment.

The task force recognizes the need for evolutionary change, rather than revolutionary change, and the operating system is the logical place for implementation. The operating system can, and must, continue to support the existing environment while changes are introduced. The changes that are made to the systems must be viewed as a foundation for the future. By adding a few basic building blocks to the operating systems, the user's interface to the system can be simplified. The user will not require knowledge of the internal computer structures and will thus find the computer easier to use. Knowledge of the computer system's internal structure is today an expensive overhead requirement of the application development process. The goal is to eliminate this overhead.

The task force has developed a set of principles for building enhanced computer systems which will substantially reduce the fragmentation that is common today. This set of principles should be used in evaluating any future systems or application program enhancements. The principles will aid in producing synergistic computer systems which will be more powerful in function, but simpler and easier to use.

The task force then applied these principles to the current systems, MVS and VM/370. A set of technical proposals were developed for evolving these systems toward an enhanced programming environment. The proposals fall into two general areas. The first set deals with functions for allowing programmers to effectively store and manipulate data in memory. The second set probes ways in which the programming environment can be simplified and enhanced if the first set is implemented. The proposals also provide a base for developing new application development and testing tools which are not feasible today.

TASK FORCE PROPOSALS

The task force has proposed the inclusion of the following primitive functions in the MVS and VM/370 operating systems:

- 1. Data in Memory
 - a. Named Spaces. Chunks of virtual memory would be "carved" out of a user's address space and assigned names. They would be owned, be shared explicitly, and be secured to prevent unauthorized access.
 - b. Sharing in Virtual Memory. The system should provide a convenient mechanism to allow applications to dynamically share programs and data.
 - c. Extended Addressing. Since 16 megabytes is not sufficient virtual space for many applications, addresses should be extended to the next logical size, using the fullword.
 - d. Device Independent I/O. The user should deal with the logical attributes of programs and data rather than the physical aspects of its storage.
 - e. Hierarchical Storage Manager (HSM). HSM should manage the interface between the system and the external devices. HSM is a single physical I/O manager for all components of the system, such as named spaces, files, and data base management systems.
- 2. Enhanced Programming Environment
 - a. Integrated Command System. The simplicity of device independent I/O would be reflected in a command language that would be consistent through all subsystems.
 - b. Data Access Control. Access control would be integrated into the system, with the key concepts of ownership, security, integrity, auditability, and dynamic sharing.
 - c. Subsystem Protection Mechanism. The system should guarantee that individual applications will not interfere with each other.
 - d. Execution and Testing Environment. This environment would allow the development of tools to aid development teams in the testing and certification of their applications.

BENEFITS

The benefits of these proposals are enormous, extending over a large base of users. The usability that is achieved by improving the operating systems benefits all users. The benefits affect three areas:

1. The task force proposals will simplify the application development process. When the new techniques have replaced the old, staffing requirements for application development will be reduced by up to 35%.

- 2. A simpler, consistent system will have powerful synergistic effects on programmers, significantly enhancing their productivity.
- 3. The application programmers will build tools to enable more noncomputer professionals to use computers effectively in their daily work.

A benefit of restructuring the operating system is the improvement in application development. The task force researched the application development process. Available literature and studies within our companies indicated that the application development cycle could be reduced by up to 35%. This means that companies could apply this saving to reduce their application backlogs.

These enhancements interact in a synergistic way to provide even greater productivity gains. The technical proposals build on each other. When looked at separately, each proposal provides tools which will aid the application developer to some extent. Yet when implemented together, the the proposals can be used to create powerful, yet conceptually simple systems. Such systems will provide a foundation for tools that benefit the application developer and the nondata processing professional even more. If the system makes sense, developers will often be able to guess the right way to communicate with the system to accomplish a given task. They will not have to consult innumerable manuals and attend months of classes to become experts in dozens of inconsistent environments. The education savings alone can total hundreds of millions of dollars a year, and programmers can spend that time productively solving problems.

Furthermore, the LSRAD proposals offer a strong foundation for the future. Over the next several years it will be possible to produce new development and testing tools and methodologies based upon this simpler, consistent foundation. Development of these tools is not cost effective today because of the fragmentation of the current computer systems. With this enhanced system as a base, a usable computing system can be offered to the secretaries, chemists, and aerodynamicists in business enterprises. Noncomputer professionals will be able to use the computer as is required for their jobs. In most corporations noncomputer people far outnumber computer people.

SUPPORT

The task force has presented these ideas to both SHARE and GUIDE. The audiences at these presentations were asked to complete a survey form at the conclusion of the presentation. The results show overwhelmingly positive support from a broad base of IBM customers. The response from a significant cross-section of large installations was nearly unanimous. Regardless of the businesses represented, usable computer systems are required to solve complex business problems.

More usable systems can increase the number of people who can use computers effectively and at the same time can make each current user more productive. Companies will then apply more data processing resources to address their ever increasing application backlogs. IBM can supply more data processing equipment and customers will use it more profitably.

FUTURE DIRECTION

In response to suggestions from the LSRAD task force, the future activities of SHARE will parallel the changing programming environment. Additional areas are under study. The task force has proposed only the foundation, the solid base for a new direction. There are still many areas that SHARE projects must address:

- Interactive subsystems
- Higher level languages
- Human factors
- Installation management
- Distributed processing

As SHARE begins to address more middle- to long-range requirements, individual projects will screen current short-range requirements for consistency with the future direction. If short-range requirements are counterstrategic, then the inconsistencies will be reviewed and their priorities revised.

PART I: CONCEPTUAL FRAMEWORK

The task force was created to examine the application development process and to recommend future directions for IBM's large operating systems, MVS and VM/370. Part I examines the state of application development, the problems it is encountering, an approach to solving those problems, and the benefits that will be derived from their solution.

Part I is divided into the following chapters:

- Defining the Problem
- The LSRAD Approach
- The Whole System

DEFINING THE PROBLEM

This chapter presents a history of the task force and of operating system development. Following that, problems facing the data processing industry are discussed, focusing particularly on fragmentation.

The first section summarizes the organization and activities of the task force.

The second section puts the task force findings into a historical perspective. It shows that data processing problems are a result of the rapid growth of the data processing industry.

The third section describes the problems perceived by the task force. The task force examined many immediate application development problems and found that some basic operating system deficiencies are to blame. Those deficiencies are described in this section.

The last section discusses fragmentation, which the task force felt was the major problem facing application developers.

Task Force History

In March, 1978, a task force was formed in the Basic Systems Division of SHARE to study productivity, a major concern of the computer industry.¹ Due to a critical shortage of qualified programmers,² most companies are experiencing expanding application backlogs.³ Coupled with the programmer shortage is a dramatic reduction in hardware costs, which allows more applications to be economically justified.⁴ The task force was asked to analyze the productivity problems that confront the users of large computer systems and to propose operating system improvements for both MVS and VM/370. The group was called the Large Systems Requirements for Application Development (LSRAD) task force.

The task force consisted of eight SHARE members, from Bell Telephone Laboratories, Boeing Computer Services, Exxon, General Motors Research, Perkin-Elmer, and Tymshare. SHARE project managers were asked to volunteer their key people for one year. Two IBM representatives were assigned to aid the task force in developing reasonable proposals. One of the strengths of the task force was the diversity of backgrounds. The members came from a variety of data processing environments and represented many different systems: MVS, SVS, TSS, and VM/370. As a result, the best ideas from each system could be drawn upon. It was discovered during discussions within the group that many ideas about improving the operating systems were not new at all. In fact, many of the ideas are implemented in one form or another in current IBM products.

TASK FORCE CHARTER

The following guidelines were established for the task force:

- 1. To propose evolutionary directions for MVS and VM/370. Upward compatibility is imperative because users have so much invested in applications that run on current IBM systems. It would not be economical to discard current applications and redevelop them on a totally new system, even if that system were more usable.
- 2. To limit the proposals to those that could be implemented in the near future. The length of the IBM development cycle was kept in mind while selecting a mid-range time frame of 1980-1985.
- 3. To examine the programming environment from a user's perspective, rather than from a system programmer's perspective.
- 4. To present the proposals to the SHARE community and determine if the suggested direction indeed meets the needs of the users.
- 5. To organize the SHARE response to these proposals and present it to IBM as a SHARE requirement.

The task force's initial meetings were devoted to developing technical proposals. Later, two working sessions were held with IBM designers and developers from the General Products Division and the Data Systems Division to present the task force's ideas and to provide a forum for discussion.

The task force also presented the proposals to an audience of about 650 people at SHARE 51 (August 1978) and to an audience of about 100 at GUIDE 47 (November 1978). Each audience was asked to complete a survey form at the conclusion of the presentation. The

Task Force History

task force considered feedback essential to establish credibility and to indicate to IBM that the proposals represent a significant segment of their customers' opinion. A positive response was received from both groups as described in the Appendix: Survey Results. The following summarizes the response:

Questionnaire Responses

| Addressing Relevant Problems | 97% |
|--|------------|
| Sound Technical Approach | 94% |
| Productivity Would Improve | 97% |
| • Easier for End Users | 95% |
| Allow New Applications Development | 81% |

Of 350 responses representing 270 unique installations, 97% of the respondents thought that the proposals addressed relevant issues. Over 90% of the respondents agreed that the task force was taking a sound technical approach, that implementation of the proposals would improve application developers' productivity and that end users would benefit significantly. Four out of five respondents agreed that the proposals would permit development of new applications which today are viewed as economically unfeasible.

9

Figure 1

Response to Technical Proposals



Figure 2

The respondents also supported the technical proposals, as shown in the summary data in Figure 2. The white represents those who agreed with the specific proposal, the black represents those who disagreed, and the grey indicates a neutral position. The respondents strongly agreed with all of the technical proposals, and were particularly in favor of device independent I/O and command language improvements.

In all cases, the positive responses outnumbered the negative ones. This does not indicate that the task force proposals are the best solutions, but it does indicate that the task force is considering pressing concerns in the data processing industry. IBM customers are demanding operating system improvements which parallel these proposals. The demand is coming from a broad spectrum of users from many different industries, including finance, insurance, public utilities, manufacturing, communications, publishing, chemical, petroleum, and local and federal governments. The demand is not by a margin of merely two or three to one. Over 90% of IBM's customers who responded to the questionnaire demand improved computing systems. The message from a significant cross-section of large installations is clear. The respondents' experience at their own installations has indicated the need for simpler, more usable computer systems.

FUTURE DIRECTIONS

The future activities of SHARE will have an impact on the effectiveness of the proposed foundation. The following topics and others must be considered to determine how they affect the entire programming environment, rather than just the subsystem in which they are implemented.

Interactive subsystems require significant work. In particular, the human interface and productivity impact of both tine sharing command systems and database systems must be addressed in detail. Benefits of the enhancements to the underlying system must be made available to users through the interactive subsystems.

High level languages must be investigated. Languages are the slowest area of data processing to change. A way must be found for the COBOL of 1990 to use the improved operating systems of that era. Requirements for productivity aids should be enumerated. Will a language that is suitable for interactive debugging make a significant difference in productivity? Can the need for the linkage editor be eliminated? These questions should be addressed now.

Human factors and productivity measurements should be investigated. The task force found many references to productivity, usability, and human interaction, but little concrete data on the subject.

The *management* of large aggregates of computing power, such as three 3033s or even ten 3033s is another area requiring investigation. It is necessary to consider the organizations, each with unique requirements, to support that amount of computing power. Many data centers are presently confronted with these decisions.

Distributed processing is expanding rapidly. The use of computers in many companies is growing. Telecommunications facilities are becoming more efficient, so that computers will continue to spread physically throughout companies. This subject requires immediate planning.

This list is not intended to be all-inclusive. Both SHARE and IBM must initiate investigations into new areas when the course of the data processing industry so dictates.

REFERENCES

- 1. T.A.Dolotta, M.I.Bernstein, R.S.Dickson Jr., N.A.France, B.A.Rosenblatt, D.M.Smith, and T.E.Steel, *Data Processing in 1980-1985*, New York, 1976.
- 2. Dale Rhodabarger, "Trends for 1979: A Survey of the Future," Computer Decisions, January 1979, p.52.
- 3. Jake Kirchner, "IBMer Sees DP Technology Booming Again," *Computerworld*, February 19, 1979, p.7.
- 4. Edith Holmes, "Technology or Applications More Crucial to Growth?" Computerworld, September 19, 1977, p.1.

History of Operating Systems

The problems that application developers encounter when dealing with the two large scale operating systems, MVS and VM/370, are a direct result of how those systems were developed. For that reason, any discussion of present problems tends to lose perspective unless one considers the industry's history and how it arrived at its present state.

FIRST AND SECOND GENERATION HARDWARE

During the late 1950s and early 1960s, many large computer centers were running basic operating systems, and some of these were designed by the individual centers. These systems had few installation-provided facilities. The facilities were limited mainly to tape or disk input and output, offline spooling procedures, and some limited error recovery. There were few facilities for job accounting, disk space allocation, or resource controls. There were no provisions for multiprogramming or multiprocessing. These systems were slightly more advanced from the "hands on" programming of the early 1950s.

THIRD GENERATION HARDWARE

When IBM introduced the System/360 hardware in 1964, an incompatible operating system was introduced, designed to improve the system throughput and p ovide many new facilities for programmers. The operating system, OS/360, included spooling facilities, an attempt at device independent I/O, multiprogramming, disk space allocation, and a number of internal programs to provide system services for user applications. PL/I, a new language, was also introduced to provide one standard language for programmers. The use of one language was expected to increase productivity.

OS/360 was widely accepted in the data processing world. and companies installed machnines as fast as they could be delivered. However, it was difficult to convert to the System/360 machines from the 7000 series. From 1965 through 1969, both IBM and its customers were confronted with overdue software commitments, project failures due to overestimating the capabilities of the new machines, and over-run budgets.¹ Eventually the OS/360 operating system stabilized at its original design specifications, installations finished their conversion efforts, and new projects were started.

Another model of the System/360, the Model 67, was developed with dynamic address translation. This facility enabled development of a virtual memory operating system. IBM introduced the Time Sharing System (TSS) using the virtual memory capability to implement a time sharing, interactive environment for the programmer. In addition, a group of IBM developers at the Cambridge Scientific Center near Boston experimented with the concept of "virtual machines" where each user of a time sharing computer system appears to be the only user of a particular operating system. This development later became known as the CP/67 operating system and was in use by a small number of customers in late 1969 and the early 1970s.

The System/370 series was introduced in early 1970, but it provided only a new price/performance ratio and few new capabilities. After the difficulties with introducing OS/360, IBM was not prepared to introduce revolutionary changes to the operating system.

Two more years passed before any other hardware changes were introduced. In the fall of 1972, virtual memory was reintroduced along with improved System/370 machines possessing dynamic address translation. To take advantage of this facility, new versions of three IBM operating systems, MFT, MVT, and CP/67, were developed. These systems, called OS/VS1, OS/VS2, and VM/370 respectively, supported virtual memory.

VIRTUAL MEMORY BACKGROUND

The introduction of virtual memory was heralded by IBM and by many users as a significant advance in operating system technology. Virtual memory increased the total amount of available addressing space, thus reducing restrictions on program size. Virtual memory programming also eliminated the need for complex program overlays and movement of data. However, this capability was not accepted immediately. There was strong resistance to the virtual concept both from computer professionals who did not understand it, and from managers concerned about performance and conversion costs.

The need to develop new applications quickly and cheaply, combined with the developer's awareness of the values of virtual memory, led to its increased acceptance and use. Even though virtual memory has been available for over a decade, it is only within the last few years that the virtual memory style of solving application problems has become widespread.² Problem solvers are effectively using virtual memory to develop applications which were not practical before.

THE LESSONS OF HISTORY

IBM has always considered itself an innovator in computer technology. System/360 with OS/360 offered customers new and extended features. OS/360 was designed to meet many of the same goals of automation and human productivity that LSRAD is requesting. Unfortunately, OS/360 was "ahead of its time." It sought to automate many computer operations when computers were not powerful enough to support such automation. At that time, experience and knowledge of software design was limited.

The System/360 - OS/360 innovation has been judged a success today, although in the 1960s it was a wrenching experience. Presently, industry is more cautious, and change is implemented more gradually. Massive innovation has become unthinkable, both for IBM and its customers. Each new concept or facility must fit with the old, so the transition is gradual.

Present computer systems are superior to those of the 1960s, and knowledge of software design and of human interaction with the computer is much improved. Due to the growing complexity of business problems, industry needs improved computer systems. The technology exists to develop new systems, but industry is not prepared to accept radical changes.

As a result of previous unsuccessful endeavors, the industry is struggling to determine the direction for the future. The task force believes that both computer manufacturers and developers must clearly perceive the present state of computer technology before they can plan for the future. The following section discusses several problems from the perspective of the application developer.

History of Operating Systems

REFERENCES

Ŷ

- 1. Frederick P. Brooks, Jr., The Mythical Man Month, Reading, Massachusetts, 1975.
- 2. J.H. Morrissey and L.S.-Y. Wu, "Software Engineering... An Economic Perspective," *Proceedings, 4th International Conference on Software Engineering, Munich, September* 1979.

The Problem Today

The intent of the LSRAD task force has been to address problems in MVS and VM/370 that impede the application development process. It has been amply documented elsewhere that application development must be improved.¹ Due to the lack of timely data processing support, business needs are either delayed or unmet.

Business needs for computing increased rapidly in the 1970s and will continue to increase in the 1980s. Additional data processing needs originate from government mandates for efficiency, safety, or environmental protection, from the increasingly worldwide scope of operations and of competition, or from the explosive growth in litigation directed against corporations.

Data processing hardware has kept pace with business needs in the past decade. Unfortunately, application development productivity has not kept pace either with business needs or with computers. While orporations quadrupled their computing loads, application development productivity barely cloubled.¹ Moreover, the cost of application developers increased at a rate even more rapid than that of inflation.

In addition to the mismatch in growth rates, the economics of the data processing industry are different today. Current systems are all derivatives of the systems developed in the early 1960s. At that time, hardware was relatively expensive, and people costs were relatively cheap. Programming staffs were small, and people worked with the minute details of the hardware. In that environment it made economic sense to optimize each program to extract maximum performance from the hardware.

In contrast, current hardware costs have dropped by orders of magnitude while programmer costs have risen continually.² It is no longer reasonable to bind each program to the hardware, given current tradeoffs.

Two types of programs are produced. The first is written to solve a specific problem and is discarded once the problem is solved. In this case, the system should be optimized to reduce the total development time and resources, which includes training and program development as well as computer resources. The second type of program is designed to run (perhaps with enhancements) for many years on both the current and unspecified future hardware. In neither case is it desirable to bind the application to specific details of the hardware. Yet, existing systems force application developers to do this.

The underlying requirement today is to produce applications fast enough to meet business needs. There is a de facto requirement to increase the productivity of individual application developers because the alternative of hiring more application developers is not possible.³ Data processing budgets are limited and cannot absorb the salaries of a large number of professionals. Also, computing professionals are in short supply.

The task force suggests the following as solutions to increasing productivity:

- 1. Automate. In most industries, automation means replacing the human work force with machines. In data processing, automation means enlisting the machine to absorb some of the developer's tasks. In either case, automation makes a process less labor-intensive and increases the output per worker.
- 2. Reduce fragmentation and confusion. Sometimes there are several parts of a process with overlapping or conflicting domains. In such a situation too many people have an excess of

The Problem Today

responsibilities. The best solution to this problem is reorganization, and this applies to a data processing system as well. Reorganization is a slow process. Goals and plans must be set on a long-range basis. The goals usually include streamlining interactions, reducing conflicts, and improving communication paths.

3. Eliminate unnecessary middlemen. Telephone companies improved productivity by making it possible for customers to complete calls without operator assistance.

The LSRAD task force carefully examined MVS and VM/370 and found that each of the above strategies for improving productivity is appropriate. All of them used together focus on most of the problems in the two operating systems, which include the following:

- 1. MVS and VM/370 require that users specify much detail, particularly with reference to I/O and memory management. Automated facilities could minimize this task.
- 2. In MVS and VM/370 there are twelve or more distinct ways to deal with data, consisting of a large number of overlapping and conflicting software facilities that generally do not support one another. Reorganization and simplification would ease the jobs of end users, application developers, system programmers, and operating system designers.
- 3. If MVS or VM/370 were simplified, end users could approach the systems without assistance from computer professionals.

Application development is hampered by the present operating system complexity, fragmentation, and lack of automation. These phenomena interfere with the entire application development process from initial development through testing, release, and maintenance. Application developers cannot solve the problems of the 1980s using tools developed in the 1960s. Removing the impediments will improve the entire application development process.

REFERENCES

- 1. T.A.Dolotta, M.I.Bernstein, R.S.Dickson Jr., N.A.France, B.A.Rosenblatt, D.M.Smith, and T.E.Steel, *Data Processing in 1980-1985*, New York, 1976.
- 2. Bureau of Labor Statistics, "Computer and Related Occupations," Occupational Outlook Handbook, 1978-79 Edition, Bulletin 1955, Washington, D.C., U.S. Government Printing Office, 1978, pp. 111-117.
- 3. Phillip S. Nyborg, Pender M. McCarter, and William Erickson, eds., Information Processing in the United States, a Quantitative Summary, Montvale, New Jersey, 1977.

Fragmentation

The task force considers fragmentation to be the application developer's main problem with MVS or VM/370.

Fragmented Programming Environment



Figure 1 shows how computer software is structured. It shows the application software resting on subsystems and access methods, which in turn, rest on the processor control program, i.e. the operating system.

Figure 1

Each block contains a wide variety of choices, with a seemingly endless list of acronyms. IMS, CICS, DMS, ADF, FORTRAN, COBOL, PL/I, BASIC, VSPC, APL, TSO, and CMS are only some of the products that fall within the boxes labelled DB/DC, High Level Lanuguages, and Interactive Programming. The lists are equally as long for the remainder of the software structure.

Each of these products was developed independently of most of the others. Each product is useful, but together they present an environment which is complex and inconsistent, or fragmented. A productivity improvement within one high level language, one data base management system or one interactive program does not provide that productivity gain to other users of the operating system. Productivity aids such as the Structured Programming Facility (SPF) could be added to each subsystem, but while that may be useful to some users, these aids would continue to fragment the environment. Keywords change meanings when one switches between subsystems. Parsing algorithms vary. All of this confuses the developer, thereby retarding productivity.

John Ehrman's presentation at SHARE 53 illustrated the fragmented programming environment and how it hampers productivity.¹ A programmer must know at least twelve distinct languages to produce even a simple program. The languages are incompatible, have syntaxes that contain idiosyncracies, and have mediocre diagnostics. The languages are often the biggest obstacles to rapid programming development.

If PL/I is chosen to develop the "simple" program, the programmer must know three distinct languages to get started. The first is the language of statement flow and process sequencing, i.e., the logical organization of the code. The second language is a formatting and conversion language. This specifies how the program's internal data must be rearranged and converted before it can be put into an external file. The formatting and conversion language is also used where data is to be read into the program. The third language involves the mapping of data into virtual memory from external storage. It specifies the internal data types and structures to be manipulated by the processing logic of the program.

Writing a program in PL/I requires knowledge of an algorithmic language that manipulates the internal data in a prescribed fashion. In addition, the programmer must know the

Fragmentation

external format and representation of all data elements and a special format conversion language that transforms the data between its internal and external structures.

A job control language (JCL) is required to inform the operating system what must be done to test the program. Learning to use and code JCL statements correctly requires a substantial investment of time and effort.

The Linkage Editor control language is required to place the program into a test library from which it can be run with some sample data. Linkage Editor language is complex and often requires the assistance of a system expert.

Program debugging usually requires the knowledge of three additional languages. The first and second languages, absolute binary machine and symbolic assembler language, are related. The symbolic assembler-like listing produced by the compiler must be correlated with the hexadecimal machine language code and data in a virtual memory dump. The third language includes the syntax and semantics of an interactive debugging system which allows the developer to trace the execution of the program on a statement-by-statement basis.

A variety of utility programs is needed to set up different test versions of the program and data (e.g., IEHLIST, AMBLIST, IEBCOPY). Each of these utilities has its own particular control statement syntax and format, which are not user-oriented. Similarly, the Virtual Storage Access Method Services language is oriented toward ease of computer scanning, not ease of use. Learning the functions of these programs, the control statements, and the JCL needed to obtain the desired results requires another substantial investment of time and effort. The "utilities language" is a grouping of many distinct and often dissimilar languages.

The text editor is an important program development tool. It is needed to manipulate the "source" (character) form of all other objects. It is fundamental to all programming tasks and should be the easiest to use, but it often is not. Most programmers adapt to the peculiarities of its command syntax or operand names because the text editor must be used so frequently.

Present operating systems provide a simple command language procedure capability that allows users to collect and combine commonly used command strings into a single grouping. Most system command languages are difficult to use without a command procedure language to control and sequence a set of programs. Examples are JCL catalogued procedures, TSO CLISTS, and the CMS EXEC facility.

A text-formatting language must be learned to produce computer-formatted documentation. Documenting the program's usage, marketing and maintenance requires English language skills.

SUMMARY OF LANGUAGES

To summarize, development of even the simplest applications requires knowing the following artificial languages:

- 1. Algorithmic statement flow
- 2. External data description and conversions
- 3. Internal data typing and structuring
- 4. Job control language
- 5. Linkage editor control

- 6. Absolute binary machine language
- 7. Symbolic assembler language
- 8. Debug and diagnostic system syntax
- 9. Utilities
- 10. Text editor
- 11. Command procedures
- 12. Text formatter

Programmers are often required to learn many other languages as well. These can include:

- 1. Different utility programs (and techniques) to maintain multiple versions of source, object and executable code, along with the patches and temporary fixes at each level
- 2. Special languages for stating problem requirements, design specifications, and project control
- 3. Special languages for describing and accessing a data base and for describing program interfaces to the data base
- 4. Langauges for producing reports and statistical analyses

Learning to program well requires fluency in many languages, and knowing which language to use in a given circumstance.

Recently, IBM and other companies have developed many new tools, such as interactive products, data base management, structured programming tools, design walk-throughs and full-screen displays. These products are all useful, yet the problems described above still exist. The problems are caused by weaknesses in the foundation of all these tools: the operating systems. Each tool serves only one segment of application development. This fragmented approach has caused programs, data structures, and programmers to communicate ineffectively at best, and sometimes not at all. Fragmentation is frustrating, time consuming, and most of all expensive. Thus, the answer to programmer productivity problems does not lie in continued enhancement of subsystems. Productivity changes should be introduced in the operating system where they can benefit all languages, all subsystems, all applications, and all users.

REFERENCES

1. John P. Ehrman, "The Babel of Application Tools," *Proceedings of SHARE 53*, New York, 1979.

THE LSRAD APPROACH

In the preceding sections, several problems facing the data processing industry in the 1980s have been described. The fragmentation of the programming environment is a major inhibitor to application development. The task force proposes to reduce fragmentation by applying the concept of synergism.

In the first section of this chapter, synergism will be defined and explained.

The second section will describe how synergism can be applied to computer systems. Guidelines will be presented for the design of flexible and usable systems that will accelerate the growth of computer usage during the 1980s. Although the task force encourages the development of new operating systems and products built with synergism as a design premise, the data processing industry cannot afford to abandon the existing systems.

The final section describes how synergism can be applied to the evolution of these systems. An approach to alleviating the application development bottleneck by providing some new primitive concepts to the basic operating systems is described in greater detail in Part II.

System Synergism

This section will define synergism, and describe some of the properties of synergistic systems. Some examples will be drawn from nature, which tend to exhibit highly synergistic behavior.¹

A system is a collection of interdependent entities forming a unified whole working toward a common goal. The term *synergism* describes this integrated behavior. Webster defines synergism as the cooperative action of discrete agencies such that the total effect is greater than the sum of the effects taken independently.²

PROPERTY OF UNPREDICTABLE WHOLE SYSTEM BEHAVIOR

This concept that "the whole is greater than the sum of its parts" has a property that is not obvious: the behavior of the whole integrated system cannot be predicted from the behavior of any of its components when viewed separately. This property is illustrated by examining the synergistic behavior of chrome-nickel-steel. This alloy is composed of several metals with the following tensile strengths:

| lron | 60,000 psi | | | |
|------------------|------------|--|--|--|
| Chromium | 70,000 psi | | | |
| Nickel | 80,000 psi | | | |
| Traces of carbon | | | | |
| and other metals | 50,000 psi | | | |

(where psi is pounds per square inch)

According to the logic that a chain is no stronger than its weakest link, the expected strength of the resulting alloy would be 50,000 psi (for the carbon). According to the logic that the whole is equal to the sum of its parts, the expected strength of the resulting alloy would be 260,000 psi In fact, the tensile strength of chrome-nickel-steel is 350,000 psi The behavior of the whole is unpredicted by the behavior of its parts.³

This property is a major stumbling block in analysis techniques used to predict system behavior. Conventional methods divide a system into its constituent components, assume the parts are unrelated, and then analyze each component in great detail. Isolating the components from their environment in the system prevents any analysis of synergism, since the components are divested of their associative potentials. Indeed, the basic assumption that the parts are unrelated has precluded the existence of any synergistic behavior. This approach gives an accurate picture of the differentiated behaviors of the components. It is useful to think of synergism as representing the integrated behaviors rather than the differentiated behaviors.

PROPERTY OF THE WHOLE SYSTEM

A second property of synergistic behavior exhibited by a system is that starting with the whole and some parts, unknown parts may be discovered. An example to illustrate this may be drawn from Euclidean geometry. The sum of the angles of a triangle totals 180 degrees.

System Synergism

Thus, if any two angles are known, the third can always be computed. Analysis of the behaviors exhibited by Uranus in the solar system led to the discovery of the eighth planet, Neptune.

PROPERTY OF DESIGN DURABILITY

Nature also teaches a lesson in design durability. The simpler and more nuclear an entity, the more frequently it is employable in building other entities. For example, hydrogen, with atomic number one, is the simplest of all the elements, and is composed of a single proton and electron. About 90% of the universe is composed of it. The elements that are more complex (and of higher atomic weight) are much less common, and generally less stable. All of the man-made elements are extremely unstable, most having half-lives of a fraction of a second. Thus, smaller and simpler systems in nature tend to occur with greater frequency than large complex systems.

SYNERGISTIC PROPERTIES

The properties presented above are summarized as follows:

- 1. Synergism is the behavior of whole systems unpredicted by the behavior of their parts taken separately.
- 2. Starting with the whole system and some parts, unknown parts may be identified.
- 3. The simpler and more nuclear the entity, the more frequently it is employable in building other entities.

REFERENCES

- 1. Ludwig Von Bertalanffy, General Systems Theory, New York, 1968.
- 2. Webster's New Collegiate Dictionary
- 3. R. B. Fuller, Synergetics, New York, 1975, pp. 3-20.

Synergistic Computer Systems

A human using a computer system reacts to the entire system and is, therefore, sensitive to the synergism of the system. As described in the section on Fragmentation, a human must interact with many components of the system to accomplish even the most simple task.

One can think of the external interfaces to these components as defining a tool. Each tool is used within a context or frame of reference. Unfortunately, as the user switches from using one tool to another, the context also changes. Inconsistencies in interfaces mean the user must enter different keywords for similiar functions. Interfaces with hardware or computer orientation are foreign to the user's experience. This requires a conceptual change of context, which is even harder to make. Trying to communicate information with foreign terms seriously handicaps the process of intuition, which helps people guess the right approach when they are not completely sure about what to do. All the components of the system should build on one another and support each other to present a consistent, logical view to their users.

The application developer attempts to solve a problem in logical terms. The programming environment should allow him to mirror his thoughts in data definitions rather than in foreign hardware terms. The programming environment should also permit simple communication with the computer system, describing what is logically needed to accomplish the task. By minimizing the programmer's need to switch his frame of reference, the job will probably be completed faster. The user's perceptions of usability and human factors is directly related to system synergism.

Since synergism is not widely understood or accepted, it is not planned or monitored during system development. A consequence of the first two properties described above is that synergistic systems must be designed from the outside-in. Current computer system design is from the inside-out. The components and subsystems tend to be developed in a stand-alone fashion. Logically, this would seem to be a good approach, since it would appear to be difficult to build a high quality system with low quality components.

Current computer design techniques are so preoccupied with the development of the components that frequently the conceptual integrity of the system considered as a whole is lost.¹ The reasoning behind this approach appears to be that combining a large number of high quality components will guarantee a high quality system. The first property of synergistic system behavior implies the unpredictability of the final outcome of this approach. Synergistic behavior, or the lack of it, generally shows up as a surprise during system test (or worse, in the customer's installation).

Before starting to build the high quality components, attention must also be paid to their integration into the final system, and to the behavior of that system considered as a whole. System developers have demonstrated their capability for producing high quality system components. However, the final step of integrating the components into a system, and the analysis of that system as a holistic entity is not performed.² Therefore, as long as IBM produces computer system components rather than computer systems, any synergism in operating systems will be accidental rather than planned.

Synergistic Computer Systems

FUTURE DEVELOPMENT

The current state of large systems is largely due to the way they are developed. Future systems must be developed with a total system approach that recognizes (and encourages) synergistic interactions of its components. This holistic approach must extend to all aspects of architecture, design, and implementation. The system should reflect one set of design ideas rather than a mixture of independent and unrelated ideas. Conceptual integrity should be the most important consideration during system design.³ Once component construction is started, trade-offs among reliability, performance, security, usability and other factors should be considered from a system wide perspective. Simple interfaces evolve from conceptual integrity. Each part of the system must reflect the same design goals and trade-offs. The interfaces should use the same syntax, format, and semantics.

Conceptual integrity in a computer system is more likely to be achieved if there is a single small set of master designers. The division of the design into many tasks performed by many people makes it difficult to achieve conceptual integrity unless each piece is constantly compared to the master plan. The goals of each component must be subordinate to the goals of the entire system. The situation is worsened if the people involved are spread across organizational boundaries or separated geographically. The chief programmer team is an attempt to achieve conceptual integrity in large projects by limiting the number of people who actually influence system design.⁴

Function alone does not define a well designed system. If the time saved by adding a function is outweighed by the additional time required to learn, remember, search manuals, and respecify the function when it does not work the first time, there is a net loss in system usability. (It is better to leave the function out than to sacrifice the conceptual integrity of the system.) Usability is enhanced by simple interfaces. However, simplicity is not the only factor in a good system design. If too simple, the system may not be powerful enough to perform useful work. Therefore, good system design strikes a balance between function and simple interfaces.

SYSTEM TRADE-OFFS

If synergism is so important for successful systems, why is it virtually ignored in current computer system development? There are two sets of trade-offs affecting decisions made during the development of a product which tend to insure that synergism is ignored: short-term versus long-term costs and builder costs versus user costs.

Planning for synergism increases the short-term cost of a product to the manufacturer. Since a smaller number of people will be involved with the product over a longer period of time, the development cycle will probably be longer. The original development cost may be higher due to prototyping work, iterative design, and continual comparison of components against original system objectives. The integration cycle will also be longer. However, even though the original development requires more resources, the total cost of the product spread over its lifetime will probably be much lower because the maintenance costs will be lower. Customer acceptance and satisfaction will be higher, and the conceptual integrity of the base system will provide a better foundation for future growth. Unfortunately, there are marketing pressures to release a product as soon as possible. Therefore, current system development favors the short-term over the long-term. Although the costs of producing systems without synergism are much higher, most of these costs are absorbed by the user of the product rather than by the builder. The hidden costs include training, incompatibility, maintenance, and long development cycles. The greatest and least obvious cost of all is the loss of flexibility caused by fragmentation. Usability is degraded. Since the manufacturer is unaware of these costs, decisions which benefit the builder will always take precedence over those which benefit the user.

Usable synergistic systems will be built only if the decision process is broadened to include the long term costs and consideration is given to the user. Systems must be measured by their usability rather than by their raw function or speed.

PRINCIPLES FOR SYNERGISTIC COMPUTER SYSTEMS

- Systems must be planned and developed with the synergistic interactions of their components as a design goal. The system must be treated as a whole and definite system goals established.
- Maintaining the conceptual integrity of the system must be the most important consideration during system design, development, and maintenance. Design trade-offs for individual system components must be subordinate to the whole system goals.
- Functions must be provided with simple interfaces.
- The definition of system costs must be broadened, when evaluating system alternatives, to include the costs borne by the user incurred by training, incompatibilities, and maintenance. Systems should be measured by their usability.
- New systems must exhibit consistency across all external interfaces. A major cause of application development overhead is the need for developers to simultaneously utilize a large number of inconsistent facilities. The overhead adds significantly to the mental effort required to solve problems.
- The systems must be human and application-oriented, not computer-oriented. It seems unnecessary for a user to have to be aware of linkage editors, loaders, compilers, allocation, and many other "basic" concepts of current operating systems.
- Future systems must encourage the development of both hardware-independent and software-independent applications. There should be no options to tell the system how to do something a little bit faster in a special situation.
- Every attempt should be made to eliminate arbitrary restrictions wherever possible. When this is not possible, a relaxation of restrictions which satisfy the requirements of the great majority of users is a desirable interim step.
- Commonly used functions should be defined as system primitives. There should be one common parsing routine, one terminal interface, and one standard file interface available to both user and programs.
- The system must manage all computer resources without user direction. Space and device management should not be the concern of the user, and he should not be aware of them.
- The system must promote standard interfaces to encourage the interchange of information in programs and data. The programmer should be able to connect any program with any file or any other program that he has the right to access.

Synergistic Computer Systems

- It is necessary to abolish the distinction between system programmer facilities and user facilities. Application developers, like system programmers, need subsystem protection, development tools, and debugging aids. The users of these tools should be controlled by the installation through access privileges.
- Future systems must make it possible to build and execute applications without preplanning. Future applications must be open-ended. Interactive applications are by nature dynamic. Frequently it is impossible to know how programs or data will eventually be used.
- The system must anticipate and encourage the development of dynamic programs and dynamic development methodologies. This includes support in the languages for dynamic program linkage and program modification.

The principles and evaluation criteria outlined above must be applied to the design of all software systems and products, not just to operating systems.

Attempts have been made to increase productivity by improving development methodologies, subsystems, and languages. Each its place, but they do not provide a solution with sufficient leverage and tend to increase fragmentation.⁵

Computer programming consists of accessing and of manipulating data. The next chapter will summarize how these two operations can be simplified by making the operating system be responsible for the mechanics of storing and retrieving data. There should be a system goal of presenting a simple, easy-to-use view to all users, whether they are system programmers, application developers, or end users.

REFERENCES

- 1. Frederick P. Brooks, Jr., The Mythical Man Month, Reading, Massachusetts, 1975.
- 2. C.West Churchman, The Systems Approach and Its Enemies, New York, 1979 pp. 3-31.
- 3. Brooks, p. 42.
- 4. Harlan D. Mills, "Chief Programmer Teams: Principles and Procedures," IBM Federal Systems Division, FSD 71-5108, June 1971.
- 5. "Trends in Application Software: 1980, 1985, 1990," New York, The Diebold Research Program, Diebold Group, Inc., September 1975.

THE WHOLE SYSTEM

This chapter consists of two sections. The first summarizes the LSRAD proposals. The second describes the benefits in programmer productivity derived from these proposals.

The technical proposals offered by the task force represent a significant attempt to develop a simple, consistent base for evolution in the 1980s. Some of the proposals apply to the underlying operating system where the hardware is managed. Some are directed to a higher level, nearer the user interface. Taken as a whole, the proposals present a consistent, comprehensible environment for all computer users. For specific details regarding the technical proposals, refer to Part II.

The proposals address the programmer productivity problem by reducing the requirements in the application development and maintenance processes. The benefits will not all come at once; productivity will not take an immediate quantum jump. The changes that are put into the operating systems can be exploited by products such as IMS and COBOL in the future. In time, all new applications will benefit as they are developed with the new subsystems and languages.

A New Direction

The task force developed a set of guidelines for producing synergistic computer systems which will be more powerful in function, but simpler and easier to use. These principles were presented in the previous chapter. The task force then applied them to the current systems, MVS and VM/370. A set of technical proposals for evolving current systems toward an enhanced programming environment was developed.



All of the LSRAD proposals are based upon an enhanced version of the present paging subsystems. On a system with these proposals, the users would deal only with virtual memory as a technology for storing data and executing programs. Users will be unaware of the physical storage technologies which are being used to support this enhanced virtual memory. Users would not need to know of rotating devices, mas storage systems or tapes, or of the intricacies of using them. IBM will be able to introduce new storage technologies at whatever pace the laboratories can develop them.

Figure 1

In essence, virtual memory would become a new interface between the bottom level of the operating system and the higher levels. Virtual memory has many properties which make it an ideal interface. It is simple, in concept, in definition, and in use. It provides a growth path for the future.

The named spaces described in Part II extend virtual memory so that it is suitable for use as a building block in many portions of the system. Named spaces, coupled with an extended addressing facility, provide a mechanism to allow programmers to access nearly all necessary data in virtual memory.

For those situations where files are still necessary, a new, simple, device independent file system based upon the paging subsystem and named spaces is proposed. The new file system would contain integral sharing facilities and would support all data types. With the new file system, the operating system, not the user would manage space. Future versions of data base management systems could also be built upon shared named spaces and device independent I/O.

Since all these facilities are based upon named spaces, integral sharing can be implemented as an inexpensive and uncomplicated natural feature. This will encourage communication among applications and among application developers. Furthermore, building all of these facilities based on virtual memory rather than on rotating memory devices makes it possible to introduce an integral hierarchical storage manager which supports named spaces, files, and data base management systems. The combination of the hierarchical storage manager, device independent I/O, extended addressing and named spaces with generalized sharing comprises the first group of LSRAD proposals. They are described in greater detail in Part II, in the chapter entitled *Data in Memory*. This group provides a conceptually simple, yet powerful, set of facilities to build both subsystems and application packages. This interface can be described with a small number of statements, thus making it comprehensible to all programmers. Each application has a large virtual memory, a few tools for allocating and sharing portions of that memory, and facilities to connect and disconnect external data structures to and from the memory. A file system interface will be provided which keeps track of the logical relationships between records without concern for physical data storage.

LSRAD Proposals



This foundation makes it possible to implement proposals to simplify the concepts which shape today's programming environment. This second group of proposals is described in more detail in *Enhanced Programming Environment* in Part II. Given that all data storage facilities can be isolated from the hardware and based upon a unified virtual memory technology, it will be much easier to develop and maintain subsystems and applications.

Figure 2

It is possible to provide an access control facility which protects all data and programs in internal and in external storage. This facility must be an integral part of the operating system. It is not acceptable to partially protect data, or to have different access authorizations depending on whether the data is in internal or external storage.

The foundation makes it much easier to implement a streamlined command system and a human-oriented execution and testing environment. Most of the hardware-oriented concepts can be eliminated, along with the need for meticulous preplanning which characterized the environment of the sixties (e.g., the need for manual allocation, linkage editing, and isolating jobsteps). Once the number of extraneous computer-oriented concepts in the programming process is reduced, the remaining concepts will become more obvious and usable. It will also be possible to build new programming tools and languages upon this base.

The subsystem protection facilities will make it possible to protect the integrity of components as they are developed. New symbolic debugging facilities can be developed which support all languages. Program and data generators will be simpler to implement when all data can be directly addressable. The mapping technologies will make it possible to have the source statements and symbol tables of a program embedded in the executable output at no cost to loading and execution. There are many enhancements to languages and subsystems which will be much easier once these proposals are implemented.

IMPLEMENTATION CONSIDERATIONS

There are many possible ways to implement the LSRAD technical proposals. The proposals are designed to produce a synergistic application programmer interface. Thus, any implementation even of the entire set of proposals, which appears fragmented to the users is counter to the LSRAD strategic proposals. Ease of use and consistency must be at the very top of the list of design considerations. The technical proposals build upon each other. For example, the dynamic loader, file system and named space facilities all depend upon the ability to map large quantities of data to an address space with no I/O operations required by the mapping process. Only when a page is referenced, will I/O operations be necessary. Otherwise, perforance would be be prohibitively poor.

The LSRAD proposals are designed to let the system manage space without user awareness. The application programmer is responsible for the development of the application package without directing the management of physical storage media. The LSRAD proposals stress the need for less "computing" in computer programming. There are additional functions which are candidates for system management rather than user management, e.g. the elimination of linkage editing.

When considered together, the technical proposals offer a new direction for restructuring the fundamentals of our application programming environment. They demonstrate that evolution is possible and that consistency can be attainable. They are not the final answer, but they are a step in the direction of future systems which have conceptual integrity, ease of use, consistency, and simplicity as design goals.

Programmer Productivity Benefits

The LSRAD proposals directly address the programmer productivity problem. The benefits are seen in three areas. First, application development will be simplified, reducing the staffing requirements by up to 35%. Second, the system will be simpler and more consistent than today's fragmented systems. This will have powerful synergistic effects on application developers, additionally enhancing their productivity. Third, a comprehensive system will enable more noncomputer professionals to use data processing effectively in their daily work.

REDUCE APPLICATION DEVELOPMENT CYCLE

The following model was developed to illustrate how basic changes to the operating system could be utilized by all computer programs and users.

A Typical Computer Program



Most computer programs consist of an input phase, a processing phase, and an output phase. The actual code that accounts for the mechanics of the input and output and which manipulates the file structures is probably distributed within the programs that form the application.

Figure 1

The process phase has two parts. One part is the implementation of the functional specification, doing what the users want done. The other part of the process phase deals with mapping the functional requirements onto the physical constraints of a computer and operating system. The mapping might include squeezing the business data into a main memory that is too small, even with virtual storage. The mapping also may include many sections of code for picking up data passed from program to program in temporary files of various formats. The shaded part, the business processing, is the part that people must do. It is the creative part of programming, developing the algorithms that implement the application. On the other hand, the unshaded areas (the input, mapping, and output) are mechanical things which machines can do very well and very cheaply.

The input, output, and mapping parts of an application can be thought of as managing the hardware boundaries between levels of storage. For example, when a file is read, the data is transferred across the boundary between external storage and virtual memory (internal storage). Today, users must understand a great deal about the computer system to move data across that boundary.

A closer examination of application development process will show where the 35% reduction in application staffing requirements comes from.
Programmer Productivity Benefits

The Application Development Process



The task force conducted research to find out what goes on in application development. The available literature was reviewed and studies conducted within the member companies to confirm the literature findings. Approximately 50% of application programming resources are devoted some type of maintenance.

Figure 2

Further research has shown that 50% is quite a conservative portion devoted to maintenance. The 50% figure is a composite number, and varies over the life of the program. The task force found numbers which varied from 40% to 75%.¹⁻⁵ The resources devoted to new development will be examined first.

New development starts with a requirements gathering phase during which the developers decide the size and basics of the problem. These activities culminate in a functional specification. At that point, the developers know what must be done. What then follows is a series of phases during which the application developers bring the computer and the business problem together. Developers spend most of the new programming resources in the later phases and are actually dealing with the machine. The LSRAD proposals can have a significant effect on these phases.

In most current application development, 15% of the effort is associated with requirements-gathering; 20% of the effort is directly associated with implementing the computational section of an application, that part which actually implements the algorithms. This is the creative part of programming, the part that challenges the data processing talent in each organization.

New Application Development



An additional 15% of the development is designing, testing, and debugging sections of code which perform the input, output, and mapping functions. The numbers are composites based on information gathered from project management statistics within the parent companies of several task force members and from available literature.⁶⁻⁷ This part of programming is a drudgery because it is routine, because there are scores of miniscule details with which to cope, and because it is fraught with potential for expensive and damaging errors.

Figure 3

This 15%, the mapping or storage management activities, would be done by the computer system if the LSRAD proposals are implemented.

The resources devoted to maintaining existing applications comprise the remaining 50% of application development resources. A useful model for system maintenance differentiates

three types of activities: corrective maintenance, perfective maintenance, and adaptive maintenance. The maintenance model was an adaptation from references |8| and |9|, adjusted by the collective experience of the task force members.

Corrective maintenance includes emergency bug fixes and operational program debugging. It comprises 10% of all application development activity.

Perfective maintenance comprises 30% of all application development, and is actually a form of new development. It includes user enhancements, improved documentation, and recoding for efficiency. It is a direct result of rapidly changing business environments and requirements.

Maintenance



Adaptive maintenance forms the remaining 10% of all application development. Adaptive maintenance is necessary today to keep production applications running as the hardware and operating systems change. The conversions of user programs and data files from 3330s to 3350s and from one release of an operating system to another take up unreasonable amounts of time, people, and money. These resources should be used in developing new applications.

Figure 4

The LSRAD proposals would nearly eliminate adaptive maintenance by decoupling the applications package from the hardware and operating system details.

Furthermore, considering perfective maintenance as a form of new development, there is the same breakdown between implementation of algorithms and input, output, and mapping which appears in the new development process. This would result in a substantial reduction in perfective maintenance, eliminating nearly 10% of the effort required for all application development activity.

Although it is not certain what portion of corrective maintenance would be eliminated, it is believed to be substantial, since program bugs increase rapidly as complexity increases.¹⁰ Applications developed with a LSRAD-based system will be simpler because boundary management is eliminated. To be conservative, however, no gain was assumed here.





When all of those pieces are added (15%) from storage management activity, 10% from adaptive maintenance, and 10% from perfective maintenance), a total of 35% of all application development activity may not be necessary.

32

Figure 5

Programmer Productivity Benefits

This is the motivation behind the two basic LSRAD proposals: data in memory and enhanced programming envoronment. With operating system facilities that support the concepts of data in memory, the application developers will not have to design and build those mapping sections time after time. With simple, logical I/O instead of today's hardware-oriented I/O, the developers will be able to concentrate more on the business requirements and less on the mechanics of hardware. The programs they build will be simpler and easier to develop and maintain. This applies both to IBM-developed packages and to customer-developed applications.

If the burdened cost of an application developer is about \$60,000 per year, 35% of that amount (about \$21,000) is going for things that the machine could do. Machine costs, which are dropping, can be traded for labor costs, which are rising.¹¹ Multiplying \$21,000 per year by the number of applications developers results in a sizable savings.

However, companies with two to five-year backlogs of new applications. are not going to layoff one-third of their staff. (The backlog figures are based on task force members experience within their own companies, as well as |12| and |13|.) Staff members presently occupied with the mechanical aspects of programming can develop new applications. This is the first benefit of restructuring the operating system. Applications developed on a LSRAD-based system would require up to 35% less programming resources than on conventional systems.

ENHANCE PRODUCTIVITY THROUGH SYNERGISM

The full strength of the LSRAD proposals lies in their synergism. The technical proposals build on each other. When looked at separately, each proposal does not seem to help the application developer a great deal. Yet, when implemented together, the proposals create a powerful, yet conceptually simple operating system.

As described in the section on Synergistic Computer Systems, a significant savings can be realized from this consistent conceptual approach to operating system design. If the system makes sense, users will often be able to guess the right way to communicate with the system to accomplish a given task. Frustration will be reduced, as a task will tend to work correctly the first time. Programmers will not have to consult innumerable manuals and attend months of classes to become experts in dozens of inconsistent environments. The educational savings alone could total hundreds of millions of dollars a year, and programmers could spend that time productively solving problems.

FOUNDATION FOR FUTURE GROWTH

The LSRAD proposals offer a strong foundation for the future. Over the next several years it will be possible to produce new development and testing tools and methodologies, based upon this simpler, consistent foundation. Development of these tools is not cost-effective today because of the fragmentation of current computer systems. When this enhanced operating system is installed with its new tools and methodologies, companies will be able to offer a usable computing system to their secretaries, chemists, and aerodynamicists. Non-computer professionals will be able to use the computer as is appropriate to their jobs. In most corporations noncomputer personnel far outweigh computer professionals. Consistent, simple interfaces would allow them to use data processing effectively.

REFERENCES

- 1. Edmund B. Daly, "Management of Software Development," IEEE Transactions on Software Engineering, May 1977, p.233.
- 2. B. P. Lientz, E. B. Swanson, and G. E. Tompkins, "Characteristics of Application Software Maintenance," Communications of the ACM, Vol. 21, No. 6 (June, 1978), p.466.
- 3. Chester C. Liu, "A Look at Software Maintenance," Datamation, November 1976, p.51.
- 4. Harlan D. Mills, "Software Development," *IEEE Transactions on Software Engineering*, December 1976, p.267.
- 5. Marvin V. Zelkowitz, "Perspectives on Software Engineering," ACM Computing Surveys, Vol. 10, No. 2 (June 1978), p.202.
- 6. John H. Lehman, "How Software Projects are Really Managed," *Datamation*, January 1979, p.129.
- 7. Zelkowitz, p.198.
- 8. E. Burton Swanson, "The Dimensions of Maintenance," Proceedings, 2nd International Conference on Software Engineering, October 1976, p.494.
- 9. Lientz, Swanson, and Tompkins, p.468.
- 10. Mills, p.270.
- 11. Philip H. Dorn, "1979 DP Budget Survey," Datamation, January 1979, p.165.
- 12. Dorn, p.163.
- 13. Jake Kirchner, "IBMer Sees DP Technology Booming Again," Computerworld, February 19, 1979, p.7.

PART II: TECHNICAL PROPOSALS

This portion of the LSRAD report presents a series of technical requirements. The requirements are synergistic and depend upon one another. Implementation of the technical requirements will produce a computing system that also meets the conceptual requirements discussed in Part I.

The requirements are presented in two chapters, each of which corresponds to a logical layer of operating system software. A third chapter contains task force recommendations on additional operating system considerations.

The first chapter, *Data in Memory*, contains requirements that pertain to the innermost layer of an operating system. This chapter offers five specific items that, if implemented, would establish the foundation for more usable systems now and in the future.

The second chapter, *Enhanced Programming Environment*, contains requirements that pertain to the next higher layer of the operating system. This layer includes command system improvements, integrated data access control, subsystem protection and an improved execution and testing environment. The chapter contains three general requirements that assist in using the facilities implemented in the innermost layer.

Each layer of system software builds on the previous one. In the highest layer, each application builds on all the layers of system software beneath it. Each layer of the system must be internally consistent and must exhibit a usable, well defined interface to the other layers. Otherwise, the system will become fragmented and will not endure.

The final chapter contains four essays discussing various is us pertinent to operating system improvements. The first two essays, Evolution versus Revolution, and Compatibility and Migration, discuss how to move from current systems to improved ones. The last two essays, On Options and On Limits, offer useful guidelines on how he improved systems might be designed.

DATA IN MEMORY

Data in Memory is a set of proposals for low level system functions. By themselves, the proposals do not have much utility. These proposals underlie all the other proposals and set the foundation for proposed future improvements.

The basic idea of data in memory is simple. The system should make the data in the memory appear extensive and homogeneous to the programmers. The virtual systems do this now, not for all storage but for a subset called internal (or working) storage. If the appearence of homogeneity were extended to include external (or permanent) storage, all levels of users would benefit.

THE PROBLEM

A significant portion of any application today consists of code which moves data across the boundary between internal and external storage. Programmers must know a great deal about computer systems to manage the boundary. Yet look at the storage on either side of the boundary (Figure 1).

| INTERNAL | EXTERNAL STORAGE | |
|--------------|---------------------|--|
| CACHE | | |
| MAIN | | |
| VIRTUAL | | |
| FILES | | |
| ONLINE ARCH | IVES | |
| OFFLINE ARCH | HIVES | |

The figure shows storage as a hierarchy. The top of the hierarchy is usually some type of cache buffer which supplies the data to the CPU. The cache is supported by main memory, which, in turn, is supported by virtual memory. On the other side of the boundary are files, and finally some kind of archival storage.

Figure 1

Users are accustomed to having the machine manage the boundary between cache and main memory. Since the advent of virtual storage systems, users have become accustomed to machines managing the boundary between main and virtual storage. On the other side of the boundary between memory and files is the IBM hierarchical storage manager transferring files from tape cartridges to virtual disk packs. The machine manages all these boundaries well.

But application developers must state explicitly when and how data is to be moved between external and internal storage. When application developers must deal with hardware boundaries, companies are employing expensive professionals as storage managers.

Fragmentation and confusion describe computer storage as implemented in MVS and VM/370 today. There is online and offline storage, internal and external storage, disk storage, drum storage, and virtual storage. To access these there are standard OS access methods and VSAM, file mechanisms and data base management systems, addressing and paging, and special access methods such as that used by APL. Each software subsystem, and sometimes each application, must be programmed to deal with all the types of storage, as shown in Figure 2.

Data in Memory

Subsystems and Applications





RECOMMENDED SOLUTION



Types of Storage

Figure 3

Characteristics of Storage Technologies

| | Virtual Memory | Files | DBMS |
|----------------------|----------------|-------|------|
| Access Speed | best | | |
| Simplicity | best | | good |
| Durable | | good | good |
| Large Capacity | 2 · · · · · | good | good |
| Sharing | | good | best |
| Protection | | good | best |
| Language Independent | | good | best |
| Data Independent | | good | best |
| Names | | good | best |
| | | | |

Figure 4

Virtual memory is not the best type of storage for all needs, but it is conceptually the simplest and most internally consistent mechanism. It is also the most pervasive because every program must deal with it. Virtual memory is the highest in perform: needs of storage technologies. Simplicity, consistency and pervasiveness are the ingredients needed in a LSRAD system. However, there is a need to enhance virtual storage to alleviate some of its weaknesses. It should be made durable, sharable, and able to be protected.

There will be a continuing need for virtual memory, files, and data base management systems in the future because each provides certain necessary services for application developers. However, all three must be further integrated into the entire application programming environment, including language processors debugging tools and documentation tools. In order

This situation is ridiculous. It makes systems difficult to learn, to use, and to build. It forces customers to wait years before a new technology such as the 3370 disk or the VSAM access method eventually reaches all subsystems. Fragmentation complicates the IBM development process, but furthermore, it seriously hampers IBM's ability to innovate.

From a theoretical point of view, the solution to the fragmentation is simple. A standard view must be defined for storage, and a single interface must be established to support and enforce that view. The software connects to one side of the interface and the hardware to the other as shown in Figure 3.

LSRAD proposes a large virtual memory as the obvious standard view. The interface becomes an improved paging subsystem. To understand this proposal and why it is obvious, consider the three mechanisms which are available for storing data in today's systems: virtual memory, files, and data base management systems. Each mechanism has advantages and disadvantages, as shown in Figure 4. to do this effectively, it is necessary to create a standard view of storage.

MANAGEMENT OF VIRTUAL STORAGE

A large virtual storage by itself is useful for a number of applications, but it is rather cumbersome. Most applications need several storage spaces with a few million bytes each. Once a large virtual storage is developed, a mechanism will be needed to break it up and to manage it.

The concept of named spaces serves this purpose. It is easier to manage something with a logical name rather than something with a binary address. In addition to the name, these spaces will require some kind of access control because some of them will contain confidential data. Finally, users will need a mechanism to share named spaces because some of them will contain data needed simultaneously in several parts of the company.

The task force proposes a large virtual memory as a standard view of data. The task force further proposes the use of named spaces, access control, and sharing mechanisms to manage the large virtual storage. Furthermore, the task force proposes extended addressing to allow named spaces to be much larger than would be possible today. A device independent I/O mechanism is also needed. Collectively, these proposals are called data in memory.

FILE SYSTEMS

Superficially it may seem that data in memory is a proposal to eliminate file systems, or at least to decrease their importance. In fact, that is not the case for two reasons.

The first reason is a short term one. Today all programs deal with files. Some of today's programs will exist in 1985, 1990, and even in the year 2000. With some effort, perhaps a minimal one, those programs and their owners and users can benefit from improved files. But the programs would likely need to be completely rearchitected to use named spaces. If evolution is to be achieved rather than revolution, then file systems must be improved along with virtual memory.

The second reason is longer term and somewhat subtle. Named spaces will give users natural access to long strings of data in virtual memory. However, people do not usually think in terms of strings of data. People think in terms of entries in files that are kept in drawers of file cabinets. People also think in terms of open files that are accessible in a work area or on someone's desk.

Data in memory is a set of underlying, primitive ideas. As such, they are useful, but are more useful as a foundation for other things. Users will need services to map the foundation concept of named spaces into individual entries, records and, perhaps, bytes. The file system offers such services.

Thus an enhanced file system benefits in the short term because programs can move to it quickly. It benefits in the long term because the file system will serve as a bridge from the user to external storage and later from the user to named spaces. With the enhanced file system users need never know that the underlying mechanism is changing.

FOUNDATION

If current systems were to be enhanced with all the proposals in data in memory, a reasonable first application would be to support subsystems such as IMS and APL. As Figure 3 illustrates, data in memory will allow a subsystem to build all of its internal structures upon a single unified view of the hardware that makes up memory. Those internal structures may be files, IMS data bases, APL variables, or large user-arrays. The subsystem (or the application) will be able to concentrate on its interface with users, not on its interface with hardware.

For purposes of technical presentation, the discussion to follow is divided into sections on Named Spaces, Sharing in Virtual Memory, Extended Addressing, and Device Independent I/O. These are followed by a discussion of a Hierarchical Storage Manager which is an integration of all the data in memory concepts. Bear in mind throughout that the basic goal is a unified view of storage and the means to manage it.

Named Spaces

The task force proposes named spaces as a way of extending virtual memory capabilities so that current systems may evolve towards more synergistic ones.

THE PROBLEM

Virtual memory has several severe limitations which restrict its usability. Virtual memory is not durable, sharable, or large enough. Data stored in virtual memory is not adequately protected. It does not support language-independent data storage or data dictionaries. Yet virtual memory is a simple, universal building block which is global in scope.

PROPOSED SOLUTION

A new system object called a *named segment of address space* or more simply a named space, is defined which has the simplicity and performance advantages of virtual memory. It has features which make it a more effective building block. A named space is a block of virtual memory with the following new capabilities and properties:

- It can hold programs, data, or text.
- It may be connected to a user's virtual memory, or it may be disconnected and maintained on long term storage media.
- It may be connected to several virtual memories at the same time.
- It has a name for identification.
- It is variable in size.
- It has an owner.
- Its owner may specify who may connect to it and with what access privileges.
- It can support execute-only protection.

RELATIVE STRENGTHS

Named spaces provide a simple foundation to support the environment of the eighties. Facilities built upon named spaces will be independent of storage technology and architecture. Named spaces provide a concrete basis for the following:

- Effective, inexpensive sharing for virtual memory, files, and data base management systems.
- Security within virtual memory. This base is available to both applications and system programs.
- Simple execution-time environment.
- Clean interface for the implementation of a transparent hierarchical storage system which supports all data and programs.

RELATIVE WEAKNESSES

- A 24-bit linear address space is too small to be used effectively for large data structures in named spaces. Extended addressing is required if named spaces are to be reasonably useful.
- Named spaces are not a replacement for data base management systems, although they are often perceived as such.
- Named spaces bind programs to the structure of data at compile time.

ON THE USE OF NAMED SPACES

The most obvious benefit for application programmers who use named spaces directly is the ability to save data structures without complexity. If a named space is thought of as containing a FORTRAN common block, an APL shared workspace, or a PL/I external structure, then when the application is brought into memory for processing the necessary structures will be implicitly connected to the program. No I/O statements will be required. The program will process the data and then disconnect when complete. The named space can be returned to permanent storage. APL already provides this type of data structure durability.¹

All data contained in named spaces should be easily and directly addressable. This will enable application programmers to work with an entire data structure rather than reading and writing individual records containing data elements. Data structures represent the logical format of data, while file systems require these structures to be decomposed into records for permanent storage. If the system paging supervisor understands the permanent storage format for named spaces, the data can be mapped into the user's virtual memory by changing entries in the user's page and segment tables without any I/O. As the program is executed, the required portions of data are transferred to main memory by the paging supervisor. Only the portions which are referenced or changed are involved in I/O operations. This makes it possible to substitute named spaces for very large files. Furthermore, there is no need to read records of sequential files to access other records. These facilities are similar to, but easier to use than the skip sequential facilities of VSAM. In effect, every file has random access capabilities when it is represented in a named space.

Programs should also exist in named spaces. If the compilers produce output in a directly executable (but not resolved or relocated) format instead of 80-byte card images, it is easier to build facilities which can map very large programs into memory without requiring any physical I/O.

The task force believes that it is possible to combine the function of files and named spaces; however it is uncertain whether this can be accomplished within 1980-1985. The task force also recognizes that existing programs will continue to run through the 1980s; thus a simplified file system is still needed.

In the long term, named spaces might supplant file systems. Data aggregates grow and shrink as records are added and deleted at arbitrary locations within the structure. When files are used, system-supplied subroutines perform the following functions: identifying the appropriate data, bookkeeping, maintaining the logical relationships, and managing storage. These management functions are still required for some users of data in named spaces, and it seems better to have system provided subroutines rather than to have each user build them. Initially the facilities (insert, delete, locate, etc.) might be provided as subroutines, but eventually they should be integrated into the high level languages. This would provide support for expanding and shrinking objects as well as for complex relationships (keyed named spaces), with none of the complications of today's file systems.

REQUIREMENTS

The extensions to current virtual memory facilities must have the following properties:

- 1. Programmers must be able to subdivide virtual memory into logically distinct segments, called named spaces.
- 2. A new *disconnect* function is required which maps a designated piece of a virtual memory into permanent storage. Disconnected named spaces should be catalogued in a manner similar to today's files.
- 3. A complementary *connect* function is required to attach one of the catalogued named spaces into a virtual memory.
- 4. The same access control facility which protects long term data storage (files) must be used to protect named spaces.
- 5. For an efficient operation, the Connect and Disconnect mechanisms should "map" data between files and virtual memory, because the most common use of the named spaces will be to connect millions of characters of data to a program. When the data is unloaded from the named space to a file, only those elements which were changed would require rewriting.
- 6. Language implications:
 - a. A consistent definition of the logical format of external structures must be established and recognized by all languages (FORTRAN common, PL/I external, APL shared variables, and so on).
 - b. The executable output of the language processors could be in a directly executable format. Rather than building text decks, we can imagine language processors building a collection of named spaces with perhaps an extra named space to contain the relocation and linkage information for the module.

REFERENCES

1. Dietrich Bartel and Dietmar W. Erwin, KFA Juelich, Juelich, Germany, "Large APL Workspaces in a Virtual Memory Environment," Session B452 at SHARE 52, San Francisco, March 1979.

Sharing in Virtual Memory

As the application development process has become more complex, the need to build upon existing programs and to extend the function of existing application packages has become crucial. As companies have begun to develop centralized data bases, the need to consolidate data while permitting access to data concurrently from several applications has also become crucial. Consequently the ability to communicate easily and cheaply between application programs and between programs and data bases is a basic requirement for application programming in the 1980s.

Shared virtual memory will provide this capability to computer systems. Both programs and data can be placed in shared memory. This section discusses shared virtual memory, and its integration with the rest of the LSRAD proposals. It is critical to note that any implementation of sharing must be integrated with data access controls.

The ability to share virtual memory is available, but limited, in both MVS and VM/370. In MVS, programs may be shared in the Pageable Link Pack Area (PLPA), and data may be shared in the Common Service Area (CSA). In VM/370, discontiguous shared segments may be used for programs or data.

THE PROBLEM

The ability to share virtual memory cannot be easily extended to application programs or to application data for the following reasons:

- 1. In both MVS and VM/370 the use of shared areas is restricted to programs included in the operating system, or to programs authorized for this special use by installation data base administrators. Many installations are reluctant to authorize the use of shared areas due to integrity and maintenance considerations.
- 2. Since the sharing is global in MVS and VM/370, any increased use of shared virtual memory for either programs or data will decrease the amount of private virtual memory available to all computer system users.
- 3. There are no system controls to restrict access to programs or data in virtual memory.
- 4. Existing storage key mechanisms cannot fully protect programs or data in shared virtual memory from destruction.
- 5. There are no system controls to limit the application's use of shared virtual memory. In MVS, the amount of CSA available is fixed after system initialization. An application could cause these system limits to be exceeded.
- 6. Data in shared memory is not supported by current read/write data access mechanisms.

BENEFITS

The benefits of sharing both application programs and data in vir ual memory are the following:

1. It will simplify both application and operating system programs by providing an efficient mechanism for communication between cooperating programs

- 2. It will reduce the total system overhead involved in executing programs and in accessing and updating data bases
- 3. It will increase the efficiency of accessing data bases concurrently from multiple applications

PROPOSED SOLUTION

The current System/370 architecture can adequately and efficiently support extensive dynamic sharing. The two-level segmentation hardware can address up to 256 separately protected units, called segments, with up to 64 kilobytes each. One of the important features of the System/370 implementation (as opposed to the MULTICS implementation of segments¹) is its ability to homogeneously address groups of segments. Thus, a programmer could have one 16-megabyte contiguous block of virtual memory, 256 blocks of 64 kilobytes each, or any combination in between. It is not necessary to program "mirrors" to create objects which are bigger than the standard system segment size.

MVS takes advantage of this homogeneity to handle its system nucleus and Pageable Link Pack Area (PLPA). All users share one copy of these pages and, in fact, share the same page tables needed to address these pages. A connection to a named space would only involve changing entries in a segment table if the object already existed in another virtual memory. If the request was to create a named space, the page table would be created and then connected to the segment table. When the object is disconnected, the part of the page table containing information about the ex ernal location of the pages could be saved with the data pages of the object.

VM/370 demonstrates the feasibility of a facility which causes segment and page tables to be dynamically modified. In VM/370, each CMS user's address space looks exactly like a homogeneous unit. VM takes advantage of the fact that most virtual machines running CMS have identical data in the first few pages of each virtual machine. VM runs with only one copy of those pages in memory and it is shared by all users. If a user changes any part of this shared address space, the system generates a private copy of the page for his private address space. All other users continue to use the shared copy. This private page is called a "virtual copy" and exists only for the address space which requires read/write access to the page.

An implementation of shared virtual memory must not require that all sharers connect to the named space at the same virtual address. When there is widespread usage of named spaces to hold both programs and data in memory, a single user will be able to have hundreds, even thousands, of named spaces in memory concurrently. Since any system can have hundreds of users, there may be millions of combinations of users and named spaces. Therefore, each named space cannot exist at the same virtual address in all of the virtual memories which contain the named space (unless the virtual memories are much larger than the present ones, such as 2⁴⁸ bytes). This conflict implies that named spaces, which are shared between various users, must not contain any absolute relocatable data. All pointers or other variables which refer to locations within the named space must contain offsets relative to the beginning of the named space. Since different users may have different named spaces at different addresses, no shared named space must be contained in an unshared named space held by each user.

While this feature complicates the task of the compiler writers, it simplifies several other portions of the operating system. For example, it is not necessary to relocate shared named

Sharing in Virtual Memory

spaces as they are moved between virtual memory and files. If it is decided to expand a named space (either by the user or by the system) beyond its current limits, and the next contiguous virtual memory addresses are allocated, the named space could be moved to another address without difficulty. This would be done by altering page tables, not physical movement of the contents of the named space. Implementation of this feature would presuppose that access to named spaces was through the equivalent of a vector-like technique and indirect addressing. The vector would exist in a separate, private named space for each user.

Any implementation of sharing must include integrated access controls. There is one significant architectural deficiency in the System/370 virtual memory implementation which must be resolved so that application developers may improve upon current programming technologies. Access rights to a page of data are now enforced through the storage key mechanisms. Unfortunately, the storage key is associated with the data, and not with the user. Thus, all sharers of a page must share it with the same access (e.g., read only, write). This will effect efficient sharing of virtual memory.

One possible way to associate a set of data with a specific user is to involve the segment and page tables in the protection mechanism. Each user has a unique segment table and, possibly, unique page tables also. When a virtual memory address is translated, the CPU can remember the access privileges from the corresponding segment table entry, the access privileges from the corresponding page table entry, and the storage key itself. The most restrictive privilege would dictate the allowable access. If this were implemented, all the storage keys might be left with full access allowed, and segment or page table entries used for security. Further considerations for protection of data and programs in virtual memory are discussed in the next chapter.

With the ability to share virtual memory and integrated access controls, it is possible to safely and efficiently provide access to concurrent data bases from multiple address spaces. A user can place the data in a named space, define the rules for its use and make it available for any selected users who agree to the rules. The current file system could exploit shared named spaces. It would be possible to have a lock byte for each record in a file. Read/write locks could be maintained continuously and inexpensively for the entire named space. Interlocks could be maintained by the equivalent of the Compare And Swap CPU instruction, which is faster than the Enqueue and Dequeue SVCs. This approach would enable each record within a data structure to be inexpensively locked when necessary. This inexpensive interlocking capability would provide sharing with integrity as a basic function of the file system. Each time a record is accessed, it could be locked. Thus, simultaneous access and update would be available to all file users.

It is now possible to extend this concept as a means to share programs and data in virtual memory between loosely coupled computer systems. The operating system, which supports named spaces, can provide an appropriate point to support loosely coupled or remote systems sharing. With this approach, multiple copies of a page of data could be maintained "transparently" on several systems. If, for example, a page were to exist on two systems for simultaneous update, each system could have a copy present and mark the user's page table read-only. When the data is changed, a protection exception will occur. The system will notice that a copy exists on another CPU and will request that the other CPU remove its copy. When that is complete, the first CPU will restart the user with read/write access to the page. If the second system wishes to access the page, a request will occur to acquire a read-only access to the page. This requires changing the protection flags to simulate read-only again. The page will exist in one of the following states:

5

- Read/write in one system
- Simultaneous read-only in many systems
- Not in any system

The control and record keeping may exist in the file controller which contains the data or in one or all of the systems which access it.

It is important to consider that the user does not need to be aware that the sharing is occurring between systems. The sharing is implemented as a system primitive and thus applies uniformly to named spaces, files, and data base management systems.

SHARING REQUIREMENTS

The requirements for sharing programs and data in virtual memory are the following:

- Sharing must be available to all application programmers without the need to interface with a data base administrator or support group.
- Sharing must be integrated with a common access control facility for both named spaces and files.
- Named spaces must be selectively and safely sharable between users at the discretion of the named space owner.
- Sharing must not require users to relinquish a portion of their virtual memory for an object which they are not sharing.
- Sharing must not require that all sharers have a named space at the same virtual address.
- Loosely coupled sharing must be transparent (except perhaps for performance) to application programmers.

LANGUAGE IMPLICATIONS

The language implications of sharing programs and data are the following:

- Languages must all w programmers to specify whether structures and code are to be shared or private.
- It should be possible to provide a considerable savings in paging overhead if all language processors can produce output in two control sections: a pure one which contains all nonrelocatable and reentrant code and may be shared by all users of the module; and a second one containing all relocatable information and variables. Each user of the module will get a private copy of the second control section.

REFERENCES

1. R.C. Daley and J.B. Dennis, "Virtual Memory, Processes and Sharing in MULTICS," *Communications of the ACM*, Vol. 11 No. 5 (May 1968), pp. 306-312.

Extended Addressing

The current implementation of virtual memory in System/370 hardware, and VM/370 and MVS operating systems is too restrictive. The current implementation of virtual memory eliminates the need to overlay programs, but it is still too confining to permit the natural mathematical representation of real-life objects. For example, a geometric representation of an automobile front quarter panel requires about 3 million bytes of memory and fits nicely into an MVS environment. Unfortunately, the representation of a door 1 itch takes about 20 million bytes. This requires that the programmer develop algorithms to overlay parts of the door latch representation into available virtual memory. This precludes ever being able to see the entire latch at one time and greatly increases the complexity of the program design.

Another example of natural representation is the mathematical model of an oil field produced from seismic data. The representation of the oil field requires from 40 to 80 million bytes of data. In one documented case, the programming required to map the mathematical representation into a limited addressing space comprises more than 70% of the effort required to analyze the data. Furthermore, it is nearly impossible to change the data reduction algorithms because the existing mapping function requires that the data be accessed with the specific pattern required by the established technique. When this application was implemented in an environment which permitted a natural addressing of the oil field data, the development cycle for new algorithms dropped from 24 months to 6 months. (Based on discussion with Robin Thornton, Chevron Oil Field Research, La Habra, California.)

A third example of the limitations of virtual memory is visible in any data base management system. A DBMS has several functions which include: relating names and data, enforcing security and providing a storage mechanism for data. Much of the DBMS implementation (and many of the restrictions) arises from the need to overlay parts of the data because the entire data base exceeds the maximum allowable size of virtual memory.

THE PROBLEM

Application programmers must be able to concurrently address and control more than 16 megabytes of space, the current limit imposed by MVS and VM/370. The user must be able to address both the smallest and the largest element with the same mechanism and with equal ease.

BENEFITS OF SOLVING THE PROBLEM

The most important advantage of a large (substantially unlimited) address space is simplicity. Extended addressing eliminates the need to overlay data and/or programs. Large data structures and data files can reside in the user's address space without conscious data management or movement required by the user. The programmer can concentrate on the application solution. The introduction of virtual memory has eliminated program overlays. With large named spaces, it will be possible to eliminate data overlays. The entire file will be addressable at one time.

A second opportunity exists for exploiting very large named spaces. The seismic data reduction application considers its data to be a three-dimensional array. Yet typically 95% of

the array is empty of no interest. Rather than compressing the interesting part into a smaller data structure, the application program used the parts of the larger array which were considered important. The remaining parts did not have to exist physically, since all values were zero. When the program accessed these parts of the array, the system could materialize a page of memory containing all zeros. Similarly, when the named space was disconnected from the application for long term storage, nonexistent or all-zero pages could be eliminated, which reduces the long term storage requirement for the named space. This sparse named space facility allows the programmer to have a large logical data structure with small storage requirements. This technique would also reduce the storage requirements for executable compiler output from compilations with large data structures.

Sparse arrays could be used in many types of operations. For example, a multidimensional array can be constructed in which each subscript is an alternate index to a data set. The design of the dataset could subset the data by assigning special meaning to each of the subscripts. Then, the program could "view" a subset of its data by merely changing a value of a given subscript.

PROPOSED SOLUTION: IMPLEMENT FULLWORD ADDRESSING

It is important to consider how to introduce extended addressing without disregarding the current investment in programs.

The virtual PSW defined in the System/370 Principles of Operation has reserved up to 40 bits for addressing.¹ Surely 1 trillion bytes of addressing space was thought adequate for most applications over the next fifteen years (assuming a 10-year life for the new system). However, further study revealed that a substantial modification, such as 5-byte address constants, new instructions to manipulate them, new registers, and so on, would be required to support such a system. Therefore, in order to obtain compatibility, the task force proposes to link the new extended addressing mechanism to the 32-bit architecture of the System 360/370 word. If this were done, the system hardware design would require only minor changes.

The System/360 Model 67 was available with a 32-bit addressing option. (The implementation of 32-bit hardware in the 360/67 uncovered a few architectural 'quirks', such as, BXLE and BXH working backwards for addresses above 2^{31} , because the comparison was algebraic rather than logical. To bypass these problems and to make it much easier for the compiler writers, 31-bit addressing seems to be a more reasonable approach than 32-bit addressing. We have often spoken loosely of fullword addressing; what is needed is substantially more than 24 bits.) In addition to the known changes implemented in the 360/67, an additional bit would be required in the PSW to indicate whether the user was currently in 24-bit or in extended addressing mode. Two additional instructions might be defined for compatibility with existing programs, branch and link into 24 or extended mode and, return to original mode. The hardware address bus would also have to be extended to accommodate full word addresses.

With this hardware base, MVS could be extended as follows. The current MVS address space architecture would be preserved, but most of the MVS code and data areas would be moved from the system nucleus, link pack, and common storage areas to a new area outside the range of $0-2^{24}$. The skeleton routines which branch to the relocated system routines would be left behind as required. These skeletons would allow users running with 100 % compatible System/370 code to have more than 15 million bytes of addressing space instead of the current IBM guarantee of 8 million bytes. If this path is chosen, it is vital that the hidden

Extended Addressing

segmentation be invisible to all programs. A similar strategy would be equally appropriate for VM/370.

ALTERNATIVE SOLUTION: SEGMENTED ADDRESSING

The fullword addressing solution requires that all necessary information coexist in a single contiguous virtual memory. A multidimensional virtual memory offers a more open-ended environment with different tradeoffs. Each named space could exist in its own virtual memory with hidden base registers used for cross memory addressing. The segmented approach clearly solves many of the problems associated with implementing variable sized named spaces. It is much more open-ended than fullword addressing alone because both the maximum number of objects and the size of each can be open-ended.

The image of arbitrarily large objects can be implemented without altering the address size, by making changes to the operating system and to the compilers. The installation of fullword addressing in addition to segmented addressing would be an option available to IBM.

Segmented addressing would require further changes to the System/370 hardware and software to support concurrent addressing between multiple virtual memories. It would require new base register manipulation instructions and linkage facilities. It is not clear how these changes would affect performance.

TRADEOFFS

Segmented addressing clearly provides a more open-ended solution than fullword addressing. Yet, it is not clear whether it can be implemented within the 1980-1985 limit. Fullword addressing would be an interim step towards segmented addressing, but it cannot be considered anything more than a midterm solution. When application developers start accessing data in memory, the requirements for addressing space will increase dramatically, and soon a virtual memory of 2^{32} will not be large enough. Consider the fact that only five years after the introduction of MVS with its "unlimited" addressing capacity, it is obvious that 16 megabytes of addressing is inadequate.

In 1974^2 and 1975^3 the users of large virtual memory systems stated that 2^{24} bytes of address space was inadequate. The consensus was that 2^{45} bytes would be adequate for about ten years. The users conceived the need for 2^{48} or 2^{60} bytes after that. As more complex applications are undertaken, the user's requirement is growing by more than one bit per year. Users also estimated that the maximum number of objects that any single user might want to address simultaneously was approximately 2^{10} (although many of the objects might be about 64 kilobytes or smaller in size). Thus any hardware architecture which does not include the image of a variable size segment should support a minimum of 2^{45} bytes of addressing. Two bounds are placed upon the minimum size of a virtual memory: the maximum contiguously addressable unit and the number of separately protectable units.

EXTENDED ADDRESSING REQUIREMENTS

The extensions to current virtual memory facilities must have the following properties.

1. User-perceived virtual memory must be substantially larger than it is today.

- 2. Programmers should be able to think of virtual memory as an N-dimensional array of infinite length spaces. Possible interim implementations would include fullword addressing and user-transparent implementations of multiple 16 Megabyte virtual memories.
- 3. If the concept of "sparse" named spaces is supported, a storage savings would be realized.

REFERENCES

- 1. IBM System/370 Principles of Operations, IBM System Products Division, GA22-7000 (August 1976).
- 2. Eugene M.Mohr, "Summary of TSS project meeting with IBM," in White Plains, New York, August 15-16, 1974 (referred to as Westchester II), IBM Research, Yorktown Heights, New York.
- 3. Robin C. Thornton, "Minutes of the TSS project meeting with IBM," White Plains, New York, May 5-6, 19¹⁵ (referred to as Westchester III). Chevron Oil Field Research Company, La Habra, California.

Device Independent I/O

The most complex part of programming a computer in MVS or VM/370 involves input and output.¹ The user must choose the correct storage media (e.g., disk, tape, or MSS), the correct model of that media (e.g., 3330 or 3350; 7-track or 9-track), the correct format for the data (e.g., block size or partitioned) and how much space will be required (e.g., cylinders, blocks or tape volumes). The choices are almost endless. Knowing what to store is only the beginning. Determining how to store the information is difficult. In comparison to I/O, the manipulation of data in virtual memory is simple and straightforward.

The most effective improvement to the application developer's productivity is to simplify I/O so that the programmer is concerned with the logical structure of data only. Programmers spend hours reading access method and device characteristic manuals to translate logical data structures into the physical structures required by the hardware. Some would argue that all of the flexibility provided in existing I/O schemes permits the programmer can make. However, application developers are currently overwhelmed with options and consequently make choices which are far from optimal. If all of these options resulted in near optimal use of the storage devices, the additional training and development might be justified. A survey of any data center will demonstrate that such is not the case. Something must be done to relieve the application developer of this burden.

Certainly the physical layout of the data cannot be ignored, but it should not be a concern of the application developer. The operating system should be responsible for the physical structure of data, leaving the application developer free to concentrate on the logical structure of the data. This section and the following one discuss input and output considerations. This section focuses on the logical structuring of data. The following section, Hierarchical Storage Manager, discusses how the logical structures should be translated by the operating system into the physical structures required by the hardware. These two sections present an implementation that satisfies the primary goals of the task force with respect to application development. It is important to recognize the goals, and that they can be satisfied. If the goals are accomplished, the specific implementation used is of little importance to the task force.

GOALS OF DEVICE INDEPENDENT I/O

- 1. Application developers should have no use for knowledge of hardware characteristics.
- 2. It should not be possible to bind an application to hardware.
- 3. Any option requested of the user should be expressed in the terms of the application and shall have reasonable defaults. (See the essay On Options in the chapter on Related Technical Material.)
- 4. The application developers should be able to restrict their concerns to the logical structure of data, without being concerned with where or how the data is stored.
- 5. The application developer's view of data should be consistent throughout the computer system. Regardless of where it is physically stored, it should have the same logical structure.

6. Compatibility with existing applications (either programs or data files) must not be lost. (See the essay on Compatibility and Migration Considerations in the chapter on Related Technical Material.)

THE PROBLEM

The current MVS and VM/370 I/O structure is too close to the hardware and too far from the users. This has the following ramifications:

- 1. Applications based on current I/O facilities are difficult to learn. Parameters such as BLKSIZE and RECFM are alien to the way people think about the problems they are solving. This impedes the extension of computing to end users and problem solvers. It also adds to the cost of application development.
- 2. Applications based on the current I/O facilities are often inefficient because of the following:
 - a. Wrong block sizes or file formats
 - b. Wasted space on partially used tracks
 - c. Overallocation to prevent ABENDS
- 3. Hardware-oriented I/O increases the complexity of both applications and operating systems.
- 4. Maintenance of systems and applications is expensive, especially when new I/O hardware is installed. Each unit of software is potentially sensitive to the physical parameters of the old devices.

BENEFITS OF SOLVING THE PROBLEM

More people will be able to learn and use the system effectively. Existing users will get fuller use of the system, and the cost of training will decrease.

Overall I/O efficiency will increase because poorly designed physical file structures will be eliminated. Application and operating system maintenance will be less costly for the user. Operating system development and maintenance will be less costly for IBM as well. The conversion to new devices will be faster and easier, and fewer people will be required. In addition, system security will increase because users will have less access to the hardware-oriented I/O instructions.

PROPOSED SOLUTION

The operating system should provide techniques that will allow the application developer to concentrate on the logical data structure only. High level languages should provide interfaces to these techniques so that high level language programs can ignore the data's physical structure. The developer should be able to reference data in the following manner:

Sequentially

52

Device Independent I/O

- Randomly by logical record
- Randomly by multiple indexes
- By logical groups of records

These techniques should include ways to insert, delete, or modify logical records in the data structure without concern for how the data will be physically restructured on the storage device. A technique should be developed to allow the user (through his program) to force previous updates to a data structure to be committed to physical storage.

Essentially, device independent I/O consists of functions to build and access data structures in virtual memory, and mechanisms to access the physical I/O routines to move the data between virtual memory and permanent storage in a manner that is transparent to the user. These mechanisms must be consistent with other requirements for data access control and data sharing. (See the section Data Access Control in the chapter on Enhanced Programming Environment. Also see the section Sharing in Virtual Memory contained in this chapter.) The following are additional considerations:

- 1. Portability. Although the basic implementation of device independent I/O requires complete transparency of the placement of a file, it is also necessary to export and import files to and from other systems that do not have this feature. It is assumed that the concept of a "volume," which will be eliminated for normal usage of files, may be required for export and import functions.
- 2. Backup and Recovery. Both file owners and computer installation management will need to back up device independent I/O files. This function will probably be done on a file-by-file basis, not on a volume by volume basis.
- 3. Integration. Device independent I/O should be an integral part of the operating system and should be supported by all of its components, such as editors, compilers, and program products. It should be possible to load and execute object code through device independent I/O facilities. As a natural extension of the paging subsystem, named spaces should serve as the basis for device independent I/O. This would gain the integral sharing and protection provided by the named space mechanism with little additional cost. Thus, the user would deal only with the logical records contained within his virtual memory.

IMPLEMENTATION CONSIDERATIONS

The device independent I/O proposal is not at all like the virtual I/O (VIO) mechanism in MVS. Users of VIO are still exposed to record formats, block sizes, track lengths, and other hardware-oriented characteristics. The intent of VIO is to improve performance while maintaining the standard device dependent user interface. By contrast, users of device independent I/O see a simpler interface without record formats, block sizes, etc. Device independent I/O will probably improve performance by eliminating CCW translation. However, improved performance is only a secondary benefit of device independent I/O. The primary benefits will be ease of use, ease of learning, and fewer user errors.

This proposal cannot be satisfied by the present implementation of VSAM, although many aspects of VSAM are consistent with the intent of the proposal. Before VSAM could meet requirements of device independent I/O, the following weaknesses must be eliminated:

VSAM has hardware-related options, such as buffer number and size.

- VSAM is unable to provide storage for all data structures (e.g., load modules).
- VSAM is not not integrated fully within the application programming environment (e.g., not fully supported by TSO).
- Sharing capabilities are incomplete and awkward.
- VSAM is too complicated for an application developer to learn and use easily and effectively.

It is unacceptable to build upon the other classical OS access methods, because that would merely be inserting another layer of software between the developer and the access methods. This would not remove the basic hardware orientation of these access methods.

COMPATABILITY AND MIGRATION

With the introduction of device independent I/O, compatibility and migration considerations are necessary for old programs and old file structures. The following considerations are required:

- An access method simulator is needed to allow old programs to access new data structures.
- Programs using device independent I/O should be able to access data files created by existing access methods.
- Existing access methods should continue to function so that old programs are able to access old file structures.
- Aids should be supplied to assist users in converting old file structures to the new structures.

The task force recognizes the resource considerations for IBM with respect to maintenance of existing code, in addition to the code required to implement device independent I/O. The task force supports the freezing of enhancements to existing access methods, requiring only that program errors be fixed for some period of time. With respect to performance and cost, a penalty for using old access methods or file structures is reasonable. Ease of development coupled with improved performance and reduced cost would serve as an impetus to use device independent I/O.

RELATIVE WEAKNESSES

The major shortcoming of this approach is that removal of user options will be perceived as decreasing efficiency. The task force recognizes that this argument will be made, but finds the position to be unconvincing. Present access methods are flexible, yet inefficiencies are common. Ease of use is a major concern, and it is the primary goal of these proposals.

REFERENCES

1. Earl Chrysler, "Some Basic Determinants of Computer Programming Productivity," Communications of the ACM, Vol. 21 No. 6 (June 1978), pp. 472-483.

Hierarchical Storage Manager

Areas in current operating systems that require improvement are the input/output interfaces between the user and the system and the interfaces between the system and the external devices. As discussed in the previous section, device independent I/O concentrated on the interfaces between the user and the system. A hierarchical storage manager (HSM) manages the interface between the system and the external devices. (In this discussion, HSM should not be confused with an IBM program product, Hierarchical Storage Management. Although the IBM program product has many of the same concepts, it is not integrated into the system and is not comprehensive enough in scope.) HSM is a single physical I/O manager for all components of the system, such as named spaces, files, and data base management subsystems.

PROBLEM

The current MVS and VM/370 I/O structure is too close to the hardware and too far from the users. Users must spend a great deal of time managing physical space; for example, allocating, compressing, and copying files from one physical device to another.

REQUIREMENTS

Given device independent I/O to manage logical storage, there is a need for a physical storage manager, or hierarchical storage manager that supports both external and internal storage constructs with a single consistent set of operations. The interface between logical and physical storage management must support a clear separation of responsibility to permit smooth and rapid evolution. The physical storage manager should not require any knowledge of the internal structure of the data in the storage it manages. Similarly, the logical storage managers should not have knowledge of the currently employed physical storage technology.

This requires that the HSM build upon a paging implementation of I/O that allows the efficient implementation of storage hierarchies and that supports all components of the system. The task force strongly supports HSM as an integral part of the system.

EFFICIENT HIERARCHY MANAGEMENT THROUGH PAGING

If all data is stored in a page format, then the positioning of any individual data element at any given time is of no interest to the user. Parts of files may exist on a tape or MSS cartridge, while other parts of the same files may exist on the MSS staging disks, on drums, in bubble memories, in main memory, and in cache memory. The user does not have to know the file location.

In any reasonable implementation of a hierarchical storage system, the lowest (slowest, cheapest) level of the hierarchy should exist in a nearly infinite quantity. Two examples of this lowest level might include dismounted magnetic tapes and dismounted MSS cartridges. HSM, not the user, should be responsible for deciding to move the data between levels of the hierarchy.

The system need not be monolithic. For example, the part of he system which maintains the available space in the cache is part of the processor complex. It initiates data transfers to

and from the cache based upon least recently used algorithms in 64-byte blocks. The system control program has a provision to back up available main memory by moving infrequently used data to an auxiliary memory. The paging subsystem moves units of 4096 bytes. This component can move blocks from a high-speed auxiliary device such as a drum to a lowerspeed auxiliary device such as a disk. The system component that moves data between virtual disk packs and MSS cartridges is a part of the mass storage subsystem, and it moves eight cylinder blocks at a time. Thus, even though there are three separate components processing the data with three different block sizes and three different algorithms, the physical organization of the data units at any level of the hierarchy is hidden from the user programs, from the data, and from the other components of the storage hierarchy. The data can be thought of as a string of bytes.

CONFIGURATION FLEXIBILITY

When new levels become technologically and economically justified, they can be introduced in the hierarchy without affecting any user programs or data.

Since the hierarchical storage manager will move infrequently used data to lower levels of the hierarchy, users do not have to concern themselves with the amount of storage available to accomplish any given task. This is analogous to the implementation of memory management with the introduction of virtual memory. Prior to virtual memory, programs were confined to partitions of predetermined sizes, based upon the availability of main memory. This situation no longer exists, and as a result applications are easier to develop.

Adequate storage will be needed at intermediate levels of the hierarchy. With a hierarchical storage manager, performance will be traded off against the entire installation's storage capacity. The installation will still be able to run any job on a smaller system at reduced performance levels. When performance seems to be inadequate, it should be easy for an installation to determine which level of the hierarchy should be expanded.

An individual installation may choose to omit certain portions of the hierarchy. A precedent is currently being set as many installations are running small VM/370 and MVS systems without a high speed backing store (fixed head disk). The lack of a particular component at one level need not be a major performance problem. It may be compensated for by adding additional storage at a higher level of the hierarchy. It is important that all applications continue to run, regardless of the storage configuration. The only considerations in configuration planning should be reliability, performance and costs.

LOCALITY OF REFERENCE SCHEMES

A storage hierarchy is used to achieve the image of high performance storage with the cost of low performance storage. The image it projects depends upon the locality of reference to data. If data is referenced within a small area, then performance will be acceptable. If this locality of reference is not present, performance approximates the lowest level of the hierarchy rather than the highest. (This discussion about locality of reference is based on communications with Jim McIlvain of IBM, General Products Division, San Jose, California.)

Fortunately most programs exhibit locality of reference and thus have acceptable performance. Some programs do not, and ways must be found to ensure that they run satisfactorily without involving application developers.

Hierarchical Storage Manager

At least three strategies are implemented within the existing hierarchy: least recently used (LRU), sequential prefetch, and alternative prefetch. The LRU strategy is used extensively throughout the system for demand access, and it works well for data with locality of reference. The sequential prefetch is used extensively both in the CPU for operand and instruction accessing and in the file systems to preread successive records in sequential files. The third strategy is used only within the CPU. Even though the instruction fetching unit knows only one branch will be used, the unit will frequently prefetch instructions and data for multiple paths after a branch or sequence of branches. The user is unable to tell the system the path of the branch.

This heuristic approach provides a key to the way to make storage hierarchies work with programs with poor locality of reference. Various hierarchical levels may use multiple strategies in an attempt to provide effective service. For example, when a user accesses one block of data, the system might prefetch the following block of data. If that block was used, then the following would be prefetched. If the successor was not used, another strategy might be attempted. If part of the file is referenced, the entire file might be moved one level up in the hierarchy, or perhaps simply prepared for the possibility of such a transfer.

Although this approach requires more system resources, it removes another constraint from application developers. The tradeoff is clear. People costs are saved as the system takes complete responsibility for a function which people do today.

USER PERSPECTIVE ON PERFORMANCE

Some installations may require a mechanism which enables a user to indicate that a piece of data should be favored in the hierarchy for performance reasons. This might occur in applications where specific transactions are relatively infrequent but quick response is required. This is common in real time or process control applications. Normally, data would trickle down to lower-cost, lower-performance devices. The user should be able to specify that fast response is required or that the data is important, and the system should bias that particular data element when migration occurs.

Hardware-oriented performance options should not be provided. The user should not be allowed to specify that a file be put on the fixed head area of the 3350 or to fix the data in any physical location. The user should not be able to fix pages in main memory. The user is not required to know about fixed heads, main memory, or spinning devices. This type of interface complicates both application and operating system design and is counterproductive.

For example, it might be appropriate to have a particular file on the fixed head disk during the day when transaction processing occurs, but if that file is not used at night, the high performance storage is lost to the system. If the system is allowed to manage space itself, it could leave the file on fixed head disk during the day but notice that it is not used at all during the night shifts and move it to a slower device during that time, even though the data was specified to be performance-critical. Specifying the importance of data can complicate the application development process. In addition, there is a tendency for programmers to assign a high priority to all data. Other operating systems which implement transparent storage hierarchies (for example, TSS and MULTICS), are able to run without this performance option facility. Perhaps it is only the historical emphasis of IBM systems on static, fixed allocation which seems to make file space management a requirement. If the system is allowed to arrange data according to use, the objective should be to do an adequate job for most of the users most of the time. The remaining cases might be handled by system programmers on a special case basis. It may be useful to allow application performance to vary based on the time of day, or on other factors. However, it is felt that the performance criteria of most applications can be met without binding them to particular hardware characteristics.

USER PERSPECTIVE ON DATA INTEGRITY

The ability to specify that a data element is critical and that the system should execute special measures to ensure its integrity is a useful option for the owner. Again, the owner might specify how important the data is, and the system could then keep two, three, or more copies of the data on different devices at each level of the hierarchy. If one copy of the data were damaged, the system would automatically fetch one of the other copies without affecting the application program. This duplexing would be expensive, but, unlike the performance option, it would be self-correcting. If the data had a high integrity requirement, the program might run somewhat slower, requiring the user to pay for more storage at each hierarchical level. If the system is reliable, few users will need to request high integrity. If the system is unreliable, all users would need high integrity. This option could also communicate to the system what backup requirements are needed for a data element. Most data could probably exist in a single copy with occasional backup.

BENEFITS

A consistent physical I/O management system (HSM) can provide the following benefits:

- 1. Users will no longer be concerned with space management.
- 2. HSM will reduce or eliminate the need for current space management options.
- 3. Space will be used more efficiently with the system managing it dynamically.
- 4. HSM will allow IBM customers to adopt new hardware technologies without requiring major changes to existing applications.

ENHANCED PROGRAMMING ENVIRONMENT

The preceding chapter, Data in Memory, describes an enhancement to the foundation of the computer system that can simplify the application development process. However, the end users will not directly use named spaces or device independent I/O. The computer system must provide ways for the end user to employ those facilities without knowing about the internal design of the computer system. The bridge between what the end user wants and how the computer system accomplishes it is vital in determining how the end user views the computer system. To understand this, it is necessary to review the application development process to determine its effects on today's end users. The application development process consists of the following three phases:

- 1. Conceptual. The application developers agree on the statement of the problem to be solved, requirements of a successful solution, and the algorithms to be employed in the solution.
- 2. Programming. The application developers translate the algorithms into instructions which can be read by the computer system.
- 3. Certification. Through an iterative process, the application developers verify that the algorithms implemented on the computer system correctly solve all elements of the problem.

The computer system is not involved in the conceptual phase, but it is a critical element in phases two and three. These later phases act as a communication bridge between the application developers and the computer system. The computer system should provide tools which make this communication as straightforward as possible. This communication should take place in a way that is natural for the developer. It should employ the same terms and logic used by the developer in the conceptual phase, and it should be logically consistent throughout the computer system. To the extent that the environment allows developers to think in their own terms, it aids the application development process. The following discussion presents an approach which will create a programming environment beneficial to application developers. It will permit the programming and certification phases of the application development process to be performed in the same terminology as the conceptual phase.

The MVS and VM/370 programming environments provide a strong ability to manipulate data once it is in memory. Engineers have little trouble using the data manipulation statements of FORTRAN or PL/I. Financial analysts can use COBOL or PL/I statements with ease. However, everyone has difficulty with the hardware-related concepts of data access, which include the internal and external data descriptions, job control and interactive command languages, debugging tools and error messages, and data access controls. These areas make application development a job which requires specialized skills. These are areas where future computer systems can have the greatest impact in simplifying the application development process.

Future systems should remove the hardware dependent concepts from the application developer's tasks. The new systems should allow data to be structured logically in the terms of the problem to be solved. A computer system which will translate logical structure into physical structure without involving the developer will reduce the application development time, increase the certification accuracy, and ease maintenance. The techniques provided to perform these translations should be consistent throughout the entire computer system.

Forcing the user to switch from one translator to another is acceptable only if the new translators give the user a consistent view of the computer system.

The computer system required to provide this new environment will be far more complex and more expensive than current systems. This initial cost will be justified if a greater savings will be realized through increased programmer productivity. The greatest potential in business today for increasing data processing growth lies in increasing programmer productivity.¹

The preceding chapter describes several concepts which provide a foundation for simplifying the translation between logical and physical data structures. This chapter describes concepts which build on this foundation to significantly enhance the programming environment. These concepts include:

- Integrated command system
- Data access control
- Subsystem protection mechanism
- Execution and testing environment

REFERENCES

1. Edith Holmes, "Technology or Applications More Crucial to Growth?" Computerworld, September 19, 1977, p.1.

Integrated Command System

The command system facilities impede rather than aid the application development process, if the user perceives that the programming environment is inconsistent and fragmented. The goal of an integrated command system is to reduce the length of the development cycle by increasing the accuracy and speed of the communication between the developer and the system.

THE PROBLEM

The concepts underlying batch job control language (JCL), terminal command languages, the structure of batch jobs, and the structure of terminal sessions appear not to have been examined since the mid-1960s. When IBM customers requested that IBM develop interactive computing facilities to help reduce the development cycle time for applications, IBM introduced the Time Sharing Option (TSO). TSO was not represented as a complete interactive facility, but rather, as the name implies, as an option for OS/360. With the introduction of MVS/TSO, TSO CLISTS, and background TSO, TSO users acquired more efficiency and control over their sessions. Added to these improvements were the Structured Programming Facility (SPF) and Session Manager, which increased the developer's productivity at the terminal.

However, these new facilities were added to the existing TSO structure independently of each other. Since the facilities were not integrated with TSO, many inconsistencies and redundant functions were introduced into the programming environment. Some of the new facilities allowed the programmer to enter commands to the system in a simplified manner, but these facilities did not reduce the number of required concepts that the programmer had to understand.

The menu-driven SPF facility requires programmers to understand MVS data management concepts to answer questions on the SPF menu panels. Programmers must understand catalogs and data set naming conventions, the organization of partitioned datasets, and the use of data set members. SPF does not compile or link edit programs for users. Application developers must still compose JCL statements and use SPF to submit statements for batch processing. Programmers are still required to understand how MVS compiles and executes programs.

The Conversational Monitor System (CMS), the command system for VM/370, also has facilities which are not integrated into the total command system. Terminal jobs cannot easily be run in batch mode. Three editors are available for 3270 terminals, all with similiar, but slightly different subcommands. Worse yet, several subcommands have completely different meanings in the different CMS editors. There are different problems in CMS than in TSO, but the lack of integration has also introduced inconsistencies into the CMS programming environment.

Integration of command system concepts is needed to consolidate functions, remove redundant function, and eliminate inconsistencies in function. Integration is also required to create a single interface to the system facilities, data access control, and the execution and testing tools.¹

Unless command system concepts are integrated, the simplification cannot be extended throughout all command system facilities. Simplification is needed to eliminate programmer involvement in areas for which the system could and should be responsible. Simplification will increase system responsibility in assisting the programmer with work, but simplification will also reduce the number of programmer tasks.

Integration and simplification of the command systems will allow more time for the programmer to develop algorithms, create language statements, and write applications.

REQUIREMENTS

The requirements for implementing an integrated and simplified command system are the following:

- 1. A human oriented, consistent environment should be provided in which developers and end users can easily communicate with the computer system to solve application problems.
- 2. A single interface should be provided to the computer system. This interface should be integrated with and include all the data access controls and execution and testing tools. This interface should be available from terminals and from batch.
- 3. Command system concepts should be independent of operating system hardware and software.
- 4. A single layer command system environment should be provided. All commands should exist and be accessible at all times during the terminal or batch job. All commands should be accessible from higher level language programs.
- 5. Symbolic debugging facilities should be integrated into the system in such a way that the capability is available for all customer written programs at any time.
- 6. Simple and easy to use commands should be provided. Users should be able to estimate the proper specification of a command without having to refer to documentation.
- 7. The distinction between user-written programs and commands and system-provided commands should be eliminated. The commands should be consistent in execution, syntax, parameter passing prompting, and message formats.
- 8. Simple system messages should be provided which can be easily understood by the user.
- 9. Central facilities should be provided for handling system messages and the means to tailor system messages for a single user. It should be possible to increase or to replace system messages for one or all users. Users should be able to control the detail provided by the system message and to request additional explanation.
- 10. Sufficient documentation should be provided in online HELP files or online manuals to aid the developer.

PROPOSED SOLUTION

It is essential to establish a single system interface for users to communicate with the operating system. This single interface would be used to execute system commands and application programs, and to communicate with any data management functions. This interface must be integrated with the data access control facilities and execution and testing tools which are described in the following sections.

The command system should not be considered a multilayered set of commands in which some commands are used only in the data management layer, some for the program debugging

Integrated Command System

layer, some from the terminal, and some from batch. All commands should exist and be accessible at all times during the job or terminal sessions. The commands should be accessible from a higher level languages through a special CALL (or other) function. There should be no distinction between developer-written commands and system-provided commands. All commands should be executed in the same manner from the terminal or from a batch job. The same format should be used in the ordering of verbs and nouns, as well as for screen paging routines, prompting routines, and message formats.

The command system should be integrated with a system-supplied symbolic debugging facility. It must not be necessary to know when a program will break in order to invoke a debugging capability before running the program. All language processors must produce symbolic debugging information which must be available on demand by the application developer. The debugging commands must have a common syntax across all languages.

The command system should also be able to be tailored for a particular user. Tailoring would include renaming system commands for a single user, abbreviating commands, and allowing a user to specify default values for command parameters. System messages could be contained in one file which could be augmented or even replaced for a single user. This would allow users in foreign countries to receive messages in their own language. Users should also be able to control different levels of detail of system messages. The tailoring should be an integral part of the command system and should not create any additional overhead.

REFERENCES

1. Ben Shneiderman, "Human Factors Experiments in Designing Interactive Systems," Computer, Vol. 12 No. 12 (December 1979), pp. 9-19.

Data Access Control

As hardware speed has increased and online storage costs have fallen, the volume of data stored in machine-readable form has risen dramatically.¹ This data often represents much of the most vital financial and technical data which a company possesses. More attention is being paid to the security of this data, who can disclose it, modify it, or destroy it. This attention has resulted in the development of data access controls to help company management answer these questions. Access controls have been appended to operating systems, rather than being integrated into their design from the beginning. As additions, these controls contain inconsistencies in their implementation which end users find confusing. This makes the controls expensive and counterproductive.

Expiration date and OS password protection in MVS, and password protection in VM/370 are particularly cumbersome for the end user who is trying to share data. In MVS, the Resource Access Control Facility (RACF) is better than these methods, but still has severe problems. Let us look at data access control from the end user's point of view to see why the data protection facilities available are not being used today.

End users consider their data files (DASD, tape or MSS) to be under their control, much as the notepads on their desks. They can write anything they want, and it is up to them to control access to the data. They assume that no one else has access unless they allow it. The possibility that someone could access their data in ways undesirable to them is seldom considered until it actually happens. End users do not consider it necessary to have to take positive steps to protect their data.

The existing data set structure using data set names and catalog entries to locate data sets was developed at a time when data security wasn't a very big problem. Online storage was costly and was too limited in capacity to permit the storage of masses of data in a form that was readily available for shared processing. Multiprogramming was in its infancy and time sharing was just being considered for IBM systems. System people (both inside and outside of IBM) were kept busy just keeping the system running. Production schedules were more important than data security.

Clearly, much of that has changed. Online storage costs per byte are a fraction of what they were ten years ago and are dropping at an incredible rate.² It is economically justifiable to put vast amounts of data online (via DASD or MSS) so that new applications can be developed that were inconceivable a few years ago. However, now data security is a problem concerning upper management because a company's future is tied to the availability, correctness, and confidentiality of that data.

THE PROBLEM

In general, any one can see or even modify any unprotected OS data set. It is difficult to establish (and almost impossible to enforce) ownership of data sets. Lack of adequate security is becoming a liability as more sensitive business systems are implemented on computers.

Lack of adequate privacy safeguards is making it difficult to meet the requirements of current and anticipated legislation. Passwords, as implemented under OS or even TSO, are inadequate because:

They do not establish ownership.

Data Access Control

- They restrict the owner as well as all others.
- They frequently appear in written form.

VM/370 approaches data sharing in a way much different from MVS. The ownership of a minidisk is easy to determine, but if multiple users are adding data to a minidisk, the identity of the creator is lost. The three levels of minidisk passwords have several problems associated with them:

- They are difficult for the minidisk owner to change.
- They must frequently appear in written form.
 - They do not provide the ability to differentiate between various files within the minidisk.

The IBM program product, Resource Access Control Facility, (RACF) has taken a reasonable conceptual approach to the data protection problem by assigning verifiable USERIDs to all users and assigning the responsibility for data protection to the creator of the data set. Yet RACF has met with anything but overwhelming acceptance from the user community. Is this because data access control is not a problem for users, or is it because RACF has features or deficiencies that make it in some way an unacceptable solution? Consider what the end user sees when he tries to use RACF to protect his data set, a manual that runs over 200 pages containing many different commands. He has an essentially simple problem, a data set with a known list of allowable users. He has been given a very complex solution and he usually gives up. What does the system programmer see when he looks at RACF's impact on the operating system for which he is responsible? He sees a RACF data set that might be accessed each time a RACF protected data set is accessed.

The system performance and reliability implications are great, especially for large multiprocessing installations which choose to protect all of their data. Should the RACF data set be split over multiple devices? If so, how should it be split? Should the RACF data set be duplexed? These are questions which can have far-reaching effects on the reliability and performance of the computer system, and which are far too easy to answer incorrectly.

Moreover, the protection that is offered by RACF isn't sufficiently adaptable. Protective safeguards should include the following:

- The ability to execute a load module should not give a user the ability to copy all of the load modules in the data set into his own library.
- The ability to extend a dataset should not give a user the ability to alter existing data within the data set or to delete it entirely.
- The ability to update existing data should not allow a user to delete the entire data set.

The problem can be summarized in this way; at present, data access control is something the user must choose to add on. In the future, it is something that should be an integral part of the system and be easy to use. RACF, as currently implemented, is inadequate because it is difficult for noncomputer people to learn and to use.

BENEFITS OF SOLVING THE PROBLEM

Improved data access control facilities have one obvious benefit: better data security. That is a benefit that has significant ramifications for large system development. With increasing frequency, new applications are being postponed or cancelled for data security considerations. Many applications are being moved onto minis for the same reason. Improved data access control is necessary for the continued growth of large scale system development.

Increased security in the system will help users meet legal requirements and will enhance the auditability of systems. This will be an important feature to meet many of the needs of data stewardship.

An integrated and natural security mechanism will make security a part of all transactions with the computing system, in the same way that security is an implicit part of all transactions with a typical manual office file system. This will increase the ability and incentive to automate more business functions.

PROPOSED SOLUTION

In the area of data access control as applied to the other proposals in the enhanced programming environment, the task force recommends no specific implementation. The requirements can be most readily met if security is an integrated part of the file system, not an afterthought.

All of the existing schemes are limited by various hardware considerations. The data in memory concepts relate to data access control. Awkward implementations such as the RACF data set could be avoided by device independent I/O. Access control could be made easier for the end user by providing useful defaults that could be set for new data sets so that access control profiles would be altered only in exceptional cases.

The SHARE MVS security project is currently developing detailed and comprehensive requirements in this area. In general, the task force will subscribe to their requirements and support them. Additionally, the task force submits the following points:

- Security is related to device independent I/O because many of the most stubborn security exposures come from users exploiting the power of the I/O system to manipulate the hardware directly.
- Security is related to the integrated command system because the system must provide for identification and enforcement of ownership and access privileges in a natural way.
- Security is related to named spaces because the system must provide sharing and protection mechanisms for data and programs read into virtual memory.
- Security must be a normal, natural part of all transactions with the computer.

In the area of data access control as applied to protecting data and programs read into virtual memory, the reader is referred to the task force proposals in the following section on the Subsystem Protection Mechanism.

ACCESS CONTROL FACILITY REQUIREMENTS

The requirements of named spaces and device independent I/O must be implemented in such a way that an access control facility is included. This access control facility may or may not be an extension of RACF. The purpose of the discussion here is not to give a complete, rigorous description of an access control facility or to criticize RACF, but rather to present the basic elements of the facility which must be provided.
Data Access Control

The user's ability to share his data is essential. This is a major feature of the proposals for named spaces and device independent I/O. For this discussion, programs are just one kind of data. Any implementation of named spaces and device independent I/O must contain a usable, consistent, central access control facility with the following properties as a minimum:

1. Personal Accountability. A meaningful access control facility must insure that users of the system know that they are uniquely and verifiably identified to the operating system, that their requests for computing resources can always be associated with them, and that records of abuses or attempted abuses can and will be kept. The RACF scheme of USERIDs and logon passwords is an adequate implementation of unique and verifiable identification. The MVS integrity commitment satisfies the need that all requests be associated with their requester. The recording of the activities through SMF has some serious usability problems as it exists today, particularly with regard to the lack of any reporting capability. An access control facility that does not include a basic reporting capability with formatted reports is incomplete.

2. Entity Ownership. Each entity within the computer system which can contain data (programs included) must have a unique owner who is responsible for that data; not just for creating it, but also for governing the authorization of others to use it. RACF presently provides this capability for tape volumes and for DASD data sets. The capability must also apply to data contained in named spaces, and must apply no matter where a named space resides, on mass storage, on online DASD, or in virtual storage.

- 3. Entity Access Authorization. Since an entity owner is responsible for control of access to the entity, it is required that the owner be given the means to positively control that access. Where management dictates that others possess the capability to alter access authorizations, records of must be kept of modifications initiated by those other than the owner. It should be kept in mind that control over access authorizations is the area where most users will interact most frequently with the access control facility. Therefore, this portion of the facility must be both easy to understand and easy to use. Users should be able to create default access authorization lists so that repetitious modification of authorization lists in one step, rather than one entity at a time. Most important, the documentation supplied to end users must be limited to the functions which are necessary and useful to them:
 - a. Controlling the use of the entities for which they are responsible
 - b. Controlling some of the information stored about each user (e.g., logon password, name, phone number and location)
 - c. Inquiring as to their own access authorization and the owner's identity for an entity which they do not own

It is essential that capabilities required only by systems programmers or security administrators not be described in the documentation provided to the end user; it causes confusion without providing benefit.

4. Type of Control. Read and Write control is not sufficient, especially when Write access implies Read access. At least the following types of access must be differentiated by the control mechanism:

a. Read

Enhanced Programming Environment

- b. Execute (but not Read)
- c. Update (but not Extend)
- d. Extend (but not Update)
- e. Write (both Update and Extend)
- f. Rename (and recatalog)
- g. Delete (and uncatalog)
- 5. Homogeneity. These controls must be available for any entity, regardless of where it is stored. Thus, the access authorizations for a named space accompany it as it moves from mass storage to disk to virtual storage, so that an authorized user could access the data at any time, regardless of where it resides, except as required by data integrity considerations.
- 6. Protected Entry Points. The owner of executable code must be able to be certain that entry to that code can be made only at defined entry points. Verification of this should be a function of the access control facility.
- 7. Access Vehicle Control. The owner of data is often concerned with more than the type of access to his data (e.g., Read, Execute, or Update). The owner may want data to be accessed only through approved vehicles. He may want to allow others to update his data only via one or more programs that reside in a library which he controls. The access control facility should provide this capability. It should be pointed out that the existing Authorized Program Facility in MVS is not satisfactory, because it extends the privileges of authorized status to programs which do not require it.

It is understood that many of the features described here would require support from outside of the access control facility. This demonstrates why it is important that the access control facility must be a central and ever-present part of the operating system, not an optional add-on.

NOTES AND REFERENCES

1. Charles P. Lecht, Waves of Change, New York, 1977, pp. 1-34.

2. Ibid, p. 17.

68

Subsystem Protection Mechanism

Currently, it is common for programs to be shared. In fact, the entire software industry relies on that premise. With named spaces that support dynamic sharing, individual programs will be simultaneously part of many application systems in many different organizations. (See the section on Sharing in Virtual Memory in the chapter on Data in Memory.)

THE PROBLEM

There is no means of protection for a program that is to be shared. Once the program is loaded into virtual memory as part of an application system, the program is exposed to all other programs in the application system. If the program must access proprietary data, all other parts of the application obtain access to that data.

Current programs requiring protection must not share memory with other programs. This restriction is antithetical to the whole concept of shared named spaces. If named spaces were shared in virtual memory without safeguards, there would be no way to protect sensitive, proprietary, or copyrighted programs. The problem can be summarized as follows:

- How can the owner of a program allow others to use it without also letting them copy it, modify it, or steal it?
- How can an application call an outside program, such as a sort program, without providing the program access to all the data that the application is processing?
- How can individual programs in an application system be restrained from interfering with one another, whether erroneously or maliciously?

BENEFITS OF SOLVING THE PROBLEM

The benefits of solving the problem include the following:

- Protection is provided for proprietary programs and data.
- Applications are more robust if they can be divided into separately protected subsystems.
- Greater protection is provided for currently available subsystems from within and without the subsystem.

PROPOSED SOLUTION

The new facility required to implement subsystem protection is a slight variation of the "Connect" operator described previously. (See the sections on Named Spaces and Sharing in Virtual Memory in the chapter on Data in Memory.) It would allow a program (possibly copyrighted) to be attached outside the requesting user's address space. A set of pages (usually one) could be shared between the two address spaces for parameter passing. The new routine could be called by an SVC, or perhaps, a new linkage instruction. It would execute separately from its caller. This implementation is a generalization of the current IMS linkage structure.

With this facility, proprietary algorithms and confidential data can exist in separate virtual memories and can be fully protected from tampering.

REQUIREMENTS

- 1. A facility is required to allow application programs to be subdivided into portions which can interact with each other through prescribed interfaces only.
- 2. This facility must ensure that protected programs are entered only at designated entry points.
- 3. This facility must ensure that protected programs may not inadvertently or deliberately interfere with each other, or have access to each other's instructions or data. This requirement could be met with an "execute-only" access mode, if the implementation could also protect the data of the execute-only program.
- 4. Subsystem protection must be both integral and consistent with any other data access control mechanism in the system. (See the section on Data Access Control in this chapter.)

TASK FORCE COMMENTARY

Subsystem protection is a special element of data access control. The task force recommends an "execute-only" access mode which will ensure that objects will be protected while they are shared within a virtual memory. Subsystem protection is described separately here because the problem it addresses is unlike classical access-control problems. However, with named spaces as a primitive construct, the solution to subsystem protection will be the same as the solution to any other access control requirement.

Execution and Testing Environment

The application development process contains a phase where the developer is going through the iterative process of executing, testing, evaluating and revising the application. During this process the developer is continually modifying the source programs and is continually invoking system utilities and routines to aid the execution and testing of the application. Application developers face severe problems in performing these conceptually simple tasks.

To execute a program, a developer must load or link edit it first. To test it, the developer must invoke a debugging environment such as TSO Test or CMS Debug. Debugging commands may be entered using hexadecimal offsets, absolute virtual addresses, or general register notation. These concepts are closer to the hardware than to the user's problem.

In addition to being hardware-oriented and difficult to use,¹ current execution and testing tools are geared for a static, not dynamic, environment. In the static environment, it is assumed that all programs to be included in the application are known by name, that the programs or shells of the programs have been coded to satisfy resolution of external references, and that overlay techniques have been developed to fit the large application into virtual memory. In a dynamic environment this information may not be known.

A static environment also does not permit the sharing of individual programs within the application package, but instead requires duplication of programs for each copy of the application load module. Tools designed for the static environment demand that the application be planned, designed, and sized prior to entering the iterative process of executing, testing, and revising.

The environment in program development organizations is extremely dynamic. Changes are frequent and affect many components of the application being developed. Application packages are using data directed linkages, which cause changes to program flow and result in different programs being invoked. Application packages are open-ended, and programs are frequently added for new function or replaced for improved function.² Application packages are being written in ways to allow them to be responsive and flexible to requests for changes from users of the application, by company management, or through government regulations.

The dynamic, creative atmosphere of development areas has started to produce application packages larger than anything considered possible ten years ago. It is not uncommon to have hundreds of individual programs in a single application package. Several installations have applications containing over 2000 programs.³ Current operating systems are evidence of this trend, and their development over the last few years has paral eled the development of large applications. The execution and testing tools are complex and hinder the development of large applications and systems.

THE PROBLEM

The application developer should be responsible only for developing and debugging algorithms and language statements. The system functions such as the linkage editor and loader, which the developer must invoke to convert programs into forms which the system can execute, are overhead which could be eliminated with present technology. The available debugging facilities are tied to operating system software and hardware and impede the testing phase of the application development cycle. Furthermore, execution and testing tools available today are not adequate to handle the dynamic, flexible environment required for today's applications.

The MVS linkage editor is an example of a system facility currently required for the execution of programs. It is not adequate because it requires the premature binding of individual programs into a single load module.

Premature binding of programs affects not only the development phase of a project cycle, but also the testing and maintenance phases. This applies to the development of large applications as well as to the development of IBM program products, system control programs, and other IBM system code.

1. Development

In the development phase of a project, the requirement to prematurely bind programs causes extensive programmer time to be spent preplanning all possible components of the program package, planning overlay structures, and dealing with current operating system inabilities to handle large packages of programs. Loss of flexibility during the creation phase causes loss of programmer productivity and even loss of programmer creativity.⁴

2. Testing

In the testing phase of a project, premature binding complicates the debugging efforts and increases the time required to test modifications to individual programs. Developers have numerous versions of load modules when they are varying and testing only one small component.

3. Maintenance

In the maintenance phase of a project, premature binding complicates and increases the time spent in applying changes and adding extensions to large packages of programs. It is not uncommon for users to link edit single routines from the package of programs into their own load modules. Maintenance must then be applied to all copies of the routines, if they can be found.

BENEFITS OF SOLVING THE PROBLEM

The benefits of implementing a simple, easy-to-use execution and testing environment are:

- Users are decoupled from the details of operating software and hardware. This will allow application developers to capitalize on new hardware and software architectures.
- The number of computer oriented concepts needed to solve any problem are minimized. This will free programmers to solve application problems rather than computer problems.
- The length of the application development cycle is decreased.

Specific benefits in the development process of large applications or IBM system code:

- The expense of employing link-editor experts is saved. Today there is at least one such expert on every installation's application development staff.
- Application development effort is reduced by simplifying program control and program extension.

Execution and Testing Environment

- Application integrity is improved by better definition of control flow and external data interfaces.
- Application extensibility is automatically provided.
- Validation of components and testing of modifications is reduced and simplified.
- Application maintenance is reduced and simplified by sharing a single copy of the application modules. Copies of individual programs link edited into users' load modules are eliminated.
- The process by which large packages of programs are created and implemented is enhanced.

Specific benefits in the production environment of large applications or IBM products:

- Virtual address space is saved because user address space need not contain programs not referenced during a particular job execution.
- Private virtual address space is saved by selective cross system sharing of a single copy of a loaded program.

EXECUTION AND TESTING REQUIREMENTS

To provide a human-oriented and consistent environment in which users can perform the iterative process of executing, testing, evaluating, and revising their application packages, the following should be considered:

- 1. Developers should be responsible only for designing applications and for developing and debugging algorithms and language statements.
- 2. There should be straightforward, easy-to-use commands to communicate with the system during the iterative process. Commands should reference program names, program variables, or source statements only.
- 3. It should be a system responsibility and a system function to perform the work needed to execute applications. Any conversion or binding of the application programs should be a system function hidden from users.

DYNAMIC LOADING FACILITY

One facility which addresses the lowest level of binding is called the *dynamic loader*. The dynamic loader delays the binding of programs until a program is actually referenced during execution. The dynamic loader should then perform the binding automatically without user involvement. The dynamic loader should have new facilities for aiding the complicated development, test, and maintenance cycles of application development.⁵ The dynamic loader facility requires no new technology and is available on several IBM systems today. It is required as an interim solution for the premature binding problem, but it is *not* the total solution for providing a human-oriented execution and testing environment. The following requirements are offered as guidelines for an implementation of the dynamic loader:

1. Dynamic Loader Load Function

The dynamic loader should be capable of searching a predetermined list of libraries and/or catalogs to resolve a control section reference.

- a. References known when the control section is selected for loading are called implicit references. Loading of additional control sections specified by these references is referred to as implicit loading. Implicit loading should be performed automatically by the dynamic loader.
- b. References not known by name when the control section is selected are called explicit references. Such references may be supplied by macro facilities or by user input to the program. Loading of additional control sections specified by these references is referred to as explicit loading. Explicit loading should be performed only when a CALL statement, LOAD macro, etc. is actually executed. Options on the CALL statement should be provided so that even if the reference is known by name, the reference can be flagged as an explicit reference. Loading is then deferred until the CALL is executed.
- c. The dynamic loader should be capable of resolving references to oth er control sections as it proceeds in the binding process.
- d. The dynamic loader should be capable of resolving a reference to a control section already loaded in the address space. There should be no duplication of control sections in the address space.
- 2. The Dynamic Loader Unload Function

The dynamic loader should provide an unbinding and unloading process, in addition to the binding and loading process described above.

- a. The dynamic loader should provide a means to unload a single object or load module.
- b. If other control sections have been brought into storage because of implicit or explicit references from the section being unloaded, the additional sections should also be unloaded.
- c. The storage occupied by the object or load module should be released.
- d. All other object or load modules brought into storage as part of the binding process should remain in storage.
- e. All references in the modules which remain in storage to the unloaded object or load module(s) should be flagged as unresolved.
- f. The dynamic loader should then be able to be reinvoked to load the control section (perhaps now a different version), and the binding process should again assign storage and resolve references in the control section. All references in those object or load modules which remained in storage throughout this process are now resolved by the dynamic loader to the newly loaded control section.
- 3. User Functions to Control the Dynamic Loader Environment

74

Execution and Testing Environment

- a. Specification of Libraries/Catalogs. There should be an easy-to-use mechanism for specifying the list of libraries or catalogs to be used to resolve references.
 - 1) There should be some easy-to-use mechanism to order the list and to change the order dynamically.
 - 2) This predetermined list should be user-specified, but should have well defined defaults if no list is specified.
 - 3) The dynamic loader should be capable of recognizing and processing both object and load modules.
 - 4) An object or load module may exist in more than one library or catalog. The dynamic loader should select the copy of the module according to the ordering of the list of libraries or catalogs, and ignore additional copies of the same module in other libraries or catalogs.
- b. User Load Command. There should be a user command to direct the system to immediately load a specific program or application package. The loading process is governed by the guidelines specified in the Load function above.
- c. User Unload Command. There should be a user command to direct the system to immediately unload a specific program or application package. The unload process is governed by the guidelines specified in the Unload function above.
- 4. The Dynamic Loader Environment
 - a. Availability. Dynamic loading should be easy to use and should be integrated within the system. Support should be available from IMS, TSO, CMS, batch, high level/low level languages, and so on. The implementation should be consistent across all users. Dynamic loading should be available during the creation of the operating system itself. Building of the system nucleus and the MVS Pageable Link Pack Area could be accomplished at IPL time, using a dynamic loader facility.
 - b. Execution Characteristics. Existing application packages should operate in the dynamic loader environment in both bound and unbound form without modification.
 - c. Debugging Facilities. The dynamic loader should be a system service, available and used by all system functions.
 - d. Mixing Environments. The dynamic loader should be able to load modules linked by the standard linkage editor, as well as output from lamguage compilers.
 - e. Future Considerations. The services of the dynamic loader should provide facilities for both production and development work and should be designed to permit use of future operating system enhancements.
 - f. Replacement of Static Loader. The current static loader may be replaced with the proposed dynamic loader. Applications may be reassembled or recompiled to use dynamic loader facilities.

- 5. Relationship to Current MVS Linkage Editor Environment
 - a. Parallel Maintenance. The current linkage editor must be maintained in parallel with the dynamic loader.
 - b. Compatibility. The dynamic loader environment must be compatible with that of the standard linkage editor and must interface with applications using those facilities.
 - c. MVS Contents Supervisor. The contents supervision component should be capable of interfacing with the dynamic loader and the enhanced paging mechanism to avoid the loading of duplicate modules into the same address space (i.e., it should be capable of resolving a reference to a control section contained within a load module already loaded in the address space).
 - d. Production Application Environment. It will often be desirable to link edit applications developed with the dynamic loader for use in production mode.

6. Integration with other LSRAD Proposals

- a. Design. The dynamic loader facility should be designed to be integrated within the total concept of data in memory and device independent I/O, and should work with and in support of the other system requirements. Upward compatibility should be assured for current programs within the constraint of reassembly or recompile. As the major benefit of the dynamic loader comes during the application development process, downward compatibility is not required.
- b. Mapping. The dynamic loader should assign enough storage to contain the program. Pages should be allocated by the loader, but the contents of the pages should not be transferred from external storage to memory by the loader. The enhanced paging mechanism proposed by the task force would perform this function:
 - 1) Pages of an executing program or named space will be transferred only if they are actually referenced.
 - 2) For a given program, only the references within a page that has been transferred will be resolved by the dynamic loader. Each page can have a bit that indicates whether or not that page has been processed by the dynamic loader (i.e., all external references have been resolved within that page). If that bit indicates that the page has not been processed, the dynamic loader will resolve the references.
- c. Sharing Address Space. The ability to share programs across address spaces is required. The dynamic loader must be able to resolve references to shared programs. Addressing space will be given up only for the programs or groups of programs selected to be shared.
- d. Output from Language Compilers. Language compilers should be modified to produce programs that can be shared. This might involve producing multiple control sections for a program in order to separate the program's variables and address-

Execution and Testing Environment

dependent constants from the reentrant, read-only source text. The dynamic loader should interface with this facility and fully support it.

REFERENCES

- 1. Terry Winograd, "Beyond Programming Languages," Communications of the ACM, Vol. 22 No. 7 (July 1979), pp. 391-401.
- 2. D.A. Fisher, "DoD's Common Programming Language Effort," Computer, Vol. 11 No. 13 (March 1978), pp. 25-33.
- 3. William Garth, "Design Console Technology at General Motors" (manuscript), General Motors Manufacturing Development Technical Center, War en Michigan, July 1974.
- 4. A.J. Conway and D.G. Harvey, "User-System/38 Interface Design Considerations," *IBM* System/38 Technical Developments, IBM General Products Division, 1979, pp. 70-73.
- 5. Peter McCabe, Bell Telephone Laboratories, Naperville, Illinois, "Dynamic Loading," Session B451, SHARE 52, San Francisco, California, March 1979.

RELATED TECHNICAL MATERIAL

The preceding chapters have described a synergistic approach to the problems which are inhibiting programmer productivity. The following chapter contains four essays discussing issues related to the technical proposals just presented. The essays are included to further clarify the task force's approach to the problems described in Part I.

- The first essay discusses some of the costs and methods of a revolutionary approach to change, as opposed to an evolutionary one.
- The second essay defines four levels of compatibility and gives guidelines for introducing an incompatibility at each level.
- The third essay discusses how options should be presented to the user.
- The fourth essay discusses user-perceived limits.

Evolution Versus Revolution

The LSRAD charter specified that the task force consider only evolutionary extensions to MVS and VM/370. At first, this constraint seemed to be a minor one, and the task force directed all of its energies to developing realistic proposals which could be implemented with completely compatibile interfaces. As the technical proposals were developed in more detail, some major compromises were required to preserve compatibility with existing systems. For the most part, the compromises involved greatly increased implementation costs resulting from transitional interfaces, conversion programs, simulators, and so forth.

TRANSITIONAL INTERFACES

For example, extended addressing proposals require a bridge for programs which were compiled to run in a 24-bit addressing environment. It seemed reasonable to restrict existing programs to residency in the lowest part of the expanded virtual memory, but those programs must be able to communicate with new programs not constrained to run in the lowest part of memory. To accomplish this, a new type of subroutine linkage is necessary, but this new linkage must be implemented without modifying existing programs. one solution is to generate skeleton entry points to all extended addressing-environment routines to transform parameter lists and addressing modes as the subroutine is entered, and restore the appropriate mode upon exit. This implementation requires modifications to all of the compilers and to the program fetch mechanisms. The skeleton entry points would have to be loaded in low memory, the new subroutines in high memory. With the new simplified file system, new simulators similar to the ISAM-VSAM bridge and SAM-E are required to support existing programs.

All of the new facilities represent substantial programming investments in essentially throwaway code. When users have converted their applications to the new environment (which may require recompilation only) the transitional code will not be required. Unfortunately, most users will not be able to convert all of their code upon installation, so transitional code seems necessary if a new function is to be added.

THE COSTS OF EVOLUTION

Other evolutionary considerations were even more serious. When MVS was initially introduced, the commitment to security was regarded as one of its most important features. Yet many existing OS programs regularly acquired information through illicit channels. When these channels were closed and new authorization features installed, many existing application programs required reprogramming. This type of "necessary" incompatibility caused many users to be reluctant to use MVS at first. It was not architecturally possible to have both security and compatibility. Fortunately, many installations insisted upon the necessary reprogramming conforming to official interfaces, which will make the transition to future systems much easier. Installations will not have to make changes to their programs as long as IBM continues to support the official interfaces. Programs will have no dependencies on the internal intricacies and data structures of the operating system.

Too much compatibility can be counterproductive. For example, one of the design goals of the VM/370 implementation of attached processor support was to minimize the impact upon existing customer modifications to the VM/370 control program. One can seriously

question whether this should have been a design goal for a project of such scope. This introduction of significant new function into the control program might have provided a convenient occasion for reorganizing or recoding appropriate portions of the control program, perhaps to provide a firm foundation for future enhancements. The minimum impact philosophy tends to produce a less synergistic control program which is more difficult to maintain and enhance in the long run.

The job of operating system designers is to balance the requirements for compatibility with the costs of implementation, maintenance, and architectural compromises. It seems clear that customers do not want 100% compatibility at all levels of the operating system, since that would mean that the entire environment must be frozen as it is to lay. However it is equally clear that compatibility which allows 80% of all existing programs to run is not adequate either. Business planners must work together with customers to determine exactly when the costs of transition to an enhanced environment exceed the perceived benefits of the new environment.

ALTERNATIVES TO EVOLUTION

It is interesting to note how this conflict has been resolved in two recent IBM product announcements.

The 8100 Distributed Computing System was introduced with two new operating systems and two new instruction sets. The evolutionary operating environment is called Distributed Processing Communicating Executive (DPCX). DPCX is a bridge to the 3790 remote computing systems announced in the early 1970s. Customers using 3790 systems can purchase 8100 systems and transfer their applications without significant conversion cost, thus gaining the price/performance advantages and some of the new function of the 8100.

The revolutionary programming environment is called Distributed Processing Programming Executive (DPPX). DPPX is a completely new programming environment which is not compatible with the 3790. It is designed specifically to improve programming effectiveness on newly written applications. In DPPX the programmers are isolated from computer-related items such as hardware, devices, instruction sets, and so forth. The programmers are responsible for algorithms only. This results in much cleaner and simpler applications which can be developed more quickly. However, the productivity associated with DPPX is provided at some cost in performance. IBM suggests that applications be developed under DPCX if performance is the primary consideration. The LSRAD task force believes most new customers will choose DPPX and trade hardware costs for people savings.

The announcement of the System/38 was substantially different. The System/38 is a single new revolutionary system. It was designed without concern for concurrent execution of old and new forms of programs. System/38 includes a comprehensive set of conversion programs which will convert almost any application program and data base on an existing System/3 or 34. The new System/38 operating system appears to satisfy all of the strategic requirements proposed by the task force (consistency, simplicity, and synergy). It is one of the most sophisticated offerings by any manufacturer to date. Application programming is integrated with the data management environment. It appears to be the first true "data base machine".

This sophistication is presented to the users through a well designed, comprehensible interface. All underlying hardware complexities are completely masked from the user's view.

Evolution Versus Revolution

At this time, however, RPG3 is the only high level language available, although others are planned. Assuming additional lanaguage processors become available in the future, the System/38 approach would provide significant productivity benefits for both application developers and for end users.

This complete conversion strategy does not appear to be a viable one for MVS or VM/370. The System/38 supports a limited range of applications compared with MVS or VM/370. Furthermore, the conversion problems are minor compared to those of MVS and VM/370 customers. Potential System/38 users traditionally coded only in high level languages and did not modify IBM-supplied systems. Standard interfaces were respected everywhere.

A recent study by Robert C. Kendall¹ observed the average life of programs to be approximately 18 months. This supports the feasibility of a split environment where short lived programs might be developed cheaply on a revolutionary, high productivity system, while existing, long running applications continue to run on more conventional systems. It might be possible to install this type of system if an interface were available to share the data between the two environments. This interface would provide for sharing of data between new and old programs, but would not provide direct communication between them. It would be more difficult to work with during the transition period, but might have substantially better human factors due to reduced compatibility requirements. It also would require each installation to support both operating environments, i.e., multiple hardware systems, and support staffs. Hypervisors could allow both environments to run on one hardware complex. However, for many customers these support costs may be prohibitive, especially if they must be continued for any extended number of years.

This revolutionary scenario may not be a practical growth alternative for MVS and VM/370 customers. Since the LSRAD charter precluded any proposals that were not evolutionary in nature, this scenario was not investigated to any depth. However, it does offer one possible avenue for the development and marketing of high programmer-productivity, synergistic systems without the enormous costs of complete compatibility with existing systems.

REFERENCES

1. Robert C. Kendall, "Management Perspectives on Programs, Programming and Productivity", White Plains, N.Y., IBM Corporation, CHQ Division, p.12.

81

Compatibility and Migration Considerations

Transition to a new system is a serious matter. It would do little good if all of the LSRAD requirements were implemented in a totally incompatible fashion so that migration from existing systems was impossible. Yet insisting on total compatibility is not the answer, either. Indeed, that is why MVS and VM/370 appear as they do today. Change is important to the future development of more sophisticated data processing capabilities. However, this change is a source of disruption to data processing installations. So the real problem is how to manage change.¹

Most data processing customers recognize that technological advance often requires change to both the hardware and software. When new capabilities are added or old functions replaced, this cannot always be done in a compatible manner.

Customer tolerance for change is based on the ability to absorb the changes while continuing to provide consistent and reliable service to the end users. The end users are generally nondata processing professionals who are interested only in solving their problems. Incompatibilities causing even the smallest amount of disruption or conversion are major problems to the end user. Therefore, general user commands and interfaces should be carried essentially unchanged from release to release. Existing programs must continue to run without the requirement for conversion activity.

As an example of the nature of this disruption, at Northwest Industries the corporate executives have logon commands hard-wired in read-only memories in their terminals. The executives push a button and the terminal dials the computer and logs on. Thus even a change in the telephone number represents a disruption.

The following discussion on managing change covers the following:

- 1. Compatibility of code in the System Control Program (SCP)
- 2. Compatibility at the system programmer level
- 3. Compatibility at the end user level
- 4. Compatibility at the product or component level

SCP COMPATIBILITY

Compatibility of code in the SCP is the extent to which customer modifications will fit into a new release of an SCP. This is the least important level of compatibility because it is an area that end users never see. Customers making modifications to the SCP must be aware of the risks they are taking. Few assumptions should be made about code remaining unchanged in future releases. Many customers go to great lengths to make their changes modular so there is a better chance of fitting them into subsequent releases.

Frequently, an incompatibility in a new release is the result of the system providing a function that was formerly provided by the customer. Generally, the customer will drop his solution in favor of the general one for ease of maintenance. However, this frequently causes disruption to the end users because functions are performed in a slightly different fashion by the system. For those functions which are in widespread use (such as those distributed on the

Compatibility and Migration

SHARE Mods tape), IBM can minimize the disruption by patterning the support after that already in use.

One reason for rewriting existing SCP code is to restructure or streamline code that has been repeatedly modified. Successive versions of MVS and VM/370 have been changed and enhanced from earlier versions. Customers can easily see the need for periodic software clean up to enhance the maintainability, reliability, and performance of the SCP. In general, customers who have modified the SCP and are affected by this type of change consider rewriting their modifications as a necessary for SCP improvements.

COMPATIBILITY TO THE SYSTEM PROGRAMMER

In general, system programmers are more tolerant of incompatibilities than the end users. However, change in a command format (whether an operator or an interactive command) so that the old form no longer works is a serious matter and should be considered carefully. Commands embedded in EXECs or CLISTs create more of a disruption, while changes to commands or macros embedded in assembler code are the worst. It is not as disruptive if the new form can be introduced while a bridge or synonym allows the old form to work. The user may be totally oblivious to the fact that the old form is really mapped into the new form.

END USER COMPATIBILITY

Compatibility at the end user level means that the execution of a command, program, or file operation should produce the same results with the new system as it did with the old one. This aspect of compatibility is a crucial factor in delivering high quality computer service.

IBM considers the end user to be the customer who installs and maintains a data processing system. For most customers this is not the case, since they provide services to another more diverse group of users who are relatively unsophisticated in computer skills. New interactive problem-solving tools such as Query-by-Example are carrying the computer directly into the offices, executive suites and board rooms at all levels of corporate responsibility. The end users are professionals, laborers, managers, secretaries, technicians, and executives, as well as professional programmers.

To these users, even the smallest incompatibility caused by a change to the software beneath their applications can cause serious disruption. Often end users are unable to deal with change without the help of the professional computing staff. These end users, working against their own deadlines to complete their work, are difficult to convince that any incompatibility is necessary. Continuity and stability are highly important factors in the way these users perceive the quality of service from their computer system. In fact, any incompatibility appears to be a bug to the user who did not hear about it in advance. Data center newsletters may not reach all users concerned.

The incompatibilities seen by these users cause the most disruption to the installation, which may go to great lengths to avoid or minimize them. This is not to say that incompatibilities cannot occur; only that they must be very carefully considered and planned. Disruption of the end user must be minimized.

It should also be noted that one of the most important elements in avoiding incompatibilities in future releases is to see that new facilities are of sound design.

COMPATIBILITY AT THE PRODUCT OR COMPONENT LEVEL

Compatibility at the product or component level requires maintanence of component modularity so that interfaces between the SCP and the major subsystems remain unchanged from release to release. It also requires minimization of the impact of changes that involve the format of any records on external storage. Clearly, device independent I/O and logical (as opposed to physical) records are a major step forward in minimizing this impact.

It is frequently impossible to cut over completely to a new release of an SCP with all related components in the same day. For example, when migrating to a new release of VM, installations will frequently bring up the new CMS first in a development mode under the old control program (CP). Then the new CP is brought up on the real machine. Eventually, the users are migrated from the old CMS to the new one. This may take some time. Clearly it is imperative that the old and new versions of CMS be able to run under both versions of CP. If new CP interfaces are needed for the new CMS, they should be provided in a maintenance release of the old CP. All interfaces should be upward compatible. In addition, problems are raised by going back and forth between old and new releases of the system. When it is the middle of the first production day and the new SCP will not stay up, migration back to the old version must be fast and orderly.

SUMMARY

To summarize, compatibility is important, but not an overriding requirement. IBM should be sensitive to the fact that data processing installations are trying to deliver service to other groups of users. These users are concerned with continuity, reliability and availability of the computer system. Although customers are willing to tolerate some incompatibilities, they feel that each one must be justified, and implemented in such a way that it minimizes disruption imposed on the end users.

When incompatibilities must occur, the following guidelines will help minimize the impact:

- 1. Incompatibilities should be clearly noted in a planning guide and in other announcement material.
- 2. Conversion aids should be provided. These aids should work in both upward and downward directions, so that the new release can be backed out if necessary.
- 3. Program maintenance for the old release should be provided so that new versions of one component can be run under old versions of another, and vice versa. Directories and data structures should have to be converted once only.
- 4. Before developing a new function, some field research should be done to find out how some customers are currently providing the function. This could avoid future incompatibility, as well as verifying that the function being delivered is the same as the one requested.

REFERENCES

1. This discussion is adapted from a file on VM compatibility placed on the VM SHARE system by Gary Schulz, Northwest Industries, Inc; Chicago, Ill.

On Options

2

This section discusses how options should be presented to the user. There is a difference between the concepts of user-related options and parameters.¹ Unfortunately, the term *parameter* has developed the connotations of complexity and uselessness because of the many parameters which exist today (e.g., SPACE and UNIT). The term *option* is used here to describe user-supplied or application program-supplied information.

Most of the options currently existing on the operating systems have been retained from the past when systems were not as sophisticated, and when the systems relied on the user to make decisions. Although software technology has progressed, many of the old interfaces remain the same.

The intent of the LSRAD proposals is to avoid coupling applications to either the hardware or to the software, as is prevalent in current large systems. The user should never have to specify options for physical device characteristics. The system should be able to obtain any physical device information necessary and retrieve it as desired.

An illustration of the benefits of decoupling may be taken from color television. When color television sets were first produced, they did not work well, so it was necessary to adjust the color level, tint, and contrast each time the channel was switched. Current television models are considerably improved, and the adjustment knobs rarely have to be touched. In many cases, the knobs have been moved to the back of the set, or inside the covers. The result is that modern television sets provide superior performance because they no longer rely on the viewer (user) for control of the picture, but rather use information encoded with the video signal, which is provided directly by the station (application). Although a function (tuning) has been removed from the viewer's (user's) responsibility, the end result is improved.

This is not to say that all options are bad. The user can provide certain information which is important. However, the system should provide a reasonable guess if the user does not provide that information. An example serves to illustrate this point. When a driver enters a parking lot, the attendant usually asks when the driver expects to return. Proper placement of cars results in better service for the customer and less work for the attendant. The attendant does not ask the customer where to park the car; the attendant makes that decision based on his knowledge of the rest of the parking lot users. The attendant also does not refuse to park the car if the driver is not sure about the time of return. Instead, the attendant makes a guess at the best place to leave the car, and all customers may suffer degraded service if that decision causes the attendant extra work moving cars. However, most of the time the system works fairly well.

What kind of options might be specified by a user? With current systems, there is no simple, human-oriented means of specifying that a program or data set needs additional reliability. If the user is interested in good performance or lower costs, these concepts also cannot be properly communicated. Instead, the user must second guess the system. The user tells the system to put a file under the fixed heads of the 3350, to keep the journal data set separate from the data base, or to move a data set to the 3850 instead of keeping it on a drum.

The system's vocabulary could be expanded to recognize the concepts of reliability, performance and cost. The user should supply the relative tradeoffs among them for his applications, rather than telling the system how to achieve them. The following statements summarize the task force's requirements on options:

- 1. Any option specified by the user should be in terms relevant to the application, user, or command being executed, rather than to the hardware or operating system.
- 2. All options should have appropriate default values or actions to be taken.

One of the characteristics of human short-term memory is that a person can remember only about seven items or "chunks" at any one time.² This is why telephone numbers are broken up into groups of threes and fours. One of the unfortunate by-products of creating a new option or keyword is that if it is useful enough for a person to remember when and how to use it, it will probably cause a person to forget the details about another option. Since current systems are intolerant of faults, forgetting one detail can result in failure of a job or transaction. Tolerance through the selection of system-generated defaults which do not cause job failure would greatly aid the end user's task.

The only time the user should be involved in the selection of options is to tell the system the tradeoffs specific to the particular application. The system would determine how best to satisfy these constraints with the best mix of resources. Whatever the decision made by the system, it should never result in the system discarding an otherwise successful execution. This concept is examined in greater depth in the following section, On Limits.

Having the computer system perform the tradeoffs based on user-supplied criteria solves two problems with the current method of user/system communication:

- 1. Currently, the user must know a considerable amount about the details of the hardware and software to achieve even moderately good tradeoffs between reliability, performance, and cost. This information takes too long to learn and changes too rapidly. Moreover, most human solutions are far from optimal.
- 2. The optimal solutions to these tradeoffs are constantly changing. Only the system has the most complete and current information as to its status. The system can apply this knowledge to achieve the best tradeoffs satisfying not only the current user but all others in the system as well. For example, if a drum were to be added to the complex, the system could automatically factor it into its analysis to provide better performance for users. None of the users would have to alter his tradeoffs among reliability, performance, and cost. However, from a system perspective the constraints would be easier to satisfy.

REFERENCES

- 1. Based on a discussion with Wayne Hathaway from NASA, Ames, Moffet Field, California, placed on the SHARE VM system.
- 2. W.J.Tracz, "Computer Programming and the Human Thought Process," IBM Federal Systems Division, Technical Report 76TPA0055, 1977.

On Limits

One of the major features of operating systems that led to their quick acceptance in the mid-60s was their ability to assist users in the sharing of resources, either serially or concurrently. Sharing had large benefits for user productivity, but it exposed users to a new problem. Anyone familiar with the behavior of small children recognizes that humans share only with some difficulty. As children grow older, they get only marginally better at considering needs and desires of others along with their own, and and this is certainly true of data processing professionals.

For example, an individual user might allocate all of the DASD space (either accidentally or on purpose) while another ran a job that got into a loop and used hours of CPU time. Only with reluctance would a user admit that something was wrong with his program and allow it to be canceled. Another user's program that looped around a print statement would waste boxes of paper while tying up a printer needed by others. At the end of the month these users' supervisors would be presented with the costs of processing. After much anguish over exceeded budgets, the supervisors would ask what could be done to make sure this never happens again.

Thus the operating system was given one more responsibility, enforcing limits on the quantity of resources any user could demand. Users were required to describe the resource requirements of their work, and the operating system would do the following:

- Schedule jobs for an optimum mix of resource requirements
- Prevent users from receiving quantities of resources that would have an excessively adverse effect on other users
- Enforce the upper limits set by the user
- Enforce budget limits set by the user or the user's organizatior

This limiting process was designed and implemented at a time when almost all users were data processing professionals. They could deal reasonably well with the hardware orientation of resource parameters (e.g., CPU time or tracks of DASD space). The users became proficient at making estimates so that they seldom lost an otherwise good run.

The task force believes that limits on allocation of computer resources are as necessary today as they were when originally developed. Operating systems can use these limits quite effectively in scheduling work, particularly as distributed processing networks become more common and greater in size.

The change in the user population to that of the non-DP professional requires that operating system designers reconsider how limits are to be implemented. The task force proposes the following requirements against which any design should be measured:

- 1. The user must be able to describe resource requests in human terms, not in computeroriented quantities. Moreover, the operating system should assist the user in creating meaningful estimates based on an examination of the work request.
- 2. The operating system must enforce its limits in a nondestructive way. Valid processing should not be lost because a resource estimate has been excee led.

These requirements have significant implications for both users and for operating system designers. The importance of these requirements lies in how they interact to assist the end

user. They are synergistic. Application of these rules will result in more accurate estimates that are implemented with far better human engineering.

Consider the problems facing the nondata processing professional when trying to run a simple compilation of a source program. Estimates must be provided for CPU time, work area storage space, print and card quantities, and storage space for the object deck. Certainly the user can accept existing defaults, but at some risk. Ultimately the user must learn how to make these estimates, probably through trial and error.

This is not productive use of the person's time or of computer processing. The operating system could make a reasonably accurate estimate of resources based solely on the size of the source file. The operating system knows that parameter before it starts the compilation, so why ask the user to supply data which the operating system could generate for itself? The only reason is to handle the exceptional case. That leads directly to the second requirement.

When an estimate is exceeded today, all processing is terminated and all intermediate work is discarded. When considered in terms of human-engineering, this is wasteful. What does a programmer do when somebody gives him a job that turns out to be bigger than he thought? Does he discard the intermediate work and inform his manager that the work will have to be assigned again because he could not get it done in the time estimated? What he does is take that intermediate work to his supervisor, along with an estimate of how much more work is needed to finish the job and a recommendation for future action on the task. The operating system should take an analogous action.

As the cost of DASD space continues to fall, it becomes reasonable to suspend processing when an estimate is exceeded. The user can be informed of what has happened. The user should be allowed to examine the intermediate output, such as partially complete print files, so that an intelligent decision can be made based on his knowledge of the problem. Suspending a job is certainly a lot less inconvenient and costly than having it totally discarded. The user should retain the ability to override the operating system's estimates, but in a more humanoriented manner. The user could supply factors to be applied to the computer-generated estimates of computer time, intermediate storage, or whatever, without having to make those estimates in computer oriented terms. Essentially, the user should be allowed to tell the operating system, "This job will print twice as many lines as you might expect, so keep that in mind when creating your estimates. If that amount is not enough, just stop where you are and let me see what is going on." The application development process is, by nature, one of trial-and-error. Implementing limits in this way allows the operating system to enhance that process.

Such a design would be invaluable to production staffs and would not impede application development. Based on information available to the operating system, it could generate resource estimates for production jobs that would be far more accurate than current seat-ofthe-pants parameters that application developers create when transferring programs to their production staffs. Would the suspension of jobs from time to time make production schedules hard to meet? Probably not. Today customers have entire staffs devoted to the error recovery process. Most production machines devote a significant portion of time to job reruns caused by errors in resource estimates. Allowing an analyst to examine intermediate output before resuming execution is certainly far less time-consuming than restarting the job.

This approach to limits is consistent with the initial reasons for implementing them. Due to budgeting constraints, users will probably not request an excessive amount of resources. This philosophy can be combined with the use of hierarchical storage management so that only

On Limits

frequently referenced data is kept on the fastest devices. This philosophy allows the operating system to do a meaningful job of budget management, which is one of the important considerations for stewardship of work. Suspension of processing for budgetary reasons gives management a valuable tool to control the expenditure of its data processing dollars. It also gives the application developers an additional motive for being aware of the significance of their actions with respect to expenses incurred.

As with the other proposals, there are secondary benefits which may be larger than the primary ones. The implications of this new limiting philosophy on distributed processing networks should be examined. At present, the user does a certain amount of processing on a local machine, such as text editing, compilations, and debugging of individual modules. A quantity of work is then assembled and shipped via the network to the node where the work is to be done. The user has to be sure that all necessary data sets are on site when the work is processed. Such a network has many advantages, including load balancing, reduced turnaround time, increased ability to share data, and reduced hardware costs. However, there are some weaknesses in this approach. When the user selects the node at which processing should be done, it may or may not be the optimum one relative to cost, turnaround time, or reliability.

As data transmission cost per byte decreases, it will be economically feasible to have the network "collect" all of the data needed by a job and transmit it to the proper node so that it is available when needed. One can envision a day when a user submits a request for work without concern for where it will be run. The network will select the proper node based on existing demand, the cost of transmitting the data and the user's desires for cost and turnaround. The various nodes would get into a bidding competition for each job: node A can do the job for x dollars with y turnaround and node B can do the job for m dollars with n turnaround. The user would then make the choice. All of this would have to be based on a meaningful, reliable description of the resource requirements of a job, and this could not be device dependent, since the nodes would in all probability be made up of various types of processors. Returning to our previous example, the user could submit the compilation of a file, and the characteristics of the job (source file size) would be part of the description that each node would use to evaluate its ability to do the job.

89

BIBLIOGRAPHY

- Attanasio, C. Richard, Peter W. Markstein, and Ray J. Phillips. "Penetrating an Operating System: A Study of VM/370 Integrity," *IBM Systems Journal*, Vol. 15, No.1 (1976), pp. 102-116.
- Baker, F.T. "Chief Programmer Team Management of Production Programming," *IBM Systems Journal*, Vol. 11, No. 1 (1972), pp. 56-73.

Bertalanffy, Ludwig Von. General Systems Theory. New York, 1968.

- Boehm, Barry W. "Software and Its Impact: A Quantitative Assessment," *Datamation*, May 1973, pp. 48-59.
- Brooks, Frederick P., Jr. The Mythical Man Month. Reading, Massachusetts, 1975.
- Bureau of Labor Statistics. "Computer and Related Occupations," Occupational Outlook Handbook, 1978-79 Edition. Bulletin 1955, Washington, D.C., 1978, pp. 111-117.
- Canning, Richard G., and Barbara McNurlin. "Micros Invade the Business World," *Datamation*, August 1978, pp. 93-95.
- Cashman, Michael. "1978 DP Budgets," Datamation, January 1978, pp. 92-95.
- Chen, Edward T. "Program Complexity and Programmer Productivity," *IEEE Transactions on* Software Engineering, Vol. SE-4, No. 3 (May 1978), pp. 187-194.
- Chrysler, Earl. "Some Basic Determinants of Computer Programming Productivity," Communications of the ACM, Vol. 21 No. 6 (June 1978), pp. 472-483.
- Churchman, C. West. The Systems Approach and Its Enemies. New York, 1979.
- Couger, J. Daniel, ed. Computing Newsletter for Instructors of Data Processing, Vol. IX, No. 2 (October 1975).
- Daley, R.C., and J.B. Dennis. "Virtual Memory, Processes and Sharing in MULTICS," Communications of the ACM Vol. 11 No. 5 (May 1968), pp. 306-312.
- Daly, Edmund B. "Management of Software Development," IEEE Transactions on Software Engineering, May 1977, pp. 229-242.
- "Data Base Directions: A presentation for IBM Management," presented by Guide International Information Management Division Steering Committee, July 26, 1978.

Bibliography

- Doherty, W.J. "Man/Computer Communication," Infotech State of the Art Conference, Amsterdam, November 6-8 1978.
- Dolotta, T.A., M.I. Bernstein, R.S. Dickson, Jr., N.A. France, B.A. Rosenblatt, D.M. Smith, and T.B. Steel. *Data Processing in 1980-1985*. New York, 1976.
- Dooley, Ann. "Analyst, Programmer Jobs Increasing--And Changing," Computerworld, April 17, 1978, p. 1.
- Dorn, Phillip H., with the *Datamation* staff. "1979 DP Budget Survey," *Datamation*, January 1979, pp. 162-170.

EDP Industry Report (IDC Corp. Newsletter), P.J. McGovern, Publisher:

Vol. 14, No. 18, February 23, 1979

Vol. 14, No. 2, July 14, 1978

Vol. 13, No. 15 and 16, February 28, 1978

Vol. 13, No. 14, February 15, 1978

Vol. 13, No. 9, November 14, 1977

Vol. 10, No. 8, December 31, 1974

Vol. 9, No. 10, February 27, 1974

Vol. 8, No. 13, April 27, 1973

Vol. 7, No. 16, June 30, 1972

- Ehrman, John P. "The Babel of Application Tools," *Proceedings of SHARE 53*, New York, August 1979.
- Fisher, D.A. "DoD's Common Programming Language Effort," Computer, Vol. 11 No. 13 (March 1978), pp. 25-33.
- Frank, Ronald A. "User Budgets for DP Expected Rise 14.4%," Computerworld, February 26, 1979, p. 1.

Fuller, R. Buckminster. Synergetics. New York, 1975.

- Garth, William. "Design Console Technology at General Motors" (manuscript), General Motors Manufacturing Development Technical Center, Warren Michigan, July 1974.
- Gilchrist, Bruce and Richard E. Weber, Eds. The State of the Computer Industry in the United States, Montvale, New Jersey, 1973.
- Goetz, Martin A. "New Software Tools Increase Productivity," Infosystems, February 1977, pp. 66-70.
- Gordon, Robert L., "The Impact of Automated Memory Management on Software Architecture," Computer USA, November 1973, pp. 31-36.
- Holmes, Edith. "Future Communications DPP Key: Branscomb," Computerworld, June 5, 1978, p. 6.

- . "Technology or Applications More Crucial to Growth?" Computerworld, September 19, 1977, p. 1.
- IBM System/370 Principles of Operation. IBM System Products Division, GA22-7000, 1976.

IBM System/38 Technical Developments. IBM General Systems Division, G580-0237, 1978.

- IBM Time Sharing System Concepts and Facilities. IBM Data Processing Division, GC28-2003, 1978.
- Johnson, James R. "A Working Measure of Productivity," *Datamation*, February 1977, pp. 106-110.
- Jones, Capers. "Program Quality and Programmer Productivity," IBM General Products Division, Technical Report TR02.764, San Jose, California, January 28, 1977.
- Kendall, Robert C. "Management Perspectives on Programs, Programming and Productivity," IBM CHQ Division, White Plains, N.Y.

Lecht, Charles P. The Waves of Change. New York, 1977.

- Kirchner, Jake. "IBMer Sees DP Technology Booming Again," Computerworld, February 19, 1979, p. 7.
- Lehman, John H. "How Software Projects are Really Managed," *Datamation*, January 1979, pp. 119-129.
- Lientz, B.P., E.B. Swanson, and G.E. Tompkins. "Characteristics of Application Software Maintenance," Communications of the ACM, Vol. 21, No. 6 (June 1978), pp. 466-471.

Liu, Chester C. "A Look at Software Maintenance," Datamation, November 1976, pp. 51-55.

- Maass, Richard C. "Finding the Right Job for a Minicomputer," Computer Decisions, August 1978, pp. 44-45.
- Mills, Harlan D. "Chief Programmer Teams: Principles and Procedures," IBM Federal Systems Division, FSC 71-5108, June 1971.
- . "Software Development," IEEE Transactions on Software Engineering, December 1976, pp. 265-273.

Morrissey, J.H., and L.S.-Y. Wu. "Software Engineering...An Economic Perspective", Proceedings, 4th International Conference on Software Engineering, Munich, September 1979.

"1978 EDP Salaries Report," Infosystems, June 1978, pp. 66-67.

- Nyborg, Phillip S., Pender M. McCarter, and William Erickson, eds. Information Processing in the United States a Quantitative Summary. Montvale, New Jersey, 1977.
- Organick, E.I. The Multics System: An Examination of its Structure. Cambridge, Massachussetts, 1972.
- Requa, Joseph E. "Virtual Memory Design Reduces Program Complexity," Computer Design, January 1978, pp. 97-106.
- Rhodabarger, Dale. "Trends for 1979-- A Survey of the Future," Computer Decisions, January 1979, pp. 50-55.
- Ripley, G. David. "Program Perspectives: A Relational Representation of Measurement Data," *IEEE Transactions on Software Engineering*, Vol. SE-3, No. 4 (July 1977), pp. 296-300.
- Severino, Elizabeth F. "When Minicomputers Are on Your Mind," Computer Decisions, November 1978, pp. 40-48.

Bibliography

- Shneiderman, Ben. "Human Factors Experiments in Designing Interactive Systems," Computer, Vol. 12 No. 12 (December 1979), pp. 9-19.
- Swanson, E.Burton. "The Dimension of Maintenance," Proceedings, 2nd International Conference on Software Engineering, October 1976, pp. 492-497.
- Tracz, W.J. "Computer Programming and Human Thought Process," IBM Federal Systems Division, Technical Report 76TPA0055, Owego, New York, 1977.
- "Trends in Application Software-- 1980, 1985, 1990," The Diebold Research Program, Diebold Group, Inc., New York, September 1975

Wagner, Frank V. "Is Decentralization Inevitable?," Datamation, November 1976, pp. 86-97.

Walsh, Myles E. "Where Have All the Minis Gone?," Infosystems, July 1978, pp. 62-73.

- Walston, C.E. and C.P. Felix. "A Method of Programming Measurement and Estimation," *IBM Systems Journal*, Vol. 16, No. 1 (1973), pp. 54-73.
- Winograd, Terry. "Beyond Programming Languages," Communications of the ACM, Vol. 22 No. 7 (July 1979), pp. 391-401.
- Withington, Frederic G. "IBM's Future Large Computers," *Datamation*, July 1978, pp. 115-120.
- Zelkowitz, Marvin V. "Perspectives on Software Engineering," ACM Computing Surveys, Vol. 10, No. 2 (June 1978), pp. 197-216.

APPENDIX: SURVEY RESULTS

Following the LSRAD presentation, "MVS/VM(1980-1985): A User Perspective," at SHARE 51 in Boston and at GUIDE 47 in Chicago, a questionnaire was distributed to obtain a reaction to the task force proposals.

The answers were compiled and results summarized by machine size and operating system. SHARE provided 302 responses from 227 unique installations; GUIDE had 49 responses from 44 unique installations. The difference in number resulted from making the SHARE presentation during a regularly scheduled session, whereas the GUIDE presentation was made at GAPS (night session). The ratio of completed questionnaires versus audience size was approximately 50% in both cases.

Respondents had machines which ranged from 138s to 3033s and operating systems which ranged from DOS to MVS. In all variations of analyses, the results were remarkably similar.

Distribution of responses to the directional questions and the technical proposals are given for total respondents, SHARE and GUIDE, SHARE alone, and GUIDE alone. The directional questions are followed by subsets of MVS installations, VM/370 installations, 158 machine size and up, and 168 machine size and up. Subsets of MVS installations and VM/370installations are provided for the technical proposals.

DIRECTIONAL QUESTIONS:

| | YE | YES | | NO | | |
|---------------------------------|--------|-------|--------|------|--|--|
| | Number | % | Number | % | | |
| 1. Addressing Relevant Problems | 335 | 97.6 | 8 | 2.3 | | |
| SHARE only | 290 | 98.3 | 5 | 1.7 | | |
| GUIDE only | 45 | 93.8 | 3 | 6.3 | | |
| MVS Installations only | 133 | 97.8 | 3 | 2.2 | | |
| SHARE | 106 | 98.1 | 2 | 1.8 | | |
| GUIDE | 27 | 96.4 | 1 | 3.6 | | |
| VM Installations only | 77 | 98.7 | 1 | 1.3 | | |
| SHARE | 73 | 98.6 | 1 • | 1.4 | | |
| GUIDE | 4 | 100.0 | 0 | 0.0 | | |
| | | | | | | |
| 158 and up | 268 | 97.4 | 7 | 2.5 | | |
| SHARE | 226 | 98.3 | 4 | 1.7 | | |
| GUIDE | 42 | 93.3 | 3 | 6.6 | | |
| | | | | | | |
| 168 and up | 170 | 97.1 | 5 | 2.8 | | |
| SHARE | 141 | 97.9 | 3 | 2.1 | | |
| GUIDE | 29 | 93.5 | 2 | 6.4 | | |
| | | | | | | |
| 2. Sound Technical Approach | 314 | 94.3 | 19 | 5.7 | | |
| SHARE only | 271 | 94.8 | 15 | 5.2 | | |
| GUIDE only | 43 | 91.4 | 4 | 8.5 | | |
| MVS Installations only | 120 | 93.0 | .9 | 6.9 | | |
| SHARE | 96 | 94.1 | 6 | 5.9 | | |
| GUIDE | 24 | 88.9 | 3 | 11.1 | | |
| VM Installations only | 74 | 97.3 | 2 | 2.6 | | |
| SHARE | 70 | 97.2 | 2 | 2.8 | | |
| GUIDE | 4 | 100.0 | 0 | 0.0 | | |
| 158 and up | 249 | 93 6 | 17 | 6.4 | | |
| SHARE | 209 | 94.1 | 13 | 5.8 | | |
| GUIDE | 40 | 90 1 | 4 | 9.1 | | |
| 168 and up | 157 | 92 9 | 12 | 7.1 | | |
| SHARE | 130 | 93.5 | 9 | 6.4 | | |
| GUIDE | 27 | 90.0 | 3 | 10.0 | | |

| | YES | | NO | | |
|--------------------------------|--------|-------|--------|------|--|
| | Number | % | Number | % | |
| 3. Productivity Should Improve | 333 | 97.6 | 8 | 2.3 | |
| SHARE only | 288 | 98.3 | 5 | 1.7 | |
| GUIDE only | 45 | 93.8 | 3 | 6.3 | |
| MVS Installations only | 129 | 95.5 | 6 | 4.4 | |
| SHARE | 104 | 97.2 | 3 | 2.8 | |
| GUIDE | 25 | 89.3 | 3 | 10.7 | |
| VM Installations only | 76 | 100.0 | 0 | 0.0 | |
| SHARE | 72 | 100.0 | 0 | 0.0 | |
| GUIDE | 4 | 100.0 | 0 | 0.0 | |
| 158 and up | 262 | 97.0 | 8 | 2.9 | |
| SHARE | 220 | 97.8 | 5 | 2.2 | |
| GUIDE | 42 | 93.3 | 3 | 6.6 | |
| 168 and up | 166 | 96.5 | 6 | 3.4 | |
| SHARE | 137 | 97.1 | 4 | 2.8 | |
| GUIDE | 29 | 93.5 | 2 | 6.4 | |
| | | | | | |
| 4. Easier for End-user | 321 | 95.3 | 16 | 4.7 | |
| SHARE only | 280 | 96.2 | 11 | 3.8 | |
| GUIDE only | 41 | 89.1 | 5 | 10.8 | |
| MVS Installations only | 122 | 92.4 | 10 | 7.6 | |
| SHARE | 101 | 95.3 | 5 | 4.7 | |
| GUIDE | 21 | 80.8 | 5 | 19.2 | |
| VM Installations only | 73 | 96.0 | 3 | 3.9 | |
| SHARE | 69 | 95.8 | 3 | 4.1 | |
| GUIDE | 4 | 100.0 | 0 | 0.0 | |
| 158 and up | 253 | 94.4 | 15 | 5.6 | |
| SHARE | 215 | 95.5 | 10 | 4.4 | |
| GUIDE | 38 | 88.3 | 5 | 11.6 | |
| 168 and up | 161 | 94.7 | 9 | 5.3 | |
| SHARE | 134 | 95.7 | 6 | 4.3 | |
| GUIDE | 27 | 90.0 | 3 | 10.0 | |

Survey Results

| | YES | | NO | | |
|-----------------------------|--------|------|--------|------|--|
| | Number | % | Number | % | |
| 5. Allow New Applic. Devel. | 262 | 81.6 | 59 | 18.4 | |
| SHARE only | 230 | 83.3 | 46 | 16.6 | |
| GUIDE only | 32 | 71.1 | 13 | 28.9 | |
| MVS Installations only | 103 | 81.1 | 24 | 18.9 | |
| SHARE | 83 | 83.0 | 17 | 17.0 | |
| GUIDE | 20 | 74.1 | 7 | 25.9 | |
| VM Installations only | 59 | 83.1 | 12 | 16.9 | |
| SHARE | 56 | 83.6 | 11 | 16.4 | |
| GUIDE | 3 | 75.0 | 1 | 25.0 | |
| 158 and up | 211 | 82.4 | 45 | 17.6 | |
| SHARE | 180 | 84.5 | 33 | 15.4 | |
| GUIDE | 31 | 72.1 | 12 | 27.9 | |
| | 137 | 83.0 | 28 | 16.9 | |
| 168 and up | 116 | 85.3 | 20 | 14.7 | |
| SHARE GUIDE | 21 | 72.4 | 8 | 27.6 | |
| | | | | | |

TECHNICAL PROPOSALS

| | AGR | EE | NEUTI | NEUTRAL | | EE |
|------------------------|------------------|-----------|--------|---------|--------------|-------|
| | Number | % | Number | % | Number | % |
| 1. Named Spaces usefu | il to Applicatio | n Develop | er? | | | |
| 1. Itumed Spuces abore | 241 | 77.0 | 53 | 16.9 | 19 | 6.1 |
| SHARE | 212 | 78.8 | 43 | 16.0 | 14 | 5.2 |
| GUIDE | 29 | 65.9 | 10 | 22.7 | 5 | 11.4 |
| | | | | | | · · · |
| MVS only | 87 | 72.5 | 22 | 18.3 | 11 | 9.2 |
| SHARE | 70 | 73.7 | 18 | 18.9 | 7 | 7.4 |
| GUIDE | 17 | 68.0 | 4 | 16.0 | 4 | 16.0 |
| XD / 1 | 50 | 041 | 6 | 07 | · - · | 7.0 |
| VM only | 58 | 84.1 | 6 | 8.7 | 5 | 7.2 |
| SHARE | 55 | 84.6 | 5 | 7.7 | 5 | 7.7 |
| GUIDE | 3 | 75.0 | 1 | 25.0 | 0 | 0.0 |
| | | | | | | |
| Named Spaces useful t | | · · · · | | | ર દેવનું દાવ | |
| | 277 | 82.9 | 40 | 12.0 | 17 | 5.1 |
| SHARE | 240 | 83.6 | 33 | 11.5 | 14 | 4.9 |
| GUIDE | 37 | 78.7 | 7 | 14.9 | 3 | 6.4 |
| MVS only | 107 | 82.9 | 14 | 10.9 | 8 | 6.2 |
| SHARE | 86 | 84.3 | 10 | 9.8 | 6 | 5.9 |
| GUIDE | 21 | 77.8 | 4 | 14.8 | 2 | 7.4 |
| | | | | | | · |
| VM only | 68 | 88.3 | 5 | 6.5 | 4 | 5.2 |
| SHARE | 65 | 89.0 | 4 | 5.5 | 4 | 5.5 |
| GUIDE | 3 | 75.0 | 1 | 25.0 | 0 | 0.0 |
| | | | | | | |

Survey Results

| | AGI | REE | NEUTI | NEUTRAL | | DISAGREE | |
|---------------------|-----------------|------------|--------|---------|--------|----------|--|
| | Number | % | Number | % | Number | % | |
| 2. Extended Address | ing useful to A | ppl. Dev.? | | | | | |
| | 232 | 76.1 | 54 | 17.7 | 19 | 6.2 | |
| SHARE | 199 | 75.9 | 50 | 19.1 | 13 | 5.0 | |
| GUIDE | 33 | 76.7 | 4 | 9.3 | 6 | 14.0 | |
| MVS only | 91 | 75.8 | 17 | 14.2 | 12 | 10.0 | |
| SHARE | 73 | 76.8 | 15 | 15.8 | 7 | 7.4 | |
| GUIDE | 18 | 72.0 | 2 | 8.0 | 5 | 20.0 | |
| VM only | 47 | 74.6 | 14 | 22.2 | 2 | 3.2 | |
| SHARE | 45 | 75.0 | 13 | 21.7 | 2 | 3.3 | |
| GUIDE | 2 | 66.7 | 1 | 33.3 | 0 | 0.0 | |
| Extended Addressing | useful to End | Users? | | | | | |
| Entended Haarebbing | 282 | 84.2 | 41 | 12.2 | 12 | 3.6 | |
| SHARE | 250 | 86.5 | 29 | 10.0 | 10 | 3.5 | |
| GUIDE | 32 | 69.6 | 12 | 26.1 | 2 | 4.3 | |
| MVS only | 111 | 85.4 | 15 | 11.5 | 4 | 3.1 | |
| SHARE | 91 | 87.5 | 9 | 8.7 | 4 | 3.8 | |
| GUIDE | 20 | 76.9 | 6 | 23.1 | 0 | 0.0 | |
| VM only | 64 | 84.2 | 10 | 13.2 | 2 | 2.6 | |
| SHARE | 61 | 84.7 | 9 | 12.5 | 2 | 2.8 | |
| GUIDE | 3 | 75.0 | 1 | 25.0 | 0 | 0.0 | |

| | AGR | EE | NEUTI | NEUTRAL | | DISAGREE | |
|--|--------------------|-----------|--------|---------|--------|----------|--|
| | Number | % | Number | % | Number | -% | |
| 3. Subsystem Prot | ection useful to A | ppl. Dev. | 2 | | | | |
| 이 영상, 비행, 비행, 비행, 비행, 비행, 비행, 비행, 비행, 비행, 비행 | 222 | 72.8 | 70 | 22.9 | 13 | 4.3 | |
| SHARE | 200 | 76.0 | 54 | 20.5 | 9 | 3.4 | |
| GUIDE | 22 | 52.4 | 16 | 38.1 | 4 | 9.5 | |
| MVS only | 80 | 67.2 | 32 | 26.9 | 7 | 5.9 | |
| SHARE | 69 | 73.4 | 21 | 22.3 | 4 | 4.3 | |
| GUIDE | 11 | 44.0 | 11 . | 44.0 | 3 | 12.0 | |
| VM only | 52 | 76.5 | 13 | 19.1 | 3 | 4.4 | |
| SHARE | 49 | 76.6 | 12 | 18.7 | 3 | 4.7 | |
| GUIDE | 3 | 75.0 | 1 | 25.0 | 0 | 0.0 | |
| Subsystem Protecti | ion waful for Ead | Hears? | | | 4) (ii | | |
| Subsystem Protect | 258 | 78.9 | 60 | 18.3 | 9 | 2.8 | |
| SHARE | 230 | 81.3 | 46 | 16.2 | | 2.5 | |
| GUIDE | 28 | 63.6 | 14 | 31.8 | 9 2 | 4.6 | |
| MVS only | 96 | 76.8 | 25 | 20.0 | 4 | 3.2 | |
| SHARE | 82 | 82.0 | 15 | 15.0 | 3 | 3.0 | |
| GUIDE . | 14 | 56.0 | 10 | 40.0 | 1 | 4.0 | |
| VM only | 65 | 84.4 | 8 | 10.4 | 4 | 5.2 | |
| SHARE | 62 | 84.9 | 7 | 9.6 | 4 | 5.5 | |
| GUIDE | 3 | 75.0 | 1 | 25.0 | 0 | 0.0 | |
| 202 | | | | | | | |

Survey Results

| | AGR | AGREE | | NEUTRAL | | DISAGREE | |
|--|---------------|----------|--------|---------|---------------------------------------|----------|--|
| | Number | % | Number | % | Number | % | |
| | | 1 . D. | 0 | | | | |
| 4. Device Independent | | | | | | 0.9 | |
| | 316 | 96.9 | 7 | 2.2 | 3 3 | | |
| SHARE | 271 | 96.8 | 6 | 2.1 | | 1.1 | |
| GUIDE | 45 | 97.8 | 1 | 2.2 | 0 | 0.0 | |
| MVS only | 123 | 96.1 | 2 | 1.6 | 3 | 2.3 | |
| SHARE | 98 | 96.1 | 1 | 1.0 | 3 | 2.9 | |
| GUIDE | 25 | 96.2 | 1 | 3.8 | 0 | 0.0 | |
| and the second | | | | | | ~ ~ ~ | |
| VM only | 69 | 97.2 | 2 | 2.8 | 0 | 0.0 | |
| SHARE | 65 | 97.0 | 2 | 3.0 | 0 | 0.0 | |
| GUIDE | 4 | 100.0 | 0 | 0.0 | 0 | 0.0 | |
| | | | | | | | |
| Device Independent I/ | O useful to E | nd User? | | | | | |
| | 275 | 81.1 | 34 | 10.0 | 30 | 8.9 | |
| SHARE | 239 | 81.9 | 27 | 9.2 | 26 | 8.9 | |
| GUIDE | 36 | 76.6 | 7 | 14.9 | 4 | 8.5 | |
| | a a porta | · | | | | | |
| MVS only | 105 | 79.5 | 17 | 12.9 | 10 | 7.6 | |
| SHARE | 87 | 82.8 | 11 | 10.5 | 7 | 6.7 | |
| GUIDE | 18 | 66.7 | 6 | 22.2 | 3 | 11.1 | |
| VM only | 66 | 85.7 | 4 | 5.2 | 7 | 9.1 | |
| SHARE | 62 | 84.9 | 4 | 5.5 | 7 | 9.6 | |
| GUIDE | 4 | 100.0 | 0 | 0.0 | 0 | 0.0 | |
| | | | | a | · · · · · · · · · · · · · · · · · · · | | |

| | AGREE | | NEUT | RAL | DISAGREE | |
|-----------------------|----------------|-----------|--------|-------|----------|--------------|
| | Number | % | Number | % | Number | % |
| | den maafrel fo | . Anni D | orr 9 | | · · · · | |
| 5. Execution-Time Loa | | | | 20.2 | 24 | 7.2 |
| | 239 | 72.4 | 67 | 20.3 | 24 | 7.3 |
| SHARE | 205 | 71.9 | 60 | 21.1 | 20 | 7.0 |
| GUIDE | 34 | 75.5 | 7 | 15.6 | 4 | 8.9 |
| | | | | | | |
| MVS only | 96 | 72.2 | 25 | 18.8 | 12 | 9.0 |
| SHARE | 77 | 72.0 | 21 | 19.6 | 9 | 8.4 |
| GUIDE | 19 | 73.1 | 4 | 15.4 | 3 | 11.5 |
| | | | | · · · | | |
| VM only | 52 | 73.2 | 16 | 22.6 | 3 | 4.2 |
| SHARE | 49 | 72.1 | 16 | 23.5 | 3 | 4.4 |
| GUIDE | 3 | 100.0 | 0 | 0.0 | 0 | 0.0 |
| Gold | · . · · | | | | | |
| | | · | | | | |
| Execution-Time Loader | useful for E | nd Users? | | | | |
| | 213 | 63.4 | 74 | 22.0 | 49 | 14.6 |
| SHARE | 188 | 64.8 | 62 | 21.4 | 40 | 13.8 |
| GUIDE | 25 | 54.3 | 12 | 26.1 | 9 | 19.6 |
| GUIDE | | 0 | | | | |
| MVS only | 69 | 53.1 | 34 | 26.2 | 27 | 20.7 |
| SHARE | 59 | 56.7 | 25 | 24.0 | 20 | 19.3 |
| GUIDE | 10 | 38.5 | 9 | 34.6 | 7 | 26.9 |
| GOIDE | 10 | 50.5 | | 51.0 | | <i>w</i> .,, |
| VM only | 57 | 73.1 | 13 | 16.7 | 8 | 10.2 |
| SHARE | 53 | 71.6 | 13 | 17.6 | 8 | 10.8 |
| GUIDE | 4 | 100.0 | . 0 | 0.0 | 0 | 0.0 |
| GOIDE | . . | 100.0 | . • | 0.0 | | 0.0 |

Survey Results

| | AGRI | | NEUTE | RAL | DISAGREE | |
|---------------------|---------------|--------------|----------|---------|-------------|-----|
| | Number | % | Number | % | Number | % |
| 6. Command Language | e Improvemen | ts useful to | Appl Dev | ? | | |
| 0. Command Language | 300 | 95.3 | 14 | 4.4 | 1 | 0.3 |
| SHARE | 259 | 95.6 | 12 | 4.4 | 0 | 0.0 |
| GUIDE | 41 | 93.2 | 2 | 4.5 | 1 | 2.3 |
| MVS only | 121 | 95.3 | 5 | 3.9 | 1 | 0.8 |
| SHARE | 96 | 95.0 | 5 | 5.0 | 0 | 0.0 |
| GUIDE | 25 | 96.2 | 0 | 0.0 | 1 | 3.8 |
| VM only | 64 | 97.0 | 2 | 3.0 | 0 | 0.0 |
| SHARE | 61 | 98.4 | 1 | 1.6 | 0. | 0.0 |
| GUIDE | 3 | 75.0 | 1 | 25.0 | 0 | 0.0 |
| | | | | | · · · · · · | |
| Command Language In | nprovements u | seful to En | nd User? | · · · . | | |
| | 273 | 82.0 | 45 | 13.5 | 15 | 4.5 |
| SHARE | 238 | 83.2 | 36 | 12.6 | 12 | 4.2 |
| GUIDE | 35 | 74.5 | 9 | 19.1 | 3 | 6.4 |
| MVS only | 104 | 78.8 | 20 | 15.1 | 8 | 6.1 |
| SHARE | 86 | 81.9 | 13 | 12.4 | 6 | 5.7 |
| GUIDE | 18 | 66.7 | 7 | 25.9 | 2 | 7.4 |
| VM only | 64 | 85.3 | 9 | 12.0 | 2 | 2.7 |
| SHARE | 61 | 85.9 | 8 | 11.3 | 2 | 2.8 |
| GUIDE | 3 | 75.0 | 1 | 25.0 | 0 | 0.0 |

103