A GUIDE TO MULTICS FOR SUBSYSTEM WRITERS

Chapter IV

Access Control and Protection

Elliott I. Organick

Draft No. 4

January 1969

Project MAC

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

TABLE OF CONTENTS

TABLE OF CONTENTS (CONT)

TABLE OF CONTENTS (CONT)

LIST OF ILLUSTRATIONS

LIST OF TABLES

CHAPTER IV

ACCESS CONTROL AND PROTECTION

4.1 INTRODUCTION

In this chapter and its successor we want to review our picture of interprocedure communication with a more realistic orientation. We want to understand this communication as one going on in the multi-access environment of Multics where different processes, undoubtedly involving a number of different users, co-exist each with separate objectives (and skills). The users often compete for the computer's resources. They "play" against each other in one way or another fair or foul. Fair play, as in a management decision game with several players, is to be encouraged. Foul play, as for instance one user inadvertantly or deliberately destroying the data or procedures of another user, or of the system itself, is to be more than merely minimized, discouraged or outlawed. It is to be prevented in toto! Multics is designed so that a large measure of fair play can be achieved by cooperating users while at the same time every type of foul play that can be anticipated is prevented.

This ambitious design objective is actually achieved inspite of the fact that encouraging fair play — i.e., permitted cooperation between users — almost inevitably invites accidents, i.e., suggests chances for damaging interaction between users or between a user and the supervisor — or so it would seem. A primary goal of Chapter 4 and, to some extent, Chapters 5, 6, and 7 is to explain how these two objectives are, in fact, achieved. Secondarily, we hope the reader of this chapter will gain confidence that Multics will indeed protect him from the inept practice or foul play of others who share the computer with him. He will also see that to a significant extent Multics can help to protect the user from himself as well.

4.1.1 Compartmentalization - General Concepts

One natural question a subsystem designer might ask is: How does a large process ever get debugged? What helpful provisions are there in Multics to effectively isolate (and insulate) procedure and data segments (or groups of them from one another)? Could one, in principle at least test isolated parts and be sure that when tested parts are put together, undesirable interaction of the parts can be avoided or

at least controlled? This principle of compartmentalization probably goes back to the early days of debugging which, in turn, certainly dates with the first computers.*

Even on these simple, stand-alone computers where a user literally "owned" the entire machine while in execution, ideas of protection began to emerge. To increase the reliability of a program, ways were sought to safeguard data areas; procedures (subroutines) were invented to subdivide large programs and attempts were made to limit the scope of procedures so that no one procedure will be allowed to access any more data than needed. Near the beginning, interpreters were invented as one of the software schemes to help achieve these measures of protection. Much later, hardware innovations provided alternative possibilities.

When batch monitor operating systems were introduced there were new problems of protection. Without benefit of hands-on control a sophisticated user had all the more reason to design large processes in a compartmentalized manner to achieve internal protection of his programs and data. But, in addition, as a consideration to other users in the batch, both ahead of him (on the output tape) and behind him the isolation of the "supervisor" became critically important. In the batch system an executing process could be thought of as having two distinct domains: the supervisory programs and their data bases (S) and the user programs and their data bases (U). To act as a "protected supervisor" in any meaningful sense, it was essential that certain procedures in S have access to the programs and data of U. On the other hand, it became apparent that programs of U should be allowed no direct access to data in S and should be capable of only certain kinds of controlled access (e.g., "trapped" calls to the I/O supervisor) to certain of the programs in S.

The latter distinction accords with the idea that only supervisory programs may execute I/O and other privileged instructions. Hardware developments have made it possible to facilitate this distinction. In many computer operating systems operating under batch monitors, "master mode", which permits execution of the full instruction repertoire, is reserved for supervisory programs. A user program can effect a transit into the master mode only by temporarily giving up direct control, such as by executing an instruction that traps his program to a master mode fault handler.

---

*Professor Maurice Wilkes, of Cambridge University, reports that his laboratory "discovered" debugging the first day the EDSAC became operational while attempting to execute a simple program for generating a table of prime numbers.

Given this type of protection of S and given S's greater freedom to interact with U, we see that

(a) if S malfunctions, it can destroy both S and U,

(b) if U malfunctions, at worst it can only destroy U, leaving S free to load and execute tasks for other users, e.g., for U_1, U_2, etc.

When we now consider an environment like Multics where we have a collection of user domains, U_1, U_2, ... , U_n, and a common supervisor, S, all our earlier incentives for isolating key compartments of a process remain. The consequences of not having adequate protection of S, however, are much worse. We must bear in mind that process 1 consists of U_1 and S; process 2 consists of U_2 and S, etc. Any time supervisory procedures in S are executing, they are maintaining data bases in S that pertain to the entire group of active user processes. If these tables are inadvertantly or deliberately tampered with by U_1 (executing in process 1) or U_2 (executing in process 2), etc., not only would S be damaged, but one or more other user processes are likely to be defeated at the same time. (Destruction of processes can now occur en masse rather than, as typical in the batch system, merely invoking a delay in use of the system by users waiting in the queue.)

There are of course new kinds of problems that need to be considered in a multiprogrammed environment which were less serious in the batch system. One of these is the matter of privacy or, more generally speaking, control over the "sharing"of segments. If a general mechanism is to be provided for allowing two or more running processes to share the same segments, there must also be a complementary capability for preventing certain segments of one user's process from being shared, peeked at, or written in by procedures of another process. Thus to insure the U_1 cannot interact with U_2, e.g., by "peeking", we must rely on a carefully conceived scheme of access control for each segment used in each process. Moreover, the actuating of these access controls must be a function confined to the supervisory procedures in S, using data bases in S.

One begins to see how really critical the design of a "foolproof", "vandal proof", and "burglar proof" protection mechanism is, if any large general purpose multiuser, multi-programmed environment is to endure. The Multics design for access control and protection is intended to be "airtight". In this chapter we hope to describe a major part of this plan.

4.1.2 Compartmentalization - as Achieved in Multics

In Multics compartmentalization is achieved through two primary mechanisms, one supplementing the other.

(a) Per-segment access control.

This is a means of denoting and controlling the type of access to a particular shared segment which may be accorded to an individual user. A segment may be shared by two or more processes, but the person who creates the segment and who "grants" permission for its shared use is able to specify the type of access accorded to each grantee.

By giving to each file's author the privilege of listing the users who shall have access to it, a user is able to safeguard the information he creates and files away for future use. It is true that Multics permits the coexistence of many processes, each of which competes for the system's physical resources and employs the same file system hierarchy. Nevertheless, sharp divisions may be maintained between the processes with respect to the information each may acquire in its address space and how such information may be used. Furthermore, the control rests where it may be most meaningfully exercised — with the user. Per-segment access control may therefore be viewed as a form of inter-process protection. Concepts of access control are introduced in Section 4-2.*

(b) Concentric rings of protection.

The ring mechanism, by contrast, offers intra-process protection of segments. The concentric ring concept is essentially a generalization of the S and U (supervisor and user) domains. The segments of any one process are associated with a set of generally two, but possibly more, concentric rings. If a process has only two concentric rings, then the inner ring corresponds to S and the outer ring to U. But, provision has been built into the Multics design so that the subsystem writer may add (as justified) additional rings. In such applications, segments of the subsystem would be associated with the most appropriate ring (category) vis-a-vis privilege and protection. In this way a designer, say when developing a teacher-student subsystem, may establish one or more extra "lines of defense". These can result in increased protection of the key parts of the subsystem (e.g., teacher-written programs) from damage or misuse by other users of the subsystem (e.g., student-written programs).

Basically, a procedure which is assigned the category of ring r is privileged during its execution to call (or to reference) any procedure (or data) segment in ring r or in any ring peripheral, i.e., "outside of" ring r. Conversely, a procedure of ring r is prevented from referencing data segments in a more "privileged", i.e., "inner" ring and is permitted call access to more privileged procedures only through specially controlled entry points called "gates".

---

*Section 4-2 should not be regarded as a complete treatment of access control. Additional material is given in Chapter 6.

The Controlled entry via gates is into procedures that may reside in any one of several inner rings. This amounts to a software-augmented generalization of the call trapping capability that is employed in conventional batch monitor systems. In these systems the caller traps, when permitted to do so, to procedures that have full privilege (master mode). In Multics, the caller can in effect trap into procedures that have intermediate degrees of privilege, as deemed appropriate by the subsystem designer.

The set of supervisory segments, when viewed as a subsystem can, in principle, also benefit through subdivision into rings. Two rings were originally thought to be desirable; the first was variously referred to as ring 0 or the hardcore ring; the second, ring 1, was also called the administrative ring.* Experience in checking out the earliest versions of Multics indicated, however, that the cost (both in space and time) for maintaining two supervisory rings using existing hardware was not justified. As currently implemented, the Multics supervisor resides essentially in one ring (ring 0). A small portion of the ring 0 segments must in fact remain resident in core at all time. Such segments are referred to as "wired down" and their absolute addresses in memory are known to other ring 0 procedures. The logical structure to support a multi-ring supervisor has been carefully retained. So, a multi-ring supervisor can be readily employed whenever hardware improvements allow it to be justified. For this reason our discussions in this chapter will retain a generality, wherever appropriate, that presumes the existence of a multi-ring supervisor.

Here we summarize the motivation for multiple rings for supervisor and/or user:

By subsetting the segments of a process into rings and by effectively controlling interactions and communication between segments of different rings (supervisory- or user-like), Multics provides the potential to isolate trouble and limit damage in the system. Different rings, in a way, may be equated to different levels of damage. Greater damage to the total system operation would, in general, result from a malfunction of or damage to a segment, the closer its ring is to the hard core or "nerve center" of the system. Conversely, damage that occurs to a segment in an outlying

*Generally speaking, ring 0 segments were those most crucial to the operation of Multics. In this category fell certain key tables and vital procedures which, for instance, govern the multiplexing of the core memory, of the processors, and of other key resources among the processes. Ring 1 segments were, generally speaking, those more numerous and less vital supervisory segments which might more readily be debugged while the system would be in full operation.

user ring, would affect only the user's process or at worst those of other users who happen to share the affected non-supervisory segments. One would correctly intuit that there are significant overhead costs incurred in implementing rings and the implied controls. For instance, extra execution time required to cross from one ring to another during a procedure call or a normal return is of the order of several milliseconds. (This is the cost using the current GE 645 hardware together with the software described in this Chapter.) The notes in Sections 4.2 and 4.3 will provide insight into the costs involved, so that you will be able to assess the tradeoffs among subsystem designs employing alternative ring structures. One type of "ring" overhead is alluded to in the next two paragraphs.

4.1.3 Alteration of the Process Stack Model

The process stack model which we developed in Chapter 3 must undergo an extension to be compatible with the concept of a process that is subdivided in the two- or more-ring sense. We can no longer continue to think of a single (common) stack that can be employed by (i.e., be read-write accessible to) all procedures in one process. For, were this the case, the hoped-for isolation between rings would be easily circumvented. Any (offending) procedure could copy information from the stack or possibly destroy information (including instructions) in it which was stored there by supervisory ("superior" in the inner-ring sense) procedures. Security and protection of information vital to the functioning of a supervisory procedure would thereby be nullified. The Multics solution is to give procedures in each ring of a process a separate stack segment. Of course, all legal communication between procedures of different rings then becomes clerically (though not necessarily conceptually) more complicated than first described in Chapter 3.

Readers can hopefully gain a full overview of all these new ideas by reading Sections 4.2 and optionally proceed to 4.3 for still further details. Protection problems also arise in connection with other types of interprocedure communication, specifically condition handling and abnormal returns. A discussion of these problems and the solutions to these as developed in Multics, is the subject of Chapter 5. The MSPM documentation on which most of this "protection" material is based comes from the BD.9XX sections of the MSPM and from Graham.*

*"Protection in an Information Processing Utility" by R. M. Graham, Communications of the ACM, Vol. 11, No. 5, May 1968, pp. 365-369 (an excellent overview)

4.2 ACCESS CONTROL AND RING BRACKET PROTECTION

In this section we provide some basic details on the two types of isolation techniques, access control and ring brackets* which, in proper combination, are fundamental to the system of protection and to the controlled sharing of data and procedures in Multics.

We have already suggested why segments within a process should be subdivided into rings and why, for each ring there should be a separate stack segment. It is proper to remark here that ring compartmentalization is carried out with some hardware aid. Multics exploits special GE 645 fault-detection hardware to detect and trap a process whenever it attempts to make a cross-ring reference requiring intervention of supervisory software. Without some direct hardware support a ring isolation scheme could be achieved only by execution in a fully interpretive mode, a prohibitively expensive alternative.

Before we can proceed further with the details of the ring mechanism, it is necessary to acquire a clear understanding of the perhaps more fundamental, per-segment access control provisions of the basic file system.

4.2.1 Per-Segment Access Control†

What follows assumes you have, at sometime in the past, read one of the several available Multics overviews on the basic file system and, in particular, the directory structure of the file system hierarchy.† A review of these topics may not be necessary now. We will assume that you have a general knowledge of the file structure:

---

*The notion of a ring bracket to be developed in this section is a slight extension of the ring concept already introduced.

†Principal MSPM references on which the discussion of access control is based are:
BG.0 Overview of the Basic File System, and
BG.9 Access Control.
Auxiliary documents are:
BG.3 Segment Control
BG.7 Directory Data Base
BG.8 Directory Control
BX.8 File System Commands

† As of July, 1967 either the paper by R. C. Daley and P. G. Neumann, pp. 223-227, "A General Purpose File System for Secondary Storage," (Proceedings of the 1965 Fall Joint Computer Conference) or the most recent account, pp. 5-16 through 5-39 of the Multics Operation System, May 1967 (Cambridge Information Systems Laboratory of G E).

(1) that it consists of a tree of directory and nondirectory files; (2) that among other things, a directory contains a set of entries called branches, † each of which points directly to and describes a file in some detail: either a directory file, i.e., another directory, or a nondirectory file, i.e., referring to a block of data or to a procedure. Branches carry unique identification and they are in one-to-one correspondence with the files in secondary storage; (3) that a nondirectory file is simply the way we refer to data or procedure segments kept in secondary storage; we think of them as files on secondary storage in the context of the file system, but as segments when we refer to them in any way as part of a particular process; and (4) file descriptions (branches) are specified either explicitly or by default rules at the time a segment is created. Each branch includes a "permission list" which names each user who is to have access to the file and which specifies the types of permitted access for each listed user. Of course, the creator of a file is automatically listed as a permitted user in the file's branch.

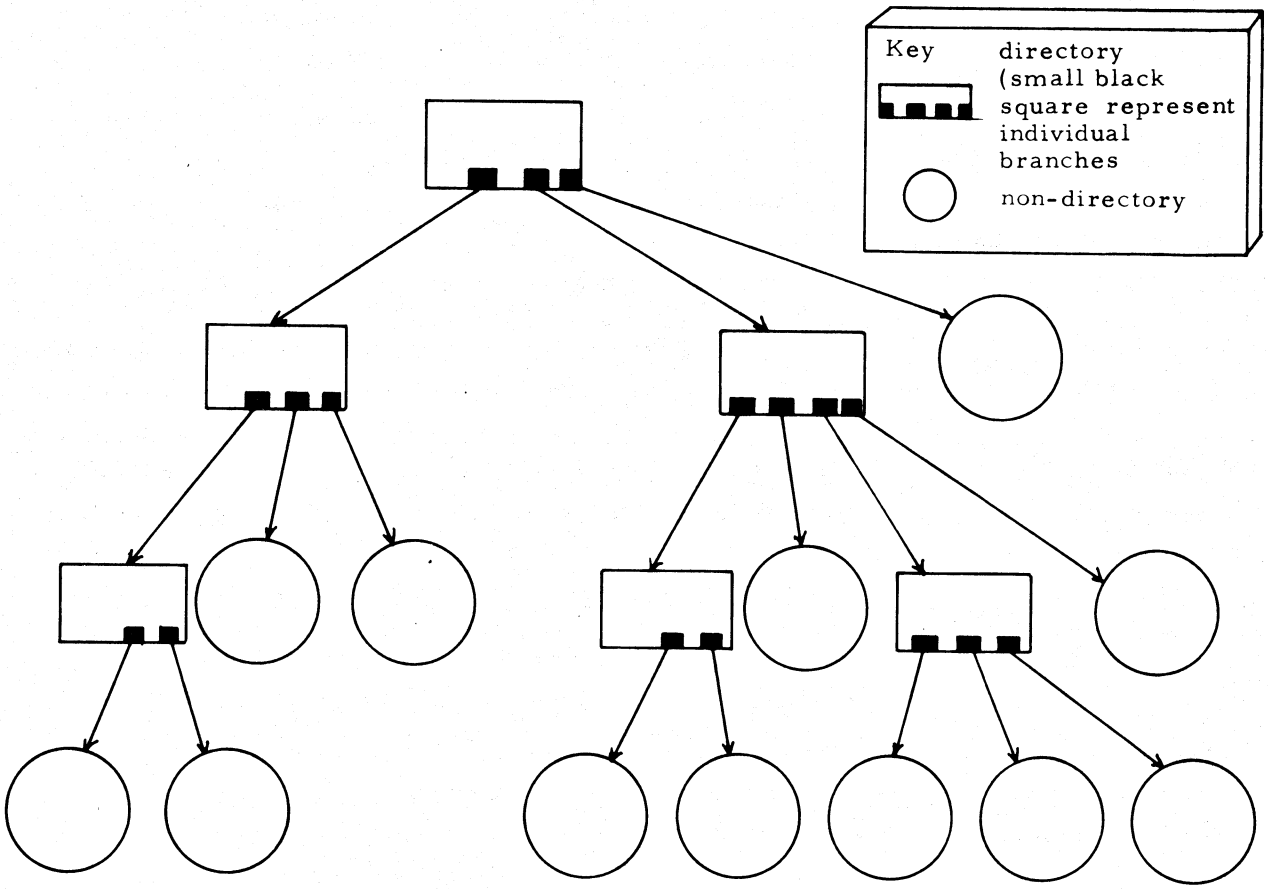
Figure 4-1 is a schematic of the directory structure. At the time he acquires "user status", each user has assigned to him a uniquely-named "user directory" whose file branch is located in a system-maintained directory called "user_directory_directory." Once a user begins executing processes in his own name, he may create new files and add these to the Multics tree. The new files will normally have their corresponding branches in the user's user directory. A user is free to create either non-directory or directory files. The ability to add directories implies that a user if he chooses, can add to the overall system hierarchy a subtree of arbitrary depth whose root is his own user directory.

The creation of a file takes place when a process calls for the creation of a new segment whose name and other descriptions correspond to the desired file. In the course of creating the segment, the Basic File System establishes the segment as a file by constructing and attaching a file branch in the appropriate directory. Subsequent to its creation, writing into the segment amounts to adding information to the file.

A process will frequently make indirect attempts to access an existing file usually by making symbolic reference to it via link faults. The Basic File System (BFS) which is involved in response to the Linker's request, will attempt to "register" the wanted file as a segment of the faulting process. Registering the file amounts to

---

†Another type of entry called a link is discussed in Chapter 6.



Each directory contains a set of "branches" (which point either to other directory files or to non-directory files).

Figure 4-1. Schematic of Directory Structure of the File System

associating with it a segment number and obtaining the information that is needed to form the appropriate segment description word (SDW). In the process, the BFS will first answer the following question. Does the process associated with the faulting procedure have any business at all making this reference, i.e., is any access at all to this segment by this process to be permitted?

To find the answer, supervisor modules will locate and then examine the particular directory* that holds the branch pointing to the desired file — i.e., to the file whose attempted acquisition as a segment caused the fault. The "access control information" found in this branch's "permission list" provides the answer.

First we consider the consequence of a no answer. In this event the desired segment is discovered to be strictly off limits to the current user. In the handling of an ft2 link fault to such a segment, the Linker module would "learn" the no-access news from the appropriate ring 0 module in charge. The Linker then gives up its attempt to establish the desired link, and transmits its failure to the Fault Interceptor.** The latter will now signal its failure to achieve the desired link so that, at least in some subsystems, corrective action may possibly be taken by the user. (We'll discuss the technical meaning of signalling in Chapter 5.)

If the answer is yes, then the file may be used as a segment in the requesting process. Other information in the directory spells out the kind of access that is to be permitted.

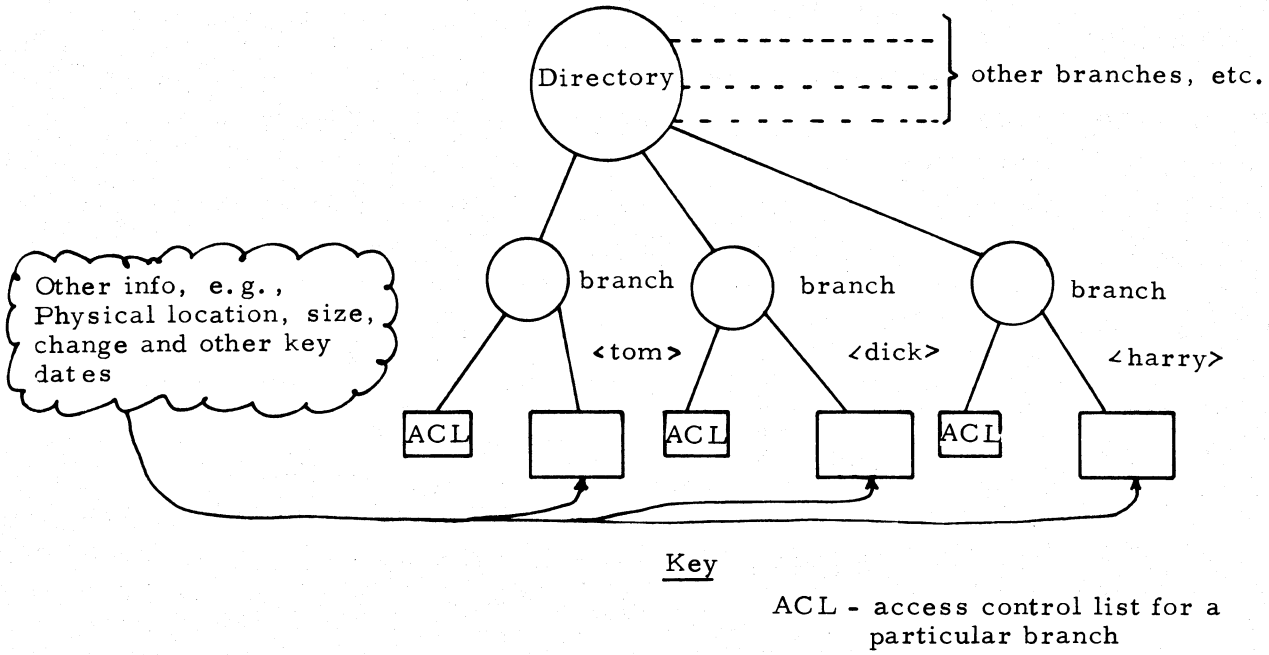
4.2.2 Some Details on Access Control Information

Now, to the details of access control information. We begin by looking at a schematic of the data structure for a directory, Figure 4-2. A directory (for our purposes here) is thought of chiefly as a list of pointers to branches. Each branch is, in turn, conceptually divided into two parts, a permission list, hereafter called an access control list (ACL) and a block of other information specific to the data of the branch, e.g., where the file is located in secondary storage, its size, etc. A schematic of these lists is displayed in part (b) of Figure 4-2. The actual storage structure is given in BG.7.

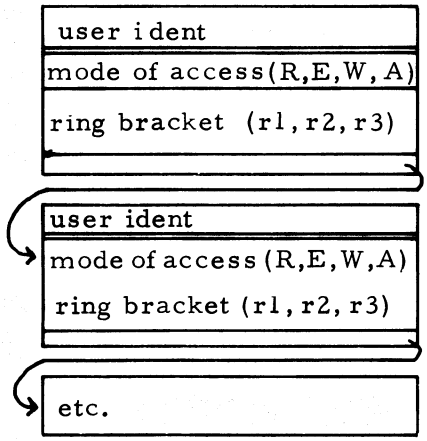
---

*The mechanics of searching and locating the desired directory is examined in Chapter 6.

**For a refresher on the role of the Fault Interceptor, refer back to Section 2.6.5 of the Guide. The design of this module is detailed in MSPM BK.3. Inspection of these details should be postponed until after study of Chapter 5.



(a) Partial view of data structure for a directory.



(b) Data structure of an access control list for an individual branch (ACL).

Figure 4-2. Schematic of Directory Data Structure

Associated with each listed user or class name,* is information that denotes the mode of access, i.e., read and/or write, etc., and a ring bracket. The latter identifies the ring(s) from which the specified access mode is permitted. Access control information may be altered only by the process(es) which enjoys write access privilege in the directory which contains the branch to the file in question. Normally, this process is the one which has responsibility for creating the segment. A subsystem writer, ssw, will typically designate his own user directory as the one to hold branches for segments which he would let others have access to. Subsequently, only processes executed by ssw (or by his proxy) would have the write access mode necessary to alter branches to such segments.

Strictly speaking, a name on an access control list in a branch is what is called a "user_id". When a user logs in, the process that is created for him, and any others which may be subsequently spawned for him during the same console session, are registered under a common user_id. The user_id is a concatenation of several components, including the user's name and his project number.

If a user is to have any access at all to a given file (segment), his user_id or a class name that includes the user_id must appear in an entry in the appropriate branch's ACL. A search will be initiated at the request of the Linker in behalf of a given process with user_id as one of the arguments.

Specifically, if the search for the segment, say <tom> leads to the branch pictured in part (a) of Figure 4-2, then access will be permitted if and only if an acceptable match can be made between user_id and a corresponding user identification in an entry of the ACL for <tom>.

Usage Attributes

Codes defining the modes of access are found in the matched ACL entry. These codes, called usage attributes, determine the kind of access to be permitted this user. A module of the Basic File System called Segment Control will employ this information in setting the descriptor field when preparing the descriptor word for the segment being acquired. Segment Control is invoked in an appropriate manner when its services are needed, e.g., by the Linker.

There are four usage attributes, each coded as on-off switch. The switches are named R (for Read), E (for Execute), W (for Write), and A (for Append). Table 4-1

*Access control lists either name individual users (user_id) or classes of users. The coding schemes for naming classes of users is explained in BX.8.00.

gives the on interpretation of the REWA switches in the typical case where the branch refers to a non-directory file (as opposed to a directory file).* These four attributes define the so-called effective mode of the segment.

TABLE 4-1

On Interpretation for the Four Usage Attributes in Non-directory Files

Attribute	Type of Segment Implied	Type of Permission
Read	Data or Procedure	Can read the contents
Execute	Procedure	Can execute as a procedure
Write	Normally data, occasionally procedure	Can truncate or rewrite existing contents (without increasing the length)
Append	Data	Can add to the segment without changing its current contents (Should be accompanied by the Write attribute)†

*Interpretation of the switches for a directory file is discussed in BX.8. Briefly: The read attribute must be on if a user wishes to examine the contents of a particular branch, e.g., to see if he is on the ACL for that branch and if so the type of access he has been granted.

The execute attribute must be on if a user wishes to search a directory to locate a particular named branch and if found to use the file to which it points.

The write attribute must be on if a user wishes to alter a branch, e.g., change access control list information, or to delete the branch entirely (and its corresponding file).

The append attribute must be on if a user wishes to add a new branch to the directory (without altering existing branches).

† Ideally the concept of the A attribute for a data or procedure segment should be fully independent of the other attributes. Thus, to be meaningful, an A attribute should carry with it an implied write privilege in the section of the segment that is appended. The GE 645 has no hardware to support this independence. As a result the A attribute by itself does not carry with it any write permission. For this reason, if the A attribute is given to a user of a segment, he should also be given the W attribute as well. Of course, this means that write permission is then given for the entire segment, not just for the append portion.

The important observation to make here, if you are a subsystem writer, is that two or more users may have entries on the same ACL with different effective modes. This can lead to a situation where, for example, the same unique copy of a data segment is acquired by two processes (active at the same time). One process is given read and write privileges to the segment, while the other is given only read privileges. This capability for the sharing of segments means it will be possible for certain key data and procedure segments of a subsystem to be under development (full access) by a subsystem writer while continuing to permit users of this system the appropriate, but limited access to such segments (e.g., Read only for the data and Read, Execute for the procedure segments).

Notice, also, that a user can have more than one process because he can have one project number, and/or more than one user_id under the same project number. This means he has the possibility of giving one of his files, say segment <a>, different effective modes for his different processes (or project numbers), thus offering the possibility where necessary of a user "protecting himself from himself".

Recall from Chapter 1, that the GE 645 address formation hardware has been especially designed to permit this "simultaneous" multi-type use of a segment. Access to a segment is not a function of the physical location of the segment in core, but is a function of the descriptor bits set in the segment descriptor word (SDW) of the particular requesting process. These bits are, of course, independent of the segment's location. Moreover, two or more active processes may share a segment in core. Each process would have an SDW pointing at the page table for this shared segment, and each of these SDW's may be set with the same or different descriptor bits.

In summary, ACL entries may be added, deleted or altered by any user who is privileged to write in the directory containing the branch to a given file. Thus, the subsystem writer who constructs a particular file will be able to select the set of valid users for each of his files and the type of access to be accorded each. The calls to the Basic File System for performing these ACL operations are described in BG.9 and the corresponding commands are described in BX.8 and BY.12.

4.2.3 Rings and Ring Brackets

The ring bracket found in each ACL entry defines the ring or bracket or rings to which the segment will belong in the process acquiring it. We initiate our discussion

by considering the simple case where, for purposes of access, a segment is associated with a single ring, deferring discussion of the perhaps more general case where a bracket of rings is involved.

Figure 4-3 reviews the ring concept for grouping the segments of a process, suggesting the idea of a set of concentric rings, each ring being identified by a number, beginning with 0. Each segment of a process, data or procedure, can now be characterized by a ring number. The numbers 0 through 3, shown in Figure 4-3, are suggestive only. Multics, in fact, provides for a maximum of 64 rings, up to 32 rings for characterizing systems programs, e.g., central supervisor (ring 0), administrative segments (ring 1), etc., and up to 32 rings available for use in characterizing user-provided subsystems and other user programs and data, i.e., rings 32 through 63.

Warning:

The fact that up to 32 rings are available to user-designed subsystems is hardly to be construed as an urgent invitation to use them. The use of each additional ring in a subsystem of course adds to the cost of programming and execution. On the other hand, the multi-ring capability is available when it is needed.

4.2.3.1 An Important Note

In the initial implementation of Multics, an attempt to maximize performance has resulted in an expedient consisting of the following simplification:

All supervisory segments reside in ring 0. A number of library routines designed to make the supervisor easier to use are placed in ring 1. In addition, user segments, which would ordinarily reside in ring 32 (the first user ring) are placed in ring 1. (Ring 32 will thus be empty.) If additional user rings are needed, they may be added, beginning with ring 33. Typically, the segments of a process will be divided between two rings, 0 and 1, with user segments sharing equal privilege with the ring 1 library routines. In the remainder of this chapter, we treat the ring system as it is eventually intended to be used, namely: user segments reside in rings ≥ 32.

4.2.3.2 Student-Teacher Subsystem Example

When more than one user ring is needed, two rings will usually suffice. As an example, a student-teacher subsystem, such as the one mentioned in Section 4.1.2, would probably require no more than two rings.

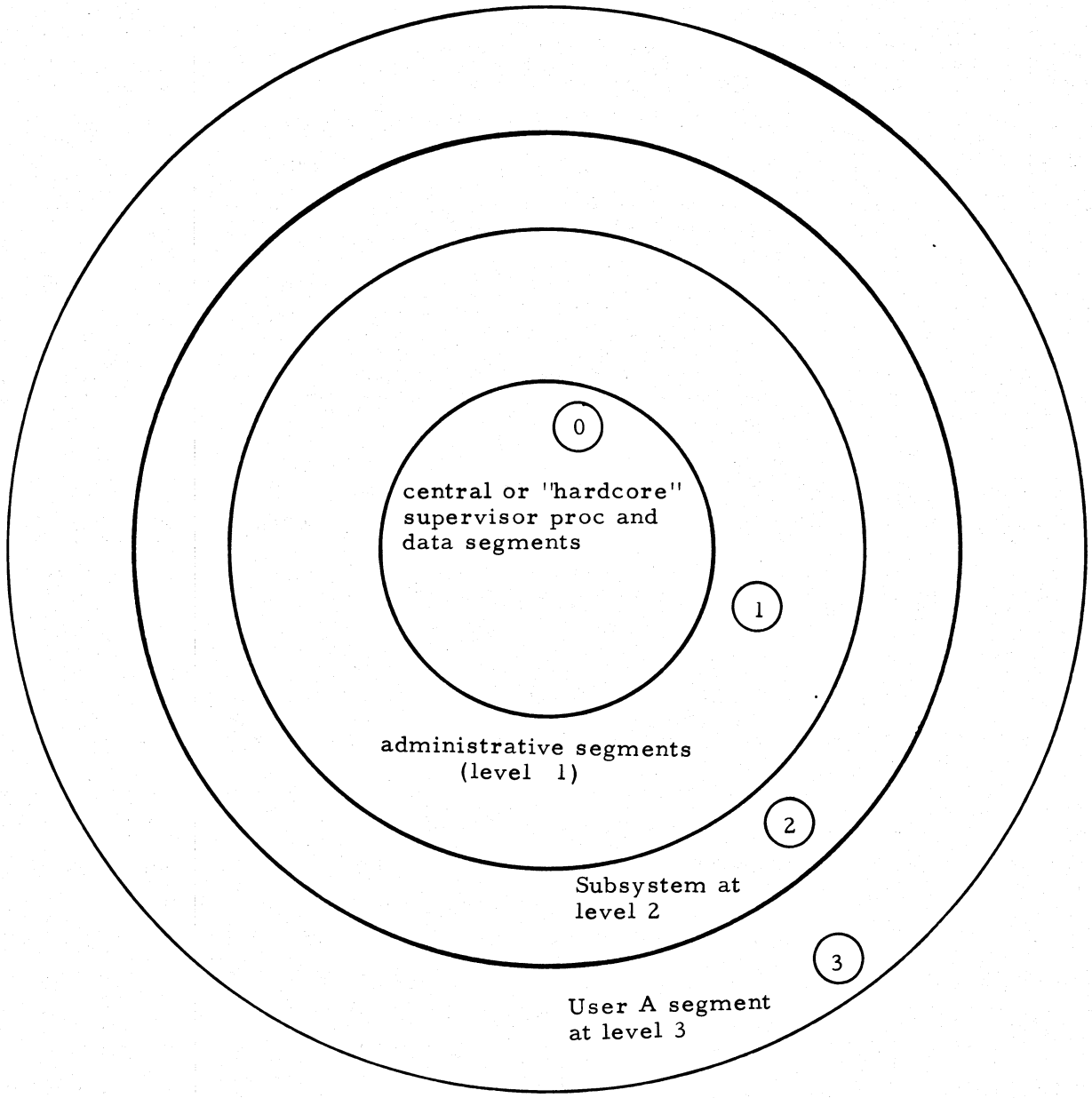


Figure 4-3. Thinking of Segment Groups Corresponding to a Set of Concentric Rings

A number of teacher-student schemes can be devised. Here is one relatively simple kind which might be used for grading of student-prepared procedures. Let us suppose teacher X has assigned the students in his Math class the task of programming a certain subroutine called < sub_stu_id>, where stu_id is any unique character string mutually agreed to by teacher and student.

Imagine that prior to the due date for this homework assignment, each student will have placed a tested version of < sub_stu_id> in X's user directory, ready to be graded. After the due date X's grading program would systematically execute calls to each of the various < sub_stu_id>'s found in X's user directory. The teacher program would somehow compare observed performance, e.g., computed results, run time, etc., with certain pre-established norms as a means of evaluating the student's work.

If < sub_stu_id> and the teacher's grading program belong to the same ring, there is no foolproof way to prevent the student's procedure from damaging the teacher's segments. In this situation, a clever student might be able to help himself to an A! Thus, upon being called, < sub_stu_id> might be so written to inspect the stack frame of the caller (the teacher) and from this information figure out a way to call on the caller, i.e., study the teacher's grading program, and determine what the right answer should be .

Here, is one of several ways to prevent this invasion of the teacher's privacy. We assume that teacher X has previously "paved the way" for each student in the class to move his < sub_stu_id> to X's user directory. To get his homework graded, each student will move his own tested version of < sub_stu_id> into X's user directory on or before the due date. Before grading each < sub_stu_id>, now in X's own directory, the teacher's program makes two crucial alterations to the branch for each student's < sub_stu_id>, via the setacl command. (The teacher can do this, since he has write access to his own directory.)

(1) In every ACL entry he deletes write access (so that neither the student author nor any "friend" of his can sneak in a change to the program after the due date and before the work is graded).

(2) X creates an ACL entry for himself (teacher_id) with Read, Execute access rights and — here is the crucial point — with the number 33 as the ring number for this segment. When the teacher program later executes < sub_stu_id> in performing the grading, it will execute as a ring 33 procedure. In this way < sub_stu_id> would not be able to gain illegal control of or inspect the teacher's segments.

Here are two details omitted in the foregoing description:

(1) The teacher "paves the way" for the student to move his < sub_stu_id> by using the file system command:

make_branch (See BX.8.05 for details.)

(2) The student moves his files when he chooses to by using the file system command:

move_branch (See BX.8.12 for details.)

The two steps above must occur in sequence. Unless the teacher has created a properly named branch for each < sub_stu_id>, (which, incidentally, he can do) a student may find he cannot successfully execute the move_branch command.

4.2.4 A Guide to the Ring Assignment of Segments

Two general principles should be kept in mind when deciding on the appropriate ring(s) for the key segments of a process:

(1) The Need to Know

A procedure < a> should have access to only those procedures and data segments necessary for < a> to do its task. Moreover, < a> should only have the mode of access to these same segments that is actually required (e.g., read, but not write, read-write, but not append, etc.). Graham's paper draws the excellent analogy with a military system of clearance. The higher the clearance (lower the ring number) the more documents one may have access to — and the fewer the number of individuals (segments) who are to be afforded such clearance.

(2) Degrees of Likely Damage

If the segments of a subsystem can be effectively segregated according to the damage which may be wrought when these segments are misused, there may be good reason for compartmentalizing the segments into two or more rings. Those segments whose misuse is likely to cause the greatest damage would be accorded the lower ring numbers. The advantage gained by placing a procedure in an inner ring is easily nullified however, if insufficient care is given to the coding of it. A procedure which can cause extensive damage when improperly called can accomplish comparable damage if it malfunctions of "its own accord". For this reason we sometimes speak about inner-ring procedures as needing to be more trustworthy. As a matter of fact, they aren't going to be more trustworthy simply by assigning them a low ring number. Below is an attempt to explain what we do mean by

trustworthiness. In one subsystem we are given two procedures, <a32>* in ring 32 and <b33> in ring 33. The likelihood that the more trustworthy <a32> will misbehave by improperly calling a procedure in ring 33 is less than the likelihood that <b33> will improperly call on a procedure in ring 32. If damage does result from an improper call, we prefer it happen in an outer ring where the damage segments will affect the fortunes of fewer users or user groups.

By the same reasoning a more trustworthy low ring number procedure is less likely to misuse a given data segment to which it makes reference (read, write or append) than will a higher ring numbered procedure. The lower the ring number of a referenced data segment, the more universal is the damage likely to be when it is misused.

For the benefit of those who might be designing a multi-ring subsystem, our discussion thus far can be summarized by three rules (essentially axioms) that are enforced by the system. We preface these rules with the following remark:

We shall often speak of a procedure as "residing in ring j", or as "executing in ring j". What we have in mind is the notion that every executing process has a state variable known as the current ring number. Conceivably, this variable could be implemented as a special hardward register. If, for instance, the procedure <s33> were to transfer control (call or return) to <t32>, we picture the "ring register" that holds the current ring number as being updated from 33 to 32. Prior to the transfer <s33> resides (or executes) in ring 33. After the transfer, <t32> resides in ring 32.

Rule 1. A procedure "residing" in ring number j should have the liberty to call any procedure segment residing in ring number j or in any ring number greater than j. The same procedure should also be permitted to make references to data segments (Read, Write or Append), as permitted by the effective mode of the particular data segment, provided the ring number of the data segment is j or greater.† The data and procedure segments in rings j, j + 1, ... etc., are said to be the domain of access for a procedure segment in ring j.

---

*By this unofficial naming scheme we hope to simplify our discussions. By appending "32" to "a" we hope unambiguously to suggest "<a> in ring 32".

†The damage caused by misuse of a data segment becomes more localized the higher the ring number of that data segment. Note, if a procedure can be trusted to use a data segment in its own ring j, it can certainly be allowed to make references to data segments in rings higher than j.

Rule 2. A procedure residing in ring j should either be denied the privilege of calling a procedure in a ring numbered i less than j (inward call), or else this access should be limited, i.e., controlled in some careful way.

Rule 3. The same procedure residing in ring j should never be given access to data segments having ring numbers less than j.

4.2.5 Ways to Recognize Attempted Ring-Crossing

If the ring model is to be implemented, it must be possible to detect and control each ring crossing that represents an inward call. In Section 4.3 we'll see that other types of legal ring crossings must also be detected and controlled, e.g., outward calls, to make inner arguments accessible to called procedures in outer rings, inward returns from outward calls, for similar reasons, and even outward returns.

From a design point of view we would like all of this detection mechanism to occur "under the surface." At least the unsophisticated user should not need to be aware of the mechanism which causes the combined hardware and supervisory software intervention at ring crossings. Certainly no special coding should be required when he, for example, executes a call to a system or subsystem procedure which happens to reside in an inner ring.

Ways could possibly be found by software alone to check for ring crossings on all calls and returns. Thus, the system could operate entirely in the interpretive mode. We are forced to reject this plan as being too expensive as a general solution. Alternatively, we could expand the standard call and return sequences by introducing additional ring-related arguments. This would prove costly enough in execution time overhead. But, how would we prevent other, strictly illegal inter-ring references by software alone?

The use of special hardware facilities which could detect all cross ring activities as faults and which would then trap to a special supervisory routine, is the only feasible approach. This is the approach used in Multics. The routine to which trapping is accomplished is called the Gatekeeper (MSPM document BD.9.01).

4.2.6 Two Hardware Approaches Have Been Designed

Using current GE 645 hardware, the protection mechanism is achieved by having the supervisor maintain separate copies of the descriptor segment for each ring used. The per-ring descriptor segments differ only in the access control bits of

corresponding segment descriptor words. If we were to look, say, at the descriptor segment for ring j, we would see that special fault-inducing access control bits are preset (by the basic file system) in SDW's that point to segments of other rings $k \neq j$. One type of fault "detects" cross-ring references to procedure and data segments of inner rings. Another type of fault detects references to procedures residing in outer rings.

A more efficient scheme has been proposed for a future implementation of Multics wherein the need for multiple copies of the descriptor segment would be eliminated. The proposal depends on altering the GE 645 hardware in the following way: First, a six-bit ring register would be added on each processor to the set of registers referred to as the "machine conditions". The ring register would at all times hold the ring number for the currently executing procedure segment. Next, the format of the segment descriptor word (SDW) would then be revised to include ring number identification for the segment coded in the SDW. A new type of hardware faulting would occur when the address formation mechanism, upon reaching the SDW, detects certain kinds of ring crossing based on a comparison between the contents of the ring register and the coded ring number in the SDW.

Section 4.3 gives a more detailed explanation of ring-crossing detection. The discussion is based entirely on current hardware.

4.2.7 Access and Call Brackets - Motivation

The simple ring model so far described is fine for protection, but it is, in fact, too good! The model implies that every segment of a process be associated with a single and fixed ring number. Two consequences of this simplicity turn out to be too restrictive. In order to circumvent each of these restrictions, when necessary, the Multics ring model has been made a bit more complicated.

4.2.7.1 The First Restriction

Consider a service routine which would be made available for use by ordinary user and supervisor alike. Suppose a single ring number is assigned to this routine. It would appear that either the supervisor or the user would invoke a ring-crossing fault* in calling this service routine, even if we were considering a two-ring model.

*Strictly speaking it is also possible to avoid these ring crossings by making multiple copies of each service routine one copy assigned to each ring in which a call to that routine is made. This approach has not been taken in Multics.

Now, whatever overhead is involved in executing this ring crossing (and we shall see these details in Section 4.3) seems unnecessary. A service routine (or at least some service routines) can certainly be designed to take calls from segments in a wide class of rings because it can be written as a pure procedure, i.e., with Write access to it prohibited. Hence, there is no reason why such a routine should be subject to damage or should cause any damage during its normal use. Extension of the model to three or more rings only strengthens the argument. It would, therefore, seem worthwhile if use of such service routines could be "exempt" from the ring-crossing overhead. The solution arrived at in Multics is to let each segment be optionally characterized by an access bracket instead of a single ring. The bracket then constitutes a band of rings such that when a reference is made to one of these segments (data or procedure) from a procedure whose ring number is within the access bracket, no ring crossing faults are invoked; (manifesting the fact that no protection is needed). A procedure called in this way is said to execute in the ring of its caller. All EPL library routines are of this type, for example.

4.2.7.2 The Second Restriction

It's easy enough to prevent outright any outer-ring procedure from calling any inner ring procedure since a ring crossing will be induced in the attempt and the fault handler can then declare the caller "guilty". The real challenge is to provide a suitable screening methodology so that some inward crossings can be regarded as legal, possibly subject to some further checks, while other inward crossing attempts can be rejected as truly illegal. This type of control, for example, is found necessary in the design of the Multics supervisor. Thus, certain modules of the basic file system that "reside" in ring 0 are designed to be called either by other ring 0 routines (with safety assured) or by certain routines in ring 1. However, calls from procedures in rings higher than 1 are considered unsafe and must be rejected. To achieve this level of control over inward calls in Multics, it is possible to associate with any procedure, when needed, a call bracket, representing a band of rings immediately outside the access bracket. The call bracket of a segment <a> would identify the rings from which a calling procedure <b> is permitted to call <a> via an inward ring-crossing. If <b> executes in a ring outside the call bracket, the fault handler rejects the call as illegal. If <b> executes in a ring within the call bracket, the fault handler will consider the call to be potentially OK and will then, before accepting the call as legal, perform a further check to be sure the target address is a specially declared entry point in <a>, called a gate. The concept of gates will be discussed in Section 4.3.

Figure 4-4 summarizes the forgoing ring bracket concepts. The hypothetical case is considered for a target procedure <a> whose access bracket is rings 32, 33 and whose call bracket is rings 34, 35.

4.2.7.3 Access Bracket - Details

In place of a single ring, k, any data or procedure segment may be optionally characterized by the band of rings from k to ℓ, where $0 \leq k \leq \ell \leq 63$. The band is represented by the pair (k, ℓ), and is called the access bracket. Ring crossing faults occur only when the ring of the executing procedure lies outside the access bracket of the target segment. The intended access discipline (a) for procedure targets, and (b) for data targets, is spelled out below and summarized in Table 4-2. For this discussion we picture some procedure, whose ring number is r, is making an attempted reference to a target segment. We speak of the referencing procedure as executing in ring r.

(a) For target procedure segments characterized by the access bracket (k, ℓ), and having no call bracket the following is to be true:

(1) A referencing procedure executing in a ring $r \leq \ell$ has what we can call "ring access" to the target. This means that actual access is governed by the effective mode of the particular target. Cross-ring (outward) faults are induced and detected in these instances only when $r < k$.

(2) Access to the target is completely denied to any procedure whose ring number lies outside the access bracket of the target, i.e., has ring number $r > \ell$. Segment faults* are detected in all instances where $r > \ell$.

(b) For target data segments having an access bracket (k, ℓ), the interpretation is quite different:

(1) A referencing procedure executing in a ring $r \leq k$ will have access to the target governed entirely by the target's effective mode. No ring crossing faults will be induced during an outward data reference from a ring $r < k$.

(2) Procedures referencing the target from rings $k + 1, k + 2, \ldots, \ell - 1, \ell$ will have access restricted. No writing in this segment will be allowed by the executing procedure even if the W bit in the effective mode is on.

*The significance of this type of fault (directed fault 3) will be discussed later.

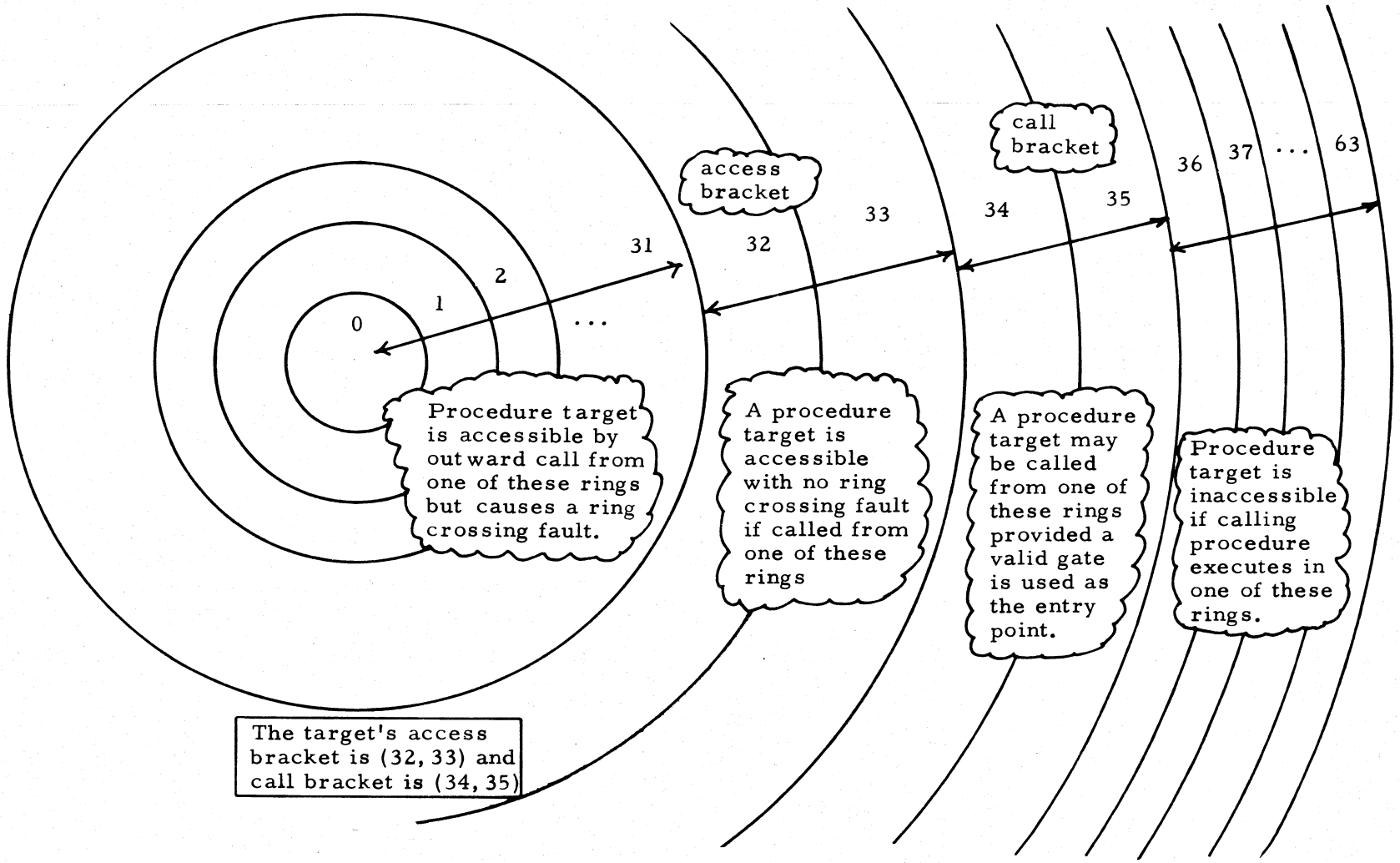


Figure 4-4. Access to a Procedure Target <a>

(3) Procedures attempting to make data reference to the target from rings $\ell + 1$, $\ell + 2$, ... , 63 will be denied all access. Segment faults are detected in all instances where $r > \ell$. For example, if <data> has the access bracket (35, 38), and if the effective mode for <data> is R, W, (i.e., read and write), procedures executing in rings ≤ 35 will be permitted to read and write in <data>, procedures executing in rings 36, 37, and 38 will be permitted read only privileges in <data>, while all access to <data> will be denied to any procedure executing in rings 39 through 63.

TABLE 4-2

Access Discipline for Procedure and Data Targets

Key: Referencing procedure executes in ring r

Target has effective mode = REWA

access bracket = (k, ℓ)

Target Type	r < k	r = k	k+1 ≤ r ≤ ℓ	r > ℓ
Procedure	REWA (but ring crossing fault is induced)	REWA	REWA	~~REWA~~ (all access denied, segment fault is induced)
Data	REWA	REWA	~~REW~~A (write access denied)	~~REWA~~ (all access denied, segment fault is induced)

4.2.7.4 Call Brackets - Details

A call bracket may be added to the access bracket in characterizing any procedure (but not a data) segment. If the pair (k, ℓ) is the access bracket, then the additional call bracket is, for economy of coding, characterized by a third number, m, such that $\ell < m \leq 63$. The call bracket is then the band of one or more rings from $\ell + 1$ to m, inclusive. Ring-crossing faults occur whenever the ring of the executing procedure is within the call bracket of the target, and segment faults occur when the ring of the executing procedure, r, exceeds m, i.e., lies outside the call bracket.

The intended access discipline here is as follows: As before, target data or procedure segments are accessible (without induced ring-crossing faults) to any procedures or data segments whose ring number lies within the access bracket. Target data segments are entirely inaccessible to procedures whose ring numbers are greater than the access bracket. Access to target procedure segments may be permitted if the referencing procedure's ring number lies in the target's call bracket, i.e., if $\ell < r \leq m$. Permission is granted in such cases only if the entry point has been established as a gate for inward calls. Gates are specially declared. When the segment's author declares a given entry point to be a gate, the compiler or assembler would then provide an entry in the linkage section having a non-standard but recognizable format. The Gatekeeper which handles the wall crossing fault for this case determines whether the faulting procedure has, in fact, been aimed at a gate of the target procedure by examining the format of the entry point. The storage structure of gates is detailed in Section 4.3.6.

4.2.7.5 Ring Brackets - Examples

A ring bracket* is recorded in the branch for each segment. It consists of a 3-tuple of numbers. The form of the three tuple depends on ring characterization for the segment as shown below:

Ring Characterization	Form of the 3-tuple
(a) Single ring of access, r.	(r, r, r)
(b) An access bracket (k, ℓ), but no call bracket.	(k, ℓ, ℓ)
(c) An access bracket (k, ℓ) and a call bracket (ℓ + 1, m).	(k, ℓ, m)
(d) A single ring of access r and a call bracket (r + 1, m).	(r, r, m)

Some typical uses of ring brackets for system routines are illustrated in Table 4-3. The examples should help you see how ring brackets would be used in characterizing user-created segments.

*The ring bracket is copied from the ACL entry in the file branch at the time the segment is first acquired by the process and subsequently kept in a more accessible per-process table called the KST (known segment table).

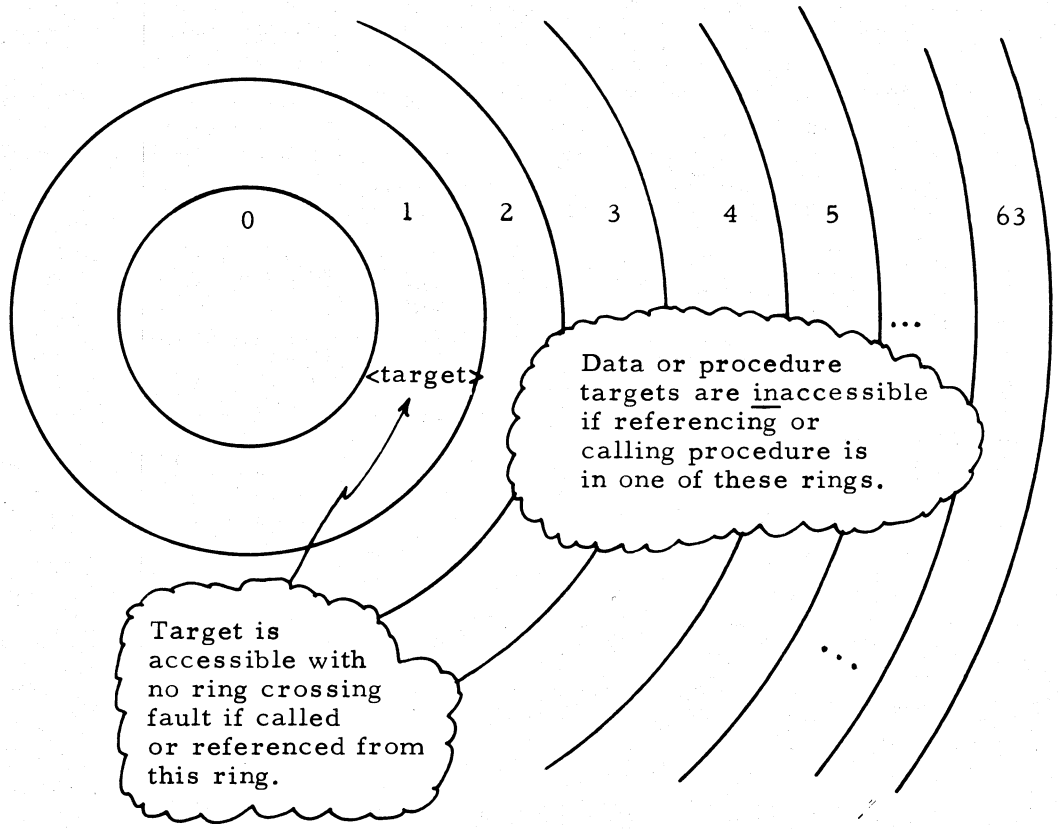
TABLE 4-3

Examples of Ring Brackets used in the System

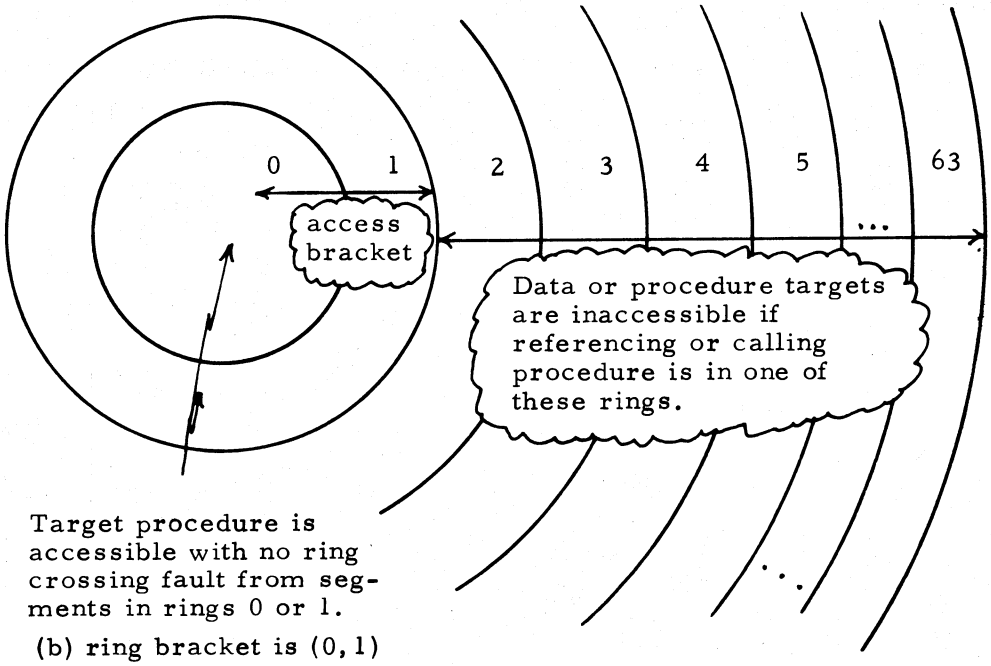
Item	Ring Bracket	Interpretation
1	0, 63, 63	Every procedure has access to this segment without invoking a ring crossing fault (target executes in ring of its caller).
2	0, 1, 63	Procedures in rings 0 and 1 can call without intervention. Procedures in rings 2 through 63 can call via inward ring-crossing fault, but desired entry point must be a gate.
3	1, 1, 63	A ring 1* procedure that may be called, as in item 2 above from rings 2 through 63.
4	0, 0, 1	A ring 0 routine. Inward calls are permitted from ring 1 via ring crossing fault, etc. Calls from rings 2 through 63 are rejected.

Figure 4-5 gives pictorial interpretation for two additional ring brackets (1, 1, 1) and (0, 1, 1). What is the ring bracket characterization for (the rather exotic case of) <a> in Figure 4-4 ? Answer: (32, 33, 35).

*A point of possible interest is that ring 0 routines may not execute outward calls. Hence if the target has a protection list (1, 1, 63), a ring 0 routine cannot call it directly.



(a) ring bracket is (1,1,1)



(b) ring bracket is (0,1)

Figure 4-5. Access to a target procedure or Data Segment

4.3 MONITORING AND CONTROLLING RING CROSSINGS FOR NORMAL CALLS AND RETURNS

We are now ready to see how ring access control has been implemented in Multics. First, we amplify three important implementation concepts. (1) A process can have, if necessary, up to 64 rings; user rings are numbered 32 through 63. (2) For each ring in which a process executes there is actually a separate descriptor segment. Ring 0 supervisory routines create and maintain these segments as needed.* The per-ring descriptor segments differ only in the way fault-inducing bit patterns are placed in the descriptors. The bit patterns are set so as to trap during address formation on all inward data references and on all inward or outward procedure references. (3) There is also a separate stack segment, called <stack_n>, created for each ring in which the process executes. Here, n is one of the integers 0 through 63 (or, strictly speaking, 00, 01, ..., 63). Supervisory routines are responsible for creating these stack segments, † but once created they are to be treated as ordinary data segments.

4.3.1 Function of the Individual Descriptor Segment

To see how the individual descriptor segments serve in the role of ring-crossing detectors we shall discuss a (hypothetical) process in miniature suggested by Figure 4-6. There are four ordinary data segments <d0>, <d1>, <d32> and <d33> and four procedure segments <p0>, <p1>, <p32>, and <p33>, one of each in the four utilized rings, 0, 1, 32, and 33. Also shown are the four stack segments which, in matters of protection, are to be considered as ordinary data segments. We do not show the four descriptor segments because these are not directly accessible to the user. The use of two user rings is purely for illustrative purposes and is not to be construed as typical.

Figure 4-7 is a detailed view of access control bits 30-35 for one of the descriptor segments (ring 32), showing how they could be coded in each SDW so as to detect ring crossings.‡ Dashed lines emanating from the SDW's indicate ring crossings that are detected, causing traps to the Fault Interceptor module. Inward crossings, e.g.,

---

*Chapter 6 gives an elaboration adequate for initial needs of the subsystems writer. Of course, subsystem and user procedures for ring i > 0 will be allowed no direct access to any descriptor segment.

†Details of the stack segment creation may be found in BD.9.01.

‡For a refresher on the hardware characteristics first review pertinent parts of Chapter 1 of this Guide, especially Table 1-1.

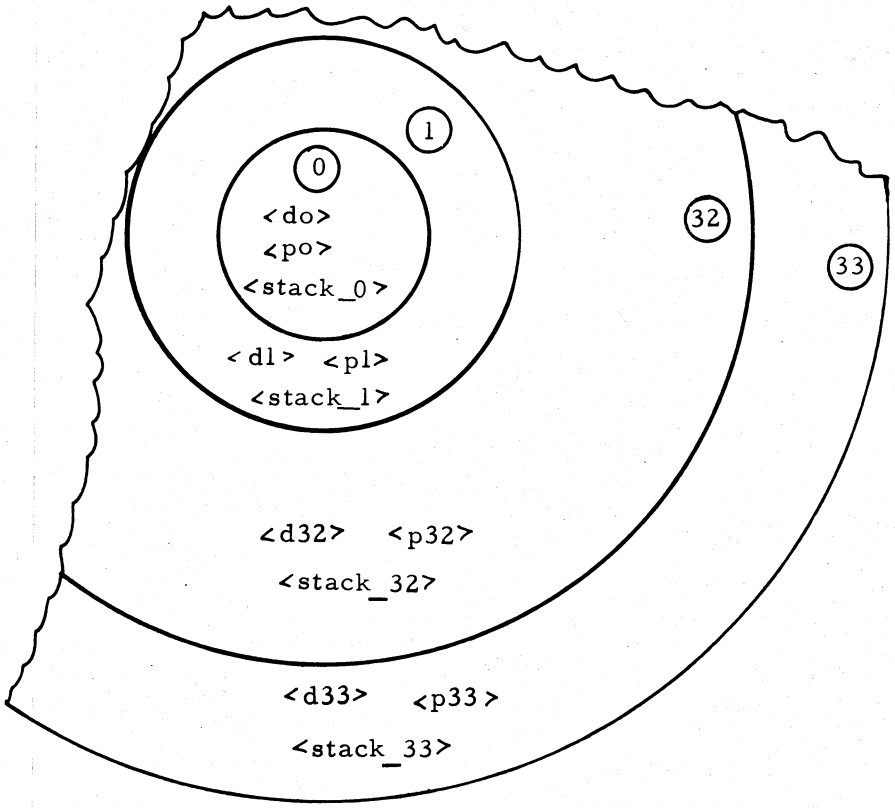
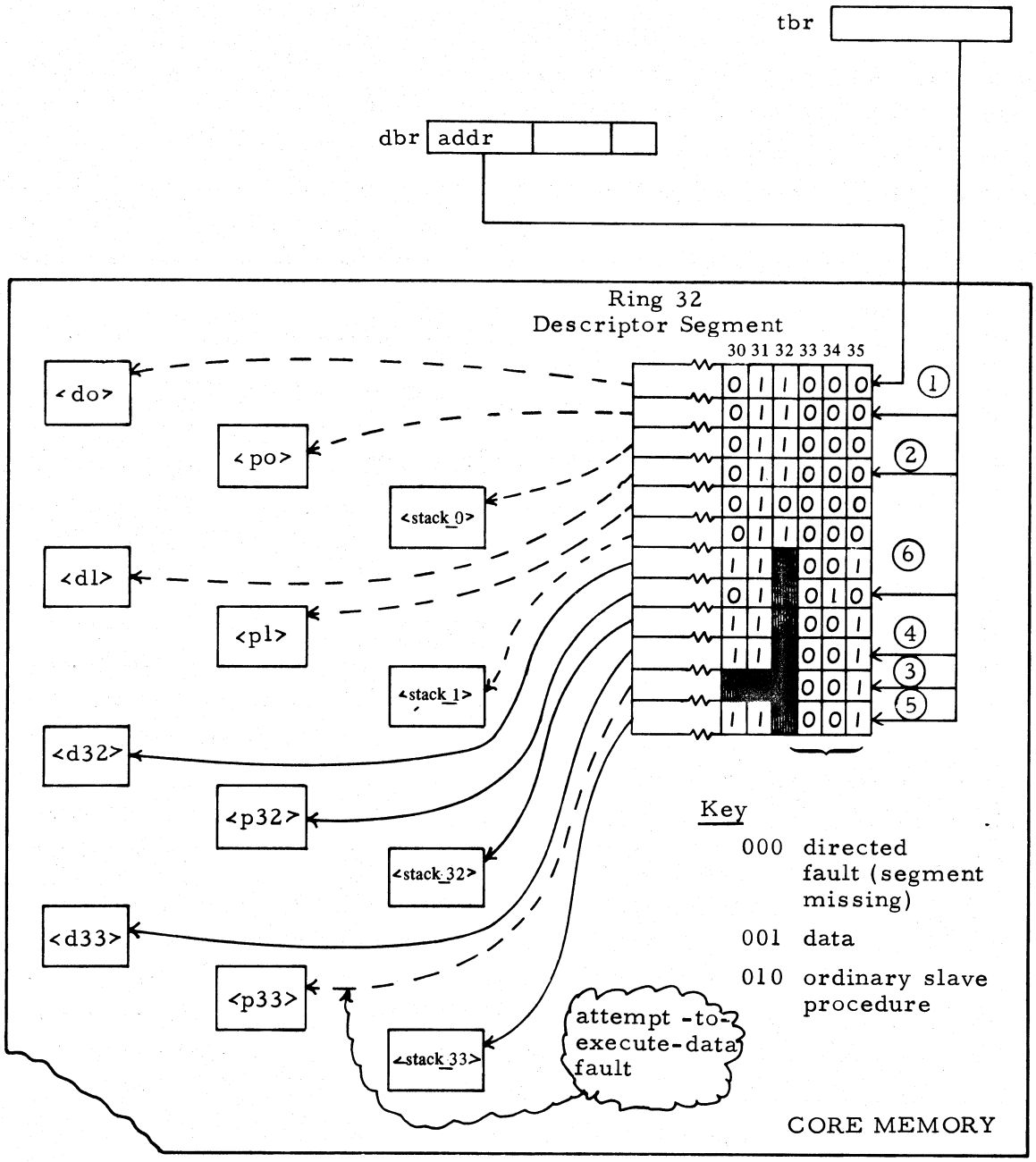


Figure 4-6. A Process in Miniature (in four rings)

line (1) to an inner ring procedure, <p0>, or line (2) to an inner ring data segment, <d1>, cause directed faults. † Outward crossings to procedures, e.g., line number (3) to an outer-ring procedure <p33>, are detected by attempt-to-execute-data faults. To achieve this type of fault, bits 33-35 for <p33> are preset to suggest data. Subsequent attempts to execute an instruction fetch will then cause a fault that forces control to the Fault Interceptor. Outward crossings to data segments are deliberately not detected, e.g., lines (4) and (5) to the outer ring data segment <d33> and <stack_33>, respectively.

†Strictly speaking, two types of directed faults are used. More about the distinction between these is given later in this section.



Consult Table 1-1 for a refresher on the significance of descriptor bits 30-35.

Figure 4-7. Using the Descriptor Segment as a Ring Crossing Detector

Figure 4-8 shows all four descriptor segments of the process ordered by ring number. Bit details of the descriptor fields in the SDW's are now replaced by schematic markings. Note, that the order in which the segments are listed in each descriptor segment of the process must be the same, in order that each segment retains the same segment number from ring to ring. The particular ordering of the segments within the descriptor segment is, however, of little concern to us. Postpone until Chapter 6 any curiosity you may develop as to how and when these access control bit-fields are preset in the various descriptor segments; such knowledge is not needed now.

This is a good time to observe why, for simplicity, we have chosen not to display SDW's for the procedures' linkage segments in the above example. Recall that entry points to procedures are kept in the corresponding linkage segment. If there is to be a change of rings in a procedure call, the ring crossing must be accomplished while executing the transfer instruction used to reach the target's linkage segment (entry point). For this reason, the ring bracket for a linkage segment is always identical with its corresponding "text" segment. Subsequent transfer from the linkage segment to the target pure procedure would never cause a ring crossing. We see, therefore, that Figure 4-8 could have been made to appear more realistic, but not too much more illuminating, if we had included SDW's for the linkage segments. They would be given schematic markings identical with those of their corresponding procedure segments.

An actual crossing over from one ring to another will take place only if a master mode supervisory routine (ring 0) is called to execute the privileged instructions necessary to "switch" descriptor segments, i.e., alter the contents of the descriptor base register (dbr) to point it at the descriptor segment of the target ring. Responsibility for calling this dbr-switching routine rests with the Gatekeeper. This is the module described in Section 4.3.2 which takes charge as a result of all attempted ring crossings.

There is no possibility that a user can either write his own master mode routine to switch dbr values or manage to somehow gain direct access to the routine that does the dbr switching and thereby circumvent the Gatekeeper. Below we state why:

Recall, mastermode is characterized by a bit that is set in the SDW for that procedure. Mastermode routines must be ring 0 because the BFS module (Segment Control) which is responsible for setting SDW words will set the mastermode bit ON for ring 0 procedures only. Moreover, no user is able to create files which have

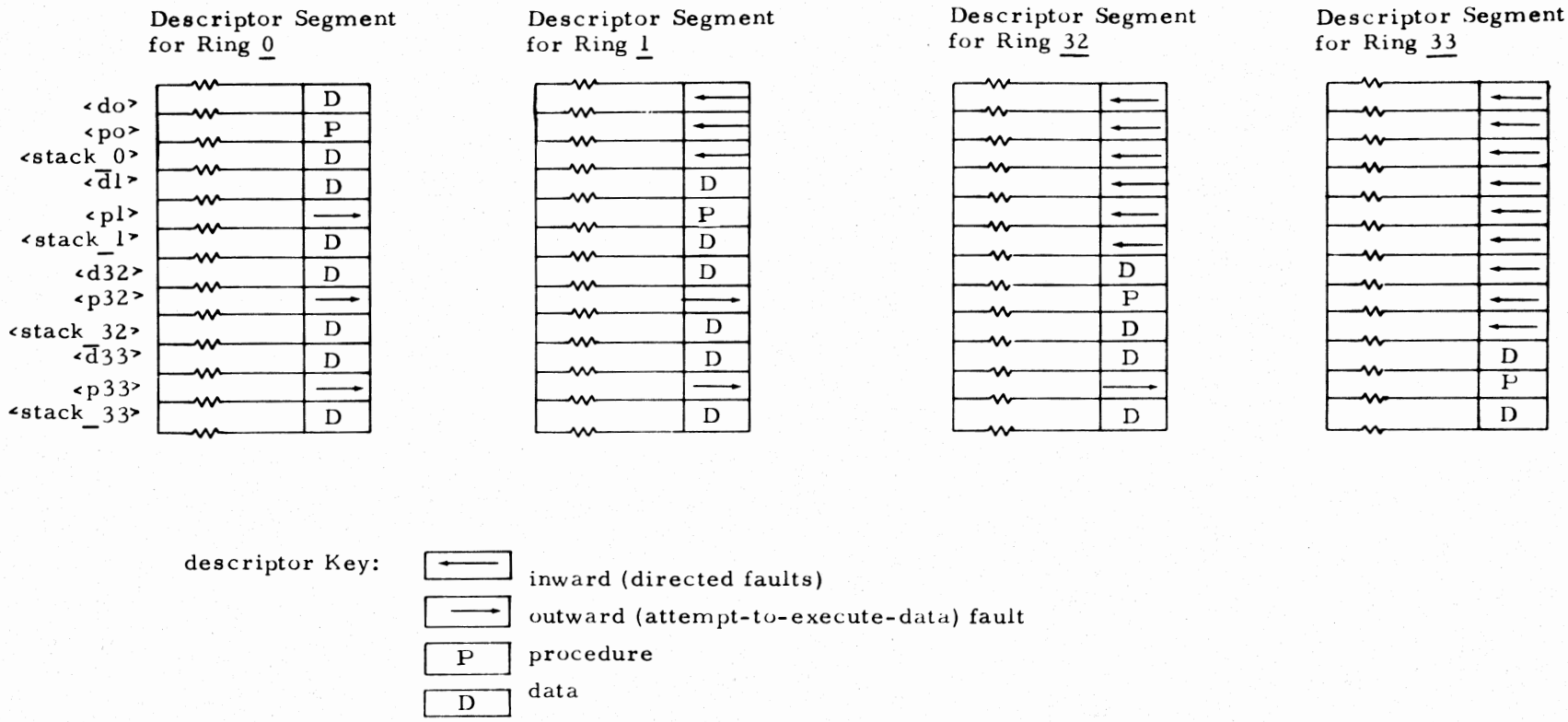


Figure 4-8. Showing all four Descriptor Segments

ring brackets that include ring 0. This is because the request to set an ACL entry, which is aimed at the Access Control module of the BFS, is screened. The lowest ring bracket value which can be posted by a user is the ring number from which his request to set an ACL entry is issued. This value is always greater than zero.

4.3.1.1 Ring Complexity of Subsystems

It is hard to say how often a subsystem will be designed to execute in more than one user ring. When such subsystems are designed, it is a safe bet that most segments written for the user rings will be characterized by single-ring ring brackets. Rarely will access and call brackets be employed and even more rarely will complicated patterns of access and call brackets be used. Since this facility is available, however, there will always be some subsystem designers who, if only to satisfy curiosity, will want to understand how more exotically-protected segments might function in a Multics subsystem. The next two sub-sections are dedicated to these avid readers. Others may wish to skip directly to the Gatekeeper.

4.3.1.2* Determining the Ring of Execution for a Segment whose Ring Bracket contains an Access Bracket

A good question to ask is: In which of the rings within a segment's access bracket will a particular segment execute when it is called? There are three cases to be considered. We shall assume <a> is the calling procedure now executing in ring r, and that <b> is to be the called or target procedure whose ring bracket is (k, ℓ, m) such that $0 \leq k < \ell < m \leq 63$.

Case (1) $k \leq r \leq \ell$. (The ring of the calling procedure lies within the access bracket of the target procedure.) <b> will execute in ring r. No ring crossing fault will be triggered.

Case (2) $r < k$. (Outward fault. The ring of the faulting procedure is less than k.) <b> will execute in ring k, the innermost ring of the target's access bracket. The design rationale for this choice is necessarily arbitrary: Pick the ring "nearest to the caller".

Case (3) $\ell < r \leq m$. (Inward fault. The ring of the faulting procedure lies within the call bracket of <b>. <b> will execute in ring ℓ, the outermost ring of the target's access bracket.) (Of course, the desired entry point must

*This section may be skipped over during a first reading without loss of continuity.

also be found to have the format of a gate.) The design rationale for this choice is again: Pick the ring nearest the caller, because it is also the ring that will involve the least risk.

If used properly, access brackets may increase the flexibility and efficiency of an otherwise complicated multi-ring subsystem, i.e., avoiding the overhead of ring crossing faults where protection measures are no longer needed. However, there are some pitfalls. If access brackets are not chosen to be functionally meaningful, superfluous ring crossing faults can occur. Thus, the unwise subsystem designer could, in practice, select a set of straddling rather than coinciding access brackets for procedures that must communicate with one another. The superfluous fault which can occur in such instances is an inward fault and, if unexpected, the supervisor would have no choice but to abort the process.

The following case will illustrate what happens when the access bracket facility is improperly (nonsensically) applied. The case is for an elaborate subsystem having segments with ring brackets shown below:

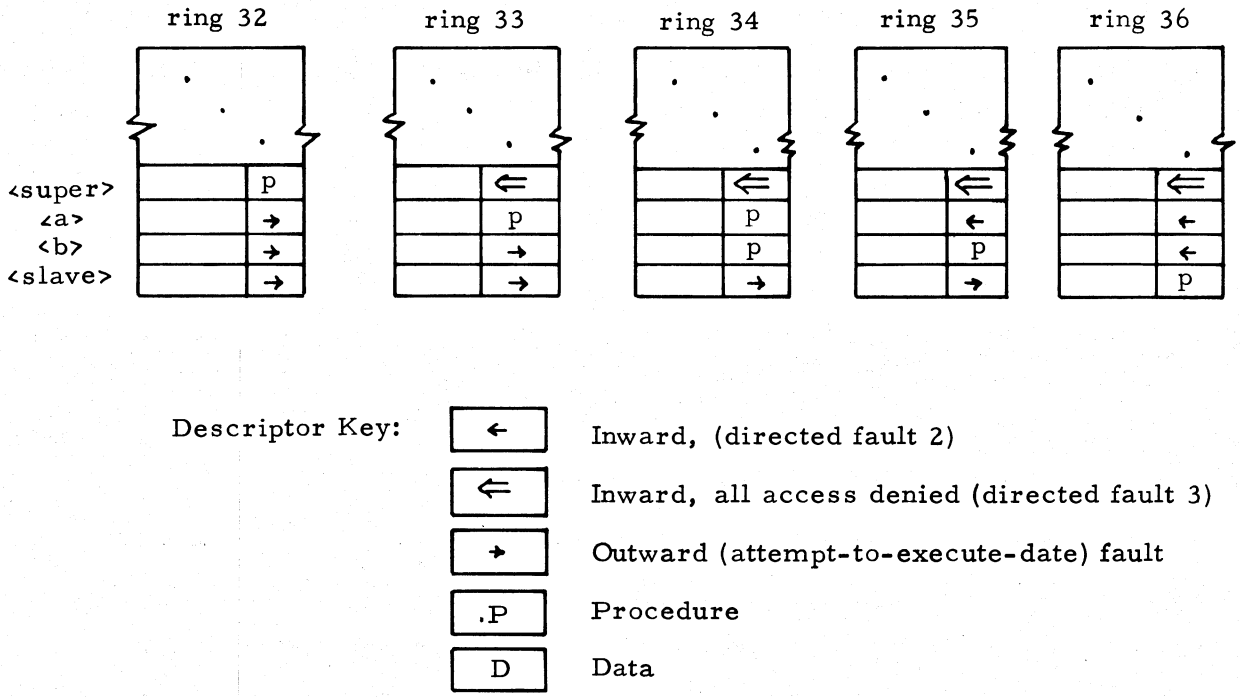
Segment	Ring Bracket
<super>	(32)
<a>	(33, 34, 36)
<b>	(34, 35, 36)
<slave>	(36)

Note: Access brackets for <a> and <b> straddle one another.

Figure 4-9 schematically illustrates the SDW's for each of these four segments in the descriptor segments for rings 32 through 36. (The descriptor key for this figure is an expansion of the one given in Figure 4-8. The significance of the new symbol (⇐) is explained in a subsequent paragraph.)

Now, consider four "case histories" shown in Figure 4-10 using Figure 4-9 and the above ring brackets as a reference. Each case is a possible chain of two calls among the segments.

In case histories (1) and (3) we see calls from <a> to <b>. An outward fault occurs in the first history because <a> happens to be executing in ring 33 rather than 34. A similar situation arises in comparing case histories (2) and (4) where calls from <b> to <a> occur. In the latter history an inward fault occurs because <b> happens to be executing in ring 35 and not 34. If the subsystem designer has failed to anticipate this event by declaring the proper entry point in <a> as a gate, the resulting fault can actually be fatal to the process.



Access bracket for <a> is (33, 34) and for <b> it is (34, 35)

Figure 4-9. Illustrating Segment Descriptor Words for Segments Having Access Brackets

Case History		Ring number of caller	Ring number of target	Governing† descriptor	Comment
(1)	<super> ↓ <a> ↓ <h>	32 33	33 34	→ fault → fault	superfluous
(2)	<super> ↓ <b> ↓ <a>	32 34	34 34	→ fault P	
(3)	<slave> ↓ <a> ↓ <b>	36 34	34 34	← fault P	
(4)	<slave> ↓ <b> ↓ <a>	36 35	35 34	← fault ← fault	superfluous and possibly disastrous

† Means the descriptor of the target in the descriptor segment for the ring of the caller.

Cases 1 and 4 show calls between segments whose access brackets straddle one another.

Figure 4-10. Cases of Superfluous Ring Crossing Faults

4.3.1.3* More Details in the Interpretation of Directed Fault 3 (All Access Denied)

There are, in fact, two types of directed fault codes used to represent attempted inward crossings, directed fault 2 and directed fault 3. The former, when detected, corresponds to a possibly valid inward call or inward return, as from a procedure whose ring number is within the target's call bracket. The latter, when detected, is interpreted by the Fault Interceptor to mean all access denied. This type of inward crossing, by being handled via a separate fault, can be rejected out-of-hand. The overhead of incurring the Gatekeeper's services to interpret this illegality is thereby avoided. In Figure 4-9 we introduce the special symbol [⇐] to mean directed fault 3 (i.e., all access denied), henceforth letting the symbol [←] mean, specifically directed fault 2.

The Basic File System also sees to it that all-access-denied fault codes are preset in the SDW's of all data segments in higher ring numbered descriptor segments. For illustrative purposes such bit coding was employed in Figure 4-7 to represent the SDW's for <d0>, <stack_0>, <d1>, and <stack_1>. Also, in that Figure we chose to code the SDW for <p0> as all-access-denied (directed fault 3) while for <p1> we coded a directed fault 2.

4.3.2 Management Control over Inter-ring Crossing (The Gatekeeper)

The Fault Interceptor calls a special ring 0 module, called the Gatekeeper, to exercise positive control over all inter-ring calls and returns. An understanding of the Gatekeeper's role and of some of the detailed steps which it carries out or oversees is an ultimate necessity for the sophisticated subsystem writer. We shall attempt to describe most of the important points about the Gatekeeper's tasks, but will not always explain them in the order they are carried out. We are more concerned with motivating and explaining the issues of "Gatekeeping." Succeeding subsections are divided arbitrarily into a discussion of problems faced by the Gatekeeper and how they are solved.

To make its tasks easier the Gatekeeper first determines which of five types of inter-ring accesses ("wall crossings") is being attempted. The five categories are:

(a) Inward calls,

(b) Outward returns,

*This section may be skipped out during a first reading without loss of continuity.

(c) Outward calls,

(d) Inward returns,

(e) Other access attempts (illegal).

The five-way resolution is relatively simple to achieve. The details are given below:

(1) Inward versus outward ring-crossing attempts are actually distinguished by the Fault Interceptor. Depending upon the type of fault (directed Fault 2 or attempt-to-execute data), the control is directed to one of two appropriate entries into the Gatekeeper, one for inward attempts, one for outward attempts.

(2) Calls versus returns are distinguished by examining the faulting instruction to see if it was a tra (call) or an rtcd (return). If neither, it is an illegal request and the Gatekeeper returns an error code to its caller, the Fault Interceptor. If the faulting instruction (on an inward crossing) is a tra, but if after checking the target's linkage section, the Gatekeeper sees the entry is not a gate, another error code is returned to the Fault Interceptor.*

4.3.2.1 Outward Versus Inward Calls - (Motivation)

It may have occurred to you to wonder if both inward and outward calls are equally useful in subsystem design. (There were none allowed, for instance, in CTSS, although a user could always call in to the supervisor.) As we shall see later, outward calls that carry argument lists generally incur a higher overhead, because calling arguments must be copied into the target ring. For this reason subsystem designers may wish to minimize their use of outward calls.

There is one type of relatively inexpensive outward call which is likely to prove very useful in the design of multi-ring subsystems. This is an argumentless call to an input responder routine which an ordinary user would make to "enter" to a subsystem. We picture here that a subsystem "X" has a special outer-ring (say ring 33) procedure segment known as <X_listener>. Whenever, after login, the user wishes to issue a series of commands in the language of subsystem X, he first issues an outward call to ring 33. (We are presuming that the user's process executes in ring 32 following login.) <X_listener> now functions as an input loop to accept subsequent commands. After interpreting each of these commands, <X_listener> issues appropriate (inward) calls to other modules of subsystem X. Since all the sensitive modules are in inner rings, there is no danger that the user can misuse or abuse the privileged segments of X.

*Remember, the Gatekeeper is spared from having to examine inward calls from a procedure that is executing "outside" the target's call bracket or, if the target procedure has no call bracket, from outside the access (or single-ring) bracket.

Another possible use of outward calls arises in cases like the teacher-student subsystem that was suggested earlier in Section 4.2.3. In this type of system the teacher in his grading process acquires and executes a student-written procedure, making sure before executing the student's procedure to "give it" a higher ring number than the teacher's segments. The call to the student's procedure then becomes an outward call.

4.3.2.2 Gatekeeper - After Determining Type of Valid Wall Crossing

The Gatekeeper performs several tasks in handling outward calls, and inward returns, etc., which guarantee the safe handling of information passed to or from inner-ring procedures from or to those in outer rings. For example, in handling an inward return for a faulting procedure <p>, it is necessary to be sure that the return location specified by <p>'s rtcd instruction is in fact, the one supplied by the inner-ring procedure that called <p>. Without this check, the outer ring could, in the disguise of a return, force an entry at any point in any inner-ring procedure, thereby defeating the protection mechanism. The technique used by the Gatekeeper to forestall such disasters is to save a copy of the return location at the time of the outward call to <p>, in a special ring-0 data base which is inaccessible to <p>. Later, the Gatekeeper will insist on a match between the safe-stored return location and the one used by <p> in its faulting rtcd instruction. If no match, the inward return will be declared invalid by returning a suitable error code to the Fault Interceptor.

We give one final example to see the kind of business the Gatekeeper is involved in before we proceed to the details at the bookkeeping level: During an outward call, the argument list and the individual arguments may very well be found in data segments accessible to the caller, but not to the target procedure. What to do?

In keeping with the Multics protection philosophy, any procedure of an inner ring, say 32, is free, and at its own risk, to copy data that is accessible to it into a data segment of any outer ring. Therefore, in an outward call, if argument lists and/or arguments are used that belong to an inner ring, but accessible to the faulting procedure, it should be perfectly OK to allow the copying of these into an outer-ring segment, putting the arguments "within reach" of the target segment.

Now, when a user writes a procedure <a> that calls on some procedure <b>, he should not, in general, have to know in advance whether <b> will be in an outer ring or, for that matter, in an inner ring. So, as a matter of design philosophy, the writer of <a> cannot and will not be asked to code the task of copying the argument values to the target ring. Instead, the Gatekeeper takes care of this chore, relieving the programmer of this nasty responsibility for argument management. The Gatekeeper gets a helper (a procedure in ring 0) called <arg_pull> to do the copying.

<Arg_pull>, in order to do its job properly, expects an argument list especially embellished with pointers to data descriptions. I.e., it cannot properly copy data without knowing its format. Further details on the required format* of these argument lists will be given in Section 4.3.5.

The Gatekeeper, also, performs the important function of validating arguments for inward and outward calls. It sees to it that every argument list element and every argument involved in such calls is indeed accessible to the faulting procedure. The basic principle that is followed here is: If a procedure, by virtue of its executing ring number, is not privileged to access a piece of data directly, that procedure should not be permitted to circumvent this restriction by getting help from another procedure which would behave either (a) as an unwitting accomplice (target of an inward call) or (b) as a deliberate accomplice (target of an outward call). Remember, the faulting procedure is ordinarily free to designate anything at all (any virtual address) as an argument pointer.

4.3.2.3 On Inward Calls

A calling procedure could in theory specify argument pointers to data objects for which the caller does not have ring access, but to which the target procedure does have ring access. We see that an effort must be made to check all argument pointers passed "inwardly" to validate that the caller actually had ring access to each of the arguments that has been passed, lest the target procedure act as an unwitting accomplice.

*The principal reference is BD.9.02, Figure 1.

4.3.2.4 On Outward Calls

A calling procedure could in theory also specify argument pointers to data objects for which the caller does not have ring access. Something must be done to prevent <arg_pull> from unwittingly copying these data objects over into the outer ring segment that would be accessible to the less privileged target procedure acting as a deliberate accomplice. An effort must be made to validate all argument pointers.* In this case, the validation must be done before calling <arg_pull>.

4.3.3 Stack Management in the Multi-ring Environment

In this section, we consider what must be involved when creating the stack frame as a result of a call to an arbitrary segment in our multi-ring environment. Let us imagine a call to <gamma> has been executed. Further assume that <gamma> is to execute in ring k. Ordinarily, as we recall from Chapter 3, <gamma>'s first duty is to execute a save sequence so as to add a new frame to its stack segment, which in this case would be <stack_k>. In addition to "creating" the frame we are reminded there is also the matter of storing in this frame the argument list pointer passed to <gamma> by the calling procedure (call it <beta>). Also there is the linking of the frame to its predecessor frame, and the resetting of the stack pointer, sp. All is well and relatively simple when <beta> itself belongs to ring k.

Are any new clerical problems introduced in creating the stack frame for <gamma> when the calling procedure <beta> is in ring $j \neq k$? Plenty! Fortunately they are all handled for us by the Gatekeeper. We now look at some of these problems and how they are solved by the Gatekeeper.

---

*The validating technique in the case of either inward or outward calls is essentially the same. For each argument pointer in the argument list the following steps are taken:

1. Determine the ring brackets for the segments defined by the argument pointer. Ring brackets are kept in a ring 0 data base called the Known Segment Table. From the ring brackets, determine s, the highest ring number in the access bracket.
2. Compare s with the ring number, t, of the faulting procedure. If $t \leq s$, the argument pointer is valid and is invalid otherwise.

The ring number t is remembered for use in the above test as a special parameter known as the validation level. Further explanation of validation levels is given in Section 4.3.4.

4.3.3.1 The Housekeeping Problem in Getting Ready to Produce the Frame for <Gamma>

The stack frame which <gamma> is to create must be placed in <stack_k>. The problem is – how is the segment number for <stack_k> determined and what is the proper offset for the <gamma> frame? Getting stack_k# is complicated by the fact that <stack_k> may not yet be known (i.e., no entry in the KST). After all, stack segments are no different from any others. They are acquired and/or created only as needed. If no procedure in ring k has ever been called, <stack_k> will be unknown.

A special pointer scheme is employed by the Gatekeeper in keeping track of segment numbers for stack segments and of offsets into them for "next" frames. The details, which be of only peripheral interest to a subsystem designer, are given in the remainder of this paragraph and in Figure 4-11. A one-per-process, ring-0 data base called the process definitions segment, <pdf>, contains a block for 64 its pair pointers to the stack segments. The Gatekeeper will find stack_k# at location

<pdf> | [stacks] + 2*k,

unless of course <stack_k> is unknown, which is indicated in the block by a null pointer. The Gatekeeper then creates the desired stack segment and initiates the pointer. The address for the last used frame in <stack_k> is then seen to be

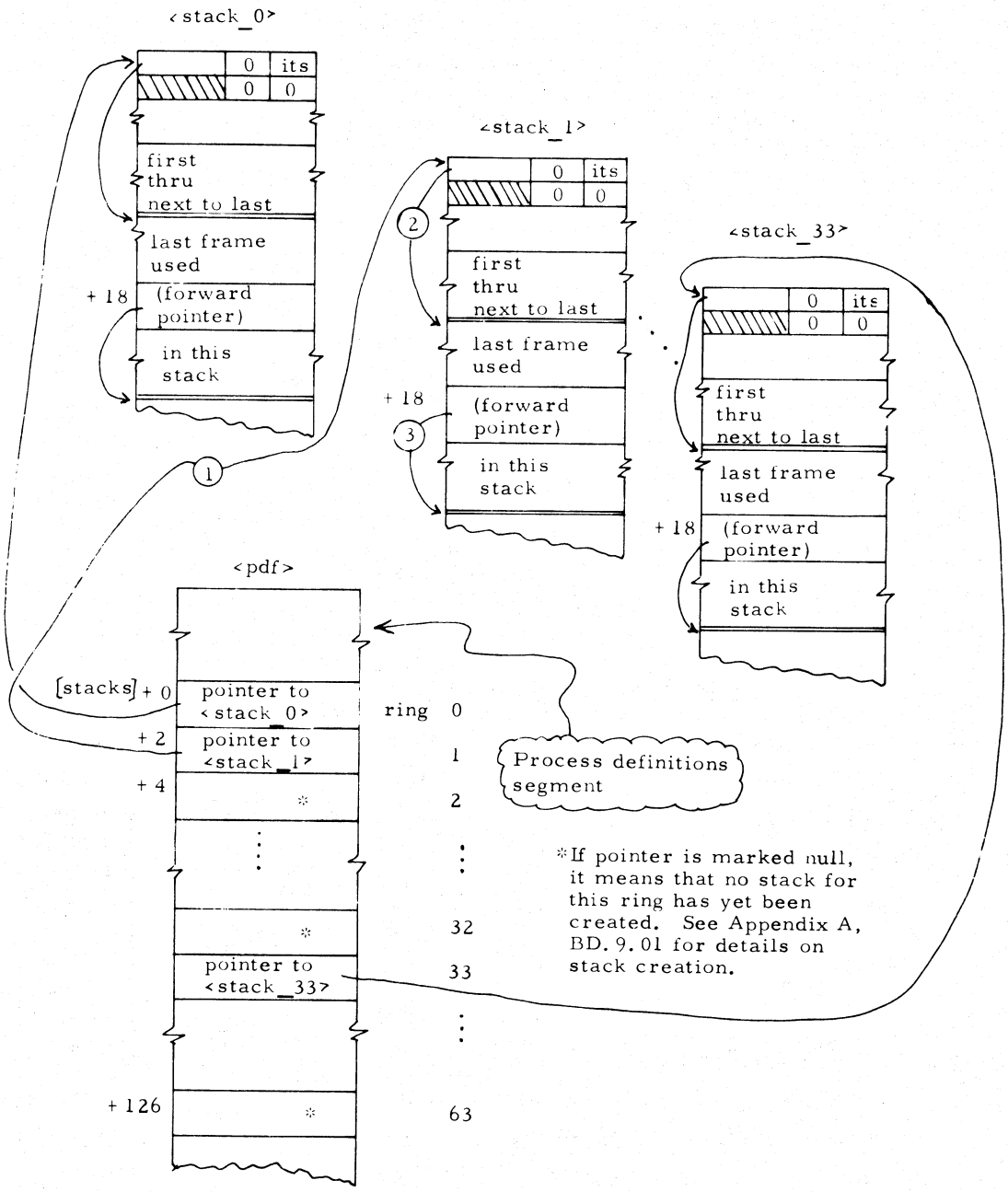
<pdf>| [stacks] + 2*k, *

The forward pointer at +18 in this frame then gives the desired location for the next frame (e.g., lines (1), (2), and (3) in Figure 4-11.

The format of a new-born stack segment is shown in Figure 4-12. It is endowed with an 8-word header followed by an essentially empty 32-word frame. This frame's back pointer (at<stack_k>| 8 + 16) is null to denote the bottom of the stack. Its forward pointer (at <stack_k>| 8 + 18) is set initially to stack_k# | 8 + 32 for starting the forward thread.

Upon creation, the first word pair in the stack is set to point to the empty frame which in this case acts as a pseudo last-used frame. The Gatekeeper updates the first pair as one of its housekeeping duties each time it supervises departure from ring k to some other ring.

The third and fourth words in <stack_k> hold the invocation number and the validation level about which we will have more to say shortly.



The Gatekeeper creates stack segments as needed and places pointers to the head of each one beginning at <pdf>|[stacks].

Figure 4-11. <pdf> is a One-per-process Ring 0 Data Base

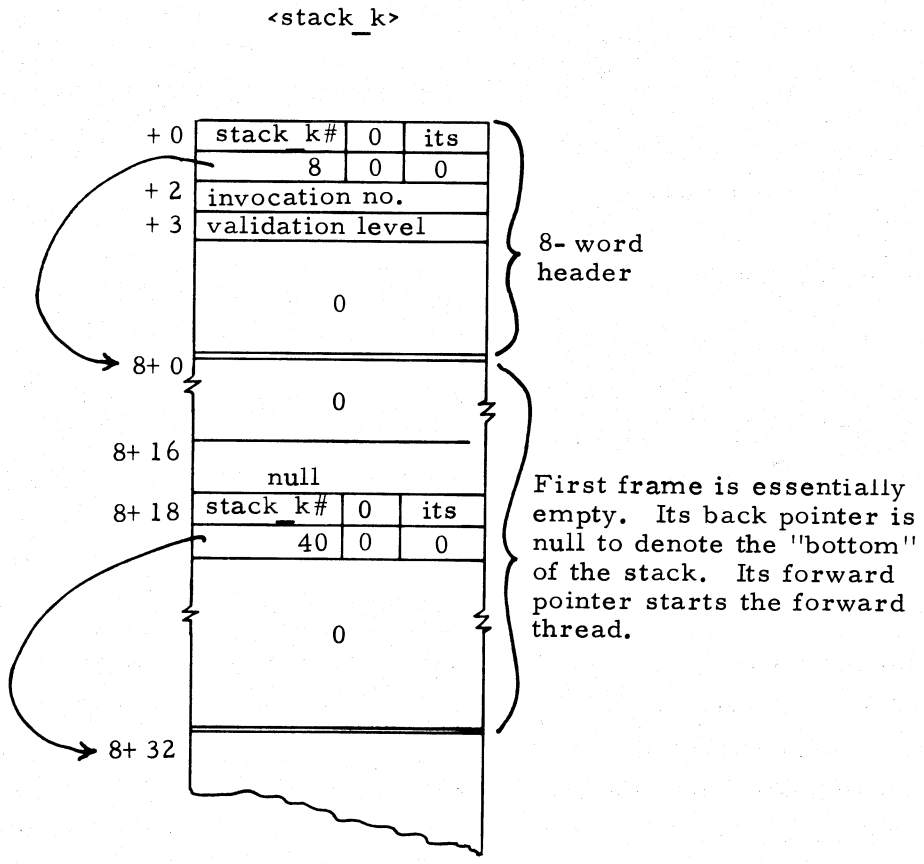


Figure 4-12. Format of a Newly Created Stack Segment

4.3.3.2 The Stack Switching Problem

The Gatekeeper has now located the place in the new stack where the about-to-be-called procedure, <gamma>, is to create its stack frame. But, more bookkeeping problems remain. The normal return sequence in <gamma>,

```
ldb     sp|16,*     reload 8 base registers
lreg    sp|8        reload 8 index registers, etc.
rtcd    sp|20       return
```

should function properly, independent of cross-ring considerations.*

The first instruction

```
ldb     sp|16,*
```

is supposed to reload the base registers (all but sb) from the stack frame of <gamma>'s caller. Assuming we are using the standard save sequence dictates that the predecessor frame must be found in the same stack segment. (This predecessor is pointed to from

```
sp|16
```

which is the back pointer of the current <gamma> frame.) But, if <gamma> has been called by <beta> from another ring j, the stack frame in question actually resides in <stack_j>, an entirely different segment. To resolve this apparent conflict, the Multics solution is to place a special copy of <beta>'s header in <stack_k> immediately ahead of the frame for <gamma>. The copy of the <beta> frame header is ordinarily referred to as the "dummy" frame.

The Gatekeeper has the responsibility for producing this dummy frame, which serves a number of useful purposes. Figures 4-13 and 4-14 picture this activity. In Figure 4-13 we show the copying of the <beta> frame header from <stack_j> to <stack_k>. We also indicate that the Gatekeeper resets the its pair at <stack_j>|0. (Dashed line (1) to point to the <beta> frame replaced by line (2).) The new value in <stack_j>|0 will be needed by the Gatekeeper whenever, at some future point in time, an inter-ring procedure call is made into ring j.

In Figures 4-14 and 4-15 we show what the Gatekeeper must do to the dummy frame before it is "usable".

Shaded portions of the dummy frame indicate the neccessary modifications:

(a) At newsp + 28 store an its pair pointing to the original stack frame for <beta>, located at stack_j#|sp_β. This pointer is called the cross-ring pointer.

*<Gamma>'s compiler will not know <gamma>'s ring number as this could be different for each process sharing <gamma>. Moreover, the compiler will not know the ring of <gamma>'s caller.

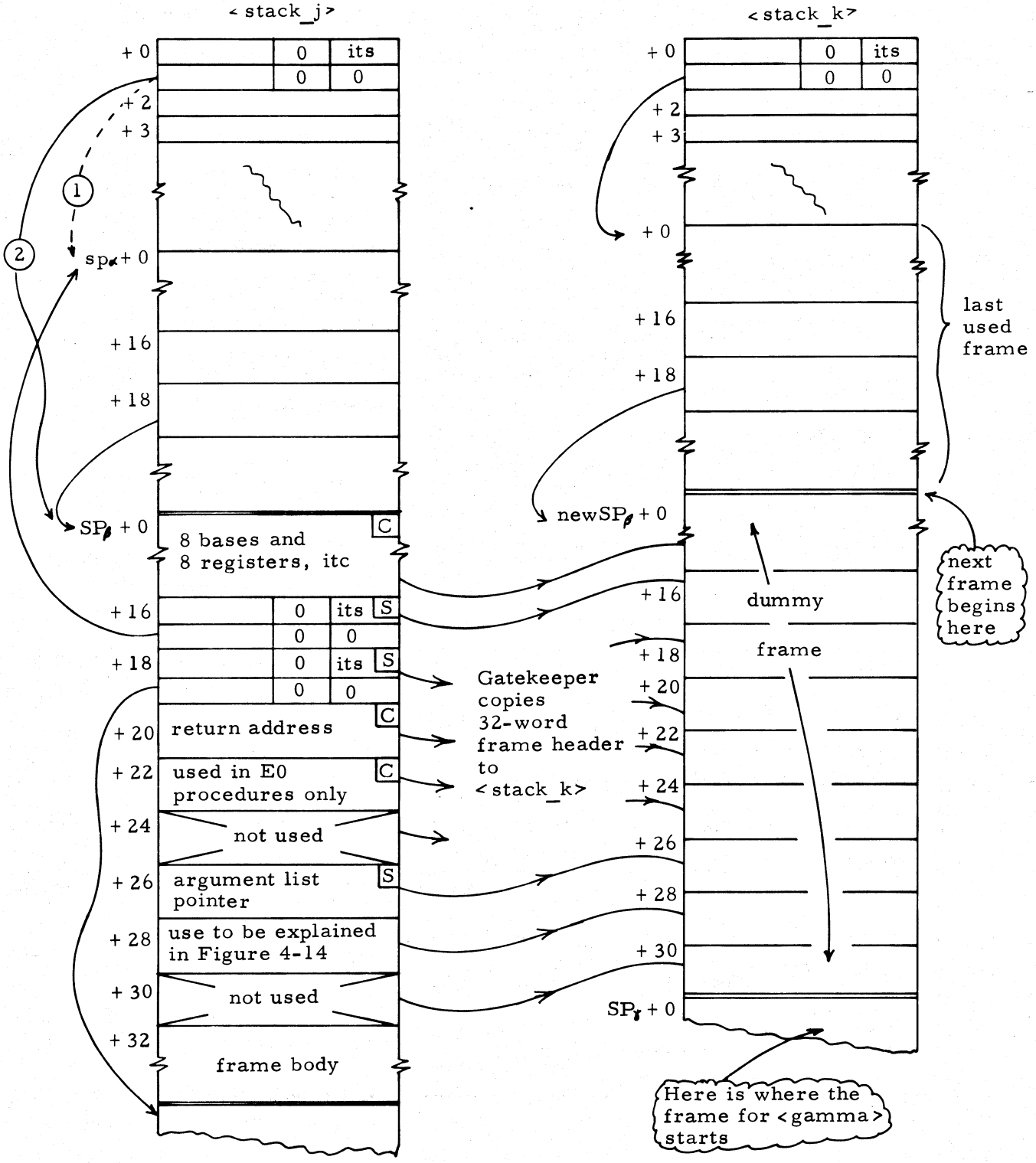
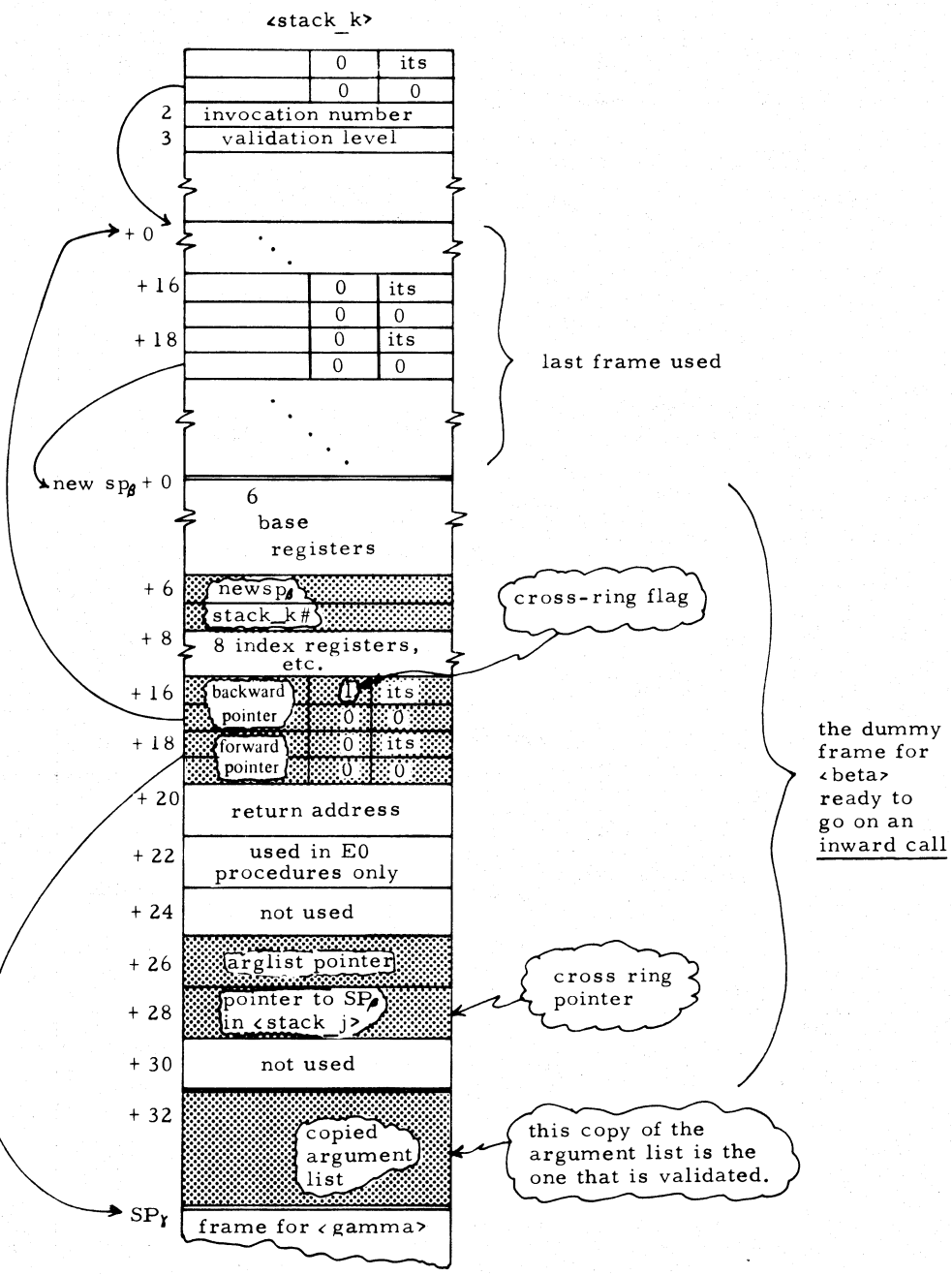
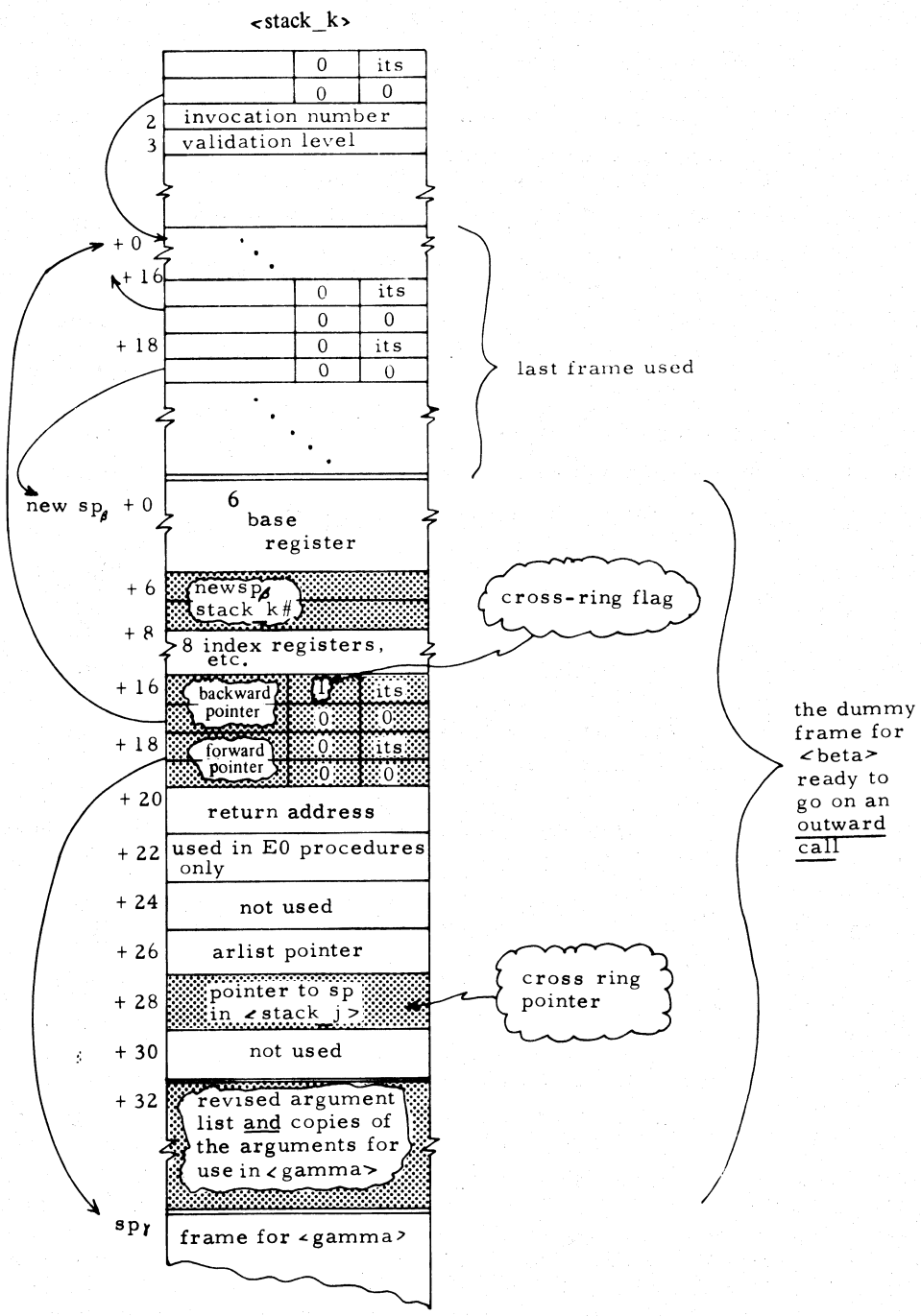


Figure 4-13. Making the Dummy Frame for <beta> in the Stack for the Ring of the Called Procedure



Note the last portion where the copied argument list has been safe stored for purposes of validation.

Figure 4-14. Dummy Frame for <beta> in <stack_k> after being Modified for an Inward Call



(Same as Figure 4-14 except for copies of arguments in the call on <gamma>.

Figure 4-15. Dummy Frame for <beta> in <stack_k> after being Modified for an Outward Call

(b) At newsp$_\beta$ + 16 place a cross-ring flag (i.e., a 1 in the op code position) to mark this frame as a dummy.

The cross ring flag and the cross ring pointer are vital to the success of the condition handling and unwinding mechanisms to be described in Chapter 5.

Other details of a housekeeping nature are:

(c) The dummy frame is chained to the preceding frame in <stack_k> by adjusting the backward pointer.

(d) The forward pointer is reset so it points to the beginning of the next frame which is to be used by <gamma>.

(e) The stack bases saved in newsp$_\beta$ + 6 and newsp$_\beta$ + 7, which refer to the old stack frame in <stack_j>, are reset to point at newsp$_\beta$ in <stack_k>. This is done in order that the instruction pair

```
ldb    sp|16,*
lreg   sp|8
```

be executable in some meaningful and consistent sense, especially for outward returns. We must bear in mind that an ldb instruction cannot reset the locked sb base register.*

On inward returns the effect of the two restore instructions is overriden by similar instructions performed by the Gatekeeper.

Multics cannot (and the Gatekeeper does not) trust an inward-returning procedure to properly carry out the restoration of bases and registers for its inner-ring caller. When the

```
rtcd   sp|20
```

is executed, it faults, of course, to the Gatekeeper — which takes no chances. The Gatekeeper repeats the restoration of the bases and registers, this time using the stack pointer for the original <beta> frame in <stack_j> where the integrity of the saved information cannot be questioned. The pointer to the frame in <stack_j> is itself safe stored in a special segment (the so-called "return stack", <rtn_stk>) about which we will say more later.

(f) On inward calls there is a special (and subtle) type of protection violation that must be prevented. It concerns the possibility of deliberate or accidental changes to argument pointers after they have been validated by the Gatekeeper mechanism, but before they have been used by the target procedure. For instance, such violations can arise if and when two cooperating processes have agreed to read-write share the outer-ring segment stack that holds the calling procedure's argument list. To prevent such postvalidation tampering, the Gatekeeper first duplicates the calling argument list in the dummy frame at sp$_\beta$|32 (in the inner ring stack segment)

*See Chapter 1, Section 1.4, for a review of the ldb instruction.

and then validates from the copied argument list. $Sp_{\beta}|18$ will, of course, have been appropriately set to point to sp_{γ}, taking into account the length of the copied argument list. The base pair ab←ap will be adjusted in the saved copy of the machine conditions for the faulting procedure, so that when the call is completed the arglist pointer in the new frame for <gamma> placed at $sp_{\gamma}|26$ will point back to $sp_{\beta}|32$. The dummy frame is now ready for use in inward calls (Figure 4-14).

(g) On outward calls the Gatekeeper, after validating the original argument list found in the caller's stack frame, then calls <arg_pull>, as mentioned in the preceding section, to prepare a revised argument list that contains pointers to copies of the arguments. Sp_{β} will, of course, have been appropriately set to point at sp_{γ} taking into account the extension of the dummy frame for <beta> to include the new arglist and the argument copies. These items are placed in the dummy frame at $sp_{\beta}|32$. The ab ← ap base pair will be adjusted in the saved copy of the machine conditions for the faulting procedure, so that when the call is completed the arglist pointer in the new frame for <gamma> (placed at $sp_{\beta}|26$) will point back to $sp_{\beta}|32$. The dummy frame is now ready for use in outward calls (Figure 4-15).*

In reviewing all these dummy-frame details, notice that the dummy is tied to other frames in two ways:

(a) to the preceding and succeeding frames in its stack (sp|16 and sp|18). The back pointer is primarily for use by system-supplied debugging tools (and for use by the Unwinder mechanism discussed in Chapter 5), and

(b) to the original copy of the frame in <stack_j> via the cross-ring pointer at sp|28.

4.3.3.3 Saving Vital Cross-ring Data on the Return Stack (<rtn_stk>)

The Gatekeeper keeps a protected record of each inter-ring call in a special, one-per-process data segment in ring 0. It is called <rtn_stk>.

The following four items are stored in <rtn_stk> as a consequence of each inter-ring call:

(a) The ring number of the faulting procedure. This value is taken from an embellished copy of the faulting procedure's machine conditions.

(b) The validation level of the faulting procedure (explained in Section 4.3.4).

(c) Pointer to the faulting procedure's stack frame i.e., a protected duplicate of the cross-ring pointer.

*Additional details are shown in Figure 4-20.

(d) Pointer to the normal return location in the faulting procedure, i.e., a protected copy of the value given in the faulting procedure's stack frame at sp|20.

Figure 4-16 shows the overall and detailed structure of <rtn_stk>. A relative pointer to the top entry of <rtn_stk>, called the invocation number is updated after each new entry is stacked or removed (popped).

Each entry consists of six words. The first word of each entry holds a back pointer to the preceding entry. The next five words hold the four saved items.

After completing this entry in <rtn_stk>, a copy of the new invocation number is stored in the target ring's stack segment at <stack_t>|2 and a copy of the saved validation level is stored in <stack_t>|3.

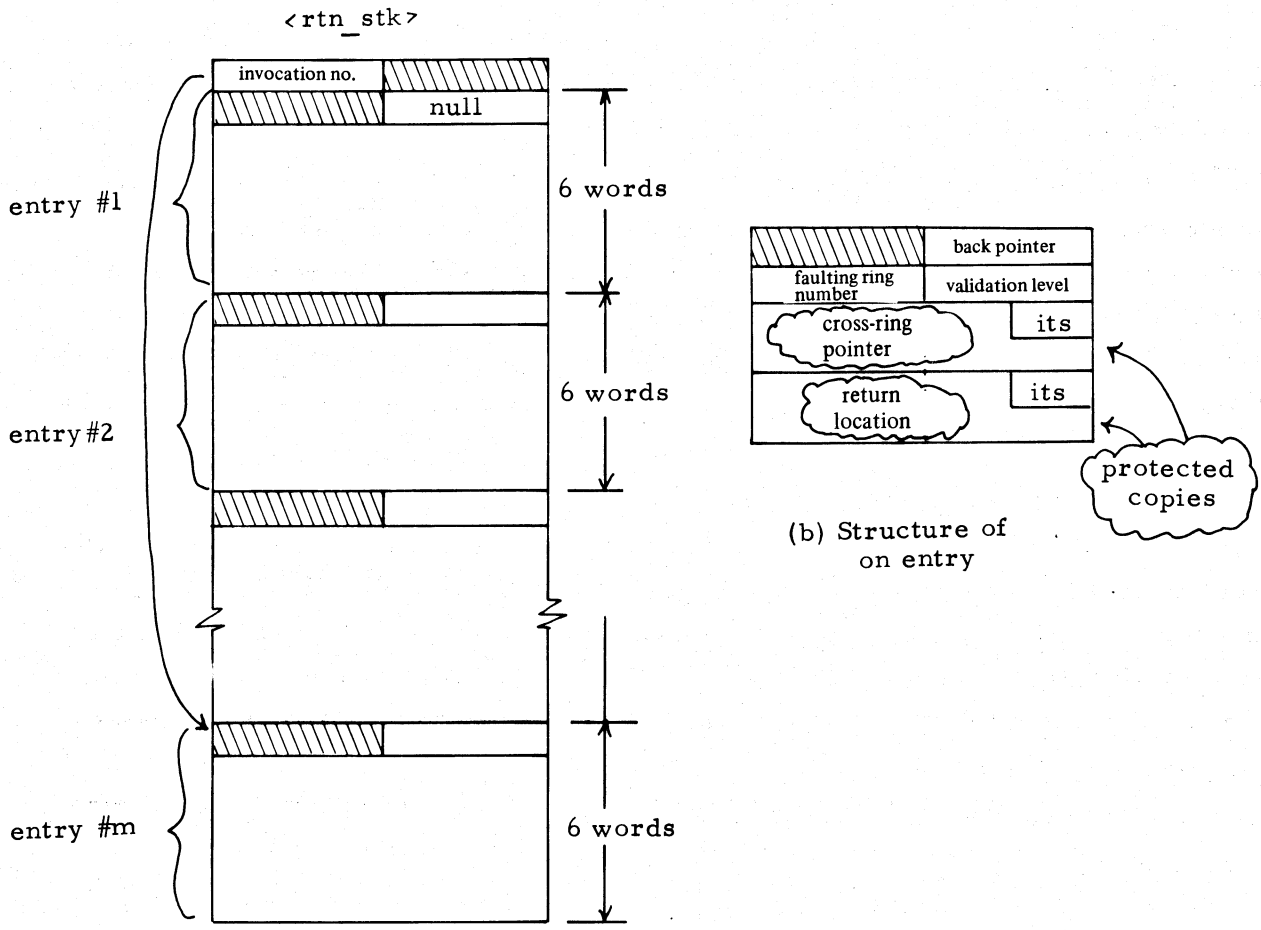
By reading the invocation number in a given stack segment, <stack_s>, ring-0 system routines (not user routines) are able to locate the corresponding entry in <rtn_stk>. This entry provides a "trail" back to the procedure (its ring number, and its stack frame) which caused the crossing into ring s. The condition handling routines and the unwinding mechanism (for abnormal returns) depend on the invocation number for tracing portions of the past history of a process.

The invocation number is also potentially useful to ordinary users as a means of recording when things happen (i.e., with respect to ring crossing history). Thus, a user, executing in ring t could associate with a certain stored block of data a copy of the current invocation number (call it curinv) found at <stack_t>|2. At any later time, the block of data can be identified by the associated value of curinv as to when it was stored.

4.3.4 Validation Levels and How They are Used

The validation level is the ring number for the segment on whose behalf the call on the ring_t segment is being made. It often arises that a user procedure will make an inward call to a "supervisory" module which, in turn, will call another procedure (either in the same ring or in an even lower ring) to perform some vital function on behalf of the user. The target supervisory procedure at the end of this "chain" may need to know the ring number (importance level) of the original caller in order to perform its task properly. In this way, the DC* routine in ring 0 will know the importance of the party it is serving and will not be outwitted into "thinking" it is serving a ring -0 procedure — its immediate caller — when in reality it may be serving a user in ring 32 or greater. It is seen, therefore, that proper use of validation levels is a means of increasing protection where needed.

*Directory Control



(a) overall structure

(b) Structure of on entry

Figure 4-16. Overall and Detail Format of <rtn_stk>

When crossing to a target ring, t, the validation level is always stored in < stack_t> | 3. The value assigned to this location is a copy of the one in the stack of the faulting procedure. (The Gatekeeper does this copying.) In this way, if there has been a chain of two or more inward calls in reaching ring t, the validation level that is set in < stack_t> | 3 will normally reflect the ring number of the procedure that initiates this chain of ring crossings. The called procedure in ring t is then free to interrogate < stack_t> | 3 as required.

It should be kept in mind, that once execution passes to a target ring, t, any procedure executing in that ring is privileged to read or write the contents of < stack_t> | 3. The Gatekeeper will not however indiscriminately copy validation level values. If the current value in < stack_t> | 3 is lower than the ring number, say r, for a faulting procedure, the value passed to the target ring will be r. In this way, it will not be possible to trick the inner ring target into believing its caller has a validation level that is less than its ring number.

The algorithm used by the Gatekeeper for setting these values is displayed in Figure 4-17. On calls, outward or inward, the validation level associated with the faulting ring is saved in < rtn_stk>, and a copy of this saved value, possibly altered in a way described below, replaces the current value of the validation level in the stack of the target ring. On inward calls if v_r, the validation level for the calling ring r, is for any reason lower than r, then r, rather than v_r, is the value passed to < stack_s>. On outward calls if v_r is less than s, the target ring number, the value s is passed to < stack_s>. On returns, inward or outward, the most recently saved value simply replaces the current value in the target ring's stack segment.

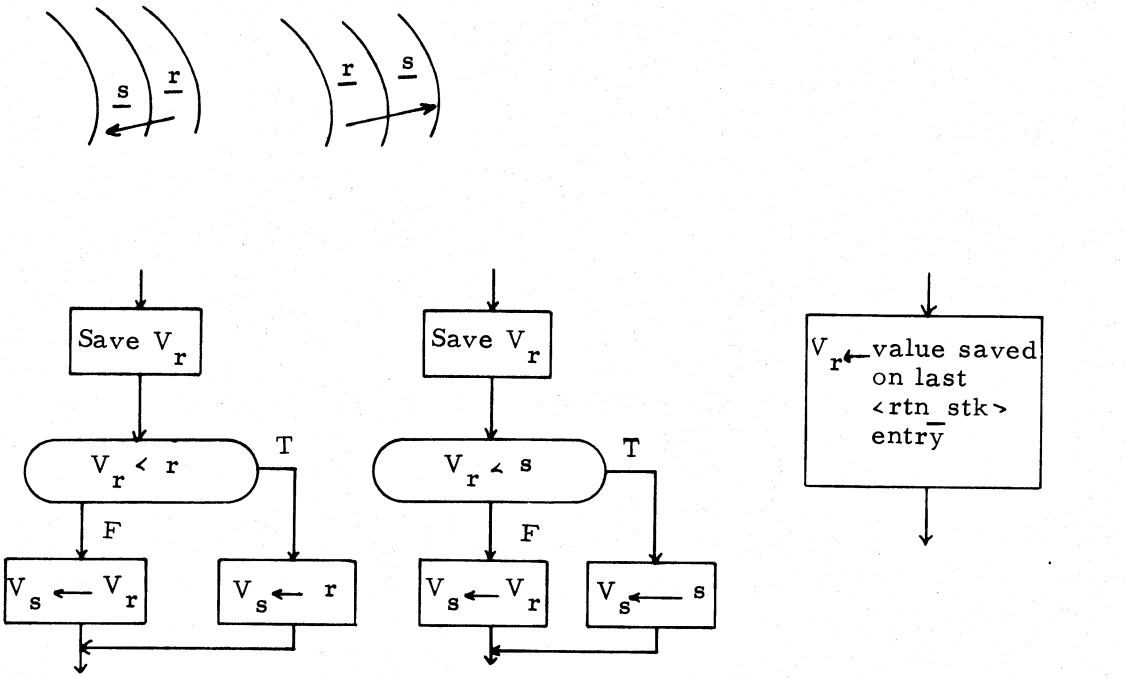
In the remainder of this section we give an example aimed at motivating subsystem applications of validation levels.

Example

Here, we illustrate a case where it is expedient to check the ring of the caller to determine the nature of the service to be rendered.

We imagine a school records subsystem which operates in four rings 35, 34, 33, and 32, as shown in Figure 4-18.

All personnel and student records are kept in data segments of ring 32. The ring-32 procedure < get_rec>, called from outer rings, retrieves desired information from any of these data files according to the arguments it is furnished and according to the validation level at < stack_32> | 3.



(a) inward call

(b) outward call

(c) restoring a validation level on a normal return to ring r

Saving and passing a validation level, v_i from ring r to ring s

Key: v_i, i = 0(1)63 means the variable whose value is the validation level for ring i. v_i is located at <stack_i>| 3. Saving means storing in the <rtn_stk> entry.

Figure 4-17. Gatekeeper's Algorithm for Saving and Passing, and for Restoring Validation Levels during Ring Crossings

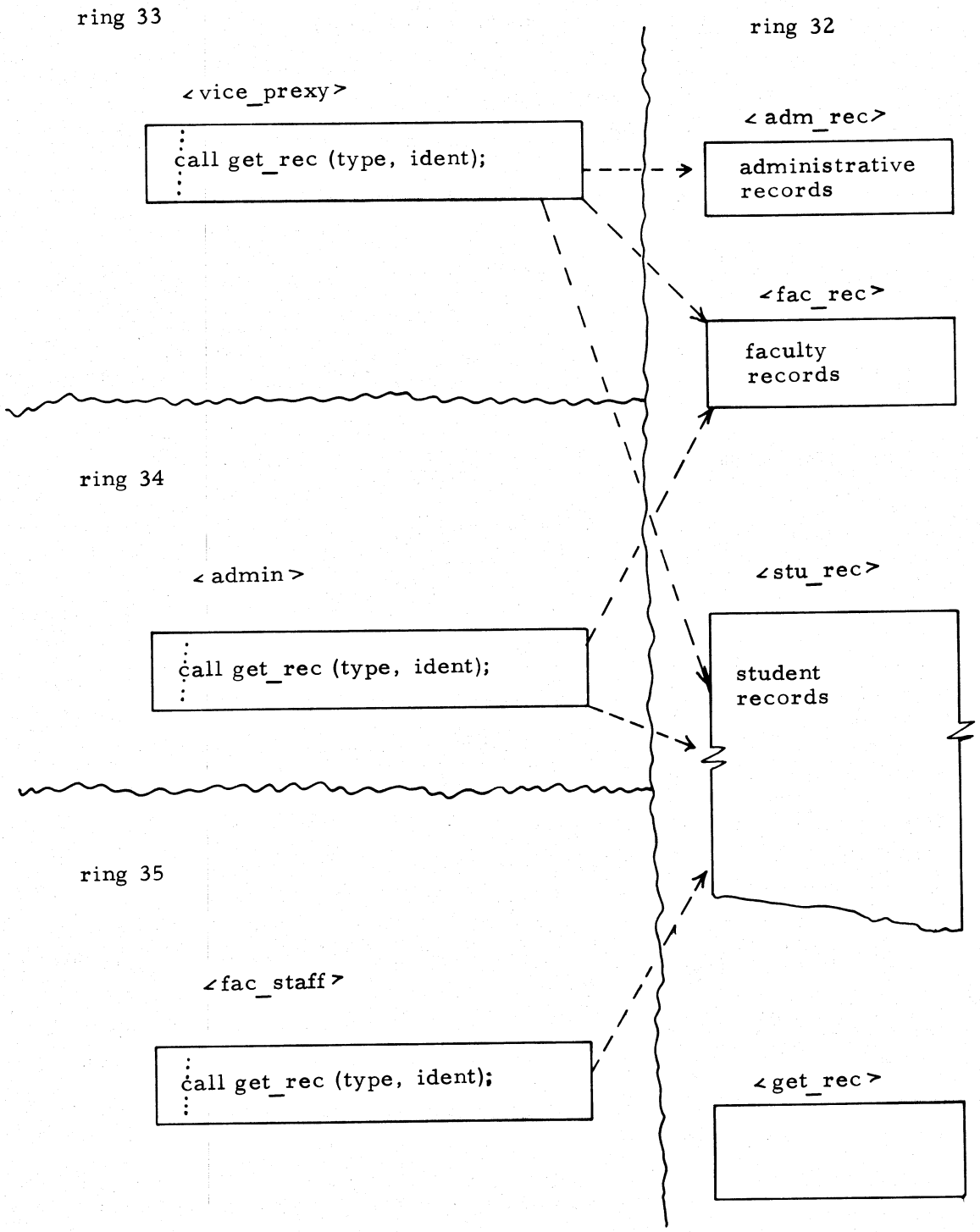


Figure 4-18. School Records Retrieval Subsystem

Thus, the validation level found at sb|3 describes the "authority" of the caller. A call from ring 33 or lower is adequate for any request for data, be it from <adm_rec>, <fac_rec>, or <stu_rec>. However, a request from ring 34 for <adm_rec> data or from ring 35 for <fac_rec> or <adm_rec> data must be rejected by <get_rec>.

4.3.5 Outward-call Argument Lists

It is possible to design many subsystems which never employ outward calls. Certainly, this would be the case for a subsystem whose segments reside entirely in ring 32. If the subsystem you are designing is in this category, you can skip the remainder of this Chapter in good conscience. If not, two cases are of interest:

(a) The subsystem itself is to be coded using outward calls, but the subsystem user is to be so controlled, e.g., by limiting his procedures and data to the outermost ring so that he cannot execute outward calls.

(b) The subsystem procedures as well as the user procedures, with the latter no longer restricted to the outermost ring, can execute outward calls.

Some important implications follow in each case:

Case (a) The subsystem must be coded in a source language whose compiler can recognize outward calls. This recognition is necessary so that the compiler can generate argument lists which are properly embellished with pointers to descriptions of the corresponding arguments.

The Multics PL/I and EPL compilers, for example, provide this recognition capability. It is achieved through use of the so-called "callback" option.* This amounts to a declaration which can be made in any external procedure <a>. In the callback option the programmer lists all procedures, e.g., <x1>, <x2>, etc., a call to which is to be regarded as an outward call. The form of the option is:

```
callback (x1, x2, etc.)
```

It should be easy to see why, in one way or another, the language processor must be supplied this type of information. Were this not so, the processor would have no way of knowing how procedures, whose names appear in the program, relate to one another vis-a-vis rings. If you write your subsystem in any other language or languages, be sure the processors are equipped with this facility. Assuming you don't have to be involved in building this type of software facility, there is no more you need to know. However, if it is your problem to do this job, then read "case (b)".

*Primary reference is BP.0.02. The options attribute is part of the (first) procedure statement of every PL/I external procedure.

Case (b) The language the user codes in need not be the same as the language(s) used for coding the subsystem. If this is the case, and if the user-coded procedures may be written with outward calls, then you must make sure that the processor which handles user codes also has the same capability for recognizing outward calls and for generating suitable argument lists.

Figure 4-19 shows the format of the argument list which must be supplied in each n-argument outward call. There are 2 x n additional words to be supplied, consisting on n its pair pointers to the argument descriptions. By consulting the data descriptions for the arguments, <arg_pull>, when called by the Gatekeeper, is able to decide how to copy each argument into the target procedure's stack.

We now illustrate what <arg_pull> does in specific instances, imagining in Figure 4-20 that the procedure <beta> makes an outward call to <proc_hi_ring> with two arguments; x, an integer, and name, a non-varying character string. The copied block of information is placed at the tail end of the dummy frame made for <beta> in the stack segment for the ring (t) of <proc_hi_ring>. (See also Figure 4-15.)

The copying work of <arg_pull> though conceptually simple, has its share of clerical complications.* Here we give a simplified two-step description of the aspects which subsystem writers should know.

4.3.5.1 Copying the Argument List

Note, in our example of Figure 4-20, that each argument pointer in the copied list is now modified to point down in <stack_t> to its respective argument datum (or to its specifier,† if it has one).

---

*Full details can be found in BD.9.02. The task is sufficiently complicated that in the initial implementation varying string arguments or arrays of same (because they involve the use of Free Storage) may not be passed on outward calls.

†Specifiers were first described in Table 3-1. (Types of arguments and their structures.) If a refresher is needed, reference this table.

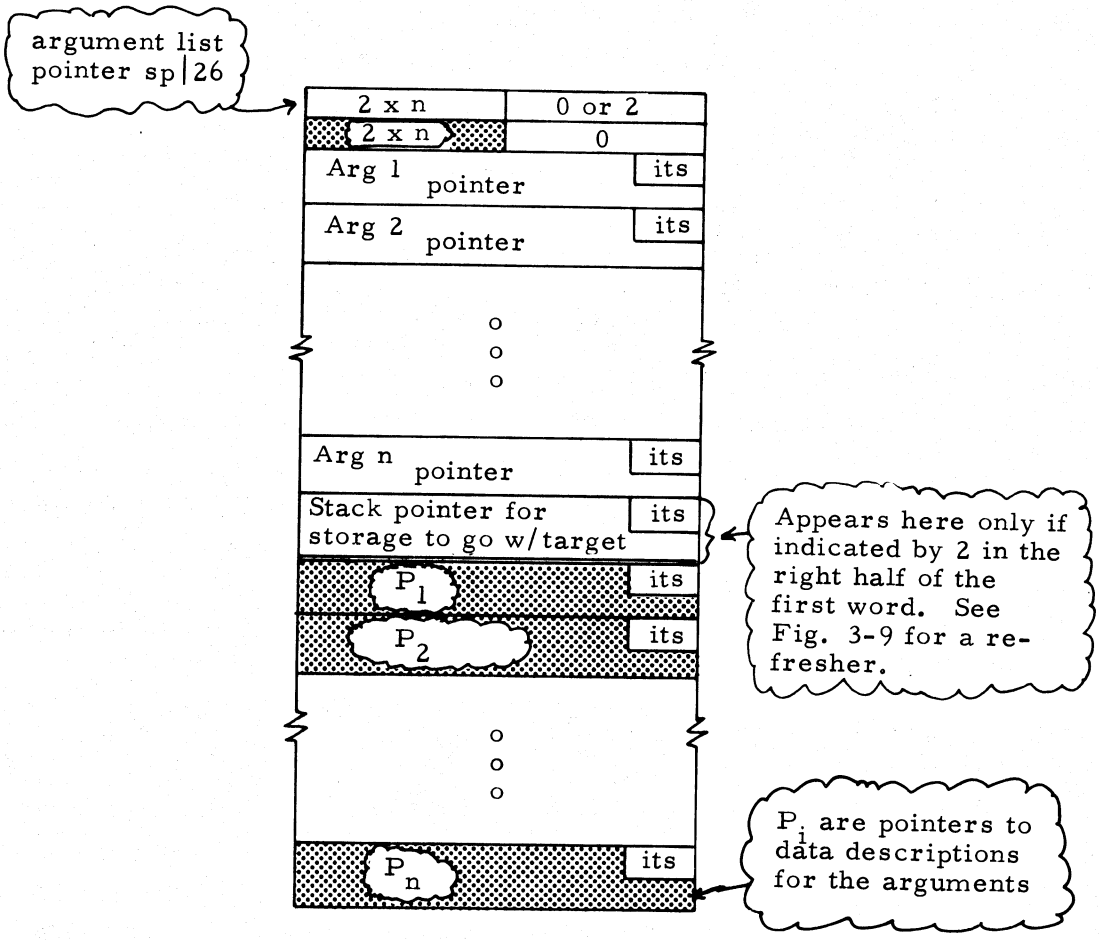
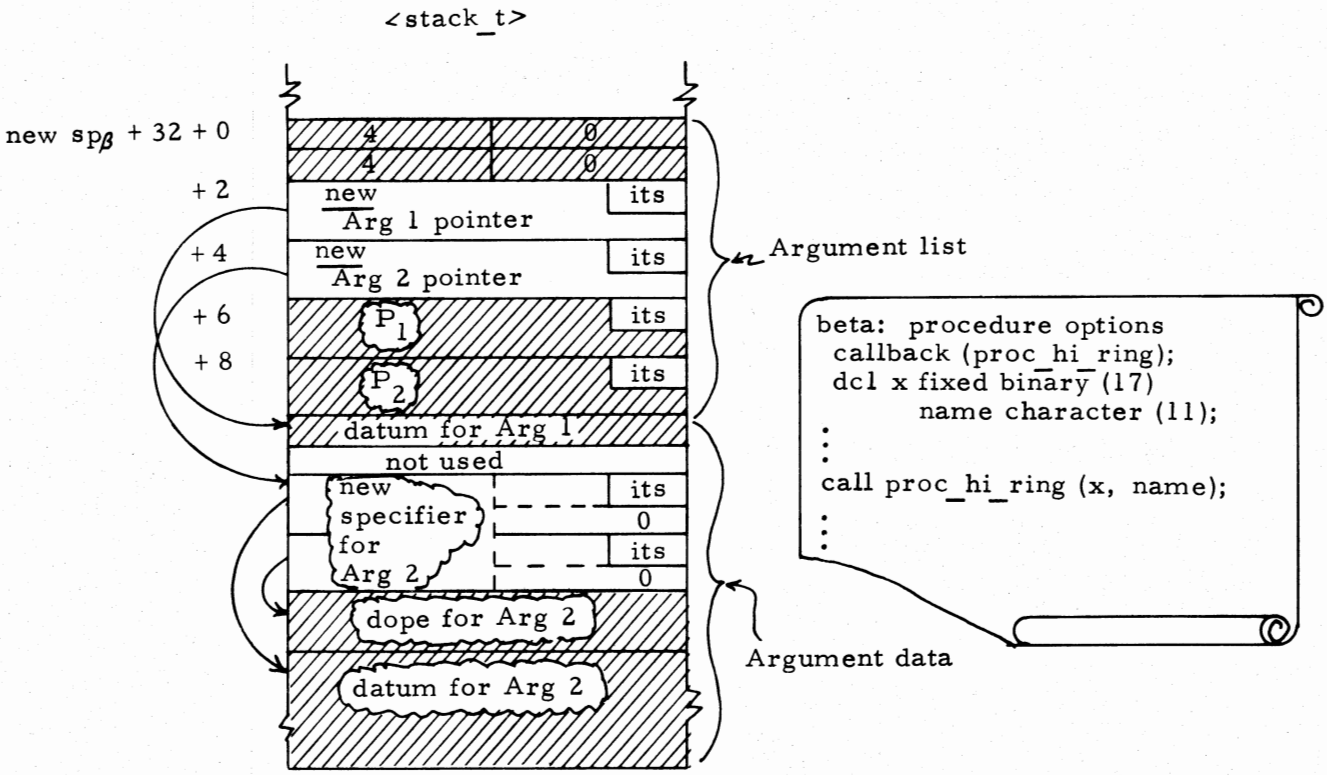


Figure 4-19. Format for an Argument List for Use in an Outward Call



This information is deposited at the end of the dummy frame created for the faulting procedure. Crosshatched areas represent exact copies placed there by <arg_pull>.

Figure 4-20. Appearance of the copied Argument List and copied Arguments placed by <arg_pull>

The pointers to the data descriptions are copied exactly as they were in the calling argument list.*

4.3.5.2 Copying the Arguments

Each data description provides <arg_pull> with the information it needs to copy the associated argument. Figure 4-21 shows the Multics standard format for an argument description.

The data type code in the description determines if the argument has a specifier and dope. In copying arguments which have specifiers, the dope and datum parts are copied exactly. The specifiers, of course, are new. They must be constructed "on the spot", to point to the dope and datum copies. This is suggested in Figure 4-20 where only the cross hatched sections represent exact copies; the rest are newly created for the purpose. Arguments which can be copied by <arg_pull> are restricted, at least in the initial Multics, to scalars and one-dimensional arrays of scalars, i.e., to data types given in Table 3-1.

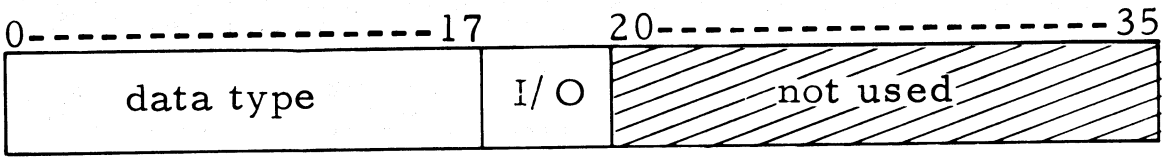
4.3.5.3 Recopying of Return Arguments on the Inward Return

On an outward call to procedure <p>, there may be (copied) arguments whose values are altered during execution of <p>. During the inward return the Gatekeeper must see to it that the possibly new values for these return or "output" arguments replace the original values pointed to in the original copy of the argument list. Output arguments can be recognized by examination of the I/O code (bits 18 and 19) of the argument description, as indicated in Figure 4-21. To deal with these arguments, if any, the Gatekeeper calls another ring 0 procedure, <arg_push>. This procedure searches down the data descriptions found in the original argument list† for arguments

*There (in the target ring), copied pointers are likely to be of no direct use to the target procedure because they will, in general, point to a data segment in an inner ring, i.e., to <x. symbol> where <x> is the calling procedure. However, if the target procedure subsequently wishes to pass any of these arguments to another procedure with an even higher ring number, <arg_pull> must again be invoked by the Gatekeeper. The copied p_i's are now used in constructing the argument list for this second outward call.

†It is not the one in the stack of the now faulting procedure executing the return, but the one pointed to in the stack for the inner ring target procedure. It is not safe to use the argument list in the outer ring stack because this may have been altered by <p> or by any of its "dynamic descendents" that had access to this stack.

Fig. 4-21



Key: Data Type is an integer code for the type of arument The different system standard types and their codes were given in Table 3-1.

I/O is a 2-bit code giving the input/output nature of the argument.

I/O Code	System-wide Interpretation
0 0	I/O nature unknown
0 1	input only
1 0	input and output (requires callback)

Figure 4-21. Multics Standard Format for an Argument Description

indicated to be of the output type (code 10). For each of these the datum (but not specifier or dope if any) are copied from their positions in the outer ring stack to their original positions wherever they may be.

A review of the foregoing on the copying and recopying of arguments has revealed one reason why a subsystem should provide argument descriptions in standard form, namely: It is a necessity if the subsystem is to interface with the Gatekeeper for processing outward calls and inward returns. Other Multics service modules including certain useful debugging facilities,* will also require access to argument descriptions in the same standard form.

Normally, the compiler or assembler used to generate code in the subsystem will have the responsibility for generating descriptions for all arguments employed in an outward call. Data descriptions for declared variables or for declared parameters (dummy variables) are usually placed in the segment symbol table produced by the compiler or assembler. Hence, a pointer to this data description can always be generated by the compiler when needed in the construction of an outward-call argument list. The fact is that in EPL-generated symbol tables, the data descriptions are compatible with, but are more elaborate in structure than the Multics standard given in Figure 4-21. The important requirement is fulfilled, however, that the first 20 bits of an EPL-generated data description has the standard interpretation. More details on EPL-coded data descriptions can be found in BD.1.

4.3.6 Gates

In the description of the linkage section given in Chapter 2 of this guide, no mention was made of gates because we then had no way of properly motivating them. We correct this omission here. As mentioned in Section 4.2.3, a gate has the form of a special entry in the linkage section. By way of review, an ordinary entry is found in the linkage section of a target procedure. It consists of a quadruplet whose form is:

```
eaplp    -*,ic
aos      2,ic
tra      linq-*,ic*
arg      0
```

where linq is the offset to the link (pointer) to the program point in the target.

*See BX.10, Interactive Debugging Aids, for more details.

A gate is a quintuple, the last four words of which are identical with the above form. The first word is a no-op instruction whose address field is the location, within the same linkage block, which contains additional information describing the gate. The form of the no-op instruction is:

```
nop     gate_info, du
```

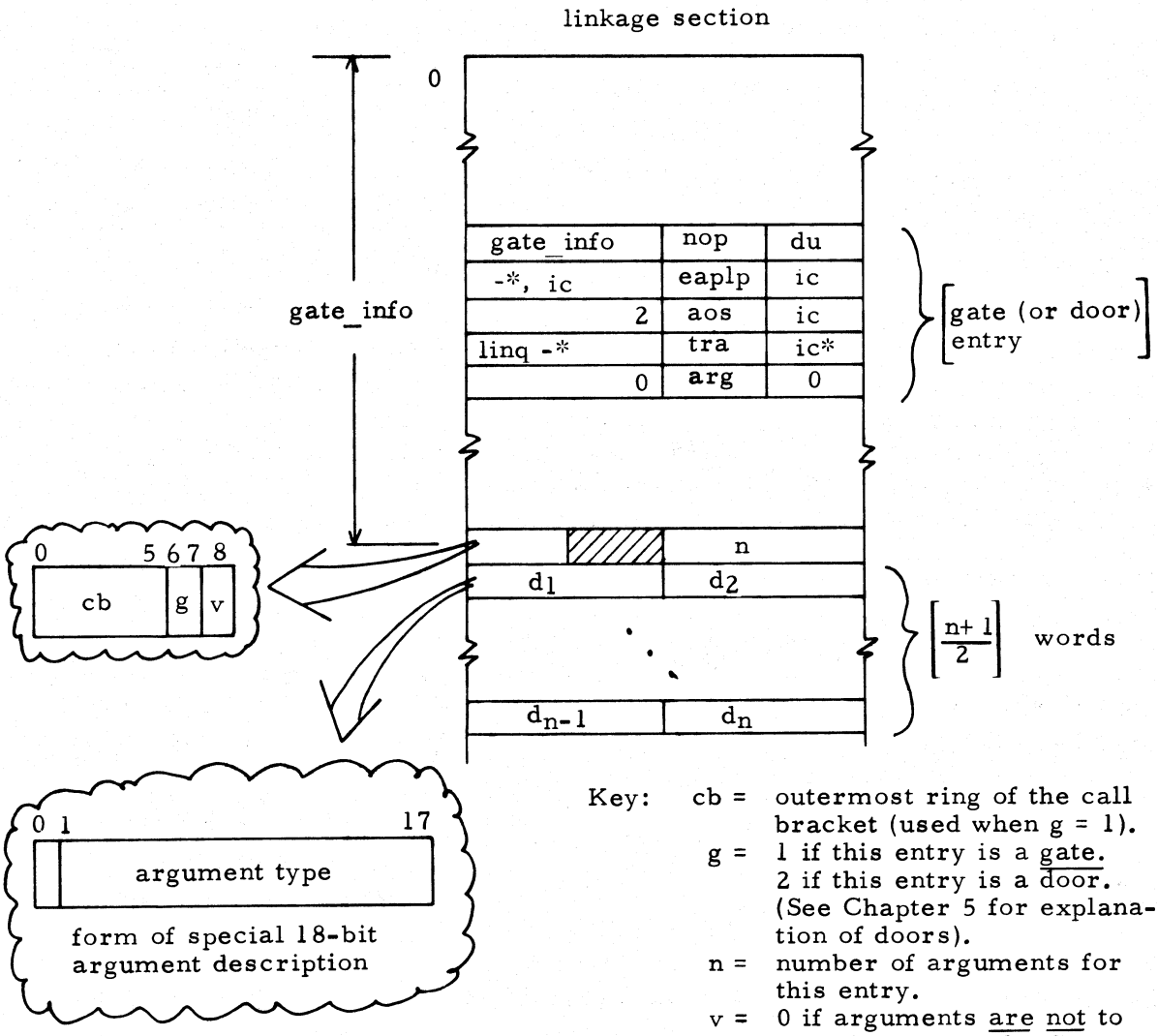
When the Gatekeeper is processing an inward call, the address of the faulting instruction should point to this no-op. Figure 4-22 shows the kind of additional information (at gate_info) which the Gatekeeper would then have available for handling this attempted inward call. Inspection of the gate information will lead the reader to some interesting inferences:

Normally, one would want the Gatekeeper to validate all arguments being passed on an inward call. Whoever writes a translator which generates gates in a linkage section can provide users of this translator with an option to ignore this Gatekeeper's service on inward calls. A user who takes the option to ignore argument validation will eliminate overhead, but will risk damage in the target's domain of access.

Whether or not the n arguments are to be validated, the next $\left\lceil \frac{n+1}{2} \right\rceil$ words provide special 18-bit descriptions. A one in the leading bit position of a description indicates a return argument. Such an argument may be used in situ by the target procedure, because it is (presumed to be) located in the domain of access of the caller. Arguments so coded will not be copied, thus avoiding the copying overhead. If the leading bit of the 18-bit description is zero, the interpretation is that the argument is to be called by value (i.e., no value is to be returned for this argument). The Gatekeeper will ask <arg_push> to make and place a copy of arguments so described in the target procedure's stack frame.

4.3.6.1 Gate Segments

The 6-bit field in gate-info that is marked "cb" (highest ring number in the call bracket), is always examined by the Gatekeeper when handling inward calls. cb is interpreted as the effective upper bound for the target's call bracket in the event the value of cb is less than the value given in the ring brackets of the segment that contains the gate itself. The cb field may be set arbitrarily ($0 \leq cb \leq 63$) by explicit



Key:
- cb = outermost ring of the call bracket (used when g = 1).
- g = 1 if this entry is a gate.
  2 if this entry is a door.
  (See Chapter 5 for explanation of doors).
- n = number of arguments for this entry.
- v = 0 if arguments are not to be validated by the Gatekeeper.
  1 if arguments are to be validated.
- r = 1 this argument is a return value (ok to use in situ).
  0 no value is allowed to be returned for this argument (call by value).

Figure 4-22. Format of Gate (and Door) Information

coding.* A small value of cb (smaller than the outer ring of the call bracket of the target) serves as additional screening of inward calls.

Most subsystem designers will find few occasions to exploit this extra screening capability for inward calls. When used, it will probably be for the purpose of reducing the linkage segments that are needed in a multi-ring subsystem. The idea would be to concentrate into one linkage segment a collection of gates for different procedures in the same ring. Such a collection, will hereafter be called a "gate segment". †

A gate segment, when properly constructed, would serve as a funnel for access into an arbitrary collection of privileged procedures, e.g., in the hardcore ring. The gate segment itself has fixed ring brackets in the executing process, i.e., (r_1, r_2, r_3) whose values are given in the file branch for this segment. However, any of the cb values found in the gate segment may be less than r_3.

In the Multics supervisor, for example, ring-0 segment named <hcs_> serves as a gate segment to minimize the number of linkage segments needed for ring-0 procedures. The ring brackets for <hcs_> are (0, 0, 32). This segment contains gates to all other procedures in ring-0 which are callable from outside ring 0. The gates for some hardcore procedures have cb = 1, while for others, cb = 32. The Gatekeeper will reject any user call from ring 32 to a user procedure whose gate has cb = 1 (even though the faulting segment's ring is not outside the call bracket for <hcs_> itself and even though the desired entry point is a gate).

4.3.6.2 Doors

The last remark we wish to make is for the benefit of readers who referred to this subsection from Chapter 5. If the data at gate-info is for a door instead of a gate, there will be no arguments involved, because this entry is being used for control of an abnormal return, not for a call. Hence, n = 0 and the gate information consists of only one word. The gate information in this case serves primarily to identify the entry as a door (thus, making gates and doors mutually exclusive).

---

*Eventually source languages like EPLBSA will be expanded so a programmer may declare an entry point to be a gate. Such a declaration would be expected to result in the generation of gate information in the format shown in Figure 4-22. When declaring a gate, one would specify the value of cb or accept a default value which would probably be 63, a value which would produce no screening effect at all.

†This segment would function something like the familiar "transfer vector" used in programs that are loaded in conventional batch operating systems.