Published: 06/27/67
(Supersedes: BD.9.00, 03/31/66)

Identification

Overview of the Multics Protection Mechanism
R. M. Graham, M. A. Padlipsky

Purpose

Although it is often taken for granted, one of the most useful aspects of Multics is the ability the system confers upon the user to share segments with other users. This sharing is basic to the approach taken in implementing the Multics supervisor, whereby each user process contains many pure-procedure supervisor segments within itself (that is, in its "address space"). Data segments may also be shared, and shared supervisor routines frequently reference system-wide data bases. The advantages of segment-sharing are explicitly dealt with in the 1965 Fall Joint Computer Conference papers on Multics and implicitly dealt with in various portions of the Multics System-Programmers' Manual, particularly in the area of the Traffic Controller. As is usually the case with advantages, however, there is a price which must be paid in order to secure the blessings of segment-sharing in Multics: In many cases, supervisor routines perform functions which can only be performed in behalf of other supervisor routines, particularly when dealing with data-bases that contain information which is in some sense "private" - either to a specific user, or to the system. A mechanism must exist which protects the supervisor from being called upon to perform forbidden tasks. In an even more basic sense, consider the problems which would arise if a user routine were inadvertently to destroy information (procedure or data) necessary to the correct execution of the supervisor. A mechanism must exist which protects the supervisor from being damaged by a process which is sharing its component segments. This mechanism, hereinafter called the Multics protection mechanism, is the means the system employs for allowing secure segment-sharing. The existence of a protection mechanism also contributes to more reliable system operation by minimizing the destructive effects of the inevitable, occasional mistake by the system programmers.

There are benefits which stem from the protection mechanism beyond the obvious ones of supervisor integrity. Of course, the protection mechanism is fundamentally intended for the protection of the supervisor; the design chosen, however, lends itself to extension in a fashion which affords protection to non-supervisor segments as well. A discussion of the conceptual model of the protection mechanism should clarify the point of extendability. Before proceeding to that discussion, though, it would perhaps be worthwhile to attempt to clarify further the purpose of the mechanism in general. There is, after all, a sophisticated apparatus in Multics for assigning "modes" to segments (see BG.9, BX.8). Why must there be a mechanism beyond modes? The answer, perhaps rather cryptically, is that modes are assigned per user, but shared supervisor segments are assigned per process. That is, user A can by a suitable choice of mode allow or prohibit user B's access to segment x which resides in A's file directory; but if <x> were, say, a segment of the Basic File System in B's working process, denying access by mode to B would be absurd. Nor would assigning execute only mode to all supervisor segments suffice, for there would still exist the possibility of <u>anyone's</u> calling supervisor routines - many of which, by their very nature, are only to be invoked under certain circumstances, and frequently only by other supervisor routines. Without beating the issue to death, then, let us simply observe that the protection of the supervisor in Multics requires a mechanism distinct from that of modes.

The Conceptual Model

Conceptually, the Multics protection mechanism is quite straightforward. Picture a series of concentric circles. Let all the segments in a process "live" somewhere in the picture, such that each segment is between the boundaries of some pair of circles, or in the innermost circle. Now label the areas contained by the circles, starting with the innermost, from R_0 to R_{63}. (The "R" stands for "ring", for reasons which probably become obvious after a glance at Figure 1.) The primary rule is that segments "in" the same ring have free access to one another, subject to any limitations prescribed by their modes. In anthropomorphic terms, you must trust the segments in your own ring. Access between rings is limited according to rules enunciated below. The first point to notice, however, is that once we have established the ring model, we provide for "walling off" ordinary user segments from those segments which belong to the supervisor by assigning the segments to different rings. Note, by the way, that at this level we are speaking of segments in general, without differentiating between procedure segments and data segments.

The primary rule of access between rings is that segments in lower-numbered rings have in general unlimited access to segments in higher-numbered rings, subject of course to mode restriction on particular segments, whereas segments in higher-numbered rings have no access to segments in lower-numbered rings except for cases where access is specifically granted by means discussed below. "Access" refers to both the ability to execute a segment and the ability to read or write it. Thus, from the outside of the ring structure looking in toward the central supervisor in ring 0, the ring boundaries are "walls". Recall that within a ring (which is to say, "between walls") life goes on unimpeded by the protection mechanism. It is when a wall must be crossed that the protection mechanism comes into play.

Those segments which comprise the central supervisor are in ring 0. It is useful to reserve ring 1 for system routines, largely administrative in nature, which are not so sensitive as the central supervisor and which cause less disastrous results in case of failure. The remainder of the low-order 32 rings are reserved for the system. The high-order 32 of the 64 rings provided for in the model are for user segments, although we will speak of them in general as being in ring 32. A built-in advantage of this structure is that users may avail themselves of "spheres of protection" just as the supervisor does. For example, an instructor might place his grading program in, say, ring 32, and invoke student-written procedures which are placed in, say, ring 33, with the assurance that his own program is secure from tampering by students.

Conventions and Terminology

The conceptual model just outlined represents the basis of the Multics protection mechanism for each process in Multics. Before presenting an overview of the supervisor procedures necessary to implement the model, it will be useful to present a list of the assumptions on which those procedures are predicated. The reminder of this section, then, may be viewed as a protection mechanism glossary. (The ordering, however, is not alphabetical; it attempts, rather, to be progressive.)

1. Rings. All the segments of a process in Multics are divided into a number of mutually exclusive subsets, called rings. A segment, <s>, is in one and only one ring.

Rings are numbered from 0 (the hard core supervisor's ring) to a possible maximum of 63. The lines between rings are called walls. If <s> is in ring n, its domain of access is those rings numbered from n to 63; it is denied access (in general, but see below) to segments in rings numbered from n-1 to 0. Thus, the hardcore supervisor has access to all segments of the process.

2. Wall-crossing. Control must, of course, be able to pass from ring to ring. After all, the segments which reside between the various walls do belong to the same process. The basic problem is how to make the system cognizant of the fact that a wall-crossing is being attempted, so that the crossing's legality can be checked. In broad terms, the solution to this basic problem lies in the construction of the descriptor segment for the process. The Basic File System, when producing an entry in a descriptor segment for a process, takes into account two factors: the ring in which control was at the time a missing-segment fault brought the file system into play, to create a descriptor for the segment in question, is one factor; the other is the Access Control List of the segment (see also BG.9). More on access control in Figure 2 and later; for now, suffice it to say that the file system produces entries in a descriptor segment such that, when control of the process is in a given ring, reference to segments in higher-numbered rings ("outwards") will produce an attempt-to-execute-data fault, and reference to segments in lower-numbered protection rings ("inwards") will produce a Directed Fault 2. These two faults are the protection, or wall-crossing, faults. The Fault Interceptor (see BK.3), on receipt of either of these faults, invokes the appropriate system procedure for legality checking and housekeeping.

3. Gates. By virtue of the ring structure's basic definitions, passing control by outward calls is legal. That is, segments in outer rings are accessible to those in inner rings. However, by virtue of those same definitions, we have yet to see a way in which an inward call could be legal. That is, segments in inner rings are in general inaccessible to those in outer rings. The means of legitimizing inward calls is to cause one or more entry points of a given procedure segment to be treated as "gates" in the protection wall. A gate, then, is an entry point to an inner ring procedure segment which may be called by an outer ring segment, using the system-standard call macros (BD.7.02, BD.7.03); gates are subject to certain refinements as

to range of rings which may call them, as described below. Gates are listed in segments' Access Control Lists (see BG.9, BX.8). For obvious reasons, the supervisor routine which the Fault Interceptor invokes to process wall-crossing faults is called the Gatekeeper (see below, and BD.9.01).

4. Doors. "Normal" returns after subroutine calls (that is, uses of the system standard return macro) are clearly in the province of the Gatekeeper. Because descriptor segments differ in different rings, inter-ring returns fall under the same faulting rules as do inter-ring calls; wall-crossing faults occur, the Fault Interceptor invokes the Gatekeeper, and all proceeds normally. However, not all returns are normal. Some subroutines must return to points other than immediately after the point they were called from. Without adding a great deal of apparatus which would in most cases be superfluous, the Gatekeeper cannot be in a position to monitor the legality of "abnormal" returns. Also, any mechanism for regulating abnormal returns could be misled by a transfer to a gate if the gate list were the sole source of access information available (this point is expanded upon in BD.9.05). Therefore, points at which abnormal inter-ring returns are to be considered legal must be specified in a similar fashion to gates, but must be distinguished from gates. As these points are also portals in the protection walls, they are called "doors". For unobvious reasons, the doorkeeping routine is called the Unwinder (see BD.9.05).

5. Brackets. To this point, the system of rings, gates, and doors has implicitly been treated in either-or terms: a segment is accessible to segments in its own ring and in inner rings, or it is inaccessible - unless it is a procedure segment with a gate or a door. In the interests of flexibility, however, a refinement can now be introduced which makes the issues somewhat less black and white. We define an "access bracket" for a segment to be a range of ring numbers within which range segments may access the segment in question as if they were in its own ring. That is, a segment with access bracket of 5:10 has the following characteristics: when the process containing it is "operating in" (that is, executing a procedure segment in) a ring from 5 to 10, access is governed only by the segment's mode; when the process is operating in a ring with ring number greater than 10, only transfer of control access may occur, and that only if to a gate or a door - with, of course, an inward wall-crossing fault. A further

useful refinement is to define a "call bracket" for a segment to be a range of ring numbers within which segments may attempt to transfer control to the segment in question, and beyond which not even the attempt to call is permitted. That is, calls from outside the call bracket will not succeed even if directed at a gate. To continue the previous example, consider a segment with the following "protection list" (see also BG.9):5:10:12. The first two numbers are the access bracket, as above; the last number is the upper limit of the call bracket (by definition, the call bracket begins immediately after the access bracket). Therefore, in addition to the access considerations already mentioned, the segment in question may only be called on an inward call from rings 11 and 12. Note that no prohibitions are set up against single-ring brackets: A protection list of 0 would define a segment accessible only in ring 0 (and the failure to define any gates for the segment would assure that it could only be called from ring 0); a protection list of 0:0:1 would define a segment accessible in ring 0 and callable only by ring 1. The command setacl (BX.8) is the user's means for making protection lists for his own segments.

6. <rtn_stk>. Throughout the protection mechanism, an item of particular interest is "the Gatekeeper's return stack". The Gatekeeper, recall, is the supervisor routine which monitors wall-crossings. In order to be able to deal with returns without the need for "return gates", the Gatekeeper preserves return information each time it is brought into play for a call in a given process. The information goes on a push-down stack in a per-process ring-0 segment called <rtn_stk>. A crucial point to note is that the <rtn_stk> may be pushed-down a number of times as a result of progressive calls before the corresponding returns take place. Further, the process may also have performed any number of intra-ring calls (and even intra-ring returns) between the times inter-ring calls are made which cause the return stack to be pushed. "Popping" of the <rtn_stk> occurs when the Gatekeeper is brought into play on inter-ring returns. (Of course, abnormal returns also require <rtn_stk> to be popped appropriately; see BD.9.05.)

7. Invocation number. Clearly, the protection mechanism must have an index to the current top of the <rtn_stk>. This index is called the "invocation number". When a process does make a series of inter-ring calls without the corresponding inter-ring returns, the invocation number

increases each time. Note that in some sense the invocation number may be taken to represent a period of residence of a process in a ring, for as noted above any number of intra-ring calls and returns may take place between inter-ring calls and returns - which is to say, between changes in invocation number. Control remains in the particular ring, in the sense that the segments which are executed are all in the ring. This "period of residence" aspect is quite important to the condition-handling mechanism (see below, and BD.9.04). The Gatekeeper maintains the invocation number for a <rtn_stk>, incrementing and storing it at <rtn_stk>|0 during calls and decrementing and storing it at <rtn_stk>|0 during returns.

8. Validation level. With the possibility of inter-ring calls open to the user, and inherent to the supervisor, it is frequently of interest to a procedure writer to be able to determine what ring his procedure was called from. Further, with the possibility of a procedure's being called from an outer ring as an interface to still another procedure in an inner ring, it could be useful to be able to specify during the second inward call that the call is being made in behalf of a routine in a ring other than the one from which that call came. Finally, inner-ring procedures have access to any segment in their rings; therefore, they must guard against the possibility that they have been called with arguments that are supposed to be in outer rings but in reality are in their own rings - and in order to validate arguments (see below, and BD.9.03), there must be a ring number to validate against. All these considerations lead to the establishing of a "validation level" as part of the protection mechanism. The validation level is a number (kept in a fixed, accessible place, as described below) which represents the ring number the current procedure was called from, or a ring number higher than that, so as to allow for calls in behalf of a farther-out ring. This number is monitored by the Gatekeeper, and is transmitted during inter-ring calls. (Naturally, a validation level less than the ring number the call is coming from will never be passed).

9. The Stack. Fundamental to Multics operation is the call-save-return "stack" (see BD.7). In a ring-structured environment, the Stack (capitalized to distinguish it from all the other stacks which abound in Multics) is actually implemented as one particular segment per ring. The segment for ring n would be called <stack_n>. For the user, the illusion is preserved by the protection

mechanism that there is only one Stack. In actual implementation, of course, this would not do, because one can always read and write one's own Stack segment and inner-ring data in the Stack must be protected - both from the point of view of privacy of data and from the point of view of assuring that the supervisor's call-return chaining in the Stack cannot be damaged by the user. The protection mechanism reserves certain locations at the base of each Stack for fixed purposes: sb|0 contains a pointer to the last Stack "frame" in use prior to a ring-crossing (this information is solely for the Gatekeeper's use), sb|2 contains the invocation number for the process the Stack belongs to (this information is primarily for the condition-handling mechanism's use; see below and BD.9.04), and sb|3 contains the validation level (this information is primarily for the use of validate_arg; see BD.9.03). Within a given Stack frame, the presence of a 1 in the op code portion of sp|16 indicates that the frame is a ring-crossing frame (see BD.9.01).

An important point to note about the Stack is that as a normal Multics segment it is in principle sharable. A consequence of this sharability permeates the implementation of the protection mechanism: Those components of the protection mechanism which employ data furnished by user procedures, particularly argument lists, must in general copy the data into secure, inner-ring segments. The reason for this copying is that it is possible for another process to acquire control (on a time-slice termination) and to alter information "out from under" the interrupted protection mechanism routine if the information resides in a segment (particularly a Stack segment) which the new process is sharing with the interrupted process. So when certain data must be validated, as frequently occurs, the validation must be performed on a secure copy, and not on a potentially-changeable one. We make this point at the overview level to offer explanation and motivation for what would otherwise be rather cryptic tactics in the implementation descriptions in the sections which follow.

The Components of the Protection Mechanism

The major procedures which comprise the protection mechanism have been alluded to above. Although the reader who is interested in their details will of course turn to the discussions in BD.9.01 - BD.9.06, brief overviews of their functions are presented here, for introductory purposes and for the benefit of those readers who do not need to pursue the intricacies of the protection mechanism at this time.

Central to the protection mechanism is the Gatekeeper (BD.9.01). Viewable as the fault-handler for the protection faults, the Gatekeeper has as its primary role the legality-checking of the wall-crossing which caused the fault it was invoked in response to. For legal wall-crossings, the Gatekeeper must arrange the threading of frames between the Stack segments involved, update the <rtn_stk> so that returns correspond to calls properly, and maintain the invocation number and the validation level appropriately. The Gatekeeper is essentially invisible to the user (it can only be called by the Fault Interceptor, which is a ring-0 routine which, in turn, can only be invoked through a fault). For the management of arguments, the Gatekeeper employs arg_pull and arg_push (BD.9.02). The point at issue here is that outward calls' arguments must be copied into the outer, target ring, for they are by definition inaccessible otherwise; on the corresponding inward returns, return arguments must be copied from the outer ring where they were generated into the inner ring where they are expected. Arg_pull and arg_push themselves have recourse to a "visible" (user-callable) routine which is also part of the protection mechanism: validate_arg (BD.9.03) checks an argument list to determine whether or not all the segments pointed to by it are accessible from a given ring.

A second large area of the protection mechanism is that which deals with the Multics primitives for "condition-handling". The notions of conditions and signals are similar to those in PL/I. BD.9.04, on the <u>condition</u>, <u>reversion</u>, and <u>signal</u> routines, offers more detail, but at this level suffice it to say that during the course of a process a condition may be encountered which requires special handling, outside the normal flow of control. Calls to <u>condition</u> establish handlers for invocation if and when named conditions occur. Calls to <u>signal</u> declare that named conditions have occurred and cause the most recently established handler to be invoked. (Calls to <u>reversion</u> cause the disestablishing of a condition handler.) It is important to note that fault-handling in Multics (see also BK.3) has been subsumed under condition-handling. That is, most hardware faults are turned by the Fault Interceptor into signals of appropriately-named conditions. Because of this policy, and because the invocation of condition handler procedures for conditions which do not arise from faults can also become involved with ring-crossing considerations, the condition-handling mechanism must be part of the protection mechanism. To anticipate the detailed discussion of condition handling a bit, it is perhaps interesting to note at the overview level that the importance of the invocation number lies in the area of determining which condition handler was established most recently: Handlers are stacked on push-down stacks on a per-ring basis. Therefore, when the most recently established handler is being searched for, the entry on the top of the push-down stack for the ring at hand (which entry contains the invocation number which applied when the handler was established) must be investigated. If the invocation number in the top entry does not agree with the invocation number at the time of the signal, the invocation number at the time of the signal is progressively decremented (once for each ring crossing in the as-yet-unsatisfied returns indicated in the <rtn_stk>) until the invocation number of a handler at the top of a stack matches it. The privitives, being callable from any ring, must be able to determine which ring they are operating in during an invocation; to this end, procedure get_ring_no (BD.9.06) is furnished.

The last major part of the protection mechanism is the Unwinder (BD.9.05). This routine is invoked by the user to perform "abnormal returns" (exit from a subroutine to a point other than where it was called from). From the point of view of the protection mechanism the primary roles of the Unwinder are to adjust the <rtn_stk> properly (for normal inter-ring returns may be bypassed during an abnormal return) and to release the Stack frames which are being bypassed. From the point of view of the user, the role of the Unwinder is to invoke any procedures which he has specified to be executed in the event of an abnormal return past a procedure. That is, a procedure, say p, may call another procedure, which in turn calls others, and sometime before the normal return to the first procedure in the sub-chain an abnormal return may be taken to a point in a procedure which is an "ancestor" of p in the call-chain; in such a case, p's storage might need to be freed, or the like. The condition primitive is used to place these "unfinished business" procedures in a known place, under the reserved condition name "cleanup".

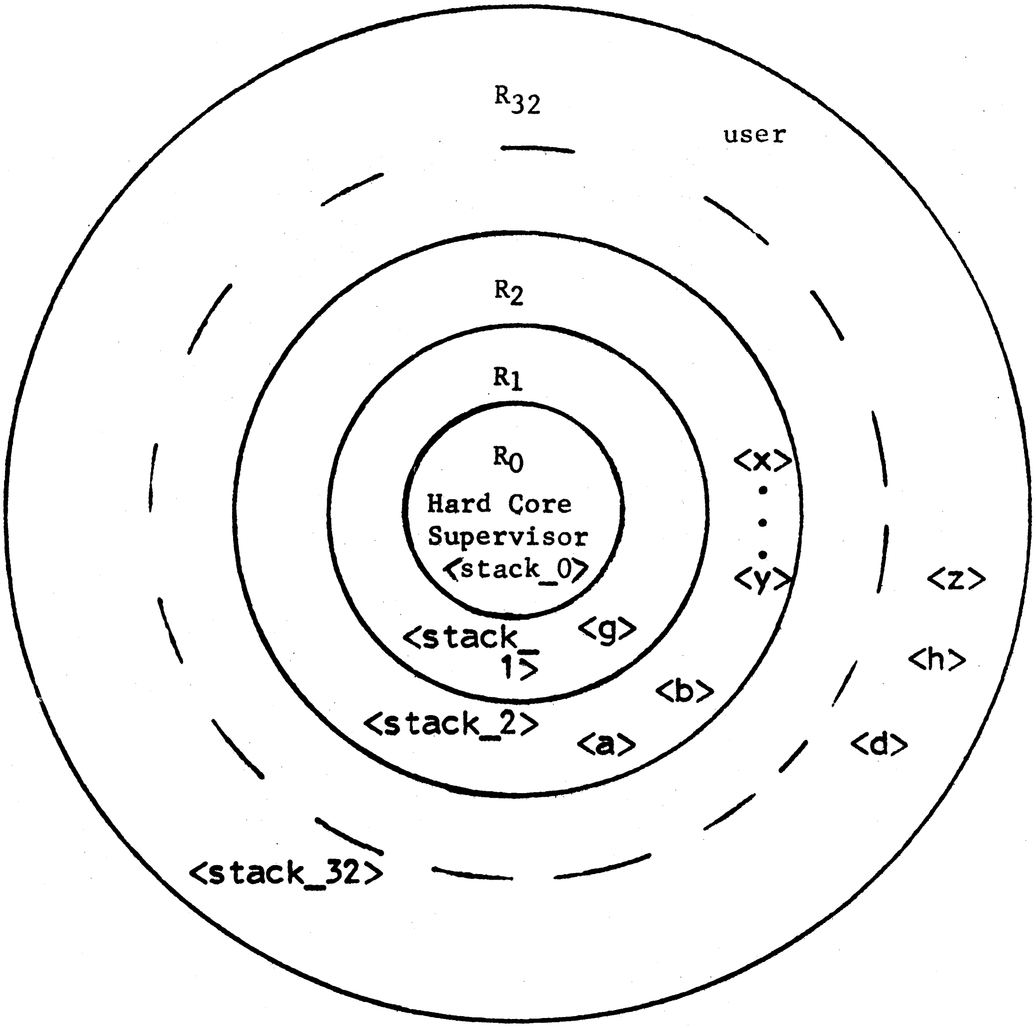

Figure 1: Division of the Segments in a Process Into Subsets, Called Rings

Figure 2: Access Controls in the D(i) for Figure 1.

Descriptor segments

Rings		D(32)	D(2)	D(1)
R(32)	<d>	proc slave access	data slave access	data slave access
	<h>	proc slave access	data slave access	data slave access
	<z>	data slave access	data slave access	data slave access
R(2)	<a>	directed fault	proc slave access	data slave access
	<b>	directed fault	proc slave access	data slave access
	<y>	master access only	data slave access	data slave access
	<x>	master access only	data slave access	data slave access
R(1)	<g>	directed fault	directed fault	proc slave access

There is a distinct descriptor segment, D(i), associated with each ring, R(i). The contents of all the descriptor segments are identical, except possibly the access control bits, i.e., the kth descriptor in each D(i) refers to the same segment. When control is in R(i) the descriptor base register, DBR, points to D(i). The domain of access of a segment in R(i) is defined by the access control bits of the descriptors in D(i). Figure 2 shows the access control of the D(i) for the example in Figure 1. When control is in R(i) only those procedures which are in R(i) are marked procedure in D(i). Any attempt to transfer control to a procedure not in R(i) results in a fault. In this fashion all crossings of a wall are detected. Inward crossings are detected by a directed fault and outward crossings are detected by an attempt-to-execute-data fault. When a wall is crossed and control passes to R(i) the stack is switched by the Gatekeeper and the DBR is set by the Basic File System to point to D(i). This changing of effective descriptor segment accomplishes the locking or unlocking of the appropriate segments.