Published: 09/29/67

## Identification

Overview of Intersegment Linkage
R. M. Graham, M. A. Padlipsky

## Purpose

"Linkage", in the broad sense of communication between separately-assembled subroutines, is of course of fundamental importance to any operating system. Conventionally, provision is made prior to execution for supplying addresses which can not be known at assembly-time, thus requiring expenditure of time to locate all external references regardless of whether or not they will be needed in a particular run. In Multics, however, linkage is typically performed dynamically - that is, the address for an external reference is found by the System during the execution of a process, at the point in time when the address is known to be needed. (Pre-linking, called "binding", can also be effected; see BD.2.) Dynamic linking, by eliminating calculation of addresses which would not be needed during a given run, should lead to considerable savings in execution time; experience with CTSS indicates that a MAD program which takes a second or two to compile, and a like amount of time to execute, still probably requires four to six seconds to "load" - where loading means essentially the same thing as linking.

A second aspect of linkage is the area of subroutine calls and returns, in the sense of conventions for performing the call and return rather than simply of acquiring the address of the routine being called. Carefully-planned call, save, and return macros are essential in Multics, in order to support the design goal of furnishing "pure" procedures. (A pure procedure is one which is not altered at all during execution - hence, a single copy is shareable amongst several users in a time-sharing environment.) Further, procedures should be "recursive"; that is, they should be capable of being re-called, either directly by themselves, or indirectly during a series of calls to other procedures, without confusion of "generations" of storage. The call, save, and return conventions must also support this latter design goal.

## Basic Mechanisms

For a topic such as dynamic linking, a dynamic overview should be particularly appropriate - that is, an overview which covers the process of dynamic linking from the point of view of the interactions of linking with the rest of the System - rather than a "static" overview, which only

deals in the abstract with the mechanisms involved in linking. Before turning to such an overview, however, there are two basic mechanisms which should be treated in the abstract, if only to allow the introduction of a notation which will be useful later.

The sine qua non of the Multics approach to linkage is the "linkage segment". Ordinarily, each segment, procedure and data, has associated with it another segment; when associated with a procedure segment, this (linkage) segment's primary role is to contain pointers (ITS or ITB pairs) to external addresses (addresses not in the procedure segment), thus allowing the procedure segment to remain "pure" (i.e., contain no information which varies from run to run) by doing all its external addressing indirectly. How these indirection pointers, or "links", are created is, of course, a major issue in the forthcoming discussion. (That portion of the linkage segment in which the links reside is called the "linkage section".) To the programmer, though, external references, as well as internal locations which are permitted to be referenced by other procedures, need only be known symbolically. The linkage segment may also be used for storage of the character strings which represent the symbolic knowledge of these intersegment references, or "link definitions". ("Link definitions", which do not change, may in some circumstances be stored in the procedure segment.) The linkage segment, then, has a part in references out of and into procedure segments, and of course, into data segments. Another role it plays also contributes to the issue of keeping procedure segments pure (and hence shareable): With the changeable information involved in intersegment references taken care of by placing it in the linkage section of the linkage segment, there still remains the problem of keeping any variables which a procedure segment has separate from the procedure segment itself. A convenient place for storage of that class of variables called "internal static" in PL/I turns out to be the linkage segment. (Other classes of variables are more conveniently stored in other places; one of the most important of these other places is discussed below.) Details of the structure of the linkage segment are to be found in BD.7.01.

A second sine qua non of the Multics approach to linkage is that mechanism known as the call-save-return Stack. The primary role of the Stack is to furnish push-down storage for the information which is necessary to allow procedures to call and be called by (that is, return to) other procedures. The Stack must be in a segment other than the procedure segment in order to preserve the pure-ness of the procedure segment. It must be a push-down stack in order to satisfy the design goal of recursiveness.

System-standard call, save, and return macros assume the existence of and operate upon the Stack, in a fashion which will be dealt with in more detail in the "dynamic" portion of this Overview. For now, however, these are a few further structural points which should be made. When a procedure is called with the Multics call macro, that macro creates a "frame" in the Stack for the called procedure. There are two portions in the frame: a fixed-ler portion, and a variable-length portion. In the fixed-length portion, chaining is preserved from frame to frame by means of forward and backward frame pointers. This chaining is important because successive calls push down the Stack in such a fashion that the corresponding "normal" returns (performed by the Multics return macro) pop off the procedures' frames in order, but "abnormal" returns (which do not go back to the caller at the point immediately after the call - see also BD.9.05) may require the process at hand to return to the procedure which is associated with a stack frame other than the one which immediately precedes the current one. In the variable-length portion of a Stack frame, storage space is provided for that class of variable known as "automatic" in PL/I. Thus, if a procedure calls itself or is called recursively, the Stack is pushed down by the call and the new frame is used for storing a new "generation" of automatic variables - again leaving the procedure segment pure. Details of the structure of the Stack and the implementation of the call, save, and return macros are to be found in BD.7.02 and BD.7.03.

## Notation

An external symbol is one which may be referenced from segments other than the one in which it is defined. Segment names (strictly speaking, "reference names") are written <seg name> and external symbols are written [ext symbol]. If [x] is defined in segment <s> then <s>|[x] is a reference from oustside <s> to the location [x] in segment <s>, where the value of [x] is defined in the linkage section of <s> and is not known by another segment until that segment actually tries to reference it during execution. However, <s>|x is a reference from outside <s> to the location x in segment <s>, where the value of x is known at translation time. If <s> is a segment, <s.l>|s.lp will refer to the origin of the linkage section for <s> and s# will be the segment number of <s>.

## System-wide Interactions

We now turn to the dynamic portion of the overview, which may be thought of as "Some Notes on the Care and Feeding of Segments".

Background (I): Addressing

The ins and outs of 645 addressing comprise a rather complex area, partially because of the history of the hardware development. However, it is not necessary to understand the subject in detail in order to abstract out the key points of the process from the perspective System treatment of segments. What we are interested in here is the means by which the hardware is made to interact with the software when segments are involved.

The first hardware feature of 645 addressing which is important to the non-hardware treatment of segments is the "fault tag". It is possible to assemble an indicator (called a Fault Tag) into an appropriate location in a machine word such that the processor, when reading that word as an address in an indirect address chain, will sense its presence and take a fault. (Indeed, indicators exist to distinguish three distinct Fault Tags, known as 1, 2, and 3.) Suffice it for now to say of the faulting process that once the hardware encounters a fault-producing condition the system gets its "hands" on the existing situation, and this is exactly what we want in terms of segment handling, as will be seen shortly.

There is one other aspect of the hardware addressing process which should be presented as background to this discussion. Procedures, which "live in" segments, can reference other ("external") segments, either as data or as targets of transfers. In the appending (i.e., address-generating) process for external references, two entities come into play which are of interest, and which taken together constitute a topic which we will call "descriptors". There is a base register called the Descriptor Segment Base Register used in the appending cycle for external references. This base register can only be set by the System. It contains a pointer to a Descriptor Segment. The contents of this segment also can only be set by the System. Now the hardware simply trusts the System to have provided appropriate information in the Descriptor Segment, because in the appending process it takes whichever entry in that segment is indicated by the combination of the base pointing to it and a base containing the "segment number" (actually, this may be thought of as a index into the Descriptor Segment) and proceeds according to address and control information contained in that entry. The upshot of it all is that the Descriptor Segment contains control information which is taken into account by the hardware. Specifically, the control information may be set (by the System) to

indicate that the referenced segment is to be treated as data (and a fault will take place if an attempt is made to execute it), or is only to be executed (and a fault will take place if an attempt is made to treat it as data), or is to be responded to by a "Directed Fault" if referenced. So in the Descriptor Segment we have a second way for the System to "get its hands" on a situation involving segment references, the Fault Tag having been the first.

Background(II): Software

We have seen above how the hardware provides "handles" for the software when segments are addressed. It turns out that the manipulating of Descriptor Segments is essentially dynamic and is better left for discussion below, as an aspect of execution. The use of the Fault Tag, however, comes into play at assembly/compilation time, and will be dealt with here.

In Multics, standard assemblers and compilers must reference external segments by means of a "linkage section". Recall that the linkage section is a known area in the linkage segment where the indirection words used by a procedure for external segment addressing are placed, by the assembler/compiler. The linkage segment's number is kept in a specific external base register (lb), and the offset within the linkage section is kept in the paired internal base (lp). That is, at any point in time the linkage section of the executing procedure is referenceable through lb←lp. How this is accomplished will be covered below. Hence, in view of the fact that the actual location and number of an external segment are unknown at translation time, a reference to an external segment is generated as follows: In the procedure, the operand of the appropriate instruction is an address in the linkage section and an indirection tag. In fixed positions in the entry in the linkage section, a Fault Tag indicator, a pointer to the symbolic name of the external segment, and any offset (symbolic or numeric) within that segment are indicated. Then, if and when the generated code is executed the appropriate fault will be taken by the hardware, and the System will come into play to find the desired segment--"knowing" the symbolic name and the offset.

Execution(I): First Reference

The foregoing background information should be enough to allow an investigation of what happens in Multics when a procedure segment, <proc>, references an external segment, <ext>, for the first time during execution. (Distinguishing between the cases when ext is procedure and when ext is data will be postponed as long as possible.)

## Fielding the Fault

The first thing that happens on the first reference to ext is that the hardware encounters the Fault Tag 2 duly planted in <proc>'s linkage section by the translator. This fault is taken and the Fault Interceptor Module (BK.3) takes over. The Fault Interceptor recognizes the Fault Tag 2 as being (by definition) a linkage fault and calls the Linker (BD.7.04).

## The Linker

The role of the Linker basically is to "fill in the blanks" in the linkage section of a procedure which had a linkage fault. That is, the Linker must take the symbolic name of the external segment which was referenced ("ext") and somehow come up with an ITS pair to <ext> which it can put into the linkage section of <proc> in place of the words which caused the fault. Actually, the Linker handles only the mechanics of this job--it "knows" the format of linkage sections and can extract and plant information there--but it passes the task of finding the segment in question along to the Segment Management Module (BD.3).

## Segment Management

The Segment Management Module (BD.3) is the primary interface to the Basic File System (BG.). In a way, its role may be thought of as the maintaining of the Segment Name Table, which on a grossly oversimplified level is a table of symbolic names of segments and corresponding segment numbers, for a "process" in Multics. (Segment numbers, of course, are indispensable parts of the ITS pair we are trying to build to put into <proc>'s linkage section.) So names go into Segment Housekeeping and numbers come out. To get the number, if the name is not already in the SNT, the Basic File system is called appropriately, where "appropriately" is meant to cover the possibility of specification of file directories to search and order to search them in (Search Module, BD.4).

## Basic File System

Once the Basic File System is entered, all manner of tables must be updated and all manner of details must be attended to. For the purposes of this discussion, only a few salient, qualitative points will be dealt with--and those largely in terms of effects rather causes. There are two basic tasks performed by the Basic File System at this point in the segment-getting process. First, it must come up with a segment number to be passed back to the Linker.

Second, it must make the segment "known", by recording the segment number and the segment's mode in the Known Segment Table. Now, the mode comes from the branch in the appropriate directory; how all this comes about is irrelevant here. At any rate, a mode is found and recorded for future use, and the segment number is returned to Segment Housekeeping, which records the number and the symbolic name ("ext") in the Segment Name Table and in turn returns the number to the Linker.

Linker

If an offset is involved, the Linker will have to repeat the above process for <ext>'s linkage segment and will then have to continue the segment-getting process in order to read the linkage section and be able to compute the offset from information in <ext.1> (<ext>'s linkage segment). For our purposes, only the result of this process matters. Assume that the Linker now is in possession of the necessary information to produce the desired ITS pair and places it in <proc>'s linkage section at the location where the linkage fault (FT2) arose. After altering the machine conditions which were stored on entry so that execution of the previously faulting instruction can continue when the Fault Interceptor restores the machine conditions, the Linker then returns to the Fault Interceptor, which called it. The Fault Interceptor then restores the machine conditions.

Procedure

Control is back in <proc> again, but not for long. The fact of the process of linking, after all, does not necessarily imply the desire for using the linked-to segment. Hence, the System does not finish the process of making the segment accessible until some indication occurs that access is really wanted. The means by which the indication is given brings us back to the Descriptor Segment. <proc> is, of course, a procedure in some working process; and that process, of course, has some Descriptor Segment associated with it. As a matter of fact, that Descriptor Segment is whichever one is pointed to by the Descriptor Segment Base Register, which only a System program can set. So a vital aspect of giving control to a process is setting up an appropriate Descriptor Segment Base Register and Descriptor Segment. This sort of thing was taken care of by the System prior to the period we are investigating; but, of course, no provision could be made then for an entry pertaining to <ext>. No explicit provision, that is. However, Descriptor Segments are always constructed

in such a fashion that "all the rest"--that is, any segment not otherwise already dealt with--causes a particular kind of fault when the appending mechanism uses the descriptor. This fault is the "missing segment fault". When it occurs, off we go again through the fault handling software to the Basic File System.

Basic File System

Once again to do an injustice to the intricacies involved, we may view this trip to the Basic File System as having two purposes. First, the segment is made "active". For our purposes, this means that certain entries are made in certain tables so that the file system's books balance - and it's necessary to be active before you can be used (if "you" are <ext>). More interesting to us is the second purpose: the production of an appropriate Descriptor Segment entry for <ext>. This is a tricky process. Briefly, there are two broad areas which are considered: modes and protection rings, information about which is maintained in the Access Control List of the file hierarchy "branch" for <ext>. The descriptor is generated such that <ext>'s mode with respect to <proc> is taken into account, and such that one of two faults will occur if <ext> is called by <proc> and they are not in the same ring. (Actually, <proc> may be in a ring which is not in the range of permission specified by <ext> if this is the case, a descriptor is generated which will cause a "no access" fault.) If <ext> is in a lower-numbered ring than <proc> provision is made for a Directed Fault 2 to occur when the descriptor is used; if <ext> is in a higher-numbered ring than <proc>, provision is made for an attempt to execute data fault to occur when the descriptor is used. Of course, if <ext> and <proc> are in the same ring no extra effort is taken with the descriptor and only the mode of <ext> is operative. Finally, if <ext> were being referenced as data rather than being called, again no extra effort is taken and only the mode is operative. At any rate, an entry is made in the Descriptor Segment, and back we go through the Fault Interceptor to <proc>.

Protection Mechanism

The story is now over in the cases mentioned above where "no extra effort" was taken in generating a descriptor. However, if <ext> was being called from a different protection ring than its own, one of the two planted faults described above will take us through the fault handling mechanism to the Gatekeeper (BD.9.01), which is the primary organ of the Multics protection mechanism. Let us assume that we are indeed dealing with a "ring-crossing". The Gatekeeper

has two basic tasks: verification and housekeeping. First, the crossing must be checked for legality. The file system is called upon to determine that the address being transferred to is a listed entry point of the target procedure (and to specify the ring number of <ext>). Second, provision must be made for enabling <proc>'s process to operate in <ext>'s ring. This provision includes switching Stacks from the one ring to the other. (The needs of the protection mechanism dictate that "the Stack" - which is conceptually a single entity from the user's point of view - is actually implemented as a separate segment for each protection ring the process enters; see BD.9.00.) Information which will be necessary when <ext> returns to <proc> is also noted. Then the fault handling mechanism is returned to (by the Gatekeeper) and suitably altered machine conditions are restored so that execution of the transfer to <ext> can continue.

Procedure

Whether or not the protection mechanism comes into play, <ext>'s status as a segment in the address space of <proc>'s process is now secure. From the point of view of the care and feeding of segments, we can consider <ext> weaned; the first reference is done.

Execution(II): Recursive References

If the same portion of <proc>'s linkage section is recursively referenced after linkage has taken place, things go smoothly. All of the above considerations are bypassed--linking, searching, "knowing", "activating", and the like--and the only impediment to direct access to <ext> is the protection mechanism, if applicable, for of course the descriptor of <ext> in <proc>'s ring's Descriptor Segment is still set up to trigger any protection faults it may have triggered at the first reference. This is as it should be, for there is no guarantee that the reference still is legitimate: a legitimate call to a protected procedure could have been followed by some sort of unscrupulous diddling of, say, the linkage section (in particular of, say, the offset), so there's no guarantee that just because the segment <ext> has been previously referenced legally, subsequent references will also be legal.

Execution(III): Other References

A final interesting note is that when some other point in <proc> refers to some other point in <ext>, considerable savings in time are achieved in the linking processing. This comes about in the Segment Management Module, which discovers that <ext> is in the Segment Name Table and does not need to go off to the file system to search for it, but can return the segment number directly to the Linker.

## Calls, Saves, and Returns

The foregoing description deserves some amplification in the area of calls and returns. Suppose that <proc> calls <ext>. Previously, when <proc> itself was called, the base pair sb←sp was set to point to the beginning of a Stack frame for <proc>. Call the beginning of that frame sp_proc, and for the purposes of this discussion call the Stack segment <stack>. The call macro begins by saving the current values of the 645 bases and registers: bases at <stack>|sp_proc+0 (eight locations' worth) and registers at <stack>|sp_proc+7 (eight locations' worth). In other words, bases and registers are saved (for subsequent restoration, when the called procedure returns to the calling procedure) at the beginning of the current Stack frame. Next, an ITS pointer to the argument list is stored into ab←ap. Then the return location information (essentially a pointer to the current location plus two) is stored at <stack>|sp_proc+20, and we transfer to <ext>. (As no particular entry is specified, the transfer will by convention be to <ext>|[ext], actually.) The transfer is really an indirect one, through the linkage section of <proc>, and generation of an address for <ext>|[ext] proceeds essentially as described above. However, to get the linkage pointer set to <ext.link>, and to execute the standard save sequence, it turns out that the address generated should be to an appropriate point in <ext.link> rather than in <ext>. The Linker determines that it is to generate an address in <ext.link> rather than <ext> on the basis of a "class code" in the external symbol definition for [ext]. (Class codes exist for causing the Linker to interpret the value in an external symbol definition as 1) an offset from the top of the segment in question, or 2) an offset from the top of that segment's linkage segment, or 3) an offset from the top of that segment's symbol table segment - see BD.1 regarding symbol tables; see BD.7.01 for details of class codes.)

Because [ext] must be declared to be an entry point (either explicitly or implicitly), its translator will have made preparations for a save sequence at [ext] in the following fashion: Two instructions are placed in the linkage segment (<ext.link>) to be executed when <ext>|[ext] is called. The first of these switches lb←lp from <proc.link> to <ext.link>, by means of a rather tricky use of the 645 eaplp instruction (BD.7.02 will repay close study on this point). The second instruction is an indirect transfer through a link in <ext.link> to [ext] in <ext>. This link, of course, will also cause the Linker to come into play on the first reference through it; that is, the indirect transfer is through a word pair containing an FT2 modifier

and pointers to a linkage definition, just as the transfer through <proc.link> was. In <ext> itself, then, the save sequence continues as follows: The first issue to deal with is the chaining of the Stack frames. <stack>|sp_proc+18 was set on entry to <proc> to point to the next frame's origin; that is, to directly after the variable-length portion of <proc>'s frame. So sp_proc, as the location of what is to become the last frame, must be stored into a conventional location in the new frame (call the beginning of the new frame sp_exp) in order to maintain back-chaining. The location is defined as <stack>|sp_ext+16. Then the length of <ext>'s frame must be calculated, in order to set <stack>|sp_ext+18, on the basis of information provided by the translator of <ext>. Details of this undertaking are to be found in BD.7.02. When the length of <ext>'s frame is known, sb←sp is set to point to sp_ext, and the argument list pointer is preserved at <stack>|sp_ext+26. Execution in <ext> may then proceed. In the event that <ext> performs any calls, the call and save sequences as just discussed will come into play - the effect being to push down the Stack by another frame for each (unreturned-from) call.

When it comes time for <ext> to return to <proc>, the return sequence simply loads the base registers indirectly through sp_ext+16 - that is, from the beginning of <proc>'s Stack frame, where <proc>'s bases were stored when <proc> called <ext>. Then sb←sp points to sp_proc again (and lb←lp points to <proc.link> again), and the registers may be restored directly from sp_proc+8. Finally, control is returned to <proc> at the point immediately after it left by means of 645 rtcd instruction, using the return location information which was stored at sp_proc+20 during the call. The return sequence pops off <ext>'s frame from the Stack. (Note that we have covered here only the normal return; for "abnormal" returns see BD.9.05.)

Details of the extensions to the above basic process for the execute only and master mode cases may be found in BD.7.03.

Linkage Utility Routines

Having concluded the "dynamic" portion of this Overview of linkage, we must make note of one more point before concluding the section at hand: Beyond the "automatic" aspects of linking as discussed above and in the subsections of BD.7, there also exist aspects which may be though of as "manual". That is, the Multics user may directly manipulate linkage segments via subroutine calls. The Linker itself may be called (see BD.7.04), and various utility routines exist (see BY.13).

Summary

The foregoing discussion having been somewhat diffuse, we conclude with a summary of the major points touched upon:

From the programmer's point of view, references to segments other than the current procedure segment are performed by specifying the symbolic names of the segments and locations (i.e., entry points or variable names) in question. From the System's point of view, such references are accomplished by means of indirect addressing through "links", where links are machine addresses created during execution of the user's procedure by a System procedure known as the Linker. The linkage segment approach allows for the inter-user sharing of pure procedure segments. Before a link has been established, its position in the linkage section is occupied by a word pair which will cause a particular fault when referenced, and by pointers to the "link definition" - that is, to the symbolic name(s) involved. When a linkage fault occurs, it is handled by the Linker, which establishes links, replacing the symbolic information with machine addresses (ITS pairs). It accomplishes this by getting a segment number for the reference name in question from the Segment Management Module (which may in turn employ the Basic File System), and by computing an appropriate offset if need be.

For subroutine calls, standard call, save, and return macros must be used for all intersegment communication involving System segments. These macros manipulate a Stack segment in such a fashion as to allow for recursive procedures, and support the address base register conventions discussed below. During calls, the Stack is pushed down; during returns, the Stack is popped up. There is a Stack "frame" for each call made.

Finally, certain hardware facts and conventions should be mentioned: First, it should be noted that the descriptor segment (that segment pointed to by the descriptor base register) defines the address space of a process in Multics; introduction of segments into the descriptor segment is managed only by the System. The address space is two-dimensional, with segment numbers representing one dimension and offsets within segments representing the other. Second, the address base registers of the 645 are always paired and assigned functions in the following fashion: ab←ap points to the argument list on calls; bb←bp is an unreserved pair, for use by compilers and by user programs; lb←lp points to the origin of the linkage section of the currently-executing procedure; sb←sp points to the current frame of the call-save-return Stack. Last, Fault Tag 2 is reserved to indicate linkage faults.