Published: 02/05/68

Identification

Combined Linkage Segments
R. C. Daley, M. A. Padlipsky

Purpose

The one segment/one linkage segment approach implied by the foregoing BD.7 sections is modified in actual practice, for in a paged environment separate linkage segments result in large losses of core space due to "breakage" (i.e., space left over in a page because segment size need not fill an integral number of pages). Further, separate linkage segments necessitate space expenditures in terms of page tables and descriptor segment entries. To prevent such inefficiencies, Multics will, in general, combine, on a per-protection-ring basis, the linkage segments of the various segments in a given protection ring, into a single, combined linkage segment. (Note that combining is performed only for segments strictly within a protection ring; segments which contain protection mechanism "gates" must have their linkage segments handled differently. See the discussions of gates in sections BD.9 and BG.9.)

Discussion

For a combined linkage segment to be usable without introducing severe inefficiences of time spent in searching it, as well as to avoid potential naming conflicts, the origins of the constituent linkage segments must remain known. Therefore, whenever a particular linkage segment is appended to a combined linkage segment, a table, known as the Linkage Offset Table (LOT), is updated to indicate the relative position (within the combined linkage segment) of the new addition. There is one LOT per protection ring; they are maintained by procedure lot_maintainer. The lot_maintainer routine also contains an entry point for causing the combining of a particular linkage segment into a combined linkage segment. An LOT is indexed by the segment number of the text segment in question and the entry is a pointer to the beginning of that (text) segment's linkage information. The linkage information is generally, but not necessarily, within the combined linkage segment for the given ring. That is, for text segment n, the nth entry in the LOT contains the lb←lp value for n. The entry points to lot_maintainer and their calling sequences are discussed below, under Usage.

There are only three areas of the system which call lot_maintainer directly. Two of these are discussed elsewhere: the system initialization pre-linker (BL.7) and the process initialization pre-linker (BJ.9) both must assume responsibility for combining linkage segments and for LOT maintenance. The third user of lot_maintainer is that module of the Linker which is responsible for deciding whether a given linkage segment is to be combined, as well as being responsible for managing the combining. This module is link_man, the entry points and calling sequences of which are also documented below. Essentially, the role of link_man is to furnish the Linker with a linkage pointer for a given text segment; at least, that is its role from the viewpoint of the Linker. In the context of the present section, however, link_man may perhaps more accurately be thought of as a - or even the - primary user of lot_maintainer.

Figure 1 attempts to present a schematic view of the LOT's role. As indicated in it, the segments numbered m and n have had their linkage information combined into a Combined Linkage Segment (CLS); hence, the mth and nth entries in the LOT are pointers to the appropriate places in the CLS. Both of these pointers are usable as linkage pairs when an lp is required, for segment numbers m and n, respectively. However, as has been noted, not all linkage segments should be combined into the CLS. This condition is indicated in the figure by the pth entry: it is a pointer to the (separate) linkage segment for segment number p. Another way of putting it is that the segment numbers in the pointers which constitute the mth and nth entries in the LOT will be the same - CLS# - while the offsets in the mth and nth entries will differ; whereas the segment number in the pointer which constitutes the pth entry will be that of <p#.link>.

Somewhat more detail can now be given as to how link_man functions. For the sake of convenience, temporarily ignore protection ring considerations. Consider the first time a segment, say <s>, is called in a process. The Linker will require a linkage pair value for <s>. Therefore, it calls link_man, which will find <s.link>, combine it into the CLS (by assumption), record a pointer to where the linkage information has been combined as the s#th entry in the LOT, and return that pointer to the Linker. Any other references to <s> will then follow the Linker - link_man route, but in these cases link_man will discover that it already has a record of the lp for <s> in the LOT and will return that pointer without further ado.

The addition of protection ring considerations requires certain extensions to the basic logic, as detailed below; however, the main point to note at this level of discussion is that when there is one LOT per protection ring, it is necessary to record copies of the pointer to <s>'s linkage information in the LOT of each ring from which <s> is referenced, when <s> is referenced. That is, in most cases it will turn out that after the first reference to a segment its lp will be that value found in the LOT of the ring in which it was first referenced. For the exceptions to this, and for details of implementation, see below. Suffice it for now to observe that link_man in general uses the lot_maintainer to record lp values, to combine linkage segments (when necessary), and to retrieve lp values.

Figure 2 depicts the situation which occurs when a segment has an access bracket. Within the access bracket, the LOT entry points to linkage information (combined or not, as the case may be) within the ring at hand. Above the access bracket, the LOT entry points to linkage information in the high ring of the access bracket; note that the fact that a pointer is available does not necessarily imply that the data are accessible from the outer ring - indeed, the data are generally not accessible, although the presence of a call bracket for the linkage segment in question may permit the linkage segment to be transferred to. Below the access bracket, the LOT entry points to linkage information in the low ring of the access bracket (in this case, the data are, by protection mechanism definition, accessible from the lower ring).

The reason why call-bracketed segments cannot have their linkage segments combined into the CLS for a given protection ring is implicit in the above. A Combined Linkage Segment is, after all, a single segment. Therefore, if it were to have any call bracket at all, it could only have a single call bracket. However, it contains the linkage information for many segments, and if those segments had call brackets this fact could not be expressed in the one call bracket available. So the linkage segments with call brackets are not combined into CLS's; in this way, each can have its own call bracket. (For the benefit of the perceptive reader who has perhaps wondered why it would not do to have the call bracket apply only to the procedure segment being called, and not to its then-combinable linkage segment, it should be pointed out that the Multics protection mechanism is predicated on the assumption that ring-crossing faults will take place when the linkage segment is referenced. This constraint is imposed, as a matter of fact, precisely to guarantee that the proper linkage pair is used on inter-ring calls. Cf. BD.9.)

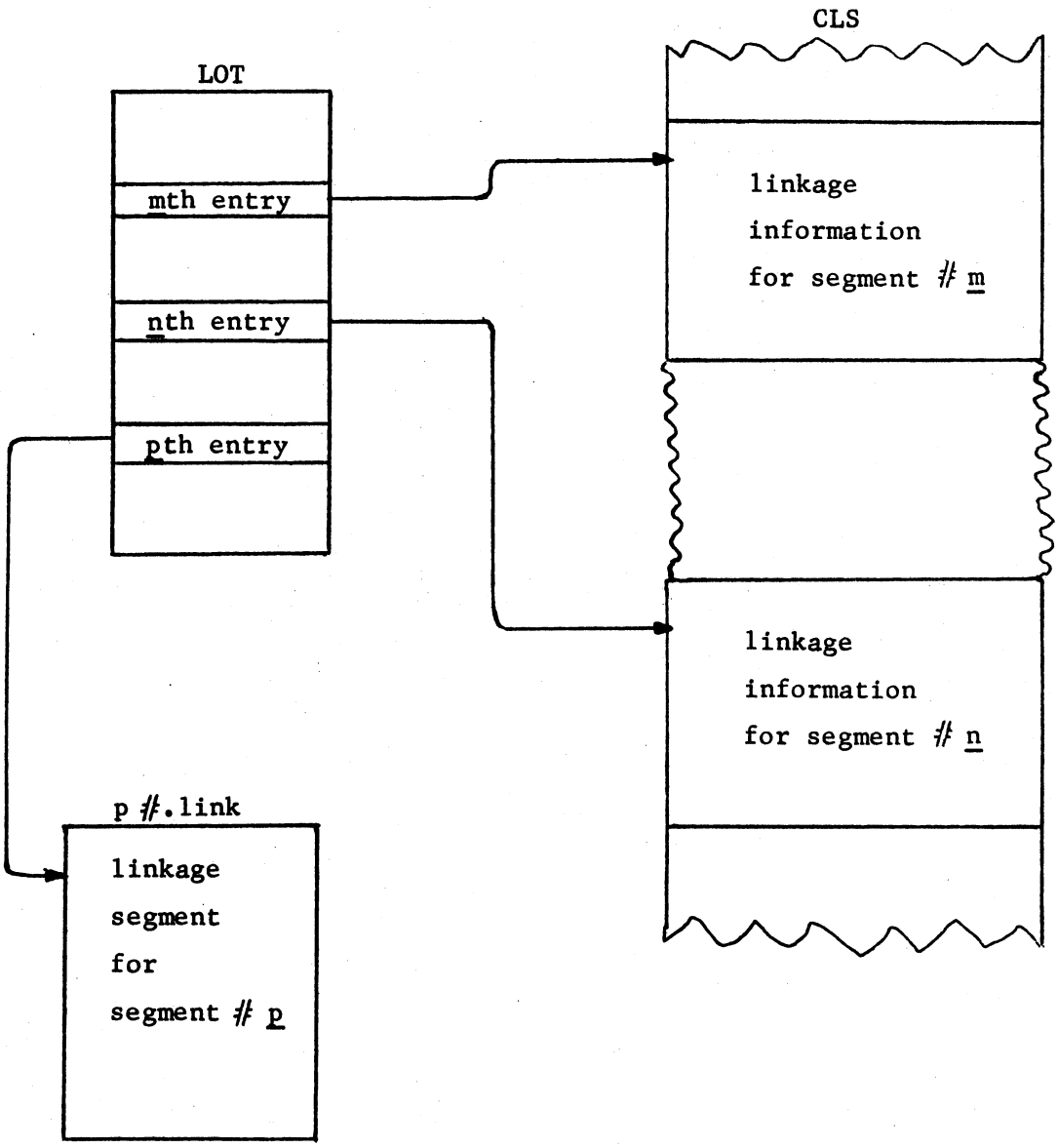

Figure 1. Role of the Linkage Offset Table

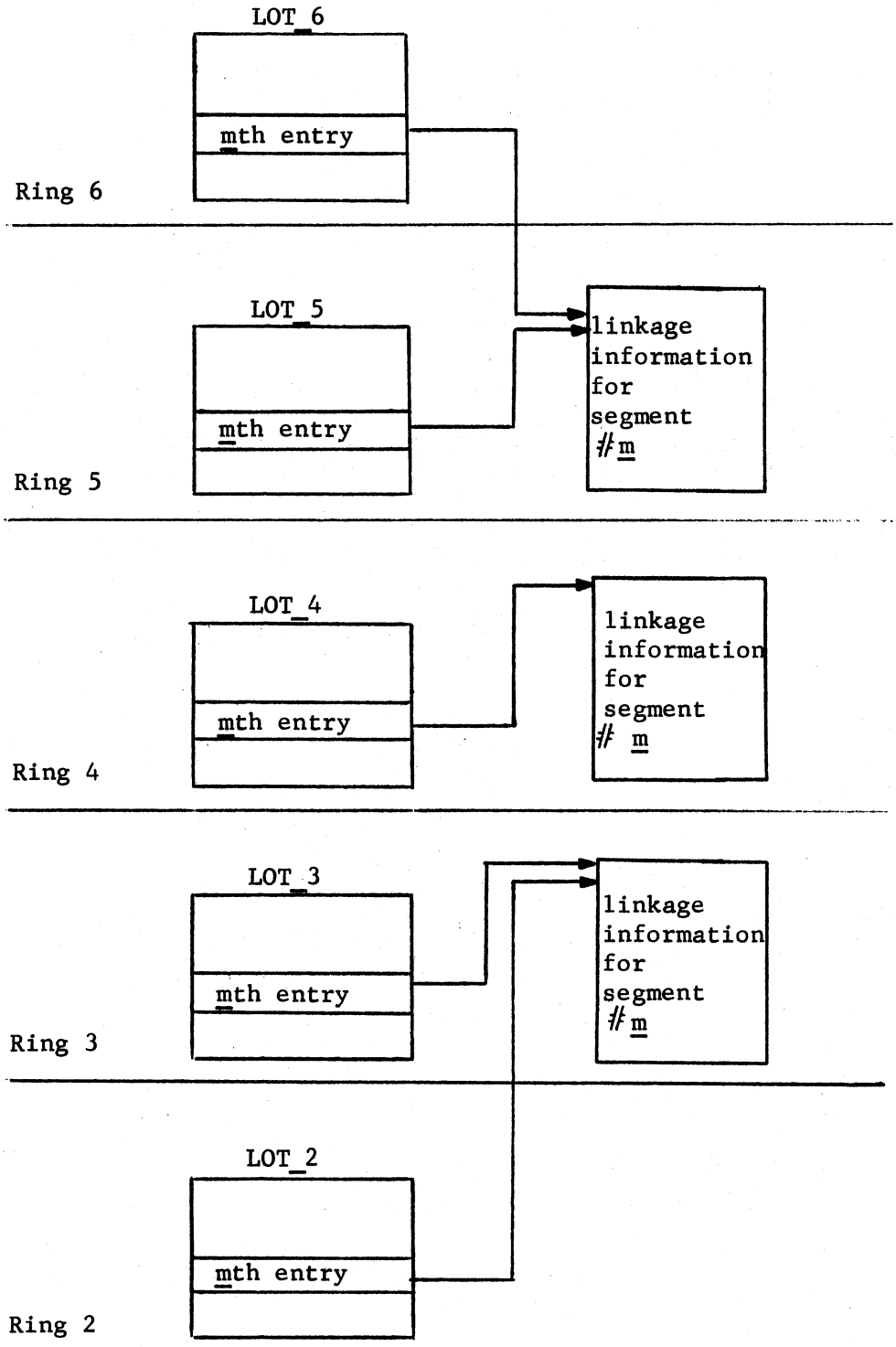

Figure 2. LOT's and Access Brackets

Segment number m is assumed to have an access bracket of 3:5

Usage of the LOT Maintainer

To cause a linkage segment to be added to a combined linkage segment:

```
call lot_maintainer$copy_linkage (clsptr,lsptr,lp,ercode);
```

with declarations

```
dcl (clsptr,lsptr,lp) ptr, ercode fixed bin (17);
```

where

clsptr — is a pointer to the combined linkage segment (input argument).

lsptr — is a pointer to a fresh copy (i.e., the links are "unsnapped") of the linkage segment to be added (input argument).

lp — is a pointer to the origin of the added linkage information within the combined linkage segment; that is, the segment number is the number of clsptr, but the offset is the offset of where the data pointed to by lsptr were added (output argument).

ercode — is an error code, which will be non-zero if the new data could not be added to the combined linkage segment (output argument).

To update an LOT and adjust the definitions pointer (see also BD.7.01) of the newly-added linkage information:

```
call lot_maintainer$set_lp (lotptr, textptr, lp, ercode);
```

with declarations

```
dcl (lotptr, textptr, lp) ptr, ercode fixed bin (17);
```

where

lotptr — is a pointer to the LOT to be updated (input argument).

textptr — is a pointer to the text segment whose linkage segment has been combined (input argument).

lp — is a pointer to the new origin of the linkage information for the segment pointed to by textptr; that is, it is the lp gotten from a call to lot_maintainer$copy_linkage (input argument).

ercode — is an error code, which indicates that lp does not point to a valid linkage section, if non-zero (output argument).

On return from lotm$get_lp, the entry (in the LOT pointed to by lotptr) corresponding to the segment number of textptr will be lp, and the "definitions pointer" (at lp+0) will have been processed as follows: If the definitions pointer indicated that the linkage definitions were in the linkage segment, the segment number of lp is placed into the definitions pointer, and any offset in lp is added to the offset portion of the definitions pointer (note that this tactic covers both the case where lp points to a Combined Linkage Segment - in which case lp will contain a meaningful offset - and the case where lp points to a non-combined linkage segment - in which case lp will contain a zero offset). If the definitions pointer indicated that the linkage definitions were in the text segment, the segment number of textptr is placed into the definitions pointer, and the offset is not altered. Finally, if the definitions pointer had already been set (detectable by the presence of an ITS modifier at lp+0), it is not changed.

To retrieve a linkage pointer from a given LOT:

```
call lot_maintainer$get_lp (lotptr, textptr, lp, ercode);
```

with declarations

```
dcl (lotptr, textptr, lp) ptr, ercode fixed bin (17);
```

where the arguments have the same meanings as above, except lp is returned by the routine, and ercode, if non-zero, indicates that the linkage information for the segment pointed to by textptr is unknown.

Usage and Logic of the Linkage Manager

As mentioned above, the link_man module serves the Linker as the source of all linkage pointers, and causes linkage segments to be combined when appropriate. Two additional entry points exist, to serve as an interface with the LOT Maintainer for such procedures as datmk, which cannot allow the linkage segments of certain segments to be combined.

These latter procedures do not call lot_maintainer directly, because that would necessitate their being cognizant of such information as the locations of the LOT and combined linkage segment of the current ring - information which link_man has available. There is also an initialization entry point.

The secondary calls are:

```
call link_man$set_lp (textptr, lp);

call link_man$get_lp (textptr, lp);
```

with declarations and interpretations identical to those of the corresponding entry points to lot_maintainer. The response to these calls by link_man is to obtain the appropriate lotptr (see below) and call the corresponding lot_maintainer entry.

The primary call is

```
call link_man$get_linkage (textptr, lp);
```

with declarations

```
dcl (textptr, lp) ptr;
```

where

textptr is a pointer to the text segment.

lp is the linkage pointer, returned by link_man.

The logic of link_man$get_linkage is as follows:

1. Get the validation level of the procedure which took the linkage fault, via a call to level$get (BY.12.01). The validation level is the number of the protection ring in behalf of which the Linker is operating; call it ring.

2. Get a pointer to the LOT for ring. These pointers are stored in an array in link_man's internal static storage. If there is no entry in the ringth position in the array, create an LOT and enter a pointer to it in the array; also, create a segment to be ring's combined linkage segment and enter a pointer to it in the array of such pointers (in link_man's internal static).

3. Determine whether the linkage segment in question is already in the current process via a call to lot_maintainer$get_lp. If ercode is zero (indicating lp found), return the lp gotten to the caller. If ercode is non-zero (indicating failure to find lp), proceed.

4. Get a pointer to the desired linkage segment as follows: call the Segment Management Module's get_segment primitive (BD.3.03) with textptr and ".link" as arguments; this will obtain a pointer to the original version of the linkage segment associated with the text segment in question.

5. If the definitions pointer in the just-acquired linkage segment has already been set, this implies that the segment should not be combined. Therefore, it is merely necessary to call lot_maintainer$set_lp for the pointer received from the call to initiate and then return this pointer to the caller. Otherwise (definitions pointer not set), proceed.

6. Next, protection ring considerations must be dealt with. Call status (BG.8.02); an access bracket, and possibly a call bracket, will be returned (along with other, irrelevant information).

   a) The following rule applies as to which ring the subsequent copying or combining (see b.) will be performed in: call the access bracket low: high; call the target ring target_ring; then if ring is less than low, target_ring equals low; if ring is greater than high, target_ring equals high; otherwise target_ring equals ring. If target_ring does not equal ring after applying the rule, call lot_maintainer$get_lp, this time for target_ring's LOT. If a linkage pointer is found by this call, record it in ring's LOT and return it to caller; otherwise, proceed.

   b) If a call bracket was indicated by the call to status, no combining is to be performed, as the presence of a call bracket implies that the linkage segment at hand belongs to a segment which has a gate. The following steps are taken: make a copy of the linkage segment "in" target_ring, with the same access and call brackets as reflected by status. ("In", here, means that a pointer to the copy is recorded in target_ring's LOT by a call to lot_maintainer$set_lp.) If ring does not equal target_ring, record the pointer in ring's LOT by another call to lot_maintainer$set_lp. Return the pointer to caller.

7. At this point in the logic, it is known that the linkage segment in question should be added to the combined linkage segment of target_ring. Therefore, call lot_maintainer$copy_linkage; then call lot_maintainer$set_lp for target_ring's LOT and, if necessary, for ring's LOT. Finally, call terminate for the linkage segment gotten by the SMM (in step 4), and return to caller the pointer to that portion of the combined linkage segment which contains the desired linkage information (gotten in the call to lot_maintainer$copy_linkage).

Finally, to initialize link_man, procedure init_admin in process initialization calls link_man$init. In response to this call, link_man obtains pointers to the LOT and combined linkage segment for ring 1 (which were created during process initialization) from the process directory, and stores them in the appropriate locations in link_man's internal static arrays of such pointers.