Published: 11/13/67

Identification

pre_linker_driver
R. L. Rappaport

Purpose

The major task involved in process initialization is the prelinking of a linker in the new process' address space. The pre_linker_driver making use of its database, the pre_linker driving table (see Section BJ.8.03), directs this pre_linking by calling upon appropriate subroutines to accomplish the individual tasks.

Introduction

The pre_linker_driver is basically a table driven program which is independent of the particular segments which are to be pre-linked. The pre_linker driving table lists the segments to be pre_linked and the pre_linking is accomplished in the following general way. The linkage sections, among the segments listed in the table, are searched to find potential linkage (fault tag 2) faults. When a fault is encountered the symbolic segment or call name to which the fault word points is obtained. A table lookup is then performed to see if this name is the call name of one of the segments listed in the pre_linker driving table. If it is, then the fault word is replaced by a pointer to the appropriate segment, and the fault will never be encountered dynamically. If the name is not found in the table, the fault word is left as is. To summarize the above, all intersegment references between segments listed in the pre_linker driving table are serviced while those references to segments not listed are ignored.

Discussion

The pre-linker driving table consists of two segments: <pre_linker_dt> and <pre_linker_nametable>. Pre_linker_dt is a table which contains a fixed length entry for each listed segment while pre_linker_nametable contains the variable length character strings which name the individual segments. The fixed length entries contain relative pointers to the appropriate character strings in the nametable.

The pre-linker driver makes three passes over its driving table in accomplishing its task. The first pass is made to establish (or map) each of the listed segments into the new address space. That is, pre_linker_driver calls

file system primitive estblseg (see BG.8.04) for each segment listed in the table. The arguments for estblseg (i.e. the pathname of the segment) are obtained from the nametable and estblseg returns a pointer to the segment just established. This pointer is stored in the fixed length entry of the driving table for later reference.

The second pass over the table is made to in order to "pre-load" each appropriately flagged linkage section among the segments listed. Pre-loading a linkage section means establishing the pointer, in the linkage section, to the definitions block (see BD.7.01) and storing the segment number of the text segment in the linkage section. Not every linkage section in the list will be pre-loaded. The reason for this is that several of these segments will be shared linkage segments which were pre-linked at system initialization time and have since been made "read only". They are only included in the list because other segments on the list refer to them and they are needed to provide information so that others may link to them. For example, hcs_.link is such a segment. It is included on the list only because other segments (notably the segment management module) refer to it. At pre-linking time, we will not attempt to pre-link hcs_.link. However, we need it in order to pre-link segment management. To summarize the above, the second pass over the table is made to find all appropriately flagged linkage segments. These are segments whose entries indicate they are linkage sections (link_sw is equal to "1"b) and that they are to be pre-linked (pre_link_sw is equal to "1"b). For each such segment, subroutine pre_linker$load_link (see MSPM BL.7.02) is called. This routine is passed pointers to the linkage segment in question and its associated text segment. The routine pre-loads the linkage segment.

The question may arise as to why all linkage segments are pre-loaded before any pre-linking takes place. This is understood immediately once it is realized that <a> cannot be linked to <b> until <b.link> is loaded.

The third and final pass over the table is made in order to actually accomplish the pre-linking. In this pass we again only look for the flagged linkage segments. This time for each such segment, subroutine pre_linker$scan_link (see BL.7.02) is called. This routine searches the linkage section and calls subroutine pre_linker$force_link (see MSPM BL.7.02) for each fault word found. Force_link handles the individual faults, servicing them when the reference is to a listed segment. Upon return from pre_linker$scan_link, all appropriate faults in a particular linkage segment have been replaced by ITS pointers.

At this point, all pre-linking has been done. The pre_linker_driver then merely destroys its now useless driving table and returns to its caller.

Pre_linker_driver is called from init_proc (see BJ.9.01) and the calling sequence is:

```
call pre_linker_driver;
```

Figure 1 is a flow diagram for pre_linker_driver.

One point has been neglected until this point. The subroutines called by pre_linker_driver are the same ones used to accomplish pre-linking at system initialization time. Since the driving table at system initialization, the segment loading table (SLT, see BL.2.01), is different in format from the pre-linker driving table, these common subroutines have been made independent of the table format. This has been accomplished by providing table manager subroutines, for the respective tables, which present identical interfaces to the shared pre-linker subroutines. The pre-linker is passed the entry points of the appropriate table manager, as an argument. In particular, pre_linker$scan_link is passed the entry point of a procedure that manages the pre-linker driving table: pre_linker_driver$tabman. Scan_link makes no use of this argument directly. It instead passes this entry point on to subroutine force_link which goes about making the appropriate call. Pre_linker_driver$tabman is described below.

Pre_linker_driver$tabman

Entry point tabman in <pre_linker_driver> is designed to interface with subroutine pre_linker$force_link (see BL.7.02). Given a symbolic call name of a segment, tabman compares this call name to the call name of each segment named in the driving table. If a match is found, a pointer to the segment and a pointer to the associated linkage segment are returned to the caller. Also, the value of a status bit is set to 1. If a match is not found, the status bit is set to 0 and no further action is taken.

The calling sequence is:

```
call pre_linker_driver$tabman (call_name_ptr, text_ptr, link_ptr,
                               found_sw);
```

where:

call name ptr is a pointer to the call name desired.

text_ptr is the return argument into which we store a pointer to the segment, if found

link_ptr is the return argument into which the pointer to the linkage segment is stored, if found

found_sw is the status bit.

The PL/I declarations for these are:

```
declare (call_name_ptr, text_ptr, link_ptr) pointer,
        found_sw fixed binary (1);
```

Figure 2 is a flow diagram of pre_linker_driver$tabman.

call pre_linker_driver;

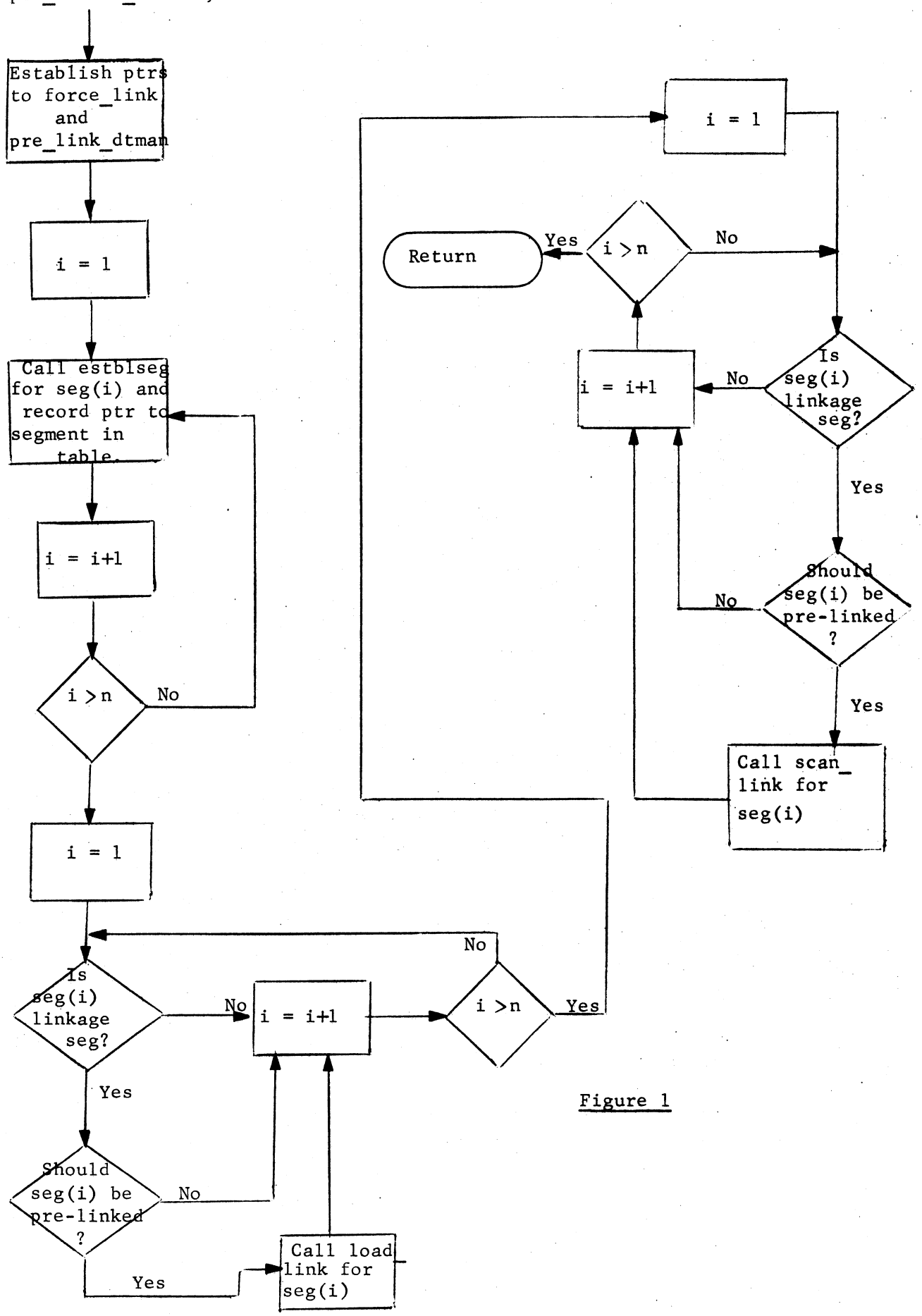

Figure 1

pre_linker_driver$tabman (call_name_ptr, text_ptr, link_ptr, found_sw)

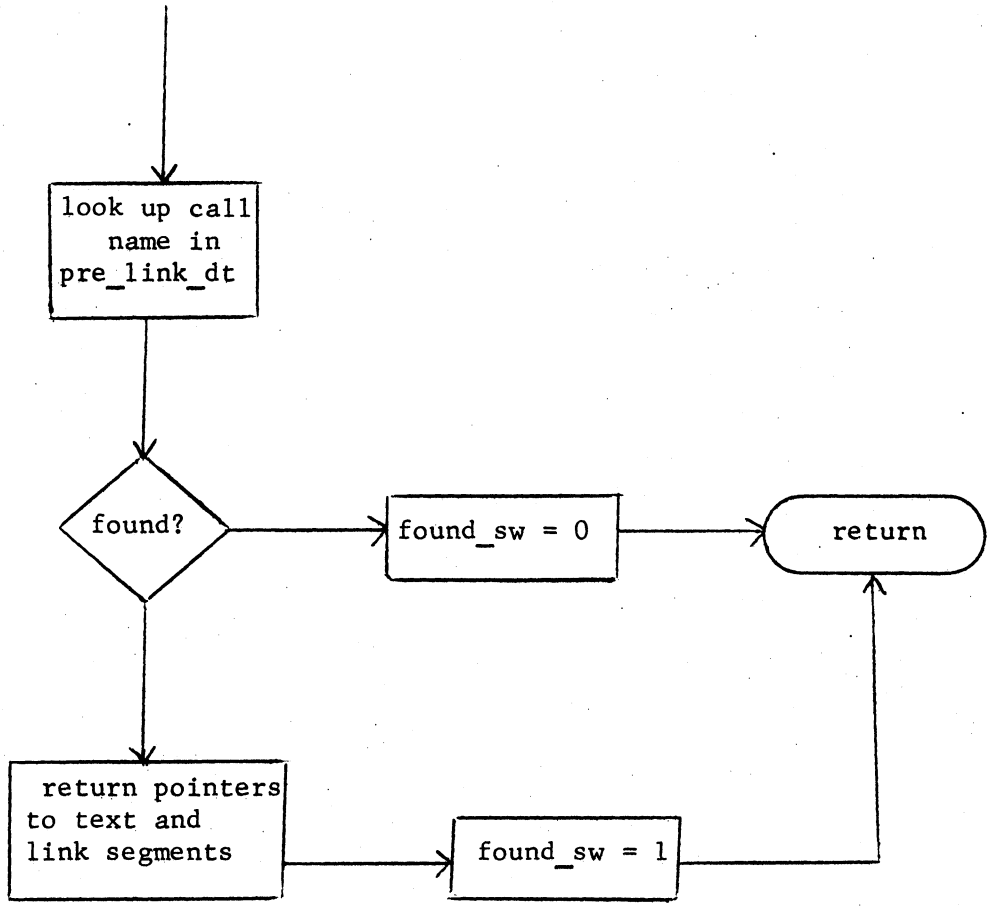

Figure 2