Published: 03/24/67
(Supersedes: BJ.5.01, 09/13/66)

Identification

Swap_dbr
R. L. Rappaport

Purpose

Swap_dbr is the procedure in which processes give up control of a processor.

Preface

The description of swap_dbr that follows is divided into three sections. The first section presents the basic outline of the subroutine. This would be an adequate description if it could be assumed that processes in the system are never unloaded and that execution of the subroutine will take place while:

1. The processor is completely masked against interrupts.
2. A global interlock is on, which denies access to the Process Exchange to all processes, except the one in which this subroutine is currently executing.

The second section presents the necessary additions to the basic outline that enable the unloading of processes to be accomplished The final section is a complete specification that describes the steps that must be taken to allow more than one process to be concurrently executing in the Process Exchange.

Basic Outline

Entry point swap_dbr, conceptually, is nothing more than an ldbr (load descriptor segment base register) instruction. However, at process-switching time many associated bookkeeping and housekeeping chores must be taken care of. These chores include accounting for processor usage, updating information in the Active Process Table, etc.

The call to swap_dbr originates in a process, the calling process, while the calling process is executing in subroutine getwork (see Section BJ.4.02). The principal argument passed to swap_dbr is a data item which indicates the

process to which control of the processor w ll be given, the target process. The actual data item passed in this call is the Active Process Table index of the target process. The complete calling sequence for swap-dbr is:

```
call swap_dbr (apt_index, error_return);
```

where apt_index is the Active Process Table index of the target process and error_return is the location of an error return in the calling procedure The error_return is not actually needed if no unloading of processes is allowed. Therefore this argument will not be mentioned again in the basic outline of swap_dbr. The stack used in this call is the Process Concealed Stack which is contained in the Process Data Segment.

As mentioned above, there are several housekeeping chores handled in swap_dbr. Before describing the steps taken in swap_dbr, it is important to understand the problems which these steps are meant to solve. These problems are enumerated and described below.

1. As swap_dbr is involved in process switching on the various processors in the system, this subroutine must assume part of the responsibility of accounting for processor usage. This accounting is handled by two steps taken in swap_dbr. At an appropriate point in the execution of swap_dbr, the Processor Metering Module is called and processor usage since the last call is metered to the account currently responsible. The account currently responsible is specified by a data item in the Processor Data Block (see Section BK.1.02). Swap_dbr then determines which account will be responsible for processor usage in the immediate future and a pointer to this account is stored in the Processor Data Block. The account to which processor usage should be charged while process A is running, is contained in process A's Active Process Table entry. Therefore, swap_dbr determines which account will be responsible by referring to the target's Active Process Table entry.

2. When swap_dbr is entered, the processor timer register contains the value stored by the calling process minus the number of memory cycles sed by this process. At some point in the procedure the timer register must be

reset to the value provided by the target's scheduler (see Section BJ.4.00) the last time the target was running.

3. The basic outline of swap_dbr assumes that all processor interrupts are masked during the execution of the subroutine. This means that any interrupts which occur will not be serviced until after swap_dbr has returned. Two types of processor interrupts exist: system interrupts and process interrupts. System interrupts are of interest to the processor itself and their servicing can be safely delayed. Process interrupts, however, are of interest to the process executing on the processor and delaying them until after swap_dbr returns means that the wrong process will be interrupted. Therefore, swap_dbr must insure that any process interrupts meant for the calling process, which are behind the processor interrupt mask, do not interfere with the target process. The strategy employed for removing these interrupts from behind the mask hinges on the nature of process interrupts. That is, all process interrupts ultimately cause the interrupted process to call swap_dbr. Since in this case the calling process is already executing in swap_dbr, any process interrupts directed towards the calling process can be safely ignored. The only way to remove interrupts from behind a mask is to allow them to occur (i.e., unmask them). Therefore, to "drain" a particular process interrupt, the particular interrupt handler is informed, by the setting of a switch, that draining is taking place, the interrupt is temporarily unmasked and then this switch is turned off again and the processor is remasked. If the interrupt was behind the mask, it would have been serviced while the processor was unmasked and the switch setting would have informed the handler that the interrupt is being drained. The switch used is the drain switch and it exists as a data item in the Processor Data Block. In swap_dbr two of the three process interrupts are drained: the timer runout interrupt and the quit interrupt. (The third process interrupt, the pre-emption interrupt, is previously drained in getwork.)

4. When an ldbr instruction is executed the address space seen by a processor changes while the other machine registers remain fixed. In particular, since the target process and the calling process need not be at the same stack level, base register sp must be reset after the ldbr is executed. Clearly then, before the ldbr, the calling process must store the current value of sp so

that it will be available the next time this process begins to run. After the ldbr the target must retrieve its stored value for sp and reload this register. The value of sp is stored in the respective Process Data Block of each process.

5. In order for a processor to handle faults and interrupts properly, the process executing on the processor must have access to the Processor Data Segment (see Section BK.1.01) that belongs to this processor. Swap_dbr assumes the responsibility for transferring the segment associated with a particular processor to the process that is due to run next on this processor. In order to pass along the Processor Data Segment to the next Process, the segment descriptor word for this segment is loaded into the A-register, by the calling process, immediately before the switch is made. Immediately after the switch (i.e., the ldbr instruction) the target process stores the A-register into its own descriptor segment at the appropriate relative location for the Processor Data Segment. (This step assumes all processes know the Processor Data Segment by the same segment number and that swap_dbr knows this number.)

6. When swap_dbr is entered the target process is still on the ready list and its execution state is defined as ready. That is, the ready switch in the target's Active Process Table entry is on. At an appropriate point swap_dbr removes the target from the ready list and redefines its execution state to running.

At this point all the basic issues faced by swap_dbr have been presented and it is possible to intelligibly specify the basic outline. Once the significance of the above problems have been understood, the actual steps taken by swap_dbr are seen to be fairly straight-forward. The steps of swap_dbr are described sequentially below and are also illustrated in figure 1.

One point should be kept in mind. Before the ldbr instruction, swap_dbr executes in the address space of the calling process and after the ldbr it executes in the address space of the target process.

1. Processor usage is accounted for. This is done by calling subroutine meter cpu (see Section BO.1.01) with two arguments: the Active Meter Table (see Section BO.1.07) index of the account currently being charged and the amount

of usage since the last call to meter_cpu. The amount of usage is determined by subtracting the current value of the processor timer register from the value of the timer register at the time of the last call. This last value is saved in the Processor Data Block. The Active Meter Table index of the account responsible is also stored in the Processor Data Block.

2. Timer runout interrupts are drained. That is, the drain switch in the Processor Data Block (of this Processor) is set on and a new processor interrupt mask is established. This new mask masks all interrupts except timer runout interrupts. The new mask remains in place for one instruction whereupon the old mask is restored and the drain switch is turned off. If a timer runout had been waiting behind the mask it would have been accepted and handled appropriately.

3. The timer register is reset with a new value and the Active Meter Table index of the account to which processor usage will be charged while the target process is running is stored into the Processor Data Block. Both of these quantities are obtained from the target process' Active Process Table entry.

4. Quit interrupts are drained. The procedure is similar to that involved with timer runouts except that the temporary mask used unmasks only quits rather than timer runouts.

5. The calling process stores the current value of base register sp into its Process Data Block.

6. The segment descriptor word of the Processor Data Segment for this processor is loaded into the A-register. It is obtained from the calling process' descriptor segment.

7. The ldbr instruction is executed. The operand of this instruction is the absolute address of the base of the target's descriptor segment.

8. The A-register is stored into the target's descriptor segment.

9. Base register sp is loaded with a new value. This value is obtained from the current (i.e., the target's) Process Data Block.

10. The target process redefines its execution state to the running state. That is, the ready switch is set off and the running switch is set on in the target's Active

Process Table entry.

11. The target process removes itself from the ready list.

12. Swap_dbr returns to its caller in the target process.

Additions to Enable Unloading of Processes

Certain modules in the hardcore supervisor perform functions whose execution cannot be interrupted by page faults. For example, all modules engaged in servicing page faults would be included in this category. This implies that all private data (e.g. stacks), belonging to a process, which might be referenced by one of these modules must be in core storage while the process is running. A process which is capable of running without causing such page faults is known as a loaded process and a process which is incapable of running without a minimum of preparation is known as an unloaded process. At any time swap_dbr may be called upon to switch control to an active (see Section BJ.1.00 for an accurate definition of this state), unloaded process. Swap_dbr assumes the responsibility of preparing active, unloaded processes so that they are able to run.

Briefly, a loaded process is an active process which has a hardcore ring descriptor segment and also has its Process Data Segment in core. The preparation swap_dbr goes through when called to switch to an unloaded process results in the unloaded process appearing to be loaded. This is done by giving the unloaded process a standard hardcore ring descriptor segment and an Interim Process Data Segment. These two segments allow the unloaded process to take control of the processor without jeopardizing critical system procedures. However once in control the process will recognize that it is unloaded and will not attempt to return from swap_dbr. It will instead call out to a procedure which will restore to core storage all data needed by the process in order to function properly. Once the process is completely restored it is then able to return from swap_dbr.

Specifically, swap_dbr, when called, determines whether or not the specified target process is unloaded. The Active Process Table entry for a process contains a switch, the "not loaded switch", whose value is a function of the state of the process. If the switch is on the process

is not loaded. If the target process is not loaded several extra steps must be taken by the calling process in swap_dbr before control is transferred and several extra steps must also be taken by the target process in swap_dbr after the switch is made.

First let us consider the additional steps taken by the calling process. The following steps can all be insterted between steps 2 and 3 of the basic outline. The first half of swap_dbr (i.e., before the ldbr) with these additions, is illustrated in figure 2. If it is determined that the target process is unloaded, entry point createseg, in Segment Control (see Section BG.3), is called twice to build both a hardcore ring descriptor segment and an Interim Process Data Segment for the unloaded process. Createseg if successful returns a parameter whose value is a segment descriptor word which points to the newly created segment. A segment created by createseg is wired down and can only be destroyed by an explicit call to entry point killseg in Segment Control. Createseg can fail to create a segment if there is a shortage of core space and in this case it performs an error return to swap_dbr.

If both calls to createseg are successful swap_dbr then proceeds to initialize the two segments by copying from template segments. That is, a template descriptor segment and a template Interim Process Data Segment (see Section BJ.5,06) are available from which swap_dbr merely copies. Swap_dbr then sets on the target's process loading switch, a data item in the target's Active Process Table entry, which defines the target process to be in the state between loaded and unloaded. Once this has been done the calling process has fulfilled its obligations to the unloaded process and swap_dbr now continues with the program outlined in the basic outline.

If either call to createseg fails swap_dbr must execute the proper error sequence. The error sequence has two objectives. The first is that a proper error return be gieven to getwork. The second is that processor usage expended in this futile loading effort be properly accessed. If the second call to createseg fails the now useless segment created by the first call must be destroyed by a call to killseg. Then the Active Meter Table index of a special "idle time" account is entered into the Processor Data Block of this processor. This account is only charged whenever swap_dbr is unsuccessful in a loading operation. Swap_dbr then performs an error return to getwork. If

the first call to createseg fails then all that needs be done is to enter the "idle time" account number in the Processor Data Block and perform the error return to getwork.

To tabulate the extra steps taken in swap_dbr by the calling process:

2.01. The target's not loaded switch is tested. If it is off (the target is loaded) go to step 3 of the basic outline.

2.02. Entry point createseg is called to create an empty wired down segment large enough for a hardcore ring descriptor segment. An error return from createseg goes to step 2.07.

2.03. Createseg is called again to create an empty wired down segment large enough to contain an Interim Process Data Segment. An error return from createseg goes to step 2.06.

2.04. The target's process loading switch is set on and the contents of the template descriptor segment and the template Interim Process Data Segment are copied into the two newly created segments.

2.05. Go to step 3.

2.06. Entry point killseg is called to destroy the segment created in step 2.02.

2.07. The Active Meter Table index of the special "idle time" account is assigned as the account to which current pr cessor usage should be charged.

2.08. Error ret rn to getwork.

Now let us consider the additional steps taken by the target process in swap_dbr to enable loading of processes. (These steps can be inserted between steps 11 and 12 of the basic outline. The second half of swap_dbr, with these add tions, is illustrated in figure 3.) These steps are only taken by the target if the target is unloaded.

Once the target process is in control and after it has taken care of the housekeeping tasks described in the basic outline, the target determines whether it is unloaded. If it is, then it must restore itself to its previous

loaded state before attempting to return to whatever it was doing the last time it was running. A process restores itself by loading itself (i.e., its Process Data Segment) and by recreating the hardcore descriptor segment that it had before it was unloaded. These two steps are handled in two distinct subroutines that are called from swap_dbr. Respectively they are, entry point pbm in the Process Bootstrap Module (see Section BJ.5.03) and entry point overlay (section BJ.5.05).

Specifically the entire procedure is as follows. If the target process finds its not loaded switch on it turns the switch off and calls entry point pbm in the Process Bootstrap Module. The Process Bootstrap Module basically restores the process' Process Data Segment to wired down core. (This module also restores to wired down core all non-standard versions of hardcore supervisor segments that this process uses. However this added complexity can be ignored by the reader since the Process Data Segment is an example of a special hardcore supervisor segment that this process uses. Everything done for the Process Data Segment is repeated for each other special segment and therefore the entire mechanism is revealed. Normal y a process uses no other special segments. See Section BJ.5.03 for a complete description of the Process Bootstrap Module.) One point should be noted when discussing the retrieval of the Pro ess Data Segment. Hardware and software restrictions in Multics impose a constraint on all processes executing on a processor in the System. This restriction is that at a known (and constant from process to process) segment number in th address space of the process, there exists a wired down stack that is usable to store machine conditions at fault and interrupt time. In the loaded process this stack is the Process Concealed Stack contained in the Process Data Segment. That is, the descriptor segments of all loaded processes have in the same relative location a segment descriptor word that points to their Proce s Data Segment. For simplicity let us suppose that this segment descriptor word is the jth such descriptor word. (I.e., this segment is segment number j.) In the loading process this wired down segment is the Interim Process Data Segment which means that all loading processes access their Interim Process Data Segment as segment number j. Therefore in order for a loading process to restore its Process Data segment to core, while retaining its Interim Process Data Segment, it must retrieve the actual segment by giving it a segment number different than j. (Similarly in retrieving any other special segments, a

process must retrieve each by a segment number distinct from that of the segment which the special segment is meant to replace.) For simplicity let us say that the Process Data Segment is retrieved as segment k.

Let us rephrase the above and continue. If swap_dbr determines that the target is unloaded, it calls the Process Bootstrap Module to load the Process. Return from this module implies the process is completely loaded. That is, its Process Data Segment and any other special segments this process uses are in wired down core. However the segments have been assigned segment numbers different from those of the segments they are meant to replace. In the case of the Process Data Segment, it has been assigned segment number k, while it is meant to replace the Interim Process Data Segment, now known by segment number j.

In order to resolve this question of segment numbers, overlay is called. Basically all this procedure does is to overlay the segment descriptor words for all special segments into the locations in the descriptor segment currently occupied by the segment due to be replaced. In the case of the Process Data Segment, overlay will pick up the segment descriptor word located at location k in the descriptor segment and deposit it into location j, thus wiping the Interim Process Data Segment out of the address space of the process. In order to retain a handle on the Interim Process Data Segment, the segment descriptor word for the segment is saved before the call to overlay. On return from overlay, entry point killseg is called in order to destroy the now useless segment.

Overlay is called using the Process Concealed Stack, therefore before the call swap_dbr switches stacks from the Interim Process Concealed Stack. Once the stacks have been switched, swap_dbr stores the needed segment descriptor word in the Process Concealed Stack and then calls overlay. On return from killseg the entire operation is complete and the target can reset its loading switch.

Actually one small point has been neglected in the above. This point has to do with the interaction of subroutine quit (see Section BJ.3.03) and processes that are loading. Processes that are loading cannot be unloaded. This is because these processes have data in segments (for example the Interim Process Data Segment) which cannot be paged out because they have no corresponding file in secondary storage. Quitting a process stops it from executing indefinitely. If a loading process were stopped indefinitely its unpageable

segments would remain in core indefinitely. Therefore it is necessary to delay quits meant for a loading process until the process is completely loaded. This is done by special considerations in subroutine quit and swap_dbr. Quit on being called to quit a loading process merely sets on the process' quit waiting switch, a data item in the process' Active Process Table entry. Swap_dbr for its part tests the quit waiting switch upon changing the target process' state from loading to loaded. If swap_dbr finds the quit waiting switch on it turns it off and itself calls quit for the target process. That is the target process calls to quit itself.

Regrouping and tabulating the entire extra sequence contained in the second half of swap_dbr:

11.01 The target's not loaded switch is tested. If it is off go to step 12.

11.02 Turn off the target's not loaded switch.

11.03 Call entry point pbm in the Process Bootstrap Module.

11.04 Switch stacks so that the Process Concealed Stack is being used.

11.05 Store the segment descriptor word of the Interim Process Data Segment into the current stack.

11.06 Call overlay.

11.07 Call killseg passing the stored segment descriptor word as an argument.

11.08 Turn the target's loading switch off.

11.09 The target's quit waiting switch is tested. If it is off go to step 12.

11.10 Turn off the target's quit waiting switch.

11.11 Call subroutine quit passing as an argument the target's Active Process Table index. (i.e., the target calls quit for itself.)

Complete Specification of Swap-dbr

With several processes possibly executing in the Process Exchange concurrently, steps must be taken to coordinate their actions. In particular, two general steps have been taken. First, certain interlocks and switches have been established in the Process Exchange data bases. By observing common rules about the interlocks and switches, the various modules are able to guarantee the integrity of the data with which they deal. Secondly, at certain well defined points while some of these interlocks are set, the processor referencing the locked data must be masked against all interrupts. This is to prevent the possibility of putting a processor into an infinite loop. (For a complete discussion of coordination in the Process Exchange, see section BJ.6.)

In swap_dbr several such coordination actions appear. In particular, two data switches, not previously mentioned in this document, are reset in swap_dbr and two interlocks are encountered while swap_dbr goes about the tasks previously described. The two switches not mentioned are the calling process' intermediate-state switch and the target's chosen switch. Both of these data items are per process switches which reside in the respective Active Process Table entries. The two interlocks are the ready list lock, an interlock on the entire ready list, and the process state lock of the target process, a per process interlock which controls access to that process' Active Process Table entry.

Before proceeding, let us look at the four data items themselves. The intermediate-state switch of a process if on indicates that the process may be "executing" even though its Active Process Table entry indicates that the process is either ready or blocked. Because a process cannot redefine its execution state and stop executing instantaneously, such an intermediate state is unavoidable. The intermediate-state switch is used merely to indicate when this is the condition of the process. A process turns on its intermediate-state switch whenever it enters subroutines block or restart. The switch of the calling process in swap_dbr is turned off by the target almost immediately after the ldbr is executed. That is, the calling process' intermediate-state switch is turned off after it is out of this intermediate state.

The chosen switch of a process can only be on if the process is on the ready list. If on the switch indicates that the process has already been chosen to run and should

not be considered a suitable candidate for running. The switch of a process is set on in getwork when getwork chooses this process. This process then becomes the target process in swap_dbr and the switch is reset when the target removes itself from the ready list. This structure, of getwork choosing processes and swap_dbr removing them from the ready list, enables getwork to accept an error-return from swap_dbr without having to place the intended target process back in the ready list.

The ready list lock is an interlock which simply controls access to the entire ready list. The ready list may not be referenced by a process unless the process has already set the lock. The ready list lock, in the jargon of the Multics File System, is a loop lock. That is, a process trying to set the lock loops until the lock is setable. All interrupt handlers must be able to use the ready list and therefore all interrupts must be masked whenever the ready list is locked. This masking is necessary to prevent putting a processor into an infinite loop.

The fourth and final data item is the process state lock of a process. This per process interlock controls access to data items contained in the Active Process Table entry in which the interlock itself is located. In particular, swap_dbr uses three of the data items controlled by this interlock: the ready switch, the running switch, and the quit waiting switch. In general, a process may only look at the controlled data items belonging to itself or another process if it has already locked this interlock. This interlock is also a loop lock and again all interrupt handlers must be able to lock the process state lock of any process. Hence all interrupts are masked in swap_dbr whenever the target's process state lock is locked.

In the basic outline of swap_dbr it was assumed that all interrupts were masked throughout the entire execution of the routine. Now we only assume that all process interrupts are masked throughout the entire routine while more encompassing masks are used for limited portions. The reason for masking process interrupts is to prevent swap_dbr from being entered recursively. Recall that all process interrupts have as their end result the calling of swap_dbr by the interrupted process. Therefore, acceptance of a process interrupt in swap_dbr would indeed mean a recursive call to swap_dbr. In contrast, system interrupts need only be masked while critical data items are locked.

Finally we are in a position to completely specify swap_dbr. First we will describe the strategy of setting the two switches and then we will describe the interlocking scheme. All additions entailed by this specification occur in the second half of swap_dbr and therefore figure 2 is still accurate as an illustration of the first half of swap_dbr.

Immediately after the target loads base register sp (step 9 of the basic outline) the target turns off the intermediate-state switch of the calling process. The other switch, the target's chosen switch, is reset immediately after the target removes itself from the ready list (step 11 of the basic outline). The target's process state lock is encountered when the target redefines its state from ready to running and also when the target tests its quit-waiting switch. Finally, the ready list lock is encountered when the target removes itself from the ready list.

Swap_dbr is completely tabulated below. The level of qualification of the step number indicates when the step was added. That is, step number x is in the basic outline, step number x.xx was added to enable loading, and step number x.xx.xx was added in the final specification.

1. Processor usage is accounted for.

2. Timer runout interrupts are drained.

2.01 The target's not loaded switch is tested. If it is off (the target is loaded) go to step 3.

2.02 Entry point createseg is called to create an empty wired down segment large enough for a hardcore ring descriptor segment. An error return from createseg goes to step 2.07.

2.03 Createseg is called again to create an empty wired down segment large enough to contain an Interim Process Data Segment. An error return from createseg goes to step 2.06.

2.04 The target's process loading switch is set on and the contents of the template descriptor segment and the template Interim Process Data Segment are copied into the two newly created segments.

2.05 Go to step 3.

2.06 Entry point killseg is called to destroy the segment created in step 2.02.

2.07 The Active Meter Table index of the special overhead account is assigned as the account to which current processor usage should be charged.

2.08 Error return to getwork.

3. The timer register is reset with a new value and a new account number is established.

4. Quit interrupts are drained.

5. The calling process stores the current value of base register sp into its Process Data Block.

6. The segment descriptor word of the Processor Data Segment for this processor is loaded into the A-register. It is obtained from the calling process' descriptor segment.

7. The ldbr instruction is executed. The operand of this instruction is the absolute address of the base of the target's descriptor segment.

8. The A-register is stored into the target's descriptor segment.

9. Base register sp is loaded with a new value. This value is obtained from the current (i.e., the target's) Process Data Block.

9.00.01 Reset calling process' intermediate-state switch.

9.00.02 The present processor mask is saved and the processor is masked against all interrupts.

9.00.03 The target's process state lock is locked.

10. The target's ready switch is set <u>off</u> and its running switch is set <u>on</u>.

10.00.01 The target's process state lock is unlocked.

10.00.02 The ready list is locked.

11. The target removes itself from the ready list.

11.00.01 The ready list is unlocked.

11.00.02 The previous processor mask is restored.

11.00.03 The target's chosen switch is turned off.

11.01 The target's not loaded switch is tested. If it is off go to step 12.

11.02 Turn off the target's not loaded switch.

11.03 Call entry point pbm in the Process Bootstrap Module.

11.04 Switch stacks so that the Process Concealed Stack is being used.

11.05 Store the segment descriptor word of the Interim Process Data Segment into the current stack.

11.06 Call overlay.

11.07 Call killseg passing the stored segment descriptor word as an argument.

11.08 Turn the target's loading switch off.

11.08.01 Save the present mask and mask all interrupts.

11.08.02 The target's process state lock is locked.

11.09 The target's quit waiting switch is tested. If it is off unlock the target's process state lock and restore the previous mask and go to step 12.

11.10 Turn off the target's quit-waiting switch.

11.10.01 Unlock the target's process state lock.

11.10.02 Restore the previous mask.

11.11 Call subroutine quit passing as an argument the target's Active Process Table index. (i.e., the target calls quit for itself.)

12. Swap_dbr returns to its caller.

Wrapup

One last thing should be noted at this point. The ldbr instruction may only be executed in master mode. In order to isolate this from the rest of swap_dbr, the actual instruction is contained in a distinct master mode segment. Actually, steps 5 through 9 are all contained in the master mode routine, ldbr1 (see section BJ.5.04), since these steps must be executed while the processor is inhibited. In this way, the rest of swap_dbr can be executed in slave mode. Figures 4a and 4b provide a complete illustration of swap_dbr.

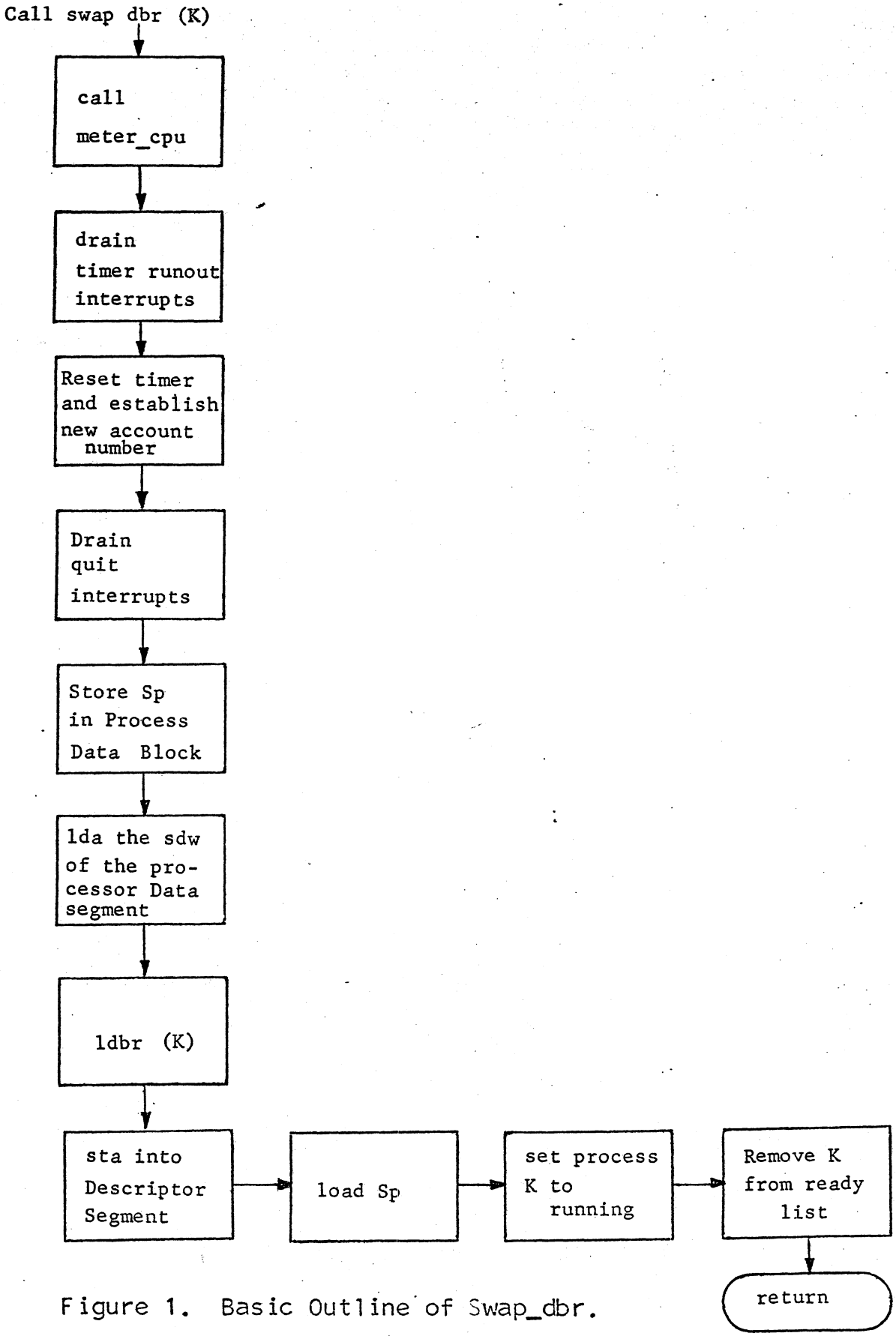

Figure 1. Basic Outline of Swap_dbr.

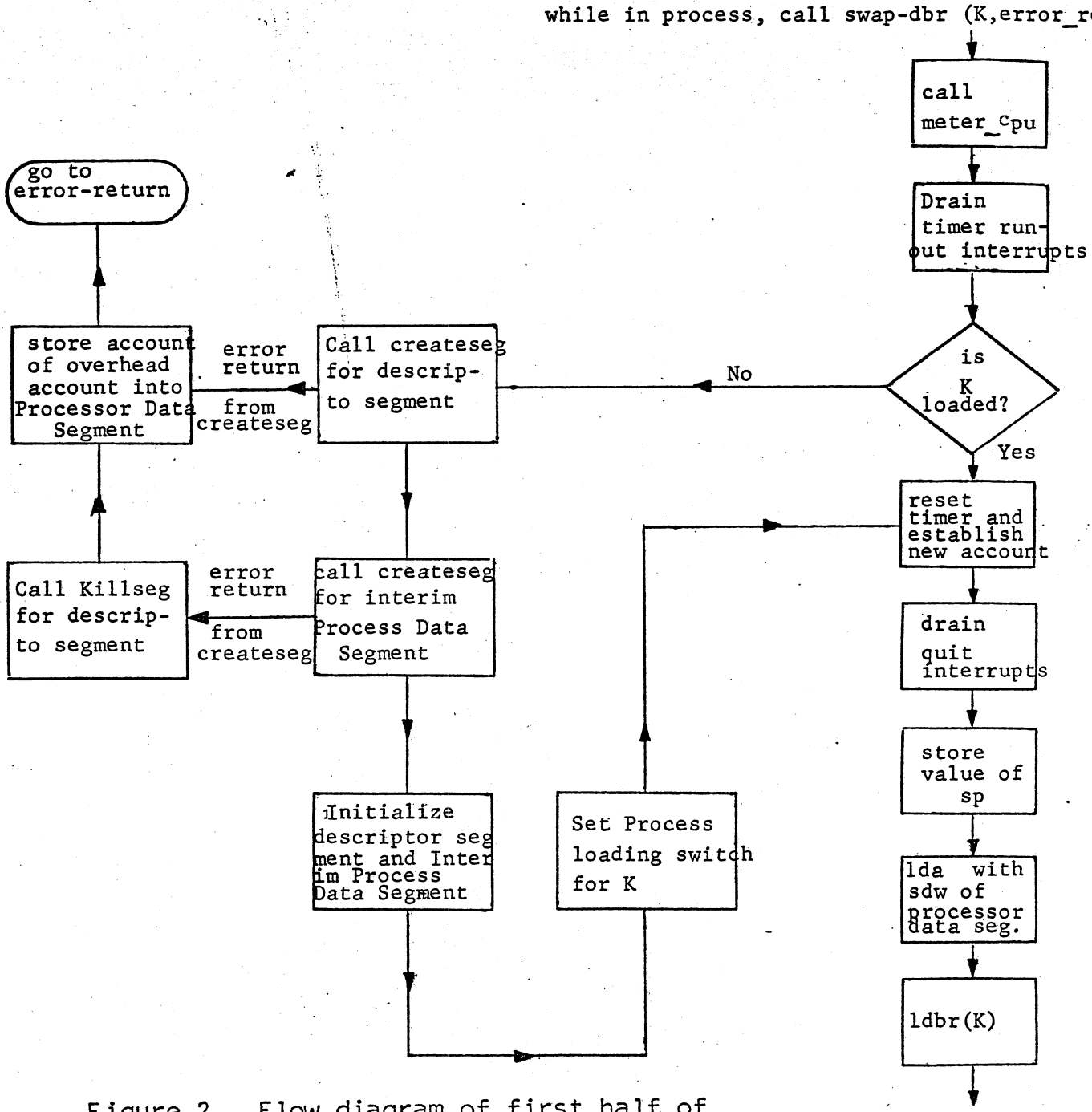

Figure 2. Flow diagram of first half of swap_dbr with additions to enable loading of processes.

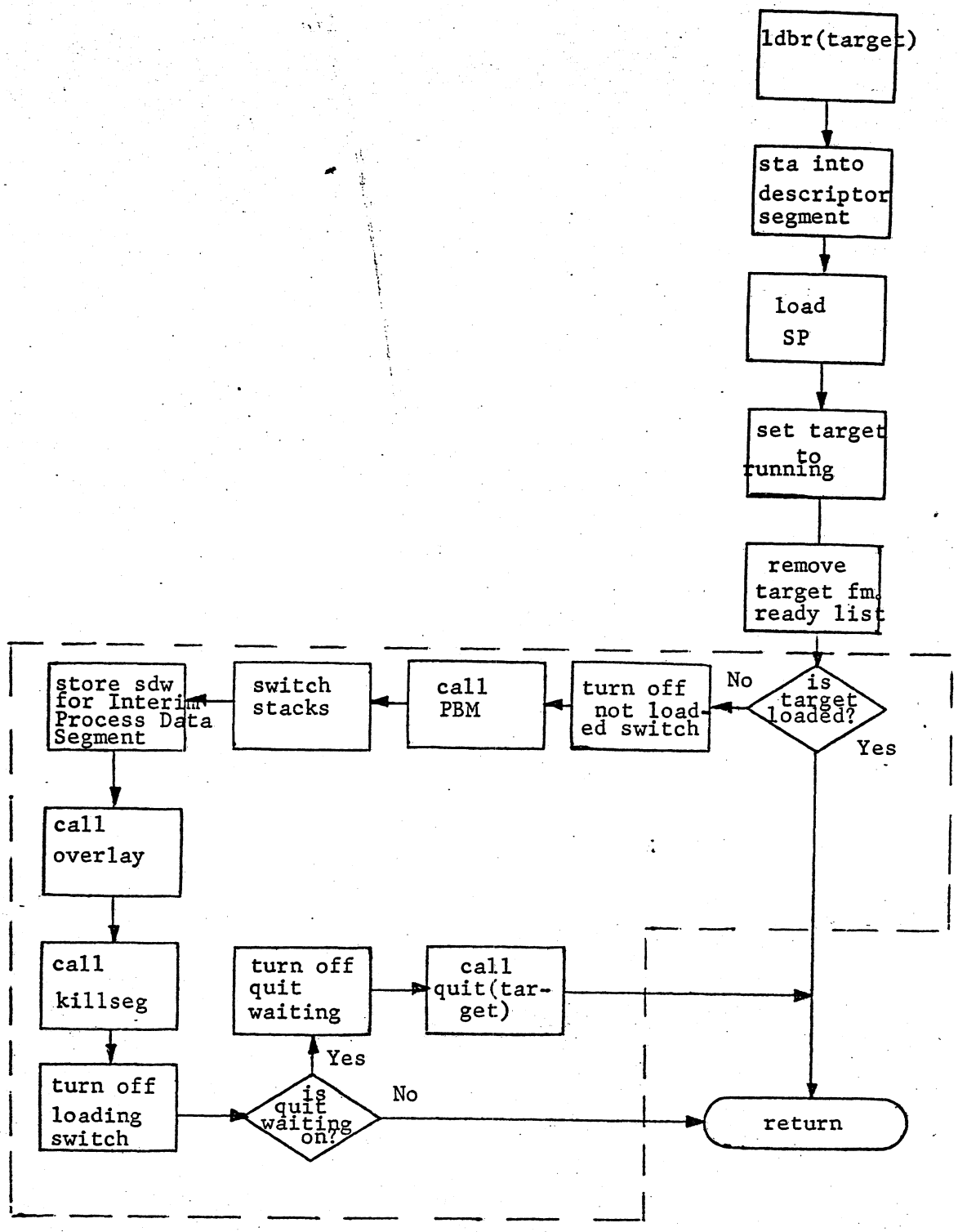

Figure 3. Second half of swap_dbr with additions to enable loading of processes. The aditions to the basic outline are enclosed in the dashed line.

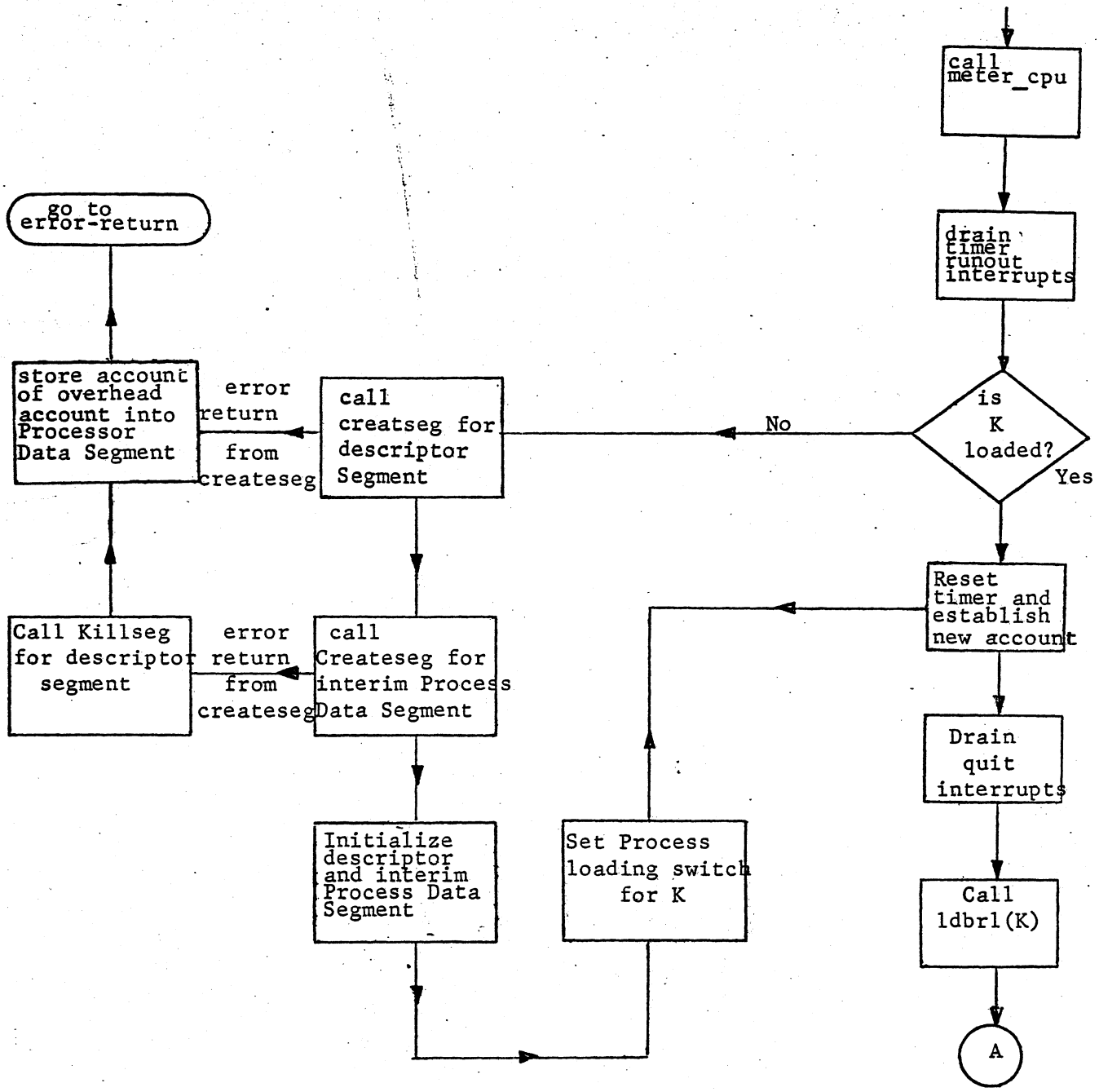

Figure 4a. Complete specification of first half of swap_dbr.

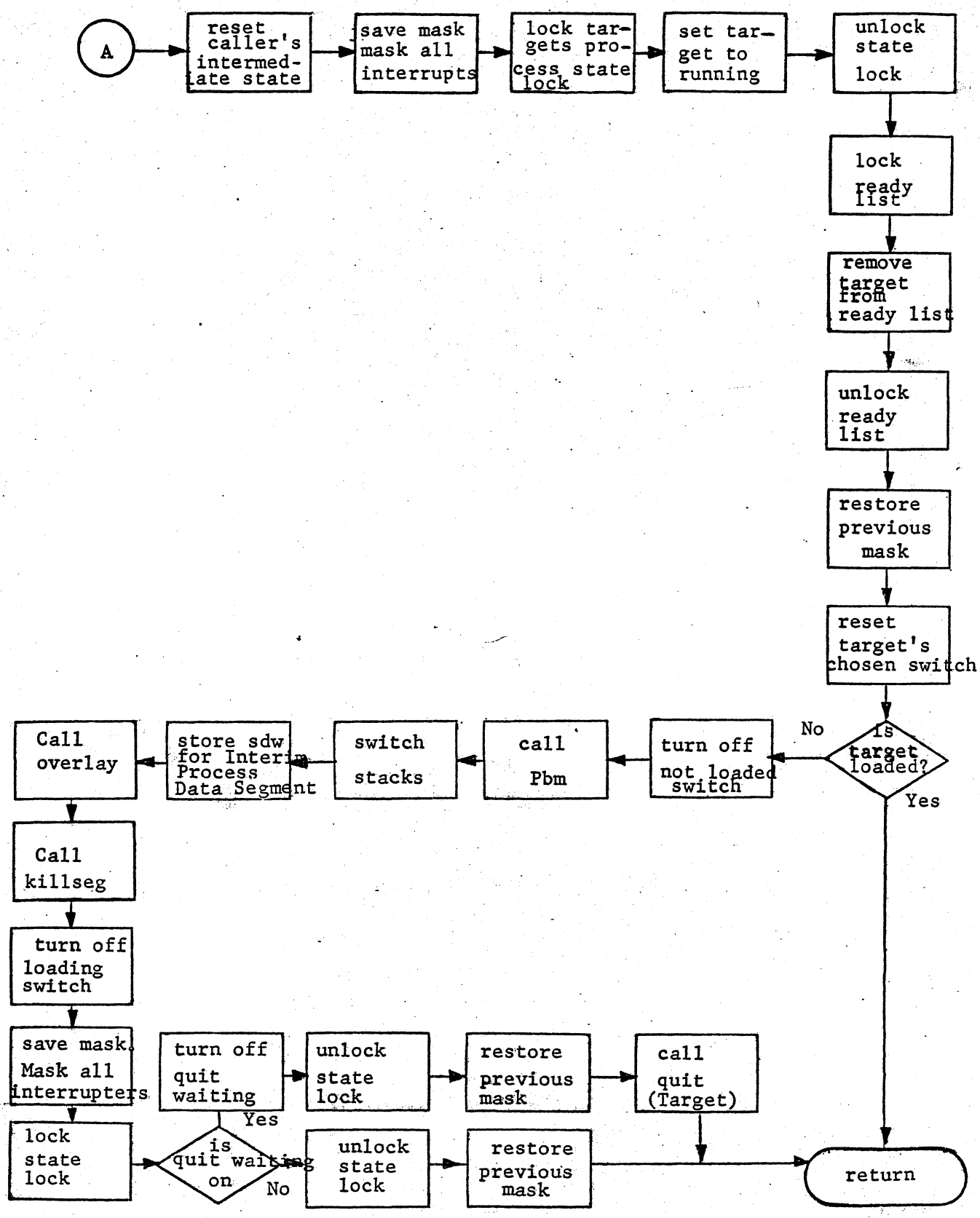

Figure 4b. Complete specification of second half of swap_dbr.