Published: 02/06/67

Introduction

Getwork
R. L. Rappaport

Purpose

Getwork, executing in the address space of a process giving up control of a processor, chooses the process to which control of this processor will be given. (This section assumes that the reader is familiar with BJ.3.01-BJ.3.04).

Preface

The description of getwork 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

A process that is about to stop running calls subroutine getwork in the Process Exchange, in order to choose the process to which the processor will be given. Basically, getwork does nothing more than the following:

1. It chooses the highest priority process on the ready list (see Section BJ.4.01).
2. It removes the calling process from the running list and inserts the chosen process into the vacated entry.

3. It "drains" pre-emption interrupts. In order to understand this step, one must be aware of the difference between system interrupts and process interrupts. This will be explained a little further on.

4. It calls swap_dbr (see Section BJ.5.01) on behalf of the chosen process.

5. When return is experienced from swap_dbr, getwork returns to its caller.

It should be noted that the return from swap_dbr is not experienced until well in the future when a third process (possibly one of the original two) has chosen this original process to run and called swap_dbr on its behalf.

Step number 3 above, states that pre-emption interrupts are "drained". Two types of processor interrupt exist in Multics: system interrupts and process interrupts. The pre-emption interrupt is a process interrupt. System interrupts are of interest to the processor itself whereas process interrupts are of interest to the process executing on the processor. Since a processor is masked against process interrupts (actually all interrupts for this basic outline) while executing in the Process Exchange, any process interrupts remaining behind the processor interrupt mask when control is switched to a new process, must not be allowed to interfere with the new process since they were not meant for the new process. In order to remove a process interrupt from behind the mask the interrupt must be allowed to occur (i.e., the processor must be partially unmasked temporarily). In order to guarantee that the interrupt causes no harm, the interrupt handler must be notified about the change in process. Therefore to "drain" a particular type of process interrupt, a switch is set on and the processor temporarily unmasks this particular interrupt. All process interrupt handlers routinely check this switch before acting and take appropriate action when they find it on. The switch is the drain switch and it exists as a data item in the Processor Data Segment. In getwork pre-emption interrupts are drained. The other two process interrupts, the timer runout interrupt and the quit interrupt, are drained in swap_dbr (BJ.5.01).

One serious problem arises in the outline above. The above sequence is meant to execute with the Process Exchange locked. If the ready list were empty, no other process would be able to help replenish it. Therefore if the ready list is empty, getwork unlocks the Process Exchange and waits for some process to appear on the ready list. When the Process Exchange is unlocked the complete mask is removed from the processor and a partial mask which masks process interrupts is substituted. This partial mask, which prevents quit interrupts, timer runout interrupts and pre-emption interrupts, prevents this process from calling the Process Exchange recursively. A process executing in getwork is already committed to giving up its processor and a recursive call into the Process Exchange would be meaningless.

The calling sequence for getwork is simply:

```
call getwork;
```

and the stack used is the calling process' Process Concealed Stack. Figure 1 illustrates the basic outline of getwork.

Additions to Enable Unloading of Processes

The existence of unloaded processes in the system means that the process chosen from the ready list may be unloaded. The problem of switching control to unloaded processes is faced directly by swap_dbr and only indirectly by getwork. That is swap_dbr does the work involved with the loading. Only when swap_dbr is unable to accomplish the loading (because of a shortage of core space) does getwork become involved. If swap_dbr is unable to perform the loading it performs an error-return to getwork. Getwork then must try again by once more choosing a new process from the ready list and calling swap_dbr on behalf of this new chosen process. Figure 2 illustrates getwork with the addition of an error entry.

Complete Specification of Getwork

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 throughout the Process Exchange. First, certain interlocks and switches have been placed in the Process Exchange data bases. By observing common rules about the interlocks the various modules are able to guarantee the integrity of the data with which they deal. Secondly, at certain times 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.). It should be noted that the Process Exchange global interlock is not one of those referred to in the above. The ones mentioned above take the place of the global interlock.

In order to fully understand the use of interlocking and masking in getwork one must first understand the coordination problems. Getwork faces three such coordination problems.

The first problem has to do with the use of the ready list and the running list. These data bases are used in several Process Exchange modules. In order to avoid errors, an interlock is necessary to control access to them. Since the running list is always used in conjunction with the ready list, one interlock on the ready list will suffice. This interlock limits access to one process at a time. Since all interrupt handlers must be able to use the ready list, all interrupts must be masked whenever the ready list is locked. That is, in getwork the ready list must be locked before it is used and the processor must be completely masked before the lock is set. The lock is reset as soon as reference to the ready list is complete and the previous processor mask is restored.

The second coordination problem has to do with the fact that a process appearing in the ready list may still be executing in the Process Exchange. If control of one processor were given to a process while it was still executing on another processor, disaster would follow. The disaster would be a direct result of the fact that both instances of the process would be using the same stack segment (process concealed stack). In order to resolve this problem an extra switch has been provided in the Active Process Table entry of each process. This switch, the intermediate-state switch, if on indicates that the process may be executing in the Process Exchange irrespective of its execution state as defined by other switches in the Active Process Table entry. Clearly getwork should not choose a process from the ready list if such a selection could possibly result in two instances of the same process on separate processors at the same time.

Such a situation could occur in only one way. It could occur if a process (process A) executing in getwork were to choose another process (process B) to run while process B's intermediate-state switch was on.

However process A could safely choose itself to run regardless of the setting of its intermediate-state switch. Therefore the strategy followed by getwork is clear. Getwork passes over a process whose intermediate-state switch is on unless the process executing in getwork and the process whose intermediate-state switch is on are identical.

The third coordination problem arises from the fact that getwork does not remove a process from the ready list once it has chosen the process to run. The removal is accomplished in swap_dbr when it is determined for sure that control will actually be switched to this process. In order to notify other processes that a particular process has been chosen, getwork sets on the process-chosen switch of the chosen process. This switch exists as a data item in the process' Active Process Table entry. If getwork experiences an error return from swap_dbr, all it need do is reset this particular process-chosen switch.

Getwork can now be completely specified. Referring to Figure 3, the steps are as follows:

1. The processor is masked against all interrupts.

2. The ready list is locked.

3. The highest priority process which has not already been chosen is chosen and we proceed with step 4. However, if the ready list is empty or if all processes on the ready list are already chosen we unlock the ready list, restore the previous mask, wait until the state of the ready list changes and then go back to step 1.

4. If the chosen process has its intermediate-state switch off or if the calling and the chosen process are identical, then the process chosen switch is set on. If the chosen process' intermediate-state switch is on, then the next process on the ready list is chosen and we test it.

5. The calling process removes itself and places the chosen process on the running list.

6. Pre-emption interrupts are drained.

7. The ready list is unlocked and the previous mask is restored.

8. Swap_dbr is called. An error-return goes to step 10.

9. Getwork returns to its caller.

The following steps are executed only if an error return is received from swap_dbr.

10. The process chosen switch for the originally chosen process is reset.

11. The processor is masked against all interrupts, and the ready list is locked.

12. The calling process puts itself back on the running list.

13. This step is exactly the same as step 3 except that, in order to prevent looping, care is taken to prevent choosing a process which swap_dbr has already failed to switch to. That is, the highest priority process which has not been chosen and which has not already caused swap_dbr to fail, is chosen, we then go to step 4. If the ready list is empty or if all processes on the ready list are already chosen or if we have failed on all ready, unchosen processes, we unlock the ready list, restore the previous mask, wait until the state of the ready list changes and then go back to step 1.

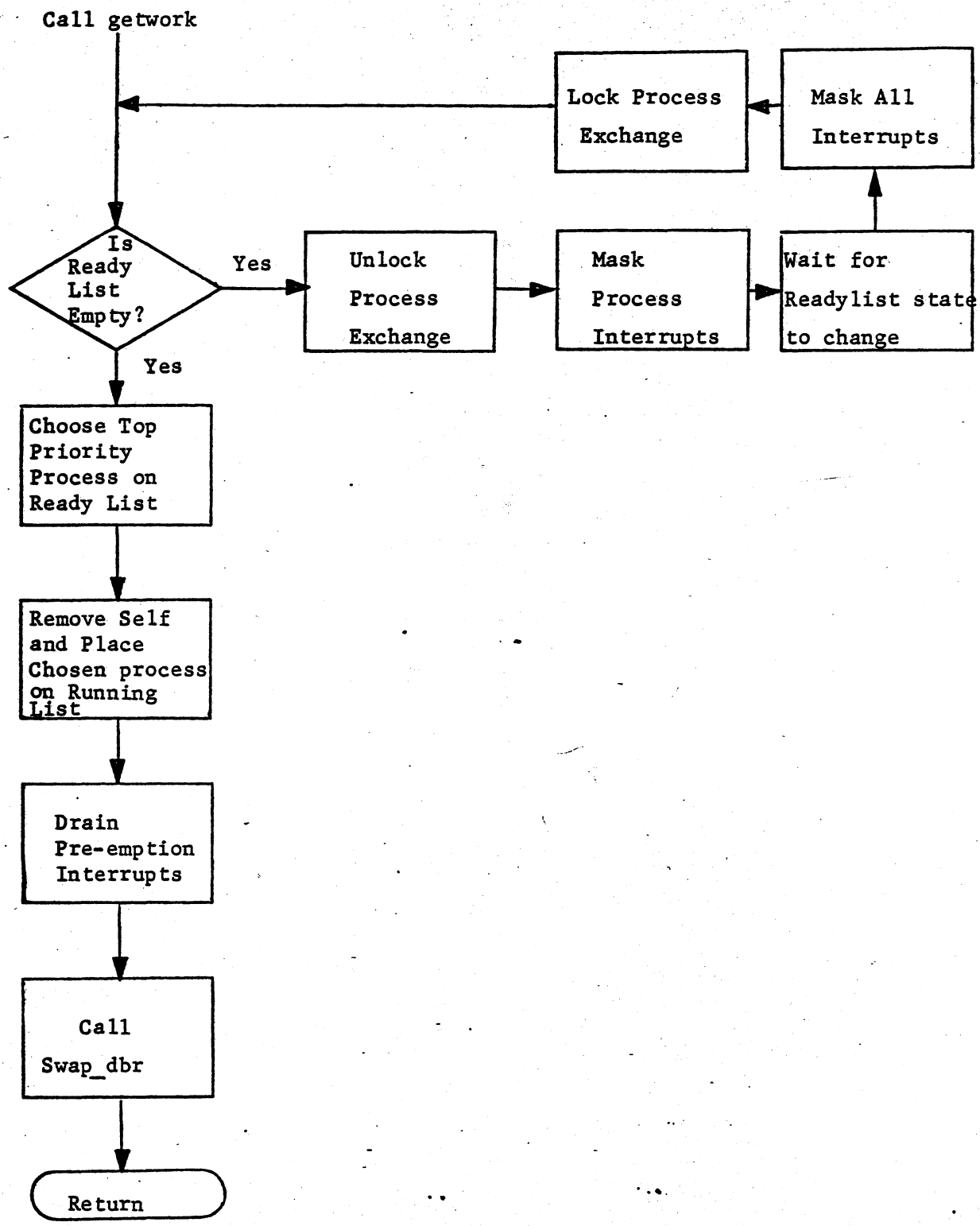

Figure 1. Basic Outline of Getwork

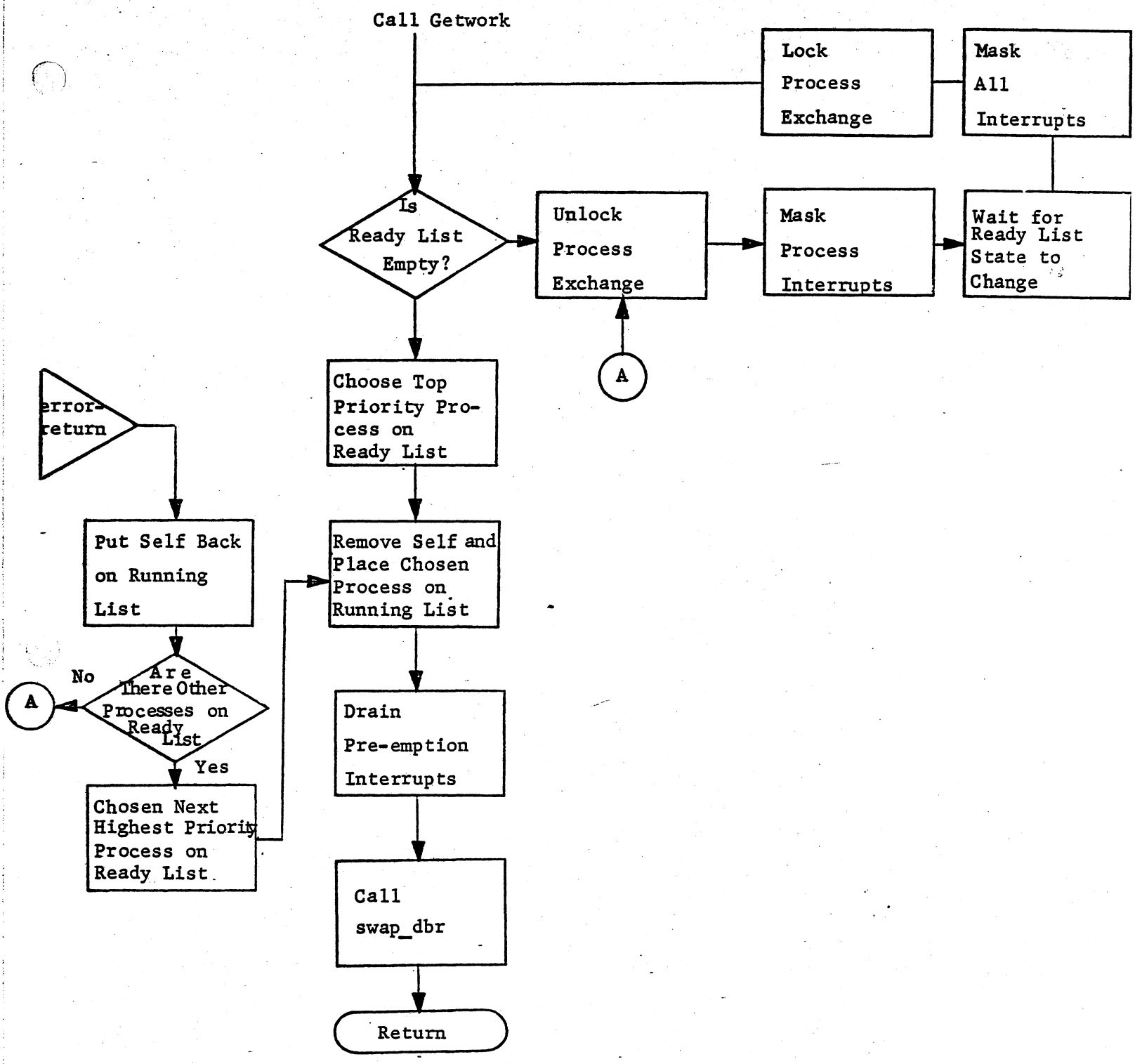

Figure 2. Basic Outline of Getwork with Additions to Enable Unloading of Processes

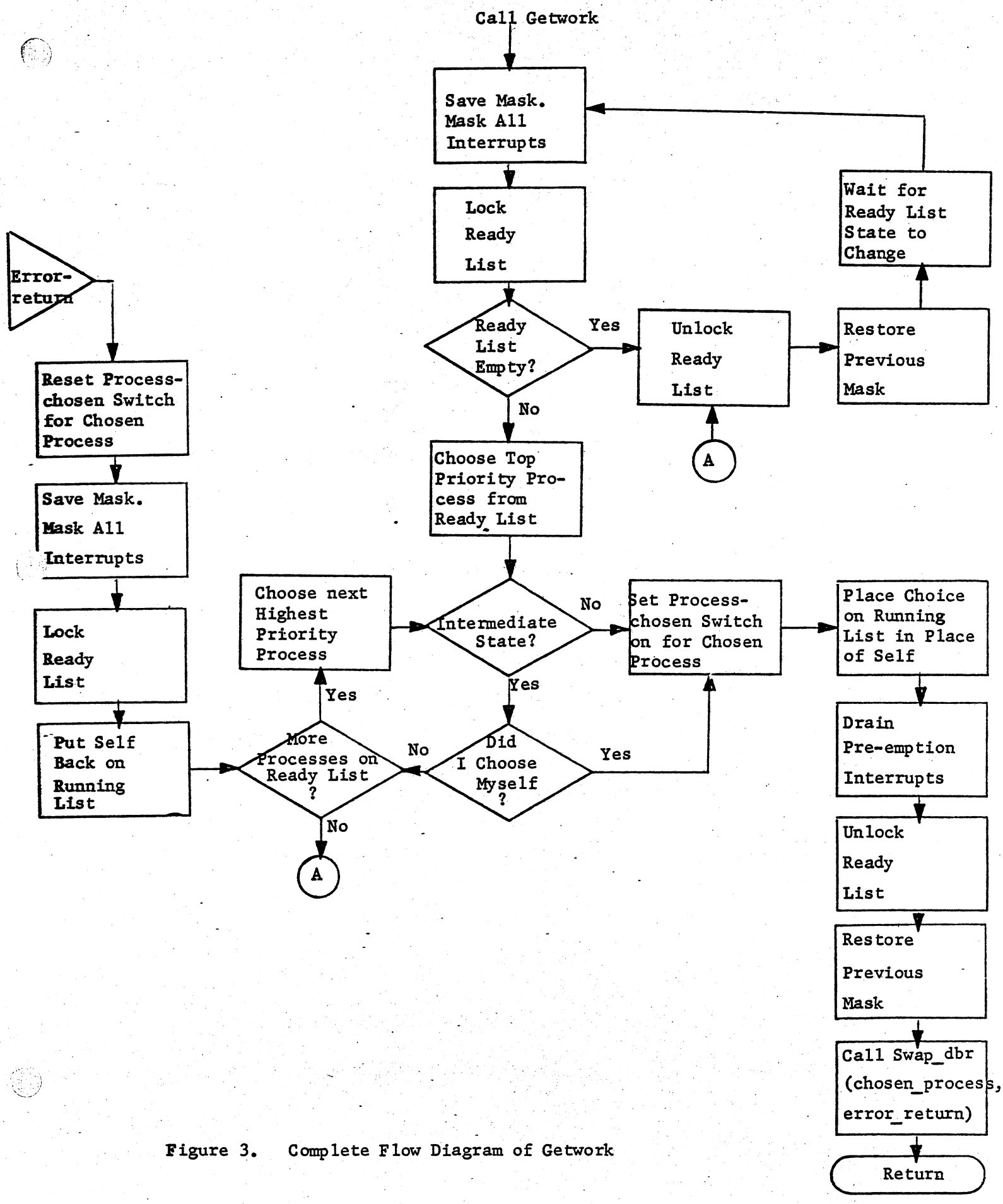

Figure 3. Complete Flow Diagram of Getwork