Published 12/766

Identification

Overview of the Process Exchange
J. H. Saltzer

Purpose

The "Process Exchange" is a group of supervisor modules concerned with dispatching and scheduling procedures among processes. The Process Exchange is driven entirely by closed subroutine calls from other supervisor modules, usually as a result of interrupts.

The Process Exchange is called for one of four different reasons:

1. A process cannot proceed until another process or hardware device sends a signal, and it wishes to give up the processor (block) for the interim.

2. A running process wishes to wake up another process which may be blocked.

3. A running process wishes to block another process which may be ready or running.

4. A running process may have just taken a particular system interrupt (timer runout or pre-emption) which implies that it must reschedule itself to continue to run later.

These four calls are made to four entry points named respectively, Block, Wakeup, Quit, and Restart. A process calling the Process Exchange will experience a return, although in the case of Block and Restart, the return will usually occur much later; other processes will have used the processor in the interim. Upon a return from the Process Exchange, the interval timer is running. As we shall see, the process may use the processor for whatever purpose it likes until either a process interrupt occurs or it cannot proceed pending a wakeup signal.

Process Switching: Swap-DBR.

The basic hardware mechanism by which a processor switches from one process to another is the "Load Descriptor Segment Base Register" (LDBR) instruction. At the instant the descriptor segment base register is reloaded, the address space (the traditional "core image") seen by the processor changes; the only explicit memory of the previous process and its address space remains in the processor registers.

In the Process Exchange, the LDBR instruction is embedded in a subroutine named Swap-DBR. This subroutine is called using a standard "Call" sequence with a standard stack. Swap-DBR begins with a standard Save sequence. However, in the course of its execution, it reloads the descriptor segment base register, thus switching to the address space of the process, and switches to the stack of the new process. If we assume that every process includes subroutine Swap-DBR in its descriptor segment, and that this is the only technique of switching between processes, then it is safe to perform a Return sequence with the new process' stack. The Return will return to the last place that this new process called Swap-DBR. The values of the descriptor segment base register and pointers to the stack are stored in an Active Process Table, which contains an entry for every active process. We note there are problems in switching to a "new" process which has never had an opportunity to call Swap-DBR. An equally difficult problem arises in switching control to a process whose descriptor segment has been paged out of core memory. (That is, it is no longer loaded.) Section BJ.5 provides a complete discussion of process switching and loading.

Processor Dispatching: Getwork and the Scheduler.

Swap-DBR is the only procedure in the system where process switching happens. It is used by the processor dispatching module, named Getwork.

Getwork is ultimately called whenever a process releases a processor for any reason; its function is to assign the processor a new task. For this purpose the Process Exchange maintains a "ready list," a list of all processes ready to run, in the order they are to be run. The ready list consists conceptually of pairs of entries; a process identification and a running time limit imposed by the scheduler. It is in fact implemented as a thread through the Active Process Table.

Let us suppose that Process K is running, and calls Getwork. Subroutine Getwork performs the following sequence:

1. Identify the highest priority process on the ready list (call it J).

2. Call Swap-DBR giving as an argument "J".

Swap-DBR will return to wherever it was called last in process J, namely to the copy of Getwork belonging to process J. If, at some later time process K should appear at the top of the ready list and some other process calls Getwork, control will appear in process K as a return

to Getwork from Swap-DBR. At this point, Getwork returns to its caller in process K.

The ready list is kept in order by a module named Schedule. The Process Exchange follows the rule that every process schedules itself, since the process knows factors which influence its own scheduling better than anyone else. This rule can allow some processes to use a different scheduler than others. Generally, then, the scheduler evaluates this process' request in comparison with the present status of the ready list, establishes a time limit for the process, and slips it into the ready list at an appropriate point. The time limit represents the maximum amount of processor resource that this process should be allowed to use before its priority is reconsidered by the scheduler. The time limit is enforced by a hardware device capable of generating a process interrupt, the "time-out interrupt."

A simple scheduler might follow these two rules:

1. Time limit = Q milliseconds, (Q constant)

2. Put process at end of ready list,

resulting in a "round-robin." A more elaborate scheduler, for example, a multi-level priority algorithm, can be plugged into this position instead. One option which is available to the scheduler is to force some processor to be relinquished by generating a process interrupt. A process interrupt generated by the scheduler will be called a "pre-emption interrupt." Section BJ.4 describes in detail the processor scheduling mechanisms and algorithms.

Block, Wakeup, Quit, and Restart entries.

The remainder of the Process Exchange consists of the four modules which accept the four types of calls, Block, Wakeup, Quit, and Restart. As an aid to visualization, figure one is a block diagram of the Process Exchange. In this figure, solid arrows are closed subroutine calls; dotted arrows are data paths. Restart is called by the Interrupt Interceptor when a timer runout or pre-emption interrupt occurs; the meaning of this interrupt is that the scheduler would like to have the processor back for someone else to use. The Restart module consists of two steps:

1. Call Schedule to reschedule this process to continue to run later.

2. Call Getwork to put the processor to work on the highest priority job available.

When the processor enters Swap-DBR, of course, it disappears from this process. Sometime later some processor reappears and performs the return from Swap-DBR.

The operation of the Wakeup and Block entry points will be more easily understood if we first review their intended use. When a process cannot proceed until some other computation is finished or some signal arrives, it first makes arrangements for one or more other processes to make a wakeup call to the Process Exchange; it then calls Block. When a process receives a return from Block, it can assume that some other process has called the Wakeup entry for it.

Stored in the Active Process Table entry for this process is the Wakeup Waiting Switch. The Wakeup Waiting switch is set on by another process whenever it sends a wakeup signal to this process. This switch is used to prevent a race to entry point Block after a process arranges for a wakeup. If the switch has turned on by the time the process gets to Block, Block will return immediately.

The Block module itself is quite simple both in purpose and in function. When a process calls Block, it has effectively committed suicide unless it first delegates some other process positive responsibility for a later Wakeup call, since the Block module immediately calls Getwork to give away the processor. (The difference between this action and the corresponding action in Restart is that in Restart the scheduler was called first to put the process back on the list of things to do.) Actually, before Block gives away the processor, it checks to see if the "Wakeup Waiting" switch is on. If so, it resets it and returns immediately to the caller.

Below we will see in detail how a call by another process to the Wakeup entry of the Process Exchange can cause our blocked process to be placed on the Ready List. For the moment, let us assume that this has happened and Getwork performs a return. This return to the Block module means that some event this process is waiting for has happened. The Block module therefore returns to its caller outside the Process Exchange. As before, the entry to Swap-DBR causes the processor to disappear; other events in the system (e.g., call to Wakeup, see below) can cause a processor to reappear in Swap-DBR and perform a return.

The process Wakeup entry to the Process Exchange is complex because the process being signaled may be ready, running, or blocked. The first step is to set the Wakeup Waiting switch on for the process being wakened, and check its execution state. If the process is not blocked, Wakeup

returns to its caller. The meaning of this condition is that the process being signaled has arranged for a wakeup, and the wakeup arrived before the process got to Block. Recall that if the process arrives at Block and finds its Wakeup Waiting Switch on, it will receive its wakeup immediately.

If the process is blocked, it must be allowed to schedule itself to run later. Wakeup therefore calls a special entry in Swap-DBR named "Ready-Him". This entry does the following:

1. Switch to the descriptor segment of the awakening process.

2. Call the scheduler to put the process on the Ready List.

3. Switch back to the descriptor segment of the calling process.

4. Return to the Wakeup module.

Note that, since the descriptor segment of the awakening process was used for the call to the scheduler, the awakening process' scheduler is used, not the caller's.

The Quit Entry

The Quit entry is used to take another process into blocked status, if it is not already blocked. The procedure is straightforward:

1. If the process in question is already blocked, nothing need be done. Quit returns to its caller.

2. If the process in question is running, the Quit module resets the Wakeup Waiting switch for the process, and generates a process interrupt, the "Quit interrupt," for the appropriate processor. The meaning of this interrupt is that the process should call Block; in the description of the Interrupt Interceptor, we see exactly how this call comes about.

3. If the process being blocked is ready, it is merely removed from the ready list and its Active Process Table entry modified to show that it is blocked.

Although the Quit module does not call other Process Exchange modules, it is considered part of the Process Exchange because its activities must be coordinated and interlocked with other Process Exchange modules.

Use of the Process Exchange

The Process Exchange provides only the most primitive functions required to implement processor multiplexing and inter-process control communication. Although it (together with the Interrupt Interceptor) could be used directly for these purposes, in a practical system two additional groups of modules must be added: A validator, and an inter-process communication facility. The validator is a procedure which checks the authority of one process to signal another; the inter-process communication facility provides a convenient mechanism for passing certain data as well as control signals between processes.

Both of these facilities can be implemented outside of the Process Exchange, therefore they are described separately in section BD.8.

On the other hand, the Process Exchange is also used by the Basic File System, to block a process until a missing page is available, for example. To minimize the number of procedures which must be always in core memory, the Basic File System calls the Process Exchange directly. Validation of calls is not an issue in the Basic File System, since it is at the same level of "trustworthiness" as the Process Exchange itself. Similarly, an elaborate inter-process communication mechanism is not required for the simple functions needed by the Basic File System.

The diagram in figure two illustrates this pattern of usage of the Process Exchange.

Interlocking and Masking in the Process Exchange.

In this overview we have so far ignored completely a difficult problem. The Process Exchange is designed to be used by several processors simultaneously, and each is capable of being interrupted and re-entering the interrupted procedure. We have neglected completely the issue of keeping the multiple processors out of each other's (and indeed their own) way.

There is in fact a very simple solution to this problem, as follows: A system-wide interlock, based, for example, on the read-alter-rewrite cycle capability of the 645 memory, is placed at the entry to the Process Exchange. This interlock must be tested and set by any processor desiring to enter the Process Exchange. When the processor leaves the Process Exchange (perhaps executing in another process) it resets the lock. If a processor coming in finds the interlock on, it is expected to loop, waiting for the interlock to be switched off by some other processor

which is presumably already inside the Process Exchange.

In order to guarantee that a processor does not set the interlock, enter the Process Exchange, take an interrupt, and attempt to re-enter the Process Exchange (presumably looping forever on the interlock) a processor is completely masked against all possible interrupts while inside the Process Exchange. It is instructive to note that in a single-processor system only masking is necessary--the interlock can in principle be dispensed with, except as a debugging aid. It is also instructive to note that 7094 CTSS, a single processor system, uses both an interlock and a mask, and is occasionally found stuck on the interlock. This effect can be the result of either hardware failures or errors in system programs.

The simple interlocking mechanism we have described would be completely adequate except for the possibility in a several processor system that the Process Exchange could become a potential bottleneck, delaying interrupt response. This is because the simple interlock forces exclusion of all but one processor at a time.

To prevent such a possibility, one can use instead a set of process-by-process interlocks in the Active Process Table. This technique, while adding considerably to the complexity of the Process Exchange, means that two processors with independent objectives can execute in the Process Exchange simultaneously, and that interrupts may be allowed except in certain well defined critical areas. Section BJ.6 describes in detail the structure of this more elaborate interlocking strategy. Also, each of the sections of BJ.3 which follow are divided into two parts, describing the way the module works assuming the simple system-wide interlock and total masking, and then describing the additional tests necessary under the limited interlock strategy.

It is planned that the initial implementation of the Process Exchange will in fact use only the system-wide interlock and total masking.

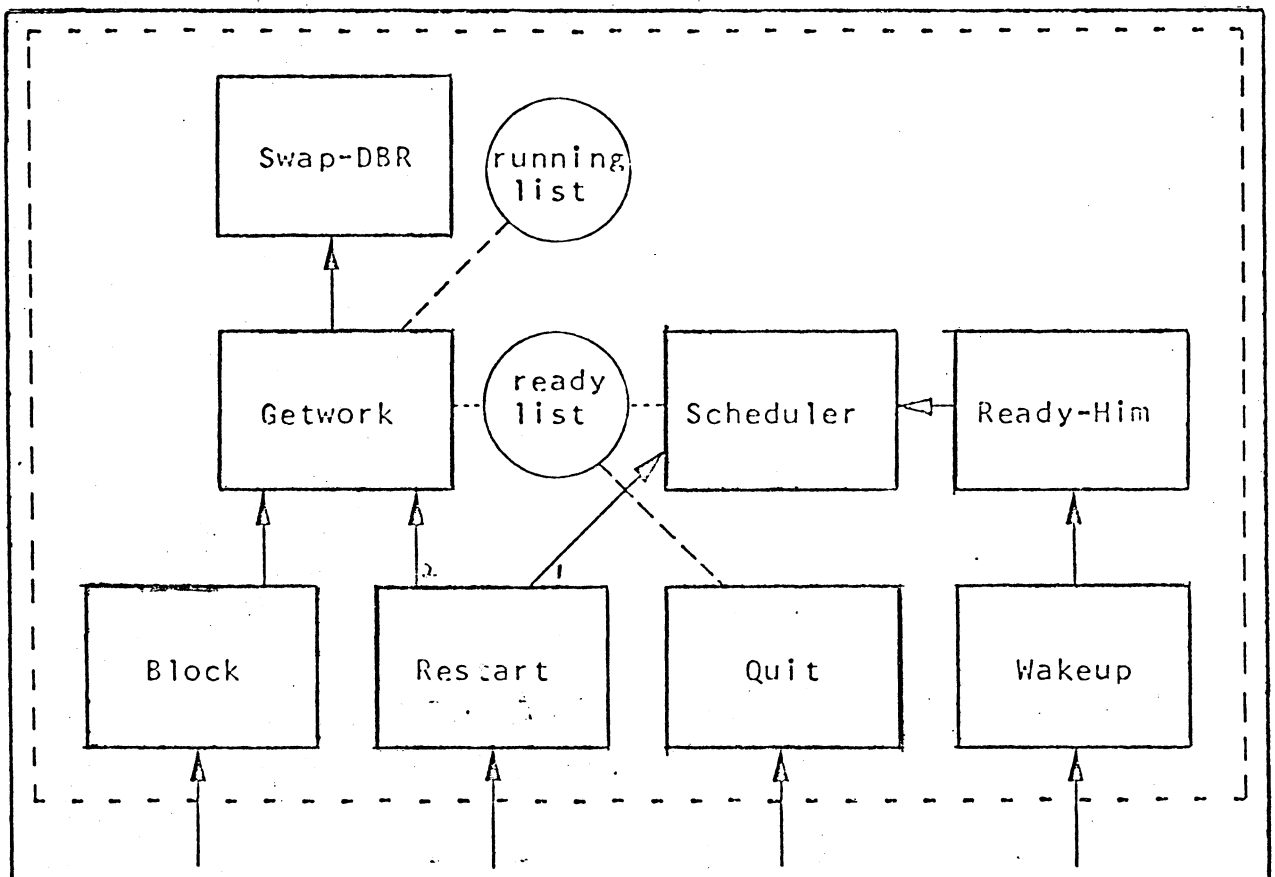

Figure 1 - -Block diagram of Process Exchange.

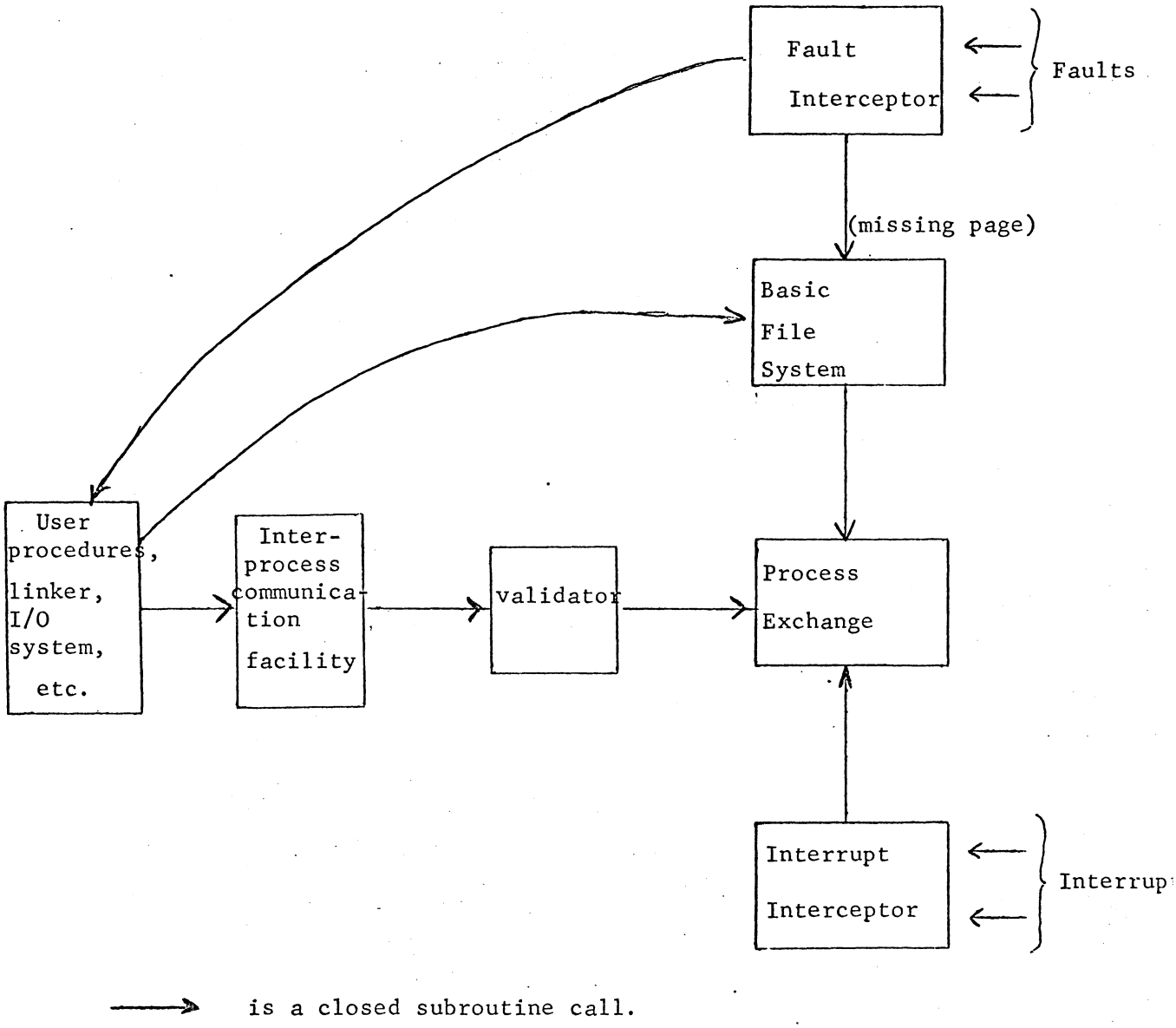

Figure 2. Usage of the Process Exchange