Working Document
David Redell
July 20, 1972

Interprocess Communication

I. Introduction.

Two properties which experiment programs require of the CRMS system are:

A. Ease of parallel programming.

B. High reliability.

Various mechanisms have been provided in general-purpose timesharing systems to insure these two properties. In particular, these have included:

A. Synchronization and intercommunication between cooperating parallel programs.

B. Protection among the various programs and users, allowing precise control over their interactions.

A serious problem with these mechanisms is frequently their overhead cost, particularly in terms of processor time. Often, programs which could benefit greatly from the use of these features are written to painfully circumvent them, in order to avoid the prohibitive overhead. In general, this overhead arises from two sources:

A. Excessive complication.

B. Software implementation.

In an attempt to avoid this overhead, the CRMS system provides a unified parallelism and protection mechanism which is:

A. Simple, yet quite powerful.

B. Implemented primarily in firmware.

The basic computational entity within the system, for both parallelism and protection purposes, is the <u>process</u>. Processes execute processor instructions to interact with other processes, just as they would to perform an addition or a transfer of control. Most of these operations fall within the Interprocess Communication mechanism, as discussed below. The separate, but related, protection mechanism is described elsewhere.

II. Goals.

The firmware-supported Interprocess Communication mechanism (IPC) is designed to meet four goals:

A. To allow coordination of cooperating parallel processes.

B. To provide a general "monitor call" operation, allowing software services to extend the machine in a flexible and general way.

C. To handle traps, i.e., errors made by a running process, which the process itself is unable/unwilling to handle.

D. To replace interrupts as the interface between software and I/O devices.

The IPC, as described below, provides for the synchronization of processes, and the transmission of messages between the address spaces of processes, with intermediate buffering provided when appropriate. The mechanism consists of firmware capable of handling the optimal (and, hopefully, most frequent) cases, and of software to handle the more difficult situations. The firmware uses its own trap mechanism to allow a graceful escape into the supporting software when necessary.

Within the same conceptual framework, somewhat different facilities are provided by the two processors, reflecting the different orientations of APL and SIMPLE. Generally speaking, the SIMPLE processor provides a larger set of low-level firmware IPC operations, while the APL processor relies more on trapping to software (written in SIMPLE) which provides APL-language-oriented communication operations.

III. Intervention Operations.

The simplest kind of interaction between processes occurs when a superior process intervenes in the affairs of a subordinate process. In a sense, this is not really interprocess communication at all, since only the superior process is actively participating, while the subordinate process is passively manipulated. Such intervention occurs, for example, when the debugger stops a subordinate's endless loop or when the memory manager suspends a process in order to swap out its memory. Usually, the intervention occurs in three steps:

A. Stop the subordinate process.

B. Examine and/or modify the process in some way.

C. Start the process running again.

Here, we are concerned with steps A and C , which are generally necessary to insure the consistency of step B . Two processor operations are provided, called <u>Stop</u> and <u>Start</u>. When a process has been stopped by another process, it cannot run under any circumstances until it has been started again, normally by the process which stopped it.

IV. Messages.

A. Format.

A message is a variable length sequence of capabilities and data. It is represented in memory by a sequence of two-word descriptors. Each descriptor represents an item in the message as follows:

1. Datum:

   a. A one-word scalar is represented by a two-word immediate descriptor.

   b. A two-word descriptor is represented by itself.

2. Capability:

   A capability is represented by a two-word capability reference (i.e., pointing into C-list of the process).

B. Transmission (see figure 1).

A message is transmitted from the top of the sender's stack to the top of the receiver's stack. Ideally, the transfer is done directly from the sender's data and capability segments to those of the receiver. This is the optimal case handled by the firmware without any software intervention. Buffering and synchronization considerations may require traps to software in other cases.

V. Message Operations.

A process uses the message mechanism by executing one of several processor operations. The basic operations are send and receive; the others are best explained in terms of these two.

Send and receive function in matching pairs. Whenever a send-receive pair is matched, the message is transferred from the sender's memory to the receiver's memory. When a send (receive) is executed before its matching receive (send), the process is blocked until the matching operation is performed. A variant of the send operation, send-buffered, allows the sending process to continue computing even if the matching receive has not been performed. This is done by trapping to software instead of blocking the sender. The software can buffer the message until the receive is executed. Note that the software is only invoked when buffering is necessary; if the receive precedes the send-buffered, the direct firmware transmission takes place as with a normal send operation.

In order that a receiving process may exercise some control over its reception of messages, each receiving process is equipped with a number of ports. Both the send and receive operations require as a parameter the port on the receiving process through which the message is to be transferred. The receiver specifies the port via an unprotected integer, while the sender must present a destination-capability which specifies both the receiving process and the port number. Thus, the receiver retains firm control over incoming message traffic.

The send-receive mechanism outlined above is essentially adequate to serve as a monitor-call mechanism. (i.e., user program sends "request" message to monitor, which performs service and sends "response" message back to user). To protect itself from unwanted blocking, however, the monitor must use the send-buffered operation to send its response message. Actual buffering of the data should not be necessary, of course, since the user program will normally wish to wait for the response before proceeding. To insure that the entire transaction can be handled by firmware, it suffices to provide an atomic send-receive sequence, called exchange, to be employed by the user program in calling the monitor. By atomically sending the user's request and receiving the monitor's response, the exchange operation guarantees that the user will ask for the response before the monitor sends it, eliminating all intervention by the buffering software.

VI. Traps.

Occasionally, during the execution of a process, the processor will detect an unusual condition which requires the attention of some responsible program. A flexible trap mechanism for the selection and activation of this program avoids the proliferation of ad hoc mechanisms to deal with each case individually.

When generating a trap, the processor first pushes onto the trapped-process stack certain information identifying the condition which caused the trap. It then examines the relevant trap-mask entries to determine what process should handle the trap.

A process may want to handle certain of its own traps. In this case, the trap is treated as an implicit function call, whose actual parameters are the information identifying the trap. Otherwise, some superior process must be located, the trap-mask of which is set for the trap being generated. The tree of processes induced by the father pointers is scanned upward (toward the root) until such a process is found. Two actions are then performed atomically by the processor:

A. A message is sent from the trapped process to the superior process on port 0. The message consists of the trap information previously pushed onto the stack, and a capability for the trapped process, which is fabricated by the processor.

B. The trapped process is stopped.

The superior process can then take corrective action, if possible, and restart the trapped process. (see figure 2)

A processor operation is also provided which simply causes a trap. The main use of this facility is by APL processes as a request for service from the APL external runtime process. (The regular exchange operation is not usable by APL processes since they have no capability lists.)

Since a trap normally occurs during the course of an instruction, the trapped process is stopped in a somewhat different state from a process which has been stopped by another process. The program counter of a trapped process points "at" the offending instruction, rather than "between" instructions. The process which handles the trap may specify (as a parameter to the start operation) whether the offending instruction is to be restarted or aborted.

VII. I/O Devices.

By reducing scheduling overhead, the firmware IPC improves the reaction times of processes sufficiently to allow direct control of each I/O device by a process, rather than by the conventional "interrupt routines," which must be activated by a special mechanism, separate from that which schedules the rest of the processor's work. Each hardware _device_ has a firmware _controller_ and a software _manager_ process. The interactions between the controller and the manager process are implemented as a stylized form of interprocess communication.

From the point of view of the process, messages are sent and received, and data is passed through shared memory. The controller appears to be another process (and can, in fact, be simulated by another process for testing purposes).

From the point of view of the controller firmware, a number of facilities are available which allow it to masquerade as a process. A dummy entry in the process table allows the controller to send messages, much as a real process would, by calling a firmware routine. The process table entry of the manager resides at a fixed location in memory, as does the communication area. (The system initialization software is responsible for creating the manager at the proper spot in the process table, with the communication area in its address space.)

VIII. Various Details.

The previous sections have described the IPC from an external or "top-down" point of view. What follows is a brief "bottom-up" discussion of the major internal components of the IPC.

A. Process Table (see figure 3).

Each process in the system is represented by an entry in the process table. This process table entry (PTE) contains the stateword of the process when it is not running, as well as various other scheduling and addressing information. The PTE of a process is core-resident; all other components of the process are segments, and are thus swappable to the disk.

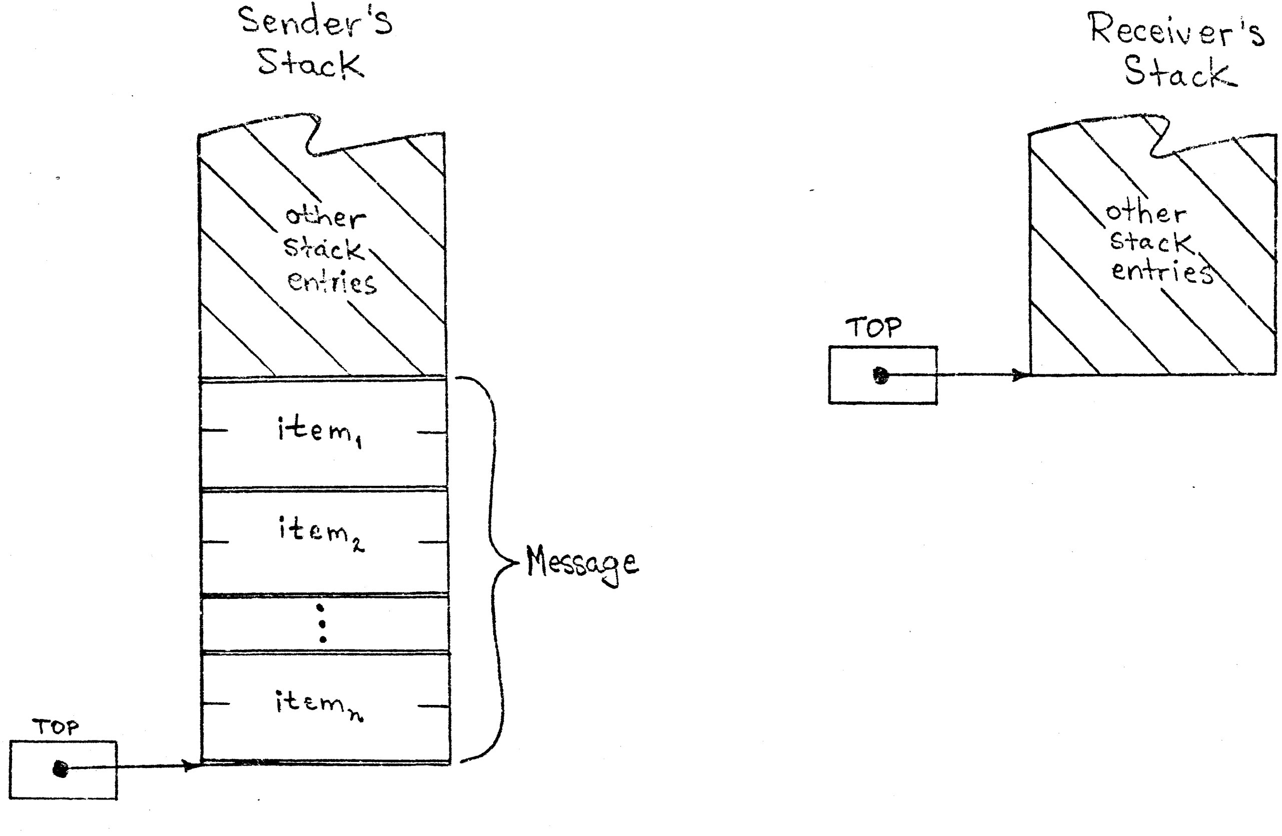
B. Ready-list (see figure 4).

Each processor has a ready-list in memory, containing all the processes which are requesting service on that processor. Each ready-list is structured to represent the various priorities of the ready processes, and each processor is to run, at all times, the highest-priority process on its ready list. Microcode primitives are available to insert and remove processes in the ready-lists; these are used by the higher level IPC operations (send/receive, stop/start, etc.). Since each processor can insert processes in the other processor's ready-list, a pre-emption signal line is provided, allowing a processor which inserts a new highest-priority process in the other processor's ready-list to so inform the second processor. This is the only hardware signal line between the two processors.

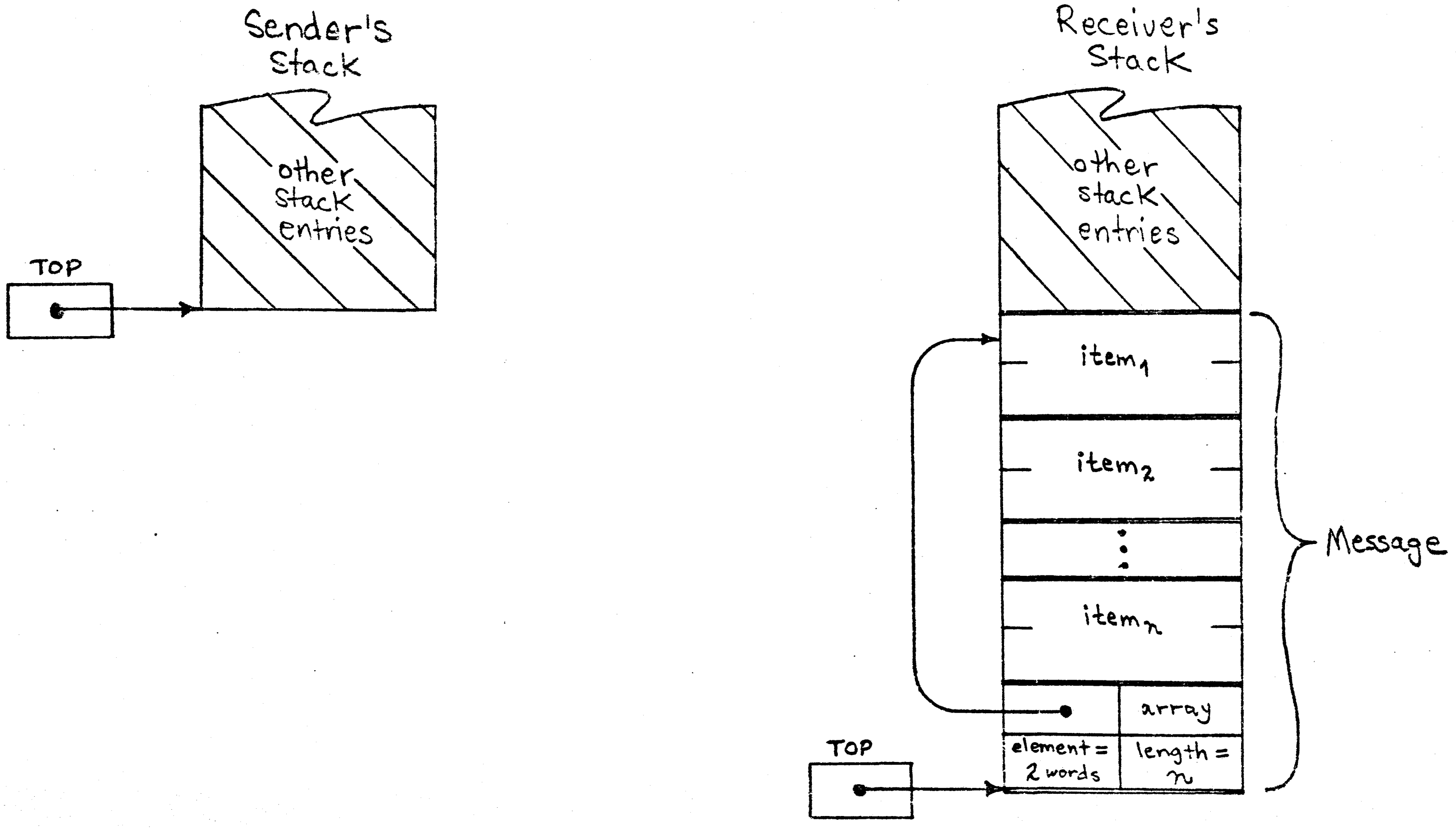
C. Quantum Timers (see figure 5).

Assigned to each process is a time quantum, which represents the

C. expected time that it will run when serviced by the processor. Whenever the processor begins running a process, this quantum is stored in a special memory cell, where it is decremented every millisecond. If the cell ever reaches zero, a quantum-overflow trap is generated, and software is invoked (presumably to reduce the priority of the process and increase its quantum).

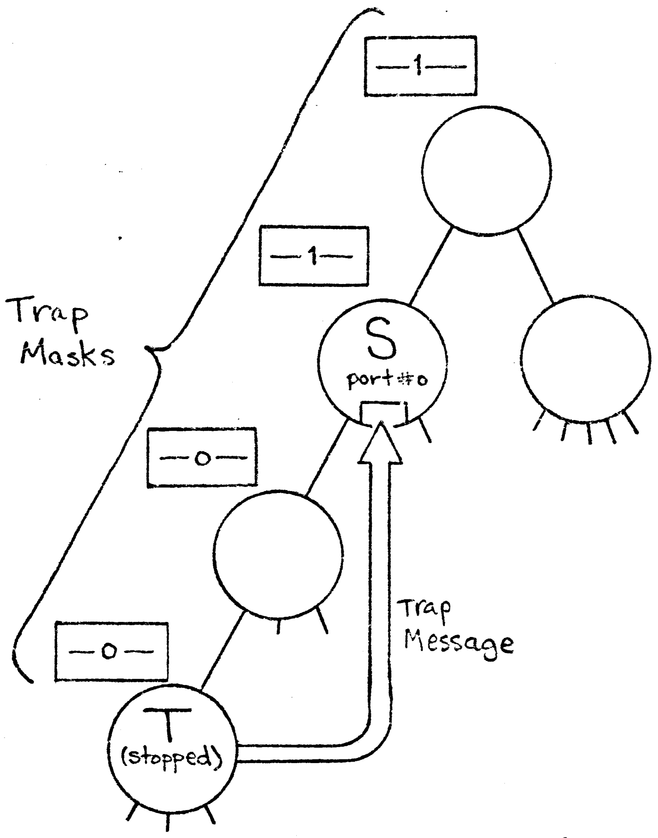


1a: Before Transfer



1b: After Transfer

Figure 1: Interprocess Message Transfer

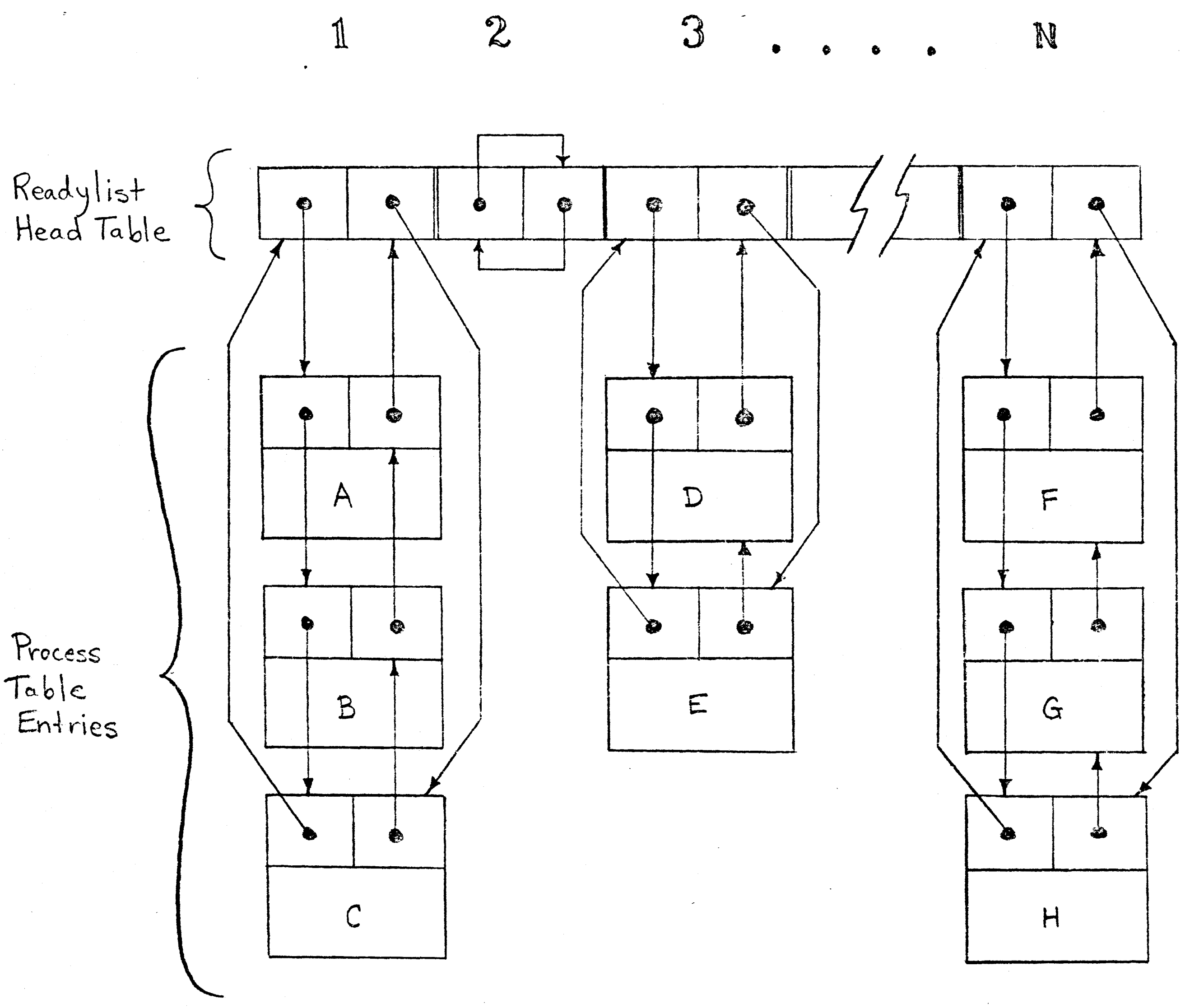


Example:
Process T has caused a trap, which will be handled by Process S.

Figure 2: Example of Trap Handling

PROCESS TABLE ENTRY:	words
Unique name	1
Father pointer	1
Scheduling data Processor type (APL/SIMPLE) Priority Quantum State	1
Message Data Message size Input port # Output port #	1
Link data Readylist links Send-queue links	2
Address Space Data	2-3
Processor Stateword	6-8
Total	approx 16 words

Figure 3: Process Table Entry



Note:

1) At priority 1, processes A, B, & C are ready
   At priority 2, no processes are ready
   At priority 3, processes D & E are ready
   At priority N, processes F, G, & H are ready

2) Given this situation, the processor should be running process A, which is the first process in the highest priority chain of the ready list.

Figure 4: Ready List

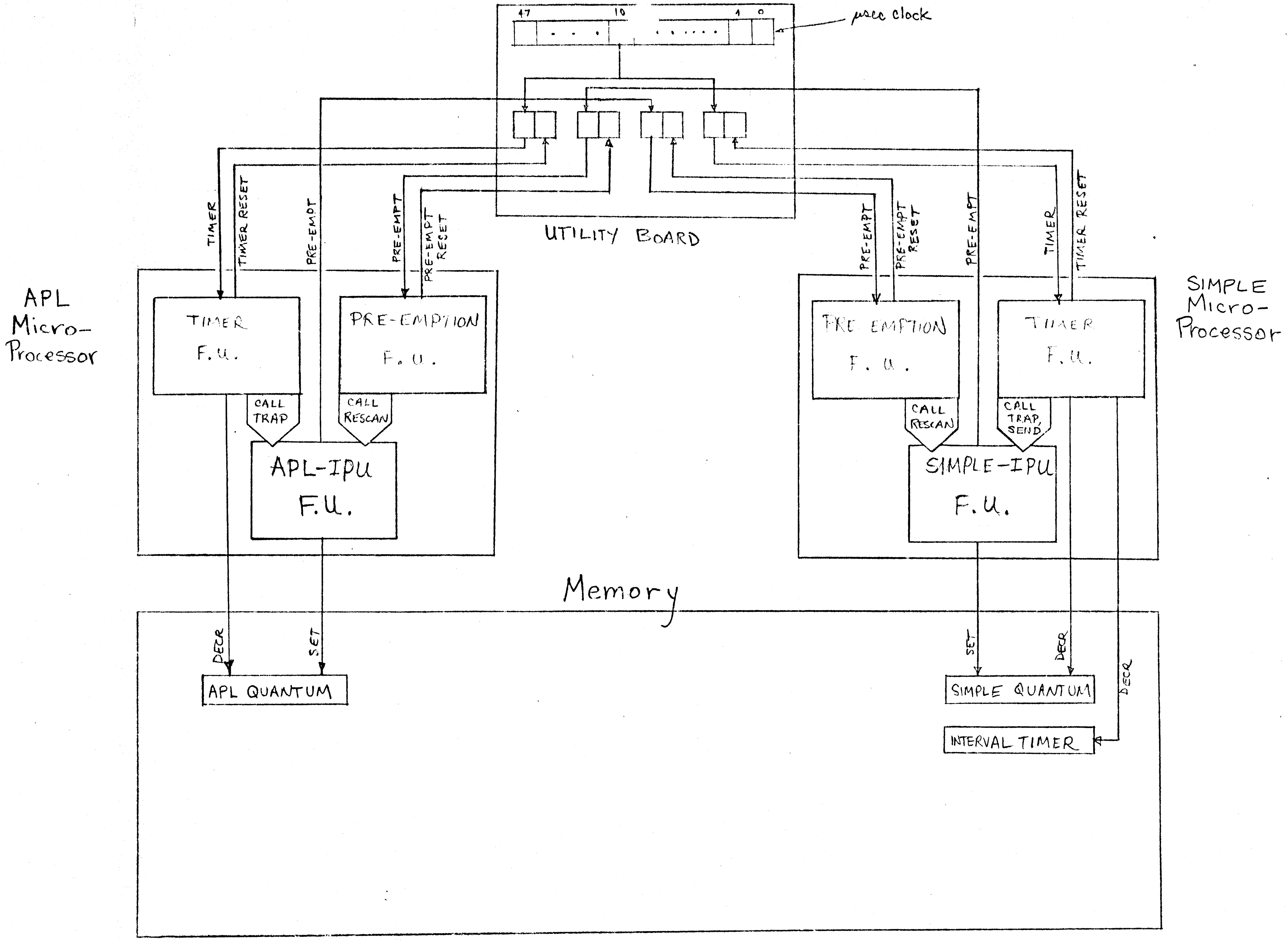


Figure 5: Quantum & Pre-emption