REVISED

AUGUST

PDP-1 COMPUTER
ELECTRICAL ENGINEERING DEPARTMENT
M.I.T.
CAMBRIDGE, MASSACHUSETTS 02139

PDP-41

PDP-1-X PROCESS SCHEDULING

June 9, 1966

PDP-1-X PROCESS SCHEDULING

One of the most important facilities provided by the PDP-1-X is parallel programming. A computation is allowed to have many processes; ~~these~~ runable processes time-share the central processor for the duration of ~~the~~ a computation's "quantum". The executive function of scheduling processes is carried out by special hardware, with minimal help from software (the executive). This memorandum describes both hardware and software aspects of process scheduling.

The processes which are subject to scheduling are those processes which are <u>active</u> (i.e. not hung in e.g. i/o wait) and will be restarted in a program ~~image section~~ field which is currently in physical core memory. Such a process is called <u>runable</u>. The runable processes are organized into a multi-level <u>process queue</u> (or <u>run queue</u>).

The run queue contains just those processes which the system is "willing" to run at any time. Before describing the scheduling policies applied to this queue, we will specify the internal organization of the queue, and discuss mechanisms which place processes in the queue or remove them from the queue.

The queue has eight priority levels, numbered from 0 (high priority) to 7 (low priority). There is a 16-word <u>queue head table</u>, which contains pointers to heads and tails of rings of processes. A process is represented by a <u>process entity</u>, which contains four pointer words and a status word

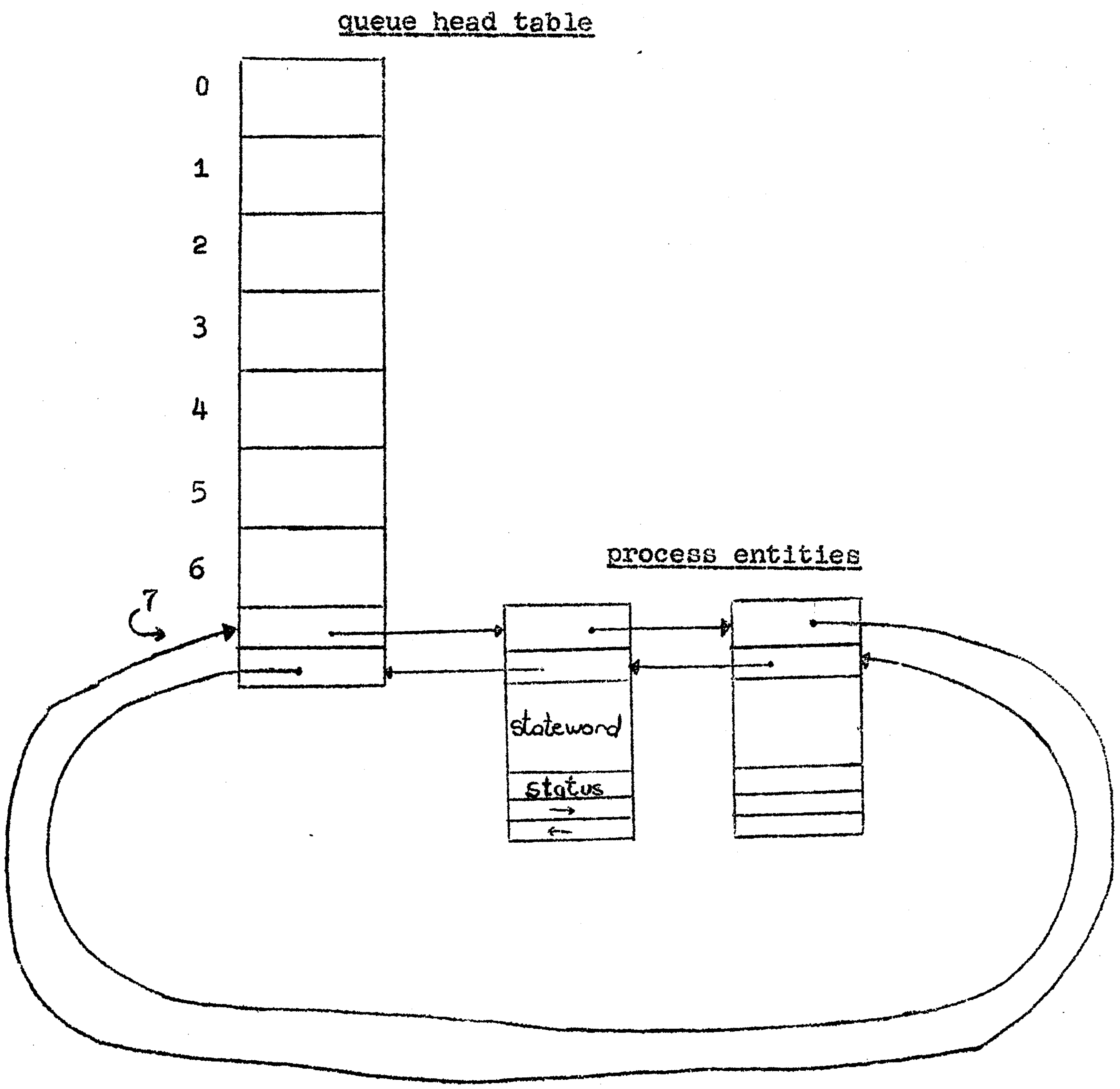

Figure 1. A ring of two processes at level 7 of the queue.

as well as the six words of stateword. In a process entity, two of the pointer words link the process in a ring of the run queue (these pointers are used for (many) other purposes when the process is not active), and the other two link the process to its computation entity. ~~(Each computation entity has four rings of process entities: one for each of its program image sections. A process running in core j will be on ring j.)~~ (See figure 1.)

There are many mechanisms which cause processes to enter or leave the queue. Most of these mechanisms affect only one process at a time, but a few can affect many processes. These are computation scheduling and the meta-instruction *stop*.

The computation scheduling algorithm will remove processes from the run queue when it swaps their program field ~~image section~~ out of core, and adds processes to the run queue when it brings their program field ~~image section~~ into core. ~~In addition, the computation scheduling algorithm runs all runable processes while switching between computations, and hence adds such processes to the queue when starting to switch, and removes them before starting the new computation's "quantum". (During a computation's "quantum", only executive processes and that computation's processes may run.)~~

The meta-instruction *stop* suspends all processes in a specified computation, and creates suspended process capabilities and corresponding new processes at the fault address of the computation which said *stop*. A process is suspended by making it inactive (and removing it from the run queue, if it was there).

The following mechanisms involve only a single process.

When an i/o function is started, the process which started the function is suspended and an entry is made in the <u>i/o function started table</u>, pointing to the newly suspended process.

When an i/o function completes, the process indicated by the i/o function started table entry is made active, and, if runable, is inserted in the queue at a level determined by the i/o function started table entry. If the process is not runable, the desired level is remembered in the process status word, ~~and~~ the associated computation entity is placed in the io completion table and its RACT bits are marked for priority restart.

On an i/o error (function busy or function tardy), the running process (which must have caused this error) is suspended, and a suspended process capability and a corresponding process at the fault address in the superior computation are created. The same action is taken on lock faults, breakpoints, illegal opcodes, illegal address snags and halt instructions.

When a process <u>quits</u>, the process is removed from the run queue and the process entity is returned to the pool of empty process entities.

When a process <u>forks</u>, a new, empty process entity is added to the queue level of the forking process, and is made the process entity of the forked process. Execution of the computation continues with the forked process.

On an address snag, the running process is ~~suspended and added to a list of processes which are waiting for the needed program image section.~~ removed from the run queue, and the corresponding ACT bit of its computation entity is set to 1. When this section reaches core memory, ~~these~~ this ~~processes are~~ process is returned to the run queue. ~~(On the first such address snag, the drum transfer request is given to the drum mover.)~~

When a process does an enter, the process is suspended and a new process is created in the executive, at the address specified in the invoked entry capability. (The new process has a pointer to the process which entered.)

Whenever any of these mechanisms is used to modify the contents of the run queue, it is always necessary to recompute the number of processes linked to each level of the queue. The running count is kept in the 8-word queue population table. A simple scan of this table allows us to find the highest (in priority) occupied level.

Finally, we discuss scheduling policy for the queue just described. The essence of the policy is to run processes from the highest occupied level of the queue, but to do so fairly.

With each level of the queue is associated a particular quantum time, which must be a multiple (between 1 and 31) of the subquantum time, which is 1.28 ms. (256 memory cycles). The scheduling algorithm allows us to switch processes only at the end of a subquantum. When we do switch processes, there are two possible reasons for doing so:

1) A process of higher priority has appeared in the queue.
2) The current process' quantum has expired, and there are other processes to run, at this queue level or lower.

In the first case, we interrupt the running process as soon as the current subquantum ends, and run the first process in the highest occupied level of the queue. The interrupted process remains at the head of its queue level, and contains in its stateword a non-zero quantum counter (which records how far along in its quantum it is). Thus, when this process resumed, it will finish its quantum (rather than starting a fresh one).

In the case of an expired quantum, the process is removed from its level of the queue and tacked onto the tail end of one of the lower levels. Its quantum counter will be zero, indicating that its quantum (at the new level) has not started. The choice of which level the process enters is made by hardware, and depends on how many quanta the process was given while it was in execution. After the expired process has been relinked to the queue, the first process in the highest occupied queue level is run.

It may be that, at the end of the running process' quantum, there is no equally deserving process. If the above policies were followed in this case, there would be some wastage of time as we stopped the process, linked it to the queue, searched the queue, and finally restarted the same process again. So this case is recognized by the hardware and no interruption occurs.

Special hardware assists in the scheduling by keeping running track of quanta and the priority of the current process; this hardware determines when either of the above trapworthy situations exists, and causes a PREEMPT trap or a RND RBN trap (for cases 1 and 2, respectively). The hardware has a queue priority register QP (3 bits plus a 1-bit extension QE which indicates when the queue is empty), a current priority register CP (3 bits plus a 1-bit extension HP which indicates not completely correct

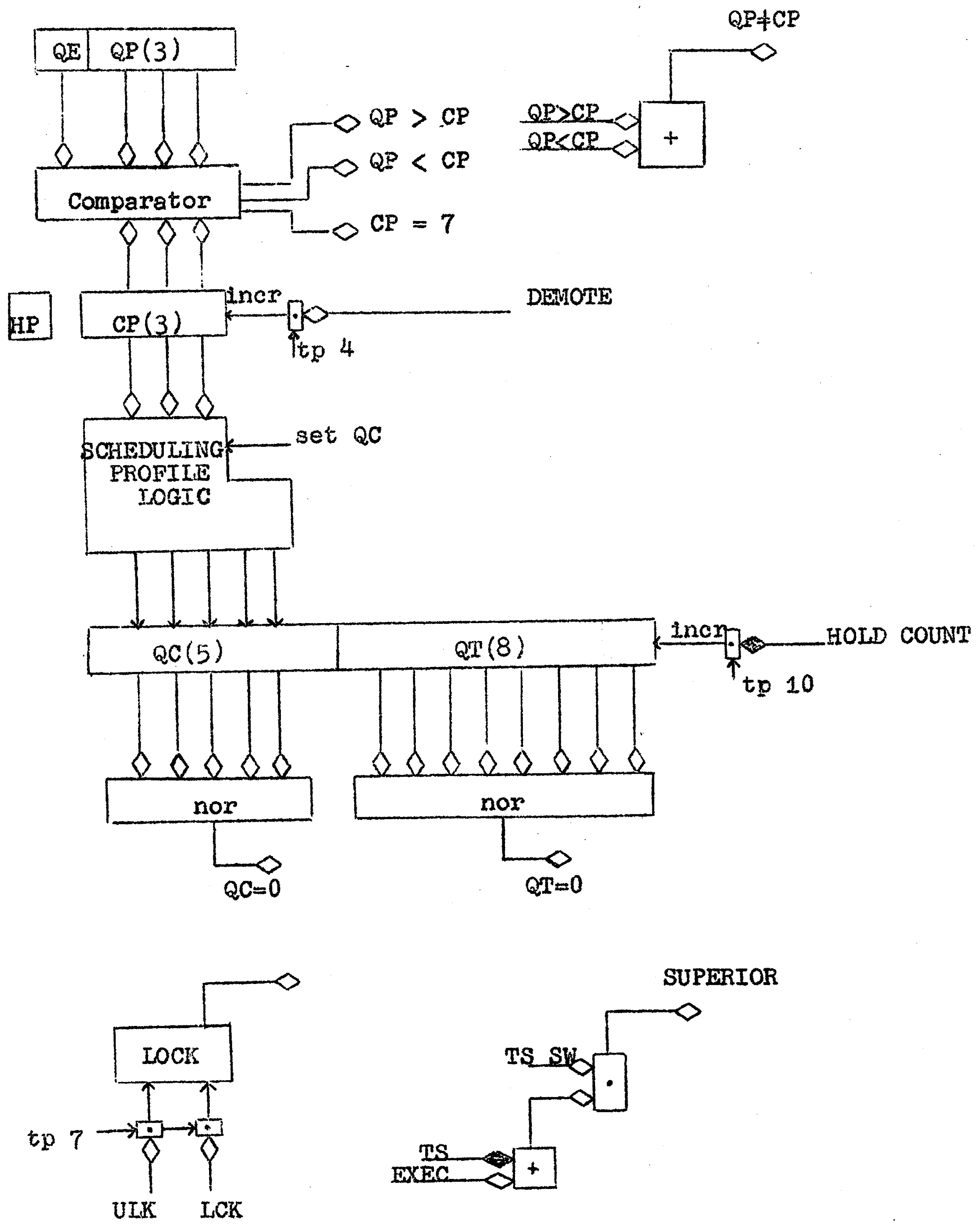

Figure 2, part 1. Process Scheduling Hardware not completely correct

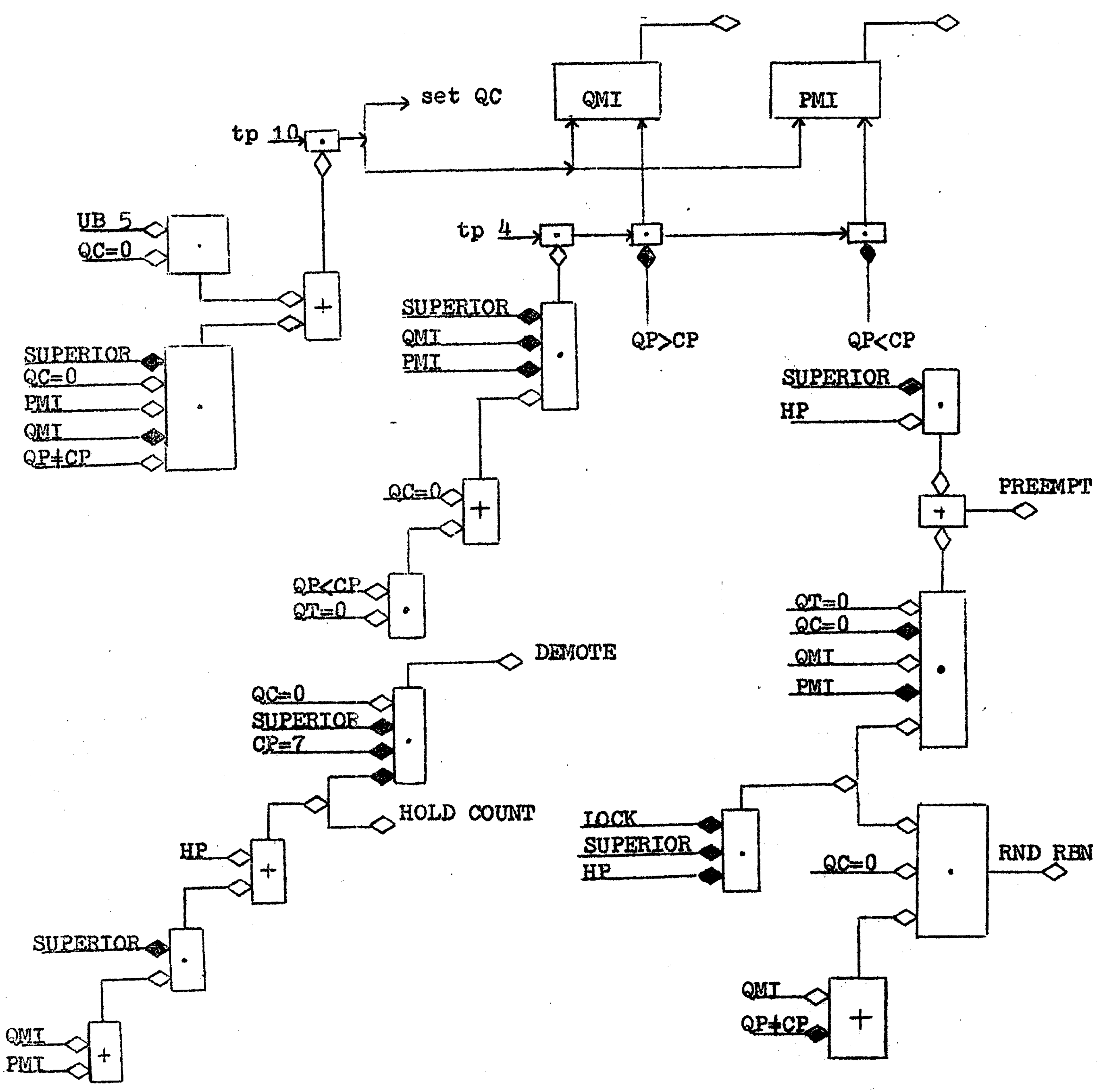

Figure 2, part 2. Process Scheduling Hardware when the PDP-1-X has nothing to do), a quantum timer QT (8 bits) and its extention, the quantum counter QC (5 bits). The register CP contains the queue level number of the running process. CP is loaded when we start to run the process and as the process runs, the hardware may change (increment) it. When the process returns to the queue, its level of insertion is determined by CP. The register QP is always reset (by software) to the level number of the highest occupied level of the queue (not counting the currently running process as being in the queue).

The process scheduling hardware is shown in figure 2.

Not shown or mentioned above, although essential to efficient operation of the PDP-1-X, is the special break) and unbreak hardware with its process pointer register. The break and unbreak hardware deposits and reloads the stateword of the running process directly from the process' process entity, rather than using preset locations in the executive core. The location of the stateword is maintained by the executive in the process pointer register PP (12 bits).

A break requires five memory cycles to deposit the stateword, while an unbreak requires ~~seven~~ eight cycles. There is a special form of unbreak, called unbreak fork, which loads the stateword from one process entity, sets the process pointer to point to a different process entity, picks up the extended address in the cell after the fork which the first process executed (we assume), and starts the new process there. This in effect copies the stateword of the forking process into the forked process at no cost to the system.

The executive uses the process scheduling hardware to decide when to run another process, and merely supplies the hardware with values for CP and QP. The only other computation required when processes are changed is a bit of pointer manipulation (in the queue head table, various process entities, and the contents of the process pointer).

The following pseudo-programs outline executive action required to start a new process, handle a PREEMPT trap, and handle a RND RBN trap.

```
newproc:  comment start running the most deserving process;
          begin find highest occupied level of queue;
                set CP to this level number;
                remove first process in this level from
                this level and fix up pointers around it;
                set process pointer to stateword of this process;
                subtract 1 from appropriate queue population
                table entry;
                find highest occupied level of queue;
                set QP to this level number;
                start the chosen process comment unbreak end;

preempt:  comment program to handle PREEMPT trap;
          read CP;
          return running process to queue at head of ring at
          this level;
          add 1 to appropriate queue population table entry;
          go to newproc;
```

```
rnd.rbn:  comment program to handle RND RBN trap;

          read CP;

          return running process to queue at tail of ring
          at this level;

          add 1 to appropriate queue population table entry;

          go to newproc;
```