Proposal

# An Implementation of an Almost-Segmented Multiprogrammed Computer System for the PDP-1 *

Leo J. Rotenberg

* Work reported herein was supported by Project MAC, an M.I.T. research program sponsored by the Advanced Research Projects Agency, Department of Defense, under Office of Naval Research Contract Number Nonr-4102(01). Reproduction in whole or in part is permitted for any purpose of the United States Government."

## Portrait of the System as Seen by the User

Familiarity with the paper Programming Semantics for Multiprogrammed Computations by Dennis and Van Horn (hereafter referred to as [*], is assumed. The system described here is similar (as much as possible) to the MCS system described in [*]. Features mentioned in [*] are available in this system except where explicitly mentioned below.

### Segments

The system is only rudimentarily segmented. There are two distinct kinds of segments, files and program images. A file is an ordered set of words which is treated very much like a file in CTSS. The naming and protection mechanisms for files are those described for segments in [*]. The program image is an executable segment.

### Protection

Spheres of protection are determined by C-lists associated with program images. Thus every program image defines the sphere of protection in which processes in the program image operate. C-list entries may be file capabilities, directory capabilities, entry capabilities, or i/o function capabilities. An ownership indicator is associated with each capability, and a read-only indicator is associated with each file capability.

### Computations and Processes

Computations and processes are as in [*]. Note, however, that all processes of a computation must share the same program image, since a

C-list is associated with each program image, and this C-list determines the sphere of protection of the computation.

Parallel Processing

The meta-instructions fork, join, and quit, are available, along with lock and unlock. Duplication of private data on a fork is permitted by the ability to specify the complete state word of the new process.

Input/Output

Direct and immediate I/O will be available through use of the meta-instruction execute I/O function. However, protected service routines will be available for I/O requiring buffering (e.g. typewriter communication, reading and punching paper tape).

Inferior Spheres of Protection

The discussion in [*] holds generally for this system, but the create sphere meta-instruction must specify an initial program image size, and there is the possibility that create sphere will fail due to lack of physical storage resources

Protected Entry Points

There will be two classes of protected entry points: entries to executive service routines, and entries to user-specified computations. Execution of an enter with respect to an entry of the second class will be done via an appropriate executive service routine. There may be some arbitrary limit on the nesting depth of enters, due to space and/or programming restrictions in the executive.

Directories

The discussion in [*], holds except for the dectape conventions outlined below. There will probably be an arbitrary limit on the number of characters in a name.

Files

Files are organized in blocks of $400_8$ words, since this is the dectape block size. References to a file may be any of the following

i := create file creates an owned 0-block file with mode RW

[read / write] [quarter / half / full] block a,b,i

(insert / delete) block c,i

where a is an address in the file, b an address in the program image, i the index of a file capability, c a block number

a must be a multiple of the number of words read or written.

Storage of Files and Directories on Dectape

A user may store his personal files and directories on dectape, remove the tape from the system, and bring it back with him when he wants to continue his use of the system. However, there are a few restrictions on the use of dectape. Firstly, a user may store a file or directory on his dectape if and only if he owns that file or directory. Secondly, capabilities in directories stored on tape which refer to objects not stored on the tape are replaced by null capabilities when the tape leaves the system. These null capabilities remain null when the tape is brought back to the system.

## Structure of the User

User programs are written with the usual PDP-1 instruction, with the exception of iot's. These instructions are replaced by the instruction "invoke" or ink, which has the following format:


The ink instruction is used to perform calls on the execution and all input/output. When the ink is executed, the six-bit CODE is taken to be a pointer into the current C-list. If the addressed C-list element is an I/O function capability, the I/O function is performed (see the next section for details). If the addressed C-list element is an entry capability, the process is suspended and a new process is started in the executive, at the address specified by the entry capability and in executive mode. This mechanism can be used for any of the following: communication with typewriters, paper tape readers and punches; reading, writing, and manipulating files; executing meta-instructions such as fork, quit, join, etc.

Referring to a null capability, file capability or directory capability with an ink is illegal and causes the "illegal trap" to the executive.

The user's C-list is stored in the first $100_8$ register of his program image. These words are protected from reading or writing by the user program, except for interpretation by the ink instruction.

Structure of the Outside World

The real world is connected to the PDP-1 through the PDP-1 I/O bus. All interaction between the PDP-1 and the real world are initiated by an ink directed to the bus. The format of such an ink is as follows:

| 7 | 4 | MODE | VAR | FN |
|---|---|---|---|---|

FN is a pointer to an I/O function capability in the current C-list; VAR is a set of variant bits which denote the action desired of the particular I/O function, and MODE determines how the PDP-1 Central Processor is to interact with the I/O bus. The four modes are

mode 0: execute and continue

The PDP-1 waits while the interaction takes place, and then continues executing instructions.

mode 1: execute and pause

The PDP-1 waits while the interaction takes place, and then traps to the executive. The executive will run other processes, and when this I/O function completes, this process will be restarted.

mode 2: (wait), execute and wait

The PDP-1 waits for the device to become free, waits while the interaction takes place, waits for the device to become free again, and then continues executing instructions.

mode 3: (wait) execute and continue

The PDP-1 waits for the device to become free, waits while the interaction takes place, and then continues executing instructions.

Whenever an I/O function has completed, the I/O controller is so informed. If the PDP-1 is not in executive mode and an I/O function has completed, the I/O controller finds the function number of the completed I/O function having the most priority, and interrupts the PDP-1, placing this I/O function number in the AC (after the AC has been saved by the trap hardware.)

## Structure of the Executive (Overview)

The resident executive is contained entirely within the executive core(core7). [However, some executive functions, especially execution of some meta-instructions, are performed by phantom users.]

There is an executive C-list in the first $100_8$ location of core 7. The executive runs in the sphere of protection defined by this C-list, but these locations are not protected from manipulation by the executive. Hence (as expected), the executive has access to every computing object in the system.

There are 6 methods of entry to the executive. These are:

1) Execution interrupt, caused by completion of an I/O function.
2) Execution interrupt, caused by expiration of the current user process's quantum.
3) Entry trap, caused by the user's invoking an entry capability.
4) I/O pause trap, caused by the user's invoking an I/O function capability, when the mode of the link is 1.
5) Illegal trap, caused by the attempted execution of some illegal operation.
6) Initial entry, which occurs only when the system is started from scratch.

There are two different modes in which the executive may operate. The first of these, called "executive mode", is the mode which is entered on any of the six entries to the executive described above. While in this mode, completions of I/O functions do not cause interrupts. The non-interruptability of executive mode is essential to performance of the meta-instruction lock and join.

The second mode in which the executive runs is called "priveleged mode". When in this mode, the executive is interruptable, but an interrupt brings the machine back into executive mode to dispose of the interrupt. Those sections of the executive which run in priveleged mode include the computation scheduling algorithm, the drum mover, and the block mover.

## Structure of the File System

The file system provides a convenient facility for use in referring to files stored on dectape. When a file is in use, parts of it reside on the magnetic drum, and any transfer of words between the user's program image and the file is made via a core-drum transfer. The file system keeps track of where all parts of a user's files are, and causes information to be moved between the drum and dectape whenever necessary.

The data residing on the drum includes those file blocks which have been most recently written on read. The oldest (in the sense of recent use) blocks on the drum are transferred to dectape when a need for space arises. (The file system will try to maintain about $40_8$ unused block spaces on the drum.)

Since part of the pointer data manipulated by the file system is contained in files, a single dectape is reserved for system use. When the file system's directory file blocks lag in usage, they will be transferred from the drum to dectape (like any other block).

One important feature of the file system is that users are freed from worrying about where (in the sense of physical addresses) a file is stored. As a result, the problem of maintaining such information is left to the file system. File pointer data is kept in the executive, in the file phantom, and in certain files.

Instead of specifying physical drum or tape addresses, a user's references to file invariably specify some file capability in his C-list. Within the file capability is a 12-bit code which uniquely identifies the file. References to files also specify some block within the file. The file system effects the translation between the code and block number, and the physical address of the desired block. Such address information is stored in the executive if the block has been used recently (relative to other blocks in the system), and is

kept in the file phantom or stored in one of the file system's files, otherwise. In the latter cases the address information will be moved to the executive if that block is referred to.

Block address information stored in the executive locates just those blocks which occupy the active drum space. If the block has not been referred to recently, it will be stored in inactive drum space or on dectape, but it will be moved to the active drum space as soon as it is read or written.

We assume that there are $400_8$ blocks of active drum space and $1400_8$ blocks of inactive drum space.

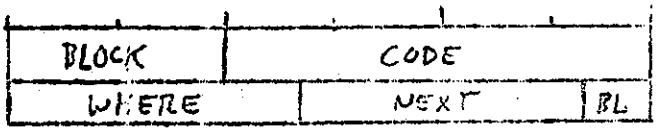
The translation between a code and block number, and a physical drum address, is performed with the help of the hash table in the executive, which is a hash-coded association of $\langle$code, block$\rangle$ pairs with drum addresses.

Suppose a user process tries to read a block of a file. Given the code and block number, the executive uses the hash table to find the physical drum address. If an appropriate hash table entry cannot be found, the executive asks the file phantom to move that block into active drum space and place appropriate address information in the hash table. When the file phantom has finished this task, it restarts the process which referred to the file. [That process had been made inactive by the executive when it called the file phantom. When the process is restarted, it makes the same file-operation request; but this time the address information for the specified block is available to the executive.]

Given an address to a block in active drum space, the executive can simply place the read request on the drum job queue, and deactivate the requesting process. When the drum has completed the physical transfer of data, the drum mover [another part of the executive] restarts the process [at its 'next' instruction.]

## Objects in the File System

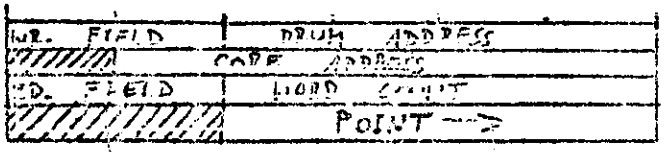
The hash table is a $1000_8$ - word table of 2-word entries, with format as follows:




The hashed initial-lookup algorithm is to take the exclusive-or of (2· block) and (8· the last 6 bits of code), and add this to the table base. BL, BLOCK, and CODE are the block and code of this table entry. WHERE is a pointer to the drum. NEXT is the pointer along the chain of entries having the same initial-lookup address.

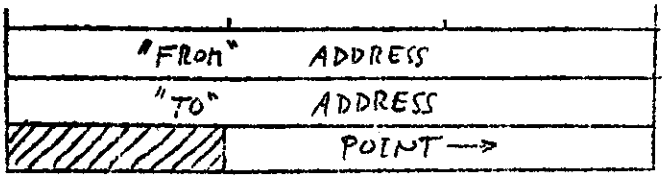
The reference buffer is a $100_8$-word ring buffer of addresses of blocks which have been referred to. The executive writes this buffer as users refer to files, and the file phantom reads the buffer to keep the drum history table up-to-date.

The drum job queue (DJQ) is a sequence of jobs to be performed by the drum. The format of a DJQ entry is as follows:




POINT —> is a pointer to a process which should be restarted when the drum transfer is completed.

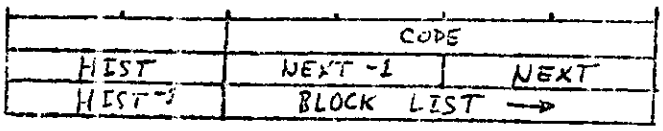
The block job queue (BJQ) is a sequence of jobs which involve movement of blocks between the drum and other parts of the drum or dectape. The format of a BJQ entry is as follows:



POINT —➤ is a pointer to a process which should be restarted when the transfer of the blocks is completed. (This will be a file phantom process.)

The hash table, reference buffer, drum job and block job queues are the only file system objects located in the executive core. The reminder of the active file addressing information is kept in the file phantom tables, which will be described next. Inactive file information is kept in the code file and block list file, which will be mentioned in due course.

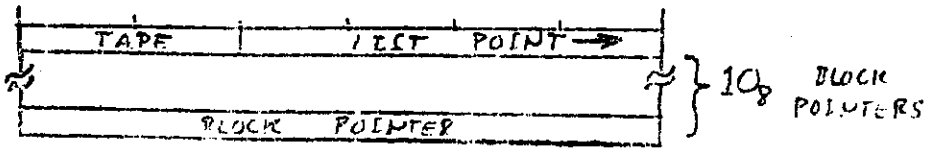
The code hash table in the file phantom is a $300_8$-word table of 3-word entries, each of which represents an active file. The format of a code hash table entry is as follows:




The last 6 bits of the 12-bit code are taken as a pointer into this table. CODE is the code of the table entry. NEXT and $NEXT^{-1}$ are 6-bit pointers along the chain of entries having the same [code mod $100_8$]. HIST and $HIST^{-1}$ are pointers along the chain which determine the relative age of entries in the table. BLOCK LIST —➤ is the pointer to the block list for the file whose code is CODE.

There is a list, within the code hash table, of unused entries; this is the free code hash table entry list.

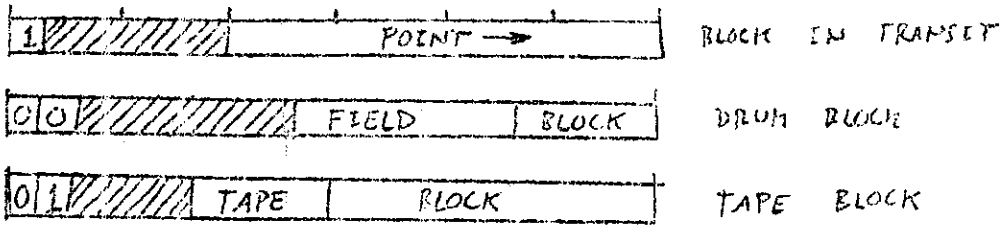
A block list is a list of block list elements, each of which is $11_8$ work long and contain $10_8$ block pointers. The format of a block list element is as follows:



A file having between 1 and $10_8$ blocks requires one block list element; two elements accomodate up to $20_8$ blocks, etc. TAPE is the number of the dectape on which the file belongs. LAST POINT —> is the pointer to the next block list element.

Unused block list elements are kept on the free block list element list.

A Block pointer generally points to a physical address on the drum or on dectape. However, at times a block is in transit between two different physical locations. If a process refers to the block at such a time, it will be deactivated, and re-activated when the block reaches its destination. When such a situation exists, the block pointer points to the list of processes (in the executive core) which referred to the block. The block pointer format is as follows:

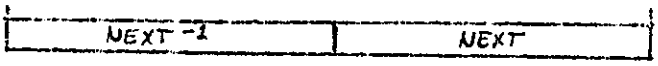



POINT —> is a pointer to a list of processes. FIELD is the number of a field on the drum. TAPE is the member of a dectape. BLOCK is a physical block number, denoting some block within the field or tape.

Information about inactive files is stored in the code file and the block list file. The code file is a file whose length must be greater than the largest code in use. A direct lookup is this file, using the code as a file address, yields a pointer into the block list file (if the code is in use) where the block list for the file starts. At times, the block list of a file

will be neither in the file phantom nor in these files, because it is being moved between them; at such times the code file will contain, for that file, a pointer to a list of processes (in the executive core) which referred to that file while it was in this special status. These processes will be re-activated when the addressing information reaches its destination.

The drum history list is used to keep track of the relative age of blocks of information on the drum. The list occupies a $400_8$ word table, one word per block (of active drum space.) The $0^{th}$ table entry represents the lowest addressed block, and so on. The format of a list entry is:




NEXT and $NEXT^{-1}$ are pointers to the next-oldest and next-youngest blocks.

The drum occupancy table and the free block list both occupy a $400_8$ word table (since no block is both occupied and free, the two usages do not clash). If a block is occupied, its entry in this table points to its block pointer (i.e., the block pointer which points to it) and to that file's code hash table entry; if a block is free its entry in this table is in the free block list. When the free block list becomes relatively empty, aged blocks of information on the drum are transferred to the inactive drum space, and free active drum space blocks are thereby created.

The inactive drum occupancy table and the inactive free block list provide similar functions for the inactive drum space, which consists of $1400_8$ blocks. When the inactive free block list becomes relatively empty, blocks are moved from this section of the drum to dectape. The choice of which blocks to move is heuristic but arbitrary. These inactive drum tables require $1400_8$ words.

The hash table extension is a $200_8$-word "compliation" of the hash table in the executive. It contains a NEXT^-1 pointer for each entry in the hash table. (These are packed two-to-a-word.)

A free tape space table is maintained (for each destape) as a pool of free tape blocks. This is an extension of the free tape space file, which is a file belonging to the file system. Whenever an FTS table becomes too full or too empty, information is transformed between the FTS table and the FTS file.

## Tasks in the File System

### Address Association

When a reference to a given block of a given file is made, the physical address of that block must be available. First the hash table is searched. If the desired entry is found, we're done. If not, the executive passes the problem along to the file phantom. The file phantom first searches the code hash table for an entry with the desired code. If none is found, such an entry is created and placed in the code hash table; and the block list for the desired file is read from the block list file into the file phantom. The block pointer for the desired block is then found, and the block is moved to active drum space by the block mover. The block pointer is reset and a hash table entry for the block is placed in the hash table. Then the process which referred to that block is restarted.

### Drum History Updating

Every time any block of a file is read or written, the drum block involved is made the youngest block on the drum. The names of all blocks used are passed to the file phantom through the reference buffer, and the file phantom updates the drum history list, and the HIST pointer in the code hash table.

### Movement of Blocks

Blocks in active drum space are moved to inactive drum space when they lag in usage, as determined by the drum history list. Blocks in inactive drum space are moved to dectape on a relatively random schedule, which tries to move the oldest blocks first. Blocks in either inactive drum space or dectape are moved to active drum space only when referred to. All of this traffic flows through the executive dectape buffers, and is scheduled by the block mover. The block mover uses the drum mover and the DJQ for all drum transfers, but does all dectape

handling itself; the block mover is responsible for all dectapes in the file system.

Movement of Address Information

The hash table in the executive holds address data for blocks in the active drum space. When a block leaves or enters active drum space, its addressing data leaves or enters the hash table. The file phantom accomplishes such movements of blocks and addressing data. A file remains active if it is used relatively frequently; in such a case its block list remains in the file phantom. When a file lags in activity, all of its blocks are moved to dectape and its block list is moved to the block list file.

## Computation Scheduling

### Changing Users

Suppose user $\alpha$ is running, and his quantum expires. Then the scheduler is run (immediately). If there is no one else to run, user $\alpha$ is then granted an infinite quantum. (Infinite quanta are revoked on completion of any I/O function.) However, if there is another user (say, user $\beta$) to run, we start to bring user $\beta$ into the machine, at an expected cost of x core swaps, as follows:

First, as soon as we have decided to run user $\beta$, $\alpha$'s processes may no longer cause drum activity. So, if an $\alpha$-process requests a drum transfer or refers to a part of $\alpha$'s program image which is on the drum, that process is removed from the process list (but remains active).

Then we wait for some physical core module to become a non-target of drum activity. [While waiting, we continue to run those processes on the process list.] Suppose core j is the first free module. Then we remove all processes which were running in core j from the process list and swap 4K of user $\beta$'s program image into core j. Finally, we add to the process list those $\beta$-processes which can run in physical core j.

The above procedure is continued until all 4K sections of $\beta$'s program image which contain active processes, plus the $0^{th}$ 4K section (which contains $\beta$'s C-list), are in core storage. The first 4K section to be brought in (if necessary) is always the $0^{th}$ section.

It may be that after we have one or two program image sections of $\beta$ in core, a $\beta$-process will refer to a section of $\beta$'s program image which is still on the drum. If this happens, that section is the next section to be swapped in. All processes in that section are then added to the process list, of course. Other sections of $\beta$'s program image, which contain active processes, will all be swapped in eventually, but they must wait for the section needed as data.

User β's quantum commences as soon as we have brought x of β's 4K program image sections into core storage, or as soon as all program image section holding active processes have been brought in, and 15 ms has passed; whichever comes first. User β has already done a great deal of computation, but his quantum begins only when a meaningful amount of his program image is available in core. Note that a 4K section of β's program image which is never referred to is not brought into core. This minimizes the time needed to swap β in and out.

The expected cost of running user α again is calculated when we decide to run user β: it is simply the number of physical core modules α has used during his quantum. This number is used by the scheduler to determine a hierarchy of computations. The hierarchical structure of {all computations} is considered in deciding who to activate when β's quantum is over.

Note that processes in α which do not use the drum may still be running while β is being brought into core and run. During β's quantum, α-processes (or any non-β, non-executive processes) will be run only if all of β's processes are waiting for I/O completion and/or drum transfers. But as soon as we decide to run some new user γ, all available active processes are again run.

Reactivation on I/O Completion

Users who are being restarted because some process' I/O function just completed get special treatment. They get very prompt service, but it is limited service.

Suppose α is running, and his quantum expires, and β is to be run next, because some process of β is being restarted after I/O wait. Then not all of β's program image is to be brought into core. Only {$0^{th}$ program image section} ∪ {program image section holding the restarted process} is brought into core, with the expected cost set to 1 or 2, depending on where the restarted

process is running. Other program image sections are brought into core only if referred to. [Of course, once a program image section is in core, all processes there are run. But not all of β's processes will be restarted on I/O reactivation.]

Furthermore, β is granted a relatively short quantum. In this way, user programs can handle I/O quickly, but long computations triggered by some I/O completion are finished on activation at a lower priority level.

The Scheduling Algorithm

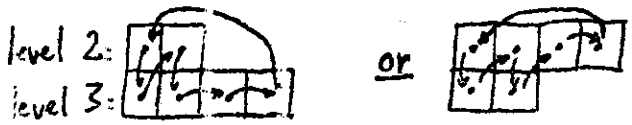
The hierarchy maintained by the scheduler is the following:

level 1: computations with just reactivated processes

level 2: computations with expected swap costs of 1 or 2

level 3: computations with expected swap costsof 3 or 4

Any users on level 1 are run as described above, with a granted quantum of 80ms. Users on levels 2 and 3 are organized into a round-robin. If there is nothing on level 1, we run levels 2 and 3, trying to alternate level 2 users with level 3 users, as shown:




Users on levels 2 and 3 are granted quanta of 200,180, 160, and 140 ms. for expected swap costs of 1,2,3, and 4 respectively.

If a section of a particular user's program image is in core and does not require a swap, the expected cost of running him is not changed. Such a coincidence simply helps to reduce system overhead------and is not probable under a normal (relatively saturated) load.

A user's quantum is now just the length of time during which his processes receive the full attention of the processor. In general, the user is running before and after his quantum, as well as during.

## Scheduling Data Structure

### General Information

The I/O completion action table tells who to restart on completion of any I/O function. Both the process to be restarted and its computation are named in the table.

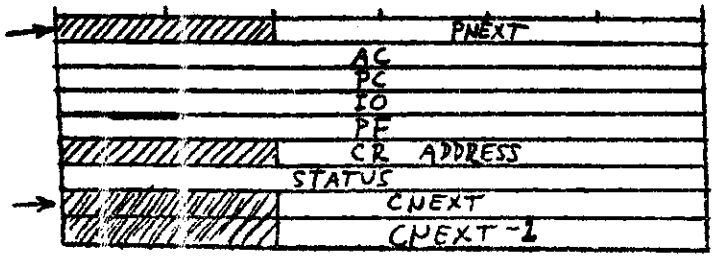
The drum mover keeps the four-word target count table, which tells how many jobs in DJQ have core j (j= 0,1,2,3) as target. There is also a target permit table, which the drum mover uses to decide whether a particular drum request is allowable. In the case where the request is not allowable, the drum mover removes the requesting process from the process list.

The scheduler keeps the four-word occupancy table which contains pointers to the computations with program image sections in core j(j=0,1,2,3).

### Structure of the Process List

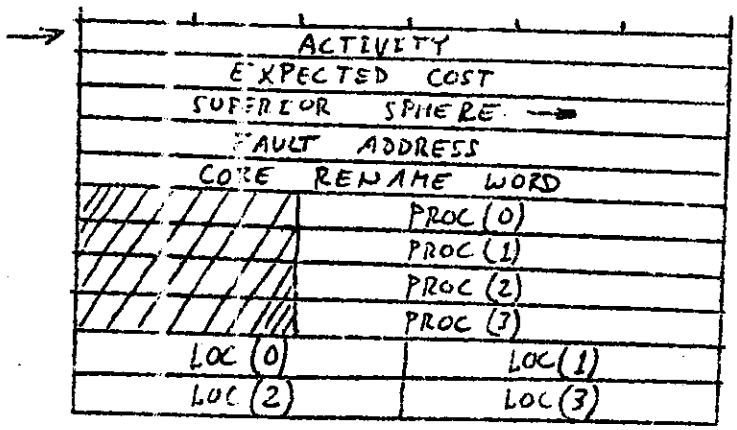
The process list is a list of active processes for which, as far as is known, sufficient data exists in core to successfully run the process for a short time. The process list is organized into eight queues of processes: after a process has been run the proper length of time for its queue, it is moved into the next lower queue. The queue quanta are 2,2,4,4,8,8,16 and 16 ms. The queue priority table is a $20_8$-word table of pointers to the various process list queues (heads and tails of the lists of processes in the same queue).

A process entity is the representation of a process in the executive. The format is:



PNEXT is a pointer to the next process in this queue or list. CR ADDRESS is the location of the core rename word for the computation. AC,PC,IO, and PF are the last ac,pc,io, and pf. CNEXT and $CNEXT^{-1}$ are pointers along the list of processes in the computation's program image section k, where k is the extension of the pc.

A computation entity is the representation of a computation in the executive. The format is:




LOC(K) cells where program image section K of this computation is. This may be a drum field name or a physical core name. PROC(K) is a pointer to the list of all processes of the computation which are in program image section k.

## Process Scheduling

### Hardware

The PDP-1 has a 12-bit process register, a 3-bit priority counter, a 3-bit queue priority register and associated infinite process quantum flop. There is a clock which decrements the priority counter every 2 ms.

The following instructions manipulate these registers:

| | |
|---|---|
| load process register | ($IO_{6-17} \rightarrow PR$) |
| read process register | ($PR \rightarrow IO_{6-17}$; $0 \rightarrow IO_{0-5}$) |
| add process register | ($AC+PR \rightarrow AC$) |
| load queue priority | ($IO_{16,17} \rightarrow QP$; $IO_{15} \rightarrow IPQ$) |
| start count | ($7 \rightarrow$ Pr.C.; reset 2 ms clock) |
| start user | (load state word and leave executive mode) |

The state word for the start user instruction is taken from the process entity addressed by the process register.

When an interrupt or trap occurs, the state word (ac,pc,io,pf, but not cr) is stored in the process entity addressed by the process register and control passes to the executive location peculiar to that interrupt or trap.

### Changing Processes

A "quantum-expired" interrupt occurs whenver the number of 1's preceeding the first 0 (from the left) in the priority counter becomes less than the contents of the queue priority register, or after the priority counter reaches zero and the IPQ flop is off, and the clock ticks again.

The queue priority register holds, at all times, the left two bits of the number of the highest occupied level of the queue priority table. Thus the currently running process will continue to run until its quantum expires (there is an equally deserving process), or until a more deserving process appears in the queue priority table.

If a more deserving process appears, it will be run immediately and the previously running process will remain in its queue. If a process runs until

a "quantum-expired" trap it will be moved to the end of the next (lower) occupied queue, and the first process in the highest occupied queue will then be run. The I/O flop is turned on only when there is exactly one process on the process list.

## Conclusion

The essential framework of the File System and Scheduling Algorithm of a PDP-1 time-sharing executive **been described. We have chosen not** to describe such things as typewriter buffering, character translation, debugging software conventions, or allocation procedures for computing objects other than file space. These particular aspects of the present system will be carried over to the new system, to a large extent. We have also not described in great detail the actual coding of the File System and Scheduling Algorithm; this is intentional. Implementation of most meta-instructions in [*] will be through coding in a phantom user (now called the extended executive) which we will call the **meta phantom.** Some meta-instructions (such as lock and lock) will be implemented in the executive, to provide fast operation. For a complete list of available meta-instructions, see [*] and the first section of **this paper.** Everything applicable will be implemented (and then some.).