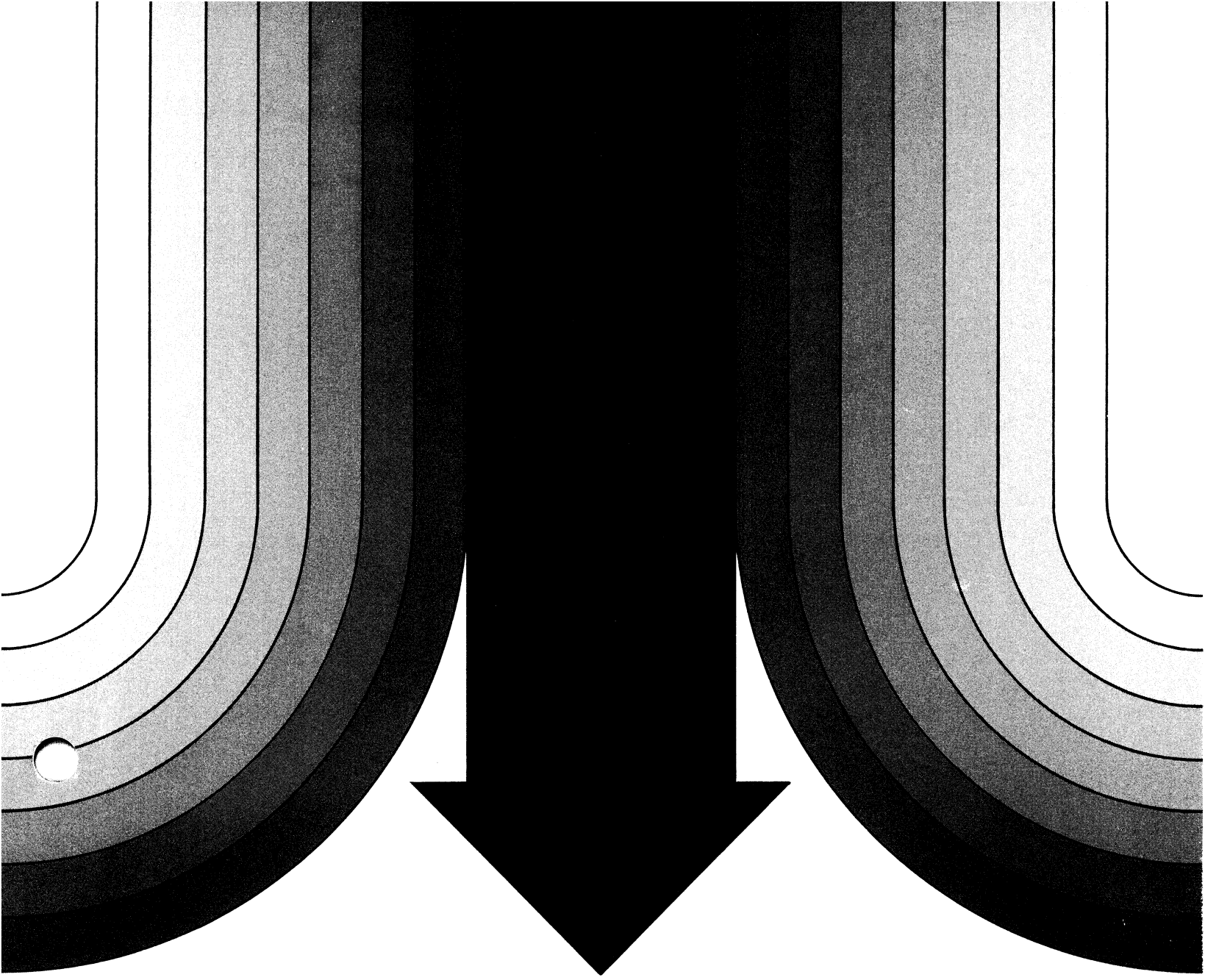


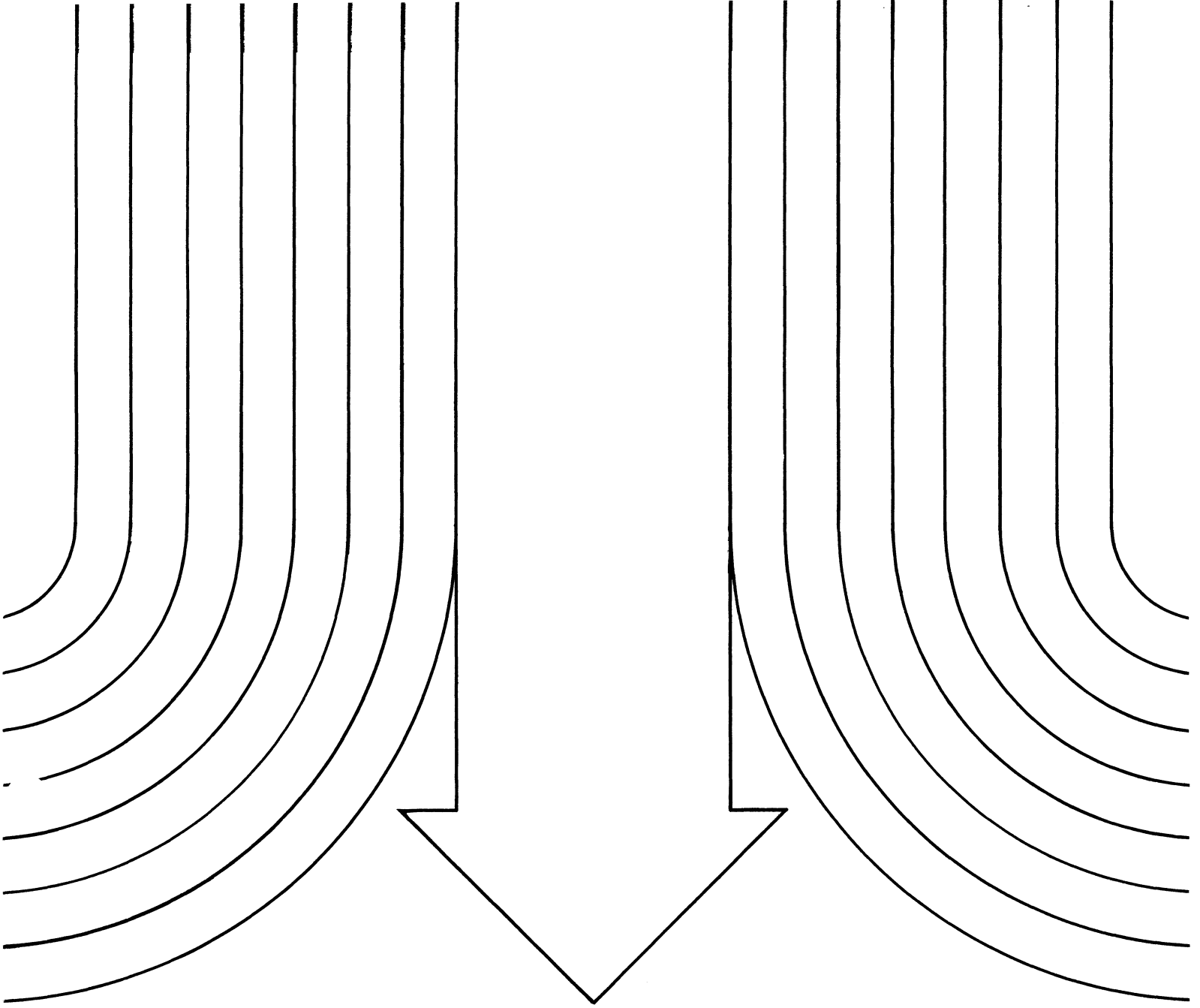
AOS/VS Internals CPU Management-The Scheduler

Data General

educational services

053-001011

NOTICE



AOS/VS Internals CPU Management-The Scheduler

DataGeneral

educational services

053-001011

Table of Contents

Introduction to the AOS/VS Scheduler

Paths and Time is the area of the operating system that schedules the system and user paths and manages the timing devices used by the system.

Paths are sequences of code. An example of a Path is part of a program written to do a matrix multiplication; the path would be the main program loop. Or a path would be one of the subroutines that does one traversal through the matrix. Another example of a path is the Scheduler in AOS/VS.

Time to the operating system is what is defined by the timing devices. These devices can be compared to a wall clock or stop watch. These devices are used to make decisions about resource use by a certain path. Like a wall clock, the shortest amount of time is called a tick.

There are three major types of paths dealing with Paths and Time. First there is the user path, which is called a process and is represented by a Process Table (PTBL). All user-visible paths are user paths. The second type of path is a system path, which works in the system on behalf of a user or is used to manage the system resources. These types of system paths are called Control Blocks (CBs). User CBs do work for the user and System CBs do system resource management. The last type of path is a system path, which is not part of a CB or a PTBL. The SCHEDULER is such a path. These paths are used to either dispatch CBs or PTBLs, or they handle special events such as interrupts that affect a path on the system, or they provide accounting services for CBs, and users.

Paths and Time dispatches system paths to allow the other major components to do system management. This ties all of AOS/VS together. Thus dispatching and accounting services are Paths and Time's connection to the other components in the system.

Below is an illustration of the components of AOS/VS and what type of path is used in that component. This will be done by using PTBL to represent a user path. User CB, System CB, and non-CB system paths will be used to represent the system paths. Remember that the point of view here is from the kernel in ring 0.

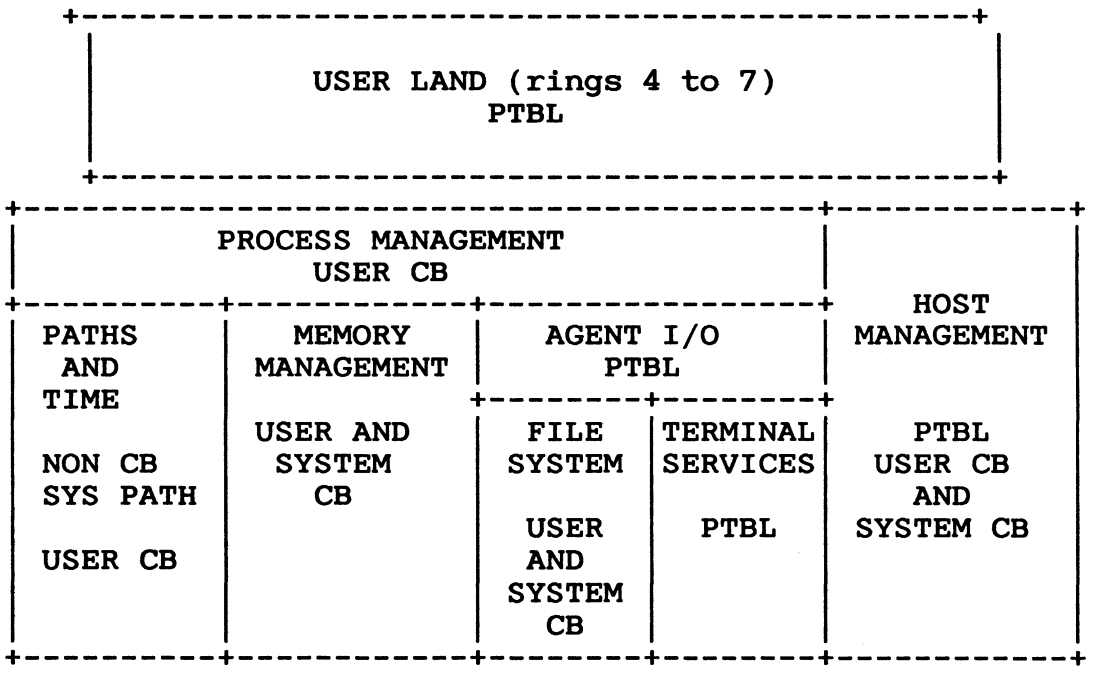


Figure 1 AOS/VS Components

There are seven major components to Paths and Time: Time Management, Job Processor Managment, Logical Processor Management, Class Management, ELQUE Management, CB Management, and PTBL Management. These components are arranged in the figure below from bottom up. The more knowledge the component has of the hardware, the lower its placement in the picture.

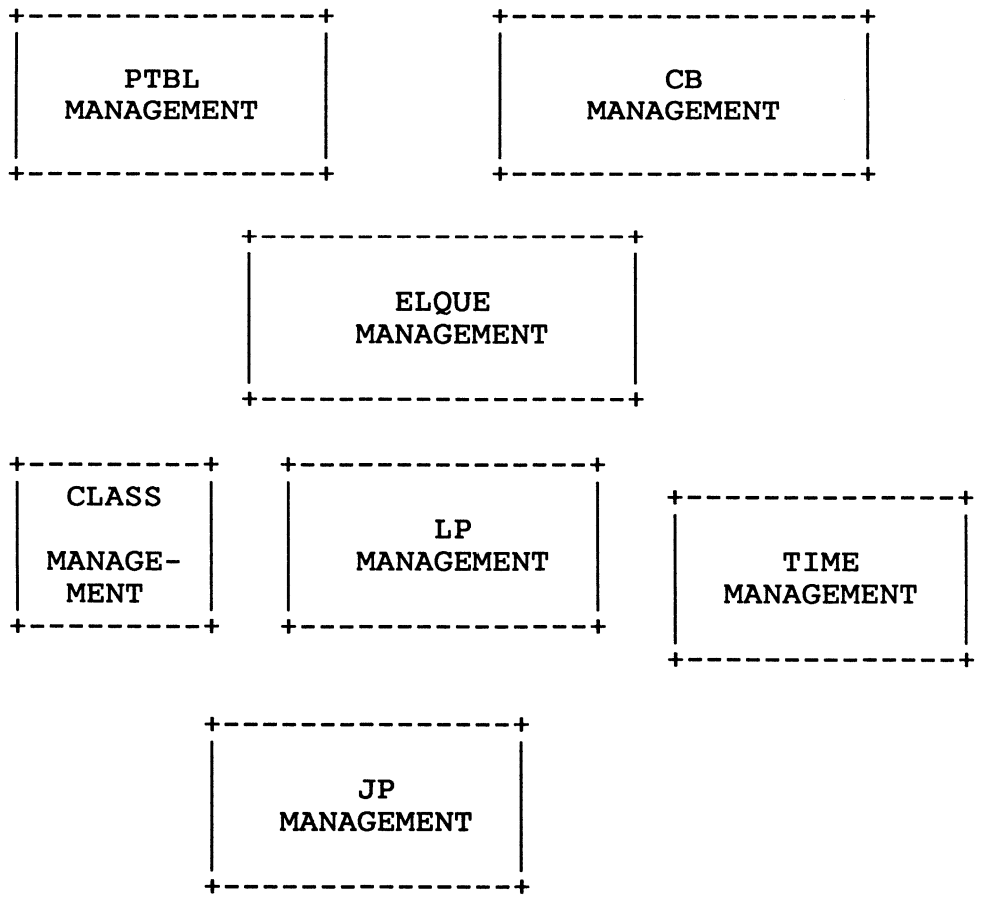


Figure 2 Paths and Time Components

Each component manages its own databases and provides services for the higher components in the system. Each component has distinct differences from the others and is considered a separate chapter, although the code is not actually broken down this way.

Each chapter in this book is arranged as follows.

I. Introduction
II. Objects managed
    A. Define objects
    B. Basic operations on objects
    C. Paths that use the basic operations
III. User services
IV. System services

- The system services show how a component of Paths and Time services other upper-level functions or provide services for the rest of AOS/VS.

- JP management manages the processors. The main service it provides to the other components is that it allows the other components to run code.

- LP management manages the user-visible representation of a processor. The service it provides is keeping class scheduling statistics and updating the class part of the scan mask used by the scheduler.

- Class management manages the class-specific databases. These databases are used by the PROC code (see Process Management Volume) to assign a class to a process. These databases are also accessed by the LP management code to assure existence of a class.

- Time management manages the time devices that provide data used by other components to manage or account for the time they use.

- ELQUE management manages the eligible queue and the other major scheduling queues. The services this component provides is dispatching the system paths (CBs) and the user path (PTBLs). This dispatching transfers control of the ability to run code to the different paths.

- Process management interface services provide user path management such as task scheduling, a system call interface, and user time usage. This component sets up the CBs for CB management.

- CB management dispatches the different types of system paths in the system. These paths are called CBs. There are two types of CBs: system CBs and user CBs. User CBs are created when a user makes a system call. System CBs do functions necessary to manage the system resources such as memory.

The figure below shows how these components interrelate.

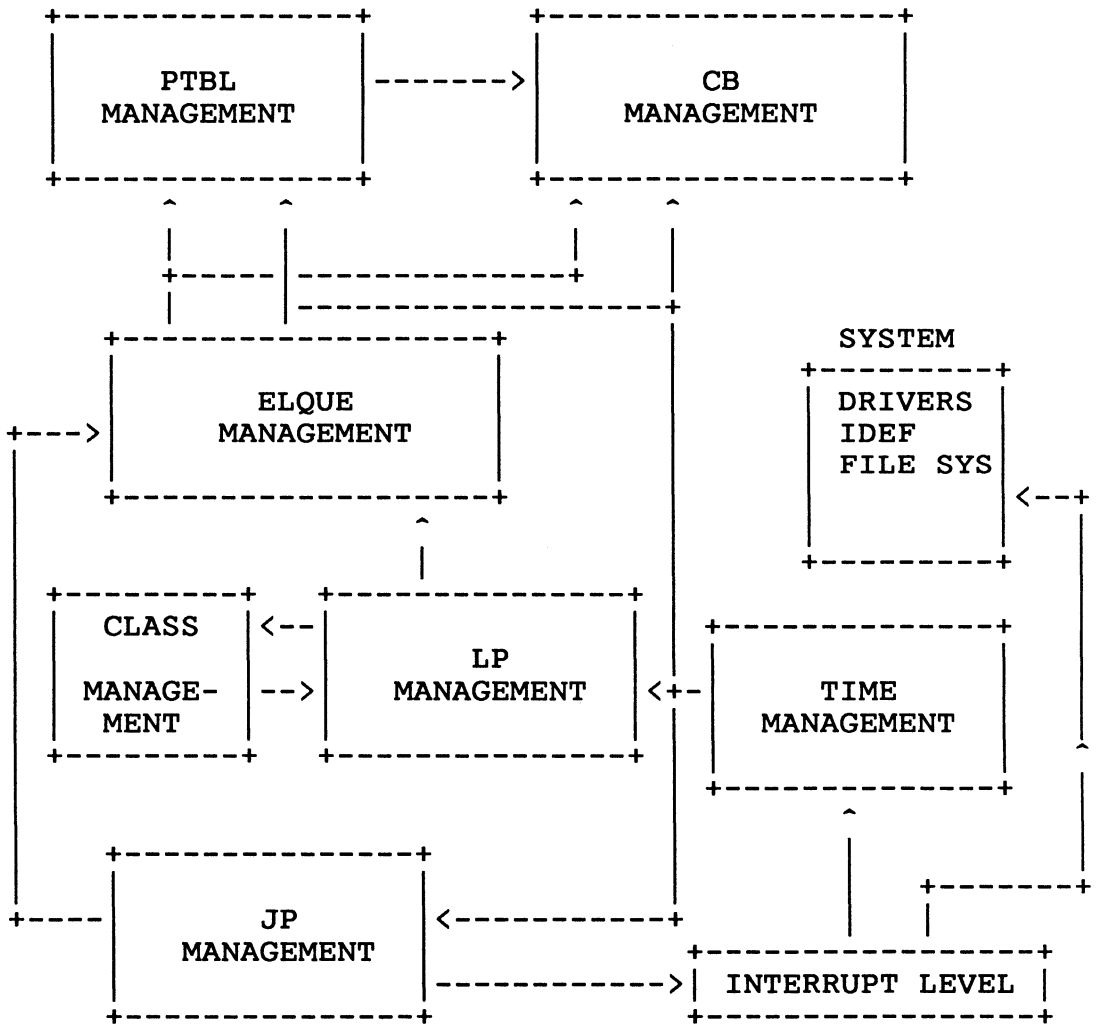


Figure 3 Paths and Time Component Relationships

Throughout the Paths and Time section, references will be made to the PTBL and CB databases. Each section that works substantially with the databases will describe sections of the PTBL and CB databases that are relevent to those sections. They will not, therefore, describe the whole database.

The table below (Table 1) shows the entire PTBL. Part of the PTBL is the PTBL/CB common area. This table is also shown in the PROCESS Management volume.

Process Table Offsets

The following is a field-by-field description of the main database for each process on the system. It is called a "Process Table." This database is built when a process is proc'd. It contains all the information needed by a process to be run by the operating system.

Each process is linked in priority order to each other when they are on any given scheduler queue. This link is established through the first four words of the process table. Process tables and Control Blocks share the first 24 16-bit words in common. This is so that they can both reside on the ELQUE and be scanned and scheduled in the scheduler.

The first column is a code that points to the section of the Internals manual where the full description of the field can be found. Codes are as follows:

S - Paths/Time
F - File system
P - Process management
M - Memory management
* - Look at individual bit definitions

Table 1 The Process Table

	Variable	offset		Lock and Meaning
P	PLNK.W	0	T	Forward Link
S	PBLNK.W	2	T	Backward Link
S	PNQF	4	L	Priority eNQue Factor
S	PCLASS	5	L	Process CLASS
S	PSTAT	6	RE	Process Status Bits for use during scheduler scanning
S	PSTAT1	7	RE	Process status bits which are not needed during scheduler scanning
S	PPC.W	10	I	Control address when scheduled
S	PLPCB.W	12	L	Logical Process Control Block to charge time used by this process
S	PGNUM.W	14	L	(CLASS #)*2 integer used by XWADD
S	PTIM.W	16	L	Current interval of time expended by a direct/indirect system call
S	TTIME.W	20	L	Accumulated time expended by a direct/indirect system call
S	CALLN.W	22	L	(SYS CALL NUMBER)*2 used during SYS CALL time accounting by XWADD
S	PKEY.W	24	X	UNPEND KEY
S	MAPFLG	26	N	MAPCON needs to be done for CP
S	PURFLG	27	N	Purge ATU needs to be done for CP

End of Common area between CB and PTBL

	Variable	offset	Lock and Meaning
P	PSELF.W	30	I Address of PTBL
S	PULOC	32	L Current User Locality
S	PPLOC	33	L Program Locality
S	PLLOC	34	L Legal localities
S	PCLAS1	35	L Copy of PCLASS for restore
S	LMAST	36	RE 'MOTHER-ONLY' Status word
P	PDAD.W	37	I PTBL Address of Father
P	PSONP.W	41	P Son pointer to son list
P	PSONL.W	43	P Son Link Word for Father's Son List
*	PFLAG	45	RE Flag word 1
*	PFLG2	46	RE Flag word 2
*	PFLG3	47	RE Flag word 3
*	PFLG4	50	RE Flag word 4
P	PEXTN.W	51	I PTBL Extender address
P	PIORR.W	53	L Blocked Receive Request chain
P	PIORB.W	55	L Backward Link
P	PIPCS.W	57	I Spool file
P	PRPRV	61	I PRIV bits assigned by Creator
P	PID	62	I PID assigned at create time
P	PERPC.W	63	L PC when RETER called
P	PCMLK.W	65	T Core Manager ENQUE Link
S	PSLEX	67	L Time Slice Exponent
P	PCRMX	70	I Max # SON procs
M	PMKEY	71	L MEM WAIT flag key
P	PPRI	72	L Father assigned pri factor
S	PDINH.W	73	? Delay current INC word
S	PDLNK.W	75	? Delay chain forward link
S	PDBLK.W	77	? Delay chain backward link
P	PCONH.W	101	I Console Port Number
P	PCONH	101	I Console Port Number (HI)
P	PCONL	102	I Console Port Number (LO)
P	PHASH	103	I Process name hash value
P	PTUP.W	104	T TCB signal bit map on SWAPIN
S	PINSU	106	L In scheduler mode flag (0 OR 1)
S	PSSEL	107	L # of subslices used since put on ELQUE
P	PKCHR	110	T ?KWAIT char
P	SWCCB.W	111	I SWAP File CCB for this process
P	PGCCB.W	113	I PAGE File CCB for this process
M	PWSET	115	L Working set size (# Phys pages)
M	PWSSH	116	L # Shared pages in WS
P	PERWD	117	L Error word
P	PSWPO	120	? # Times swapped out
P	PSWID	121	? SWAP unique ID
P	PTRGC	122	L Target call counter
P	PSRNG	123	L Server Ring Bit map (bits 0-7)

	Variable	offset	Lock and Meaning
P	PSFDF.W	124	L Fwd Link Spool File Dir Chain
P	PSFDB.W	126	L BKWD Link Spool File Dir Chain
P	PSFRC.W	130	L Spool File Entry Count per Ring
S	PSIDIR	134	L # Enqueued TCBS W/Indirect calls
*	PFLG5	135	RE Flag word 5 (added at end to keep offsets the same)
P	PRNGTP.W	136	T Ring type - New or Old
S	PSTATE	140	T VS/MP Process State Word
P	PMXPR	141	L Maximum Process Priority
P	PSWPSIZE	142	L Usable Swapfile space (WS pages) [Invariant for life of process]
P	PDIPC	143	N # of TCB's pended on ?IREC
P	PRCCB.W	144	L Break File CCB address
F	PTUNL.W	146	? Bit Mask of tasks to unlock (FLOCK)
F	PLCNT	150	? Count of active ?FLOCKS
S	PCBLK	151	? PLOCK fail counter

PLN 152 Length of PTBL

Process Table bit parameters for first status word (PSTAT). Note that this is the status word where bits are defined that will be looked at during the ELQUE dispatch scan.

Status Bits

Status bits have common meaning for both CBs and PTBLs.

PSRDY	100000	Not ready to run
PSRUN	040000	Running
PSEW	020000	Sched action
PSNCB	010000	Don't look - process can only use a CB

The following status bits are used in SCHED as a dispatch value. Highest priority function has lowest bit assignment.

PSBRK	004000	OP interrupt
PSBAG	002000	SWAP OUT process
PSBLK	001000	BLOCK process
PSDP	000400	START UP DAEMON
PSMWT	000200	Wait for memory key to change
PSTSU	000100	Timeslice is up

The following bits are not used via DSPA:

PMAST	000040	Mother-only element (1=Mother-only)
PLCK	000020	Process table lock bit
PSETR	000010	Don't enter
PSFSY	000004	System page fault
PTRAN	000002	Element transition bit

-- A critical state/counter being viewed/changed

Process table bit parameters for second status word (PSTAT1). Note that this is the status word where bits are put that will not be looked at during the ELQUE dispatch scan.

PSYST	000001	System CB bit -- CM,SM,DM
PTYP	040000	Element type (1=Control Block)
PNAD	020000	No address space to release(BPLCK)
PNFST	010000	Sys call did not run 'FAST'

MULTIPLE BIT DEFINITIONS

PFPTA	000000	Process type A
PFPTB	000002	Process type B
PFPTC	000003	Process type C

Bits in PFLAG

PFNIN	100000	^C^A interrupts will wait
PFPRE	40000	Preemptive Resident
PFDEB	20000	DEB entry
PFFIR	10000	First execution and load
PFELG	04000	Process is ELIGIBLE (in core)
PFTRP	02000	Trap bit
PFINT	01000	Run ^C^A DAEMON
PFBRP	00400	Breakfile (^C^E) requested
PFTRM	00200	Run TERM DAEMON
PFMBL	00100	Proc can only be explicitly UNBLOCKED
PFILT	00040	Prot trap at interrupt level
PFRSH	00020	Resched flag
PFWSC	00010	Page added to WS since last PFF
PFNFR	00004	Narrow process becoming non-resident
PFSWP	00002	Swapping task
PFEBL	00001	Waiting for son termination

Bits in PFLG2

PFSP	100000	SWAP OUT/IN in progress
PFSU	40000	UNPEND someone waiting for SWAP IN/OUT
PFCMQ	20000	Process is enqueued to CM
PFDUP	10000	UNPEND TCB at head of delat chain
PFIEB	04000	PTBL on IEBLK queue
PFUCF	02000	UNPEND waiters when cleanup finishes
PFOIQ	01000	PTBL is on IEQUE
PFBLE	00400	Scheduler can block process
PFWSL	00200	Waiting on .SGNL
PFSWO	00100	Process is being swapped out
PFCIP	00040	Cleanup in progress
PFATL	00020	Process termed by system
PFSUP	00010	Superuser mode
PFNTR	00004	Narrow process becoming resident
PFWSG	00002	Process is waiting for ?SIGNAL
PFQSC	00001	Inhibit scan of backed up TCB request

Bits in PFLG3

PFTSE	100000	At least 1 time slice ended
PFUBD	40000	Unblock FATHER on RTN
PFPTM	20000	Processing a TERM
PFPCN	10000	Processing a CHAIN
PFCIE	04000	Has created an IPC type entry
PFIWC	02000	Interrupt (^C^B,^C^E) term of PROC
PFIRS	01000	Int world interrupted task
PFATC	00400	AGENT term work completed
PFSTM	00200	Process self-termination
PFDIN	00100	Delayed ^C^A waiting
PFATT	00040	AGENT term task is running
PFMBQ	00020	Process on MBLKQ
PFARBS	00010	All ref bits set in working set
PFPCH	00004	Hold on >1 parallel call
PFMGR	00002	SYSMGR mode
PFOPCH	00001	Parallel call overwrite(TARGET)

Bits in PFLG4

PFBRK	100000	Process wants breakfile on trap
PFDIS	40000	Disconnect of modem occurred
PFTBS	20000	Terminated by superior process
PFSPR	10000	Superprocess mode
PFMRL	04000	MAX CPU limit in use
PFPBS	01000	Block after initial load
PFOBQ	00400	Process is on blocked queue
PFACL	00200	User default ACL enabled
PFSRV	00100	Process is a server
PFMDP	00040	Process wants MDUMP on trap
PFNRO	00020	Process is narrow (16 BIT)
PFDP1	00010	Pass MDUMP flag to AGENT on term
PFKWB	00004	Process has task doing a ?KWAIT
PFKIB	00002	Process has all ^C^X disabled
PFISS	00001	INT SEQ received

Bits in PFLG5

PFENB	20000	Process is the target of an ?ENBRK
PFDWS	10000	System default working set limits
PFTDC	04000	TERM if FATHER chains
PEXTIV	02000	PTABLE Extender invalid(SWAPped out)
PLWAIT	01000	Address space waiter bit
PTWAIT	00400	Waiter bit for target count(PTRGC)=0
POPER	00200	Process is a global operator
PFREQ	00100	At least one task of this process has request block memory of ?OPER database
PFXPT	00002	Extended (POST REV6) program type
PFHRP	00001	High range PID

Chapter 1
ELQUE Management

ELQUE management is the part of Paths and Time that manages the major scheduling queues. The queues that will be discussed in this chapter are: ELQUE, PELEMQ and HANDQ.

- ELEMENTs are the items on the scheduling queues. There are two types of elements: PTBLs and CBs. Within CBs there are system CBs, user CBs, and DAEMONS.

- A PTBL is the system representation of a process (user path) in the system. The PTBL holds information necessary to run the user's code paths and keep user statistics.

- A CB is used to either run a system call or perform a system service. The CBs used for system calls are called user CBs. CBs used to perform system functions are called system CBs.

- A DAEMON is a special kind of CB that performs a function for the system on behalf of a user, i.e., such as process termination. This kind of CB runs at very high priority.

Figure 1.1 shows the connections between ELQUE management and Paths and Time.

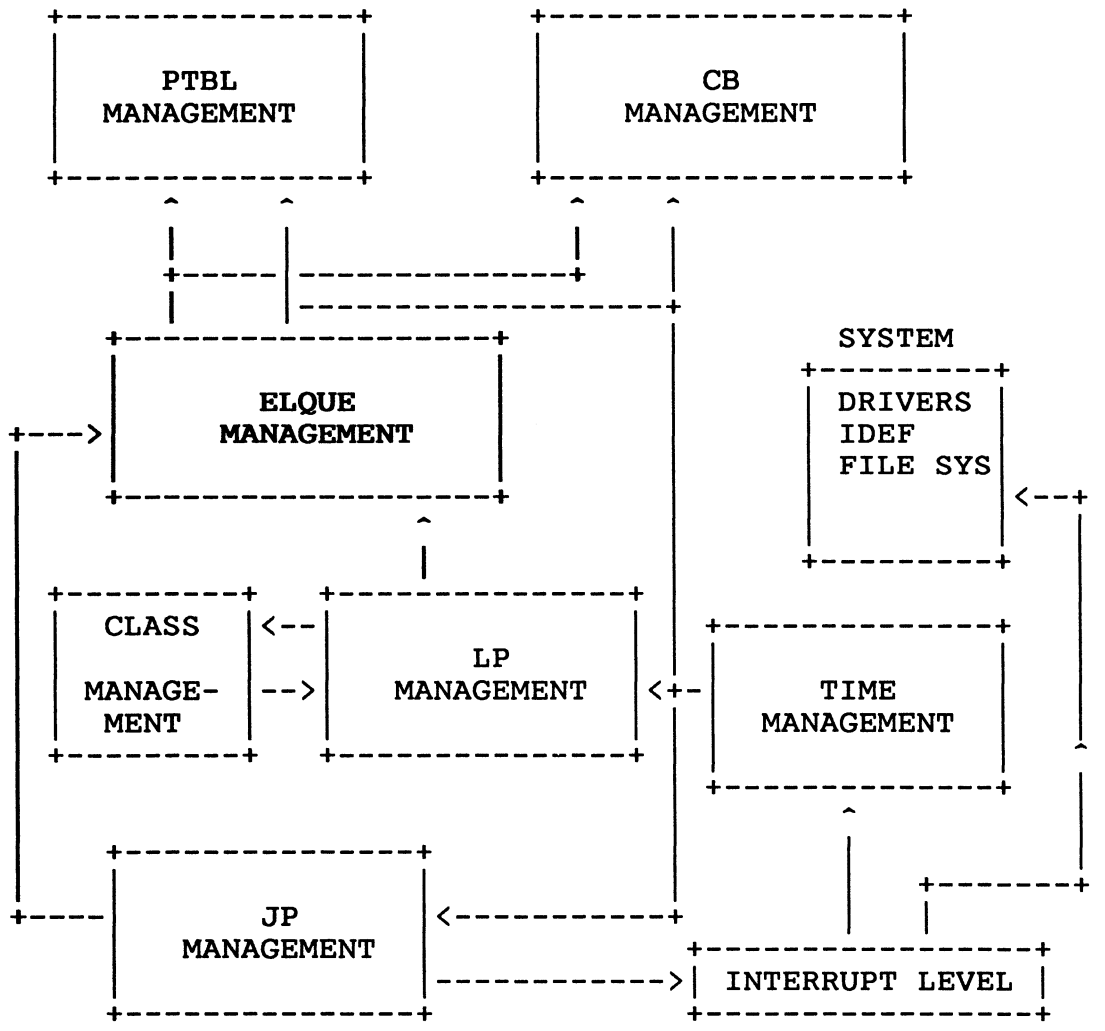


Figure 1.1

ELQUE Management interacts with four other parts of paths and time: JP, LP, CB, and PTBL management. ELQUE management runs code on the JP and dispatches elements that it selects to PTBL or CB management. ELQUE management gets the Class scan mask from LP management that it uses to select an element.

This chapter is organized as follows:

- The objects
- The operations that work on the objects
- The internal paths of ELQUE Management
- Service provided to the rest of AOS/VS

1.1 Objects

This section describes the queueing structures in ELQUE Management: ELQUE, PELEMQ, and HANDQ.

1.1.1 The QUEUE Structure

AOS/VS manages system resource users by way of the Queue. When a user is on a certain Queue, then that queue reflects the state of the user. For example, if a process is on the ELQUE then the process is in an ELIGIBLE state. Each Queue described in this chapter is a doubly linked list of elements. The first and last elements of a queue are pointed to by the header block. The header block is a structure that has the following offsets.

0	QHEAD	Contains the pointer to the head
2	QTAIL	Pointer to the tail of the Queue
4	QSTATUS	Qstatus word
5	QUSERS	Queue users count
6	QSCAN	Queue scanners count(ELQUE ONLY)

Figure 1.2 Queue Header Block

QHEAD holds the pointer to the first element in the queue. For ELQUE this value will always point to the Disk manager (DMTSK).

QTAIL holds the pointer to the tail of the queue. On ELQUE this will always contain the pointer to the root process table.

QSTATUS contains the status bits for this queue. Currently there is one bit defined for this one word entry and that is the Qlock bit. This bit is used as a spin lock. (See JP management.) This bit is used to lock the queue in order to allow modification of the queue or to increment the QSCAN counter.

QUSERS is a 16-bit counter that reflects the number of elements attached to the queue.

QSCAN is a counter of all the scanners of the queue. This is only used for ELQUE. This counter is used to arbitrate shared and exclusive access to a Queue by the queueing routines. The usage of the scan count is as follows:

> When the scanner wishes to scan ELQUE it tries to first get the QLOCK (a spin lock). If successful, the scan count is incremented and then the QULCK is released. After the scan completes, the scanner decrements the scan count without locking ELQUE.
>
> When a path is trying to modify ELQUE, the path first tries to get the QLOCK. When the path gets the lock, it then spins on the scan count. When the count goes to zero the queue can be modified. After the queue is modified then the QLOCK is released.

There are four major queues used in ELQUE management. They are the ELigible QUEue (ELQUE), The HANDler Queue (HANDQ), the Pended ELEMent Queue (PELEMQ), and the IDle System Control Block Queue (IDSCBQ).

1.1.1.1 ELQUE

ELQUE is the queue of eligible elements. This queue is used by the scanner to find an element to dispatch. ELQUE contains PTBLs and CBs. To access ELQUE one or more of the ELQUE locks must be held.

Except for a few cases, ELQUE is ordered by element type and Priority ENQue Factor (PNQF). This ordering allows the scanner to get the highest ready-to-run element first. Figure 1.3 shows the ordering of the Eligible queue.

1.1.1.2 ELQUE Ordering

ELQUE: This is the eligible queue of process tables and control blocks. This is the primary queue used by the scheduler.

* The DISK MANAGER Control Block is always first on this queue. (This CB is always on ELQUE.)

* The CORE MANAGER Control Block is always second or third on this queue if not idle.

* The SYSTEM MANAGER Control block is always second or third if not idle and the Core Manager is on ELQUE.

* The active Group 1 Control Blocks are next. These are in FIFO order by time.

* The group 1 process tables are next. These are in order based on a PNQF. These include the PMGR process table which is permanently on the ELQUE.

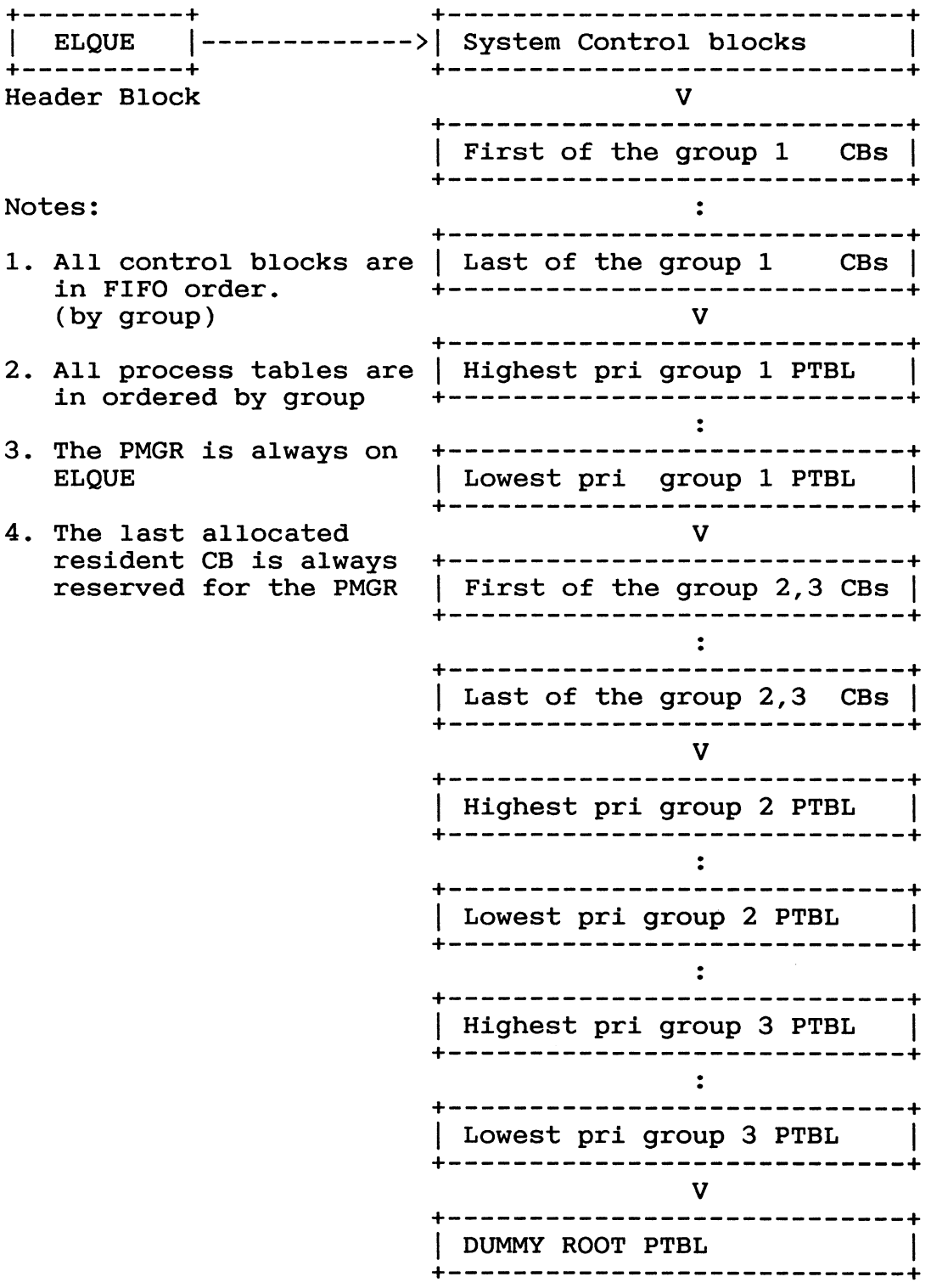
* Next comes the Group 2 Control Blocks. These are in FIFO by time.

* The group 2 process tables are next, ordered by PNQF.

* The group 3 process tables are next, ordered by PNAF.

* Last on queue is a dummy process table, called the Root Process Table. This never requires time, but is used to mark the end of the ELQUE. The root process table has a PID of 0, and is considered the father of the PMGR and OP:CLI processes.

Figure 1.3 shows the ordering of ELQUE.



Notes:

1. All control blocks are in FIFO order. (by group)
2. All process tables are in ordered by group
3. The PMGR is always on ELQUE
4. The last allocated resident CB is always reserved for the PMGR

Figure 1.3

1.1.2 Process Scheduling

AOS/VS schedules eligible processes based on their priority numbers and scheduling characteristic; the range of process priority numbers (1 through 511) spans three scheduling groups.

Group 1 ranges from 1 to a number, "G1", which is set during VSGEN. AOS/VS schedules any process whose priority number places it in Group 1 on a round-robin basis. Under this scheme, each process is allocated a uniform slice of time during which it may execute. Once a process of a specified priority temporarily stops executing (having used up its time slice), it is not chosen to execute again until all other processes of that priority have been chosen to execute.

Group 2 ranges from G1+1 to a number, "G2", which is also set during VSGEN. AOS/VS schedules any Group 2 process heuristically, which means that the system takes the process' past behavior into account when allotting it an interval of time during which it may execute.

Group 3 ranges from G2+1 to 511. AOS/VS handles processes in this group on a round-robin basis.

NOTE: If you need to maintain compatibility with AOS, G1 and G2 must be set to 255 and 258, respectively.

Group 1 processes are always more important (that is, more likely to be chosen for execution) than those in Group 2 or 3, and Group 2 processes are always more important than those in Group 3. Within each group, the lower the priority number, the greater the importance of the process.

If an executing process cannot proceed, you can issue the ?RESCHED system call, which allows the calling process to give up control of the CPU and forces AOS/VS to immediately schedule another process for execution.

1.1.2.1 Priority Numbers

Eligible processes are placed on ELQUE partly based on their individual priority numbers. AOS/VS uses priority numbers to determine each process' priority. When you create a process, you may assign it a priority number.

Priority numbers range from 1 (the highest priority) through 511 (the lowest). These numbers span three scheduling groups (with no overlap and no gaps), whose boundaries are determined during VSGEN.

1.1.2.2 Priority Changes

If a process wants to change its own priority, and it has change priority privilege, it may issue the ?PRIPR system call. To change the priority of another process, however, the calling process must be in Superprocess mode.

1.1.2.3 Changing Type

The priority of a process may also change when you change its type with either ?CTYPE or ?PROC. Given that the boundaries of the 3 scheduling groups are:

```
Group 1 = 1 - G1
Group 2 = G1+1 - G2
Group 3 = G2+1 - 511
```

then the following tables summarize the changes in priority that occur when a process changes type. Notice that a swappable process can never assume a priority of 1, 2, or 3, but it may APPEAR to do so because of the way priority numbers get mapped. (See mapping in the next section.)

Priority Changes Going from a
Resident or Preemptible to Swappable Type

Priority Before Change	Priority After Change
1 - 3	1 - 3 * **
4 - G1	2 * **
G1+1 - G1+3	1 - 3 **
G1+4 - G2	G1+4 - G2
G2+1 - 511	G2+1 - 511

* This parallels what happens under AOS.

** Although you would see these numbers if you displayed the priority of a process with the CLI PRIORITY command, the actual priorities would be G1+1 - G1+3. See "Mapping" below.

Priority Changes Going from a
Swappable to a Resident or Preemptible Type

Priority Before Change	Priority After Change
1 - 3 **	1 - 3 *
4 - G1	4 - G1
G1+4 - 511	G1+4 - 511

* This parallels what happens under AOS.

** Although you would see these numbers if you displayed the priority of a process with the CLI PRIORITY command, the actual priorities would be G1+1 - G1+3.

1.1.2.4 Priority Mapping

A resident or preemptible process can assume any of the priority numbers 1 through 511. The system uses this number in gauging the importance of the process during scheduling and displays this same number if you request the process' priority.

To maintain compatibility with AOS, however, AOS/VS has to map priority numbers for swappable processes. As a result, the actual number the system uses in its scheduling calculations and the number it displays when you request the process' priority may differ.

The discrepancy between actual and displayed priority numbers occurs in three cases:

1) If you assign a swappable process priority of 1, 2, or 3.

2) If you assign a swappable process priority of G1+1 - G1+3.

3) If a resident/preemptible process with priority 1, 2, or 3 changes its type to swappable.

In all three cases, AOS/VS uses a priority number of G1+1 - G1+3 when scheduling the process because a swappable process cannot have a priority of 1, 2, or 3. The system cannot, however, display the numbers G1+1 - G1+3 for a swappable process, and so displays 1 - 3.

In all other cases (4 - G1 and G1+4 - 511), the actual number is the same as the displayed number.

Remember, however, that if you do assign a swappable process a priority of 1 and then it changes type to resident (or preemptible), the resident process will have an actual priority of 1, even though the swappable process could not.

1.1.2.5 Examples of Mapping

1) If a resident process with a priority of 2 changes its type to swappable, the system displays a priority of 2, but it actually uses G1+2 when scheduling the swappable process.

2) If a resident process with a priority of 3 changes its type to preemptible, the system displays and uses a priority of 3 for the preemptible process.

3) If a preemptible process with a priority of G1+3 changes its type to swappable, the system displays a priority of 3, but uses G1+3 in scheduling the swappable process.

4) If a preemptible process with a priority of G2+44 changes its type to swappable, the system displays and uses a priority of G2+44 for the swappable process.

5) If a swappable process with a display priority of 3 (meaning its real priority is G1+3) changes its type to resident, the system displays and uses a priority of 3 for the resident process.

6) If a swappable process with a priority of 5 changes its type to preemptible, the system displays and uses a priority of 5 for the preemptible process.

1.1.2.6 PNQF

The Priority eNQue Factor (PNQF) is the value by which ELQUE is ordered. For G1 processes the PQNF is equal to the process priority. For G2 processes it is a calculation. This calculation allows for subgroups within each priority.

```
If the process is in Group 1 then:
    PNQF = Priority
         E = 6

If the process is in Group 2 then:
    PNQF = G1 + 1 + (7 *(process priority - g1)) + E

If the process is in Group 3 then:
    PNQF = G1 + 1 + (7 *(process priority - g1)) + E
         E = 6
    G1 is the genned max value for a G1 type process.
```

Before the 7 in the PNQF calculation can be described the "E" in the calculation must be discussed. The E is the time slice Exponent. This is a value from 1 - 6 and varies based on process behavior. The lower the E value the more interactive a process is and, therefore, the process will have a higher priority. Therefore, AOS/VS favors the more interactive processes over CPU bound process within Group 2. The E value allows the ELQUE manager to subdivide each G2 priority group into 6 subqueues. These subqueues are managed by round-robin. The E value gets set up based on the amount of time slice residue the process has left. There are two times when E is touched from the scheduling point of view: 1) after unblocking a process and 2) after the process uses up its time slice. The algorithm below shows how E is calculated.

```
/* ********************************************** */
/*  This block of code calculates a new Time Slice    */
/*  Exponent after a process unblocks.                */
/*                                                    */
/*                                                    */
/* ********************************************** */
    residue = ptbl.pextn.pscln;
    current_exponent = ptbl.pslex;
    total_subslices = 2** current_exponent;
    remaining_subslices = total_subslices - residue;
    E = lead_bit(remaining_subslices) /*see LOB instruction*/
    if (E >> 5)
         E = 5;
    if (E << 1)
         E = 1;
```

During time slice end processing, E is merely incremented unless E is already at the maximum of 6.

The 7 is used to divide each priority grouping into subgroups. These subgroups are numbered from 0 to 6. (See Figure 1.4.)

ELQUE PNQF SUBGROUPS

G1 slots	G2 CB	G2 1st	G2 2nd	G2 3rd	G3 1st
PNQF = PRI	slot 0	1 -- 6	1 -- 6	1 -- 6	1 -- 6

Figure 1.4

The ELQUE subgroups for G1 are a one-to-one relationship, e.g., PNQF = priority. The subgroups for G2 and G3 allow for a subgroup 0. This subgroup allows a spot for a G2-G3 control block. Since the G2-G3 CBs must go before the G2 processes then there must be an assigned place on ELQUE for the G2-G3 CBs. Subgroup 0 is used for that purpose. Because there are 6 possible subgroups allowed by the Exponent subgroup 0 is accounted for by the 7 in the PNQF calculation. Slot 0 in the other G2 and G3 subgroups will not be used, but that is not a problem because no extra space is taken up by the holes.

The difference between the process priority and G1 is the value of the G2 mapped priority. (See Priority Mapping above.)

1.1.2.7 BIAS Factors

The locations BIAS and HBIAS, in STABLE, define the AOS/VS bias value. The bias factor is used to manage the size of ELQUE by setting an upper and lower limit on the number of non-interactive processes on ELQUE. BIAS contains the minimum number of non-interactive processes that AOS/VS attempts to keep on ELQUE, while HBIAS represents the maximum number. A non-interactive process in the AOS/VS sense is a swappable process having a time slice exponent of 6. This mechanism is used when trying to preempt or swap a process on ELQUE. If the number of CPU bound processes is greater than HBIAS, then some of the processes will be preempted or swapped.

1.1.3 HANDQ -- unpended queue

There are some problems in working with the Eligible Queue. The main problem comes from the locking mechanisms used to manage ELQUE. If ELQUE is locked when trying to put an element on ELQUE, the code path must do something to avoid spinning on the QLOCK. Therefore, the system puts the element on another queue called the HANDler Queue (HANDQ). This way the code does not have to spin waiting for a lock.

The HANDQ is a queue of elements that have become unpended or unblocked but could not be put on the ELQUE because ELQUE was locked. This queue removes the need for a system code path that is trying to put something on ELQUE to spin on the ELQUE locks. The code that puts an element onto ELQUE tries to get QULCK, which is a spin lock. If the lock is held, then the code puts the element onto HANDQ. (See PENQ.)

1.1.4 PELEMQ -- pended element queue

The PELEMQ is a queue of control blocks that have pended on some event, system control blocks such as core manager and system manager, and specific user CBs which are pended on MKEY. The purpose of this queue is to shorten the length of the ELQUE as memory size in MV's increases thereby allowing for more processes to be potentially active. A long ELQUE increases the amount of time the search instruction takes to find an element.

1.1.5 The Globals

QMIDDLE is used by the ELQUE enqueue routines, which do pseudo-binary searches. (See PENQ.) To do any kind of binary search a middle point must be defined. For ELQUE, AOS/VS must use a PNQF value. The value of QMIDDLE is 417. This is the location of the priority 256.(400 octal) where E=1. (See PNQF calculation.)

QMIDCNT is a counter that reflects which side of the ELQUE the PSEUDO binary search falls on. If the QMIDCNT is positive, then the majority of the searches were on the low side of the QMIDDLE (e.g., PNQF < 417). If QMIDDLE is negative, then the search spends more time on the high side of QMIDDLE (e.g., PNQF > 417).

G1RANGE is two double words used to define the range of group1. It is set up at SINIT time to test what group a process is in. During a proc this range is used to decide what time slice exponent to give to the blocking caller. If the caller is in G1RANGE then the callers exponent is set to 6.

G1 is the Genned value of the highest possible G1 process priority.

G2 is the Genned value of the highest G2 priority.

G4 is the highest priority on the system. The value of G4 is 511..

SCMASK is a constant that holds the initial scan mask. The value of this constant is 160023. For more information on the bits of this constant see the Process table PSTAT word.

NCBDEQ.W counts the number of times CBDEQ was called.

The next group of counters are used as collision counters. These counters are updated every time a spin lock collision occurs. These counters are only used for development and debugging purposes.

CQMVS.W counts the number of source queue collisions in the QMOVE routine.

CQMVD.W counts the number of destination queue lock collisions in the QMOVE routine.

CCBLK.W is used to count the number of times CBDEQ encountered a lock of ELQUE.

CCBDQ.W counts the number of times CBDEQ had to use HANDQ.

CPDEQ.W counts the number of collisions encountered in the general dequeueing routine PDEQ.

CPQHL.W counts the collisions in the ENQH routine.

CPQTL.W counts the collisions in the ENQT routine.

1.2 Queueing Operations

There are 7 basic queueing operations that work with the queues discussed in the previous operations. The general queueing operations are PENQ, PENQG, PDEQ, QMOVE, CBDEQ, PENQT, and PENQH. These operations do whatever locking operations are needed to maintain the integrity of the queues they're working with. Below is the pseudocode for the queueing routines.

1.2.1 PENQ

PENQ enqueues an element onto the ELQUE only. The element is positioned on ELQUE by PNQF. The routine searches ELQUE to find the place to put an element. This is done by a PSEUDO binary search of the ELQUE. If the location of the position on the queue is found in the first half of the queue, then the routine returns; otherwise, the second half is scanned from the end and returns. The middle of the queue is defined as QMIDDLE.

Ordering by PNQF(p = PNQF)

```
ELQUE
 |                    QMIDDLE
 |                       V
 > p0|p2|...|p400|p416|p417|p420|....|p776|p777|
   -------->--------->-----|------<----------<--
       PNQF <= 417    ^         ^   PNQF >> 417
                 DIRECTION OF SEARCH
```

Figure 1.5 Pseudo-Binary Search of ELQUE

```
/* $$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$ */
/*                   PENQ(element)                     */
/* This routine enqueues entities onto ELQUE.          */
/*                                                     */
/* $$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$ */

penq(element)
    {
    pnqf = element->PNQF;
```

```
/* ************************************************ */
/* If the PNQF of the element is in the first half of */
/* the queue then scan the first half of the queue.   */
/*                                                    */
/* ************************************************ */

    if (pnqf <= QMIDDLE)
         {
         pnqf ++; /* put at end of priority group */
         qmidcnt ++;
         get_q_lock(&ELQUE,qlock,&CPENQ.W);
         location_found = NFSLE(ELQUE,pnqf,element->PNQF);

/* ************************************************ */
/* If the scan of the ELQUE found the right location  */
/* in the queue to put an element then enqueue the    */
/* element to that location.                          */
/* ************************************************ */

         if (location_found->PNQF <= qmiddle);
              {

/* ************************************************ */
/* Wait for the scanner's count to go to zero before  */
/* enqueueing to it. This prevents the queue from     */
/* being altered while other paths are looking at it. */
/*                                                    */
/* ************************************************ */

              if (ELQUE.QSCAN >> 0)
                   while (ELQUE.QSCAN >> 0)
                        {}
              ENQH(location_found,element);
              return();
              } /* if found location */
         else

/* ************************************************ */
/* Wait for the scanner's count to go to zero before  */
/* enqueueing to it.  This prevents the queue from    */
/* being altered while other paths are looking at it. */
/*                                                    */
/* ************************************************ */

              {
              if (ELQUE.QSCAN >> 0)
                   while (ELQUE.QSCAN >> 0)
                        {}
              ENQT(location_found,element);
              return();
              } /* did not find location */
         }/* search first half */
```

```
/* ************************************************ */
/* Search the second half of the eligible queue.    */
/* If the PNQF of the element was not less than or  */
/* QMIDDLE value.  This section does a backward     */
/* search of the second half of ELQUE.              */
/* ************************************************ */

    QMIDCNT --;
    pnqf = element->PNQF;
    get_q_lock(queue,qlock,penq.w);
    location_found = WBSGE (ELQUE,pnqf,element->PNQF);

/* ************************************************ */
/* If the scan of the queue found the right location */
/* in the queue to put an element, then enqueue the  */
/* element to that location.                         */
/* ************************************************ */

    if (location_found->PNQF >= QMIDDLE);
         {

/* ************************************************ */
/* Wait for the scanner's count to go to zero before */
/* enqueueing to it.  This prevents the queue from   */
/* being altered while other paths are looking at it. */
/*                                                   */
/* ************************************************ */

         if (ELQUE.QSCAN >> 0)
              while (ELQUE.QSCAN >> 0)
                   {}
         ENQT(location_found,element);
         return();
         } /* if found location */
    else

/* ************************************************ */
/* Wait for the scanner's count to go to zero before */
/* enqueueing to it.  This prevents the queue from   */
/* being altered while other paths are looking at it. */
/*                                                   */
/* ************************************************ */

         {
         if (ELQUE.QSCAN >> 0)
              while (ELQUE.QSCAN >> 0)
                   {}
         ENQH(location_found,element);
         return();
         } /* did not find location */
    } /* PENQ  */
```

1.2.2 PENQG

PENQG is used as a general queueing routine to enqueue by PNQF. The routine searches the queue to find the place to put an element. This is done by a PSEUDO binary search of the queue. If the location of the position on the queue is found in the first half of the queue, then the routine returns; otherwise, the second half is scanned from the end. This routine uses the same logic as PENQ but is designed to work with any queue.

```
/* $$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$ */
/*                    PENQG(queue,element)                       */
/* This routine works with queues other than ELQUE.              */
/* It is assumed that the queue is modified by PNQF.             */
/*                                                               */
/* $$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$ */

penqg(queue,element);
    pnqf = element->PNQF;

/* ********************************************** */
/* If the PNQF of the element is in the first half of */
/* the queue, then scan the first half of the queue.  */
/*                                                    */
/* ********************************************** */

    if (pnqf <= QMIDDLE)
         {
         pnqf ++;
         qmidcnt ++;
         location_found = WFSLE(queue,pnqf,element->PNQF);

/* ********************************************** */
/* If the scan of the queue found the right location  */
/* in the queue to put an element, then enqueue the   */
/* element to that location.                          */
/* ********************************************** */

         if (location_found->PNQF <= qmiddle);
              {
              ENQH(location_found,element);
              return();
              } /* if found location */
         else
              ENQT(location_found,element);
              return();
              } /* did not find location */
         }/* search first half */
```

```
/* ************************************************ */
/* Search the second half of the QUEUE.             */
/* If the PNQF of the element was not less than or  */
/* QMIDDLE value.  This section does a backward     */
/* search of the second half.                       */
/* ************************************************ */

    QMIDCNT --;
    pnqf = element->PNQF;
    location_found = WBSGE (queue,pnqf,element->PNQF);

/* ************************************************ */
/* If the scan of the queue found the right location  */
/* in the queue to put an element, then enqueue the   */
/* element to that location.                        */
/* ************************************************ */

    if (location_found->PNQF >= qmiddle);
         {
         ENQT(location_found,element);
         return( );
         } /* if found location */
    else
         {
         ENQH(location_found,element);
         return( );
         } /* did not find location */
    } /* routine  */
```

1.2.3 PDEQ

PDEQ is the routine that removes a PTBL from a QUEUE. The routine has two arguments passed to it: the queue address and the PTBL address.

```
/* ************************************************ */
/*                 PDEQ(queue,ptbl)                 */
/* This routine removes an element from a queue     */
/* supplied to the routine.                         */
/*                                                  */
/* ************************************************ */

pdeq(queue,ptbl)
    {
    get_q_lock(queue,qlock,cpdeq.w);

/* ************************************************ */
/* Are we working with ELQUE?  If so, check for     */
/* validity of the element to be dequeued.          */
/*                                                  */
/* ************************************************ */

    if (queue == ELQUE)
         {

/* ************************************************ */
/* Are we trying to dequeue the root process table? */
/* If so, panic with a 14627.                       */
/*                                                  */
/* ************************************************ */

         if (ptbl->plink.w == -1)
              panic(14627);

/* ************************************************ */
/*  Wait for any scanners to finish before dequeueing */
/*  the element from ELQUE.                         */
/*                                                  */
/* ************************************************ */

         if (ELQUE.qscan >> 0)
              while(ELQUE.qscan >> 0)
                   {}
         }
    DEQUE(queue,ptbl)
    release_q_lock(queue)
    queue.qusers --;
    return();
    }/* PDEQ */
```

1.2.4 QMOVE

QMOVE is the operation that moves an element from one queue to another. The routine does any locking that is necessary to work with the queues, but the caller must set any element transition locks. If necessary, the caller must turn off interrupts.

```
/* ************************************************ */
/*              QMOVE(sque,element,dque)            */
/* This routine moves the element from the source   */
/* queue(sque) to the destination queue(dque).      */
/* both queues get locked for the move.             */
/*                                                  */
/* ************************************************ */


qmove(sque,element,dque)
    {
    get_q_lock(sque,qlock,CQMVS.W);


/* ************************************************ */
/* Are we working with ELQUE?  If so, check for     */
/* validity of the element to be dequeued.          */
/*                                                  */
/* ************************************************ */

    if (sque == ELQUE)
         {

/* ************************************************ */
/* Are we trying to dequeue an element that is not on */
/* a queue?  If so, panic with a 14630.             */
/*                                                  */
/* ************************************************ */

         if (ptbl->plink.w == -1)
              panic(14627);
```

```
/* ************************************************ */
/* Wait for any scanners to finish before dequeueing  */
/* the element from ELQUE.                            */
/*                                                    */
/* ************************************************ */

          if (ELQUE.qscan >> 0)
               while(ELQUE.qscan >> 0)
                    {}
          }
     DEQUE(sque,ptbl)
     release_q_lock(sque)

/* ************************************************ */
/*  Is the destination Queue ELQUE?  If so, then      */
/*  check for scanners of the queue.                  */
/* ************************************************ */

     if (dque == ELQUE)

     {
       PENQ(element);
            Return ();
     }

     get_q_lock(dque,qlock,CQMVD.W)
     ENQT(dque,element);
     release_q_lock(dque);
     return();
     }/* Qmove */
```

1.2.5 CBDEQ

This routine is called by UNPEND and UNPNDN to remove an element from PELEMQ and put it on the right queue.

```
/* ************************************************ */
/*                    CBDEQ                         */
/* This routine dequeues elements from PELEMQ and   */
/* enqueues them onto the ELQUE or HANDQ.           */
/* The argument is the address of the element to be */
/* dequeued.                                        */
/* ************************************************ */

cbdeq(cb)
    {
    NCBDEQ.W++;

/* ************************************************ */
/*  If the element is not pended, then panic with a */
/*  14572.                                          */
/*                                                  */
/* ************************************************ */

    if (bit (cb->pstat,psrdy) == 0)
         panic(14572);
    clear_bit(cb->pstat,psrdy)

/* ************************************************ */
/*                                                  */
/*  Clear the pend key and "not ready to run bit" in */
/*  the CB.                                         */
/*                                                  */
/* ************************************************ */

    cb->ckey = 0;
    DEQ(pelemq,cb);
```

```
/* ************************************************ */
/*  If ELQUE is locked, then use the HANDQ.         */
/*  ENQUEUE the element to HANDQ and return.        */
/*  Otherwise lock ELQUE.                           */
/* ************************************************ */

    if (elque_is_locked)
         {
         CCBDQ.W ++;
         ENQT(handq,cb);
         set_bit(MPPCB->ppstat,prsch);
         return();
         } /* if */
    else
         {
         get_q_lock(elque);

/* ************************************************ */
/*  If there are other paths scanning ELQUE, wait for */
/*  the count to go to zero.  We do not want to touch */
/*  ELQUE while there are scanners because we may     */
/*  corrupt the queue.                              */
/*  After we're done, return.                       */
/* ************************************************ */

         if (elque_scanners >> 0)
              while (elque_scanners >> 0)
              {}
         ENQ(elque,cb,pnqf);
         event(); /* See JP management. */
         return();
         }  /* else */
    }/* end of cbdeq */
```

1.2.6 PENQT

When a path wishes to enqueue an element to the tail of a queue the routine PENQT is called. PENQT stands for Ptbl ENQue to Tail. This routine does all the necessary locking to work with the QUEUE.

```
/* ************************************************** */
/*                  PENQT(queue,ptbl)                 */
/* This routine enqueues a PTBL to the tail of a queue.*/
/* ************************************************** */

penqt(queue,ptbl)
    {
    get_q_lock(queue,qlock,CPQTL.W);
    ENQT(queue,ptbl);
    release_q_lock(queue);
    return();
    }/* PENQT */
```

PENQH

When a path wishes to enqueue an element to the head of a queue, the routine PENQH is called. PENQH stands for Ptbl ENQue to Head. This routine does all the necessary locking to work with the QUEUE.

```
/* ************************************************** */
/*                  PENQH(queue,ptbl)                 */
/* This routine enqueues a PTBL to the head of a queue.*/
/* ************************************************** */

penqt(queue,ptbl)
    {
    get_q_lock(queue,qlock,CPQHL.W);
    ENQH(queue,ptbl);
    release_q_lock(queue);
    return();
    }/* PENQT */
```

1.3 The Scanner

The scanner is the path, in the module SCHED, that scans ELQUE to find a CB or PTBL to run. This path is not called "scanner" in the code, it has several entry points. (See below.) The scanner cleans up HANDQ by removing elements from it and putting the elements onto ELQUE. The scanner then builds a scan mask for its scan, scans ELQUE and dispatches the element found. If no element is found then class scheduling is reset. (See LP Management) or the scanner will go to the Checksum loop. (See JP Management.)

The major entry points to the scanner are used for optimization of locking and uncontrollable changes of state in the system. The entry points are listed as follows:

- RESCH is the entry point for the top of the scanner. It is the place that paths go to when they find the need for a reschedule.

- SMON0 is at the same location as RESCH. Most of the system paths that go to the scanner go to SMON0. This is different than RESCH only in name. A branch to SMON0 implies that this is just a normal reschedule; e.g., a CB pended.

- M6 is the entry point of the element dispatch. This entry is used by TRTN (see CB management) to allow a PTBL that made a "fast" call to continue running.

In this chapter the scanner is presented from the ELQUE management point of view. The reason for this, is that the scanner in the system imbeds three logically separate points of view in line. For example, consider the following lines of C code.

```
A:  if (bit(myppcb.w.cpstat,cpmast) !=1) /*daughter?*/
    setbit(mask,process_mother_bit);

B:  element = SCAN(*ELQUE,mask);

C:  if (mask == mylpcb.lpciu.w)
        {}
```

"A:" is supplied by JP management because the if statement uses the global MYPPCB.W to find out if this is a daughter processor. (See JP Management.)

"B:" is supplied by ELQUE management because the scan function uses ELQUE, which is managed in ELQUE management.

"C:" is supplied by LP management because the initial scan mask comes from the LPCB.

In the above example, three major areas of Paths and Time are used in three nearly consecutive commands. This is the way the code is really presented in the system, but to modularize the scanner for each section the scanner is presented with different emphasis.

There are three possible things that happen from the scan of ELQUE.

1) The scan could find a PTBL. In this case, the PTBL is sent to the PTBL dispatcher. (See PTBL scheduling.)

2) The scan could find a CB. In this case, the CB is sent to the CB dispatcher (TACT or TACT1). (See CB management.)

3) The scan could not find anything. In this case, if class scheduling is on (see LP management) the scanner could do a reset or go to secondary classes. If class scheduling is not on or there is nothing more to be done even with class scheduling on, then the scanner will go into the Checksum loop (see JP management).

The pseudocode below shows the scanner from the LP point of view.

```
/* $$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$ */
/*                                                               */
/*                    The Scanner                                */
/*     The scanner will loop forever unless it is told           */
/*     to go_idle. (See JP management.)                          */
/*                                                               */
/* $$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$ */

#define loop_forever true
scanner( )
    {
    int  model_tier; /* variable used for call to
                                   set_mask*/
SMON0:
RESCH:

/* ********************************************************** */
/*                                                            */
/*  This is the top of the scanner.  Show that this           */
/*  JP is not running a user.                                 */
/*                                                            */
/* ********************************************************** */

    CC.W = 0;
    MYPPCB.W->cpelm.w = -1;

/* ********************************************************** */
/*                                                            */
/*  Clean out the Handler Queue.  Take from HANDQ and         */
/*  put onto ELQUE. First check if there is anything          */
/*  on the queue and try to lock it.  If either               */
/*  fails then don't work with the HANDQ.  Let                */
/*  another JP work with it.  If HANDQ is locked              */
/*  then someone else is working with it and therefore        */
/*  it will get cleaned out eventually so continue            */
/*  onto the scan.                                            */
/* ********************************************************** */

    interrupts (on);
    if (HANDQ.head != -1)
         if (bit(HANDQ.qlock,0) != 1)
              {
```

```
/* ************************************************ */
/*                                                  */
/* This is the loop to clean off of HANDQ.  If there  */
/* is more than one JP in the system, then see if the */
/* JPs reschedule flag can be set.  This allows any   */
/* JP that is in the checksum loop to do a reschedule.*/
/*                                                  */
/* ************************************************ */

            while (HANDQ.head != -1)
                 {
                 element = DEQ(HANDQ);
                 interrupts(off);
                 PENQ(element);
                 interrupts(on);
                 if (MAXCP > 1)
                     EVENT();   /* see JP management */
                 }
            }

/* ************************************************ */
/*                     Lock ELQUE                   */
/* ************************************************ */

    model_tier = 0; /* initialize value passed to    */
                          /* get_mask routine LP management*/
    while(loop_forever)
      {
/* ************************************************ */
/*                                                  */
/*    Get the current mask.  If child processor then  */
/*    mask out the mother bit and scan ELQUE.         */
/*    MYPPCB.W is the PPCB for that JP.               */
/*    mylpcb.w is the address of this lpcb.           */
/*    setmask is a function in JP management.         */
/*                                                  */
/*    The Scan mask used by the scanner is broken into*/
/*    two parts.  The first part(I call classmask)    */
/*    comes from the LPCB. (See LP management.) The   */
/*    second part (I call regmask) comes from         */
/*    SCMASK.W.  These two masks are concatenated to  */
/*    be used in the scan.  If both the bit in the    */
/*    PTBL/CB and the corresponding bit in the mask   */
/*    are set then that element will not be selected. */
/* ************************************************ */

        regmask = SCMASK.W;
        if (bit(MYPPCB.W->cpstat,cpmast) !=1) /*daughter?*/
             setbit(regmask,pmast);
```

```
/* ************************************************ */
/*                                                  */
/*  If there are no more stacks (see CB             */
/*  management) and no more free memory that can be */
/*  used to get more stacks, then set the           */
/*  "run only CBs" bit in the mask.  This is done   */
/*  to prevent selection of an element that we can't */
/*  run.                                            */
/*                                                  */
/* ************************************************ */

        if ((SSTKCT <= 0) && (SSTUSE == 0) &&
        (((FBLKC.W+UNMODCN.W) <= 5) !! (SSTMAX == 0)))
             setbit(regmask,psncb);
        classmask = set_mask(mode,mylpcb,ppcb->cpmode);

        /*  Build the scan mask by putting the class  */
        /*  mask in the high part of the double word  */
        /*  and the regular mask in to low part.      */

        mask = concat(classmask,regmask);
/* ************************************************ */
/*                      RESCN                       */
/*  This is an entry point used when the rescan bit */
/*  is set.  The rescan does not change the mask, it */
/*  just locks ELQUE and rescans it for an element. */
/*  The check for rescan is made when the scanner is */
/*  about to go into the Checksum loop.             */
/*  The Rescan Bit is set when an event occurs which */
/*  causes a reschedule.  When reschedule is set    */
/*  Rescan is also set.  See EVENT in JP management. */
/* ************************************************ */

RESCN:
        get_q_lock(ELQUE);
        ELQUE.QSCAN++;
        element = SCAN(*ELQUE,mask);
```

```
/* ************************************************ */
/*  If the scan was successful the element will get  */
/*  dispatched.  In the code the dispatch is a JMP   */
/*  @PPC.W.  This means that the element knows the   */
/*  dispatcher it will use.  For user CBs the        */
/*  dispatcher is TACT. (See CB management.)         */
/*  For PTBLs the dispatcher is PCALL. (See PTBL     */
/*  management.) For system CBs coming from IDSCBQ   */
/*  the dispatchers are the startup sections of the  */
/*  code.  For system CBs that pended, the dispatcher */
/*  is TACT1.                                        */
/* ************************************************ */

        ELQUE.QSCAN --;  /* one less scanner */

        if (successful_scan)
             switch(element_type)
                  {
                  case ptbl:
                      goto PCALL;
                      break;
                  case CB:
                      goto TACT;
                      break;
                  case sys_cb:
                      if (just_woke_up) && (cormanager)
                           goto CMINT;
                      else
                           if ((just_woke_up) && (sysmanager))
                                goto SMINT;
                           else
                                goto TACT1;
                                break;
                  }  /* switch    */
             } /* if  */
```

```
/* ************************************************ */
/*                                                  */
/*  If not successful then check to see if          */
/*  class scheduling in on.  If not then go idle.   */
/*  If class scheduling is on then check if a mode  */
/*  change is necessary.   If the current           */
/*  mask is the same as the initial scan mask,      */
/*  RESET and change mode.  This is considered a    */
/*  sufficient check because the scan failed with   */
/*  the initial mask  meaning there are no more     */
/*  primary classes ready to run.  To avoid a second */
/*  unnecessary scan, RESET to run the secondary    */
/*  classes.                                        */
/* ************************************************ */

             if (bit(MYLPCB.W->lpstat,lpoff) == 1)
                  {

                  /* Before going idle check to see if a  */
                  /* Rescan is necessary.  if so then goto*/
                  /* RESCN in this routine.               */

                  if (bit(MYPPCB.W->cpstat,crescn) == 1)
                      {
                      clearbit(MYPPCB.W->cpstat,crescn);
                      goto RESCN;
                      }
                  goto SMONDD;  /* see JP management */
             if (classmask == mylpcb.lpciu.w)
               {
                  reset(mylpcb.w);
               }  /* if */
             else

/* ************************************************ */
/*                                                  */
/*  If not changing the mode, then check it to see  */
/*  what mode we're in.  If mode 0, then reset the  */
/*  lp databases and get a new mask. If not, get the */
/*  next tier if possible.                          */
/* ************************************************ */
                  if (MYPPCB.W.cpmode == 0)
                      reset_(MYLPCB.W);  /* LP management */
                      mode1_tier = 0;
                  else
             /* check if we're at the last tier. */
                      if ((MYPPCB.W->cptmk.w+2 >
                      MYPPCB.W->cphmk.w+32) !!
                      (MYPPCB.W->cphmk.w+2 == -1))
```

```
                        {
                        reset();
                        MYPPCB.W->cpmode = 0;
                        model_tier = 0;
                        }
                    else
                        {
                        MYPPCB.W->cptmk.w += 2;/*next tier */
                        model_tier =  MYPPCB.W->cptmk.w;
                        }/* else */
    } /* while loop */
} /*  scanner  /*
```

1.4 Locking

There are two types of locking discussed in paths and time. The first is "spin" locking. (See JP management.) The second is pend locking.

1.4.1 Pend Locks

Pend locking is only used by pendable paths (CBs). A pend lock causes a CB trying to get the lock to pend if the lock is held. The reason for using pend locks is that the particular lock is a long term lock. This means that the lock may be held for an indeterminate amount of time. For a path to spin the lock must be a short term lock.

Example:

A CB is trying to get a change lock on the global lock JPLPLOCK.W. (See LP management.) The CB finds the lock is held, so the CB will pend waiting for the lock. When the lock is released the unlocking routine unpends all the CBs pended on the lock.

To get a pend lock there is a two-level locking scheme used. The first part is getting the transition lock and the second is getting the pend lock.

The first part of the locking scheme is getting the transition lock. The transition lock allows the Code Path to set up for a long term action, such as get a long term lock or do some quick operation with an element. If the path tries to get the long term lock and cannot then the path will pend, but before pending the path must release the transition lock. The routine that this manual has designated to do this type of locking is get_lock. (See LP management.)

1.4.2 Element and Queue Locking

In this section two groups of locking routines will be discussed. The first group of locking routines used in ELQUE management are the queue locks. The second group of locking routines discussed are the pend locks that deal with JP and LP databases.

1.4.3 Element Locking

In ELQUE management it is necessary to lock elements and queues to maintain their integrity for specific operations. For example, when the element is accessed for modification, the queue lock (qlock) must be held on that queue before the queue can be touched. Elque has a special extra lock, which will be discussed later.

1.4.3.1 Element Locks

An element gets locked when it has some system call working with it, such as a system call (CB) doing an operation to a PTBL; or when a PTBL is being dispatched to run. The PTBL long term lock, PLOCK, is a form of a pend lock. The difference between the PTBL lock and the normal pend lock is, if the PTBL lock is set, the dispatchers, which are nonpendable paths, simply do not use the PTBL and return to the scanner for another element.

1.4.3.2 Queue Locks

There are two routines used by this manual to lock queues. They are: get_q_lock and release_q_lock. The locking scheme used for queues is spin locking. These locking routines are not real locking routines, they are implemented inline. The reason these functions are implemented inline is because of the speed of not having to go to a subroutine. The routines are shown in the pseudocode below.

1.4.3.3 GET_Q_LOCK

Get_q_lock tries to get a lock for the caller. When the locking succeeds then the routine returns. If the locking fails the routine increments the collision counter supplied to the routine and spins until it gets the lock.

```
/* $$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$ */
/*          get_q_lock(queue,counter)                        */
/* This routine gets a lock for the queue                    */
/* supplied as an argument.                                  */
/*                                                           */
/* $$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$ */
```

```
get_q_lock(queue,counter)
    {

    /* ****************************************** */
    /*  if there lock is locked then increment the  */
    /*  counter and spin.                           */
    /*  After the lock is released then set the     */
    /*  lock.                                       */
    /* ****************************************** */

    if (bit(queue.qstat,QLOCK) == 1)
         counter ++;
         while (bit(queue.qstat,QLOCK) == 1)
              {}
    setbit(queue.qstat,QLOCK);
    return( );
    }
```

1.4.3.4 RELEASE_Q_LOCK

Release_q_lock releases the qlock for the queue passed as an argument.

```
    /* ****************************************** */
    /*         release_q_lock(queue)              */
    /* This roiutine releases the queue lock for  */
    /* queue passed to the routine.               */
    /*                                            */
    /* ****************************************** */

release_q_lock(queue)
    {
    clearbit(queue.qstat,QLOCK);
    return;
    }
```

1.4.4 ELQUE Locking

ELQUE has a special extra locking scheme, which is used for readers of the queue. This lock is called the scan count. The scan count is used to keep track of scanners. This prevents a path from modifying ELQUE while other paths are reading it. After a path gets the QLOCK on ELQUE it must also wait for the scan count to go to zero before enqueueing or dequeueing from ELQUE. The scan counter is useful because scans of ELQUE occur more frequently than do modifications. The scan count for ELQUE is called QSCAN. QSCAN is a part of the ELQUE structure so it is accessed by ELQUE.QSCAN.

Chapter 2
CB Management

2.1 Introduction

Control Block or CB management is the part of Paths and Time that manages the databases and paths used by CBs. CBs are system paths that run on behalf of a user request or are used to manage the system resources. There are three types of CBs: user CBs, system CBs, and daemons.

Control blocks are used for cases in which the system needs a stack to handle a code path or there is a possibility that the path will pend. These stacks may be allocated when a user makes a system call that requires a stack, or they may be allocated for system use (daemons).

Both the control block and process table share some common header area information so the scheduler can accommodate them on the scheduler queues. This eliminates the need for separate queues for process tables and CBs when scanning for eligible elements to run. Elements are members of a queue; in this case these elements are PTBLs and CBs.

- User CBs are used whenever a user makes a system call that goes into ring 0. The CB runs until some event occurs to cause the CB to Wait for some condition to be met. This "waiting" is called pending. An example of pending is CB waiting for a disk request to return.

- System CBs do system resource management services such as manage memory.

- A Daemon is a special kind of CB that performs a function for the system on behalf of a user, such as termination.

- CB Management is the highest-level of the object management components of Paths and Time. CB management gets services from Process scheduling, ELQUE management, and Time management. From process scheduling, CB management gets the CB that will be run. CB management gets the CBs to dispatch from ELQUE management. CB management gets timing data (in PIT ticks) for the CBs from time management.

Figure 2.1 below shows the connections between CB management and Paths and Time. The connections are services provided to CB management by the other components of Paths and Time.

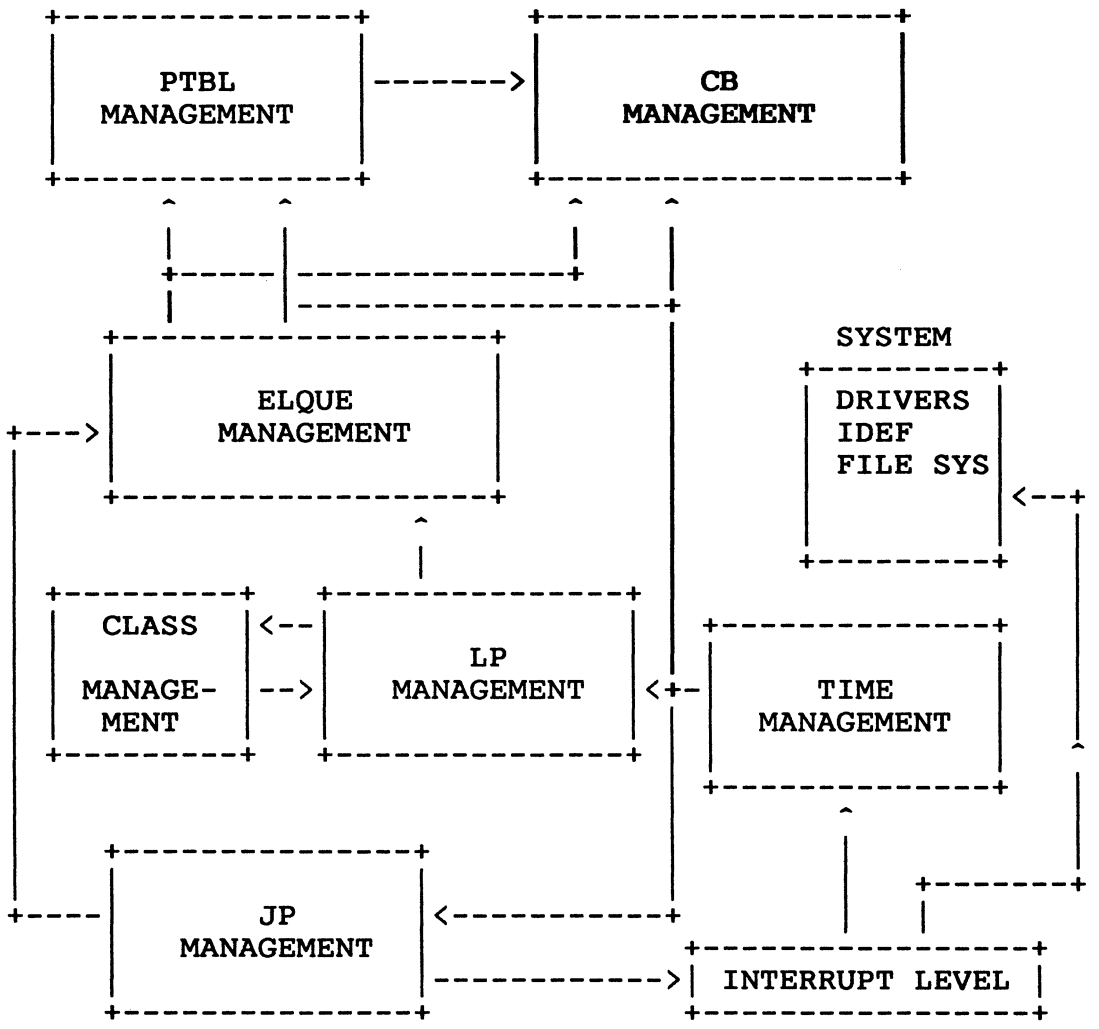


Figure 2.1

This chapter is organized as follows:

The objects
The operations that work on the objects
The internal paths of CB management
Service provided to the rest of AOS/VS

2.2 Objects

This section describes the objects in CB management: the PTBL, CB, CB management Queues, and the globals.

2.2.1 The PTBL/CB

The process table (PTBL) is used by AOS/VS to manage a process. The first 24 words, however, are used as a common area between the CB and the PTBL. To access a common area, certain locking conventions may need to be used. These locks are defined below. The letter representing the type of lock is shown in the PTBL/CB database offsets section.

*** PTBL LOCKING CODES ***

For each PTBL/PEXTN definition there is a one-letter code that indicates what lock, if any, is needed to access the value. A more detailed explanation is supplied in the section on locking. The one-letter codes are listed below:

T = The PTRAN lock controls this value.

L = The PLOCK lock controls this value.

N = NONE No lock is needed to access this value.

I = INVARIANT the value does not change for the life of the process and no lock is needed to access the value. [Note, however, that the caller may need to ensure the continued existence of the process (via counter PTGRC, for example).]

X = NOT APPLICABLE the corresponding value is not applicable to PTBLs and, thus, does not fall within the realm of the PTBL locking scheme (e.g., a value's definition is for CB only).

LA = PLOCK/ATOMIC to access this value you must either acquire PLOCK or use an atomic instruction.

RE = REFER refer to another definition to get this value's locking code. (For example, bit definition words do not have a locking code, you must refer to the individual bits.)

P = PID LOCK the PTBL pointer values to sons and fathers (PSONP.W, PSONL.W) fall under pids lock PIDSLK.W.

U = UNUSED the value is not used and should be removed from PTBL.

S = SPECIAL locks one or more different locks not previously mentioned. For example, an offset may require some sort of interrupt mask as well as a normal locking code.

The following is a field-by-field description of the main database for each process on the system. It is called a "Process Table." This database is built when a process is at ?proc time. The fields described below are only the fields used by Paths and Time.

PTBL OFFSETS COMMON TO BOTH CBs And PROCESSES

WORD	OFFSET	LOCK	USAGE SUMMARY
0	PLNK.W	T	Forward Link
2	PBLNK.W	T	Backward Link
4	PNQF	L	Priority eNQue Factor
5	PCLASS	L	Process CLASS
6	PSTAT	RE	Process Status Bits for SCAN
7	PSTAT1	RE	Process Status Bits not FOR SCAN
10	PPC.W	I	Control address when scheduled
12	PLPCB.W	L	LPC Block to charge run time
14	PGNUM.W	L	(CLASS #)*2 integer used by XWADD
16	PTIM	L	Current interval of time expended by a system call
17	TTIME	L	Accumulated time expended by a system call
20	CALLN.W	L	(SYS CALL NUMBER)*2 used during SYS CALL time accounting by XWADD
22	PKEY.W	X	UNPEND KEY

End of common area between CB and PTBL

Figure 2.2 Process Table Offsets

The CB Unique Offsets

WORD	OFFSET	SUMMARY DESCRIPTION
22	CKEY.W	CB UNPEND key
26	CATCB.W	User TCB address
30	CSTK.W	Frame Pointer
32	CBFEH.W	CB fatal error handler address
36	CSTKC.W	Stack base
40	CPTAD.W	A(PTBL which made system call)
42	CTEMP	Utility
43	CERWD	System call error word
44	CBDLS.W	Dynamic Logical Slot for CB
46	CBULA.W	Logical address for DLS
54	CERPC.W	PC when RETER called
56	CLKPT.W	A(PTBL CB holds PLOCK on)

CBLEN = 60 Length of Control Block

The PSTAT and PSTAT1 bits are defined in the tables below, with an extra column used to show whether a PTBL(P), a CB(C), or both(B) use a particular bit.

PSTAT BITS USED FOR SCANNING

USE	BIT OFFSET	SUMMARY DESCRIPTION
P	0 PSRDY	Not ready to run
B	1 PSRUN	Running
P	2 PSEW	Sched action
P	3 PSNCB	Don't look - process can only use a CB
B	11 PLCK	Process table lock bit
B	14 PTRAN	Element transition bit
P	15 PNTCB	Run only CBs

PSTAT BITS KNOWN AS "PRIORITY BITS"

USE	BIT OFFSET	SUMMARY DESCRIPTION
P	4 PSBRK	OP interrupt
P	5 PSBAG	SWAP OUT process
P	6 PSBLK	BLOCK process
P	7 PSDP	START UP DAEMON
P	8 PSMWT	Wait for memory key to change
P	9 PSTSU	Time slice is up

PSTAT BITS NOT USED BY SCANNER

USE	BIT OFFSET	SUMMARY DESCRIPTION
B	10 PMAST	Mother-only element (1=Mother-only)
P	12 PSETR	Don't enter
P	13 PSFSY	System page fault

PSTAT1 BIT DEFINITIONS

Process table bit parameters for second status word. Note that this is the status word where bits that will not be looked at during the ELQUE dispatch scan are put.

USE	BIT OFFSET	SUMMARY DESCRIPTION
C	15 PSYST	System CB bit -- CM,SM,DM
B	1 PTYP	Element type (1=Control Block)
C	2 PNAD	No address space to release(BPLCK)
C	3 PNFST	Sys call did not run 'FAST'

2.2.2 PTBL/CB Offset Explanations

PLINK.W - This offset is the forward link word for the process table. When this value is not -1, then the PTBL is on a queue. If the value is -1, then the PTBL is at the end of a queue. Typically if the value is -1 and the PTBL is on the ELQUE, then the PTBL is the root process table. In order to modify this offset, the PTBL transition lock must be set. This value is most often modified by queueing instructions.

PBLINK.W- This offset is the backward link word for the process table. When this value is not -1, then the PTBL is on a queue. If the value is -1, then the PTBL is at the beginning of a queue. In order to modify this offset, the PTBL transition lock must be set. This value is most often modified by queueing instructions.

PNQF - This is the Priority Enque Factor word. This word is used for placement on ELQUE. See ELQUE section for calculation. In order to modify this value, the PTBL lock must be held. This value is set at process creation time and is recalculated at time slice end.

The next two words are used during the scan. When doing a scan for a PTBL, the scanner searches ELQUE with a scan mask. The scan mask is tested against the concatination of PCLASS and PSTAT. If a bit in those two words matches the corresponding bit in the scan mask, the PTBL will not be selected.

- Word 1 is a class mask word. If a bit is set, the class canNOT be selected by the scanner (initialize to 0).

- Word 2 is a mask for the PTBL's PSTAT word. If a bit is set, the LP will NOT select the process.

PCLASS - This is the process class word. The PTBL will have the associated bit set for the class it belongs to, so during the scan if that class is masked out this PTBL will not be selected. To modify this entry the Plock must be held.

PSTAT - This is the process status word. It is used during scanning to find out whether to choose this PTBL or not. Not all of the status word is used for scanning. The status bits used are defined as follows:

PSRDY is the "Not Ready To Run" bit. If this bit is set, the process will not be picked by the scanner. The way a process becomes "not ready to run" is if there are no system TCBs to run and there are no user TCBs to run. During a dispatch if the PTBL is found to meet the above conditions then PSRDY is set. Typically a process will have PSRDY set if the process is single threaded and is doing a system call or the process is multitasked and all the tasks are suspended (see process management interface services). The PSRDY bit is cleared after a system call finishes or a task wakes up from a delay.

PSRUN is the running bit. If a process is running on a JP, this bit will be set. This bit is set in the dispatcher immediately after a CB has been found by the scanner. The reason for this is to prevent other JPs from getting the same CB. The bit is cleared if the CB is not able to run. For example: If process is selected to run and one of the priority bits is set then PSRUN is cleared. If a CB is set up on behalf of the caller before the CB runs, the PSRUN bit will be cleared on the PTBL. PSRUN will be cleared if an event occurs such as Subslice End.

PSEW is the scheduler action bit. This bit is set when the system is doing something to the process. For example: If the process is chaining, the PSEW bit will be set to ensure that the process will not be scheduled on another JP while it is chaining. This bit gets cleared when the operation on the process finishes.

PSNCB - 'Don't Look' - This bit is set in a group 2/3 process' PTBL whenever it needs a CB before it can do anything else. The scheduler includes this flag in its SCAN MASK whenever there are no group 2/3 CBs available. Therefore, the system WON'T LOOK at processes that can't proceed because no group 2/3 CBs are available.

Specificaly, there are three instances that cause this bit to be set in a group 2/3 process' PTBL:

1) Attempt to start up a daemon failed for lack of CB.
2) Attempt to get a CB for a TCB waiting to start a call failed and nothing else can be done for the process because its 'Don't Enter' (PSETR, BPSEN) bit is set.
3) Task scheduler finds no ready TCBs, but there are TCBs waiting to start system calls. In this case, 'Don't Enter' (PSETR, BPSEN) is always set, and, if the process is group 2 or 3, PSNCB is also set.

PLCK is the PTBL lock bit. This bit gets set to prevent another processor from accessing this particular CB. This lock is considered a long term lock.

PTRAN is the CB transition lock. When locking a PTBL or CB, PTRAN gets set for the short term then the PLOCK can be obtained. If PTRAN is already set, the JP will spin on the lock (see LOCKING) as soon a PLOCK is obtained and PTRAN is released. This is a short term lock.

PNTCB is the Run only CB bit. When set, this bit will not schedule a process until some CB has Run. This bit is set when trying to dispatch a CB running on behalf of a PTBL. The dispatcher tried to get the PLOCK on the PTBL and couldn't. After setting the bit, the dispatcher goes back to the scanner. The bit gets cleared when the CB dispatcher gets the PLOCK on the PTBL.

The next group of bit offsets do not affect scanning. These are considered the PRBITS (priority bits). If one of these bits is set, the system must do some other action before running this PTBL. After a PRBIT function is performed a reschedule is done.

PSBRK is the operator interrupt flag. This means that the user has done a ^C^B and the process must abort.

PSBAG is the process swap flag. If this bit is set the core manager needs to be woken up to allow the process to be swapped.

If PSBLK is set then process management interface services must be called to put this PTBL on the BLKQ.

If PSDP is set then a daemon must be set up on behalf of this process. Control will go to CB creation when this occurs.

PTSTSU is the Time Slice Up bit. This bit gets set when a system call charge is made and the time slice ends. Control will go to time slice end processing.

The following bit offsets are other status bits not used by the scanner.

PMAST is the "Mother Only" bit. If this bit is set a CB can only run on the mother. If the CB element is a PTBL then the bit in LMAST is also set signifying that there is an outstanding IDEF. If the CB element is a CB, then the CB is performing some task that falls into the mother-only category. (See mother-only calls table.)

PSETR is the "Don't Enter" bit. This bit is set when there are no outstanding system TCBs enqueued in the PTBL extender. This bit gets cleared when a TCB is put on the PSWD.W chain.

PSFSY is the system page fault flag. This bit is not used.

PSTAT1- This offset is not used for scanning. These bits are used to find a characteristic of a CB.

PSYST is used to show that a CB is a system CB. If the CB is a system CB then this bit is set.

PTYP is used to show the type of element we are currently working with. If this bit is set the element we are working with is a CB. Otherwise the element is a PTBL.

PNAD is used to show that the CB is working with a target PTBL. This bit is used by the dispatcher to allow locking of the target PTBL so the CB can work with the target process. This is used when a system call such as ?ISEND is used on a target PTBL. To do the call the caller must get the PLOCK on the target process.

PNFST is used to show whether or not a CB ran "fast." A "fast" running CB is a CB that does not pend.

PPC.W - The control address is the address of the type of dispatcher this element will go to. If the element is a CB, this offset will contain the address of TACT. If this is a PTBL, then the offset will contain the address of PCALL.

PLPCB.W - This offset contains the address of the LPCB to charge time used to. The LPCB that this process is running on has statistics for class scheduling that must be updated after the process runs. The process, however, can run on more than one LP. Therefore, before a process is allowed to run, PLPCB.W is compared to MYLPCB.W, which is the LPCB that the current JP is attached to. If they are different, PLPCB.W is changed to ensure that time used is charged to the correct LPCB. To touch this field the PLOCK must be held.

PGNUM.W - This offset contains the number of the class that this process belongs to multiplied by two. This value is used as an offset into the LPCB class statistics table. The reason that the class is multiplied by two is the table consists of double- word entries. To touch this field the PLOCK must be held.

PTIM.W - This counter contains the current amount of time used in a system call. This value starts at zero when a CB runs. When the PIT interrupts while processing the system call, PTIM.W is incremented by the amount of time used in the current run. To update this field the PLOCK must be held.

TTIM - This counter measures the total amount of time expended on a system call. This value is updated after a CB runs for a period of time (not necessarily to completion). To update this field the system takes the current total and adds it to the contents of PTIM.W. This value is zeroed at CB creation. To update this field the PLOCK must be held.

CALLN.W - This offset is only used if the element is a CB. This value holds the system call number * 2. The reason for this is to help in keeping time accounting for the table of system calls. The reason the contents of CALLN.W is multiplied by 2 is the system call table is a group of double-word entries.

PKEY.W - This offset holds the position for the unpend key for a CB when it is on PELEMQ. This offset is not touched if this is a PTBL. See the CB section for the redefinition of this offset.

This ends the description of the common area between PTBLs and CBs.

2.2.3 CB Unique Offset Explanations

CKEY.W holds the unpend key for a CB. This key, when used, has the value that will be used to unpend the CB. When a CB is not pended, PKEY should be zero. When pend is called, one of the keys will be used as the unpend condition. When unpend is called, all the CBs with matching pend keys will be unpended. For more information on pending, see "Pending" in this chapter.

CATCB.W contains the address of the user TCB. This offset is used to get the address of the packet for a system call. The users TCB contains the necessary state information to get the packet information.

CSTK.W is the frame pointer for this CB. When this CB gets rescheduled this offset will be used to reset the stack. This is considered part of the CB's state information. This is used to reset the frame pointer when the CB is ready to run. This offset is set up when the CB pends.

CBFEH.W holds the address of the trap handler. When used, this contains the starting address of the system call trap handler. If a trap occurs the system will try to go to the trap handler. This prevents the system from panicking on errors that are considered recoverable.

CSTKC.W holds the address of the stack base. This is part of the CB state area. This value is used to reset the stack base when a CB is scheduled to run. This value is set when a CB is going to pend.

CPTAD.W contains the address of the PTBL that made the system call. This offset is used when dispatching a CB to map the associated PTBL so the CB can work with the PTBL. This offset is also used after the CB finishes running to allow the PTBL to either run immediately or allow the PTBL to be scheduled.

CTEMP is a double word used as a utility location. Currently it is used to temporarily store error codes.

CERWD is used to hold the error word if the system call takes an error. This word is used after the system call runs to tell the user that the system call took an error. The error word is put into the AC1 of the process. (See TRTN)

CBDLS contains the address of the Dynamic Logical Slot that will be, or is, currently being used by the CB. (See Memory Management.)

CBULA is not used.

CERPC.W contains the address of the routine that took an error while the CB was running. This is not used.

LKPT.W contains the address of the target PTBL of a call. When a system call, such as ?ISEND, is made the receiver or target of the system call must be held so the information the system call is sending can be sent to the target.

2.2.4 Control Block Pages

The Control Block is a whole page. The reason for this is that the control block needs a page to hold its components.

Each control block page is divided into:

1) Stack prefix area
2) Stack itself
3) Space reserved for stack overflow fault blocks
4) A context block
5) The control block itself

CB stacks have data words relative to the stack base. The offsets and definitions are:

OFFSET	DESCRIPTION	
0	CBSL.W	Stack limit for CB
2	CBCB.W	This CBs address
4	CBCX.W	Fault context block for this CB
6		Stack base
1156		Overflow area
1206		Context block for this CB
1715		The CB

Figure 2.3 The Control Block in Memory

While CB pages are not being used they are on a chain called SSTKQ (System STacK Queue). The pages are attached to the list at the forward and backward link in the CB.

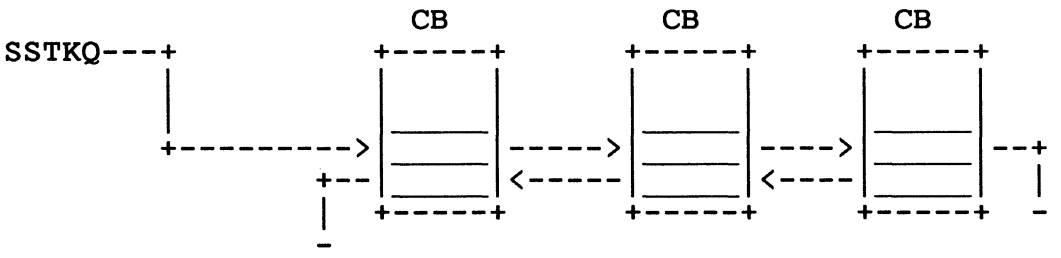


Figure 2.4 CB Pages on SSTKQ

Each control block has associated with it a fault context block, a stack, and a dynamic logical slot. When the control block is selected as the CB to run, the MV's hardware registers are set up to point to the appropriate corresponding values for the control block. The memory needed for a CB is allocated from GSMEM except for the first group 1 control block (CB000), the CMTSK CB, and the system manager task CB.

Unallocated CBs (those not on ELQUE) are enqueued to SSTKQ. (see SSTKQ or CB allocation)

There are four types of control blocks and of these the first two are user CBs: G1; G2/3 or system CBs; and DAEMONs. Each control block has its own stack. There are three system CBs: the disk manager, core manager, and system manager.

2.2.5 Types of CBs

There are five CBs discussed in this section: user CBs (G1 and G2), disk manager, core manager, system manager, and daemons.

2.2.5.1 G1 and G2/3 CBs

G1 and G2/3 control blocks are used when a G1 or G2/3 process makes a system call. For example, a G2/3 CB is used when a G2 process makes a system call.

2.2.5.2 Disk Manager

The disk manager runs as the highest priority control block on the ELQUE. The disk manager runs all IOCBs when they are ready to run. When N_T_RUN.W is set, the scheduler branches to RUNLC in DSKIO to schedule IOCBs. All active IOCBs are run. As long as there are ready IOCBs, they are run. When there are no more ready IOCBs, the disk manager control block is changed by setting the 'not ready to run' bit and resetting the running bit. The disk manager is readied by a call to the routine DWAKE.

2.2.5.3 Core Manager

The core manager runs as the second-highest priority control block on ELQUE. The core manager manages memory. It remains dormant until a code path calls the routine CWAKE, which sets the `ready to run bit' in the status word for the CB. In addition, it sets the words or bits needed to indicate which action the core manager should process when it gets control of the CPU. Requests to the core manager are indicated by SMFLG.

2.2.5.4 System Manager

The system manager is currently used for five purposes. The first is to report device errors, the second is to report over-subscribed memory, the third is to enable look-ahead faulting, the fourth is to enable look-ahead flushing, and the fifth is to handle SCP error reporting. The unit errors are detected by the controllers, which set up error status words in the appropriate UDB (unit device block). The error routine then calls SWAKE, which will cause the system manager to wake up the next time a scan is made of the eligible queue (much as CWAKE does for the core manager). The over-subscribed memory condition (no memory available, no pre-emption possible) is detected by the pre-emption code, which then calls SWAKE. The system manager reports the error to error-log.

The requests to the system manager are indicated in CMFLG. (The core manager equivalent is SMFLG)

2.2.5.5 Daemons

A daemon can be considered an AOS/VS initiated system call (as opposed to the user oriented or standard call). When AOS/VS needs something done, and the code path required might pend, AOS/VS will use a daemon for the processing.

Daemons are currently used for the following:

1. Process terminations (Four types: normal, trap, fatal error, and ^C^B)

2. Process initial load. (The path can pend waiting for the disk)

3. Process a 16-bit process changing to/from resident (we will wire in the pages of a resident 16-bit process)

4. Process keyboard interrupts (other than ^C^A)

Daemons are started by setting the request daemon bit in PSTAT and running off control blocks. They can be identified by examining offset CATCB.W (12) of the CB. It will contain a 0.

2.2.6 Primary, Secondary and Temp CBs

This section describes the use of these CB Pointers. These point to currently used CBs in the system.

2.2.6.1 The Primary CB

The Primary CB (PCB) is the stack that each JP always uses while it is running system code. System code can be either a control block or the Scheduler. The Primary CB is pointed to by PCB.W. When a CB pends, the Primary CB is put onto the Pended element queue (see Pending). The primary CB is allocated during system initialization. The only time a primary CB gets released is when a JP is released from the system.

2.2.6.2 The Secondary CB

When a path pends the Primary CB is put onto PELEMQ and the code goes to the Scheduler. The problem is that if the Primary CB is on PELEMQ then the system needs a new primary CB to be able to run the scheduler. This new primary comes from the secondary CB (SCB.W). The secondary CB is the backup stack. The secondary CB is pointed to by SCB.W. When a system call is made the secondary CB must be allocated (see CB allocation). If the allocation attempt fails for any reason, then the system call will not run until the system can allow allocation of a CB. This mechanism, although it may sometimes penalize the user, allows the system to efficiently go from a pended code path to the Scheduler to get another element. The figure below shows the exchange from secondary to primary.

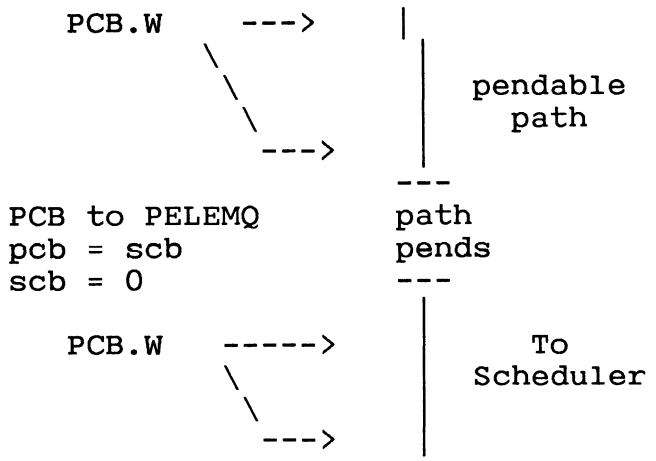


Figure 2.5 PCB/SCB Exchange

2.2.6.3 The Temp CB

When the system dispatches a CB, it must somehow make that CB the Primary. This is done by assigning the CB to PCB.W. But what happens to the Primary CB? The primary cannot be lost! Therefore, what the system does is assign PCB.W to SCB.W. But what if SCB.W is already defined? The solution is the Temporary CB (TMPCB). This CB is pointed to by TMPCB.W. This is a temporary holder for the CB if SCB is already defined. It is the first choice for becoming the primary CB. Therefore, in the previous figure the part that shows PCB.W = SCB.W actually looks like.

```
if (TMPCB.W != 0)
     PCB.W = TMPCB.W;
else
     PCB.W = SCB.W;
```

The temp CB is the only CB that gets deallocated when a system call completes.

2.2.7 The CB Management Globals

The CB management Globals are used to either work with CBs or are used as counters. The Globals in this section are broken into three groups: the general purpose globals, the event count globals, and the collision counters.

The general globals work with CBs. They are: IDSCBQ, SSTKQ, SSTKCT, RSTKCT, SSTUSE, RSTUSE, SSTMAX, RSTMAX, SSTMIN, RSTMIN, PCB.W, SCB.W, and TMPCB.W.

IDSCBQ -- Idle System Control Block Queue

When a system CB puts itself to "sleep," it puts itself on the IDSCBQ. Putting as system CB to "sleep" is effectively the same as pending, with a major difference. That difference is that a system CB puts itself to sleep only if it has nothing more to do. For more information on system CBs, see Memory Management and the File System volumes of this manual.

The System STacK Queue (SSTKQ) is a minor Queue of CBs. SSTKQ Queue is called the free control block Queue and is also referred to as the "CB pool." This Queue contains a linked list of all the allocated but unused CBs in the system. When the system needs a CB, it is provided from this queue. This Queue is created at SINIT time when the system allocates the CBs. This queue contains all unused CBs irrelevant of type. This queue is managed by two counters. Those counters are SSTKCT (Swappable STacK CounT) and RSTKCT (Resident StacK CounT). The combined values of these counters add up to the total number of CBs that are in SSTKQ.

SSTKCT (Swappable STacK CounT) is the counter used to manage the G2/3 CBs in the CB pool. During system initialization, the system allocates CBs until SSTKCT is equal to the minimum number of G2/3 CBs. This number is currently 15 (octal). This number can increase to a maximum of 200 (octal).

RSTKCT (Resident STacK CounT) is the counter used to manage the G1 CBs in the CB pool. During system initialization, the system allocates CBs until RSTKCT is equal to the minimum number of G1 CBs. This number is currently 5. This number can increase to a maximum of 100 (octal). There must always be at least one free G1 CB in the free pool for PMGR.

SSTUSE is a counter of all the currently allocated Group 2/3 CBs. It is used to keep track of the CBs that have been allocated but are not in use.

RSTUSE is a counter of all the Group 1 CBs that have been allocated.

SSTMAX is the maximum number of Group 2/3 CBs that can be allocated. This constant is set to 200. (decimal). When allocating CBs the allocation code will check whether another CB should be allocated or the requestor should wait for a CB.

RSTMAX is the maximum number of Group 1 CBs that can be allocated. This constant is set to 100. (decimal). When allocating CBs the allocation code will check whether another CB should be allocated or the requestor should wait for a CB.

SSTMIN is the minimum number of Group 2/3 CBs that must be allocated. This constant is 10. When the de-allocation routine DALCB1 is called to de-allocate CBs, the CBs are de-allocated to SSTMIN.

RSTMIN is the minimum number of Group 1 CBs that must be allocated. RSTMIN is a constant with the value 5. During de-alloction, DALCB1 de-allocates to RSTMIN.

PCB.W contains the pointer to the Primary CB.

SCB.W contains the pointer to the Secondary CB.

TMPCB.W contains a pointer to a Temporary CB.

The next group of globals are event counters. An event counter is used to count the number of significant events that happen to a CB. For example, if a CB pends the appropriate counter is updated.

NUNPDN.W counts the number of times UNPNDN is called.

NUNPD.W counts the number of times UNPEND is called.

NPEND.W counts the number of times a CB pends.

NCBDEQ.W counts the number of times CBDEQ was called.

RSTDAL.W counts the number of G1 CBs that get de-allocated.

SSTDAL.W is the counter for the total number of G2/3 CB de-allocations.

CTRTN.W counts the ELQUE collisions in the CB return routine TRTN.

2.3 Operations on CBs

There are three basic types of operations that work with the objects in CB management: CB allocation/de-allocation, pending/unpending, and FIXCB.

2.3.1 CB Allocation

When a user does a system call the system must allocate a CB to do the system call. Most of the time there are enough CBs in the CB pool so there is no need to allocate the CB. If there are no "known" CBs in the CB pool and memory management cannot allocate more memory for CBs, then a TCB is attached to the PTBL of the caller to wait for a CB to become available.

The system has two types of allocation routines used to manage the free pool. The first one manages the counters and the second manages the queue. ALSTK1 and ALSTK2 are used to get a CB from the free pool and update the counters. The routine called differs depending upon the group of the caller. If the CB is allocated on behalf of a GROUP 1 process, then ALSTK1 will be called. If the caller is a group 2 or 3 process, then ALSTK2 is called. If the counters go below the minimum amount of free CBs, then these routines call a routine to add to the queue.

The system has two routines that add CBs to the free chain. They are called ALCB1 and ALCB2. The routine called differs depending upon the group of the caller. If the CB is allocated on behalf of a Group 1 process, then ALCB1 will be called. If the caller is a Group 2 or 3 process, then ALCB2 is called.

ALCB1/ALCB2 allocates a CB and stack space for the CB. To do this the routines call memory management to get the memory for the CB. If there is no memory available to allocate a CB, then the CB allocation routines will take an error.

The CB allocation routines are called from three places in the system. Two of the routines are called at system initialization. The first one calls ALCB2 to set up an extra CB for the JP in the system. The second allocates CB up to the minimum number of CBs allowed. The third time the allocation routine is called is when the Secondary CB is needed to run a CB. The allocation routine will be called if the number of free blocks in the pool is less than the minimum number of allowable blocks in the pool.

The main reason for two separate allocation routines is accounting. AOS/VS updates different counters depending upon the Group the caller is from.

After a CB completes, the CB must get de-allocated. To de-allocate a CB, the routines DALSTK1 and DALSTK2 are called by TRTN depending on the caller. TRTN will only de-allocate the CB in TMPCB.W. Otherwise, the CB will not get de-allocated but instead is returned to the SSTKQ pool.

During memory contention, memory management will take CBs from the free pool. To do this the core manager will call DALCB1 and DALCB2 to de-allocate pages until the minimums are reached. Calling the de-allocation routines frees up a few pages of memory for use by the users.

The figure below shows the stages of a CB from being free memory through the different allocation schemes and back to free memory.

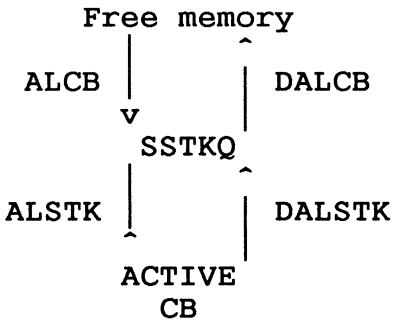


Figure 2.6 CB Allocation

2.3.2 Pending

Pending is a way to stop a CB that has to wait for some condition so the system can run other paths. An example of a pendable CB is an I/O operation, where the code has to wait for the device to respond to the I/O request.

When a CB pends the PEND routine is called. PEND is given a key word that represents the reason this CB pended. This pend key must be satisfied before the CB can be unpended. PEND takes the currently running CB and puts it onto the Pended ELeMent Queue (PELEMQ) to wait for the pend condition to be satisfied.

Below is a table of all the predefined pend keys used when a CB pends.

SKTRM	== 1	Wait for son to term
SKTRG	== 2	Wait for target call completion
SKOOM	== 4	Task waiting for memory
SKSWP	== 5	Waiting for swap to finish
SKSIO	== 6	Wait for shared read (not used)
SKBUF	== 7	Base level waiting for a system buffer
SKDED	== 10	Wait for special unpend (not used)
SKNWU	== -1	Never wake up key (not used)
CPLCK.W	***	JP lock word (see JP management)

Figure 2.7 System Pend Keys

When a CB pends the CB is put onto PELEMQ. This means, if necessary, taking the CB off of ELQUE and putting it onto PELEMQ. There are four routines that perform the pending/unpending operation. These operations do all the necessary locking for enqueueing and dequeueing the queues. The operations are: PEND, MPEND, UNPEND, and UNPNDN. PEND and MPEND put CBs onto PELEMQ. UNPEND and UNPNDN take CBs off of PELEMQ and put them onto ELQUE or HANDQ (see ELQUE management for information on ELQUE and HANDQ).

The pseudocode below shows how these operations work.

2.3.2.1 PEND/MPEND

This operation pends the current CB. It puts the CB onto the PELEMQ. The routine also sets the pend condition into CB-> CKEY. If the CB is on ELQUE, move the CB from ELQUE to PELEMQ. This routine goes to a routine called PEND0. PEND0 is an entry point that is used by PEND and MEPEND.

```
/* $$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$ */
/*                      PEND/MPEND                                */
/* This routine puts a CB onto the PELEMQ to wait for             */
/* whatever unpend condition passed to the routine.               */
/*                                                                */
/* MPEND gets called when a lock is held and the CB               */
/* is pending.  MPEND will release the lock before                */
/* pending.                                                       */
/* $$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$ */

pend(unpend_key)/mpend(xtran,unpend_key,xtran_address);
     {
     cb=CC.W;       /* get the current CB */
     NPEND.W ++;    /* one more pend      */

/* ************************************************** */
/*  If this is a CB that never pended then set the    */
/*  CB up so it can pend.  To do this operation the   */
/*  FIXCB routine is called.  If plnk.w is not -1 then*/
/*  the CB has been on some queue before and therefore*/
/*  has pended.                                       */
/* ************************************************** */

     if (cb->plnk.w == -1)
        fixcb(cb);
     interrupts(disable);

/* ************************************************** */
/* If there is not a valid pend key in the CB then    */
/* panic with a 14434.  A valid pend key is either 0  */
/* or the same as the value passed as an argument to  */
/* this routine.                                      */
/* ************************************************** */

     if (cb->ckey.w !=0) && (cb->ckey != pend_key)
           panic(14434);
     cb->ckey.w = pend_key;
     set_bit(cb->pstat,psrdy);
```

```
/* ************************************************ */
/* If the CB was on ELQUE then move it from ELQUE to  */
/* PELEMQ.  If not then ENQUE to PELEMQ at the tail.  */
/*                                                    */
/* ************************************************ */

     if (cb->plnk.w != -1)
        qmov(elque,cb,pelemq);/* see ELQUE Management */
     else
        enqt(pelemq,cb);

/* ************************************************ */
/*  If the routine MPEND was called then clear the    */
/*  xtran bit in the lockword.  xtran is a transition */
/*  lock of any kind.  It is called "x"tran because   */
/*  the routine cannot define which tran lock is used.*/
/* ************************************************ */

     if (mpend)
        clear_bit(xtran_lockword,xtran);

/* Falls through to next page                         */


/* ************************************************ */
/*                      pend0                       */
/* This is an entry for CBs being "handed off" to the */
/* mother processor.  This routine releases the ptbl  */
/* locks, makes the secondary CB a primary CB, and    */
/* returns to the top of the scheduler.               */
/*                                                    */
/* ************************************************ */

pend0:

     interrupts(enable);
     cb = cc.w;

/* ************************************************ */
/* If this CB is not a system CB then check to see if */
/* we need to release the process table locks.        */
/* If we need to release the PTBL locks call relptbl; */
/*                                                    */
/* ************************************************ */
```

```
     if (bit(cb->pstat1,plock) ==0)
        {
        if (bit(cb->pstat1,pnad) != 0)
           cb->clkpt.w = cmap.w;
        else
           ptbl =  cb->cptad.w;
        if (ptbl != 0)
           relptbl(ptbl);
        }
     tcsys();/* go from CB timing to system timing */
     clear_bit(ptbl->ptbl,psrun);

/* ************************************************ */
/*  If there is a temporary CB then make that CB    */
/*  the primary CB.  If there is no temporary CB then  */
/*  make the secondary CB the primary.              */
/*                                                  */
/* ************************************************ */

     if (TMPCB.W != 0)
        {
        PCB.W  = TMPCB.W;
        TMPCB.W = 0;
        }/* tmpcb.w */
     else
        {
        pcb.w = scb.w;
        scb.w = 0;
        }/* else */

/* ************************************************ */
/* Set up the fault block for the primary CB and    */
/* go to the top of the scanner.                    */
/*                                                  */
/* ************************************************ */

     FLTBLK.W = SYSTEM_FLTBLK.W
     SET_UP_STACK();
     GOTO scanner();
     } /* pend */
```

2.3.2.2 Unpending

Unpending a CB allows the path to be scheduled. A CB can be unpended only if the pend condition is satisfied. There can be multiple CBs pended on the same type of condition. Unpend does not check indiviual conditions, it just unpends every CB with a particular condition. The CB that is unpended has the responsibility of checking whether it is the right CB to unpend.

For example: a CB is doing an I/O operation that causes it to pend. There are other CBs pended on some I/O operation. An I/O operation completes, which causes the related CBs to unpend. The operation completing only satisfied one CB. Therefore, the other CBs must pend again.

There are two unpend routines. These routines are called UNPEND and UNPNDN. UNPEND unpends all the CBs that match the pend conditions. UNPNDN unpends a certain number of CBs.

Below is the pseudocode for UNPEND and UNPNDN.

```
/* $$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$ */
/*  UNPEND(unpend_key)/UNPNDN(unpend_key,max_unpend)                  */
/*                                                                    */
/*  These routines unpend the elements that satisfy                   */
/*  the unpend condition(unpend_key).  UNPNDN unpends                 */
/*  CBs up to the max number allowed(max_unpend).                     */
/*                                                                    */
/* $$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$ */


unpend(unpend_key)/unpndn(unpend_key,max_unpend);
     {
```

```
/* ************************************************ */
/*                  The local vars                  */
/* ints_test is used to reflect whether interrupts are*/
/*   on(<> 0) of off (== -1).                       */
/* count_bef_int is used to count the number of     */
/*   elements to work with before interrupting.     */
/* count_test is used to check if the above count   */
/*   should be used.(<> 0 == use the count)         */
/*                  (=  0 == don't use the count)   */
/* temp is a general purpose variable.              */
/* ************************************************ */

#define loopcnt = 5;  /* loop counter for max number  */
                      /* of iterations through this   */
                      /* code with interrupts off.    */
     int   ints_test,
           count_bef_int,
           count_test,
           temp;

/* ************************************************ */
/* If we came in through unpend or unpndn then      */
/* different initialization is done.                */
/* Check if interrupts were off when we came in so we */
/* can leave with the same interrupt state as we came */
/* in.                                              */
/* ************************************************ */

     if (unpndn)
        {
        count_bef_int = max_unpend;
        if (interupt  == off);
           ints_test = -1;
        else
           ints_test = max_unpend;
        NUMPDN.W ++;
     else
        {
        count_test = 0;
        ints_test = loopcnt;/* non zero */
        if (interrupt == off)
           ints_test = -1;
        NUMPD.W ++;
        }

     interrupts(disable);
     get_q_lock(pelemq);
```

```
/* ************************************************ */
/*                  top of the loop                 */
/* This is the top of the unpending loop.  This     */
/* section of code will unpend all CBs that match the */
/* unpend condition passed to the routine.          */
/* ************************************************ */

top_of_loop:
     if (PELEMQ != -1) /* is PELEMQ empty?  */
        {
        temp = pelemq;
        for (i=0;i<=count_bef_int;i++)
           {

/* ************************************************ */
/*  If there are no more elements then break out of */
/*  the unpend loop.                                */
/*                                                  */
/* ************************************************ */

           if (temp = -1)
              break;
           cb = 0;
           cb = searchq(pelemq,unpend_key)

/* ************************************************ */
/* If we found a CB then call CBDEQ to dequeue the  */
/* element from the  PELEMQ and enqueue it to ELQUE. */
/* This is done in a routine called CBDEQ.  If no CB */
/* was found break out of the unpend loop.          */
/* ************************************************ */

           if (cb != 0)
              {
              temp = cb->plink;
              cbdeq(cb);
              }
           else
              temp = -1;
           count_test --;
           } /* for loop   */
```

```
/* ************************************************ */
/* If we aren't ingoring the interrupt limit and    */
/* we have more elements to hunt for then enable    */
/* interrupts for a short while and come back to    */
/* unpend more elements.                            */
/* ************************************************ */

        if ((count_test == 0) && (temp != -1))
           {
           release_q_lock(pelemq);
           interrupts (enable);
           interrupts (disable);
           goto top_of_loop;
           }/* if */
        }  /* if not pelemq empty */
     release_q_lock(pelemq);
     if (ints_test != -1)
        interrupts(enable);
     return( );
     }/* end of unpend */
```

2.3.3 FIXCB

FIXCB is a routine that sets up a CB to go onto the Eligble queue. The reason this routine is needed is that a CB does not need to go onto ELQUE unless it pends. FIXCB is an entry point in SCHED. There is another part of FIXCB that is not an entry point, but is called from other parts of SCHED. This entry is called FIXCB1. The difference between FIXCB and FIXCB1 is that FIXCB actually enqueues the CB onto ELQUE and FIXCB1 does not. FIXCB is called from the PTBL dispatcher (see process management interface services) in the routine NODCL to enqueue a new CB that cannot run immediately onto ELQUE. FIXCB1 is called from pend. (See Pend code.) Very little of the CB is changed by Fix CB. Most of the CB is set up by the code that creates the CB. The code that sets up the CB is in the PMIS chapter.

```
/* $$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$ */
/*                    FIXCB/FIXCB1(CB)                           */
/*  This routine sets up a CB to go onto ELQUE.  If              */
/*  FIXCB is called then put the CB onto ELQUE.                  */
/* $$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$ */

FIXCB/FIXCB1(cb)
     {

/* ************************************************************ */
/* Is the CB a DAEMON?  If it is then set the "mother           */
/* only" bit in the pstat word and clear PCLASS in              */
/* the CB.                                                      */
/* ************************************************************ */

     if (cb->calln.w = -1)
        }
        setbit(cb->pstat,pmast);
        cb->pclass = 0;
        }
     setbit (cb->pstat1,pnfst);

/* ************************************************************ */
/* If we came in through FIXCB1 then we do not want             */
/* the CB put onto ELQUE.  So just return.                      */
/*                                                              */
/* ************************************************************ */

     if (FIXCB1)
        return;
```

```
/* ************************************************ */
/* We have to be sure that interrupts are turned off  */
/* before enqueuing to ELQUE. Therefore, if interrupts*/
/* are on, turn them off before calling PENQ and turn */
/* them back on afterward.                            */
/*                                                    */
/* ************************************************ */

     if (interrupts == off)
        PENQ(ELQUE,cb);
     else
        {
        interrupts(off);
        PENQ(ELQUE,cb);
        interrupts(on);
        }  /* else */
     return;
     }/* FIXCB/FIXCB1 */
```

2.4 Internal Paths

There are three general internal paths that CB management uses: the scanner, the dispatcher, and the idle loop.

2.4.1 The CB Dispatcher

When a CB is selected to run by the scanner, the CB dispatcher routine sets up and runs the CB. This dispatcher is divided into two different entry points, one for the system CBs(TACT1) and the other for user CBs(tact). The only time this dispatcher is used is if a CB has pended or put itself to sleep (see system CBs).

This dispatcher restores the state of a CB and runs it. The pseudocode below shows how TACT and TACT1 work.

```
/* $$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$ */
/*         TACT(PPCB,CB) /  TACT1(PPCB,CB)                           */
/* This routine restores the state of a CB, maps its                 */
/* PTBLs memory, and runs the CB.  The arguments                     */
/* supplied to this routine are the PPCB (see JP                     */
/* Management) and the CB address.                                   */
/* $$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$ */

tact(ppcb,cb)/tact1(ppcb,cb)
     {

/* ********************************************** */
/*  Test and set the running bit for the CB.      */
/*  If the CB is already running then go get another.*/
/*  Leave this routine and go to RUNEX1.          */
/*                                                */
/* ********************************************** */

     if (bit(cb->pstat,psrn)
        runex1();
     else
        set_bit(cb->pstat,psrn);

/* ********************************************** */
/*                                                */
/* Came in through TACT1.                         */
/*                                                */
/* ********************************************** */

     if (tact1)
        {
        ptbl = cb->cptad.w;
        }
```

```
/* ************************************************ */
/*                                                  */
/* Came in through TACT.                            */
/*                                                  */
/* ************************************************ */

     else
        {

/* ************************************************ */
/* Is the CB dealing with a target PTBL?  If so,    */
/* the PTBL to be locked is in CLKPT.W in the CB.   */
/* If not, then use the PTBL in CPTAD.W in the CB.  */
/* ************************************************ */

        if (bit(cb->pstat1,pnad) == 0)
           ptbl = cb->cptad.w;
        else
           ptbl = cb->clkpt.w;

/* ************************************************ */
/* Is there a PTBL defined?  If not then the CB deals*/
/* with the system and there is no need to try to   */
/* lock it.  If there is a PTBL defined then lock it.*/
/* ************************************************ */

        if (ptbl != 0)
           {

/* ************************************************ */
/* Is the target PTBL locked?  If so,  then decrement*/
/* the "tried to get the lock counter" and return to */
/* the scanner.                                     */
/* ************************************************ */

           if (bit(ptbl->pstat,plck) == 1)
              {
              clear_bit(cb->pstat,psrun);
              ptbl->pcblk --;

/* ************************************************ */
/* If we have tried to lock this PTBL more than 500 */
/* times, then set the "run only CB" bit in the PTBL.*/
/* This will allow the CB to run on the target PTBL. */
/* This happens when the PTBL is locked and we can't */
/* get the lock.                                    */
/* ************************************************ */
```

```
            if (ptbl->pcblk != 0)
               {
               set_bit(ptbl->pstat,pntcb);
               tcblck.w ++;
               }/* if count is zero */
            runex1();
            } /* if ptbl locked */
         else
            {
               ptbl->pcblk = 500;
               clear_bit(ptbl->pstat,pntcb);
            }/* not locked */
         }/* there is a ptbl address */
      }/* came in through tact. */
   cc.w = cb;
   elque.qscan --;
   tcscl(cb);/* change to cb timings */
   interrupts(enable);

/* ************************************************ */
/*  If the current LP is not the same as the LP in   */
/*  the CB, then do some work to set up the correct  */
/*  LPCB in the CB.                                  */
/* ************************************************ */

   if (mylpcb.w != cb->plpcb.w)
      {

/* ************************************************ */
/* Is this a mother-only process?  If it is, then the*/
/* scanner took care of mother only by adjusting     */
/* the scan mask. Therefore, if this is a mother-only*/
/* process and we have the CB, then we must be the   */
/* mother.  If not, call CBUPDT and put the MYLPCB.W */
/* into the CB.                                      */
/* ************************************************ */

      if (bit(cb->plpcb->lpstat,pmst) != 1)
         {
         cbupdt(cb);
         cb->plpcb = mylpcb;
         }
      }
```

```
/* ********************************************** */
/* Is there a secondary CB? If there is no secondary */
/* CB defined, then define the primary to be the     */
/* secondary and the primary will be the current CB. */
/* If it is defined, use temp CB to hold the Primary */
/* CB so we don't destroy the secondary.             */
/* The CB in tmpcb.w will run before the CB in SCB.W.*/
/* ********************************************** */

     if (scb.w != 0)
        tmpcb.w = pcb.w;
     else
        scb.w = pcb.w;
     pcb.w = cb;
     mapcon(ptbl);

/* ********************************************** */
/*  Now set up the user stack and run the CB.        */
/*                                                  */
/* ********************************************** */

     interrupts(disable);
     stack_ptr = cb->cstk.w;
     frame_ptr = cb->cstk.w;
     stack_base = cb->cstkc.w;
     stack_limit = cb->cstl.w;
     fbk.w = cb->cbcx;
     interrupts(enable);
     return(); /* returns to call to pend */
     }/* tact/tact1 */
```

2.4.2 TRTN/TGRTN

When a CB finishes running it does a WRTN, either TRTN (for error returns) or TGRTN (for good returns). The routine cleans up the CB and either returns to the PTBL that made the call or returns to the scanner.

```
/* ************************************************ */
/*                    TRTN/TGRTN()                  */
/* This routine is used when returning from running */
/* a CB or DAEMON.  This routine cleans up the CB   */
/* and returns to either the appropriate PTBL or    */
/* to the scanner.                                  */
/* ************************************************ */

trtn/tgrtn()
     {

/* ************************************************ */
/*  If we entered through TRTN, do the error rtn    */
/*  processing.  Check to see if the TCB is a       */
/*  DAEMON; if not, check for a restart.            */
/* ************************************************ */

     cb = CC.W;
     tcb = cb->catcb;
     if (trtn)
        {
        error = cb->cerwd;
        if (tcb != 0)
           {

/* ************************************************ */
/* The CB is not a daemon.  If the TCB needs to do a */
/* restart to get memory, the daemon must enqueue   */
/* the TCB onto the tcb chain in the process table. */
/* Set the MKEY priority bit in the PTBL and set the */
/* global memkey so the PTBL will not get scheduled */
/* until some memory gets freed.  Once the memory   */
/* gets released the global key will be reset to    */
/* allow the PTBL to be rescheduled.                */
/* ************************************************ */
```

```
            if (error == ERRST)
               {
               ptbl = cb->cptad.w
               setbit(ptbl->pstat,psmwt);
               clearbit(ptbl->pstat,psncb);
               PMKEY = MKEY;
               IRSTRT ++;
               ptbl->psidir ++;
               setbit(cb->pstat1,pnfst);
               NQTCB(cb);
               }

/* ************************************************ */
/*                                                  */
/* The CB is not a daemon and not doing a restart.  */
/* The CB is taking an error.  Set up the error     */
/* return and go to TGRTN to complete the cleanup for */
/* the CB.                                          */
/* ************************************************ */

            else
               {
               tcb->tac0.w = error;
               tcb->tpc.w --;
               goto tgrtn( );
               }
            }
         }

/* ************************************************ */
/* If we came in through TGRTN, then the CB completed */
/* its work and now we must set up to try to return */
/* to the calling PTBL or the SCANNER.              */
/* ************************************************ */


      if (tgrtn)
         {

/* ************************************************ */
/*                                                  */
/*  If we are working with a daemon, clear the pend */
/*  bit in the TCB. (For explanation of TCBs, see   */
/*  Process Management Interface Services.)          */
/*                                                  */
/* ************************************************ */

         if (tcb == 0)
            clearbit(cb->tstat,tspn);
         }
```

```
/* ************************************************ */
/*  Is the CB on ELQUE?  If so, remove the element  */
/*  from the queue.                                 */
/* ************************************************ */

     if (cb->plink != -1)
        pdeq(ELQUE,cb,CTRTN.W);
     cb->cbtp1.w = 0;
     cb->cbtp2.w = 0;
     TCSYS();
     CBUPDT(cb);/* see LP management */

/* ************************************************ */
/* Update the CPU time used during the system call. */
/* Do this in the routine called CBSTAT.  CBSTAT takes*/
/* the time used and adds the cputime used to the   */
/* corresponding offset in the system call counter  */
/* table.  This table is a Global table whose address */
/* is in SSTBL. SSTBL.w is a pointer to a table of  */
/* system call timing counters.                     */
/*                                                  */
/* ************************************************ */

     If (STTBL.W != 0)
        CBSTAT (cb);

/* ************************************************ */
/* If we used a temporary CB, release the CB to the */
/* free queue.  This frees up TMPCB.w cleanup so when */
/* another call is made the tepm is available for use.*/
/* ************************************************ */

     if (TMPCB.W != 0)
        enqt(SSTKQ,cb);

/* ************************************************ */
/* Update the counters for G1 or G2 CB usage.  This */
/* allows the system to keep track of what kind of  */
/* CBs use system resources.                        */
/* ************************************************ */

     if (PCB.W->pnqf  == 0)
        RSTKCT ++;
     else
        SSTKCT ++;
     ptbl = cb->cptad.w;
```

```
/* ************************************************ */
/* When the term daemon runs, the high-order bit of   */
/* the PTBL address in the CB is set.  So to work     */
/* with the PTBL databases we must fix the address    */
/* which is invalid by clearing the high-order bit.   */
/*                                                    */
/* ************************************************ */

     if (ptbl <= 0)
        {
        clearbit (cb->pstat1,pnfs);
        FBK.W = FLTBK.W;
        ptbl &= '177777777777';
        clearbit (ptbl->pstat,psrun);

/* ************************************************ */
/* Does the CB hold an address space lock?  If so,    */
/* call RELPTBL (see Memory Management) to release the*/
/* PTBL address space lock.  If not, just clear the   */
/* "no address lock" bit and go to the top of the     */
/* SCANNER which is at SMON0.                         */
/* ************************************************ */

        if (bit(ptbl->pstat1,pnad) == 1)
           clearbit(ptbl->pstat1,pnad);
        else
           RELPTBL(ptbl);
        goto SCANNER;  /* SMON0  */
        }

/* ************************************************ */
/* The CB is not a DAEMON.  Did the user do a Parallel*/
/* Call?  If so, clear the bit and check for waiters. */
/* ************************************************ */

     if (bit (ptbl->pflag3,pfpch) == 1)
        {
        clearbit(ptbl->pflag3,pfpch);

/* ************************************************ */
/* Are there any waiters on the Parallel call that    */
/* PTBL did?  If so, unpend the waiters and continue  */
/* CB cleanup.                                        */
/* ************************************************ */
```

```
        if (bit(ptbl->pflag5,pfpchw);
           {
           clearbit(ptbl->pflag5,pfpchw)
           unpend (ptbl,cb);
           }
        }

/* ************************************************ */
/* Did the CB run "FAST"?  Running "fast" means the CB*/
/* did not pend.  If so,  check the TCB to see if it  */
/* ran fast.                                          */
/* ************************************************ */

     if (bit(cb->pstat1,pnfst) == 0)
        {
        tcb = cb->catcb;

/* ************************************************ */
/* We have a TCB address from the CB.  Is there       */
/* an address?                                        */
/* ************************************************ */

        if (tcb != 0)
           {

/* ************************************************ */
/* There is a TCB address.  Did the TCB run "fast"?   */
/* A TCB runs "fast" if it does not have to wait for  */
/* memory.  If the TCB did not run "fast" clear the   */
/* bit and continue cleanup to go to the Scanner.     */
/*                                                    */
/* ************************************************ */

           if (bit(tcb->tcbfl,?scfast) == 1)
              {
              clearbit(tcb->tcbfl,?scfast);
              } /* if tcb ran "FAST" */
           else
              {

/* ************************************************ */
/* Both the CB and the TCB ran "fast", so start       */
/* setting up to run the user.  Check if the user did */
/* not use up their time slice.  If not, run the user.*/
/* ************************************************ */

              if (bit(ptbl->pstat,pstsu) == 0)
                 goto run_user;
              }/* else */
           } /* have a TCB */
        else
           {
```

```
/* ************************************************ */
/* We weren't running a TCB.  Then is this TCB a    */
/* DAEMON?  If so, continue and check the reschedule */
/* flag.  If not, drop out to do the cleanup before */
/* going to the scanner.                            */
/*                                                  */
/* ************************************************ */

          if (cb->calln.w == -1)
             {
             ppcb = myppcb;

/* ************************************************ */
/* Did something else on the system request a       */
/* reschedule?  If not,  go RERUN the daemon.       */
/* ************************************************ */

             if (bit(myppcb->cpstat,resch) == 0)
                {
                setbit(ptbl->pstat,psrun);
                CC.W = ptbl;
                ptbl->pextn.w->psqct --;
                goto RERUN;
                }
             }
          }/* no TCB    */
       }/* if not fast */
    clearbit(cb->pstat1,pnfst);

/* ************************************************ */
/*  This label is jumped to from the run_user routine */
/*  to go to the top of the scanner if there are any  */
/*  conditions that keep the PTBL from being run.     */
/* ************************************************ */

to_scanner:
    ptbl->pextn.w->psqct --;
    clearbit(ptbl->pstat,psetr);
    clearbit(ptbl->pstat,psncb);
    clearbit(ptbl->pstat,psrdy);
    clearbit(ptbl->pstat,psrun);
    setbit(ptbl->psflag,pfrsh);
    FBK.W = FLTBK.W;

/* ************************************************ */
/* Did the user use up its time slice?  If so, set up */
/* to go to TSUP (time slice end processing).       */
/* ************************************************ */

    if (bit(ptbl->pstat,pstsu) == 1)
       {
```

```
/* ************************************************ */
/* Is there an address space lock?  If not, check for */
/* PLOCK.                                           */
/* ************************************************ */

        if (bit (cb->pstat1,pnad) == 0)

/* ************************************************ */
/* Time slice has expired.  Test for PLOCK.  If held, */
/* clear the lock bits and the time slice up bit.   */
/* Go to TSUP.                                      */
/* ************************************************ */

           if (bit(ptbl->pstat,plock)  == 0)
              {
              setbit(ptbl->pstat,plock);
              clearbit (cb->pstat1,pnad);
              }
        clearbit(cb->pstat,pstsu);
        goto TSUP(ptbl);
        }

/* ************************************************ */
/* If the Address space lock is being held on this  */
/* PTBL then go unlock it.                          */
/* ************************************************ */

     if (bit(cb->pstat1,pnad) == 0)
        RELPTBL(cb,ptbl);
     clearbit(cb->pstat1,pnad);

/* ************************************************ */
/* If the PTBL is swapping then go do a reschedule  */
/* after locking ELQUE.                             */
/* ************************************************ */

     if (bit(ptbl->pstat,psbag) = 0)
        goto SCANNER; /* SMON0 */
     get_q_lock(ELQUE,QLOCK);
     goto SCANNER;  /*  M6 */


/* ************************************************ */
/* We hit the situation where we can run the user.  */
/* Check if the reschedule flag was set.            */
/* ************************************************ */

run_user:

     if (bit(myppcb->cpstat,presch)
        goto to_scanner;
```

```
/* ************************************************ */
/* If there is no address space, lock on the PTBL we  */
/* want to run and go to the SCANNER.                 */
/* ************************************************ */

     if (bit(cb->pstat1,pnad) == 1)
        goto to_scanner;

/* ************************************************ */
/* Is there some BIT set in the PSTAT word except     */
/* those masked out in the process table.             */
/* Go to the scanner.                                 */
/* ************************************************ */

     if ((cb->pstat & CSBIT) != 0)
        goto to_scanner;
     setbit(ptbl->pstat,psrun);
     CC.W = ptbl;
     ptbl->pextn.w->psqct --;

/* ************************************************ */
/* Was the task in the PTBL faulting?  If not, call   */
/* the task scheduler.  If so, run the user.          */
/* ************************************************ */

     if (bit (tcb->tstat,?tswp) == 0)
        {
        FSYSCL.W ++;
        goto PSCHD(ptbl);
        }
     else
        {
        mytcb = ptbl->ctsk.w;
        RSTPR1(ptbl,mytcb);
        FPGFLT.W ++;
        clearbit(mytcb->tstat,?tswp);
        interupts (on);
        goto USER;  /*  start the user via WDPOP */
        } /*  else  */
     } /* TRTN/TGRTN  */
```

2.5 Locking

There are two types of locking discussed in paths and time. The first is "spin" locking (see JP management). The second is pend locking.

Pend Locks

Pend locking is only used by pendable paths (CBs). A pend lock causes a CB trying to get the lock to pend, if the lock is held. The reason for using pend locks is the particular lock is a long-term lock. This means that the lock may be held for an indeterminate amount of time. For a path to spin the lock must be a short-term lock.

Example:

A CB is trying to get a change lock on the global lock JPLPLOCK.W. (see LP management). The CB finds the lock is held, so the CB will pend waiting for the lock. When the lock is released, the unlocking routine unpends all the CBs pended on the lock.

To get a pend lock there is a two-level locking scheme used. The first part is getting the transition lock and the second is getting the pend lock.

The first part of the locking scheme is getting the transition lock. The transition lock allows the lock to set up for a long-term action, such as to get a long-term lock or do some quick operation with an element. If the path tries to get the long-term lock and cannot, then the path will pend, but before pending the path must release the transition lock. The routine that does this type of locking is get_lock (see LP management).

Element and Queue Locking

In this section two groups of locking routines will be discussed. The first group of locking routines used in ELQUE management are the queue locks. The second group of locking routines discussed are the pend locks that deal with JP and LP databases.

Element Locking

In ELQUE management it is necessary to lock elements and queues to maintain their integrity for specific operations. For example, when the element is accessed for modification the queue lock(qlock) must be held on that queue before the queue can be touched. Elque has a special extra lock which will be discussed later.

Element Locks

An element gets locked when it has some system call working with it, such as a system call (CB) doing an operation to a PTBL; or when a PTBL is being dispatched to run. The PTBL long-term lock, PLOCK, is a form of a pend lock. The difference between the PTBL lock and the normal pend lock is if the PTBL lock is set the dispatchers, which are nonpendable paths, simply do not use the PTBL and return to the scanner for another element.

Queue Locks

There are two routines used to lock queues: get_q_lock and release_q_lock. The locking scheme used for queues is spin locking. These locking routines are not real locking routines, they are implemented inline. The reason these functions are implemented inline is because of the speed of not having to go to a subroutine. The routines are shown in the pseudocode below.

GET_Q_LOCK

Get_q_lock tries to get a lock for the caller. When the locking succeeds, then the routine returns. If the locking fails, the routine increments the collision counter supplied to the routine and spins until it gets the lock.

```
    /* $$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$ */
    /*          get_q_lock(queue,counter)                */
    /* This routine gets a lock for the queue            */
    /* supplied as an argument.                          */
    /*                                                   */
    /* $$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$ */

get_q_lock(queue,counter)
    {

    /* ***************************************** */
    /*  If there lock is locked then increment the  */
    /*  counter and spin.                          */
    /*  After the lock is released then set the    */
    /*  lock.                                      */
    /* ***************************************** */

    if (bit(queue.qstat,QLOCK) == 1)
         counter ++;
         while (bit(queue.qstat,QLOCK) == 1)
              {}
    setbit(queue.qstat,QLOCK);
    return();
    }
```

RELEASE_Q_LOCK

Release_q_lock releases the qlock for the queue passed as an argument.

```
    /* ****************************************** */
    /*         release_q_lock(queue)              */
    /* This routine releases the queue lock for   */
    /* queue passed to the routine.               */
    /*                                            */
    /* ****************************************** */

release_q_lock(queue)
    {
    clearbit(queue.qstat,QLOCK);
    return;
    }
```

ELQUE LOCKING

ELQUE has a special extra locking scheme that is used for readers of the queue. This lock is called the scan count. The scan count is used to keep track of scanners. This prevents a path from modifying ELQUE while other paths are reading it. After a path gets the QLOCK on ELQUE, it must also wait for the scan count to go to zero before enqueuing or dequeuing from ELQUE. The scan counter is useful because scans of ELQUE occur more frequently than do modifications. The scan count for ELQUE is called QSCAN. QSCAN is a part of the ELQUE structure so it is accessed by ELQUE.QSCAN.

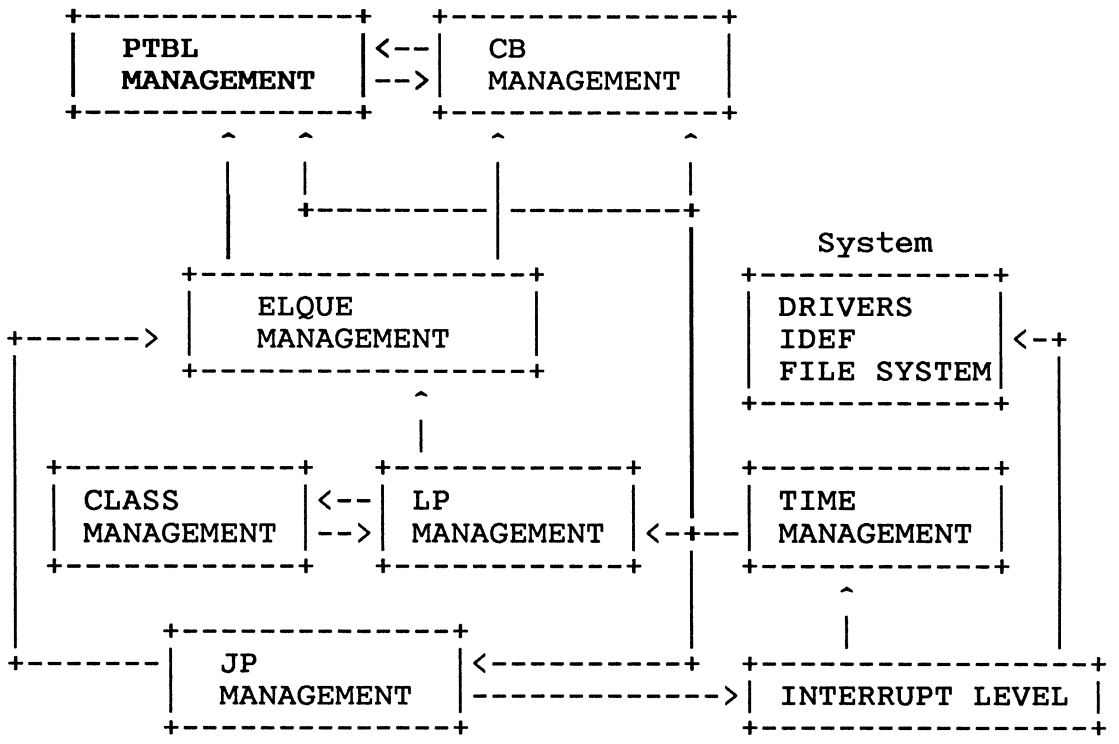
Chapter 3
Process Scheduling

3.1 Introduction

The Process Scheduling chapter of Paths and Time, documents the activities within the system that are controlled by the process table and its state. The structure of the chapter progresses from high-level discussion of the topics to detailed information about the databases and the system paths.

3.1.1 Relation to Other Parts of Paths and Time

The overall view of Paths and Time is shown below. Process Scheduling has close connections to both the system scheduler (ELQUE management) and CB management. It also takes input from time management.



Process Scheduling is closely associated with ELQUE management for scheduling of activity on, and for a process. The ELQUE scanner depends on Process Scheduling to do a detailed check of the PTBL after it is picked to run. Process Scheduling will either run the process, run a CB on behalf of the process, or return control back to the ELQUE scanner. If it returns to the scanner then the scanner may restart or continue the scan.

CB management is tied to Process Scheduling to handle the allocation and scheduling of CBs started on behalf of the process. This is done to start system calls and daemons.

Time management interfaces to Process Scheduling when a subslice or a timeslice has occurred by setting flags in the PTBL. It also provides the handlers, and switches CPU timing from general system overhead to either a specific CB or to the user.

3.2 Time Handling

Process Scheduling takes notice of time, primarily at the end of subslices and timeslices. It makes use of routines provided by Time Management to charge and reallocate time and to handle the PIT.

3.2.1 Accounting and Charging

Time is counted by the system in several different ways. CPU time is tracked by the PIT in CBs, PTBLs, and global counters. System calls are charged to the PTBL at a standard rate in the interest of fairness and repeatability. Histograms sample the program counters at interrupts and count the samples in buckets according to the PC value.

Actual CPU time (PIT ticks) are accumulated in the LPCBs and system call timing tables. These numbers are quite accurate and will usually match histogram information over reasonable periods.

User CPU time however is the sum of PIT ticks spent on PTBLs plus the standard system call charges. This is what is reported by the ?RUNTM system call. Since the standard system call charge doesn't necessarily match the actual time spent in the system call, the sum of user time, system time and idle time will not usually equal 1 CPU-second per second. It is this composite user CPU time that has the most effect within Process Scheduling.

3.2.2 Timeslicing

CPU time is allocated to PTBLs according to a timeslicing algorithm. A timeslice is the amount of time that a process is allowed to use before the process' timeslice and priority is re-evaluated. CPU time is tracked and charged in units of PIT ticks. Each tick is 1/10,000 of a second. It is allocated in multiples of ticks called subslices. The default size of a subslice is 320 ticks or 32 milliseconds.

Timeslicing is in effect only when a PTBL has control of a JP. If the process loses control, then its remaining PIT count, the residue, is saved for later use, and timing against the subslice is suspended. Each time a user task is started by Process Scheduling the PIT is loaded with the current subslice residue from the process table extender. If a PTBL is not in control (i.e., a CB or system path is running) the PIT is used to accumulate timing data but there is no timeslicing.

A subslice end is detected either by a PIT interrupt or when a system call charge overruns a subslice. When this happens, subslice end processing is performed. This stops the process, updates the running total CPU time in the PTBL extender, and makes several checks on the PTBL. Process rescheduling is also signaled at subslice end.

If the process has a CPU time limit and exceeds it, then the process is marked for termination. If round-robin task scheduling is not disabled, then the PTBL is marked for task rescheduling. If this subslice was the last one in a timeslice then the PTBL is marked for timeslice end processing.

Timeslice end processing involves reassigning a timeslice exponent (TSE), subslice count (PSLCN), and scheduling priority (PNQF) for the process. Memory management also is run, and a process' working set may be trimmed if the system is in a memory contention state.

3.2.3 Timeslice Exponents (TSE)

A group 1 process is always assigned a TSE of six. The TSE of a group 2 or 3 process depends on its prior behavior and this results in heuristic scheduling.

When a group 2 or 3 process uses its entire timeslice without blocking significantly its TSE will increase by one unless it is already at the maximum TSE of six. If it blocks significantly before using up its entire timeslice then its TSE will be recalculated.

A blocking event is significant if it blocked the process for at least 2 tenths of a second and if the event is one that is not masked out by global location TUNPBLK. The blocking events that can be masked out are: ?PMGR write request, ?SIGNL, ?SIGWT, ?WTSIG, ?IREC, and ?WDELAY. By default all of these events are significant except ?PMGR writes.

The recalculation of a group 2 or 3 timeslice exponent is done according to the following rules. If the process used less than one whole subslice before blocking its TSE is set to 1. If the process used 32 or more subslices then its TSE is set to 5. Otherwise its TSE is set to the smallest value such that 2 raised to the TSE power is greater than or equal to the number of whole subslices used. For example if a process used 5.00 subslices before blocking then its next TSE would be 3, ((2^3)=8 which is >= 5); if a process ran for 4.99 subslices its next TSE would be 2. Fractions of subslices are not actually used in the calculation but they do occur in the real world.

3.2.4 Subslice Count (PSLCN)

The calculation of the subslice count is done using the formula PSLCN = 2^TSE with one exception. If the process is "Swappable priority 1" (the highest group 2 priority) and its TSE is 6 then the number of subslices is PSLCN = 2^(TSE+1). In other words it is given twice the normal number of subslices.

The resulting timeslices (assuming 32 msec subslices) range from 64 milliseconds to 2.048 seconds (or 4.096 seconds for the exceptional case).

3.2.5 Process Scheduling Priority (PNQF)

The scheduling priority of a PTBL on the ELQUE called the PNQF or Priority Enqueue Factor. For a group 1 or group 3 process the PNQF is a function of the process group and process priority. For a group 2 process the PNQF calculation also takes into account the process' timeslice exponent (TSE). In the formulas below PPRI is a process' assigned priority, TSE is its timeslice exponent, G1 is the largest priority value assignable to a group 1 process and G2 is the largest priority value assignable to a group 2 process. G1 and G2 are constant for any given system.

The PNQF of a group 1 process is equal to its priority. The calculation is simply PNQF = PPRI.

The PNQF of a group 2 ("swappable") process is a function of both its priority and its timeslice exponent. The calculation is PNQF = G1 + (7*(PPRI-G1)) + TSE + 1. This formula gives each group 2 priority a range of values. For any given priority the specific PNQF value is determined by a process' TSE. Thus, for a given priority, processes with lower TSEs will have greater effective priority.

The PNQF of a group 3 process is calculated according to the formula PNQF = 7 * (G2-G1+1) + PPRI.

The overall allocation of PNQFs reserves PNQF=0 for system and group 1 process' CBs, followed by a contiguous range of group 1 PNQF's, PNQF=(G1+1) reserved for CBs of group 2 and 3 processes, a spread out range of values for heuristic scheduling of group 2 processes, and lastly a contiguous range for group 3 processes.

3.3 System Calls

A system call originates as a request from a user task, but to run in the system it must convert to a system task so it can use system resources. It is this transition that enables system calls to run in parallel with other tasks in the user program. The transition also makes it complicated to start a call.

3.3.1 Initial System Call Handling

System call requests enter the system through a gate via an LCALL. The very first thing the system does is make the transition from the user world to the system world. This means switching the PIT timing from the user to the system, establishing the system stack and saving the user's task state and pending the task. If the user has made a system call from interrupt level or has exceeded his CPU time limit then the process will be marked for termination.

Next the user is charged for the call. There are two different charges. The standard charge is 10 PIT ticks, expensive calls are charged 100 PIT ticks. Illegal calls are not charged.

3.3.2 Starting or Queueing a System Call

When a system call first enters the system it may not start if conditions prevent it from running or if resources are not available. These calls become queued TCB requests. They are queued off of the PTBL extender through the TCBs.

A system call that has already started may also queue itself to be restarted if it finds that required resources are not available. This is usually because a system call needs memory but cannot get it.

Every call requires that a system CB be available. The CB may or may not be used but one must be available and it must be available from the proper allocation pool. There are two CB pools, one for group 1 processes and one for all others. If the process already has used its maximum number of CBs or if none are available then the call will be queued. Those calls that are known as "parallel calls" will be queued if any other calls are already running or if the process is the target of another process' call. A call will be queued if there is already a parallel call running.

Queued system calls starting takes precedence over user tasks when a process is run by the scheduler. A queued call may be unable to start when the process is scheduled for the same reasons that it was queued in the first place. In some cases (such as a call pends and the user is multitasked) the user code may then be allowed to run even if the system call could not.

3.3.3 Running the System Call

Once the system has decided to run a system call it can take two paths. The least expensive in terms of CPU time is to simply point the system to the CB and jump right into the handler. The other path requires that the CB be filled in and placed on the system ELQUE for later scheduled execution. The particular path taken depends on the call and the current JP identity and state. Only if the JP is eligible to run the particular call and it does not need to reschedule can the fast path be taken.

If a call does "run fast" (a technical term) and it never pends during execution then the process will regain control of the CPU without rescheduling. Otherwise, the process loses control to be scheduled again later.

3.3.4 Concurrency

Because each system call runs as a separate system task with its own CB there can be more than one call active in the system at the same time. And because the PTBL represents another system task it can be active at the same time as the system call(s).

If there are multiple JPs in the system then it is possible to have multiple paths being executed for the user simultaneously. Note that while multiple system elements (CBs and PTBLs) can be running on different JPs at the same time, it is not possible to have multiple user tasks of the same process running at the same time because each user process is represented in the system by only one PTBL.

3.3.5 Page Faults and Daemons

The mechanism used by system calls for running concurrently on a CB is a general purpose mechanism. User page faults and daemons are handled like system calls except that they are not initiated by the user directly.

Page faults join the system call path just after the initial user to system transition. Page faults are very similar to system calls in that they can be queued off of the process table extender and their start can be delayed.

Daemons are started only when a process has no other calls active and join the system call path at the point where a system call will definitely be started. Daemons are also discussed below.

3.4 The Process Databases

The activities of Process Scheduling are centered around the two databases that describe a process to the system, the process table (PTBL) and the process table extender (PEXTN). Together they provide the driving force for Process Scheduling.

The other major data structures that Process Scheduling is concerned with are the CB, task control blocks (TCBs) and user status tables (USTs). These are described later.

3.4.1 Importance/Use in Process Scheduling

The process table (PTBL) remains in memory as long as the process exists but the process table extender (PEXTN) swaps with the process. By the design of these two databases and the placement of the status information Process Scheduling will attempt actions that require information from the extender only when a process is eligible (swapped in). Thus it can ignore the possibility that an extender might be unavailable.

The PTBL holds the data with which the system schedules and maintains process unique information. It also holds information about operations done to or capable of being done to a process while it is swapped out. Of course it also holds pointers to the other process related data structures.

The PEXTN holds data that describes the state of the process and its activities in greater detail. Information needed only when the process is running or eligible to run can be kept here.

Process Scheduling is concerned primarily with data used for scheduling the process. Most of this driving data are various status and flag bits. The flags used by Process Scheduling fall functionally into three groups plus a few miscellaneous ones. The first group is the show stoppers. They are checked at the beginning of process scheduling and if any are set the process is skipped. This group includes the locking flags and the running flag. The second group contains the "high-priority" bits that cause certain actions to be taken before any actual scheduling can occur. The flags of the first two groups are in word PSTAT. The final major group of flags is known as the daemon request flags. These flags are scattered throughout PSTAT1 and the PFLAG words.

There are many other data items in the PTBL and PEXTN used by Process Scheduling, they are defined in Section 3.8 and their use is discussed in the code walkthroughs and shown in the pseudocode.

3.4.2 States of the PTBL

A process can be in many different states. Within the system the process is represented by its PTBL for the entire life of the process. The states of the process are determined by the queue the PTBL is found on and by status information in the PTBL. The queue the PTBL is on reflects the major state of the process.

When a process is eligible its process table extender and current working set are actually in memory although not necessarily mapped. The PTBL of an eligible process will be on either the ELQUE, BLKQ or MBLKQ. The MBLKQ links PTBLs of processes that are explicitly blocked by the ?BLOCK or ?PROC system call. The BLKQ holds PTBLs of processes that are blocked because they are waiting for something to complete. The ELQUE is the queue of PTBLs of processes ready to run or needing some action by the scheduler. Only if the PTBL is on the ELQUE and is selected to run by the system scheduler will the process scheduler be called for the process.

When a process is swapped out the PTBL is moved to one of the swap queues IEBLK, IERES, IEQUE, or IESWP depending on the process type and its state before being swapped. Even while the process is swapped out to disk the PTBL remains in memory to represent the processes. The process extender is swapped to disk as part of the ring 1 to 7 context of the process. Other subsystems will still keep information referring to the process in their data structures. For instance, the connection management subsystem will still have references to the process in its connection database.

The process management volume of the internals manual has a more detailed description of the various queues that the PTBL can be on and the process states that each represents.

3.4.3 Blocking and Unblocking

During the life of a process it is quite normal for it to block and unblock. It will be blocked when the process scheduler finds no TCBs ready to run, while it is waiting to terminate, or if it is explicitly blocked by a system call. When a process is blocked its PTBL is put onto one of the blocked queues. If the process is explicitly blocked it is placed on queue MBLKQ, otherwise it is placed on queue BLKQ. A process unblocks if a TCB is unpended by the occurrence of some event, or if the system needs to start a daemon for the process, or if the process is explicitly unblocked.

The process of blocking and unblocking a heuristically scheduled (group 2) process affects the process' ELQUE priority. In theory if a process blocks before using its full timeslice then it is displaying interactive behavior and the system should give the process a higher ELQUE priority (but shorter timeslices) to enhance interactive response times. See the discussion of timeslicing above for more information.

The problem with the theory is that a non-interactive process can display similar blocking behavior. One example is a program that continuously writes data to a terminal. The process will block when the output buffer fills and unblock when the buffer empties. This behavior looks interactive if the only factor taken into account is the blocking and unblocking.

To counter this AOS/VS allows the system manager to prevent certain specific blocking events from affecting process ELQUE priority if the event didn't block the process for very long. Global location TUNPBLK is a mask of those events to be discriminated against. The blocking events that can be specified are PMGR reads and writes, ?SIGNL, ?SIGWT, ?WTSIG, and ?DELAY system calls, and IPC messages. The value of TUNPBLK can be set for the needs of a specific environment by patching the system .PR file or by using the SYSTUNE utility. See PARSA.SR or STABLE.SR for specific bit assignments.

When a process on the BLKQ is unblocked a check is made to see if it was blocked for one of the events in TUNPBLK. This is done by logically AND'ing global TUNPBLK with the contents of TBLKFLG in the PTBL. If the result is zero then the blocking event was not one being discriminated against and the process is deemed interactive.

If it was blocked for an event whose bit is set in TUNPBLK then a check is made for how long the process was blocked. If the process was blocked for more than two tenths (0.2) seconds then the process will also be considered interactive.

If a process appears interactive during an unblocking then it may have its timeslice exponent lowered. This lowers its PNQF and improves interactive response. This also shortens the timeslice allocated to the process when it is next scheduled.

If the process was blocked less than the 0.2 second time limit on an event type marked in TUNPBLK then the blocking is not considered significant and the process' timeslice exponent will be left unaltered. The process may eventually use up its full timeslice without encountering a significant blocking event. At that time its timeslice exponent (and PNQF) may be raised, thus limiting the process' ability to compete for CPU time. This is the appropriate action for non-interactive programs.

3.4.4 High-Priority Activities

After the scheduler has determined that the PTBL is not locked or already running it must check certain high-priority status flags. These flags indicate that the process or PTBL is in a state that requires attention before the system can consider scheduling the user code. The flags are all in word PSTAT of the PTBL. In the source and in the discussions and pseudocode below the bits are collectively known as PRBITS.

The processing of these conditions is handled during the scheduling of the process since each is specific to the process and, as far as the system is concerned, the priority of the actions are the same as the priority of the process. It does not make sense for these actions to be completed at higher or lower priorities.

In order of decreasing priority the flags are:

PSEW	scheduler activity is in progress
PSBRK	a ^C^X interrupt is pending
PSBAG	process should be swapped out
PSBLK	the process is blocked and the PTBL needs to be moved to a blocked queue
PSDP	a daemon needs to be run
PSMWT	the process needs memory
PSTSU	timeslice end processing needs to be done before running the user.

By this ordering you can tell that, for instance, swapping a process out has higher-priority than checking if it should try to get memory again.

PSEW is set if the PTBL is in a state where it should not be checked in any greater detail or if a system task or daemon is active for the process. This does not include active system calls however. If PSEW is set the scheduler immediately skips the PTBL and continues the ELQUE scan.

If PSBAG or PSBLK are true then the scheduler hands the PTBL off to another part of the system to be processed. For PSBAG the PTBL is moved to the core manager queue. This may fail if the process state changed since the flag was set. For PSBLK the scheduler attempts to move the PTBL to a blocked queue. It may fail if the appropriate queue is locked. For both PSBAG and PSBLK if the PTBL was removed from the ELQUE the scheduler is restarted otherwise the scheduler continues the ELQUE scan with the next element on the ELQUE.

Both PSBRK and PSDP signal that a daemon needs to run before the user regains control. Even though ^C interrupts are handled by a daemon they are given their own priority flag because the interrupts are more important than process blocking or swapping. If the interrupt was a ^C^B it would not make sense to hold up the process termination. The scheduler takes the same path for both priority flags, possibly after resetting PSBRK. Daemons are handled in the system as a special case of system call processing. They differ in that they are not initiated by the user program and do not return to the user. When a daemon is started bit PSEW is set indicating that system activity is taking place for

the process and no user code or system calls will be scheduled to run for the process. When a daemon finishes it is the daemon's responsibility to update the PTBL status flags to allow it to run later.

When PSMWT is true there is at least one system call waiting. The flag is set if a system call cannot continue for lack of free memory. When this happens the call copies system global MKEY into PTBL word PMKEY, backs out to its starting point and puts its TCB onto the chain of waiting system calls. The high-priority action taken is to compare the current value of global MKEY with the stored PMKEY. Since MKEY is changed when the release of memory adds a block to a previously empty chain, if PMKEY differs from MKEY then memory may be available and the system call can be re-attempted. It is quite possible that not enough memory was freed or a higher-priority process has taken the memory and the system call may fail again. But this mechanism does ensure that the processes compete for memory in a manner that takes into account their relative priority and available CPU time.

PSTSU indicates that a process' timeslice expired because of a system call charge and that timeslice end processing is waiting. The possible effects are to change (lower) the processes priority, or to terminate the process for using up its CPU time allocation. After timeslice end processing the system scheduler is restarted from the top.

3.5 Process Scheduler Use of CBs

3.5.1 CBs in a Process World

Despite its emphasis on PTBLs, the process scheduler is also responsible for spawning the vast majority of CBs within the system. This responsibility is given to the process scheduler because the coordination and priority of these system activities are based on the PTBL.

CBs are allocated, initialized and started by the process scheduler to handle user system calls and daemons. System calls and daemons are similar in that they both execute code paths within the system (ring 0) address space and are managed via CBs. System calls are started at the request of the user or his Agent. They represent a continuation of a user task within the system and they return control to the task when they finish (unless the call terminates the process). Daemons on the other hand are started to perform process activity independently of any user task.

When looking at a CB in a system dump a daemon can be identified by the contents of fields CALL.N and CATCB.W. CALL.N will contain a -1 indicating no valid system call number. CATCB.W will contain a 0 indicating no associated user TCB.

3.5.2 Setting Up and Dispatching CBs

The process scheduler always allocates and initializes the CB but it may either dispatch the CB directly or link it onto the ELQUE for later execution. CBs are dispatched directly by the process scheduler only if there are no higher-priority ready CBs or PTBLs, no reschedule flags set, and if the CB can run on the current job processor.

When allocating CBs, the Process Scheduler maintains (or calls routines that maintain) several system globals. Locations PCB.W, SCB.W and TMPCB.W defined in SZERO.SR hold the address of the current, backup, and possibly temporary backup CBs for a specific job processor. Locations RSTCKT and SSTCKT defined in STABLE.SR count the number of CBs available within the entire system for group 1 or group 2/3 process requests.

Before dispatching a CB or putting it onto the ELQUE to be scheduled later the process scheduler must initialize the CB. Entries that are set to initial values include its status, the timing and error entries, the pointers to the CB's stack and the address of the associated PTBL. Data that define the execution path of the CB include its priority enqueue factor (PNQF), system call number and associated TCB address for system calls and class code for class scheduling.

If the process scheduler is going to transfer control directly to the CB then it must put the addresses of the logical processor and job processor control blocks into the CB. In this case the process scheduler must also duplicate the action of the ELQUE scanner in setting up the system global databases and the processor states. (See CB management elsewhere in this volume.)

3.5.3 Time and CBs

AOS/VS keeps track of the actual CPU time spent while running on CBs and PTBLs. The CPU times accumulated are totalled in the data bases of each logical and physical processor and in the table of system call timings. These actual times are not the times charged to a user process. The user process is assessed the total of the actual time spent on the PTBL (in user space) and system call charges which are determined by the system call number.

When a CB completes a system call or daemon, CBUPDT is called to update the appropriate time counters in the processor data bases (LPCB and PPCB). If class scheduling is on or class timings are being accumulated then the appropriate counters and allocations are updated. After system calls CBSTAT is called to accumulate the CB time into the table of system call times pointed to by STTBL.W. CBSTAT, CBUPDT and the related PTUPDT routines are discussed in detail in the LP management chapter.

3.5.4 Concurrency

Activity for a process can be represented in the system by both its PTBL and CBs. This makes it possible to have multiple threads of execution eligible at the same time for a process if the program is multitasking with one or more outstanding system calls. If the hardware configuration includes more than one job processor it is possible to have more than one code path running simultaneously. However since the user code is represented by only a single control block, the PTBL, it is not possible to have more than one user task running at the same time. This greatly simplifies the job of the task scheduler in the Agent.

Several situations can prevent concurrency. For instance, daemons require that there be only one active execution thread. If user code, or a system call CB, or another daemon is running then a daemon will not start, and user code and other system calls will not be started once a daemon is running.

Even after starting concurrent paths, the various locks in the PTBL and system limit simultaneity. So long as a lock is held, any other code paths needing the same lock will not run.

3.6 User Task Scheduling

3.6.1 Overview

Part of the function of the AOS/VS process scheduler is to dispatch user tasks. This includes handling task state switching and round-robin rotation of tasks with the same priority. Most of the actual management of tasks however is handled by the Agent in ring 3. This section will not delve into services provided by the Agent.

Task scheduling is performed as the final phase in the scheduling of a specific process. The priorities of ready tasks plays no part in the selection of a process to run. One result of this is that for processes of equal priority a task of lower priority on one process can run before a higher-priority task of another process if the process with the lower-priority task was nearer the head of the ELQUE.

This emphasis on scheduling of processes is a major difference between AOS/VS and AOS/RT32. Where AOS/VS keeps PTBLs on its eligible queue, AOS/RT32 keeps tasks on its queue. This allows AOS/RT32 to interleave priorities of tasks on its eligible queue across processes.

3.6.2 Task Scheduling

When the task scheduler scans the active TCB queue it selects the first TCB with no pend bits set. Since the queue is in priority order this is the TCB of the highest-priority ready task. If no ready TCB is found the process is marked "not ready to run" and if the process can be blocked the PTBL will be moved to queue BLKQ.

The next step in task scheduling is implementing the round-robin scheduling policy. This means putting the TCB of the selected task behind other TCBs with the same priority in the queue. This is done before starting the task so that no matter how the task loses control the round-robin policy is enforced. The check is simple: if there is another TCB in the queue behind the selected TCB and it is at the same priority then the shuffle must be done.

A task context switch is also needed if the selected task is different than the last task that ran. Each task has several global state variables that must be saved and restored during the switch. These include stack pointers, fixed and floating point registers and status, trap handlers, and the extended save area. See the code descriptions or pseudocode below for implementation.

A task loses control when it makes a system call that must pend, when it makes a task call with rescheduling enabled, at the end of a CPU subslice (32 milliseconds), if a higher priority task in the same process becomes ready, or if a higher priority process becomes ready.

3.6.3 The TCB

The TCB is the repository of the scheduling state of each task in a process. The TCB plus the task's stacks in rings 3 to 7 and possibly an extended save area in ring 7 contain all the information needed to run a task.

Unlike process PTBLs, all TCBs for a process' active tasks are on a single queue no matter what their state. The head of the queue is in the UST.

Each task has a priority from 0 to 255 with 0 being the highest-priority. This priority is kept in entry TPR of the TCB. The initial task of a process is given priority 0 by the LINK utility. The priority of all other tasks is determined by the user code that creates the new tasks. The TCBS in the active queue are linked in task priority order.

In addition to being linked onto the UST for task scheduling, the TCB may also be linked onto one of two chains off of the process table extender. If one or more tasks are waiting on the completion of a ?WDELAY then location PDFR.W in the extender points to the TCB of the task with the shortest wait and offset TSYS.W in the TCB points to the next one, if any. If one or more tasks have issued system calls that were unable to start then location PSWD.W in the extender points to the first TCB and if there are any other TCBs they are linked through TSLK.W.

The scheduling state of an active task is determined by the pend flags in entry TSTAT. Bit ?TSPN indicates that the task is pended in the system. The other pend bits in TSTAT are managed by the Agent and honored by the scheduler. By having all the pend conditions in a single word the MV queue instructions can be used to scan the queue. See the TCB data structure in Section 3.8 for descriptions of the pend flag bits.

See Section 3.8 for a short description of the other TCB entries used by process scheduling. For a complete description of the entire TCB see the Process Management volume.

3.6.4 The UST and Ring 3

The UST is the central database for managing all of a process' TCBs. The UST is shared by the Agent and the AOS/VS kernel to schedule and keep track of user tasks and global process state. Each TCB is on either the active chain (queue header USTAQD1 and USTAQD2) or on the free chain (USTFQD1 and USTFQD2).

Process scheduling is concerned only with the entries in the UST that deal with the scheduling of tasks. These are the extended task state area address, the active task and, of course, the active TCB chain. For a short description of the entries used by the process scheduler see Section 2.8. For a complete description of the UST see the Process Mangagement volume.

When the Agent needs to change the task scheduling queues or databases it sets word ?TSMA in ring 3. When this flag word is non-zero the task scheduler in the kernel returns control directly to the place in the user program that it was executing when it lost control. This will always be in the Agent who will continue to process the task database changes and clear ?TSMA when done.

3.7 Interfaces to the Rest of AOS/VS

Of course Process Scheduling does not stand alone in the system. It relies on many other parts of the system and it provides specific services to them.

Services that Process Scheduling provides to ELQUE are the final check of the PTBL for runnability and the dispatching of code to be run. The service provided to CB management is the restarting of a process after a CB has completed its task but before a subslice has expired. This same service is used within Process Scheduling to continue a process in the middle of a subslice after a task rescheduling.

The service required of ELQUE management is the insertion of CBs onto the ELQUE for later execution.

3.7.1 External Routines Used by Process Scheduling

Process Scheduling depends on many other areas of the system to implement the daemons and the individual system calls. They are too numerous to mention individually. The system call dispatch table MCCT.W is the link to the system calls. Table BTBL in the scheduler contains the links to the daemon routines. Do not confuse table BTBL in the scheduler with global BTBL, which is the interrupt vector table.

Routines Called by Process Scheduling

Symbol	Module	Action or Service
ALSTK	SCHED	Allocate CB/Stack according to PTBL group
CTBLK	CORM2	Move PTBL from ELQUE
FIXCB	SCHED	Puts CB onto ELQUE w/proper MP status
MAPCON	SCHED	Set ATU map for process address space
MEVENT	SCHED	Resched mother if event is higher priority
PRBAG	CORM2	Start process swap-out
RELPTBL	MPUTIL	Release PTBL Unlocks the PTBL, releases its address space and unpends any CB/PTBL pending on it.
RESCH	SCHED	Start ELQUE scan (same as SCHED:SMON0)
RUNEX1	SCHED	Resume ELQUE scanning The PTBL is not locked, the scan mask is still on the stack, the scanner count is still incremented.
RUNEXP	SCHED	Resume ELQUE scanning PTBL is locked, address space is not mapped.
RUNPTP	SCHED	Resume ELQUE scanning PTBL is locked, address space is mapped
RSTPRC	SCHED	Restore task state from TCB + PTBL sets up FPU and fault handler, checks for subslice up and restarts PIT on process
SMON0	SCHED	Same as SCHED:RESCH
TCBAD	SCPRC	Abort process for of bad TCB chain in AGENT
TDSCL	SCHED	Start PIT timing on CB
TSKRST	SCHED	Restore task stack state from TCB
TSKSAV	SCHED	Save task stack state in TCB
TSUP	CORM2	End of timeslice processing
XLOCK	MPUTIL	Standard spin lock

3.7.2 Global Data Used by Process Scheduling

Like any system function, Process Scheduling relies on global system data and storage. Some of the data controls the decisions made, while others reflect the results of those decisions.

The scheduling functions depend especially on globals defined in module SZERO. Each JP has its own copy of the data in SZERO. This is the data describing and controlling the scheduling state of each processor.

Those items from module STABLE are system-wide and affect all job processors. Many of these data items are protected by locks and others must be updated using indivisible instructions. Most of the monitoring and statistical data generated by the system is located in this module.

The table below lists the external data locations used by Process Scheduling.

Symbol	Module	Description
CC.W	SZERO	current control block (CB or PTBL) address
CHTBL	SCPRC	bit map of calls legal for child JP
CTSK.W	SZERO	addr of current TCB in current process
CWTB	SCPRC	bit map of parallel system calls
ELQUE	STABLE	offset QSCAN in queue descriptor decremented if PTBL not locked
FBK.W	PARS	hardware defined location (32 octal) addr of context fault block
G1	STABLE	lowest group 1 priority
MAXSYS	SCPRC	highest legal system call number
MKEY	STABLE	times memory page(s) released (mod 64K)
MYLPCB.W	SZERO	addr of LP for current JP
MYPPCB.W	SZERO	addr of Physical Processor Ctrl Blk for JP
NMONLY	STABLE	metering location number of times child JP couldn't process a system call because it was mother-only
NNODCL.W	STABLE	metering location number of times a TCB request was started (system calls)
NNORQ1.W	STABLE	metering location number of times only the current TCB in a process could be scheduled
NPSCDR.W	STABLE	metering location number of times task rescheduling is found disabled
NPSLP.W	STABLE	metering location number of times TCB chain scanned
PCB.W	SZERO	addr of CB currently running
PTMP1.W	SZERO	process scheduling temp holds system call number (or 0 if daemon) across the transition from PTBL to CB
PTMP2.W	SZERO	process scheduling temp holds TCB address of task making a system call, or 0 if daemon
PTMP3.W	SZERO	process scheduling temp holds entry point address of system call or daemon code
RSTCKT	STABLE	number of group 1 CBs available
SCB.W	SZERO	addr of CB to replace PCB.W
SCTBL.W	STABLE	metering location address of system call count table
SSTCKT	STABLE	number of group 2/3 CBs available
TINDST.W	STABLE	metering location number of times activity started on CB, includes system calls, daemons and page faults
TMPCB.W	SZERO	addr of CB to replace PCB.W or SCB.W
UST	PARSA,PARU.32	ring 3 User Status Table

3.8 Databases Used by Process Scheduling

Process Scheduling maintains or references several databases to perform its services. The most important of these are the process table (PTBL) in ring 0 and its extender (PEXTN) in ring 1, and the task control blocks (TCBs) and the user status table (UST) in ring 3. Those parts of the databases used by Process Scheduling are described below.

3.8.1 Process Table

The following is a description of the parts of the PTBL used during process scheduling. See the table in the Process Management chapter or PARS for a more complete listing and for the locking rules for modifying each item.

Symbol/Value		Meaning
PCLASS	5	Process class Contains a bit indicating the class of the PTBL or CB, or zero if the PTBL is mother-only or if class scheduling is not enabled. Set in PTBL at ?PROC. Set in CB when initialized or cleared if a system call needs to become mother-only.
PSTAT	6	Process status bits (see below)
PSTAT1	7	Process status bits, word 2 (see below)
CALLN.W	22	SYS CALL number times 2 Used to index into system call handler address table and into system call count and timing table. Also used as an indication of whether a CB is running a system call or a daemon (CALLN.W = -1).
PFLAG	45	Flag word 1 (see below)
PFLG2	46	Flag word 2 (see below)
PFLG3	47	Flag word 3 (see below)
PFLG4	50	Flag word 4 (see below)
PEXTN.W	51	PTBL Extender address Physical address of the process table extender. Set when process is initially loaded or whenever process is swapped into memory. Contains a zero if the extender is not in memory.
PID	62	PID assigned at create time
PMKEY	71	MEM WAIT flag and key Set to the current value of MKEY when the process pends for memory. See also PSTAT bit PSMWT.
PINSU	106	In scheduler mode flag (0 OR 1)
PTRGC	122	Target call counter
PSIDI	134	# Enqueued TCBS W/Indirect calls
PFLG5	135	Flag word 5 (see below)

Below are descriptions of the various flags and bits in the PTBL used by process scheduling. Those bits that have special meaning for Process Scheduling have more detailed descriptions.

PSTAT

Flag bits that will be checked during the ELQUE scan must be in this word. It is adjacent to word PCLASS and together they hold the status used by the scan. The ELQUE scan bits have common meaning for both CBs and PTBLs. These bits are:

PSRDY 00001 Not ready to run

PSRUN 40000 Running
This bit indicates that a PTBL or CB is running and is also used to prevent a CB from being selected after it is put onto the ELQUE but before it is fully initialized. This is done when a CB to run a system call is being put onto the ELQUE.

PSEW 20000 Sched action
Usually means that a DAEMON is running or is on the ELQUE.

PSNCB 10000 Don't look - process can only use a CB
Set if there are system calls waiting to run and either there is no CB available for the call or there is no user task ready. Never set for group 1 processes.

PMAST 00040 Mother-only element (1=Mother-only)
Set in PTBLs of processes that are mother-only, or in CBs of daemons, or in CBs of mother-only system call paths.

PLCK 00020 Process table lock bit

PSETR 00010 Don't enter
Indicates that the user code should not be run for some reason.

PSFSY 00004 System page fault
Used to determine what type of return to the user ring should be used. If set then a WDPOP must be used to restore the context and possibly restart an instruction.

PTRAN 00002 Element transition bit

The following PSTAT bits are used in PRBITS as a dispatch value. The highest-priority function has the leftmost (lowest bit number) assignment.

PSBRK 04000 OP interrupt
Even though ^C^x interrupts are handled by daemons they have their own priority bit because they are higher priority than swapping or blocking.

PSBAG 02000 SWAP OUT process

PSBLK 01000 BLOCK process
If a PTBL couldn't be put onto the blocked queue because of a lock then this bit will be set. The scheduler will try again to move the PTBL if this bit is set.

PSDP 00400 START UP DAEMON
If this bit is set then other bits in the PTBL are checked to determine what daemon to run. The individual daemon bits are indicated in the flag words below. See table BTBL for the relative priority of the bits.

PSMWT 00200 Wait for memory key to change
Global MKEY is copied into the PTBL and this bit is set if the process must wait for memory. When the PTBL is selected the global MKEY is checked against the stored value. If there is no change then the process cannot get memory and will not be started.

PSTSU 00100 Time slice is up

PSTAT1

PNFST 10000 Sys call did not run 'FAST'
Set if a system call or daemon had to pend or was put onto the ELQUE to be run on another JP. If this bit is clear then a system call path can return to the process via RERUN without going through a full reschedule.

PFLAG

PFFIR 10000 Initial Load
Used in daemon dispatch table BTBL. If PSTAT bit PSEW is not set then the initial load daemon should be started.

PFINT 01000 Run ^C^A DAEMON
Used in daemon dispatch table BTBL.

PFTRM 00200 Run TERM DAEMON
Used in daemon dispatch table BTBL.

PFRSH 00020 Resched flag

PFNFR 00004 Narrow process becoming non-resident
Used in daemon dispatch table BTBL.

PFLG2

PFBLE 00400 Scheduler can block process
Clear if the process is resident or cannot be blocked. Checked when the PTBL scheduler finds that there are no user tasks to run and no system activity running or waiting to run for the process. If set then the scheduler will try to move the PTBL to the blocked queue.

PFATL 00020 Process termed by system
Used in daemon dispatch table BTBL.

PFNTR 00004 Narrow process becoming resident
Used in daemon dispatch table BTBL.
PFQSC 00001 Inhibit scan of backed up TCB request
Checked before looking to start or restart system calls waiting on TCBs. Set just before major housecleaning is done to the process as is done for swapping it out.

PFLG3
PFIWC 02000 Interrupt (^C^B,^C^E) term of PROC
Used in daemon dispatch table BTBL.
PFIRS 01000 Int world interrupted task
PFPCH 00004 Hold on >1 parallel call
Set if a parallel system call is started for a process. Checked before a system call is started.

PFLG4
PFNRO 00020 Process is narrow (16 BIT)
Set at ?PROC. Used to determine whether to restore a wide or narrow stack when restoring a task's state.
PFISS 00001 Interrupt sequence (^C^x) received
Used in daemon dispatch table BTBL.

PFLG5
PFUTC 100000 Run user context trap daemon.
Used in daemon dispatch table BTBL.
PLWAIT 01000 Address space waiter bit
If set then RELPTB will call UNPEND with the PTBL address as the key to wake up anyone waiting on this PTBL's address space.

3.8.2 Process Table Extender (PEXTN)

The process table extender holds information not critical to the management of AOS/VS when the process is swapped out. It is allocated in ring 1 address space and is swapped out with the process. The table below defines the offsets used by Process Scheduling.

Symbol/Value		Meaning
PSQCT	0	Active system path count Indicated the number of CBs running on behalf of the process including system calls and daemons.
PSQMX	1	Maximum active system calls Set at ?PROC to limit the number of system calls that the process can have active at the same time within the system. It is compared to PSQCT to determine whether to start a system call or link the TCB making the request on the chain pointed to by PSWD.W.

PSWD.W	12	Link of TCBs waiting to start system calls Contains the address of the first TCB whose system call couldn't start immediately. The TCBs are linked through TCB offset TSLK.W. Set to zero if no TCBs have calls waiting.
PSL	17	Sub-slice residue
PSBRS.W	22	Start of array of Segment Base Registers (SBRs) The SBRs of rings 1 to 7 are kept here for MAPCON and REMAPCON. The SBR of ring N is at offset PSBRS+2*(N-1).
PSIOC	44	Count of outstanding MCA and LPB I/Os Used to see if a process can be swapped out.
PCEXV.W	162	Addr of TCB whose extended state is current When a program uses the extended task state save area the scheduler keeps the address of the TCB of the last task that had its variables moved into the global area. If the same task is the next to run with a save area then no save/restore is needed even if other tasks without the save area have run.
PMARG.W	206	First Dblword of stack return block When the user enters the system via an LCALL the return block from the stack in the user's ring is copied into words PMARG.W to PMPC.W in the extender. Used to construct a return block on the ring 0 stack to return to the user. Copied into the TCB also.
PMAC0.W	210	AC0 from return block
PMAC1.W	212	AC1 from return block
PMAC2.W	214	AC2 from return block
PMAC3.W	216	AC3 from return block
PMPC.W	220	PC + Carry from return block (see also PMARG.W) Used to determine the ring of the caller by extracting it from the PC. The length of the return block is PMPC.W-PMARG.W+2.
PUGFH.W	464	Address of ring 3 GIS fault handler Restored during RSTPRC.

3.8.3 TCB - Task Control Block

Each task in the user process has a TCB. There is at least one task and one TCB for each process. The TCB holds the state of one execution path of the process across all of the non-system rings. State data saved for each task includes the stack pointers, the PC and register contents, the arithmetic overflow/underflow detection flag, extended save area and other task specific data.

TCBs are created and managed in ring 3 by the Agent for the process. They are scheduled by the PTBL scheduler in ring 0. The offsets used by Process Scheduling are described below.

Symbol	Value	Meaning
TLNK.W	0	Forward link among TCBs Used (with TLNKB.W) to maintain the queue of TCBs in ring 3. Modified by the process scheduler to implement round-robin scheduling of equal priority tasks.
TLNKB.W	2	Backward link among TCBs (see TLNK.W)
TSTAT	4	Task status word (see individual bits below)
TCBFL	5	Task flag word (see individual bits below)
TSTACKS.W	6	Start of per ring stack save areas For rings 1 to 7 each task has its own stack. The information saved for each ring is the stack overflow handler address, frame pointer, stack pointer, stack limit, and stack base. The start of each ring's stack save area is at TSTACKS.W+10.*(ring-1). Used in TSKSAV and TSKRST when a task's stack state is saved and restored.
TOVF.W	114	Overflow mask for task Saved each time the task enters the Agent or the system. Copied from the PTBL extender if the task directly entered a ring 0 gate.
TAC0.W	116	AC0 save area for task (see TOVF.W)
TAC1.W	120	AC1 save area for task (see TOVF.W)
TAC2.W	122	AC2 save area for task (see TOVF.W)
TAC3.W	124	AC3 save area for task (see TOVF.W)
TPC.W	116	PC save area for task (see TOVF.W) Used to determine the ring of the PC of the task.
TUSPS.W	130	USP save area for rings 1 to 7 The USP for ring N is at offset TUSPS.W+2*(N-1).
TFPA	146	Floating point save area The task's floating point status registers and accumulators are kept here. RSTPRC mangles the stack pointers to make a WFPOP instruction load the FPU state from this part of the TCB.

TELN.W	172	Addr of extended save area for task. If this task allocates a save area for itself then the scheduler will ensure that the task's data is in the global extended state area when the task is running. Set to zero if no save area exists.
TUCD.W	174	Current descriptor
TSYS.W	176	System call word Set when a task makes a system call. Used by Process Scheduling when it starts a system call that had been waiting to start.
TID	200	Task ID
TPR	201	Task priority Used for round robin scheduling of tasks.
TSLK.W	202	System call link Used to link TCBs who have system calls waiting to start. PSWD.W in the PTBL extender points to the first TCB in the chain.
TFSYS.W	204	Saved TSYS.W used during fault

TSTAT

Most of the bits in this word indicate that the task is not ready to run. Mask CRUMS (175674 octal) defines those bits and is used for the queue search instruction to find a TCB to run. The bits that are included in the CRUMS mask are marked with an asterisk in the table.

	Symbol	Value	Meaning
*	?TSPN	1b0	Task is pended This general status bit is set when conditions other than those defined in TSTAT prevent the task from running. While several pend bits in TSTAT could be set simultaneously, only one bit elsewhere in the TCB can be the reason for this bit being set.
*	?TSSG	1b1	Waiting on .XMTW or .REC
*	?TSSP	1b2	Task is suspended
*	?TSRC	1b3	Waiting for TRCON
*	?TSOV	1b4	Waiting for overlay
	?TSWP	1b5	Task is faulting The scheduler checks this bit to see if it should restart the task via a WDPOP or a WPOPB. A WDPOP must be used to restore microcode state if the task had faulted.
*	?TSGS	1b6	Pended due to agent synchronization
*	?TSAB	1b7	Pended awaiting ?GABORT
*	?TSTL	1b8	Pended awaiting ?TUNLOCK by another task
	?TSYG	1b9	Task has been signaled
*	?TSDR	1b10	Pended by ?DRSCH

*	?TSLK	1b11	Pended on ?FLOCK request
*	?TSXR	1b12	Pended on XMT or REC
*	?TWSG	1b13	Pended on ?WTSIGNAL
	?TSUT	1b14	Awaiting return from user ?UTSK code
	?TSUK	1b15	Awaiting return from user ?UKIL code

TCBFL

TCB flag word. Contains status flags that couldn't fit into TSTAT and also some task data. Status bits that indicate the task is pended are reflected in TSTAT bit ?TSPN.

Symbol	Value	Meaning
?NREMSK	7b2	Ring to return to after fault
?TCBFLAL	1b3	Agent can skip ?ALLOCATE on ?SPAGE
?TCBGIF	1b4	Task is resolving a GIS fault
?SCFAST	1b5	System call didn't execute fast
?TCBSEND	1b6	Pended on ?OPER send
?TCBOPER	1b7	Pended on ?OPER receive
?UTIDMSK	377	Mask for unique task ID (1..32)

3.8.4 User Status Table

This ring 3 database contains the state of the user process. It is created by the system after mapping the Agent code for this process. Part of the UST is initialized from the preamble of the ring 7 program file. It is always located at location 400 octal in ring 3. The structure of the UST is defined in parameter files PARU.32.SR and PARSA.SR.

The UST is maintained by the Agent and the process scheduler. Process Scheduling uses it to find the process' status and locate the task control blocks. Process Scheduling updates it with the address of the task selected to run and the task whose extended save area is current.

Symbol	Value	Meaning
UST	400	Location of UST in ring
USTEZ	0	Size of per-task extended variable save area
USTES	1	Address of global extended variable save area This is a 16 bit address so the location is limited to the first 64K of the process address space.
USTSS	2	Start of symbol table
USTSE	4	End of symbol table
USTDA	6	Address of Debugger (or -1)
USTRV	10	PR file revision
USTTC	12	Number of tasks created
USTBL	13	Number of impure blocks
USTOD	14	Address of overlay table
USTST	16	Shared starting block number
USTIT	20	Interrupt address
USTSZ	22	Shared size in blocks
USTPR	24	PR file type
USTKL	25	Address of .KILL table
USTBM	27	Address of BOMB routine
USTSH	31	Physical starting page of shared area
USTCT	33	Address of currently active TCB Used by the process scheduler to determine if task state needs changing when a TCB is selected to run. Updated if a different TCB is run.
USTAC	35	Start of active TCB chain (aka USTAQD1) Used as the input to the queue search instruction that locates the highest-priority task to run. Modified if the first TCB on the chain is selected to run and there are other TCBs with the same priority (round robin).
USTAQD2	37	End of active TCB chain (see USTAC)
USTFC	41	Start of free TCB chain (aka USTFQD1)
USTFQD2	43	End of free TCB chain
USTFL	45	UST flag word (see below)

User Status Table Flags - USTFL

?UFDR	1b2	Inhibit scheduling
?UFDB	1b3	Process is being debugged
?UFPH	1b5	Scheduling disabled by Agent

3.9 PTBL Scheduling Details

Scheduling a process' activity is a complex problem. The remainder of this chapter is devoted to the details. This section presents some reasons and background for the actions taken by various paths in the code. It may be helpful to skim this section before trying to read the pseudocode later or the system sources themselves. This section does not stand alone, but is intended to be a companion to the system sources and pseudocode.

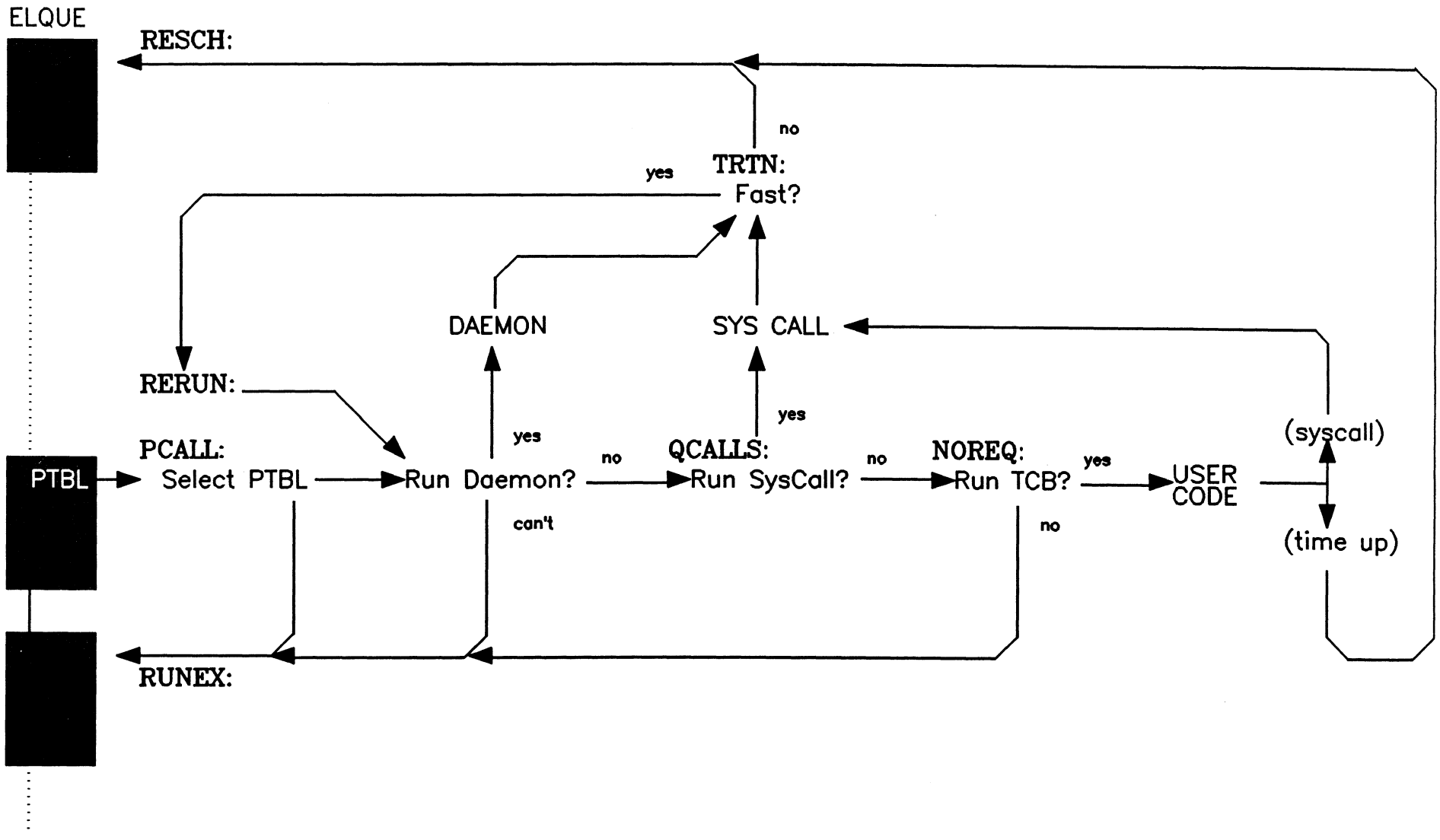
3.9.1 General Outline

When a process table (PTBL) is selected from the ELQUE to run, the process scheduler goes through three major steps. First the PTBL status is checked in more detail to see if it is truly eligible and able to run. Then the PTBL is checked for ring 0 actions that need to take place. Lastly, the user task world is scheduled and dispatched if all the other checks allow.

There are three basic ways that the process gives up its control. If the PTBL cannot run or spawn activity it will usually transfer control back to the ELQUE scanner so it can continue. Daemons and system calls return to TRTN, which will check to see if the CB ever pended. If it did pend, then the ELQUE scan will resume from the top. If the call or daemon never pended, then the PTBL will be restarted at RERUN. If user code is interrupted for a subslice end then the system will be rescheduled at RESCH (SMON0).

This overall flow is shown in the figure.

PTBL Scheduling Flow



3.9.2 The Process Scheduling Code Segments

Both the detail code discussion and the pseudocode are divided into sections that roughly correspond to major activity or decision points in the code. What follows is an overview of how they fit together.

PCALL is the first entry point for scheduling a PTBL. It is called from the system scheduler or sometimes if the PTBL scheduling must be restarted. This section may abandon the PTBL scheduling and go back to the scheduler or continue at PCAL1 below.

RERUN is the entry point used to reschedule a PTBL if its time is not up. Since the PTBL is the currently active entity in the system, the processing here only needs to check for exceptional conditions. RERUN also continues at PCAL1.

PCAL1 is the main path of scheduling PTBL activity from either PCALL or RERUN. PCAL1 is not an external entry point. Depending on PTBL status, the code may go to PRBITS, QCALLS, or NOREQ.

PRBITS is the path that evaluates and dispatches high-priority activity for a PTBL and also executes some of these actions. These actions are high-priority because they must be taken care of before system calls or user code can even be considered. This section may vector off to PBLOCK, PDMPR, or QCALLS within Process Scheduling, or it may call external routines which do not return.

PBLOCK implements the high-priority action of moving a blocked PTBL off of the ELQUE. This code section will jump back to PCAL1 if the process is terminating, since it makes no sense to block it. Otherwise PBLOCK exits Process Scheduling. On exit the PTBL will either have been moved to a blocked queue or it will still be marked blocked on the ELQUE.

PDMPR handles the high-priority activity of invoking daemons. Depending on the state of the PTBL and the individual daemon, request flags PDMPR may transfer control to RERUN or STKCT, or it may leave Process Scheduling entirely.

QCALLS checks for waiting system calls. If one or more are waiting and can be started, QCALLS will transfer to STKCT which is the main path for starting up a CB. If an illegal system call is found waiting QCALLS will handle it and resume PTBL processing at RERUN. If system calls can't or won't be run, then QCALLS transfers to NOREQ or REQ (within the NOREQ section).

STKCK is the common path for spawning a CB on behalf of a process. Both daemons and system calls take this path. STKCK is an entry point used by the system call entry point from the user or his Agent. After reserving and setting up a CB the system can go in several possible directions. The code may exit Process Scheduling and proceed on the CB just set up. It may have to delay starting the CB and exit Process Scheduling back to the system scheduler. Lastly it may put the CB on the ELQUE to be run on another JP and jump back to RERUN to look for something new to do.

NOREQ is encountered if no other action is possible or necessary and the user code can be considered. If the process is in a state where the user cannot be run then NOREQ leaves Process Scheduling for the system scheduler. If the process is in a state that requires that it be restarted exactly where it left off then NOREQ will exit to the user. Otherwise it will transfer to PSCHD to schedule or reschedule a user task.

PSCHD is the user task scheduler. It will transfer either to PTFN1 if a task is selected to run or to PNTRY if no task can be selected.

PTFN1 is the last Process Scheduling segment executed during the scheduling or rescheduling of user code. It will exit Process Scheduling either by starting a user task or by going back to the system scheduler if something forces rescheduling.

PNTRY handles the situation of finding no user task ready to run. It may continue in Process Scheduling at SYST3 or exit Process Scheduling to the system scheduler.

SYST3 decides if a process should be blocked when no user tasks are found ready to run. It will exit Process Scheduling to the system scheduler either to continue the ELQUE scan or to restart it.

3.9.3 Detailed Discussion of Code Segments

PCALL - Start running a process

This is the entry from the system scheduler (ELQUE management) when a PTBL is selected from the ELQUE to be run.

A PTBL is selected when it is the highest-priority element that satisfies the scheduler's queue search mask of PSTAT flags and PCLASS bits. All process tables contain the address of SCHED:PCALL in offset PPC.W. The code that vectors through PPC.W is at SCHED:M6.

While it only takes a few instruction cycles to start running a process table, the logic and the reasoning behind it takes some explaining.

To run a PTBL it must be marked "running." To do that the scheduler must lock the PTBL by setting flag PSTAT.PLOCK. This in turn requires transition lock PSTAT.PTRAN. This is a single bit, which if set indicates that the PTBL is unstable for a short period of time. Each of these locking requirements is shown in PARS.SR.

Normally the transition lock is acquired through the "spin-lock" mechanism. With interrupts off the code attempts to get the lock using an indivisible bit test and set instruction. If it fails then the code spins in a tight loop with interrupts on waiting for the lock bit to clear before trying all over again. Only in a multi-processor system will a spin-lock actually spin. In single CPU systems the test and set will always succeed.

The scheduler however will not go into a tight spin here. It will instead abandon the PTBL and continue the ELQUE scan if it finds the PTBL already in transition.

After putting the process table in transition the scheduler uses the same test and set instruction to mark the PTBL locked (set PSTAT.PLOCK true). If it is already locked then again the scheduler will skip the PTBL and continue the ELQUE scan, after clearing the transition lock of course.

Once the PTBL has been locked by setting flag PSTAT.PLOCK then SCHED abandons its quick return to the ELQUE scan by popping the queue search mask off of the stack and decrementing ELQUE's scanner count. The scheduler may later resume the ELQUE scan but it will have to re-establish the search mask and increment the scanner count.

With the PTBL locked, SCHED sets the current system context in CC.W to the PTBL address, marks the PTBL running (sets PSTAT.PSRUN), and finally clears the transition lock bit PSTAT.PTRAN to release other spinning waiters.

Now that the critical testing is done and the PTBL is locked the scheduler re-enables interrupts and goes to SCHED:PCAL1. PCAL1 is the common path for either running a process (this path) or re-running a process (at SCHED:RERUN).

RERUN - Continue running a process

RERUN is called to continue a process after some ring 0 activity if it didn't pend and didn't use up its subslice. The PTBL must be running and locked and it must be the current context in CC.W. Other code paths in SCHED return to this location after changing the PTBL status or spawning a CB to run on another processor. It is called from SCMOD if a user task pends but another task might be able to run during the process' subslice. SCPRC comes here if it was unable to start a user's system call but another task might run.

The only action taken at RERUN is to check the status of the physical processor's reschedule flag. If the flag is found off (=0) then the code jumps to PCAL1 to join the PCALL code. If the reschedule flag is found set then the PTBL is released, which unpends any waiters, and its running flag is reset. Note that resetting the running flag (PSTAT.PSRUN) does not require that the PTBL be locked nor that it be put in the transition state (PSTAT.PTRAN set). The code then jumps to SCHED:RESCH which scans the ELQUE from the beginning.

PCAL1 - Common path to schedule activity for a PTBL

The first step in scheduling a process is to see if there are any high-priority actions that need to be taken. These actions are flagged by bits in the PSTAT word of the PTBL. If any of these bits are true then the code continues at PRBITS.

Then bit PFQSC in PFLG2 (bit offset BPFQS) is checked to see if the chain of waiting system calls should be checked. If this bit is set then the code goes directly to NOREQ to try to run user code. This flag is set when the process is about to be swapped out.

PRBITS - Evaluate high-priority actions

One or more of the high-priority action bits in PSTAT are set. The relative priority of each action is determined by the location of the bit in PSTAT. Lower number bits are used for higher priority actions. The flag bits and their meaning are shown below in decreasing order of priority.

Name	Bit Mask	Bit Offset	High-Priority Action indicated
PSEW	1B2	BPSEW	skip PTBL, scheduling activity in progress
PSBRK	1B4	BPSBR	handle operator interrupt (console abort)
PSBAG	1B5	BPSBG	swap process out
PSBLK	1B6	BPSBL	block the process
PSDP	1B7	BPSDP	start a daemon
PSMWT	1B8	BPSMW	waiting for memory, check global MKEY
PSTSU	1B9	BPSTU	timeslice is up

Bit PSEW indicates that the process cannot be run because there is a CB in the system performing some action to the process, either a daemon or a system call. This check is necessary even though PSEW is in the ELQUE scan mask, because an interrupt or another processor may have set the bit in the interval between selection and locking of the PTBL. If this bit it set then the scheduler resumes the ELQUE scan with the element after this PTBL.

PSBRK is a separate high-priority item, even though it is handled by a daemon, because the action of aborting a process is a higher priority than swapping or blocking. Before running the daemon the PTBL status must be adjusted. PSTAT flag PSNCB ("needs CB") is forced to zero so that the PTLB can more effectively compete for CBs to finish the termination. Flag PSBRK is reset and flag PSDP ("start a daemon") is set to have the daemon path taken. Then the code jumps to PDMPR to actually run the daemon.

PSBAG, swap out the process, invokes routine PRBAG in core manager module CORM2. This routine does not return.

PSBLK set indicates that the process is currently blocked but is still on the ELQUE for some reason. The code jumps to PBLOCK to attempt to clear this up.

PSDP indicates that one or more daemons is waiting to run. Go to PDMPR to try to start the highest-priority one.

PSMWT is set if a system call started earlier had to give up because it couldn't get all the memory it needed to complete. When it set itself up to be restarted, it copied global MKEY into PMKEY. If MKEY still matches PMKEY then no memory has been released and there is no sense in trying to start the system call now, so the code returns to the ELQUE scan where it left off. Otherwise, the PSMWT flag is reset and the code jumps to QCALLS to try to restart the waiting system call which may complete this time.

If PSTSU is set then the process' full timeslice has expired since the last time the process was scheduled. When this is detected the flag is reset, the process' address space is mapped in to make it available to the core manager, and the scheduler transfers control to TSUP to handle timeslice end in CORM2. This routine does not return.

If none of the bits mentioned above are found set then the scheduler panics with code 14003. The only PSTAT flag that can cause this is PSRDY ("not ready to run"). Since entry at PCALL from the ELQUE scanner only occurs when this flag is clear, the scheduler must have been entered at RERUN or improperly at PCALL.

PBLOCK - PTBL is blocked but still on ELQUE

The reasons this can happen include finding the blocked element queue in use (locked) when the process is blocked. The scheduler will try to move it off the ELQUE and onto the blocked queue now.

Several types of status that might prevent the PTBL from being blocked are locked by the element transition bit (PTRAN in PSTAT). So this lock is aquired via standard spin lock routine XLOCK.

Next the PTBL is rechecked to see if it is still blocked. If another JP or interrupt level has changed its status, then the scheduler skips the PTBL and continues the ELQUE scan.

Since is makes no sense to block a process that is terminating (and could cause a deadlock) the PFTRM bit is checked also. If set then the block is aborted, the transition lock released, and the code jumps back to PCAL1 to resume PTBL activity which should process or enable the termination.

If everything looks okay to allow the process to be blocked, release the transition lock and call CTBLK to move the PTBL to the blocked queue. If CTBLK fails then continue with the ELQUE scan. If the PTBL is moved to the blocked queue then other system paths that might be waiting for the PTBL to free up are unpended and the ELQUE scan is restarted at RESCH (SMON0).

PDMPR - Select and run a daemon

Before actually running a daemon, conditions must be verified. A daemon is like a parallel system call in that it cannot run if other actions are going on. If the PTBL is the target of another path's actions (PTRGC > 0) or if the process has system calls running (PSQCT in the extender > 0) then the PTBL is skipped and the ELQUE scan resumed.

If a daemon is started, then use the CB pointed to by global PCB.W and make sure that another is available to replace it in SCB.W. ALSTK is called to make sure that SCB.W does point to a replacement and that the replacement can come from the pool allocated to the PTBL's group (G1 or G2/3). If no suitable CB is available, the scheduler will skip this PTBL and continue the ELQUE scan after one more check. If this is a group 2/3 PTBL, then status bit PSNCB ("needs CB") is set so that the PTBL will be selected later only when group 2/3 CBs are available.

Conditions are now right to start a daemon. Set up the globals to indicate that a daemon is to be run. The system call number (CALL.W) in the CB pointed to by PCB.W is set to -1 and PTMP1, 2 and 3.W are set to zero. PTMP3.W will also be used below in the search for daemons needing to be run. Map the PTBL's address space for later use. Clear the daemon request bit under the assumption that only one daemon request will be found and it will be spawned.

The code segment at BSR1 goes through the daemon request table (SCHED:BTBL, not be confused with global BTBL) looking for one or more daemon request bits. BTBL contains bit offset - daemon routine address pairs. The end of the table is marked by a bit offset of -1. For each bit offset the loop tests the indicated flag and if the bit is set then a daemon request has been found. If PTMP3.W is zero then this is the first daemon request found and its handler address is stored in PTMP3.W and the loop continues. If PTMP3.W is non-zero then this is the second daemon request found and status bit PSDP ("run daemon") is set in PSTAT to be found later. The loop is also exited, since there is no sense looking further.

If PTMP3.W is still zero after the loop, then no specific daemon request bit was found. This fact is simply shrugged off and the PTBL scheduler restarts itself at RERUN assuming that this happened because of some race condition.

If PTMP3.W is non-zero then everything has already been set up to spawn a daemon on a CB. PTMP2.W is zero indicating no TCB is involved in the system task, PTMP3.W is the address of the code path, and CALLN.W in the CB (pointed to by PCB.W) contains -1 indicating a daemon. PTMP1.W isn't used. The daemon startup joins system call startup at STKCT in the NODCL segment.

QCALLS - Check for waiting system calls

After finding no high-priority actions and before trying to run the user the scheduler tries to start any system calls that were held up. A system call can be held up if it is a parallel type call requiring complete control of the PTBL but a call is already running, or if the PTBL is the target of another call. It can also be held up if the user already is running his limit of active system calls (PSQMX in the extender). A system call can restart itself if it fails to get the memory it needs to complete and goes to TRTN with error ERRST ("memory restart"). When this happens the system call error path will set high-priority bit PSMWT (bit offset BPSMW) and copy the global MKEY into PTBL location PMKEY. When the global memory key changes, then the PRBITS path will come here.

To spawn a system call a CB must be available in global SCB.W to replace the one in PCB.W which will be used for the call. The system limits the system paths (CBs) that can be active for group 1 processes independently of group 2 and 3 processes. Routine ALSTK handles this complication by checking the appropriate CB pool counter and assigning a CB to SCB.W according to the group of the PTBL. ALSTK does not actually take a CB from the pool since the call might not start for other reasons.

If no CB is available then a check is made to see if the user code can be entered. If so then the path jumps to REQ to schedule a user task. If not then the PTBL is skipped and the ELQUE scan continued.

The process of spawning a system call can now proceed in earnest. The user's full address space is mapped into the ATU so that the system call data can be read from the TCB. The system call number is checked to see if it is in the range of legal numbers. This check must be made because the call number will later be used to index into various tables. If the system call number is illegal then the task is set up to take the error return with error ERICM "Illegal system call" and process scheduling is started over at RERUN.

Next the scheduler checks to see if the call can continue under current conditions. If there is already a parallel call running or if the new call is a parallel call and other activity is in progress or if the user is already running his quota of active system calls, then the new call must wait and the scheduler proceeds to NOREQ as if this had never happened.

At this point it is almost certain that the waiting call will be spawned off on a CB so the TCB is unlinked from the chain of waiting calls and PTMP2.W is set to the TCB address. In an attempt to avoid doing a lot of checks later only to find the process unable to run, make a couple of checks now. If no other TCBs are waiting to start system calls or if the PSTAT bit PSETR ("don't enter user") is set then the PSTAT bit PSRDY ("not ready to run") will be set. At the same time, PSETR and PSNCB ("needs CB") are reset, since they will no longer be needed to prevent selection of this PTBL. In theory, this will save time later.

At SCHED:NODCL the code verifies that the call is indeed valid by indexing into table MCCT.W to get the system call handler address. If the handler address is -1 then the call will be aborted. The abort is handled at CMER2 where the TCB is set up to take the error return with error ERICM and the PTBL is conditioned to allow the user task to resume. The scheduler restarts at RERUN to continue scheduling the process.

If the call is valid then the call number is put into CALLN.W of the CB that will be used (pointed to by PCB.W). If the system call count table was allocated then the call will be counted.

STKCK - Spawning a CB for system call or daemon processing

At this point running a system call and running a daemon are almost identical. Global variables PTMP1, 2, & 3.W contain the system call number, TCB address of user task, and code path address respectively for a system call. For a daemon PTMP1 & 2.W are zero. The CALLN.W of the primary CB (PCB.W) also contains the system call number or a -1 for a daemon.

The active system path count for the process is incremented to show that a CB is active for the process.

If a daemon is to be started then status bit PSEW ("scheduler action") is set in PSTAT to prevent the PTBL from being scheduled again until the daemon is finished.

Either RSTKCT (for group 1) or SSTKCT (for group 2/3) is decremented to indicate that the CB allocated earlier by ALSTK in SCB.W is really being used as the backup to the CB in PCB.W.

Next the CB is initialized. Errors and times are zeroed since nothing has happened to the CB yet. The fatal error handler address is set to -1 so the system will panic if a fatal error occurs while running this CB unless some other arrangements are made. Globals are used to identify the associated TCB, PTBL, and logical processor. Data is copied from the PTBL to identify the class the CB should run under. Mother-only system calls or daemons are assigned no class mask bits so they can run independent of class.

The stack pointers in the CB and the stack registers in the hardware are initialized for an empty stack in the CB. At this point the system is really running off of the CB, only CC.W points to the PTBL to indicate that the switch is not complete.

The scheduler must check if the code path can be run on the JP that is setting all this up. If the current JP is not the mother processor and either the path is a daemon or the call is a mother-only call then the CB cannot run on the current JP and must be switched to the mother JP. This is done at MONLY1 below.

The final check is to see if something has happened to require the JP to reschedule while the CB was being set up. If the reschedule flag for the JP is set then both the CB and PTBL must be released onto the ELQUE to be scheduled later. The CB is moved to the ELQUE with the required status by FIXCB. SETUP is called to put a return block into the CBs stack and replenish PCB.W from SCB.W. Both the CB and PTBL running flags (PSRUN in PSTAT) are cleared, and the PTBL is released. Finally the scheduler restarts the ELQUE scan at RESCH (SMON0).

If the reschedule flag is clear then the CB will be started. The JP's fault block (FBLK.W at harware location 32 octal) is set to the block in the CB. The PTBL is released by clearing bit PSRUN ("running") in its status word. CC.W, the address of the current context block (PTBL or CB) is set to the address of the CB. Timing is switched from the PTBL to the CB. The count of CBs started (TINDST.W) is incremented. And finally the code path is started by jumping through PTMP3.W.

At MONLY1 the code must put the CB onto the ELQUE for the mother processor to schedule. Again FIXUP and SETUP are called and the CB released by clearing its running flag. MEVENT is called to request the mother to reschedule if the new CB is a higher priority than what she is doing. Global metering location NMONLY is bumped to count the number of times that a CB was transferred to the mother JP. Finally the scheduler either restarts the PTBL if it is still ready and the JP hasn't been told to reschedule, or the scheduler skips this PTBL and continues the ELQUE scan.

NOREQ - Not starting ring 0 activity, run user?

If PSTAT bit PSETR ("don't enter user") is set then the scheduler just resumes the ELQUE scan after the PTBL. PSETR (bit offset BPSEN) is set when the task scheduler below finds no user task ready to go but there is at least one system call waiting to start. If there were no system calls waiting then it would have set PSRDY ("not ready to run").

At SCHED:REQ the scheduler expects to start some user code path so it calls MAPCON to map the process' address space into the ATU and establish it as the current context. If flag word PINSU in the PTBL is zero then the code goes to PSCHD below to schedule a task. Otherwise, the process was interrupted during some system activity and must be restarted.

To clear the PTBL and restart where it left off, the scheduler clears flag bit PFIRS ("Interrupted by system," bit offset BPFIR) in PFLG3 and fills in the global current task pointer CTSK.W from the current task pointer USTCT.W in the process' UST. It also bumps counter NNORQ1.W to monitor how many times this happens. Routine RSTPRC is called to restore the process' state and start charging the user before transferring control.

Bit ?TSWP in TCB status word ?TSTAT is true if the task was interrupted by a page fault. In this case, the scheduler returns via a WDPOP instruction which uses the fault context block whose address is in FBLK.W (hardware location 32 octal). Otherwise, the return block in PTBL extender locations PMARG.W to PMPC.W is copied onto the ring 0 stack and a WPOPB returns control to the process.

PSCHD - The process task scheduler

Because the process is being scheduled or rescheduled, bit PFRSH ("please reschedule process") in flag word PFLAG is cleared before any other processing continues.

The first check is whether or not task rescheduling should be done. If flag bit ?UPPH in USTFL of the UST is set then there is no multitasking and hence no rescheduling is needed. If bit ?UFDR is set then tasking is disabled and no rescheduling should be done. If location ?TSMA is non-zero in ring 3 then the Agent has the UST or TCBs in transition and they cannot be trusted to do a reschedule. If any of these tests are true, then the scheduler jumps to PSCDR to restart the process where it left off.

At PSCDR the scheduler checks to see if the current task can be started or if the process cannot continue. If none of the pend bits in TCB status word TSTAT are set or the task is returning from a page fault then the task should be restarted and the scheduler jumps to PTFN1 below to do so. It is returning from a page fault if flag bit ?TSWP ("faulting") is set and ?TSPN ("pended on fault") is clear in TCB status word TSTAT. If the task is pended but not returning from a page fault, then it cannot be restarted and the code goes to PNTRY (below) to figure out what to do next.

If task scheduling is found enabled, then monitoring location NPLSP.W is incremented to count the number of times that this was found true.

The scheduler next scans the TCB queue in ring 3 for the highest-priority ready task. The search criteria is that no pend bits be set in TCB status word TSTAT. The mask of pend bits is called CRUMS and is defined above. If the scan finds no ready tasks, then the scheduler jumps to PNTRY to determine how to proceed.

Once a TCB has been selected it is made the current TCB in UST location USTCT. Next the scheduler checks to see if the TCB should be moved back in the TCB queue to implement a round-robin. Only if there are other TCBs of the same priority behind the selected TCB will the TCB have to be moved. It doesn't matter if other TCBs with the same priority are ahead of the selected TCB.

If a round-robin needs to be done, the location in the queue for the TCB is found by searching backwards in the queue for the last TCB with a priority equal to the selected TCB. Once the location is found, then the selected TCB is dequeued from its original position and enqueued in its new place in the queue.

PTFN1 - A task has been selected to run

First point the JP's current task address to the selected TCB the code decides how the task's stack should be established. If this is the same task as was last run, then its stack is already okay. If this is the first task run (PEXTN item PCTSK.W = 0) then its stack needs to be set up by TSKRST but nothing else is required. If this is a different task from the last one, then the earlier task's stack must be saved by TSKSAV before TSKRST sets up the new task's stack. Stacks are set up in every user ring that has a valid page zero in the map.

Next the extended variable save area must be set up for the selected task. If this is not a new task then this is skipped. If the last task to use the extended save feature was this task then this is skipped. Otherwise the extended variable area must be changed. If the extended variable area was never used (PEXTN location PCEXV = 0) then nothing needs to be saved. If it was used, then the data in the process' common area must be saved into the area set aside by the prior task. After the WBLM then PCEXV is set to zero to indicate that no save is yet required. If the selected task does use the extended variable feature then its data is copied from its save area into the common area and PCEXV is set to its TCB address. This procedure protects a task's extended variables even if a task is run that doesn't use the feature.

The final state restored (other than the PC and accumulators) is the FPU. This is done by RSTPRC which also does a final check for JP rescheduling requests and starts charging CPU time to the user. If RSTPRC finds the JP's reschedule flag set it will not return but will abort this entire operation and go to RESCH (SMON0).

At this point the user will definitely be started, only the method of transferring needs to be determined. If the user entered the system for a fault, the return is via the fault block on the stack. Otherwise, the system forces flag PFIRS in PFLG3 to zero for use later and builds a return block on the ring 0 stack for the WPOPB.

Voila! The user process is running.

PNTRY - Cannot run user task

What happens next is dependent on what happened while the scheduler was going through the PTBL and TCBs. If an interrupt occurred and set flag PFIRS, then a task may have become ready and the scheduler will try again from PSCHD (after resetting PFIRS).

Now that it is certain that the user process cannot be run, the system will try to set up the PTBL so it doesn't waste time going all the way through this again. If the process has no TCBs waiting to start system calls (PEXTN location PSWD = 0) and no system calls active (PSQCT = 0) and no outstanding LPB or MTA requests (PSIOC = 0) then it is a candidate for blocking at SYST3 below.

If it cannot be blocked, then something can still be set. If there are no TCBs waiting to start system calls, then setting PSRDY ("not ready to run") will prevent this PTBL from being selected from the ELQUE. If there are TCBs waiting, then setting PSETR ("don't enter user") will mean that the scheduler won't try to run user code.

If this is a group 2 or 3 process and there are waiting system calls then setting PSNCB ("needs CB") will mean that the PTBL won't be selected unless group 2/3 CBs are available.

Finally the scheduler unlocks the PTBL, unpends any waiters, and resumes ELQUE scanning at RUNPTP.

SYST3 - Process is a candidate for being blocked

Since the process is idle, set PSRDY ("not ready to run") so it won't be selected to run until that changes. If the process cannot be blocked (bit PFBLE, "process can be blocked", in PFLG2 is zero), then jump to RUNPTP to release the PTBL and waiters and continue the ELQUE scan.

Block the process by setting bit PSBLK ("process is blocked") and clearing bit PSRUN ("running") in PSTAT. Also clear PSNCB ("needs CB") so that the PTBL won't later have to wait.

Call CTBLK to move the PTBL from the ELQUE to the blocked queue. If CTBLK fails then resume ELQUE scanning at RUNEXP. If CTBLK succeeds then the ELQUE has been changed and the scheduler must start all over again at RESCH (SMON0). In either case process scheduling has come to an end.

3.9.4 Pseudocode

```
/* ============================================================ */
/* SCHED:PCALL                                                  */
/* ------------------------------------------------------------ */
/* Run a process. Called from SCHED:M6 when a PTBL is dispatched */
/* (through PPC.W in the PTBL) by the scheduler from the ELQUE.  */
/* ------------------------------------------------------------ */

PCALL(,,ppcb,ptbl):

   PPCB *ppcb;    /* Phys Processor Control Blk addr in AC2 */
   PTBL *ptbl;    /* PTBL address in AC3 */

   /* To mark PTBL running we must acquire the lock bit.  To get */
   /* the lock bit we must acquire the transition bit. If either */
   /* of these bits is already set then we simply skip this PTBL */
   /* instead of "spinning" in the scheduler.                    */

   interrupts_off;

   /* Mark the PTBL "in transition" or skip it */
   if (ptbl->PSTAT.PTRAN)
      goto SCHED:RUNEX1(,,,ptbl);
   ptbl->PSTAT.PTRAN = 1;

   /* Lock the PTBL or skip it */
   if (ptbl->PSTAT.PLCK)
      {
      ptbl->PSTAT.PTRAN = 0; /* clear transition flag */
      goto SCHED:RUNEX1(,,,ptbl);
      }
   ptbl->PSTAT.PLCK = 1;

   /* At this point we won't continue the ELQUE scan so take the   */
   /* queue search mask from the stack and decrement the ELQUE     */
   /* scanner count.                                               */

   (void) WPOP();
   ELQUE.QSCAN--;

   /* We now have this PTBL.  Set it up as the current context and */
   /* mark it running and not in transition.                       */
```

```
   CC.W = ptbl;
   ptbl->PSTAT.PSRUN = 1;
   ptbl->PSTAT.PTRAN = 0;

   interrupts_on;

   goto PCAL1; /* continues below RERUN */

/* ============================================================== */
/* SCHED:RERUN                                                    */
/* -------------------------------------------------------------- */
/* Called from other locations in SCHED when conditions prevent a */
/* previously selected action to start but other actions might be */
/* possible.                                                      */
/* Called from SCMOD if a task pends on a signal but the process  */
/* timeslice is not up and the process might be able to continue. */
/* Called from SCPRC when a system call cannot be started         */
/* immediately but other tasks in the process might be able to    */
/* continue.                                                      */
/* In all cases the process is already running and is the current */
/* context (in CC.W).                                             */
/* -------------------------------------------------------------- */

RERUN(,,,):

   PTBL *ptbl;

   /* If the reschedule bit is set in our JP's control block then */
   /* stop the current process and reschedule the JP.  Otherwise  */
   /* continue running the same process.                          */

   ptbl = CC.W;
   if (MYPPCB.W->CPSTAT.PRESCH)
      {
      RELPTBL( ptbl );
      ptbl->PSTAT.PSRUN = 0;
      goto SCHED:RESCH(,,,);
      }
   goto PCAL1;

/* ============================================================== */
/* SCHED:PCAL1                                                    */
/* -------------------------------------------------------------- */
/* Common path from PCALL and RERUN.                              */
/* -------------------------------------------------------------- */
```

```
PCAL1:
   /* pseudocode variables */

   short int hi_pri;         /* subset of PTBL.PSTAT bits */
   short int PSBIT =         /* mask of high-priority PSTAT bits */
      ~(  PSRUN || PSETR || PSFSY || PSNCB
        || PMAST || PLCK  || PTRAN || PNTCB );
        /* (Note: Mask is built by excluding don't care bits. */
        /*        PSRDY ("not ready to run") is NOT excluded. */

   /* ------------------------------------------------------------ */
   /* Before looking for tasks check for any high-priority things */
   /* to do on/for/to the process.                                 */

   hi_pri = ptbl->PSTAT & PSBIT;
   if (hi_pri)
      {
      /* SCHED:PRBITS ------------------------------------------- */
      /* This pseudocode is written using in-line if statements  */
      /* to show that machine instructions LOB followed by (ZEX  */
      /* and) LDSP result in an ordered test for flag bits.      */
      /* The priority of each flag is determined by its location */
      /* in the word, lower number bits are higher priority.     */
      /* The code below reflects this priority ordering.         */

      if (hi_pri.PSEW) /* Scheduler action pending */
         goto SCHED:RUNEXP(,,,ptbl); /* skip this PTBL */

      else if (hi_pri.PSBRK) /* ^C^B pending */

         /* SCHED:PRTRM -------------------------------------- */
         /* Even though console abort is handled by a daemon it is */
         /* a higher priority than swap or block processing and so */
         /* has its own high-priority bit.  Set up PTBL.PSTAT like */
         /* other daemon requests and clear the "needs CB" flag    */
         /* (PSTAT.PSNCB) so the abort won't get hung up on it and */
         /* join common daemon start-up path.                      */
         {
         ptbl->PSTAT.PSBRK = 0;
         ptbl->PSTAT.PSDP  = 1;
         ptbl->PSTAT.PSNCB = 0;
         goto PDMPR;
         }
      else if (hi_pri.PSBAG) /* Start swap-out */
         goto CORM2:PRBAG(,,,ptbl);

      else if (hi_pri.PSBLK) /* Process was blocked */
         goto PBLOCK;        /* move PTBL off the ELQUE */

      else if (hi_pri.PSDP ) /* Run a Daemon */
         goto PDMPR;

      else if (hi_pri.PSMWT) /* Process waiting for memory */
```

```
      /* SCHED:PMWT ------------------------------------------ */
      /* Check global MKEY against PTBL.PMKEY which was the    */
      /* MKEY value when the process tried last time.  If MKEY */
      /* is the same then process still cannot get memory      */
      /* so just skip it.                                      */
      {
      if (ptbl->PMKEY == MKEY) goto SCHED:RUNEXP();
      ptbl->PSTAT.PSMWT = 0;
      goto QCALLS;
      }

   else if (hi_pri.PSTSU) /* TimeSlice is Up */

      /* SCHED:PTUP ------------------------------------------ */
      /* Timeslice expired because of activity on a CB. Map in */
      /* process context for timeslice end routine.            */
      {
      ptbl->PSTAT.PSTSU = 0;
      MAPCON( ptbl );
      goto CORM2:TSUP(,,,ptbl);
      }

   /* Because of the construction of mask PSBIT we can only    */
   /* get here if bit ptbl.PSTAT.PSRDY ("not ready to run") is */
   /* true.  But since this path should only be taken if the   */
   /* PTBL is ready then we panic.  The scheduler ELQUE scan   */
   /* mask always checks for PSRDY so only the RERUN path can  */
   /* cause this panic.                                        */

   else PANIC (PNICE+P14AC); /* 14003 */
   }


/* ---------------------------------------------------------- */
/* There are no high-priority tasks to do for this PTBL so    */
/* now check to start system calls that couldn't be started   */
/* before.  If flag PFLG2.PFSQC is set then the calls weren't */
/* able to get a CB/stack allocated and still cannot be started*/
/* until a suitable CB is available and the flag is reset.    */

if (ptbl->PFLG2.PFQSC) goto NOREQ;

goto QCALLS; /* jump around daemon processing */
             /* (source just falls through ) */


/* SCHED:PBLOCK ---------------------------------------------- */
/* The PTBL is marked as being blocked but found on the ELQUE, */
/* called from the check of high-priority PSTAT bits above.   */
/* This code will verify that it should indeed be blocked and */
/* if so then try to move the PTBL from the ELQUE to the      */
/* blocked queue.                                             */
```

```
PBLOCK:
   /* Moving from the ELQUE to the blocked queue is protected by  */
   /* the transition lock.                                        */

   XLOCK( BPTRAN, ptbl);

   /* If process isn't still blocked then lets skip it */

   if (ptbl->PSTAT.PSBLK == 0)
      goto SCHED:RUNPTP(,BPTRAN,,ptbl);

   /* Before blocking it make sure it doesn't need to terminate */

   if (ptbl->PFLAG.PFTRM == 1)

      /* Process is about to terminate.  Zap the blocked bit,    */
      /* release the PTBL, and let PCAL1 select term daemon.     */
      {
      ptbl->PSTAT.PSBLK = 0;
      ptbl->PSTAT.PTRAN = 0; /* releases PTBL */
      goto PCAL1;
      }

   /* Process can be blocked. Reset the PTBL running bit, call    */
   /* CTBLK to move it to the blocked queue, release the PTBL and */
   /* anyone waiting on its resources and finally reschedule from */
   /* the top.  If there is a problem moving the PTBL to the      */
   /* blocked queue then just skip over the PTBL in the ELQUE.    */

   ptbl->PSTAT.PSRUN = 0;

   if (OK == CTBLK(ptbl))
      {
      RELPTBL( ptbl );
      goto RESCH(,,,);   /* source uses SMON0 but RESCH is same */
      }
   else
      goto RUNEXP(,,,ptbl);


   /* SCHED:BTBL ------------------------------------------------ */
   /* Daemon request list used by the PDMPR code below to select a */
   /* specific daemon to run.                                      */
   /* Each entry in the list consists of the bit offset to a flag  */
   /* somewhere in the PTBL that signals a daemon action and the   */
   /* address of the routine to execute the daemon.  The end of the */
   /* list is indicated by an entry with its bit offset set to -1. */

   typedef struct Daemon_Request_List
      {
      short int    dmon_bit_loc;
      extern void *dmon_code_addr( );
      };
```

```
   /* For each bit offset in the list the comment shows the name */
   /* of the status bit and the PTBL entry containing the bit.   */

   static Daemon_Request_List
      BTBL [] =
         {BPFTM, RET.P},      /* PFLAG.PFTRM termination */
         {BPFTL, PRNCI.P},    /* PFLG2.PFATL fatal error */
         {BPFIR, IPRLD.P},    /* PFLAG.PFFIL initial load */
         {BPFIW, PRNCI.P},    /* PFLG3.PFIWC ^C^B console abort */
         {BPUCT, UCTD.P},     /* PFLG5.PFUTC process trap */
         {BPFNT, NTRES.P},    /* PFLG2.PFNTR 16bit become resident */
         {BPFNF, NFRES.P},    /* PFLAG.PFNFR 16bit become nonresident */
         {BPFIN, PROIN.P},    /* PFLAG.PFINT ^C^A console interrupt */
         {BPISS, PRKIN.P},    /* PFLG4.PFISS user console interrupts */
         -1;                  /* end of list */


   /* SCHED:PDMPR ------------------------------------------------- */
   /* Run a DAEMON.                                                 */

PDMPR:

   /* pseudocode variables ---------------------------------------- */
   Daemon_Request_List *btbl_ptr;

   /* ------------------------------------------------------------- */
   /* A daemon can only run for a process that is not the target of  */
   /* another process' action and has no system calls in progress.   */
   /* If the associated counters in the PTBL and extender are not    */
   /* zero then skip this PTBL.                                      */

   if (( ptbl->PTRGC != 0 ) || ( ptbl->PEXTN.W->PSQCT != 0 ))
      goto SCHED:RUNEXP(,,,ptbl);

   /* Before we start a call and use the primary CB (*PCB.W) we must */
   /* allocate a secondary CB (*SCB.W).  ALSTK does this and checks  */
   /* that we aren't exceeding the allocation of active CBs for the  */
   /* process' group (G1 or G2/3).                                   */

   if (NOT_OK == ALSTK( ptbl ))

      /* No CB available to start daemon so we will skip the PTBL on */
      /* this ELQUE pass.  To save scheduler time later we first set */
      /* the "needs CB" flag (PSTAT.PSNCB) if the process is not a   */
      /* group 1 (G1) process.                                       */
      {
      if (ptbl->PPRI > G1)
         ptbl->PSTAT.PSNCB = 1;
      goto SCHED:RUNEXP(,,,ptbl);
      }
```

```
/* OK to run a daemon.  Set up CB *PCB.W and the global process   */
/* temps in case we actually do run one (SCHED:STKCK uses global  */
/* PTMP2.W to indicate a daemon or system call).  Map in the      */
/* process' context and clear the "run daemon" flag under the     */
/* assumption that there is only one daemon request flag set.     */

PCB.W->CALLN.W = -1; /* no system call word */
PTMP1.W = 0;  /* ?? */
PTMP2.W = 0;  /* usually TCB address, 0 = daemon (not TCB) */
PTMP3.W = 0;  /* daemon code address, 0 = no daemon found */

MAPCON( ptbl );


/* SCHED:BSR1 ------------------------------------------------- */
/* Figure what daemon, if any, we are going to run.               */
/* We have to scan the table past the first one found to see if   */
/* more than one daemon needs to run.  If so we will have to make */
/* flag PSTAT.PSDPR true for later.                               */

for (btbl_ptr = &BTBL[0];
     btbl_ptr->dmon_bit_loc != -1;
     btbl_ptr++)
   {
   /* Test PTBL bit pointed to by this dmon_bit_loc.  */

   if (Bit_is_set( ptbl, dmon_bit_loc))

      /* Found a daemon request flag set.
      if (PTMP3.W == 0)

         /* Found first daemon request.  Save addr and continue. */
         PTMP3.W = btbl_ptr->dmon_code_addr;

      else
         /* More than one daemon requested.  Set "run daemon"    */
         /* flag and clear "needs CB" in PTBL so other daemon(s) */
         /* can go as soon as possible.  Then jump into the      */
         /* daemon/system call dispatcher STKCK.                 */
         {
         ptbl->PSTAT.PSDP  = 1;
         ptbl->PSTAT.PSNCB = 0;
         goto STKCK(,,,ptbl);
         }
   }
/* BTBL scan complete and there weren't multiple requests.  If  */
/* PTMP3.W is still zero then no daemon request was found and   */
/* we will assume there is some race condition in the system    */
/* and just restart this PTBL.                                  */
```

```
   if (PTMP3.W == 0)
      goto RERUN(,,,);
   else
      goto STKCK(,,,ptbl);


   /* SCHED:QCALLS ------------------------------------------- */
   /* We've decided to start (or restart) a waiting system call.  */
   /* Of course we must check that there really is one waiting.   */
   /* The high-priority path above that found MKEY changed enters */
   /* here to continue a waiting system call.                     */

   /* pseudocode variables ----------------------------------- */
   TCB       *tcb,*next_tcb /* address of TCBs */
   long int  syscall_nbr;   /* system call number from TCB */

QCALLS:

   /* get TCB address and verify there is one waiting */
   tcb = ptbl->PEXTN.W->PSWD.W;
   if (tcb == 0) goto NOREQ;

   /* The system call/daemon dispatcher (SCHED:STKCK) expects     */
   /* global PTMP2.W to contain either a TCB address indicating a */
   /* system call or zero indicating a daemon.  PTMP2.W also      */
   /* saves the TCB address across the call to ALSTK as mentioned */
   /* in the source.                                              */

   PTMP2.W = tcb;

   /* There is at least one system call waiting to start.  Before */
   /* starting it and using up the primary CB (*PCB.W) we must    */
   /* check that the CB allocation for the process' group (G1 or  */
   /* G2/3) hasn't been used up and that a secondary CB (*SCB.W)  */
   /* can be allocated.                                           */

   if (NOT_OK == ALSTK( ptbl ))

      /* No CB available but we might still be able to run a task */
      /* in the user ring.  Only when the "don't enter user ring" */
      /* flag (PSTAT.PSETR) is set do we give up on the process.  */
      /* In that case we mark low-priority (G2/3) processes as    */
      /* "needs CB" (PSTAT:PSNCB) so the scheduler won't pick     */
      /* them again until CBs are available.                      */
      {
      if (ptbl->PSTAT.PSETR == 0) goto REQ; /* run user task */
```

```
   if (ptbl->PPRI > G1)
      ptbl->PSTAT.PSNCB = 1;
   goto SCHED:RUNEXP(,,,ptbl); /* skip this PTBL */
   }
/* SCHED:TCBCL ------------------------------------------------ */
/* We now have a TCB and CB and can think about starting a     */
/* waiting system call.  Based on the system call number we    */
/* may 1) bag it because the system call number is invalid, or */
/* 2) be unable to run it because it conflicts with other      */
/* system calls in progress, or 3) start the call.  In cases   */
/* 1 and 2 we will rerun the PTBL from the start just in case  */
/* the process can do something else.                          */

/* restore PTBL and TCB address and map PTBL extensions */
ptbl = CC.W;
MAPCON( ptbl );
tcb = PTMP2.W;

/* Verify that the system call number is in range.  Otherwise */
/* the other checks that use the call number as an index into */
/* a table won't work correctly.  If the system call is not   */
/* valid then get rid of it and rerun this PTBL.  The process */
/* will definitely have something to do even if it is only to */
/* have the system call return with error.                    */

syscall_nbr = tcb->TSYS.W & (?PCMSK-SYSTRT);
if ((0 > syscall_nbr) || (syscall_nbr > MAXSYS))

   /* SCHED:CMER1 ------------------------------------------- */
   /* The system call number is out of range.  Remove this    */
   /* call from the queue, set up TCB and PTBL for error      */
   /* return ERICM ("Illegal system call"), and restart the   */
   /* PTBL from the top.                                      */
   {
   ptbl->PSIDIR--; /* one less indirect call queued */
   ptbl->PEXTN.W->PSWD.W = tcb->TSLK.W;
   run_invalid_syscall( ptbl, tcb );          /* SCHED:CMER3 */
   goto RERUN(,,,);
   }

/* Verify that we can start another call.  First check that   */
/* there isn't a parallel call already running and then see   */
/* that the process hasn't reached the limit on the number of */
/* calls it can have active concurrently.                     */

if (ptbl->PFLG3.PFPCH)
   goto NOREQ;
if (ptbl->PEXTN.W->PSQCT >= ptbl->PEXTN.W->PSQMX)
   goto NOREQ;
```

```
/* Finally check table CWTB to see if the call is a parallel  */
/* type call.  Parallel calls are like a daemons in that they */
/* require exclusive "use" of the process.  They can only run */
/* if there are no other calls active and if the process is   */
/* not the target of another process' actions.  The target    */
/* call count is in the PTBL and the active call count is in  */
/* the extender.  If we cannot start the parallel call then   */
/* go check for other things to do.  Otherwise, indicate that */
/* a parallel call is running so no other action can start.   */

if (CWTB[ syscall_nbr ])
   {
   if ((ptbl->PEXTN->PSQCT !=0) || (ptbl->PTRGC != 0))
      goto NOREQ;
   ptbl->PFLG3.PFPCH = 1; /* mark parallel call running */
   }

/* SCHED:DEQIT ------------------------------------------------ */
/* At this point we have the address of a TCB waiting to start */
/* or restart an indirect system call.  Take this TCB off of   */
/* the front of the chain of waiting TCBs and decrement the    */
/* count of waiting indirect calls.                            */

next_tcb = tcb->TSLK.W;
ptbl->PEXTN.W->PSWD.W = next_tcb;
ptbl->PSIDIR--;

/* To save unnecessary scheduling in the future we next check  */
/* if the process would be able to do anything after the       */
/* waiting system call is started.  If there are no more TCBs  */
/* waiting to start a system call and if the process is marked */
/* "don't enter user" then the process will be set so it won't */
/* be selected to run again until something changes.            */

if ((next_tcb == 0) && (ptbl->PSTAT.PSETR))

   /* The PTBL doesn't have anything else to do immediately so */
   /* so set flag PSTAT.PSRDY.  Also clear PSTAT.PSNCB ("needs */
   /* CB") and PSTAT.PSETR ("don't enter") to be set as needed */
   /* later.                                                   */
   /* Because these status bits are protected by the PTBL      */
   /* transition lock we must get that lock first and clear it */
   /* when we are done.                                        */
   {
   XLOCK( ptbl->PSTAT.PTRAN ); /* returns with interrupts off */
   ptbl->PSTAT.PSETR = 0;
   ptbl->PSTAT.PSNCB = 0;
   ptbl->PSTAT.PSRDY = 1;
   ptbl->PSTAT.PTRAN = 0; /* unlocks transition state */
   interrupts_on;
   }
```

```
   goto SCHED:NODCL(,syscall_nbr,,ptbl);


   /* SCHED:NOREQ ------------------------------------------------ */
   /* There are either no system calls waiting or they cannot be   */
   /* started.  User code finally gets a chance to run if flag    */
   /* PSTAT:PSETR ("don't enter user") is clear.                   */

NOREQ:
   if (ptbl->PSTAT.PSETR)
      goto SCHED:RUNEXP(,,,ptbl); /* skip this PTBL */

   /* SCHED:REQ -------------------------------------------------- */
   /* All is clear to run a user task, if one is there.  Map in  */
   /* the user context, and check to see if the user task world  */
   /* should be rescheduled by PSCHD (if PTBL.PINSU = 0) or if   */
   /* it must be restarted where it left off.                    */

REQ:
   MAPCON( ptbl );
   if (ptbl->PINSU == 0)
      goto SCHED:PSCHD(,,,); /* assumes PTBL address in CC.W */

   /* SCHED:NORQ1 ------------------------------------------------ */
   /* The process lost control to the interrupt world.  We must   */
   /* restart the task where if left off.  Clear the interrupted  */
   /* flag, bump the global count of how many times this occurred */
   /* and re-establish the global task pointer.                   */

   ptbl->PFLG3.PFIRS = 0;
   NNORQ1.W++;
   CTSK.W = ring3.UST.USTCT;

   /* Restore the task's machine state before returning.  Routine */
   /* RSTPRC will also handle PIT timing and rescheduling of JPs. */
   /* It will only return here if the process can indeed continue.*/

   RSTPRC(tcb);

   /* We can return to the task two different ways depending on   */
   /* how it was interrupted.  If a page fault occurred then bit  */
   /* ?TSTAT.?TSWP in the TCB will be set and a fault block will  */
   /* exist for returning to the process.  Otherwise, we must copy*/
   /* the stack return block from the extender.  Because one      */
   /* instruction following an INTEN is executed with interrupts  */
   /* off, the WDPOP or WPOPB will return all the way out to the  */
   /* task code before another interrupt will be recognized.      */
```

```
   if (CTSK.W->?TSTAT.?TSWP)
      {
      interrupts_on;
      WDPOP();       /* exits to process >>>>>>>>>>>>>>>>>>>>>>>> */
      }
   else
      /* Build a context block on the stack so we can return to   */
      /* the task via a WPOPB.  The data in the PTBL extender     */
      /* from PMARG.W to PMPC.W is in the same format as the      */
      /* context block.
      /* Pseudocode variable "stack_pointer" refers to the stack  */
      /* pointer in the MV/Eclipse hardware.                      */
      /* (Note: RSTPRC returns the PTBL extender address in AC2.) */
      {
      WBLM( &ptbl->PEXTN.W->PMARG.W, stack_pointer, -OARG.W);
      stack_pointer = stack_pointer +(-OARG.W) -2;
      interrupts_on;
      WPOPB();       /* good-bye scheduler >>>>>>>>>>>>>>>>>>>>> */
      }

/* ========================================================== */
/* SCHED:NODCL                                                */
/* ---------------------------------------------------------- */
/* Setup and try to run a system call on a CB.  Called by SCPRC */
/* to process a user system call and from SCHED when starting   */
/* a system call that was queued up on a PTBL.                */
/* ---------------------------------------------------------- */

   /* pseudocode declarations */
   long int syscall_nbr;           /* system call number */
   PTBL    *ptbl;                  /* address of PTBL */
   entry    bag_invalid_syscall(); /* quick bad syscall handler */

   /* GLOBALS assumed */

   entry    (*MCCT.W)[];       /* table of system call handlers */
   long int (*SCTBL.W)[];      /* table of system call counters */

NODCL(,syscall_nbr,,ptbl):

   /* The system call number must at this point be in the range */
   /* of legal system calls.  Here we check that the system     */
   /* is actually implemented.                                  */

   PTMP3.W = MCCT.W[ syscall_nbr ];
   if (PTMP3.W == -1)
```

```
   /* The call has no handler and is thus not implemented.   */
   /* Prepare the TCB and PTBL to take the "illegal system   */
   /* command" error return and rerun the PTBL from the top. */
   {
   bag_invalid_syscall( ptbl, tcb);           /* SCHED:CERM2 */
   goto RERUN(,,,);                           /* SCHED:CERM3 */
   }

/* General accounting. Bump the count of the times this path */
/* was taken. If system call counting is enabled then count  */
/* the times each specific system call was made.             */

NNODCL ++;
if (SCTBL.W != 0)
   (*SCTBL.W)[ syscall_nbr ] ++;


/* SCHED:STKCK ------------------------------------------- */
/* This is where daemon processing (from SCHED:PDMPR) joins  */
/* system call processing.  They are similar in that they    */
/* both must prepare a CB for running a code path that was   */
/* initiated while running on a PTBL.                        */


/* GLOBALS assumed */
PTBL      *CC.W;     /* address of our PTBL */
CB        *PCB.W;    /* address of CB for call or daemon */
long int   PTMP1.W;  /* system call number (0 if daemon ) */
TCB       *PTMP2.W;  /* TCB address for call (0 if daemon ) */
void      *PTMP3.W();/* address of syscall or daemon handler */
LPCB      *MYLPCB.W; /* addr of current Logical proc ctl blk */
PPCB      *MYPPCB.W; /* addr of current Physical proc ctl blk */
bit      (*CHTBL.W)[]; /* map of calls that can run on child CPU */
void       FIXCB();  /* puts CB onto ELQUE & set proper status */
void       RESCH();  /* SCHED:RESCH to rescan ELQUE */
void       TDSCL();  /* Initiate system call timing on CB */
void       MEVENT(); /* Let mother know she has work to do */
void       RUNEXP(); /* skip to next item on ELQUE */
void       RERUN();  /* start PTBL check from the top */
short int  RSTCKT;   /* size of allocation of resident CBs */
short int  SSTCKT;   /* size of allocation of swappable CBs */
short int  G1;       /* lowest Group 1 process priority */
long int   NMONLY;   /* count of times CB moved to mother */

/* pseudocode declarations */

CB        *cb;       /* address of CB being worked on */
PTBL      *ptbl;     /* address of our PTBL */

Hardware_FP hw_frame_pointer /* hardware registers */
Hardware_SP hw_stack_pointer /* hardware registers */
```

```
SCHED:STKCK(,,,ptbl):

   /* The pseudocode below will use pseudo-variable "cb" to     */
   /* address the CB being manipulated.  The source uses PCB.W  */
   /* as the source of the address in most cases but sometimes  */
   /* depends on the CB address returning from subroutines.     */

   cb = PCB.W;

   /* Bump the count of active system paths for the process.    */

   ptbl->PEXTN.W->PSQCT ++;

   /* If this is a daemon then signal "scheduler action" on the */
   /* PTBL so it won't be run again until the daemon is done.   */

   if (PTMP2.W == 0)       /* TCB address test, 0 = daemon */
      ptbl->PSTAT.PSEW = 1;


   /* We only get here if we are definitely going to use the CB */
   /* pointed to by PCB.W.  We will either start a system call  */
   /* or daemon with the CB as the current context or put the   */
   /* CB onto the ELQUE for later scheduling.                   */
   /* Charge the CB against the appropriate pool (G1 vs G2/3)   */
   /* and determine what its PNQF will be.  We can check the    */
   /* MYPPCB flag because it was set by ALSTK when it allocated */
   /* a backup CB appropriate to the process in SCB.W.          */

   if (MYPPCB.W->CPSTAT.LSTACK == 1)

      /* Group 1 process (G1) CBs come from the resident stack  */
      /* (pool) count and run with a PNQF of 0.                 */
      {
      RSTKCT --;
      cb->PNQF = 0;
      }
   else
      /* Group 2 & 3 process CBs come from the swappable stack  */
      /* allocation and run with a PNQF just lower than group 1 */
      /* (G1) processes.                                        */
      {
      SSTKCT --;
      cb->PNQF = G1+1;
      }
```

```
/* Set up the CB based on the PTBL and the specific call.    */

cb->CERWD    = 0;                  /* no errors yet */
cb->CERPC.W  = 0;                  /* so no error PC either */
cb->PTIM.W   = 0;                  /* no time used yet */
cb->CBFEH.W  = -1;                 /* no fatal error handler */
cb->CATCB.W  = PTMP2.W;            /* associated TCB if any */
cb->CPTAD.W  = CC.W;               /* PTBL address */
cb->PGNUM.W  = CC.W->PGNUM.W;      /* same class nbr as PTBL */
cb->PLPCB    = MYLPCB.W;           /* curr logical processor */

cb->PSTAT    = 0;                  /* no status bits yet */
cb->PSTAT.PSRUN = 1;               /* CB will be running */

/* Set CB data dependent on call's "mother only" requirement */

if (CHTB[ PTMP1.W ])  /* CHild executable TaBle */
   {
   /* Call or daemon can only run on mother processor */
   cb->PCLASS = 0;                 /* no class bit if "MO" */
   cb->PSTAT.PMAST = 1;            /* indicate "mother-only" */
   }
else
  /* Call can run on any processor */
  cb->PCLASS = ptbl->PCLASS;     /* class bit same as PTBL */

/* Reset the stack.  At this point we abandon the past history */
/* of the PTBL and must go forward.  Pseudocode variables      */
/* "hw_stack_pointer" and "hw_frame_pointer" below are the     */
/* hardware stack registers.  Since we are still using the same*/
/* actual stack area we don't need to change the stack base or */
/* the stack limit.                                            */

interrupts_off;
cb->CSTK.W = cb->CSTKC.W;          /* reset FP within stack */
hw_frame_pointer = cb->CSTKC.W;    /* and HW frame pointer */
hw_stack_pointer = cb->CSTKC.W;    /* and HW stack pointer */
interrupts_on;

/* What we do now depends on whether our JP (MYPPCB) is the    */
/* mother processor and whether the CB path is a "mother-only" */
/* call or a daemon.  If path cannot run on our JP then go to   */
/* MONLY1 to move the CB onto the ELQUE for the mother JP to   */
/* schedule later.                                             */

if ( (MYPPCB->CPSTAT.CPMAST == 0)     /* MYPPCB is not mother */
   && ( (PTMP2.W == 0)                /* CB is for daemon */
      ||(CHTB[ syscall_nbr ] == 0))) /* call is mother-only */
   goto MONLY1;
```

```
/* We could possibly run on our JP.  But first to see if our   */
/* JP has been told to reschedule.  This is the last thing     */
/* that might prevent the CB from running the call now.        */

if (MYPPCB->CPSTAT.PRESCH == 1)

   /* SCHED:MONLY ------------------------------------------- */
   /* Our JP has been told to reschedule.  Before we can, we   */
   /* must put the CB onto the ELQUE and finish setting it up. */
   /* Until now the CB hasn't been on any queue in hopes of    */
   /* avoiding the overhead.  Since we set the "CB running"    */
   /* status bit above, the CB won't be selected to run by any */
   /* other JP after we put it onto the ELQUE but before we    */
   /* are done with it (and turn off PSTAT.PSRUN).             */
   /* We also must give up the PTBL for the reschedule.        */
   {
   FIXCB( cb );            /* put CB on ELQUE */
   SETUP();                /* swap to new PCB.W */
   cb->PSTAT.PSRUN = 0;    /* the CB is fair game now */
   ptbl->PSTAT.PSRUN = 0; /* so is the PTBL */
   RELPTBL( ptbl );        /* release others waiting for PTBL */
   goto SCHED:RESCH(,,,);
   }

/* We are going to start the call/daemon.  To switch context   */
/* we must change the JP's global TCB context block FBK.W and  */
/* global CB/PTBL context pointer CC.W.  We must also mark     */
/* the PTBL as no longer running.                              */

interrupts_off;
FBK.W = cb->CSTKC.W->CBCX.W;
ptbl->PSTAT.PSRUN = 0;
CC.W = cb;

/* Initiate system call timing on the CB. */

TDSCL( cb );
interrupts_on;

/* Finally, count the number of calls started on CBs and jump */
/* to the code for the call or daemon.                        */

TINDST.W ++;
goto (PTMP3.W); /* vamoose >>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>> */


/* SCHED:MONLY1 ------------------------------------------- */
/* The system call or daemon must run on the mother processor */
/* but our JP (MYPPCB) is not the mother.  Put the CB onto    */
/* the ELQUE for later scheduling by the mother processor and */
/* complete its initialization.  The CB won't be available to */
/* the mother until we reset its running bit (PSTAT.PSRUN).   */
```

```
   FIXCB( cb );          /* moves CB to ELQUE w/proper status */
   SETUP();              /* setup CB for TACT & fix PCB.W,SCB.W */
   cb->PSTAT.PSRUN = 0; /* the CB is fair game now */

   /* Tickle the mother processor so that she will reschedule    */
   /* (if needed) and run the mother-only CB we just readied to  */
   /* run.  Also count the number of times this path was taken.  */

   MEVENT( cb );
   NMONLY ++;

   /* Next see if it's useful to try to do anything else for the */
   /* PTBL.  If the mother processor actually woke up or if this */
   /* PTBL is no longer able to run, then let RUNEXP skip to the */
   /* next ELQUE element.  Otherwise rerun the PTBL from the top */
   /* to look for other actions we could take on it.             */

   if ((MPPCB.W->CPSTAT.PRSCH == 1) || (ptbl->PSTAT.PSRDY == 1))
      goto RUNEXP(,,,ptbl);
   else
      goto RERUN(,,,);


/* ========================================================== */
/* SCHED:SETUP                                                */
/* ---------------------------------------------------------- */
/* Used in system call CB startup paths to set up the stack of */
/* the primary CB (*PCB.W) so TACT will properly dispatch the CB */
/* from the ELQUE.  SETUP also assigns a new primary CB (PCB.W). */
/* ---------------------------------------------------------- */

   /* pseudocode declarations */
   CB_or_PTBL   *cb;
   Hardware_FP   hw_frame_pointer;   /* hardware registers */
   Hardware_SP   hw_stack_pointer;
   Hardware_SL   hw_stack_limit  ;
   Hardware_SB   hw_stack_base   ;
   Instruction   wssvr() = {WSSVR};  /* MV instruction */

   /* GLOBALS assumed */
   CB_or_PTBL *PCB.W;          /* addr of CB to be set up */
   CB_or_PTBL *SCB.W;          /* candidate for next primary CB */
   CB_or_PTBL *TMPCB.W;        /* candidate for next primary CB */
   sys_code() *PTMP3.W;        /* address of system path for CB */
   external TRTN();            /* error return for system call */
```

```
SETUP(,,,):
   /* ------------------------------------------------------------ */
   /* Set up the CB stack contents for dispatch by TACT.  TACT     */
   /* expects the stack to be set up so a WRTN will go to the      */
   /* code with the proper ACs and stack/frame pointer.  The       */
   /* system call code will need the CB address in AC2 and the     */
   /* WRTN will expect the code address in AC3.  The sys call      */
   /* path will also expect to have its return address (OFP)       */
   /* to be the system call error return handler TRTN.             */

   cb = PCB.W;
   wssvr(,,cb,PTMP3.W);
   cb->CSTK.W = hw_frame_pointer;
   cb->OFP.W  = &TRTN();

   /* Now assign a new primary CB (PCB.W).  If there is a temp  */
   /* CB in TMPCB.W then use it, otherwise use the secondary CB */
   /* from SCB.W.  One or the other was allocated by ALSTK.     */

   if (TMPCB.W != 0)
      {
      PCB.W = TMPCB.W;
      TMPCB.W = 0;
      }
   else
      {
      PCB.W = SCB.W;
      SCB.W = 0;
      };

   /* Finally switch hardware stack to new primary CB. The */
   /* source calls routine STKST1 for this but it is shown */
   /* here in-line for clarity.                            */

   hw_stack_limit   = PCB.W->CBSL.W;
   hw_stack_base    = PCB.W->CSTKC.W;
   hw_stack_pointer = PCB.W->CSTKC.W;
   hw_frame_pointer = PCB.W->CSTKC.W;

   return(cb,,,);

/* ============================================================ */
/* SCHED:CERM2,3                                                */
/* ------------------------------------------------------------ */
/* This code is referred to as "bag_invalid_syscall" in the call*/
/* processing code of SCHED.  It cuts short the standard system */
/* call path by running on the PTBL while updating the user TCB */
/* to take the error return with error ERICM "illegal system    */
/* command."  This avoids the overhead of switching to a CB to  */
/* run something.  In the source the TCB address is obtained    */
/* from either the input AC2 or from PTMP2.W.  But for clarity, */
/* this code shows both the TCB and PTBL address coming from    */
/* the caller.                                                  */
```

```
bag_invalid_syscall( tcb, ptbl ):

   TCB     *tcb;      /* TCB address of task making call */
   PTBL    *ptbl;     /* PTBL address of process making call */

   /* Set up the TCB to take the error return with error ERICM  */
   /* in AC0.  On entry to the system the task's return address */
   /* (TPC.W) was set for a normal return so we must decrement  */
   /* it for the error return.                                  */

   tcb->TPC.W --;
   tcb->TAC0.W = ERICM;     /* "illegal system call" */

   /* Set up the PTBL to indicate that it has something to do   */
   /* by resetting flags PSENT ("don't enter user", BPSEN) and  */
   /* PSNCB ("needs CB", BPSNC) and PSTAT.PSRDY ("not ready to  */
   /* run", BPSRY) in PSTAT.                                    */

   ptbl->PSTAT.PSENT = 0;    /* can now enter user code */
   ptbl->PSTAT.PSNCB = 0;    /* no longer has to have CB to run */
   ptbl->PSTAT.PSRDY = 0;    /* now is ready to run */

   /* Return to pseudocode.  The source actually jumps to RERUN */
   /* but for clarity the pseudocode is written so that this    */
   /* routine returns and the caller does a "goto RERUN".       */

   return;


/* ========================================================== */
/* SCHED:PSCHD                                                */
/* ---------------------------------------------------------- */
/* AOS/VS task scheduler.                                     */

/* GLOBALS */

   void       XLOCK();     /* master lock, interrupts off at return */
   void       RELPTBL();   /* release PTBL and others waiting on it */
   void       RSTPRC();    /* restore FPU, PTBL state & timing */
   Boolean    CTBLK();     /* move PTBL to blocked queue */

   label      RUNPTP();    /* release then skip PTBL on ELQUE */
   label      RUNEXP();    /* skip over this PTBL on ELQUE */
   label      TCBAD();     /* aborts process for bad TCB chain */
   label      SMON0();     /* SCHED:RESCH rescan ELQUE */

   UST        ?ARING.UST;  /* User status table in agent ring */
   PTBL      *CC.W;        /* address of our PTBL */
   TCB       *CTSK.W;      /* address of current TCB in process */
   short int  G1;          /* lowest priority for group 1 processes */
   long int   NPSLP;       /* count: times TCB chain is scanned */
   long int   NPSCDR.W;    /* count: task rescheduling found disabled */
```

```
/* pseudocode declarations */

   PTBL          *ptbl;       /* address of our PTBL */
   PTBL_EXTENDER *pextn;      /* address of our PTBL's extender */
   TCB           *tcb;        /* address of TCB selected to run */
   TCB           *other_tcb;  /* address of some other TCB */
   TCB.TSTAT      CRUMS;      /* mask of TSTAT pended flags */

   /* These next two pseudo-routines represent the NFSAC and NBSE  */
   /* queue search instructions used to scan the TCB chain.  They  */
   /* take as arguments the accumulators and search mask needed.   */
   /* They return OK if a matching TCB is found and NOT_OK if the  */
   /* search failed.  Since the source simply resumes interrupted  */
   /* searches, the pseudocode also ignores interrupts.           */

   Boolean   tcb_fscan(); /* Fwd search of TCB chain (SCHED:PSLP) */
   Boolean   tcb_bscan(); /* Bkwd search of TCB chain (SCHED:PRLP) */

   Hardware_FP hw_frame_pointer;    /* hardware registers */
   Hardware_SP hw_stack_pointer;
   Hardware_SB hw_stack_base;

   Instruction wpsh() = {WPSH 0,2}; /* MV instrs in pseudocode */
   Instruction wblm() = {WBLM};
   Instruction wpopb()= {WPOPB};
   Instruction wdpop()= {WDPOP};
   Instruction deque()= {DEQUE};
   Instruction enqt() = {ENQT};
   Instruction interrupts_off() = {INTDS};
   Instruction interrupts_on()  = {INTEN};


PSCHD(,,,):
   /* CRUMS is a mask of the bits in TCB.TSTAT that indicate    */
   /* that the TCB is pended on something.  It is used for the  */
   /* queue search instruction scan of the TCB chain and to     */
   /* quickly isolate the flags that would prevent a TCB from   */
   /* being run.  The source builds the mask at assembly time   */
   /* but it is pseudocoded here to clearly show which bits are */
   /* included in the mask.                                     */


   CRUMS.?TSPN = 1;  /* general purpose pended flag */
   CRUMS.?TSSG = 1;  /* waiting for ?XMTW or ?REC */
   CRUMS.?TSSP = 1;  /* suspended */
   CRUMS.?TSRC = 1;  /* waiting for TRCON */
   CRUMS.?TSOV = 1;  /* waiting for overlay */
   CRUMS.?TSGS = 1;  /* pended for agent synchronization */
   CRUMS.?TSAB = 1;  /* awaiting ?GABORT */
```

```
CRUMS.?TSTL = 1;  /* awaiting ?TUNLOCK by another task */
CRUMS.?TSDR = 1;  /* pended by ?DRSCH */
CRUMS.?TSLK = 1;  /* pended on an ?FLOCK */
CRUMS.?TSXR = 1;  /* pended on XMT or REC */
CRUMS.?TWSG = 1;  /* pended on ?WTSIGNAL */
/* bits NOT set in the CRUMS mask */
CRUMS.?TSUT = 0;  /* expecting return from ?UTSK */
CRUMS.?TSUK = 0;  /* expecting return from ?UKIL */
CRUMS.?TSYG = 0;  /* task has beeb ?SIGNALed */
CRUMS.?TSWP = 0;  /* task is faulting */

/* Pseudocode will use these variables to address the PTBL */
/* and its extender but the source will often use global   */
/* CC.W to find them.                                      */

ptbl = CC.W;              /* PTBL address */
pextn = ptbl->PEXTN.W;    /* PTBL extender address */


/* REAL start of PSCHD ------------------------------------ */
/* Clear the process' reschedule flag.                      */

ptbl->PFLG.PFRSH = 0;

/* First check to see if rescheduling is off or disabled.    */
/* If the user disabled it then USTFL.?UFDR will be set or   */
/* if the process is just getting started and scheduling is  */
/* not yet in effect then USTFL.?UPPH will be set.  If the   */
/* AGENT doesn't want rescheduling then it sets word ?TSMA   */
/* non-zero.                                                 */

if ( (?ARING.UST.USTFL.?UFDR = 1)
    ||(?ARING.UST.USTFL.?UFPH = 1)
    ||(?ARING.?TSMA != 0) )

   /* SCHED:PSCDR ------------------------------------------- */
   /* Rescheduling is off or disabled, check to restart the    */
   /* current task.  We should if all its pend flags are clear */
   /* or if it's continuing after a fault.                     */
   /* If we cannot rerun the current TCB then let PNTRY figure  */
   /* out how to relinquish control or if we should try again. */
   {
   tcb = ?ARING.UST.USTCT;
   NPSCDR.W ++;
```

```
   if (  /* task didn't just complete a page fault */
        !( (tcb->TSTAT.?TSWP == 1)  /* faulting? */
          &&(tcb->TSTAT.?TSPN == 0)) /* not pended for fault? */
       && /* and task has any pend flags set */
         ((tcb->TSTAT & CRUMS) != 0) )
      /* then cannot restart current task */
      goto PNTRY;
   else; /* Restart current task (fall through to PTFN1). */
   }
else
   /* Rescheduling is enabled, count the times this was true. */
   {
   NPSLP.W ++;

   /* Search forward through the TCB chain for the first TCB  */
   /* having no pended bits set in the TSTAT word.  If no TCB */
   /* is ready to run then let PNTRY decide whether to retry  */
   /* or give up control of the JP.  If a ready TCB is found  */
   /* its address will be returned in TCB (through AC1).      */

   tcb = ?ARING.UST.USTAC;
   if (NOT_OK == tcb_fscan( (&?ARING.UST.USTAC), /* ac0 */
             tcb,,(TSTAT), (CRUMS) ))  /* ac1,2,3 & mask */
      /* no ready TCB found. */
      goto PNTRY;

   /* We have found the next TCB to run.  Make it the new */
   /* active TCB.                                         */

   ?ARING.UST.USTCT = tcb;

   /* To implement round-robin scheduling we must make sure */
   /* that the TCB just selected is in the TCB chain behind */
   /* all other TCBs with the same priority.  First see if  */
   /* that is already true.  The TCB is okay if it is at the*/
   /* end of the chain (TLNK = -1), or if the next TCB in   */
   /* the chain has a different priority.                   */

   other_tcb = tcb->TLNK;
   if ((other_tcb != -1) && (other_tcb->TPRI != tcb->TPRI))

      /* Looks like the TCB needs to be moved behind other   */
      /* TCBs with the same priority.  Use a backward queue  */
      /* search through the TCB chain to find the last TCB   */
      /* with the same priority as the one we selected to    */
      /* run.  USTAQD2 in the UST points to the last TCB in  */
      /* the active chain.                                   */
      {
      other_tcb = ?ARING.UST.USTAQD2;
      if (NOT_OK == tcb_bscan( (&?ARING.UST.USTAC),
                 other_tcb,,(TPR),(tcb->TPR) ))
```

```
         /* The backward scan didn't find a match.  This could */
         /* only happen if the TCB chain is corrupted.  Abort  */
         /* the PTBL before anything gets worse.               */

         goto TCBAD(,,,ptbl);

      /* Now we must take the TCB being started out of the  */
      /* chain and put it back in following the one we just */
      /* found.                                             */

      if (tcb != other_tcb ) /* how can this happen? */
         {
         deque( (&?ARING.UST.USTAC), tcb,,);
         enqt ( (&?ARING.UST.USTAC), other_tcb, tcb,);
         }
      } /* end of round-robin path */
   } /* end of reschedule path */


/* SCHED:PTFN1 ------------------------------------------------ */
/* A task has been selected to run.  It was selected either by */
/* the reschedule algorithm or because it was the previously   */
/* running task and must be restarted.  Set the system global  */
/* task pointer and reset the system stack to prepare to start */
/* a new task.                                                 */

CTSK.W = tcb;
hw_stack_pointer = hw_stack_base;
hw_frame_pointer = hw_stack_base;

/* Restore the user stack state from the TCB.  The action to   */
/* take depends on whether this task was the last task to run. */
/* The pseudocode checks the prior TCB address (other_tcb) in  */
/* a different order than the source to simplify the nesting   */
/* of the if statements.                                       */

other_tcb = pextn->PCTSK.W;

if (other_tcb != tcb)

   /* This task isn't the previous task so we must update */
   /* the task and stack states of the process.  Save the */
   /* state of the previous task if there was one.        */
   {
   if (other_tcb != 0)
      TSKSAV(,other_tcb,,);
```

```
   /* Now set up the stacks for the new task to run and */
   /* update the PTBL entry pointing to the last task.  */

   TSKRST(,,tcb,);
   pextn->PCTSK.W = tcb;
   }


/* SCHED:SAMTSK */
/* Now handle the extended variable save area(s) if the  */
/* task selected to run is not the same as the last task */
/* run that used the extended variable save area.        */

other_tcb = pextn->PCEXV.W;

if (tcb != other_tcb)

   /* This task wasn't the last one run with an extended  */
   /* variable area.  If another task with such an area   */
   /* has run then we must save its variable area.        */
   {
   if (other_tcb != 0)
      {
      wblm(,(?ARING.UST.USTEZ),    /* nbr of words to save */
            (?ARING.UST.USTES),    /* addr of user variable area */
            (other_tcb->TELN.W)); /* addr of save area for task */

      pextn->PCEXV = 0;    /* save is complete */
      }

   /* Now we see if this task has an extended variable    */
   /* area defined.  If so then we need to put its saved  */
   /* data into the process' variable area.               */

   if (tcb->TELN.W != 0)
      {
      wblm(,(?ARING.UST.USTEZ),  /* size of variable area in wds */
            (tcb->TELN.W),       /* where task's vars were saved */
            (?ARING.UST.USTES)); /* process variable area addr */

      pextn->PCEXV.W = tcb; /* this task vars are active */
      }
   } /* end of extended task variable handling */


/* SCHED:NOEXV */
/* The final task-specific state to restore is the FPU state.  */
/* RSTPRC will restore it and then make last minute checks of  */
/* the process' timeslice and the JP's reschedule flag. It     */
/* may abandon the process startup and go to RESCH.            */
```

```
   RSTPRC(,,tcb,);

   /* We are committed now to running the selected task in the    */
   /* selected process.  Interrupts were turned off by RSTPRC so  */
   /* that we can get all the way into the user before interrupts */
   /* can occur.  Finally determine whether to return to the      */
   /* user via a WPOPB after a fault or to use a WDPOP.           */

   if (tcb->TSTAT.TSWP == 1)


      /* Task had faulted so we return to the user through the    */
      /* fault block on his stack.  Clear the faulting bit before */
      /* going back since we just finished handling it.           */
      {
      tcb->TSTAT.TSWP = 0;
      interrupts_on();
      wpopb();  /* enter user world >>>>>>>>>>>>>>>>>>>>>>>>>>>>> */
      }
   else
      /* Task was stopped for something other than a fault.  We   */
      /* must build an LCALL return block on the stack to get     */
      /* back to him.
      {
      wpsh( tcb->TOVF.W,                /* PSR and arg count */
            tcb->TAC0.W, tcb->TAC1.W); /* ac's 0 and 1 */
      wpsh( tcb->TAC2.W, tcb->TAC3.W,  /* ac's 2 and 3 */
            tcb->TPC.W);               /* carry & PC */

      /* Clear the "processes stopped by interrupt" flag since it */
      /* no longer matters how it stopped.                        */

      ptbl->PFLG3.PSIRS = 0;

      /* Transfer control to the user process.  This location is */
      /* given the entry point UTRAP.1 so that it can be easily  */
      /* found by the debugger.                                  */

      interrupts_on();
UTRAP.1:
      wpopb();   /* jumps to the user >>>>>>>>>>>>>>>>>>>>>>>> */


   /* SCHED:PNTRY ------------------------------------------- */
   /* PSCHD found no active TCBs ready to run.                */
   /* However it is possible that an interrupt during the TCB */
   /* scan has readied a TCB.  If so then PTBL.PFLG3.PFIRS will */
   /* have been set and we should try PSCHD again.            */
```

```
PNTRY:
   XLOCK(ptbl->PSTAT.PTRAN);  /* lock PTBL, turn intrpts off */

   if (ptbl->PFLG3.PFIRS == 1)

      /* The interrupt world did interrupt us and could have */
      /* made a task ready.  Clear the flag, unlock the PTBL */
      /* and start PSCHD over again to check it out.         */
      {
      ptbl->PFLG3.PFIRS = 0;
      ptbl->PSTAT.PTRAN = 0;  /* releases PTBL */
      interrupts_on;
      goto PSCHD;
      }


   /* SCHED:SYST1 - User tasks are definitely not ready to run.  */
   /* Check if there is any system activity or if the process    */
   /* is totally idle and should be blocked.  System activity is */
   /* determined by checking the list of TCBs waiting to start   */
   /* syscalls (PSWD) and the total of the number of outstanding */
   /* MTA/LPB requests (PSIOC) and the number of active system   */
   /* calls the process has running in the system (PSQCT).       */

   if ((pextn->PSWD != 0) || ((pextn->PSIOC + pextn->PSQCT) != 0)))

      /* Process cannot be blocked but we still want to minimize   */
      /* the time spent on the PTBL later.                         */
      /* Depending on whether or not the PTBL has system calls     */
      /* waiting to start we will set either flag PSTAT.PSETR      */
      /* ("don't enter user") or PSTAT.PSRDY ("not ready to run"). */
      {
      if (pextn->PSWD == 0)
         ptbl->PSTAT.PSRDY = 1;
      else
         ptbl->PSTAT.PSETR = 1;

      /* See if we can also set the "needs CB" flag.  This flag is */
      /* legal for group 2 and 3 PTBLs with indirect system calls  */
      /* waiting to start.  (Who knows why we would actually want  */
      /* to set it in this case, but that's VS for you.)           */

      if (  (ptbl->PPRI > G1) && (ptbl->PSIDIR != 0))
         ptbl->PSTAT.PSNCB = 1;

      /* Done with the PTBL although it is still on the ELQUE. */

      goto RUNPTP(,BPTRAN,,ptbl);
      }
```

```
else
   /* SCHED:SYST3 - Process really is idle so set PSTAT.PSRDY */
   /* ("not ready to run").                                   */
   {
   ptbl->PSTAT.PSRDY = 1;

   /* See if we should or can block the process, PFLG2.PFBLE   */
   /* is on if the process can be blocked.  Swappable and      */
   /* preemptible processes can but resident processes cannot.  */
   /* If the process cannot be blocked, just skip over its PTBL */
   /* in the ELQUE.                                           */

   if (ptbl->PFLG2.PFBLE == 0)
      goto RUNPTP(,BPTRAN,,ptbl);

   /* Set up the flags that indicate the process is blocked. */

   ptbl->PSTAT.PSRUN = 0;        /* not running */
   ptbl->PSTAT.PSBLK = 1;        /* is blocked */
   ptbl->PSTAT.PSNCB = 0;        /* not waiting for G2/3 CB */

   /* Try to move it to the blocked queue. If successful then  */
   /* release the PTBL and anyone waiting on it and reschedule */
   /* from the start of the ELQUE.  Otherwise just skip over   */
   /* the PTBL in the ELQUE.                                   */

   if (OK == CTBLK(,,ptbl,))
      {
      RELPTP(,,ptbl,);
      goto SCHED:SMON0(,,,);  /* (same as SCHED:RESCH) */
      }
   else
      goto RUNEXP(,,ptbl,);
   }
```

Chapter 4
Logical Processor Management

4.1 Introduction

4.1.1 Purpose

The purpose of this chapter is to describe the Logical Processor (LP), how it is managed, and how it relates to the other parts of Paths and Time.

4.1.2 Overview

This chapter will describe the internals of LP management. An LP is an abstract entity or resource used in class scheduling and CPU accounting. An LP represents the aggregate CPU power of 0 or more JPs. An LP is the way the user perceives the CPU. The state of the LP is kept in the LPCB.

An LP can be compared with a LDU in the file system. Where an LDU consists of one or more physical units. The LDU is an environment where a file structure is self-contained. The LP is a self-contained environment of classes of processes. The LP is more flexible than an LDU, because a user can dynamically add phyiscal units to an LP where that is not possible with an LDU.

Figure 4.1 below shows how LP management relates to the other components of Paths and Time.

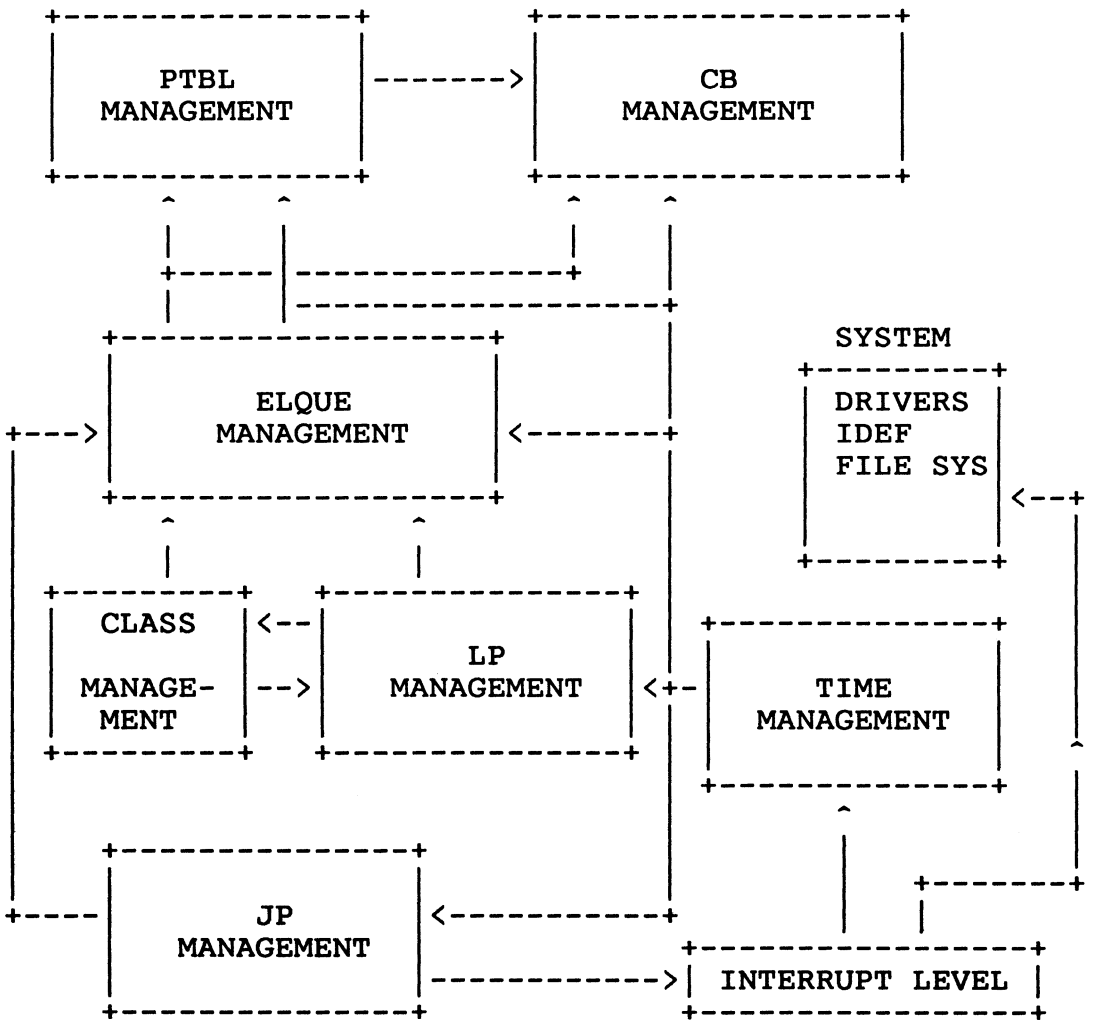


Figure 4.1

The LP provides and receives services from the other components of Paths and Time. The LP provides scan mask management for ELQUE management and provides statistics for class management. LP management gets time information from the time management component and gets class time definitions from class management. (See Figure 4.1.)

This chapter is divided into three major parts: the LPCB, user services, and system services.

The LPCB is the Logical Processor Control Block. It is the major database of LP management. The structure and fields will be defined. The basic operations and main paths that affect the LPCB will be shown in a who, how, and why fashion. This chapter will show reasons, as well as, who and how the LPCB is affected. Database locking to maintain the integrity of the LPCB also will be defined.

The user interfaces section describes the system calls that the user makes that affect the LP. For example, the ?LPCREA sets up the LPCB - this would be a user service.

The system services are services provided for the rest of paths and time. An example of a system service is providing a scan mask to ELQUE management to scan for a CB or PTBL to run.

4.2 Logical Processor Management Objects

Logical processor and Class management manage and manipulate ALL the objects with LP or CL at the beginning of the object name. This section is divided into two subsections: the LPCB and the LP globals.

4.2.1 The Logical Processor Control Block (LPCB)

The LPCB is used to keep track of the scheduler mode and keep class scheduling statistics. All LPCBs are created dynamically except the first LP(LP0). LP0 is created at system initialization. The LPCB is divided into five sub-databases: the general information section, the scan mode 0 section, the scan mode 1 section, the user summary section, and the user supplied class percentages.

The general information summary is as follows:

WORD OFFSET		USAGE SUMMARY
LPSTAT	0	Status word (see next page)
LPID	1	LPID for this LP
LPJPCNT	2	# of JPs attached to this LP
LPJPCNT.W	3	Bit map of attached JPs
LPMVCNT	5	"Move in progress" count
LPMODE	6	Scheduler Mode #
LPDUM	7	Even word align the following
Scan mode 0 information:		
LPCSM.W	10	Current Scan Mask
LPCTM.W	12	Current Time - Class #1 ** 16 classes (16 double words) **
LPCIU.W	52	Current Interval Usage
LPISM.W	54	Initial Scan Mask
LPITM.W	56	Initial Time - Class #1 ** 16 classes (16 double words) **
LPIIU.W	116	Initial Interval Usage
Scan Mode 1 information:		
LPTIM	120	Time Interval
LPTMK.W	121	Current Tier Mask Address
LPHMK.W	123	Hierarchical Mask - Class #1 ** 16 classes (16 double words) **

Figure 4.2

General Usage Information

WORD OFFSET		USAGE SUMMARY
LPSUM.4	163	Summary Class time Class #1 ** 16 classes (16 Quad words) **
LPTUSE.4	263	Total interval time usage
LPTPT.4	267	Total Run-Anywhere Ring 1-7 proc usage
LPTMPT.4	273	Total Mother-Only Ring 1-7 proc usage
LPTSYS.4	277	Total Non-Daemon 'R-A' Sys Call usage
LPTMSY.4	303	Total Non-Daemon 'M-O' Sys Call usage
LPTDAE.4	307	Total Daemon Sys Call usage
LPTINT.W	313	Total # Time Intervals
User-supplied class percentage values (for mode 0)		
LPPCT	315	User-specified percentage values ** 16 Classes (16 words) **
LPCBLN 335		LPCB length = 335 words.

LPSTAT BIT DEFINITIONS

BIT OFFSET	STATUS SUMMARY
LPLCK (0)	LPCB Lock bit
LPOFF (1)	Class Scheduling is off for this LPCB
LPACC (2)	This LP is not Accumulating stats
LPMOM (3)	The "Mother" JP is attached to this LP

Figure 4.2 (Continued)

4.2.1.1 LPCB Offset Explanations

General Information summary:

This section contains the information used to manage the LP, the JPs attached to this processor, and the mode of scheduling used on this LP.

LPSTAT - Contains the status word for this LP.

LPLCK - This is the lock bit for this LPCB. This is a spin lock, which means that the requestor must spin while waiting for the lock. LPLCK is used to keep other JPs from touching some of the LP databases while this JP is working on the LPCB. For example: JP 0 uses this lock while changing the scan mask in the LPCB. The lock guarantees that only one JP at a time can change the scan mask.

LPOFF - If this bit is set then this LP is not running class scheduling, which means that classes are not being enforced. This bit is set when class scheduling is disabled. If this bit is 0 then the LPACC must be 0.

LPACC - If this bit is set then this LP is not in Accumulate mode. If LPOFF is not set then this bit will also be cleared, because in order to enforce class scheduling accumulation of statistics must be done. This bit can be 0 with the LPOFF bit being set(= 1). This means that the LP is in ACCUMULATE mode. This bit will be checked in the scheduler to see if it is necessary to keep statistics. This bit is set or cleared by ?CLSCHD.

LPMOM - This bit will be set whenever the mother processor is attached to this LP. LPMOM is set during a ?JPMOV. This bit is checked when the user is getting charged for time. If this bit is clear, then a mother-only process could not have run on this LP. Therefore, no charge could be assessed to the mother-only counters in this LPCB.

LPID - This is the Logical Processor ID for this LP. It is assigned at SINIT time for LP 0 and at ?LPCREA time for the other LPs. The LPID is used by the system manager to identify the logical processor being worked with and by JP management to identify which LP a JP is attached to.

LPJPCNT - Is the number of JPs attached to this LP. This counter is affected when a JP is attached to or detached from the LP.

LPJPMP.W - This 32-bit integer is used as a bit map of JPs attached to this LP. If a bit in this field is set then the corresponding JP is attached to this LP. For example if bit 4 is set then JP four is attached to this LP. This field is modified when a JP is moved to or from this LP. See ?JPMOV.

LPMVCNT - This 16-bit integer is used to show how many JPs are currently being moved to this LP. This field is used to ensure that an LP cannot be deleted while a JPMOV is in progress. To delete this LP, LPMVCNT must be 0.

LPMODE - Is the scheduler mode for this LP. The mode of the LP dictates which part of the LPCB is used for scheduling. The normal scheduling mode is mode 0. This means that LP will run Primary classes defined by the user. If class scheduling is not being used, the scheduler will stay in mode 0. If the scheduler is in mode 1, it is is running secondary classes. This value changes from mode 0 to mode 1 when the scheduler is about to go idle but still may have secondary classes defined. If none are defined, the mode gets RESET and the LP (JP attached to it) will go idle.

LPDUM - This field even word aligns the rest of the LPCB.

Scan Mode 0 Information:

Scan Mode 0 is the mode where only the Primary Classes are run. This means that all secondary and stranded classes are masked out. The following section describes the offsets for Scan Mode 0:

LPCSM - This is the current mode 0 scan mask. This mask is used by the Scanner to find a process that can run. Anything that is not a primary class (if defined) or has used up its percentage will be masked out. If class scheduling is not used, the mask will not mask out any class.

LPCTM.W - These 16 double words hold the amount of time that has been used in this particular interval for each possible class. When a RESET is done during scheduling these fields are cleared and timing restarts. When one of these fields matches the contents of the (LPITM + class offset) then the scan mask is updated for that class masking it out.

LPCIU.W - This double word holds the amount of the interval that has been used. When the amount of LPCIU exceeds the amount in LPIIU, a RESET occurs.

LPISM.W - This contains the initial scan mask. This scan mask reflects what the user has defined as the primary classes. When a RESET occurs this value is put into LPCSM.

LPITM.W - These 16 double words contain the maximum number of ticks each class is allowed during a specific time interval. These values are calculated at ?LPCLASS time. The calculation is (LPITM.W+offset) = interval ticks*class percentage/100. If the number contained in (LPCTM.W + class offset) exceeds the corresponding value in LPITM.W then the current scan mask is updated to deny that class of process CPU time.

The following example shows an interval broken into ticks for a 10 second interval. The example then breaks up the interval into the number of ticks per interval that the corresponding class will get.

```
INTERVAL     =   10  SECONDS       *  USER VIEW    *
     TICKS   =  (SECONDS * 1000)   * MILLISECONDS *
INTERVAL     =  10000 PIT TICKS    *  SYSTEM VIEW *

 CLASS 0(A)  = 20 %                 *  USER VIEW    *
CLASS 0(A)   = 2000   PIT TICKS     *  SYSTEM VEIW *

 CLASS 1(B)  = 10 %                 *  USER VIEW    *
CLASS 1(B)   = 1000   PIT TICKS     *  SYSTEM VEIW *

 CLASS 2(C)  = 50 %                 *  USER VIEW    *
CLASS 2(C)   = 5000   PIT TICKS     *  SYSTEM VEIW *

 CLASS 3(D)  = 20 %                 *  USER VIEW    *
CLASS 3(D)   = 2000   PIT TICKS     *  SYSTEM VEIW *
```

LPIIU.W - This double word contains the number of ticks in the time interval for this LP. When this value is exceeded by the value contained in LPCIU, the RESET occurs. This number is set during the ?LPCLASS system call.

The following example shows an interval broken into ticks. The result of this calculation is put into LPIIU.W.

```
INTERVAL     =   10  SECONDS       *  USER VIEW    *
     TICKS   =  (SECONDS * 1000)  * MILLISECONDS   *
INTERVAL     =  10000 PIT TICKS    *  SYSTEM VIEW *
```

SCAN MODE 1 INFORMATION:

Scan Mode 1 is the mode where the scanner only schedules secondary classes. All primary and stranded classes are masked out. In this mode LP management supplies the scan mask for a tier of secondary class. A secondary class tier is a level of secondary classes. (See CLASP manual.) The following section describes the offsets of scan mode 1:

LPTIM - This word is supposed to contain the time interval but it is not used.

LPTMK.W - This double word contains the address of the current scan mask within secondary class level. The secondary class level is one of the 16 possible secondary class levels that can be defined in class scheduling. (See CLASP manual.) These 16 class levels are in LPHMK.W.

LPHMK.W - These 16 double words contain the scan masks for the secondary class hierarchy. When class scheduling is in mode 1 the LPTMK.W points to an offset in this table so the correct scan mask may be used by the scanner. This table is defined at ?LPCREA time. If the bit for a certain class is 0 in this mask, then that class is a valid secondary class for this tier. If the bit in LPHMK.W is set, then the class is not on this tier. If LPHMK.W is -1, then there are no secondary classes on this tier and no secondary classes in any more secondary class tiers.

The next group of LPCB offsets are used for gathering statistics. These counters are 64-bit integers. Because the offsets are 64 bits or four words long, they have a "4" as part of the offset name. These offsets are used to hold the total pit ticks a path uses when it runs on this LP. For example: after a path such as a ?LPCREA (which is a mother-only operation) runs on this LP, the time used (in pit ticks) is added to offset LPTMPT.4 in the LPCB. These offsets accumulate from the time the LP is created.

LPSUM.4 - This a group of 16 quadruple words used to hold cumulative data on the amount of time used by each class on this LP. These databases are updated at each subslice end and at process termination.

LPTUSE.4 - These four words are used to count the total interval time used. An interval is a system manager defined period of time from which the class scheduling percentages are converted into pit ticks. This counter is updated each time a user runs if class scheduling (LPACC is 0) is in Accumulate mode. This counter is zero when the LP is created.

LPTPT.4 - These four words are used to count the total number of nonsystem ticks. Nonsystem ticks are defined as non-ring 0 ticks. This includes time used by AGENT EXEC and PMGR. This counter holds the total for all JPs attached to this LP. This field is updated at each subslice end and at process termination.

LPTMPT.4 - These four words count the total "mother only" user time ticks. This field is updated at each subslice end and at termination.

LPTSYS.4 - These four words count the total number of ticks used by system calls that can run on any JP. This database is updated after a CB for this type of system call run.

LPTMSYS.4 - These four words contain the total number of ticks used by "mother only" system calls. This database is updated after a "mother only" system call runs. One type of system call that is considered "mother only" is ?OPEN, which uses the file system.

LPTDAE.4 - These four words contain the total number of ticks used by Daemons on the LP. After a CB runs, the LP accounting path checks if the CB that ran was a daemon. If so, this database gets updated.

LPTINT.W - This double word contains the total number of intervals used on this LP. An interval is a system-manager defined period of time from which the class scheduling percentages are converted into pit ticks. When an interval is used up, then the system clears the time used parts of class scheduling. This database is only updated when an interval ends. This is significant because this value only reflects the Total number of FULL intervals used, not RESETs of intervals based on ELQUE behavior. (See RESET.)

User Supplied Class Percentage Values:

The last part of the LPCB is the user-defined area. This area is affected by the ?LPCLASS system call.

LPPCT.W - These 16 words contain the user defined Class percentages. These percentages are defined at ?LPCLASS time. These values are used to set up LPITM.W.

LPCBLN - This value is the length of the LPCB, which is 335 words.

** LPBLM is not a global and is not an offset in the LPCB; it is the number of words that get BLMed (Block moved) into the LPCB at RESET time. The BLM will start from offset LPCSM.W and move LPBLM words of zeros into this area. The value of LPBLM is 44 (octal). This value is only useful when looking through the RESET code.

4.2.2 The Globals

In AOS/VS there are Global symbols that are needed so multiple areas of the system can get at data. These symbols are defined in the modules called SZERO.LS and STABLE.LS.

Table 4.1 shows all the globals associated with LP Management.

Table 4.1 The LP Management Globals

NAME	DESCRIPTION
LP.W	THE TABLE OF LPCBs
MYLPCB.W	POINTER TO THE CURRENT LPCB ONE FOR EACH JP
LPCNT	TOTAL NUMBER OF LPS ON THE SYSTEM
LPMAP.W	BITMAP OF USER-VISIBLE LPCBs
LPCBMAP.W	BITMAP OF ALL EXISTING LPCBs
LPCB.TMP	A PREDEFINED LPCB
JPLPLK.W	THE GLOBAL LP JP LOCK WORDS
MLPCB.W	POINTER TO MOTHER LPCB
MXLPCB	MAXIMUM NUMBER OF LPS ON THE SYSTEM
CLSCHD.W	BIT MAP OF LPs THAT HAVE CLASS SCHEDULING ENABLED
CLACC.W	BIT MAP OF LPs THAT HAVE CLASS ACCOUNTING ENABLED

4.2.2.1 Global Definitions

The following section provides more detail on the globals defined in Table 4.1.

LP.W is a table of double-word pointers to LPCBs. This table is indexed by LPID and is used by the function findlp(). The table is also touched by the user service ?LPCREA. Figure 4.3 below shows the LP.W table:

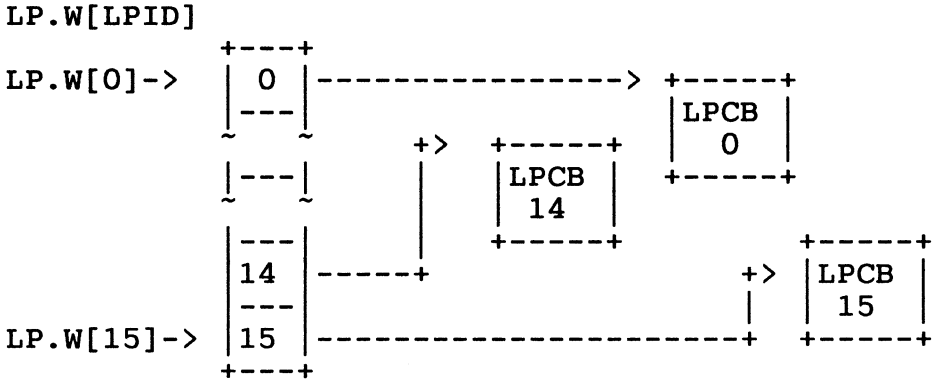


Figure 4.3

MYLPCB.W is a pointer to the LP that this JP is attached to. This field is used by each processor when it is trying to quickly find the address of the LP and is much faster than constantly looking at the table of LPs(LP.W) to find the LP the JP is attached to. There is a separate copy of this field for each JP on the system. (See JP Management.) This field is initialized at SINIT for the mother processor and by ?JPINIT for daughter processors. The field is modified by ?JPMOV. The reason for this pointer's existence is speed. This field is used at interrupt level, during scheduling, and during statistic gathering operations.

LPCNT is a one-word integer used for counting the number of LPCBs in the system. This field is set to 1 at assembly time because LP0 exists when the system comes up. LPCNT is incremented when an LP is ?LPCREAted and decremented when an LP is ?LPDELeted. During ?LPCREA this field is also compared to MXLPCB to prevent the user from creating more than the maximum number of allowable LPCBs.

LPMAP.W is a 32-bit field used as a bit map of all user visible LPs in the system. A user visible LP is an LP that the user can query by system call. LPMAP.W is initialized with the zero bit set to show that LP0 exists. This field is indexed by LPID. During ?LPCREA this field is modified to reflect the new LP added to the system. This field is also queried during ?LPCLASS to check for the existence of an LP. LPMAP is also copied into the user's packet in the ?LPSTAT call. During ?LPDEL the bit for the corresponding LP being deleted is cleared. The reason for this is ?LPDEL does not de-allocate the LPCBs, it merely makes them invisible to the user. When the LP is deleted, there still may be some time charges to the LP so it cannot be deleted.

LPCBMAP.W is a 32-bit field used as a bitmap of all existing LPCBs in the system. LPCBMAP.W is initialized with the zero bit set to show that LP0 exists. The bits in this field are indexed by LPID. During ?LPCREA This field is checked to see if the LPCB for this LPID already exists. (See Section 4.3.2 ?LPDEL for why this field is used.) If the LPCB does exist, there is no need to get memory for the LPCB; the routine merely reuses the memory. This field is not affected by ?LPDEL.

LPCB.TMP is the address of an internal blank copy of an LPCB. The internal copy contains all the field definitions and some initial settings so the new LPCB can be created in one instruction (WBLM) with LPCB.TMP as a template instead of clearing individual fields. This block has the initial settings of the LPCB. This field is used at ?LPCREA time as a template for initialization of the LPCB.

JPLPLK.W is a 32-bit lock field used as a global lock for the entire JP and LP environment. This field is used as a pend and a spin lock. This lock is used to prevent all other JPs on the system from modifying any LPs or affecting attachments of JPs to LPs. For more information on how this field is used, see Section 4.2.5 LP Locking; for more on the concepts of global locking, see ELQUE Management.

MLPCB.W is a 32-bit pointer to the MOTHER LPCB. This field contains the address of the LPCB that contains the MOTHER JP. This field gets changed during ?JPMOV if the mother JP is being moved to another LP. MLPCB.W is also set during system initialization when the initial LP(LP0) is initialized.

MXLPCB is a one-word integer constant containing the maximum number of LPs that can exist on the system. The value of MXLPCB is 020(16.).

CLSCHD.W is a 32-bit field that contains a bit map of LPs that have class scheduling enabled. This field is indexed by LPID. If the bit corresponding to an LP is set, then the LP has Class scheduling enabled. This field is initially 0. CLSCHD.W is modified during the ?CLSCHD system call if the user requests that class scheduling be enabled or disabled.

CLACC.W is a 32-bit field that contains a bit map of LPs that have class scheduling ACCUMULATE mode enabled. This field is indexed by LPID. If the bit corresponding to the LPID is set then that LP has ACCUMULATE mode enabled. CLACC.W is modified during the ?CLSCHD system call if the caller requests ACCUMULATE mode to be enabled or disabled.

4.2.3 Basic Operations

There are seven basic operations that affect the LPCB: attach, detach, update class times, manage interval, reset, update total time counters, and update scan mask. The seven operations are explained and pseudocoded in the sections below.

4.2.3.1 Attach

Attach is an operation that affects both the JP and LP. It is the operation that attaches a JP to an LP. This operation increments the LP's JP count, puts the JP in the LP's JP bit map and puts the LPCB address in the PPCB. To use this function the LPJPLOCK must be held.

```
/* $$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$ */
/*                                                 */
/*                    ATTACH                       */
/*                                                 */
/*  This routine attaches a JP to an LP.           */
/*                                                 */
/*                                                 */
/* $$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$ */

attach(lpcb,ppcb)
  {
/* *********************************************** */
/*                                                 */
/*   If the PPCB is already attached to an LP then */
/*   panic with a 14615.  This is the attach panic.*/
/*                                                 */
/* *********************************************** */

     if   (ppcb.lpcb.w != 0)
        panic(14615);
     else
        ppcb.lpcb.w = lpcb;

     jpid = ppcb.jpid;
     lpcb.lpjpcnt++;
/* *********************************************** */
/*                                                 */
/*   If the LP already shows that the JP is        */
/*   attached to it then panic with a 14615.       */
/*   Otherwise, set the bit in the bit map for     */
/*   that JPID.                                    */
/*                                                 */
/* *********************************************** */
     if   (bit(lpcb.lpjpmp.w,jpid) != 0)
        panic(14615);
     else
        setbit(lpcb.lpjpmp.w,jpid);
```

```
/* ********************************************** */
/*                                                */
/*  If this is the mother JP we're attaching then  */
/*  set the "mother is attached to this lp" LPMOM  */
/*  bit in the LPCB.                              */
/*                                                */
/* ********************************************** */

     if (bit(ppcb.cpstat,cpmast) == 1)
        {
        lock_lpcb(lpcb);  /* this turns off interrupts*/
        setbit(lpcb.lpstat,lpmom);
        unlock_lpcb(lpcb); /* interrupts enabled */
        }
     return();
  }/* attach operation */
```

4.2.3.2 Detach

This operation detaches a JP from an LP. This is called when doing a JPMOV, a JPREL, or if a JPINIT fails. This operation affects the LPCB by decrementing the JP count and removing the JP from the LP's JP bit map. To use this function the LPJPLOCK must be held.

```
/* $$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$ */
/*                                                           */
/*                      DETACH                               */
/*                                                           */
/*  This routine detaches a JP from an LP.                   */
/*                                                           */
/*                                                           */
/* $$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$ */

detach(lpcb,ppcb)
  {
/* ********************************************** */
/*                                                */
/*   If there is no LPCB attached to the PPCB then */
/*   panic with a 14616.  This is the detach panic.*/
/*                                                */
/* ********************************************** */
     if (ppcb.lpcb.w == 0)
        panic(14616);
     else
        ppcb.lpcb.w = 0;

     jpid = ppcb.jpid;
     lpcb.lpjpcnt--;
/* ********************************************** */
/*                                                */
/*   If the JP to be detached from the LP is not  */
/*   there then panic with a 14615.               */
/*   Otherwise, clear the bit in the bit map for   */
/*   that JPID.                                   */
/*                                                */
/* ********************************************** */
     if   (bit(lpcb.lpjpmp.w,jpid) == 0)
        panic(14615);
     else
        clearbit(lpcb.lpjpmp.w,jpid);
```

```
/* ********************************************** */
/*                                                */
/*  If this is the mother JP we're detaching then  */
/*  clear the "mother is attached to this lp" LPMOM*/
/*  bit in the LPCB.                              */
/*                                                */
/* ********************************************** */

     if (bit(ppcb.cpstat,cpmast) == 1)
      {
        lock_lpcb(lpcb);
        clearbit(lpcb.lpstat,lpmom);
        unlock_lpcb(lpcb);
       }  /*  if */
     return();
  } /* detach operation */
```

4.2.3.3 Update Class Timings

This function updates the timing variables for class scheduling. To make decisions in class scheduling, class statistics must be kept current. After a PTBL runs, this operation is called if class scheduling is "on" or in "accumulate" mode. This function is not explicitly implemented as a function in the code, it is inline code.

```
/* $$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$ */
/*            update_class_timings                */
/*                                                */
/*  This routine updates a class time in the LPCB. */
/*  It is assumed that class scheduling is on or   */
/*  accumulating.  Element_addr is the address of  */
/*  the PTBL or CB being charged.                  */
/* $$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$ */

update_class_timings(lpcb,element_addr,time_used);
  {
     lock_lpcb(lpcb);
/* ********************************************** */
/*  Subtract from the total allowable time for that*/
/*  class.  If there is no time left then call     */
/*  update the class mask.                         */
/*                                                 */
/* ********************************************** */

     class_offset = element_addr.pgnum.w;
     lpcb.lpctm.w[class_offset] -= time_used;
     if (lpcb.lpctm.w[class_offset] <= 0)
        update_class_mask(class_offset,lpcb);
     return();
  } /* operation */
```

4.2.3.4 Manage Interval

This function manages the class scheduling time interval. Each time a process stops running the time it used is subtracted from the interval. If time ran out then restart the interval. This function is part of inline code, it is not actually a separate routine.

```
/* $$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$ */
/*                    manage_interval                      */
/*                                                         */
/*  This routine updates and restarts the time             */
/*  interval (if needed).  Assume class scheduling         */
/*  is on.                                                 */
/* $$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$ */

manage_interval(lpcb,time_used)
  {
     lpcb.lpciu.w -= time_used;
/* ********************************************************* */
/*                                                         */
/*  If the time interval ran out, then we must             */
/*  restart it.  First update the total interval           */
/*  counter.                                               */
/* ********************************************************* */

     if (lpcb.lpciu.w <= 0)
       {
        lpcb.lptint ++;
        reset(lpcb);
       }  /* if */
     unlock_lpcb(lpcb);
     return();
  } /* manage interval operation */
```

4.2.3.5 Reset

This operation is similar to restarting the time interval. It resets the class mask and class counters. This function is a routine in the module SCHED but is not always called. There are some places inline that do this functionality but do not make the call to RESET.

```
/* $$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$ */
/*                        reset                                */
/*  This function resets the Class scheduling                  */
/*  values.   It does this by BLMing part of the               */
/*  fixed values of the LPCB into the fields that              */
/*  can change during the interval.                            */
/* $$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$ */

reset(lpcb)
  {
     lock_lpcb(lpcb);
     BLM(&lpcb.lpcsm,&lpcb.lpism,lpblm);
     unlock_lpcb(lpcb);
     return();
  }  /* reset operation */
```

4.2.3.6 Update Totals Counters

This function updates the time totals in the LPCB. The routine will be called with the offset into the LPCB to update. This function is implemented inline and, therefore, does not explicitly exist as a function. This does not reflect collision counters that are updated when a lock collision is encountered. A lock collision is an attempt to lock a locked lock.

```
/* $$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$ */
/*              update_totals_counters                         */
/*  This routine updates the total of the counter              */
/*  supplied in the offset with the amount of time             */
/*  used.  If there is a collision the global                  */
/*  counter CLPDE.W is updated. (Collision counting*/
/*  is not in pseudocode.)  A collision occurs when*/
/*  the calling routine fails to get a lock on a               */
/*  database (in this case a control block).                   */
/*                                                             */
/* $$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$ */

update_totals_counters(lpcb,offset,time_used)

     lock_lpcb(lpcb);
     (long)lpcb.offset  += time_used;
     unlock_lpcb(lpcb);
     return();
```

4.2.3.7 Update Scan Mask

This operation modifies the scan mask because a class has used up its interval. Each time a class uses its portion of an interval, all members of that class must be masked out for the rest of the interval. This operation is only called if class scheduling is ENABLED or "on." For more information on how this function is used see Section 4.2.4 Scanner. This function does not explicitly implement as a function in the source, but is implemented inline.

```
/* $$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$ */
/*              update scan mask                  */
/*  This routine updates the scan mask using      */
/*  the class offset supplied to the routine.     */
/*  Also supplied is the LPCB address.            */
/* $$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$ */

update_scan_mask(lpcb,class_offset)
   {
     setbit(lpcb.lpcsm[class_offset]);
     return();
   } /* update scan mask operation */
```

FINDLP:

Findlp is a function that tries to find an LPCB address. This function uses the LPID to find an LPCB in LP.W. If the routine does not find the LP it takes the error return.

```
/* $$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$ */
/*              findlp(lpid)                      */
/* This routine tries to find an LPCB in LP.W.    */
/* If successful then the routine returns an LPCB */
/* address. If not successful then the routine    */
/* takes an error return.                         */
/*                                                */
/* $$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$ */

findlp(lpid)
     {
```

```
/* ********************************************** */
/*                                                */
/*  First check in the LP bit map (LPMAP.W) to see */
/*  if the LP exists.  If not take the error      */
/*  return.  If the LP exists then return the     */
/*  LPCB address.                                 */
/* ********************************************** */

     if (bit (lpmap.w,lpid) != 1)
        return(error_return);
     else
        return(lp.w[lpid]);
     }  /* findlp() */
```

4.2.4 Paths that Affect the LPCB

There are three main paths that affect the LPCB: the Scanner, LPCB accounting, and user service routines (defined in the User services section).

4.2.4.1 The Scanner

The scanner is the path, in the module SCHED, that scans ELQUE to find a CB or PTBL to run. This path is not called "scanner" in the code, it has several entry points. For an explanation of the entry points into the scanner, see ELQUE Management. The scanner affects the LPCB when there is a RESET of the LPCB databases after a scan of ELQUE fails. The scanner also affects the LPCB when there is a mode change.

The scanner is presented from the LP point of view. The reason for this is that the scanner in the system imbeds three logically separate points of view inline. For example, consider the following lines of C code.

```
A:    if (bit(myppcb.w.cpstat,cpmast) !=1)/*daughter?*/
          setbit(mask,process_mother_bit);

B:    element = SCAN(*ELQUE,mask);

C:    if (mask == mylpcb.lpciu.w)
          {}
```

"A:" is supplied by JP management because the if statement uses the global MYPPCB.W to find out if this is a daughter processor (see JP management).

"B:" is supplied by ELQUE management because the scan function uses ELQUE, which is managed in ELQUE management.

"C:" is supplied by LP management because the initial scan mask comes from the LPCB.

In the above example, three major areas of Paths and Time are used in three nearly consecutive commands. This is the way the code is really presented in the system, but to modularize the Scanner for each section the scanner is presented with different emphasis.

The pseudocode below shows the scanner from the LP point of view.

```
/* $$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$ */
/*                                                           */
/*                    The Scanner                            */
/*     The scanner will loop forever unless left             */
/*     to go_idle.  (See ELQUE management.)                  */
/*                                                           */
/* $$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$ */

#define loop_forever true
scanner( )
{
int  model_tier; /* variable used for call to set_mask*/

model_tier = 0;
while(loop_forever)
  {
/* ************************************************** */
/*                                                    */
/*     Get the current mask.  If child processor then */
/*     mask out the mother bit and scan ELQUE.        */
/*     MYPPCB.W is the PPCB for that JP.              */
/*     mylpcb.w is the address of this LPCB.          */
/*     Setmask is a function in JP management.        */
/* ************************************************** */

     mask = set_mask(mode,mylpcb,model_tier);
     if (bit(myppcb.w.cpstat,cpmast) !=1)/*daughter?*/
        setbit(mask,process_mother_bit);
     element = SCAN(*ELQUE,mask);

/* ************************************************** */
/*  If the scan was successful the element will get   */
/*  dispatched.  (See ELQUE management.)              */
/*                                                    */
/* ************************************************** */

     if (successful_scan)
        dispatch_element(element);
     else
```

```
/* ************************************************** */
/*                                                    */
/*  If not successful then check to see if a mode     */
/*  change is necessary.   If the current             */
/*  mask is the same as the initial scan mask         */
/*  RESET and change mode.  This is considered a      */
/*  sufficient check because the scan failed with     */
/*  the initial mask  meaning there are no more       */
/*  primary classes ready to run.  To avoid a second  */
/*  unnecessary scan, RESET to run the secondary      */
/*  classes.                                          */
/* ************************************************** */

        if (mask == mylpcb.lpciu.w)
          {
           reset(mylpcb.w);
          }  /* if */
        else

/* ************************************************** */
/*                                                    */
/*  If not changing the mode then check to see        */
/*  what mode we're in.  If mode 0 then reset the     */
/*  LP databases and get a new mask.  If not, get the */
/*  next tier if possible.                            */
/* ************************************************** */
           if (myppcb.current_mode == 0)
               reset_(mylpcb.w);
               model_tier = 0;
           else
               if (myppcb.cptmk.w+2 > myppcb.cphmk.w+32)
                   {
                   myppcb.current_mode = 0;
                   model_tier = 0;
                   }
               else
                   {
                   myppcb.cptmk.w += 2;       /* JP */
                   model_tier =  myppcb.cptmk.w;
                   }/* else */
  }  /* while loop */
 }  /*  scanner  /*
```

4.2.4.2 LP Accounting

The second path that affects the LPCB is the LP accounting path. After an element runs the correct LPCB databases must be updated. For example, if class scheduling is enabled and the element running was a PTBL (see ELQUE management) then update the timings for the class that element belongs to and decrement the amount of time left in the interval. If the interval runs out then RESET the interval and mode.

The pseudocode below shows how the LP accounting path works. There are two separate paths to do the statistic updates: one is for PTBLs (entry point is PTUPDT) and the second is for CBs (entry point CBUPDT). For more information on the PTBL and CB see ELQUE management.

```
/* $$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$ */
/*         update user run stats  ptupdt                          */
/* This routine is called each time a PTBL stops                  */
/* running.                                                       */
/* The argument to this routine is the PTBL address.              */
/* (See ELQUE management.)                                        */
/*  A global counter is incremented to count the                  */
/*  number of times this is done.  The global is                  */
/*  NPTUPDT.W. (See ELQUE management.)                           */
/* $$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$ */
```

```
ptupdt(ptbl)
  {
     nptupdt.w ++;
     pextn = ptbl.pextn.w;
/* ************************************************** */
/*  If the current subslice residue (PSL) and the     */
/*  old subslice residue are equal, then the process  */
/*  ran for zero time and we do not need to update    */
/*  the statistics.                                   */
/*                                                    */
/* ************************************************** */
     time_used = pextn.psl - pextn.opsl;
     if (time_used == 0)  /* see ELQUE mgnt  */
        return();
/* ************************************************** */
/* If the JP count for the LPCB is zero, then we cannot*/
/* charge time to that LPCB because without a JP we   */
/* couldn't have run.  JPs run code, not LPs.  What   */
/* must have happened was the JP was somehow moved    */
/* while the PTBL was running and it was on the mother */
/* JP and it was the caller of the ?JPMOV.  This      */
/* means that the timing for this run would be lost.  */
/*                                                    */
/* ************************************************** */
     lpcb = pextn.plpcb.w;
     if (lpcb.jpcnt == 0)
        return();
```

```
/* ************************************************ */
/*                                                  */
/*  If we are a "mother only" call and the LPCB status*/
/*  says the mother is attached to this LPCB, then    */
/*  update the mother-only stats, or else update the  */
/*  "run anywhere" stats.  These stats are generalized*/
/*  (not class specific).  If the PTBL and            */
/* the LPCB mom bits disagree then this process must  */
/* have JPMOVed the mother processor while it ran.    */
/* ************************************************ */

     if ((bit(ptbl.pstat,pmast) == 1)&
     (bit(lpcb.lpstat,lpmom) == 1))
        update_totals_counters(lpcb,lptmpt.4,time_used);
     else
        update_totals_counters(lpcb,lptpt.4,time_used);

/* ************************************************ */
/* If we are not accumulating satistics then        */
/* return.                                          */
/* ************************************************ */

     if (bit(lpcb.lpstat,lpacc) == 0)
        return();
/* ************************************************ */
/*   Otherwise add the time used to the total       */
/* time used and to the total time used by that class.*/
/* Call the general accounting routine to charge    */
/* general statistics and deal with the time interval.*/
/* Charge user is called COMN3 which is not an entry  */
/* point into SCHED.                                */
/* ************************************************ */
     charge_user(lpcb,ptbl,time_used);
     return();
  }
```

The second update path that affects the LPCB deals with CBs (see ELQUE management). The entry point for this section is CBUPDT (CB UPDaTe). This routine updates the LPCB general total counters.

```
/* $$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$ */
/*                      CBUPDT                                    */
/*  After a CB runs the time it ran needs to be                   */
/*  accounted for in the LPCB.  Depending on the type             */
/*  of CB, different counters will be updated.                    */
/*  The argument passed to the routine is the CB                  */
/*  address.                                                      */
/*  A global counter is incremented to count the                  */
/*  number of times this is done.  The global is                  */
/*  NCBUPDT.W.                                                    */
/* $$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$ */

cbupdt(cb)
  {
     ncbupdt.w++;
/* ************************************************************ */
/*  Get the time used.  If the time is zero then no             */
/*  time was used so return.  Otherwise add time to             */
/*  total system call time in the CB and clear PTIME.           */
/*                                                              */
/* ************************************************************ */

     time_used = cb.ptim;
     if (time_used == 0);
        return();
     cb.ttime += time_used;
     cb.ptime = 0;
```

```
/* ************************************************ */
/*  Get the system call number out of the CB.  If the */
/*  call number is -1 then the CB is a daemon.  Update*/
/*  the daemon counter in the LPCB.                   */
/* ************************************************ */

     call_number = cb.calln;
     if (call_number == -1)
       {
        lpcb = mylpcb.w;
        update_totals_counters (lpcb,lptdae.4,time_used);
       }
     else
/* ************************************************ */
/*  If the "mother only" bit is set in the CB then    */
/*  the mother-only counter is updated on the LP.     */
/*                                                    */
/* ************************************************ */

        if (bit(cb.pstat,pmst) == 1)
          {
           lpcb = mylpcb.w;
           update_totals_counters (lpcb,lptmsy.4,time_used);
          }
        else
/* ************************************************ */
/* The CB is not a daemon or a "mother only" then it  */
/* is a "run anywhere" system call.  Update the "run  */
/* anywhere" system call counter.                     */
/* ************************************************ */

           {
           lpcb = cb.plcb.w;
           update_totals_counters (lpcb,lptsys.4,time_used);
           }
      }
```

```
/* ************************************************ */
/* If the LP is not in accumulate mode then return.   */
/*                                                    */
/* ************************************************ */

     if (bit(lpcb.lpstat,lpacc) == 0)
        return();
/* ************************************************ */
/*  If the LPCB in the CB is different from the LPCB  */
/*  we're currently charging against, then make the   */
/*  LPCB from the CB the charged LPCB.                */
/*  This check is done to ensure that the LPCB has not*/
/*  changed since the CB ran.                         */
/* ************************************************ */

     if (cb.plpcb.w != lpcb)
       {
        lpcb = cb.plpcb.w;
/* ************************************************ */
/* If the JP count in the LPCB is 0 then we cannot    */
/* charge for time if we have no processor to run on, */
/* so return.  This will happen if the CB ran on the  */
/* mother and the CB did a ?JPMOV of the mother.      */
/* ************************************************ */

        if (cb.plpcb.jplpcnt == 0)
           return();
/* ************************************************ */
/* Call the general accounting routine to charge      */
/* general statistics and deal with the time interval.*/
/*                                                    */
/* ************************************************ */

     charge_user(lpcb,ptbl,time_used);
     return();
  }
```

```
/* $$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$ */
/*                    Charge User                                      */
/*                                                                     */
/* This is the general routine called from both                        */
/* CBUPDT and PTUPDT to do general statistics and                      */
/* work with the time interval.                                        */
/* This common function charges the LPCB for total                     */
/* time used in the LP and charges a particular                        */
/* class for the time used by an element.                              */
/* Charge_user is a routine called COMN3 in SCHED.                     */
/* COMN3 is not an entry point, therefore, it has been                 */
/* renamed.                                                            */
/* $$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$ */

charge_user(lpcb,ptbl,time_used)
   {
     update_totals_counters (lpcb,lptuse.4,time_used);
     class_offset = ptbl.pgnum;
     update_totals_counters (lpcb,lptsum.4,time_used);

/* ************************************************************** */
/*                                                                */
/*   If class scheduling is on, then manage the time              */
/*   interval.  ** NOTE if LPOFF = 1 then class                   */
/*   scheduling is off. **                                        */
/*                                                                */
/* ************************************************************** */

     If (bit(lpcb.lpstat,lpoff) == 0)
       {
        update_class_timings(lpcb,ptbl,time_used);
        manage_interval(lpcb,time_used);
       }
     return();
  }
```

4.2.5 LP Locking

When doing work with the LPCB, locks must be held to prevent other JPs from touching a database while this JP is trying to work with it. There are two levels of locking used for working with the LPCB: the first is the Global JP LP lock, and the second is the local lock.

The main JP LP lock used is called JPLPLCK.W. JPLPLCK.W is a global lock used when doing something that could affect more than one LP or JP in the system. While this lock is held, all other LPJP requests must spin or pend depending on the type of JPLPLCK it held. For more information on the JPLPLCK.W lock words, see ELQUE management.

JPLPLCK is only used by the MP system calls (except ?PROC). These are paths that can pend on the lock because they run as control blocks (see ELQUE management). These paths can also spin on a lock. For example: The system call ?LPDEL makes a call to the JPLPLCK routine to keep all other JPs away from this LPCB. Keeping the other JPs away means no other LP or JP system calls (MP system calls) will be allowed to work on any LPCB or PPCB. All LP system calls use this mechanism when working on the MP environment.

JPLPLCK is currently an unnecessary lock because all MP system calls run "MOTHER ONLY". This lock was designated for the future when more than one JP will be allowed to service MP system calls. The only non-MP call that uses this lock is ?PROC and that call is also a "MOTHER ONLY" call.

Below is the pseudocode for the general locking and unlocking routines used in LP management.

4.2.5.1 GET_LOCK

This routine locks JPLPLCK.W for "read" or "change" depending upon the call type.

```
/* $$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$ */
/*          get_lock (change or read)                */
/*  In reality this is two routines called           */
/*  GCLOCK and GRLOCK, respectively.                 */
/*  This routine is the general locking routine.     */
/*  It gets a transition lock from the lock,         */
/*     word, locks it, locks the change lock,        */
/*     and clears the transition lock.               */
/*  This routine will also pend if the change        */
/*  lock is held.                                    */
/* $$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$ */

get_lock(type,lock)
 {
begin:

/* ********************************************** */
/*  This part spins waiting for the transition    */
/*  lock to be released.                          */
/*                                                */
/* ********************************************** */

      while (bit(lock.ptran) = 1)
      {}
   setbit(lock.ptran);   /* set transition lock*/

/* ********************************************** */
/*  Now we have the transition lock, so we can    */
/*  try to see if the change lock is held. If     */
/*  it is, pend and show there are waiters on     */
/*  the lock.                                     */
/* ********************************************** */
   if (bit(lock.pdchng)
     {
      setbit(lock.pdwait);
      clearbit(lock.pdtran);
      pend(jplplock.w);
      goto begin;
       }

/* ********************************************** */
/*  If the requestor wants the change lock,       */
/*  then set the change lock.                     */
/*                                                */
/* ********************************************** */

   if (type == change)
      setbit(lock.pdchng);
```

```
/* ****************************************** */
/*  We have the locks we want now, so clear   */
/*  pdtran, increment the lock user count, and */
/*  return.                                   */
/* ****************************************** */

   lock.pdusers +=;
   setbit(lock.pdtran);
   return( );
```

4.2.5.2 Release Lock

This routine releases the global (JPLPLCK.W) and unpends the waiters if there are any.

```
/* $$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$ */
/*       release_lock (change or read)         */
/*  In reality this is two routines called     */
/*  GCUNLOCK and GRUNLOCK, respectively.       */
/*  This routine is the general unlocking      */
/*  routine. It gets a transition lock from the */
/*     lock word, locks it, unlocks the change  */
/*     lock, and clears the transition lock.    */
/*  This routine will also unpend anyone waiting*/
/*  for the change lock.                       */
/* $$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$ */

release_lock(type,lock)
 {
/* ****************************************** */
/*  This part spins waiting for the transition */
/*  lock to be released.                      */
/*                                            */
/* ****************************************** */
      while (bit(lock.ptran) == 1)
      {}
   setbit(lock.ptran);   /* set transition lock*/
/* ****************************************** */
/*  Now we have the transition lock, so we can */
/*  clear  the change lock.                   */
/*                                            */
/* ****************************************** */
   clearbit(lock.pdchng);
```

```
/* ****************************************** */
/*  The change lock is now released, unpend the */
/*  waiters if any, release ptran,  and return. */
/*                                             */
/* ****************************************** */
   setbit(lock.pdtran);
   if (bit(lock.pdwait) == 1)
      upend(jplplock.w);
   return( );
```

4.2.5.3 Lock LPCB

This operation is the local locking mechanism for access to the LPCB. The local lock refers to the lock in the LPCB status word. There is a lock bit in the status word that allows locking called LPLCK. To modify the LPCB this lock must be held. This lock is a spin lock. This function sets the lock bit and turns off interrupts. This is a lower-level function than the JP LP locking function. Lower level means the use of this lock does not affect any other LP in the system. The pseudocode below shows how this operation works.

```
/* $$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$ */
/*                    lock lpcb                    */
/* This function locks the LPCB that will be       */
/* changed.  It sets the bit in the status word    */
/* that locks the LPCB.  If it is locked then the  */
/* routine will spin on the lock.                  */
/* LPCB is the address of the LPCB to be locked.   */
/* $$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$ */

lock_lpcb(lpcb)
   {
     while(bit(lpcb.lpstat[lplck]) == 1)
       {}
     interrupts(off);  /* turn off interrupts */
     setbit(lpcb.lpstat.[lplck]);
   }  /*  lock LPCB operation  */
```

Unlock LPCB

This operation unlocks the LPCB. Since the lock function spins on a lock, it spins on the assumption that it won't spin for long, therefore, after performing an operation on the LPCB the unlock function must be called. This operation clears the lock bit in the status word of the LPCB whose address is supplied to the routine. The pseudocode below shows how this routine works.

```
/* $$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$ */
/*                 unlock lpcb                      */
/* This routine clears the LPCB lock bit.  The LPCB*/
/* address is supplied to this routine.             */
/* $$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$ */

unlock_lpcb(lpcb)
  {
     clearbit(lpcb.lpstat[lplck]);
     interrupts(on);  /* turn on interrups  */
     return();
  } /* unlock lpcb operation */
```

4.3 User Services

LP and CLASS management provide the user with different user services. These services can be accessed by system calls. The system calls that LP management services are ?LPCREA, ?LPDEL, ?LPSTAT, and ?LPCLASS. The sections below will show how the system call works and why it is used.

4.3.1 ?LPCREA

?LPCREA is the call used to create the Logical Processor. This call creates an LPCB. The entry point for this call is LPCREA.P. When an LP is created, class 0 (or A in CLASP terms) is the only class that is defined. The default interval is 4 seconds. Figure 4.4 shows the default LP or LP0. (For more information see the CLASP manual.)

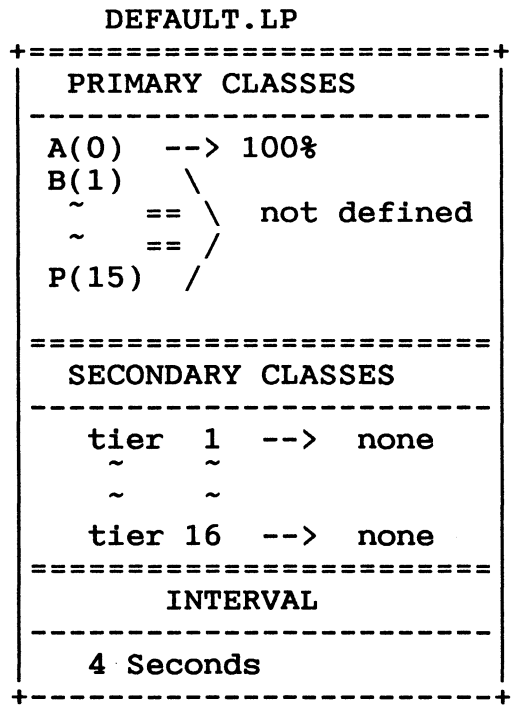


Figure 4.4

```
/* $$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$ */
/*                      lpcrea.p                                  */
/*  This service creates the LP so it can be used                 */
/*  in the MP environment.                                        */
/*                                                                */
/* $$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$ */

lpcrea.p()
   {
/* ********************************************************** */
/*                                                            */
/* If the user does not have the MP privilege(sysprv)         */
/* then exit with the "not privileged for this                */
/* action" error.                                             */
/* This function calls a routine called mp_priv_check.        */
/* The mp_priv_check routine checks to see if the             */
/* user has "SYSTEM MANAGER" privilege "on" and logs          */
/* the attempt.   If "SYSTEM MANAGER" is not on for           */
/* the caller then mp_priv_check will return an               */
/* error.                                                     */
/* ********************************************************** */

     if (!(mp_priv_check())
        return(not_priv_for_action);

/* ********************************************************** */
/* There are two locks used to work with the LPCB.            */
/* The reason for this is that some other JPs may be          */
/* running on this LPCB, so we do not want the LPCB           */
/* to be invalid while we are trying to read it (read         */
/* lock)  and we do not want updates to occur while we        */
/* are changing the database.(change lock)                    */
/* ********************************************************** */

     get_lock(change);

/* ********************************************************** */
/*                                                            */
/* If we are at the max number of LPs (16) then               */
/* exit with "too many LPs" error.                            */
/* lpcnt is the LP count global. mxlpcb is the global         */
/* for the maximum number of LPs allowed in the               */
/* system.                                                    */
/* ********************************************************** */

if (lpcnt <= mxlpcb)
     {
     release_lock();
     return(exceded_max_lp_count);
     }
```

```
/* ********************************************** */
/*                                                */
/* If the LP map of allocated lpcbs (lpcbmap.w) and  */
/* the map of user visible LPs (lpmap.w) are      */
/* different, then the LPCB does not need to be   */
/* allocated. There is an lpcb already allocated. The*/
/* reason for this ?LPDEL does not de-allocate the   */
/* LPCBs, it merely makes them invisible to the user.*/
/* When the LP is deleted there still may be some    */
/* time charges to the LP so it cannot be deleted.   */
/* ********************************************** */

     if (lpmap.w != lpcbmap.w)
       {

/* ********************************************** */
/*  Now loop and find the bit in the maps that are   */
/*  different.  Once found break out of the       */
/*  loop.   Then get the LPCB address from the global*/
/*  table of lpcb addresses called lp.w.          */
/* ********************************************** */

        for (i=0;i<32;i++)
           if (bit(lpmap.w[i]) != bit(lpcbmap.w[i]))
               break;
        lpcb = lp.w[i];
       }
     else
/* ************************************************** */
/*                                                    */
/*  If lpmap.w and lpcbmap.w are the same then we     */
/*  must call memory management to allocate the memory */
/*  for the LPCB.   lpcbln is the length of the lpcb.  */
/*  GSMNW is the memory management function to get    */
/*  memory with no wait.                              */
/*  If there is no memory to be had, then the call will*/
/*  take an error return.  If the memory call does,   */
/*  then exit with a memory error.                    */
/* ************************************************** */

       {
        lpcb = gsmnw(lpcbln);
        if(error_return)
           {
           release_lock();
           return(memory_error);
           }
```

```
/* ************************************************** */
/*  Now that we have the memory for the lpcb and the  */
/*  address in variable lpcb, we must find a place in */
/*  the lp.w table to put the lpcb address.           */
/*                                                    */
/* ************************************************** */

        for (i=0;i<16;i++)
           if (lp.w[i] == 0)
               break;
        } /* for loop */

/* ************************************************** */
/*  Now that we have an LPCB, we have to set it up with*/
/*  initial values.  We will BLM a skeleton LPCB      */
/*  which is a global call lpcb.tmp.  This global     */
/*  can be found in the STKS module in the sources.   */
/*  Add the new lpcb to the LP global databases and   */
/*  return to the caller.                             */
/* ************************************************** */

     BLM(lpcb,*lpcb.tmp,lpcbln);
     lpcb.lpid = i; /* i is used in the for loops above*/
     lpcb.lptmk.w = lpcb.lphmk.w;
     lpcnt ++; /* increment total # of LPs in system */
     lp.w[lpcb.lpid*2] = lpcb; /* double word table */
     setbit(lpmap.w[lpcb.lpid]);
     setbit(lpcbmap.w[lpcb.lpid]);
     release_lock();
     Set_up_trap_handler();
     packet_address = CC.W->CATCB.W->TAC2.W;
     packet_address.lpid = lpcb.lpid;
     clear_trap_handler();
     return();
```

4.3.2 ?LPDEL

To complement the ?LPCREA system call there is a ?LPDEL system call. One unique thing that the LPDEL does is to delete the LP but keep its memory around. The reason for not deleting the memory is for the scheduler. During the course of scheduling, various values are taken from the CB/PTBL and added to the LPCB database. This addition is only done when "necessary" (not on every reschedule). The problem is that if the LPCB were deleted, the scheduler could very well add the statistics to the LPCB memory (which is now used for something else) and the system will likely die a humorous and non-reproducible death. The scheduler cannot check for the LPCB existence because it would take too long for the heavily traveled routine. So keep it around and let the scheduler update it with meaningless statistics.

LPs can have JPs attached to it or not. Since the user can change the number of JPs attached to LPs or even delete LPs outright, the system has no way to guaranty that an LP will always be there for accounting. Thus, to avoid PANICing the system keeps deleted LPCB memory around.

```
/* $$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$ */
/*                          lpdel.p                                */
/* This routine deletes an LP from the system.                     */
/* It deletes the LP as far as the user is concerned,              */
/* but leaves the LPCB still allocated and on the                  */
/* user invisible LP map.                                          */
/*                                                                 */
/* $$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$ */

lpdel.p()
  {
     if (!(mp_priv_check()))
        return(not_priv_for_action);
     set_up_trap_handler();
     packet = cc.w->catcb->tac2;
     lpid = packet->lpid;
     clear_trap_handler();

/* ************************************************************** */
/*  Validate the user supplied LP ID.  It must be                  */
/*  >= 0 and <= 15.                                                */
/*                                                                 */
/* ************************************************************** */

     if (!((lpid >= 0) & (lpid <= 15.)))
        return(invalid_lpid);
```

```
/* ************************************************ */
/*                                                  */
/*  Make sure the supplied LP ID is not zero.  The  */
/*  user is not allowed to delete LP0.              */
/*  If ok then lock the JPLP lock and find the LP   */
/*  based on the LP ID.                             */
/* ************************************************ */

     if (lpid = 0)
        return(can_not_del_lp0);
     get_lock(change);
     lpcb = find_lp(lpid);
     if (error_rtn)
        return(lp_not_found);

/* ************************************************ */
/*                                                  */
/* The user is not allowed to delete any LP that has*/
/* a JP attached to it.                             */
/*                                                  */
/* ************************************************ */

     if (lpcb.jpcnt > 0)
        return(jps_attached);

/* ************************************************ */
/*                                                  */
/*  The user is not allowed to delete any LP that   */
/*  has a JP move in progress.                      */
/*                                                  */
/* ************************************************ */

     if (lpcb.lpmovcnt > 0)
        return(jps_attached);

/* ************************************************ */
/*                                                  */
/*  Remove the LP from the user visible databases   */
/*  and return to the caller.                       */
/*                                                  */
/* ************************************************ */

     clearbit(lpmap.w[lpid]);
     lpcnt --; /* decrement total # of LPs */
     release_lock();
     return();
  }
```

4.3.3 ?LPSTAT

Just as with JPs, we can inquire as to the status of LPs. This call has the capability to be a general type call or a specific call. The general call will tell you how many LPs there are and pass back a bitmap of the numbers of the LPs (LPIDs). The specific call will give back information on a specific LP. Let us now look at the code path followed.

```
/* $$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$ */
/*                     lpstat.p                        */
/* This routine supplies the user with either           */
/* a general status of the LPs on the system or         */
/* status of a specific LP on the system.               */
/*                                                      */
/*                                                      */
/* $$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$ */

lpstat.p();

     if (!(mp_system))
        return(not_an_mp_sys);
     set_up_trap_handler();
     packet = cc.w->catcb.w->tac2.w;
     sub_packet = packet->subpacket_addr;
/* **************************************************** */
/*  If the function code in the packet is a 0 then      */
/*  the user wants a general LP status.  The general    */
/*  LP stat returns the total number of LPs and the     */
/*  user-visible LP bitmap.                             */
/*                                                      */
/*                                                      */
/* **************************************************** */

     switch (subpacket->function)
     {
     case 0:        /*  general function */
        get_lock(read);
        subpacket->lp_count = lpcnt;
        subpacket->lp_map   = lpmp.w;
        release_lock();
        clear_trap_handler();
        return();

     case 1:           /* specific function */
        lpid = packet.lpid;
        clear_trap_handler();
```

```
/* ************************************************** */
/*  Validate the LP ID supplied by the user.          */
/*     0 <= lpid =< 15                                */
/*  after validation find the LP from the LP LPCB     */
/*  table based on the LPID supplied.                 */
/* ************************************************** */

     if (!((lpid >= 0) & (lpid <= 15)))
        return(invalid_lpid);
     get_lock(read);
     find_lp(lpid);
     if (error_return)
       {
        release_lock();
        return(lp_not_inited);
       }
/* ************************************************** */
/*  Put the total number of JPs attached to the LP    */
/*  into the packet.  Put the JP bit map in the       */
/*  packet.                                           */
/*  Return to the caller.                             */
/* ************************************************** */

     set_up_trap_handler();
     subpacket.jp_count = lpcb.lpjpcount;
     subpacket.jp_map   = lpcb.lpjpmp.w;
     release_lock();
     clear_trap_handler();
     return();

     default:
        return(invalid_function);
     }  /* switch  */
} /* ?LPSTAT   */
```

4.3.4 ?LPCLASS

Classes (see class environment) are applied to LPs in the system and there is a system call designed for that purpose, ?LPCLASS. This call defines the class environment on the particular LP. The ?LPCLASS defines what classes run on the LP and the percentage of CPU they get, as well as whether they are primary or secondary.

The strict definition of primary vs. secondary is controlled in the way specified. Primary classes are executed first, followed by secondary. If there is extra CPU time, then the secondary classes get what is left. If there is no extra time, then the secondary processes do not get to run.

The ?LPCLASS system call sets or gets the logical processor class assignments. The pseudocode below shows how this call works.

```
/* $$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$ */
/*                        lpclass.p                               */
/* This routine gets (or sets) the class scheduling               */
/* from (or to) a specified LP.                                   */
/*                                                                */
/* $$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$ */

lpclass.p()
   {
     set_up_trap_handler();
     lpid = packet.lpid;
     switch (packet.get_set_code)
       {
/* ************************************************************ */
/*                                                              */
/*  If the function code is zero then the user wants            */
/*  the get function.                                           */
/*                                                              */
/* ************************************************************ */

        case 0:

           clear_trap_handler();
/* ************************************************************ */
/*                                                              */
/*  validate the LP ID that the user suplied.                   */
/*    0 <= lpid <= 15                                           */
/*  Find the LP from the lpcb table.                            */
/* ************************************************************ */
```

```
            if (!((lpid>=0) & (lpid <= 15)))
                return(invalid_lpid);
            get_lock(read);
            lpcb = findlp(lpid);
            if (error_rtn)
                return(lp_not_inited);

/* ************************************************** */
/*                                                    */
/*  Get the total amount of interval time left in     */
/*  tenths of seconds.  Put the class times used into */
/*  the packet.                                       */
/*                                                    */
/* ************************************************** */

            set_up_trap_handler();
            packet.time = 1000. / *lpcb.lpiiu.w;
            BLM(&packet.base,&lpcb.lppct.w,class_count);
            release_lock();
            clear_trap_handler();
            return();

/* ************************************************** */
/*                                                    */
/*  The user wishes to use the set function.          */
/*  So get the packet information that the user wants */
/*  put in the LPCB and store it on the stack (called */
/*  temp) so it can be validated and used.           */
/*                                                    */
/* ************************************************** */

        case 1:

            interval = packet.interval;
            BLM(&stack_area.percentages,
            &packet.base,class_count);
            clear_trap_handler();
/* ************************************************** */
/*  If the user does not have the privilege to do this */
/*  return an error.  Validate the interval:          */
/*     1 <= interval <= 144.(in tenths of a second)   */
/*                                                    */
/* ************************************************** */

            if (!(mp_priv_check()))
                return(not_priv);
            if (!((interval >= 1) & (interval <= 144.)))
                return(invalid_interval);
            if (!((lpid >= 0) & (lpid <= 15)))
                return(illegal_lp);
            get_lock(read);
```

```
/* ************************************************ */
/*                                                  */
/* If the LP does not exist then return an error.   */
/*                                                  */
/* ************************************************ */

          if (bit(lpmap.w[lpid] != 1))
             {
              release_lock();
              return(LP_not_inited);
             }
          release_lock();

/* ************************************************ */
/* Set up the initial scan mask by using the global */
/* initial scan mask scmask.w.  The upper word is set */
/* later in this code.  This global can be found in */
/* PARS.  Move zeros to the class counters.  Move -1 to*/
/* the secondary class tiers.  If a bit is cleared in a*/
/* tier then that class is in this tier.            */
/* class_count is equal to the number of classes (max) */
/* in the system.  The value of class count is,     */
/* therefore, 16.(020)                              */
/* ************************************************ */

          stack_area.standard_mask = scmask.w;
          stack_area.lcitm.w = 0;
              BLM (stack_area.lcitm.w,stack_area.lcitm.w,
              class_count);
          stack_area.lchmk.w = -1;
          BLM(stack_area.lchmk.w,
          stack_area.lchmk.w+1,class_count-1);

/* ************************************************ */
/* Loop through the class assignments to see if they */
/* are primary or secondary class assignments.      */
/*                                                  */
/* ************************************************ */

          for (i=0;i<16;i++)
         {
/* ************************************************ */
/*  If the class assignment is zero, then the class */
/*  has not been assigned to this LP.  If it is     */
/*  non-zero, then split up the word into the right */
/*  byte and the left byte to see if the class      */
/*  assignment is for a secondary class tier (left  */
/*  byte) or a primary class percentage (right byte). */
/* ************************************************ */
```

```
            if (stack_area.percentages[i] != 0)
                {
                left =
                left_byte(stack_area.percentages[i]);
                right = right_byte
                (stack_area.percentages[i]);

/* ************************************************** */
/* If the left byte is non-zero then the class        */
/* assignment is a secondary class.  Validate         */
/* the tier of the secondary class.  1 <= tier <= 16. */
/* If valid, clear the bit in the tier for that class, */
/* for example, if class 3(class D) were in tier 1    */
/* then the table would look like:                    */
/*  lpcb.lphmk.w (1) 0167777  <--reflects the change  */
/*  lpcb.lphmk.w (2) 0177777                          */
/*           -                                        */
/*           -                                        */
/*  lpcb.lphmk.w (16) 0177777                         */
/*  The above table shows the final result in the LPCB.*/
/*  In the code we're modifying the stack temporary.  */
/* ************************************************** */
                if (left != 0)
                    if (!((left>=1)&(left<= 16.)))
                        clearbit
                        (stack_area.lchmk.w[left],i);
                else

/* ************************************************** */
/* If the left byte is zero then the right byte is    */
/* validated.  1 <= percentage <= 0144(in tenths of a */
/* second)  If valid then assign it to the class      */
/* offset on the stack.                               */
/*                                                    */
/*                                                    */
/* ************************************************** */
                    if(!((right>=1)&(right<=0144)))
                      {
                        clearbit (stack_area.lcism[i]);
                        stack_area.lcitm.w[i] =
                        (lciiu*right)/100;
                      } /* if */
                    else
                        return(illegal_percentage_value);
                  } /* if percentages != 0  */
              }  /* for loop  */
```

```
/* ************************************************** */
/* Validate the secondary class tiers.  This means that*/
/* a tier that does not a have a negative one(-1) in it*/
/* cannot be below a tier that does.                   */
/* tier (1)  0167777                                   */
/* tier (2)  0177777 <-- {  this is  an illegal        */
/* tier (3)  0176777 <-- {  secondary class assignment.*/
/* ************************************************** */

          neg1.flag = 0;
          for(i=0;1<16;i++)
            {
              if (stack_area.lchmk[i] == -1)
                  neg1.flag=1;
              else
                {
                  if (neg1.flag == 1)
                      return(invalid_hierarchy);
            }
          get_lock(change);
          lpcb = find_lp(lpid);
          if (error_rtn)
              {
              release_lock();
              return(lp_does_not_exist);
              }
          lock_lpcb(lpcb);
/* ************************************************** */
/* Put the stack temps into the LPCB and return.       */
/*                                                     */
/* ************************************************** */

          lpcb.lpiiu.w = stack_area.liiu.w;
          lpcb.lpism.w = stack_area.lcism.w;
          BLM(lpcb.lpitm.w,stack_area.lcitm.w,lp_blm-2);
          BLM(lpcb.lpcsm.w,lpcb.lpism,lp_blm);
          BLM(lpcb.lphmk.w,stack_area.lchmk.w,
          class_count*2);
          BLM(lpcb.lppct,stack_area.lchlpv.w,class_count);
          unlock_lpcb(lpcb);
          release_lock(lpid);
          return();

       default:

          return(invalid_code);
   } /* switch */
} /* LPCLASS.P  */
```

4.4 System Services

LP management performs two main system services. The first is managing the class part of the scan mask. The second service is accounting for class and total time used on that LP.

The class part of the scan mask is the first word of the two-word scan mask. It is altered by using the UPDATE_SCAN_MASK elementary operation (see basic operations). This service is only performed if class scheduling is being enforced. Class scheduling is being enforced if the LPOFF bit in the LPCB status word is clear (zero).

The class mask service is performed when the time charging for the run of a process is complete. The LPCB offset for the class the process belongs to is decreased by the amount of PIT ticks the process ran. When the class offset reaches zero, or in some cases less than zero, the associated bit in the scan mask is set(bit = 1). Once this is set the scanner will not hit any PTBL of that class.

Another part of the mask management is the management of the time interval. The time interval is the amount of USER time the user defines. This amount of time is calculated to PIT ticks and then subdivided by the percentage for each class.

The following example shows an interval broken into ticks and those ticks divided into four classes.

```
INTERVAL     =   10  SECONDS       *  USER VIEW    *
   TICKS   =  (SECONDS * 1000) * MILLI SECONDS *
INTERVAL     =  10000 PIT TICKS    *  SYSTEM VIEW *

CLASS 0(A)   = 20 %                *  USER VIEW    *
CLASS 0(A)   = 2000   PIT TICKS    *  SYSTEM VEIW *

CLASS 1(B)   = 10 %                *  USER VIEW    *
CLASS 1(B)   = 1000   PIT TICKS    *  SYSTEM VEIW *

CLASS 2(C)   = 50 %                *  USER VIEW    *
CLASS 2(C)   = 5000   PIT TICKS    *  SYSTEM VEIW *

CLASS 3(D)   = 20 %                *  USER VIEW    *
CLASS 3(D)   = 2000   PIT TICKS    *  SYSTEM VEIW *
```

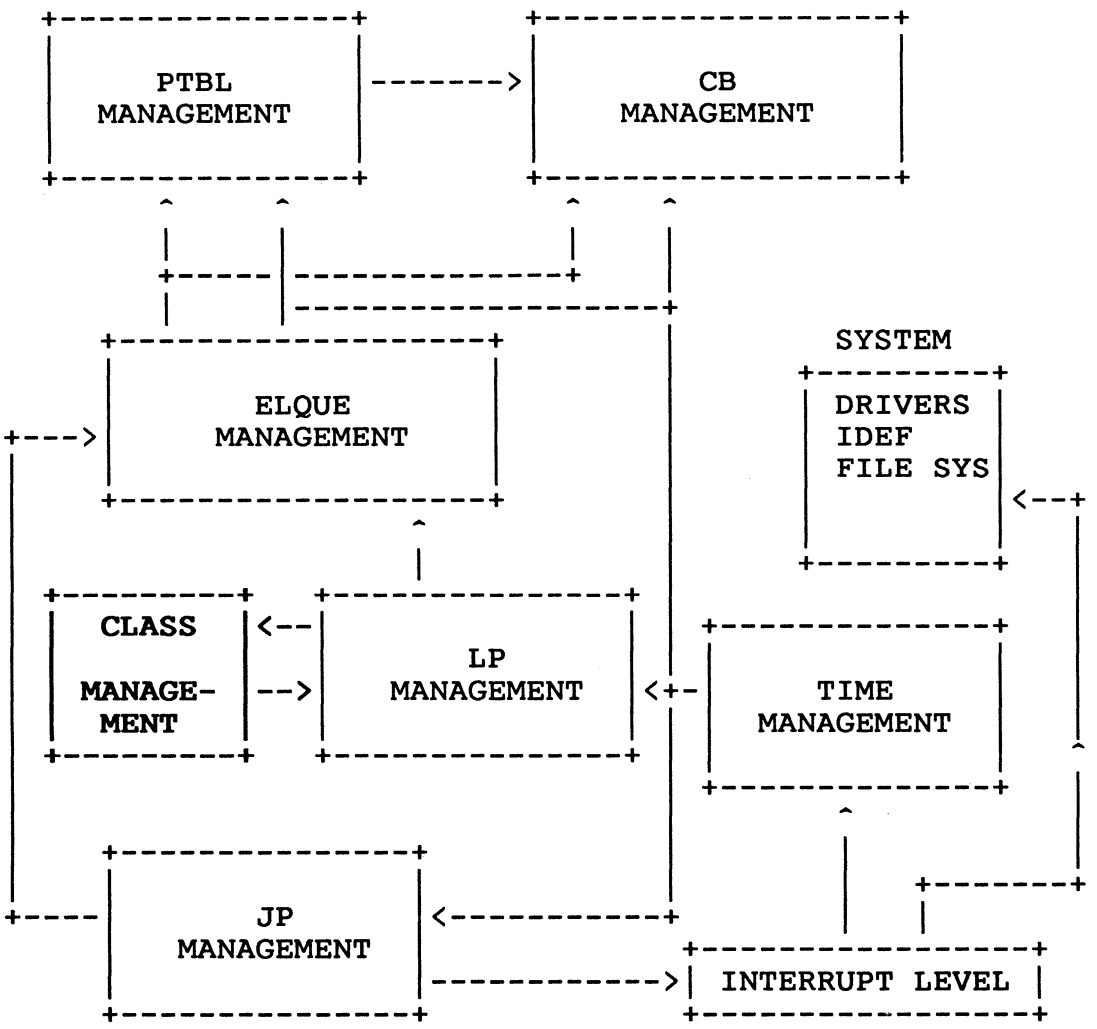
Each time a user or CB runs, the time used gets subtracted from the total number of ticks in the interval. When the interval counter reaches zero OR less than zero, the LPCB class scheduling area gets RESET with a new counter and class time limits for the LP. After a RESET occurs, then class scheduling is in a new interval.

The second system service performed by LP management is totals updating. This updating keeps an accounting of the amount of time used by certain entities on the LP. For example, after a ?OPEN call is made by a user the system creates a CB for the ?READ and it runs. After the CB runs the time used by this CB would be totaled into the mother-only counter, which is called LPTMPT.4. The reason for this is that the ?OPEN is a file system call, which forces the system call to be a mother-only call. Therefore, if this is the LP that has JPO attached to it, this LP gets credit for the time used.

Chapter 5
Class Management

Class Management is the part of Paths and Time that manages the major databases used to put new processes onto the proper LP. Class Management supports the LP by providing the management information needed. For example, without the class matrix (see below) the ?PROC code would not be able to decide which class a process was in and, therefore, would not be able to enforce class scheduling on that particular process. Class Management uses two types of objects: the Class Matrix (see CLASP Manual) and the CLass Control Block (CLCB).

This chapter is divided into two main sections, which deal with each major object in Class Management. The sections are Section 5.1, the Class Matrix and Section 5.2, the CLass Control Block (CLCB).



Class management only interracts with LP management to show existence of a class. This prevents time charges from going to a nonexistent class.

5.1 The Class Matrix

The class matrix is a 16-by-16 matrix that is accessed by the User Locality (y axis) and the Program Locality (x axis). In AOS/VS, the matrix is implemented as a one dimensional array of 400 (octal) consecutive words. There is one class matrix for the entire system. The class matrix exists at boot time. A class is defined in the matrix by having the class ID in a cell. Initially, the matrix contains all zeros, which is class zero (class A). The global **CLMATRIX** is the location at the beginning of the matrix.

Figure 5.1 shows a class matrix for a system. Note that this matrix is in the same format as shown in CLASP. Each letter in the matrix represents a class. The matrix normally has numeric representations of the classes instead of letters, but for this example letters are easier to read.

		Program Locality															
		0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
	0	A	B	C	D	E	F	A	A	A	A	A	A	A	A	A	A
	1	G	G	G	G	G	G	G	G	G	G	G	G	G	G	G	G
U	2	A	B	C	D	E	F	A	A	A	A	A	A	A	A	A	A
s	3	A	B	C	D	E	F	A	A	A	A	A	A	A	A	A	A
e	4	A	B	C	D	E	F	A	A	A	A	A	A	A	A	A	A
r	5	A	B	C	D	E	F	A	A	A	A	A	A	A	A	A	A
	6	A	B	C	D	E	F	A	A	A	A	A	A	A	A	A	A
L	7	A	B	C	D	E	F	A	A	A	A	A	A	A	A	A	A
o	8	A	B	C	D	E	F	A	A	A	A	A	A	A	A	A	A
c	9	A	B	C	D	E	F	A	A	A	A	A	A	A	A	A	A
a	10	A	B	C	D	E	F	A	A	A	A	A	A	A	A	A	A
l	11	A	B	C	D	E	F	A	A	A	A	A	A	A	A	A	A
i	12	A	B	C	D	E	F	A	A	A	A	A	A	A	A	A	A
t	13	A	B	C	D	E	F	A	A	A	A	A	A	A	A	A	A
y	14	A	B	C	D	E	F	A	A	A	A	A	A	A	A	A	A
	15	A	B	C	D	E	F	A	A	A	A	A	A	A	A	A	A

Figure 5.1 The Class Matrix

To get a process class, the system must find the cell in the matrix. The system does this by getting the offset in the matrix. To calculate the offset into the matrix, multiply the program locality max and the program locality then add the user locality. (See the formula below.)

```
    process_class = CLMATRIX + Offset;
Where:
    Offset = (prog_locality * prog_locality_max) +
                          user_locality
```

5.1.1 Basic Operations on the Class Matrix

There are two basic operations performed on the class matrix: Get Class Value and Set Class Value. These operations are represented in pseudocode below.

5.1.1.1 Get Class Value

This operation gets a value contained in a cell of the matrix. The inputs to this operation are the user and program localities. This function assumes that the JPLPLOCK is held. (See ELQUE management.) In the code this routine is called GCMATRIX.

```
/* $$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$ */
/*                      get_class_value                          */
/* This routine gets the class value from a cell                 */
/* in the class matrix.  User_loc and prog_loc are               */
/* the locality arguments passed to this routine.                */
/* This function returns the class in the cell.                  */
/* $$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$ */


get_class_value(user_loc,prog_loc)
  {
    max_loc = 020;   /* max_loc is a local variable   */
                     /* used as a multiplier into     */
                     /* the class matrix              */
/* ********************************************** */
/*                                                */
/*  Calculate the offset into the matrix.         */
/*                                                */
/* ********************************************** */
    offset = (prog_loc * max_loc) + user_loc;
    return(clmatrix[offset];
  }
```

5.1.1.2 Set Class Value

This operation sets an offset in the matrix with the value passed as an argument. The function assumes that the JPLPLOCK is held and that the arguments passed are valid. In the code, the name of this routine is SCMATRIX.

```
/* $$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$ */
/*                    set_class_value                            */
/* This routine puts the value of the class passed               */
/* into the Class Matrix at the location calculated by           */
/* using the localities passed.  The arguments                   */
/* are user_loc, prog_loc and class_id.                          */
/*                                                               */
/*  To change the value of a cell in the matrix                  */
/*  we must remove the old class from the matrix.                */
/*  This entails dealing with the CLCB.  The CLCB is             */
/*  defined in the CLCB section.                                 */
/*                                                               */
/* $$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$ */

set_class_value(user_loc,prog_loc,class_id)
  {
    max = 020;        /* max is a local variable   */
                      /* used as a multiplier into */
                      /* the class matrix          */
    class = get_class_value(user_loc,prog_loc);
/* ********************************************************** */
/*  If we come back on the error return from the              */
/*  find CLCB routine, then the system will panic with        */
/*  a 14617.  Taking the error return means the CLCB          */
/*  was not found.  This means that we are trying             */
/*  to get a class from the matrix that is not defined        */
/*  even though the class is in the matrix.                   */
/* ********************************************************** */
    clcb_ptr =  findcl(class); /* see CLCB section  */
    if (error_return)
         panic(14617);
    clcb_ptr.clmatn --; /* one less user */
```

```
/* ************************************************ */
/*  If we come back on the error return from the    */
/*  find CLCB routine, then panic the system with   */
/*  a 14617.  Taking the error return means the CLCB */
/*  was not found.  This means that we are trying   */
/*  to put a class into the matrix that is not      */
/*  defined, which would corrupt the matrix.        */
/* ************************************************ */
    clcb_ptr = findcl(class_id)
    if (error_return)
         panic(14617);
    clcb_ptr.clmatn ++;     /* one more user */
                          /* see CLCB section */
    offset = (prog_loc * max) + user_loc;
    clmatrix[offset] = classid;
    return();
    }  /* set class value  */
```

5.1.2 Class Matrix User Services

The user service that works with the Class Matrix is the ?CMATRIX system call (internally called CMATR.P). This system call redefines the contents of the class matrix. The pseudocode below shows how this function works.

```
/* $$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$ */
/*                          cmatrix.p                            */
/* This user service allows the user to get values               */
/* from or redefine class matrix.  If the get option             */
/* is used, then the matrix is not touched.  If the              */
/* set option is used, then the call changes all the             */
/* cells the user specifies.                                     */
/*                                                               */
/* $$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$ */

cmatrix.p( )
  {
    set_up_trap_handler(cb);
    packet = CC.W->catcb->tac2;
                          /* get packet from tasks AC 2*/
    num_of_cells = packet->count;

/* ************************************************************* */
/* The user matrix is an array of double words.                  */
/* This will be treated as a high-level array instead            */
/* of offsetting into the array by twos.                         */
/*                                                               */
/* ************************************************************* */

    user_matrix_ptr = packet->matrix;
    get_set_code = packet->getset;
```

```
/* ************************************************ */
/* Test to see which operation is to be done.       */
/* The get/set code in the packet should be a 0 for */
/* get or a 1 for set.                              */
/*                                                  */
/* ************************************************ */

    switch (get_set_code)
      {
    case 0:        /* get operation  */

/* ************************************************ */
/*  Loop through the user packet to validate the user */
/*  matrix information.                             */
/* ************************************************ */
         for (i=0;i<=num_of_cells;i++)
           {
/* ************************************************ */
/*                                                  */
/* Since the matrix supplied by the user is a group */
/* of double words, the function left_word gets the */
/* left word of the double-word entry.              */
/*                                                  */
/*  If the user locality is invalid then return.    */
/*  A valid locality must be an integer between     */
/*  0 and 15.  Therefore, "locality_valid" is as    */
/*  follows:                                        */
/*                                                  */
/*      0 <= locality <= 15                         */
/*                                                  */
/* ************************************************ */

             localities = left_word(matrix_ptr+i);
             user_locality = left_byte(localities);
             if (!     (user_locality >=0) & (user_locality
                       <= 15))
                  return(illegal_locality);
/* ************************************************ */
/*                                                  */
/*  If the program locality is invalid then return. */
/*  A valid locality must be an integer between     */
/*  0 and 15.  Therefore, "locality_valid" is as    */
/*  follows:                                        */
/*                                                  */
/*      0 <= locality <= 15                         */
/*                                                  */
/* ************************************************ */

             prog_locality = right_byte(localities);
             if (!     (prog_locality >=0) & (prog_locality
                       <= 15))
                  return(illegal_locality);
          }   /*  FOR LOOP  */
```

```
/* ************************************************ */
/*  Loop through the user matrix and get the        */
/*  associated values out of the class matrix.      */
/*                                                  */
/* ************************************************ */
        get_lock(read);  /* get global lock */
        for (i=0;i<=num_of_cells;i++)
          {
             localities = left_word(matrix_ptr+i);
             user_locality = left_byte(localities);
             prog_locality = right_byte(localities);
             user_matrix_ptr[i].right_word =
                       get_matrix_value(user,prog);
           } /*  FOR LOOP  */
        release_lock();
        clear_trap_handler(cb);
        return();

/* ************************************************ */
/*                                                  */
/* Now we are doing the set operation.              */
/*                                                  */
/* If the user does not have SYSPRV (system manager */
/* privilege) then return to the caller.            */
/*                                                  */
/* ************************************************ */

    case 1:

        if (!(syspriv))
             return(caller_not_privileged

/* ************************************************ */
/*  Loop through the user packet to validate the user */
/*  matrix information, including the class info.   */
/* ************************************************ */
        for (i=0;i<=num_of_cells;i++)
          {
/* ************************************************ */
/*                                                  */
/* Since the matrix supplied by the user is a group */
/* of double words, the function left_word gets the */
/* left word of the double-word entry.  The function */
/* right word gets the right word of the entry.     */
/* ************************************************ */

             localities = left_word(matrix_ptr+i);
             class  =  right_word(matrix_ptr+i);
```

```
/* ************************************************ */
/*                                                  */
/*  If the user locality is invalid then return.    */
/*  A valid locality must be an integer between     */
/*  0 and 15.   Therefore, "locality_valid" is as   */
/*  follows:                                        */
/*                                                  */
/*      0 <= locality <= 15                         */
/*                                                  */
/* ************************************************ */

             user_locality = left_byte(localities);
             if (!     (user_locality >=0) & (user_locality
                       <= 15))
                  return(illegal_locality);
/* ************************************************ */
/*                                                  */
/*  If the program locality is invalid then return. */
/*  A valid locality must be an integer between     */
/*  0 and 15.  Therefore, "locality_valid" is as    */
/*  follows:                                        */
/*                                                  */
/*      0 <= locality <= 15                         */
/*                                                  */
/* ************************************************ */

             prog_locality = right_byte(localities);
             if (!     (prog_locality >=0) & (prog_locality
                       <= 15))
                  return(illegal_locality);
/* ************************************************ */
/*                                                  */
/*  If the class value is invalid then return.      */
/*  A valid class must be an integer between        */
/*  0 and 15.  Therefore, "class_valid" is as       */
/*  follows:                                        */
/*                                                  */
/*      0 <=   class  <= 15                         */
/*                                                  */
/* ************************************************ */

             if (!     (class >=0) & (class <= 15))
                  return(illegal_class_id);
          } /* for Loop  */
```

```
/* ************************************************ */
/*  Get the JPLP lock for change.                   */
/*                                                  */
/* ************************************************ */

        get_lock(change);

/* ************************************************ */
/*  Now loop to validate the existence of the class.*/
/*                                                  */
/* ************************************************ */
        for (i=0;i<=num_of_cells;i++)
          {
             class = right_word(matrix_ptr+i);
             if(!(bit(clmap,class) == 1))
                  return (invalid_locality);
          }/* for Loop  */

/* ************************************************ */
/* Now loop through the array and change the class  */
/* matrix.                                          */
/*                                                  */
/* ************************************************ */
        for (i=0;i<=num_of_cells;i++)
          {
             localities = left_word(matrix_ptr+i);
             class = right_word(matrix_ptr+i);
             user = left_byte(localities);
             prog = right_byte(localities);
             set_matrix_value(user,prog,class);
          }/* for Loop  */
        release_lock();
        clear_trap_handler();
        return();
    } /*switch statement */
  }/* function  */
```

5.2 The Class Control Block (CLCB)

The second major object used in Class Management is the CLCB. The CLCB contains information useful in managing the classes in the system. There is one CLCB for each defined class on the system. Figure 5.2 below contains the CLCB offsets.

THE CLCB

OFFSET		SUMMARY EXPLANATION
CLLPN ->	0	Number of LPs using this class
CLPIDN ->	1	The number of pids with this class
CLMATN ->	2	The number of matrix cell w/ this class
CLNAME ->	3	The name of the class

CLLEN = 24 (octal) ** length of CLCB

Figure 5.2 The CLCB

5.2.1 CLCB Offset Explanations

Offset CLLPN is the number of LPs attached to this class. This offset is not currently used.

Offset CLPIDN is the total number of PIDs currently using this class. This offset is not currently used.

CLMATN is the number of cells this class has in the class matrix. This offset is incremented when putting this class into the class matrix and is decremented when removed. This offset is also used when deciding to delete a class from Class Management. If this offset is greater than zero, then the class cannot be deleted from Class Management.

CLNAME uses 20 (octal) words for the name of the class. This offset is not currently used because AOS/VS does not want user supplied names in ring zero.

5.2.2 The CLCB Globals

There are two Globals used in class management: CL.W and CLAP. There is one copy of these databases on the system.

CL.W is a block of 16. double words containing the pointers to the CLCBs. When a class is created the pointer to its CLCB is placed in CL.W. When a class is removed from the system, the corresponding address is set to zero. CL0.W, which is the address of class 0 (class A), is put on the table at assembly time. CL0.W is always in CL.W. This table of CLCBs is used because the CLCBs can be anywhere in memory. This method of managing the CLCBs saves on system memory use because the system releases the memory for the CLCB when it is deleted. Figure 5.3 shows the relationship between CL.W and the CLCBs.

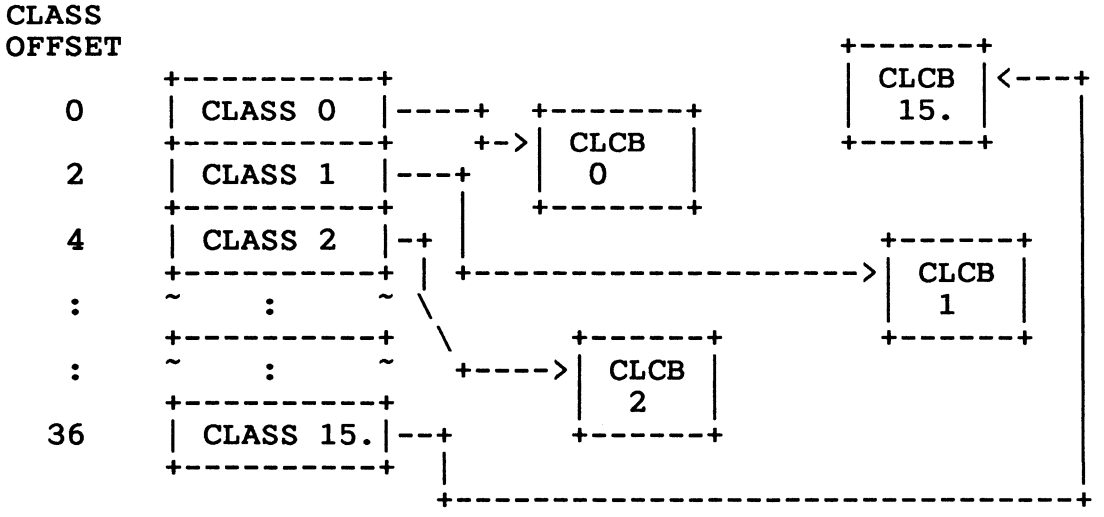


Figure 5.3

The other global database used with CLCBs is CLMAP. This is a one-word bit map of the classes that exist in the system. This field is used to show existence of a class on the system. If the CLCB exists, then the corresponding bit in this word will be set. This field is updated when a class is added to or removed from the system. At assembly time this field has the zero bit set for class 0 (class A). The reason for this is that class zero always exists.

5.2.3 Basic Operations on the CLCB, CL.W, and CMAP.

There are five basic operations that work on the CLCB, CL.W and CMAP. The operations are create a CLCB, delete a CLCB, find a CLCB, add a CLCB, and remove a CLCB from CL.W.

5.2.3.1 Create a CLCB

This operation calls memory management to get memory for the CLCB. If the memory request is successful, then the routine returns the address of the new CLCB.

```
/* $$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$ */
/*                        create_clcb                             */
/* This routine calls the memory manager to allocate  */
/* memory for the CLCB to be created.  If the memory  */
/* request fails then return the error.  If no errors */
/* then return the address of the new CLCB.           */
/*                                                    */
/* $$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$ */

create_clcb( )
  {
    clcb_address = GSMEM(cllen,nowait); /* see memory */
                              /* management */
                              /*cllen= 024 */
    if (error_return)
        return(no_mem);
    return(clcb_address);
  } /* create a CLCB */
```

5.2.3.2 Delete a CLCB

This operation first checks if it is legal to delete the class. If it is legal to delete the class then return. If not, take the error return.

```
/* $$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$ */
/*                      delete_a_clcb                          */
/* This routine checks if it is legal to delete a              */
/* class by looking at the offset in the CLCB called           */
/* CLMATN.  If the value is greater than zero, then            */
/* the class cannot be deleted.  The caller is also            */
/* not allowed to delete class zero.  The argument             */
/* passed to this routine is the class id.                     */
/* $$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$ */

delete_a_clcb(class_id)
   {
    if (class_id = 0)
        return(can_not_del_class_zero);
    if (clcb.cmatn > 0)
        return(class_in_matrix);
    return();         /* it is ok to delete the class */
   } /* delete_a_class */
```

5.2.3.3 Find a CLCB

This operation tries to find an entry in CL.W offset by the class id to see if the class exists. A class exists if the table entry is non-zero. The entry point for this operation is called FINDCL.

```
/* $$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$ */
/*                        findcl                               */
/* This routine looks at the array CL.W to see if the          */
/* entry in the array is non-zero.  If so then the             */
/* class exists.  The argument passed to this routine          */
/* is the class id.                                            */
/* $$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$ */

findcl(class_id)
  {
    if (cl.w[class_id*2] = 0)
        return(not_found);
    else
        return();
  }
```

5.2.3.4 Add a CLCB

This operation puts the address of the CLCB into the CL.W table and is an entry point called ADDCLASS.

```
/* $$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$ */
/*                    addclass                                    */
/* This operation adds a CLCB to CL.W.  The table                 */
/* is a double-word table.  This routine also sets                */
/* the bit in the class bitmap called CLMAP.                      */
/* The arguments passed to this routine are the class             */
/* id and the address of the new CLCB.                            */
/* $$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$ */

addclass(class_id,clcb)
   {
    setbit(clmap,class_id);

/* ************************************************************ */
/*                                                              */
/* If there is an entry for this offset in CL.W then            */
/* we're trying to add an already existing class.  If           */
/* this is true then panic with a 14614.  If the entry          */
/* is 0 then put the CLCB address in the table.                 */
/*                                                              */
/* ************************************************************ */

    if (cl.w[class_id*2] == 0)
        cl.w[class_id*2] = clcb;
    else
        panic(14614);
    return();
  } /* addclass  */
```

5.2.3.5 Remove a Class

This operation removes a class from CL.W and CLMAP. The routine first removes the class from the bit map (CLMAP). The routine then tries to find the class. If the class does not exist, then the routine returns an error. If found, remove the CLCB from CL.W, release the memory and return. In AOS/VS this routine has an entry point called DELCLASS.

```
/* $$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$ */
/*                          delclass                               */
/* This routine deletes an existing CLCB from the                  */
/* CL.W array.  The argument passed to this function               */
/* is the class id.                                                */
/*                                                                 */
/* $$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$ */

delclass(class_id)
   {
    clearbit(clmap,class);
    findcl(class_id);
    if (error_return)
        return();
    else
       {
        cl.w[class_id*2] = 0;
        rsmem(cllen);  /* see memory management  */
      }
    return();
  }
```

5.3 User Services

The paths that use the elementary functions in Class Management are the class system call routines. These user services are ?CLASS, ?PCLASS, ?CLSCHD, and ?CLSTAT. The user services are explained in the pseudocode below.

5.3.1 ?CLASS

This system call gets or sets up the classes for Class Management. (See System Call Dictionary.) The entry point for this function is called CLASS.P.

```
/* $$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$ */
/*                   class.p                          */
/* This function gets or sets the classes in the      */
/* class environment.                                 */
/*                                                    */
/* $$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$ */

class.p()
   {
    set_up_trap_handler(mptrap);
    packet = cc.w->catcb.w->*tac2.w;
                 /* get packet address */
    user_cl_map = packet.clmap;
    user_getset = packet.getset;

/* ************************************************** */
/*                                                    */
/* If "get" then store the system class bitmap into   */
/* the user packet and return.                        */
/*                                                    */
/* ************************************************** */

    switch (user_getset)
       {
        case 0:

        set_up_trap_handler(mptrap.r)/*read lock */
        get_lock(read);
        packet.clmap = clmap;
        release_lock();
        clear_trap_handler();
        return();
```

```
/* ************************************************ */
/*                                                  */
/* The user wishes to set up some classes.  Check   */
/* the system privilege.  If the user has the       */
/* privilege then continue with the call.  If not then*/
/* return the "user not privileged for this action" */
/* error.                                           */
/* ************************************************ */

    case 1:
    clear_trap_handler();
    if (!(sysprv()))
        return(caller_not_priveliged);

/* ************************************************ */
/*                                                  */
/* Clear the stack called new_clcb.                 */
/*                                                  */
/* ************************************************ */

    for (i=0;i<16;i++)
        new_clcb[i] = 0;
    get_lock(change);

/* ************************************************ */
/*                                                  */
/* Now loop through the users supplied classes for  */
/* validity.  This loop compares the user bitmap    */
/* to the system class bitmap to see what should be */
/* done.  "i" is the offset into the bitmap.        */
/*    clmap[i]   user_clmap[i]      action          */
/*                                                  */
/*       0           0             no action        */
/*       0           1             add a class      */
/*       1           0             delete a class   */
/*       1           1             no action        */
/* ************************************************ */

    for (i=0;i<16;i++)
      {
        user_bit = bit(user_clmap,i);
        sys_bit  = bit(clmap,i);
        if ((user_bit == 1) & (sys_bit == 0))
            {
            new_clcb[i] = create_clcb(i);
            if (error_rtn)
                return(cannot_get_memory);
            }
        else
            if ((user_bit == 0) & (sys_bit == 1)
                if (!(delete_class (i)))
                    return(class_can_not_be_deleted);
      }
```

```
/* ************************************************ */
/* Now that the requests are valid and the memory for */
/* new classes have been allocated, attach or remove  */
/* the classes from the class globals.                */
/* Now loop again through the user-supplied classes   */
/* to do the class work.  This loop compares the user */
/* bitmap to the system class bitmap to see what      */
/* should be done.  "i" is the offset in the bitmap.  */
/*    clmap[i]   user_clmap[i]      action            */
/*                                                    */
/*       0           0             no action        */
/*       0           1             add a class      */
/*       1           0             delete a class   */
/*       1           1             no action        */
/* ************************************************ */

    for (i=0;i<16;i++)
      {
        user_bit = bit(user_clmap,i);
        sys_bit  = bit(clmap,i);
        if ((user_bit == 1) & (sys_bit == 0))
            addclass(i,new_clcb[i]);
        else
            if ((user_bit == 0) & (sys_bit == 1)
                delclass(i);
      }
    release_lock();
    return();
  } /* ?CLASS  */
```

5.3.2 ?PCLASS

This system call is used to find a process' user and program locality and the class to which a process belongs. (See the System Call Dictionary.) The entry point for this user service is called PCALL.P.

```
/* $$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$ */
/*                    pclass.p                                */
/* This routine returns the locality and class                */
/* information for a process to the caller.                   */
/*                                                            */
/* $$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$ */

pclass.p()
   {
    set_up_trap_handler(mptrap);
    packet = cc.w->*catcb.tac2;
    pid_pname_code = packet.pin_pname_code;
    stack_pid = packet.pid;
    pname_len = packet.pname_len;
    bp_pname = packet.bp_pname;
    clear_the_trap_handler();

/* ************************************************** */
/*  If the user supplied a process name, then save    */
/*  the name.                                         */
/*                                                    */
/* ************************************************** */

    if (bp_pname != 0)
        BLM(pname,packet.pname,pname_len);
    clear_trap_handler();
```

```
/* ************************************************ */
/* If pid code = 0 then the user is called by pid.    */
/* If the caller supplied a pid check to see if       */
/* the pid is a -1 (refer to own pid), get the PTBL   */
/* for callers pid. If its a real pid get the PTBL for*/
/* that pid.  If the user called with a process name, */
/* resolve the process name to a pid and get the      */
/* PTBL address.                                      */
/* ************************************************ */

    if(pid_pname_code == 0)
        if (stack_pid == -1)
            {
            ptbl = cc.w->cptad.w;
            stack_pid = ptbl.pid;
            {
        else
            {}
    else
        {
        stack_pid = nmtid.p(pname); /* convert a process */
                                    /* to a pid          */
        if (error_rtn)
            return(name_not_found);
        }

/* ************************************************ */
/*                                                  */
/* A "pid_valid" is a pid that is between 1 and       */
/* genpid and bit corresponding to the pid in pidbt.w */
/* (See Process Management) is set.  If not valid then*/
/* return the error not in hierarchy.                 */
/*                                                  */
/* ************************************************ */

    get_plock(ptran,pids1k.w);/* lock word for pidtb.w   */
                              /* and pidbt.w             */
    if  (!(stack_pid >= 0) & (stack_pid <= GENPID) &
        (bit(pidbt.w,stack_pid) ==1))
        return(process_not_in_hierarchy);
    ptbl = pidtb.w(stack_pid);
    release_plock(pids1k.w);
    get_plock(ptran,ptbl);
    packet.u_locality = ptbl.ulocal;
    packet.p_locality = ptbl.plocal;
    packet.class_id = ptbl.class_id;
    release_plock();
    return();
  } /* ?pclass */
```

5.3.3 ?CLSCHD

The ?CLSCHED system call allows the user to either query (get) or change (set) the mode of class scheduling for a specified LP. There are three modes to class scheduling:
1) Disabled-class scheduling is not being used;
2) Enabled-class scheduling is being enforced;
3) Accumulate-class scheduling is not being enforced but class statistics are being kept.

The LPCB status word is used to make decisions in this user service: specifically, the two class scheduling bits LPOFF and LPSTAT. When these bits are set, then the class scheduling function is not being done.

The following truth table shows how the decisions are made.

LPOFF	LPACC	Class Mode
1	1	Disable mode
1	0	Accumulating stats
0	0	Enable mode
0	1	Illegal condition

The user service uses the entry point CLSCHD.P. The pseudocode below shows how this operation works.

```
/* ************************************************ */
/*                      clschd.p                    */
/*  This operation gets or sets the class scheduling */
/*  mode for a specified LP.                        */
/*                                                  */
/* ************************************************ */

clschd.p()
   {
    set_up_trap_handler(mptrap);
    packet = cc.w->catcb.w->tac2.w;
    lp_map = packet.lp_map;
    class_code = packet.class_code;
    getset = packet.getset;
    clear_trap_handler;
```

```
/* ************************************************ */
/*                                                  */
/* If the getset code is 0 then the caller wishes to  */
/* get information about class scheduling.          */
/*                                                  */
/* ************************************************ */

    switch (getset)
      {
        case 0:

        get_lock(read);

/* ************************************************ */
/*                                                  */
/* We must find out what information the caller wants.*/
/* If the caller wants to find all the LPs in       */
/* accumulate mode, then the class_code will equal 2. */
/* Otherwise, the caller wants to know all the LPs    */
/* that are in enable mode.                         */
/*                                                  */
/* ************************************************ */

        if (class_code == 2)
            sched_bit_map = clacc.w; /* see LP management*/
        else
            sched_bit_map = clschd.w;/* see LP management*/
        release_lock();
        set_up_trap_handler(mptrap);
        packet.sched_bit_map = sched_bit_map;
        clear_trap_handler();
        return();
/* ************************************************ */
/*                                                  */
/* The caller wants to set the class scheduling mode. */
/*                                                  */
/* ************************************************ */

    case 1:

    if (!(syspriv()))
        return(not_priv_this_action);
    get_lock(change);
```

```
/* ************************************************ */
/* If the user wants to set class scheduling to enable*/
/* or disable mode, then set the mode for the LPs     */
/* specified by the user.  ?LPID_MAX is a global.     */
/* Its value is 17 (octal).                           */
/* ************************************************ */

    if (class_code == 1)
      {
        for (i=0;i<=?lpid_max;i++)
          {

/* ************************************************ */
/*                                                  */
/*  If the LP exists then continue.  Existence is   */
/*  defined as the corresponding bit to the LP in the */
/*  global LPMAP is set.                            */
/*                                                  */
/* ************************************************ */

            if (bit(lpmap,i) == 1)
              {
                user_bit = bit(lp_map,i);
                sys_bit  = bit(clschd.w,i);

/* ************************************************ */
/* The following truth table shows what decisions     */
/* are to be made.  "i" is used to represent the      */
/* current class being worked with.                   */
/*                                                    */
/*  lp_map[i]       CLSCHD.W[i]        Action          */
/*     1               1              do nothing      */
/*     1               0              set enable      */
/*     0               1              set disable     */
/*     0               0              do nothing      */
/*                                                    */
/* ************************************************** */
```

```
/* ************************************************ */
/*                                                  */
/*  This situation causes the class scheduling to be  */
/*  enabled.  This automatically turns on accumulation*/
/*  mode.                                           */
/* ************************************************ */

                if ((user_bit = 1) & (sys_bit = 0))
                  {
                    lpcb = find_lp(i);
                    if (error_return)
                        panic(14612);
                    lock_lpcb(lpcb);
                    reset(lpcb);
                    bit(lpcb.lpstat,lpoff) = 0;
                    bit(lpcb.lpstat,lpacc) = 0;
                    unlock_lpcb(lpcb);
                    bit(clschd.w,i) = 1;
                    bit(clacc.w,i)  = 1;
                  } /* if */


/* ************************************************ */
/*                                                  */
/* If the user_bit is 0 and the system_bit is 1 then  */
/* the user wishes to disable scheduling for that LP. */
/* This also turns off the accumulate mode.         */
/* ************************************************ */

                if ((user_bit = 0) & (sys_bit = 1))
                  {
                    lpcb = find_lp(i);
                    if (error_return)
                        panic(14612);
                    lock_lpcb(lpcb);
                    reset(lpcb);
                    bit(lpcb.lpstat,lpoff) = 1;
                    bit(lpcb.lpstat,lpacc) = 1;
                    unlock_lpcb(lpcb);
                    bit(clschd.w,i) = 0;
                    bit(clacc.w,i)  = 0;
                  } /* if /*
              } /* if bit (lpmap,i) == 1) */
          } /* for loop  */
        release_lock();
        return();
        } /* if class_code == 1 */
    else
       {
```

```
/* ************************************************ */
/*                                                  */
/*  If not enable or disable mode then set accumulate */
/*  mode.  Loop through the user's bitmap and set the */
/*  requested LPs in accumulate mode.               */
/* ************************************************ */

    for(i=0;i<?lp_max;i++)
      {
        if (bit(lpmap,i) = 1)
          {
/* ************************************************ */
/*                                                  */
/* The caller can request accumulate mode be turned  */
/* on or off.   CLSMAP.W is a global accumulate mode */
/* bitmap.                                          */
/*  lp_map [i]      CLSMAP.W         Action         */
/*                                                  */
/*    1                1             none           */
/*    1                0             set accumulate */
/*    0                1             clear accumulate */
/*    0                0             none           */
/*                                                  */
/* ************************************************ */

            user_bit = bit(lp_map,i);
            sys_bit  = bit(clsmap.w,i);
/* ************************************************ */
/*                                                  */
/*  This is the case where accumulate mode is turned */
/*  on.                                             */
/*                                                  */
/* ************************************************ */

            if ((user_bit == 1) & (sys_bit == 0))
              {
                lpcb = find_lp(i);
                if (error_rtn)
                    panic(14612);
                reset();
                bit(lpcb.lpstat,lpacc) = 0;
              } /* if */
```

```
/* ************************************************ */
/*                                                  */
/*  This is the case where accumulate mode is turned  */
/*  off.                                            */
/*                                                  */
/* ************************************************ */

            if ((user_bit == 0) & (sys_bit == 1))
              {
                lpcb = find_lp(i);
                if (error_rtn)
                    panic(14612);
                reset();
                bit(lpcb.lpstat,lpacc) = 1;
              } /* if */
          } /* if */
      } /* for loop */
   }/* else */
    release_lock();
    return();
}/* ?CLSCHD  */
```

5.3.4 ?CLSTAT

?CLSTAT is used to find information about a specific LP in the system. (See System Call Dictionary.) The user has the option of getting just LP information or getting the JP information. The entry point for this user service is called CLSTAT.P.

```
/* $$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$ */
/*                          clstat.p                              */
/* This routine provides the user with LP statistical            */
/* information.                                                  */
/*                                                               */
/* $$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$ */

clstat.p()
  {
/* ********************************************************** */
/*                                                            */
/*  Zero out the temporary areas that will be used            */
/*  in this routine.  The temporaries are called:             */
/*  stack_lpcb     - looks like the LPCB stats area.          */
/*  stack_ppcb     - looks like the PPCB stats area.          */
/*  stack_sub_pkt  - looks like the sub packet.               */
/*  Do this by BLMing 203 words of zeros.   In this           */
/*  routine a routine called fill_mem will put zeros          */
/*  onto the stack.                                           */
/* ********************************************************** */

    fill_mem(&stack_lpcb,0203,0);
```

```
/* ************************************************ */
/*                                                  */
/* Get the necessary information from the packet to */
/* make decisions and to find the LP the user wishes */
/* to look at.                                      */
/*                                                  */
/* ************************************************ */

    set_up_trap_handler(mptrap);
    packet = cc.w->catcb->tac2;
    function_code = packet->function_code;
    sub_address = packet->sub_packet_address;
    lp_id = packet->lp_id;
    clear_trap_handler;
/* ************************************************ */
/*                                                  */
/*  If the LP id is invalid then return an error.   */
/*  lp_valid is defined as follows:                 */
/*                                                  */
/*            0 <= lp_id <= 15.                     */
/*                                                  */
/* ************************************************ */

    if (!(lp_valid >= 0) & (lpid <= 15.))
        return(invalid_lp_id);
    get_lock(read);
    lpcb = find_lp(lp_id);
    if (error_return)
        return(lp_does_not_exist);
    num_of_jps = lpcb->lpjpcnt;
    jps_bit_map = lpcb->lpjpmp.w;

/* ************************************************ */
/*                                                  */
/*   Blam the sums area from the LPCB to a temporary */
/*   area called temp.  This area is 132(octal) words.*/
/*                                                  */
/* ************************************************ */

    lock_lpcb(lpcb);
    BLM(lpcb->lpsum.4,stack_lpcb->lpsum,0132);
    unlock_lpcb(lpcb);
```

```
/* ************************************************ */
/*                                                  */
/*  If the user wishes to get JP information, the   */
/*  "function_code" will be a 2.  If not the code will*/
/*  be a 1.  If neither, then take the error "illegal */
/*  function code".                                 */
/*                                                  */
/*                                                  */
/* ************************************************ */

    switch (function_code)
      {

    case 1:
        break;

/* ************************************************ */
/*                                                  */
/* If there are JPs attached to this LP, then loop  */
/* through the JP list and accumulate the JP        */
/* statistics for that LP and add them to an        */
/* internal version of the subpacket called         */
/* stack_sub_pkt.  The PPCB stats are blam'ed into a */
/* temporary structure that looks like the PPCB stats */
/* area called stack_ppcb.                          */
/*                                                  */
/* ************************************************ */

    case 2:
        if (num_of_jps != 0)
          {
            for (i=0;i<16;i++)
              {
                jp_bit = bit(jp_bit_map,i);
                if(jp_bit == 1)
                  {
                    ppcb = findjp(i);
                    if(error_return)
                        panic(14621);
                    BLM(ppcb->cpsys.4,stack_ppcb->sys,34);
                    stack_sub_pkt->sys += stack_ppcb->sys;
                    stack_sub_pkt->cmt += stack_ppcb->cmt;
                    stack_sub_pkt->smt += jptemt.smt;
                    stack_sub_pkt->dmt += stack_ppcb->dmt;
                    stack_sub_pkt->idl += stack_ppcb->idl;
                    stack_sub_pkt->int += stack_ppcb->int;
                    stack_sub_pkt->resi += stack_ppcb->resi;
                  }/* if */
              } /* for  */
          } /*  if  */
    break;
```

```
    default:
        return(illegal_function_code);
/* ********************************************** */
/*                                                */
/*  Put the information into the user packet.     */
/*                                                */
/*                                                */
/* ********************************************** */

    release_lock();
    set_up_trap_handler(mptrap);
    packet->intervals = stack_lpcb->tint;
    packet->total = stack_lpcb->use;
    packet->user  = stack_lpcb->tpt;
    packet->user_mom = stack_lpcb->stmpt;
    BLM(stack_lpcb->stsum,packet->class,100);
    if(sub_address != 0)
        BLM(stack_sub_pkt,&sub_address,50);
    clear_trap_handler();
    return();
  }
```

Chapter 6
Job Processor Management

6.1 Introduction

6.1.1 Purpose

The purpose of this chapter is to describe the Job Processor (JP) and its databases.

6.1.2 Overview

This chapter is divided into four main sections: "Real JP," the PPCB, user services, and system services.

- "Real JP" is the physical entity known as a CPU in the system. There may be more than one JP in the system. In fact, the scheduler is designed to handle more than one CPU as the default case.

- PPCB is the internal database that represents the JP. Since there can be more than one JP in the system, there must be one PPCB for each JP. There are different code paths that access this database and, therefore, there must be some synchronization to allow the system to modify the PPCB safely.

- User Services are the JP system calls such as ?JPINIT, ?JPREL, and ?JPMOV. These system calls allow the user (if they have the privilege) to manage the physical processors on the system.

- System Services are the services that JP management provides to the rest of the Paths and Time environment. An example of a system service is keeping a count of the amount of time the system spends at interrupt level.

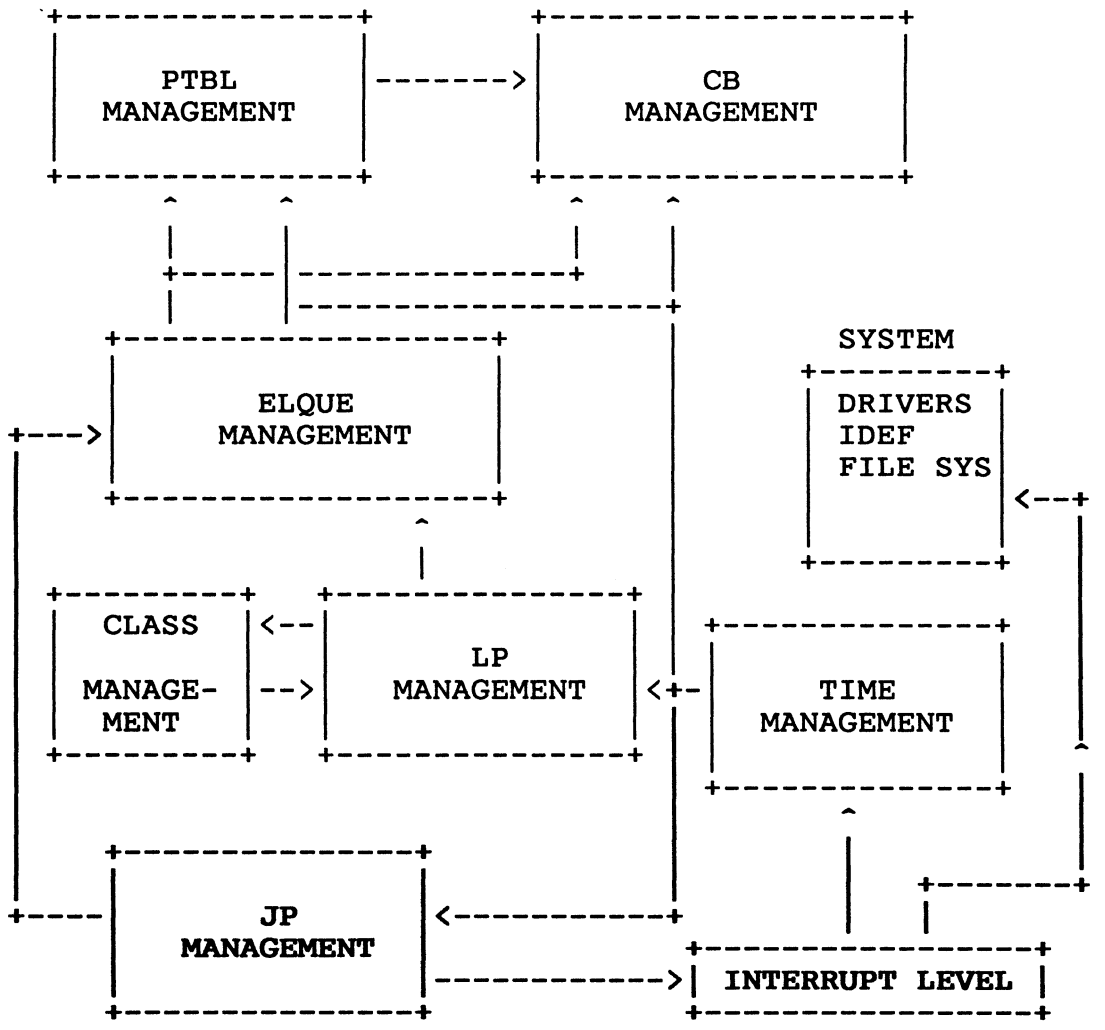


Figure 6.1

JP management supplies services to both time and ELQUE management in the form of code. The JP allows ELQUE and Time management to run code. JP management receives the timing data from time management, which it uses for its counters in the JPs PPCB.

6.2 The "REAL JP"

6.2.1 Introduction

This section discusses the physical entity known as the JP (the CPU). This section will describe the major state of a processor, the JP instructions used for JP management, and running operations on this processor.

The two major states of a JP are Halted or Running:

- When a job processor is halted, it does not exist as far a AOS/VS is concerned.
- A JP is considered Running when the microcode has been loaded, the PPCB is created, and a starting address is given to the processor and the processor is started.

The microcode loaded for a new JP can be either loaded from a user specified file, the file of the CPU type being started, or the existing microcode. The user can specify a microcode file for the JP to use, but the file must be a compatible microcode file for that JP. The user can specify to use existing microcode and the current microcode will be used. This option implies that the JP was previously initialized and released. The last option will force the system to find a microcode file for that type of processor. After these options are satisfied the microcode gets loaded for that processor (JPLCS instruction).

To manage a JP, AOS/VS must initialize and update some databases before a JP can exist. The database that must be created is the PPCB for this JP. Databases that must be updated for the LPCB are the JP count (LPJPCNT) and the JP bit map (LPJPCNT.W) for the system and the CP.W table of JPs.

The JP is given a starting address of JPSWART.P. Memory management is called from here to set up the JPs memory environment.

6.2.2 JP Instructions

In JP management, several privileged JP instructions are used: JPSTATUS, JPLCS, JPSTART, JPID, CINTR, IMODE, and JPSTOP. These instructions can only be called from ring 0. In fact, on many machines (e.g., ECLIPSE MV/8000), the instructions are not even defined.

The JPSTATUS instruction is used to find the current status of a processor. For example, when initializing a CPU a JPSTATUS instruction is used to find if the processor, a) exists and b) has returned a correct status. The instruction can take an error (or not skip) if the processor does not respond. If the instruction does skip the next instruction, then the JP did respond. The status bit "CPU IS OK" is returned in the status word if the CPU is running properly.

The JP Load Control Store (JPLCS) is used when loading microcode into the "control store" of the machine. This instruction is used at microcode load time only.

The JPSTART instruction is used to start a processor executing code at a specific address. At JP start time the JPSTART is issued with the address of JPSWART.P.

The JPID instruction is used to identify the JP a path is currently running on. This instruction is used during histogramming updates to keep the caller from cross interrupting itself.

The CINTR instruction is used to cross interrupt another processor. For example, this instruction is used during the RTC interrupt when histogramming to tell other processors to do histogramming. Currently this is the only cross interrupt defined in AOS/VS.

IMODE is the instruction that defines which processor will get interrupted. There are two modes to this instruction: 0 - dedicated mode and 1 - device dedicated mode. Mode 0 allows only one processor to be interrupted by the IOCs. Mode 1 allows the devices to interrupt a particular processor. AOS/VS only uses mode 0, therefore only the mother processor can receive interrupts. This instruction is used at Device Control Table (DCT) initialization.

The JPSTOP instruction requests the processor to stop running. The JPSTAT instruction can be used to find out when the processor actually stops running. This is used at JP release time.

6.2.3 JP Running Operations

The JP basically runs code. The JP does not care what the code is because it allows ELQUE management to either run or allow an element's code to run. The JP continues running code at what is called base level until one of two events occurs: 1 - an interrupt occurs or 2 - a fault occurs.

6.2.3.1 Interrupts

When an interrupt occurs the JP saves the state of the entity it was running and services the interrupt. Saving the state of the entity at interrupt time is done to the wide system interrupt stack. In AOS/VS this stack is called SS. To service the interrupt the processor jumps to the contents of location 1. Location 1 contains the address of the interrupt service routine. The interrupt service routine does a state save and dispatches control through the interrupt vector table in page one of that processor. This is done in one instruction, the XVCT instruction. (See the "32-Bit Principles of Operation" manual.)

While the interrupt is running it is keeping track of the time (PIT time) it is using. It does this by calling time management to read the pit. When the interrupt is dismissed, time management is again called to read the PIT again so that the CPINT.4 can be updated.

When returning from an interrupt the DISMISS routine checks the pre-interrupt state. This check is made to decide whether or not to restore the state of the interrupted path. The checks made are as follows:

> If the interrupted path was in the system and in the idle loop, then the dismiss routine transfers control to the top of the schedule. If the interrupted path was in the system but not in the idle loop, then the dismiss routine returns control back to the system path. If the interrupt path was a user path, then the DISMISS routine checks if the interrupt was on behalf of a higher priority process than the currently mapped user. This will cause a reschedule to occur. Otherwise, control is returned to the interrupted path.

For more information on how interrupts are handled, see the "32-Bit Principles of Operation" manual.

6.2.3.2 Faults

The JP code path could be diverted do to a fault. Faults are hardware violations that must be resolved before returning to a code path. Some faults cannot be resolved, so the JP may never return to that code path.

The following table shows the different types of protection faults that a code path could encounter and what component will process the fault.

FAULT CODE	FAULT TYPE
0	Read Violation
1	Write Violation
2	Execute Violation
3	Validity Violation
4	Inward Address Reference
5	Defer Violation
6	Illegal Gate Violation
7	Inward Call
10	Outward Call
11	Privileged Instruction Violation
12	I/O Protection Violation
15	Unimplemented Instruction

Figure 6.2 The Protection Fault Code Table

NOTE: For more information on the fault codes, please see the "Protection Fault Codes" table in Chapter 5 of the "32-Bit Principles of Operations" manual (014-000704-04).

6.3 The PPCB

The physical processor control block (PPCB) is a database that is used to represent each Job Processor (JP) in the system. The PPCB is used to hold status information used at schedule time for a processor and to maintain statistical information for a JP.

WORD OFFSET		USAGE SUMMARY
CPSTAT	0	Status word
CPJPIDT	1	JPID of this processor
CPLCK.W	2	PPCB lock words
CPMODE	4	LP Mode on ELQUE scan start
CPPADD	5	Padd to even word align
CPELM.W	6	Current Element Address
CPMSK.W	10	Current/Last scan mask (Mode 0/1)
CPRMK.W	12	Current reschedule mask (Mode 0 only)
CPLPCB.W	14	PPCB is attached to this LPCB
CPSADDR.W	16	Address of jp state block
CPSYS.4	20	# Ticks used by Operating System
CPCMT.4	24	# Ticks used by Core Manager
CPSMT.4	30	# Ticks used by System Manager
CPDMT.4	34	# Ticks used by Disk Manager
CPIDL.4	40	# Ticks used in Idle loop
CPINT.4	44	# Ticks used at Interrupt Level
CPESI.4	50	This CP's system sub-slice residue
CPRESI	53	(Lo-order 16-bits)
CPTMK.W	54	Current CP Tier Mask Address
CPREQ.W	56	Cross-interrupt request word

CPLEN = 60 Length of PPCB

Figure 6.3 The PPCB Offsets

BIT OFFSET		USAGE SUMMARY
PRESCH	(0)	Processor reschedule bit
CPMAST	(1)	Mother Processor bit
CPSYS	(2)	JP running system task
CPIDLE	(3)	"Go idle" bit
LSTACK	(4)	Last stack used by G1 process
CPSTOP	(5)	JP in process of being stopped
CPMOV	(6)	JP in process of being moved
CRESCN	(7)	If entering Idle loop, rescan ELQUE
CPEQL	(8)	Equal PNQF element was woken up

Figure 6.4 CPSTAT Status Word Bit Definitions

6.3.1 Offset Explanation

The following section describes the PPCB offsets and their uses in detail.

The PPCB status word CPSTAT reflects the state of this JP. The bits are described as follows:

PRESCH - This bit is checked in the idle loop and before starting up a process to see if a significant event has occurred to force a reschedule. In the case of the idle loop, the reschedule bit is the only way the idle loop can know that an event occurred. To preempt a process setup, a more significant event had to occur to force a reschedule.

CPMAST - This bit is set if the PPCB belongs to the mother processor on the system. CPMAST is set at SINIT time and is used when a system call is made to decide whether or not to run the call. If the call is mother only and this is the mother (e.g., CPMAST is set) then run the call. If not, set up the call to be run on the mother.

CPSYS - This bit is set if the JP is running a system function (task). This bit is used to check if the processor being checked is running in the system.

CPIDLE - This bit is set when the JP is changing state. This bit is checked at the top of the scheduler. If the bit is set, the JP will go to a routine and stays in a tight loop waiting for this bit to be cleared. The path that sets this bit will pend itself waiting for the Target JP to go idle. When a ?JPMOV system call is issued to assign the JP to another LP, the "mother" sets CPIDLE to force that processor to go idle while the JPMOV is being done.

LSTACK - The last stack used by this processor was used by a G1 process. This offset is used to decide whether the CB will be a G1 or G2/3 call. Typically, when a process does a system call, the CB is allocated during that subslice. If the CB used for the last system call was a G1 CB, then the CB must be changed to a G2/3 CB.

CPSTOP - This bit is set when a processor is being ?JPRELeased. This will allow the JP to finish what it is doing and reschedule to find that it must go idle and get released. This bit is checked at ?JPREL time to assure that the JP is not already being released.

CPMOV - This bit is set if the processor is being assigned to another logical processor.

CRESCN - This bit is set when a rescan is necessary for this PPCB. The bit is cleared at reschedule time. This bit is checked before going to the Checksum Loop. If the bit is set, the path gets its current scan mask and goes to a point just before the scan. This bit differs from PRESCH because PRESCH causes the path to go to the top of the scheduler, which works with queues and rebuilds the scan mask.

CPEQL - An element of equal PNQF was woken up. This bit is used to show that an event has occurred and the PNQF of the process the interrupt was for was the same as the PNQF of the currently mapped process.

The Job Processor ID (JPIDT) is the number assigned to the processor. If the processor is the mother, the JPIDT will be 0. This value is only touched at ?JPCREA time and SINIT.

The JP lock words are used for locking other processors out when touching databases relating to this processor. This is used most when an event occurs and the event handler is trying to set the reschedule flag on the PPCB. If the lock is held, the PPCB cannot be modified until the lock is released.

The CPMODE reflects what scheduler mode this JP is in currently. This is used when deciding which type (or mode) of scan mask will be used at scan time. (See LP Management.)

The CPADD just-even word aligns the rest of the databases in this database.

The Current Element address CPELM.W contains the address of the element that is currently mapped in the system. This field should have the same value as CC.W for that processor. If this field is zero, then the JP is in the scheduler.

The Current/last scan mask (mode 0/1) CPMSK.W contains the current or last scan mask used by this processor. The mode of scheduling may have changed since the last scan, so the mask is irrelevant of mode. When a rescan is needed, then the scanner gets the mask from CPMSK.W.

The Current reschedule mask (mode 0 only) CPRMK.W is the mask to be used if a reschedule should occur. This offset is set up in the scan section and tested against during EVENT to see if the element to run on that JP can run. For example, if the class that the element belongs to is masked out on this JP, then we should try to run the element on another JP.

CPLPCB.W contains the address of the Current LPCB that this JP is attached to. This offset is changed by attach and detach. (See Sections 6.4.3 and 6.4.4.)

CPSADDR.W contains the address of the JP state block. This is used and supplied by the JP instructions JPSTOP and JPSTART, respectively.

The following values are used to keep accounting statistics. These statistics are kept in clock ticks. A tick is defined as the amount of time the PIT takes to increment one of its counters by one. The way these timings are done is by a call to time management when the operation starts running and another call to time management when the operation finishes. The difference between these timings is the amount of time the operation took to run.

The CPSYS.4 four words are used to maintain statistics of how many ticks this JP spent in the system. This time is measured from the time the scheduler is entered to the time an element runs and includes the Idle loop timing.

The CPCMT.4 four words are used as a counter to maintain statistics of how many ticks this JP used by the core manager task. This offset is updated by the core manager task.

The CPSMT.4 four words are used to maintain statistics of how many ticks this JP used by the system manager task.

The CPDMT.4 four words are used to maintain statistics of how many ticks this JP used by the disk manager task. This offset is updated by the disk manager task.

The CPIDL.4 four words are used to maintain statistics of how many ticks this JP spent in the idle loop. This offset is only updated by the idle loop.

The CPINT.4 four words are used to maintain statistics of how many ticks this JP used at interrupt level. Each interrupt handler does the initial time management call and the DISMISS routine call for the database update.

The CPESI.4 four words are used to hold the system subslice residue. CPRESI is the low-order word of CPESI.4. This is currently not used in AOS/VS.

CPTMK.W is used to hold the mask address of the tier currently in use. This gets incremented when the scanner changes tiers to get the scan mask for the next tier. If the mode is 0 this value is not used. (See mode management in the ELQUE Management chapter.)

The CPREQ.W contains the JP cross interrupt word. This word holds the interrupts currently defined as cross interrupts for this JP. The only cross interrupt defined in AOS/VS is for MP Histogramming during the RTC interrupt.

CPLEN is the length of the PPCB. This is currently 60 words.

6.3.2 The JP Globals

In the multiprocessor environment all processors share the system resources such as memory. Therefore, all processors can access anything in memory. To maintain the integrity of a JP state however, there must be some memory that must be JP unique. This memory is the JP's pages zero and one. Page zero contains all the information needed to maintain the state of that JP. Information such as CC.W (current element running on that JP) and INTLV (interrupt level) are kept in the JP's page zero. In the JP's page one is the interrupt vector table for that JP. In the sources these variables are referenced in "SZERO.LS".

There are some globals that are not defined as JP unique. Examples of these are ELQUE (the ELigible QUEue) and LRUCH.W (the Least Recently Used Chain - see Memory Management). In the sources these globals are found in "STABLE.LS".

There are 12 global variables used by JP management: JPUMBASE, JPUMSIZE.W, MAXCP, MAXJP, JPCNT, JPUBUFF.W, MYPPCB.W, MYJPID, CP.W, PP.W, CKSUM, and INCHK.

JPUMBASE.W is the pointer to the table of JP unique pages. This pointer is used to find the pages of a specific JP. The calculation used to find a JP's page zero (using FED notation) is as follows:

```
JP1(page zero) = JPUMBASE.W\(address of table)\
                          (JP0s page zero)
               .+(the contents of JPUMSIZE.W)*JPID\
```

JPUMSIZE.W contains the number of words that are JP unique. In revision 7.50, this number is 4000 (octal). This is used in the above calculation to find the JP unique pages.

MAXCP is a count of the maximum number of Command Processors (CPs) allowed in the system. The CP is the same as the JP to AOS/VS. For rev 7.50 the maximum number of JPs allowed in the system is two.

MAXJPID is the highest JPID allowed in the system. Since there are only two processors allowed in the system, and we count from 0, the max JPID is 1.

JPUBUFF.W contains a pointer to the microcode buffer used at ?JPINIT time if the current microcode is used to start the newly inited processor.

MYPPCB.W contains a pointer to the PPCB for that processor. This is a quick way for the JP to find its PPCB. This is unique to each processor.

MYJPID contains an integer value, which is the JPID for that processor. This is used to find the JPID for a processor when doing cross interrupts or when mapping a process. In the Cross interrupt code a check is made to see if the JP is cross interrupting itself. In the map user code a check is made to see if the ATU for that JP needs to be purged before mapping a user.

PP.W is the starting point for an array of pointers to the PPCBs of all the inited JPs in the system. This list is ordered by JPID. There is one copy of this array for the entire system.

CP.W is the starting point for an array of pointers to the PPCBs of all the active JPs in the system. There is no ordering to this array, although location 0 of this array is likely to be the address of the PPCB for JP0.

MYTIME.W is a pointer to the offset in the table of CPU times. Each time this JP goes through the scheduler, the time of day (in seconds) is stored through MYTIME.W into the table offset for this JP. There is one copy of MYTIME.W for each JP.

CPUTIM.W is the table of watch dog timers for all the active JPs in the system. The contents of this table are updated each time the JP goes through the scheduler. Before the JP goes idle, this table is scanned to see if any of the processors have not updated their respective entry in the table in the last 15 seconds. If the limit is reached, then the JP send a message to the OP console: "WARNING: Processor #(JPID) has not rescheduled for (table entry) seconds. May be hung." Then the processor continues checking and eventually goes into the Checksum loop.

CKSUM is the checksum value used in the Checksum loop. This is an integer that contains the XORed values of that JP's static locations. This value is built during system initialization. During JPINIT, CKSUM for the new JP is copied from the JP0 CKSUM.

INCHK is the word that shows if a JP is in the Checksum loop. This global is JP specific. If the value of INCHK is zero, then the JP is not in the Checksum loop. If INCHK is 1, then the JP is in the Checksum loop.

6.4 Basic Operations on the PPCB

The PPCB is changed by using certain basic operations. These operations may or may not be explicitly referenced in the code as routines. The basic functions that affect the PPCB are: Alloc.ppcb, Dealloc.ppcb, attach.ppcb, dettach.ppcb, update.time, get_mask, update mode, Lock.ppcb, Idle_jp, and EVENT.

6.4.1 Alloc.ppcb

This operation allocates memory for the PPCB and the JP state block. The pseudocode below shows how this operation is done. This operation is only used when starting a JP and at system initialization.

```
/* $$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$ */
/*                                                           */
/*                    ALLOC PPCB                             */
/*                                                           */
/*    This routine allocates space for a PPCB and            */
/*    its jpstate block.  The memory management              */
/*    function used is GSMEMNW (get memory no wait)          */
/*                                                           */
/* $$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$ */

ALLOC.PPCB( )
  {
    ppcb_address = GSMEMNW(getmem_no_wait(ppcb_len));

/* ********************************************** */
/*                                                */
/*  If we are not an MP system then there is no   */
/*  need for a JP state block.                    */
/*                                                */
/* ********************************************** */

    if   (mp_system)
         ppcb.saddress = GSMEMNW
         (getmem_no_wait(ppcb_state_blklen));
    return(ppcb_address);
  }
```

6.4.2 Dealloc.ppcb

This operation deallocates memory that the PPCB and the JPstate block held. This operation is used if the user does a JPREL or if the JPINIT loading of microcode fails. In AOS/VS this routine is called FREE.PPCB. The pseudocode below shows how this operation works:

```
/* $$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$ */
/*                                                 */
/*               DEALLOC PPCB                      */
/*                                                 */
/*    This routine frees the memory allocated to the*/
/*    PPCB and the JPstate block.                  */
/*                                                 */
/* $$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$ */

dealloc.ppcb(ppcb_address)
  {
    ppcb=ppcb_address;

/* ************************************************ */
/*                                                 */
/* If we are not an MP system, then there is no    */
/* JPstate block to deallocate.                    */
/*                                                 */
/* ************************************************ */

    if (mp_system)
         mem_manage (release_mem,ppcb.saddress,jp_state_len);
    RSMEM(ppcb,ppcb_len);
    return();
  }
```

6.4.3 Attach a PPCB to an LP

This operation attaches a PPCB to an LP so that it can get a mask from somewhere to do class scheduling if class scheduling is enabled. This function is called each time a JP is inited into the system, when a JPMOV is done, and during system initialization. The pseudocode below shows how this operation works.

```
/* $$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$ */
/*                                                           */
/*                      ATTACH                               */
/*                                                           */
/*  This routine attaches a JP to an LP.                     */
/*                                                           */
/*                                                           */
/* $$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$ */

ATTACH(lpcb,ppcb)
  {

/* ********************************************** */
/*                                                */
/*  If the LPCB field in the ppcb is not zero then */
/*  panic with a 14615.  If this happens then the  */
/*  JP is already attached to an LP.  This is the  */
/*  attach panic.                                 */
/*                                                */
/* ********************************************** */

    if (!(ppcb.lpcb.w == 0))
         panic(14615);
    else
         ppcb.lpcb.w = lpcb;

    jpid = ppcb.jpid;
    lpcb.lpjpcnt++;

/* ********************************************** */
/*                                                */
/*   LPJPMP.W is a bit map.  If the bit for the   */
/*   JPID is 1 then panic with a 14615.           */
/*   Otherwise, set the bit in the bit map for    */
/*   that JPID.  If the bit is already set, then  */
/*   the JP is already assigned to the LP.        */
/* ********************************************** */

    if (!(bit(lpcb.lpjpmp.w,jpid) == 0))
         panic(14615);
    else
         bit(lpcb.lpjpmp.w,jpid) = 1;  /* set bit */
```

```
/* ********************************************** */
/*                                                */
/*  If this is the mother JP we're attaching, then */
/*  set the "mother is attached to this LP" LPMOM  */
/*  bit in the LPCB.                              */
/*                                                */
/* ********************************************** */

    if (bit(ppcb.cpstat,cpmast) == 1)
     {
        lock(lpcb);  /* this turns off interrupts*/
         bit(lpcb.lpstat,lpmom) = 1;
         unlock(lpcb);
         interrupts(enable);
      }
    return();
  }
```

6.4.4 Detach

This operation detaches a JP from an LP. This is called when doing a JPMOV, a JPREL, or if a JPINIT fails. The pseudocode below shows how this operation works.

```
/* $$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$ */
/*                                                           */
/*                      DETACH                               */
/*                                                           */
/*  This routine detaches a JP from an LP.                   */
/*                                                           */
/*                                                           */
/* $$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$ */

detach(lpcb,ppcb)
  {
/* ********************************************************* */
/*                                                           */
/*   If the LPCB field in the ppcb is zero, then             */
/*   panic with a 14616.  This is the detach panic.          */
/*   This error means that we're trying to detach            */
/*   something that is not attached to any LP.               */
/* ********************************************************* */
    if (ppcb.lpcb.w == 0)
         panic(14616);
    else
         ppcb.lpcb.w = 0;
    jpid = ppcb.jpid;
    lpcb.lpjpcnt--;

/* ********************************************************* */
/*                                                           */
/*   LPJPMP.W is a bit map.  If the bit for the              */
/*   JPID is 0, then panic with a 14615.                     */
/*   Otherwise, clear the bit in the bit map for             */
/*   that JPID.  This error means that the JP is             */
/*   not attached to this LP.                                */
/* ********************************************************* */
    if (!(bit(lpcb.lpjpmp.w,jpid) == 1))
         panic(14615);
    else
         bit(lpcb.lpjpmp.w,jpid) = 0;  /*clear bit */
```

```
/* ********************************************** */
/*                                                */
/*  If this is the mother JP we're detaching, then */
/*  clear the "mother is attached to this lp" LPMOM*/
/*  bit in the LPCB.                              */
/*                                                */
/* ********************************************** */

    if (bit(ppcb.cpstat,cpmast) == 1)
     {
        lock(lpcb);  /* this turns off interrupts*/
         bit(lpcb.lpstat,lpmom) = 0;
         unlock(lpcb);
         interrupts(enable);
      }
    return();
  }
```

6.4.5 Update Time

This operation updates the time counters in the PPCB. This routine does not really exist, but it functionally does. To see the paths that access this operation, refer to the paths section below. The pseudocode below shows how this function works.

```
/* $$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$ */
/*                                                */
/*                     UPDATE_TIME                */
/*                                                */
/*   This function updates the time that a time   */
/*   counter in the ppcb.                         */
 *                                                */
/*                                                */
/* $$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$ */

update_time(start_time,end_time,function)

    ppcb.[function] += end_time - start_time;
```

6.4.6 Set Mask

This function stores the current scan mask into the PPCB. The mask will be used when a rescan is needed. Otherwise, the mask is not used.

```
/* ********************************************** */
/*                                                */
/*              SET MASK                          */
/*   This routine gets the current scan mask to   */
/*   be used from the LPCB and stores it in the   */
/*   PPCB.  If the mode is 1, then we offset into */
/*   tiers table.                                 */
/* ********************************************** */

set_mask(mode,lpcb,mask_offset)
   {
    if (mode == 1)
         ppcb.cpmsk.w = lpcb.lphmk.w[mask_offset];
    else
         ppcb.cpmsk.w = lpcb.lpcsm.w;
  }
```

6.4.7 Lock PPCB

This operation tries to lock the PPCB to do some work on it. This operation can spin on the lock. In the code this operation is called XLOCK (See Process Management Interface Services for a general explanation.)

```
/* ********************************************** */
/*                    XLOCK                       */
/*  This routine sets the lock on a lock word     */
/*  passed to this routine.  This is a short-term  */
/*  lock.  This routine will spin waiting for lock.*/
/* ********************************************** */

xlock(bit_offset,word_address)
  {
/* ********************************************** */
/*  Initially do an atomic test and set operation. */
/*  An atomic test and set is necessary to maintain*/
/*  the integrity of the lock.  In the assembly,   */
/*  the instruction is a WSZBO instruction.  While */
/*  this instruction is running no other operation */
/*  that works with memory can run.  In the inner  */
/*  loop a non-atomic test operation loops (SPINS)  */
/*  until false.  Once false the routine must again*/
/*  try the atomic test before leaving the routine,*/
/*  because the lock may have been set during the  */
/*  transition from the inner to the outer loop.   */
/* ********************************************** */
   while (test_n_set(bit(word_address,bit_offset) == 1))
         while (bit(word_address,bit_offset) == 1)
              {}
  }
```

6.4.8 Idle JP

This operation sets the bit in the PPCB status word to tell that JP to go idle. The operation then pends waiting for the JP to go idle. This idling is not the idle loop; it is a place that allows the JP to effectively pend. The pseudocode below shows how this operation works.

```
/* ************************************************ */
/*                    IDLE_JP                       */
/* This routine sets the "go idle" bit in the PPCB  */
/* and pends waiting for the JP to go idle.         */
/*                                                  */
/* ************************************************ */

idle_jp(ppcb)
  {
```

```
/* ************************************************ */
/*                                                  */
/* If the caller tries to idle the mother, a panic  */
/* 14622 will result.                               */
/*                                                  */
/* ************************************************ */

    if (bit(ppcb.status.cpmast) == 1)
         panic(14622);
/* ************************************************ */
/*                                                  */
/*  Get the ptran lock and set the cpidle and waiters */
/*  bits.  Release the ptran lock and pend.  When   */
/*  the JP is "idle" this routine gets unpended.    */
/*  We get the global jplp lock and return.         */
/* ************************************************ */

    xlock(ptran,ppcb.status);
    bit(ppcb.cpstatus.cpidle) = 1;
    bit(ppcb.cpstatus.wait) = 1;
    bit(ppcb.cpstatus.ptran) = 0;
    pend(idle);
    get_jplp_lock(change);
    return();
 }
```

6.4.9 IDLE

This routine is the spin routine that is forced because the CPIDLE bit in the PPCB status word has been set. This bit is only set when the PPCB for this JP is going to have something happen to it that cannot allow this JP to be doing anything that may affect the LPCB or the PPCB. In other words, the JP must somehow PEND. The JPIDLE bit is tested at the top of the scheduler so there is nothing happening on this JP. It is in a state that allows the JP to go into the "IDLE" state.

```
/* $$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$ */
/*                        IDLE(PPCB)                             */
/*  This routine spins on the CPIDLE bit in the PPCB status word. */
/*                                                               */
/* $$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$ */

idle(ppcb)

    {
    MYTIME.W = 0;
    unpend(ppcb->cplck.w);/* Unpend the code that set */
                          /* the lock so we don't stay*/
                          /* idle forever.            */
    while(bit(ppcb->cpstat,cpidle) == 1)
         {}
    }
```

6.4.10 EVENT

EVENT is called when something happens in the system that causes the reschedule flag to be set. This event that causes a reschedule usually is an interrupt. In an MP environment, the routine will check all the processors to see which one is running the lowest priority path. The priority of a path is defined by PNQF (see ELQUE Management) if it is a user path or CB. If the path is a system path, then the check is made in the JP in the Checksum Loop. Periodically in the system there are checks made for the reschedule flag being set on that JP. If the reschedule flag is set, then that JP will go to the Scanner.

In the code there are two entry points related to EVENT: EVENT and MEVENT. MEVENT is only used when the JP that needs to be rescheduled can only run on the mother.

```
/* $$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$ */
/*                  EVENT/MEVENT                   */
/*  Set the reschedule flag on the JP running the  */
/*  lowest priority path. MEVENT sets the reschedule*/
/*  flag on the Mother.                            */
/* $$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$ */

EVENT(element)/MEVENT(element)

     {
     int ppcb,epnqf,epclass,i;


/* ************************************************** */
/* If MEVENT was called, then do the "mother only"    */
/* work.  Increment the "MEVENT called" global counter*/
/* NMEVENT.W.  The label mother_only is used if EVENT */
/* finds that the element that caused the event is    */
/* "mother only".                                     */
/* ************************************************** */

     if (MEVENT)
        {
        NMEVENT.W++;  /*  ONE MORE MEVENT  */
    mother_only:

/* ************************************************** */
/* Is the reschule flag already set?  If so, then     */
/* return.  If the flag is not set then go see if     */
/* we should set the reschedule flag.  To do the      */
/* checking call the routine MATCH1.  See Section     */
/* 6.4.11 MATCH.                                      */
/* ************************************************** */

        if (bit(MPPCB.W->cpstat,presch) != 1)
           if (MATCH1
           (element->pclass,element->pnqf,&ppcb))
              setbit(ppcb->cpstat,presch);
        }/*  MEVENT */

/* ************************************************** */
/* If MEVENT was not called, then EVENT was called.   */
/* Check if the element that interrupted is already   */
/* running.  If so, return.                           */
/* ************************************************** */

     else
        {
           if ((bit(element->pstat,psrun) != 1)
              {
```

```
/* ************************************************ */
/* Are we a single processor system?  If so, then   */
/* check our reschedule flag and check if the element */
/* can cause a reschedule to occur without doing    */
/* other work.  This is a short cut for nonMP systems */
/*                                                  */
/* ************************************************ */

            if (MAXCP == 1)
               {
               if (bit (MYPPCB.W->cpstat,presch != 1)
                  if (MATCH
                  (element->pclass,element->pnqf,&ppcb)
                     setbit(ppcb->cpstat,presch);
               }/* 1 JP  */
            else
            {

/* ************************************************ */
/* If the interrupting element is "mother only", go to*/
/* the mother_only label in the MEVENT section.     */
/* ************************************************ */

            if (bit(element->pstat,pmast) == 1)
               goto mother_only;

/* ************************************************ */
/* Start looping through the processors to find the */
/* processor that is able to be rescheduled.  A     */
/* JP can get the reschedule flag set if the element */
/* running on it is of lower priority or if the JP  */
/* is in the Checksum Loop.                         */
/* ************************************************ */

            for (i=0;i<=MAXCP;i++;)
               {
               epnqf = element->pnqf;
               epclass = element->pclass;
```

```
/* ************************************************ */
/* Is this JP in the JP's table CP.W defined?  If so, */
/* go to the next JP in the table (e.g., loop).  If   */
/* not, then continue checking.                       */
/* ************************************************ */

                if ((CP.W[i] != -1)
                   {
                   ppcb = CP.W[i];

/* ************************************************ */
/* If this JP is not "IDLE" and the reschedule flag   */
/* is not set for that JP, then continue.  "IDLE" is  */
/* the state the processor can be in if the JP is     */
/* being moved or released.                           */
/*                                                    */
/* ************************************************ */

                   if ((bit(ppcb->cpstat,cpidle) != 1)
                   !! (bit(ppcb->cpstat,presch) !=1))

/* ************************************************ */
/* Can the element set the reschedule flag for this   */
/* JP?  If so, set it and break out of the loop.      */
/* ************************************************ */

                      if (MATCH(epclass,epnqf,&ppcb))
                         {
                         setbit(ppcb->cpstat,presch);
                         break;
                         }/* if there is a match */
                   }/* if defined   */
                } /*  for loop  */
             }   /* not 1 JP  */
          }/* element not running */
       }/* not MEVENT  */
    return();
    }  /* EVENT/MEVENT   */
```

6.4.11 MATCH/MATCH1

MATCH and MATCH1 are Boolean functions that check if a processor can have its reschedule flag set. Match first checks if the class the element is in is defined on the LP that this JP is attached to. If the class is not defined, then the function returns FALSE without setting the rescan flag (see below). MATCH1 is the beginning of the common code for both functions.

There are two cases where these functions will return TRUE: 1) The JP is in the checksum loop; 2) The JP is running an element of lower priority than the one that these functions are called with.

If the functions return FALSE, they will also set the Rescan flag (rescn) in the PPCB. The functions will return FALSE if: 1) The JP is running system code other than the checksum loop; 2) The PNQF of the element running on the processor is of higher or equal priority than the one supplied by the caller. If the priority is equal to the element supplied by the caller, then set the BCPEQL flag; 3) The disk manager task is running. This CB (see CB management) is the highest priority element in the system; 4) The element is in a class that is not defined or is masked out on the LP that this JP is attached to. This is a special case and will not set the rescan flag.

6.5 Paths that Affect the PPCB

There are two main paths that access the PPCB: the scanner and the time accounting path. There is a third path that reflects the state of the JP, but does not affect the PPCB; that path is called the Checksum loop.

6.5.1 The Scanner

The scanner is the path, in the module SCHED, that scans ELQUE to find a CB or ptbl to run. This path is not called "scanner" in the code, it has several entry points. For an explanation of the entry points into the scanner, see ELQUE Management. The scanner affects the LPCB when there is a RESET of the lpcb databases after a scan of ELQUE fails. The scanner also affects the LPCB when there is a mode change.

The scanner is presented from the JP point of view. The reason for this is the scanner in the system imbeds three logically separate points of view in line. For example, consider the following lines of C code.

```
A:   if (bit(myppcb.w.cpstat,cpmast)  != 1)
     /*daughter?*/
        setbit(mask,process_mother_bit);

B:      element = SCAN(*ELQUE,mask);

C:      if (mask == mylpcb.lpciu.w)
           {}
```

"A:" is supplied by JP management because the if statement uses the global MYPPCB.W to find out if this is a daughter processor (see JP management).

"B:" is supplied by ELQUE management because the scan function uses ELQUE, which is managed in ELQUE management.

"C:" is supplied by LP management, because the initial scan mask comes from the LPCB.

In the above example, three major areas of Paths and Time are used in three nearly consecutive commands. This is the way the code is really presented in the system, but to modularize the scanner for each section the scanner is presented with different emphasis.

The pseudocode below shows the scanner from the ELQUE management point of view.

```
/* $$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$ */
/*                                                    */
/*               The Scanner                          */
/*    The scanner will loop forever unless left       */
/*    to go_idle.  (See ELQUE  Management.)           */
/*                                                    */
/* $$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$ */

#define loop_forever true
scanner()
{
int  model_tier; /* variable used for call to set_mask*/

model_tier = 0;

/* ************************************************** */
/*                                                    */
/*    Put the current time stamp in seconds into our  */
/*    offset in the Watchdog timer array.  This will  */
/*    keep the other processors from thinking that    */
/*    this JP is hung.                                */
/*                                                    */
/*    Check to see if the "go idle" bit is set.  If   */
/*    it is then stay idle on this bit while another  */
/*    processor does work on this one.                */
/*                                                    */
/* ************************************************** */

CPUTIME [MYPPCB.W->jpid] = TODH.W;
if (bit(MYPPCB.W,cpidle) == 1)
     IDLE (MYPPCB.W);
while(loop_forever)
  {
/* ************************************************** */
/*                                                    */
/*    Get the current mask.  If child processor then  */
/*    mask out the mother bit and scan ELQUE.         */
/*    MYPPCB.W is the PPCB for that JP.               */
/*    mylpcb.w is the address of this lpcb.           */
/*    setmask is a function in JP management.         */
/* ************************************************** */

    mask = set_mask(mode,mylpcb,model_tier);
    if (bit(myppcb.w.cpstat,cpmast) !=1)/*daughter?*/
       setbit(mask,process_mother_bit);
    element = SCAN(*ELQUE,mask);
```

```
/* ************************************************ */
/*  If the scan was successful the element will get  */
/*  dispatched.  Below is a representation of the    */
/*  GOTO dispatcher logic.  It will show the different*/
/*  dispatchers the scanner can send an element to.  */
/*  There are four dispatchers the scanner can send  */
/*  an element to:  PCALL for PTBLs, TACT for user   */
/*  CBs, Tact1 for system CBs, and CMINT for the Core */
/*  Manager CB (a special case).  In the code the    */
/*  scanner merely jumps indirect through location   */
/*  PPC.W in the PTBL/CB common area.                */
/* ************************************************ */

     if (successful_scan)
        {
        dispatch(element); /* see elque management */
        }  /* successful scan */
        else

/* ************************************************ */
/*                                                  */
/*  If not successful, then check if class scheduling */
/*  is on.  If not, go to the checksum loop.  If class*/
/*  scheduling is on then check to see if a mode     */
/*  change is necessary.  If the current mask is     */
/*  the same as the initial scan mask, RESET and     */
/*  change mode.  This is considered a sufficient    */
/*  check because the scan failed with the initial   */
/*  mask, meaning there are no more primary classes  */
/*  ready to run.  To avoid a second unnecessary scan,*/
/*  RESET to run the secondary classes.  (See LP     */
/*  Management for mode information).               .*/
/* ************************************************ */

        If (bit(MYLPCB.W->lpstat,lpoff) == 1)
           goto SMONDD;
        if (mask == mylpcb.w->lpciu.w)
          {
           reset(mylpcb.w);
          }  /* if */
        else
```

```
/* ************************************************ */
/*                                                  */
/*  If not changing the mode, then check if to see  */
/*  what mode we're in.  If mode 0 then reset the   */
/*  LP databases and get a new mask; if not, get the */
/*  next tier if possible.                          */
/* ************************************************ */
          if (myppcb.current_mode == 0)
             reset_(mylpcb.w);
             mode1_tier = 0;
          else
              if (myppcb.cptmk.w+2 > myppcb.cphmk.w+32)
                 {
                 myppcb.current_mode = 0;
                 mode1_tier = 0;
                 }
                 else
                 {
                 myppcb.cptmk.w += 2;   /* JP */
                 mode1_tier =  myppcb.cptmk.w;
                 }/* else */
  }  /* while loop */
 }  /*  scanner  /*
```

6.5.2 Time Accounting

Whenever the system is entered the time used by the system must be accounted for. This is done by the JP that the system is running on. (See Time Management.) The timings done by the system are saved in the PPCB in the following manner:

```
/* ************************************************ */
/*                                                  */
/*  Take a reading of the PIT and save it in        */
/*  the function base.  Do the function...          */
/*  Take a reading, subtract base from the reading, */
/*  and store in ppcb for that function type.       */
/* ************************************************ */

enter_sys_function:
     function_base = LOOK.PIT;

     DO_FUNCTION_WORK;

     temp = LOOK.PIT;
     update_time(function_base,temp);
     exit;
```

The types of functions that these timings are saved for are as follows.

FUNCTION	BASE NAME	PPCB ENTRY
Operating System	SBASE	CPSYS.4
Core Manager	CBTIME *	CPCMT.4
System Manager	CBTIME *	CPSMT.4
Disk Manager	CBTIME *	CPDMT.4
Idle loop	CBTIME **	CPIDL.4
Interrupt Level	IBASE	CPINT.4

Figure 6.5

* Offset in the CB. (See CB Management)
** Offset in the CB. (See CB Management) Uses whatever CB is addressed by PCB.W as a temporary storage CB.

For more information on the databases relating to these functions, see the PPCB definitions.

6.6 The Idle Loop (Checksum)

AOS/VS will go into an "IDLE" mode if it can find nothing on the ELQUE to run. This means that no PTBLs or CBs are eligible to run. Since a JP cannot be doing "nothing" the JP will go into a loop that checksums the system page zero. This is called the checksum loop (idle loop). Before going into the checksum loop, JP management must prepare to go into the "IDLE" state.

There are three entry points into the idle loop from the rest of the system: SMONDD, SMOND, and COK. SMONDD is the beginning of the idle loop preparation routine. (See below.) SMOND is the beginning of the actual idle loop. COK is the beginning of the "leave the idle loop" routine.

6.6.1 Preparing for the Idle Loop

Before entering the idle loop, some system sanity checks must be made. These checks deal with the watch dog timer and system deadlock checking. The entry point for this section is SMONDD.

The watch dog timer is used to see if the other JP in the MP environment is HUNG. The timer is a double-word time stamp used to hold the current time of day. This field is changed each time the JP goes through the scheduling code. The time stamp is compared with the current time of day. If the time stamp is greater than 15 seconds, then the checking JP assumes the other JP is hung. If this conclusion is reached, then the JP will send a message to the operator console telling the user that the other JP is hung.

JP management does some deadlock checking before going into the idle loop. This checking is done to prevent the swap waiter (see Memory Management) from deadlocking the system. If a process is being swapped and has not finished yet, and if another process is trying to do some move operation into the memory of the waiter, then the process will wait for the memory and possibly wait forever. The pseudocode below shows how the deadlock checking is done.

```
/* ********************************************** */
/*                Dead lock checking              */
/* This code shows how the swapwaiters deadlock   */
/* check is done.                                 */
/*                                                */
/* If we're in ESD then ignore this checking.     */
/*                                                */
/* ********************************************** */

if (ESDSW != -1)
    {
/* ********************************************** */
/*                                                */
/*  If there are swap waiters on the system then... */
/*                                                */
/* ********************************************** */

     if (SWAP_WAITERS != -1)
        {
/* ********************************************** */
/*                                                */
/*  check to see if there were any swap waiters the */
/*  last time we checked.  If not set up the timer */
/*  for the length of time there is a waiter.     */
/*  If so, check if the waiter we set the timer for */
/*  is the same as the current waiter.            */
/*                                                */
/* ********************************************** */

          if (SWAP_TIMER.W == 0)
             SWAP_TIMER = time_of_day;
          else
/* ********************************************** */
/*                                                */
/* If we have a different waiter then reset the   */
/* waiter timer.  Otherwise, check for how long we */
/* waited.                                        */
/* ********************************************** */
             if (last_swap_event == current_swap)
                SWAP_TIMER = time_of_day;
             else
/* ********************************************** */
/*                                                */
/* If the waiters have been here for more than three */
/* seconds, then unpend the waiter.               */
/*                                                */
/* ********************************************** */
                 if (SWAP_TIMER > 3)
                     unpend(SWAP_WAITER);
        } /* waiters */
    } /* not ESD  */
```

6.7 The Checksum Loop

After doing all the preparation the system finally enters the "idle" state and goes into the Checksum Loop. The checksum loop is a loop that checksums the JPs page zero. The checksum loop does its memory work with the lights (interrupts) on, so it will not miss a necessary event such as a device interrupt. Before entering this loop the JP sets the global INCHK to 1. This word lets the system know that the system is idle.

Knowing the system is idle is important because if something significant happens, such as an interrupt that sets an entity ready to run, the system should go immediately to schedule the entity. Therefore, when dismiss finds out that INCHK == 1; instead of restoring state back to the checksum loop, dismiss will go to the top of the scheduler (scanner). Checking the system idle word avoids unnecessary waiting by the user while the system is "idle".

The checksum loop compares a memory location against itself. If the memory location contains a different value, then panic because the hardware is probably broken (BROKE!). This testing is done for all of the JP's page zero. After going through the JP's page zero, a comparison is made to see if the page zero checksums. After one iteration through the checksum loop, control will go to the top of the scanner.

The pseudocode below shows how the checksum loop works.

```
/* ********************************************** */
/*               The Check Sum Loop.              */
/*                                                */
/* The entry point for this routine is SMOND.     */
/* This loop checks for valid memory comparing the */
/* contents of the location against the same      */
/* location.  If they are different, then panic.  */
/* ********************************************** */

SMOND( )
 {
     int check_value,i,temp;
     check_value = 0;
     INCHK = 1;
     interrupts(enable);
     for (i=o;i==ZCKST;i++)
        {

/* ********************************************** */
/*                                                */
/* If the system reschedule flag is set, then go out */
/* of the checksum loop.                          */
/*                                                */
/* ********************************************** */

        if (bit(myppcb.w->cpstat,presch) == 1)
           goto leave_idle;

/* ********************************************** */
/*                                                */
/* If the memory location is not the same as itself, */
/* then the hardware must be broken.  So panic with */
/* an 11002.                                      */
/* ********************************************** */

        if (page_zero[i] != page_zero[i])
           panic(11002);
        check_value ^= page_zero[i];
        }

/* ********************************************** */
/*                                                */
/* If the check_value does not match the system   */
/* checksum value, then panic.                    */
/*                                                */
/* ********************************************** */

     if (cksum != 0)
        if (check_value != cksum)
           panic(11001);
```

```
/* ********************************************** */
/*                                                */
/* "leaving_idle" is an entry point called "COK" in  */
/* the code.  This section transfers control back to */
/* the top of the scheduler. (scanner)               */
/* ********************************************** */

leave_idle:  /*  COK  */
     interrupts(disable);
     inchk = 0;
     temporary = pcb.w;
     tcsys();
     myppcb.w->cpidl.4 += time_used;
     goto scanner;
  }/* smond */
```

6.8 Locking

There are three basic types of locks at the JP level: interrupt disable, interrupt masking, and the spin lock.

The interrupt disable lock is used when a JP is doing something that puts it into an unstable state. For example, the JP should have interrupts off when modifying its stack because if the stack is in an unstable state when the interrupt comes in, the stack could be declared "INVALID" by the microcode when restoring the state. This is also used to lock out interrupt code that could also modify a data base.

Unfortunately, disabling interrupts is not good if what you want to do is drive devices as much as possible. Therefore, disabling interrupts should be done for as little time as possible.

The second type of locking scheme is masking interrupts. This method modifies the interrupt mask (CMSK). This scheme allows you to modify a critical area that a particular device will affect. For example, when putting a process on the delay chain (see process management interface services) the Real Time Clock (RTC) (see time management) must be masked out because there may be a RTC interrupt putting some PTBL on the delay chain.

The third type of locking scheme is spin locking. This scheme spins while waiting to get access to a lock bit. Access to a lock bit is allowed when the bit is off(0). The problem with this method of locking is that the JP is not doing anything else while waiting for this lock. Spinning is shown below in an assembly routine:

```
     AC1 -> Bit Displacement of lock.
     AC2 -> Word Address of lock.

XLOCK:  INTDS        ; TURN INTERRUPTS OFF
        WSZBO 2,1       ; ATOMIC TEST AND SET
        WBR      SPIN   ; SPIN ON THE LOCKED LOCK
        WPOPJ        ; RETURN TO CALLER

SPIN:   INTEN        ; TURN INTERRUPTS ON
        WSZB     2,1       ; NON ATOMIC TEST OF LOCK
        WBR      SPIN      ; STILL LOCKED KEEP SPINNING
        WBR      XLOCK ; GO TRY TO ATOMICALLY SET
                    ;  LOCK
```

There is another routine that uses the same locking scheme. This routine is called BSLOCK.

NOTE: For more information on this routine, see spin lock PPCB in Section 6.4.7 for basic operations.

6.9 User Services

When booted and during initialization, AOS/VS sets up and starts LP0 and PP0. This is to ensure that there is at least one physical and logical processor in the system.

Once the system is running, there are several system calls (User Services) which can be issued to manage the JP environment. They are:

```
?JPINIT - Initialize a JP
?JPREL  - Release a JP
?JPMOV  - Move a JP to a new LP
?JPSTAT - Get JP status information
```

Assuming that AOS/VS is up and running and we have a multi-processor system, what do we do next to use the multi-processor capabilities of the hardware?

Actually there is already one JP running and one LP available, since these are needed to get AOS/VS up and running. This booted processor is called the "mother processor" (also called the "initial processor"). Any future additional processors will be called "child" processors.

To add a JP to the multiprocessor system, the system manager must initialize the new JP. The system call to do this is ?JPINIT.

6.9.1 ?JPINIT

The purpose of JPINIT is to initialize a JP into the system. It has several options that allow the system manager some flexibility in using the multiprocessor environment.

The first and most important check is to make sure the caller has the MP privilege. The packet used contains some other very important parameters. They are:

1. JPID
2. Flag word
3. LPID
4. uCode options
5. length of uCode string
6. Byte pointer to pathname of microcode

Flag word is used to tell JPINIT how to handle the microcode loading procedure. If flag word has a value of 10, then use the microcode file pointed to in the packet. If it is 1, then use the microcode that already exists in the JP. If it is 0, then use the default microcode based on the JP's CPUID.

The microcode option word is used with the JPLCS instruction. With this option word, the machine will be able to determine what it may have to load. It has the capability to load microcode (i.e., use FP microcode or not).

The following is a description of what happens when the ?JPINIT system call is issued.

```
/* $$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$ */
/*                                                               */
/*  If the user does not have the MP privilege                   */
/*  leave the call.                                              */
/*                                                               */
/* $$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$ */

/* This is the user packet supplied to the system call.*/
```

LOC	Name	Summary
0	pckid	Packet identifier
2	reserved	
3	flags	JPINIT flag bits
4	jpid	jp id for the JP being inited
5	Lpid	LPID of LP to assign the JP to
6	uc_option	Microcode options word
7	uc_bptr	Byte pointer to microcode file
11	uc_size	size of the microcode pathname

```
JPINIT.P(packet)
     {
    if (SYS_PRV)
         return(user_not_privileged);

/* ********************************************************** */
/*                                                            */
/*  Set up the stack the JP will use when it                  */
/*  runs.  Setting up the stack means allocating a            */
/*  Primary CB for that JP.  Check if system can              */
/*  have more than one JP.  Currently only MV/20000s*/
/*  can be MP systems.                                        */
/* ********************************************************** */

     SET_UP_STACK();

     if (!(mp_system))
        return(not_an_MP_system);

/* ********************************************************** */
/*                                                            */
/*  Set up the trap handler in the CB in case                 */
/*  there is a serious problem we are unprepared              */
/*  to deal with.  Get the packet info.                       */
/*  See above for packet info.                                */
/*                                                            */
/* ********************************************************** */
     set_up_trap_handler();
     Get_packet_info();
```

```
/* ********************************************** */
/*                                                */
/*  If supplied, get the pathname of the microcode */
/*  file.                                         */
/*                                                */
/* ********************************************** */

     if (pathname_supplied)
        move_to_system_space();
     Clear_trap_handler();  /* now we can handle errors*/

/* ********************************************** */
/*                                                */
/* The user supplied JPID and LPID must be valid  */
/* before the JP can be brought into the system.  */
/*                                                */
/* ********************************************** */

     if ((packet->jpid << 0) !! (packet->jpid >> 1))
        return(invalid_jpid);
     if ((packet->lpid << 0) !! (packet->lpid >> 15))
        return(invalid_lpid);

/* ********************************************** */
/*  If the user supplied the path of a microcode  */
/*  file, then the user will have set the bit in  */
/*  the flag word in the packet.  If so, get and  */
/*  validate the byte pointer to the pathname.    */
/* ********************************************** */

     if (bit(packet->flags,0) == 1)
        if (packet->UC_NAME == 0)
           return (invalid_byte_pointer);

/* ********************************************** */
/*                                                */
/* Now we lock the JP LP database (see pend       */
/* locking) and see if PPCB is already there.     */
/*                                                */
/* ********************************************** */
     LOCK_JP_LP();

     if (ppcb_exists)
        return(JP_already_exists);
     save = packet->lpcb;
```

```
/* ********************************************** */
/*                                                */
/* Now set up to load microcode.  See if the JP is  */
/* ok, has microcode running on it, and is stopped. */
/*                                                */
/* ********************************************** */
     JPSTAT();
     If !(jp_ok)
        return(hardware_error);
     if !(uCODE_loaded)
        load_ucode();
     if !(compatable_machine) */ are the model ids the*/
        return(incompatable_ucode); */ same    */
     if !(jp_stopped)
        return(jp_is_not_stopped);

/* ********************************************** */
/*                                                */
/* Now set up part of the PPCB.  Put the JP into the*/
/* PPCB list.      Attach the JP to the LP in LPID. */
/*                                                */
/* ********************************************** */
     GSMEM(len_of_ppcb);
     ppcb.jpid = jpid;
     PPC.W_LIST[jpid] = *ppcb;
     ATTACH_JP(jpid,lpid);

/* ********************************************** */
/*                                                */
/*  Now start the JP at the address JPSWART.P,      */
/*  put the PPCB on the system JP list CP.W,        */
/*  and exit.                                     */
/* ********************************************** */
     JPSTART(JPSWART.P);
     CP.W[JPCNT] = *ppcb;
     return();    /* good return  */
```

One of the steps in initializing a JP is to allocate its unique memory. Each JP has a unique page 0 page, which is allocated from system memory. Using the JPID the system takes and multiplies it by the value of JPUMSIZE.W and adds the result to JPUMBASE.W to allocate the right spot for this JP's unique memory. For example, if this were JPID 2 the formula would be:

```
(JPID * JPUMSIZE.W) + JPUMBASE.W
(  2  * 10000) + 122532 = 132532
```

The next step is to validate and allocate these pages in the PTEs for those pages. Next the PTPs for these pages are created so the new JP can use them for its SBRs and PTPs. Once this is done, the pages are actually filled with data. Page zero is copied and modified from the mother processor. A second page is created containing pointers to the undefined interrupt-device, interrupt service routine, since the child cannot process hardware interrupts. In this page of interrupt handlers there are actually three devices that it can handle: (1) PIT, (2) Power Fail, and (3) Cross Interrupt. Once all this is done, the system builds a special control block and waits for it to be built. Once built, it creates the stack registers in the new page zero to use this new primary control block. The last thing done is to load the page tables into the SBR for this ring 0 so that the JP can run its software ATU. The JP is started and we go back to the mainstream of JPINIT.

If you wanted to look at the SBRs for a particular JP, you would have to do the following:

1. Take JPID * JPUMSIZE.W

2. Add it to the value in JPUMBASE.w

3. Add the value of SSBRTAB to the above

This will give you the SBRs for the particular JP. At this point you could look at all the pages in use by
this JP.

To find a particular JPs PPCB you would have to take its JPID and multiply it by two and add it to the value of PP.W to get the address of the PPCB. From here you can get to the LPCB of the JP by using the offset pointer in the PPCB for the LP. There is also an offset in the PPCB to get the JP state block. Or just using MYPPCB.W for the in that JPs page zero. The following figure illustrates the relationships.

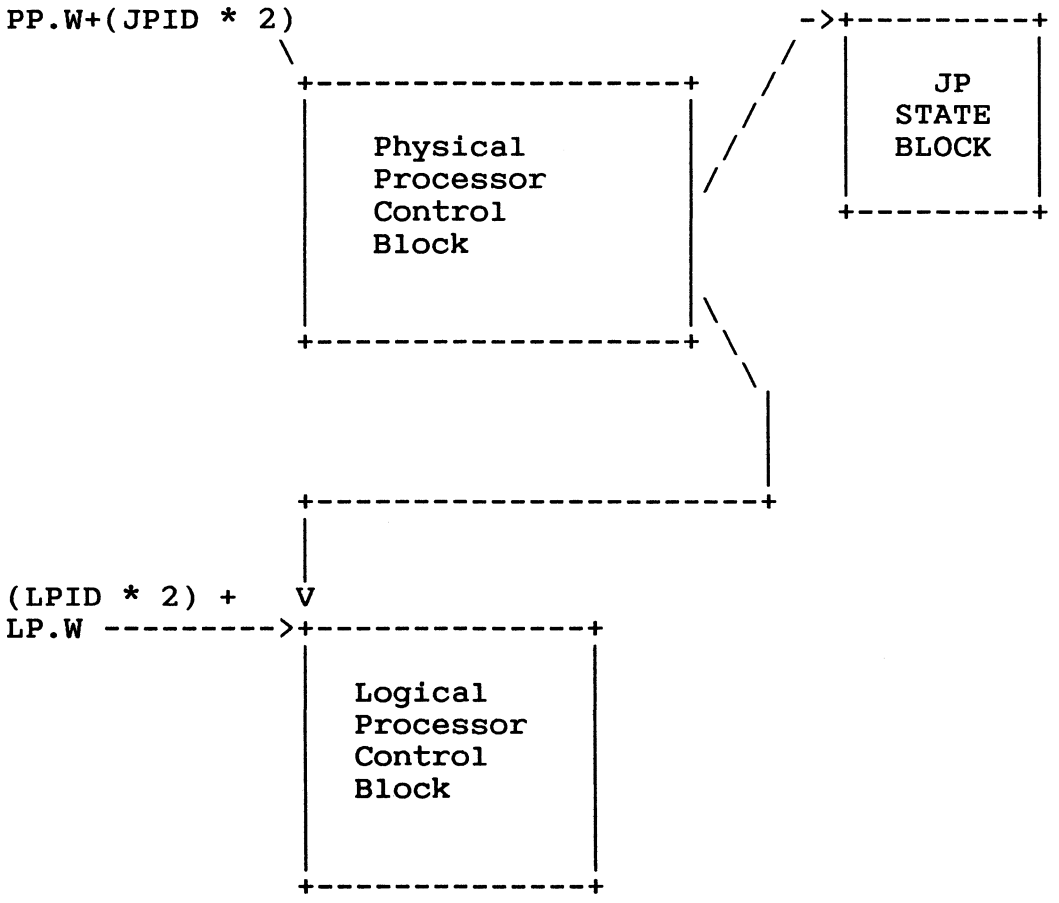


Figure 6.6 JP Database Relationships

6.9.2 ?JPMOV

Once you have created a JP it can be moved from one LP to another by executing the ?JPMOV system call. You might ask why would we want to move a JP from one LP to another LP? It is a case for giving more computer power to a particular class of user. This is a very useful way for the system manager to control his resources to give better service to the user community.

```
/* $$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$ */
/*                        JPMOV.P                                  */
/* This routine moves a JP from one LP to another.                 */
/* This routine can pend.                                          */
/*                                                                 */
/*                                                                 */
/* $$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$ */


/* This is the user packet supplied to the system call.*/
```

LOC	Name	Summary
0	pckid	Packet identifier
2	reserved	
3	flags	JPMOV flag bits
4	jpid	jp id for the JP being inited
5	Lpid	LPID of LP to assign the JP to

```
JPMOV.P(packet);
  {
/* ************************************************** */
/*                                                    */
/*  Does the user have the MP privilege? If not, exit.*/
/*                                                    */
/* ************************************************** */

     IF (!(syspriv))
        return(not_privileged);
     set_up_stack();
     Set_up_trap_handler();
     jpid = packet->jpid;       /* get packet */
     flag = packet->flag;       /* information */
     lpid = packet->lpid;
     Clear_trap_handler();
```

```
/* ************************************************ */
/*                                                  */
/* Check for a valid Jpid.  A valid jpid is an      */
/* integer between 0 and 15 octal.  If not valid,   */
/* return the "Invalid JPID".                       */
/* ************************************************ */

     if (!((jpid>=0)&(jpid<=15))
        return(invalid_jpid);
/* ************************************************ */
/*                                                  */
/*  Check for a valid lpid.  A valid lpid is an     */
/*  integer between 0 and 15.  If not valid, exit   */
/*  with the "Invalid LPID" error.                  */
/* ************************************************ */

     if (!((lpid>=0)&(lpid<=15))
        return(invalid_lpid);

     get_JPLPlocks();

/* ************************************************ */
/*                                                  */
/*  Find_jp is used to loop each time the routine   */
/*  pends to ensure the JP still exists.            */
/*  The findjp function is called to get a ppcb     */
/*  for the jpid supplied in the packet.            */
/* ************************************************ */

   find_jp:
     newppcb = findjp(jpid);
     if (error_rtn)
        return(JP_does_not_exist);
/* ************************************************ */
/*                                                  */
/*  If the JP is stopping, then the JP no longer    */
/*  exists in our eyes. Return an error.            */
/*                                                  */
/* ************************************************ */

     if(bit(new.ppcb.cpstop) == 1)
        return(jp_not_inited);
```

```
/* ************************************************ */
/*                                                  */
/*  If the JP is moving, then this routine must pend */
/*  waiting for the other JPMOV to complete.        */
/*  After pending, loop to find_jp.                 */
/*                                                  */
/* ************************************************ */

     if (bit(newppcb.cpmov) == 1)
       {
        pend(wait_on_jpmov);       /*  pend ... */
        goto find_jp;
       }
/* ************************************************ */
/*                                                  */
/*  Get the target LP.  If it doesn't exist, return */
/*  "LP does not exist".                            */
/* ************************************************ */

     newlpcb = findlp(lpid);
     if (error_rtn)
        return(lp_does_not_exist);
/* ************************************************ */
/*                                                  */
/*  If the new lp is the same as ours, then exit.   */
/*                                                  */
/* ************************************************ */

     if (newlpcb == lpcb)
        return();

/* ************************************************ */
/*                                                  */
/*  If this is the last jp on the old lp, then the  */
/*  error return is taken.  This is only done if the */
/*  user wishes to be sure that the LP has at least */
/*  one JP attached to it.                          */
/* ************************************************ */

     if ((bit (packet->flags,0) == 1) &&
        (PP.W[packet->jpid]->cplpcb.w->jplpcnt <= 1))
        return(cannot_rel_last_jp);
```

```
/* ************************************************** */
/*                                                    */
/*   If I'm moving myself then do it.                 */
/*                                                    */
/* ************************************************** */


     if (myppcb.w == oldppcb)
       {
        interrupts(off);
        detach(mylpcb.w,myppcb.w);
        attach(newlpcb,myppcb.w);
        mylpcb.w = newlpcb;
        mlpcb.w = newlpcb;
        interrupts(on);
       } /* if moving self */
     else
/* ************************************************** */
/*                                                    */
/*   Else move the other JP.  Idle_jp will pend until */
/*   JP is "idle".  Once the target JP is idle, then  */
/*   detached from the old LP  and attach to the new. */
/* ************************************************** */

       {
        bit(newppcb.cpmov) = 1;  /* move in progress */
        idle_jp(newppcb);
        detach(oldlpcb,newppcb);
        attach(newlpcb,newppcb);
       } /*  else  */

/* ************************************************** */
/*                                                    */
/*  Unlock the JPLP lock word and return.             */
/*                                                    */
/* ************************************************** */

     release_jplp_locks();
     return();
  } /* end of ?JPMOV */
```

6.9.3 ?JPREL

Until now we have discussed how to get a JP into the system, but how do we get one out of the system? This is done by issuing the JPREL system call. This call will release a JP from the system. It will not work on the "mother" processor. Care has to be taken to not leave an LP "unattached" when all of its JPs are released.

```
/* $$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$ */
/*                  JPREL.P                           */
/*  This call is used to release a JP from the        */
/*  system.                                           */
/*                                                    */
/* $$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$ */

JPREL.P(packet)
  {

/* This is the user packet supplied to the system call.*/

 LOC   Name       Summary
+---+----------+--------------------------------------+
| 0 | pckid    | Packet identifier                    |
| 2 | reserved |                                      |
| 3 | flags    | JPREL  flag bits                     |
| 4 | jpid     | jp id for the JP being inited        |
+---+----------+--------------------------------------+

/* ************************************************** */
/*                                                    */
/*    Make sure the user has the privilege to do this */
/*    call.  If not exit.                             */
/*                                                    */
/* ************************************************** */

     if (!(syspriv))
        return(caller_not_privileged);
     if (!(mp_system))
        return(not_mp_system);
     set_up_trap_handler();
     jpid = packet->jpid;
     flag = packet->flag;
     clear_trap_handler();
```

```
/* ************************************************ */
/*                                                  */
/*  If the jpid is invalid, exit.  JPID is valid if */
/*  it is between 0 and 15.  Get the LPJPlock.      */
/* ************************************************ */

     if (!(jpid>= 0)&(jpid <= 15.))
        return(invalid_jpid);
     get_lpjp_lock();

/* ************************************************ */
/*                                                  */
/*  Find_jp will be looped to if this code pends.   */
/*  Get the new ppcb from the supplied jpid.        */
/*  On error, exit.                                 */
/*                                                  */
/* ************************************************ */

  find_jp:
     newppcb = findjp(jpid);
     if (error_rtn)
        return(jp_not_inited);

/* ************************************************ */
/*                                                  */
/*  If the user is trying to release the mother, exit.*/
/*  If the processor is running a system task, exit.*/
/*  If the JP is already stopping, exit.            */
/*                                                  */
/* ************************************************ */

     if (newppcb == myppcb)
        return(can_not_rel_mom);
     if (bit(newppcb.cpsys) == 1)
        return(can_not_rel_during_sys_task);
     if (bit(newppcb.cpstop) == 1)
        return(jp_not_inited);
/* ************************************************ */
/*                                                  */
/*   If the JP is moving pend... waiting for move   */
/*   to complete.                                   */
/*   Go to find_jp.                                 */
/*                                                  */
/* ************************************************ */

     if (bit(newppcb.cpmov) == 1)
       {
        pend(mov_complete);
        goto find_jp;
       }
```

```
/* ************************************************ */
/*                                                  */
/*   Check to see if the user does not wish to have */
/*   the LP that the "being released" JP is the last */
/*   JP on that LP.                                 */
/*                                                  */
/* ************************************************ */

     if    ((bit(flag,0) == 1) &&
            (PP.W[jpid]->cplpcb.w->jplpcnt <= 1))
        return(can_not_rel_last_jp);

/* ************************************************ */
/*                                                  */
/*   Set the stop bit and "idle" the JP.            */
/*                                                  */
/* ************************************************ */

     bit(newppcb.cpstat.cpstop) = 1;
     idle_jp(newppcb);
     cp.w[jpid] = 0;
     JPSTOP;

/* ************************************************ */
/*                                                  */
/*   Wait for the JP to stop.                       */
/*                                                  */
/* ************************************************ */

     while (!(newppcb.saddr.stopped))
        JPSTATUS;
/* ************************************************ */
/*                                                  */
/*  Release the resources that the JP had (e.g., its */
/*  page zero, its PPCB memory).                    */
/* ************************************************ */

     release_jp_pages();
     detach(newppcb.lpcb.w,newppcb);
     dealloc_ppcb(newppcb);
     rel_jplp_lock();
     return();
  } /* ?JPREL  */
```

6.9.4 ?JPSTAT

We have talked about JP management until this point, but how does one see if there is problem with a JP or if it is running? There are watchdog timers associated with each JP in the scheduler to tell if a given processor is running or stuck in a loop/hang as well as panicking. There is a ?JPSTAT system call for this very purpose. It will give the status for general information or specific information.

```
/* ************************************************ */
/*                      JPSTAT.P                    */
/*  This routine provides status information about  */
/*  the jp.  This provides two kinds of status:     */
/*  1) The JPmap and JP count.                      */
/*  2) The type of JP and status info.              */
/* ************************************************ */


/* This is the user packet supplied to the system call.*/
```

LOC	Name	Summary
0	pckid	Packet identifier
2	function	Function code (type of status)
3	subpkt	Word pointer to subpacket (2 words)

General Information subpacket

LOC	Name	Summary
0	pckid	Subpacket identifier
2	jpcnt	# of JPs initialized
3	jpmap	Bitmap on initialized JPs (2 words)

Specific Information Subpacket

LOC	Name	Summary
0	pckid	Subpacket identifier
2	jpid	User supplied JP id
3	state	Current JP state
4	status	JP status bits
6	model	model # of the JP
7	uc_rev	Microcode revision (uc -> ucode)
8	flags	Flag bits
9	lpid	LP the JP is attached to

```
JPSTAT.P(packet)
   {
     if (!(mp_system))
        return(non_mp_sys);
     Set_up_trap_handler();
     subpkt = packet->subpkt;
     function = packet->function;

/* ************************************************ */
/*                                                  */
/*  Is the system call request a general call       */
/*  type 1 above or a specific call type 2 above.   */
/*                                                  */
/* ************************************************ */

     switch (function)
        {
/* ************************************************ */
/*                                                  */
/*   A general call.                                */
/*                                                  */
/* ************************************************ */

     case 0:

        get_jplp_lock(read);
        subpkt->jpcnt = jpcnt.w;
        subpkt->jpmap = jpmp.w;
        rel_jplp_lock(read);
        clear_trap_handler();
        return();
       }
/* ************************************************ */
/*                                                  */
/*   A specific call.                               */
/*                                                  */
/* ************************************************ */

     case 1:

        clear_trap_handler;

/* ************************************************ */
/*  Test for validity of the jpid in the subpacket. */
/*  A valid JPID meets the following rule:          */
/*              0 <= jpid <= 15.                    */
/* ************************************************ */

        if (!(jpid>=0) & (jpid <=15))
           return(invalid_jpid);
        get_jplp_lock(read);
        ppcb = findjp(jpid);
        if (error_rtn)
           return(jp_not_inited);
```

```
/* ************************************************ */
/*                                                  */
/*  Set up packet from the ppcb.                    */
/*  If "mother" set the mother bit in subpacket flags */
/*  word.   Store the lpid this jp is attached to in  */
/*  the subpacket.                                  */
/*                                                  */
/* ************************************************ */

        setbit(subpkt->flags,mom) = 1;
        lpcb = ppcb->cplpcb.w;
        subpkt->lpid = lpcb->lpid;

/* ************************************************ */
/*                                                  */
/*  Since the JPstatus instruction returns three    */
/*  pieces of info jpstate, jpstatus, and jpmodel,  */
/*  store this info in the user packet.             */
/*                                                  */
/* ************************************************ */

        JPSTATUS(&jpstate,&jpstatus,&JPmodel);
        subpkt->jpstate = jpstate;
        subpkt->jpstat  = jpstatus;
        subpkt->jpmodel = jpmodel;
        rel_jplp_lock(read);
        return();
```

6.10 System Services

The JP manager provides one service to the rest of Paths and time other than allowing JP resource usage. That service is system accounting.

Whenever a system function is done, timings are taken. (See Time Management.) These timings are in PIT ticks and must be added to the PPCB. The reason the PPCB holds these timings is that each JP runs in the system independently of the others. For example, when an interrupt happens each interrupt service routine saves the PIT reading in a base. When dismissing the interrupt another reading is taken. The difference between the base and the reading is the amount of time spent at interrupt level. This accounting is done for system time.

Chapter 7
Time Management

7.1 Introduction

7.1.1 Purpose

The purpose of this chapter is to present AOS/VS time management and how it relates to the rest of Paths and Time.

7.1.2 Overview

Time management is the part of Paths and Time that interfaces between the timing devices and the rest of AOS/VS. These timing devices are called the Programmable Interval Timer (PIT) and the Real Time Clock (RTC). These devices can be either real devices or implemented in microcode and are used to maintain different kinds of timing services.

The primary service that time management provides is supplying amounts of time to the requestors. These amounts of time are called ticks. A tick is a different amount of time for each device. For the PIT, a tick is 0.1 millisecond. For the RTC, a tick varies based on the gen of the system. The main users of time management, outside of paths and time, are Memory Management and the File System.

There are several relationships between the components of paths and time and time management. These relationships are shown in Figure 7.1 Time management provides services to JP management, LP management, entity management, and Process Management Interface Services. Time management is serviced by the JP management.

JP management provides interrupt-level service so the devices can interrupt. Time management provides JP management with timings for the counters in the JP's PPCB. Time management also provides timing services for LP and entity management in the form of timing device management. ?DELAY and subslice management done in process management interface services use each of the devices in time management.

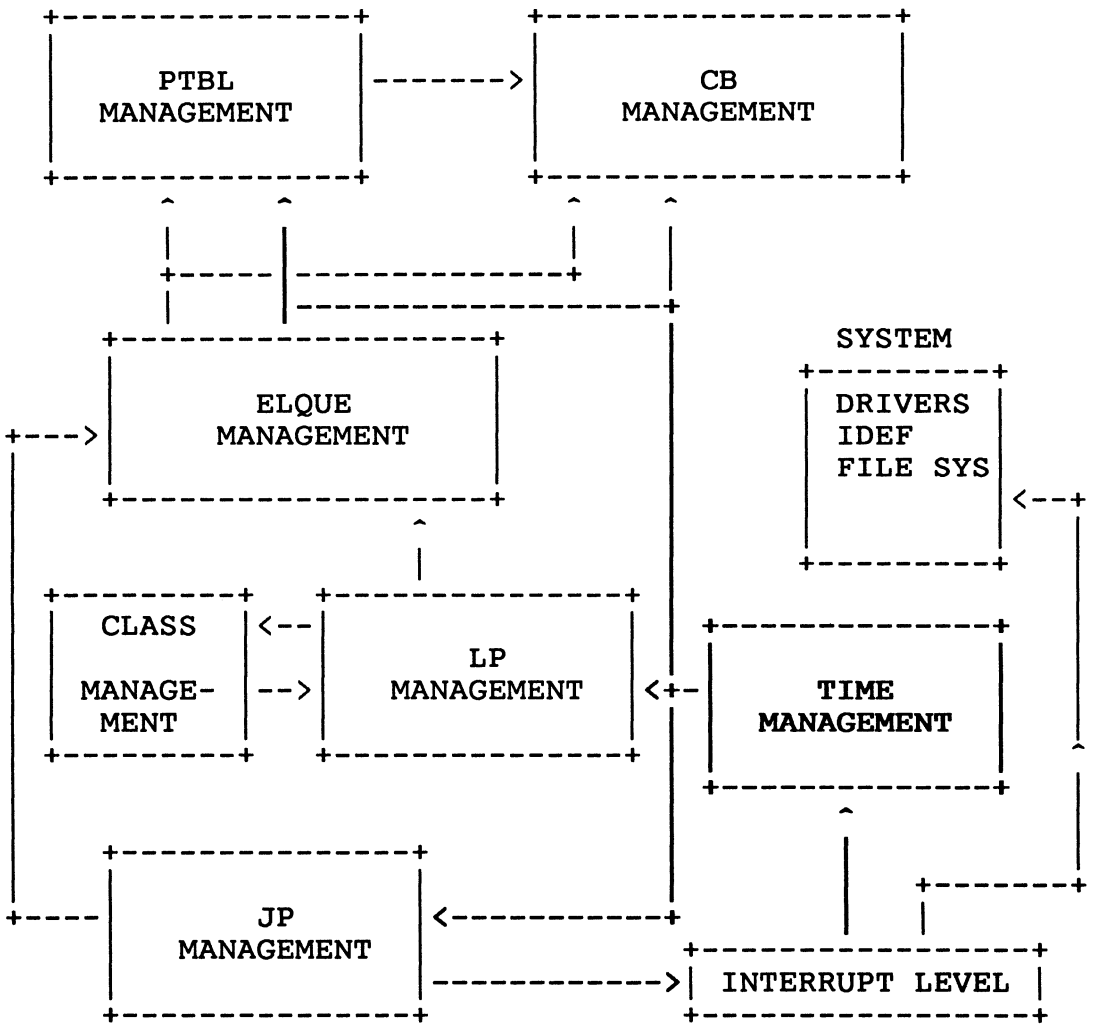


Figure 7.1

Time management is allowed to run interrupt-level code on the JP by JP management. Time management provides timing data to JP management for "in system" timings. Timing data is provided to LP management to enforce class scheduling. Time management provides time data to PTBL management to help PTBL management manage the users subslice and timeslice. Time management provides time data to CB management to allow CBs to be timed for JP purposes. (See PPCB in JP Management.)

7.2 The PIT

7.2.1 The Objects

The Programmable Interval Timer (PIT) is a programmed I/O (PIO) device that counts, in ticks, where one tick is equivalent to 0.1 milliseconds. The PIT uses a 16-bit counter. To access a PIO device, the program must use PIO instructions such as DOAS, DIA, etc. The PIT is assigned a negative number of ticks and counts upward until it reaches zero or gets stopped. For example, a process gets set up to run. Before it runs a process, the dispatcher starts the PIT with a negative value (e.g., -20). (See pseudocode, Section 7.2.4.2). When the PIT reaches zero, it will interrupt the system.

Each JP in the system has its own PIT and handles its own interrupts from its PIT. Because the PIT is used for scheduling subslice and timeslice operations, each JP must have its own PIT to allow each processor to schedule processes independently of other JPs.

All PITs are started with a negative number and count (tick) toward zero. At zero, the PIT will interrupt the system. After the PIT reaches zero, the PIT may behave differently depending upon the model of the MV. There are three different kinds of PITs depending upon the MV processor model the PIT is on. The three PITs are "stop at zero," "keep ticking," and "reset last value and keep ticking." To AOS/VS these PITs are a device, but in actuality the PIT may be implemented in microcode.

- The "Stop at Zero" PIT will interrupt then stop and will not continue until restarted.

- The "Keep Ticking" PIT will interrupt then continue beyond zero, thus going positive.

- The "Reset Last Value and Continue" PIT will interrupt and reset itself to the same value it was started with and continue. This may cause problems with time accounting. Because accounting could be affected by a PIT that restarts and continues, there has to be adjustments made in the operations that work with it. (See Operations.)

7.2.2 The Globals

To use the PIT to calculate time, some globals must be used. There are three globals used with the PIT: IBASE, SBASE, and TSLSV. Each JP has its own copy of these globals.

IBASE is an integer that contains the "enter interrupt level" sampling of the PIT. IBASE stands for Interrupt level BASE. When leaving interrupt level the difference between IBASE and the PIT reading is the time used at interrupt level.

SBASE is an integer that contains the "enter the system" PIT reading. SBASE stands for System BASE time. When leaving the system to go into user mode, the PIT is sampled again and the amount of time spent in the system is the new PIT reading - the contents of SBASE.

TSLSV is an integer that gets the amount of time in PIT ticks the user has left in its subslice. TSLSV stands for Time SLice SaVe variable. This value holds the PIT reading when entering interrupt level to save the state of the entity running at the point of interrupt. This will be used to restart the PIT when the interrupt gets dismissed.

7.2.3 Basic Operations

There are some basic operations performed on the PIT. In AOS/VS the operations are implemented in the form of macros. Some of these operations work with the different types of PITs. The basic operations performed on the PIT are:

```
I.PIT.S
RUN.PIT  (AC0)
STOP.PIT
LOOK.PIT
CHECK.PIT (pit_residue)
```

7.2.3.1 I.PIT.S

When an interrupt occurs (except for IPIT - See paths below), if we are at interrupt level 1, then the PIT is read using this macro. I.PIT.S is the only macro that uses globals. I.PIT.S reads the PIT and stores the value of the reading in two places. These places are TSLSV and IBASE. IBASE is used as the initial value to calculate the amount of time the system spends at interrupt level (this is used in JP management). TSLSV holds the PIT residue so that the state of the PIT gets restored when the interrupt-service routine is finished processing the interrupt.

7.2.3.2 RUN.PIT (Input in AC0)

The RUN.PIT function stores a value in the PIT and starts it. The value that this macro starts the PIT with is in AC0. This is used when dispatching a process or when an interrupt is dismissed. When dispatching a process, the PIT is started with the contents of PSL. (See CB or Ptable Management.) When dismissing an interrupt, the contents of TSLSV are supplied to the PIT.

7.2.3.3 STOP.PIT (Returns a value in AC0)

STOP.PIT reads and stops the PIT. This is used when a reading of the PIT is needed to update statistics and the PIT will be restarted with some other value. For example, when dispatching a user a STOP.PIT is used to read the PIT to update the amount of time the system has used (by subtracting the contents of SBASE from the current value held in AC0) and then does a RUN.PIT with the user's subslice residue (see above).

7.2.3.4 LOOK.PIT (Value returned in AC0)

LOOK.PIT reads the PIT without stopping it. This is used when entering the system. The PIT is read and the value is stored in SBASE. SBASE is used as the base for system timing. The reason the PIT is just looked at and not stopped and restarted is that stopping and restarting the PIT has too much overhead.

7.2.3.5 CHECK.PIT (PIT residue is in AC0)

CHECK.PIT reads the PIT, checks to see if the PIT wrapped since the last reading. This is done by checking the PIT to see if it has interrupted the system since the last check. This situation would exist if interrupts were off when the PIT fired (caused an interrupt). If the PIT has fired, then this operation checks to see if the reading (residue) is negative. If the reading is negative, then restart the PIT with a zero. The effect this has is it prevents the PIT from interrupting during a system activity. Otherwise, restart the PIT with the value in the reading.

7.2.4 Paths in the PIT World

There are two different environments or levels where the PIT is accessed. The first is called base level and the second is interrupt level. Base level is the normal running mode on a JP. When the PIT expires, it interrupts the JP, which puts the JP at interrupt level. The JP then makes decisions based on the JP state.

7.2.4.1 Base Level

At base level a number of functions can be performed on the PIT. The PIT can be started, stopped, or sampled.

- The PIT is started when a user is dispatched to run.
- The PIT is stopped when a transition from system to user is about to occur.
- The PIT is sampled when entering a major component of the system. This sampling is used for time accounting done in JP management. The way this is done is the PIT is sampled on entry into a component of the system (LOOK.PIT) and the value is stored into a BASE variable. When leaving the component, the PIT is sampled again and the amount of time used is calculated by subtracting the amount of time stored in the BASE variable and the result stored in the PPCB.

There are situations in which the PIT is less than the base value. In this case, the PIT has wrapped. Wrapping means the PIT has counted beyond the 16 bits that it is allotted. If the PIT wraps then the calculation for accounting is reversed.

The following pseudocode shows how the PIT would be used:

```
int  PIT_val;
GLOBAL int  sbase,ppcb.cpsys.4;/* see PPCB for cpsys.4 */
    enter_sys:  PIT_val = LOOK.PIT( );
                sbase= PIT_val;
                 DO_SYS_ACTIONS;
                PIT_val = LOOK.PIT( );
                if (PIT.val < sbase)
                  ppcb.cpsys.4 = sbase - PIT_val
                else
                  ppcb.cpsys.4 = PIT_val - sbase;
                LEAVE_SYS;
```

7.2.4.2 Interrupt Level

The PIT is read at interrupt level using the I.PIT.S function. This function will set up the variables IBASE and TSLSV for the interrupt dismiss (DISMISS) routine. DISMISS will restore the user's context, including TSLSV, and return to the user. The only exception to this is the PIT interrupt.

When the PIT interrupts the system, its counter has gone to zero. The following pseudocode shows what happens when a PIT interrupt occurs.

```
/* ************************************************ */
/*                                                  */
/* IPIT is the entry point the interrupt is dispatched*/
/* to when it occurs.  This routine stops and reads  */
/* the PIT, checks to see if we were running a user. */
/* If so, then process the subslice end.            */
/* ************************************************ */

    IPIT:     pit_val = STOP.PIT();
/* ************************************************ */
/*                                                  */
/*  If interrupts are off or the PIT is masked out, */
/*  the counter may have been incremented before the */
/*  interrupt gets serviced.  The PIT may go beyond 0.*/
/*  Therefore, check if it did and if so clear the  */
/*  PIT_VAL.                                        */
/*                                                  */
/* ************************************************ */

             if (pit_val >= 0)
               {
               pit_val = 0;
/* ************************************************ */
/*                                                  */
/*  If we were running in the system, save the PIT_VAL*/
/*  in the base and the restore area.               */
/*  The PIT can interrupt the system at any time.   */
/*  Since the PIT's only operation is the PIT       */
/*  incrementing to zero, and the PIT is not        */
/*  necessarily restarted upon entering the system, */
/*  the PIT can and does interrupt the operating    */
/*  system.                                         */
/*                                                  */
/* ************************************************ */
               if (SYSIN == 1)
                 {
                 IBASE = pit_val;
                 TSLSV = pit_val;
                 }
               }/* if >= 0 */
```

```
/* ************************************************ */
/*                                                  */
/*  Restart the PIT for interrupt-level timing.     */
/*                                                  */
/* ************************************************ */
            START.PIT(pit_val);
/* ************************************************ */
/*                                                  */
/*  If we were running at interrupt level 1, save  */
/*  interrupt level time accounting value.          */
/*                                                  */
/* ************************************************ */
              if (INTLV == 1)
                 IBASE = pit_val;
/* ************************************************ */
/*                                                  */
/*  If we were running a user, do the subslice end  */
/*  processing.  Call USLICE.                       */
/*                                                  */
/* ************************************************ */
            if (SYSIN == 0)
             {
                 USLICE(); /* see process scheduling*/

/* ************************************************ */
/*                                                  */
/*  If we were running a user and could do the      */
/*  subslice short cut, give the user a new subslice. */
/*                                                  */
/* ************************************************ */
                 if !(short_cut)  /* see USLICE   */
                   TSLSV = -320;  /* new subslice */
                 else
                   TSLSV = 0;
             }
            goto DISMISS;  /* dismiss the interrupt*/
```

7.3 The Real Time Clock (RTC)

7.3.1 The Objects

The real-time clock is a device that keeps real time. The RTC is the device that AOS/VS uses to keep the values of Time of Day and Current Date. The RTC will interrupt the system 10, 50, 60, 100, or 1000 times per second depending on the system gen. (See "How To Generate and Run Your AOS/VS System" manual.) Once started the RTC continues counting until it interrupts the system. The interrupt service routine then restarts the RTC.

The RTC is a PIO device. This means that it is started with a DOAS instruction. The device is started with the frequency it will run. The frequency is the number of times per second the RTC will interrupt the system. The value that the RTC is started with is not really the frequency, but some value in a variable called RTCI representing the frequency. (See Figure 7.2)

RTCI VALUE	FREQUENCY
0	50 OR 60 HZ DEPENDING ON LINE FREQUENCY
1	10 HZ (VSGEN DEFAULT)
2	100 HZ
3	1000 HZ

Figure 7.2 RTC Frequency

Once the RTC is started, its running can be VISUALIZED as a simple loop. This is not actual code from the RTC.

```
/* ************************************************ */
/*                                                  */
/*  This code shows the basic loop of the Real time */
/*  clock.  (This is an interpretation of the RTC)  */
/* ************************************************ */

          START_RTC(rtci);
    loop: cnt = cnt + 1;

/* ************************************************ */
/*                                                  */
/*  If the RTC has run to a point where it needs to */
/*  interrupt the system, do so and reset the counter */
/*  loop.                                           */
/* ************************************************ */
           if (cnt == rtci_setting)
           {
             INTERRUPT_VS;
             cnt = 0;
           }
           goto loop;  /* restart function in IRTC */
```

7.3.2 The RTC Globals

The Real Time Clock is considered a critical region because both base and interrupt level do things such as work with ?DELAYS. Therefore, a locking mechanism for the RTC must be used. The globals that exist for the RTC are used for locking and collision counting. The globals are: RTCLCK.W, RTCSPN.W, and RTCISPN.W. There is a fourth global called RTCTMP.W, which is used at base level to hold the return address of the caller.

RTCLCK.W is the lock double word used for the RTC. This lock is a spin lock (see JP management). This lock is used to keep interrupt level or base level from working with the delay chain while the other is doing work with it. The delay chain holds the processes doing ?DELAYs. The other reason for using this lock is that in an MP environment the system doesn't have to lose time because of interrupts being off during base-level operations with the RTC.

RTCSPN.W is an integer that holds the number of base-level collisions encountered with RTCLCK.W. This value is incremented each time a lock collision occurs.

RTCISPN.W is an integer that contains the number of interrupt-level collisions with RTCLCK.W. This counts the number of times that interrupt level encountered a lock RTC. This number reflects the number of times that RTC interrupt level has to spin on a lock.

7.3.3 Paths that Access the RTC

The RTC can be accessed at both base and interrupt level.

7.3.3.1 Base Level

At base level there are two ways to access the RTC: one is done at SINIT time and the other is done when a process wishes to go on the delay chain.

- At SINIT time the RTC is started with the value in RTCI.

- When a process wants to do a delay, the process is put on the delay chain. To do this in a UNI environment the RTC interrupt must be masked out. The reason for this is that in a UNI environment, base level can be setting up a delay and get interrupted by the RTC, which affects the delay chain. In the MP environment, the RTCLCK.W must be held because masking an interrupt on the daughter processor does not prevent the mother from being interrupted. This means that there are two levels of locks held on the RTC at base level. This locking mechanism seems to be redundant because the ?DELAY is a "mother only" call, but the mechanism is designed for when the daughter will also be allowed to handle interrupts.

To mask out the RTC interrupt, the RTC lock (RTCLCK.W) is used. If this lock is held, others must spin on the lock. If the unlock routine must spin, one of two counters are incremented depending on whether we are at base level (RTCSPN.W) or a interrupt level (RTCISPN.W).

7.3.3.2 Interrupt Level

The RTC interrupt service routine is called IRTC. It performs a number of services. Some services IRTC performs are at each tick and others are performed at each second. The services RTC performs allow other AOS/VS modules (e.g., memory management) to do timing functions. For example, at each tick IRTC calls memory management to do PFF timing. Memory management will make PFF decisions based on these timings. The pseudocode below shows what IRTC does and who it calls.

```
/* ************************************************ */
/*                      IRTC                        */
/*  This is the interrupt service routine for the real*/
/*  time clock.  At each tick this routine will      */
/*  increment the elapsed time counter and call the  */
/*  histogramming function, memory management, and   */
/*  process management services.  Each second this   */
/*  will call memory management, the files system,   */
/*  and process management interfaces.               */
/*                                                   */
/* ************************************************ */

IRTC()    I.PIT.S  /* see PIT section  /*
  {
/* ************************************************ */
/*                                                   */
/*  If the RTC is locked then someone is doing some  */
/*  operation with the RTC and, therefore, spin on   */
/*  this lock.  Spinning assumes that the lock will  */
/*  not be held very long.                           */
/*                                                   */
/* ************************************************ */
    if (bit(rtclck.w,0) == 1)
         {
         rtcispn.w ++;
         while(bit(rtclck.w,0) == 1)
              {}
         }
    setbit(rtclck.w,0);
         RESTART_RTC;
/* ************************************************ */
/*                                                   */
/*  If we are PFFing call memory management to do PFF */
/*  timing update.                                   */
/*                                                   */
/* ************************************************ */

    if (PFF_FLAG)
         MEM_MANAGE(do_pff);
```

```
/* ************************************************ */
/*                                                  */
/*  If a second has been used, then call process    */
/*  management interface services (PROC_MAN_INT_SERV) */
/*  to update time of day.                          */
/*  We need the Time of Day lock to do the checking. */
/*  Memory management is called to do PFF and state */
/*  3 to state 4 checking and conversions.          */
/*  The file system is called to do disk histogramming*/
/*  and device timeouts.                            */
/*                                                  */
/* ************************************************ */

    INTERRUPTS_OFF();  /* See JP management */
    LOCK_TOD;   /* lock Time of DAY */
    ticks_per_sec --;
    if (ticks_per_sec == 0)
         {
          ticks_per_sec = total_ticks_in_sec;
          PROC_MAN_INT_SERV(sec_elapse);
          MEMORY_MANAGEMENT(do_state_3_4_work);
          FILE_SYSTEM(do_disk_histo);
          FILE_SYSTEM(do_device_timeouts);
         }
    UNLOCK_TOD();      /* unlock Time of Day */
    INTERRUPTS_ON();  /* See JP management */

/* ************************************************ */
/*                                                  */
/*  Each tick PROC_MAN_INT_SERV must be called to do */
/*  histogramming.                                  */
/*                                                  */
/* ************************************************ */

    PROC_MAN_INT_SERV(do_histo);

/* ************************************************ */
/*                                                  */
/*  If there are any PTBLs on the delay chain, then */
/*  call PROC_MAN_INT_SERV to handle any delays.    */
/*  Release the RTC lock and go to DISMISS.         */
/* ************************************************ */

    if (outstanding_delays)
         PROC_MAN_INT_SERV(do_delays);
    clearbit(rtclck.w,0);
    DISMISS();   /* Dismiss interrupt */
```