To: Distribution

From: R F Mabee

Date: 11/26/74

Subject: A Proposal for Processes to be Used in the Supervisor

Summary

This memo proposes that a facility to provide special processes for use within the hardcore supervisor be made part of the standard Multics system.

The introduction shows why a special class of processes should be available to the supervisor, and how these processes must differ from the standard processes. The next section describes the actual implementation at a moderate level of detail. The last section presents a scheme for using such a process for the TTY interrupt handler.

A glossary of jargon terms is provided, as Appendix V.

This facility has been implemented and tested in an experimental version of the Multics system. Work is underway by several people to make use of these processes to simplify certain areas of the hardcore supervisor.

Introduction

Multics currently makes no use whatever of multiprogramming within the supervisor. This results in highly convoluted coding in many parts of the system where a module running in any one process tries to multiplex itself so part of its algorithm seems to be executed asynchronously. For example, the TTY Device Control Module (DCM) simulates a process for each terminal, with its own scheduler and undocumented synchronization facility. In many other cases, something is done in-line that doesn't really need to be done synchronously. For example, in the page fault path the faulting process currently checks the paging device to see if it is getting too full, and if so moves some pages to disk. This causes an unnecessary delay for the faulting process, and requires the page-moving algorithm to execute in a severely

limited environment (fault-side, interrupts masked, can't wait for I/O or locks). For another example, some I/O interrupt handlers currently execute long programs (taking up to two CPU seconds) in the same severely limited environment, requiring complicated (undocumented) conventions for co-operation with the processes that requested the I/O.

One can view the page-moving program or interrupt handler as a special kind of process that has absolute priority (it always runs to completion) but must run in a limited environment. By locks or by masking, the programs ensure a single sequential flow of control, as by:

```
check_paging_device: procedure ();
     set local lock;
     if should_run then run;
     unlock local lock;
     return;
     end;
```

A program like this can be made into a real process. The preceding fragment might become:

```
paging_device_process: procedure ();
     while true do;
               wait for wakeup;
               if should_run then run;
               end;
     end;
```

and a call to check_paging_device would become a call to send a wakeup.

In summary, there are three reasons why a program may need one or more dedicated processes: first, the algorithm may require a process per device, as in the TTY DCM; second, it may be inconvenient to perform some complex operation in the limited environment in which one happens to discover that it needs to be done; and third, it may be inefficient to perform the operation in the critical path in which one happens to discover that it needs to be done. The last point is meant to include the case of a program that requires more CPU time than one process can get, in order to scale up its performance in a very large system.

These problems are shared by programs in all rings, both user programs and system programs; however, I shall attempt a solution only for the hardcore supervisor (ring zero). Let us assume that processes are readily available in ring zero for any purpose, and examine some likely applications to get a feel for the properties such processes must have. This choice of examples does affect the resulting design.

The handler for any external interrupt could run in a

process of its own, and the interrupt would merely cause a wakeup. Where interrupts are multiplexed (as by the IOM) each channel's handler could have a process. Such a process would be started when the I/O device (or whatever) was attached, and would destroy itself when the device was detached. Its program should be specified when it is created; if the program is shared (e.g. printer driver shared by all printer processes) then an argument to the program should specify which device to run. This leads to the primitive

FORK (procedure, argument)

which creates a new process that starts with the call

procedure (argument)

and the primitive

DESTROY_ME ()

which stops and obliterates the process which calls it. Clearly the handler needs to block while awaiting the next interrupt, so a full set of IPC primitives should be available to it. The program should be allowed to use the virtual memory (take page faults) so it can run in a more normal environment, and avoid the expense of wired code and data. The scheduler should provide as fast response as the I/O device may require.

Another application is in resource managers to remove pages from core or from the paging device, to remove segments from the AST, to remove processes from the eligible list or from the APT, etc. Such processes must be created very early in initialization, when the function they help implement is not yet usable by FORK. Thus page faults are not allowed in creating, scheduling, or running a page control process.

These examples show processes that still run in a somewhat limited environment: they must not use the facility that they are implementing, and must be trusted by the supervisor because they must execute entirely in ring zero. Finally, using processes in any application has to be competitive in "cost" so that no programmer has to choose between readability and efficiency.

An ordinary process of the sort currently created for each user could meet most of these requirements, with suitable changes to keep it in ring zero. However, it is cumbersome, and has features which cannot even be initialized by the creating process until system initialization is nearly complete -- for example, it has a per-process directory (PDIR) which clearly cannot be created until page control, segment control, and the file system are all in operation. A simpler type of process must be introduced for use inside the supervisor. Let us dub the new type H-process and the old (ordinary) type M-process, for this

discussion. As a design goal, I choose to make the H-process as simple as is consistent with providing a normal program-execution environment. This should also minimize the "cost" of the H-process. The approach taken is to strip away all costly features that don't seem to be needed by all processes. By and large, the H-process could regain a feature by explicitly initializing it.

First, an H-process can run only in ring zero; thus we can eliminate the stack array used by the ring-crossing hardware. The programs it can run are totally pre-linked; the linker is unused and may be disabled. The address space could only be extended for data segments and only by explicit calls. Here is a very definite design choice: I choose to disallow this extension of the address space, in consequence of which I discard the KST. This means that the process can never take a segment fault; it can't use the file system; it can address non-hardcore segments only through explicit calls on segment control. Now the PDIR can't be touched, so discard it; it normally contains a segment called PIT by which the system passes initial arguments to a M-process -- discard this too, using a few words in the PDS for the (greatly reduced) initial conditions. At this point, only two per-process segments are left, PDS and DSEG, without which the H-process could not run at all. We have reduced the cost of the H-process to four pages + two ASTE's; Appendix II describes a way to reduce the cost to one page + one ASTE.

Notice that I have removed features by removing data bases. The features that are left, such as inter-process synchronization, paged memory, etc. seem to have very little incremental (per-process) cost, perhaps because their data bases and code are global.

An H-process can take page faults, service interrupts, and compete with M-processes in the scheduler's queues. The restrictions on it are less severe than those on fault-side or interrupt side programs which it might replace. It can totally avoid taking page faults (e.g. for a page control process) by executing only in wired-down code, and can therefore be used as deep in hardcore as required. However, it is poorly suited to the outer layers of the supervisor since it can't readily use the file system, and therefore can't interface to user processes. M-processes should be made available for outer-level applications in the supervisor (including ring one), and for user applications, but that is outside the scope of this project.

Details of proposed implementation

Multiprogramming is provided by pxss, using tc_data as the principal data base. It must be turned on by execution of tc_init before it will function normally; however, pxss$wait and related entries are simulated during initialization by looping in wired_fim. tc_init is currently invoked very late in initialization, so that page control (as a test case) cannot use multiprogramming. I propose to call tc_init early in Collection One, before page control is initialized. In this environment, all segments are unpaged and in core. This state is called the high-water mark because the core requirement is at its maximum.

tc_init contains two steps: first, initialize all the threaded lists and other data in tc_data; second, create the initializer process and all idle processes. The first step does not involve any references to data or procedures not present in Collection One, and therefore causes no problems. The second step starts any extra CPU's, and creates a PRDS for each such CPU, as well as a PDS and DSEG for each idle process. Let us assume that the extra CPU's are not started until late in initialization (to avoid two-cpu bugs); the remaining problem is the creation of two new segments for the single idle process. Any additional processes which may be created (e.g. for page control) will also require two new segments. The initializer process gets to keep the original PDS and DSEG.

Other conditions to be met in order for pxss to perform properly: those faults and interrupts used by pxss must be set up; a number of routines and data segments must be moved into Collection One; FORK and DESTROY_ME subroutines must be provided. However, the only problems worth further discussion arise from the requirement for a segment-creating primitive available to process creation, which must be able to work even before paging is available.

Segments (for PDS or DSEG) could be created unpaged initially, like segments read in during Collection One; however, update_sst_pll, which makes segments paged later on, would have difficulty finding the new segments. Any time after init_sst is run (which is very early) a paged segment can be created, taking a free ASTE and free page frames from appropriate lists. Existing page control entries could be used to create and wire pages; this approach was taken in the first experiments. However, these entries (e.g. wire_wait) ought not to be invoked when page control is not yet initialized -- if, for example, no free page existed, they might reference the FSDCT before it is addressable.

A new subroutine, GETSEG, will be written, to be used during both initialization and normal operation. It will get an unthreaded ASTE and (if during initialization) will assign page frames. It will not wire the pages; that remains the caller's

responsibility.

It is essential that there be sufficient core left when Multics is at the high-water mark for several tasks to be created. This requirement is about four pages per task. The high-water mark is already very close to the 128K minimum size of Multics main memory, but testing can proceed using a 256K system. Appendix I describes one way to reduce the high-water mark, by removing segments from Collection One.

Of course, wiring down more pages of core will of necessity degrade system performance. Most PDS's and DSEG's can be unloaded by traffic_control, but at least some hardcore tasks won't allow that. It is useful to reduce the memory requirements of H-processes to reduce the impact on system performance and on the high-water mark; Appendix II describes a scheme for shrinking the per-process segments. Each H-process also costs two small ASTE's for its private segments, and one APTE, amounting to 64 words of core. Since the AST and APT can readily be made larger, this cost is important only for applications requiring hundreds of H-processes.

Some increase in overhead of traffic control should be expected, due to more frequent interactions by H-processes. This loss of throughput can be countered by a better implementation of the process-switcher. The only other performance degradation to be expected is an increase in response time when interrupt-side programs are moved into supervisor tasks, and this would probably not affect system throughput. On the other hand, system throughput may be improved by moving certain housekeeping functions out of critical paths and by making use of multiple CPU's in bottleneck areas.

An H-process may demand very fast response, which should be controlled by a priority attribute used by pxss. Such an improvement is not part of this proposal, since acceptable performance can be achieved by using a different WAIT entry that guarantees fast response. Nevertheless, it has to be done sometime. Some scheduling requirements may not be adequately expressable by static priorities. This is an example of a limitation in pxss that may prevent optimum performance; such problems become more complex as more processes co-operate on particular computations.

Moving TTY DIM interrupt side processing into an H-process

Currently the Datanet-355 front-end processor returns status events by sending Multics a particular interrupt. The handler for this interrupt, dn355$interrupt, examines a mailbox at location 1400 to find the status word, performing an involved inter-computer ritual. For each status word it calls tty_inter. Every three seconds pxss calls tty_inter$poll, in case there aren't enough interrupts to drive the program. There is an interlock between tty_inter and tty_inter$poll so both are not active at once.

It is possible to restructure this as follows: The handler for the 355 interrupt, tty_wired$interrupt, merely sends a wakeup. A dedicated H-process, executing dn355$tty_process, receives the wakeup, then performs the inter-computer ritual and calls tty_inter as required. Every three seconds pxss calls tty_wired$poll, which sends the same wakeup and sets a flag. If dn355$tty_process finds the flag set, it calls tty_inter$poll. dn355$tty_process goes blocked when it runs out of work to do.

This scheme permits dn355, tty_inter, their utility modules, and two data bases to be unwired, releasing about ten pages of core. No further change is required except to fix a locking strategy that only works when interrupts and page faults are <u>not</u> allowed. All other interrupt handlers get better response since they no longer have to wait while tty_inter runs. (tty_inter takes up to two seconds; to make matters worse, the 355 is assigned the highest priority interrupt cell.)

On the other hand, each 355 interrupt might page in all ten of the pages we just unwired, plus two pages of stack. The extra core is really available only when 355 traffic is light. Furthermore, the TTY DIM will respond more slowly to interrupts, since the scheduler imposes a considerable delay. This is a serious problem since the TTY DIM is optimized for 1050-type terminals that require program intervention to go from writing to reading; the program ignores characters typed in before it changes its internal state from writing to reading even if no external action was required. The user with a non-locking keyboard may begin typing before the TTY DIM begins listening, even in the current system.

This problem can be solved without delving into the 355 code: the write DCW list created by tty_inter could chain into the read DCW list instead of terminating. This would result in a noticeable improvement even over the current system and make TTY process response relatively unimportant.

The restructuring (but not the DCW list chaining) has been done and tested in an experimental system, using the improved WAIT' (see Appendix III). Response time was found to average .2±.2 seconds worse than that of the standard system. The

experiment should be performed again to refine this measurement.

Appendix I
Reduction of the high-water mark

Currently, Collection One would nearly fill a 128K system, leaving insufficient room to create an H-process. Fortunately, there is an abundance of code and data that is not needed in Collection One, but is loaded at that time for historical reasons. The following table indicates which segments can be easily moved into Collection Two, at some cost in page breakage for those which are currently kept unpaged. Of course, segment_loader would be modified to implement the "wired" attribute for Collection Two.

Name	Size	Problems if moved to Collection Two
dn355_data	1050	
dn355	2136	scs_init copies address of $inter.
dn355_util	16	
dn355_init	422	Called too early by init_collections.
tty_ctl	2694	
tty_free	640	
tty_inter	3604	pxss calls $poll.
gioc_stat	174	
printer_status	264	
tdcm_status	446	pxss calls $poll.
imp_status_wired	26	
pl1_operators	2714	Fixed in standard system.
disk_traffic_data	1024	Touched by device_control$init.
temp_copyseg_1	1024	
get_disk_meters	246	
restart_fault	202	Stored into by initialize_faults.
return_to_ring_zero_	48	Stored into by initialize_faults.
DST	762	pxss uses DST to find end of ITT!
total words	17492	

For the immediate future my core needs can be met by these simple changes. If it should become necessary, I can remove another 11K of wired I/O buffers that are loaded with Collection One in order to remain unpaged. These could be loaded later if either unpaged segments could be loaded in Collection Two or DCM's were modified to allow discontiguous buffers. Another 8K is redundant, since the combined and uncombined linkage segments both sit in core when at the high-water mark. In all, up to 36K can be recovered as needed, with varying effort. This allows modest proliferation of hardcore tasks without undue difficulty. Since the currently available development system has 256K of main memory, the high-water mark is not an obstacle to development.

Appendix II
Reduction of memory requirements for H-processes

The additional memory for each H-process is required for the per-process segments PDS (about three pages) and DSEG (one page). The PDS contains 1400 words of fixed-format data, most of which is of no use to an H-process, plus the execution stack which may be less than a page for a simple-enough task. The bulk of this data can be moved to a new per-process segment, process_info, which need not even exist for an H-process. Of course, this costs one small ASTE per M-process, reducing the paging pool by one page in all. The PDS will contain a minimum of fixed data (about 200 words), leaving enough room in the first page for a minimal execution stack. The following table lists the items of the current PDS that an H-process will keep in its stack. (Another 30 words will be added for new features.)

Item name	Length
stack header	48
last_sp	2
processid (process_id)	1
lock_id	1
process_group_id	8
validation_level	1
apt_ptr	2
arg1, arg2, arg3, arg4	8
time_1, time_2, time_v_temp	6
post_purged, pc_call, wakeup_flag	3
delayed_stop, delayed_timer	2
pre_empt_pending	1
interaction_switch	1
virtual_time_at_eligibility	2
quota_inhib	1
base_addr_reg, pl1_machine	2
fim_data, page_fault_data	96
virtual_delta, cpu_time	4
number_of_pages_in_use	1
page_waits, pd_page_faults	2
dstep	1
ips_pending	1
ips_mask	8
auto_mask	8
alarm_ring	1
ring_alarm_val	8
trace (truncated)	16
total	235

Currently, references to the PDS are prelinked, and therefore corresponding data items must appear in the same location in the PDS in each process. It is sufficient for our

goals to continue using this fixed data layout for the PDS and PROCESS-INFO, but it may later prove too inflexible. An H-process should be able to grow by adding to itself some of the features normally associated only with an M-process. In order to avoid reserving large blocks of data in all processes' stacks for features that only some use, we could reserve a relatively small block of pointers, accessed by name, that would point to the data items allocated in whatever segment is most appropriate. The Network software already uses such a scheme -- its only cell in the PDS contains an index into a system-wide table.

The DSEG is currently a paged segment of which only about 256 words are used for hardcore segments. Clearly it can be made an unpaged segment if core control is made able to handle such; alternatively, page size could be reduced to 256. But closer examination of the DSEG suggests an even more fascinating solution: the only SDW's for which our hardcore DSEG differs from the template are those for the PDS, PRDS, the DSEG itself, and several abs-segs. This suggests that we can save core (at the expense of simplicity) by fabricating the DSEG whenever the process is to be run. The SDW for the PDS can be saved in the APTE; the PRDS SDW is already being patched every time an LDBR is done; the DSEG SDW would not be changed since it would always point to the scratch DSEG it lies in; and the abs-seg SDW's can be saved in the PDS. This can be thought of as sharing the current idle process DSEG with other H-processes.

Combining these tricks can reduce the per-process memory requirements by almost 75% for the hardcore-only tasks.

Both of these changes have been made and tested in an experimental system.

Appendix III
Miscellaneous changes required by this system

A. Descriptor segment creation.

A DSEG is normally created by plm$hc, which in the current system copies the hardcore-segment-number portion of whatever DSEG it is running with. (template_dseg is still being initialized but is never used.) plm has to be moved into Collection One, modified to run before paging is available, and modified to use the SLT to determine which SDW's to copy.

If plm$hc is invoked early in Collection One, it produces a DSEG with the segments unpaged. A routine, set_sdw_in_all_dsegs, has to be provided, to be called by update_sst_pl1 whenever it changes an SDW in the initializer's DSEG with the intention that it affect all address spaces. segment_loader, initialize_dims, and delete_segs can use set_sdw_in_all_dsegs too.

One field in a DBR value contains a segment number for an array of stack segments, for use in automatic ring crossings. Fortunately an H-process doesn't need this field. Its value is not determined until all segments are loaded, at which time init_sys_var fills it in for the initializer; init_sys_var has to be changed to set it in the APTE and in the register.

These changes have been made and tested.

B. PDS creation.

build_template_pds copies a stack header and a stack frame into template_pds; in so doing it messes up the initializer's stack. This module is eliminated, since the header can be merged with the template by those programs that create new PDS's. build_template_pds very cutely initializes the stack such that a "return" will transfer control to init_proc, the normal M-process starting point. However, pxss has to observe that it is running a process for the first time in order to do the proper return. The requirement that an H-process start in an arbitrary procedure forces a change: pxss executes "call stack_0$first_proc (stack_0$first_arg)" in the special case instead of "return". It turns out that init_processor receives control in this way when it starts the bootload CPU, since to pxss the initializer process looks like it has never run before.

bootstrap2 can't initialize the pointer to signal_ in the stack header (although the comment says it does) so initialize_faults$fault_init_two does it later. By moving signal_ into Collection One, this can be cleaned up.

These changes have been made and tested.

C. Control flags.

Assorted flags have to be added to the APTE: .hardcore_process, .use_hardcore_dseg, and .always_loaded. If .use_hardcore_dseg then the .dbr cell is really an SDW for the PDS. Other flags have to be added to wired_hardcore_data: $page_fault_works, $segment_fault_works, and $init_segs_gone.

D. Certain deficiencies in the scheduler.

One little known property of the current scheduler is that a process cannot lose its absolute priority (eligibility) unless it either takes a timer-runout/pre-empt interrupt while running in an outer ring or explicitly calls BLOCK. Since part of BLOCK is outside of ring zero and therefore not available to an H-process, and since interrupts are masked while running in ring zero, an H-process will keep its eligibility even if it uses WAIT, the normal ring-zero synchronization method, and will attain the highest possible priority. (If any process loops in ring zero, it will tie up the CPU forever.)

Allowing loss of eligibility by pre-emption in ring zero has other implications, requiring that eligibility be given up by WAIT because of assumptions embedded in all hardcore locking strategies, etc. I performed some experiments in this direction, concluding that even if I could find all the ramifications of such a change, including re-tuning the system, the change would have to be made and defended separately. This area remains open to anyone wth a particular interest in performance effects.

For the H-process running the TTY DCM, WAIT was an unsatisfactory synchronization primitive, as it left the process loaded and eligible indefinitely. I could have introduced a slightly different version, WAIT_and_do_what_I_want, but instead I adapted a different fundamental mechanism originated by David Reed, that has been advocated as a primitive capable of implementing both WAIT and BLOCK. Reed will soon publish an RFC describing his model, so I shall merely describe the implementation.

A shared memory cell is used to pass the information as to whether or not an event has occurred. This cell is provided by the caller of WAIT' or NOTIFY', which is not inconvenient when a shared data base exists anyway, and which avoids the allocation problems of WAIT and BLOCK. The cell changes to a new state (in fact it is incremented) every time the event occurs (every time NOTIFY' is called). WAIT' is given both the cell and the state it had when the caller first decided to wait; it returns whenever the cell contains some newer state. A list of processes waiting in this way is needed, so NOTIFY' can awaken them. In my implementation, the cell must be wired (since it is examined under the APT lock) and at the same address in every process

using it (since the address is used as a readily-available unique identifer).

NOTIFY' always awards high priority to an awakened process to improve response to interactions. Since the average delay for the process to become eligible in the normal way is three seconds (unless system load is very light), I had to make NOTIFY' award eligibility as well. The process will run as soon as the lowest-priority running process leaves ring zero and gets pre-empted.

For best response, the pre-empt should be allowed even in ring zero. Most of the problems with pre-emption in ring zero can be avoided if the pre-empt doesn't take away eligibility, but merely causes the highest-priority process to regain the CPU. This scheme should be tried as it should make TTY process response adequate for emulation of interrupt-side behavior.

The response time should be determined by a priority parameter associated with the process rather than by which WAIT the process calls. Future applications of H-processes will make such a feature in pxss more desirable.

The WAIT' primitives have been tested in an experimental system.

Appendix IV
Calling sequences of new routines

A. FORK

Usage:

```
declare create_supervisor_task entry (char (*), entry (pointer),
                                      pointer, bit (36), bit (36));
call create_supervisor_task (group_id, F, arg_pointer,
                                      return_proc_id, return_code);
```

1) group_id — is process group name of new process. (Input)

2) F — is starting procedure of new process. (Input)

3) arg_pointer — is passed to F in the new process. (Input)

4) return_proc_id — identifies the new process. (Output)

5) return_code — is zero if no error occurred. (Output)

This interface is intended to remain changeable so that additional features can be put in, such as an indication that the tricks described in Appendix II are to be used.

This entry creates an H-process and starts it running. The call to F in the new process is equivalent to:

```
call F (arg_pointer);
```

F must not be an internal procedure. The data pointed to by arg_pointer must not lie in a per-process segment (such as the stack).

Entry: create_supervisor_task$make_process

```
declare create_supervisor_task$make_process entry (1 like sdw,
                    char (*), bit (36), pointer, bit (36));
call create_supervisor_task$make_process (pds_sdw, group_id,
                    return_proc_id, return_apt_ptr, return_code);
```

1) pds_sdw — is an SDW describing the PDS to be used by the new process. (Input)

2) group_id — as above. (Input)

3) return_proc_id — as above, except right half must be set by caller. (Input/Output)

4) return_apt_ptr points to the newly created APT entry. (Output)

5) return_code as above. (Output)

This entry provides a process in the stopped state. It is used in creating the idle process.

B. DESTROY_ME

This entry does not yet exist as it is not needed.

C. GETSEG

Usage:

```
declare get_segment entry (pointer, fixed binary,
                           1 like sdw aligned, bit (36));
call get_segment (template, length, return_sdw, return_code);
```

1) template is a pointer containing a segment number which can be used to look up segment attributes in the SLT. (Input)

2) length is the number of words which must be created. (Input)

3) return_sdw is an SDW which describes this segment. (Output)

4) return_code is zero iff the operation succeeded. (Output)

For example, a PDS for a new process may be created by:

```
call get_segment (addr (pds$), pds$copy_length, pds_sdw, code);
```

D. WAIT' and NOTIFY'

Entry: pxss$wait_on_counter

```
declare pxss$wait_on_counter entry (fixed binary, fixed binary,
                                    fixed binary (71));
call pxss$wait_on_counter (event_cell, last_state, timeout);
```

1) event_cell is the shared state cell. (Input)

2) last_state is a saved copy of the event cell. (Input)

3) timeout is an upper bound on wait time. (Input)

Notes

timeout is not presently implemented; it is a placeholder. This entry is normally used as follows:

```
L:      last_state = event_cell;
        if should_run then run;
        else call pxss$wait_on_counter (event_cell, last_state,
                                                        3e6);
        goto L;
```

Entry: pxss$step_counter

```
declare pxss$step_counter entry (fixed binary);
call pxss$step_counter (event_cell);
```

1) event_cell as above. (Input/Output)

This entry is used to record the occurrence of an event.

Appendix V
Jargon explained

PDS stands for Process Data Segment. It contains data blocks that once were in three distinct per-process segments (pds, pdf, and process_info). Some of the data must remain in core, so the first page is wired as long as the process is eligible. The data items are referenced through links (e.g. declare pds$apt_ptr external;) so they must have the same virtual address in all processes, although the data is per-process. This is accomplished by using the same segment number in each process for the per-process segment, and by having the same data layout within each segment. The PDS also serves as execution stack for ring zero for both call-side and fault-side programs. So that the ring-crossing hardware will work, the PDS is also reachable by another segment number which is the first in a group of eight reserved for stacks.

PRDS stands for Processor Data Segment. There is one PRDS for each CPU in the system. It contains a fairly small data block and an execution stack for those faults and interrupts that must not cause further faults, e.g. page faults and I/O interrupts. The entire PRDS is wired down.

DSEG stands for Descriptor Segment. This is used by the hardware to map segment numbers into segments; it defines the address space. It may be thought of as a set of hardware registers. The first page of the DSEG is temp-wired whenever the process is loaded.

The machine instruction LDBR is used to switch the CPU to a new DSEG described by a given DBR (Descriptor Base Register) value.

SDW stands for Segment Descriptor Word. Each entry in the DSEG is an SDW for one segment. The SDW merely points to the page table for the segment, or specifies that a segment fault is to be caused.

An abs-seg is a reserved hardcore segment number for which a null SDW is present most of the time. Supervisor programs fabricate an SDW, stick it in the DSEG, reference the segment for a while, then clear the DSEG slot. This is useful in getting around such problems as addressing directories not known in this address space.

ASTE stands for Active Segment Table Entry. The primary content of the ASTE is a page table. The AST is therefore the data block containing all page tables, and is part of the wired segment sst.

APTE stands for Active Process Table Entry. The APTE is forty-eight words long, and contains all the data about a

particular process needed by traffic control. The APT is therefore the data block containing an APTE for each process, and is allocated in the wired segment tc_data.

pxss stands for Process Exchange Switch Stack, combining the names of two older traffic control modules. Currently pxss contains the bulk of traffic control.

Eligible processes are those to which enough core has been committed for them to run. Eligibility can be revoked after one cpu second if the process is running outside ring zero; otherwise it can only be lost by an explicit call to BLOCK. Eligibility entitles the holder to absolute pre-emptive priority over any process which subsequently becomes eligible.

To load a process, the first pages of PDS and DSEG are read into core and temp-wired. A process will be loaded (by pxss) as soon as possible after it is awarded eligibility, and unloaded when it loses eligibility.

KST stands for Known Segment Table. It is primarily used at segment-fault time to find a segment that must be made active. Hardcore segments are not in the KST as they are always active.

PDIR stands for Process Directory. This is a per-process directory in which an M-process may create its temporary segments.

PIT stands for Process Initialization Table. It is not used by hardcore. The outer ring programs of an M-process can find their process-creation parameters in this segment.

DST stands for Device Signal Table. This is a data block used by some I/O interfaces, and is currently allocated in tc_data immediately after the ITT, although it has nothing to do with traffic control.

ITT stands for Interprocess Transmission Table. It is a message queue used to pass information with interprocess wakeups. It is currently allocated in tc_data between the APT and the DST.

M-process is a new term, from MPM process. It designates the type of process Multics has traditionally supplied for each user: cumbersome and expensive, but able to use all of the features of the Multics environment. An M-process always has an unshared address space, implemented with an unshared KST and DSEG. It always has a distinct PDIR in which to store its many per-process segments.

H-process is a new term, from hardcore-only process. It designates a process which can reference only hardcore segments, using a mostly-shared address space. Since even M-processes already have identical address spaces for most ring zero

segments, and the system is coded to take advantage of this, the H-process does not create any unusual programming restrictions.

An idle process is a fiction of traffic control. There is one per CPU, and it is run whenever there is nothing useful for that CPU to do. It is not supposed to take page faults because that might cause it to become unrunnable. An idle process has an unshared PDS and DSEG, and in the current implementation may be considered an H-process.