Laboratory for Computer Science
Computer Systems Research Division

September 8, 1976
Request for Comments No. 123

A Two-level Implementation of Processes for Multics.

September 8, 1976 21:23
R. Frankston

This is a description of an implementation of Multics Processes using multiple levels of abstraction. The implementation is being done in conjunction with David Reed and is based on the model described in his Master's Thesis titled **Processor Multiplexing in a Layered Operating System.**

This draft contains many implementation details, some of which have been modified in actually writing the code and will be described in a later memo. Some sections are only superficial and are meant as a guide for later revisions and extensions. Warning: Since this document is being modified as design changes are being made without a complete rewrite there may be inconsistencies in the descriptions.

If you have comments, suggestions or questions either see me personally or send mail to Frankston.CompSys@MIT-Multics or RMF@MIT-MC.

Table of Contents

Two-level Process Implementation

Introduction

The description of the implementation below is concerned with relatively narrow issues involved in actually coding algorithms which implement the model described in David Reed's thesis. The implementation includes some arbitrary decisions necessary for the embodiment of the algorithms. This description assumes familiarity with the current Multics system. David Reed's thesis should be consulted for a fuller discussion of the issues involved. To make the document at least somewhat readable for a wider audience as well as to reduce the problem of the proliferation of strange abbreviations there is a glossary on page 42.

The key difference between the current Multics implementation and the multilevel one is that a distinction is made between scheduling decisions (i.e. traffic control) that involve policy and those that don't. For the ones that don't involve policy the decision is relatively trivial -- the next processor available to run will be run, a relatively cheap operation. In order to achieve this simplicity the primitive level, level one, consists of a fixed number of virtual processors that are considered at higher levels to be always assigned to a processor. In fact physical processors are a relatively expensive and therefore scarce resource requiring the basement of the implementation to, in fact, multiplex the virtual processors on physical processors on a first-come, first-served basis within a predetermined priority assignment.

The advantages of the two level approach to traffic control include:

i. The system is simplified since one can view a Multics process as being built upon the relatively simple semantics of a virtual processor as opposed to the complex semantics of the current traffic control and interrupt structure.

ii. The implementation of the system primitives for process coordinations can be more efficient than the current ones because of the simplified environment in which they run.

iii. By improving the structuring of the system, the system an become more understandable and thereby more reliable.

iv. Robustness is enhanced by isolating Virtual Processor multiplexing within the PAM. One can assign properties such as encachability to individual processors. Since the PAM does all storing and restoring of physical processor states it can be responsible for all the complexity of maintaining such states.

v. By handling the fault within the PAM outside of the virtual processor, the VP itself need not be capable of handling page faults thereby simplifying the semantics and removing special restrictions which require the wiring of the descriptor segment. Further more faults due to processor failures can be handled by another VP that does not use the particular feature. For example, the can be a process that does not rely on the cache so that it can diagnose cache failures.

vi. By separating processor multiplexing from scheduling the implementation of the policy portions of the scheduler are simplified by separating them out and are infrequent enough to remove the need for the efficiency of assembly language programming.

The current implementation plan consists of three parts:

1. A basic level one system without paging.
2. Level one with paging.
3. A full Multics system with the second level traffic controller.

At present a basic version of level 1 has been debugged and run. It is described on page page 40.

The Processor Assignment Manager and primitives

This basement (level zero) program (corresponding to the GPP algorithm in the thesis) is referred to as the Processor Assignment Manager (PAM)[i]. The PAM is to be considered as part of the physical processor -- there exists one logical instance of the PAM per processor. In addition to the function of multiplexing the physical processors, the PAM also serves to enhance the basic 68/80 processor by rationalizing its operation so as to provide a better basis for the other levels of implementation.

The PAM is entered whenever an interrupt or fault occurs. The currently executing virtual processor is unbound from the physical processor by saving its state in its Virtual Processor Table Entry (VPTE). As part of saving the state of the process the metering information is updated and a check is made to see if the process has exceeded its limit for CPU usage. The next step in processing depends on the reason for entering the PAM.

External interrupts are transformed into events that can be serviced by processes awaiting their occurance. If an internal interrupt (fault) can be handled by the VP itself, the fault information is saved in a communications area in the VPTE, the VP is marked as being unable to process further faults and its state is modified to execute its fault handler. If the fault cannot be handled by the VP, the VP is marked as *unsafe* and the Virtual Processor Coordinator (described below) is expected to do further processing. One fault is handled specially; the mme4 executed in a priviliged segment is treated as a `callp` operation by the PAM and serves to extend the capabilities of the physical processor. `callp` is described in more detail below. When the PAM has finished the interrupt processing, it places the VP into a new state. If nothing that affects the ability to run the VP has occured, it is placed in the *runnable* state.

The states that a VP may be in are:

> *running* indicates that the VP state is currently being interpretted by a physical processor and that the version in the VPTE is therefore invalid.
>
> *runnable* indicates that the VP may be assigned to a physical processor as soon as there are no higher priority runnable VPs. A VP enters the *runnable* state when it is unbound from a physical processor, but may continue to execute.
>
> *unsafe* indicates that the VP cannot be run without further handling by the Virtual Processor Coordinator. A VP enters the unsafe state if it takes a fault it cannot handle or does something the PAM does not expect. *Currently this state is not used, instead the VP is simply placed in the stopped state for examination by the level two traffic controller.*

[i] For historical reasons this module is also referred to as the Processor Binding Manager (PBM).

stopped indicates that a VP is no longer runnable and will not be handled further by the VPC. Once a VP enters this state it is eligible for unbinding by the level two traffic controller. Furthermore that is the only operation that may be performed on it. A VP enters this state when it exceeds its resource limits, or otherwise requires higher level processing to continue. The level two traffic controller explicitly places a VP in this state when it wishes to unbind it so that the L2TC my modify its state. *Stopped* VP's are kept on a queue for action by the L2TC.

awaiting is a state the VP enters when it goes blocked waiting for an eventcount to be advanced.

VPC blocked is a special state indicating the VPC is waiting for something to do. The VPC may only be in this state, *runnable* or *running*.

After placing the VP in its new state the PAM can do some standard processing including processing requests for clearing the cache and possibly deleting the CPU on which it is running. (Some of this standard processing is done earlier in the sequence than indicated in this description in order to minimize the time between entering the PAM and performing the function.)

Once the PAM has finished its processing, it then searches the VPT for next *runnable* VP. It places the VP in the *running* state to indicate that no other processor may examine the VPTE state. After checking to make sure that the VP may indeed run on the available CPU, it then loads the VP's state in effect binding it to the processor and running the VP

The support of the virtual processors is split between the PAM and a dedicated VP; the Virtual Processor Coordinator. This support includes the handling of faults and interrupts and mapping them into the appropriate functions. It also includes the support of the extended operations described in the section on VPI and on the CALLP operator. The VPC runs in a Virtual Processor so that it may take advantage of the process environment to simplify its implementation. The details of the VPC operation are given in a later section of this memo. The VPC is made runnable whenever an event occurs that requires its attention. The VPC is always the highest priority process so that it runs as soon as it is made runnable. Events requiring the VPC include the transition of a process to the *unsafe* or *stopped* states, the occurance of an interrupt or the transmission of a message to the VPC via `callp` as described below

Other dedicated VP's perform functions such as interrupt handling and page fault handling. A key dedicated processor is the policy module for scheduling user processes. This process is referred to as the level two traffic controller. Because of the limited number of virtual processors the level two scheduler must multiplex these processors. The details of this operations are not relevent for this memo. What is important is how a user (or level two) process is bound to a virtual processor and later unbound. This is similar to the function performed by the PAM and is done via the `VP1$bind` and `VP1$unbind` primitives.

PAM Details

There are a number of details associated with actually accomplishing the functions required of the PAM. These are discussed in the relatively unordered sections below. Detailed knowledge of the 68/80 processor is assumed. This information is contained in the GMAP manual, the 6180 processor manual and the Multics debuggers handbook. None of them fully or accurately described the current 68/80 processor.

General flow through the PAM.

i. The PAM is entered via the interrupt or fault vector.

ii. The control unit state and processor registers are saved. The current value of the real time clock is saved.

iii. Any requests to clear the cache of an associative memory are honored. This is described below under heading of connect fault processing.

iv. Virtual CPU time is computed. If there is a process awaiting the realtime event count, it is is notified.

v. Any special processing associated with the particular fault or interrupt is done.

vi. The virtual processor that was executing is placed in a new state. Normally it is placed into the *runnable* state unless the fault handling changes the process' characteristics. If the resource limit for virtual CPU time has been exceeded the process is placed into the *stopped* state.

vii. If the CPU is to be deleted, it notes that it in fact has been deleted and then goes to sleep here. The interrupt indicating that it has been added back continues from this point after intializing the processor state.

viii. The VPT is locked. If there is a pending wakeup for the VPC and the VPC is in the *VPC_blocked* state, it is made *runnable.*

ix. A virtual processor that is *runnable* and does not have any restriction against the current physical processor is placed in the *running* state.

x. The timer register is set as described below.

xi. The state of the virtual processor is loaded into the physical processor and begins execution.

Operating Modes

Since the PAM is meant to act as an extension of the processor and form the basis for other mechanisms it operates in absolute mode so as not to depend on the correct functioning of the memory management software or hardware. This also removes the need to treat the descriptor segment specially (such as wiring the zeroth page) since the PAM is even more primitive than the levels relying on the appending hardware. When the PAM does use the appending hardware in implementing the `callp` operation, it is able to take faults in the same manner that any other hardware instruction might and processes them as if they had occured in an arbitrary hardware instruction. Since PAM processes interrupts by simply noting that the event took place and then restoring the processor state it operates inhibited.

Interrupt and Fault Handling

The 68/80 does not have any physical processor registers that can be used to distinguish between physical processors when addressing memory to store the machine state when an interrupt is taken. Furthermore there is only one address associated with each interrupt handler, without regard to the processor on which the interrupt is taken. Because interrupts are handled by processes, the processor need not be masked for interrupts at any time it is assigned a virtual processor. Therefore there is no need for complex masking strategies -- the processor can run with all interrupts unmasked at all times with the PAM using the inhibit bit to prevent interrupts.

Since any interrupt can be taken on any processor it is necessary to be able to save the machine state without regard to the processor it is taken on until sufficiently far into the PAM to enable the program to determine which processor it is on and where the associated VPTE is for deassigning the virtual processor. The algorithm used was inspired by Andre Bensoussan's work and worked out in conjunction with Bob Mabee (of course Dave Reed contributed, but then his contributions are assumed throughout). There exist two tables with enough capacity to store SCU data for each processor that may be configured. There is a pointer with a delta modifier equal to the length of an SCU entry. The interrupt vector is initialized to store the SCU data using an AD modifier. Thus when the interrupt occurs an address is obtained to store the current data and the pointer is updated in storage in an atomic operation so that if any other processor takes a fault it will not interfere. Control is then transferred to a common disambiguating routine that operates under a lock. The lock itself is grabbed using the sznc instruction which does not require the use of registers. The rest of the registers are then stored, the processor id is determined and thus the per processor storage address to which the registers are transfered. The pointer to the SCU table is then reset to point to the beginning of the other table and the first table is scanned from its beginning using the AD modify. Each entry is checked to see if it belongs to the currently running physical processor. If it does, then the data is simply copied out into per processor storage. If it does not, the data is then copied into the new table, again using an AD modifier to grab and reserve a slot. When this processing is done, the lock is released (via an `stcl`) and the next processor looping on the lock can repeat the operation with SCU tables switched.

Fault processing is similar to interrupt processing except that we can have a separate fault vector for each processor to save the need for having to determine dynamically the identity of the processor on which the handler is running. The processing for both faults and interrupts is the same once we have copied the machine conditions into per processor storage.

Faults while in the PAM.

When the PAM is processing a `callp` request or a page fault, a further fault may be taken. In order to handle these a separate fault handler is used that assumes the fault is expected and that the PAM is in a "good" state. The handler does not save any registers and assumes that control registers (pointers to the VPT entry and the perprocessor information) are intact. The detailed handling depends on the PAM state. If a `callp` operation is being performed then the machine conditions are set to indicate that the fault occured while processing the `callp` operation itself and the fault is processed as if it had occured at the beginning of the operation. For page faults a message is sent to the page fault process for the fault (which must be on the descriptor segment) and the machine conditions are set to continue with the appending cycle when the descriptor segment becomes available.

The descriptor segment.

It should be noted that by operating in absolute mode, the PAM avoids dependence upon the descriptor segment. Current Multics takes advantage of appending mode by using the fact that the descriptor segment can be used to address different memory in the PRDS for each processor. The elaborate scheme described above is complicated by not having this mechanism available but as a consequence removes the requirement that descriptor segments be different on each processor and allows processes to share descriptor segments. This can be of great importance in permitting many small process with a single descriptor segment. The idle process is a simple example of a process sharing a single descriptor segment.

Details of callp implementation.

The `callp` is supposed to look like a normal machine instruction that may take faults. It is first validated to make sure that the instruction was executed in a priviliged segment (maybe just the VP program's segment?). If not, it is treated as a standard (mme4) fault and reflected back to the virtual processor. If the instruction is acceptable, the pam state is set to indicate that the `callp` is being processed and a copy of the machine conditions is saved. The operation number in the `A-register` is then examined. If it is invalid the virtual processor is made *unsafe* and the VPC is notified (this should never occur).

The specific processing is done according to the request. Typically it would involve copying the data pointed to by `pointer register 0` into VPTE or copying the data from the VPTE. The detailed operation of each `callp` is described in the section on the `callp` operator.

When the processing is done, the PAM continues by placing the virtual processor into the *runnable* state and resetting the callp-in-progress flag. The PAM then continues as for any other fault.

If a fault occurs while the `callp` is being processed, the fault conditions are reset to those at the beginning of the `callp` instruction with the exception of the data address being referenced which is taken from the new SCU data associated with the fault. When (and if) the `callp` is restarted after the fault, it will begin from the beginning of the instruction. This allows the fault handling program to use the `callp` operation itself and not have restrictions on using the communications area in the VPTE.

Page fault processing.

The SCU data is examined to determine the type of fault. A message is sent to the page fault process consisting of the ASTE entry pointer, a unique segment id (in case the AST entry is deactivated), the descriptor segment AST entry pointer, the page number and a eventcounter associated with the fault. The process is then left awaiting this event, ready to continue address evaluation.

Processing the connect fault

The processing of the connect fault is very simple -- it is ignored. Its purpose is to force a processor to enter the PAM. It achieves its effect since whenever the PAM is entered it performs standard housekeeping functions. In particular a connect fault is issued after a message is left when clearing the cache or when adding/deleting a processor.

Clearing the Cache

The table of pending clears has one entry per processor. When the PAM wants to clear the cache in other processors, it places in each table entry the appropriate instruction. It does this via a stacq instruction to make sure that it is replacing a nop. If it does find an instruction other than a nop, it assumes that another processor has left a instruction and loops attempting to execute the instruction in its entry and leaving an instruction for the other processor. It makes sure the other processor enters the PAM by issuing a connect to the other processor.

Process addition and deletion.

When a processor is added, after some initialization, it enters the code to scan the VPT and find work to do. When a processor is being deleted, it checks for he request immediately priori to scanning the VPT for more work to do and disables itself. In either case an eventcount is incremented and the VPC is notified of the change.

Making the VPC runnable and processing the VPT

Whenever there is an event that requires the VPC's attention, a wakeup-waiting flag associated with the VPC is set using the stcl instruction. The last part of the PAM locks the VPT. The wakeup-waiting switch is cleared with an sznc instruction. If it was set, then the VPC is placed in the *runnable* state from the *VPC_blocked* state, using the sznc instruction.

The VPT is then scanned for the first (and therefore highest priority) process that is in the *runnable* state. One will always be found since there is always a lowest priority idle process available. When the entry is found, it is placed in the *running* state. A check is made to see if the process has a restriction against the current processor and if so, makes it again *runnable* and continues the scan. Otherwise the VPT is unlocked and the virtual processor is run.

Running the VP

This is the final part of processing that is done after a VP has been found in the VPTE and has been placed into the *running* state. The appropriate pointers are set in the per processor tables for storing fault data and referencing the VPTE, the clock time is saved for computing virtual CPU time and the registers are loaded. If the VP is being run on a different processor than it had last time, the cache for the current processor is cleared. Final processing is done with separate code per processor so that the appropriate SCU data may be restored. The VP is then off and running.

Process Signals (IPS)

The process signalling mechanism corresponds to the current IPS mechanism. It is implemented by setting a flag in the VPTE to indicate that an interrupt is pending. When the virtual processor is to be run a check is made to see if the flag is set and faults are permitted. If so a fault is simulated. If faults are not permitted, the action is deferred until the flag is reset to indicate that it is safe for the virtual processor to take faults again. The details of using this signal are discussed in the section on notification.

The interrupt pending flag is set by the L2TC. If a running process is to be interrupted, it is first stopped, the flag is set and then it is rebound to a VP. The choice of this method is motivated by a desire to minimize primitives available for accessing the VPTE. A tradeoff can be made between number of such primitives and the frequency with which the L2TC must unbind a VP in order to access parts of its description.

Special machine state information

This section explains how history registers, fault registers, alarm register are managed. In addition there is software state information such as the VP state which is discussed elsewhere. This will not be addressed at the moment since it is more a matter of retaining current Multics details without requiring a major changes for the PAM. Note, however, that since the PAM is aware of the VPs, it is feasible, possibly, to control history register handling on a per-VP basis (and therefore on a per process basis.

Virtual CPU time measurement and limits

Associated with each processor running a VP is the clock time at which the currently running virtual processor started running (the PAM was last exited). When the PAM is entered the starting time is subtracted from the clock time at which the virtual processor stopped (the PAM was entered) to determine how long the VP has been executing. This value is added to the value accumulating the in the VPTE. A check is then made against the VCPU limit for the VP. If the limit has been exceeded, the process is stopped for deassignment by the level two traffic controller.

As a refinement to this scheme is an estimate of the overhead involved in invoking the PAM before the clock is read on entry and after the is read on exit. This can be subtracted from the VCPU in an attempt to isolate the charge for a processor from that of running the PAM.

Timer register setting and usage by PAM

The timer register is used to make sure that the PAM gets invoked periodically so as to enforce quantum length restrictions (i.e. virtual time quota) and to make sure the VPC gets invoked so that it can advance the real time eventcount. For simplicity the PAM is run at least every 50(?) milliseconds. The alternative would be to calculate the minimum of the virtual time limit for the process being bound and the time the VPC is to be run. This would be more complicated and the additional resolution is not necessary.

Other processes

Proper operation of the PAM depends on two kinds of VP's. The first is the Virtual Processor Coordinator that is described in great detail below. It is always the highest priority virtual processor and is made runnable whenever there is something requiring its attention and therefore run immediately. Second are the lowest priority processors -- the idle processors. There is one idle processor for each physical processor. Since the idle VP is lowest priority it is run only if there is nothing else for the physical processor to do. The idle processors are quite cheap since they can share a descriptor segment or run in absolute mode without a descriptor segment. Other than that no special consideration need be given to the idle process.

The callp operator

As noted above the callp instruction is used to extend the operation of the virtual processor. It is implemented within the PAM. It takes an operation number in the A-register and a data pointer, if any, in pointer register zero. Like any other normal instruction, it may take faults. When the fault occurs the machine conditions are set to restart the execution of the instruction from the beginning so that there is no need to save partial state information associated with copying information into the VPTE buffers.

The operations are:

1: AWAIT takes a list of eventcount names and values (as described below under VP1$await and places the process in the *awaiting* state until one of the named events is notified. It is possible for one of the awaited events to be advanced while the process is being placed in the *awaiting* state. It is therefore necessary to make sure that the none of the eventcounts has passed the awaited value after the process is in the *awaiting* state. Since the process is no longer considered *running* it is necessary that no faults occur. In order to prevent faults the *absa* is used to get the address in primary memory of the counter value for each eventcount. A fault can occur during this operation in which case the normal page fault processing is done and the await is restarted from the beginning. This pointer can then be used to reference the value while the process is *awaiting*. We are assured that no fault will occur since primary memory addresses are being used for the reference and the virtual memory support is not invoked. We are assured that the address is valid since any other processor that is updating the page tables cannot assume all references to the page frame are completed until it receives an acknowledgement form the other processors. The processor performing the await will not give this acknowledgement until it finishes processing the await request.

The real time clock is a special eventcount in that the minimum value of all such events must be stored so that the timer can be set to notify the event at the specified real time.

2: WAKE VPC is used when a change is made to a VPT entry that requires VPC attention. For example, when a message is queued for the VPC.

3: STOP is used to forcefully stop a specified process. If a process is in an atomic operation, but is to be stopped, a flag is set to indicate that it is to be stopped when the atomic operation count reaches zero.

4: BEGIN ATOMIC OPERATION is used when a process is executing a critical section of code. It increments an atomic operation counter in the VPTE.

5: END ATOMIC OPERATION decrements the atomic operation counter. If the count reaches zero and a stop is pending, the process is placed in the *stopped* state.

6: GET FAULT DATA copies fault data out of the process' state into pageable storage. Note that page faults are permitted during this operation since they are handled by another process. Segment faults are not permitted because they are handled by the faulting process and will require the use of the fault data area. Note that the atomic operation counter was incremented at the time of the fault and the process was marked as not being safe to take faults. The safe_to_take_fault_flag is reset by this operation. The atomic operation count must be decremented by restoring the processor state or explicitly ending the atomic operation.

7: RESTORE PROCESSOR STATE restores the machine conditions as specified and decrements the atomic operation counter. If this interface is not used the end atomic operation interface must be used to decrement the counter.

8: ADD CPU sends an ADD CPU message to the VPC.

9: DELETE CPU sends a DELETE CPU message to the VPC.

10: CLEAR CACHE used when an object loses encachability. Its parameters consist of a suboperation number and the page id for suboperation cache clearing by page. The suboperations are:

1. Clear PTW cache via a camp.
2. Clear SDW cache and PTW cache via cams and camp.
3. Clear PTW cache and memory cache by page - camp 4 + page id.
4. Clear memory cache, SDW cache and PTW cache with cams 4 and camp.

These are used by (1,3) page control, (2) segment control and (4) access control. They apply to all processors. The actual method by which the processors execute the instructions is explained in the section on PAM details.

11: VPC BLOCK is used by the VPC so as to cause checking of the VPC's wakeup waiting switch. It takes as a parameter the next real time before which the VPC is to be run.

The VP1 interface

The VP1 program provides a PL/I compatable interface to the callp instruction, the VPC and the VPT. It limits the operations the can be performed; no other interface exists. The use of the common segment name of VP1 is primarily for convenience; the entries are essentially independent.

A basic service provided by the VP1 routine is the management of assignment of level two processes (those managed by the level two traffic controller) to virtual processors. There are a number of semantic models that can be associated with this operation. The primary one is that of binding and unbinding. An alternative view is that one loads and unloads a processor state to and from a virtual processor much as one loads and unloads a process the current Multics implementation. A better understand of what is actually happening can be achieved by realizing that the bind operation is really taking a processor state description maintained by the level two TC which has no existence other than as an entry in a database and is creating a level one processor with an initial state for execution. The unload operation destroys this processor and returns a description of its final state. Key to the understanding is that the PAM does not enforce any continuity between the process description returned by an unbind operation and that provided to a bind operation. While the description is being maintained by the level two traffic controller, the L2TC is permitted to perform arbitrary operations on its description including fabricating new descriptions and discarding old ones.

VP1 communicates with the VPC via a communications queue. The queue is managed without the use of explicit locks. The stacq instruction is used to perform interlocking.

The information maintained in the VPTE consists of two parts -- that which is communicated via the VP1 interface and that which is internal to VP support. For convenience the portion that is passed through the interface is kept in the same format by the level two traffic controller as in the VPTE, but this is not necessary.

The Process_Description portion of the description is used to store information that maintains the identity of a Multics process as seen by the user.

```
declare 1 Process_Description based aligned, /* 16 words aligned!      */
          2 process_id bit(36),
          2 lock_id bit(36),
          2 excluded_processors aligned,
            3 excluded_processor(0:3) bit(1) unaligned,
            3 padding bit(32) unaligned,
          2 BAR bit(36),               /* For 6080 emulation          */
          2 DSBR bit(72),              /* Descriptor Segment Base Reg*/
          2 ring_alarm_word bit(36),
          2 PD_flags aligned,
            3 safe_to_take_faults bit(1) unaligned,
                                       /* Fault data can be copied?  */
            3 pending_process_interrupt bit(1) unaligned,
          2 resource_metering,         /* Metering and limits         */
            3 virtual_time_used fixed binary(71),
            3 virtual_time_limit fixed binary(71),
            3 memory_usage_meter_reference like meter_reference,
          2 processor_state,
            3 machine_conditions like mc;
```

The VP_Description contains information that is only available to the VP support and is not passed through the VP interface.

```
declare 1 VP_Description based aligned,
          2 next_VPTE like VPT_ptr aligned,
          2 VP_id bit(36),              /* Identification of this VP  */
          2 VP_state fixed bin,         /* runnable when bound        */
          2 VP_priority fixed binary,
          2 last_processor fixed bin(2), /* For cache maintainance"   */
          2 atomic_operation_count fixed bin(35), /* Initially zero   */
          2 pad16(10) bit(36) aligned,
          2 fault_conditions like processor_state,
                                        /* Communication with handler */
                    /* For simplicity I am putting the awaited events
                       in the VPTE.  Eventually they will be managed
                       separately by the VPC. */
          2 eventcounts,
            3 number_events fixed binary,
            3 event_names(4) like global_eventname aligned,
                     /* 4 = max_number_of_11_events                   */
          2 VPD_flags aligned,
            3 pending_stop bit(1) unaligned,
            3 padding bit(35) unaligned,
          2 pad8b(6) bit(36) aligned;

declare 1 VPT_ptr based aligned,        /* Pointer entry for VPT      */
          2 abs_ptr bit(18) unaligned,  /* For use in absolute mode   */
          2 rel_ptr bit(18) unaligned;  /* For use in appending mode  */
```

The VPTE itself contains both parts:

```
declare 1 VPTE based,
          2 VP_info like VP_Description,
          2 Process_info like Process_Description;
```

The awaiting_events_table is used in the interface between V1$await and callp/await.

```
declare 1 awaiting_events_table based,
          2 number_events fixed binary,
          2 events(max_number_of_11_events),
            3 local_name pointer,    /* Only valid in owner's address
                                        space */
            3 global_name like global_eventname,
            3 value fixed binary(35); /* Value process is awaiting */

declare 1 global_eventname based aligned,
          2 segment_unique_id bit(36) unaligned,
          2 word_offset bit(18) unaligned,
          2 pad bit(18) unaligned;
```

VP1$bind

```
     declare VP1$bind entry (bit(36),1 like Process_Description, fixed
                            binary(35));

     call VP1$bind (VP_id, process_description, code);
```

The semantics of the bind operation has been discussed above. The caller of VP1$bind should set the appropriate flag in the ASTE to keep the descriptor segment of the specified process active. It initializes the values in VP_info as part of the transformation from the representation maintained by the L2TC and that in the VPTE. The process_state is *stopped*, the last processor is "-1" (i.e. none), and the atomic operation count is zeroed. It then uses the callp/load operation to load it into a free VPTE. The operation will fail if there are no VPTE slots available. It would be expected, however, that the second level TC will not call the primitive unless it knows that there is one available.

VP1$unbind

```
     declare VP1$unbind entry (bit(36), 1 like Process_Description, fixed
                              binary(35));

     call VP1$unbind (VP_id, process_description, code);
```

The semantics of unbinding has been discussed above. It issues a callp/unload operation to request the contents of an stopped VPTE be returned. When this operation has been done the VPTE is available for a subsequent bind operation. It is expected that VP1$unbind would be used repeatedly to unbind all stopped virtual processors so that the associated process descriptions would be available to the level two traffic controller. Note that an eventcount is incremented any time a process is stopped so that by awaiting that event count the L2TC can immediately perform the unbind operation.

VP1$stop

```
declare VP1$stop entry (bit(36), fixed binary(35));

call VP1$stop (VP_id, code);
```

The stop entry is used to force a process associated with a VP to stop executing. The details a discussed in the description of the `callp/stop` operation. The VP1$stop operation is used whenever the level two traffic controller needs to manipulate the process' description. For example, to destroy a process, the L2TC would note that it wants a particular process destroyed. If it already has full control over the description, i.e. the process is not bound to a VP, it can perform the operation immediately. Otherwise it would issue a `VP1$stop` for the process. As soon as the process is stopped, the "stop process" eventcount would be incremented, `VP1$next_stopped` would locate the VP, and `VP1$unbind` would copy out the process description. For each process description returned by the `VP1$unbind` operation the L2TC would check the notes associated with the it and perform any necessary operations; in this case the process would get destroyed.

VP1$next_stopped

```
declare VP1$next_stopped entry (bit(36), fixed binary(35));

call VP1$next_stopped (VP_id,code);
```

This entry is used by the L2TC to get the id of the next available stopped VP. It is invoked in response to an advance on the *stopped* eventcount.

VP1$run

```
declare VP1$run entry (bit(36), fixed binary(35));

call VP1$run (VP_id, code);
```

This places a makes a stopped VP runnable. It is normally used after the VP1$bind operation.

VP1$await

```
declare VP1$await entry (1 (*), 2 pointer, 2 fixed binary(35), fixed
                            binary, fixed binary);

call VP1$await (events, number_events, advanced);
```

The parameters consists of a table of event names (pointers) and values to be awaited. The number parameter specifies the number (up until the maximum value) of events that are to be awaited. The index of the event which caused the return from awaiting is given as "advanced".

The table of event_counts is completed by filling the event name as derived by the VP interface from the segment id and the word address and passed to the callp/await operation. Note that there is a maximum for the number of entries in this table. The user level interface to VP1$await must permit an arbitrary number of event names to be specified while only passing a limited number of event names to VP1$await. The details of this are described in the section on notification.

VP1$advance

```
declare VP1$advance entry (1 like awaiting_events);

call VP1$advance (event_table);
```

As with VP1$await, the event_name is filled in. The await_value is, in this case also filled in after incrementing the associated counter with the new value. The table is then passed to callp/notify

VP1$add_cpu

```
declare VP1$add_cpu entry (fixed binary, fixed binary(35));

call VP1$add_cpu (cpu_number, code);
```

This entry interfaces to callp/add_cpu.

<u>VP1$delete_cpu</u>

```
declare VP1$delete_cpu entry (fixed binary, fixed binary(35));

call VP1$delete_cpu (cpu_number, code);
```

This entry interfaces to callp/delete_cpu.

<u>VP1$crash_system</u>

```
declare VP1$crash_system entry ();

call VP1$crash_system ();
```

Deletes all physical processors from the system, and forces one of the processors to execute a special debugging program.

<u>VP1$clear</u>

```
declare VP1$clear entry (fixed binary, bit(18), fixed binary(35));

call VP1$clear (suboperation, page_id, code);
```

Interfaces to callp/clear_cache to clear cache the specified associative memory.

<u>VP1$begin_atomic_operation</u>

```
declare VP1$begin_atomic_operation entry ();

call VP1$begin_atomic_operation ;
```

Interface to callp/begin_atomic_operation.

<u>VP1$end_atomic_operation</u>

```
declare VP1$end_atomic_operation entry ();

call VP1$end_atomic_operation ;
```

Interface to callp/end_atomic_operation.

<u>VP1$get_fault_data</u>

```
declare VP1$get_fault_data entry (1 like fault_conditions);

call VP1$get_fault_data (fault_conditions);
```

Interface to callp/get_fault_data.

<u>VP1$restore_processor_state</u>

```
declare VP1$restore_processor_state entry (1 like processor_state);

call VP1$restore_processor_state (processor_state);
```

Interface to callp/restore_processor_state.

VPC Operation

As noted above, the VPC is run whenever an event occurs that needs it attention. For example, a process leaving the runnable (or running) state, an interrupt event occuring or a message being sent from a process. In later implementation some of these occurances might bypass the coordinator, but for now it is assumed that all complicated low level operations involve the coordinator.

The basic operation of the VPC consists of three loops:

1. Scanning for processes by state, i.e. unsafe and exceeded limits.
2. Scanning for advanced interrupt cells. This means that there is an implicit, rather than an explicit advance done on the cells by the PAM.
3. Processing of explicit messages to the coordinator.

Note that each loop is entered only if an associated flag has been set to indicate that there may be work of the specified type to be performed. When the processing is done the VPC unbinds a set of physical processors so that they may adjust to the new state of the world. It is only necessary to unbind those processors that are running the "n" lowest priority processes where "n" is the number of processes that have been made runnable by the VPC.

In more detail, the processing consists of:

1. This loop scans the Virtual Processor Table (VPT) examining the state of each process that is found. Each *stopped* VP is removed from the chain of runnable processors and an eventcount is advanced to notify the level two traffic controller. Note that kernel processes should never be stopped. If an *unsafe* process is found, a debugging process should be notified or the system crashed. ????

2. Next the interrupt and fault counters are scanned for any that have been incremented by comparing against an earlier set stored in the VPC and the appropriate waiting processes are notified. (For the interim implementation with a single "interrupt side" processor there is an additional event counter to indicate that any interrupt has occured). As a special case of interrupt handling, the system clock can be interogated and compared with the value for the next timer event of interest.

3. Scan for messages from other processes.

    i. RUN. Places the specified VP into the runnable state and chains it into the queue of runnable VP's.

    ii. NOTIFY notifies processors that are AWAITing that counter.

iii. DELETE CPU. Leave a note for the specified processor to deconfigure itself and then unbind from any virtual processor it may be running it via a connect.

iv. ADD CPU. Leave a message telling a CPU to come to life and send a connect to it, forcing it to initialize itself.

A final note on locking. Normally the VPC looks at the VPT without setting a lock because it is the only process that may change the VPT. When it does change the VPC it loop locks to prevent conflicts with the PAM that may be searching the chain. The VPC itself is run whenever its wakeup-waiting switch is set by the PAM indicating that there may be work for it to do. This flag is reset whenever the VPC is placed in the runnable. Any events of interest that occur after this time will set the VPC wakeup-waiting switch in case it hasn't done all of its processing in its previous incarnation. Thus for example, if no paging communication buffer is available when the VPC looks and one becomes available while the VPC is running, no race condition arises because the VPC_run flag will be set anyway so that the VPC will be run again to make use of the buffer immediately after it unbinds to wait.

Also some efficiency considerations. As pointed out above it is possible to bypass some of the mechanism described above should the running of the VPC be considered too expensive. The VPC need not be expensive. Its operations are simple and it avoids the major expensive operation in PL/I, the full subroutine call. The only call it needs make is to an ALM procedure that is used for basic utility operations. This call only involves minimal housekeeping making it more efficient than a full PL/I call.

Modifications to page control

Unlike the current Multics, a page fault is not handled by the process taking the fault. This approach greatly simplifies the construction of a process because it removes the need to handle "awkward" situation such as a page fault occuring when the fault handler is copying fault data out of the VPTE. It also makes it possible to take a page fault on any page of the user's descriptor segment removing the necessity for wiring any pages of a process since the other requirement for wired pages -- external interrupt handling, is also removed by having interrupts handled by dedicated processes.

The page fault processing itself is simplified since the use of a process dedicated to this functions greatly reduces the locking problems associated with page fault handling. The modifications to page fault handling are minimal since page fault already runs in an environment that has little to do with its host process and is thus easily decoupled. Some consideration has been given to using the modified version of page control designed by Andy Huber and refined by Bob Mabee.

The PAM generates a message to the page fault process by extracting the relevent data from the SCU data. Faults on page zero of the descriptor segment are permitted. The messages is placed in a ring buffer. The format on an entry is:

```
declare 1 page_request based,
          2 pointer fixed binary,        /* In AMT or WMT */
          2 segment,
            3 astep pointer,             /* ASTE Entry */
            3 uid bit(36) aligned,       /* To make sure still same. */
          2 eventcount_index fixed binary;      /* To notify process */
```

The meter pointer is discussed in more detail below in the discussion of the Active Metering Table. When the request is queue the AMTE wire count is incremented. After the meter is incremented to charge for the processing, the wire count is decremented to release the meter. The event count is derived from the segment unique-id and the page number within the segment. This value is hashed into a wired table of page events. It is the index of this entry that is placed in the page request. The use of a preallocated table removes the problem of allocating wired storage. We can use a small table without limiting the number of outstanding page faults by not requiring that the assignments of eventcounts to paging operations be unique. There is no requirement that the event be unique, it is only a matter of efficiency. At worst, a processor may get a spurious notify, attempt to execute, and fault again.

The modifications to page control consist of:

Removal of the code that handles the fault directly as this is now done by the PAM.

Removal of the explicit interactions with pxss.

Removal of the code involved in locking the page table since this process has exclusive access to its databases.

Changing the references to metering data in the APT entries to use the AMT.

The Active Metering Table

Note: The discussion of the active metering table is included for completeness. The actual details of the mechanism are not yet fully worked out and the implementation of a layered system need not be dependent upon the current AMT design.

In a "real" system it is necessary to account for resource usage and to limit such usage against predetermined limits. In the current Multics system, many of the resource measurements are associated with processes. Since the processes are known to the lowest levels of the system, not even deactivated, the Active Process Table (APT) has become a repository for such information, or at least the resource measurement information.

In the multilevel system, only virtual processors exist at the lower levels. Since the processes assigned to this virtual processors do not exhibit the continuity of the present Multics processes it is necessary to develop a separate mechanism for measuring resource usage. Furthermore, if we look beyond just supporting the current measurements, a restructuring of the metering would permit the offering of improved mechanisms such as resource limits and shared meters at the base level; mechanisms which have been proposed in the past but which have not been implemented.

There are two primary components to resource measurement -- the long term and the short term. The long term measurements in current Multics are stored in the PDT (Project Definition Table) and consist of dollar usage and more detailed resource usage measurements. Short term measurements are maintained in the APT. Periodically the Answering Service copies measurements from short term to long term storage.

In the proposed Multics a similar mechanism is used except that the choice of short term meters is more explicit and not directly related to processes. At present we are mainly concerned with meters that must be available to ring zero[ii] -- those that correspond to the APT information. In addition, to simplify the design of page control, the meter (and limit) for storage system usage is also of interest. For the duration of its existence, each such meter resides in the Active Meter Table. It is only necessary for a meter to exist as such while the resource it is measuring may incur charges. For example, the meter of a process' processor usage can only be incremented while the processor is bound to a VP. Thus the level two traffic controller can create the meter at the time that it the process gets assigned to a VP and destroy it (after reading out the value) when the process is deassigned[iii]. In contrast a process can incur memory usage charges after the process has been

[ii] Need better term

[iii] In fact, the VCPU meter is a special case and is kept in the VPTE in the current PAM design; but could reasonably be incorporated into the AMT mechanism as soon as the operation of the AMT is better described, i.e. when I finish writing this section

deassigned from its VP. A third example of a meter is the storage quota meter. Since this meter must be accessible from page control when assigning additional pages to a segment, it seems logical to associate the information with the wired AST entry. Because the meter is actually shared by Multiple segments, it is actually kept separately in the AMT. Note that as a benefit of this aproach the quota limit is independent of the directory hierarchy and that storage system usage can be associated directly with accounts instead of just to superior quota.

Note that the meters described thus far share a special property - they must be accessible without taking a page fault; i.e. they must be wired. This is accomplished by maintaining a Wired Meter Table (WMT).

An entry in the Active Metering Table takes the form:

```
declare 1 AMTE based,
          2 id bit(72),
          2 value fixed binary(71),
          2 limit
            3 limit_set bit(1),
            3 value fixed binary(71),
          2 eventcount fixed binary(71),
          2 wire_count fixed binary;
```

When a meter is to be incremented (via amtm$add), the meter id is used to hash into the WMT and then the AMT to find the entry. If none is found, one is created in the AMT. To make the search more efficient, a meter_reference is used which contains a meter_index in addition to address the table entry. When the entry is found via the index, it is checked to make sure the meter_id in the entry matches that in the reference, if it does not, the hash search must be used and the index is updated to make the next reference more efficient.

```
declare 1 meter_reference based,
          2 index fixed binary,       /* Index in AMT or WMT */
          2 home fixed binary(1),     /* AMT or WMT          */
          2 id bit(72);
```

A meter may reside in either the AMT or the WMT, but not both in order to make limit checking work. When the wire count changes to or from zero the entry is moved. This move is not necessary if the meter is being created in one or the other, or is being read and cleared.

The AMT is managered by the active_meter_table_manager (amtm). The following entries are available.

```
        declare amtm$set_limit entry (1 like amte, 1 like meter_reference,
                                      fixed binary(35));

        call amtm$set_limit (amte, meter_reference, code);
```

As noted above, meter entries are created when an attempt is made to use them. For entries such as page quotas, it is necessary to initialize the entries with a limit value. It is necessary for programs setting and using limits to cooperate such that programs do not check limits unless the limits have been set. For example, as part of activating a segment, a quota limit is set in the AMT. This entry is cleared when all segments sharing that limit are deactivated.

```
        declare amtm$read entry (1 like amte, 1 like meter_reference, fixed
                                 binary(35));

        call amtm$read (amte, meter_reference, code);
```

Returns values for the specified meter entry. If the entry does not exist, zeros are returned for the values.

```
        declare amtm$read_clear entry (1 like amte, 1 like meter_reference,
                                       fixed binary(35));

        call amtm$read_clear (amte, meter_reference, code);
```

Same as the read entry, except clears the value. This is the entry used to read a meter out so it can be updated in a higher level table. The AMT entry may be deleted if it is not wired and does not have a limit set.

```
        declare amtm$read_clear_limit entry (1 like amte, 1 like
                                             meter_reference, fixed
                                             binary(35));

        call amtm$read_clear_limit (amte, meter_reference, code);
```

This entry is similar to the previous but also clears the limit setting so that the entry may be deleted from the AMT if not wired.

```
        declare amtm$add entry (fixed binary(71), 1 like meter_reference, fixed
                                binary(35));

        call amtm$add (value, meter_reference, code);
```

Adds the specified value to the given meter. A code is returned if the value exceeds the meters limit. If the meter does not exist, it is created.

```
declare amtm$add_conditionally entry (fixed binary(71), 1 like
                                      meter_reference, fixed
                                      binary(35));

call amtm$add_conditionally (value, meter_reference, code);
```

This is like the add entry, except the meter value is left unchanged if the limit is exceeded.

```
declare amtm$wire entry (1 like meter_reference, fixed binary(35));

call amtm$wire (meter_reference, code);
```

The wire count for the specified meter is incremented. If the meter is already in the AMT, it is moved to the WMT.If it sdoes not exist at all, it is created in the WMT.

```
declare amtm$unwire entry (1 like meter_reference, fixed binary(35));

call amtm$unwire (meter_reference, code);
```

The wire count for the specified meter is decremented. If the count reaches zero, it is moved from the WMT to the AMT.

```
declare amte$unwire_read_clear entry (1 like amte, 1 like
                                      meter_reference, fixed
                                      binary(35));

call amte$unwire_read_clear (value, meter_reference, code);
```

Combines unwire and read_clear.

Notification and Events

The basic mechanism for coordinating processes in the proposed system is the event. More precisely, event counts are used to store state information about events.The eventcounts are discussed in detail in a CSR/RFC by Dave Reed and Raj Kanodia. When an event occurs the value of the eventcount associated with the event is *advanced*. A process interested in the occurance of the event can *await* this advance.

Eventcounts are identified by eventcount names. To the user an eventcount is simply a word in memory and thus its name is its address. To convert this into a system-wide address the segment number is replaced by the segment-unique id. The eventcount can then be referenced by the system-wide name in order to do a notification. The actual reference to the value of the eventcount within the process awaiting or advancing the primitive is done using the pointer for efficiency.

Eventcounts form a robust mechanism because, though a process may await a transition, the eventcounter itself always maintains its state for later examination. Since the counter is monotonically increasing the *await* operation can be implemented by simply comparing the current value of the counter with a previous value. If the previous value has not been surpassed the process can loop waiting for the change, or can go blocked. This block is actually implemented via the `callp/await` primitive described above. Complementary to going blocked is the mechanism for getting awakened. This is the notification mechanism.

The notification is performed by the VPC as a result of a `callp/notify` operation. This primitive is invoked by the VP1$advance interface. Note that only the advance interface is available outside the PAM. While this is not strictly necessary it does preserve the semantics of eventcounts. When the VPC gets a message to perform a notification, it scans the VPTEs which are in the *awaiting* state and places them in the runnable state. For efficiency, the VPC can actually check to make sure the value awaited has been reached since the value is copied into the VPTE, but this is not strictly necessary since the VPC can simply compare eventcount names.

Spurious notifies are not harmful since the `callp/await` primitive checks the values anyway before returning. `callp/await` also checks the eventcount values after putting the process into the *awaiting* state to prevent any loss of notifies sent just before the process entered the awaiting state.

Eventcounts associated with interrupts and page fault processing completion must be wired and preallocated. To simplify this a Wired Event Table is maintained. We can go further and require that all events originating at level one be in this table. Note that, unlike current IPC, the use of a wired table does not have the danger of overflowing since no messages are placed in the table, eventcounts are simply incremented.

We can take advantage of the restriction on level one originated requests when implementing the level two primitive for event counts. Observe that there is a fixed maximum for the number of events upon which a process may wait. The user interface need not, and should not, have such a restriction. The level two traffic controller can implement its own await/notify mechanism similar to the lower level mechanism except using virtual memory to get around the restriction on the number of events.

A level one process (i.e. a kernel process) can simply use the VP1 event count interface (advance and notify) directly. For level 2 processes, there is a VP2 interface for these primitives. Since a level two process may have an arbitrary large number of events and may be unbind from a VP while awaiting, it is necessary for the level two interface to provide much of the functionality of the interface. To aid level two a special event count is provided that is advanced whenever a level one event count is advanced, the *outward_signal* counter. This is discussed in more detail in the description of the implementation of the level two traffic controller. Other event countes used for communicating with the level two traffic controller include the *stopped* event advanced whenever a VP is stopped and the *clock* event that is advanced at fixed intervals.

As described above eventcounts are passive in that they don't affect a process unless the process examines its value or *awaits* an *advance*. This is not sufficient to implement the current IPS mechanism. What is needed is a means of faulting a process so that it can examine eventcounts which it thinks are important. This consists of setting a process' pending interrupt flag while unbound at level two. When the process is to be run, the flag is examined by the PAM which will cause a fault to be simulated. Note that the fault itself doesn't tell the process what has happened; the process is simply told that something of immediate interest has occured. To give the effect of current IPS, there would be an eventcounter associated with the terminal I/O channel for quits, the real time clock and the virtual clock.

The Level Two Traffic Controller

The lowest levels of Multics described above do not provide all of the functionality of the current system. The implementation requires a second level of control that multiplexes the virtual processors among user processes. This second level is conceptually much like the lower level in that it multiplexes a limited number of processors to give the effect of a larger number. While the first level emphasises simplicity, the second level emphasises function. The second level removes restrictions on the number of processors provided and the number of events that can be observed. It is able to do so because it can make use of the virtual memory mechanisms for managing its databases. Note that the term process is used in the conventional Multics sense, of a user's address space and control point. The level two process is representation of the logical processor that executes a user's instructions.

The Implementation of old IPC and IPS

Basic to the design of any change to Multics is the requirement that the new mechanism provide an external interface that is compatable to any preexisting interface. The Interprocess Communications Mechanism of Multics is basic to many programs and must be supported.

IPC is relatively simple to implement and offers a subset of the facilities of the eventcount mechanism. Most significantly IPC lacks the access controls afforded by using normal memory words a means of communications and coordination. To implement IPC a per-process segment of eventcounts associated with IPC channels can be maintained. In addition a per-system segment could be used to transmit messages between users. An alternative is to provide each process with a segment for receiving its messages so that the access control can be used.

Much of the complexity of IPC comes from the requirements of wired programs and programs requiring a very high degree of efficiency. Since the wired programs will be converted to use eventcounts, the IPC implementation is greatly simplified. Similarly for programs using fast IPC channels, they can be converted to use eventcounts, though they can still operate using IPC during a transition period.

The implementation of IPS has been discussed in the section on notification. The mechanism has been generalized to separate the occurance of the signal from the message associated with it. Thus one is not limited to the signals currently defined in the APT entry. For example, the quit signal can be associated with the terminal as an I/O device without requiring that it have special significance as the process' controlling terminal.

The IPC facility offers an ability not offered by event counts alone -- the sending of mesages in addition to the wakeup. This can be accomplished by using the message segment facility accompanied by eventcounts within the message segments.

Implementation

Both top-down and bottom-up views of the implementation of the layered system are applicable. The top-down views entails examining the existing Multics implementation and determining what one must change to retain is functionality. The section on **initialization** examines the implementation from the bottom-up view. The following section on **transition** examines the implementation from the view of modifying and preserving the existing Multics system.

Initialization

The bottom-up view begins by recognizing that level one of the layered Multics is sufficient for supporting a simple operating system directly without the features provided by level two. In fact this is an environment that is much more sophisticated than BOS in that it permits the use of processes and programming in PL/I.

By making the first stage of implementation the programming of an environment consisting of just level 1 primitives. An environment can be brought up without requiring the modification of the existing Multics. Most importantly, such an implementation result in a running system that can support a set of debugging tools for the later software. The psychological value of having a completely running piece of software should not be ignored. The level implementation also provides a starting point for the initialization of Multics itself and is thus a necessary first step.

The level one implementation consists of relatively few programs:

1. A program to initialize the level one system within collection one. Associated with this is a program to generate a relocation dictionary for the PAM. In addition to initializing the PAM tables, the program also creates processes for the VPC, the idles processes and an interrupt side process.

2. The PAM.

3. The VPC.

4. An interrupt side process. In order to simplify implementation I/O programs will continue to run much as they do now except all programs that normally run in response to interrupts will run in a single processes in response to the correspond eventcount being advanced. The old interrupt handlers themselves should be able to run unchanged.

5. A debugger.

That is all that is strictly necessary. An additional nicety might be to implement the existing BOS within a process so that its functions can slowly be spread to multiple processes without the need to continue to support a second 68/80 operating system and without the alternative of rewriting all of the code from scratch.

Initialization consists of loading the kernel processes necessary to support the full level one environment and then the ones needed for level two. There is a discussion on page 39 of creating VP's as necessary as part of the operation. To fill out the level one environment the following functionality must be brought up:

1. Disk Control

2. Segment Control

3. Page Control

4. The Level 2 Traffic Controller

Once the level 2 traffic controller is brought up Multics is essentially running. An answering service process can be created to create user processes. Given that processes can be created easily, the answering service does not need the primacy it currently enjoys.

Transition

One question that must be considered if the implementation of the two level traffic controller is to be taken seriously is that of how to get from the current implementation of Multics to the new one. The difficulty is that a complete transition is necessary. This is not an insurmountable obstacle in that we have had such transitions in the past as in the case of the new storage system and earlier file system flag days. While the need to convert over completely is present, the difficulty is not comparable to that of a major change to the file system. Most of the Multics system will continue to operate as it presently does. The changes consist of

I. Changes requiring new software

1. A level one initialization program must be written.

2. The basic mechanisms of the PAM, VP1 and VPC must be implemented. The VPC would be implemented in PL/I.

3. The initialization path must be modified to build up a system from one running at unadorned level one to a full Multics environment.

4. The level two traffic controller must be implemented While it must acquire all of the functionality of pxss, the level two traffic controller function is less critical -- the vast majority of the scheduling decisions are made by the PAM and the VPC. Thus the initial implementation need not be highly optimized for demonstration of its feasibility.

5. A primitive version of the amtm must be implemented to support basic accounting functions.

II. Modifications to existing software

1. A replacement must be provided for IPC using events.

2. Page control must be removed to its own process. Much of the work has been done already. This task is simplified by the fact that the page control environment is already very constrained so as not to be dependent upon the process in which it is a parasite. This is discussed in detail on page 23.

3. The interrupt handlers must be moved to their own processes. As with page control, they already operate in a constrained environment and thus providing them with their own process will not deprive them of features and will simplify them by the removal of the need to do direct interrupt handling and will remove the need for separate interrupt side and user side components. As an interim implementation all interrupt side programs

can be written unchanged within a single processes with only iom_manager begin modified.

4. pxss would simply be removed from the system.

5. System initialization must be modified and possibly redone. Much of the existing software can be used. For example disk support must still be initialized. The initialization would, however, be done as part of setting up the disk control process.

6. Present H-Procs could be simplified by replacing them with kernel processors.

7. The accounting software must be supported.

Extensions

The thesis has been concerned mainly with presenting a clean model processor multiplexing. In actual implementation some additional issues can be considered. Some of this are simple extensions and others represent a different point of view on the part of the implementor.

I. Robustness

The layered implementation provides a much cleaner structure than the current Multics system. This structuring provides an environment in which the implementation of features to make the higher levels more robust by providing a low level in which the implementation of such support facilities is simplified

1. A Level 1 debugging process.

2. Ability to recover from trouble faults -- spare repair processes.

3. Ease of timeouts and error recovery by I/O processes.

4. Daemon kernel processes.

II. Taking advantage of the implementation

This section lists some ways of taking advantage of the existing software in implementing facilities on Multics.

1. Waiting on messages.

One can associate an event counter with each message segment (or mailbox) that gets advanced whenever a message gets placed in it. This is an effective and much more powerful replacement for IPC. Some of the advantages include the ability to have InterProcess (message) Communication with access control. There is also no limit to the number of processes that can be awaiting the message. Since the transmission of the message is via a segment in the hierarchy the problem of setting up and communicating IPC channel numbers is eliminated. One final advantage of the proposed implementation is that any process with access to await a message can specify immediate attention (i.e. an interrupt) when the value is changed.

These facilities can provide a basis for a number of features. It is possible to implement notification upon the receipt of mail. Alternatively a server can be awaiting messages and then create processes the handle them (i.e. potential processes).

III. Changes to the model

1. One of the basic assumptions in the model is that Virtual Processors at level 1 are neither created or destroyed. This assumption actually complicates the system by requiring that all uses of kernel processes be predetermined. In particular the initialization of the system must be carefully planned with respect to the use of VP's. This is similar to requiring that all Multics tables used in managing the system such as the AST be determined when the system is generated, as opposed to during initialization as is presently done.

   The reason for the restriction on VP's comes from two primary sources: the need for simplicity and the attempt to carefully structure management of memory. The simplicity argument is not one of absolute simplicity but a choice of what to simplify. One must pay the price of carefully preplanning use of these processors. In particular when one dynamically reconfigures the system to add a new device (logical or physical) and one needs to dedicate a virtual processor to its management, one cannot tolerate the lack of availability of such a processor, nor can one reduce the number of virtual processors managed by level 2 since that would change the level of multiprogramming of the system.

   While the requirement of a program that is able to assign primary memory addressable by the PAM might add additional complexity to the system, it does not affect the layering of memory memory management since it is not dependent the management of virtual memory. In fact in an ideal processor such a mechanism would be simply structured such that it can be shared by both the page frame allocation mechanism and the primary memory allocation interface. The 68/80 processor is a little more complicated in that the PAM is unable to easily address more than the first 256K of memory. But this requirement is already present for I/O buffer management. To summarize, this mechanism must exist anyway for performing I/O and fits within the structure of the memory management hierarchy so that it does not really add complexity to the system.

   Thus the ability to dynamically create virtual processors would simplify the implementation without affecting the layered model of the system.

The existing implementation

A test implementation of the basic level one portion of the two level system has been completed. It supports the functions of level 1 with the exception of paging and the handling of faults reflected to user processes.

It is a modification of collection one of Multics initialization. Interrupt and fault processing have been replaced by the PAM and the VPC. The VPI interfaces for "run", "await", "advance", "crash_system" and "clear_cache" are supported. The system spawns kernel processors (including the VPC and the idle processors).

The only I/O device supported is the console typewriter. The interrupt side processing for the I/O is performed in a processor dedicated to that function. The stopped (to indicate a processor entering the stopped state) and the clock events are supported. The idle processes share a descriptor segment.

The following changes were made to the system:

1. The PAM was implemented to handle all faults and interrupts.

2. The VPC was implemented to:

   a. Convert interrupts (as noted by the PAM) into notified events.

   b. Manage the clock event.

   c. Advance the stopped event when a VPT stops.

   d. Process run and notify messages.

3. init_collections was modified to call init_basic_11 and not to call initialize_faults. PVT initialization and tape initialization was also eliminated.

4. init_basic_11 was implemented to initialize the PAM and the VPT. It spawns the VPC and idle processors.

5. create_kernel_process was implemented to initialize a VPT entry.

6. init_11_get_segment was implemented to create segments for processes' dsescriptor segment and pds.

7. The prds was eliminated.

8. privileged_master_mode_ut was modified to use the pam for entering BOS and for clearing the cache and associative memories.

9. init_sst (and the sst) was modified to remove masks was for inhibiting and generating interrupts.

10. pxss was eliminated. So was tc_data.

11. The fim and ii were replaced by stubs since at this point the system is unable to handle reflected faults. These routines will have to be redone. The same goes for emergency_shutdown and related programs.

12. The pds was cleaned up to remove unneeded storage for fault data in the header.

13. VP1 and VP_util were implemented to interface to the pam and to support the idle process.

14. run_basic_11 was implemented as a process to give periodic status messages. The moritician was implemented in a similar manner to monitor stopped processors. It uses status_report which, in turn, uses octal for typeouts.

15. interrupt_process_driver was implemented to manage the interrupt side process.

16. ocdcm_ was modified to use eventcounts to govern contention on locks.

17. A pxss was implemented to provide a write-around to addevent and notify primitives.

Glossary

Some suffixes are commonly associated with abbreviations. "E" is used to indicate an entry in table and "p" is used to designate a pointer.

AD	The Add Delta modifier causes the effect address to be computed using an indirect word and increments the value of the word by a specified amount. It is of interest because it is atomic with respect to other instructions using the modifier.
AMT	Active Meter Table.
amtm	Active Meter Table Manager.
APT	Active Process Table. The APT in current Multics would be replaced by three databases. At levels zero and one there is the VPT. The level two traffic controller maintains the APT, and for efficiency, an IPT.
AST	Active Segment Table.
BOS	Basic Operating System. This is a standalone operating system for the H68/80. It provides utility functions when the full Multics environment is not available. Such as when actually bootloading or debugging Multics.
callp	"Call Processor", an instruction implemented using the faulting mme4 and interpretted by the PAM.
camp	Clear Associative Memory PTWs.
cams	Clear Associative Memory SDWs.
IPT	Inactive Process Table. This is maintained by the level two traffic controller and corresponds to the APT, except that for reasons of locality the entries that are referenced infrequently are moved into the IPT.
L2TC	Level Two Traffic Controller.
mme4	The Master Mode Entry 4 instruction simply causes a fault. The fault handler will interpret this to be a callp operation if the fault is taken while executing in a priviliged segment.
PAM	Processor Assignment Manager.
PBM	Processor Binding Manager; older term for PAM.

PDT	Project Definition Table.
PTW	Page Table Word
SDW	Segment Descriptor Word
stcl	Store Instruction Counter plus one. This instruction is used to set a flag to be tested with sznc. It is of interest because it does not affect registers, is atomic with respect to sznc and stores a nonzero value.
sznc	Set Zero Negative and Clear. This instruction is used to test a flag set by stcl. It does not affect registers and rests the flag after test Since it is atomic with respect to stcl it is good for low level synchronization primitives.
VCPU	Virtual Central Processing Usage. A measure of the time assigned and executing.
VP	Virtual Processor.
VP1	The procedure that interfaces to the callp instruction.
VPC	Virtual Processor Coordinator.
VPT	Virtual Processor Table.
WET	Wired Event Table
WMT	Wired Meter table