PROJECT MAC
Computer Systems Research Division

December 23, 1974
Request for Comments No. 66

SEMANTICS OF VIRTUAL PROCESSOR CONTROL PRIMITIVES

By David P. Reed

The process, as implemented in the current version of Multics, is a very complex thing. Not all processes require all of the features provided in the current Multics implementation, and in fact, some of the computations currently carried on in page control, scheduling, and other operating system support algorithms are simply a variant of the process notion implemented using mechanisms such as interrupts. By using the process mechanism to structure such computations, much simpler algorithms can be used.

As part of my master's thesis research, I have been investigating a multi-level traffic control implementation which provides several classes of processes. These classes can be hierarchically ordered, in the sense that lower level processes are used in the implementation of higher level processes. A much simpler and more understandable process implementation will probably result from this structuring.

In this paper I discuss the semantics of the lowest level of process implementation, which implements an abstraction I call "virtual processors". Occasionally I will allude to the structure of the higher level process implementation, but the complete exposition of that structure will be found in a future RFC.

Note: this paper describes only the semantics of the virtual processor implementation, not the actual details of the code and data bases used in the implementation. There are many possible implementations which have these semantics, and I would like the structures to be judged on grounds of simplicity and utility, rather than on the efficiency of implementation. I will make an unjustified claim at this point by saying that I believe that I can implement these primitives very efficiently. I hope the reader will suspend any disbelief he has along this line until I actually implement these ideas in a trial Multics.

Overview

The lowest level of my proposed process implementation creates the abstractions I will call virtual processors (for brevity, I will often abbreviate "virtual processor" as VP). These objects are sources of computing resources which are much more tractable to manage than the actual physical processors that are used to implement Multics. Obviously, the ultimate source of computing resources is the physical processor, but the details of manipulating the state of the actual physical processors, and actually multiplexing them among apparently simultaneous computations are handled by the software which implements the virtual processor abstraction.

I hope to convince the reader that the semantics of these virtual processor objects are sufficiently simple and complete that they may be used to provide useful multiprocessing capability for many kernel tasks, such as dealing with external I/O devices, implementing scheduling policy, and so forth. In fact, I feel that the primitives are quite suitable for implementing directly in hardware, or in microcode. A hardware implementation of these primitives would probably be much simpler than any possible software implementation, and would certainly improve efficiency of the multiprocessor system greatly. On the other hand, I don't mean to suggest that it is difficult to implement these concepts efficiently in software on the present Multics; I think that a very efficient implementation can be made there also.

The main advantage of the virtual processor interface over that provided by the physical processor interface is due to the elimination of detail. Control of multiple physical processors such as those on the Multics system is fairly difficult due to the complexity of the processor state, and the kinds of mechanisms available for intercommunication among processors and other active modules in the computer system. Hiding these complexities inside the VP implementation will simplify the Multics kernel, which is currently constructed with far too many modules having knowledge of processor control mechanisms such as interrupts. The details of the short-term multiplexing of physical processors are also hidden behind the virtual processor interface. This fact allows isolation of long-term resource scheduling policy so it may be implemented without concern for the details of manipulating physical processors. The effects of changing the algorithms which implement resource scheduling policy can be more effectively confined in a system designed around the VP concept than in a system where scheduling policy and short-term multiprogramming mechanism each interact directly with physical processors.

The relationship of physical processors, virtual processors, and processes in the computer system can be characterized as a process of binding and unbinding through time. The binding between physical processors and virtual processors will be called assignment. Thus I will say that a physical processor is assigned to a particular virtual processor at a particular instant of time. By this I mean that the physical processor is interpreting instructions on behalf of the virtual processor, in order to carry out the function of that particular virtual processor. There is also a binding relationship between virtual processors and processes, since processes are implemented on virtual processors. This two level binding relationship is shown in Figure 1. The bindings between physical processors and VPs are changed by a simple processor multiplexing algorithm which is part of the virtual processor implementation. The bindings

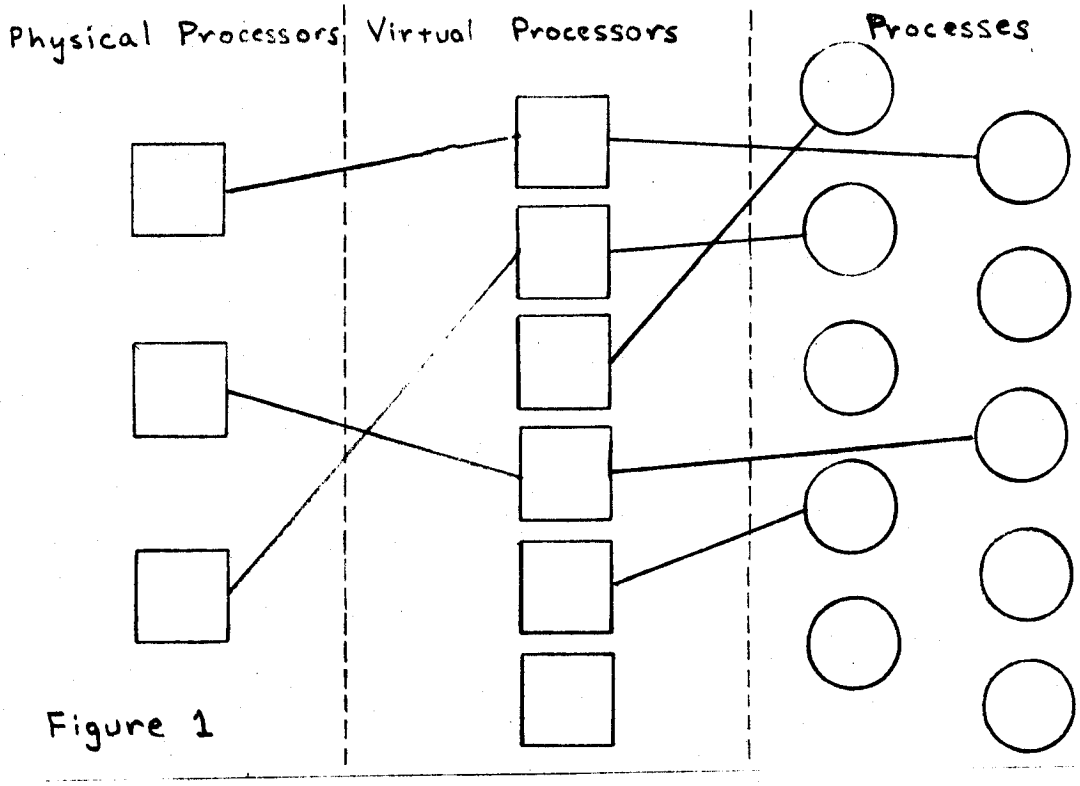

Figure 1

between virtual processors and processes are created and changed by several parts of the kernel; I shall discuss this later.

The number of VPs in the system is fixed, small, and set at system initialization. The number of physical processors is an even smaller number which varies over long periods of time due to reconfiguration. There may be a large number of processes in the system; this number varies depending on the number of users and the types of computations needed by them. The advantage of fixing the number of virtual processors at system initialization time is that it allows the physical processor reconfiguration algorithms, the virtual memory management algorithms, and the process management algorithms to be largely ignorant of each other's operations. Thus, a CPU can be configured out of the system without fear of affecting the proper execution of processes in the system. Conversely, it is not possible for the process management algorithms to interfere with ongoing reconfiguration. In addition, there is no mutual dependency between the virtual memory manager and the VP implementation. The VP implementation does not require the virtual memory manager for its correct operation, although the virtual memory manager uses processes provided by the VP implementation.

The exact number of virtual processors in the system is relatively unimportant, but should be set with the following conditions in mind. First of all, there should be more virtual processors than actual physical processors on the system; two processors cannot be simultaneously assigned to the same VP, so a lack of VPs can inhibit effective utilization of processors. Secondly, since storing information about unused virtual processors takes a small amount of room, and since the virtual processor implementation cannot use virtual memory to hold its data bases, there should not be excessively many. The correct number of virtual

processors for a system can be determined by starting with the number of VPs assigned to kernel tasks, which I will discuss in detail later. Enough additional VPs are added for use in implementing user processes to allow efficient multiprogramming of the physical processor resources potentially available to the system. That is, the number of virtual processors capable of executing instructions at any instant in time should be greater than or equal to the number of physical processors. For a 68/80 Multics configuration such as that found at MIT, there might be about 15 or 20 virtual processors, with about 10 or so dedicated to kernel tasks, and the rest available for user processes.

Interrupts will not be visible outside of the virtual processor interface. I will define an interrupt in a way very similar to the way Honeywell defines interrupts on their Series 60 processors (such as the 68/80, which runs Multics): an interrupt is a forced transfer of control which arises from an event not directly due to the execution of instructions by the processor. Other forced transfers of control, which result from programming errors like illegal opcodes or from specially set up traps encountered while trying to execute instructions, I will term faults. Some faults, such as the timer runout fault, will not be visible outside the virtual processor implementation either. Though faults and interrupts each result in a forced change of control sequence, the interrupt mechanism is much more awkward semantically. The problem is that the occurrence of interrupts is much less predictable than the occurrence of faults. One may inspect an instruction sequence and predict all of the possible faults which may occur. For example, one may look at an instruction with an immediate operand, and be certain that accessing the operand will not result in a page fault. No such guarantees are possible with interrupts. In order to simplify the semantics of the rest of the kernel, the VP implementation will handle all interrupts. The side effects which occur in interrupt handlers will be confined to a simple changes to memory words called event cells. The computations executing on virtual processors will be able to observe the external happenings which normally result in interrupts by testing and waiting on changes to event cells. I will discuss the relationship between interrupts and events further when I describe the interprocess control communication mechanisms provided by virtual processors.

VP Functional Capability

Let me describe exactly what a virtual processor is. First of all, the function of a virtual processor is to execute instructions. In this sense, a virtual processor is similar to a physical processor, except for the deletion of some functions and the addition of some new primitive "instructions", which I describe in this paper. There are certain functions available on the physical processor which are not allowed outside the domain of the virtual processor implementation. Such functions are the "load DBR" instruction, interrupt masking, the "load timer" instruction, and so forth, which are used to implement virtual processors in terms of physical processors. Restricting use of such functions could be accomplished with some kind of hardware domain mechanism which restricts certain instructions and features to particular domains. The present Multics hardware has such a mechanism for controlling instruction use, but the three domains provided (absolute mode, privileged mode, and slave mode) are not fine grained enough in control of function use. Most of the features which I wish to restrict are allowed in privileged mode, and disallowed in slave mode; unfortunately, there are some privileged mode instructions, such as the "read switches" instruction and the "connect" instruction for starting I/O channels, which will be useful at higher levels of the kernel. Consequently, auditing techniques will be required to ascertain that certain functions are not used in the kernel outside of the domain of the VP implementation.

When a VP is computing on behalf of a process, it is in one of two states, corresponding to the states of the process which is bound to the VP. These states are the bound-running and the bound-waiting states. A VP in the bound-running state is proceeding to execute instructions, at some rate determined by the physical processor multiplexing which occurs in the VP implementation. Of course, inside the VP implementation a VP may either be bound to a physical processor or it may be unbound and not actually executing instructions. Outside the VP implementation, this distinction is largely unimportant, so the distinction is not preserved at the interface. As far as the rest of the kernel is concerned, then, VPs in the bound-running state are always executing instructions.

The other computational state of a VP is the bound-waiting state. This state does not correspond to a function of a physical processor, but is a function provided by the VP implementation as a whole. A VP in the bound-waiting state consumes no physical processor resources; instead, it waits until a particular event occurs. Upon notice of the occurrence of the event, the VP enters the bound-running state. The bound-waiting state is provided by the VP for interprocess control communication. I will discuss this mechanism further in a later section.

A virtual processor has several other states which are not directly concerned with computation. These states are used in the allocation of VPs to processes. The usage state of a VP may be characterized in two ways. First of all, a VP needs to have the state data of a process in hand to be able to compute on behalf of that process. Secondly, the VP needs permission to use physical processor resources in order to execute the process. Generally this permission will be granted on a revocable basis; the resources may be denied automatically by the exhaustion of a quantum of resources, or by an explicit call in some part of the kernel, for example as part of the destruction of a process.

I will use the term bound to signify a VP which has the current copy of some process's state data loaded, and the term free to signify that this is not the case. Similarly, the term stopped will signify a VP which has been denied the ability to compute; I have no term for the opposite of stopped, so a VP which does not have the stopped attribute implicitly has permission to compute. There are three combinations of these attributes that make sense; each combination

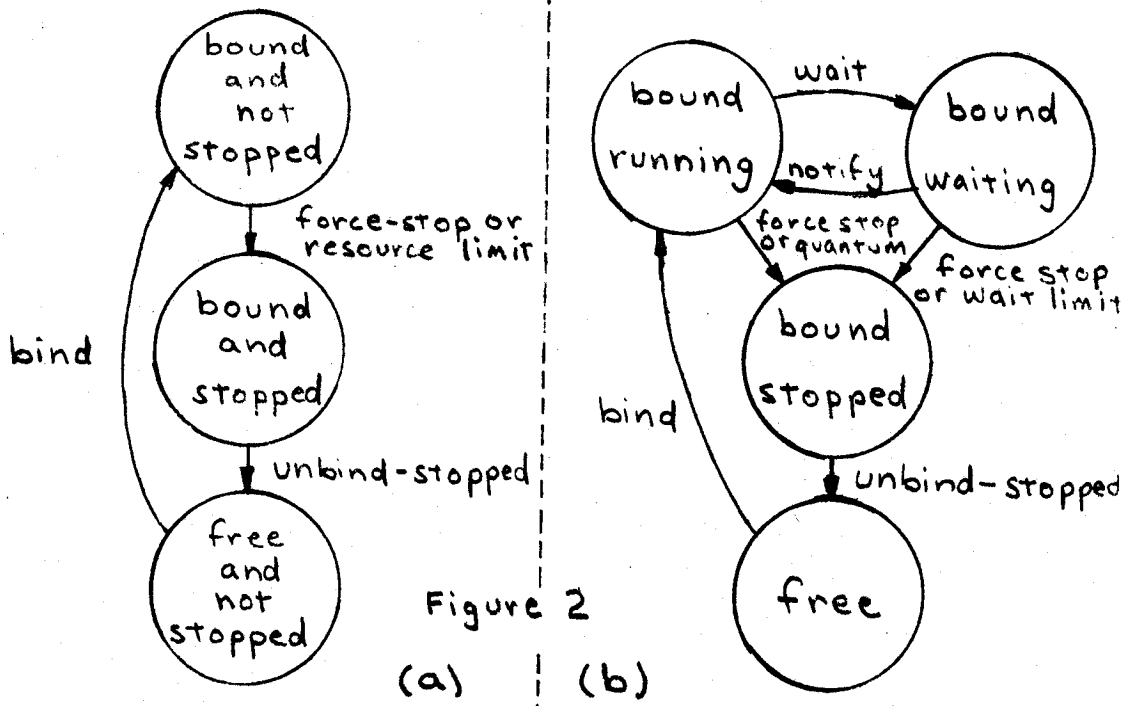

Figure 2

corresponds to a state of VP usage as shown in Figure 2a. The fourth combination, free and stopped, does not make sense because there is no need to deny permission to compute on a free VP, which has nothing to compute anyway. The bound and not stopped state is the state which performs computation in behalf of a process. This state can be divided into two states as specified earlier, bound-running and bound-waiting. Figure 2b is the composite state diagram of a VP showing this division, and the operations which cause state transitions.

The need for revocable allocation of VP computational resources is similar to the problem of reconfiguration of resources in a computer system. Schell, in his thesis, "Dynamic Reconfiguration in a Multiplexed Computer System," provides a model for the reconfiguration process which is quite similar to the semantics of the usage states I have presented. I will try to indicate the correspondence between his model and mine. My terms, bound and free, correspond directly to his terms of the same name. Where Schell says that a resource is available or unavailable for use, I use the terms not stopped and stopped. I think my terms are more specific to the case of VP resource control, and therefore more useful in description. Figure 3 shows Schell's states and their transitions. Comparing the state diagram of VP usage (fig. 2a) and Schell's model, it is clear that the VP states are a simplified form of Schell's model, with one

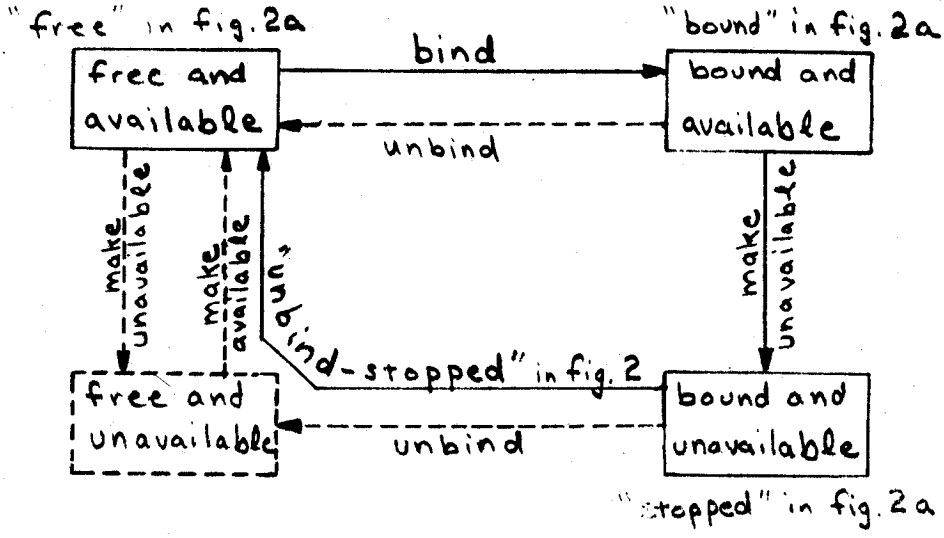

Figure 3

missing state, and several missing transitions. The missing state, free and unavailable, and its associated transitions are unnecessary in the VP semantics because we never remove VPs from the system. Consequently, we go directly from bound and unavailable to free and available. It should become clear in the following section that we never need to unbind a process from a VP without first denying it resources, so the transition from bound and available to free and available is unnecessary.

Resource Control Primitives

Since virtual processors all have exactly the same computational capabilities, there is no need to distinguish among them at the time processes are bound to virtual processors. For this reason, virtual processors have no individual identifiers visible at the VP interface. The primitive provided by the virtual processor implementation to bind a process to a virtual processor does not require the specification of a particular VP for this reason; it binds the process to any available free VP, returning a special status code if no VP is in the free state.

Virtual processors will be used in two ways. The obvious use is to implement the processes which perform computations for users in Multics. To implement such user processes, in the large numbers which Multics allows, an algorithm which I call the process scheduler will multiplex some subset of the virtual processors among user processes. This multiplexing is done primarily to control the rate of resource usage by user processes, and is a policy algorithm. The other use will be to implement various kernel algorithms which can best be implemented as cooperating sequential processes, such as I/O device control algorithms. These kernel processes should not be subject to the resource allocation policies applied to Multics user processes, so they should not be handled by the process scheduler, whose job it is to implement those policies. Since kernel processes are part of the Multics kernel, we can trust them to limit their use of resources. There is thus no reason why kernel processes can't be permanently bound to virtual processors. An example of a kernel process which must be permanently bound to a VP is the process scheduler itself. Other such kernel processes will be the processes corresponding to I/O interrupt handlers, such as the IMP and TTY device control software, and processes used to implement the page removal algorithms for primary memory and paging device. Depending on the hardware configuration, some of these processes are unnecessary. For this reason, there will be provision for creating these kernel processes at times other than initialization, and provision for self-destruction of kernel processes. The problem of reservation of VPs for uncreated kernel processes will be solved at the next higher level of process implementation.

We can presume, then, that a subset of the virtual processors will be permanently bound to the kernel processes used to implement such things as page control, process scheduling, I/O device control, etc. Some of these kernel processes have critical response time requirements for reasons of efficiency or otherwise. One example of such process might be a a process whose job it is to echo characters typed on a teletype. Another might be the process which handles a queue of I/O requests to a fast secondary storage device; it needs to respond quickly to the completion of I/O operations in order to begin new operations without losing effective channel utilization.

If all VPs were treated equally in the physical processor multiplexing algorithm, it would be possible for user processes to interfere with the responsiveness of kernel processes to events. For this reason, I introduce a static priority assignment on virtual processors which are in one of the bound states. Each virtual processor which is bound to a process will have associated with it a priority for physical processor resources. The priority value, an integer, will be constant for the duration of a binding between VP and process, and will be set at the time VP and process are bound. The virtual processor

implementation will assure that a virtual processor with high priority will receive more physical processing resource than a VP with low priority. The exact nature of this priority can be described by a statement about physical processor assignment:

> If a physical processor is assigned to a VP with priority n, all VPs in the bound-running state with priority m>n are assigned to physical processors.

This is a priority pre-emption scheme, since a change from the bound-waiting state to the bound-running state may involve pre-emption of a physical processor from assignment to another virtual processor, if that virtual processor is of lower priority.

In order to protect kernel processes from being starved of resources and responsiveness by user processes, the VPs which are bound to kernel processes will be assigned higher priorities than the priorities given VPs which implement user processes. While it may be useful to assign different priority values to each kernel process, depending on its need for quick response, there doesn't seem to be much need for assigning user processes different priority values while bound to VPs. Consequently, I intend that user processes will all be assigned the same priority value while bound to VPs, and that their responsiveness will be controlled by the process scheduler, whose job it is to implement user processes by binding them to and unbinding them from VPs.

I haven't so far specified how physical processor resources are assigned to VPs of the same priority. It can be argued that system throughput is increased by favoring VPs which have been bound to processes for a longer period of time, since such processes will have had a chance to accumulate a large share of system resources. Consequently, I extend the idea of priority pre-emption to take into account the time of binding of VP to process. Thus if we have one physical processor and two VPs in the bound-running state at the same priority, the processor will be assigned to the VP which was bound to its process first. I could have made other choices, but this rule is simple to implement, and corresponds to the ordering of eligible processes in the current Multics implementation.

In order to do its job, the process scheduler needs tools to control the resource usage of user processes while bound to virtual processors. Typically, a process scheduler will bind a process to a virtual processor for a certain amount of resource usage, called a quantum, and upon the exhaustion of those resources, the process scheduler will unbind the process from the VP, and make a new scheduling policy decision. A virtual processor in the bound-stopped state represents a process which has exhausted its quantum of resources. The primitive which binds a process to a virtual processor allows the quantum of resources to be specified. Upon exhaustion fo those resources, the virtual processor enters the bound-stopped state, and an event called the stop event is signalled. The process scheduler waits upon the stop event when it has no work to do, and upon receiving notification of the stop event, extracts all processes which are in the bound-stopped state. It can then bind more user processes to VPs. I also provide an additional way to force a VP into the bound-stopped state before its quantum runs out. A primitive allows immediate halting of a process in the case of a change of heart by the process scheduler, or the user hitting the attention key on his terminal, etc.

Although we can view VPs as interchangeable entities, there is some need, on the other hand, to run Test and Diagnostic programs on specific physical processors. This complicates the semantics of the virtual processor implementation only slightly, but is still a small wart on the design. The virtual processor control primitives will provide the capability to force a VP to be implemented by a specific physical processor while it is bound to a particular process. To handle this restriction, the primitive which binds processes to VPs will have a parameter which can be set to specify that the VP to which the process is bound may only be implemented on one specific physical processor. If the physical processor is deconfigured from the system, the VP will enter the bound-stopped state, as soon as it is possible to do so. The process scheduler will presumably take note of the fact that the process may no longer run on the deconfigured processor, and take appropriate action at this point.

Let me review the primitives which correspond to the ideas I have developed so far. I will term these primitives, as a group, the virtual processor resource control primitives. The primitives, with representative calling sequences[1], are:

VP$bind (process_id, DSEG, state_seg, priority, CPU_quantum,
wait_limit, CPU_restriction, status)

VP$unbind_stopped (process_id, resources_used, status)

VP$force_stop (process_id, status)

Briefly, the semantics of these primitives are as follows. VP$bind binds the specified process to a virtual processor, if there are any virtual processors which are currently in the free state. The DSEG and state_seg parameters specify the address space and state segment of the process. The state segment is a wired-down segment which holds all of the per-process machine state information (such as program counter, stack pointer, saved registers) which must be known to the VP implementation. It is specified by a segment number relative to the address space of DSEG. The priority of the virtual processor is set from the priority parameter, and the resource limits specified by CPU_quantum and wait_limit are set up. CPU_restriction specifies whether the virtual processor is to be implemented only by a specific physical processor. The status code indicates whether the binding was successful.

VP$unbind_stopped finds some VP which is in the bound-stopped state, and returns the name of the process bound to the VP after unbinding them. If no VPs are in the bound-stopped state, an appropriate value is returned in status. This primitive will be used in a loop to unbind all processes bound to stopped VPs. To save a call, a special status value will be returned when the last stopped VP is unbound. The resource usage of the process during the quantum is returned by this call as well. This eliminates the need for the process scheduler to have in its address space the resource accounting area maintained in each process's address space by kernel accounting algorithms.

---

[1] In the parameter lists, input values are unadorned names, while output values are underscored.

VP$force_stop forces the VP bound to the specified process into the bound-stopped state. If there is no VP bound to the specified process, appropriate error status is returned. The stop event is signalled when the VP actually enters the bound-stopped state.

Critical Sections

Many algorithms in the kernel will need to read and update kernel data bases. In order to maintain the consistency of such data bases, some kind of interlocking strategy must be used to prevent distinct processes from simultaneously modifying a kernel data base. The Multics hardware provides many good ways of accomplishing this interlocking. One mechanism commonly used in the kernel is the lock word, which is essentially a mutex semaphore. There is a problem which results from the use of this particular kind of interlocking, which is that arbitrarily long computations are executed with a data base lock locked. If a user process is accessing a kernel data base, it is possible for it to lock the data base lock, and then be unbound from its virtual processor. A particularly bad case might involve the process scheduler data base; if a process locks that data base, then becomes stopped due to lack of resources, or a force_stop, the process scheduler will not be able to access its own data base, and the system will be deadlocked. Slightly less disastrous, but still bad from an efficiency point of view, would be a case where a user locked a page control data base, then was not rescheduled for 10 minutes or so.

The examples above point out the need to provide the ability to provide processes executing kernel algorithms the ability to avoid being stopped during critical sections of code. I provide two very simple primitives to solve this problem. The first primitive, VP$critical_begin, prevents the process executing it from being stopped due to resource exhaustion or force_stops, until the other primitive, VP$critical_end, is executed. These primitives can thus be used to bracket critical sections of code.

Though these primitives allow solution of a potential deadlock problem which can arise in the implementation of the kernel, they also add a new kind of deadlock to the realm of possibility. It is obvious from the semantics of these primitives that incautious use of them in constructing the kernel can cause the virtual processors assigned to user processes to be permanently bound to those user processes, with no recourse available for the process scheduler process to deschedule them. Thus it is possible to have a system deadlock on the virtual processor resource pool. We have this problem currently in Multics. Ring zero is effectively a critical section, since a process cannot lose eligibility while executing in ring zero. If a ring zero algorithm could go into a loop, the process executing the loop would remain eligible forever. Since there is a small upper bound on the number of eligible processes, a few processes executing such an algorithm could deny service to all other users of the system.

I don't believe it is possible to design primitives which allow solution of this problem in an easy way. One common suggestion is to limit the duration of critical sections by a method similar to the 68/80 processor lockup fault mechanism. This mechanism causes a fault to occur in the processor if the processor executes for too long without sampling interrupts (that is if the user inhibits interrupt sampling for too long). The fault handler for this fault presumably samples interrupts and continues, negating the effect of inhibiting interrupts. We could limit the time duration of critical sections by using a timer, but if a kernel algorithm requires the critical section to be so long in order to operate correctly, there is no option available at the time we discover an overlong critical section other than to crash the system. The VP implementation could log the trouble if desired, so that debugging would be

possible.

Fortunately, it is almost always possible to assure that critical sections are short in duration by careful design. For example, if one always accesses a shared data base through a pointer to the contents, and if one keeps track in the data base of the time of last modification, one can reduce the critical section size required to update the data base to an exchange of pointers conditioned on the time of last modification of the data base. Such techniques are well known, although rarely exploited fully in the design of Multics. I would propose that by making use of such techniques one can reduce the size of critical sections in the Multics kernel.

Professor Saltzer has suggested that one could carry this idea to an extreme, and require that the only parts of the kernel which use the critical section primitives I have proposed be a small set of simple subroutines which implement very stylized strategies for preventing interference when updating shared data bases. These subroutines could then be proven to avoid deadlock on the pool of virtual processors.

For a different approach, one may observe that interlocked critical sections of code may be viewed as a kind of sequential process, since only one control point may be within the critical section at any one time. If there were sufficiently few critical sections in the system, we might create a separate kernel process for each, and permanently bind these kernel processes to VPs. One could then have a kernel data base updated by queueing a request in a queue intended for the "critical section process", and allowing that process to perform the modification.

If this strategy were fully utilized in the Multics kernel, there would be no need to have the two primitives I have just described. However, the resultant system would have one kernel process for each kernel data base lock, of which there are an incredible number (consider all the directories in the system). Though this scheme is obviously highly impractical in pure form, we can adapt it to reduce the number of long critical sections in the kernel, by replacing some of them with kernel processes.

Interprocess Control Communication

In order to facilitate control communication between processes bound to virtual processors and external happenings, such as I/O channel completions, things happening in other virtual processors, etc., I have provided a very simple interprocess control communication (IPCC) mechanism. This mechanism is related to the Process Wait and Notify mechanism documented in the MSPM, and discussed by Rapaport in his thesis, "Implementing Multi-Process Primitives in the Multics System."

One disadvantage of the Process Wait and Notify mechanism as described in the MSPM is that it cannot handle processes interested in multiple events at the same time efficiently. Such situations might arise from taking a page fault while testing some condition which may require waiting on an event, since taking a page fault requires waiting on an event also. Handling such a situation efficiently would require the process implementation mechanism (e.g., our virtual processor mechanism), to be able to allocate arbitrary amounts of storage to hold data about events that the implemented processes were interested in. This has serious disadvantages for the virtual processor implementation, which by its nature can only access primary memory. The Multics system overcomes this problem by two restrictions. First, the Process Wait and Notify mechanism can only be used inside the kernel, so user processes cannot interfere with control paths used by kernel algorithms. Second, the primitives allow only one event per process to be active at any time, thus bounding the amount of storage needed to remember the occurrence of control signals. The kernel algorithms which use events are structured in such a way as to avoid activating multiple events.

I propose a different mechanism to get around these problems. In my mechanism, the algorithms which use the IPCC primitives allocate storage for keeping track of the events which are active, and which have occurred. The virtual processor provides primitive instructions for observing and signalling occurrences of events.

I will define an event cell to be a user (or kernel) allocated word of memory. An event cell has a state, which is the contents of the memory word, and a name, which is the address of the memory word[1]. Invoking the primitive VP$notify with the memory word as argument results in a change to the memory word, which can be observed by the primitive VP$wait. For now, let us assume that VP$notify increments the memory word by one.

In testing some complex condition, such as the presence of characters in an I/O buffer, one runs into a race problem due to the fact that one cannot test the condition and then wait without the possibility of the condition becoming true after the decision has been made to wait, but before the actual wait state has been entered. Saltzer solves this problem with a wakeup waiting switch, in his IPCC primitives block and wakeup. The IPCC primitives I propose allow resolution of this problem by allowing one to specify the state of the event

---

[1] Here I require a system-wide unique address. One can construct one, for example, by concatenating the unique id of the segment containing the word with the offset of the word in the segment.

cell which obtained prior to the test of the condition, in the the call to VP$wait. This solution is equivalent to maintaining one wakeup waiting switch per occurrence of an event, per process.

An example of use of these primitives to obtain the next character out of a buffer updated asynchronously by another process is:

1. get current state of event cell associated with buffer.
2. see if any characters in buffer. If so, OK; otherwise continue with step 3.
3. call VP$wait with event cell and its old state obtained in step 1 as arguments.
4. go back to step 1.

The process which puts characters in the buffer must call VP$notify on the event cell after each character or block of characters is added to the buffer.

The IPCC primitives I provide are thus the following:

```
VP$wait (event_cell, old_cell_state, timeout)

VP$notify (event_cell)
```

The semantics of these primitives are fairly simple. VP$notify increments its event_cell argument, and then places all virtual processors which are in the bound-waiting state waiting for a change to that event_cell in the bound-running state. VP$wait checks to see if event_cell's state has been incremented since old_cell_state was copied from it. If the state has been incremented, VP$wait returns immediately, otherwise the virtual processor is placed in the bound-waiting state, and the name of the event cell is associated with it so that future VP$notify calls will awaken it.

To aid in the construction of a robust system, even in the face of hardware failure (loss of interrupts) or software bugs, an additional feature is provided in the wait primitive. The timeout value specifies a time when the VP will be forced out of the waiting state, even if the state of the event it has waited on has not changed. This feature also is used to provide an alarmclock facility which I describe later.

In order to allow virtual processors to reflect the occurrence of external events to processes, there is a set of special event cells provided in wired down storage, which are incremented by the VP$notify primitive every time an interrupt occurs in a physical processor. Each one of these special events corresponds to a different interrupt number; there will be 32 of these cells. In addition there will be special events notified by the VP implementation which do not correspond to hardware interrupts. The stop event mentioned in an earlier section is a special event which does not correspond to an actual external processor interrupt, but which is notified by part of the virtual processor implementation algorithm. By using these special events, we can create kernel processes which replace the function of interrupt handlers in the current version of Multics.

The astute reader may wonder what might happen if one passed to the virtual

processor wait and notify primitives an event cell which resided in a page out of primary memory. The solution to such a problem can be easily seen by viewing the wait and notify primitives as instructions in the virtual processor[1]. If any fault occurs while accessing the contents of the event cell, the wait or notify operation is set back to its beginning, and the fault is then passed on to whatever fault handler wishes to deal with it. This is analogous to the taking of a page fault in the physical processor: the instruction is backed up to its beginning and a fault is signalled, so that the instruction will be restarted from the beginning upon return from the fault handler.

The VP$notify primitive cannot signal any processes which are not bound to VPs at the time of the notify. Consequently, notification of events which are observable by unbound user processes must be implemented by a primitive which is capable of transmitting information to the process scheduler. This is a primitive of the second level of process implementation, which I do not intend to discuss in this paper. However, it should be fairly clear that such a mechanism can be constructed, using the IPCC primitives I have described, and a queue of messages to the process scheduler.

The IPCC primitives provided at the user level will be implemented in terms of these IPCC primitives. I will discuss these primitives and their implementation in my next RFC.

---

[1] Remember, I did say that the virtual processor might be simpler if implemented in hardware!

Alarmclock Interface

Since the Multics kernel provides an interface which allows processes to wait until the real-time clock reading exceeds a certain value, some mechanism must be provided by the virtual processor implementation to implement such an alarmclock facility. On the GE645 processor, there was provision for a register in the real-time clock which generated an interrupt when the real-time clock reading matched the value in that register. There is no such hardware on the 68/80 processor; however, its effect can be simulated by periodic polling of the real-time clock value by the virtual processor implementation.

The interface provided at the virtual processor level to allow virtual processors to wait on real-time events is quite simple. The timeout facility in the wait primitive is sufficient to allow waiting for an arbitrary real time. One simply calls the wait primitive with an event cell which will never be notified, and with a timeout value corresponding to the time wait must return.

Fault Interface

Since the virtual processor implementation hides some faults from the processes executing on VPs, and since the code executed in the beginning of the fault path operates in a very constrained environment, I choose to simplify the fault interface at the VP interface level. The major simplification I provide from the point of view of the process bound to a VP is that the process may specify a fault handler in terms of its own segmented address space, rather than having to deal with absolute addresses. In addition, the process need not worry about interrupt masking.

The interface is very simple, and similar to that provided in the current Multics. Each process has a "fault vector" specified by a pointer in the state segment known to the virtual processor. This fault vector is composed of two components, a fault transfer vector, and a fault store vector. Each of these vectors is composed of pointers, one for each type of fault. The occurrence of a fault which is reflected to the process associated with a VP causes a transfer through the appropriate pointer in the fault transfer vector, after storing the machine state in the place indicated by the appropriate pointer in the fault store vector. A primitive, VP$restore_machine_state, will restore the (possibly modified) machine state.

Having this fault vector settable on a per-process basis allows one to have kernel processes which respond in different ways to the same fault. An example of this might be a process which belongs to page control, which wants to crash the system if a page fault occurs in its operation, as opposed to a user process, which wants to invoke part of the kernel page control mechanism on a page fault. In general, processes which implement different levels of the system will handle some faults differently. The alternative use of a system-wide fault-handler makes certification of the kernel more difficult by reducing the clear separation between levels of the kernel. In the proposed scheme, it is easy to segregate the fault handlers for different level functions into distinct program modules; the system-wide fault handler cannot be so easily split up for certification.

I have proposed a fault vector, rather than a single fault transfer pointer and fault store pointer, for two reasons. One, the processor already has determined the type of fault, and this is an efficient way to reflect this information to the process. Two, the path of different faults is decoupled, so the page fault handler need not be concerned with arithmetic overflows. This allows certification of the page fault path independently of the path used to handle arithmetic overflows.

The proposed fault interface can be as efficient as the fault mechanism in the current Multics. The fault vector is pure and sharable among processes which execute at the same level of the kernel; user processes, for example, may share their fault vectors. Although the fault vector entries for page faults must point to wired-down memory, and the fault vector entries for segment faults must point to active segments, most fault vector entries may point to any kernel segment. Thus, overflows, illegal procedure faults, etc., need not have wired-down handlers in ordinary user processes. The proposed scheme's extra level of indirection is a small cost, which is offset by the fanning out of faults to different modules saving the need for code to do the fan out.

Processor Memory Control

The 68/80 processor has a large amount of internal memory in the form of registers, associative memories for virtual memory data, and cache memory containing copies of the contents of primary memory. In a multiprocessor system where the processors share primary memory, the copies of data from primary memory kept in each processor's associative memory and cache must be kept up to date with changes to such data made in primary memory. The mechanism for controlling the contents of the physical processor associative memory and cache contents belongs in the virtual processor implementation, since it makes use of the physical processor intercommunication mechanisms which are invisible outside of the virtual processor implementation.

A simple primitive is provided for this purpose:

```
VP$clear (selection)
```

The primitive VP$clear clears the associative memories, and/or cache, of all processors. The selection argument may specify any combination of associative memories and cache to be cleared. Control is returned to the caller after all outdated copies of the selected information has been purged from all processors' memories.

Typically, the VP$clear primitive is called after some particular change to encacheable memory, SDWs, or PTWs has been made, in order to propagate that change to all other VPs in the system. For example, if a segment is to be deactivated, first the SDW is invalidated, then VP$clear is called to clear out old copies of that SDW, after which the page table for that segment may be freed, since no processor can have a copy of the SDW from before the invalidation.

Reconfiguration of Physical Processors

The final set of primitives made available in the virtual processor implementation is a pair of primitives which allow physical processors to be added to and removed from the set which are assigned to virtual processors. The two primitives and their parameters are:

VP$add_cpu (processor_id)

VP$del_cpu (processor_id)

The semantics of these primitives are simple. VP$add_cpu starts the named processor available to the physical processor multiplexing algorithm. It does so by sending the processor a connect fault. VP$del_cpu forces the processor specified to mask itself from interrupts sent through the memory controller, and then places the processor in a dormant state. When the last CPU is deleted, it does not enter a dormant state, but rather transfers control to BOS. Normal shutdown of the system thus requires stopping all of the VPs in an appropriate order, except for the VP doing the shutdown, then deleting all processors. The system may be crashed by deleting all the CPUs. The last call to VP$del_cpu is very much like the primitive pmut$bos_and_return in the current Multics, since BOS may restart the system by transfering control back to Multics at the point after the last call to del_cpu.

In passing, let me point out that I plan to remove the concept of "bootload processor" from Multics by taking advantage of the hardware feature of the 68/80 which allows any of the processors connected to a memory controller to accept interrupts directed through that memory controller. This obviates the need for simulating interrupts while the operator changes the switches on the processor panel to make a new processor the bootload processor. In fact, the operator will have fewer switches to play with as a result of this change.