POLICY/MECHANISM SEPARATION IN HYDRA

R. Levin, E. Cohen, W. Corwin, F. Pollack, W. Wulf[1]
Carnegie-Mellon University
Pittsburgh, Pa.

Abstract

The extent to which resource allocation policies are entrusted to user-level software determines in large part the degree of flexibility present in an operating system. In Hydra the determination to separate mechanism and policy is established as a basic design principle and is implemented by the construction of a kernel composed (almost) entirely of mechanisms. This paper presents three such mechanisms (scheduling, paging, protection) and examines how external policies which manipulate them may be constructed. It is shown that the policy decisions which remain embedded in the kernel exist for the sole purpose of arbitrating conflicting requests for physical resources, and then only to the extent of guaranteeing fairness.

Keywords and Phrases: policy, mechanism, operating system, resource allocation, scheduling, paging, protection.

1. Introduction

An important goal of the Hydra system is to enable the construction of operating system facilities as normal user programs [WLP75]. Most such facilities provide some form of virtual resource (e.g. a file, a communication channel) and require base system resources (processor cycles, memory, input/output) for their implementation. We must therefore allow user-level control of the policies which determine the utilization of these resources. These policies are a major dimension of operating system variability. As many of us know from bitter experience, the policies provided in extant operating systems, which are claimed to work well and behave fairly "on the average", often fail to do so in the special cases important to us. By allowing these policies to be defined by user-level (i.e. non-privileged) programs, we make them more amenable to adaptation and tuning than they might be if buried deep in the system's kernel. Moreover, to permit each application to tune the system to its own needs, we wish to allow multiple policies governing the same resource to exist simultaneously, where appropriate.

At this point, practicality intrudes; in fact, it intrudes in several ways. First, we must assume that any user-level program contains bugs and may even be malevolent. We therefore cannot allow any single user or application to "commandeer" the system to the detriment of others. By implication, we must prevent programs which define policies direct access to hardware or data which could be (mis)used to destroy another program. That is, such programs must execute in a protected environment.[2] Further, we must not permit such a program to monopolize any resource, whether it does so intentionally or not. We must assure some "fairness" among competing policies. In addition, we must recognize that many policy decisions must be made rapidly (e.g. fast scheduling decisions are essential in order to achieve reasonable response). Given that user-level policy programs must execute in their own protection domains, and that domain switching is costly on C.mmp, it is impractical to invoke such programs each time a policy decision is required.

Thus, we compromise. We give this compromise a name: the principle of *policy/mechanism separation*. Policies are (by definition) encoded in user-level software which is external to, but communicates with, the kernel. Mechanisms are provided in the kernel to implement these policies. In this context we use the phrase "kernel mechanisms" to mean two distinct but related things.

In the first instance we mean simply a "safe" (protected) image of a hardware operation. Thus, for example, we never allow a C.mmp user to manipulate i/o device control registers directly. To do so would allow that user, possibly inadvertently, to overwrite an arbitrary portion of memory. We do, on the other hand, provide a mechanism, a kernel operation, whose only effect is to manipulate such device control registers after appropriate validation.[3] Mechanisms such as this exist purely to insulate the system and other users from a misbehaving policy program.

In the second instance a kernel mechanism may actually be a parameterized policy. Parameterized policies provide the means by which overall, long-term policies can be enforced by user-level software, and at the same time avoid a ponderous domain-switching mechanism for decisions which must be made rapidly. The existence of such kernel mechanisms seems to

[1] This work was supported by the Defense Advanced Research Projects Agency under Contract F44620-73-C-0074 and is monitored by the Air Force Office of Scientific Research.

[2] Obviously, all programs must be denied such liberties, but policy-making programs frequently require access to information which might normally be considered privileged.

[3] To perform such an operation the user must have a capability for the device object with the appropriate rights [CJ75].

contradict the earlier assertion that policies are strictly user-level pieces of software; hence, we will first attempt to defend the appropriateness of parameterized policies in the kernel.

The kernel cannot possibly support all conceivable user-defined policies, since some will violate fairness guarantees or protection requirements. At best it can provide a mechanism adequate to implement a large class of desirable resource allocation policies. The decision to exclude certain policies is itself a lower-level (i.e. kernel) policy of a sort, but we intend it in a practical sense to be a non-limiting restriction which disallows only "undesirable" policies. Nonetheless, it is still possible that the precise nature of a particular kernel mechanism discriminates against certain acceptable policies, making them intolerably difficult or expensive to implement. In such cases we have obviously failed to attain the desired goal. The class of implementable policies is clearly determined by the type and number of parameters provided by the kernel mechanisms, and the extent of that class is therefore dependent upon the designer's insight.

The identification of a piece of software as "policy" or "mechanism" is a relative one. The implementor of a (virtual) resource establishes policies for the use of that resource; such policies are implemented with mechanisms provided by external software. Thus a hierarchy exists (under the relation "is implemented using") in which higher-level software views higher-level facilities as mechanisms. At the base of this hierarchy rest the kernel mechanisms, several of which we consider below. Thus, in principle, the kernel contains only those facilities which we feel could not profitably be implemented using lower-level mechanisms.[4] These lowest-level software mechanisms are themselves in fact policies for user of the hardware resources.

The remaining sections of this paper discuss three important examples of policy/mechanism separation in Hydra. The examples are representative of the applications of this principle but do not exhaust the situations in which it is relevant. In the sequel the discussion frequently refers to basic Hydra concepts, e.g. the particular notions of capability and object that Hydra defines. Familiarity with these notions is assumed; an adequate understanding can be acquired from [WLP75,Wul74a].

2. KMPS

KMPS (the Kernel Multiprocessing System) is that portion of the kernel which implements a mechanism to support policies for scheduling user processes. Parameterized schedulers are not a new idea [BS71], but the intent in Hydra differs somewhat from most other systems in which this concept is employed. Our goals with respect to this mechanism were: (1) to permit a process scheduler to run as a user level process, and (2) to allow multiple schedulers to run concurrently. We call a user-level scheduler a ***Policy Module*** (PM).

Each process has an associated PM that is responsible for making the scheduling decisions related to that process; typically a single PM will be responsible for scheduling several user processes. Thus at some point in time, one PM might be controlling normal "time sharing" user processes. A second PM may simultaneously be scheduling background "batch" tasks. A third PM might have special knowledge of the manner in which its processes are cooperating and use this information to effect a more efficient use of machine resources than otherwise would be possible. Lastly, we may have a fourth PM being debugged. KMPS guarantees that an error in one PM can have no deleterious effect on the remaining PMs.

The KMPS mechanism is in fact a parameterized policy, for two reasons. First, scheduling decisions must be made rapidly; we cannot afford to invoke the overhead involved with switching protection domains (i.e. a Procedure call) each time a scheduling decision must be made. Second, KMPS serves as a focal point at which it is possible to adjudicate the competing demands of distinct PMs. Since KMPS bases its short-term scheduling decisions on the parameters set by the PMs, it provides the mechanism by which short-term scheduling policy can be made by the PM. Since any short-term policy must be implemented by use of this mechanism, not all scheduling policies are possible or practical; however, we believe the mechanism provided is general enough to permit a large number of interesting ones.

Before proceeding with the details, we present a general description of the interaction between a process, the PM which controls it, and KMPS:

> A process is an object. The PM which is responsible for the process must have a capability for it with appropriate rights. Assuming that the PM has such a capability, it may perform certain kernel-defined operations. For example, it may set the parameters which will control the short-term behavior of KMPS with respect to that process and may also "start" the process, that is, allow KMPS to schedule it.
>
> The act of starting a process yields control of it (temporarily) to KMPS. When a process is started, its pages will be brought into core (by the paging mechanism to be discussed in section 3), and will remain in core until the process "stops" (i.e. leaves KMPS). As soon as the process is present in core, KMPS will begin scheduling the process for execution, and the process will then compete for processor cycles with all other processes which have been started by some PM. As noted previously, the behavior of KMPS with respect to the process is determined by the parameters set by the PM.
>
> Under any of several conditions the process may be "stopped" (removed from KMPS); we shall mention some of these conditions later. For the moment the important property is that when a process is stopped, KMPS returns control of it to the PM, and the core it occupies is freed. KMPS always sends notification of the fact that a process has been stopped back to its controlling PM. The PM then has the option of restarting the process immediately or waiting some period before doing so. The PM may also alter the process's scheduling parameters at this time.

[4] In practice, we include a few facilities in the kernel which are logically non-primitive, but which, for efficiency reasons, require more direct access to the hardware. Alternative hardware architectures would permit exclusion of these mechanisms from the kernel.

Thus, if we look at the execution history of a single process, we would see periods during which the process is idle (stopped) and under exclusive control of the PM, and other periods during which it is being actively multiplexed onto the processor resources by KMPS. Closer examination of the various active periods might show quite different behavior if its scheduling parameters were set differently each time it was started.

2.1 Scheduling Mechanism

Viewed externally, KMPS defines two types of objects: objects of type *process* and objects of type *policy*. A process object corresponds to the usual informal notion of a process; that is, it is an entity which may be scheduled for independent execution. The data part of a process object, called the *Process Context Block* (PCB), holds state information (e.g. scheduling parameters). The C-list of a process object holds capabilities corresponding to a stack of LNS (*Local Name Space*, see [Wul74a]) objects. The 'top' LNS in this stack defines the current 'protection domain' of the process. Each LNS is a dynamic incarnation of a PROCEDURE. Both LNS and PROCEDURE (as well as PROCESS and POLICY objects) are object types predefined and supported by the Hydra kernel.

A policy object is the kernel's image of a PM; furthermore, since each process object points to precisely one policy object, the kernel knows which PM to inform when a process is stopped. Actually, a policy object is not a PM; rather, it is a 'mailbox' to which the kernel sends 'messages' to inform a PM that one of its processes has been stopped. We shall see the significance of the distinction between PMs and policy objects in a moment.

KMPS defines a number of operations[5] on process objects:

> START(process): START of a process causes KMPS to enter the process into its scheduling queues. KMPS will then select the process to run based on its parameters as set by its PM.
>
> STOP(process): STOP will cause the specified process to be stopped (if it is running) and removed from the KMPS scheduling queues; notification that the process has been stopped is returned to the PM through the mechanism described below.
>
> SETPCB(process,data): SETPCB allows the PM to set those fields in the PCB (process context block) which control KMPS scheduling decisions.
>
> GETPCB(buffer,process): GETPCB allows the PM to retrieve information from the PCB. Specifically, certain useful process state information (e.g. elapsed runtime) may be determined in this way.

KMPS also defines a number of operations on objects of type POLICY; the following are simplified versions of two of these operations:

> SETPOLICY(process,policyobj): This operation associates the given policy-object with the process.
>
> RCVPOLICY(buffer,policyobj): This operation performs a 'receive' from the mailbox of messages in the policy-object concerning processes under this PM's control. If the mailbox is empty, the process executing the RCVPOLICY operation is blocked until a message arrives. When a message is eventually received, it is stored into the specified buffer in the process's core. This information identifies a particular process (under the PM's control) and indicates why KMPS felt a policy decision was required on the process's behalf.

Earlier we skirted the issue of exactly what a policy module is. It should now be clear that a PM is nothing more than a process which possesses a capability for a policy object (with appropriate rights). In addition, a PM holds capabilities for the processes under its control, again with rights adequate to perform the above operations. In fact, we can establish an operational definition of a PM as a set of programs (procedures and processes) which possess such adequately endowed capabilities for some collection of processes and a policy object. Notice that, with this scheme, what is logically a single PM may be in fact implemented as a number of processes. For example, several processes may be waiting on the mailbox of the same policy object, in effect acting as multiple servers, and cooperating to effect a single scheduling policy. This 'multiple server' ability may be essential to smooth response in a busy interactive environment.

2.2 The Parameterized KMPS Policy

The scheduling policy parameters of a process as set by its PM include priority, processor mask, time quantum, and maximum current pageset (CPS) size (see section 3). KMPS uses a priority scheduling algorithm; thus a process will run before another process scheduled by the same PM at a lower priority.

The processor mask of a process specifies which processors in the system are permitted to run the process. The mask is necessary since not all processors in C.mmp are identical; for example, only certain processors have floating point hardware. Thus a process exploiting the additional hardware will want to restrict its scheduling to the appropriate processors.

The time quantum specifies the amount of execution time a process is to receive before it is to be stopped. The time quantum is broken up into time slices (0.5 seconds maximum) and number of time slices, both of which are specified by the PM. At the end of a process's time slice, KMPS may elect to run another process. After the specified number of time slices, i.e. at the end of the time quantum, the process is stopped and returned to the PM via the policy object mailbox.

[5] For clarity, we omit discussion of the rights requirements imposed on the parameters to these operations. It should be obvious that appropriate rights are necessary to control application of these operations to processes. In addition, descriptions of the operations presented here are somewhat simplified.

A process may stop for other reasons, e.g. blocking on a semaphore or returning from its base LNS (process termination). In these cases, as when the time quantum ends or an explicit STOP occurs, KMPS ceases execution of the process and places an entry in the appropriate policy object mailbox. Thus the PM is informed when processes stop (or, as in the case of semaphores, can be restarted) for non-PM-induced reasons. Let us consider the case of semaphores in more detail.

When a process blocks on a semaphore, KMPS stops it as previously described. However, the fact that the process is blocked is not lost, and when the process is subsequently unblocked (by a V of the semaphore by another process), KMPS again places an entry in the policy object mailbox. This second entry is simply a notification that the mechanism of unblocking has occurred; the policy of rescheduling the infeasible process for execution is still the responsibility of the PM. Until a subsequent START operation is performed, the process will not be rescheduled. Thus KMPS only implements (and enforces) the blocking and unblocking mechanisms, leaving scheduling policy to the PM.

There are a number of additional policy/mechanism separation issues relating to parameterized policies. We will deal with them at length in the section on Control Protection.

2.3 Multiple Policy Modules

At the beginning of this section we stated two goals: (1) to permit user-level control of the scheduling policy, and (2) to support multiple scheduling policies simultaneously. The preceding discussion has focused on the first goal; we now turn our attention to the second.

The desire to allow multiple PMs (controlling disjoint sets of processes) is a natural one. We would like to assign some fraction of the processing resources to each PM, and guarantee that the individual PMs stay within their allocated limits. This puts KMPS in the position of enforcing a fixed policy for CPU usage, a policy that has very little dynamic variability. Clearly, then, the kernel is not devoid of policy, yet the policy/mechanism distinction does not break down; where policy is clearly embedded in the kernel, it is there for the sole purpose of assuring fairness to (competing) user-level policies. As we shall see, KMPS in its role of PM-adjudicator is not trying to optimize short- or long-term CPU utilization, but is only assuring each PM that it may use an agreed-upon fraction of CPU resources.

We will not discuss the details of the guarantee algorithm, since it has not been implemented (though some simulations have been performed). In general terms, however, the algorithm allocates to each PM a "rate guarantee". That is, the PM can expect to receive, upon request, a fixed percentage of the CPU cycles deliverable over a given time interval. This is a guaranteed minimum; available excess cycles will be made available to each PM. We refine this guarantee somewhat to account for the heterogeneous nature of the processors by partitioning the CPUs into classes and providing each PM a rate guarantee within each class. By "guarantee" in this context, we mean that the PM will, with high probability, receive on demand its allocated fraction of the resources.

To illustrate that the policies which the kernel is obligated to implement are introduced to enforce fairness criteria alone, we list the goals of the scheduling guarantee algorithm:

(1) Each PM should receive a guaranteed percentage of the CPU time available within each processor class.

(2) If a PM does not consume its guarantee during any interval, the excess should be distributed fairly among other PMs.

(3) If a PM fails to receive its guarantee during an interval, an attempt should be made to give it slightly more than its guarantee in succeeding interval(s).

(4) A process's priority should only affect its scheduling with respect to other processes of the same PM.

(5) When a process of a given PM is selected to run on a processor, it is generally the highest priority process started by that PM that can run on the available processor. Processes assigned the same priority level by a PM are scheduled in a round-robin manner.

3. Paging

This section examines the policy/mechanism issue in the context of paging operations. As with KMPS, the mechanism is in fact a parameterized policy in the kernel; but as we shall see, control over paging is more indirect than control over scheduling. Before proceeding to the issue of policy, however, we must first examine the user-visible properties of the paging mechanism. These properties are, unfortunately, strongly affected by the hardware architecture of C.mmp.

The single largest impact results from the PDP-11 processor; specifically from the fact that it is able to generate only a 16-bit address. Thus user programs at any instant may address at most 64K bytes, or 32K words. The second largest impact arises from the fact that the relocation hardware divides the user's address space into eight 8K-byte units called *page frames*. Since this is a rather small address space, much of the design of the paging system is oriented toward making these restrictions somewhat more comfortable. A third impact of the hardware architecture is that the relocation hardware is incapable of supporting demand paging (in the conventional sense of that term). A single PDP-11 instruction may access as many as six distinct pages and may have side effects on the processor registers between some of these accesses. The C.mmp relocation hardware does not retain sufficient processor state to allow these side effects to be undone if a fault occurs. As a result, Hydra must insure that pages referenced by the relocation hardware are actually present in primary memory.

In the following material we shall usually use the term *page* to refer to an object, in the Hydra-technical sense of that word, of type PAGE. In some contexts the term *page* may also mean the information contained in the PAGE object. The term *page frame*, or simply *frame*, will usually be used to refer to the area of physical primary memory (core) in which the information content of a page object resides. The term *frame* is also used to indicate a portion (1/8th) of the user's address space. Context should disambiguate these uses.

3.1 Paging Mechanism

Since pages are objects, a user program may, and generally will, have one or more capabilities which reference specific pages. These capabilities may be in an executing LNS or contained in some object, which can be named by a path rooted in the LNS. Possession of a capability for a page, however, does not make it addressable. In particular, it is possible that many more pages may be named through some particular LNS than can be simultaneously addressed by the C.mmp hardware. Thus the paging system defines means by which the user may specify and alter the set of page objects which are physically present in primary memory and which may be directly accessed at some instant.

Each active LNS has associated with it a CPS (*current page set*) and an RPS (*relocation page set*). The set of pages referenced by the CPS is guaranteed to be in core while the LNS is executing. The set of pages in the RPS (a subset of those in the CPS) is precisely the set of page frames which are currently named by the relocation hardware of C.mmp. Thus the pages in the RPS are those whose information may be accessed directly by instructions executed by the user's program. Of necessity the RPS must refer to eight or fewer pages. No such size restriction exists for the CPS.

The LNS, CPS, and RPS effectively define a three-level memory system -- those pages nameable by, or through, the LNS, those named in the CPS, and those named in the RPS. Normally each of these is a subset of the preceding (the exception being that once a page is loaded into the CPS it may be deleted from the LNS). Each of the sets also implies more rapid access than its predecessors. The pages in the RPS are both in core and addressable, those in the CPS are in core but not addressable, and those in the LNS may not even be in core.

For small programs the three sets may be identical, and the user need not concern himself with paging. For larger programs, larger than 32K words, the user is required to manage these sets; the way in which the user does this may significantly affect the performance of the program. Several kernel operations are provided to manage these page sets; slightly simplified versions of two of these follow:

> CPSLOAD(Cpsslot,Page) causes a capability for a page object to be loaded into the specified slot of the current CPS. There are two side effects of this action. First, any previous capability in this CPS slot is implicitly deleted; the page object named by the deleted capability is no longer required to be in core by the current LNS, hence it may be eligible to be swapped out. Second, the page named by the capability just placed into the CPS may now have to be brought into core; that is, it may have to be swapped in.
>
> RRLOAD(Rpsslot,Cpsslot) causes appropriate information to be placed into the hardware relocation register associated with 'Rpsslot' so that the page referenced by the capability in the specified slot of the CPS may be directly addressed. Of course the information previously contained in this relocation register is removed at the same time; however, this page remains in the CPS and hence is still resident in core.

Several additional points need to be made before we deal with the policy/mechanism issues. First, notice that a page is conceptually in core as soon as a user moves a capability for a page object into the CPS. In practice the page need not be physically present until it is moved into the RPS. It is therefore reasonable to allocate a page frame as soon as a CPSLOAD is performed, and to initiate an i/o transfer if necessary, but not to suspend the program unless an RRLOAD is performed before the transfer is complete.

Second, only the "top" CPS of each process, that is the CPS associated with its current LNS, needs to be core-resident.

Third, each procedure object contains information about the initial CPS/RPS configuration to be associated with an LNS. This information is given in the form of a set of pairs of indices; these pairs invoke implicit CPSLOAD and RRLOAD operations when a procedure is called. Thus whenever a procedure is called, it begins executing in an environment in which a specified set of pages are in core and addressable. This initialization is performed after the LNS has been incarnated; thus page objects passed as actual parameters may be part of the initial environment.

Fourth, the kernel mechanism which provides facilities to control paging policy is subject to several constraints -- including the physical memory size. The most important constraint, however, is imposed by the assumption made by KMPS that the RPS for every feasible process is in core. Note that as in all allocation systems, allocation decisions for distinct resources are not independent. Even if it were feasible to completely decouple, for example, paging and scheduling decisions, it would probably be unwise to do so. The particular coupling described below is possibly too strong, but it is the only formulation we have as yet found to be practical to implement.

3.2 Paging Policy

There are three issues of paging policy that we have had to face in designing the Hydra kernel: process paging, guarantees, and replacement. We will present each briefly and then discuss them in more detail.

All pages in the top CPS of each process in KMPS must have a page frame allocated in core. Two alternatives present themselves. Either the PM directly handles the paging for each process under its control or as much of the mechanism as possible (and hence, some of the policy) is left to the kernel. We have chosen the latter alternative, tightly coupling the mechanisms for paging and scheduling.

Multiple PMs may be competing for limited memory in which to run their processes. We will describe a mechanism that allocates a guaranteed number of pages to each PM in order to assure fairness.

Whenever pages are placed in core, other pages may have to be displaced. Thus, there is a need for a page replacement policy. One might assume that each PM could manage its own guaranteed set of pages. The kernel would only need to provide a mechanism that would allow a PM to designate which page among its guaranteed set could be replaced.[6] For various reasons, that assumption is inadequate and a more sophisticated mechanism (one which includes quite a bit of policy) has been provided instead.

3.2.1 Process Paging

We have decided to let the kernel handle process paging essentially automatically. There are three situations in which this automatic paging is done:

(1) Whenever a PM "starts" a process, i.e. gives control of it to KMPS, the paging mechanism will cause the pages in the "top" CPS to be brought into primary memory.

(2) Whenever a process "stops", i.e. KMPS yields control of the process back to the PM, the pages of the "top" CPS are made eligible to be transferred out of primary memory (except, of course, for any shared pages which are referenced by the "top" CPS of some other runnable process).

(3) Whenever the top CPS of a process changes (i.e. via call or return of an LNS) the pages in the new CPS are automatically brought into core and those in the old CPS are made eligible to be transferred out of primary memory (except, as above, for shared pages).

Obviously many paging policy decisions are implicit in the PM's actions in starting or stopping processes. By choosing which processes will run at any instant, the PM is also choosing the pages to be core-resident. This is not, however, adequate control. An individual process may alter the size of its CPS, and hence its memory requirement. It may do this directly, by CPSLOAD's, or indirectly by calling (or returning from) LNSs with different, possibly radically different, storage requirements. Thus, unless more control were provided, it would be possible for wild fluctuations to occur in the storage requirements of a collection of processes unbeknownst to the PM controlling that collection.

One could, of course, have the kernel mechanism stop a process and send a message to the PM each time a paging decision must be made, i.e. each time the process invokes CPSLOAD or changes its LNS. Just as with KMPS, we choose not to do this, but to have the PM yield control over "local", or "short term" paging decisions to the kernel. As with KMPS we also parameterize the kernel paging policy and allow the PM to set these parameters -- thus influencing the short term policy.

In the case of paging there is at present only one (per process) parameter. The PM may specify, for each process under its control, the maximum CPS size for that process. Thus, in effect, the PM may place a limit on the number of page frames a process may use at any instant. Only if a process, as the result of either CPSLOADs, calls or returns, exceeds this limit, is the process automatically stopped. At such points, the PM may intercede and decide to allow the process to proceed (by altering the CPS limit) or allow the process to remain stopped until the requisite memory resources are available. The point, of course, is that as long as a process operates within specified resource limits there is no need for the PM to interfere (or to evoke the overhead associated with doing so).

3.2.2 Paging Guarantees

Each PM is guaranteed a specified number of page frames. The sum of these guarantees across all PMs is equal to the number of available page frames. Thus the guarantee acts as a limit as well, for the sum of the CPS limits for the processes under the control of a particular PM and which are runnable (in the sense of being under the control of KMPS) is constrained not to exceed the guarantee. Thus, each PM can, and should, attempt to control the processes for which it is responsible in a way which effectively utilizes the memory guaranteed to it.

Unfortunately, this paging guarantee is too strong, since it restricts resource allocation rather than merely insuring it, and thereby leads to inefficient use of primary memory. For example, a PM has no easy way to know to what extent the processes it controls share pages. Hence the number of frames used by a collection of processes may be less than the sum of their CPS limits and the PM may not know it. Under such circumstances the PM will use the memory resource less effectively than it might. This problem might be resolved fairly easily for pages shared between processes controlled by the same PM, but it is difficult to see how to resolve for pages shared between processes controlled by distinct PMs.

A possible solution to this problem allows a PM to exceed its guarantee as long as page frames are available. Availability may be due not only to sharing of pages by processes within or across PMs, but may be due to underutilization of page frames by another PM.

This solution is not without difficulties. The extra page frames may suddenly become unavailable, if, for example, a PM under guarantee wishes to utilize its entire guarantee. In that case, we must embed another bit of policy in the kernel to determine how those extra pages are to be reclaimed from PMs over guarantee. An obvious example solution involves stopping the lowest priority process of the PM most over guarantee and continuing doing this until the requisite number of pages become available.[7] Other factors that might affect the policy include the number of non-shared pages used by the process and the length of time it has been in KMPS. We have not yet determined the best criteria for this kernel paging policy.

3.2.3 Page Replacement

Whenever a new page is made core-resident, it is necessary to choose a frame to hold it. This choice is called a *replacement policy* because it (potentially) implies replacing a page which is already in core. This choice can significantly affect system performance. One hopes that if the policy is clever, frequently used pages will not be replaced and hence will not have to be reloaded into primary memory.

[6] The kernel would of course have to check that the page did not belong to the top CPS of any process associated with that PM.

[7] Remember that when a process stops, all pages in its top CPS (except those shared with runnable processes) become eligible to be transferred out of primary memory.

There are two factors that make page replacement more complex than originally suggested in the introduction to this section:

(1) As long as a process does not exceed its CPS limit, it need not be stopped and may load and unload pages from its CPS without its PM's knowledge. Thus the PM does not know exactly which of its pages are actually in core.

(2) Sharing of pages, especially by processes under the control of different PM's makes it difficult for a PM to know when a page is really eligible to be swapped out.

Even if the information needed could be made available to a PM, it might not be valid when used, since it can change quite dynamically. The difficulties associated with (1) are a direct result of our decision to let the kernel handle "short-term" paging decisions. But, the difficulties associated with (2) are independent of the amount of direct control a PM has over the paging of its processes. It appears to us that unless the kernel is the ultimate arbiter of replacement, memory will be under-utilized in the case of sharing of the degree we expect in Hydra.

The yielding of replacement policy to the kernel has another benefit, one associated with protection. We neither require nor expect that PMs will be correct or even trustworthy. Therefore, it is important that a PM be able to gain as little information as possible about processes under its control. Paging mechanisms different from the one adopted for Hydra have the disadvantage that far more information regarding the paging behavior of a process (perhaps even including the identity of individual pages) would be made available to a PM. This would provide an additional covert channel for leaking sensitive information [Lam73,CJ75].

A substantial body of literature exists on the behavior of various replacement algorithms [e.g. Den68,Bel66,Den70]. Two properties of the current context suggest an algorithm slightly different from the usual ones. First, the CPS is, in effect, the "working set" of a process. These are guaranteed to be core-resident. The replacement algorithm chooses only among the frames occupied by pages not present in the CPS of any processes which can be scheduled by KMPS. Second, the kernel is able to keep track of information about how various pages are used -- it can determine, for example, whether the page belongs to the "top" CPS of a "stopped" process, the nth CPS (from the top of a CPS stack) of a running process, whether it is not in any CPS, etc. Thus the kernel can make a reasonable guess at the likelihood that a page will be needed again in the near future.

Whenever a page becomes eligible to be swapped out, the first thing done is to make sure that a valid copy of the page exists in secondary memory. The "dirty bit" in the hardware relocation registers is examined to determine whether the page has been modified since it was last written out; if the dirty bit is set, the page is copied to secondary memory and the bit reset. As soon as it is certain that a valid copy of the page exists, its frame is made eligible for replacement.

The scheme for replacement is to assign a "priority" to each replaceable page and to replace the pages in reverse-priority order. Pages of the same priority are replaced on a least-recently-used basis. The priority of a replaceable page, P, is defined as

$$\text{Priority}(P) = \sum_{r \in \text{Ref}(P)} C(r)$$

where 'Ref(P)' is the set of references (in some CPS) to P and 'C(r)' is a cost-function. The value of C(r) depends upon the nature of the reference (from a stopped process, second position of the CPS stack of a runnable process (one in KMPS), etc.). If the page again becomes part of the top CPS of a process in KMPS before it has been replaced, a transfer from secondary memory is avoided. The nature of C has not yet completely been fixed[8] and will depend upon additional experience and simulation.

At present, a PM has no way to specify information it may have concerning page utilization to the kernel mechanism. A PM may know that a process may not be stopped for long. In that case, it would be desirable that pages referenced from that process's CPS have a higher cost value. One might even imagine extending the mechanism so that, if a PM had a way to increase the priority of a process's pages prior to starting it, the kernel mechanism would try to insure, by prepaging if necessary, that the process's pages will be in core. One might also have to provide a way for a PM to designate a certain number of its guaranteed page slots be reserved for such highly valued pages even though they are not actually in use by runnable processes.

A user program also may have information relevant to the paging mechanism. For example, an LNS which constantly moves pages in and out of its CPS may want to identify certain "important" pages. Such pages, though not in the CPS, should be retained in core if possible and at the expense of other pages not present in the CPS. In terms of the current formulation of the paging mechanism, the LNS wants to influence the priority calculation for its "important" pages. At present, there is no way to accomplish this weighting of the page priority, although, as with weighting desired by the PM described above, it could probably be accommodated without much difficulty.

3.2.4 The Effect of the Hardware

The hardware architecture has had a major impact on the form of the mechanism for paging presented to the user. The inadequacy of the information retained in case of a fault combined with a limited address space prevents demand paging and forces the user to explicitly define a working set (CPS). Yet while these design "features" have had profound effects on the mechanism, they are actually quite independent of the policy/mechanism separation issues discussed above.

If hardware permitted an extremely fast context swap, PMs could exercise much tighter control over process paging. Each time a process reduced or increased the size of its CPS, the PM could be invoked, thereby removing short-term paging decisions from the kernel. (To an extent, this applies to scheduling decisions as well.)

[8] At present C(r) is 2 if r is referenced in the top CPS of a runnable process, 1 if r is from the next to top CPS of a runnable process, and 0 otherwise.

However, as with scheduling, the real need for certain kernel-centered policies arises from the presence of multiple policy modules, no one of which is guaranteed to be correct. The problems related to guarantees, sharing and replacement, are all induced by multiple PM's and are present no matter whether CPSs change implicitly or explicitly.

4. Protection

In contrast to scheduling and paging, the Hydra protection structure provides a clear separation of mechanism and policy. It is the intent of the design to provide a set of facilities which can solve a broad class of protection problems, but which a user program need not use if it is not concerned with protection. In this regard the kernel protection mechanism is a passive one, while scheduling and paging demand at least a limited amount of PM action.

4.1 Access Protection

Hydra provides a capability-based system which very naturally supports selective access to information. The particular character of the protection mechanism exhibits some aspects of policy, especially in the sense that certain kinds of protection structures can be constructed more naturally than others. For example, that type of protection best known as information hiding [Par72b] is available directly in Hydra through the construction of subsystems built around extendible types [WLP75]. Only indirectly (though quite naturally) through the construction of subsystems can users build other protection structures, for example, access control lists or military security classification systems [JoW74]. At the same time, this structure provides an elegant framework for mechanisms which permit the solution of a number of important protection problems, including confinement and selective revocation [CJ75].

In general, the kernel does not force a user to share information with or execute code supplied by another user. Where cooperation is desirable, the kernel supplies a variety of mechanisms that permit users to generate policies that will protect them insofar as is possible. These are described in [CJ75] and will not be dealt with here.

4.2 Control Protection

In one case, the kernel does force interaction. If a user program expects to get any work done, it must interact with a PM that it may not necessarily trust.[9] Even though a PM does have a capability for all processes under its aegis, it in fact cannot gain access to any capabilities or data in any LNS or CPS associated with the process. In addition, the nature of the paging and scheduling mechanisms make little other information available to the PM. So, interaction with a PM does not provide any significant additional concern over access protection. Protection of control is another matter.

A procedure may be called by any program that can gain an appropriate capability for the procedure. Thus, an LNS (incarnated from an arbitrary procedure) in general has little control over the PM under which it is being scheduled. A PM that schedules it erratically (or perhaps not at all) may cause as much (or more) harm as a failure in access protection. The kernel provides a number of mechanisms that support policies aimed at protection of control.

4.2.1 PM Identification

As an example, consider a spooling subsystem. Through the kernel's general message system (which will be described in a later paper), a number of processes send output to the subsystem. The subsystem is simply an LNS that has capabilities that permit it to receive these outputs plus a capability that permits it to output to the line printer. This LNS executes in some process which runs under some PM. The builder of the spooling subsystem is very aware that unless the process is scheduled reasonably, it will not be able to satisfy its specifications. It requires a trustworthy PM with a reasonable collection of paging and scheduling guarantees.

Associated with each policy object is some data that can be altered only by the "system administrator".[10] By convention, this data includes information about the trustworthiness and reliability of a particular PM, as well as its scheduling and paging policies and other services it may provide.[11] Any LNS (in particular the spooler) can access all of this information for the PM under which it is being scheduled, and may use the data to decide whether or not it will even try to provide its service. Here Hydra provides simply an informational mechanism, but one that permits subsystems to posit complex policies regarding the conditions under which they will attempt to provide service.

4.2.2 Negotiation with a PM

A more severe requirement for control protection arises in the execution of critical sections. Consider, for example, a data base subsystem. Only by calling a procedure supplied by this subsystem can users access the data base. Since multiple LNSs may be instantiated from this procedure, each in a different process, each attempting access, the procedure code contains a number of critical sections (which can, but need not, be implemented using semaphore objects). Once a process has entered a critical section, it is fairly important that it not be stopped until the critical section is exited, else other processes might be unnecessarily delayed. In the most extreme case, a PM may stop a process in the midst of a critical section and never restart it.

To solve this problem, the kernel provides a number of mechanisms that allow negotiation to take place between a process and its PM and that guarantee the results of the

[9] It may, of course, demand its own certified PM, but we are concerned with less inflexible users.

[10] This is enforced, of course, by appropriate rights to the policy object.

[11] There also exists a mechanism by which a PM can associate with each process under its aegis a set of capabilities. Any LNS executing under such a process can obtain any of these capabilities. Presumably these capabilities can be exercised to share information with the PM that will permit it to schedule or page the process more effectively.

negotiation. Before presenting our solution to the problem of critical sections, we will look at several other instances where negotiation mechanisms are useful.

We noted that processes are stopped whenever they are blocked on a semaphore and can be restarted by a PM after the process is subsequently unblocked. In fact, the PM may restart the process before it has been unblocked. The process will be notified of the PM's unusual action by getting an error return from the P operation on the semaphore. Ordinarily, in the case of mutual exclusion, a process passing a P may expect that it has sole access to some resource. By allowing a PM to arbitrarily restart the process, this need no longer be true. To facilitate debugging and recoverability, we might imagine that the PM could establish a non-kernel mechanism that could be used to notify a process in the case of abnormal restarts. The decision fixing the circumstances under which a process is to be restarted could be negotiated through that mechanism. However, this is inadequate in the case of erroneous or malicious PMs. Instead, the kernel guarantees the result of the negotiation by providing an error return on abnormal starts.

A process's processor mask provides another example of negotiation. In our earlier discussions of KMPS we noted that the mask is set in each process by the PM. In fact, the mask is actually determined by the top LNS executing under a process. It is necessary that the mask be able to change at least when the top LNS of a process changes. Each LNS is an incarnation of a distinct PROCEDURE and each may have distinctly different processor needs. Accordingly, each LNS has a field containing the mask of processors on which it will permit itself to run.

Each time an operation is performed which would necessitate a change in processor mask (e.g. changing LNSs), the process is stopped. The stop message sent to the policy object mailbox includes the allowable mask. The PM may then restart the process after changing the mask to the allowable one, or some subset of it. Suppose, however, the requested mask is not acceptable to the PM (e.g. too many other processes have requested the same special processor). Of course, the PM could keep the process suspended until the desired special processor became free. The mechanism provided, however, is a bit more robust. As in the previous example, if the process is restarted with an unacceptable mask, the operation which required the change in mask will fail with an error return. This again puts the kernel in the role of guaranteer of a negotiation.

CPS limits provide another example of a case in which negotiation is desirable, and the kernel mechanism supplied is the same as in the preceding situations. When a process attempts to exceed its CPS limit (for example, by a CPSLOAD), it is stopped. If the PM restarts it without increasing its CPS limit, the operation fails with an error return. In this case (as in the case of the processor mask) the process is given a chance to make do with less, i.e., with fewer pages allocated to it than it might find desirable.

Lastly, we return to the issue of critical sections. The kernel provides an operation, RUNTIME(time,pages), that solves this problem. A process invoking RUNTIME is guaranteed not to stop for "time" time-units (and runs at high priority during that period). In order to guarantee that it will not be stopped for exceeding its CPS limit (or in case its PM is over its page guarantee), "pages" page frames are allocated to it. Of course, this might cause havoc if the PM had no control over the execution of RUNTIMEs. It does. Unless "time" remains in the processes current time quantum and the CPS limit is greater than or equal to pages, the process is stopped. As with the previous example, if the PM restarts the process without changing these parameters, RUNTIME returns with an error.

5. Summary

We have discussed the reasons in favor of conscious separation of mechanism and policy and have examined some important places in Hydra where the separation is apparent. We have tried to show that, although not all policies are possible, Hydra's kernel mechanisms allow considerable flexibility at the user level in the choice of resource control policies. The particular set of mechanisms we have implemented is not the best possible; several known shortcomings have been identified and discussed. It is our hope that by consciously applying the policy/mechanism principle in the design of new kernel facilities, we will eventually acquire the insight to determine a successful, minimal set.

6. References

Bel66 Belady, L. A., "A Study of Replacement Algorithms for Virtual Storage Computers", **IBM Systems Journal 5,** 2 (1966).

BS71 Bernstein, A. and Sharp, J., "A Policy-Driven Scheduler for a Time-Sharing System", **Communications of the ACM 14,** 2 (Feb. 1971).

CJ75 Cohen, E. and Jefferson, D., "Protection in the Hydra Operating System", **Proceedings of the 5th Symposium on Operating System Principles,** Austin, Texas, Nov. 1975.

Den68 Denning, P. J., "The Working Set Model for Program Behavior", **Communications of the ACM 11,** 5 (May 1968).

Den70 Denning, P. J., "Virtual Memory", **Computing Surveys 2,** 3 (Sept. 1970).

JoW74 Jones, A. K., and Wulf, W. A., "Towards the Design of Secure Systems", **International Workshop on Protection in Operating Systems,** IRIA, 1974.

Lam73 Lampson, B., "A Note on the Confinement Problem", **Communications of the ACM 16,** 10 (October 1973).

Par72b Parnas, D., "On the Criteria to be Used in Decomposing Systems into Modules", **Communications of the ACM 15,** 12 (December 1972).

Wul74a Wulf, W., et al., "HYDRA: The Kernel of a Multiprocessor Operating System", **Communications of the ACM 17,** 6 (1974).

WLP75 Wulf, W., Levin, R., Pierson, C., "An Overview of the HYDRA Operating System Development", **Proceedings of the 5th Symposium on Operating System Principles,** Austin, Texas, Nov. 1975.