A Proposal for WM Interprocess Communication

Wm. A. Wulf

Computer Science Report No. TR-91-04
March 4, 1991

Department of Computer Science
University of Virginia
Charlottesville, Va. 22903

A Proposal for WM Interprocess Communication

Wm. A. Wulf

Computer Science Department
University of Virginia
Charlottesville, Va.

March 1991

1. Introduction

This report proposes and explores an interprocess synchronization and communication mechanism compatible with the proposed WM protection mechanism -- which is a "capability" based mechanism[Den66]. Despite the advantages of capabilities, early capability systems, including one by the author [Wul80], failed to have a major impact on practical security primarily because of the overhead of domain switching [Col88]. Thus, before committing to a capability mechanism for WM, we want to be sure that we don't make the same mistake again. Thus, this report explores the mechanism in just enough detail to answer whether or not it is practical.

Although our main goal is a mechanism for synchronization and communication among protection domains, if possible we would also like a *single* mechanism that can be used in the place of conventional interrupts, traps, "supervisor calls", entries to protected subsystems, and simple user-to-user interprocess communication.

The mechanism we will explore is much-simplified version of the Ada rendezvous [Ada83] in which the WM hardware performs all of the the necessary housekeeping, but software is still able to specify macroscopic policy decisions. At the outset, we stipulate that what we propose may be "too much to put into hardware". But it may not. Our purpose here is to define the mechanism is sufficient detail to be able to meaningfully ask and answer the question "is a hardware implementation practical and desirable".

2. The User's Model

We presume the WM protection mechanism based on a capability/domain model, with typed segments as discussed in [Wul91]. For the present, only a few properties of this model are relevant

- there is a type *task* (with hardware-defined operations to be discussed later),
- there is a type *domain* that contains capabilities -- only domains contain capabilities,
- a task has an associated domain that defines the segments accessible by it, and
- a task may name segments in it's domain, and cause them to be loaded/stored into/from a set of "capability parameter registers" in a manner to be discussed shortly. The ability to load/store capabilities from domains and capability registers is controlled by a set of access rights that won't be discussed here.

Since the model we'll explore is a simplification of the Ada 'rendezvous', first we'll briefly describe it and try to motivate its use.

The rendezvous model involves a subroutine like 'call' that appears synchronous to the caller, but is handled by an asynchronous task with something closer to a message style receive-reply mechanism. It is therefore similar semantically to many "remote procedure call" mechanisms, but is restricted to executing on a single CPU.

The model we shall use involves three basic operators: *ecall*, *accept*, and *reply*.

- *ecall* T(a1, ..., aN) is a synchronous subroutine call, except that T is a task. The parameters a1-aN are capabilities passed "by value/result"; in effect this means that the segments named by a1-aN are passed "by reference", but the called task's ability to modify then is tempered by the capabilities' rights.

- *accept*(f1, ..., fN) is executed by a called task to signify its readiness to handle an ecall. If an ecall is pending for this task, it is 'accepted' and processed. If no ecall is pending, the task is blocked ('suspended') until an ecall on it is executed.

- *reply*(r1, ..., rM) is executed by the called task to return results to its caller and to enable the caller to resume execution.

A common server/client scenario using this sort of mechanism is shown graphically below. T1 is a task that wishes to obtain a service from a task T2. T1 performs an *ecall* ("entry call") on T2, passing actual parameters a1-aN; it blocks until T2 provides the service, and then proceeds normally. From T1's perspective, the ecall appears to have been a synchronous procedure call.

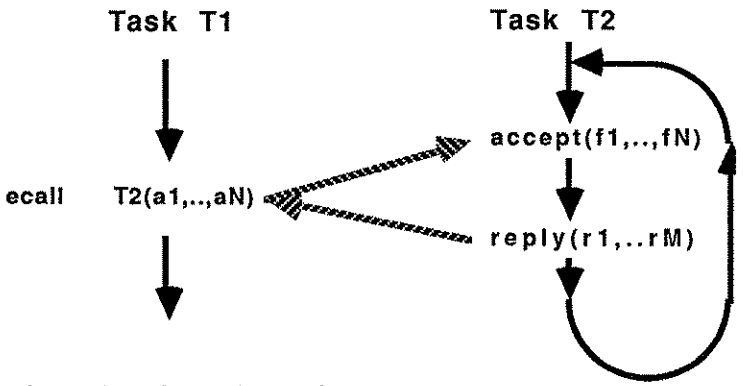


Figure 1: A Typical Client/Server Communication Paradigm

From T2's perspective, things look rather different. T2 performs an *accept* to assert its readiness to handle a call; if no caller is waiting, it blocks. If a caller is waiting, however, it proceeds using formal parameters f1-fN initialized from the caller's actuals, and eventually sends a *reply*, including results r1-rM. T2 does not block after the reply, but proceeds to other computations and eventually another accept.

The utility of rendezvous-style interprocess communication has been widely discussed and debated in the Ada literature. We don't wish to enter that debate; we simply note that this simplification of the Ada model is *adequate* for our purpose; that is:

- it is adequate as a model for "supervisor call"; indeed, since the call is synchronous from the caller's perspective, the fact that it may be serviced asynchronously is irrelevant, and it is therefore semantically indistinguishable from current SVC's.

- it is adequate as a model for traps; a trap is a synchronous invocation of a handler, and hence, since the ecall is synchronous as far as the caller is concerned, it is semantically indistinguishable from current traps.

- it is adequate as a model for interrupts. A clock or io device can be thought of as an asynchronous task that occasionally communicates with its handler task by performing an ecall. A handler, in turn, is much like the cyclic server in the earlier figure. Normally one expects it to be blocked on its accept, and that an ecall -- that is, an interrupt -- will wake it up. It then "masks" further interrupts until ready to service subsequent ones by the simple expedient of not executing an accept. Finally it cycles back and executes another accept to "re-enable" itself for the next interrupt.

- it is a *usable* model for user-to-user task communication, and can be used to implement message-passing or other synchronization/communication paradigms if they are preferable for some reason. Not all experts would agree that it is a *adequate* or *convenient* for all styles of interprocess interaction, but we do not need that for this exploration.

The notion of a single mechanism that serves all these functions, and does so in a manner that seems reasonably natural for all of them, is certainly aesthetically pleasing. Whether or not it is practical, however, hinges on the complexity and cost of implementation -- particularly since we expect the implementation to be in hardware. That is the subject of the next section.

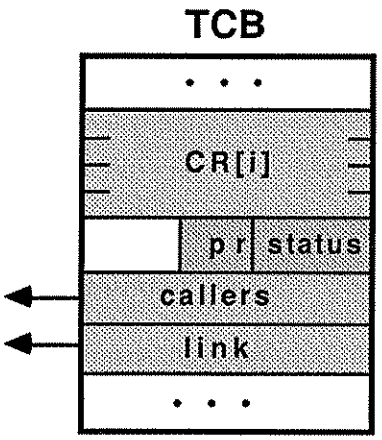
3. An Implementation Model

In this section we suggest an implementation model in terms of which we can consider the question of practicality of the mechanism. To do this we proceed in several steps

- positing an implementation of the three rendezvous operations,
- positing a scheduling mechanism, and
- positing a context-switching mechanism.

The three are related, but independent, decisions so we'll discuss them separately and merge the results later. But first, we need some definitions and support.

Task State: A task (in addition to the rest of its state) has several properties:



- a priority, *pr*
- a (small) set of *capability registers*, CR[i], to hold parameters during an ecall,
- a (possibly null) queue of callers, the head of this queue is sometimes referred to as the (current) caller.
- a link (non null only when in a queue),
- a 'status' with at least the following values

APC	asleep, pending call
APR	asleep, pending reply
RDY	asleep, but ready
RUN	running

Capability Registers: We are going to use capability registers primarily to pass parameters between tasks; There are only four operations on CR's

- loadCR	i, j	load CR[i] from the j^{th} capability slot in this task's domain
- storeCR	i, j	store CR[i] into the j^{th} capability slot in this task's domain
- loadCCR	i, j	load CR[i] from the caller's CR[j]
- storeCCR	i, j	store CR[i] into the caller's CR[j]

The right to perform these operations on particular capabilities will, of course, be subject to possessing appropriate rights. We ignore that complication here.

When an ecall is done, no parameters are really passed. Instead, prior to the ecall the caller is assumed to have placed the relevant parameters in its own capability registers. The called task then accesses these parameters directly from the caller's CR's using the LoadCCR. Similarly, results are returned by having the called task store capabilities in the callers CR's using StoreCCR. Thus, the communication operations are not concerned with parameter passing, but only with control.

3.1 The Basic Operations

The details of the implementation of each of the three basic operations depends on the scheduling mechanism, which is the subject of the next section. However, the basic operation of each of these operations is shown below; each is indivisible.

ecall t
enqueue self on t.callers
if t.status = APC then set it to RDY
set self.status to APR
select and swap to a new task

accept
if self.callers is null
    set self.status to APC
    select and swap to a new task

reply
dequeue a task from self.callers and set it's status to RDY
perhaps select and swap to a new task

These implement the informal model described in Section 2. However, note the somewhat odd definition of *accept*. If used in the manner described in Figure 1, namely a loop of alternating *accept* and *reply's*, it behaves as desired. However, if two accepts are executed in sequence without an intervening reply (and hence dequeuing of the caller) the same caller capability will be used for both. This is not a problem, particularly, but one has to be aware of it. We defined it this way because it significantly simplifies the mechanism.

Also note that a task can be in only one of a few states -- one corresponding to each of the status field values.

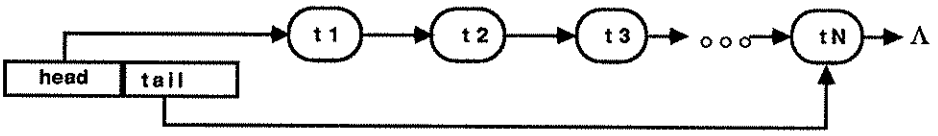
- it can be running, in which case its status is RUN, and it is not necessarily enqueued anywhere,
- it can be runnable, but asleep because other tasks have control of the available CPUs. It's status is RDY, and presumably it is enqueued somehow by the scheduling mechanism,
- it can be asleep waiting for a server to reply, in which case its status is APR and it is enqueued on the *callers* field of the server, or
- it can be asleep because it executed an accept, and so is waiting for a call; it's status is APC, and it is not necessarily enqueued anywhere.

The transitions between these states seems relatively simple.

3.2 Scheduling

In the following we will explore the implementation of two scheduling policies, and code the three basic operations more completely. First, however, it's useful to add some more support:

Queue Management: Here and later we will want queues of TCBs; queue headers (such as *callers*) will consist of two fields: *head* and *tail*. These will point to the first and last entries in the queue respectively[1].



Note that the 'pointers' in *callers* and *link* are capabilities for the tasks they point to; this will be important later. To shorten later explanations, we'll define two operations for adding a task to the end of a queue and for removing the first element of one[2]:

```
enq (q: queue, t: tcb)        -- append t to the end of q
    t.link := null;
    if q.head = null
        then    q.head := q.tail := t;
        else    q.tail.link := t; q.tail := t

deq (q: queue): tcb           -- dequeue and return the 1st TCB in q
    t mp := q.head;
    if tmp = null then error;
    q.head := tmp.link;
    return tmp;
```

There are a number of options for implementing the scheduler; we'll examine just two here -- they may not be the best, but they are a feasibility demonstration. In particular, we'll examine a simple 'round robin' scheme and a priority based one as two extremes of what might be reasonable.

A Round Robin Scheduler:

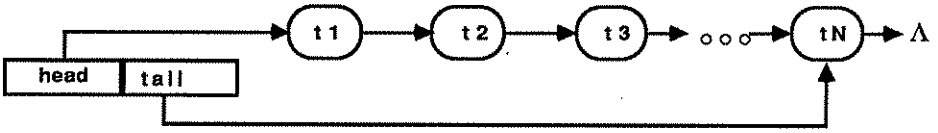
Perhaps the simplest scheduler is the familiar "round robin" scheme. As tasks become ready they are placed at the end of a single queue (called READY here); a new task is selected to run from the front of the queue.

---

[1] Obviously other linked structures could have been used, and some others have advantages. However, this is merely a feasibility exploration, and this particular representation has the charm of using relatively few memory references.

[2] These and the subsequent operation definitions are indivisible.

READY:



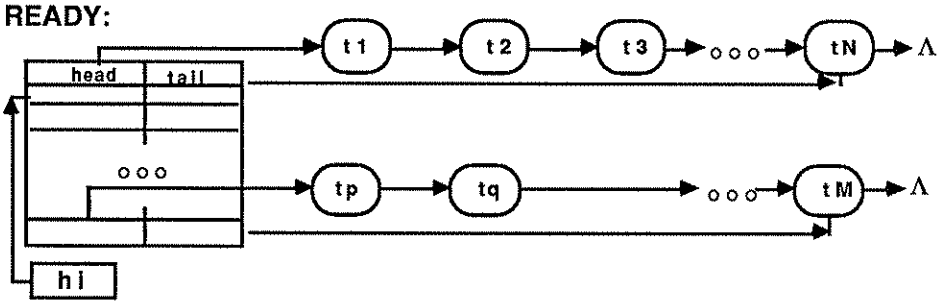
Below is recoding of them that implements round-robin scheduling. In this coding: (a) "swapto" is the hardware context swap operation, and (b) the currently executing task is not enqueued on any list.

```
ecall t     enq(t.callers, self);
            if t.status = APC then {t.status := RDY;  enq(READY, t)}
            self.status := APR
            swapto(deq(READY));

accept      if self.callers = null then {self.status := APC; swapto(deq(READY))}

reply       self.callers.status := RDY; enq(READY, deq(self.callers));
```

A Priority Scheduler:

A priority scheduler can be implemented easily with an array of queues (one queue per priority level) plus a pointer to the highest priority, non-empty queue.

READY:




```
ecall t     enq(t.callers, self);
            if t.status = APC then {t.status := RDY;  enq(READY[t.pr], t)}
            self.status := APR
            if t.pr > self.pr then
                hi := t.pr;
            else
                while READY[hi].head = null do hi := hi-1;
            swapto(deq(READY[hi]));
```

```
accept    if self.callers = null then {
              self.status := APC;
              while READY[hi].head = null do hi := hi-1;
              swapto(deq(READY[hi]))}

reply     tmp := deq(self.callers);
          tmp.status := RDY;
          if tmp.pr ≤ self.pr then
              enq(READY[tmp.pr]), tmp)
          else{
              enq(READY[self.pr], self);
              hi := t.pr;
              swapto(tmp)}
```

As can be seen, these versions of the operations are somewhat more complicated than the previous round-robin versions -- they require accessing the priority field, indexing into the array of queue headers, and there is a loop to set 'hi' properly when the current task goes to sleep -- although with a modest number of priority levels, one would probably not implement this as a loop.

It's not clear how to assess the complexity of these operations, but below we have attempted to provide a rough count of the number of memory references, RISC-like operations, and context swaps for the longest and shortest paths through them. In making these counts, I assumed that the READY headers were in registers, a small amount of information from the running task is kept in registers, and that the resetting of 'hi' could be done in unit time.

	short path			long path		
operation	mem	ops	swp	mem	op	swp
RR call	3	8	1	6	14	1
RR accept	1	1	0	3	4	1
RR reply	3	9	0	3	9	0
Pr call	7	10	1	10	15	1
Pr accept	1	1	0	3	5	1
Pr reply	4	11	0	5	11	0

These counts are rough, may be slightly wrong, and certainly would be different for a different set of assumptions. However, their general magnitude suggests several things:

- their complexity is larger than one could tolerate in a pure RISC-style implementation (but see section 4),
- their complexity is similar to that of some extant CISC instructions, eg, the VAX CALL instruction.
- the "short paths" through *accept* and *reply* are very nice. These are the cases where one wants minimal overhead -- there is work to do and we want to get on about doing it.
- The cost of *call* is higher than that of the other operations. In addition, since a context swap will always be done for calls, the real question of its cost hinges on the topic of the next section -- context switching.

My own opinion is that, if one buys into this sort of scheme at all, then the added complexity of a priority scheme is justified by the flexibility it provides.

3.2 Context Switching

WM has a lot of processor state! It was designed that way to make individual programs execute rapidly -- but it does mean that a context swap may be costly. Since the present proposal assumes just such a swap for all interrupts, traps, and SVCs there is reason to ensure that the cost of a swap is reasonable.

The WM state potentially consists of several logical entities, as listed below; rough estimates are given for the sizes of these entities for a $WM_{32,64,0}$ -- a member of the WM family roughly corresponding to today's workstation machines.

State Entity	Rough size (in bytes)	
- the IEU registers	116	29 registers, 4 bytes wide
- the IEU instruction queue	16	4 deep, 4 bytes wide
- the IEU r0 & r1 FIFOs	80	4 queues, each 4 deep and 4 wide plus 16 bytes of status and control
- the FEU registers	232	29 registers, 8 bytes wide
- the FEU instruction queue	16	4 deep, 4 bytes wide
- the FEU r0 & f1 FIFOs	144	4 queues, each 4 deep and 4 wide plus 16 bytes of status and control
- misc (PC, etc.)	??	
total	604+	

Obviously, we're creeping up on a kilobyte without even considering a vector execution unit. From other considerations, we will want a memory system that is capable of handling at least two 4-byte requests per instruction cycle[3]. Thus, a kilobyte of context saved and another restored translates to about 256 instruction times. While this may not be terrible for the general case of task switching, it is excessive for the subroutine-like uses that we have in mind.

While logically all of this state must be saved and restored on each swap, physically, we may be able to do better. Many programs don't use floating point, for example; others will use only a subset of FIFOs, and some will not stream. Some programs, in fact, will use only a subset of the registers in one of the execution units. Tasks such as interrupt handlers, in particular, are often constructed to have very little state.

Therefore, suppose we define a context-swap control word, CCW, as part of the task state; state would be saved/restored under control of this word. For example, it might have bits such as:

[3] In order to keep up with the streamed dot product loop, for example.

ir0	if one, save/restore integer registers r2-r7
ir8	if one, save/restore integer registers r8-r15
ir16	if one, save/restore integer registers r16-r23
ir24	if one, save/restore integer registers r24-r30
if0i	if one, save/restore integer input fifo 0
if0o	if one, save/restore integer output fifo 0
if1i	if one, save/restore integer input fifo 1
if1o	if one, save/restore integer output fifo 1

and similar bits for the floating point and vector state.

There are two obvious interpretations that one might place on these bits:

- they are 'touched' bits, dynamically set by the hardware, or
- they are 'valid' bits, set by the compiler and/or operating system.

In the first case, the bit would be set the first time that a task touched a resource represented by it. Forever after, the resource would be saved and restored. In the second case, the systems software would determine *a priori* which resources will have to be saved and restored; in this case, the hardware might even make it illegal to access the resources not flagged in the CCW.

Unfortunately, both of these schemes force saving/restoring forever -- even if a resource is used once and never touched again by the program. In general, the hardware cannot tell whether the value in a resource will be used at some time in the future, so help from the program itself is necessary. Suppose one uses the 'touched' interpretation, and adds an instruction

```
safe <mask>
```

the interpretation is that the mask is anded with the touched bits -- thus allowing the program to turn off some of the bits and, in effect, assert that certain resources are not in use at the time that the 'safe' instruction is executed.. A typical interrupt handler, for example, might take the form

```
while true do
        safe 0;          -- assert no resources are in use
        accept;          -- sleep until interrupt happens
        <whatever>       -- process it
        od;
```

In such a case, the time to restore the handler will be very small. Unfortunately, if a task with a lot of context is running at the time of an interrupt it may still take a long time to save that task's state before the handler can be initialized. Or will it?

Under the second interpretation of the CCW, and adopting the attitude that it is illegal for a task to access resources not flagged in the CCW, a new task could proceed as soon as a subset of the old task's resources had been saved -- specifically those that the new task might touch. In such a regime, the context swap would alternate between saving state of the old task and restoring that of the new. As soon as enough of the new task's state were present, it would be allowed to run while saving of the old task's state would continue in the background.

In this scheme, an interrupt handler that requires very little context could begin to run almost immediately, even if the interrupted task has a great deal of context. Unfortunately, note that we have used both interpretations, and we can't place both of them on the same bits simultaneously. The resolution of this is obvious, of course -- use two CCW's, one with each interpretation:

CCWR defines the set of resources required for the task to run; this is set by software, and defines those resources of the prior task that must be saved before this task can begin execution. It is illegal for a task to access a resource not named in its CCWR.

CCWT defines the set of resources that were 'touched' by the task, and hence need to be restored before it can begin execution. It is CCWT that is changed by the 'safe' instruction. Necessarily, CCWT $\subseteq$ CCWR.

With this scheme the time to perform a context swap is dependant solely on the properties of the task being swapped *to*. Tasks such as interrupt handlers that 'safe' themselves appropriately can begin quickly. The only real complexity added seems to be for checking the legality of resource usage and supporting context saving as a background activity. Neither needs to add to the length of the "critical path" timing for normal instructions; legality checking can be done in parallel with execution and abort an illegal instruction after the fact, and context saving can be an asynchronous activity.

Note, however, that since we are probably unwilling to support the complexity of saving the context of more than one task simultaneously, we will probably inhibit a new context swap while another is completing. Thus the maximum rate of processing interrupts will be limited to something like one per 256 instruction times. I doubt this will be a problem; with a 50mH clock, this limit is roughly one interrupt every 5 microseconds.

4. The Implementation in the WM Context

In most machines, I would consider the mechanisms considered here too complex for hardware implementation. For these machines the only reasonable implementation technique is microcode, with the resulting negative impact on the execution speed of the more common arithmetic instructions.

In the case of WM, however, the case is not so clear. As discussed in [Wul90], WM is composed of several execution units that are kept in loose synchrony through a set of FIFO queues; an instruction fetch mechanism dispatches instructions to these units after trivial decoding. The figure below shows a macroscopic view of WM's decomposition.

There is no reason that the implementation technique for the various execution units need be the same. In particular the Integer and Floating Execution Units (IEU & FEU respectively) are designed for simple hardwired control, and can use a RISC-like implementation. Execution of control instructions is similarly designed to be extremely simple. The "special" instructions, on the other hand, are relatively rare and there is no reason not to microcode this portion of the machine if it results in an overall performance gain.

In short then, although the mechanism proposed here certainly increases the size (gate count) of the machine, that need not slow down the machine's execution of "ordinary" programs.

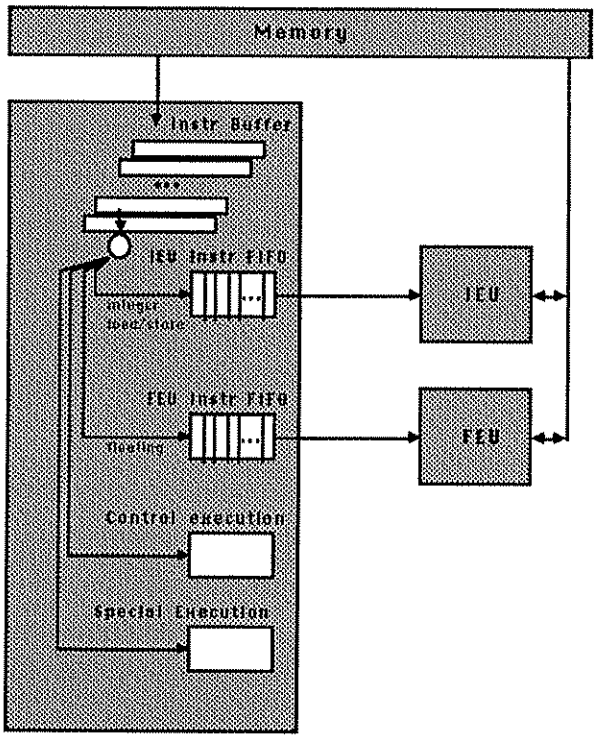


Figure 2: Decomposition of WM Execution Units

5. Extensions

One can imagine a number of useful extensions of the basic three operations defined so far; we'll consider three of them here. In each case the issue is adding functionality that would be difficult or impossible to obtain otherwise.

5.1 The "conditional accept" Operation

There are cases where a task may wish to test whether there is a caller waiting for it, but not block if there isn't. In terms of the WM conditional jump semantics, this merely requires the insertion of a true/false bit into the integer condition code FIFO.

```
Caccept  if self.callers = null
             then insert false into cc-fifo
             else insert true in cc-fifo
```

5.2 The 'pass' Operation

It is relatively common to want a single logical server for a service, but to provide the service with a larger set of tasks. The *pass* operation is intended to facilitate this.

Suppose T1 performs an ecall on T2, and T2 performs an accept; we now say T1 is the "caller" of T2. If T2 now performs a *pass* to T3, T1 will become the caller of T3 and will be removed as the caller of T2 -- that is, T2 has "passed the call" to T3. T2 is then the apparent single "point of contact" for a particular service, but to passes out the work of actually providing the service to a set of other tasks. Thus, T2 might look something like:

```
task T2 is
servers: array[0..N-1] of task;
i: int;
...
i :=0;
while true do
        accept;                 -- accept an ecall
        epass servers[i];       -- pass the call on to a server
        i := (i+1) mod N;       -- select the next server
        od;
end task;
```

The implementation of *pass* is straightforward (here we show the implementation for a priority scheduler):

```
pass t      enq(t.callers, deq(self.callers));
            if t.status = APC then {
                t.status := RDY;
                if t.pr ≤ self.pr then
                    enq(READY[t.pr]), t)
                else{
                    enq(READY[self.pr], self);
                    hi := t.pr;
                    swapto(t)} }
```

5.3 Storing Tasks in Data Structures

The four operations described so far imply FIFO handling of calls; there is no way for a server to suspend processing one caller, service a second, and then resume handling the first. Such a mechanism would add a great deal of flexibility, and is relatively inexpensive. While several options exist, consider two operations:

GetCaller i CR[i] := deq(self.callers);

SetCaller i enqf(self.callers, CR[i]);

(where *enqf* is enqueue-at-front, which means that the caller of the task executing SetCaller immediately becomes the one whose capability was in CR[i]).

Of course, once a task has moved a capability for the calling task into it's capability registers, it can move it to its domain (or a domain reachable from its domain); thus an arbitrary data structure of capabilities for tasks can be constructed[4].

[4] This is one of a number of cases where we would like to ensure that there is at most one extant capability for an object; we will deal with the issue in a general fashion, so it is not discussed here.

6. Summary

Based on the examination of the design alternatives presented here, I find no *timing* reason to reject the proposed mechanism (but I wish I could make a stronger statement).

- the basic operations, including scheduling, are implementable with standard microcode techniques; the amount of microcode is not excessive.
- the existence of a microcode interpreter for these operations need not slow down the execution of the more common instructions.
- the cost of a full context swap for interrupts can be made small enough to be acceptable.

The cost of the mechanism is more than just the impact on execution speed, of course. In particular, we need to examine the real cost of a mixed hardwaired-microcode strategy.

Finally, whether the mechanism is justified depends on whether it affords sufficient benefits -- and that can only be evaluated in a larger context than is considered here. I hope, however, that the interaction of this mechanism with the protection mechanism will have significant implications for the overall performance of systems.

In short -- this level of analysis was not adequate to reject the proposed scheme, but it certainly was not adequate to adopt it either.

Bibliography

[Ada83] U.S. Department of Defense, "Reference Manual for the Ada Programming Language", ANSI/MIL-std 1815-a, Jan 1983.

[Col88] Colwell, R.P., Gehringer, E.F., and Jensen, E.D., "Performance Effects of Architectural Complexity in the Intel 432", ACM Transactions on computer Systems, v6 n3, August 1988, pp 296-339.

[Den66] Dennis, J., and van Horn, E., "Programming Semantics for Multiprogrammed Computations", CACM v9 n3, March 1966, pp 143-55.

[IBM90] IBM Corp., "IBM RISC System/6000 POWERstation and POWERserver Hardware Technical Reference - General Information", SA 23-2643-00 First Edition, 1990.

[Lev75] Levin, R., et al., "Policy/Mechanism Separation in Hydra," Fifth Symposium on Operating Systems Principles, November 1975.

[Nee77] Needham, R., and Walker, R., "The Canbridge CAP Computer and Its Protection System", Poceedings of thew Sixth Symposium on Operating System Principles, Nov. 1977, pp 1-10.

[SUN87] Sun Microsystems, Inc., "The SPARC Architecture Manual, Part No: 800-1399-08, 1987.

Wul74] Wulf, W. , et al, "Hydra - The Kernel of a Multiprocessor Operating System," CACM, June 1974.

[Wul80] Wulf, W. , et. al., "Hydra/C.mmp: An Experimental Computer System". McGraw-Hill Publishing Co., 1980.

[Wul90] Wulf, W., "The WM Computer Architecture -- Principles of Operation", U. Virginia Computer Science TR-90-2, Jan 1990.

[Wul91] Wulf, W., and Jones, A., "WM Protection: The Base Mechanism", U. Virginia Technical Report, March 1991.