1980

Low Contention Semaphores and Ready Lists

Peter J. Denning

T. Don Dennis

Jeffrey Brumfield

Report Number:
80-332

LOW CONTENTION SEMAPHORES AND READY LISTS

Peter J. Denning
T. Don Dennis
Jeffrey Brumfield

Computer Sciences Department
Purdue University
W. Lafayette, IN 47907

CSD-TR-332

February 1980
Revised June 1980

Abstract. A method for reducing semaphore and ready list contention in multiprocessor operating systems is described. Its correctness is established. Its performance is compared with conventional implementations. A method of implementing the ready list with a ring network is proposed and evaluated.

This work was supported in part by NSF Grant MCS78-01729 at Purdue University.

June 18, 1980

THE PROBLEM

Modern operating systems implement semaphores for synchronizing multiple concurrent processes. Unless the primitive operations for starting, stopping, and scheduling processes and for manipulating semaphores are well supported by the hardware, context switching and interprocess signaling can be major overheads [1]. To avoid these overheads, many operating systems use fast, but unreliable ad hoc methods for synchronizing processes. The number of processors (CPUs) that can be kept busy can be limited by contention at the ready list or at semaphores.

We will present a process manager that overcomes these problems. The hardware architecture is tightly coupled with the software and data structure. Five indivisible operations -- start, stop, and schedule a process, plus wait and signal on semaphores -- are implemented as microprograms in each processor's instruction set.* This significantly reduces the holding times of locks on the semaphores and on the ready list. Tagged memory can be used to ensure the integrity of semaphores and data structures but is not essential. We will compare standard implementations of the wait and signal operations with our proposals. We will extend the design to include I/O control via private semaphores that honor higher priorities among device driver processes. We will also show that ready list contention

*Wait and signal were microprogrammed on the VENUS machine, an experimental uniprocessor [9]. The GEC 4080, a commercial machine, comes closest to meeting the design objectives discussed here [8].

and the "multiprocessor priority problem" can be virtually eliminated by implementing the ready list not as a passive data structure but as a circulating ring of process indices.

OVERVIEW OF A PROCESS MANAGER

The process manager is the portion of the operating system that implements processes and semaphores. It abstracts away from the details of scheduling and switching the several processors among the processes. It replaces busy-waiting on locks with process suspension on semaphores.

Data Structure

The internal data structure of the process manager comprises the process list, the ready list, and the semaphore list. Figure 1 illustrates.

The process list (PL) is an array of process control blocks (PCBs) identified by process indices. Each PCB contains a stateword field and a link field. The stateword field contains a copy of the values of all processor registers defining the environment of the process -- e.g., program counter, general registers, page table base, stack pointer, and interrupt masks. It occupies s memory words. The link field contains the index of the next process on the same queue; queues are implemented as linked lists.

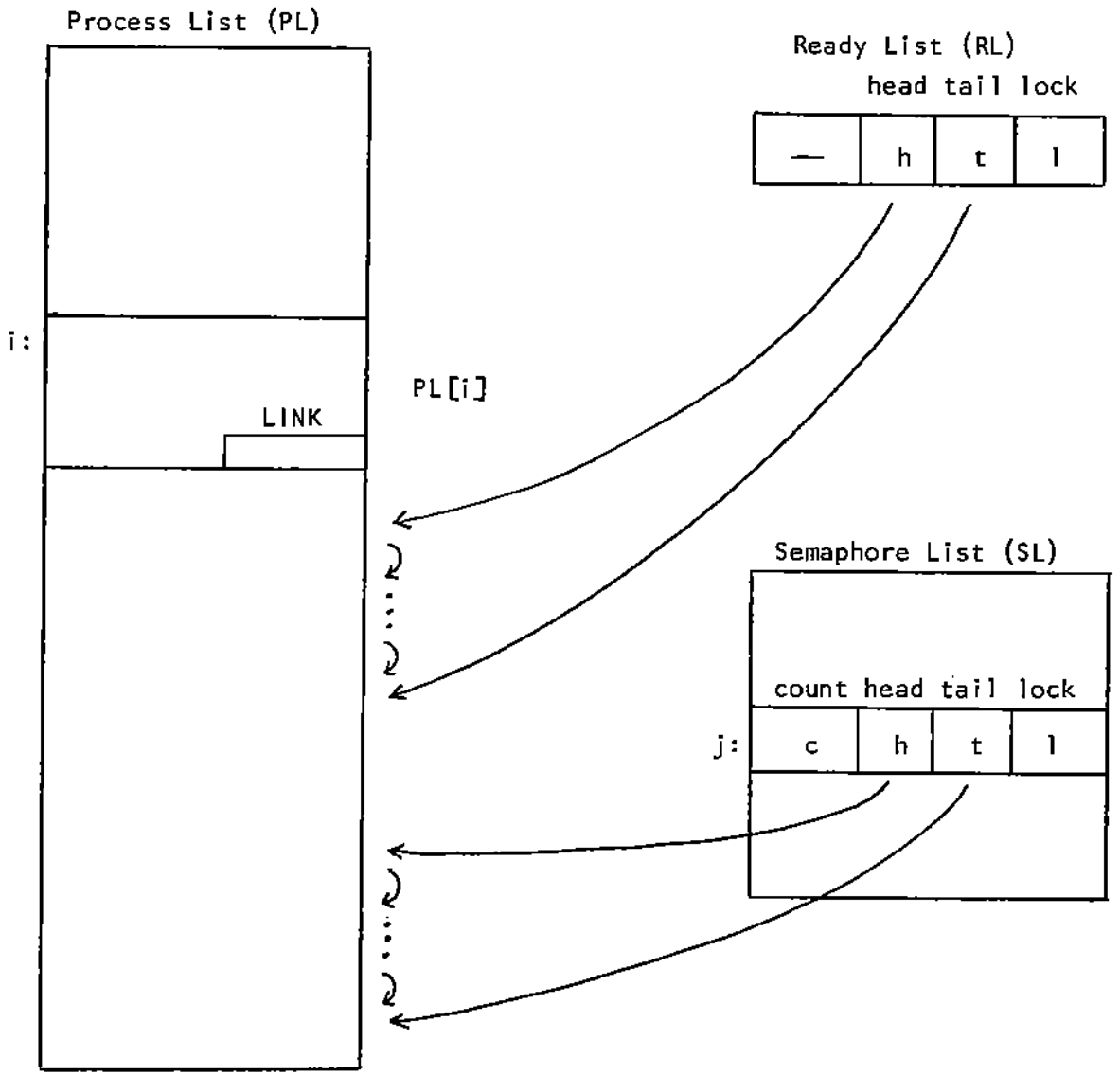

FIGURE 1: Process manager data structures.

Expressed as Pascal declarations:

```
type processindex: 1..N

type PCB = record
             stateword: ... ;
             link: processindex
           end

var PL: array[processindex] of PCB
```

Note that the allowable process indices are 1,...,N. For convenience, we will use the notation LINK[i] for the link field in PL[i], rather than the formally correct notation PL[i].link.

The ready list (RL) is a queue containing indices of all processes enabled to run on a processor. It emanates from a descriptor containing a (head, tail) linked-list specifier and a lock bit. The queue itself is the chain of processes found by tracing through successive link fields starting from the head process and terminating at the tail process. The link field of the tail process is set to 0. The lock bit is used to restrict access to the ready list to at most one processor at a time. Expressed as a Pascal declaration:

```
var RL: record
          head, tail: processindex;
          lock: Boolean
        end
```

It is possible to represent several ready lists -- e.g., one for each priority level -- by an array of such records.

The semaphore list (SL) is an array of semaphore descriptors, each containing a count field registering the excess of sent over received signals, a (head, tail) linked-list specifier defining a queue of processes waiting for signals, and a lock bit to prevent simultaneous access by two or more processors. A Pascal definition:

```
type semindex: 1..M

var SL: array[semindex] of
          record
            count: integer;
            head, tail: processindex;
            lock: Boolean
          end
```

Following convention, the initial value of a semaphore's count must be nonnegative; a nonnegative count indicates an empty queue, whereas a negative count indicates a queue whose length is the magnitude of the count.

Locks

The foregoing definitions show a lock bit in the ready list descriptor and in each semaphore descriptor. These bits must be set while any processor is using the associated data structure. They are set using a LOCK instruction, reset using UNLK (unlock). Let Mem[x].lock denote the low-order bit of the memory word at address x. The microprogram for LOCK x follows this schema:

```
LOCK x:
        ----------------------------
        b := Mem[x].lock
        IF b = 1 THEN "retry at LOCK"
        "disable interrupts"
        Mem[x].lock := 1
        ----------------------------
```

The microprogram for UNLK follows the schema:

```
        ----------------------------
UNLK x: Mem[x].lock := 0
        "enable interrupts"
        ----------------------------
```

The LOCK-UNLK pair is intended to enclose a critical section of instructions. The interrupts of the processor inside the critical section must be disabled to guarantee the indivisibility of the critical operation. The LOCK instruction takes (at least) two memory reference times (at least one read and a write) and the UNLK one.*

The LOCK instruction itself must also be indivisible: once started, the addressed lock bit must be fetched, modified, and returned to memory; no other processor may access the addressed memory location until the instruction is complete. This requirement is easily enforced by the usual protocol at the processor-memory interface. Having placed an address in its memory address register, the processor raises an address-request

---

*As specified, LOCK is not the same as "test-and-set" on the IBM 370. In fact, LOCK requires three instructions, and UNLK two, on the IBM 370.

line, A, and waits. When the addressed memory bank becomes idle, the memory arbiter selects a waiting processor (one with A=1) and pulses the proceed line to that processor. As soon as it receives the proceed pulse, the processor performs the memory access (read and/or write) on the addressed location. At the completion of the memory access, the processor lowers the address-request line (sets A=0), which informs the memory arbiter that the addressed bank is again idle. In the case of the LOCK instruction, both a read and a write operation are performed while A=1. The protocol requires a processor to release the A-line before loading a new address into its memory address register.

The dashed lines in the microprogram for LOCK and UNLK represent points at which the processor must relinquish its control over the memory bank. It must set A=0 in order to cross a dashed line.

If the LOCK instruction is begun when the lock is set, the processor will perform the "retry at LOCK" action. This means that the processor must release exclusive access to the addressed memory location and restart the LOCK operation. It also means that a processor waiting for a lock engages in busy waiting.

Busy waiting increases lock contention: by stealing memory cycles from the processor inside the critical section, the waiting processors prolong the holding time of the lock. This degradation can be mitigated by changing the retry action to

"retry after delay T",

where T should be about half the time a processor will remain in the critical section. Except in special cases, however, it is impossible to know a priori what T will be.

Starting, Stopping, and Scheduling Processes

We suppose that there are three (uninterruptible) operations for manipulating the process list and the ready list. These operations refer to a processor register, 'self', that contains the process index of the process currently running on that processor. A running process is not on the ready list. The instructions are:

SAVESW	--	Used to stop a process from running. The processor registers are copied into PL[self].stateword.
LOADSW	--	Used to start a process running. Removes the head process from RL and sets 'self' to this value. Loads the processor registers from PL[self].stateword and proceeds.
READY(i)	--	Used to schedule a process. Inserts process index i at the tail of the RL.

The SAVESW is equivalent to an instruction sequence for saving all general registers, all control registers, and the program status word (PSW) on the IBM 370. The LOADSW consists of a ready-list manipulation followed by the equivalent of the

instruction sequence that loads all registers and the PSW on the IBM 370. A program for LOADSW is:

```
LOADSW:  with RL do
            LOCK lock
            self := head
            head := LINK[head]
            LINK[self] := 0
            "copy PL[self].stateword into registers"
            UNLK lock
         end
```

Note that LINK[self] is set to 0 to indicate that no process follows self on any queue. A program for READY is:

```
READY(i):  with RL do
              LOCK lock
              if head = 0
                 then head := i
                 else LINK[tail] := i
                 fi
              tail := i
              UNLK lock
           end
```

Wait and Signal Operations

The WAIT operation is used to receive a signal from a semaphore; the calling process will be delayed if the count is zero or less at the time of the attempted reception. The SIGNAL operation is used to transmit a signal through a semaphore; the head waiting process is released if the count is less than zero at the time of the attempted transmission. Both operations must be indivisible in the sense that, while a WAIT or SIGNAL is in progress on a given semaphore, no other WAIT or SIGNAL on that

same semaphore may be initiated. This implies that both operations must be uninterruptible. A program for the wait operation is:

```
WAIT: procedure(j: semindex)
      with SL[j] do
         LOCK lock
         count := count - 1
         if count < 0 then
            SAVESW
            if head = 0
               then head := self
               else LINK[tail] := self
               fi
            tail := self
            LOADSW
            fi
         UNLK lock
      end
```

A program for the signal operation is:

```
SIGNAL: procedure(j: semindex)
        with SL[j] do
           LOCK lock
           count := count + 1
           if count ≤ 0 then
              i := head
              head := LINK[head]
              LINK[i] := 0
              READY(i)
              fi
           UNLK lock
        end
```

Note that the ready list may be locked for a subinterval of the semaphore lock. This will occur if count $\leq$ 0 in WAIT (LOADSW will be executed) and if count < 0 in SIGNAL (READY will be executed).

Correctness

The correctness of the above implementation derives from four facts. First, the instruction SAVESW is uninterruptible once begun on a given processor because it is a microprogram that does not inspect the interrupt indicators. The programs LOADSW and READY are uninterruptible once begun because they are enclosed in LOCK-UNLK pairs.

Second, ready list manipulations are mutually excluded because the ready list is locked by LOADSW and READY, the only two programs that operate on it. While these locks are set, interrupts are disabled by the LOCK instruction.

Third, the critical sections of the WAIT and SIGNAL programs are enclosed by a LOCK-UNLK pair. This ensures their mutual exclusion for any given semaphore and prevents the interruption of the processor inside the critical section.

Fourth, each process index is either in some one 'self' register, on the ready list, or on some one semaphore list. The moving of a process index between pairs of these places cannot be interfered with for the reasons summarized in Table 1.

Deadlock is not possible because the holding of RL locks is strictly nested inside the holdings of semaphore locks. Because its interrupts are disabled while it is in a locked region, a processor cannot be diverted to another program containing an attempted lock on another semaphore or on the ready list.

Transition	Operation	Reasons
RL —> self	LOADSW	RL is locked by processor executing the LOADSW, and 'self' is private to that processor.
self —> SL[j]	WAIT	SL[j] is locked by the processor performing the wait, and 'self' is private to that processor.
SL[j] —> RL	SIGNAL	Both SL[j] and RL are locked by the processor performing the signal.

TABLE 1: Correctness of process index transitions.

Performance

To estimate space and time requirements, we hand coded the five process management operations for the IBM 370 and VAX-11/780 instruction sets. We assumed that these instruction sequences would be put in line, as macros, to avoid the additional overhead of procedure calls [7]. The results are summarized in Table 2.

In the IBM 370, the stateword comprises 16 general registers, 16 control registers, 4 floating-point registers, and the program status word (PSW). LOADSW and SAVESW each include s = 37 operand references for all these registers; they also include instructions for disabling and enabling interrupts. In the VAX, the stateword comprises 16 general registers and 2 control registers. LOADSW and SAVESW each include s = 18 operand references for these registers.

	IBM 370					VAX 11/780				
	SAVESW	LOADSW	PUT	WAIT	SIGNAL	SAVESW	LOADSW	PUT	WAIT	SIGNAL
Instruction Storage (bytes)	42	86	60	254	148	1	36	26	74	57
Instruction Fetches										
Short Path	11	22	14	19	19	1	9	6	7	7
Long Path	11	22	14	62	38	1	9	6	20	14
Operand References										
Short Path	38	47	8	17	17	18	34	11	6	6
Long Path	38	47	9	101	29	18	34	11	65	39

TABLE 2: Space and time requirements of operations.

The figures in Table 2 do not include the delays for busy waiting on locks or for memory cycles lost while other processors cycle at LOCK operations. The "short path" cases of instruction fetching and operand referencing arise when the semaphore counts are high enough to avoid queueing. The "long path" cases arise when queues must be manipulated and contexts switched. Compared to the IBM 370, the VAX implementation requires roughly 1/3 the space and 1/2 the execution time, or roughly 1/6 the space-time.

The wait and signal overheads in the worst case are sufficiently high that communications among operating systems processes, which typically occur from 100 to 300 times per second, cannot be handled efficiently by programs such as we have given earlier. These operations must be incorporated into the machine's instruction set.

A SOLUTION

Suppose that the basic machine has tagged memory: each word of memory contains a tag field containing the type of information stored therein. Tagged memories were an integral part of the Rice University Machine [4] and of the Burroughs B6700 [12]. Advanced forms reduce space overhead by tagging regions of memory rather than individual words [10,5].

Suppose that a semaphore is implemented as a semaphore word, as in Figure 2, and that the wait and signal operations are part

of the instruction set of each processor. For this environment, the instructions WAIT x and SIGNAL x operate on a semaphore word stored in Mem[x]. These instructions are uninterruptible because their microprograms do not examine interrupt indicators.

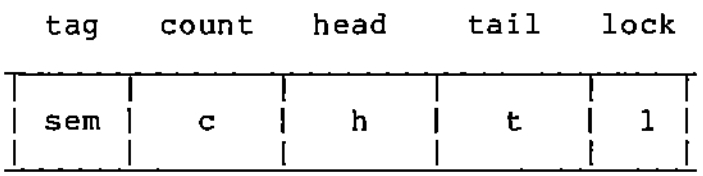

FIGURE 2: Semaphore word.

Tagging permits distributing semaphores throughout data structures without endangering the integrity of wait and signal operations. Tagging increases software reliability by preventing locking operations from being applied to nonsemaphore locations. Note, however, that a type-checking compiler also provides the same advantage. For this application, tagging is more a convenience than a necessity.

While a semaphore operation is in progress on Mem[x], the lock bit is set. Any other processor attempting a semaphore operation on Mem[x] must pause, retrying the operation after a delay. The delay should be about half the time required to complete the operation.

Once in control of a semaphore word, a processor adjusts the count field and, if necessary, moves a process index between the

semaphore queue and the ready list. The ready list emanates from a semaphore word stored in Mem[RL], whose count field is not used. A processor attempting use of the ready list semaphore word must pause and retry after a delay if the ready list is locked. Access to the ready list is embedded in both the WAIT and SIGNAL instructions.

Figures 3 and 4 specify microprograms for the SAVESW, LOADSW, and READY instructions. The dashed lines represent points at which the addressing protocol requires the processor to release the addressed memory bank. SAVESW simply copies the processor registers to the current process control block. LOADSW locks the ready list, removes the head process, unlocks the ready list, and loads the processor registers from the new process control block.

```
SAVESW:  "copy registers into PL[self].stateword"

LOADSW:
         (tag, c, h, t, lock) := Mem[RL]
         IF tag ≠ sem THEN ERROR
         IF lock = 1 THEN "retry after delay 2"
         Mem[RL].lock := 1
         ---------------------------------------
         self := h
         h := Mem[LINK + h]
         ---------------------------------------
         Mem[RL] := (tag, c, h, t, 0)
         ---------------------------------------
         Mem[LINK + self] := 0
         ---------------------------------------
         "load registers from PL[self].stateword"
```

FIGURE 3: Instructions for context switching.

```
READY(i):
         (tag, c, h, t, lock) := Mem[RL]
         IF tag ≠ sem THEN ERROR
         IF lock = 1 THEN "retry after delay 2"
         IF h = 0 THEN Mem[RL] := (tag, c, i, i, 0)
                  ELSE Mem[RL] := (tag, c, h, i, 1)
         ---------------------------------------
                       Mem[LINK + t] := i
         ---------------------------------------
                       Mem[RL].lock := 0   FI
```

FIGURE 4: The READY(i) instruction microprogram.

Both LOADSW and READY begin with a tag check and test of the lock bit; after the ready list has been modified, the lock bit is reset. (Compare with the specifications of the LOCK and UNLK instructions given earlier. Explicit LOCK and UNLK instructions are no longer needed.) LOADSW locks the ready list for four memory reference times; READY locks it for at most four. If the lock bit is set, the action

"retry after delay 2"

means: "release the memory, wait two memory reference times, then restart the microprogram." The delay of 2 is about half the time another processor executing a LOADSW or READY will hold the ready list.

Executing LOADSW when there is only one process, say k, in the ready list will leave RL.head = 0 because LINK[k] = 0. The subsequent READY(i) instruction tests for the empty ready list (head = 0) and sets both head and tail to i in this case. Manipulations of head and tail pointers can be performed during the same memory access that manipulates the lock bit.

The high-level specifications of Figures 3 and 4 are to be understood as descriptions of microprograms. For examples, the symbols

tag, c, h, t, lock, i, self, LINK

refer to registers in the processor. The links (PL[ ].link) are assumed to be stored in a linear array whose base address is in

the local register LINK. The action

```
(tag, c, h, t, lock) := Mem[x]
```

specifies a memory read operation, while

```
Mem[x] := (tag, c, h, t, lock)
```

specifies a write operation. The action

```
self := h
```

specifies a register-register transfer.

Figures 5 and 6 specify microprograms for the WAIT and SIGNAL instructions. These microprograms begin with tag and lock checking. If the semaphore's lock is set, the retry delay is T memory reference times, where T represents half the time another processor will hold the lock in the worst case. If the count c > 0, the WAIT instruction will write c-1 into the count field without setting the lock, completing in 2 memory reference times. If the count $c \geq 0$, the SIGNAL instruction will write c+1 into the count field without setting the lock, completing in 2 memory reference times. Otherwise, these instructions set the semaphore lock and proceed to their critical sections.

WAIT x:

```
(tag, c, h, t, lock) := Mem[x]
IF tag ≠ sem THEN ERROR
IF lock = 1 THEN "retry after delay T"
IF c > 0 THEN Mem[x] := (tag, c-1, h, t, 0)
         ELSE Mem[x].lock := 1
---------------------------------------------------------
         SAVESW
---------------------------------------------------------
         IF h=0 THEN Mem[x] := (tag, c-1, self, self, 0)
                ELSE Mem[LINK + t] := self
---------------------------------------------------------
                     Mem[x] := (tag, c-1, h, self, 0) FI
---------------------------------------------------------
         LOADSW  FI
---------------------------------------------------------
```

FIGURE 5: WAIT instruction microprogram.

SIGNAL x:

```
(tag, c, h, t, lock) := Mem[x]
IF tag ≠ sem THEN ERROR
IF lock = 1 THEN "retry after delay T"
IF c ≥ 0 THEN Mem[x] := (tag, c+1, h, t, 0)
         ELSE Mem[x].lock := 1
-----------------------------------------
              i := h
              h := Mem[LINK + h]
-----------------------------------------
              Mem[x] := (tag, c+1, h, t, 0)
-----------------------------------------
              Mem[LINK + i] := 0
-----------------------------------------
              READY(i)   FI
-----------------------------------------
```

FIGURE 6: SIGNAL instruction microprogram.

The critical section of the WAIT instruction saves the current stateword and attaches 'self' to the semaphore's queue. The new count (c-1), head, tail, and lock value (0) are written to memory in one memory reference time. The LOADSW operation can be placed outside the critical section because the processor has completely dumped the stateword and the old 'self' value; an arbitrary delay can be tolerated until the (uninterruptible!) processor picks up a new process index for execution. This reduces the holding time of the semaphore lock to 4+s memory reference times in the worst case, and makes the ready list locking interval disjoint from the semaphore locking interval.

The critical section of the SIGNAL instruction removes the process index from the head of the semaphore's queue, holding it in a local register, i. This permits the READY(i) microprogram to be executed outside the critical section (but within the context of an uninterruptible microprogram). It reduces the holding time of the semaphore lock to 4 memory reference times in the worst case.

The correctness of these instructions follows from that of the software WAIT and SIGNAL implementation given earlier: the microprograms simulate the previous case. The only changes are putting ready list operations outside semaphore critical sections for reasons noted above.

Performance

The overall space and time requirements of this proposal are summarized in Table 3 and compared with IBM 370 and VAX implementations. "Execution times" are simply the sums of instruction fetches and operand references. Compared with the VAX implementation, the proposed solution with s = 18 runs in roughly 1/3 the time and 1/20 the space, or roughly 1/60 the space-time. The proposed solution also reduces the Ready List lock holding time significantly.

In the long run, as many WAITs will be executed as SIGNALs. This means that the semaphore lock retry delay should be half the average semaphore lock time, or

$$T = \frac{1}{2}\left(\frac{4+s+4}{2}\right) = 2 + \frac{s}{4}$$

memory reference times.

These figures can be used to evaluate the tolerable overhead in tagging semaphore words. Suppose that a single bit were used to distinguish semaphore words from all others. Suppose that WAIT and SIGNAL operations appear statically in approximately equal numbers. On the IBM 370, the average of the WAIT and SIGNAL macros is about 200 bytes (1600 bits) longer than the proposed WAIT and SIGNAL instructions. Therefore programs on the tagged machine could contain 1600 times as many semaphore operations without being longer than their counterparts on the IBM

370. The corresponding figure for the VAX is 500.

	IBM 370		VAX 11/780		Proposal	
	WAIT	SIGNAL	WAIT	SIGNAL	WAIT	SIGNAL
Instruction Storage (bytes)	254	148	74	57	3	3
Execution Time (memory refs)						
Short Path	36	36	13	13	3	3
Long Path	163	67	85	53	10+2s	10
Semaphore Lock Time (memory refs)						
Long Path	145	49	78	46	4+s	4
RL Lock Time (memory refs)						
Long Path	17	20	19	12	4	4

TABLE 3: Comparisons with proposed solution.

A real tagged memory would use larger tags, say 4 bits, to identify more types of data objects in memory. But there is a corresponding savings because other types of macros (e.g., for mixed mode expression evaluation) can be eliminated from object code. Myers [10] reports data showing that most programs become shorter when compiled for a tagged memory instruction set -- com-

mon, replicated macros can be eliminated in favor of one microprogram for the same operation. Dennis [2] reports similarly that tagging easily reduces program size by factors of up to 2.

PRIVATE SEMAPHORES AND I/O OPERATIONS

A private semaphore is a semaphore on which only one process can wait. Private semaphores are especially useful for communicating with input/output processes and for receiving completion-signal interrupts from devices. Every process will have a private semaphore, kept in a field PL[self].psem of its control block.

We suppose that all user processes operate at priority 0 and that system device driver processes, which start devices and I/O controllers and receive completion signals from them, operate at higher priorities. All other processes must interface their I/O operations through driver processes. A device driver process must usually be run soon after the completion of the previous I/O task in order to maintain I/O device utilization as high as possible. To this end, each processor contains a 'priority' register telling the (fixed) priority of the current process (self). The priority register can either be a field of the self register or a component of the stateword. Private semaphores will display

the priority of the waiting process.

Private Semaphores

A private semaphore (Figure 7) can be stored as a field in a process control block or as a component of any other data structure. Its tag is 'psem'. The process index field (i) will be nonzero whenever a process is waiting on the private semaphore. The priority field (p), which contains the priority number of the waiting process, is used to determine if the waiting process must preempt the signaling process. The wakeup waiting bit (w) records a signal sent before the receiving process sought it. The lock bit prevents a signaler from interfering with a receiver.

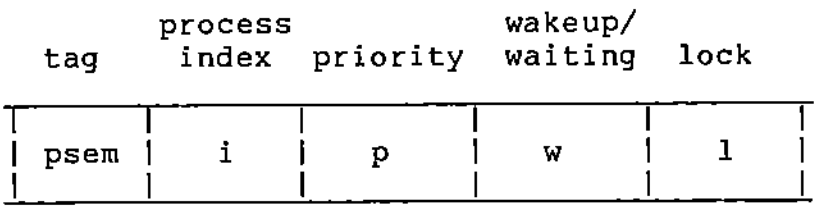

FIGURE 7: Private semaphore.

The machine instructions PWAIT x and PSIGNAL x are used to receive and send, respectively, via a private semaphore word in Mem[x]. Alternatively, the WAIT (SIGNAL) microprogram can be generic, taking the PWAIT (PSIGNAL) action if the tag is 'psem'

rather then 'sem'.

Figure 8 specifies the microprogram for PWAIT; it is simpler than WAIT (Figure 5). It takes 2 memory reference times for the short path and 8+2s for the long. It needs to lock the semaphore while SAVESW is in progress to prevent a signaler from attempting a wakeup before the receiver is fully blocked.

Figure 9 specifies the microprogram for PSIGNAL. Except for actions (a), necessitated by device driver process priorities, it is simpler than SIGNAL (Figure 6). It takes 2 memory reference times for the short path and 10+2s for the long. If no process is waiting (i=0), PSIGNAL sets the wakeup waiting bit. Otherwise, PSIGNAL checks the priority of the waiting process: if higher than that of self, it moves self to the head of the ready list (using a new operation, PUSH, Figure 10) and switches to the waiting process; if not higher, it moves the waiting process either to the head or the tail of the ready list, depending on its priority. The semaphore need not be locked during context switching because the two process indices, self and i, are in private registers of the processor and nowhere else; no other process can attempt to manipulate either PL[self] or PL[i].

```
PWAIT x:
        ------------------------------------------------
        (tag, i, p, w, lock) := Mem[x]
        IF (tag, i) ≠ (psem, 0) THEN ERROR
        IF lock = 1 THEN "retry after delay s/2"
        IF w = 1 THEN Mem[x] := (tag, 0, 0, 0, 0)
                 ELSE Mem[x] := (tag, self, priority, 0, 1)
        ------------------------------------------------
                      SAVESW
        ------------------------------------------------
                      Mem[x].lock := 0
        ------------------------------------------------
                      LOADSW  FI
        ------------------------------------------------
```

FIGURE 8: Wait operation for private semaphore.

```
PSIGNAL x:
        ------------------------------------------------
        (tag, i, p, w, lock) := Mem[x]
        IF (tag, w) ≠ (psem, 0) THEN ERROR
        IF lock = 1 THEN "retry after delay s/2"
        IF i = 0 THEN Mem[x] := (tag, 0, 0, 1, 0)
                 ELSE Mem[x] := (tag, 0, 0, 0, 0)
        ------------------------------------------------
      /                IF p > priority
      |                   THEN SAVESW
      |                        PUSH(self)
      |                        self := i
  (a) /                        "load registers from
      \                           PL[self].stateword"
      |                   ELSE IF p > 0 THEN PUSH(i)
      |                                 ELSE READY(i)
      |                                 FI
      |                   FI
      \           FI
        ------------------------------------------------
```

FIGURE 9: Signal operation for private semaphore.

PUSH(i):

```
-----------------------------------------
(tag, c, h, t, lock) := Mem[RL]
IF tag≠ sem THEN ERROR
IF lock = 1 THEN "retry after delay 2"
IF h = 0 THEN Mem[RL] := (tag, c, i, i, 0)
         ELSE Mem[RL] := (tag, c, i, t, 1)
-----------------------------------------
              Mem[LINK + i] := h
-----------------------------------------
              Mem[RL].lock := 0  FI
-----------------------------------------
```

FIGURE 10: Push operation.

Ideally, a multiprocessor system will solve the "priority problem", which requires that the lowest priority running process must have priority at least as high as the highest priority ready process. In practice this means that a preemption must occur within a short time as soon as a process is enabled whose priority exceeds that of a running process.

The priority mechanism of PSIGNAL does not solve this problem. This is because PSIGNAL may awaken a process whose priority is less than that of 'self' but greater than that of a process running on another processor. The priority mechanism guarantees only that the next LOADSW will give preference to some high priority process. (On a single processor system, however, this mechanism will always run the highest priority enabled process.) An interprocessor broadcast mechanism is required to achieve faster preemption in favor of high priority processes. The ready

list ring proposed in the next section has this property -- it obviates the PUSH operation and eliminates all the steps (a) from Figure 9.

I/O Control

Many systems channel all requests to any given I/O unit through a device driver process in charge of that unit. A driver process has the sole authority to issue STARTIO commands to its device and to receive the completion signals from its device. It also maintains a work queue of requests from all other processes for tasks at that device. All the details of interacting with a given device, from setting up channel programs, to scheduling tasks, and to error recovery, are hidden away inside the driver process.

The work queue of a device driver will contain entries of the form (i, r) where i is a process index and r an I/O-request descriptor. A semaphore 'wsem' counts the number of entries in the work queue. To make a request, a process follows this schema:

```
r := "description of request"
"attach (self, r) to work queue"
SIGNAL wsem
PWAIT PL[self].psem
```

where PL[self].psem is a private semaphore kept in the control block of a process. In the simplest case, where the device

accepts only one request at a time, the driver process follows the schema:

```
1:  WAIT wsem
    (i, r) := "remove request from work queue"
    "generate channel program for request r,
        at starting address j"
    STARTIO(j)
    PWAIT PL[self].psem
    PSIGNAL PL[i].psem
    goto 1
```

The driver's private semaphore is used to receive the device completion signal that eventually results after a STARTIO. The driver then informs the requestor (i) of the task's completion via the requestor's private semaphore.

The last command (HALT) of a channel program instructs the device to enter its "idle" state and generate a completion signal interrupt to the processor that started it. In response, the processor issues the command

```
PSIGNAL x
```

where x is the address of the private semaphore of the device driver that started the I/O operation.

THE READY LIST

The solution outlined above locks the ready list for a minimal time (four memory reference times per operation). Ready list lock contention can still be a problem if there is a lot of process switching.* The contention can be eliminated if each processor has a private window into the ready list, such that each window contains at most one process index and process indices move among the windows.

Ready List Ring

One possible implementation of this principle is a circulating ring of slots (packets), each capable of holding the index of a ready process and its priority number. As sketched in Figure 11, each processor has its own port into the ring. Two of the previous operations are redefined:

LOADSW -- Wait until a used slot comes by; load the (self, priority) registers from the packet and mark the slot as unused. Load the registers from PL[self].stateword and proceed.

*Suppose that the ready list lock holding time is A and the mean interval between ready list accesses by a given processor is B. There can be at most 1/A processors per second completing ready list operations. Then, by Little's Formula, there can be at most an average of B/A processors not waiting at the ready list. Therefore an average of at least N-B/A processors can be lost to ready list contention (N is the number of processors).

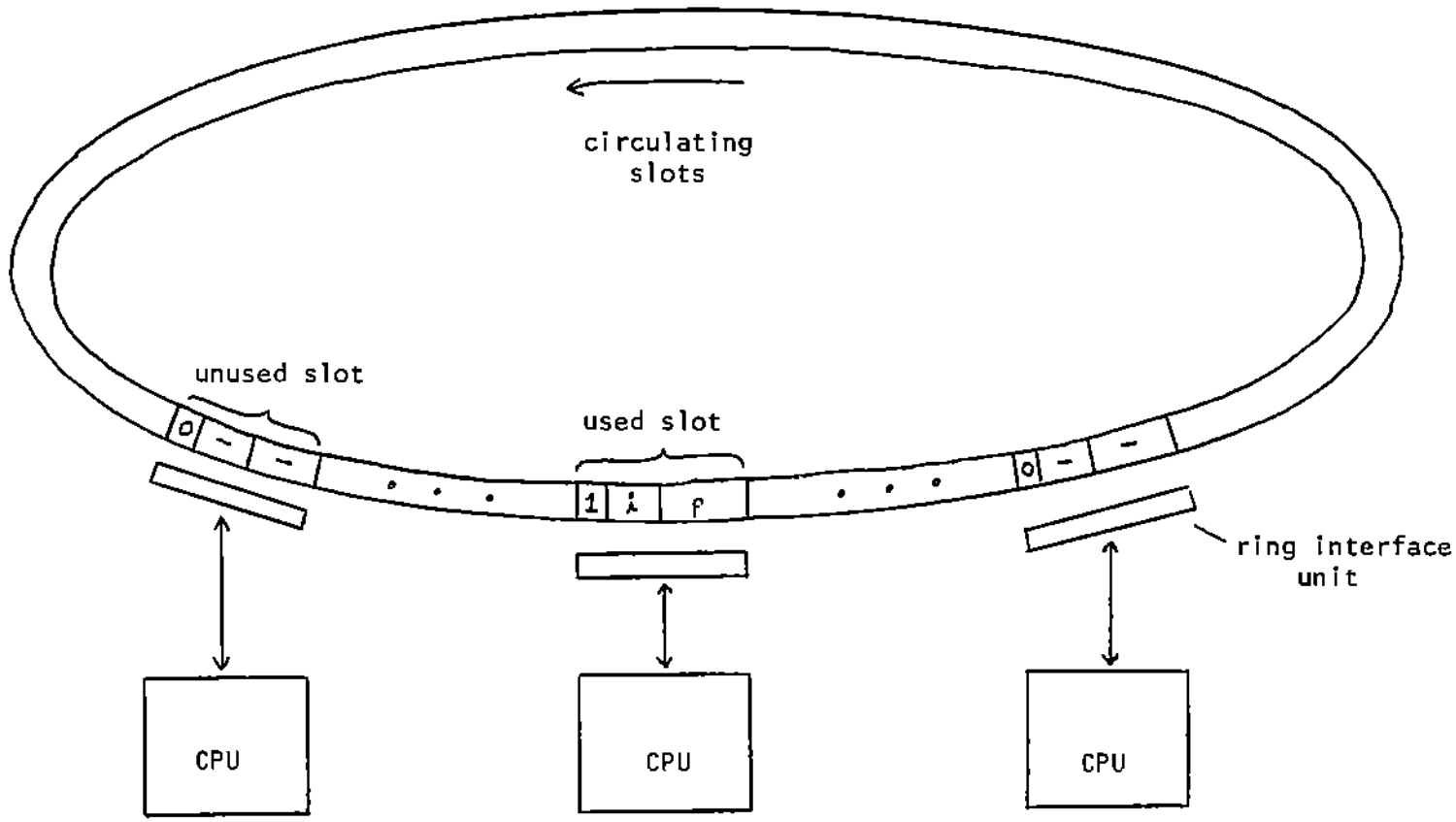

FIGURE 11: Ready list ring.

READY(i,p) -- Wait until an unused slot comes by; store (i,p) in it and mark the slot as used.

These operations replace the microprograms defined in Figures 3 and 4. Note that READY(i,0) can be used in the SIGNAL operation to avoid looking up p in the process list. If the priority numbers are part of process indices, p is implicitly inserted by the ordinary READY(i) operation.

Each processor's ring interface unit monitors the priorities of passing used slots. If a slot (i,p) comes by for which p > priority, the ring interface unit removes the packet, marks the slot as unused, and triggers the following sequence in the processor:

```
READY(self, priority)
SAVESW
(self, priority) := (i,p)
"Load registers from PL[self].stateword"
```

This solves the multiprocessor priority problem noted earlier: a high priority process will preempt an available processor within one ring circuit time. This implies that the PSIGNAL microprogram can be simplified by eliminating all the steps (a) in Figure 9; its shortest and longest path times are then both the same, namely 2 memory reference times.

A special processor can perform "ring management". Its goal is to separate each pair of used slots by an empty slot so as to equalize the LOADSW and READY times under heavy load. If the density of empty slots is too low, this processor can remove

process indices and hold them in its local store. It can insert processes indices back again when the density of empty slots rises. The capacity of the ring is increased by the size of the local store of the ring manager.

The ready list ring can be modelled as a queue with random selection for service. If there are n process indices in the ready list, a given one will be selected by the next LOADSW operation probability 1/n. The mean number of LOADSWs until selection is n, the same as for a FIFO queue. A process will be unselected after k successive LOADSW operations with probability $(1-1/n)^k$, which for large n is approximately $e^{-k/n}$; thus the probability that a process is still waiting after 4n selections is about e^{-4} or 1.8%. Because the nonselection probability decays exponentially, there is no need for a special mechanism to guarantee the eventual selection of a ready process index. In other words, "starvation" is no problem.

Other Implementations

The principle of the ring can be simulated in a conventional system by letting the processors cycle their ports through slots fixed in memory. The RL can be a vector of bytes, RL[1..N-1], in which RL[i] = 1 if process i is ready and 0 otherwise. Let TR(x) denote a test-and-reset operation on the byte Mem[x]; this operation returns the value of the byte and sets it to 0 in one indivisible step. The LOADSW and READY operations become:

```
LOADSW:  while TR( RL[self] ) = 0 do
             self := self + 1 mod N  od
         "load registers from PL[self].stateword"

READY(i): RL[i] := 1
```

CONCLUSION

We have demonstrated that a modest amount of hardware support can significantly reduce the space and time requirements of the primitive operations for process context switching and semaphore management. The proposed WAIT and SIGNAL operations have roughly 1/360 the space-time of the corresponding IBM 370 implementation, and roughly 1/60 the space-time of the corresponding VAX implementation. The proposed implementation reduces ready-list lock holding times to 4 memory reference times. These operations are efficient enough to permit process management without shortcuts and to permit a greater number of processors to be used.

Tagged memory is not critical to our implementation. A type checking compiler, such as for Concurrent Pascal or Ada, can verify that the addresses supplied to WAIT and SIGNAL machine instructions are in fact for semaphore words. The main purpose of tagging is a defense against unreliable programs. Obviously, the combination of a type checking compiler and a tagged memory machine is more reliable than either would be alone. The tagged memory permits semaphores to be distributed among data

structures, which tends to reduce the complexity of the operating system [2].

The discussion of the ready list ring illustrates that a multiport list is not prone to be a bottleneck under heavy use. The probability of ultimate "starvation" is zero even though the list becomes a random selection queue. The technology of ring networks is already well developed -- e.g., the University of Cambridge Ring for connecting machines [11] and the University of Manchester's Dataflow Machine's ring of enabled instructions [7].

ACKNOWLEDGEMENTS

We are grateful to Walter Tichy for carefully reading drafts of this paper, to P. M. Meliar-Smith for pointing out the GEC machine, and to P. Feiler [3] for suggesting improvements in LOADSW and READY. We are also grateful to the National Science Foundation, which supported some of this work through grant MCS78-01729 at Purdue University.

References

1. Batson, A. P. and Brundage, R. E., "Segment Sizes and Lifetimes in Algol 60 Programs," Comm. ACM Vol. 20(1) pp. 36-44 (Jan. 1977).

2. Dennis, T. D., "A Capability Architecture," PhD Thesis, Purdue University (May 1980).

3. Feiler, P. H., "Letter to the Editor," Operating Systems Review, (July, 1980).

4. Feustel, E. A., "On the Advantages of Tagged Architecture," IEEE Trans. Comptrs. Vol. C-22(7) pp. 644-656 (July 1973).

5. GEC Computers Limited, GEC 4000 Series Computers, GEC Computers Limited, Hertfordshire WD6 1RX, England (Nov. 1976).

6. Gehringer, E. F., "Variable-Length Capabilities as a Solution to the Small-Object Problem," Proc. 7th Symp. on Operating Systs. Princs., pp. 131-142 ACM SIGOPS, (Dec. 1979).

7. Gurd, J., Watson, I., and Glaurt, J., "A Multi-layered Data Flow Computer Architecture," Technical Report, Department of Computer Science, University of Manchester, Manchester M13 9PL, England (July 1978).

8. Habermann, A. N., Flon, L., and Cooprider, L., "Modularization and Hierarchy in a Family of Operating Systems," Comm. ACM Vol. 19(5) pp. 266-272 (May 1976).

9. Liskov, B., "The Design of the VENUS Operating System," Comm. ACM Vol. 15(3) pp. 144-149 (March 1972).

10. Myers, G. J., Advances in Computer Architecture, McGraw-Hill (1978).

11. Needham, R. M., "System Aspects of the Cambridge Ring," Proc. 7th Symp. on Operating Systs. Princs., pp. 82-85 ACM SIGOPS, (Dec. 1979).

12. Organick, E. I., Computer System Organization: The B5700/B6700 Series, Academic Press, New York, N.Y. (1973).