Table of Contents

1. The Kernel

The kernel is a small set of subroutines which can be used to provide an environment in which real-time, multiple process applications may be executed. The term "kernel" refers to the fact that these subroutines constitute the inner-most level of software in a simple layered operating system. The kernel contains routines necessary for the creation, scheduling and destruction of processes, process synchronization, and interprocess communication.

The kernel manipulates variables which are global throughout the entire system. For this reason, it is imperative that only a single sequential section of code, either process or interrupt service routine, be allowed access to the kernel at a time. Otherwise the kernel would be prone to the same race conditions that it is intended to prevent.

To insure that only one program can access the kernel at a time, interrupts are automatically disabled whenever the CPU is executing code inside the kernel. This permits the kernel subroutine that is called to execute to completion without interruption. Since, in this implementation, there is only a single CPU which fetches and executes one instruction at a time, no other program can be executing code in the kernel simultaneously.

Some computers have more than one CPU or "engine" sharing memory and executing instructions concurrently. Examples range from the DEC Rainbow personal computer, which contains both a Zilog Z80 and an Intel 8086 microprocessor, to the IBM 3031AP mainframe, which contains two 370 engines. For such machines, our simple technique for enforcing exclusive access to the kernel is not sufficient. Machine instructions which permit busy waiting must be used to serialize kernel access. This also assumes that some sort of hardware memory interlock, even on multiport memories, is available.

Even our technique of disabling interrupts has a flaw. Some peripheral devices are smart enough to access and modify the processor's random access memory directly, rather than requiring the execution of machine instructions by the CPU. This method of I/O data transfer is called Direct Memory Access or DMA (on DEC systems it is often referred to as Non-Processor Request or NPR). A DMA device could read or overwrite kernel data structures even

while interrupts are disabled.

We circumvent this problem with good software design, not through any special instructions or complicated code. It interesting to note, however, that this "hole" has been a traditional method of circumventing memory protection on many production operating systems. It has been said that IBM 360 OS/MFT was particularly vulnerable to this form of attack.

The kernel subroutines which run with interrupts disabled are referred to as Kernel Service Routines, or KSRs. KSRs are said to be atomic or primitive: as far as other software is concerned, a KSR cannot be broken up into smaller executable pieces. Once the execution of a kernel service routine begins, it continues to completion without interruption. In some operating systems, routines of this type are said to have the property "system must complete".

KSRs share this property with the machine instructions executed by the processor. In fact, nearly all computers execute their machine instructions as atomic, uninterruptable computations.

There are a few exceptions to this, the most notable being the IBM 370 series of processors which have several machine instructions which are interruptable and restartable. Most, such as MoVe Character Long and Compare Logical Character Long, are used to manipulate arbitrarily long blocks of data. They achieve this by keeping all intermediate results in general purpose registers, which, as we shall see shortly, are saved and restored whenever the current process loses control of the CPU.

Our discussion of the kernel will include both detailed descriptions of the data structures manipulated by the kernel, and the algorithms used to implement the various kernel service routines. Some of the data structures are unique, existing as a single copy inside the kernel, and others are generic, having instantiations scattered in processes throughout the real-time application. We will also explain exactly how Kernel Service Routines are made to be atomic.

Most of the code for the kernel will be presented in both C and PDP-11 assembly language. Although C is useful for studying the logic of the algorithms, it occasionally lacks the precision necessary to allow the reader to really understand some of the

very low-level implementation details.

This is particularly true with much of the stack manipulation, since stack management is carried out automatically by the C run-time routines, and is not directly accessible to the programmer. Since so much of what the kernel does is stack manipulation (as we shall see shortly), we have chosen to implement the kernel in assembly language.

This is not unusual. It has only been relatively recently that systems programmers have turned to higher-level languages as their implementation language. For example, even though most of the UNIX operating system is written in C, much of its own kernel is written in assembly language, both out of design necessity and for speed.

We have chosen the Digital Equipment Corporation PDP-11 processor as our implementation machine for a variety of reasons. It is well known, extensively documented, and popular for real-time applications and in education and research. Its instruction set is very orthogonal, which makes it simple to understand and to use (much simpler, at any rate, than any other powerful processor). Each of its machine instructions can be implemented with a sequence of one or more machine instructions on other processors. The VLSI microcomputer version, the LSI-11, has extended the PDP-11 architecture into applications requiring small, dedicated machines.

On the other hand, the same kernel discussed here has been implemented and used on Zilog Z80 and Motorola 68000 microprocessors, and in various combinations of assembly language, FORTH and C, so porting the kernel to other processors and architectures is possible (and even encouraged).

In this implementation, the kernel consists of two Macro-11 source files: SY.MAC and CB.MAC. Macro-11 is the standard DEC assembler for the PDP-11.

SY.MAC contains the pure, executable portion of the kernel. This includes the code for the various user-callable subroutines, some subroutines local to the kernel, and initialization code executed once when the kernel begins a run.

CB.MAC contains the impure portion of the kernel. This includes all of the data structures used internally by the kernel

to manage the multiprogramming environment.

These two source files are intended to be compiled once. They do not need to be recompiled for every real-time application unless it becomes necessary to modify the original source files, for example, to allow a larger number of processes to run concurrently. The corresponding object files SY.OBJ and CB.OBJ need only be linked into every real-time application that requires kernel services. The entry point for the resulting executable module is always the main entry point to the initialization code in the kernel, labelled appropriately enough with the global symbol "KERNEL".

These two files contain the complete source for the kernel. There are no hidden machine instructions or macros. Although some of the code in the kernel is admittedly subtle, it is far simpler than the kernels of commercial multiprogramming operating systems. At the same time, the concepts implemented in the kernel are found in all such operating systems.

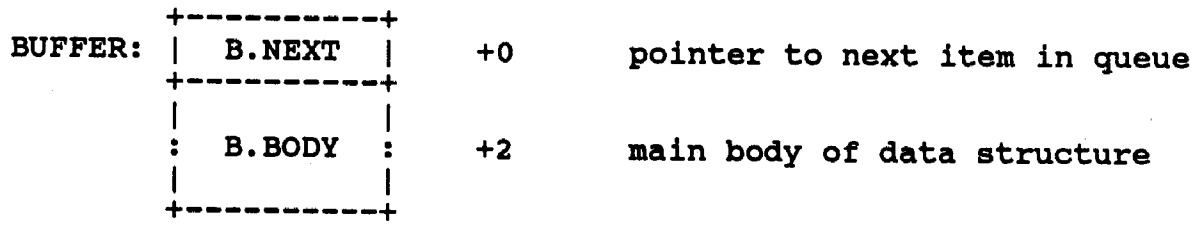
1.1. Queues

The queue is the single most important data structure used by the kernel. A queue is a FIFO list: items are added only to the end of the list, and removed only from its front. The queues used by the kernel are implemented as linked lists. Items are added and removed by the manipulation of pointers, not by the actual movement of data. Two subroutines that are local to the kernel, PUT and GET, are used exclusively to manipulate queues.

1.1.1. The BUFFER Data Structure

All items that may be entered into a queue have in common a simple, external structure, which we refer to generically as a buffer. A buffer contains two fields: B.NEXT and B.BODY.

Diagram



Definition

```
typedef struct  {
                char    **b_next;
                char    b_body[N_BYTES];
                } BUFFER;
```

Declaration

```
BUFFER example_buffer;
```

Figure 1: BUFFER

The first word of a buffer is always a pointer, B.NEXT, which is used as a link to the next entry on the queue. If the buffer is at the end of a queue, B.NEXT contains a special value called NIL, and the link is said to be "grounded". In this implementation, the value of NIL is zero.

After B.NEXT comes a variable length data field, B.BODY. Although we usually think of the body as a variable length character array, its actual length and structure depends upon the type of data structure being queued.

1.1.2. The Queue Header

The identity of a queue is established by a data structure called a queue header. A unique queue header forms the head of every queue. A queue header contains four fields: Q.NUM, Q.SLOTS, Q.FRONT and Q.REAR.

Q.NUM is the negative of the count of entries in the queue. If there are four items linked the queue, Q.NUM equals minus four (-4). As we shall see later, if Q.NUM is positive it has quite a different meaning from the one defined here. If the queue is empty, Q.NUM has the value zero. Q.NUM is decremented every time an item is added to the end of the queue, and it is incremented every time an item is removed from the front of the queue.

Q.SLOTS is the number of additional entries permitted in the queue. If the queue is empty, Q.SLOTS has the value of the total number of entries allowed in the queue. Q.SLOTS is decremented every time an item is added to the queue, and incremented every time an item is removed. Q.SLOTS should never take on a negative value.

Q.FRONT is a pointer to the first item in the queue. If the queue is empty, Q.FRONT has the value NIL.

Q.REAR is a pointer to the last entry in the queue. Unlike Q.FRONT, if the queue is empty, Q.REAR is not grounded. Its value in this case depends upon the address of the last item removed from the queue, as we shall see when we discuss the kernel service routines which operate on queue headers. In any case, the precise value of Q.REAR is not meaningful when the queue is empty.

For example, a queue DOGS with three entries, SPOT, SHEP and FRED, is shown in below. No more than eight DOGS are allowed in

Diagram

```
          +----------+
QUEUE:    | Q.NUM    |      +0      number of items in queue
          +----------+
          | Q.SLOTS  |      +2      number of free slots in queue
          +----------+
          | Q.FRONT  |      +4      pointer to first item in queue
          +----------+
          | Q.REAR   |      +6      pointer to last item in queue
          +----------+

          Q.BYTES = Q.REAR + 2
          Q.WORDS = Q.BYTES / 2
```

Definition

```
typedef struct  {
                int     q_num, q_slots;
                BUFFER  **q_front, **q_rear;
                } QUEUE;
```

Declaration

```
QUEUE example_queue;
```

Figure 2: QUEUE

this queue (why? [1]).

[1] Q.NUM equals -3, indicating 3 entries on the queue. Q.SLOTS equals 5, indicating 5 more remaining "free" positions. The total number of entries allowed on the queue is the sum of the two. If the queue were empty, NUM would equal 0, and SLOTS 8.

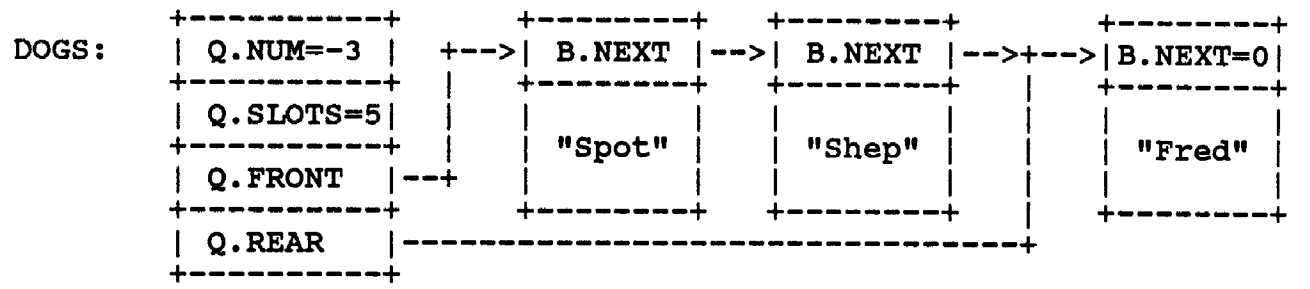


Figure 3: A Queue of DOGS

1.1.3. The PUT Subroutine

There is a single subroutine, PUT, used exclusively by kernel service routines to insert an item at the end of a queue.

PUT receives two arguments: a pointer, Q, to the queue header, and another pointer, B, to the item to be inserted.

An attempt to PUT an item in a queue where Q.SLOTS is zero causes fatal error. This error is provided for control and debugging purposes. The initial value of Q.SLOTS is determined by the user-written code which initializes the queue header.

The PDP-11 assembly language version of PUT uses two registers for parameter passing. R4 must contain the address of the queue header, and R5 must contain the address of the item to be inserted.

PUT is a subroutine in the usual PDP-11 sense. PUT is not itself atomic, but it is local to the kernel and is effectively

[2] PUT is called only by kernel service routines, which are atomic. PUT cannot be called by code outside the kernel. Thus, PUT is logically a part of whatever kernel service routine calls it.

atomic (why? [2]).

1.1.4. The GET Subroutine

In a manner similar to PUT, there is a single routine, GET, used exclusively by kernel service routines to remove the first item from the front of a queue.

GET receives one argument, Q, a pointer to a queue header, and it returns a pointer to the item, B, that it removed. If GET is applied to a empty queue (Q.NUM is not less than zero), a fatal error occurs.

The PDP-11 assembler version of GET receives the address of the queue header in R4, and returns the address of the item in R5. Like PUT, GET is not itself atomic.

1.1.5. The Q.REAR Pointer

Note the handling of Q.FRONT and Q.REAR when GET removes the last item from the queue. Q.FRONT always receives the contents of the Q.NEXT pointer from the item, so Q.FRONT is always set to NIL when the last item is removed (why? [3]).

When PUT places an item in the queue, it always adjusts Q.REAR to point to the new item. GET, on the other hand, does not modify Q.REAR under any circumstances. Thus, after the last item is removed from a queue, Q.FRONT is grounded, but Q.REAR still contains the address of the last item.

This has a couple of useful side effects. Since Q.REAR should be initialized to NIL when the queue header is created, the programmer can discriminate between a queue that was never used, and a queue that was used but is currently empty. This information can be useful when debugging, although one should be careful about assigning too much meaning to the actual value of Q.REAR when the queue is empty.

---

[3] It is because the B.NEXT field of each item PUT into the queue is always set to NIL. Since items are always PUT at the end of a queue, the last item in the queue always has B.NEXT equal to NIL.

Algorithm

```
VOID put(q,b)
QUEUE *q;
BUFFER *b;
        {
        if (q->q_slots > 0)
                {
                if (q->q_num >= 0)
                        q->q_front = b;
                else
                        *q->q_rear = b;
                q->q_rear = b;
                *b = GROUND;
                q->q_slots--;
                q->q_num--;
                }
        else
                error("PUT: fatal error; SLOTS <= 0");
        }
```

Implementation

```
PUT:
        TST     Q.SLOTS(R4)
        BLE     30$
        TST     Q.NUM(R4)
        BLT     10$
        MOV     R5,Q.FRONT(R4)
        BR      20$
10$:
        MOV     R5,@Q.REAR(R4)
20$:
        MOV     R5,Q.REAR(R4)
        CLR     @R5
        DEC     Q.SLOTS(R4)
        DEC     Q.NUM(R4)
        RTS     PC
30$:
        HALT
```

Calling Sequence

```
        MOV     Q,R4            ; pointer to queue header
        MOV     B,R5            ; pointer to buffer
        JSR     PC,PUT
```

Figure 4: PUT

Algorithm

```
BUFFER *get(q)
QUEUE *q;
        {
        BUFFER *b;
        if (q->q_num < 0)
                {
                b = q->q_front;
                q->q_front = *b;
                q->q_slots++;
                q->q_num++;
                return b;
                }
        else
                error("PUT: fatal error; NUM >= 0");
        }
```

Implementation

```
GET:
        TST     Q.NUM(R4)
        BGE     10$
        MOV     Q.FRONT(R4),R5
        MOV     @R5,Q.FRONT(R4)
        INC     Q.SLOTS(R4)
        INC     Q.NUM(R4)
        RTS     PC
10$:
        HALT
```

Calling Sequence

```
        MOV     Q,R4            ; pointer to queue header
        JSR     PC,GET
        MOV     R5,B            ; pointer to buffer
```

Figure 5: GET

1.2. Concurrent Processes

A process is a sequential computation which has a virtual CPU and its own state. That is, each process appears to have its own processor, with exclusive access to its own general purpose registers, stack, status word and so forth.

This is, of coarse, a fiction. In this implementation, there is only a single processor with a single set of registers, program counter, and status word, which all processes must share. The information unique to each process which is stored in these shared variables when the process is running is what we refer to as the process' state or context.

Many operating systems refer to processes as "tasks". While "process" is probably more generic, the two terms are usually used interchangeably. Likewise, the term "multitasking" is analogous to "multiprogramming". On the other hand, "multiprocessing" actually means something completely different. For historical reasons, this term refers to an environment which contains more than one CPU.

1.2.1. The Process Control Block

In the kernel, the simulation of multiple processors is implemented by representing each process with a unique data structure: a Process Control Block or PCB. A PCB contains five fields: P.NEXT, P.PC, P.PS, P.SP and P.STACK.

When a process is not running, its PCB contains the process's stack and all of the necessary state information to resume execution of that process. Thus, only the current process, the process whose instructions are actually being executed by the CPU, "owns" the CPU registers and status word. The contents of these locations are saved in the current process' PCB when that process loses control of the CPU, and restored whenever its computation is resumed.

The P.NEXT field is used similarly to the B.NEXT in a buffer. It is necessary so that GET and PUT may be used to insert and remove PCBs in queues.

The P.PC and P.PS fields contain the initial values of the Program Counter and the Processor Status word for that process. These fields are provided only as a convenience for debugging. In

Diagram

```
        +----------+
PCB:    |  P.NEXT  |      +0       pointer to next PCB in queue
        +----------+
        |  P.PC    |      +2       initial value of Program Counter
        +----------+
        |  P.PS    |      +4       initial value of Processor Status
        +----------+
        |  P.SP    |      +6       saved value of Stack Pointer
        +----------+
        |          |
        :          :
        | P.STACK  |      +N.STACK base of stack
        +----------+

        P.BYTES = P.STACK
        P.WORDS = P.BYTES / 2
```

Definition

```
typedef struct pcb      {
                        pcb     *p_next;
                        PROCESS (*p_pc)();
                        int     p_ps;
                        LONG    p_sp;
                        LONG    p_stack[N_STACK];
                        } PCB;
```

Declaration

```
PCB example_pcb;
```

Figure 6: PCB Declaration

the current implementation, the kernel never makes use of them once they have been initialized. P.PC and P.PS are often used to determine which Process Control Block is associated with which process, since the P.PC field contains the address of that pro-

cess' entry point.

The P.SP field is used to store the value of the process' Stack Pointer when that process is not current. The value of P.SP should always point somewhere within the stack inside that process' PCB; otherwise stack underflow or overflow has occurred.

When the process is current, the value stored in the P.SP field is not especially relevant. In this case, it is the actual Stack Pointer register which references the top of the stack.

The stack inside the PCB is the usual PDP-11 stack accessible to the process. Besides being used for normal computation and subroutine linkage, the stack is used to save the state of the process when it is not current. The state is saved in the stack in the format shown below.

The actual Program Counter and Processor Status are saved on the stack (near the bottom of the figure) along with the other general purpose registers whenever the current process loses control of the CPU. It is always possible for the stack to already contain data when this state information is saved on it. This could be information pushed onto the stack by the process, or it could be the result of nested calls to kernel service routines (more about that later).

Most operating systems which support multiple processes have some data structure analogous to the PCB, sometimes referred to as the TCB or Task Control Block.

1.2.2. The CP Current Process Pointer

The kernel has a one word variable, CP, which it uses to remember the address of the PCB of the currently running process. CP is not only necessary for process management within the kernel, its value can be of use when debugging. CP is modified only by code inside the kernel.

1.2.3. The READY Queue

The kernel has a queue header, READY, which it uses to form a queue of the PCBs of all of the processes in the system that

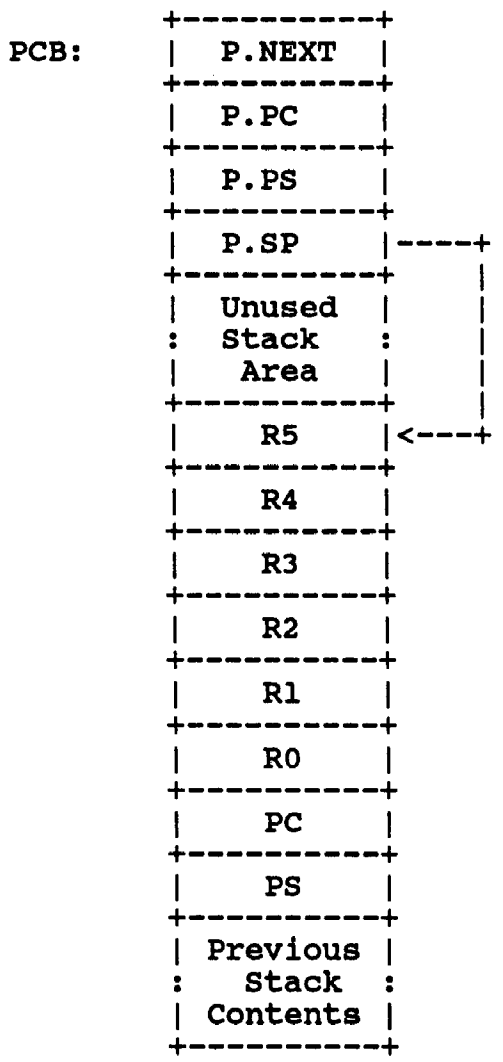


Figure 7: Process State in a PCB

are ready to run, but are not the current process.

Since the kernel uses PUT and GET exclusively to manipulate the READY queue, the PCBs of ready processes circulate in a strictly round-robin fashion. Thus, the kernel is completely fair

```
Declaration

PCB *cp;

Implementation

CP:     .BLKW
```

Figure 8: CP

```
Declaration

QUEUE ready;

Implementation

READY:  .BLKW   Q.WORDS
```

Figure 9: READY

in its scheduling and dispatching of processes for execution.

Other strategies are possible and often necessary. Many operating systems use a prioritized ready queue to give preference to certain tasks. These might include handlers for high-speed I/O devices or on-line applications offering fast response time. However, whenever the scheduling of ready processes is not fair, policies should be implemented to prevent starvation of low-priority processes. A common technique is to raise the scheduling priority of a ready process each time it is passed over in favor of a higher-priority task.

So far the round-robin scheduling strategy used by the kernel has offered fairness with little loss in efficiency. This is because most of the processes managed by the kernel tend to be highly I/O bound; they do not consume much CPU time before they

relinquish the processor to another ready process. This is generally true of real-time applications.

Once again, the contents of READY are valuable when debugging, and READY is modified only by code inside the kernel.

1.3. Requesting Kernel Services

Each kernel service routine is called by issuing the PDP-11 machine instruction EMT, or EMulator Trap. EMT is one of several PDP-11 machine instructions which cause a software trap. Other such instructions are TRAP and IOT (or I/O Trap, so called for historical reasons, not because it intrinsically has anything to do with input/output).

Software traps are handled in a manner very similar to I/O interrupts. When an EMT instruction is executed, the current values of the PS and PC are saved on the stack (PC on top). A new PS and PC are loaded from a particular trap vector. The loading of the PC transfers control to the trap handling software.

The use of a software trap instruction to request operating system services is common. The DEC RT-11 operating system for the PDP-11 also uses the EMT instruction, while the multiuser RSX-11M uses TRAP. The Motorola 68020 microprocessor provides a similar TRAP instruction, while IBM/370 mainframe has an SVC instruction (SuperVisor Call or SerVice Call depending on how old your IBM manual is).

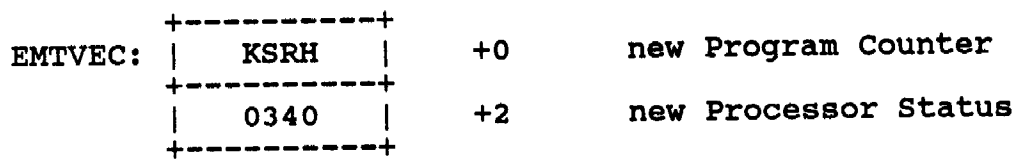


Figure 10: Example Trap Vector

1.3.1. The Kernel Service Request Handler Routine

The trap vector for the EMT instruction is at location 030(8). In this implementation, the PC portion of the vector points into the kernel at the Kernel Service Request Handler, or KSRH, routine, and the PS portion contains a CPU priority of seven, or value 0340(8). The kernel's own initialization code

```
KSRH:
        MOV     R0,-(SP)        ; save R0
        MOV     2(SP),R0        ; retrieve PC (points past EMT)
        MOVB    -2(R0),R0       ; fetch index from EMT instruction
        JMP     @TABLE(R0)      ; jump on index
TABLE:
        .WORD   $SW             ; entry point of SW
        .WORD   $P              ; entry point of P
        .WORD   $V              ; entry point of V
        :       :
   (remainder of KSRH table)
```

Figure 11: KSRH

assigns appropriate values to the EMT trap vector.

This design has two advantages.

First, the application software never needs to know the actual location in memory of the kernel, since this information is supplied by the kernel itself in the EMT trap vector.

Second, when the EMT is executed, interrupts are automatically disabled (how? [4]) and control is transferred to KSRH in a single machine instruction.

The EMT instruction has the opcode 104000(8). The right-hand byte of the instruction is ignored by the PDP-11 CPU. Thus the instructions 104000, 104002 and 104145 are all the same EMT machine instruction.

The kernel uses the lower byte of the EMT instruction to indicate which kernel service is being requested. For example, the P service routine is called by executing the opcode 104002,

[4] Because the PS portion of the EMT vector contains a CPU priority of seven, a priority higher or equal to the interrupting priority of any peripheral device. Since the EMT machine instruction itself is atomic, there is no window in which I/O interrupts can occur once the EMT instruction has been issued. Software traps of any type are always serviced regardless of the current CPU priority in the PS.

while the V service routine is called with the opcode 104004. In either case, the EMT software trap is handled by the processor in an identical fashion.

The KSRH routine is, for all intents and purposes, an interrupt service routine. Instead of a peripheral device requesting an I/O service, KSRH is invoked by a process requesting kernel services through the use of the appropriate EMT instruction.

When an EMT trap occurs, the processer automatically pushes the PC and PS of the interrupted code onto the stack in the current PCB. Once entered, the KSRH routine saves R0 on the same stack so that it may be used as a work register. This is how the first three words of the current process' state get placed on the stack. KSRH retrieves the PC from the stack (it's now one word down from the top). This PC points just past the EMT instruction which caused the trap (why? [5]). KSRH uses the PC to get the EMT instruction, and uses the lower byte of that instruction, the part that the CPU ignores, as an index into a table of addresses. Each address is an entry point into a different kernel service routine.

Because of this minimal decoding, the space of allowable EMT instructions in this implementation is one fourth of those actually possible: the lower byte must be a positive, even number (104000, 104002, 104004, The reason for this restriction is not just for convenience in decoding. Interrupts from peripheral devices are disabled for the duration that the processor is executing inside the kernel. Thus, data could be lost if the time spent in the kernel, at CPU priority seven, is too long. Because kernel calls are made so frequently, and KSRH must be executed once for every such call, it is vital that the EMT decoding be as short and efficient as possible.

KSRH serves as the common entry point for all kernel service routines. There is a corresponding common exit point, EXIT. EXIT explicitly restores R0 from the stack, and executes a ReTurn from Interrupt, or RTI, instruction. RTI restores the PC and PS from the stack, causing the processor to resume execution just past

---

[5] The PDP-11 always updates the Program Counter after fetching the next instruction, and before actually executing it. Thus, when the EMT is executed, the PC is already pointing to the next instruction past the EMT.

```
EXIT:
        MOV     (SP)+,R0        ; restore R0
        RTI                     ; return to caller
```

Figure 12: EXIT

the EMT instruction.

1.3.2. Coding Kernel Service Requests

We should say a word or two now about how the kernel service routines are actually invoked by the programmer from a Macro-11 source program.

A KSR is called by merely coding its name. For example, the programmer codes "P" as if it were an instruction mnemonic in order to invoke the P KSR. The "P" is in fact not a instruction mnemonic nor is it a macroinstruction, and the arguments to each KSR must be set up in the appropriate registers by the programmer prior to the execution of the P. The Macro-11 assembler will mark each kernel service request as an unresolved external reference.

In the SY.MAC file, each EMT instruction used to call a Kernel Service Routine has equated to it a unique symbolic name. For example, the symbol "P" is equated to the opcode 104002 (EMT 2). These equated symbols are also declared to be global. Thus, they can be referenced by separately compiled modules, and resolved at link time.

When the Macro-11 assembler finds a number coded in the opcode field in open code, it assumes that a .WORD directive should have preceded it. Thus, a "2" is assumed to be ".WORD 2".

When Macro-11 finds an undefined mnemonic without any arguments, it assumes the same thing. Thus, "P" is assumed to be ".WORD P".

When symbols are used in a .WORD directive, the numeric definitions to which they are equated are inserted into the

object code at that point.

When an undefined symbol is used, a zero is inserted into the object code and the reference is marked as an unresolved external reference, left to be filled in at link time. The linker utility finds the symbol "P" defined in SY.OBJ, and resolves the reference ".WORD P", inserting the value 104002 into the object code.

Thus, when a "P" appears in a Macro-11 source program, the corresponding EMT instruction is eventually inserted at the appropriate location in the resulting executable module at link time.

It is considerably easier done than said.

1.4. Context Switching

The act of saving a process' state in its PCB, and restoring another process' state from its PCB, is called a context switch. The number of context switches a particular combination of CPU and operating system can perform per second is often used as a measure of system performance. This metric can be a heavily weighted factor, because in order to provide services to a large number of on-line users, respond to different external events in real-time, or schedule many concurrent processes, it may be necessary to perform thousands of context switches per second.

Slow context switching can lead to long response times, lost data, or lengthy execution delays. Some processors, like the Motorola 68000 and the Digital Equipment Corporation VAX, provide specialized machine instructions to save and restore the state, or context, of a process.

1.4.1. The PUSH and POP Subroutines

There are two subroutines local to the kernel, PUSH and POP, which are used to save and restore a process' state.

PUSH saves the state of the current process on the stack in the PCB pointed to by CP, in preparation for that process to relinquish the CPU.

POP restores the state of a process from its PCB, and establishes that PCB as that of the current process by storing the PCB address in CP, just prior to that process resuming computation.

Just prior to the calling PUSH, the current process' stack contains the PS, PC and R0 placed there by the EMT instruction and the KSRH code. A picture of the stack at that point is shown in below.

PUSH then saves the remaining general purpose registers on the stack. This is first done automatically by the PDP-11 Jump to SubRoutine instruction which is used to call PUSH. R1 is saved on the stack, and at the same time the return address of

---

[6] The use of R1 as the argument of the JSR instruction may be less familiar than the usual JSR PC. In fact, any register may be used. Although the CPU performs exactly the same

---

Implementation

```
PUSH:
        MOV     R2,-(SP)        ; save remaining registers
        MOV     R3,-(SP)
        MOV     R4,-(SP)
        MOV     R5,-(SP)
        MOV     CP,R5           ; save address of current process
        MOV     SP,P.SP(R5)     ; save final stack pointer
        JMP     @R1             ; return to caller
```

Calling Sequence

```
        JSR     R1,PUSH
```

Figure 13: PUSH

---

the caller of PUSH is placed in R1 [6].

PUSH then explicitly saves the remaining registers, R2 through R5, on the stack. It then stores the current SP (now pointing to the contents of R5 on top of the stack) in the special P.SP field in the PCB, and returns the address of the current process' PCB (as found in CP) in R5. The stack now looks like the figure below.

POP is the inverse of PUSH. It expects the address of a PCB to be in R5. POP stores R5 in CP, retrieves the Stack Pointer from the P.SP field of the PCB, restores registers R5 through R1, and falls through to the EXIT routine. EXIT, as discussed earlier, completes the restarting of the process by restoring R0 and executing an RTI which in turn restores the PC and PS. Once the RTI is executed, the process' computation is resumed by the CPU. In many operating systems the process is said to have been

---

sequence of operations regardless of what register was specified, the end results, due to the special nature of the Program Counter, are quite different.

Implementation

```
POP:
        MOV     R5,CP           ; establish as current process
        MOV     P.SP(R5),SP     ; restore stack pointer
        MOV     (SP)+,R5        ; restore almost all registers
        MOV     (SP)+,R4
        MOV     (SP)+,R3
        MOV     (SP)+,R2
        MOV     (SP)+,R1
EXIT:                           ; same EXIT as defined above
        MOV     (SP)+,R0
        RTI
```

Calling Sequence

```
        BR      POP
```

Figure 14: POP

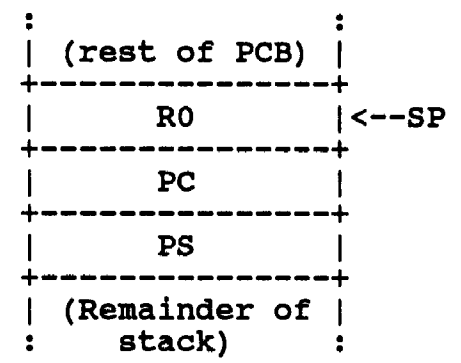


Figure 15: Stack before PUSH

---

dispatched.

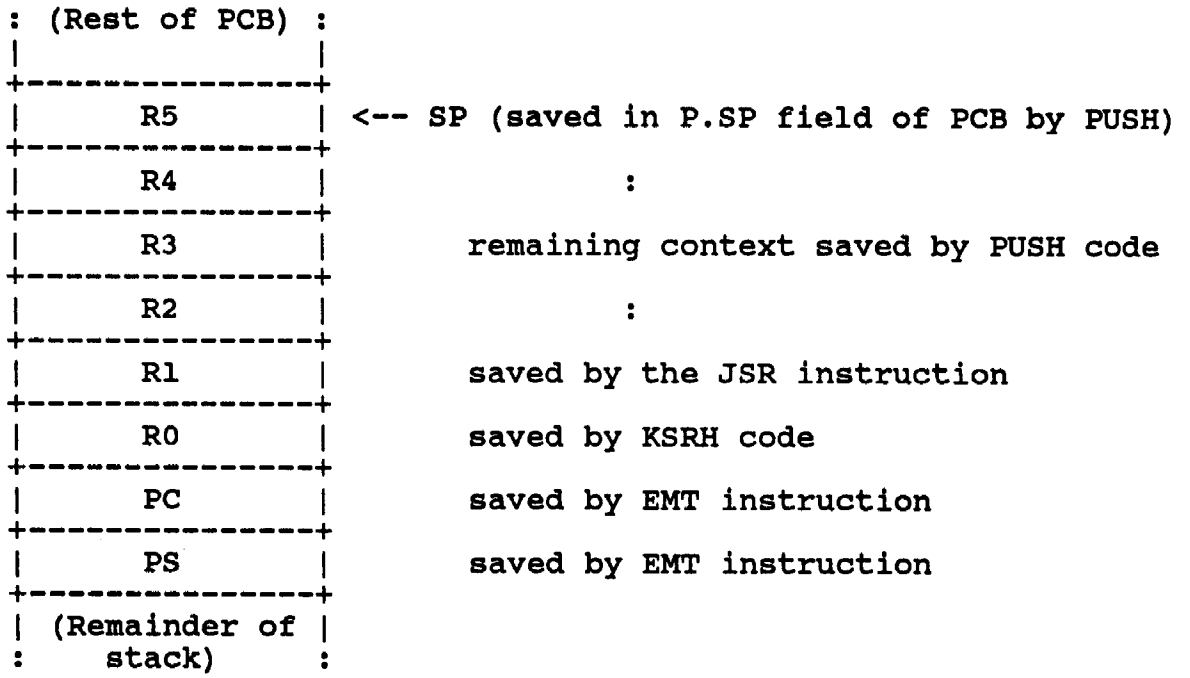


Figure 16: Stack after PUSH

We should mention a few subtleties about PUSH and POP.

First, a call to POP does not usually directly follow a call to PUSH. An arbitrary amount of kernel code may be executed after PUSH saves the SP in one PCB, and before POP restores a new SP from another PCB. What stack does the intervening code use? The answer is that it uses the stack pointed to by the Stack Pointer register (which has already been saved in P.SP); that is, the stack in the old PCB. It is just that any changes made in the SP will not be reflected in the stored stack pointer in the PCB. We often say that the intervening code "borrows" stack space from the process losing control of the CPU.

Second, the process that initially requests a kernel service via an EMT instruction is often not the process to which control returns upon execution of the RTI in EXIT. If PUSH and POP were executed by the kernel, then the SP points to a stack inside a different PCB. The PC restored by RTI is different from that saved by EMT.

The switching to a different stack in another PCB is the essence of the context switch. It is the point at which the execution of one process suspends and the execution of another resumes.

1.4.2. The SW Kernel Service Routine

Occasionally a process may wish to voluntarily relinquish control of the processor for an indefinite amount of time. The process may be waiting on a non-interrupting event, and, not wanting to tie up the CPU with busy waiting, request that it be rescheduled to run at a later time.

The SHORT WAIT, or SW, KSR provides this function in a simple, elegant manner.

Algorithm

```
ATOMIC VOID sw()
        {
        put(ready,push());
        pop(get(ready));
        }
```

Implementation

```
$SW:
        JSR     R1,PUSH
        MOV     #READY,R4
        JSR     PC,PUT
        JSR     PC,GET
        BR      POP
```

Calling Sequence

```
        SW
```

Figure 17: SW

SW simply performs a PUSH, PUTs the PCB on the READY queue, GETS a PCB off of the READY queue, and executes POP. This has the effect of placing the current process at the end of the queue of ready-to-run processes, and restarting the first ready process in it place. If there are no other ready processes, the process merely resumes computation immediately. If there are other ready processes, the process is guaranteed that it will eventually run again as soon as its turn at the head of the READY queue arrives.

The amount of time that the execution of the process is actually suspended is unknown and essentially non-deterministic. It depends upon the number of processes ahead of it in the READY queue, and how long each of those processes will run before relinquishing the CPU.

SW has no arguments. None are necessary, since the context switch is unconditional.

1.4.3. The Null Process

We have briefly discussed the READY queue, and we have mentioned the fact that the typical real-time application is highly I/O bound. What happens when all processes in the system are waiting for I/O (or some other event) to complete? Specifically, what does the kernel run when there are no ready processes?

This is not as simple a question as it might seem. Central processing units have a habit of continuously fetching instructions to execute. Whatever operations the processor might be given to wile away the time between performing useful work must not interfere with the handling of I/O and the associated interrupts. Moreover, whenever a process finally becomes ready, it should be scheduled to run and dispatched as soon as possible.

This implementation solves this problem by having a Null Process. Null is a process which performs no useful work and repeatedly issues SW to request an unconditional context switch.

Thus, there is always at least one ready process: Null. If no other processes are ready to run, the Null Process' PCB merely circulates continuously between CP and READY; the READY queue is empty only when Null is current, and it always has at least one

---

[7] Perusal of the code for SW reveals that the current PCB is PUT on READY before the next PCB is removed. If the Null

---

Algorithm

```
PROCESS nu_pr()
        {
        for (;;)
                sw();
        }
```

Implementation

```
NU.PR:
        SW
        BR      NU.PR
```

Figure 18: Null Process

---

entry when the Null Process performs the short wait (why? [7]).

When another process become ready (that is, its PCB is PUT on the READY queue), its execution will be resumed as soon as Null issues its short wait.

There is one disadvantage to this implementation. Although SW is coded like a single machine instruction, it in fact invokes a Kernel Service Routine consisting of many instructions, all executed at CPU priority seven. The only part of the Null Processes that executes with interrupts enabled is the branch. Thus, Null spends the vast majority of its time with interrupts disabled.

The design of the Null Process is therefore somewhat self defeating. If a real-time application is highly I/O bound, then most or all processes will be waiting for I/O (and interrupts) to be serviced, and the system will spend most of its time in the Null Process. But during the period when interrupts are most likely to occur, the processor spends most of its time with

---

Process is the only ready process, the current and next PCBs are the same Process Control Block.

interrupts disabled.

So far this has not been a significant problem. Apparently there are sufficient "windows" during which interrupts may be serviced that data is not lost. However, an alternative algorithm for Null offers the advantage of spending most of its time with interrupts enabled. This is illustrated below. The new algorithm is a dirtier design in that Null must now directly access the READY queue, a data structure that (at least conceptually) is local to the kernel.

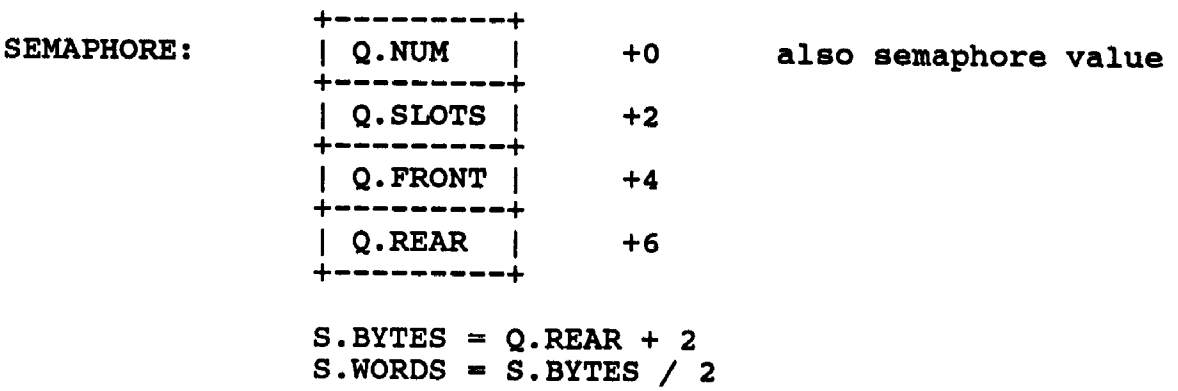
```
PROCESS nu_pr()
        {
        for (;;)
                {
                while (empty(ready)) do;
                sw();
                }
        }
```

Figure 19: Alternative Null Process

1.5. Semaphores

Process synchronization is implemented, in part, by the semaphore data structure. Along with two Kernel Service Routines, P and V, semaphores provide a means of conditional context switching. In this implementation, all semaphores are Dijkstra's counting semaphores.

Diagram



Definition

```
typedef QUEUE SEMAPHORE;
```

Declaration

```
SEMAPHORE example_sem;
```

Figure 20: SEMAPHORE

A semaphore really consists of two data structures: a counter and a queue header. The counter is used to enumerate the number of available resources represented by the semaphore, while the queue header is used to queue the PCBs of processes waiting for those resources. These two variables are combined into a single four-word data structure whose format is similar to a queue

header.

When there are no processes waiting for the resource, the value of Q.NUM is greater than or equal to zero. In this case, Q.NUM is the counter. Its value is a positive number representing the number of available resources.

When one or more processes are waiting for one unit of the resource, their PCBs are queued on the queue header. In this case, the use of the semaphore is identical to that of a queue header. In particular, Q.NUM is the negative of the number of PCBs linked on the queue (or, the number of processes waiting for a unit of resource). Since the data structure in question is a queue, the PCBs are linked in the same order in which they made their requests.

Now we have explained the meaning of a positive number in the Q.NUM field of a queue header. It may be so only when the queue header is in fact being pressed into double duty as a semaphore. Note that the READY queue is never used as a semaphore. Thus, its Q.NUM field should never have a value greater than zero.

1.5.1. The P Kernel Service Routine

The P KSR implements Dijkstra's WAIT primitive.

P receives one argument, S, a pointer to a semaphore. In the Macro-11 implementation, R4 is expected to point to the semaphore.

If the value of Q.NUM is zero (the resource is not available) or negative (in addition, one or more processes are already waiting to use the resource), the current (calling) process is "put to sleep" on the semaphore. Its state is PUSHed onto the stack in its PCB (pointed to by CP) and the PCB is PUT onto the end of the semaphore queue header. A PCB of a ready process is removed from the front of READY queue and made current by POP.

If the value of Q.NUM is greater than zero (one or more units of the resource is available), the current (calling) process proceeds to execute and, presumably, use the acquired resource.

Note that the Q.NUM field of the semaphore is always decremented, either directly by the else clause of P, or indirectly by

Algorithm

```
ATOMIC VOID p(s)
SEMAPHORE *s;
        {
        if (s->q_num <= 0)
                {
                put(s,push());
                pop(get(ready));
                }
        else
                s->q_num--;
        }
```

Implementation

```
$P:
        TST     Q.NUM(R4)
        BGT     10$
        JSR     R1,PUSH
        JSR     PC,PUT
        MOV     #READY,R4
        JSR     PC,GET
        BR      POP
10$:
        DEC     Q.NUM(R4)
        BR      EXIT
```

Calling Sequence

```
        MOV     S,R4            ; pointer to semaphore
        P
```

Figure 21: P

the execution of PUT.

P is similar to SW in the sense that the process that calls P may not be the process to which P returns. That is determined

by the value of the semaphore.

1.5.2. The V Kernel Service Routine

The V KSR implements Dijkstra's SIGNAL primitive.

Algorithm

```
ATOMIC VOID v(s)
SEMAPHORE *s;
        {
        if (s->q_num < 0)
                put(ready,get(s));
        else
                s->q_num++;
        }
```

Implementation

```
$V:
        TST     Q.NUM(R4)
        BGE     10$
        MOV     R5,-(SP)
        JSR     PC,GET
        MOV     R4,-(SP)
        MOV     #READY,R4
        JSR     PC,PUT
        MOV     (SP)+,R4
        MOV     (SP)+,R5
        BR      EXIT
10$:
        INC     Q.NUM(R4)
        BR      EXIT
```

Calling Sequence

```
        MOV     S,R4            ; pointer to semaphore
        V
```

Figure 22: V

V receives one argument, S, a pointer to a semaphore. In the Macro-11 implementation, R4 is expected to point to the semaphore.

If the value of Q.NUM is negative (at least one process is waiting to use the resource just released), the first PCB linked to the semaphore queue header is removed and PUT onto the READY queue. That process has been woken up, and will execute as soon as its turn arrives at the front of READY. The current (calling) process continues to execute.

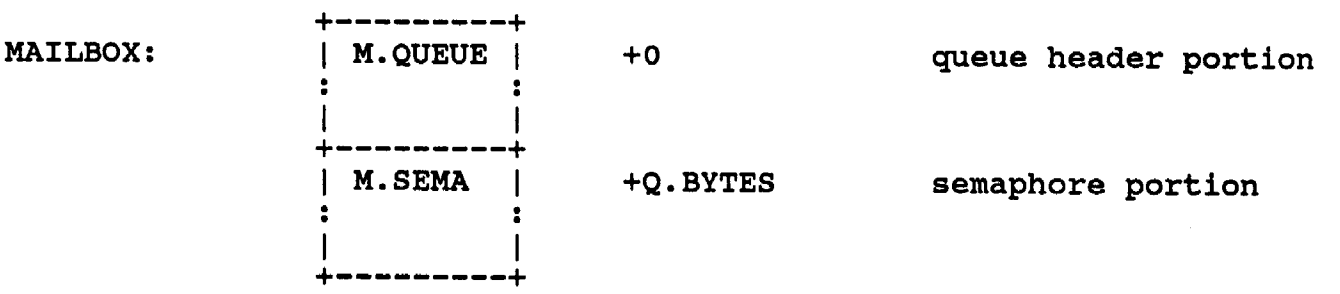
If the value of Q.NUM is zero or larger, then there are no waiting processes. The current (calling) process continues to execute.

In either case, the Q.NUM field of the semaphore is always incremented, either directly by the else clause of V, or indirectly by the execution of GET.

1.6. Mailboxes

Interprocess communication is implemented, in part, by the mailbox data structure. Along with two Kernel Service Routines, SEND and RECV (RECeiVe), mailboxes provide a mechanism for asynchronous message passing between processes. In other operating systems, mailboxes are sometimes referred to as "message queues" or "ports" (not to be confused with hardware I/O ports).

Diagram



Definition

```
typedef struct  {
                QUEUE           m_queue;
                SEMAPHORE       m_semaphore;
                } MAILBOX
```

Declaration

```
MAILBOX example_mailbox;
```

Figure 23: MAILBOX

A message is any data structure that has a format compatible with PUT and GET (that is, it conforms to the type definition of BUFFER). Any other details of its body are strictly up to the SENDing and RECeiVing processes. As we shall see later in the

KSRs for creating and destroying processes, the kernel occasionally treats Process Control Blocks as messages, moving them in and out of mailboxes with SEND and RECV.

We say that the message passing is asynchronous in this implementation because the sending process does not wait for the receiving process to actually get the message. The sender issues the SEND KSR and continues to execute. An arbitrarily long time may elapse before the receiving process becomes ready, then current, and actually completes the RECV. Messages are not automatically acknowledged, although this may easily implemented.

Mailboxes offer a more sophisticated method of synchronization than do semaphores. Processes which use mailboxes not only wait and signal each other, but they exchange data as well.

A mailbox consists of two data structures: a queue header followed by a semaphore. Messages sent to a mailbox are linked to the queue header until requested by the receiving process. If a process attempts to receive a message and the queue is empty, its PCB is queued on the semaphore. If a process is queued on the semaphore when a message arrives, its PCB is removed and placed on the READY queue. As we might expect, PUT, V, P and GET are used to implement this.

1.6.1. The SEND Kernel Service Routine

The SEND KSR implements the sending portion of the asynchronous message passing.

SEND expects two arguments: M, a pointer to a mailbox, and B, a pointer to a message buffer. In the Macro-11 implementation, R4 is assumed to point to the mailbox, and R5 to the message buffer.

SEND first PUTs the buffer on to the queue header portion of the mailbox. It then Vs the semaphore portion.

If no processes are waiting for the message that has just arrived, the semaphore value will merely be incremented, indicating that there is now one more unit of the resource available. A process that attempts a RECV on the mailbox will receive the message immediately and continue execution without waiting.

If one or more processes are waiting, V wakes the first process up by removing its PCB from the semaphore queue and placing

---

Algorithm

```
ATOMIC VOID send(m,b)
MAILBOX *m;
BUFFER *b;
        {
        put(m->m_queue,b);
        v(m->m_semaphore);
        }
```

Implementation

```
$SEND:
        JSR     PC,PUT
        ADD     #M.SEMAPHORE,R4
        V
        SUB     #M.SEMAPHORE,R4
        BR      EXIT
```

Calling Sequence

```
        MOV     M,R4            ; pointer to mailbox
        MOV     B,R5            ; pointer to message buffer
        SEND
```

Figure 24: SEND

---

it on the READY queue. The message buffer will remain linked to the queue header until the receiving process becomes current and resumes execution.

In either case, the sending process continues execution after completion of the SEND.

1.6.2. The RECV Kernel Service Routine

The RECV KSR implements the receiving portion of the asyn-

Algorithm

```
ATOMIC QUEUE *recv(m)
MAILBOX *m;
        {
        p(m->m_semaphore);
        return(get(m->m_queue));
        }
```

Implementation

```
$RECV:
        ADD     #M.SEMAPHORE,R4
        P
        SUB     #M.SEMAPHORE,R4
        JSR     PC,GET
        BR      EXIT
```

Calling Sequence

```
        MOV     M,R4            ; pointer to mailbox
        RECV
        MOV     R5,B            ; pointer to message buffer
```

Figure 25: RECV

---

chronous message passing.

RECV expects one argument: M, a pointer to a mailbox. In the Macro-11 implementation, R4 is assumed to point to the mailbox, and when RECV returns, R5 contains the address of the received message buffer.

RECV first Ps the semaphore portion of the mailbox.

If resources are available (that is, one or more messages are linked to the queue header portion), the first message is removed from the queue header with GET. The receiving process receives the address of the message and continues execution. The value of the semaphore is decremented by P to indicate that one

less unit of resource is now available.

If the queue header is empty, the value of the semaphore will be zero. When RECV applies the P, the receiving process will go to sleep on the semaphore, perhaps joining other processes, until resources are once again available. When the receiving process resumes execution, it will execute the GET, removing the very message placed there by the sending process.

1.7. Process Creation and Destruction

Processes are created, started and destroyed through the use of the CRP (CReate Process), STP (STart Process), and EOP (End Of Process) Kernel Service Routines. Any process in the system may use CRP and STP to create and start a new process, providing there is an available Process Control Block in the kernel. A process can be destroyed, and its PCB made available to the kernel, only by itself issuing a suicidal call to EOP. Processes cannot destroy one another.

1.7.1. The PCBS Free PCB Mailbox

Every process must be represented by its own unique Process Control Block. When the kernel creates a new process, which is done whenever the CRP request is issued, it must have available an unused PCB which it initializes with the state specific to the new process.

Many operating systems create new PCBs by allocating memory dynamically from a large pool of unused storage. However, not only is dynamic memory management beyond the scope of our topic, the method of allocating, releasing and managing memory dynamically can be so expensive in terms of time that many real-time systems cannot afford to use it.

Instead the kernel allocates PCBs statically at compile time. This means that the total number of PCBs in the system is fixed when the real-time application using the kernel is compiled. Thus, there is a limit to the number of processes that may be running at any one time.

The kernel has a mailbox, PCBS, that contains unused Process Control Blocks.

PCBS does not itself contain the actual storage allocation for the Process Control Blocks. It is merely a mailbox in which unused PCBs are queued. In this sense we are treating PCBs as messages which may inserted and removed from PCBS using SEND and RECV.

The actual allocation for the Process Control Blocks is in FPCBS, the Free PCB Pool. FPCBS is an linear array of PCBs allocated at compile time. The Process Control Blocks are queued up in PCBS when the kernel initializes these data structures at run

---

Declaration

MAILBOX pcbs;

Implementation

```
PCBS:    .BLKW    M.WORDS
```

Figure 26: PCBS

---

time.

Declaration

PCB fpcbs[n_pcbs];

Implementation

```
FPCBS:   .BLKW    <N.PCBS*P.WORDS>
```

Figure 27: FPCBS

1.7.2. The CRP Kernel Service Routine

The CRP KSR is used to allocate and format an unused PCB in order to create, or "spawn", a new process. CRP does not start the new process. It merely returns the address of the formatted PCB to the calling program.

CRP has as its arguments the entry point, E, and initial status word, L, of a program. CRP creates a new process by receiving an unused Process Control Block from PCBS and formatting it so that when the PCB is POP'ed as it becomes current, the

restored context will cause the program to begin execution.

In this implementation, the "saved" context of the new process consists of zeroed general purpose registers R1 through R5. When the new process begins execution, it will find in its R0, on the other hand, the address of its own PCB. The process may choose to use this address or discard it.

The Macro-11 implementation of CRP expects to find the entry point address in R4 and the initial value of the status word in R5. CRP returns a pointer to the PCB in R5.

Note that CRP does not place the PCB on the READY queue. This is so that the parent process is free to massage the PCB of its child process in any way it sees fit prior to actually submitting the process for execution. For example, the parent may place parameter values in the saved registers in the PCB. These values will be restored by POP when the child eventually becomes current.

Since PCBS is a mailbox, and CRP uses RECV to acquire an unused PCB, it is always possible that there will be no Process Control Block available, and the requesting parent process will go to sleep on the semaphore portion of the PCBS mailbox. The parent process will remain asleep until a PCB finally becomes available.

1.7.3. The STP Kernel Service Routine

The STP KSR actually starts, or "dispatches" the new process. It does this by queueing the PCB formatted by CRP on to the READY queue.

The Macro-11 implementation of STP expects R5 to point to the PCB. STP calls PUT to place the PCB into the READY queue.

1.7.4. The EOP Kernel Service Routine

The EOP KSR provides a mechanism for a process to "self-destruct". EOP has no arguments. Since the kernel request can only be issued by the current process, EOP can always get the

---

Algorithm

```
ATOMIC PCB *crp(e,l)
PROCESS (*e)();
unsigned l;
        {
        PCB *c;
        LONG *sp;
        c=recv(pcbs);
        c->p_next=GROUND,c->p_ps=l,c->p_pc=e;
        sp=&(c->p_stack);
        *(--sp)=c->p_ps,*(--sp)=c->p_pc,*(--sp)=c;
        *(--sp)=0,*(--sp)=0,*(--sp)=0,*(--sp)=0,*(--sp)=0;
        c->p_sp=sp;
        return(c);
        }
```

Implementation

```
$CRP:
        MOV     R4,-(SP)
        MOV     R5,-(SP)
        MOV     #PCBS,R4
        RECV
        CLR     P.NEXT(R5)
        MOV     (SP)+,P.PS(R5)
        MOV     @SP,P.PC(R5)
        MOV     R5,R4
        ADD     #P.STACK,R4
        MOV     P.PS(R5),-(R4)
        MOV     P.PC(R5),-(R4)
        MOV     R5,-(R4)
        CLR     -(R4)
        CLR     -(R4)
        CLR     -(R4)
        CLR     -(R4)
        CLR     -(R4)
        MOV     R4,P.SP(R5)
        MOV     (SP)+,R4
        BR      EXIT
```

Calling Sequence

```
        MOV     #E,R4           ; entry point of new process
        CLR     R5              ; initial status of new process
        CRP
        MOV     R5,C            ; pointer to PCB of new process
```

Figure 28: CRP

Algorithm

```
ATOMIC VOID stp(c)
PCB *c;
        {
        put(ready,c);
        }
```

Implementation

```
$STP:
        MOV     R4,-(SP)
        MOV     #READY,R4
        JSR     PC,PUT
        MOV     (SP)+,R4
        BR      EXIT
```

Calling Sequence

```
        MOV     C,R5            ; pointer to PCB of new process
        STP
```

Figure 29: STP

address of the current PCB from CP.

EOP merely SENDs the current PCB back to PCBS, the mailbox of unused Process Control Blocks, where the PCB may be reformatted and reused by CRP at a later time. If a process is waiting on PCBS for an available Process Control Block with which to spawn a new process, the waiting process is woken up and made ready in the usual fashion.

Finally, EOP GETs the PCB of the next ready process and POPs

Algorithm

```
ATOMIC VOID eop()
        {
        send(pcbs,cp);
        pop(get(ready));
        }
```

Implementation

```
$EOP:
        MOV     #PCBS,R4
        MOV     CP,R5
        SEND
        MOV     #READY,R4
        JSR     PC,GET
        BR      POP
```

Calling Sequence

```
        EOP
```

Figure 30: EOP

---

its context.

1.7.5. Reentrant Programs and Multiple Processes

So far we may have left the reader with the impression that in this implementation each process has associated with it a unique executable code segment. Although this is usually true, it is not necessarily the case.

Code is said to be reentrant when it is written so that it can be used concurrently by several different processes. Reentrant code is "pure"; it has no local variables in memory and cannot be self-modifying. The stack and the general purpose

---

[8] The stack is inside the process' PCB, and the registers

registers are, however, fair game (why? [8]).

The use of reentrant code is common in multiprogramming operating systems. For example, a single reentrant code segment may be written to serve a single terminal port. A different driver process may be created to serve each separate port, and all of these terminal servers point into the same reentrant code segment. The only difference between each driver process is the address of the port that it services, and the mailboxes, semaphores and other local variables it may use. These variables are allocated from the stack, or kept in registers, and so are unique to each process.

Another application would be an editor. Text editors are usually very memory intensive utilities. A reentrant editor can contribute a substantial amount to reduced system overhead. On a multi-user system, each programmer communicates with a unique editor process, but only a single copy of the actual editor program resides in memory. On the other hand, each editor process has its own local variables and scratchpad editing area.

Compilers for modern languages such as Pascal and C usually generate reentrant code. The compilers themselves can be reentrant, so that they can be used concurrently by several users. Writing reentrant code in assembly language for processors such as the PDP-11 requires some care but it is not difficult.

Since, in our implementation, only a single process executes at a time, how can several processes use a reentrant routine concurrently?

If the routine is a subroutine used by several interrupt service routines which can interrupt one another, then several ISRs can be inside the routine at a time, even though only one ISR is actually being executed by the CPU.

If the routine is used by several processes, and it contains kernel service requests such as P, RECV or SW, then several processes may be in the midst of using the routine, even though

---

are saved and restored on the stack when the process loses and regains control of the CPU. Thus, the stack and the registers are local to each process, not to the code segment which uses them.

only a single process is current at any one time.

Finally, if the real-time application contains a watch-dog timer, a process may time-out while in the reentrant routine, and another process allowed to run which calls the same routine. This scenario is a combination of the nested interrupt case and the sleeping process case discussed above.

1.8. Initialization

Finally, we will discuss how a real-time application using the kernel begins execution; how the kernel initializes its own data structures, how user processes are created and started, and how the user processes themselves are expected to initialize.

1.8.1. The Initialization Process

The kernel, as part of its own initialization, creates exactly two processes: the Null Process, which we have already discussed, and the Initialization Process.

The code for Null is defined inside the kernel source file SY.MAC, so that its entry point is known to the kernel at compile time. The Initialization Process, on the other hand, is defined by the programmer. Its source is in a separately compiled module, and its entry point must be a globally defined name. The address of this entry point is resolved at link time.

The sole purpose for Init is to create and start all other processes. Init may also be used to initialize any global variables. Typically once the Initialization Process has completed it self-destructs with a call to EOP. This is traditional, but there is no reason why Init could not take a more active part in the application program.

The reason for having Init is simple. The kernel needs to know about only two processes: Null and Init. The programmer writing the real-time application does not need to modify the source for the kernel to add or remove processes. The programmer merely codes a single process, Init, which does all the necessary grunge work.

The example Initialization Process shown below creates two processes with entry points "producer" and "consumer", and then destroys itself. Immediately after EOP returns the now unused PCB to the PCBS pool, it will dispatch the next ready process to execute. This process will be the first process started by Init, namely producer.

In the Macro-11 implementation the kernel assumes that the name of the Initialization Process is "INI.PR". This cannot be

---

Algorithm

```
PROCESS ini_pr()
        {
        stp(c(&p&producer,0));
        stp(crp(&consumer,0));
        eop();
        }
```

Implementation

```
        .GLOBL  INI.PR
INI.PR:
        MOV     #PRODUCER,R4
        CLR     R5
        CRP
        STP
        MOV     #CONSUMER,R4
        CLR     R5
        CRP
        STP
        EOP
```

Figure 31: Initialization Process

---

changed without modifying the kernel.

1.8.2. User Process Initialization

We have described the semaphore and mailbox data structures that processes use to synchronize and communicate. Clearly since at least two processes or interrupt service routines must access a semaphore or mailbox in order to perform Ps, Vs, SENDs and RECVs, such a structure is a shared variable between two or more routines. One of the processes must be responsible for the initialization of that variable.

Regardless of which process performs the initialization, the other processes cannot issue a KSR operation against the variable until it is guaranteed that the initialization has been

performed. A semaphore might be used to synchronize the processes, holding up all others until one signals that the initialization is complete. Unfortunately this solution begs the question, since the "initialization semaphore" is subject to the same synchronization problem.

We solve this by enforcing a simple rule among our programmers. All processes, upon execution, will immediately initialize all local variables and any shared variables that are in their source module or designated their responsibility. After the initialization is complete, the process will issue a Short Wait (SW). Under no circumstances will any process access any shared data structure prior to issuing this first SW.

Algorithm

```
PROCESS producer()
        {
                :
        initialization
                :
        sw();
        for (;;)
                {
                        :
                body of process
                        :
                }
```

Implementation

```
PRODUCER:
                :
        initialization
                :
        SW
BODY:
                :
        body of process
                :
        BR BODY
```

Figure 32: Process Initialization

Consider the state of the system upon completion of the Initialization Process. All of the user processes, and the Null Process, are lined up in the READY queue. The first process in line will be dispatched as soon as the EOP in Init completes. If each process performs its own initialization and then issues an SW, then each process will have a turn at execution before any process begins to perform the body of its code. Even the process started last (and so is at the end of the READY queue) is assured of initializing before any of the previous processes try to communicate with it.

1.8.3. Kernel Initialization

Finally we will examine the kernel's own initialization code, that part of the real-time program that is actually executed first. As we have said previously, the entry point to any application using the kernel is the global symbol "KERNEL", which is at the head of the kernel initialization code.

When the kernel begins execution, it performs the following steps.

(1) The PDP-11 machine instruction RESET is executed. This places all peripherals on the bus in a known state.

(2) The Stack Pointer (R6) is initialized with the address of a temporary "BOOT" stack. This stack is used only during this initialization phase. As soon as the first process (which happens to be the Null Process) is dispatched, the system will begin using the stack in that process' PCB.

(3) The error trap vectors at locations 4(8) and 10(8) are set so that the system will halt if the processor encounters a memory or illegal instruction error.

(4) The EMT vector is initialized with the address of the Kernel Service Request Handler and processor priority seven, so that KSRs may be called.

(5) PCBS, the free PCB queue header, and FPCBS, the associated PCB pool, are formatted. Since no processes have yet been created, PCBS is full of unused Process Control Blocks.

(6) The READY queue is formatted. Since no processes have yet been started, READY is empty.

(7) The Null Process is created and started. PCBS now has one less Process Control Block, and READY contains the PCB of Null.

(8) The Initialization Process is created, but not started. Recall that CRP leaves the address of the new PCB in R5, and POP expects R5 to point to the PCB of the process to be dispatched. The kernel merely jumps to POP. POP restores the context of the Initialization Process, and the application begins to execute.

1.8.4. Debugging the Initialization

A brief note on debugging real-time applications using the kernel that can save hours of time pouring over memory dumps or staring blankly at a terminal display: debug the initialization code first.

Just prior to the EOP in the Initialization Process, place an SW request followed by a HALT instruction. If the processor halts prior to the HALT in Init, then there are already serious bugs in the initialization code. If it halts in appropriately Init, take a dump and verify that all semaphores, mailboxes, CP, READY, and all other variables, shared or not, have correct values.

Consider the state of the system when the HALT in Init is executed. Since we placed the SW and HALT at the end of Init, all process have been created and any variables that Init is responsible for have been initialized. Since we have inserted an SW in Init, all processes in the system have taken one turn at being current and so have executed their initialization code. There is simply no better time to verify that all is well before proceeding with the rest of the application.

Remember, the body of the application cannot possibly run correctly if the initialization is in error. Real-time programs are difficult enough to debug as it is, without allowing the execution of the body to destroy the very evidence that would lead to the bug in the code.