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.

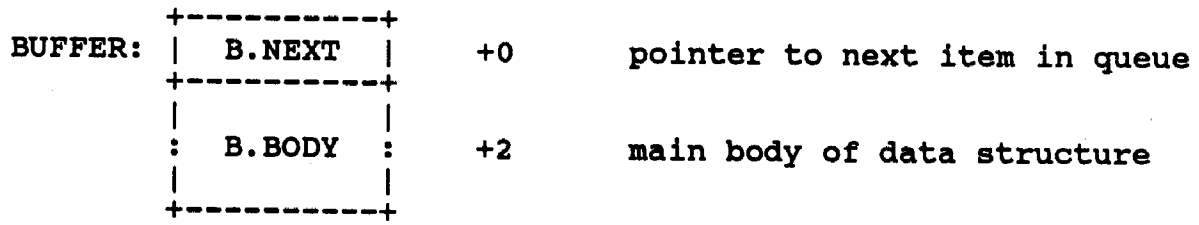
1.1. Queues

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.

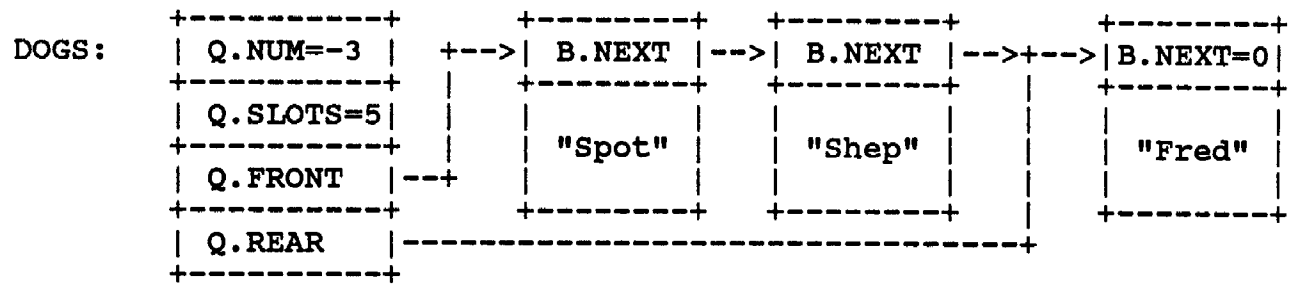


Figure 3: A Queue of DOGS

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

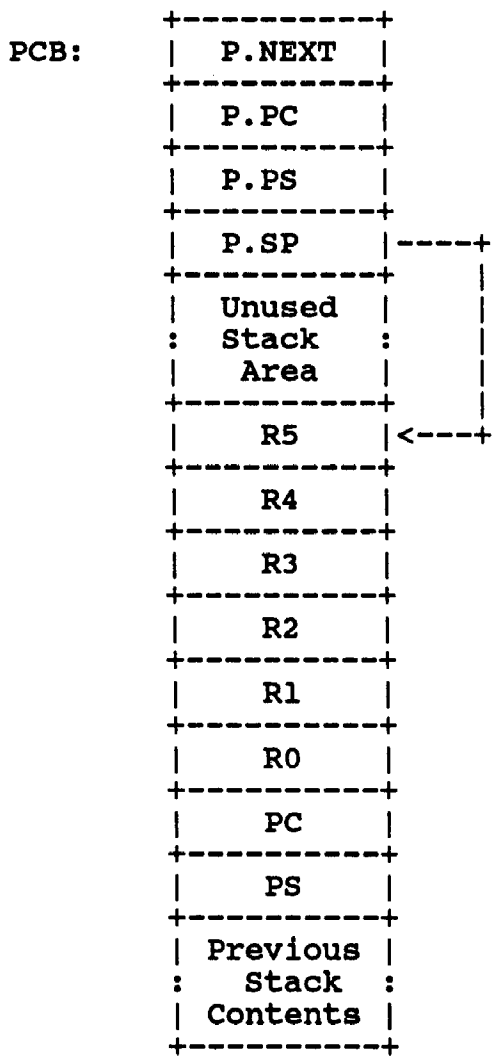


Figure 7: Process State in a PCB

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).

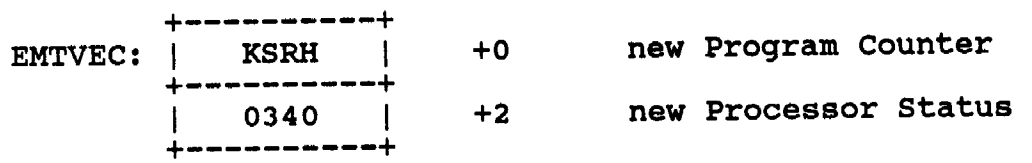


Figure 10: Example Trap Vector

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

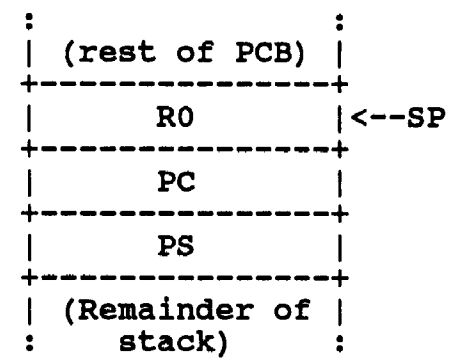


Figure 15: Stack before PUSH

---

dispatched.

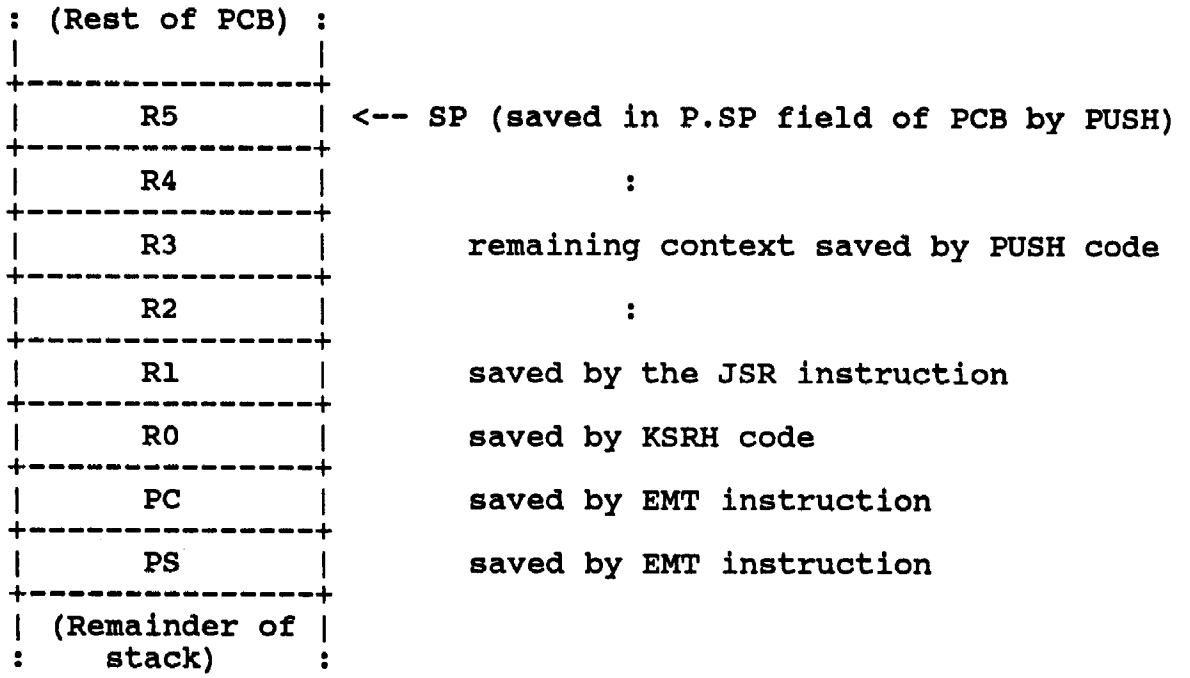


Figure 16: Stack after PUSH

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