

Table of Contents

The Kernel	127
Queues	131
The BUFFER Data Structure	131
The Queue Header	132
The PUT Subroutine	134
The GET Subroutine	135
The Q.REAR Pointer	135
Concurrent Processes	138
The Process Control Block	138
The CP Current Process Pointer	140
The READY Queue	140
Requesting Kernel Services	144
The Kernel Service Request Handler Routine	144
Coding Kernel Service Requests	147
Context Switching	149
The PUSH and POP Subroutines	149
The SW Kernel Service Routine	153
The Null Process	154
Semaphores	157
The P Kernel Service Routine	158

The V Kernel Service Routine	160
Mailboxes	162
The SEND Kernel Service Routine	163
The RECV Kernel Service Routine	164
Process Creation and Destruction	167
The PCBS Free PCB Mailbox	167
The CRP Kernel Service Routine	168
The STP Kernel Service Routine	169
The EOP Kernel Service Routine	169
Reentrant Programs and Multiple Processes	172
Initialization	175
The Initialization Process	175
User Process Initialization	176
Kernel Initialization	178
Debugging the Initialization	179

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 is 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.

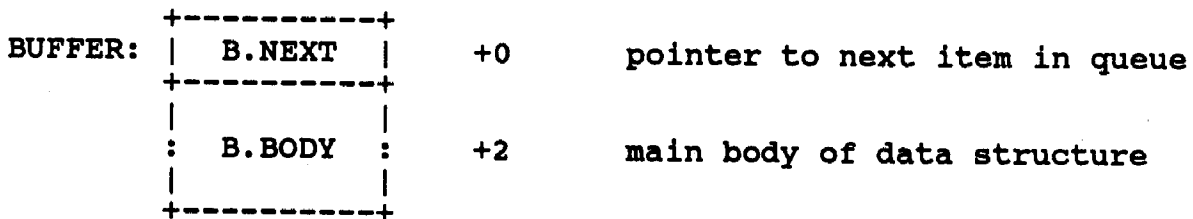
1.1. Queues

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 to 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.

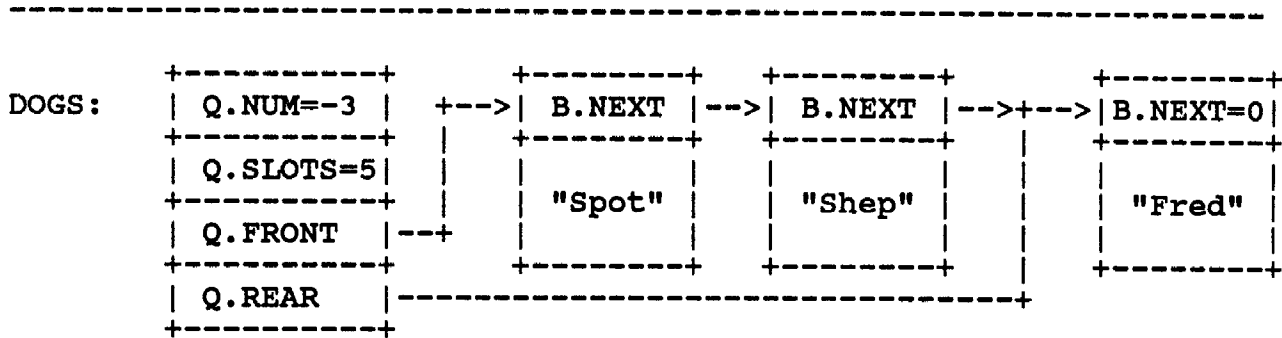


Figure 3: A Queue of DOGS

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 course, 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

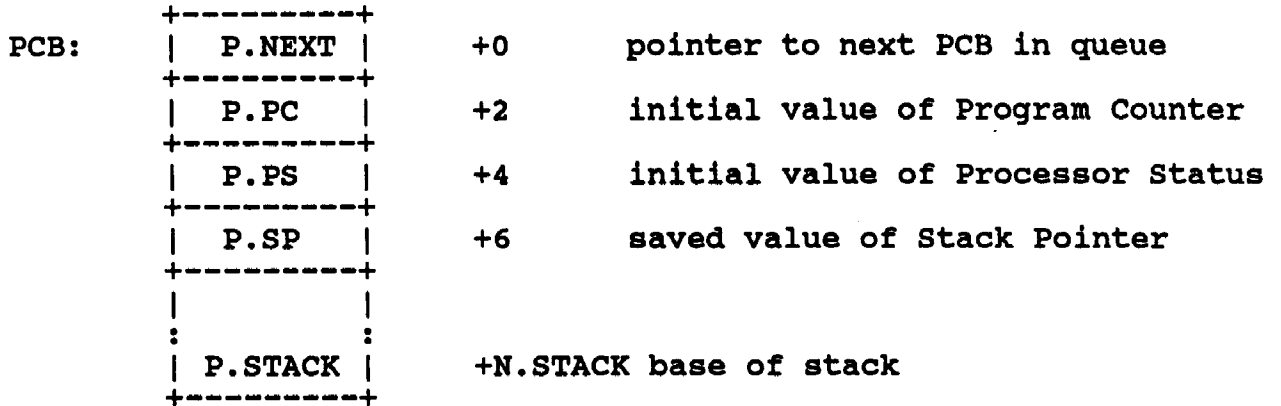
In the kernel, the simulation of multiple processors is implemented by representing each process with a unique data structure: a Process Control Block or PCB. A PCB contains five fields: P.NEXT, P.PC, P.PS, P.SP and P.STACK.

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



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-

