OPERATING SYSTEMS
DESIGN AND IMPLEMENTATION

Third Edition

ANDREW S. TANENBAUM
ALBERT S. WOODHULL

Chapter 2
Processes

The Process Model (1)

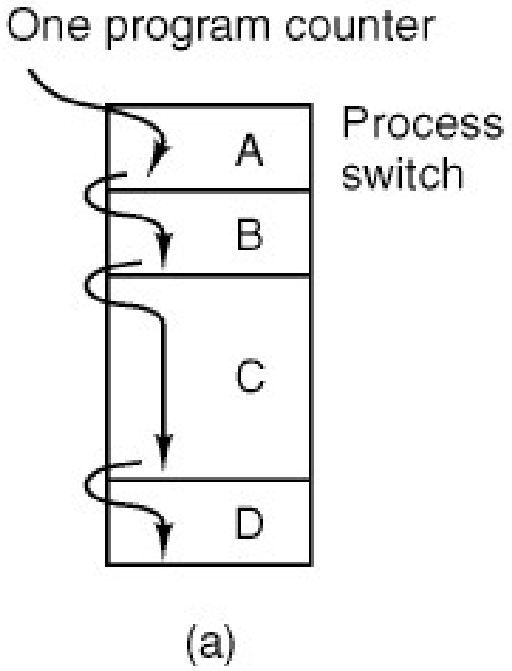
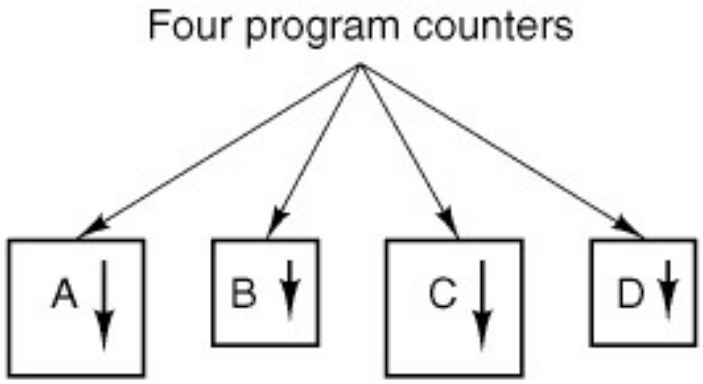


Figure 2-1 (a) Multiprogramming of four programs.

The Process Model (2)



(b)

Figure 2-1 (b) Conceptual model of four independent, sequential processes.

The Process Model (3)

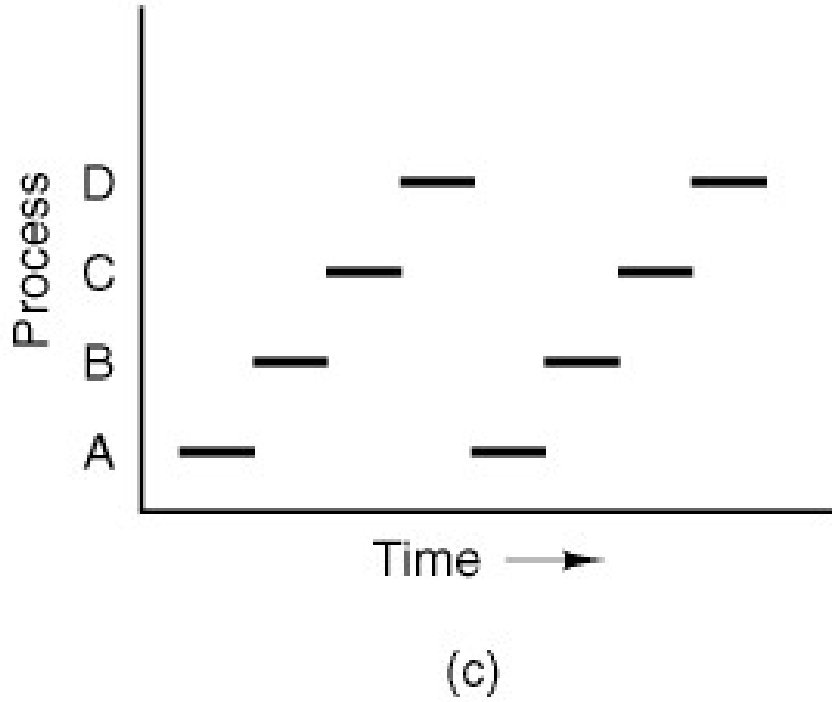


Figure 2-1 (c) Only one program is active at any instant.

Process Creation

Principal events that cause processes to be created:

1. System initialization.
2. Execution of a process creation system call by a running process.
3. A user request to create a new process.
4. Initiation of a batch job.

Process Termination

Conditions that cause a process to terminate:

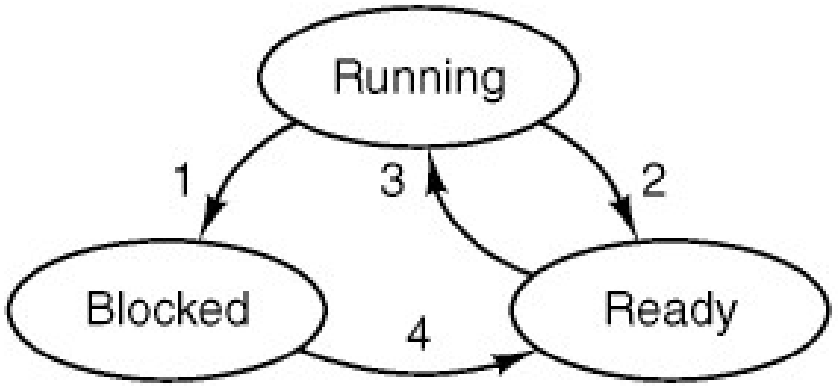
1. Normal exit (voluntary).
2. Error exit (voluntary).
3. Fatal error (involuntary).
4. Killed by another process (involuntary).

Process States (1)

Possible process states:

1. Running
   (actually using the CPU at that instant).
2. Ready
   (runnable; temporarily stopped to let another process run).
3. Blocked
   (unable to run until some external event happens).

Process States (2)



1. Process blocks for input
2. Scheduler picks another process
3. Scheduler picks this process
4. Input becomes available

Figure 2-2 A process can be in running, blocked, or ready state. Transitions between these states are as shown.

Process States (3)

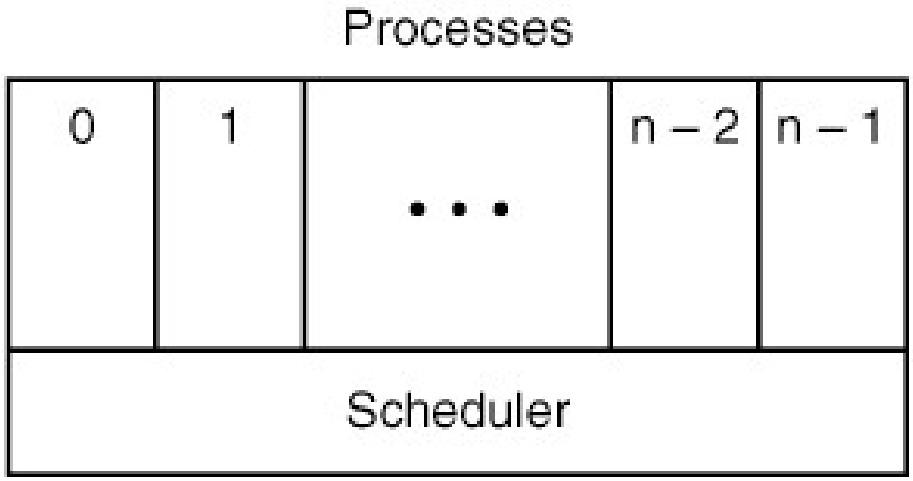


Figure 2-3 The lowest layer of a process-structured operating system handles interrupts and scheduling. Above that layer are sequential processes.

Implementation of Processes

Process management	Memory management	File management
Registers	Pointer to text segment	UMASK mask
Program counter	Pointer to data segment	Root directory
Program status word	Pointer to bss segment	Working directory
Stack pointer	Exit status	File descriptors
Process state	Signal status	Effective uid
Time when process started	Process id	Effective gid
CPU time used	Parent process	System call parameters
Children's CPU time	Process group	Various flag bits
Time of next alarm	Real uid	
Message queue pointers	Effective uid	
Pending signal bits	Real gid	
Process id	Effective gid	
Various flag bits	Bit maps for signals	
	Various flag bits	

Figure 2-4. Some of the fields of the MINIX 3 process table. The fields are distributed over the kernel, the process manager, and the file system.

Interrupts

1. Hardware stacks program counter, etc.
2. Hardware loads new program counter from interrupt vector.
3. Assembly language procedure saves registers.
4. Assembly language procedure sets up new stack.
5. C interrupt service runs (typically reads and buffers input).
6. Scheduler marks waiting task as ready.
7. Scheduler decides which process is to run next.
8. C procedure returns to the assembly code.
9. Assembly language procedure starts up new current process.

Figure 2-5 Skeleton of what the lowest level of the operating system does when an interrupt occurs.

Threads (1)

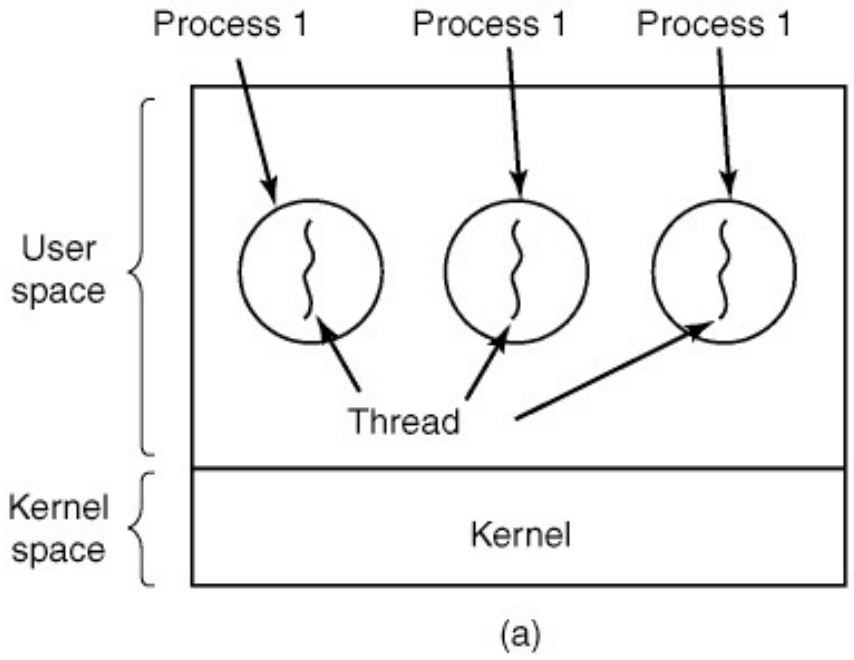


Figure 2-6 (a) Three processes each with one thread.

Threads (2)

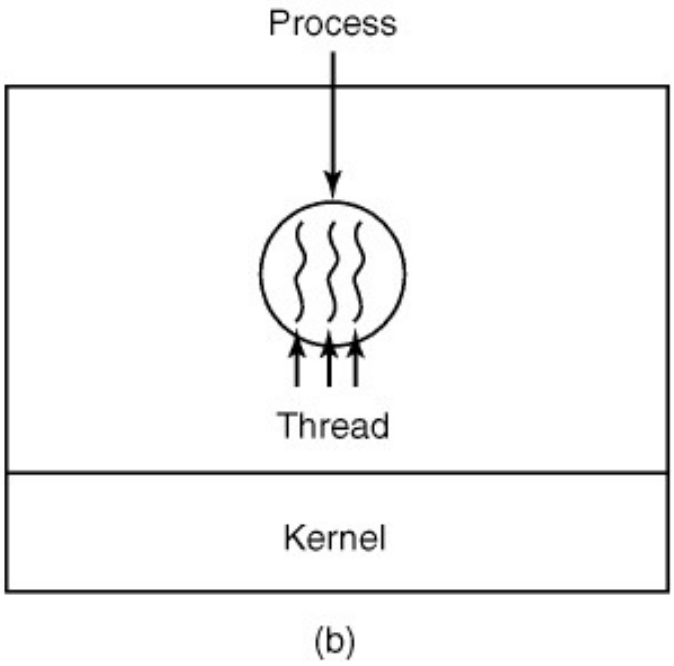


Figure 2-6 (b) One process with three threads.

Threads (3)

Per process items	Per thread items
Address space	Program counter
Global variables	Registers
Open files	Stack
Child processes	State
Pending alarms	
Signals and signal handlers	
Accounting information	

Figure 2-7. The first column lists some items shared by all threads in a process. The second one lists some items private to each thread.

Race Conditions

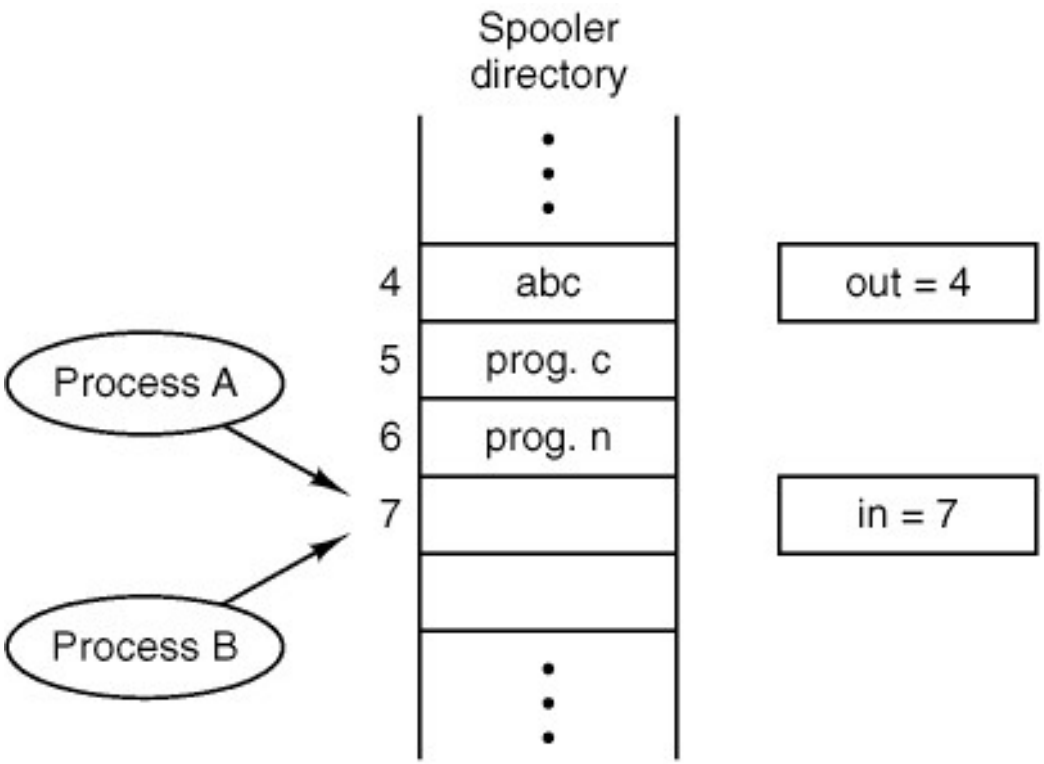


Figure 2-8 Two processes want to access shared memory at the same time.

Critical Sections

Necessary to avoid race conditions:

1. No two processes may be simultaneously inside their critical regions.
2. No assumptions may be made about speeds or the number of CPUs.
3. No process running outside its critical region may block other processes.
4. No process should have to wait forever to enter its critical region.

Mutual Exclusion with Busy Waiting

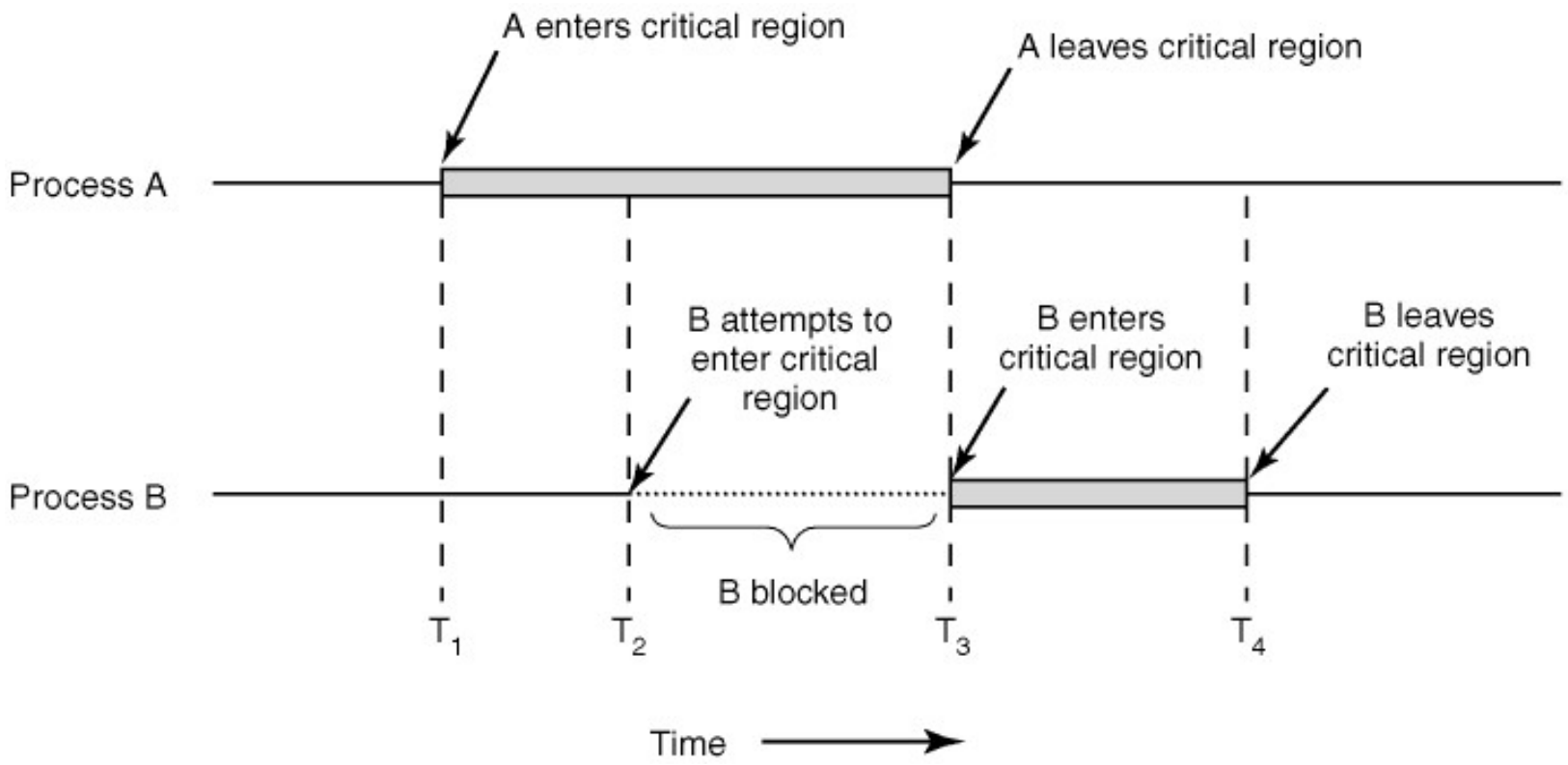


Figure 2-9 Mutual exclusion using critical regions.

Strict Alternation

```
while (TRUE) {
    while (turn != 0)        /* loop */ ;
    critical_region( );
    turn = 1;
    noncritical_region( );
}
```

(a)

```
while (TRUE) {
    while (turn != 1)        /* loop */ ;
    critical_region( );
    turn = 0;
    noncritical_region( );
}
```

(b)

Figure 2-10. A proposed solution to the critical region problem. (a) Process 0. (b) Process 1. In both cases, be sure to note the semicolons terminating the while statements.

Peterson's Solution (1)

```
#define FALSE  0
#define TRUE   1
#define N      2                        /* number of processes */

int turn;                               /* whose turn is it? */
int interested[N];                      /* all values initially 0 (FALSE) */

void enter_region(int process)          /* process is 0 or 1 */
{ . . .
```

Figure 2-11 Peterson's solution for achieving mutual exclusion.

Peterson's Solution (2)

```
void enter_region(int process)          /* process is 0 or 1 */
{
     int other;                         /* number of the other process */

     other = 1 – process;               /* the opposite of process */
     interested[process] = TRUE;        /* show that you are interested */
     turn = process;                    /* set flag */
     while (turn == process && interested[other] == TRUE) /* null statement */ ;
}

void leave_region(int process)          /* process: who is leaving */
{
     interested[process] = FALSE;       /* indicate departure from critical region */
}
```

Figure 2-11 Peterson's solution for achieving mutual exclusion.

The TSL Instruction

```
enter_region:
        tsl register,lock       | copy lock to register and set lock to 1
        cmp register,#0         | was lock zero?
        jne enter_region        | if it was non zero, lock was set, so loop
        ret                     | return to caller; critical region entered

leave_region:
        move lock,#0            | store a 0 in lock
        ret                     | return to caller
```

Figure 2-12. Entering and leaving a critical region using the TSL instruction.

The Producer-Consumer Problem (1)

```
#define N 100                                   /* number of slots in the buffer */
int count = 0;                                  /* number of items in the buffer */

void producer(void)
{
     int item;

     while (TRUE) {                             /* repeat forever */
          item = produce_item( );               /* generate next item */
          if (count == N) sleep( );             /* if buffer is full, go to sleep */
          insert_item(item);                    /* put item in buffer */
          count = count + 1;                    /* increment count of items in buffer */
          if (count == 1) wakeup(consumer);     /* was buffer empty? */
     }
}
     . . .
```

Figure 2-13. The producer-consumer problem with a fatal race condition.

The Producer-Consumer Problem (2)

```
    . . .
void consumer(void)
{
      int item;

      while (TRUE) {                                  /* repeat forever */
            if (count == 0) sleep( );                 /* if buffer is empty, got to sleep */
            item = remove_item( );                    /* take item out of buffer */
            count = count – 1;                        /* decrement count of items in buffer */
            if (count == N – 1) wakeup(producer); /* was buffer full? */
            consume_item(item);                       /* print item */
      }
}
```

Figure 2-13. The producer-consumer problem with a fatal race condition

The Producer-Consumer Problem (3)

```
#define N 100                                   /* number of slots in the buffer */
typedef int semaphore;                          /* semaphores are a special kind of int */
semaphore mutex = 1;                            /* controls access to critical region */
semaphore empty = N;                            /* counts empty buffer slots */
semaphore full = 0;                             /* counts full buffer slots */

void producer(void)
{
     int item;

     while (TRUE) {                             /* TRUE is the constant 1 */
          item = produce_item( );               /* generate something to put in buffer */
          down(&empty);                         /* decrement empty count */
          down(&mutex);                         /* enter critical region */
          insert_item(item);                    /* put new item in buffer */
          up(&mutex);                           /* leave critical region */
          up(&full);                            /* increment count of full slots */
     }
}
. . .
```

Figure 2-14. The producer-consumer problem using semaphores.

The Producer-Consumer Problem (4)

```
. . .

void consumer(void)
{
     int item;

     while (TRUE) {                          /* infinite loop */
          down(&full);                       /* decrement full count */
          down(&mutex);                      /* enter critical region */
          item = remove_item( );             /* take item from buffer */
          up(&mutex);                        /* leave critical region */
          up(&empty);                        /* increment count of empty slots */
          consume_item(item);                /* do something with the item */
     }
}
```

Figure 2-14. The producer-consumer problem using semaphores.

Monitors (1)

```
monitor example
  integer i;
  condition c;

  procedure producer(x);
  .
  .
  .
  end;

  procedure consumer(x);
  .
  .
  .
  end;
end monitor;
```

Figure 2-15. A monitor.

Monitors (2)

```
monitor ProducerConsumer
    condition full, empty;
    integer count;

    procedure insert(item: integer);
    begin
        if count = N then wait(full);
        insert_item(item);
        count := count + 1;
        if count = 1 then signal(empty)
    end;

    function remove: integer;
    begin
        if count = 0 then wait(empty);
        remove = remove_item;
        count := count - 1;
        if count = N - 1 then signal(full)
    end;

    count := 0;
end monitor;
```

Figure 2-16. An outline of the producer-consumer problem with monitors. Only one monitor procedure at a time is active. The buffer has N slots

Monitors (3)

```
procedure producer;
begin
        while true do
        begin
                item = produce_item;
                ProducerConsumer.insert(item)
        end
end;

procedure consumer;
begin
        while true do
        begin
                item = ProducerConsumer.remove;
                consume_item(item)
        end
end;
```

Figure 2-16. An outline of the producer-consumer problem with monitors. Only one monitor procedure at a time is active. The buffer has N slots

Message Passing (1)

```
#define N 100                                   /* number of slots in the buffer */

void producer(void)
{
     int item;
     message m;                                 /* message buffer */

     while (TRUE) {
          item = produce_item( );               /* generate something to put in buffer */
          receive(consumer, &m);                /* wait for an empty to arrive */
          build_message(&m, item);              /* construct a message to send */
          send(consumer, &m);                   /* send item to consumer */
     }
}
          . . .
```

Figure 2-17. The producer-consumer problem with N messages.

Message Passing (2)

```
. . .

void consumer(void)
{
      int item, i;
      message m;

      for (i = 0; i < N; i++) send(producer, &m);  /* send N empties */
      while (TRUE) {
            receive(producer, &m);          /* get message containing item */
            item = extract_item(&m);        /* extract item from message */
            send(producer, &m);             /* send back empty reply */
            consume_item(item);             /* do something with the item */
      }
}
```

Figure 2-17. The producer-consumer problem with N messages.

The Dining Philosophers Problem (1)

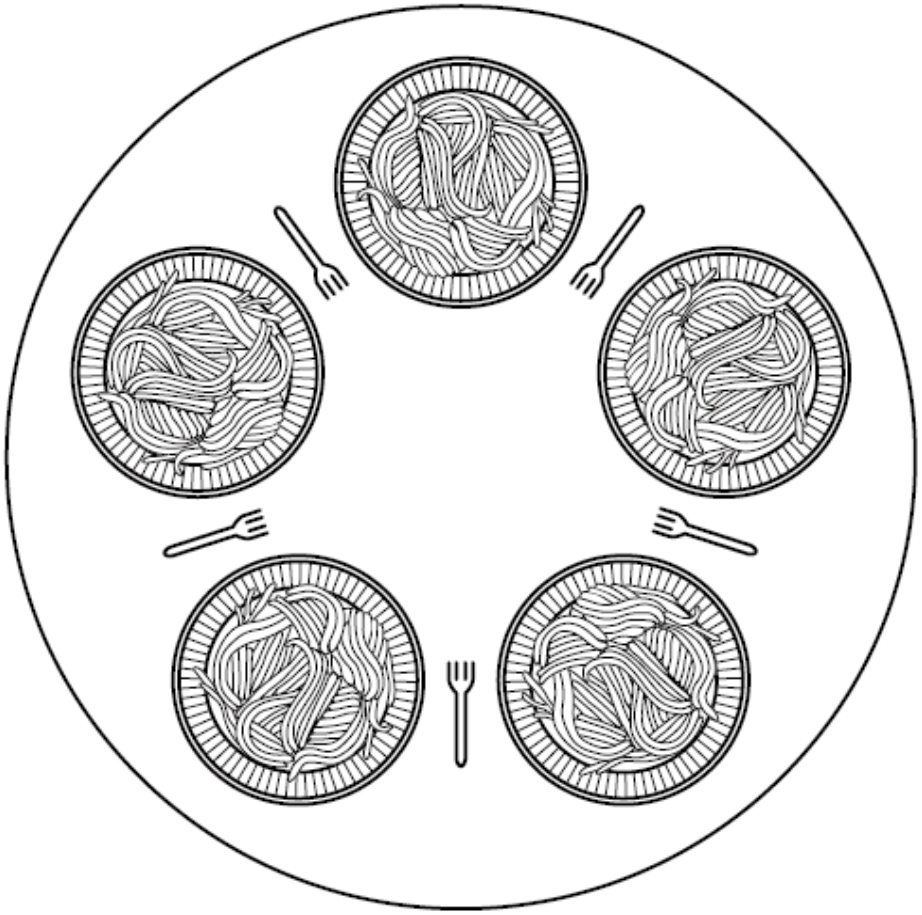


Figure 2-18. Lunch time in the Philosophy Department.

The Dining Philosophers Problem (2)

```
#define N 5                                     /* number of philosophers */

void philosopher(int i)                         /* i: philosopher number, from 0 to 4 */
{
      while (TRUE) {
            think( );                           /* philosopher is thinking */
            take_fork(i);                       /* take left fork */
            take_fork((i+1) % N);               /* take right fork; % is modulo operator */
            eat( );                             /* yum-yum, spaghetti */
            put_fork(i);                        /* put left fork back on the table */
            put_fork((i+1) % N);                /* put right fork back on the table */
      }
}
```

Figure 2-19. A nonsolution to the dining philosophers problem.

The Dining Philosophers Problem (3)

```
#define N                5                  /* number of philosophers */
#define LEFT             (i+N–1)%N          /* number of i's left neighbor */
#define RIGHT            (i+1)%N            /* number of i's right neighbor */
#define THINKING         0                  /* philosopher is thinking */
#define HUNGRY           1                  /* philosopher is trying to get forks */
#define EATING           2                  /* philosopher is eating */
typedef int semaphore;                      /* semaphores are a special kind of int */
int state[N];                               /* array to keep track of everyone's state */
semaphore mutex = 1;                        /* mutual exclusion for critical regions */
semaphore s[N];                             /* one semaphore per philosopher */

    . . .
```

Figure 2-20. A solution to the dining philosophers problem.

The Dining Philosophers Problem (4)

```
. . .
void philosopher(int i)                    /* i: philosopher number, from 0 to N–1 */
{
     while (TRUE) {                        /* repeat forever */
          think( );                        /* philosopher is thinking */
          take_forks(i);                   /* acquire two forks or block */
          eat( );                          /* yum-yum, spaghetti */
          put_forks(i);                    /* put both forks back on table */
     }
}

void take_forks(int i)                     /* i: philosopher number, from 0 to N–1 */
{
     down(&mutex);                         /* enter critical region */
     state[i] = HUNGRY;                    /* record fact that philosopher i is hungry */
     test(i);                              /* try to acquire 2 forks */
     up(&mutex);                           /* exit critical region */
     down(&s[i]);                          /* block if forks were not acquired */
}
     . . .
```

Figure 2-20. A solution to the dining philosophers problem.

The Dining Philosophers Problem (5)

```
. . .

void put_forks(i)                       /* i: philosopher number, from 0 to N–1 */
{
        down(&mutex);                   /* enter critical region */
        state[i] = THINKING;            /* philosopher has finished eating */
        test(LEFT);                     /* see if left neighbor can now eat */
        test(RIGHT);                    /* see if right neighbor can now eat */
        up(&mutex);                     /* exit critical region */
}

void test(i)                            /* i: philosopher number, from 0 to N–1 */
{
        if (state[i] == HUNGRY && state[LEFT] != EATING && state[RIGHT] != EATING) {
                state[i] = EATING;
                up(&s[i]);
        }
}
```

Figure 2-20. A solution to the dining philosophers problem.

The Readers and Writers Problem (1)

```
typedef int semaphore;                  /* use your imagination */
semaphore mutex = 1;                    /* controls access to 'rc' */
semaphore db = 1;                       /* controls access to the database */
int rc = 0;                             /* # of processes reading or wanting to */

void reader(void)
{
     while (TRUE) {                     /* repeat forever */
          down(&mutex);                 /* get exclusive access to 'rc' */
          rc = rc + 1;                  /* one reader more now */
          if (rc == 1) down(&db);       /* if this is the first reader ... */
          up(&mutex);                   /* release exclusive access to 'rc' */
          read_data_base( );            /* access the data */
          down(&mutex);                 /* get exclusive access to 'rc' */
          rc = rc – 1;                  /* one reader fewer now */
          if (rc == 0) up(&db);         /* if this is the last reader ... */
          up(&mutex);                   /* release exclusive access to 'rc' */
          use_data_read( );             /* noncritical region */
     }
}
     . . .
```

Figure 2-21. A solution to the readers and writers problem.

The Readers and Writers Problem (2)

```
. . .

     void writer(void)
     {
          while (TRUE) {                    /* repeat forever */
               think_up_data( );            /* noncritical region */
               down(&db);                   /* get exclusive access */
               write_data_base( );          /* update the data */
               up(&db);                     /* release exclusive access */
          }
     }
```

Figure 2-21. A solution to the readers and writers problem.

Process Behavior (1)

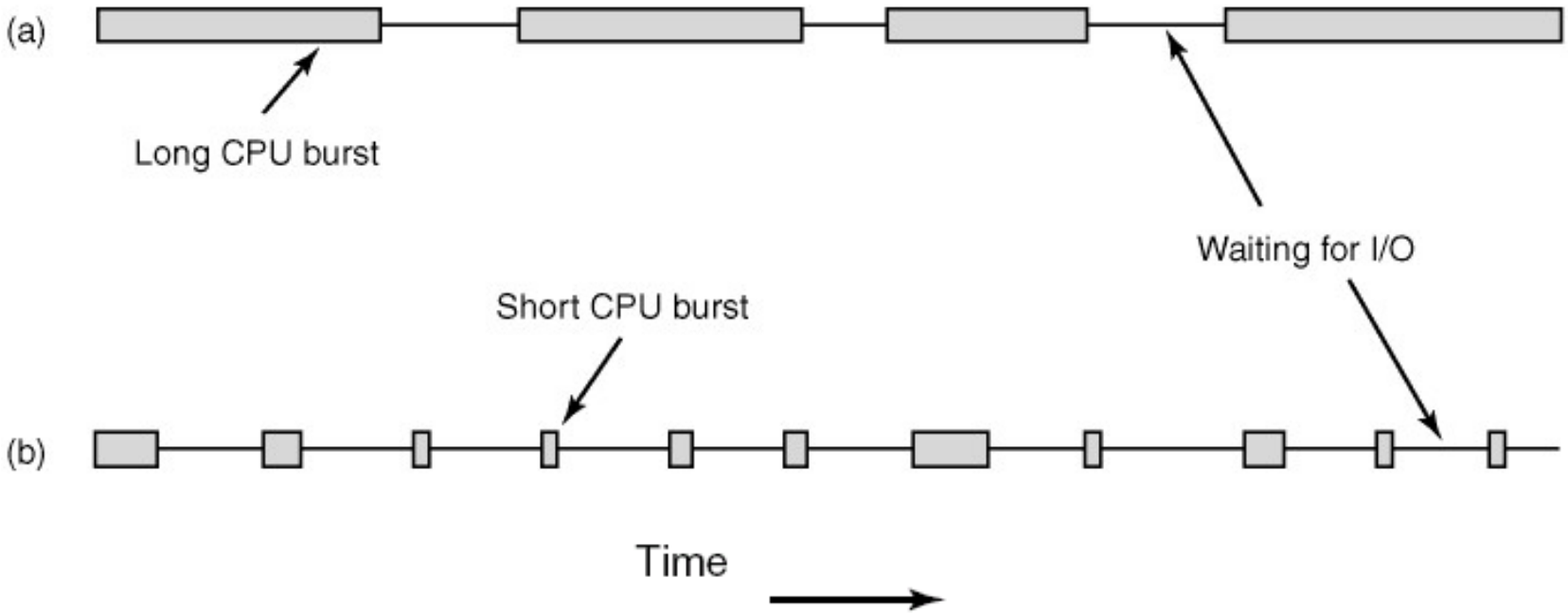


Figure 2-22. Bursts of CPU usage alternate with periods of waiting for I/O. (a) A CPU-bound process. (b) An I/O-bound process.

When to Schedule

When scheduling is absolutely required:

1. When a process exits.
2. When a process blocks on I/O, or a semaphore.

When scheduling usually done (though not absolutely required)

1. When a new process is created.
2. When an I/O interrupt occurs.
3. When a clock interrupt occurs.

Scheduling Algorithms (2)

All systems

Fairness — giving each process a fair share of the CPU
Policy enforcement — seeing that stated policy is carried out
Balance — keeping all parts of the system busy

Batch systems

Throughput — maximize jobs per hour
Turnaround time — minimize time between submission and termination
CPU utilization — keep the CPU busy all the time

Interactive systems

Response time — respond to requests quickly
Proportionality — meet users' expectations

Real—time systems

Meeting deadlines — avoid losing data
Predictability — avoid quality degradation in multimedia systems

Figure 2-23. Some goals of the scheduling algorithm under different circumstances.

Scheduling Algorithms (2)

Figure 2-24. An example of shortest job first scheduling.
(a) Running four jobs in the original order.
(b) Running them in shortest job first order.

Three Level Scheduling (1)

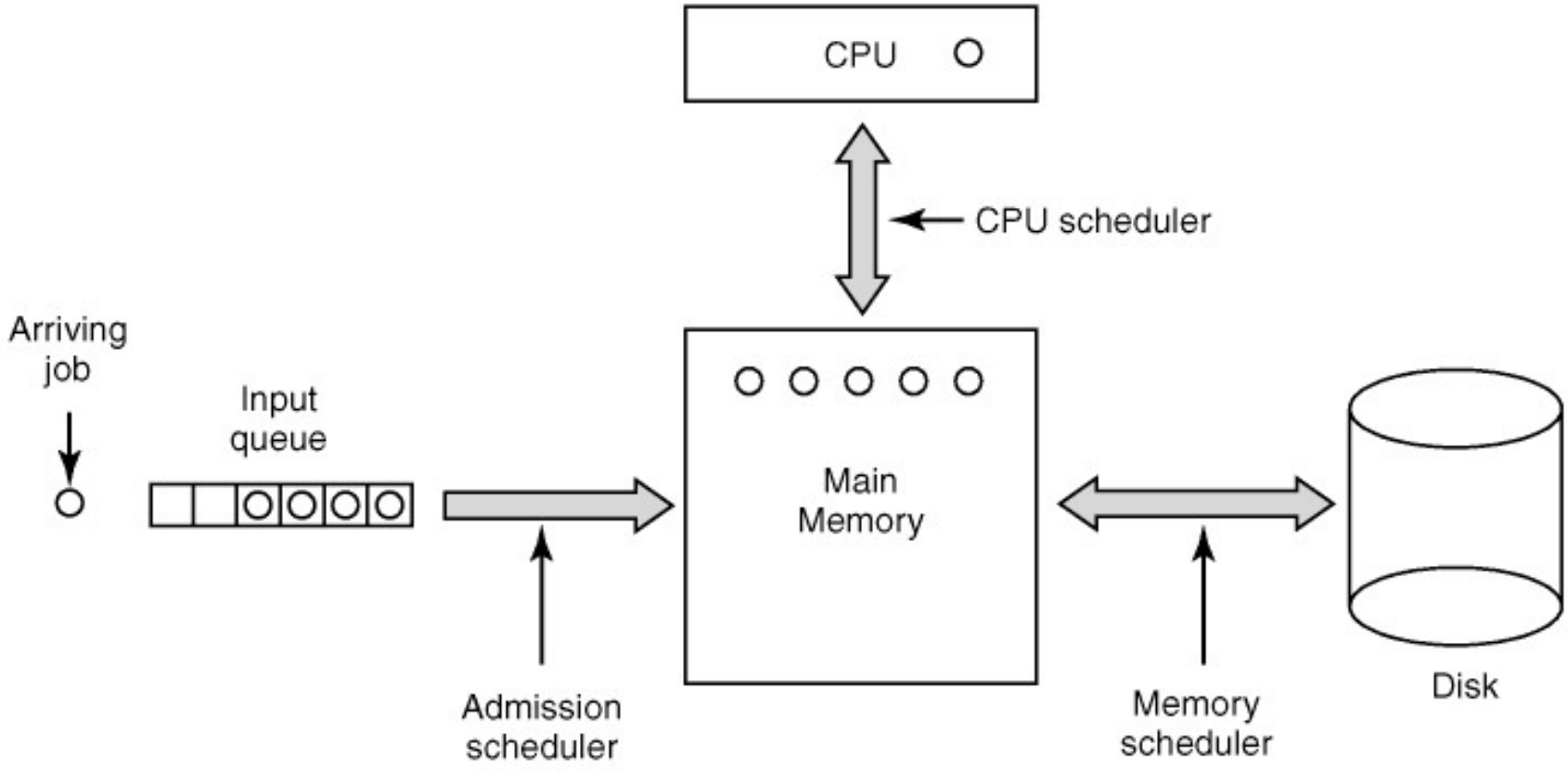


Figure 2-25. Three-level scheduling.

Three Level Scheduling (2)

Criteria for deciding which process to choose:

- How long has it been since the process was swapped in or out?
- How much CPU time has the process had recently?
- How big is the process? (Small ones do not get in the way.)
- How important is the process?

Round-Robin Scheduling

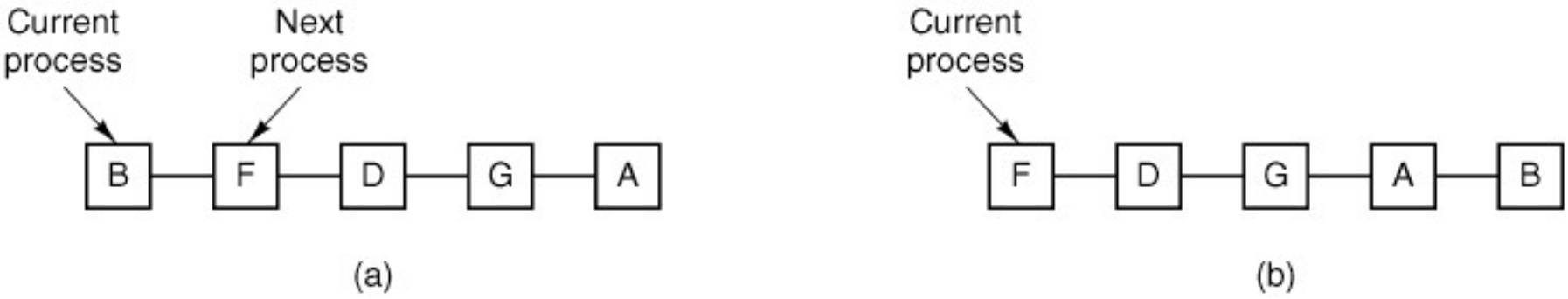


Figure 2-26. Round-robin scheduling.

(a) The list of runnable processes.

(b) The list of runnable processes after B uses up its quantum.

Priority Scheduling

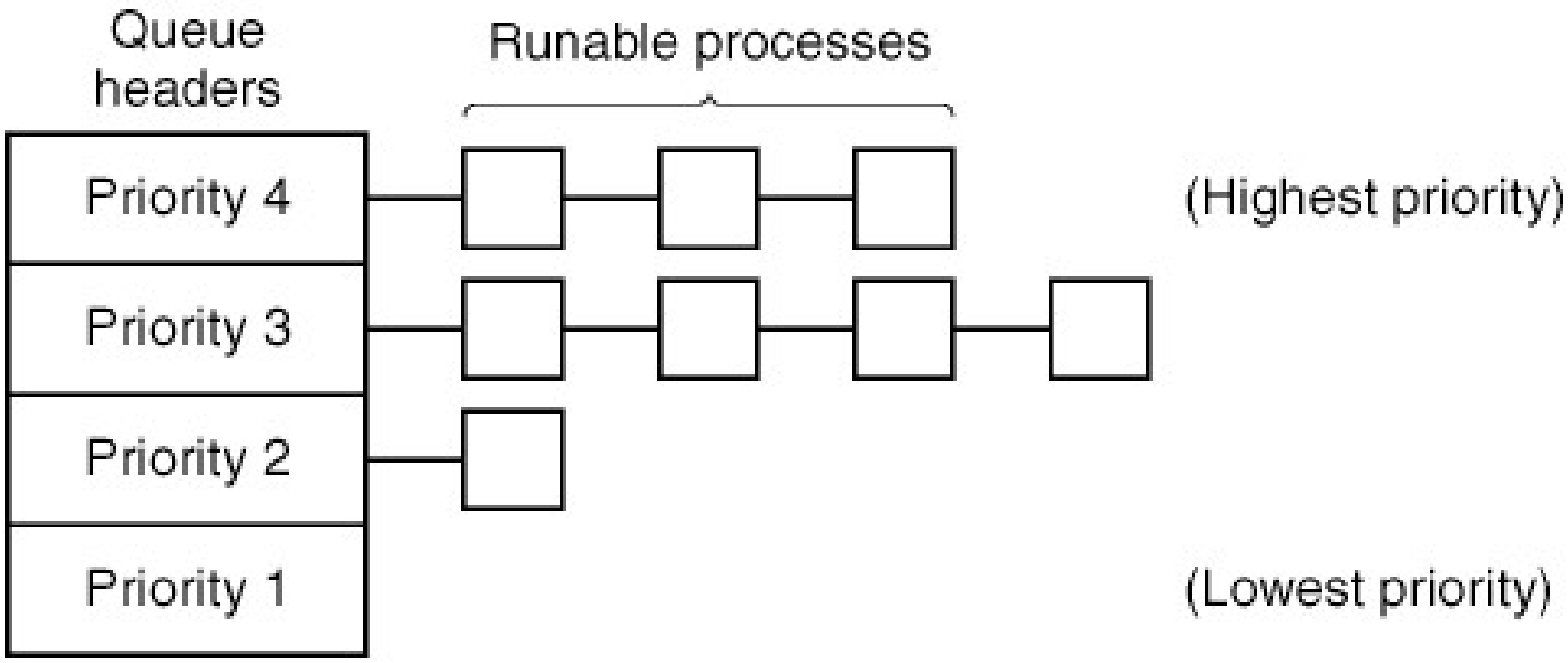
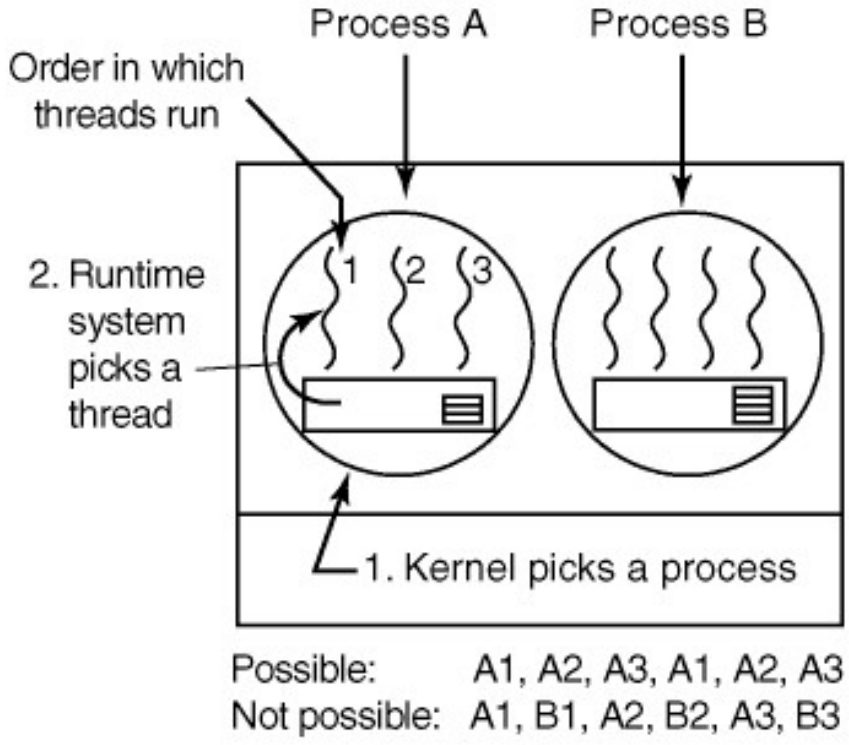


Figure 2-27. A scheduling algorithm with four priority classes.

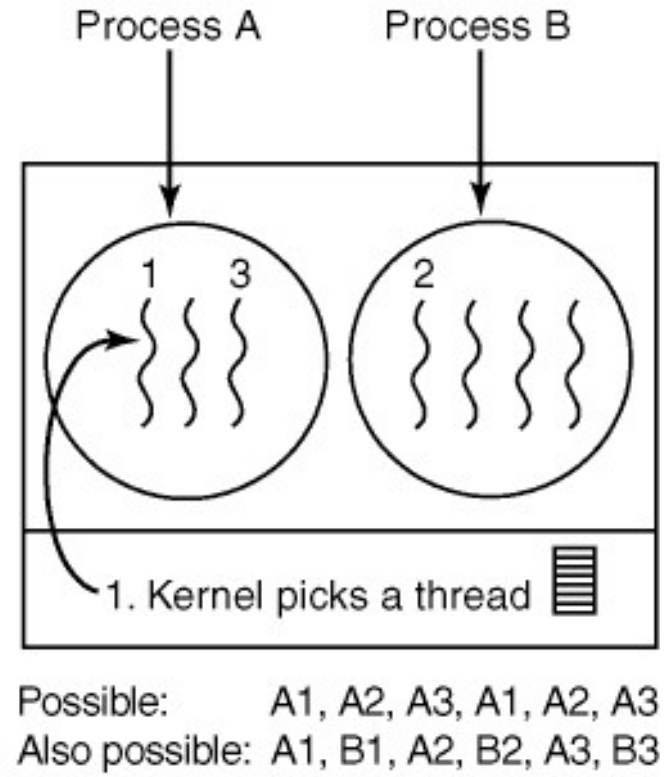
Thread Scheduling (1)



(a)

Figure 2-28. (a) Possible scheduling of user-level threads with a 50-msec process quantum and threads that run 5 msec per CPU burst.

Thread Scheduling (2)



(b)

Figure 2-28. (b) Possible scheduling of kernel-level threads with the same characteristics as (a).

The Internal Structure of MINIX

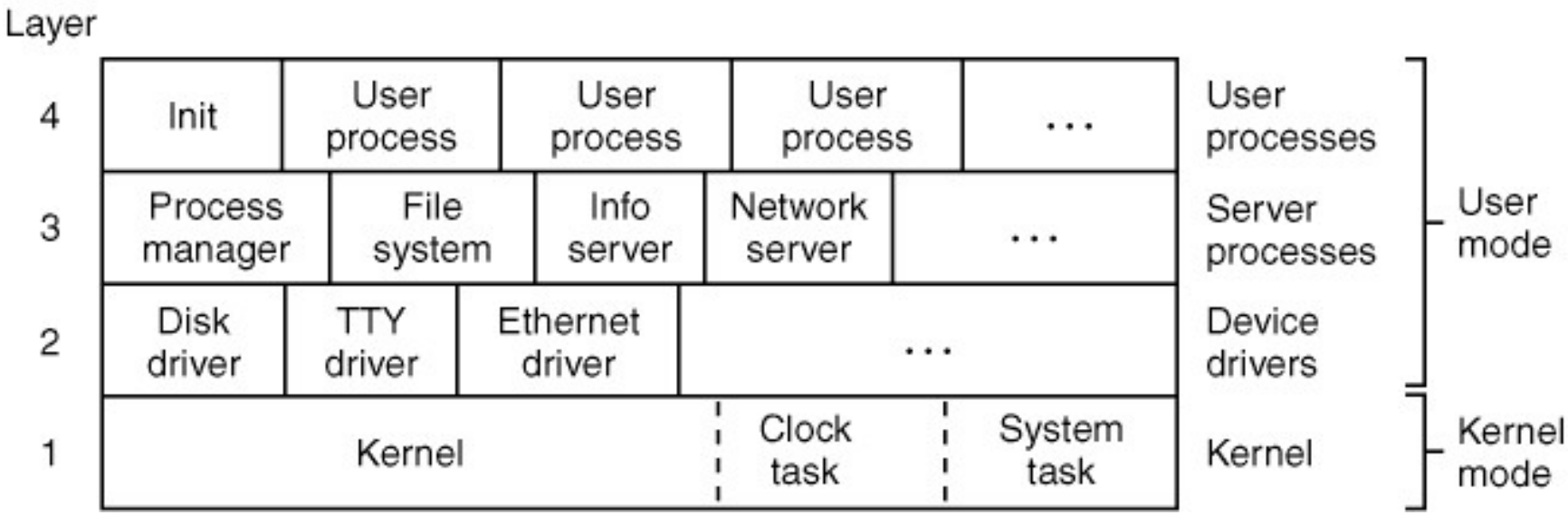


Figure 2-29. MINIX 3 is structured in four layers. Only processes in the bottom layer may use privileged (kernel mode) instructions.

MINIX 3 Startup

Component	Description	Loaded by
kernel	Kernel + clock and system tasks	(in boot image)
pm	Process manager	(in boot image)
fs	File system	(in boot image)
rs	(Re)starts servers and drivers	(in boot image)
memory	RAM disk driver	(in boot image)
log	Buffers log output	(in boot image)
tty	Console and keyboard driver	(in boot image)
driver	Disk (at, bios, or floppy) driver	(in boot image)
init	parent of all user processes	(in boot image)
floppy	Floppy driver (if booted from hard disk)	/etc/rc
is	Information server (for debug dumps)	/etc/rc
cmos	Reads CMOS clock to set time	/etc/rc
random	Random number generator	/etc/rc
printer	Printer driver	/etc/rc

Figure 2-30. Some important MINIX 3 system components.

Others such as an Ethernet driver and the inet server may also be present.

MINIX Memory (1)

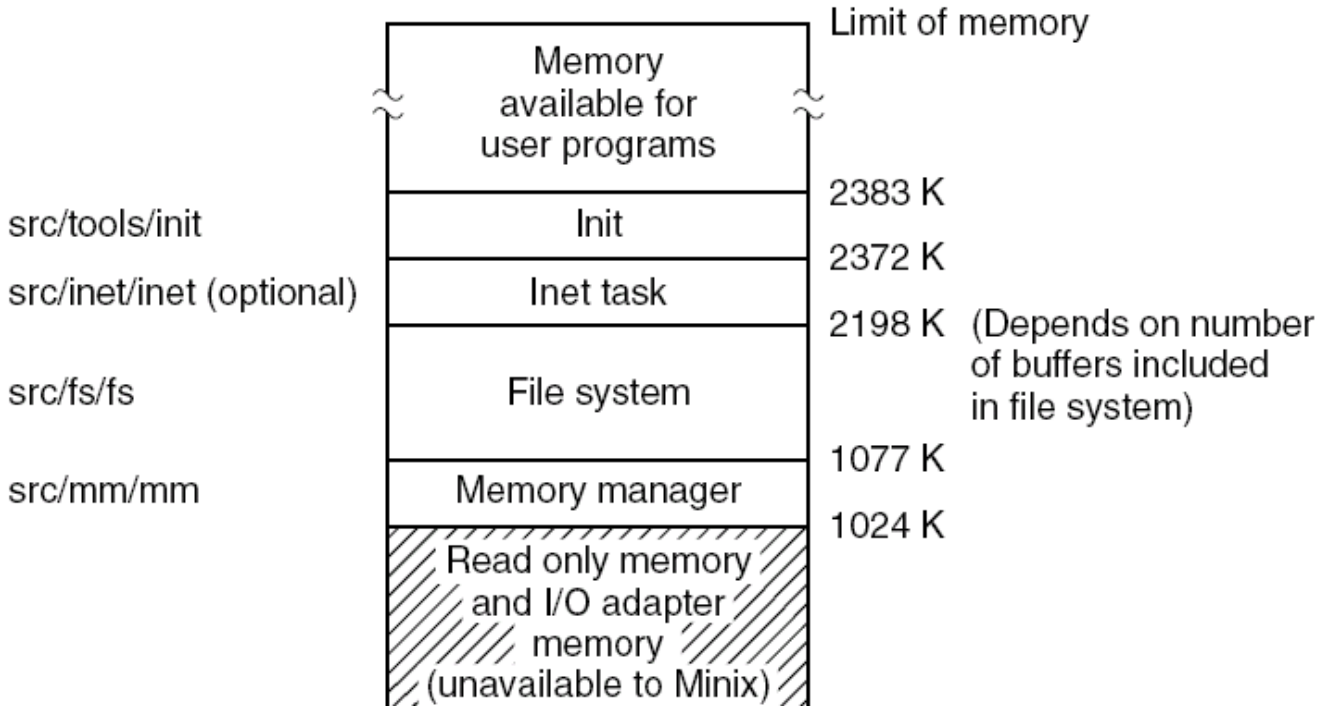


Figure 2-31. Memory layout after MINIX 3 has been loaded from the disk into memory. The kernel, servers, and drivers are independently compiled and linked programs, listed on the left. Sizes are approximate and not to scale.

MINIX Memory (2)

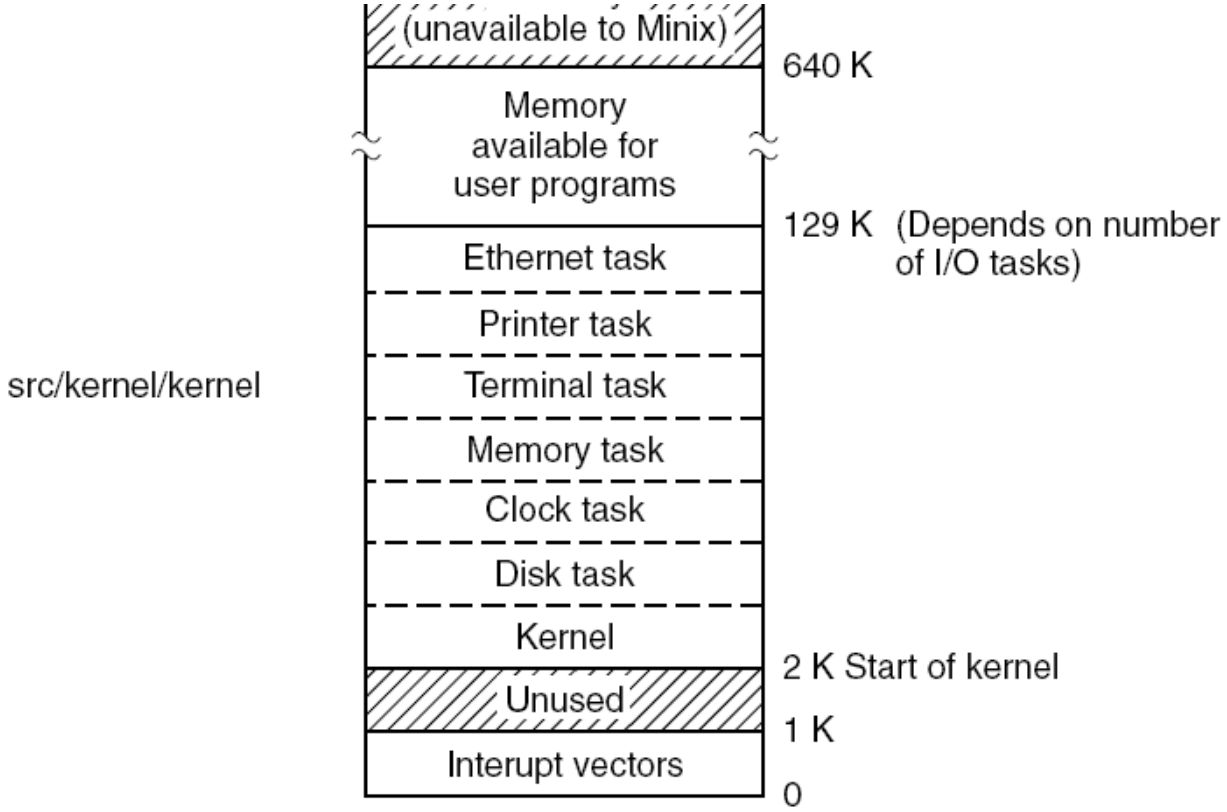


Figure 2-31. Memory layout after MINIX 3 has been loaded from the disk into memory. The kernel, servers, and drivers are independently compiled and linked programs, listed on the left. Sizes are approximate and not to scale.

MINIX Header File

```
#include <minix/config.h>      /* MUST be first */
#include <ansi.h>              /* MUST be second */
#include <sys/types.h>
#include <minix/const.h>
#include <minix/type.h>
#include <limits.h>
#include <errno.h>
#include <minix/syslib.h>
#include "const.h"
```

Figure 2-32. Part of a master header which ensures inclusion of header files needed by all C source files. Note that two *const.h* files, one from the *include/* tree and one from the local directory, are referenced.

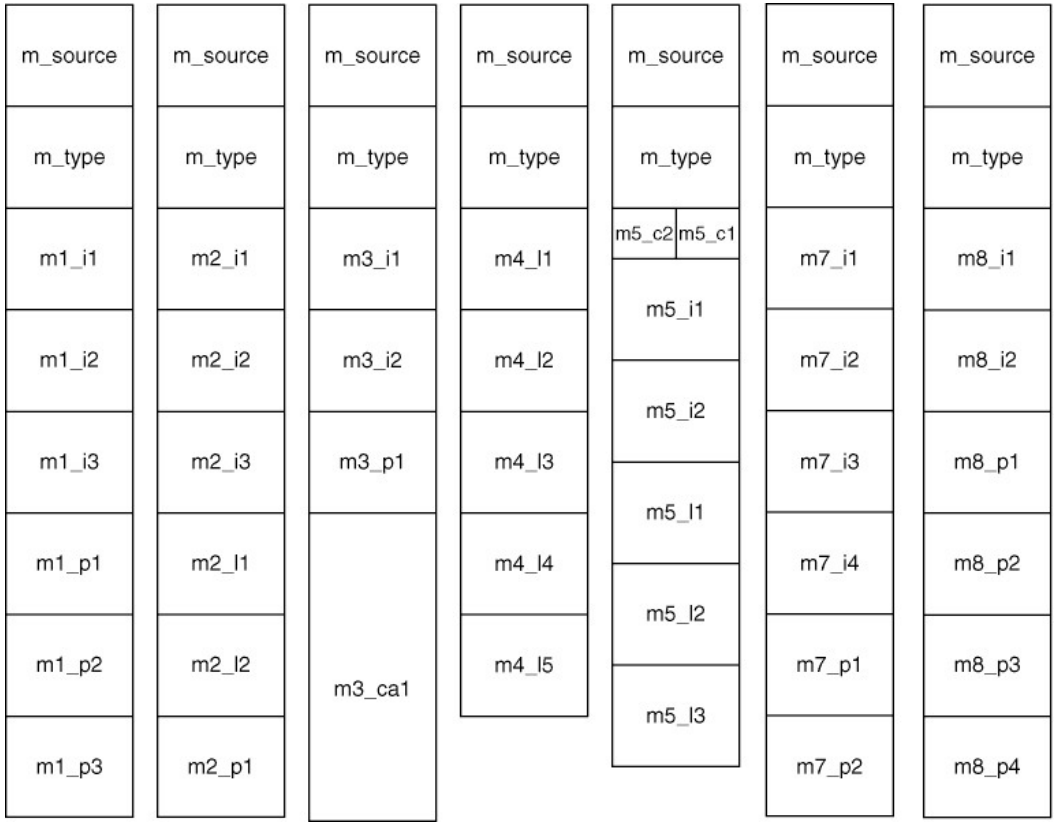
Sizes of Types in MINIX

Type	16-Bit MINIX	32-Bit MINIX
gid_t	8	8
dev_t	16	16
pid_t	16	32
ino_t	16	32

Figure 2-33. The size, in bits, of some types on 16-bit and 32-bit systems.

MINIX Message Types

Figure 2-34. The seven message types used in MINIX 3. The sizes of message elements will vary, depending upon the architecture of the machine; this diagram illustrates sizes on CPUs with 32-bit pointers, such as those of Pentium family members.



Debug Dump

--nr-	-id-	-name-	-flags-	-traps-	-ipc_to mask------
(-4)	(01)	IDLE	P-BS-	-----	00000000 00001111
[-3]	(02)	CLOCK	---S-	--R--	00000000 00001111
[-2]	(03)	SYSTEM	---S-	--R--	00000000 00001111
[-1]	(04)	KERNEL	---S-	-----	00000000 00001111
0	(05)	pm	P--S-	ESRBN	11111111 11111111
1	(06)	fs	P--S-	ESRBN	11111111 11111111
2	(07)	rs	P--S-	ESRBN	11111111 11111111
3	(09)	memory	P--S-	ESRBN	00110111 01101111
4	(10)	log	P--S-	ESRBN	11111111 11111111
5	(08)	tty	P--S-	ESRBN	11111111 11111111
6	(11)	driver	P--S-	ESRBN	11111111 11111111
7	(00)	init	P-B--	E--B-	00000111 00000000

Figure 2-35. Part of a debug dump of the privilege table. The clock task, file server, tty, and init processes privileges are typical of tasks, servers, device drivers, and user processes, respectively. The bitmap is truncated to 16 bits.

Bootstrapping MINIX (1)

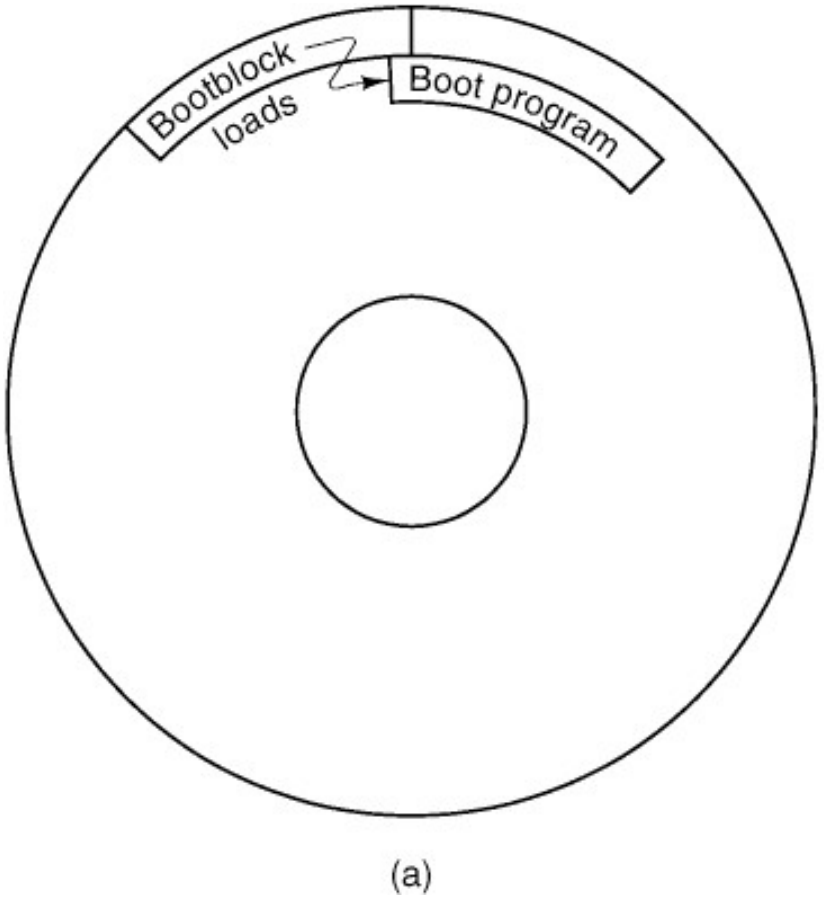


Figure 2-36. Disk structures used for bootstrapping.
(a) Unpartitioned disk. The first sector is the bootblock.

Bootstrapping MINIX (2)

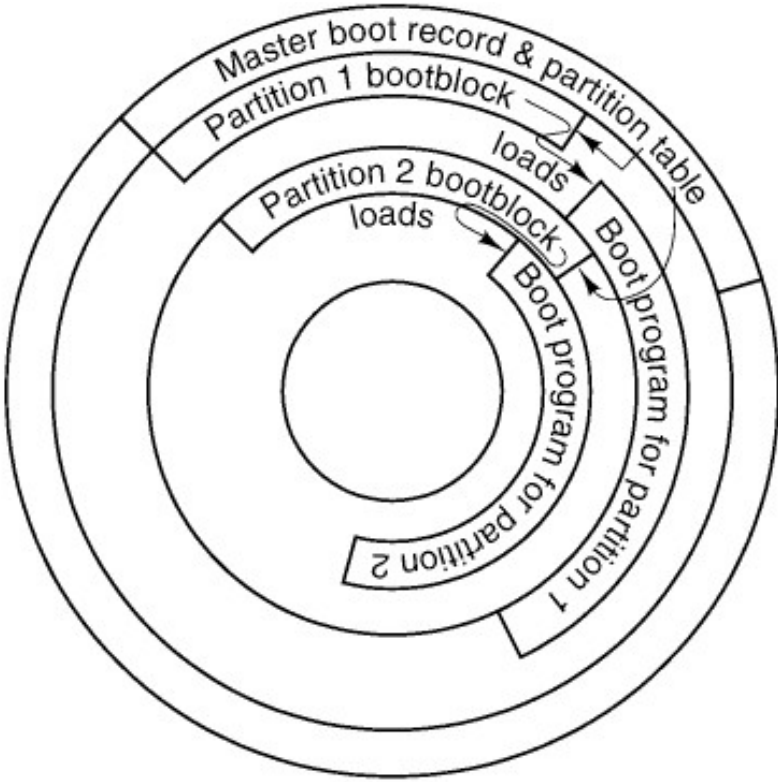


Figure 2-36. Disk structures used for bootstrapping. (b) Partitioned disk. The first sector is the master boot record, also called masterboot.

Boot Time in MINIX

```
rootdev=256
ramimagedev=916
ramsize=4096
processor=586
bus=at
video=vga
chrome=color
memory=800:92880,100000:2F00000
c0=at
image=/minix/2.0.3r5
```

Figure 2-37. Boot parameters passed to the kernel at boot time in a typical MINIX 3 system.

System Initialization in MINIX

```
#include <minix/config.h>
#if _WORD_SIZE == 2
#include "mpx88.s"
#else
#include "mpx386.s"
#endif
```

Figure 2-38. How alternative assembly language source files are selected.

Interrupt Handling in MINIX (1)

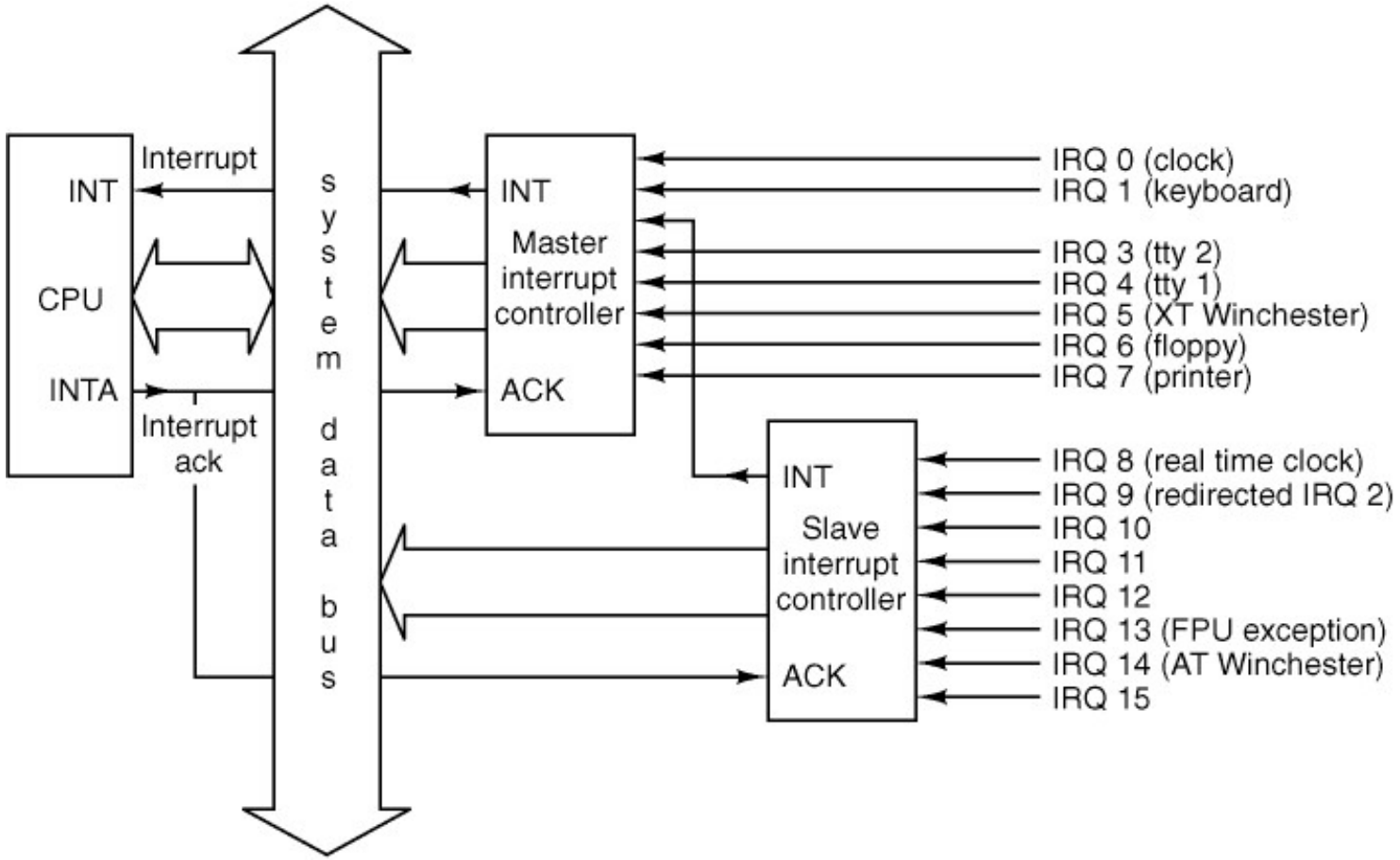
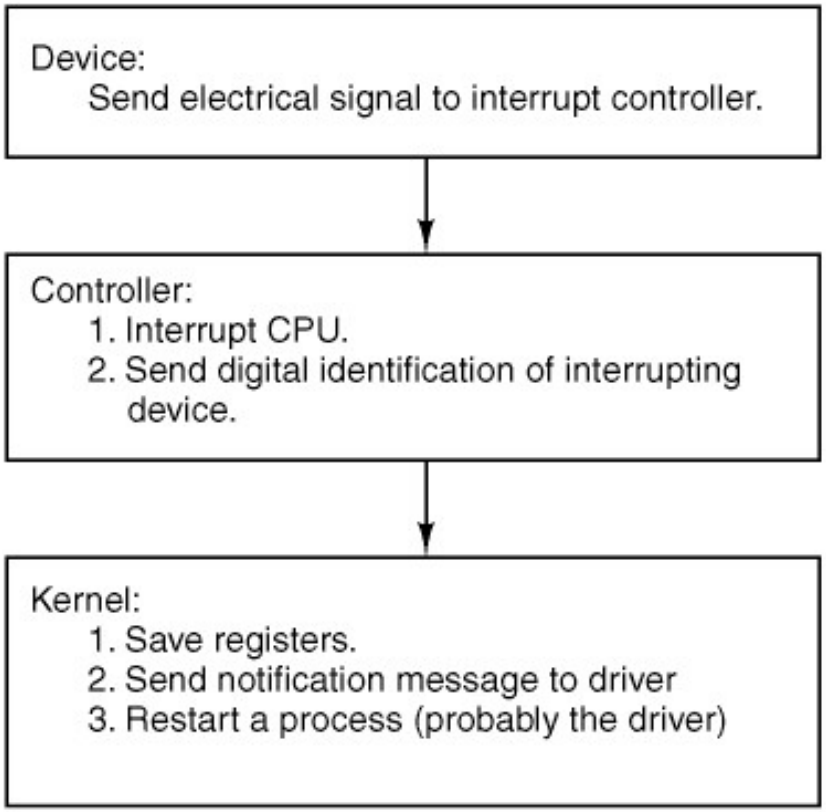


Figure 2-39. Interrupt processing hardware on a 32-bit Intel PC.

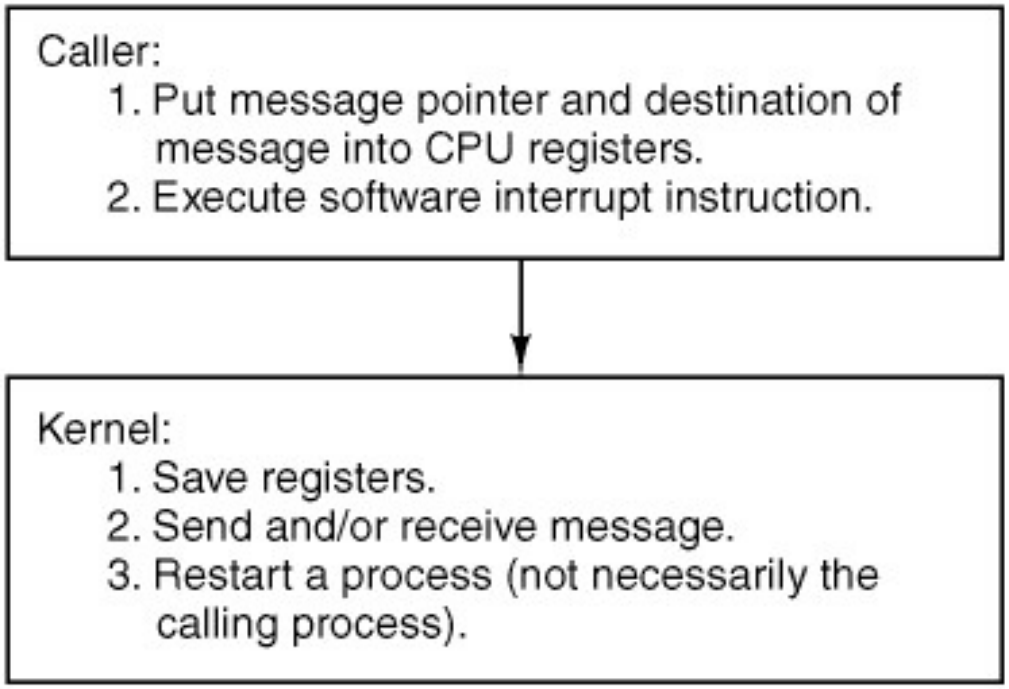
Interrupt Handling in MINIX (2)



(a)

Figure 2-40. (a) How a hardware interrupt is processed.

Interrupt Handling in MINIX (3)



(b)

Figure 2-40. (b) How a system call is made.

Restart

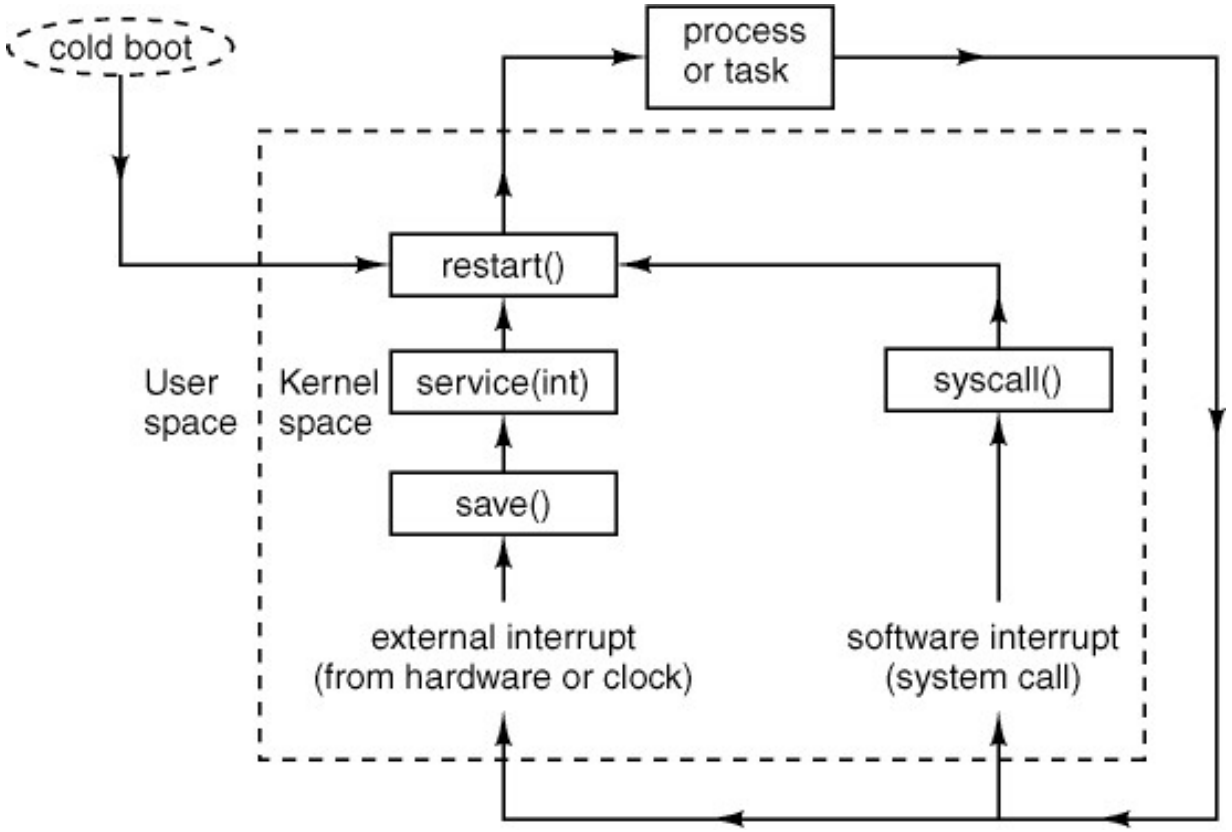


Figure 2-41. Restart is the common point reached after system startup, interrupts, or system calls. The most deserving process (which may be and often is a different process from the last one interrupted) runs next. Not shown in this diagram are interrupts that occur while the kernel itself is running.

Queueing

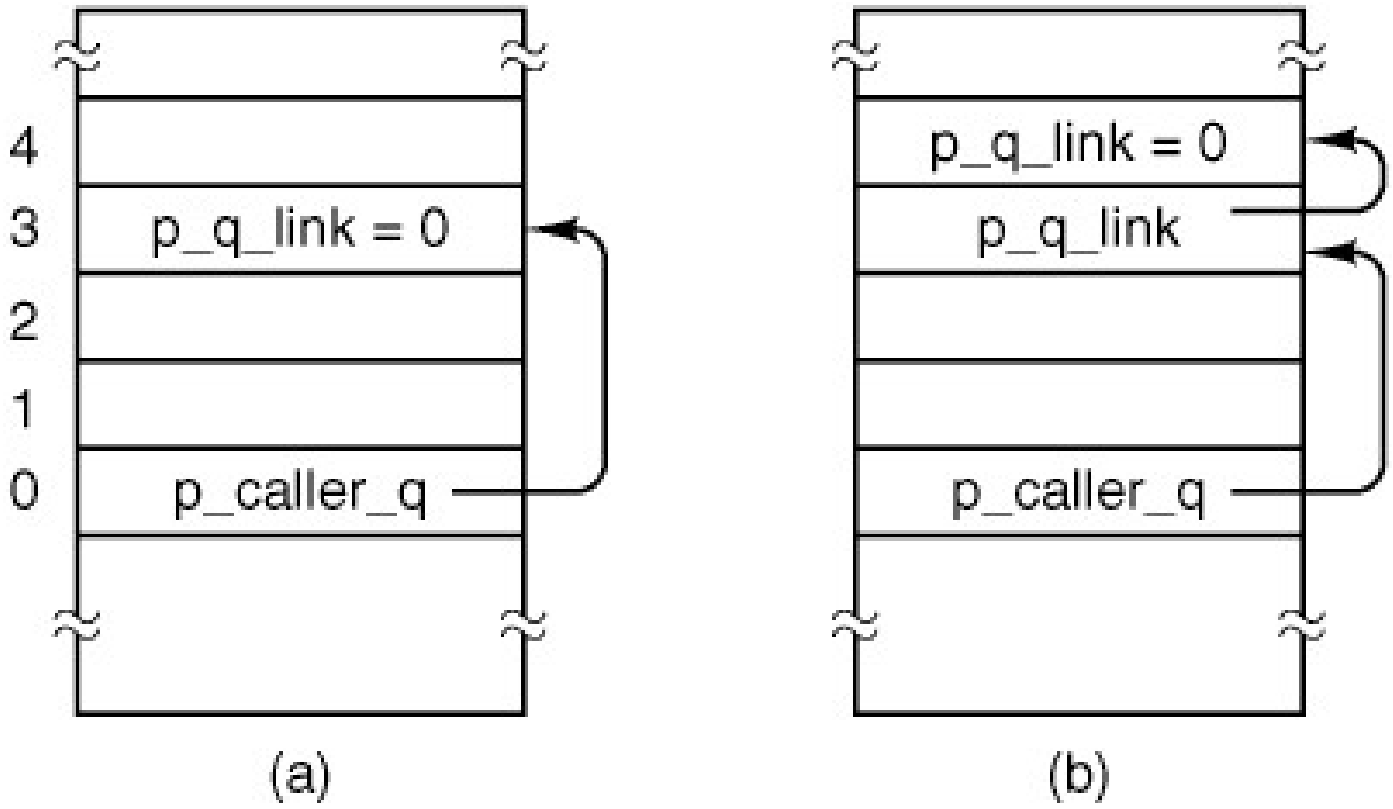


Figure 2-42. Queueing of processes trying to send to process 0.

Scheduling in MINIX

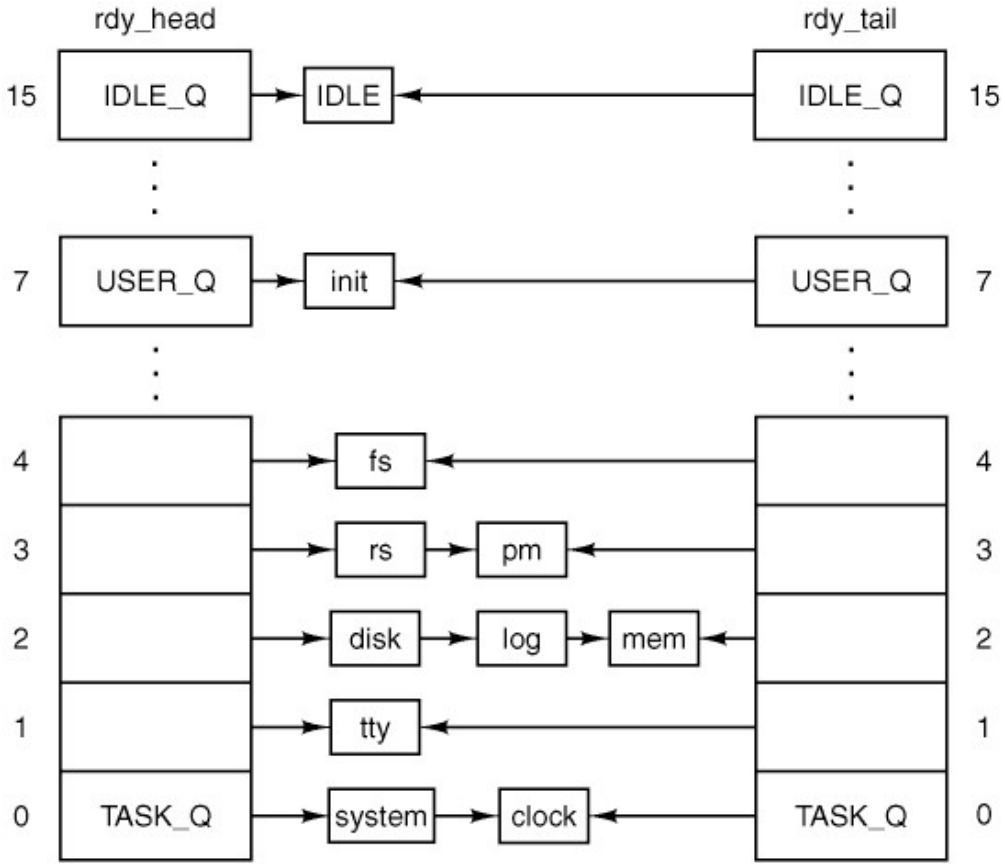


Figure 2-43. The scheduler maintains sixteen queues, one per priority level. Shown here is the initial queuing process as MINIX 3 starts up.

Hardware-Dependent Kernel Support

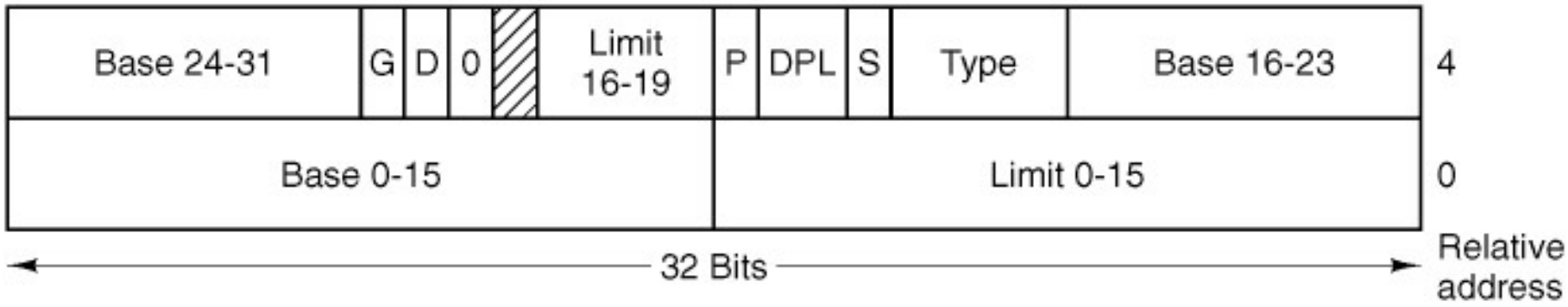


Figure 2-44. The format of an Intel segment descriptor.

Overview of System Task (1)

Message type	From	Meaning
sys_fork	PM	A process has forked
sys_exec	PM	Set stack pointer after EXEC call
sys_exit	PM	A process has exited
sys_nice	PM	Set scheduling priority
sys_privctl	RS	Set or change privileges
sys_trace	PM	Carry out an operation of the PTRACE call
sys_kill	PM,FS, TTY	Send signal to a process after KILL call
sys_getksig	PM	PM is checking for pending signals
sys_endksig	PM	PM has finished processing signal
sys_sigsend	PM	Send a signal to a process
sys_sigreturn	PM	Cleanup after completion of a signal
sys_irqctl	Drivers	Enable, disable, or configure interrupt

Figure 2-45. The message types accepted by the system task. "Any" means any system process; user processes cannot call the system task directly

Overview of System Task (2)

Message type	From	Meaning
sys_devio	Drivers	Read from or write to an I/O port
sys_sdevio	Drivers	Read or write string from/to I/O port
sys_vdevio	Drivers	Carry out a vector of I/O requests
sys_int86	Drivers	Do a real-mode BIOS call
sys_newmap	PM	Set up a process memory map
sys_segctl	Drivers	Add segment and get selector (far data access)
sys_memset	PM	Write char to memory area
sys_umap	Drivers	Convert virtual address to physical address
sys_vircopy	FS, Drivers	Copy using pure virtual addressing
sys_physcopy	Drivers	Copy using physical addressing
sys_virvcopy	Any	Vector of VCOPY requests
sys_physvcopy	Any	Vector of PHYSCOPY requests
sys_times	PM	Get uptime and process times
sys_setalarm	PM, FS, Drivers	Schedule a synchronous alarm
sys_abort	PM, TTY	Panic: MINIX is unable to continue
sys_getinfo	Any	Request system information

Figure 2-45. The message types accepted by the system task. "Any" means any system process; user processes cannot call the system task directly

The Clock Task in MINIX 3

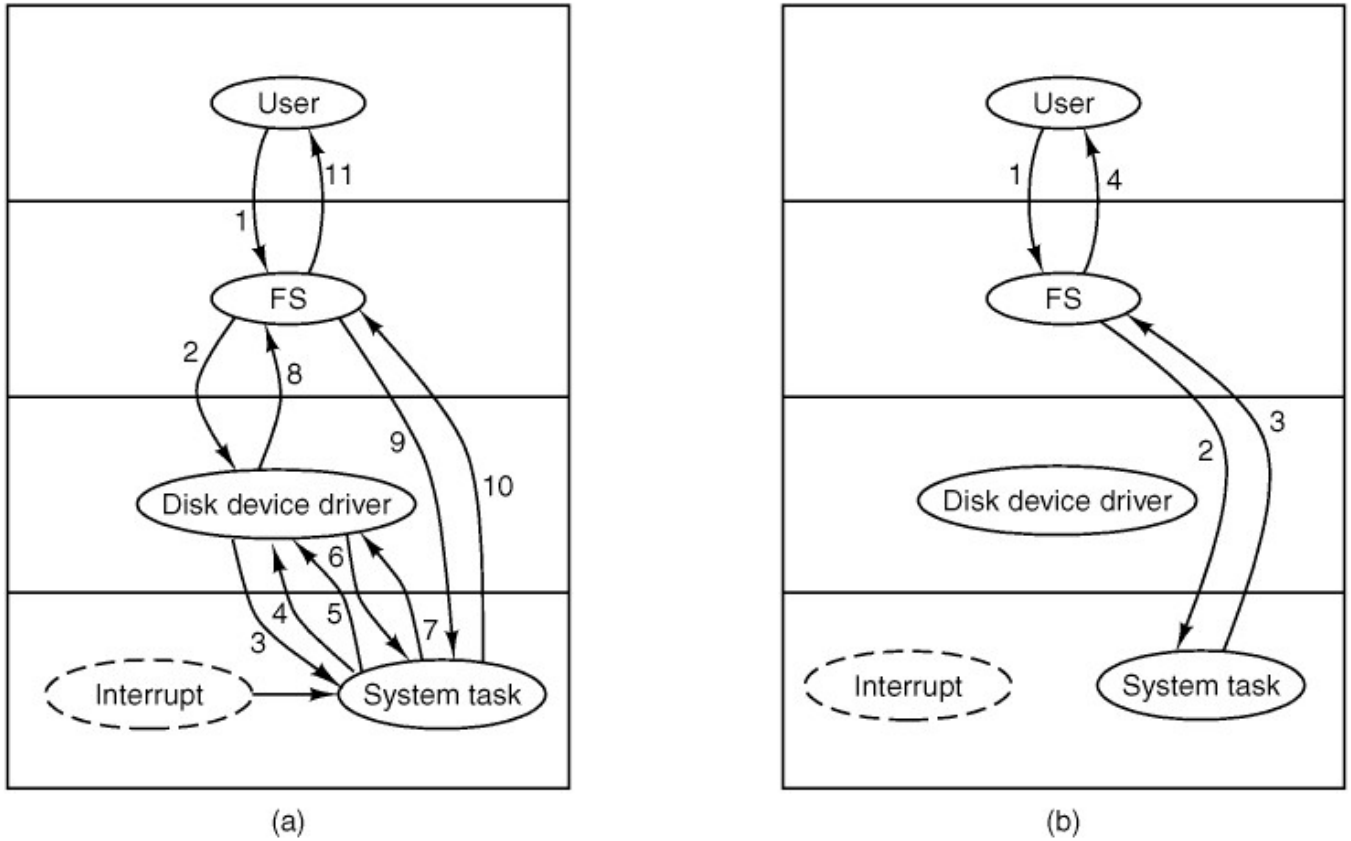


Figure 2-46. (a) Worst case for reading a block requires seven messages. (b) Best case for reading a block requires four messages.

Clock Hardware

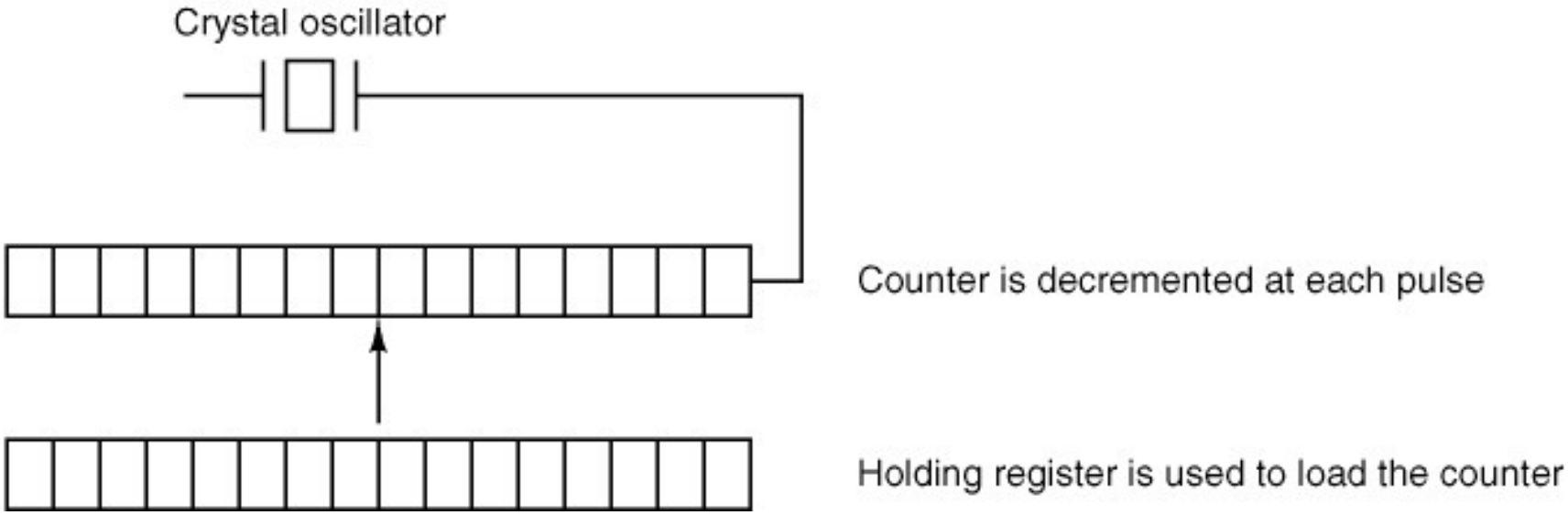


Figure 4-47. A programmable clock.

Clock Software (1)

Typical duties of a clock driver.

1. Maintain time of day
2. Prevent processes from running longer than allowed
3. Accounting for CPU usage
4. Handling alarm system call by user processes
5. Providing watchdog timers for parts of system itself
6. Doing profiling, monitoring, and statistics gathering

Clock Software (2)

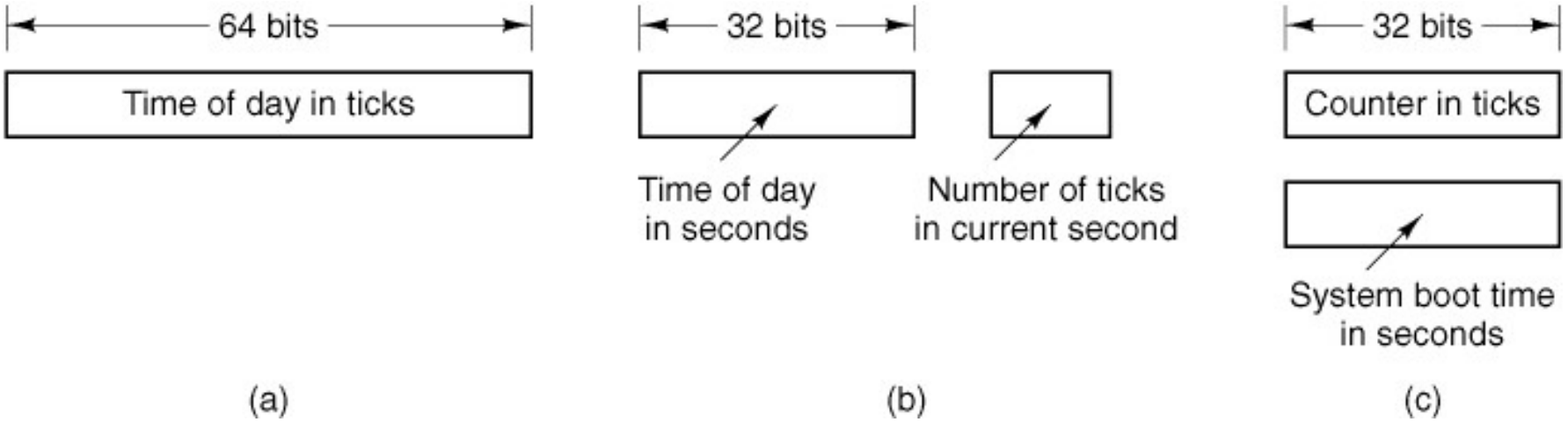


Figure 2-48. Three ways to maintain the time of day.

Clock Software (3)

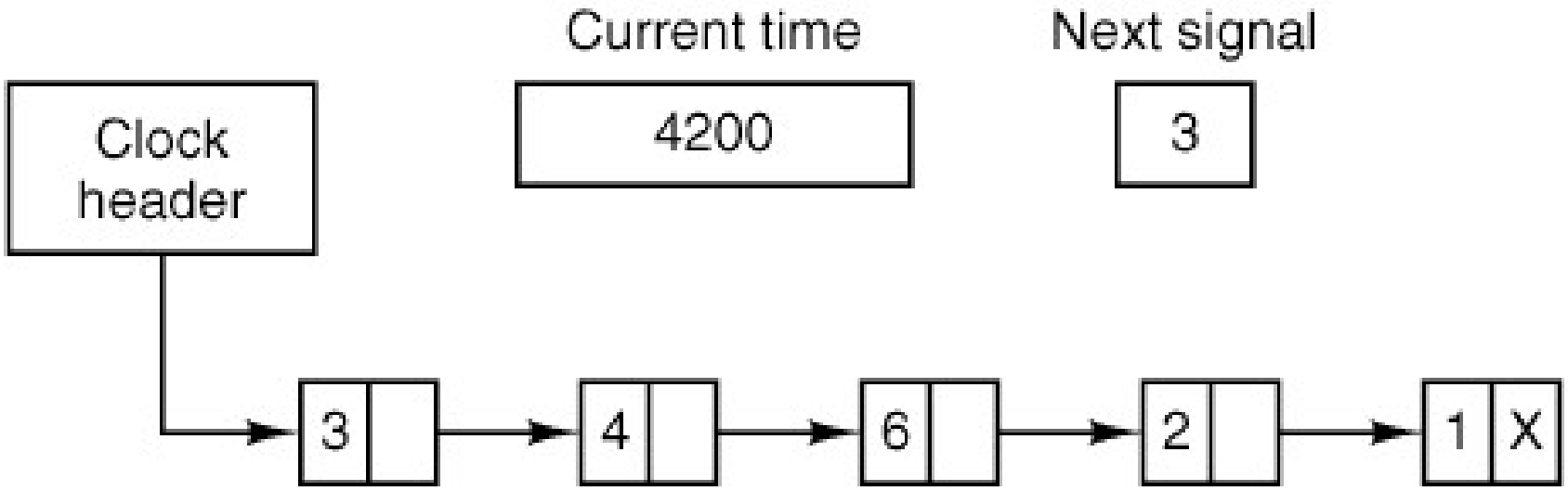


Figure 2-49. Simulating multiple timers with a single clock.

Summary of Clock Services

Service	Access	Response	Clients
get_uptime	Function call	Ticks	Kernel or system task
set_timer	Function call	None	Kernel or system task
reset_timer	Function call	None	Kernel or system task
read_clock	Function call	Count	Kernel or system task
clock_stop	Function call	None	Kernel or system task
Synchronous alarm	System call	Notification	Server or driver, via system task
POSIX alarm	System call	Signal	User process, via PM
Time	System call	Message	Any process, via PM

Figure 2-50. The time-related services supported by the clock driver.