2

PROCESSES

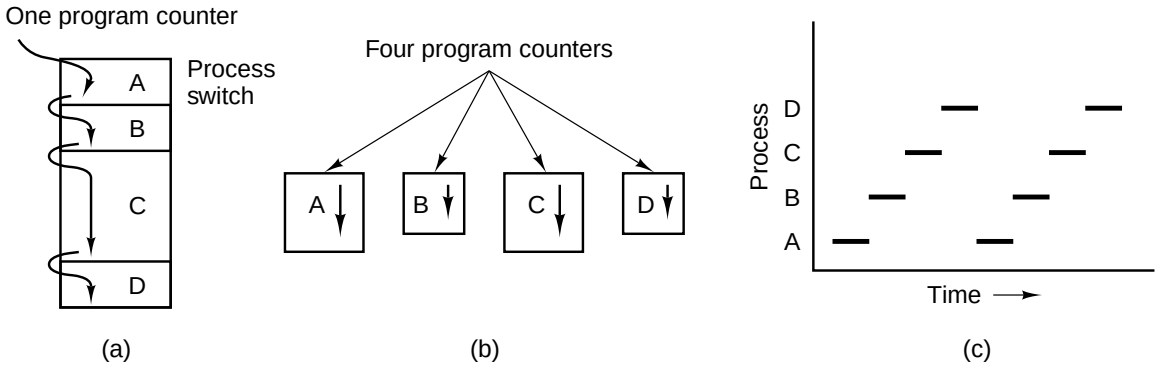

Figure 2-1. (a) Multiprogramming of four programs. (b) Conceptual model of four independent, sequential processes. (c) Only one program is active at any instant.

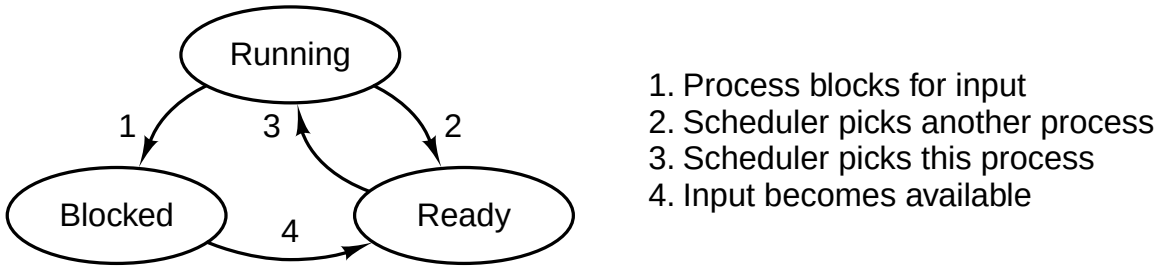

Figure 2-2. A process can be in running, blocked, or ready state. Transitions between these states are as shown.

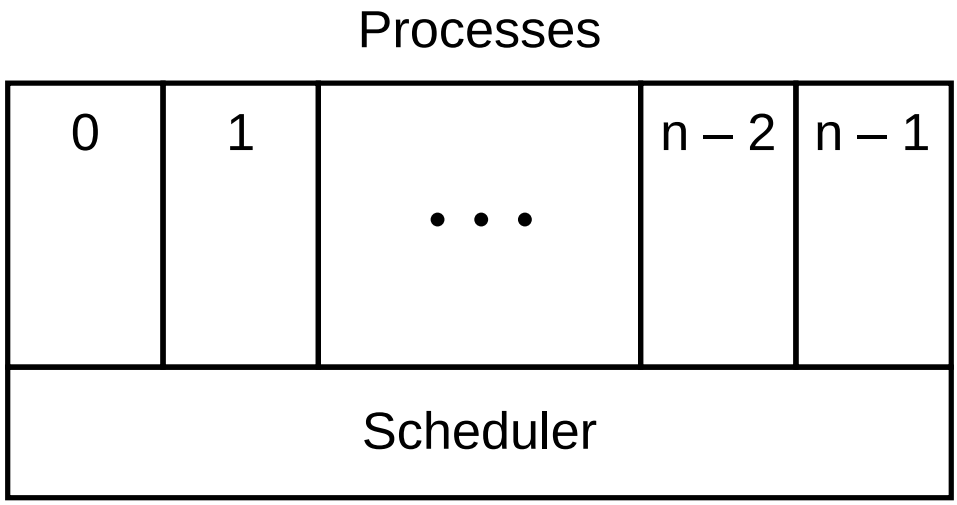

Figure 2-3. The lowest layer of a process-structured operating system handles interrupts and scheduling. Above that layer are sequential processes.

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	
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 process table.

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.

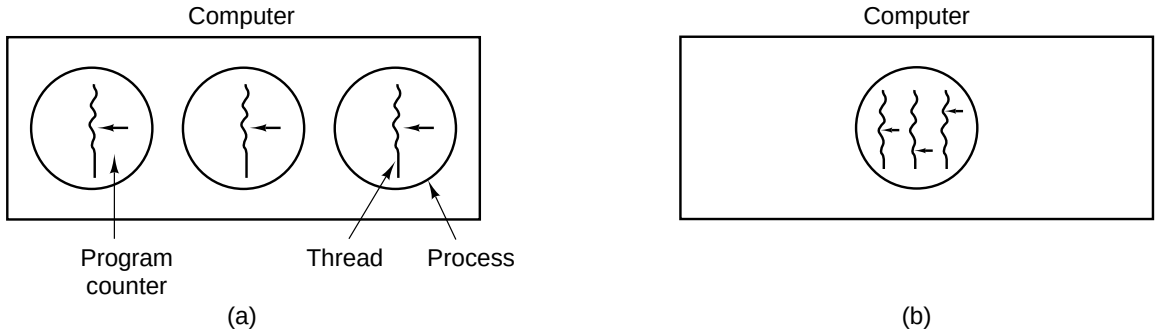

Figure 2-6. (a) Three processes each with one thread. (b) One process with three threads.

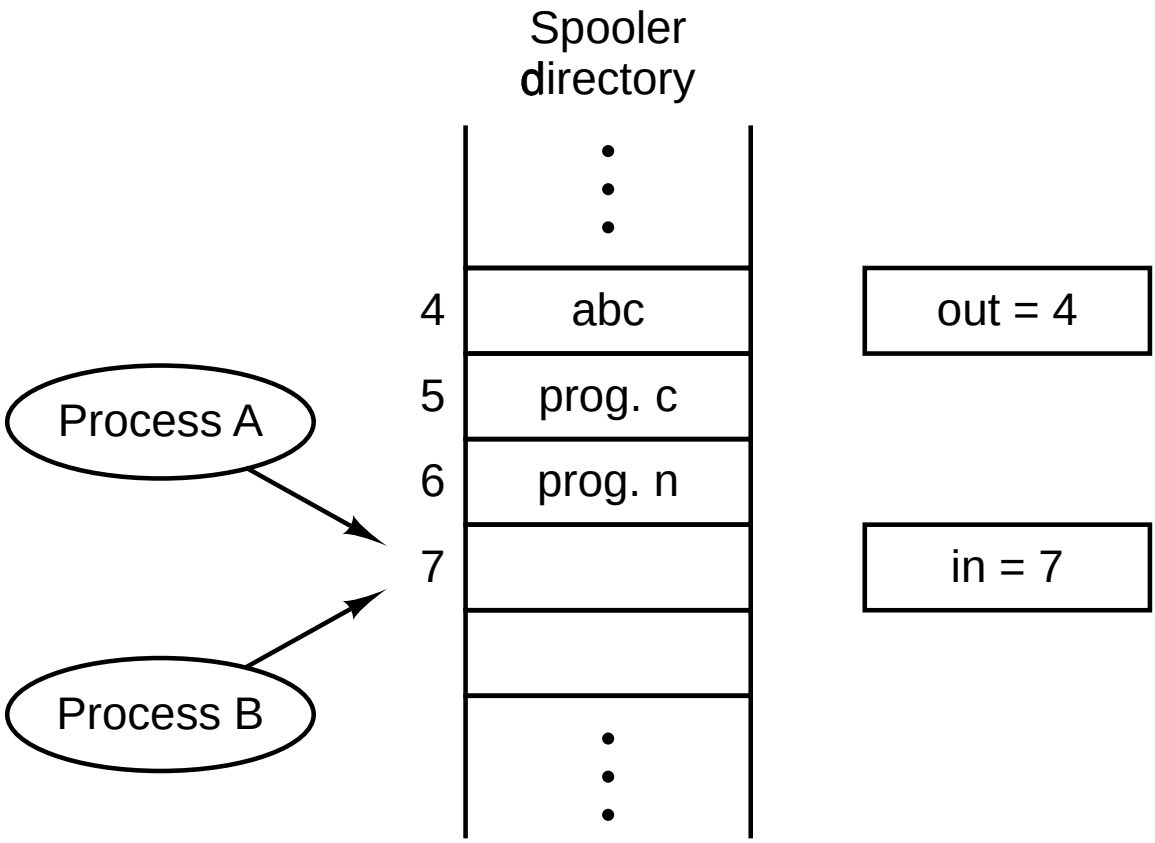

Figure 2-7. Two processes want to access shared memory at the same time.

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

(a)

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

(b)

Figure 2-8. A proposed solution to the critical region problem.

```
#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 */
{
  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 2-9. Peterson's solution for achieving mutual exclusion.

```
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 2-10. Setting and clearing locks using TSL.

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

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


void consumer(void)
{
  while (TRUE) {                    /* repeat forever */
        if (count == 0) sleep();    /* if buffer is empty, got to sleep */
        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();             /* print item */
  }
}
```

Figure 2-11. The producer-consumer problem with a fatal race condition.

```
#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 */
        produce_item(&item);        /* generate something to put in buffer */
        down(&empty);               /* decrement empty count */
        down(&mutex);               /* enter critical region */
        enter_item(item);           /* put new item in buffer */
        up(&mutex);                 /* leave critical region */
        up(&full);                  /* increment count of full slots */
  }
}

void consumer(void)
{
  int item;

  while (TRUE) {                    /* infinite loop */
        down(&full);                /* decrement full count */
        down(&mutex);               /* enter critical region */
        remove_item(&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 2-12. The producer-consumer problem using semaphores.

```
monitor example
  integer i;
  condition c;

  procedure producer(x);
  .
  .
  .
  end;

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

Figure 2-13. A monitor.

```
monitor ProducerConsumer
  condition full, empty;
  integer count;
  procedure enter;
  begin
    if count = N then wait(full);
    enter_item;
    count := count + 1;
    if count = 1 then signal(empty)
  end;
  procedure remove;
  begin
    if count = 0 then wait(empty);
    remove_item;
    count := count – 1;
    if count = N – 1 then signal(full)
  end;
  count := 0;
end monitor;
procedure producer;
begin
  while true do
  begin
    produce_item;
    ProducerConsumer.enter
  end
end;
procedure consumer;
begin
  while true do
  begin
    ProducerConsumer.remove;
    consume_item
  end
end;
```

Figure 2-14. The producer-consumer problem with monitors.

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

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

  while (TRUE) {
        produce_item(&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 */
  }
}

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 */
        extract_item(&m, &item);    /* extract item from message */
        send(producer, &m);         /* send back empty reply */
        consume_item(item);         /* do something with the item */
  }
}
```

Figure 2-15. The producer-consumer problem with *N* messages.

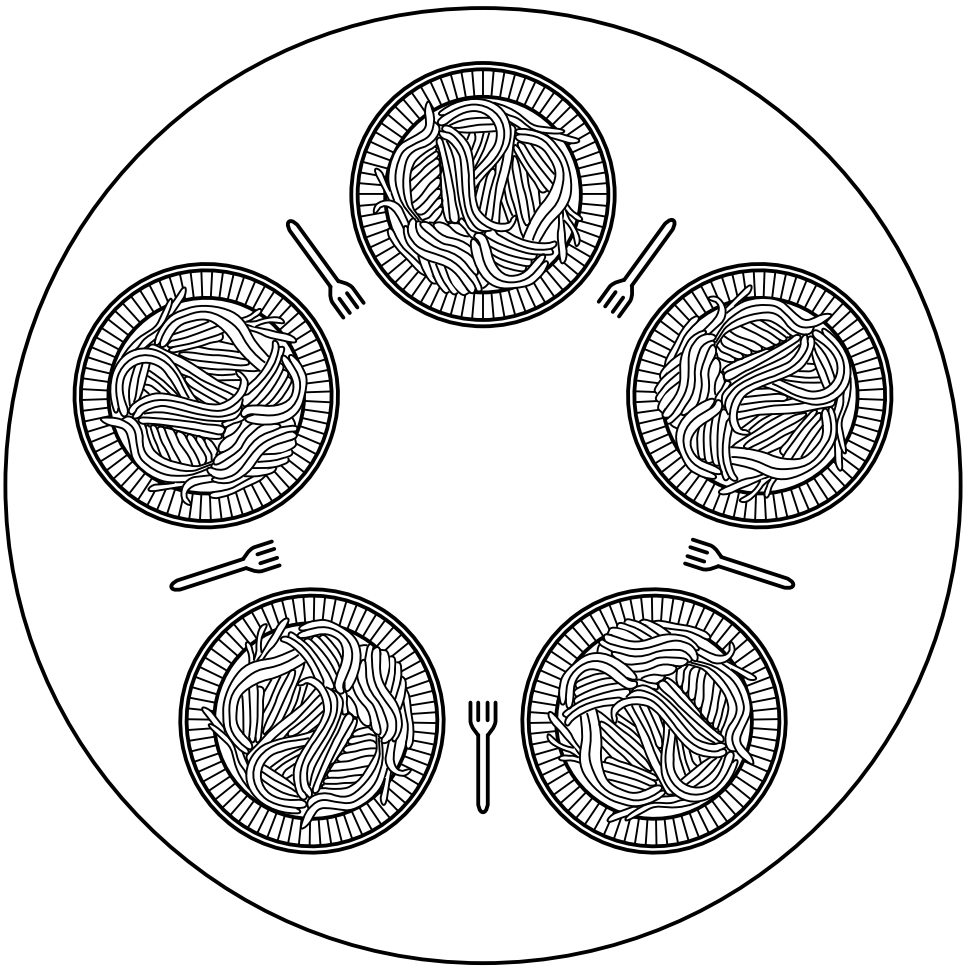

Figure 2-16. Lunch time in the Philosophy Department.

```
#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 2-17. A nonsolution to the dining philosophers problem.

```
#define N                   5                   /* number of philosophers */
#define LEFT    (i-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 */

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 */
}

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 2-18. A solution to the dining philosopher's problem.

```
typedef int semaphore;          /* use your imagination */
semaphore mutex = 1;            /* controls access to 'rc' */
semaphore db = 1;               /* controls access to the data base */
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 */
  }
}


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 2-19. A solution to the readers and writers problem.

Figure 2-20. The sleeping barber.

```
#define CHAIRS 5                /* # chairs for waiting customers */

typedef int semaphore;          /* use your imagination */

semaphore customers = 0;        /* # of customers waiting for service */
semaphore barbers = 0;          /* # of barbers waiting for customers */
semaphore mutex = 1;            /* for mutual exclusion */
int waiting = 0;                /* customers are waiting (not being cut) */

void barber(void)
{
  while (TRUE) {
        down(customers);        /* go to sleep if # of customers is 0 */
        down(mutex);            /* acquire access to 'waiting' */
        waiting = waiting - 1;  /* decrement count of waiting customers */
        up(barbers);            /* one barber is now ready to cut hair */
        up(mutex);              /* release 'waiting' */
        cut_hair();             /* cut hair (outside critical region) */
  }
}

void customer(void)
{
  down(mutex);                  /* enter critical region */
  if (waiting < CHAIRS) {       /* if there are no free chairs, leave */
        waiting = waiting + 1;  /* increment count of waiting customers */
        up(customers);          /* wake up barber if necessary */
        up(mutex);              /* release access to 'waiting' */
        down(barbers);          /* go to sleep if # of free barbers is 0 */
        get_haircut();          /* be seated and be serviced */
  } else {
        up(mutex);              /* shop is full; do not wait */
  }
}
```

Figure 2-21. A solution to the sleeping barber problem.

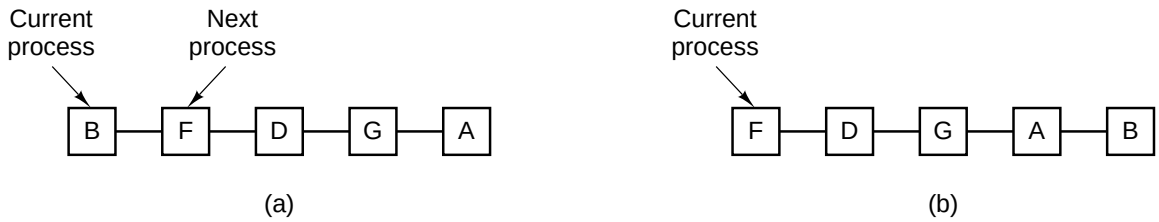

Figure 2-22. Round robin scheduling. (a) The list of runnable processes. (b) The list of runnable processes after *B* uses up its quantum.

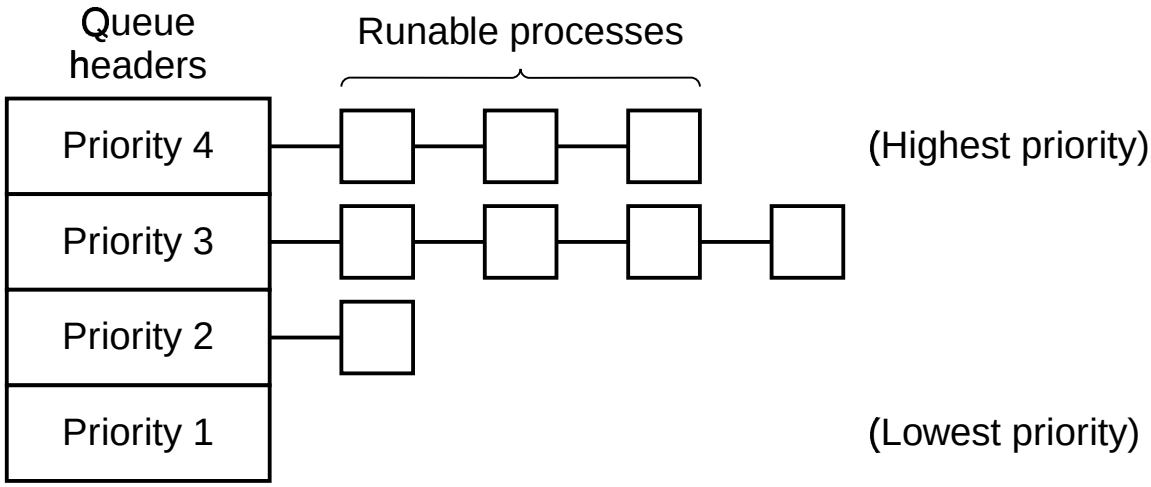

Figure 2-23. A scheduling algorithm with four priority classes.

Figure 2-24. An example of shortest job first scheduling.

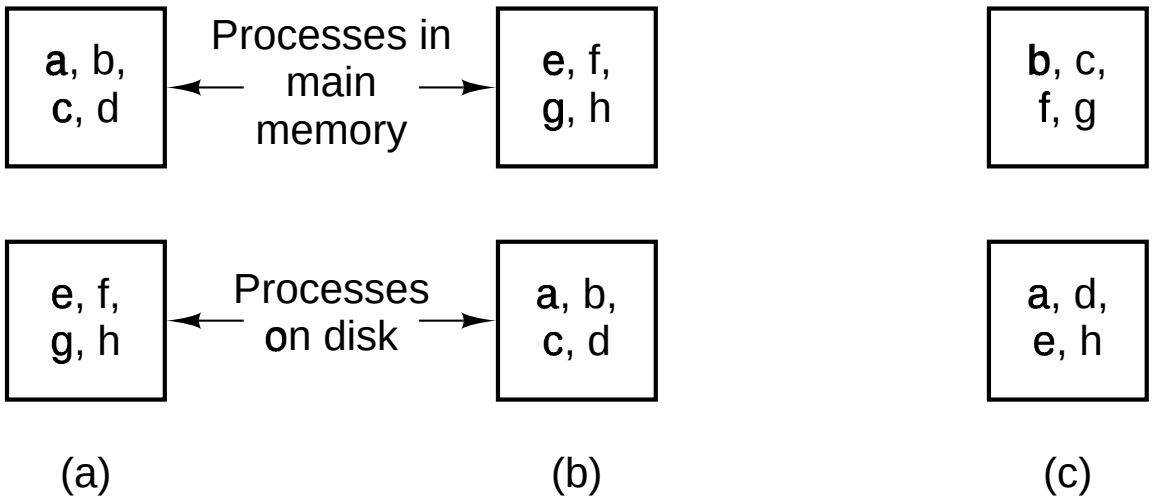

Figure 2-25. A two-level scheduler must move processes between disk and memory and also choose processes to run from among those in memory. Three different instants of time are represented by (a), (b), and (c) .

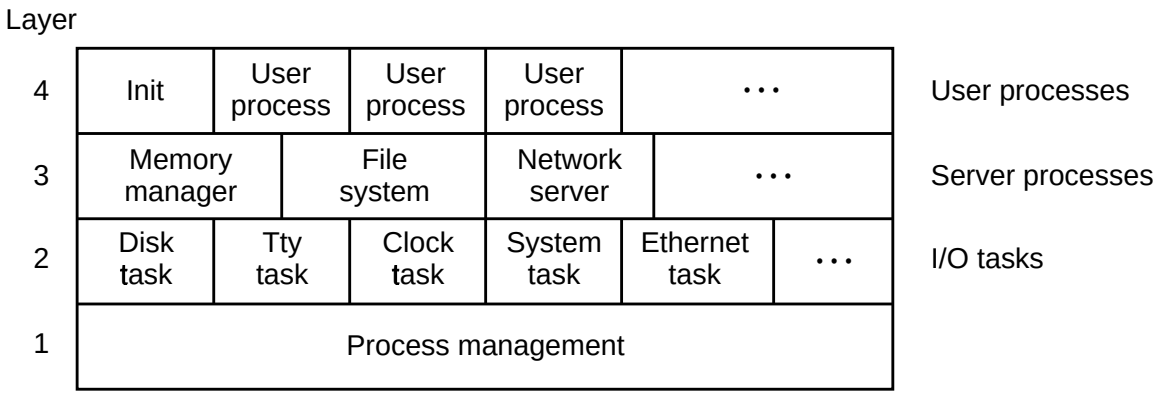

Figure 2-26. MINIX is structured in four layers.

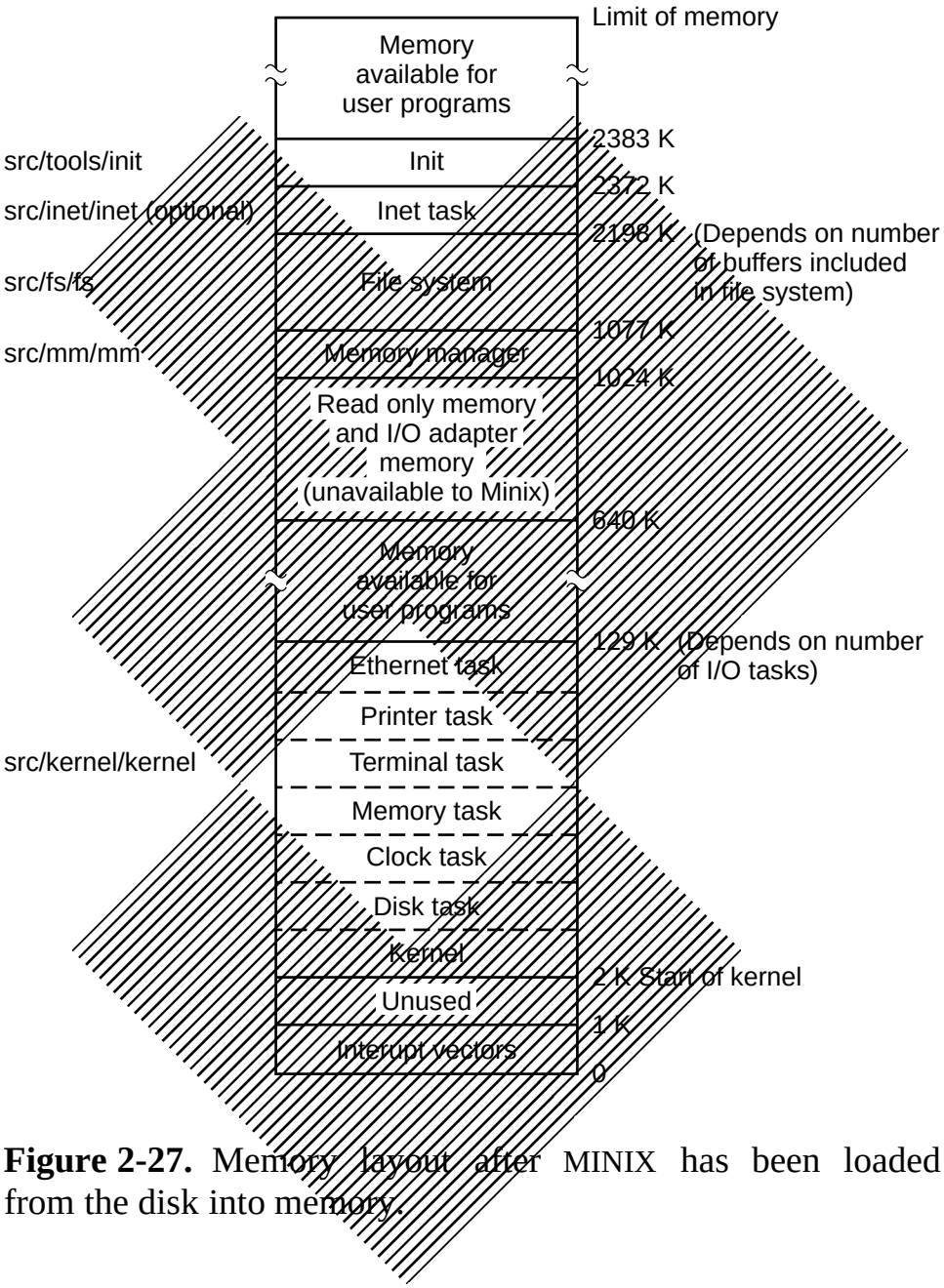

Figure 2-27. Memory layout after MINIX has been loaded from the disk into memory.

```
#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>
```

Figure 2-28. Part of a master header which ensures inclusion of header files needed by all C source files.

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

Figure 2-29. The size, in bits, of some types on 16-bit and 32-bit systems.

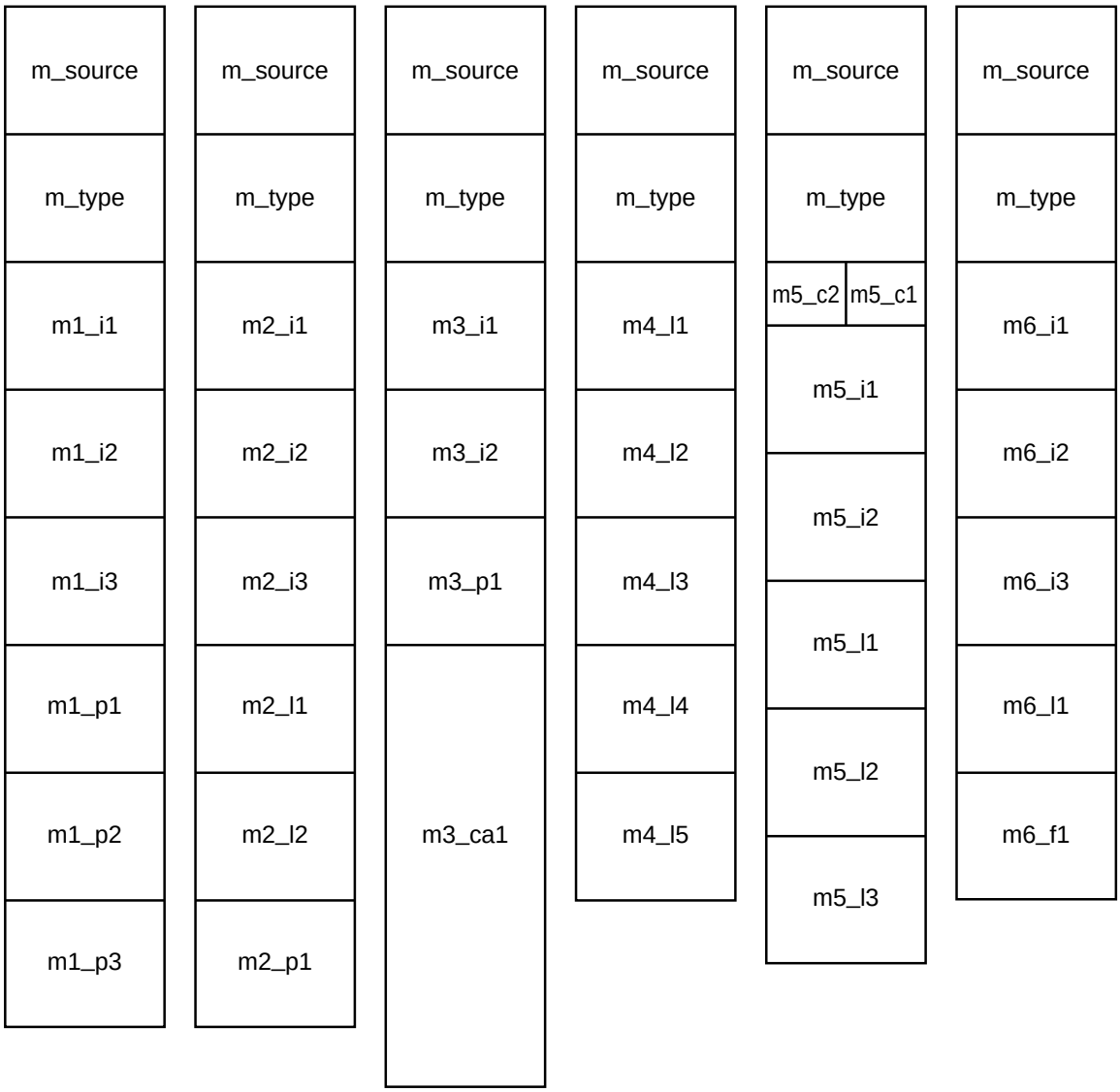

Figure 2-30. The six messages types used in MINIX. The sizes of message elements will vary, depending upon the architecture of the machine; this diagram illustrates sizes on a machine with 32-bit pointers, such as the Pentium (Pro).

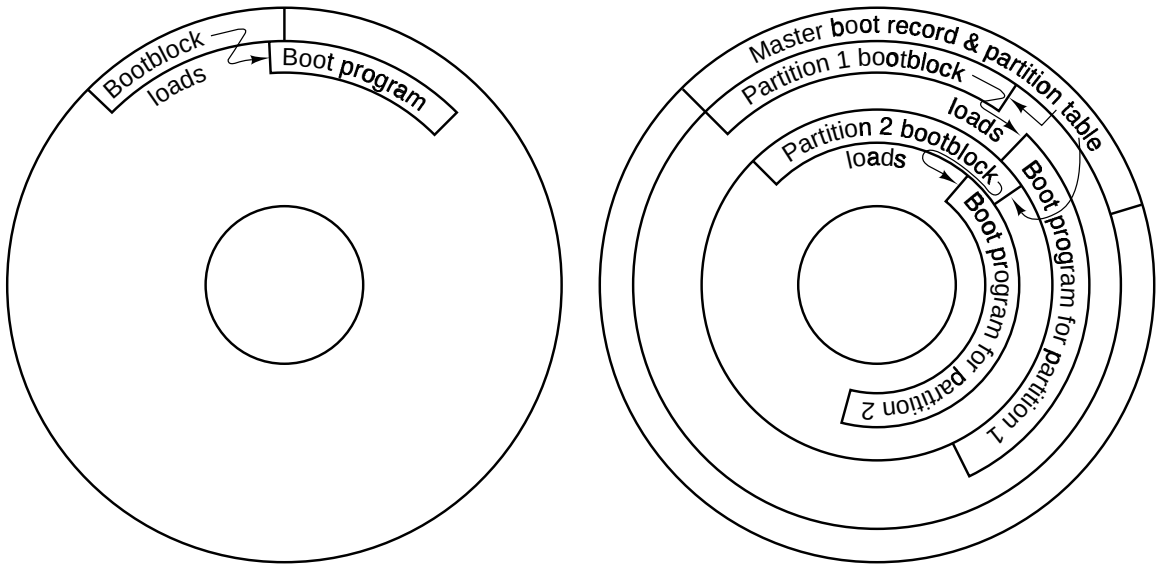

Figure 2-31. Disk structures used for bootstrapping. (a) Unpartitioned disk. The first sector is the bootblock. (b) Partitioned disk. The first sector is the master boot record.

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

Figure 2-32. How alternative assembly language source files are selected.

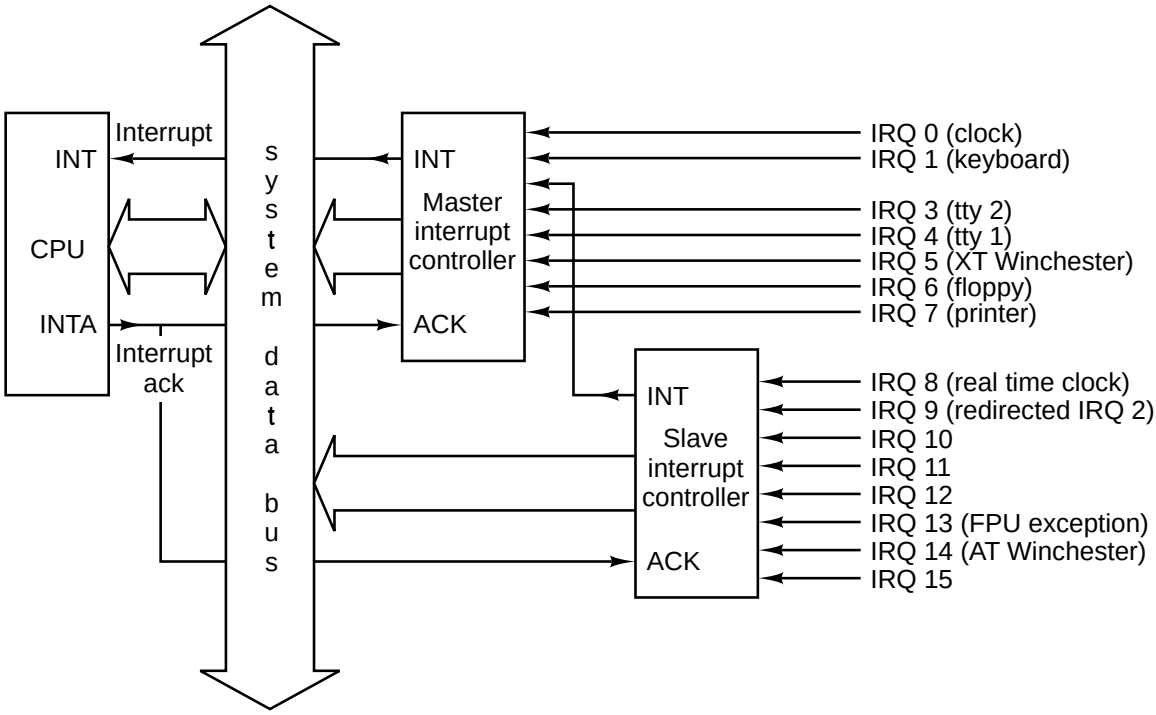

Figure 2-33. Interrupt processing hardware on a 32-bit Intel PC.

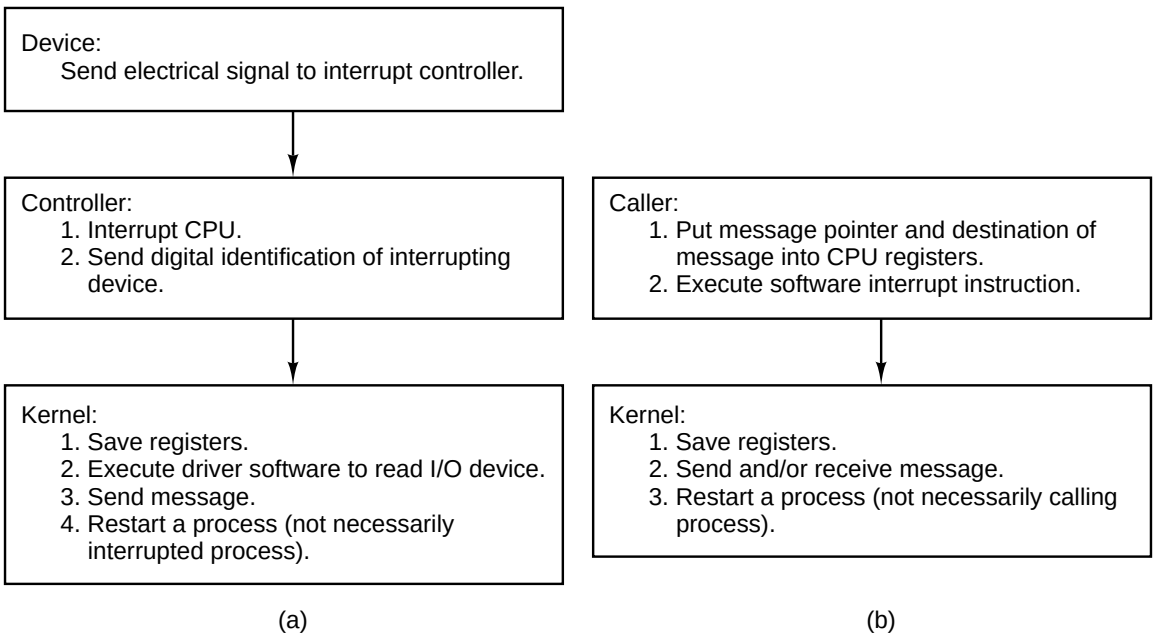

Figure 2-34. (a) How a hardware interrupt is processed. (b) How a system call is made.

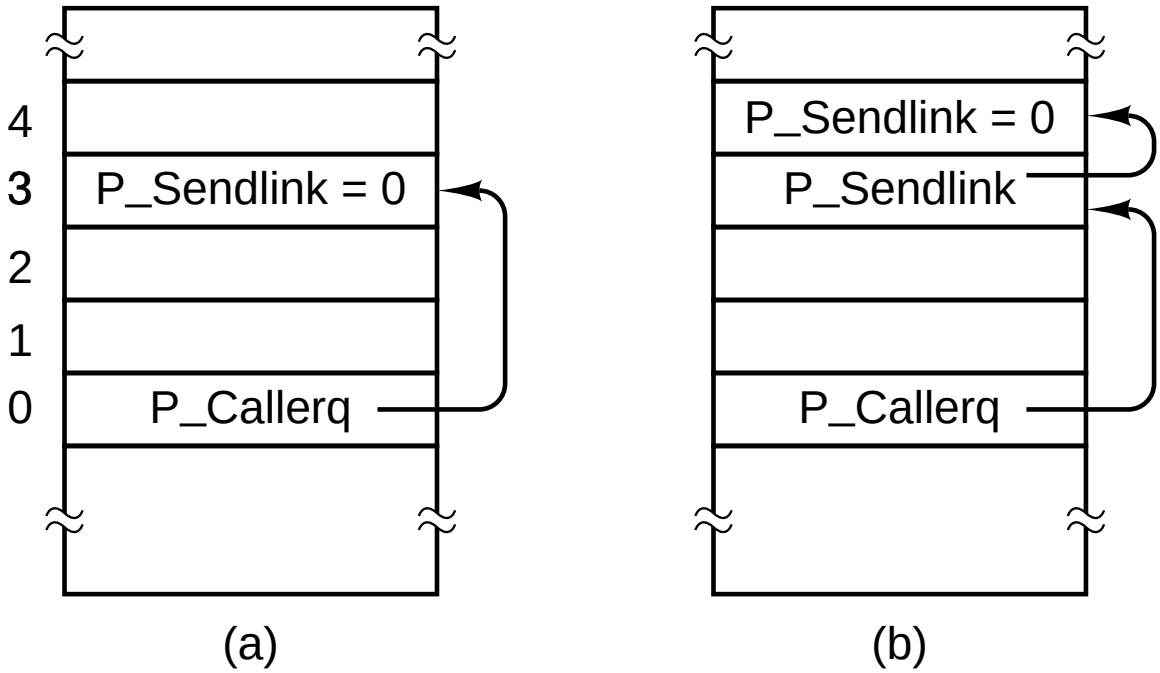

Figure 2-35. Queueing of processes trying to send to process 0.

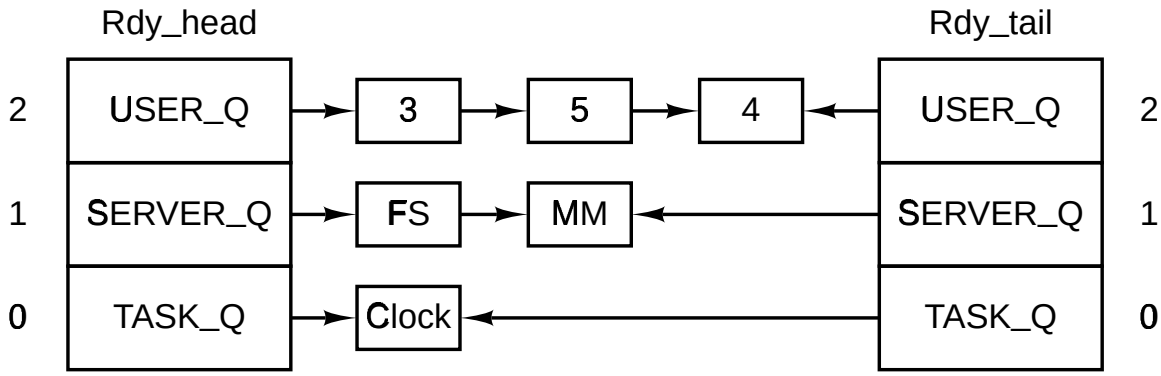

Figure 2-36. The scheduler maintains three queues, one per priority level.

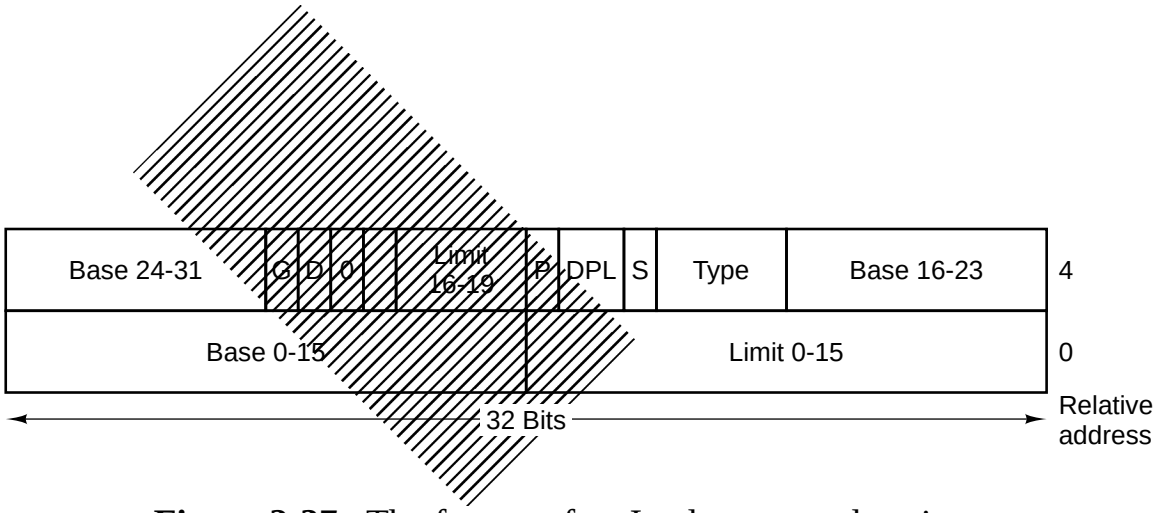

Figure 2-37. The format of an Intel segment descriptor.