2

PROCESSES AND THREADS

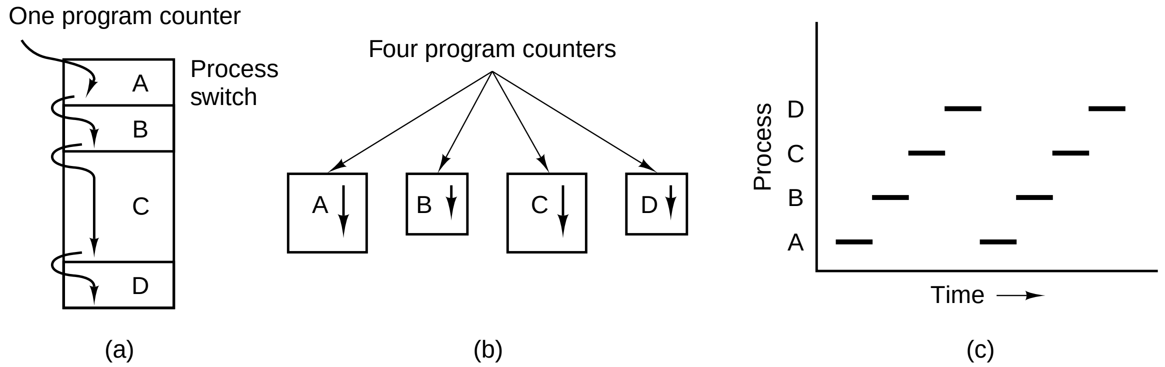
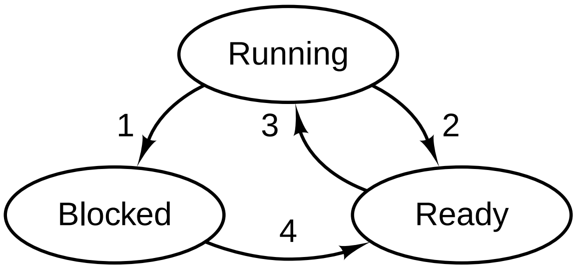


Fig. 2-1. (a) Multiprogramming of four programs. (b) Conceptual model of four independent, sequential processes. (c) Only one program is active at once.



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

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

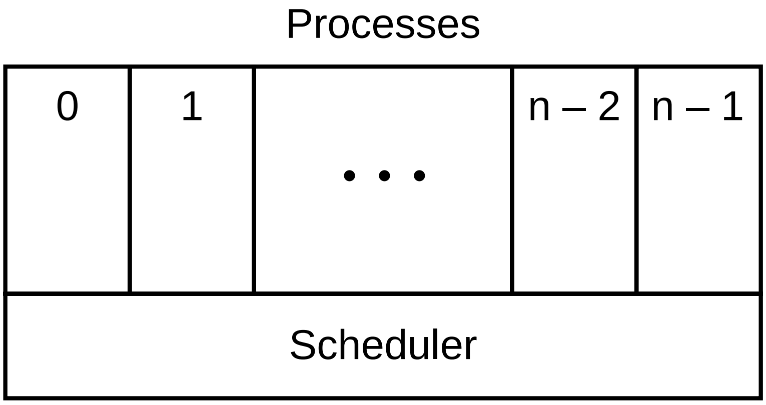


Fig. 2-3. The lowest layer of a process-structured operating system handles interrupts and scheduling. Above that layer are sequential processes.

Process management	**Memory management**	**File management**
Registers Program counter Program status word Stack pointer Process state Priority Scheduling parameters Process ID Parent process Process group Signals Time when process started CPU time used Children’s CPU time Time of next alarm	Pointer to text segment Pointer to data segment Pointer to stack segment	Root directory Working directory File descriptors User ID Group ID

Fig. 2-4. Some of the fields of a typical process table entry.

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 decides which process is to run next.
7. C procedure returns to the assembly code.
8. Assembly language procedure starts up new current process.

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

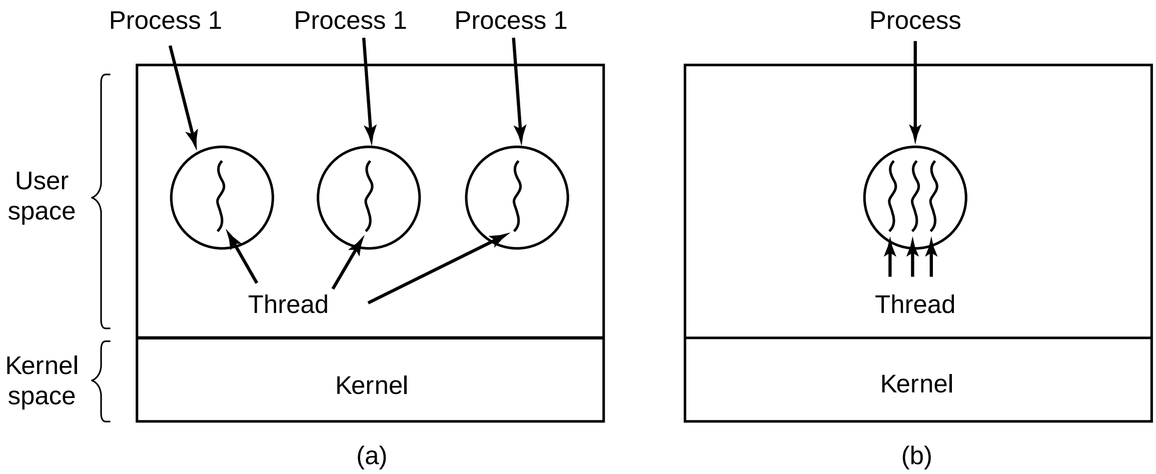


Fig. 2-6. (a) Three processes each with one thread. (b) One process with three threads.

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

Fig. 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.

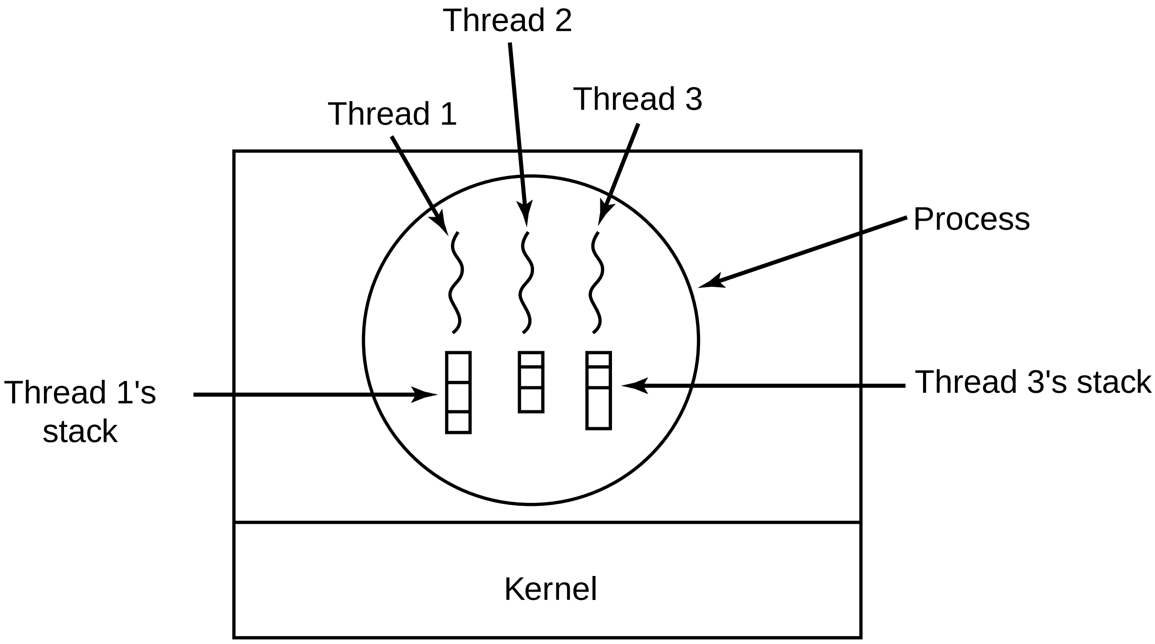


Fig. 2-8. Each thread has its own stack.

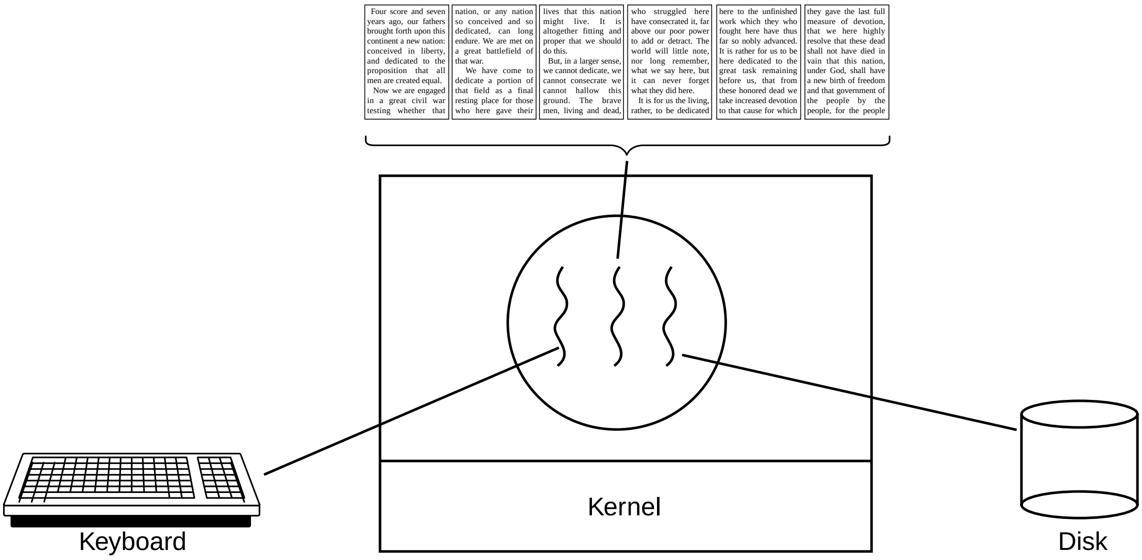


Fig. 2-9. A word processor with three threads.

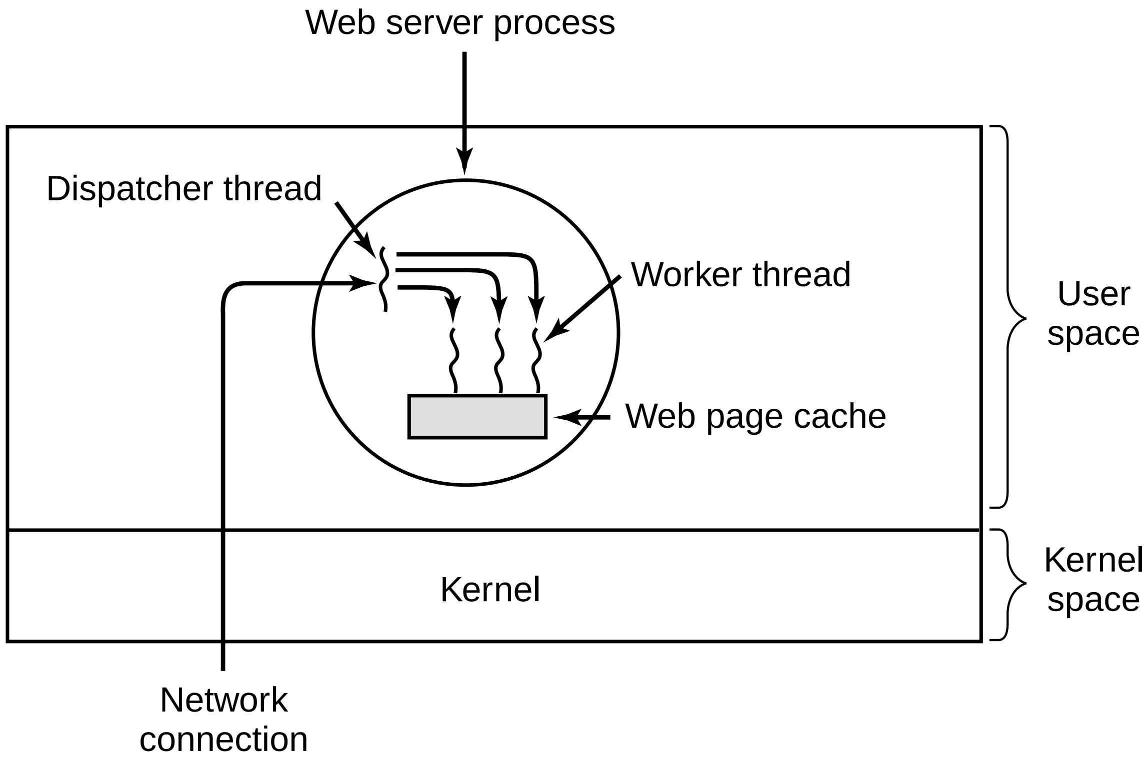


Fig. 2-10. A multithreaded Web server.

```
while (TRUE) {
    get_next_request(&buf);
    handoff_work(&buf);
}
```

(a)

```
while (TRUE) {
    wait_for_work(&buf)
    look_for_page_in_cache(&buf, &page);
    if (page_not_in_cache(&page))
        read_page_from_disk(&buf, &page);
    return_page(&page);
}
```

(b)

Fig. 2-11. A rough outline of the code for Fig. 2-10. (a) Dispatcher thread. (b) Worker thread.

Model	**Characteristics**
Threads	Parallelism, blocking system calls
Single-threaded process	No parallelism, blocking system calls
Finite-state machine	Parallelism, nonblocking system calls, interrupts

Fig. 2-12. Three ways to construct a server.

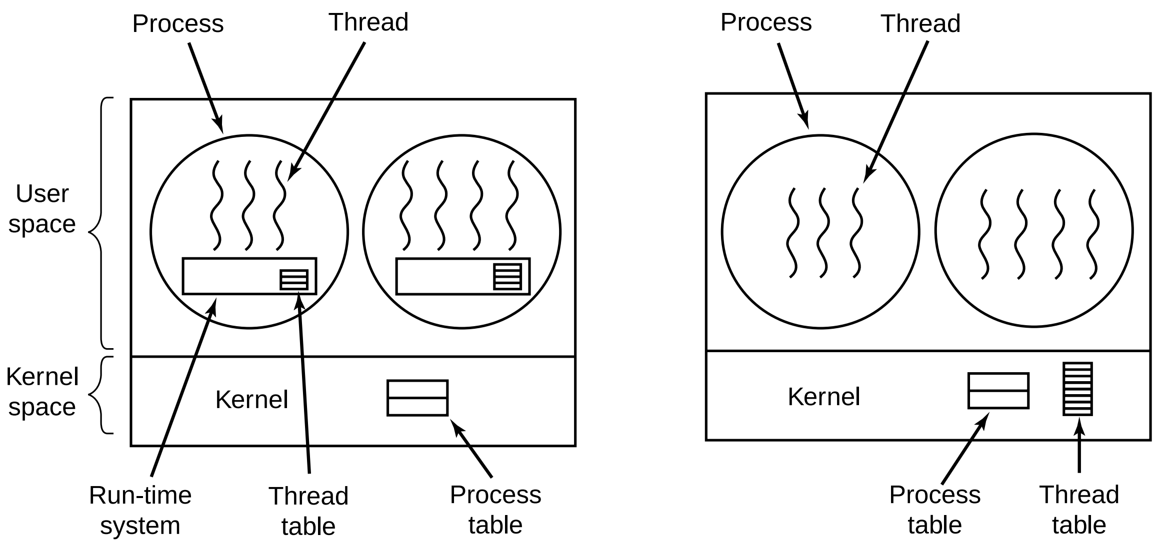


Fig. 2-13. (a) A user-level threads package. (b) A threads package managed by the kernel.

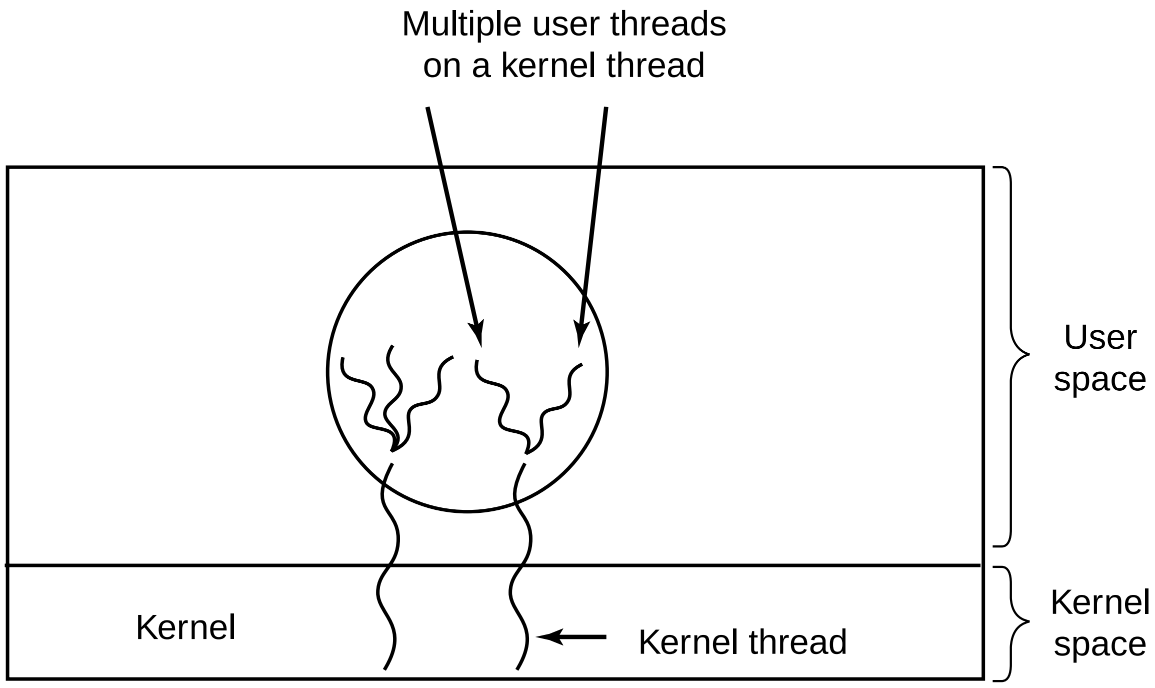


Fig. 2-14. Multiplexing user-level threads onto kernel-level threads.

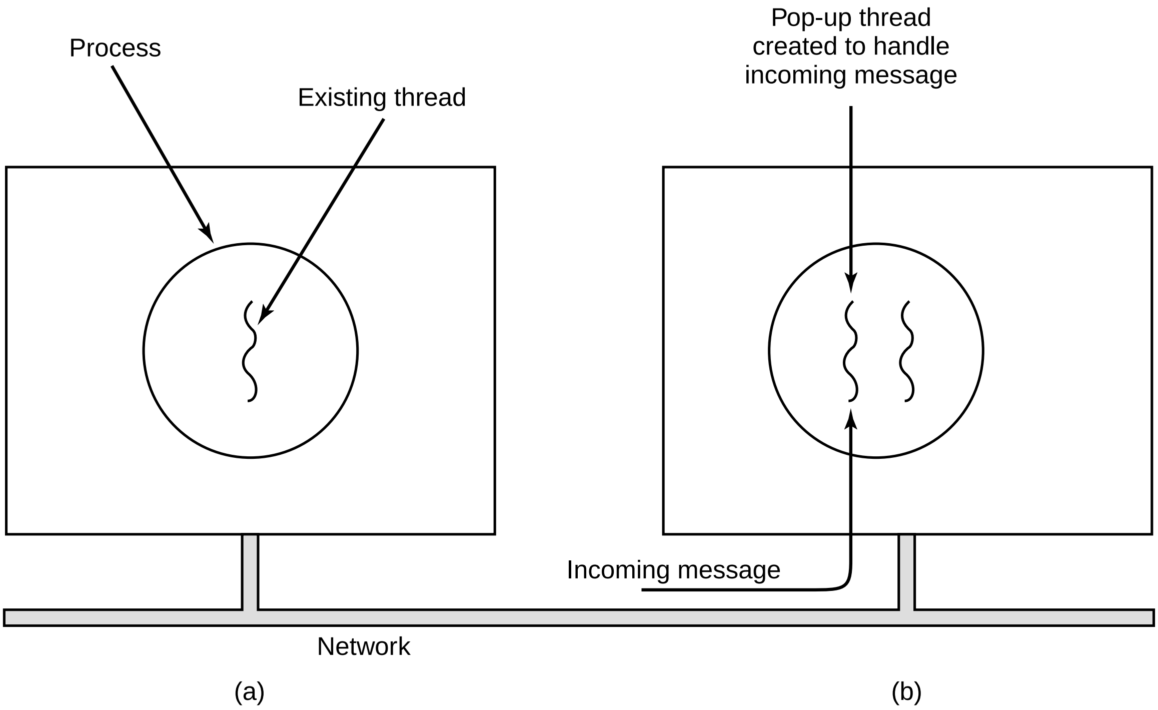


Fig. 2-15. Creation of a new thread when a message arrives. (a) Before the message arrives. (b) After the message arrives.

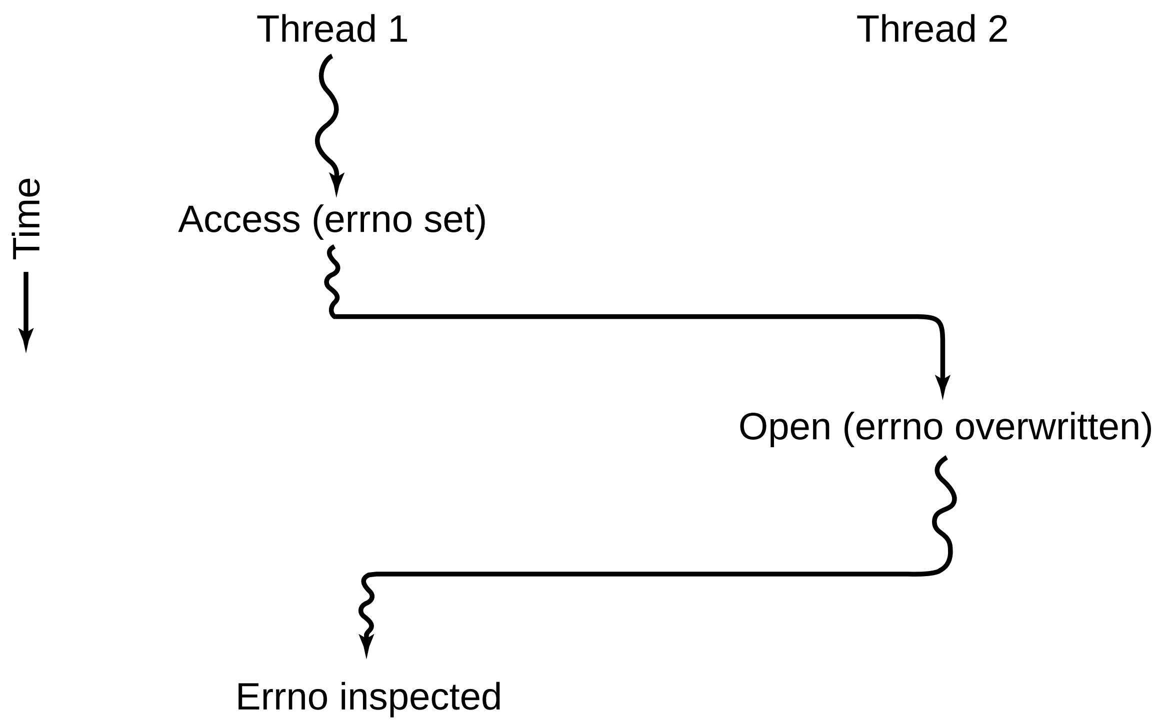


Fig. 2-16. Conflicts between threads over the use of a global variable.

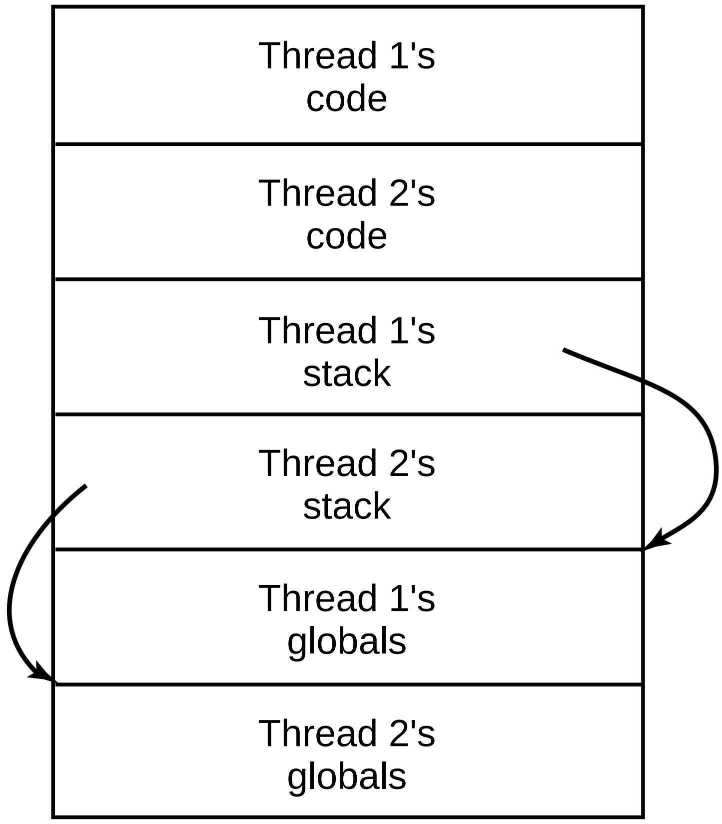


Fig. 2-17. Threads can have private global variables.

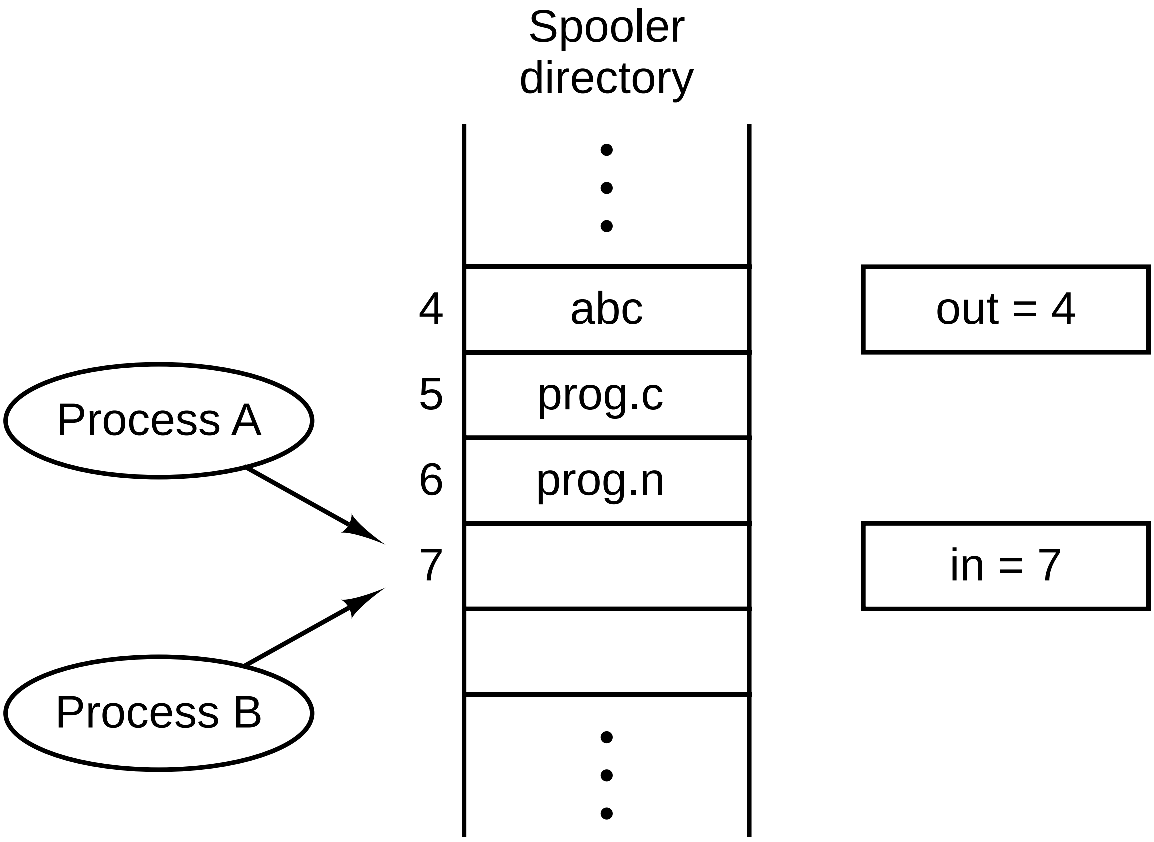


Fig. 2-18. Two processes want to access shared memory at the same time.

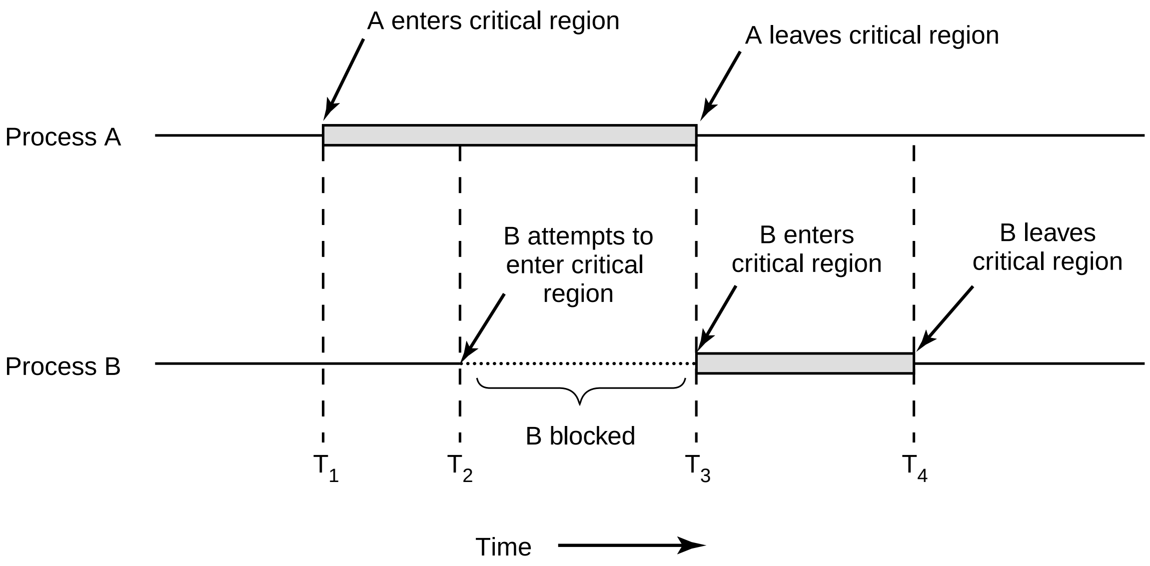


Fig. 2-19. Mutual exclusion using critical regions.

```
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)

Fig. 2-20. 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.

```
#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 */
}
```

Fig. 2-21. 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
```

Fig. 2-22. Entering and leaving a critical region using the TSL instruction.

```
#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? */
     }
}


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 */
     }
}
```

Fig. 2-23. 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 */
          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 */
     }
}


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 */
     }
}
```

Fig. 2-24. The producer-consumer problem using semaphores.

```
mutex_lock:
       TSL REGISTER,MUTEX          | copy mutex to register and set mutex to 1
       CMP REGISTER,#0             | was mutex zero?
       JZE ok                      | if it was zero, mutex was unlocked, so return
       CALL thread_yield           | mutex is busy; schedule another thread
       JMP mutex_lock              | try again later
ok:    RET                         | return to caller; critical region entered

mutex_unlock:
       MOVE MUTEX,#0               | store a 0 in mutex
       RET                         | return to caller
```

Fig. 2-25. Implementation of *mutex_lock* and *mutex_unlock*.

```
monitor example
      integer i;
      condition c;

      procedure producer( );
      .
      .
      .
      end;

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

Fig. 2-26. A monitor.

```
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;

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;
```

Fig. 2-27. An outline of the producer-consumer problem with monitors. Only one monitor procedure at a time is active. The buffer has *N* slots.

```
public class ProducerConsumer {
        static final int N = 100;       // constant giving the buffer size
        static producer p = new producer( );    // instantiate a new producer thread
        static consumer c = new consumer( ); // instantiate a new consumer thread
        static our_monitor mon = new our_monitor( );      // instantiate a new monitor

        public static void main(String args[ ]) {
           p.start( );       // start the producer thread
           c.start( );       // start the consumer thread
        }

        static class producer extends Thread {
           public void run( )  {// run method contains the thread code
              int item;
              while (true) {      // producer loop
                 item = produce_item( );
                 mon.insert(item);
              }
           }
           private int produce_item( ) { ... }      // actually produce
        }

        static class consumer extends Thread {
           public void run( )  {run method contains the thread code
              int item;
              while (true) {      // consumer loop
                 item = mon.remove( );
                 consume_item (item);
              }
           }
           private void consume_item(int item) { ... }        // actually consume
        }

        static class our_monitor {   // this is a monitor
           private int buffer[ ] = new int[N];
           private int count = 0, lo = 0, hi = 0;  // counters and indices

           public synchronized void insert(int val) {
              if (count == N) go_to_sleep( );     // if the buffer is full, go to sleep
              buffer [hi] = val; // insert an item into the buffer
              hi = (hi + 1) % N;        // slot to place next item in
              count = count + 1;      // one more item in the buffer now
              if (count == 1) notify( );        // if consumer was sleeping, wake it up
           }
```

```
        public synchronized int remove( ) {
            int val;
            if (count == 0) go_to_sleep( );     // if the buffer is empty, go to sleep
            val = buffer [lo]; // fetch an item from the buffer
            lo = (lo + 1) % N;        // slot to fetch next item from
            count = count – 1;      // one few items in the buffer
            if (count == N – 1) notify( ); // if producer was sleeping, wake it up
            return val;
        }
        private void go_to_sleep( ) { try{wait( );} catch(InterruptedException exc) {};}
    }
}
```

Fig. 2-28. A solution to the producer-consumer problem in Java.

```
#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 */
     }
}

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 */
     }
}
```

Fig. 2-29. The producer-consumer problem with *N* messages.

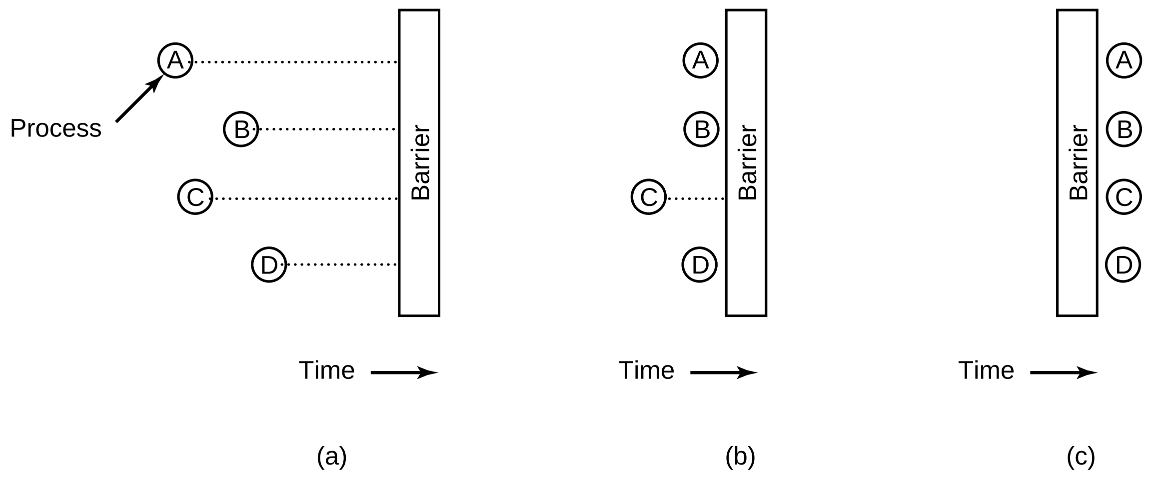


Fig. 2-30. Use of a barrier. (a) Processes approaching a barrier. (b) All processes but one blocked at the barrier. (c) When the last process arrives at the barrier, all of them are let through.

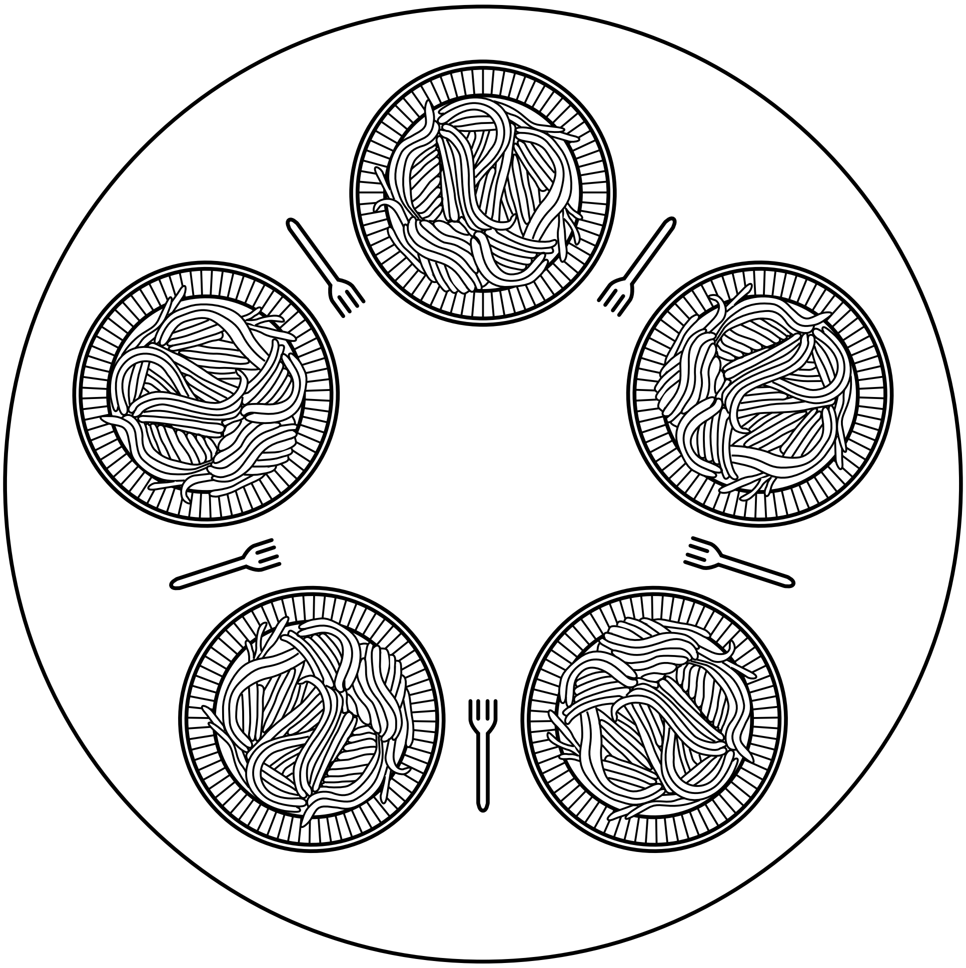


Fig. 2-31. Lunch time in the Philosophy Department.

```
#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 */
    }
}
```

Fig. 2-32. A nonsolution to the dining philosophers problem.

```
#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 */

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]);
    }
}
```

Fig. 2-33. A solution to the dining philosophers problem.

```
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 */
    }
}


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 */
    }
}
```

Fig. 2-34. A solution to the readers and writers problem.

Fig. 2-35. 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 */
    }
}
```

Fig. 2-36. A solution to the sleeping barber problem.

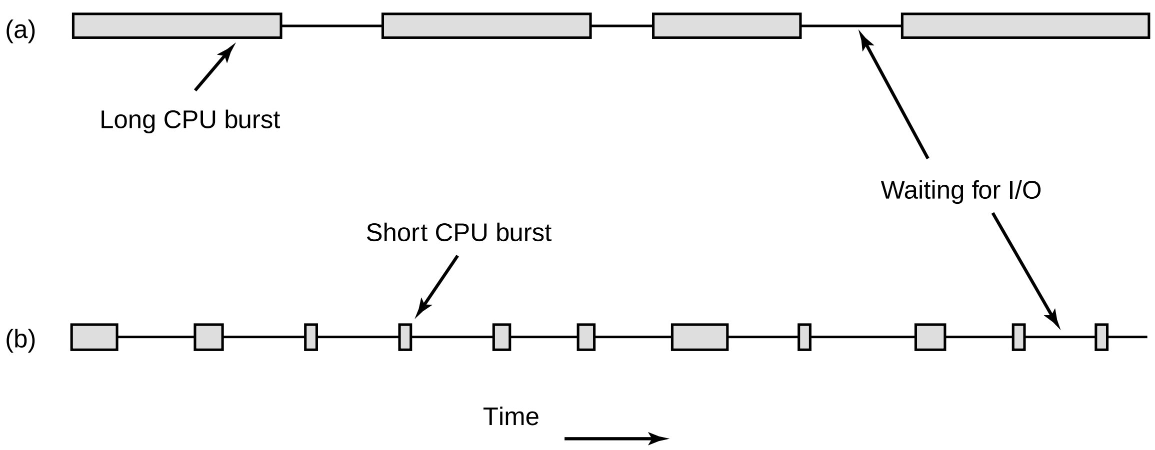


Fig. 2-37. Bursts of CPU usage alternate with periods of waiting for I/O. (a) A CPU-bound process. (b) An I/O-bound process.

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

Fig. 2-38. Some goals of the scheduling algorithm under different circumstances.

Fig. 2-39. An example of shortest job first scheduling. (a) Running four jobs in the original order. (b) Running them in shortest job first order.

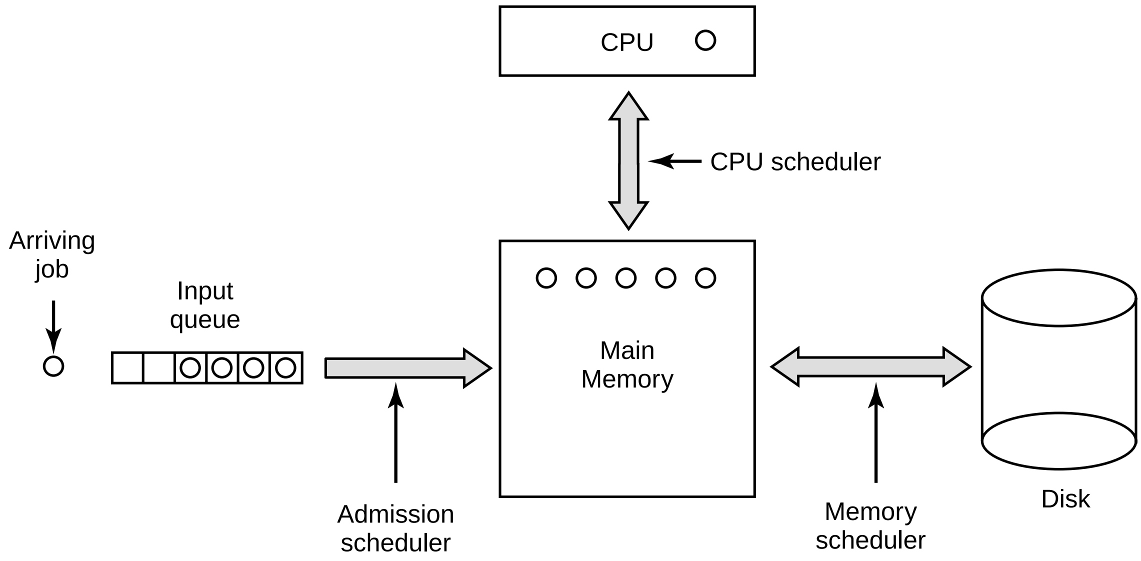


Fig. 2-40. Three-level scheduling.

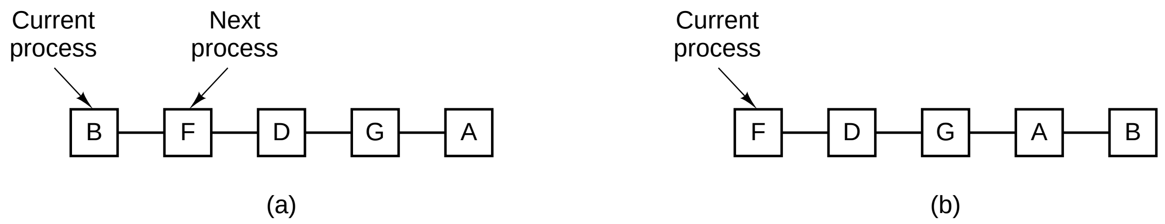


Fig. 2-41. Round-robin scheduling. (a) The list of runnable processes. (b) The list of runnable processes after *B* uses up its quantum.

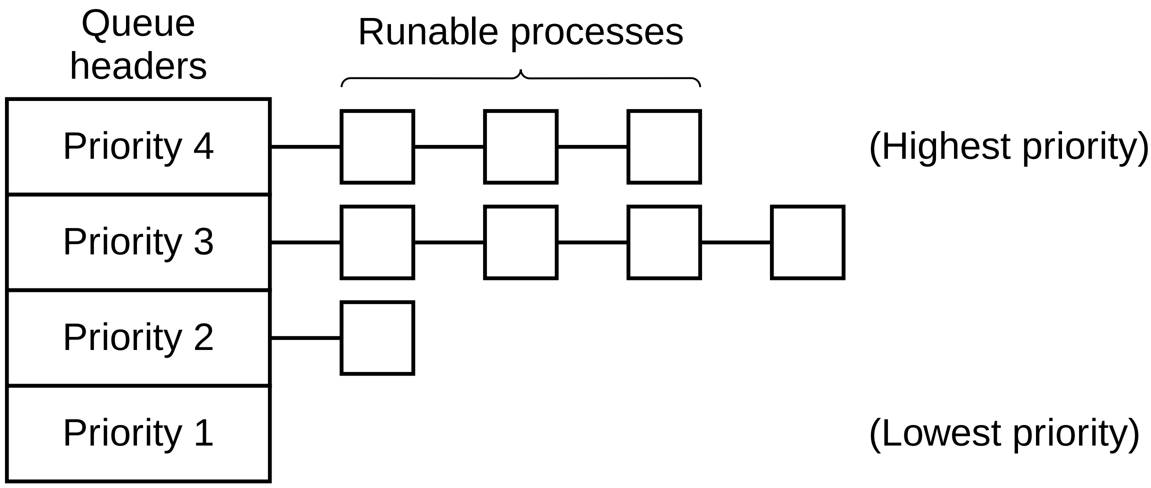


Fig. 2-42. A scheduling algorithm with four priority classes.

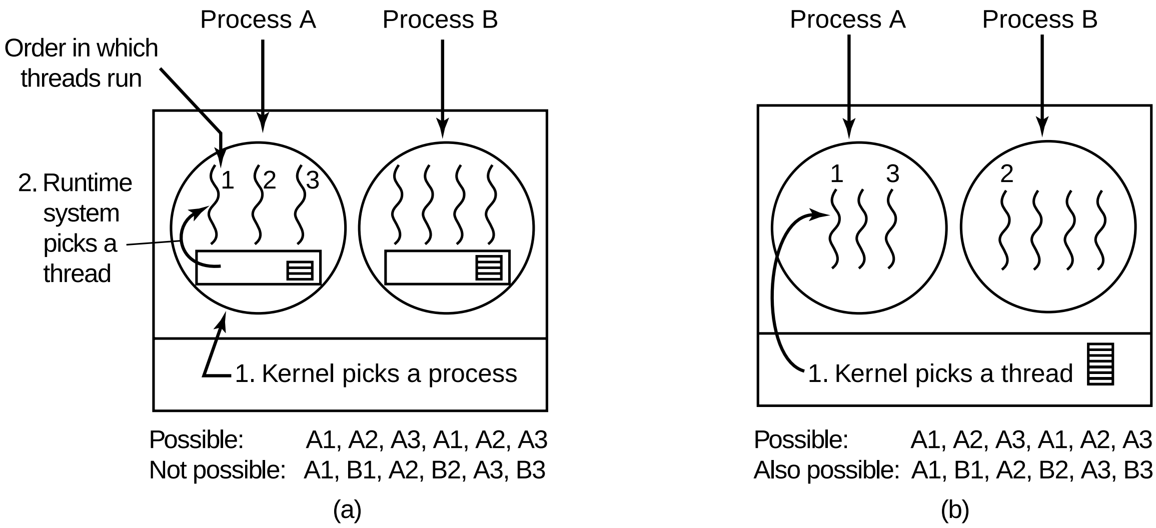


Fig. 2-43. (a) Possible scheduling of user-level threads with a 50-msec process quantum and threads that run 5 msec per CPU burst. (b) Possible scheduling of kernel-level threads with the same characteristics as (a).