Processes and Threads

Chapter 2

The Process Model (1)

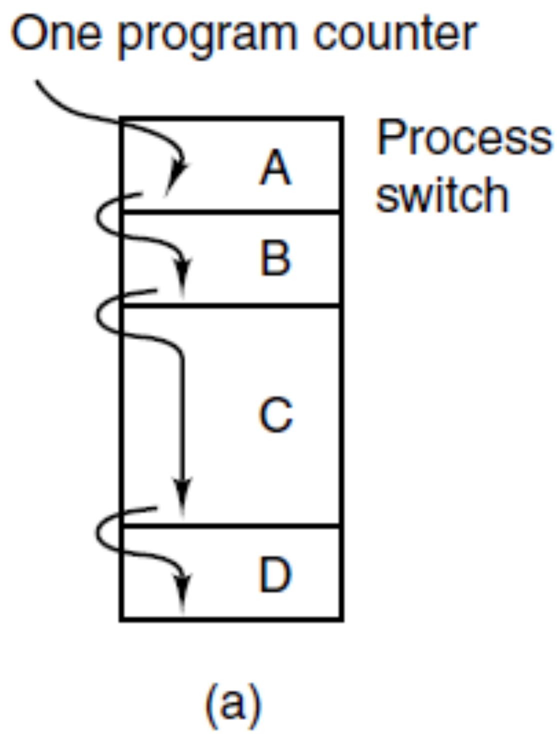


Figure 2-1. (a) Multiprogramming of four programs.

The Process Model (2)

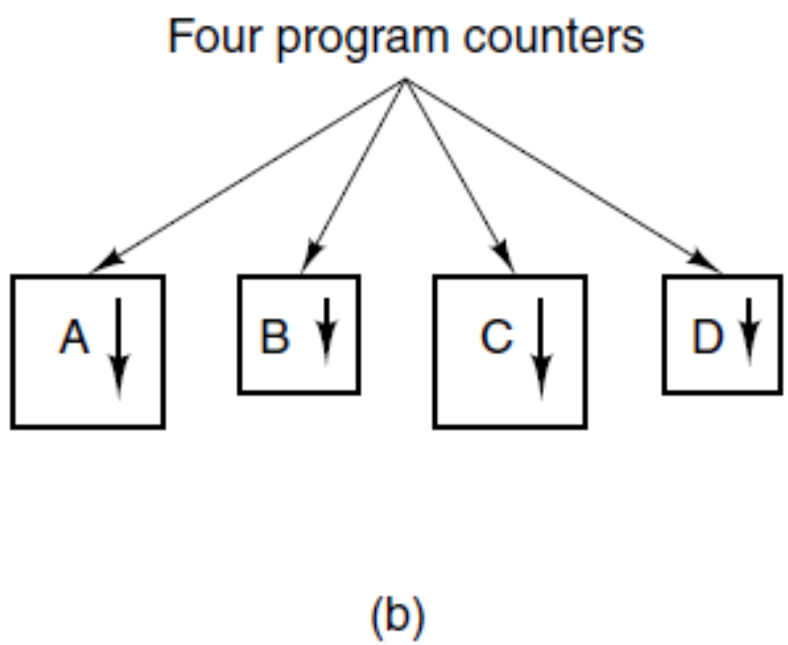


Figure 2-1. (b) Conceptual model of four independent, sequential processes.

The Process Model (3)

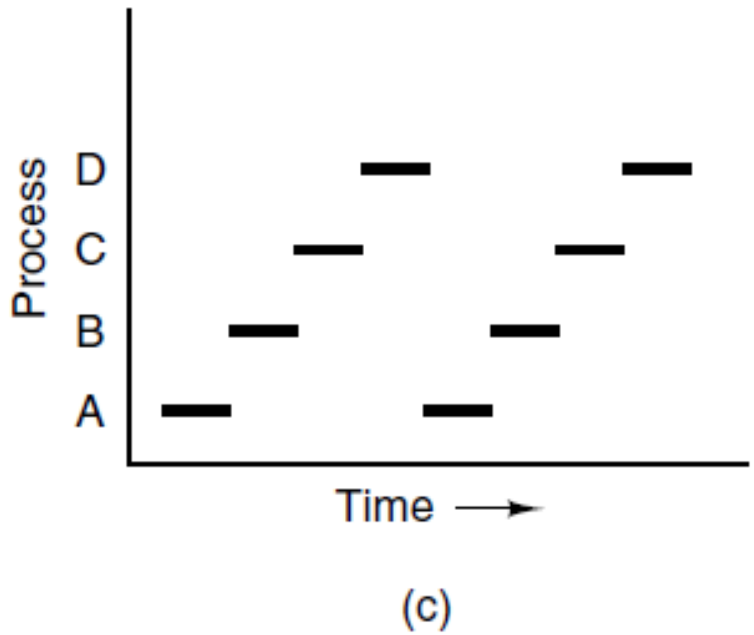


Figure 2-1. (c) Only one program is active at once.

Process Creation

Four 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

Typical conditions which terminate a process:

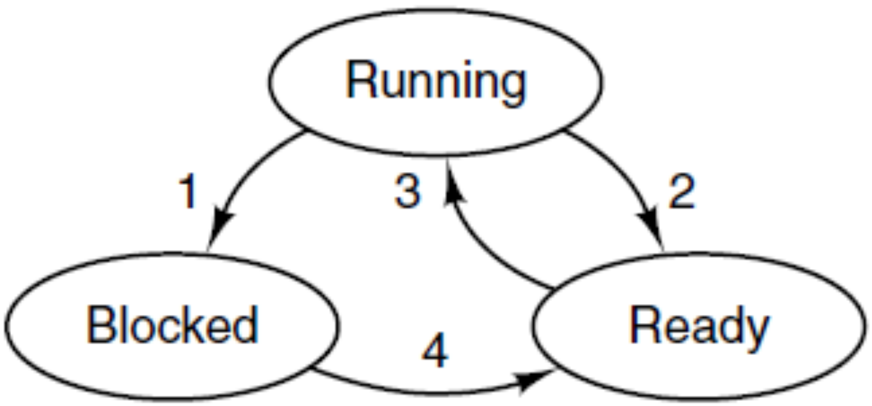
1. Normal exit (voluntary).
2. Error exit (voluntary).
3. Fatal error (involuntary).
4. Killed by another process (involuntary).

Process States (1)

Three states a process may be in:

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)

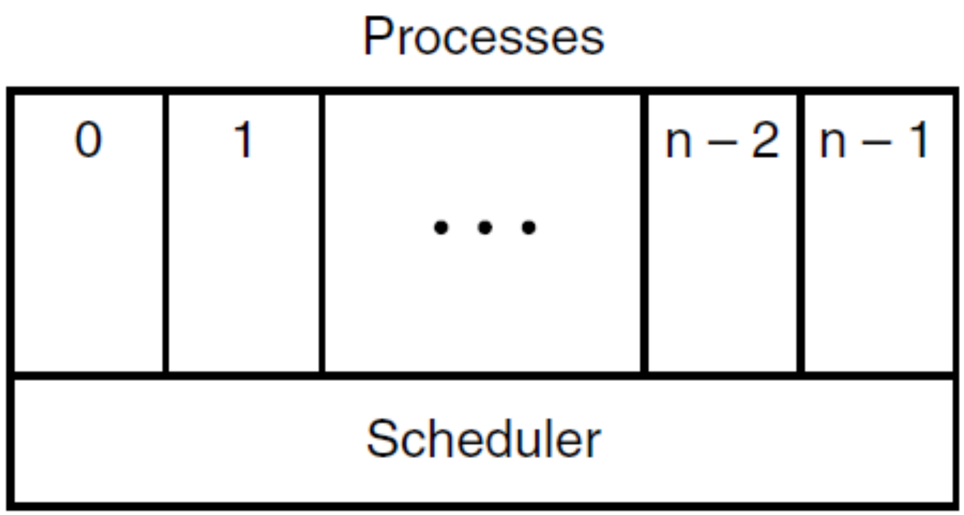


Figure 2-3. The lowest layer of a process-structured operating system handles interrupts and scheduling. Above that layer are sequential processes.

Implementation of Processes (1)

Process management	Memory management	File management
Registers	Pointer to text segment info	Root directory
Program counter	Pointer to data segment info	Working directory
Program status word	Pointer to stack segment info	File descriptors
Stack pointer		User ID
Process state		Group ID
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		

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

Implementation of Processes (2)

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.

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

Modeling Multiprogramming

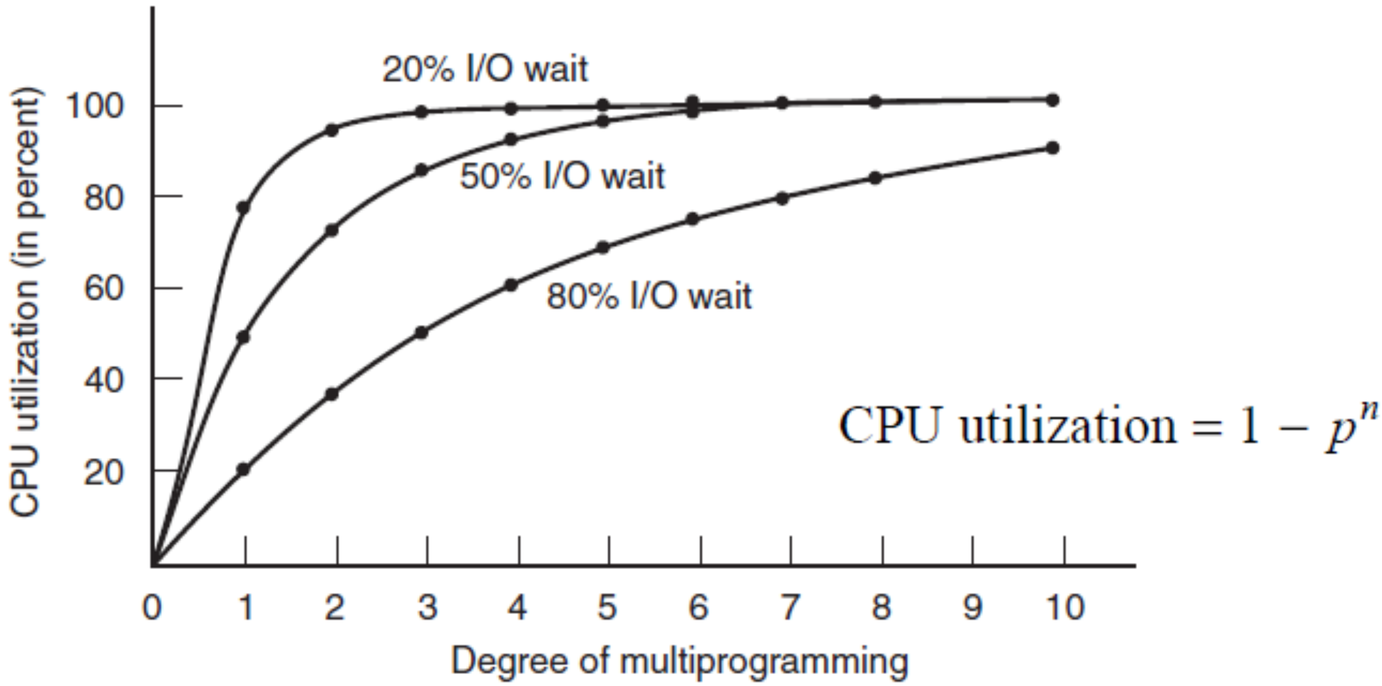


Figure 2-6. CPU utilization as a function of the number of processes in memory.

Thread Usage (1)

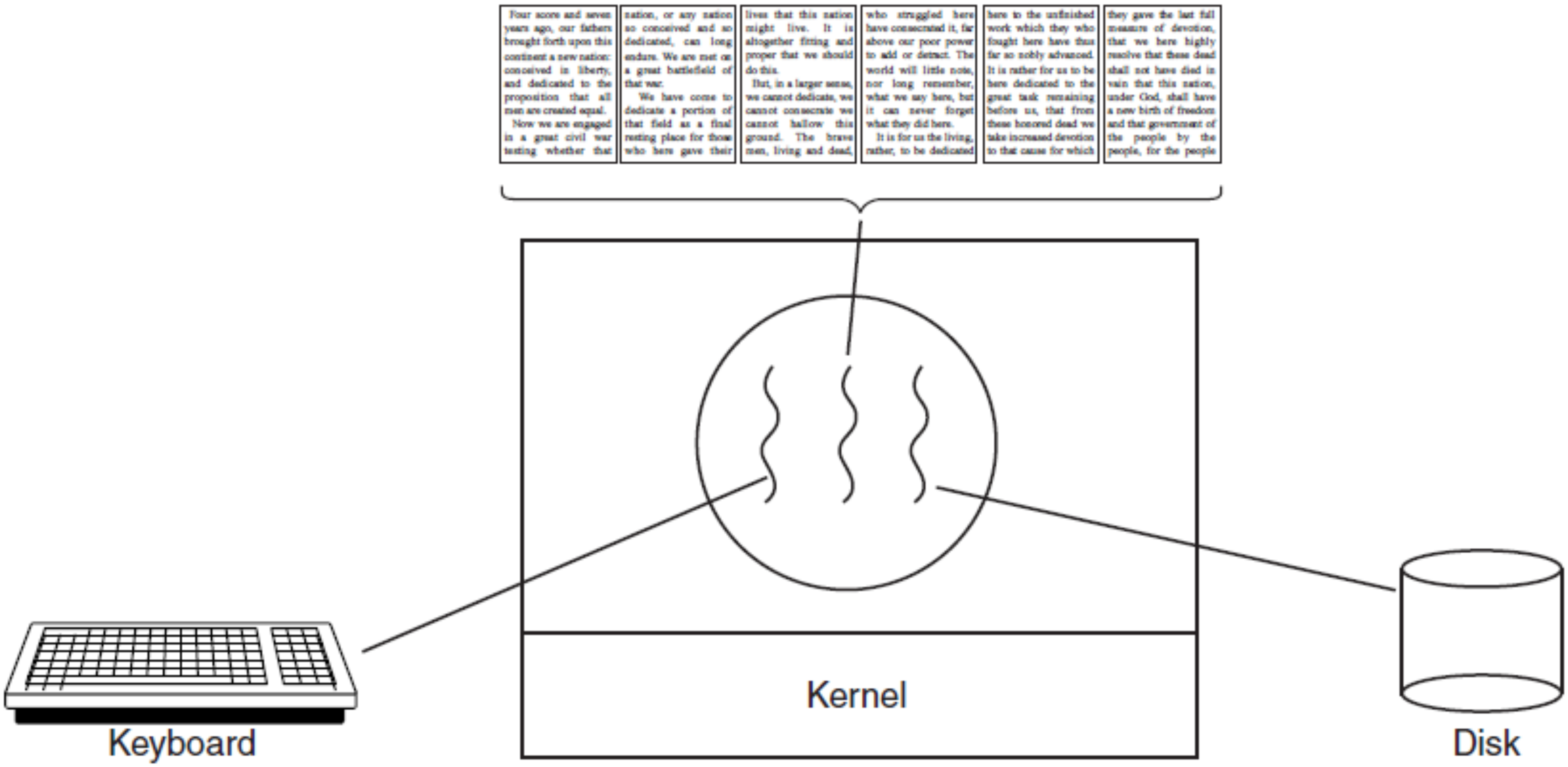


Figure 2-7. A word processor with three threads.

Thread Usage (2)

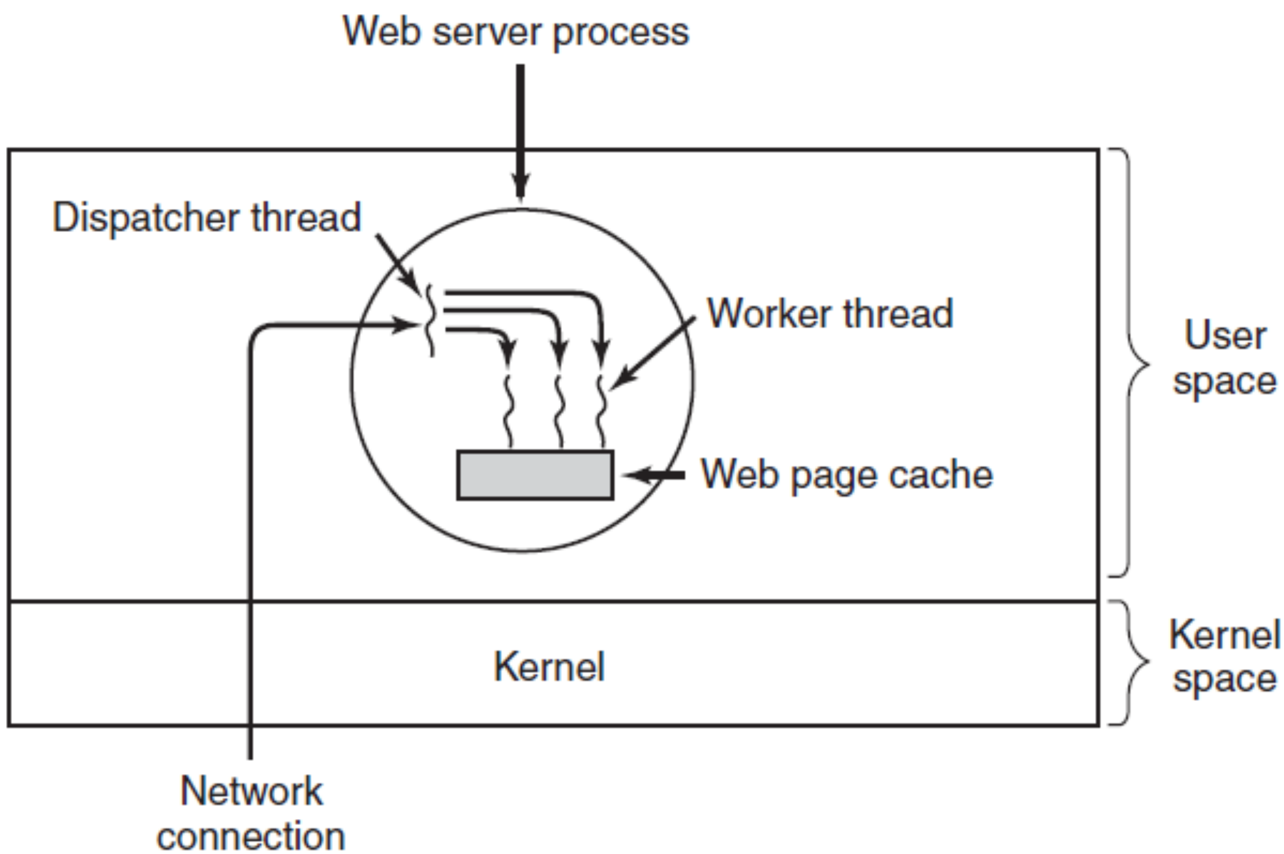


Figure 2-8. A multithreaded Web server.

Thread Usage (3)

```
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)

Figure 2-9. A rough outline of the code for Fig. 2-8.
(a) Dispatcher thread. (b) Worker thread.

Thread Usage (4)

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

Figure 2-10. Three ways to construct a server.

The Classical Thread Model (1)

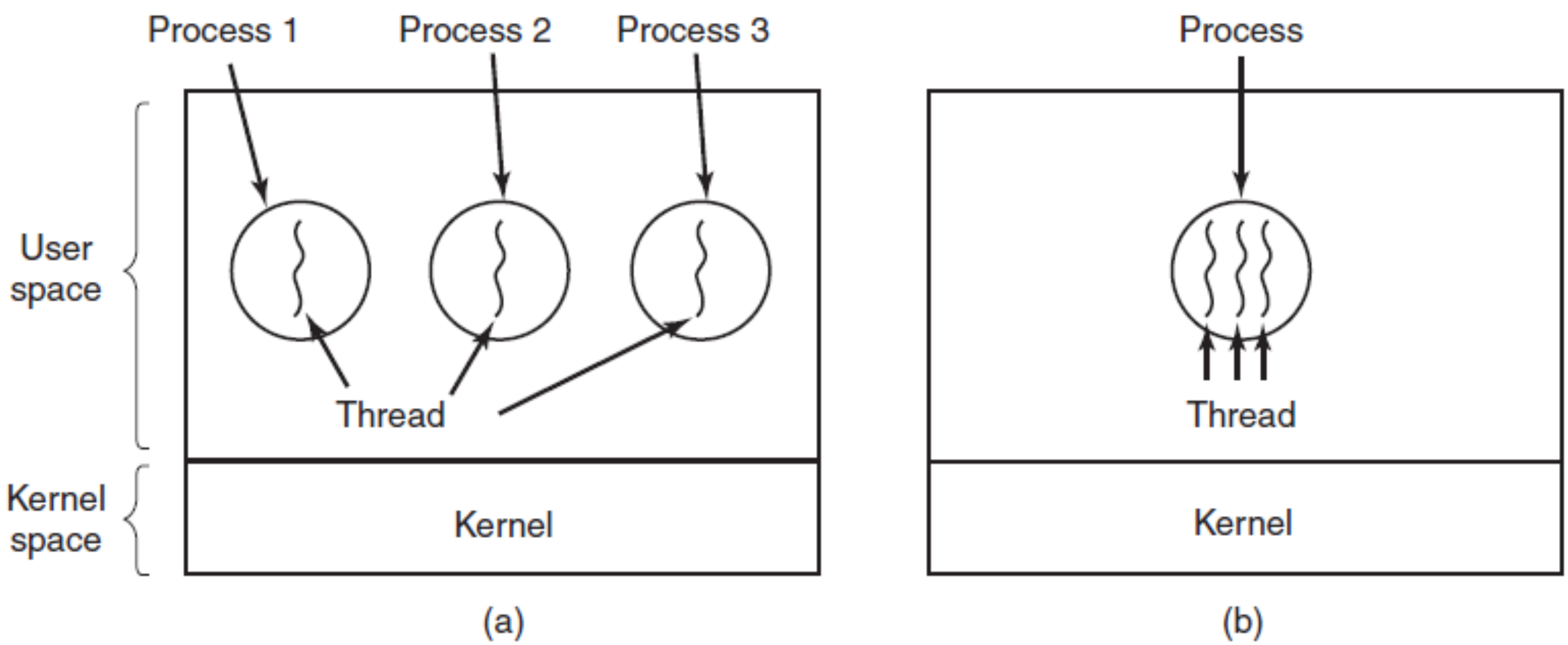


Figure 2-11. (a) Three processes each with one thread.
(b) One process with three threads.

The Classical Thread Model (2)

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 2-12. The first column lists some items shared by all threads in a process. The second one lists some items private to each thread.

The Classical Thread Model (3)

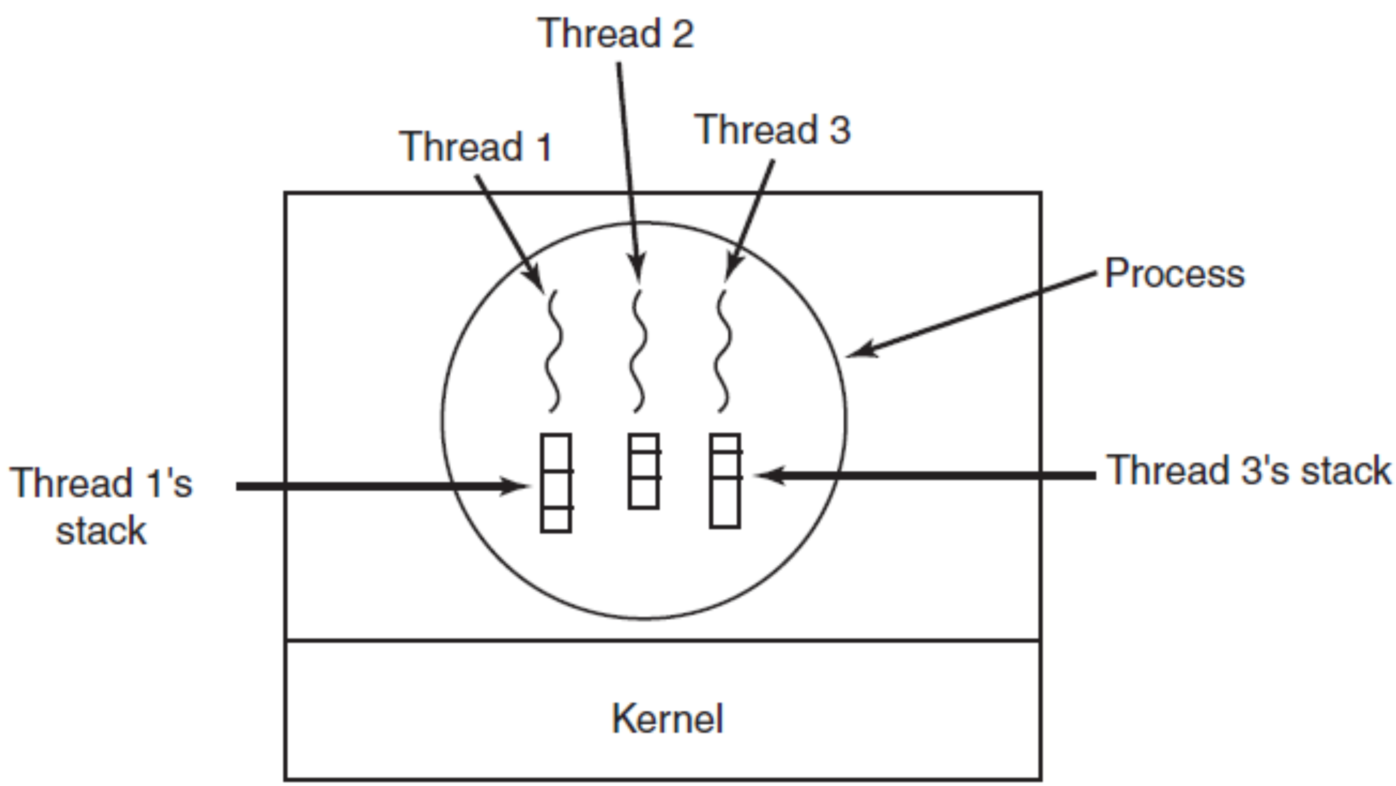


Figure 2-13. Each thread has its own stack.

POSIX Threads (1)

Thread call	Description
Pthread_create	Create a new thread
Pthread_exit	Terminate the calling thread
Pthread_join	Wait for a specific thread to exit
Pthread_yield	Release the CPU to let another thread run
Pthread_attr_init	Create and initialize a thread's attribute structure
Pthread_attr_destroy	Remove a thread's attribute structure

Figure 2-14. Some of the Pthreads function calls.

POSIX Threads (2)

```
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>

#define NUMBER_OF_THREADS    10

void *print_hello_world(void *tid)
{
      /* This function prints the thread's identifier and then exits. */
      printf("Hello World. Greetings from thread %d\n", tid);
      pthread_exit(NULL);
}

int main(int argc, char *argv[])
{
      /* The main program creates 10 threads and then exits. */
      pthread_t threads[NUMBER_OF_THREADS];
      int status, i;

      for(i=0; i < NUMBER_OF_THREADS; i++) {
            printf("Main here. Creating thread %d\n", i);
            status = pthread_create(&threads[i], NULL, print_hello_world, (void *)i);
```

Figure 2-15. An example program using threads.

POSIX Threads (3)

```
    int status, i;

    for(i=0; i < NUMBER_OF_THREADS; i++) {
        printf("Main here. Creating thread %d\n", i);
        status = pthread_create(&threads[i], NULL, print_hello_world, (void *)i);

        if (status != 0) {
            printf("Oops. pthread_create returned error code %d\n", status);
            exit(-1);
        }
    }
    exit(NULL);
}
```

Figure 2-15. An example program using threads.

Implementing Threads in User Space

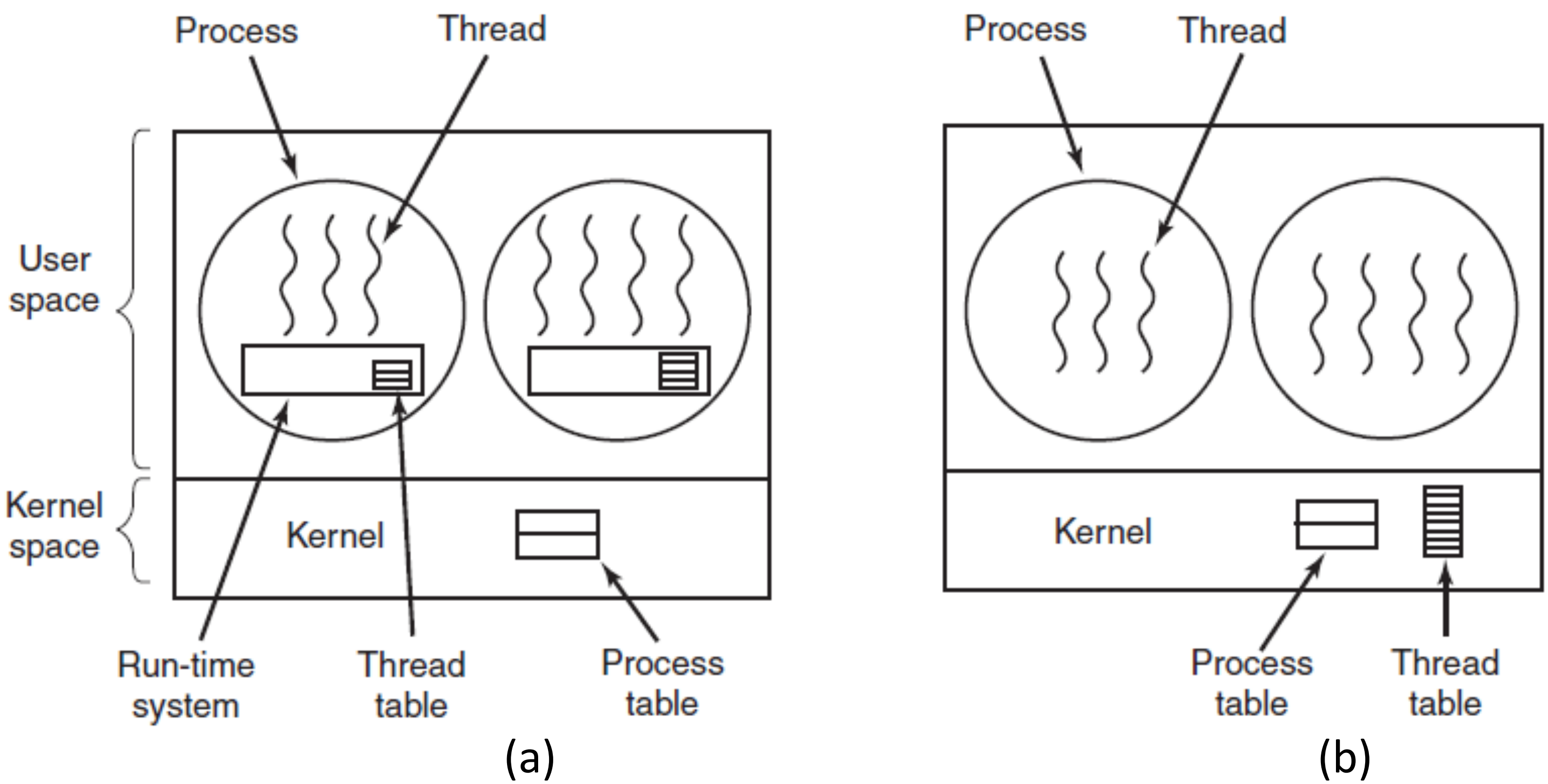


Figure 2-16. (a) A user-level threads package.
(b) A threads package managed by the kernel.

Hybrid Implementations

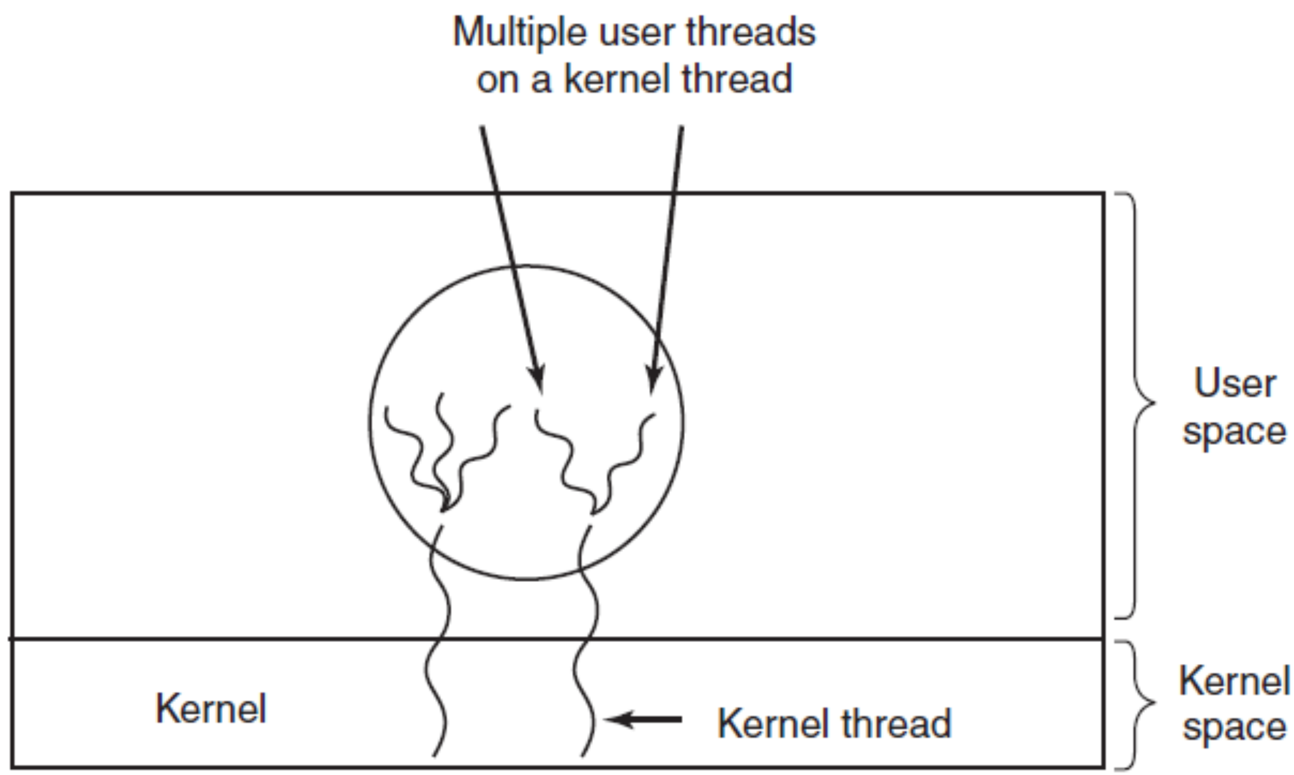


Figure 2-17. Multiplexing user-level threads onto kernel-level threads.

Pop-Up Threads

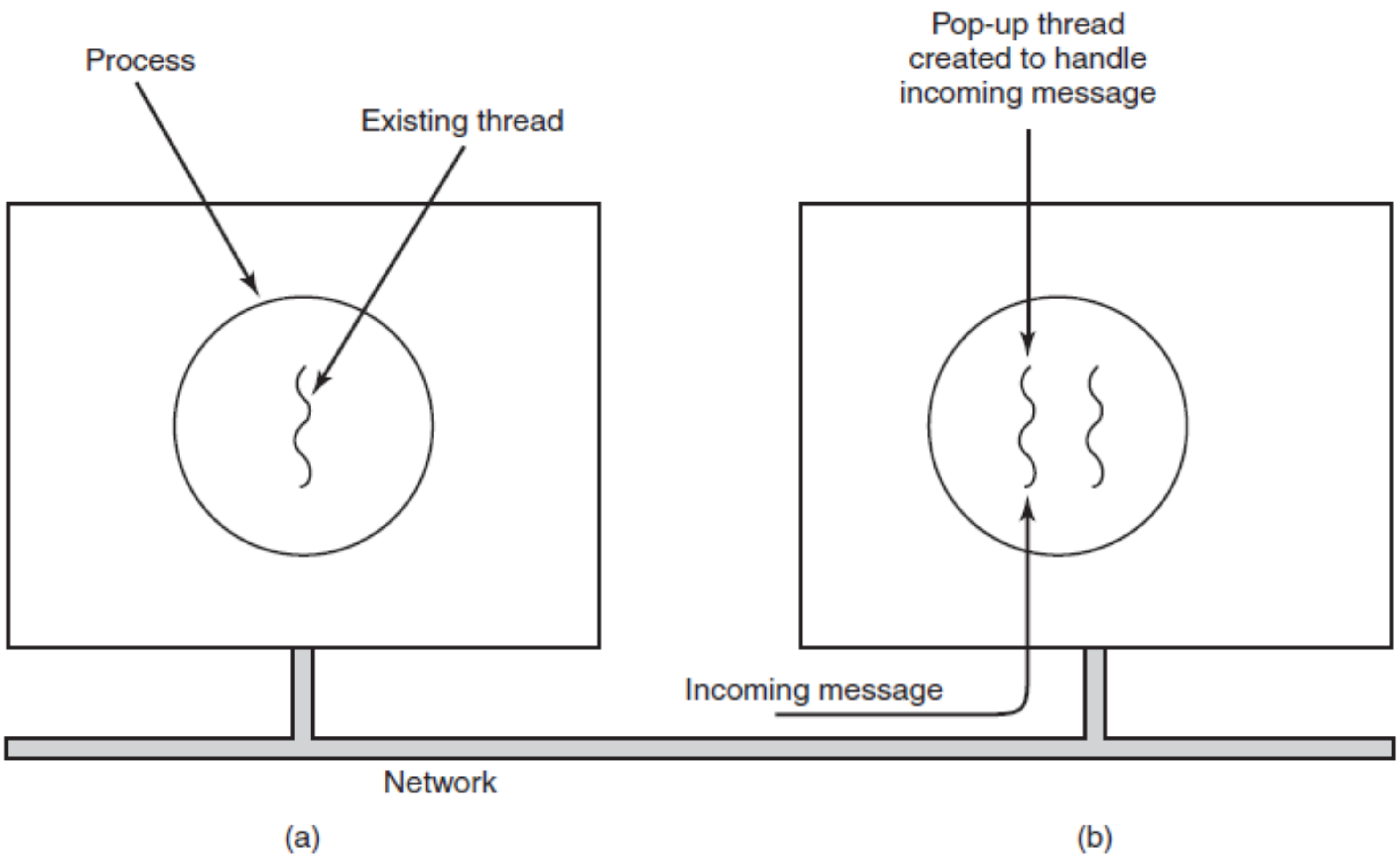


Figure 2-18. Creation of a new thread when a message arrives. (a) Before the message arrives. (b) After the message arrives.

Making Single-Threaded Code Multithreaded (1)

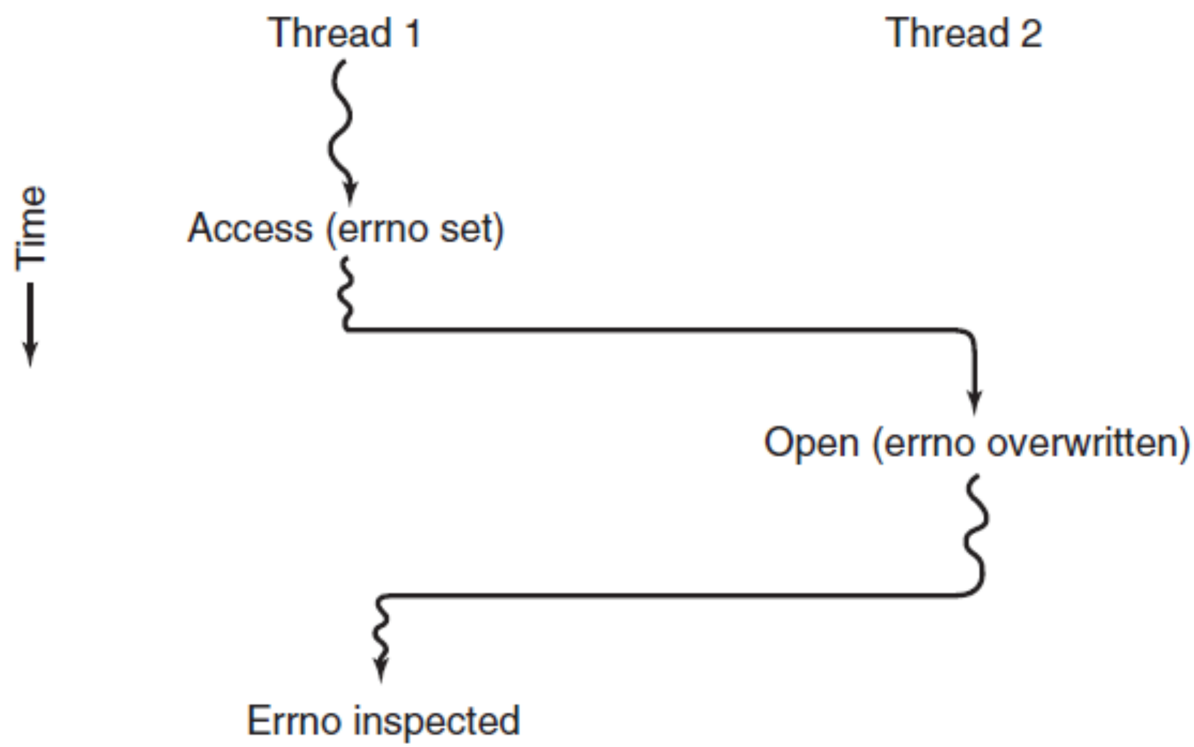


Figure 2-19. Conflicts between threads over the use of a global variable.

Making Single-Threaded Code Multithreaded (2)

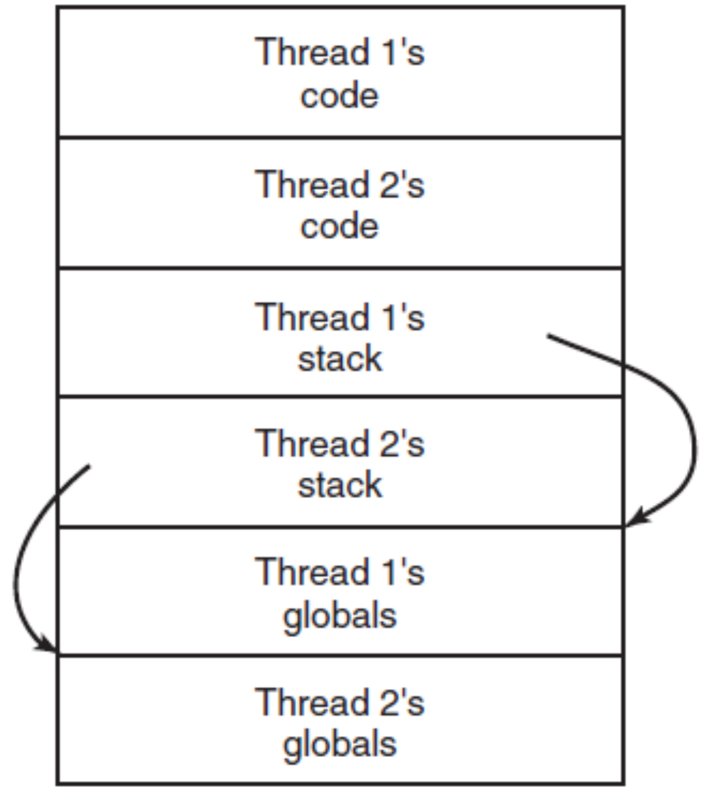


Figure 2-20. Threads can have private global variables.

Race Conditions

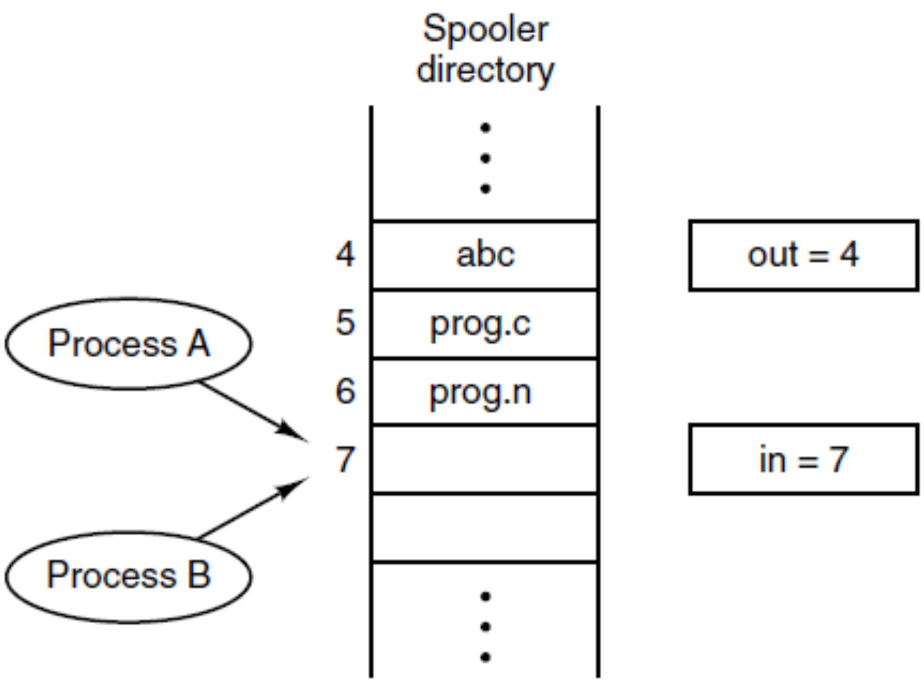


Figure 2-21. Two processes want to access shared memory at the same time.

Critical Regions (1)

Requirements 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.

Critical Regions (2)

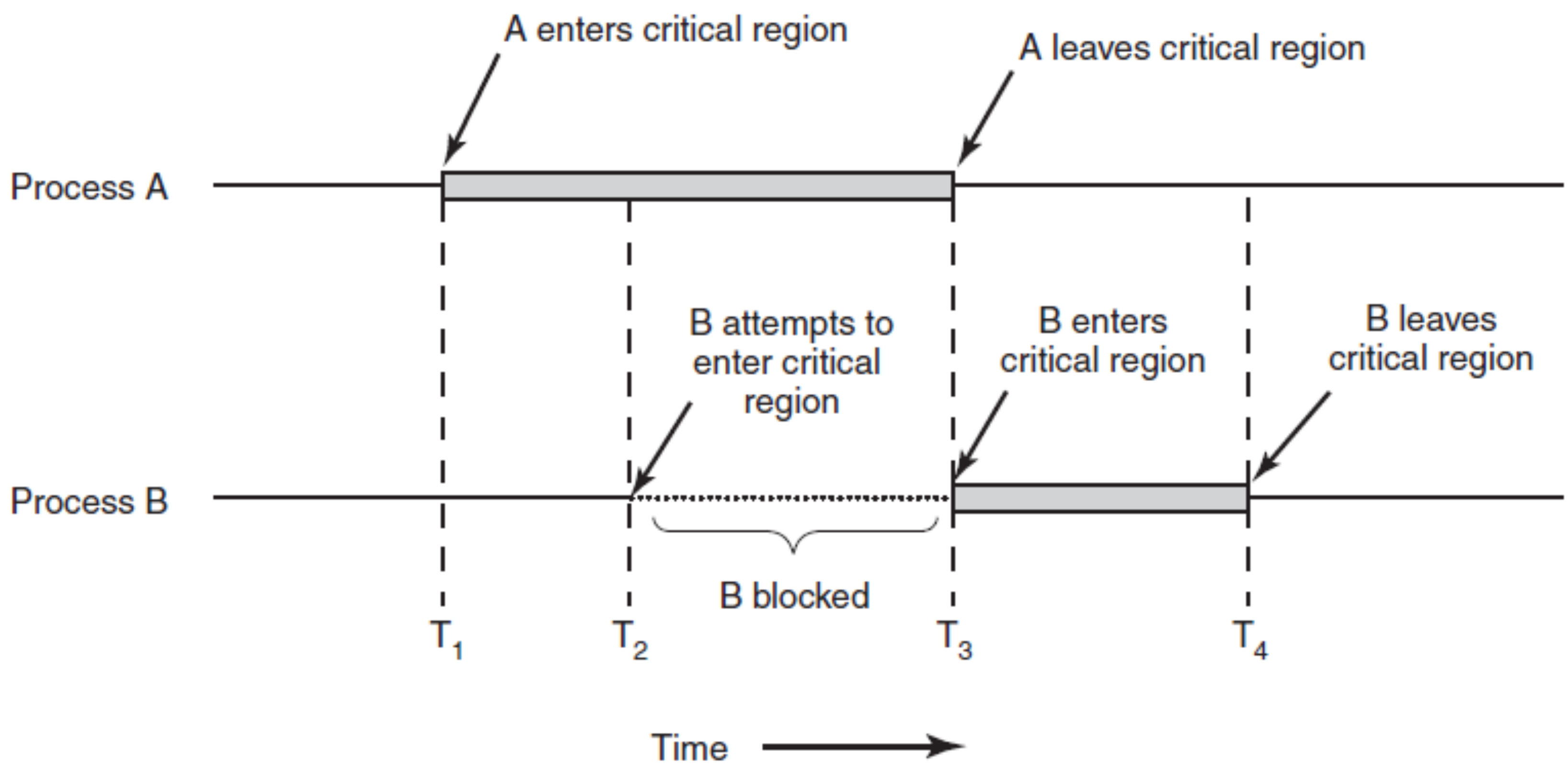


Figure 2-22. Mutual exclusion using critical regions.

Mutual Exclusion with Busy Waiting:
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 2-23. 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.

Mutual Exclusion with Busy Waiting:

Peterson's Solution

```
#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 2-24. Peterson's solution for achieving mutual exclusion.

Mutual Exclusion with Busy Waiting:
The TSL Instruction (1)

```
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 nonzero, 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 2-25. Entering and leaving a critical region using the TSL instruction.

Mutual Exclusion with Busy Waiting:
The TSL Instruction (2)

```
enter_region:
        MOVE REGISTER,#1               | put a 1 in the register
        XCHG REGISTER,LOCK             | swap the contents of the register and lock variable
        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 2-26. Entering and leaving a critical region using the XCHG instruction

Sleep and Wakeup
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? */
     }
}


void consumer(void)
{
```

Figure 2-27. The producer-consumer problem with a fatal race condition.

Sleep and Wakeup
The Producer-Consumer Problem (2)

```
        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 */
    }
}
```

Figure 2-27. The producer-consumer problem with a fatal race condition.

Semaphores (1)

```
#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)
```

Figure 2-28. The producer-consumer problem using semaphores.

Semaphores (2)

```
        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 */
    }
}
```

Figure 2-28. The producer-consumer problem using semaphores.

Mutexes

```
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
ok:     RET                         | return to caller; critical region entered

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

Figure 2-29. Implementation of *mutex_lock* and *mutex_unlock*.

Mutexes in Pthreads (1)

Thread call	Description
Pthread_mutex_init	Create a mutex
Pthread_mutex_destroy	Destroy an existing mutex
Pthread_mutex_lock	Acquire a lock or block
Pthread_mutex_trylock	Acquire a lock or fail
Pthread_mutex_unlock	Release a lock

Figure 2-30. Some of the Pthreads calls relating to mutexes.

Mutexes in Pthreads (2)

Thread call	Description
Pthread_cond_init	Create a condition variable
Pthread_cond_destroy	Destroy a condition variable
Pthread_cond_wait	Block waiting for a signal
Pthread_cond_signal	Signal another thread and wake it up
Pthread_cond_broadcast	Signal multiple threads and wake all of them

Figure 2-31. Some of the Pthreads calls relating to condition variables.

Mutexes in Pthreads (3)

```
#include <stdio.h>
#include <pthread.h>
#define MAX 1000000000                          /* how many numbers to produce */
pthread_mutex_t the_mutex;
pthread_cond_t condc, condp;                    /* used for signaling */
int buffer = 0;                                 /* buffer used between producer and consumer */

void *producer(void *ptr)                       /* produce data */
{       int i;

        for (i= 1; i <= MAX; i++) {
                pthread_mutex_lock(&the_mutex);     /* get exclusive access to buffer */
                while (buffer != 0) pthread_cond_wait(&condp, &the_mutex);
                buffer = i;                         /* put item in buffer */
                pthread_cond_signal(&condc);        /* wake up consumer */
                pthread_mutex_unlock(&the_mutex);/* release access to buffer */
        }
        pthread_exit(0);
}
```

Figure 2-32. Using threads to solve the producer-consumer problem.

Mutexes in Pthreads (4)

```
        pthread_exit(0);
}

void *consumer(void *ptr)                       /* consume data */
{       int i;

        for (i = 1; i <= MAX; i++) {
                pthread_mutex_lock(&the_mutex);   /* get exclusive access to buffer */
                while (buffer ==0 ) pthread_cond_wait(&condc, &the_mutex);
                buffer = 0;                       /* take item out of buffer */
                pthread_cond_signal(&condp);      /* wake up producer */
                pthread_mutex_unlock(&the_mutex); /* release access to buffer */
        }
        pthread_exit(0);
}

int main(int argc, char **argv)
```

Figure 2-32. Using threads to solve the producer-consumer problem.

Mutexes in Pthreads (5)

```
        pthread_exit(0);
    }

    int main(int argc, char **argv)
    {
        pthread_t pro, con;
        pthread_mutex_init(&the_mutex, 0);
        pthread_cond_init(&condc, 0);
        pthread_cond_init(&condp, 0);
        pthread_create(&con, 0, consumer, 0);
        pthread_create(&pro, 0, producer, 0);
        pthread_join(pro, 0);
        pthread_join(con, 0);
        pthread_cond_destroy(&condc);
        pthread_cond_destroy(&condp);
        pthread_mutex_destroy(&the_mutex);
    }
```

Figure 2-32. Using threads to solve the producer-consumer problem.

Monitors (1)

```
monitor example
        integer i;
        condition c;

        procedure producer( );
        .
        .
        .
        end;

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

Figure 2-33. 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 2-34. 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 2-34. An outline of the producer-consumer problem with monitors. Only one monitor procedure at a time is active. The buffer has N slots.

Monitors (4)

```
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 {
```

Figure 2-35. A solution to the producer-consumer problem in Java.

Monitors (5)

```
    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) {
```

Figure 2-35. A solution to the producer-consumer problem in Java.

Monitors (6)

```
        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) {};}
}
```

Figure 2-35. A solution to the producer-consumer problem in Java.

The Producer-Consumer Problem with 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 */
        }
}

void consumer(void)
```

Figure 2-36. The producer-consumer problem with *N* messages.

The Producer-Consumer Problem with Message Passing (2)

```
        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 */
    }
}
```

Figure 2-36. The producer-consumer problem with *N* messages.

Barriers

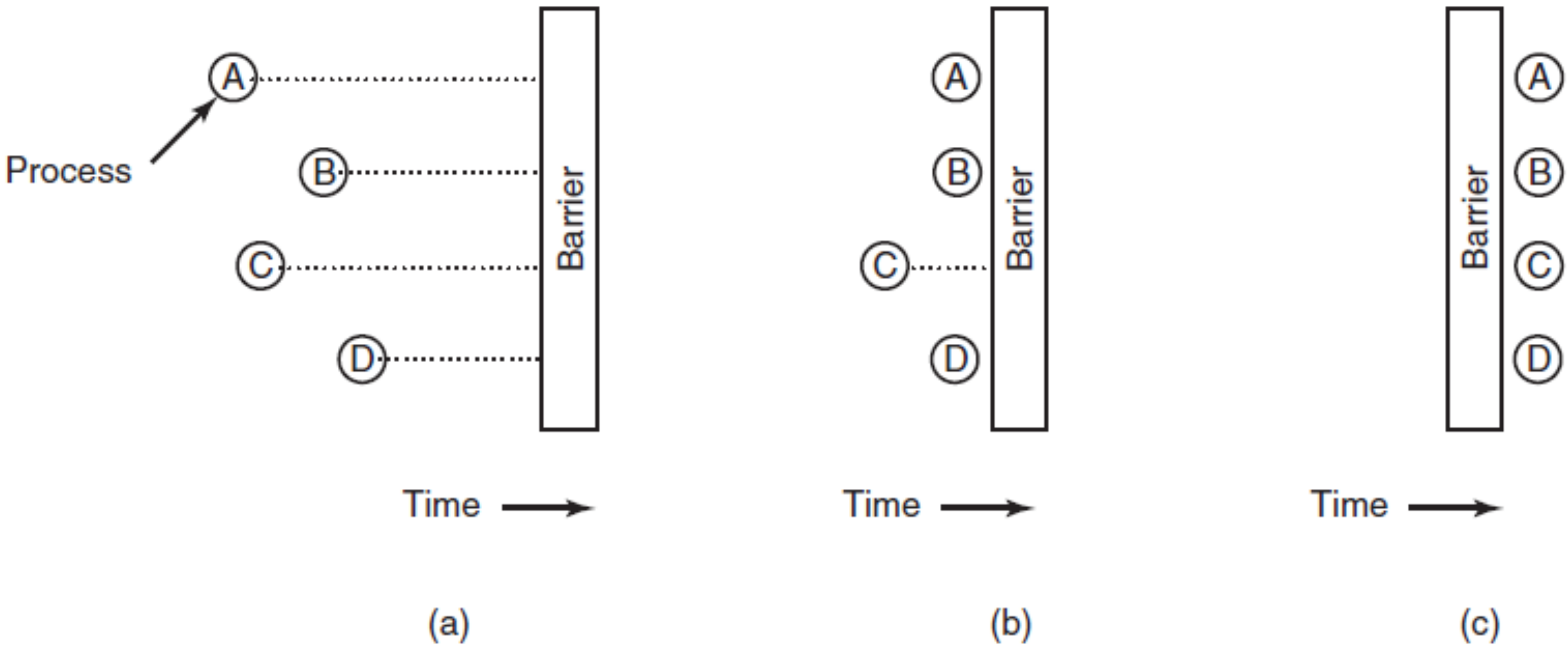


Figure 2-37. Use of a barrier. (a) Processes approaching a barrier. (b) All processes but one blocked at the barrier. (c) When the last process arrives at the barrier, all of them are let through.

Avoiding Locks: Read-Copy-Update (1)

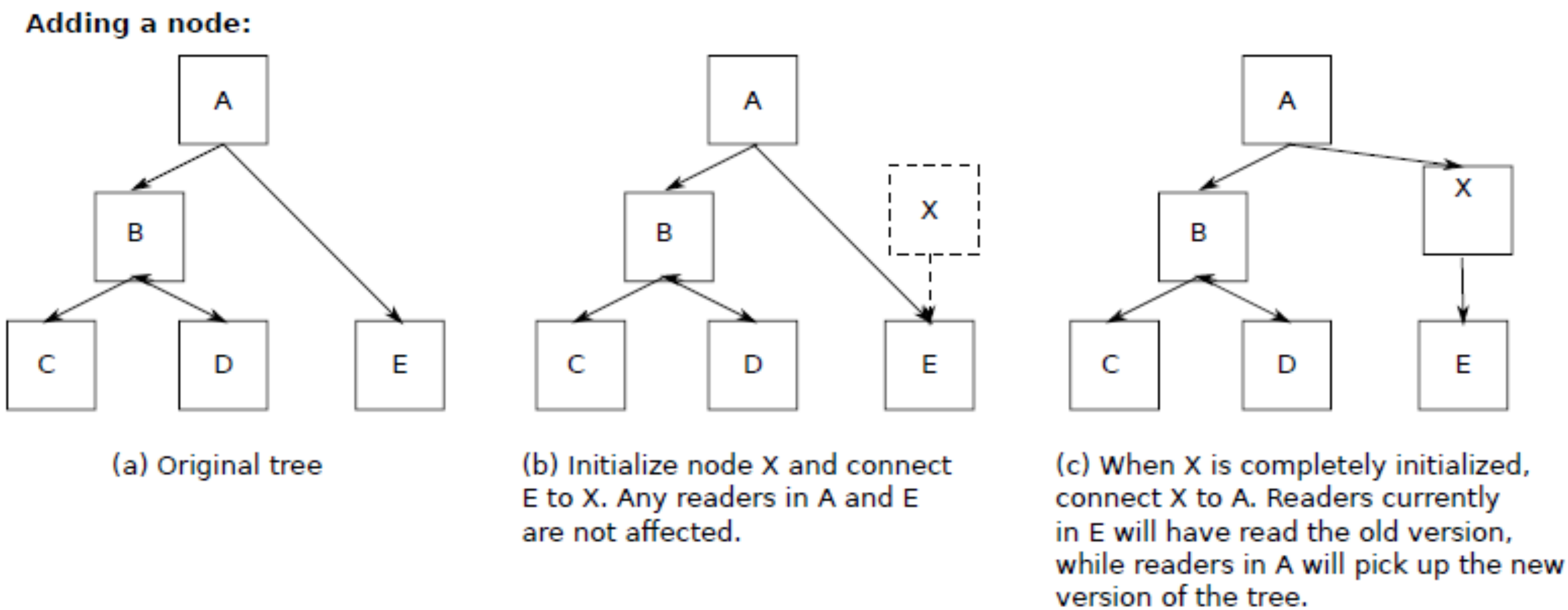
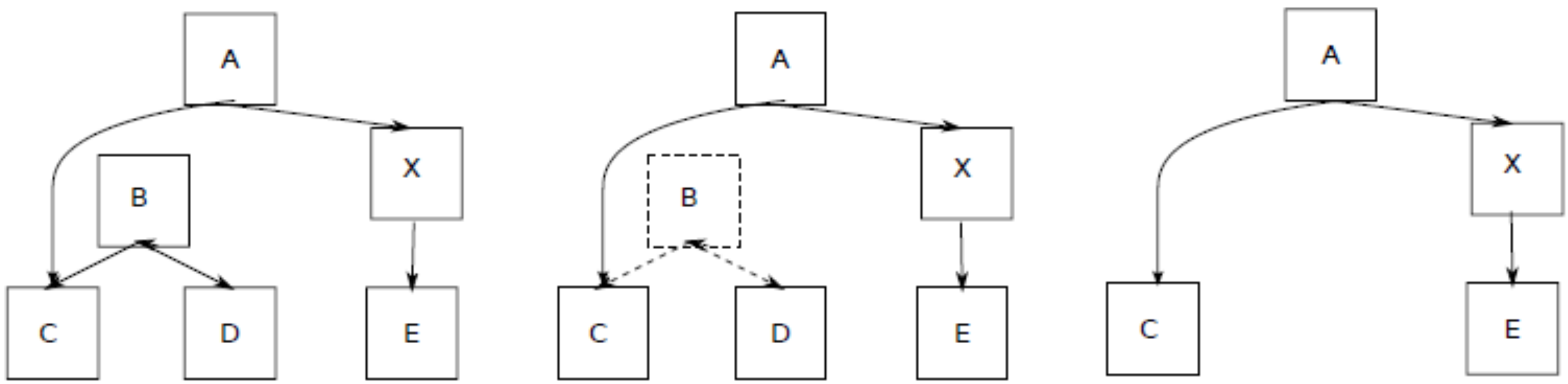


Figure 2-38. Read-Copy-Update: inserting a node in the tree and then removing a branch—all without locks

Avoiding Locks: Read-Copy-Update (2)

Removing nodes:



(d) Decouple B from A. Note that there may still be readers in B. All readers in B will see the old version of the tree, while all readers currently in A will see the new version.

(e) Wait until we are sure that all readers have left B and C. These nodes cannot be accessed by anymore.

(f) Now we can safely remove B and D

Figure 2-38. Read-Copy-Update: inserting a node in the tree and then removing a branch—all without locks

Introduction to Scheduling
Process Behavior

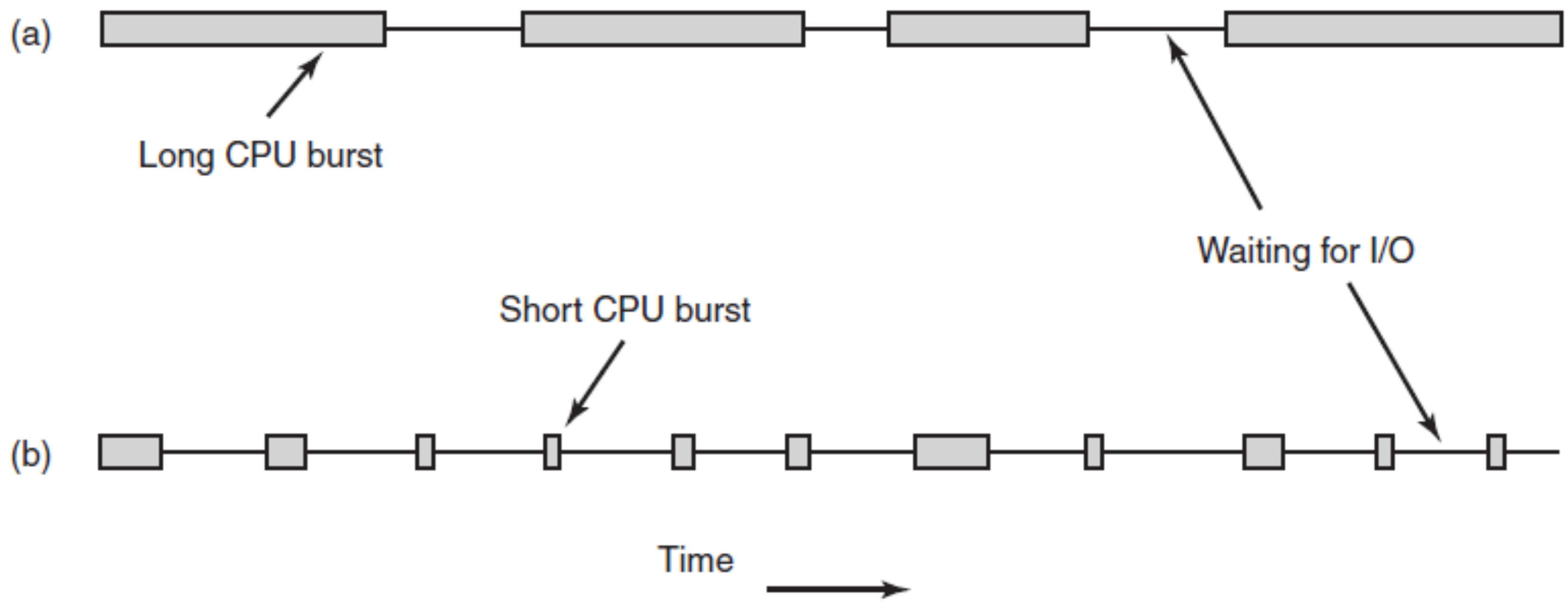


Figure 2-39. Bursts of CPU usage alternate with periods of waiting for I/O. (a) A CPU-bound process. (b) An I/O-bound process.

Categories of Scheduling Algorithms

1. Batch.
2. Interactive.
3. Real time.

Scheduling Algorithm Goals

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 2-40. Some goals of the scheduling algorithm under different circumstances.

Scheduling in Batch Systems

- First-Come First-Served
- Shortest Job First
- Shortest Remaining Time Next

Shortest Job First

Figure 2-41. An example of shortest job first scheduling.
(a) Running four jobs in the original order.
(b) Running them in shortest job first order.

Scheduling in Interactive Systems

- Round-Robin Scheduling
- Priority Scheduling
- Multiple Queues
- Shortest Process Next
- Guaranteed Scheduling
- Lottery Scheduling
- Fair-Share Scheduling

Round-Robin Scheduling

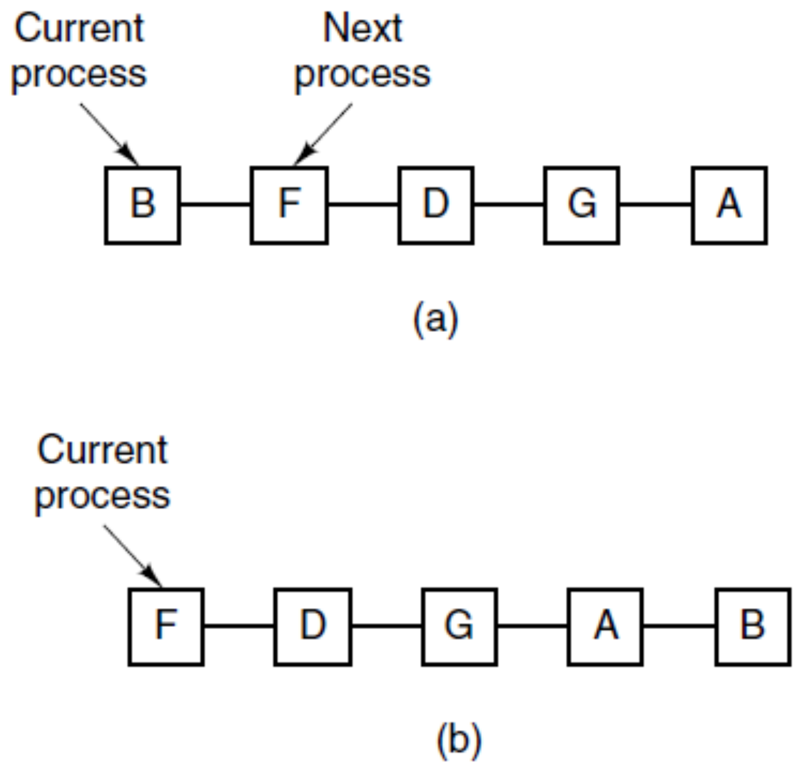


Figure 2-42. Round-robin scheduling. (a) The list of runnable processes. (b) The list of runnable processes after *B* uses up its quantum.

Priority Scheduling

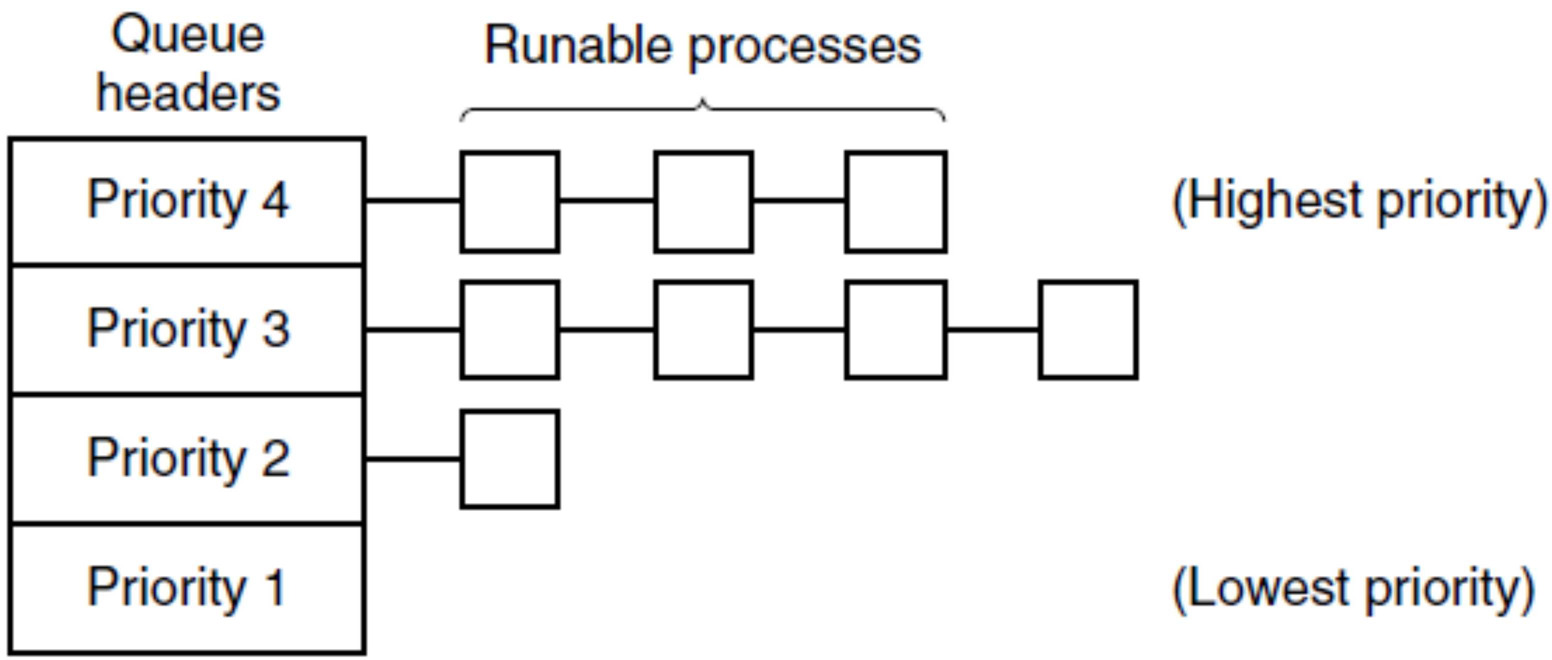


Figure 2-43. A scheduling algorithm with four priority classes.

Scheduling in Real-Time Systems

- Time plays an essential role
- Categories
  - Hard real time
  - Soft real time
  - Periodic or aperiodic
- Schedulable satisfies

$$\sum_{i=1}^{m} \frac{C_i}{P_i} \leq 1$$

Thread Scheduling (1)

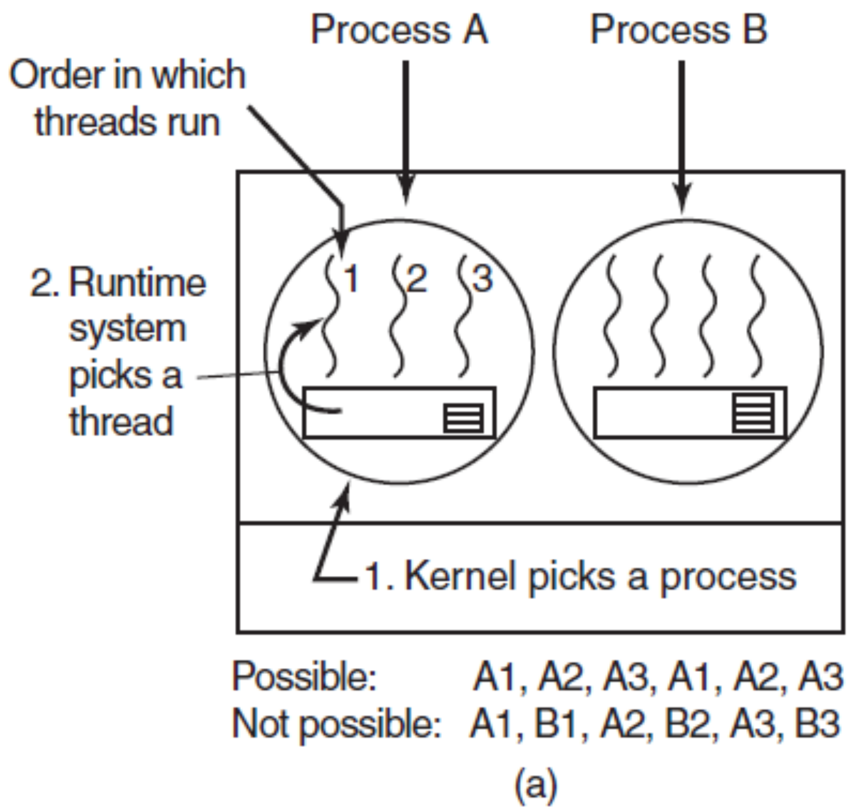


Figure 2-44. (a) Possible scheduling of user-level threads with a 50-msec process quantum and threads that run 5 msec per CPU burst.

Thread Scheduling (2)

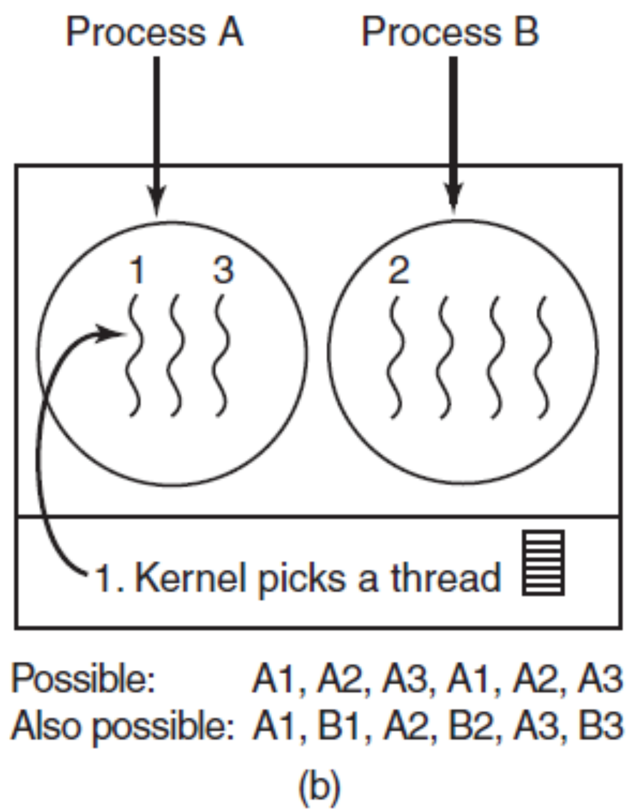


Figure 2-44. (b) Possible scheduling of kernel-level threads with the same characteristics as (a).

The Dining Philosophers Problem (1)

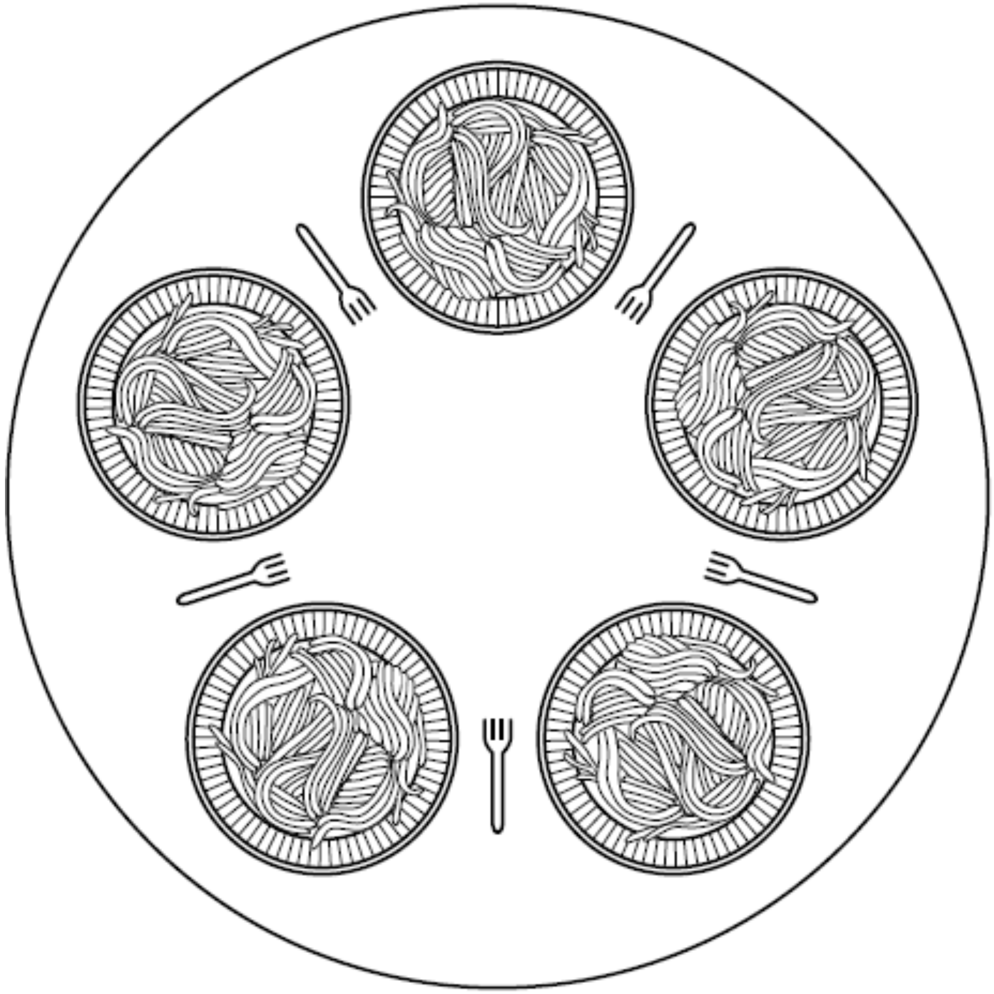


Figure 2-45. Lunch time in the Philosophy Department.

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 2-46. 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 */

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 */
     }
}
```

Figure 2-47. A solution to the dining philosophers problem.

The Dining Philosophers Problem (4)

```
        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 */
```

Figure 2-47. 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 2-47. 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 */
      }
}

void writer(void)
```

Figure 2-48. A solution to the readers and writers problem.

The Readers and Writers Problem (2)

```
        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 2-48. A solution to the readers and writers problem.

End

Chapter 2