

MODERN OPERATING SYSTEMS

Third Edition

ANDREW S. TANENBAUM

Chapter 2

Processes and Threads

The Process Model

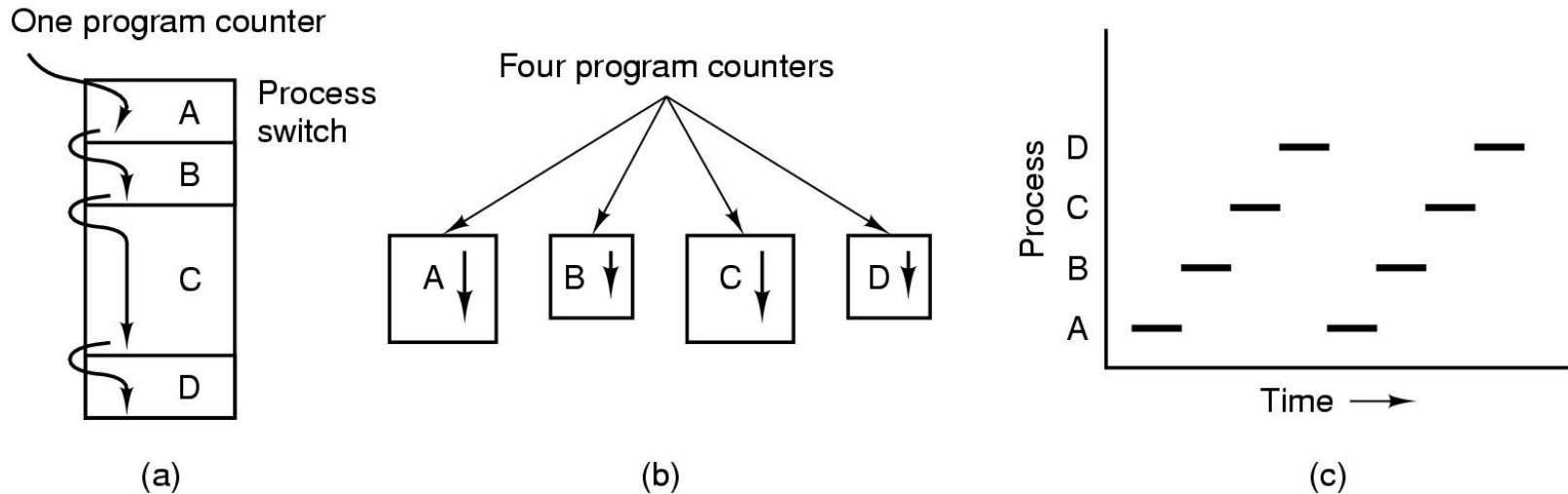


Figure 2-1. (a) Multiprogramming of four programs. (b) Conceptual model of four independent, sequential processes. (c) Only one program is active at once.

Process Creation

Events which cause process creation:

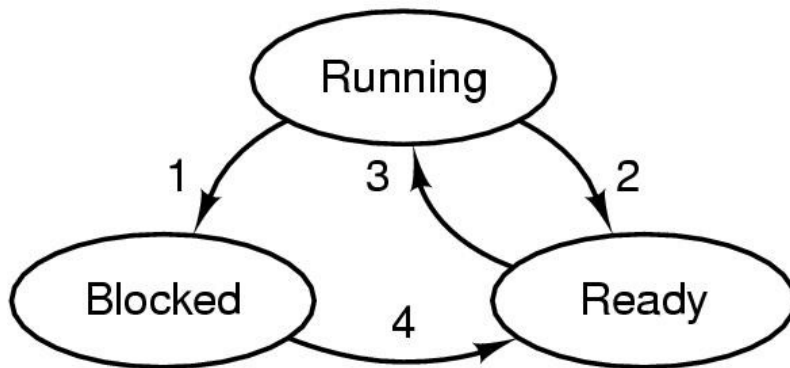
- System initialization.
- Execution of a process creation system call by a running process.
- A user request to create a new process.
- Initiation of a batch job.

Process Termination

Events which cause process termination:

- Normal exit (voluntary).
- Error exit (voluntary).
- Fatal error (involuntary).
- Killed by another process (involuntary).

Process States



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.

Implementation of Processes (1)

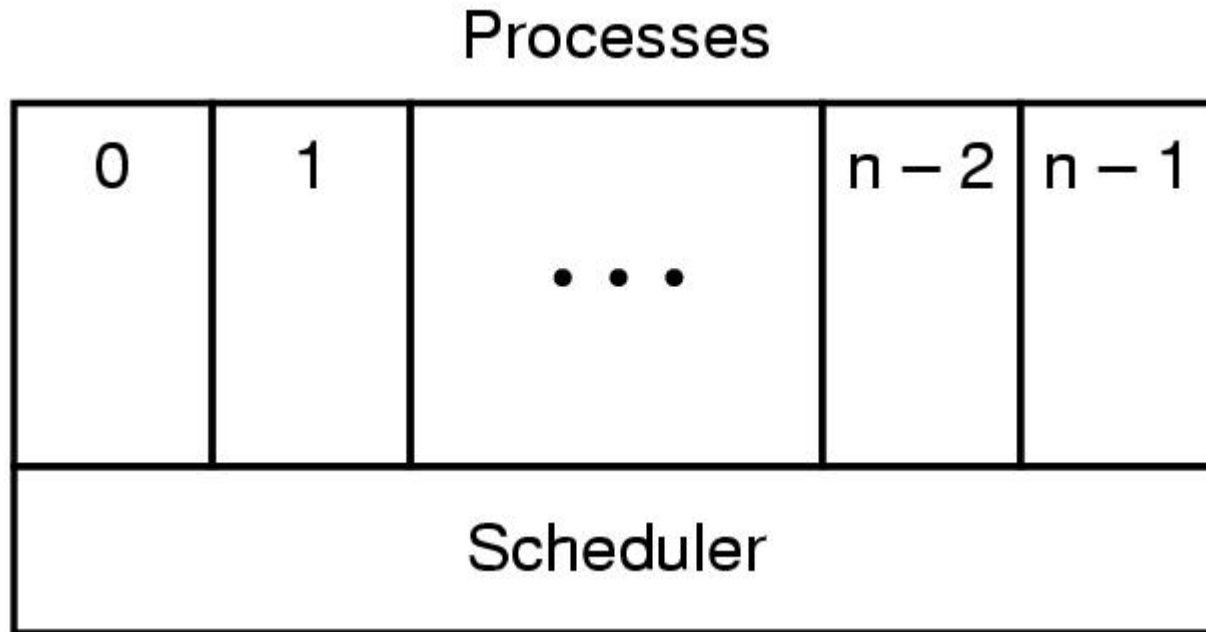


Figure 2-3. The lowest layer of a process-structured operating system handles interrupts and scheduling. Above that layer are sequential processes.

Implementation of Processes (2)

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 info Pointer to data segment info Pointer to stack segment info	Root directory Working directory File descriptors User ID Group ID

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

Implementation of Processes (3)

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

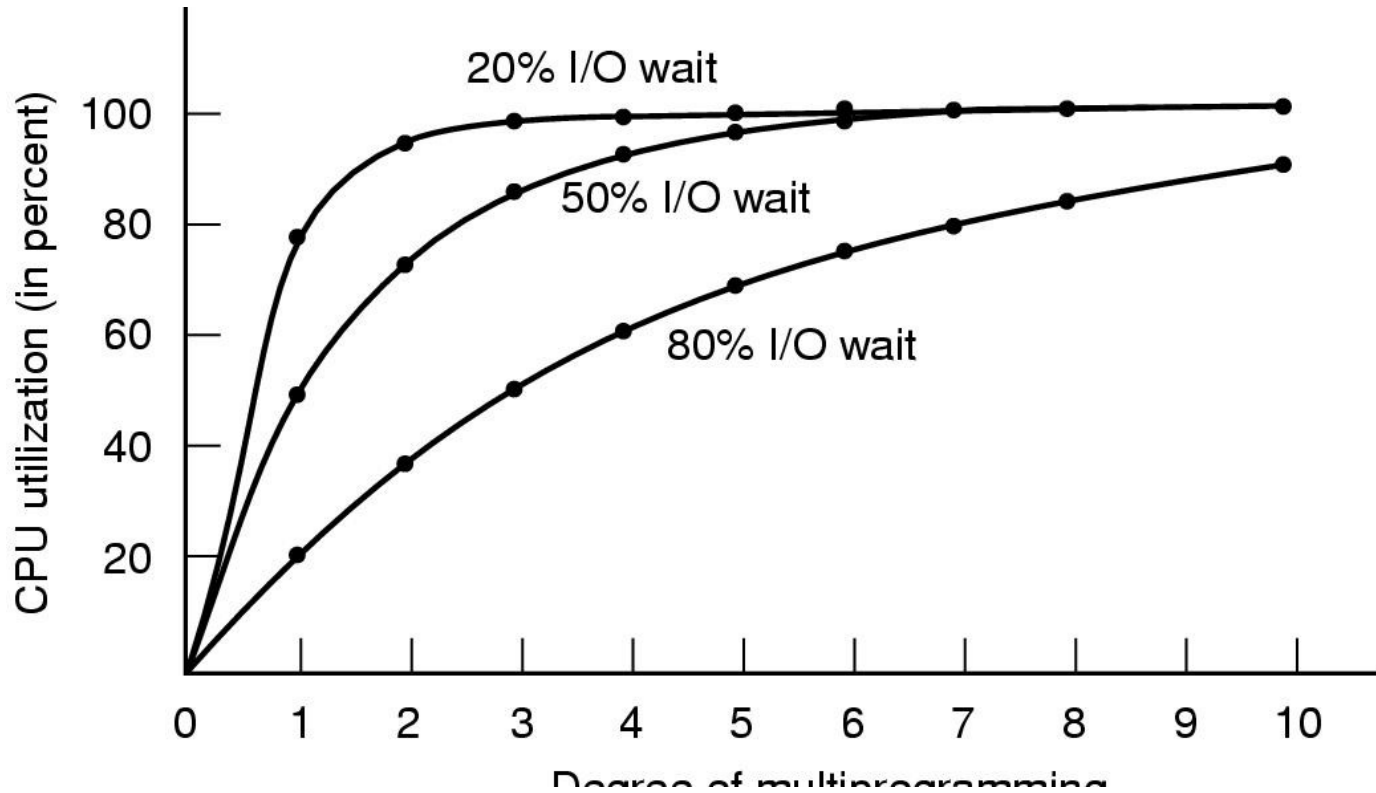


Figure 2-6. CPU utilization as a function of the number of processes in memory.

Thread Usage (1)

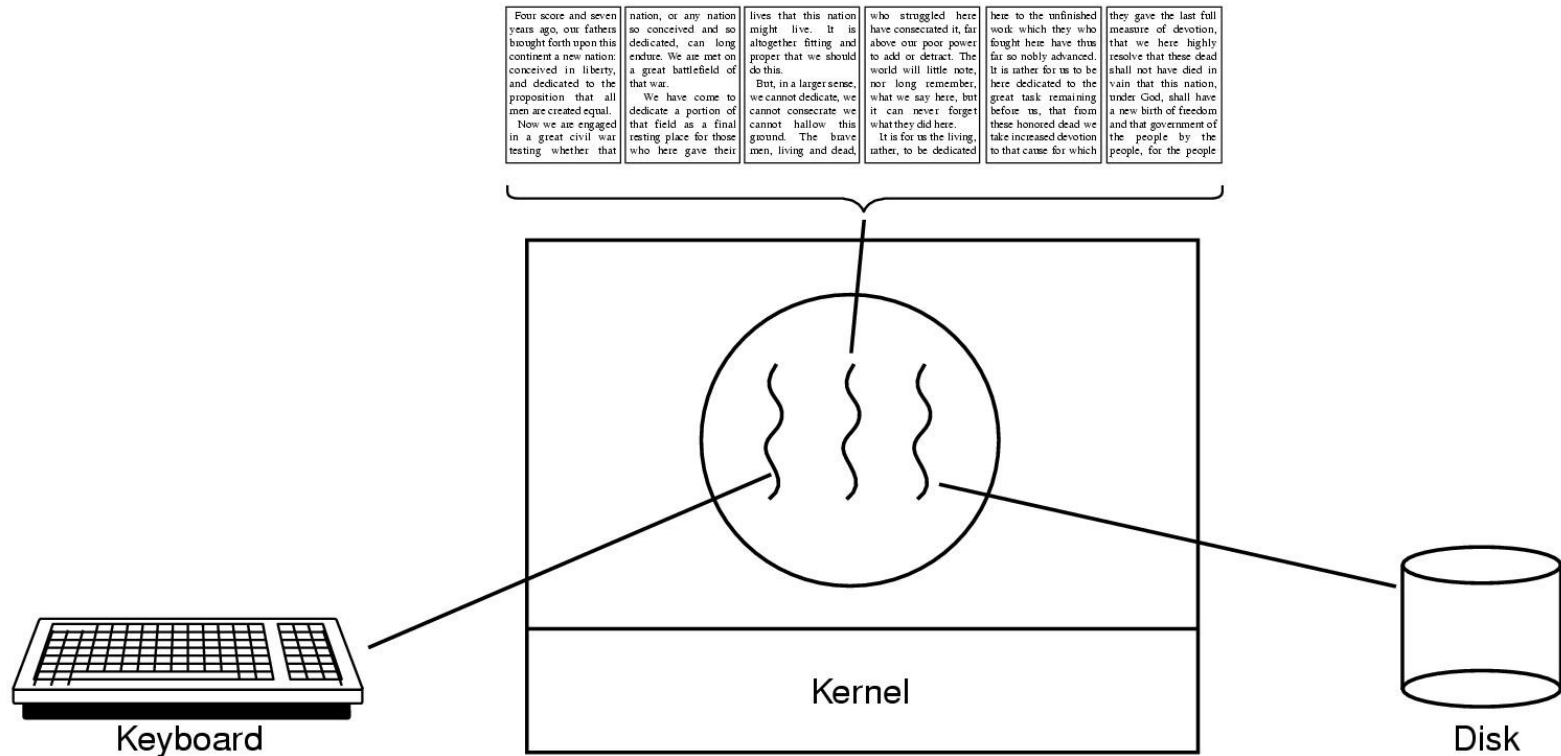


Figure 2-7. A word processor with three threads.

Thread Usage (2)

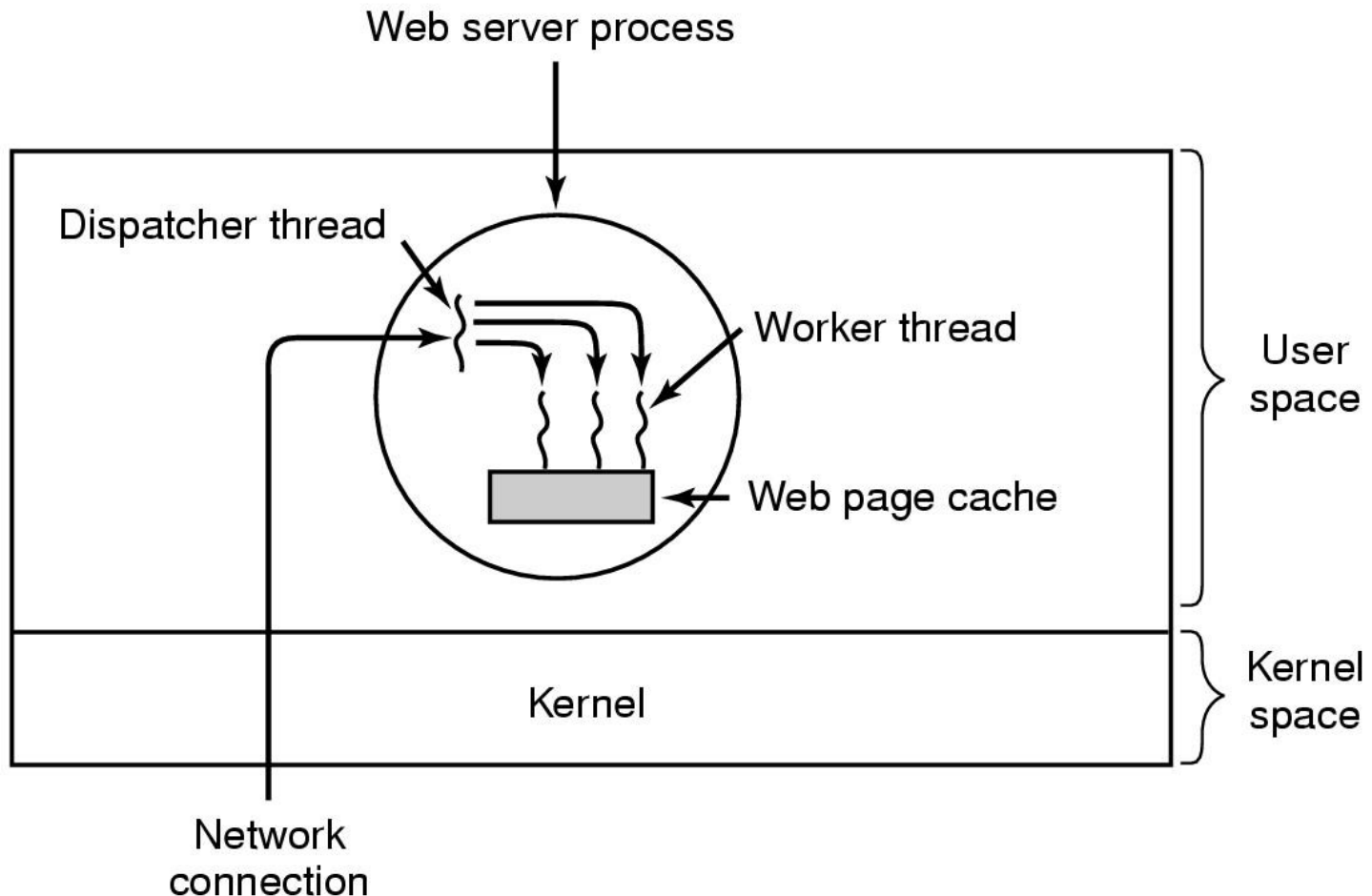


Figure 2-8. A multithreaded Web server.

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)

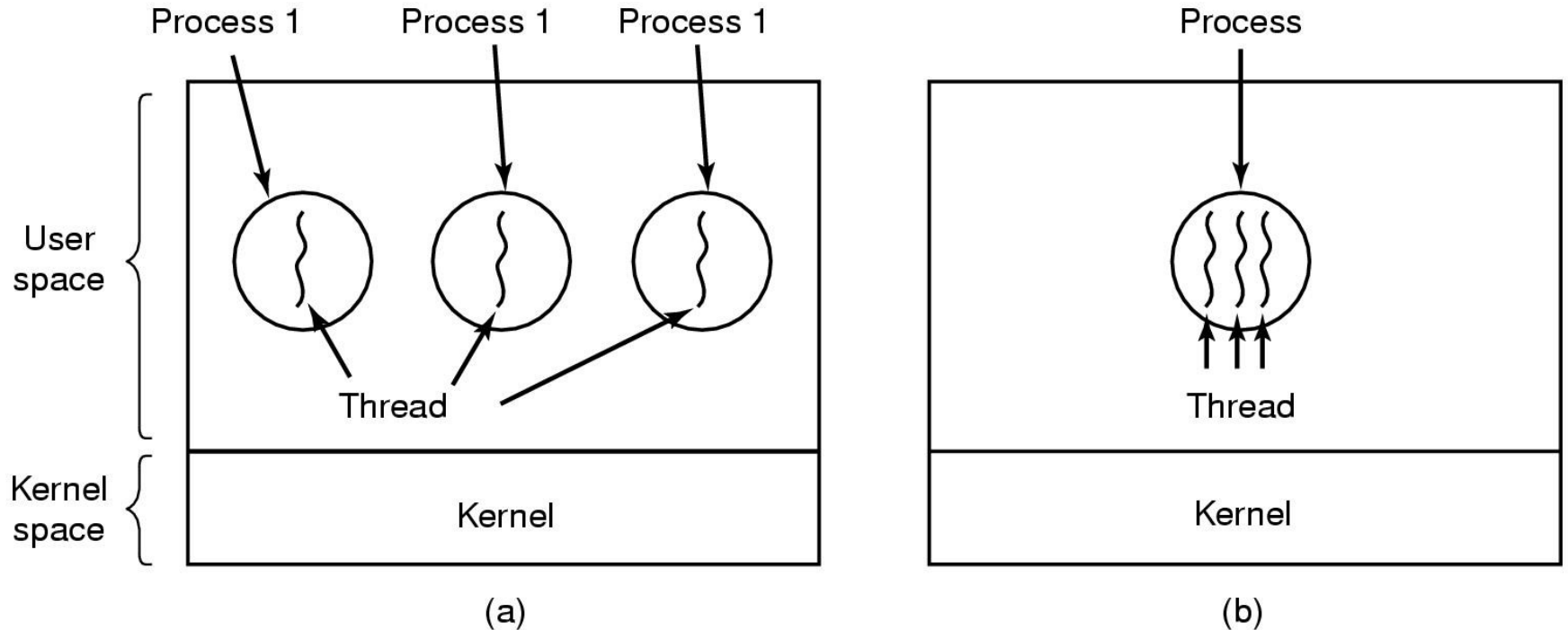


Figure 2-11. (a) Three processes each with one thread. (b) One process with three threads.

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)

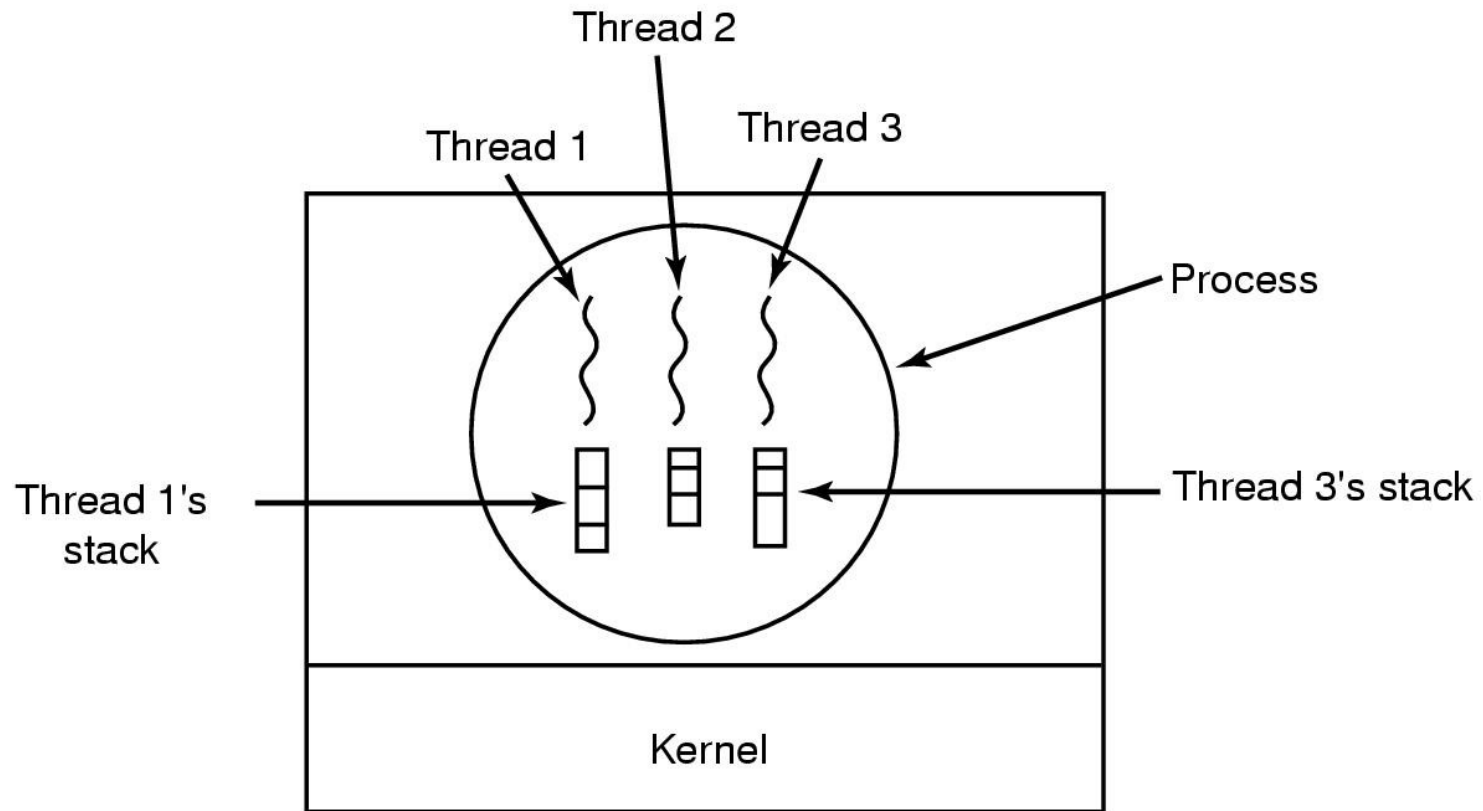


Figure 2-13. Each thread has its own stack.

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 %d0, 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 %d0, i);
        status = pthread_create(&threads[i], NULL, print_hello_world, (void *)i);

        if (status != 0) {
            printf("Oops. pthread_create returned error code %d0, status);
            exit(-1);
        }
    }
    exit(NULL);
}
```

Figure 2-15. An example program using threads.

Implementing Threads in User Space

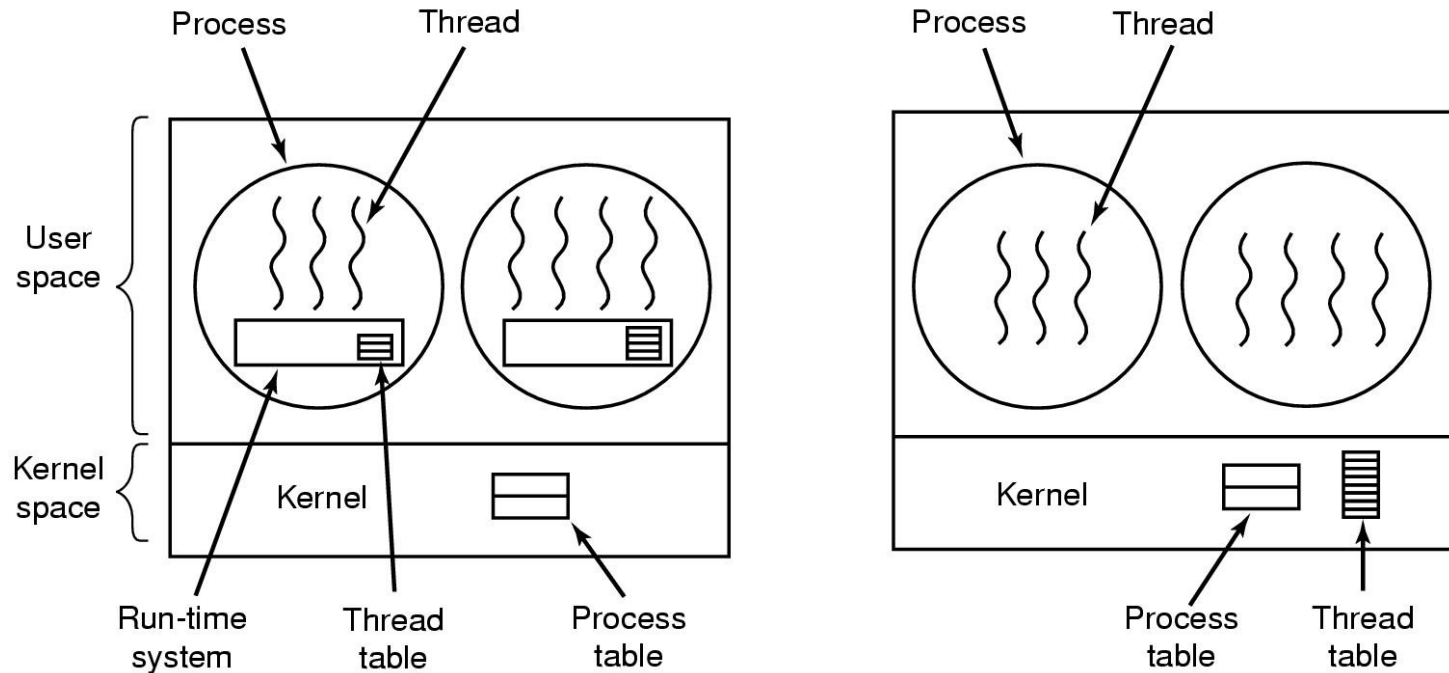


Figure 2-16. (a) A user-level threads package. (b) A threads package managed by the kernel.

Hybrid Implementations

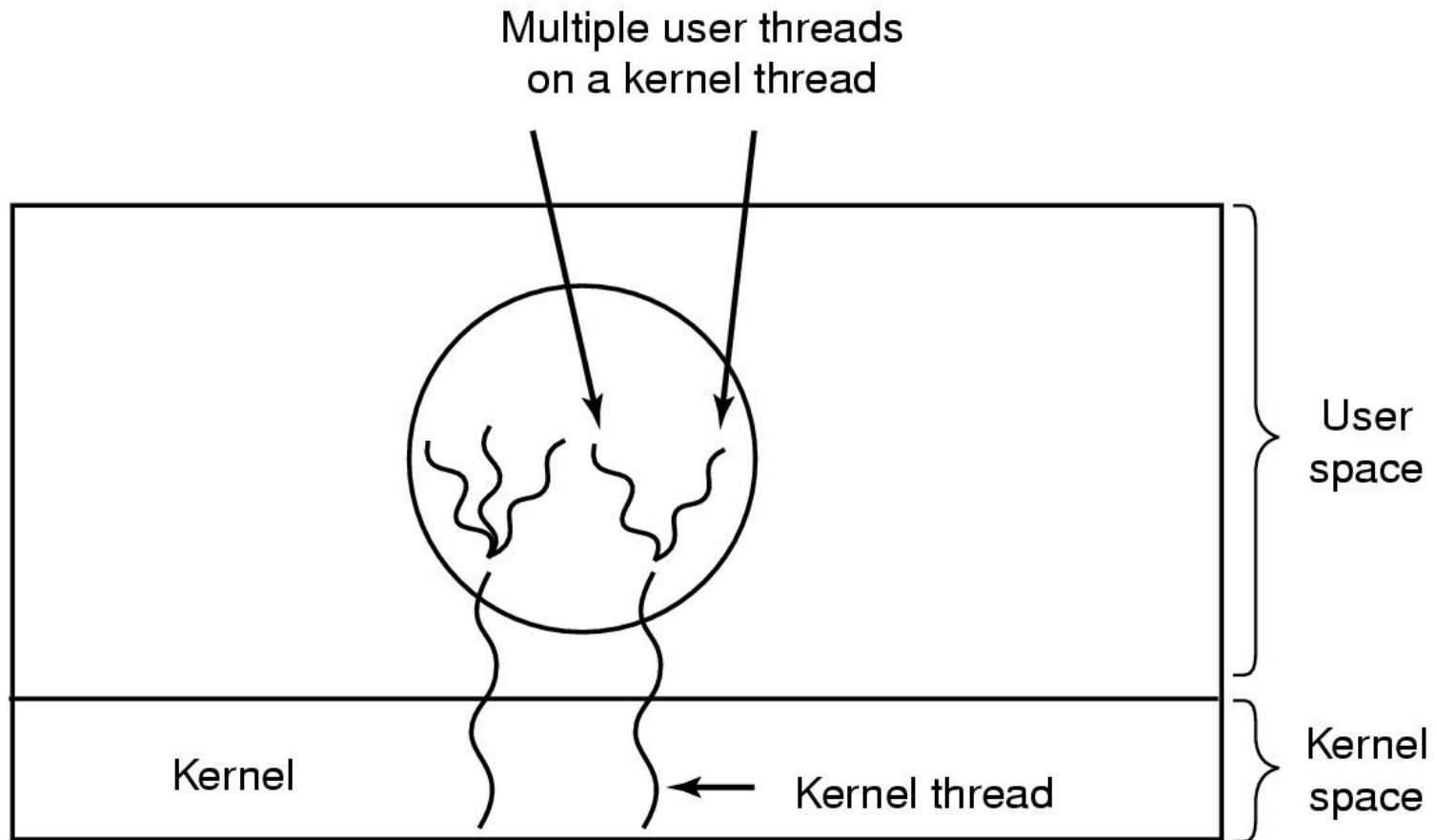


Figure 2-17. Multiplexing user-level threads onto kernel-level threads.

Pop-Up Threads

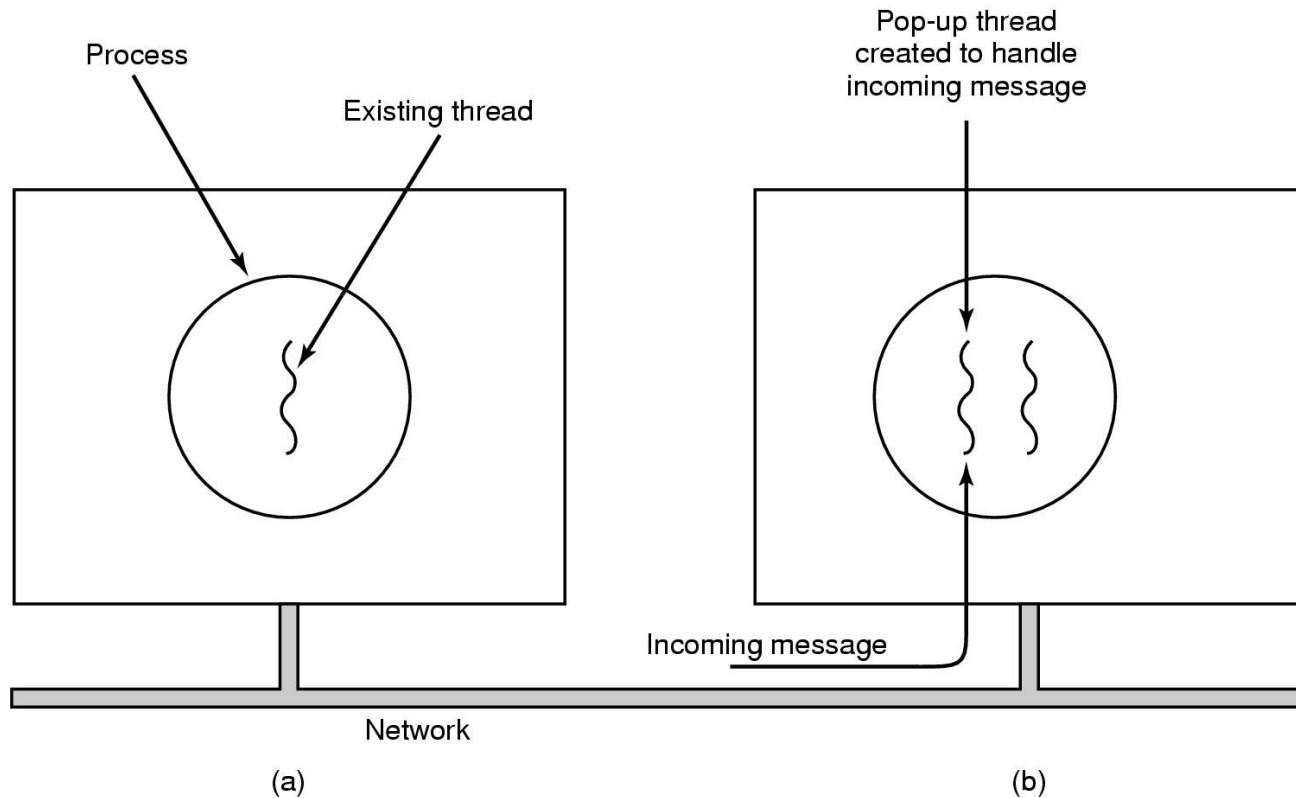


Figure 2-18. Creation of a new thread when a message arrives.
(a) Before the message arrives.
(b) After the message arrives.

Making Single-Threaded Code Multithreaded (1)

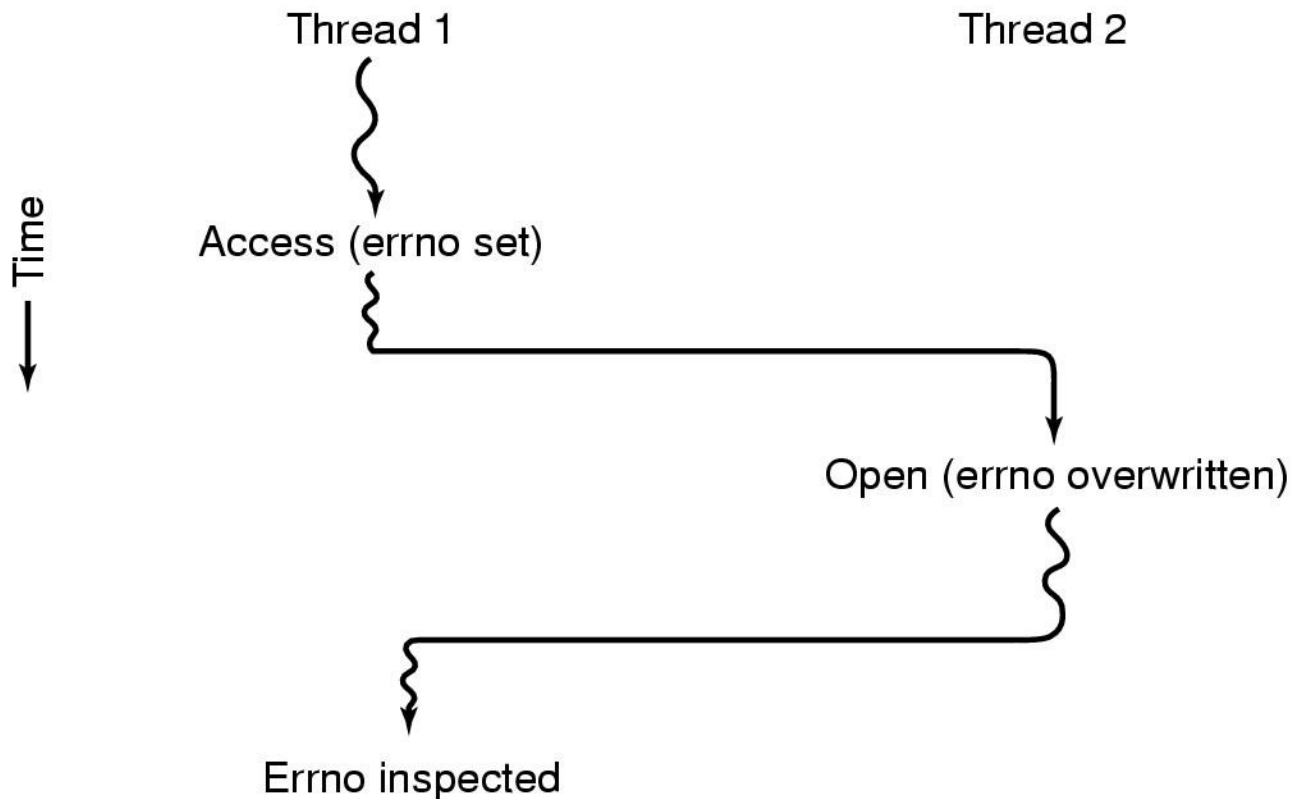


Figure 2-19. Conflicts between threads over the use of a global variable.

Making Single-Threaded Code Multithreaded (2)

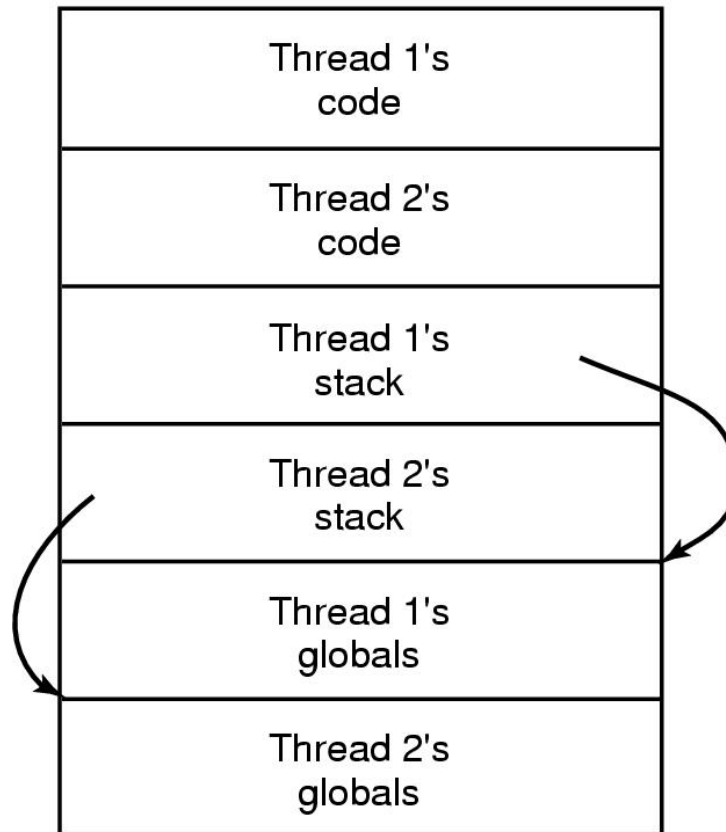


Figure 2-20. Threads can have private global variables.

Race Conditions

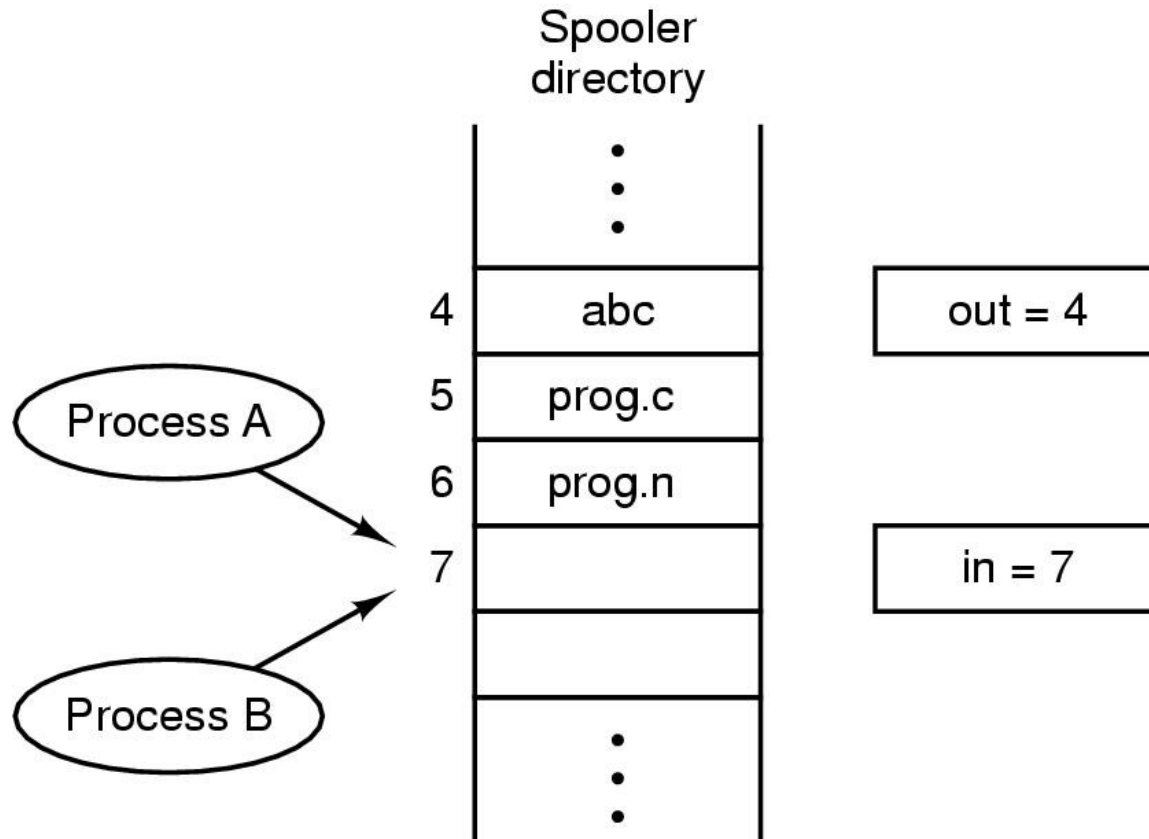


Figure 2-21. Two processes want to access shared memory at the same time.

Critical Regions (1)

Conditions required to avoid race condition:

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

Critical Regions (2)

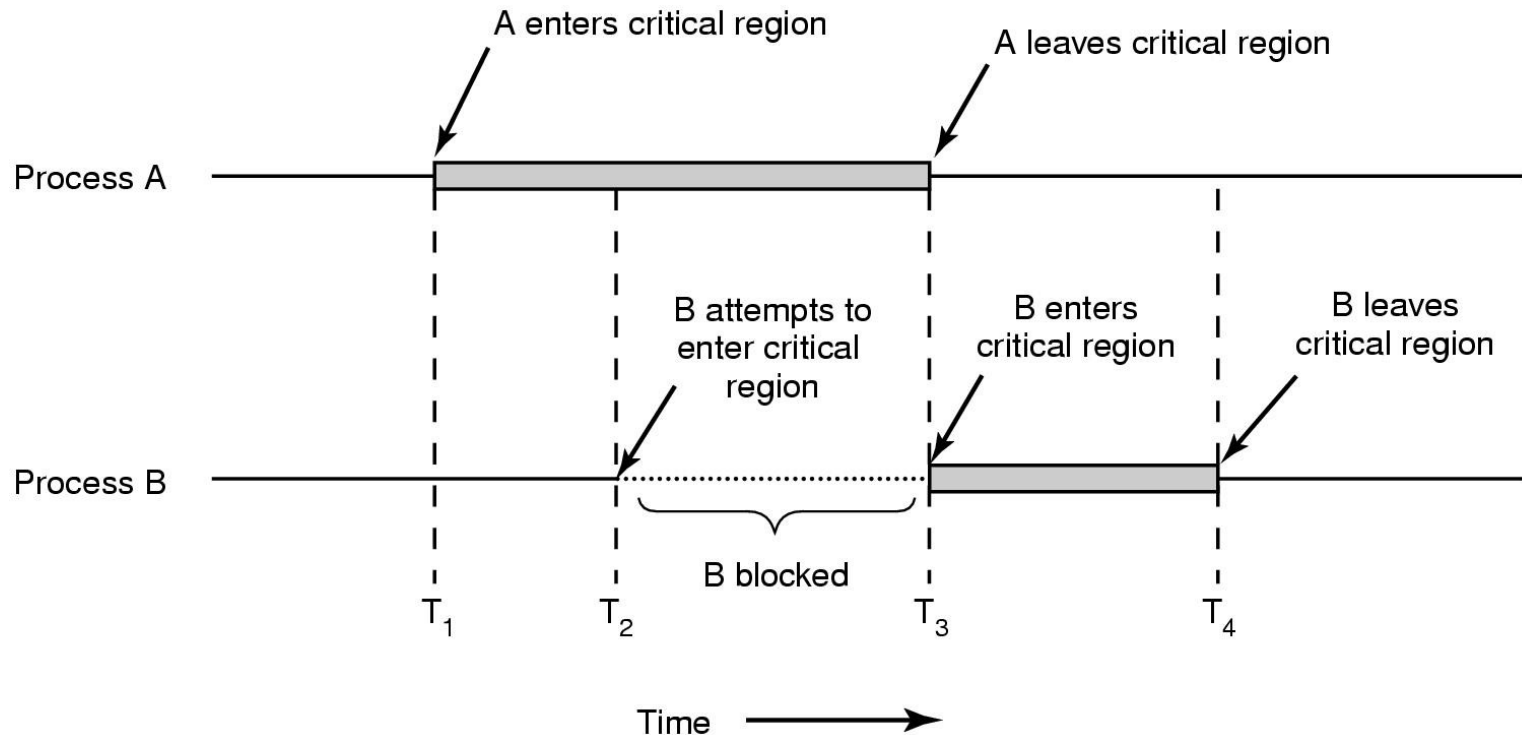


Figure 2-22. Mutual exclusion using critical regions.

Mutual Exclusion with Busy Waiting

Proposals for achieving mutual exclusion:

- Disabling interrupts
- Lock variables
- Strict alternation
- Peterson's solution
- The TSL instruction

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.

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.

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.

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.

The Producer-Consumer Problem

```
#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 */
    }
}
```

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

Semaphores

```
#define N 100
typedef int semaphore;
semaphore mutex = 1;
semaphore empty = N;
semaphore full = 0;

/* number of slots in the buffer */
/* semaphores are a special kind of int */
/* controls access to critical region */
/* counts empty buffer slots */
/* counts full buffer slots */

void producer(void)
{
    int item;

    while (TRUE) {
        item = produce_item();
        down(&empty);
        down(&mutex);
        insert_item(item);
        up(&mutex);
        up(&full);
    }

    /* TRUE is the constant 1 */
    /* generate something to put in buffer */
    /* decrement empty count */
    /* enter critical region */
    /* put new item in buffer */
    /* leave critical region */
    /* increment count of full slots */

}

void consumer(void)
{
    int item;

    while (TRUE) {
        down(&full);
        down(&mutex);
        item = remove_item();
        up(&mutex);
        up(&empty);
        consume_item(item);
    }

    /* infinite loop */
    /* decrement full count */
    /* enter critical region */
    /* take item from buffer */
    /* leave critical region */
    /* increment count of empty slots */
    /* do something with the item */

}

. . }
```

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

Mutexes

mutex_lock:

TSL REGISTER,MUTEX

CMP REGISTER,#0

JZE ok

CALL thread_yield

JMP mutex_lock

ok: RET

| copy mutex to register and set mutex to 1
| was mutex zero?
| if it was zero, mutex was unlocked, so return
| mutex is busy; schedule another thread
| try again
| return to caller; critical region entered

mutex_unlock:

MOVE MUTEX,#0

RET

| store a 0 in mutex
| 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;
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);
}

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)
{
    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;

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.

Message Passing (1)

```
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
    }
}
```

• • •

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

Message Passing (2)

• • •

```
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
    }
}
```

• • •

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

Message Passing (3)

• • •

```
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 fewer 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.

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 */
    }
}

. . .
```

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

Producer-Consumer Problem with 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 2-36. The producer-consumer problem with N messages.

Barriers

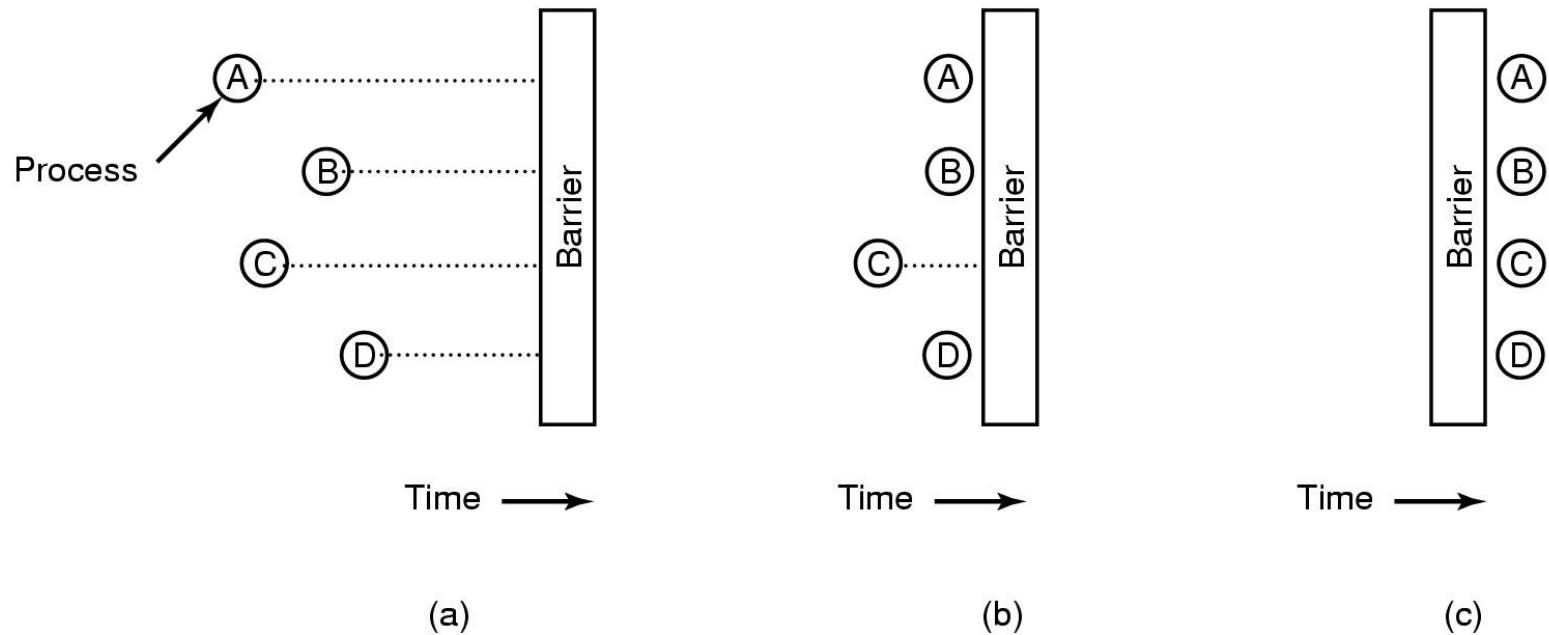


Figure 2-37. Use of a barrier. (a) Processes approaching a barrier. (b) All processes but one blocked at the barrier. (c) When the last process arrives at the barrier, all of them are let through.

Scheduling – Process Behavior

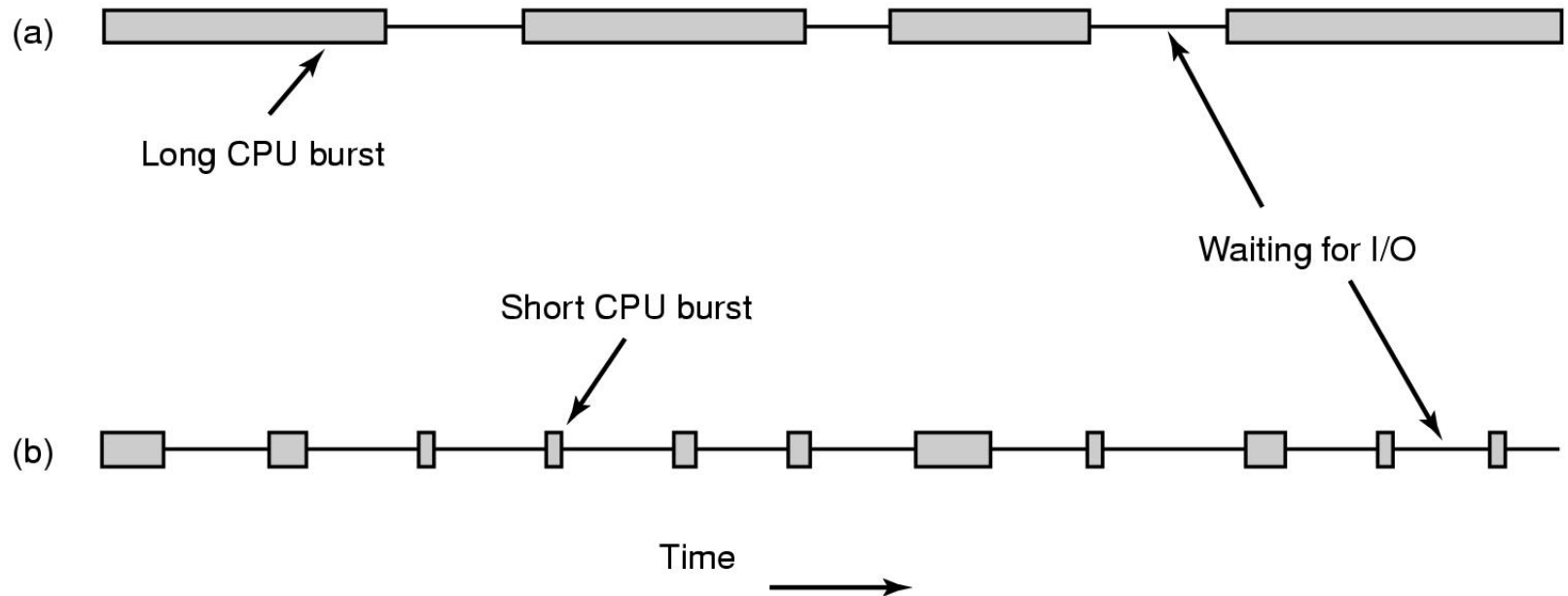


Figure 2-38. Bursts of CPU usage alternate with periods of waiting for I/O. (a) A CPU-bound process. (b) An I/O-bound process.

Categories of Scheduling Algorithms

- Batch
- Interactive
- 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

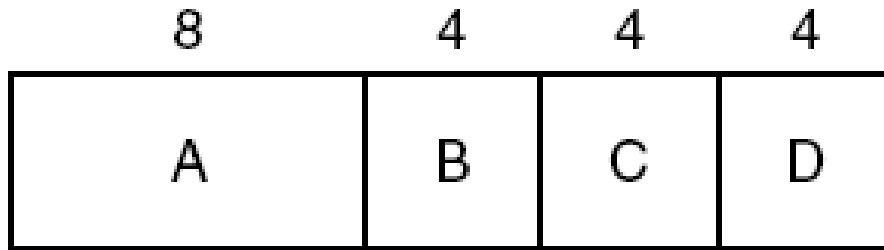
Predictability - avoid quality degradation in multimedia systems

Figure 2-39. Some goals of the scheduling algorithm under different circumstances.

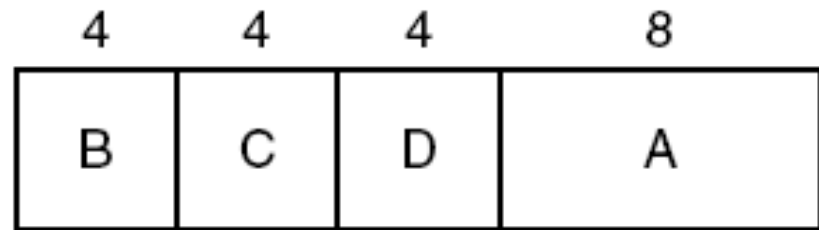
Scheduling in Batch Systems

- First-come first-served
- Shortest job first
- Shortest remaining Time next

Shortest Job First



(a)



(b)

Figure 2-40. 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

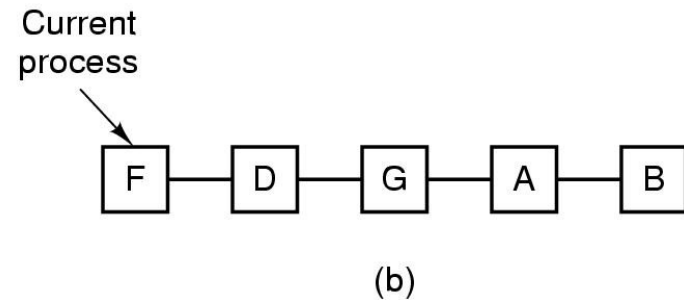
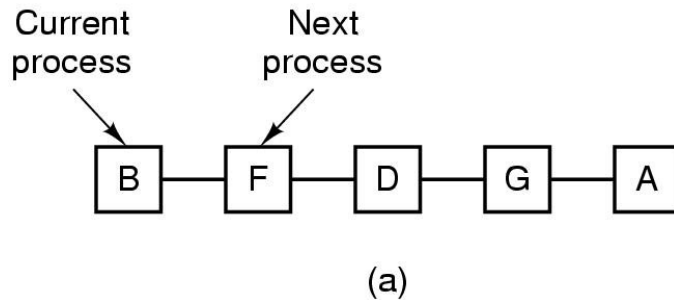


Figure 2-41. Round-robin scheduling.
(a) The list of runnable processes. (b) The list of runnable processes after B uses up its quantum.

Priority Scheduling

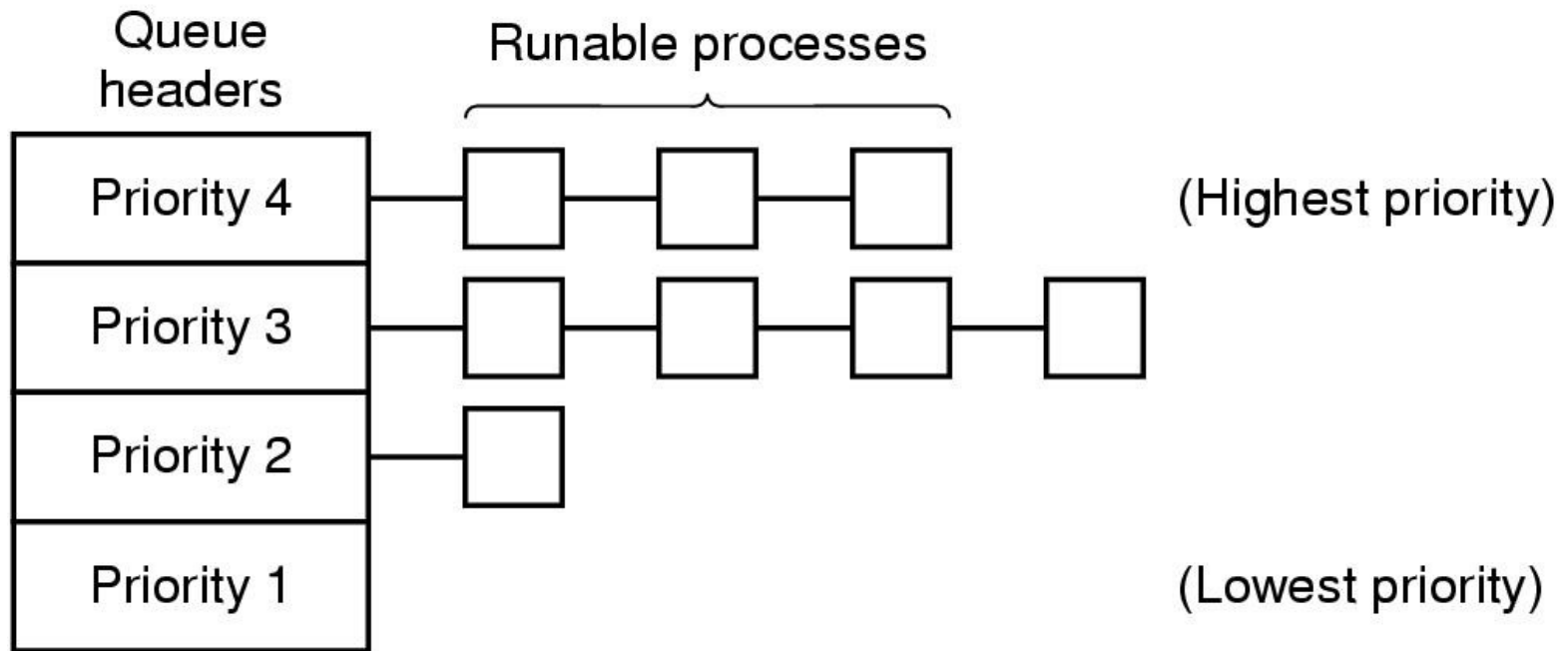
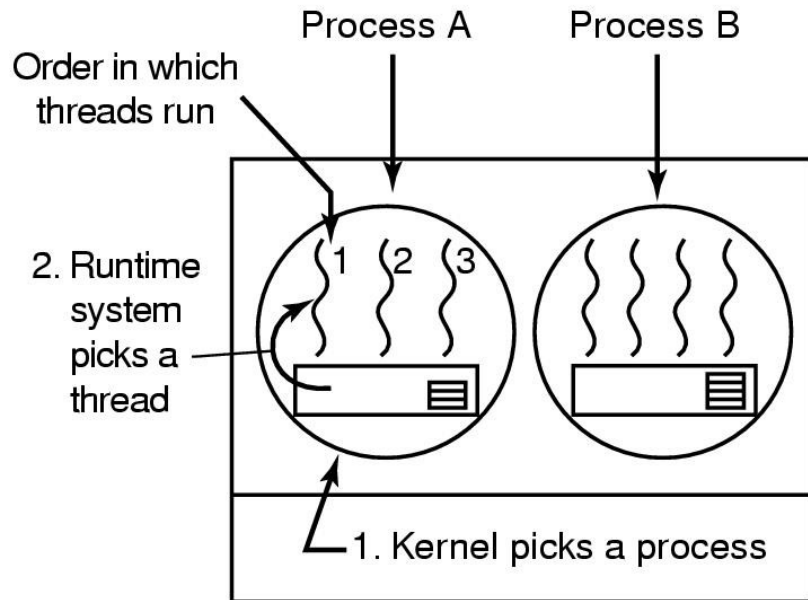
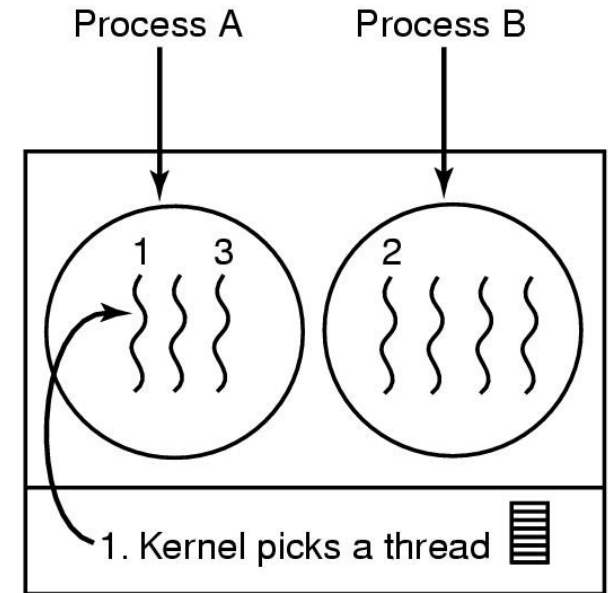


Figure 2-42. A scheduling algorithm with four priority classes.

Thread Scheduling (1)



Possible: A1, A2, A3, A1, A2, A3
 Not possible: A1, B1, A2, B2, A3, B3



Possible: A1, A2, A3, A1, A2, A3
 Also possible: A1, B1, A2, B2, A3, B3

Figure 2-43. (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)

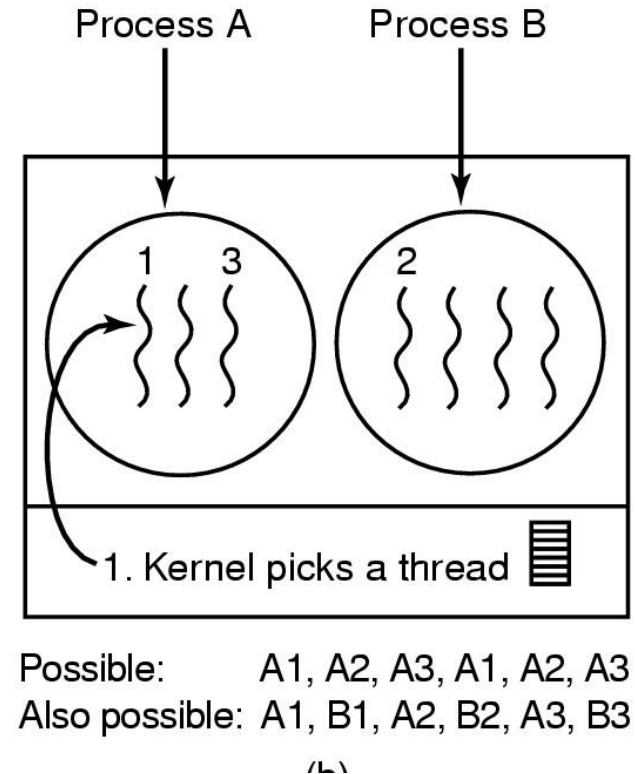
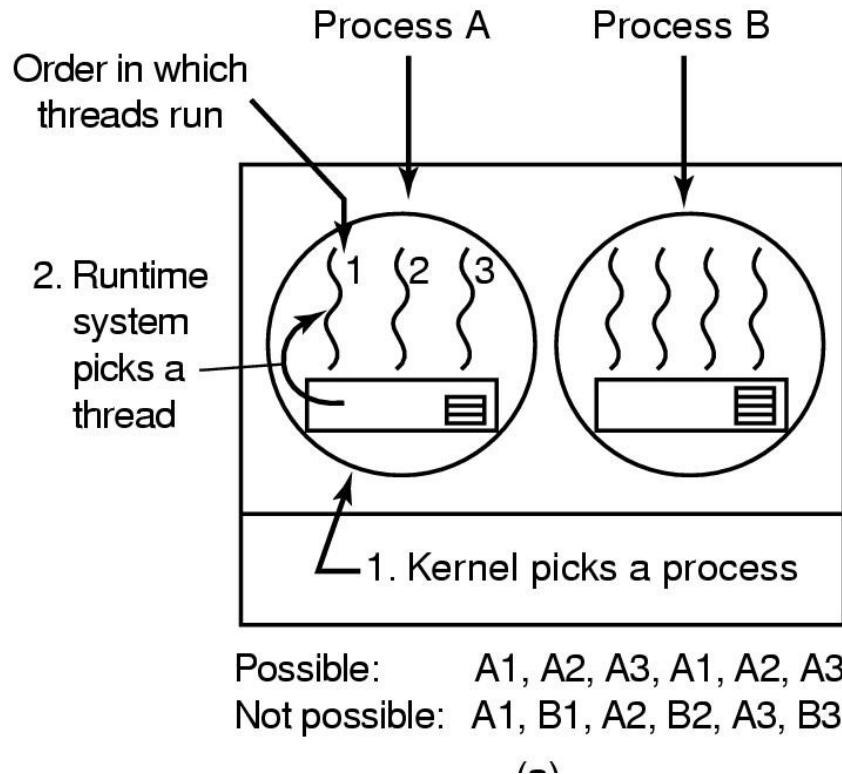


Figure 2-43. (b) Possible scheduling of kernel-level threads with the same characteristics as (a).

Dining Philosophers Problem (1)

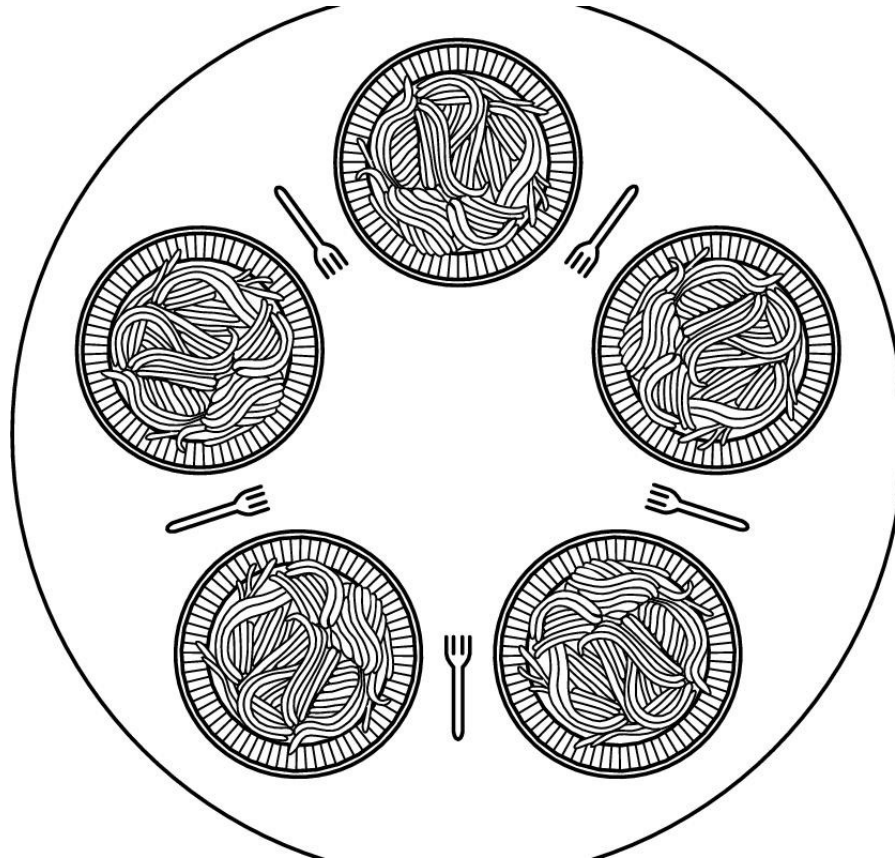


Figure 2-44. Lunch time in the Philosophy Department.

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 2-45. A nonsolution to the dining philosophers problem.

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 2-46. A solution to the dining philosophers problem.

Dining Philosophers Problem (4)

• • •

```
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 2-46. A solution to the dining philosophers problem.

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 2-46. A solution to the dining philosophers problem.

The Readers and Writers Problem (1)

```
typedef int semaphore;
semaphore mutex = 1;
semaphore db = 1;
int rc = 0;

void reader(void)
{
    while (TRUE) {
        down(&mutex);
        rc = rc + 1;
        if (rc == 1) down(&db);
        up(&mutex);
        read_data_base();
        down(&mutex);
        rc = rc - 1;
        if (rc == 0) up(&db);
        up(&mutex);
        use_data_read();
    }
}
```

```
/* use your imagination */
/* controls access to 'rc' */
/* controls access to the database */
/* # of processes reading or wanting to */

/* repeat forever */
/* get exclusive access to 'rc' */
/* one reader more now */
/* if this is the first reader ... */
/* release exclusive access to 'rc' */
/* access the data */
/* get exclusive access to 'rc' */
/* one reader fewer now */
/* if this is the last reader ... */
/* release exclusive access to 'rc' */
/* noncritical region */
```

...

Figure 2-47. A solution to the readers and writers problem.

The Readers and Writers Problem (2)

• • •

```
void writer(void)
{
    while (TRUE) {
        think_up_data();
        down(&db);
        write_data_base();
        up(&db);
    }
}
```

/* repeat forever */
/* noncritical region */
/* get exclusive access */
/* update the data */
/* release exclusive access */

Figure 2-47. A solution to the readers and writers problem.