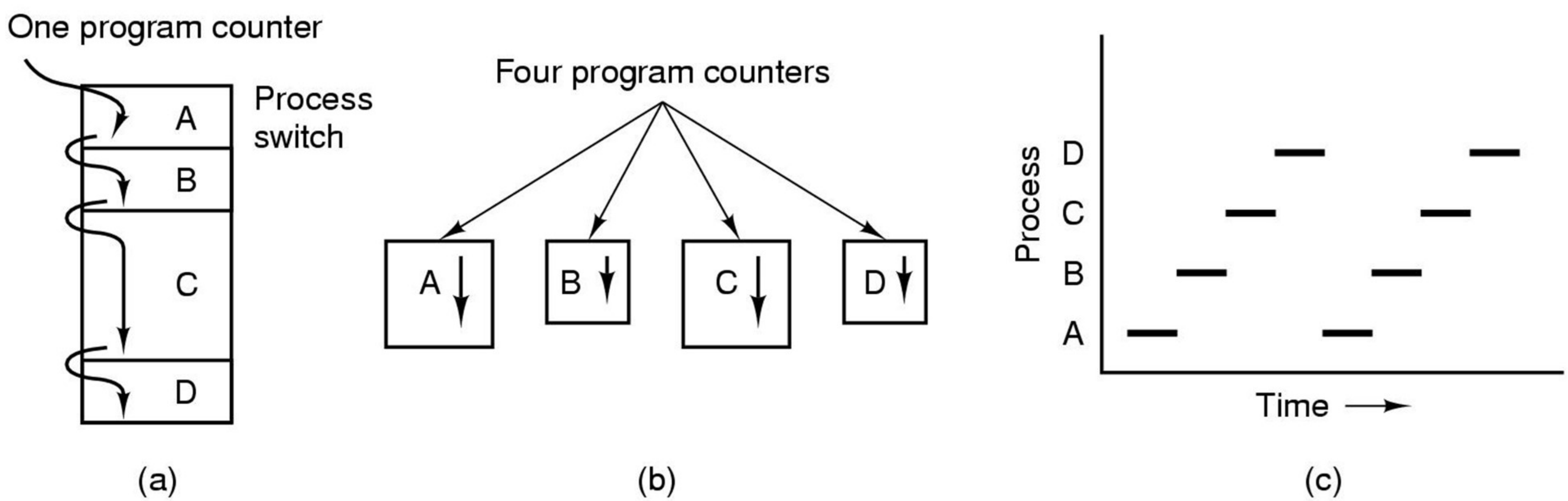
Chapter 2

PROCESSES AND THREADS

2.1 Processes
2.2 Threads
2.3 Interprocess communication
2.4 Classical IPC problems
2.5 Scheduling

Processes
The Process Model

- Multiprogramming of four programs
- Conceptual model of 4 independent, sequential processes
- Only one program active at any instant



Process Creation

- Principal events that cause process creation
  - System initialization
  - Execution of a process creation system
  - User request to create a new process
  - Initiation of a batch job

Process Termination

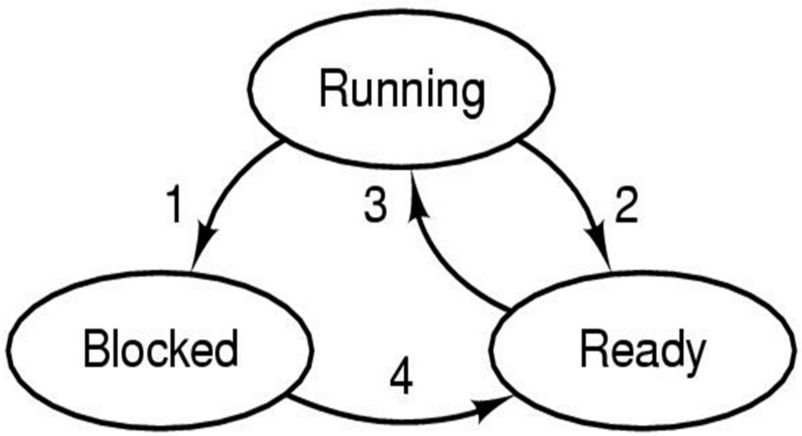
- Conditions which terminate processes
  - Normal exit (voluntary)
  - Error exit (voluntary)
  - Fatal error (involuntary)
  - Killed by another process (involuntary)

Process Hierarchies

- Parent creates a child process, child processes can create its own process
- Forms a hierarchy
  - UNIX calls this a "process group"
- Windows has no concept of process hierarchy
  - all processes are created equal

Process States (1)

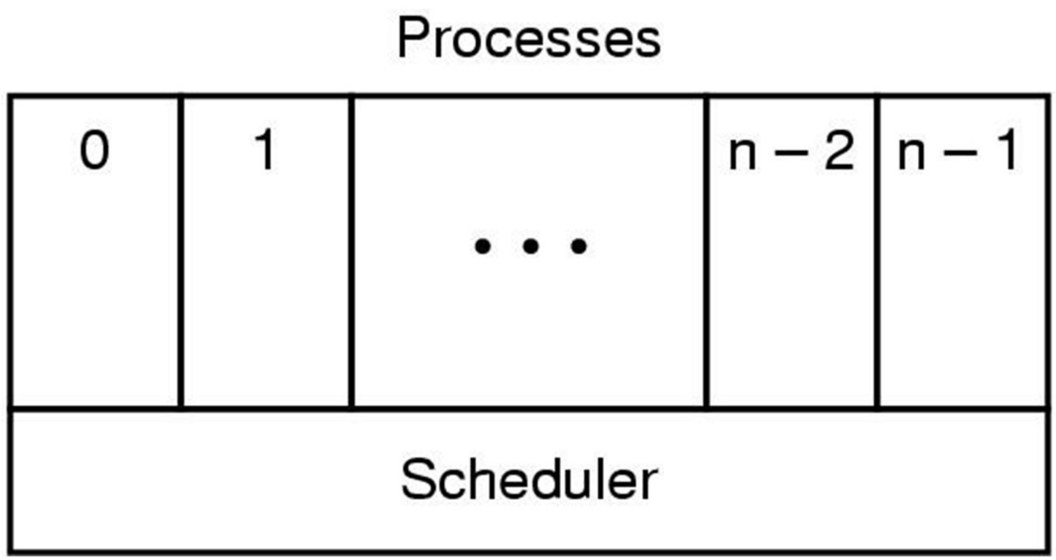
- Possible process states
  - running
  - blocked
  - ready
- Transitions between states shown



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

Process States (2)

- Lowest layer of process-structured OS
  - handles interrupts, scheduling
- Above that layer are sequential processes



Implementation of Processes (1)

- Fields of a process table entry

Process management	Memory management	File management
Registers	Pointer to text segment	Root directory
Program counter	Pointer to data segment	Working directory
Program status word	Pointer to stack segment	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		

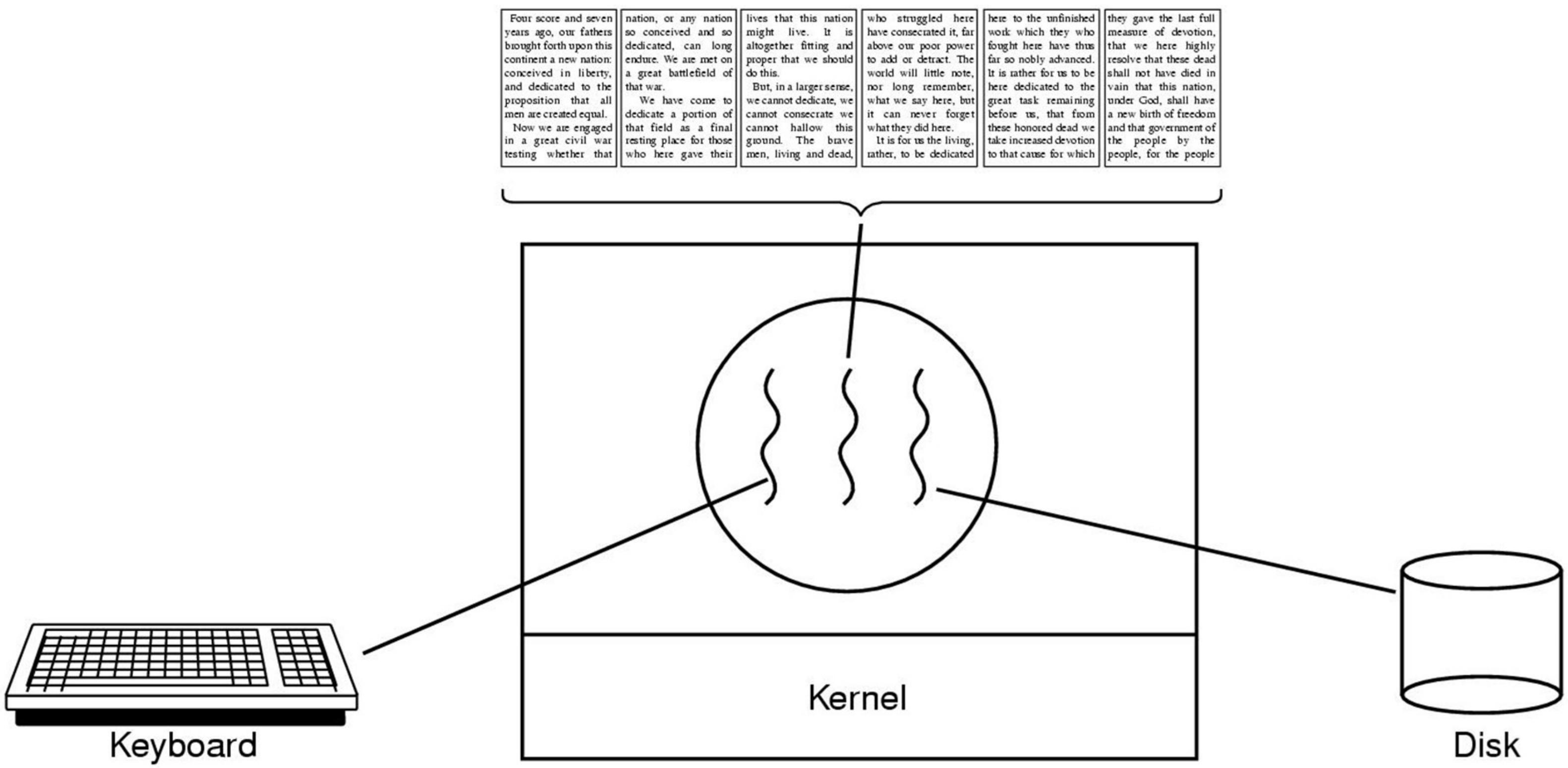
Implementation of Processes (2)

- Skeleton of what lowest level of OS does when an interrupt occurs

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.

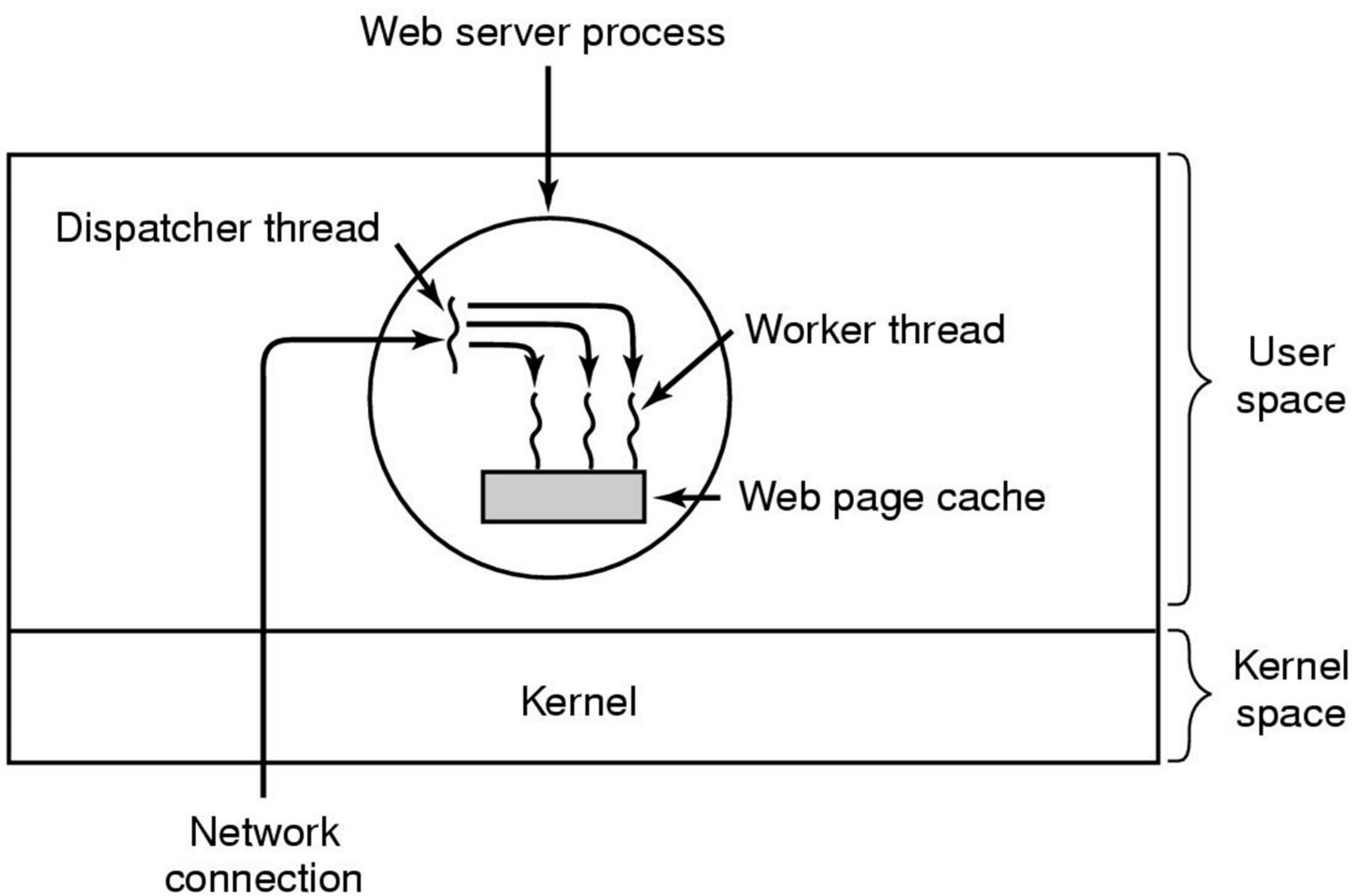
Thread Usage (1)

- A word processor with three threads



Thread Usage (2)

- A multithreaded Web server



Thread Usage (3)

- Rough outline of code for previous slide
  - (a) Dispatcher thread
  - (b) Worker thread

```
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)

Thread Usage (4)

- Three ways to construct a server

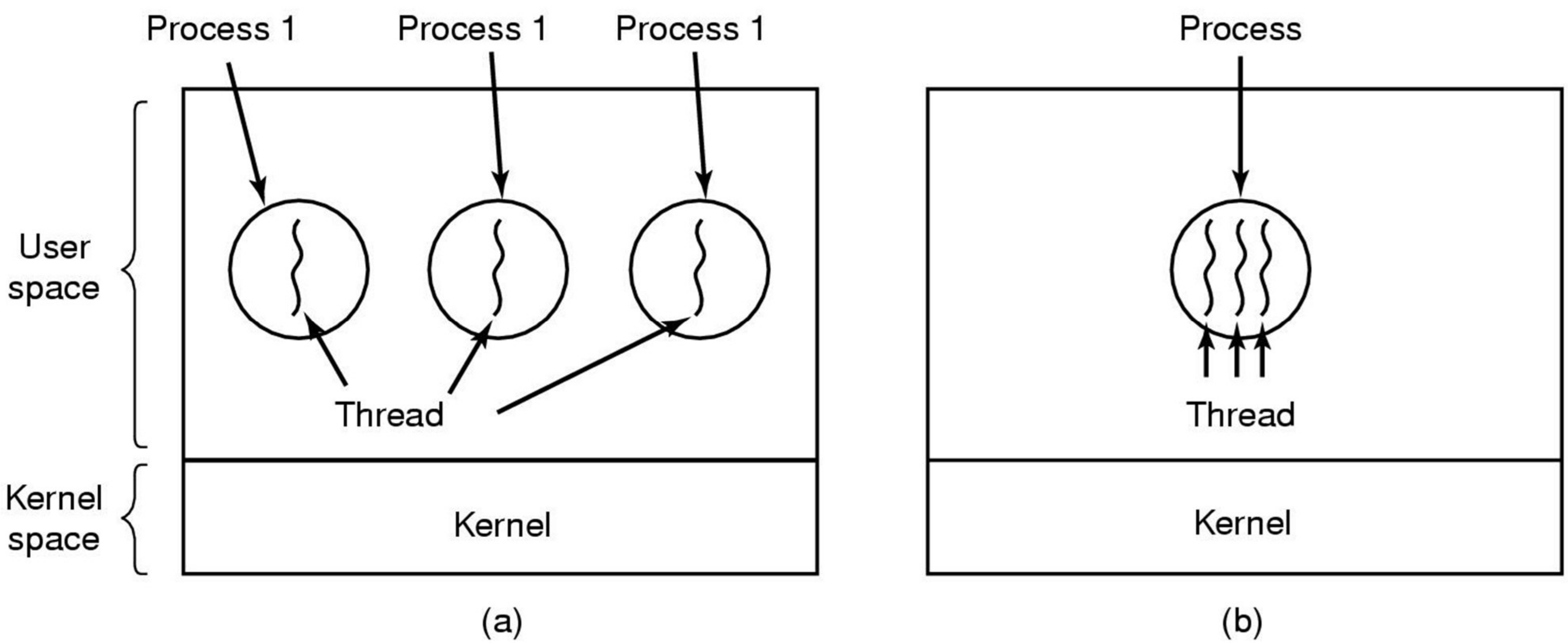
Model	Characteristics
Threads	Parallelism, blocking system calls
Single-threaded process	No parallelism, blocking system calls
Finite-state machine	Parallelism, nonblocking system calls, interrupts

Thread Usage (5)

- Reasons for using threads
  - Many applications need to do multiple activities at once
  - They are lighter weight than processes
  - Speed up gain by overlapping computing and I/O
  - Real parallelism on systems with multiple CPUs
  - They make it possible to retain sequential processes that make blocking calls and still achieve parallelism

Threads
The Thread Model (1)

- (a) Three processes each with one thread
- (b) One process with three threads



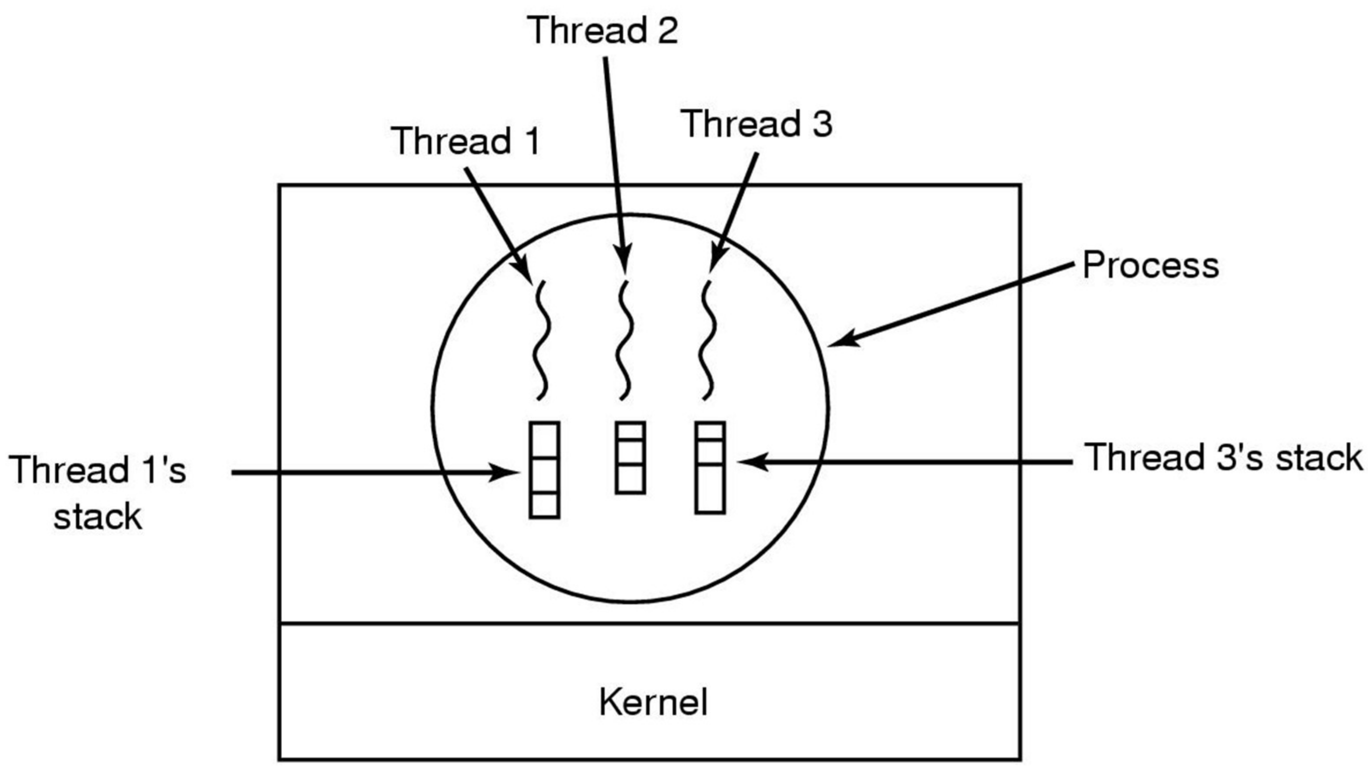
The Thread Model (2)

- Items shared by all threads in a process
- Items private to each thread

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	

The Thread Model (3)

- Each thread has its own stack



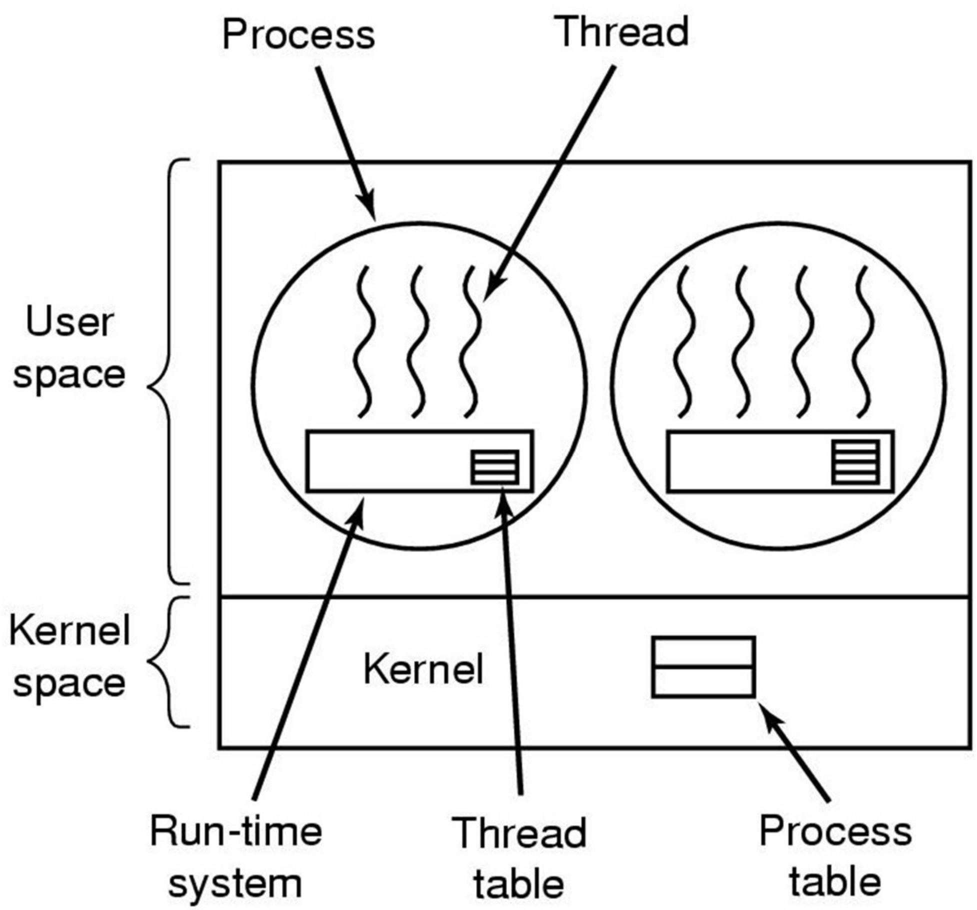
POSIX Threads

- Pthreads
  - IEEE standard 1003.1c
    - defines over 60 function calls

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

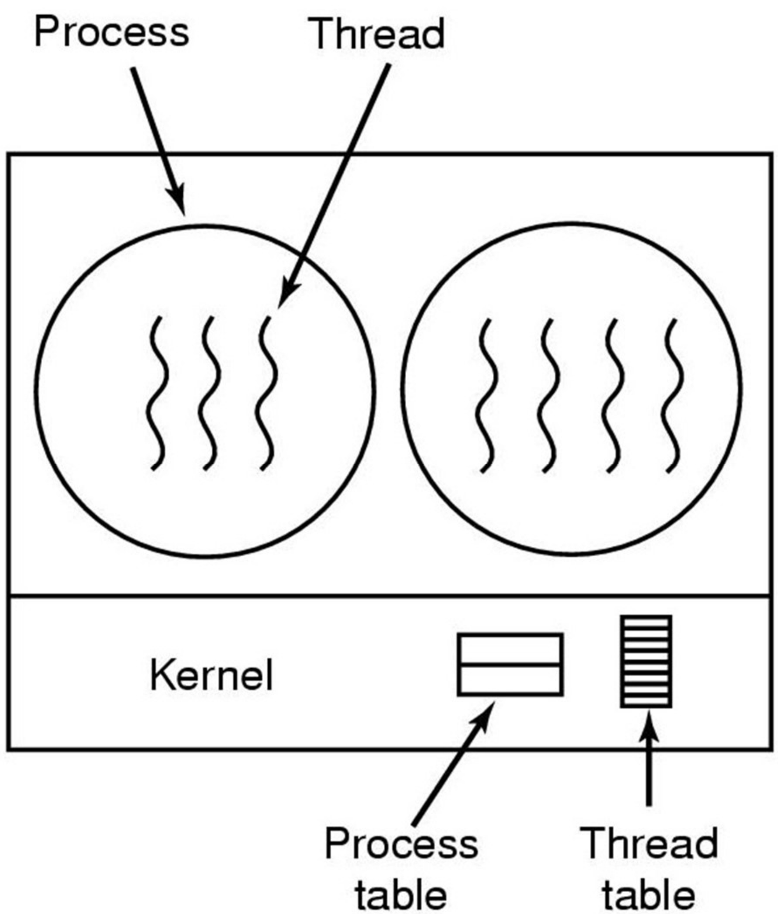
Implementing Threads in User Space

- A user-level threads package



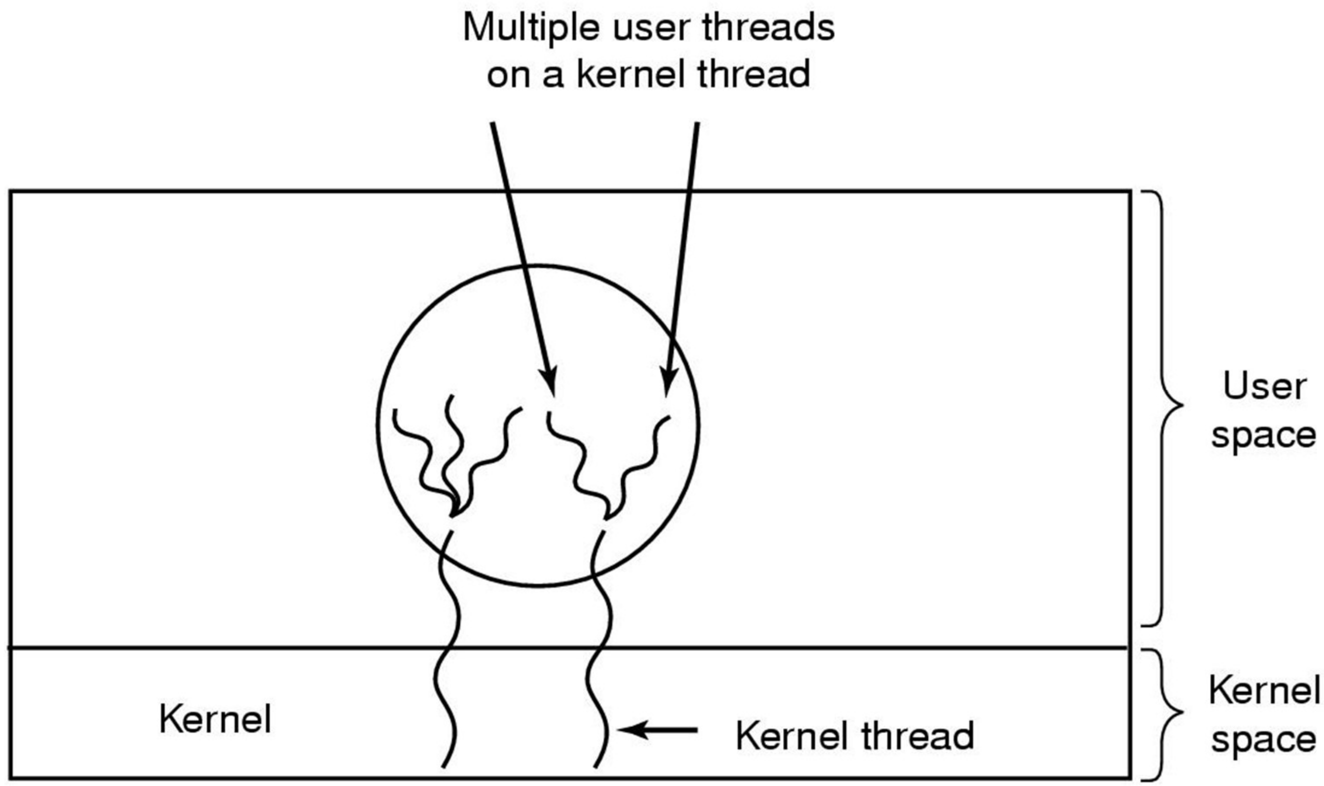
Implementing Threads in the Kernel

- A threads package managed by the kernel



Hybrid Implementations

- Multiplexing user-level threads onto kernel-level threads

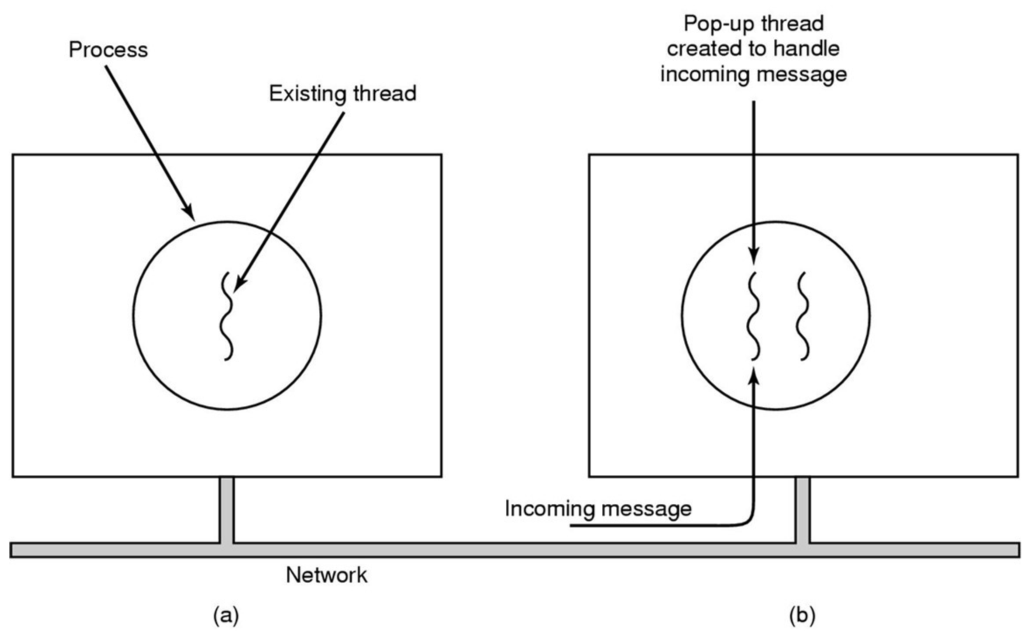


Scheduler Activations

- Goal – mimic functionality of kernel threads
  - gain performance of user space threads
- Avoids unnecessary user/kernel transitions
- Kernel assigns virtual processors to each process
  - lets runtime system allocate threads to processors
- Problem:
  - Fundamental reliance on kernel (lower layer) calling procedures in user space (higher layer)

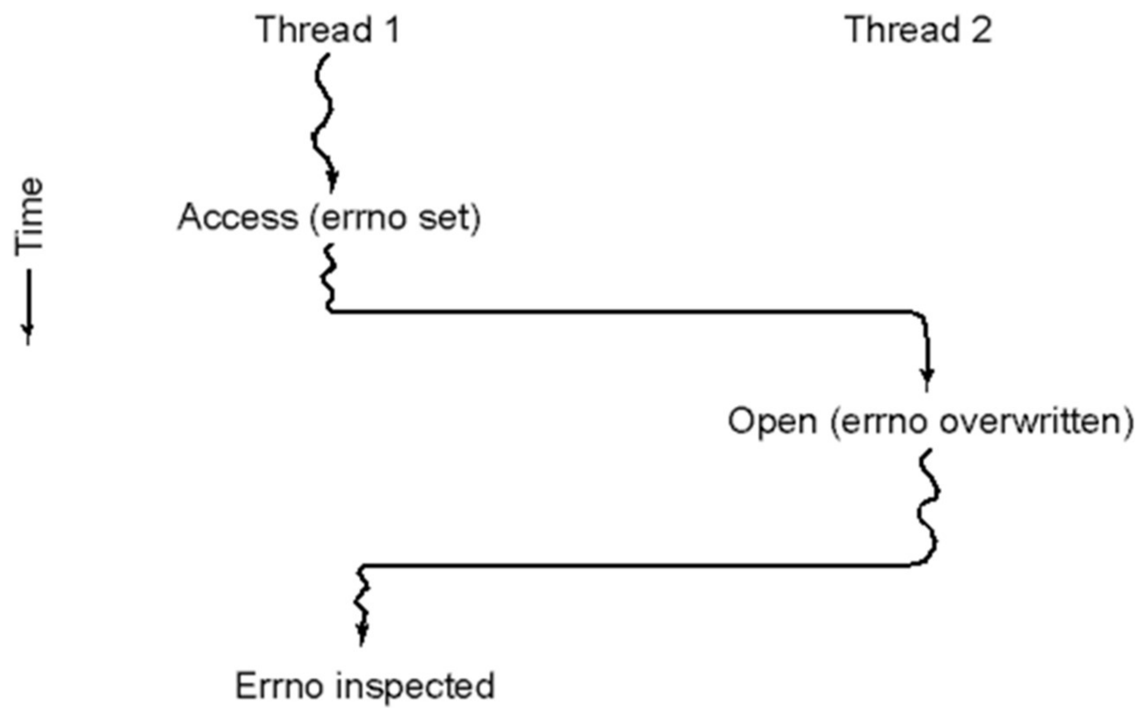
Pop-Up Threads

- Creation of a new thread when message arrives
  - (a) before message arrives
  - (b) after message arrives



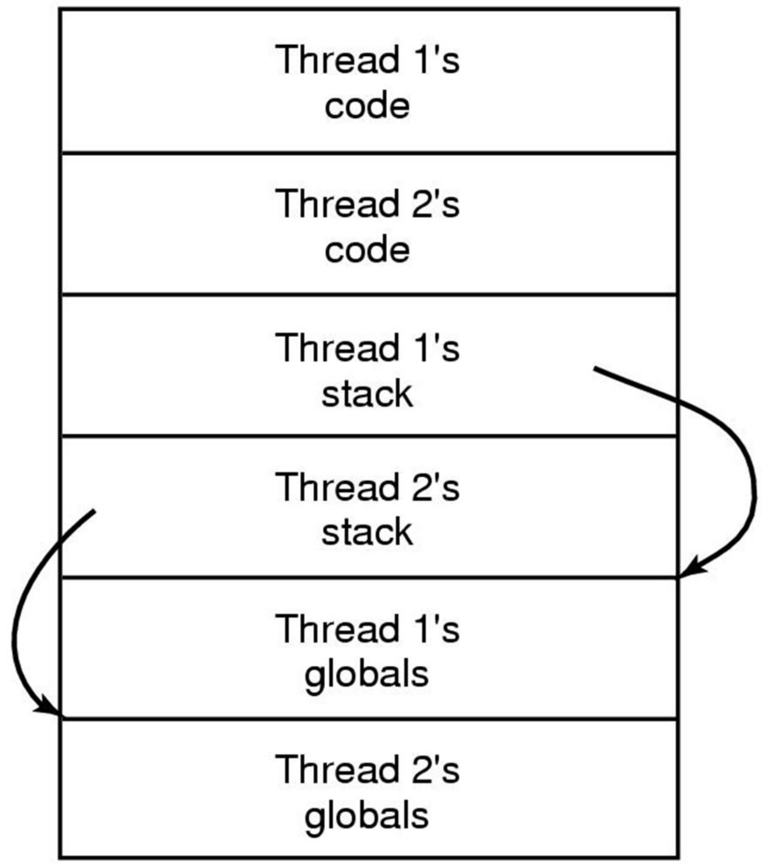
Making Single-Threaded Code Multithreaded (1)

- Conflicts between threads over the use of a global variable



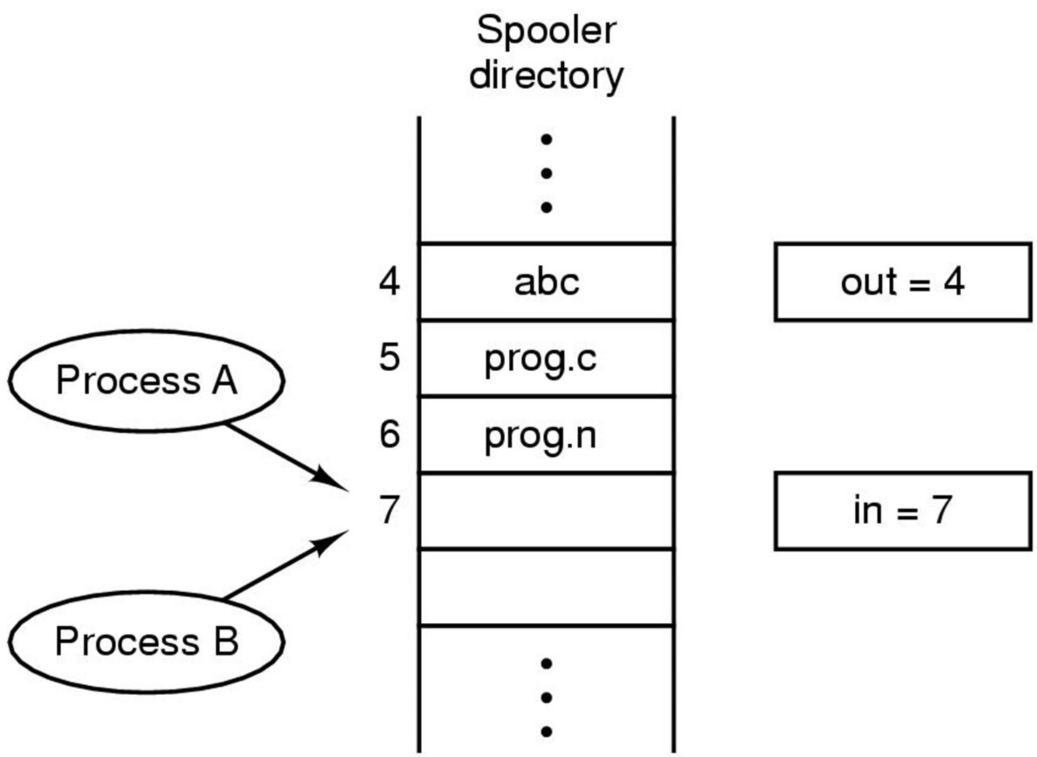
Making Single-Threaded Code Multithreaded (2)

- Threads can have private global variables



Interprocess Communication Race Conditions

- Two processes want to access shared memory at same time

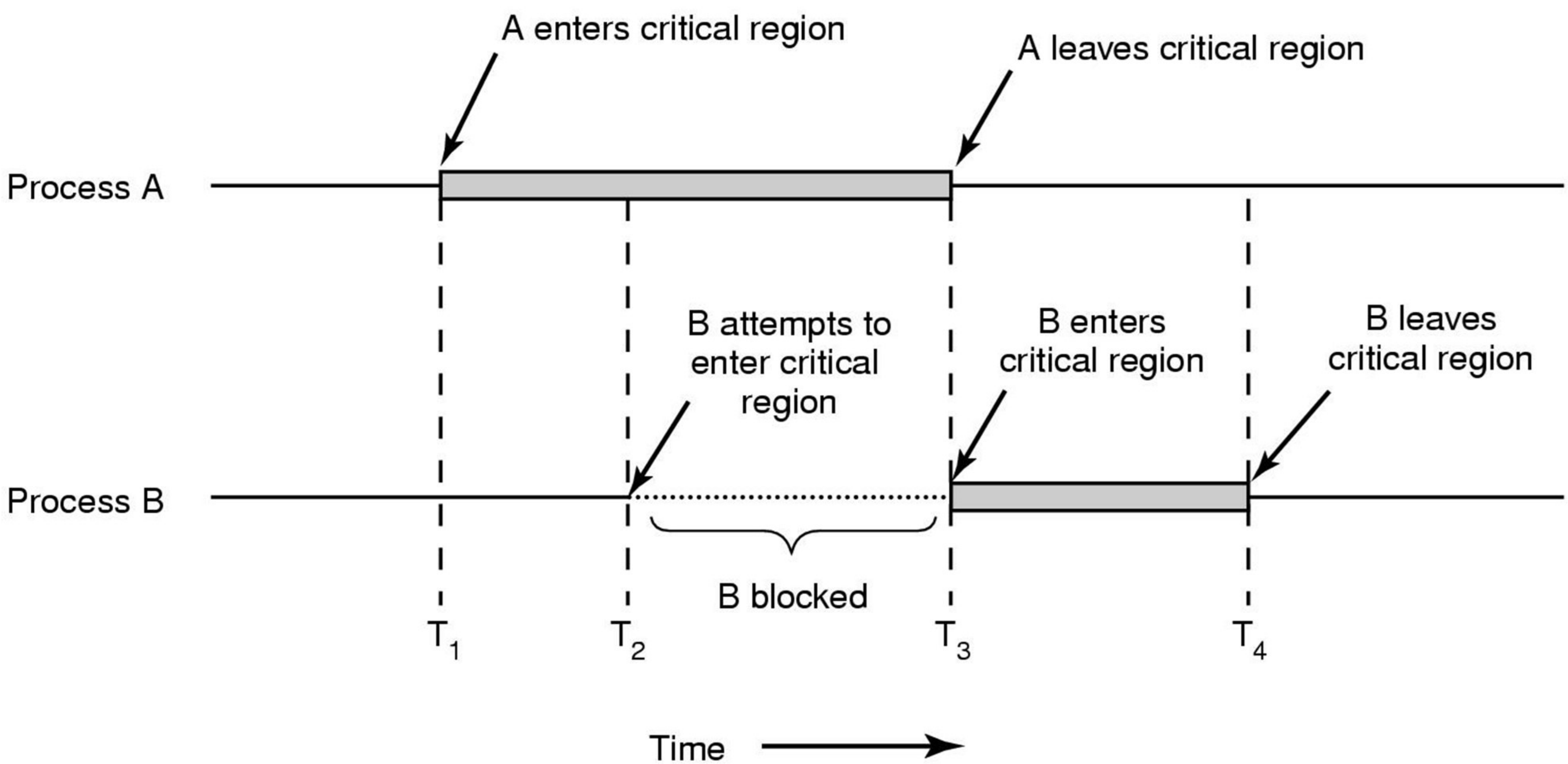


Critical Regions (1)

- Four conditions to provide mutual exclusion
  - No two processes simultaneously in critical region
  - No assumptions made about speeds or numbers of CPUs
  - No process running outside its critical region may block another process
  - No process must wait forever to enter its critical region

Critical Regions (2)

- Mutual exclusion using critical regions



Mutual Exclusion with Busy Waiting (1)

- Two simple methods
  - Disabling interrupts
  - Lock variables

Mutual Exclusion with Busy Waiting (2)

- Strict alternation
  - (a) Process 0. (b) Process 1.

```
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)

Mutual Exclusion with Busy Waiting (3)

- Peterson's solution for achieving mutual exclusion

```
#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 */
}
```

Mutual Exclusion with Busy Waiting (4)

- Entering and leaving a critical region using the TSL instruction

```
enter_region:
    TSL REGISTER,LOCK          | copy lock to register and set lock to 1
    CMP REGISTER,#0            | was lock zero?
    JNE enter_region           | if it was non zero, lock was set, so loop
    RET | return to caller; critical region entered

leave_region:
    MOVE LOCK,#0               | store a 0 in lock
    RET | return to caller
```

Mutual Exclusion with Busy Waiting (5)

- XCHG: An alternative instruction to TSL

```
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
```

Sleep and Wakeup

- Producer-consumer problem with fatal race condition

```
#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 */
     }
}
```

Semaphores

- The producer-consumer problem using semaphores

```
#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 */
     }
}
```

Mutexes

- Implementation of mutex_lock and mutex_unlock

```
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
```

Monitors (1)

- Example of a monitor

```
monitor example
        integer i;
        condition c;

        procedure producer( );
        .
        .
        .
        end;

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

Monitors (2)

- Outline of producer-consumer problem with monitors
  - only one monitor procedure active at one time
  - buffer has N slots

```
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;
```

```
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;
```

Monitors (3)

- Solution to producer-consumer problem in Java (part 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
      }
      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
      }
```

Monitors (4)

- Solution to producer-consumer problem in Java (part 2)

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

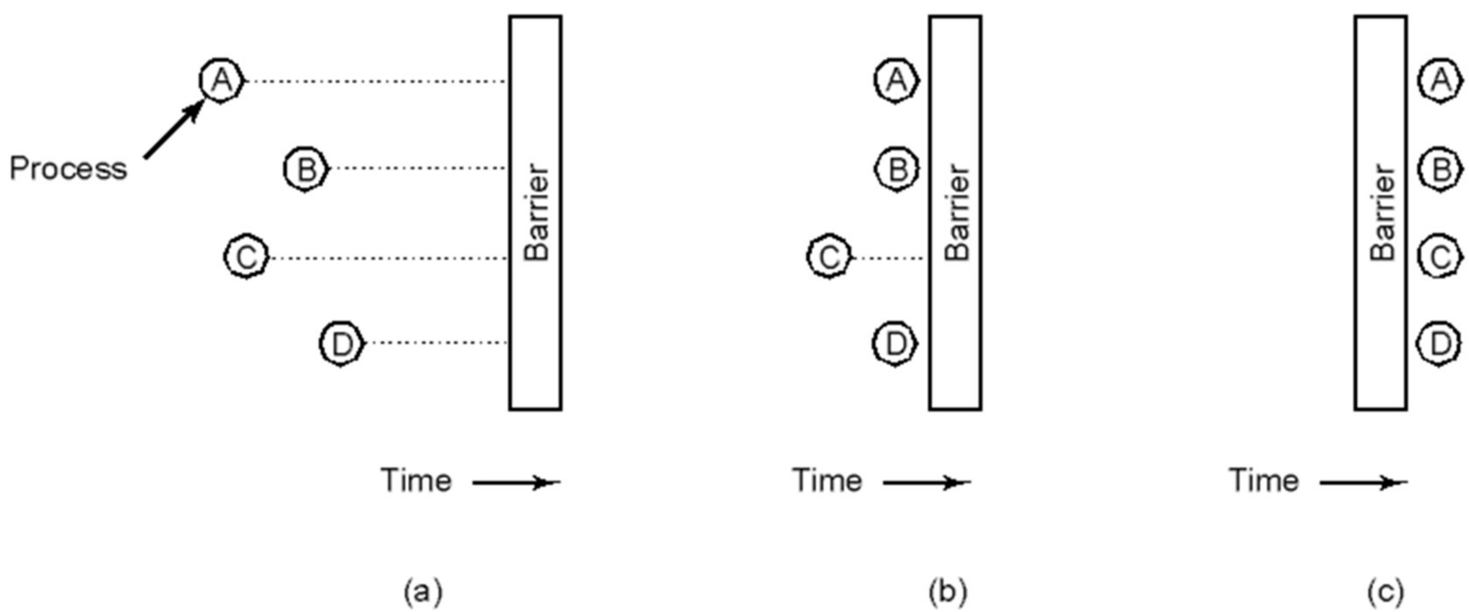
Message Passing

- The producer-consumer problem with N messages

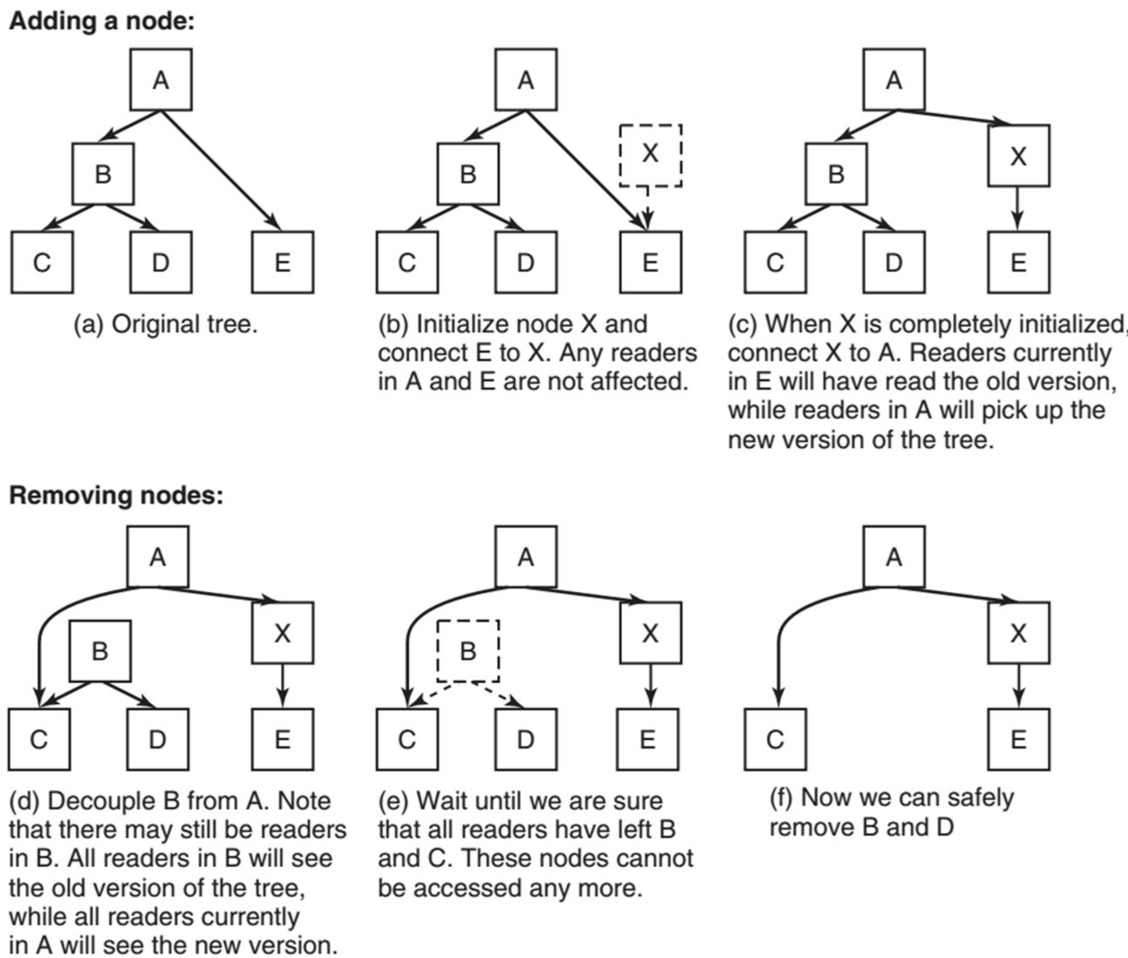
```
#define N 100                                   /* number of slots in the buffer */

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 */
     }
}
```

Barriers

- Use of a barrier
  - processes approaching a barrier
  - all processes but one blocked at barrier
  - last process arrives, all are let through



Avoiding Locks: Read-Copy-Update



(a) Original tree.

(b) Initialize node X and connect E to X. Any readers in A and E are not affected.

(c) When X is completely initialized, connect X to A. Readers currently in E will have read the old version, while readers in A will pick up the new version of the tree.

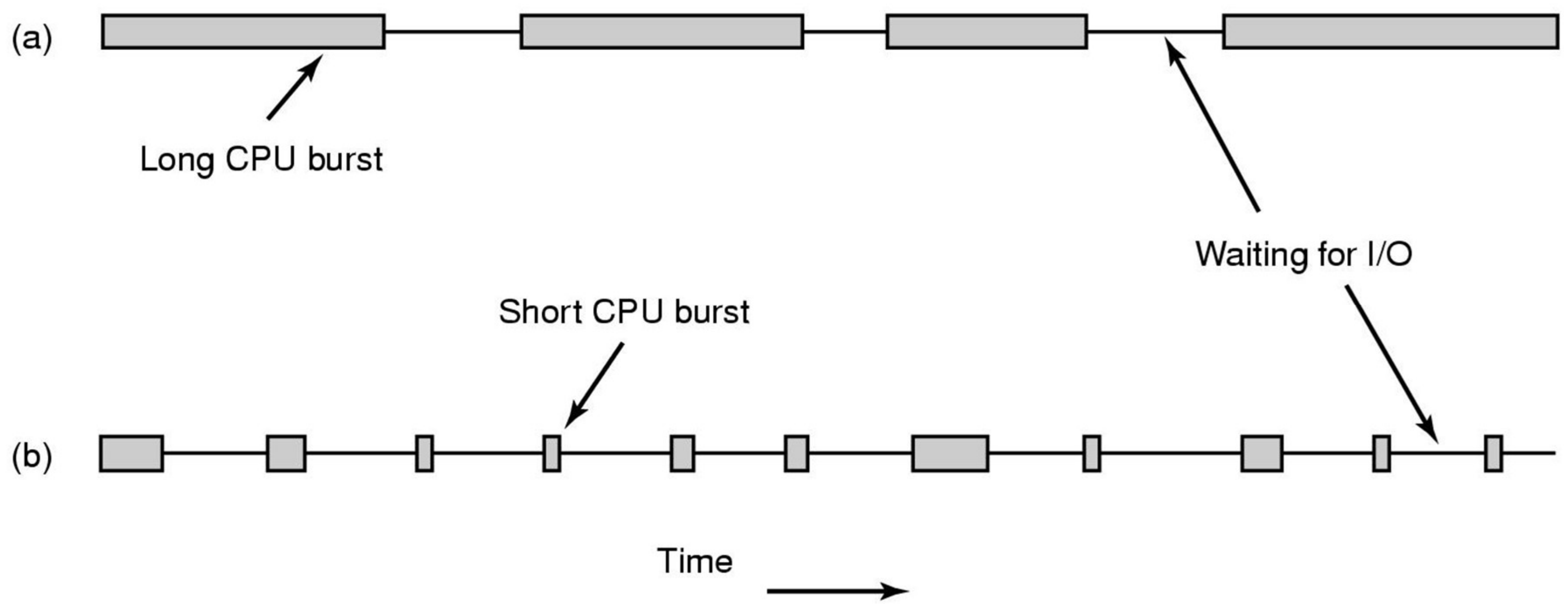
(d) Decouple B from A. Note that there may still be readers in B. All readers in B will see the old version of the tree, while all readers currently in A will see the new version.

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

(f) Now we can safely remove B and D

Introduction to Scheduling (1)

- Bursts of CPU usage alternate with periods of I/O wait
  - a CPU-bound process
  - an I/O bound process



Introduction to Scheduling (2)

- When to schedule
  - Creation of a new process
  - Exiting of a process
  - Blocking a process
  - I/O interrupt
  - Clock period
    - Preemptive
    - Non-preemptive

Introduction to Scheduling (3)

- Categories of Scheduling Algorithms
  - Batch
  - Interactive
  - Real-time

Introduction to Scheduling (4)

- 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

Scheduling in Batch Systems

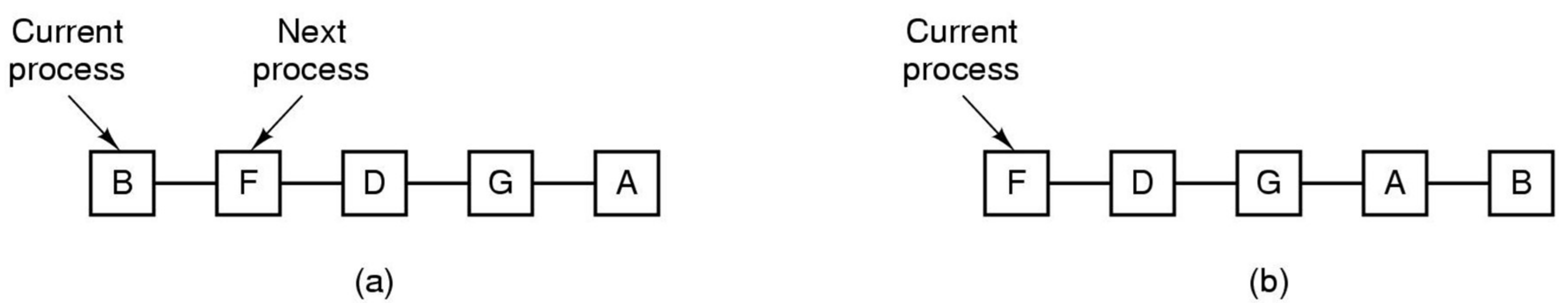
- First-Come, First-Served
  - Adv: easy to understand and program
  - disAdv: long delays for io-bound processes
- shortest job first

(a) (b)

- Shortest Remaining Time Next

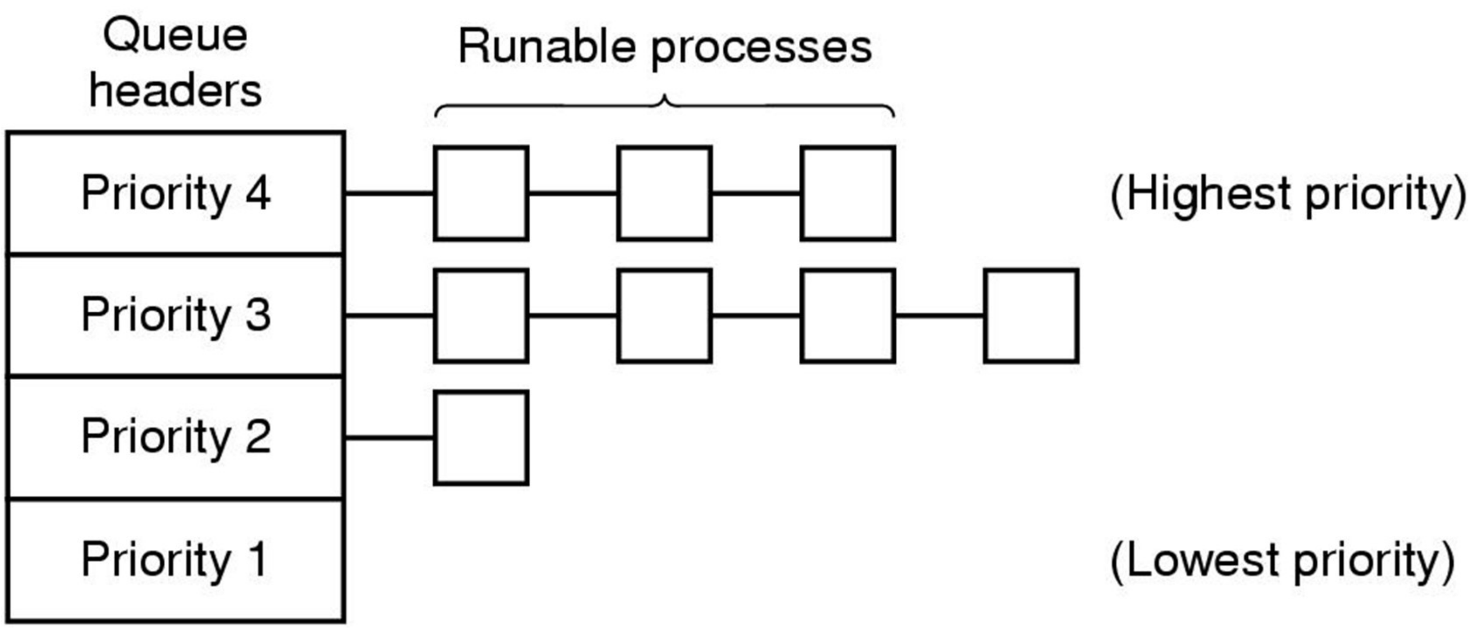
Scheduling in Interactive Systems (1)

- Round Robin Scheduling
  - list of runnable processes
  - list of runnable processes after B uses up its quantum



Scheduling in Interactive Systems (2)

- A scheduling algorithm with four priority classes



Scheduling in Interactive Systems (3)

- Multiple Queues
  - Idea: occasionally large quantum for CPU-bound processes
  - Implementation:
    - Set up priority classes
    - Highest class: one quantum
    - Next-highest: two quanta
    - Next one: four quanta, etc.
    - Move down process that used all of its quanta one class
  - Problem
    - punishing process that runs for a long time and becomes interactive later

Scheduling in Interactive Systems (4)

- Shortest Process Next
  - Idea: use SJF for interactive processes
  - Implementation:

$$aT_0 + (1 - a)T_1$$

- Guaranteed Scheduling
  - Idea: promise about 1/n of the CPU cycles
  - Implementation:
    - Ratio: actual assigned time / entitled time
    - Run the process with the lowest ratio

Scheduling in Interactive Systems (5)

- Lottery Scheduling
  - Idea: give processes lottery tickets
  - Interesting properties
    - Highly responsive
    - Possible exchanging of tickets
    - Can solve problems that are difficult for other methods
      - Example: video server with different frame rates

Scheduling in Interactive Systems (6)

- Fair-Share Scheduling
  - Idea: taking into account the owner of processes
  - Implementation:
    - Allocate some fraction of CPU time to each user
    - Run processes in such a way to enforce it

Scheduling in Real-Time Systems

- Schedulable real-time system
  - Given
    - m periodic events
    - event i occurs within period Pi and requires Ci seconds
  - Then the load can only be handled if

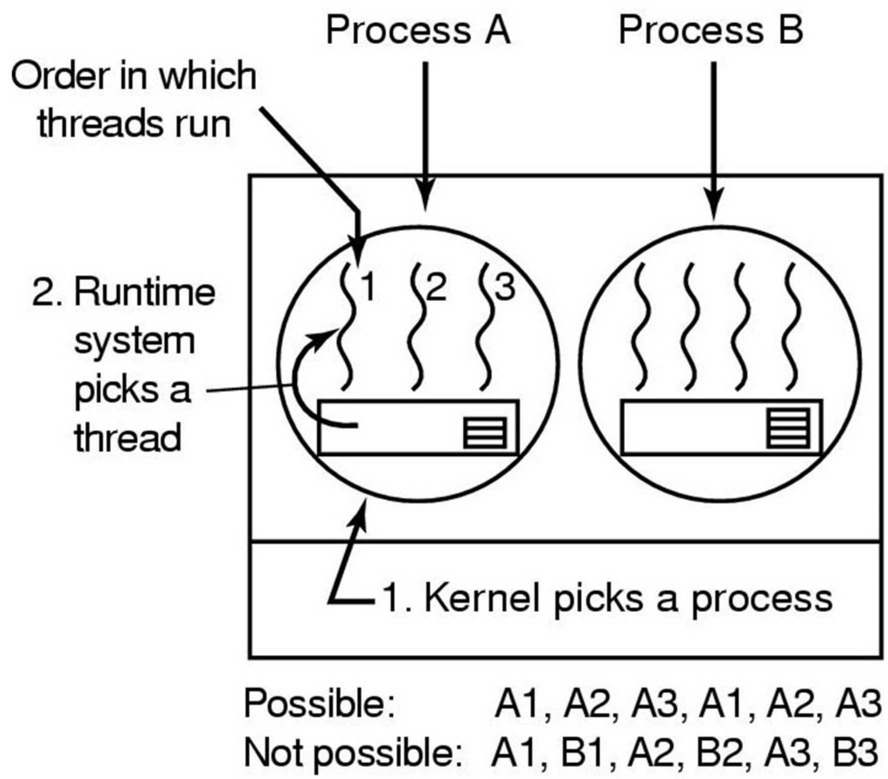
$$\sum_{i=1}^{m} \frac{C_i}{P_i} \leq 1$$

Policy versus Mechanism

- Separate what is allowed to be done with how it is done
  - a process knows which of its children threads are important and need priority
  - Scheduling algorithm parameterized
    - mechanism in the kernel
  - Parameters filled in by user processes
    - policy set by user process

Thread Scheduling (1)

- Possible scheduling of user-level threads
  - 50-msec process quantum
  - threads run 5 msec/CPU burst



Thread Scheduling (2)

- Possible scheduling of kernel-level threads
  - 50-msec process quantum
  - threads run 5 msec/CPU burst