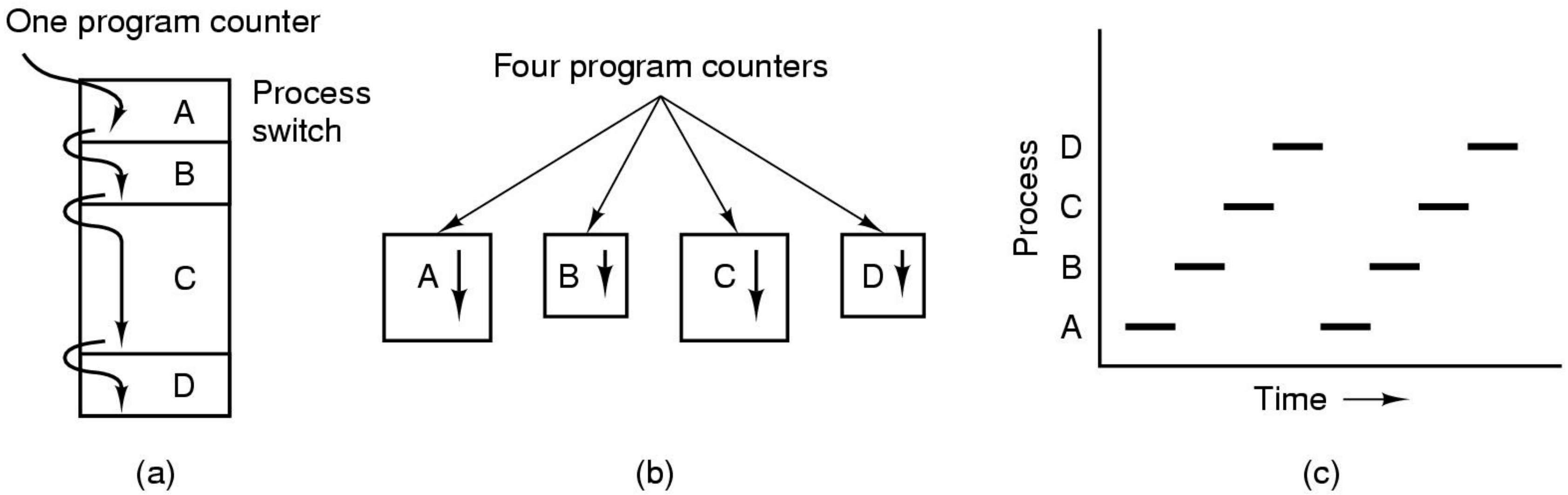
Lecture 2

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

1. System initialization
- Execution of a process creation system
1. User request to create a new process
2. Initiation of a batch job

Process Termination

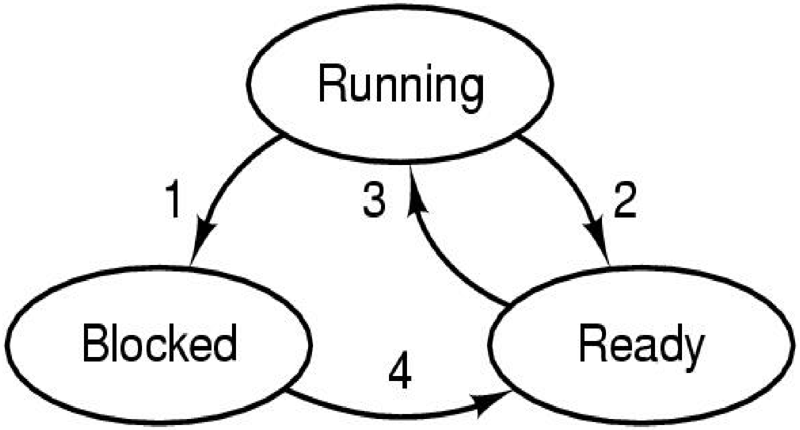
Conditions which terminate processes

1. Normal exit (voluntary)
2. Error exit (voluntary)
3. Fatal error (involuntary)
4. 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

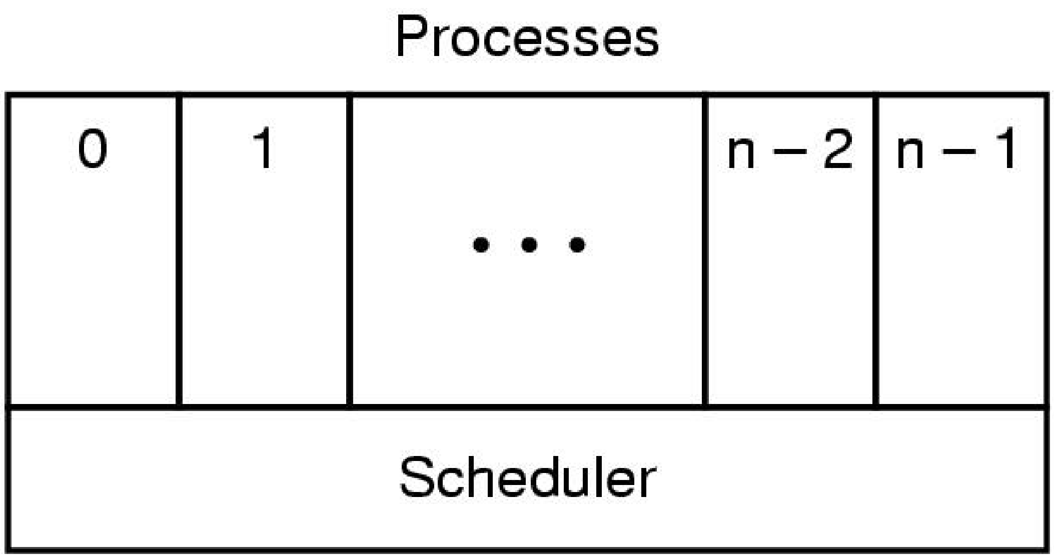
Process States (1)



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

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

Process States (2)



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

Implementation of Processes (1)

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		

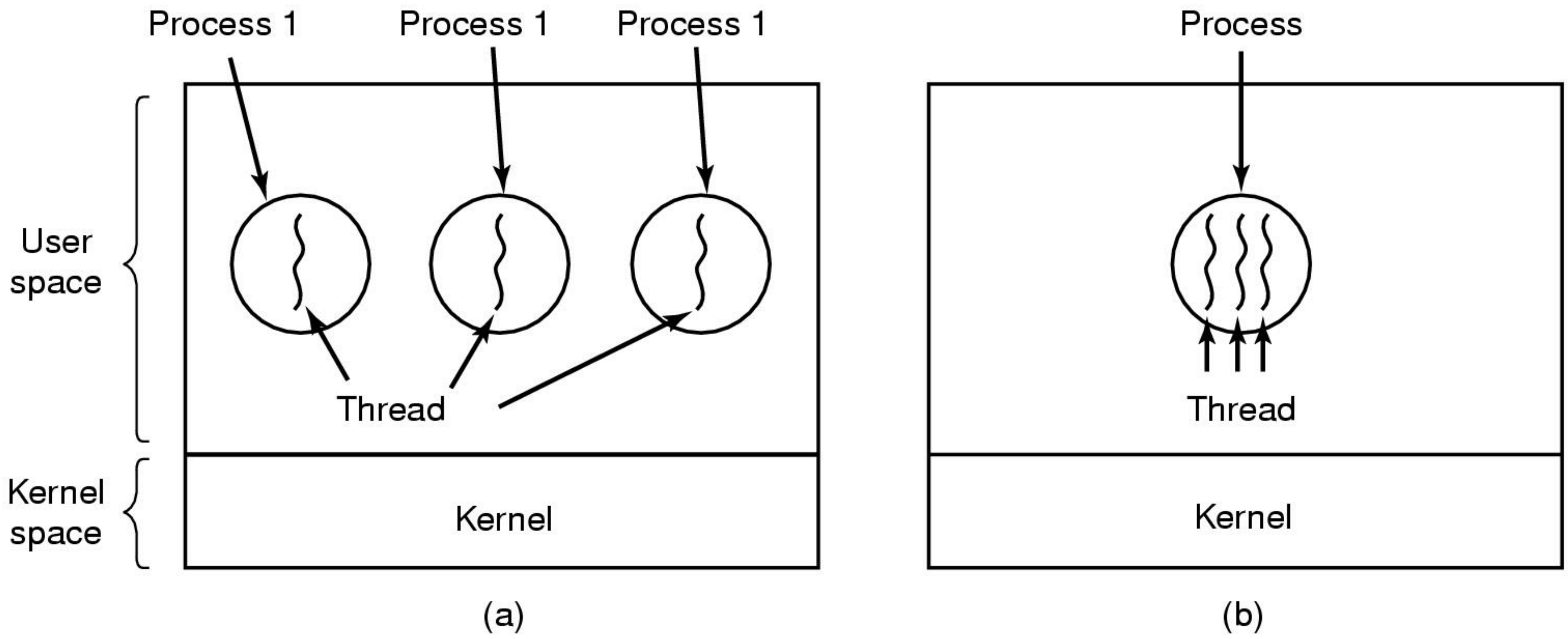
Fields of a process table entry

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.

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

Threads
The Thread Model (1)



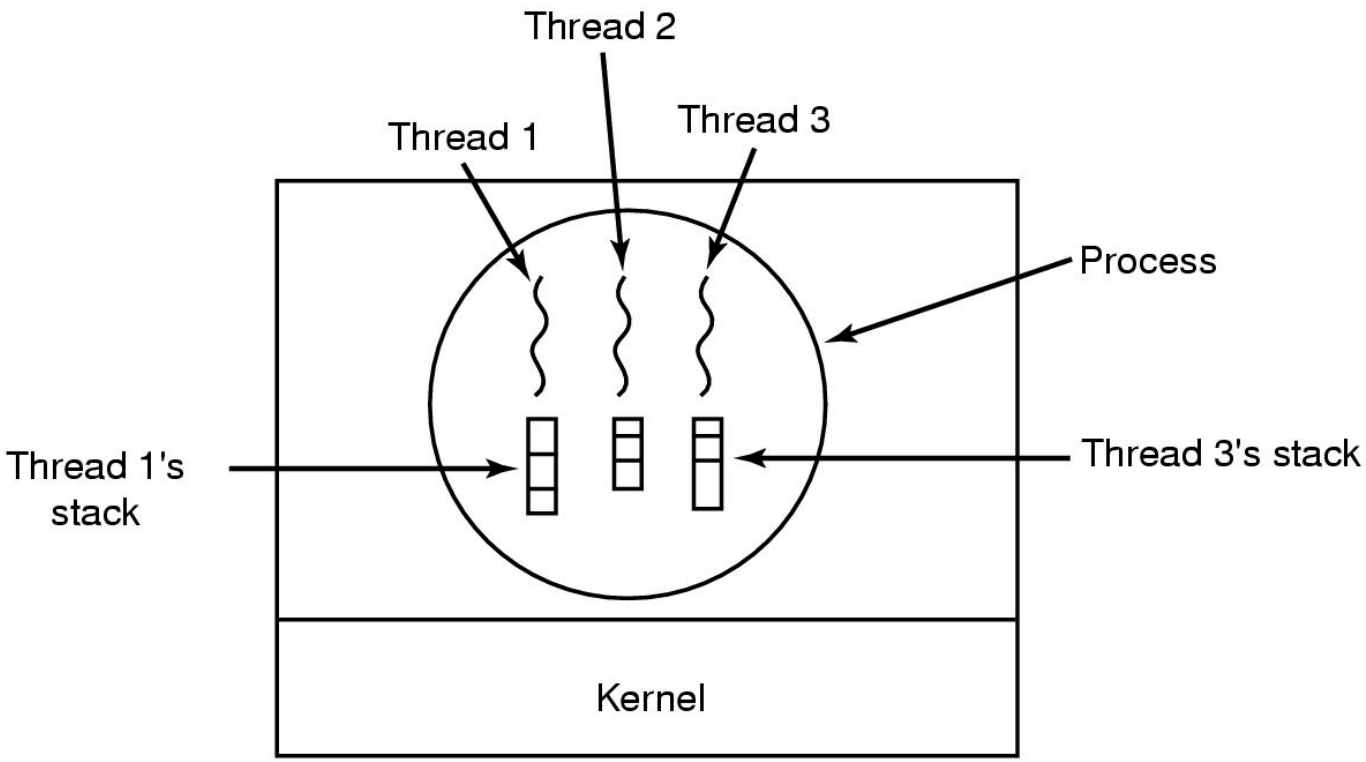
(a) Three processes each with one thread
(b) One process with three threads

The Thread Model (2)

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

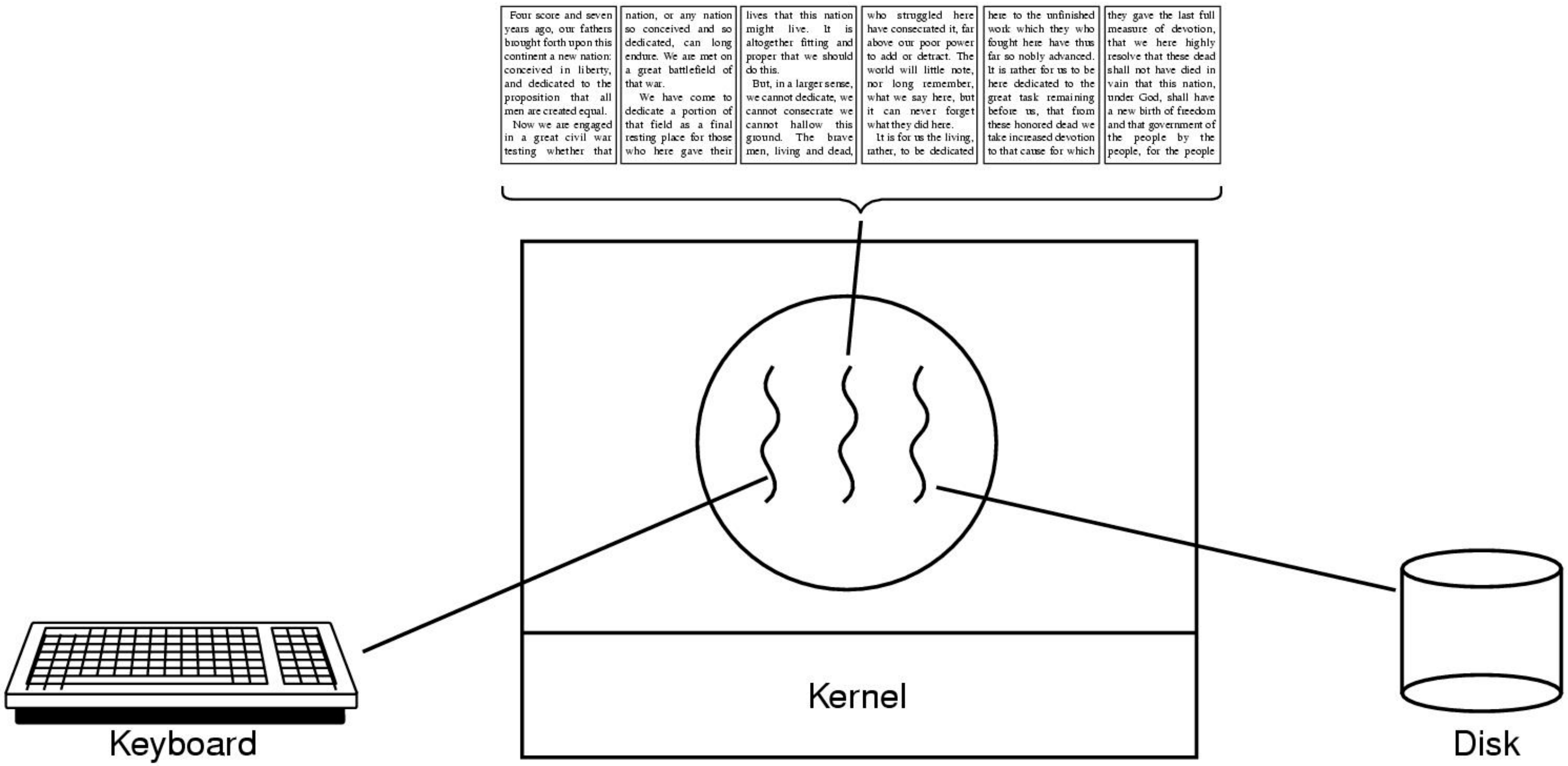
- Items shared by all threads in a process
- Items private to each thread

The Thread Model (3)



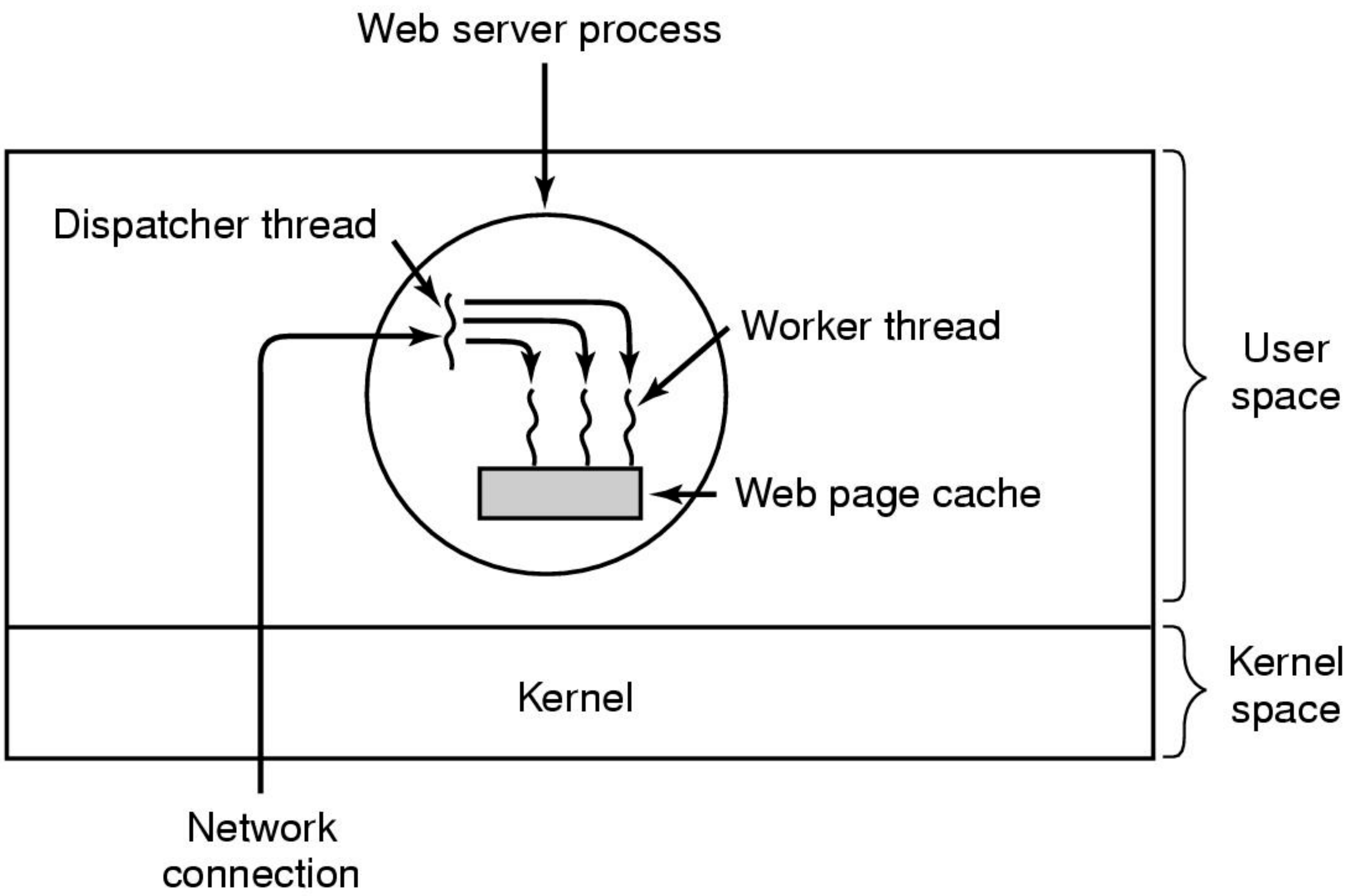
Each thread has its own stack

Thread Usage (1)



A word processor with three threads

Thread Usage (2)



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)

- Rough outline of code for previous slide

  (a) Dispatcher thread

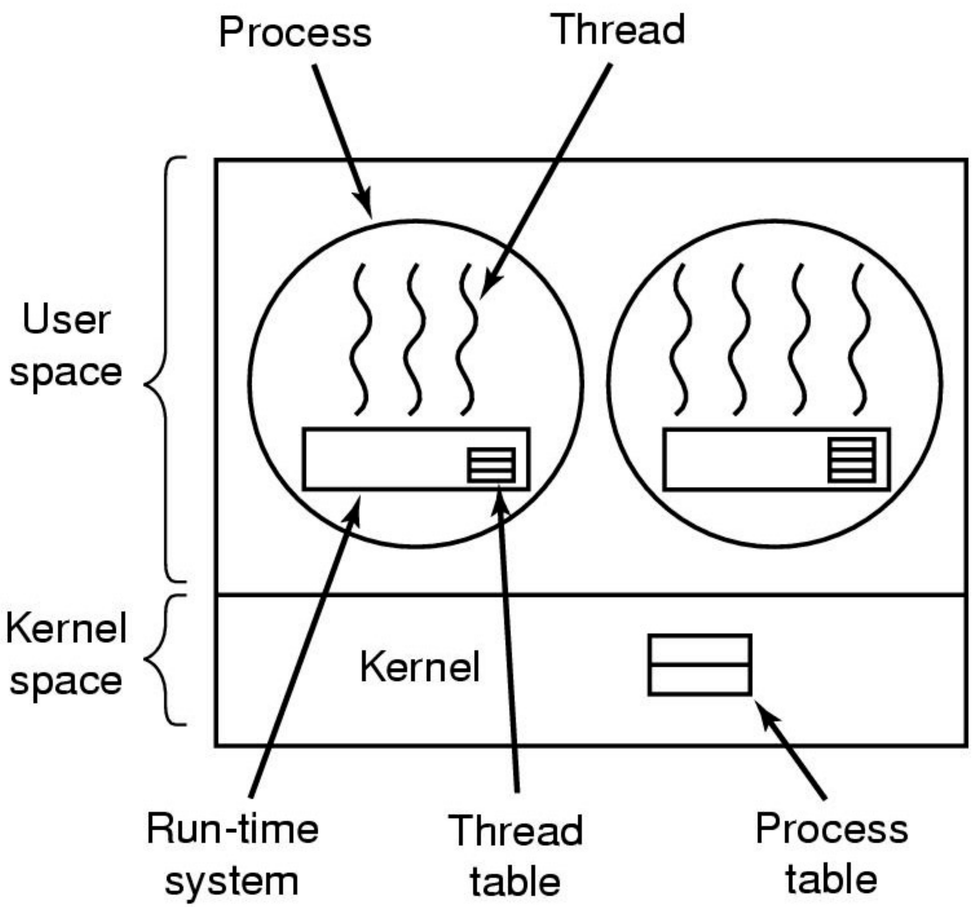
  (b) Worker thread

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

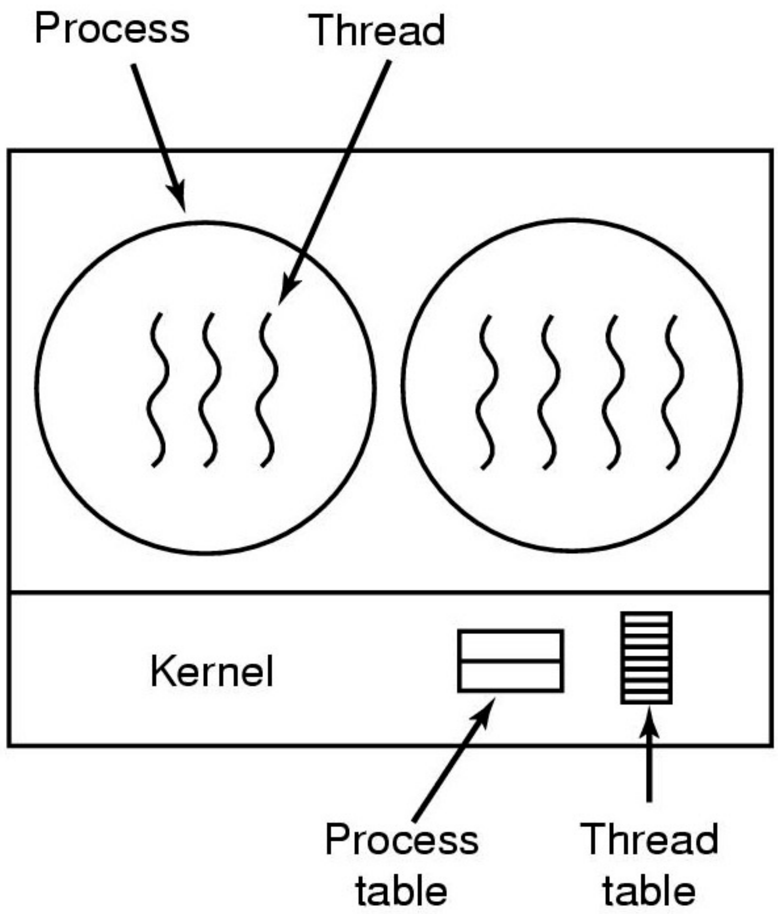
Three ways to construct a server

Implementing Threads in User Space



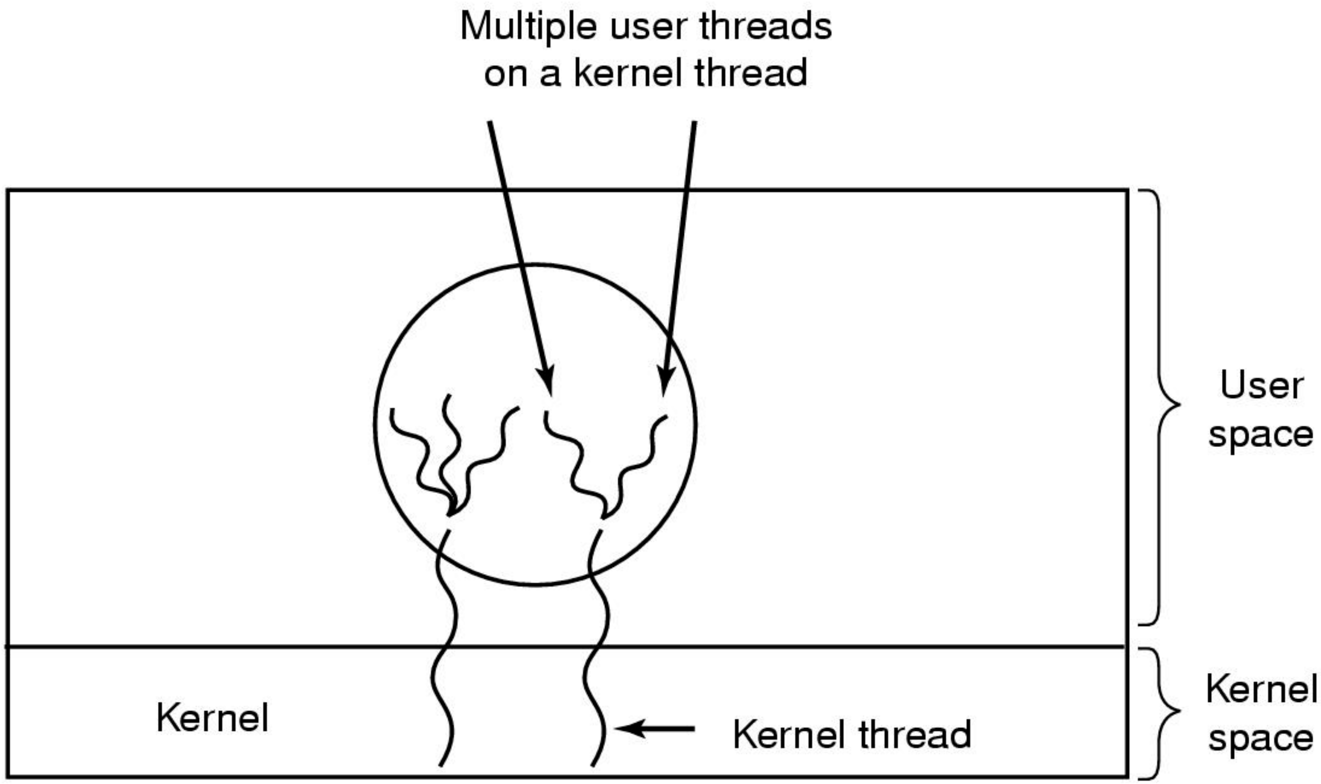
A user-level threads package

Implementing Threads in the Kernel



A threads package managed by the kernel

Hybrid Implementations

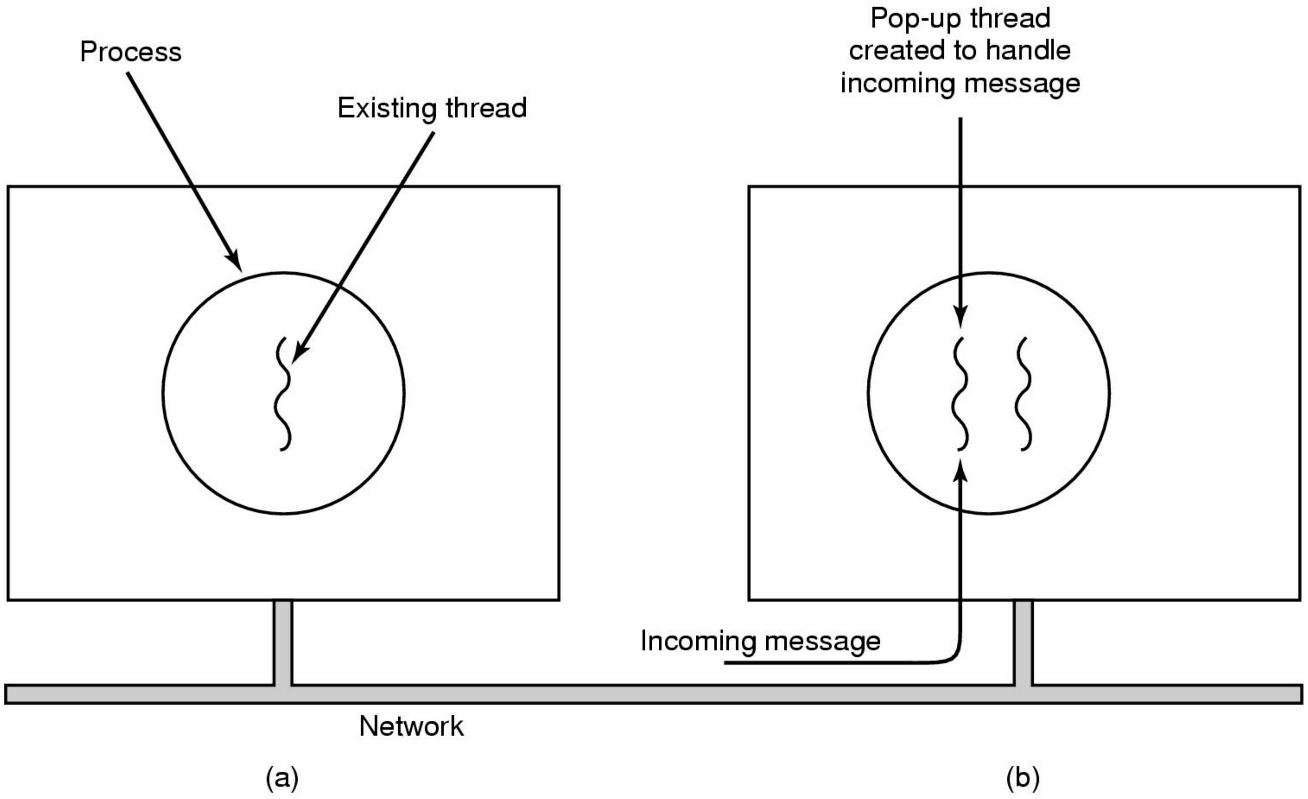


Multiplexing user-level threads onto kernel-level threads

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

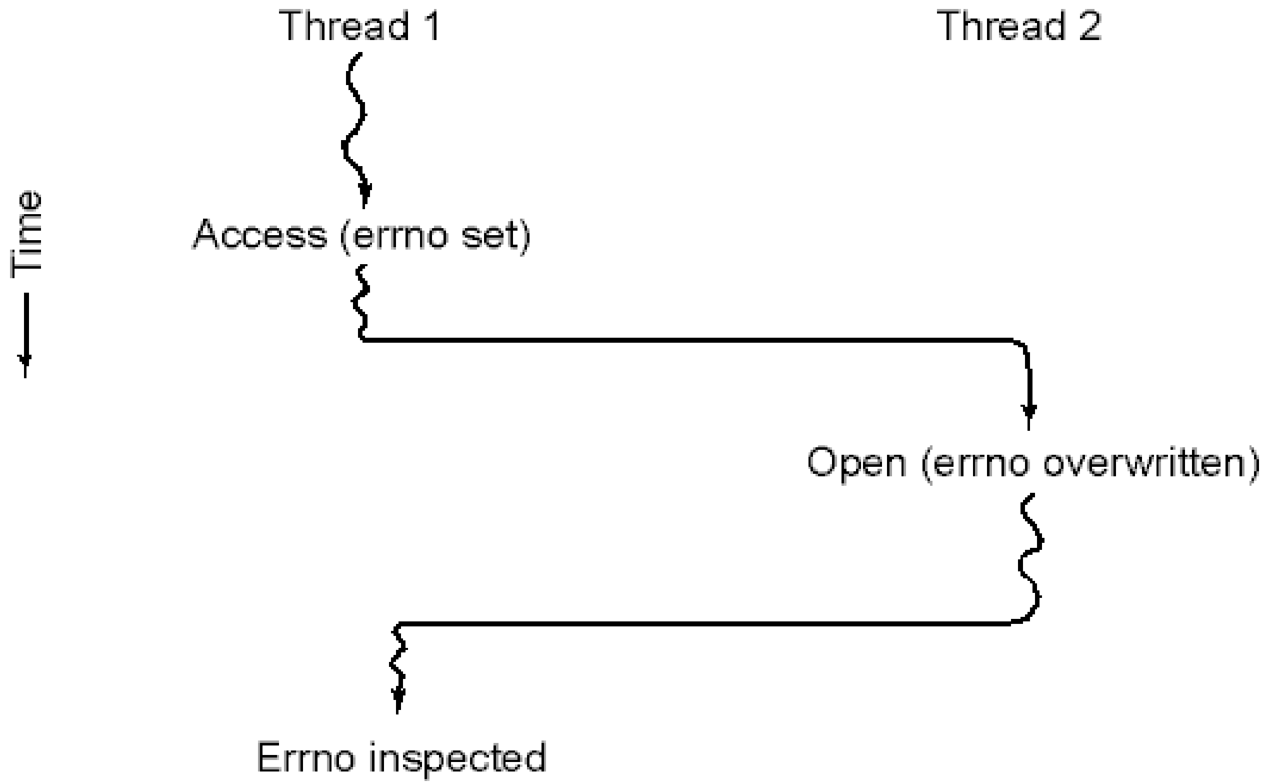


- Creation of a new thread when message arrives

  (a) before message arrives

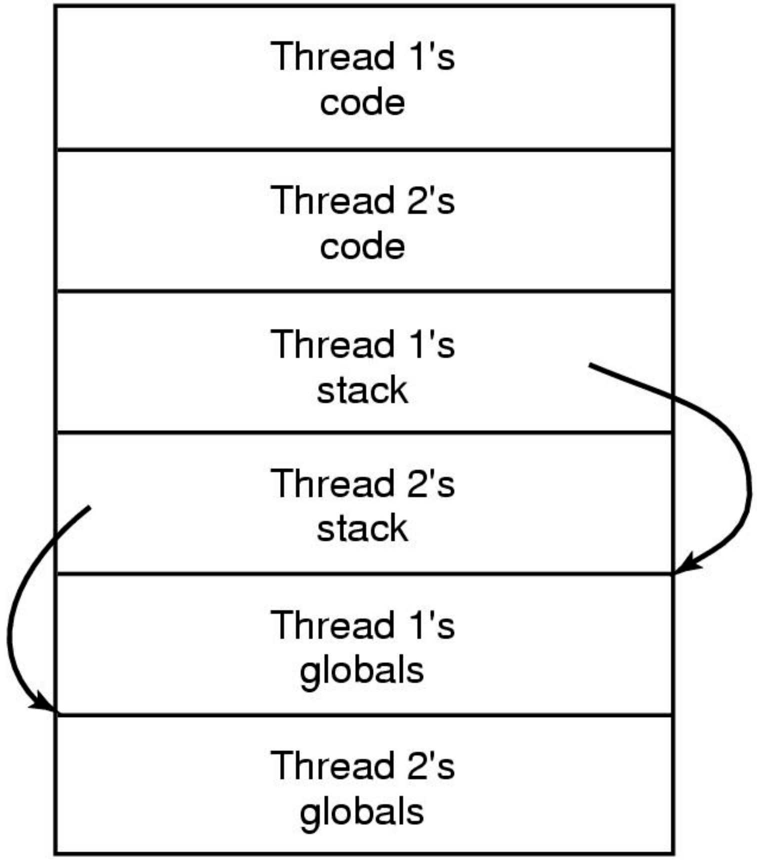
  (b) after message arrives

Making Single-Threaded Code Multithreaded (1)



Conflicts between threads over the use of a global variable

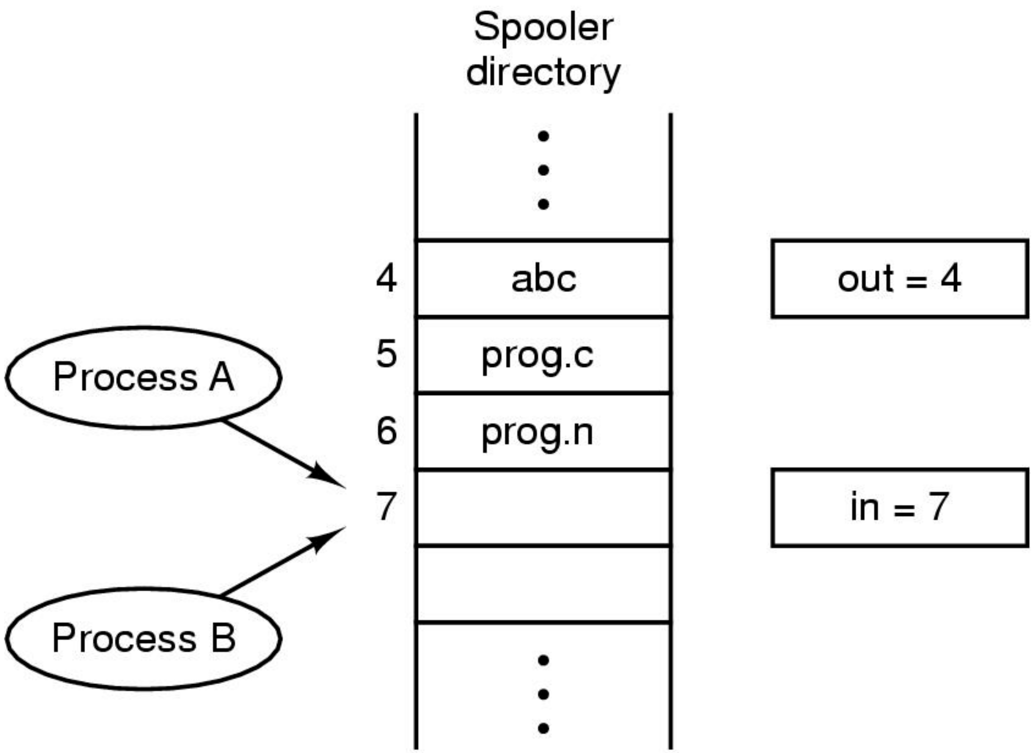
Making Single-Threaded Code Multithreaded (2)



Threads can have private global variables

Interprocess Communication

Race Conditions



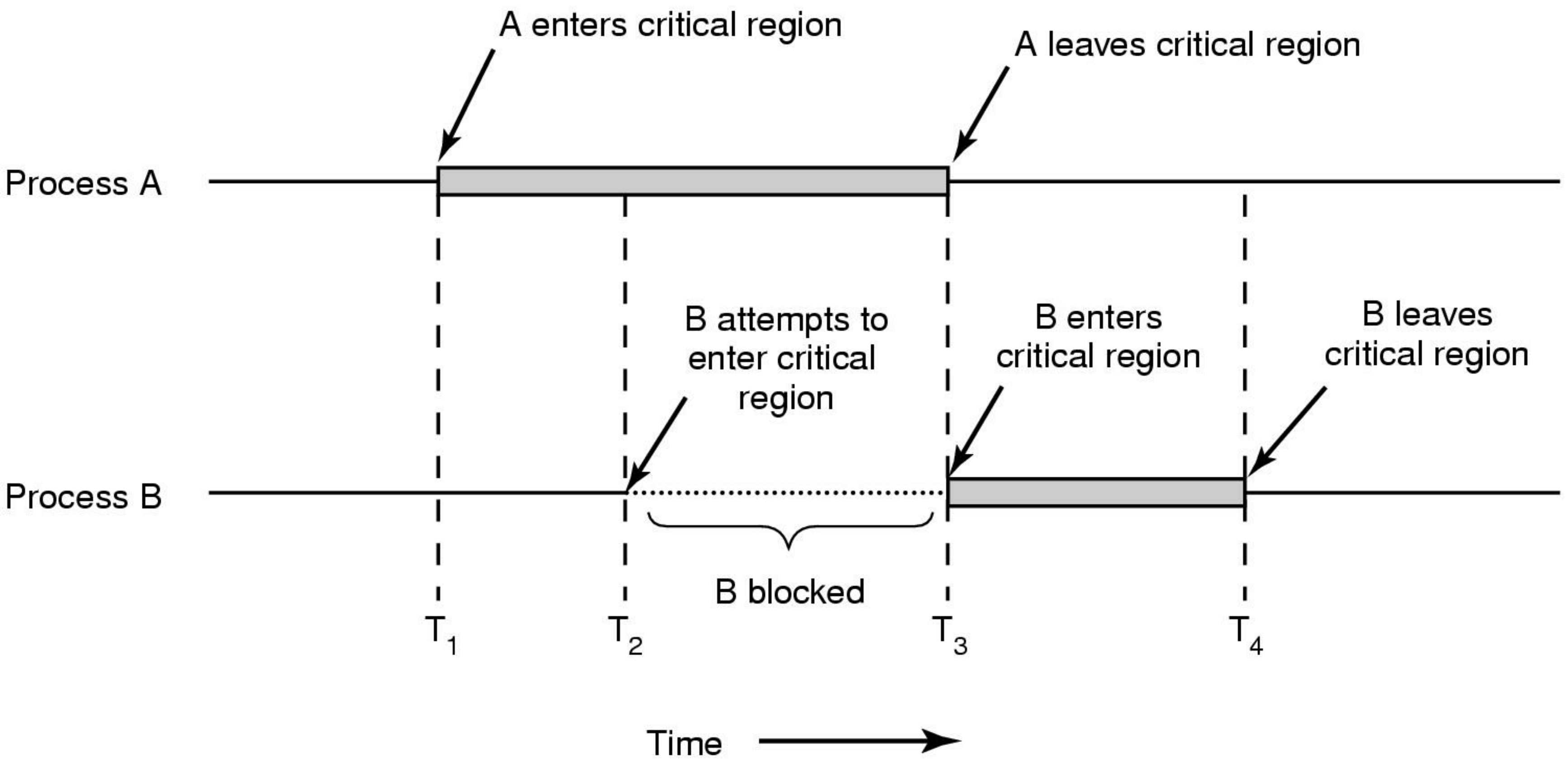
Two processes want to access shared memory at same time

Critical Regions (1)

Four conditions to provide mutual exclusion

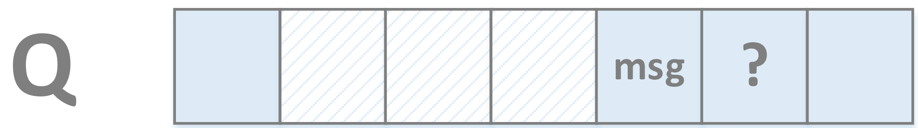
1. No two processes simultaneously in critical region
2. No assumptions made about speeds or numbers of CPUs
3. No process running outside its critical region may block another process
4. No process must wait forever to enter its critical region

Critical Regions (2)



Mutual exclusion using critical regions

Critical Region - Queue



P1

1) Q[I]<-msg1
2) I<-I+1

P2

3)Q[I]<-msg2
4)I<-I+1

Critical region problem

Mutual Exclusion with Busy Waiting (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)

Proposed solution to critical region problem

(a) Process 0. (b) Process 1.

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

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

Implementation of *mutex_lock* and *mutex_unlock*

Monitors (1)

```
monitor example
        integer i;
        condition c;

        procedure producer( );

        .
        .
        .
        end;

        procedure consumer( );

        .
        .
        .
        end;
end monitor;
```

Example of 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;
```

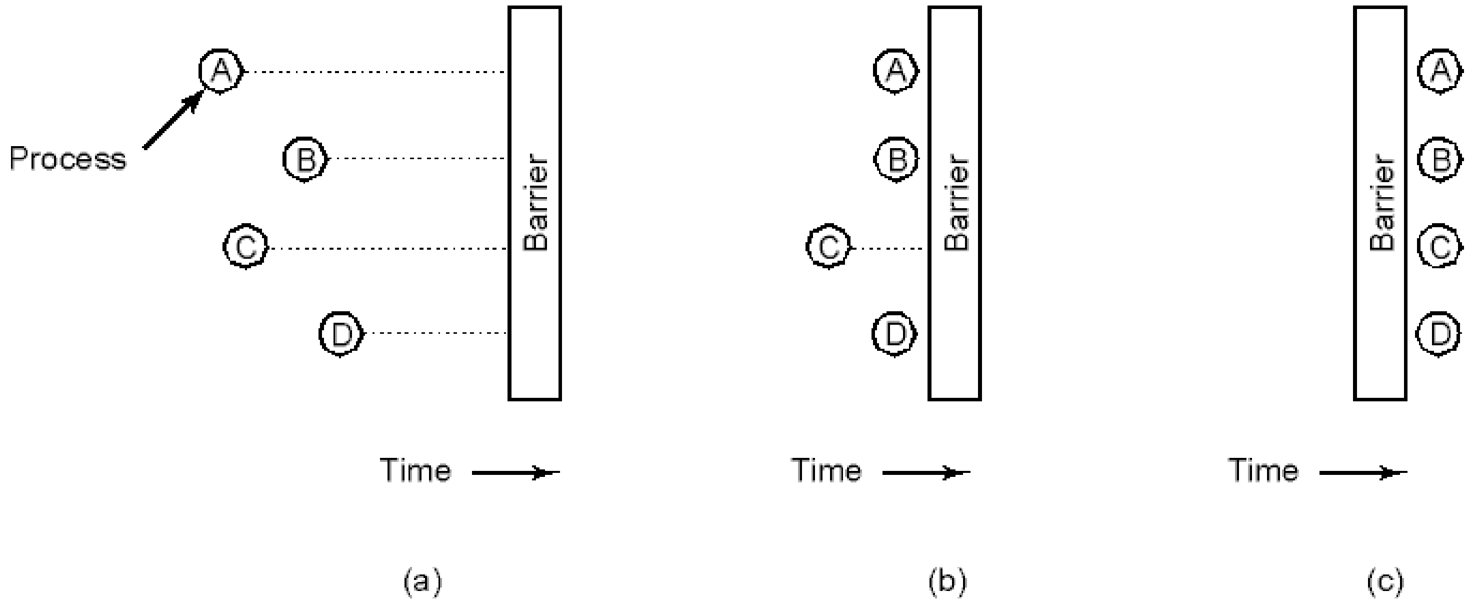
- Outline of producer-consumer problem with monitors
    - only one monitor procedure active at one time
    - buffer has *N* slots

Message Passing

```
#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 */
    }
}
```

The producer-consumer problem with N messages

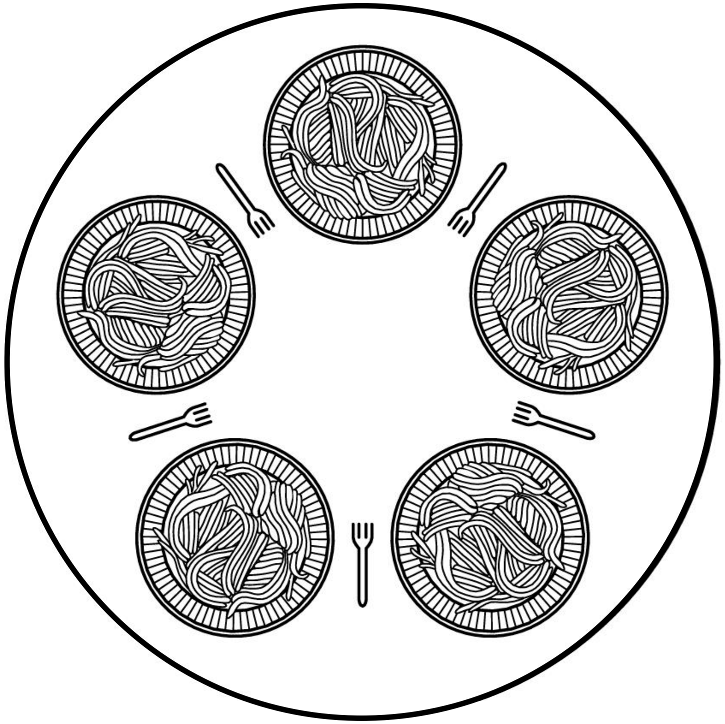
Barriers



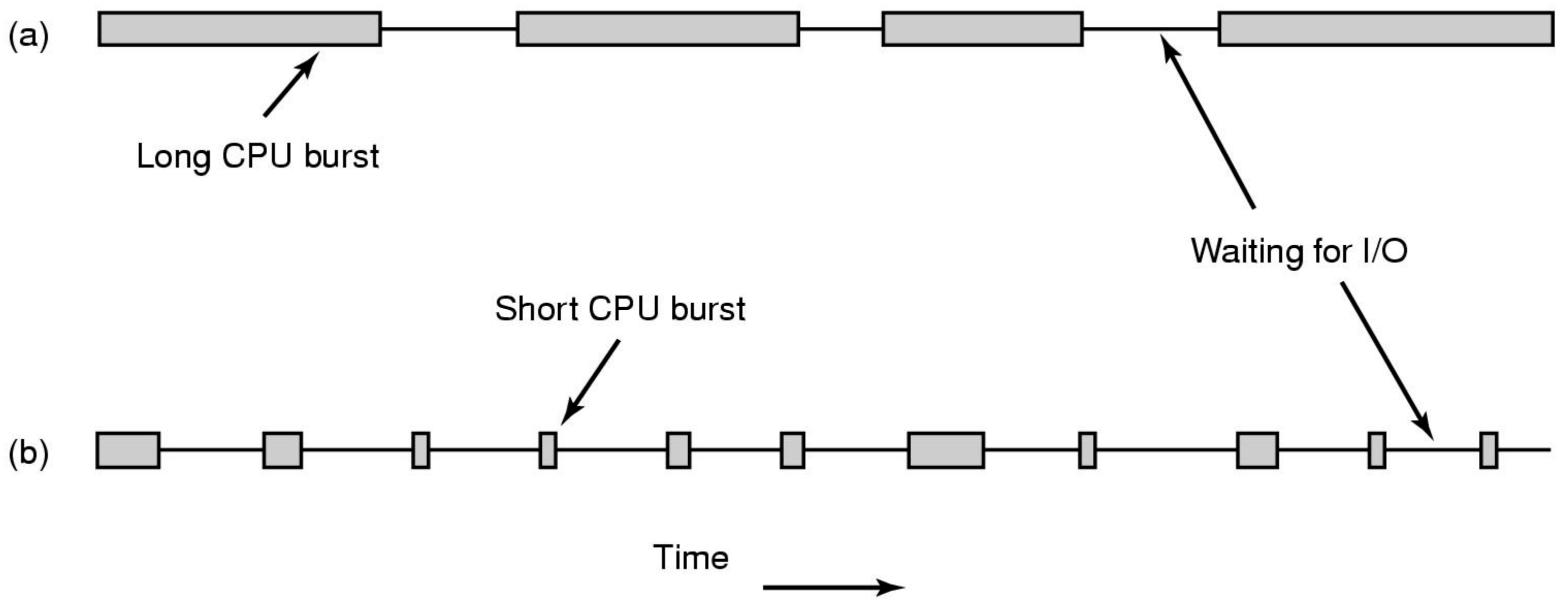
- Use of a barrier
  - processes approaching a barrier
  - all processes but one blocked at barrier
  - last process arrives, all are let through

Dining Philosophers (1)

- Philosophers eat/think
- Eating needs 2 forks
- Pick one fork at a time
- How to prevent deadlock



Scheduling
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)

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

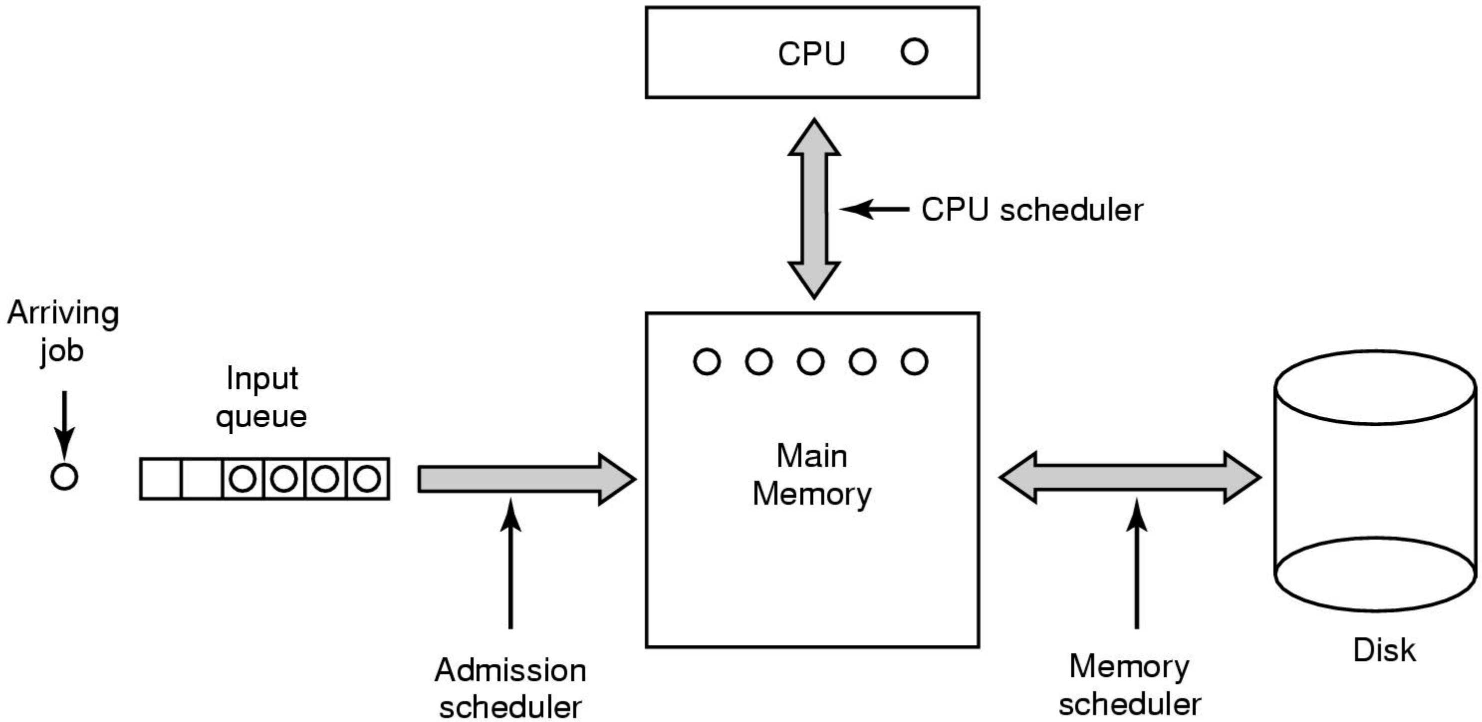
Meeting deadlines - avoid losing data
Predictability - avoid quality degradation in multimedia systems

Scheduling Algorithm Goals

Scheduling in Batch Systems (1)

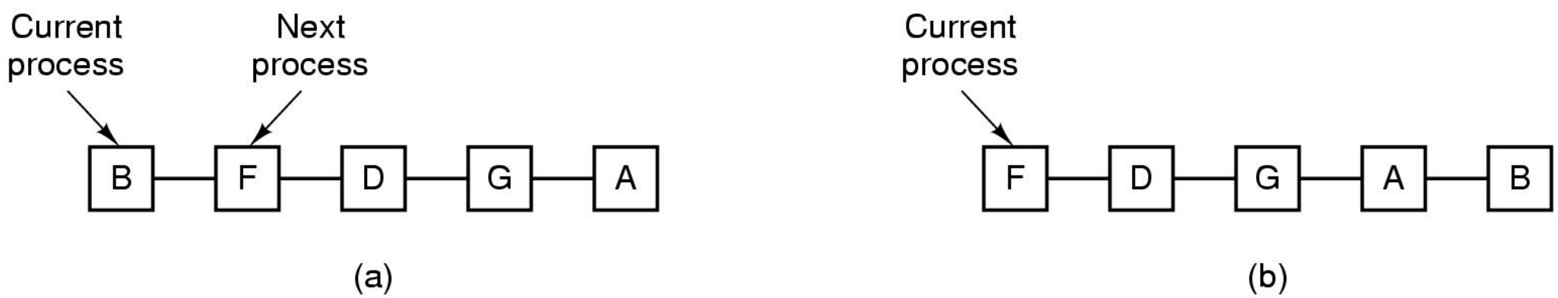
An example of shortest job first scheduling

Scheduling in Batch Systems (2)



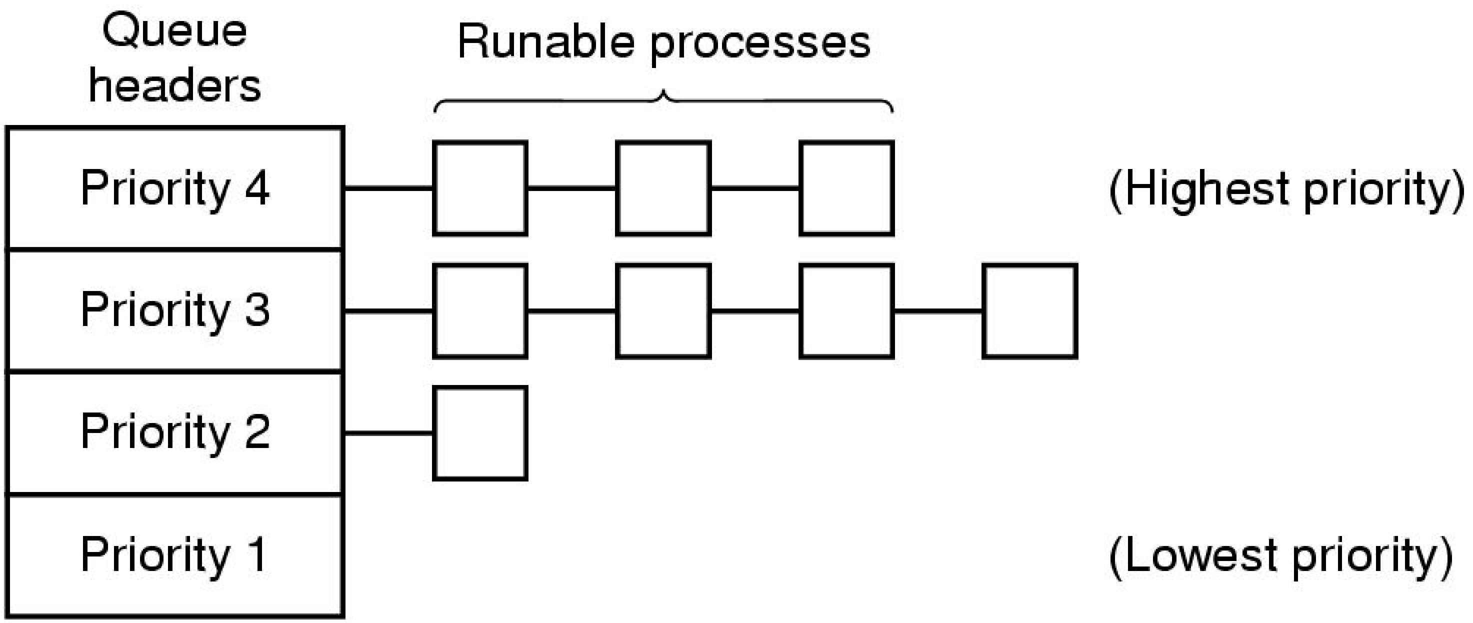
Three level scheduling

Scheduling in Interactive Systems (1)



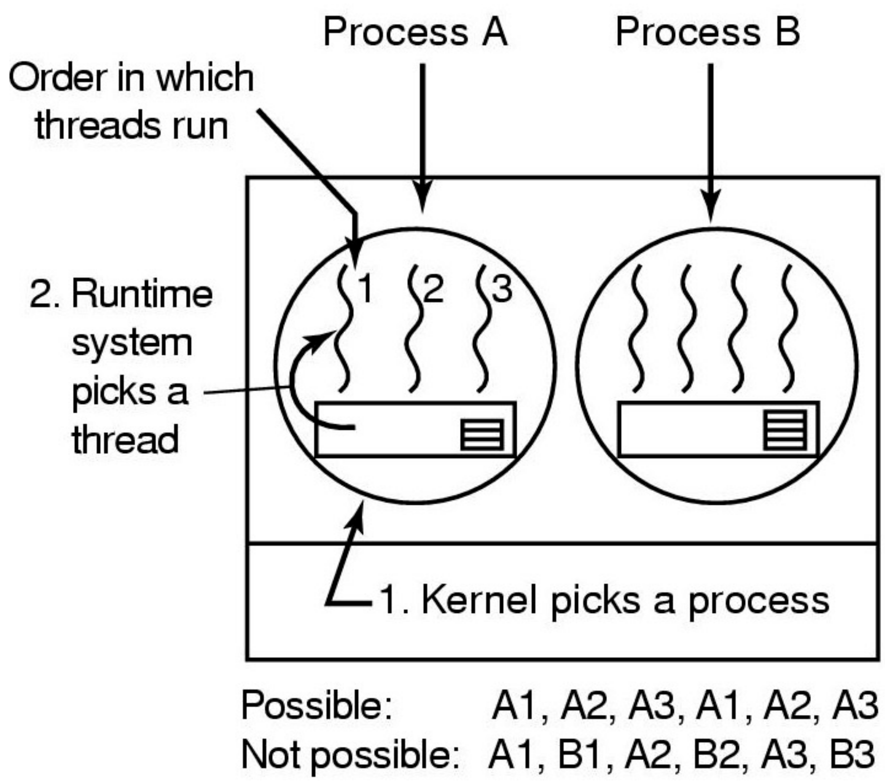
- Round Robin Scheduling
  - list of runnable processes
  - list of runnable processes after B uses up its quantum

Scheduling in Interactive Systems (2)



A scheduling algorithm with four priority classes

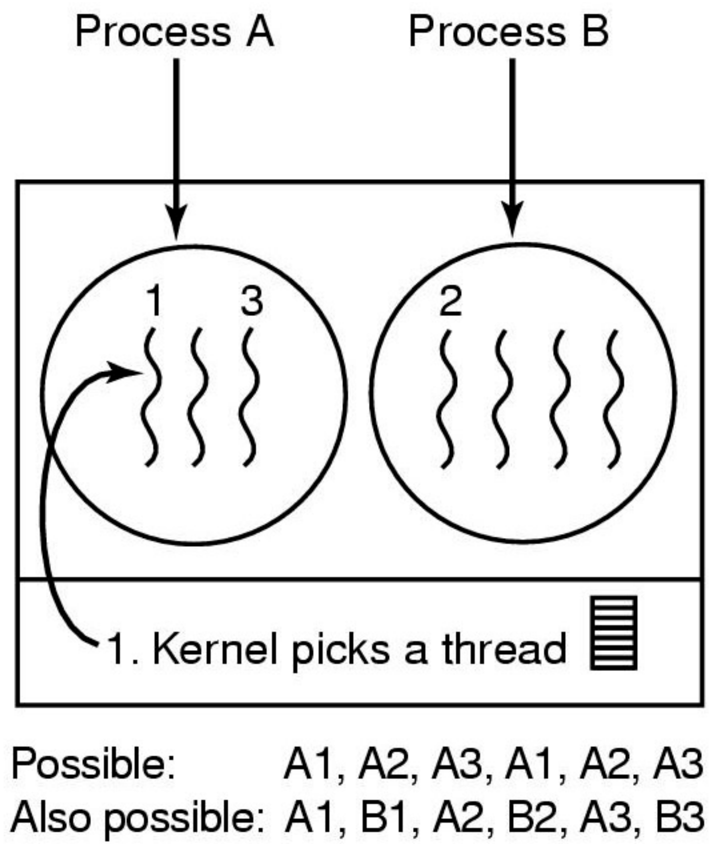
Thread Scheduling (1)



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