Distributed Operating Systems

Inter-Process Communication

Process Scheduling

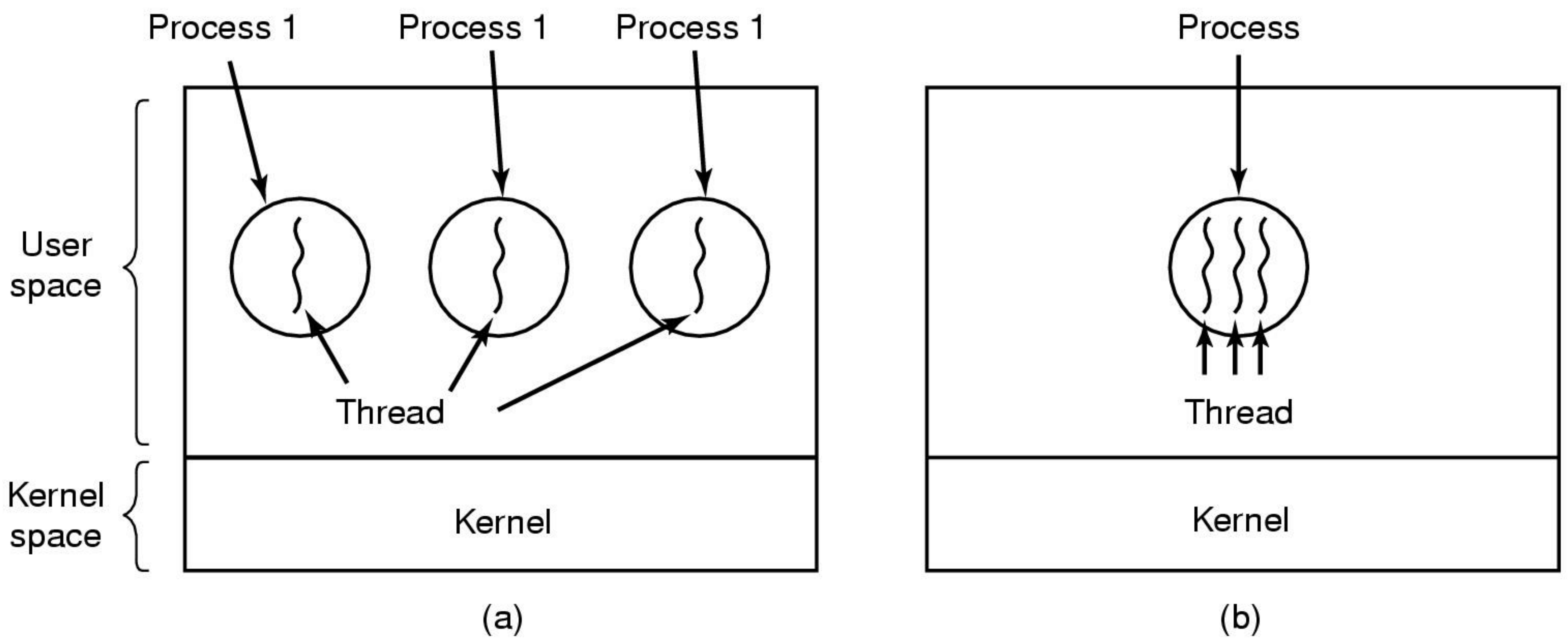
Process Concept

- Process – a program in execution; process execution must progress in sequential fashion
- A process includes:
  - program counter
  - stack
  - data section

Threads

- A **thread** of execution is the smallest sequence of programmed instructions that can be managed independently by an operating system scheduler.
- The processor switches between different threads (context switching)

Threads
The Thread Model (1)



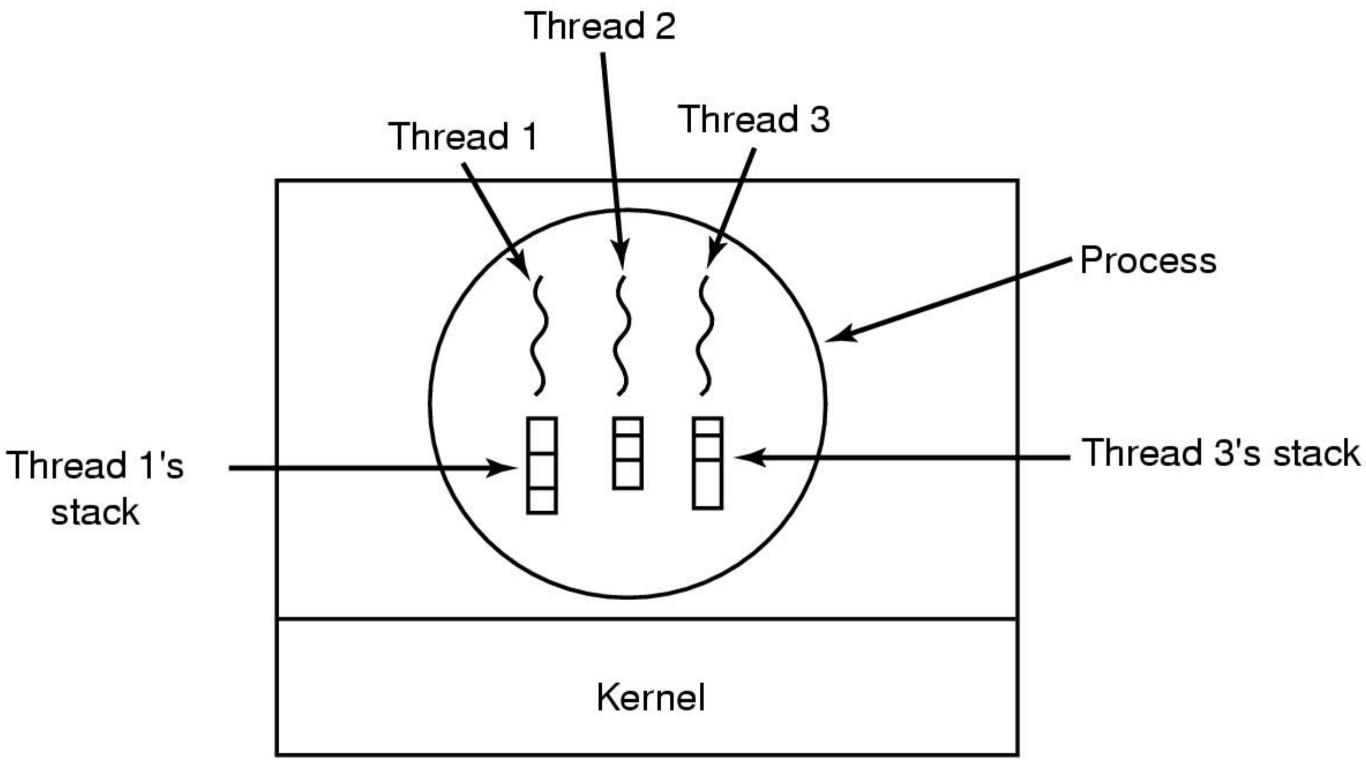
(a) Three processes each with one thread
(b) One process with three threads

The Thread Model (2)

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

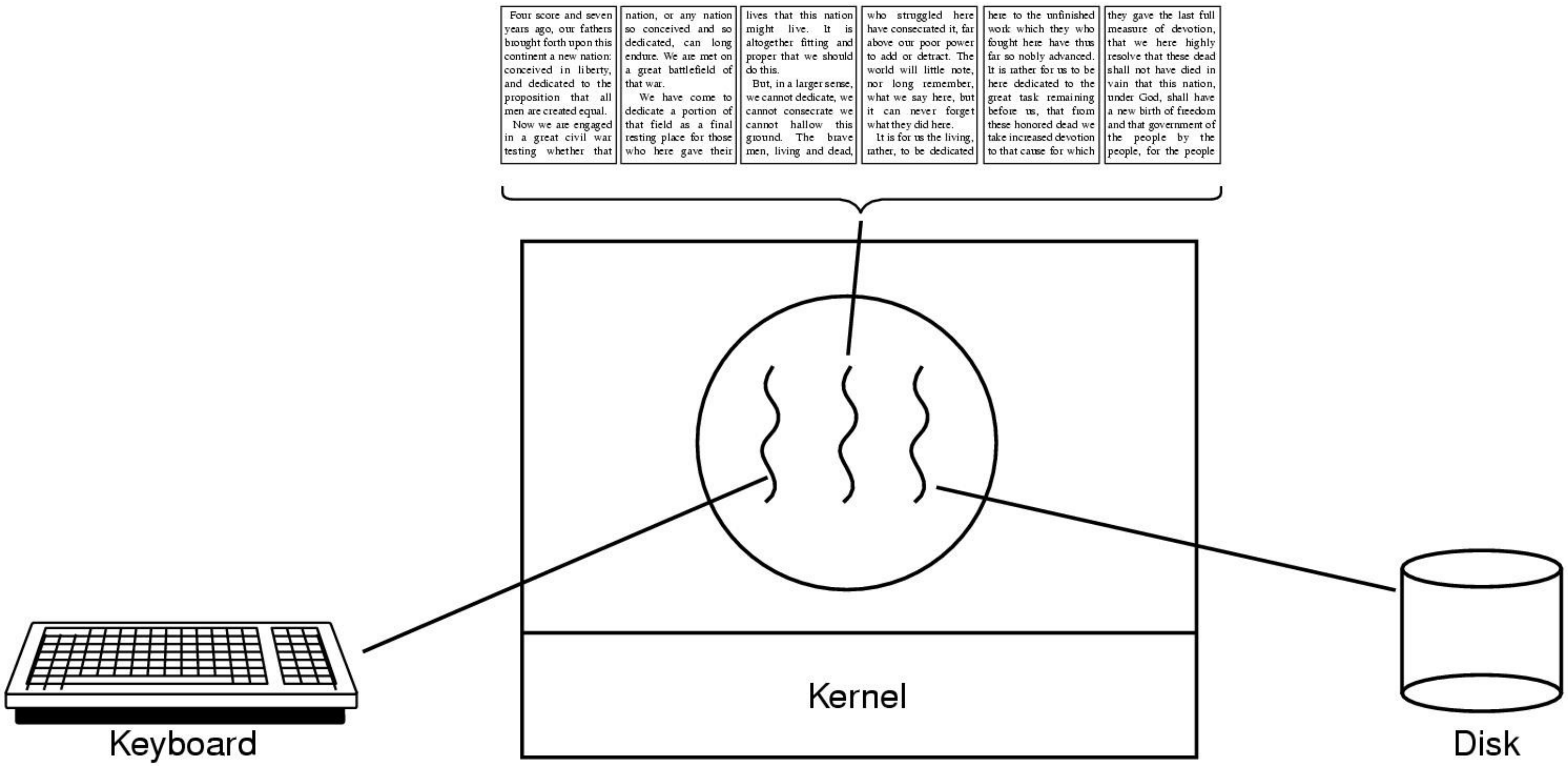
- (Left) Items shared by all threads in a process
- (Right) Items private to each thread

The Thread Model (3)



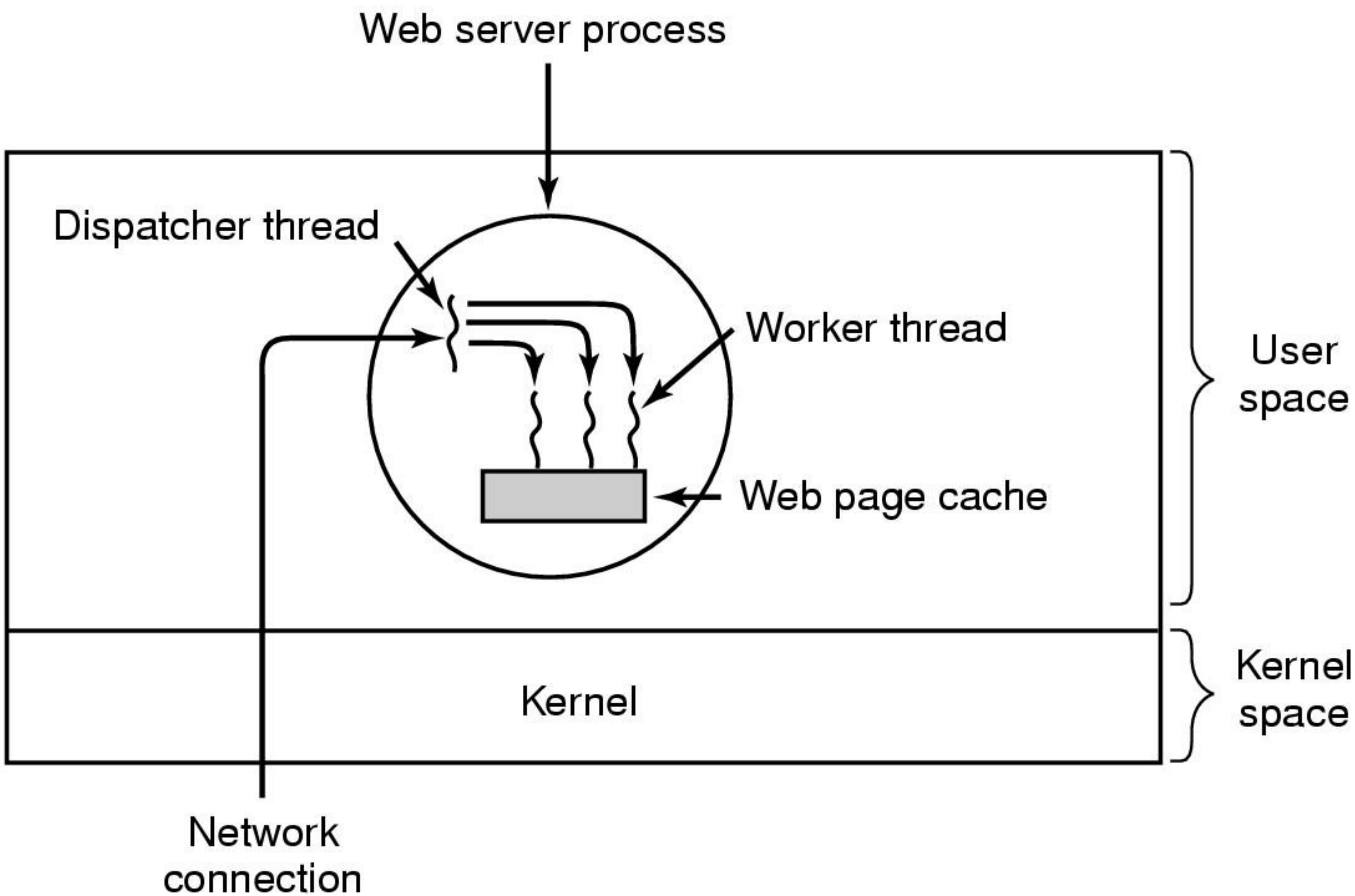
Each thread has its own stack

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);
}
```

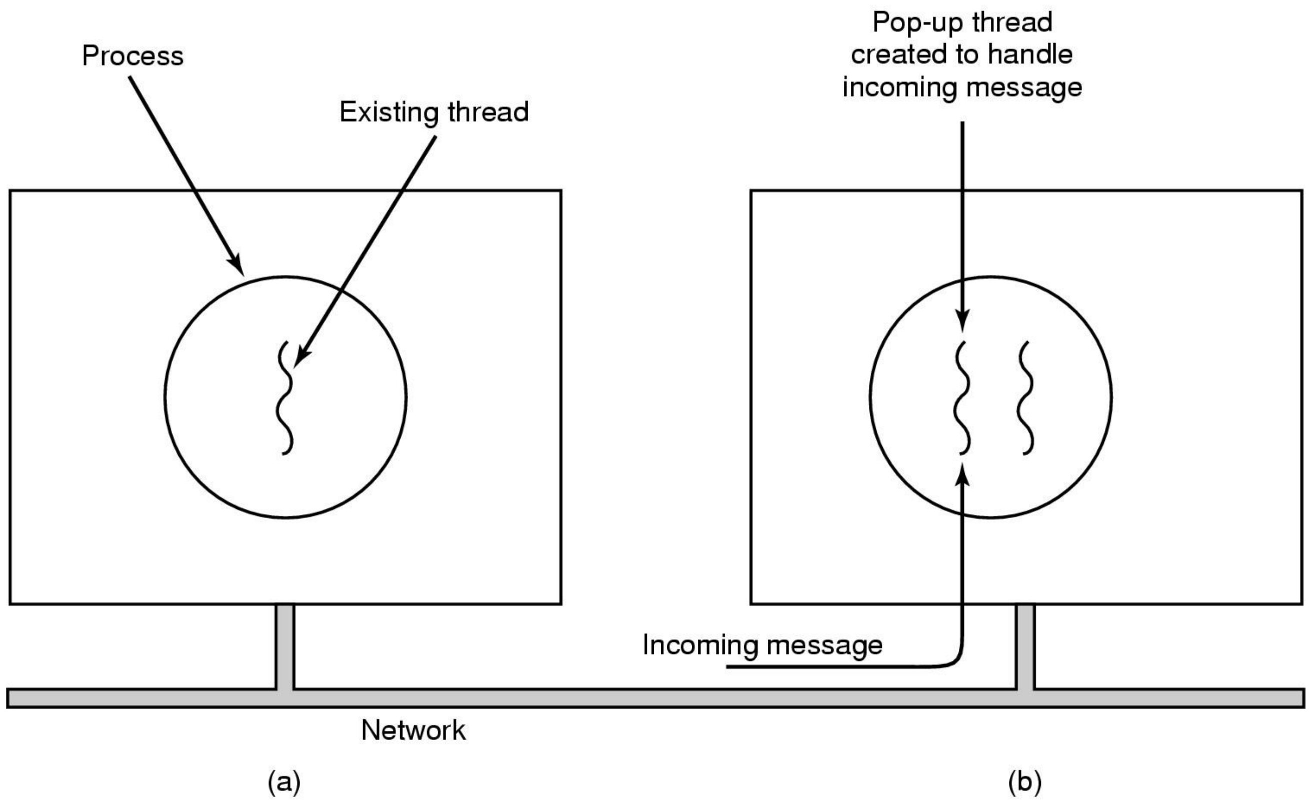
(b)

- Rough outline of code for previous slide

  (a) Dispatcher thread

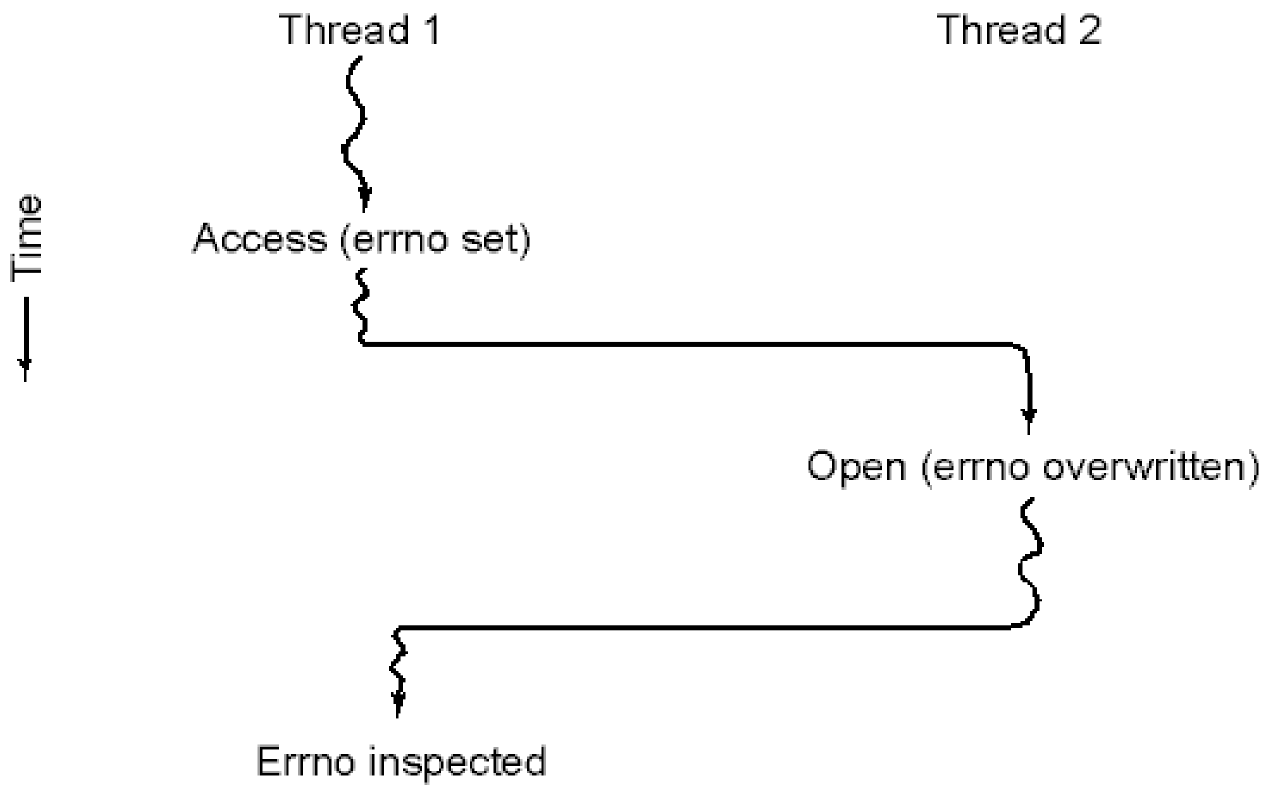
  (b) Worker thread

Pop-Up Threads



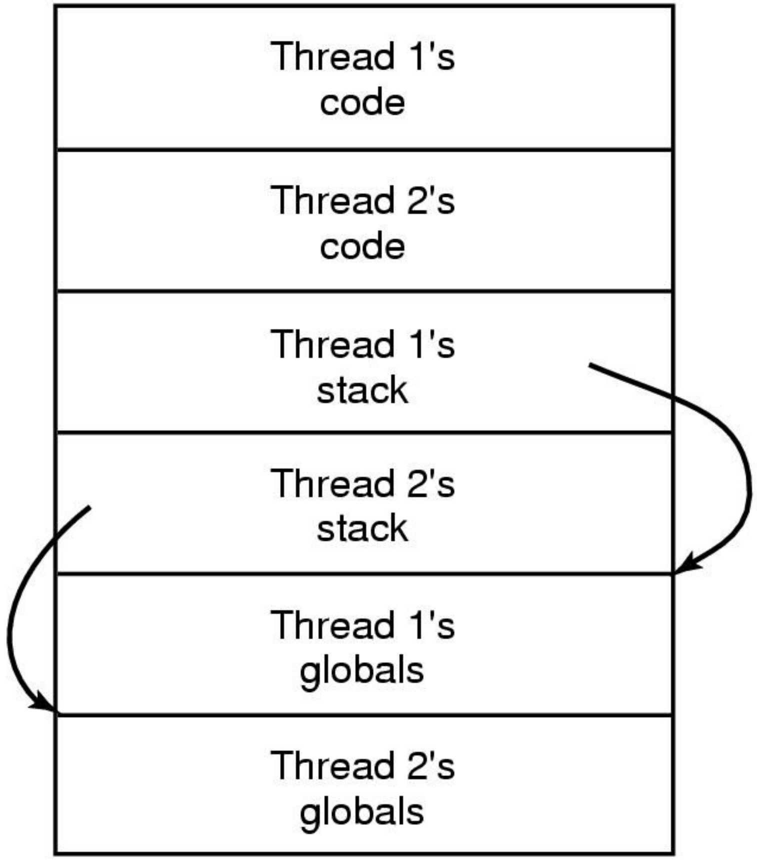
- Creation of a new thread when message arrives
  (a) before message arrives
  (b) after message arrives

Conflicts in Multithreaded Systems (1)



Conflicts between threads over the use of a global variable

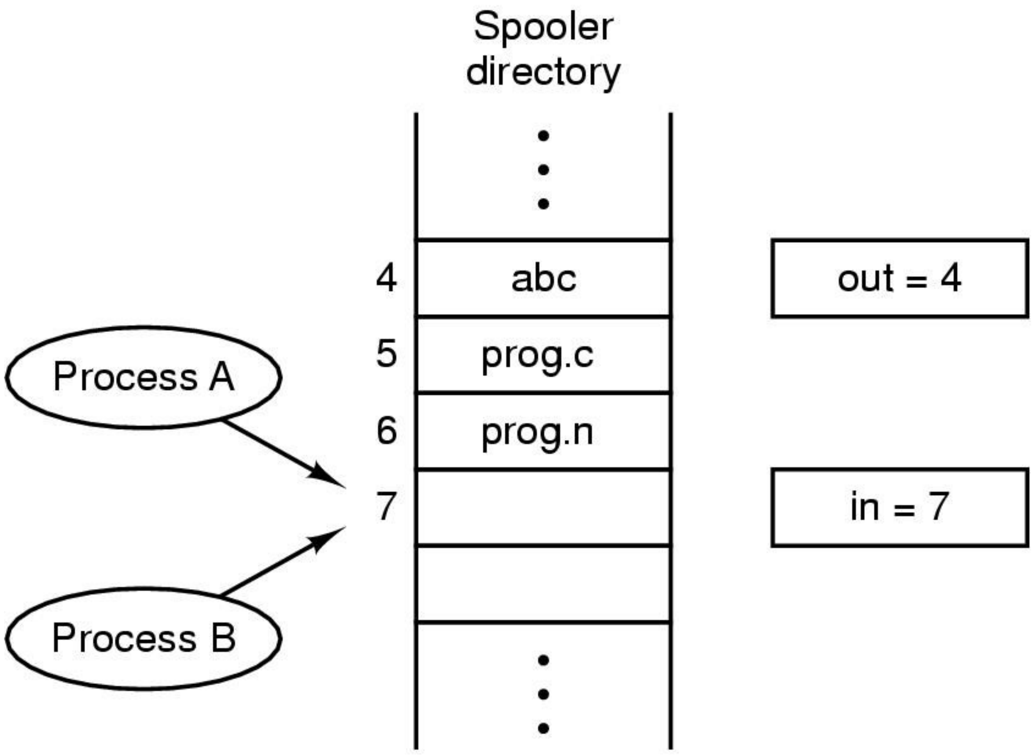
Conflicts in Multithreaded Systems (2)



Threads can have private global variables

Interprocess Communication

Race Conditions



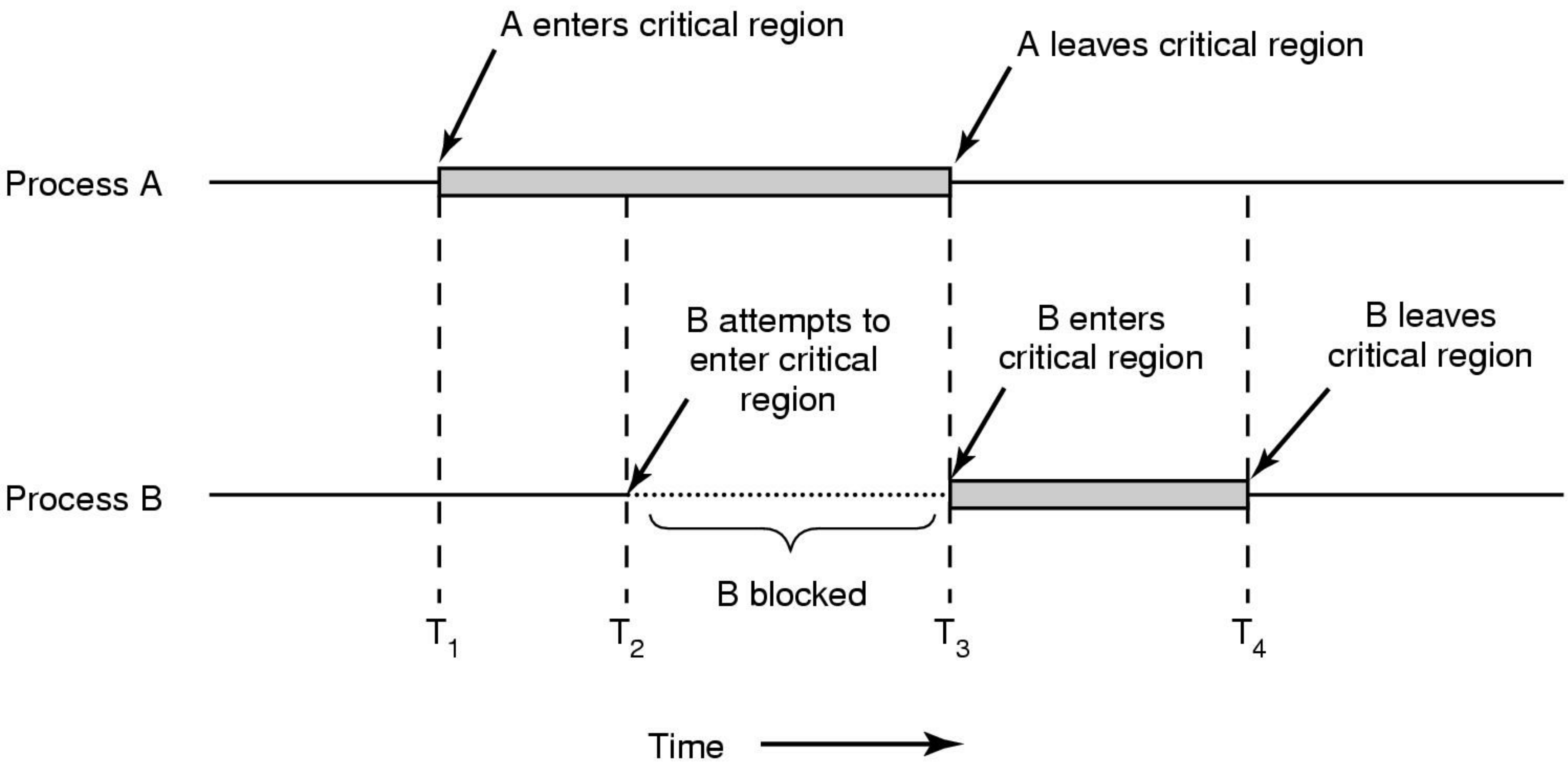
Two processes want to access shared memory at same time

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

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.

Mutual Exclusion with Busy Waiting (2)

```
#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 */
}
```


Peterson's solution for achieving mutual exclusion

Mutual Exclusion with Busy Waiting (3)

```
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
```

Entering and leaving a critical region using the TSL instruction

Sleep and Wakeup

```
#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 */
     }
}
```

Producer-consumer problem with fatal race condition

Semaphores

- Special type of variables in which:
  - Initialization possible only at declaration
  - The only possible operations are:
    - Increment by one using **up**(.) function
    - Decrement by one using **down**(.) function only if the current value is positive (>1)
    - If the semaphore's value is zero, the process is blocked at **down**(.) function call

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 */
     }
}
```


The producer-consumer problem using semaphores

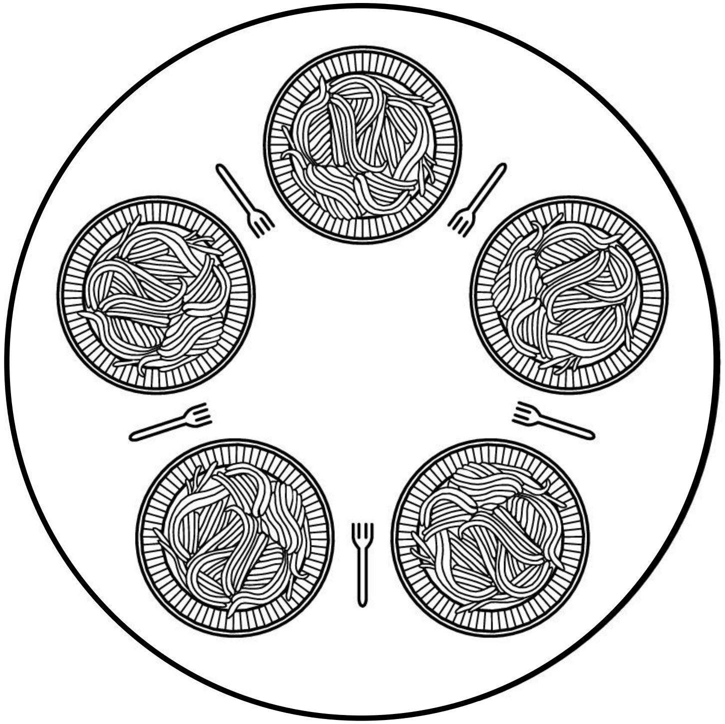
Message Passing

```
#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

Dining Philosophers (1)

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



Dining Philosophers (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 */
    }
}
```

A nonsolution to the dining philosophers problem

Dining Philosophers (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 */
    }
}
```

Solution to dining philosophers problem (part 1)

Dining Philosophers (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 */
}

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]);
     }
}
```


Solution to dining philosophers problem (part 2)

The Readers and Writers Problem

```
typedef int semaphore;             /* use your imagination */
semaphore mutex = 1;               /* controls access to 'rc' */
semaphore db = 1;                  /* controls access to the database */
int rc = 0;                        /* # of processes reading or wanting to */

void reader(void)
{
     while (TRUE) {                /* repeat forever */
          down(&mutex);            /* get exclusive access to 'rc' */
          rc = rc + 1;             /* one reader more now */
          if (rc == 1) down(&db);  /* if this is the first reader ... */
          up(&mutex);              /* release exclusive access to 'rc' */
          read_data_base();        /* access the data */
          down(&mutex);            /* get exclusive access to 'rc' */
          rc = rc − 1;             /* one reader fewer now */
          if (rc == 0) up(&db);    /* if this is the last reader ... */
          up(&mutex);              /* release exclusive access to 'rc' */
          use_data_read();         /* noncritical region */
     }
}


void writer(void)
{
     while (TRUE) {                /* repeat forever */
          think_up_data();         /* noncritical region */
          down(&db);               /* get exclusive access */
          write_data_base();       /* update the data */
          up(&db);                 /* release exclusive access */
     }
}
```

A solution to the readers and writers problem

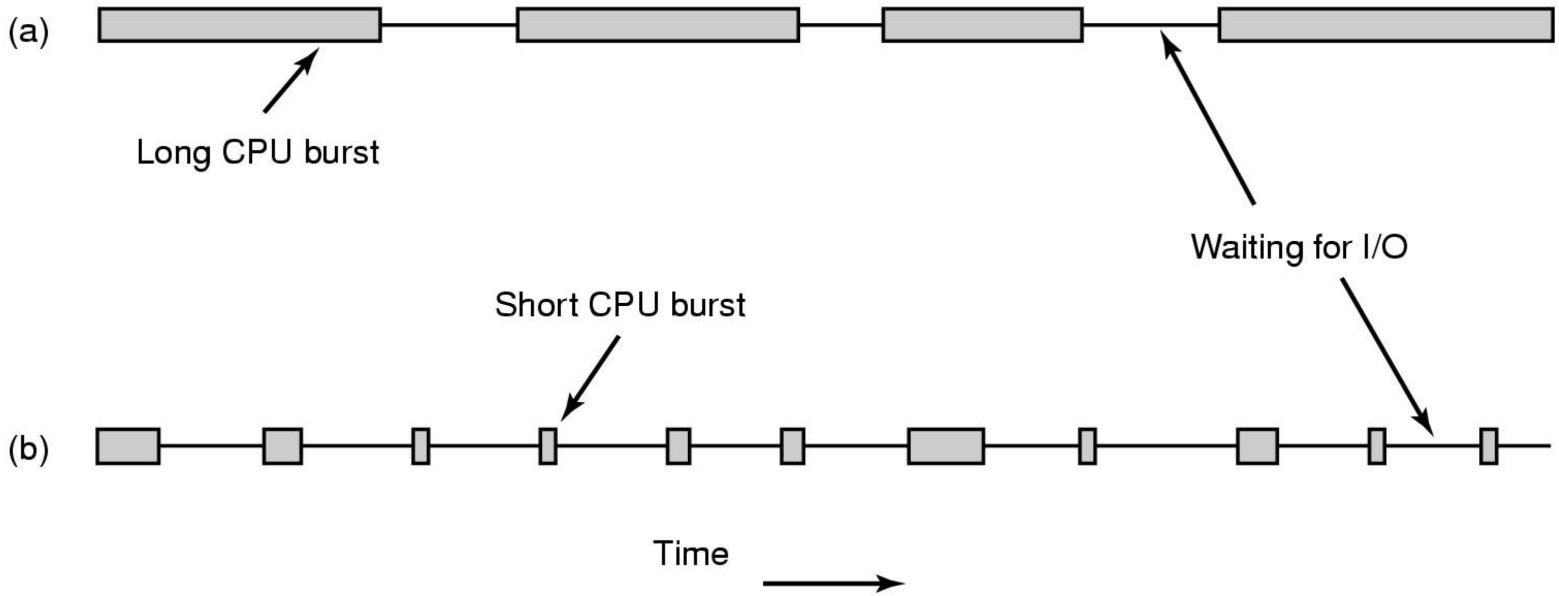
The Sleeping Barber Problem (1)

The Sleeping Barber Problem (2)

```
#define CHAIRS 5                    /* # chairs for waiting customers */

typedef int semaphore;              /* use your imagination */

semaphore customers = 0;            /* # of customers waiting for service */
semaphore barbers = 0;              /* # of barbers waiting for customers */
semaphore mutex = 1;                /* for mutual exclusion */
int waiting = 0;                    /* customers are waiting (not being cut) */

void barber(void)
{
    while (TRUE) {
        down(&customers);           /* go to sleep if # of customers is 0 */
        down(&mutex);               /* acquire access to 'waiting' */
        waiting = waiting – 1;      /* decrement count of waiting customers */
        up(&barbers);               /* one barber is now ready to cut hair */
        up(&mutex);                 /* release 'waiting' */
        cut_hair( );                /* cut hair (outside critical region) */
    }
}


void customer(void)
{
    down(&mutex);                   /* enter critical region */
    if (waiting < CHAIRS) {         /* if there are no free chairs, leave */
        waiting = waiting + 1;      /* increment count of waiting customers */
        up(&customers);             /* wake up barber if necessary */
        up(&mutex);                 /* release access to 'waiting' */
        down(&barbers);             /* go to sleep if # of free barbers is 0 */
        get_haircut( );             /* be seated and be serviced */
    } else {
        up(&mutex);                 /* shop is full; do not wait */
    }
}
```

Solution to sleeping barber problem.

Scheduling (1)



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

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

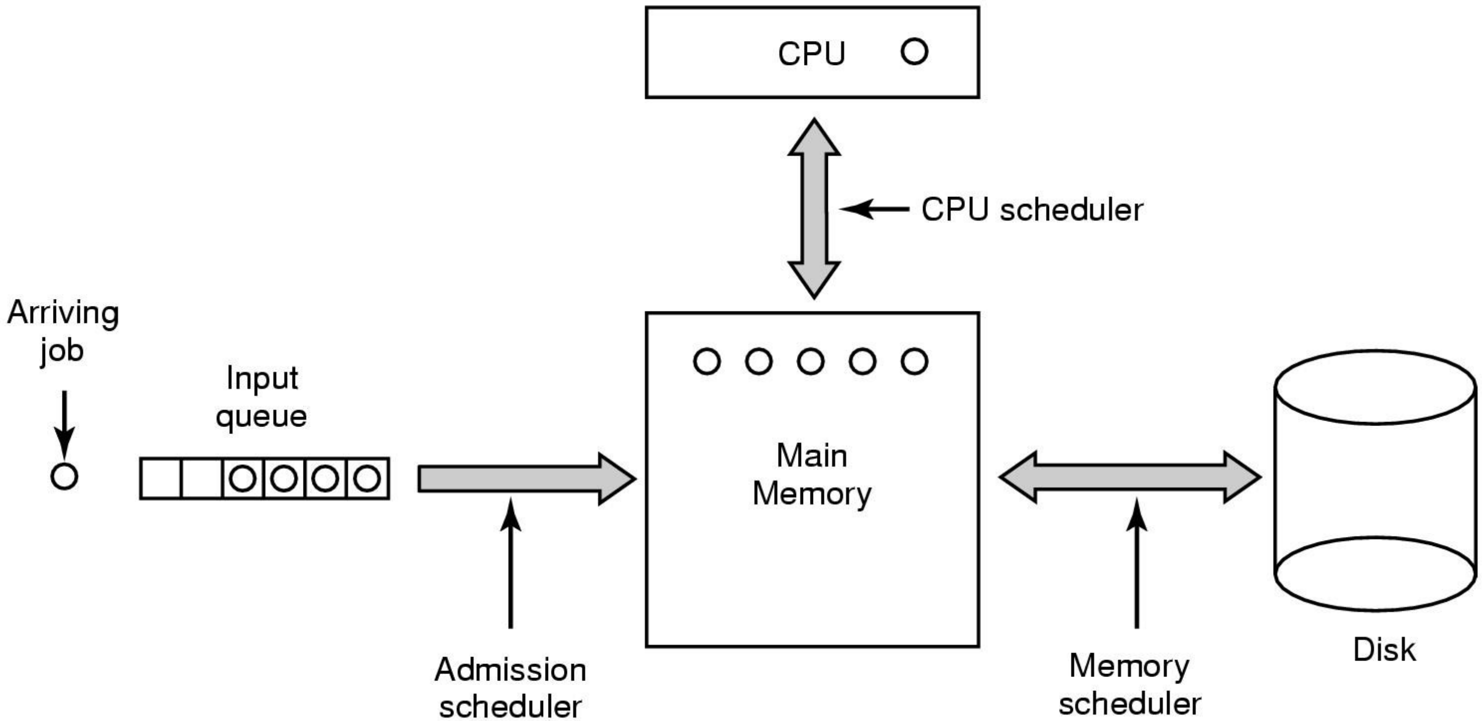
Meeting deadlines - avoid losing data
Predictability - avoid quality degradation in multimedia systems

Scheduling Algorithm Goals

Scheduling in Batch Systems (1)

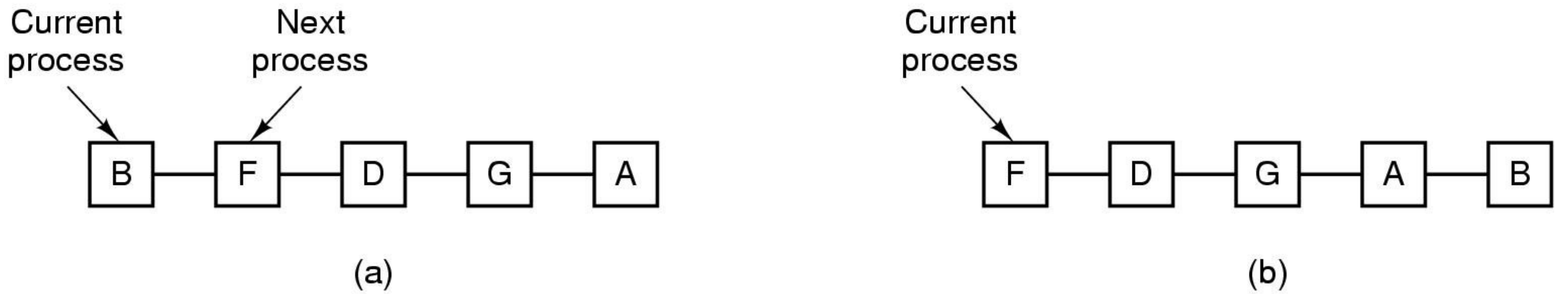
An example of shortest job first scheduling

Scheduling in Batch Systems (2)



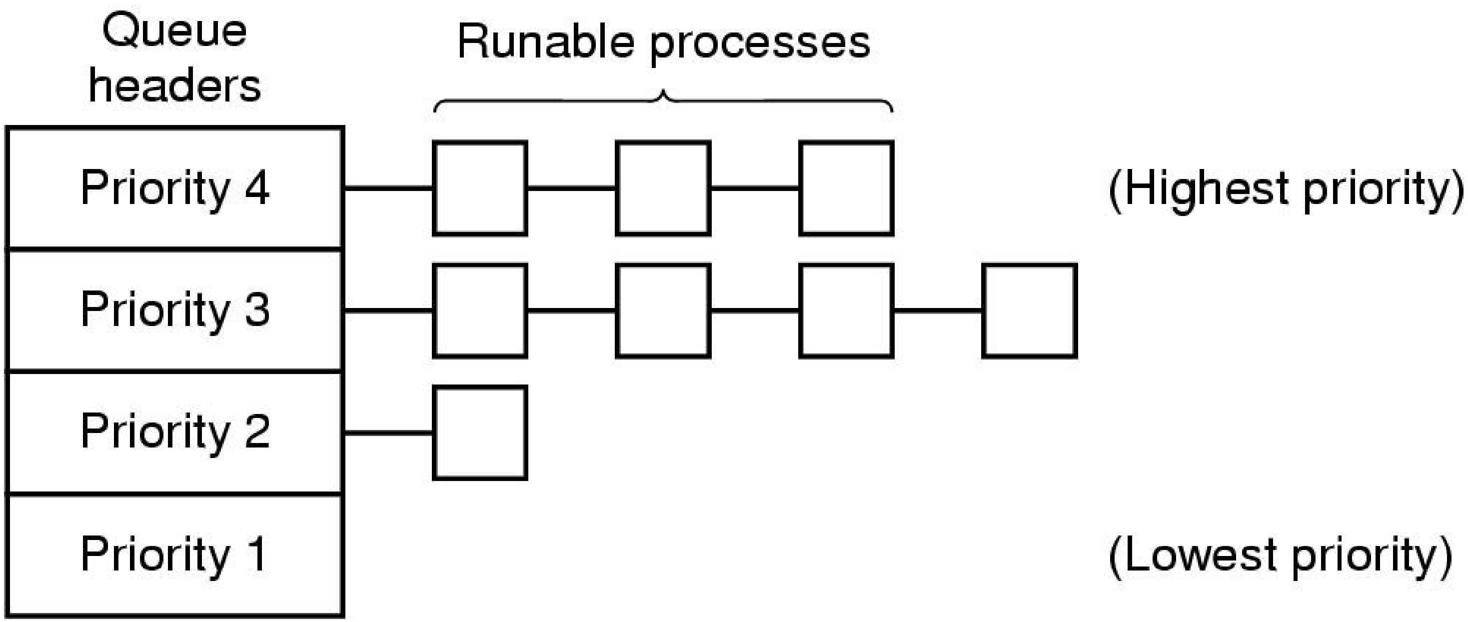
Three level scheduling

Scheduling in Interactive Systems (1)



- Round Robin Scheduling
  - list of ready processes
  - list of ready processes after B uses up its time slice

Scheduling in Interactive Systems (2)



A scheduling algorithm with four priority classes

Scheduling in Real-Time Systems

Schedulable real-time system

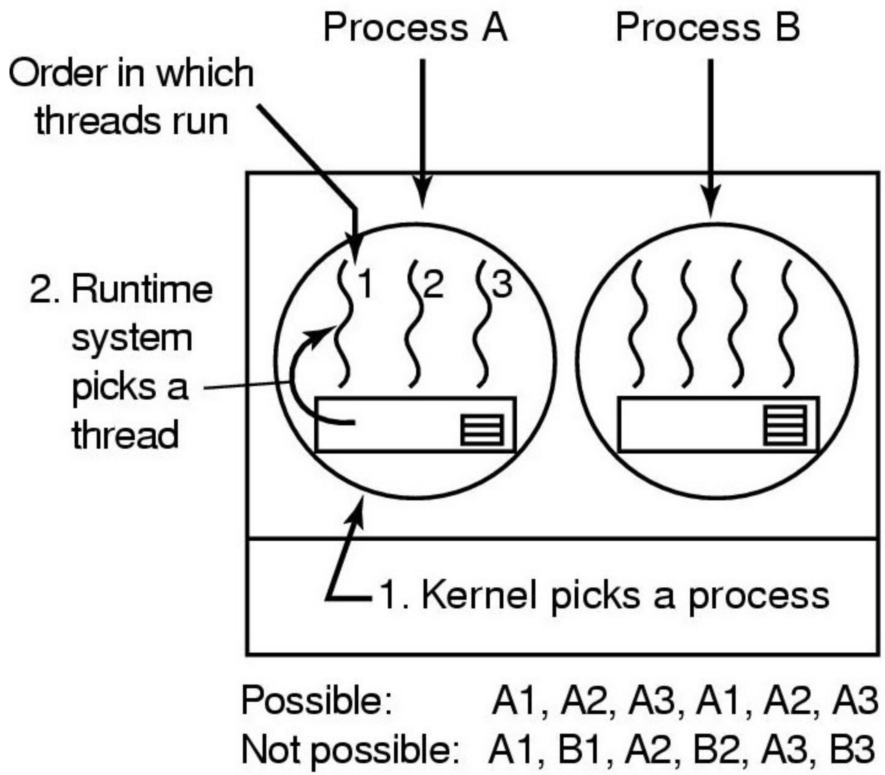
- Given
  - *m* periodic events
  - event *i* occurs within period P_i and requires C_i seconds
- Then the load can only be handled if

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

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

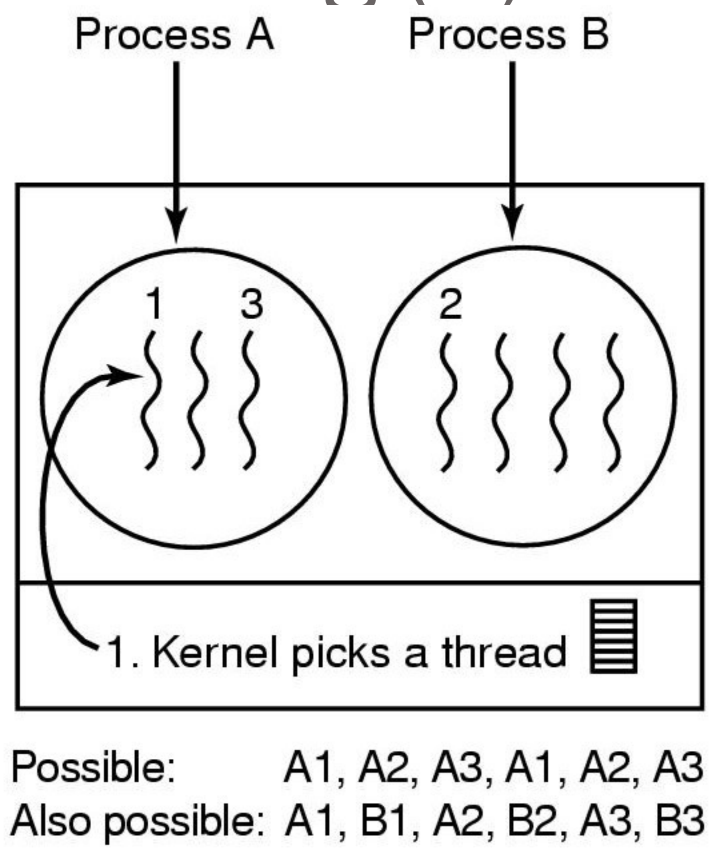
Thread Scheduling (1)



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

Questions?