# Threads

CSCE 351: Operating System Kernels

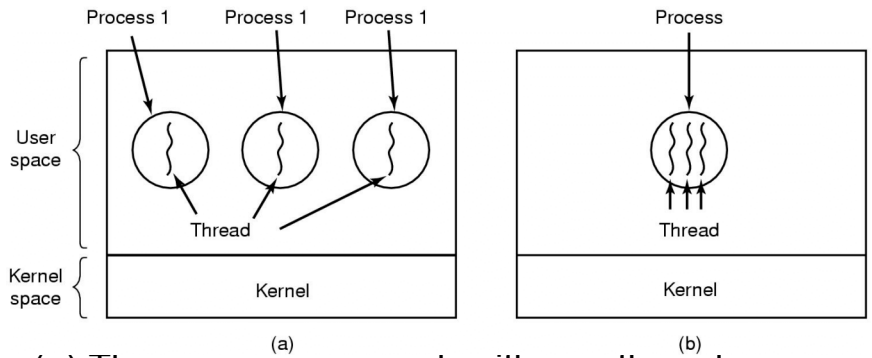
Witawas Srisa-an

Chapter 4-5

1

# Threads

## The Thread Model (1)




(a) Three processes each with one thread

(b) One process with three threads

2

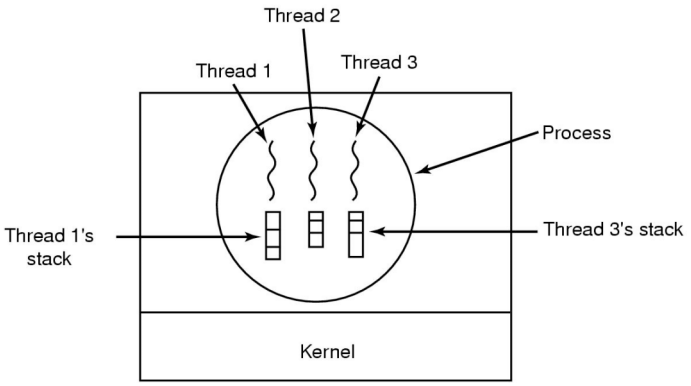
## The Thread Model (2)

| 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 | |

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

3

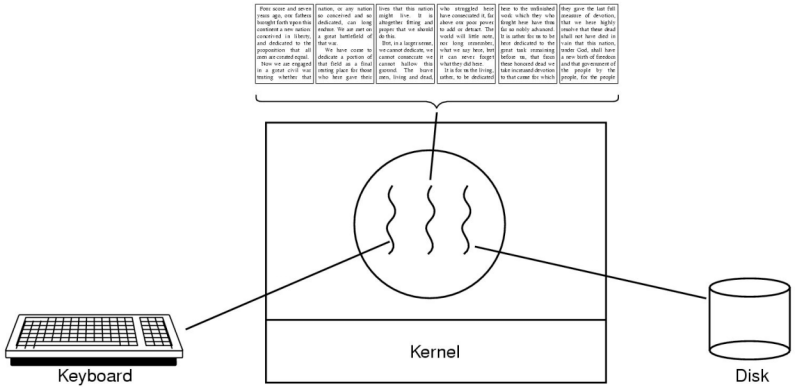
## The Thread Model (3)




Each thread has its own stack

4

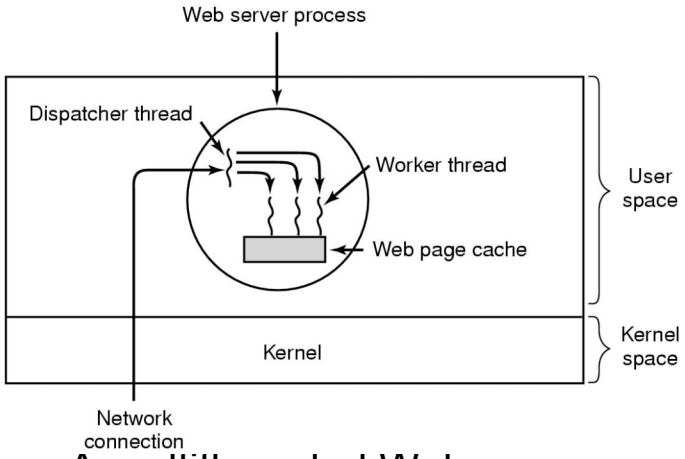
# Thread Usage (1)




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

7

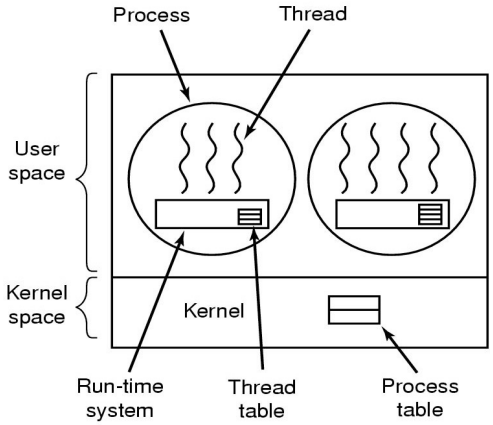
## Thread Usage (4)

| 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

8

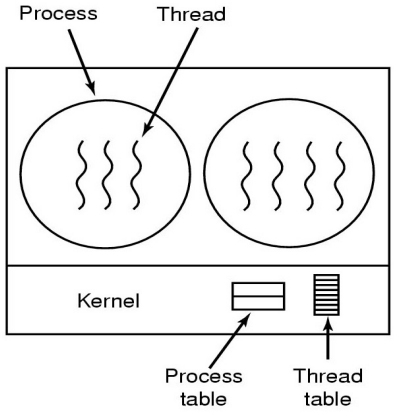
# Implementing Threads in User Space




A user-level threads package

9

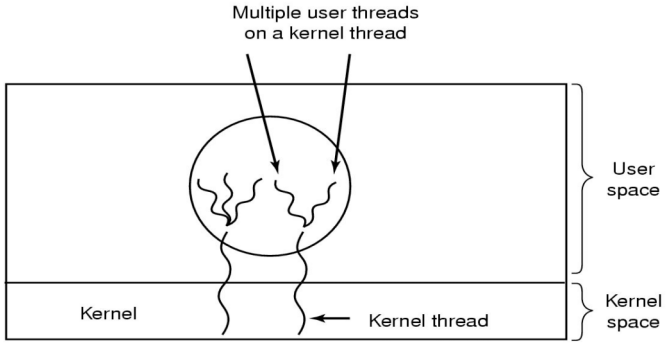
# Implementing Threads in the Kernel




A threads package managed by the kernel

10

## Hybrid Implementations



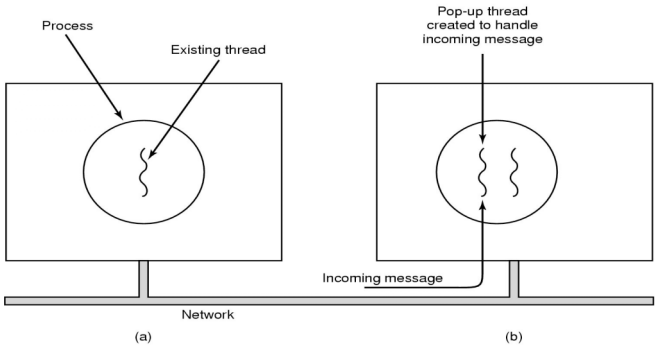

Multiplexing user-level threads onto kernel-level threads



## 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)

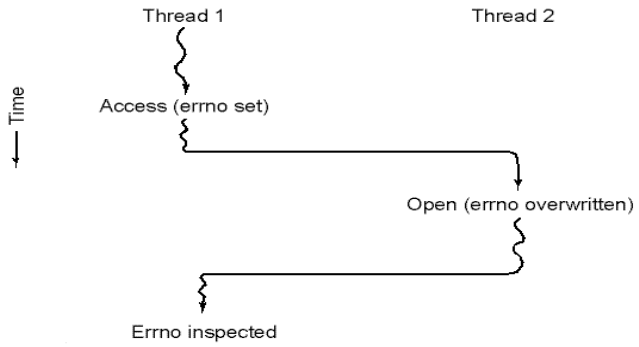
# Pop-Up Threads




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

13

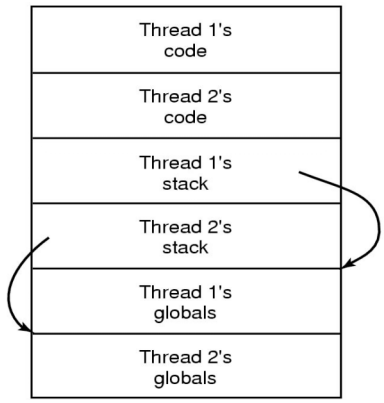
# Making Single-Threaded Code Multithreaded (1)




Conflicts between threads over the use of a global variable

14

## Making Single-Threaded Code Multithreaded (2)

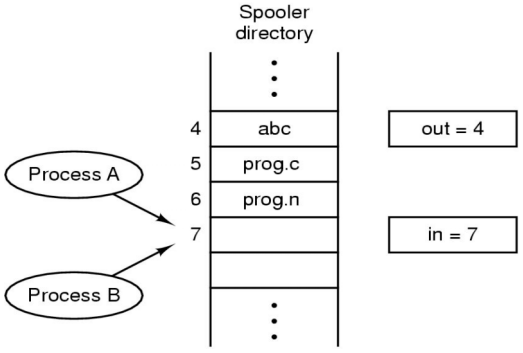



Threads can have private global variables

15

## Interprocess Communication

### Race Conditions




Two processes want to access shared memory at same time

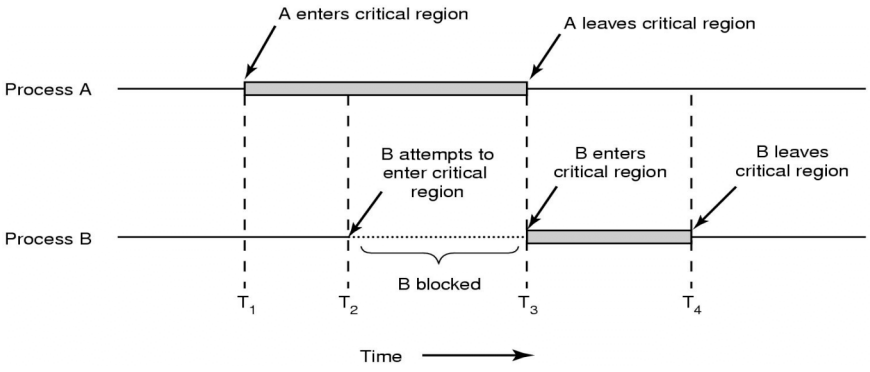
16

## 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

17

## Critical Regions (2)




Mutual exclusion using critical regions

18

## 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.

No process running outside its critical region may block another process

19

## Mutual Exclusion with Busy Waiting (2)

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 */
}
```

20

## 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

21

## 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

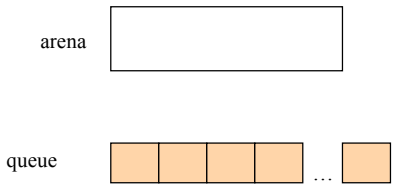
22

## Semaphores

- A variable type
  - 0 or any positive values (counting)
  - 0 or 1 (binary)
- Support two operations
  - down (p)
    - if value > 0 then decrement
    - if value = 0 then suspend process without completing the down
    - indivisible atomic action
  - up (v)
    - increment the semaphore value and if there are processes sleeping on the semaphore, wake one of them up

23

## Semaphore




24

# 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

25

# 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*

26

## Example Program

- suspend.c and wake.c

27

## Monitors

- Language construct
  - higher level synchronization primitive
- Only one process can be active in a monitor at any instant
- Use condition variables to block processes
  - wait operation
  - signal operation

28

# 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;
```

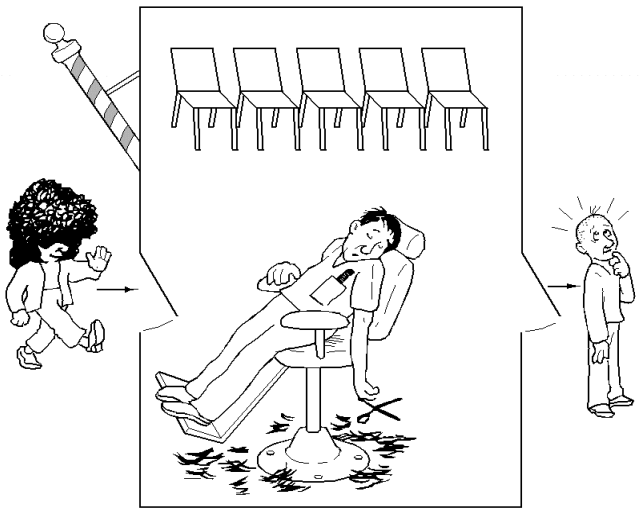
- Outline of producer-consumer problem with monitors
  - only one process can be active in a monitor at one time
  - buffer has *N* slots

# 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 31

# The Sleeping Barber Problem (1)



32

# The Sleeping Barber Problem (2)

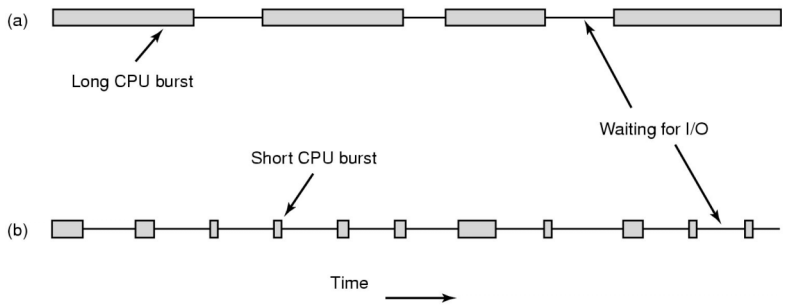
```
#define CHAIRS 5                /* # chairs for waiting customers */

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.

33

# 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

34

# 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

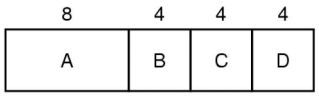
**Interactive systems**
- Response time - respond to requests quickly
- Proportionality - meet users' expectations

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

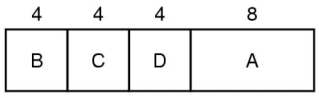
Scheduling Algorithm Goals

35

# Scheduling in Batch Systems (1)




(a)




(b)

(8 + 12 + 16 + 20) / 4

= 14 units

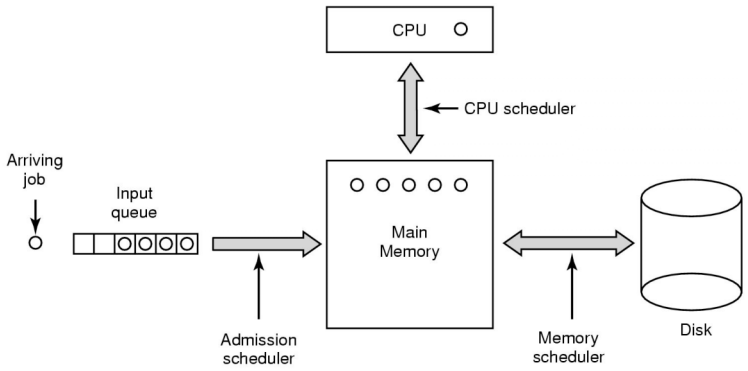
(4 + 8 + 12 + 20) / 4

= 11 units

An example of shortest job first scheduling

36

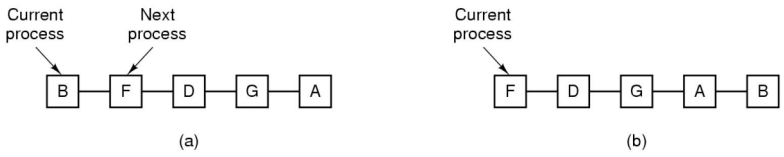
## Scheduling in Batch Systems (2)




Three level scheduling

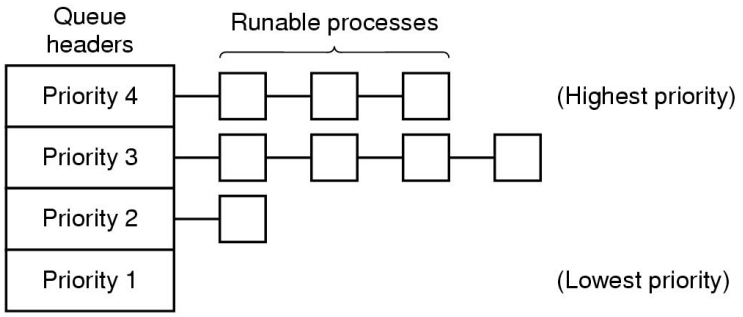


## Scheduling in Interactive Systems (1)




- 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

39

## Scheduling in Real-Time Systems (1)

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

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

40

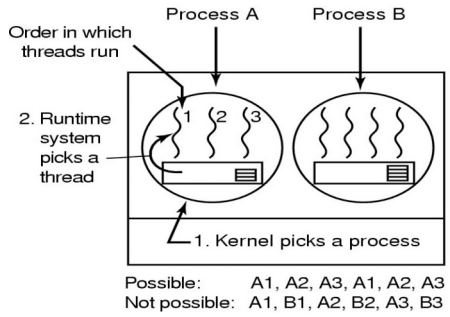
## Scheduling in Real-Time Systems (2)

- Example
  - four periodic events: 100, 200, 400, 600 ms
  - required CPU times: 25, 40, 100, 120 ms

  Is this system schedulable?

41

## Thread Scheduling (1)

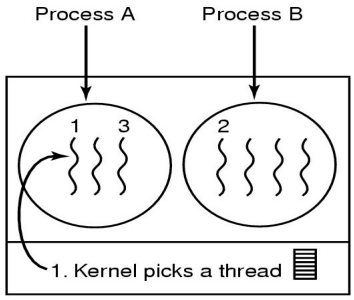



Possible scheduling of user-level threads

- 50-msec process quantum
- threads run 5 msec/CPU burst

42

# Thread Scheduling (2)




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

Possible scheduling of kernel-level threads

- 50-msec process quantum
- threads run 5 msec/CPU burst

43