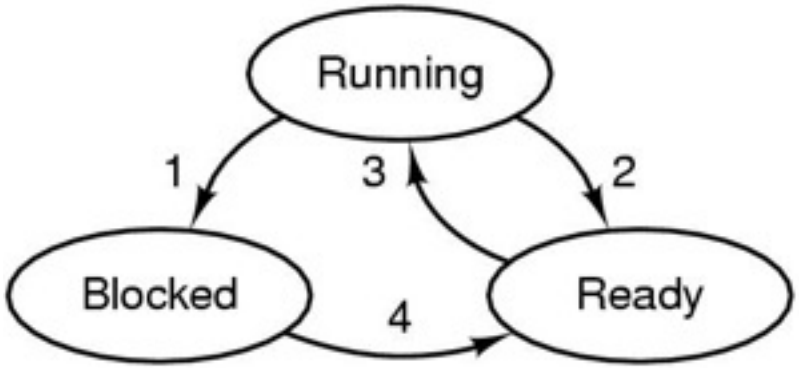
Chapter 2

Scheduling

Outline

- CPU Scheduling
  - Importance of scheduling in diff. environment
  - CPU bound process, I/O bound process
  - Preemptive, non-preemptive scheduling
- Batch system scheduling algorithms
  - FCFS, Shortest Job First, Shortest Remaining First
- Interactive system scheduling
  - RR, Priority scheduling, Lottery Scheduling
- Realtime system scheduling
- Thread Scheduling
  - User-level thread
  - Kernel-level thread

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.

Tanenbaum, Modern Operating Systems 3 e, (c) 2008 Prentice-Hall, Inc. All rights reserved. 0-13-**6006639**

Modeling Multiprogramming

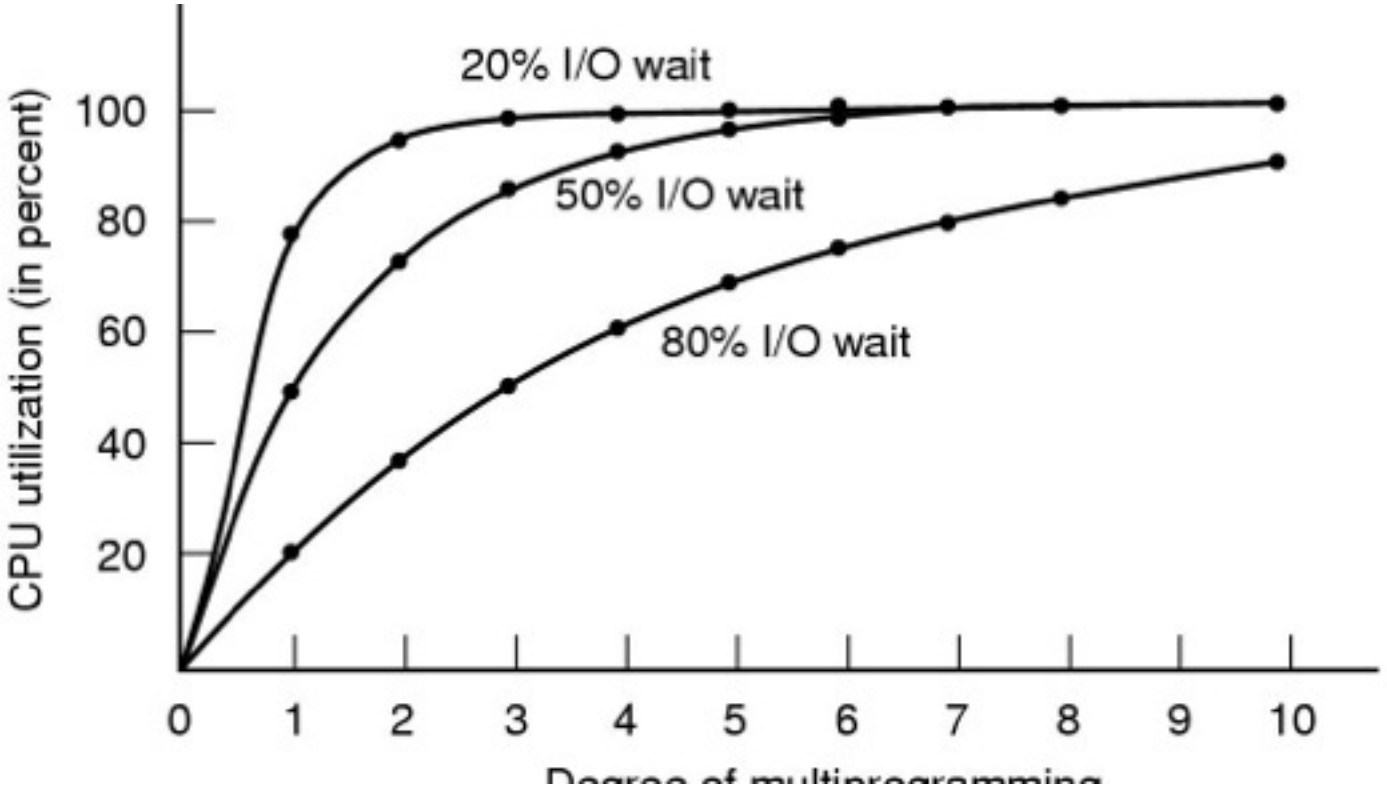


Figure 2-6. CPU utilization as a function of the number of processes in memory.

Tanenbaum, Modern Operating Systems 3 e, (c) 2008 Prentice-Hall, Inc. All rights reserved. 0-13-**6006639**

Process Control Table

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

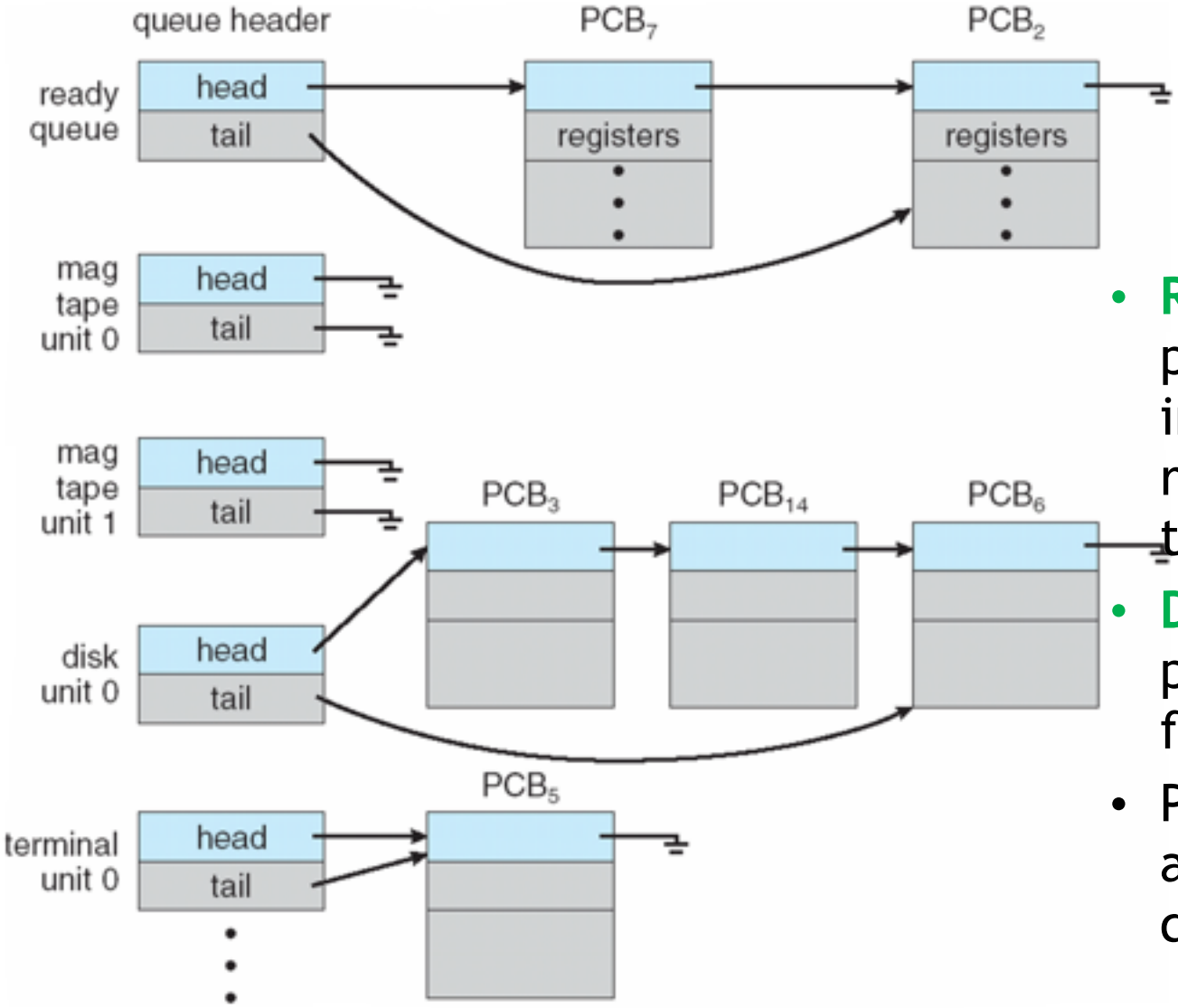
Figure 2-4. Some of the fields of a typical process table entry.

Tanenbaum, Modern Operating Systems 3 e, (c) 2008 Prentice-Hall, Inc. All rights reserved. 0-13-**6006639**

Process Scheduling Queues

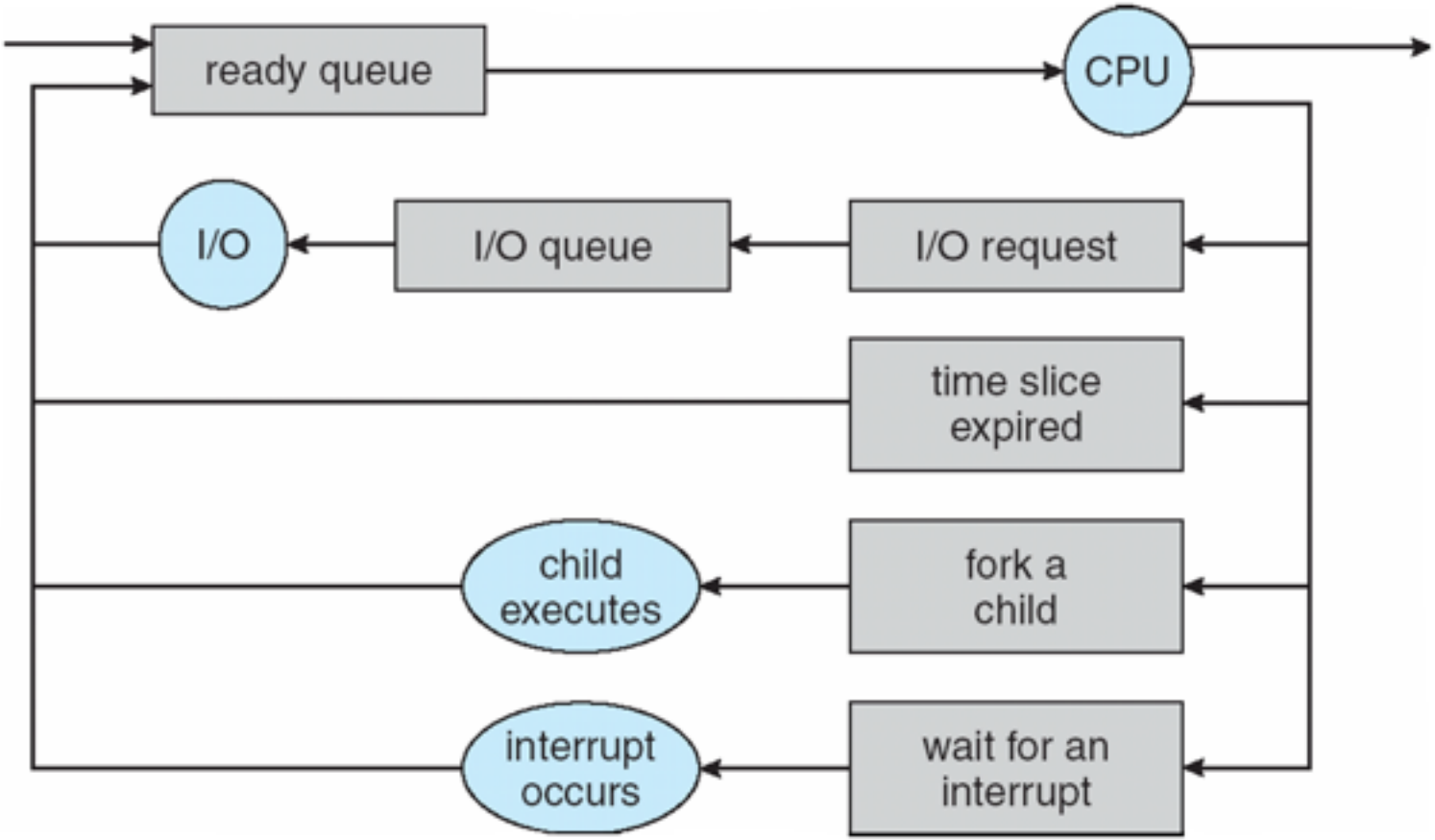
- **Job queue** – set of all processes in the system
- **Ready queue** – set of all processes residing in main memory, ready and waiting to execute
- **Device queues** – set of processes waiting for an I/O device
- Processes migrate among the various queues

Ready Queue And Various I/O Device Queues



- **Ready queue** – processes residing in main memory, ready and waiting to execute
- **Device queues** – processes waiting for an I/O device
- Processes migrate among the various queues

Representation of Process Scheduling



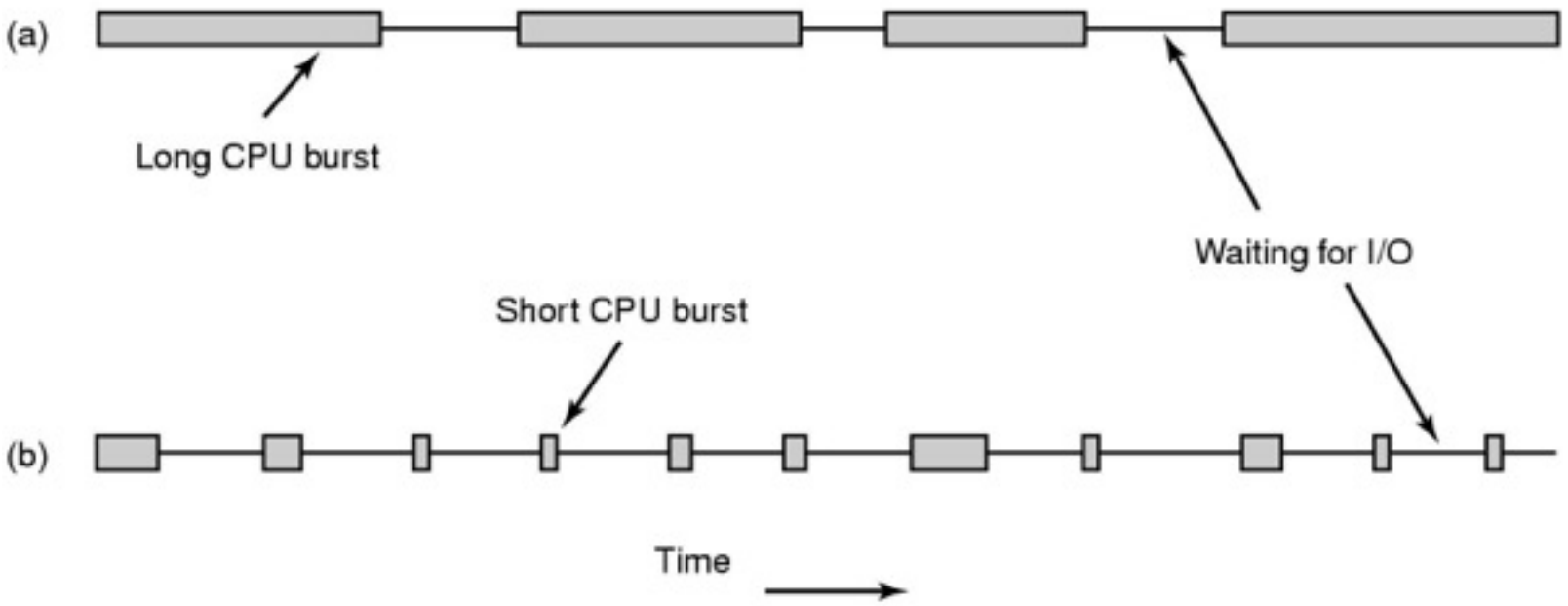
Schedulers

- When to schedule?
  - when a process is created, or exits
  - when a process blocks on I/O, semaphore, mutex…
  - interrupts: hardware or software
  - timer interrupt: nonpreemptive vs preemptive scheduling
- **Long-term scheduler** (or job scheduler) – selects which processes should be brought into the ready queue
- **Short-term scheduler** (or CPU scheduler) – selects which process should be executed next and allocates CPU

Schedulers

- Short-term scheduler is invoked very frequently (milliseconds) ⇒ (must be fast)
- Long-term scheduler is invoked very infrequently (seconds, minutes) ⇒ (may be slow)
- The long-term scheduler controls the *degree of multiprogramming*
- Processes can be described as either:
  - **I/O-bound process** – spends more time doing I/O than computations, many short CPU bursts
  - **CPU-bound process** – spends more time doing computations; few very long CPU bursts

Process Behavior



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

Tanenbaum, Modern Operating Systems 3 e, (c) 2008 Prentice-Hall, Inc. All rights reserved. 0-13-**6006639**

Categories of Scheduling Algorithms

- Batch system: business application, no end users
  - non-preemptive, preemptive
- Interactive system: with interactive users, or server (with multiple remote interactive users)
  - preemption
- Real time system:

Tanenbaum, Modern Operating Systems 3 e, (c) 2008 Prentice-Hall, Inc. All rights reserved. 0-13-**6006639**

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

Tanenbaum, Modern Operating Systems 3 e, (c) 2008 Prentice-Hall, Inc. All rights reserved. 0-13-**6006639**

Scheduling in Batch Systems

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

Tanenbaum, Modern Operating Systems 3 e, (c) 2008 Prentice-Hall, Inc. All rights reserved. 0-13-**6006639**

First Come First Served

- First-come first-served
  - processes assigned CPU in order of request
  - when running process blocks, schedule next one in queue
  - when blocking process becomes ready, enter end of queue
- Pros: simple, easy to implement, fair (in some sense)
- Cons:
  - short jobs arrive after a very long job
  - one compute-bound process (1 sec at a time), many IO-bound processes (perform 1000 disk reads) => take a long time to finish I/O bound process

Tanenbaum, Modern Operating Systems 3 e, (c) 2008 Prentice-Hall, Inc. All rights reserved. 0-13-**6006639**

Shortest Job First

- Assumption: run time for processes are known in advance
- Scheduler: among equally important jobs in ready queue, pick the one with the shortest run time.
- Proof: Shortest Job First yields smallest average turnaround time, if all jobs are available simultaneously.

Tanenbaum, Modern Operating Systems 3 e, (c) 2008 Prentice-Hall, Inc. All rights reserved. 0-13-**6006639**

Shortest Job First

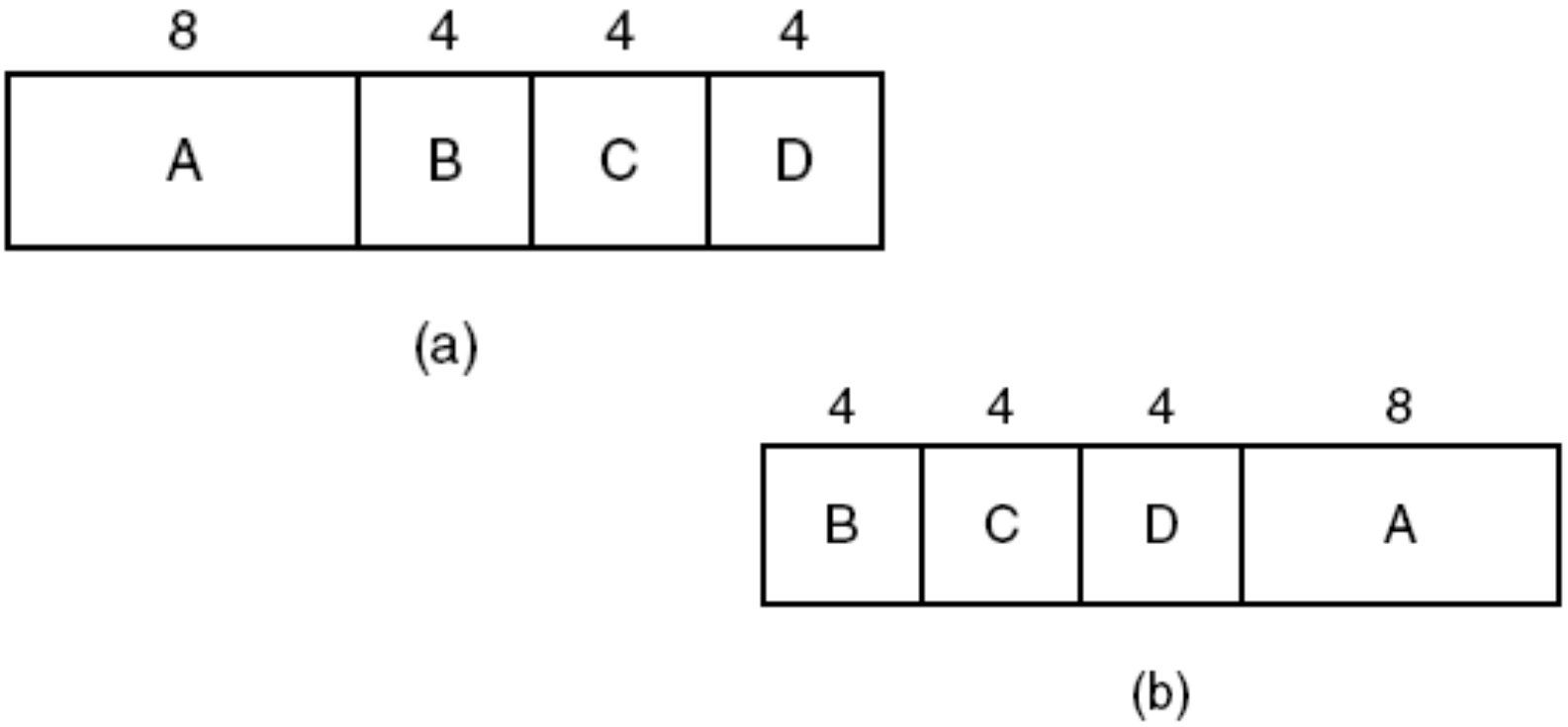


Figure 2-40. An example of shortest job first scheduling. (a) Running four jobs in the original order. (b) Running them in shortest job first order.

Tanenbaum, Modern Operating Systems 3 e, (c) 2008 Prentice-Hall, Inc. All rights reserved. 0-13-**6006639**

Shortest Remaining Time next

- Jobs/Processes can arrive at different time
- Preemptive version of Shortest Job First
  - When new job arrives, if its run time is smaller than current process's remaining time, schedule the new job
- Some kind of greedy algorithm: keep the ready queue as short as possible
- Question: Does this scheme minimize average turnaround time?

Tanenbaum, Modern Operating Systems 3 e, (c) 2008 Prentice-Hall, Inc. All rights reserved. 0-13-**6006639**

Scheduling in Interactive Systems

- Round-robin scheduling
- Priority scheduling
- Multiple queues
- Shortest process next
- Guaranteed scheduling
- Lottery scheduling
- Fair-share scheduling

Tanenbaum, Modern Operating Systems 3 e, (c) 2008 Prentice-Hall, Inc. All rights reserved. 0-13-**6006639**

Round-Robin Scheduling

- A process, when scheduled to run, is assigned the CPU for a time interval, quantum
  - If process blocks or finishes before quantum expires, CPU switches to run other process
  - If still running at end of quantum, preempt and schedule other process to run
- Length of quantum
  - too short => too much context switch overhead
  - too long => system not responsive/interactive
  - typical setting: 20-50 msec

Tanenbaum, Modern Operating Systems 3 e, (c) 2008 Prentice-Hall, Inc. All rights reserved. 0-13-**6006639**

Round-Robin Scheduling

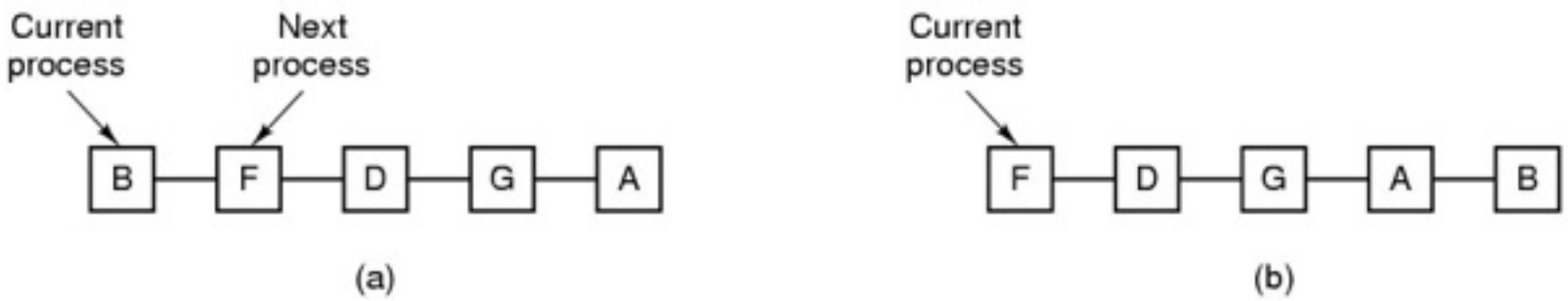


Figure 2-41. Round-robin scheduling.
(a) The list of runnable processes. (b) The list of runnable processes after B uses up its quantum.

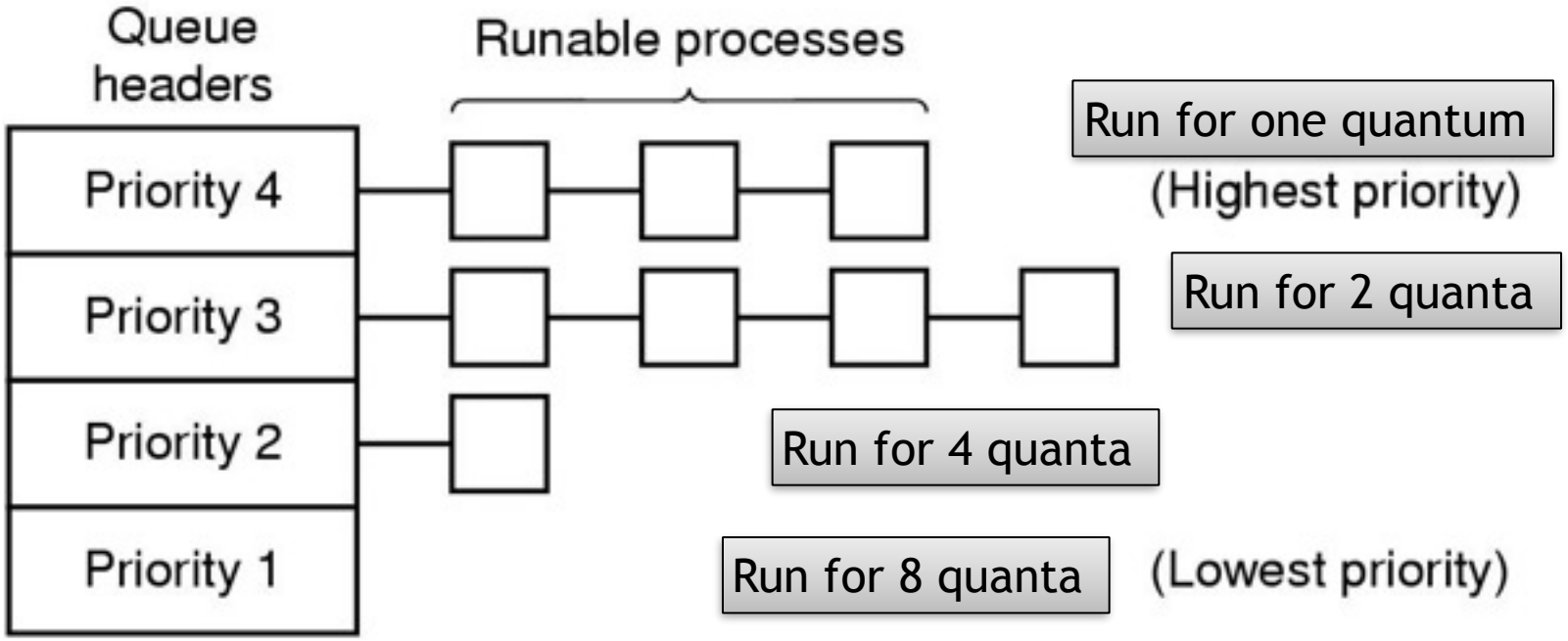
Tanenbaum, Modern Operating Systems 3 e, (c) 2008 Prentice-Hall, Inc. All rights reserved. 0-13-**6006639**

Priority Scheduling

- Idea: assign each process a priority, ready processes with highest priority is scheduled to run
- Setting priority
  - based upon the process's user ID: position, payment
  - higher priority for interactive process, lower priority for background process => to be responsive
  - dynamically assigned
    - e.g., give higher priority to I/O bound process

Tanenbaum, Modern Operating Systems 3 e, (c) 2008 Prentice-Hall, Inc. All rights reserved. 0-13-**6006639**

Priority Scheduling: data structure



Example (CTSS)

- CPU bound process will sink to long priority queue
  - If used up quantum, move down one class
  - larger quantum => cut context switch overhead
- I/O bound process will stay at high priority queue

Tanenbaum, Modern Operating Systems 3 e, (c) 2008 Prentice-Hall, Inc. All rights reserved. 0-13-6006639

Interactive Systems: shortest process next

- Interactive process:
  1. wait for command
  2. execute command
  3. go back to 1
- To minimize response time (step 2 above), schedule process with shortest running time
- Estimate running time of a process's step 2 using history (weighted average, aging)
  - Te' = aTe + (1-a) Ti
  - Te: current estimation
  - Te': new estimation
  - Ti: current measured running time

Tanenbaum, Modern Operating Systems 3 e, (c) 2008 Prentice-Hall, Inc. All rights reserved. 0-13-**6006639**

Interactive Systems: Lottery Scheduling

- A randomized scheme
- each process given lottery tickets for CPU resource
- Scheduler: choose a lottery ticket at random, the process holds the ticket is the scheduled to run
  - The more tickets a process holds, the higher probability of scheduled to run
- Pros:
  - proportional allocation of CPU
  - allow transferring of tickets among cooperating processes,

Tanenbaum, Modern Operating Systems 3 e, (c) 2008 Prentice-Hall, Inc. All rights reserved. 0-13-**6006639**

Scheduling in Real-time Systems

- Realtime system: must react to external events within a fixed amount of time
- Periodic events vs aperiodic events
  - e.g., in voIP system, incoming audio packets are periodic events
  - in intrusion detection system, detected abnormal signal is an aperiodic event
- Hard real time
  - absolute deadline
- Soft real time
  - soft deadline: ok to miss occasionally, e.g., multimedia system

Tanenbaum, Modern Operating Systems 3 e, (c) 2008 Prentice-Hall, Inc. All rights reserved. 0-13-6006639

Outline

- CPU Scheduling
  - Importance of scheduling in diff. environment
  - CPU bound process, I/O bound process
  - Preemptive, non-preemptive scheduling
- Batch system scheduling algorithms
  - FCFS, Shortest Job First, Shortest Remaining First
- Interactive system scheduling
  - RR, Priority scheduling, Lottery Scheduling
- Realtime system scheduling
- Thread Scheduling
  - User-level thread
  - Kernel-level thread

Thread Scheduling (1)

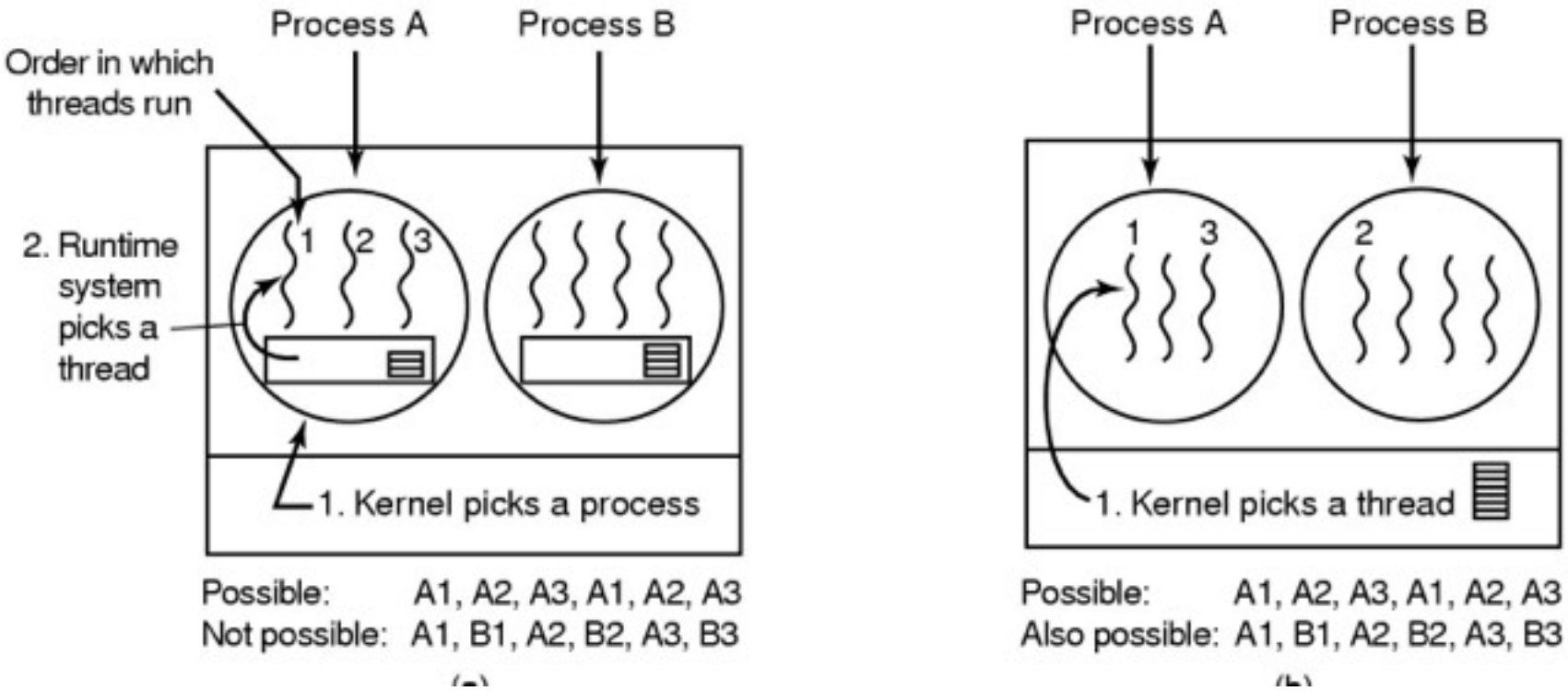


Figure 2-43. (a) Possible scheduling of user-level threads with a 50-msec process quantum and threads that run 5 msec per CPU burst.

Tanenbaum, Modern Operating Systems 3 e, (c) 2008 Prentice-Hall, Inc. All rights reserved. 0-13-**6006639**

Thread Scheduling (2)

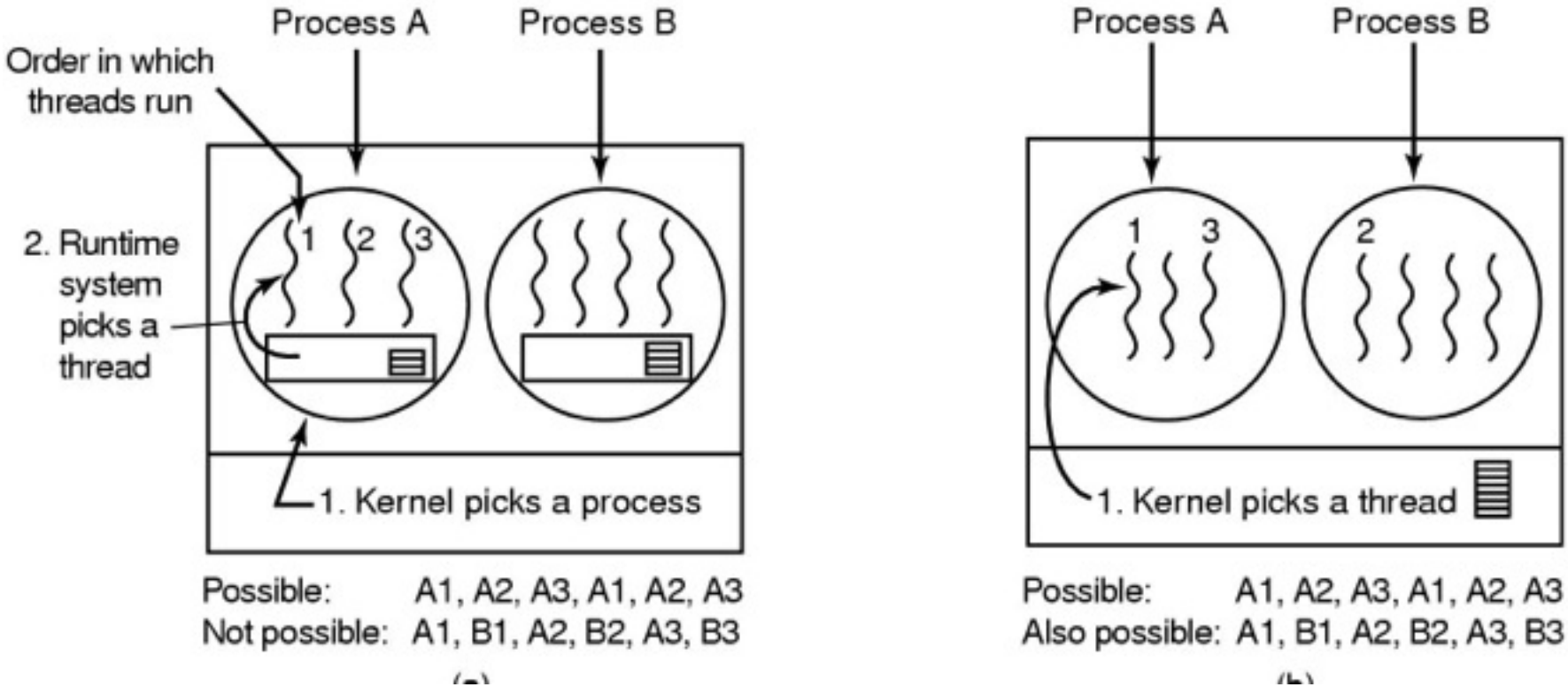


Figure 2-43. (b) Possible scheduling of kernel-level threads with the same characteristics as (a).

Tanenbaum, Modern Operating Systems 3 e, (c) 2008 Prentice-Hall, Inc. All rights reserved. 0-13-**6006639**

Dining Philosophers Problem (1)

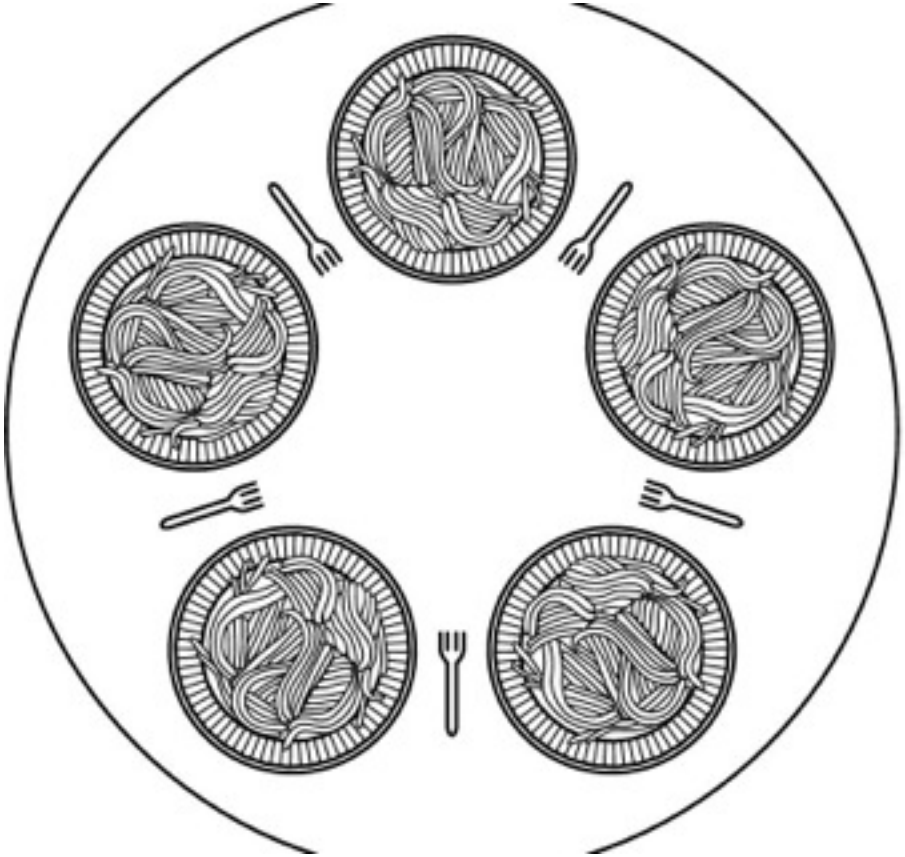


Figure 2-44. Lunch time in the Philosophy Department.

Tanenbaum, Modern Operating Systems 3 e, (c) 2008 Prentice-Hall, Inc. All rights reserved. 0-13-**6006639**

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.

Tanenbaum, Modern Operating Systems 3 e, (c) 2008 Prentice-Hall, Inc. All rights reserved. 0-13-**6006639**

Dining Philosophers Problem (3)

```
#define N                5                  /* number of philosophers */
#define LEFT             (i+N–1)%N          /* number of i's left neighbor */
#defi #define N                5           /* number of philosophers */
#defi #define LEFT             (i+N–1)%N   /* number of i's left neighbor */
#defi #define RIGHT            (i+1)%N     /* number of i's right neighbor */
#defi #define THINKING         0           /* philosopher is thinking */
type( #define HUNGRY           1           /* philosopher is trying to get forks */
int st #define EATING           2           /* philosopher is eating */
sema typedef int semaphore;                 /* semaphores are a special kind of int */
sema int state[N];                          /* array to keep track of everyone's state */
void  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.

Tanenbaum, Modern Operating Systems 3 e, (c) 2008 Prentice-Hall, Inc. All rights reserved. 0-13-6006639

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.

Tanenbaum, Modern Operating Systems 3 e, (c) 2008 Prentice-Hall, Inc. All rights reserved. 0-13-**6006639**

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.

Tanenbaum, Modern Operating Systems 3 e, (c) 2008 Prentice-Hall, Inc. All rights reserved. 0-13-**6006639**

The Readers and Writers Problem (1)

```
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 */
    }
}
```

. . .

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

Tanenbaum, Modern Operating Systems 3 e, (c) 2008 Prentice-Hall, Inc. All rights reserved. 0-13-**6006639**

The Readers and Writers Problem (2)

. . .

```
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 */
     }
}
```

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

Tanenbaum, Modern Operating Systems 3 e, (c) 2008 Prentice-Hall, Inc. All rights reserved. 0-13-**6006639**