

# Chapter 2

## Processes and Threads

- Processes
- Threads
- Interprocess communication
- Classical IPC problems
- Scheduling

# The Process Model

- Process: an abstraction of a running program.
- All runnable software is organized into a number of sequential processes.
- Each process has its own flow of control (i.e. program counter, registers and variables).
- In a multiprogramming environment, processes switch back and forth.
- No built-in assumption about timing in programs (use interrupts instead).

# The Process Model

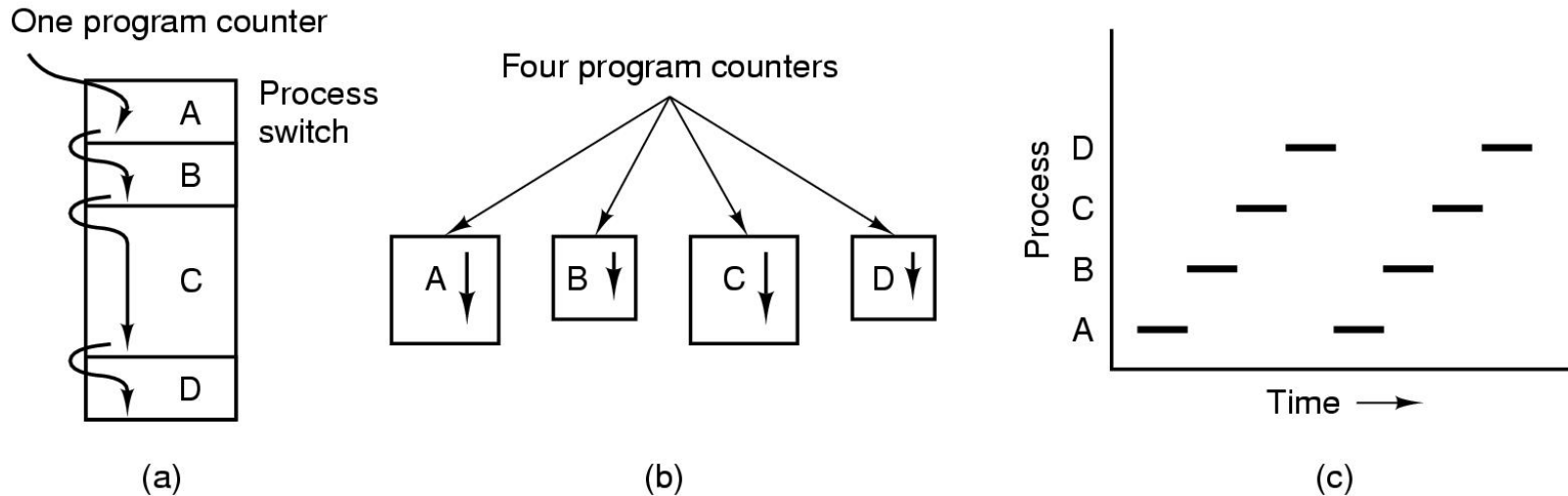


Figure 2-1. (a) Multiprogramming of four programs. (b) Conceptual model of four independent, sequential processes. (c) Only one program is active at once.

# Process Creation

Events which cause process creation:

- System initialization.
- Execution of a process creation system call by a running process (e.g., `fork()` in unix).
- A user request to create a new process (shell command or click an icon).
- Initiation of a batch job.

# Process Termination

Events which cause process termination:

- 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 child process
- Forms a hierarchy
  - UNIX calls this a "process group"
- Windows has no concept of process hierarchy
  - all processes are created equal

# Process States

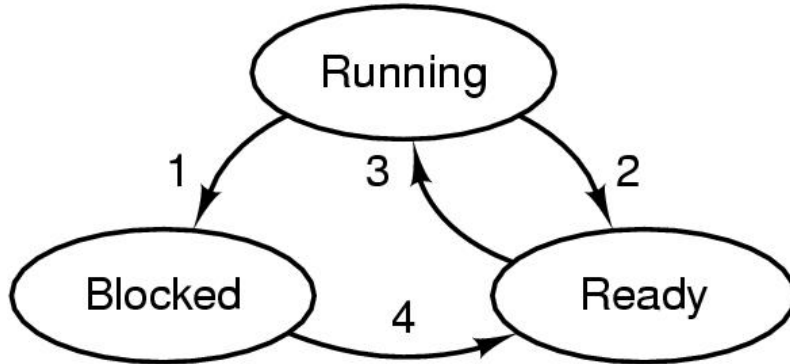
Three state model:

1. Running: using CPU currently.
2. Blocked: unable to run until some external event (e.g. I/O completes) happens.
3. Ready: runnable, temporarily stopped to let another process run.

Four possible transitions:

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

# 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.



# Process States

What happens when all processes in memory are blocked for I/O?

- Swapping: move a process from memory to disk (suspend) and bring another process on disk to memory.

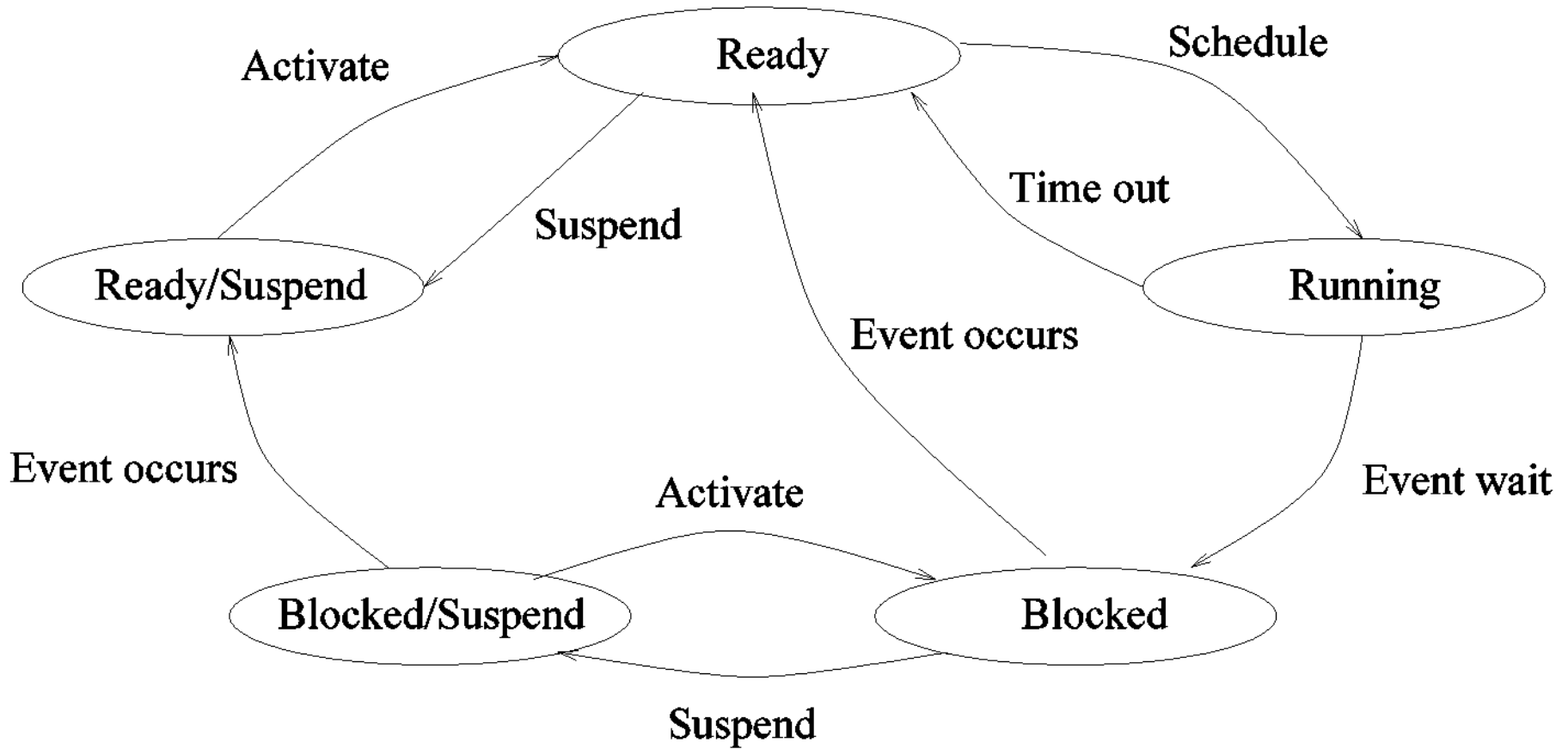
- Five state model, add two more states:

1. Blocked/Suspend: the process is in secondary memory and waiting an event.

2. Ready/Suspend: the process is in secondary memory but is available for execution as soon as it is loaded into memory.

- Transitions among states:

# Five State Model



# Implementation of Processes (1)

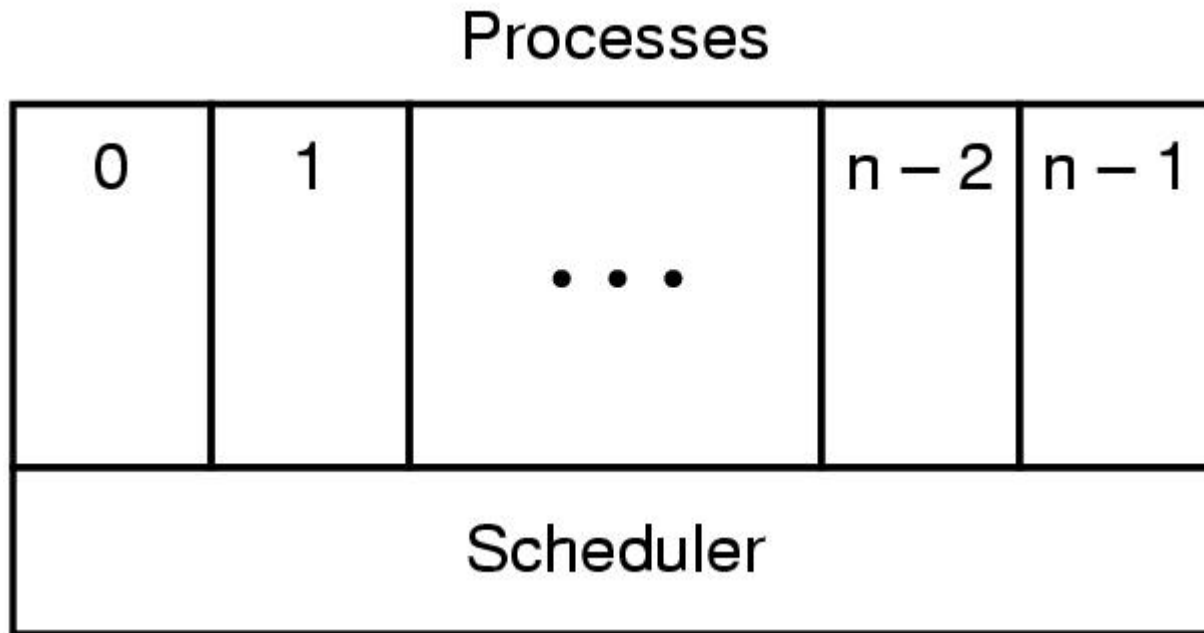


Figure 2-3. The lowest layer of a process-structured operating system handles interrupts and scheduling. Above that layer are sequential processes.

# Implementation of Processes (2)

<b>Process management</b>	<b>Memory management</b>	<b>File management</b>
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.

# Implementation of Processes (3)

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.

Figure 2-5. Skeleton of what the lowest level of the operating system does when an interrupt occurs.

# Thread Usage (1)

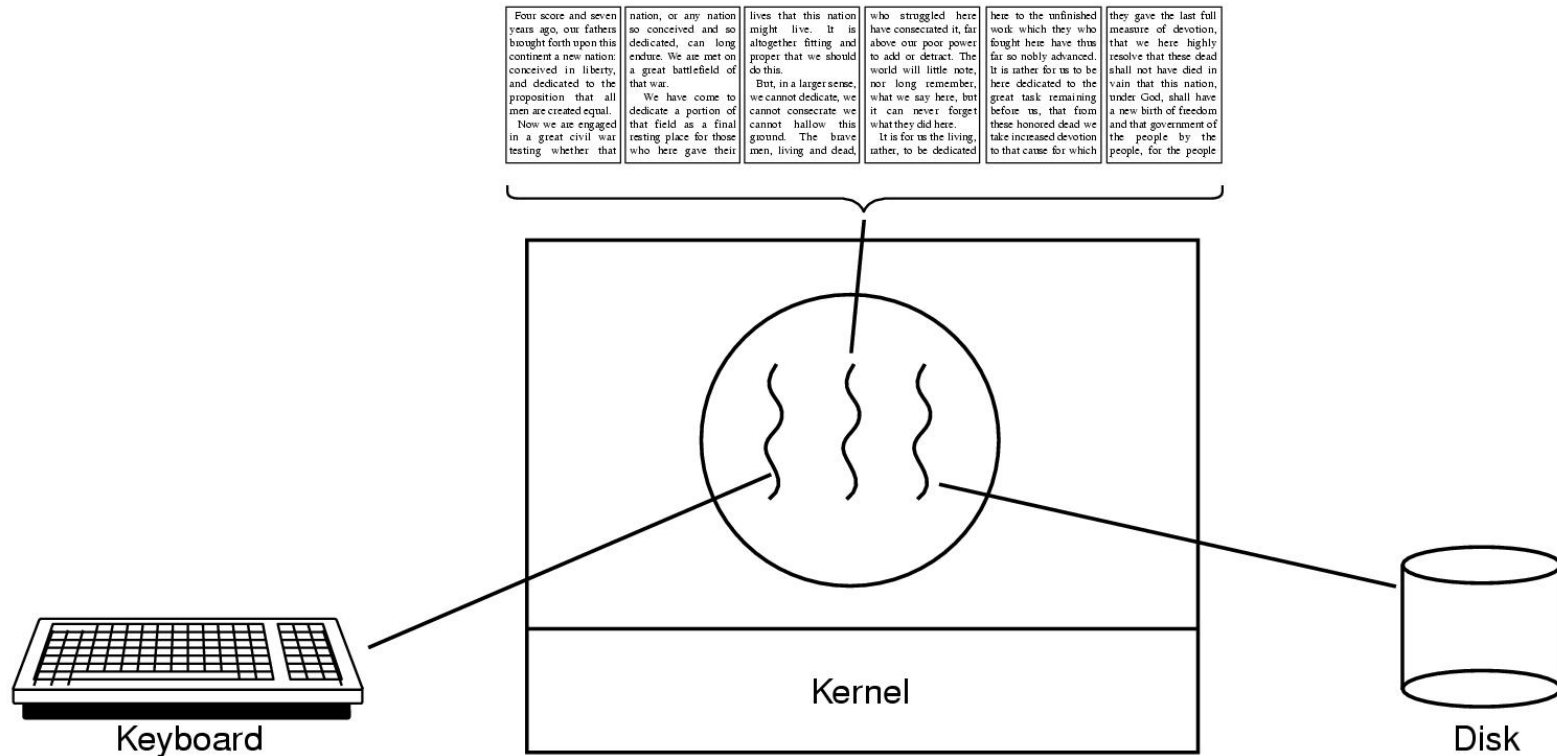


Figure 2-7. A word processor with three threads.

# Thread Usage (2)

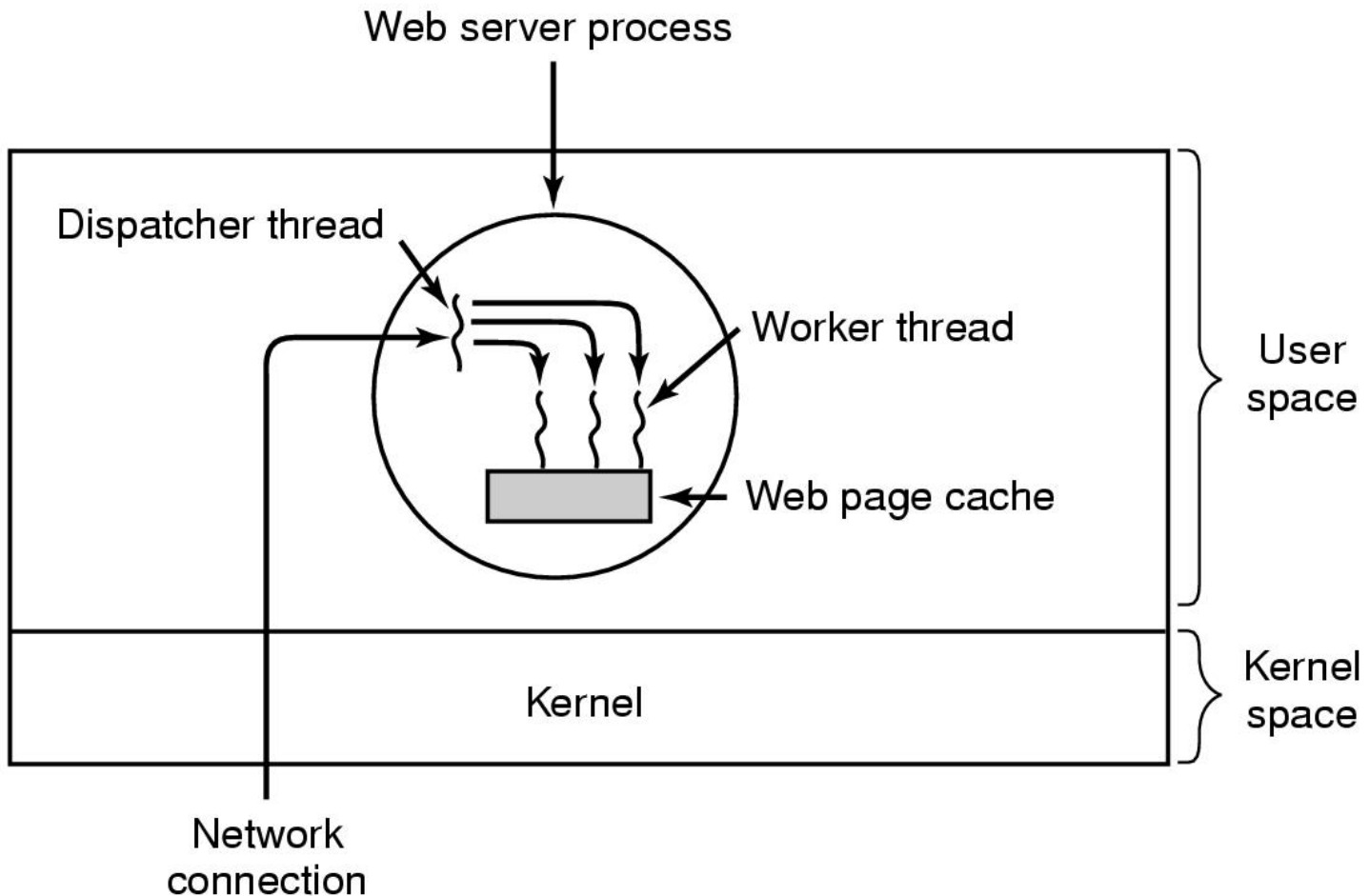


Figure 2-8. A multithreaded Web server.

# 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)

Figure 2-9. A rough outline of the code for Fig. 2-8. (a) Dispatcher thread. (b) Worker thread.



# Thread Usage (4)

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

Figure 2-10. Three ways to construct a server.

# The Classical Thread Model (1)

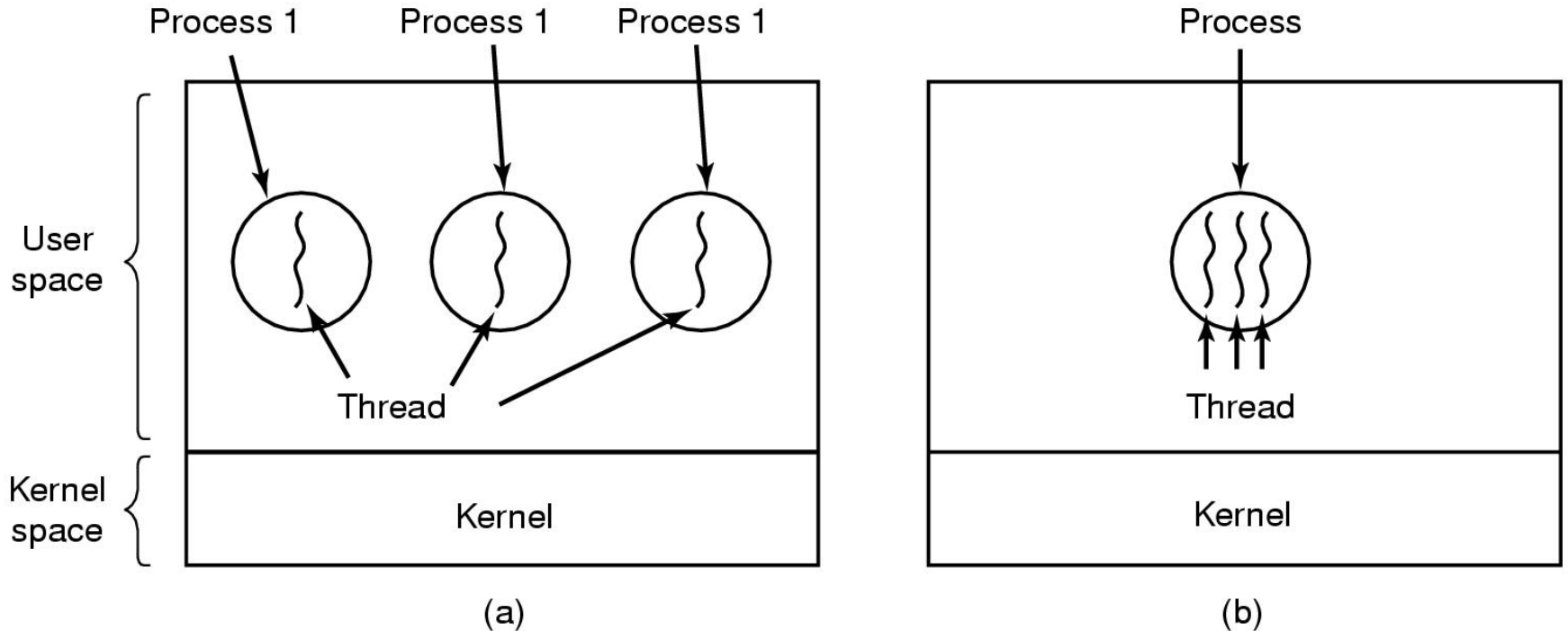


Figure 2-11. (a) Three processes each with one thread. (b) One process with three threads.

# The Classical Thread Model (2)

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

Figure 2-12. The first column lists some items shared by all threads in a process. The second one lists some items private to each thread.

# The Classical Thread Model (3)

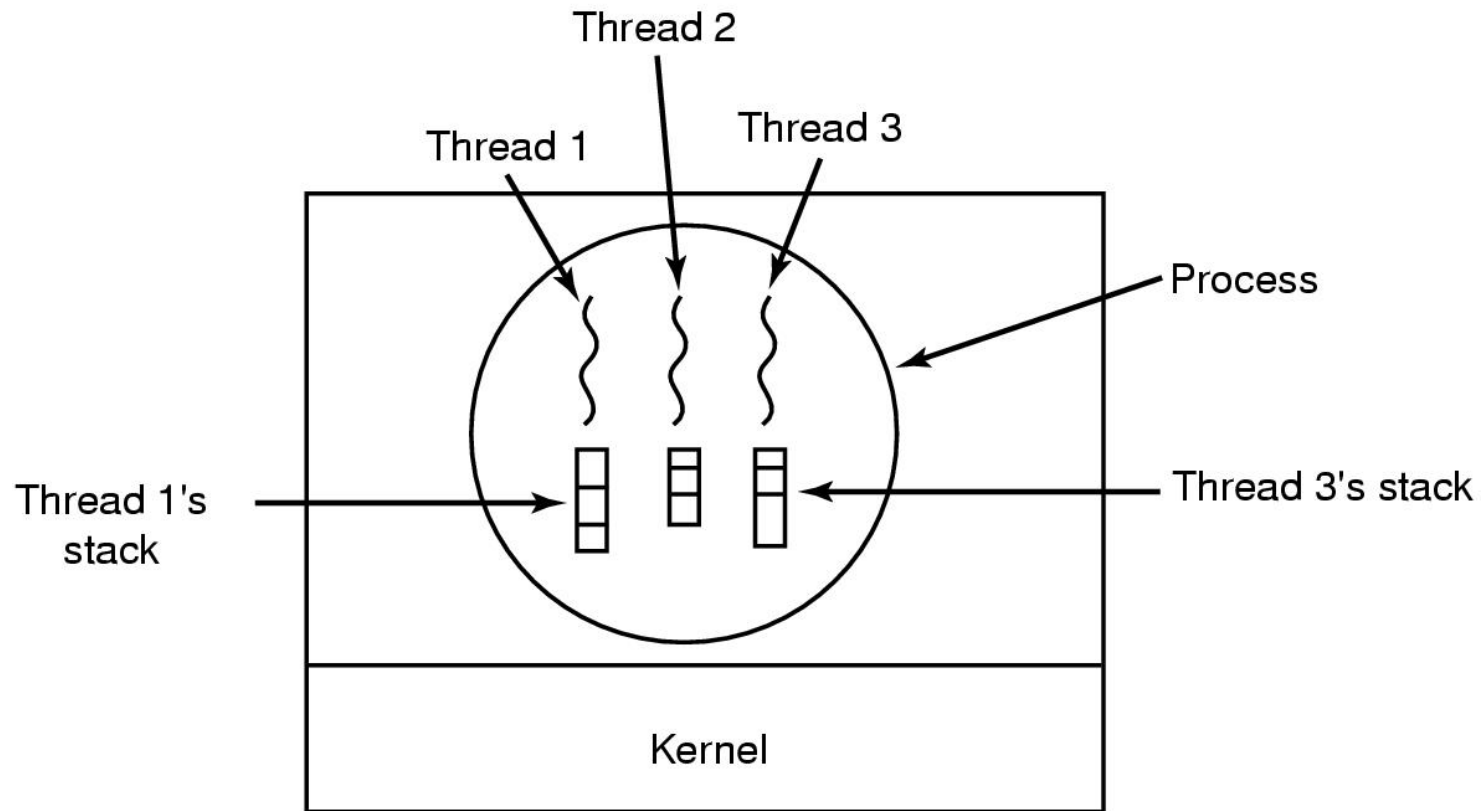


Figure 2-13. Each thread has its own stack.

# POSIX Threads (1)

<b>Thread call</b>	<b>Description</b>
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

Figure 2-14. Some of the Pthreads function calls.

# Implementing Threads in User Space

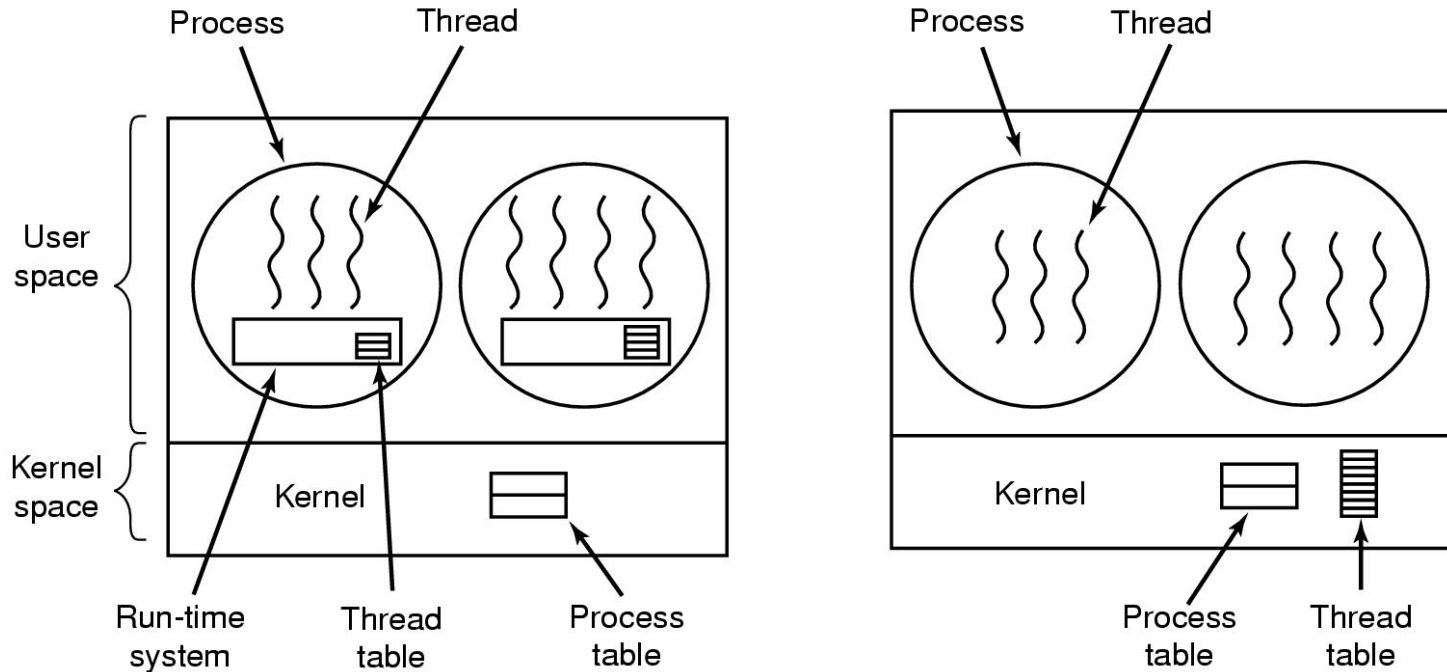


Figure 2-16. (a) A user-level threads package. (b) A threads package managed by the kernel.

# Hybrid Implementations

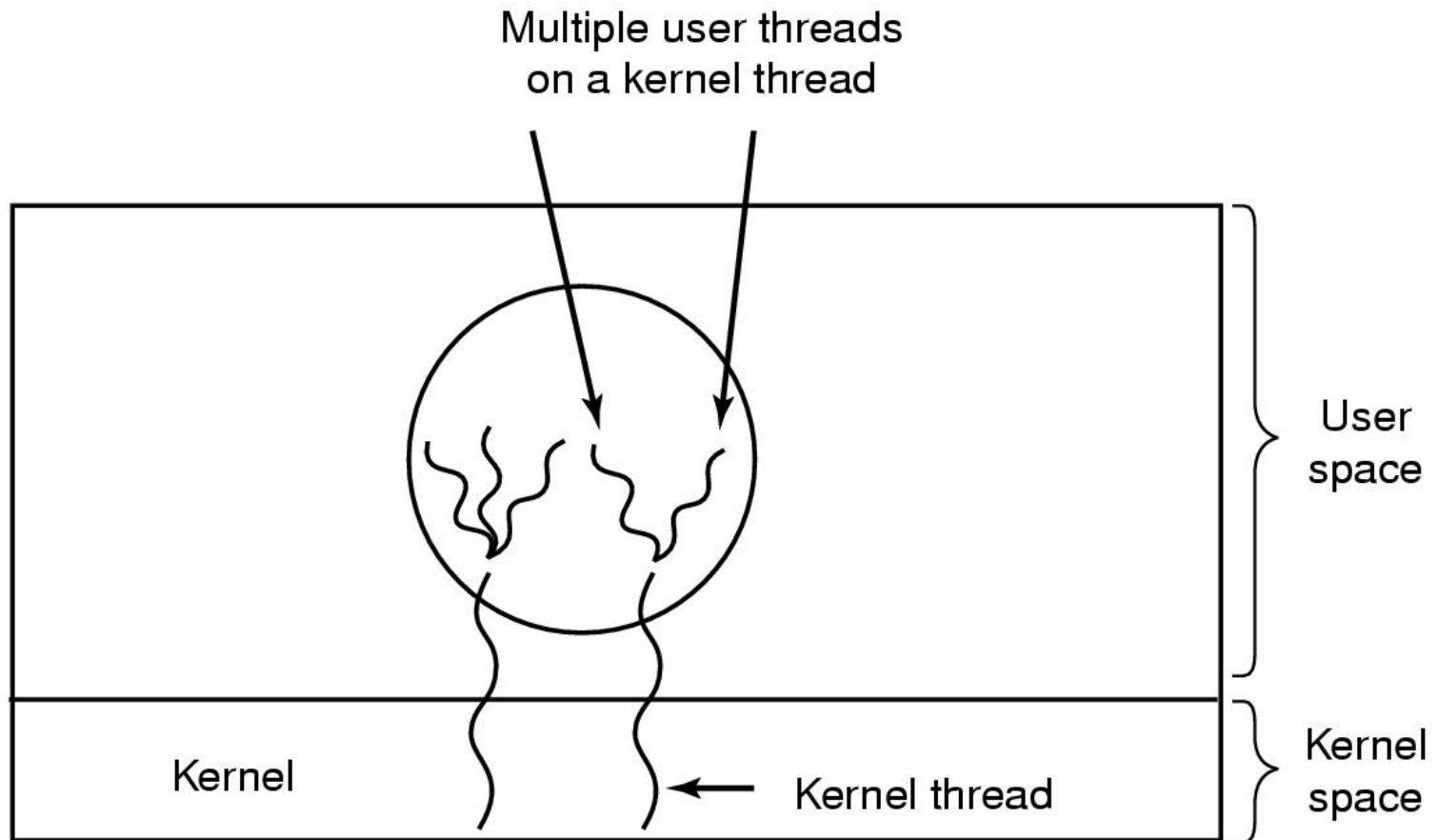


Figure 2-17. Multiplexing user-level threads onto kernel-level threads.

# Pop-Up Threads

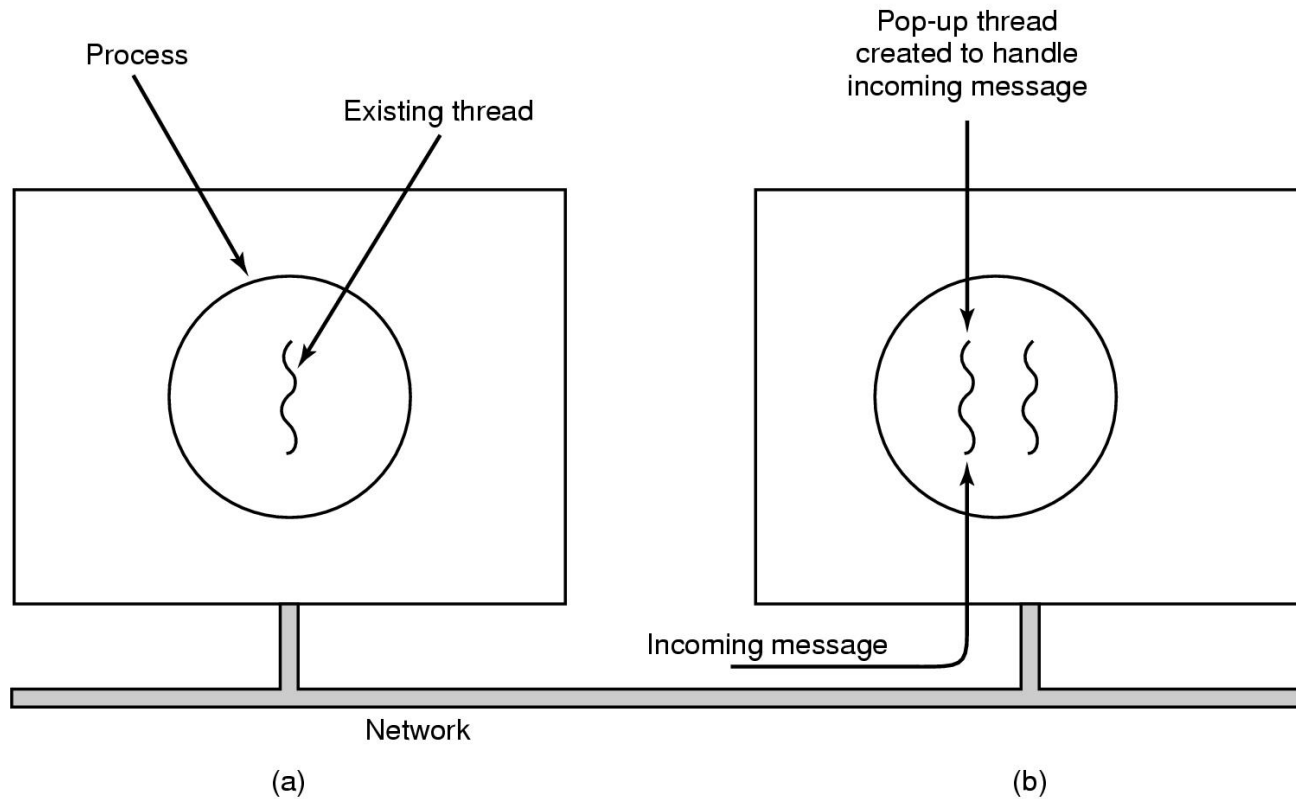


Figure 2-18. Creation of a new thread when a message arrives.  
(a) Before the message arrives.  
(b) After the message arrives.



# Inter-process Communication (IPC)

Processes communicate through shared memory

Spooling: Simultaneous Peripheral Operation On Line

Possible problems (race condition)

Example: a print spooler.

- To print a file, a process enters the file name in a designated Spooler directory (an array implemented with circular queue).
- Another process, printer daemon, prints the files and removes them from the directory.
- Shared variables: in, out.
- Print procedure:
  1. in ->next-free-slot (local variable)
  2. Put the file name to print in the array location indexed by next-free-slot
  3. Increment next-free-slot
  4. next-free-slot ->in.

# Race Conditions

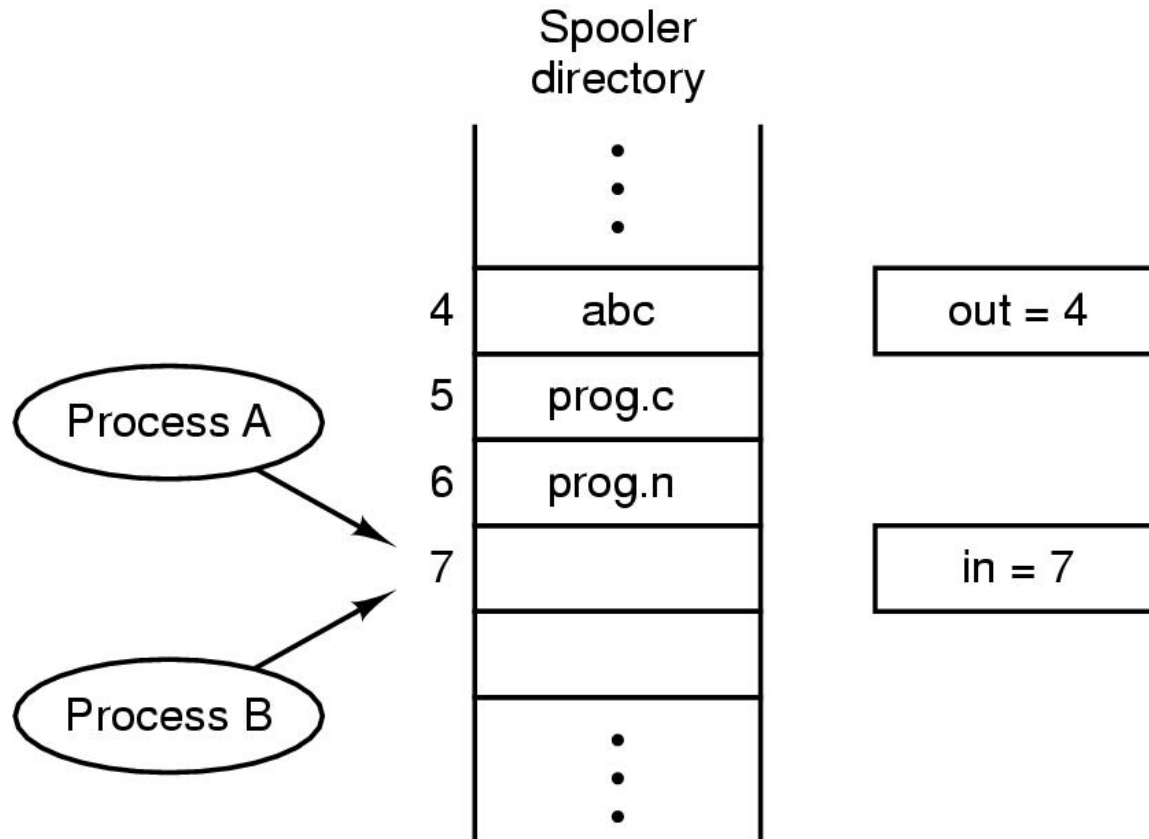


Figure 2-21. Two processes want to access shared memory at the same time.

# Race Conditions

Assume two processes A & B and processes can be switched out during execution.

A sequence of actions which can cause problems:

1. Process A: in(= 7) -> next-free-slot
2. A is switched out and B is running.
3. Process B: in(=7) -> next-free-slot
4. B puts file name to print in Slot 7.
5. B increments its local variable to 8.
6. B stores 8 into in.
7. A is scheduled to run again.
8. A puts file name to print into Slot 7.
9. A increments its local variable to 8.
10. A stores 8 into in.

**Race condition:** Several processes access and manipulate the same data concurrently, and the outcome of the execution depends on the particular access order.

# Race Conditions

Another example:

Bookkeeping application. Need to maintain data coherence, i.e. keep  $a = b$ .

Process 1:  $a = a + 1$ ;  $b = b + 1$

Process 2:  $b = 2 \times b$ ;  $a = 2 \times a$

Initially  $a = b$

Execution sequence:

$a = a + 1$

$b = 2 \times b$

$b = b + 1$

$a = 2 \times a$

At the end  $a \neq b$ .

**Critical section (region):** Portion of a program that accesses shared variables

**Mutual exclusion:** Mechanism which makes sure that two or more processes do not access a common resource at the same time.

# Critical Regions (1)

Four conditions required to avoid race condition:

- No two processes may be simultaneously inside their critical regions.
- No assumptions may be made about speeds or the number of CPUs.
- No process running outside its critical region may block other processes.
- No process should have to wait forever to enter its critical region.

# Critical Regions (2)

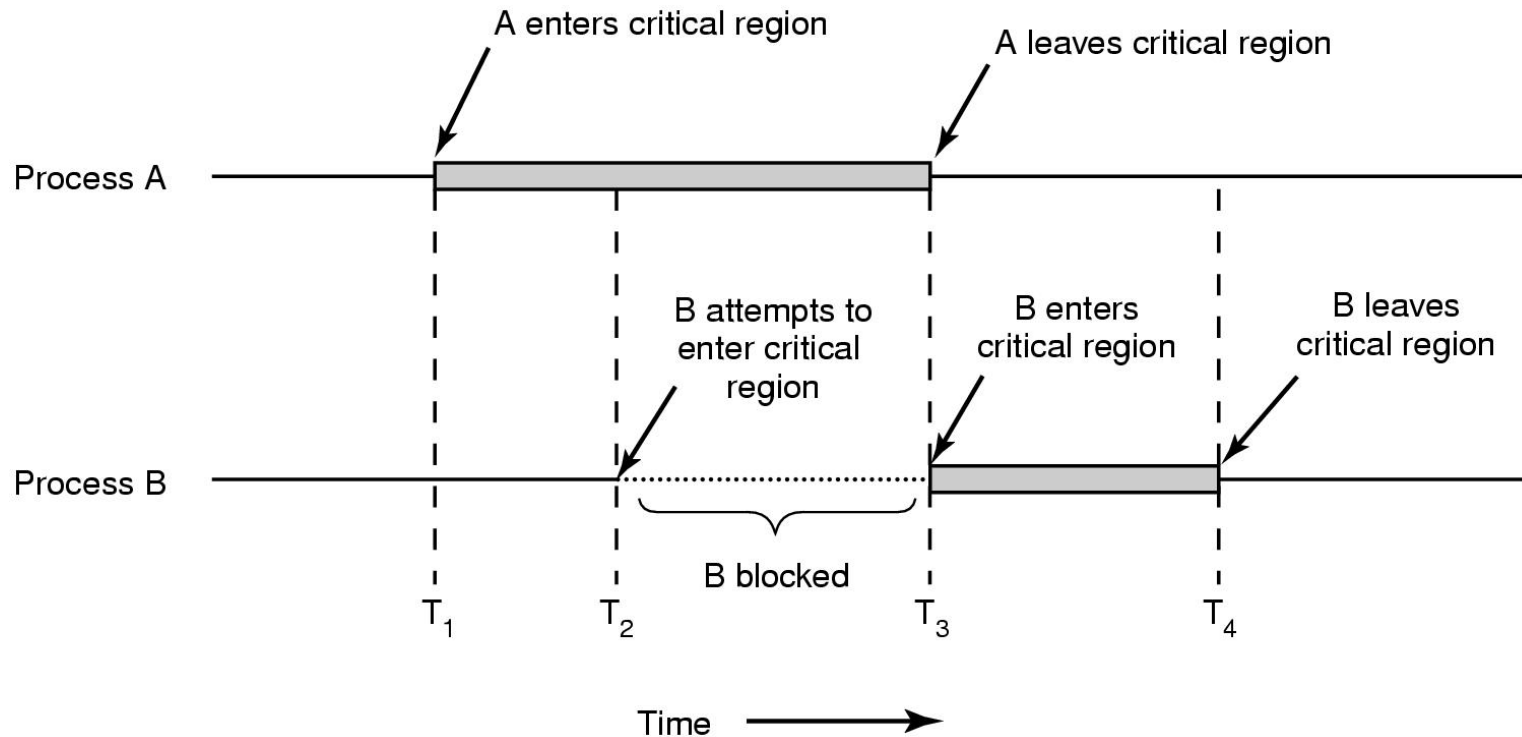


Figure 2-22. Mutual exclusion using critical regions.

# Mutual Exclusion with Busy Waiting

Proposals for achieving mutual exclusion:

- Disabling interrupts
- Lock variables
- Strict alternation
- Peterson's solution
- The TSL instruction

# Disabling Interrupts

A hardware solution:

1. Disable interrupts
2. Enter critical section
3. Do something in critical section
4. Exit critical section
5. Re-enable interrupts

Give too much power to user processes.  
Only works for single CPU.



# Lock Variables

A binary shared variable lock.

lock = 1: critical region occupied

lock = 0: critical region unoccupied

The code for entering critical section:

1. loop: if lock == 1 then goto loop;
2. lock = 1;
3. critical-section();
4. lock = 0;

A possible execution sequence:

1. Process A executes (1) and finds lock = 0. Drops from loop.
2. Process A is switched out.
3. Process B checks lock and sees lock = 0 and drops from loop.
4. Process B sets lock = 1 and enters critical section.
5. Process A wakes up, sets lock = 1 (again) and enters critical section.

# Strict Alternation

Processes take turns to enter critical section

For two processes, use a variable turn:

turn = 0: process 0 can enter critical section

turn = 1: process 1 can enter critical section

Limitations:

1. The faster process has to adapt to the pace of the slower process
2. Two processes have to take turns to enter their critical section. No one can enter twice in a row.

# Strict Alternation

```
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)

Figure 2-23. A proposed solution to the critical region problem.  
(a) Process 0. (b) Process 1. In both cases, be sure to note the semicolons terminating the while statements.

# Peterson's Solution

Combine lock and take turns

Four possibilities for condition:

(turn=process && interested[other]=true)

from the point of view of process 0.

Case 1: turn = 0, interested[1] = false

Process 1 is not in critical region.

Process 0 enters critical region.

Case 2: turn = 0, interested[1] = true

Process 1 is in critical region.

Process 0 waits.

Case 3: turn = 1, interested[1]=false

Impossible.

Case 4: turn = 1, interested[1]=true

Process 1 is trying to enter critical region, but process 0's turn first.

Process 0 enters critical region.

# 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 */
}
```

Figure 2-24. Peterson's solution for achieving mutual exclusion.

# The TSL Instruction

Need hardware support (machine must have this special instruction)

TSL: combine

(Mem) -> R and 1 -> Mem into an atomic operation.

# 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 nonzero, 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
```

Figure 2-25. Entering and leaving a critical region using the TSL instruction.

# Sleep and Wakeup

Avoid busy waiting. Use sleep and wakeup.

Sleep: a system call that causes the caller to block until another process wakes it up.

Wakeup(p): wakeup process p.

How to handle wakeup if sent to a process not asleep:

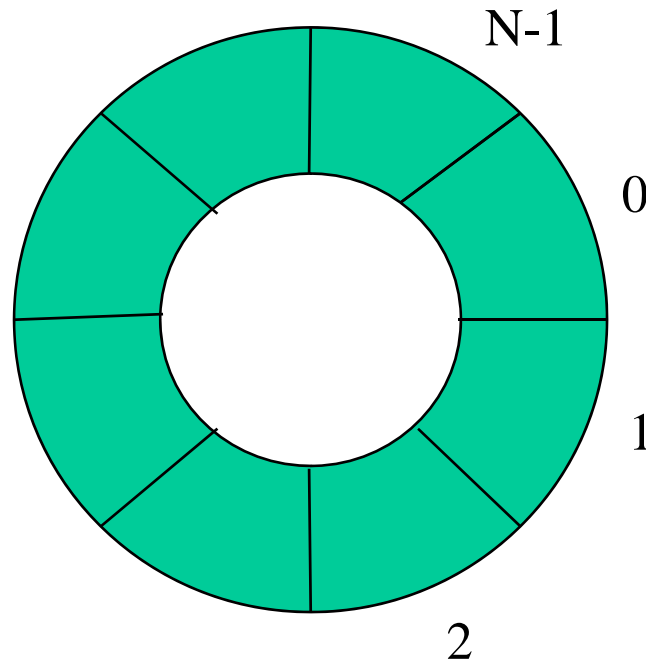
- Ignore
- Queue



# The Producer-Consumer Problem

A circular buffer has  $n$  slots. Producer puts an item into buffer each time. Consumer takes an item out of the buffer each time.

Use sleep and wakeup to write procedures for producer and consumer.



# The Producer-Consumer Problem

```
#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 */
    }
}
```

Figure 2-27. The producer-consumer problem with a fatal race condition.

# The Producer-Consumer Problem

Problem: wakeup sent to a process that has not gone to sleep.

Example:

- Buffer empty.
- Consumer reads count = 0 and is switched out (not sleep yet).
- Producer enters an item in buffer and increments the counter.
- Producer sends wakeup. Wakeup lost.
- Consumer is scheduled to run again.
- Consumer goes to sleep.
- Producer eventually fills buffer and goes to sleep.

Quick fix:

Set wakeup waiting bit if wakeup is sent to a non-sleeping process.

If a process tries to go sleep and the bit is on, clears the bit and stays awake.

More than one wakeup ?

# Semaphores

- A synchronization integer variable.
- Two atomic operations: **down** and **up**
- A queue for blocking
- Implementation of semaphores

```
type semaphore = record  
  value : integer  
  l: queue of processes  
end;
```

```
down(s): If s.value >= 1 then  
  s.value = s.value - 1  
else block the process on the semaphore queue s.l  
(i.e. add the process to queue s.l)
```

```
up(s): If some processes are blocked on s  
then unblock a process (remove a process from queue s.l)  
else s.value = s.value + 1
```

# Semaphores

## Two types of semaphores

- Binary semaphore:

- Two values 0 and 1, used for mutual exclusion (i.e. to ensure that only one process is accessing shared information at a time)

- semaphore mutex = 1

- down(mutex);

- critical-section();

- up(mutex);

- Counting semaphore:

- Used for synchronizing access to a shared resource by several concurrent processes (i.e. to control how many processes can concurrently perform operations on the shared resource).

# Semaphores

Semaphores are not supported by hardware, but can be easily implemented using TEST and SET LOCK instruction and enable/disable interrupts.

Example: Solving the producer consumer problem by semaphores

# Semaphores

```
#define N 100
typedef int semaphore;
semaphore mutex = 1;
semaphore empty = N;
semaphore full = 0;

void producer(void)
{
    int item;

    while (TRUE) {
        item = produce_item();
        down(&empty);
        down(&mutex);
        insert_item(item);
        up(&mutex);
        up(&full);
    }
}

void consumer(void)
{
    int item;

    while (TRUE) {
        down(&full);
        down(&mutex);
        item = remove_item();
        up(&mutex);
        up(&empty);
        consume_item(item);
    }
}
. . }
```

*/\* number of slots in the buffer \*/*  
*/\* semaphores are a special kind of int \*/*  
*/\* controls access to critical region \*/*  
*/\* counts empty buffer slots \*/*  
*/\* counts full buffer slots \*/*

*/\* TRUE is the constant 1 \*/*  
*/\* generate something to put in buffer \*/*  
*/\* decrement empty count \*/*  
*/\* enter critical region \*/*  
*/\* put new item in buffer \*/*  
*/\* leave critical region \*/*  
*/\* increment count of full slots \*/*

*/\* infinite loop \*/*  
*/\* decrement full count \*/*  
*/\* enter critical region \*/*  
*/\* take item from buffer \*/*  
*/\* leave critical region \*/*  
*/\* increment count of empty slots \*/*  
*/\* do something with the item \*/*

Figure 2-28. The producer-consumer problem using semaphores.

# Semaphores

The sequence of down and up operations matters.

Reverse the sequence of downs in producer:

1. Buffer empty.
2. Run producer through while loop n times.
3. Buffer full.
4. Run producer again.
5. Producer sleeps on semaphore empty.
6. Run consumer.
7. Consumer sleeps on semaphore mutex.
8. Deadlock.



# Semaphores

Reverse the sequence of downs in consumer:

1. Buffer empty.
2. Run consumer.
3. Consumer sleeps on semaphore full.
4. Run producer.
5. Down(empty), ok.
6. Producer sleeps on semaphore mutex.
7. Deadlock again.

# Semaphores

Although semaphores provide a simple and sufficiently general scheme for IPC, they suffer from the following drawbacks:

1. A process that uses a semaphore has to know WHICH other processes use these semaphore. May also have to know HOW these processes are using the semaphore.
2. Semaphore operations must be carefully installed in a process. The OMISSION of an up or down may result inconsistencies or deadlocks.
3. Programs using semaphores can be extremely hard to verify for correctness.

# Monitors

Monitor is a high-level synchronization primitive

Combine three features:

1. Shared data
2. Operations on the data
3. Synchronization

Programming constructs, implemented by compiler.

Only one process active in a monitor at a time (implicitly controlled by monitor lock)

Easier and safer to use.

# Monitors

<Monitor name>: monitor

begin

Declaration of data local to the monitor

...

procedure <name> (<formal parameters>);

begin

procedure body

end;

Declaration of other procedures.

...

begin

Initialization of local data of the monitor

end;

end;

# Monitors

Need some way to wait, two choices:

- (1) Busy-wait inside monitor
- (2) Put the process to sleep inside monitor

Condition variables (things to wait on):

- **wait**(condition): release monitor lock, and put the process to sleep. When process wakes up again, re-acquire monitor lock immediately.
- **signal**(condition): wake up one process waiting on condition variable (FIFO).  
If no body is waiting, do nothing (no history).
- **broadcast**(condition): wake up all processes waiting on the condition variable.

# Monitors

Need to decide who gets the monitor lock after a signal:

- On signal, signaler keeps monitor lock. Awakened process waits for monitor lock with no special priority.
- On signal, awakened process gets the monitor lock. Signaler exits from monitor immediately.

# Monitors

```
monitor example
  integer i;
  condition c;

  procedure producer( );
  :
  :
  end;

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

Figure 2-33. A monitor.

# Monitors

```
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



# Message Passing

Why use message passing

- Two parts of communication can be totally separated (no shared data)
- No invisible side effects
- No need to know the other part

Message:

A piece of information that is passed from one process to another.

Mailbox:

A shared data structure where messages are stored between the time they are sent and the time they are received.

# Message Passing

Operations:

**send:** copy a message into mailbox. If the mailbox is full wait until there is enough space in the mailbox.

Format: **send**(destination, message)

**receive:** copy a message out of mailbox, and delete from mailbox. If the mailbox is empty, then wait until a message arrives.

Format: **receive**(source, message)

# Message Passing

Design issues of message system

Addressing: how to specify the sending and receiving processes.

Direct addressing: sender and receiver communicate directly.

send: a specific identification of the destination process, such as  
process@machine.domain

receive:

- (a) explicit addressing
- (b) implicit addressing

Indirect addressing: messages are sent to a shared data structure called mailboxes (queues that can temporarily hold messages)

Relationship between mailboxes and processes

- (a) One mailbox per process. Use process name in send, no name in receive.
- (b) No strict mailbox-process association, use mailbox name.

# Message Passing

Extent of buffering

Buffering

None - rendezvous protocol

Blocking vs. non-blocking operations

Blocking receive:

receive message if mailbox is not empty, otherwise wait until message arrives

Non-blocking receive:

receive message if mailbox is not empty, otherwise return.

Blocking send:

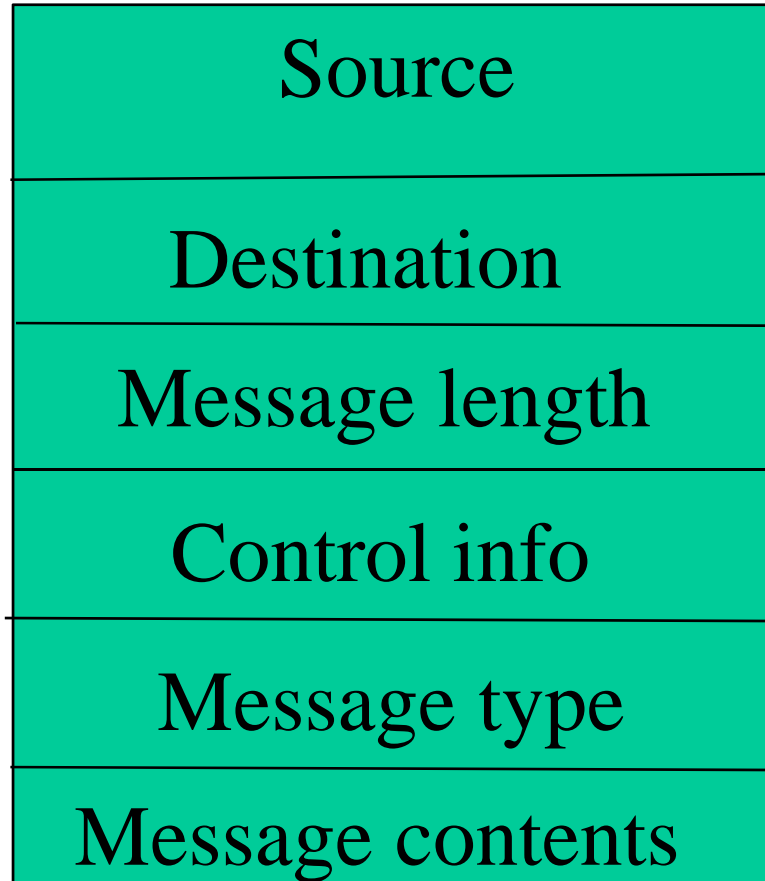
wait until mailbox has space.

Nonblocking send:

return "full" if no space in mailbox.

Four possible send and receive combinations.

# Message Passing



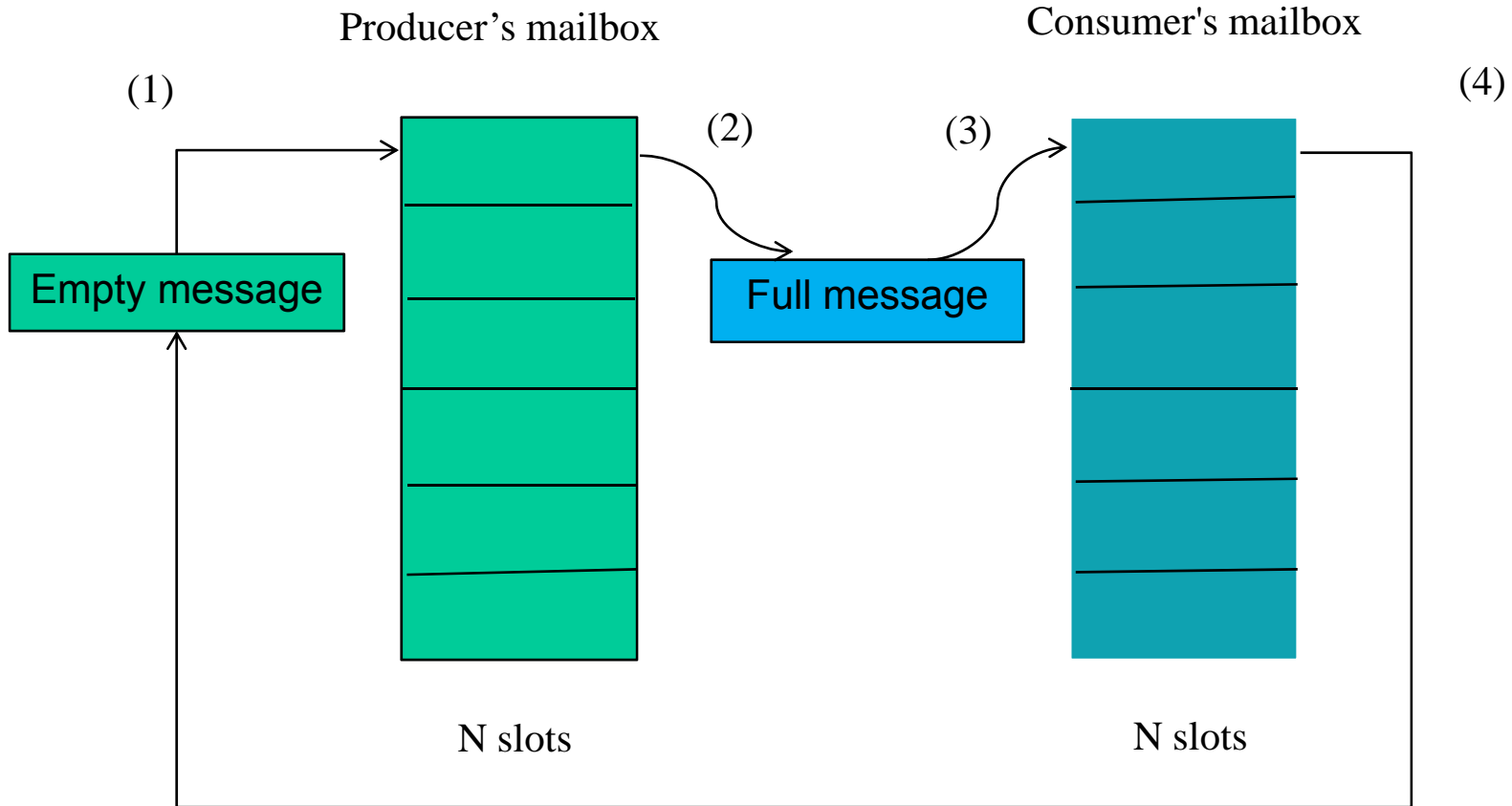
# Message Passing

Queueing discipline:

First in first out (FIFO)

Priority

# Message Passing



# Message Passing

- (1) Consumer sends empty message to producer's mailbox.
- (2) Producer takes empty message and builds full message.
- (3) Producer sends full message to consumer's mailbox.
- (4) Consumer takes full message out and consumes it.



# Producer-Consumer Problem with 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 */
    }
}
```

...

Figure 2-36. The producer-consumer problem with N messages.

# Producer-Consumer Problem with Message Passing

• • •

```
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 */
    }
}
```

Figure 2-36. The producer-consumer problem with N messages.

# Equivalence of Primitives

Semaphores, monitors and messages are equivalence. Each of these methods can be used to implement the other methods.


## 1. Implement monitors with semaphores

Associate with each monitor a binary semaphore mutex (monitor lock), initially 1.

Associate with each condition variable a semaphore, initially 0.

Translate:

wait(c)  up(mutex); down(c); down(mutex)

signal(c)  up(c)

# Monitors

```
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;
```

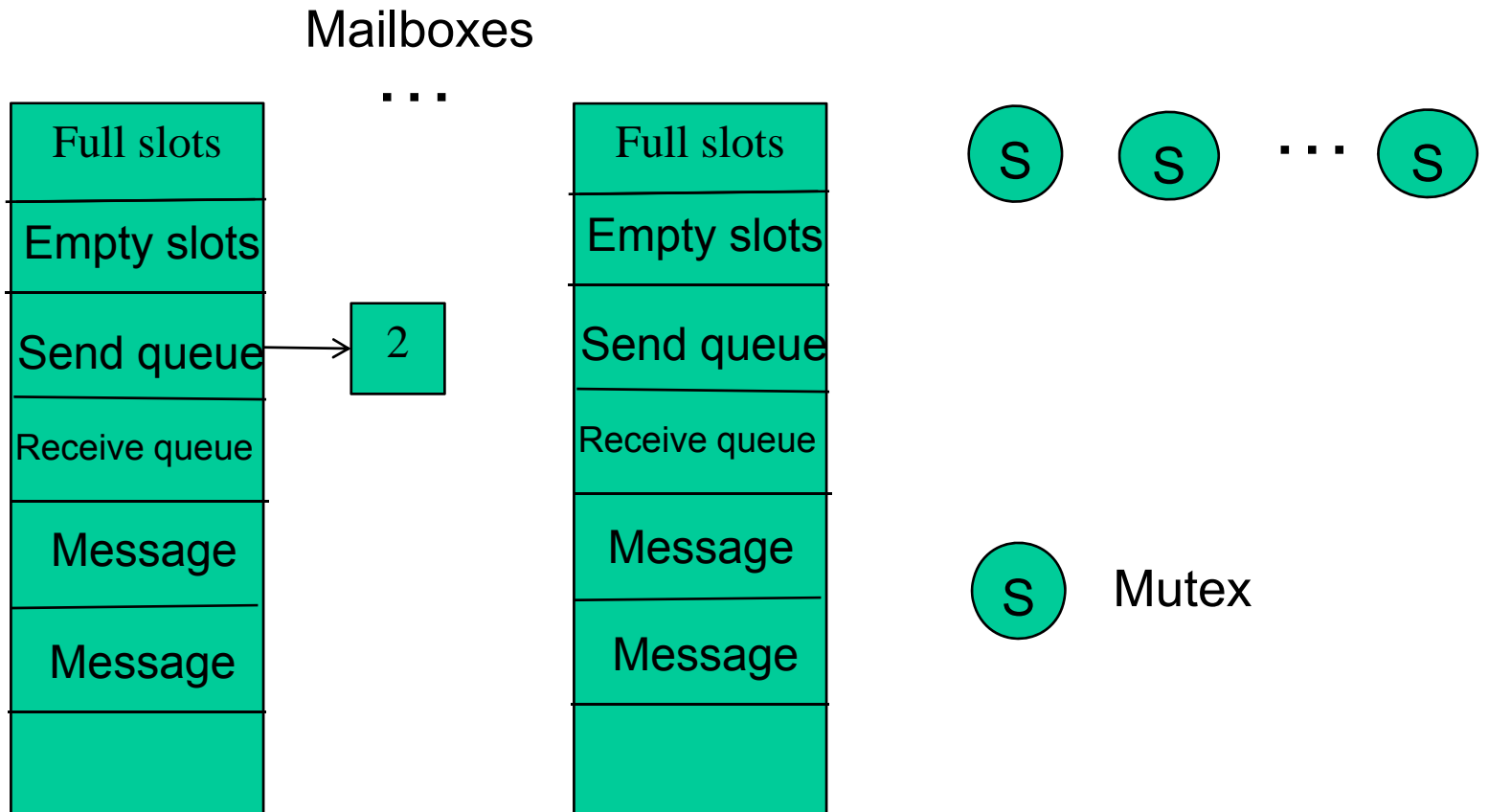
Outline of producer-consumer problem with monitors

# Equivalence of Primitives

## 2. Implement message passing with semaphores.

- Associate with each process a semaphore, initially 0, on which it will block.
- A shared buffer area holds mailboxes. Each mailbox contains:
  - # full slots
  - # empty slots
  - send queue (those processes which cannot send their messages to the mailbox)
  - receive queue (those processes which cannot receive their message from the mailbox)
  - messages linked together
- A semaphore, mutex, to protect the shared buffer area.

# Equivalence of Primitives



# Equivalence of Primitives

**send/receive** operations:

Case 1. Mailbox has at least one empty or full slot:

down(mutex)

insert/remove message

update counters and links

up(mutex)

Case 2. Process  $i$  does receive on an empty mailbox:

down(mutex)

enter receive queue

up(mutex)

down( $P_i$ )

down(mutex)

Case 3. Process  $i$  does send on a full mailbox:

down(mutex)

enter send queue

up(mutex)

down( $P_i$ )

down(mutex)

# Equivalence of Primitives

How to wake up sleeping processes?

- If a receiver receives a message from the full mailbox, wakes up (does up) the first process in the send queue.
- If a sender sends a message to the empty mailbox, wakes up the first process in the receive queue.



# Equivalence of Primitives

## 3. Implement semaphore with monitors

Associate with each semaphore a counter and a linked list.

- Counter stores the value of the semaphore
- Linked list stores the processes sleeping on the semaphore

Associate with each process a condition variable

Operations:

down(s) : If  $\text{counter}_s > 0$ , then  $\text{counter}_s --$   
else {enter linked list of s; wait (Pi)}

up(s) : If linked list not empty, then {remove one process from the list,  
say, Pi; signal(Pi)}  
else  $\text{counter}_s ++$ ;

# Equivalence of Primitives

4. Implement messages with monitors.

- Associate with each process a condition variable for blocking
- A shared buffer
- Similar to semaphores except no mutex necessary.

# Equivalence of Primitives

5. Implement semaphores with messages.

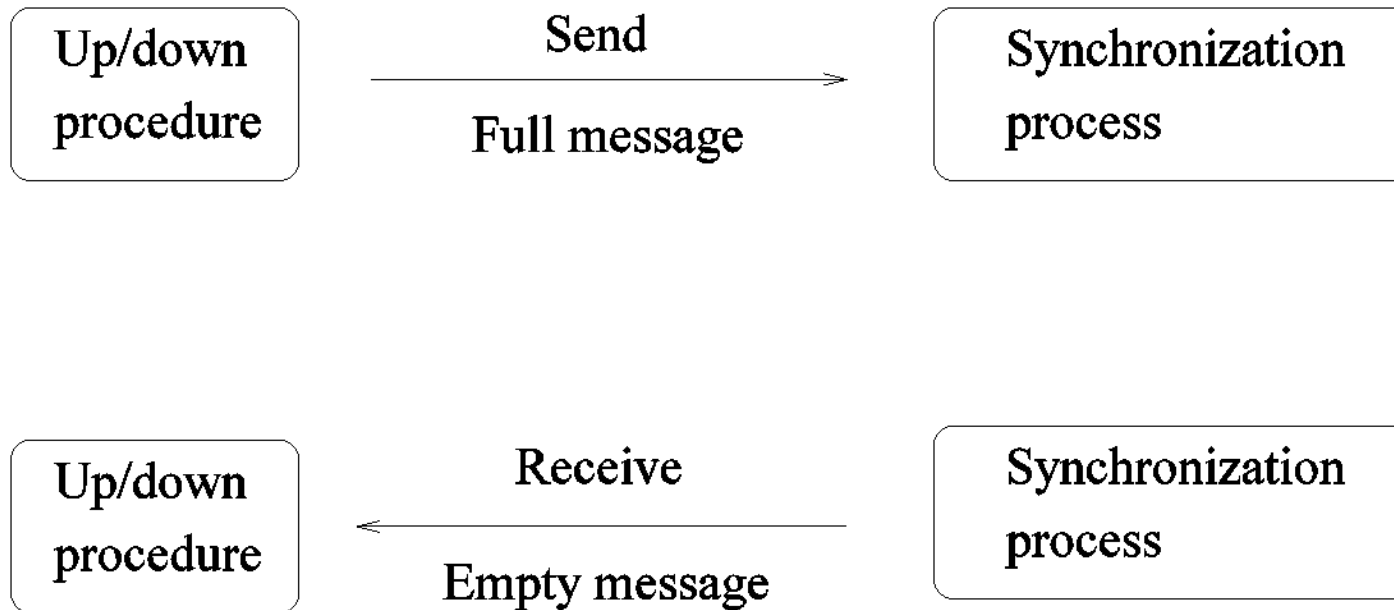
For mutual exclusion, introduce a new process, synchronization process.

Associate each semaphore with a counter and a linked list of waiting processes.

Operations:

To do up or down on a semaphore, call the corresponding library procedure up or down.

# Equivalence of Primitives



# Equivalence of Primitives

Synchronization process does:

down: If count > 0 {counter--; send back empty message}  
else {enter caller into queue and does not send reply;}

up: If counter = 0 {move one process out of queue; send  
this process a reply }  
else counter++;

# Equivalence of Primitives

## 6. Implement monitor with message passing

Combine (5) and (1). That is, using messages to implement semaphores first, then using semaphores to implement monitors.

# Barriers

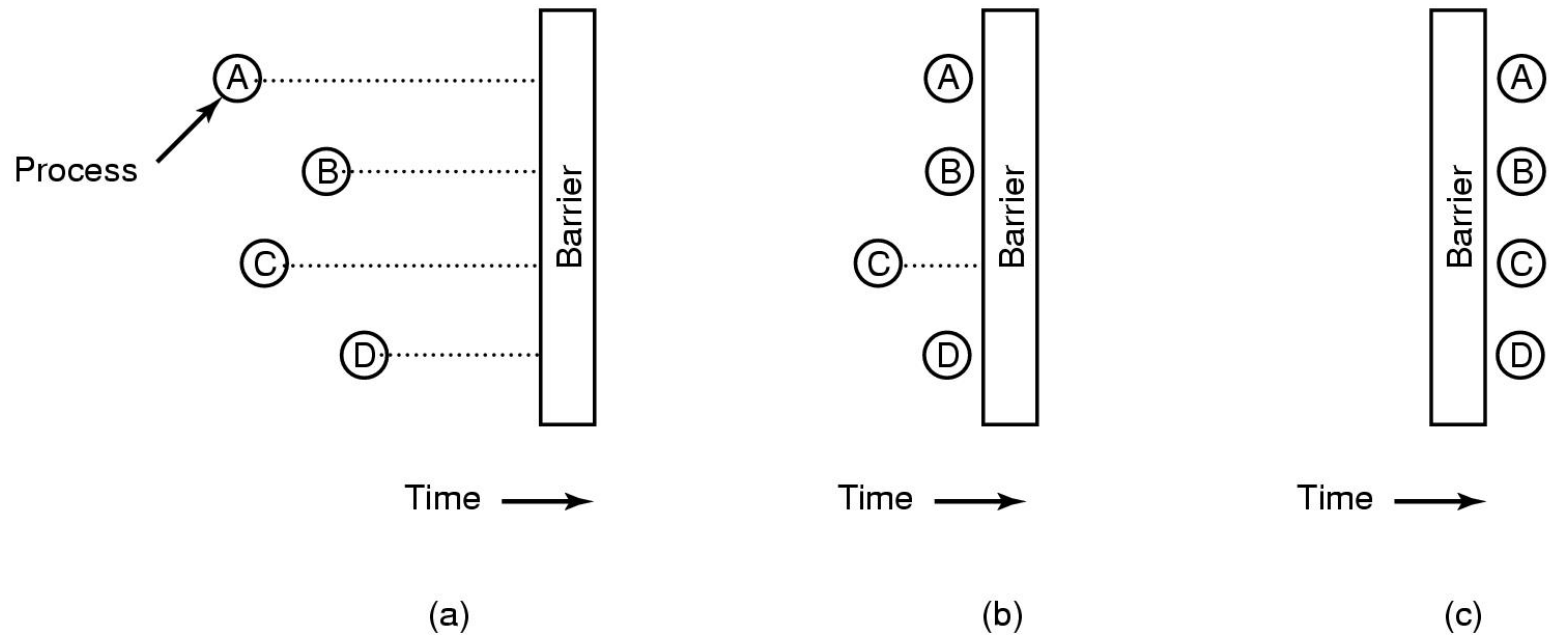


Figure 2-37. Use of a barrier. (a) Processes approaching a barrier. (b) All processes but one blocked at the barrier. (c) When the last process arrives at the barrier, all of them are let through.

# Classical IPC Problems

The dining philosophers problem

Problem description:

Five philosopher sit around a round table, and each of them has one fork.

Activities: eating and thinking.

To eat, need two adjacent forks.

Goal: no starvation.

Useful for modeling processes that are competing for exclusive access to a limited number of resources, such as tape drive or other I/O devices.



# Dining Philosophers Problem

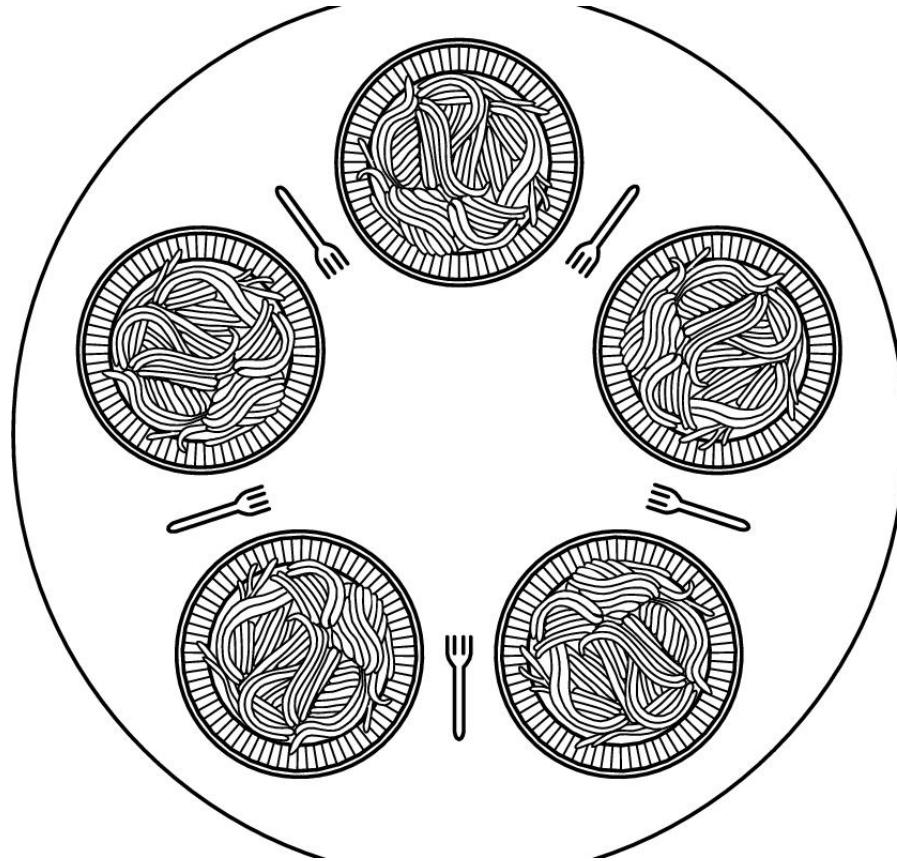


Figure 2-44. Lunch time in the Philosophy Department.

# Dining Philosophers Problem

```
#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.

# Dining Philosophers Problem

```
#define N          5          /* number of philosophers */
#define LEFT      (i+N-1)%N  /* number of i's left neighbor */
#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 */
typedef enum {
#define HUNGRY    1          /* philosopher is trying to get forks */
#define EATING    2          /* philosopher is eating */
} enum state;
typedef int semaphore;      /* semaphores are a special kind of int */
semaphore 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 */
    }
}
```

Figure 2-46. A solution to the dining philosophers problem.

# Dining Philosophers Problem

...

```
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.

# Dining Philosophers Problem

• • •

```
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.

# Dining Philosophers Problem

Problems with solution 2:

Assume Philosophers 1 and 4 eat for a long time and Philosophers 2 and 3 eat for a short time. A possible execution sequence:

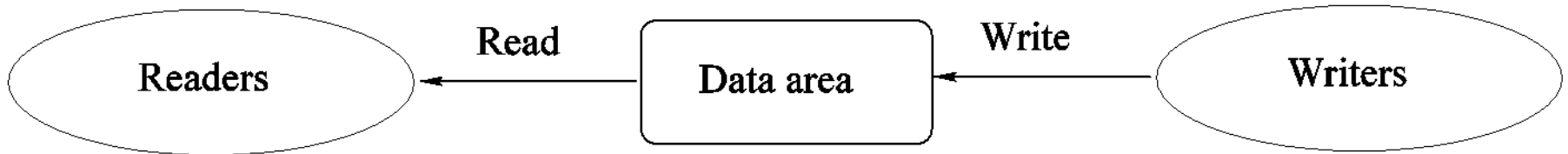
1. 4 and 1 eating.
2. 0 and 3 become hungry (blocked).
3. 4 finishes eating and checks neighbors.
4. 0 IS NOT ALLOWED TO EAT, but 3 is allowed to eat.
5. 4 becomes hungry again.
6. 3 finishes eating and allows 4 to eat again.
7. 1 finishes, 0 IS STILL NOT ALLOWED TO EAT because of 4.
8. 2 is allowed to eat.
9. 1 becomes hungry again.
10. 2 finishes and allows 1 to eat.
11. repeat, 0 IS NEVER ALLOWED TO EAT.

# Dining Philosophers Problem

A working solution:

```
#define N 5
typedef int semaphore;
semaphore fork[N]; /* initially 1; */
semaphore room = 4; /* allow 4 in the dining room */
philosopher(i)
int i;
while (TRUE) {
  think();
  down(room); /* get into dining room */
  down(fork[i]); /* get left fork */
  down(fork[(i+1)%5]); /* get right fork */
  eat();
  up(fork[(i+1)%5]); /* put back right fork */
  up(fork[i]); /* put back left fork */
  up(room); /* get out of dining room */
}
```

# The Readers and Writers Problem



Problem description:

A data area (file or memory) shared among a number of processes.  
Some processes (readers) only read the data area.  
Other processes (writers) only write to the data area.

Conditions must be satisfied:

1. Any number of readers may simultaneously read the data area.
2. Only one writer at a time may write to the data area.
3. If a writer is writing to the data area, no readers may read it.



# The Readers and Writers Problem

Special type of mutual exclusion problem. A special solution can do better than a general solution.

- Solution one:

Readers have priority. Unless a writer is currently writing, readers can always read the data.

- Solution two:

Writers have priority. Guarantee no new readers are allowed when a writer wants to write.

- Other possible solutions:

Weak reader's priority or weak writer's priority.

Weak reader's priority: An arriving reader still has priority over waiting writers. However, when a writer departs, both waiting readers and waiting writers have equal priority.

# The Readers and Writers Problem

```
typedef int semaphore;
semaphore mutex = 1;
semaphore db = 1;
int rc = 0;

void reader(void)
{
    while (TRUE) {
        down(&mutex);
        rc = rc + 1;
        if (rc == 1) down(&db);
        up(&mutex);
        read_data_base();
        down(&mutex);
        rc = rc - 1;
        if (rc == 0) up(&db);
        up(&mutex);
        use_data_read();
    }
}
. . .
```

*/\* use your imagination \*/*  
*/\* controls access to 'rc' \*/*  
*/\* controls access to the database \*/*  
*/\* # of processes reading or wanting to \*/*

*/\* repeat forever \*/*  
*/\* get exclusive access to 'rc' \*/*  
*/\* one reader more now \*/*  
*/\* if this is the first reader ... \*/*  
*/\* release exclusive access to 'rc' \*/*  
*/\* access the data \*/*  
*/\* get exclusive access to 'rc' \*/*  
*/\* one reader fewer now \*/*  
*/\* if this is the last reader ... \*/*  
*/\* release exclusive access to 'rc' \*/*  
*/\* noncritical region \*/*

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

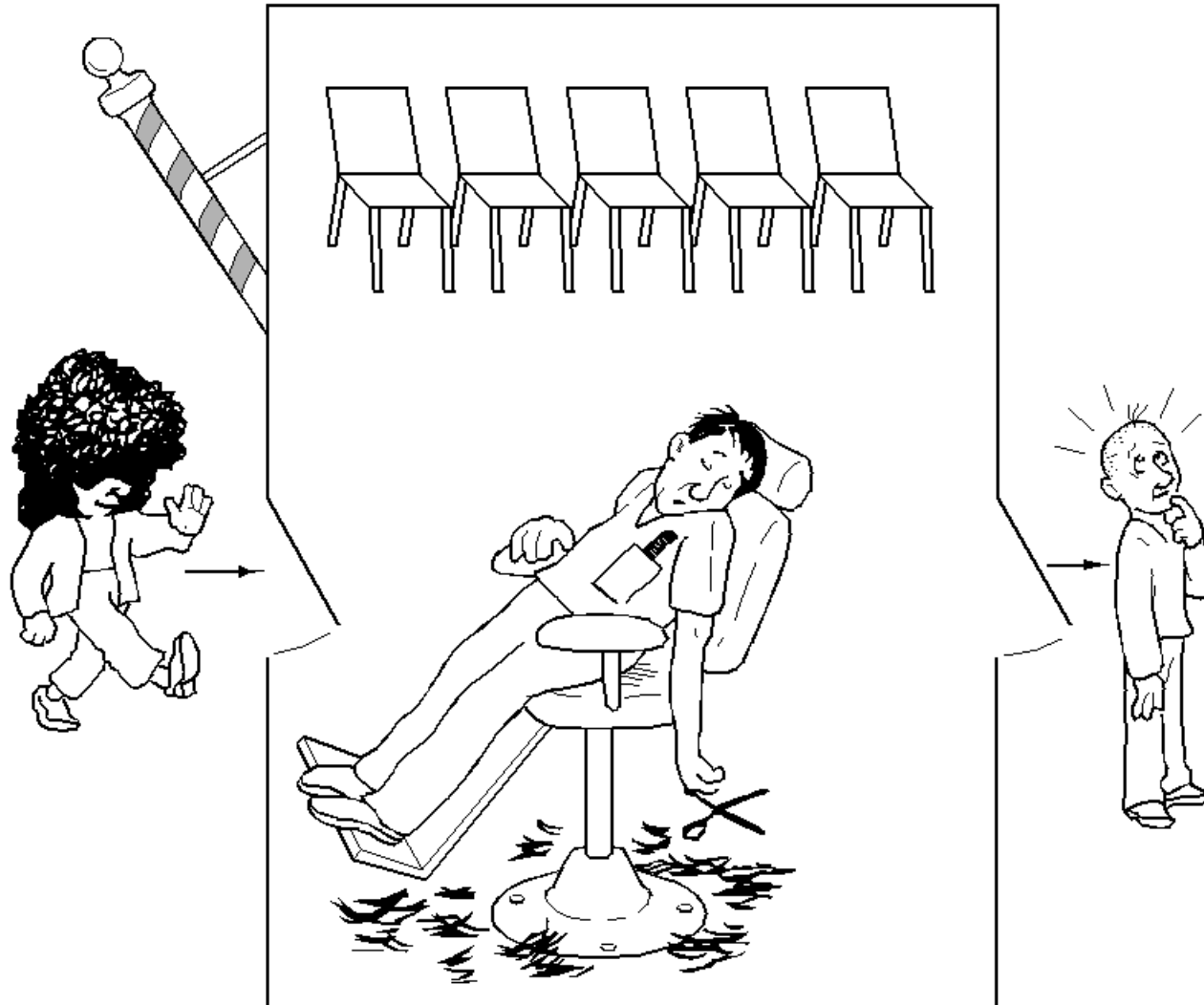
# The Readers and Writers Problem

• • •

```
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 */
    }
}
```

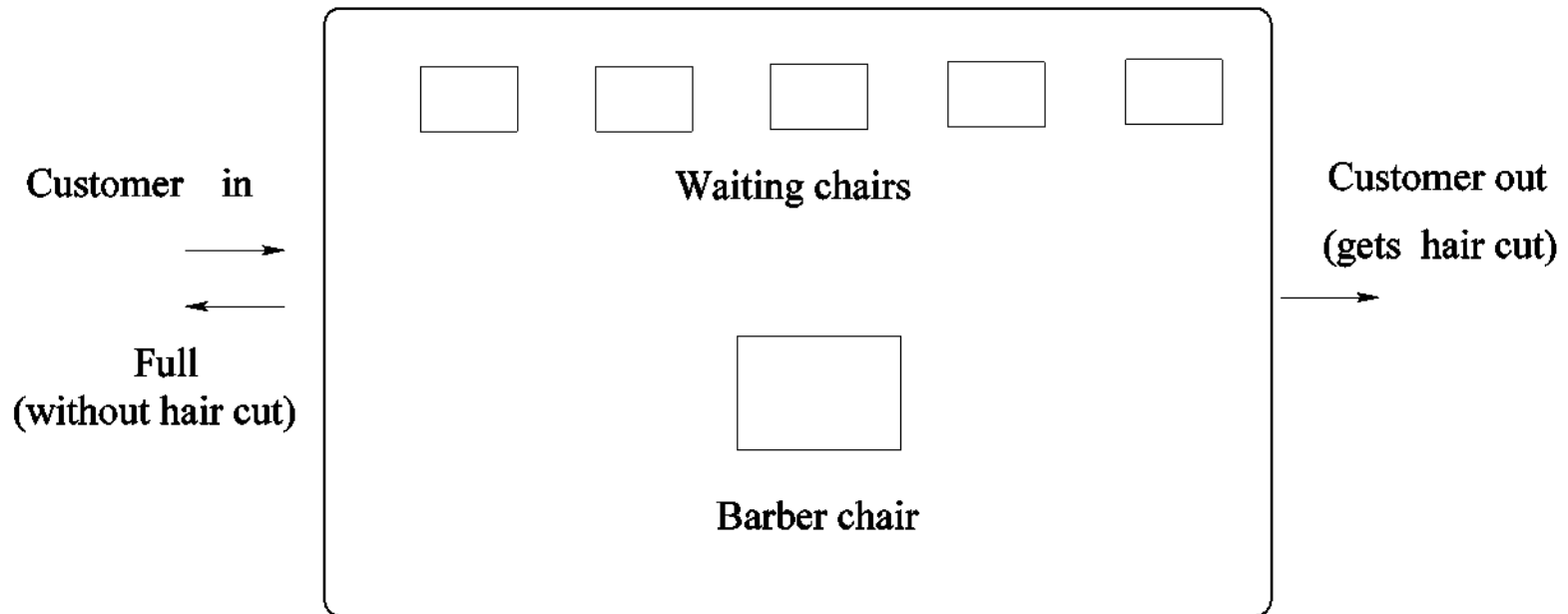
Figure 2-47. A solution to the readers and writers problem.

# The Sleeping Barber Problem



# The Sleeping Barber Problem

Problem description:



When no customer, barber sleeps. If no waiting chair, customer leaves.

# The Sleeping Barber Problem

```
#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.

# CPU Scheduling

Resources: the things operated on by processes.

Resources ranges from CPU time to disk space, to I/O channel time.

Resources fall into two classes:

(1) Preemptive:

OS can take resource away. Use it for something else, and then give it back later.

Examples: processor or I/O channel.

(2) Non-preemptive:

Once given, it cannot be reused until the process gives it back.

Examples: file space and terminal.

Anything is preemptive if it can be saved and restored.

# CPU Scheduling

O.S. makes two related kinds of decisions about resources:

(1) Allocation:

Who gets what? Given a set of requests for resources, which process should be given which resources in order to make most efficient use of the resources?

(2) Scheduling: How long can they keep it? When more resources are requested than can be granted immediately, in which order should they be served?

Examples: processor scheduling and memory scheduling (virtual memory).

Resource # 1: the processor.

Processes may be in any one of three general scheduling states: Running, ready and blocked.



# CPU Scheduling

Criteria for a good scheduling algorithm:

1. Fairness: every process gets its fair share of CPU.
2. Efficiency (utilization): keep CPU busy.
3. Response time: minimize response time for interactive users.
4. Throughput: maximize jobs per hour.
5. Minimize overhead (context swaps).

Context switch: changing process. Include save and load registers and memory maps and update misc tables and lists.

Clock interrupt: Clock interrupt occurs at fixed time interval (for 60 Hertz AC frequency, 60 times per second) and O.S. scheduler can run. Every interrupt is called a clock tick (a basic time unit in computer systems).

# Scheduling – Process Behavior

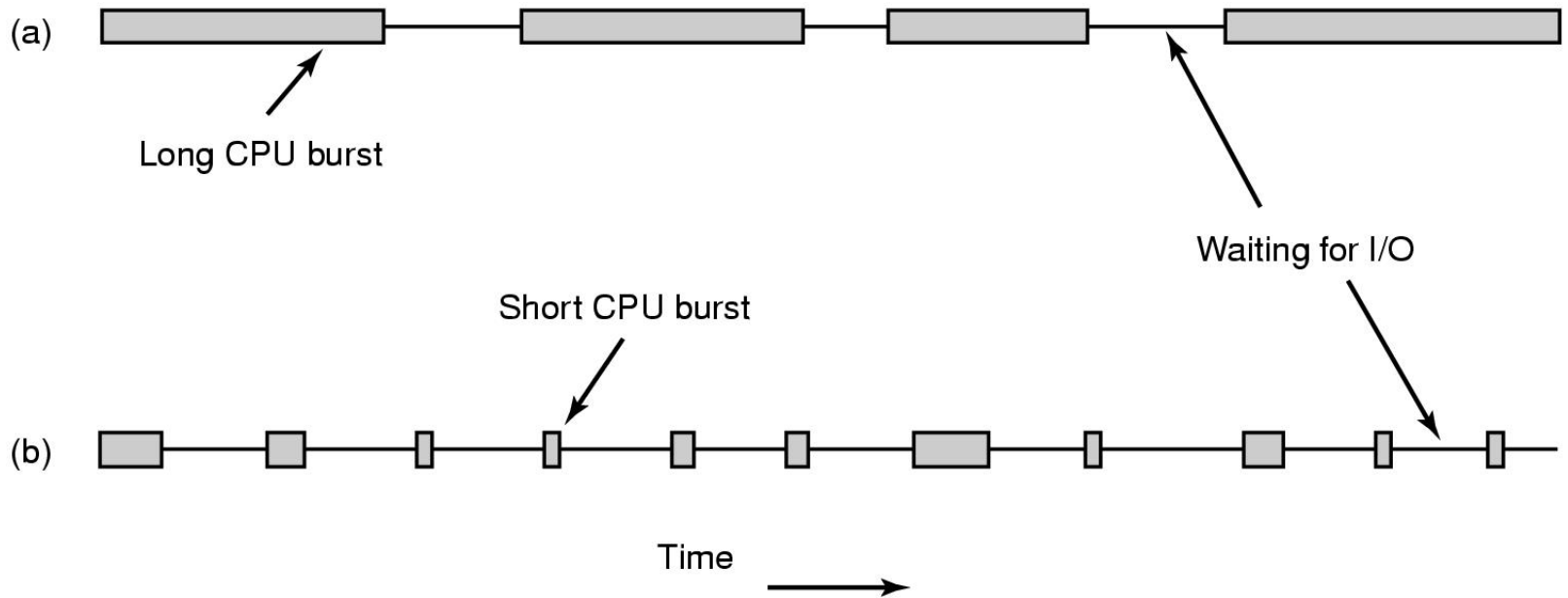


Figure 2-38. Bursts of CPU usage alternate with periods of waiting for I/O. (a) A CPU-bound process. (b) An I/O-bound process.

# Categories of Scheduling Algorithms

- Batch
- Interactive
- Real time

# 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

Figure 2-39. Some goals of the scheduling algorithm under different circumstances.

# Scheduling in Batch Systems

- First-come first-served (FCFS)
- Shortest job first
- Shortest remaining time next

# First IN First Out (FIFO or FCFS)

Run until finished, usually "finished" means "blocked."  
Process goes to the back of run queue when ready.

Problem: one process can dominate the CPU.

Solution: limit the maximum time that a process can run without a context switch. The time is called quantum or time slice.

# Shortest Job First (SJF)

- Suitable to batch system.
- Non-preemptive policy.
- Must know the runtime or estimate runtime of each process.
- All jobs are available at system start-up time.
- Schedule the jobs according to their runtimes.
- Optimal with respect to average turnaround time.

# Shortest Job First

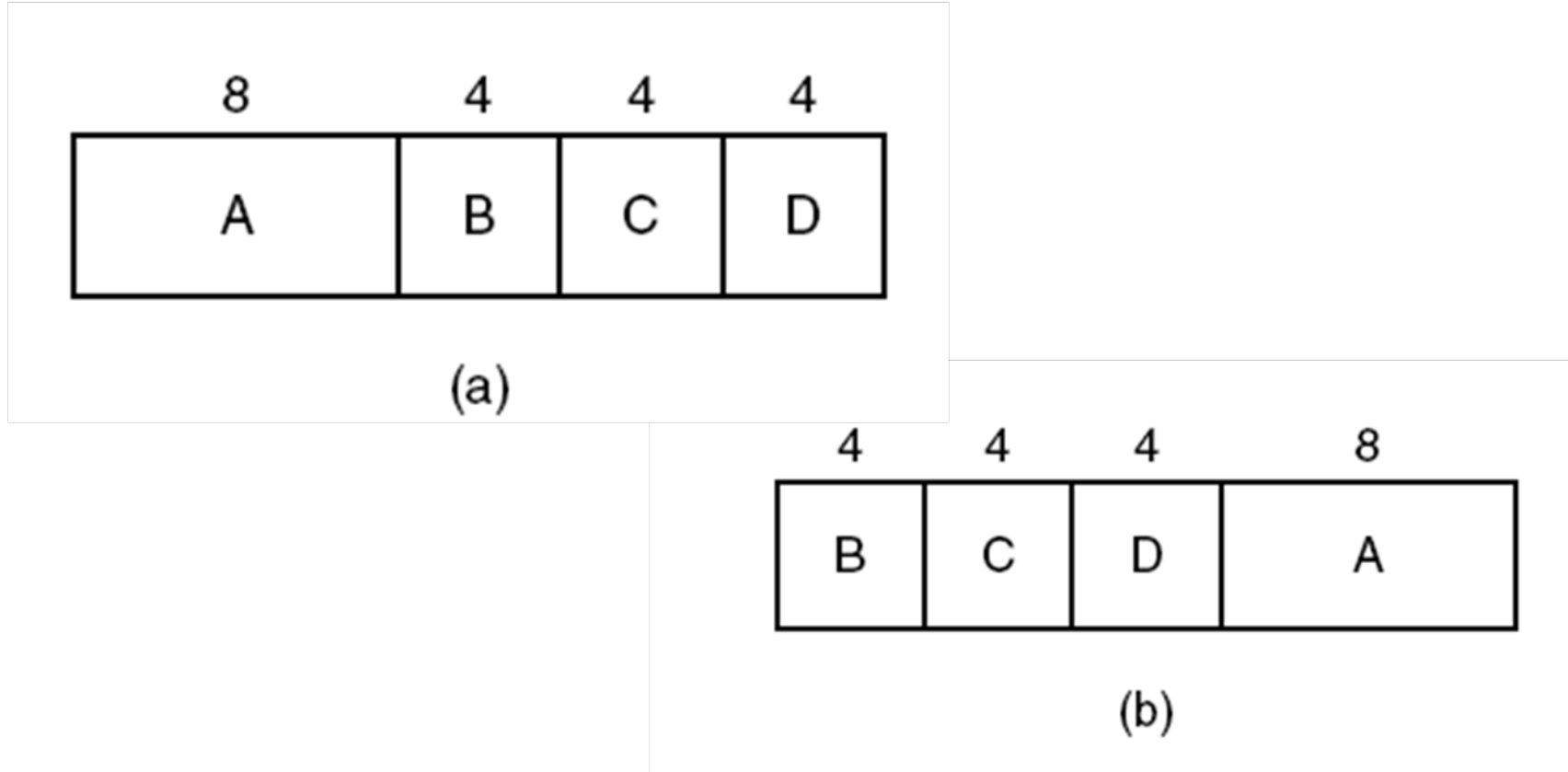


Figure 2-40. An example of shortest job first scheduling.  
(a) Running four jobs in the original order. (b) Running them in shortest job first order.



# Shortest Job First

The runtimes of jobs A B C D (min): 8, 4, 4, 4

1. Run in order A B C D:

Jobs Turnaround times

A: 8

B:  $8 + 4 = 12$

C:  $8 + 4 + 4 = 16$

D:  $8 + 4 + 4 + 4 = 20$

Average Turnaround Time

$ATT = (8+12+16+20)/4 = 14$  (min)

2. Run in the order B C D A:

Jobs Turnaround times

B: 4

C:  $4 + 4 = 8$

D:  $4 + 4 + 4 = 12$

A:  $4 + 4 + 4 + 8 = 20$

Average Turnaround Time

$ATT = (4+8+12+20)/4 = 11$  (min)

# Shortest Job First

Shortest job first achieves the shortest average turnaround time.

General proof:

1. For four jobs.

Suppose the runtimes for jobs A, B, C and D are a; b; c and d respectively.

Run in the order A B C D.

Average Turnaround Time:

$$ATT = [a + (a + b) + (a + b + c) + (a + b + c + d)]/4 = (4a + 3b + 2c + d)/4$$

a has the largest coefficient and must be the smallest. B must be the second smallest,...

# Shortest Job First

2. For  $n$  jobs.

$n$  jobs  $J_1 J_2 \dots J_n$

Runtimes  $T_1, T_2, \dots, T_n$

Run in order  $J_1; J_2; \dots, J_n$ .

Jobs Turnaround times

$J_1: T_1$

$J_2: T_1 + T_2$

$J_3: T_1 + T_2 + T_3$

...

$J_i: T_1 + T_2 + \dots + T_i$

...

$J_n: T_1 + T_2 + \dots + T_n$

Average Turnaround Time:

$$ATT = [nT_1 + (n-1)T_2 + (n-2)T_3 + \dots + (n-i+1)T_i + \dots + 2T_{n-1} + T_n]/n$$

# Shortest Job First

We want to prove:

if  $T_1 \leq T_2 \leq \dots \leq T_n$ , then ATT is optimal (smallest).

Prove it by contradiction.

Suppose for some  $1 \leq i < j \leq n$ , we have  $T_i > T_j$ .

Note that we have  $(n - i + 1)T_i$  and  $(n - j + 1)T_j$  in ATT, but  $n - i + 1 > n - j + 1$ .

We can always reduce the ATT by exchanging the running order of  $J_i$  and  $J_j$ . That is,

$$(n-i+1)T_j + (n-j+1)T_i < (n-i+1)T_i + (n-j+1)T_j$$

Thus, shortest job first is optimal.

# Shortest Job First

If jobs are not available at the beginning, the shortest job first may not be optimal.

A counterexample.

Five jobs: A B C D E

Runtimes: 2, 4, 1, 1, 1

Arrive times: 0, 0, 3, 3, 3

•Run in order A B C D E (shortest job first).

Jobs Turnaround times

A: 2

B:  $2 + 4 = 6$

C:  $6 - 3 + 1 = 4$

D:  $4 + 1 = 5$

E:  $5 + 1 = 6$

ATT =  $(2+6+4+5+6)/5 = 23/5$

# Shortest Job First

Run in order B C D E A.

Jobs Turnaround times

B: 4

C:  $4 - 3 + 1 = 2$

D:  $2 + 1 = 3$

E:  $3 + 1 = 4$

A:  $4 + 3 + 2 = 9$

$ATT = (4+2+3+4+9)/5 = 22/5$

# Shortest Remaining Time First

- A preemptive version of shortest job first
- Scheduler always chooses the process whose remaining run time is the shortest.
- Allows new short jobs to get good service

# Scheduling in Interactive Systems

- Round-robin scheduling
- Priority scheduling
- Multiple queues
- Shortest process next
- Guaranteed scheduling



# Round-Robin Scheduling

- Maintain a list of runnable processes.
- Run a process for one quantum then move to the back of queue. Each process gets equal share of the CPU.
- If process blocks (say, for I/O or semaphore) then remove it from the queue and start to run the next process in queue.
- If a process becomes runnable, add it to the end of queue.
- Length of quantum:
  - Short: too much overhead
  - Long: poor response time
- In general: about 100 ms (or about 10K -100K instructions)

# 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.

# Round-Robin Scheduling

Round robin may produce bad results:

Example: Ten processes, each requires 100 quantum.

In round robin: each takes about 1000 ( $10 \times 100$ ) quantum to finish.

In FIFO, they would require average 500 quantum to finish.

How can we minimize the average response time (or turnaround time)?

Use shortest job first.

# Priority Scheduling

Assign each process a priority. Run the runnable process with the highest priority.

How to assign priority:

I/O bound jobs have higher priority

CPU bound jobs have lower priority

If a job uses  $1/f$  of the quantum, then  $\text{Priority} = f$ .

Unix command "nice" allows a user to lower the job priority voluntarily.

Problem: high priority job may dominate CPU.

Solution: decrease priority of the running process at each clock tick (dynamic priority).

# Priority Class Scheduling (Multiple Queues)

Combine round robin and priority.

Group processes into priority classes.

Use priority scheduling among the classes.

Use round robin within each class.

Classes may have different quantum.

Adaptively change quantum: exponential queue

Give a newly runnable process a high priority and a very short quantum.

If the process uses up the quantum without blocking then decrease priority by 1 and double quantum for next time.

# Priority Class Scheduling

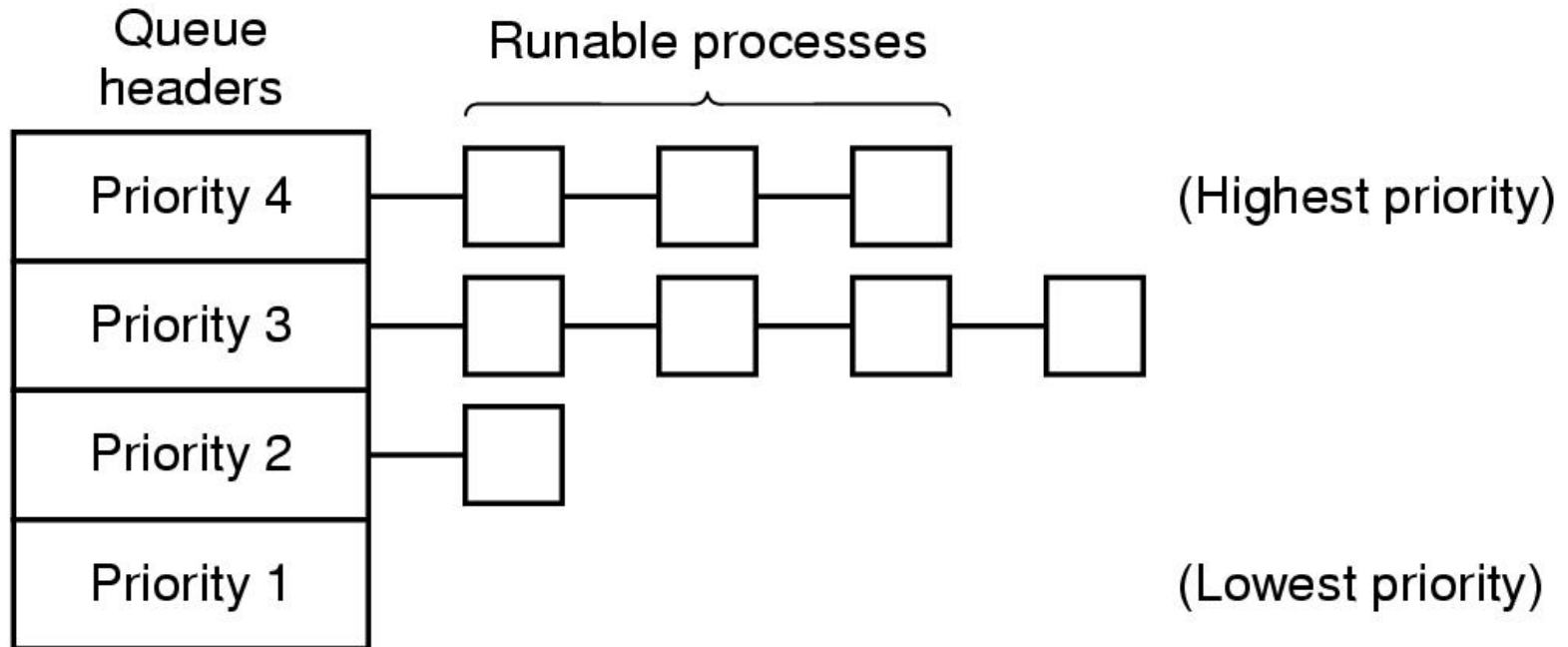


Figure 2-42. A scheduling algorithm with four priority classes.

# Priority Class Scheduling

Example:

Two processes P1 and P2.

P1: doing 1 ms computation followed by 10 ms I/O.

P2: doing all computation.

Initial quantum = 100 ms.

- P1: priority 100 uses 1 ms CPU, blocked for 10 ms and then becomes ready again.
- P2: priority 100 uses 100 ms CPU, switched out.
- P1: priority 100 uses 1 ms CPU, blocked for 10 ms and then becomes ready again.
- P2: priority 99 uses 200 ms CPU, switched out.

...

# Shortest Process Next (Aging Algorithm)

How to use shortest job first in an interactive system?

Consider each command as a job

Estimate the runtime for a command (job) based on past runtime.

$T_0$ : estimated runtime per command for some terminal

$T_i$ : runtime for the  $i$ th command ( $i \geq 1$ )

$S_i$ : predicted runtime for the  $i$ th ( $i \geq 1$ ) command

Recurrence:

$$S_1 = T_0$$

$$S_{n+1} = aT_n + (1-a)S_n; 0 \leq a \leq 1; n > 1$$

Closed form:

$$S_{n+1} = aT_n + (1-a)aT_{n-1} + \dots + (1-a)^i aT_{n-i} + \dots + (1-a)^{n-1} aT_1 + (1-a)^n T_0$$



# Shortest Process Next (Aging Algorithm)

Let  $a = \frac{1}{2}$

$$S_{n+1} = \frac{1}{2} T_n + \frac{1}{2^2} T_{n-1} + \dots + \frac{1}{2^{i+1}} T_{n-i} + \dots + \frac{1}{2^n} T_1 + \frac{1}{2^n} T_0$$

Aging algorithm:

Estimate the next value in a series by taking the weighted average of the current measured value and previous estimate.

Possibility of starvation of longer jobs.

# Guaranteed Scheduling

Also called fair share scheduling

n users logged in, each user receives about  $1/n$  of the CPU time.

(1) Keep track of:

- How long each user logged in
- How much time a user used

(2) Compute:

CPU time entitled for a user = logged in time/n

Ratio = used time/ entitled time

(3) Choose the lowest ratio process to run until its ratio has moved above its closest competitor.

# Two Level Scheduling

Lower-level: scheduling among the processes in memory

Higher-level: scheduling between disk and memory

Criteria for higher-level scheduling:

1. Time the process stays in memory
2. CPU time of the process
3. Size of the process
4. Priority of the process

# Scheduling in Real-Time Systems

Real-Time Systems:

- Hard real-time systems: The deadlines must be met.
- Soft real-time systems: The deadlines should be met most of the time.

Periodic events: Events occur at regular intervals

Aperiodic events: Events occur unpredictably

# 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

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

A real-time system that meets this criteria is said **schedulable**.

# Scheduling in Real-Time Systems

Example:

$$P_1 = 100, P_2 = 200, P_3 = 500, P_4 = 1000$$

$$C_1 = 50, C_2 = 30, C_3 = 100, C_4 = ?$$

$$50/100 + 30/200 + 100/500 + C_4/1000 \leq 1$$

$$C_4 \leq 150$$

# Thread Scheduling (1)

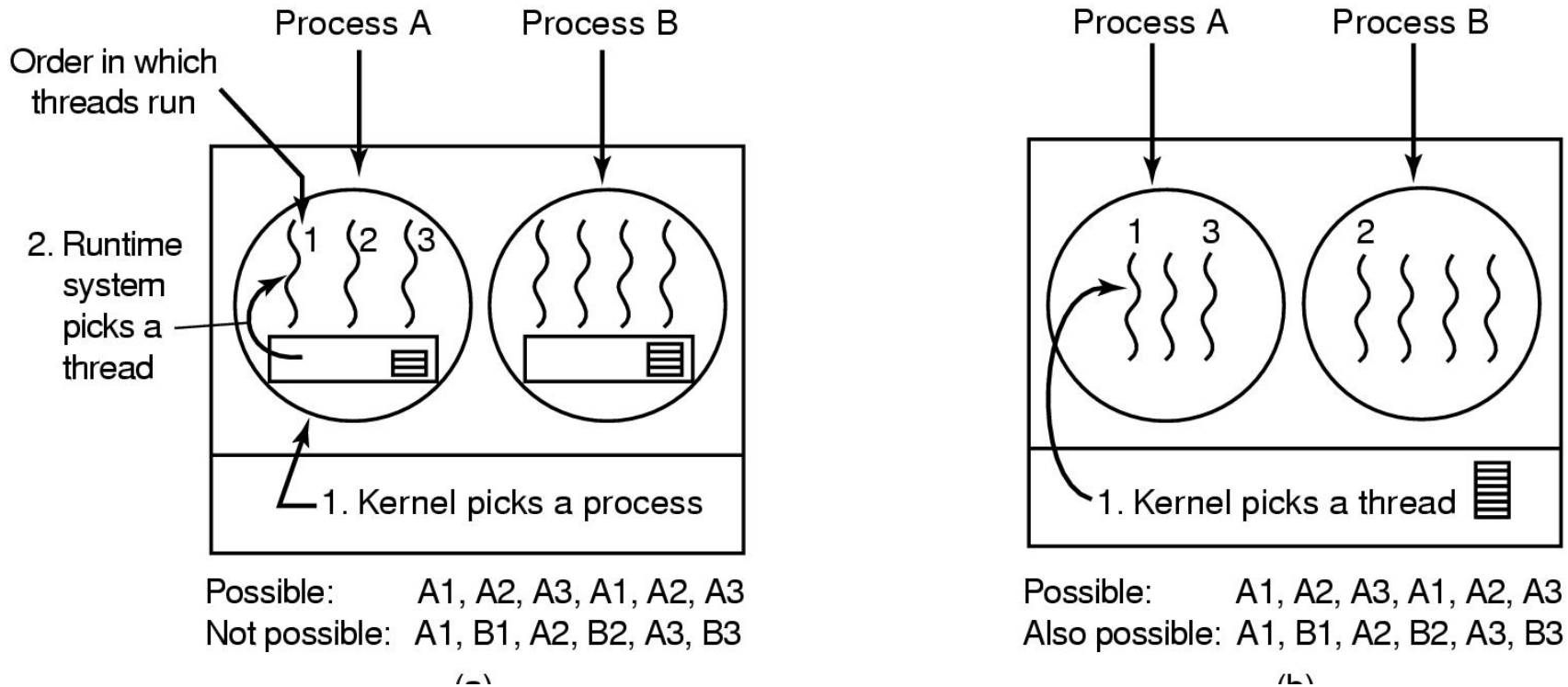
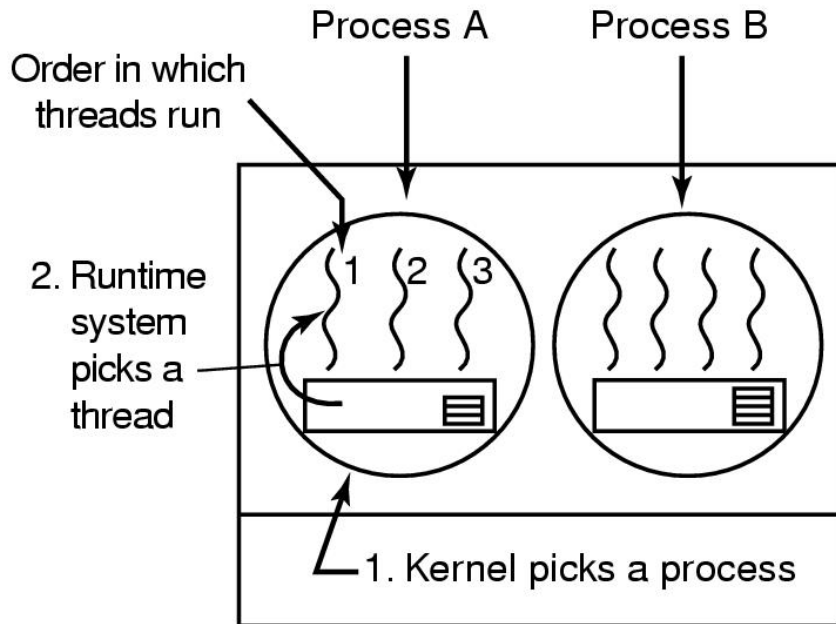
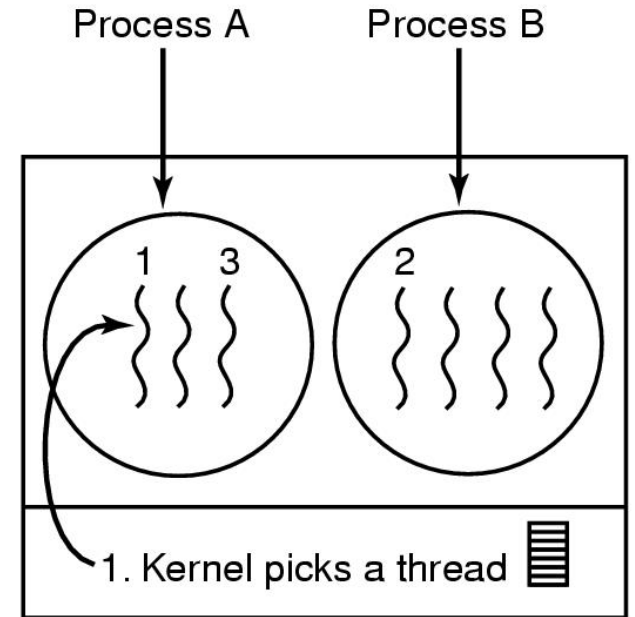


Figure 2-43. (a) Possible scheduling of user-level threads with a 50-msec process quantum and threads that run 5 msec per CPU burst.

# Thread Scheduling (2)



Possible: A1, A2, A3, A1, A2, A3  
 Not possible: A1, B1, A2, B2, A3, B3



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

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



# Summary of Scheduling Algorithms

In principle, scheduling algorithm can be arbitrary, since the system should produce the same results in any event (get the job done).

However, the algorithms have strong effects on the system's overhead, efficiency and response time.

The best schemes are adaptive. To do absolutely best, we have to be able to predict the future.

Best scheduling algorithms tend to give highest priority to the process that needs the least!