Module 6: Process Synchronization

- Background
- The Critical-Section Problem
- Synchronization Hardware
- Semaphores
- Classical Problems of Synchronization
- Critical Regions
- Monitors
- Synchronization in Solaris 2
- Atomic Transactions

Quiz

Which of the following scheduling algorithms could result in starvation?

a. First-come, first-served

b. Shortest job first

c. Round robin

d. Priority

Background

- Concurrent access to shared data may result in data inconsistency.
- Maintaining data consistency requires mechanisms to ensure the orderly execution of cooperating processes.
- Shared-memory solution to bounded-buffer problem (Chapter 4) allows at most $n - 1$ items in buffer at the same time. A solution, where all N buffers are used is not simple.
  - Suppose that we modify the producer-consumer code by adding a variable *counter*, initialized to 0 and incremented each time a new item is added to the buffer

Bounded-Buffer

- Shared data

```
type item = … ;
var buffer array [0..n-1] of item;
in, out: 0..n-1;
counter: 0..n;
in, out, counter := 0;
```

- Producer process

```
repeat
    …
    produce an item in nextp
    …
    while counter = n do no-op;
    buffer [in] := nextp;
    in := in + 1 mod n;
    counter := counter +1;
until false;
```

Bounded-Buffer (Cont.)

- Consumer process

```
repeat
    while counter = 0 do no-op;
    nextc := buffer [out];
    out := out + 1 mod n;
    counter := counter – 1;
        …
    consume the item in nextc
        …
until false;
```

- The statements:
  - *counter* := *counter* + 1;
  - *counter* := *counter* - 1;

  must be executed *atomically*.

Bounded-Buffer (Cont.)

T_0:	*producer*	execute	$register_1$ = `counter`	$\{register_1 = 5\}$
T_1:	*producer*	execute	$register_1 = register_1 + 1$	$\{register_1 = 6\}$
T_2:	*consumer*	execute	$register_2$ = `counter`	$\{register_2 = 5\}$
T_3:	*consumer*	execute	$register_2 = register_2 - 1$	$\{register_2 = 4\}$
T_4:	*producer*	execute	`counter` = $register_1$	$\{counter = 6\}$
T_5:	*consumer*	execute	`counter` = $register_2$	$\{counter = 4\}$

The Critical-Section Problem

- n processes all competing to use some shared data
- Each process has a code segment, called *critical section*, in which the shared data is accessed.
- Problem – ensure that when one process is executing in its critical section, no other process is allowed to execute in its critical section.
- Structure of process P_i

repeat

entry section

critical section

exit section

reminder section

until *false*;

Solution to Critical-Section Problem

1. **Mutual Exclusion**. If process *Pi* is executing in its critical section, then no other processes can be executing in their critical sections.
2. **Progress**. If no process is executing in its critical section and there exist some processes that wish to enter their critical section, then the selection of the processes that will enter the critical section next cannot be postponed indefinitely.
3. **Bounded Waiting**. A bound must exist on the number of times that other processes are allowed to enter their critical sections after a process has made a request to enter its critical section and before that request is granted.
   - Assume that each process executes at a nonzero speed
   - No assumption concerning relative speed of the *n* processes.

Initial Attempts to Solve Problem

- Only 2 processes, P_0 and P_1
- General structure of process P_i (other process P_j)

repeat

entry section

critical section

exit section

reminder section

until *false*;

- Processes may share some common variables to synchronize their actions.

Algorithm 1

- Shared variables:
  - **var** *turn*: (0..1);
    initially *turn* = 0
  - *turn* - *i* $\Rightarrow$ P_i can enter its critical section
- Process P_i

repeat

while *turn* $\neq$ *i* **do** *no-op*;

critical section

turn := *j*;

reminder section

until *false*;

- Satisfies mutual exclusion, but not progress
- Taking turns is not a good idea when one of the processes is much slower than the other. Violate the progress condition.

Algorithm 2

- Shared variables
  - **var** *flag*: **array** [0..1] **of** *boolean*;
    initially *flag* [0] = *flag* [1] = *false*.
  - *flag* [*i*] = *true* $\Rightarrow$ P_i ready to enter its critical section
- Process P_i

repeat

flag[*i*] := *true*;
while *flag*[*j*] **do** *no-op*;

critical section

flag [*i*] := *false*;

remainder section

until *false*;

- Satisfies mutual exclusion, but not progress requirement.

Algorithm 2 (Cont.)

In this solution, the mutual-exclusion requirement is satisfied. Unfortunately, the progress requirement is not met. To illustrate this problem, we consider the following execution sequence:

T_0: P_0 sets `flag[0] = true`
T_1: P_1 sets `flag[1] = true`

Now P_0 and P_1 are looping forever in their respective `while` statements.

Algorithm 3

- Combined shared variables of algorithms 1 and 2.
- Process P_i

repeat

flag [*i*] := *true*;
turn := *j*;
while (*flag* [*j*] and *turn* = *j*) **do** *no-op*;

critical section

flag [*i*] := *false*;

remainder section

until *false*;

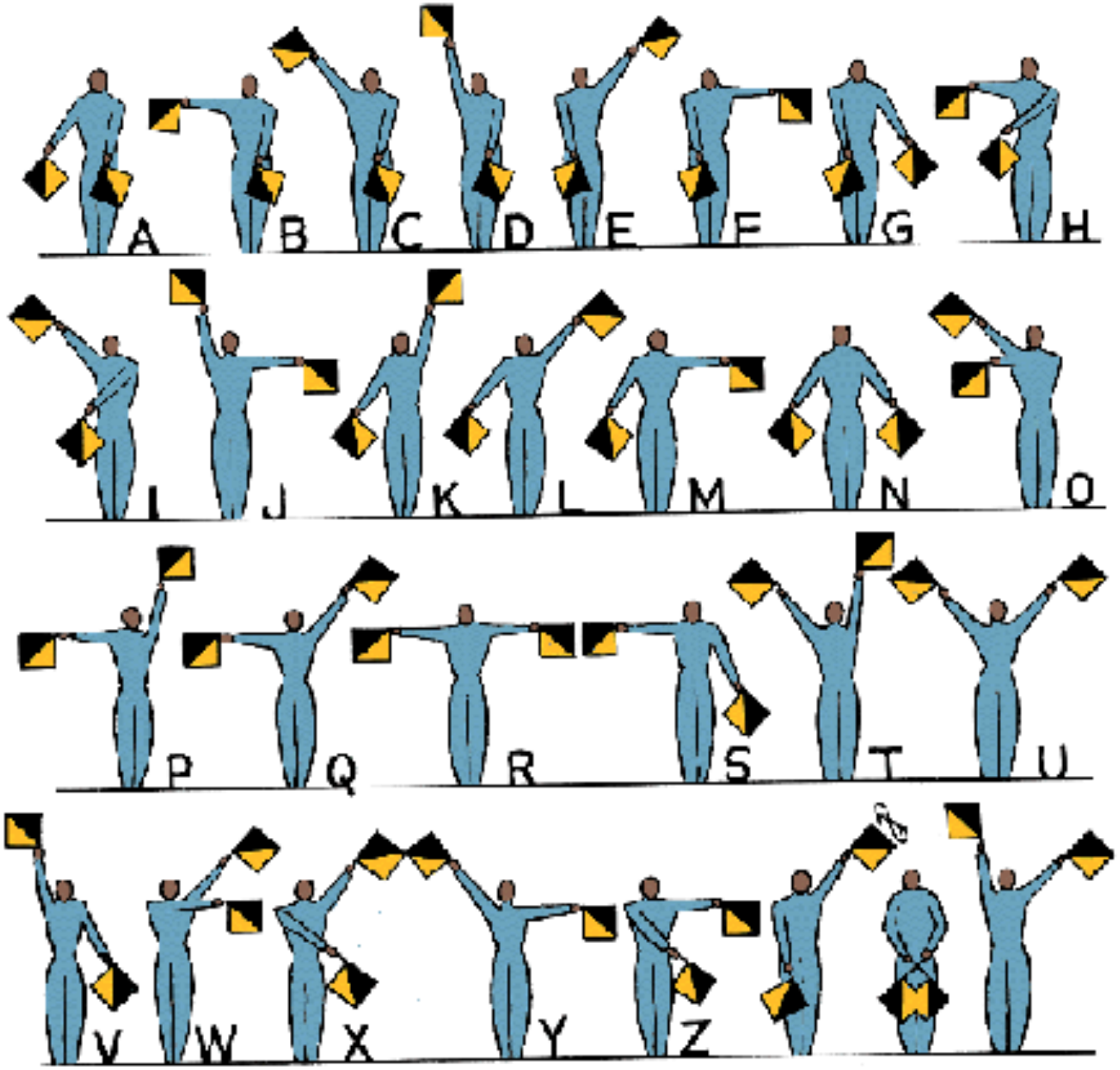
- Meets all three requirements; solves the critical-section problem for two processes.

Semaphore

- Synchronization tool that does not require busy waiting.
- Semaphore S – integer variable
- can only be accessed via two indivisible (atomic) operations

wait (S): **while** $S \leq 0$ **do** *no-op*;
$S := S - 1$;

signal (S): $S := S + 1$;


A
B
C
D
E
F
G
H
I
J
K
L
M
N
O
P
Q
R
S
T
U
V
W
X
Y
Z

Algorithm for Process Pi

```
do {
    acquire lock
        critical section
    release lock
        remainder section
}
```

6.5 Semaphore

- It's a hardware based solution
- Semaphore *S* –integer variable

- Two standard operations modify S: wait() and signal()

6.5 Semaphore

Can only be accessed via two indivisible (atomic) operations

```
wait (S) {
    while S <= 0
        ; // no-op
    S--;
}

signal (S) {
    S++;
}
```

6.5 Semaphore

- Binary semaphore –integer value can range only between 0 and 1; can be simpler to implement

- Counting semaphore –integer value can range over an unrestricted domain

6.5 Semaphore

Provides mutual exclusion

Semaphore S; // initialized to 1

```
do {

wait (S);
  //Critical Section
signal (S);
  //Remainder Section

} while (true)
```

6.5 Semaphore

- Must guarantee that no two processes can execute wait ()and signal ()on the same semaphore at the same time

- Atomic = non-interruptable

Semaphore as General Synchronization Tool

- Execute *B* in P_j only after *A* executed in P_i
- Use semaphore *flag* initialized to 0
- Code:

P_i	P_j
⋮	⋮
A	*wait*(*flag*)
signal(*flag*)	*B*

6.5 Semaphore

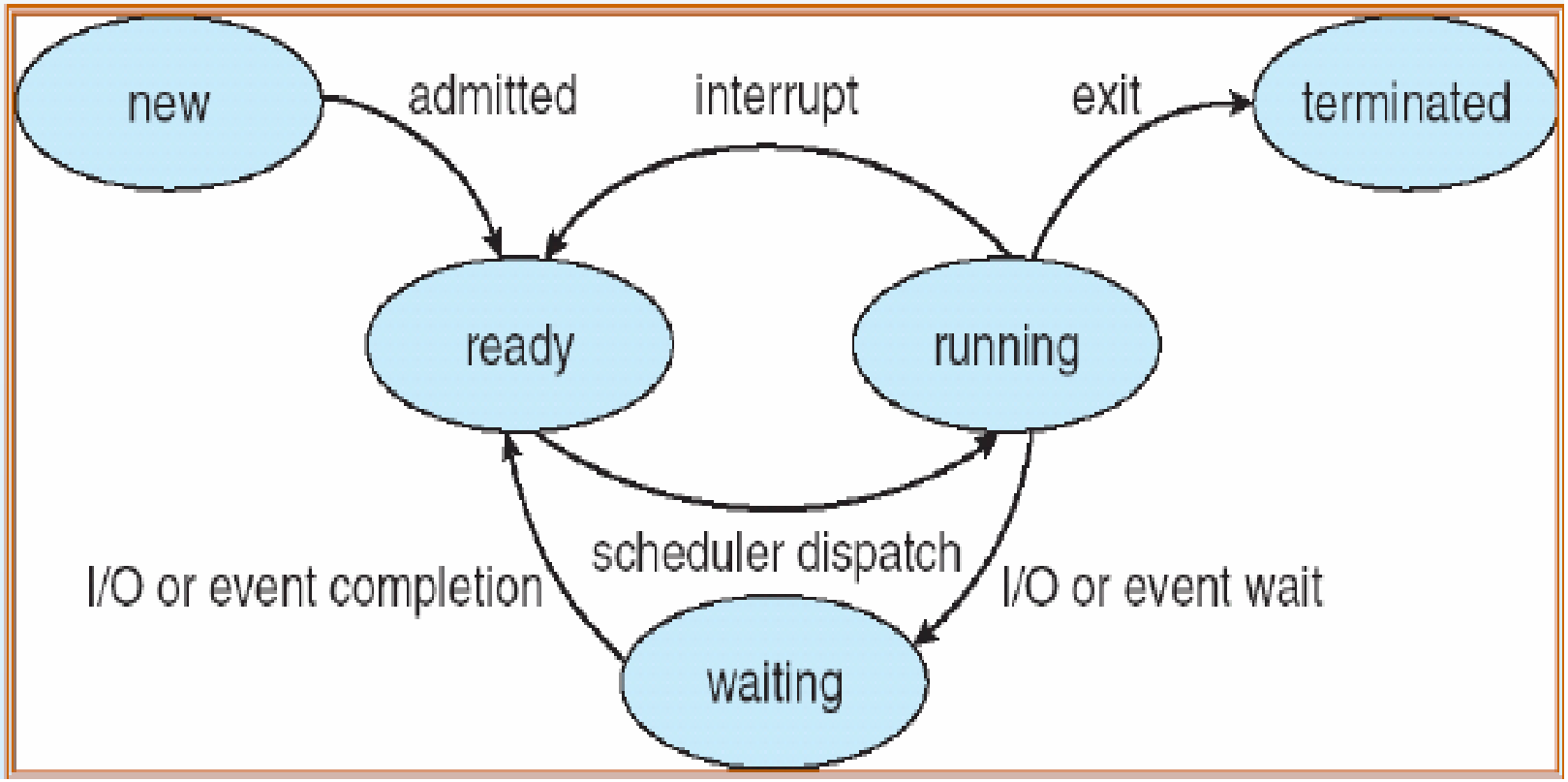
- The main disadvantage of the semaphore is that it requires **busy waiting**, which wastes CPU cycle that some other process might be able to use productively
- This type of semaphore is also called a **spinlock** because the process "spins" while waiting for the lock

6.5 Semaphore

To overcome the busy waiting problem, we create two more operations:

- block–place the process invoking the operation on the appropriate waiting queue.
- wakeup –remove one of processes in the waiting queue and place it in the ready queue.

Diagram of Process State



Semaphore Implementation with no Busy waiting

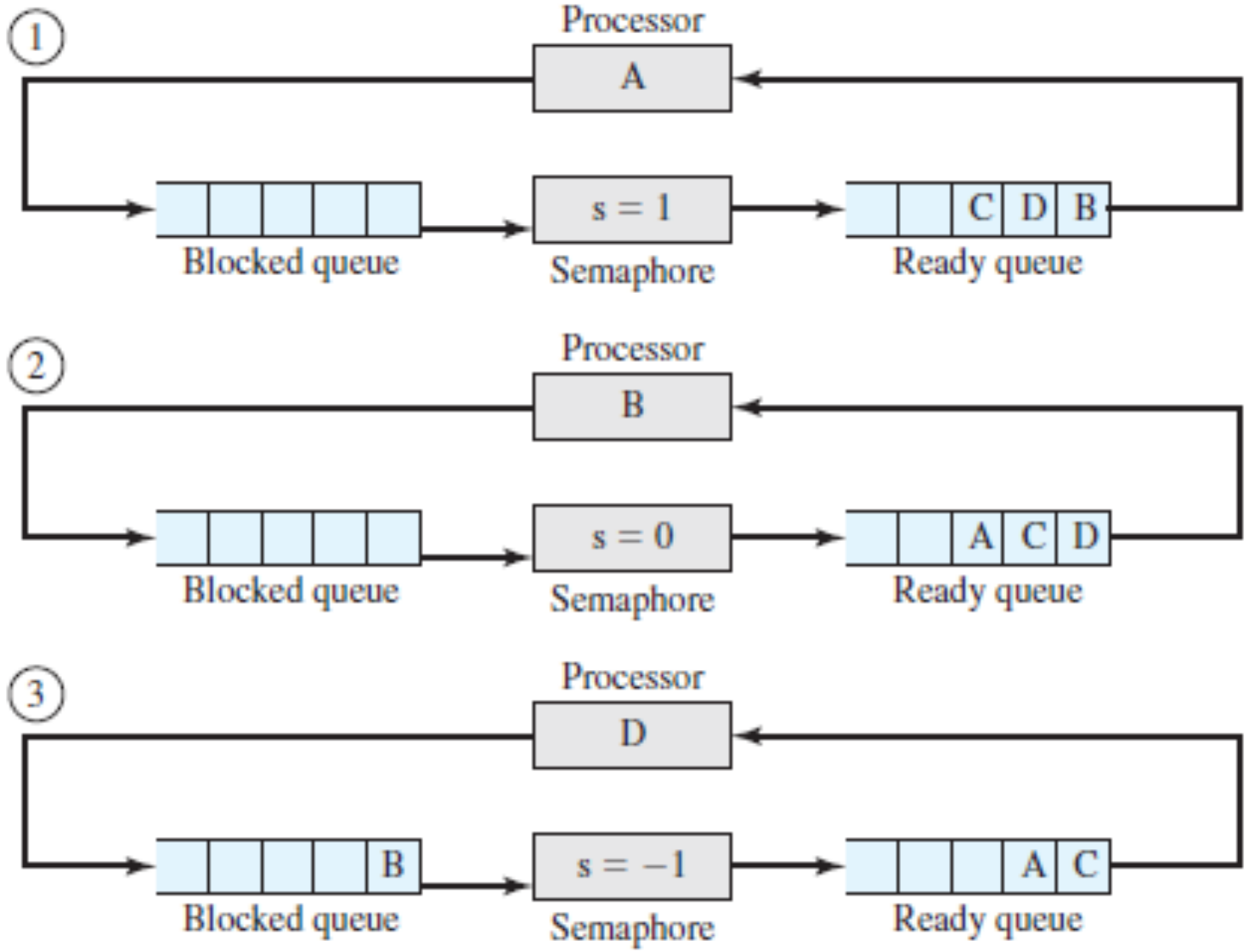
- Implementation of wait:

```
wait (S){
        value--;
        if (value < 0) {
                add this process to waiting queue
                 block();  }
 }
```

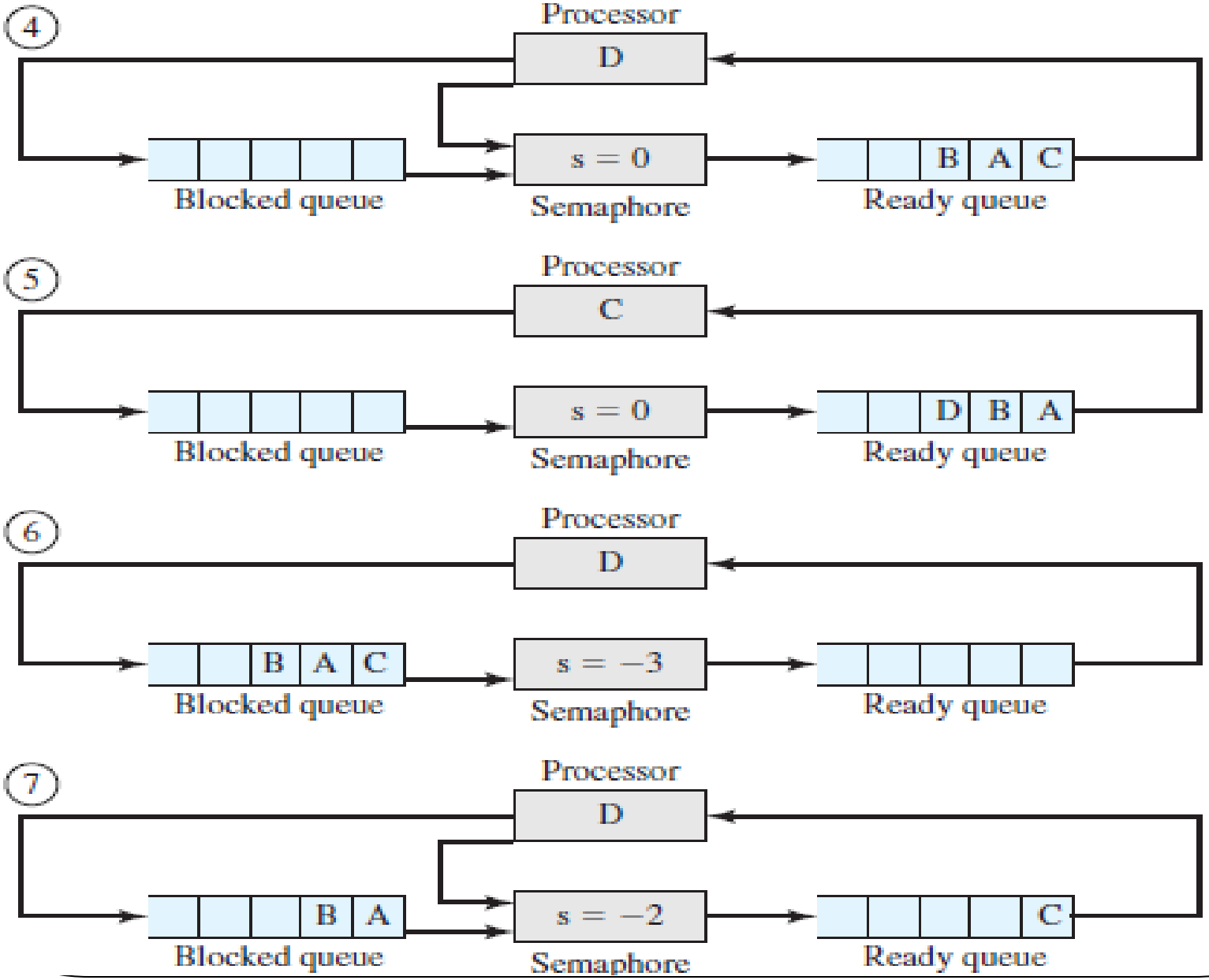
- Implementation of signal:

```
Signal (S){
          value++;
           if (value <= 0) {
                   remove a process P from the waiting queue
                    wakeup(P);  }
}
```

Example


1
Processor
A
s = 1
C D B
Blocked queue
Semaphore
Ready queue
2
Processor
B
s = 0
A C D
Blocked queue
Semaphore
Ready queue
3
Processor
D
B
s = −1
A C
Blocked queue
Semaphore
Ready queue

Example(Cont.)



Deadlock and Starvation

- Deadlock –two or more processes are waiting indefinitely for an event that can be caused by only one of the waiting processes

- Starvation–indefinite blocking. A process may never be removed from the semaphore queue in which it is suspended.

Two Types of Semaphores

- *Counting* semaphore – integer value can range over an unrestricted domain.
- *Binary* semaphore – integer value can range only between 0 and 1; can be simpler to implement.
- Can implement a counting semaphore *S* as a binary semaphore.

- Observations
  - Sem value is negative --> Number of waiters on queue
  - Sem value is positive --> Number of threads that can be in c.s. at same time

Critical Section: Review

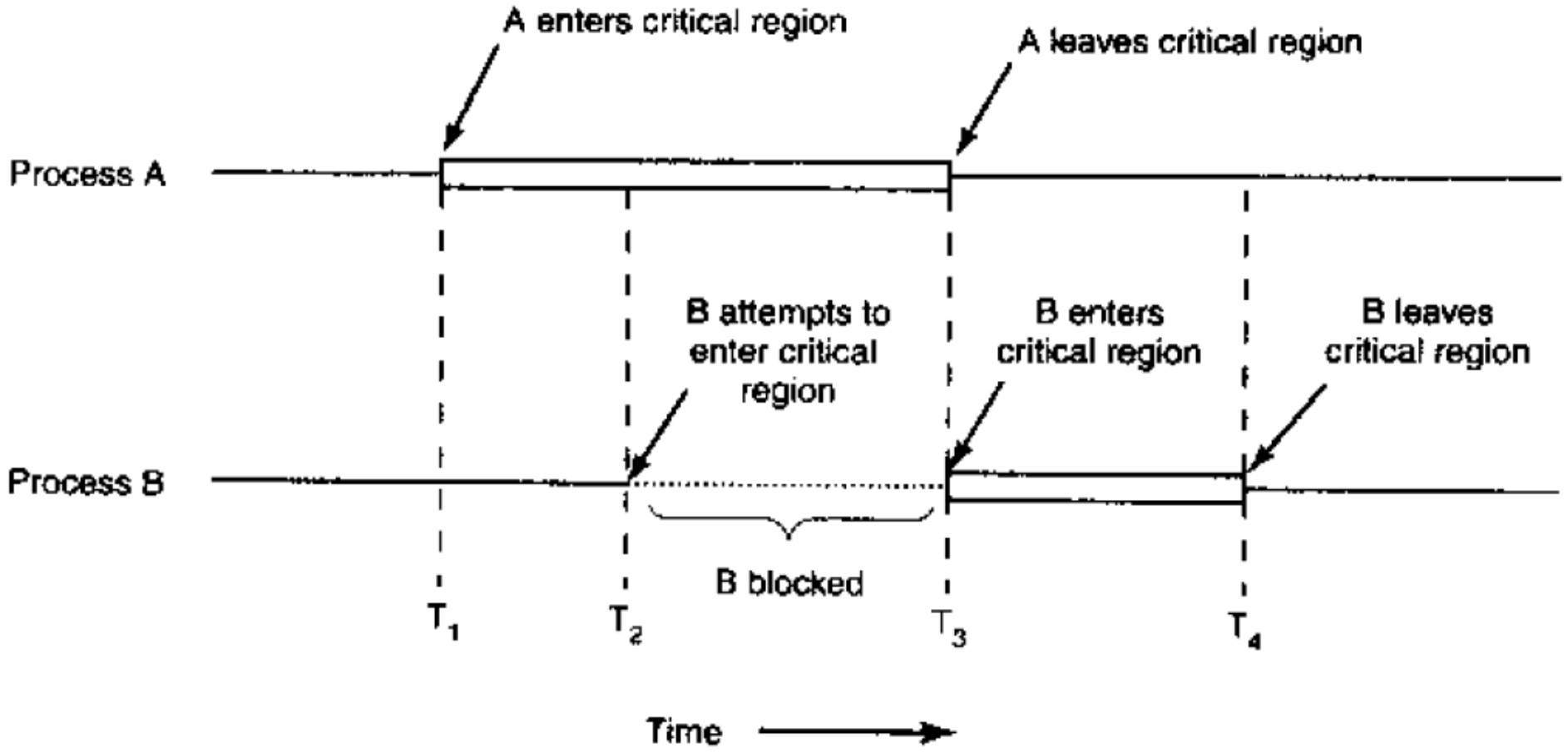


Figure 2-19. Mutual exclusion using critical regions.

Disabling Interrupts

- The simplest solution is to have each process disable all interrupts just after entering its critical region and re-enable them just before leaving it.
- The CPU is only switched from process to process as a result of clock or other interrupts. Without interrupts, the CPU will not be switched to another process.
- It is unwise to give user processes the power to turn off interrupts. What if the user is never turned them on again? That could be the end of the system.

Disabling Interrupts…

- If the system is a multiprocessor, with two or more CPUs, disabling interrupts affects only the CPU that executed the disable instruction. The other CPUs will continue running and can access shared memory.
- The possibility of achieving mutual exclusion by disabling interrupts is becoming less everyday due to the increasing number of multicore chips.
- The conclusion is: disabling interrupt is often a useful technique within the operating system itself but is not appropriate as a general mutual exclusion mechanism for user processes.

Synchronization Hardware

- Test and modify the content of a word atomically.

```
function Test-and-Set (var target: boolean): boolean;
  begin
    Test-and-Set := target;
    target := true;
  end;
```

(Test and Set Lock) that works as follows. It reads the contents of the memory word *lock* into register RX and then stores a nonzero value at the memory address *lock*. The operations of reading the word and storing into it are guaranteed to be indivisible—no other processor can access the memory word until the instruction is finished. The CPU executing the TSL instruction locks the memory bus to prohibit other CPUs from accessing memory until it is done.

Mutual Exclusion with Test-and-Set

- Shared data: **var** *lock*: *boolean* (*initially false*)
- Process P_i

repeat

while *Test-and-Set* (*lock*) **do** *no-op*;

critical section

lock := *false*;

remainder section

until *false*;

To use the TSL instruction, we will use a shared variable, *lock*, to coordinate access to shared memory. When *lock* is 0, any process may set it to 1 using the TSL instruction and then read or write the shared memory. When it is done, the process sets *lock* back to 0 using an ordinary move instruction.

Mutual Exclusion with Test-and-Set (Cont.)

```
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
```

Figure 2-22. Entering and leaving a critical region using the TSL instruction.

Mutual Exclusion with Test-and-Set – Do a trace

```
enter_region:
        TSL REGISTER,LOCK
        CMP REGISTER,#0
        JNE enter_region
        RET | return to caller; critica

leave_region:
        MOVE LOCK,#0
        RET | return to caller
```

```
enter_region:
        TSL REGISTER,LOCK
        CMP REGISTER,#0
        JNE enter_region
        RET | return to caller; critica

leave_region:
        MOVE LOCK,#0
        RET | return to caller
```

Classical Problems of Synchronization

- Bounded-Buffer Problem
- Readers and Writers Problem

Bounded-Buffer Problem

- Shared data

type *item* = …

var *buffer* = …

full, empty, mutex: semaphore;

nextp, nextc: item;

full :=0; *empty* := *n*; *mutex* :=1;

Bounded-Buffer Problem (Cont.)

- Producer process

```
repeat
        …
    produce an item in nextp
        …
    wait(empty);        // decrement empty count
    wait(mutex);        // prepare to enter critical region
        …
    signal(mutex);      // exit critical region
    signal(full);       // increment count of full slots
until false;
```

Bounded-Buffer Problem (Cont.)

- Consumer process

```
repeat
    wait(full)          // decrement full count
    wait(mutex);        // prepare to enter critical region
        …
    remove an item from buffer to nextc
        …
    signal(mutex);      // exit critical region
    signal(empty);      // increment count of empty slots
        …
    consume the item in nextc
        …
until false;
```

Bounded-Buffer Problem (Cont.) –Do a trace

- Producer process

```
repeat
    …
    produce an item in nextp
    …
    wait(empty);
    wait(mutex);

      …
    signal(mutex);
    signal(full);
until false;
```

- Consumer process

```
repeat
    wait(full)
    wait(mutex);
    …
    remove an item from buffer to nextc
     …
    signal(mutex);
    signal(empty);
    consume the item in nextc
     …
until false;
```

Readers-Writers Problem

- The readers/writers problem is defined as follows:
  - There is a data area shared among a number of processes.
  - The data area could be a file, a block of main memory, or even a bank of processor registers.
  - There are a number of processes that only read the data area (readers) and a number that only write to the data area (writers).

Readers-Writers Problem…

- The conditions that must be satisfied are as follows:
  - Any number of readers may simultaneously read the file.
  - Only one writer at a time may write to the file.
  - If a writer is writing to the file, no reader may read it.

Readers-Writers Problem…

- Thus, readers are processes that are not required to exclude one another and
- writers are processes that are required to exclude all other processes, readers and writers alike.

Readers-Writers Problem…

- Shared data

  var *mutex*, *wrt* : *semaphore* (=1);

  readcount : *integer* (=0);

- The mutex semaphore is used to ensure mutual exclusion when the variable readcount is updated.
- The readcount variable keeps track of how many processes are currently reading the object.
- The semaphore wrt functions as a mutual-exclusion semaphore for the writers.

Readers-Writers Problem…

- Writer process

wait(*wrt*);

…

writing is performed

…

signal(*wrt*);

Readers-Writers Problem (Cont.)

- Reader process

```
wait(mutex);
   readcount := readcount +1;
   if readcount = 1 then wait(wrt);
signal(mutex);
        …
   reading is performed
        …
wait(mutex);
   readcount := readcount – 1;
   if readcount = 0 then signal(wrt);
signal(mutex):
```

Readers-Writers Problem – Do a trace!

```
wait(mutex);
    readcount := readcount +1;
    if readcount = 1 then wait(wrt);
signal(mutex);
     …
    reading is performed
     …
wait(mutex);
    readcount := readcount – 1;
    if readcount = 0 then signal(wrt);
        signal(mutex):
```

```
wait(mutex);
    readcount := readcount +1;
    if readcount = 1 then wait(wrt);
signal(mutex);
     …
    reading is performed
     …
wait(mutex);
    readcount := readcount – 1;
    if readcount = 0 then signal(wrt);
        signal(mutex):
```

Readers-Writers Problem…

- As long as at least one reader is still active, subsequent readers are admitted. As long as there is a steady supply for readers, they will all get in as soon as they arrive.
- The writer will be kept suspended until no reader is present. If a new reader arrives, say, every 2 seconds, and each reader takes 5 seconds to do its work, the writer will never get in.

Readers-Writers Problem. Priority for writers

```
/* program readersandwriters */
int readcount,writecount;
semaphore x = 1, y = 1, z = 1, wsem = 1, rsem = 1;
void reader()
{
    while (true){
     semWait (z);
          semWait (rsem);
               semWait (x);
                    readcount++;
                    if (readcount == 1)
                         semWait (wsem);
                    semSignal (x);
               semSignal (rsem);
          semSignal (z);
          READUNIT();
          semWait (x);
               readcount--;
               if (readcount == 0) semSignal (wsem);
          semSignal (x);
    }
}
```

Readers-Writers Problem. Priority for writers

```
void writer ()
{
    while (true){
        semWait (y);
            writecount++;
            if (writecount == 1)
                semWait (rsem);
        semSignal (y);
        semWait (wsem);
        WRITEUNIT();
        semSignal (wsem);
        semWait (y);
            writecount;
            if (writecount == 0) semSignal (rsem);
        semSignal (y);
    }
}
```

Answer to Quiz

- Shortest job first and priority-based scheduling algorithms could result in starvation.