Concurrency and IPC

Chapters 5 and 6

Operating Systems:
Internals and Design Principles, 9/E
William Stallings

Concurrency

When several processes/threads have access to some shared resources

- Multiple applications
- Structured applications
- Operating system structure
- Multithreaded processes

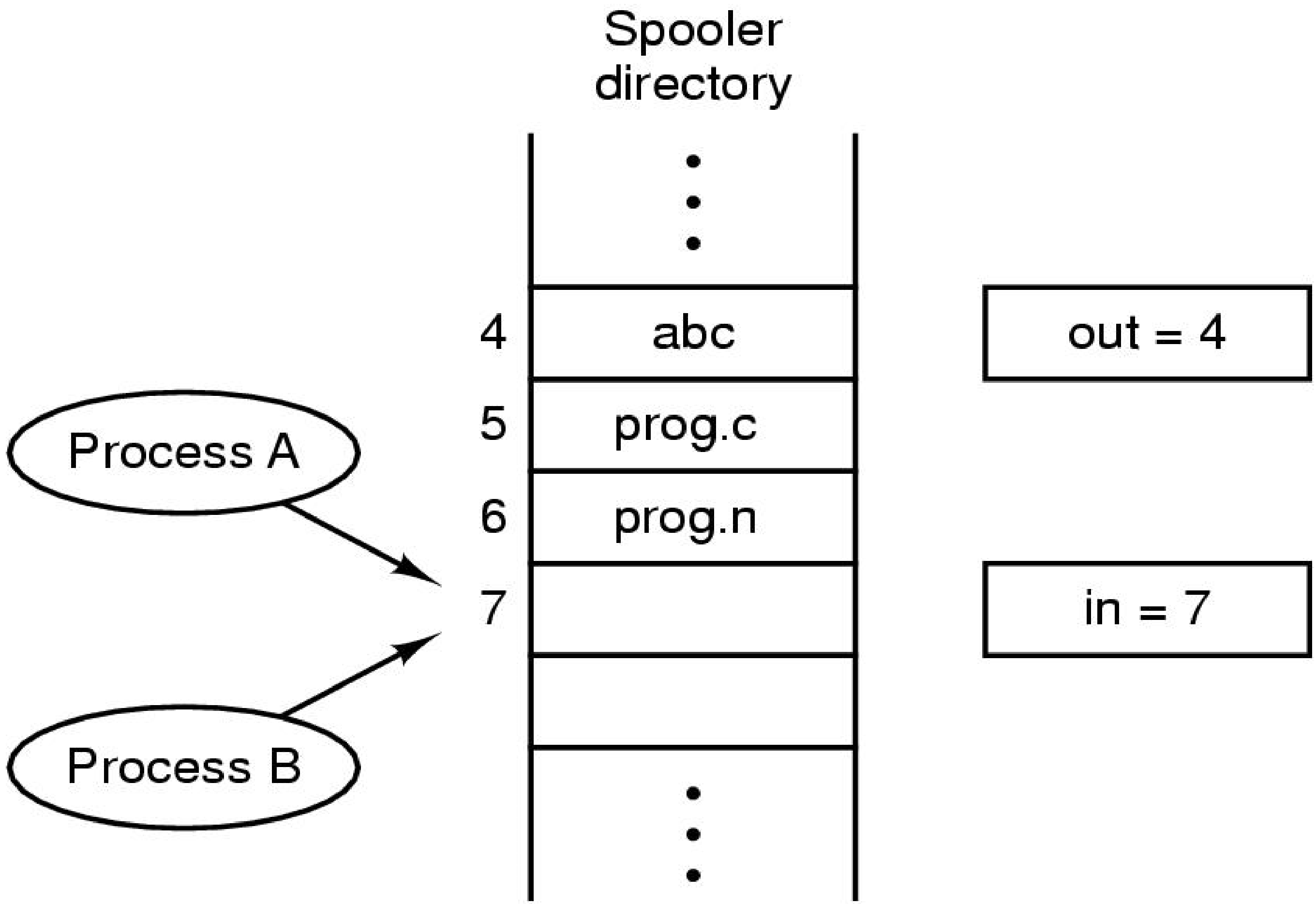
Table 5.1 Some Key Terms Related to Concurrency

atomic operation	A sequence of one or more statements that appears to be indivisible; that is, no other process can see an intermediate state or interrupt the operation.
critical section	A section of code within a process that requires access to shared resources and that must not be executed while another process is in a corresponding section of code.
deadlock	A situation in which two or more processes are unable to proceed because each is waiting for one of the others to do something.
livelock	A situation in which two or more processes continuously change their states in response to changes in the other process(es) without doing any useful work.
mutual exclusion	The requirement that when one process is in a critical section that accesses shared resources, no other process may be in a critical section that accesses any of those shared resources.
race condition	A situation in which multiple threads or processes read and write a shared data item and the final result depends on the relative timing of their execution.
starvation	A situation in which a runnable process is overlooked indefinitely by the scheduler; although it is able to proceed, it is never chosen.

Difficulties of Concurrency

- Sharing of global resources – coordinated access to shared resources
- Operating system managing the allocation of resources optimally – do we know the future behaviour (e.g. needed resources) of (interactive) processes?
- Difficult to locate programming errors

Race Conditions



Two processes want to access shared memory at the same time.

The final result depends on who runs precisely when (determined by the scheduler).

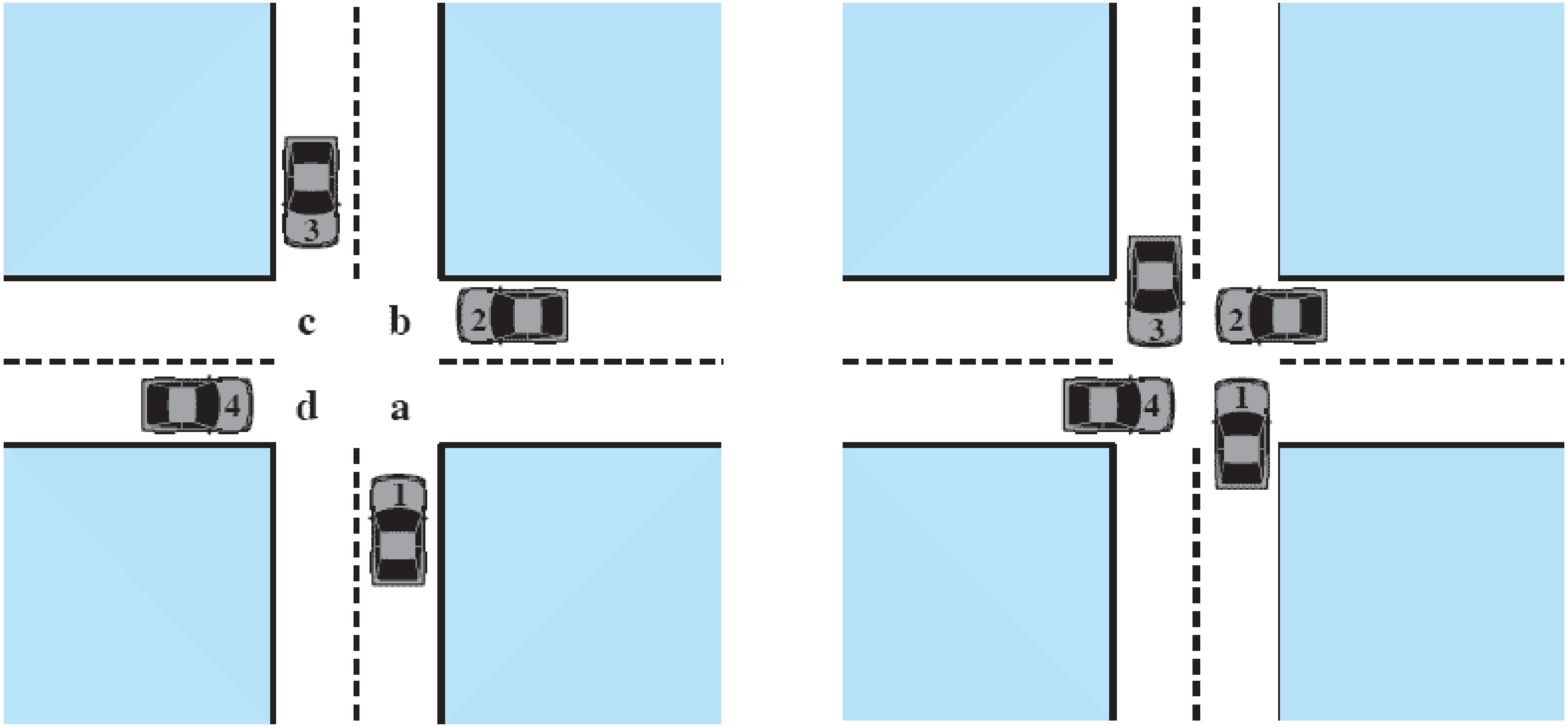
Potential Problems

- **Data incoherency**
- **Deadlock/Livelock**: processes are “frozen” because of mutual dependency on each other
- **Starvation**: some of the processes are unable to make progress (i.e., to execute useful code)

Deadlock

- Permanent blocking of a set of processes – typically they compete for system resources or communicate with each other
- No efficient solution
- Involve conflicting needs for resources by two or more processes

Deadlock in Traffic



(a) Deadlock possible

(b) Deadlock

Figure 6.1 Illustration of Deadlock

Non-deadlock - Joint Progress Diagram

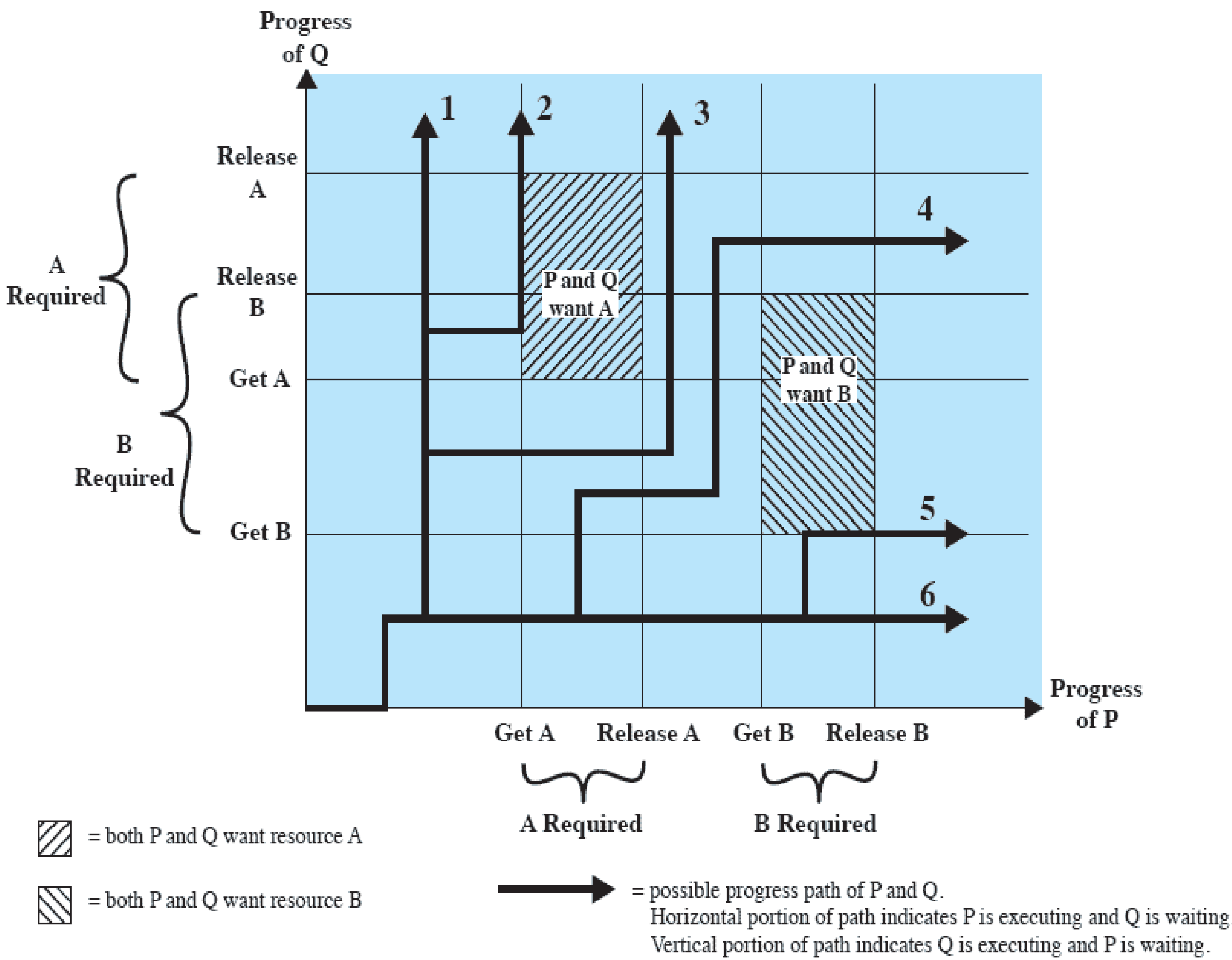


Figure 6.3 Example of No Deadlock [BACO03]

Deadlock in a Computer – Fatal Region

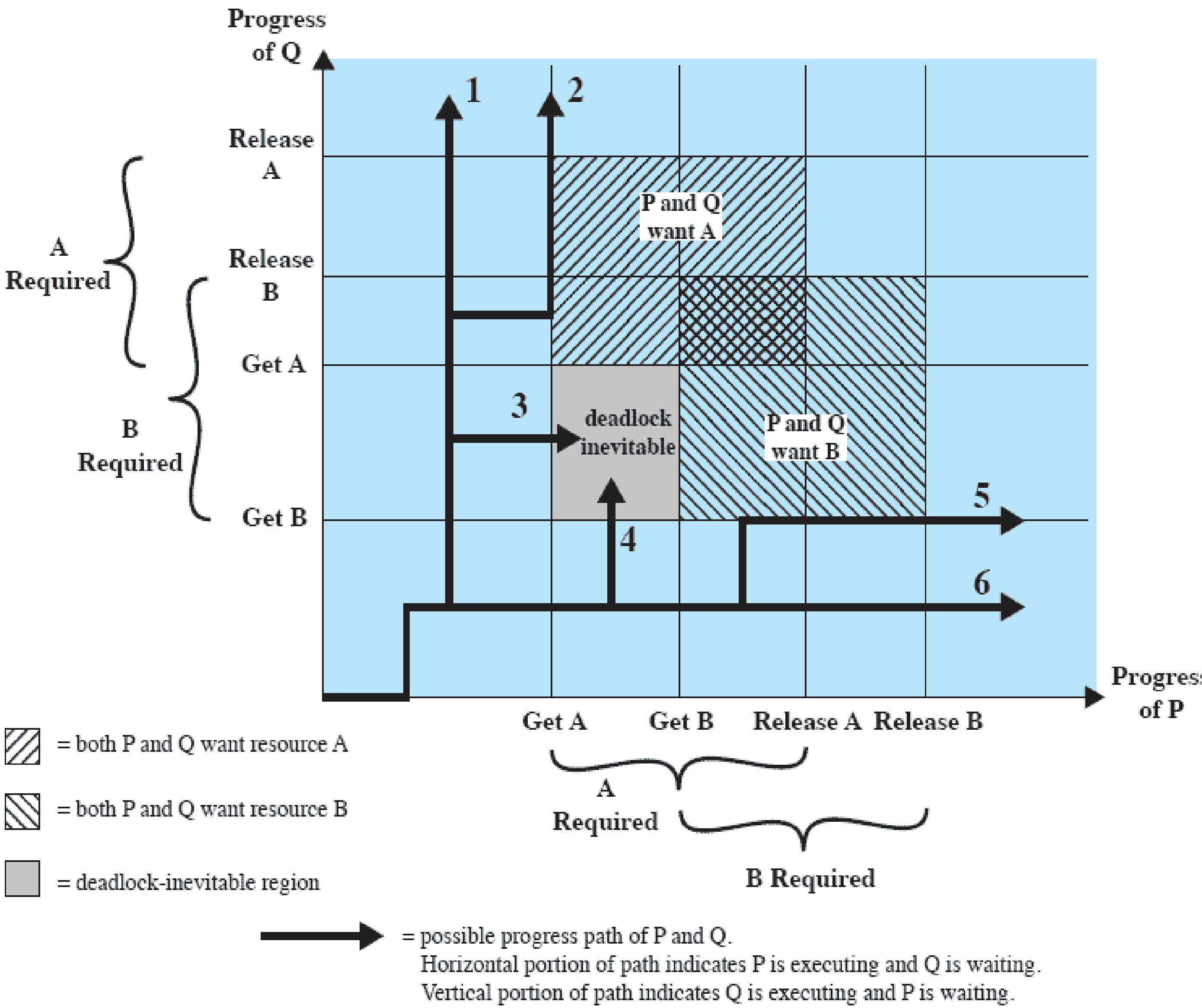


Figure 6.2 Example of Deadlock

Deadlock Definition

- Formal definition :

A set of processes is deadlocked if each process in the set is waiting for an event that only another process in the set can cause.

- Usually the event is to release a currently held resource
- None of the processes can …
  - run
  - release resources
  - be awakened

How Deadlock Can Occur

Process P

Step	Action
p_0	Request (D)
p_1	Lock (D)
p_2	Request (T)
p_3	Lock (T)
p_4	Perform function
p_5	Unlock (D)
p_6	Unlock (T)

Process Q

Step	Action
q_0	Request (T)
q_1	Lock (T)
q_2	Request (D)
q_3	Lock (D)
q_4	Perform function
q_5	Unlock (T)
q_6	Unlock (D)

Figure 6.4 Example of Two Processes Competing for Reusable Resources

Conditions for Deadlock

- **Mutual exclusion**
  - Only one process may use a resource at a time
- **Hold-and-wait**
  - A process may hold allocated resources while awaiting assignment of others

Conditions for Deadlock

- **No preemption**
  - No resource can be forcibly removed from a process holding it
- **Circular wait**
  - A closed chain of processes exists, such that each process holds at least one resource needed by the next process in the chain

Deadlock Prevention

- Mutual Exclusion
  - Spooling
- Hold and Wait
  - Require that a process request all of its required resources at one time
  - Requests would be granted/denied simultaneously

Deadlock Prevention (cont.)

- No Preemption
  - Process must release resource and request it again
  - OS may preempt a process and require it to release its resources
- Circular Wait
  - Define a linear ordering of resources
  - Require that processes request resources according to this ordering

Deadlock Detection

- Multiple instances of resource types
- Resource vector (RV): all resources
- Claim matrix (CM): needs of processes
- Allocation matrix (AM): how resources are allocated to processes
- Request matrix (RM = CM – AM): pending requests by processes
- Availability vector (AV): currently available resources (i.e. not allocated yet)

Deadlock Detection Example

	R1	R2	R3	R4	R5
P1	0	1	0	0	1
P2	0	0	1	0	1
P3	0	0	0	0	1
P4	1	0	1	0	1

Request matrix Q

	R1	R2	R3	R4	R5
P1	1	0	1	1	0
P2	1	1	0	0	0
P3	0	0	0	1	0
P4	0	0	0	0	0

Allocation matrix A

R1	R2	R3	R4	R5
2	1	1	2	1

Resource vector

R1	R2	R3	R4	R5
0	0	0	0	1

Allocation vector

Figure 6.10 Example for Deadlock Detection

Strategies Once Deadlock Detected

- Abort all deadlocked processes
- Back up each deadlocked process to some previously defined checkpoint, and restart all process - original deadlock may re-occur
- Successively abort deadlocked processes until deadlock no longer exists
- Successively preempt resources until deadlock no longer exists

Deadlock Avoidance

- A decision is made dynamically whether the current resource allocation request will, if granted, potentially lead to a deadlock
- Requires knowledge of future process requests!

Resource Allocation Denial

- Referred to as the *Banker's Algorithm*
- State of the system is the current allocation of resources to processes
- *Safe state* is where there is at least one sequence of execution of processes that does not result in deadlock
- *Unsafe state* is a state that is not safe

Determination of a Safe State:

Allocate R3 to P2?

Claim matrix C

	R1	R2	R3
P1	3	2	2
P2	6	1	3
P3	3	1	4
P4	4	2	2

Allocation matrix A

	R1	R2	R3
P1	1	0	0
P2	6	1	2
P3	2	1	1
P4	0	0	2

C – A

	R1	R2	R3
P1	2	2	2
P2	0	0	1
P3	1	0	3
P4	4	2	0

Resource vector R

R1	R2	R3
9	3	6

Available vector V

R1	R2	R3
0	1	1

(a) Initial state

Determination of a Safe State

	R1	R2	R3
P1	3	2	2
P2	0	0	0
P3	3	1	4
P4	4	2	2

Claim matrix C

	R1	R2	R3
P1	1	0	0
P2	0	0	0
P3	2	1	1
P4	0	0	2

Allocation matrix A

	R1	R2	R3
P1	2	2	2
P2	0	0	0
P3	1	0	3
P4	4	2	0

C – A

R1	R2	R3
9	3	6

Resource vector **R**

R1	R2	R3
6	2	3

Available vector **V**

(b) **P2 runs to completion**

Determination of an Unsafe State

Claim matrix **C**

	R1	R2	R3
P1	3	2	2
P2	6	1	3
P3	3	1	4
P4	4	2	2

Allocation matrix **A**

	R1	R2	R3
P1	1	0	0
P2	5	1	1
P3	2	1	1
P4	0	0	2

C – A

	R1	R2	R3
P1	2	2	2
P2	1	0	2
P3	1	0	3
P4	4	2	0

Resource vector **R**

R1	R2	R3
9	3	6

Available vector **V**

R1	R2	R3
1	1	2

(a) **Initial state**

Claim matrix **C**

	R1	R2	R3
P1	3	2	2
P2	6	1	3
P3	3	1	4
P4	4	2	2

Allocation matrix **A**

	R1	R2	R3
P1	2	0	1
P2	5	1	1
P3	2	1	1
P4	0	0	2

C – A

	R1	R2	R3
P1	1	2	1
P2	1	0	2
P3	1	0	3
P4	4	2	0

Resource vector **R**

R1	R2	R3
9	3	6

Available vector **V**

R1	R2	R3
0	1	1

(b) P1 requests one unit each of R1 and R3

Deadlock Avoidance Logic

```
struct state {
       int resource[m];
       int available[m];
       int claim[n][m];
       int alloc[n][m];
}
```

(a) global data structures

```
if (alloc [i,*] + request [*] > claim [i,*])
     < error >;                                   /* total request > claim*/
else if (request [*] > available [*])
     < suspend process >;
else {                                                  /* simulate alloc */
     < define newstate by:
     alloc [i,*] = alloc [i,*] + request [*];
     available [*] = available [*] - request [*] >;
}
if (safe (newstate))
     < carry out allocation >;
else {
     < restore original state >;
     < suspend process >;
}
```

(b) resource alloc algorithm

Deadlock Avoidance Logic

```
boolean safe (state S) {
   int currentavail[m];
   process rest[<number of processes>];
   currentavail = available;
   rest = {all processes};
   possible = true;
   while (possible) {
      <find a process Pk in rest such that
         claim [k,*] – alloc [k,*] <= currentavail;>
      if (found) {                              /* simulate execution of Pk */
         currentavail = currentavail + alloc [k,*];
         rest = rest - {Pk};
      }
      else possible = false;
   }
   return (rest == null);
}
```

(c) test for safety algorithm (banker's algorithm)

Figure 6.9 Deadlock Avoidance Logic

Deadlock Avoidance

- Maximum resource requirement must be stated in advance
- Processes under consideration must be independent; no synchronization (order of execution) requirements
- No process may exit/block while holding resources

How (not) to Write Concurrent Code

- So far: limitation of OS techniques to avoid deadlock
- They do not address the problems of

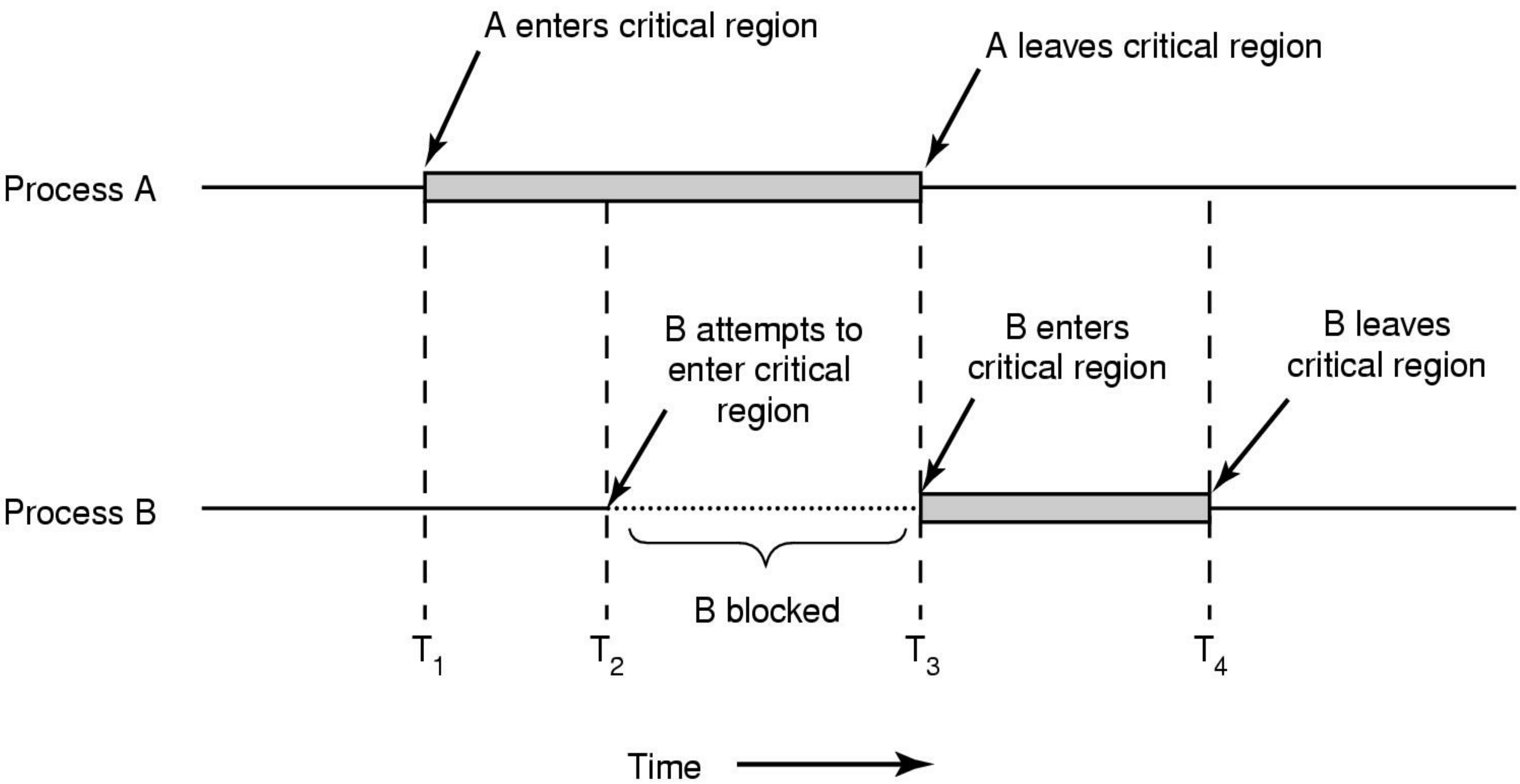
data incoherency,

starvation,

dependencies between processes

- When writing concurrent code (e.g. multithreaded process) we have to make extra care
- **Process switches can occur any time!!!**

Critical Regions



Mutual exclusion using critical regions

Mutual exclusion

Critical region: part of the program where shared resource (memory) is accessed.

Four conditions for correct and efficient communication:

1. *Mutual exclusion*: No two processes simultaneously in their critical regions
2. No assumptions made about speeds (or numbers) of CPUs
3. *Progress*: No process running outside its critical region may block another process to enter
4. *Fairness*, i.e., *no starvation*: No process must wait forever to enter its critical region (assuming fair scheduling!)

1st attempt for two processes: P0 and P1

```
int flag[2] = {false, false};
void critical_region (int i)
{
while (true) {
    while (flag[1-i]);        // loop
    flag[i] = true;
    critical();
    flag[i] = false;
    noncritical();
}
}
```

2nd attempt

```
int flag[2] = {false, false};
void critical_region (int i)
{
while (true) {
    flag[i] = true;
    while (flag[1-i]);        // loop
    critical();
    flag[i] = false;
    noncritical();
}
}
```

Strict Alternation for P0 and P1

```
while (TRUE) {
    while (turn != 0)        /* loop */ ;
    critical_region( );
    turn = 1;
    noncritical_region( );
}
```

(a)

```
while (TRUE) {
    while (turn != 1)        /* loop */ ;
    critical_region( );
    turn = 0;
    noncritical_region( );
}
```

(b)

Proposed solution to critical region problem

(a) Process 0 (b) Process 1

Invariance: turn=id of the process in c.s.

Peterson’s Solution for P0 and P1

```
#define FALSE  0
#define TRUE   1
#define N      2                        /* number of processes */

int turn;                               /* whose turn is it? */
int interested[N];                      /* all values initially 0 (FALSE) */

void enter_region(int process);         /* process is 0 or 1 */
{
     int other;                         /* number of the other process */

     other = 1 – process;               /* the opposite of process */
     interested[process] = TRUE;        /* show that you are interested */
     turn = process;                    /* set flag */
     while (turn == process && interested[other] == TRUE) /* null statement */ ;
}

void leave_region(int process)          /* process: who is leaving */
{
     interested[process] = FALSE;       /* indicate departure from critical region */
}
```

Peterson's Solution (ctd.)

- `Interested(process)=False =>` process is not in and does not want to enter critical section
- If both are interested, a process can enter only if it is the other’s turn (the other process arrived later)
- Works only for two processes (generalization is possible)
- Works in distributed systems (no special instruction needed)
- Process loops when unable to enter c.s.

Mutual Exclusion: Disabling Interrupts

- A process runs until it invokes an operating system service or until it is interrupted
- Disabling interrupts guarantees mutual exclusion
- Processor is limited in its ability to interleave programs
- Will not work in multiprocessor architecture

Should a user process be allowed to disable interrupts?

Mutual Exclusion: Exchange

- Exchange instruction

```
void exchange (int register, int memory)
{
   int temp;
   temp = memory;
   memory = register;
   register = temp;
}
```

Mutual Exclusion with Exchange

```
int bolt = 0;
int key[N] = 1;
void critical_region (int i)
{
while (true) {
    while (key(i) == 1) {exchange (key(i), bolt)};
    critical();
    exchange (key(i), bolt);
    noncritical();
}
}
```

Invariance

bolt + Σ key(i) = N

If process(i) is in c.s. then bolt = 1 and key(i)=0.

Thus other processes cannot enter c.s.

Mutual Exclusion Machine-Instruction: Advantages

–Applicable to any number of processes on either a single processor or multiple processors sharing main memory

–It can be used to support multiple critical sections

Mutual Exclusion Machine-Instruction: Disadvantages

–**Busy-waiting** consumes processor time: processes spin on variable

–**Livelock** is possible: process waits on a variable while other process waits on another variable – none of them can release

–**Priority inversion problem**: low priority process in critical section, high priority process wants to enter the critical section

Semaphores

- Special variable called a semaphore is used for signalling
- If a process is waiting for a signal, it is blocked until that signal is sent

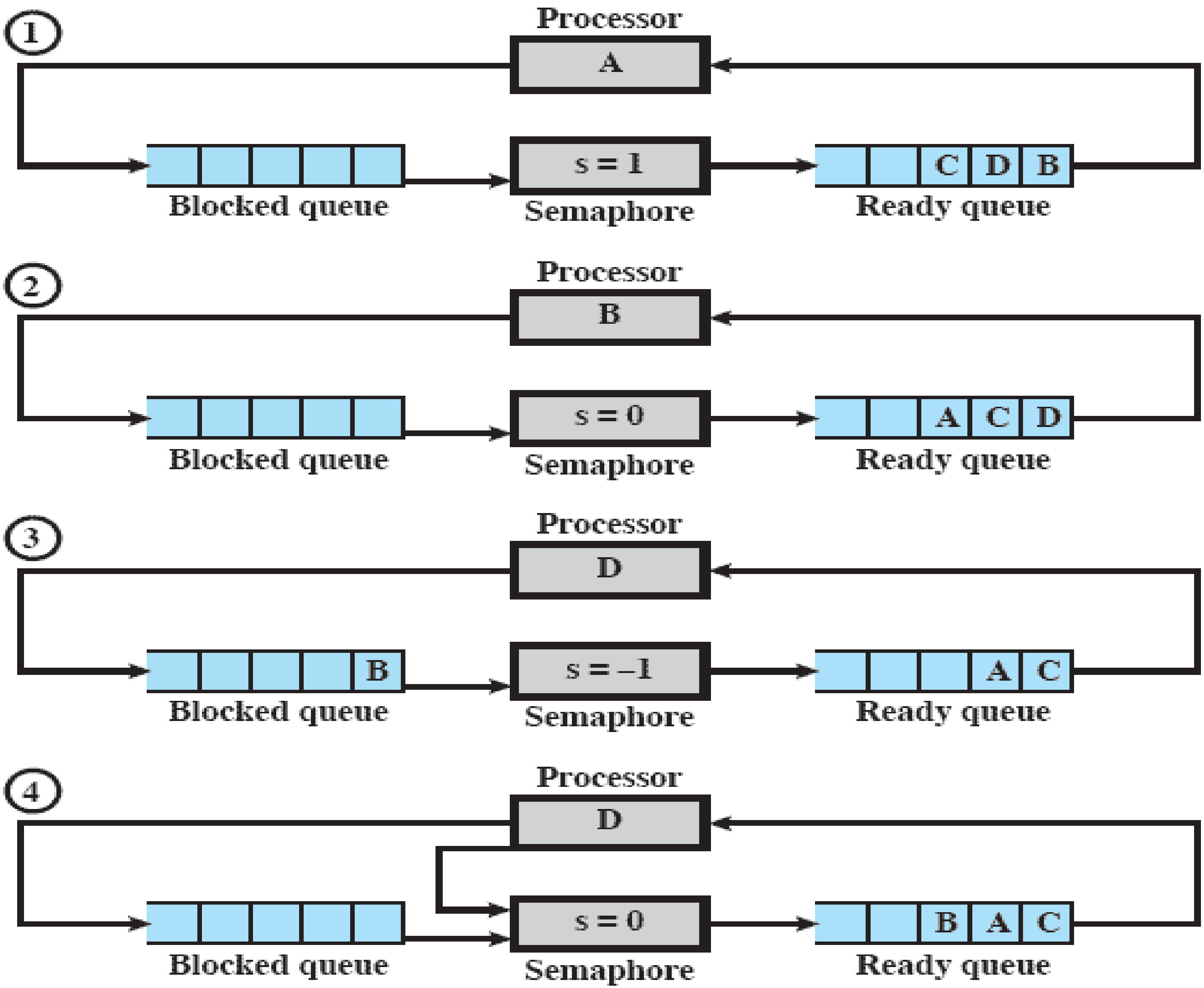
Semaphore Operations

- Semaphore is a variable that has an integer value
  - May be initialized to a non-negative number
  - **Wait** (down, request) operation decrements semaphore value; if the new value is negative, process is blocked
  - **Signal** (up, release) operation increments semaphore value; one of the blocked processes (if any) is unblocked

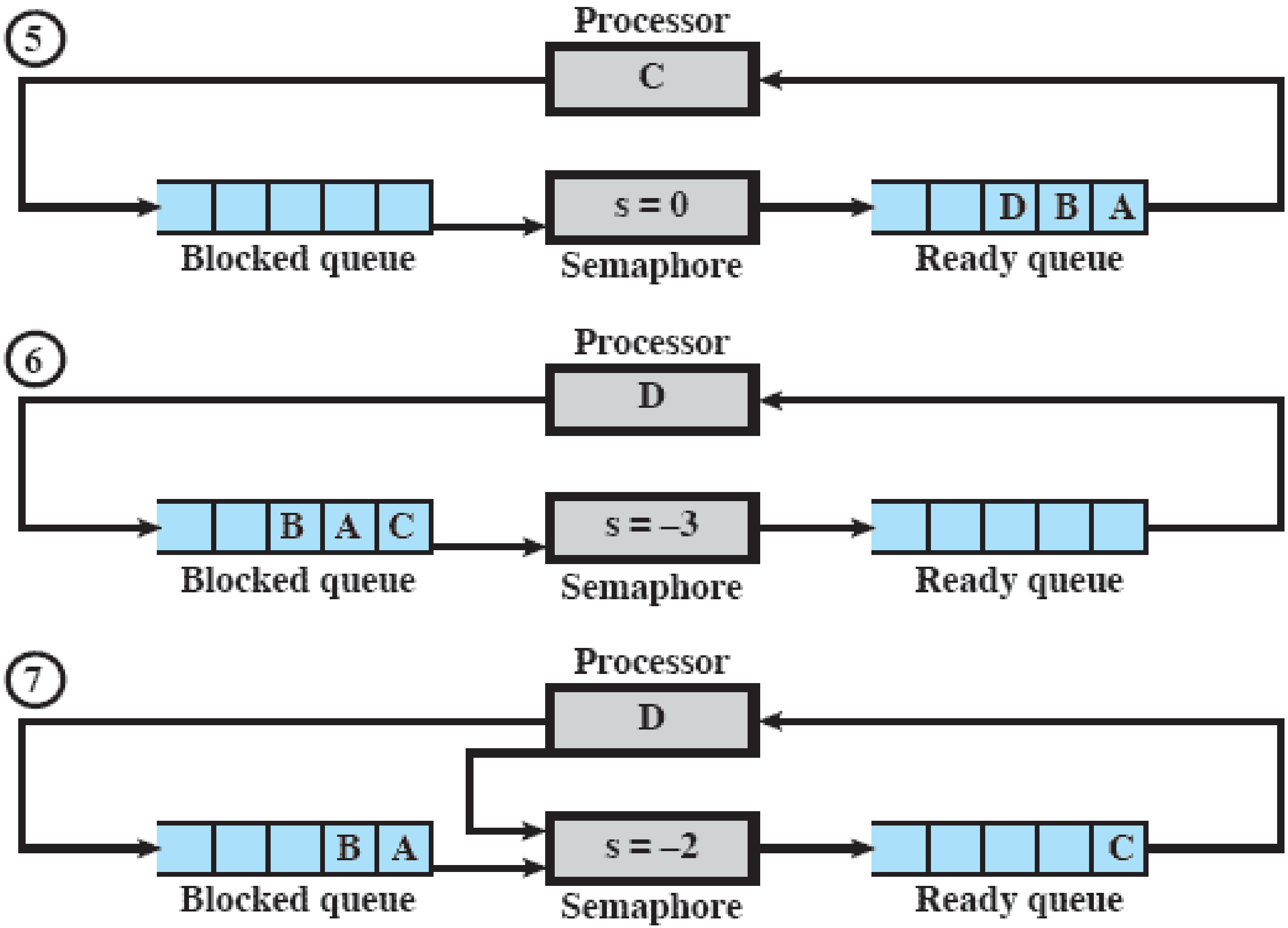
```c
struct semaphore {
      int count;
      queueType queue;
};
void semWait(semaphore s)
{
      s.count--;
      if (s.count < 0) {
            /* place this process in s.queue */;
            /* block this process */;
      }
}
void semSignal(semaphore s)
{
      s.count++;
      if (s.count <= 0) {
            /* remove a process P from s.queue */;
            /* place process P on ready list */;
      }
}
```

Figure 5.3 A Definition of Semaphore Primitives

Example of Semaphore Mechanism: D performs signal



Example of Semaphore Mechanism



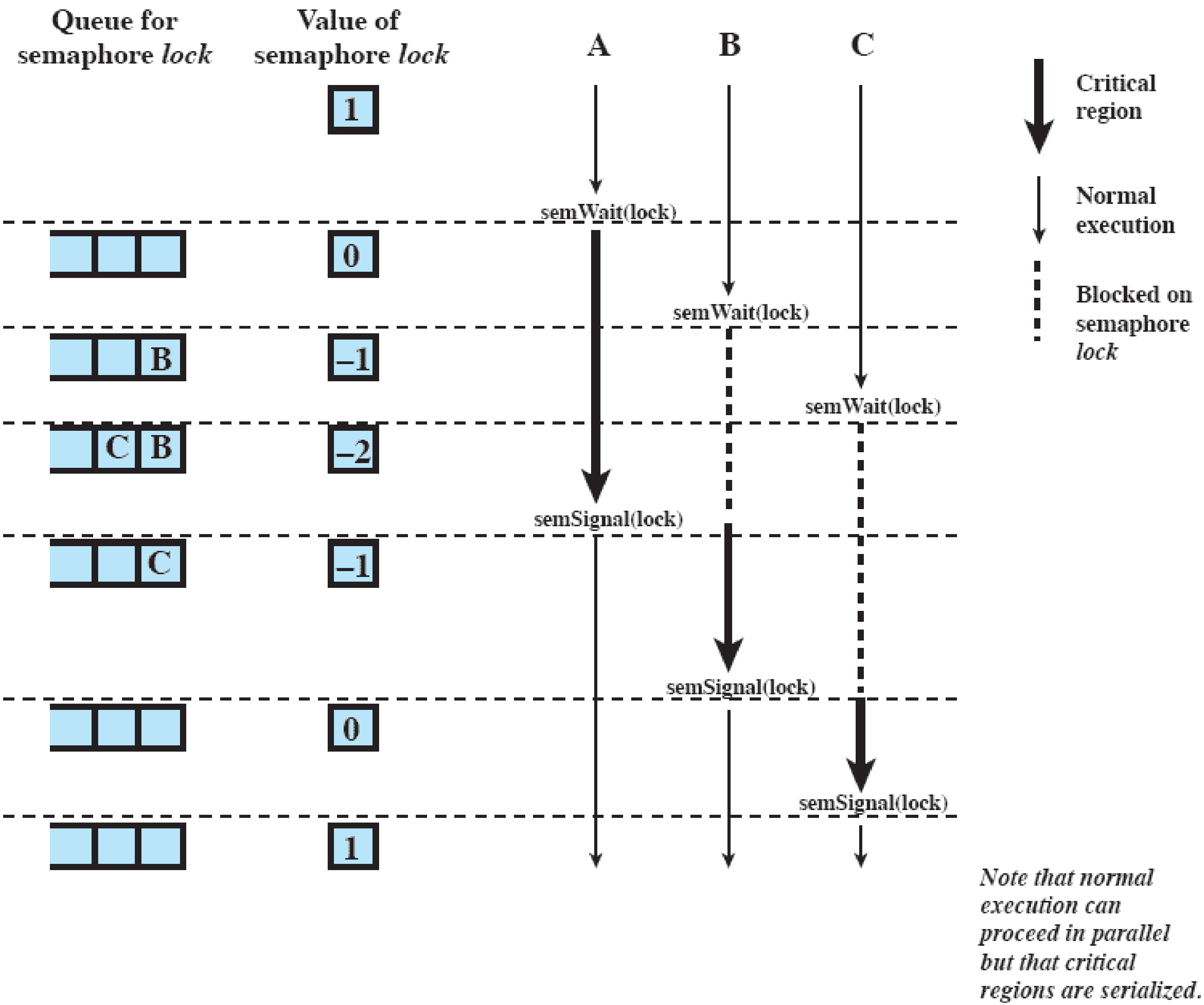
```
struct binary_semaphore {
      enum {zero, one} value;
      queueType queue;
};
void semWaitB(binary_semaphore s)
{
      if (s.value == one)
            s.value = zero;
      else {
               /* place this process in s.queue */;
               /* block this process */;
      }
}
void semSignalB(semaphore s)
{
      if (s.queue is empty())
            s.value = one;
      else {
               /* remove a process P from s.queue */;
               /* place process P on ready list */;
      }
}
```

Figure 5.4 A Definition of Binary Semaphore Primitives

```
/* program mutualexclusion */
const int n = /* number of processes  */;
semaphore s = 1;
void P(int i)
{
     while (true) {
          semWait(s);
          /* critical section   */;
          semSignal(s);
          /* remainder   */;
     }
}
void main()
{
     parbegin (P(1), P(2), . . ., P(n));
}
```

Figure 5.6 Mutual Exclusion Using Semaphores

Processes Using Semaphore



Dining Philosophers Problem

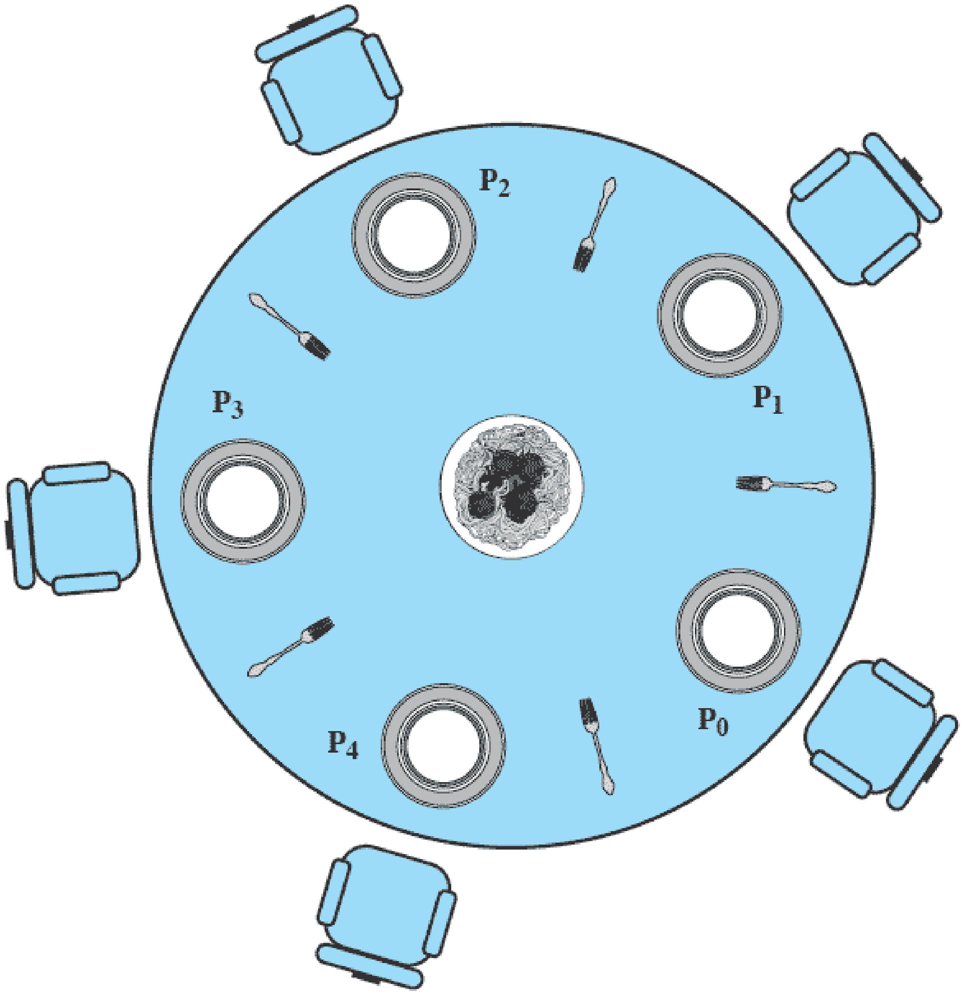


Figure 6.11 Dining Arrangement for Philosophers

Dining Philosophers Problem

```
/* program     diningphilosophers */
semaphore fork [5] = {1};
int i;
void philosopher (int i)
{
     while (true) {
          think();
          wait (fork[i]);
          wait (fork [(i+1) mod 5]);
          eat();
          signal(fork [(i+1) mod 5]);
          signal(fork[i]);
     }
}
void main()
{
     parbegin (philosopher (0), philosopher (1), philosopher
(2),
          philosopher (3), philosopher (4));
     }
```

Figure 6.12 A First Solution to the Dining Philosophers Problem

Dining Philosophers Problem with Semaphores

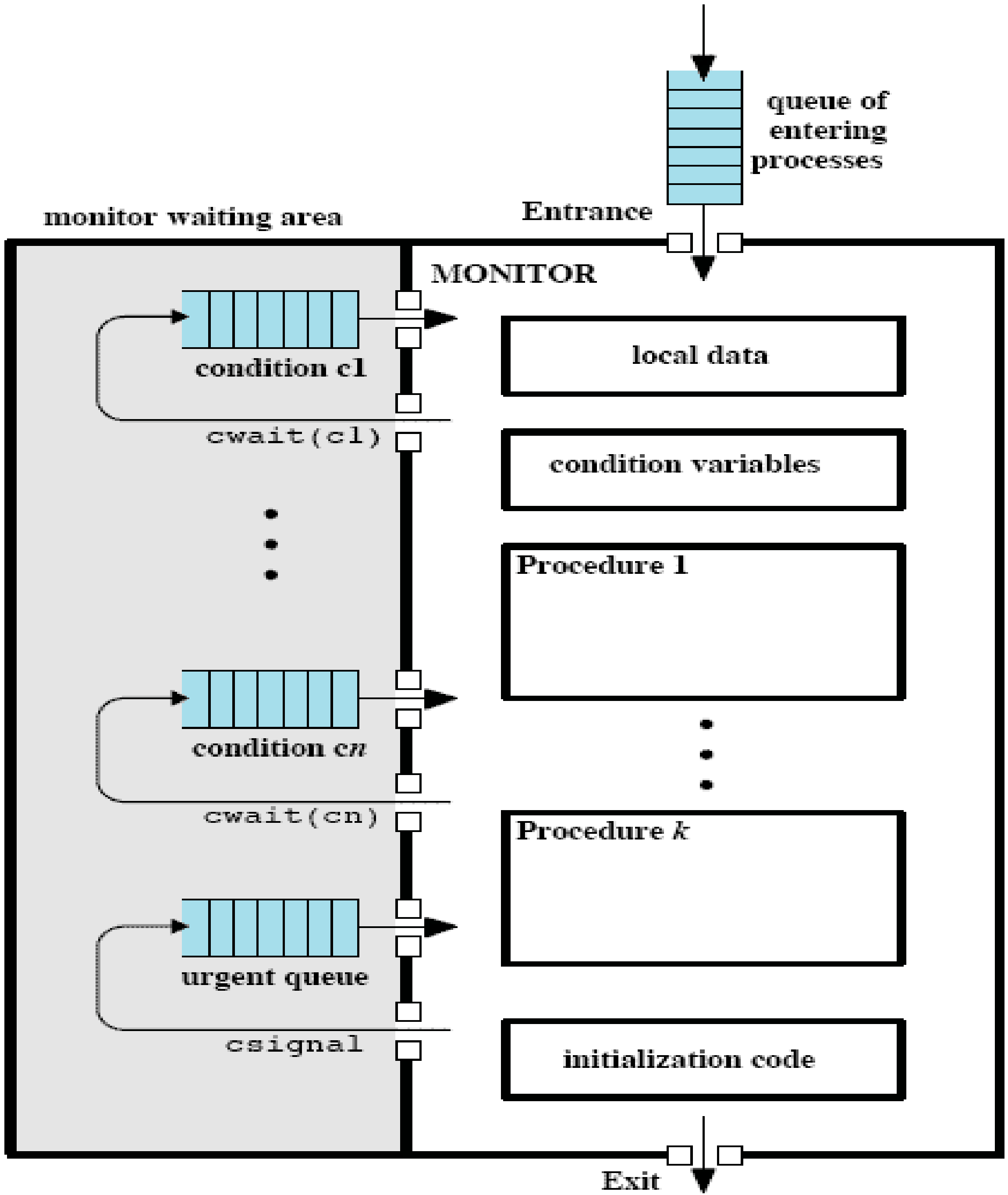
```
/* program diningphilosophers */
semaphore fork[5] = {1};
semaphore room = {4};
int i;
void philosopher (int i)
{
    while (true) {
     think();
     wait (room);
     wait (fork[i]);
     wait (fork [(i+1) mod 5]);
     eat();
     signal (fork [(i+1) mod 5]);
     signal (fork[i]);
     signal (room);
    }

}
void main()
{
    parbegin (philosopher (0), philosopher (1), philosopher (2),
         philosopher (3), philosopher (4));
}
```

Figure 6.13 A Second Solution to the Dining Philosophers Problem

Monitors

- Monitor is a software module
- Chief characteristics
  - Local data variables are accessible only by the monitor
  - Process enters monitor by invoking one of its procedures
  - Only one process may be executing in the monitor at a time – **mutual exclusion is guaranteed**
  - Condition variables for synchronization

Structure of a Monitor



Dining Philosophers Problem with Monitor

```
void philosopher[k=0 to 4]          /* the five philosopher clients */
{
   while (true) {
      <think>;
      get forks(k);           /* client requests two forks via monitor */
      <eat spaghetti>;
      release forks(k);       /* client releases forks via the monitor */
   }
}
```

Figure 6.14 A Solution to the Dining Philosophers Problem Using a Monitor

Dining Philosophers Problem with Monitor

```
monitor dining_controller;
cond ForkReady[5];          /* condition variable for synchronization */
boolean fork[5] = {true};         /* availability status of each fork */

void get_forks(int pid)           /* pid is the philosopher id number */
{
   int left = pid;
   int right = (++pid) % 5;
   /*grant the left fork*/
   if (!fork(left)
      cwait(ForkReady[left]);          /* queue on condition variable */
   fork(left) = false;
   /*grant the right fork*/
   if (!fork(right)
      cwait(ForkReady(right);          /* queue on condition variable */
   fork(right) = false:
}
void release_forks(int pid)
{
   int left = pid;
   int right = (++pid) % 5;
   /*release the left fork*/
   if (empty(ForkReady[left])      /*no one is waiting for this fork */
      fork(left) = true;
   else                        /* awaken a process waiting on this fork */
      csignal(ForkReady[left]);
   /*release the right fork*/
   if (empty(ForkReady[right])     /*no one is waiting for this fork */
      fork(right) = true;
   else                        /* awaken a process waiting on this fork */
      csignal(ForkReady[right]);
}
```

Message Passing

- Enforce mutual exclusion
- Exchange information

- send (destination, message)
- receive (source, message)

Synchronization with Messages

Non-blocking send, blocking receive

Sender continues on

Receiver is blocked until the requested message arrives

Indirect addressing

Messages are sent to a shared data structure consisting of queues

Queues are called mailboxes

One process sends a message to the mailbox and the other process picks up the message from the mailbox

```
/* program mutualexclusion */
const int n = /* number of processes  */;
void P(int i)
{
    message msg;
    while (true) {
     receive (box, msg);
     /* critical section   */;
     send (box, msg);
     /* remainder   */;
    }
}
void main()
{
    create mailbox (box);
    send (box, null);
    parbegin (P(1), P(2), . . ., P(n));
}
```

Figure 5.20 Mutual Exclusion Using Messages

Producer/Consumer (aka Bounded Buffer) Problem

- One or more producers are generating data and placing these in a buffer
- A single consumer is taking items out of the buffer one at time
- Only one producer or consumer may access the buffer at any one time
- Producer can’t add data into full buffer and consumer can’t remove data from empty buffer

Producer/Consumer Messages

```
const int
    capacity = /* buffering capacity */ ;
    null =/* empty message */ ;
int i;
void producer()
{   message pmsg;
    while (true) {
     receive (mayproduce, pmsg);
     pmsg = produce();
     send (mayconsume, pmsg);
    }
}
void consumer()
{   message cmsg;
    while (true) {
     receive (mayconsume, cmsg);
     consume (cmsg);
     send (mayproduce, null);
    }
}

void main()
{
    create_mailbox (mayproduce);
    create_mailbox (mayconsume);
    for (int i = 1; i <= capacity; i++) send (mayproduce, null);
    parbegin (producer, consumer);
}
```

Correct Solution for Bounded Buffer with Semaphores

```
/* program boundedbuffer */
const int sizeofbuffer = /* buffer size */;
semaphore s = 1, n= 0, e= sizeofbuffer;
void producer()
{
     while (true) {
          produce();
          semWait(e);
          semWait(s);
          append();
          semSignal(s);
          semSignal(n);
     }
}
void consumer()
{
     while (true) {
          semWait(n);
          semWait(s);
          take();
          semSignal(s);
          semSignal(e);
          consume();
     }
}
void main()
{
     parbegin (producer, consumer);
}
```

Solution Using Monitor

```
void producer()
{
      char x;
      while (true) {
      produce(x);
      append(x);
      }
}
void consumer()
{
      char x;
      while (true) {
        take(x);
        consume(x);
      }
}
void main()
{
      parbegin (producer, consumer);
}
```

Solution Using Monitor (cont.)

```
/* program producerconsumer */
monitor boundedbuffer;
char buffer [N];                                          /* space for N items */
int nextin, nextout;                                       /* buffer pointers */
int count;                                          /* number of items in buffer */
cond notfull, notempty;              /* condition variables for synchronization */

void append (char x)
{
     if (count == N) cwait(notfull);        /* buffer is full; avoid overflow */
     buffer[nextin] = x;
     nextin = (nextin + 1) % N;
     count++;
     /* one more item in buffer */
     csignal(notempty);                          /* resume any waiting consumer */
}
void take (char x)
{
     if (count == 0) cwait(notempty);    /* buffer is empty; avoid underflow */
     x = buffer[nextout];
     nextout = (nextout + 1) % N;
     count--;                                      /* one fewer item in buffer */
     csignal(notfull);                           /* resume any waiting producer */
}
{                                                             /* monitor body */
     nextin = 0; nextout = 0; count = 0;           /* buffer initially empty */
}
```

Readers-Writers Problem

- 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 Have Priority

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

void main()
{
    readcount = 0;
    parbegin (reader, writer);
}
```

Writers Have Priority

```
/* 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);
    }
}
```

Writers Have Priority (continued)

```
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);
    }
}
void main()
{
    readcount = writecount = 0;
    parbegin (reader, writer);
}
```