UTN FRD – Sistemas Operativos
Unidad III – Sincronización entre Procesos

Multiple Processes

- Central to the design of modern Operating Systems is managing multiple processes
  - Multiprogramming
  - Multiprocessing
  - Distributed Processing
- Big Issue is Concurrency
  - Managing the interaction of all of these processes

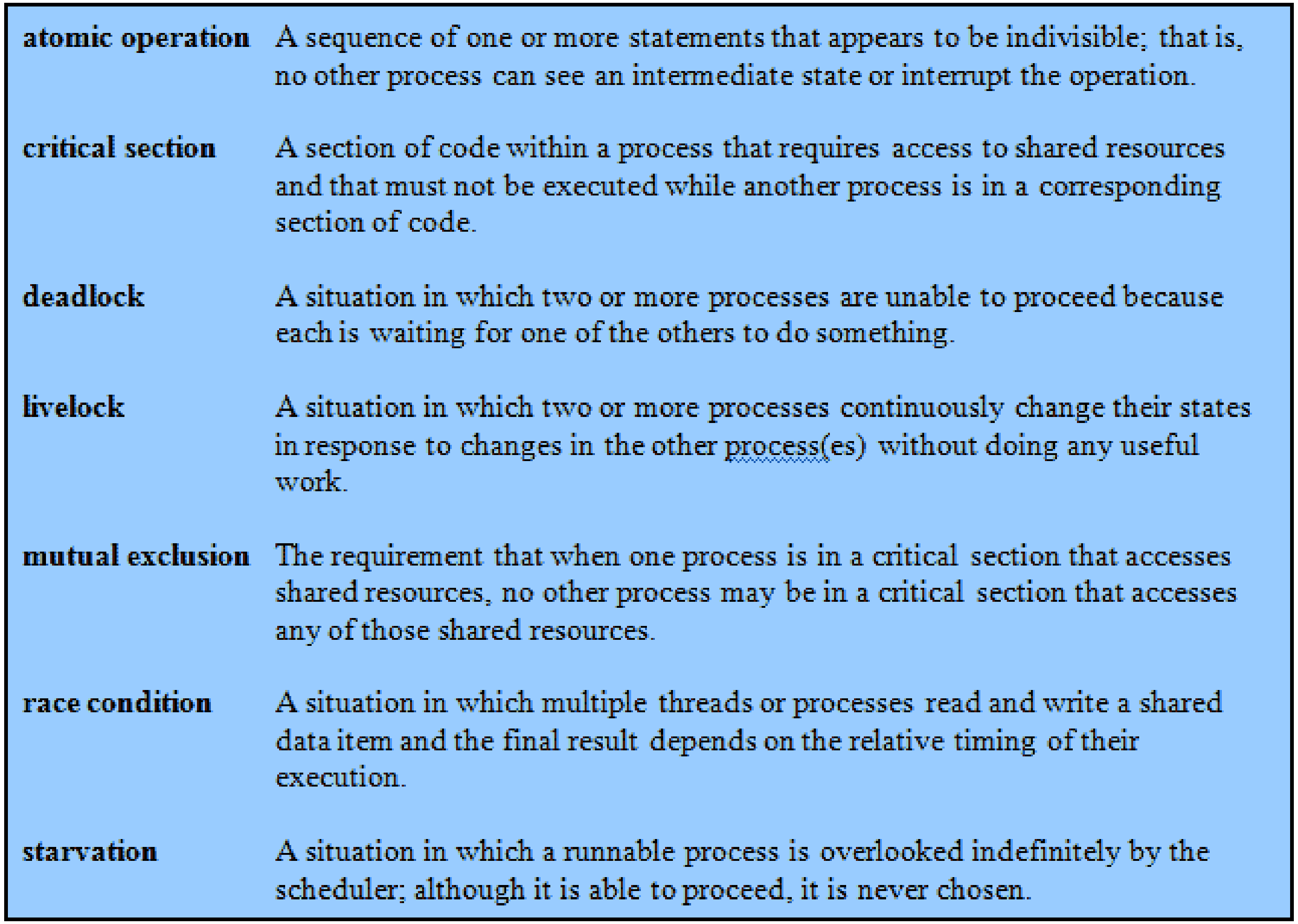
Concurrency

Concurrency arises in:

- Multiple applications
  - Sharing time
- Structured applications
  - Extension of modular design
- Operating system structure
  - OS themselves implemented as a set of processes or threads

Key Terms

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.

Interleaving and Overlapping Processes

- Earlier (Ch2) we saw that processes may be interleaved on uniprocessors

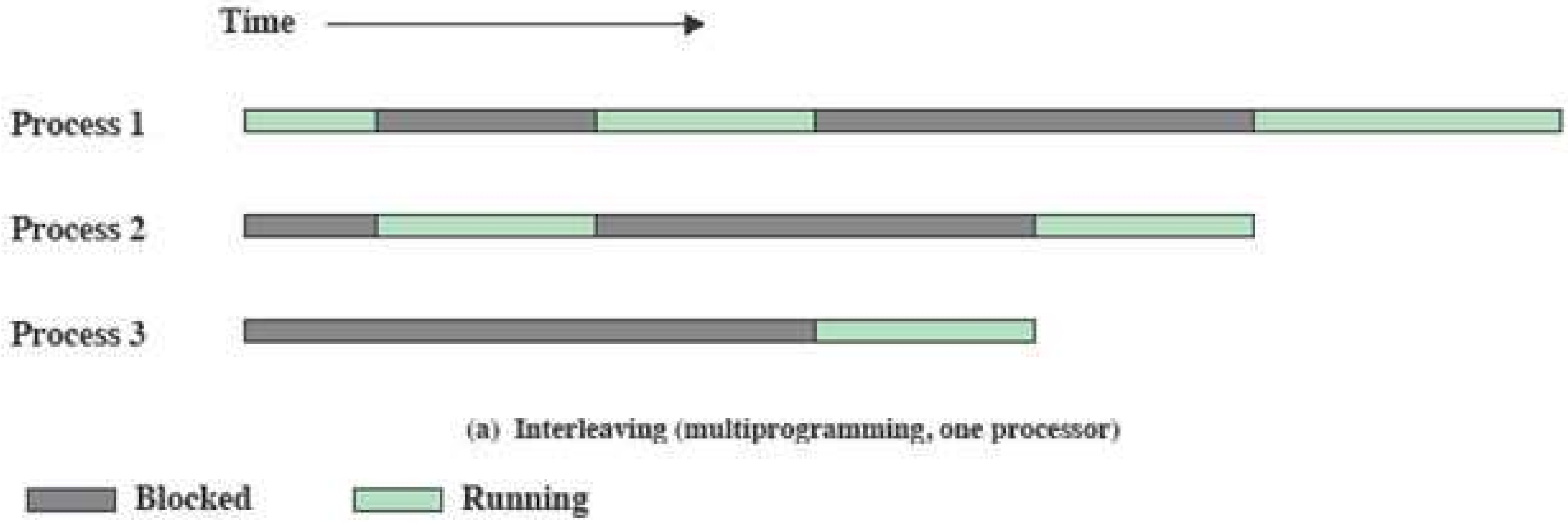
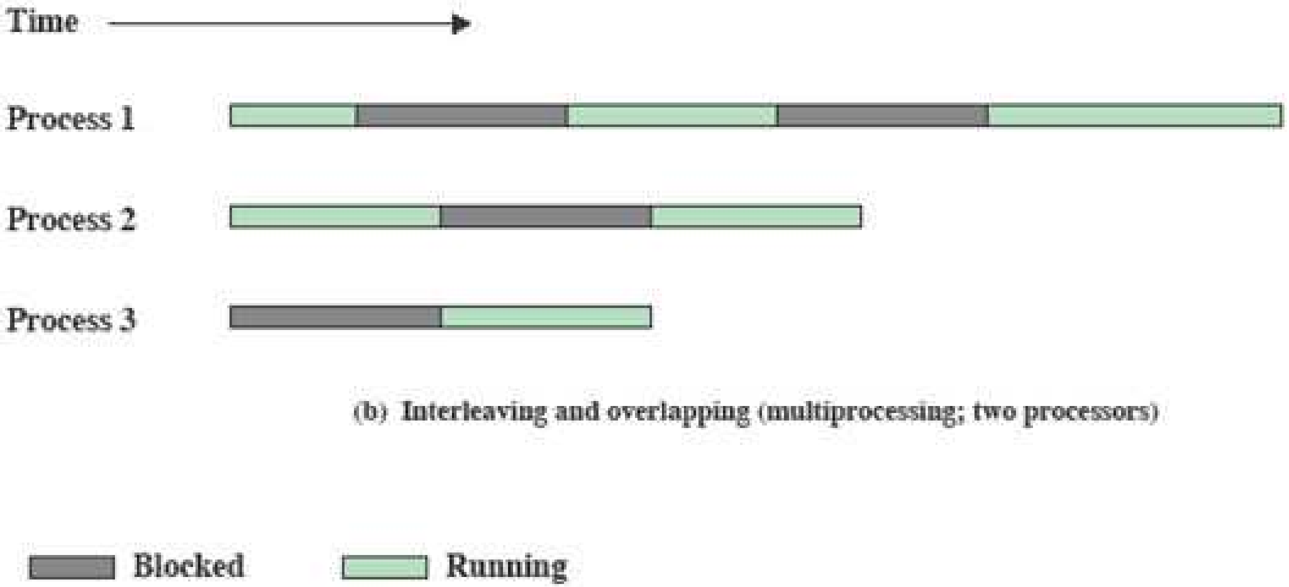


Figure 2.12 Multiprogramming and Multiprocessing

Interleaving and Overlapping Processes

- And not only interleaved but overlapped on multi-processors



(b) Interleaving and overlapping (multiprocessing; two processors)

Figure 2.12 Multiprogramming and Multiprocessing

Difficulties of Concurrency

- Sharing of global resources
- Optimally managing the allocation of resources
- Difficult to locate programming errors as results are not deterministic and reproducible.

A Simple Example

```
void echo()
{
   chin = getchar();
   chout = chin;
   putchar(chout);
}
```

A Simple Example:
On a Multiprocessor

Process P1	Process P2
.	.
chin = getchar();	.
.	chin = getchar();
chout = chin;	chout = chin;
putchar(chout);	.
.	putchar(chout);
.	.

Enforce Single Access

- If we enforce a rule that only one process may enter the function at a time then:
- P1 & P2 run on separate processors
- P1 enters echo first,
  - P2 tries to enter but is blocked – P2 suspends
- P1 completes execution
  - P2 resumes and executes echo

Race Condition

- A race condition occurs when
  - Multiple processes or threads read and write data items
  - They do so in a way where the final result depends on the order of execution of the processes.
- The output depends on who finishes the race last.

Operating System Concerns

- What design and management issues are raised by the existence of concurrency?
- The OS must
  - Keep track of various processes
  - Allocate and de-allocate resources
  - Protect the data and resources against interference by other processes.
  - Ensure that the processes and outputs are independent of the processing speed

Process Interaction

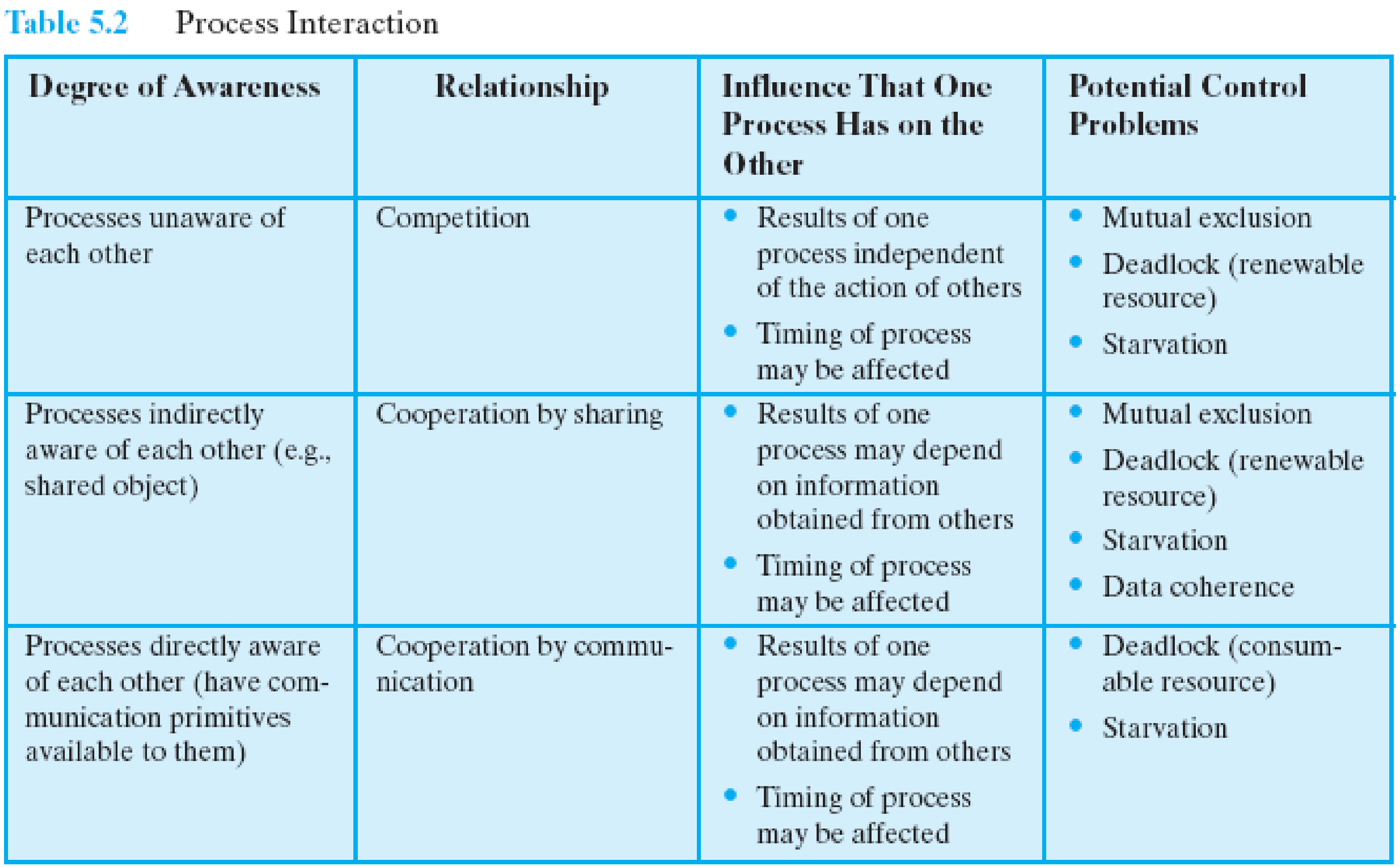


Table 5.2 Process Interaction

Degree of Awareness	**Relationship**	**Influence That One Process Has on the Other**	**Potential Control Problems**
Processes unaware of each other	Competition	• Results of one process independent of the action of others • Timing of process may be affected	• Mutual exclusion • Deadlock (renewable resource) • Starvation
Processes indirectly aware of each other (e.g., shared object)	Cooperation by sharing	• Results of one process may depend on information obtained from others • Timing of process may be affected	• Mutual exclusion • Deadlock (renewable resource) • Starvation • Data coherence
Processes directly aware of each other (have communication primitives available to them)	Cooperation by communication	• Results of one process may depend on information obtained from others • Timing of process may be affected	• Deadlock (consumable resource) • Starvation

Competition among Processes for Resources

Three main control problems:

- Need for Mutual Exclusion
  - Critical sections
- Deadlock
- Starvation

Requirements for Mutual Exclusion

- Only one process at a time is allowed in the critical section for a resource
- A process that halts in its noncritical section must do so without interfering with other processes
- No deadlock or starvation

Requirements for Mutual Exclusion

- A process must not be delayed access to a critical section when there is no other process using it
- No assumptions are made about relative process speeds or number of processes
- A process remains inside its critical section for a finite time only

Disabling Interrupts

- Uniprocessors only allow interleaving
- Interrupt Disabling
  - A process runs until it invokes an operating system service or until it is interrupted
  - Disabling interrupts guarantees mutual exclusion
  - Will not work in multiprocessor architecture

Pseudo-Code

```
while (true) {
    /* disable interrupts */;
    /* critical section */;
    /* enable interrupts */;
    /* remainder */;
}
```

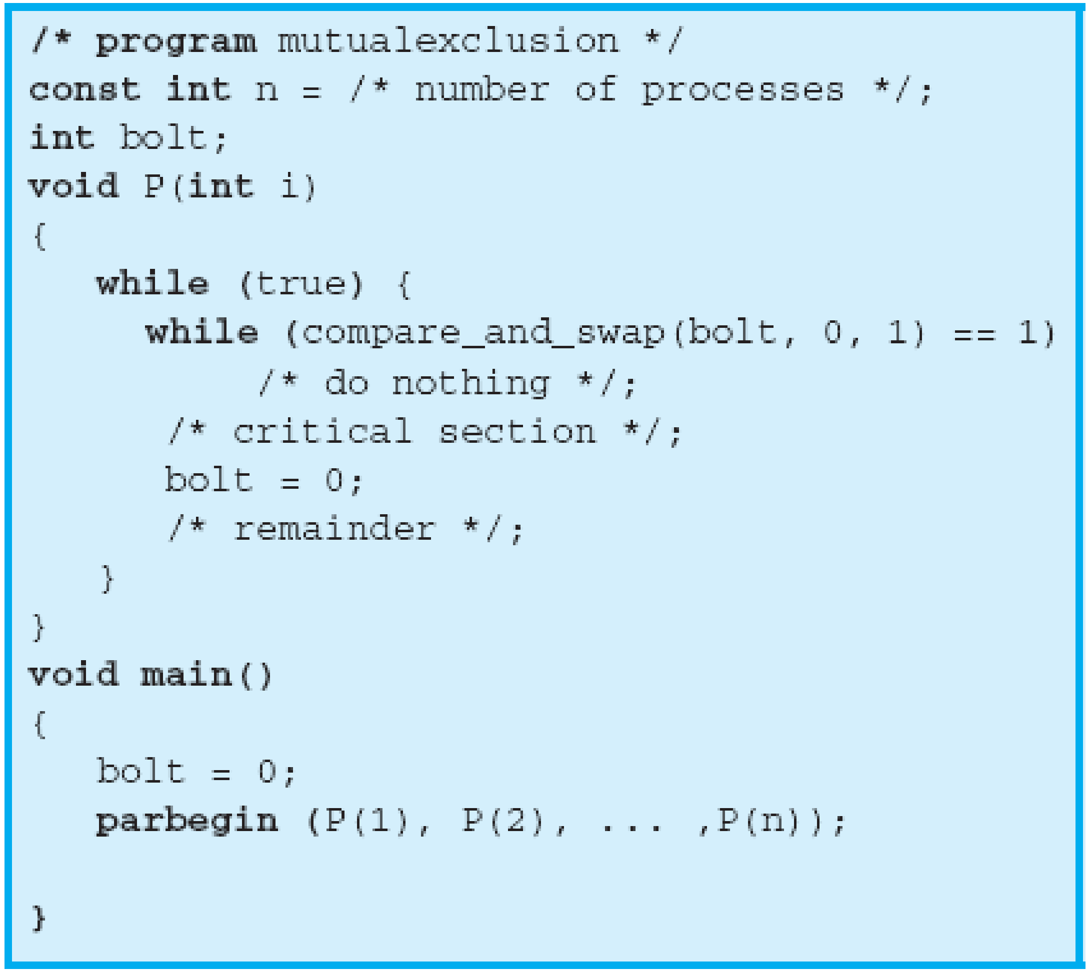
Special Machine Instructions

- Compare&Swap Instruction
  - also called a “compare and exchange instruction”
- Exchange Instruction

Compare&Swap Instruction

```
int compare_and_swap (int *word,
  int testval, int newval)
{
   int oldval;
   oldval = *word;
   if (oldval == testval) *word = newval;
   return oldval;
}
```

Mutual Exclusion (fig 5.2)



```
/* program mutualexclusion */
const int n = /* number of processes */;
int bolt;
void P(int i)
{
   while (true) {
     while (compare_and_swap(bolt, 0, 1) == 1)
          /* do nothing */;
      /* critical section */;
      bolt = 0;
      /* remainder */;
   }
}
void main()
{
   bolt = 0;
   parbegin (P(1), P(2), ... ,P(n));

}
```

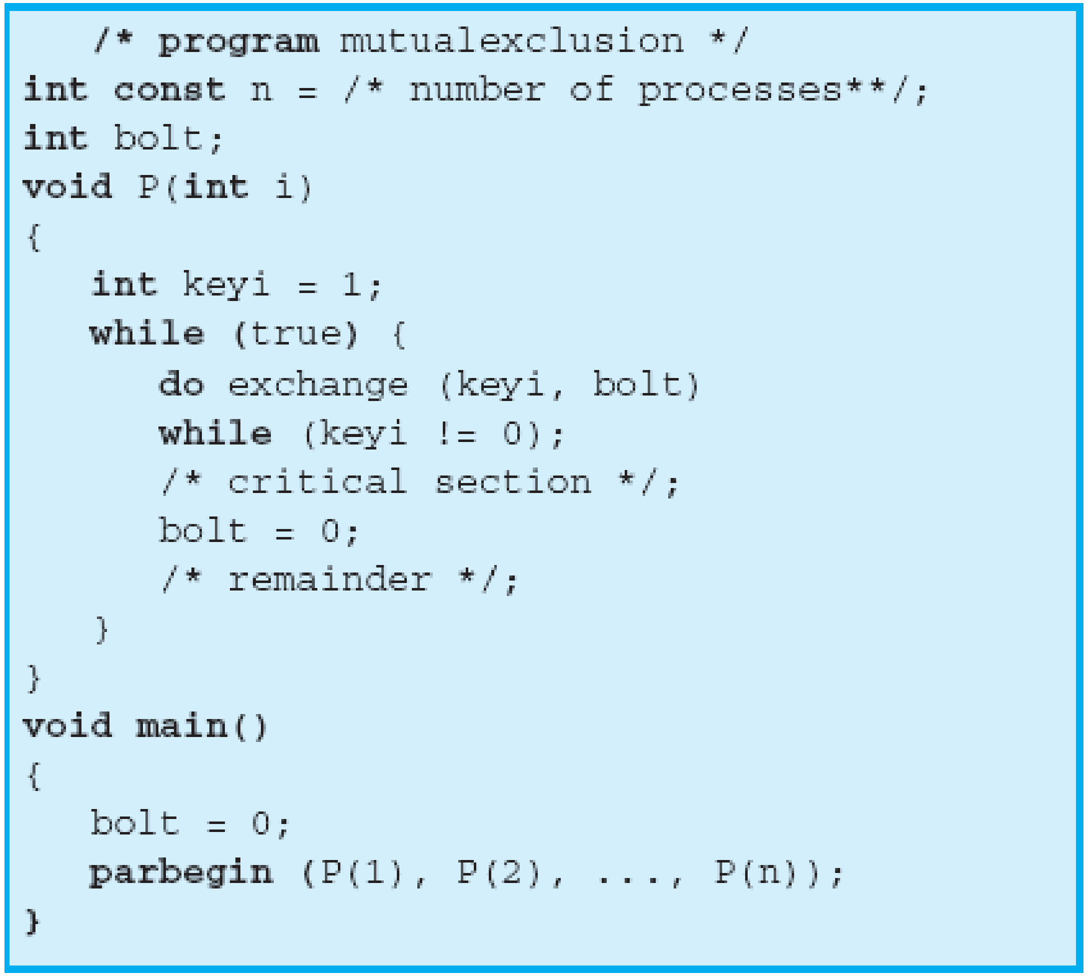
(a) Compare and swap instruction

Exchange instruction

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

Exchange Instruction

(fig 5.2)



```
   /* program mutualexclusion */
int const n = /* number of processes**/;
int bolt;
void P(int i)
{
   int keyi = 1;
   while (true) {
      do exchange (keyi, bolt)
      while (keyi != 0);
      /* critical section */;
      bolt = 0;
      /* remainder */;
   }
}
void main()
{
   bolt = 0;
   parbegin (P(1), P(2), . . ., P(n));
}
```

(b) Exchange instruction

Hardware Mutual Exclusion: Advantages

- Applicable to any number of processes on either a single processor or multiple processors sharing main memory
- It is simple and therefore easy to verify
- It can be used to support multiple critical sections

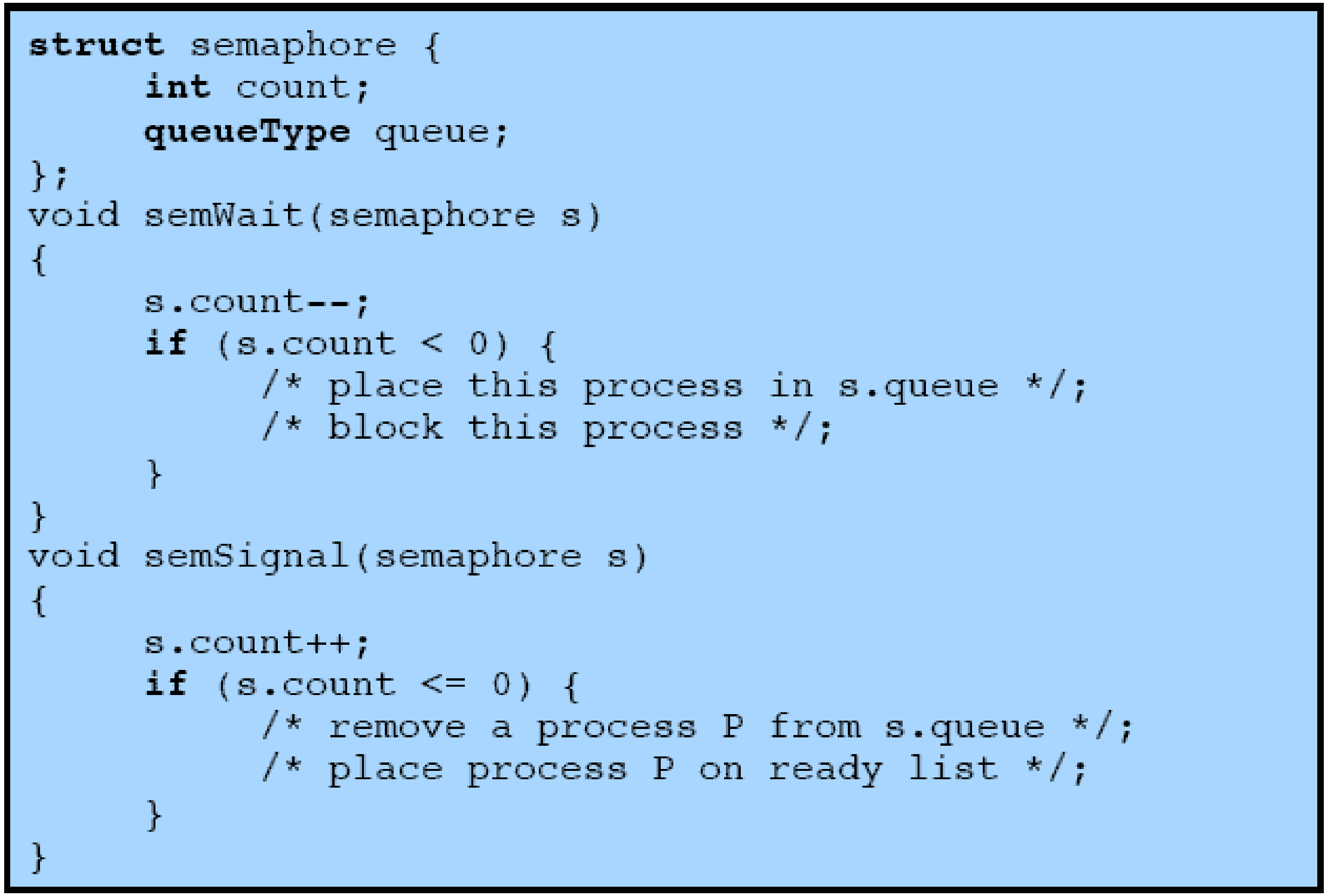
Hardware Mutual Exclusion: Disadvantages

- Busy-waiting consumes processor time
- Starvation is possible when a process leaves a critical section and more than one process is waiting.
  - Some process could indefinitely be denied access.
- Deadlock is possible

Semaphore

- Semaphore:
  - An integer value used for signalling among processes.
- Only three operations may be performed on a semaphore, all of which are atomic:
  - initialize,
  - Decrement (`semWait`)
  - increment. (`semSignal`)

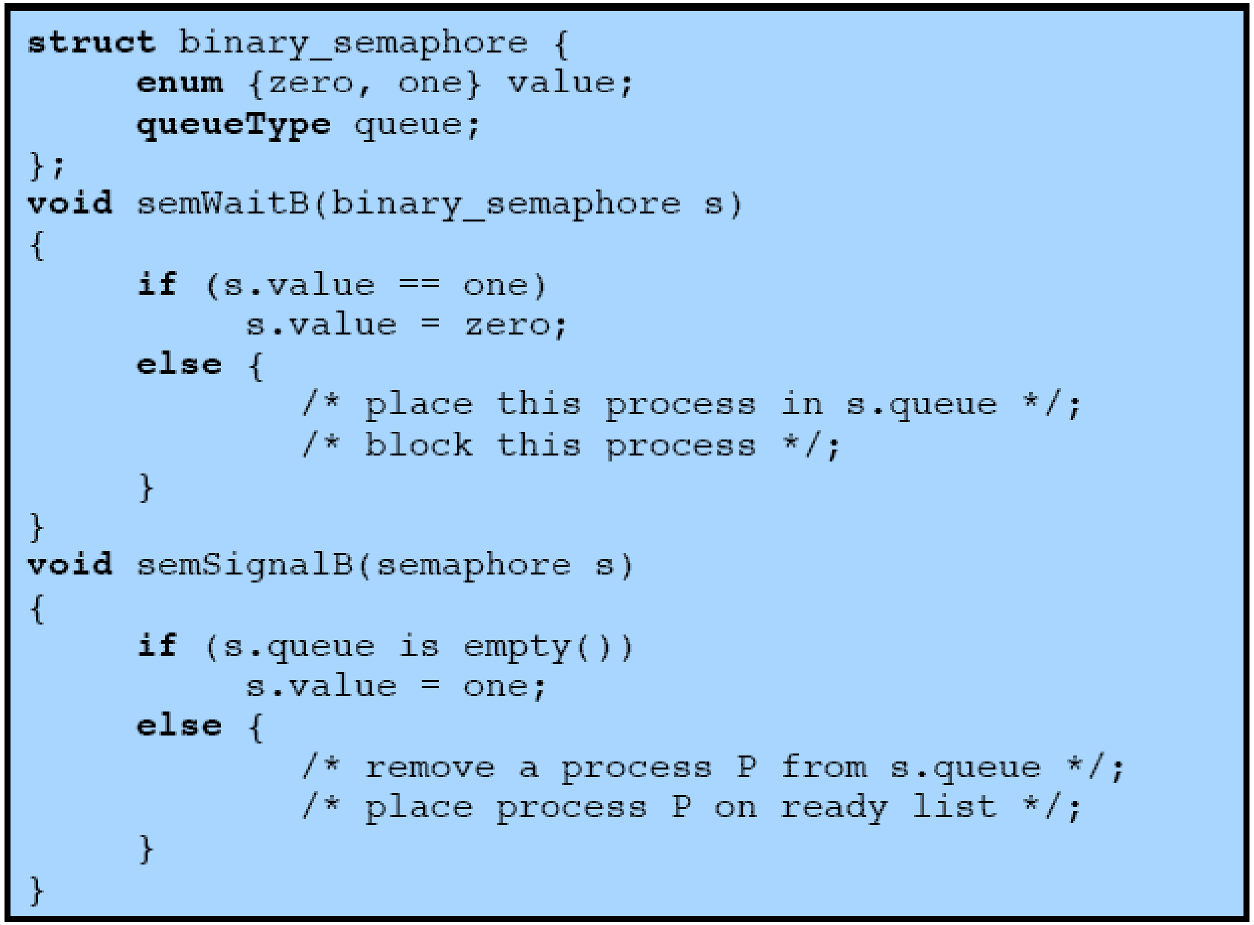
Semaphore Primitives



```
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

Binary Semaphore Primitives



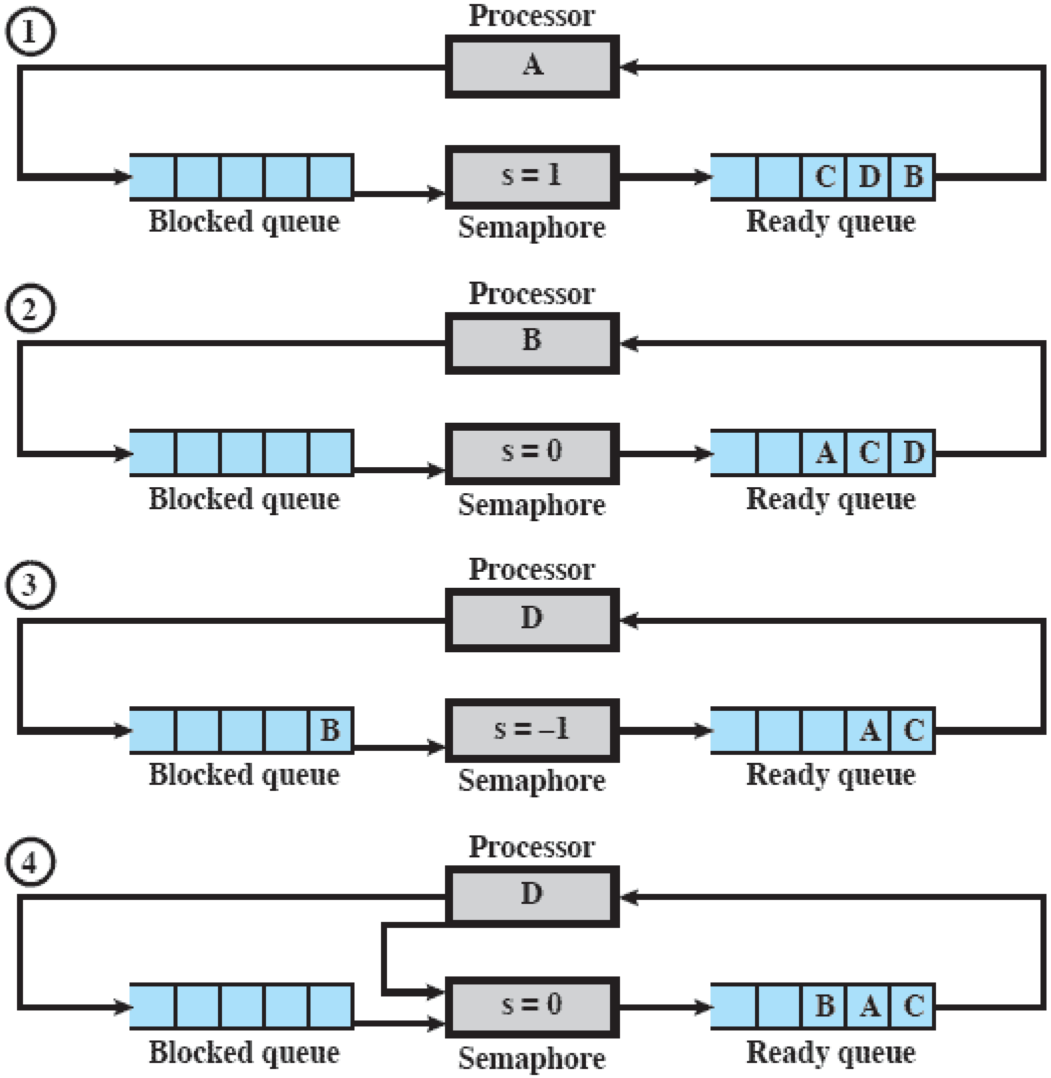
```
struct binary_semaphore {
     enum {zero, one} value;
     queueType queue;
};
void semWaitB(binary_semaphore s)
{
     if (s.value == one)
          s.value = zero;
     else {
            /* place this process in s.queue */;
            /* block this process */;
     }
}
void semSignalB(semaphore 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

Strong/Weak Semaphore

- A queue is used to hold processes waiting on the semaphore
  - In what order are processes removed from the queue?
- ***Strong Semaphores*** use FIFO
- ***Weak Semaphores*** don’t specify the order of removal from the queue

Example of Strong Semaphore Mechanism



Example of Semaphore Mechanism

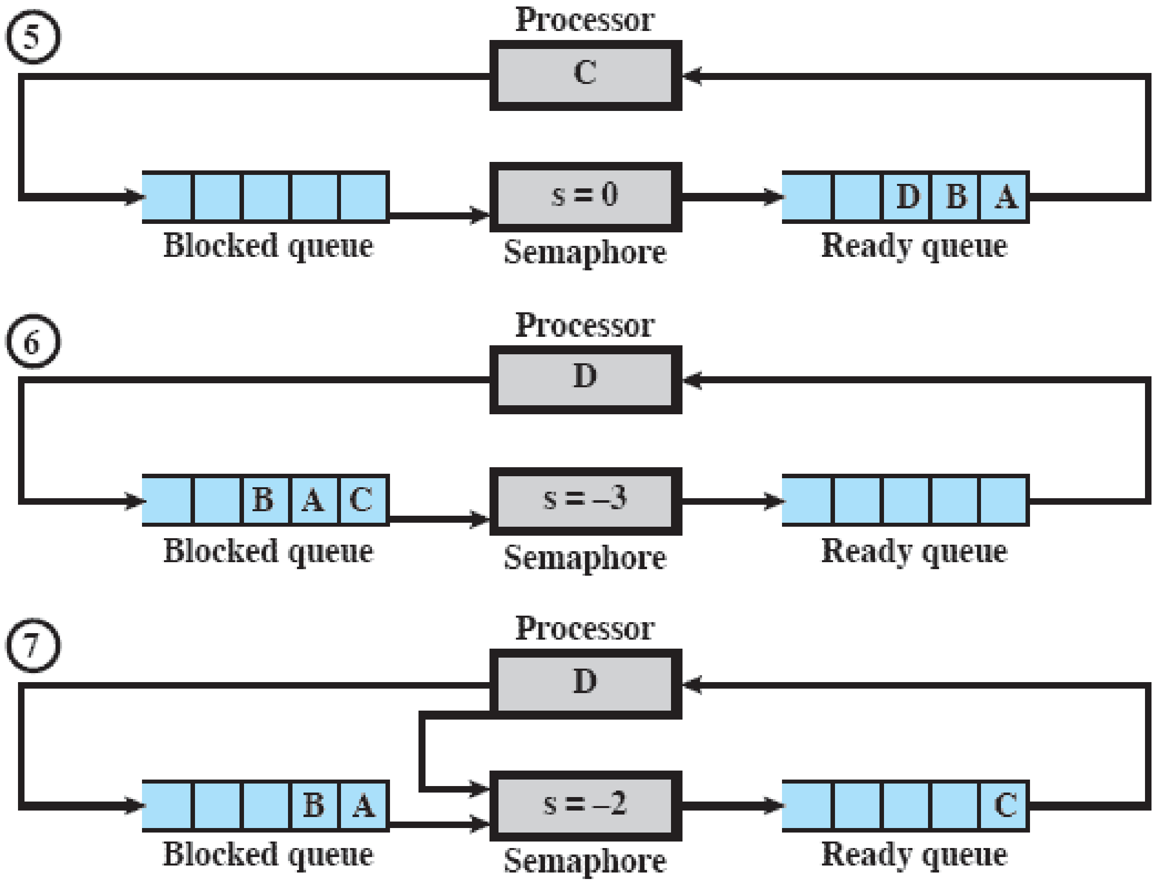
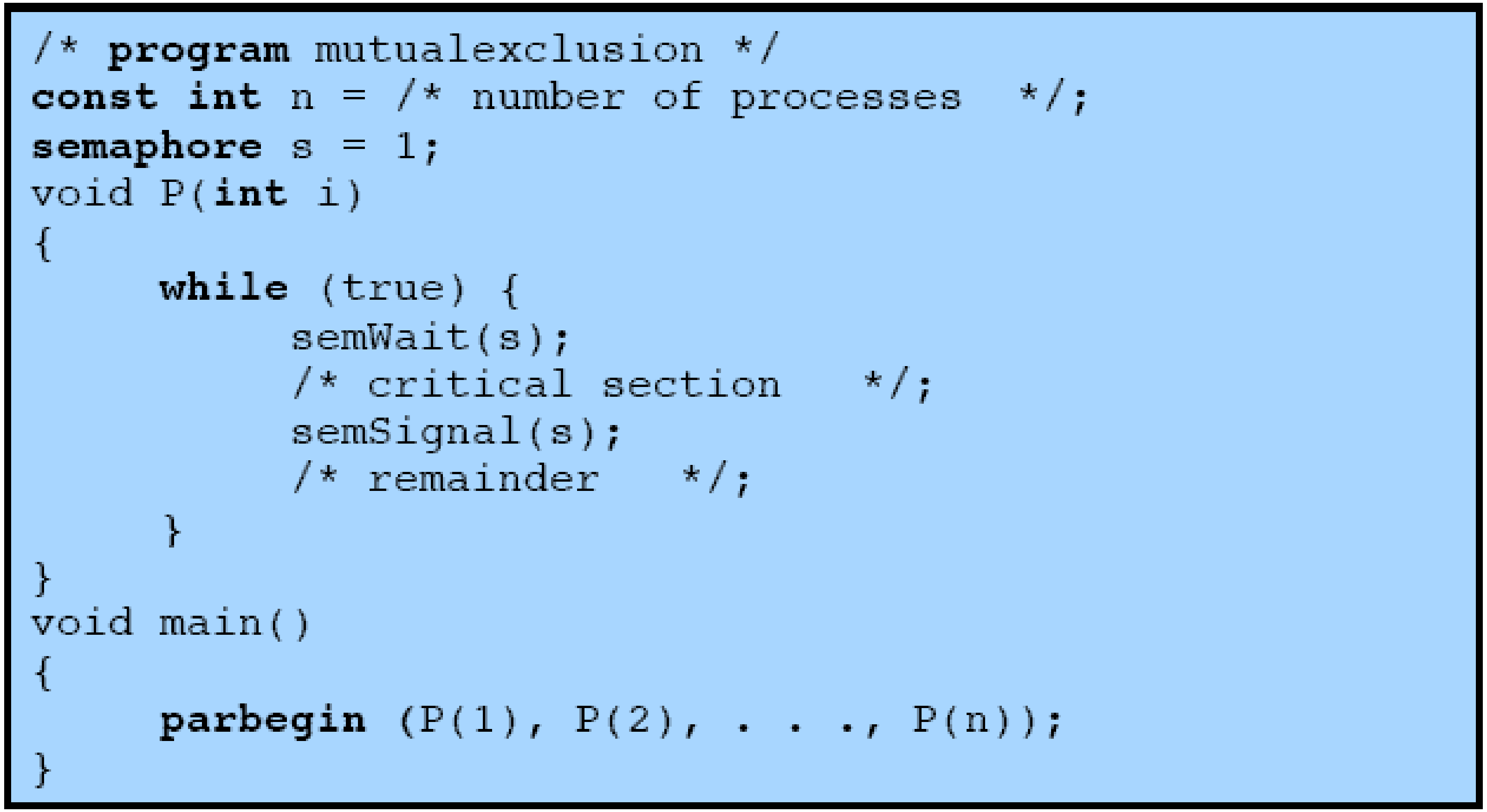


Figure 5.5 Example of Semaphore Mechanism

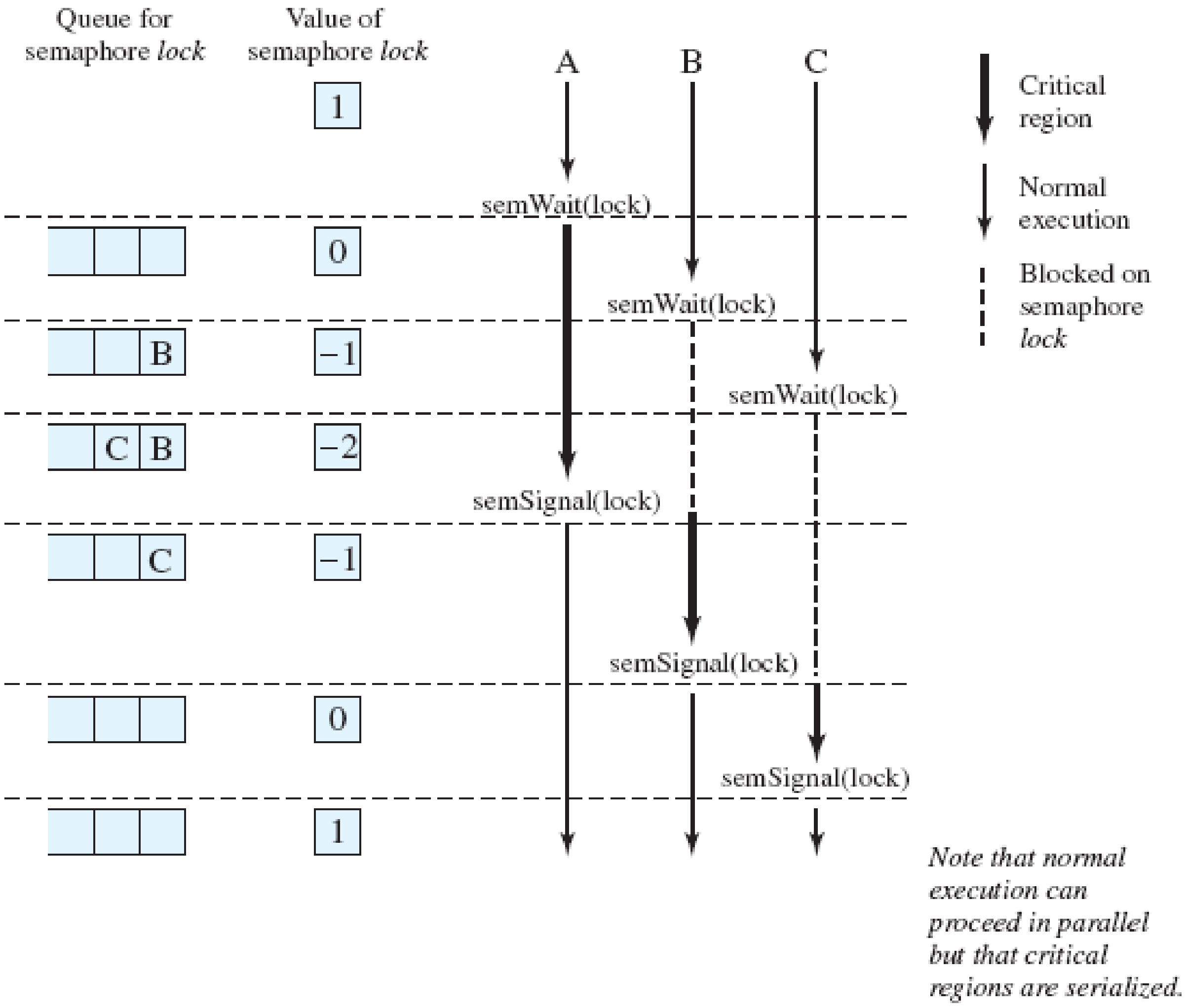
Mutual Exclusion Using Semaphores



```
/* 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



Note that normal execution can proceed in parallel but that critical regions are serialized.

Figure 5.7 **Processes Accessing Shared Data Protected by a Semaphore**

Producer/Consumer Problem

- General Situation:
  - 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
- The Problem:
  - Ensure that the Producer can’t add data into full buffer and consumer can’t remove data from empty buffer

Producer/Consumer Animation

Functions

- Assume an infinite buffer ***b*** with a linear array of elements

Producer

```
while (true) {
      /* produce item v */
      b[in] = v;
      in++;
}
```

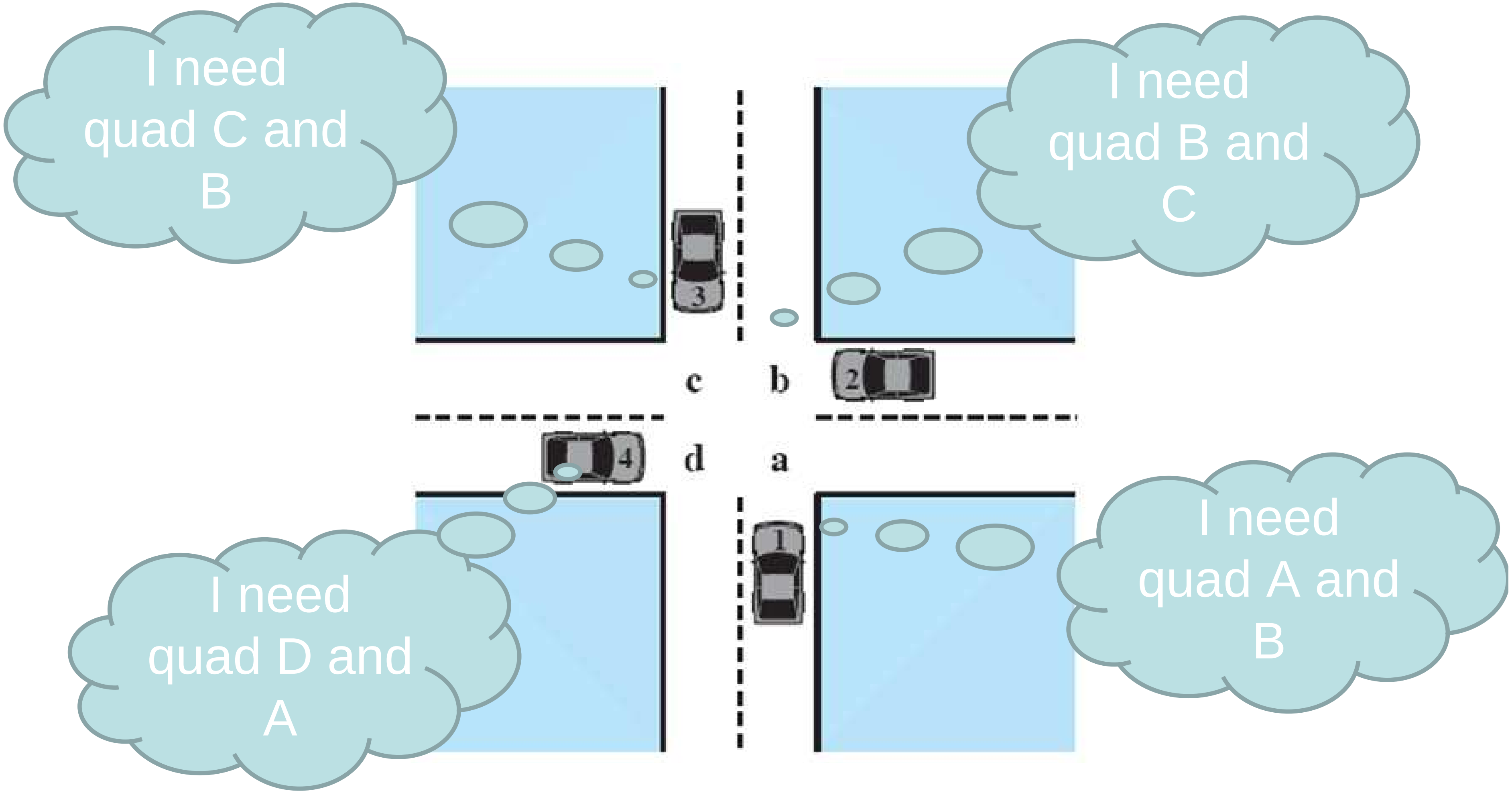
Consumer

```
while (true) {
       while (in <= out)
      /*do  nothing */;
      w = b[out];
      out++;
      /* consume item w */
}
```

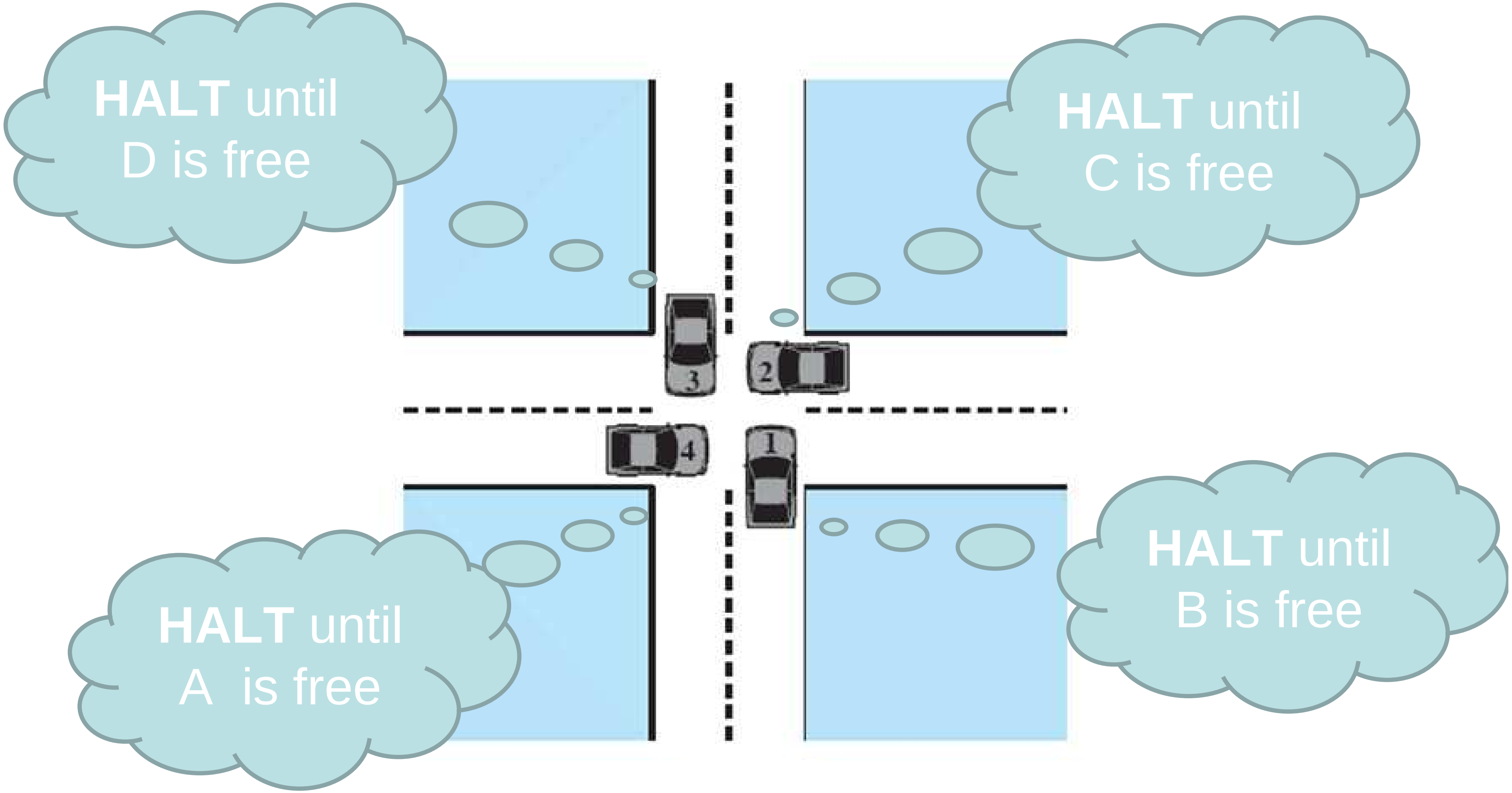
Deadlock

- A set of processes is deadlocked when each process in the set is blocked awaiting an event that can only be triggered by another blocked process in the set
  - Typically involves processes competing for the same set of resources
- No efficient solution

Potential Deadlock



Actual Deadlock



HALT until
D is free

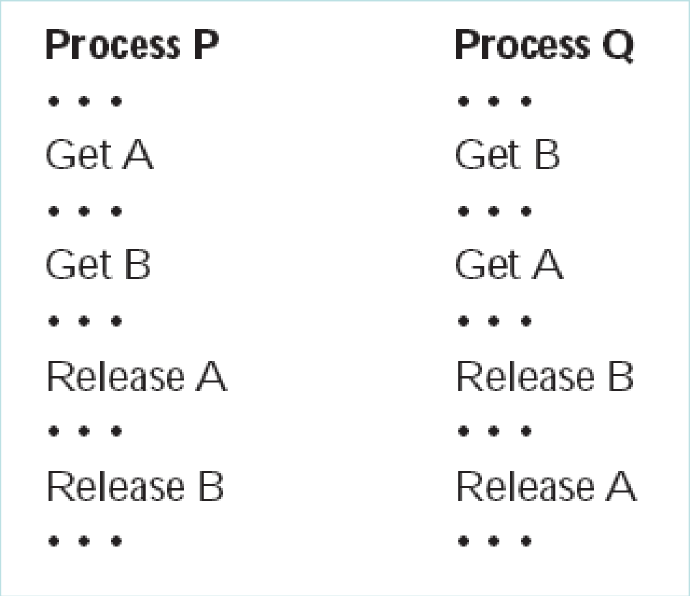
HALT until
C is free

HALT until
A is free

HALT until
B is free

Two Processes P and Q

- Lets look at this with two processes P and Q
- Each needing exclusive access to a resource A and B for a period of time



Process P	**Process Q**
• • •	• • •
Get A	Get B
• • •	• • •
Get B	Get A
• • •	• • •
Release A	Release B
• • •	• • •
Release B	Release A
• • •	• • •

Joint Progress Diagram of Deadlock

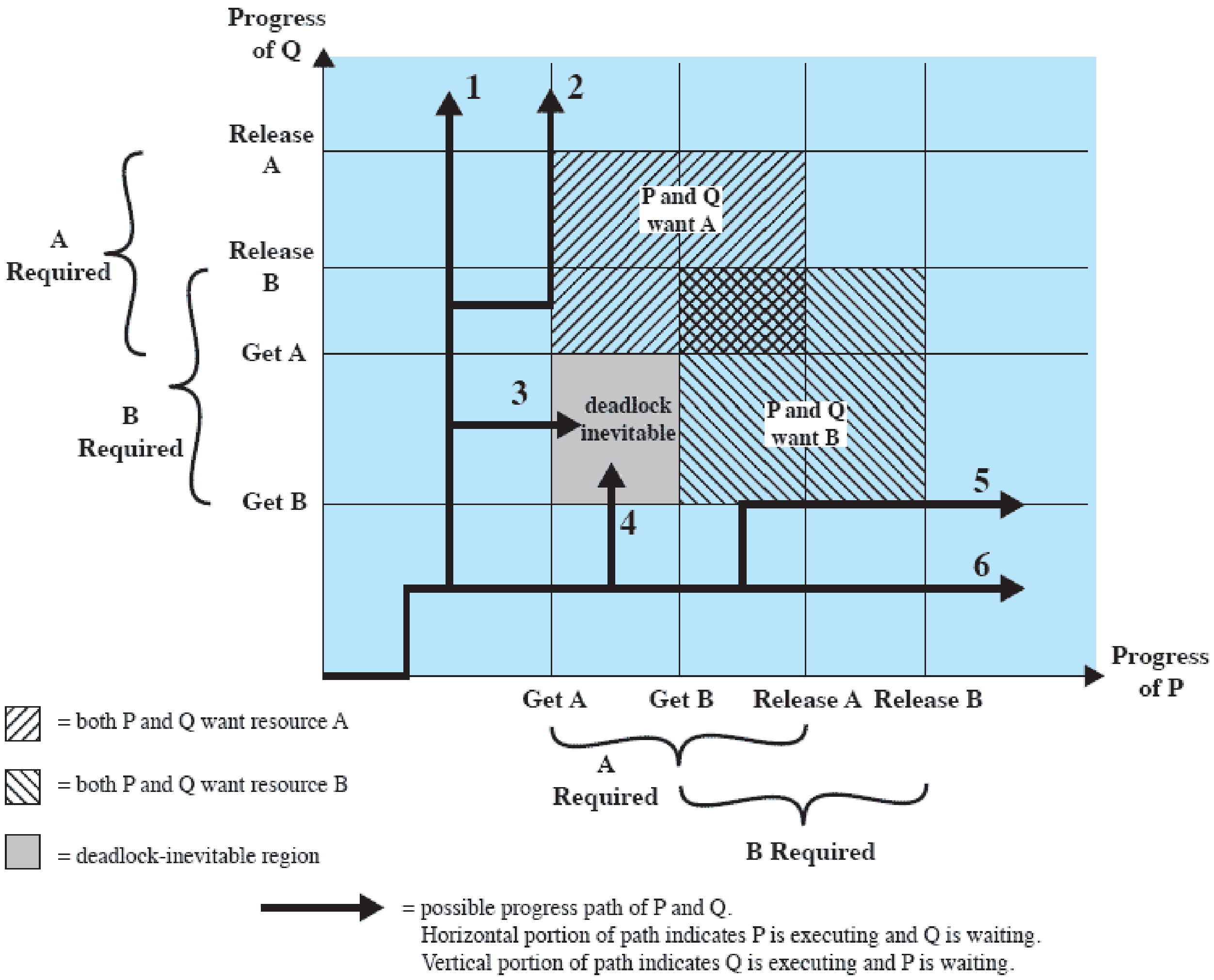
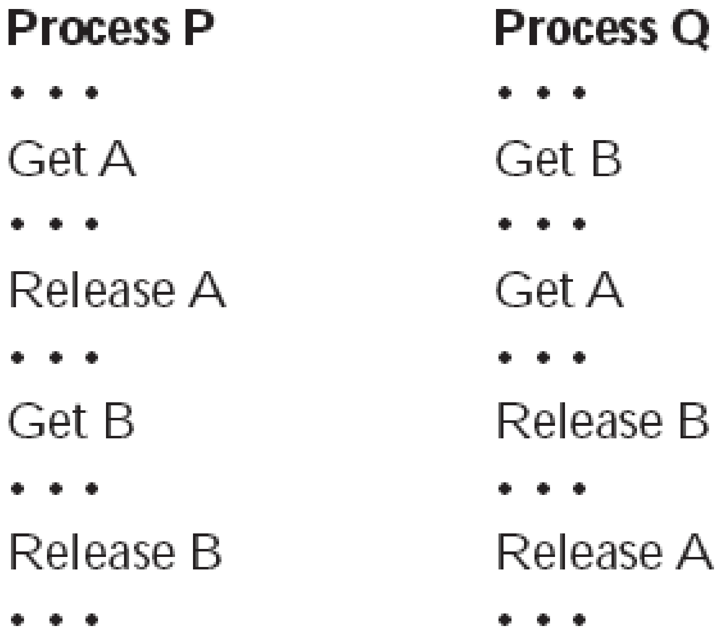


Figure 6.2 Example of Deadlock

Alternative logic

- Suppose that P does not need both resources at the same time so that the two processes have this form



Process P	Process Q
• • •	• • •
Get A	Get B
• • •	• • •
Release A	Get A
• • •	• • •
Get B	Release B
• • •	• • •
Release B	Release A
• • •	• • •

Diagram of alternative logic

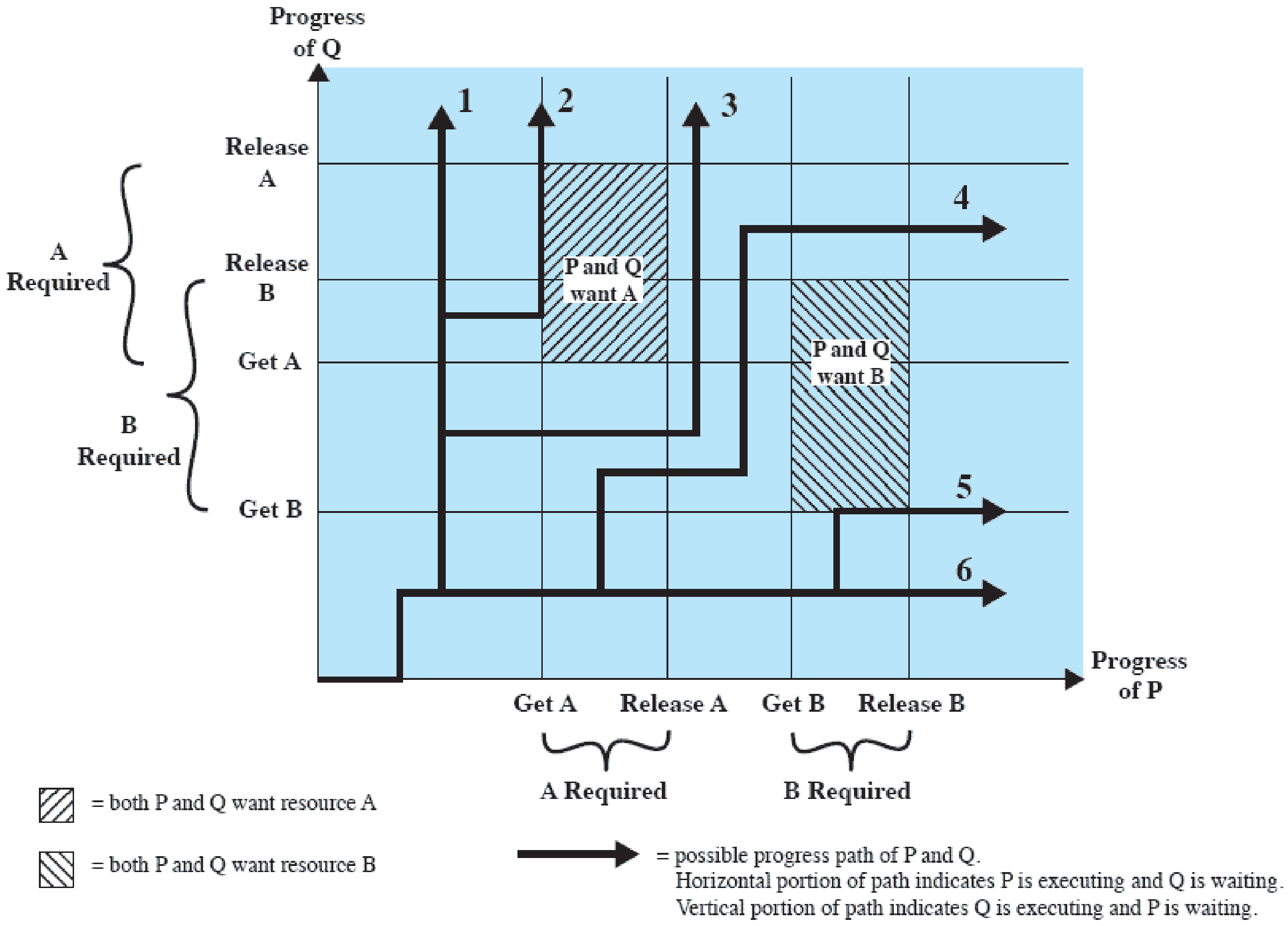


Figure 6.3 Example of No Deadlock [BACO03]

Resource Categories

Two general categories of resources:

- Reusable
  - can be safely used by only one process at a time and ***is not depleted*** by that use.
- Consumable
  - one that can be created (***produced***) and destroyed (***consumed***).

Reusable Resources

- Such as:
  - Processors, I/O channels, main and secondary memory, devices, and data structures such as files, databases, and semaphores
- Deadlock occurs if each process holds one resource and requests the other

Consumable Resources

- Such as Interrupts, signals, messages, and information in I/O buffers
- Deadlock may occur if a Receive message is blocking
- May take a rare combination of events to cause deadlock

Resource Allocation Graphs

- Directed graph that depicts a state of the system of resources and processes

(a) Resouce is requested

(b) Resource is held

Conditions for ***possible*** Deadlock

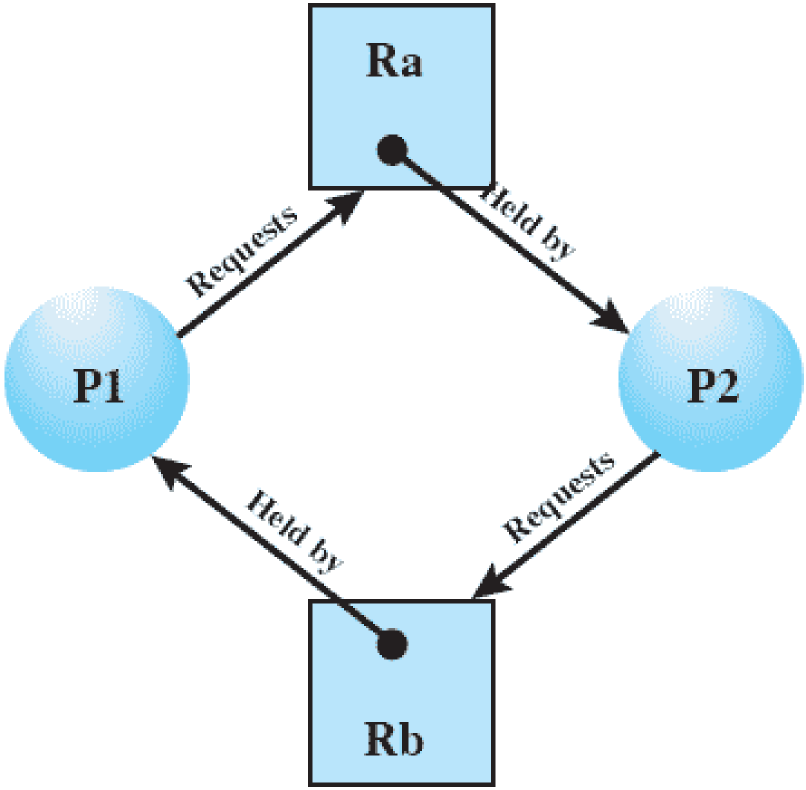
- Mutual exclusion
  - Only one process may use a resource at a time
- Hold-and-wait
  - A process may hold allocated resources while awaiting assignment of others
- No pre-emption
  - No resource can be forcibly removed form a process holding it

Actual Deadlock Requires …

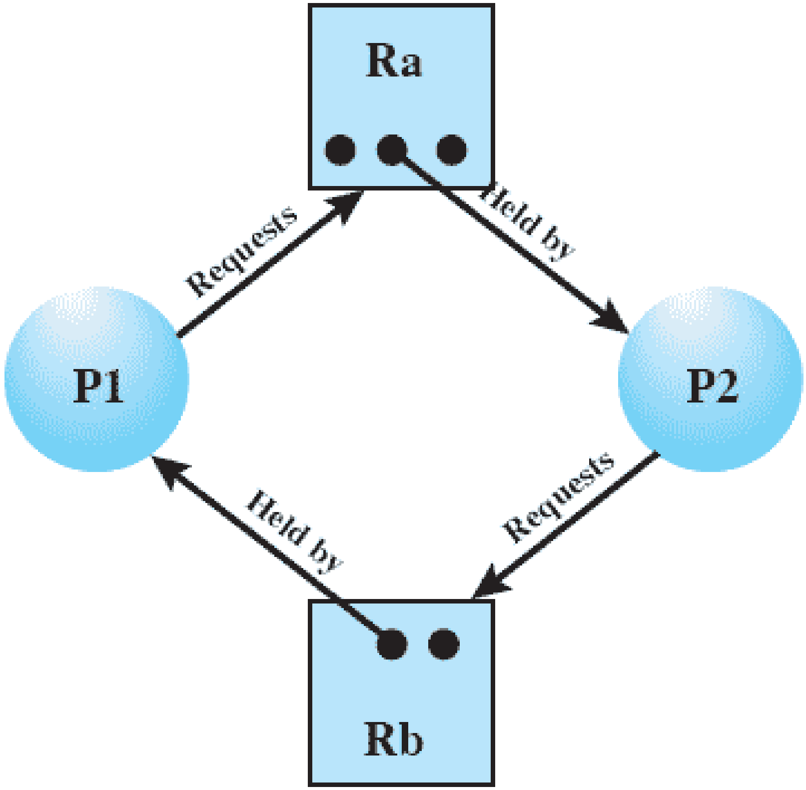
All previous 3 conditions plus:

- 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

Resource Allocation Graphs of deadlock



(c) Circular wait



(d) No deadlock

Resource Allocation Graphs

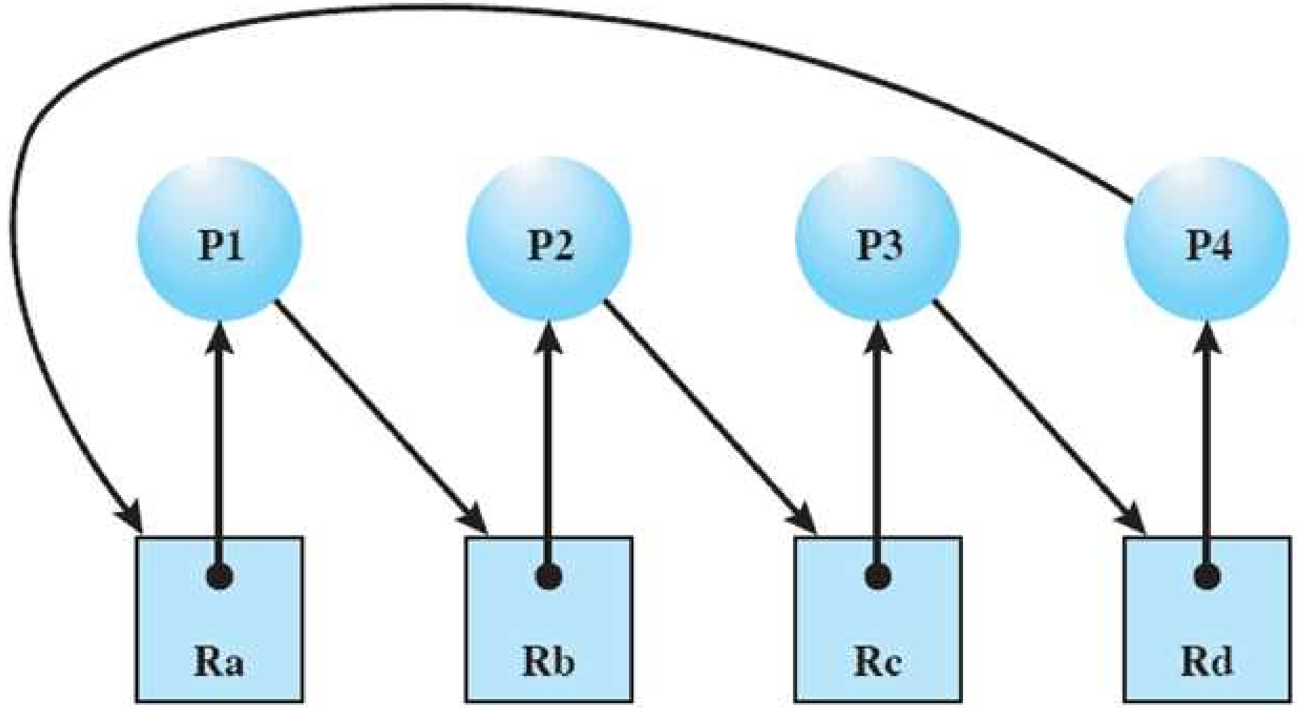


Figure 6.6 Resource Allocation Graph for Figure 6.1b

Dealing with Deadlock

- Three general approaches exist for dealing with deadlock.
  - Prevent deadlock
  - Avoid deadlock
  - Detect Deadlock

Deadlock Prevention Strategy

- Design a system in such a way that the possibility of deadlock is excluded.
- Two main methods
  - Indirect – prevent all three of the necessary conditions occurring at once
  - Direct – prevent circular waits

Deadlock Prevention Conditions 1 & 2

- Mutual Exclusion
  - Must be supported by the OS
- Hold and Wait
  - Require a process request all of its required resources at one time

Deadlock Prevention Conditions 3 & 4

- No Preemption
  - Process must release resource and request again
  - OS may preempt a process to require it releases its resources

- Circular Wait
  - Define a linear ordering of resource types

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

Two Approaches to Deadlock Avoidance

- Process Initiation Denial
  - Do not start a process if its demands might lead to deadlock

- Resource Allocation Denial
  - Do not grant an incremental resource request to a process if this allocation might lead to deadlock

Process
Initiation Denial

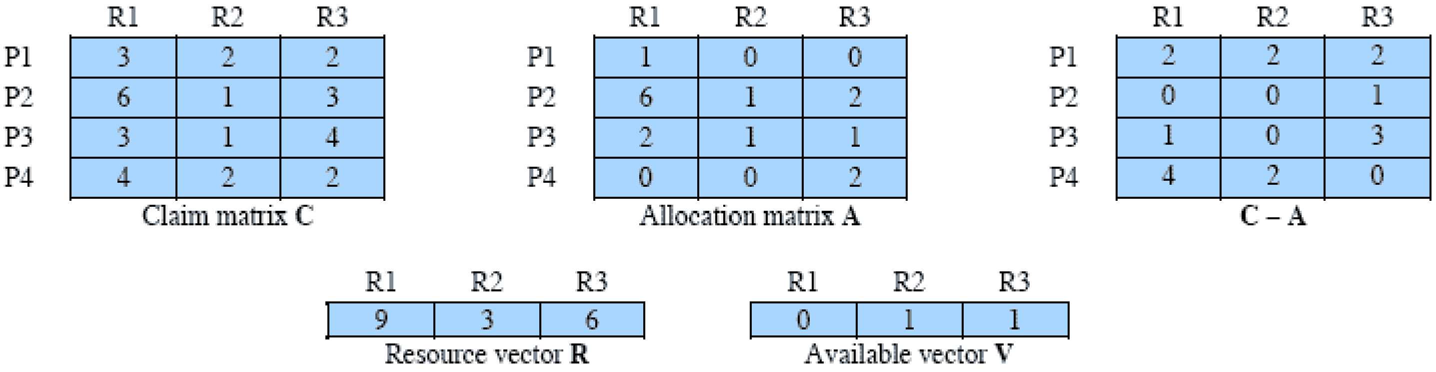
- A process is only started if the maximum claim of all current processes plus those of the new process can be met.
- Not optimal,
  - Assumes the worst: that all processes will make their maximum claims together.

Resource Allocation Denial

- Referred to as the banker’s algorithm
  - A strategy of resource allocation denial
- Consider a system with fixed number of resources
  - ***State*** of the system is the current allocation of resources to process
  - ***Safe state*** is where there is at least one sequence that does not result in deadlock
  - ***Unsafe state*** is a state that is not safe

Determination of Safe State

- A system consisting of four processes and three resources.
- Allocations are made to processors
- ***Is this a safe state?***



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

Claim matrix **C**

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

Allocation matrix **A**

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

C – A

R1	R2	R3
9	3	6

Resource vector **R**

R1	R2	R3
0	1	1

Available vector **V**

(a) **Initial state**

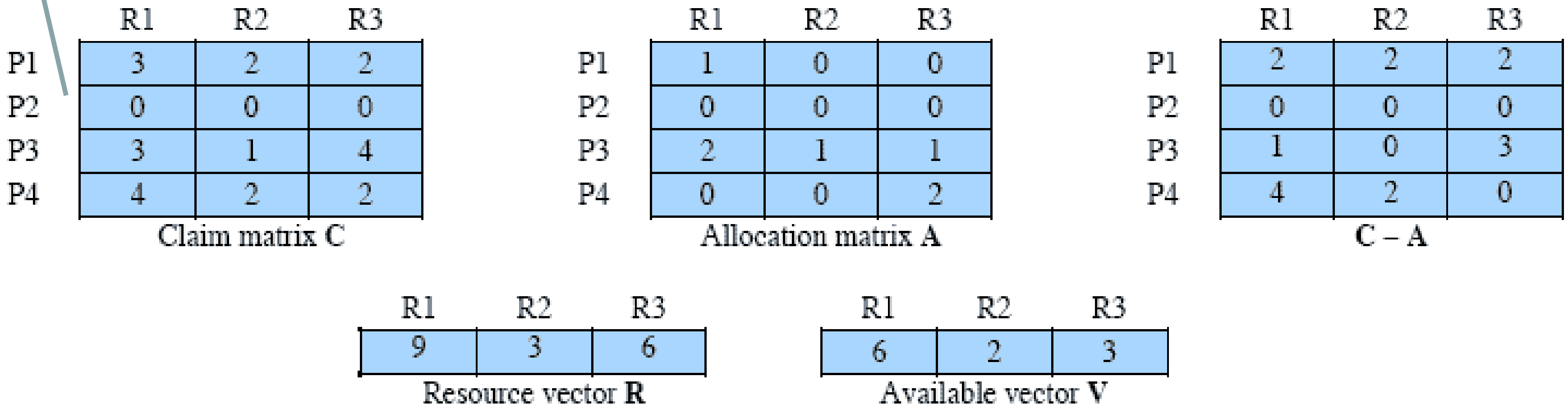
Process *i*

- $C_{ij} - A_{ij} \leq V_j$, for all j
- This is not possible for P1,
  - which has only 1 unit of R1 and requires 2 more units of R1, 2 units of R2, and 2 units of R3.
- If we assign one unit of R3 to process P2,
  - Then P2 has its maximum required resources allocated and can run to completion and return resources to ‘available’ pool

After P2 runs to completion

- Can any of the remaining processes can be completed?

Note P2 is completed



	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

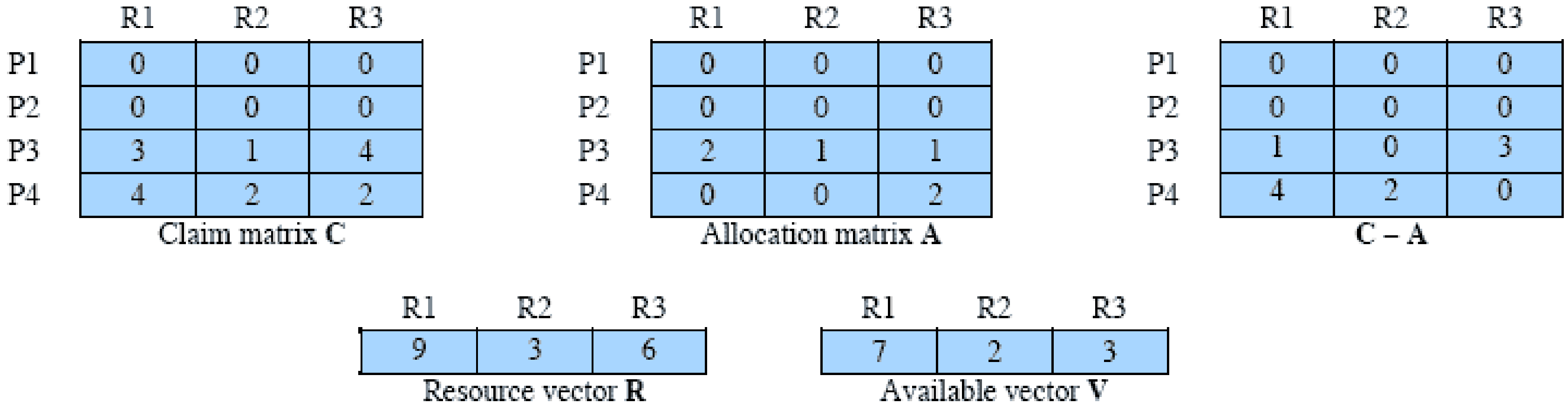
Resource vector **R**

R1	R2	R3
6	2	3

Available vector **V**

(b) **P2 runs to completion**

After P1 completes



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

Claim matrix **C**

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

Allocation matrix **A**

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

C – A

R1	R2	R3
9	3	6

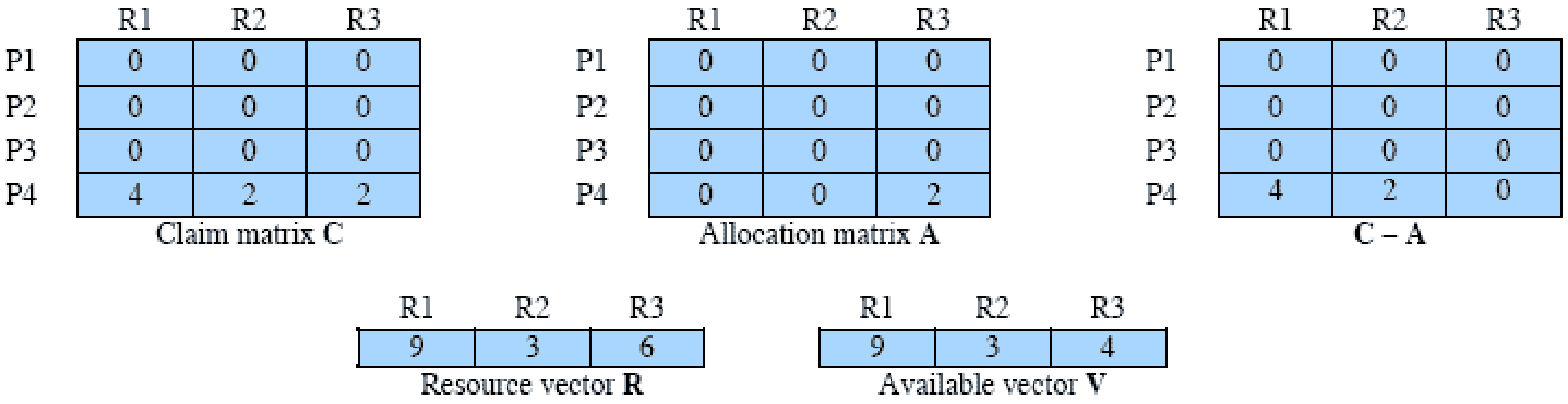
Resource vector **R**

R1	R2	R3
7	2	3

Available vector **V**

(c) **P1 runs to completion**

P3 Completes



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

Claim matrix **C**

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

Allocation matrix **A**

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

C – A

R1	R2	R3
9	3	6

Resource vector **R**

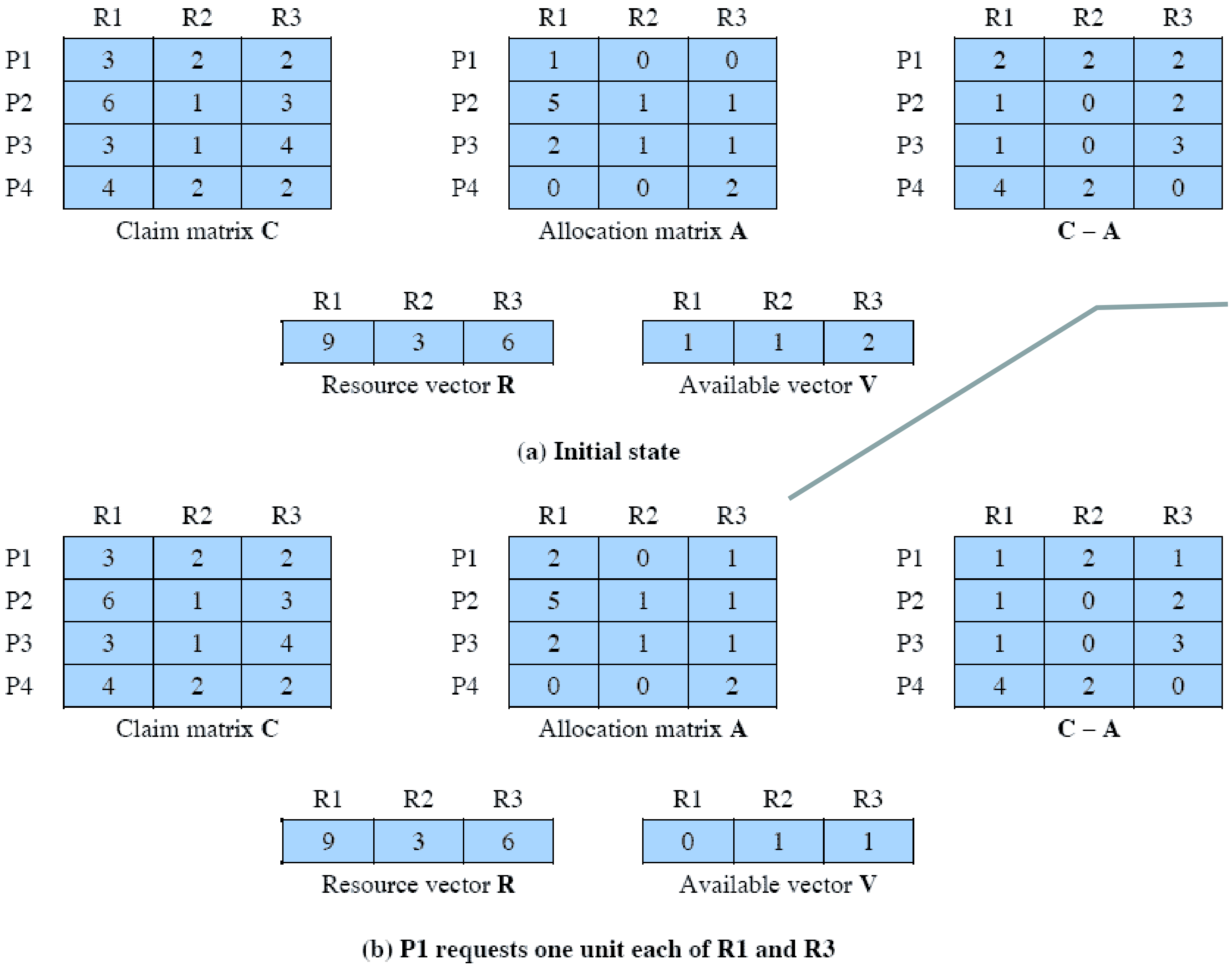
R1	R2	R3
9	3	4

Available vector **V**

(d) P3 runs to completion

Thus, the state defined originally is a safe state.

Determination of an Unsafe State



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

Claim matrix **C**

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

Allocation matrix **A**

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

C – A

R1	R2	R3
9	3	6

Resource vector **R**

R1	R2	R3
1	1	2

Available vector **V**

(a) **Initial state**

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

Claim matrix **C**

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

Allocation matrix **A**

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

C – A

R1	R2	R3
9	3	6

Resource vector **R**

R1	R2	R3
0	1	1

Available vector **V**

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

This time Suppose that P1 makes the request for one additional unit each of R1 and R3.
Is this safe?

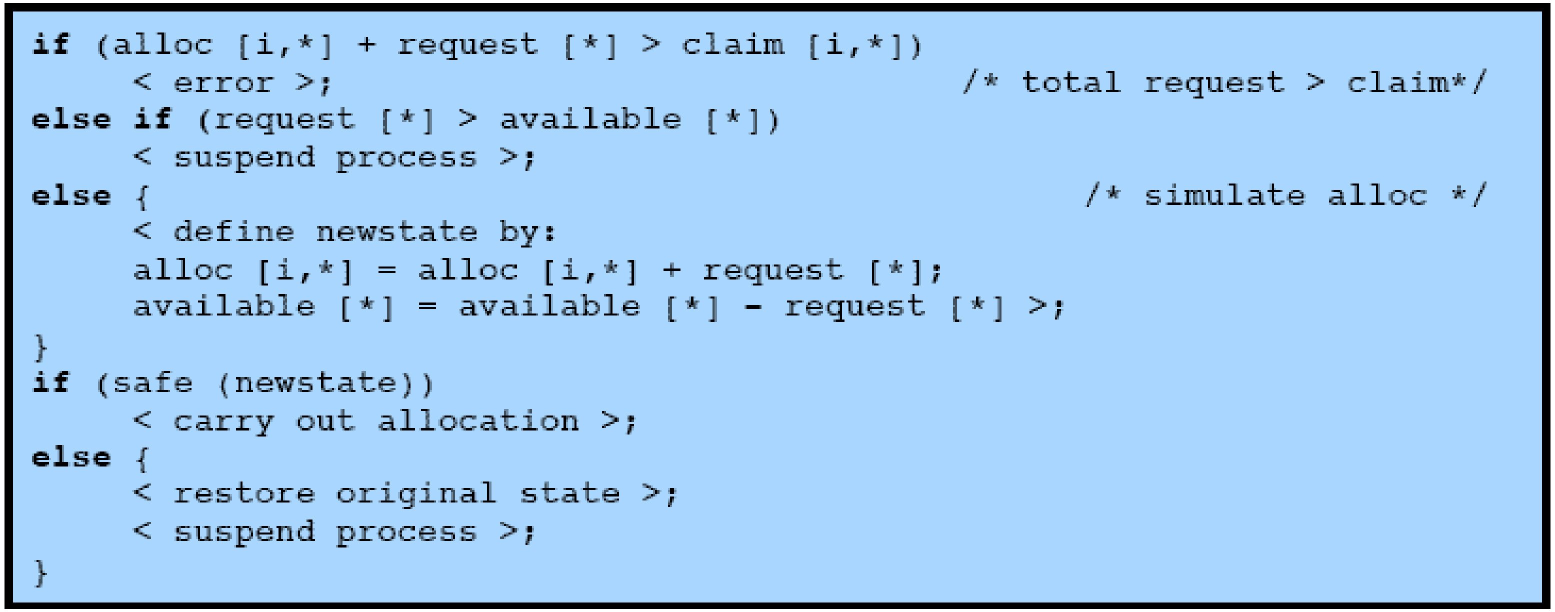
Deadlock Avoidance

- When a process makes a request for a set of resources,
    - assume that the request is granted,
    - Update the system state accordingly,
- Then determine if the result is a safe state.
    - If so, grant the request and,
    - if not, block the process until it is safe to grant the request.

Deadlock Avoidance Logic

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

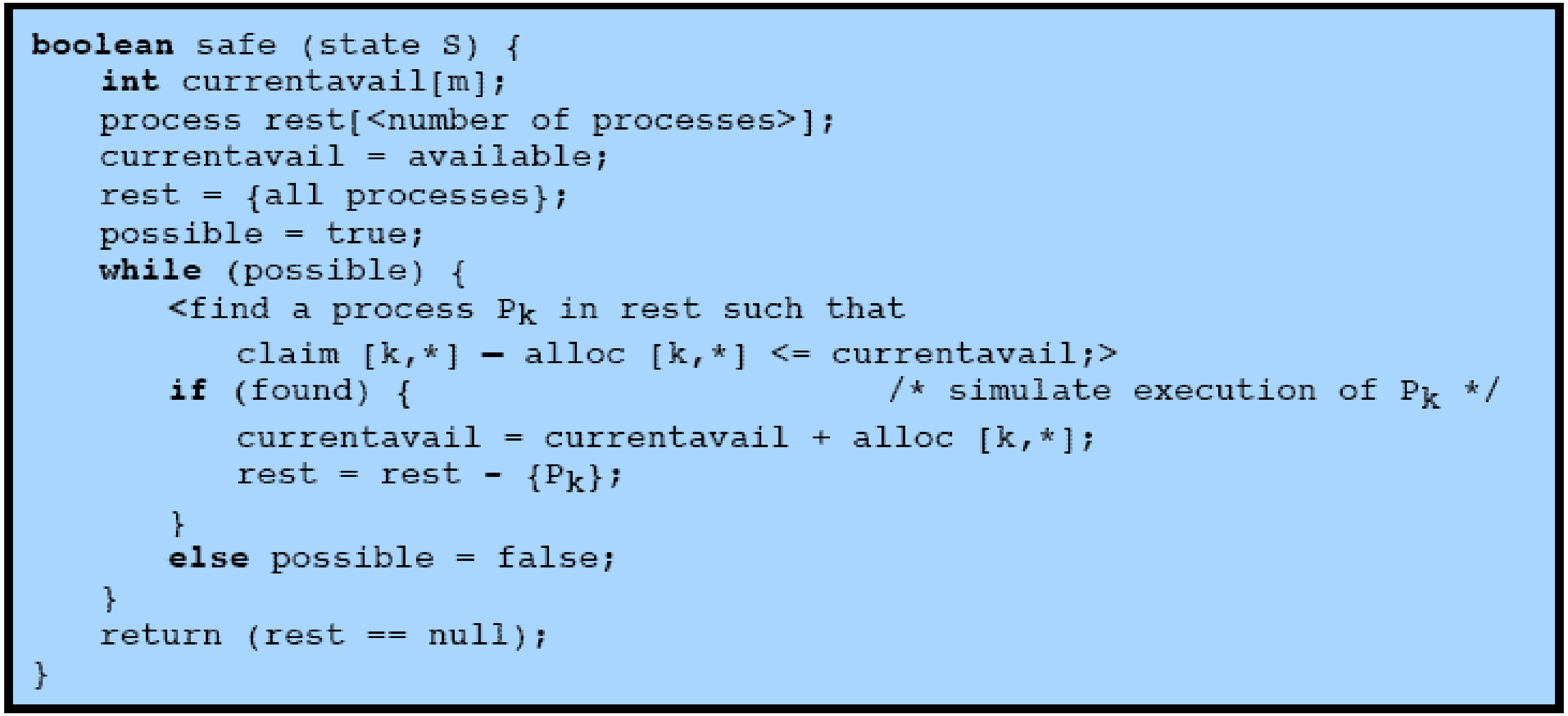
(a) global data structures



```
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 $P_k$ in rest such that
            claim [k,*] – alloc [k,*] <= currentavail;>
        if (found) {                             /* simulate execution of $P_k$ */
            currentavail = currentavail + alloc [k,*];
            rest = rest - {$P_k$};
        }
        else possible = false;
    }
    return (rest == null);
}
```

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

Figure 6.9 Deadlock Avoidance Logic

Deadlock Avoidance Advantages

- It is not necessary to preempt and rollback processes, as in deadlock detection,
- It is less restrictive than deadlock prevention.

Deadlock Avoidance Restrictions

- Maximum resource requirement must be stated in advance
- Processes under consideration must be independent and with no synchronization requirements
- There must be a fixed number of resources to allocate
- No process may exit while holding resources

Roadmap

- Principals of Deadlock
  - Deadlock prevention
  - Deadlock Avoidance
  - → Deadlock detection
  - → An Integrated deadlock strategy
- Dining Philosophers Problem
- Concurrency Mechanisms in UNIX, Linux, Solaris and Windows

Deadlock Detection

- Deadlock prevention strategies are very conservative;
  - limit access to resources and impose restrictions on processes.
- Deadlock detection strategies do the opposite
  - Resource requests are granted whenever possible.
  - Regularly check for deadlock

A Common
Detection Algorithm

- Use a Allocation matrix and Available vector as previous
- Also use a request matrix ***Q***
  - Where ***Qij*** indicates that an amount of resource ***j*** is requested by process ***I***
- First ‘un-mark’ all processes that are not deadlocked
  - Initially that is all processes

Detection Algorithm

1. Mark each process that has a row in the Allocation matrix of all zeros.
2. Initialize a temporary vector **W** to equal the Available vector.
3. Find an index ***i*** such that process ***i*** is currently unmarked and the ***i***th row of Q is less than or equal to ***W***.
   - i.e. $Q_{ik} \leq W_k$ for $1 \leq k \leq m$.
   - If no such row is found, terminate

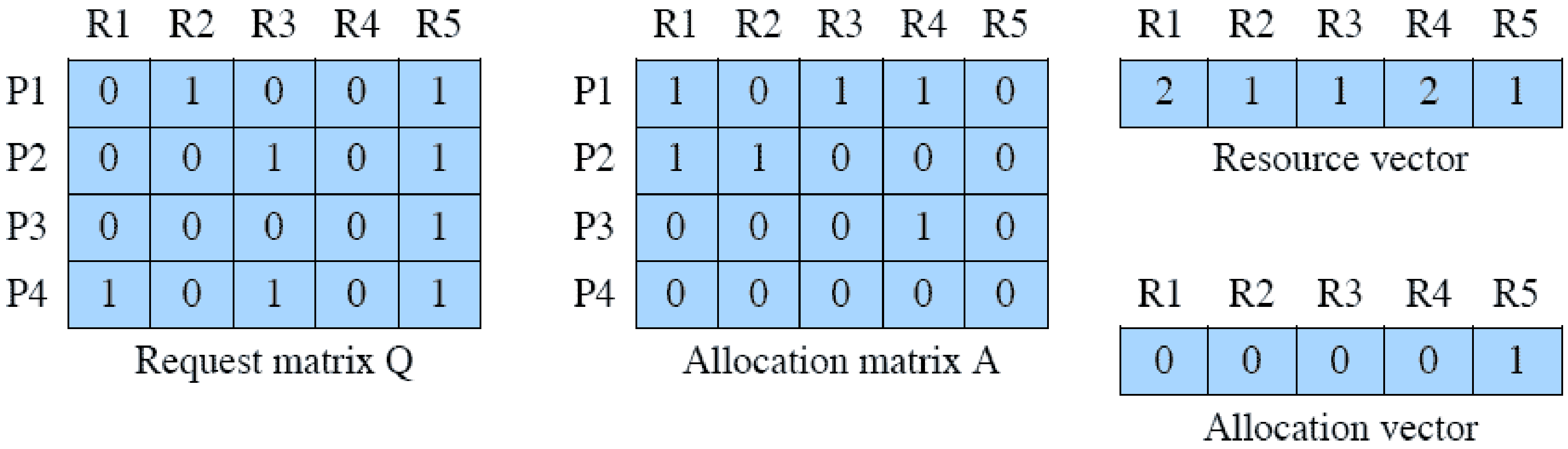
Detection Algorithm cont.

4. If such a row is found,
   - mark process *i* and add the corresponding row of the allocation matrix to W.
   - i.e. set $W_k = W_k + A_{ik}$, for $1 \le k \le m$

Return to step 3.

- A deadlock exists if and only if there are unmarked processes at the end
- Each unmarked process is deadlocked.

Deadlock Detection



	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

Recovery Strategies
Once Deadlock Detected

- Abort all deadlocked processes
- Back up each deadlocked process to some previously defined checkpoint, and restart all process
  - Risk or deadlock recurring
- Successively abort deadlocked processes until deadlock no longer exists
- Successively preempt resources until deadlock no longer exists

Advantages and Disadvantages

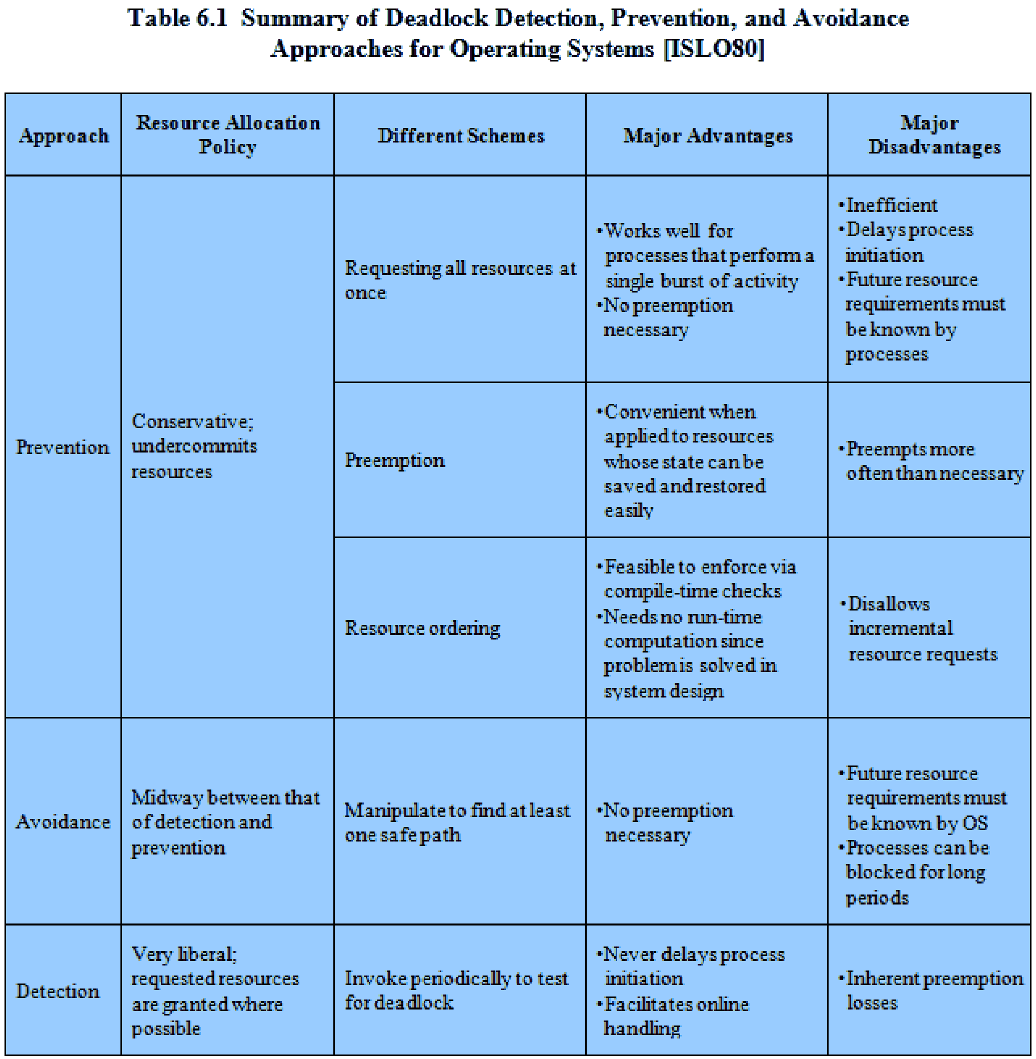


Table 6.1 Summary of Deadlock Detection, Prevention, and Avoidance Approaches for Operating Systems [ISLO80]

Approach	Resource Allocation Policy	Different Schemes	Major Advantages	Major Disadvantages
Prevention	Conservative; undercommits resources	Requesting all resources at once	•Works well for processes that perform a single burst of activity •No preemption necessary	•Inefficient •Delays process initiation •Future resource requirements must be known by processes
		Preemption	•Convenient when applied to resources whose state can be saved and restored easily	•Preempts more often than necessary
		Resource ordering	•Feasible to enforce via compile-time checks •Needs no run-time computation since problem is solved in system design	•Disallows incremental resource requests
Avoidance	Midway between that of detection and prevention	Manipulate to find at least one safe path	•No preemption necessary	•Future resource requirements must be known by OS •Processes can be blocked for long periods
Detection	Very liberal; requested resources are granted where possible	Invoke periodically to test for deadlock	•Never delays process initiation •Facilitates online handling	•Inherent preemption losses

Roadmap

- Principals of Deadlock
  - Deadlock prevention
  - Deadlock Avoidance
  - Deadlock detection
  - An Integrated deadlock strategy
- Dining Philosophers Problem
- Concurrency Mechanisms in UNIX, Linux, Solaris and Windows

Dining Philosophers Problem: Scenario

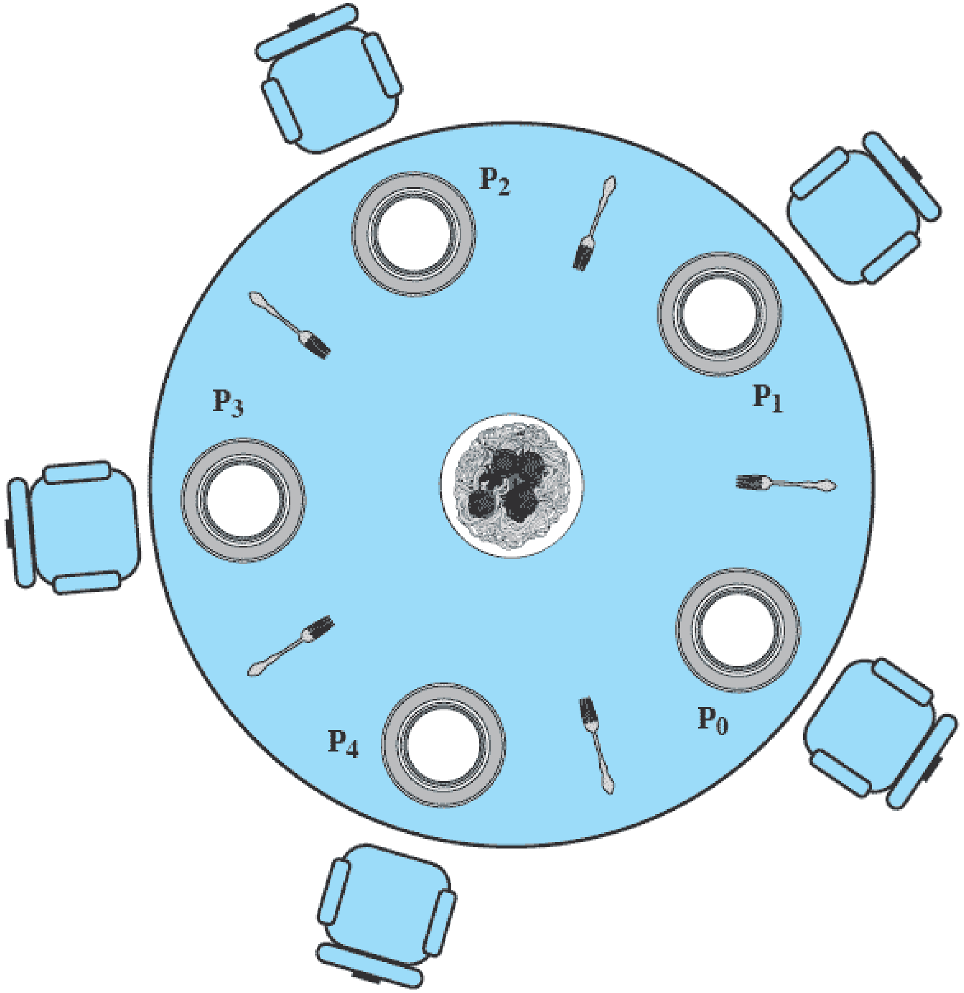
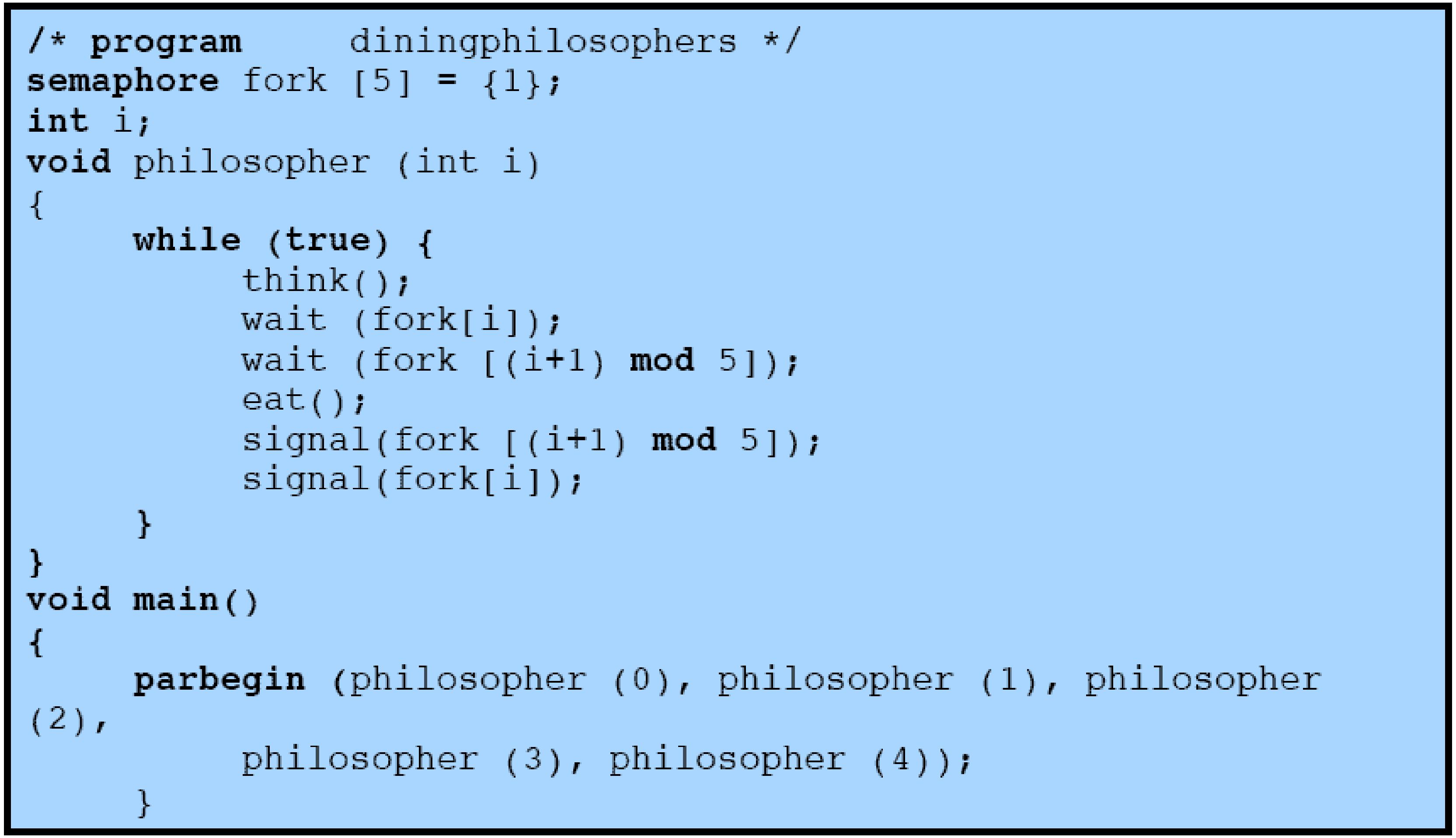


Figure 6.11 Dining Arrangement for Philosophers

The Problem

- Devise a ritual (algorithm) that will allow the philosophers to eat.
  - No two philosophers can use the same fork at the same time (mutual exclusion)
  - No philosopher must starve to death (avoid deadlock and starvation … literally!)

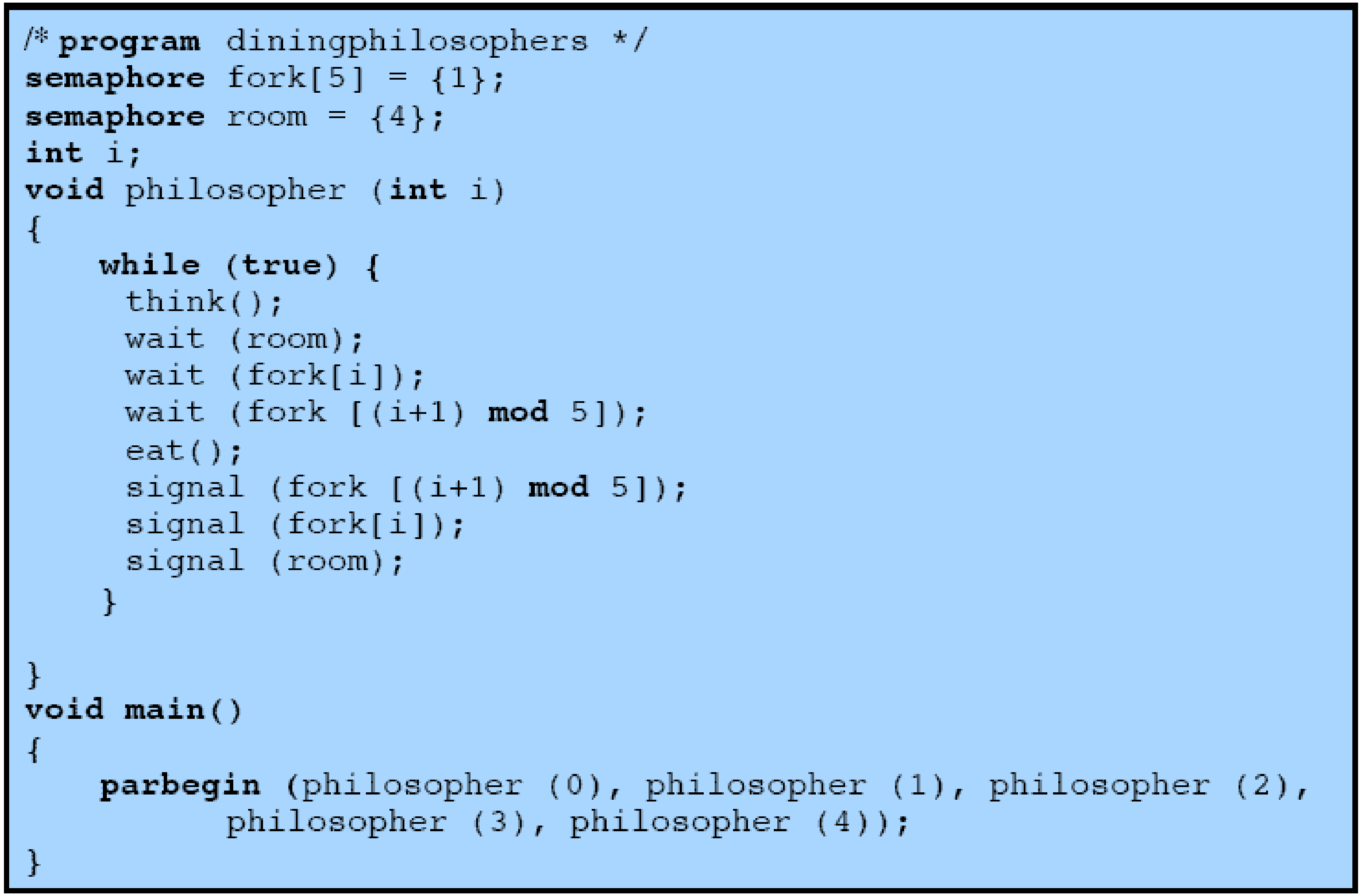
A first solution using semaphores



```
/* 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

Avoiding deadlock



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

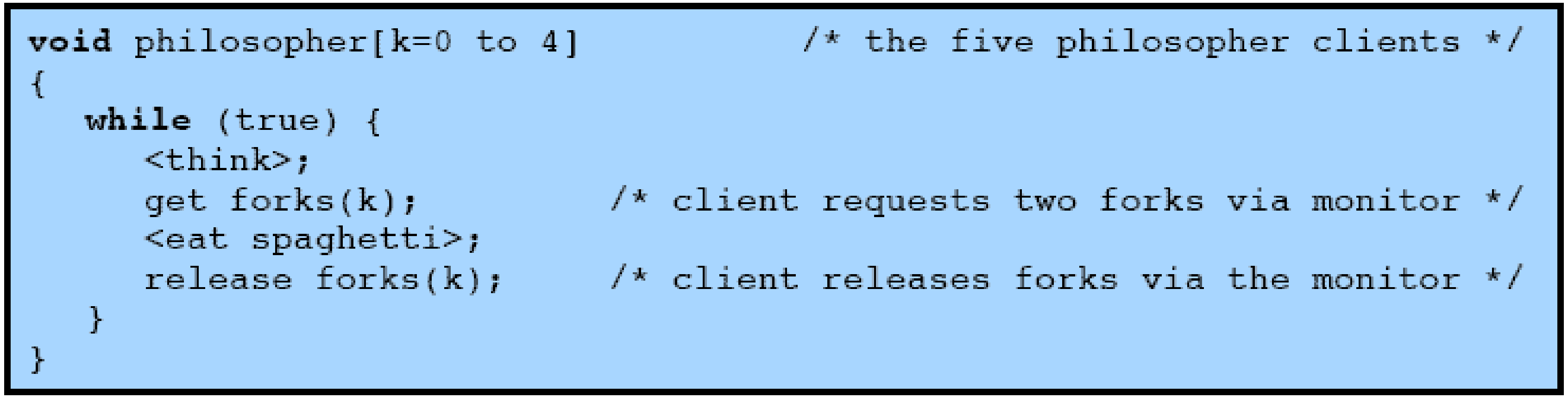
Figure 6.13   A Second Solution to the Dining Philosophers Problem

Solution using Monitors

```
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]);
}
```

Monitor solution cont.



```
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

Roadmap

- Principals of Deadlock
  - Deadlock prevention
  - Deadlock Avoidance
  - Deadlock detection
  - An Integrated deadlock strategy
- Dining Philosophers Problem

→ Concurrency Mechanisms in UNIX, Linux, Solaris and Windows

UNIX Concurrency Mechanisms

- UNIX provides a variety of mechanisms for interprocessor communication and synchronization including:
  - Pipes
  - Messages
  - Shared memory
  - Semaphores
  - Signals

Pipes

- A circular buffer allowing two processes to communicate on the producer-consumer model
  - first-in-first-out queue, written by one process and read by another.
- Two types:
  - Named:
  - Unnamed

Messages

- A block of bytes with an accompanying type.
- UNIX provides ***msgsnd*** and ***msgrcv*** system calls for processes to engage in message passing.
- Associated with each process is a message queue, which functions like a mailbox.

Shared Memory

- A common block of virtual memory shared by multiple processes.
- Permission is read-only or read-write for a process,
  - determined on a per-process basis.
- Mutual exclusion constraints are not part of the shared-memory facility but must be provided by the processes using the shared memory.

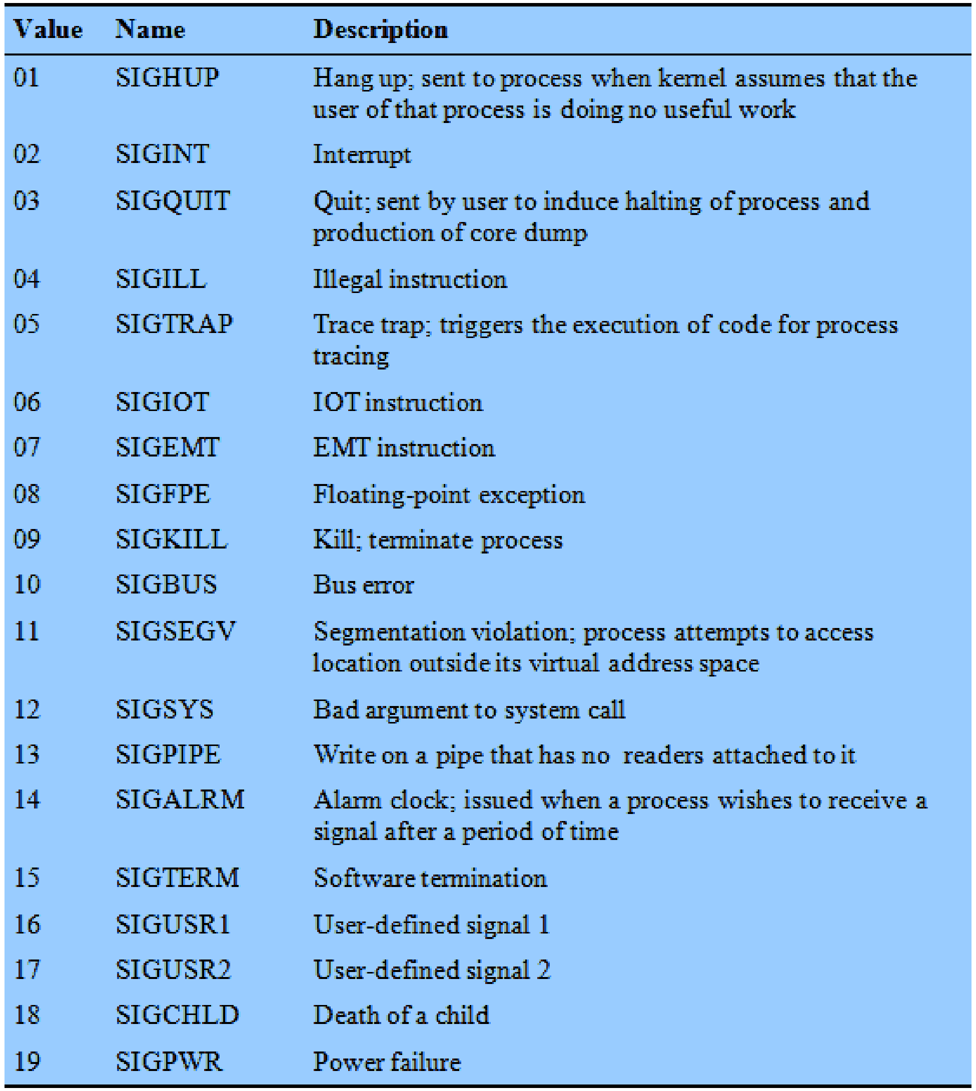
Semaphores

- SVR4 uses a generalization of the ***semWait*** and ***semSignal*** primitives defined in Chapter 5;
- Associated with the semaphore are queues of processes blocked on that semaphore.

Signals

- A software mechanism that informs a process of the occurrence of asynchronous events.
  - Similar to a hardware interrupt, without priorities
- A signal is delivered by updating a field in the process table for the process to which the signal is being sent.

Signals defined for UNIX SVR4.



Value	Name	Description
01	SIGHUP	Hang up; sent to process when kernel assumes that the user of that process is doing no useful work
02	SIGINT	Interrupt
03	SIGQUIT	Quit; sent by user to induce halting of process and production of core dump
04	SIGILL	Illegal instruction
05	SIGTRAP	Trace trap; triggers the execution of code for process tracing
06	SIGIOT	IOT instruction
07	SIGEMT	EMT instruction
08	SIGFPE	Floating-point exception
09	SIGKILL	Kill; terminate process
10	SIGBUS	Bus error
11	SIGSEGV	Segmentation violation; process attempts to access location outside its virtual address space
12	SIGSYS	Bad argument to system call
13	SIGPIPE	Write on a pipe that has no readers attached to it
14	SIGALRM	Alarm clock; issued when a process wishes to receive a signal after a period of time
15	SIGTERM	Software termination
16	SIGUSR1	User-defined signal 1
17	SIGUSR2	User-defined signal 2
18	SIGCHLD	Death of a child
19	SIGPWR	Power failure

MUTEX Lock

- A mutex is used to ensure only one thread at a time can access the resource protected by the mutex.
- The thread that locks the mutex must be the one that unlocks it.

Condition Variables

- A condition variable is used to wait until a particular condition is true.
- Condition variables must be used in conjunction with a mutex lock.