Concurrency & Deadlock

ROADMAP - Concurrency

- **Key terms**
- **Principles of concurrency :**
- **Requirement of mutual exclusion**
- Semaphores

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.

Concurrency

- **Concurrency** in operating systems refers to the ability of an OS to manage and execute multiple tasks or processes simultaneously. It allows multiple tasks to overlap in execution, giving the appearance of parallelism even on single-core processors. Concurrency is achieved through various techniques such as multitasking, multithreading, and multiprocessing.
- **Multitasking** involves the execution of multiple tasks by rapidly switching between them. Each task gets a time slot, and the OS switches between them so quickly that it seems as if they are running simultaneously.
- **Multithreading** takes advantage of modern processors with multiple cores. It allows different threads of a process to run on separate cores, enabling true parallelism within a single process.
- **Multiprocessing** goes a step further by distributing multiple processes across multiple physical processors or cores, achieving parallel execution at a higher level.

Principles of Concurrency in Operating Systems

Process Isolation:
Each process should have its own memory space and resources to prevent interference between processes. This isolation is critical to maintain system stability.

Synchronization:
Concurrency introduces the possibility of data races and conflicts. Synchronization mechanisms like locks, semaphores, and mutexes are used to coordinate access to shared resources and ensure data consistency.

Deadlock Avoidance:

OSs implement algorithms to detect and avoid deadlock situations where processes are stuck waiting for resources indefinitely. Deadlocks can halt the entire system.

Fairness:

The OS should allocate CPU time fairly among processes to prevent any single process from monopolizing system resources.

Live Lock

- A thread often acts in response to the action of another thread.
- If the other thread's action is also a response to the action of another thread, then *live lock* may result.
- As with deadlock, livelocked threads are unable to make further progress.
- However, the threads are not blocked — they are simply too busy responding to each other to resume work.
- E.g. :- This is comparable to two people attempting to pass each other in a corridor:
  - ✓A moves to his left to let B pass,
  - ✓While B moves to his right to let A pass.
  - ✓Seeing that they are still blocking each other,
  - ✓A moves to his right, while B moves to his left.
  - ✓They're still blocking each other, so..

A Simple Example

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

- A program that will provide a character echo procedure;
  - input is obtained from a keyboard one keystroke at a time.
  - Each input character is stored in variable chin.
  - It is then transferred to variable chout
  - and finally sent to the display.
- Any program can call this procedure repeatedly to accept user input and display it on the user's screen.
- Now consider that we have a single-processor multiprogramming system
  - The user can jump from one application to another, and each application uses the same keyboard for input and the same screen for output.

A Simple Example: On a Multiprocessor

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

The result is that the character input to P1 is lost before being displayed, and the character input to P2 is displayed by both P1 and P2.

ROADMAP - Concurrency

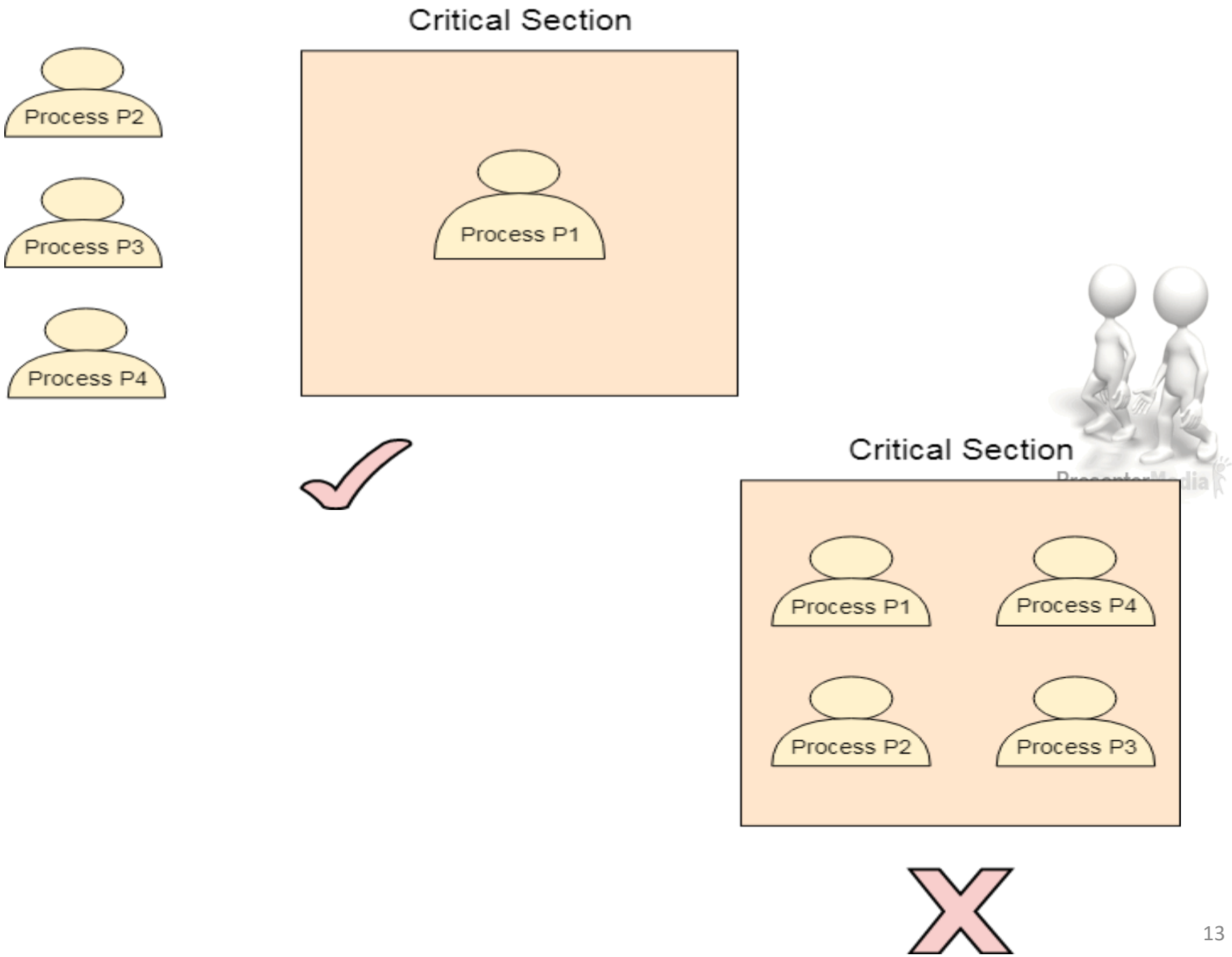
- Key terms
- Principles of concurrency :
- Requirement of mutual exclusion
- **Semaphores**

ROADMAP – Mututal Exclusion

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 non-critical section must do so without interfering with other processes
- No deadlock or starvation

- 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


Critical Section
Process P2
Process P3
Process P4
Process P1
Critical Section
Process P1
Process P4
Process P2
Process P3

SEMAPHORE

What operations can be performed on a semaphore?

- **Semaphore:**
  - •A semaphore is an integer value used for signalling among processes.
  - •Only three operations may be performed on a semaphore, all of which are atomic:

  1. Initialize:-

  A semaphore may be initialized to a nonnegative integer value.

  2.Decrement (semWait):- // To receive a signal
  - •The semWait operation decrements the semaphore value.
  - •If the value becomes negative, then the process executing the semWait is blocked.
  - • Otherwise, the process continues execution.

  3. increment (semSignal):- // To transmit a signal
  - • The semSignal operation increments the semaphore value.
  - •If the resulting value is less than or equal to zero, then a process blocked by a semWait operation, if any, is unblocked.

What are semaphores?

Semaphores are the variables.

Semaphores are used for signaling among processes.

Three kinds of operations are performed on semaphores;

- To initialize the semaphore
- To increment the semaphore value
- To decrement the semaphore value

What are binary semaphores?

Binary semaphores take only the values between 0 to 1.

What are counting semaphores?

Counting semaphores have the non-negative integer value.

How can processes get the critical section?

A critical section is controlled by semaphores by following operations;

•Wait:

Any process can't enter into the critical section.

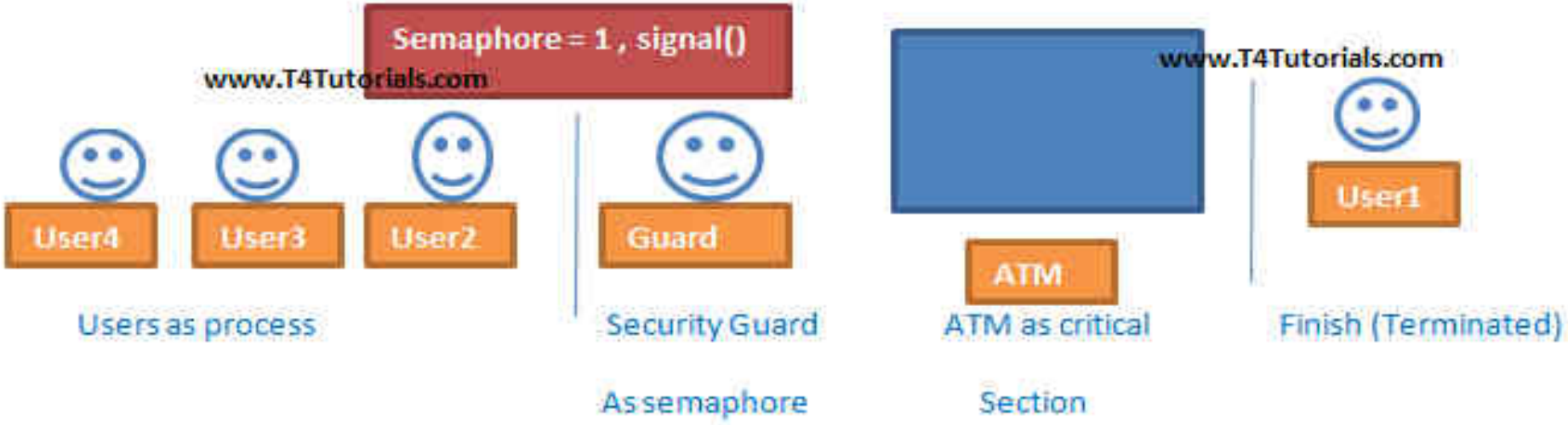
The semaphore value is decremented.

•Signal:

The process can enter into the critical section.

The semaphore value is incremented.

User1(Process1) is using ATM(Critical section), Guard (Semaphore) is performing wait operation and stops the user2, user3 and user4



User1 finish its work and ATM (critical section) is free. Signal is given to all other process.

```
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

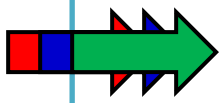
Semaphore Example 1

```
semaphore s = 2;
Pi {
   while(1)  {
        semWait(s);
        /* CS */
        semSignal(s);
        /* remainder */
   }
}
```

- What happens?
- When might this be desirable?

Semaphore Example 1

```
semaphore s = 2;
Pi {
   while(1)  {
      semWait(s);
      /* CS */
      semSignal(s);
      /* remainder */
   }
}
```



s = ~~2~~ ~~1~~ ~~0~~ -1

- What happens?
- When might this be desirable?

Semaphore Example 1

```
semaphore s = 2;
Pi {
   while(1)  {
        semWait(s);
        /* CS */
        semSignal(s);
        /* remainder */
   }
}
```

- What happens?
  - Allows up to 2 processes to enter CS
- When might this be desirable?
  - Need up to 2 processes inside CS
    - e.g., limit number of processes reading a var
  - Be careful not to violate mutual exclusion inside CS!

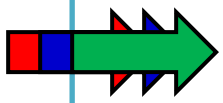
Semaphore Example 2

```
semaphore s = 0;
Pi {
   while(1)  {
        semWait(s);
        /* CS */
        semSignal(s);
        /* remainder */
   }
}
```

- What happens?
- When might this be desirable?

Semaphore Example 2

```
semaphore s = 0;
Pi {
   while(1)  {
        semWait(s);
        /* CS */
        semSignal(s);
        /* remainder */
   }
}
```



s = ~~0~~ ~~-1~~ ~~-2~~ -3

- What happens?
- When might this be desirable?

Semaphore Example 2

```
semaphore s = 0;
Pi {
   while(1)  {
        semWait(s);
        /* CS */
        semSignal(s);
        /* remainder */
   }
}
```

- What happens?
  - No one can enter CS! Ever!
- When might this be desirable?
  - Never!

Binary Semaphore

• A more restrictive semaphore which may only have the value of 0 or 1

1. A binary semaphore may be initialized to 0 or 1.
2. The semWaitB operation checks the semaphore value. If the value is zero, then
   the Process executing the semWaitB is blocked. If the value is one, then the value
   is Changed to zero and the process continues execution.
3. The semSignalB operation checks to see if any processes are blocked on this
   semaphore (semaphore value equals zero).
   If so, then a process blocked by a semWaitB operation is unblocked.
   If no processes are blocked, then the value of the semaphore is set to one.

• To contrast the two types of semaphores, the nonbinary semaphore is often
   referred to as either a **counting semaphore or a general semaphore.**

Binary Semaphore Primitives (locks & mutex)

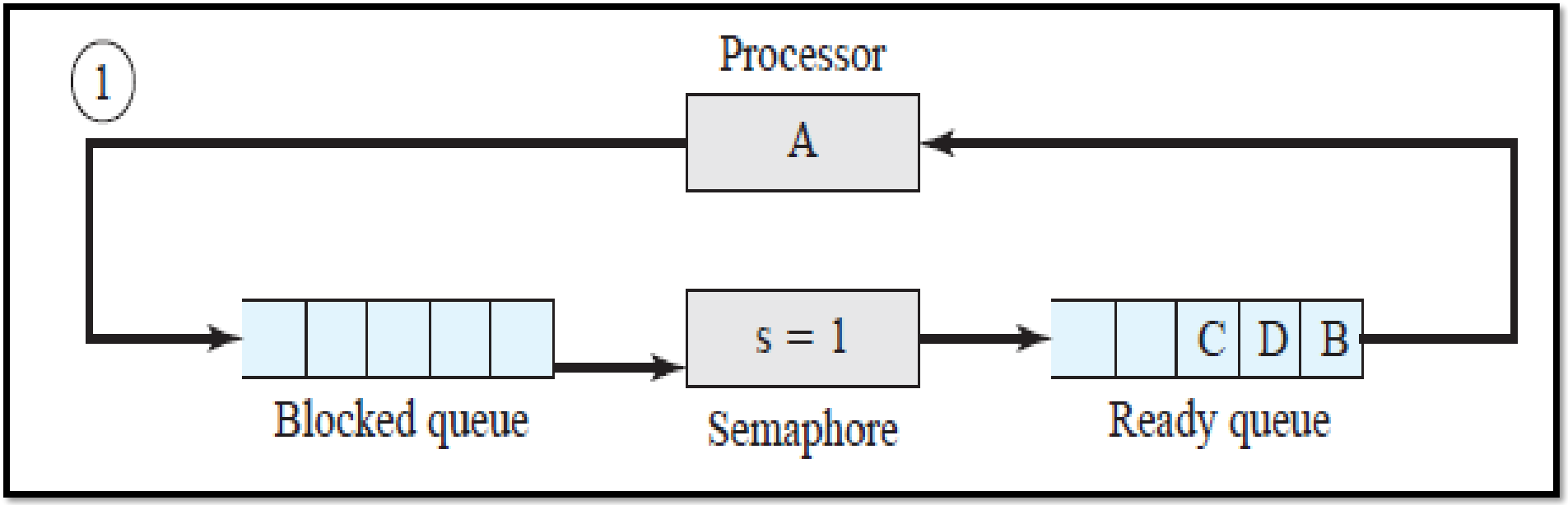
```
struct binary_semaphore {
      enum {zero, one} value;
      queueType queue;
};
void semWaitB(binary_semaphore s)
{
      if (s.value == one)
            s.value = zero;
      else {
               /* place this process in s.queue */;
               /* block this process */;
      }
}
void semSignalB(semaphore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

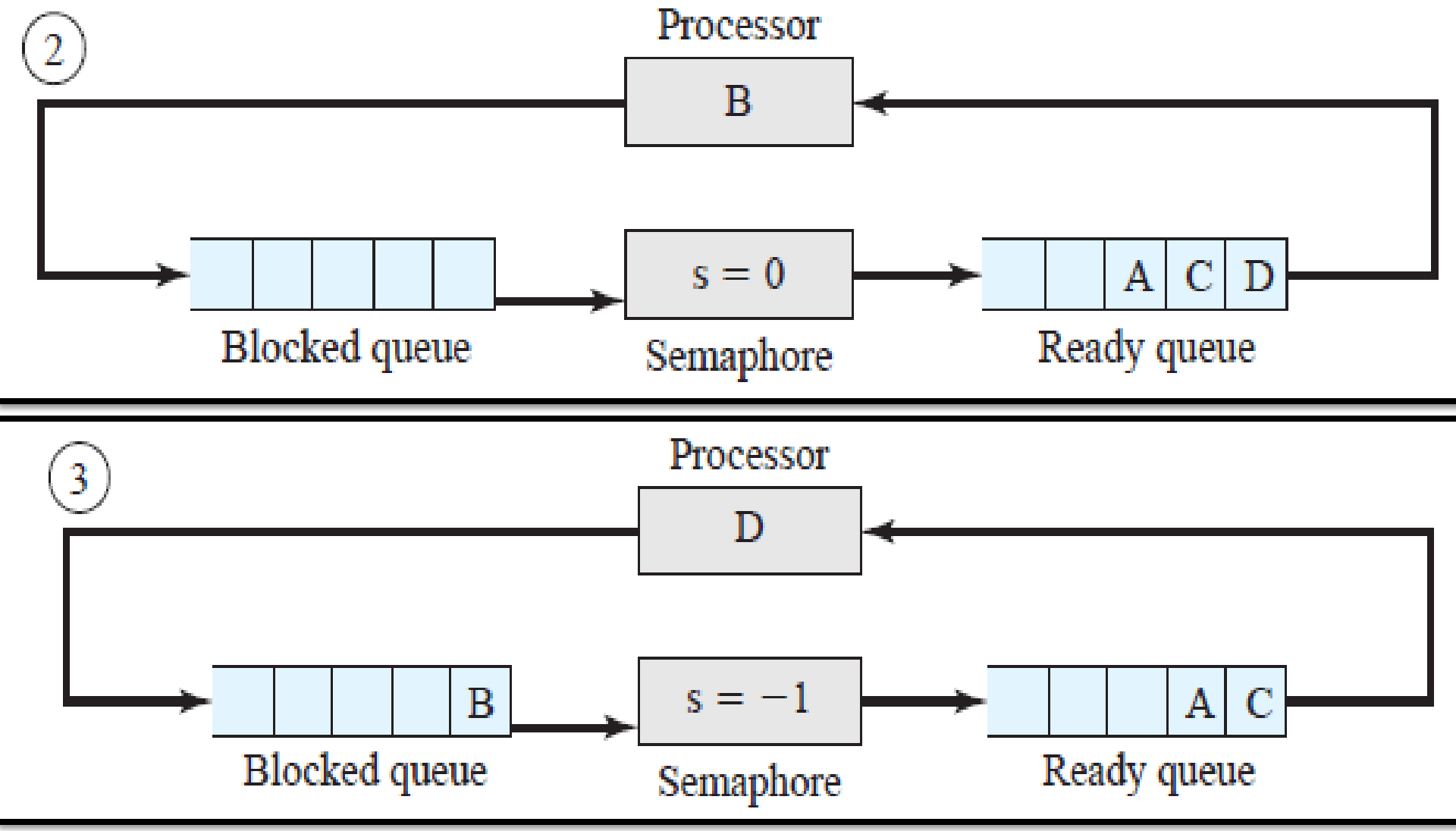
Strong Semaphore Vs. Weak Semaphore

- It is important to decide the order in which processes are removed from a blocked queue.
- The fairest removal policy is first-in-first-out (FIFO):
  - The process that has been blocked the longest is released from the queue first; a semaphore follows this policy is called a **strong semaphore.**
- A semaphore that does not specify the order in which processes are removed from the queue is a **weak semaphore**.
- **Strong semaphores** guarantee freedom from starvation, while
- **Weak semaphores** do not guarantee freedom from starvation.
- **Strong semaphores** are more convenient because this is the form of semaphore typically provided by operating systems.
- **Weak semaphores** are not convenient as strong semaphores.

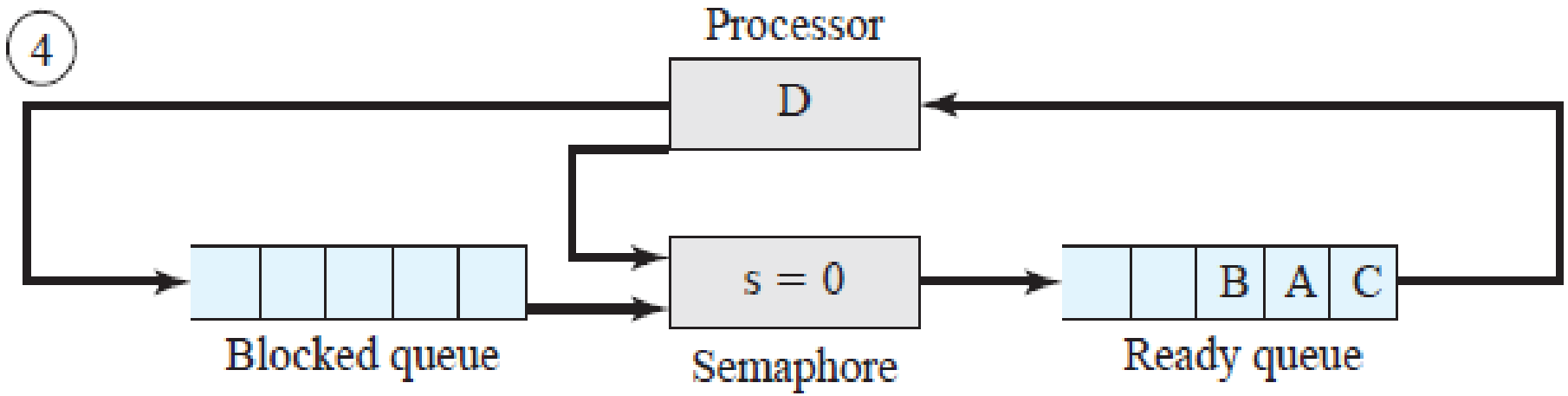
Example of Strong Semaphore Mechanism



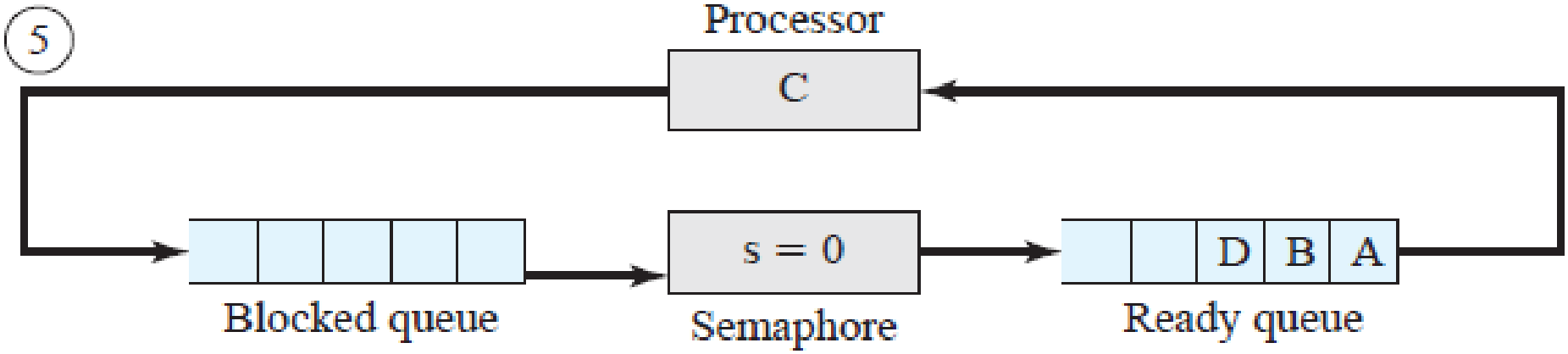
- Initially **(1),** A is running;
- B, C, and D are ready;
- The semaphore count is 1, indicating that one of D's results is available.
- That means up to 1 process can enter critical section.
- When A issues a semWait instruction on semaphore **s**, the semaphore decrements to 0, and A can continue to execute;
- Subsequently it rejoins the ready queue.



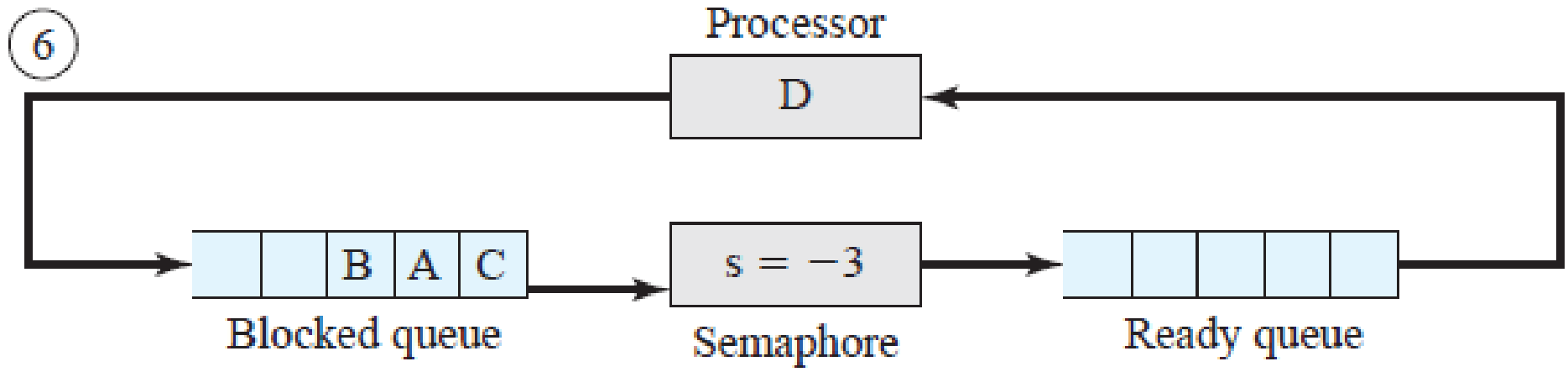
(2) Then B runs, eventually issues a semWait instruction, and is blocked, allowing D to run **(3)**.



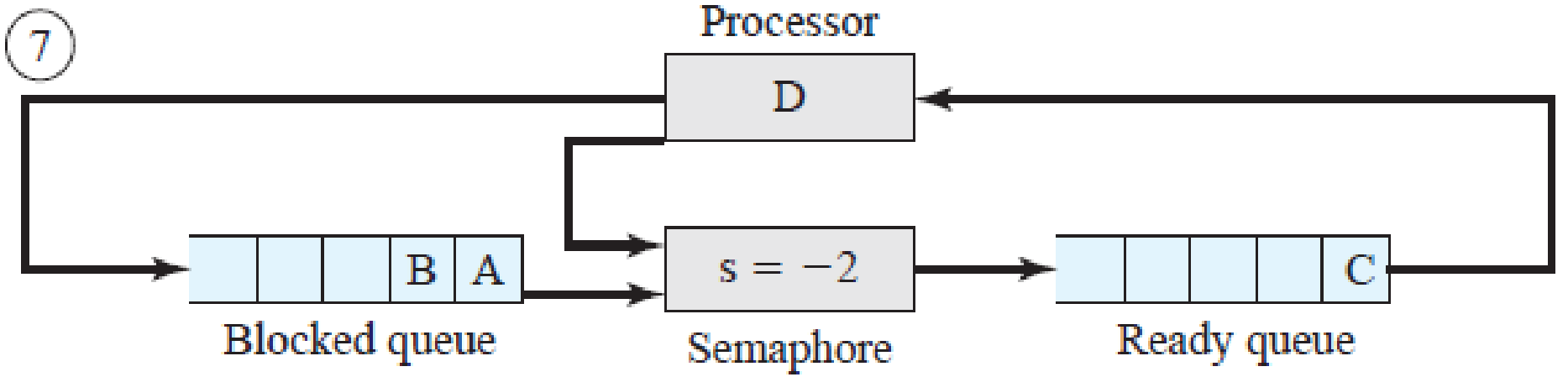
(4) When D completes a new result, it issues a semSignal instruction, which allows B to move to the ready queue.



(5) D rejoins the ready queue and C begins to run but is blocked when it issues a semWait instruction.



Similarly, A and B run and are blocked on the semaphore, allowing D to resume execution **(6)**



When D has a result, it issues a semSignal, which transfers C to the ready queue. Later cycles of D will release A and B from the Blocked state.

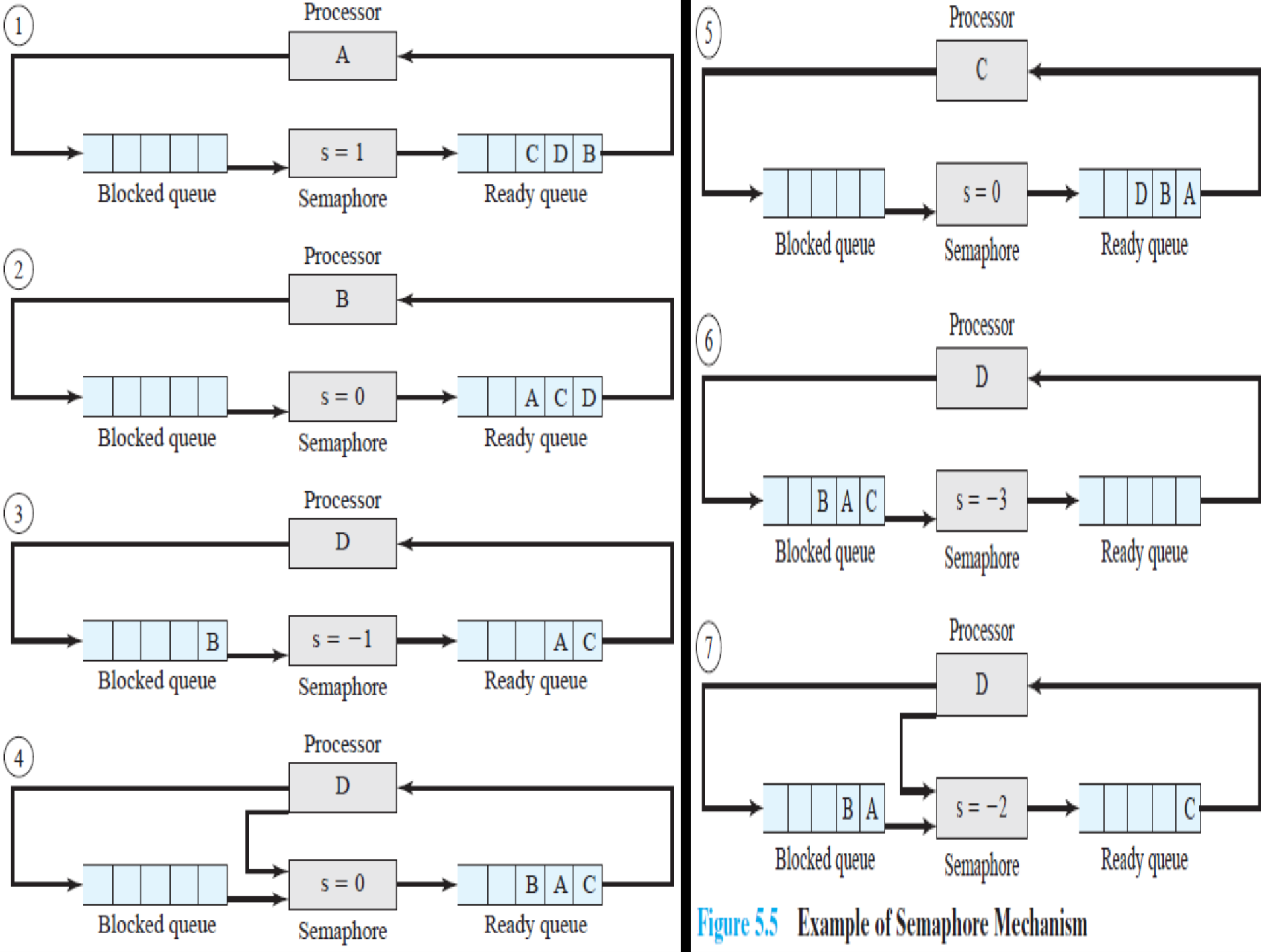


Figure 5.5 Example of Semaphore Mechanism

Mutual Exclusion Using Semaphore

```
/* 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 Accessing Shared Data Protected by a Semaphore

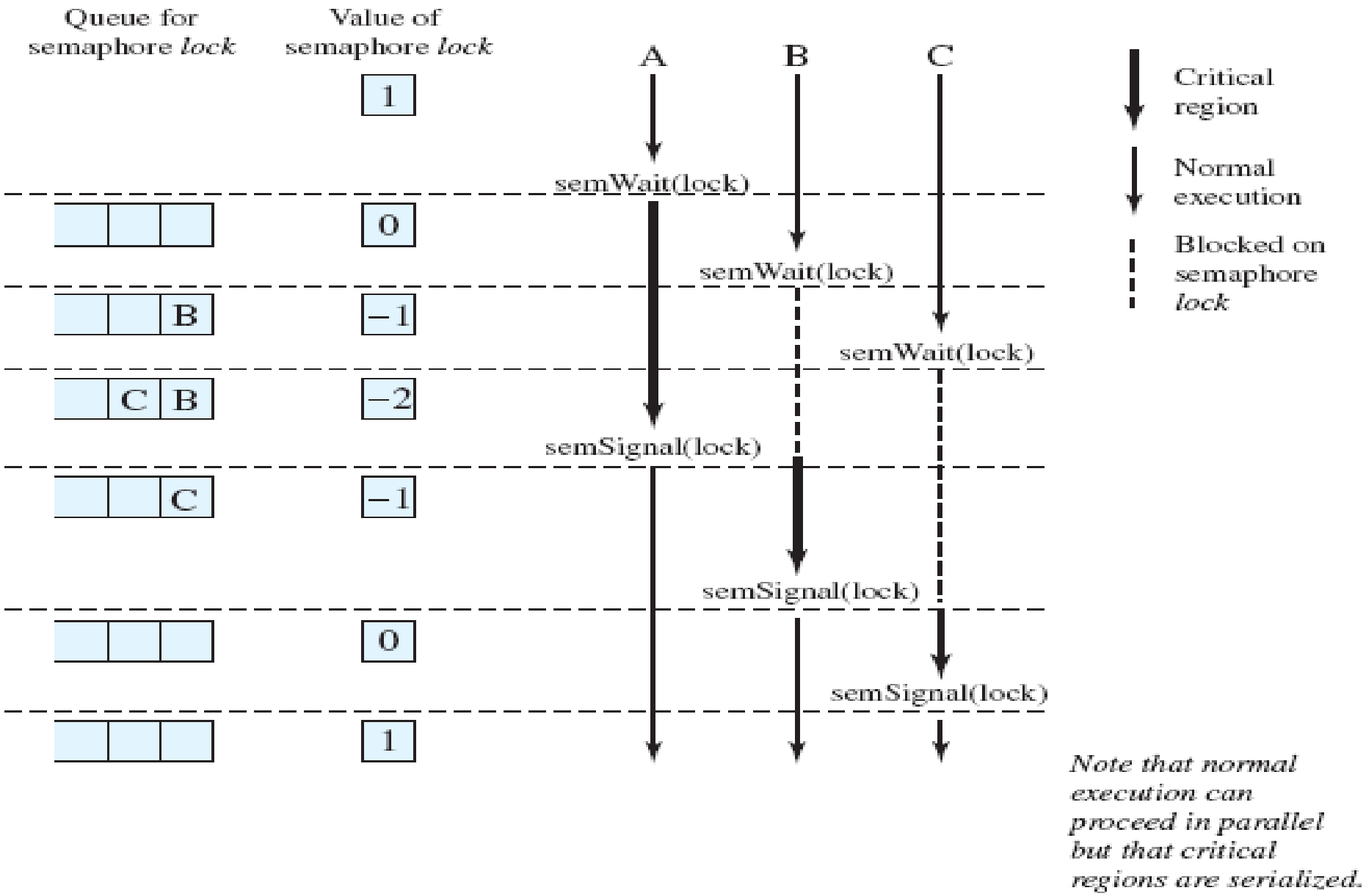


Figure 5.7 Processes Accessing Shared Data Protected by a Semaphore

ROADMAP - Deadlock

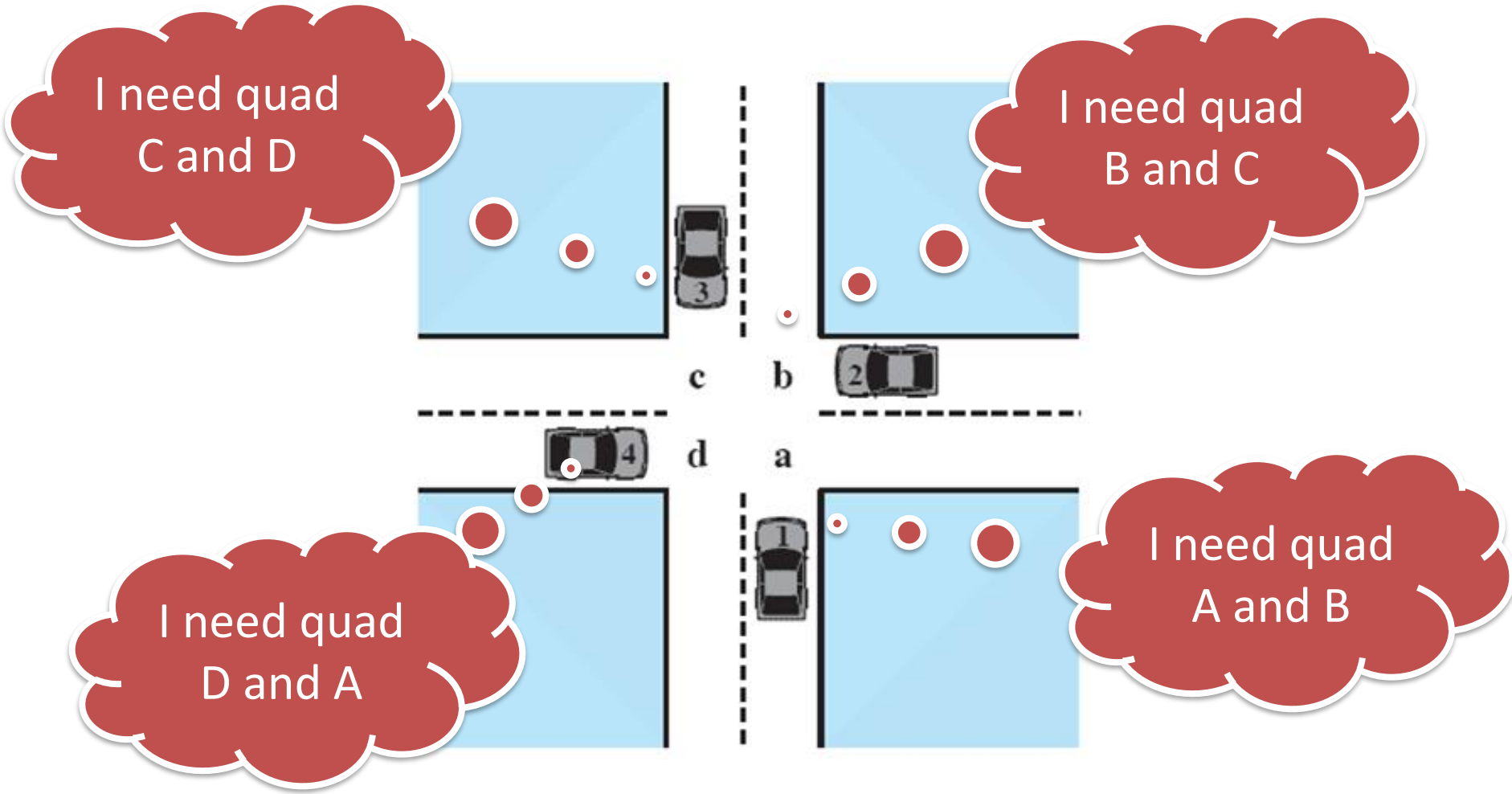
Principles of Deadlock

- Deadlock Prevention
- Deadlock Avoidance
- Deadlock Detection
- Dinning philosophers problem: Solution using semaphores

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 processes are waiting the freeing up of some requested resource.
- Deadlock is permanent because none of the events is ever triggered.
- Unlike other problems in concurrent process management, there is no efficient solution in the general case.

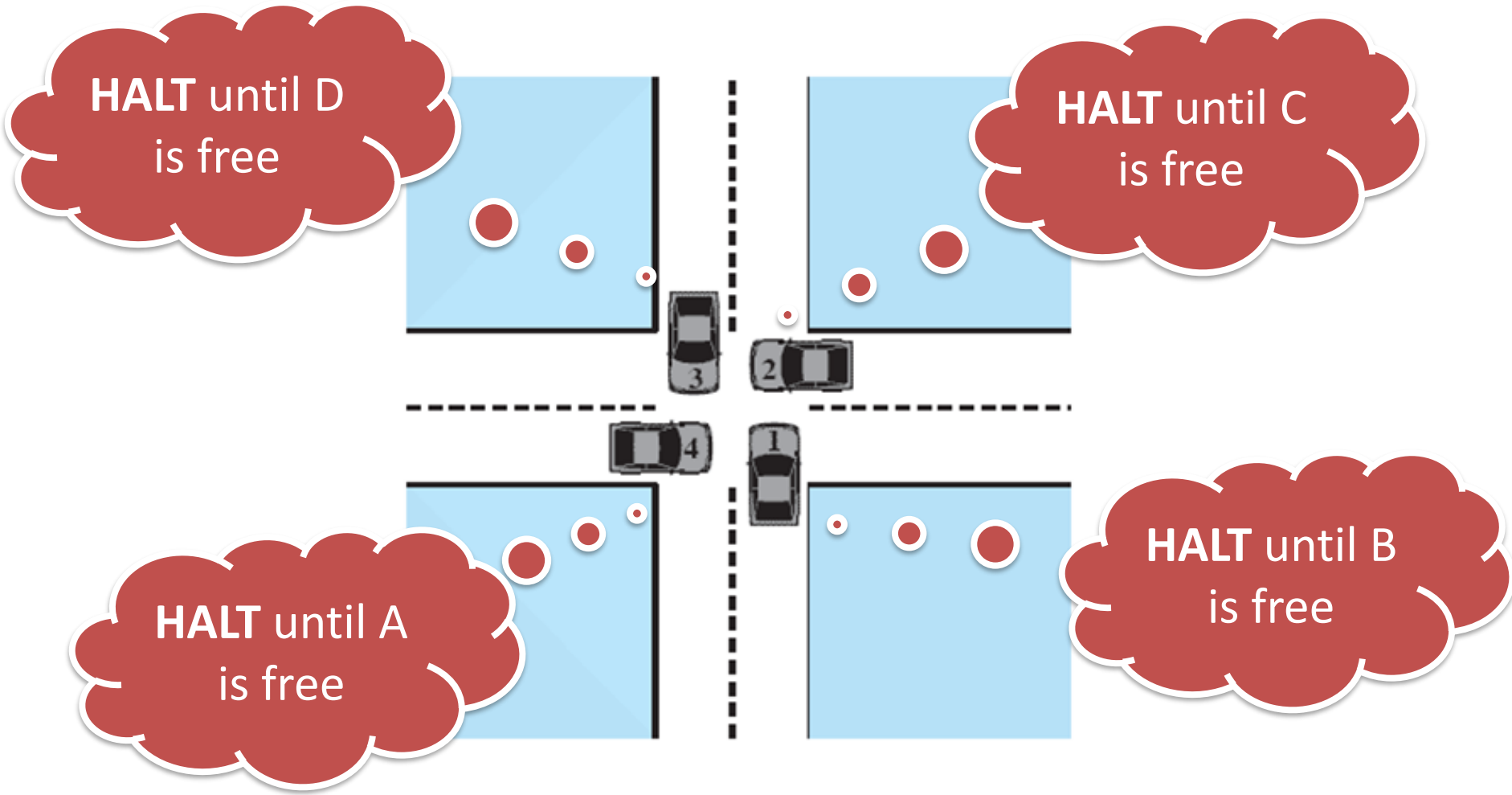
Potential Deadlock



Potential Deadlock

- All deadlocks involve conflicting needs for resources by two or more processes. A common example is the traffic deadlock.
- The typical rule of the road in the United States is that a car at a four-way stop should defer to a car immediately to its right.
- This rule works if there are only two or three cars at the intersection.
- If all four cars arrive at about the same time, each will refrain from entering the intersection, this causes a **potential deadlock.**
  - The deadlock is only potential, not actual, because the necessary resources are available for any of the cars to proceed.
  - If one car eventually does proceed, it can do so.

Actual Deadlock



Actual Deadlock

- ***But*** if all four cars ignore the rules and proceed (cautiously) into the intersection at the same time, then **each car seizes one resource** (one quadrant) but cannot proceed because the required second resource has already been seized by another car.

- This is an actual deadlock.

Two Processes : p & q

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

- 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

- Each process needs exclusive use of both resources for a certain period of time.
- Two processes, P and Q, have the following general form:

Joint Progress Diagram of Deadlock

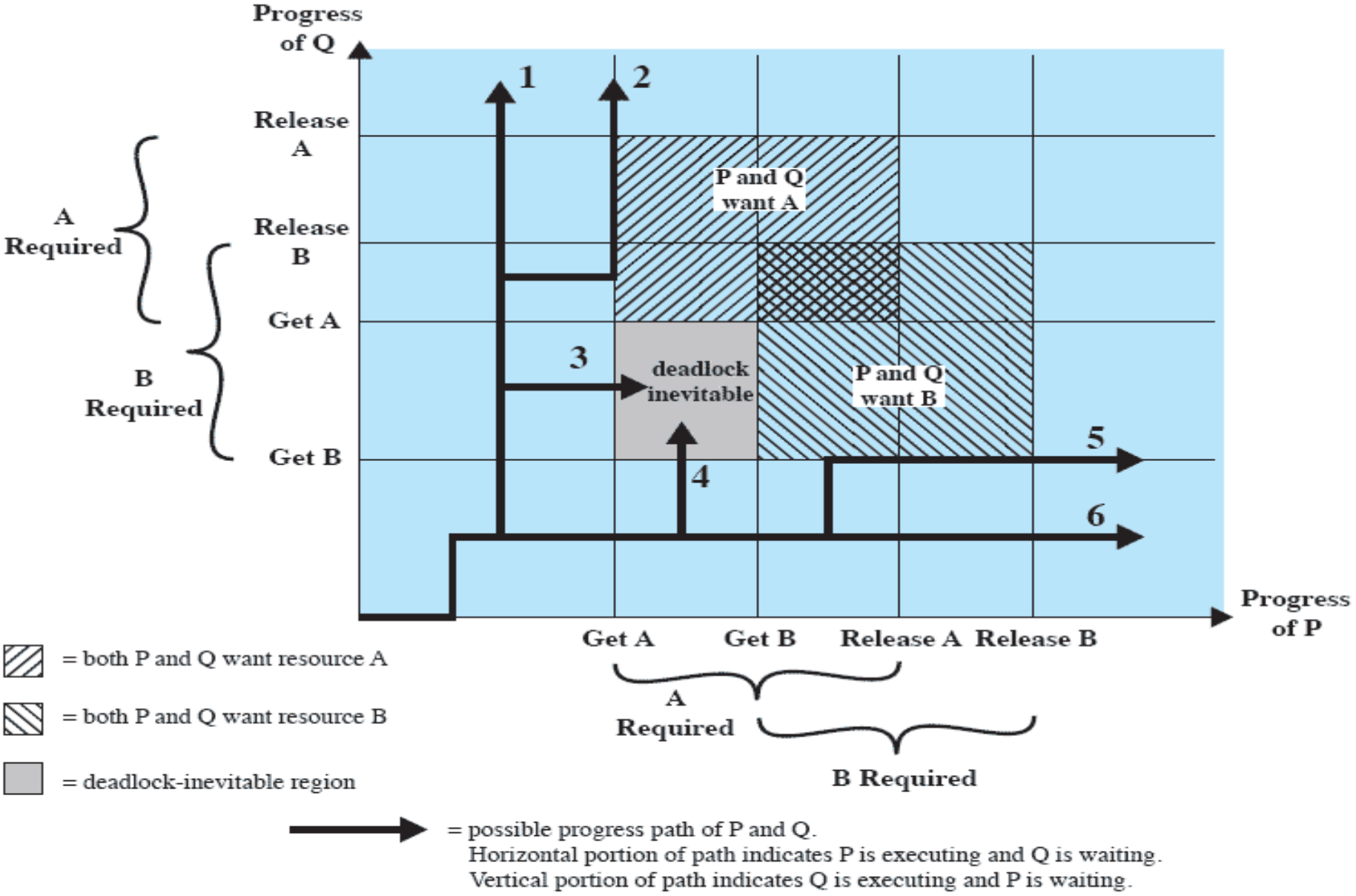


Figure 6.2 Example of Deadlock

Joint Progress Diagram of Deadlock

•This illustrates the progress of two processes competing for two resources.

- The x-axis represents progress in the execution of P
- The y-axis represents progress in the execution of Q.

•Six different paths of execution are shown:-

1. Q acquires B and then A and then releases B and A.
   When P resumes execution, it will be able to acquire both resources.
2. Q acquires B and then A. P executes and blocks on a request for A.
   Q releases B and A.
   When P resumes execution, it will be able to acquire both resources.
3. Q acquires B and then P acquires A.
   Deadlock is inevitable, because as execution proceeds,
   Q will block on A and P will block on B.
4. P acquires A and then Q acquires B.
   Deadlock is inevitable, because as execution proceeds,Q will block on A and
   P will block on B.

Joint Progress Diagram of Deadlock

5. P acquires A and then B.Q executes and blocks on a request for B.

P releases A and B.

When Q resumes execution, it will be able to acquire both resources.

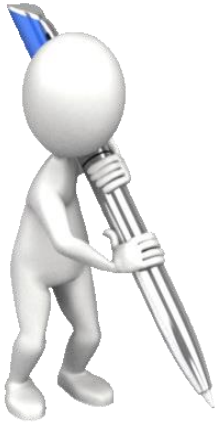
6. P acquires A and then B and then releases A and B.

When Q resumes execution, it will be able to acquire both resources.

- The gray-shaded area can be referred to as a ***fatal region***, applies to paths 3 and 4.
  - If an execution path enters this fatal region, then deadlock is inevitable.

- Note that the existence of a fatal region depends on the logic of the two processes.
  - However, deadlock is only inevitable if the joint progress of the two processes creates a path that enters the fatal region.

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
• • •	• • •

Whether or not deadlock occurs depends on both the dynamics of the execution and on the details of the application.

Diagram of Alternative Logic

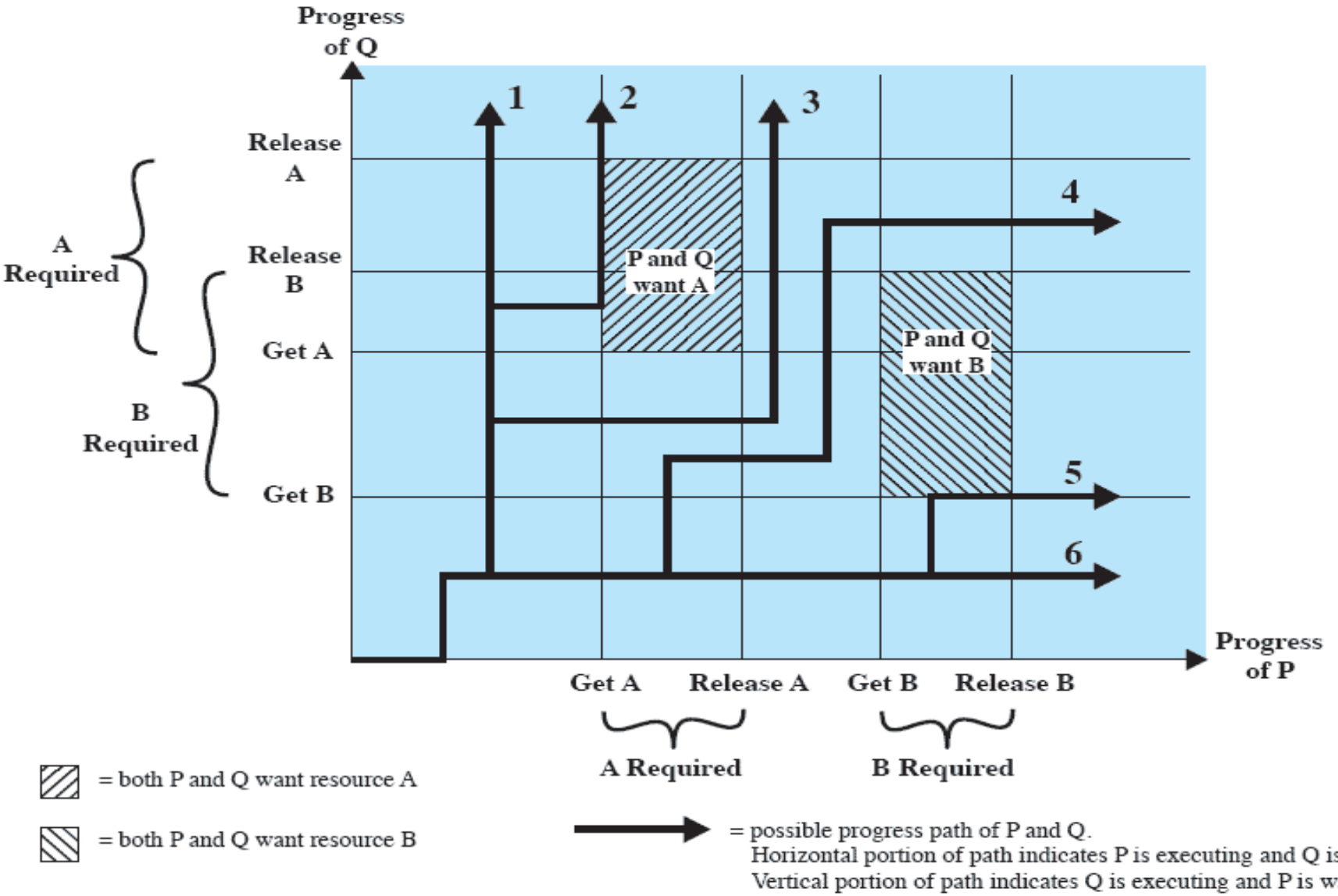


Figure 6.3 Example of No Deadlock [BACO03]

Conditions for possible Deadlock

(1) Mutual exclusion:-

- Only one process may use a resource at a time.
- No process may access a resource unit that has been allocated to another process.

(2)Hold-and-wait:-

- A process may hold allocated resources while awaiting assignment of other resources.

(3)No pre-emption:-

- No resource can be forcibly removed form a process holding it.

- In many ways these conditions are quite desirable.
- For example, mutual exclusion is needed to ensure consistency of results and the integrity of a database.
- Similarly, preemption should not be done arbitrarily.
- For example, when data resources are involved, preemption must be supported by a rollback recovery mechanism.
- All three must be present for deadlock to occur.

Actual Deadlock Requires …

All previous 3 conditions plus:

(4) 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.
- This is actually a potential consequence of the first three.
- Given that the first three conditions exist, a sequence of events may occur that lead to an unresolvable circular wait.
- The unresolvable circular wait is in fact the definition of deadlock.
  - The circular wait listed as condition 4 is unresolvable because the first three conditions hold.
  - Thus, the four conditions, taken together, constitute necessary and sufficient conditions for deadlock.

Possibility of Deadlock	Existence of Deadlock
1. Mutual exclusion 2. No preemption 3. Hold and wait	1. Mutual exclusion 2. No preemption 3. Hold and wait 4. Circular wait

ROADMAP - Deadlock

- Principles of Deadlock
- **Deadlock Prevention**
- Deadlock Avoidance
- Deadlock Detection
- Dinning philosophers problem: Solution using semaphores

Deadlock prevention Staregy

- Deadlock prevention is strategy simply to design a system in such a way that the possibility of deadlock is excluded.
- We can view deadlock prevention methods as falling into two classes.
  - I**ndirect** method of deadlock prevention is to prevent the occurrence of one of the three necessary conditions listed previously (items 1 through 3).
  - **Direct** method of deadlock prevention is to prevent the occurrence of a circular wait (item 4).
- We now examine techniques related to each of the four conditions.

Deadlock prevention Conditions 1 & 2

(1) Mutual Exclusion:-

- The first of the four listed conditions cannot be disallowed (in general).
- If access to a resource requires mutual exclusion, then mutual exclusion must be supported by the OS.
- Some resources, such as files, may allow multiple accesses for reads but only exclusive access for writes.
- Even in this case, deadlock can occur if more than one process requires write permission.

Deadlock prevention Conditions 1 & 2

(2) Hold and Wait:-

- Can be prevented by requiring that a process request all of its required resources at one time and blocking the process until all requests can be granted simultaneously.

This approach is inefficient in two ways.

1) a process may be held up for a long time waiting for all of its resource requests to be filled, when in fact it could have proceeded with only some of the resources.

2) resources allocated to a process may remain unused for a considerable period, during which time they are denied to other processes.

- Another problem is that a process may not know in advance all of the resources that it will require.
- There is also the practical problem created by the use of modular programming or a multithreaded structure for an application.
  - An application would need to be aware of all resources that will be requested at all levels or in all modules to make the simultaneous request.

Deadlock prevention Conditions 3 & 4

(3)No Preemption:-

- can be prevented in several ways.
  1) If a process holding certain resources is denied a further request, that process must release its original resources and, if necessary, request them again together with the additional resource.
  2) If a process requests a resource that is currently held by another process, the OS may preempt the second process and require it to release its resources.

- This latter scheme would prevent deadlock only if no two processes possessed the same priority.
  - This approach is practical only with resources whose state can be easily saved and restored later, as is the case with a processor.

(4)Circular Wait:-

- Can be prevented by defining a linear ordering of resource types.
- As with hold-and-wait prevention, circular-wait prevention may be inefficient, slowing down processes and denying resource access unnecessarily.

ROADMAP - Deadlock

- Principles of Deadlock
- Deadlock Prevention
- **Deadlock Avoidance**
- Deadlock Detection
- Dinning philosophers problem: Solution using semaphores

Deadlock Avoidance

- Deadlock avoidance allows the three necessary conditions
  - but makes judicious choices to assure that the deadlock point is never reached.
- Avoidance allows more concurrency than prevention.
- With deadlock avoidance, a decision is made dynamically whether the current resource allocation request will, if granted, potentially lead to a deadlock.
- Deadlock avoidance requires knowledge of future process resource 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.
- This strategy is hardly optimal, because it assumes the worst:
  - **that all processes will make their maximum claims togather.**

Data Structures:-

Consider a system of n processes and m different types of resources. Let us define the following vectors and matrices:

[Resource = $\mathbf{R} = (R_1, R_2, \ldots, R_m)$	total amount of each resource in the system
Available = $\mathbf{V} = (V_1, V_2, \ldots, V_m)$	total amount of each resource not allocated to any process
Claim = $\mathbf{C} = \begin{bmatrix} C_{11} & C_{12} & \ldots & C_{1m} \\ C_{21} & C_{22} & \ldots & C_{2m} \\ \vdots & \vdots & \vdots & \vdots \\ C_{n1} & C_{n2} & \ldots & C_{nm} \end{bmatrix}$	C_{ij} = requirement of process i for resource j
Allocation = $\mathbf{A} = \begin{bmatrix} A_{11} & A_{12} & \ldots & A_{1m} \\ A_{21} & A_{22} & \ldots & A_{2m} \\ \vdots & \vdots & \vdots & \vdots \\ A_{n1} & A_{n2} & \ldots & A_{nm} \end{bmatrix}$	A_{ij} = current allocation to process i of resource j

Process Initiation Denial

- The matrix Claim gives the maximum requirement of each process for each resource, with one row dedicated to each process.
- This information must be declared in advance by a process for deadlock avoidance to work.
- The matrix Allocation gives the current allocation to each process.
- The following relationships hold:

1. $R_j = V_j + \sum_{i=1}^{n} A_{ij}$, for all j — All resources are either available or allocated.
2. $C_{ij} \leq R_j$, for all i, j — No process can claim more than the total amount of resources in the system.
3. $A_{ij} \leq C_{ij}$, for all i, j — No process is allocated more resources of any type than the process originally claimed to need.

Start a new process P*n+1 only if*

$$R_j \geq C_{(n+1)\,j} + \sum_{i=1}^{n} C_{ij} \quad \text{for all } j$$

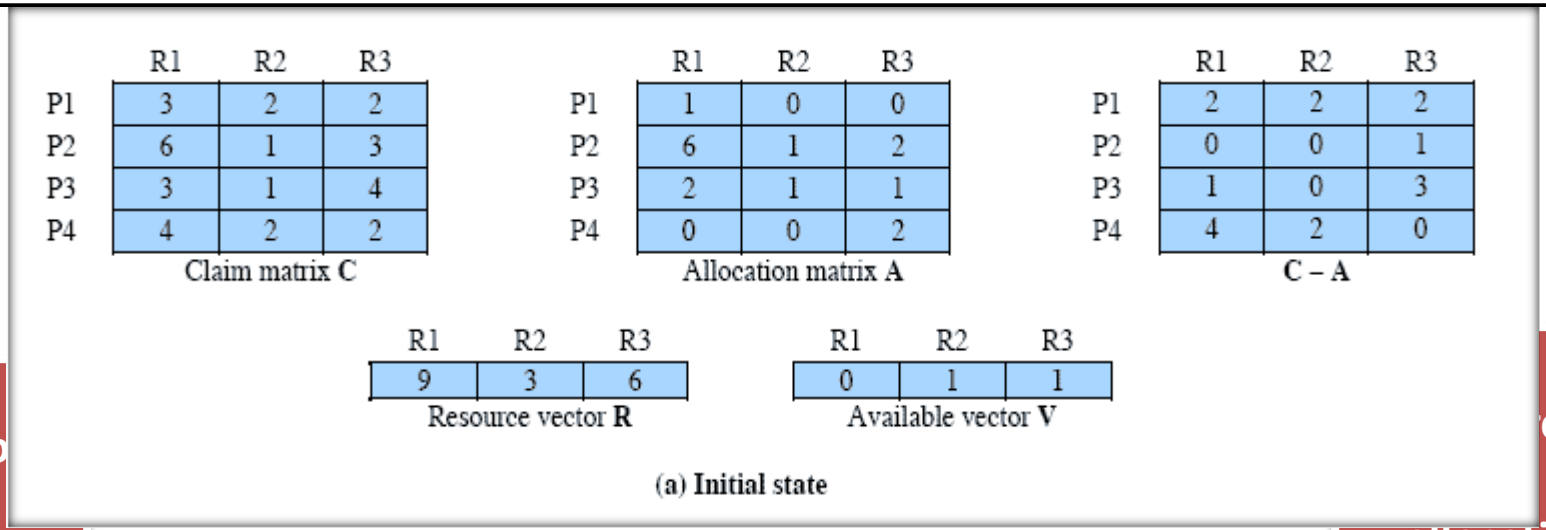
That is, a process is only started if the maximum claim of all current processes plus those of the new process can be met.

Resource Allocation Denial

- A strategy of resource allocation denial is referred to as the **banker's algorithm.**
- Consider a system with fixed number of processes and a fixed number of resources.
- At any time a process may have zero or more resources allocated to it.
  - ***State*** of the system is the current allocation of resources to process
  - ***Safe state*** is where there is at least one sequence of resource allocations to processes that does not result in deadlock.
  - i.e., all of the processes can be run to completion.
  - ***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?***



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

Amount o
Existing
Resources

ces
after
allocation

Figure 6.7 Determination of a Safe State

- This figure shows the state of a system consisting of four processes and three resources.
- Total amount of resources
  - R1 = 9 R2 = 3 R3 = 6

Determination of Safe State

- In the current state allocations have been made to the four processes, leaving available **1 unit of R2** and **1 unit of R3**
- **Is this a safe state?**
- To answer this question, we ask an intermediate question:
  - Can any of the four processes be run to completion with the resources available?
  - That is, can the difference between the maximum requirement and current allocation for any process be met with the available resources?

➢ **$C_{ij} - A_{ij} \leq V_j$, 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.

- The resulting state is shown in Figure 6.7b.

After P2 runs to completion

- Can any of the remaining processes can be completed?
- In this case, each of the remaining processes could be completed as shown on the next slides.

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

Figure 6.7 Determination of a Safe State

After P1 completes

Suppose we choose P1,

- allocate the required resources,
- complete P1,
- and return all of P1's resources to the available pool.

We are left in the state shown in Figure 6.7c on this slide

	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

Figure 6.7 Determination of a Safe State

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

Figure 6.7 Determination of a Safe State

- P3 completes, resulting in the state of Figure 6.7d shown on this slide
- Finally, we can complete P4. At this point, all of the processes have been run to completion.
- Thus, the state defined by Figure 6.7a is a safe state.

Thus, the state defined originally is a safe state.

Determination of an Unsafe State

- This is ***not*** a deadlocked state. It merely has the potential for deadlock.
- It is possible, for example, that if P1 were run from this state it would subsequently release one unit of R1 and one unit of R3 prior to needing these resources again.
- If that happened, the system would return to a safe state.
- Thus, the deadlock avoidance strategy does not predict deadlock with certainty; it merely anticipates the possibility of deadlock and assures that there is never such a possibility.

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

	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

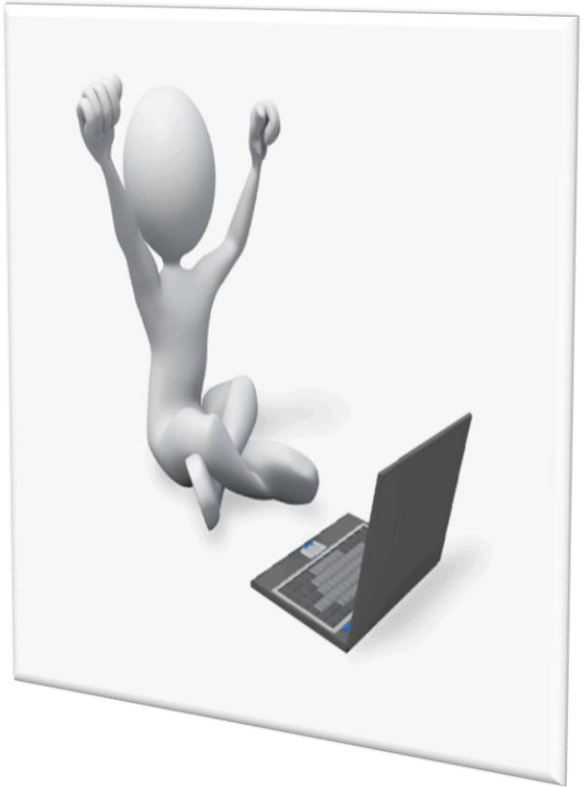
Figure 6.8 **Determination of an Unsafe State**

Deadlock Avoidance

- This suggests the following deadlock avoidance strategy, which ensures that the system of processes and resources is always in a safe state.

- 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

Figure 6.9 Deadlock Avoidance Logic

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 Advantages

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

Deadlock Avoidance Restrictions

However, it does have a number of restrictions on its use

- 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 - Deadlock

- Principles of Deadlock
- Deadlock Prevention
- Deadlock Avoidance
- **Deadlock Detection**
- Dinning philosophers problem: Solution using semaphores

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.
  - Periodically, the OS performs an algorithm that allows it to detect the circular wait condition

Deadlock Detection Algorithm

- A check for deadlock can be made as frequently as each resource request.
- Checking at each resource request has two advantages:
  - It leads to early detection,
  - The algorithm is relatively simple because it is based on incremental changes to the state of the system.

- On the other hand, such frequent checks consume considerable processor time.

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 all processes are unmarked.

Steps for 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 **ith** row of **Q** is less than or equal to **W**. That is, $\mathbf{Q_{ik} <= W_k}$, for **1 <= k<= m.** If no such row is found, terminate the algorithm.

4. If such a row is found, mark process **i** and add the corresponding row of the allocation matrix to **W.** That is, set **Wk = Wk + Aik**, for for **1 <= k<= m.**
Return to step 3.

Deadlock Detection

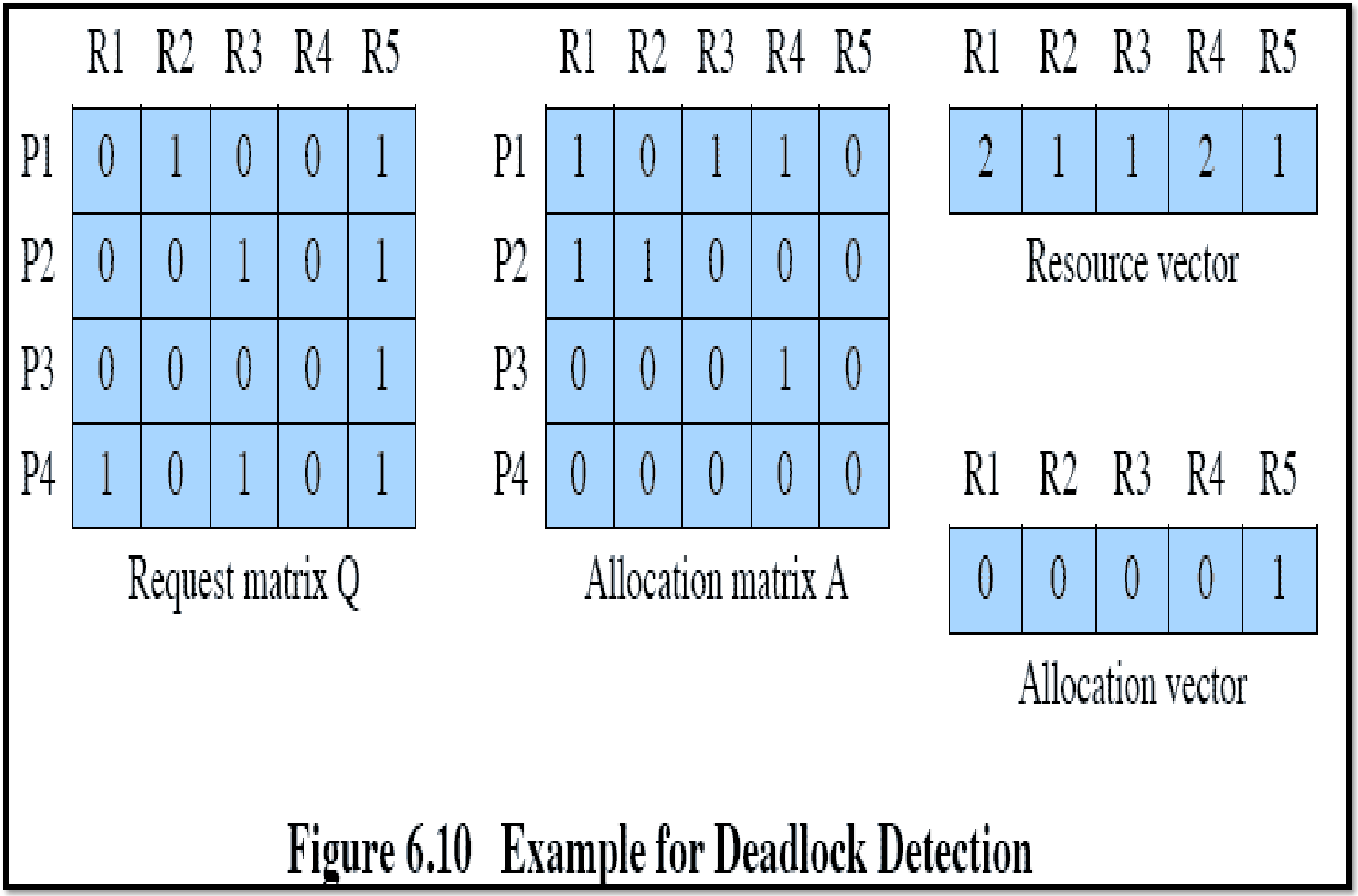


Figure 6.10 Example for Deadlock Detection

Deadlock Detection

1. Mark P4, because P4 has no allocated resources.
2. Set **W** = (0 0 0 0 1).
3. The request of process P3 is less than or equal to **W**, so mark P3 and set W=W + (0 0 0 1 0) = (0 0 0 1 1).
4. No other unmarked process has a row in Q that is less than or equal to W.
   Therefore, terminate the algorithm.

The algorithm concludes with P1 and P2 unmarked, indicating that these processes are deadlocked.

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 - Deadlock

- Principles of Deadlock
- Deadlock Prevention
- Deadlock Avoidance
- Deadlock Detection
- **Dinning philosophers problem: Solution using semaphores**

Dining Philosophers Problem: Scenario

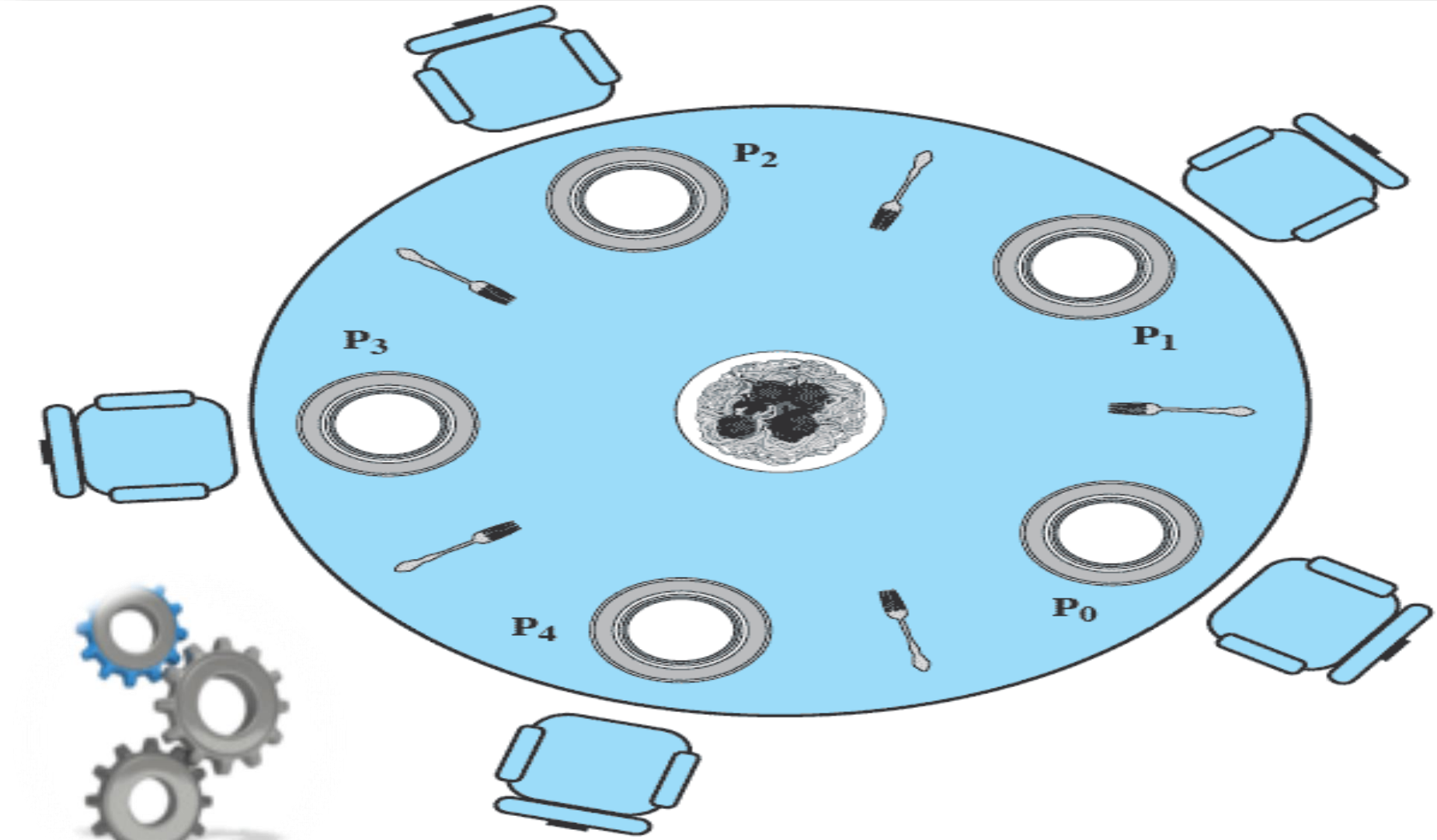


Figure 6.11 Dining Arrangement for Philosophers

Dining Philosophers Problem: Scenario

- Five philosophers live in a house, where a table is laid for them.
- The life of each philosopher consists principally of thinking and eating, and through years of thought, all of the philosophers had agreed that the only food that contributed to their thinking efforts was spaghetti.
- Due to a lack of manual skill, each philosopher requires two forks to eat spaghetti.
- A philosopher wishing to eat goes to his or her assigned place at the table and, using the two forks on either side of the plate, takes and eats some spaghetti.

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

A first solution using semaphores

- Each philosopher picks up first the fork on the left and then the fork on the right.

- After the philosopher is finished eating, the two forks are replaced on the table.

- This solution, leads to deadlock:
  - If all of the philosophers are hungry at the same time, they all sit down, they all pick up the fork on their left, and they all reach out for the other fork, which is not there.

- In this undignified position, all philosophers starve.

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

Avoiding deadlock

- We could consider adding an attendant who only allows four philosophers at a time into the dining room.

- With at most four seated philosophers, at least one philosopher will have access to two forks.

- This slide shows such a solution, again using semaphores. This solution is free of deadlock and starvation.

UNIT 3 COMPLETED