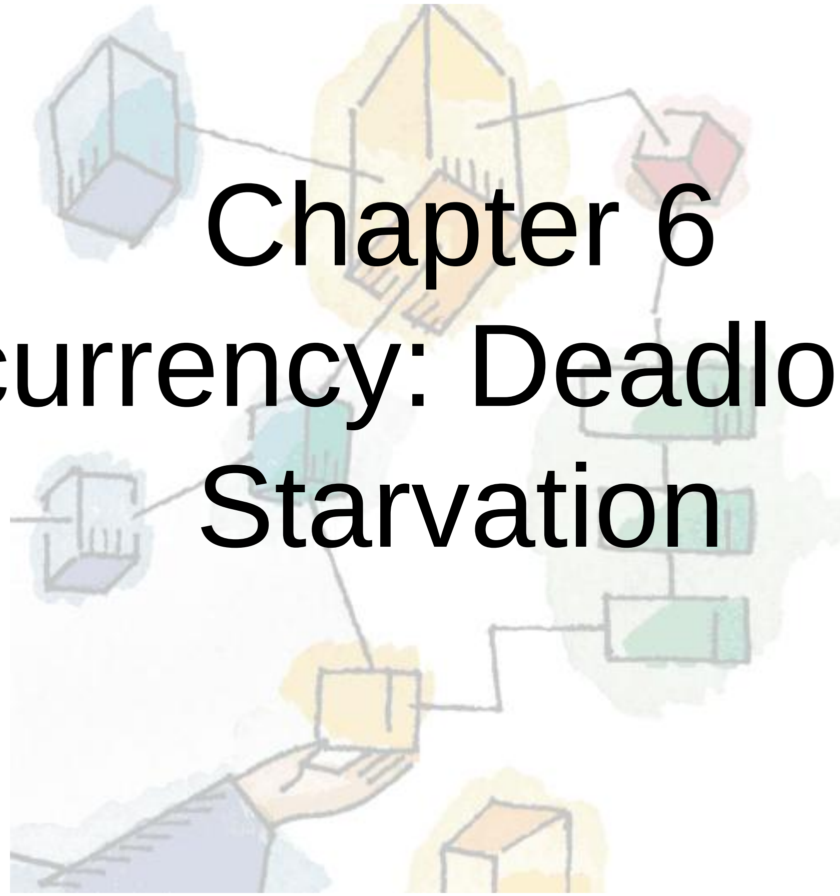
Operating Systems:
Internals and Design Principles, 6/E
William Stallings



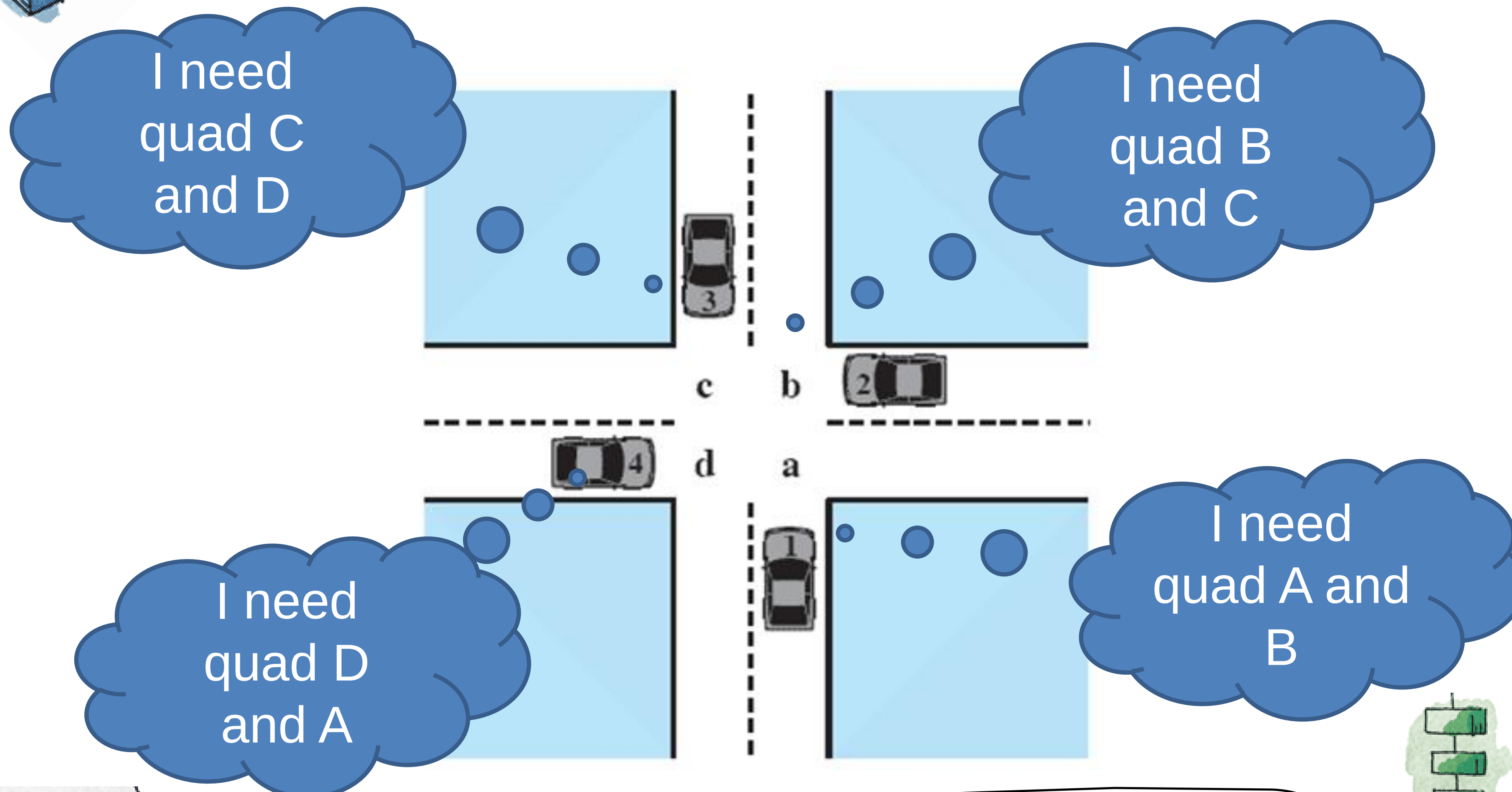
Chapter 6
Concurrency: Deadlock and Starvation

Deadlock

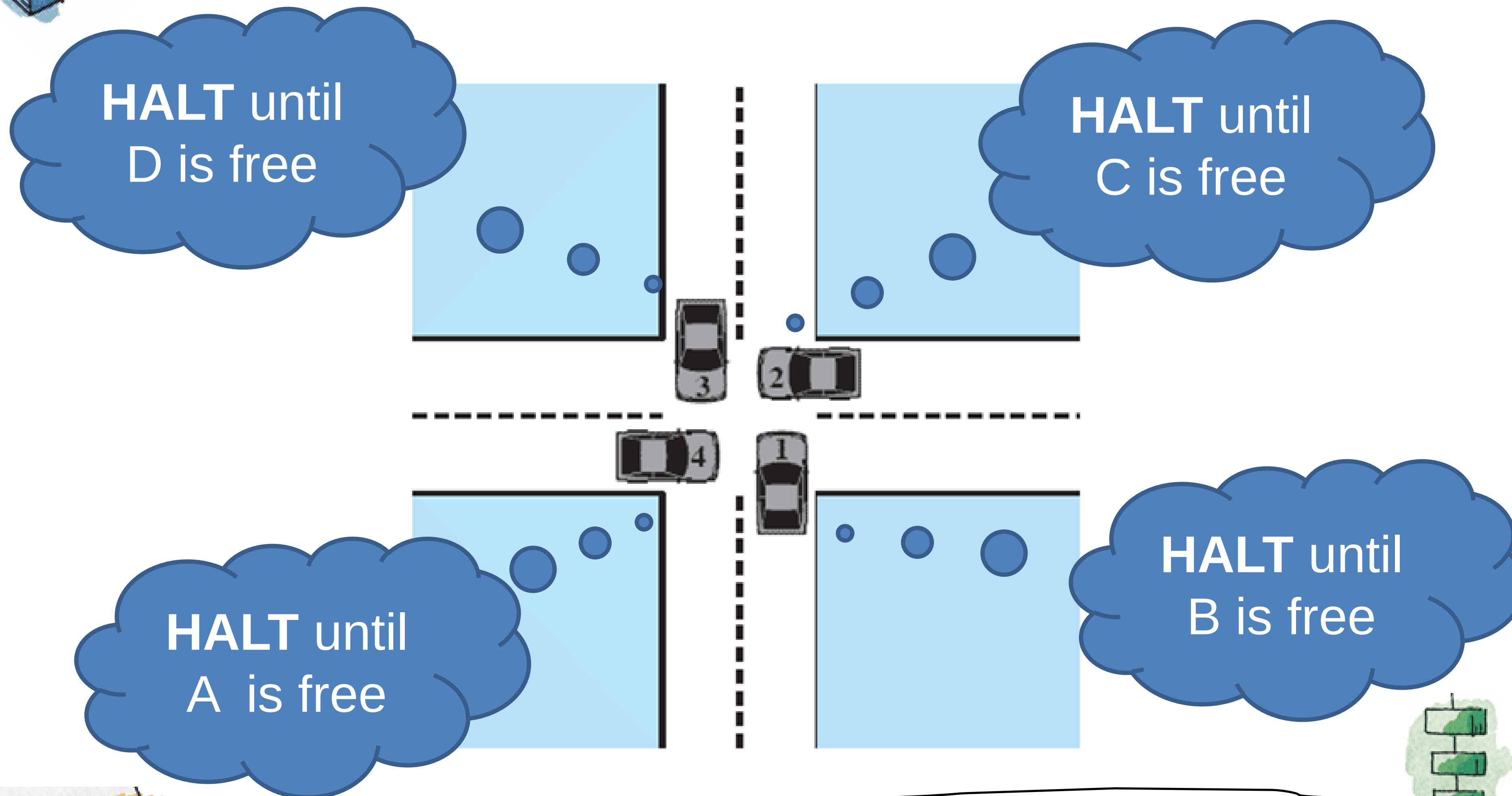
- A set of processes is deadlocked when each process in the set is blocked awaiting an event that can only be triggered by another blocked process in the set
  - Typically involves processes competing for the same set of resources

- Why deadlock is permanent?
  - Because none of the event is ever triggered

Potential Deadlock



Actual Deadlock



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
- For uniprocessor system, only one process can execute at a time
- Hence, six different execution paths are possible

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

Two Processes P and Q

1. Q gets B and then A, Releases B and A, P runs and acquires both A & B
2. Q gets B and then A, P executes and blocks on request for A, Q releases B and A, P resumes and gets both A & B
3. Q gets B and P gets A
4. P gets A and Q gets B
5. P gets A and B, Q runs and block on request of B, P releases A and B, Q resumes and gets A & B both
6. P gets A and B, P releases A and B, Q runs and acquires both A and B

Two Processes P and Q

- 3 and 4 define "Fatal region" in the joint progress diagram
- If the execution path enters fatal region, deadlock is inevitable
- If the joint progress does not enter fatal region, deadlock will not occur

- Hence, occurrence of deadlock depends on
  - Execution (Path of execution)
  - Application Logic (Programming)

Joint Progress Diagram of Deadlock

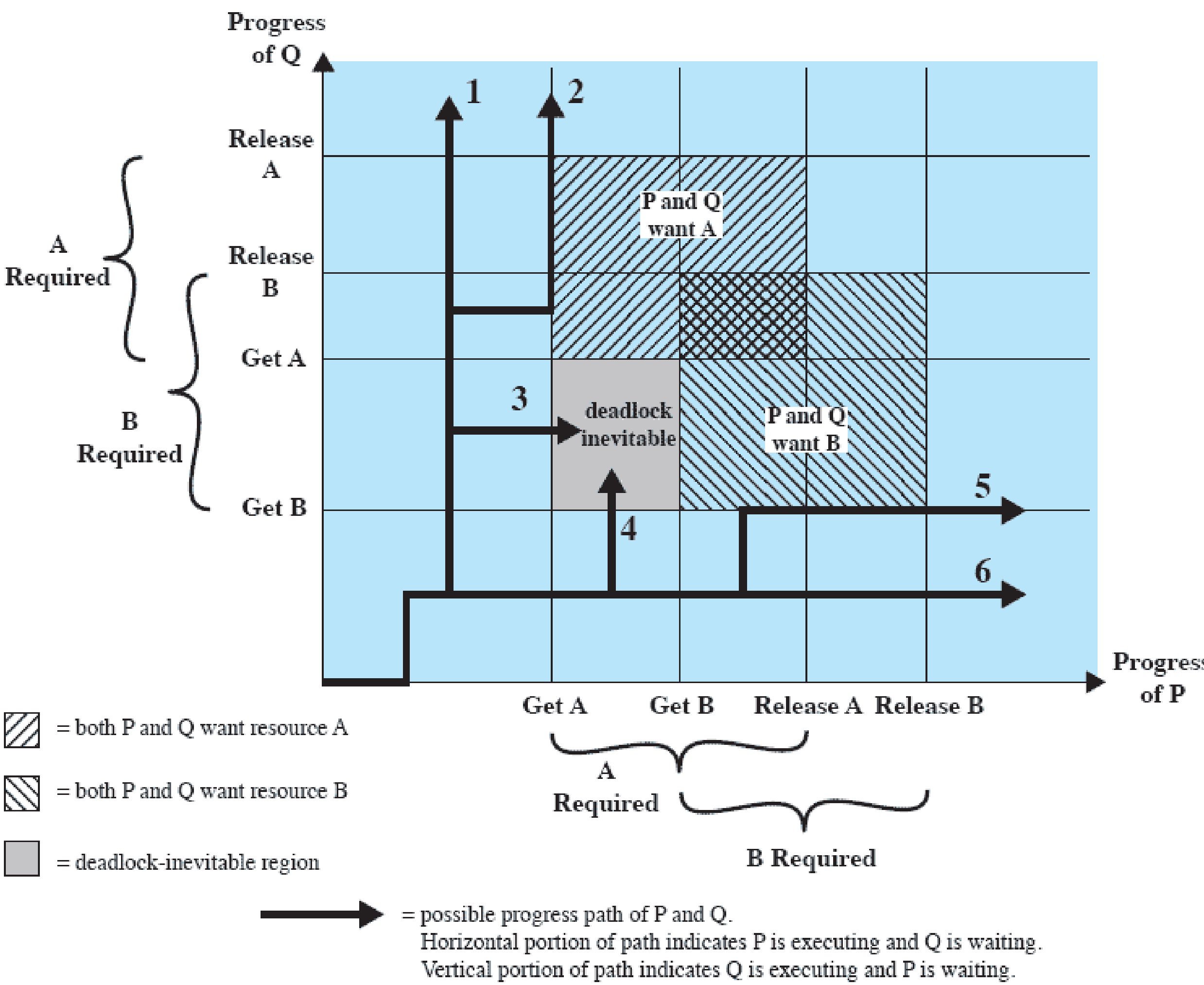


Figure 6.2 Example of Deadlock

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
• • •	• • •

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.
  - E.g. I/O channels, Memory
- Consumable
  - One that can be created (***produced***) and destroyed (***consumed***).
  - No limit on the number of resources of any type

Reusable Resources

- Such as:
  - Processors, I/O channels, main and secondary memory, devices, and data structures such as files, databases, and semaphores

- Consider two processes that compete for exclusive access to a disk D and a tape drive T.
- Deadlock occurs if each process holds one resource and requests the other.

Reusable Resources Example-1

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

Reusable Resources Example-1

- Execution sequence leading to deadlock can be p0,p1,q0,q1,p2,q2
- The cause can be embedded in complex program logic, so detection can be difficult
- Solution:
  - Set constraints on the order of resource requests

Example 2: Memory Request

- Space is available for allocation of 200Kbytes, and the following sequence of events occur

P1	P2
. . .	. . .
Request 80 Kbytes;	Request 70 Kbytes;
. . .	. . .
Request 60 Kbytes;	Request 80 Kbytes;

- Deadlock occurs if both processes progress to their second request
- Solution:
  - Use of Virtual Memory

Consumable Resources

- Such as Interrupts, signals, messages, and information in I/O buffers
- Deadlock may occur if a Receive message is blocking
- Consider a pair of processes, in which each process attempts to receive a message from the other process and then send a message to the other process

P1	**P2**
...	...
Receive (P2);	Receive (P1);
...	...
Send (P2, M1);	Send (P1, M2);

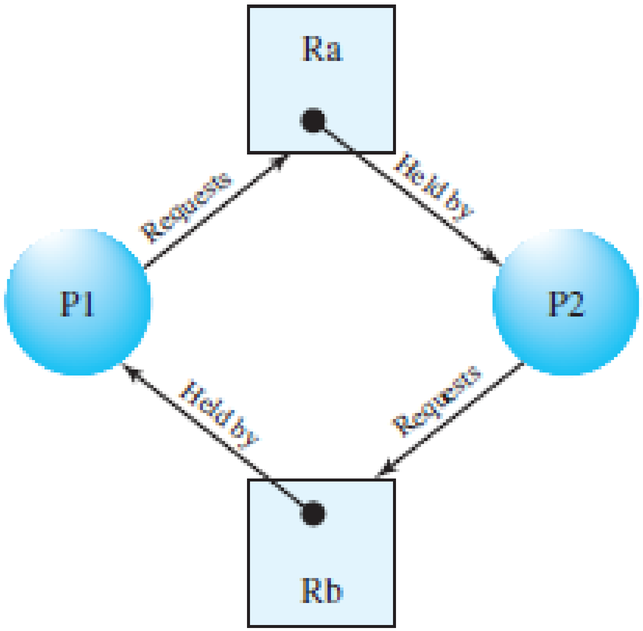
Resource Allocation Graphs

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

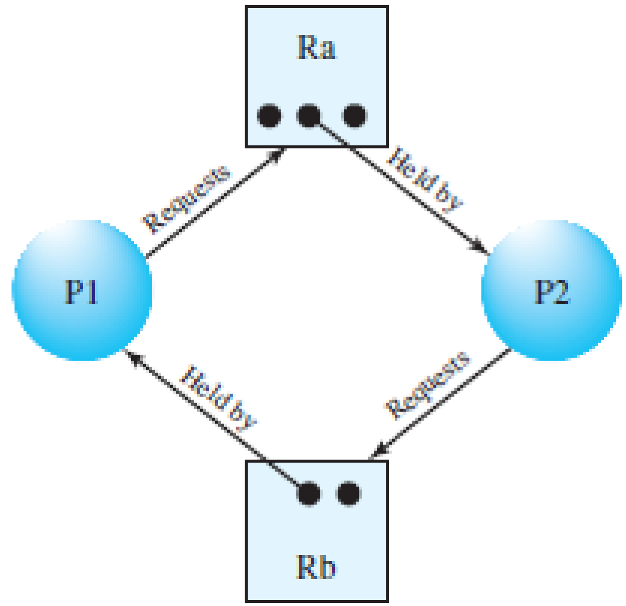
(a) Resouce is requested

(b) Resource is held

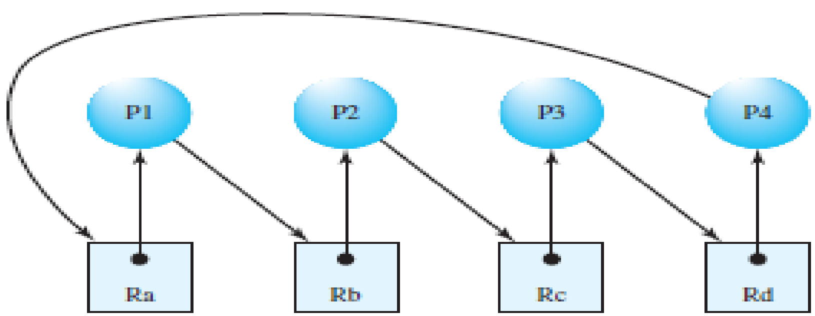
Resource Allocation Graphs



(c) Circular wait



(d) No deadlock



Conditions for ***possible*** Deadlock

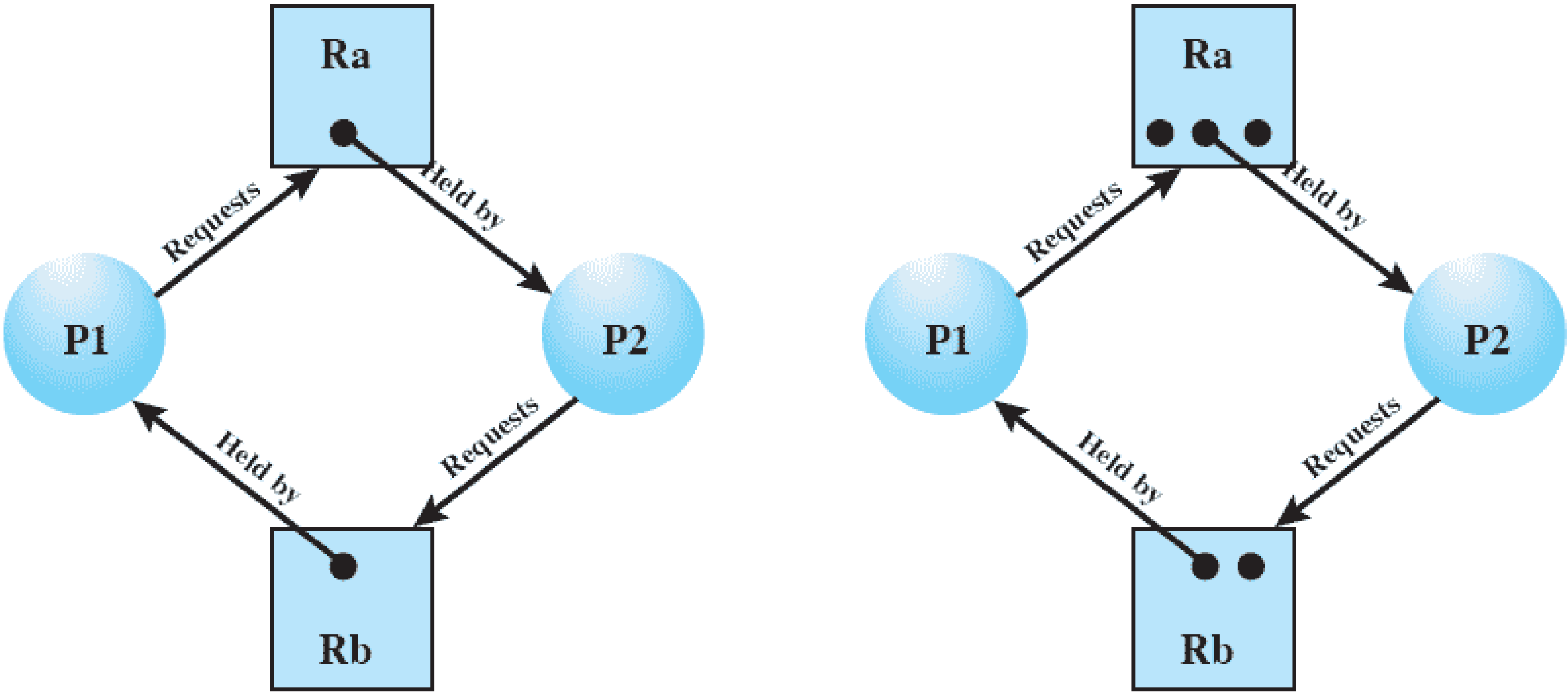
- Mutual exclusion
  - Only one process may use a resource at a time
- Hold-and-wait
  - A process may hold allocated resources while awaiting assignment of others
- No pre-emption
  - No resource can be forcibly removed form a process holding it

- These three conditions called necessary conditions for potential deadlock

Actual Deadlock Requires …

- All previous 3 conditions plus one more is needed for actual deadlock
- 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
  - Circular wait can’t be resolved because first three conditions hold
  - “Fatal Region” exits if first three conditions are met
    - If any one of the condition is not met, deadlock won’t occur

Resource Allocation Graphs of deadlock



(c) Circular wait

(d) No deadlock

Resource Allocation Graphs

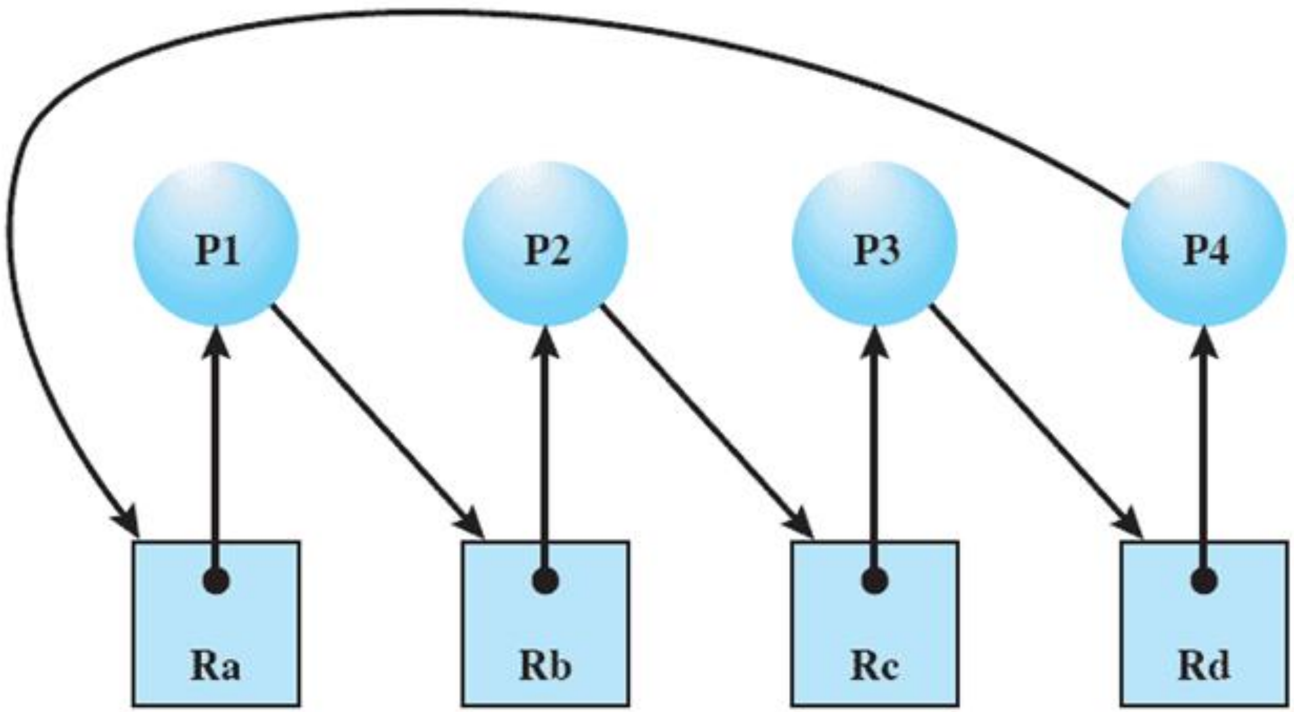


Figure 6.6 Resource Allocation Graph for Figure 6.1b

Dealing with Deadlock

- Three general approaches exist for dealing with deadlock.
  - Prevent deadlock
    - Eliminate one of the four conditions
  - Avoid deadlock
    - Make dynamic choices based on current allocation state
  - Detect Deadlock
    - Detect conditions 1-4 and recover

Deadlock Prevention Strategy

- Design a system in such a way that the possibility of deadlock is excluded.

- Two main methods
  - Indirect – prevent occurrence of one of the three necessary conditions
  - Direct – prevent circular wait

Deadlock Prevention Conditions 1 & 2

- Mutual Exclusion
  - If access to resource needs mutual exclusion, it has to be enforced

- Hold and Wait
  - Can be prevented, but has issues
  - Require a process request all of its required resources at one time and block the process until all requests can be granted simultaneously

Deadlock Prevention Condition 3

- No Preemption
  - Can be avoided in two ways
  1. If the process holding a resource is denied further requests, then it must release original resources
  2. If a process requests a resource held by another process, OS may preempt the second process and get the required resource
  - Resource details have to be maintained

Deadlock Prevention Condition 4

- Circular Wait
  - Define a linear ordering of resource types (Ri, Rj …)
  - If a process has been allocated resource of type R, then it may request only the resources of types following R in the ordering
  - E.g.
    - Pa acquires Ri, can request Rj
    - Pb acquires Rj, can't request Ri, can request Rk

Deadlock Avoidance

- In case of deadlock prevention, we have inefficient use of resources as well as processes execute in inefficient manner
- For 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

Data Structures Used for n processes and m types of resources

[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

Relationships

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.

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.

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

- 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 of allocation to processes that does not result in deadlock (All processes can complete)
  - ***Unsafe state*** is a state that is not safe

Resource Allocation Denial

- When a new process enters the system, it must declare the number of instances of each resource type it may need
  - This number should not exceed the total resources in the system
- When a process requests a set of resources, the system must determine whether the allocation will keep the system in safe state or not
  - If yes, then allocation is done otherwise process has to wait

Resource Allocation Denial :Data Structures

- **Available:**
  - Vector of length m
  - Number of resources available for each type
- **Max:**
  - Matrix (n*m), that defines maximum demand of each process
- **Allocation:**
  - Matrix (n*m), that defines number of each type of resources allocated
- **Need:**
  - Matrix (n*m), that indicates the remaining resource need of each process
  - Calculated as Max - Allocation

Safety Algorithm

1. Let ***Work*** and ***Finish*** be vectors of length *m* and *n*, respectively. Initialize:

   Work = Available

   Finish [i] = false **for *i* = 0, 1, …, *n*- 1**

2. Find an ***i*** such that both:

   (a) ***Finish [i] = false***

   (b) ***Need**$_i$* $\leq$ ***Work***

   If no such ***i*** exists, go to step 4

3. ***Work = Work + Allocation**$_i$*
   Finish[i] = true
   go to step 2

4. If ***Finish [i] == true*** for all ***i***, then the system is in safe state

Resource-Request Algorithm for Process P_i

- ***Request$_i$*** = request vector for process ***P$_i$***

1. If ***Request$_i$*** $\leq$ ***Need$_i$*** go to step 2. Otherwise, raise error condition, since process has exceeded its maximum claim
2. If ***Request$_i$*** $\leq$ ***Available***, go to step 3. Otherwise ***P$_i$*** must wait, since resources are not available
3. Pretend to allocate requested resources to ***P$_i$*** by modifying the state as follows:

Available = Available – Request$_i$;

Allocation$_i$ = Allocation$_i$ + Request$_i$;

Need$_i$ = Need$_i$ – Request$_i$;

- If safe $\Rightarrow$ the resources are allocated to ***P$_i$***
- If unsafe $\Rightarrow$ ***P$_i$*** must wait, and the old resource-allocation state is restored

Example: Safety Algorithm

- 5 processes P_0 through P_4;
- 3 resource types:

  A (10 instances), *B* (5 instances), and *C* (7 instances)

- Snapshot at time T_0:

	Allocation	Max	Available
	A B C	*A B C*	*A B C*
P_0	0 1 0	7 5 3	3 3 2
P_1	2 0 0	3 2 2	
P_2	3 0 2	9 0 2	
P_3	2 1 1	2 2 2	
P_4	0 0 2	4 3 3	

Example: Safety Algorithm

- The content of the matrix ***Need*** is defined to be ***Max – Allocation***

	Need
	A B C
P_0	7 4 3
P_1	1 2 2
P_2	6 0 0
P_3	0 1 1
P_4	4 3 1

- The system is in a safe state since the sequence < P_1, P_3, P_4, P_2, P_0> satisfies safety criteria

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
- There must be a fixed number of resources to allocate

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

- Uses Allocation matrix and Available vector as previous

- Also uses 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

Detection Algorithm

- **When should we check for the deadlock?**
  - Very Frequently
    - With each resource request
    - Gives early detection advantage
    - But can consume a lot of CPU time
  - Less Frequently
    - Depending upon how likely deadlock occurs

Recovery Strategies Once Deadlock Detected

1. Abort all deadlocked processes
2. Back up each deadlocked process to some previously defined checkpoint, and restart all process
   – Risk or deadlock recurring
3. Successively abort deadlocked processes until deadlock no longer exists
   – Select one process at a time and check for deadlock once aborted
4. Successively preempt resources until deadlock no longer exists

Dining Philosophers Problem: Scenario

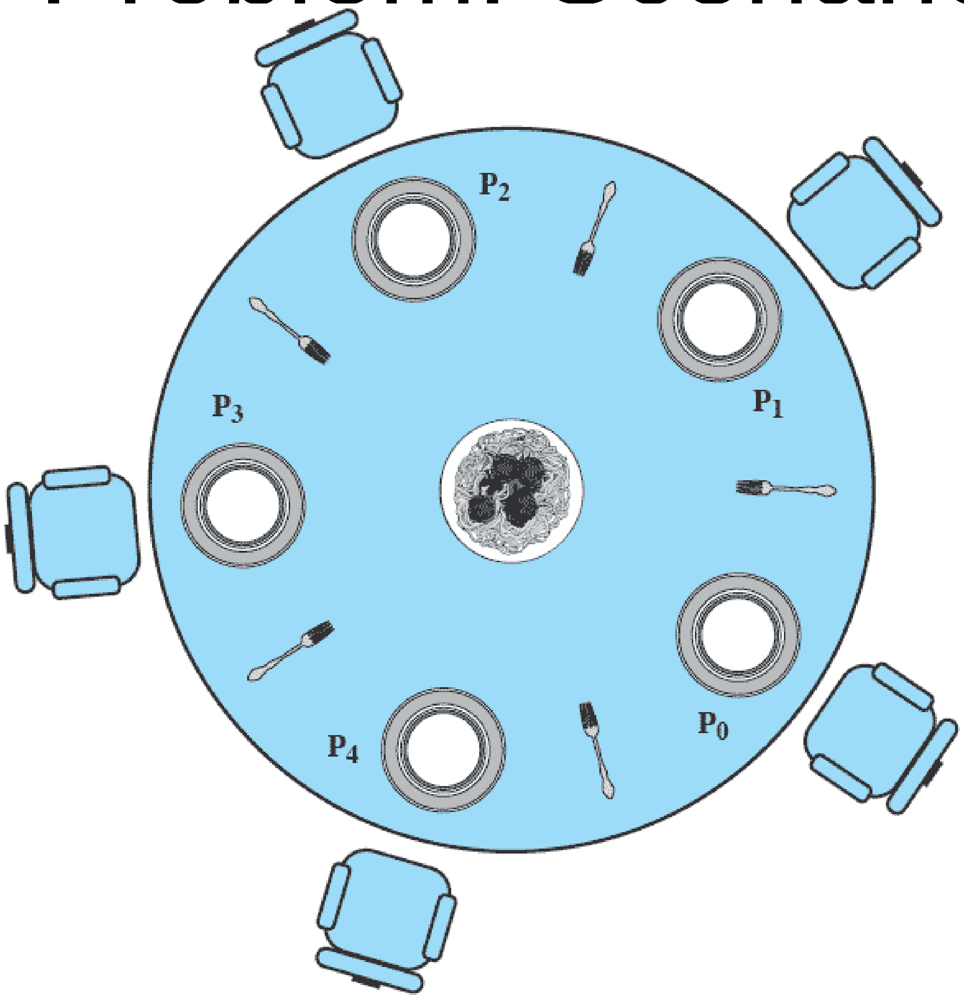


Figure 6.11 Dining Arrangement for Philosophers

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