

Chapter 7: Deadlocks





Chapter 7: Deadlocks

- System Model
- Deadlock Characterization
- Methods for Handling Deadlocks
- Deadlock Prevention
- Deadlock Avoidance





Chapter Objectives

- To develop a description of deadlocks, which prevent sets of concurrent processes from completing their tasks
- To present a number of different methods for preventing or avoiding deadlocks in a computer system

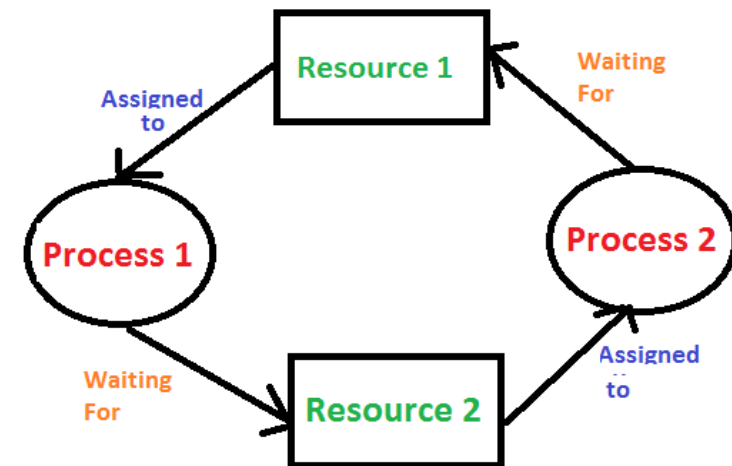




System Model

A **deadlock** is a situation where a set of processes are blocked because each process is holding a resource and waiting for another resource acquired by some other process.

- Consider a system consists of resources
- Resource types R_1, R_2, \dots, R_m
CPU cycles, memory space, I/O devices
- Each resource type R_i has W_i instances.
- Each process utilizes a resource as follows:
 - request
 - use
 - Release





Deadlock Characterization

Deadlock can arise if **four conditions** hold simultaneously

(Necessary Conditions)

- **Mutual exclusion**: Two or more resources are **non-shareable**. only one process at a time can use a resource
- **Hold and wait**: a process holding **at least one resource** is **waiting** to acquire additional resources held by other processes
- **No preemption**: a resource can be released only **voluntarily** by the process holding it, after that process has completed its task
- **Circular wait**: there exists a set $\{P_0, P_1, \dots, P_n\}$ of waiting processes such that P_0 is waiting for a resource that is held by P_1 , P_1 is waiting for a resource that is held by P_2 , ..., P_{n-1} is waiting for a resource that is held by P_n , and P_n is waiting for a resource that is held by P_0 .





Resource-Allocation Graph

A **resource allocation graphs** shows which resource is held by which process and which process is waiting for a resource of a specific kind.

A set of vertices V and a set of edges E .

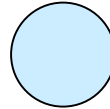
- V is partitioned into two types:
 - $P = \{P_1, P_2, \dots, P_n\}$, the set consisting of all the processes in the system
 - $R = \{R_1, R_2, \dots, R_m\}$, the set consisting of all resource types in the system
- **request edge** – directed edge $P_i \rightarrow R_j$
- **assignment edge** – directed edge $R_j \rightarrow P_i$





Resource-Allocation Graph (Cont.)

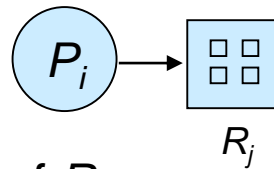
- Process



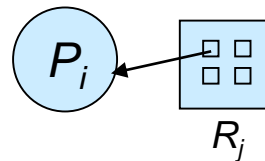
- Resource Type with 4 instances



- P_i requests instance of R_j

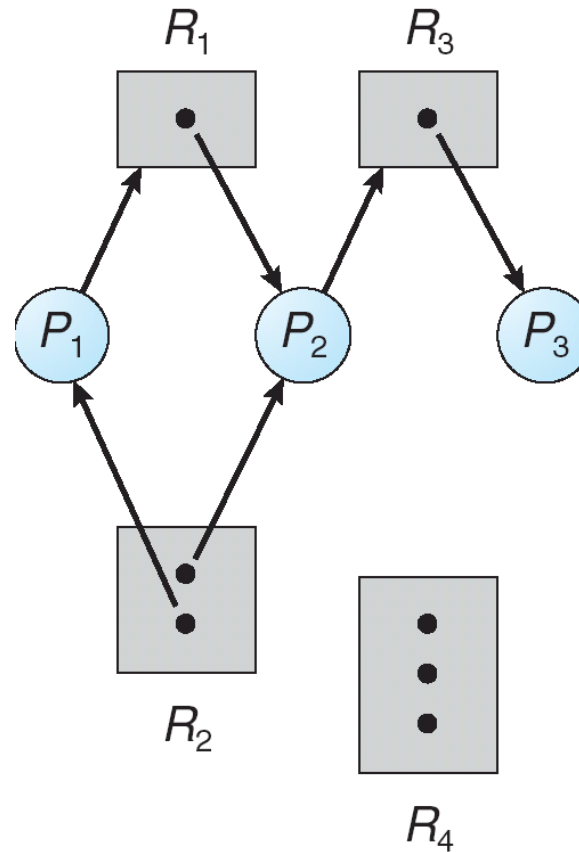


- P_i is holding an instance of R_j



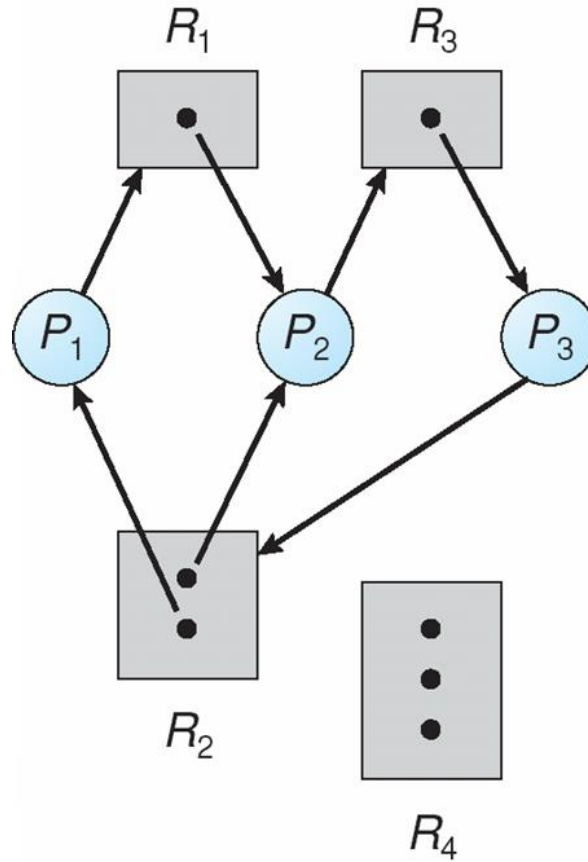


Example of a Resource Allocation Graph



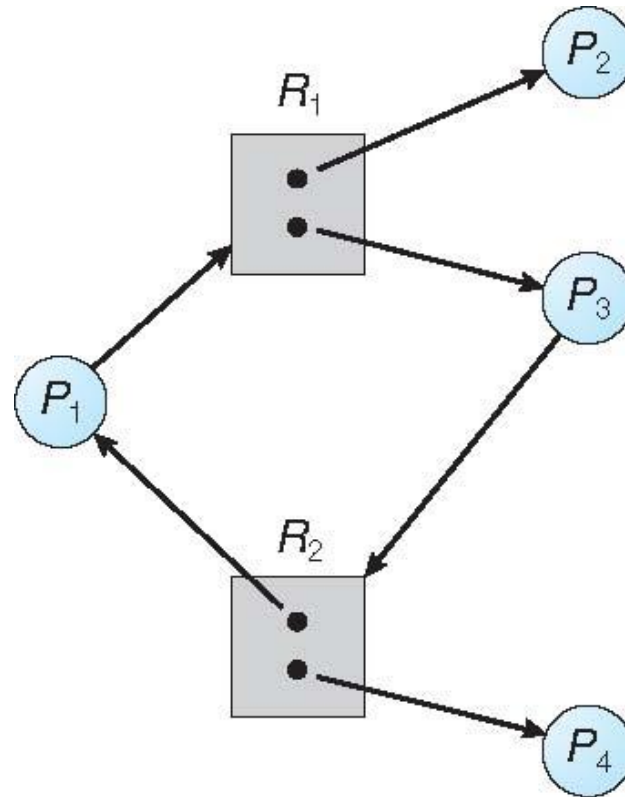


Resource Allocation Graph With A Deadlock





Graph With A Cycle But No Deadlock





Basic Facts

- If graph contains no cycles \Rightarrow no deadlock
- If graph contains a cycle \Rightarrow
 - if only one instance per resource type, then deadlock
 - if several instances per resource type, possibility of deadlock





Methods for Handling Deadlocks

Deal with the deadlock problem in one of **three ways**:

1) **Deadlock prevention or avoidance:**

Prevention: Ensure that the system will **never** enter a deadlock state. provides a set of methods to ensure that at least one of the **necessary conditions cannot hold**. These methods prevent deadlocks by constraining how requests for resources can be made.

Avoidance: requires that the operating system be given **additional information** in advance concerning which resources a process will request and use during its lifetime.

2) **Deadlock detection and recovery:** We can allow the system to enter a deadlocked state, **detect it, and recover**.

3) **Deadlock ignorance:** We can ignore the problem altogether and pretend that deadlocks never occur in the system. **If a deadlock is very rare, then let it happen and reboot the system.**





Deadlock Prevention

Restrain the ways request can be made

- **Mutual Exclusion** – not required for sharable resources (e.g., read-only files); must hold for non-sharable resources
- **Hold and Wait** – must guarantee that whenever a process requests a resource, it does not hold any other resources
 - Require process to **request and be allocated all its resources before it begins execution**
 - Allow process to **request resources** only when the process has **none allocated** to it.
 - Process must release all resources before requesting any new resource

Disadvantages

- **Low resource utilization:** allotted resources may not be used for long time
- **Starvation:** a process may need to wait for long time for a resource





Deadlock Prevention (Cont.)

- **No Preemption** : Resources allocated are not preempted(to solve this use the following protocol)
 - If a process that is holding some resources and requests another resource that cannot be immediately allocated to it, then all resources currently being held are released
 - Preempted resources are added to the list of resources for which the process is waiting
 - Process will be restarted only when it can regain its old resources, as well as the new ones that it is requesting
- **Circular Wait** – impose a total ordering of all resource types, and require that each process requests resources in an increasing order of enumeration to **eliminate circular wait**.

Example: Each resource will be assigned a **numerical number**. A process can request the resources to increase/decrease order of numbering. If the **P1** process is allocated **R5** resources, now next time if **P1** asks for **R4, R3** lesser than R5 such a request will **not be granted**, only a request for resources more than R5 will be granted.





Deadlock Example

```
/* thread one runs in this function */
void *do_work_one(void *param)
{
    pthread_mutex_lock(&first_mutex); // F(first_mutex)=1
    pthread_mutex_lock(&second_mutex); // F(second_mutex)=5
    /** * Do some work */
    pthread_mutex_unlock(&second_mutex);
    pthread_mutex_unlock(&first_mutex);
    pthread_exit(0);
}

/* thread two runs in this function */
void *do_work_two(void *param)
{
    pthread_mutex_lock(&second_mutex);
    pthread_mutex_lock(&first_mutex);
    /** * Do some work */
    pthread_mutex_unlock(&first_mutex);
    pthread_mutex_unlock(&second_mutex);
    pthread_exit(0);
}
```

Only defining order will not help. Developers must follow the ordering to prevent deadlock





Deadlock Avoidance

Requires that the system has some additional ***a priori*** information available

- Simplest and most useful model requires that each process declare the ***maximum number*** of resources of each type that it may need
- The deadlock-avoidance algorithm dynamically examines the resource-allocation state to ensure that there can never be a **circular-wait condition**
- Resource-allocation *state* is defined by the number of **available and allocated resources**, and the maximum demands of the processes





Safe State

- When a process requests an available resource, system must decide if immediate allocation leaves the system in a **safe state**
- System is in **safe state** if there exists a sequence $\langle P_1, P_2, \dots, P_n \rangle$ of all processes in the systems such that for each P_i , the resources that P_i can still request can be satisfied by currently **available resources + resources** held by all the P_j , with $j < i$
- That is:
 - If P_i resource needs are **not immediately available**, then P_i can wait until all **P_j have finished**
 - When P_j is finished, P_i can obtain needed resources, execute, return allocated resources, and terminate
 - When **P_i terminates**, P_{i+1} can obtain its needed resources, and so on

If no such sequence exists then the system safe is said **unsafe**.





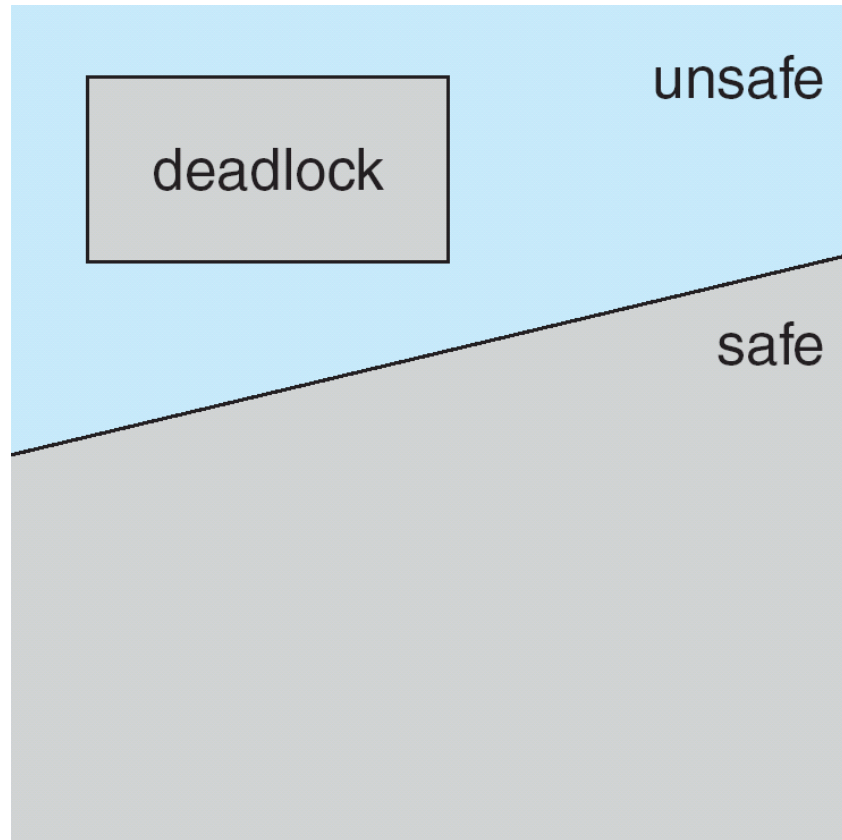
Basic Facts

- If a system is in safe state \Rightarrow no deadlocks
- If a system is in unsafe state \Rightarrow possibility of deadlock
- A safe state is not a deadlocked state.
- Avoidance \Rightarrow ensure that a system will never enter an unsafe state.





Safe, Unsafe, Deadlock State spaces





Avoidance Algorithms

- Single instance of a resource type
 - Use a resource-allocation graph algorithm
- Multiple instances of a resource type
 - Use the banker's algorithm





Resource-Allocation Graph Scheme

- If we have a resource-allocation system with only **one instance of each resource type**, we can use a **variant of the resource-allocation graph** for deadlock avoidance.
- Here, in addition to the **request** and **assignment** edges we introduce a new type of edge, called a **claim edge**
- A **Claim edge** $P_i \rightarrow R_j$ indicated that process P_i may request resource R_j in future
- Claim edge is represented by a **dashed line**





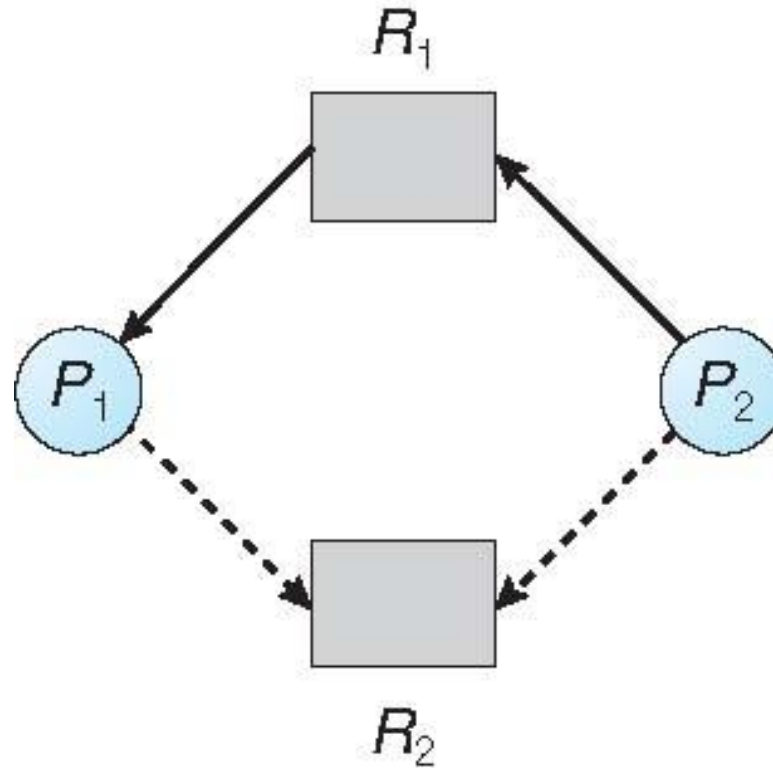
Resource-Allocation Graph Scheme

- When a process **requests** a resource, **claim edge** converts to **request edge**
- When the resource is **allocated** to the process **request edge** converted to an **assignment edge**
- When a resource is **released** by a process, **assignment edge** reconverts to a **claim edge**
- Resources must be claimed *a priori* in the system, thus before process P_i starts executing, all its **claim edges** must already appear in the resource-allocation graph





Resource-Allocation Graph



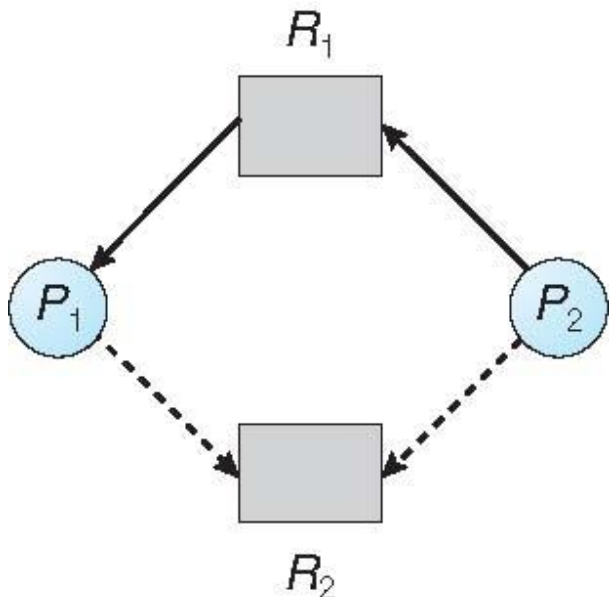
Resource-allocation graph for deadlock avoidance



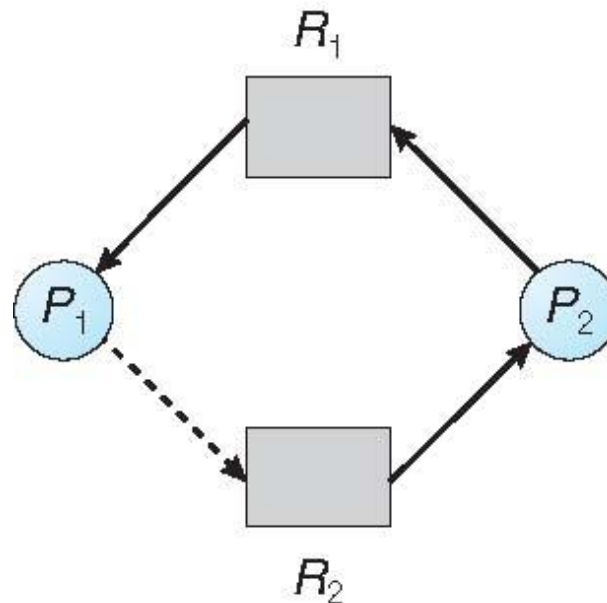


Resource-Allocation Graph Algorithm

- Suppose that process P_i requests a resource R_j
- The request can be granted only if converting the **request edge** to an **assignment edge** does not result in the formation of a **cycle** in the resource allocation graph
- If a cycle is found, in that case, process P_i will have to wait for its requests to be satisfied.
- Consider the example shown, that illustrate the formation of cycle



Resource-allocation graph
for deadlock avoidance



An unsafe state in a
resource-allocation graph





Banker's Algorithm

- The resource-allocation-graph algorithm is not applicable to a resource allocation system with **multiple instances** of each resource type
- Each process must declare the maximum number of instances of each resource type that it may need
- When a user requests a set of resources, the system must determine whether the allocation of these resources will leave the system in a safe state
- If it will, the resources are allocated; otherwise, the process must wait until some other process releases enough resources.
- When a process gets all its resources it must return them in a finite amount of time





Data Structures for the Banker's Algorithm

Let n = number of processes, and m = number of resources types.

Following **data structures** are maintained to implement the banker's algorithm.

- **Available:** Vector of length m . If $available[j] = k$, there are k instances of resource type R_j available
- **Max:** $n \times m$ matrix. If $Max[i,j] = k$, then process P_i may request at most k instances of resource type R_j
- **Allocation:** $n \times m$ matrix. If $Allocation[i,j] = k$ then P_i is currently allocated k instances of R_j
- **Need:** $n \times m$ matrix. If $Need[i,j] = k$, then P_i may need k more instances of R_j to complete its task

$$Need[i,j] = Max[i,j] - Allocation[i,j]$$





Safety Algorithm

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

Work = Available

Finish [i] = **false** for $i = 0, 1, \dots, n-1$

2. Find an i such that both:

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

(b) **Need** _{i} ≤ **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 a safe state





Resource-Request Algorithm for Process P_i

Next, describe the algorithm for determining whether requests can be safely granted.

Let, **$Request_i$** = request vector for process P_i . If **$Request_i[j] = k$** then process P_i wants k instances of resource type R_j

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 of Banker's 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 :

	<u>Allocation</u>	<u>Max</u>	<u>Available</u>
	<i>A B C</i>	<i>A B C</i>	<i>A B C</i>
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 (Cont.)

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

	<u>Need</u>		
	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 $\langle P_1, P_3, P_4, P_0, P_2 \rangle$ satisfies safety criteria





Applying the Safety algorithm on the given system,

Step 1 of Safety Algo

$m=3, n=5$

Work = Available

Work =

3	3	2
---	---	---

0 1 2 3 4

Finish =

false	false	false	false	false
-------	-------	-------	-------	-------

Step 2:

For $i = 0$

Need₀ = 7, 4, 3

Finish [0] is false and Need₀ > Work

So P₀ must wait

But Need ≤ Work

Step 2:

For $i = 1$

Need₁ = 1, 2, 2

Finish [1] is false and Need₁ < Work

So P₁ must be kept in safe sequence

Step 3

Work = Work + Allocation₁

Work =

5	3	2
---	---	---

0 1 2 3 4

Finish =

false	true	false	false	false
-------	------	-------	-------	-------

Step 2:

For $i = 2$

Need₂ = 6, 0, 0

Finish [2] is false and Need₂ > Work

So P₂ must wait

Step 2:

For $i = 3$

Need₃ = 0, 1, 1

Finish [3] is false and Need₃ < Work

So P₃ must be kept in safe sequence

Step 3

Work = Work + Allocation₃

Work =

7	4	3
---	---	---

0 1 2 3 4

Finish =

false	true	false	true	false
-------	------	-------	------	-------

Step 2:

For $i = 4$

Need₄ = 4, 3, 1

Finish [4] is false and Need₄ < Work

So P₄ must be kept in safe sequence

Step 3

Work = Work + Allocation₄

Work =

7	4	5
---	---	---

0 1 2 3 4

Finish =

false	true	false	true	true
-------	------	-------	------	------

Step 2:

For $i = 0$

Need₀ = 7, 4, 3

Finish [0] is false and Need₀ < Work

So P₀ must be kept in safe sequence

Step 3

Work = Work + Allocation₀

Work =

7	5	5
---	---	---

0 1 2 3 4

Finish =

true	true	false	true	true
------	------	-------	------	------

Step 2:

For $i = 2$

Need₂ = 6, 0, 0

Finish [2] is false and Need₂ < Work

So P₂ must be kept in safe sequence

Step 3

Work = Work + Allocation₂

Work =

10	5	7
----	---	---

0 1 2 3 4

Finish =

true	true	true	true	true
------	------	------	------	------

Step 4

Finish [i] = true for $0 \leq i \leq n$

Hence the system is in Safe state

The safe sequence is P₁, P₃, P₄, P₀, P₂





Example: P_1 Request (1,0,2)

- Check that Request \leq Available (that is, $(1,0,2) \leq (3,3,2) \Rightarrow$ true

	<u>Allocation</u>	<u>Need</u>	<u>Available</u>
	A B C	A B C	A B C
P_0	0 1 0	7 4 3	2 3 0
P_1	3 0 2	0 2 0	
P_2	3 0 2	6 0 0	
P_3	2 1 1	0 1 1	
P_4	0 0 2	4 3 1	

- Executing safety algorithm shows that sequence $\langle P_1, P_3, P_4, P_0, P_2 \rangle$ satisfies safety requirement
- Can request for (3,3,0) by P_4 be granted?
- Can request for (0,2,0) by P_0 be granted?





Applying the Safety algorithm on the given system,

What will happen if process P_1 requests one additional instance of resource type A and two instances of resource type C?

Request₁ = $\begin{matrix} A & B & C \\ 1, & 0, & 2 \end{matrix}$

To decide whether the request is granted we use Resource Request algorithm

Step 1
 $\begin{matrix} 1, 0, 2 & 1, 2, 2 \\ \text{Request}_1 < \text{Need}_1 \end{matrix}$ ✓

Step 2
 $\begin{matrix} 1, 0, 2 & 3, 3, 2 \\ \text{Request}_1 < \text{Available} \end{matrix}$ ✓

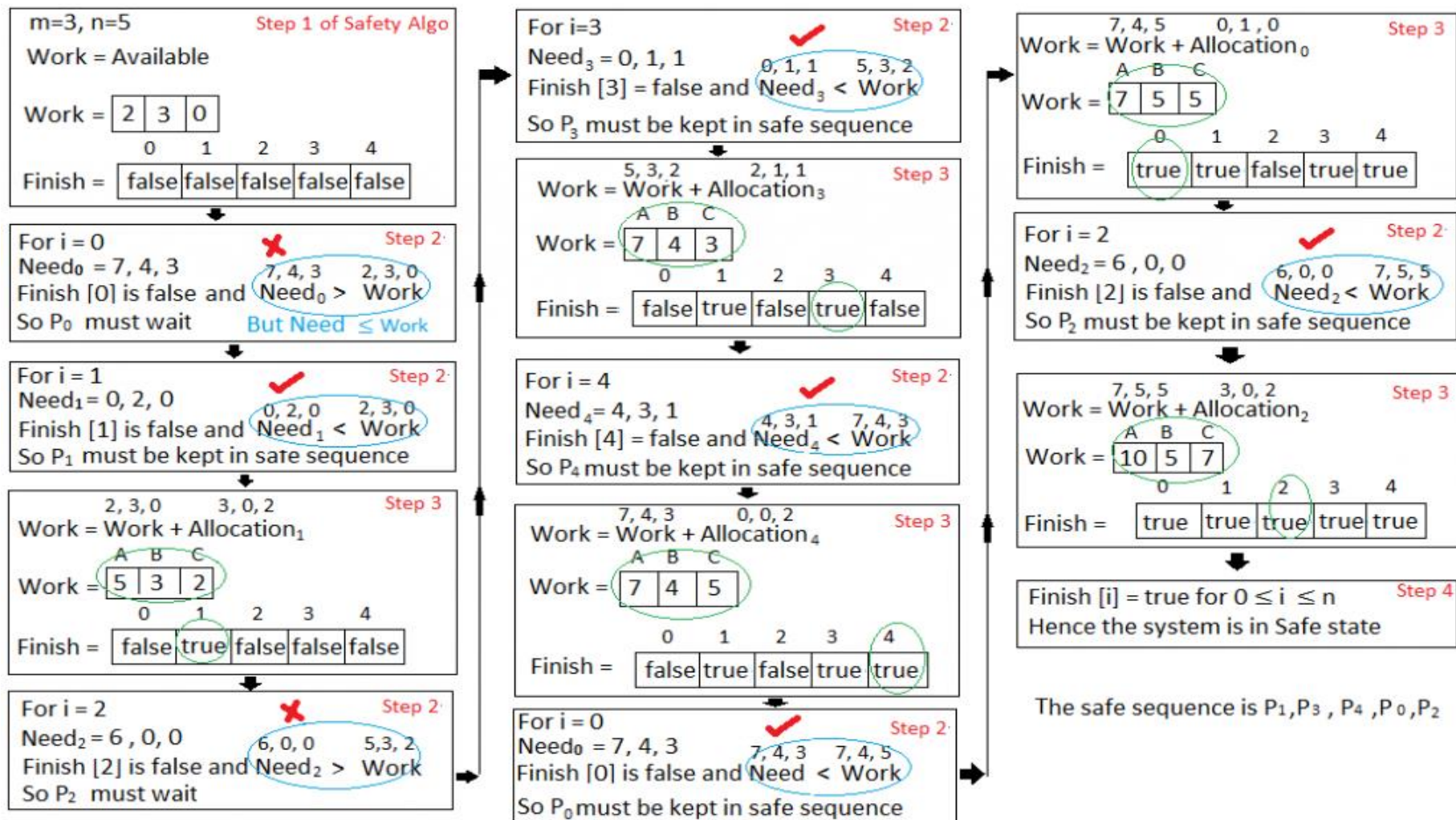
Step 3						
Available = Available – Request ₁						
Allocation ₁ = Allocation ₁ + Request ₁						
Need ₁ = Need ₁ - Request ₁						
Process	Allocation			Need		
	A	B	C	A	B	C
P ₀	0	1	0	7	4	3
P ₁	3	0	2	0	2	0
P ₂	3	0	2	6	0	0
P ₃	2	1	1	0	1	1
P ₄	0	0	2	4	3	1
				Available		
				2	3	0





Applying the Safety algorithm on the given system,

We must determine whether this new system state is safe. To do so, we again execute Safety algorithm on the above data structures.





Example (Cont.)

- P_2 requests an additional instance of type **C**

	<u>Request</u>		
	A	B	C
P_0	0	0	0
P_1	2	0	2
P_2	0	0	1
P_3	1	0	0
P_4	0	0	2

- State of system?
 - Can reclaim resources held by process P_0 , but insufficient resources to fulfill other processes; requests
 - Deadlock exists, consisting of processes P_1 , P_2 , P_3 , and P_4



End of Chapter 7

