

Chapter 7: Deadlocks

Prof. Li-Pin Chang
National Chiao Tung University

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

System Model

- Resource R_1, R_2, \dots, R_m
 - CPU cycles, memory space, I/O devices
- Each resource type R_i has W_i instances.
 - For example, DMA channels
- Each process utilizes a resource as follows:
 1. request
 2. use
 3. release

```

/* thread_one runs in this function */
void *do_work_one(void *param)
{
    pthread_mutex_lock(&first_mutex);
    pthread_mutex_lock(&second_mutex);
    /**
     * 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);
}

```

DEADLOCK CHARACTERIZATION

Deadlock Characterization

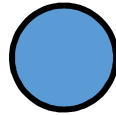
- If a deadlock arises, then the four conditions hold *simultaneously*
 - **Mutual exclusion**: only one process at a time can use a resource
 - **Hold and wait**: a process holding at least one resource is waiting to acquire additional resources held by other processes
 - **No preemption**: a resource 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 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.)

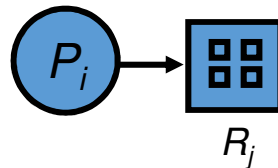
- Process



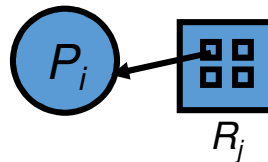
- Resource Type with 4 instances



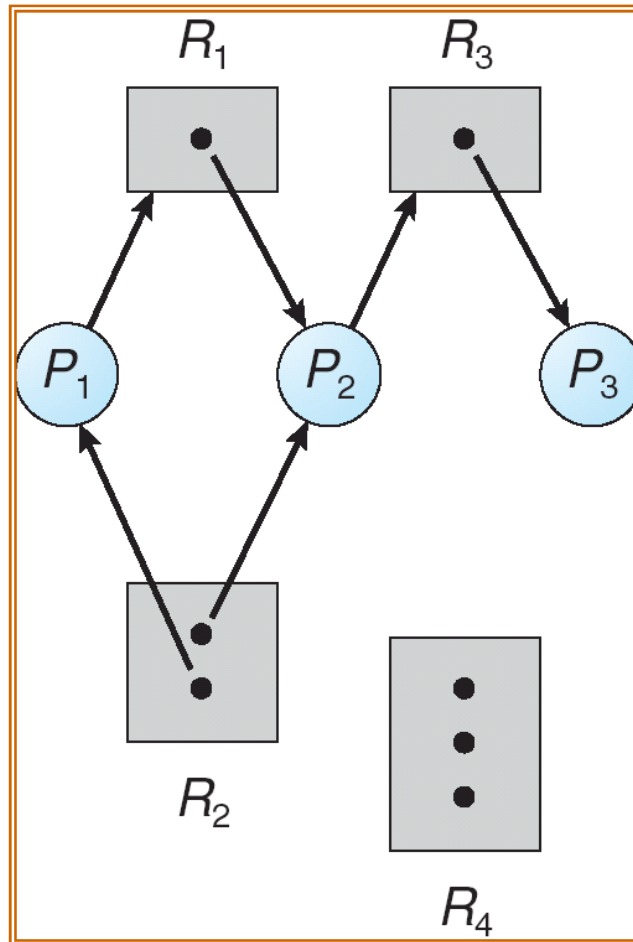
- P_i requests instance of R_j



- P_i is holding an instance of R_j



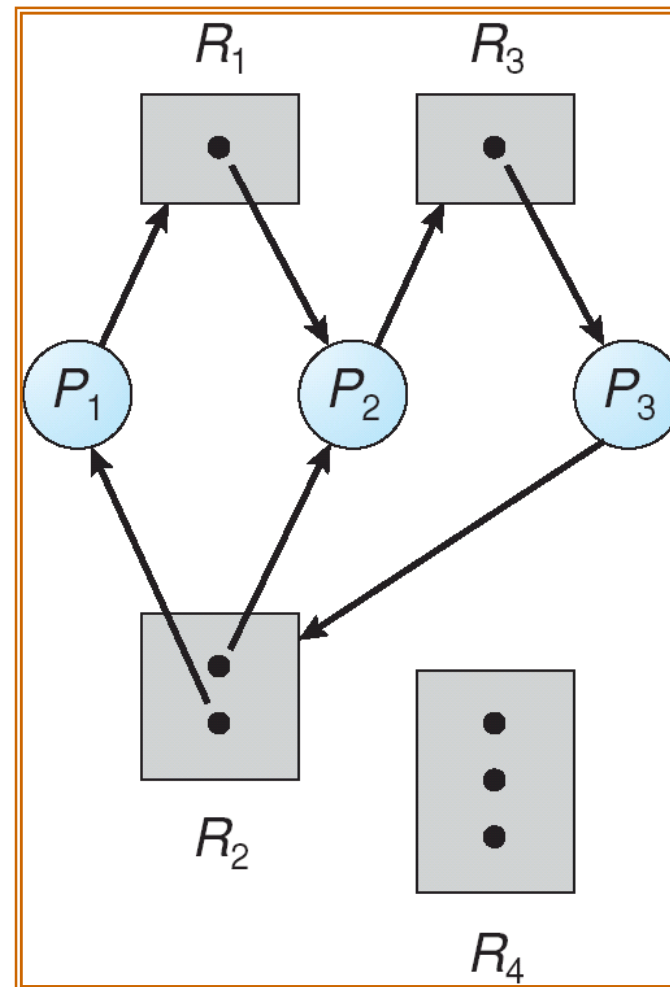
Example of a Resource Allocation Graph



Resource Allocation Graph With A Deadlock

The system is deadlocked

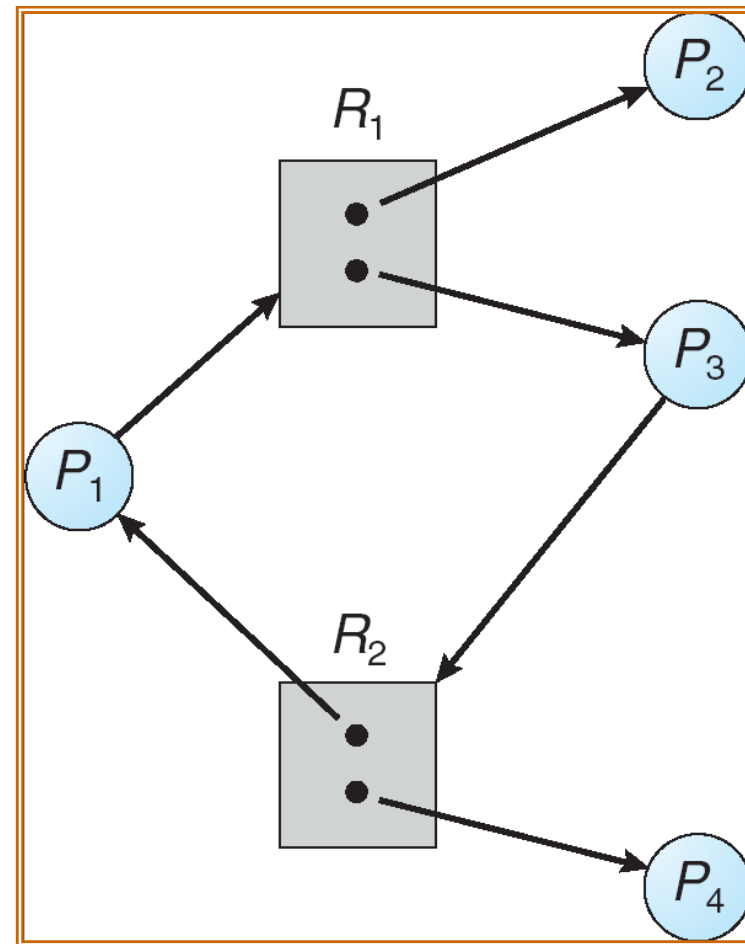
There is a cycle in the graph



Resource Allocation Graph With A Cycle But No Deadlock

The system is **not** deadlocked

There is a cycle in the graph



Basic Facts

- Resources have single instance
 - There is a cycle \leftrightarrow deadlock
- Resources have multiple instances
 - Deadlock \rightarrow there is a cycle
- If graph contains no cycles \Rightarrow no deadlock
- If graph contains a cycle \Rightarrow
 - Resources have single instance, then deadlock
 - Resources have multiple instances, then *possible* deadlock

METHODS FOR HANDLING DEADLOCKS

Methods for Handling Deadlocks

- Ensure that the system will never enter a deadlock state (**prevention or avoidance**),
- allow the system to enter a deadlock state and then recover (**detection and recovery**), or
- ignore the problem and pretend that deadlocks never occur in the system; used by most operating systems, including UNIX.

Methods for Handling Deadlocks

- Ensure that the system will never enter a deadlock state
 - Deadlock **prevention**: a set of rules that guarantees that one or more the necessary conditions **never** happens (changing the programming model or the way OSes managing resources)
 - Deadlock **avoidance**: to **test** whether a request to resources is safe or not; a request may be delayed even when the requested resource is available

DEADLOCK PREVENTION

Deadlock Prevention

- Mutual Exclusion – must be true for serially reusable resources
- Hold and Wait – must guarantee that whenever a process requests a resource, it is not holding any other resources.
 - All or none
 - $[R1-----[R2-----]-----] \rightarrow [Rv-----]$
 - Low concurrency among processes due to long critical sections

Deadlock Prevention (Cont.)

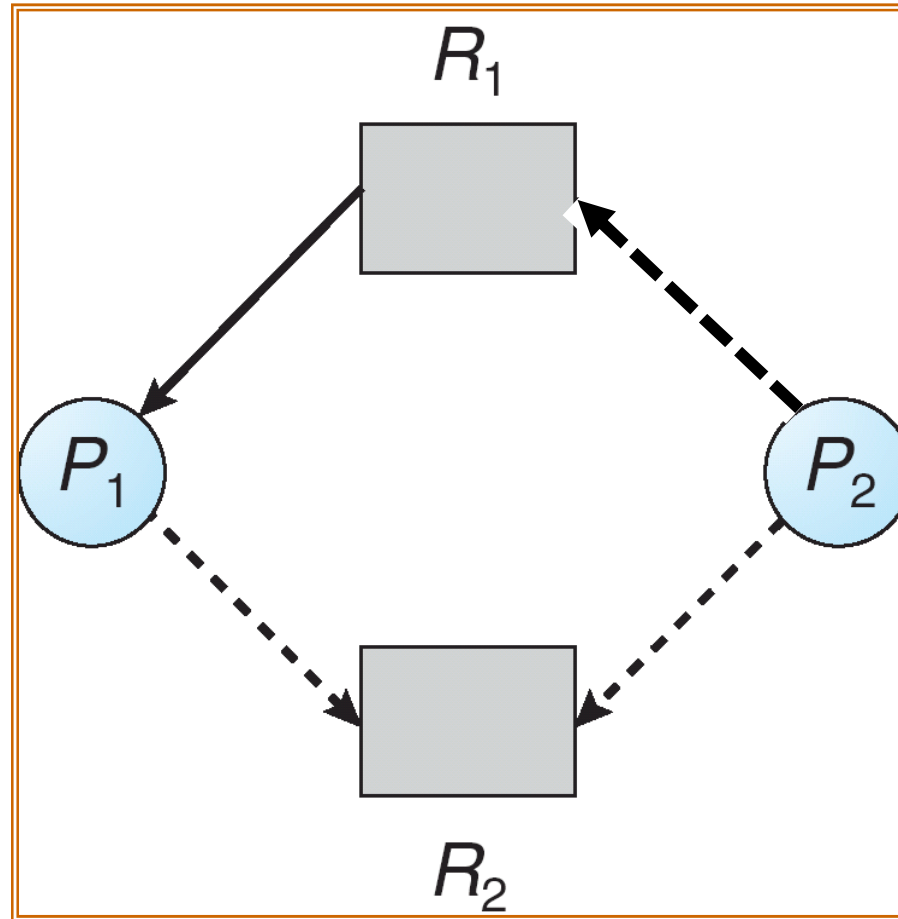
- No Preemption –
 - If a process (victim) holds a resource R but is waiting for another resource, R will be preempted when another process tries to acquire R
 - The victim process will be **restarted** when R is available again
 - Requiring a **checkpointing** mechanism
- Circular Wait – impose **a total ordering or partial ordering on all resource types**, and require that each process requests resources in an increasing order of enumeration.
 - E.g., $R1 \rightarrow R2$ but no $R2 \rightarrow R1$
 - Either total ordering or partial ordering

DEADLOCK AVOIDENCE

Deadlock Avoidance

- 1 instance per resource
 - Deadlock \leftrightarrow cycle (s)
 - Resource acquisition must not create cycle(s) in the resource allocation graph
- Deadlock avoidance based on cycle detection in resource allocation graphs

Resource-Allocation Graph For Deadlock Avoidance



Claim edge: may use a resource at some time

Request edge: is requesting a resource

Assignment edge: is holding a resource

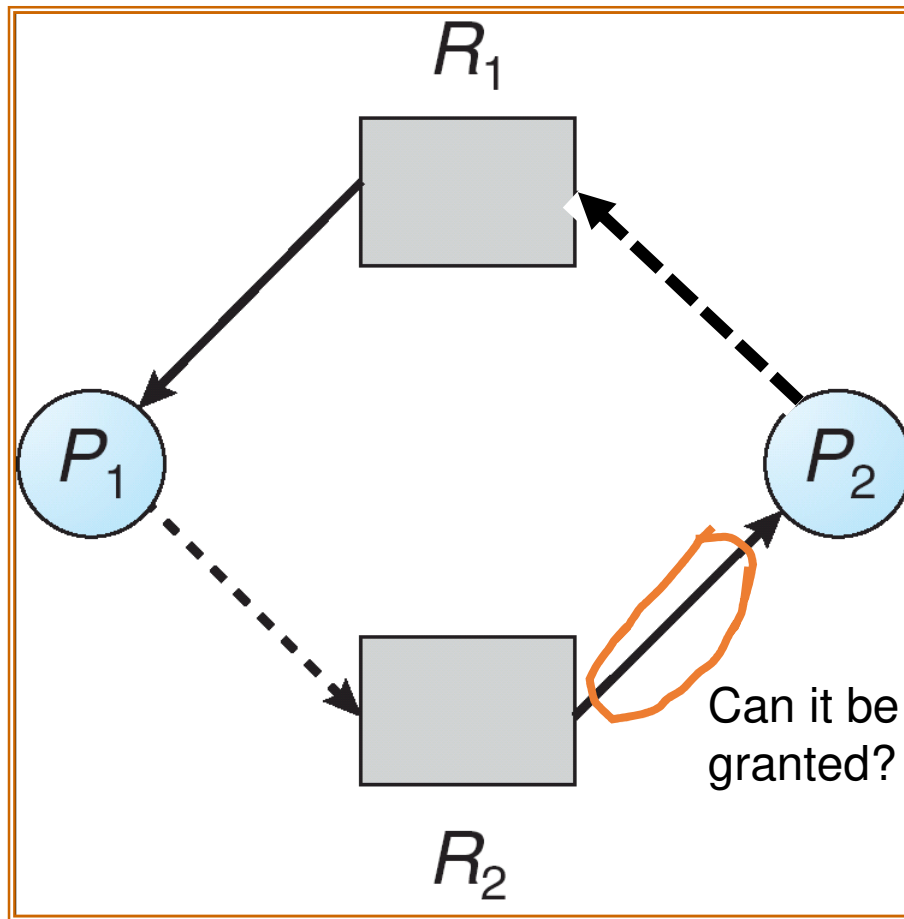
Resource-Allocation Graph Algorithm

- **Claim edge** $P_i \rightarrow R_j$ indicated that process P_j **may** request resource R_j ; represented by a dashed line
- Claim edge converts to **request edge** when a process requests a resource
- When a resource is released by a process, **assignment edge** reconverts to a claim edge
- Resources must be claimed *a priori* in the system.

Deadlock Avoidance for 1-Instance Resources

1. Initially, put all claim edges
2. When a process requests a resource, convert the claim edge into request edge
3. If the resource is available, **tentatively** change the request edge into assignment edge and check if there are any new cycles(s) in the resource-allocation graph
4. If new cycle(s) exist, revert the allocation edge back to request edge and put the process waiting; Otherwise, the resource is allocated to the process

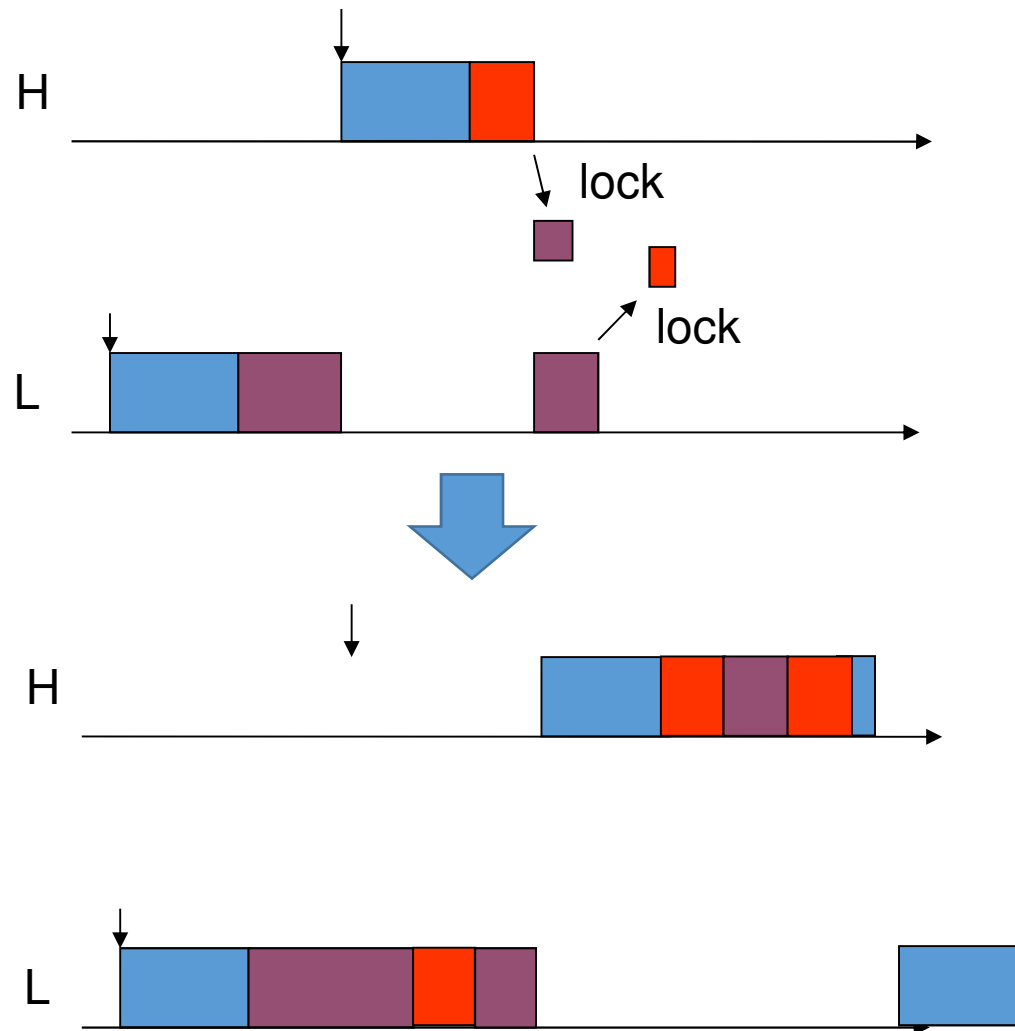
Unsafe State In Resource-Allocation Graph



To detect cycles before an request can be granted

How to detect cycle(s) in the resource-allocation graph?

Another Deadlock Avoidance Strategy: Highest Locker's Protocol in RTOS



A process's priority
is boosted to the
highest among the
lockers' priorities

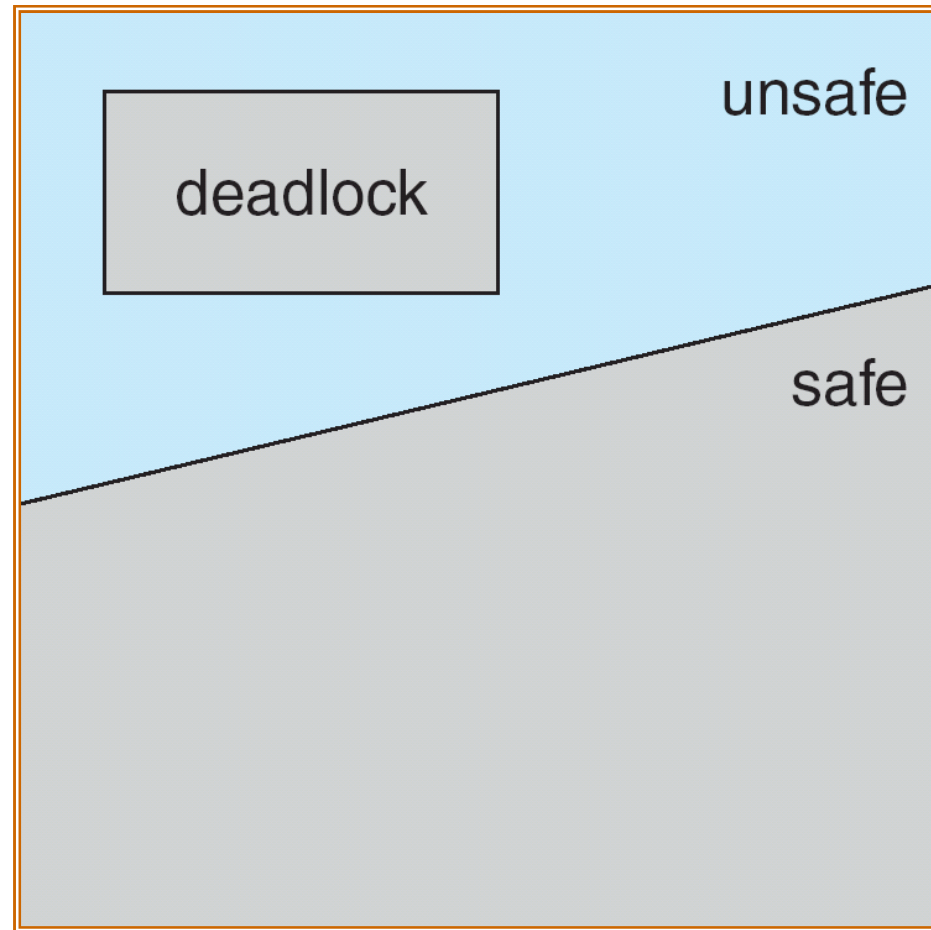
Deadlock Avoidance

- N instances per resource
 - The graph-based approach is still applicable
- A more general approach
 - Safe/unsafe-state method
 - A system is safe → the system has no deadlock
 - The system must always be in a safe state; resource acquisition cannot put the system in a unsafe state
 - Need a definition on “safe state”

Basic Facts

- If a system is in safe state \Rightarrow no deadlocks
- If a system is in unsafe state \Rightarrow possibility of deadlock
- Avoidance \Rightarrow ensure that a system will never enter an unsafe state

Safe, Unsafe , Deadlock State



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 safe sequence of all processes**
- Sequence $\langle P_1, P_2, \dots, P_n \rangle$ is safe if 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 $j < i$**
 - 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

Banker's Algorithm

- Multiple instances
- Each process must *a priori* claim maximum use
- When a process requests a resource it may have to wait
- 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.

- *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

n: process #; m: resource #

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

Work = *Available*

Finish [*i*] = *false* for $0 \sim n$

2. Find and *i* such that both:

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

(b) $Need_i \leq Work$

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

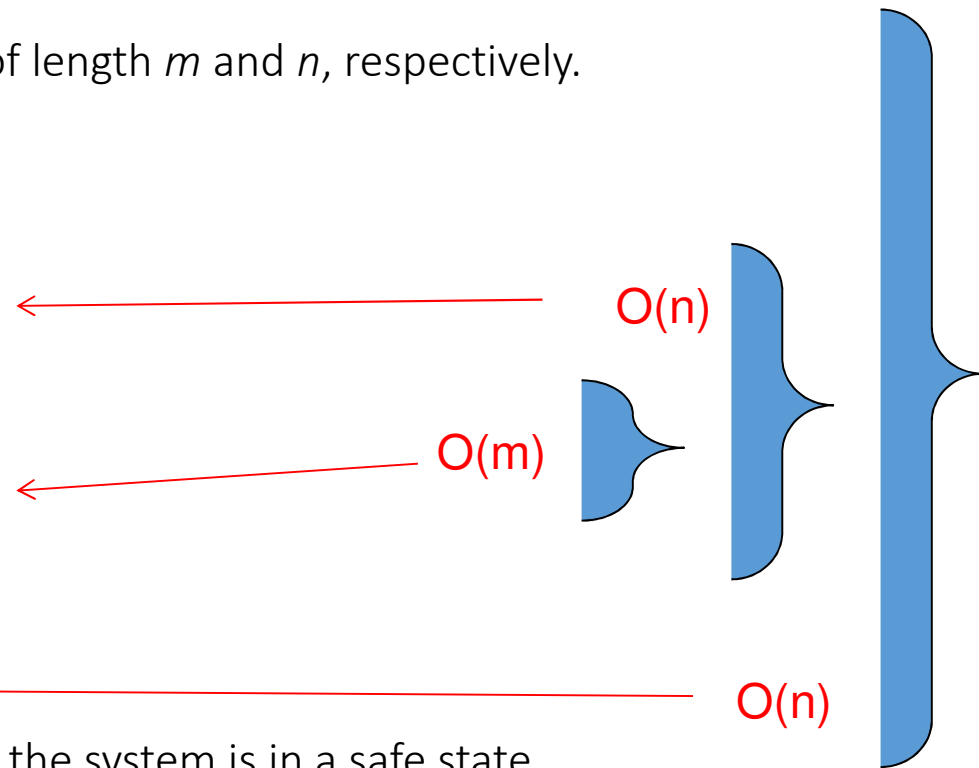
3. *Work* = *Work* + *Allocation*_{*i*}

Finish[*i*] = *true*

go to step 2.

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

$O(m \cdot n^2)$



Resource-Request Algorithm for Process P_i

Request = 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>
	$A \ B \ C$	$A \ B \ C$	$A \ B \ C$
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 (Cont.)

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

	<u>Need</u>
	<i>A B C</i>
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 $\langle P_1, P_3, P_4, P_2, P_0 \rangle$ satisfies safety criteria.

Example (Cont.)

	Allocation	Need	Available
	A B C	A B C	A B C
P0	0 1 0	7 4 3	3 3 2
P1	2 0 0	1 2 2	
P2	3 0 2	6 0 0	
P3	2 1 1	0 1 1	
P4	0 0 2	4 3 1	

- The system is in a safe state since the sequence $\langle P_1, P_3, P_4, P_2, P_0 \rangle$ satisfies safety criteria.

Example P_1 Request (1,0,2) (Cont.)

- Check that Request \leq Available (that is, $R_1(1,0,2) \leq (3,3,2) \Rightarrow \text{true}$).

	<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 1	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?

If P_0 (0,2,0) was made...

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

Discussions: Safe State

- Why all processes make their largest requests in the checking procedure?
- Rationale: if processes do not request the largest amount of resources, the problem becomes easier

End of Chapter 7