Chapter 7

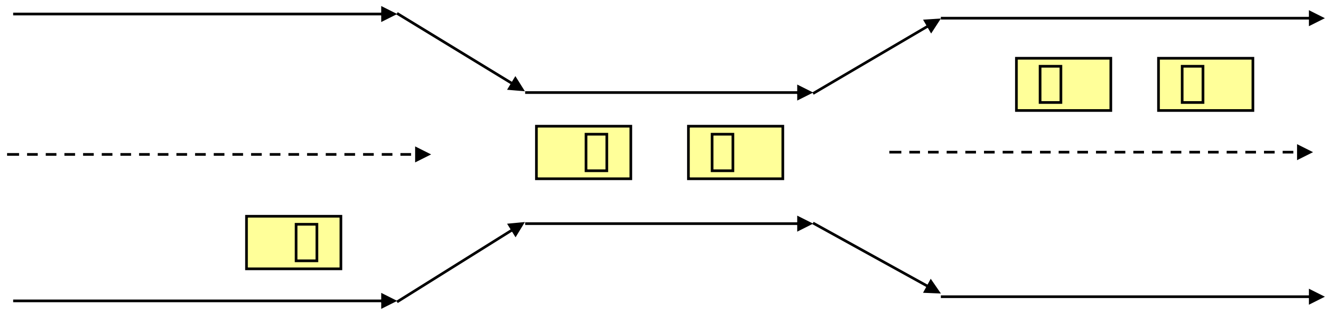
Deadlocks

The Deadlock Problem

- A deadlock consists of a set of blocked processes, each holding a resource and waiting to acquire a resource held by another process in the set
- Example
  - A system has 2 disk drives
  - P_1 and P_2 each hold one disk drive and each needs the other one
- Example
  - Semaphores A and B, initialized to 1

P_0	P_1
wait (A);	wait(B)
wait (B);	wait(A)

Bridge Crossing Example



- Traffic only in one direction
- The resource is a one-lane bridge
- If a deadlock occurs, it can be resolved if one car backs up (pre-empt resources and rollback)
- Several cars may have to be backed up if a deadlock occurs
- Starvation is possible

System Model

- Resource types $R_1, R_2, . . ., R_m$

  CPU cycles, memory space, I/O devices

- Each resource type R_i has *1 or more* instances
- Each process utilizes a resource as follows:
  - **Request :** Process Pi requests for resource, if request is not guaranteed, Pi must wait until it acquires resource
  - **Use :** Process can operate on resource (can use that resource)
  - **Release:** Process releases the resource (after using)

- **Request and Release** resource are **System Calls**, done through **wait() and signal()**

Deadlock Characterization

Necessary Conditions for Deadlock to Occur:

Deadlock can arise if **<u>four</u>** conditions **hold simultaneously**.

These conditions must occur for deadlock to occur.

- **1. Mutual Exclusion:** only one process at a time can use a resource
- **2. Hold and Wait:** a process holding at least one resource is waiting to acquire additional resources held by other processes

Deadlock Characterization

Necessary Conditions for Deadlock to Occur:

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

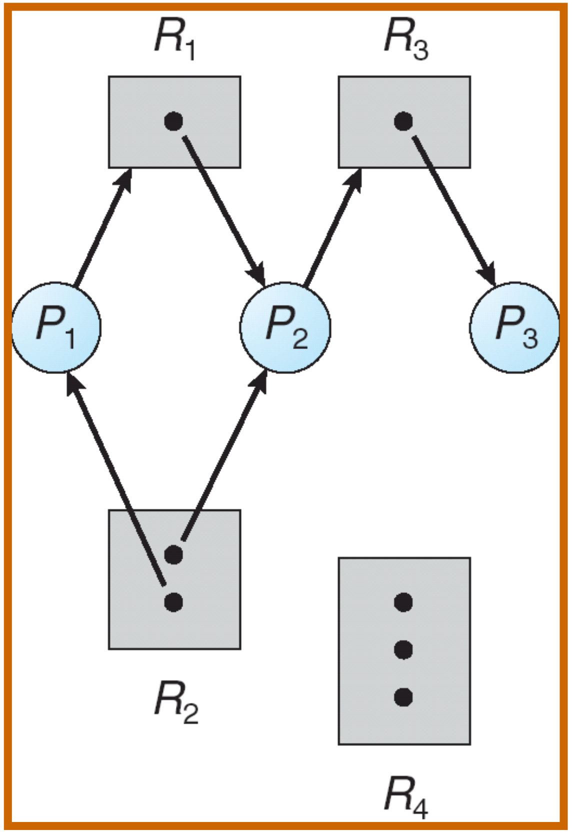
- **3. No Pre-emption:** a resource can be released only voluntarily by the process holding it after that process has completed its task, resources can't be pre-empted

- **4. Circular Wait:** There exists a set $\{P_0, P_1, \ldots, P_0\}$ 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.

Representation of Deadlock

Resource-Allocation Graph

- **Deadlocks can be described in terms of Directed Graph, called Resource Allocation Graph.**
- Set of vertices *V* and a set of edges *E*.
  - V is partitioned into two types:
    - $P = \{P_1, P_2, \ldots, P_n\}$, the set consisting of all active processes in the system
      $R = \{R_1, R_2, \ldots, R_m\}$, the set consisting of all resource types in the system



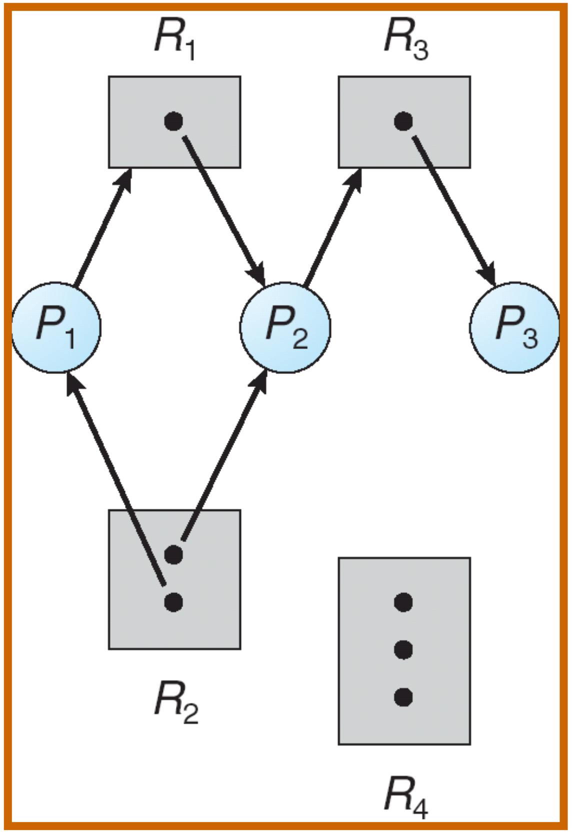
Resource-Allocation Graph

Request Edge – directed edge $P_i \rightarrow R_j$

Process Pi is requesting for instance of Resource Rj

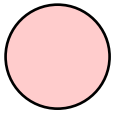
Assignment Edge – directed edge $R_j \rightarrow P_i$

Resource Rj is allocated to process Pi.

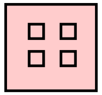


Resource-Allocation Graph (Cont.)

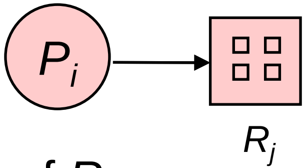
- Process



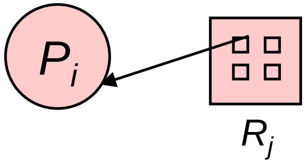
- Resource Type with 4 instances



- P_i requests instance of R_j

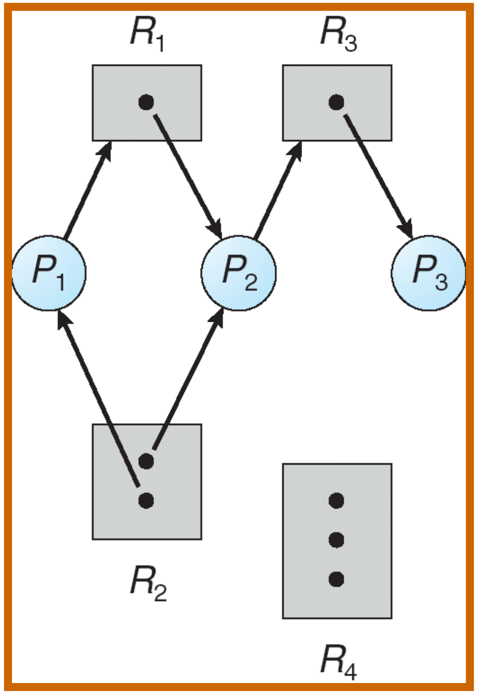


- P_i is holding an instance of R_j

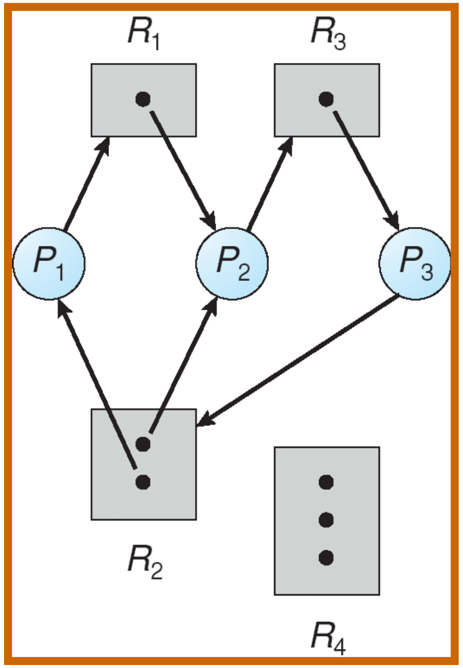


Resource Allocation Graph With A Deadlock

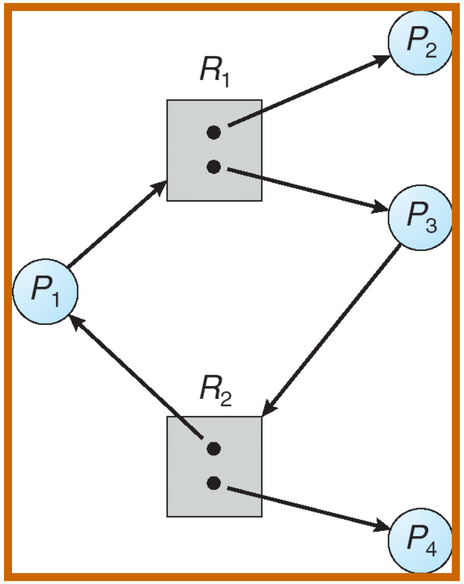
Before P_3 requested an instance of R_2



After P_3 requested an instance of R_2



Graph With A Cycle But No Deadlock



Process P_4 may release its instance of resource type R_2. That resource can then be allocated to P3, thereby breaking the cycle.

Relationship of cycles to deadlocks

- If a resource allocation graph contains no cycles $\Rightarrow$ no deadlock
- If a resource allocation graph contains a cycle and if only one instance exists per resource type $\Rightarrow$ deadlock
- If a resource allocation graph contains a cycle and if several instances exists per resource type $\Rightarrow$ possibility of deadlock

Methods for Handling Deadlocks

- **Prevention**
  - Ensure that the system will *never* enter a deadlock state
- **Avoidance**
  - Ensure that the system will *never* enter an unsafe state
- **Detection**
  - Allow the system to enter a deadlock state and then recover
- **Do Nothing**
  - Ignore the problem and let the user or system administrator respond to the problem; used by most operating systems, including Windows and UNIX

1. Deadlock Prevention

•**To prevent deadlock, we can restrain the ways that a request can be made**

•**Deadlock Prevention Methods:** Prevent deadlocks by making a constraint on how requests for resources can be made

- **Mutual Exclusion** – The mutual-exclusion condition must hold for non-sharable resources
  - (Where as) Shared resources such as read-only files do not lead to deadlocks.
  - Some resources, such as printers and tape drives, require exclusive access by a single process.

Deadlock Prevention

- **To prevent deadlock, we can restrain the ways that a request can be made**
- **Deadlock Prevention Methods:** Prevent deadlocks by making a constraint on how requests for resources can be made

- **Hold and Wait** – OS must guarantee that whenever a process requests a resource, **it <u>does not</u> hold any other resources**
  - **1. Allocate all its resources <u>before</u> process begins execution**
  - **2. Allow a process to request resources <u>only</u> when the process has no resource**

Drawback / Result: Low resource utilization; starvation possible

Deadlock Prevention (Cont.)

- **No Pre-emption:**
  - One approach is that if a process is forced to wait when it requests new resources, then all r**esources previously held by this process are implicitly released**, (preempted), forcing this process to re-acquire the old resources along with the new resources in a single request (**if one process goes to waiting state all its previously held resources must be released implicitly)**
  - <u>**Pre-empted resources are added to the list of resources**</u> for which the process is waiting
  - A process will be restarted only when it can regain its old resources, as well as the new ones that it is requesting

Deadlock Prevention (Cont.)

- **Circular Wait**
  - One way to avoid circular wait is to number all resources, and to require that processes request resources only in strictly increasing (or decreasing) order.
  - In other words, in order to request resource Rj, a process must first release all Ri such that i >= j.

1. Assign a number to resources. If R7 is requested by process no other resource lower than 7 can be granted to process.
2. In case process needs R3, then it will have to release R7 in order to get R3.

One big challenge in this scheme is determining the relative ordering of the different resources

2. Deadlock Avoidance

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

- 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
- **A resource-allocation *state* is defined** by the number of available and allocated resources, and the maximum demands of the processes

State and Safe State

- State: State of system represents currently allocated resources to the process. (data held by process at sometime)

- Safe State: If the system can allocate available resources to the process in some order to avoid deadlock.

- Unsafe State:
  - OS can not prevent processes from requesting the resources
  - Allocating the resources, which may leads to deadlock.

Safe State

- A system is in a safe state only if there exists a <u>safe sequence</u>

<u>if resources are allocated to the process in that sequence deadlock doesnot occur.</u>

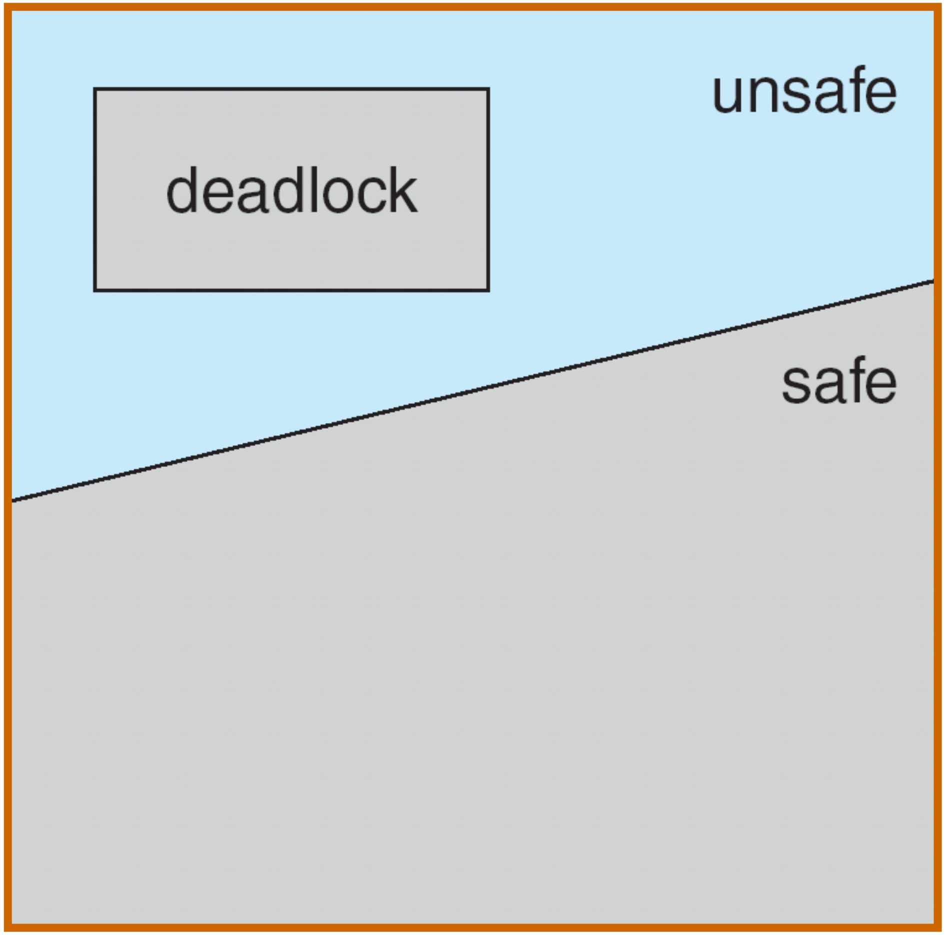
- A sequence of processes <P_1, P_2, …, P_n> is a safe sequence,

If request made by process can be satisfied with the available resources + already held resources

Safe State (continued)

- 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



Example of safe and unsafe state

- Consider system with 12 DVD and 3 processes: (p0, p1, p2

Process	Maximum Need	Currently Held	Need
P0	10	5	5
P1	4	2	2
P2	9	2	7

Avoidance algorithms

- For a **single instance of a resource** type, use a resource-allocation graph

- For **multiple instances of a resource** type, use the banker's algorithm

Resource-Allocation Graph Scheme: For Single Instance

- Introduce a new kind of edge called a claim edge (a dotted line)

- *Claim edge* $P_i \dashrightarrow R_j$ indicates that process P_j may request resource R_j; which is represented by a dashed line

- **A claim edge converts to a request edge** when a process **requests** a resource

- **A request edge converts to an assignment edge** when the resource is **allocated** to the process

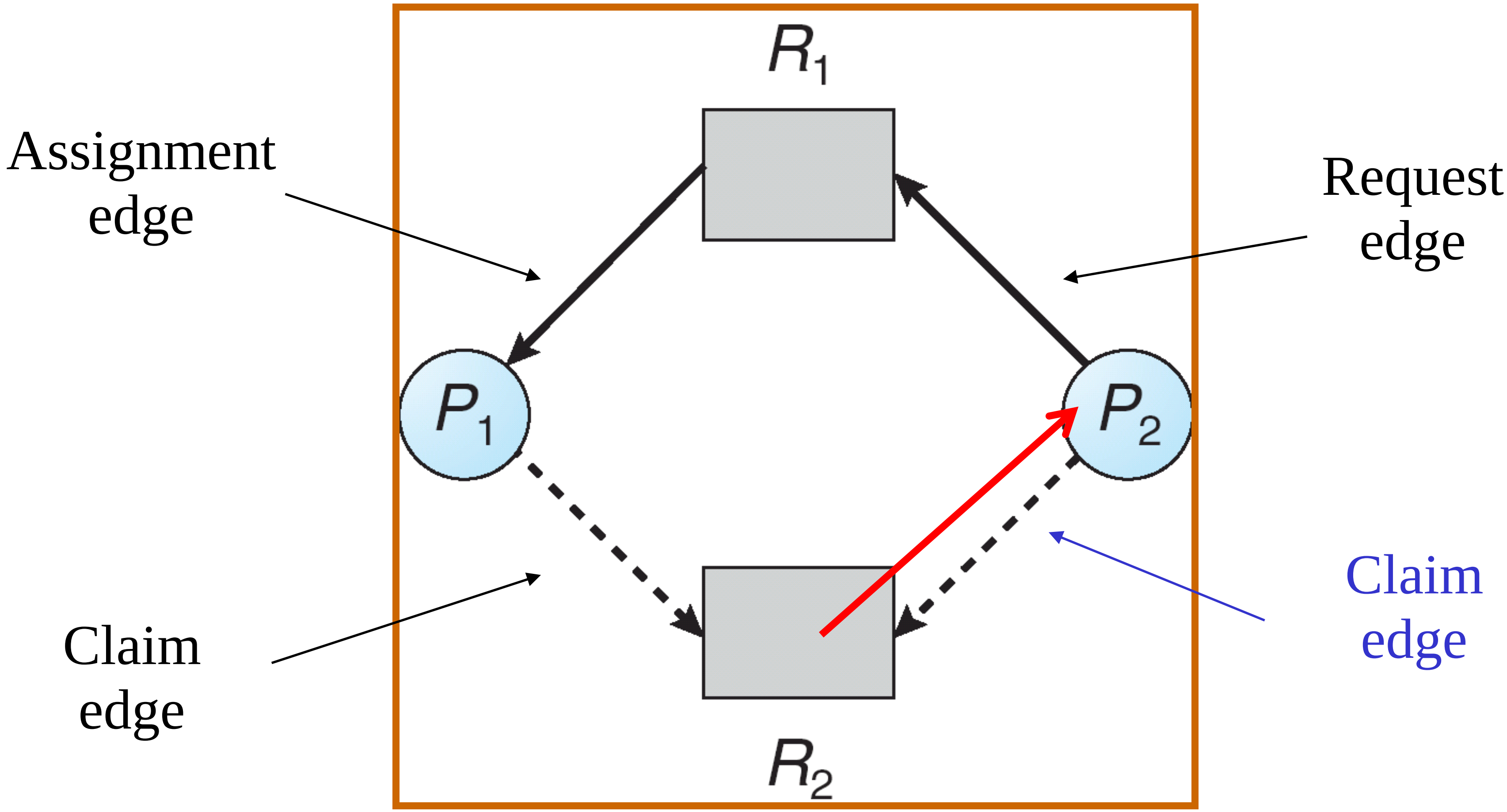
- When a resource is **released** by a process, an **assignment edge reconverts to a claim edge**

- Resources must be **claimed *a priori*** in the system

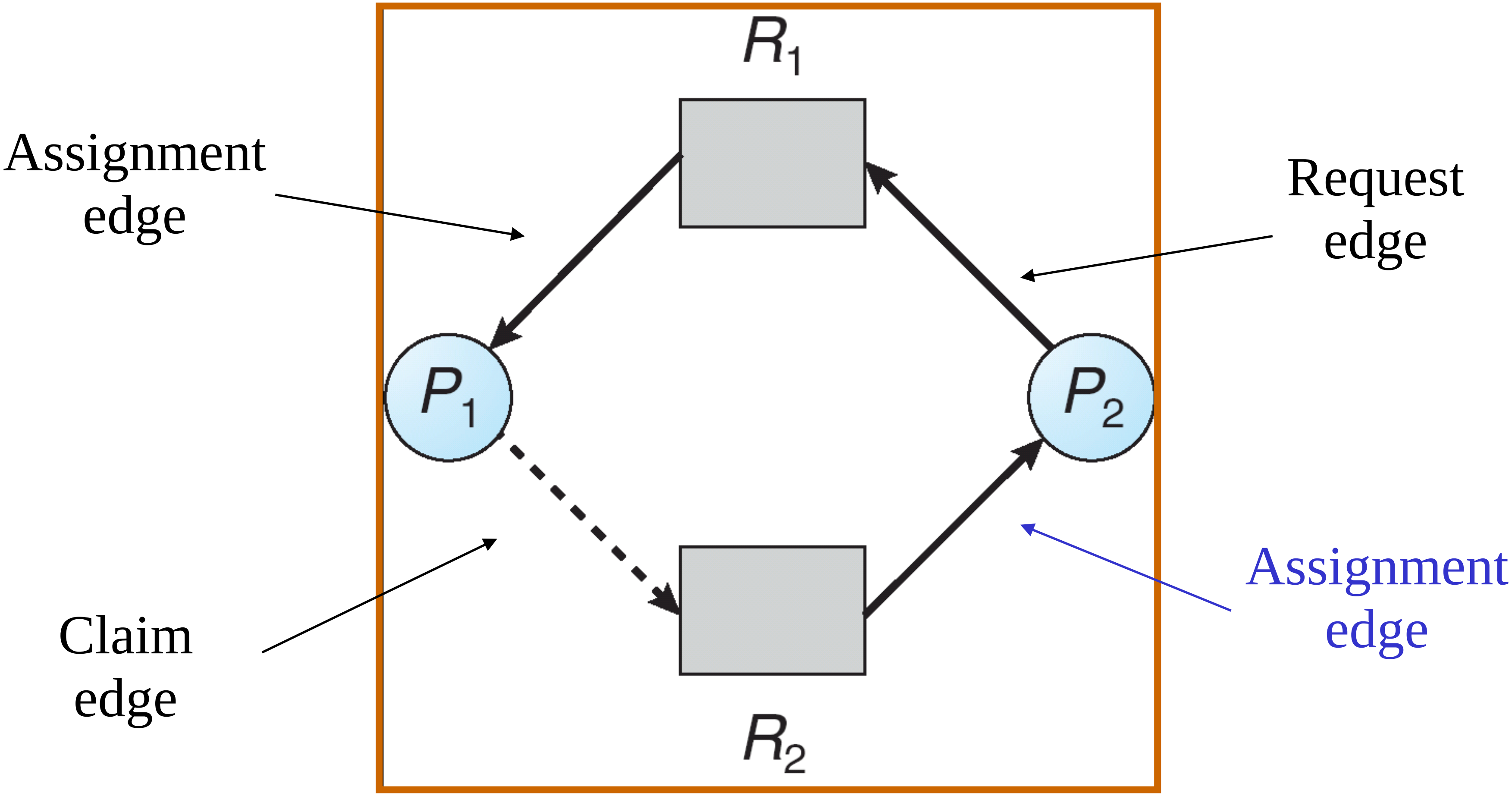
Resource-Allocation Graph Scheme: For Single Instance

- Claim Edge →Request Edge : when a process **requests** a resource
- Request edge → Assignment Edge : when the resource is **allocated** to the process
- Assignment edge → Claim Edge: When a resource is **released** by a process,

Resource-Allocation Graph with Claim Edges



Unsafe State In Resource-Allocation Graph



Resource-Allocation Graph Algorithm: For Multiple Instances

Banker's Algorithm

- Used when there exists **multiple** instances of a resource type
- Each process must **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.

i: Process and j: Resource instances

- **Available**: Vector of length m.

If available [j] = k, there are k instances of resource type R_j available.

- **Max**: Maximum available resources that a process can request.

If Max [i , j] = k, then process P_i may request at most k instances of resource type R_j.

Data Structures for the Banker's Algorithm

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

- **Allocation**: Resources that can be allocated

If Allocation[i , j] = k then P_i is currently allocated k instances of R_{j}.

- **Need**: 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]`

Bankers Algorithm

Let `Request` be the request vector for process P_i.

If $\mathbf{Request_i[j] = k}$ then process P_i wants k instances of resource type R_j

1. If $\mathbf{Request_i} \le \mathbf{Need_i}$ go to step 2. Otherwise, raise error condition, since process has exceeded its maximum claim

2. If $\mathbf{Request_i} \le \mathbf{Available}$, go to step 3. Otherwise P_i must wait, since resources are not available

Bankers Algorithm

3. Pretend to allocate requested resources to P_i by modifying the state as follows:

$$\textbf{Available} = \textbf{Available} - \textbf{Request}_i$$

$$\textbf{Allocation}_i = \textbf{Allocation}_i + \textbf{Request}_i$$

$$\textbf{Need}_i = \textbf{Need}_i - \textbf{Request}_i$$

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

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
//{Process I needs resources, resources are still available}
```

2. Find an `i` such that

`Finish[i] = false AND 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

Example: Bankers Algorithm

Consider a system with 5 processes P0 to P4, three resource types A,B,C. Resource A has 10 instances , B has 5 instances, C has 7 instances. Suppose foll. Is system snapshot at time T0. Find 1. Need Matrix 2. Is system in safe state?

Process	Allocation	Maximum	Available	Need
	A B C	A B C	A B C	A B C
P0	0 1 0	7 5 3	3 3 2	7 4 3
P1	2 0 0	3 2 2		1 2 2
P2	3 0 2	9 0 2		6 0 0
P3	2 1 1	2 2 2		0 1 1
P4	0 0 2	4 3 3		4 3 1
	<7 2 5>			

Available= [(10-7), (5-2), (7-5)] = < 3 3 2>

Need= Maximum – Allocation

Available = Available + Allocated

Safe Sequence<P1, P3, P4, P0, P2>

Process	Allocation	Maximum	Available	Need
	A B C	A B C	A B C	A B C
P0	0 1 0	7 5 3		
P1	2 0 0	3 2 2		
P2	3 0 2	9 0 2		
P3	2 1 1	2 2 2		
P4	0 0 2	4 3 3		
	<7 2 5>			

Example: Bankers Algorithm

For p0: need>available

P1: need<available: update available matrix by

Available = Available + Allocated <5 3 2>

P2: can not be fulfilled

P3: Available = Available+allocated <7 4 3>

P4: Available = Available+allocated <7 4 5>

P0: Available = Available+allocated <7 5 5>

P2: Available = Available+allocated <10 5 7>

Safe seq: <P1, P3, P4, P0, P2>

Example: Bankers Algorithm

Find: 1. Whether system is in safe state or not?

2. Safe sequence

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

7.6 Deadlock Detection

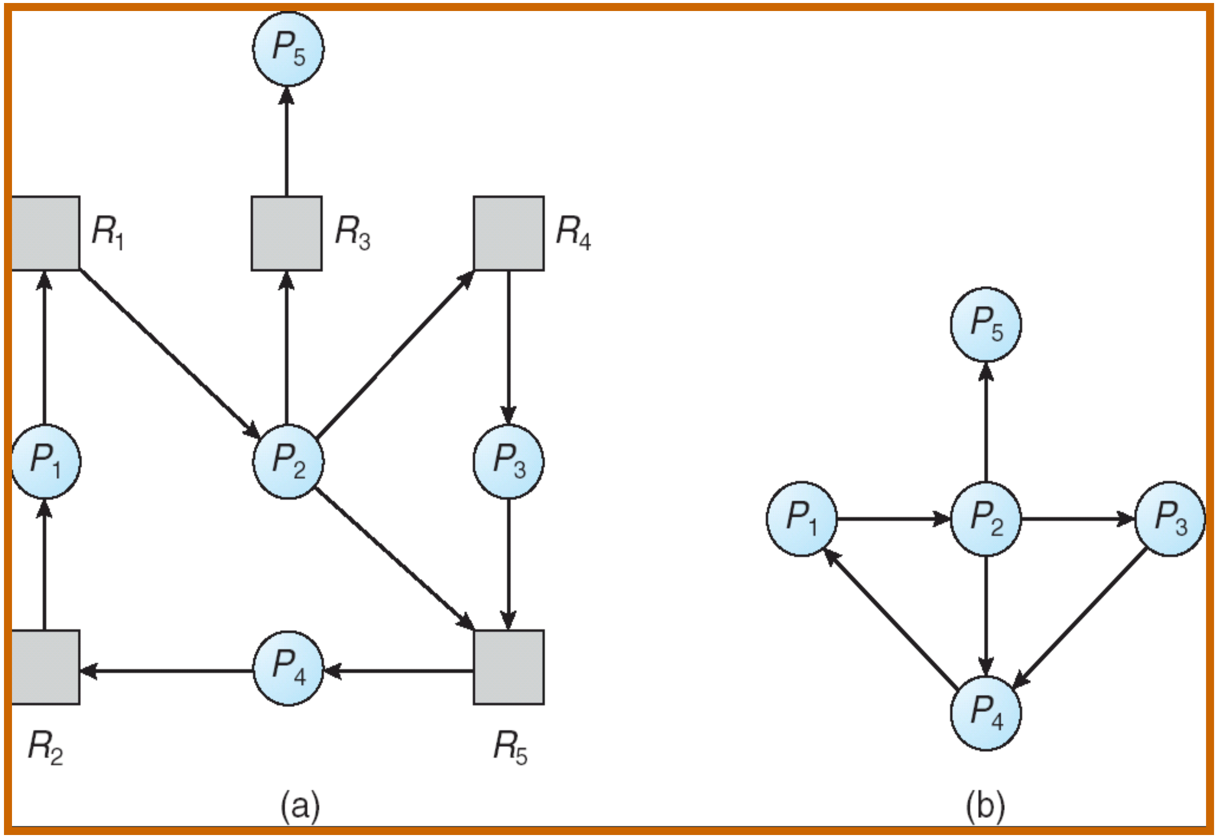
Deadlock Detection

- For deadlock detection, the system must provide
  - An algorithm that **examines the state of the system** to <u>detect</u> whether a deadlock has occurred
  - An algorithm to <u>recover</u> from the deadlock

For Single Instance of Each Resource Type

- **Requires the creation and maintenance of a *wait-for* graph**
  - The graph is obtained by **removing** the resource nodes from a resource-allocation graph and **collapsing** the appropriate edges
  - $P_i \rightarrow R_q \qquad R_q \rightarrow P_j$
  - $P_i \rightarrow P_j$ if P_i is waiting for resource held by P_j.
  - If there is a cycle in Wait-for-Graph, there exists a deadlock

Resource-Allocation Graph and Wait-for Graph



Resource-Allocation Graph Corresponding wait-for graph

For Multiple Instances of a Resource Type

Required data structures:

- ***Available**:* A vector of length *m* indicates the **number of available resources** of each resource type.
- ***Allocation**:* An *n* x *m* matrix defines the **number of resources** of each type **currently allocated** to each process.
- ***Request**:* An *n* x *m* matrix indicates the current request of each process. If Request [i , j] = k, then process P_i is requesting *k* more instances of resource type. R_j.

Detection Algorithm

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

   Work = Available$_m$

   Finish[i] = *false if Allocation !=0*

   = *true if Allocation = 0*

   Request$_i$ ≤ Work

2. Find an `i` such that

   Finish[i] = *false* AND Request$_i$ ≤ Work

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

If resources are available:

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

4. If **Finish[i] == False** for some i then there is a deadlock.

Example: Deadlock Detection

Process	Allocation	Request	Available
	A B C	A B C	A B C
P0	0 1 0	0 0 0	0 0 0
P1	2 0 0	2 0 2	
P2	3 0 3	0 0 0 **<0 0 1>**	
P3	2 1 1	1 0 0	
P4	0 0 2	0 0 2	
	<7 2 6>		

In above situation, there is no dead lock. But if **P2 needs** more resources **< 0 0 1>**

System in in Deadlock now, because no available resources.

Detection-Algorithm Usage

- To invoke the detection algorithm depends on:
  - **How often** is a **deadlock** likely to **occur**?
  - **How many processes** will be **affected by deadlock** when it happens?

- If the detection **algorithm is invoked randomly:**
  - difficult to tell **which process "caused" the deadlock**
- If the detection algorithm is invoked **for every resource request:**
  - will incur a **overhead** in computation time

- A less expensive alternative is to invoke the algorithm when CPU utilization drops **below 40%.**

7.7 Recovery From Deadlock

Recovery from Deadlock

- Let the user or system administrator respond to the problem
- Two Approaches:
  - **Process termination** : terminate processes to break the circular wait.
  - **Resource Pre-emption** : Pre-empt resources from deadlocked processes

Process Termination

- **Abort all deadlocked processes**
  - This approach will break the deadlock, but at great expense

- **Abort one process at a time until the deadlock cycle is eliminated**
  - After each process is aborted, a deadlock-detection algorithm must be re-invoked to determine whether any processes are still deadlocked

- **Many factors may affect which process is chosen for termination**
  - What is the **priority of the process**?
  - How much process has run and **how much time it needs to complete**?
  - **How many** and **what** type of **resources** has the **process used**?
  - **How many** more **resources** does the **process need** in order to finish its task?
  - How many processes will need to be terminated?
  - Is the process interactive or batch?

Resource Pre-emption

- Pre-empt some resources from processes and give these resources to other processes until the deadlock cycle is broken
- When pre-emption is required to deal with deadlocks, then <u>three</u> issues need to be addressed:
  - **Selecting a victim** – Which resources and which processes are to be pre-empted?
  - **Rollback** – If we pre-empt a resource from a process, what should be done with that process?
  - **Starvation** – How do we ensure that starvation will not occur?

That is, how can we guarantee that resources will not always be pre-empted from the same process?

Summary

- Four necessary conditions must hold in the system for a deadlock to occur
  - Mutual exclusion
  - Hold and wait
  - No preemption
  - Circular wait
- Four principal methods for dealing with deadlocks
  - Use some protocol to (1) **prevent** or (2) **avoid** deadlocks, ensuring that the system will never enter a deadlock state
  - Allow the system to enter a deadlock state, (3) **detect** it, **and** then **recover**
    - Recover by **process termination** or **resource preemption**
  - **(4) Do nothing;** ignore the problem altogether and pretend that deadlocks never occur in the system (used by Windows and Unix)
- To prevent deadlocks, we can ensure that **at least one** of the four necessary conditions **never holds**

End of Chapter