Deadlocks

Dr. Yingwu Zhu

Deadlocks

- Synchronization is a live gun – we can easily shoot ourselves in the foot
  - Incorrect use of synchronization can block all processes
  - You have likely been intuitively avoiding this situation already
- More generally, processes that allocate multiple resources generate dependencies on those resources
  - Locks, semaphores, monitors, etc., just represent the resources that they protect
- If one process tries to allocate a resource that a second process holds, and vice-versa, they can never make progress
- We call this situation deadlock, and we'll look at:
  - Definition and conditions necessary for deadlock
  - Representation of deadlock conditions
  - Approaches to dealing with deadlock

Deadlock Definition

- A set of blocked processes each holding a resource and waiting to acquire a resource held by another process in the set.
- Example
  - System has 2 tape drives.
  - P_1 and P_2 each hold one tape drive and each needs another one.
- Example
  - semaphores *A* and *B*, initialized to 1

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

Deadlock Characterization

Deadlock can arise if four conditions hold simultaneously.

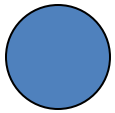
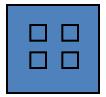
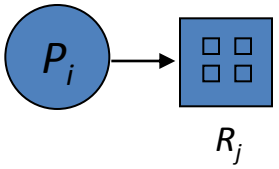
- **Mutual exclusion:** only one process at a time can use a resource.
- **Hold and wait:** a process holding at least one resource is waiting to acquire additional resources held by other processes.
- **No preemption:** a resource can be released only voluntarily by the process holding it, after that process has completed its task.
- **Circular wait:** there exists a set $\{P_0, P_1, \ldots, P_0\}$ 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.

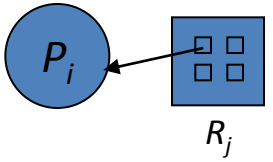
Resource-Allocation Graph

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

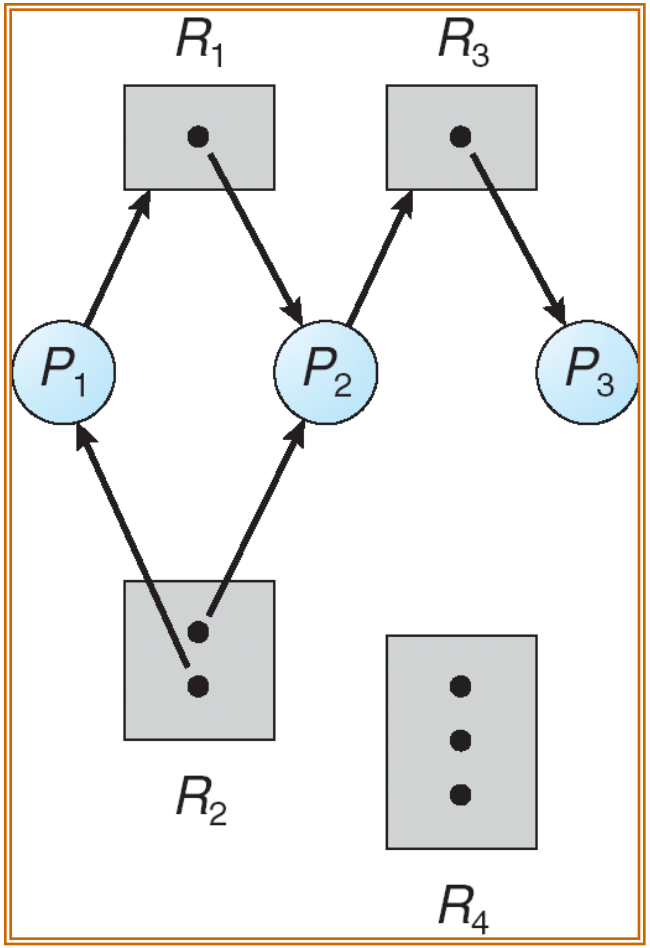
- More precisely describe deadlocks
- ***V*** is partitioned into two types:
  - $P = \{P_1, P_2, \ldots, P_n\}$, the set consisting of all the processes in the system.
  - $R = \{R_1, R_2, \ldots, R_m\}$, the set consisting of all resource types in the system.
  - request edge – directed edge $P_i \rightarrow R_j$
  - assignment edge – directed edge $R_j \rightarrow P_i$
- If the graph has no cycles, deadlock cannot exist
- If the graph has a cycle, deadlock may exist

Resource-Allocation Graph (Cont.)

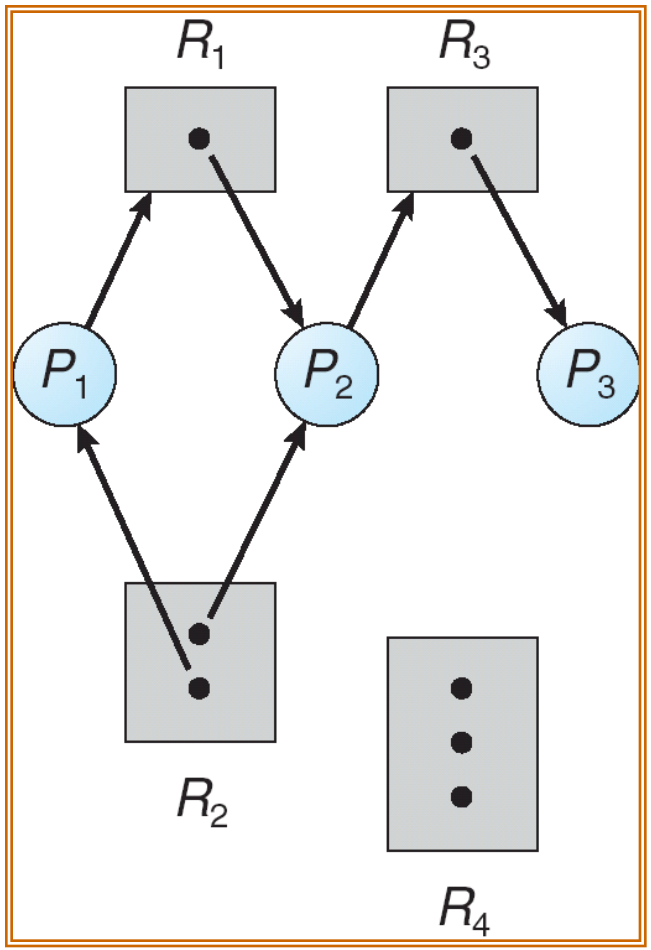
- Process 
- Resource Type with 4 instances 
- P_i requests instance of R_j 

- P_i is holding an instance of R_j 

Example of a Resource Allocation Graph

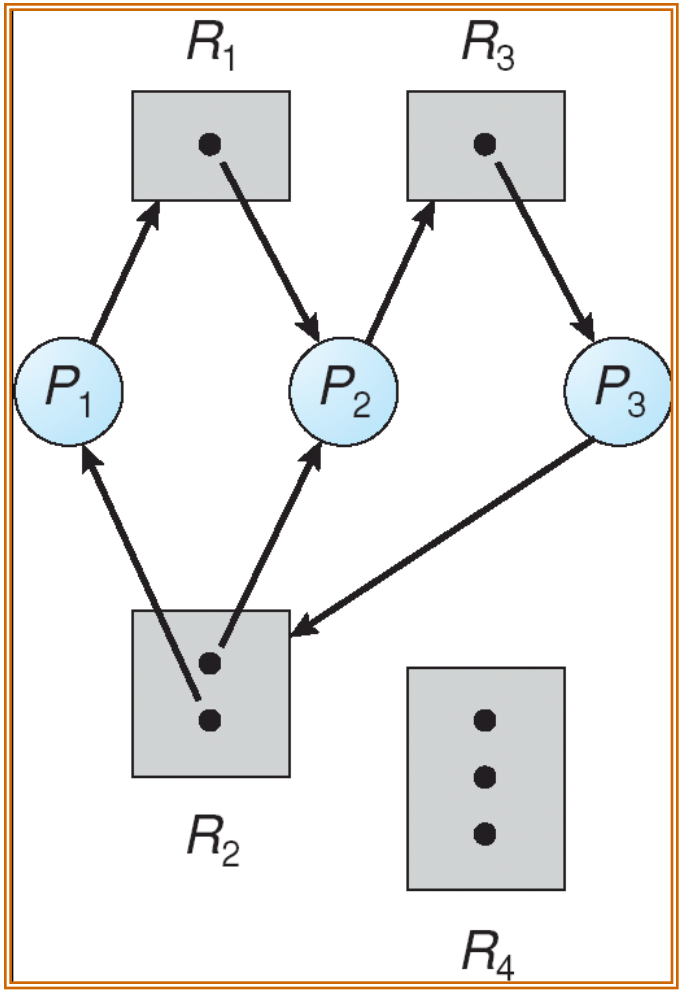


Example of a Resource Allocation Graph



If the graph contains no cycles, then no process is deadlocked

Resource Allocation Graph

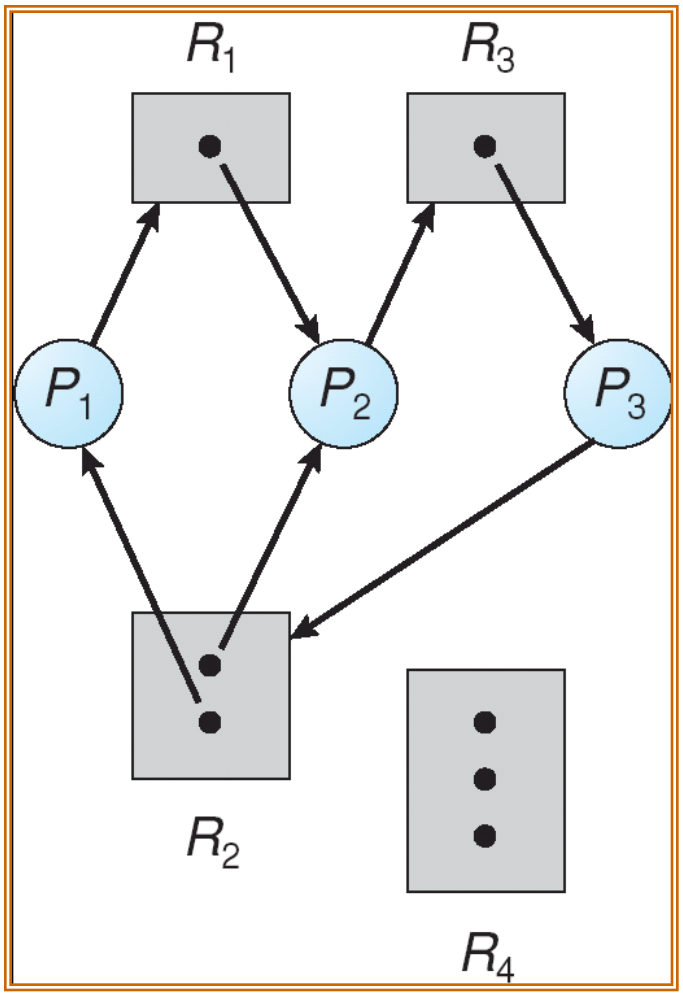


Resource Allocation Graph

If the graph contains a cycle, then a deadlock **MAY** exist!

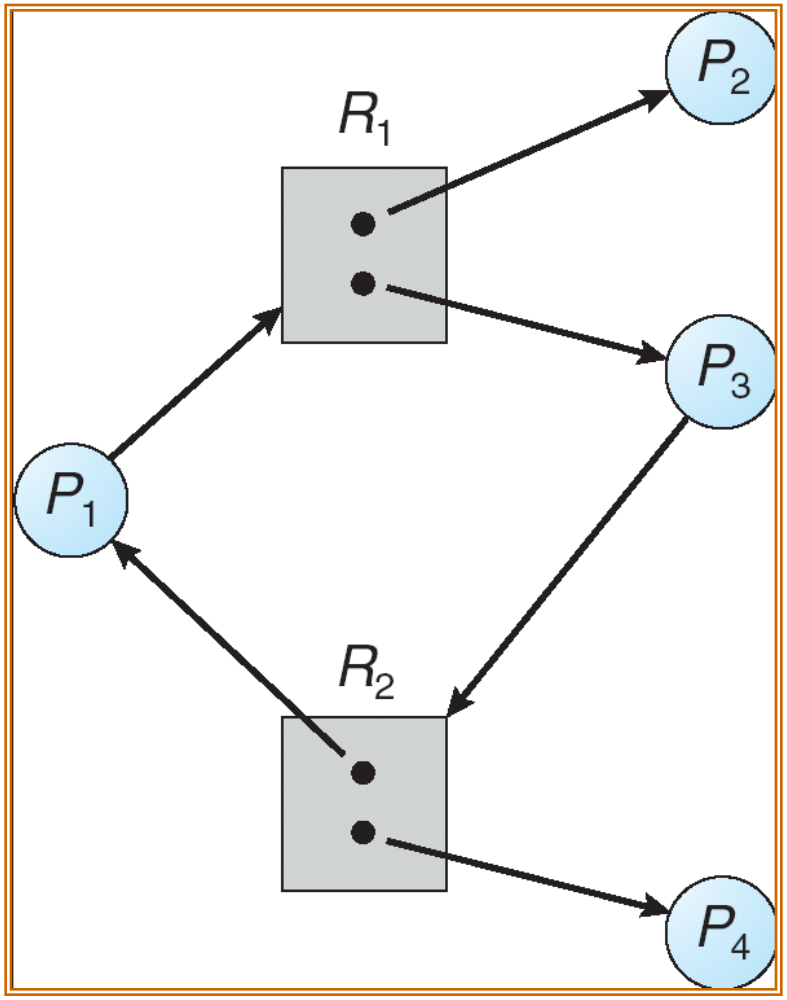
[1] If each resource type has exactly one instance, then a cycle implies that a deadlock has occurred.

[2] If each resource type has several instances, then a cycle does not necessarily imply a deadlock. The cycle is just a necessary but not a sufficient condition

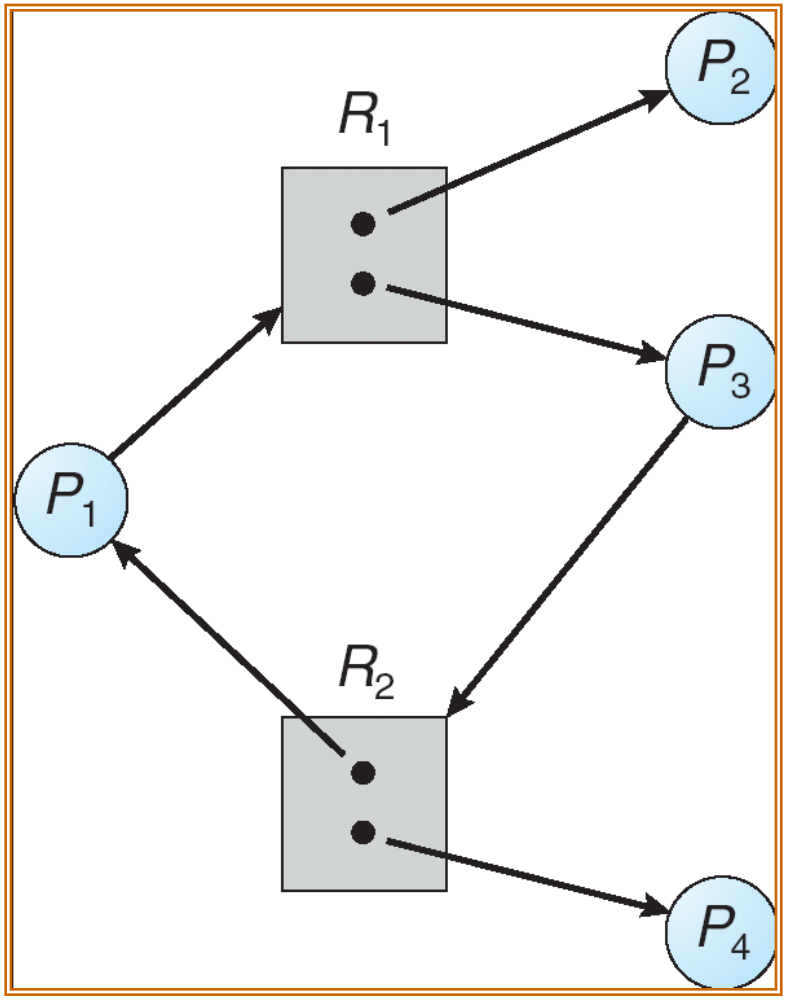


3 processes are deadlocked!

Resource Allocation Graph With A Cycle But No Deadlock



Resource Allocation Graph With A Cycle But No Deadlock



No deadlock! P4 breaks the cycle

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.

Handling Deadlocks

- Prevention – make it impossible for deadlock to happen
- Avoidance – control allocation of resources
- Detection and Recovery – look for a cycle in dependencies
- Ignore it – Ignore the problem and pretend that deadlocks never occur in the system; used by most operating systems, including UNIX.
  - It is up to the app. Developer to write programs that handle deadlocks

Deadlock Prevention

Break one of the four conditions to prevent deadlock

- Mutual exclusion
  - Make resources sharable (not generally practical)
- Hold and wait
  - Process cannot hold one resource when requesting another
  - Process requests, releases all needed resources at once
  - Low resource utilization; possible starvation
- Preemption
  - OS can preempt resource (costly)
  - Not practical for many resources (printers, tape drives)!
- Circular wait
  - Impose an ordering (numbering) on the resources and request them in order (popular implementation technique)

Deadlock Avoidance

- Avoidance
  - Provide information in advance about what resources will be needed by processes to guarantee that deadlock will not happen
  - System only grants resource requests if it knows that the process can obtain all resources it needs in future requests
  - Avoids circularities (wait dependencies)
- Tough
  - Hard to determine all resources needed in advance
  - Good theoretical problem, not as practical to use

Deadlock Avoidance

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

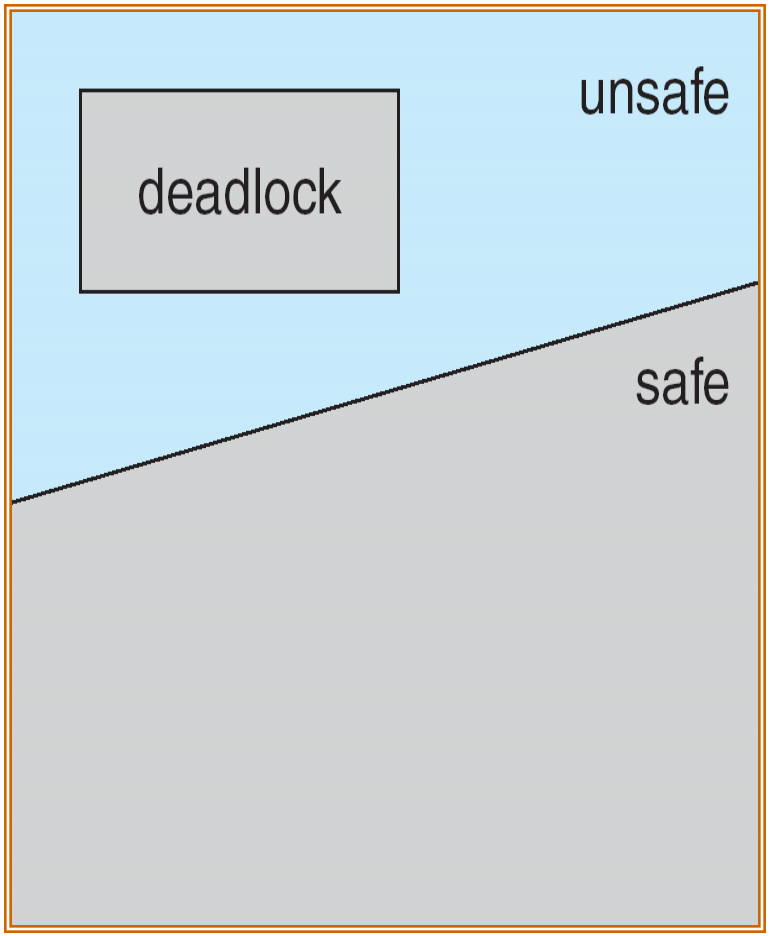
- Requires that each process declare the *maximum number* of resources of each type that it may need.
- The deadlock-avoidance algorithm dynamically examines the resource-allocation state to ensure that there can never be a circular-wait condition.
- Resource-allocation *state* is defined by the number 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 <P_1, P_2, …, P_n> 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.

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.



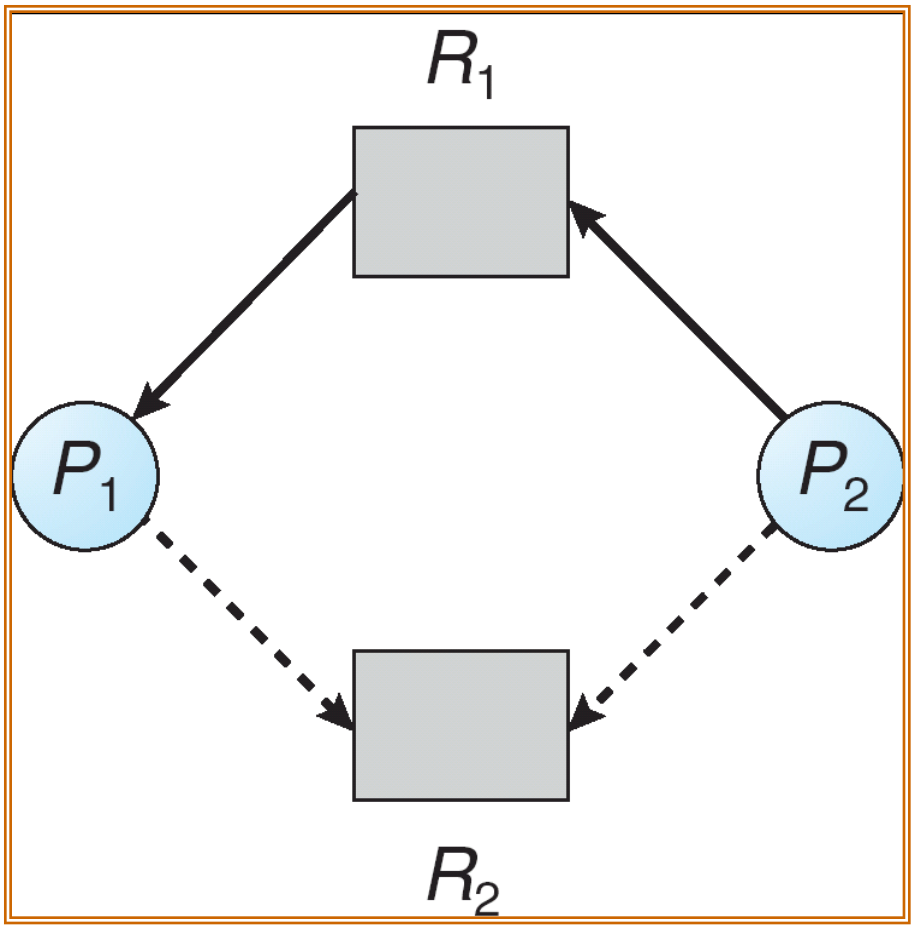
Resource-Allocation Graph Algorithm

Only works for resource types each with one instance!

- ***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.
  - Before a process executes, all its claim edges must already appear in the graph!

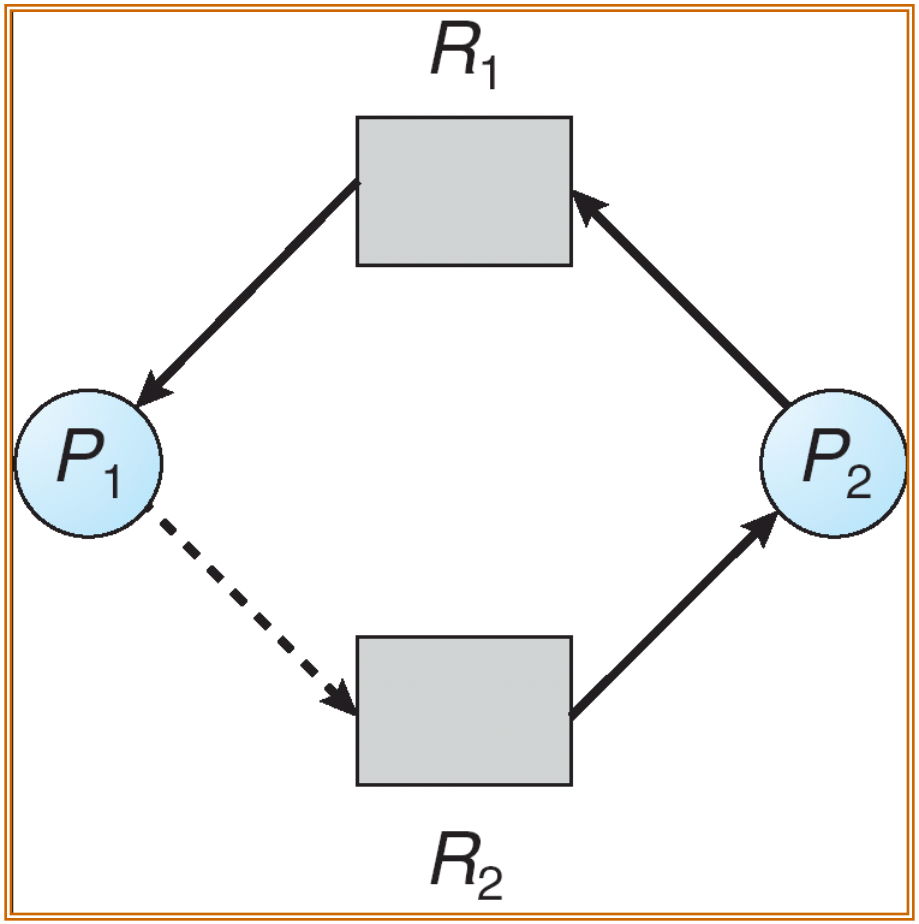
Resource-Allocation Graph For Deadlock Avoidance

Can the request P2-->R2 be granted?



Unsafe State In Resource-Allocation Graph

NO. converting to assignment edge forms a cycle, unsafe state!



Banker's Algorithm

- **The resource-allocation-graph algorithm is not applicable to a resource allocation with multiple instances of each resource type**

- 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 x m* matrix. If *Max* [*i,j*] = *k*, then process P_i may request at most *k* instances of resource type R_j.
- ***Allocation:*** *n* x *m* matrix. If Allocation[*i,j*] = *k* then P_i is currently allocated *k* instances of R_j.
- ***Need:*** *n* x *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*]

Banker's Algorithm

Goal: find a safe sequence of processes!

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

   Work = Available

   Finish [*i*] = *false* for *i* = 1,2, …, *n.*

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 a safe state.

Require $O(m \times n^2)$ operations!!!

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:

	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	

- 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.

Resource-Request Algorithm for Process P_i

-- To determine if a request should be granted or not?

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 Pi.*
   - *If unsafe $\Rightarrow$ Pi must wait, and the old resource-allocation state is restored*

Example P_1 Request (1,0,2)

	Allocation	*Need*	*Available*
	A B C	*A B C*	*A B C*
P_0	0 1 0	7 4 3	3 3 2
P_1	2 0 0	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	

- Can this request be satisfied?

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

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

	Allocation	*Need*	*Available*
	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 <P1, P3, P4, P0, P2> satisfies safety requirement.
- **Question:** Can request for (3,3,0) by P4 be granted?
- **Question:** Can request for (0,2,0) by P0 be granted?

Detection and Recovery

- Detection and recovery
  - If we don't have deadlock prevention or avoidance, then deadlock may occur
  - In this case, we need to detect deadlock and recover from it
- To do this, we need two algorithms
  - One to determine whether a deadlock has occurred
  - Another to recover from the deadlock
- Possible, but expensive (time consuming)
  - Implemented in VMS
  - Run detection algorithm when resource request times out

Deadlock Detection

- Detection
  - Traverse the resource graph looking for cycles
  - If a cycle is found, preempt resource (force a process to release)
- Expensive
  - Many processes and resources to traverse
- Only invoke detection algorithm depending on
  - How often or likely deadlock is
  - How many processes are likely to be affected when it occurs

Dead Recovery

- Once a deadlock is detected, we have two options…
- 1. Abort processes
  - Abort all deadlocked processes
    - Processes need start over again
  - Abort one process at a time until cycle is eliminated
    - System needs to rerun detection after each abort
- 2. Preempt resources (force their release)
  - Need to select process and resource to preempt
  - Need to rollback process to previous state
  - Need to prevent starvation

Summary

- Deadlock occurs when processes are waiting on each other and cannot make progress
  - Cycles in Resource Allocation Graph (RAG)
- Deadlock requires four conditions
  - Mutual exclusion, hold and wait, no resource preemption, circular wait
- Four approaches to dealing with deadlock:
  - Ignore it – Living life on the edge
  - Prevention – Make one of the four conditions impossible
  - Avoidance – Banker's Algorithm (control allocation)
  - Detection and Recovery – Look for a cycle, preempt or abort