Deadlocks

Dept. of Computer Science

Hanyang University

The Deadlock Problem

- 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
P (A);	*P(B)*
P (B);	*P(A)*

Deadlock Characterization

Deadlock can arise if 4 conditions hold simultaneously (necessary condition)

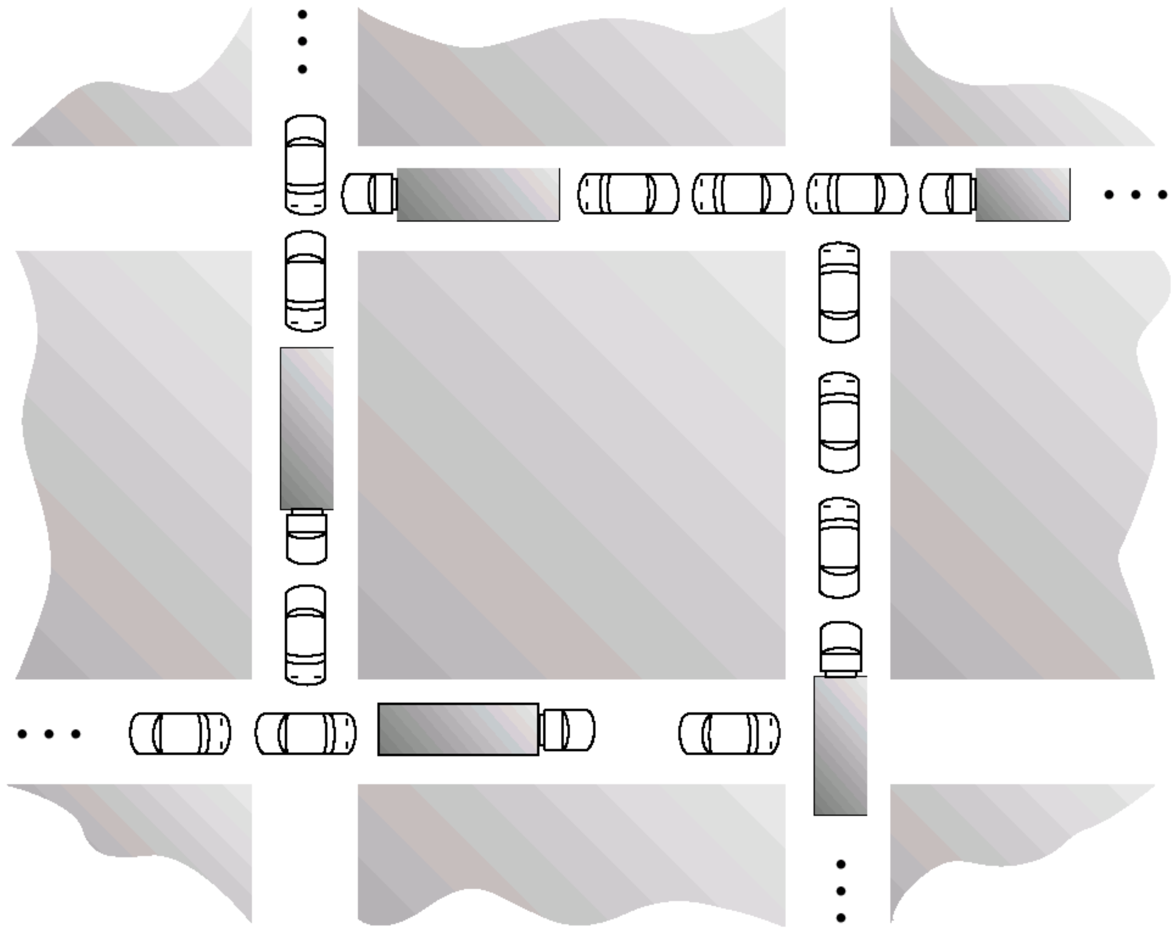
- **Mutual exclusion:** only one process at a time can use a resource
- **No preemption:** a resource can be released only voluntarily by the process holding it, after that process has completed its task
- **Hold and wait:** a process holding at least one resource is waiting to acquire additional resources held by other processes
- **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 $P_2, \ldots,$

  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.

Traffic Deadlock



Real World Traffic Deadlock

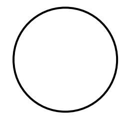
Resource-Allocation Graph

A set of vertices V and a set of edges E

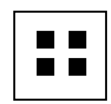
- V is partitioned into two types:
  - $P = \{P_1, P_2, \ldots, P_n\}$, the set consisting of all the processes in the system
  - $R = \{R_1, R_2, \ldots, R_m\}$, the set consisting of all resource types in the system
- request edge – directed edge $P_1 \rightarrow R_j$
- assignment edge – directed edge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

Resource-Allocation Graph

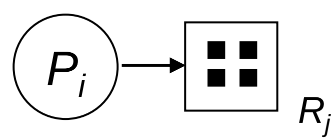
- Process



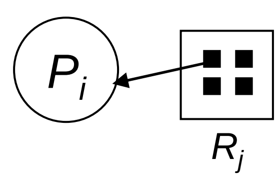
- Resource Type with 4 instances



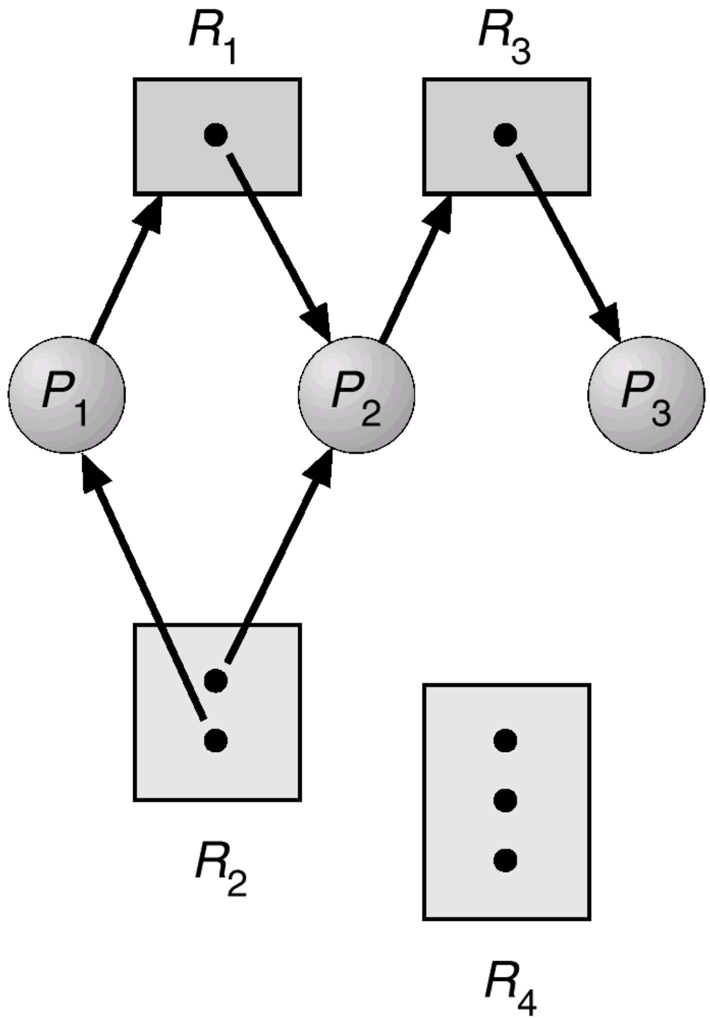
- P_i requests an instance of R_j



- P_i is holding an instance of R_j



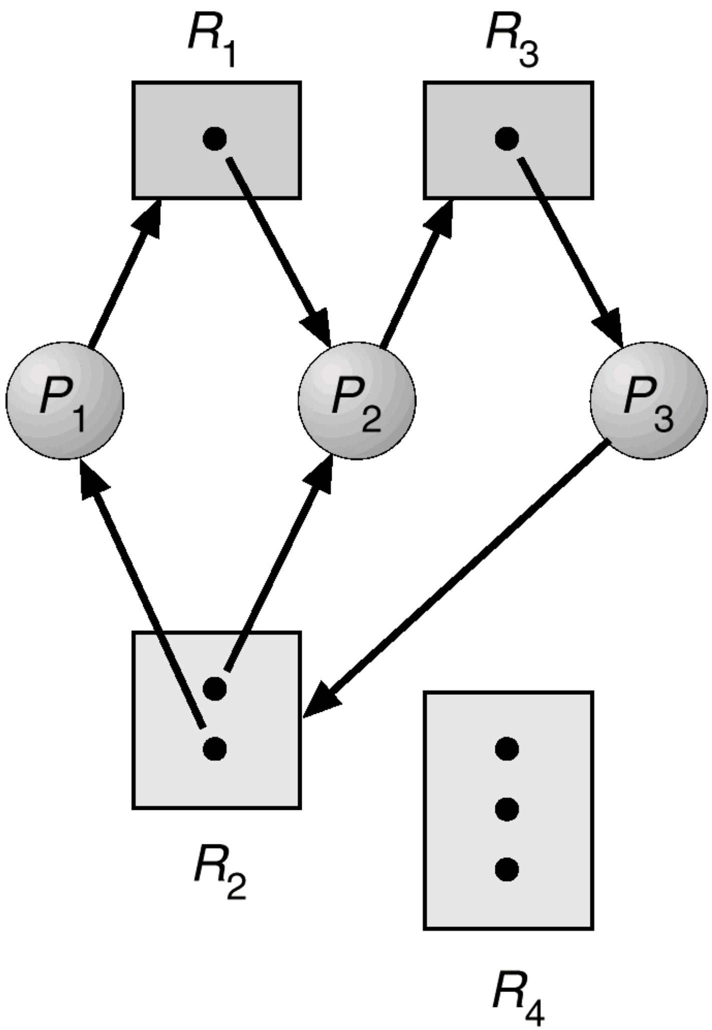
Example of a Resource Allocation Graph



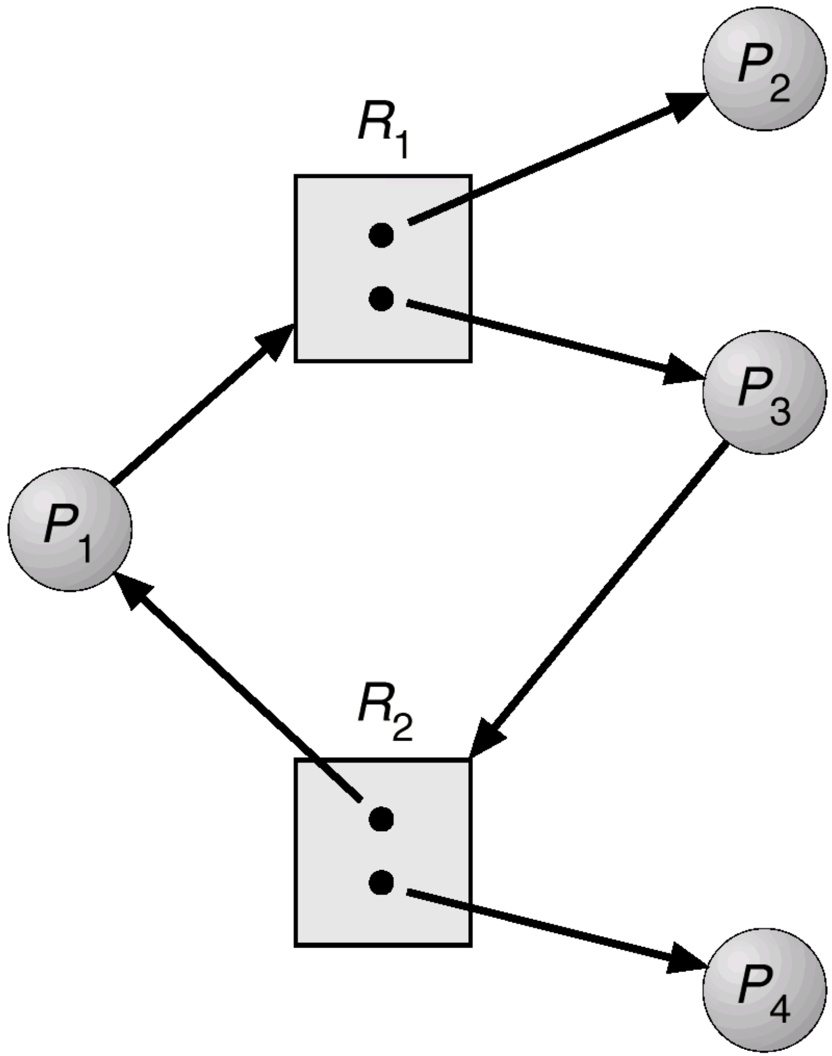
Basic Facts

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

Resource Allocation Graph with a Deadlock



Resource Allocation Graph with a Cycle But No Deadlock



Methods for Handling Deadlocks

- Ensure that the system will *never* enter a deadlock state

  (prevent, avoid)

- Allow the system to enter a deadlock state and then recover

  (after detection)

- Ignore the problem and pretend that deadlocks never occur in the system; used by most operating systems, including UNIX

Deadlock Prevention

Restrain the ways request can be made so that any one of necessary conditions does not hold.

- Mutual Exclusion
  - Not required for sharable resources
  - Must hold for non-sharable resources (enforcement impossible for intrinsically 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, or allow process to request resources only when the process has none (all or nothing)
  - Low resource utilization; starvation possible

Deadlock Prevention (Cont.)

- No Preemption
  - If a process that is holding some resources 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.

** Low resource utilizations and reduced system throughput

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

- 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 $\mathrm{P_i}$ 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

- System is in <u>safe</u> state if there exists a safe sequence of all processes

Basic Facts

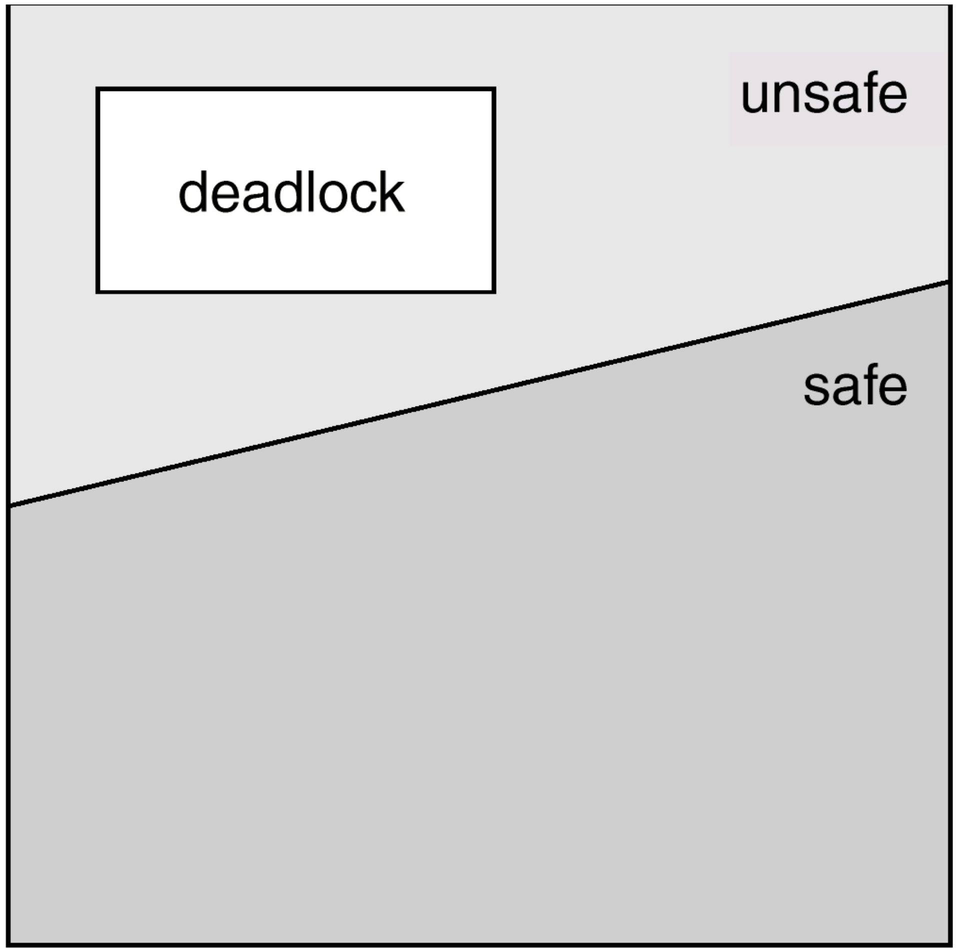
- If a system is in safe state ⇒ no deadlocks
- If a system is in <u>unsafe</u> state ⇒ <u>possibility of deadlock</u>
  - When all processes request maximum amount of resources of all types

- Avoidance ⇒ <u>ensure</u> that a system will never enter an unsafe state

“grant the request if it results in a safe state,

do not grant it otherwise”

Safe, unsafe , deadlock state spaces



Avoidance algorithms

- Single instance of a resource type
  - Use a resource-allocation graph

- Multiple instances of a resource type
  - Use the banker's algorithm

Case A: One instance per resource types : Resource Allocation Graph Algorithm

- ***claim edge***

  $P_i \rightarrow R_j$ indicates that process P_i may request resource R_j (dashed line)

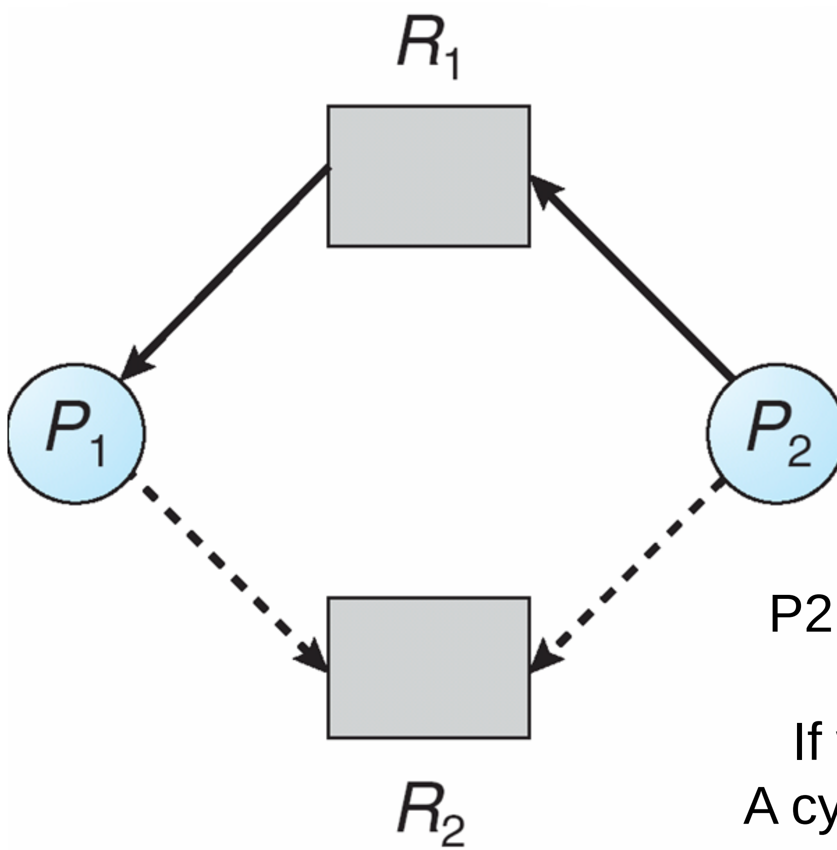
- ***request edge***

  Claim edge converts to request edge when a process requests a resource

- ***assignment edge***
  - Request edge converts to assignment edge when the resource is allocated to the process
  - When a resource is released by a process, assignment edge reconverts to a claim edge

- Algorithm: (Resources must be claimed *a priori* in the system)
  - 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

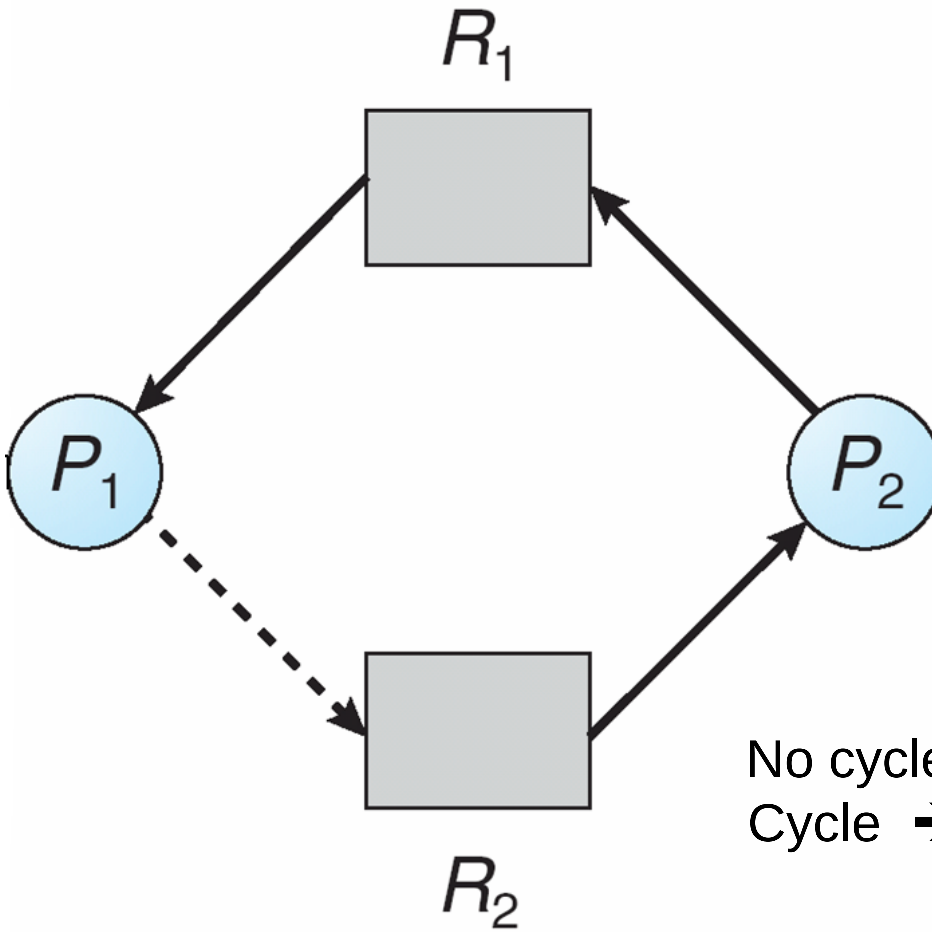
Resource-Allocation Graph for Deadlock Avoidance



P2 requests R2

If we grant it,
A cycle is formed
(unsafe)

Unsafe State in a Resource-Allocation Graph



No cycle ➔ safe ➔ grant
Cycle ➔ unsafe ➔ deny

Case B: Multiple instances per resource types : Banker's Algorithm

- 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

vector

- **Available**:

  Available[j] = k : k instances of resource type R_j are available

n x m **matrix**

- **Max**: *Max* [i,j] = k : P_i may request at most k instances of R_j.
- **Allocation**:

  Allocation[i,j] = k : P_i is currently allocated k instances of R_j.
- **Need**: If *Need*[i,j] = k : P_i may need k more instances of R_j.

  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* = 1,2, …, *n.*

2. Find an *i* such that both:

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

   (b) $Need_i \le Work$

   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

$Request_i$ = 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), B (5), and C (7) instances. 10 5 7
- Snapshot at time T_0:

	Allocation	*Max*	*Available*	*Need*
	A B C	*A B C*	*A B C*	*A B C*
P_0	0 1 0	7 5 3	3 3 2	7 4 3
P_1	2 0 0	3 2 2		1 2 2
P_2	3 0 2	9 0 2		6 0 0
P_3	2 1 1	2 2 2		0 1 1
P_4	0 0 2	4 3 3		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

Example (Cont.): P_1 request (1,0,2)

- Check that Request ≤ Available (that is, $(1,0,2) \le (3,3,2) \Rightarrow$ *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 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 $<P_1, P_3, P_4, P_0, P_2>$ 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?

Deadlock Detection

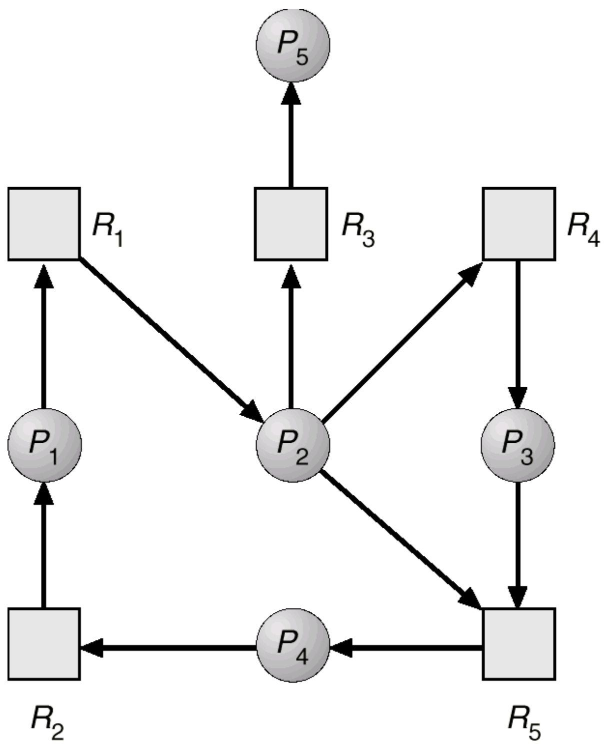
- Allow system to enter deadlock state
- Detection algorithm
- Recovery scheme

- 2 cases
  - A: single instance per resource type: cycle → deadlock
  - B: multiple instance per resource type: ?

Single Instance of Each Resource Type

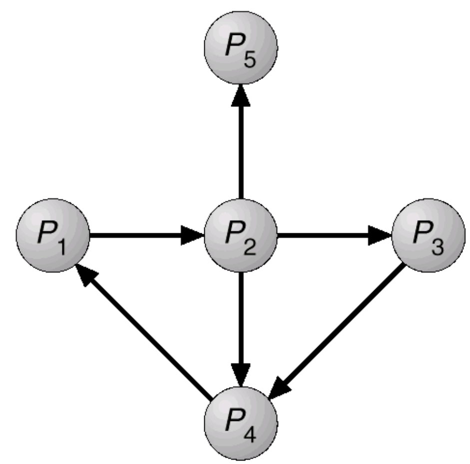
- Maintain ***wait-for*** graph
  - Nodes are processes
  - $P_k \rightarrow P_j$ if P_k is waiting for P_j
- Invoke an algorithm that searches for a cycle.
- An algorithm to detect a **cycle** in a graph requires an O(n^2) operations, where n is the number of vertices in the graph

Resource-Allocation Graph and Wait-for Graph



(a)

Resource-Allocation Graph



(b)

Corresponding wait-for graph

Several Instances of a Resource Type

- Use Deadlock Detection Algorithm
- Data structures
  - *Available*: vector of length m indicates the number of available resources of each type
  - *Allocation*: n x m matrix defines the number of resources of each type currently allocated to each process
  - *Request*: 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*

   For i = 1,2, …, n

   Finish[i] = false, if $Allocation_j$ *is not 0*

   Finish[i] = true, otherwise

2. Find an index i such that both:

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

   (b) $Request_i \leq Work$

   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] = *false* for some i, $1 \leq i \leq n$, then the system is in a deadlock state. Moreover, if *Finish[I] = false*, then process P_i is deadlocked.

Example of Detection Algorithm

- Five processes P_0 through P_4; three resource types A (7 instances), *B* (2 instances), and *C* (6 instances)
- Snapshot at time T_0:

	Allocation	*Request*	*Available*
	A B C	*A B C*	*A B C*
P_0	0 1 0	0 0 0	0 0 0
P_1	2 0 0	2 0 2	
P_2	3 0 3	0 0 0	
P_3	2 1 1	1 0 0	
P_4	0 0 2	0 0 2	

- Sequence <P_0, P_2, P_3, P_1, P_4> will result in *Finish*[*i*] = true for all *i*

Detection Algorithm (Cont.)

Algorithm requires O($m \times n^2$) operations to detect whether the system is in deadlocked state.

m: resource types

n: processes

If m=n, algorithm requires O(n^3) operations

Problem: How frequently the detection algorithm will be invoked?

every request?(large overhead)

every request not allocated immediately?

periodically? (if deadlock probability is low)

Recovery from Deadlock: Process Termination

- Abort all deadlocked processes
- Abort one process at a time until the deadlock cycle is eliminated
- In which order should we choose to abort?
  - Priority of the process
  - How long process has computed, and how much longer to completion
  - Resources the process has used
  - Resources process needs to complete
  - How many processes will need to be terminated
  - Is process interactive or batch?

Recovery from Deadlock: Resource Preemption

- Selecting a victim : minimize cost
- Rollback
  - return to some safe state
  - restart process from that state
- Starvation
  - same process picked as victim repeatedly
  - include number of rollback in cost factor

Avoidance v.s. Detection

Differences of two algorithms

- Avoidance: we assumed that processes behave the worst
  - Worst case assumption: every process claims maximum resources all at the same time
  - Assign resources only if there is a safe sequence assuming the worst case!
  - May waste resources
- Detection: we assume every process is dormant
  - Best case assumption: no process will request resources furthermore
  - Detects deadlock based on the current state (best case assumption)