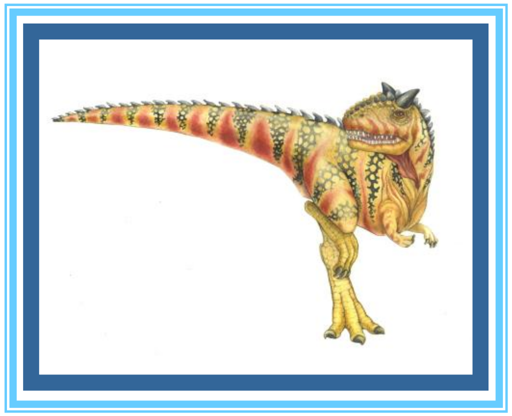
Chapter 7: Deadlocks



Chapter 7: Deadlocks

- System Model
- Deadlock Characterization
- Methods for Handling Deadlocks
- Deadlock Prevention
- Deadlock Avoidance
- Deadlock Detection
- Recovery from Deadlock

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 consists of resources
- Resource types $R_1, R_2, \ldots, R_m$ *(Physical or logical)

  CPU cycles, memory space, I/O devices
- Each resource type R_i has W_i instances.
- Each process utilizes a resource as follows:
  - **request** * (system call such as open & close file, allocate memory or wait & signal semaphore)
  - **use**
  - **release** *(Same as above)

Deadlock Characterization

Deadlock can arise if four conditions hold simultaneously.

★ (________ condition)

- **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, …, 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.

⋆(Circular wait is for just waiting without ________)

Deadlock with Mutex Locks

- Deadlocks can occur via system calls, locking, etc
- See example box in text page 313 for mutex deadlock

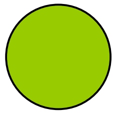
Resource-Allocation Graph

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

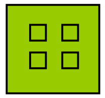
- V is partitioned into two types:
  - $P = \{P_1, P_2, \ldots, P_n\}$, the set consisting of all the processes in the system
  - $R = \{R_1, R_2, \ldots, R_m\}$, the set consisting of all resource types in the system
- **request edge** – directed edge $P_i \rightarrow R_j$
- **assignment edge** – directed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

Resource-Allocation Graph (Cont.)

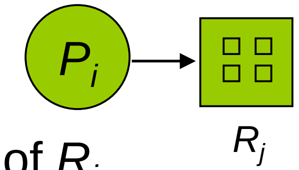
- Process



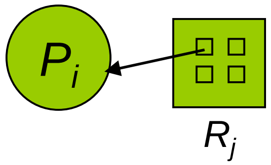
- Resource Type with 4 instances



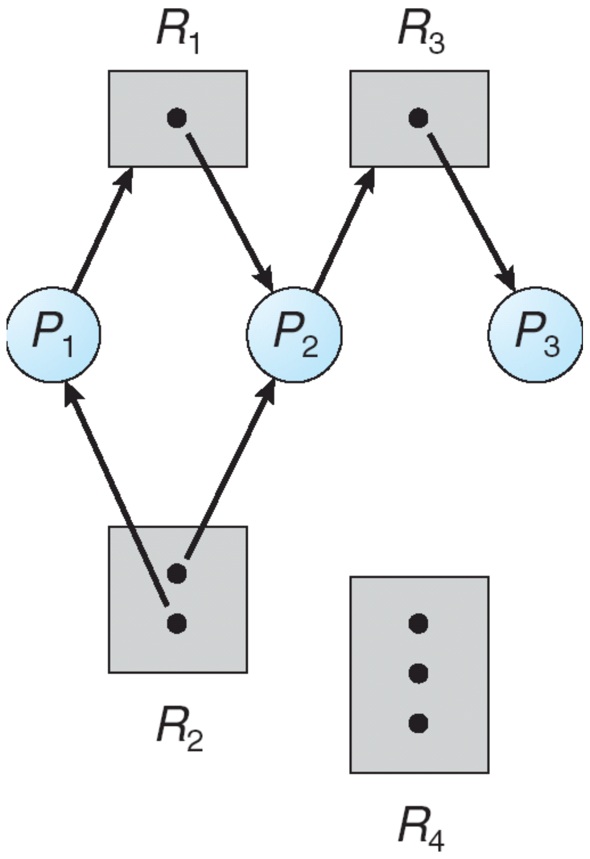
- P_i requests instance of R_j



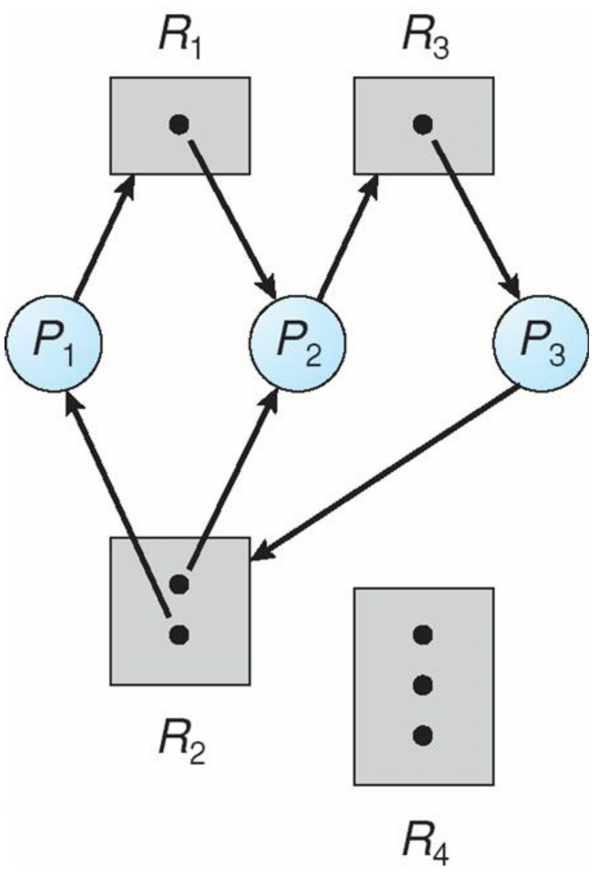
- P_i is holding an instance of R_j



Example of a Resource Allocation Graph



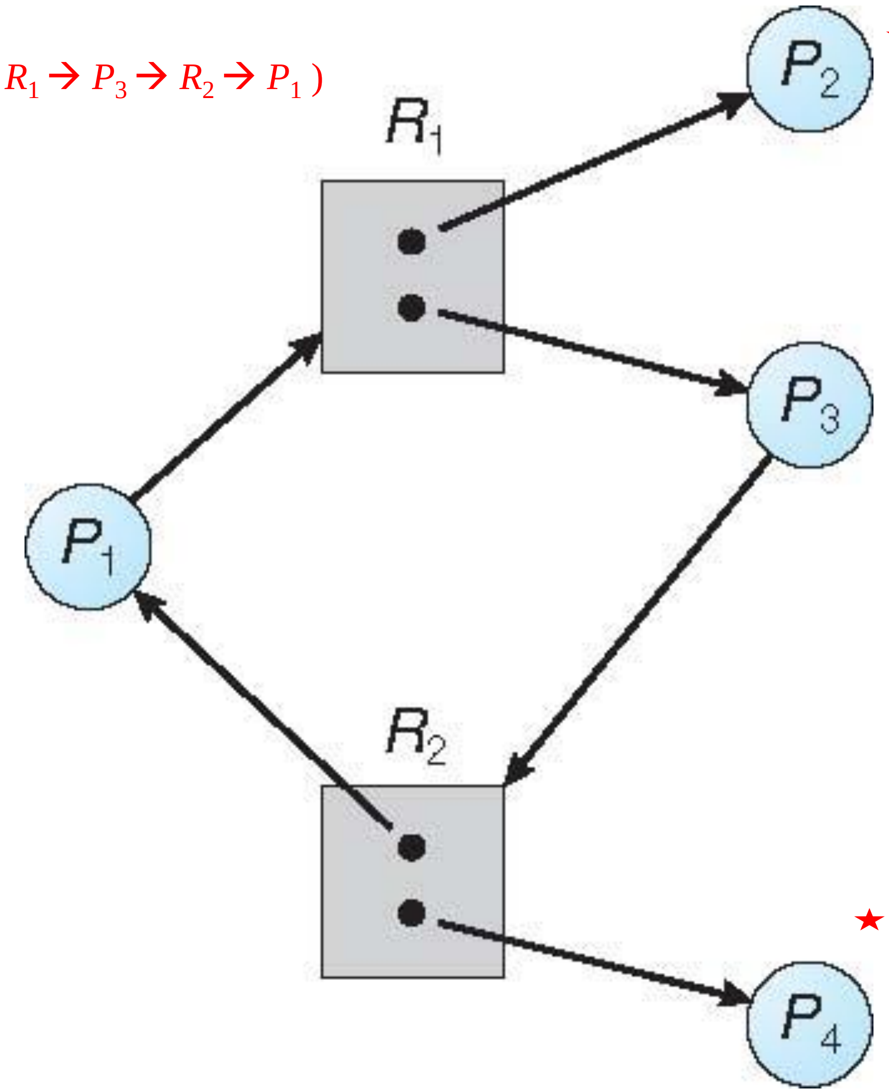
Resource Allocation Graph With A Deadlock



★(2 cycles: $P_1 \rightarrow R_1 \rightarrow P_2 \rightarrow R_3 \rightarrow P_3 \rightarrow R_2 \rightarrow P_1$; $P_2 \rightarrow R_3 \rightarrow P_3 \rightarrow R_2 \rightarrow P_2$)

Graph With A Cycle But No Deadlock

★ (cycle: $P_1 \rightarrow R_1 \rightarrow P_3 \rightarrow R_2 \rightarrow P_1$)



★ (If P_2 releases R_1, then it can be allocated to P_1 which eliminates he cycle. When P_1 releases R_2, __ can proceed.)

★ (If P_4 releases R_2, then it can be allocated to P_3 which eliminates the cycle. When P_3 releases R_1, __ can proceed.)

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

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

  * (by prevention or _________)

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

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

  * (Prevention: ensure at least one __________ condition cannot hold by constraining how request for resources can be made)

  * (Avoidance: OS is given in advance the information concerning which _________ a process will request and use during its lifetime)

Deadlock Prevention

Restrain the ways request can be made

- **Mutual Exclusion** – not required for sharable resources; must hold for nonsharable 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
  - Low resource utilization; starvation possible

* (Ex) Copy data from tape to disk, sort, and then send to printer

* (Solution 1) Acquire all three devices first. Use and then release (the printer is then idle most of time until the last stage)

* (Solution 2) Acquire tape & disk, use and then release. Acquire disk & printer, use and then release.

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

* (Ex) P_0 needs one more resource R_A to start, but they are now all allocated to P_1 and P_2.

- **Circular Wait** – impose a total ordering of all resource types, and require that each process requests resources in an increasing order of enumeration

* (Solution 1) Check if an R_A is allocated to a process waiting for other resource. If P_1 is waiting and P_2 is running, preempt R_A from __ , and allocate it to P_0. Start P_0.
(Solution 2) If P_1 and P_2 are running, P_0 needs to ____ . During waiting, the resources allocated to P_0 may be preempted if requested.

Deadlock Example

```
/* thread one runs in this function */
void *do work one(void *param)
{
   pthread mutex lock(&first mutex);*  1
   pthread mutex lock(&second mutex);* 3
   /** * 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);* 2
   pthread mutex lock(&first mutex); *4
   /** * Do some work */
   pthread mutex unlock(&first mutex);
   pthread mutex unlock(&second mutex);
   pthread exit(0);
}
```

Deadlock Example with Lock Ordering

```
void transaction(Account from, Account to, double amount)
{
   mutex lock1, lock2;
   lock1 = get lock(from);
   lock2 = get lock(to);
   acquire(lock1);
      acquire(lock2);
         withdraw(from, amount);
         deposit(to, amount);
      release(lock2);
   release(lock1);
}
```

* T_0:

```
. . .
acquire(A);
 acquire(B);
  (*deadlock*)
. . .
```

T_1:

```
. . .
acquire(B);
 acquire(A);
  (* deadlock*)
. . .
```

* (Deadlock if T_0 executes transaction (A, B, 50) and T_1 does transaction (B, A, 100) simultaneously)

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

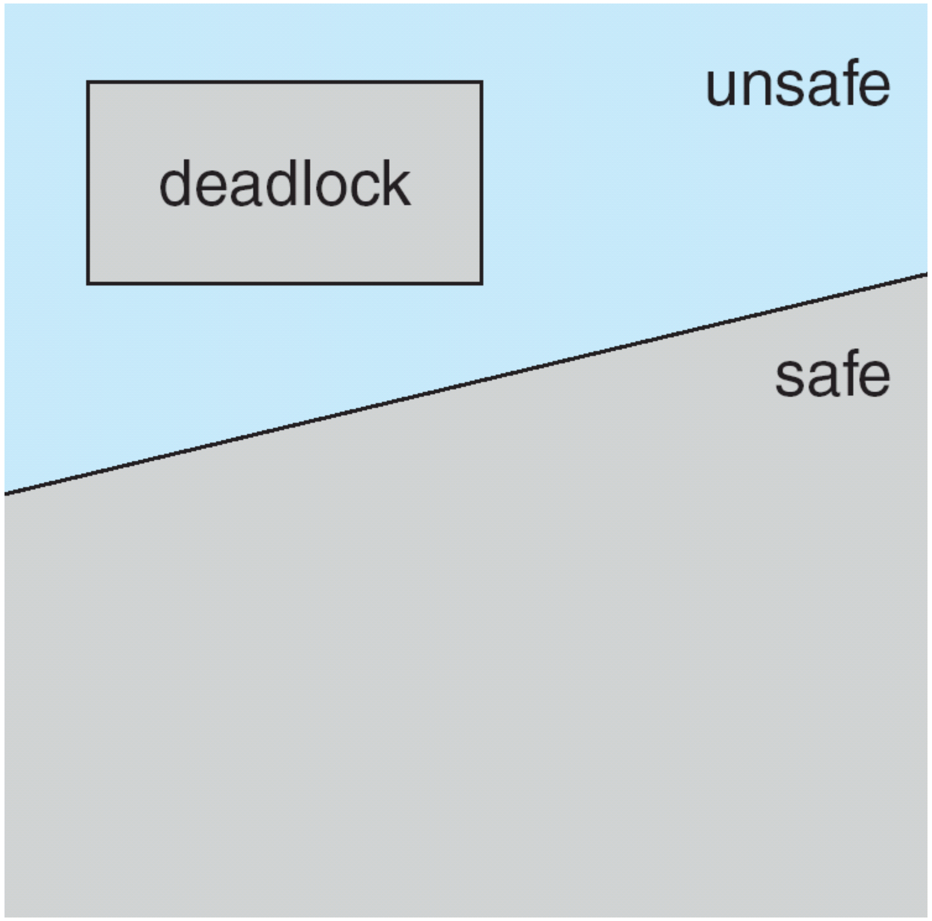
- System is in **safe state** if there exists a sequence $<P_1, P_2, \ldots, P_n>$ of ALL the processes in the systems such that for each P_i, the resources that P_i can still request can be satisfied by currently available resources + resources held by all the P_j, with $j < i$

- 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

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



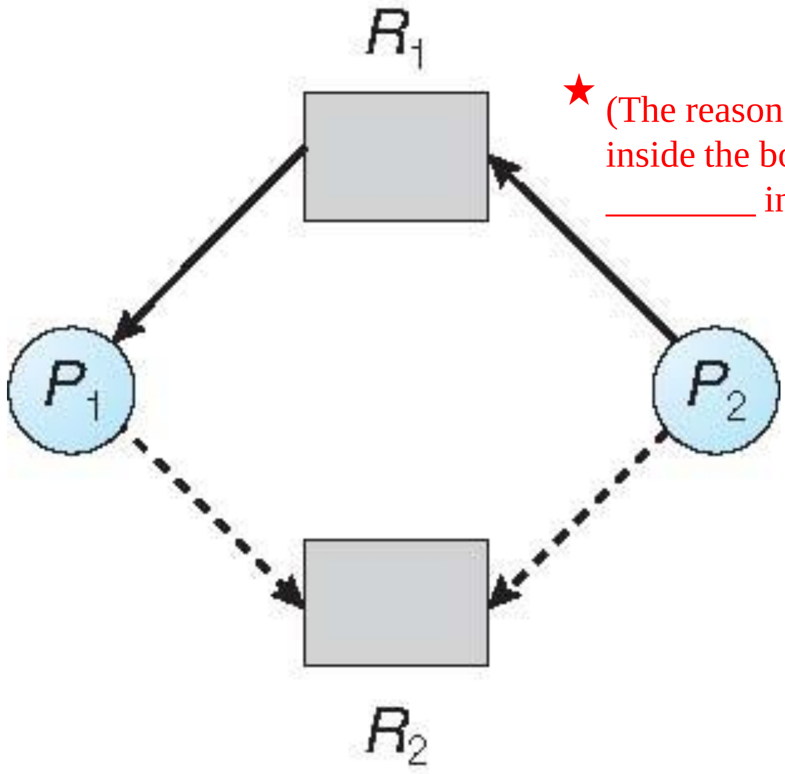
Avoidance algorithms

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

Resource-Allocation Graph Scheme

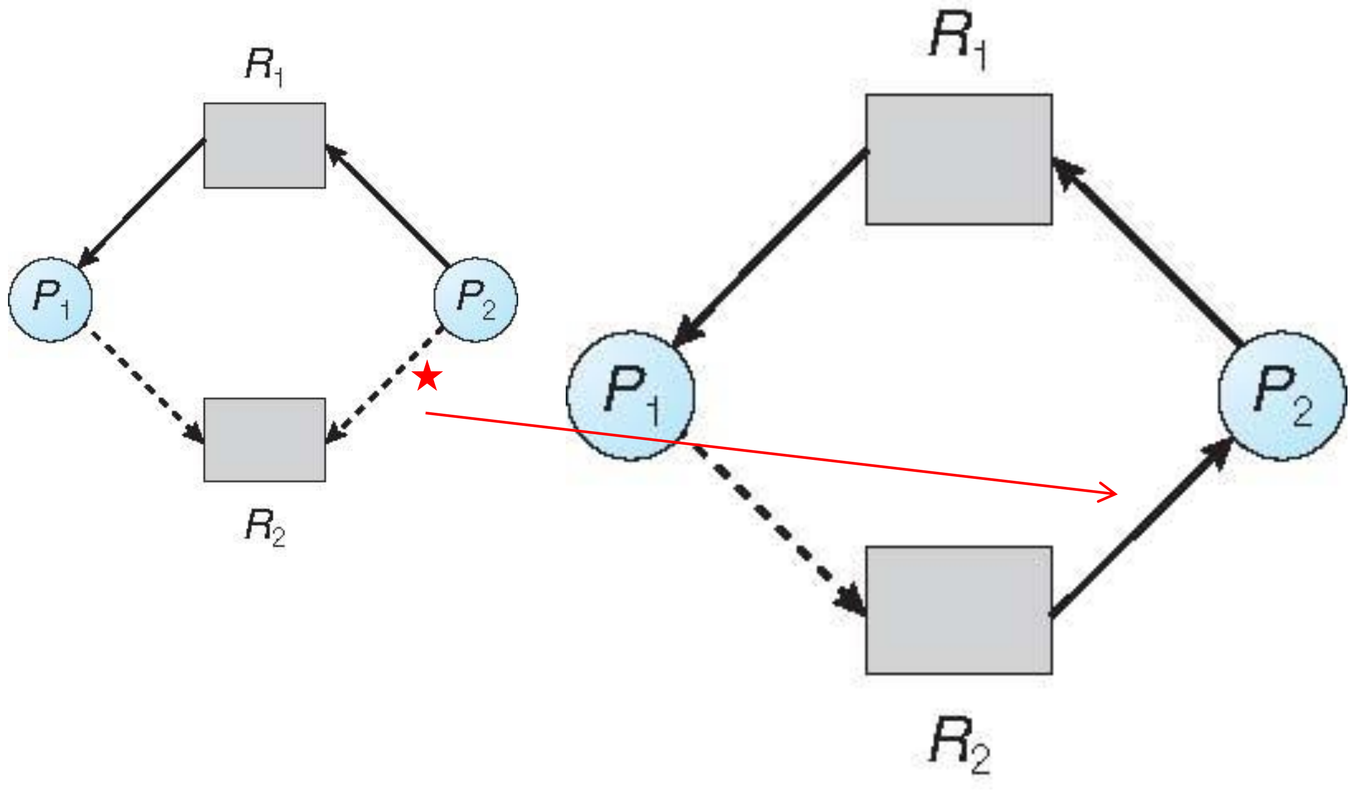
- **Claim edge** $P_i \rightarrow R_j$ indicated that process P_j may request resource R_j; represented by a dashed line
- Claim edge converts to request edge when a process requests a resource
- Request edge converted to an assignment edge when the resource is allocated to the process
- When a resource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

Resource-Allocation Graph



★ (The reason why no instance is drawn inside the box is that there is only one ________ in it)

Unsafe State In Resource-Allocation Graph



★ (P_2 requests R_2. If R_2 is allocated to P_2 , it will create a cycle. So, $P_i \rightarrow R_j$ (request) is changed to $R_j \rightarrow P_i$(allocation) in the resource allocation graph only if no _____ occurs.)

★ (It takes n^2 times to detect a _____ when n is the number of processes.)

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

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

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

   Work = Available

   ***Finish* [*i*] = *false* for *i* = 0, 1, …, *n*- 1**

2. Find an ***i*** such that both:

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

   (b) ***Need*$_i$ ≤ *Work*** *(If true, we can allocate what P_i needs.)

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

3. ***Work* = *Work* + *Allocation*$_i$** *(Work now contains the item currently available and those already allocated that will be eventually released)
   Finish*[*i*] = *true
   go to step 2

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

   * (O(mn^2))

Resource-Request Algorithm for Process P_i

Request = request vector for process P_i. If $\boldsymbol{Request_i\,[j] = k}$ then process P_i wants k instances of resource type R_j

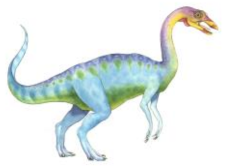
1. If $Request_i \leq Need_i$ go to step 2. Otherwise, raise error condition, since process has exceeded its maximum claim
2. If $Request_i \leq Available$, go to step 3. Otherwise P_i must wait, since resources are not available
3. Pretend to allocate requested resources to P_i by modifying the state 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 * (Checked using ______ algorithm)
- 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:

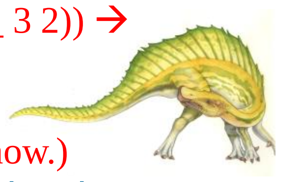
  A (10 instances), *B* (5instances), and *C* (7 instances)

Snapshot at time T_0:

(Number of instances – Allocation)

	Allocation	*Max*	★*Available*	*Need*★ (Max – Allocation)
	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
	* (7 2 5)			

★(Work= (3 _ 2)) → (P_1(1 2 2) < Work, so assign, finish, return resource) → (Work= (_ 3 2)) →
(P_3(0 1 1) < Work) → (Work= (7 4 3)) → (P_4(4 3 1) < Work) →
(Work = (7 4 5)) → (P_2(6 0 0) < Work) → (Work = (10 4 7)) →
(Work = (10 5 7)) after P_0 is finished) (Hence it is in _______ state now.)



Example (Cont.)

- 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

Example: P_1 Request (1,0,2)

- Check that Request ≤ Need (that is, (1,0,2) ≤ (1,2,2)) ⇒ true)
- Check that Request ≤ Available (that is, (1,0,2) ≤ (3,3,2)) ⇒ true)

	Allocation		*Need(n)*		*Available(n)*	*Need(old)*	*Available(n)*
	A B C		*A B C*		*A B C*	*A B C*	*A B C*
P_0	0 1 0		7 4 3		2 3 0	7 4 3	3 3 2
P_1	3 0 2*	((2,0,0)+ (1,0,2))	0 2 0*	((1,2,2)– (1,0.2))		1 2 2	
P_2	3 0 2		6 0 0			6 0 0	
P_3	2 1 1		0 1 1			0 1 1	
P_4	0 0 2		4 3 1			4 3 1	

- Executing safety algorithm shows that sequence < $\boldsymbol{P_1, P_3, P_4, P_0, P_2}$> satisfies safety requirement * (Work= (2 3 0)) → (P_1 finish; Work = (5 3 2)) → (P_3 ; Work = (7 4 3)) → (P_4 ; Work = (7 4 5)) → (P_0 ; Work = (7 5 5)) → (P_2 ; Work = (10 5 7))
- Can request for (3,3,0) by $\boldsymbol{P_4}$ be granted?

  ★ ((3 3 0) < (4 3 1) ok. But no grant since (3 3 0) < (2 3 0))
- Can request for (0,2,0) by $\boldsymbol{P_0}$ be granted?

  ★ ((0 2 0) < (7 4 3) ok. (0 2 0) < (2 3 0), and so far fine) →

(P_0 Allocation = (0 3 0), Need (7 2 3); Work = (2 1 0) → (No sequence is found, and it is unsafe. Answer is NO)

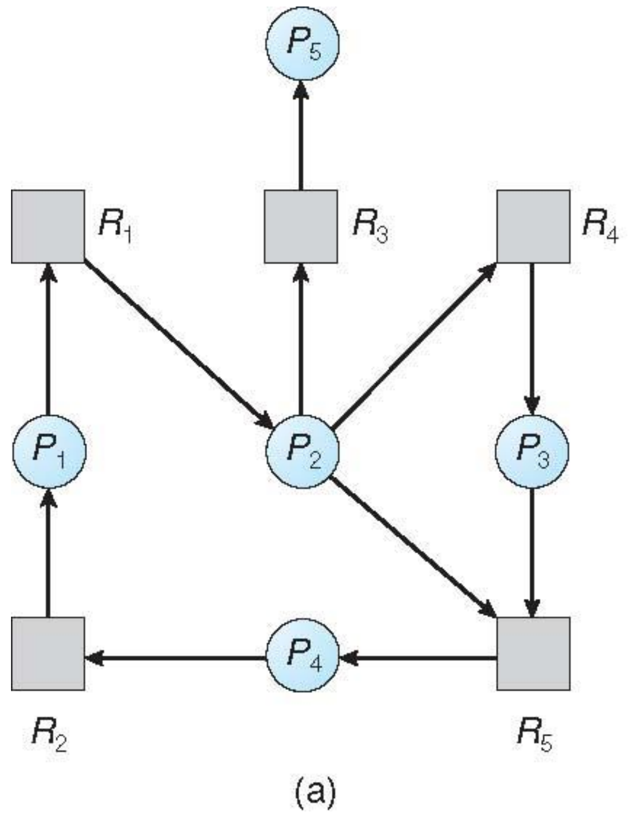
Deadlock Detection

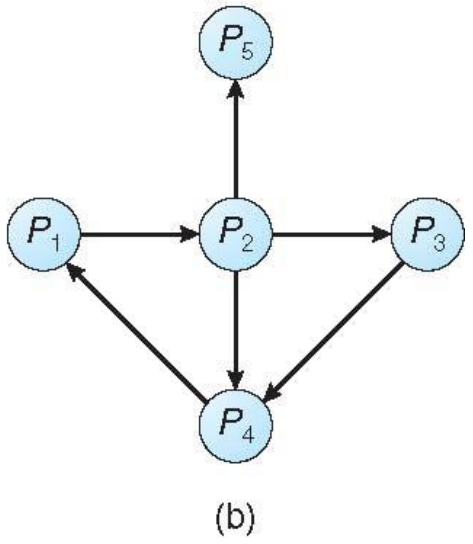
- Allow system to enter deadlock state
- Detection algorithm * (Overhead: maintaining necessary info & executing detection algo)
- Recovery scheme

Single Instance of Each Resource Type

- Maintain **wait-for** graph
  - Nodes are processes
  - $P_i \rightarrow P_j$ if P_i is waiting for P_j
- Periodically invoke an algorithm that searches for a cycle in the graph. If there is a cycle, there exists a deadlock
- An algorithm to detect a cycle in a graph requires an order of n^2 operations, where n is the number of vertices in the graph

Resource-Allocation Graph and Wait-for Graph





Resource-Allocation Graph

Corresponding wait-for graph

- $((P_1, P_2, P_3, P_4), (P_1, P_2, P_4))$
- (Deadlock exists iff the wait-for graph contains a cycle)

Several Instances of a Resource Type

- **Available***:* A vector of length ***m*** indicates the number of available resources of each 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:

   (a) ***Work = Available***

   (b) For ***i* = 1,2, …, *n***, if $\boldsymbol{Allocation_i \neq 0}$, then ***Finish*[i] = *false***; otherwise, ***Finish*[i] = *true***

   ∗(Different from satety algorithm. *Allocation* = 0 means that the process is not in the wait-for graph.)

2. Find an index ***i*** such that both:

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

   (b) $\boldsymbol{Request_i \leq Work}$ *
   (‘*Request*’ is ‘____’ in Banker’s algorithm)

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

Detection Algorithm (Cont.)

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

4. If ***Finish[i]* == *false***, for some ***i***, 1 ≤ ***i*** ≤ ***n***, then the system is in deadlock state. Moreover, if ***Finish*[*i*] == *false***, then ***P$_i$*** is deadlocked

Algorithm requires an order of O(m x n^2) operations to detect whether the system is in deadlocked state

Example of Detection Algorithm

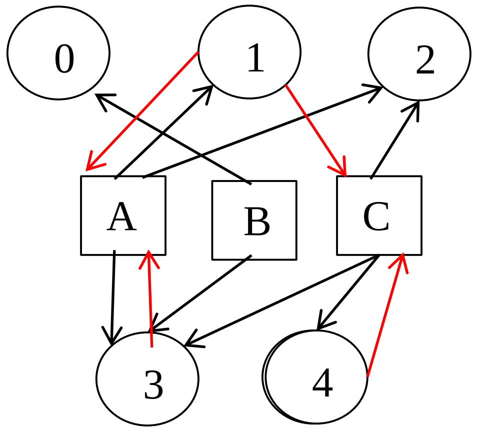
- Five processes ***P*$_0$** through ***P*$_4$**; three resource types A (7 instances), *B* (2 instances), and *C* (6 instances)

- Snapshot at time ***T*$_0$**:

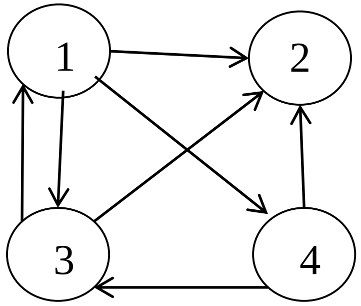
	Allocation	*Request*	*Available*
	A B C	*A B C*	*A B C*
P_0	0 1 0	0 0 0	0 0 0
P_1	2 0 0	2 0 2	
P_2	3 0 3	0 0 0	
P_3	2 1 1	1 0 0	
P_4	0 0 2	0 0 2	

★ (7 2 6)

★



★ (Cycle: (P_1, P_3) (P_1, P_4, P_3))



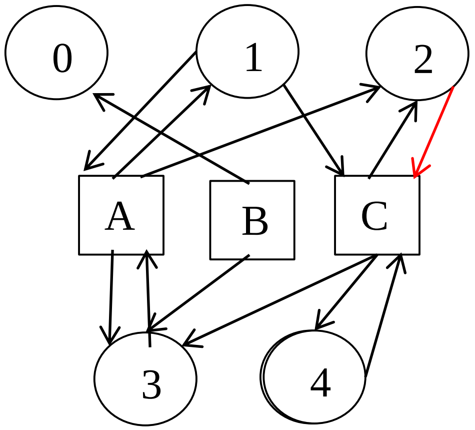
- Sequence <***P*$_0$, *P*$_2$, *P*$_3$, *P*$_1$, *P*$_4$**> will result in ***Finish[i] = true*** for all ***i*** *(Even with 2 cycles, no deadlock due to ________________)

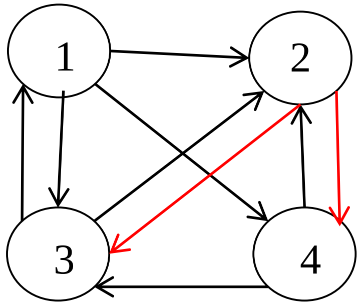
Example (Cont.)

- ***P*$_2$** requests an additional instance of type ***C***

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

* (Additional cycles: (P_1, P_2, P_4 , P_3) (P_1, P_2 , P_3) (P_2, P_3) (P_2, P_4))





- State of system?
  - Can reclaim resources held by process ***P*$_0$**, but insufficient resources to fulfill other processes' requests

    (Work= (0 0 0)) → (P_0 finish; Work = (0 1 0)) → (No way)
  - Deadlock exists, consisting of processes ***P*$_1$**, ***P*$_2$**, ***P*$_3$**, and ***P*$_4$**

Detection-Algorithm Usage

- When, and how often, to invoke depends on:
  - How often a deadlock is likely to occur?
  - How many processes will need to be rolled back?
    - one for each disjoint cycle

  *(One approach is to invoke the algo whenever a request cannot be granted. Then the set of proc that is deadlocked and the specific one causing the deadlock can be identified.)

- If detection algorithm is invoked arbitrarily, there may be many cycles in the resource graph and so we would not be able to tell which of the many deadlocked processes "caused" the deadlock.

  * (Too much overhead for deadlock detection → once per hour or when CPU utilization drops below __%)

Recovery from Deadlock: Process Termination

- Abort all deadlocked processes * (causes lost of ________ computation results)
- Abort one process at a time until the deadlock cycle is eliminated

  * (causes overhead of deadlock detection for each abortion)
- In which order should we choose to abort?
  1. Priority of the process *(___)
  2. How long process has computed, and how much longer to completion *(shorter, longer)
  3. Resources the process has used *(_____ use)
  4. Resources process needs to complete *(_____ need)
  5. How many processes will need to be terminated *(small)
  6. Is process interactive or batch? *(_____ since it can be restarted from the beginning)

Recovery from Deadlock: Resource Preemption

- **Selecting a victim** – minimize cost
- **Rollback** – return to some safe state, restart process for that state
- **Starvation** – same process may always be picked as victim, include number of rollback in cost factor