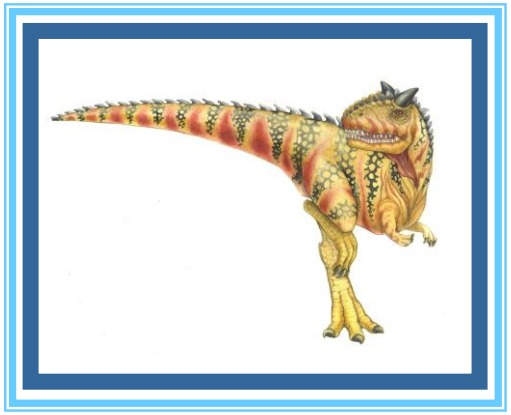
Chapter 7: Deadlocks



Silberschatz, Galvin and Gagne ©2013

Chapter 7: Deadlocks

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

- **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 $R_1, R_2, \ldots, R_m$
  - *e.g., CPU cycles, memory space, I/O devices*
- Each resource type R_i has W_i instances.

- Each process utilizes a resource as follows:
  - **request**
  - **use**
  - **release**

Deadlock with Mutex Locks

- Deadlocks with Mutex Lock

```
Void *thread_A(void *param)
{
   pthread_mutex_lock(&mutex1);
   Pthread_mutex_lock(&mutex2);
    /* Do some work */
   pthread_mutex_unlock(&mutex2);
   Pthread_mutex_unlock(&mutex1);
   Pthread_exit(0);
}
```

```
Void *thread_B(void *param)
{
   pthread_mutex_lock(&mutex2);
   Pthread_mutex_lock(&mutex1);
    /* Do some work */
   pthread_mutex_unlock(&mutex1);
   Pthread_mutex_unlock(&mutex2);
   Pthread_exit(0);
}
```

Deadlock is possible if thread_A acquires mutex1 while thread_B acquires mutex2.

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 $\{P_0, P_1, \ldots, P_n\}$ 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

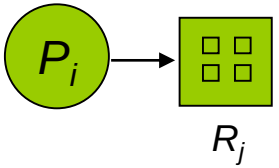
A set of vertices V and a set of edges E.

- V is partitioned into two types:
  - $P = \{P_1, P_2, \ldots, P_n\}$, 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 $R_j \rightarrow P_i$

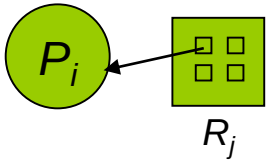
- **Process**

- **Resource Type with 4 instances**

- **P_i requests instance of R_j**

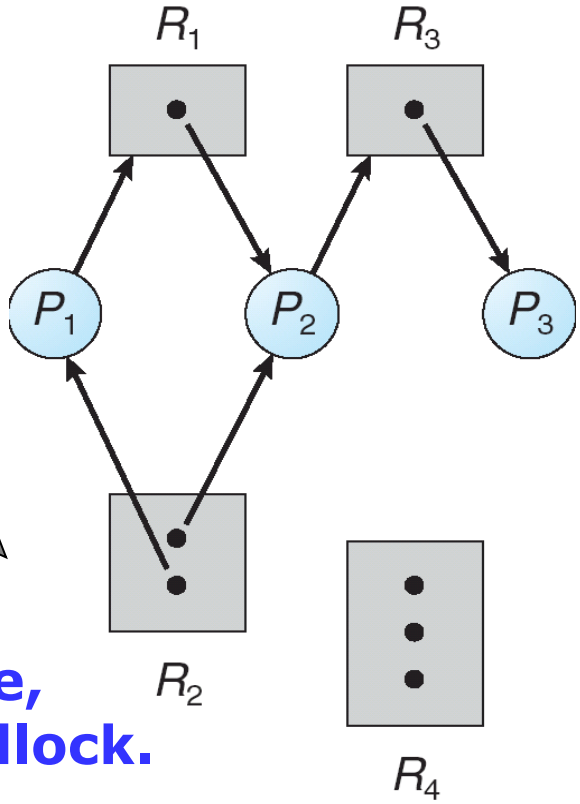


- **P_i is holding an instance of R_j**



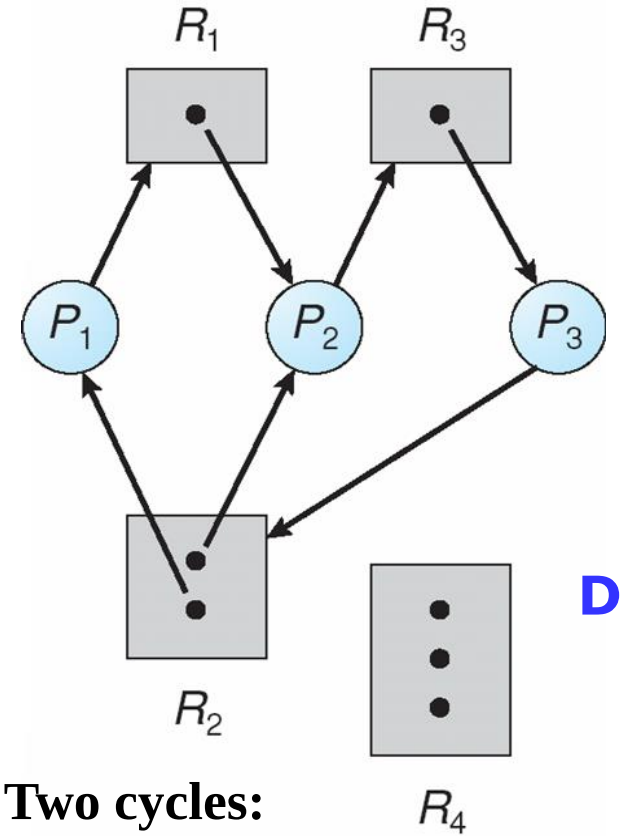
Resource-Allocation Graph (Cont.)

P = {P1, P2, P3}
R = {R1, R2, R3, R4}
R1: 1 instance, R2: 2 instances
R3: 1 instance, R4: 3 instances
E = {P1→R1, P2→R3 , R1→P2 ,
R2→P2, R2→P1, R3 →P3}



No cycle, no deadlock.

Add P3→R2



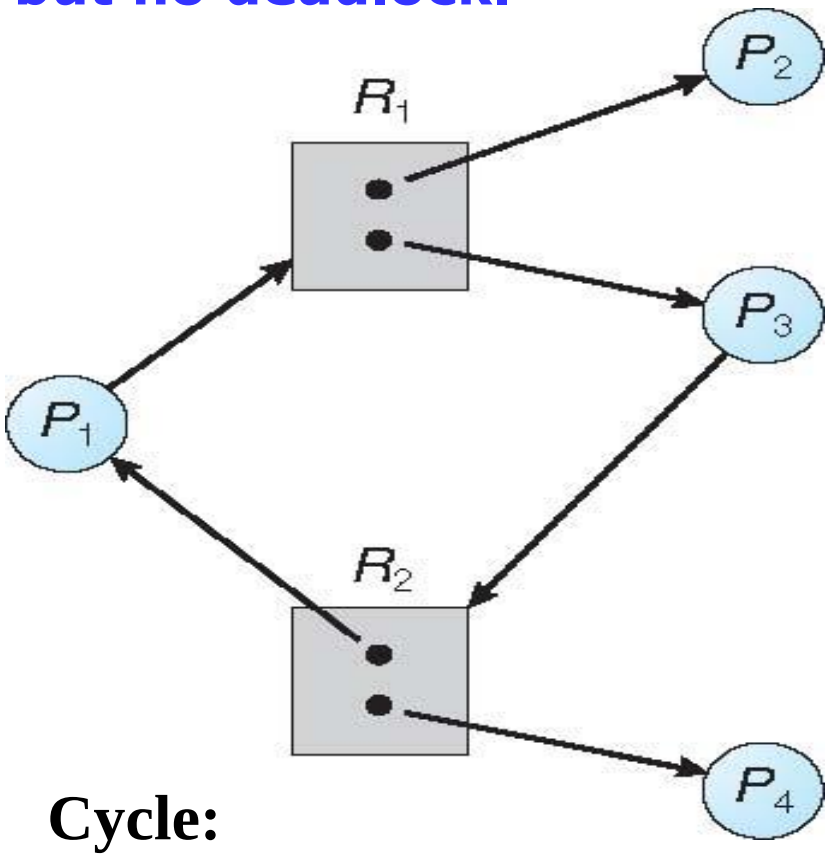
Deadlock

Two cycles:

P1 → R1 → P2 → R3 → P3 → R2 → P1
P2 → R3 → P3 → R2 → P2

Resource-Allocation Graph (Cont.)

With a cycle,
but no deadlock.



Cycle:

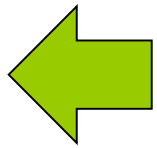
P1 → R1 → P3 → R2 → P1

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***

Chapter 7: Deadlocks

- System Model
- Deadlock Characterization
- Methods for Handling Deadlocks
  - *Deadlock prevention*
  - *Deadlock avoidance*
  - *Deadlock detection*
  - *Deadlock recovery*

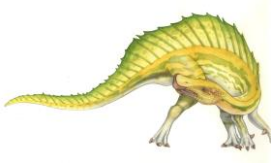


Methods for Handling Deadlocks

- Ensure that the system will ***never*** enter a deadlock state
  - Deadlock prevention
  - Deadlock avoidance
- Allow the system to enter a deadlock state and then recover
  - Deadlock detection
  - Deadlock recovery
- Ignore the problem and pretend that deadlocks never occur in the system; used by most operating systems, including UNIX

Deadlock Prevention

- By ensuring that at least one of four necessary conditions for deadlock cannot hold.
- Constrain the ways request can be made
  - **Mutual Exclusion**
    - In general, we cannot prevent deadlocks by denying the mutual-exclusion condition because some resources are intrinsically nonsharable.
  - **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 (release all the resources it has first)
    - Drawbacks: Low resource utilization



Deadlock Prevention (Cont.)

- **No Preemption of resources** –
  - 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

  ➔ Cannot be applied to resources as mutex locks and semaphores

- **Circular Wait** – impose an ordering of all resource types, and each process must request resources in an increasing order of enumeration
  - Assign to each resource type a unique number F: R → N
  - A process holding Ri can acquire Rj if and only if F(Rj) > F(Ri)
  - A process acquire Rj must release Rk it had if F(Rk) > F(Rj)
  - if several instances of the same resource type are needed, a **single** request for all of them must be issued.

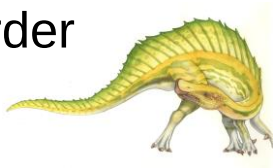
Deadlock Prevention (circular wait)

- If the lock ordering for the example in page 7.4 was
  - F(mutex1) = 1
  - F(mutex2) = 5
- then thread_B cannot request the locks out of order ➔ no deadlock.

- F should be defined according to normal order of resource usage
  - e.g., if tape drive is usually needed before the printer, then it is reasonable to define F(tape drive) < F(printer)

- **witness**: a Lock-order verifier in FreeBSD
  - Dynamically maintain the relationship of lock orders in a system.
  - For example in page7.4, assume thread_A is the first to acquire the locks, witness records the relationship that F(mutex1) < F(mutex2).

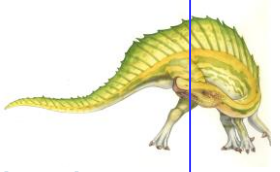
    ➔ gives a warning if thread_B later acquires locks out of order



Deadlock Example with Lock Ordering

- Imposing a lock ordering does not guarantee deadlock prevention if locks can be acquired dynamically
  - e.g., two threads simultaneously invoke transaction() as follows.
    thread1 – transaction(account1, account2, 25);
    thread2 – transaction(account2, account1, 50);

```
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);
}
```



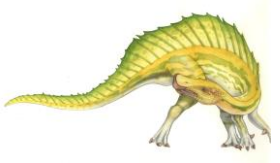
Deadlock Avoidance

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 with $j < i$ 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



Safe State (conti.)

- Consider a system with 12 tape drives and three processes: P0, P1, P2
- at t0

	Max needs	current ~~Needs~~ allocation
P0	**10**	**5**
P1	**4**	**2**
P2	**9**	**2**

Safe state

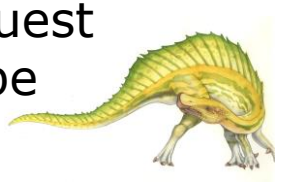
The sequence **<P1, P0, P2>** satisfies the safety condition.

- at t1

	Max needs	current ~~Needs~~ allocation
P0	**10**	**5**
P1	**4**	**2**
P2	**9**	**3**

unSafe state

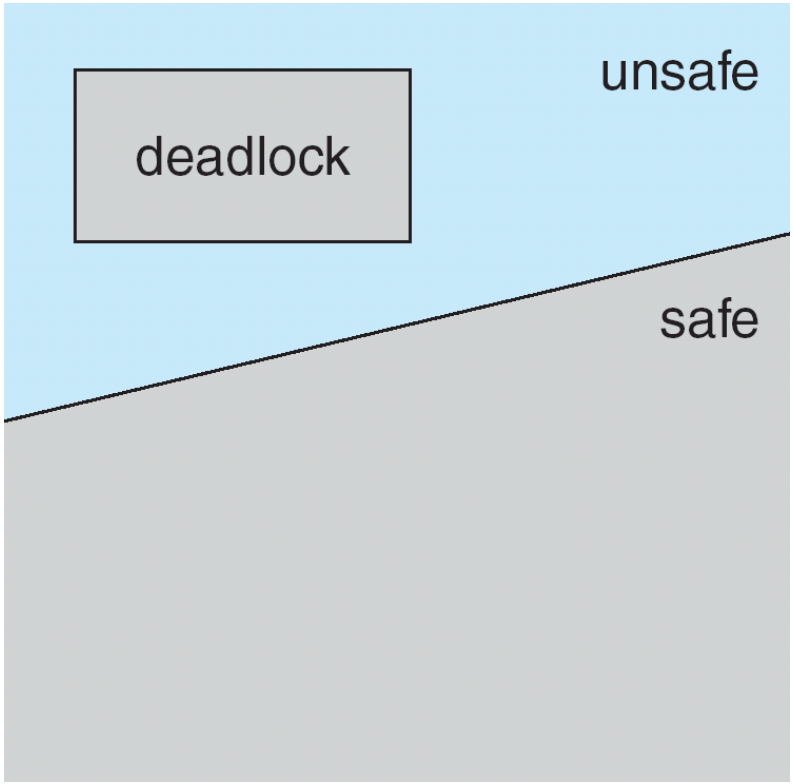
Should not grant the request from P2 for one more tape

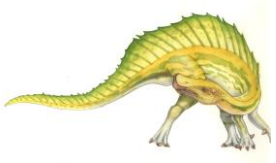


Basic Facts

- If a system is in safe state
  ⇒ no deadlocks
- If a system is in unsafe state
  ⇒ possibility of deadlock
- Avoidance

  ⇒ ensure that a system will never enter an unsafe state.





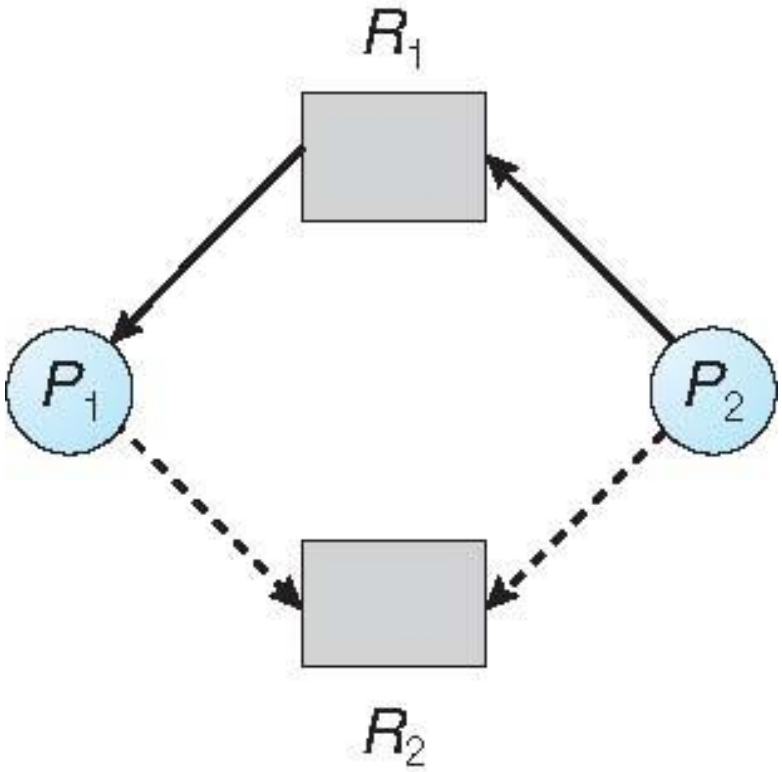
Avoidance algorithms

- 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



Safe state

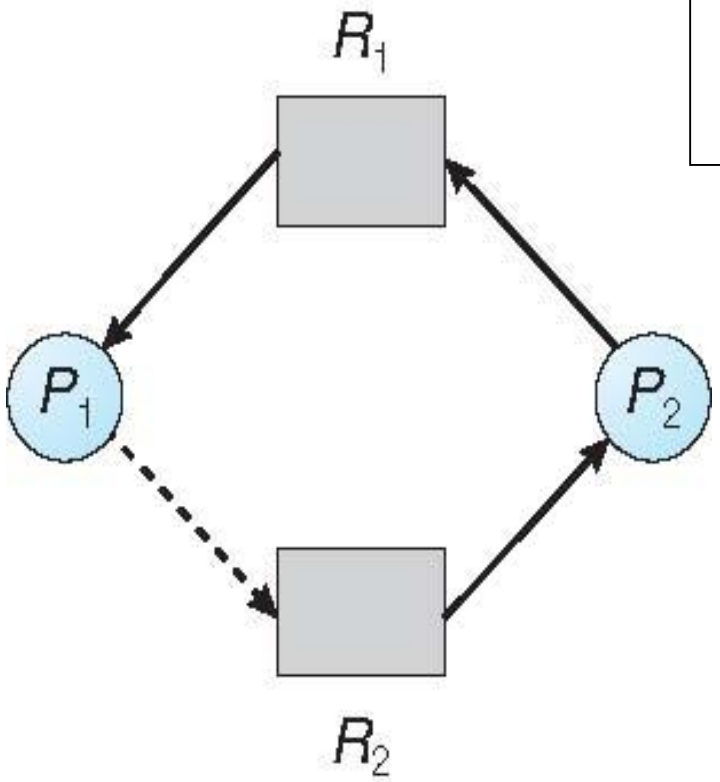
Suppose that process P_i requests a resource R_j.

Still safe?

Unsafe State In Resource-Allocation Graph

The request can be granted only if

➔ converting the request edge Pi → Rj to an assignment edge Rj → Pi does not result in a cycle in the resource-allocation graph

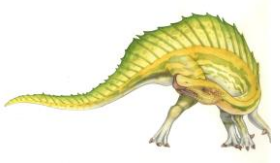


Cannot allocate R2 to P2 because

Unsafe state

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

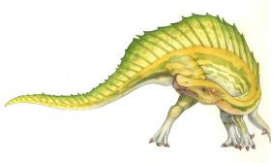


Data Structures for the Banker's Algorithm

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 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

$$Need\ [i,j] = Max[i,j] - Allocation\ [i,j]$$



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**

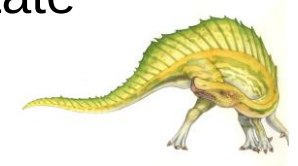
   (b) $Need_i \leq Work$

   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



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

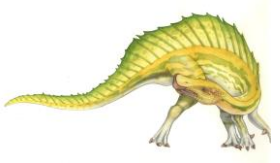
1. If $Request_i \leq Need_i$ go to step 2. Otherwise, raise error condition, since process has exceeded its maximum claim
2. If $Request_i \leq Available$, go to step 3. Otherwise P_i must wait, since resources are not available
3. Pretend to allocate requested resources to P_i by modifying the state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

$$Available = Available - Request;$$

$$Allocation_i = Allocation_i + Request_i;$$

$$Need_i = Need_i - Request_i;$$

- If safe $\Rightarrow$ the resources are allocated to ***Pi***
- If unsafe $\Rightarrow$ ***Pi*** must wait, and the old resource-allocation state is restored



Example of Banker's Algorithm

- 5 processes P_0 ~ P_4;

  3 resource types:

  ***A* (10 instances),**

  ***B* (5 instances),**

  ***C* (7 instances)**

- at time T_0 :

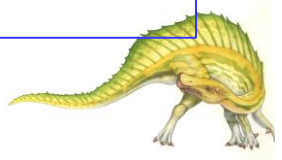
	Allocation	Max	Available
	A B C	*A B C*	*A B C*
P_0	0 1 0	7 5 3	3 3 2
P_1	2 0 0	3 2 2	
P_2	3 0 2	9 0 2	
P_3	2 1 1	2 2 2	
P_4	0 0 2	4 3 3	

- ***Need*** is defined as ***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

In safe state

because **< P_1, P_3, P_4, P_2, P_0>** satisfies safety criteria

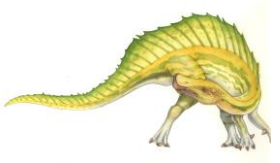


Example: P_1 Request (1,0,2)

- Can request for (1,0,2) by P_1 be granted?
- Check that Request ≤ Available (that is, (1,0,2) ≤ (3,3,2) ⇒ 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
- At this stage, can request for (3,3,0) by P_4 be granted?
- At this stage, can request for (0,2,0) by P_0 be granted?



Example: P_0 Request (0,2,0)

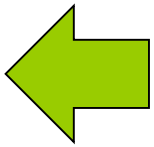
Can request for (0,2,0) by P_0 be granted?

- check if request < available (it is true because (0,2,0) < (2,3,0))
- Pretend that this request has been fulfilled, then we arrive at the new state as follows.
- new state is **unsafe** ➔ request cannot be granted.

	Allocation	Need	Available
	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 2	6 0 0	
P_3	2 1 1	0 1 1	
P_4	0 0 2	4 3 1	

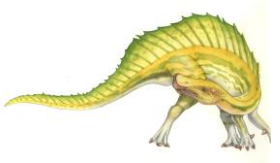
Chapter 7: Deadlocks

- System Model
- Deadlock Characterization
- Methods for Handling Deadlocks
  - *Deadlock prevention*
  - *Deadlock avoidance*
  - *Deadlock detection*
  - *Deadlock recovery*



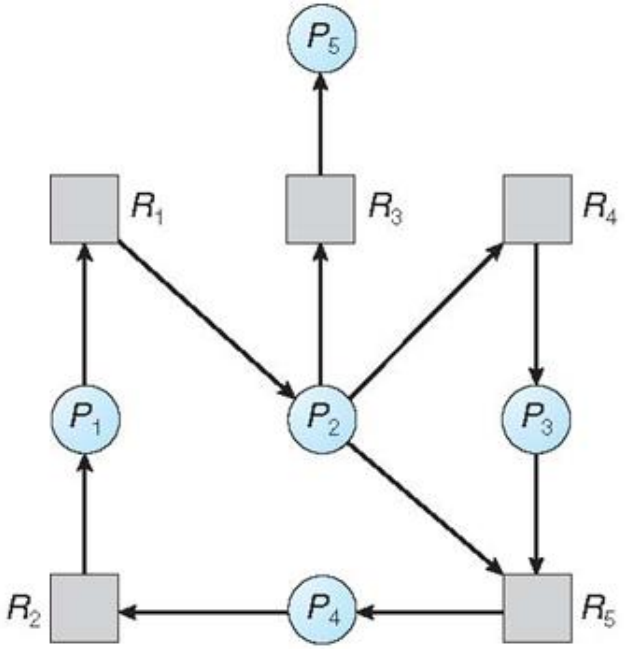
Deadlock Detection

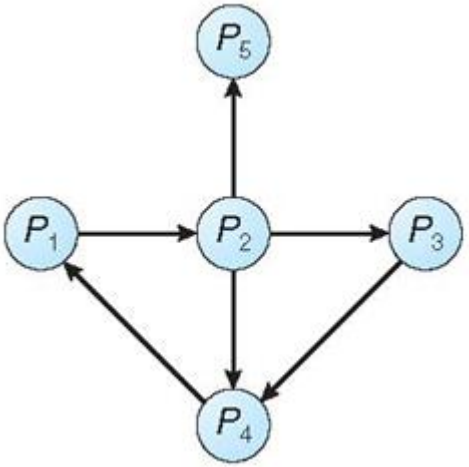
- Allow system to enter deadlock state
- Detection algorithm
- Recovery scheme



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
- algorithm complexity: $O(n^2)$, n is the number of vertices in the graph

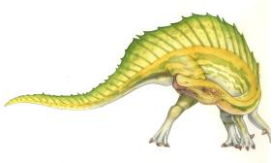




Wait-for graph

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 ***Allocation*$_i$ ≠ 0**,
   then ***Finish*[i] = *false***;
   otherwise, ***Finish*[i] = *true***

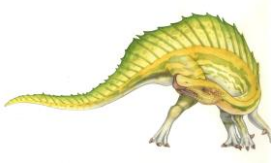
2. Find an index ***i*** such that both:
   (a) ***Finish*[*i*] == *false***
   (b) ***Request*$_i$ ≤ Work**
   If no such ***i*** exists, go to step 4

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

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

Algorithm of detecting whether the system is in deadlocked state:

O(m x n^2) operations



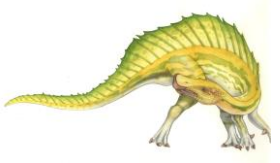
Example of Detection Algorithm

- 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***



Example (Cont.)

- P_2 requests an additional instance of type ***C***

	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 1**	
P_3	2 1 1	1 0 0	
P_4	0 0 2	0 0 2	

- State of system?
  - Although we can reclaim resources held by process P_0, insufficient resources to fulfill requests of other processes/
  - Deadlock exists, consisting of processes P_1, P_2, P_3, and P_4

Detection-Algorithm Usage

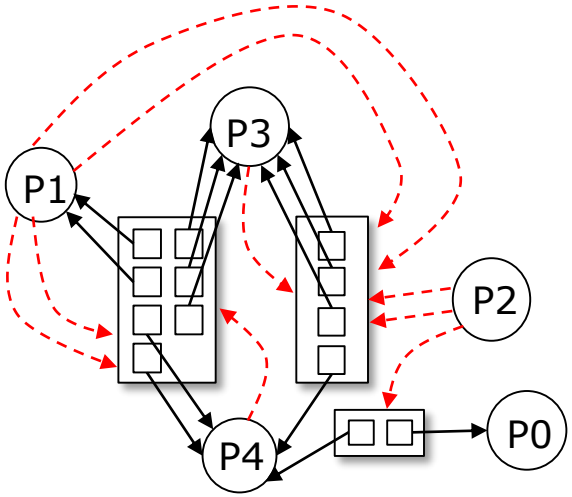
- When, and how often, to invoke depends on:
  - How often a deadlock is likely to occur?
  - How many processes will need to be rolled back?
    - one for each disjoint cycle
- If detection algorithm is invoked arbitrarily, there may be many cycles in the resource graph and so we would not be able to tell which of the many deadlocked processes “caused” the deadlock.

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?
  1. Priority of the process
  2. How long process has computed, and how much longer to completion
  3. Resources the process has used
  4. Resources process needs to complete
  5. How many processes will need to be terminated
  6. Is process interactive or batch?

Recovery from Deadlock: Resource Preemption

- Issues to be addressed for resource preemption
  - **Selecting a victim** – minimize cost (e.g., the number of resources a deadlocked process is holding, the amount of time the process has consumed thus far.
  - **Rollback** – return the process to some safe state, restart it for that state (hard to determine safe state for a process, so the simplest way is a total rollback.
  - **Starvation** – same process may always be picked as victim ➔ to include number of rollback in cost factor in order to ensure that a process can be picked as a victim only a small number of times .


P3
P1
P2
P0
P4