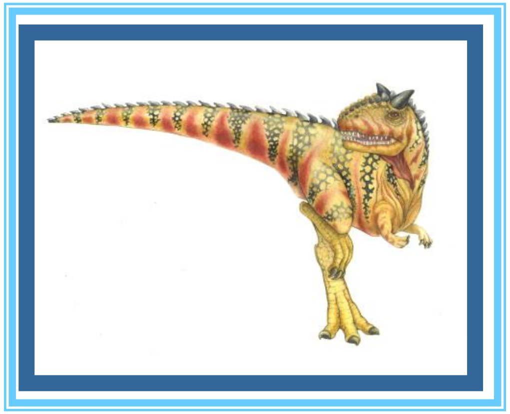
Chapter 7: Deadlocks



Chapter 7: Deadlocks

- The Deadlock Problem
- 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;

The Deadlock Problem

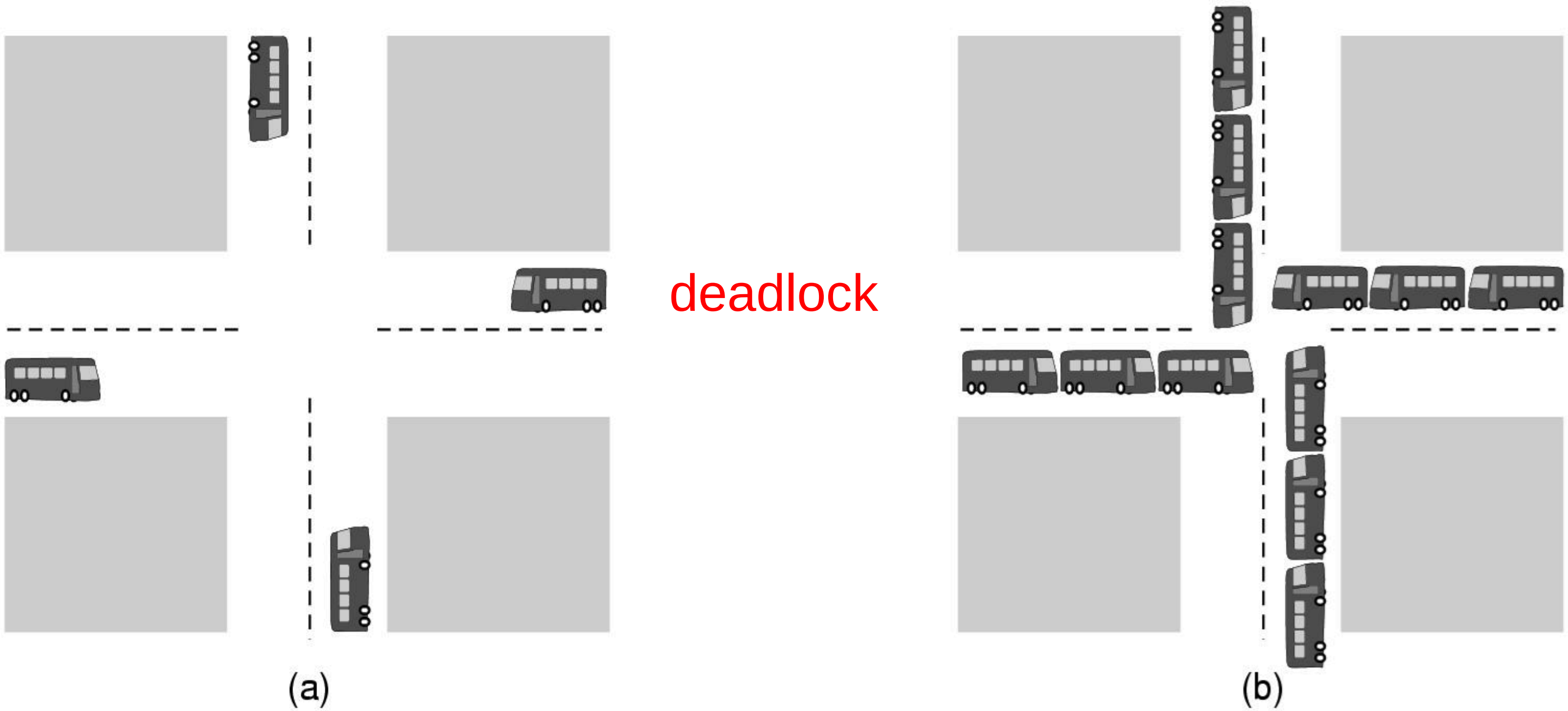
- A set of blocked processes each holding a resource and waiting to acquire a resource held by another process in the set.
- The resources may be either physical resources or logical resources.

The Deadlock Problem

- Example
  - ✓ System has 2 disk drives
  - ✓ P_1 and P_2 each hold one disk 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)

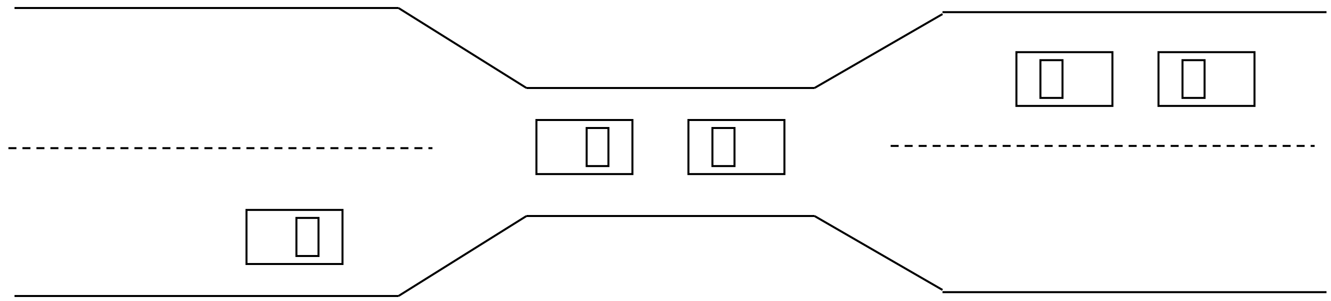
Deadlock Example



(a) A potential deadlock

(b) an actual deadlock

Bridge Crossing Example



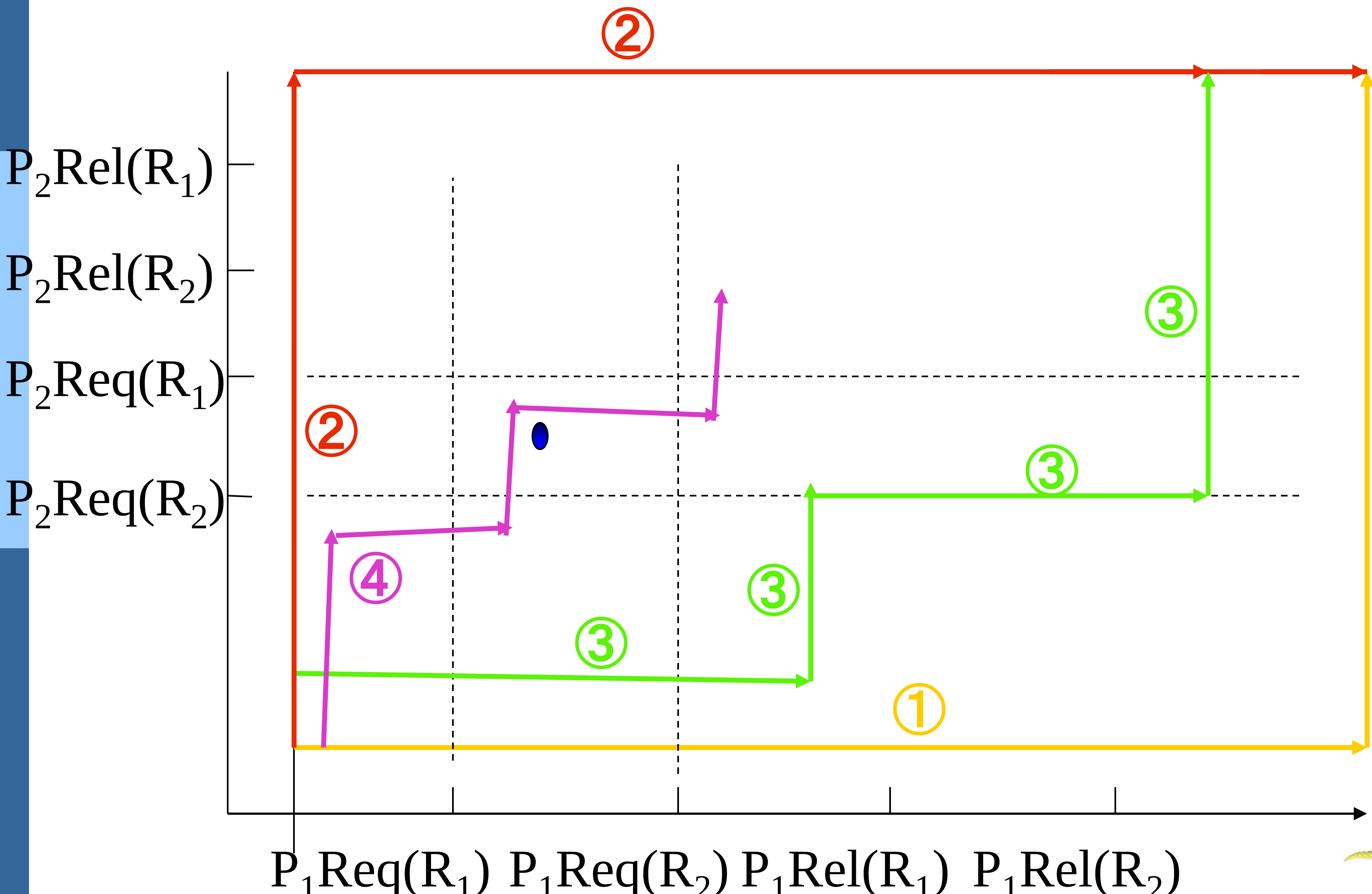
- Traffic only in one direction
- Each section of a bridge can be viewed as a resource
- If a deadlock occurs, it can be resolved if one car backs up (preempt resources and rollback)
- Several cars may have to be backed up if a deadlock occurs
- Starvation is possible
- Note – Most OSes do not prevent or deal with deadlocks

Why Deadlock?

- Resource competition
  - ✓ The number of resources is not sufficient for processes requirements
- Advance order of Concurrent processes is not appropriate. In a multiprogramming system, it is asynchronous for concurrent processes execution
  - ✓ In some order, each process can complete successfully
  - ✓ In some order, there will be a deadlock

Concurrent processes P1,P2; resources R1 and R2 has only one instance respectively

Curve4 will lead to unsafe area



System Model

- Resource types R_1, R_2, . . ., R_m
  - *CPU cycles, memory space, I/O devices*
- Each resource type R_i has W_i instances.
- The number of resources requested may not exceed the total number of resources available in the system.

How process utilizes a resource?

- Each process utilizes a resource as follows:
  - **request**
  - **use**
  - **Release**
- The request and release are system calls, such as open() and close() file, and allocate() and free() memory system calls.

Deadlock with Semaphores

- Data:
  - A semaphore $\mathbf{s}_1$ initialized to 1
  - A semaphore $\mathbf{s}_2$ initialized to 1
- Two threads T_1 and T_2
- T_1:

  wait(s_1)

  wait(s_2)
- T_2:

  wait(s_2)

  wait(s_1)

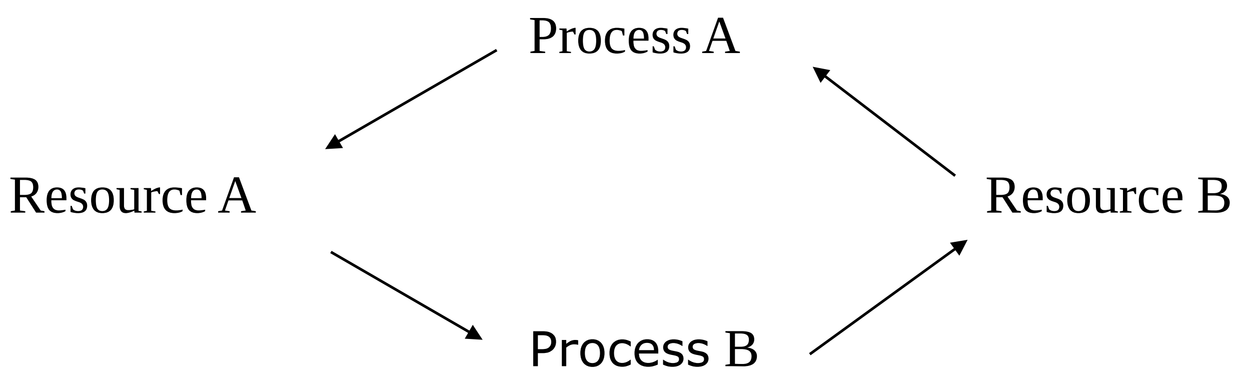
Necessary Conditions for Deadlock

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

Necessary Conditions for Deadlock(Cont)

- **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.

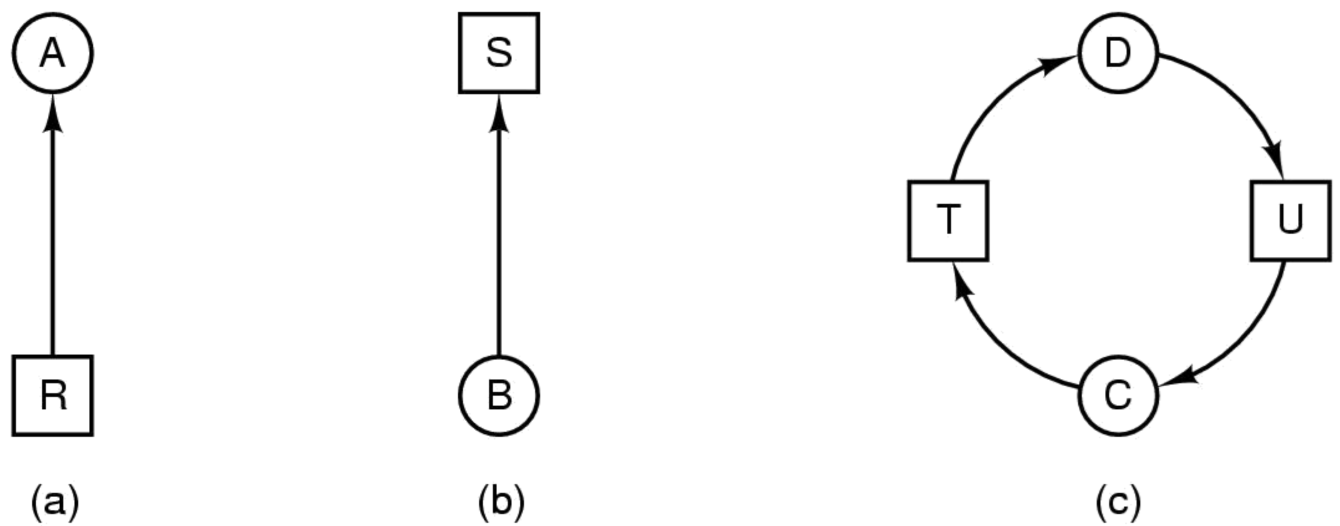


Resource-Allocation Graph

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

- 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 $R_j \rightarrow P_i$

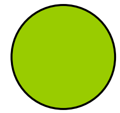
Resource-Allocation Graph



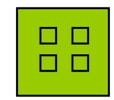
- Resource R assigned to process A
- Process B is requesting/waiting for resource S
- Process C and D are in deadlock over resources T and U

Resource-Allocation Graph (Cont.)

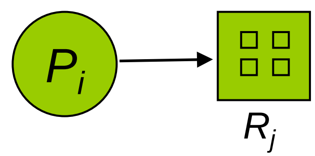
- Process



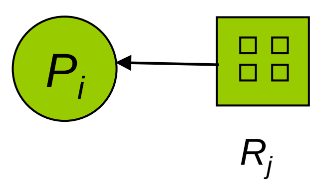
- Resource Type with 4 instances



- P_i requests instance of R_j

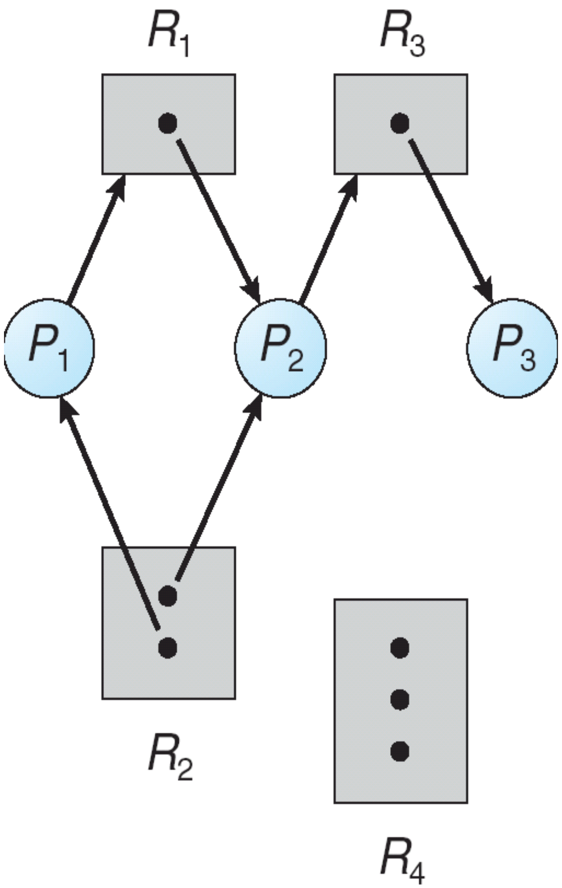


- P_i is holding an instance of R_j

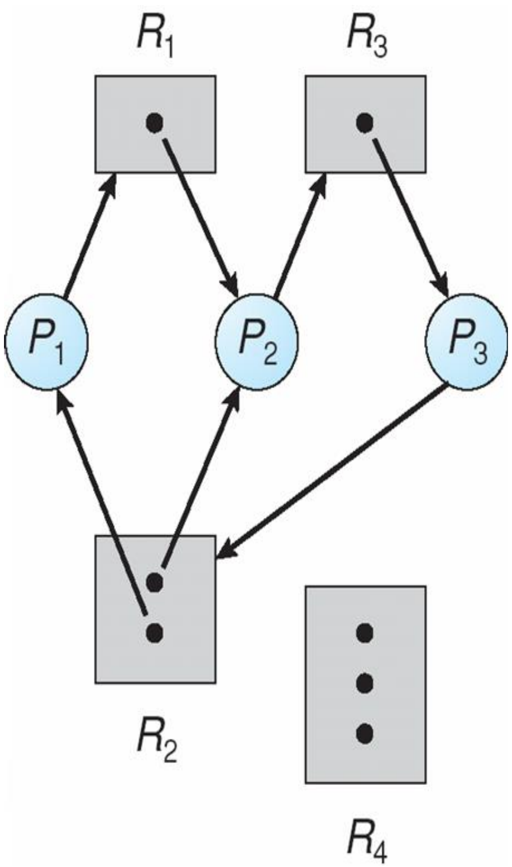


Resource Allocation Graph

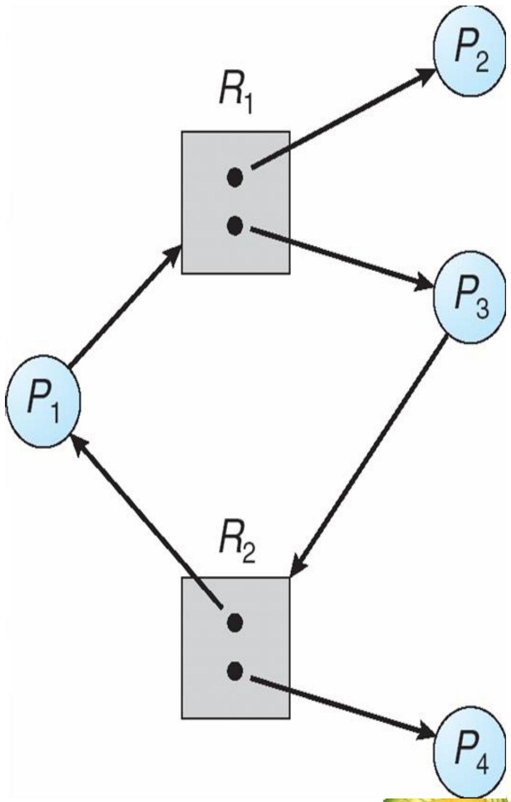
Example of a Resource Allocation Graph



Resource Allocation Graph With A Deadlock



Graph With A Cycle But No Deadlock



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
  - Deadlock prevention（静态预防）
  - Deadlock avoidance（动态避免）
- Allow the system to enter a deadlock state and then recover
  - Deadlock detection and recovery（动态消除）
- Ignore the problem and pretend that deadlocks never occur in the system; used by most operating systems, including UNIX: Ostrich policy

Deadlock Prevention

- Restrain the ways request can be made, so that at least one of the necessary conditions for deadlock can not occur.
- Possible side effects:
  - Low device utilization
  - Reduced system throughput

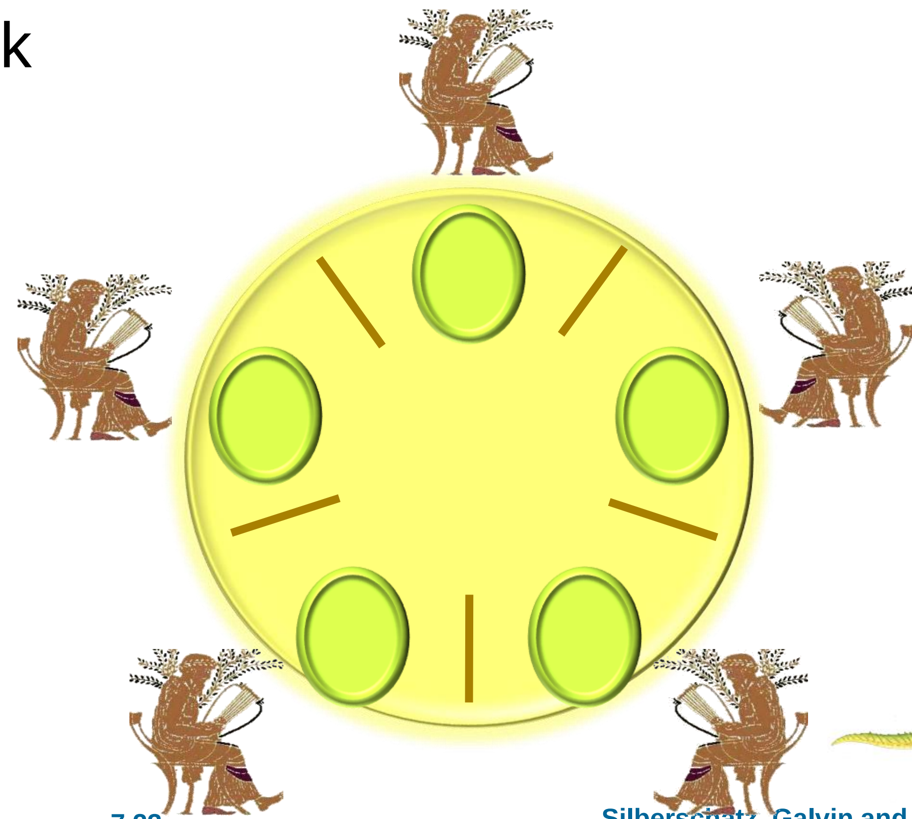
- **Mutual Exclusion**
  - not required for sharable resources; must hold for nonsharable resources.

Deadlock Prevention

- **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

Dining Philosophers

- 5 philosophers sitting around a round table
- 1 chopstick in between each pair of philosophers
  - 5 chopsticks total
- Each philosopher needs two chopsticks to eat
- How to prevent deadlock



Deadlock Prevention (Cont.)

- **No Preemption** –

  方案1

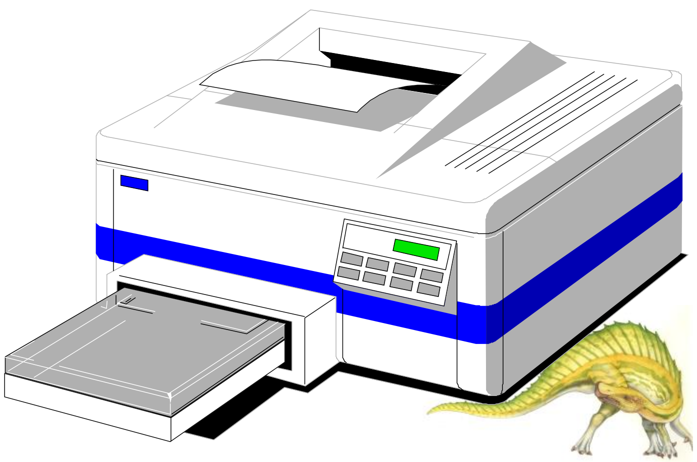
  - If a process that is holding some resources requests another resource that cannot be immediately allocated to it, then all resources currently being held are released
  - Preempted resources are added to the list of resources for which the process is waiting
  - Process will be restarted only when it can regain its old resources, as well as the new ones that it is requesting

  方案2

  - When process A requests some unavailable resources, we check whether they are allocated to another process B that is waiting for other resources;
  - If so, we preempt the desired resources from B and allocate them to A.
  - Otherwise, A must wait.

Deadlock Prevention (Cont.)

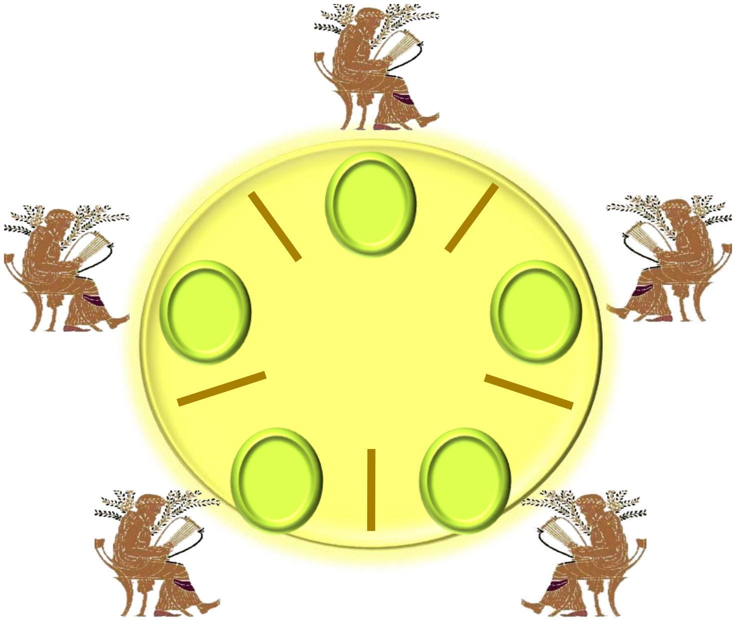
- Allow Preemption!
- Can preempt CPU by
  - saving its state to thread control block and resuming later
- Can preempt memory by
  - swapping memory out to disk and loading it back later
- Can we preempt the holding of a lock?
- Some resource cannot be preempted
- Consider a process given the printer
  - halfway through its job
  - now forcibly take away printer
  - !!??



Deadlock Prevention (Cont.)

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

For example, Dining Philosophers



Deadlock Avoidance

Requires that the system has some additional *a priori* information available. E.g, which type resources a process needs? How many instances of each resource type a process needs at most?

Deadlock Avoidance

- 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;
- When a process requests an available resource, system must decide if immediate allocation leaves the system in a safe state.

Safe State

- System is in safe state if there exists a sequence $< P_1, P_2, \ldots, P_n >$, 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: $< P_1, P_2, \ldots P_j\ P_i, \ldots P_n >$
  - 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

example

- processes:P_1、P_2 and P_3. 12 tapes. Snapshot at T_0 :

process	max	allocated	need	available
P_1	10	5	5	3
P_2	4	2	2	
P_3	9	2	7	

- sequence（P_2、P_1、P_3）satisfies the safety condition：
  - Allocate 2 of the 3 available tapes to P_2, which satisfies the max need of P_2. when P_2 finishes, the system will have 5 available tapes.
  - Then allocate the 5 tapes to P_1.when P_1 finishes, the system will have 10 available tapes.
  - At last, allocate 7 of the 10 tapes to P_3
  - **So, system is safe at T_0.**

System goes from a safe state to an unsafe state

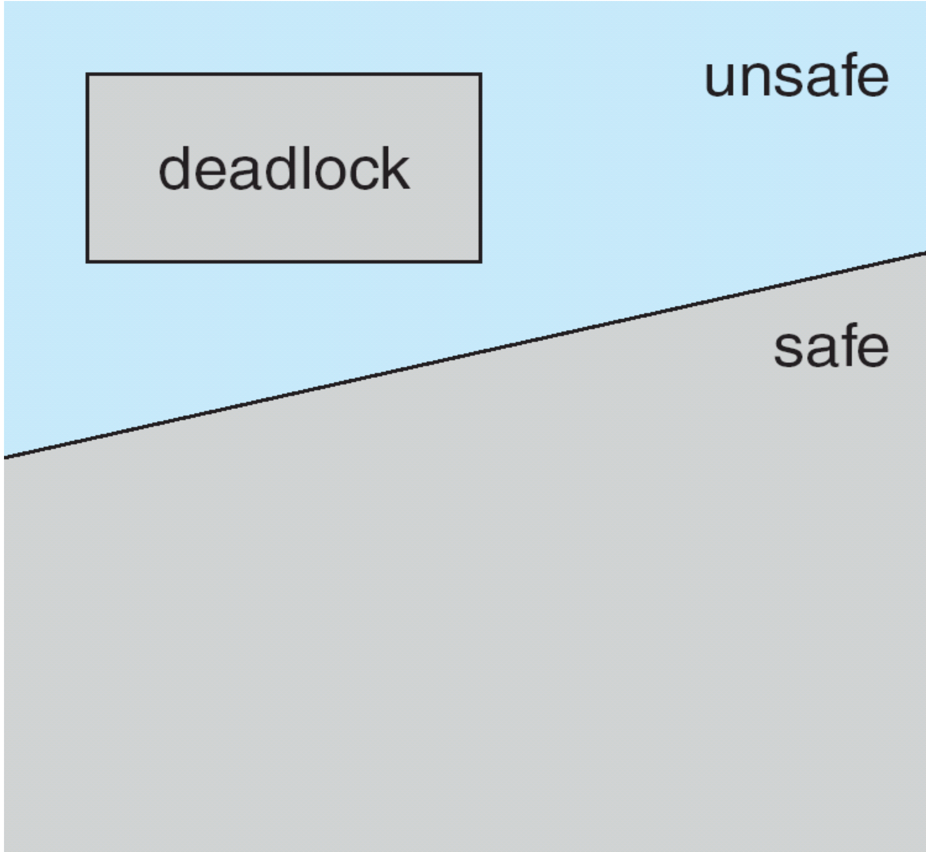
- Suppose that, at $T_{0,}$ P_3 requests a tape and is allocated one tape. Then system goes from T_0 to $T_{1.}$

Process	max	allocated	need		available before allocating	available after releasing
P_1	10	5	5	>		
P_2	4	2	2	=<	2	4
P_3	9	3	6	>		

- At $T_{1,}$ we can not find a sequence satisfying safety condition, so, it is unsafe.
- Therefore, when P_3 requests a tape at $T_{0,}$ we must make it wait to avoid deadlock.

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

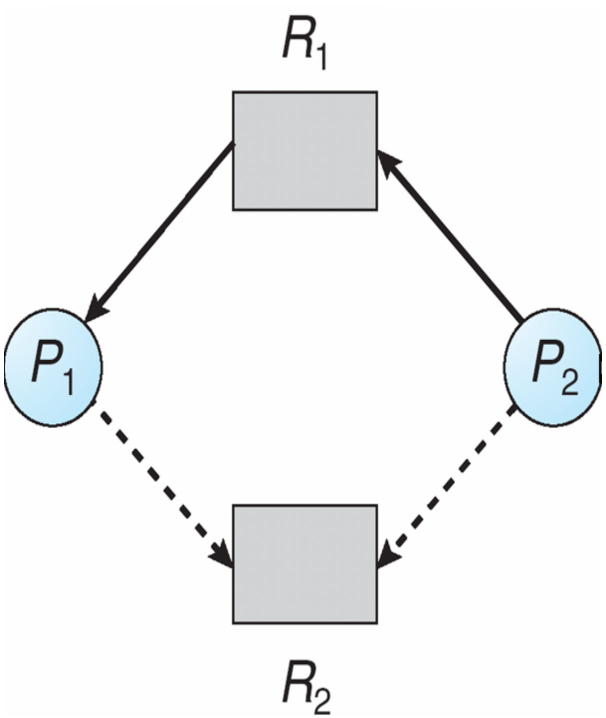
Avoidance algorithms

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

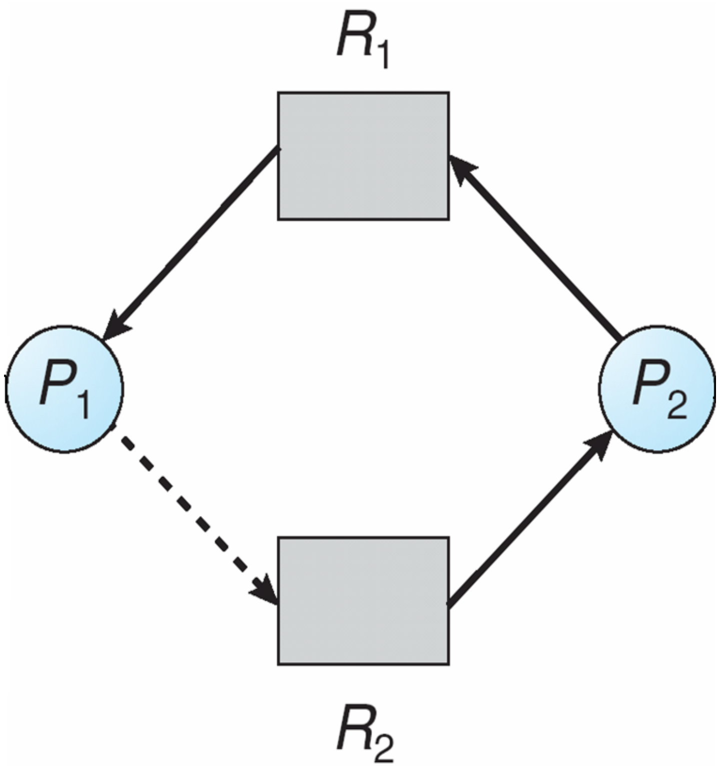
Resource-Allocation Graph Scheme

- 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



Unsafe State In Resource-Allocation Graph



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, 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 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 and *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 $Request_i \le Need_i$ go to step 2. Otherwise, raise error condition, since process has exceeded its maximum claim
2. If $Request_i \le Available$, 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 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	

- *Need* = *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

Example (Cont.)

- To find a safe sequence using safety algorithm

  Work[]=available=(3,3,2); Finish[i]=false (i=0..4);

	Work	need	allocation	finish
P1	3 3 2	1 2 2	2 0 0	T
P3	5 3 2	0 1 1	2 1 1	T
P4	7 4 3	4 3 1	0 0 2	T
P2	7 4 5	6 0 0	3 0 2	T
P0	10 4 7	7 4 3	0 1 0	T

- 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.)

- Usually, there would be more than one safe sequences if the system is in safe state.

Work[]=available=(3,3,2); Finish[i]=false (i=0..4)

	Work	need	allocation	finish
P3	3 3 2	0 1 1	2 1 1	T
P4	5 4 3	4 3 1	0 0 2	T
P1	5 4 5	1 2 2	2 0 0	T
P2	7 4 5	6 0 0	3 0 2	T
P0	10 4 7	7 4 3	0 1 0	T

- Safe sequence：（P3，P4，P1，P2，P0）

Example: P_1 Request (1,0,2)

- 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 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 < 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?

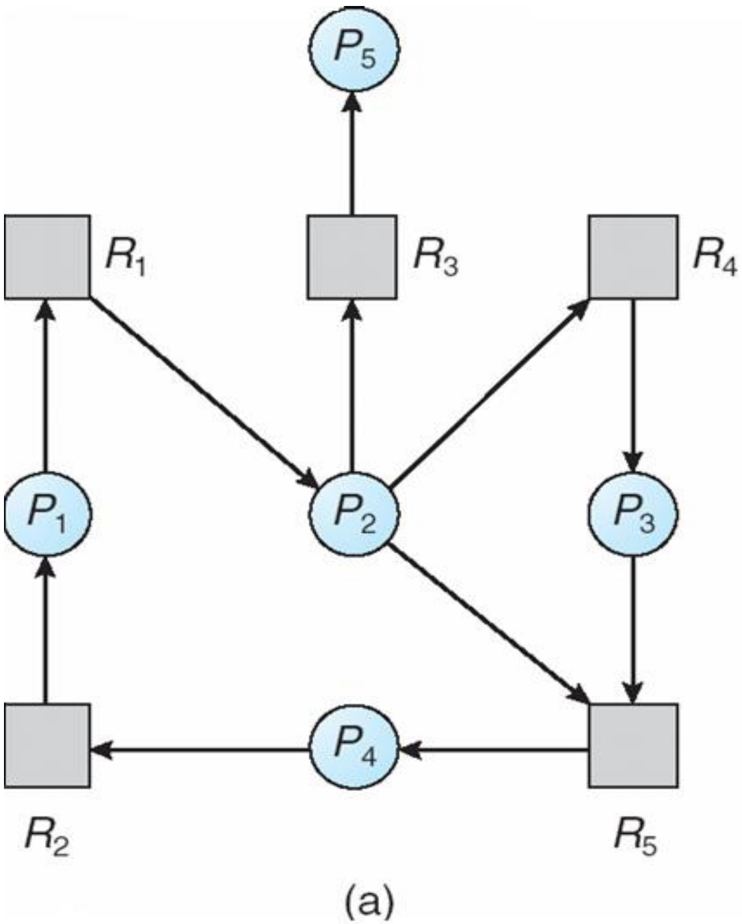
Deadlock Detection

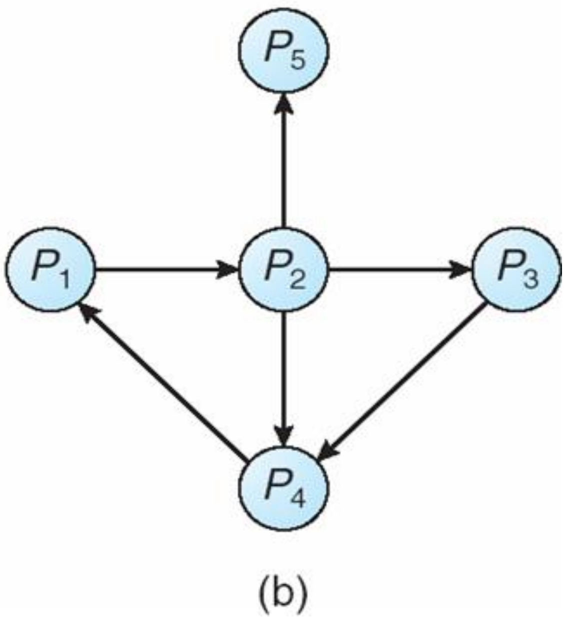
- Allow system to enter deadlock state
- Detection algorithm
- Recovery scheme

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

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 $Allocation_i \neq 0$, then
   Finish[i] = false; otherwise, *Finish*[i] = *true*

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 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	

- 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

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

- State of system?
  - Can reclaim resources held by process P_0, but insufficient resources to fulfill other processes requests
  - 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

Deadlock occurs only when some process makes a request that can not be granted immediately:

- We can invoke deadlock detection algorithm every time a resource request cannot be granted immediately.
  - Lead to a considerable overhead in computation time.
- A less expensive alternative is to invoke the algorithm at less frequent intervals.
  - Once per hour
  - Whenever CPU utilization drops below 40%.

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 for that state
- Starvation – same process may always be picked as victim, include number of rollback in cost factor

assignment

1. 7.1
2. 7.9
3. 7.13