Deadlocks

Operating Systems (ECEg-4181)

Mequanent Argaw Muluneh

Wednesday, April 29, 2020

Outline

- ❖ 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

- A system consists of a finite number of resources to be distributed among a number of competing processes.
- Resources can be partitioned into several types each consisting of some number of identical instances. Types:
  - CPU cycles, memory space, I/O devices
- Each process utilizes a resource in the following sequence:
  - Request
  - Use
  - Release
- A set of processes is in a deadlocked state when every process in the set is waiting for an event that can be caused only by another process in the set.

Deadlock Characterization

Necessary Conditions

- Deadlock can arise if the following 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.

Deadlock Characterization ...

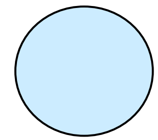
Resource-Allocation Graph

- Deadlocks can be described more precisely in terms of a directed graph called a ***system resource-allocation graph*** which consists of a set of vertices V and a set of edges E.
- V is partitioned into two different types of nodes:
  - $P = \{P_1, P_2, ..., P_n\}$, the set consisting of all the processes in the system.
  - $R = \{R_1, R_2, ..., R_m\}$, the set consisting of all resource types in the system.
- **An edge may be either:**
  - **a request edge** – a directed edge $P_i \rightarrow R_j$ or
  - **an assignment edge** – a directed edge $R_j \rightarrow P_i$

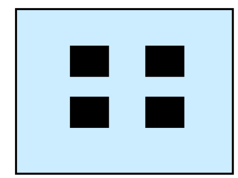
Deadlock Characterization ...

Resource-Allocation Graph ...

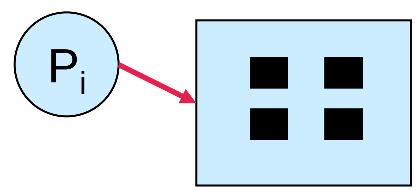
- A process



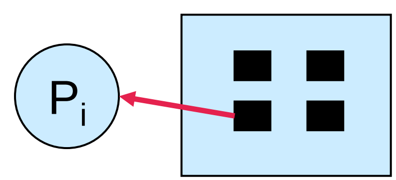
- A resource type with 4 instances



- P_i requests instance of R_j



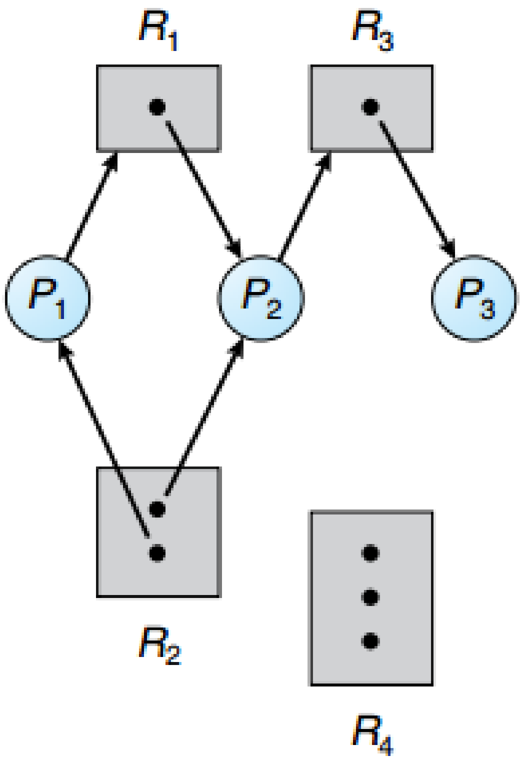
- P_i holds an instance of R_j



Deadlock Characterization ...

Resource-Allocation Graph ...

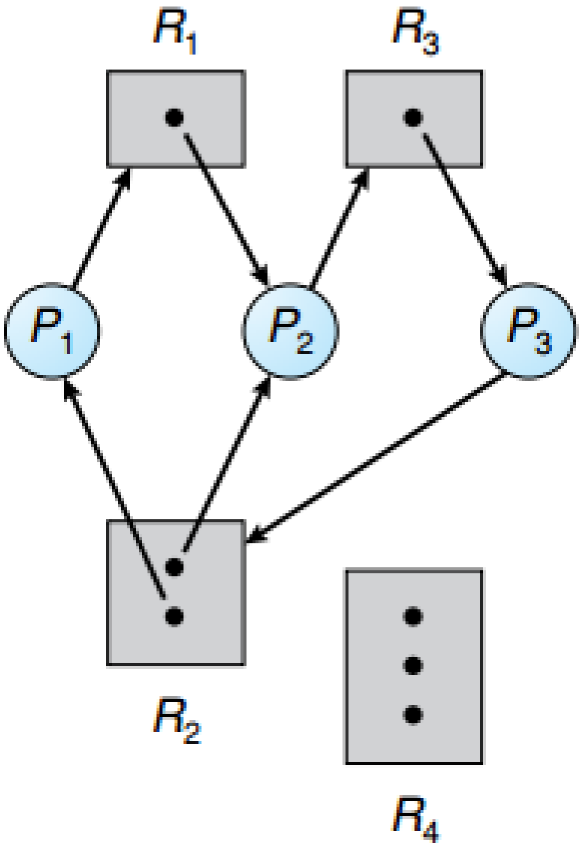
- ❖ Example of a resource-allocation graph
- ❖ Circles represent processes whereas rectangles represent resources.
- ❖ Request edges extend from circles to rectangles whereas assignment edges extend from a specific instance inside the rectangle (resource type) to the requesting circle (process).



Deadlock Characterization …

Resource-Allocation Graph …

- Example of a resource allocation graph with a deadlock
  - Two circles
    - P1, R1, P2, R3, P3, R2, P1
    - P2, R3, P3, R2, P2

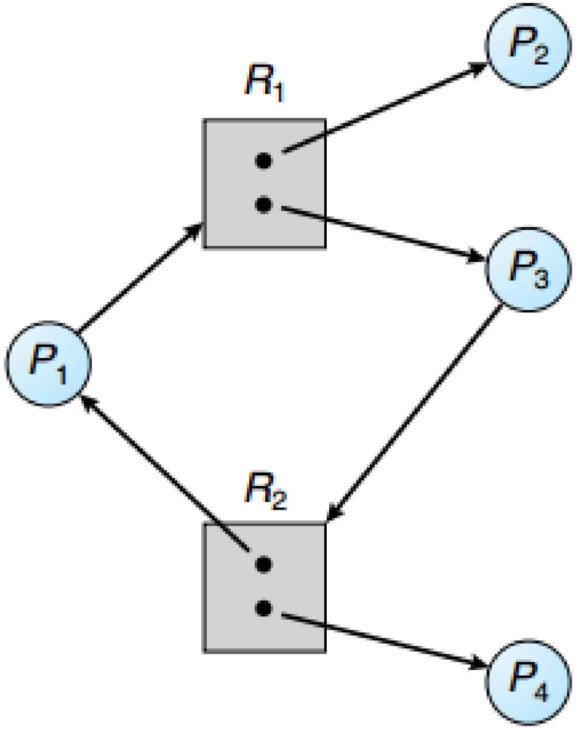


Deadlock Characterization …

Resource-Allocation Graph …

- ❖ Example of a resource allocation graph with a cycle but no deadlock.

- ❖ The circle P1, R1, P3, R2, P1 may not be a deadlock since P4 can release one of the instances of R2.



Deadlock Characterization ...

Resource-Allocation Graph ...

- Basic Facts
  - If a graph contains no cycles ⇒ no deadlock state will occur.
  - If a graph contains a cycle ⇒
    - if only there is one instance per resource type, then deadlock state occurs.
    - if there are several instances per resource type, there will be a possibility of deadlock state.

Methods for Handling Deadlocks

- We can deal with the deadlock problem in one of three ways:
  - 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.
  - Ignore the problem and pretend that deadlocks never occur in the system; used by most operating systems, including Linux and Windows.

Deadlock Prevention

- We can **prevent** deadlock occurrence by ensuring that at least one of the ***four*** necessary conditions cannot hold.
- **Mutual Exclusion** – not required for sharable resources (e.g., read-only files); must hold for non-sharable resources.
- **Hold and Wait** (never occurs)– we must guarantee that whenever a process requests a resource, it does not hold any other resources.
  - Two protocols: require a process to request and be allocated all its resources before it begins execution, **or** allow a process to request resources only when the process has none allocated to it.
  - Both protocols may lead to low resource utilization and starvation.

Deadlock Prevention …

- **No Preemption**: is the third necessary condition for deadlocks.
- To ensure that this condition does not hold, we can use the following protocol.
- 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.

Deadlock Prevention …

- **Circular Wait**: one way to ensure that this condition never holds is to impose a total ordering of all resource types, and require that each process requests resources in an increasing order of enumeration.
- Let R = {R1, R2, …, Rm} be the set of resource types having a unique integer number for each.
- Formally, we define a one-to-one function F: R → N, where N is the set of natural numbers. E.g.:
  - F(R1) = 1, R1 may be a tape derive
  - F(R2) = 5, R2 may be a disk derive
  - F(R3) = 12, R3 may be a printer

Deadlock Prevention …

- **Circular Wait** …
- Now, two protocols can be considered to prevent deadlocks.
  - Each process can request resources only in an increasing order of enumeration. After a process requests for Ri, it can request instances of Rj if and only if F(Rj) > F(Ri).
  - A process requesting an instance of resource type Rj must have released any resources Ri such that F(Ri) ≥ F(Rj).
- If these two protocols are used, then the circular-wait condition cannot hold.

Deadlock Avoidance

- An alternative method for avoiding deadlocks is to require additional information about how resources are to be requested.
- Simplest and most useful model requires that each process declares the ***maximum number*** of resources of each type that it may need.
- A 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.

Deadlock Avoidance …

Safe State

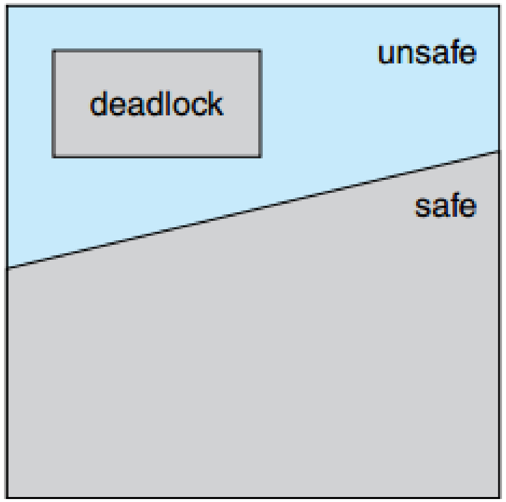
- When a process requests an available resource, system must decide if immediate allocation leaves the system in a safe state.
- A system is in a **safe state** if there exists a sequence <P_1, P_2, …, P_n> of all the processes in the system such that for each P_i, the resources that P_i can still request can be satisfied by currently available resources plus the resources held by all P_j, with $j < i$.
- That is:
  - If the resources that P_i needs are not immediately available, then P_i can wait until all P_j have finished.
  - When all P_j have finished, P_i can obtain all of its needed resources, execute, return allocated resources, and terminate.
  - When P_i terminates, P_{i+1} can obtain its needed resources, and so on.
- If no such sequence exists, then the system state is said to be *unsafe*.

Deadlock Avoidance ...

Safe State ...

❖ 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.



Deadlock Avoidance …

Avoidance Algorithms

- If there is single instance of a resource type, use a resource-allocation graph for deadlock avoidance.
- If there are multiple instances of a resource type, use the banker's algorithm.

Deadlock Avoidance …

Avoidance Algorithms …

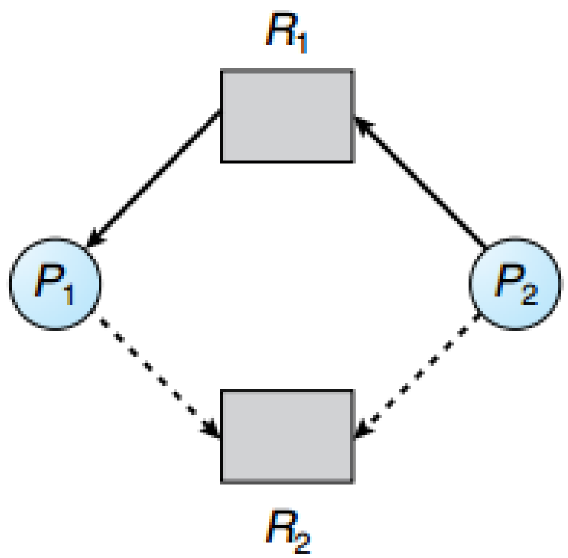
Resource-Allocation Graph Algorithm

- A new type of edge called a **claim edge,** a **dashed** line**,** is introduced to indicate that process P_j may request resource R_j.
- A claim edge is converted to a request edge when a process requests the resource.
- Request edge is converted to an assignment edge when the resource is allocated to the process.
- When a resource is released by a process, assignment edge is reconverted to a claim edge.

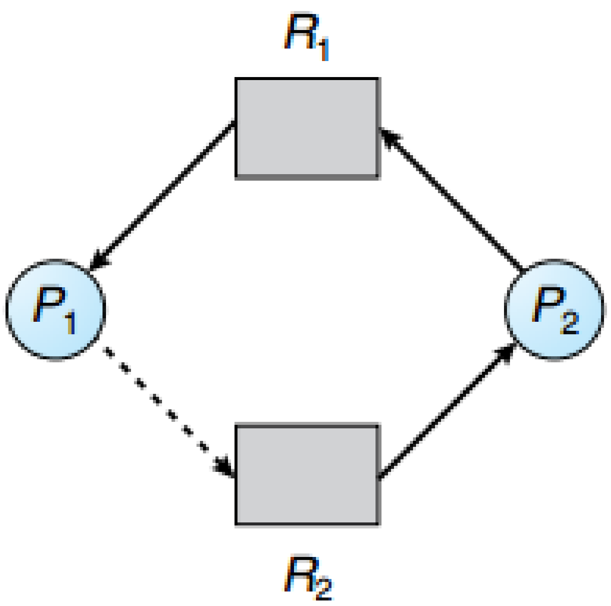
Deadlock Avoidance …

Avoidance Algorithms …

Resource-Allocation Graph Algorithm …



Resource-allocation graph for deadlock avoidance



An unsafe state in resource-allocation graph

Deadlock Avoidance ...

Avoidance Algorithms ...

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.

Deadlock Avoidance …

Avoidance Algorithms …

Banker's Algorithm

- It is used for resource types having multiple instances.
- A new process entering the system must declare the maximum number of instances of each resource type that it may need.
- This number may not exceed the total number of resources in the system.
- When a user requests a set of resources, the system must determine whether the allocation of these resources will leave the system in a safe state.
- If it will, the resources are allocated; otherwise, the process must wait until some other process releases enough resources.

Deadlock Avoidance ...

Avoidance Algorithms ...

Banker's Algorithm ...

Data Structures used for the Banker's Algorithm

Let n = number of processes, and m = number of resources types.

- **Available**: a vector of length m. If Available[j] = k, there are k available instances of resource type R_j.
- **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.
  - $Need[i, j] = Max[i, j] – Allocation[i, j]$

Deadlock Avoidance …

Avoidance Algorithms …

Banker's Algorithm …

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 index ***i*** such that both:

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

   (b) $\boldsymbol{Need_i \leq Work}$

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

3. $\boldsymbol{Work = Work + Allocation_i}$
   Finish*[*i*] = *true
   go to step 2

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

Deadlock Avoidance …

Avoidance Algorithms …

Banker's Algorithm …

Resource-Request Algorithm for Process P_i

Let $Request_i$ be the 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 an 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 have allocated the 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

Deadlock Avoidance ...

Avoidance Algorithms ...

Banker's Algorithm ...

Example

- 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*	*Need*	*Available*
	A B C	*A B C*	*A B C*	*A B C*
P_0	0 1 0	7 5 3	7 4 3	3 3 2
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	

Need* = *Max* – *Allocation

The system is in a safe state since the sequence < P_1, P_3, P_4, P_2, P_0> satisfies safety criteria.

Deadlock Avoidance …

Avoidance Algorithms …

Banker's Algorithm …

Example of Banker's Algorithm …

***Suppose 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 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? Unavailable Resource
- Can request for (0, 2, 0) by P_0 be granted? Resulted in unsafe state

Deadlock Detection

- If a system does not employ either a deadlock-prevention or a deadlock avoidance algorithm, then a deadlock situation may occur. In this environment, the system may provide:
  - An algorithm that examines the state of the system to determine whether a deadlock has occurred.
  - An algorithm to recover from the deadlock.

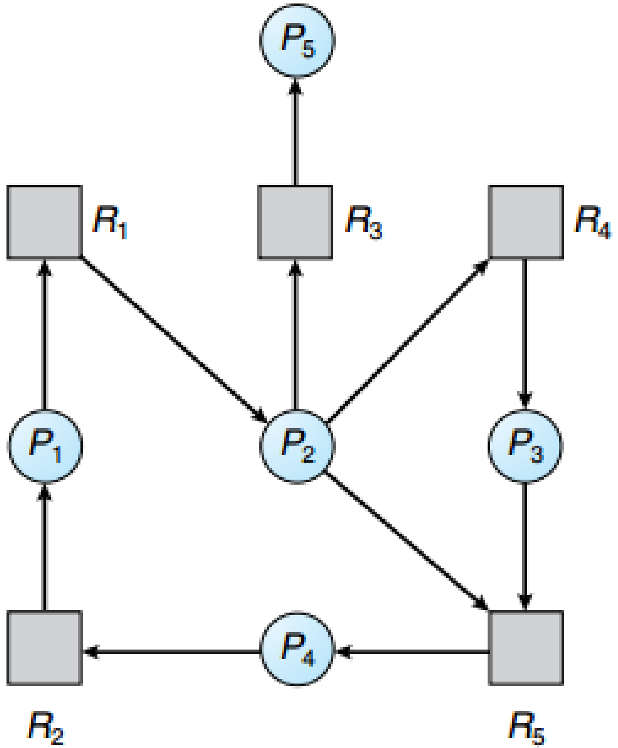
Deadlock Detection …

Single Instance of Each Resource Type

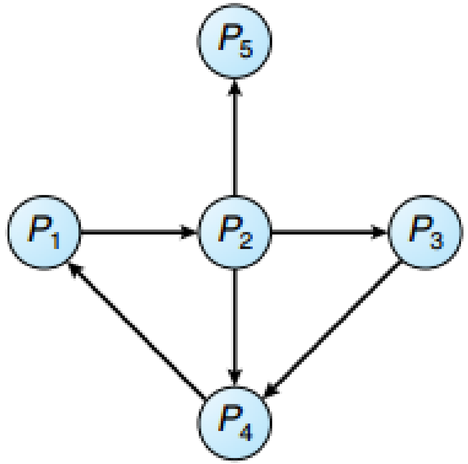
- A deadlock detection algorithm that uses a variant of the resource-allocation graph, called a **wait-for** graph, can be used for single instances.
- In a **wait-for** graph, nodes are processes where resources are not included.
  - $P_i \rightarrow P_j$ if P_i is waiting for P_j to release a resource that P_i needs.
- To detect deadlocks, the system needs to maintain the wait-for graph and periodically ***invoke an algorithm*** that searches for a cycle in the graph.
- 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.

Deadlock Detection …

Single Instance of Each Resource Type …



Resource-Allocation Graph



Corresponding wait-for graph

Deadlock Detection …

Several Instances of a Resource Type

- A deadlock detection algorithm applicable for systems with multiple instances of each resource type employs several time-varying data structures that are similar to those used in the banker's algorithm.
- **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.

Deadlock Detection …

Several Instances of a Resource Type …

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 a deadlock state. Moreover, if ***Finish*[*i*] == *false***, then P_i is deadlocked.

Deadlock Detection …

Several Instances of a Resource Type …

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.***

Deadlock Detection …

Several Instances of a Resource Type …

Example of Detection Algorithm …

- Suppose $\boldsymbol{P_2}$ requests an additional instance of type $\boldsymbol{C}$

Request

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

- State of system?
  - We can reclaim resources held by process $\boldsymbol{P_0}$, but insufficient to fulfill requests of other processes.
  - Deadlock exists, consisting of processes $\boldsymbol{P_1}$, $\boldsymbol{P_2}$, $\boldsymbol{P_3}$, and $\boldsymbol{P_4}$

Deadlock Detection …

Detection-Algorithm Usage

- When, and how often, to invoke the detection algorithm depends on:
  - How often a deadlock is likely to occur?
  - How many processes will be affected by deadlock when it happens?
- In the extreme, we can invoke the deadlock detection algorithm every time a request for allocation cannot be granted immediately.
- In this case, we can identify not only the deadlocked set of processes but also the specific process that “caused” the deadlock.

Recovery from Deadlock

- When a detection algorithm determines that a deadlock exists, several alternatives are available.
  - Inform the operator that a deadlock has occurred and let the operator deal with the deadlock manually.
  - Let the system recover from the deadlock automatically.
- There are two options for breaking a deadlock.
  - Abort one or more processes to break the circular wait.
  - Preempt some resources from one or more of the deadlocked processes.

Recovery from Deadlock ...

Process Termination

- We can use one of the two methods to abort processes.
  - Abort all deadlocked processes.
  - Abort one process at a time until the deadlock cycle is eliminated.
- Many factors may affect which process is chosen for abortion:
  1. What is the priority of the process
  2. How long process has computed, and how much longer the process will compute before completion.
  3. How many and what types of resources the process has used
  4. How many more resources the process needs to complete
  5. How many processes will need to be terminated
  6. Is process interactive or batch

Recovery from Deadlock ...

Resource Preemption

- We *successively* preempt some resources from processes and give them to other processes until the deadlock cycle is broken.
- If preemption is required to deal with deadlocks, then ***three*** issues need to be addressed:
  - **Selecting a victim**: determine order of preemption to minimize cost.
  - **Rollback**: return the process of which the resource is preempted to some safe state and restart it from that safe state.
  - **Starvation**: the same process may always be picked as victim. To solve this, include the number of rollbacks in the cost factor.

Reference: Silberschatz et al., Operating System Concepts, Ninth Edition, 2013.

Questions???