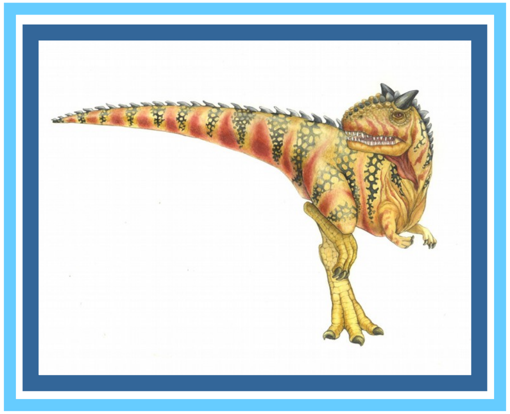
Chapter 8: Deadlocks



Chapter 8: Deadlocks

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

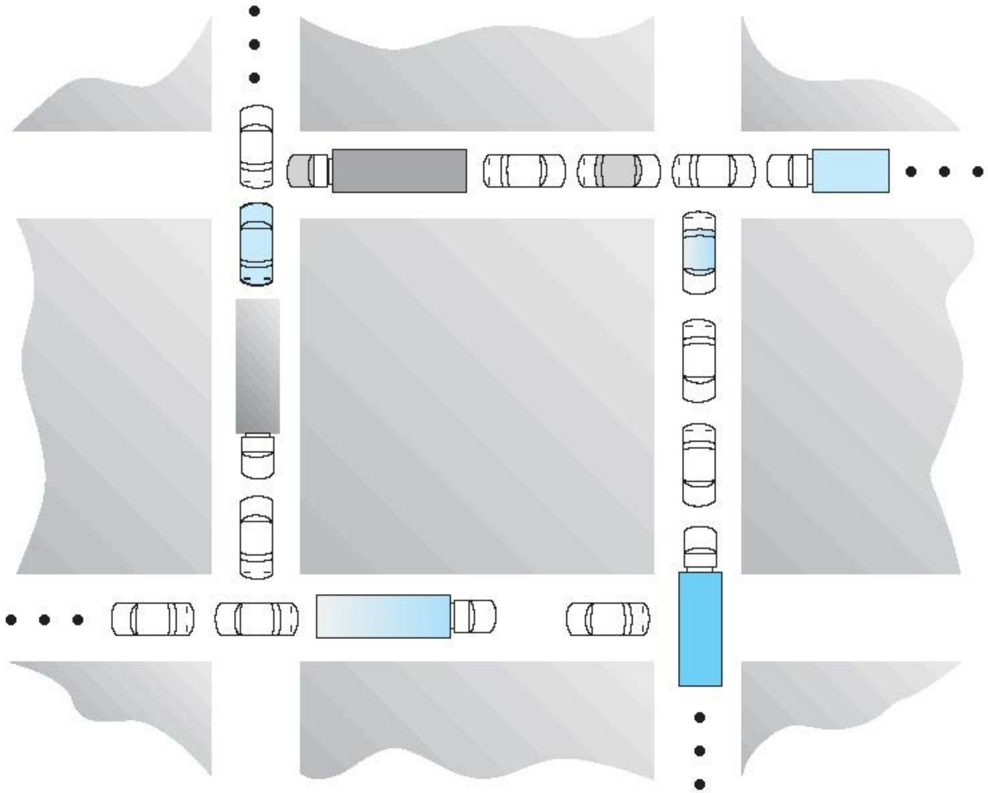
Chapter Objectives

- 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

Deadlock Example – One Way Bridge

- Draw a one way bridge
- Can a "far away" can know that it is involved in a deadloc

Deadlock Example – Traffic Gridlock



System Model

- System consists of resources
- Resource types $R_1, R_2, . . ., R_m$

  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 Example with Mutex locks

- Two mutex locks are created in the following code
  - pthread_mutex_t   first_mutex;
  - pthread_mutex_t   second_mutex;

- The two mutex locks are initialized in the following code
  - pthread_mutex_init (&first_mutex, NULL);
  - pthread_mutex_init(&second_mutex, NULL);

- Two threads-- thread_one and thread_two are created, and both these threads have access to both mutex locks.

Deadlock Example with Mutex locks (Cont.)

```
/* thread one runs in this function */
void *do_work_one(void *param)
{
   pthread_mutex_lock(&first_mutex);
   pthread_mutex_lock(&second_mutex);
   /** * 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);
   pthread_mutex_lock(&first_mutex);
   /** * Do some work */
   pthread_mutex_unlock(&first_mutex);
   pthread_mutex_unlock(&second_mutex);
   pthread_exit(0);
}
```

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.

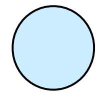
Resource-Allocation Graph

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

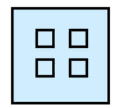
- V is partitioned into two types:
  - $P = \{P_1, P_2, \ldots, P_n\}$, the set consisting of all the processes in the system
  - $R = \{R_1, R_2, \ldots, R_m\}$, the set consisting of all resource types in the system
- **request edge** – directed edge $P_i \rightarrow R_j$
- **assignment edge** – directed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

Resource-Allocation Graph (Cont.)

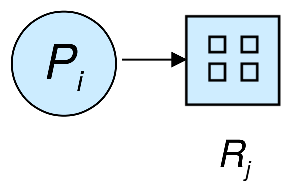
- Process



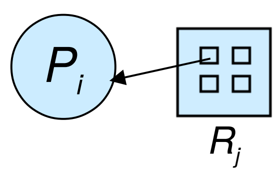
- Resource Type with 4 instances



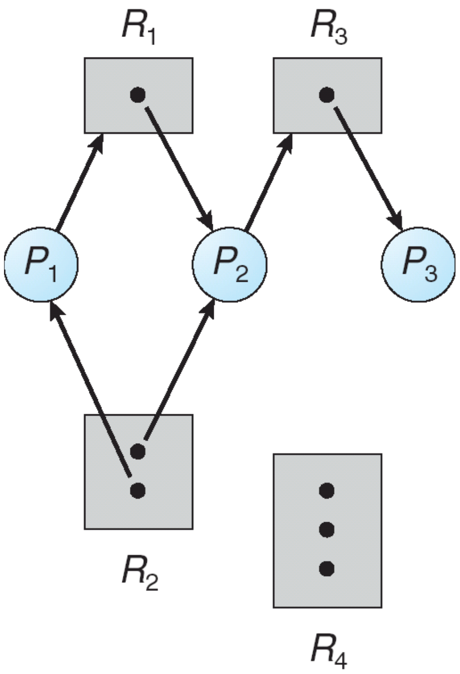
- P_i requests instance of R_j



- P_i is holding an instance of R_j



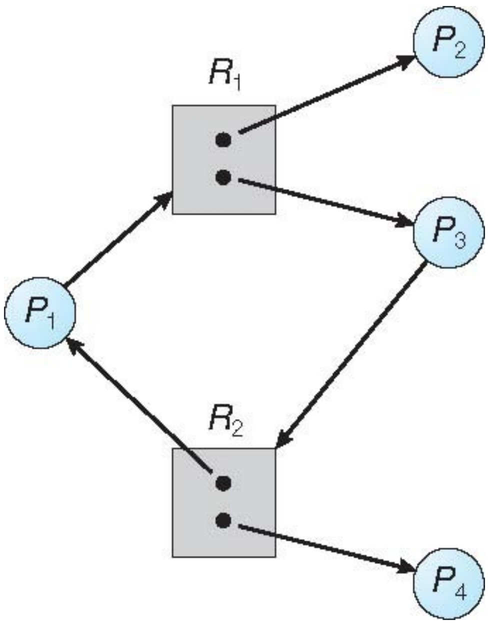
Example of a Resource Allocation Graph



Basic Facts

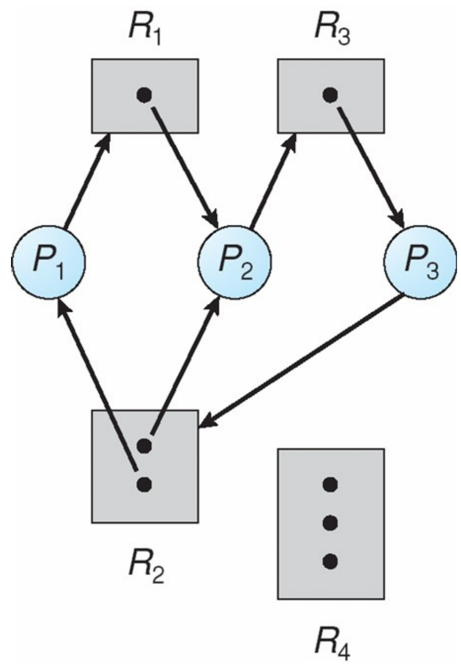
- If graph contains no cycles $\Rightarrow$ no deadlock
- If graph contains a cycle $\Rightarrow$
  - If only one instance per resource type, then deadlock exist
  - If several instances per resource type, then possibility of deadlock

Resource Allocation Graph With a Cycle



Is there a deadlock?

Resource Allocation Graph With a Cycle



Is there a deadlock?

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

Deadlock Prevention

- Ensure that at least one of the necessary condition for deadlocks does not hold. Can be accomplished restraining the ways request can be made
- **Mutual Exclusion** – Must hold for non-sharable resources that can be accessed simultaneously by various processes. Therefore cannot be used for 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 allocated to it.
  - Low resource utilization; starvation possible

Deadlock Prevention (Cont.)

- **No Preemption** – not practical for most systems
  - If a process A that is holding some resources requests another resource that cannot be immediately allocated to it, then all resources currently being held by A are released
  - Preempted resources are added to the list of resources for which the process is waiting
  - Process will be restarted only when it can regain its old resources, as well as the new ones that it is requesting
- **Circular Wait** – impose a total ordering of all resource types, and require that each process requests resources in an increasing order of enumeration. Can be used in practice.

Deadlock Avoidance

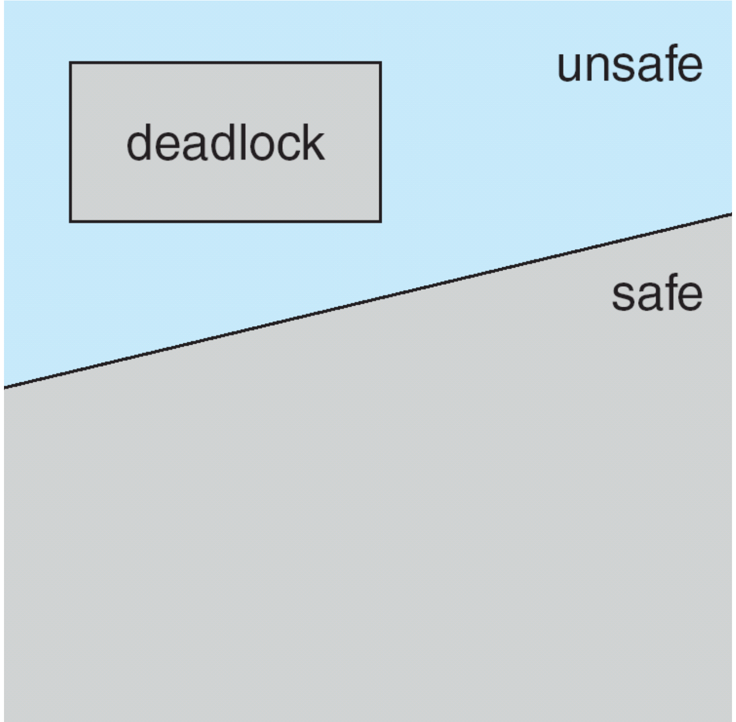
- Ensure that the system will ***never*** enter a deadlock state
- Requires that the system to have some additional ***a priori*** information available on possible resource requests.
  - Simplest and most useful model requires that each process declare the ***maximum number*** of resources of each type 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

- System is in **safe state** if there exists a sequence $\langle P_1, P_2, \ldots, P_n \rangle$ 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 processes P_j ($j < i$) have finished executing.
  - When they have finished executing they release all their resources and then P_i can obtain the 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



Deadlock Avoidance Algorithms

- Deadlock avoidance $\Rightarrow$ ensure that a system will never enter an unsafe state.
- Single instance of a resource type
  - Use a variant of the resource-allocation graph
- Multiple instances of a resource type
  - Use the banker's algorithm

Resource-Allocation Graph Scheme

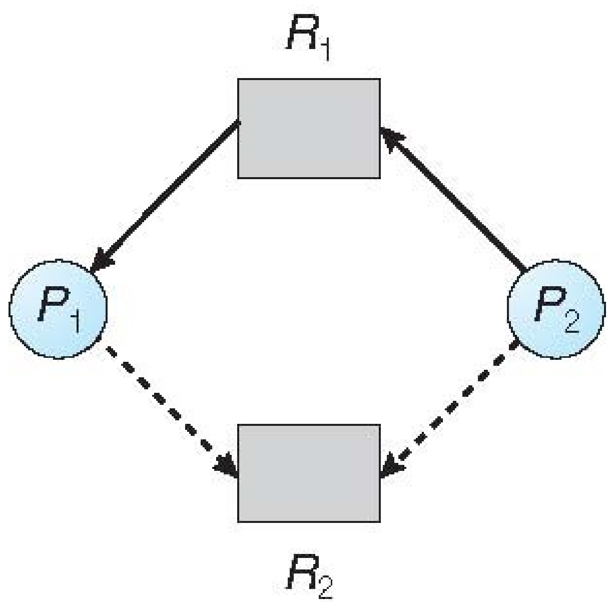
- Single instance of a resource type
- Each process must a priori claim maximum resource use
- Use a variant of the resource-allocation graph with claim edges.
- **Claim edge** $P_i \rightarrow R_j$ indicated that process P_j may request resource R_j; represented by a dashed line
- Claim edge converts 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
- A cycle in the graph implies that the system is in unsafe state

Resource-Allocation Graph Scheme

- Single instance of a resource type
- Each process must a priori claim maximum resource use
- Use a variant of the resource-allocation graph with claim edges.
- **Claim edge** P_i ----→ 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
- A cycle in the graph implies that the system is in unsafe state

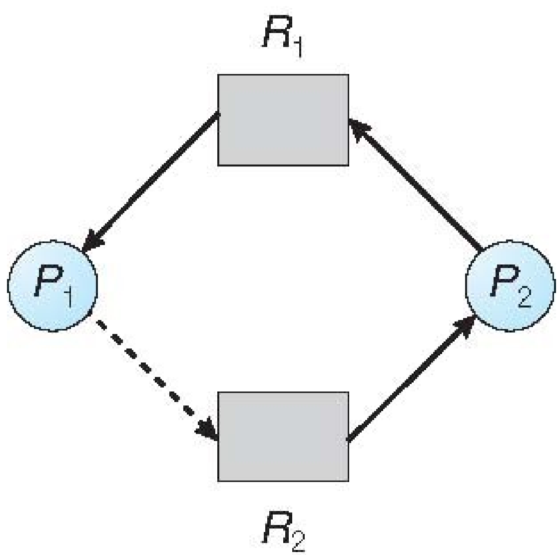
Resource-Allocation Graph with claim edges

- P1 is holding resource R1 and has a claim on R2
- P2 is requesting R1 and has a claim on R2
- No cycle. So system is in a safe state.



Example of a Resource Allocation Graph

- The graph of slide 8.25 with a claim edge from P2 to R2 is changing to an assignment edge.
- There is a cycle in the graph ➔ unsafe state.
- Is there a deadlock?



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
- Otherwise, the process must wait

Banker's Algorithm

- Multiple instances of a resource type
- 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.
- Think of an interest-free bank where:
  - A customer establishes a line of credit.
  - Borrows money in chunks that together never exceed the total line of credit.
  - Once it reaches the maximum, the customer must pay back 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, then 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 at most 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) $\boldsymbol{Need_i \leq Work}$

    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. Otherwise, in an unsafe state.

Example of Safety Algorithm

- 5 processes -- P_0 ….. 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	

Example of Safety Algorithm (Cont.)

- The content of the matrix ***Need*** is defined to be ***Max – Allocation***

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

Resource-Request Algorithm for Process P_i

Let $\textbf{\textit{Request}}_i$ [...] be the request vector for process $\boldsymbol{P_i}$.

$\textbf{\textit{Request}}_i\,[j] = \boldsymbol{k}$. Process $\boldsymbol{P_i}$ wants $\boldsymbol{k}$ instances of resource type $\boldsymbol{R_j}$

1. If $\textbf{\textit{Request}}_i \le \textbf{\textit{Need}}_i$ go to step 2. Otherwise, raise error condition, since process has exceeded its maximum claim
2. If $\textbf{\textit{Request}}_i \le \textbf{\textit{Available}}$, go to step 3. Otherwise $\boldsymbol{P_i}$ must wait, since resources are not available
3. Pretend to allocate requested resources to $\boldsymbol{P_i}$ by modifying the state as follows:

$$\textbf{\textit{Available}} = \textbf{\textit{Available}} - \textbf{\textit{Request}}_i$$

$$\textbf{\textit{Allocation}}_i = \textbf{\textit{Allocation}}_i + \textbf{\textit{Request}}_i$$

$$\textbf{\textit{Need}}_i = \textbf{\textit{Need}}_i - \textbf{\textit{Request}}_i$$

- If safe $\Rightarrow$ the resources are allocated to $\boldsymbol{P_i}$
- If unsafe $\Rightarrow$ $\boldsymbol{P_i}$ 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* (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	

- We have shown that the system is in a safe state

Example: P_1 Request (1,0,2)

- Check that Request ≤ Available (that is, (1,0,2) ≤ (3,3,2) ⇒ true
- State of system after resources allocated to $\boldsymbol{P_1}$

	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	

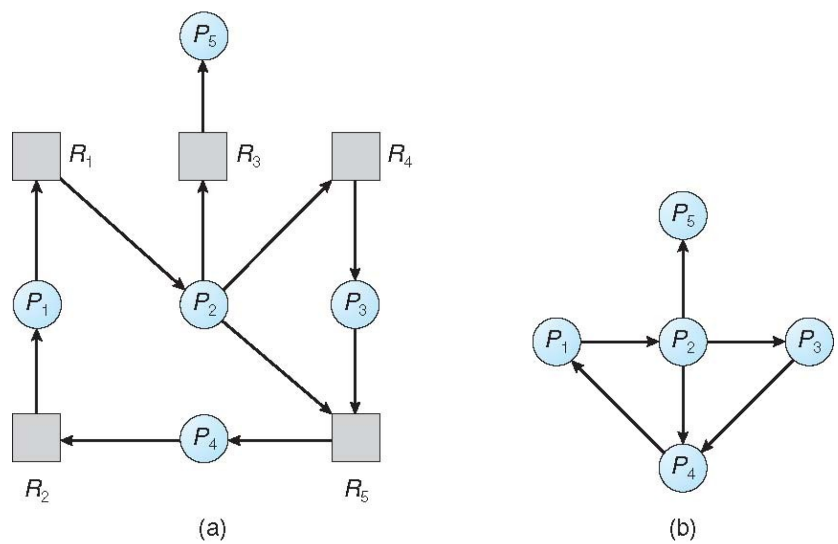
- Executing safety algorithm shows that sequence < $\boldsymbol{P_1, P_3, P_4, P_0, P_2}$> satisfies safety requirement
- Given the above state -- can request for (3,3,0) by $\boldsymbol{P_4}$ be granted?
- Given the above state -- can request for (0,2,0) by $\boldsymbol{P_0}$ be granted?

Methods for Handling Deadlocks: Detection

- Allow system to enter deadlock state
- Detection algorithm
  - Single instance of a resource type
  - Multiple instances of a resource type.
- Recovery scheme

Single Instance of Each Resource Type

- Maintain a **wait-for** graph
  - Nodes are processes
  - $P_i \rightarrow P_j$ if P_i is waiting for P_j
- Converting a resource-allocation graph to a wait-for graph.



Resource-Allocation Graph Corresponding wait-for graph

Detection Algorithm for Single Instance

- Maintain a **wait-for** graph
- 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

Several Instances of a Resource Type

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

- **Available**: Vector of length m. If available [j] = k, then there are k instances of resource type R_j available

- **Allocation**: $n \times m$ matrix. If Allocation[i,j] = k, then P_i is currently allocated k instances of R_j

- **Request**: $n \times m$ matrix that indicates the current request of each process. If Request[i,j] = k, then process P_i is requesting k additional instances of resource type R_j.

Detection Algorithm

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

1. Initialization

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

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

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

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

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

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

3. $\boldsymbol{Work} = \boldsymbol{Work} + \boldsymbol{Allocation_i}$
   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

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 $\langle P_0, P_2, P_3, P_1, P_4 \rangle$ will result in ***Finish[i] = true*** for all ***i***

Example of Detection Algorithm (Cont.)

- P_2 requests one 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?
  - 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

- If a deadlock is detected we must abort (rollback) some of the processes involved in the deadlock (see next slide)
- Need to decide when, and how often, to invoke the dedalock detection algorithm, which 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.

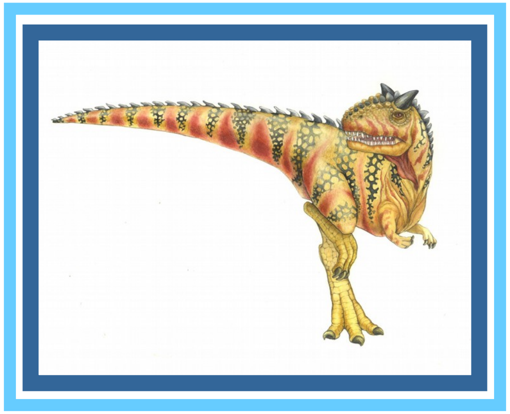
Recovery from Deadlock: Process Termination

- Abort all deadlocked processes
- Abort one process at a time until the deadlock cycle is eliminated
- In which order should we choose to abort?
  - 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

End of Chapter 8



Deadlock Example with Lock Ordering

- Two bank transactions 1 and 2 execute concurrently.
  - Transaction 1 transfers $25 from account A to account B
  - Transaction 2 transfers $50 from account B to account A
- Program

```
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);
}
```