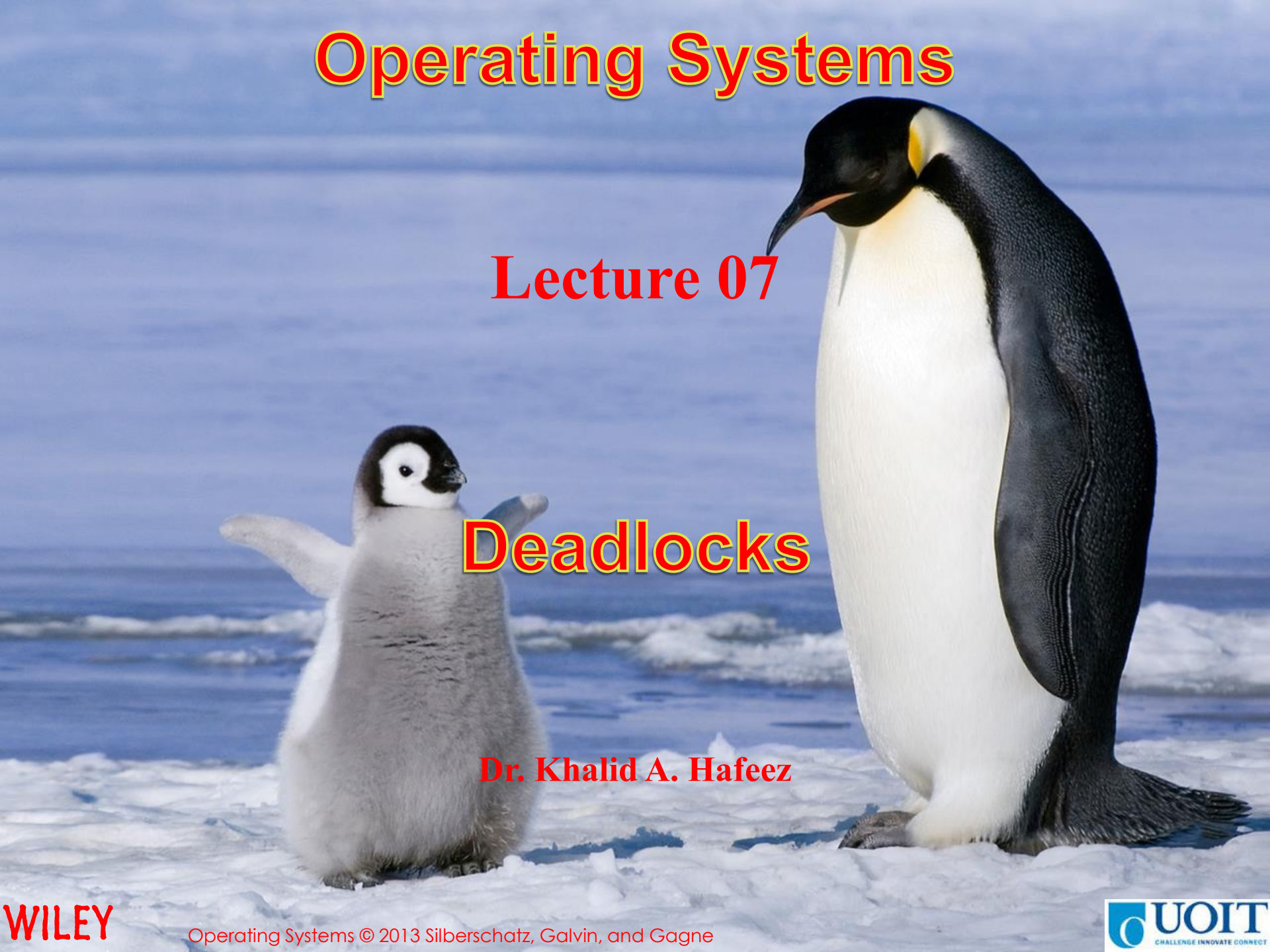
Operating Systems

Lecture 07

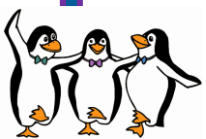
Deadlocks

Dr. Khalid A. Hafeez



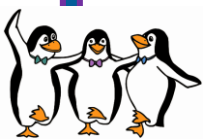
Deadlocks

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



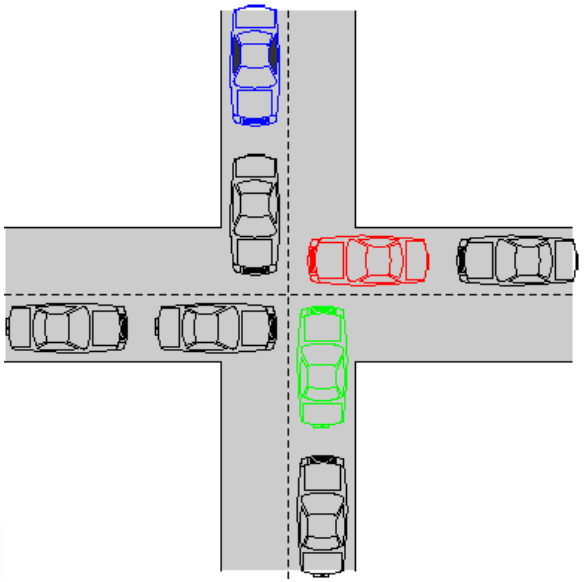
Objectives

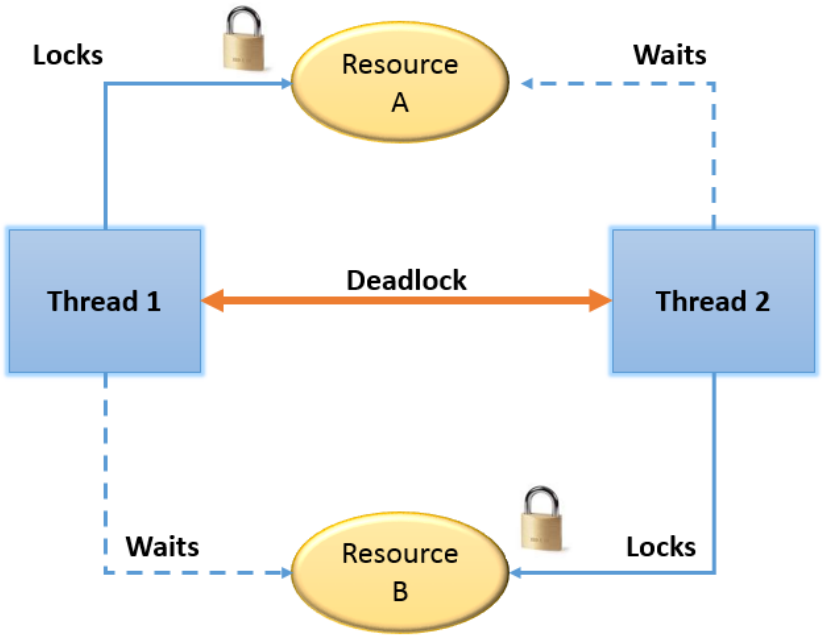
- To develop a description of deadlocks, which prevent sets of concurrent processes from completing their tasks
- To present a number of different methods for preventing or avoiding deadlocks in a computer system



Introduction

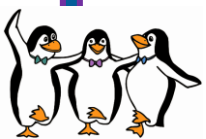
- Deadlock: a waiting process is never again able to change state, because the resources it has requested are held by other waiting processes.
  - What methods an OS can use to prevent or deal with deadlocks.
    - Deadlock problems are very common, given current trends, including larger numbers of processes, multithreaded programs, many more resources within a system, and an emphasis on long-lived file and database servers rather than batch systems.





System Model

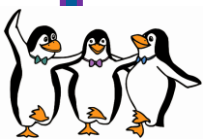
- **A system consists of a finite number of resources to be distributed among a number of competing processes.**
  - Resource types R_1, R_2, . . ., R_m
    - *CPU cycles, memory space, I/O devices, …*
  - Each resource type R_i has W_i instances.
    - a system may have two or more printers, …
- Each process utilizes a resource in the following sequence:
  - Request: If the request cannot be granted immediately, then the requesting process must wait until it can acquire the resource.
  - use
  - Release
- The resources may be :
  - Physical resources (printers, tape drives, memory space, and CPU cycles,)
  - Logical resources (semaphores, mutex locks, and files, …)



Deadlock Characterization

- Deadlock can arise if four conditions hold **simultaneously** in a system:

1. **Mutual exclusion:** only one process at a time can use the resource, other processes requesting the same resource must wait.
2. **Hold and wait:** a process holding at least one resource is waiting to acquire additional resources held by other processes
3. **No preemption:** Resources cannot be preempted; a resource can be released only voluntarily by the process holding it, after that process has completed its task
4. **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.



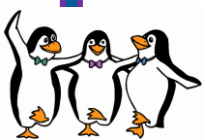
Deadlock Characterization

- Example of deadlock with mutex locks:
  - There is a possibility of a deadlock, depends on when the threads been scheduled:

```
/* Create and initialize the mutex locks */
Pthread_mutex_t first_mutex;
Pthread_mutex_t second_mutex;
Pthread_mutex_init(&first_mutex, NULL);
Pthread_mutex_init(&second_mutex, NULL);

/* 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);
}
```

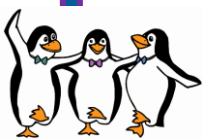


Deadlock Characterization

- Example of deadlock for memory requests:
  - Space is available for allocation of 200Kbytes, and the following sequence of events occur:
  - Deadlock occurs if both processes progress to their second request

P1
. . .
:
Request 80 Kbytes;
:
Request 60 Kbytes;

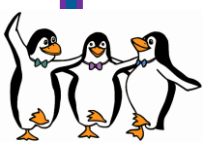
P2
. . .
:
Request 70 Kbytes;
:
Request 80 Kbytes;



Deadlock Characterization

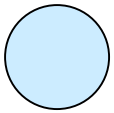
Resource-Allocation Graph

- Deadlocks can be described by a directed graph called a system resource-allocation graph
  - It consists of a set of vertices ***V*** and a set of edges ***E***.
- **The set of vertices V is partitioned into two types:**
  - $P = \{P_1, P_2, \ldots, P_n\}$, the set consisting of all active processes in the system
  - $R = \{R_1, R_2, \ldots, R_m\}$, the set consisting of all resource types in the system
- **Request edge**: it is a directed edge $P_i \rightarrow R_j$
  - A process P_i has requested an instance of resource type R_j and is currently waiting for that resource.
- **Assignment edge**: it is directed edge $R_j \rightarrow P_i$
  - An instance of resource type R_j has been allocated to process P_i

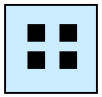


Deadlock Characterization

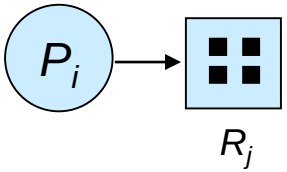
- Resource-Allocation Graph
  - A process (P_i) is represented as a circle



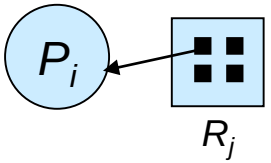
  - A resource R_j is represented as a rectangle
    - Resource Type with 4 instances



  - P_i requests instance of R_j



  - P_i is holding an instance of R_j

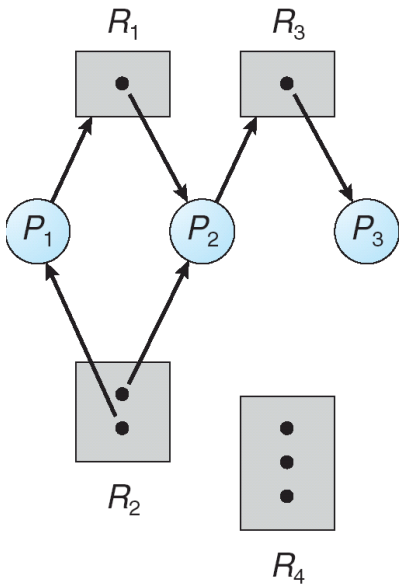


- When process Pi requests an instance of resource type Rj , a request edge is inserted in the resource-allocation graph.
  - When the request is fulfilled, the edge is instantaneously transformed to an assignment edge.
  - When the process releases the resource, the assignment edge is deleted.

Deadlock Characterization

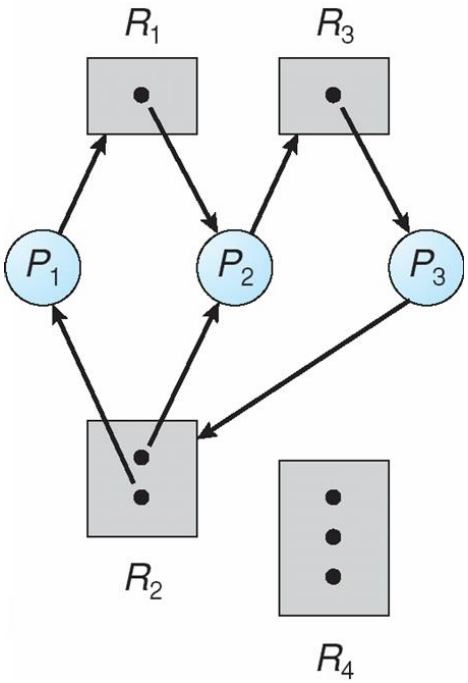
Resource-Allocation Graph

- Example of a Resource Allocation Graph
- The resource-allocation graph depicts the following situation.
  - The sets P, R, and E:
    - P = {P1, P2, P3}
    - R = {R1, R2, R3, R4}
    - E = {P1 → R1, P2 → R3, R1 → P2, R2 → P2, R2 → P1, R3 → P3}
  - Resource instances:
    - One instance of resource type R1
    - Two instances of resource type R2
    - One instance of resource type R3
    - Three instances of resource type R4
  - Process states:
    - Process P1 is holding an instance of resource type R2 and is waiting for an instance of resource type R1.
    - Process P2 is holding an instance of R1 and an instance of R2 and is waiting for an instance of R3.
    - Process P3 is holding an instance of R3.



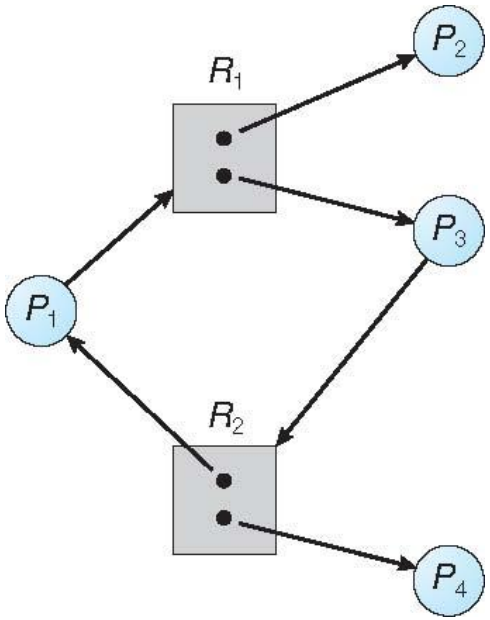
Deadlock Characterization

- Resource-Allocation Graph
  - Resource Allocation Graph With A Deadlock
    - If each resource type has exactly one instance, then a cycle implies that a deadlock has occurred
    - Suppose that process P3 requests an instance of resource type R2.
    - Since no resource instance is currently available, we add a request edge P3→ R2 to the graph.
    - At this point, two minimal cycles exist in the system:
      - P1 → R1 → P2 → R3 → P3 → R2 → P1
      - P2 → R3 → P3 → R2 → P2
    - Processes P1, P2, and P3 are **deadlocked**



Deadlock Characterization

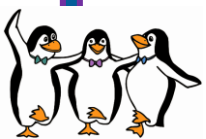
- Resource-Allocation Graph
  - Graph With A Cycle But No Deadlock
    - We also have a cycle: P1 → R1 → P3 → R2 → P1
    - Observe that process P4 may release its instance of resource type R2.
      - That resource can then be allocated to P3, breaking the cycle



    - *If there is a cycle, then the system **may** or **may not** be in a deadlocked state.*

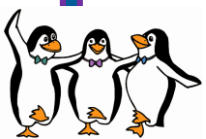
Deadlock Characterization

- Basic Facts
  - If the graph contains no cycles $\Rightarrow$ no deadlock
  - If the graph contains a **cycle** $\Rightarrow$
    - if only one instance per resource type is available, then there is a **deadlock**
    - if several instances per resource type are available, there is a **possibility** of deadlock



Methods for Handling Deadlocks

- We can deal with the deadlock problem in one of three ways:
  1. Use a protocol to ensure that the system will ***never*** enter a deadlock state by using:
     - **Deadlock prevention scheme**
       - Adopt a policy that eliminates one of the necessary conditions by constraining how requests for resources can be made
     - **Deadlock avoidance scheme**
       - Make the appropriate dynamic choices based on the current state of resource allocation
         - » The system must consider the resources that currently available, allocated, and the future requests and releases of each process.
  2. Allow the system to enter a deadlock state and then recover
  3. Ignore the problem and pretend that deadlocks never occur in the system;
     - Used by most operating systems, including Linux and Windows
       - It is up to the application developer to write programs that handle deadlocks.

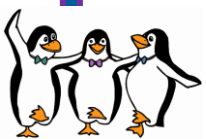


Deadlock Prevention

- For a deadlock to occur, each of the four necessary conditions must hold.

- By ensuring that at least one of these conditions cannot hold, we can prevent the occurrence of a deadlock

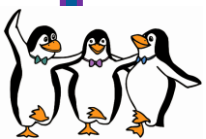
  1. **Mutual Exclusion**
     - It is not required for sharable resources (e.g., read-only files); thus cannot be involved in a deadlock
     - It must hold for non-sharable resources (e.g. mutex locks)



Deadlock Prevention

2. Hold and Wait

- To ensure that the hold-and-wait condition never occurs in the system:
  - We must guarantee that whenever a process requests a resource, it does not hold any other resources. Two protocols are possible:
    1. First protocol requires from a process to request and be allocated all its resources before it begins execution. It can be implemented as:
       - » Let the system calls requesting resources for a process precede all other system calls.
    2. Second protocol allows a process to request resources only when it releases currently allocated resources.
  - **Both protocols have disadvantages:**
    - Low resource utilization: resources may be allocated but unused for a long period
    - Starvation is possible: a process that needs several popular resources may have to wait indefinitely

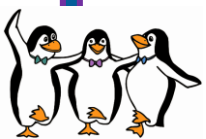


Deadlock Prevention

3. No Preemption

- **A protocol that can be used to ensure that this condition never holds:**
  - 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

- This protocol can be used with resources whose state can be easily saved and restored later, such as CPU registers and memory space.
- It cannot be applied to such resources as mutex locks and semaphores.

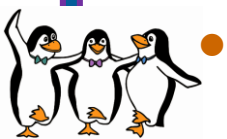


Deadlock Prevention

4. Circular Wait

- **A protocol that can be used to ensure that this condition never holds:**
  - Impose a total ordering of all resource types, and require that each process requests resources in an increasing order of enumeration
    - **Example**: If the set of resource types R includes tape drives, disk drives, and printers, then the function **F** might be defined as follows:

      F(tape drive) = 1
      F(disk drive) = 5
      F(printer) = 12
  - If a process has the recource Ri, it can request instances of a resource type Rj iff 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 several instances of the same resource type are needed, a single request for all of them must be issued.
- A programmer should write programs that follow the correct ordering to prevent deadlock.
- The function **F** should be defined according to the normal order of usage of the resources in a system.



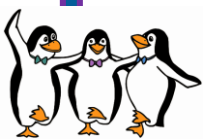
Deadlock Prevention

- **Deadlock Example**

```
/* 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);
}
```

If the lock ordering in this Pthread program is:
F(first mutex) = 1
F(second mutex) = 5
then thread two could not request the locks out of order.

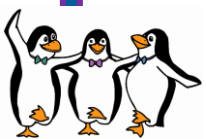


- A software tool (called **witness**) used to verify that locks are acquired in the proper order: works on FreeBSD. It generates a warning message on the console.

Deadlock Prevention

Deadlock Example with Lock Ordering

- Note that imposing a lock ordering does not guarantee deadlock prevention if locks can be acquired dynamically.
- Example,
  - Assume that we have a function that transfers funds between two accounts.
  - To prevent a race condition, each account has an associated mutex lock that is obtained from a get_lock() function



Deadlock Prevention

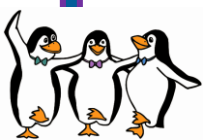
- **Deadlock Example with Lock Ordering**

```
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);
}
```

- Deadlock is possible if two threads simultaneously invoke the transaction() function, transposing different accounts:
  - Transaction 1 transfers $25 from account A to account B:

    transaction(checking_account, savings_account, 25);
  - Transaction 2 transfers $50 from account B to account A

    transaction(savings_account, checking_account, 50);

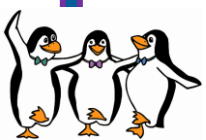


Deadlock Prevention

Deadlock Example with Lock Ordering

- To solve this problem. add a new lock to this function.
  - This third lock must be acquired before the two locks associated with the accounts are acquired.
  - The transaction() function now appears as follows:

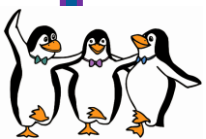
```
void transaction(Account from, Account to, double amount){
   mutex lock1, lock2 , lock3;
   acquire(lock3);
   lock1 = get_lock(from);
   lock2 = get_lock(to);
   acquire(lock1);
      acquire(lock2);
         withdraw(from, amount);
         deposit(to, amount);
      release(lock3);
      release(lock2);
   release(lock1);
}
```



Deadlock Avoidance

- Avoiding deadlocks requires that the system has some additional a ***priori*** information available
  - To satisfy a request, the system should consider the resources currently available, the resources currently allocated to each process, and the future requests and releases of each process.

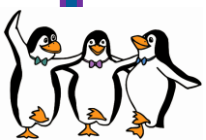
- **Many algorithms use this approach:**
  - The simplest and most useful model requires that each process **declares** the ***maximum number*** of resources of each type that it may need
  - The deadlock-avoidance algorithm dynamically examines the resource-allocation state to ensure that there can never be a circular-wait condition
    - **Resource-allocation *state*:** the number of available and allocated resources, and the maximum demands of the processes.
- **Two deadlock-avoidance algorithms:**
    - Safe State
    - Resource-Allocation-Graph Algorithm



Deadlock Avoidance

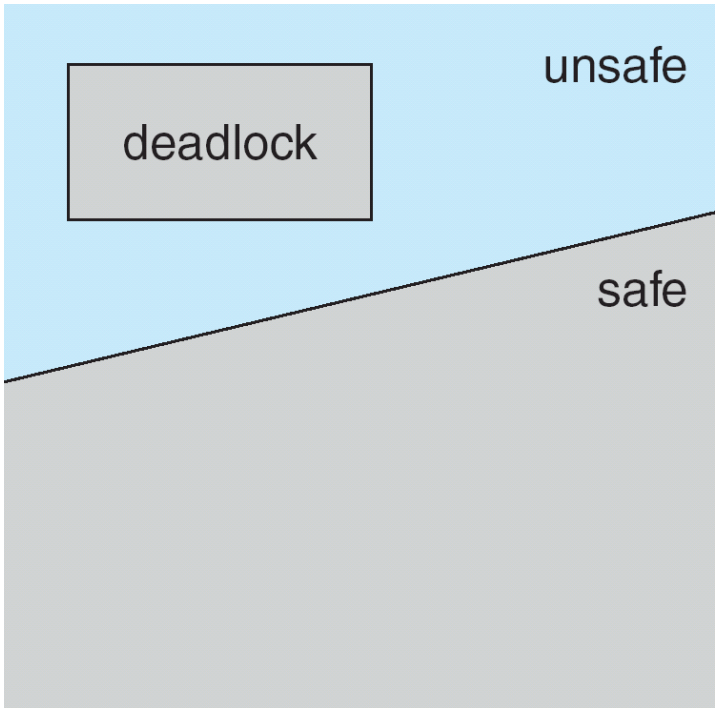
Safe State

- A state is ***safe*** if the system can allocate resources to each process (up to its maximum) in some order and still avoid a deadlock
- 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 **safe sequence** $<P_1, P_2, \ldots, P_n>$ 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 the currently available resources plus the resources held by all 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
- If no such sequence exists, then the system state is said to be ***unsafe***.
  - An unsafe state **may** lead to a deadlock.



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.

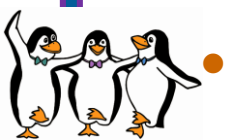


Deadlock Avoidance

- Safe State: Example:
  - Consider a system with **12** tape drives and three processes: P0, P1, and P2.
  - Suppose that, at time $\mathbf{t_0}$, the current needs are:

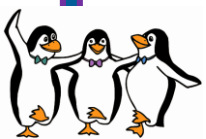
	Maximum Needs	Current Needs
P0	10	5
P1	4	2
P2	9	2

  - At time $\mathbf{t_0}$, the system is in a safe state. (the remaining is 3 tape drives):
    - The sequence <P1, P0, P2> satisfies the safety condition.
      - P1 can immediately be allocated 2 tape drives, then return all of them (5 now available)
      - P0 can get its remaining 5 tape drives and return them all (10 now available);
      - Finally P2 can get its remaining 7 tape drives and return them all (12 now available).
  - Suppose that, at time $\mathbf{t_1}$, process P2 requests and is allocated one more tape drive. The system is no longer in a safe state:
    - Only P1 can be allocated all its tape drives. When it returns them (4 now available).
    - Since P0 is allocated 5 tape drives but has a maximum of 10, it may request 5 more, similarly P2 may request 6 tape drives, resulting in a deadlock.
  - If we let P2 waits until either P0 or P1 had finished and released its resources, then the deadlock can be avoided.



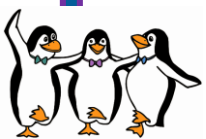
Deadlock Avoidance

- Safe State
  - Avoidance algorithms to ensure that the system will never deadlock
    - Initially, assume that the system is in a safe state.
    - Whenever a process requests a resource that is currently available, the request is granted only if the allocation leaves the system in a safe state.
    - Disadvantage:
      - A process may have to wait even if the resource is currently available. This results in lower resource utilization



Deadlock Avoidance

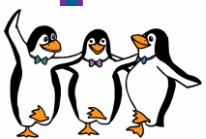
- Avoidance algorithms to ensure that the system will never deadlock
  - Single instance of a resource type
    - Use a variant of the resource-allocation graph algorithm
  - Multiple instances of a resource type
    - Use the banker's algorithm



Deadlock Avoidance

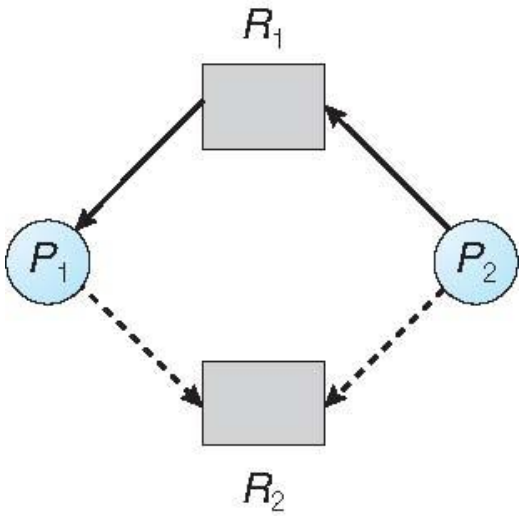
- Resource-Allocation Graph Scheme
  - In addition to request and assignment edges, we define a new edge:
    - **Claim edge** $P_i \rightarrow R_j$ means process P_j may request resource R_j in the future; represented by a dashed line -----→
  - Claim edge will be converted to request edge ($\rightarrow$) when a process requests the resource
  - Request edge is converted to an assignment edge when the resource is allocated to the process.
  - When the resource is released by a process, assignment edge reconverts to a claim edge
  - Resources must be claimed *a priori* in the system

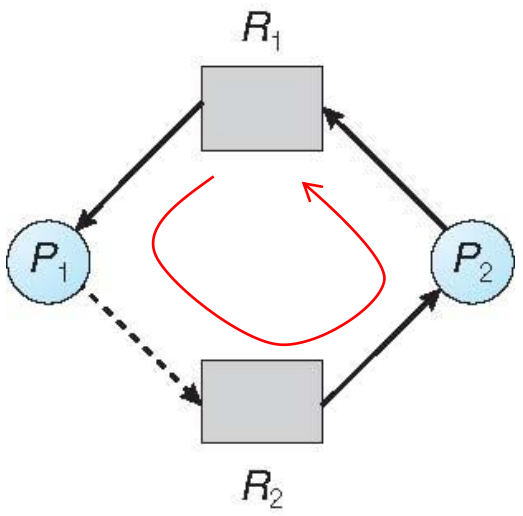
  - **Resource-Allocation Graph Algorithm**
    - Suppose that process P_i requests a resource R_j
    - The request can be granted **only** if converting the request edge to an assignment edge does not result in the formation of a cycle in the resource allocation graph



Deadlock Avoidance

- Resource-Allocation Graph Scheme
  - **Example:**
    - Consider this resource-allocation graph, suppose that P2 requests R2.
      - Although R2 is currently free, we cannot allocate it to P2, since this action will create a cycle in the graph.
      - A cycle, indicates that the system is in an unsafe state.
        - » If P1 requests R2, and P2 requests R1, then a deadlock will occur



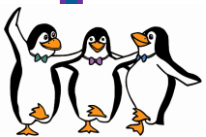


Unsafe State In Resource-Allocation Graph

Deadlock Avoidance

Banker's Algorithm

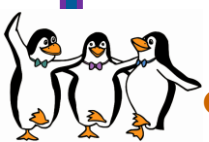
- It is named by Banker's algorithm because it can be used in a banking system to ensure that the bank never allocated its available cash in such a way that it could no longer satisfy the needs of all its customers.
- It works with multiple instances of each resource
- Each process must a ***priori*** claim the maximum number of instances from each resource type that it may need
- When a process requests a resource it may have to wait if it will put the system in unsafe state
- When a process gets all its resources it must return them in a finite amount of time



Deadlock Avoidance

Banker's Algorithm

- **Many data Structures are needed for the Banker's Algorithm:**
  - Let n = number of processes, and m = number of resources types.
- **Available**: a vector of length ***m*** indicates the number of available resources of each type.
  - If available[j] = k, then k instances of resource type R_j are available
- **Max**: an ***n x m*** matrix defines the maximum demand of each process.
  - If $Max\ [i,j] = k$, then process P_i may request at most k instances of resource type R_j
- **Allocation**: *an* ***n x m*** matrix defines the number of resources of each type currently allocated to each process.
  - If $Allocation[i,j] = k$ then P_i is currently allocated k instances of R_j
- **Need**: *an* ***n x m*** matrix indicates the remaining resource need of each process.
  - If $Need[i,j] = k$, then P_i may need k more instances of R_j to complete its task: **Note that:** $Need\ [i,j] = Max[i,j] - Allocation\ [i,j]$
- *The previous data structures vary over time in both size and value.*



Deadlock Avoidance

Banker's Algorithm

- **The following notation is used to simplify the banker's algorithm:**
  - Let X and Y be vectors of length n:
    - We say that $X \leq Y$ if and only if $X[i] \leq Y[i]$ for all i = 1, 2, ..., n.
      - For example, if X = (1,7,3,2) and Y = (0,3,2,1), then $Y \leq X$.
    - In addition, $Y < X$ if $Y \leq X$ and $Y \neq X$.
  - Each row in the matrices **Allocation** and **Need** can be treated as vectors and refer to them as
    - $Allocation_i$ specifies the resources currently allocated to process Pi.
    - $Need_i$ specifies the additional resources that process Pi may still request to complete its task.

	Max				Allocation				Need				Available			
	A	B	C	D	A	B	C	D	A	B	C	D	A	B	C	D
P0	6	0	1	2	4	0	0	1	2	0	1	1				
P1	1	7	5	0	1	1	0	0	0	6	5	0				
P2	2	3	5	6	1	2	5	4	1	1	0	2				
P3	1	6	5	3	0	6	3	3	1	0	2	0				
P4	1	6	5	6	1	4	1	2	0	2	4	4				
													2	0	1	1

After P_0 completes P_3 can be allocated. 1020 from released 6012 and available 2011(Total 80 23) and $<P_0, P_3, P_4, P_2, P_1>$ is a safe sequence.

Deadlock Avoidance

- Banker's Algorithm
  - **Safety Algorithm**
    - **Used to find out whether or not a system is in a safe state:**

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

   Work = Available

   Finish*[*i*] = *false for $i = 0, 1, \ldots, n-1$

2. Find an index ***i*** such that both conditions are held:

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

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

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

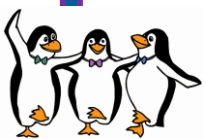
3. $\boldsymbol{Work = Work + Allocation_i}$

   Finish*[*i*] = *true

   go to step 2

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

- It may need an $m \times n^2$ operations to determine if a state is safe.



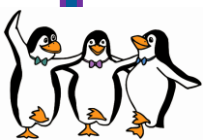
Deadlock Avoidance

Banker's Algorithm

- **Resource-Request Algorithm:** To determine whether requests can be safely granted:
  - Let $\boldsymbol{Request_i}$ be the request vector for process $\boldsymbol{P_i}$
    - If $\boldsymbol{Request_i[j] == k}$ then process $\boldsymbol{P_i}$ wants $\boldsymbol{k}$ instances of resource $\boldsymbol{R_j}$
    - When a request made by P_i, the following actions are taken:
      1. If $\boldsymbol{Request_i \leq Need_i}$ go to step **2**. Otherwise, raise error condition, since process has exceeded its maximum claim
      2. If $\boldsymbol{Request_i \leq Available}$, 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:

$$\boldsymbol{Available = Available - Request_i;}$$
$$\boldsymbol{Allocation_i = Allocation_i + Request_i;}$$
$$\boldsymbol{Need_i = Need_i - Request_i;}$$

    - If the state is safe $\Rightarrow$ the resources are allocated to $\boldsymbol{P_i}$
    - If the state is unsafe $\Rightarrow$ $\boldsymbol{P_i}$ must wait, and restore the state

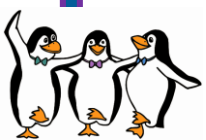


Deadlock Avoidance

Banker's Algorithm

- Example:
  - Suppose that we have 5 processes P_0 to P_4 and 3 resource types A, B, and C:
    - *A* has 10 instances
    - *B* has 5 instances
    - *C* has 7 instances
  - Suppose that at time T_0, the following snapshot of the system has been taken:

	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	



Deadlock Avoidance

Banker's Algorithm

- Example (continue …):
  - The content of the matrix ***Need*** is defined to be ***Max – Allocation*** as follows:

	Allocation	*Max*	*Available*	*Need*
	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

  - The system is in a **safe state** since the sequence < P_1, P_3, P_4, P_2, P_0> satisfies the safety criteria
    - Work = Available = 3 3 2 → 5 3 2 → 7 4 3 → 7 4 5 → 10 4 7 →10 5 7

Deadlock Avoidance

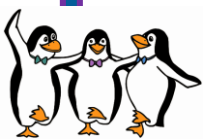
- Banker's Algorithm: Example (continue …):
  - Suppose that P_1 requests → $Request_1 = (1,0,2)$
  - Check that $Request_1 \leq Available$ (that is, $(1,0,2) \leq (3,3,2) \Rightarrow$ **true**)
    - Then pretend that this request has been fulfilled, and we arrive at the following new state:

	Allocation	*Max*	*Available*	*Need*
	A B C	*A B C*	*A B C*	*A B C*
P_0	0 1 0	7 5 3	**2 3 0**	7 4 3
P_1	**3 0 2**	3 2 2		**0 2 0**
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

  - Executing safety algorithm shows that sequence $< \boldsymbol{P_1}, \boldsymbol{P_3}, \boldsymbol{P_4}, \boldsymbol{P_0}, \boldsymbol{P_2} >$ satisfies the safety requirement, then grant the request of P1.
    - Work: 2 3 0 → 5 3 2 → 7 4 3 → 7 4 5 → 7 5 5 → 10 5 7
  - Can request of (3,3,0) by $\boldsymbol{P_4}$ be granted? NO: 3,3,0 > 2, 3, 0
  - Can request of (0,2,0) by $\boldsymbol{P_0}$ be granted? NO: 2,1,0 < all Needs (**unsafe**)

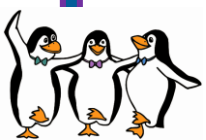
Deadlock Detection

- 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
  - These two requirements apply to systems with a single (or several) instance(s) of each resource type.



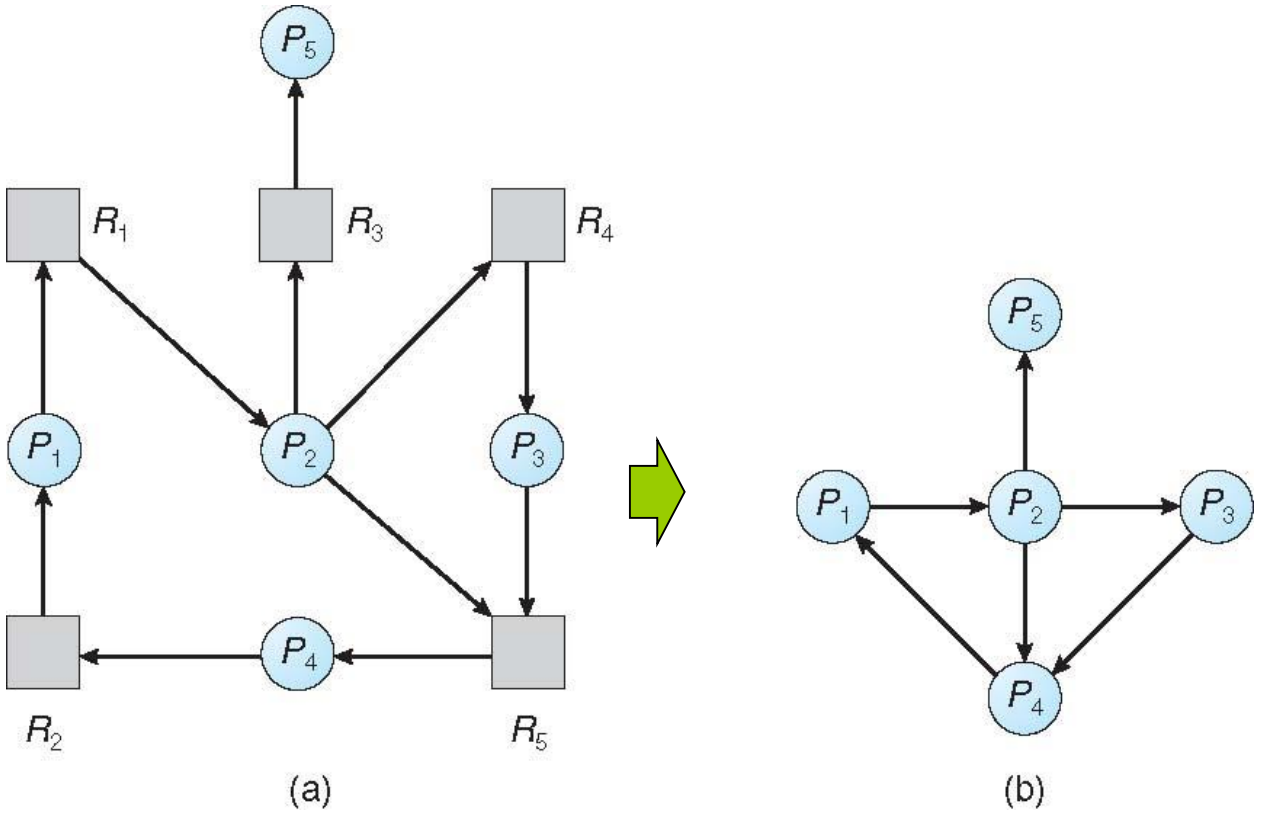
Deadlock Detection

- For Systems with Single Instance of Each Resource Type:
  - Maintain a **wait-for** graph to determine if a deadlock has occurred
    - It is obtained from the resource-allocation graph by removing the resource nodes and collapsing the appropriate edges
      - $P_i \rightarrow P_j$ implies that P_i is waiting for P_j to release a resource that *Pi* needs.
      - An edge *Pi* → *Pj* exists in a wait-for graph if and only if the corresponding resource allocation graph contains two edges *Pi* → *Rq* and *Rq* → *Pj* for some resource *Rq*.
    - 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
    - The wait-for graph scheme is not applicable to a resource-allocation system with multiple instances of each resource type



Deadlock Detection

- For Systems with Single Instance of Each Resource Type

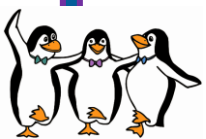


Resource-Allocation Graph

Corresponding wait-for graph

Deadlock Detection

- For Systems of Several Instances of a Resource Type
  - The algorithm employs several time-varying data structures:
    - **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

- For Systems of Several Instances of a Resource Type: Detection Algorithm

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

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

   (b) For ***i* = 0,2, …, *n*-1**

   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***, 0 ≤ ***i*** < ***n***, then the system is in a deadlock state. Moreover, if ***Finish*[*i*] == *false***, then ***P*$_i$** is deadlocked

We assume that Pi will require no more resources to complete its task and will return all resources. If this assumption is incorrect, a deadlock may occur later which will be detected in the next round.

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

Deadlock Detection

- For Systems of Several Instances of a Resource Type: Detection Algorithm
  - **Example**
    - Assume that we have 5 processes P_0 to P_4; and 3 resource types A (7 instances), *B* (2 instances), and *C* (6 instances)
      - Available = [7 2 6]
    - Suppose that at time T_0, we have the following resource-allocation state:

	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	

    - We assume that the system is not in a deadlock state, since the sequence <P_0, P_2, P_3, P_1, P_4> will result in ***Finish[i] = true*** for all ***i***
      - Work= 0 0 0 → 0 1 0 → 3 1 3→ 5 2 4→ 7 2 4→ 7 2 6

Deadlock Detection

- For Systems of Several Instances of a Resource Type: Detection Algorithm
  - **Example (continue …)**
    - Suppose that P_2 requests an additional instance of type ***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 P_0 (since it can finish without the need to get any resources), but there is no sufficient resources to fulfill other processes requests
        - Work= 0 0 0 → 0 1 0 →X not enough for any of the requests
      - Deadlock exists, consisting of processes P_1, P_2, P_3, and P_4

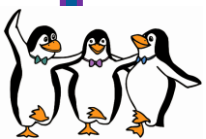
Deadlock Detection

Detection-Algorithm Usage

- When, and how often, to invoke the detection algorithm. It depends on:
  - How often a deadlock is likely to occur?
  - How many processes will be affected (rolled back) by deadlock when it happens?
    - one for each disjoint cycle

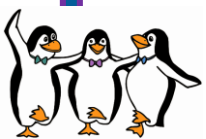
- If a process makes a request that cannot be granted immediately, then, we can invoke the deadlock detection algorithm
  - This request may be the final request that completes a chain of waiting processes → that process is the one that "**caused**" the deadlock

- A less expensive solution: invoke the algorithm at defined intervals (e.g. when the CPU utilization drops below 40%)
  - 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.



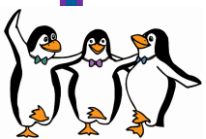
Recovery from Deadlock

- When it is determined that a deadlock exists, alternatives are available:
  - Inform the operator so he/she can deal with it manually.
  - Let the system recover from the deadlock automatically:
    - Abort one or more processes at a time until the deadlock cycle is eliminated.
    - Preempt some resources from one or more of the deadlocked processes.



Recovery from Deadlock

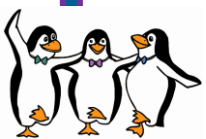
- Process Termination
  - Two methods:
    - Abort all deadlocked processes: very expensive
    - Abort one process at a time until the deadlock cycle is eliminated.
      - High overhead, since after every termination the deadlock detection algorithm has to be invoked
      - **In which order should we choose to abort to minimize the cost?**
        1. Priority of the process
        2. How long a process has computed, and how much longer to completion
        3. How many and what type of resources a process has used
        4. How many resources a process needs to complete
        5. How many processes will need to be terminated
        6. Is a process interactive or batch?
  - **Solution**: define a cost function in which the above criteria have different weights; then choose a process to abort that minimizes the cost



Recovery from Deadlock

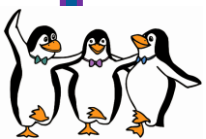
Resource Preemption

- Preempt some resources from some processes and give them to other processes until the deadlock cycle is broken
- **Three issues need to be addressed:**
  - **Selecting a victim:** we must determine the order of preemption to minimize cost
  - **Rollback**: a victim process must return to some safe state, then restart the process for that state
    - It is difficult to determine what a safe state is, so use a total rollback
  - **Starvation**: same process may always be picked as a victim, then include the number of rollbacks in the cost factor. So a process can be picked only a finite number of times



Hybrid Approach to Deadlock

- Researchers have argued that none of the basic approaches alone is appropriate for the entire spectrum of resource-allocation problems in operating systems.
- The basic approaches can be a combined one, however, allowing us to select an optimal approach for each class of resources in a system.



Examples:

- Q1): Consider the following snapshot of a system:

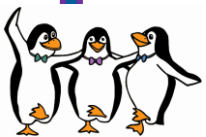
	Allocation				Max				Available			
	A	B	C	D	A	B	C	D	A	B	C	D
P_0	0	0	1	2	0	0	1	2	1	5	2	0
P_1	1	0	0	0	1	7	5	0				
P_2	1	3	5	4	2	3	5	6				
P_3	0	6	3	2	0	6	5	2				
P_4	0	0	1	4	0	6	5	6				

- Answer the following questions using the banker's algorithm:

a) What is the content of the matrix Need?

b) Is the system in a safe state?

c) If a request from process P1 arrives for (0,4,2,0), can the request be granted immediately?



Examples:

- Q2) In a real computer system, neither the resources available nor the demands of processes for resources are consistent over long periods (months). Resources break or are replaced, new processes come and go, new resources are bought and added to the system.
- If deadlock is controlled by the banker's algorithm, which of the following changes can be made safely (without introducing the possibility of deadlock), and under what circumstances?
  a) Increase Available (new resources added)
  b) Decrease Available (resource permanently removed from system)
  c) Increase Max for one process (the process needs or wants more resources than allowed).
  d) Decrease Max for one process (the process decides it does not need that many resources)
  e) Increase the number of processes
  f) Decrease the number of processes