Chapter 7

Deadlocks

Chu–Sing Yang

Department of Electrical Engineering
National Cheng Kung University
Chapter 7: Deadlocks

- Introduction
- 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
Introduction

- In a multiprogramming environment
  - Several processes may compete for a finite number of resources
  - A process requests resources
    - If the resources are not available at that time, the process enters a waiting state
    - Deadlock
      - A waiting process is never again able to change state
      - Because the resources it requested are held by other waiting processes

- Describe the methods that an OS can use to prevent or deal with deadlocks
  - Most current OSs do not provide deadlock-prevention facilities
    - But such features will probably be added soon
    - Deadlock problems can become more common
      - Large numbers of processes
      - Multithreaded programs
      - Many more resources within a system
      - 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 $R_1, R_2, \ldots, R_m$
    - CPU cycles, memory space, files, and I/O devices (printers & DVD drivers)
  - Each resource type $R_i$ has $W_i$ instances
    - If a process requests an instance of a resource type, the allocation of any instance of the type will satisfy the request
    - If it will not, then the instances are not identical

- Each process utilizes a resource as follows:
  - Request
    - If the request cannot be granted immediately, then the requesting process must wait until it can acquire the resource
  - Use
    - The process can operate on the resource
  - Release
    - The process releases the resource
Deadlock Example

- Two mutex locks are created

```c
/* 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);
```
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);
}
Deadlock can arise if four conditions hold simultaneously in a system:

- **Mutual exclusion**
  - At least one resource must be held in a nonsharable mode
    - Only one process at a time can use a resource
    - If another process requests that resource,
      » The request process must be delayed until the resource has been released

- **Hold and wait**
  - A process holding at least one resource is waiting to acquire additional resources held by other processes

- **No preemption**
  - Resources cannot preempted
  - 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, ..., P_n\} of waiting processes
    - \(P_0\) 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 \(P_n\)
    - \(P_n\) is waiting for a resource that is held by \(P_0\)
Deadlocks can be described in terms of a directed graph called a system resource-allocation graph

- A set of vertices $V$
  - $V$ is partitioned into two types:
    - $P = \{P_1, P_2, \ldots, P_n\}$, the set consisting of all the active processes in the system
    - $R = \{R_1, R_2, \ldots, R_m\}$, the set consisting of all resource types in the system

- A set of edges $E$
  - Request edge
    - A directed edge $P_i \rightarrow R_j$
    - Process $P_i$ has requested an instance of resource type $R_j$ and is currently waiting for that resource
  - Assignment edge
    - A directed edge $R_j \rightarrow P_i$
    - An instance of resource type $R_j$ has been allocated to process $P_i$
Resource-Allocation Graph

- 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
Resource Allocation Graph With A Deadlock

- Two minimal cycles exist in the system
  - $P_1 \rightarrow R_1 \rightarrow P_2 \rightarrow R_3 \rightarrow P_3 \rightarrow R_2 \rightarrow P_1$
  - $P_2 \rightarrow R_3 \rightarrow P_3 \rightarrow R_2 \rightarrow P_2$
Resource Allocation Graph With A Cycle But No Deadlock
Basic Facts

- If graph contains no cycles ⇒ no deadlock

- If graph contains a cycle ⇒
  - if only one instance per resource type, then deadlock.
  - if several instances per resource type, possibility of deadlock.
Methods for Handling Deadlocks

- Ensure that the system will *never* enter a deadlock state
  - Use a protocol to prevent or avoid deadlocks
- 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 & 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.

**Mutual Exclusion**
- Must hold for nonsharable resources
  - Printer cannot be shared by several processes
- Not required for sharable resources
  - Read-only files are sharable resources
- We cannot prevent deadlocks by denying the mutual-exclusion
  - Because some resources are nonsharable
Deadlock Prevention (Cont.)

- **Hold and Wait**
  - Must guarantee that whenever a process requests a resource, it does not hold any other resources
    1. Require process to request and be allocated all its resources before it begins execution
    2. Allow process to request resources only when the process has none
  - A process copies data from a DVD driver to a file on disk, sorts the file, and then prints the results to a printer
    1. Process initially requests the DVD driver, disk file, and printer
    2. Process initially requests the DVD driver and disk file
      - It copies from the DVD driver to the disk and then release both the DVD driver and the disk file
      - Process then again request the disk file and the printer
      - After copying disk file to printer, it releases these two resources and terminates
  - **Two main disadvantages**
    - Low resource utilization
      - Resources may be allocated but unused for a long period
    - Starvation is possible
      - A process that needs several resources may have to wait indefinitely
No Preemption of resources that have already been allocated

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 preempted
  - 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

- Alternatively, if a process requests some resources
  - If they are available, we allocate them
  - If they are not, we check to whether they are allocated to some other process that is waiting for additional resources
    - If so, we preempt the desired resources from the waiting process and allocate them to the requesting process
    - If the resources are neither available nor held by a waiting process, the request process must wait
  - A process can be restarted only when it is allocated the new resources it is requesting and recovers any resources that were preempted while it was waiting
Circular Wait

- Impose a total ordering of all resource types, and require that each process requests resources in an increasing order of enumeration.
- Let $R = \{R_1, R_2, \ldots, R_m\}$ be the set of resource types.
- Define a one-to-one function $F: R \rightarrow N$, where $N$ is the set of natural numbers.
  
  $F($tape driver$) = 1$
  $F($disk driver$) = 5$
  $F($printer$) = 12$

- Each process can request resources only in an increasing order of enumeration.
  - A process can initially request any number of instances of a resource type $R_i$.
    - After that, the process can request instances of resource type $R_j$ if and only if $F(R_j) > F(R_i)$.
    - If several instances of the same resource type are needed, a single request for all of them must be issued.
  - Alternatively, we can require that, whenever a process requests an instance of resource type $R_j$, it has released any resources $R_i$ such that $F(R_i) \geq F(R_j)$. 

Deadlock Prevention (Cont.)

- Two protocols are used, then the circular-wait condition cannot hold.
  Let the set of processes involved in the circular wait be \( P = \{ P_0, P_1, \ldots, P_n \} \),
  where \( P_i \) is waiting for a resource \( R_i \), which is held by process \( P_{i+1} \).
  Since process \( P_{i+1} \) is holding resource \( R_i \) while requesting resource \( R_{i+1} \),
  we must have \( F(R_j) < F(R_i) \), for all \( i \).
  But this condition means that \( F(R_0) < F(R_1) < \ldots < F(R_n) < F(R_0) \).
  By transitivity, \( F(R_0) < F(R_0) \), which is impossible.
  Therefore, there can be no circular wait.

- Ensuring that resources are acquired in the proper order is the responsibility of application developers.
  - Certain SW can be used to verify that locks are acquired in the proper order and to give appropriate warnings when locks are acquired out of order and deadlock is possible.
  - One lock-order verifier works on FreeBSD.
    - Witness uses mutual-exclusion locks to protect critical sections.
      - It works by dynamically maintaining the relationship of lock orders.
Deadlock Avoidance

- Deadlock-prevention algorithms prevent deadlocks by retraining how requests can be made
  - The restraints ensure that at least one of the necessary conditions for deadlock cannot occur and that deadlocks cannot hold
  - Side effects of preventing deadlocks are low device utilization and reduced system throughput
- An alternative method for avoiding deadlocks is to require additional information about how resources are to be requested
- Requires that the system has some additional *a priori* information available
  - Simplest and most useful model requires that each process declare the maximum number of resources of each type that it may need
  - The deadlock-avoidance algorithm dynamically examines the resource-allocation state to ensure that there can never be a circular-wait condition
  - Resource-allocation state is defined by the number of available and allocated resources, and the maximum demands of the processes
Safe State

- A state is safe if the system can allocate resources to each process 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 of all processes.
- Sequence \(<P_1, P_2, \ldots, P_n>\) is safe if 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 \(j<i\).
  - If resources that \(P_i\) 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.
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
- Consider a system with 12 magnetic tape drives and three processes: $P_0$, $P_1$, and $P_2$,

<table>
<thead>
<tr>
<th>Maximum Needs</th>
<th>Current Needs</th>
</tr>
</thead>
<tbody>
<tr>
<td>$P_0$</td>
<td>10</td>
</tr>
<tr>
<td>$P_1$</td>
<td>4</td>
</tr>
<tr>
<td>$P_2$</td>
<td>9</td>
</tr>
</tbody>
</table>

- At time $t_0$, the system is in a safe state
  - Sequence $\langle P_1, P_0, P_2 \rangle$ satisfies the safety condition
- System can go from a safe state to an unsafe state
  - At time $t_1$, process $P_2$ requests and is allocated one more tape drive
  - The system is no longer in a safe state
  - Only process $P_1$ can be allocated all its tape drives
Safe, Unsafe, Deadlock State
Resource-Allocation Graph Algorithm

- A resource-allocation system with only one instance of each resource type
  - Request edges and assignment edges
  - Claim edge $P_i \rightarrow R_j$ indicated that process $P_i$ may request resource $R_j$ at some time in the future
    - Resembles a request edge in direction but is represented in the graph by a dashed line
  - Claim edge converts to request edge when a process requests a resource
  - When a resource $R_j$ is released by a process $P_i$, assignment edge reconverts to a claim edge
  - Resources must be claimed "a priori" in the system
    - Before process $P_i$ starts executing, all its claim edges must already appear in the RAG
  - Process $P_i$ requests resource $R_j$
    - The request can be granted only if converting the request edge $P_i \rightarrow R_j$ to an assignment edge $R_j \rightarrow P_i$ does not result in the formation of a cycle in the RAG
  - Use a cycle-detection algorithm to check for safety
    - If no cycle exists, then the allocation of resource will leave the system in a safe state
    - If a cycle is found, then the allocation will put the system in an unsafe state
      - Process $P_i$ will have to wait for its requests to be satisfied
Resource-Allocation Graph For Deadlock Avoidance

- $P_2$ requests $R_2$
- Although $R_2$ is currently free, we cannot allocate it to $P_2$, since this action will create a cycle in the graph.
Unsafe State In Resource-Allocation Graph

- Suppose that $P_2$ requests $R_2$
- Although $R_2$ is currently free, we cannot allocate it to $P_2$, since this action will create a cycle in the graph
- A cycle indicates that the system is in an unsafe state
- If $P_1$ requests $R_2$, and $P_2$ requests $R_1$, then a deadlock will occur
Banker’s Algorithm

- Is applicable to multiple instances of each resource type
  - Is less efficient than RAG scheme
  - 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
- When a new process enters the system, it must declare maximum # instances of each resource type
  - When a process requests a set of resource, system must determine whether the allocation of the resources will leave the system in a safe state
    - If it will, the resources are allocated
    - Otherwise, the process must wait
  - When a process gets all its resources it must return them in a finite amount of time
Data Structures for the Banker’s Algorithm

- Let \( n \) be the number of processes, and \( m \) be the number of resources types
  - **Available**
    - Vector of length \( m \) indicated # available resources of each type
    - If available \([j] = k\), there are \( k \) instances of resource type \( R_j \) available
  - **Max**
    - An \( n \times m \) 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 \( n \times m \) matrix defines # 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 \( n \times m \) 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
Safety Algorithm

1. Let Work and Finish be vectors of length m and n, respectively.
   Initialize:
   
   \[ \text{Work} = \text{Available} \]
   
   \[ \text{Finish}[i] = \text{false} \text{ for } i = 0, 1, \ldots, n-1 \]

2. Find an \( i \) such that both:
   a. \( \text{Finish}[i] == \text{false} \)
   b. \( \text{Need}_i \leq \text{Work} \)

   If no such \( i \) exists, go to step 4.

3. \( \text{Work} = \text{Work} + \text{Allocation}_i \)
   \( \text{Finish}[i] = \text{true} \)
   go to step 2.

4. If \( \text{Finish}[i] == \text{true} \text{ for all } i \), then the system is in a safe state.
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 $R_j$.

- When a request for resources is made by process $P_i$, the following actions are taken:
  1. If $Request_i \leq Need_i$, go to step 2.
     Otherwise, raise an error condition, since the process has exceeded its maximum claim.
  2. If $Request_i \leq Available$, go to step 3.
     Otherwise, $P_i$ must wait, since resources are not available.
  3. Pretend to allocate requested resources to $P_i$ by modifying the state as follows:
     \[ Available = Available - Request_i; \]
     \[ Allocation_i = Allocation_i + Request_i; \]
     \[ Need_i = Need_i - Request_i; \]
     - If state is safe $\Rightarrow$ the resources are allocated to $P_i$.
     - If unsafe $\Rightarrow$ $P_i$ 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 $T_0$:

\[
\begin{array}{llll|llll|llll}
$P_0$ & & & & 0 & 1 & 0 & & 7 & 5 & 3 & & 3 & 3 & 2 & & \\
$P_1$ & & & & 2 & 0 & 0 & & 3 & 2 & 2 & & & & & & \\
$P_2$ & & & & 3 & 0 & 2 & & 9 & 0 & 2 & & & & & & \\
$P_3$ & & & & 2 & 1 & 1 & & 2 & 2 & 2 & & & & & & \\
$P_4$ & & & & 0 & 0 & 2 & & 4 & 3 & 3 & & & & & & \\
\end{array}
\]
## 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 $T_0$:

<table>
<thead>
<tr>
<th>Allocation</th>
<th>Max</th>
<th>Available</th>
</tr>
</thead>
<tbody>
<tr>
<td>$A$</td>
<td>$B$</td>
<td>$C$</td>
</tr>
<tr>
<td>$P_0$</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>$P_1$</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>$P_2$</td>
<td>3</td>
<td>0</td>
</tr>
<tr>
<td>$P_3$</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>$P_4$</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>
Example (Cont.)

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

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>Need</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$P_0$</td>
<td>7</td>
<td>4</td>
<td>3</td>
</tr>
<tr>
<td>$P_1$</td>
<td>1</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>$P_2$</td>
<td>6</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>$P_3$</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>$P_4$</td>
<td>4</td>
<td>3</td>
<td>1</td>
</tr>
</tbody>
</table>

- The system is in a safe state since the sequence $< P_1, P_3, P_4, P_2, P_0 >$ satisfies safety criteria.
### Example of Banker’s Algorithm

<table>
<thead>
<tr>
<th>Allocation</th>
<th>Max</th>
<th>Need</th>
<th>Available</th>
</tr>
</thead>
<tbody>
<tr>
<td>Finish</td>
<td>A B C</td>
<td>A B C</td>
<td>A B C A B C</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>3 3 2</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>P_0</th>
<th>0 1 0</th>
<th>7 5 3</th>
<th>7 4 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>F</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>P_1</th>
<th>2 0 0</th>
<th>3 2 2</th>
<th>3 2 2</th>
<th>1 2 2</th>
<th>2 1 0</th>
<th>5 3 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>F</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>T</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>P_2</th>
<th>3 0 2</th>
<th>9 0 2</th>
<th>6 0 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>F</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>P_3</th>
<th>2 1 1</th>
<th>3 2 2</th>
<th>2 2 2</th>
<th>0 1 1</th>
<th>4 2 1</th>
<th>7 4 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>F</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- The system is in a safe state since the sequences <P_1, P_3, P_4, P_2, P_0>, <P_1, P_3, P_4, P_0, P_2>, <P_1, P_3, P_0, P_2, P_4> satisfy safety criteria.
Example $P_1$ Request (1,0,2)

- Check that Request $1 \leq$ Available (that is, (1,0,2) $\leq$ (3,3,2) $\Rightarrow$ true.

<table>
<thead>
<tr>
<th>Allocation</th>
<th>Need</th>
<th>Available</th>
</tr>
</thead>
<tbody>
<tr>
<td>A B C</td>
<td>A B C</td>
<td>2 3 0</td>
</tr>
<tr>
<td>$P_0$</td>
<td>0 1 0</td>
<td>7 4 3</td>
</tr>
<tr>
<td>$P_1$</td>
<td>3 0 2</td>
<td>3 2 2 0</td>
</tr>
<tr>
<td>$P_2$</td>
<td>3 0 1</td>
<td>6 0 0</td>
</tr>
<tr>
<td>$P_3$</td>
<td>2 1 1</td>
<td>3 2 2 0</td>
</tr>
<tr>
<td>$P_4$</td>
<td>0 0 2</td>
<td>4 3 1</td>
</tr>
</tbody>
</table>

- Executing safety algorithm shows that sequence $<P_1, P_3, P_4, P_0, P_2>$ satisfies safety requirement.
  - We can grant the request of process $P_1$

- Can request for (3,3,0) by $P_4$ be granted?
  - The resources are not available

- Can request for (0,2,0) by $P_0$ be granted?
  - Cannot be granted, even though the resources are available
Deadlock Detection

A deadlock situation may occur if a system does not employ either a deadlock-prevention or a deadlock avoidance algorithm

- Deadlock detection algorithm
  - An algorithm that examines the state of the system to determine whether a deadlock has occurred

- Recovery scheme
  - An algorithm to recover from the deadlock
Single Instance of Each Resource Type

- Defines a wait-for graph
  - A resource-allocation graph with only a single instance
  - Obtained from the resource-allocation graph by removing the resource nodes and collapsing the appropriate edges

- Maintain wait-for graph
  - Nodes are processes.
  - $P_i \rightarrow P_j$ if $P_i$ is waiting for $P_j$.
  - $P_i \rightarrow P_j$ exists iff the corresponding resource-allocation graph contains two edges $P_i \rightarrow R_q$ and $R_q \rightarrow P_j$

- To detect deadlocks
  - The system needs to maintain the wait-for graph
  - 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.
Resource-Allocation Graph & Wait-for Graph

Resource-Allocation Graph  Corresponding wait-for graph

(a)  (b)
Several Instances of a Resource Type

- *Wait-for* graph is not applicable to a resource-allocation system with multiple instances of each 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 $n \times m$ matrix defines the number of resources of each type currently allocated to each process.
  - *Request*
    - An $n \times m$ 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 $R_j$. 

*Operating System Concepts*

Silberschatz, Galvin and Gagne c2005
Detection Algorithm

1. Let Work and Finish be vectors of length $m$ and $n$, respectively.
   Initialize:
   
   (a) Work = Available
   
   (b) For $i = 0, 1, 2, \ldots, n-1$, 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 $i$, $0 \leq i < n$, then the system is in
   deadlock state. Moreover, if Finish[$i$] == false, then $P_i$ is deadlocked

- Algorithm requires an order of $O(m \times n^2)$ operations to detect whether
  the system is in deadlocked state
Example of Detection Algorithm

- Five processes $P_0$ through $P_4$;
- three resource types A (7 instances), B (2 instances), and C (6 instances).

- Snapshot at time $T_0$, we have the following resource-allocation state:

<table>
<thead>
<tr>
<th>Allocation</th>
<th>Request</th>
<th>Available</th>
</tr>
</thead>
<tbody>
<tr>
<td>$A$</td>
<td>$B$</td>
<td>$C$</td>
</tr>
<tr>
<td>$P_0$</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>$P_1$</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>$P_2$</td>
<td>3</td>
<td>0</td>
</tr>
<tr>
<td>$P_3$</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>$P_4$</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

- Sequence $<P_0, P_2, P_3, P_1, P_4>$ will result in $Finish[i] = true$ for all $i$. 
Example of Detection Algorithm

- Suppose that $P_2$ requests an additional instance of type $C$.
- The request matrix is modified as follows

<table>
<thead>
<tr>
<th>Allocation</th>
<th>Request</th>
<th>Available</th>
</tr>
</thead>
<tbody>
<tr>
<td>$A$</td>
<td>$B$</td>
<td>$C$</td>
</tr>
<tr>
<td>$P_0$</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>$P_1$</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>$P_2$</td>
<td>3</td>
<td>0</td>
</tr>
<tr>
<td>$P_3$</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>$P_4$</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

- 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 $P_4$.  

Silberschatz, Galvin and Gagne c2005
Detection—Algorithm Usage

- When, and how often, to invoke the detection algorithm depends on:
  - How often a deadlock is likely to occur?
    - If deadlocks occur frequently, then the detection algorithm should be invoked frequently
  - 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
  - If the deadlock-detection algorithm is invoked for every resource request, this will incur a considerable overhead in computation time
  - A less expensive alternative is simply to invoke the algorithm at less frequent intervals
    - Once per hour
    - Whenever CPU utilization drops below 40%
Recovery from Deadlock

- When a detection algorithm determines that a deadlock exists
  - 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
- 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

To eliminate deadlocks by aborting a process, we use one of two methods:

- **Abort all deadlocked processes**
  - At great expense
  - Deadlocked processes may have computed for a long time, and the results of these partial computations must be discarded and probably will have to be recomputed later

- **Abort one process at a time until the deadlock cycle is eliminated.**
  - Incurs considerable overhead

In which order should we choose to abort?

- What the priority of the process is
- How long process has computed, and how much longer to completion
- How many and what type of resources the process has used
- How many more resources the process needs to complete
- How many processes will need to be terminated
- Whether the process is interactive or batch
Recovery from Deadlock: Resource Preemption

- Preempt some resources from processes and give these resources to other processes until the deadlock cycle is broken
  - Selecting a victim - minimize cost
    - Cost factors to determine the order of preemption
      - The number of resources a deadlocked process is holding
      - The amount of time the process has thus far consumed during its execution
  - Rollback
    - The process cannot continue with its normal execution, it is missing some needed resource
    - Roll back the process to some safe state and restart it from that state.
    - The simplest solution is a total rollback: Abort the process and then restart it
    - To rollback the process only as far as necessary to break the deadlock
  - Starvation
    - How can we guarantee that resources will not always be preempted from the same process?
    - Same process may always be picked as victim, include number of rollback in cost factor.
End of Chapter 7