Deadlock

• A state of indefinite waiting of a process
• May occur when processes are competing for resources or when attempting to interact with one another
Resource Allocation and Deallocation

When a process needs resources, it will normally follow the sequence:

1. **Request** a number of resource instances. If the resources requested are not available, then process must **wait**
2. **Acquire** resource(s), the OS allocates resources
3. **Use** the resource(s) for a finite interval.
4. **Release** the resource(s).
Synchronizing Allocation of Resources

Semaphores can be used to synchronize the allocation and de-allocation of resources.
Multiple Resource Allocation Problem

- Several processes may compete for multiple resources
- If the allocation of multiple resources to processes is not done with proper synchronization, deadlock might occur
- This waiting state is much more involved than starvation
Deadlock Problem

- A set of blocked (suspended) processes
- Each process holding one or more resources
- Processes waiting to acquire one or more resources held by other processes
Example of Deadlock

- A system has two tape drives, which processes P1 and P2 need
- Processes P1 and P2 each acquire one tape drive and each wait for the other tape drive
- Processes P1 and P2 are blocked waiting indefinitely to acquire the other tape unit
Deadlock – General Definition

• A set of processes is in a deadlock state when every processes in the set is waiting for an event that can only be caused by another process in the set.

• The events of interest are:
  – allocations of resources
  – release of resources

• The resources are: CPU cycles, memory spaces, files, I/O devices in the computer system.
Disadvantage of Deadlock

- Deadlocked processes can never complete execution because they are waiting indefinitely.
- System resources are tied up because they are held by deadlocked processes.
- Deadlocks are undesirable because they normally slow down a system.
Another Example of Deadlock

• Consider a system with one printer and one disk unit. Processes P and Q both need the two resources.
• Suppose that process P is holding the printer, and process Q is holding the disk unit. Now P waits for the disk unit and Q waits for the printer.
• Process P is waiting for process Q to release the disk unit, and process Q is waiting for process P to release the printer.
Both processes are **deadlocked**, they cannot proceed.
A resource allocation graph shows the relationships between processes and resources. It consists of the following components:

- A set of processes $P = \{ P_0, P_1, \ldots, P_i, \ldots, P_n \}$
- A set of resource types, $R = \{ R_0, R_1, \ldots, R_j, \ldots, R_m \}$
- A set of directed edges
Directed Edges

• A request edge \((Pi, Rj)\) from process \(Pi\) to resource \(Rj\). Thus, \(Pi\) is waiting for an instance of resource type \(Rj\).

• An assignment edge \((Rj, Pi)\) from resource \(Rj\) to process \(Pi\). It indicates that an instance of \(Rj\) has been allocated to process \(Pi\).
A Resource Allocation Graph
Conditions for Deadlock

A deadlock arises if four conditions hold simultaneously in a system:

- **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.
Conditions for Deadlock (Cont.)

- **No preemption**: a resource can be released only voluntarily by the process holding it, after that process has completed its operations.
- **Circular wait**: there exists a set \{P0, P1, ...Pn\} of waiting processes such that P0 is waiting for a resource that is held by P1, but P1 is waiting for a resource held by P2, ..., Pn-1 is waiting for a resource held by Pn, and Pn is waiting for a resource held by P0.
About These Conditions

These conditions are necessary but not sufficient for deadlock to occur.
Resource Allocation Graph with a Cycle
Analysis of This Graph

• This graph has a cycle
• Processes p1, p2, and p3 are deadlocked.
Another Resource Allocation Graph
Analysis of Resource Allocation Graph

• If the graph contains no cycles, no deadlock exists.

• If the graph contain cycles
  – if each resource type has only 1 instance, then a deadlock exits.
  – otherwise, a deadlock may exist.
Dining Philosophers

• Illustrates processes that are competing for exclusive access to more than one resource, such as tape drives or I/O devices.

• A monastery’s dining table
  – Circular table
  – Five plates
  – Five forks (critical)
  – Plate of noodles at center of table (endless supply)
  – Philosophers can only reach forks adjacent to their plate (left and right forks)
Life of a Philosopher

- Spends some time thinking
- Gets hungry
- Sits at the table
- Attempts to get the left fork
- Attempts to get the right fork
- Starts eating, this activity takes a finite time
- Releases the left and right forks
- Returns to the initial state (thinking)
Dining Table
Why Synchronize?

• When a philosopher is hungry
  – Obtains 2 forks (1 at a time)
  – Proceeds to eat
• When a philosopher has satisfied hunger
  returns both forks and goes back to think
• Problem: There are 5 competing philosopher
  processes
• Using semaphores as before, is not sufficient for
  solving the problem
Dining Philosophers 1st Attempt

• Suppose all philosophers want to eat at the same time
• Each one will pick up the left fork first then block trying to pickup the right fork
• All processes will now block indefinitely - deadlock
Dining Philosophers 2nd Attempt

• After taking the left fork, if the right fork is not available, philosopher puts back the left fork and waits for a while.

• Problem: Suppose all philosophers want to eat at the same time:
  – Each will pick up the left fork first and then try to pick up the right fork.
  – The right fork is not available, so all philosophers put back left forks and wait.
  – After some time, philosophers pick up the left fork and try again - the cycle repeats.
Dining Philosophers 3rd Attempt

- Use a *mutex* semaphore to make eating mutually exclusive.
- A philosopher is guaranteed to get both forks in this case.
- Problem: Only one philosopher can be eating at any given time.
4th Attempt to a Solution

• A philosopher’s neighbors are defined by macros LEFT & RIGHT.
• A philosopher may move only into eating state if neither neighbor is eating.
• Use an array, *state*, to keep track of whether a philosopher is eating, thinking or hungry (trying to acquire forks).
• Use an array of semaphores, one per philosopher, so hungry philosophers can block if the needed forks are busy.
Handling Deadlocks

General approaches

• Ensure that the system will never enter a deadlock state.

• Allow the system to enter a deadlock state and then recover.

• Ignore the problem and pretend that deadlocks never occur in the system.
Methods for Dealing with Deadlocks

• **Deadlock prevention**: disallow the existence of one of the four necessary conditions for deadlocks to occur.

• **Deadlock avoidance**: for each resources request which can be satisfied, determine whether the request should be delayed in order to avoid a possible deadlock in the future.

• **Deadlock detection and recovery**: detect the existence of deadlock; if it has occurred, take actions to remove it.
Deadlock Prevention

• Non-existence of the **hold and wait** condition
• Non-existence of the **circular wait** condition
Deadlock Prevention (1)

Non-existence of hold and wait.

• Allocation only if all resources requested are available

• Allocation of resources only when there is no previous allocation.

• Low resource utilization

• Starvation possible
Deadlock Prevention (2)

• Non-existence of circular wait.
  Impose a linear ordering of resource types.
  Processes must request resources in an increasing order of enumeration.
Disallow Circular Wait

• Impose a total ordering of all resource types, $R = \{R_1, R_2, \ldots, R_m\}$.
• Define a one to one function, $F: R \rightarrow N$
• Require that each process requests resources in an increasing order of enumeration.
• Process $P$ can request resources of type $R_i$, then resources of type $R_j$, only if $F(R_j) > F(R_i)$
General Cycle
Example of Linear Ordering

• A system has the following resources types:
  \[ R = \{\text{Disk, Tape, Printer}\} \]
• The following ordering is defined for the resources with function \( F \):
  \[ F(\text{Disk}) = 1; \ F(\text{Tape}) = 3; \ F(\text{Printer}) = 7 \]
• A process, \( P \), has to request first an instance of Disk, then an instance of Tape, then an instance of Printer.
Deadlock Prevention (3)

- Non-existence of mutual exclusion; make each resource shareable (impossible for printers).
- Existence of preemption:
  - Process must release resources held when no further resource allocations are possible.
  - Preempt the desired resources from a waiting process which hold the resource.
  - A Process will be restarted only when it can regain its old resources, and acquire the new resources requested.
Common Problems For Deadlock Prevention Methods

• Low device utilization
• Reduced system throughput

Why?
Simulation Models Deadlock Prevention

• The simulation models that apply the deadlock prevention techniques are based on the Five Philosophers Problem.

• The first model always deadlocks – no deadlock techniques is applied in this model.

• The second model applies the absence of the hold and wait condition.

• The third model applies the absence of the circular wait condition.
Dining Philosophers

• A monastery’s dining table
  – Circular table
  – Five plates
  – Five forks (critical resources)
  – Plate of noodles at center of table (endless supply)
  – Philosophers, Po, P1, P2, P3, and P4, can only reach forks adjacent to their plate (left and right forks)
Dining-Philosophers Problem

Every philosopher needs two forks (or forks) and these must be accessed in a mutual exclusive manner

// Shared data
Semaphore fork[] = new Semaphore[5];
Output Listing of a Simulation Run

Project: Dining Philosophers - Deadlock
Run at: Sat Oct 01 17:15:39 EDT 2006 by jgarrido on Windows XP

-----------------------------------
Input Parameters
Simulation Period: 740
Activity Stop: 400
Mean Think Time: 17.5
Mean Eat Time: 12.75
Project: Dining Philosophers - Deadlock
Run at: Sat Oct 01 17:15:40 EDT 2005
Start Simulation
Time 0000.000 Philosopher0 starts
Time 0000.000 Philosopher1 starts
Time 0000.000 Philosopher2 starts
Time 0000.000 Philosopher3 starts
Time 0000.000 Philosopher4 starts
Informal Solution to Deadlock

- Deadlock will not occur when the philosophers do not all sit at the table at the same time, therefore, if the five philosophers start thinking before eating, deadlock will not always happen.
- The reason for this is that the time interval to think is randomly generated so that the philosophers will not normally start to eat at the same time, consequently, deadlock will not always occur.
- This solution to deadlock shows that the difference in the ordering of events can make a big difference.
- The Java simulation model is in: philos2.jar; the C++ model is in philos2.cpp.
Simulation Model – Informal Solution

• The simulation model consists of three user-defined classes: *Philosopher*, *Philos2*, and *SimInputs*.
• The second class *Philos2* is the main class.
• A *Philosopher* object attempts to acquire the left fork, then attempts to acquire the right fork.
• Each *Philosopher* object generates a random time interval for the duration of its *eating* activity, and another random time interval for the duration of its *thinking* activity.
• As mentioned before, this solution for deadlock specifies that a philosopher first thinks then eats, and continues this cycle endlessly.
Partial Trace of a Simulation Run

Time 0000.000 Philosopher0 start think activity
Time 0000.000 Philosopher1 start think activity
Time 0000.000 Philosopher2 start think activity
Time 0000.000 Philosopher3 start think activity
Time 0000.000 Philosopher4 start think activity
Time 0005.263 Philosopher0 requesting left fork
Time 0005.263 Philosopher0 acquired left fork
Time 0005.263 Philosopher0 requesting right fork - waiting
Time 0005.263 Philosopher0 acquired right fork
Time 0005.263 Philosopher0 starts eating
Time 0008.710 Philosopher1 requesting left fork
Time 0009.097 Philosopher0 releasing forks
Time 0009.097 Philosopher0 start think activity
Time 0009.097 Philosopher1 acquired left fork
Time 0009.097 Philosopher1 requesting right fork - waiting
Time 0009.097 Philosopher1 acquired right fork
Time 0009.097 Philosopher1 starts eating
Time 0015.443 Philosopher1 releasing forks
Time 0015.443 Philosopher1 start think activity
Simulation Results

Results of simulation: Five Dining Philosophers

Results of simulation:
Number of eating cycles Philosopher0 = 10
Total waiting time Philosopher0 = 0203.9259
Number of eating cycles Philosopher1 = 10
Total waiting time Philosopher1 = 0131.0330
Number of eating cycles Philosopher2 = 9
Total waiting time Philosopher2 = 0164.6285
Number of eating cycles Philosopher3 = 11
Total waiting time Philosopher3 = 0120.0247

Number of eating cycles Philosopher4 = 7
Total waiting time Philosopher4 = 0112.4418
Average time waiting: 0146.4108
Model That Disallows Hold and Wait

- The two basic user-defined classes in the model are *Philoshw* and *Philosopher*.
- These classes are implemented in files *Philoshw.java* and *Philosophers.java* and are stored in the archive file *philoshw.jar*.
- The C++ implementation of this model is stored in file: *philoshw.cpp*.
Results of Simulating Hold and Wait

Number of eat cycles Philosopher0 = 8
Total waiting time to eat Philosopher0 = 31.6265
Number of eat cycles Philosopher1 = 3
Total waiting time to eat Philosopher1 = 50.6937
Number of eat cycles Philosopher2 = 3
Total waiting time to eat Philosopher2 = 11.0344
Number of eat cycles Philosopher3 = 6
Total waiting time to eat Philosopher3 = 43.5825
Number of eat cycles Philosopher4 = 10
Total waiting time to eat Philosopher4 = 50.2282
Average time waiting to eat: 37.4331

End of simulation Concurrent philosophers problem - H & W
Model That Disallows Circular Wait

• The two basic user-defined classes in the model are *Philoscw* and *Philosopher*.

• These classes are implemented in files *Philoscw.java* and *Philosophers.java* and are stored in the archive file *philoscw.jar*.

• The C++ implementation of this model is stored in file: *philoscw.cpp*. 
Results of Simulating Circular Wait

Number of eating cycles phil 0 = 5
Total waiting time to eat phil 0 = 87.4548
Number of eating cycles phil 1 = 5
Total waiting time to eat phil 1 = 38.2743
Number of eating cycles phil 2 = 4
Total waiting time to eat phil 2 = 29.492
Number of eating cycles phil 3 = 8
Total waiting time to eat phil 3 = 87.8746
Number of eating cycles phil 4 = 3
Total waiting time to eat phil 4 = 45.7055
Average time waiting to eat: 57.7602

End of simulation
Concurrent philosophers problem Circular Wait
Deadlock Avoidance

Requires that the system have additional a priori information available.

• Each process declares the maximum need of resources.

• The algorithm dynamically examines the resource-allocation state to ensure non-existence of circular-wait.

• The system is kept in a safe state.
Resource Allocation State

The current resource-allocation state is defined by the following data structures:

• AVAILABLE, a vector of (current) available resources units, for each resource type
• A[i], allocated resource units for process Pi of each resource type
• Max[i], maximum resource demand of process Pi, of each resource type
Informal Definition of Safe State

- Banker’s algorithm
- 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
- A system is in a safe state only if there exits a safe sequence of allocations
Safe Sequence of Resource Allocation

• A system is in a safe state if a *safe allocation sequence* of processes \(<P_1, P_2, \ldots, P_n>\) exists for the current resource allocation state

• A sequence of processes is a *safe sequence* for a given resource allocation state if for each process \(i\), its maximum resource request can be satisfied by the current resources plus the total resources held by each \(P_j\) with \(0 < j < i\)
Unsafe State

• If a system is in an unsafe state, then some sequence of requests may lead “unavoidably” to a deadlock (i.e., the deadlock cannot be avoided by delaying the requests from the processes).

• An unsafe state is not necessarily a deadlock state and does not necessarily lead to deadlock state.
Banker’s Algorithm

- There exist multiple instances of each resource type
- Each process must a priori claim maximum use of resources
- 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 period of time.
Banker’s Algorithm (2)

When a request for resources is made by process Pi:

• If \( \text{Request}_i \leq \text{Need}_i \) then goto step 2. Otherwise there is an error since Pi has exceeded its maximum claim.

• If \( \text{Request}_i \leq \text{available} \) then goto step 3, otherwise Pi must wait.

• The OS pretends to have allocated the requested resources to Pi by modifying the state as follows:
  
  \[
  \text{AVAILABLE} := \text{AVAILABLE} - \text{Request}_i \\
  \text{Allocation}_i := \text{Allocation}_i + \text{Request}_i \\
  \]

• If the resulting state is safe, Pi is allocated the resources. However, if the new state is unsafe, then Pi must wait and the state is restored.
Data Structures Used

- N is the number of processes, and m the resource types
- Available, a vector of length m indicating the number of available resources of each type.
- Max, an n x m matrix defining the maximum demand of each process, example: Max [i,j] = k.
- Allocation, an n x m matrix defining the number of resources of each type currently allocated to each process.
- Need, an (n x m) matrix for the remaining resource need of each process. Need [i, j] = max [i, j] - allocation [i,j].
- We can treat each row in the matrices Allocation and Need as vectors, and associate them with process Pi.
Description of Banker’s Algorithm

• A sequence \( s = <s_1, s_2, \ldots, s_n> \) of all processes is safe sequence for a given resource allocation state if for \( 0 < i \leq n \), \( \text{Max} [s_i] \leq \text{AVAILABLE} + \text{sum of } A[S_j], \ 0 < j \leq i. \)

• If the resource need of process \( P_i \) is not available, then \( P_i \) could wait until all \( P_j \) has finished.
Example 1

A system exhibits the following state: there are 3 processes P0, P1, P2; total system resources: 12 units of magnetic tape.

<table>
<thead>
<tr>
<th>Process</th>
<th>Max_Need</th>
<th>Allocation</th>
</tr>
</thead>
<tbody>
<tr>
<td>P0</td>
<td>10</td>
<td>5</td>
</tr>
<tr>
<td>P1</td>
<td>4</td>
<td>2</td>
</tr>
<tr>
<td>P2</td>
<td>9</td>
<td>2</td>
</tr>
</tbody>
</table>

Find out whether the system is in a safe state.
Solution Procedure

1. Compute the resources currently available, which is $(12 - 9) = 3$

2. Build the Need column with the numbers of resources that processes currently need to reach the maximum claim. These are 5, 2, and 7 (for P1, P2, and P3 respectively)

3. Select a process that can proceed to acquire its current need of resources

4. When the process releases the resources, compute the total number of available resources

5. Repeat from step 3
Final Solution

• Safe sequence of allocations found: 
  <P1, P0, P2>
• Therefore, the given state is a **safe** state.
Example 2

Assume that when in state 1, the OS allocates 1 more unit of tape. The new state is:

<table>
<thead>
<tr>
<th>Process</th>
<th>Max_Need</th>
<th>Allocations</th>
<th>Need</th>
</tr>
</thead>
<tbody>
<tr>
<td>P0</td>
<td>10</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>P1</td>
<td>4</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>P2</td>
<td>9</td>
<td>3</td>
<td>6</td>
</tr>
</tbody>
</table>

Resources currently available: 2
Safe State?

• After process P1 completes and releases all the resources it held, the total resources available is 4, which is not sufficient to satisfy the needs of P0 or P2.
• Therefore, the state given is **unsafe**, no safe sequence of allocations is possible.
Example 3

A system has five processes \{PO, P1, \ldots, P4\} and three resource types \{A, B, C\}, with 10, 5, and 7 instances respectively.

<table>
<thead>
<tr>
<th>PROCESS</th>
<th>ALLOCATION</th>
<th>MAX</th>
<th>NEED</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>A B C</td>
<td>A B C</td>
<td>A B C</td>
</tr>
<tr>
<td>P0</td>
<td>0 1 0</td>
<td>7 5 3</td>
<td>7 4 3</td>
</tr>
<tr>
<td>P1</td>
<td>2 0 0</td>
<td>3 2 2</td>
<td>1 2 2</td>
</tr>
<tr>
<td>P2</td>
<td>3 0 2</td>
<td>9 0 2</td>
<td>6 0 0</td>
</tr>
<tr>
<td>P3</td>
<td>2 1 1</td>
<td>2 2 2</td>
<td>0 1 1</td>
</tr>
<tr>
<td>P4</td>
<td>0 0 2</td>
<td>4 3 3</td>
<td>4 3 1</td>
</tr>
</tbody>
</table>
Solution

The system is currently in a safe state. The sequence: \(<P1, P3, P4, P2, P0>\) satisfies the safety criteria.
Example 4 of Banker’s Algorithm

Suppose P1 requests (1,0,2). The OS needs to check if (1,0,2) \leq (3,3,2), which is true. The OS then pretends to grant this request, and determine if the new state is a safe state.

The sequence <P1, P3, P4, P0, P2> is a safe sequence. Can request for (3,3,0) by P4 be granted? Can request for (0,2,0) by P0 be granted?
Deadlock Detection

- Allow system to enter deadlock state
- Run detection algorithm
- Recovery Scheme
Detection and Recovery

A detection and recovery scheme requires overhead that includes:

• Maintaining information and running the detection algorithm

• The potential losses inherent in recovering from deadlock

© 2012 Jones & Bartlett Learning, LLC
www.jblearning.com
Deadlock Detection

A detection algorithm is invoked periodically to determine whether a deadlock has occurred.

• A system has a deadlock if and only if it is impossible to satisfy the resource requests of some processes

• Formally, a system is in a deadlock-free state if there exists a deadlock-free sequence for the current resource allocation state.
Detection Approach

• The operating system can check for deadlock every time a resource is allocated; this is early detection.

• Other algorithms are used to detect cycles in the resource allocation graph. The techniques are based on incremental changes to the system state.

• The algorithms attempt to find a process with its resource requests that can be allocated with the currently available resources, the resources are allocated to the process, the resources are used by the process, and then the resources are released. This is repeated with the other processes.

• The processes that are in deadlock are identified and a procedure is initiated to stop deadlock.
Frequency of Running Deadlock Detection

• The deadlock detection algorithm can be invoked every time a process requests a resource that cannot be immediately granted (allocated). In this case, the operating system can directly identify the specific process that caused deadlock, in addition to the set of processes that are in deadlock.

• The detection algorithm is invoked periodically using a period not too long or not too short. Since deadlock reduces the CPU utilization and the system throughput, detection algorithm may be invoked when the CPU utilization drops below 40%.
System Recovery

• Abort all deadlocked processes
• Abort processes one at a time
• Preempt resources one at a time
• Rollback operation
Recovery: Process Termination

• Abort all deadlocked processes.
• Abort one process at a time until the deadlock cycle is eliminated.
• In which order should processes be aborted?
  – 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 (2)

- Selection of resources for preemption and processes to abort - successively preempt some resources from processes and allocate these resources to other process until the deadlock cycle is broken.
- Rollback: the system saves snapshots of a process’ state at periodic intervals. To break a deadlock, the system aborts a process and later resume it at the most recent checkpoint; (technique also useful in system crashes).
Deadlock Recovery (3)

- Process termination - abort (terminate) one or more processes to break the circular-wait condition.
- **Starvation** is possible, the same process is selected from which to preempt resources, and it never completes.
Deadlock Detection Issues

- How often does a deadlock occur?
- How many processes will be affected by deadlock, when it happens? It is possible that the last request which could not be granted, completes a chain of waiting processes?
- Invoke the detection algorithm whenever the CPU utilization drops below 40%.
- If the detection algorithm is invoked at arbitrary points in time, there may be many cycles in the resource graph.
Combined Approach to Deadlock

- Resources are partitioned into hierarchically ordered classes within each class the most appropriate technique is used. Consider a system which consists of the following classes of resources:
  - Internal resources. (used by the OS, e.g. PCBs).
  - Central memory. Technique: preemption
  - Job resources.Assignable devices and files. Technique: avoidance.
Safety Algorithm

To find out whether a system is in a safe state or not:

1. Initialize \( \text{WORK} := \text{AVAILABLE} \), vector of length \( M \)
   \( \text{GRANT}[ip] := \text{FALSE} \) for all \( ip = 1, 2, ..., N \).

2. Find \( i \) such that:
   a. \( \text{GRANT}[i] = \text{FALSE} \), and
   b. \( \text{NEED} \geq \text{WORK} \).
   If no such \( i \) exists, go to step 4.

3. \( \text{WORK} := \text{WORK} + \text{ALLOCATION}[i] \)
   \( \text{GRANT}[i] := \text{True} \)
   go to step 2.

4. If \( \text{GRANT}[i] = \text{True} \) for all \( i \), then the OS is in a safe state.
Peterson’s Algorithm

Initially, flag[0], flag[1] = false, turn = 0

Entry Section:
  flag [i] := true;
  j := (i + 1) mod 2;
  turn := j;
  While ( flag[j] and turn = j) do skip;

Critical Section:
  .
  .

Exit Section:
  flag[i]:= false;