Deadlocks
The Deadlock Problem (1)

- A set of blocked processes each holding a resource and waiting to acquire a resource held by another process in the set.

Example
- System has 2 disk drives.
- $P_1$ and $P_2$ each hold one disk drive and each needs another one.

Example
- semaphores $A$ and $B$, initialized to 1

```
P_0
 wait (A);
 wait (B);

P_1
 wait (B);
 wait (A)
```
The Deadlock Problem (2)

- Traffic only in one direction
- Each section of a bridge can be viewed as a resource
- If a deadlock occurs, it can be resolved if one car backs up (preempt resources and rollback)
- Several cars may have to be backed up if a deadlock occurs
- Starvation is possible
System Model

- Resource types $R_1, R_2, \ldots, R_m$
  - CPU cycles, memory space, I/O devices
- Each resource type $R_i$ has $W_i$ instances.
- Each process utilizes a resource as follows:
  - request
  - use
  - release
Deadlock Characterization (1)

• Necessary and sufficient conditions for a deadlock
  ◦ Mutual exclusion: only one process at a time can use a resource
  ◦ Hold and wait: a process holding at least one resource is waiting to acquire additional resources held by other processes
  ◦ No preemption: a resource can be released only voluntarily by the process holding it, after that process has completed its task
  ◦ Circular wait: there exists a set \{P0, P1, ..., Pn\} of waiting processes such that P0 is waiting for a resource that is held by P1, P1 is waiting for a resource that is held by P2, ..., Pn–1 is waiting for a resource that is held by Pn, and P0 is waiting for a resource that is held by P0
Deadlock Characterization (2)

- Resource-allocation graph

If no cycle exists then there is no deadlock
(necessary condition)
If a cycle exists then there may be a deadlock
(a cycle is a necessary and sufficient condition for deadlock when
resources involved have a single instance)
```java
class A implements Runnable {
    private Lock first, second;

    public A(Lock first, Lock second) {
        this.first = first;
        this.second = second;
    }

    public void run() {
        try {
            first.lock();
            // do something
            second.lock();
            // do something else
        }
        finally {
            first.unlock();
            second.unlock();
        }
    }
}

class B implements Runnable {
    private Lock first, second;

    public A(Lock first, Lock second) {
        this.first = first;
        this.second = second;
    }

    public void run() {
        try {
            second.lock();
            // do something
            first.lock();
            // do something else
        }
        finally {
            second.unlock();
            first.unlock();
        }
    }
}
```
Methods for Handling Deadlocks (1)

- Use a protocol that prevents or avoids deadlocks
  - Deadlock prevention: ensure that one of the necessary conditions cannot hold
  - Deadlock avoidance: keep track of available and allocated resources and future requests and releases to decide whether the current request should be satisfied or not
- Allow deadlocks to occur but detect them and recover
- Ignore the problem
  - Most popular approach (deadlocks tend to be rare)
    - Unix, Windows, Java
  - Manual restart to recover (mechanism may be useful in other situations too)
- Different approaches may be preferable for different resources
Methods for Handling Deadlocks (2)

- Handling deadlocks in Java
  - `suspend()`, `resume()`, `stop()`

```java
public class ClockApplet extends Applet implements Runnable {
    private Thread clockThread;
    private boolean ok = false;
    private Object mutex = new Object();

    public void run() {
        while (true) {
            try {
                // sleep for 1 second
                Thread.sleep(1000);

                // repaint the date and time
                repaint();

                // see if we need to suspend ourself
                synchronized (mutex) {
                    while (ok == false)
                        mutex.wait();
                }
            }
            catch (InterruptedException e) { }
        }
    }

    public void start() { // Figure 7.7
        ok = true;
        if (clockThread == null) {
            clockThread = new Thread(this);
            clockThread.start();
        }
        else {
            synchronized(mutex) {
                mutex.notify();
            }
        }
    }

    // this method is called when we leave the page the applet is on
    public void stop() { // Figure 7.7
        ok = false;
    }

    public void paint(Graphics g) {
        g.drawString(new java.util.Date().toString(), 10, 30);
    }
}
```
Deadlock Prevention

- **Mutual exclusion**
  - Sharable and non-sharable resources
  - For non-sharable resources this is not an option

- **Hold and wait**
  - Request all resources at the beginning
  - Request resource only when you hold none
  - - poor resource utilisation
  - - starvation is possible

- **No preemption**
  - Preempt all currently held resources when a request cannot be immediately granted
  - Preempt resource currently held by a waiting process
  - Only applicable to resources whose state can be easily saved and restored

- **Circular wait**
  - Impose total ordering on resources, acquire resources in ascending order (all instances at the same time)
    - Follow the normal order of usage
  - Works only if developers follow the protocol!
    - Witness: lock order verifier
Deadlock Avoidance (I)

- 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
  - If there exists a sequence \( <P_1, P_2, \ldots, P_n> \) of ALL the processes in the systems such that for each \( P_i \), the resources that \( P_i \) can still request can be satisfied by currently available resources + resources held by all the \( P_j \), with \( j < i \)
    - An unsafe state may lead to a deadlock
    - As long as the state is safe, the OS can avoid unsafe states (and deadlocks)
  - resource utilisation may be low
Deadlock Avoidance (2)

- Resource allocation graph
  - System with only one instance of each resource
  - Claim edges: dashed line, the process may request the resource in the future
    - Claim edge converts to request edge when a process requests a resource
    - Request edge converted to an assignment edge when the resource is allocated to the process
    - When a resource is released by a process, assignment edge reconverts to a claim edge
    - Resources must be claimed a priori in the system
  - Requests 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
    - Cycle detection algorithm of order $n^2$ in the number of processes
Deadlock Avoidance (3)

P2 request R2
Deadlock Avoidance (4)

- **Banker’s algorithm**
  - Multiple instances of each resource type
  - Less efficient than the resource allocation graph algorithm
  - Let n = number of processes, and m = number of resources types
    - **Available**: Vector of length m. If available \([j] = k\), there are \(k\) instances of resource type \(R_j\) available.
    - **Max**: \(n \times m\) matrix. If \(Max[i,j] = k\), then process \(P_i\) may request at most \(k\) instances of resource type \(R_j\).
    - **Allocation**: \(n \times m\) matrix. If \(Allocation[i,j] = k\) then \(P_i\) is currently allocated \(k\) instances of \(R_j\).
    - **Need**: \(n \times m\) matrix. If \(Need[i,j] = k\), then \(P_i\) may need \(k\) more instances of \(R_j\) to complete its task.

\[
Need[i,j] = Max[i,j] - Allocation[i,j]
\]
Deadlock Avoidance (5)

- Safety algorithm \( m \times n^2 \)

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

2. Find and \( i \) 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 \( \text{Finish}[i] == \text{true} \) for all \( i \), then the system is in a safe state.
Deadlock Avoidance (6)

Request \(_i\) = request vector for process \(P_i\).
If \(\text{Request}_i[j] = k\) then process \(P_i\) wants \(k\) instances of resource type \(R_j\).

1. If \(\text{Request}_i \leq \text{Need}_i\) go to step 2. Otherwise, raise error condition, since process has exceeded its maximum claim.

2. If \(\text{Request}_i \leq \text{Available}\), 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:

\[
\begin{align*}
\text{Available} & = \text{Available} - \text{Request}; \\
\text{Allocation}_i & = \text{Allocation}_i + \text{Request}_i; \\
\text{Need}_i & = \text{Need}_i - \text{Request}_i;
\end{align*}
\]

- If safe \(\Rightarrow\) the resources are allocated to \(P_i\)
- If unsafe \(\Rightarrow\) \(P_i\) must wait, and the old resource-allocation state is restored
Deadlock Avoidance (7)

- Example
  - 5 processes P0 through P4;
  - 3 resource types: A (10 instances), B (5 instances), and C (7 instances)

Snapshot at time T0:

<table>
<thead>
<tr>
<th></th>
<th>Allocation</th>
<th>Max</th>
<th>Available</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>3  3  2</td>
</tr>
<tr>
<td>P1</td>
<td>2  0  0</td>
<td>3  2  2</td>
<td></td>
</tr>
<tr>
<td>P2</td>
<td>3  0  2</td>
<td>9  0  2</td>
<td></td>
</tr>
<tr>
<td>P3</td>
<td>2  1  1</td>
<td>2  2  2</td>
<td></td>
</tr>
<tr>
<td>P4</td>
<td>0  0  2</td>
<td>4  3  3</td>
<td></td>
</tr>
</tbody>
</table>
Deadlock Avoidance (8)

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>P0</td>
<td>7</td>
<td>4</td>
<td>3</td>
</tr>
<tr>
<td>P1</td>
<td>1</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>P2</td>
<td>6</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>P3</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>P4</td>
<td>4</td>
<td>3</td>
<td>1</td>
</tr>
</tbody>
</table>

The system is in a safe state since the sequence < P1, P3, P4, P2, P0> satisfies safety criteria.
Deadlock Avoidance (9)

P1 Request (1,0,2)
Check that Request ≤ Available (that is, (1,0,2) ≤ (3,3,2) ⇒ 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>A B C</td>
</tr>
<tr>
<td>P0</td>
<td>0 1 0</td>
<td>7 4 3</td>
</tr>
<tr>
<td>P1</td>
<td>3 0 2</td>
<td>0 2 0</td>
</tr>
<tr>
<td>P2</td>
<td>3 0 1</td>
<td>6 0 0</td>
</tr>
<tr>
<td>P3</td>
<td>2 1 1</td>
<td>0 1 1</td>
</tr>
<tr>
<td>P4</td>
<td>0 0 2</td>
<td>4 3 1</td>
</tr>
</tbody>
</table>

Executing safety algorithm shows that sequence < P1, P3, P4, P0, P2> satisfies safety requirement.

- Can request for (3,3,0) by P4 be granted?
- Can request for (0,2,0) by P0 be granted?
Deadlock Detection (1)

- Single instance of each resource type
  - Wait-for graph
    - Nodes are processes
    - $P_i \rightarrow P_j$ if $P_i$ is waiting for $P_j$
  - 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
Deadlock Detection (2)

- Several instance of each resource type
  - **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 $\text{Request}_{ij} = k$, then process $P_i$ is requesting $k$ more instances of resource type $R_j$
  - Detection algorithm
    1. Let $\text{Work}$ and $\text{Finish}$ be vectors of length $m$ and $n$, respectively. Initialize:
       (a) $\text{Work} = \text{Available}$
       (b) For $i = 1, 2, \ldots, n$, if $\text{Allocation}_i \neq 0$, then $\text{Finish}[i] = \text{false}$; otherwise, $\text{Finish}[i] = \text{true}$.
    2. Find an index $i$ such that both:
       (a) $\text{Finish}[i] == \text{false}$
       (b) $\text{Request}_{i} \leq \text{Work}$
       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 $\text{Finish}[i] == \text{false}$, for some $i$, $1 \leq i \leq n$, then the system is in deadlock state. Moreover, if $\text{Finish}[i] == \text{false}$, then $P_i$ is deadlocked.
**Deadlock Detection (3)**

- Five processes P0 through P4; three resource types A (7 instances), B (2 instances), and C (6 instances)

Snapshot at time T0:

<table>
<thead>
<tr>
<th></th>
<th>Allocation</th>
<th>Request Available</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>A B C</td>
<td>A B C</td>
</tr>
<tr>
<td>P0</td>
<td>0 1 0</td>
<td>0 0 0</td>
</tr>
<tr>
<td>P1</td>
<td>2 0 0</td>
<td>2 0 2</td>
</tr>
<tr>
<td>P2</td>
<td>3 0 3</td>
<td>0 0 0</td>
</tr>
<tr>
<td>P3</td>
<td>2 1 1</td>
<td>1 0 0</td>
</tr>
<tr>
<td>P4</td>
<td>0 0 2</td>
<td>0 0 2</td>
</tr>
</tbody>
</table>

Sequence <P0, P2, P3, P1, P4> will result in Finish[i] = true for all i.
Deadlock Detection (4)

- P2 requests an additional instance of type C.

<table>
<thead>
<tr>
<th>Request A B C</th>
</tr>
</thead>
<tbody>
<tr>
<td>P0 0 0 0</td>
</tr>
<tr>
<td>P1 2 0 1</td>
</tr>
<tr>
<td>P2 0 0 1</td>
</tr>
<tr>
<td>P3 1 0 0</td>
</tr>
<tr>
<td>P4 0 0 2</td>
</tr>
</tbody>
</table>

State of system?

Can reclaim resources held by process P0, but insufficient resources to fulfill other processes; requests

Deadlock exists, consisting of processes P1, P2, P3, and P4.
Deadlock Detection (5)

- When, and how often, to invoke depends on:
  - How often a deadlock is likely to occur?
  - How many processes will need to be rolled back?
    - one for each disjoint cycle

- If detection algorithm is invoked arbitrarily, there may be many cycles in the resource graph and so we would not be able to tell which of the many deadlocked processes “caused” the deadlock
Recovery from Deadlock

- **Process termination**
  - Abort all deadlocked processes
  - Abort one process at a time until the deadlock cycle is eliminated
  - In which order should we choose to abort?
    - Priority of the process
    - How long process has computed, and how much longer to completion
    - Resources the process has used
    - Resources process needs to complete
    - How many processes will need to be terminated
    - Is process interactive or batch

- **Resource preemption**
  - Selecting a victim — minimize cost
  - Rollback — return to some safe state, restart process for that state
  - Starvation — same process may always be picked as victim, include number of rollback in cost factor
Consider the deadlock situation that could occur in the dining philosophers problem when the philosophers obtain the chopsticks one at a time. Discuss how the four necessary conditions for deadlock indeed hold in this setting. Discuss how deadlocks could be avoided by eliminating any one of the four conditions.

Compare the circular-wait scheme with the deadlock-avoidance schemes (like the banker’s algorithm) with respect to the following issues:

- Runtime overheads
- System throughput

Java’s locking mechanism (the synchronized statement) is considered reentrant. That is, if a thread acquires the lock for an object (by invoking a synchronized method or block), it can enter other synchronized methods or blocks for the same object. Explain how deadlock would be possible if Java’s locking mechanism were not reentrant.
For contemplation (2)

- Consider the traffic deadlock depicted in the figure.
  - Show that the four necessary conditions for deadlock indeed hold in this example.
  - State a simple rule for avoiding deadlocks in this system.

- Consider the following snapshot of a system:

<table>
<thead>
<tr>
<th></th>
<th>Allocation</th>
<th>Max</th>
<th>Available</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>A B C D</td>
<td>A B C D</td>
<td>A B C D</td>
</tr>
<tr>
<td>P0</td>
<td>0 0 1 2</td>
<td>0 0 1 2</td>
<td>1 5 2 0</td>
</tr>
<tr>
<td>P1</td>
<td>1 0 0 0</td>
<td>1 7 5 0</td>
<td></td>
</tr>
<tr>
<td>P2</td>
<td>1 3 5 4</td>
<td>2 3 5 6</td>
<td></td>
</tr>
<tr>
<td>P3</td>
<td>0 6 3 2</td>
<td>0 6 5 2</td>
<td></td>
</tr>
<tr>
<td>P4</td>
<td>0 0 1 4</td>
<td>0 6 5 6</td>
<td></td>
</tr>
</tbody>
</table>

Answer the following questions using the banker’s algorithm:
- What is the content of the matrix Need?
- Is the system in a safe state?
- If a request from process P1 arrives for (0,4,2,0), can the request be granted immediately?
For contemplation (3)

- Consider a system consisting of four resources of the same type that are shared by three processes, each of which needs at most two resources. Show that the system is deadlock-free.

- Consider a system consisting of \( m \) resources of the same type, being shared by \( n \) processes. Resources can be requested and released by processes only one at a time. Show that the system is deadlock free if the following two conditions hold:
  - The maximum need of each process is between 1 and \( m \) resources
  - The sum of all maximum needs is less than \( m + n \)