Deadlock

- Permanent blocking of a set of processes that either compete for system resources or communicate with each other
- No Efficient solution
- Involve conflicting needs for resources by two or more processes

Reusable Resources

- Description:
  - Used by one process at a time and not depleted by that use
  - Processes obtain resources that they later release for reuse by other processes
- Example:
  - Processor time, I/O channels, main and secondary memory, files, databases, and semaphores
  - Deadlock occurs if each process holds one resource and requests the other

Bridge Crossing Example

- 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.
**Consumable Resources**

- **Description:**
  - Created (produced) and destroyed (consumed) by a process
- **Examples:**
  - Interrupts, signals, messages, and information in I/O buffers
- **Deadlock:**
  - May occur if a Receive message is blocking
  - May take a rare combination of events to cause deadlock

**Consumable Resource Example**

- Deadlock occurs if receive is blocking

```
P1
  ... Receive(P2);
  Send(P2);

P2
  ... Receive(P1);
  Send(P1);
```

**System Model - Reusable Resources**

- 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

**Necessary Conditions for Deadlock**

- 1) Mutual exclusion:
  - One process hold a resource in a non-sharable mode. Other processes requesting resource must wait for resource to be released.
- 2) Hold-and-wait:
  - A process must hold at least one allocated resource while awaiting other resources (one or more) held by other processes.
**Necessary Conditions for Deadlock**

- **3) No preemption:**
  - No resource can be forcibly removed from a process holding it. That is, resources are voluntarily released by the process holding it.

- **4) Circular wait:**
  - A closed chain of processes exists, such that each process holds at least one resource needed by the next process in the chain.

---

**Resource-Allocation Graph**

- A set of Vertices $V$ and a set of Edges $E$
- $V$ is partitioned into two types:
  - $P = \{P_1, P_2, ..., P_n\}$, the set consisting of all the processes in the system.
  - $R = \{R_1, R_2, ..., R_m\}$, the set consisting of all resource types in the system.

- **request edge** - directed edge $P_i \rightarrow R_j$
- **assignment edge** - directed edge $R_j \rightarrow P_i$

---

**Example of a Resource Allocation Graph**

- **Process $P_i$**
- **Resource Type with 4 instances**
- $P_i$ requests instance of $R_j$
- $P_i$ is holding an instance of $R_j$
With A Deadlock

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.

Approaches to Deadlock Handling

• Ensure system never enters a deadlocked state:
  Use either:
  - deadlock prevention scheme - ensure that at least one of the necessary conditions cannot hold.
  - deadlock avoidance scheme - requires the OS know in advance the resource usage requirements of all processes. Then for each request the OS decides if it could lead to a deadlock before granting.

• Allow the system to enter a deadlock state and then recover.
  - system implements an algorithm for deadlock detection, if so then recover

• Assume deadlocks never occur in the system
  - used by most operating systems, including UNIX.
Deadlock Prevention

- Restrain ways a request can be made:
  - Mutual Exclusion - not required for sharable resources; must hold for nonsharable resources.
  - Hold-and-Wait - must guarantee that whenever a process requests a resource, it does not hold any other resources.
    - Require process to request and be allocated all its resources before it begins execution, or allow process to request resources only when the process has none.
    - Low resource utilization; starvation possible.

Deadlock Prevention - Indirect

- No Preemption -
  - 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.

- Circular Wait - impose a total ordering of all resource types, and require that each process requests resources in an increasing order of enumeration.

Deadlock Avoidance

- Require system has a priori information
- 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

- 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 $<P_1, P_2, ..., P_n>$ 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 $P_i$ resource needs are not immediately available, then $P_i$ can wait until all $P_j$ have finished.
  - When $P_i$ 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.

Resource-Allocation Graph Algorithm

- Claim edge $P_i \rightarrow R_j$ indicated that process $P_i$ may request resource $R_j$ represented by a dashed line.
- Claim edge converts to request edge when a process requests a resource.
- When a 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: Deadlock Avoidance

Unsafe State: Resource-Allocation Graph
**Process Initiation Denial**

### Definitions

<table>
<thead>
<tr>
<th>Resource</th>
<th>Available</th>
<th>Max</th>
<th>Allocation</th>
<th>Need</th>
</tr>
</thead>
<tbody>
<tr>
<td>( (R_1, R_2, ..., R_m) )</td>
<td>( (V_1, V_2, ..., V_m) )</td>
<td>( M = \begin{bmatrix} M_{11} &amp; M_{12} &amp; \cdots &amp; M_{1n} \ M_{21} &amp; M_{22} &amp; \cdots &amp; M_{2n} \ \vdots &amp; \vdots &amp; \ddots &amp; \vdots \ M_{m1} &amp; M_{m2} &amp; \cdots &amp; M_{mn} \end{bmatrix} )</td>
<td>( A = \begin{bmatrix} A_{11} &amp; A_{12} &amp; \cdots &amp; A_{1n} \ A_{21} &amp; A_{22} &amp; \cdots &amp; A_{2n} \ \vdots &amp; \vdots &amp; \ddots &amp; \vdots \ A_{m1} &amp; A_{m2} &amp; \cdots &amp; A_{mn} \end{bmatrix} )</td>
<td>( N = \max - A )</td>
</tr>
</tbody>
</table>

### Relationships

1. \( R_i = V_i + \sum_{k=1}^{n} A_{ki} \), for all \( i \)
2. \( M_{ki} \leq R_i \), for all \( k, i \)
3. \( A_{ki} \leq M_{ki} \), for all \( k, i \)

**Deadlock Avoidance Policy**

- Start if the max claim of all process can be met (including new process)
- Start a new process \( P_{n+1} \) only if the below inequality holds

\[
R_i \geq M_{(n+1)i} + \sum_{k=1}^{n} M_{ki} \text{ for all } i
\]

---

**Resource Allocation Denial**

- Banker's Algorithm
  - Multiple instances.
  - Each process must a priori claim maximum use: can not exceed the total number of resources in the system.
  - When a process requests a resource it may have to wait.
  - When a process gets all its resources it must return them in a finite amount of time.

**Safety Algorithm**

1. Let \( Work \) and \( Finish \) be vectors of length \( m \) and \( n \), respectively. Initialize:
   - \( Work = Available \)
   - \( Finish[i] = false \) for \( i = 1, 2, \ldots, n \).
2. Find an \( i \) (that is, a \( P_i \)) such that both:
   - \( Finish[i] = false \), \( i \) has not completed
   - \( Need[i] \leq Work[i] \), can allocate remaining Resources
   - If no such \( i \) exists, go to step 4.
3. Set \( Work = Work + Allocation[i] \)
   - Set \( Finish[i] = True \)
   - go to step 2.
4. If \( Finish[i] = True \) for all \( i \), then the system is in a safe state. Otherwise it is unsafe.
Resource-Request Algorithm for $P_i$

If $\text{Request}_{i[j]} = k$, then process $P_i$ wants $k$ instances of resource type $R_j$.

1. If $\text{Request}_{i} \leq \text{Need}_{i}$, then go to step 2. Otherwise, raise error condition, (process exceeded its claim).
2. If $\text{Request}_{i} \leq \text{Available}$, then go to step 3. Otherwise $P_i$ must wait, since resources are not available.
3. Pretend to allocate requested resources to $P_i$:
   \[
   \begin{align*}
   \text{Available} &= \text{Available} - \text{Request}_{i} \\
   \text{Allocation}_{i} &= \text{Allocation}_{i} + \text{Request}_{i} \\
   \text{Need}_{i} &= \text{Need}_{i} - \text{Request}_{i}
   \end{align*}
   \]
   • 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.

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 $\text{Max} - \text{Allocation}$.

<table>
<thead>
<tr>
<th>Need</th>
<th>$A$</th>
<th>$B$</th>
<th>$C$</th>
</tr>
</thead>
<tbody>
<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 $P_1$ Request (1,0,2)

- Check that $\text{Request}_{1} \leq \text{Available}$
  - (that is, $(1,0,2) \leq (3,3,2) => true$.

<table>
<thead>
<tr>
<th>Allocation</th>
<th>Need</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>3</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>

- Executing safety algorithm shows that sequence $<P_1, P_3, P_4, P_2, P_0>$ satisfies safety requirement.
- Can request for $(3,3,0)$ by $P_4$ be granted?
- Can request for $(0,2,0)$ by $P_2$ be granted?
Deadlock Detection

- Allow system to enter deadlock state
- Detection algorithm
- Recovery scheme

Single Instance each Resource Type

- Maintain 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.
- 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 and Wait-for Graphs

- 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 \( \text{Request}[i,j] = k \), then process \( P_i \) is requesting \( k \) more instances of resource type \( R_j \).

Several Instances ea Resource Type
Detection Algorithm

1. Let Work and Finish be vectors of length \( m \) and \( n \), respectively. Initialize:
   (a) Work = Available
   (b) For \( i = 1, 2, \ldots, n \), if Allocation\(_i\) ≠ 0, then
       Finish\(_i\) = false; otherwise, Finish\(_i\) = true.
2. Find an index \( i \) such that both:
   (a) Finish\(_i\) = false // has allocated resources
   (b) Request\(_i\) ≤ Work // request can be filled
   If no such \( i \) exists, go to step 4.

3. Work = Work + Allocation, Finish\(_i\) = true
   go to step 2.

4. If Finish\(_i\) = false, for some \( i, 1 ≤ i ≤ n \), 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), B (2), and C (6).
• Snapshot at time \( T_0 \):

<table>
<thead>
<tr>
<th>Allocation</th>
<th>Request</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>( P_0 ) 0 1 0</td>
<td>0 0 0</td>
<td>0 0 0</td>
</tr>
<tr>
<td>( P_1 ) 2 0 0</td>
<td>2 0 2</td>
<td></td>
</tr>
<tr>
<td>( P_2 ) 3 0 3</td>
<td>0 0 0</td>
<td></td>
</tr>
<tr>
<td>( P_3 ) 2 1 1</td>
<td>1 0 0</td>
<td></td>
</tr>
<tr>
<td>( P_4 ) 0 0 2</td>
<td>0 0 2</td>
<td></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 (Cont.)

• \( P_2 \) requests an additional instance of type C.

<table>
<thead>
<tr>
<th>Request</th>
<th>A B C</th>
</tr>
</thead>
<tbody>
<tr>
<td>( P_0 )</td>
<td>0 0 0</td>
</tr>
<tr>
<td>( P_1 )</td>
<td>2 0 2</td>
</tr>
<tr>
<td>( P_2 )</td>
<td>0 0 1</td>
</tr>
<tr>
<td>( P_3 )</td>
<td>1 0 0</td>
</tr>
<tr>
<td>( P_4 )</td>
<td>0 0 2</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 \( P_4 \).
Detection-Algorithm Usage

• 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: Process Termination

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

Recovery: 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.

Combined Approach

• Combine the three basic approaches
  - prevention
  - avoidance
  - detection
  allowing the use of the optimal approach for each of resources in the system.

• Partition resources into hierarchically ordered classes.

• Use most appropriate technique for handling deadlocks within each class.