Deadlock

Chapter 6
Deadlock

- A set of processes is deadlocked when each process in the set is blocked awaiting an event that can only be triggered by another blocked process in the set.
- Permanent blocking of a set of processes that either compete for system resources or communicate with each other.
- Involves conflicting needs for resources by two or more processes.
- Bad news: There is no satisfactory solution in the general case.
Example: Potential Deadlock

I need quad C and D

I need quad D and A

I need quad B and C

I need quad A and B
Example: Potential Deadlock
Four Conditions for Deadlock

- **Mutual exclusion**
  - Only one process may use a resource at a time

- **Hold-and-wait**
  - A process may hold allocated resources while awaiting assignment of others

- **No preemption**
  - No resource can be forcibly removed from a process holding it

- **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
Four Conditions for Deadlock

- Deadlock occurs if and only if the circular wait condition is unresolvable.
- The circular wait condition is unresolvable when the first 3 policy conditions hold.
- Thus the 4 conditions taken together constitute necessary and sufficient conditions for deadlock.

**Possibility of Deadlock**
1. Mutual exclusion
2. No preemption
3. Hold and wait

**Existence of Deadlock**
1. Mutual exclusion
2. No preemption
3. Hold and wait
4. Circular wait
Resource Allocation Graph

- Directed graph that depicts a state of the system of resources and processes
- Process is represented by circle
- Resource is represented by square

(a) Resource is requested
(b) Resource is held
Example of Resource Allocation Graphs
Dealing with Deadlock

- Three general approaches exist for dealing with deadlock
  - Prevent deadlock: disallow 1 of the 4 necessary conditions of deadlock occurrence
  - Avoid deadlock: do not grant a resource request if this allocation might lead to deadlock
  - Detect deadlock: always grant resource request when possible, but periodically check for the presence of deadlock and then recover from it
Deadlock Prevention

- Idea: invalidate one of the four conditions for deadlock (not a dynamic process)
  - Mutual exclusion
  - Hold-and-wait
  - No preemption
  - Circular wait

- Indirect methods: attacking the first three conditions

- Direct methods: attacking the last condition (e.g., circular wait)
Attacking Mutual Exclusion

- Some devices (such as printer) can be spooled
  - Only the printer daemon uses printer resource
  - Thus deadlock for printer eliminated
- In general, we cannot (or don’t want to remove mutual exclusion), and not all devices can be spooled

Principle
- Avoid assigning resource when not absolutely necessary (e.g., consider how mutex lock for critical sessions)
- As few processes as possible actually claim the resource
Attacking Hold and Wait

- Require processes to request and get all resources before starting
  - A process never has to wait for what it needs

- Problems
  - May not know required resources at start of run
  - May wait for long to acquire all resources
  - Ties up resources other process could be using

- Variation: before requesting a new resource
  - Process must give up all previous resources
  - Then request all immediately need resources
Attacking No Preemption

- As we have discussed, disabling preemption is not a viable option
- Consider a process given the printer
  - Halfway through its job of printing a page
  - Now forcibly take away the printer
  - No a solution!!
Attacking Circular Wait

- There is a solution to attack circular wait:
  - Every resource has a unique number
  - A process must request resources in an increasing number order
  - For example:
    - A process initially request a $R_i$ with an index $O(R_i)$
    - After that, the process can request another resource $R_j$ if and only if $O(R_j) > O(R_i)$
  - This protocol prevents deadlock but will often deny resources unnecessarily (inefficient) because of the ordering imposed on the requests
Deadlock Avoidance

- A decision is made *dynamically* whether the current resource allocation requires will, if granted, potentially lead to a deadlock.

- Allows more concurrency than prevention, but require knowledge of future process requests.

- Two approaches:
  - **Process Initiation Denial**: do not start a process if its demand might lead to deadlock.
  - **Resource Allocation Denial**: do not grant an incremental resource request if this allocation might lead to deadlock.

- In both cases: maximum requirements of each resource must be stated in advance.
Resources in a system are “partitioned” in resources types

Each resource type in a system exists with a certain amount. Let $R(i)$ be the total amount of resource type $i$ present in the system. E.g.:

- $R(\text{main memory}) = 128$ MB
- $R(\text{disk drives}) = 8$
- $R(\text{printers}) = 5$

The partition is system specific (e.g.: disk drives may be further partitioned...).
Let $C(k,i)$ be the amount of resource type $i$ claimed by process $k$.

To be admitted in the system, process $k$ must show $C(k,i)$ for all resource types $i$.

$C(k,i)$ is the maximum value of resource type $i$ permitted for process $k$.

Let $U(i)$ be the total amount of resource type $i$ unclaimed or available in the system:

$$U(i) = R(i) - \sum_k C(k,i)$$
A new process \( n \) is admitted in the system only if \( C(n,i) \leq U(i) \) for all resource type \( i \).

This policy ensures that deadlock is always avoided since a process is admitted only if all its requests can always be satisfied (no matter what will be their order).

Not optimal since it assumes the worst: that all processes will make their maximum claims together at the same time.
Resource Allocation Denial: the Banker’s Algorithm

- Metaphor: processes are like customers wanting to borrow money (resources) from a bank...
- A banker should not allocate cash when it cannot satisfy all the (future) needs of a customer, constrained by other customers
- Consider a system with fixed number of resources
  - **State** of the system is the current allocation of resources to processes (i.e., encoded by \( R(i) \), \( C(j,i) \), and others)
  - **Safe state** is where there is **at least one** sequence that does not result in deadlock
  - **Unsafe state** is the state that is not safe
The Banker’s Algorithm

- We also need the amount of resource type $i$ that are allocated to process $j$ for all $(j,i)$, denoted as $A(j,i)$.

- The total amount of available resource for each type $i$ is defined by $V(i)$ that satisfies $V(i) = R(i) - \sum_k A(k,i)$.

- We also use an intermediate variable “need $N(j,i)$” of resource type $i$ required by process $j$ to complete its task: $N(j,i) = C(j,i) - A(j,i)$.

- To decide if a resource request made by a process should be granted, the banker’s algorithm test if granting the request will lead to a safe state, by testing $C(j,i) - A(j,i) \leq V(i)$.

  - If the resulting state is safe then grant request; else do not grant the request.
# The Banker’s Algorithm

**Resource** $= \textbf{R} = (R_1, R_2, \ldots, R_m)$  
**Available** $= \textbf{V} = (V_1, V_2, \ldots, V_m)$

<table>
<thead>
<tr>
<th>Claim = $\textbf{C}$</th>
<th>Allocation = $\textbf{A}$</th>
<th>Total amount of each resource in the system</th>
<th>Total amount of each resource not allocated to any process</th>
</tr>
</thead>
<tbody>
<tr>
<td>$C_{ij}$</td>
<td>$A_{ij}$</td>
<td>$m$ is the number of resource types</td>
<td>$n$ is the number of processes</td>
</tr>
<tr>
<td></td>
<td>$A_{ij}$</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

$C_{ij}$ = requirement of process $i$ for resource $j$

$A_{ij}$ = current allocation to process $i$ of resource $j$

### Constraints

1. $R_j = V_j + \sum_{i=1}^{n} A_{ij}$, for all $j$  
   All resources are either available or allocated.

2. $C_{ij} \leq R_j$, for all $i, j$  
   No process can claim more than the total amount of resources in the system.

3. $A_{ij} \leq C_{ij}$, for all $i, j$  
   No process is allocated more resources of any type than the process originally claimed to need.
The Banker’s Algorithm

- The banker’s algorithm is all about the determination of a safe state.
- For example: a system consisting of 4 process and 3 resource types with the following configuration at a time point, *is this a safe state?*

![Diagram showing initial state of a system with matrices](image)

Note: These two are usually unknown and must be computed from others.
The Banker’s Algorithm

- We need to check \( C(j,i) - A(j,i) \leq V(i) \) for all \( j \).

- For P1 (i.e., \( j=1 \)), this is not possible for P1.
  - Which requests 2 units of R1, but no R1 is available.
  - Similarly, the system doesn’t have sufficient R2 and R3 for P1.

![Table of Resource Requirements]

![Table of Allocations and Available Resources]

Note: These two are usually unknown and must be computed from others.
The Banker’s Algorithm

- We need to check \( C(j,i) - A(j,i) \leq V(i) \) for all \( j \)
- For P2, if we assign 1 unit of R3 to P2
  - Then P2 has its maximum required resources allocated and can run to completion

### Resource Matrix

<table>
<thead>
<tr>
<th></th>
<th>R1</th>
<th>R2</th>
<th>R3</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1</td>
<td>3</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>P2</td>
<td>6</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>P3</td>
<td>3</td>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td>P4</td>
<td>4</td>
<td>2</td>
<td>2</td>
</tr>
</tbody>
</table>

Claim matrix \( C \)

### Allocation Matrix

<table>
<thead>
<tr>
<th></th>
<th>R1</th>
<th>R2</th>
<th>R3</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>P2</td>
<td>6</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>P3</td>
<td>2</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>P4</td>
<td>0</td>
<td>0</td>
<td>2</td>
</tr>
</tbody>
</table>

Allocation matrix \( A \)

### Available Vector

<table>
<thead>
<tr>
<th></th>
<th>R1</th>
<th>R2</th>
<th>R3</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>P2</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>P3</td>
<td>1</td>
<td>0</td>
<td>3</td>
</tr>
<tr>
<td>P4</td>
<td>4</td>
<td>2</td>
<td>0</td>
</tr>
</tbody>
</table>

Available vector \( V \)

Note: These two are usually unknown and must be computed from others.
The Banker’s Algorithm

- We need to check $C(j,i) - A(j,i) \leq V(i)$ for all $j$
- For P2, if we assign 1 unit of R3 to P2
  - Then P2 has its maximum required resources allocated and can run to completion
  - And it will return resources to the “available” pool

Note: P2 is completed
The Banker’s Algorithm

- We need to check $\text{C}(j,i) - \text{A}(j,i) \leq V(i)$ for all $j$
- After P2 runs to completion, can any of the remaining processes be completed?
  - Start over the checking again (from the beginning by convention)

Yes, now the available resources are sufficient for P1

Note: P2 is completed
The Banker’s Algorithm

- We need to check $C(j,i) - A(j,i) \leq V(i)$ for all $j$.
- After P1 completes:
  - The system has more resources, which are sufficient for both P3 and P4.
  - By convention, we check from the beginning of the process list, and P3 will start before P4.
The Banker’s Algorithm

- After P3 completes:
  - The system has even more resources, which can be all allocated to P4

Thus, the state defined originally is a safe state
If we accidently give P1 one additional unit of R1 and R3, is this new state safe?

Thus, this new state is unsafe.
Banker’s Algorithm: Comments

- **Pros:**
  - It is less restrictive than deadlock prevention
  - It is not necessary to preempt and rollback processes, as in deadlock detection

- **Cons:**
  - Maximum resource requirement must be stated in advance
  - Processes under consideration must be independent and with no synchronization requirements
  - There must be a fixed number of resources to allocate
  - No process may exit while holding resources
Deadlock Detection

- Deadlock prevention strategies are very conservative
  - Limit access to resources and impose restrictions on processes during the system design

- Deadlock detection strategies do the opposite
  - Resource requests are granted whenever possible
  - Regularly check for deadlock
Deadlock Recovery

- Needed when deadlock is detected.
- The following methods are possible:
  - Abort all deadlocked processes (one of the most common solution adopted in OS!!)
  - Rollback each deadlocked process to some previously defined checkpoint and restart them (original deadlock may reoccur)
  - Successively abort deadlock processes until deadlock no longer exists
  - Successively preempt resources until deadlock no longer exists