Outline

1. Deadlocks Overview
2. Deadlock Prevention
3. Deadlock Avoidance
4. Deadlock Detection & Recovery
5. Handling Deadlocks in Reality
What are Deadlocks

- **Deadlock**: Set of processes $P_1, \cdots, P_N$ deadlocked iff
  - every $P_i$ is blocked, and
  - every $P_i$ is waiting for an event doable only by some $P_j$
    // event: release, signal, V, interrupt enable, ...

- **Deadlock freedom**: desired property of multi-threaded programs
  - ensuring this is hard for a general multi-threaded program
  - but easier for a resource-manager system examined next

- **Aside**: **Livelock** is deadlock without blocking
  - processes are in fruitless loops
  - harder to detect (unless loops are very localized)
  - deadlock can be livelock at a lower (spin-lock) level
Resource Manager System

- System = resource manager + user processes
  - processes: request resources, get them, release them
  - $RES$: set of all resources, initially held by manager
  - $alloc(p)$: resources currently held by (user process) $p$
  - $avail$: resources currently held by manager

- Function $req(p, res)$: request by $p$ for resources $res$
  - call only if $res + alloc(p) \subseteq RES$
  - blocking call
  - $p$ gets $res$ at return; happens only if $res \subseteq avail$

- Function $rel(p, res)$: release by $p$ of $res$
  - call only if $res \subseteq alloc(p)$
  - nonblocking

- System can deadlock without further constraints
- 3 approaches: prevention, avoidance, detection/recovery
Outline

1. Deadlocks Overview
2. Deadlock Prevention
3. Deadlock Avoidance
4. Deadlock Detection & Recovery
5. Handling Deadlocks in Reality
Deadlock Prevention Approach

- Impose further constraints on \textit{req} calls to preclude deadlock
  - no further constraints on \textit{req} returns

- Step 1: identify a necessary condition for deadlock, eg:
  - resource that is non-shareable and non-preemptable
  - process holds a resource and requests more resources
  - cycle of processes: each requesting a resource held by the next

- Step 2: constrain \textit{req} calls to preclude a necessary condition

- Henceforth assume non-shareable/non-preemptable resources

- Examples of deadlock prevention rules
  - \textit{req}(p, res) can be called only when \textit{alloc}(p) is empty
  - Impose a total ordering on all resources in \textit{RES}
    \textit{req}(p, res) can be called only when \textit{res} > \textit{max(alloc}(p))
Outline

1. Deadlocks Overview
2. Deadlock Prevention
3. Deadlock Avoidance
4. Deadlock Detection & Recovery
5. Handling Deadlocks in Reality
Deadlock Avoidance Approach

- **Deadlock avoidance:**
  - Impose further constraints on `req` returns to preclude deadlock
  - So `req(p, res)` return may wait even if `res \subseteq avail`
  - May also involve weak constraints on `req` calls
    - E.g., limit on total resources that a process can hold
  - Can allow more parallelism than deadlock prevention
  - Burden is on manager (unlike deadlock prevention)

- Classical deadlock avoidance solution uses the “Banker’s algorithm”
Deadlock Avoidance Solution

- Resources: organized into types 1, ⋯ , M
  - \( Tot = [Tot_1, \cdots, Tot_M] \) \hspace{1cm} // total # of each resource type

- Processes: 1, ⋯ , N
  - \( Max_i: [Max_{i,1}, \cdots, Max_{i,M}] \) \hspace{1cm} // max total need of process \( i \)

- Variables
  - \( alloc_i: [alloc_{i,1}, \cdots, alloc_{i,M}] \) \hspace{1cm} // resources held by process \( i \)
  - \( avail: [avail_1, \cdots, avail_M] \) \hspace{1cm} // resources held by manager
  - \( req_i: [req_{i,1}, \cdots, req_{i,M}] \) \hspace{1cm} // process \( i \)'s ongoing request
  - \( need_i: Max_i - alloc_i \) \hspace{1cm} // process \( i \)'s max possible request
- **Assumption:** If a process $i$ always gets the resources it asks for, it eventually releases all its resources.
- So if $\text{need}_i \leq \text{avail}$ and the manager grants only requests of $i$, then it eventually gets $\text{alloc}_i$ back.

- A state is **safe** iff it has a safe sequence.
- A **safe sequence** is a permutation $i_1, \ldots, i_N$ of process ids s.t.
  - $\text{need}_{i_1} \leq \text{avail}$
  - $\text{need}_{i_2} \leq \text{avail} + \text{alloc}_{i_1}$
  - $\ldots$
  - $\text{need}_{i_N} \leq \text{avail} + \text{alloc}_{i_1} + \cdots + \text{alloc}_{i_{N-1}}$

- A safe state is not deadlocked and cannot lead to a deadlock.
**Banker’s Algorithm:** determines whether or not a state is safe

- **Variables**
  
  - `xavail ← avail`  
  - `done[i] ← false, for i = 1, · · · , N`  

- While (there is an `i` s.t. `done[i] = false and need_i ≤ xavail`)
  
  - `xavail ← xavail + alloc_i`
  - `done[i] ← true`

- Safe iff `done[i] = true` for every `i`

- Return `req(p, res)` only if the resulting state would be safe, i.e., apply Banker’s algorithm to the current state with
  
  - `avail` decreased by `res`
  - `alloc_i` increased by `res`
Banker’s Algorithm Example

- 5 processes, 3 resource types
- $Tot: [10 \ 5 \ 7]$
- State

<table>
<thead>
<tr>
<th></th>
<th>Max</th>
<th>alloc</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1</td>
<td>7 5 3</td>
<td>0 1 0</td>
</tr>
<tr>
<td>P2</td>
<td>3 2 2</td>
<td>2 0 0</td>
</tr>
<tr>
<td>P3</td>
<td>9 0 2</td>
<td>3 0 2</td>
</tr>
<tr>
<td>P4</td>
<td>2 2 2</td>
<td>2 1 1</td>
</tr>
<tr>
<td>P5</td>
<td>4 3 3</td>
<td>0 0 2</td>
</tr>
</tbody>
</table>

- Safe?
Banker’s Algorithm Example

- 5 processes, 3 resource types
- \textit{Tot}: [10 5 7]
- State

<table>
<thead>
<tr>
<th></th>
<th>Max</th>
<th>alloc</th>
<th>need</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1</td>
<td>7 5 3</td>
<td>0 1 0</td>
<td>7 4 3</td>
</tr>
<tr>
<td>P2</td>
<td>3 2 2</td>
<td>2 0 0</td>
<td>1 2 2</td>
</tr>
<tr>
<td>P3</td>
<td>9 0 2</td>
<td>3 0 2</td>
<td>6 0 0</td>
</tr>
<tr>
<td>P4</td>
<td>2 2 2</td>
<td>2 1 1</td>
<td>0 1 1</td>
</tr>
<tr>
<td>P5</td>
<td>4 3 3</td>
<td>0 0 2</td>
<td>4 3 1</td>
</tr>
</tbody>
</table>

- Safe?
### Banker’s Algorithm Example

- 5 processes, 3 resource types
- **Tot:** \([10 \ 5 \ 7]\)

#### State

<table>
<thead>
<tr>
<th></th>
<th>Max</th>
<th>alloc</th>
<th>need</th>
<th>avail</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1</td>
<td>7 5 3</td>
<td>0 1 0</td>
<td>7 4 3</td>
<td></td>
</tr>
<tr>
<td>P2</td>
<td>3 2 2</td>
<td>2 0 0</td>
<td>1 2 2</td>
<td></td>
</tr>
<tr>
<td>P3</td>
<td>9 0 2</td>
<td>3 0 2</td>
<td>6 0 0</td>
<td></td>
</tr>
<tr>
<td>P4</td>
<td>2 2 2</td>
<td>2 1 1</td>
<td>0 1 1</td>
<td></td>
</tr>
<tr>
<td>P5</td>
<td>4 3 3</td>
<td>0 0 2</td>
<td>4 3 1</td>
<td></td>
</tr>
</tbody>
</table>

Safe?
Banker’s Algorithm Example

- 5 processes, 3 resource types
- **Tot:** [10 5 7]
- State

<table>
<thead>
<tr>
<th></th>
<th>Max</th>
<th>alloc</th>
<th>need</th>
<th>done</th>
<th>avail</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1</td>
<td>7 5 3</td>
<td>0 1 0</td>
<td>7 4 3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>P2</td>
<td>3 2 2</td>
<td>2 0 0</td>
<td>1 2 2</td>
<td>P2</td>
<td>5 3 2</td>
</tr>
<tr>
<td>P3</td>
<td>9 0 2</td>
<td>3 0 2</td>
<td>6 0 0</td>
<td>P4</td>
<td>7 4 3</td>
</tr>
<tr>
<td>P4</td>
<td>2 2 2</td>
<td>2 1 1</td>
<td>0 1 1</td>
<td>P5</td>
<td>7 4 5</td>
</tr>
<tr>
<td>P5</td>
<td>4 3 3</td>
<td>0 0 2</td>
<td>4 3 1</td>
<td>P1</td>
<td>7 5 5</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>P3</td>
<td>10 5 7</td>
</tr>
</tbody>
</table>

- Safe? Yes. Safe sequence: P2, P4, P1, P3
1. Deadlocks Overview
2. Deadlock Prevention
3. Deadlock Avoidance
4. Deadlock Detection & Recovery
5. Handling Deadlocks in Reality
Deadlock Detection and Recovery

- Do not constrain $req$ calls or returns
- Instead periodically check for deadlock. If yes, choose a process $i$ and forceably release $alloc_i$

Deadlock detection algorithm for $M$ resource types
// variation of Baker’s algorithm

- Variables
  
  \[
  xavail \leftarrow avail \quad \text{// temporary avail}
  
  done[i] \leftarrow \text{false, for } i = 1, \ldots, N \quad \text{// true iff } i \text{ accounted for}
  
  \]

- While (there is an $i$ s.t.
  
  \[
  done[i] = \text{false and } req_i \leq xavail
  \]
  
  \[
  xavail \leftarrow xavail + alloc_i
  
  done[i] \leftarrow \text{true}
  \]

- If $done[i] = \text{true for every } i$, then no deadlock. Otherwise, processes whose $done$ is false are in a deadlock.
1. Deadlocks Overview
2. Deadlock Prevention
3. Deadlock Avoidance
4. Deadlock Detection & Recovery
5. Handling Deadlocks in Reality
What happens in real-life

- Resources are increasingly shareable
  - disks (vs tapes)
  - demand-paging (vs entire process space in physical memory)
  - virtualization of everything
- Hence livelock (or thrashing) is more common than deadlock
- Hence deadlock prevention/avoidance/detection is rarely used
- Instead, if system “appears” to be in deadlock (or livelock), kill and/or restart processes or entire system