Operating Systems: Deadlocks

Shankar

May 7, 2015
Outline

Deadlocks Overview
Deadlock Prevention
Deadlock Avoidance
Deadlock Detection & Recovery
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 $p$
  - $avail$: resources currently held by manager

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

- Function $rel(p, res)$: release by $p$ of $res$
  - callable iff $res \subseteq alloc(p)$
  - nonblocking

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

Deadlocks Overview
Deadlock Prevention
Deadlock Avoidance
Deadlock Detection & Recovery
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
  - \( req(p, res) \) can be called only when \( alloc(p) \) is empty
  - Impose a total ordering on all resources in \( RES \)
    - \( req(p, res) \) can be called only when \( res > \max(alloc(p)) \)
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`
    - eg, 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”
- Resources: organized into types $1, \cdots, M$
- $\text{Tot} = [\text{Tot}_1, \cdots, \text{Tot}_M]$  // total # of each resource type
- Processes: $1, \cdots, N$
- $\text{Max}_i$: $[\text{Max}_{i,1}, \cdots, \text{Max}_{i,M}]$  // max total need of process $i$

**Variables**
- $\text{alloc}_i$: $[\text{alloc}_{i,1}, \cdots, \text{alloc}_{i,M}]$  // resources held by process $i$
- $\text{avail}$: $[\text{avail}_1, \cdots, \text{avail}_M]$  // resources held by manager
- $\text{req}_i$: $[\text{req}_{i,1}, \cdots, \text{req}_{i,M}]$  // process $i$’s ongoing request
- $\text{need}_i$: $\text{Max}_i - \text{alloc}_i$  // 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 $need_i \leq avail$ and the manager grants only requests of $i$, then it eventually gets $alloc_i$ back

■ A state is **safe** iff it has a safe sequence

■ A **safe sequence** is a permutation $i_1, \cdots, i_N$ of process ids s.t.
  - $need_{i_1} \leq avail$
  - $need_{i_2} \leq avail + alloc_{i_1}$
  - ...
  - $need_{i_N} \leq avail + alloc_{i_1} + \cdots + alloc_{i_{N-1}}$

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

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

While (there is an \( i \) s.t.
\( \text{done}[i] = \text{false and } \text{need}_i \leq x_{avail} \))
- \( x_{avail} \leftarrow x_{avail} + alloc_i \)
- \( \text{done}[i] \leftarrow \text{true} \)

Safe iff \( \text{done}[i] = \text{true} \) for every \( i \)

Return \( \text{req}(p, res) \) only if the resulting state would be safe,
ie, apply Banker’s algorithm to the current state with
- \( \text{avail} \) 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
- **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?**
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>3 3 2</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>3 3 2</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
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.

\[ done[i] = \text{false and } req_i \leq xavail \])

\[ xavail \leftarrow xavail + alloc_i \]
\[ done[i] \leftarrow \text{true} \]

Deadlock iff \( done[i] = \text{false} \) for every \( i \)
Outline

Realities

Deadlocks Overview
Deadlock Prevention
Deadlock Avoidance
Deadlock Detection & Recovery
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