Deadlocks

Gabriel Parmer

Slides evolved from Silberschatz and West
Deadlocks: Synchronization Gone Wild

- A set of blocked processes each
  - Hold a resource (critical section, using device, mem)
  - Wait to acquire a resource held by another of the processes in the set
  - Can cause starvation

- An example:

<table>
<thead>
<tr>
<th>thread 1</th>
<th>thread 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>wait(s1)</td>
<td>wait(s2)</td>
</tr>
<tr>
<td>wait(s2)</td>
<td>wait(s1)</td>
</tr>
<tr>
<td>process()</td>
<td>process()</td>
</tr>
<tr>
<td>signal(s2)</td>
<td>signal(s1)</td>
</tr>
<tr>
<td>signal(s1)</td>
<td>signal(s2)</td>
</tr>
</tbody>
</table>
Traffic and Resource Contention

one-way roads

traffic
Traffic and Resource Contention

Contended Resources

one-way roads

traffic
Traffic and Resource Contention

Contended Resources

one-way roads

traffic
System Model

- Different resource types $R_1, R_2, R_3, \ldots$
  - CPU, Devices, Memory, Data-structures

- Each resource type $R_i$ has $W_i$ instances
  - Amount of memory, multiple CPUs, counting semaphore

- Each process uses a resource as follows:
  - request()
  - use()
  - release()
Deadlock can arise if 4 conditions hold simultaneously

1) **Mutual Exclusion**: single processes uses resource

2) **Hold and Wait**: process holding at least one resource is waiting to acquire additional resources held by other processes

3) **No Preemption**: a resource can be released only voluntarily by the process holding it after use

4) **Circular wait**: there exists a set \( \{P_0, P_1, \ldots, P_n\} \) of waiting processes such that \( P_0 \) is waiting for a resource held by \( P_1 \), \( P_1 \) is waiting for a resource that is held by \( P_2 \), \ldots, and \( P_n \) is waiting for a resource that is held by \( P_0 \).
Resource Allocation Graph

- $G = (V, E)$

- Two types of $V$:
  - $P = \{P_0, P_1, \ldots, P_n\}$, processes in the system
  - $R = \{R_1, R_2, \ldots, R_n\}$, resource types in the system

- Each edge in set $E$ is either:
  - A directed request edge: $P_i \rightarrow R_j$
  - A directed assignment edge: $R_j \rightarrow P_i$
Resource Allocation Graph II

- **Process:**

- **Resource Type with 4 instances:**

  - $P_i$ requests instance of $R_j$:
    - Call `wait(semaphore)`
    - Call `malloc(10)`
  
  - $P_i$ is assigned an instance of $R_j$:
    - `wait(semaphore)` returns
    - `malloc(10)` returns a pointer
Example Resource Allocation Graph

Is there a deadlock?
Example Resource Alloc. Graph II

Is there a deadlock?
Example Resource Alloc. Graph III

Is there a deadlock?
Conditions for Deadlock

• If a graph contains no cycles, no deadlock!

• If graph contains cycle:
  • If one instance per resource type → deadlock
  • If several instances per resource type, deadlock is possible but not certain
Traffic Resource Allocation Graph
Methods for Handling Deadlocks

- Ensure system will *never* enter a deadlock state
  - *Prevention* versus *Avoidance*
- Allow deadlocks to happen, then recover
  - *Detection* and *Recovery*
- Ignorance, luck, and crossed fingers
  - Most systems take this approach
Deadlock Prevention

Prevent any of the 4 conditions for deadlock

• *Mutual Exclusion*: can't compromise here

• *Hold and Wait*: guarantee that when process requests a resource, it holds no others
  • Processes allocated all its resources before it begins execution and requests resources only when it has none
    → low resource utilization and starvation possible
Deadlock Prevention II

- **No Preemption**: If a process holds a resource, and makes a request for another that cannot be satisfied, release all currently held resources
  - Resources added list of resources process is waiting for
  - Process restarted when it can acquire *all* these resources
- **Circular Wait**: Impose a total ordering on resources
  - Ensure that processes request resources in increasing order
  - *Informally, this is a pervasively used technique*
Deadlock Avoidance

- Dynamically observe pattern of resource allocation given system state and decide if its safe to allocate resources
  - Each process declares *maximum number* of resources of each type it will need (a-priori)
  - Deadlock avoidance algorithm dynamically examines resource allocation state; ensure no circular wait condition
  - Resource allocation *state* defined by the number of available and allocated resources *and* the maximum demands of processes
Deadlock Avoidance Algorithms

- Single instance of all system resource types
  - Avoid cycles in resource-allocation graph

- Multiple instances of resource types
  - Dijkstra's Banker's Algorithm
Required Notion: Safe State

- System in *Safe State* if there exists a sequence \(<P_1, P_2, ..., P_n>\) of all processes such that for each \(P_i\), the resources that \(P_i\) can still request can be satisfied by currently available resources and resources held by all \(P_k\), \(k < i\).

- Thus:
  - If \(P_i\) can't currently access all its resources, it can wait for all \(P_k\) to complete.
  - When \(P_i\) terminates, we know that \(P_{i+1}\) can run.

- Safe state sufficient condition to avoid deadlock!
Safe State?

- Assume in *this* case that
  - Maximum resources required = all held and requested resources
Banker's Algorithm

• High level:
  • When a resource request is made, ensure that the allocation will result in a safe state, or
  • Wait for resources until a safe state is possible
  • While other processes compute and eventually release their resources

• Good resource utilization
  • Processes concurrently execute that use “complementary” resources
  • Considers both worst case, and actual allocations
Banker's Algorithm II

- System has $n$ processes, $m$ resource types
- $\text{available}[j] = k$, there are $k$ instances of resource $j$ available
  - vector of length $m$
- $\text{max}[i, j] = k$, $P_i$ will request at most $k$ instances of $R_j$
  - $n \times m$ matrix
- $\text{allocation}[i, j] = k$, $P_i$ is currently allocated $k$ instances of $R_j$
  - $n \times m$ matrix
- $\text{need}[i, j] = k$, $P_i$ may require $k$ more instances of $R_j$ to complete its task
  - $n \times m$ matrix
  - $\text{need}[i, j] = \text{max}[i, j] - \text{allocation}[i, j]$
Safety Algorithm

\[
\text{finished}[n] = \{\text{false, ...}\} \quad \text{// is a process finished executing?}
\text{track\_avail}[m] = \text{available} \quad \text{// copy allocation vector}
\]

\[
\text{while (1) { }
\text{next} = i \text{ where}
\quad \text{finished}[i] = \text{false} \&\& \text{(need}[i, j] \leq \text{track\_avail}[j] \quad \text{forall} \quad j)
\quad \text{if (next doesn't exist) { }
\quad \text{if (finished}[i] \text{ == true forall i) { }
\quad \quad \text{return system is in safe state}
\quad \quad \text{else { }
\quad \quad \quad \text{return system is NOT in a safe state}
\quad \quad \text{}}
\quad \quad \text{}}
\quad \text{}}
\quad \text{/* Process “next” ran successfully.}
\quad \text{* Return its resources to the system */}
\quad \text{finished}[\text{next}] = \text{true}
\quad \text{track\_avail}[j] += \text{allocation}[\text{next, j} \quad \text{forall} \quad j}
\text{}}\]
Resource Request Algorithm

request[i,j] = k // \( P_i \) is requesting \( k \) instances of \( R_j \)

if (request[i,j] > need[i, j] for all j) {
    Error! \( P_i \) requested more than it said it would!
}

while (request[i,j] < available[j] for all j) {
    \( P_i \) must block and wait until more resources become available
}

available[j] -= request[i,j] for all j
allocation[i,j] += request[i,j] for all j
need[i,j] -= request[i,j] for all j

if (system is safe) {
    Resources allocated to \( P_i \)
} else {
    Undo changes to available, allocation, and need, and \( P_i \) waits
}
Banker's Algorithm Example

- Processes $P_0$ through $P_4$ and 3 resource types: $A(10)$, $B(5)$, $C(7)$
- System state at time $t_0$

<table>
<thead>
<tr>
<th></th>
<th>Allocation</th>
<th>Max</th>
<th>Need</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>
<td>$A$ $B$ $C$</td>
</tr>
<tr>
<td>$P0$</td>
<td>0 1 0</td>
<td>7 5 3</td>
<td>7 4 3</td>
<td>3 3 2</td>
</tr>
<tr>
<td>$P1$</td>
<td>2 0 0</td>
<td>3 2 2</td>
<td>1 2 2</td>
<td></td>
</tr>
<tr>
<td>$P2$</td>
<td>3 0 2</td>
<td>9 0 2</td>
<td>6 0 0</td>
<td></td>
</tr>
<tr>
<td>$P3$</td>
<td>2 1 1</td>
<td>2 2 2</td>
<td>0 1 1</td>
<td></td>
</tr>
<tr>
<td>$P4$</td>
<td>0 0 2</td>
<td>4 3 3</td>
<td>4 3 1</td>
<td></td>
</tr>
</tbody>
</table>

Safe State: $<P_1', P_3', P_4', P_2', P_0'>$
Other Safe States???
Example: $P_1$ Requests (1, 0, 2)

- Check that Request $\leq$ Available (i.e. $(1,0,2) \leq (3,3,2) \Rightarrow \text{true}$)

<table>
<thead>
<tr>
<th></th>
<th>Allocation</th>
<th>Need</th>
<th>Available</th>
</tr>
</thead>
<tbody>
<tr>
<td>$P_0$</td>
<td>0 1 0</td>
<td>7 4 3</td>
<td>2 3 0</td>
</tr>
<tr>
<td>$P_1$</td>
<td>3 0 2</td>
<td>0 2 0</td>
<td></td>
</tr>
<tr>
<td>$P_2$</td>
<td>3 0 2</td>
<td>6 0 0</td>
<td></td>
</tr>
<tr>
<td>$P_3$</td>
<td>2 1 1</td>
<td>0 1 1</td>
<td></td>
</tr>
<tr>
<td>$P_4$</td>
<td>0 0 2</td>
<td>4 3 1</td>
<td></td>
</tr>
</tbody>
</table>

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

- Periodically check to see if system is deadlocked
- Doesn't consider *maximum* resources required: practical
- Single instance resources:

![Resource-Allocation Graph](a)

![Corresponding wait-for graph](b)

Resource-Allocation Graph  Corresponding wait-for graph
Deadlock Recovery

• Process Termination
  • Abort all deadlocked processes
  • Abort deadlocked processes one at time, till resolved
  • In which order???
  • OOM killer in Linux