Deadlocks: Prevention, Avoidance, Detection, Recovery

Problematic Emergent Properties

Starvation: Process waits forever
Deadlock: A set of processes exists, where each is blocked and can become unblocked only by actions of another process in the set.

Musings on Deadlock & Starvation

Deadlock vs Starvation
- Starvation: some thread's access to a resource is indefinitely postponed
- Deadlock: circular waiting for resources
- Deadlock implies Starvation, but not vice versa

"Subject to deadlock" does not imply "Will deadlock"
- Testing is not the solution
- System must be deadlock-free by design
System Model

- Set of resources requiring “exclusive” access
  - might be “k-exclusive access” if resource has capacity for k
  - Examples: CPU, printers, memory, locks, etc.
- Acquiring a resource can cause blocking:
  - if resource is free, then access is granted; process proceeds
  - if resource is in use, then process blocks
  - process uses resource
  - process releases resource

Necessary Conditions for Deadlock

- Deadlock possible only if all four hold
  - Bounded resources (Acquire can block invoker)
    - A finite number of threads can use a resource; resources are finite
  - No preemption
    - the resource is mine, MINE! (until I release it)
  - Hold & Wait
    - holds one resource while waiting for another
  - Circular waiting
    - T, waits for T, and holds a resource requested by T,
      - sufficient only if one instance of each resource

A Graph Theoretic Model of Deadlock

- Computer system modeled as a RAG, a directed graph G(V, E)
- V = \{P_1, ..., P_n\} \cup \{R_1, ..., R_n\}
- E = \{edges from a resource to a process\} \cup \{edges from a process to a resource\}

Resource Allocation Graph

- Deadlock possible only if all four hold
- Bounded resources (Acquire can block invoker)
- A finite number of threads can use a resource; resources are finite
- No preemption
- Hold & Wait
- Circular waiting

Not sufficient in general

RAG Reduction

- Deadlock? NO! (no cycles)
  - Step 1: Satisfy P_3 requests
  - Step 2: Satisfy P_2 requests
  - Step 3: Satisfy P_1 requests
  - Schedule \([P_1, P_3, P_2]\) completely eliminates edges
More Musings on Deadlock

- Does the order of RAG reduction matter?
  - No. If $P_i$ and $P_j$ can both be reduced, reducing $P_i$ does not affect the reducibility of $P_j$.

- Does a deadlock disappear on its own?
  - No. Unless a process is killed or forced to release a resource, we are stuck!

- If a system is not deadlock at time $T$, is it guaranteed to be deadlock-free at $T+1$?
  - No. Just by requesting a resource (never mind being granted one) a process can create a circular wait!

Proactive Responses to Deadlock: Prevention

- Negate one of deadlock’s four necessary conditions
  - Remove “Acquire can block invoker”
    - Make resources sharable without locks
    - Wait-free synchronization
  - Make more resources available (duh!)

- Remove “No preemption”
  - Allow OS to preempt resources of waiting processes
  - Allow OS to preempt resources of requesting process if not all available
Proactive Responses to Deadlock: Prevention

Negate one of deadlock’s four necessary conditions

- Remove “Hold & Wait”
  - Request all resources before execution begins
    - Processes may not know what they will need
    - Starvation (if waiting for many popular resources)
    - Low utilization (if resource needed only for a bit)
  - Release all resources before asking anything new
    - Still has the last two problems...

Negate one of deadlock’s four necessary conditions

- Remove “Circular waiting”
  - Single lock for entire system?
  - Impose total/partial order on resources
    - Makes cycles impossible, since a cycle needs edges to go from low to high, and then back to low

Havender’s Scheme (OS/360)

Hierarchical Resource Allocation
Every resource is associated with a level.

Rule H1: All resources from a given level must be acquired using a single request.

Rule H2: After acquiring from level L_j must not acquire from L_i where i<j.

Rule H3: May not release from L_i unless already released from L_j where j>i.

Dining Philosophers (Again)

N philosophers; N plates; N chopsticks

P: do forever
    acquire(min(i, i+1 mod 7)
    acquire(max(i, i+1 mod 7)
    eat
    release(min(i, i+1 mod 7)
    release(max(i, i+1 mod 7)
end
Living dangerously: Safe, Unsafe, Deadlocked States

Why is George Bailey in trouble?

If all his customers ask at the same time to have back all the money they have lent, he is going bankrupt.

But his bank is actually in a safe state!
- If only lenders delayed their requests, all would be well.
- Spoiler alert: this is exactly what happens...

It still begs the question:
- How can the OS allocate resources so that the system always transitions among safe states?

Proactive Responses to Deadlock: Avoidance

The Banker’s Algorithm

Processes declare worst-case needs (big assumption!), but then ask for what they “really” need, a little at a time.
- Sum of maximum resource needs can exceed total available resources

Algorithm decides whether to grant a request
- Build a graph assuming request granted
- Check whether state is safe (i.e., whether RAG is reducible)
- A state is safe if there exists some permutation of \([P_1, P_2, \ldots, P_n]\) such that, for each \(P_i\), the resources that \(P_i\) can still request can be satisfied by the currently available resources plus the resources currently held by all \(P_j\), for \(P_j\) preceding \(P_i\) in the permutation.

Available = 3

<table>
<thead>
<tr>
<th>Process</th>
<th>Max</th>
<th>Holds</th>
<th>Needs</th>
</tr>
</thead>
<tbody>
<tr>
<td>(P_0)</td>
<td>10</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>(P_1)</td>
<td>4</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>(P_2)</td>
<td>9</td>
<td>2</td>
<td>7</td>
</tr>
</tbody>
</table>

Yes! Schedule: \([P_1, P_0, P_2]\)
Proactive Responses to Deadlock: Avoidance

The Banker’s Algorithm

E.W. Dijkstra & N. Habermann

Processes declare worst-case needs (big assumption!), but then ask for what they “really” need, a little at a time

- Sum of maximum resource needs can exceed total available resources

Algorithm decides whether to grant a request

- Build a graph assuming request granted
- Check whether state is safe (i.e., whether RAG is reducible)

A state is safe if there exists a permutation of \(P_1, P_2, \ldots, P_n\) such that, for each \(P_i\), the resources that \(P_i\) can still request can be satisfied by the currently available resources plus the resources currently held by all \(P_j\), for \(P_j\) preceding \(P_i\) in the permutation

Due to space constraints, the table is not included in the natural text. However, it is implied that the table contains information about processes, their maximum needs, the resources they hold, and the resources they need to complete their tasks.

Suppose \(P_2\) asks for 2 resources

Safe?

The Banker’s books

Assume \(n\) processes, \(m\) resources

- \(\text{Max}_{ij} = \text{max amount of units of resource } R_j \text{ needed by } P_i\)
- \(\text{MaxClaim}_{ij} \text{ is a vector of size } m \text{ such that } \text{MaxClaim}_{ij} = \text{Max}_{ij}\)
- \(\text{Hold}_{ij} = \text{current allocation of } R_j \text{ held by } P_i\)
- \(\text{HasNow}_{ij} \text{ is a vector of size } m \text{ such that } \text{HasNow}_{ij} = \text{Hold}_{ij}\)
- \(\text{Available} \text{ is a vector of size } m \text{ such that } \text{Available}_{ij} = \text{units of } R_j \text{ available}\)

A request by \(P_k\) is safe if, assuming the request is granted, there is a permutation of \(P_1, P_2, \ldots, P_n\) such that, for all \(P_i\) in the permutation

\[
\text{Needs}_{ij} = \text{MaxClaim}_{ij} - \text{HasNow}_{ij} + \sum_{j=1}^{i-1} \text{HasNow}_{ij}
\]

An Example

5 processes, 4 resources

<table>
<thead>
<tr>
<th>Max</th>
<th>Hold</th>
<th>Available</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0 1 2</td>
<td>0 0 1 2</td>
</tr>
<tr>
<td>1</td>
<td>7 5 0</td>
<td>1 0 0 0</td>
</tr>
<tr>
<td>2</td>
<td>3 5 6</td>
<td>1 3 5 3</td>
</tr>
<tr>
<td>0</td>
<td>6 5 2</td>
<td>0 6 3 2</td>
</tr>
<tr>
<td>0</td>
<td>6 5 6</td>
<td>0 0 1 4</td>
</tr>
</tbody>
</table>

Is this a safe state?
An Example

5 processes, 4 resources

<table>
<thead>
<tr>
<th>Max</th>
<th>Holds</th>
<th>Available</th>
<th>Needs</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 2</td>
<td>0 0 1 2</td>
<td>2 1 0 0</td>
<td>0 0 0 0</td>
</tr>
<tr>
<td>1 7 5 0</td>
<td>1 0 0 0</td>
<td>0 7 5 0</td>
<td></td>
</tr>
<tr>
<td>2 3 5 6</td>
<td>1 3 5 3</td>
<td>1 0 0 3</td>
<td></td>
</tr>
<tr>
<td>0 6 5 2</td>
<td>0 6 3 2</td>
<td>0 0 2 0</td>
<td></td>
</tr>
<tr>
<td>0 6 5 6</td>
<td>0 0 1 2</td>
<td>0 6 4 2</td>
<td></td>
</tr>
</tbody>
</table>

Is this a safe state? P₁, P₄, P₂, P₃, P₅

- While safe permutation does not include all processes:
  - Is there a Pᵢ such that Needsᵢ ≤ Availᵢ?
    - If no, exit with unsafe
    - If yes, add Pᵢ to the sequence and set Availᵢ = Availᵢ + HasNowᵢ
- Exit with safe

An Example

5 processes, 4 resources

<table>
<thead>
<tr>
<th>Max</th>
<th>Holds</th>
<th>Available</th>
<th>Needs</th>
</tr>
</thead>
<tbody>
<tr>
<td>P₁ 0 0 1 2</td>
<td>P₄ 0 0 1 2</td>
<td>2 1 0 0</td>
<td>0 0 0 0</td>
</tr>
<tr>
<td>P₂ 1 7 5 0</td>
<td>P₃ 0 4 2 0</td>
<td>1 3 3 0</td>
<td></td>
</tr>
<tr>
<td>P₃ 2 3 5 6</td>
<td>P₃ 1 3 5 3</td>
<td>1 0 0 3</td>
<td></td>
</tr>
<tr>
<td>P₄ 0 6 5 2</td>
<td>P₄ 0 6 3 2</td>
<td>0 0 2 0</td>
<td></td>
</tr>
<tr>
<td>P₅ 0 6 5 6</td>
<td>P₅ 0 0 1 4</td>
<td>0 6 4 2</td>
<td></td>
</tr>
</tbody>
</table>

P₂ want to change its holdings to 0 4 2 0

Safe?

An Example

Max       Holds       Available       Needs
R₁ R₂ R₃ R₄       R₁ R₂ R₃ R₄       R₁ R₂ R₃ R₄       R₁ R₂ R₃ R₄
P₁ 0 0 1 2       P₄ 0 0 1 2       2 1 0 0       0 0 0 0
P₂ 1 7 5 0       P₃ 0 4 2 0       1 3 3 0       |
P₃ 2 3 5 6       P₃ 1 3 5 3       1 0 0 3       |
P₄ 0 6 5 2       P₄ 0 6 3 2       0 0 2 0       |
P₅ 0 6 5 6       P₅ 0 0 1 4       0 6 4 2       |

P₂ want to change its holdings to 0 4 2 0

Safe?

Reactive Responses to Deadlock

Deadlock Detection
- Track resource allocation (who has what)
- Track pending requests (who’s waiting for what)

When should it run?
- For each request?
- After each unsatisfiable request?
- Every hour?
- Once CPU utilization drops below a threshold?
5 processes, 3 resources. We no longer (need to) know Max.

<table>
<thead>
<tr>
<th>Holds</th>
<th>Available</th>
<th>Pending</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>R₁  R₂  R₃</td>
<td>R₁  R₂  R₃</td>
</tr>
<tr>
<td>P₁</td>
<td>0  1  0</td>
<td>0  0  0</td>
</tr>
<tr>
<td>P₂</td>
<td>2  0  0</td>
<td>2  2  0</td>
</tr>
<tr>
<td>P₃</td>
<td>3  0  3</td>
<td>0  0  0</td>
</tr>
<tr>
<td>P₄</td>
<td>2  1  1</td>
<td>1  0  2</td>
</tr>
<tr>
<td>P₅</td>
<td>0  0  2</td>
<td>0  0  2</td>
</tr>
</tbody>
</table>

Given the set of pending requests, is there a safe sequence?
- If no, deadlock

---

5 processes, 3 resources. We no longer (need to) know Max.

<table>
<thead>
<tr>
<th>Holds</th>
<th>Available</th>
<th>Pending</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>R₁  R₂  R₃</td>
<td>R₁  R₂  R₃</td>
</tr>
<tr>
<td>P₁</td>
<td>0  1  0</td>
<td>0  0  0</td>
</tr>
<tr>
<td>P₂</td>
<td>2  0  0</td>
<td>2  2  0</td>
</tr>
<tr>
<td>P₃</td>
<td>3  0  3</td>
<td>0  0  0</td>
</tr>
<tr>
<td>P₄</td>
<td>2  1  1</td>
<td>1  0  2</td>
</tr>
<tr>
<td>P₅</td>
<td>0  0  2</td>
<td>0  0  2</td>
</tr>
</tbody>
</table>

Given the set of pending requests, is there a safe sequence?
- If no, deadlock
Detecting Deadlock

5 processes, 3 resources. We no longer (need to) know Max.

Given the set of pending requests, is there a safe sequence?
- If no, deadlock

Detecting Deadlock

5 processes, 3 resources. We no longer (need to) know Max.

Given the set of pending requests, is there a safe sequence?
- If no, deadlock

Detecting Deadlock

5 processes, 3 resources. We no longer (need to) know Max.

Given the set of pending requests, is there a safe sequence?
- If no, deadlock

Detecting Deadlock

5 processes, 3 resources. We no longer (need to) know Max.

Given the set of pending requests, is there a safe sequence?
- If no, deadlock
Detecting Deadlock

5 processes, 3 resources. We no longer (need to) know Max.

<table>
<thead>
<tr>
<th>Holds</th>
<th>Available</th>
<th>Pending</th>
</tr>
</thead>
<tbody>
<tr>
<td>P₁: 0</td>
<td>R₁: 7</td>
<td>0</td>
</tr>
<tr>
<td>P₂: 0</td>
<td>R₂: 2</td>
<td>0</td>
</tr>
<tr>
<td>P₃: 0</td>
<td>R₃: 4</td>
<td>0</td>
</tr>
<tr>
<td>P₄: 0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>P₅: 0</td>
<td>0</td>
<td>2</td>
</tr>
</tbody>
</table>

Given the set of pending requests, is there a safe sequence?

- If no, deadlock

Yes, there is a safe sequence!
Detecting Deadlock

5 processes, 3 resources. We no longer (need to) know Max

<table>
<thead>
<tr>
<th></th>
<th>R1</th>
<th>R2</th>
<th>R3</th>
</tr>
</thead>
<tbody>
<tr>
<td>P0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>P1</td>
<td>2</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>P2</td>
<td>3</td>
<td>0</td>
<td>3</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>

<table>
<thead>
<tr>
<th></th>
<th>R1</th>
<th>R2</th>
<th>R3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Holds</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Avail</td>
<td>2</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>Pending</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>0</td>
<td>2</td>
</tr>
</tbody>
</table>

Given the set of pending requests, is there a safe sequence?
- If no, deadlock
- Can we avoid deadlock by delaying granting requests?
  - Deadlock triggered when request formulated, not granted!

Deadlock Recovery

- Blue screen & reboot
- Kill one/all deadlocked processes
  - Pick a victim (how?); Terminate; Repeat as needed
    - Can leave system in inconsistent state
- Proceed without the resource
  - Example: timeout on inventory check at Amazon
- Use transactions
  - Rollback & Restart
  - Need to pick a victim...

Summary

- Prevent
  - Negate one of the four necessary conditions
- Avoid
  - Schedule processes carefully
- Detect
  - Has a deadlock occurred?
- Recover
  - Kill or Rollback