Chapter 8: Deadlock Questions?

- What is a deadlock?
- What causes a deadlock?
- How do you deal with (potential) deadlocks?

Deadlock: What is a deadlock?

- All are waiting for a resource that is held by another waiting entity. Since all are waiting, none can provide any of the things being waited for.
- Example: narrow bridge (resource) —
  - if a deadlock occurs, resolved if one car back up (preempts resource and rollback).

Example: Two Threads?

- Two threads access two shared variables, A and B
  - Variable A is protected by lock a
  - Variable B by lock b
- How to add lock and unlock statements?

Example: Maria & Tucker

- Maria gets lock a
- Tucker gets lock b
- Maria waits for lock b
- Tucker waits lock b

Representing Deadlock

- Two common ways of representing deadlock:
  - Vertices
    - threads (or processes) in system
  - resources [types] (e.g., locks, semaphores, printers)
  - Edges: indicates ‘waiting for’, ‘wants’, ‘held by’
**Conditions for Deadlock**

- **Mutual exclusion:**
  - Resource cannot be shared
  - Requests are delayed until resource is released
- **Hold and wait:**
  - Thread holds one resource while waits for another
- **No preemption:**
  - Previously granted resources cannot forcibly be taken away
- **Circular wait:**
  - Circular dependencies exist in "wants-for" or "resource-allocation" graphs
  - Each is waiting for a resource held by next member of the chain.

All for conditions must hold simultaneously

**Deadlock prevention:**

**Mutual Exclusion**

- **Approach**
  - Ensure 1 of 4 conditions cannot occur
  - Negate each of the 4 conditions
- **No single approach is appropriate (or possible) for all circumstances**
- **No mutual exclusion --> Make resource sharable examples:**
  - Read-only files
  - Printer daemon needs exclusive access to the printer, there is only one printer daemon -- uses spooling.

**Deadlock Prevention**

**Hold and Wait**

- **Two Approaches:**
  1. Only request resources when it does not hold other resources
  - Only request resources before requesting new ones
- **Deadlock avoidance**
  - Ensure deadlock does not happen
  - Use information about resource requests to dynamically avoid unsafe situations

**Deadlock Prevention**

**Hold and Wait**

- **Two Approaches:**
  1. Atomically acquire all resources at once
  - Example: Single lock to protect all (other variations - e.g., release access to one variable earlier)
- **Problems:**
  - Low resource utilization: ties up resources other processes could be using
  - May not know required resources before run
  - Starvation: A thread that need popular resources may wait forever
## Deadlock Prevention

### No Preemption

- **Two Approaches:**
  1. Preempt requestors resource
     - Example: B is holding some resources and then requests additional resources that are held by other threads, then B releases all its resources
  2. Preempt holders resource
     - Example: A waiting for something held by B (and B is waiting for something else), then take resource away from B and give to A

- **Not possible if resource cannot be saved and restored**
  - Can’t take away a lock without causing problems
- **Only works for some resources (e.g., CPU and memory)**

---

## Deadlock Prevention

### Circular Wait Condition

- **Impose ordering on resources**
  - Give all resources a ranking; must acquire highest ranked first

## Deadlock Detection

1. **Allow system to enter deadlock state**
2. **Detection algorithm**
3. **Recovery scheme**

---

## Deadlock Detection

### Single Instance of Each Resource Type

- Maintain `wait-for` graph
  - Nodes are processes.
  - Removes resource nodes and collapsing edges
  - Periodically invoke an algorithm that searches for a cycle in the graph.
  - An algorithm to detect a cycle in a graph requires an order of $n^2$ operations, where $n$ is the number of vertices in the graph.

---

## Deadlock Detection

### Depth first search

For each node in the graph:

```plaintext
L = {empty list} and Nodes = {unvisited};
current node = initial node;
while( current node is not the initial node twice )
    L.enqueue(current node); // add to node to end of L
    if( current node is in L twice )
        there is a cycle = cycle and return
    if( there is an unmarked arc )
        mark the arc as visited and use destination as new current node
    else
        go back to previous node
back to initial node there is no cycle
```

---

## Deadlock Detection

### Deadlock detection (1 resource of each)

- **Do a depth-first-search on the resource allocation graph**
  - $D, E, G$ ? are deadlocked
  - $A, C, F$ ? are not deadlocked because $S$ can be allocated to either and then the other two can take turn to complete
Example: Deadlock Detection

- Do a depth-first-search on the resource allocation graph

  Initialize a list to the empty list, designate arcs as 'unvisited'

Deadlock Detection with Multiple Resources

- **Theorem:** If a graph does not contain a cycle then no processes are deadlocked
  - A cycle in a RAG is a necessary condition for deadlock
  - Is it a sufficient condition?

Deadlock Detection: Multiple Resource Instances

- Resources in existence
  
  \[ (E_1, E_2, \ldots, E_m) \]

- Resources available
  
  \[ (A_1, A_2, \ldots, A_m) \]

- **Current allocation matrix:**
  
  \[
  \begin{bmatrix}
  C_{11} & C_{12} & \cdots & C_{1m} \\
  C_{21} & C_{22} & \cdots & C_{2m} \\
  \vdots & \vdots & \ddots & \vdots \\
  C_{n1} & C_{n2} & \cdots & C_{nm}
  \end{bmatrix}
  \]

- **Request matrix:**
  
  \[
  \begin{bmatrix}
  R_{11} & R_{12} & \cdots & R_{1m} \\
  R_{21} & R_{22} & \cdots & R_{2m} \\
  \vdots & \vdots & \ddots & \vdots \\
  R_{n1} & R_{n2} & \cdots & R_{nm}
  \end{bmatrix}
  \]

- **Row n** is current allocation to process n

- **What I have (now):**

- **What I am requesting now:**

  If Request \([i] = k\), then process \(P_i\) is requesting \(k\) more instances of type \(R_j\).
Example

Is there a sequence of running process such that all the resources will be returned?

<table>
<thead>
<tr>
<th>Task class</th>
<th>Process 1</th>
<th>Process 2</th>
<th>Process 3</th>
<th>Process 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU</td>
<td>2</td>
<td>3</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>Memory</td>
<td>4</td>
<td>2</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>Disk Space</td>
<td>2</td>
<td>3</td>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td>Output</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
</tr>
</tbody>
</table>

E = \begin{pmatrix} 4 & 2 & 3 & 1 \end{pmatrix} \quad A = \begin{pmatrix} 2 & 1 & 0 & 0 \end{pmatrix}

Current allocation matrix

<table>
<thead>
<tr>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 0</td>
</tr>
<tr>
<td>2 0 1</td>
</tr>
<tr>
<td>0 1 2</td>
</tr>
</tbody>
</table>

Request matrix

<table>
<thead>
<tr>
<th>R</th>
</tr>
</thead>
<tbody>
<tr>
<td>2 0 1</td>
</tr>
<tr>
<td>1 0 1</td>
</tr>
<tr>
<td>2 1 0</td>
</tr>
</tbody>
</table>

Detection algorithm

1. Look for an unmarked process \( P_i \), for which the \( i \)-th row of \( R \) (need) is less than or equal to \( A \).
2. If such a process is found, add the \( i \)-th row of \( C \) to \( A \), mark the process and go back to step 1.
3. If no such process exists the algorithm terminates.

If all marked, no deadlock.
Detection algorithm

<table>
<thead>
<tr>
<th>E</th>
<th>A</th>
</tr>
</thead>
<tbody>
<tr>
<td>(4 2 3 1)</td>
<td>(2 1 0 0)</td>
</tr>
<tr>
<td>2 2 2 0</td>
<td></td>
</tr>
</tbody>
</table>

No deadlock!

Detection algorithm

<table>
<thead>
<tr>
<th>E</th>
<th>A</th>
</tr>
</thead>
<tbody>
<tr>
<td>(4 2 3 1)</td>
<td>(2 1 0 0)</td>
</tr>
<tr>
<td>2 2 2 0</td>
<td></td>
</tr>
</tbody>
</table>

Current allocation matrix

C = [0 0 1 0]
2 0 1
0 0 0

Request matrix

R = [2 0 0 1]
1 0 1 0

Deadlock detection issues

- How often should the algorithm run?
  - After every resource request?
  - Periodically?
  - When CPU utilization is low?
  - When we suspect deadlock because some thread has been asleep for a long period of time?

Recovery from deadlock

- What should be done to recover?
  - Abort deadlocked processes and reclaim resources
  - Temporarily reclaim resource, if possible
  - Abort one process at a time until deadlock cycle is eliminated
- Where to start?
  - Low priority process
  - How long process has been executing
  - How many resources a process holds
  - Batch or interactive
  - Number of processes that must be terminated

Other deadlock recovery techniques

- Recovery through rollback
  - Save state periodically
    - take a checkpoint
    - start computation again from checkpoint
  - Done for large computation systems

Recovery through rollback
Review: Handling Deadlock

- Ignore
  - Easiest and most common approach (e.g., UNIX).
- Deadlock prevention
  - Ensure deadlock does not happen
  - Ensure at least one of 4 conditions does not occur
- Deadlock detection and recovery
  - Allow deadlocks, but detect when occur
  - Recover and continue
- Deadlock avoidance
  - Ensure deadlock does not happen
  - Use information about resource requests to dynamically avoid unsafe situations

Deadlock avoidance

- Detection vs. avoidance...
  - Detection – “optimistic” approach
    - Allocate resources
    - “Break” system to fix it
  - Avoidance – “pessimistic” approach
    - Don’t allocate resource if it may lead to deadlock
    - If a process requests a resource...
      ... make it wait until you are sure it’s OK
  - Which one to use depends upon the application

Process-resource trajectories

- Process A
  - Time t1, t2, t3, t4
  - Requests Printer
  - Releases Printer

- Process B
  - Time t1, t2, t3, t4
  - Requests CD-RW
  - Releases CD-RW
  - Request CD-RW

Ostrich algorithm
Both processes hold CD-RW

Forbidden Zone

B makes progress, A is not running

Trajectory showing system progress
Process-resource trajectories

Process A

Process B

B requests the CD-RW

A runs & makes a request for printer

Request is granted; A proceeds

B runs & requests the printer... MUST WAIT!

A runs & requests the CD-RW
A... holds printer requests CD-RW
B... holds CD-RW requests printer

Process A

time

Process B

RC
RP
RLC
RLP

Process B

RC
RP
RLC
RLP

Process A

time

DEADLOCK!

Process A

RC
RP
RLC
RLP

Process A

RC
RP
RLC
RLP

This area is “unsafe”

Within the “unsafe” area, deadlock is inevitable. We don’t want to enter this area. The OS should make A wait at this point!

B requests the printer, B releases CD-RW, B releases printer, then A runs to completion!
Safe states

- The current state:
  "which processes hold which resources"

- A "safe" state:
  - No deadlock, and
  - There is some scheduling order in which every process can run to completion even if all of them request their maximum number of units immediately

- The Banker’s Algorithm:
  - Goal: Avoid unsafe states!!!
  - When a process requests more units, should the system grant the request or make it wait?

Avoidance with multiple resource types

- Resources in existence
  \((E_1, E_2, \ldots, E_m)\)

- Resources available
  \((R_1, R_2, \ldots, R_n)\)

Current allocation matrix

<table>
<thead>
<tr>
<th>Process #1</th>
<th>Process #2</th>
<th>Process #3</th>
<th>Process #4</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Request matrix

\[
\begin{bmatrix}
R_{11} & R_{12} & R_{13} & \cdots & R_{1n} \\
R_{21} & R_{22} & R_{23} & \cdots & R_{2n} \\
\vdots & \vdots & \vdots & \ddots & \vdots \\
C_{m1} & C_{m2} & C_{m3} & \cdots & C_{mn}
\end{bmatrix}
\]

Flow \(n\) is current allocation to process \(n\)

Note: These are the max. possible requests, which we assume are known ahead of time
Banker’s algorithm for multiple resources

- Look for a row, R, whose unmet resource needs are all smaller than or equal to A. If no such row exists, the system will eventually deadlock since no process can run to completion.
- Assume the process of the row chosen requests all the resources that it needs (which is guaranteed to be possible) and finishes. Mark that process as terminated and add all its resources to A vector.
- Repeat steps 1 and 2, until either all process are marked terminated, in which case the initial state was safe, or until deadlock occurs, in which case it was not.

Avoidance modeling

**Total resource vector**
- Resources in existence: R_1, R_2, ..., R_n

**Current allocation matrix**
- C: [C_1, C_2, ..., C_n]

**Available resource vector**
- Resources available: A_1, A_2, ..., A_n

**Maximum Request Vector**
- R: [R_1, R_2, ..., R_n]

Row n is current allocation to process n

RUN ALGORITHM ON EVERY RESOURCE REQUEST

Avoidance algorithm

**Max request matrix**

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>2</td>
<td>2</td>
<td>0</td>
</tr>
</tbody>
</table>

**Current allocation matrix**

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>0</td>
</tr>
</tbody>
</table>

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Avoidance algorithm

**Max request matrix**

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>0</td>
</tr>
</tbody>
</table>

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Avoidance algorithm

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>2</td>
<td>0</td>
</tr>
</tbody>
</table>

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>
## Deadlock avoidance

- **Deadlock avoidance is usually impossible**
  - because you don’t know in advance what resources a process will need!