Deadlocks

System Model

- There are non-shared computer resources
  - Maybe more than one instance
  - Printers, Semaphores, Tape drives, CPU
- Processes need access to these resources
  - Acquire resource
    - If resource is available, access is granted
    - If not available, the process is blocked
  - Use resource
  - Release resource
- Undesirable scenario:
  - Process A acquires resource 1, and is waiting for resource 2
  - Process B acquires resource 2, and is waiting for resource 1
⇒ Deadlock!

For example: Semaphores

```c
semaphore: mutex1 = 1    /* protects resource 1 */
mutex2 = 1    /* protects resource 2 */
Process A code:
{ /* initial compute */
P(mutex1)
P(mutex2)
/* use both resources */
V(mutex2)
V(mutex1)
}
Process B code:
{ /* initial compute */
P(mutex2)
P(mutex1)
/* use both resources */
V(mutex2)
V(mutex1)
}
```

Four Conditions for Deadlock

- Coffman et. al. 1971
- Necessary conditions for deadlock to exist:
  - Mutual Exclusion
    - At least one resource must be held in non-sharable mode
  - Hold and wait
    - There exists a process holding a resource, and waiting for another
  - No preemption
    - Resources cannot be preempted
  - Circular wait
    - There exists a set of processes \( P_i, P_j, \ldots, P_n \), such that
      - \( P_i \) is waiting for \( P_j \), \( P_j \) for \( P_k \), \ldots , and \( P_n \) for \( P_i \)
All four conditions must hold for deadlock to occur

Resource Allocation Graph

- Deadlock can be described using a resource allocation graph, RAG
- The RAG consists of:
  - set of vertices \( V = P \cup R \)
  - where \( P=P_1, P_2, \ldots, P_n \) of processes and \( R=R_1, R_2, \ldots, R_m \) of resources.
  - Request edge: directed edge from a process to a resource, \( P_i \rightarrow R_j \) implies that \( P_i \) has requested \( R_j \)
  - Assignment edge: directed edge from a resource to a process, \( R_j \rightarrow P_i \) implies that \( R_j \) has been allocated to \( P_i \)
- If the graph has no cycles, deadlock cannot exist.
- If the graph has a cycle, deadlock may exist.
**Dealing with Deadlocks**

- **Proactive Approaches:**
  - Deadlock Prevention
    - Negate one of 4 necessary conditions
    - Prevent deadlock from occurring
  - Deadlock Avoidance
    - Carefully allocate resources based on future knowledge
    - Deadlocks are prevented
- **Reactive Approach:**
  - Deadlock detection and recovery
  - Let deadlock happen, then detect and recover from it
- **Ignore the problem**
  - Pretend deadlocks will never occur
  - Ostrich approach (real Oss!!!)

**Deadlock Prevention**

- Can the OS prevent deadlocks?
- Prevention: Negate one of necessary conditions
  - Mutual exclusion:
    - Make resources sharable
    - Not always possible (spooling?)
  - Hold and wait
    - Do not hold resources when waiting for another
    - Request all resources before beginning execution
    - Processes do not know what all they will need
    - Starvation (if waiting on many popular resources)
    - Low utilization (Need resource only for a bit)
    - Alternative: Release all resources before requesting anything new
      - Still has the last two problems
- No preemption:
  - Make resources preemptable (2 approaches)
    - Preempt requesting processes’ resources if all not available
    - Preempt resources of waiting processes to satisfy request
    - Good when easy to save and restore state of resource
    - CPU registers, memory virtualization
  - Circular wait: (2 approaches)
    - Single lock for entire system? (Problems)
    - Impose partial ordering on resources, request them in order

**Breaking Circular Wait**

- Order resources (lock1, lock2, …)
- Acquire resources in strictly increasing/decreasing order
- When requests to multiple resources of same order:
  - Make the request a single operation
- Intuition: Cycle requires an edge from low to high, and from high to low numbered node, or to same node
  - Ordering not always possible, low resource utilization

**Two phase locking**

- Acquire all resources, if block on any, release all, and retry
  ```
  print_file:
  lock(file);
  acquire printer
  acquire disk;
  ...do work...
  release all
  ```
- Pro: dynamic, simple, flexible
- Con:
  - Cost with number of resources?
  - Length of critical section?
  - Hard to know what’s needed a priori
Deadlock Avoidance

- If we have future information
  - Max resource requirement of each process before they execute
- Can we guarantee that deadlocks will never occur?
- Avoidance Approach:
  - Before granting resource, check if state is safe
  - If the state is safe $\Rightarrow$ no deadlock!

Safe State

- A state is said to be safe, if it has a process sequence $(P_1, P_2, \ldots, P_n)$, 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 held by all $P_j$, where $j < i$
- State is safe because OS can definitely avoid deadlock
  - by blocking any new requests until safe order is executed
- This avoids circular wait condition
  - Process waits until safe state is guaranteed

Safe State Example

- Suppose there are 12 tape drives
  - max need | current usage | could ask for
  - p0 | 10 | 5 | 5
  - p1 | 4 | 2 | 2
  - p2 | 9 | 2 | 7
- 3 drives remain
- current state is safe because a safe sequence exists: $<p_1,p_0,p_2>$
  - $p_1$ can complete with current resources
  - $p_0$ can complete with current+$p_1$
  - $p_2$ can complete with current +$p_1$+$p_0$
- If $p_2$ requests 1 drive, then it must wait to avoid unsafe state.

Safe State Example

- (One resource class only)

<table>
<thead>
<tr>
<th>process</th>
<th>holding</th>
<th>max claims</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>4</td>
<td>6</td>
</tr>
<tr>
<td>B</td>
<td>4</td>
<td>11</td>
</tr>
<tr>
<td>C</td>
<td>2</td>
<td>7</td>
</tr>
</tbody>
</table>
- unallocated: 2
  - safe sequence: A,C,B
  - If C should have a claim of 9 instead of 7, there is no safe sequence.

RAG Algorithm

- Works if only one instance of each resource type
- Algorithm:
  - Add a claim edge, $P_i \rightarrow R_j$, if $P_i$ can request $R_j$ in the future
    - Represented by a dashed line in graph
  - A request $P_i \rightarrow R_j$ can be granted only if:
    - Adding an assignment edge $R_j \rightarrow P_i$ does not introduce cycles
      - Since cycles imply unsafe state

However, this sequence is not safe:
If C should have 7 instead of 6 requests, deadlock exists.
Banker's Algorithm

- Decides whether to grant a resource request.
- Data structures:

  m: integer  # of processes
  n: integer  # of resources
  available[1..m]: available[i] is # of avail resources of type i
  max[1..n,1..m]: max demand of each Pi for each Ri
  allocation[1..n,1..m]: current allocation of resource Rj to Pi
  need[1..n,1..m]: max # resource Rj that Pi may still request
  request[i]: vector of # of resource Rj Process Pi wants

Let's assume that we satisfy the request. Then we would have:

\[
\begin{align*}
\text{available} &= \text{available} - \text{request}[i] \\
\text{allocation}[i] &= \text{allocation}[i] + \text{request}[i] \\
\text{need}[i] &= \text{need}[i] - \text{request}[i]
\end{align*}
\]

Now, check if this would leave us in a safe state:
- if yes, grant the request,
- if no, then leave the state as is and cause process to wait.

Basic Algorithm

1. If request[i] > need[i] then
   error (asked for too much)
2. If request[i] > available[i] then
   wait (can't supply it now)
3. Resources are available to satisfy the request
   Let's assume that we satisfy the request. Then we would have:
   \[
   \begin{align*}
   \text{available} &= \text{available} - \text{request}[i] \\
   \text{allocation}[i] &= \text{allocation}[i] + \text{request}[i] \\
   \text{need}[i] &= \text{need}[i] - \text{request}[i]
   \end{align*}
   \]
   Now, check if this would leave us in a safe state:
   - if yes, grant the request,
   - if no, then leave the state as is and cause process to wait.

Safety Check

- free[1..m] = available  /* how many resources are available */
- finish[1..n] = false (for all i)  /* none finished yet */

Step 1: Find an i such that finish[i]=false and need[i] <= work
  /* find a proc that can complete its request now */
  if no such i exists, go to step 3  /* we're done */

Step 2: Found an i:
  finish[i] = true  /* done with this process */
  free = free + allocation[i]  /* assume this process were to finish, and its allocation 
                              back to the available list */
  go to step 1

Step 3: If finish[i] = true for all i, the system is safe. Else Not

Banker's Algorithm: Example

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

This is still safe: safe seq <P1, P3, P4, P0, P2>

In this new state,
- P4 requests (3,3,0)
  - not enough available resources
- P0 requests (0,2,0)
  - let's check resulting state

Banker's Algorithm: Example

<table>
<thead>
<tr>
<th>Allocation</th>
<th>Max</th>
<th>Available</th>
</tr>
</thead>
<tbody>
<tr>
<td>A B C</td>
<td>A B C</td>
<td>A B C</td>
</tr>
<tr>
<td>P0 0 3 0</td>
<td>7 5 3</td>
<td>2 1 0</td>
</tr>
<tr>
<td>P1 0 3 0</td>
<td>3 2 2</td>
<td>1 1 1</td>
</tr>
<tr>
<td>P2 0 0 2</td>
<td>4 3 3</td>
<td>0 0 2</td>
</tr>
</tbody>
</table>

This is unsafe state (why?)
- So P0's request will be denied

Problems with Banker's Algorithm?
Deadlock Detection & Recovery

- If none of these approaches are used, deadlock can occur
- This scheme requires:
  - Detection: finding out if deadlock has occurred
    - Keep track of resource allocation (who has what)
    - Keep track of pending requests (who is waiting for what)
  - Ways to recover from it
- Expensive to detect, as well as recover

RAG Algorithm

- Suppose there is only one instance of each resource
- Example 1: Is this a deadlock?
  - P1 has R2 and R3, and is requesting R1
  - P2 has R4 and is requesting R3
  - P3 has R1 and is requesting R4
- Example 2: Is this a deadlock?
  - P1 has R2, and is requesting R1 and R3
  - P2 has R4 and is requesting R3
  - P3 has R1 and is requesting R4
- Use a wait-for graph:
  - Collapse resources
  - An edge $P_i \rightarrow P_k$ exists only if RAG has $P_i \rightarrow R_j$ & $R_j \rightarrow P_k$
  - Cycle in wait-for graph $\Rightarrow$ deadlock!

2nd Detection Algorithm

- What if there are multiple resource instances?
- Data structures:
  - $n$: integer # of processes
  - $m$: integer # of resources
  - available[1..m] available[i] is # of avail resources of type i
  - request[1..n,1..m] max demand of each $P_i$ for each $R_i$
  - allocation[1..n,1..m] current allocation of resource $R_j$ to $P_i$
  - finish[1..n] true if $P_i$’s request can be satisfied

Example

Finished = {F, F, F, F};
Work = Available = (0, 0, 1);

<table>
<thead>
<tr>
<th>Allocation</th>
<th>Request</th>
</tr>
</thead>
<tbody>
<tr>
<td>$R_1$</td>
<td>$R_2$</td>
</tr>
<tr>
<td>$P_1$</td>
<td>1</td>
</tr>
<tr>
<td>$P_2$</td>
<td>2</td>
</tr>
<tr>
<td>$P_3$</td>
<td>1</td>
</tr>
<tr>
<td>$P_4$</td>
<td>1</td>
</tr>
</tbody>
</table>

2nd Detection Algorithm

1. work[]:=available[]
   for all $i < n$, if allocation[] $\neq 0$
     then finish[]:=false else finish[]:=true
2. find an index $i$ such that:
   - finish[]:=false;
   - request[]:=work
   - if no such $i$ exists, go to 4.
3. work:=work+allocation[]
   finish[]:=true, go to 2
4. if finish[] = false for some $i$,
   then system is deadlocked with $P_i$ in deadlock

Example

Finished = {F, F, T, F};
Work = (1, 1, 1);

<table>
<thead>
<tr>
<th>Allocation</th>
<th>Request</th>
</tr>
</thead>
<tbody>
<tr>
<td>$R_1$</td>
<td>$R_2$</td>
</tr>
<tr>
<td>$P_1$</td>
<td>3</td>
</tr>
<tr>
<td>$P_2$</td>
<td>2</td>
</tr>
<tr>
<td>$P_3$</td>
<td>1</td>
</tr>
<tr>
<td>$P_4$</td>
<td>1</td>
</tr>
</tbody>
</table>

RAG Algorithm

- Suppose there is only one instance of each resource
- Example 1: Is this a deadlock?
  - P1 has R2 and R3, and is requesting R1
  - P2 has R4 and is requesting R3
  - P3 has R1 and is requesting R4
- Example 2: Is this a deadlock?
  - P1 has R2, and is requesting R1 and R3
  - P2 has R4 and is requesting R3
  - P3 has R1 and is requesting R4
- Use a wait-for graph:
  - Collapse resources
  - An edge $P_i \rightarrow P_k$ exists only if RAG has $P_i \rightarrow R_j$ & $R_j \rightarrow P_k$
  - Cycle in wait-for graph $\Rightarrow$ deadlock!
Example

\[
\begin{array}{ccc}
R_1 & R_2 & R_3 \\
P_1 & 1 & 1 & 1 \\
P_2 & 2 & 1 & 2 \\
P_3 & 1 & 1 & 0 \\
P_4 & 1 & 1 & 1 \\
\end{array}
\]

Allocation

\[
\begin{array}{ccc}
R_1 & R_2 & R_3 \\
P_1 & 3 & 2 & 1 \\
P_2 & 2 & 2 & 1 \\
P_3 & 1 & 1 & 0 \\
P_4 & 1 & 1 & 1 \\
\end{array}
\]

Request

Example

\[
\begin{array}{ccc}
R_1 & R_2 & R_3 \\
P_1 & 3 & 2 & 1 \\
P_2 & 2 & 2 & 1 \\
P_3 & 1 & 1 & 0 \\
P_4 & 1 & 1 & 1 \\
\end{array}
\]

Allocation

\[
\begin{array}{ccc}
R_1 & R_2 & R_3 \\
P_1 & 3 & 2 & 1 \\
P_2 & 2 & 2 & 1 \\
P_3 & 1 & 1 & 0 \\
P_4 & 1 & 1 & 1 \\
\end{array}
\]

Request

When to run Detection Algorithm?

- For every resource request?
- For every request that cannot be immediately satisfied?
- Once every hour?
- When CPU utilization drops below 40%?

Deadlock Recovery

- Killing one/all deadlocked processes
  - Crude, but effective
  - Keep killing processes, until deadlock broken
  - Repeat the entire computation
- Preempt resource/processes until deadlock broken
  - Selecting a victim (# resources held, how long executed)
  - Rollback (partial or total)
  - Starvation (prevent a process from being executed)

What happens today?

- Ostrich Approach
- Deadlock avoidance and prevention is often impossible
- Thorough detection of all scenarios too expensive
- All operating systems have potential deadlocks
- Engineering philosophy:
  The price of infrequent crashes in exchange for performance and user convenience is worth it

SQL Server

- Runs detection algorithm:
  - Periodically, or
  - On demand
- Recovers by terminating:
  - Least expensive process, or
  - User specified priority

Transaction (Process ID xxx) was deadlocked on (xxx) resources with another process and has been chosen as the deadlock victim. Rerun the transaction.
Windows DDK Driver Verifier

- DDK = Device Driver Kit
- Added in XP and later
- Uses Deadlock Prevention, by breaking circular-wait
  - Checks for a hierarchy in your locking mechanism
- Will bugcheck even if your system has not deadlocked!
  - (0xc4), fatal error

- You would not use it in a production system
  - Useful in development