Dining Philosophers

- Philosophers think
- Thinking makes them hungry
- When they want to eat, they need to grab the fork left and right of them
- Liveness vs. Safety

Deadlock

- A set of processes is deadlocked when every process in the set is waiting (blocked) for an event that can only be generated by some process in the set.

![Deadlock Diagram]

Starvation vs. Deadlock

- Starvation
  - Threads wait indefinitely (e.g., some other thread always gets the resource)

- Deadlock
  - Circular waiting for resources (all processes wait)

- Deadlock $\rightarrow$ starvation, but not the other way

Resource Allocation Graph

- Basic components of any resource allocation problem
  - Processes and resources

- Model the state of a computer system as a directed graph
  - $G = (V, E)$
  - $V$ = the set of vertices $= \{P_1, ..., P_n\} \cup \{R_1, ..., R_m\}$

- $E$ = the set of edges $= \{\text{edges from a resource to a process}\} \cup \{\text{edges from a process to a resource}\}$

![Resource Allocation Graph]
Theorem: If a resource allocation graph does not contain a cycle then no processes are deadlocked

A cycle in a RAG is a necessary condition for deadlock

Is the existence of a cycle a sufficient condition?

An Operational Definition of Deadlock

A set of processes is deadlocked iff the following conditions hold simultaneously:
1. Mutual exclusion is required for resource usage (serially useable)
2. All processes are in a “hold-and-wait” state
3. Preemption of resource usage is not allowed
4. Circular waiting exists (a cycle exists in the RAG)

Dealing with Deadlock

Deadlock Prevention
- Guarantee that deadlock does not occur

Deadlock Avoidance
- Only grant requests that can’t lead to deadlock, otherwise, block

Deadlock detection and recovery
- Admit the possibility of deadlock occurring and periodically check for it
Deadlock Prevention

- Attack one of the four conditions:
  1. Mutual Exclusion
     - Consequence of sharing in the presence of concurrency
  2. Hold-and-wait
     - Prevent process from requesting resource if it already holds another
     - Require process to acquire all resources at once
  3. No Preemption
     - Fundamental to atomicity
  4. Circular Wait
     - Here, there ARE things we can do.
     - E.g., impose an ordering of resources

Deadlock Avoidance

- Require each process to declare the maximum number of resources of each type it might need
- Examine each resource request and determine whether or not granting the request could lead to deadlock
- Define a set of vectors and matrices that characterize the current state of all resources and processes

\[
\begin{array}{cccc}
R_1 & R_2 & R_3 & \ldots & R_R \\
\begin{bmatrix}
P_1 \\
P_2 \\
P_3 \\
P_4 \\
P_5 \\
\end{bmatrix}
& \begin{bmatrix}
\begin{bmatrix}
R_1 & R_2 & R_3 & \ldots & R_R \\
\end{bmatrix}
& \begin{bmatrix}
n_{1,1} & n_{1,2} & n_{1,3} & \ldots & n_{1,R} \\
\end{bmatrix}
& \begin{bmatrix}
n_{2,1} & n_{2,2} \\
\end{bmatrix}
& \begin{bmatrix}
n_{3,1} & \vdots \\
\end{bmatrix}
& \begin{bmatrix}
n_{4,1} & \ldots & n_{4,R} \\
\end{bmatrix}
& \begin{bmatrix}
n_{5,1} & \ldots & n_{5,R} \\
\end{bmatrix}
\end{array}
\end{array}
\]

resource allocation state matrix

\[
\begin{align*}
\text{Max}_i &= \text{the maximum number of units of resource } j \text{ that the process } i \text{ will ever require simultaneously} \\
\text{Alloc}_i &= \text{the number of units of resource } j \text{ held by process } i \\
\text{Avail}_j &= \text{the number of units of resource } j \text{ that are unallocated}
\end{align*}
\]

Deadlock Avoidance – Banker’s Algorithm

- The OS will examine each resource request and determine whether or not granting the request could lead to deadlock
  - If, after we grant this request, if all processes simultaneously make their maximum claim, will the system deadlock?
  - Can we satisfy the maximum claims of processes in some order?
  - This is the Banker’s Algorithm (Dijkstra and Habermann)

<table>
<thead>
<tr>
<th>ALLOCATION</th>
<th>MAX CLAIM</th>
<th>AVAILABLE</th>
</tr>
</thead>
<tbody>
<tr>
<td>R_1 R_2 R_3 R_4</td>
<td>R_1 R_2 R_3 R_4</td>
<td>R_1 R_2 R_3 R_4</td>
</tr>
<tr>
<td>P_1 0 0 1 2</td>
<td>0 0 1 2</td>
<td>1 5 2 0</td>
</tr>
<tr>
<td>P_2 1 0 0 0</td>
<td>1 7 5 0</td>
<td>2 3 5 6</td>
</tr>
<tr>
<td>P_3 1 3 5 3</td>
<td>2 3 5 6</td>
<td>0 6 5 2</td>
</tr>
<tr>
<td>P_4 0 6 3 2</td>
<td>0 6 5 2</td>
<td>0 6 5 6</td>
</tr>
<tr>
<td>P_5 0 0 1 4</td>
<td>0 0 1 4</td>
<td>0 0 1 4</td>
</tr>
</tbody>
</table>

Deadlock Detection & Recovery

- How are deadlocks detected?
  - First, the detector must be aware of every thread that “holds” a resource.
    - Is this generally true?
    - What about users of atomic instructions?
    - What about users of pthread_mutex?
  - How would we recover from a deadlock?
    - We could release a lock, but what if data is inconsistent?
      - How do we detect that?
      - What would we do about it?
    - The scheduler could infer deadlocks by examining blocked processes/threads.
      - But what would it do about it?

Completely contained programming environments (like Java) can detect deadlocks, but besides reporting the deadlock, can’t do anything about it.
The Six Commandments

- Thou shalt always do things the same way
- Thou shalt always synchronize with locks and condition variables
- Thou shalt always acquire the lock at the beginning of a function and release it at the end
- Thou shalt always hold lock when operating on a condition variable
- Thou shalt always wait in a while loop
- (Almost) Never sleep()