Dinning Philosopher Problem

Repeat
  wait(chopstick[I]);
  wait(chopstick[I+1 mod 5]);
  ....
  Eat
  ....
  Signal(chopstick[I]);
  Signal(chopstick[I+1 mod 5]);
  ....
  Think
  ....
  Until false;

Is there a problem?  How can we avoid the problem?

At most 4 can eat together
wait till both are available
odd picks right/even picks left
Starvation?

Necessary Conditions for Deadlock

Mutual Exclusion
Hold and Wait
No Preemption
Circular Wait

Methods for Dealing

Avoidance
Prevention
Recover
Do nothing
Resource Allocation Graph

- A Set of All Resources \( R = \{ R_1, R_2, R_3, \ldots, R_m \} \)
- A Set of All Processes \( V = \{ V_1, V_2, V_3, \ldots, V_n \} \)
- A Set of Request Edges \( (P_i \rightarrow R_j) \)
- A Set of Assignment Edges \( (R_j \rightarrow P_i) \)

A process \( P_i \) requests instance of \( R_j \)
A process \( P_i \) holds an instance of \( R_j \)

A RAG with Cycles

- A RAG with no cycle \( \Rightarrow \) no deadlock
- A RAG with cycle \( \Rightarrow \)
  - if only one instance per resource type, then deadlock
  - otherwise, possibility of deadlock
Deadlock Prevention

- Prevent Mutual Exclusion?
  - Some resources are intrinsically nonsharable.

- Prevent Hold and Wait
  - Protocol-1: request everything at once and complete
  - Protocol-2: before every new request release all

- Prevent No preemption
  - If a new request cannot be allocated then take away all.
  - If a process is waiting by holding a resource which is requested by another new process, then take it away from the first process.

- Prevent Circular wait
  - Impose a total ordering on all resources. All request must be in order. Example:
    - tape 1, disk 5, printer 12
    - proof?

Deadlock Avoidance

- Deadlock Prevention schemes can reduce resource utilization and system throughput.
- Deadlock avoidance schemes utilizes some additional info (for example how the processes will request resources, what will be their maximum requests, etc.) to avoid deadlock.

- Safe State:
  - A state is safe, if the system can allocate resources to each process (up to its maximum) in at least one order and still avoid deadlock.
Bankers Algorithm

Available[M]:
Max[NxM]:
Allocation[NxM];
Need[NxM]: Need[i,j] = Max[i,j] - Allocation[i,j] Tavailable[M], and Tfinish[N]
(we will refer to entire vector by Need[i], or allocation[i])

1. Tavailable[i] = Available[i] for all j
   Tfinish[i] = false for all i
2. Find an i such that
   Tfinish[i] = false and
   Need[i] < Tavailable;
   If No such i exists go to step 4.
3. Tavail = Tavail + Allocation[i]  /*consider process i done*/
   Tfinish[i] = true;
   go to step 2.
4. If Tfinish[i] = true for all i then it is in safe state

Examples:

<table>
<thead>
<tr>
<th>Total 12</th>
<th>Max.</th>
<th>Allocation1</th>
</tr>
</thead>
<tbody>
<tr>
<td>P0</td>
<td>10</td>
<td>5</td>
</tr>
<tr>
<td>P1</td>
<td>4</td>
<td>2</td>
</tr>
<tr>
<td>P2</td>
<td>9</td>
<td>2</td>
</tr>
<tr>
<td>Is it Safe?</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Total A=10 B=5 C=7</th>
<th>Max.</th>
<th>Allocation</th>
</tr>
</thead>
<tbody>
<tr>
<td>ABC</td>
<td>ABC</td>
<td></td>
</tr>
<tr>
<td>P0</td>
<td>753</td>
<td>010</td>
</tr>
<tr>
<td>P1</td>
<td>322</td>
<td>200</td>
</tr>
<tr>
<td>P2</td>
<td>902</td>
<td>302</td>
</tr>
<tr>
<td>P3</td>
<td>222</td>
<td>211</td>
</tr>
<tr>
<td>P4</td>
<td>433</td>
<td>002</td>
</tr>
<tr>
<td>Available:</td>
<td>332</td>
<td></td>
</tr>
<tr>
<td>Is it safe? (yes)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

What if P1 requests additional 102?
What if now P4 requests additional 330?
What if P0 requests additional 020?
Questions?

- Is a deadlock state also an unsafe state?
- Does an unsafe state always lead to a deadlock state?

Recovery

- When should OS check?
  - How often deadlock is likely to occur?
  - How many processes are affected?
  - When often recovery should be initiated?
- Should we abort all processes involved?
- Should we abort one at a time?
- Should we preempt one resource at a time?
- Can there be starvation?
- Combined approach:
  - Internal resources (PCB etc.): use ordering.
  - Central memory: use preemption.
  - Job resources (printers etc.): Use avoidance.
  - Swappable space (backup scratch pad): use preallocation.