Administrivia

- Project discussion
  - Walk-through due Thursday to give you more time to ask questions
  - Individual, but please write your team information on the walkthrough
- Change in the project
  - Locks and condition variable code will be given to you
  - Extra part added instead
- Last time: Finished scheduling, started deadlock
- Today: continue with deadlock

Highest Response Ratio Next

<table>
<thead>
<tr>
<th>Process</th>
<th>Arrival Time</th>
<th>Burst Length</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>P2</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>P3</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>P4</td>
<td>7</td>
<td>2</td>
</tr>
</tbody>
</table>

- Intuition – favor short processes, but give credit for long processes that have been waiting a long time
- Optimize the normalized turnaround value
- Define the Response Ratio as $\frac{w + s}{n}$, where $w$ is the wait time and $s$ is the service time
- Non-preemptive (can we have a preemptive version? this was a homework problem last year)

Multi-Level Queue Scheduling

- Have multiple queues based on some property of the process (e.g., foreground/background)
- Schedule higher priority queues more often
- State Dependent Algorithms
  - Adapt the algorithm to the state of the system (e.g., how many processes are ready)
  - Example: adjust $q$, such that $q_n = \frac{w}{n}$ where $n$ is the number of ready processes

Multiple Level Feedback

- Intuition: react to the length of the burst instead of trying to guess it
- A process comes in into the highest priority queue
- Time quota runs out: process placed in the next lower queue
- When a process reaches lowest queue, it stays there
- Each Queue may have a different quota; for example, $2^nq$
- Solaris scheduling
Priority Scheduling

- Similar to your NachOS assignment (first part), schedule the highest priority process next
- Problem: low priority processes can starve
  - Possible Solution: Age the priority, so that a process waiting a long time gets higher priority
- Preemptive or non-preemptive?
- Shortest Job First is priority scheduling with process run time used as priority
- Priorities in use in most Commercial operating systems, including unix/linux

Cycles in the Resource Graph

- A system MAY be deadlocked if there is a cycle in the resource graph
- An arrow from a process to a resource box means that the process is waiting on a resource of that type
- An arrow from a resource to a process means that the process currently owns it
- Each "dot" in a resource box is an instance of that resource (e.g., multiple tape-drives)
- A system MAY be deadlocked if there is a cycle in the resource graph

Deadlock Example 2

- Although deadlock can occur, whether it actually happens depends on the order of events
What Types of Shared Resources are there?

- Permanent (Disks, CPUs, channels, critical Regions, etc...)
- Consumable (e.g., messages)
- Preemptable (CPU, Memory)
- Non-preemptable (but serially reusable)
- Discrete/Continuous
- Bounded/Infinite
- We are interested in non-preemptible, discrete, bounded resources

Recipe for a Deadlock

- Mutual Exclusion – only one process may use the resource at a time
- Hold and Wait – A process holds an allocated resource while waiting for others
- No Preemption
- Circular wait – A closed circle of processes exists where each process holds at least one resource that the next process in the chain needs

Handling Deadlock

Three General Approaches:
1. Use a protocol that will guarantee deadlock cannot occur
   - Deadlock prevention: ensure that one of the ingredients necessary for deadlock cannot happen
   - Deadlock avoidance: a smarter way of avoiding deadlock (later)
2. Allow deadlock to happen but detect it if it happens and recover
3. Do nothing (?); actually used by many OS’s including unix
   - If it is infrequent, why worry about it?
   - If user processes are deadlocked, they will eventually kill them (a slow form of deadlock recovery?)

Deadlock Prevention Techniques

- Prevent any of the ingredients of the Recipe
- Mutual Exclusion – resource dependent and cannot generally be prevented correctly
- Think about deadlock prevention policies based on the other three
- Policies can be very liberal (allow any process to get a resource if its available) to very conservative (serialization – only one process may get resources at a time)
Hierarchical Allocation

- Enumerate the resources (or classes of resources)
- Cannot take resources at a higher level until you have taken all the resources you need at all lower levels
- How does this prevent deadlock; what ingredient does it stop?
- What does this correspond to in the dining philosophers problem?

One shot Allocation Algorithm

- A process must acquire all the resources it will need at the beginning; if any of them is not available, the process is blocked
- Does this prevent deadlock?
- Which ingredient(s) does it prevent?
- Is this a good/bad policy; what are the problems?
- Is it liberal or conservative?
- Inefficient
- What does this correspond to in the dining philosophers problem?

Deadlock Prevention – Discussion

- Are these policies the best we can do?
- They are rather conservative. Can you come up with situations where these policies are too heavy handed?
- Liberal vs. Conservative revisited
- Major flow is that they do not consider the state of the system

Preemption

- If a process requires an unavailable resource, it must release all its resources and retry acquiring all of them together
- Any problems with this scheme?
- What are the difficulties? Are all resources preemtpable?
- A different flavor is if a process tries to acquire a resource owned by another process, the second process is forced to relinquish all its resources
- What does this correspond to in the dining philosophers problem?
Deadlock Avoidance

- Deadlock avoidance is a more sophisticated way of preventing deadlock that makes decisions based on the state of the system.
- Honor a resource request only if you can tell that it will not lead to deadlock (the resulting state is a *safe state*).
- How do I know that a state is safe? How can I guarantee that the processes are not already destined to deadlock?
- A state is safe if there exists an order [P1, P2, ..., Pn] for the processes to execute to completion:
  - P1 finishes using only the available resources.
  - P2 finishes using the available resources + P1's resources.
  - P3 finishes with available resources + P1 and P2's resources, etc..

Resource Allocation Graph Algorithm

- Can we use the resource allocation graph to determine if a state is safe?
  - Yes – but only for resources that have a single instance; why?
- Algorithm:
  - Every process must claim the resources it may need ahead of time.
  - Add claims to the graph as edges with the same direction as request edges but make them dashed to differentiate.
  - Allow a request if: (i) the resource is available; and (ii) converting the edge from claim to hold does not cause a cycle to happen.

Deadlock Avoidance – Definitions

- **Resources Vector**, \((R_1, R_2, R_3, \ldots)\), is the vector of resources the OS is managing. Example \((1, 3, 2, 64\text{ Mbyte}, \ldots)\) to say 1 printer, 3 disc devices, 2 CPUs and 64Mbyte memory.
- **Available Vector**, \((V_1, V_2, V_3, \ldots)\), is the vector of free resources; \(\forall_i V_i \leq R_i\).
- **Claim Matrix** (Max in book): Each Process defines the maximum of each resource expects to ask for:

\[
C = \begin{pmatrix}
C_{1,1} & C_{1,2} & \ldots & C_{1,R} \\
C_{2,1} & C_{2,2} & \ldots & C_{2,R} \\
\vdots & \vdots & \ddots & \vdots \\
C_{N,1} & C_{N,2} & \ldots & C_{N,R}
\end{pmatrix}
\]
Possible Algorithm – Initiation Denial

- Do not start a process $P_{N+1}$ unless its claims do not bring the total claims on any resource to be above $R$. More specifically:

$$\forall i, R_i > C_{N+1}, i + \sum_{k=1}^{N} C_{k,i}$$

- How can we do this for our dear philosophers?

- Claim: identical effect to One-shot allocation

- Assumes that all the processes will ask for the maximum resources at the same time

Banker’s Algorithm

$S =$ set of processes;

while ($S$ is not empty)(

1. Find a process $p$ such that
   foreach $i$ Need[$p,i$] <= $V[i]$

2. If impossible -- fail; state is unsafe

3. Remove $p$ from $S$;
   add $p$’s resources Available pool
)

- A safe state is a realizable state where there exits at least one sequence for the processes to run to completion

- Allow a resource request if the resulting state is safe

- How do we determine if a state is safe?

- Algorithm above is $O(n^2)$ – more efficient implementation exists (Habbermann’s algorithm)

Algorithm

- When a resource request system call occurs

- Check if request[$i$] is less than or equal to need[$p,i$] (why?)

- Check if request[$i$] is less than or equal to $V[i]$; otherwise block

- Pretend that you honor the request and see if the resulting state is safe:
  - $A[p,i] += request[i]$;
  - Need[$p,i$] -= request[$i$];
  - $V[i] -= request[i]$

- If the state is safe, allow request, otherwise block

- Still Conservative!! However, it is the most liberal policy that will guarantee no deadlock

Definitions (contd)

- Allocation Matrix: The resources currently held by each process

$$A = \begin{pmatrix}
A_{1,1} & A_{1,2} & \ldots & A_{1,R} \\
A_{2,1} & A_{2,2} & \ldots & A_{2,R} \\
\vdots & \vdots & \ddots & \vdots \\
A_{N,1} & A_{N,2} & \ldots & A_{N,R}
\end{pmatrix}$$

- Need Matrix is $C - A$: how many more resources could a process still ask for

- A State is a set of values for the Resource and Available vectors as well as the Claim and Allocation Matrix
  - A realizable state is a state that “makes sense”
    - No one claim is for more than the total resources available
    - No process is holding more than its claim
    - The sum of all held fields is not more than the total resources available
Example

<table>
<thead>
<tr>
<th>Process</th>
<th>Allocation A</th>
<th>Allocation B</th>
<th>Allocation C</th>
<th>Claim A</th>
<th>Claim B</th>
<th>Claim C</th>
</tr>
</thead>
<tbody>
<tr>
<td>P0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>7</td>
<td>5</td>
<td>3</td>
</tr>
<tr>
<td>P1</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>3</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>P2</td>
<td>3</td>
<td>0</td>
<td>2</td>
<td>9</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>P3</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>P4</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>4</td>
<td>3</td>
<td>3</td>
</tr>
</tbody>
</table>

\[ V = [3 \ 3 \ 2] \]

- Is the current state safe? Is it deadlocked?
  - Generate need matrix
  - Apply safety algorithm
- New request from P1 (1, 0, 2); allow it or not?

Deadlock Detection

- Never block a process if enough resources are available
- Detect deadlock and recover once it occurs
- How to detect deadlock? (think about detecting a deadlock-free state)
  - Add a matrix \( Q \) for each process currently outstanding resource requests
  - Update banker’s algorithm to pick a process that “may” finish (given its currently held resources + outstanding requests)
    * Replace need matrix with the outstanding request matrix
    * Apply “safety” algorithm; if it succeeds, no deadlock – why?
- When to run the deadlock detection algorithm?
- What to do if there is deadlock?

What to do if Deadlock is Detected?

- Two related approaches
  - Kill processes
  - Preempt some resources
- Kill process
  - Kill all deadlocked processes?
  - Kill one by one (priority based?)
- Preempt resources; but resources are not preemptable??
  - Process must be restored to a consistent state
  - Must roll back to a previous point in the execution (need checkpoints)
  - One available checkpoint is the initial process image – this becomes equivalent to killing the process and restarting it
  - Identical to the “preempt” deadlock prevention algorithm
- Starvation is a consideration