Administrivia

- NachOS discussion? Again, please get started early
  - Walkthrough due 3/11

- Midterm on 3/6
  - Review session is also on 3/6, 8:30–9:55AM EB 110 (sorry!)
  - Include CPU scheduling?

- Today
  - Finish Process Scheduling
Walkthrough

• Due 3/11
  – one per person – please email me team information (2 preferred, 1 is fine, 3 please talk to me)

• What is in a walkthrough? three things
  – A brief trace of the code as it is now explaining how it produces the output that it does
  – Your plan for implementing any two parts in the project
  – Answers to the questions in the walkthrough link on class page

• 2-3 pages max.
Potential Criteria

- Response time: minimize the amount of delay experienced by users (average? max? variance?)
- Turnaround time: minimize time between beginning and end of each job (normalized)
- CPU utilization: maximize the amount of work done by the CPU
- Throughput: maximize the number of processes completed per time unit
- Other?
We went over the following algorithms

- First Come First Serve: problems?
- Shortest Job First
- Shortest Remaining Job (preemptive SJF)
- Round Robin
- Gantt charts (space time charts) are used to illustrate scheduling
### 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 \( \frac{w+s}{s} \), 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 $q_n = \frac{q_0}{n}$ 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
- Split the ready queue into multiple queues
- A process comes in into the highest priority queue
- When its time quota runs out, it gets placed in the next lower queue
- When a process reaches lowest queue, it stays there (getting scheduled round-robin with other processes in this class)
- Each Queue may have a different quota; for example, $2^nq$
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
Recap and Discussion

- Try to build the best algorithm to optimize your criteria
- Several scheduling criteria exist, may be conflicting
- Preemptive algorithms give fairer access to the CPU, but have more overhead because of the redundant process switches
- Scheduling in the presence of threads
- What if I start many processes so that I get more chances to be scheduled on the CPU – fair-share scheduling
Solaris Scheduling

- Multi-level feedback
  - If quantum expires, new priority is lower than old priority

- Priority inversely proportional to quantum (higher priority = smaller quantum)
  - Good response time for interactive processes; good throughput for CPU bound processes
Linux scheduling

- Two algorithms: time-sharing and real-time

- Time-sharing
  - Prioritized credit-based – process with most credits is scheduled next
  - Credit subtracted when timer interrupt occurs
  - When credit = 0, another process chosen
  - When all processes have credit = 0, recrediting occurs
    * Based on factors including priority and history

- Real-time
  - Soft real-time
  - Posix.1b compliant – two classes
    * FCFS and RR
    * Highest priority process always runs first
Performance Evaluation

- How do you analyze the performance of a scheduling algorithm?
- New theories/inventions: trial and error? is there a method to the madness?
- How do scientists/engineers study their systems? Example
  - A Mechanical engineer who wants to evaluate an improvement to a car engine design
  - A mathematician that wants to evaluate a new heuristic for the travelling salesman problem
  - A biomedical who wants to test a new laser based technique for brain surgery
  - A nuclear physicist that wants to test a method to increase the effect of H-Bombs
Three Methods for Scientific Experiments

- Mathematical Modeling: Find a way to model the problem using math, and see how it works given your set of initial variables –
  - Pros/Cons? How can we apply that to scheduling?
- Physical Modeling: Build the thing (or maybe a scaled down model?) and see how it works
  - Why not just do that?
- Computer Modeling (simulation): Build a computer model of it and simulate it
  - You are doing a little of that in your second NachOS assignment
Queueing Theory

- The “math” for scheduling and several other computer science problems

- We will look at very basic queueing theory; most to understand what they are, not necessarily how to use them

- Simplest Model: Single Queue Single Server
Single Queue Single Server

- $q$, items in the system; $w$, items in the queue; $\lambda$, arrival rate
- Clients arrive to the system according to some arrival rate $\lambda$
- Each stands in the queue, and when it is its time, it get serviced (taking a variable service time $t_s$)
- Can analyze different models depending on the distributions assumed for arrival and service rates (e.g., exponential, normal, poisson, etc..)
Some General Results

• The queue size will be large if the system is saturated (requests are coming in about as fast as they are processed)

• Maximum theoretical arrival rate $\frac{1}{T_s}$, after that the queue will grow in size indefinitely ($T_s$ is the average service time)

• Little’s Theorem: number of items in a system (queue or otherwise) is the arrival rate multiplied by the average wait time. So:
  \begin{align*}
  q &= \lambda T_q \\
  w &= \lambda T_w \\
  T_q &= T_w + T_s
  \end{align*}

• Applying Little’s Theorem to the CPU:
  \begin{align*}
  \rho &= \lambda T_s \\
  q &= w + \rho
  \end{align*}
Analytical Models – Discussion

- Can model multiple queues, multiple servers, chained queues, etc...

- Are the results accurate?
  - What if assumed distributions are inaccurate?
  - How can I model specific policies?
  - How can I factor in switching overhead?

- Limited: cannot model a lot of scheduling policies accurately (e.g., cannot track history of process behavior)
Is Simulation any Better?

- Physical modeling: build the thing and evaluate it
- Not always possible, economic, acceptable, or convenient
- Simulation: Advantages
  - Highly flexible
  - Highly observable
  - Can be cheaper than physical modeling
  - Can be more sophisticated than analytical modeling (especially for complex/dynamic systems)
- Disadvantages/Pitfalls
  - Validation and Verification (V and V)
  - Can require extensive modeling time, execution time, or both
  - Choosing aspects to model
  - Selecting inputs
Roadmap

- So far we have:
  - Overviewed Computer Organization and OS Organization
  - Process and process management
  - Threads
  - Concurrency and Synchronization
  - Scheduling

- Next up:
  - Deadlock and Starvation
  - Memory Management/Virtual Memory
  - File Management
  - Input Output Management
  - Case Studies
  - Optional (Distributed OS, Networking, Security, etc...)
Deadlock – Example

Scenario 1:
A requests a tape drive; OS resource manager allocates it
B requests a tape drive; resource manager (RM) allocates
A requests a tape drive; not available, RM blocks A
B requests a tape drive; not available, RM blocks B
... deadlock ...

Scenario 2:
A requests a tape drive; Clever RM allocates it
B requests a tape drive; Clever RM blocks B
A requests a tape drive; Clever RM allocates it
A releases tape drive; Clever RM gives it to B
B requests a tape drive; not available, OS blocks B
A release tape drive; Clever RM gives it to B
B releases tape drive
B releases tape drive

• What causes deadlock?

• What can we do to solve the deadlock problem?
Although deadlock can occur, whether it actually happens depends on the order of events happening.
Deadlock

- *Permanent* blocking of a set of processes
  - Occurs when there is competition for shared resources (or when the processes communicate)
  - What types of resources can cause deadlock?
  - Sometimes symptom of incorrectly synchronized concurrent activities

- Under what conditions will deadlock occur?

- Why didn't we see it in memory management or CPU scheduling?

- How can we handle it?
• Each “dot” in a resource box is an instance of that resource (e.g., multiple tape-drives)

• An arrow from a resource instance to a process means that the process currently owns it

• An arrow from a process to a resource box means that a process is waiting on a resource of that type

• A system MAY be deadlocked if there is a cycle in the resource graph
Cycles in the Resource Graph

\[
\begin{align*}
\text{R1} & \rightarrow \text{P1} \\
\text{P1} & \rightarrow \text{R2} \\
\text{R2} & \rightarrow \text{P2} \\
\text{P2} & \rightarrow \text{R3} \\
\text{R3} & \rightarrow \text{P3} \\
\text{P3} & \rightarrow \text{R4} \\
\text{R4} & \rightarrow \text{P1}
\end{align*}
\]
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?)