Recall: Scheduling Policy Goals/Criteria

- Minimize Response Time
  - Minimize elapsed time to do an operation (or job)
  - Response time is what the user sees:
    » Time to echo a keystroke in editor
    » Time to compile a program
  - Real-time Tasks: Must meet deadlines imposed by World
- Maximize Throughput
  - Maximize operations (or jobs) per second
  - Throughput related to response time, but not identical:
    » Minimizing response time will lead to more context switching than
      if you only maximized throughput
  - Two parts to maximizing throughput
    » Minimize overhead (for example, context-switching)
    » Efficient use of resources (CPU, disk, memory, etc)
- Fairness
  - Share CPU among users in some equitable way
  - Fairness is not minimizing average response time:
    » Better average response time by making system less fair

Recall: Example of RR with Time Quantum = 20

- Example:

<table>
<thead>
<tr>
<th>Process</th>
<th>Burst Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>P_1</td>
<td>53</td>
</tr>
<tr>
<td>P_2</td>
<td>8</td>
</tr>
<tr>
<td>P_3</td>
<td>68</td>
</tr>
<tr>
<td>P_4</td>
<td>24</td>
</tr>
</tbody>
</table>

- The Gantt chart is:

<table>
<thead>
<tr>
<th>P_1</th>
<th>P_2</th>
<th>P_3</th>
<th>P_4</th>
<th>P_1</th>
<th>P_3</th>
<th>P_4</th>
<th>P_1</th>
<th>P_3</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>20</td>
<td>28</td>
<td>48</td>
<td>68</td>
<td>88</td>
<td>108</td>
<td>125</td>
<td>145</td>
</tr>
</tbody>
</table>

- Waiting time for
  \[ P_1 = (68-20) + (112-88) = 72 \]
  \[ P_2 = (20-0) = 20 \]
  \[ P_3 = (28-20) + (88-48) + (125-108) = 85 \]
  \[ P_4 = (48-0) + (108-68) = 88 \]

- Average waiting time = \( (72 + 20 + 85 + 88) / 4 = 66 \frac{1}{4} \)
- Average completion time = \( (125 + 28 + 153 + 112) / 4 = 104 \frac{1}{2} \)

- Thus, Round-Robin Pros and Cons:
  - Better for short jobs, Fair (+)
  - Context-switching time adds up for long jobs (-)
Comparisons between FCFS and Round Robin

- Assuming zero-cost context-switching time, is RR always better than FCFS?
- Simple example: 10 jobs, each take 100s of CPU time
  RR scheduler quantum of 1s
  All jobs start at the same time
- Completion Times:

<table>
<thead>
<tr>
<th>Job #</th>
<th>FIFO</th>
<th>RR</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>100</td>
<td>991</td>
</tr>
<tr>
<td>2</td>
<td>200</td>
<td>992</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>9</td>
<td>900</td>
<td>999</td>
</tr>
<tr>
<td>10</td>
<td>1000</td>
<td>1000</td>
</tr>
</tbody>
</table>

- Both RR and FCFS finish at the same time
- Average response time is much worse under RR!
  » Bad when all jobs same length
- Also: Cache state must be shared between all jobs with RR but can be devoted to each job with FIFO
- Total time for RR longer even for zero-cost switch!

Earlier Example with Different Time Quantum

<table>
<thead>
<tr>
<th>Quantum</th>
<th>P1</th>
<th>P2</th>
<th>P3</th>
<th>P4</th>
<th>Average</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>32</td>
<td>0</td>
<td>85</td>
<td>8</td>
<td>31⅓</td>
</tr>
<tr>
<td>8</td>
<td>84</td>
<td>22</td>
<td>85</td>
<td>57</td>
<td>62</td>
</tr>
<tr>
<td>32</td>
<td>8</td>
<td>20</td>
<td>85</td>
<td>58</td>
<td>61⅔</td>
</tr>
<tr>
<td>64</td>
<td>8</td>
<td>8</td>
<td>85</td>
<td>56</td>
<td>57⅔</td>
</tr>
<tr>
<td>96</td>
<td>8</td>
<td>10</td>
<td>85</td>
<td>68</td>
<td>61⅔</td>
</tr>
<tr>
<td>128</td>
<td>72</td>
<td>20</td>
<td>85</td>
<td>88</td>
<td>66⅔</td>
</tr>
<tr>
<td>160</td>
<td>68</td>
<td>145</td>
<td>0</td>
<td>121</td>
<td>83⅓</td>
</tr>
<tr>
<td>192</td>
<td>85</td>
<td>153</td>
<td>32</td>
<td>69⅓</td>
<td></td>
</tr>
<tr>
<td>256</td>
<td>137</td>
<td>30</td>
<td>153</td>
<td>81</td>
<td>100⅓</td>
</tr>
<tr>
<td>384</td>
<td>135</td>
<td>28</td>
<td>153</td>
<td>82</td>
<td>99½</td>
</tr>
<tr>
<td>512</td>
<td>133</td>
<td>16</td>
<td>153</td>
<td>80</td>
<td>95½</td>
</tr>
<tr>
<td>640</td>
<td>135</td>
<td>18</td>
<td>153</td>
<td>92</td>
<td>99½</td>
</tr>
<tr>
<td>768</td>
<td>125</td>
<td>28</td>
<td>153</td>
<td>112</td>
<td>104⅓</td>
</tr>
<tr>
<td>992</td>
<td>121</td>
<td>153</td>
<td>68</td>
<td>145</td>
<td>121⅔</td>
</tr>
</tbody>
</table>

Handling Differences in Importance: Strict Priority Scheduling

- Execution Plan
  » Always execute highest-priority runnable jobs to completion
  » Each queue can be processed in RR with some time-quantum
- Problems:
  » Starvation:
    » Lower priority jobs don’t get to run because higher priority jobs
  » Deadlock: Priority Inversion
    » Not strictly a problem with priority scheduling, but happens when low priority task has lock needed by high-priority task
    » Usually involves third, intermediate priority task that keeps running even though high-priority task should be running
- How to fix problems?
  » Dynamic priorities – adjust base-level priority up or down based on heuristics about interactivity, locking, burst behavior, etc.

Scheduling Fairness

- What about fairness?
  » Strict fixed-priority scheduling between queues is unfair (run highest, then next, etc):
    » long running jobs may never get CPU
    » Urban legend: In Multics, shut down machine, found 10-year-old job ⇒ Ok, probably not…
  » Must give long-running jobs a fraction of the CPU even when there are shorter jobs to run
  » Tradeoff: fairness gained by hurting avg response time!
Scheduling Fairness

- How to implement fairness?
  - Could give each queue some fraction of the CPU
    » What if one long-running job and 100 short-running ones?
    » Like express lanes in a supermarket—sometimes express lanes get so long, get better service by going into one of the other lines
  - Could increase priority of jobs that don’t get service
    » What is done in some variants of UNIX
    » This is ad hoc—what rate should you increase priorities?
    » And, as system gets overloaded, no job gets CPU time, so everyone increases in priority⇒Interactive jobs suffer

Lottery Scheduling

- Yet another alternative: Lottery Scheduling
  - Give each job some number of lottery tickets
  - On each time slice, randomly pick a winning ticket
    » NOTE: Not a “real” random number generator; instead pseudo-random number generators can make sure that every ticket picked once before repeating!
  - On average, CPU time is proportional to number of tickets given to each job

  - How to assign tickets?
    – To help with responsiveness, give short running jobs more tickets, long running jobs get fewer tickets
    – To avoid starvation, every job gets at least one ticket (everyone makes progress)
  - Advantage over strict priority scheduling: behaves gracefully as load changes
    – Adding or deleting a job affects all jobs proportionally, independent of how many tickets each job possesses

Lottery Scheduling Example (Cont.)

- Lottery Scheduling Example
  - Assume short jobs get 10 tickets, long jobs get 1 ticket

<table>
<thead>
<tr>
<th># short jobs/ # long jobs</th>
<th>% of CPU each short jobs gets</th>
<th>% of CPU each long jobs gets</th>
</tr>
</thead>
<tbody>
<tr>
<td>1/1</td>
<td>91%</td>
<td>9%</td>
</tr>
<tr>
<td>0/2</td>
<td>N/A</td>
<td>50%</td>
</tr>
<tr>
<td>2/0</td>
<td>50%</td>
<td>N/A</td>
</tr>
<tr>
<td>10/1</td>
<td>9.9%</td>
<td>0.99%</td>
</tr>
<tr>
<td>1/10</td>
<td>50%</td>
<td>5%</td>
</tr>
</tbody>
</table>

  - What if too many short jobs to give reasonable response time?
    » If load average is 100, hard to make progress
    » One approach: log some user out

How to Evaluate a Scheduling algorithm?

- Deterministic modeling
  – takes a predetermined workload and compute the performance of each algorithm for that workload
- Queueing models
  – Mathematical approach for handling stochastic workloads
- Implementation/Simulation:
  – Build system which allows actual algorithms to be run against actual data – most flexible/general
How to Handle Simultaneous Mix of Diff Types of Apps?

• Consider mix of interactive and high throughput apps:
  – How to best schedule them?
  – How to recognize one from the other?
    » Do you trust app to say that it is “interactive”?
  – Should you schedule the set of apps identically on servers, workstations, pads, and cellphones?

• For instance, is Burst Time (observed) useful to decide which application gets CPU time?
  – Short Bursts ⇒ Interactivity ⇒ High Priority?

• Assumptions encoded into many schedulers:
  – Apps that sleep a lot and have short bursts must be interactive apps – they should get high priority
  – Apps that compute a lot should get low(er?) priority, since they won’t notice intermittent bursts from interactive apps

• Hard to characterize apps:
  – What about apps that sleep for a long time, but then compute for a long time?
  – Or, what about apps that must run under all circumstances (say periodically)

What if we Knew the Future?

• Could we always mirror best FCFS?

• Shortest Job First (SJF):
  – Run whatever job has least amount of computation to do
  – Sometimes called “Shortest Time to Completion First” (STCF)

• Shortest Remaining Time First (SRTF):
  – Preemptive version of SJF: if job arrives and has a shorter time to completion than the remaining time on the current job, immediately preempt CPU
  – Sometimes called “Shortest Remaining Time to Completion First” (SRTCF)

• These can be applied to whole program or current CPU burst
  – Idea is to get short jobs out of the system
  – Big effect on short jobs, only small effect on long ones
  – Result is better average response time

Administrivia

• Midterm 1 is Thursday (2/27)!
  – Topics: All material up to Today (emphasis on scheduling in today’s lecture)
  – You get 1 sheet of handwritten notes, both sides

• Review Session: Tonight after class
  – Right here from 6:30-8:00pm (after class)

• Watch Piazza for Midterm room assignments:
  – There are 5 rooms!
  – DSP midterms are separate and you should have received email from us. Let us know immediately if that is not the case!

Discussion

• SJF/SRTF are the best you can do at minimizing average response time
  – Provably optimal (SJF among non-preemptive, SRTF among preemptive)
  – Since SRTF is always at least as good as SJF, focus on SRTF

• Comparison of SRTF with FCFS
  – What if all jobs the same length?
    » SRTF becomes the same as FCFS (i.e. FCFS is best can do if all jobs the same length)
  – What if jobs have varying length?
    » SRTF: short jobs not stuck behind long ones
Example to illustrate benefits of SRTF

- Three jobs:
  - A, B: both CPU bound, run for week
  - C: I/O bound, loop 1ms CPU, 9ms disk I/O
  - If only one at a time, C uses 90% of the disk, A or B could use 100% of the CPU
- With FCFS:
  - Once A or B get in, keep CPU for two weeks
- What about RR or SRTF?
  - Easier to see with a timeline

SRTF Example continued:

- Disk Utilization: 9/201 ~ 4.5%
- Disk Utilization: ~90% but lots of wakeups!
- Disk Utilization: 90%

SRTF Further discussion

- Starvation
  - SRTF can lead to starvation if many small jobs!
  - Large jobs never get to run
- Somehow need to predict future
  - How can we do this?
  - Some systems ask the user
    - When you submit a job, have to say how long it will take
    - To stop cheating, system kills job if takes too long
  - But: hard to predict job’s runtime even for non-malicious users
- Bottom line, can’t really know how long job will take
  - However, can use SRTF as a yardstick for measuring other policies
  - Optimal, so can’t do any better
- SRTF Pros & Cons
  - Optimal (average response time) (+)
  - Hard to predict future (-)
  - Unfair (-)

Predicting the Length of the Next CPU Burst

- Adaptive: Changing policy based on past behavior
  - CPU scheduling, in virtual memory, in file systems, etc
  - Works because programs have predictable behavior
    - If program was I/O bound in past, likely in future
    - If computer behavior were random, wouldn’t help
- Example: SRTF with estimated burst length
  - Use an estimator function on previous bursts:
    \[ \tau_n = f(t_{n-1}, t_{n-2}, t_{n-3}, \ldots) \]
  - Function \( f \) could be one of many different time series estimation schemes (Kalman filters, etc)
  - For instance, exponential averaging
    \[ \tau_n = \alpha \tau_{n-1} + (1-\alpha)\tau_{n-1} \] with \( 0 < \alpha \leq 1 \)
Multi-Level Feedback Scheduling

- Another method for exploiting past behavior (first use in CTSS)
  - Multiple queues, each with different priority
    - Higher priority queues often considered “foreground” tasks
  - Each queue has its own scheduling algorithm
    - e.g. foreground – RR, background – FCFS
    - Sometimes multiple RR priorities with quantum increasing exponentially (highest: 1ms, next: 2ms, next: 4ms, etc)

- Adjust each job’s priority as follows (details vary)
  - Job starts in highest priority queue
  - If timeout expires, drop one level
  - If timeout doesn’t expire, push up one level (or to top)

Long-Running Compute Tasks Demoted to Low Priority

Scheduling Details

- Result approximates SRTF:
  - CPU bound jobs drop like a rock
  - Short-running I/O bound jobs stay near top
- Scheduling must be done between the queues
  - Fixed priority scheduling:
    - serve all from highest priority, then next priority, etc.
  - Time slice:
    - each queue gets a certain amount of CPU time
    - e.g., 70% to highest, 20% next, 10% lowest

Case Study: Linux O(1) Scheduler

- Priority-based scheduler: 140 priorities
  - 40 for “user tasks” (set by “nice”), 100 for “Realtime/Kernel”
  - Lower priority value \(\Rightarrow\) higher priority (for nice values)
  - Highest priority value \(\Rightarrow\) Lower priority (for realtime values)
  - All algorithms \(O(1)\)
    - Timeslices/priorities/interactivity credits all computed when job finishes time slice
    - 140-bit bit mask indicates presence or absence of job at given priority level
- Two separate priority queues: “active” and “expired”
  - All tasks in the active queue use up their timeslices and get placed on the expired queue, after which queues swapped
  - Timeslice depends on priority – linearly mapped onto timeslice range
    - Like a multi-level queue (one queue per priority) with different timeslice at each level
    - Execution split into “Timeslice Granularity” chunks – round robin through priority

Countermeasure: user action that can foil intent of the OS designers

- For multilevel feedback, put in a bunch of meaningless I/O to keep job’s priority high
- Of course, if everyone did this, wouldn’t work!
- Example of Othello program:
  - Playing against competitor, so key was to do computing at higher priority the competitors.
    - Put in printf’s, ran much faster!
O(1) Scheduler Continued

- Heuristics
  - User-task priority adjusted ±5 based on heuristics
    - \( p->\text{sleep\_avg} = \text{sleep\_time} - \text{run\_time} \)
    - Higher \( \text{sleep\_avg} \) ⇒ more I/O bound the task, more reward (and vice versa)
  - Interactive Credit
    - Earned when a task sleeps for a “long” time
    - Spend when a task runs for a “long” time
    - IC is used to provide hysteresis to avoid changing interactivity for temporary changes in behavior
  - However, “interactive tasks” get special dispensation
    - To try to maintain interactivity
    - Placed back into active queue, unless some other task has been starved for too long…

- Real-Time Tasks
  - Always preempt non-RT tasks
  - No dynamic adjustment of priorities
  - Scheduling schemes:
    - SCHED_FIFO: preempts other tasks, no timeslice limit
    - SCHED_RR: preempts normal tasks, RR scheduling amongst tasks of same priority

Linux Completely Fair Scheduler (CFS)

- First appeared in 2.6.23, modified in 2.6.24
- “CFS doesn't track sleeping time and doesn't use heuristics to identify interactive tasks—it just makes sure every process gets a fair share of CPU within a set amount of time given the number of runnable processes on the CPU.”
- Inspired by Networking “Fair Queueing”
  - Each process given their fair share of resources
  - Models an “ideal multitasking processor” in which \( N \) processes execute simultaneously as if they truly got \( 1/N \) of the processor
    - Tries to give each process an equal fraction of the processor
  - Priorities reflected by weights such that increasing a task’s priority by 1 always gives the same fractional increase in CPU time – regardless of current priority

CFS (Continued)

- Idea: track amount of “virtual time” received by each process when it is executing
  - Take real execution time, scale by weighting factor
    - higher priority ⇒ real time divided by larger weight
    - Actually – multiply by sum of all weights/current weight
  - Keep virtual time advancing at same rate
- Targeted latency (\( T_L \)): period of time after which all processes get to run at least a little
  - Each process runs with quantum \( (W_p/\sum W_i) \times T_L \)
  - Never smaller than “minimum granularity”
- Use of Red-Black tree to hold all runnable processes as sorted on vruntime variable
  - \( O(\log n) \) time to perform insertions/deletions
    - Cash the item at far left (item with earliest vruntime)
  - When ready to schedule, grab version with smallest vruntime
    (which will be item at the far left).

CFS Examples

- Suppose Targeted latency = 20ms, Minimum Granularity = 1ms
- Two CPU bound tasks with same priorities
  - Both switch with 10ms
- Nice values scale weights exponentially: Weight=1024/(1.25)^nice
- Two CPU bound tasks separated by nice value of 5
  - One task gets 5ms, another gets 15ms
- 40 tasks: each gets 1ms (no longer totally fair)
- One CPU bound task, one interactive task same priority
  - While interactive task sleeps, CPU bound task runs and increments vruntime
  - When interactive task wakes up, runs immediately, since it is behind on vruntime
- Group scheduling facilities (2.6.24)
  - Can give fair fractions to groups (like a user or other mechanism for grouping processes)
  - So, two users, one starts 1 process, other starts 40, each will get 50% of CPU
Real-Time Scheduling (RTS)

- Efficiency is important but **predictability** is essential:
  - We need to predict with confidence worst case response times for systems
  - In RTS, performance guarantees are:
    » Task- and/or class centric and often ensured a priori
  - In conventional systems, performance is:
    » System/throughput oriented with post-processing (… wait and see …)
  - Real-time is about enforcing predictability, and does not equal fast computing!!!

- **Hard Real-Time**
  - **Attempt to meet all deadlines**
  - EDF (Earliest Deadline First), LLF (Least Laxity First), RMS (Rate-Monotonic Scheduling), DM (Deadline Monotonic Scheduling)

- **Soft Real-Time**
  - **Attempt to meet deadlines with high probability**
  - Minimize miss ratio / maximize completion ratio (firm real-time)
  - Important for multimedia applications
  - CBS (Constant Bandwidth Server)

---

Example: Workload Characteristics

- Tasks are preemptable, independent with arbitrary arrival (=release) times
- Tasks have deadlines (D) and known computation times (C)
- Example Setup:
### Choosing the Right Scheduler

<table>
<thead>
<tr>
<th>I Care About:</th>
<th>Then Choose:</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU Throughput</td>
<td>FCFS</td>
</tr>
<tr>
<td>Avg. Response Time</td>
<td>SRTF Approximation</td>
</tr>
<tr>
<td>I/O Throughput</td>
<td>SRTF Approximation</td>
</tr>
<tr>
<td>Fairness (CPU Time)</td>
<td>Linux CFS</td>
</tr>
<tr>
<td>Fairness - Wait Time to Get CPU</td>
<td>Round Robin</td>
</tr>
<tr>
<td>Meeting Deadlines</td>
<td>EDF</td>
</tr>
<tr>
<td>Favoring Important Tasks</td>
<td>Priority</td>
</tr>
</tbody>
</table>

### A Final Word On Scheduling

- **When do the details of the scheduling policy and fairness really matter?**
  - When there aren’t enough resources to go around
- **When should you simply buy a faster computer?**
  - (Or network link, or expanded highway, or …)
  - One approach: Buy it when it will pay for itself in improved response time
    - Perhaps you’re paying for worse response time in reduced productivity, customer angst, etc…
    - Might think that you should buy a faster X when X is utilized 100%, but usually, response time goes to infinity as utilization approaches 100%
- **An interesting implication of this curve:**
  - Most scheduling algorithms work fine in the “linear” portion of the load curve, fail otherwise
  - Argues for buying a faster X when hit “knee” of curve

### Summary (1 of 2)

- **Scheduling Goals:**
  - Minimize Response Time (e.g. for human interaction)
  - Maximize Throughput (e.g. for large computations)
  - Fairness (e.g. Proper Sharing of Resources)
  - Predictability (e.g. Hard/Soft Realtime)
- **Round-Robin Scheduling:**
  - Give each thread a small amount of CPU time when it executes; cycle between all ready threads
  - Pros: Better for short jobs
- **Shortest Job First (SJF)/Shortest Remaining Time First (SRTF):**
  - Run whatever job has the least amount of computation to do/least remaining amount of computation to do
  - Pros: Optimal (average response time)
  - Cons: Hard to predict future, Unfair
- **Multi-Level Feedback Scheduling:**
  - Multiple queues of different priorities and scheduling algorithms
  - Automatic promotion/demotion of process priority in order to approximate SJF/SRTF

### Summary (2 of 2)

- **Lottery Scheduling:**
  - Give each thread a priority-dependent number of tokens (short tasks ⇒ more tokens)
- **Linux CFS Scheduler: Fair fraction of CPU**
  - Approximates a “ideal” multitasking processor
- **Realtime Schedulers such as EDF**
  - Guaranteed behavior by meeting deadlines
  - Realtime tasks defined by tuple of compute time and period
  - Schedulability test: is it possible to meet deadlines with proposed set of processes?