Recall: CPU Scheduling

- Earlier, we talked about the life-cycle of a thread
  - Active threads work their way from Ready queue to
    Running to various waiting queues.
- Question: How is the OS to decide which of several
  tasks to take off a queue?
  - Obvious queue to worry about is ready queue
  - Others can be scheduled as well, however
- Scheduling: deciding which threads are given access
to resources from moment to moment

Recall: Round Robin (RR)

- Round Robin Scheme
  - Each process gets a small unit of CPU time
    \((\text{time quantum})\), usually 10-100 milliseconds
  - After quantum expires, the process is preempted
    and added to the end of the ready queue.
  - \(n\) processes in ready queue and time quantum is \(q\) \(\Rightarrow\)
    - Each process gets \(1/n\) of the CPU time
    - In chunks of at most \(q\) time units
    - No process waits more than \((n-1)q\) time units
- Performance
  - \(q\) large \(\Rightarrow\) FIFO
  - \(q\) small \(\Rightarrow\) Interleaved (really small \(\Rightarrow\) hyperthreading?)
  - \(q\) must be large with respect to context switch, otherwise
    overhead is too high (all overhead)
- How do you choose time slice?
  - What if too big?
    - Response time suffers
  - What if infinite (∞)?
    - Get back FIFO
  - What if time slice too small?
    - Throughput suffers!
Example of RR with Time Quantum = 20

- Example:
  - Process | Burst Time
  - P₁ | 53
  - P₂ | 8
  - P₃ | 68
  - P₄ | 24

  - The Gantt chart is:

    | P₁ | P₂ | P₃ | P₄ | P₁ | P₃ | P₄ | P₁ | P₃ | P₃ |
    | 0  | 20 | 28 | 48 | 68 | 88 | 108| 125| 145| 153|

  - Waiting time for:
    - \( P₁ = (20 - 0) + (88 - 0) = 108 \)
    - \( P₂ = (20 - 0) + (88 - 0) = 108 \)
    - \( P₃ = (28 - 0) + (125 - 108) = 85 \)
    - \( P₄ = (48 - 0) + (108 - 88) = 88 \)
  - Average waiting time = \( 108 + 108 + 85 + 88 \)/4 = 98
  - Average completion time = \( 125 + 28 + 153 + 112 \)/4 = 104

- 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:
    | Job # | FIFO | RR |
    |-------|------|----|
    | 1     | 100  | 991|
    | 2     | 200  | 992|
    | 3     | 900  | 999|
    | 4     | 1000 | 1000|

  - 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></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0 8 32 85 153</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<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>Best FCFS</td>
<td>32</td>
<td>0</td>
<td>85</td>
<td>8</td>
<td>31½</td>
</tr>
<tr>
<td>Q = 1</td>
<td>84</td>
<td>22</td>
<td>85</td>
<td>57</td>
<td>62</td>
</tr>
<tr>
<td>Q = 5</td>
<td>82</td>
<td>20</td>
<td>85</td>
<td>58</td>
<td>61½</td>
</tr>
<tr>
<td>Q = 8</td>
<td>80</td>
<td>8</td>
<td>85</td>
<td>56</td>
<td>57½</td>
</tr>
<tr>
<td>Q = 10</td>
<td>82</td>
<td>10</td>
<td>85</td>
<td>68</td>
<td>61½</td>
</tr>
<tr>
<td>Q = 20</td>
<td>72</td>
<td>20</td>
<td>85</td>
<td>88</td>
<td>66½</td>
</tr>
<tr>
<td>Worst FCFS</td>
<td>68</td>
<td>145</td>
<td>0</td>
<td>121</td>
<td>83½</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Wait Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Best FCFS</td>
</tr>
<tr>
<td>Q = 1</td>
</tr>
<tr>
<td>Q = 5</td>
</tr>
<tr>
<td>Q = 8</td>
</tr>
<tr>
<td>Q = 10</td>
</tr>
<tr>
<td>Q = 20</td>
</tr>
<tr>
<td>Worst FCFS</td>
</tr>
</tbody>
</table>

Lottery Scheduling

- Yet another alternative: Lottery Scheduling
  - Give each job some number of lottery tickets
  - On each time slice, randomly pick a winning ticket
  - On average, CPU time is proportional to number of tickets given to each job

  - How to assign tickets?
    - To approximate SRTF, short running jobs get more, long running jobs get fewer
    - 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**

- 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 job gets</th>
<th>% of CPU each long job 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 users out

**Strict Priority Scheduling**

- Execution Plan
  - Always execute highest-priority runnable jobs to completion
- Problems:
  - starvation:
    > Lower priority jobs don’t get to run because higher priority tasks always running
  - 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...

**Administrivia**

- Midterm I: Wednesday 3/13 – Next Wednesday
  - 4:00PM-7:00PM, in this Room
  - No class on day of exam (We will have extra OH)
- Topics:
  - All topics up to that Monday (3/11) are fair game
  - Closed book, 1 Sheet for notes (both sides, handwritten)
- Reading for this lecture and Lab 2
  - Probably want to read Love Chapter 4
  - Also, CBS scheduling algorithm in: “Integrating Multimedia Applications in Hard Real-Time Systems”
Recall: Assumption: CPU Bursts

- Execution model: programs alternate between bursts of CPU and I/O
  - Program typically uses the CPU for some period of time, then does I/O, then uses CPU again
  - Each scheduling decision is about which job to give to the CPU for use by its next CPU burst
  - With timeslicing, thread may be forced to give up CPU before finishing current CPU burst

How to handle simultaneous mix of different types of applications?

- Can we use Burst Time (observed) to decide which application gets CPU time?
- 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?
- 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 the 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 either to a whole program or the current CPU burst of each program
  - 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

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 and RR
  - 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 (and RR): 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 FIFO:
  - Once A or B get in, keep CPU for two weeks
- What about RR or SRTF?
  - Easier to see with a timeline

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: Even non-malicious users have trouble predicting runtime of their jobs
- 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:
    - Let \( t_{n-1}, t_{n-2}, t_{n-3}, \ldots \) be previous CPU burst lengths.
    - Estimate next burst \( \tau_n = f(t_{n-1}, t_{n-2}, t_{n-3}, \ldots) \)
    - Function \( f \) could be one of many different time series estimation schemes (Kalman filters, etc)
  - For instance, exponential averaging
    - \( \tau_n = \alpha t_{n-1} + (1-\alpha)\tau_{n-1} \) with \( 0 < \alpha \leq 1 \)
Multi-Level Feedback Scheduling

• Another method for exploiting past behavior
  - First used 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)

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
• Countermeasure: user action that can foil intent of the OS designer
  - 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!

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
    » In Multics, shut down machine, found 10-year-old job
  - 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!
• 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

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 ⇒ higher priority
  - 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:
  - The “active queue” and the “expired queue”
    » All tasks in the active queue use up their timeslices and are removed from the expired queue, after which queues swapped
    » 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
O(1) Scheduler Continued

- Heuristics
  - User-task priority adjusted ±5 based on heuristics
    » p->sleep_avg = sleep_time - run_time
    » Higher 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

- 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

Characteristics of a RTS

- Extreme reliability and safety
  - Embedded systems typically control the environment in which they operate
  - Failure to control can result in loss of life, damage to environment or economic loss

- Guaranteed response times
  - We need to be able to predict with confidence the worst case response times for systems
  - Efficiency is important but predictability is essential
    » In RTS, performance guarantees are:
      - Task- and/or class centric
      - Often ensured a priori
    » In conventional systems, performance is:
      - System oriented and often throughput oriented
      - Post-processing (… wait and see …)

- Soft Real-Time
  - Attempt to meet deadlines with high probability
  - Important for multimedia applications

Summary

- Scheduling: selecting a waiting process from the ready queue and allocating the CPU to it
- FCFS Scheduling:
  - Run threads to completion in order of submission
  - Pros: Simple
  - Cons: Short jobs get stuck behind long ones
- 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
  - Cons: Poor when jobs are same length
- Lottery Scheduling:
  - Give each thread a priority-dependent number of tokens (short tasks ⇒ more tokens)
  - Reserve a minimum number of tokens for every thread to ensure forward progress/fairness

Summary (Con’t)

- 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
  - Automatic promotion/demotion of process priority in order to approximate SJF/SRTF
- Linux O(1) Scheduler: Priority Scheduling with dynamic Priority boost/retraction
  - All operations O(1)
  - Fairly complex heuristics to perform dynamic priority alterations
  - Every task gets at least a little chance to run
- Realtime Schedulers: RMS, EDF, CBS
  - All attempting to provide guaranteed behavior