Recall: Scheduling

- Question: How is the OS to decide which of several tasks to take off a queue?
- Scheduling: deciding which threads are given access to resources from moment to moment
  - Often, we think in terms of CPU time, but could also think about access to resources like network BW or disk access

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: FCFS Scheduling (Cont.)

- Example continued:
  - Suppose that processes arrive in order: P2, P3, P1
  - Now, the Gantt chart for the schedule is:

<p>| | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>P2</td>
<td>P3</td>
<td>P1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>3</td>
<td>6</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>6</td>
<td>30</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- Waiting time for P1 = 6; P2 = 0; P3 = 3
- Average waiting time: \( \frac{6 + 0 + 3}{3} = 3 \)
- Average Completion time: \( \frac{3 + 6 + 30}{3} = 13 \)

- In second case:
  - Average waiting time is much better (before it was 17)
  - Average completion time is better (before it was 27)

- FIFO Pros and Cons:
  - Simple (+)
  - Short jobs get stuck behind long ones (-)
    » Safeway: Getting milk, always stuck behind cart full of items!
    » Upside: get to read about Space Aliens!
Round Robin (RR) Scheduling

• FCFS Scheme: Potentially bad for short jobs!
  – Depends on submit order
  – If you are first in line at supermarket with milk, you don’t care who is behind you, on the other hand…
• Round Robin Scheme: Preemption!
  – Each process gets a small unit of CPU time (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

Round Robin (RR) Scheduling (Cont.)

• Performance
  – \( q \) large \( \Rightarrow \) FCFS
  – \( q \) small \( \Rightarrow \) Interleaved (really small \( \Rightarrow \) hyperthreading?)
  – \( q \) must be large with respect to context switch, otherwise overhead is too high (all overhead)

Example of RR with Time Quantum = 20

• Example:
  - Process | Burst Time |
  - \( P_1 \)  | 53         |
  - \( P_2 \)  | 8          |
  - \( P_3 \)  | 68         |
  - \( P_4 \)  | 24         |

  - The Gantt chart is:
    \[ \begin{array}{ccccccccc}
    P_1 & P_2 & P_3 & P_4 & P_1 & P_4 & P_3 & P_3 \\
    0 & 20 & 28 & 48 & 68 & 88 & 108 & 125 & 145 & 153
    \end{array} \]

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

  - 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 (-)

How to Implement RR in the Kernel?

• FIFO Queue, as in FCFS
• But preempt job after quantum expires, and send it to the back of the queue
  – How? Timer interrupt!
  – And, of course, careful synchronization

Project 2: Scheduling
Round-Robin Discussion

- 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!
- Actual choices of timeslice:
  - Initially, UNIX timeslice one second:
    » Worked ok when UNIX was used by one or two people.
    » What if three compilations going on? 3 seconds to echo each keystroke!
  - Need to balance short-job performance and long-job throughput:
    » Typical time slice today is between 10ms – 100ms
    » Typical context-switching overhead is 0.1ms – 1ms
    » Roughly 1% overhead due to context-switching

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 |
| ...   | ...  | ... |
| 9     | 900  | 999 |
| 10    | 1000 | 1000 |

- Both RR and FCFS finish at the same time
- Average completion 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>Best FCFS:</th>
<th>P_2</th>
<th>P_4</th>
<th>P_1</th>
<th>P_3</th>
<th>Average</th>
</tr>
</thead>
<tbody>
<tr>
<td>Quantum</td>
<td>P_2</td>
<td>P_1</td>
<td>P_3</td>
<td>P_4</td>
<td>Average</td>
</tr>
<tr>
<td>Wait Time</td>
<td>P_2</td>
<td>P_3</td>
<td>P_1</td>
<td>P_4</td>
<td>Average</td>
</tr>
<tr>
<td>Best FCFS</td>
<td>0</td>
<td>85</td>
<td>8</td>
<td>32</td>
<td>31%</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>61%</td>
</tr>
<tr>
<td>Q = 8</td>
<td>80</td>
<td>8</td>
<td>85</td>
<td>56</td>
<td>57%</td>
</tr>
<tr>
<td>Q = 10</td>
<td>82</td>
<td>10</td>
<td>85</td>
<td>68</td>
<td>61%</td>
</tr>
<tr>
<td>Q = 20</td>
<td>72</td>
<td>20</td>
<td>85</td>
<td>88</td>
<td>66%</td>
</tr>
<tr>
<td>Best FCFS</td>
<td>68</td>
<td>145</td>
<td>0</td>
<td>121</td>
<td>83%</td>
</tr>
<tr>
<td>Completion Time</td>
<td>P_2</td>
<td>P_3</td>
<td>P_1</td>
<td>P_4</td>
<td>Average</td>
</tr>
<tr>
<td>Best FCFS</td>
<td>85</td>
<td>8</td>
<td>153</td>
<td>32</td>
<td>69%</td>
</tr>
<tr>
<td>Q = 1</td>
<td>137</td>
<td>30</td>
<td>153</td>
<td>81</td>
<td>100%</td>
</tr>
<tr>
<td>Q = 5</td>
<td>135</td>
<td>28</td>
<td>153</td>
<td>82</td>
<td>99%</td>
</tr>
<tr>
<td>Q = 8</td>
<td>133</td>
<td>16</td>
<td>153</td>
<td>80</td>
<td>95%</td>
</tr>
<tr>
<td>Q = 10</td>
<td>135</td>
<td>18</td>
<td>153</td>
<td>92</td>
<td>99%</td>
</tr>
<tr>
<td>Q = 20</td>
<td>125</td>
<td>28</td>
<td>153</td>
<td>112</td>
<td>104%</td>
</tr>
<tr>
<td>Worst FCFS</td>
<td>121</td>
<td>153</td>
<td>68</td>
<td>145</td>
<td>121%</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
    » Happens when low priority task has lock needed by high-priority task
    » Usually involves third, intermediate priority task preventing high-priority task from 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

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

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%
- RR 100ms time slice
- Disk Utilization: ~90% but lots of wakeups!
- RR 1ms time slice
- Disk Utilization: 90%
- SRTF

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 (-)

Administrivia

- Midterm I:
  - Grading done today by EOD. Sorry for the delay!
  - Solutions up off the Resources page
- Project 1 final report is due Tuesday March 1st
- Also due Tuesday March 1st: Peer evaluations
  - These are a required mechanism for evaluating group dynamics
  - Project scores are a zero-sum game
    - In the normal/best case, all partners get the same grade
    - In groups with issues, we may take points from non-participating group members and give them to participating group members!
- How does this work?
  - You get 20 points/partner to distribute as you want:
    - Example—4 person group, you get 3 x 20 = 60 points
      - If all your partners contributed equally, give the 20 points each
      - Or, you could do something like:
        - 22 points partner 1
        - 22 points partner 2
        - 16 points partner 3
  - DO NOT GIVE YOURSELF POINTS!
    - You are NOT an unbiased evaluator of your group behavior
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 \( t_n = f(t_{n-1}, t_{n-2}, t_{n-3}, \ldots) \)
  – Function \( f \) could be one of many different time series estimation schemes
    (Kalman filters, etc)
  – For instance:
    \[
    t_n = \alpha t_{n-1} + (1-\alpha)t_{n-1}
    \]
    with \( 0 < \alpha \leq 1 \)

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

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

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>N/A</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 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)

Multi-Level Feedback Scheduling

- Another method for exploiting past behavior (first use in CTSS)
  - Multiple queues, each with different priority
  - 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 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 print's, ran much faster!
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 (for realtime values)
  - Highest priority value ⇒ Lower priority (for nice 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

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
  » 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

So, Does the OS Schedule Processes or Threads?

- Many textbooks use the "old model"—one thread per process
- Usually it's really: threads (e.g., in Linux)
- One point to notice: switching threads vs. switching processes incurs different costs:
  » Switch threads: Save/restore registers
  » Switch processes: Change active address space too!
    » Expensive
    » Disrupts caching
- Recall, However: Simultaneous Multithreading (or "Hyperthreading")
  » Different threads interleaved on a cycle-by-cycle basis and can be in different processes (have different address spaces)

Multi-Core Scheduling

- Algorithmically, not a huge difference from single-core scheduling
- Implementation-wise, helpful to have per-core scheduling data structures
  » Cache coherence
- Affinity scheduling: once a thread is scheduled on a CPU, OS tries to reschedule it on the same CPU
  » Cache reuse
Recall: Spinlocks for multiprocessing

- Spinlock implementation:
  ```c
  int value = 0; // Free
  Acquire() { // spin while busy
    while (test&set(&value));
    Release() {
      value = 0; // atomic store
    }
  }
  ```
  - Spinlock doesn't put the calling thread to sleep—it just busy waits
  - When might this be preferable?
    - Waiting for limited number of threads at a barrier in a multiprocessing (multicore) program
    - Wait time at barrier would be greatly increased if threads must be woken inside kernel
  - Every `test&set()` is a write, which makes value ping-pong around between core-local caches (using lots of memory!)
    - So—really want to use `test&test&set()`!
  - As we discussed in Lecture 8, the extra read eliminates the ping-ponging issues:
    ```c
    Implementation of test&test&set():
    Acquire() {
      do {
        while(value); // wait until might be free
      } while (test&set(&value)); // exit if acquire lock
    }
    ```

Gang Scheduling and Parallel Applications

- When multiple threads work together on a multi-core system, try to schedule them together
  - Makes spin-waiting more efficient (inefficient to spin-wait for a thread that's suspended)

- Alternative: OS informs a parallel program how many processors its threads are scheduled on (Scheduler Activations)
  - Application adapts to number of cores that it has scheduled
  - “Space sharing” with other parallel programs can be more efficient, because parallel speedup is often sublinear with the number of cores

Conclusion

- 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

- Multi-Level Feedback Scheduling:
  - Multiple queues of different priorities and scheduling algorithms
  - Automatic promotion/demotion of process priority in order to approximate SJF/SRTF