CS162
Operating Systems and Systems Programming
Lecture 10

Scheduling 1: Concepts and Classic Policies
Goal for Today

- Discussion of Scheduling:
  - Which thread should run on the CPU next?

- Scheduling goals, policies

- Look at a number of different schedulers
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
Scheduling: All About Queues
Scheduling Assumptions

• Many implicit assumptions for CPU scheduling:
  – One program per user
  – One thread per program
  – Programs are independent

• Clearly, these are unrealistic but they simplify the problem so it can be solved
  – For instance: is “fair” about fairness among users or programs?
    » If I run one compilation job and you run five, you get five times as much CPU on many operating systems

• The high-level goal: Dole out CPU time to optimize some desired parameters of system
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 in next CPU burst
  - With timeslicing, thread may be forced to give up CPU before finishing current CPU burst
Scheduling Policy Goals/Criteria

• Minimize Response Time
  – Minimize elapsed time to do an operation (or job)

• Maximize Throughput
  – Maximize operations (or jobs) per second

• Fairness
  – Share CPU among users in some equitable way
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
Scheduling Policy Goals/Criteria

• 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)
Scheduling Policy Goals/Criteria

• 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
Useful metrics

– Waiting time for $P$: time before $P$ got scheduled

– Average waiting time: Average of all processes’ wait time.

– Completion time (response time): Waiting time + Run time.

– Average completion time (response time): Average of all processes' completion time
First-Come, First-Served (FCFS) Scheduling

• First-Come, First-Served (FCFS)
  – Also “First In, First Out” (FIFO) or “Run until done”
    » In early systems, FCFS meant one program scheduled until done (including I/O)
    » Now, means keep CPU until thread blocks

• Example:

<table>
<thead>
<tr>
<th>Process</th>
<th>Burst Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>(P_1)</td>
<td>24</td>
</tr>
<tr>
<td>(P_2)</td>
<td>3</td>
</tr>
<tr>
<td>(P_3)</td>
<td>3</td>
</tr>
</tbody>
</table>

– Suppose processes arrive in the order: \(P_1, P_2, P_3\)
The Gantt Chart for the schedule is:

\[
\begin{array}{ccc}
0 & 24 & 27 & 30 \\
P_1 & P_2 & P_3 & \\
\end{array}
\]

– Waiting time for \(P_1 = 0\); \(P_2 = 24\); \(P_3 = 27\)
– Average waiting time: \(\frac{0 + 24 + 27}{3} = 17\)
– Average Completion time: \(\frac{24 + 27 + 30}{3} = 27\)

• Convoy effect: short process stuck behind long process
With FCFS non-preemptive scheduling, convoys of small tasks tend to build up when a large one is running.
FCFS Scheduling (Cont.)

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

  ![Gantt chart](image)

  - 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 (-)
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 \( q \) ⇒
    » 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
The magic number

• What should \( q \) be?

  – \( q \) large \( \Rightarrow \) FCFS

  – \( q \) small \( \Rightarrow \) Interleaved

  – \( q \) must be large with respect to context switch, otherwise overhead is too high (all overhead)
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>0</th>
<th>20</th>
<th>28</th>
<th>48</th>
<th>68</th>
<th>88</th>
<th>108</th>
<th>112</th>
<th>125</th>
<th>145</th>
<th>153</th>
</tr>
</thead>
<tbody>
<tr>
<td>$P_1$</td>
<td>$P_2$</td>
<td>$P_3$</td>
<td>$P_4$</td>
<td>$P_1$</td>
<td>$P_3$</td>
<td>$P_4$</td>
<td>$P_1$</td>
<td>$P_3$</td>
<td>$P_3$</td>
<td></td>
</tr>
</tbody>
</table>
```

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

- Average waiting time = (72+20+85+88)/4 = 66¼
- Average completion time = (125+28+153+112)/4 = 104½
Decrease Response Time

- $T_1$: Burst Length 10
- $T_2$: Burst Length 1

- $Q = 10$
  - Average Response Time = $(10 + 11)/2 = 10.5$

- $Q = 5$
  - Average Response Time = $(6 + 11)/2 = 8.5$
Same Response Time

- $T_1$: Burst Length 1
- $T_2$: Burst Length 1

- $Q = 10$

  \[
  \begin{array}{c|c|c}
  T_1 & T_2 \\
  \hline
  0 & 1 & 2 \\
  \end{array}
  \]

  - Average Response Time $= (1 + 2)/2 = 1.5$

- $Q = 1$

  \[
  \begin{array}{c|c|c}
  T_1 & T_2 \\
  \hline
  0 & 1 & 2 \\
  \end{array}
  \]

  - Average Response Time $= (1 + 2)/2 = 1.5$
Increase Response Time

• $T_1$: Burst Length 1
• $T_2$: Burst Length 1

• $Q = 1$

\[ \begin{array}{c}
0 \\
1 \\
2 \\
\end{array} \]

– Average Response Time = $(1 + 2)/2 = 1.5$

• $Q = 0.5$

\[ \begin{array}{c}
0 \\
2 \\
\end{array} \]

– Average Response Time = $(1.5 + 2)/2 = 1.75$
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 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:

<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>(P_1)</th>
<th>(P_2)</th>
<th>(P_3)</th>
<th>(P_4)</th>
<th>Average</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Best FCFS</strong></td>
<td>32</td>
<td>0</td>
<td>85</td>
<td>8</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><strong>Worst FCFS</strong></td>
<td>68</td>
<td>145</td>
<td>0</td>
<td>121</td>
<td>83%</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Quantum</th>
<th>(P_1)</th>
<th>(P_2)</th>
<th>(P_3)</th>
<th>(P_4)</th>
<th>Average</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Best FCFS</strong></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><strong>Worst FCFS</strong></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 \( \Rightarrow \)
      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%

Disk Utilization: ~90% but lots of wakeups!

Disk Utilization: 90%
• 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:
    - Let $t_{n-1}$, $t_{n-2}$, $t_{n-3}$, etc. 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 $0 < \alpha \leq 1$
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 (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
Multi-Level Feedback Scheduling

- 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. Otherwise, push up one level
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
Scheduling Details

• 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!
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
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?

• For instance, is Burst Time (observed) useful to decide which application gets CPU time?
  – Short Bursts $\Rightarrow$ Interactivity $\Rightarrow$ 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
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
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
Conclusion

• 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

• Lottery Scheduling:
  – Give each thread a priority-dependent number of tokens (short tasks⇒more tokens)