Today: Scheduling

- Basis for multiprogrammed/multithreaded OSes

- Scheduling:
  - Given a runqueue of threads that are runnable/ready
  - Select one of those threads to execute next
  - $N$-processor systems require the scheduler to choose up to $N$ threads to execute
  - Aside: processes vs. threads – use interchangeably

- Main question: *how do we choose the next thd to run?*
When is CPU Scheduling done?

- CPU scheduling occurs when
  1) A process is created and is put in a ready state
  2) A process voluntarily *yields* the CPU
  3) A process switches from running to waiting state
     - e.g. blocking on I/O
  4) A process switches from running to ready state
     - e.g. because it is interrupted and its timeslice expires
  5) A process switches from waiting to ready state
     - e.g. an interrupt signifies that I/O is complete
  6) A process terminates

- **Non-preemptive scheduling** includes all but 4) and 5)
- **Preemptive scheduling** includes all of the above
  - Characterized by interrupts that result in some process being moved from *running* to *ready* state – being *preempted*
Dispatching

- Scheduler decides *which* thread to run next

- Dispatcher switches to that thread
  - Switch register contents
  - Change virtual address spaces if next process is different than last
  - Resume execution in user-space

- These overheads define the *dispatch latency*
  - overhead!
Scheduling Goals/Criteria

- What should a scheduler try to
  - Maximize?
  - Minimize?
Scheduling Goals/Criteria

- **CPU Utilization**: % of time CPU is running thd
- **Turnaround Time**: life-time of a thread
- **Waiting Time**: time thd. spends in runqueue
- **Response Time/Interactivity**: time from beginning of execution, to when process can output or input
- **Fairness**: Are threads treated comparably?
  - **Starvation**: bounded turnaround time for all thds?

- **Tradeoffs**: Maximize which? Minimize which?
Scheduling Policies

- Goals of scheduler dictate
  - Algorithm/policy used to select next thread
  - Data structures used by algorithm
    - e.g. ready-queue data structure
CPU Burst Histogram
First-Come, First-Serve Scheduling

- One of the simplest scheduling policies
  - non-preemptive

<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 process arrive in order: $P_1$, $P_2$, $P_3$
- The schedule is:

```
0   24   27   30
P_1 P_2 P_3

Waiting time for $P_1 = 0$; $P_2 = 24$; $P_3 = 27$
Average waiting time: $(0 + 24 + 27)/3 = 17$
Suppose the processes arrive in order: $P_2, P_3, P_1$

- The schedule is:

```
  P_2 | P_3 | P_1
  0   | 3   | 6   | 30
```

- Waiting time for $P_1 = 6$; $P_2 = 0$; $P_3 = 3$
- Average waiting time:  \( \frac{6 + 0 + 3}{3} = 3 \)
- Much better than previous case
- *Convoy effect* short process behind long process

- Fairness/Starvation?
- Interactivity/Responsiveness?
Round Robin Scheduling

- Timesharing systems that wish to provide “fair” distribution of CPU resources
  - One thread cannot monopolize CPU
- FCFS with preemption
- Each thread executes a single timeslice (or quantum) before it is preempted
  - Preempted threads placed at end of runqueue
  - Requires timer interrupt to measure timeslices and preempt
Round Robin Scheduling II

- \( N \) threads in runqueue
- Time quantum of \( Q \)
- *Fairness*: each thread gets \( 1/n \) of the CPU, in chunks of size \( Q \)
- No thread waits for more than \((N-1)Q\) time units before next quantum
- Size of \( Q \)?
  - \( Q == \) infinity is FCFS
  - \( Q == 0 \); is this possible?
  - Best fairness?
  - Best throughput? (what overheads are there?)
**RR Example, Q = 3**

<table>
<thead>
<tr>
<th>Process</th>
<th>Burst Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>$P_1$</td>
<td>6</td>
</tr>
<tr>
<td>$P_2$</td>
<td>3</td>
</tr>
<tr>
<td>$P_3$</td>
<td>1</td>
</tr>
<tr>
<td>$P_4$</td>
<td>7</td>
</tr>
</tbody>
</table>

- Compared to FIFO
  - Turnaround time?
  - responsiveness?
Quantum Effects Turnaround Time

<table>
<thead>
<tr>
<th>process</th>
<th>time</th>
</tr>
</thead>
<tbody>
<tr>
<td>$P_1$</td>
<td>6</td>
</tr>
<tr>
<td>$P_2$</td>
<td>3</td>
</tr>
<tr>
<td>$P_3$</td>
<td>1</td>
</tr>
<tr>
<td>$P_4$</td>
<td>7</td>
</tr>
</tbody>
</table>
Shortest Job First (SJF) Scheduling

- Consider each process' next CPU burst (job) length
  - use these to schedule the process with the shortest next burst
- Preemptive SJF is optimal in that it minimizes average waiting time for a set of processes
Shortest Job First II

<table>
<thead>
<tr>
<th>Process</th>
<th>Arrival Time</th>
<th>Burst Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>P₁</td>
<td>0</td>
<td>7</td>
</tr>
<tr>
<td>P₂</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>P₃</td>
<td>4</td>
<td>1</td>
</tr>
<tr>
<td>P₄</td>
<td>5</td>
<td>4</td>
</tr>
</tbody>
</table>

- SJF (preemptive)

Average waiting time = \( (9+1+0+2)/4 = 3 \)
- Non-preemptive optimal?
Job Burst Length

• How do we know a job's burst time?
  • Before it actually executes!

• Become fortune tellers?

• General strategy in systems: Predict the future from past behavior
  • Is this a good idea? Does it really work?
Determine Job Burst Length

- Take average of process' past burst lengths
  - Do we want to keep an exact average?

- Weighted Moving Average:
  - Measured length of nth burst = $t_n$
  - Predicted value for burst n = $\tau_n$
  - Then for a weight, $\alpha$, where $0 \leq \alpha \leq 1$:
    \[ \tau_{n+1} = \alpha t_n + (1-\alpha)\tau_n \]
Job Length Prediction Using WMA
Weighted Moving Average III

\[ \tau_{n+1} = \alpha t_n + (1-\alpha)\tau_n \]

- If \( \alpha = 0 \), then \( \tau_{n+1} = \tau_0 \)
  - Recent job burst lengths aren't counted
- If \( \alpha = 1 \), then \( \tau_{n+1} = t_n \)
  - Only the most recent job length counts
- Expand the formula: 
  \[ \tau_{n+1} = \alpha t_n + (1-\alpha) \alpha t_{n-1} + \ldots \]
  \[ + (1-\alpha)^j \alpha t_{n-j} + \ldots \]
  \[ + (1-\alpha)^{n+1} \tau_0 \]
  - Exponentially decrease the influence of older measurements
Priority Scheduling

- *priority* associated with each thread
  - Scheduler selects thread with highest priority
  - Both preemptive and non-preemptive variants

- Problem $\rightarrow$ starvation
  - Low priority processes may never execute

- One solution $\rightarrow$ aging
  - As a thread uses more execution time, dynamically decrease its priority
Multilevel Queue Scheduling

- Ready-queue partitioned into separate queues
- Each queue has its own scheduling policy
  - I/O-bound/interactive task queue – RR
  - CPU-bound/background/batch queue – FCFS
- Scheduling done between queues
  - Fixed priority – some queues have higher priority
    - Possible starvation
  - Proportional allocation – background gets 20% CPU
Multilevel Queue Scheduling II

highest priority

- system processes

- interactive processes

- interactive editing processes

- batch processes

- student processes

lowest priority
Multilevel Feedback Queuing

- How make thread ↔ queue mapping?
- Want interactive/I/O bound threads in higher priority queues

- Threads can move between different queues
  - Aging to avoid starvation
- Multilevel feedback queuing parameters:
  - # of queues
  - Scheduling algorithm for each queue
  - Policy to *promote* a thread to higher queues
  - Policy to *demote* a thread to lower queues
  - Entry queue for new threads
Multilevel Feedback Example

- Three queues – in order of decreasing priority
  - $Q_0$: RR with a timeslice of 8 time units
  - $Q_1$: RR with a timeslice of 16 time units
  - $Q_2$: FCFS
- New Jobs arrive in $Q_0$, until they expend 8 time units, demote to $Q_1$...
- Thread *promoted* when placed in “runqueue” after waiting on I/O
- Starvation???
Multilevel Feedback Example II

quantum = 8

quantum = 16

FCFS