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: 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:
  - Process | Burst Time
  - $P_1$ | 24
  - $P_2$ | 3
  - $P_3$ | 3

  Suppose processes arrive in the order: $P_1, P_2, P_3$
  The Gantt Chart for the schedule is:

  - Waiting time for $P_1 = 0$; $P_2 = 24$; $P_3 = 27$
  - Average waiting time: $(0 + 24 + 27)/3 = 17$
  - Average Completion time: $(24 + 27 + 30)/3 = 27$

  Convo effect: short process behind long process

  - Example continued:
    - Suppose that processes arrive in order: $P_2, P_3, P_1$

  Now, the Gantt chart for the schedule is:

  - Waiting time for $P_1 = 6$; $P_2 = 0$; $P_3 = 3$
  - Average waiting time: $(6 + 0 + 3)/3 = 3$
  - Average Completion time: $(3 + 6 + 30)/3 = 13$

  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 small items. Upside: get to read about space aliens!
Round Robin (RR)

- 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
  - 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 $\frac{1}{n}$ 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 ⇒ FCFS
  - $q$ small ⇒ Interleaved (really small ⇒ hyperthreading?)
  - $q$ must be large with respect to context switch, otherwise overhead is too high (all overhead)

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!
**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

---

**Administrivia**

- **Midterm #1**
  - Not the result we expected! Exam was too long for everyone
  - Class is scaled—think of the grades as being ~20 points higher
  - More important is your grade relative to mean (more than 2 std devs below is cause for concern – contact staff)
  - We graded very leniently: regrades open until Mon 10/10 11:59PM

- **Upcoming deadlines:**
  - HW 2 due today Mon 10/3
  - Project 1 final code due Wed 10/5, final report due Fri 10/7
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

• 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>50%</td>
<td>N/A</td>
</tr>
<tr>
<td>1/10</td>
<td>9.9%</td>
<td>0.99%</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
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 Diff Types of Apps?

- 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” (SRTC)
- 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 Example continued:

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:
    - \( t_{n-1}, t_{n-2}, t_{n-3}, \ldots \) 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, exponential averaging
    - \( t_n = \alpha t_{n-1} + (1-\alpha) t_{n-1} \) with (0<\(\alpha\)≤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!

Case Study: Linux O(1) Scheduler

- Priority-based scheduler: 140 priorities
  – 40 for “user tasks” (set by “nice”), 100 for “Real-time/Kernel”
  – Lower priority value ⇒ higher priority (for nice values)
  – Highest priority value ⇒ Lower priority (for real-time values)
  – All algorithms O(1) – schedule n processes in constant time
    » Compute timeslices/priorities/interactivity credits when job finishes time slice
    » 140-bit bit mask indicates presence or absence of job(s) at given priority level
- Two separate priority queues (arrays): “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 multi-level queue (1 queue per priority) with diff timeslice at each level
  – Execution split into “Timeslice Granularity” chunks – RR through priority
**O(1) Scheduler Continued**

- **Heuristics**
  - User-task priority adjusted ±5 based on heuristics
    - \( p \rightarrow sleep_{\text{avg}} = sleep_{\text{time}} - run_{\text{time}} \)
    - Higher \( sleep_{\text{avg}} \) => more I/O bound the task, more reward (and vice versa)
  - Interactive Credit
    - Earned when task sleeps for “long” time, Spent when task runs for “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 another task has starved for too long…

- **Real-Time Tasks**
  - Always preempt non-RT tasks and 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 Queuing”
  - 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 Examples**

- Suppose Targeted latency = 20ms and Minimum Granularity = 1ms

- Two CPU bound tasks with same priorities
  - Both switch with 10ms

- 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 – miss target latency)

- One CPU bound task, one interactive task same priority
  - While interact task sleeps, CPU bound task runs, increments \( vruntime \)
  - When interact task wakes up, runs immediately (it’s behind on \( vruntime \))

- Group scheduling facilities (2.6.24)
  - Can give fair fractions to groups (user or other process group)
  - So, two users, one starts 1 process, other starts 40, each gets 50% 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)

Summary (1 of 2)

- 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

- Real-time Schedulers such as EDF
  - Guaranteed behavior by meeting deadlines
  - Real-time tasks defined by tuple of compute time and period
  - Schedulability test: is it possible to meet deadlines with proposed set of processes?

EDF: Schedulability Test

Theorem (Utilization-based Schedulability Test):
A task set $T_1, T_2, \ldots, T_n$ with $D_i = P_i$ is schedulable by the earliest deadline first (EDF) scheduling algorithm if

$$\sum_{i=1}^{n} \left( \frac{C_i}{D_i} \right) \leq 1$$

Exact schedulability test (necessary + sufficient)
Proof: [Liu and Layland, 1973]