CIS 415: Operating Systems

Scheduling

Spring 2013
Prof. Kevin Butler
• Last class:
  ‣ Threads

• Today:
  ‣ Intro to Scheduling

• Remember: Project 1 due tonight!
In a multiprogramming system, we need to share resources among the running processes.

What are the types of OS resources?

Question: Which process gets access to which resources?

To maximize performance.
Resource Types

• Memory: Allocate portion of finite resource
  ‣ Virtual memory tries to make this appear infinite
  ‣ Physical resources are limited

• I/O: Allocate portion of finite resource and time with resource
  ‣ Store information on disk
  ‣ A time slot to store that information

• CPU: Allocate time slot with resource
  ‣ A time slot to run instructions

• We will focus on **CPU scheduling** for now
Types of Scheduling

• **Long-term (admission) scheduling**: determining whether to add to the pool of processes to be executed

• **Medium-term scheduling**: determining whether to add to the number of processes partially or fully in memory

• **Short-term scheduling**: determining which process will be executed by the processor

• **I/O scheduling**: determining which process’s pending I/O request will be handled by an available I/O device
CPU Scheduling Examples

• Single process view
  ‣ GUI request
    • Click on the mouse
  ‣ Scientific computation
    • Long-running, but want to complete ASAP

• System view
  ‣ Get as many tasks done as quickly as possible
  ‣ Minimize waiting time for processes
  ‣ Get full utilization from the CPU
Process Scheduling

- Running to Ready: Dispatched (CPU assigned)
- Ready to Running: Pre-empted (CPU yanked)
- Blocked to Ready: Event Occurred
- Ready to Blocked: Wait For Event (e.g. I/O)
- Running to Process Terminates
- Ready to New process creation
Scheduling Problem

• Choose the *ready/running* process to run at any time
  ‣ Maximize “performance”

• Model/estimate “performance” as a function
  ‣ System performance of scheduling each process
    • \( f(\text{process}) = y \)
  ‣ What are some choices for \( f(\text{process}) \)?

• Choose the process with the best \( y \)
  ‣ Estimating overall performance is intractable
    • E.g., scheduling so all tasks are completed as soon as possible is NP-complete, then add in pre-emption...
When Scheduling Occurs

• CPU scheduling decisions may take place when a process:
  ‣ 1. Switches from running to waiting state
  ‣ 2. Switches from running to ready state
  ‣ 3. Switches from waiting to ready
  ‣ 4. Terminates

• Scheduling for events 1 and 4 do not preempt a process
  ‣ Process volunteers to give up the CPU
Preemption

- Can we reschedule a process that is actively running?
  - If so, we have a *preemptive* scheduler
  - If not, we have a *non-preemptive* scheduler

- Suppose a process becomes ready
  - E.g., new process is created or it is no longer waiting

- It may be better to schedule this process
  - So, we preempt the running process

- In what ways could the new process be better?
Bursts

- A process runs in CPU and I/O Bursts
  - Run instructions (CPU Burst)
  - Wait for I/O (I/O Burst)
- Scheduling is aided by knowing the length of these bursts
  - More later…
CPU Burst Duration

![Graph showing the frequency of CPU burst duration vs. duration in milliseconds. The graph has a peak at around 8 milliseconds with a frequency of 160, and the frequency decreases as the duration increases, approaching a constant value of around 20.]
Dispatcher

- Dispatcher module gives control of the CPU to the process selected by the short-term scheduler; this involves:
  - Switching context
  - Switching to user mode
  - Jumping to the proper location in the user program to restart that program

- *Dispatch latency* — time it takes for the dispatcher to stop one process and start another running
Scheduling Criteria

- **Utilization/efficiency**: keep the CPU busy 100% of the time with useful work
- **Throughput**: maximize the number of jobs processed per hour.
- **Turnaround time**: from the time of submission to the time of completion.
- **Waiting time**: Sum of time spent (in Ready queue) waiting to be scheduled on the CPU.
- **Response Time**: time from submission until the first response is produced (mainly for interactive jobs)
- **Fairness**: make sure each process gets a fair share of the CPU
Scheduling Algorithms

- Some may seem intuitively better than others
- But a lot has to do with the type of offered workload to the processor
- Best scheduling comes with best context of the tasks to be completed
• First-Come, First-Served (FCFS)
  ‣ Serve the jobs in the order they arrive.
  ‣ Non-preemptive
  ‣ Simple and easy to implement: When a process is ready, add it to tail of ready queue, and serve the ready queue in FCFS order.
  ‣ Very fair: No process is starved out, and the service order is immune to job size, etc.
**FCFS**

<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 that the processes arrive in the order: $P_1$, $P_2$, $P_3$
- The *Gantt Chart* for the schedule is:

  ![Gantt Chart](image)

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

Suppose that the processes arrive in the order

\[ P_2, P_3, P_1 \]

- The Gantt chart for the schedule is:

```
       P2   P3   P1
  0 3  6  30
```

- Waiting time for \( P_1 = 6; P_2 = 0; P_3 = 3 \)
- Average waiting time: \( (6 + 0 + 3)/3 = 3 \)
- Much better than previous case
- **Convoy effect**: short process behind long process
Shortest-Job-First (SJF)

• Associate with each process the length of its next CPU burst. Use these lengths to schedule the process with the shortest time

• Two schemes:
  ‣ **Non-preemptive** – once CPU given to the process it cannot be preempted until completes its CPU burst
  ‣ **Preemptive** – if a new process arrives with CPU burst length less than remaining time of current executing process, preempt. This scheme is known as the Shortest-Remaining-Time-First (SRTF)

• SJF is **optimal** – gives minimum average waiting time for a given set of processes
  ‣ So we always use it, right?
### Non-Preemptive SJF

<table>
<thead>
<tr>
<th>Process</th>
<th>Arrival Time</th>
<th>Burst Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>$P_1$</td>
<td>0.0</td>
<td>7</td>
</tr>
<tr>
<td>$P_2$</td>
<td>2.0</td>
<td>4</td>
</tr>
<tr>
<td>$P_3$</td>
<td>4.0</td>
<td>1</td>
</tr>
<tr>
<td>$P_4$</td>
<td>5.0</td>
<td>4</td>
</tr>
</tbody>
</table>

- SJF (non-preemptive)

\[
\text{Average waiting time} = \frac{0 + 6 + 3 + 7}{4} = 4
\]
Preemptive SJF

<table>
<thead>
<tr>
<th>Process</th>
<th>Arrival Time</th>
<th>Burst Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>$P_1$</td>
<td>0.0</td>
<td>7</td>
</tr>
<tr>
<td>$P_2$</td>
<td>2.0</td>
<td>4</td>
</tr>
<tr>
<td>$P_3$</td>
<td>4.0</td>
<td>1</td>
</tr>
<tr>
<td>$P_4$</td>
<td>5.0</td>
<td>4</td>
</tr>
</tbody>
</table>

- SJF (preemptive)

- Average waiting time $= (9 + 1 + 0 + 2)/4 = 3$
Determining Next CPU Burst

• We can only estimate what the duration of the next CPU burst will be

• Use length of previous CPU bursts as a guide, and exponential averaging to predict next burst
  ‣ If $t_n$ is the actual length of the $n$th CPU burst, and
  ‣ $\tau_{n+1}$ is the predicted value of the next CPU burst, then
  ‣ Given some parameter $\alpha$, $0 \leq \alpha \leq 1$
  ‣ Define $\tau_{n+1} = \alpha t_n + (1 - \alpha) \tau_n$
Determining Next CPU Burst

• If $\alpha = 0$, no weighting to recent history (e.g., current conditions are transient)
• If $\alpha = 1$, no weighting to old history
• Typically, choose $\alpha = 1/2$ which gives equal weighting to recent and past history

1. $t_n = \text{actual length of } n^{th} \text{ CPU burst}$
2. $\tau_{n+1} = \text{predicted value for the next CPU burst}$
3. $\alpha, 0 \leq \alpha \leq 1$
4. Define: $\tau_{n+1} = \alpha t_n + (1 - \alpha) \tau_n$.
Exponential Averaging

• If we expand the formula, we get:

\[ \tau_{n+1} = \alpha t_n + (1 - \alpha)\alpha t_{n-1} + \ldots \]
\[ + (1 - \alpha)^j \alpha t_{n-j} + \ldots \]
\[ + (1 - \alpha)^{n+1} \tau_0 \]

• Since both \( \alpha \) and \( 1 - \alpha \) are less than or equal to 1, each successive term has less weight than its predecessor
CPU Burst Prediction

<table>
<thead>
<tr>
<th>CPU burst ($t_i$)</th>
<th>6</th>
<th>4</th>
<th>6</th>
<th>4</th>
<th>13</th>
<th>13</th>
<th>13</th>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;guess&quot; ($\tau_i$)</td>
<td>10</td>
<td>8</td>
<td>6</td>
<td>6</td>
<td>5</td>
<td>9</td>
<td>11</td>
<td>12</td>
</tr>
</tbody>
</table>
Scheduling Algorithms

• First-come, First-serve (FCFS)
  ‣ Non-preemptive
  ‣ Does not account for waiting time (or much else)
    • Convoy problem

• Shortest Job First
  ‣ May be preemptive
  ‣ Optimal for minimizing waiting time (how?)

• Lots more… And what do real systems use?
Priority Scheduling

• Each process is given a certain priority “value”.
• Always schedule the process with the highest priority.
<table>
<thead>
<tr>
<th></th>
<th>Duration(s)</th>
<th>Priority</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1</td>
<td>10</td>
<td>3</td>
</tr>
<tr>
<td>P2</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>P3</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>P4</td>
<td>1</td>
<td>5</td>
</tr>
<tr>
<td>P5</td>
<td>5</td>
<td>2</td>
</tr>
</tbody>
</table>

Gantt Chart for Priority Scheduling
Priorities

• Note that FCFS and SJF are specialized versions of Priority Scheduling
  ‣ i.e. there is a way of assigning priorities to the processes so that Priority Scheduling would result in FCFS/SJF.

• What would examples of those priority functions be?
Round Robin (RR)

• Each process gets a small unit of CPU time (time quantum)
  ‣ Usually 10-100 milliseconds
  ‣ After this time has elapsed, the process is preempted and added to the end of the ready queue

• Approach
  ‣ If there are $n$ processes in the ready queue and the time quantum is $q$
    ‣ Then each process gets $1/n$ of the CPU time
    ‣ In chunks of at most $q$ time units at once.
    ‣ No process waits more than $(n-1)q$ time units
## Round Robin

### Arrival Time (s) vs. Job length (s)

<table>
<thead>
<tr>
<th></th>
<th>Arrival Time (s)</th>
<th>Job length (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1</td>
<td>0</td>
<td>24</td>
</tr>
<tr>
<td>P2</td>
<td>0</td>
<td>3</td>
</tr>
<tr>
<td>P3</td>
<td>0</td>
<td>7</td>
</tr>
</tbody>
</table>

### Time Quantum = 4 s

```
0  4  7  11  15  18  22  26  30  34
P1 P2 P3 P1 P3 P1 P1 P1 P1
```
Round robin is virtually sharing the CPU between the processes giving each process the illusion that it is running in isolation (at $1/n$-th the CPU speed).

Smaller the time quantum, the more realistic the illusion (note that when time quantum is of the order of job size, it degenerates to FCFS).

But what is the drawback when time quantum gets smaller?
RR Time Quantum

• For the considered example, if time quantum size drops to 2s from 4s, the number of context switches increases to ????

• But context switches are not free!
  ‣ Saving/restoring registers
  ‣ Switching address spaces
  ‣ Indirect costs (cache pollution)
Scheduling Desirables

• SJF
  ‣ Minimize waiting time
    • Requires estimate of CPU bursts

• Round robin
  ‣ Share CPU via time quanta
    • If burst turns out to be “too long”

• Priorities
  ‣ Some processes are more important
  ‣ Priorities enable composition of “importance” factors

• No single best approach -- now what?
Round Robin with Priority

• Have a ready queue for each priority level.

• Always service the non-null queue at the highest priority level.

• Within each queue, you perform round-robin scheduling between those processes.
Round-Robin with Priority

Priority Levels

[Diagram of priority levels and related elements]
What is the problem?

• With fixed priorities, processes lower in the priority level can get *starved out!*

• In general, you employ a mechanism to “age” the priority of processes.
Multilevel Queue

• Ready queue is partitioned into separate queues: foreground (interactive) & background (batch)

• Each queue has its own scheduling algorithm, foreground – RR & background – FCFS

• Scheduling must be done between the queues.
  ‣ Fixed priority scheduling; (i.e., serve all from foreground then from background). Possibility of starvation.
  ‣ Time slice – each queue gets a certain amount of CPU time which it can schedule amongst its processes; i.e., 80% to foreground in RR
  ‣ 20% to background in FCFS
Multilevel Feedback Queue

• A process can move between the various queues; aging can be implemented this way

• Multilevel-feedback-queue scheduler defined by the following parameters:
  ‣ number of queues
  ‣ scheduling algorithms for each queue
  ‣ method used to determine when to upgrade a process
  ‣ method used to determine when to demote a process
  ‣ method used to determine which queue a process will enter when that process needs service
Multilevel Feedback Queue

• Three queues:
  ‣ $Q_0$ – RR with time quantum 8 milliseconds
  ‣ $Q_1$ – RR time quantum 16 milliseconds
  ‣ $Q_2$ – FCFS

• Scheduling
  ‣ A new job enters queue $Q_0$ which is served FCFS. When it gains CPU, job receives 8 milliseconds. If it does not finish in 8 milliseconds, job is moved to queue $Q_1$.
  ‣ At $Q_1$ job is again served FCFS and receives 16 additional milliseconds. If it still does not complete, it is preempted and moved to queue $Q_2$. 
Multilevel Feedback Queues

quantum = 8

quantum = 16

FCFS
Performance for Schedulers

• *Queueing Theory Analysis* - uses well-established mathematical models and techniques.

• *Simulation* - create a model of the system and simulate its performance using simulation software.

• *Empirical Experiments* - implement and test the algorithms in a real system.
Queuing Theory Analysis

Single-server Queue:

- Arrivals
  - $\lambda = \text{arrival rate}$
- Waiting line (queue)
  - $w = \text{items waiting}$
  - $T_w = \text{waiting time}$
- Dispatching discipline
- Server
  - $T_s = \text{service time}$
  - $\rho = \text{utilization}$
- Departures
- $q = \text{items in queuing system}$
- $T_q = \text{queuing time}$

Figure A.2 Queuing System Structure and Parameters for Single-Server Queue
Queuing Theory Analysis

• Inputs:
  ‣ *arrival rate* - from a probability distribution (usually Poisson which implies random arrivals)
  ‣ *service time* - from a probability distribution (often exponential)
  ‣ *scheduling discipline/algorithm*
Queuing Theory Analysis

• Outputs:
  ‣ Items waiting
  ‣ Waiting time
  ‣ Items queued
  ‣ Queuing time
Queuing Theory Analysis

Single-server Queue:

Little’s Formula: $n = \lambda W$  
(n = queue length)
Simulation Analysis

- Discrete-event Simulation
  - Often uses models similar to queueing analysis
  - More detailed or more realistic parameters (e.g. trace driven)
  - Simulates events step by step and gathers statistics rather than using mathematical formulas
Empirical Experiments

• Run experiments on live system

• Properties:
  ‣ Costly and time-consuming
  ‣ Sometimes not possible
  ‣ More realistic
Traditional UNIX Scheduling

- Multilevel feedback queues

- 128 priorities possible (-64 to +63)

- 1 Round Robin queue per priority

- Every scheduling event the scheduler picks the highest priority (lowest number) non-empty queue and runs jobs in round-robin
UNIX Process Scheduling

• Negative numbers reserved for processes waiting in kernel mode (just woken up by interrupt handlers) (why do they have a higher priority?)

• Time quantum = 1/10 sec (empirically found to be the longest quantum that could be used without loss of the desired response for interactive jobs such as editors)
  ‣ short time quantum means better interactive response
  ‣ long time quantum means higher overall system throughput since less context switch overhead and less processor cache flush.

• Priority dynamically adjusted to reflect
  ‣ resource requirement (e.g., blocked awaiting an event)
  ‣ resource consumption (e.g., CPU time)
Linux Scheduler

• Kernel 2.4 and earlier: essentially the same as the traditional UNIX scheduler

• Kernel 2.6: $O(1)$ scheduler
  ‣ time to select process is constant regardless of system load or the number of processors
  ‣ separate queue for each priority level
  ‣ CPU affinity (keeps processes on same CPU)

• More recently (kernel 2.6.23 and up): CFS
  ‣ Completely fair scheduler (runs $O(\log N)$)
    • uses red-black trees rather than runqueues
Linux Scheduling

- Two algorithms: time-sharing and real-time
- Time-sharing (still abstracted)
  - Two queues: *active* and *expired*
  - In active, until you use your entire time slice (*quantum*), then expired
    - Once in expired, wait for all others to finish (*fairness*)
  - Priority recalculation -- based on waiting vs. running time
    - From 0-10 milliseconds
    - Add waiting time to value, subtract running time
    - Adjust the static priority

- Real-time
  - Soft real-time
  - Posix.1b compliant – two classes
    - FCFS and RR; Highest priority process always runs first
## The Relationship Between Priorities and Time-Slice length

<table>
<thead>
<tr>
<th>numeric priority</th>
<th>relative priority</th>
<th>time quantum</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>highest</td>
<td>200 ms</td>
</tr>
<tr>
<td>99</td>
<td></td>
<td></td>
</tr>
<tr>
<td>100</td>
<td></td>
<td></td>
</tr>
<tr>
<td>140</td>
<td>lowest</td>
<td>10 ms</td>
</tr>
</tbody>
</table>
List of Tasks Indexed According to Priorities
Summary

• CPU Scheduling
  ‣ Algorithms
  ‣ Combination of algorithms
    • Multi-level Feedback Queues

• Scheduling Systems
  ‣ UNIX
  ‣ Linux
• Next time: Synchronization