CS370 Operating Systems

Colorado State University
Yashwant K Malaiya
Spring 2018 Lecture 9
Scheduling

Slides based on
- Text by Silberschatz, Galvin, Gagne
- Various sources
Questions from last time

- Prediction of next burst
  - Based on actual recent duration and predicted value (which is based on past actual values)
  - More recent data points get more weight (based on alpha). This is termed exponential averaging.
    \[
    \tau_{n+1} = \alpha t_n + (1 - \alpha)\tau_n.
    \]
    \[
    \tau_{n+1} = \alpha t_n + (1 - \alpha)\alpha t_{n-1} + \ldots + (1 - \alpha)^j\alpha t_{n-j} + \ldots + (1 - \alpha)^n \tau_0
    \]

- Shortest Job First (SJF) vs Preemptive SJF
  - SJF is not preemptive
  - Preemptive SJF (Shortest remaining time first)
  - A new process that will take shorter time will preempt a process with longer remaining time.
  - Thus processes with a shorter remaining time have a higher priority.
Questions from last time

• Is main in C the process and functions are threads?
  – The whole program runs as a process, some functions may run as separate threads concurrently/in parallel within the same process.

• Child process vs function
  – A function is not a separate process.
Scheduling Criteria

- **CPU utilization** – keep the CPU as busy as possible: **Maximize**
- **Throughput** – # of processes that complete their execution per time unit: **Maximize**
- **Turnaround time** – time to execute a process from submission to completion: **Minimize**
- **Waiting time** – amount of time a process has been waiting in the ready queue: **Minimize**
- **Response time** – time it takes from when a request was submitted until the first response is produced, not output (for time-sharing environment): **Minimize**
First-Come, First-Served (FCFS) Scheduling

<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: $P_1$, $P_2$, $P_3$ but almost the same time.
  The Gantt Chart for the schedule is:

```
+-----+-----+-----+
| P_1 | P_2 | P_3 |
+-----+-----+-----+
| 0   | 24  | 27  |
+-----+-----+-----+
```

- Waiting time for $P_1 = 0$; $P_2 = 24$; $P_3 = 27$
- Average waiting time: $(0 + 24 + 27)/3 = 17$
- Throughput: $3/30 = 0.1$ per unit
Example of SJF

<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>8</td>
</tr>
<tr>
<td>$P_3$</td>
<td>7</td>
</tr>
<tr>
<td>$P_4$</td>
<td>3</td>
</tr>
</tbody>
</table>

- All arrive at time 0.
- SJF scheduling chart

- Average waiting time for $P_1, P_2, P_3, P_4 = \frac{3 + 16 + 9 + 0}{4} = 7$
Determining Length of Next CPU Burst

• Can only estimate the length – should be similar to the recent bursts
  – Then pick process with shortest predicted next CPU burst

• Can be done by using the length of previous CPU bursts, using *exponential averaging*

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

• Commonly, $\alpha$ set to $\frac{1}{2}$
• Preemptive version called *shortest-remaining-time-first*
Prediction of the Length of the Next CPU Burst

Blue points: guess
Black points: actual
α = 0.5

Ex:
0.5x6 + 0.5x10 = 8

CPU burst ($t_i$): 6 4 6 4 13 13 13 ...
"guess" ($\tau_i$): 10 8 6 6 5 9 11 12 ...
Examples of Exponential Averaging

• $\alpha = 0$
  - $\tau_{n+1} = \tau_n$
  - Recent history does not count

• $\alpha = 1$
  - $\tau_{n+1} = \alpha t_n$
  - Only the actual last CPU burst counts

• $\tau_{n+1} = \alpha t_n + (1 - \alpha) \tau_n$.

• If we expand the formula, substituting for $\tau_n$, 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.
Now we add the concepts of varying arrival times and preemption to the analysis.

<table>
<thead>
<tr>
<th>Process</th>
<th>Arrival Time</th>
<th>Burst Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>$P_1$</td>
<td>0</td>
<td>8</td>
</tr>
<tr>
<td>$P_2$</td>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td>$P_3$</td>
<td>2</td>
<td>9</td>
</tr>
<tr>
<td>$P_4$</td>
<td>3</td>
<td>5</td>
</tr>
</tbody>
</table>

**Preemptive SJF Gantt Chart**

Average waiting time for P1,P2,P3,P4

$= \frac{((10-1)+(1-1)+(17-2)+(5-3))}{4} = \frac{26}{4} = 6.5$ msec
Priority Scheduling

- A priority number (integer) is associated with each process
- The CPU is allocated to the process with the highest priority (smallest integer = highest priority)
  - Can be Preemptive (evict low priority process)
  - Or Nonpreemptive
- SJF is priority scheduling where priority is the inverse of predicted next CPU burst time
- Problem = Starvation – low priority processes may never execute
  - Solution = Aging – as time progresses increase the priority of the process

MIT had a low priority job waiting from 1967 to 1973 on IBM 7094!
Example of Priority Scheduling

<table>
<thead>
<tr>
<th>Process</th>
<th>Burst Time</th>
<th>Priority</th>
</tr>
</thead>
<tbody>
<tr>
<td>(P_1)</td>
<td>10</td>
<td>3</td>
</tr>
<tr>
<td>(P_2)</td>
<td>1</td>
<td>1 (highest)</td>
</tr>
<tr>
<td>(P_3)</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>(P_4)</td>
<td>1</td>
<td>5</td>
</tr>
<tr>
<td>(P_5)</td>
<td>5</td>
<td>2</td>
</tr>
</tbody>
</table>

- Arrived at time 0 in order \(P_1, P_2, P_3, P_4, P_5\) (which does not matter)
- Priority scheduling Gantt Chart

- Average waiting time for \(P_1, \ldots, P_5\): \((6+0+16+18+1)/5 = 8.2\) msec
Round Robin (RR) with time quantum

• Each process gets a small unit of CPU time (time quantum $q$), usually 1-10 milliseconds. After this, the process is preempted, added to the end of the ready queue.
• 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.
• Timer interrupts every quantum to schedule next process.
• Performance
  – $q$ large $\Rightarrow$ FIFO
  – $q$ small $\Rightarrow$ $q$ must be large with respect to context switch, otherwise overhead is too high (overhead typically in 0.5% range)
Example of RR with **Time Quantum = 4**

<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>

- Arrive a time 0 in order $P_1$, $P_2$, $P_3$: The Gantt chart is:

  - Waiting times: 10-4 = 6, 4, 7, average $17/3 = 5.66$ units
  - Typically, higher average turnaround than SJF, but better *response*

- $q$ should be large compared to context switch time
- $q$ usually **10 ms to 100 ms**, context switch < 10 µsec

**Response time:** Arrival to beginning of execution
**Turnaround time:** Arrival to finish of execution
Time Quantum and Context Switch Time

- **process time = 10**

<table>
<thead>
<tr>
<th></th>
<th>quantum</th>
<th>context switches</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>12</td>
<td>0</td>
</tr>
<tr>
<td>6</td>
<td>6</td>
<td>1</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>9</td>
</tr>
</tbody>
</table>

- **Diagram**
  - Each bar represents a quantum of time.
  - The numbers indicate the number of context switches.
Turnaround Time Varies With The Time Quantum

Rule of thumb: 80% of CPU bursts should be shorter than q

Illustration
q=7:
Turnaround time:
6,9,10,17  av = 10.5
Similarly for q =1, ..6
Turnaround Time Varies With The Time Quantum

Rule of thumb: 80% of CPU bursts should be shorter than q

Illustration
q=7:
Turnaround time:
6,9,10,17 \ av = 10.5
Similarly for q =1, ..6
**Multilevel Queue**

- Ready queue is partitioned into separate queues, e.g.:
  - **foreground** (interactive)
  - **background** (batch)
- Process permanently in a given queue
- Each queue has its own scheduling algorithm, e.g.:
  - 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. Or
  - 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 Queue Scheduling

highest priority

- system processes
- interactive processes
- interactive editing processes
- batch processes
- student processes

lowest priority
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
Example of Multilevel Feedback Queue

• Three queues:
  – $Q_0$ – RR with time quantum 8 milliseconds
  – $Q_1$ – RR time quantum 16 milliseconds
  – $Q_2$ – FCFS (no time quantum limit)

• 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 $Q_2$

*CPU-bound: priority falls, quantum raised, I/O-bound: priority rises, quantum lowered*
Thread Scheduling

- Thread scheduling is similar
- Distinction between user-level and kernel-level threads
- When threads supported, threads scheduled, not processes

Scheduling competition
- Many-to-one and many-to-many models, thread library schedules user-level threads to run on LWP
  - Known as process-contention scope (PCS) since scheduling competition is within the process
  - Typically done via priority set by programmer
- Kernel thread scheduled onto available CPU is system-contention scope (SCS) – competition among all threads in system

LWP layer between kernel threads and user threads in some older OSs
FAQ

• Time Quantum review
• Turnaround time using time quantum review
• Context switching when there is only a single process? No. But it is unlikely situation in most modern computers.
• We assume we know the true process burst time. What if guess is wrong? SJF, pSJF
• Idea: table for comparing turnaround time etc.
• Idea: compare function call, forking a child, starting a thread
Multiple-Processor Scheduling

• CPU scheduling more complex when multiple CPUs are available.

• **Assume Homogeneous processors** within a multiprocessor

• **Asymmetric multiprocessing** – individual processors can be dedicated to specific tasks at design time

• **Symmetric multiprocessing (SMP)** – each processor is self-scheduling,
  
  – all processes in common ready queue, **or**
  
  – each has its own private queue of ready processes
    - Currently, most common

• **Processor affinity** – process has affinity for processor on which it is currently running **because of info in cache**
  
  – **soft affinity**: try but no guarantee
  
  – **hard affinity** can specify processor sets
Note that memory-placement algorithms can also consider affinity. Non-uniform memory access (NUMA), in which a CPU has faster access to some parts of main memory.
Multiple-Processor Scheduling – Load Balancing

• If SMP, need to keep all CPUs loaded for efficiency

• **Load balancing** attempts to keep workload evenly distributed
  – **Push migration** – periodic task checks load on each processor, and if found pushes task from overloaded CPU to other CPUs
  – **Pull migration** – idle processors pulls waiting task from busy processor
  – Combination of push/pull may be used.
Multicore Processors

• Recent trend to place multiple processor cores on same physical chip
• Faster and consumes less power
• Multiple threads per core now common
  – Takes advantage of memory stall to make progress on another thread while memory retrieve happens
  – See next
Multithreaded Multicore System

This is temporal multithreading. Simultaneous multithreading allows threads to compute in parallel.

Memory stalls due to cache miss
Real-Time CPU Scheduling

• Can present obvious challenges
  – **Soft real-time systems** – no guarantee as to when critical real-time process will be scheduled
  – **Hard real-time systems** – task must be serviced by its deadline

• For real-time scheduling, scheduler must support preemptive, priority-based scheduling
  – But only guarantees soft real-time

• For hard real-time must also provide ability to meet deadlines
  – **periodic** ones require CPU at constant intervals
Virtualization and Scheduling

- Virtualization software schedules multiple guests onto CPU(s)
- Each guest doing its own scheduling
  - Not knowing it doesn’t own the CPUs
  - Can effect time-of-day clocks in guests
- VMM has its own scheduler
- Various approaches have been used
  - Workload aware, Guest OS cooperation, etc.
• Solaris scheduling: 6 classes, Inverse relationship between priorities and time quantum

• Windows XP scheduling: 32 priority levels (real-time, not real-time levels)

• Linux scheduling schemes have continued to evolve.

• Linux Completely fair scheduler (CFS, 2007):
  – Variable time-slice based on number and priority of the tasks in the queue.
  – Maximum execution time based on waiting processes (Q/n).
  – Processes kept in a red-black binary tree with scheduling complexity of $O(\log N)$
  – Process with lowest weighted spent execution (virtual run time) time is picked next. Weighted by priority (“niceness”).