CS 370: OPERATING SYSTEMS
[CPU SCHEDULING]

CS370: Operating Systems [Fall 2018]
Dept. Of Computer Science, Colorado State University

October 9, 2018

Topics covered in this lecture
- Scheduling Algorithms
- Wrap-up of SJF
- Priority Scheduling
- Round robin scheduling
- Lottery scheduling
- Multilevel feedback queues
- Multiprocessor/core Environments

SHORTEST JOB FIRST (SJF)

Use of SJF in long term schedulers
- Length of the process time limit
  - Used as CPU burst estimate
- Motivate users to accurately estimate time limit
  - Lower value will give faster response times
  - Too low a value
    - Time limit exceeded error
    - Requires resubmission

The SJF algorithm and short term schedulers
- No way to know the length of the next CPU burst
  - So try to predict it
- Processes scheduled based on predicted CPU bursts

Frequently asked questions from the previous class survey
- Could we record burst times in the PCB and use this to inform scheduling?
- Could a higher priority process starve?
- How is SJF implemented if we don’t know burst times?

CS370: Operating Systems [Fall 2018]
Dept. Of Computer Science, Colorado State University

October 9, 2018
Prediction of CPU bursts:
Make estimates based on past behavior

- \( t_n \): Length of the \( n \)th CPU burst
- \( \tau_n \): Estimate for the \( n \)th CPU burst
- \( \alpha \): Controls weight of recent and past history
- \( \tau_{n+1} = \alpha t_n + (1-\alpha) \tau_n \)
- Burst is predicted as an exponential average of the measured lengths of previous CPU bursts

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

Value of \( t_n \) contains our most recent information, while \( \tau_n \) stores the past history

Each successive term has less weight than its predecessor

The choice of \( \alpha \) in our predictive equation

- If \( \alpha = 1/2 \)
  - Recent history and past history are equally weighted
- With \( \alpha = \frac{1}{2} \), successive estimates of \( \tau \)
  \[ t_0/2 + t_0/4 + t_1 + t_0/8 + t_1/2 \]
  - By the 3rd estimate, weight of what was observed at \( t_0 \) has dropped to \( 1/8 \).

The choice of \( \alpha \) in our predictive equation

- \( \tau_{n+1} = \alpha t_n + (1-\alpha) \tau_n \)
- If \( \alpha = 0 \), \( \tau_{n+1} = t_n \)
  - Current conditions are transient
- If \( \alpha = 1 \), \( \tau_{n+1} = t_n \)
  - Only most recent bursts matter
  - History is assumed to be old and irrelevant

\( \alpha \) controls the relative weight of recent and past history

- \( \tau_{n+1} = \alpha t_n + (1-\alpha) \tau_n \)
- If \( \alpha = 0 \), \( \tau_{n+1} = t_n \)
  - Current conditions are transient
- If \( \alpha = 1 \), \( \tau_{n+1} = t_n \)
  - Only most recent bursts matter
  - History is assumed to be old and irrelevant

An example: Predicting the length of the next CPU burst

<table>
<thead>
<tr>
<th>CPU burst (s)</th>
<th>“Guess” (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>6</td>
</tr>
<tr>
<td>6</td>
<td>6</td>
</tr>
<tr>
<td>13</td>
<td>13</td>
</tr>
<tr>
<td>13</td>
<td>13</td>
</tr>
<tr>
<td>10</td>
<td>8</td>
</tr>
<tr>
<td>8</td>
<td>6</td>
</tr>
<tr>
<td>5</td>
<td>4</td>
</tr>
<tr>
<td>9</td>
<td>13</td>
</tr>
<tr>
<td>12</td>
<td>13</td>
</tr>
</tbody>
</table>

Priority Scheduling
Priority Scheduling

- **Priority** associated with each process
- CPU allocated to process with highest priority
- Can be preemptive or nonpreemptive
  - If preemptive: Preempt CPU from a lower priority process when a higher one is ready

**Depiction of priority scheduling in action**

```
<table>
<thead>
<tr>
<th>Process</th>
<th>Burst Time</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>
```

Here: Lower number means higher priority

Depiction of priority scheduling in action

```
P2  P5      P1  P3  P4
0   1  6   18   16  19
Wait time = (6 + 0 + 16 + 18 + 1)/5 = 8.2
```

**How priorities are set**

- Internally defined priorities based on:
  - Measured quantities
  - Time limits, memory requirements, # of open files, ratio (averages) of I/O to CPU burst
- External priorities
  - Criteria outside the purview of the OS
  - Importance of process, $ paid for usage, politics, etc.

**Issue with priority scheduling**

- Can leave lower priority processes waiting indefinitely
- Perhaps apocryphal tale:
  - MIT’s IBM 7094 shutdown (1973) found processes from 1967!

**Coping with issues in priority scheduling:**

- Aging
  - Gradually increase priority of processes that wait for a long time
  - Example:
    - Process starts with a priority of 127 and decrements every 15 minutes
    - Process priority becomes 0 in no more than 32 hours

**Can SJF be thought of as a priority algorithm?**

- Priority is inverse of CPU burst
- The larger the burst, the lower the priority
  - Note: The number we assign to represent priority levels may vary from system to system
ROUND ROBIN SCHEDULING

Round Robin: Choosing the quantum
- Context switch is time consuming
  - Saving and loading registers and memory maps
  - Updating tables
  - Flush and reloading memory cache
  - What if quantum is 4 ms and context switch overhead is 1 ms?
    - 20% of CPU time thrown away in administrative overhead

Round Robin: Improving efficiency by increasing quantum
- Let’s say quantum is 100 ms and context switch is 1 ms
  - Now wasted time is only 1%
- But what if 50 concurrent requests come in?
  - Each with widely varying CPU requirements
  - 1st one starts immediately, 2nd one 100 ms later, ...
  - The last one may have to wait for 5 seconds!
  - A shorter quantum would have given them better service

If quantum is set longer than mean CPU burst?
- Preemption will not happen very often
  - Most processes will perform a blocking operation before quantum runs out
  - Switches happen only when process blocks and cannot continue

Quantum: Summarizing the possibilities
- Too short?
  - Too many context switches
  - Lowers CPU efficiency
- Too long?
  - Poor responses to interactive requests
A round-robin analogy

- Hyperkinetic student studying for multiple exams simultaneously
  - If you switch between paragraphs of different textbooks? [Quantum is too short]
  - You won’t get much done
  - If you never switch? [Quantum is too long]
  - You never get around to studying for some of the courses

---

Lottery scheduling

- Give processes lottery tickets for various system resources
  - E.g. CPU time
- When a scheduling decision has to be made
  - Lottery ticket is chosen at random
  - Process holding ticket gets the resource

---

Lottery Scheduling: Properties [1/2]

- Highly responsive
  - Chance of winning is proportional to tickets
- Cooperating processes may exchange tickets
  - Process A sends request to B, and then hands B all its tickets for a faster response
- Avoids starvation
  - Each process holds at least one ticket ... is guaranteed to have a non-zero probability of being scheduled

---

Lottery Scheduling: Properties [2/2]

- Solves problems that are difficult to handle in other scheduling algorithms
  - E.g. video server that is managing processes that feed video frames to clients
    - Clients need frames at 10, 20, and 25 frames/sec
    - Allocate processes 10, 20 and 25 tickets
    - CPU divided into approximately 10:20:25

---

Lottery Scheduling

- More important processes are given extra tickets
  - Increase their odds of winning
- Let’s say there are 100 outstanding tickets
  - 1 process holds 20 of these
  - Has 20% chance of winning each lottery
- A process holding a fraction \( f \) of tickets
  - Will get about a fraction \( f \) of the resource

---

All processes are equal, but some processes are more equal than others
Most commercial OS including Windows, MacOS, and Linux use this scheduling algorithm.

**MULTI-LEVEL FEEDBACK QUEUES (MFQ)**

October 9, 2018

MFQ is designed to achieve several simultaneous goals:

- Responsiveness: Run short tasks quickly as in SJF
- Low Overhead: Minimize number of preemptions, as in FIFO
  - Minimize time spent making scheduling decisions
- Starvation-Freedom: All tasks should make progress, as in Round Robin
- Background tasks: Defer system maintenance tasks, such as defragmentation, so they do not interfere with user work
- Fairness

Does MFQ achieve all of these?

- As with any real system that must balance several, conflicting goals ...
  - MFQ does not perfectly achieve any of these goals
- MFQ is intended to be a reasonable compromise in most real-world cases

MFQ: Example with 4 priority levels

<table>
<thead>
<tr>
<th>Priority</th>
<th>Time Slice (ms)</th>
<th>Round Robin Queues</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>10</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>20</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>40</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>60</td>
<td></td>
</tr>
</tbody>
</table>

Tasks and priorities:

- Tasks at a higher priority preempt lower priority tasks
- Tasks at the same priority level are scheduled in round robin fashion
- Higher priority tasks have shorter time quanta than lower priority tasks

---

SLIDES CREATED BY: SHRIDEEP PALICKARA

CS370: Operating Systems [Fall 2018]
Dept. Of Computer Science, Colorado State University
Tasks are moved between priority levels to favor short tasks over long ones.

- Every time a task uses up its time quantum, it drops a priority level.
- Every time a task yields the processor because it is waiting on I/O, it stays at the same level, or is bumped up a level.
- If the task completes, it leaves.

Impact on CPU and I/O bound processes:

- A new CPU bound process will start at high priority.
- But it will quickly exhaust its time quantum and fall to the next lower priority, and then the next ...
- An I/O bound process with a modest amount of computing will always be scheduled quickly.
- Also, keeps the disk busy.
- Compute bound tasks run with a long time quantum to minimize switching overhead while sharing processor.

What about starvation and fairness?

- If there are too many I/O bound tasks, the compute bound tasks may receive no time on the processor.
- MFQ scheduler monitors every process to ensure that it is receiving its fair share.
  - For e.g. at each level, Linux actually maintains two queues:
    - Tasks whose processes have already reached their fair share are only scheduled if other processes at the same level have also reached their fair share.
    - Periodically, processes not receiving their fair share have their tasks increased in priority.
    - Tasks that receive more than their fair share have their priority reduced.

Adjusting priority addresses strategic behavior:

- From a selfish point of view, a task can keep its priority high by doing a short I/O request just before its quantum expires.
- With MFQ this will be detected, and its priority reduced to its fair level.

Dispatcher in Windows XP:

- Use a queue for each scheduling priority.
- Traverse the queues from highest to lowest.
- Until it finds a thread that is ready to run.
- If no ready thread is found?
  - Dispatcher will execute a special thread: idle thread.
Idle thread in Windows

- Primary purpose is to **eliminate a special case**
- Cases when no threads are runnable or ready
- Idle threads are always in a **ready** state
  - If not already running
- Scheduler can always find a thread to execute
- If there are other eligible threads?
- Scheduler will never select the idle thread

**Windows thread priorities go from 0-31**
- Idle thread priority can be thought of as -1
- Threads in the system idle process can also implement CPU power saving
  - On x86 processors, run a loop of `halt` instructions
  - Causes CPU to turn off internal components
  - Until an interrupt request arrives
- Recent versions also reduce the CPU clock speed

Time consumed by the idle process

- It may seem that the idle process is monopolizing the CPU
  - It is merely acting as a placeholder during free time
  - Proof that no other process wants that CPU time

Load balancing: Migration based approaches

- **Push migration**
  - Specific task periodically checks for imbalance
  - Balances load by **pushing** processes from overloaded to less-busy processors.
- **Pull migration**
  - Idle processor pulls a waiting task from busy processor
- Schemes **not mutually exclusive**: used in parallel
  - Linux: Runs a load-balancing algorithm
    - Every 200 ms: (PUSH migration)
    - When processor run-queue is empty: (PULL migration)

Multicore processors place multiple processor cores on same physical chip

- Each core has its own register set
  - Appears to the OS as a separate physical processor
- Recent designs implement 2 or more hardware threads per core
  - If there is a memory stall (due to cache miss) on one thread, **switch** to another hardware thread
Coping with memory stalls

- **Compute cycle**
  - Thread
  - Thread

- **Memory cycle**
  - Thread
  - Thread

Multithreading a processor

- **Coarse grained**
  - Thread executes on processor till a memory stall
  - Switch to another thread

- **Switching between threads**
  - Flush the instruction pipeline
  - Refill pipeline as new thread executes

- **Finer grained (or interleaved)**
  - Switch between threads at the boundary of an instruction cycle
  - Design includes logic for thread switching; overheads are low

Tiered scheduling on multicore processors

- **First-level: OS**
  - OS chooses which software thread to run on each hardware thread

- **Second-level: Core**
  - Decides which hardware thread to run

- **UltraSPARC T1**
  - 8 cores, and 4 hardware threads/core
  - Round robin to schedule hardware threads on core

The contents of this slide-set are based on the following references

