Lecture 7: Scheduling (2)

Instructor: Chengyu Song
Slide contributions from
Nael Abu-Ghazaleh, Harsha Madhyvasta and Zhiyun Qian
Scheduling Overview

- Scheduler runs when we context switching among execution units (jobs/processes/threads) to pick who runs next
  - Under what situation does this occur?
  - What should it do? Does it matter?
- Making this decision is called scheduling
- Now, we’ll look at:
  - The goals of scheduling
  - Starvation
  - Various well-known scheduling algorithms
  - Standard Unix scheduling algorithm
Round Robin vs. FIFO

- Many context switches can be costly
- Other than that, is Round Robin always better than FIFO?
Mixed Workload

Tasks

I/O bound

\[ \downarrow \text{issues} \]

\[ \downarrow \text{I/O} \text{ request} \]

I/O completes

CPU bound

I/O completes

\[ \uparrow \text{gets} \]

\[ \uparrow \text{CPU} \]

CPU bound

Time
Max-Min Fairness

- How do we balance a mixture of repeating tasks:
  - Some I/O bound, need only a little CPU
  - Some compute bound, can use as much CPU as they are assigned

- One approach: maximize the minimum allocation given to a task
  - Schedule the smallest task first, then split the remaining time using max-min
Priority Scheduling

- Priority Scheduling
  - Choose next job based on priority
    - Airline checkin for first class passengers
  - Can implement SJF, priority = 1/(expected CPU burst)
  - Also can be either preemptive or non-preemptive

- Problem?
  - Starvation – low priority jobs can wait indefinitely

- Solution
  - “Age” processes
    - Increase priority as a function of waiting time
    - Decrease priority as a function of CPU consumption
More on Priority Scheduling

- For real-time (predictable) systems, priority is often used to isolate a process from those with lower priority. *Priority inversion* is a risk unless all resources are jointly scheduled.

```

P_L
x->Acquire()            x->Release()

P_H
x->Acquire()

```

- How can this be avoided?
Priority Inheritance

- If lower priority process is being waited on by a higher priority process it inherits its priority
  - How does this help?
  - Does it prevent the previous problem?

- Priority inversion is a big problem for real-time systems
  - Mars pathfinder bug (link)
Problems of basic algorithms

- FIFO: Good: fairness; bad: turnaround time, response time
- SJF: good: turnaround time; bad: fairness, response time, need to estimate run-time
- RR: good: fairness, response time; bad: turnaround time
- Is there a scheduler that balances these issues better?
  - Challenge: limited information about a process in the beginning
  - Challenge: how to prevent gaming the scheduler to get more run-time
MLQ: combining algorithms

- Scheduling algorithms can be combined
  - Have multiple queues
  - Use a different algorithm for each queue
  - Move jobs among queues

- Example: Multiple-level feedback queues (MLFQ)
  - Multiple queues representing different job types
    » Interactive, CPU-bound, batch, system, etc.
  - Queues have priorities, jobs on same queue scheduled RR
  - Jobs can move among queues based upon execution history
    » Feedback: Switch from interactive to CPU-bound behavior
Multi-level Feedback Queue

- Goals:
  - Responsiveness
  - Low overhead
  - Starvation freedom
  - Some tasks are high/low priority
  - Fairness (among equal priority tasks)

- Not perfect at any of them!
  - Used in Linux (and probably Windows, MacOS)
MLFQ

- Set of Round Robin queues
  - Each queue has a separate priority
- High priority queues have short time slices
  - Low priority queues have long time slices
  - Why?
- Scheduler picks first thread in highest priority queue
- Jobs start in highest priority queue
  - Assumes good behavior
- If a job used up the entire time slice, its priority drops one level
  - Otherwise it retains its priority
### MLFQ

<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>time slice expiration</td>
</tr>
<tr>
<td>3</td>
<td>40</td>
<td>new or I/O bound task</td>
</tr>
<tr>
<td>4</td>
<td>80</td>
<td></td>
</tr>
</tbody>
</table>
MLFQ: Starvation and Gaming

- Wait ... this design still allows starvation
  - Why?

- How to solve this issue?
  - After some period S, reset the priority by moving every job to the highest priority

- Can a job abuse the scheduler to increase its running time?
  - Jobs can deliberately relinquish the CPU before slice expires
  - Solution: using allotment time instead of one slice
Unix Scheduler

● The canonical Unix scheduler uses a MLFQ
  ◆ 3-4 classes spanning ~170 priority levels
    » Timesharing: first 60 priorities
    » System: next 40 priorities
    » Real-time: next 60 priorities
    » Interrupt: next 10 (Solaris)

● Priority scheduling across queues, RR within a queue
  ◆ The process with the highest priority always runs
  ◆ Processes with the same priority are scheduled RR

● Processes dynamically change priority
  ◆ Increases over time if process blocks before end of quantum
  ◆ Decreases over time if process uses entire quantum
Motivation of Unix Scheduler

- The idea behind the Unix scheduler is to reward interactive processes over CPU hogs.
- Interactive processes (shell, editor, etc.) typically run using short CPU bursts:
  - They do not finish quantum before waiting for more input.
- Want to minimize response time:
  - Time from keystroke (putting process on ready queue) to executing keystroke handler (process running).
  - Don’t want editor to wait until CPU hog finishes quantum.
- This policy delays execution of CPU-bound jobs:
  - But that’s ok.
Other Scheduling Ideas

- Lottery scheduler: Give processes tickets proportional to their priority
  - Linux cgroup

- Stride Scheduler (also known as proportional share): Like lottery but more predictable
  - Basis for Linux Completely Fair Scheduler

- Scheduling for heterogeneous systems
- Scheduling for distributed systems/cloud
- …
Multiprocessor Scheduling

- This is its own topic, we wont go into it in detail
  - Could come back to it towards the end of the quarter

- What would happen if we used MFQ on a multiprocessor?
  - Contention for scheduler spinlock
  - Multiple MFQ used – this optimization technique is called distributed locking and is common in concurrent programming

- A couple of other considerations
  - Co-scheduling for parallel programs
  - Core affinity
Scheduling Summary

- Scheduler (dispatcher) is the module that gets invoked when a context switch needs to happen
- Scheduling algorithm determines which job runs next, where jobs are placed on queues
- Many potential goals of scheduling algorithms
  - Utilization, throughput, wait time, response time, fairness, etc.
- Various algorithms to meet these goals
  - FCFS/FIFO, SJF, RR, Priority
- Can combine algorithms
  - Multiple-level feedback queues
  - Unix example
Next class

- Thread
- Preparation
  - Read Module 25, 26, and 27