CS 112/212 Project 1: Threads

January 7, 2022
Today’s Topics

● Project Overview
● Project 1 Requirements
  ○ Alarm Clock
  ○ Priority Scheduler
  ○ Advanced Scheduler
● Getting Started
## Project Overview

**Reference Implementation:**

<table>
<thead>
<tr>
<th>File Name</th>
<th>Changes</th>
</tr>
</thead>
<tbody>
<tr>
<td>devices/timer.c</td>
<td>42 ++</td>
</tr>
<tr>
<td>threads/fixed-point.h</td>
<td>120 ++++++++</td>
</tr>
<tr>
<td>threads/synch.c</td>
<td>88 ++++++</td>
</tr>
<tr>
<td>threads/thread.c</td>
<td>196 ++++++++++++++++++++++++++</td>
</tr>
<tr>
<td>threads/thread.h</td>
<td>23 +</td>
</tr>
</tbody>
</table>

5 files changed, 440 insertions(+), 29 deletions(-)

- Most changes in threads and device directories
- Also look in lib/kernel for useful data structures: list, hash, bitmap
Synchronization

Serializing access to shared resource

- **Disabling interrupts:**
  - Turns off thread preemption; only one thread can run
  - Undesirable unless absolutely necessary
- **Synchronization primitives:** (*threads/synch.h*)
  - Semaphores
  - Locks
  - Condition Variables
Thread Basics

- New
- Admitted
- Ready
- Waiting
- Event Completion
- Scheduled
- Interrupted
- IO or wait
- Running
- Exit
- Dying
Project 1 Requirements

(Chapter 2.2)
Alarm Clock

- **Reimplement** `timer_sleep()` **to avoid busy waiting**
- `void timer_sleep(int64_t ticks)`
  - Suspends execution of the calling thread until time as advanced by at least `ticks` timer ticks
  - Existing implementation uses “busy waiting”
- Details in [Section 2.2.2](#)
Priority Scheduling

- **Replace round-robin scheduler with a priority-based scheduler**
  - Always run a thread with the highest priority among all ready threads
    - Round-robin threads of the same highest priority
    - Yield immediately when a higher priority thread is ready
    - May starve other threads
  - Most code will be in `thread.h/c`
- **Implement “Priority Donation”** (solves “Priority Inversion”)
- Details in [Section 2.2.3](#)
Priority Inversion

- **Priority Inversion:** A low priority thread holds a resource needed by a higher priority thread

- H is blocked while waiting on L, and M has a higher priority than L
- H can’t run because L can’t release its lock because M is running
- Solution: *priority donation*
Priority Donation

- **Priority Donation:** A higher priority thread “donates” its priority to the lower priority thread it is blocked on

- H “donates” its priority to L so that L runs with high effective priority
- When L releases the lock, L’s priority returns to its old value
- H then runs immediately
Priority Donation

Things to consider:

● To how many threads can a donor donate its priority?
● From how many threads may a donee receive priority?
● What happens when a priority recipient donates to another thread?
Advanced Scheduler

- Implement a multilevel feedback queue scheduler similar to the 4.4 BSD Scheduler
- Multilevel feedback queue scheduler tries to be fair with CPU time
  - No priority donation
  - Give highest priority to thread that has used the least CPU time recently
  - Prioritizes interactive and I/O-bound threads
  - De-prioritizes CPU-bound threads
- The scheduling algorithm must be configurable at startup time
- Details in Section 2.2.4 and Appendix B
Advanced Scheduler

\[
priority = PRI_{MAX} - \frac{recent\_cpu}{4} - nice \times 2
\]

- Details in Appendix B.2
Advanced Scheduler: nice

- nice allows threads to declare how generous they want to be with their own CPU time
- Integer value between -20 and 20
  - nice > 0: lower effective priority, gives away CPU time
  - nice < 0: higher effective priority, takes away CPU time from other threads
- Details in Appendix B.1
Advanced Scheduler: recent_cpu

- recent_cpu: amount of CPU time a thread has “recently” received
- Exponentially weighted moving average
- Incremented every clock tick when a thread is running
- Recomputed for all threads every second:
  \[
  \text{recent_cpu} = \frac{2\times\text{load_avg}}{2\times\text{load_avg} + 1} \times \text{recent_cpu} + \text{nice}
  \]
- Details in Appendix B.3
Advanced Scheduler: load_avg

- load_avg: Average number of ready threads in the last minute
- Single value system-wide
- Initialized to zero
- Recomputed every second:
  \[ \text{load}_\text{avg} = (59/60) \times \text{load}_\text{avg} + (1/60) \times \text{ready}_\text{threads} \]
- Details in Appendix B.4
Getting Started

- Start early!
- Read the documentation and the source code
- Setup/use version control (git)
  - Remember to keep your repositories private
- Design your solution, data structure, and synchronization scheme before you start coding
- Work together: meet/commit/merge often
- Grading: 50% project tests, 50% code and write-up
Git Commands

- git clone
- git add
- git commit
- git branch
- git merge
- git stash
- git pull
- git push
- git rebase
Git Recommendations

Some guidelines & ideas:

- Write helpful commit messages. They exist only for you and your team!
- Host your code on Github or Bitbucket as a “master” copy. **Use a private repository!**
- Create per-assignment branches. Work on topic branches; merge into assignment branches and delete once the topic is “done”.
- Stay synchronized with your team: fetch and push often.
- Commit often. Use `git bisect` to find regression bugs.

Read or skim [Pro Git](https://git-scm.com/book) for fuller advice.