CS 140 Project 1: Threads

January 10, 2020

Today's Topics

- Overview
- Project 1 Requirements
    - Alarm Clock
    - Priority Scheduler
    - Advanced Scheduler
- Getting Started

Project Overview

Reference Implementation:

```
devices/timer.c         |   42 +++++-
threads/fixed-point.h   |  120 ++++++++++++++++++
threads/synch.c         |   88 ++++++++++++-
threads/thread.c        |  196 ++++++++++++++++++++++++++----
threads/thread.h        |   23 +++
5 files changed, 440 insertions(+), 29 deletions(-)
```

- Most changes in `threads` and `devices` directories.

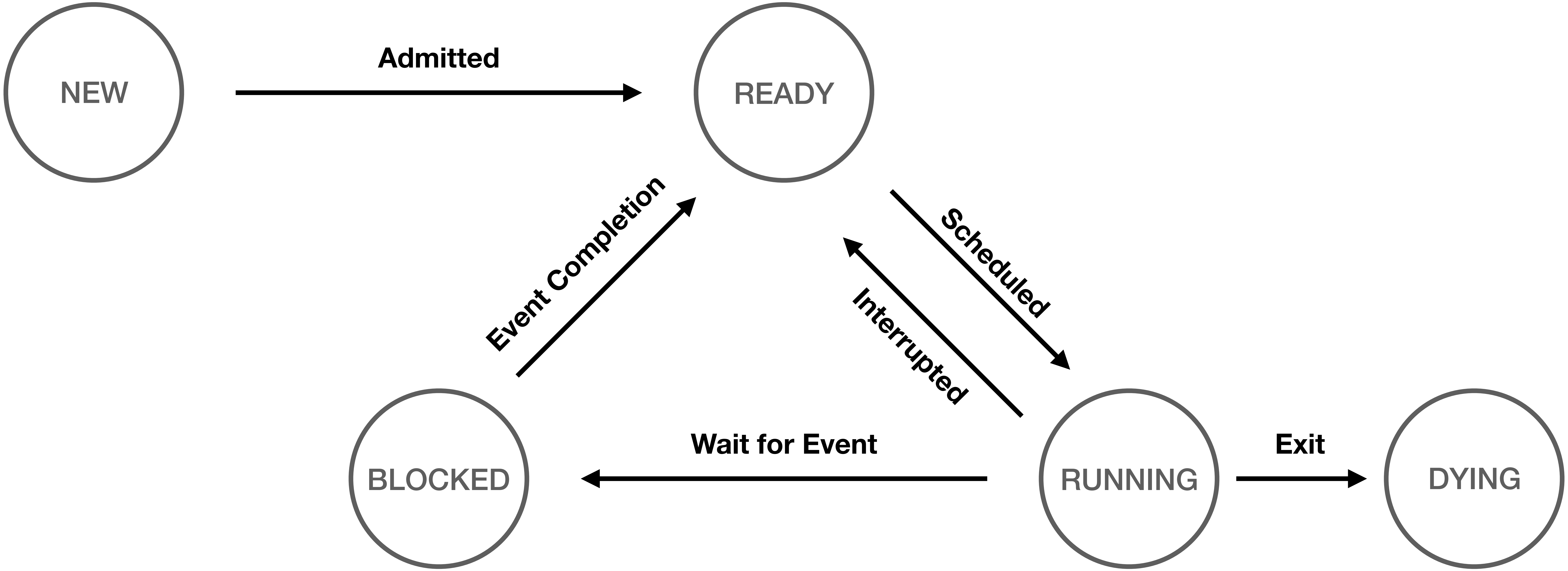
- Also look in `lib/kernel` for useful data structures: `list`, `hash`, `bitmap`.

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

Pintos Thread States



Project 1 Requirements

(Chapter 2.2)

Alarm Clock

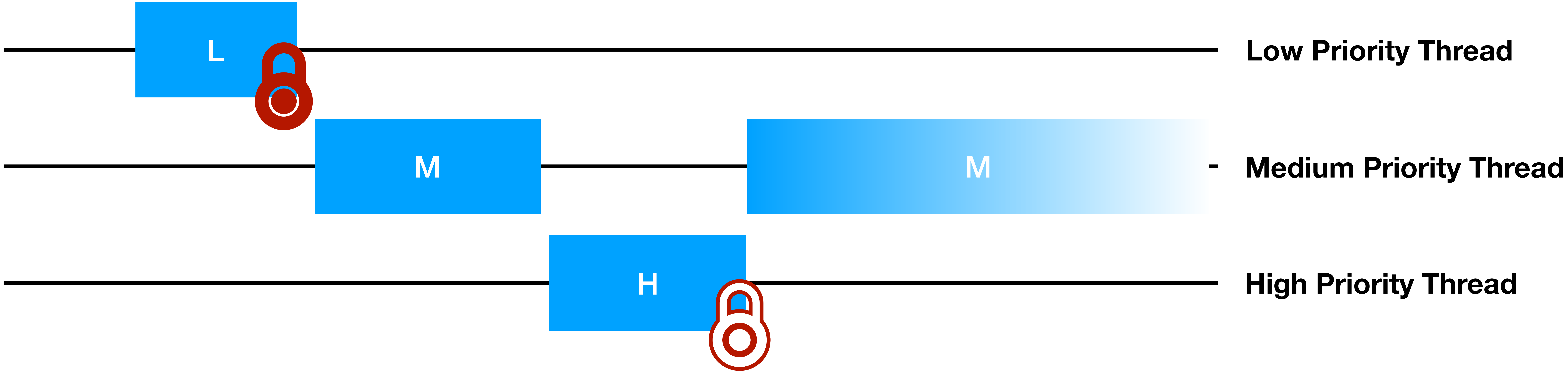
- **Reimplement `timer_sleep()` to avoid busy waiting**
- `timer_sleep()` in “devices/timer.c”
  - `void timer_sleep (int64_t ticks)`
  - Suspends execution of the calling thread until time has advanced by at least `ticks` timer ticks.
  - Existing implantation uses “busy waiting.”
- Details in section 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 Scheduling: *Priority Inversion*

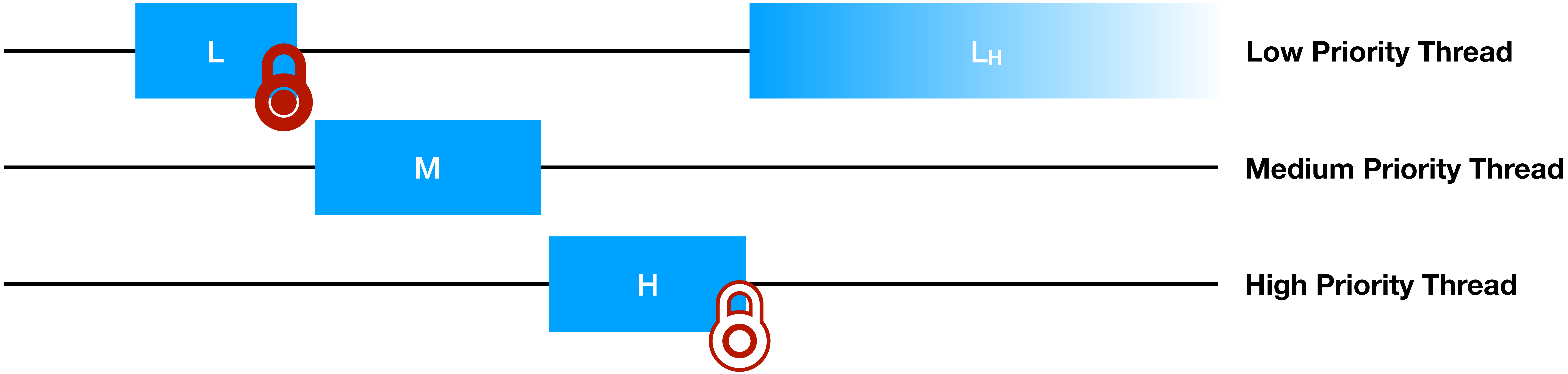
- **Priority Inversion:** *A low priority thread holds a resource needed by a higher priority thread*



- H is blocked on L and M has higher priority than L
- H is can’t run because L can’t run because M is running
- Solved using “*Priority Donation*”

Priority Scheduling: *Priority Donation*

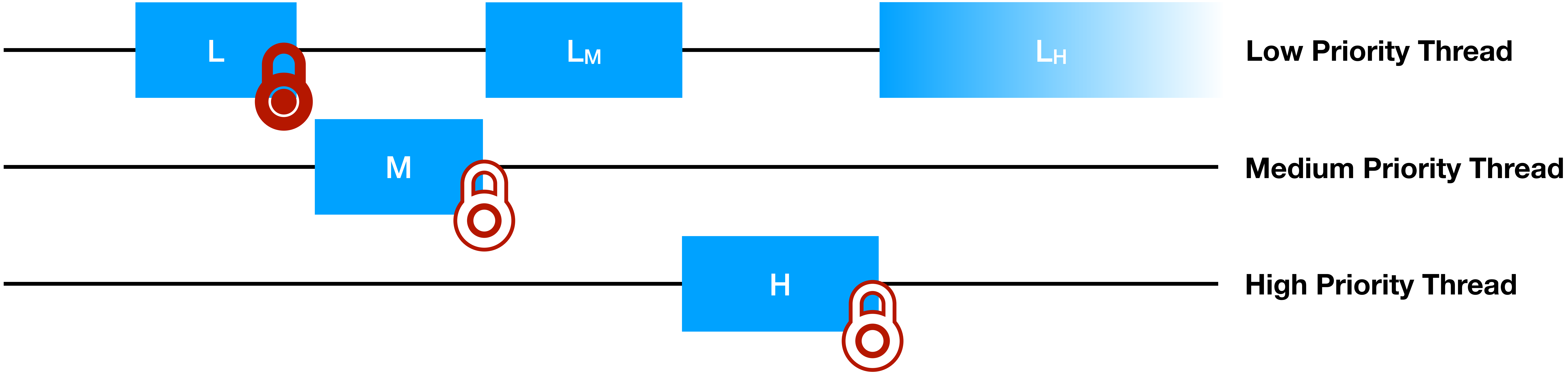
- **Priority Donation:** *A higher priority thread “donates” it’s priority to the lower priority thread it is blocked on*



- H "donates" its priority to L so L runs with high effective priority
- When L releases lock, L's priority returns to old value
- H then runs immediately

Priority Scheduling: *Multiple Priority Donation*

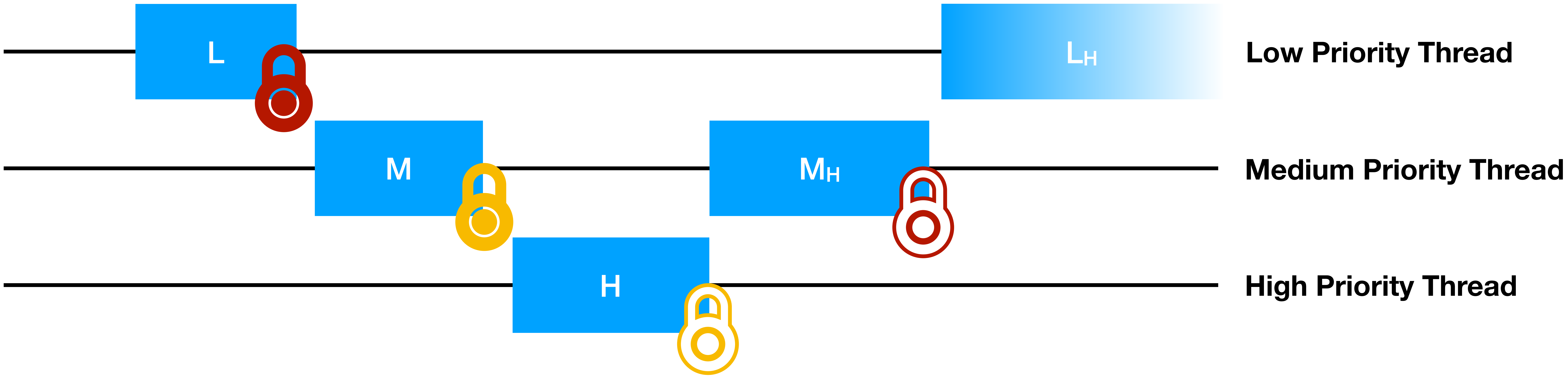
- **Multiple Priority Donation:** *Effective priority is the max of donated priorities*



- L had donated priorities from both M and H
- L runs at the highest of its donated priorities

Priority Scheduling: *Chained Priority Donation*

- **Chained Priority Donation:** *Donated priorities propagate through a chain a dependancies*



- H donates priority to M
- M donates priority to L

Priority Scheduling

- `void thread_set_priority (int new_priority)`
  - Set the current thread’s priority to `new_priority`
  - Yield if the thread no longer has the highest priority
  - If thread has donated priority, it still operates at the donated priority
- `int thread_get _priority ()`
  - Returns the current thread’s priority
  - With priority donation returns the higher (donated) priority

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

- Details in section 2.2.4 and Appendix B

Advanced Scheduler

```
priority = PRI_MAX - recent_cpu/4 - nice*2
```

Details in Appendix B.2

Advanced Scheduler: `nice`

- `nice` allows threads to declare how generous they want be with there 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:` CPU time a thread has “recently” received
- Exponentially waited moving average
- Incremented every clock tick when a thread is running
- Recomputed for all threads every second:

```
recent_cpu = (2*load_avg)/(2*load_avg + 1) * recent_cpu + 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:

```
load_avg = (59/60)*load_avg + (1/60)*ready_threads
```

- Details in [Appendix B.4]()

Advanced Scheduler

- add -mlfqs kernel option
  - Must allow the scheduling algorithm to be configured at startup time
  - add to parse_options()
- No priority donation
  - `thread_set_priority()` should do nothing
  - `thread_get_priority()` returns priority calculated by scheduler
- Details in section 2.2.4 and Appendix B

Getting Started

- Start early!
- Read the documentation and the source code
- Setup/use version control (git)
    - Remember to **keep your repositories private**
- Design 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 and stash 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 for fuller advice.