CS 112/212 Section 1, Project 1: Threads

Gordon Martinez-Piedra

Plan for Today

- How to get started with Pintos
- Explain Project 1 Requirements
- Tip & Tidbits!

How to get started with Pintos?

READ BEFORE WRITE!

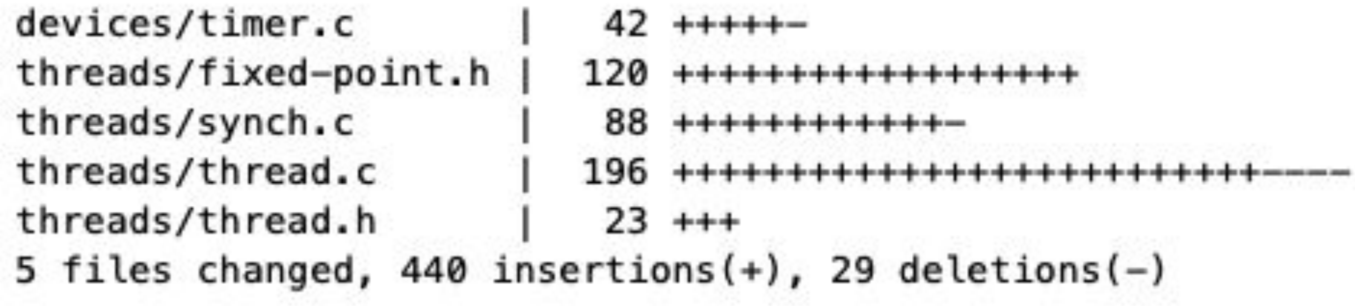
Read Appendices A1-A5 - essential for doing this project A2,3,4 are MOST important

Then start setting up Pintos

Familiarize yourself with debugging in pintos

Try tracing through program execution with GDB

Read code in the relevant files that you will be changing and trace through execution with help of A1 and A2 and Proj 1.



```
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(-)
```

Overview of Requirements

Requirements

1. Alarm Clock
    a. Re-implement timer_sleep() without busy waiting
2. Priority Scheduler
    a. Threads set their own priorities, and run according to these priorities
    b. Priority donation for locks
3. Advanced Scheduler
    a. Thread priorities are calculated by the system, and run according to these priorities
    b. No priority donation
4. Design Doc
    a. Answer questions regarding your design and implementation for parts 1-3

Grading

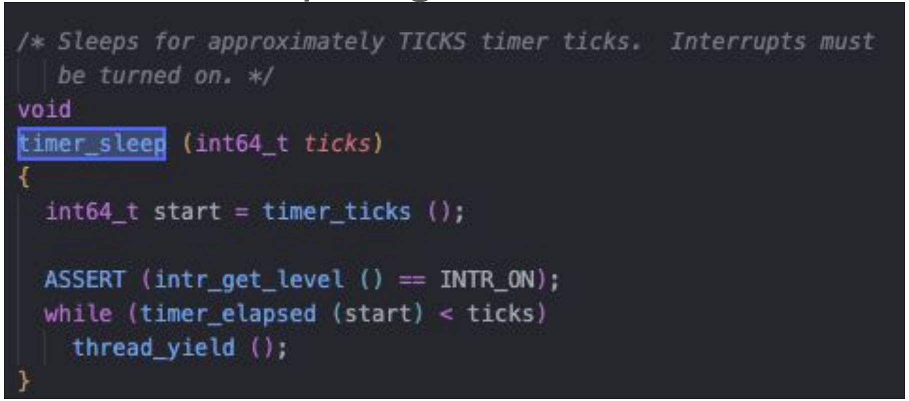
50% DESIGN

50% Tests

Requirement 1: Alarm Clock

Alarm Clock Overview

Call timer_sleep() to make current thread sleep for give # of ticks



```
/* Sleeps for approximately TICKS timer ticks.  Interrupts must
   be turned on. */
void
timer_sleep (int64_t ticks)
{
  int64_t start = timer_ticks ();

  ASSERT (intr_get_level () == INTR_ON);
  while (timer_elapsed (start) < ticks)
    thread_yield ();
}
```

Currently we do busy waiting with thread_yield() loop.

Alarm Clock Overview

Your job - re-implement timer_sleep() with synchronization primitives

Key Considerations:

- How will you avoid busy waiting?
- How will you keep track of sleeping threads?
- Where in the code will you wake up sleeping threads?
- **Check out the design doc** to see what race conditions you should watch out for!

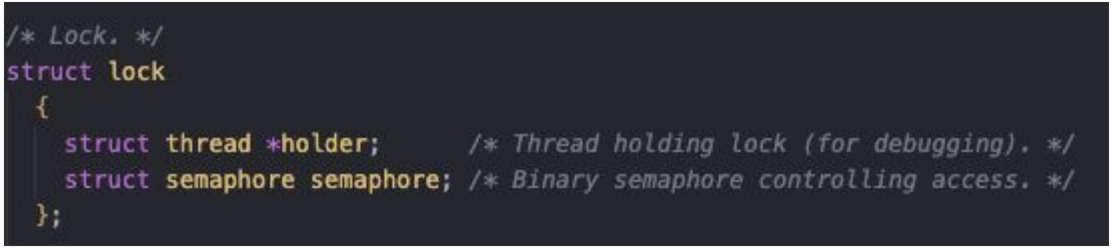
Questions?

Requirement 2: Priority Scheduling

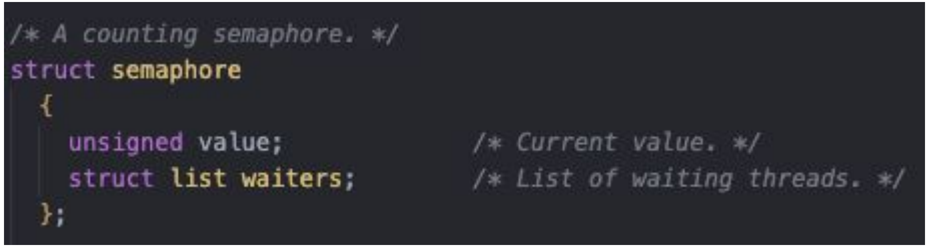
Priority Scheduling Overview

- Threads with higher priority should be run first (minimum priority = 0, PRI_DEFAULT=31, PRI_MAX = 63)
- When threads are waiting on synchronization primitives (lock, semaphore, condition variable) highest priority waiting thread should be wakened first
- Implement priority donation for locks to deal with PRIORITY INVERSION

Priority Inversion Problem



```
/* Lock. */
struct lock
  {
    struct thread *holder;      /* Thread holding lock (for debugging). */
    struct semaphore semaphore; /* Binary semaphore controlling access. */
  };
```



```
/* A counting semaphore. */
struct semaphore
  {
    unsigned value;             /* Current value. */
    struct list waiters;        /* List of waiting threads. */
  };
```

The Priority Inversion Problem

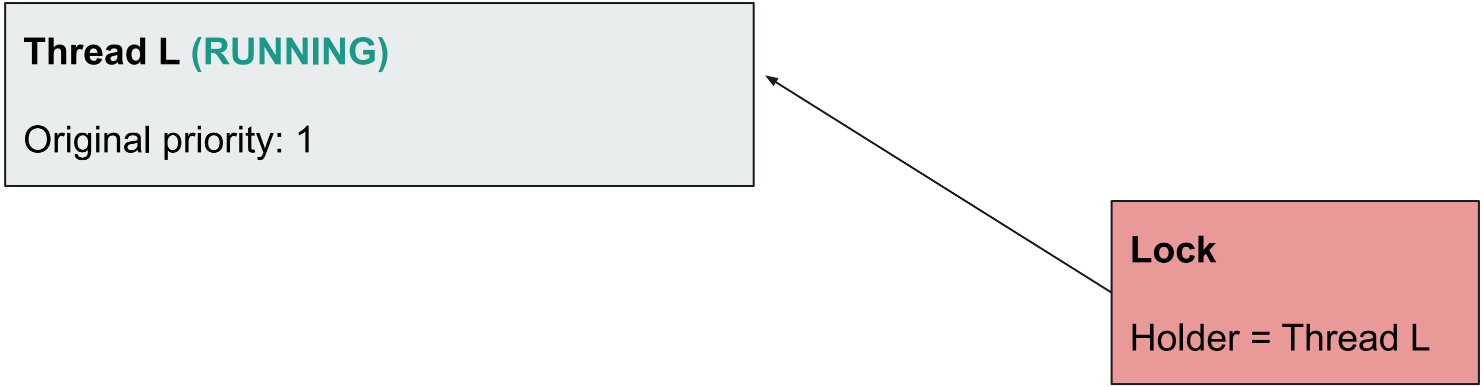
Thread L

Original priority: 1

Lock

Holder = NULL

The Priority Inversion Problem



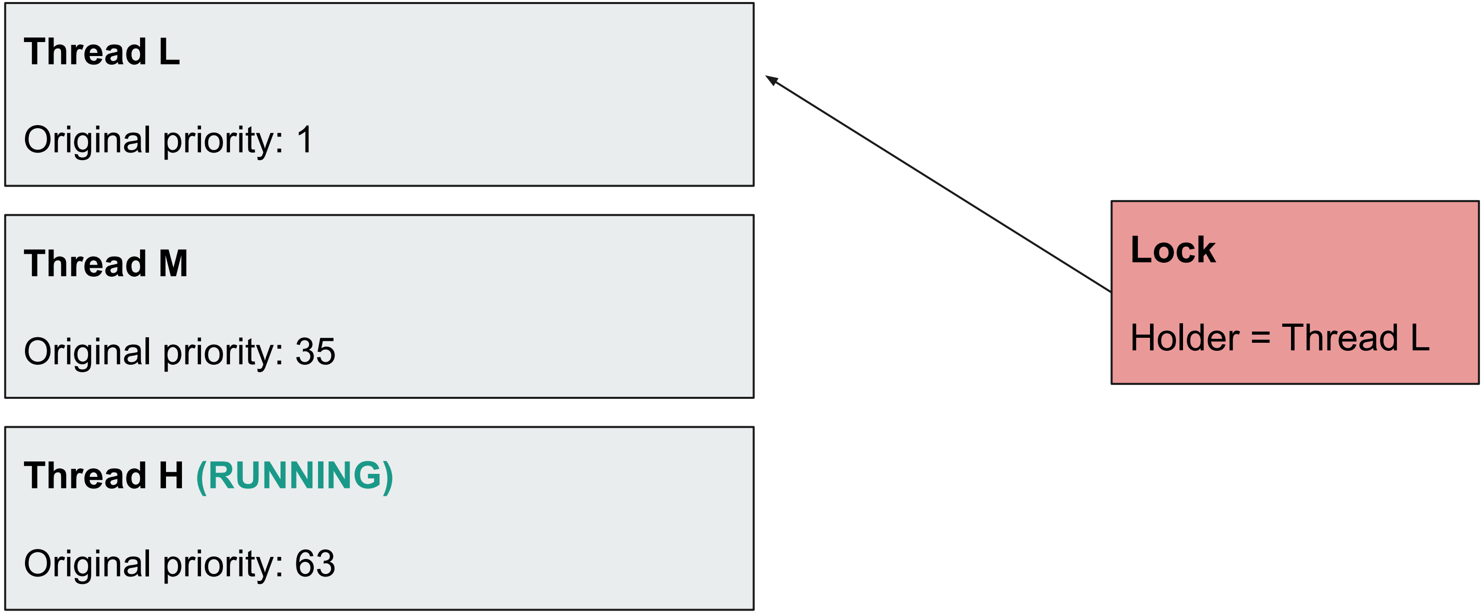
Thread L (RUNNING)

Original priority: 1

Lock

Holder = Thread L

The Priority Inversion Problem



Thread L

Original priority: 1

Thread M

Original priority: 35

Thread H (RUNNING)

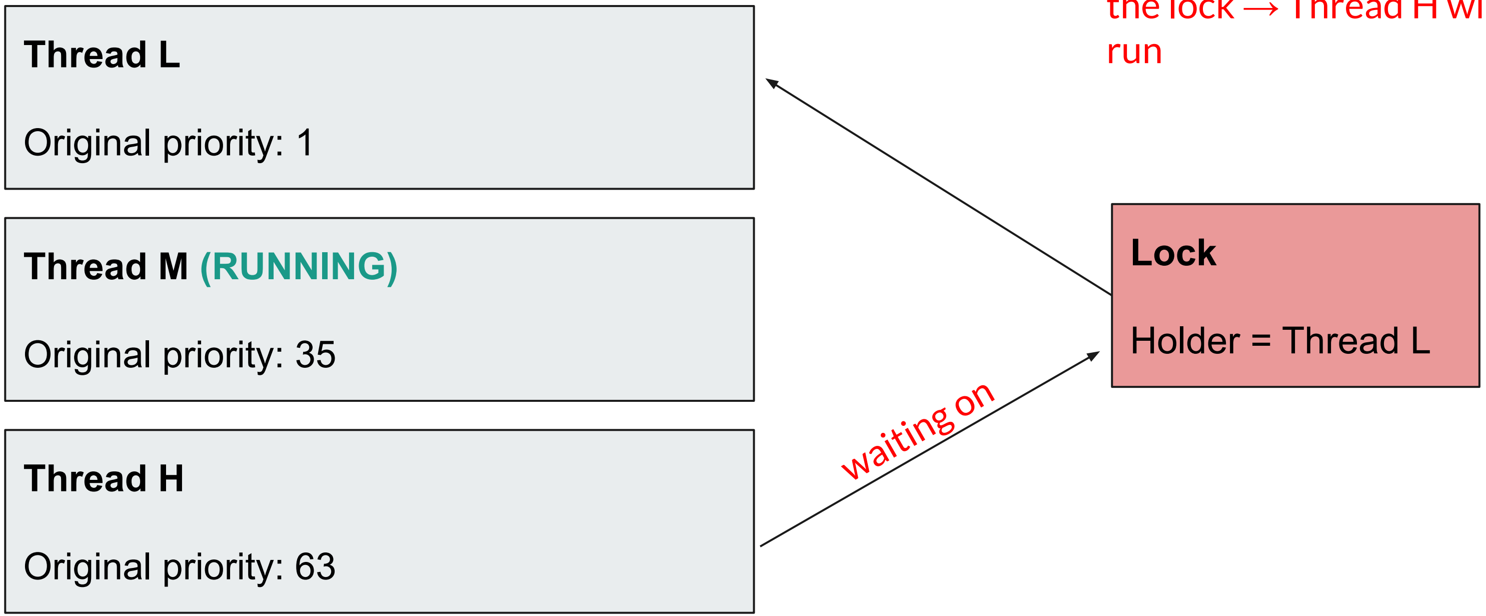
Original priority: 63

Lock

Holder = Thread L

The Priority Inversion Problem

- Thread H is taken off of CPU, because it is waiting for Lock
- Thread M will run because it has a higher priority than Thread L
- Therefore, Thread L will not release the lock → Thread H will not get to run



Thread L

Original priority: 1

Thread M (RUNNING)

Original priority: 35

Thread H

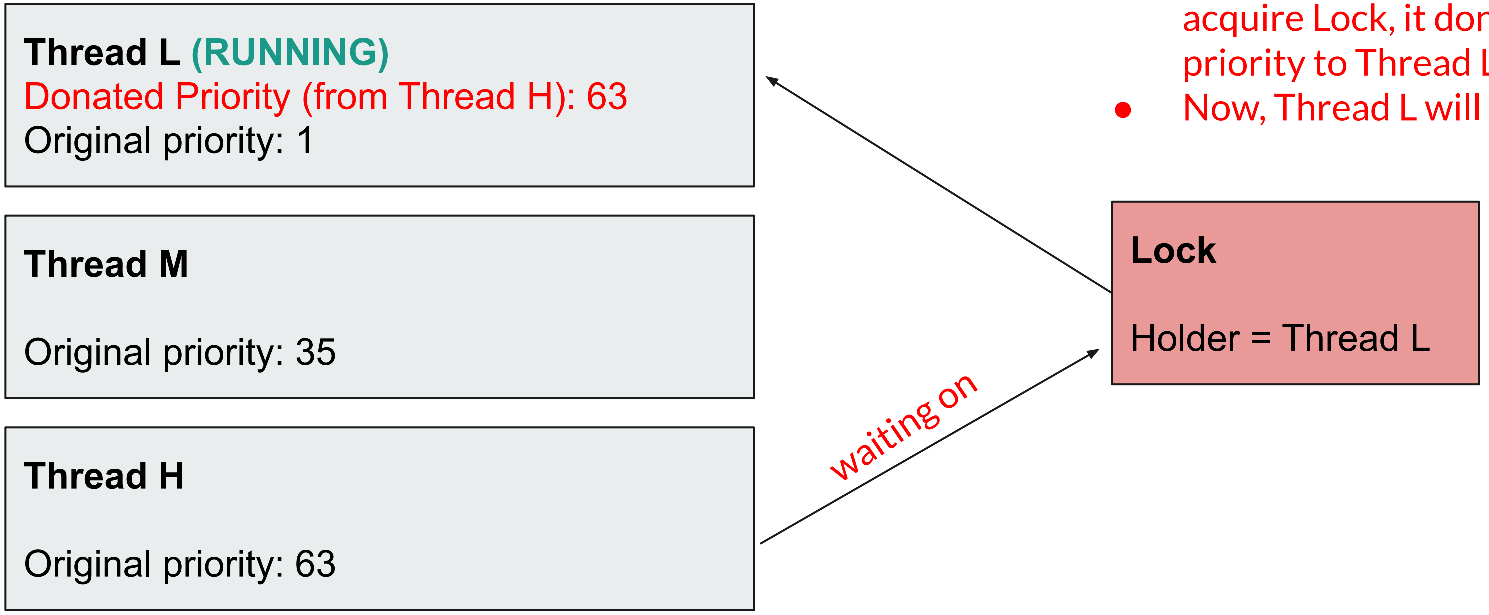
Original priority: 63

Lock

Holder = Thread L

Priority Donation: Example 1 (to Fix Priority Inversion)

- When Thread H tries to acquire Lock, it donates its priority to Thread L
- Now, Thread L will get to run



Thread L **(RUNNING)**
Donated Priority (from Thread H): 63
Original priority: 1

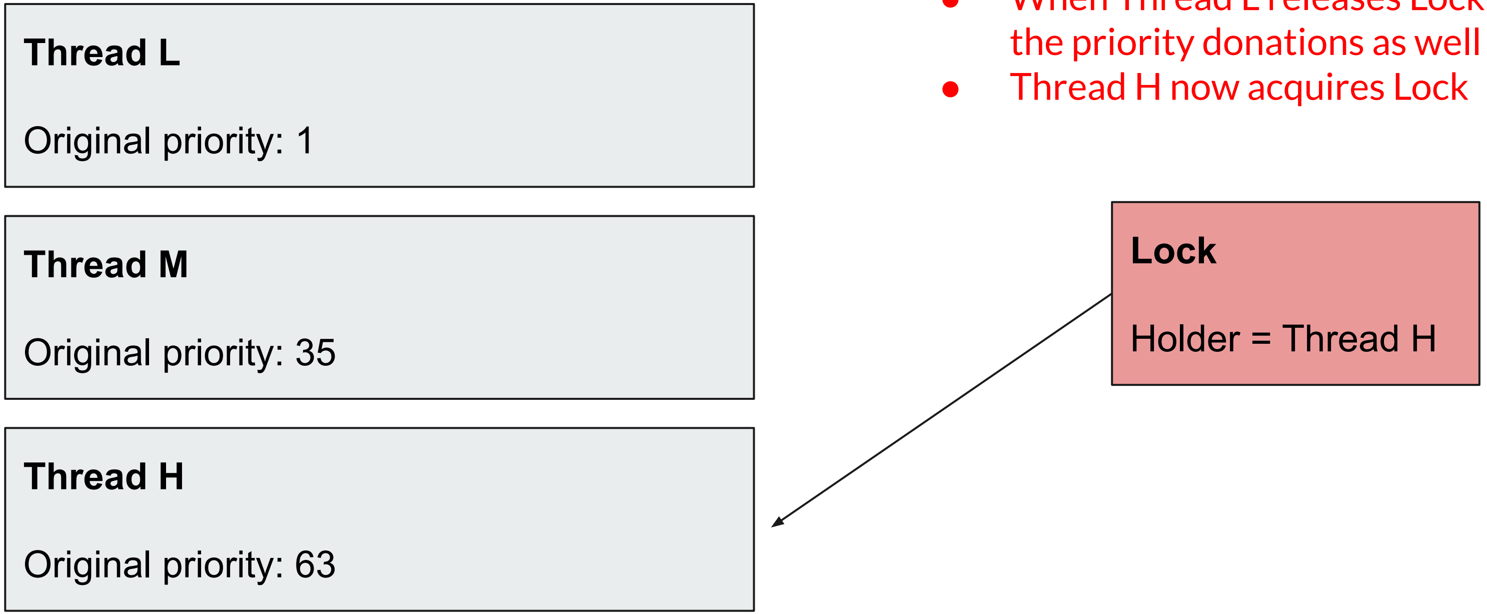
Thread M
Original priority: 35

Thread H
Original priority: 63

Lock
Holder = Thread L

Priority Donation: Example 1

- When Thread L releases Lock, it releases the priority donations as well
- Thread H now acquires Lock



Thread L

Original priority: 1

Thread M

Original priority: 35

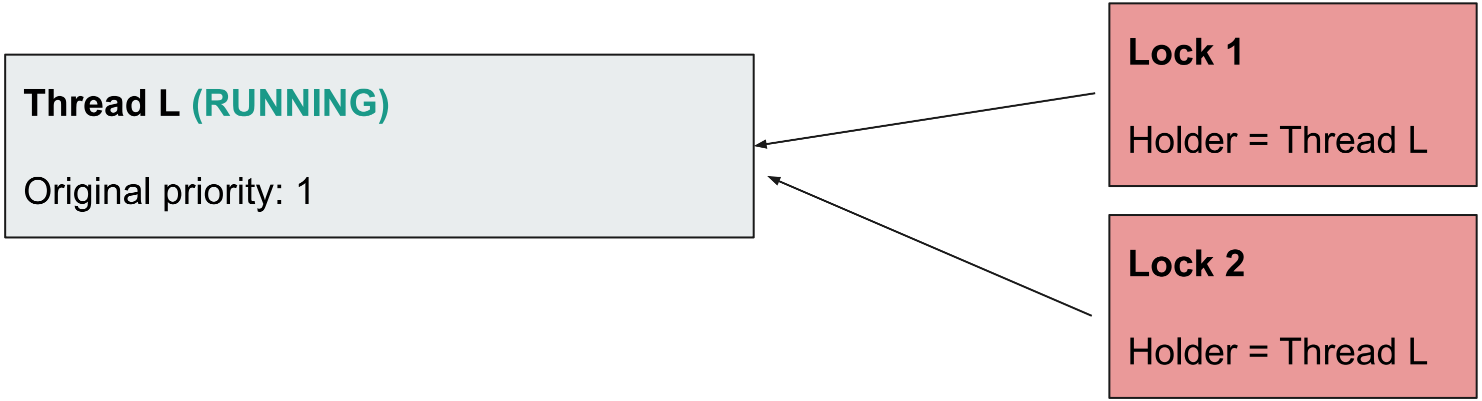
Thread H

Original priority: 63

Lock

Holder = Thread H

Priority Donation Example 2: Multiple Donations



Thread L (RUNNING)

Original priority: 1

Lock 1

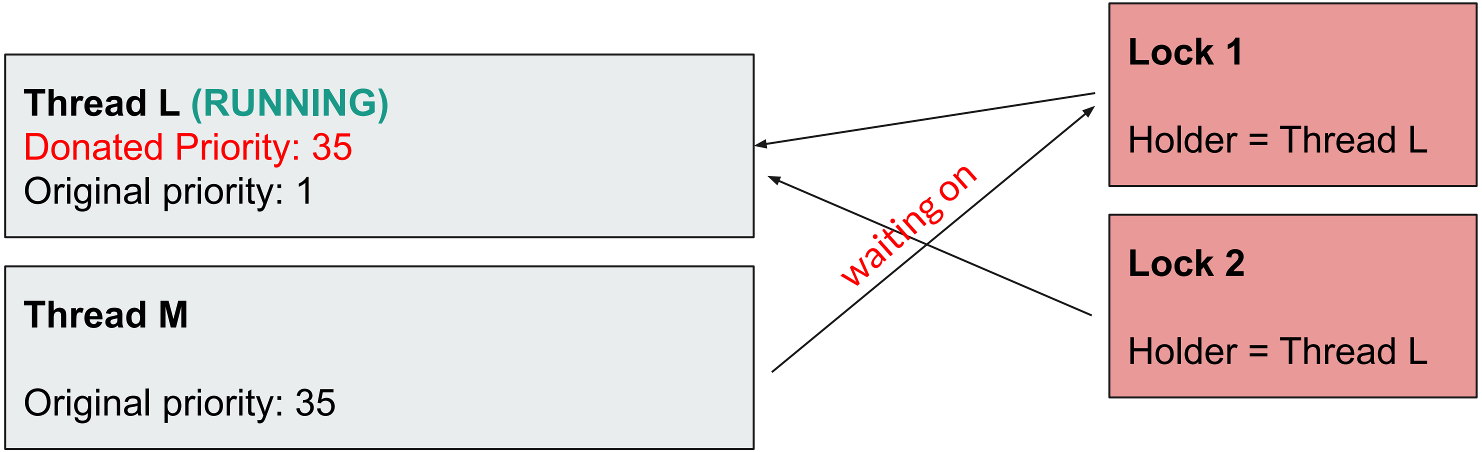
Holder = Thread L

Lock 2

Holder = Thread L

Thread M tries to acquire Lock 1, so donates its priority to Thread L

Priority Donation Example 2: Multiple Donations



Thread L **(RUNNING)**
Donated Priority: 35
Original priority: 1

Thread M

Original priority: 35

Lock 1

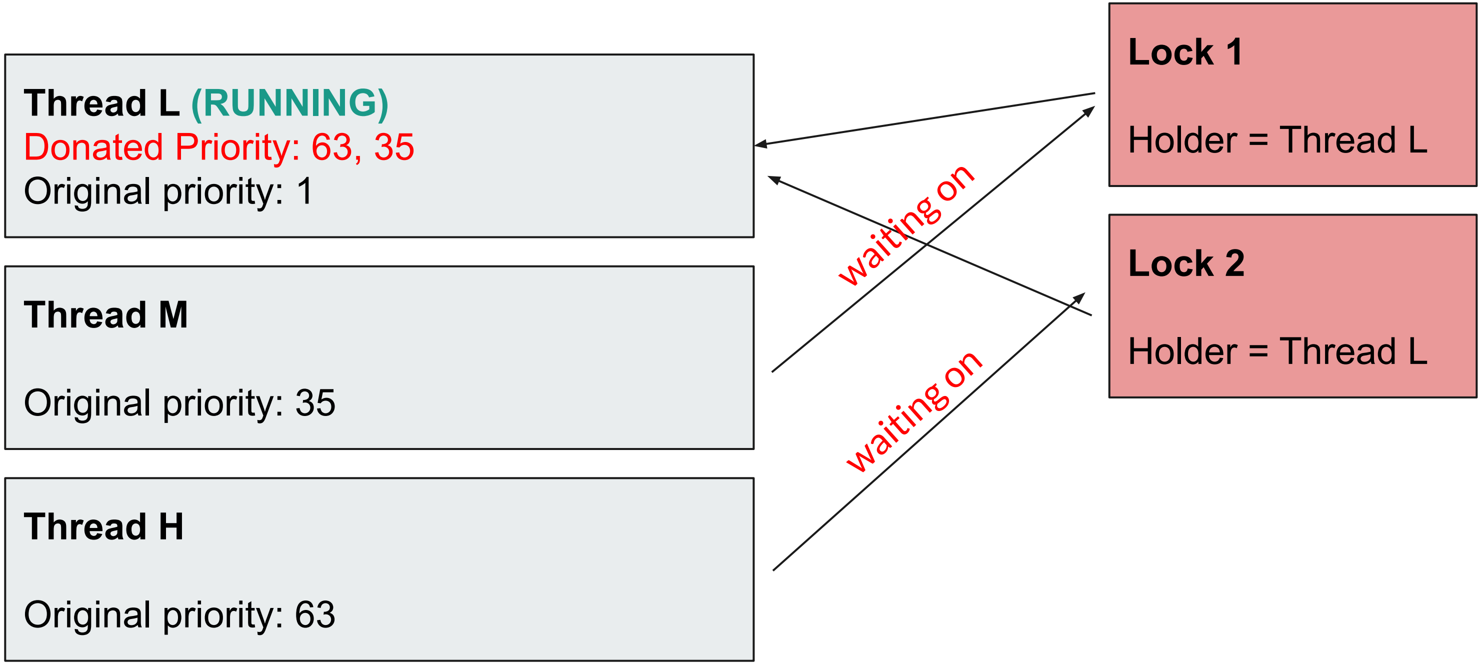
Holder = Thread L

Lock 2

Holder = Thread L

Thread H tries to acquire Lock 2, so donates its priority to Thread L

Priority Donation Example 2: Multiple Donations



Thread L (RUNNING)
Donated Priority: 63, 35
Original priority: 1

Thread M

Original priority: 35

Thread H

Original priority: 63

Lock 1

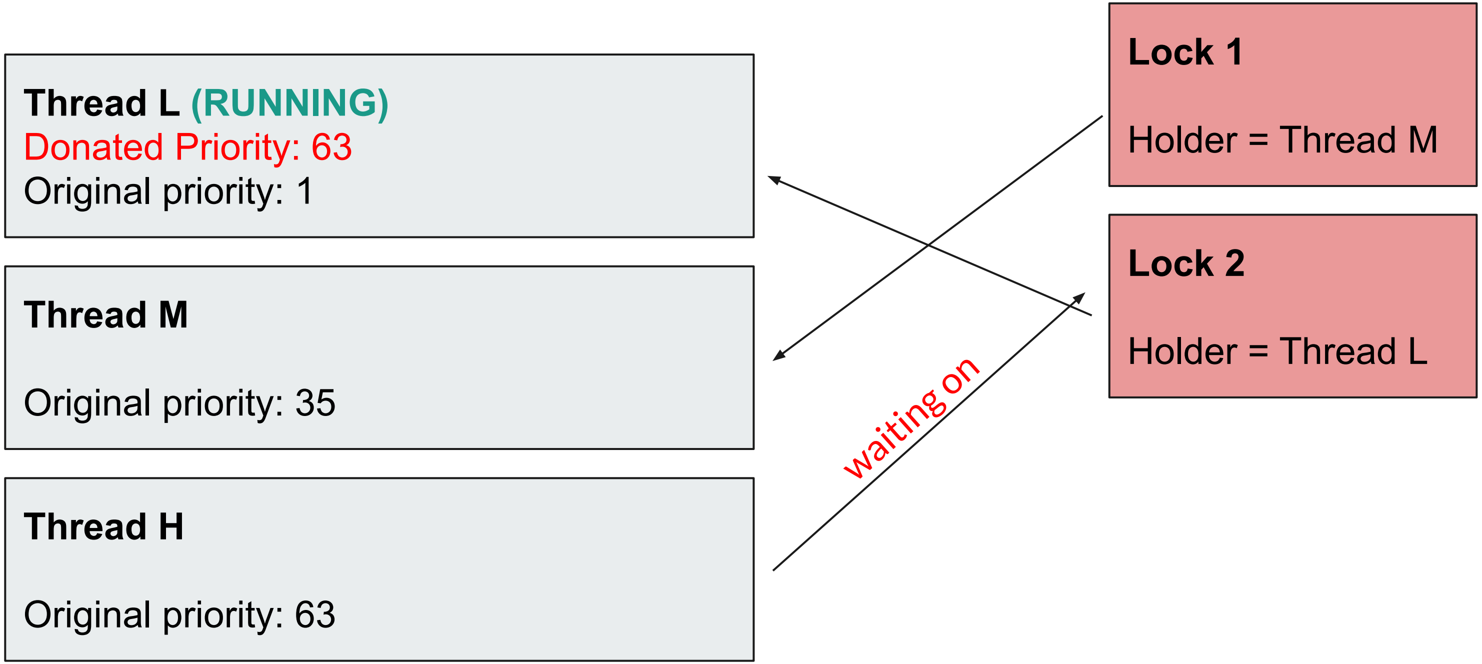
Holder = Thread L

Lock 2

Holder = Thread L

Thread L releases Lock 1 and gives back its donation

Priority Donation Example 2: Multiple Donations



Thread L (RUNNING)
Donated Priority: 63
Original priority: 1

Thread M

Original priority: 35

Thread H

Original priority: 63

Lock 1

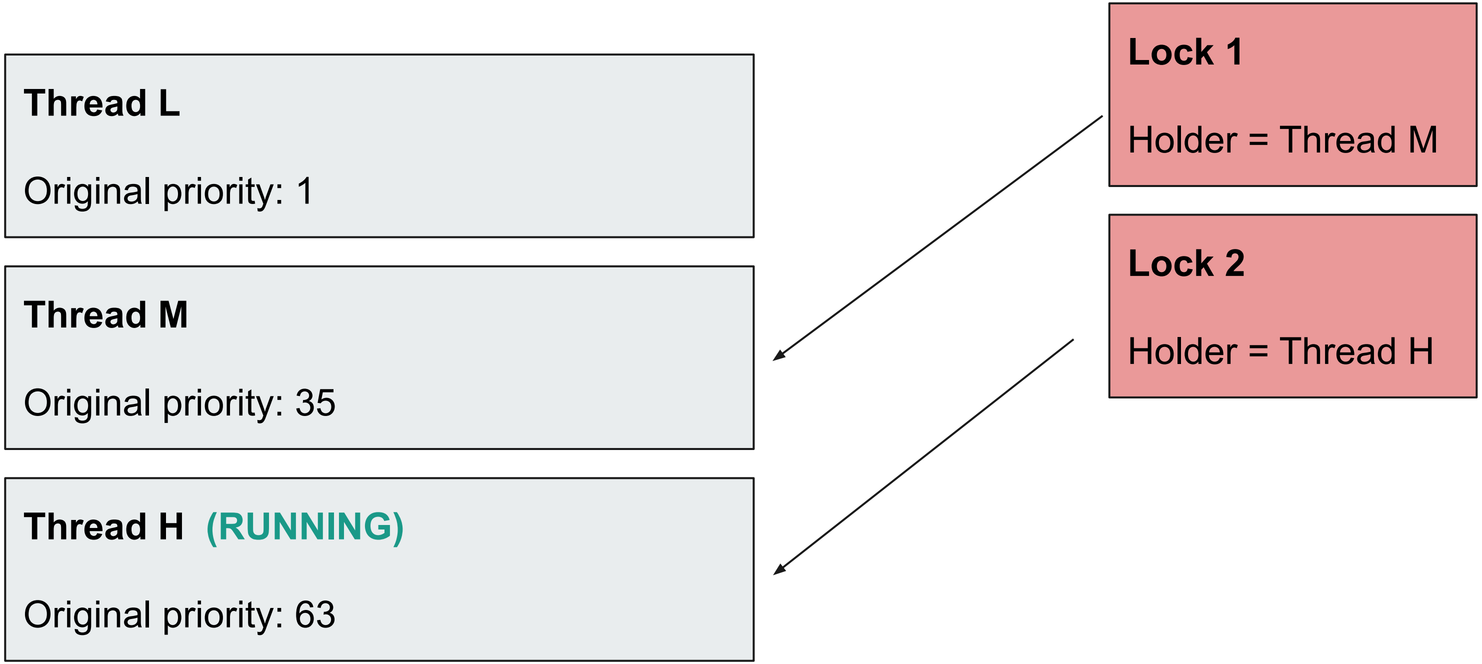
Holder = Thread M

Lock 2

Holder = Thread L

Thread L releases Lock 2 and gives back its donation

Priority Donation Example 2: Multiple Donations



Thread L

Original priority: 1

Thread M

Original priority: 35

Thread H (RUNNING)

Original priority: 63

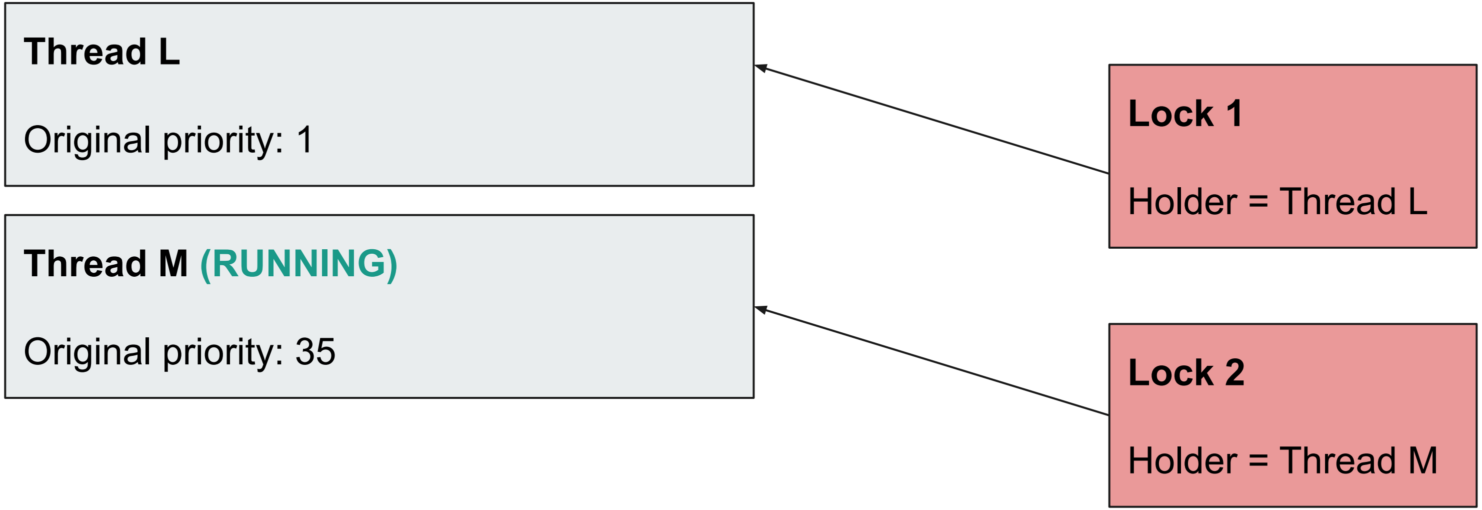
Lock 1

Holder = Thread M

Lock 2

Holder = Thread H

Priority Donation Example 3: Nested Donations



Thread L

Original priority: 1

Thread M (RUNNING)

Original priority: 35

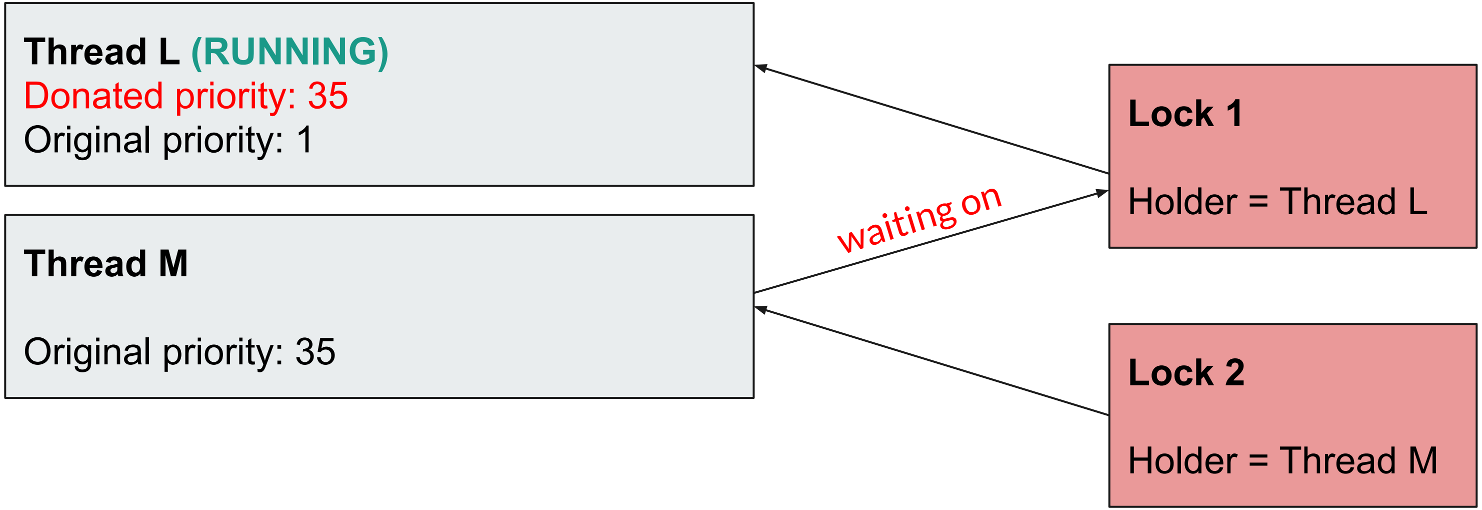
Lock 1

Holder = Thread L

Lock 2

Holder = Thread M

Priority Donation Example 3: Nested Donations



Thread L **(RUNNING)**
Donated priority: 35
Original priority: 1

Thread M

Original priority: 35

Lock 1

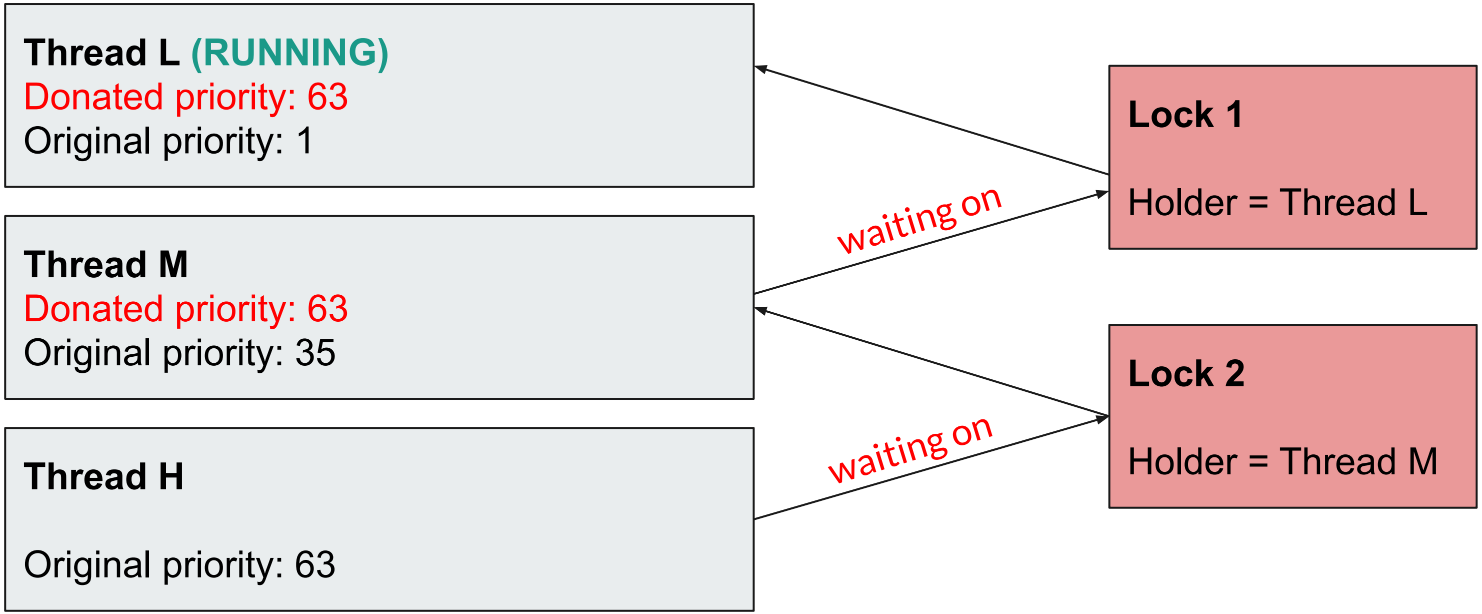
Holder = Thread L

Lock 2

Holder = Thread M

Note: You may impose a reasonable limit on depth of nested priority donation, such as 8 levels

Priority Donation Example 3: Nested Donations



Thread L (RUNNING)
Donated priority: 63
Original priority: 1

Thread M
Donated priority: 63
Original priority: 35

Thread H
Original priority: 63

Lock 1
Holder = Thread L

Lock 2
Holder = Thread M

Priority Scheduling: Key Questions

- What data structure will you use to track priority donations?
- When are priority donations given, when are they returned?
- How will you ensure that the highest priority thread waiting on a synchronization primitive is woken up first?

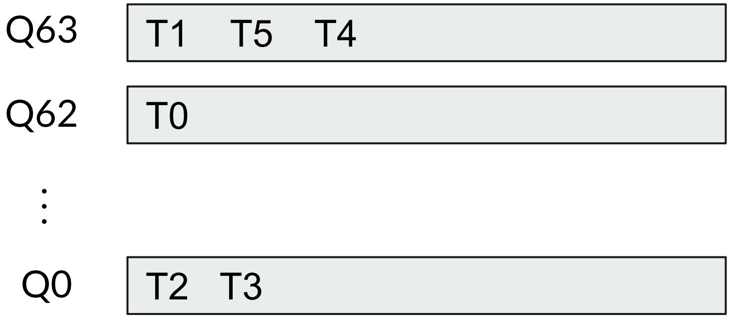
Questions?

Requirement 3: Multi-level feedback queue scheduler

MLFS: Overview

- Scheduler chooses a thread from the highest-priority non-empty queue

- If the highest-priority queue contains multiple threads, then they run in "round robin" order



Q63 T1 T5 T4

Q62 T0

⋮

Q0 T2 T3

MLFS: Overview

- Thread priority is dynamically determined by the scheduler using a formula given below, recalculated once every fourth timer tick for every thread for which recent_cpu has changed
  - *priority = PRI_MAX - (recent_cpu / 4) - (nice * 2)*
  - Detailed explanations of how/when to calculate *recent_cpu* and *nice* are here: **B. 4.4BSD Scheduler**
  - **Also covered in Week 2, Lecture 2 - Scheduling**
  - No priority donation
- **We recommend that you have the priority scheduler working, except possibly for priority donation, before you start work on the advanced scheduler**

MLFS: Fixed Point Math

- Calculations for the MLFS e.g. *recent_cpu* and *load_avg* involve both integers and real numbers
- Floating-point arithmetic in the kernel would complicate and slow the kernel → Pintos and other real kernels do not support it → calculations on real quantities must be simulated using integers
- **You will have to carefully implement fixed point arithmetic to perform calculations for your advanced scheduler**

Questions?

Requirement 4: Design Doc

Design Doc: Overview

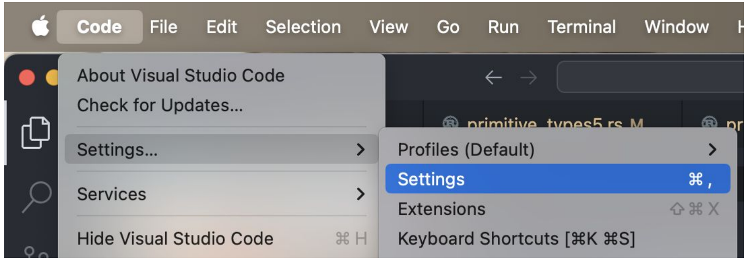
- Not like previous CS classes you have taken - DESIGN IS 50% OF YOUR GRADE
- Read through the design doc BEFORE writing any code – it will help you understand the important design problems you need to solve
- I would recommend writing your design doc for each section of the project BEFORE writing any code to ensure you are meeting design criteria, then update it as your design evolves!

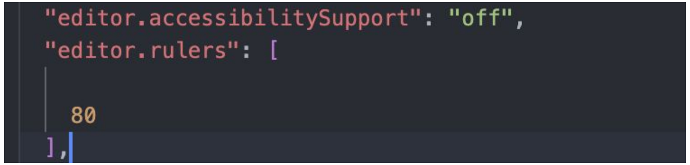
Design Advice

Make sure each line is fewer than 80 characters

In VS Code Code >> Settings >> Settings

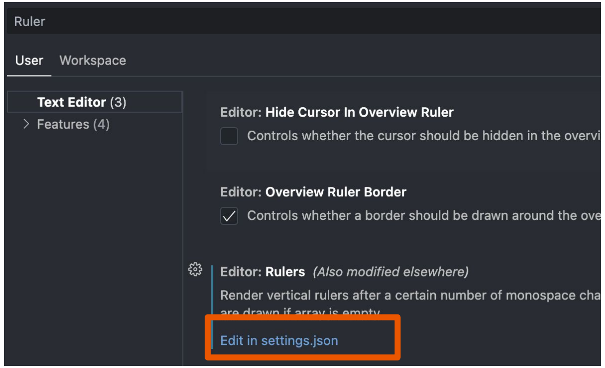
User search bar and type in “Ruler” and click on “Edit in settings.json”, add value 80!





```
"editor.accessibilitySupport": "off",
"editor.rulers": [

  80
],
```



Editor: Hide Cursor In Overview Ruler
□ Controls whether the cursor should be hidden in the overvi

Editor: Overview Ruler Border
☑ Controls whether a border should be drawn around the ove

Editor: Rulers *(Also modified elsewhere)*
Render vertical rulers after a certain number of monospace cha
are drawn if array is empty.

Edit in settings.json

Design Advice

Meet up as a team and figure out design together

Spend as much time working all together as a team as possible!

DO NOT RUSH TO START CODING - read documentation, codebase and design doc thoroughly

Key things to understand before:

- How do timer interrupts work?
- How does sleeping work now?
- How does synchronization work now?

Integrate code changes early and often (small, incremental commits)

Conform to style of the given code - guidelines

Good Luck!