

CS318 Pintos Project

Lab1 Overview

William Wang (xwill@bu.edu)

Administrivia

- Lab 1 deadline: Thursday 10/13 11:59 pm EDT
 - Estimated time: ~ 50 person-hours per group
- Late policy: 6-day (144hrs) tokens each team in total
 - Use it wisely. Recommend reserving them for later labs
 - Fill out the late hours form **before the deadline** (link in lab 1 description web page)
 - Fill out the late hours form **again** to indicate you are done (don't forget to)

Outline

- Background

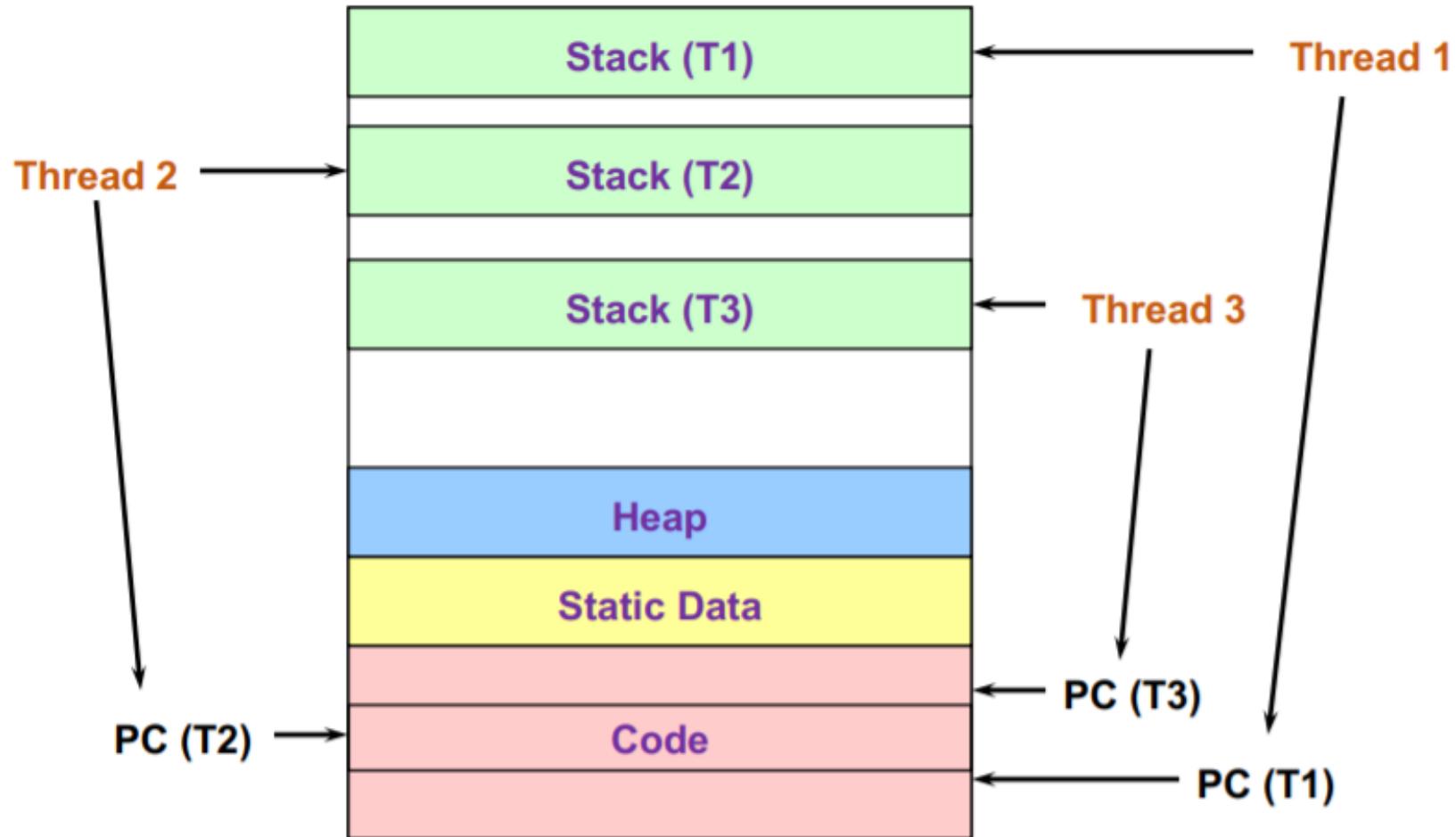


- Lab 1
- Tips

Thread: A Quick Review

- Threads are a sequential execution stream within a process
- Thread control blocks (TCB) save state of a thread
- Thread is bound to a single process
 - Shared heap, static data and code
 - Dedicated stack

Thread In A Process: A Quick Review



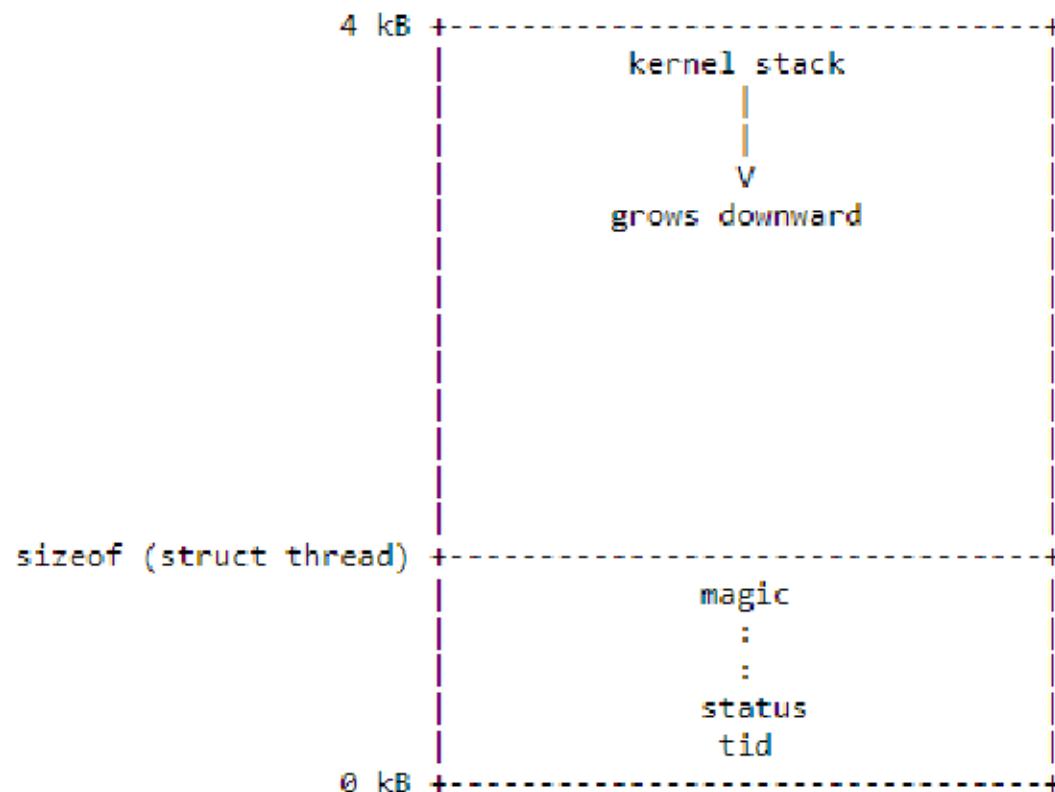
Thread In Pintos

- Pintos thread struct
 - Represents a thread or a user process
 - Will be **modified** in lab 1

```
struct thread
{
    tid_t tid;                      /* Thread identifier. */
    enum thread_status status;      /* Thread state. */
    char name[16];                  /* Name (for debugging purposes). */
    uint8_t *stack;                 /* Saved stack pointer. */
    int priority;                   /* Priority. */
    struct list_elem allelem;       /* List element for all threads list. */
    struct list_elem elem;          /* List element. */
    unsigned magic;                 /* Detects stack overflow. */
};
```

The Thread Stack In Pintos

- A small, fixed-size execution stack (4 kB)



Thread System

- `thread_create()` starts new threads
 - Added to *all_list* and *ready_list*
- Periodically, the timer interrupt fires
 - Current thread stops running
 - Timer interrupt calls `schedule()`

Thread Schedule

```
static void schedule (void) {  
    struct thread *cur = running_thread ();  
    struct thread *next = next_thread_to_run ();  
    struct thread *prev = NULL;  
  
    if (cur != next) prev = switch_threads (cur, next);  
    thread_schedule_tail (prev);  
}
```

Choose the next
running thread from
the ready_list

Context switch

Idle Thread

- There is always one thread running in the system
- Known as the **idle thread**
 - Executes when there are no other threads to run

Important Thread Functions(1)

- **thread_tick (void):**
 - Called by the **timer interrupt** at each timer tick
- **thread_block ()**
- **thread_unblock (struct thread **thread*)**
- **thread_current (void)**
 - Returns the running thread

Important Thread Functions(2)

- **thread_yield** (void)
 - Yields the CPU to the scheduler
- **thread_FOREACH** (thread_action_func **action*, void **aux*)
 - Iterates over all threads t and invokes action on each

In lab 1, you need to **use** all these functions and **modify** most of them

Race condition

- Concurrent threads read/modified/written a **shared variable**

`x` is a global variable initialized to 0

Thread 1

```
void foo()
{
    x++;
}
```

Thread 2

```
void bar()
{
    x--;
}
```

- After thread 1 and thread 2 finishes, `x` could be
 - 0, 1, -1

Synchronization

- How to prevent race condition?
 - Serializing access to shared resource
- **Disabling interrupts:** Turns off thread preemption, so only one thread can run
- **Synchronization primitives:** in threads/synch.h
 - Semaphores
 - Locks
 - Condition variables

Synchronization In Pintos: Disabling Interrupts

- The crudest way to do synchronization is to disable interrupts
 - Prevent the CPU from responding to interrupts
 - No other thread will preempt the running thread
 - Most of times, you should **not** use it directly
 - use synchronization primitives instead
- Reference:
<https://yigonghu.github.io/EC440/fall25/projects/reference/synchronization>

Interrupts APIs

- **intr_get_level (void)**
 - Returns the current interrupt state
- **intr_set_level (enum intr_level *level*)**
 - Turns interrupts on or off according to *level*
 - Returns the previous interrupt state
- **intr_enable (void)**
 - Turns interrupts on. Returns the previous interrupt state
- **intr_disable (void)**
 - Turns interrupts off. Returns the previous interrupt state

Synchronization In Pintos: Semaphores

- A *semaphore* is a nonnegative integer with two operators
 - "Down": wait for the value to become positive, then decrement it
 - "Up" : increment the value and wake up one waiting thread
- A semaphore initialized to 0
 - Waiting for an event that will happen exactly once
- A semaphore initialized to 1
 - Controlling access to a resource

Semaphore APIs

- Defined in “threads/synch.h”
- **void sema_init (struct semaphore *sema, unsigned value)**
 - Initializes *sema* with the given initial *value*
- **void sema_down (struct semaphore *sema)**
- **void sema_up (struct semaphore *sema)**

Synchronization In Pintos: Lock

- A lock is like a semaphore with an initial value of 1
 - “release” operation \approx “up”
 - “acquire” operation \approx “down”
 - Only the owner of lock can release the lock

Lock APIs

- void **lock_init** (struct lock *lock)
- void **lock_acquire** (struct lock *lock)
- void **lock_release** (struct lock *lock)

Synchronization In Pintos: Monitor

- A monitor = data + lock (monitor lock) + condition variables
 - Acquiring the monitor lock
 - Waiting for a condition to become true and release the lock
 - If the condition is true, wake up one waiter or “broadcast”
 - Access the data

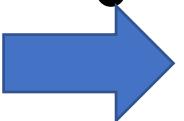
Condition Variable APIs

- **void cond_init (struct condition **cond*)**
- **void cond_wait (struct condition **cond*, struct lock **lock*)**
 - Atomically releases *lock* and waits for *cond* to be signaled
 - When it returns, lock is acquired again
- **void cond_signal (struct condition **cond*, struct lock **lock*)**
- **void cond_broadcast (struct condition **cond*, struct lock *)**

Outline

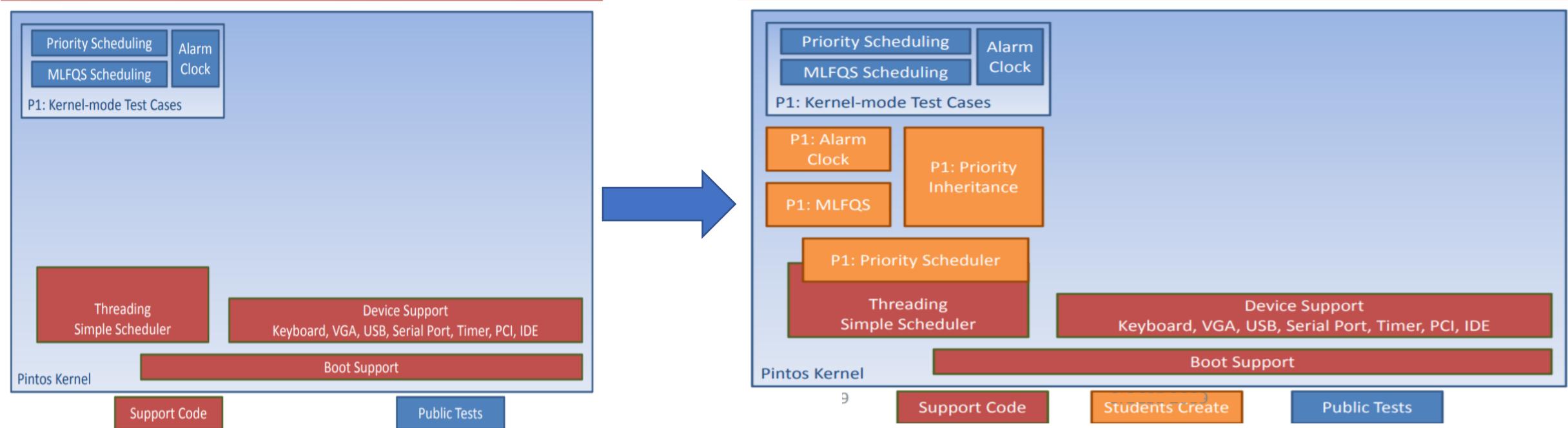
- Background

- Lab 1



- Tips

The Goal of Lab 1



The Goal of Lab 1

- Reimplement the timer_sleep() function
 - Avoid busy waiting
- Implement priority scheduling
 - High priority threads execute before low priority
- Implement a multilevel feedback queue scheduler
 - Reduce the average response time for running jobs

Alarm Clock

- Reimplement *timer_sleep(num_ticks)*

```
void timer_sleep (int64_t ticks) {  
    int64_t start = timer_ticks();
```

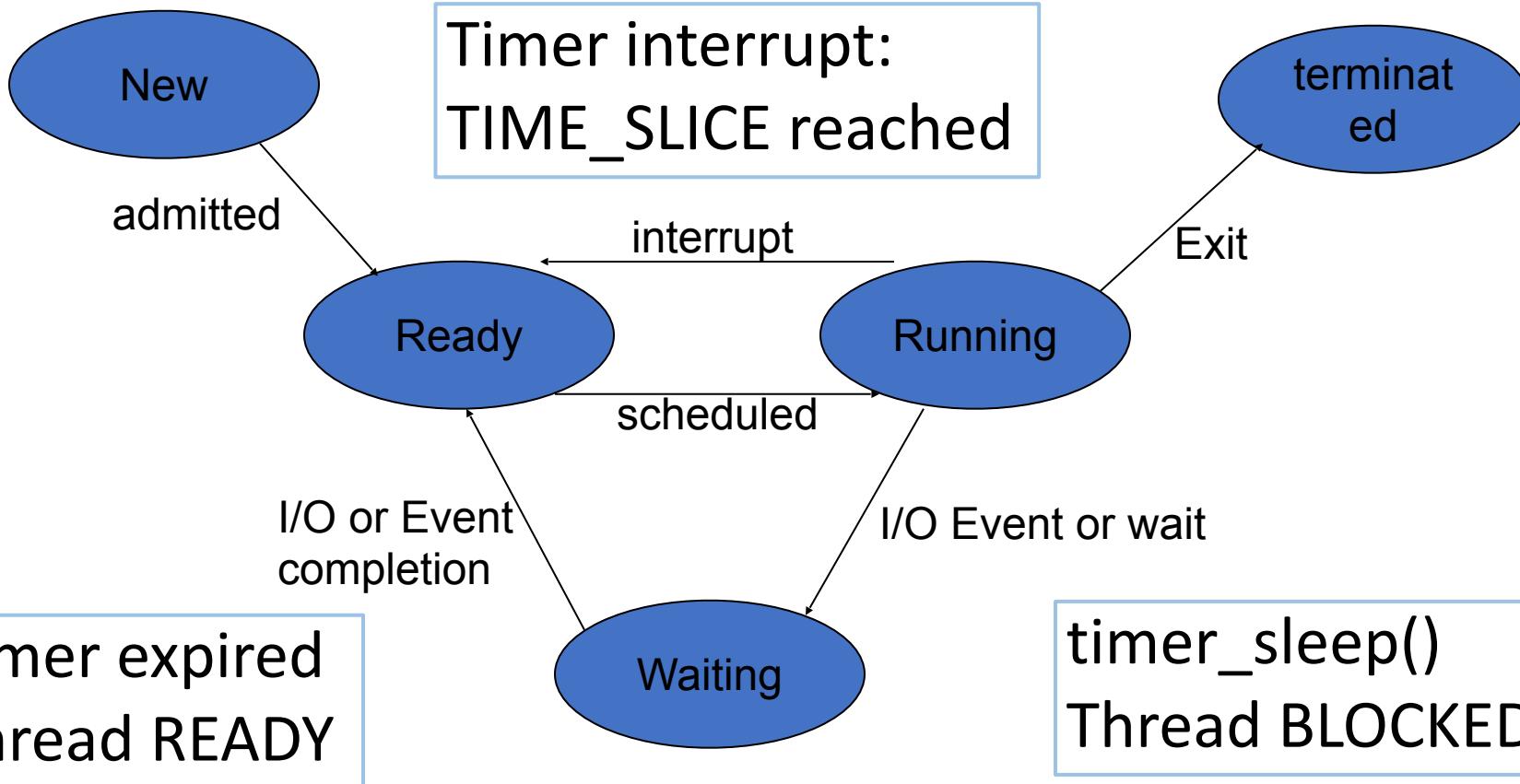
your implementation

}

Avoid busy wait

Busy Wait

Alarm Clock



Timer Interrupt

- PIT (Programmable Interval Timer)
 - E.g. Intel 8254
 - Resides on the motherboard
 - External microcontroller (“external” as to CPU)
 - Generate timer interrupt pulse at a fixed rate
 - Can’t be simulated with sequential instruction execution
 - OS kernel sets up timer at startup
 - “event-driven”

Timer Interrupt

```
/* Sets up the timer to interrupt TIMER_FREQ times per second,
   and registers the corresponding interrupt. */
void
timer_init (void)
{
    pit_configure_channel (0, 2, TIMER_FREQ);
    intr_register_ext (0x20, timer_interrupt, "8254 Timer");
}
/* Timer interrupt handler. */
static void
timer_interrupt (struct intr_frame *args UNUSED)
{
    ticks++;
    thread_tick ();
}
```

Pintos:

timer.c

Timer Interrupt

- Interrupt state when entering *timer_interrupt*
 - **INTR_OFF**
 - An external interrupt (typically generated by hardware)
 - Other kind of interrupt may choose to keep INTR_ON
 - E.g.: syscall, most exceptions
- Interrupt state after return (after *iret* in *intr-stub.S*)
 - **INTR_ON**

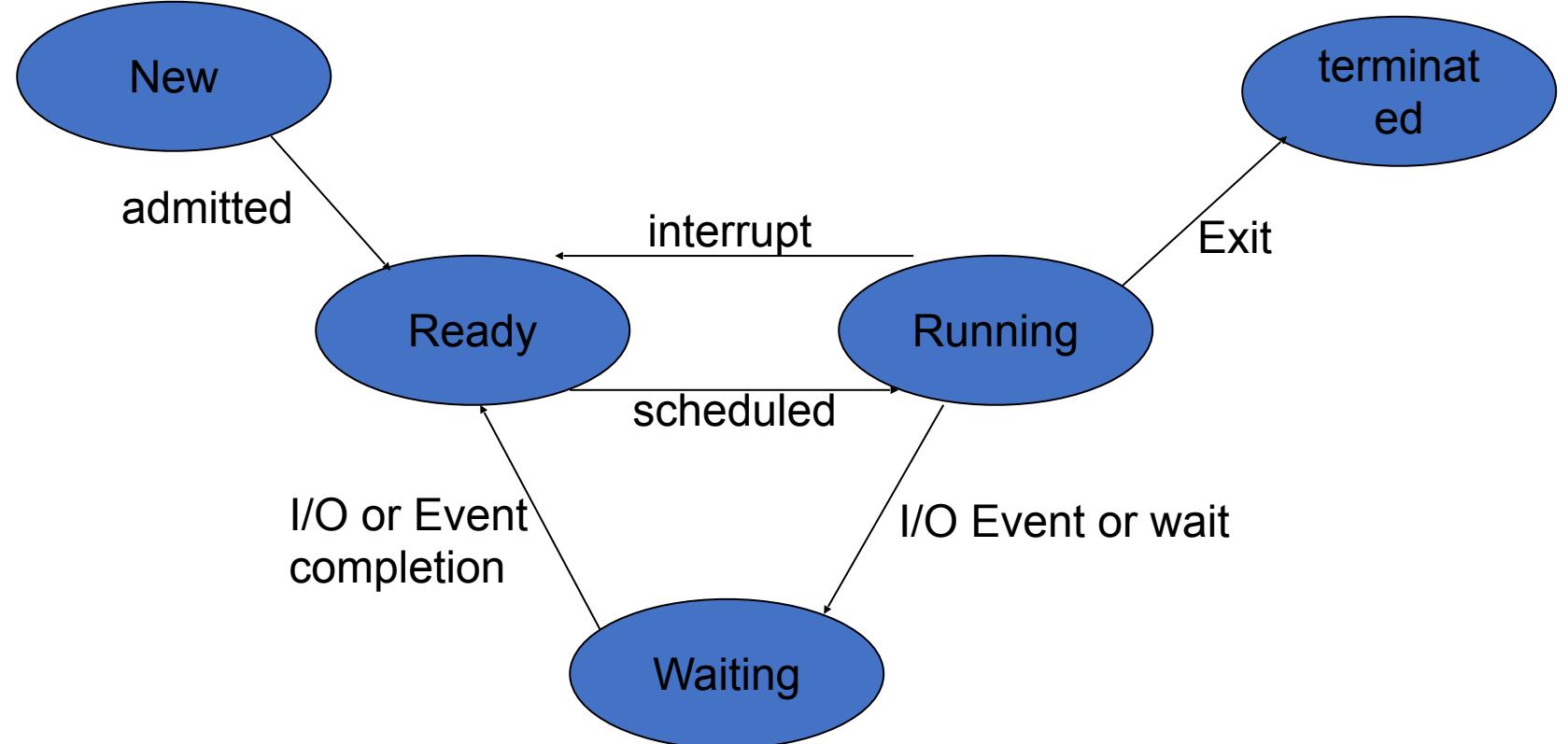
Design Hints for Alarm Clock

- How to sleep a thread?
 - Leveraging existing APIs that can sleep the thread
 - Looking at the synchronization slides
- How to find a sleeping thread?
- How to wake up a sleep thread?
 - Look at the timer_interrupt handler in Pintos

Scheduling

- Scheduling Policy

- FIFO
- Priority
- MLFQ

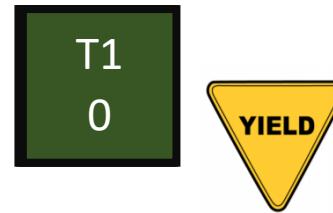


Priority Scheduling

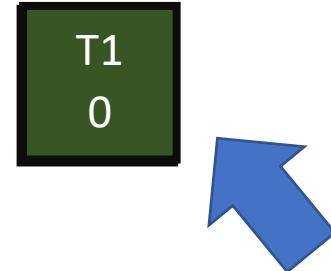
- Implement priority scheduling in Pintos
 - Thread with highest priority is always running
 - Highest priority waiting thread should be awoken first
 - 0-63 priorities, higher number represents higher priority
 - E.g., priority 6 > priority 2
- The challenging part of priority scheduling
 - Priority donation, multiple donations and nest donation

Priority Scheduling Examples

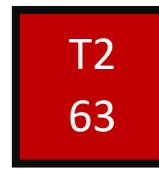
Running thread



Ready queue



Wait queue



CPU scheduler

Thread 1
Priority 0

Running

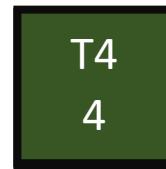
Thread 2
Priority 63

Running



Scenario 1 : Acquiring Lock

Running thread



Ready queue



Largest Priority



CPU scheduler

Thread 1
Priority 10

Running



Block

Thread 4
Priority 4

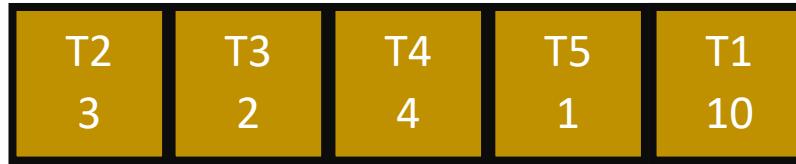
Running

Scenario 2 : Unlock Thread

Running thread



Ready queue



Wait queue



CPU scheduler

Thread 1
Priority 10

Running

block

Thread 7
Priority 20

Unblock

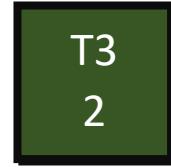


Running

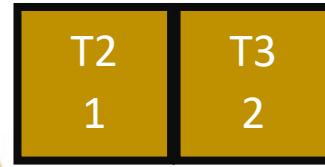
Largest Priority

Problematic Scenario: Priority Inversion

Running thread



Ready queue



Wait queue



Degrade to priority 1

CPU scheduler

Thread 2
Priority 1



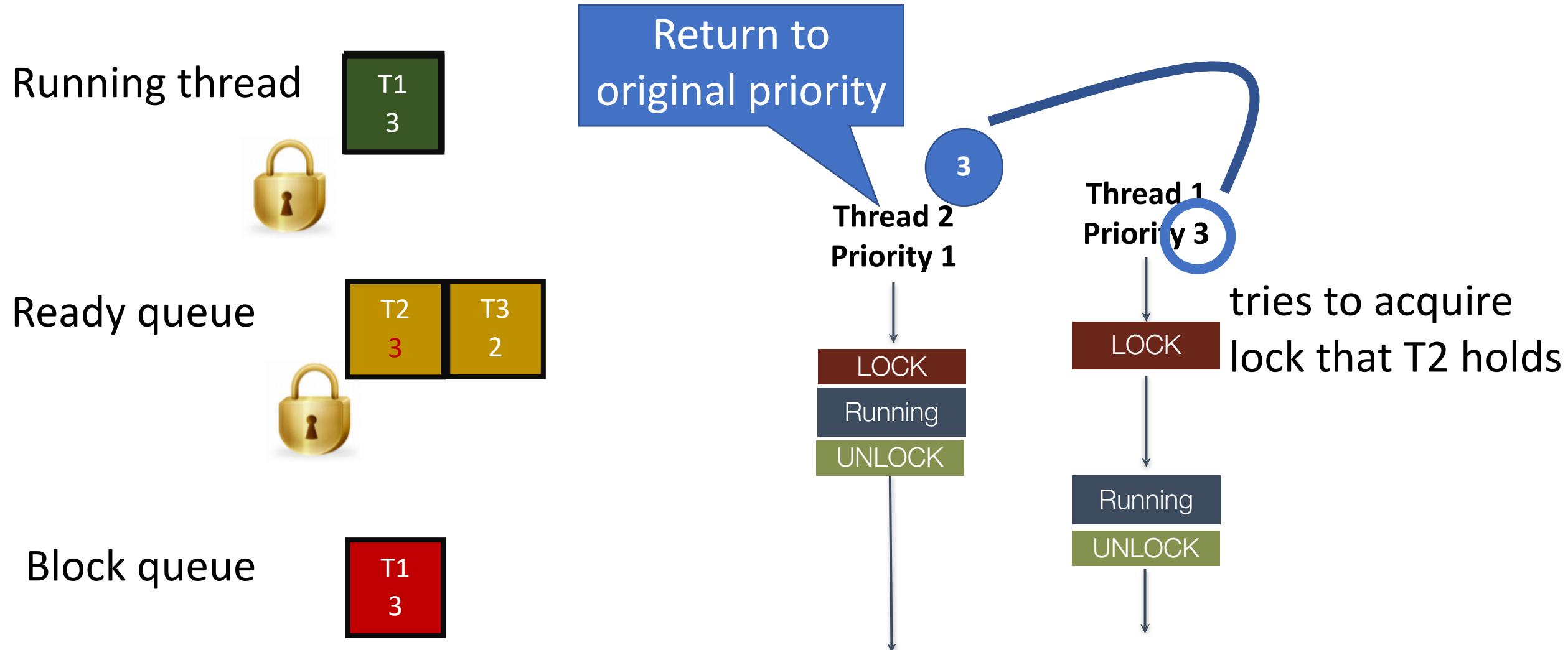
Thread 1
Priority 3

Running

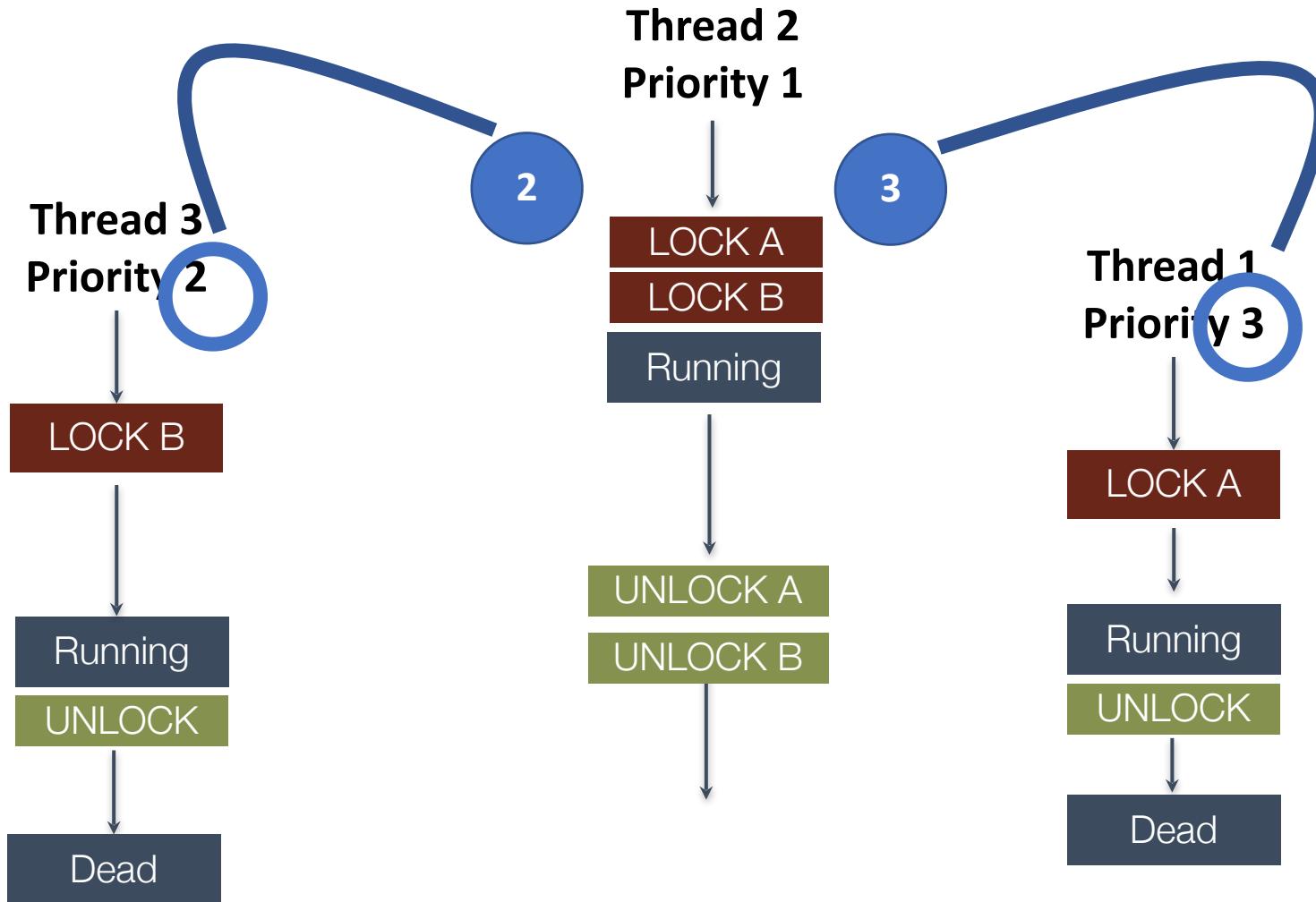
LOCK



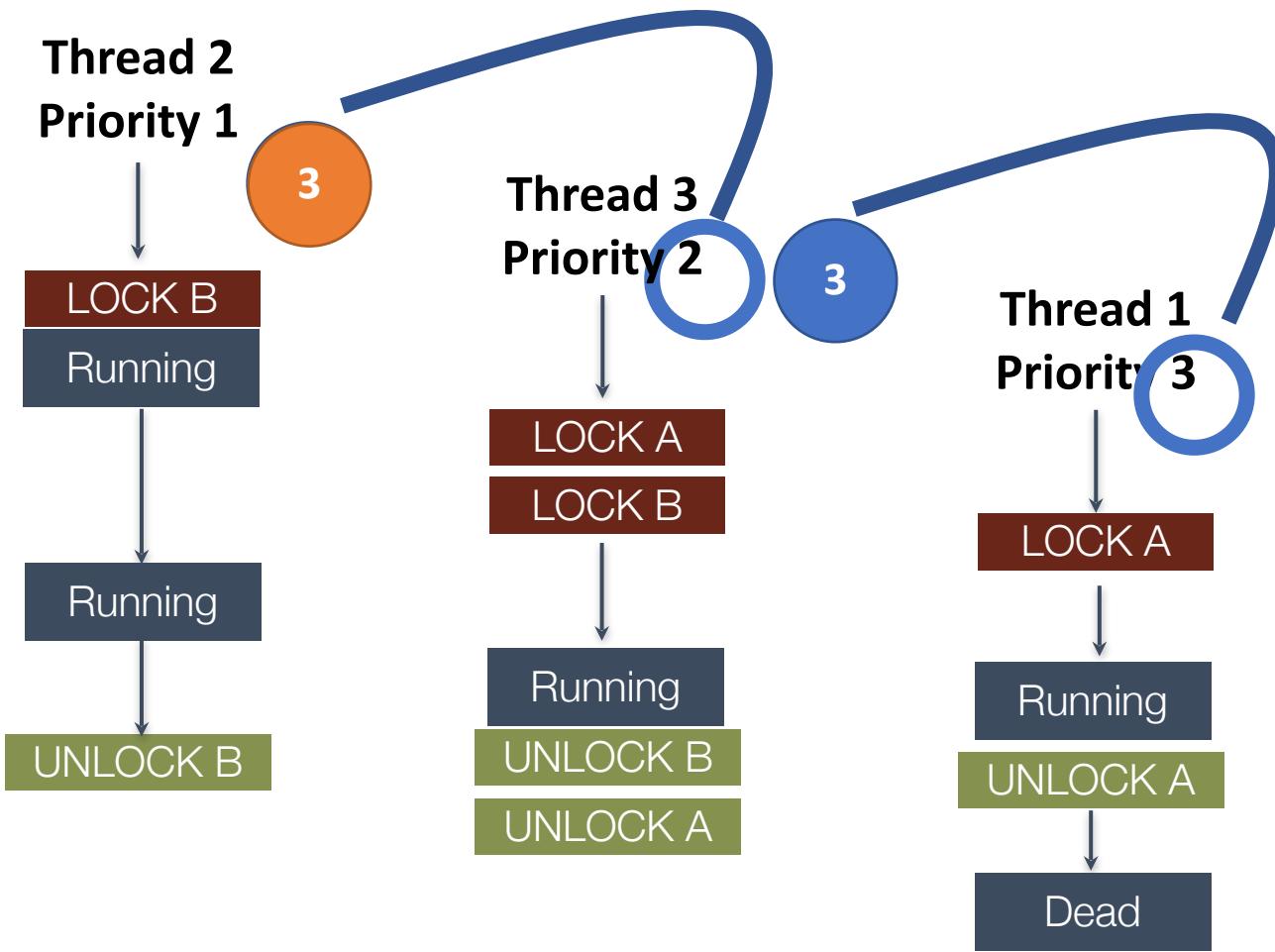
Priority Donation



Multiple Donation



Nested Donation



Advanced Scheduler

- BSD scheduler computes thread CPU usage statistics to calculate thread priorities
 - `thread_set_priority` should be ignored in this mode
- No priority donation
- **-mlfqs** kernel option:
 - Choose a scheduling algorithm policy at Pintos startup time
- Implement a simple fixed-point arithmetic library

Advanced Scheduler - Niceness

- Thread priority is dynamically determined by the scheduler using a formula
- Niceness
 - 0 -> does not affect thread priority
 - >0 -> decreases the priority of a thread (max of 20)
 - <0 -> tends to take away CPU time from other threads (min of -20)

Niceness API

- `int thread_get_nice (void)`
 - Returns the current thread's nice value
- `void thread_set_nice (int new_nice)`
 - Sets the current thread's nice value to `new_nice`
 - Recalculates the thread's priority

Advanced Scheduler - Calculate Priority

- Priorities range from 0 to 63
- Calculated every 4rth
 - For every thread where `recent_cpu` has changed

$$priority = PRI_MAX - (recent_cpu / 4) - (nice * 2),$$

$$recent_cpu = (2 * load_avg) / (2 * load_avg + 1) * recent_cpu + nice.$$

$$load_avg = (59/60) * load_avg + (1/60) * ready_threads.$$

Fixed-Point Real Arithmetic

- `recent_cpu` and `load_avg` are real numbers
- You need to implement a library for fixed-Point Arithmetic
- Reference

<https://yigonghu.github.io/EC440/fall25/projects/reference/bsd#6-fixed-point-real-arithmetic>

Outline

- Background
- Lab 1
- Tips

Git

- Distributed version management
- With git, you can
 - Easily track the changes
 - Recover files in the past
 - Concurrently develop the project and safely merge parts

Git – basic operations

- Locations
 - Working tree
 - Index (or called “staging area”)
 - Local repo
 - Remote repo
- Sync files between different locations
 - Clone
 - Add
 - Commit
 - Push

git clone

- Copy the whole remote repo to the local
- *git clone <remote repo> [<local directory>]*
- \$ *git clone <https://github.com/jhu-cs318-fall22/pintos-lab-xyz>*

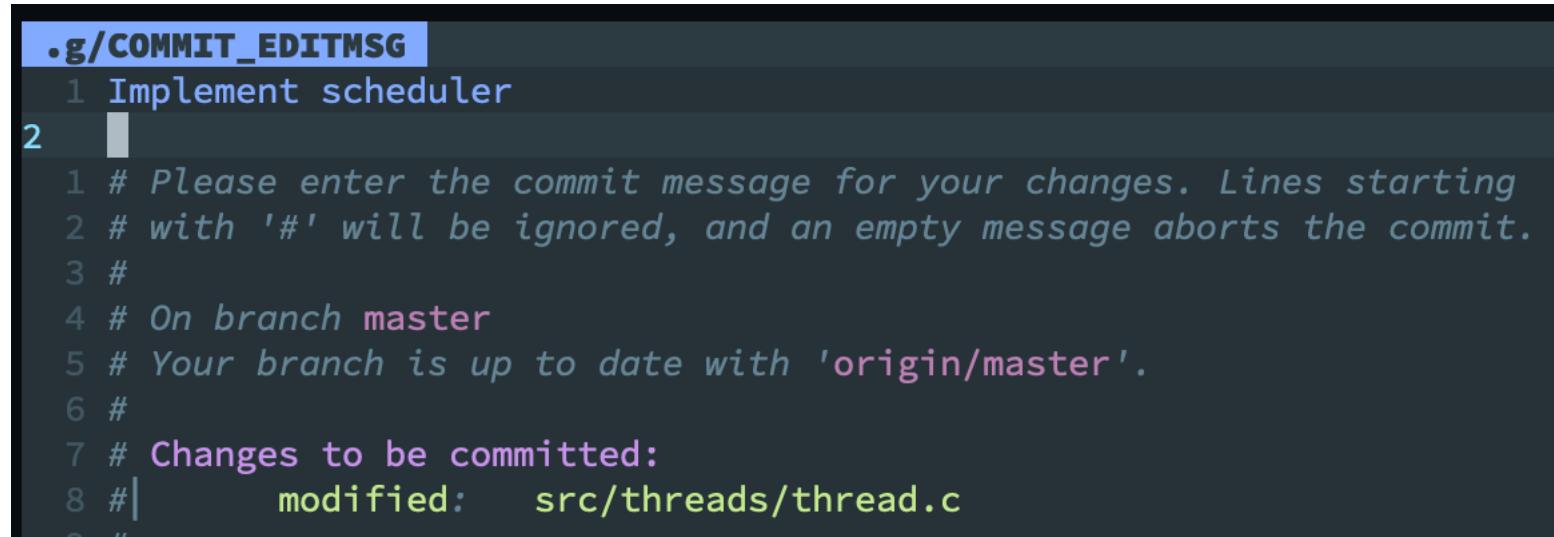
```
> git clone https://github.com/jhu-cs318/pintos.git
Cloning into 'pintos'...
remote: Enumerating objects: 716, done.
remote: Counting objects: 100% (52/52), done.
remote: Compressing objects: 100% (33/33), done.
remote: Total 716 (delta 21), reused 32 (delta 19), pack-reused 664
Receiving objects: 100% (716/716), 362.48 KiB | 4.65 MiB/s, done.
Resolving deltas: 100% (123/123), done.
```

git add

- Copy files from workspace to the index
- ***git add <file path...>***
- ***\$ git add src/threads/thread.c***

git commit

- Copy files from index to the local repo
- ***git commit***
- *Edit the commit message in the opened editor and save it*



The screenshot shows a terminal window with the title bar ".g/COMMIT_EDITMSG". The main area contains the following text:

```
1 Implement scheduler
2
1 # Please enter the commit message for your changes. Lines starting
2 # with '#' will be ignored, and an empty message aborts the commit.
3 #
4 # On branch master
5 # Your branch is up to date with 'origin/master'.
6 #
7 # Changes to be committed:
8 #|     modified:   src/threads/thread.c
9 #"
```

git push

- Update commits on a branch from local repo to the remote
- *git push [<remote repo>] [<branch name>]*
- \$ *git push*

```
> git push
Enumerating objects: 9, done.
Counting objects: 100% (9/9), done.
Delta compression using up to 8 threads
Compressing objects: 100% (5/5), done.
Writing objects: 100% (5/5), 454 bytes | 454.00 KiB/s, done.
Total 5 (delta 3), reused 0 (delta 0), pack-reused 0
remote: Resolving deltas: 100% (3/3), completed with 3 local objects.
To https://github.com/superobertking/pintos-showcase.git
    ab69995..b4fffb7  master -> master
```

GitHub

- Branches
- Merge
- Pull request

Demo

Git & GitHub

- More...
 - Revert changes
 - Cherry-picking
 - Edit commit history
 - Squash commits
 - ...

Coding Tips

- Read the document thoroughly & carefully before coding
 - Understand Pintos thread system and synchronization
 - Read base code also helps understanding
- Read the design document template first and work on it as you write code and debug
- Don't code and design together
 - Have a design/solution in your mind first, then starting coding

Debugging Suggestion

- Debug with gdb, not with printf
 - Get used to debugger
 - printf internally uses locks and changes interrupt state
 - Can mess up with scheduling
- Look at the test code during the debugging
- When encountering kernel panic or assertion violations
 - Figure out the call stack first
 - Double check the workflow by GDB

Grading

- To receive full credit
 - Working, **well designed** code that completes all tests(**70%**)
 - A complete, **well written** design document(**30%**)
- Your coding style and design matters
- All code will be scanned by plagiarism detection software

Thanks for attending this session!