Project 1: Threads

Preliminaries

> Fill in your name and email address.

Yixin Yang (杨艺欣) yangyixin@stu.pku.edu.cn

> If you have any preliminary comments on your submission, notes for the TAs, please give them here.

> Please cite any offline or online sources you consulted while preparing your submission, other than the Pintos documentation, course text, lecture notes, and course staff.

Alarm Clock

DATA STRUCTURES

> A1: Copy here the declaration of each new or changed struct or struct member, global or static variable, typedef, or enumeration. Identify the purpose of each in 25 words or less.

```
/** src/devices/timer.c */
/** List of threads sleeping */
static struct list sleep_list;

/** src/threads/thread.h */
struct thread
  {
    /** ... */
    struct list_elem sleep_elem;    /* List element in sleep_list */
    int64_t sleep_ticks;            /* Ticks to sleep */
  }

int64_t thread_get_sleep_ticks (void);    /* Get current thread's sleep ticks */
void thread_set_sleep_ticks (int64_t ticks);    /* Set current thread's sleep ticks */
```

ALGORITHMS

> A2: Briefly describe what happens in a call to timer_sleep(),
> including the effects of the timer interrupt handler.

In `timer_sleep()`:

1. Check if the argument is valid (if not positive, return immediately).
2. Disable interrupts.
3. Set the current thread's sleep ticks, add it to the sleep list, and block it.
4. Enable interrupts.

In `timer_interrupt()`:

1. Update global ticks. Call `thread_tick()`.
2. Disable interrupts.
3. For each thread in the sleep list, update its sleep ticks. If a thread's sleep ticks is 0, remove it from the sleep list and unblock it (if it has a higher priority than the current thread, call `intr_yield_on_return`).
4. Enable interrupts.

> A3: What steps are taken to minimize the amount of time spent in
> the timer interrupt handler?

Do only necessary work. Use `thread_block()` and `thread_unblock ()` instead of other higher-level synchronization primitives.

SYNCHRONIZATION

> A4: How are race conditions avoided when multiple threads call
> timer_sleep() simultaneously?

By disabling interrupts before modifying global data structures (e.g. `sleep_list`) and enabling interrupts after modifying them.

> A5: How are race conditions avoided when a timer interrupt occurs
> during a call to timer_sleep()?

By disabling interrupts as well.

RATIONALE

> A6: Why did you choose this design? In what ways is it superior to
> another design you considered?

Compared to using higher-level synchronization primitives, directly disabling interrupts and calling `thread_block()` and `thread_unblock()` is simpler and more efficient. Also, it reduced the dependency on synchronization primitives.

Using a `list` to contain all the sleeping thread is the simplest way since `list` is implemented already and efficient enough.

We create a `sleep_elem` in the `thread` struct to link the thread to the `sleep_list`, this is much easier to implement. Compared to using the existing `elem`, this will not add much overhead but make the code much easier to write and maintain.

Priority Scheduling

DATA STRUCTURES

> B1: Copy here the declaration of each new or changed struct or struct member, global or static variable, typedef, or enumeration. Identify the purpose of each in 25 words or less.

```
/** src/threads/thread.h */
struct thread
  {
    /** ... */
    int priority;                       /**< Priority. */
    int base_priority;                  /**< Base priority. */
    int donated_priority;               /**< Donated priority. */
    struct lock *waiting_lock;          /**< Lock waiting for. */
    struct list locks;                  /**< List of locks held. */
  }

/** src/threads/synch.h */
struct lock
  {
    /** ... */
    int max_priority;           /**< Maximum priority of all threads
                                     waiting on this lock. */
    struct list_elem elem;      /**< List element for locks list. */
  }

/** src/threads/synch.c */
#define MAX_DONA_DEPTH 8    /* Maximum depth of priority donation */
#define MAX(a, b) ((a) > (b) ? (a) : (b))
/** Compare function for lock list based on max_priority */
static bool list_lock_greater(const struct list_elem *a,
                              const struct list_elem *b, void *aux UNUSED);
/** Compare function for cond->waiters based on priority */
static bool list_sema_greater(const struct list_elem *a,
                              const struct list_elem *b, void *aux UNUSED);
```

> B2: Explain the data structure used to track priority donation.
> Use ASCII art to diagram a nested donation. (Alternately, submit a .png file.)

Every thread has a `waiting_lock` field to track the lock it is waiting for, and a `priority` field to indicate its priority.

> In the submitted code, I also added a `donated_priority` field to track the priority donated to the thread. This is actually a bit redundant and can be merged into the `priority` field.

Every lock has a `max_priority` field to track the maximum priority of all threads waiting for it, as well as a `holder` field to track the thread holding it.

For example,
Thread 1 (priority 1) holds lock A (max_priority 1), thread 2 (priority 2) holds lock B (max_priority 2). If

thread 2 wants to acquire lock A, then it will donate its priority to thread 1 through lock A.

If thread 3 (priority 3) wants to acquire lock B, then it will donate its priority to thread 2 through lock B, and since now thread 2 has a higher priority, it will also donate its new priority to thread 1 through lock A.

The nested donation when thread 3 wants to acquire lock B is as follows:

```
Initially:
Thread 1 (priority 2) --holds--> Lock A (max_priority 2) <--waits for-- Thread 2 (priority 2) '
--holds--> Lock B (max_priority 2) <--waits for-- Thread 3 (priority 3)

After donation:
Thread 1 (priority 3) --holds--> Lock A (max_priority 3) <--waits for-- Thread 2 (priority 3) '
--holds--> Lock B (max_priority 3) <--waits for-- Thread 3 (priority 3)
```

ALGORITHMS

> B3: How do you ensure that the highest priority thread waiting for a lock, semaphore, or condition variable wakes up first?

In `sema_down ()`, use `list_insert_ordered ()` to insert the thread into the waiters list based on its priority. In `sema_up ()`, use `list_sort ()` to sort the waiters list after removing the thread. This ensures that the waiters list is always sorted by priority.

Locks and condition variables are build upon semaphores, so they will also maintain the order.

> B4: Describe the sequence of events when a call to lock_acquire() causes a priority donation. How is nested donation handled?

In `lock_acquire ()`:

1. Disable interrupts.
2. Set the current thread's `waiting_lock` to the lock.
3. If the lock is held, perform (nested) donation
4. Call `sema_down ()`
5. Now the thread holds the lock, update the lock's `max_priority`, `holder`, and the current thread's `locks`, `waiting_lock`.
6. Enable interrupts.

Nested donation:

- An example is given in B2.

- It is basically recursively going through the waiting chain by `lock->holder` and update the current lock's `max_priority` as well as the current holder's `priority` (and `donated_priority`).
- The depth is limited to `MAX_DONA_DEPTH`.

> B5: Describe the sequence of events when lock_release() is called on a lock that a higher-priority thread is waiting for.

1. Remove the lock from the current thread's `locks` list.
2. Update the current thread's `donated_priority` based on the remaining locks.
3. Clear the lock's `holder`.
4. Call `sema_up()`.
5. Reset the current thread's priority to `base_priority`.

SYNCHRONIZATION

> B6: Describe a potential race in thread_set_priority() and explain how your implementation avoids it. Can you use a lock to avoid this race?

In my implementation, `thread_set_priority()` only affects the `base_priority` of the thread, the real priority is calculated based on maximum of the new priority and the donated priority.

Race: If after a new priority is set, another thread donates its priority to the current thread, the just-computed priority will be stale.

However in the submitted code I did not recognize this problem so my implementation was not able to avoid this.

This can be avoided by using a lock to protect the priority setting.

RATIONALE

> B7: Why did you choose this design? In what ways is it superior to another design you considered?

It is straightforward to come up with and implement, and thus easy to understand.

It is efficient enough to use `list`. Implementing a priority queue was my first thought, but it is not necessary and easy to introduce bugs. To ensure the order, we use `list_insert_ordered()` and `list_sort()`. (Although it makes the code a little more complex since we must remember to call `list_sort()` when necessary)

Add fileds like `waiting_lock`, `locks` to the `thread` struct makes it easy to track the donation chain, and the `max_priority` field in the `lock` struct makes it easy to track the maximum priority of all threads waiting for it.

Advanced Scheduler

DATA STRUCTURES

> C1: Copy here the declaration of each new or changed struct or struct member, global or static variable, typedef, or enumeration. Identify the purpose of each in 25 words or less.

```
/** src/threads/thread.h */
struct thread
  {
    /** ... */
    int nice;           /**< Nice value. */
    fp recent_cpu;      /**< Recent CPU usage. */
  };

/** src/threads/thread.c */
static fp load_avg = int_to_fp(0);  /**< System load average. */

/** mlqfs functions */
static void update_recent_cpu(struct thread *t, void *aux UNUSED);
static void update_priority(struct thread *t, void *aux UNUSED);
static void update_load_avg(void);

/** src/threads/fixed-point.h */  /** This is a new file */
typedef int fp;    /**< Fixed-point number. */
```

ALGORITHMS

> C2: How is the way you divided the cost of scheduling between code inside and outside interrupt context likely to affect performance?

In `thread_tick ()` :

- Only call `list_sort ()` when necessary (every 4 ticks)
- Move `if (ticks & TIMER_FREQ == 0)` inside `if (ticks % 4 == 0)` , reducing branch prediction miss.
- Instead of calling `timer_ticks ()` , use global variable `ticks`
    - I tried to inline `timer_ticks ()` but the compiler seemed to ignore it.

RATIONALE

> C3: Briefly critique your design, pointing out advantages and disadvantages in your design choices. If you were to have extra

> time to work on this part of the project, how might you choose to refine or improve your design?

Advantages:

- Instead of maintaining 64 lists for 64 priorities, I only maintain one list ordered by priority. This achieves the same effect but is much simpler.
- Using functions like `update_recent_cpu ()` to calculate statistics instead of writing them directly in `thread_tick ()`. This makes the code cleaner and easier to maintain.

Disadvantages:

- Using `thread_mlqfs` everywhere in the code to distinguish between the original scheduler and the advanced scheduler. This is not very elegant.
- Even when `thread_mlqfs` set to false, `thread` struct still has `nice` and `recent_cpu` fields. This is a bit redundant.

Improvements:

- Use a more elegant way to distinguish between the two schedulers. Maybe object-oriented programming can help.
- Remove the `nice` and `recent_cpu` fields when `thread_mlqfs` is false. Maybe use a macro.

> C4: The assignment explains arithmetic for fixed-point math in detail, but it leaves it open to you to implement it. Why did you decide to implement it the way you did? If you created an abstraction layer for fixed-point math, that is, an abstract data type and/or a set of functions or macros to manipulate fixed-point numbers, why did you do so? If not, why not?

I implemented fixed-point math using macros, basically as is described in the document.

An abstraction layer will significantly improve the readability of the code, also makes it much easier to write. I typedefed `int` to `fp` to make fixed-point numbers distinguishable from common integers. I implement `fp_mul` and `fp_div` as macros to make them inline and save the overhead of function calls.

I did not implement `fp_add` and `fp_sub`, because common `+` and `-` can do the same thing (and much more readable too since it prevents lots of nesting).