Project 1: Threads

Preliminaries

> Fill in your name and email address.

Hao Wang tony.wanghao@stu.pku.edu.cn

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

The grade my submission got on my local laptop is 100%.

> 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.

Stack overflow - function prototype

Priority Donation

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.

In `devices/timer.c`:

```
static struct list sleeping_thread_list;
```

An ordered list in which the threads are all sleeping and the waking time is increasing so each time a timer interrup happens we could wake up the corresponding threads.

In `threads/thread.h`:

```
int64_t waken_time; // in the struct thread
```

The member indicating when we should wake this thread up.

ALGORITHMS

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

When `timer_sleep()` is called, the current thread is added into the `sleeping_thread_list` and then blocked. When a timer interrupt happens, the timer interrupt handler iterates through the `sleeping_thread_list` and unblock the threads that need to be waken up.

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

I introduced a new list `sleeping_thread_list` to avoid checking all the threads. The list `sleeping_thread_list` is sorted so the handler only need to find the first thread of which `waken_time` is larger than current `timer_ticks()` and returns.

SYNCHRONIZATION

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

Interrupt is turned off when the global `ready_list` is modified in order to avoid messing up the scheduler.

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

Interrupt is turned off when the `ready_list` shared between kernel and user is modified in order to avoid messing up the scheduler.

RATIONALE

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

I only considered this design. Since we need to avoid busy waiting, we need to move the tick-checking part to where `tick++` really happens, which is when timer interrupt is received. We also need to check if any sleeping threads can be waken up. I thought at first to directly check `all_list` to find all the awakable sleeping threads, but it would be too inefficient, so I used a list to avoid that.

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.

In `threads/thread.h`:

```
// In struct thread
int real_priority;
struct lock *waiting_on_lock;
struct list locks_held;
```

`real_priority` is the thread's own priority, without donation. This is declared to deal with `set_priority`.

`waiting_on_lock` is the lock this thread is waiting on. This helps updating chain donation when the priority of the current thread is changed.

`locks_held` is the list of locks that this thread holds. This helps determine what's the priority of this thread after donation.

In `threads/synch.h`:

```
// In struct lock
struct list_elem elem;
int max_priority;
```

`elem` is for add the lock in `locks_held`.

`max_priority` is the maximum priority over all the threads that are waiting for this lock. This helps the lock holder to calculate the donated priority faster.

In `threads/synch.c`:

```
// In struct semaphore_elem
struct thread *thread_waiting;
```

`thread_waiting` is the thread this `semaphore_elem` concerns and represents the thread that's waiting on a condition.

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

The lock has a `max_priority` to track the maximum priority donation among its waiters. Each thread can also track its donors/donees by check `locks_held` and `waiting_on_lock`.

For the nested donation example, we consider here a thread L with priority 0 holding a lock L1, a thread M with priority 31 holding a lock L2 watiing for L1, and a thread H with priority 63 trying to acquire L2.

First, the environment is prepared (thread H hasn't been created).

```
+-----------------------+ +-----------------------+ +-----------+
| thread L              | | thread M              | | thread H |
| priority: 31          | | priority: 31          | | yet       |
| real_priority: 0      | | real_priority: 31     | | to        |
| locks_held: [L1]      | | locks_held: [L2]      | | be        |
| waiting_on_lock: NULL | | waiting_on_lock: L1   | | created..|
+-----------------------+ +-----------------------+ +-----------+
+---------------------+ +---------------------+
| L1                  | | L2                  |
| holder: thread L    | | holder: thread M    |
| waiters: [thread M] | | waiters: []         |
| max_priority: 31    | | max_priority: 0     |
+---------------------+ +---------------------+
```

Then, thread H is created trying to acquire L2.

```
+------------------------+ +----------------------+ +----------------------+
| thread L               | | thread M             | | thread H             |
| priority: 31           | | priority: 31         | | priority: 63         |
| real_priority: 0       | | real_priority: 31    | | real_priority: 63    |
| locks_held: [L1]       | | locks_held: [L2]     | | locks_held: []       |
| waiting_on_lock: NULL  | | waiting_on_lock: L1  | | waiting_on_lock: NULL|
+------------------------+ +----------------------+ +----------------------+
+----------------------+ +----------------------+
| L1                   | | L2                   |
| holder: thread L     | | holder: thread M     |
| waiters: [thread M]  | | waiters: []          |
| max_priority: 31     | | max_priority: 0      |
+----------------------+ +----------------------+
```

As thread H failed to acquire L2, it calls function `thread_update_donation`, which calls another function `chain_update_donation` that first update the priority of thread M.

```
+------------------------+ +----------------------+ +----------------------+
| thread L               | | thread M             | | thread H             |
| priority: 31           | | priority: 63         | | priority: 63         |
| real_priority: 0       | | real_priority: 31    | | real_priority: 63    |
| locks_held: [L1]       | | locks_held: [L2]     | | locks_held: []       |
| waiting_on_lock: NULL  | | waiting_on_lock: L1  | | waiting_on_lock: L2  |
+------------------------+ +----------------------+ +----------------------+
+----------------------+ +----------------------+
| L1                   | | L2                   |
| holder: thread L     | | holder: thread M     |
| waiters: [thread M]  | | waiters: [thread H]  |
| max_priority: 31     | | max_priority: 63     |
+----------------------+ +----------------------+
```

Upon the update of M's priority, `update_lock_priority` is called to modify the `max_priority` of the lock it's waiting for.

```
+------------------------+ +----------------------+ +----------------------+
| thread L               | | thread M             | | thread H             |
| priority: 31           | | priority: 63         | | priority: 63         |
| real_priority: 0       | | real_priority: 31    | | real_priority: 63    |
| locks_held: [L1]       | | locks_held: [L2]     | | locks_held: []       |
| waiting_on_lock: NULL  | | waiting_on_lock: L1  | | waiting_on_lock: L2  |
+------------------------+ +----------------------+ +----------------------+
+----------------------+ +----------------------+
| L1                   | | L2                   |
| holder: thread L     | | holder: thread M     |
| waiters: [thread M]  | | waiters: [thread H]  |
```

```
| max_priority: 63     | | max_priority: 63     |
+----------------------+ +----------------------+
```

Then, since L2's `waiting_on_lock` isn't NULL, `chain_update_donation` is recursively called for thread L.

```
+------------------------+ +----------------------+ +----------------------+
| thread L               | | thread M             | | thread H             |
| priority: 63           | | priority: 63         | | priority: 63         |
| real_priority: 0       | | real_priority: 31    | | real_priority: 63    |
| locks_held: [L1]       | | locks_held: [L2]     | | locks_held: []       |
| waiting_on_lock: NULL  | | waiting_on_lock: L1  | | waiting_on_lock: L2  |
+------------------------+ +----------------------+ +----------------------+
+----------------------+ +----------------------+
| L1                   | | L2                   |
| holder: thread L     | | holder: thread M     |
| waiters: [thread M]  | | waiters: [thread H]  |
| max_priority: 63     | | max_priority: 63     |
+----------------------+ +----------------------+
```

Since L is not waiting for any locks, `chain_update_donation` returns. `ready_list` is sorted and L starts running. Suppose L releases lock L1 eventually, which cause M and L to stop donation.

```
+------------------------+ +-----------------------+ +----------------------+
| thread L               | | thread M              | | thread H             |
| priority: 0            | | priority: 63          | | priority: 63         |
| real_priority: 0       | | real_priority: 31     | | real_priority: 63    |
| locks_held: []         | | locks_held: [L1, L2]  | | locks_held: []       |
| waiting_on_lock: NULL  | | waiting_on_lock: NULL| | waiting_on_lock: L2  |
+------------------------+ +-----------------------+ +----------------------+
+----------------------+ +----------------------+
| L1                   | | L2                   |
| holder: thread M     | | holder: thread M     |
| waiters: []          | | waiters: [thread H]  |
| max_priority: 0      | | max_priority: 63     |
+----------------------+ +----------------------+
```

Then, M starts running and also releases L2, which returns the donation and cause H to preempt the CPU.

```
+----------------------+ +----------------------+ +----------------------+
| thread L             | | thread M             | | thread H             |
| priority: 0          | | priority: 31         | | priority: 63         |
| real_priority: 0     | | real_priority: 31    | | real_priority: 63    |
| locks_held: []       | | locks_held: [L1]     | | locks_held: [L2]     |
| waiting_on_lock: NULL| | waiting_on_lock: NULL| | waiting_on_lock: NULL|
```

```
  +----------------------+ +----------------------+ +----------------------+
  +---------------------+ +---------------------+
  | L1                  | | L2                  |
  | holder: thread M    | | holder: thread L    |
  | waiters: []         | | waiters: []         |
  | max_priority: 0     | | max_priority: 0     |
  +---------------------+ +---------------------+
```

And thus finishing our journey of how nested donation is processed in my pintos kernel.

ALGORITHMS

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

Each time we find the thread with the highest priority in the lock/semaphore/condition variable's waiter list and unblock it only. We use the `list_max` function for this purpose.

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

Each time a thread becomes waiting for a lock, it's added into the lock's waiter list. Addtionally, the lock's `max_prioity` is updated to the max priority over all the waiters. Then, we update the `priority` of the `lock_holder`. If the `lock_holder` is also waiting for a lock, we update the `priority` of the `lock_holder` of that lock too, and so on recursively.

Nested donation is handled naturally by iterating through all the locks a thread holds and so it could receive donation from different waiters of different locks. Once a thread releases a lock, that lock is removed from its `locks_held`. A new lock holder is chosen from the previous waiters and both the old and the new holder's priority is updated.

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

When `lock_release()` is called, the lock is removed from the lock holder's `locks_held`. The lock holder's priority is updated consequently and recursively for the holder of the lock it's waiting on and so on. Then, the waiter with the highest priority is waken up to acquire the lock and its priority is updated with the donations from the waiters of this lock.

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?

After setting the priority, one potential race is that when updating the priority of other threads (chain donation), a thread switch might happens, causing the priority to be incorrectly calculated and (potentially) the donation relation to change, leading to a different result or even a crash.

To avoid this, I turned off interrupt before modifying the priorities of the threads.

I don't think a lock could solve this since a lock could only protect a variable from being read/written simultaneously. The problem here is that the race might cause the kernel to wrongly schedule the threads, so before this lock takes effect the kernel has already switched to a wrong thread.

RATIONALE

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

I considered first to save a `donors` and `donate_to` in each thread but the problem is this design couldn't effectively handle nested donation since it's removing all the waiters of a lock just released from `donors` would be messy.

I chose this design mainly because of the natural hierarchical structure of the donation. In fact, each change of the donors to a thread is done for all the waiters of a lock. When a thread acquires a lock successfully, all the waiters become its donors. It's also more efficient since we only need to calculate once for each lock its `max_priority`.

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.

In `threads/thread.c`:

```
static real load_avg;
```

`load_avg` is, as described in the [4.4BSD] chapter, the system load average.

In `threads/thread.h`:

```
// In struct thread
int nice;
real recent_cpu;
```

`nice` is the niceness of this thread.

`recent_cpu` is a measure for how much cpu time this thread has occupied lately.

In `lib/read.h`:

```
#define PRECISION 14
typedef int real;
```

Here we define the precision of the fixed-point arithmetic as 17.14 and typedef int to real. This might seem useless since we could well use int but using `real` in our main codes allows for a better readability.

ALGORITHMS

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

First of all, within interrupt context, we couldn't effectively update the `load_avg`, `recent_cpu` since timer interrupt is automatically disabled. If the in-context work occupies too much time, eventually the `priority` of each thread would be messed up due to the fact that multiple timer interrupts could be blocked and discarded within interrupt context.

Additionally, even if we don't do that much work inside interrupt context, there's still chance it could harm the performance of the scheduler since the time spent in interrupt is also a part of the time occupied by the current thread. Thus, we are depleting valuable time for threads to do their own work. Even worse, this can cause the `recent_cpu` to abnormally increase. This is analogous to an overwhelming administration cost in real life. Thus, the thread would run slower and the scheduler might be less impartial.

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?

I think the advantages of my design are that

- I used a `real_priority` so setting the priority of a thread is naturally done by taking the maximum over `real_priority` and the `max_priority` of the locks.
- There no need to track the users since the donation comes with acquiring locks and modifying `locks_held`.
- I added basic assertions and `const` declarations to avoid potential coding mistakes.

The disadvantages are that

- I used only the fixed-point arithmetic, which itself isn't very precise and might loss a few bits when doing multiplication and division.
- The way I deal with chained donation is time-consuming in that it used a recursive design.
- I sort `ready_list` each time a donation/priority update happens, which requires O(nlogn) time complexity.
- The `list_max` function I used to obtain the waiter with the highest priority is not efficient enough.

If I had enough time, I would first implement 64 priority queues for `ready_list` according to the method mentioned in [4.4BSD]. This made the cost of each donation update O(n) and finding the next thread to switch to O(1). I would also re-implement the `update_donation` in a way that does not require recursion. If ideal, I would establish a new `real` struct such that it supports floating-point arithmetics. Finally, I could introduce a new data structor `heap` to substitute the lists used for waiter lists so that each heap adjustment and heap-poping requires less time.

> 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 added a whole new pair of files `lib/real.c`, `lib/real.h` into the pintos kernel. I also provided an new abstract data type `real` coupled with the functions like `real_mul` for it.

I think such labor to create a set of new APIs for an abstract data type is necessary mainly due to three reasons.

- new APIs supports better readability since all the functions and self-explainable. It also provides better encapsulation such that we don't need to worry about tempering the basic codes for manipulating real values.
- APIs have better support for future development. If we want to change the implementation of our real arithmetic, we don't need to `grep` all the places where we implement it. Instead, we just change the data type and the corresponding functions.
- Although C is not a strong typing language, introducing a new data type helps prevent stupidity in coding in that it indicates in what form should a variable be in.

I've actually thought about changing all the functions to macros, but functions are easier to maintain and develop in the future, so I settle with using build-in functions.