# Project 1: Threads

## Preliminaries

> Fill in your name and email address.

Jianan Ji 2000012995@stu.pku.edu.cn

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

Maybe next time a message chould be given that the first exercise may influence the following ones because of the possible low efficiency. I am one of those students who take a long time to find this before changing the way I implement it. Of course, if this is one of the aims of the teams, you could just ignore this ~

> 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.

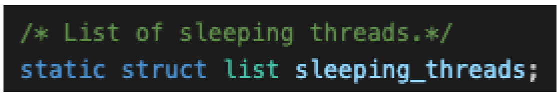
## Alarm Clock

#### 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.

**In timer.c**

Global variable sleeping_threads:

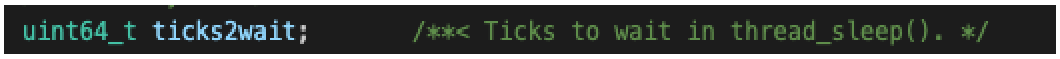


```
/* List of sleeping threads.*/
static struct list sleeping_threads;
```

To store threads sleeping.

**In struct thread**

struct member ticks2wait:



```
uint64_t ticks2wait;        /**< Ticks to wait in thread_sleep(). */
```

To store the remaining times to wait.

#### ALGORITHMS

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

Firstly, it will examine whether the ticks is less than 0. It will return here if so.

Secondly, give the int "ticks" to the variable ticks2wait in struct thread;

Thirdly, it will push the thread back to the list of thread sleeping and block the thread using thread_block();

And every time time_interrupt is called, it will check through the list of sleeping threads (rather than every thread), to minus their ticks2wait by 1 and see whether or not it becomes zero. If so, this thread will be unblocked by function thread_unblock() and removed from sleeping_threads list.

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

I use sleeping threads list to only see through sleeping ones to make the process faster by sacrifising some space used.

#### SYNCHRONIZATION

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

I use the pair of codes:

```
enum intr_level old_level = intr_disable ();
...
intr_set_level (old_level);
```

These functions work by disabling and then restoring the interrupts, which ensures that the critical section of the code is executed atomically. intr_disable () will disable interrupts temporarily and return the current interrupt level. By disabling interrupts, it ensures that no other thread can be scheduled to run during this time, effectively making the critical section atomic. And intr_set_level (old_level) re-enables interrupts and allows the system to resume normal operation, including executing interrupt handlers and scheduling other threads to run.

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

I use the pair of codes:

```
enum intr_level old_level = intr_disable ();
...
intr_set_level (old_level);
```

intr_disable () will disable interrupts temporarily and return the current interrupt level, intr_set_level (old_level) will then restore the original interrupt level after ... is completed. By disabling interrupts and then restoring them, you ensure that the critical section of the code is executed atomically, preventing any interference from timer interrupts while the calling thread is inside the critical section.

#### RATIONALE

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

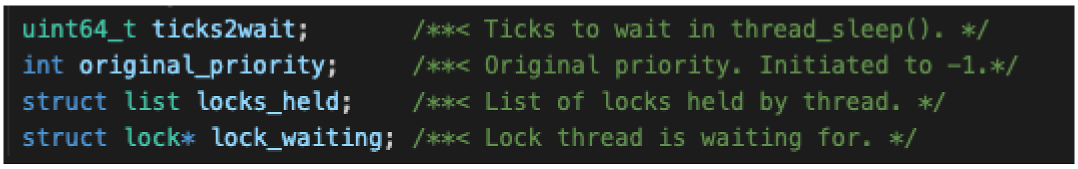
At first, I want to merely use thread_foreach to go through every single thread, but later I found this is not efficient and that's why I use sleeping threads list to only see through sleeping ones to make the process faster by sacrifising some space used.

## Priority Scheduling

### DATA STRUCTURES

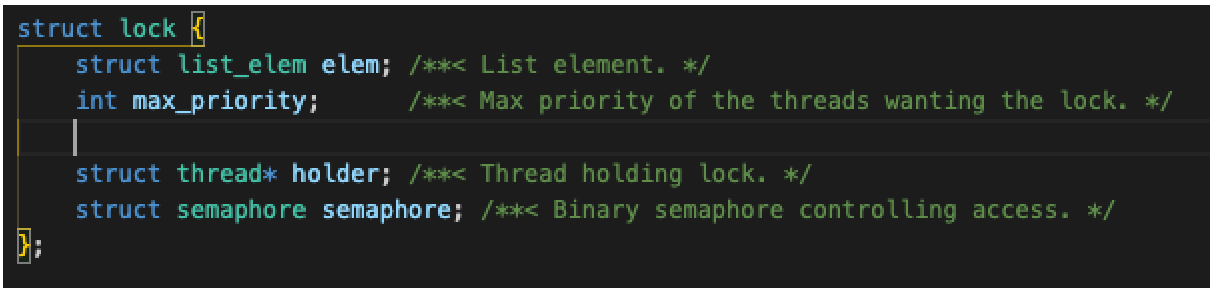
> B1: Copy here the declaration of each new or changed struct or struct member, global or static variable, typedef, or enumeration. Identify the purpose of each in 25 words or less.

**In struct thread**



```
uint64_t ticks2wait;        /**< Ticks to wait in thread_sleep(). */
int original_priority;      /**< Original priority. Initiated to -1.*/
struct list locks_held;     /**< List of locks held by thread. */
struct lock* lock_waiting; /**< Lock thread is waiting for. */
```

**In struct lock**



```
struct lock {
    struct list_elem elem; /**< List element. */
    int max_priority;       /**< Max priority of the threads wanting the lock. */

    struct thread* holder; /**< Thread holding lock. */
    struct semaphore semaphore; /**< Binary semaphore controlling access. */
};
```

> 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 locks_held list to store locks is holds and a pointer pointing to the lock it is waiting for. Every lock has a max_priority to store the highest priority of threads waiting for it.

```
A (high priority)   B (medium priority)   C (low priority)
 |                      |                     |
 +--------------------> L1                    |
                        |                     |
                        +-------------------> L2
```

Thread A is a high-priority thread that needs access to a shared resource, protected by lock L1. Thread B is a medium-priority thread that currently holds lock L1 but needs access to another shared resource, protected by lock L2. Thread C is a low-priority thread that currently holds lock L2. In order to prevent priority inversion, Thread A donates its priority to Thread B, which is represented by the arrow from A to L1 (indicating that A is waiting for the lock). Similarly, Thread B donates its elevated priority (inherited from Thread A) to Thread C, which is represented by the arrow from B to L2 (indicating that B is waiting for the lock).

### ALGORITHMS

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

Threads will be inserted into the ready_list using list_insert_ordered function which will maintain the sequence of the priority. So we could simply use list_min and list_remove to remove and get the thread with the highest priority. I maintain a priority queue, therefore the waiting threads will be stored in sequence from higher to lower priority.

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

Firstly, the thread's variation lock_waiting will be changed to the current lock. And the lock's max_priority will be changed to the thread acquiring the lock.

Secondly, as the realization of the donation, the current holder of the lock's priority will be temporarily modified to the higher priority.

Thirdly, the process of nested donation is handled. the whole process is in a loop, recursively going through every thread on the lock chain. it will change their priority into the current highest value as well as their waiting lock's max_priority.

When the lock is finally available, the sema_down() function is run, decreasing the lock's semaphore.

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

Basically, when lock_realease is called, the priority should be recovered to the original one.

Firstly, list_remove(&lock->elem) will be called to remove the lock from the locks_held list of the thread, meaning that the thread doesn't hold the lock anymore.

Sercondly, the thread's priority will be recovered to the original one using the unchanged int original_priority which was initialized to the priority when the thread is initialized.

And then, as one of the locks held is removed, we need to reconsider the current priority. Therefore, we will use list_front function to get the first lock element in the held list to get the current biggest priority of the locks and change the thread's to the bigger one, as the lost is maintained as a priority queue.

Fourthly, we will point the lock's holder variation to NULL meaning that the lock is now without a holder.

Lastly, we will use sema_up to increase the value of the lock's semaphore to make others able to acquire the valible 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?

If there are many threads with the same priority that need to lower their priority, and the operation takes a long time, the priority FIFO scheduling mechanism and the thread switch in timer_interrupt may delay the priority-lowering operation of the threads for a long time. Disabling interrupts can solve this problem. Using a lock to solve this issue isn't the best option because acquiring and releasing the lock in thread_set_priority() may cause other threads to compete in critical sections. Moreover, acquiring locks in interrupt handlers is not safe because the lock may already be held, leading to deadlocks. Therefore, in this case, disabling interrupts may be a better solution.

#### RATIONALE

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

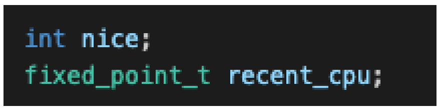
First of all, the design is direct and easy to think of and implement. What's more, we could seperate the whole implement into different parts which make it easier to maintain and understand.

## 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.

1. typedef int fixed_point_t; To seemingly make a new type for fixed_point numbers for convenience. It also has a series of macros to calculate them (with int).



```
int nice;
fixed_point_t recent_cpu;
```

2. recent_cpu and nice in thread. They are what their names are.

3. global fixed_point_t ariable load_avg to restore the calculation of load_avg.

#### ALGORITHMS

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

My approach involves performing the majority of the work inside the interrupt context. As a result, other processes may be blocked while the scheduler is running, causing higher latency and decreased overall system throughput. However we can ensure that it has a current and accurate view of the situation of the whole PintOS. This allows it to have better performance.

#### 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?

My code use direct and understandable ways to implement and is very safe for multiple threads.

When meeting a bug, I always think maybe I should add more

```
enum intr_level old_level = intr_disable ();
...
intr_set_level (old_level);
```

in my code to make it safer, leading to a consequence that I don't really know in which place I use it properly and in which place it is actually useless, which shows I should still dig into it to know more clearly about what these codes can do and what exactly happen behind them.

My code is also not efficient enough and several places should be modified further if I have enough time (T_T). This gives me a warn that I should start the following projects much much much much much much much much earlier!!!!!

> 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 create a header file named fixed_point.h and use macros to implement it. I also typedef fixed_point_t as int.

I did so because firstly, it gives the related codes readability. It simplifies the code and makes it more readable, as the operations on fixed-point numbers are explicitly defined through the abstraction layer's macros with understandable names.

What's more, this improves maintainability, as any changes to the fixed-point representation or arithmetic can be made within the abstraction layer without affecting the rest of the code.

Last but not least, I use macros because macros are expanded at compile-time, which may result in faster execution since there is no function call overhead.