CS 130 Project 1: Threads
Design Document

Tianyi Zhang
School of Information Science and Technology
2018533074
zhangty2@shanghaitech.edu.cn

Haoran Dang
School of Information Science and Technology
2018533259
danghr@shanghaitech.edu.cn

0 PRELIMINARIES

0.1 Preliminary Comments

No preliminary comment for this project.

0.2 References

- https://www.runoob.com/cprogramming/c-enum.html: To understand `enum` in the code.

1 ALARM CLOCK

1.1 Data Structures

1.1.1 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 file `threads/threads.h`

```
struct thread
{
   ...
   /* A counter of remaining sleeping ticks */
   int64_t sleeping_ticks;
   ...
}
```

1.2 Algorithms

1.2.1 Briefly describe what happens in a call to `timer_sleep()`, including the effects of the timer interrupt handler. When `timer_slee p` is invoked, it set a counter inside the thread as a countdown of remaining sleeping ticks, and calls `thread_block` to avoid it from running. In the counter, 0 stands for not sleeping and positive number stands for the remaining ticks.

In each tick, timer interrupt is called, in which the total tick counters of the OS `ticks`, of the idle thread `idle_ticks`, and of the kernel thread `kernel_ticks` are updated. Then, the sleeping status of all threads is checked by calling `thread_sleep_monitor` through `thread_foreach`, which will subtract the counter by 1 of all sleeping threads. When a counter reaches 0, its thread is unblocked by using `thread_unblock`, which will put it into the ready list.

1.2.2 What steps are taken to minimize the amount of time spent in the timer interrupt handler? In the handler, only one operation

```
thread_foreach (&thread_sleep_monitor, t);
```

is added. Since Pintos does not maintain a list of sleeping items, and it is easy to check whether a thread is sleeping simply using `status` and `sleeping_ticks`, it does not require much time even if we look into all the threads.

1.3 Synchronization

1.3.1 How are race conditions avoided when multiple threads call `timer_sleep()` simultaneously? Inspired by function `timer_ticks`, we can use

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

to ensure a non-interruptible operation on the current thread.

First we call `intr_disable`, which will make the process uninterruptible and returns the old status. Then we do our operations which are now uninterruptible. Finally, we restore the interrupt status by `intr_set_level`.

Thus, since the thread is uninterruptible, we can ensure the value of `sleeping_ticks` is correct when the thread is blocked.

1.3.2 How are race conditions avoided when a timer interrupt occurs during a call to `timer_sleep()`? Similar to 1.3.1, all the operations in `timer_sleep()` is uninterruptible, so the race condition of interrupts is avoided.

1.4 Rationale

1.4.1 Why did you choose this design? In what ways is it superior to another design you considered? `thread_foreach`, `thread_block` and `thread_unblock` is mentioned in the project guide, which are convenient to use to set or update the sleeping status of a thread.

We considered using a special number like `-1` for threads which are not sleeping, and in each tick we unblock the threads whose `sleeping_ticks` is 0. But such method requires more operations, and will cause one more cycle of sleep.

We also considered maintaining a list of sleeping threads, but as mentioned in 1.2.2, it is costy to maintain an extra list when we can detect whether a thread is sleeping just by inspecting `status` and `sleeping_ticks` of a thread.

2 PRIORITY SCHEDULING

2.1 Data Structures

2.1.1 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 file `threads/threads.h`

```
struct thread
{
   ...
```

```
  /* List of holding locks. */
  struct list locks;
  /* Priority without donation. */
  int priority_wo_donation;
  /* The lock a thread is waiting for. */
  struct lock *lock_wait;
  ...
}
```

- In file `threads/synch.h`

```
struct lock
{
  ...
  /* Priority donated by this lock. */
  int donated_priority;
  ...
}
```

2.1.2 Explain the data structure used to track priority donation. Use ASCII art to diagram a nested donation. (Alternately, submit a `.png` *file.)* We use `lock* lock_wait` to find the thread we are donating. And we store the locks held by the current thread in `thread.locks`. In the locks we store the priority donated by another thread which means we can find the max donated priority.

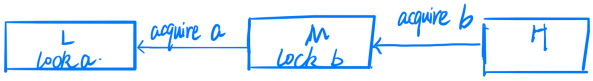

Figure 1: Nested Donation

2.2 Algorithms

2.2.1 How do you ensure that the highest priority thread waiting for a lock, semaphore, or condition variable wakes up first? We have already push all the blocked threads in the `sema.waiters` or `cond.waiters` when we block the threads. Therefore the only thing we need to do is sort the waiting list with the order of the thread's priority.Then the front element is the thread we want.

2.2.2 Describe the sequence of events when a call to `lock_acquire()` *causes a priority donation. How is nested donation handled?* When we call the `lock_acquire()` we check whether the lock has a holder. Then we use the lock to locate its holder, update the holder's priority if the current thread's priority is higher. After that we will check the holder's `lock_wait` then we can update the priority donated by holder if there is a thread holds a lock that holder wants to acquire. After doing the iterations, we will update the priority in a nested situation.

2.2.3 Describe the sequence of events when `lock_release()` *is called on a lock that a higher-priority thread is waiting for.* We will clear the `lock->holder` then reomove the lock from current thread's locks list. Then we call `thread_update_priority()` to update current thread's priority. Then we will use `sema_down()` to unblock the thread which tried to acquire the lock, pop it from the waiting list. Then use `thread_yield` to switch to the new thread.

2.3 Synchronization

2.3.1 Describe a potential race in `thread_set_priority()` *and explain how your implementation avoids it. Can you use a lock to avoid this race?* When we call `thread_set_priority()` to set a thread which has been donated by a higher priority, we cannot just change this thread's priority. We need to store the priority and then update to this priority after the donation.

We just have a variable named `priority_wo_donation` to store the base priority. We just change this value if current thread is donated by another thread. And we cannot use a lock to avoid this race.

2.4 Rationale

2.4.1 Why did you choose this design? In what ways is it superior to another design you considered? We choose the design because we could use the `thread.locks` to get the highest priority donated by multiple threads easily. `thread.lock_wait` enablesus to find the thread we donate to. `lock.donated_priority` will avoid the race for changing the priority of the current thread.

3 ADVANCED SCHEDULER

3.1 Data Structures

3.1.1 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 file `threads/threads.h`

```
...
/* Lowest "nice" value. */
#define NICE_MIN -20
/* Default "nice" value. */
#define NICE_DEFAULT 0
/* Highest "nice" value. */
#define NICE_MAX 20
...
struct thread
  {
    ...
    /* "nice" value of a thread */
    int nice;
    /* "recent_cpu" value of a thread */
    fixed_point recent_cpu;
    ...
  }
```

- In file `threads/threads.c`

```
/* System load average */
fixed_point load_avg;
```

3.2 Algorithms

3.2.1 Suppose threads A, B, and C have nice values 0, 1, and 2. Each has a `recent_cpu` *value of 0. Fill in the table below showing the scheduling decision and the* `priority` *and* `recent_cpu` *values for each thread after each given number of timer ticks:*

Timer ticks	recent_cpu			priority			Thread to run
	A	B	C	A	B	C	
0	0	0	0	63	61	59	A
4	4	0	0	62	61	59	A
8	8	0	0	61	61	59	B
12	8	4	0	61	60	59	A
16	12	4	0	60	60	59	B
20	12	8	0	60	59	59	A
24	16	8	0	59	59	59	C
28	16	8	4	59	59	58	B
32	16	12	4	59	58	58	A
36	20	12	4	58	58	58	C

3.2.2 Did any ambiguities in the scheduler specification make values in the table uncertain? If so, what rule did you use to resolve them? Does this match the behavior of your scheduler? Yes, since the specification does not mention what the shceduler will do if the calculated priority of the two threads is the same.

Just as our implementation in task 1 and 2, the ready list is maintained as a priority list, and the sorting method will insert the item at the place after the items with same priority in the list. The value in this table uses the method similar to our implementation.

3.2.3 How is the way you divided the cost of scheduling between code inside and outside interrupt context likely to affect performance? Either when changing the `nice` by the thread itself or during an interrupt, the ready thread will be put into `ready_list` according to the priority, so there seems to be small difference between these two situations in our implementation.

3.3 Rationale

3.3.1 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?

(1) **Advantages:** Easy to implement and resource-saving, as it uses the original scheduler in task 2 and maintains only one list according to the calculated priority.
(2) **Disadvantages:** The stability and performance of the scheduler still need to be improved. Also, complex scheduling algorithms considering not only calculated priority is not convenient to implement under this design.
(3) **Things to do:** Figure out more about execution inside and outside interrupt context, and improve the stability by marking atomic operations, etc.

3.3.2 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? Fixed-point numbers and opeartions are implemented in `threads/fixed-point.h` and `fixed-point.c`, in which we use a `struct`, containing only one `int` number, and a set of function.

Although (maybe) not all the functions are used, implementing it will reduce the workload afterwards and will make the code neat than directly showing a lot of computation processes. Meanwhile, using a `struct` instead of `typedef int fixed_point` can directly show misuse of the types in the compiler, so that we don't need to worry about wrong type conversion.

4 SURVEY QUESTIONS

In your opinion, was this assignment, or any one of the three problems in it, too easy or too hard? Did it take too long or too little time? Tasks in project 1 are not quite "hard" yet but it does require a lot of time since the operating system is a complex task.

Did you find that working on a particular part of the assignment gave you greater insight into some aspect of OS design? Working on `timer_sleep` and `timer_interrupt` in task 1 and 3 offers us a direct impression on what happens when CPU ticks. Meanwhile, in 2 we considered a lot of different situations, which inspires us to understand how nested changes take place in the operating system.

Is there some particular fact or hint we should give students in future quarters to help them solve the problems? Conversely, did you find any of our guidance to be misleading? Actually no. But I guess it would be slightly better if the TAs can introduce some Git GUI softwares as they can increase the efficiency when using complex functions sometimes.

Do you have any suggestions for the TAs to more effectively assist students, either for future quarters or the remaining projects? Not yet.

Any other comments? Not yet.

CONTRIBUTORS

Task		Tianyi Zhang	Haoran Dang
Task 1 Alarm Clock	Concept	✓	✓
	Implementation	✓	✓
	Debugging	✓	✓
	Design Document		✓
Task 2 Priority Scheduling	Concept	✓	✓
	Implementation	✓	✓
	Debugging	✓	
	Design Document	✓	
Task 3 Advanced Scheduler	Concept	✓	✓
	Implementation	✓	✓
	Debugging	✓	✓
	Design Document		✓