CS130 Project 1: Threads
Design Document

Group 13

Tianyao YAN
yanty@shanghaitech.edu.cn

Wenfei YU
yuwf@shanghaitech.edu.cn

I. Introduction

This document answers the questions in the template for Project 1.

II. Alarm Clock

A. Data Structures

- 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 thread.h:

```
struct thread
{
    ...
    /* Record the remaining ticks the
       thread needs to sleep. */
    int64_t ticks_to_sleep;
    ...
}
```

B. Algorithms

- Briefly describe what happens in a call to timer_sleep(), including the effects of the timer interrupt handler.
  1) Disable interrupt.
  2) Set ticks_to_sleep of current thread to ticks.
  3) Block the current thread.
  4) Recover old interrupt level.
- What steps are taken to minimize the amount of time spent in the timer interrupt handler?
  1) Record remaining ticks to sleep of every thread.
  2) Update the remaining ticks when ticking.
  3) Check whether a thread should wake up or not.
  4) If any thread wakes up, unblock the thread, interrupt the current one, and reschedule the threads.

C. Synchronization

- How are race conditions avoided when multiple threads call timer_sleep() simultaneously?

  By only operating on the current running thread.
- How are race conditions avoided when a timer interrupt occurs during a call to timer_sleep()?

  By disabling interrupts.

D. Rationale

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

  We would have switched to another design without hesitation if we had found one.

III. Priority Scheduling

A. Data Structures

- 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 thread.h:

```
struct thread
{
    ...
    /* Original priority. */
    int original_priority;
    /* List of locks the thread holds.
         */
    struct list lock_hold;
    /* Lock the thread is waiting for.
         */
    struct lock *lock_wait;
    ...
}
```

In synch.h:

```
struct lock
{
    ...
    /* Max priority among lock holder
        and threads acquiring the lock.
         */
    int priority;
    /* List element. */
    struct list_elem elem;
    ...
}
```

- Explain the data structure used to track priority donation. Use ASCII art to diagram a nested donation.

  1) When a thread obtains a lock, it will donate its priority to the lock holder if the holder's priority is

lower than that of the acquiring thread. This priority donation process recurs if the lock is still held by another thread. After releasing the lock, the thread's original priority will be recovered.
2) If a thread receives priority donations from multiple threads, the current priority will be adjusted to max_priority.
3) When setting the priority of a thread, if the priority of the thread is donated and has not recovered yet, set original_priority. Then, if the priority set is greater than the current one, the current one will be changed. When the thread's priority is restored, it will revert to its original_priority.
4) When releasing a lock, set the priority to the max priority of the waiting threads. If there is no waiting thread, set the priority to PRI_MIN.

```
Thread 1-----+
   |         |acquire
   |         v
   |donate  lock 1
   |         ^
   v         |hold
Thread 2-----+
   |         |acquire
   |         v
   |donate  lock 2
   |         ^
   v         |hold
Thread 3-----+
......
```

B. Algorithms

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

  We made ready_list always sorted by priority.
- Describe the sequence of events when a call to lock_acquire() causes a priority donation. How is nested donation handled?
  1) Recursively donate priority when the acquired lock needs to wait for another lock.
  2) Call sema_down() to decrease the semaphore.
  3) Hold the lock.
- 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.
  2) Recover the priority.
  3) Call sema_up() to increase the semaphore.

C. Synchronization

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

  Call thread_yield() to rearrange the order of the threads to execute.

D. Rationale

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

  We will choose another design as soon as god tells us one.

IV. Advanced Scheduler

A. Data Structures

- 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 thread.h:

```c
struct thread
{
    ...
    /* Nice value of this thread. */
    int nice;
    /* Recent CPU value of this thread. */
    fp recent_cpu;
    ...
}
```

In thread.c:

```c
/* Global variable: system load average. */
fp load_avg;
```

In fixed_point.h:

```c
/* Define fixed_point type based on int. */
# typedef int fp;
```

B. Algorithms

- 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:
- 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?

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

1) The specification does not specify the order of updating recent_cpu and priority. We chose to update recent_cpu before priority.
2) There is no explicit specification to address situations in which multiple threads share the same priority. We used default sorting specifications in function list_insert_ordered based on function thread_compare_priority.

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

    Increasing recent_cpu is operated inside the interrupt context. The function of updating the status of the threads, including load_avg and recent_cpu, is called inside the interrupt context. The actual instructions of the function are realized outside the interrupt context and in thread.c, since load_avg and all_list are in thread.c. Updating priority is in both thread.c and the interrupt context. Therefore, we could decrease the cost of calling function.

C. *Rationale*

- 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) The advantage is that it is simple and nothing wrong happens when running this simple system.
    2) The disadvantage is that it is very rough and maybe not suitable in some complex operating system. If we had enough time, we would optimize it.
- 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?

    The new type fp, standing for fixed-point, is actually based on int. The use of that type is more conducive to readability when employed in the context of fixed-point arithmetic. In addition, macros are more efficient.

V. Survey Questions

Answering these questions is optional, but it will help us improve the course in future quarters. Feel free to tell us anything you want–these questions are just to spur your thoughts. You may also choose to respond anonymously in the course evaluations at the end of the quarter.

- 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?

    Task 2 of this project is so hard for us that we have experienced recurring nightmares about our inability to complete the project before the deadline.
- Did you find that working on a particular part of the assignment gave you greater insight into some aspect of OS design?

    Maybe.
- 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?

    Students need to be told which functions need to be modified or added.
- Do you have any suggestions for the TAs to more effectively assist students, either for future quarters or the remaining projects?

    For the physical and mental health of students, TA had better provide more tips.
- Any other comments?

    No.

References

- https://blog.csdn.net/u013058160/article/details/45393555/