CS 130: Project 1 - Threads Design Document

Group Information

Please provide the details for your group below, including your group's name and the names and email addresses of all members.

- **Group Name**: *lorin&lucy*
- **Member 1**: Lufei Li `<lilf2024@shanghaitech.edu.cn>`
- **Member 2**: Jintong Luo `<luojt2022@shanghaitech.edu.cn>`

Preliminaries

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

No.

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.

new member:

```
struct thread
{
   int64_t block_time;      /* Remaining blocked time. */
};
```

purpose: store the remaining blocked time

Algorithms

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

First, check whether the parameter is valid. Second, disable interrupt.

Then, set `block_time`.

After that, block the current thread, and timer interrupt handler will check whether the thread should be unblocked every tick.

Finally, restore interrupt.

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

Check whether `block_time` equals to 0 every tick.

Synchronization

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

By blocking the current thread.

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

By disabling interrupt at the beginning before set `block_time`.

Rationale

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

It's a simple way with only one addtional parameter to solve busy waiting.

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.

```
struct thread
{
   struct list locks;       /*locks held by the thread*/
   struct lock *waiting_lock;   /*lock that is waited for by the thread*/
   int original_priority;   /*stores the original priority of the thread*/
};

struct lock
{
    struct list_elem elem;  /*used in thread.c*/
    int max_priority;       /*max priority of threads aquiring the lock*/
};
```

> **B2:** Explain the data structure used to track priority donation. Describe how priority donation works in a nested scenario using a detailed textual explanation.

When a thread is trying to acquire a lock:

1. If the lock is already held by another thread, check the priority of the lock holder.
2. If the current thread's priority is greater than that of the lock holder, the current thread donates its priority to the lock holder.

3. If the lock holder itself is waiting for other locks, the priority donation is performed recursively.

When a thread releases a lock:

1. The thread's priority is restored to its `original_priority`.
2. If the thread still has other lock, its priority is the greater one of `original_priority` and `max_priority`.

Algorithms

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

By maintaining a priority queue, the first element in the queue is the one of the max priority.

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

1. Recursively donate the priority. (As is explained in B2)
2. Sema down.
3. Hold the lock.
4. Update the `max_priority` of the lock and the locks list of the thread.

> **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.
2. Update the priority of the thread. (As is explained in B2)
3. Sema up.

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?

A thread may be terminated immediately after set priority, so we yield the thread to rearrange the order.

Rationale

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

Using `list_insert_ordered()` to manage a priority queue is faster than finding the largest priority each time.

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.

```
struct thread
{
   int nice;            /*store the nice value for the thread*/
   int recent_cpu;      /*store recent cpu*/
};

int64_t  load_avg;      /*a global variable to store load average*/

implement fixed-point.h
```

Algorithms

> **C2:** 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:

Fill in the table.

Timer Ticks	recent_cpu A	recent_cpu B	recent_cpu C	Priority A	Priority B	Priority C	Thread to Run
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

> **C3:** 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?

When multiple threads have the same priority, the values became uncertain. We choose the thread that have been waiting for the longest time to run. It matches the behavior of our scheduler.

> **C4:** How is the way you divided the cost of scheduling between code inside and outside interrupt context likely to affect performance?

The code inside interrupt context need to be simple and fast. So we put checks such as whether to update priority, `recent_cpu` inside interrupt context, but the operations are outside the interrupt context.

Rationale

> **C5:** 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?

We've achieved goals of the first project by using simple algorithms according to the instructions, and it's easy to understand. However, simple algorithms may not be effitient or may encounter error in more sophisticated cases.

> **C6:** 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 (i.e., 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?

We use functions to manipulate fixed-point math, because it's more readable and maintainable. What's more, functions provide type check to ensure type safety.

Survey Questions

> Do you have any suggestions for the TAs to more effectively assist students, either for future quarters or the remaining projects? Any other comments?

No.