+--------------------+

| CS 140 |

| PROJECT 1: THREADS |

| DESIGN DOCUMENT |

+--------------------+

---- GROUP ----

10132510125 – 龚月云<[redeyeflight@outlook.com](mailto:redeyeflight@outlook.com)>

10121540104 – 孙诗韵

10132510150 – 胡豪杰

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 thread.h, added variable to thread struct:

int64\_t block\_ticks;

/\* Record the time the thread has been blocked. \*/

* In thread.c, added function

Void block\_check(struct thread \*t,void \*aux UNUSED);

/\* Decrease the value of block\_ticks by 1,

Wake up the thread when block\_ticks=0. \*/

---- ALGORITHMS ----

>> A2: Briefly describe what happens in a call to timer\_sleep(),

>> including the effects of the timer\_interrupt() call;

>> including the effects of the thread\_create() call which has also been modified;

In timer\_sleep():

1. Check for valid ticks argument (ticks > 0).
2. Set the interrupt status to OFF and store former status into old\_level;
3. Set the value of current thread’s block\_tick with ticks;
4. Block the thread;
5. Restore the interrupt status with old\_level;

In timer\_interrupt():

1. Add ticks by 1
2. For each thread, function block\_check() will be execute once

In block\_check():

1. If the thread has been blocked and its block\_ticks>0,
2. Decrease block\_ticks by 1,
3. Unblock the thread when block\_ticks=0

---- SYNCHRONIZATION ----

---- RATIONALE ----

>> Why did you choose this design? In what ways is it superior to

>> another design you considered?

* Before we modified the busy waiting, in timer\_sleep(), the thread just did the yield and put itself into ready list, which wasted quite a lot of cpu resourse, so we choose this design, do the block\_check every time when timer interrupt happens, and wake sleeping thread up when its block\_ticks decreased to 0.

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 thread.h, the following variables are added to the thread struct:

int old\_priority; /\* Old priority. \*/

struct list locks;/\* The lock the thread is holding. \*/

struct lock \*lock\_waiting;/\* The lock the thread is waiting for. \*/

* In synch.h, the following variables are added to the thread lock:

Struct list\_elem elem; /\* List element for priority donation. \*/

Int max\_priority; /\* Max priority among the threads acquiring the lock. \*/

* In thread.c, the following functions are added:

bool thread\_pri\_less (struct list\_elem \*elem1, struct list\_elem \*elem2, void \*aux UNUSED);

/\* priority compare function. \*/

Void thread\_hold\_the\_lock(struct lock \*lock);

/\* Let thread hold a lock \*/

Void thread\_donate\_priority (struct thread \*t);

/\* Donate current priority to thread t. \*/

Void thread\_remove\_lock(struct lock \*lock);

/\* Remove a lock from thread’s locks list. \*/

Void thread\_update\_priority(struct thread \*t);

/\* update thread’s priority after changed. \*/

* In synch.c, the following functions are added:

Bool lock\_pri\_less(struct list\_elem \*elem1, struct list\_elem \*elem2, void \*aux UNUSED);

/\* priority compare function for locks\*/

Bool cond\_pri\_less (const struct list\_elem \*elem1, const struct list\_elem \*elem2, void \*aux UNUSED);

/\* priority compare function for conditions\*/

---- ALGORITHMS ----

In thread\_donate\_priority():

1. Modify the priority of the thread,
2. Reinsert this thread into ready list after changing its priority

In thread\_remove\_lock():

1. Remove a lock from thread’s locks list,
2. Update the priority list of the thread,

In thread\_update\_priority():

1. Reinsert the thread into ready list.

In thread\_unblock():

1. Change the status of the blocked thread into READY,
2. Insert the thread into ready list in order

In thread\_yield():

1. Change the status of the running thread into READY,
2. Insert the thread into ready list orderly

In init\_thread():

1. Initialize the attributes of thread,
2. Set thread’s old priority with argument priority,
3. Initialize the lock list,
4. Initialize thread’s lock waiting list as null,
5. Insert the thread into ready list orderly

In thread\_set\_priority():

1. Set the priority of current thread with argument new\_priority,
2. Yield current thread to reinsert it into the ready list.

In thread\_create():

1. If the current thread has been blocked,
2. Yield this thread.

In lock\_release():

1. Remove lock from thread holding locks list.

In lock\_acquire():

1. The realization of priority donation before semaphore action,
2. Get the hold of the lock after being wake up

In cond\_signal():

In sema\_up():

In sema\_down():

1. Modify the waiting list into priority ordered list.

---- SYNCHRONIZATION ----

---- RATIONALE ----

>> Why did you choose this design? In what ways is it superior to

>> another design you considered?

Because there are lots of list struct in thread.c, it’s convenient to keep those lists as priority ordered list. To do this we added some priority compare functions into thread.c and synch.c to get right result.

We also added some attributes into the codes to achieve the goal, for instance the locks list ,the lock\_waiting list, and modified related function to get better result.

ADVANCED SCHEDULER

==================

---- 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, the following variables are added to the thread struct:

int nice;

/\* The thread's current nice value. \*/

Fixed\_t recent\_cpu;

/\* The thread's most recently calculated recent\_cpu value. \*/

* In thread.c, the following global variable is added:

int load\_avg;

/\* The system's most recently calculated load average value. \*/

---- ALGORITHMS ----

In timer\_interrupt():

1. Update the value of load\_avg of system and value of recent\_cpu of ALL thread every TIMER\_FREQ time,
2. Update the thread’s priority every 4 timer\_ticks() call,

In thread\_mlfqs\_increase\_recent\_cpu\_by\_one():

1. Increase the value of recent\_cpu of thread by 1 for every thread which is not idle,

In thread\_mlfqs\_update\_load\_avg\_and\_recent\_cpu():

1. Every per second to refresh load\_avg and recent\_cpu of ALL threads.

In thread\_mlfqs\_update\_priority():

1. Update priority of ALL threads.

---- 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?

Advantages:

- Simple to understand.

- Minimal adding of extra variables, kernel threads remain relatively

small. Also makes for easier readability.

-No busy waiting which wastes quiet a number of cpu resource.

-Decrease waiting time of the operating system.

Disadvantages:

- Performance suffers due to turning off interrupts instead of locking

variables.

- Ordered inserting and sorting lists takes O(n) and O(n log n)

respectively. Using binary tree / other more efficient data structures

may be needed. (Also, bad caching in linked lists)

To refine my design, I would implement the following features:

- Automatic deadlock detection

This is equivalent to finding cycles in the lock donation graph, which is

iterated through in the thread\_donate\_priority function. A maximum iteration

depth of 8 is set in case there is a deadlock cycle (without a depth

parameter, this would cause an infinite loop).

What would be optimal is to detect deadlock cycles and lower all the

deadlocked threads priorities to PRI\_MIN / kill the threads.

- Detect overflows in fix\_point.h

The values for priorities and nice are clamped, but the values for

recent\_cpu and load\_avg are not. Hence, this leaves the possibility for

overflow, which I currently do not check for. To be correct, the OS needs

to check for this.

I implemented fix-point math in a header file. The conversions between

integers and fixed-point and arithmetic was abstracted away in this

file. I used the standard functions as described in the Pintos

documentation and called these functions in my mlqfs calculation functions

in thread.c. Abstracting the fixed-point functions allowed for better

readability when calculating the mlqfs thread.c functions.