Final Report for Project 1: Threads

Name: 卓尔 (Zhuo Er)
SID: 11611026

Changes since Design Document

Task 1: Efficient Alarm Clock

For task 1, I just follow the idea in my initial design document.

Task 2: Priority Scheduler

- **Change 1**: Instead of using a list of locks to keep track of all the locks held by a thread, I implement a list called `donors` to record a list of threads wanting the locks held by a given thread.
- **Reason**: My previous idea was to loop around multiple data structures including `locks` to find the candidate donors. However, I make this change because I find that candidate donors can be directly tracked when a thread is trying to acquire a lock: the acquiring thread is a candidate donor of the lock holder. Therefore, directly add the acquiring thread into the donors list will be more efficient.

- **Change 2**: Instead of iterate through the `ready_list` when calling `next_thread_to_run()`, I make the `ready_list` a priority queue by calling `list_insert_ordered()` instead of `list_push_back`. Besides, I sort the `ready_list` by `list_sort()` in each donation carried out by `thread_donate_priority` to make each change in priority an ordered operation. In such case, `next_thread_to_run()` is not modified, and it only needs to pop the front element of the ordered `ready_list` by calling `list_pop_front()`, which is of highest priority.
- **Reason**: Previously, I thought that `list_insert_ordered` is called with `O(n)` time, which is at the same level of complexity as iterating the `ready_list` to find the thread with highest priority. However, although these two versions of `thread_yield()` are of the same order of complexity (`list_push_back()` + loop v.s. `list_insert_ordered()` + `list_pop_front`), there are cases that only `schedule()` is called for switching using `next_thread_to_run()` (e.g. in `thread_blcok()` and `thread_exit()`). In such cases, a more efficient implementation of `next_thread_to_run` is preferred.

- **Change 3**: Instead of calling `thread_donate_priority()` in `lock_acquire()`, I move it to `sema_down()`. Besides, I call `thread_yield_test()` to check if the current thread has the highest priority and yield if not, instead of directly calling `thread_yield()` after priority donation.
- **Reason**: Previously, I did not take into account semaphores other than locks, so I deceided to implement priority donations under `lock_acquire()`. However, I realize that `sema_down()` and `sema_up()` are more general for all semaphores, so I move `thread_donate_priority()` to `sema_down()` to make it available for different semaphores. Besides, I realize that `thread_yield()` is needless to call if the

current thread still holds the highest priority among threads in the `ready_list`. Therefore, I replace `thread_yield ()` with `thread_yield_test ()` after priority changes.

- **Change 4**: First, I add `donors_release()` to release donors during `lock_release()`. Instead of updating the `priority` of the thread to `max(base_priority, max(the max priority of each lock held by the thread))`, I update it to `max(base_priority, max(the donors' priorities))` using `priority_update()`. Also, I sort the semaphore waiters by `list_sort()` before calling `thread_unblock()` to unblock the one with the highest priority. I also use `thread_yield_test()` in `sema_up()` instead of `thread_yield()`.
- **Reason**: The reasons why I change to use `donors` are explained in Change 1. The reasons why I change to use `thread_yield_test()` are explained in Change 3. Because I realize that priority donation is not an ordered operation for semaphore waiters (`thread_donate_priority` directly change the lock holder's priority without keeping the holder's order in lists ordered by priority), I need to sort the waiters before unblocking them. I decide to sort the waiters in `sema_up()` since it is safer to do so right before unblocking and priority donations may happen more frequently but waiters only need to be sorted once here.

- **Change 5**: This change is about `thread_set_priority()`. Instead of setting the `priority` of the thread to `max(base_priority, max(the max priority of each lock held by the thread))`, I set it to `max(base_priority, max(the donors' priorities))` using `priority_update()` if `new_priority` is smaller than `priority`. I also call `thread_yield_test()` instead of `thread_yield()`.
- **Reason**: The reasons for these two changes have been explained in Change 1 and Change 3, respectively.

Task 3: Multi-level Feedback Queue Scheduler

- **Change 1**: Update the current thread's `priority` and call `thread_yield_test()` when changing its `nice` using `thread_set_nice()`. Also update the `priority` when updating `recent_cpu`.
- **Reason**: This is because in the MLFQS mode, `priority` is related to `nice` and `recent_cpu` and needs to be updated for each change in `nice` and `recent_cpu`. `thread_yield_test()` or `thread_yield_interrupt()`follows the change in priority as usual.

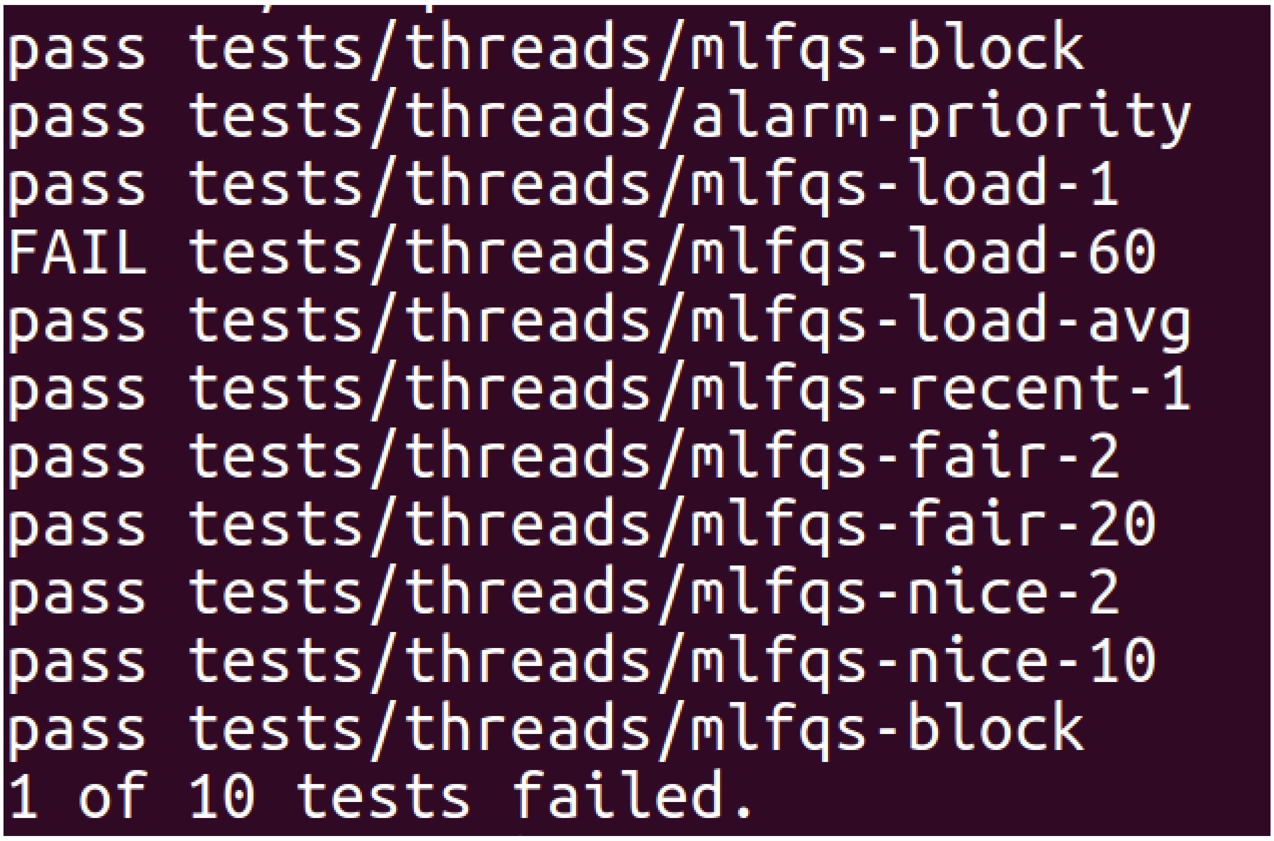
- **Change 2**: Wrap each thread get and set method with interrupt disable statements.
- **Reason**: This is for thread-safe and synchronous issues.

- **Change 3**: Split `thread_mlfqs_update_load_avg_and_recent_cpu()` into two methods updating `load_avg` and `recent_cpu`, respectively.
- **Reason**: This is just for lowering coupling.

GDB Test Report

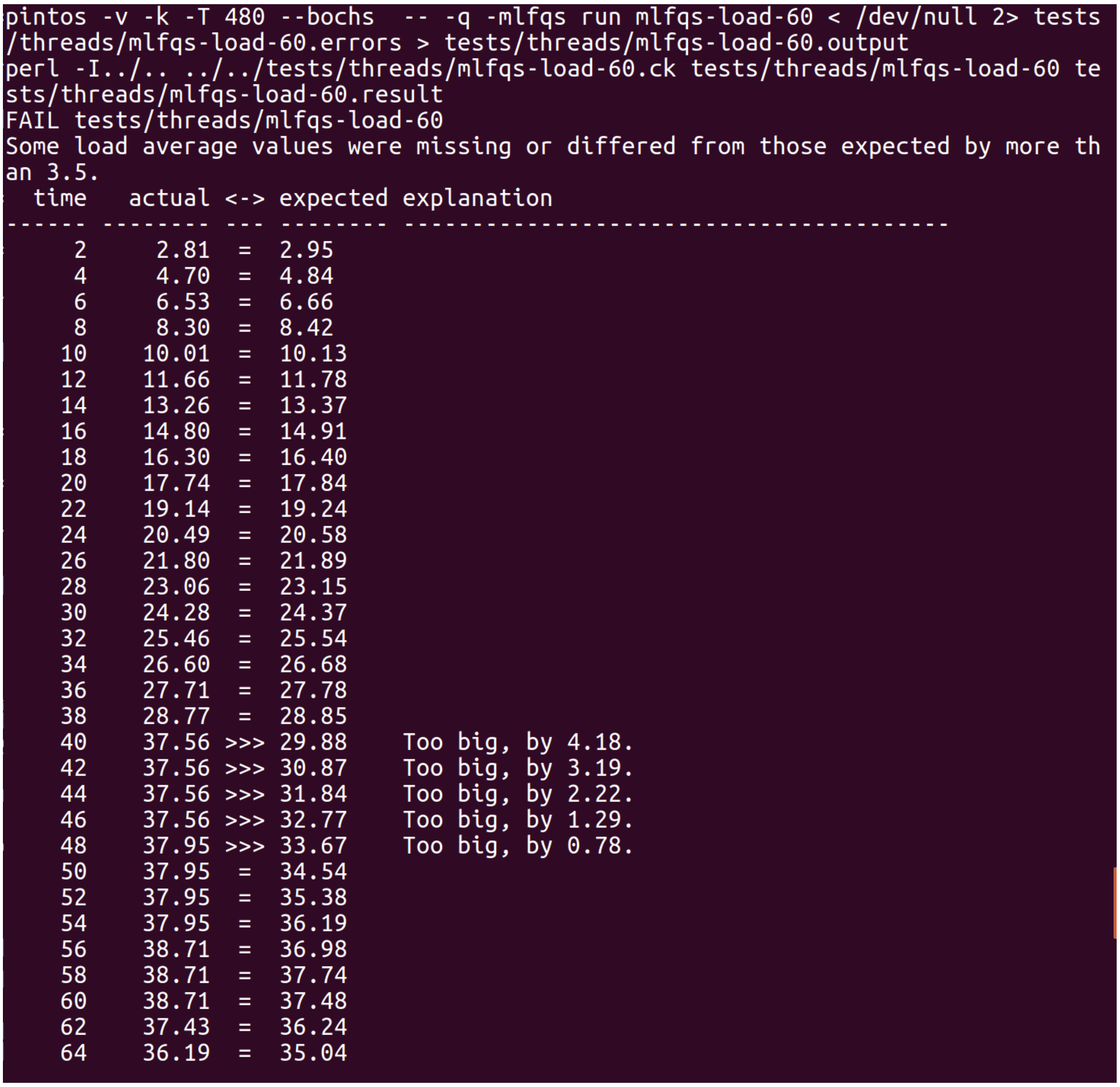
After running `make check` on the released src for this GDB task, the results show that test `mlfqs-load-60` fails due to several value leaps happening during the update of `load_avg`. The testing results are shown as follows.

Fig.1: Test result



```
pass tests/threads/mlfqs-block
pass tests/threads/alarm-priority
pass tests/threads/mlfqs-load-1
FAIL tests/threads/mlfqs-load-60
pass tests/threads/mlfqs-load-avg
pass tests/threads/mlfqs-recent-1
pass tests/threads/mlfqs-fair-2
pass tests/threads/mlfqs-fair-20
pass tests/threads/mlfqs-nice-2
pass tests/threads/mlfqs-nice-10
pass tests/threads/mlfqs-block
1 of 10 tests failed.
```

Fig.2: Test result



```
pintos -v -k -T 480 --bochs  -- -q -mlfqs run mlfqs-load-60 < /dev/null 2> tests
/threads/mlfqs-load-60.errors > tests/threads/mlfqs-load-60.output
perl -I../.. ../../tests/threads/mlfqs-load-60.ck tests/threads/mlfqs-load-60 te
sts/threads/mlfqs-load-60.result
FAIL tests/threads/mlfqs-load-60
Some load average values were missing or differed from those expected by more th
an 3.5.
  time   actual <-> expected explanation
------ -------- --- -------- ----------------------------------------
     2     2.81  =  2.95
     4     4.70  =  4.84
     6     6.53  =  6.66
     8     8.30  =  8.42
    10    10.01  =  10.13
    12    11.66  =  11.78
    14    13.26  =  13.37
    16    14.80  =  14.91
    18    16.30  =  16.40
    20    17.74  =  17.84
    22    19.14  =  19.24
    24    20.49  =  20.58
    26    21.80  =  21.89
    28    23.06  =  23.15
    30    24.28  =  24.37
    32    25.46  =  25.54
    34    26.60  =  26.68
    36    27.71  =  27.78
    38    28.77  =  28.85
    40    37.56 >>> 29.88    Too big, by 4.18.
    42    37.56 >>> 30.87    Too big, by 3.19.
    44    37.56 >>> 31.84    Too big, by 2.22.
    46    37.56 >>> 32.77    Too big, by 1.29.
    48    37.95 >>> 33.67    Too big, by 0.78.
    50    37.95  =  34.54
    52    37.95  =  35.38
    54    37.95  =  36.19
    56    38.71  =  36.98
    58    38.71  =  37.74
    60    38.71  =  37.48
    62    37.43  =  36.24
    64    36.19  =  35.04
```

In order to find out what causes this failure, we need to review the formula for calculating `load_avg` as well as `priority` and `recent_cpu`.

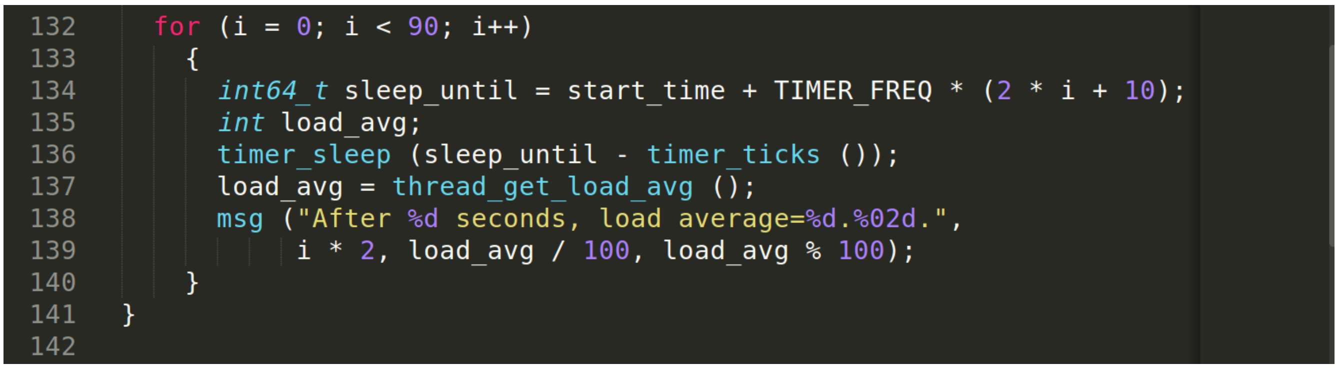
`priority = PRI_MAX - (recent_cpu / 4) - (nice * 2)`

`recent_cpu = (2 * load_avg) / (2 * load_avg + 1) * recent_cpu + nice`

`load_avg = (59 / 60) * load_avg + (1 / 60) * ready_threads`

As we can see from the test results, there is an unexpected increase of `load_avg` between time 38 and 40, resulting in this failure. However, the `load_avg` increases normally before time 38. Therefore, this unexpected change should be attributed to the value of `ready_threads`. Since the number of threads in `ready_list` is related to threads' priorities, the unexpected change in priority may cause this accident. Moreover, priority is calculated using `recent_cpu` and `nice`. It is possible that unexpected changes in `recent_cpu` and `nice` cause this failure. Anyway, we first find out at which line in `mlfqs-load-60.c` does `load_avg` update its value using the formula.

Fig.3: Line of code for updating `load_avg` in `mlfqs-load-60`

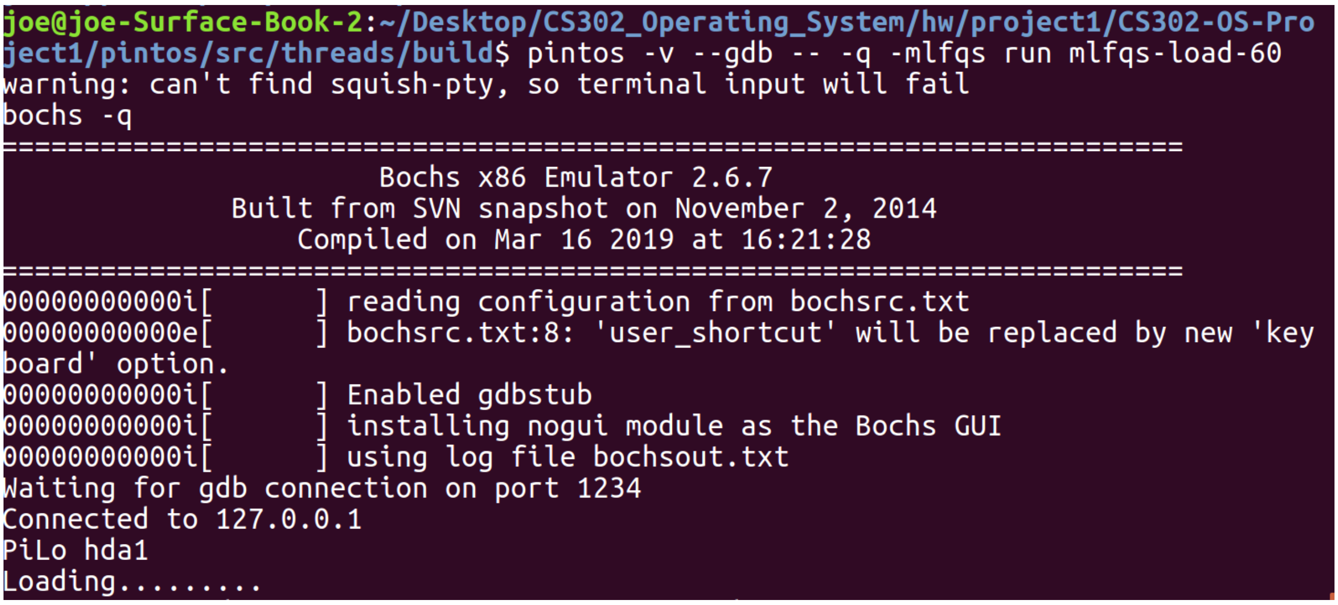


```
132   for (i = 0; i < 90; i++)
133     {
134       int64_t sleep_until = start_time + TIMER_FREQ * (2 * i + 10);
135       int load_avg;
136       timer_sleep (sleep_until - timer_ticks ());
137       load_avg = thread_get_load_avg ();
138       msg ("After %d seconds, load average=%d.%02d.",
139            i * 2, load_avg / 100, load_avg % 100);
140     }
141 }
142
```

As we can see from above, `load_avg` is updated at line 137. We can then use GDB to track the value changes of `recent_cpu` and `nice` at line 137.

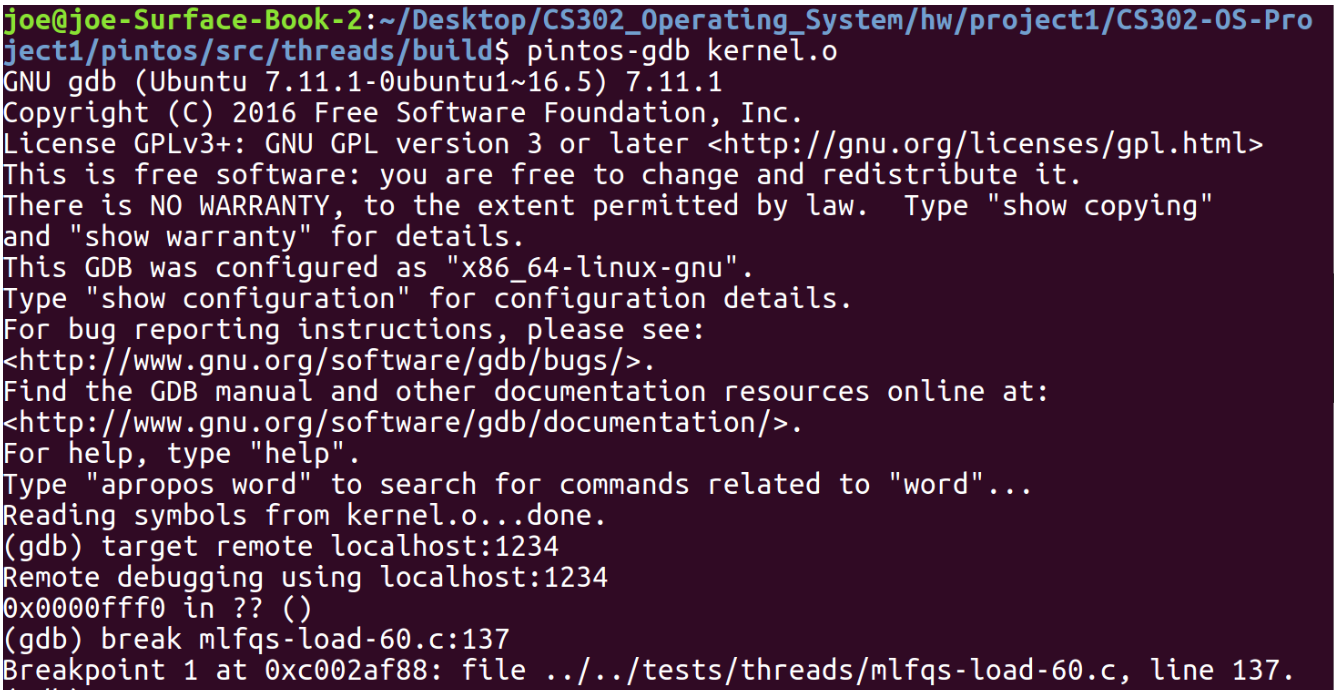
We can open two terminals and enter the following commands in each terminal to set line 137 as a breakpoint in `mlfqs-load-60.c`.

Fig.4: Terminal 1



```
joe@joe-Surface-Book-2:~/Desktop/CS302_Operating_System/hw/project1/CS302-OS-Pro
ject1/pintos/src/threads/build$ pintos -v --gdb -- -q -mlfqs run mlfqs-load-60
warning: can't find squish-pty, so terminal input will fail
bochs -q
========================================================================
                       Bochs x86 Emulator 2.6.7
              Built from SVN snapshot on November 2, 2014
                  Compiled on Mar 16 2019 at 16:21:28
========================================================================
00000000000i[      ] reading configuration from bochsrc.txt
00000000000e[      ] bochsrc.txt:8: 'user_shortcut' will be replaced by new 'key
board' option.
00000000000i[      ] Enabled gdbstub
00000000000i[      ] installing nogui module as the Bochs GUI
00000000000i[      ] using log file bochsout.txt
Waiting for gdb connection on port 1234
Connected to 127.0.0.1
PiLo hda1
Loading........
```

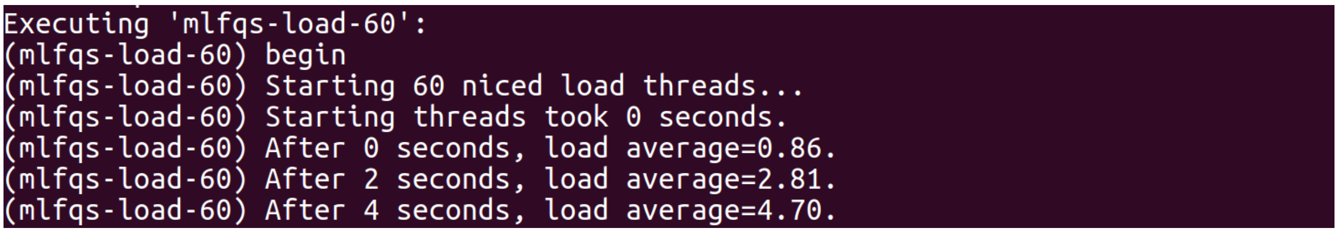
Fig.5: Terminal 2



```
joe@joe-Surface-Book-2:~/Desktop/CS302_Operating_System/hw/project1/CS302-OS-Pro
ject1/pintos/src/threads/build$ pintos-gdb kernel.o
GNU gdb (Ubuntu 7.11.1-0ubuntu1~16.5) 7.11.1
Copyright (C) 2016 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.  Type "show copying"
and "show warranty" for details.
This GDB was configured as "x86_64-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
<http://www.gnu.org/software/gdb/documentation/>.
For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from kernel.o...done.
(gdb) target remote localhost:1234
Remote debugging using localhost:1234
0x0000fff0 in ?? ()
(gdb) break mlfqs-load-60.c:137
Breakpoint 1 at 0xc002af88: file ../../tests/threads/mlfqs-load-60.c, line 137.
```

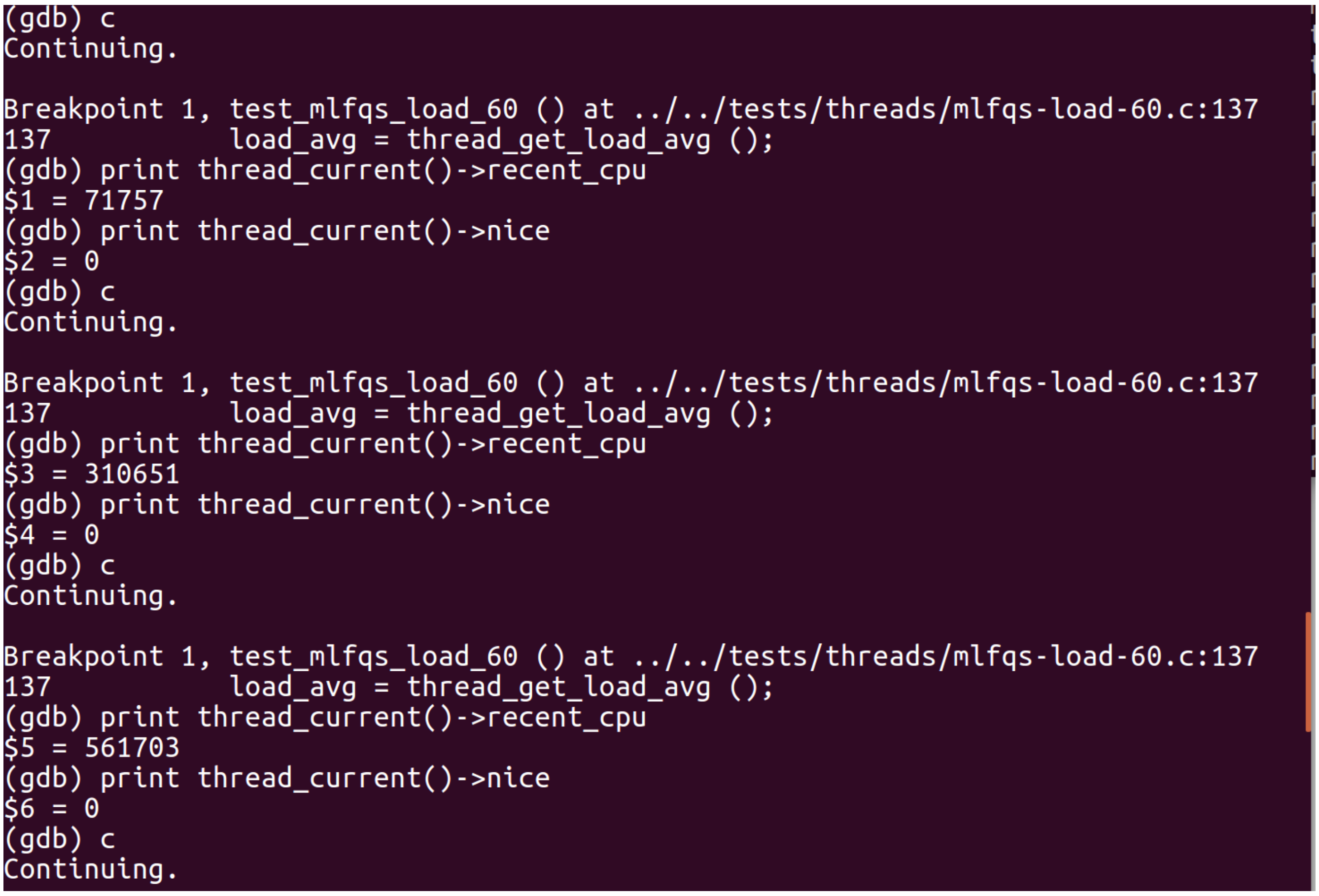
We can then print out values of `recent_cpu` and `nice` during the first several seconds.

Fig.6: Values of `recent_cpu` and `nice` during 0-4 seconds (Terminal 1)



```
Executing 'mlfqs-load-60':
(mlfqs-load-60) begin
(mlfqs-load-60) Starting 60 niced load threads...
(mlfqs-load-60) Starting threads took 0 seconds.
(mlfqs-load-60) After 0 seconds, load average=0.86.
(mlfqs-load-60) After 2 seconds, load average=2.81.
(mlfqs-load-60) After 4 seconds, load average=4.70.
```

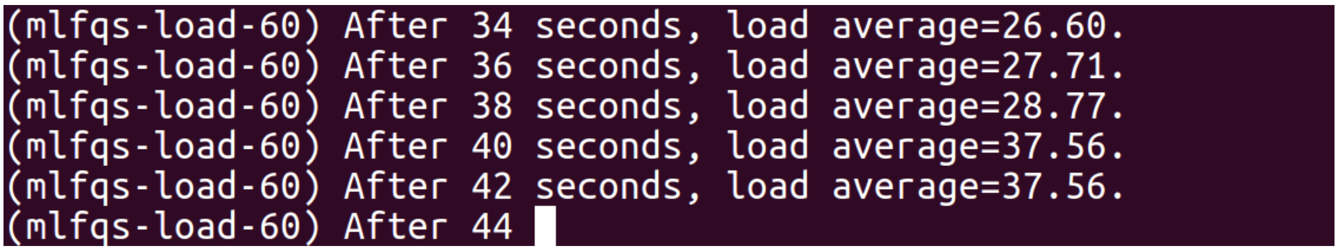
Fig.7: Values of `recent_cpu` and `nice` during 0-4 seconds (Terminal 2)



```
(gdb) c
Continuing.

Breakpoint 1, test_mlfqs_load_60 () at ../../tests/threads/mlfqs-load-60.c:137
137             load_avg = thread_get_load_avg ();
(gdb) print thread_current()->recent_cpu
$1 = 71757
(gdb) print thread_current()->nice
$2 = 0
(gdb) c
Continuing.

Breakpoint 1, test_mlfqs_load_60 () at ../../tests/threads/mlfqs-load-60.c:137
137             load_avg = thread_get_load_avg ();
(gdb) print thread_current()->recent_cpu
$3 = 310651
(gdb) print thread_current()->nice
$4 = 0
(gdb) c
Continuing.

Breakpoint 1, test_mlfqs_load_60 () at ../../tests/threads/mlfqs-load-60.c:137
137             load_avg = thread_get_load_avg ();
(gdb) print thread_current()->recent_cpu
$5 = 561703
(gdb) print thread_current()->nice
$6 = 0
(gdb) c
Continuing.
```

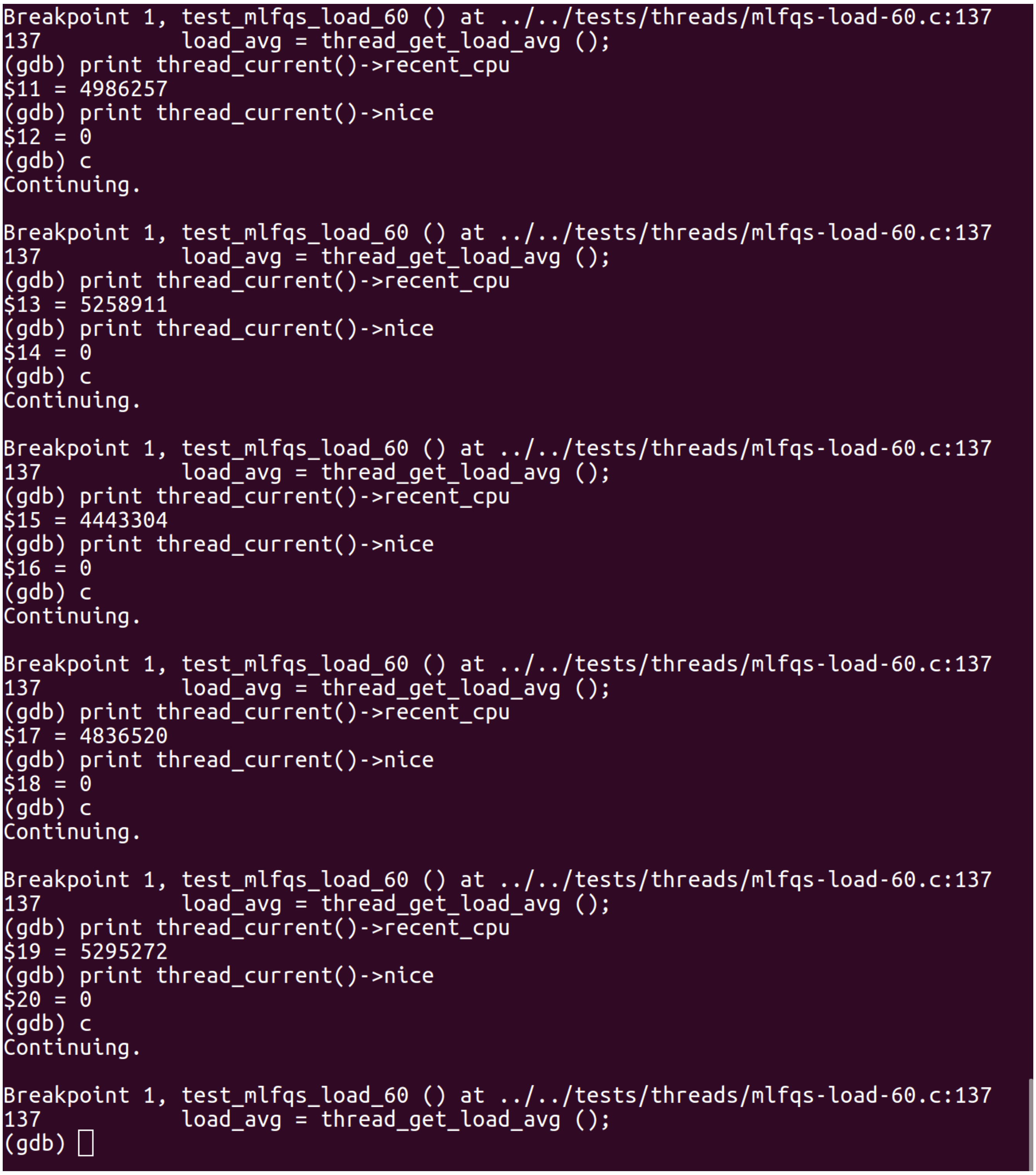
As we can see from above, the value of `nice` remains 0, while the value of `recent_cpu` increases normally. We then skip to the bugged time ticks and see the values of `nice` and `recent_cpu`.

Fig.8: Values of `recent_cpu` and `nice` during 34-44 seconds (Terminal 1)



```
(mlfqs-load-60) After 34 seconds, load average=26.60.
(mlfqs-load-60) After 36 seconds, load average=27.71.
(mlfqs-load-60) After 38 seconds, load average=28.77.
(mlfqs-load-60) After 40 seconds, load average=37.56.
(mlfqs-load-60) After 42 seconds, load average=37.56.
(mlfqs-load-60) After 44
```

Fig.9: Values of `recent_cpu` and `nice` during 34-44 seconds (Terminal 2)



```
Breakpoint 1, test_mlfqs_load_60 () at ../../tests/threads/mlfqs-load-60.c:137
137             load_avg = thread_get_load_avg ();
(gdb) print thread_current()->recent_cpu
$11 = 4986257
(gdb) print thread_current()->nice
$12 = 0
(gdb) c
Continuing.

Breakpoint 1, test_mlfqs_load_60 () at ../../tests/threads/mlfqs-load-60.c:137
137             load_avg = thread_get_load_avg ();
(gdb) print thread_current()->recent_cpu
$13 = 5258911
(gdb) print thread_current()->nice
$14 = 0
(gdb) c
Continuing.

Breakpoint 1, test_mlfqs_load_60 () at ../../tests/threads/mlfqs-load-60.c:137
137             load_avg = thread_get_load_avg ();
(gdb) print thread_current()->recent_cpu
$15 = 4443304
(gdb) print thread_current()->nice
$16 = 0
(gdb) c
Continuing.

Breakpoint 1, test_mlfqs_load_60 () at ../../tests/threads/mlfqs-load-60.c:137
137             load_avg = thread_get_load_avg ();
(gdb) print thread_current()->recent_cpu
$17 = 4836520
(gdb) print thread_current()->nice
$18 = 0
(gdb) c
Continuing.

Breakpoint 1, test_mlfqs_load_60 () at ../../tests/threads/mlfqs-load-60.c:137
137             load_avg = thread_get_load_avg ();
(gdb) print thread_current()->recent_cpu
$19 = 5295272
(gdb) print thread_current()->nice
$20 = 0
(gdb) c
Continuing.

Breakpoint 1, test_mlfqs_load_60 () at ../../tests/threads/mlfqs-load-60.c:137
137             load_avg = thread_get_load_avg ();
(gdb) 
```

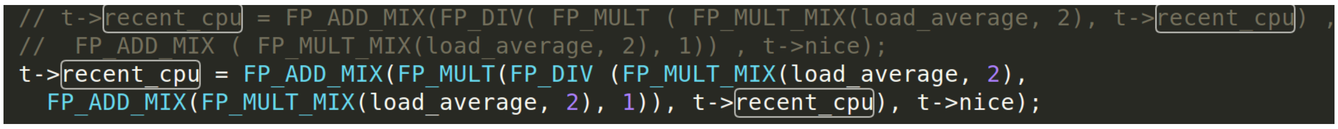
As we can see, the value of `nice` still remains 0, but the value of `recent_cpu` changes from 4986257 to 5258911 then back to 4443304. It follows that `recent_cpu` encounters an **overflow**. Maybe the calculation of `recent_cpu` is problematic. We need to check it out.

Fig.10: The original calculation of `recent_cpu`

```
t->recent_cpu = FP_ADD_MIX(FP_DIV( FP_MULT ( FP_MULT_MIX(load_average, 2), t->recent_cpu) ,
  FP_ADD_MIX ( FP_MULT_MIX(load_average, 2), 1)) , t->nice);
```

As we can see, this calculation is problematic, because it multiplies `load_avg * 2` with `recent_cpu` before dividing it by `load_avg * 2 + 1`. The former multiplication will result in an overflow. Instead, we should do the division before multiplication as follows:

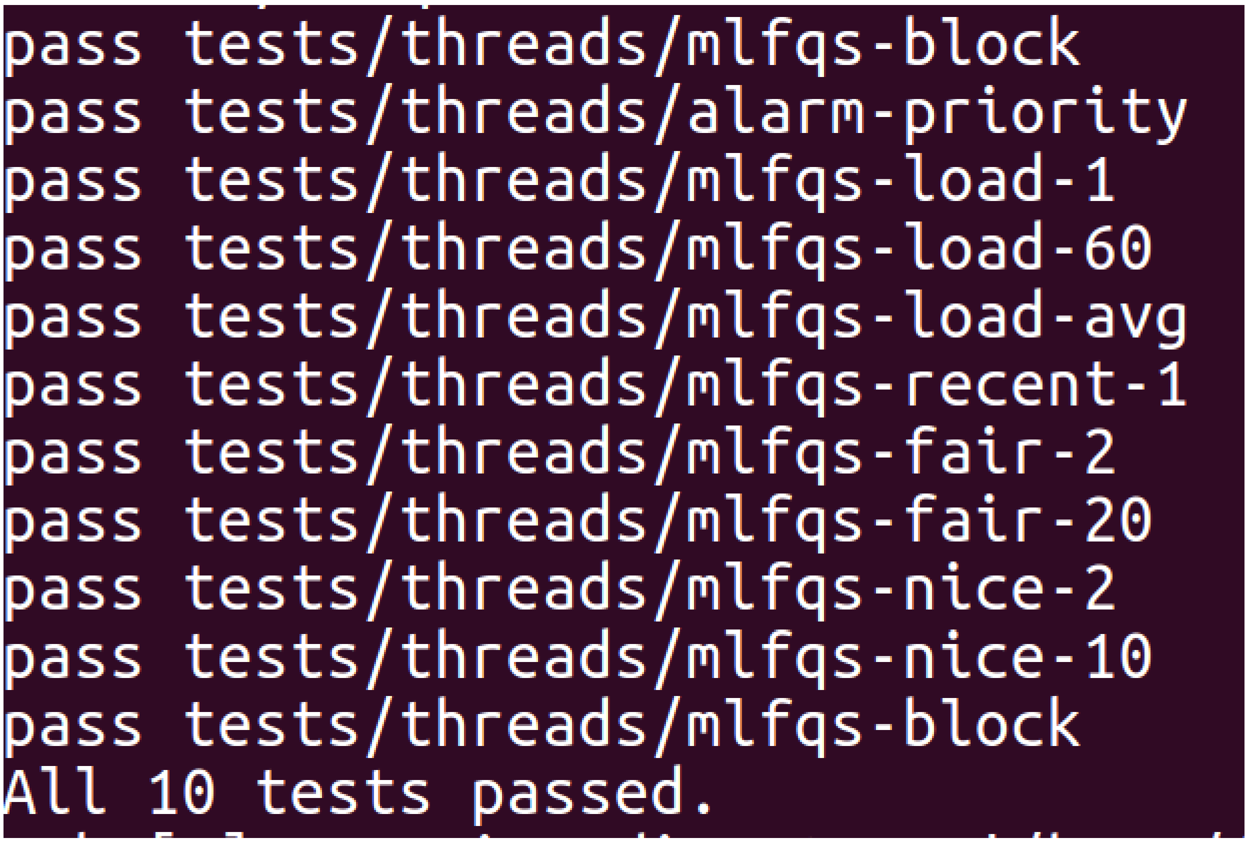
Fig.11: The modified calculation of `recent_cpu`



```
// t->recent_cpu = FP_ADD_MIX(FP_DIV( FP_MULT ( FP_MULT_MIX(load_average, 2), t->recent_cpu) ,
//   FP_ADD_MIX ( FP_MULT_MIX(load_average, 2), 1)) , t->nice);
t->recent_cpu = FP_ADD_MIX(FP_MULT(FP_DIV (FP_MULT_MIX(load_average, 2),
  FP_ADD_MIX(FP_MULT_MIX(load_average, 2), 1)), t->recent_cpu), t->nice);
```

We then run the test again and pass all the tests this time.

Fig.12: The test result after debugging



```
pass tests/threads/mlfqs-block
pass tests/threads/alarm-priority
pass tests/threads/mlfqs-load-1
pass tests/threads/mlfqs-load-60
pass tests/threads/mlfqs-load-avg
pass tests/threads/mlfqs-recent-1
pass tests/threads/mlfqs-fair-2
pass tests/threads/mlfqs-fair-20
pass tests/threads/mlfqs-nice-2
pass tests/threads/mlfqs-nice-10
pass tests/threads/mlfqs-block
All 10 tests passed.
```

Reflection

1. For Tasks 1-3, I was not pushed by ddl. This is because I found the process of exploring documents and cracking tests quite interesting, and I started in advance. I was amazed by the elegant design of pintos as well as those interesting test cases. However, for Task 4, I was pushed by ddl. This is because I was not so familiar with GDB. I did not want to step out my comfort zone and explore GDB until the ddl is really approaching. I think I should try to be more open to new and difficult things in my later study.
2. I would say Tasks 1-3 went well, although Task 2 is challenging. Before really implementing my design, I read through all the test cases and analyzed them carefully one by one. I think this process is quite critical, since it enables me to have a general road map of what I am going to do. I also queried a lot of documentations of pintos when I felt confused. Thanks to the exhaustive and detailed documentation, I got some basic senses of the structure and mechanism of pintos without spending too much time. Therefore, when I really began to implement, I did not encounter too many bugs and unexpected situations.

3. One thing needs to be improved is my desire to learn new things by myself. Actually, I have followed the provided reference and tried to use GDB but failed at the beginning. I felt a little bit frustrated and did not feel like exploring GDB again until being pushed by ddl. During the day before ddl, I started to learn about GDB through the Internet and found it quite useful. I think it will be better if I can take the initiative to explore new things earlier next time.
4. Last but not least, I really appreciate the help from Prof. Bo Tang and all TAs.