**Pintos Project 3 : Threads**

**(Project Report)**

Subject: :[CSE4070] Operating Systems

Faculty Member: Prof. Youngjae Kim

Group Number / Members: 53team, 20151562 신은우, 20151616 최승환

Development period: 2019/11/26 – 2019/11/30

**Project Title : Pintos Project 3 Threads**

**Submission Date : 2019/11/30**

**Members : 신은우, 최승환**

1. **Goal of Development**

* **Systematical, Cooperative Workflow**
  + This project follows workflow of ‘Agile Development’ technique. According to the changing situation, one should elastically change the plans accordingly, without losing consistency of the initial plan.
* **Stable, Reversible version control**
  + This project uses Github through Git commands. Throughout the whole project, turning points or potential danger zones should be noticed to each other teammate. Fatal errors should be prevented by commits and merges between the branches at the right moments, retaining valuable information about previous versions.
* **Single queue priority scheduling**
  + This Operating System should be able to schedule a suggested manner of priority scheduling on single queue.  
    (total of 13 including ‘priority-lifo’)
* **BSD scheduling (additional requirements)**
  + This Operating System should be able to schedule a multi-level feedback queue manner of priority scheduling.  
    (total of 9)

1. **Range and Contents of Development**
   1. **Range**

* **Alarm semaphore**- timer interrupt : check & wake up sleeping threads on ticks  
  - timer sleep : sleep thread using semaphore and put them into sleep queue  
  - sleep queue : push and pop from sleep queue
* **Priority scheduling**- aging : alter priority as time passes  
  - ready queue : manage ready queue and wait queue
* **BSD scheduling**- recent\_cpu : calculate recent\_cpu value per 1 second and add 1 recent\_cpu value per 1 tick  
  - niceness : set and use nice value to calculate other parameters  
  - load average : calculate load average every second.
* **Overall debugging**- test and reimplement modules according to test standards
  1. **Contents**
* **Timer interrupt and Ticks**
  + Timer interrupt is an interrupt to check the current time. It is an external interrupt called every tick. In the ‘timer\_interrupt’ function, a variable called ‘tick’ is increased. This ‘tick’ represents the time spent after booting. Also identical to the number of ‘timer\_interrupt’ called after booting. Using this variable, current time can be checked. Every time you check the current time, all sleeping threads should be checked. This is done by ‘sleep\_queue’ that is implemented and managed throughout ‘timer\_interrupt’ and ‘timer\_sleep’.
* **Ready queue / Wait queue**
  + Ready queue is where all threads that are ready to be scheduled, while wait queue is where all threads waiting (waiting for I/O or Sleeping) resides. When time slice expired for a thread that still hasn’t finished its job, the job will be pushed into the ready queue, while calling sleep will cause it to be pushed into wait queue. Managing the running thread, ready queue, and wait queue was the main job of this project.
* **Schedule**
  + High priority thread always preempts low priority threads. So each time to insert thread in ready list, maintain descending order by priority. Also to prevent low priority from excluding on using cpu, change priority properly.
* **Fixed point**
  + Although pintos doesn’t provide Real value arithmetic, it is needed when calculating recent\_cpu & load average. so implemente real arithmetic by integer variable, considering left-side 17 bits integer value and right-side 14bit are fraction value.

1. **Schedule and Methodology of Development**
   1. **Scheduling**

* **Timer (11.26 ~ 11.27)**
* **Priority scheduling (11.26 ~ 11.29)**
* **BSD scheduling (11.29 ~ 11.29)**
* **Debugging (11.27 ~ 11.30)**
  + **Timer (11.27)**
  + **Priority scheduling (11.29)**
  + **BSD scheduling (11.29 ~ 11.30)**
* **Github branch management (11.26 ~ 11.30)**
  1. **Methodology**
* **Timer**  
  - **Alarm semaphore** Original Implementation of ‘timer\_sleep’ was a simple busy waiting. This causes a lot of problem on multithreaded environment. This implementation was replaced by semaphore technique to utilize wait queue and sleep.  
  - **Timer interrupt** ‘timer\_interrupt()’ is where the actual value of ‘ticks’ is increased. This means the system’s time flows according to ‘timer\_interrupt’. To wake up sleeping threads, ‘timer\_interrupt’ checks the wake up time of all the sleeping threads in the sleep\_queue.  
  - **Sleep queue management** Checking the wake up time of all threads is a clear way to handle sleeping threads. But it might cause a big overhead if the sleep\_queue becomes larger, so a technique is used to maintain queue in a restricted time. In ‘timer\_interupt()’, only the first element of the sleep\_queue will be checked every tick. This is possible if all nodes in the sleep queue remains sorted at all times. ‘list\_insert\_ordered()’ solves this problem, because on inserting into sleep queue, all nodes will be checked and be put in a sorted manner. This caused the OS to save huge amount of time on every call of ‘timer\_interrupt()’.
* **Priority scheduling**  
  - **Priority implementation** ‘Higher priority threads will always be executed first.’ To implement this, threads ready to ‘run’ will be inserted into a ready\_list in a decreasing order. Every time the running thread returns to ready\_state, the first thread of the ready\_list will be selected by the OS. Therefore, inserting it in a decreasing order will always make sure the highest priority will always be executed first.   
  - **Aging** The downside of Priority scheduling is a risk of starvation, where high priority will keep the CPU away from the low priority thread for a long time. Solution for this problem is called aging. Aging technique increases the ready state threads as time goes. This will ensure, any low priority thread will eventually be able to use the CPU
* **BSD scheduling  
  - Niceness** Nice is the value which decreases priority when recalculating priority. Bigger niceness will cause other threads to take more CPU time. Default nice value is set to 0 and can be set or get through certain functions  
  - **Load\_average** Load average saves the average number of threads in the ready\_list. This value influences the value of recent\_cpu, eventually influencing the priority of a thread. This value is used as a global variable and is changed every second. For efficient calculation, moving average technique is used and it fomula is as below.  
   **load\_avg = (59/60) \* load\_avg + (1/60) \* ready\_run\_threads**  
   Value is saved in a real value using the fixed point calculation in the Development part.  
    
    
  - **Recent\_cpu** This value measures how much cpu time the thread got recently. This value increases by 1 every tick when a thread is running on the CPU. Every second, currently executing or ready threads will recalculate its recent CPU according to this formula

**recent\_cpu = (2\*load\_avg)/(2\*load\_avg + 1) \* recent\_cpu + nice**

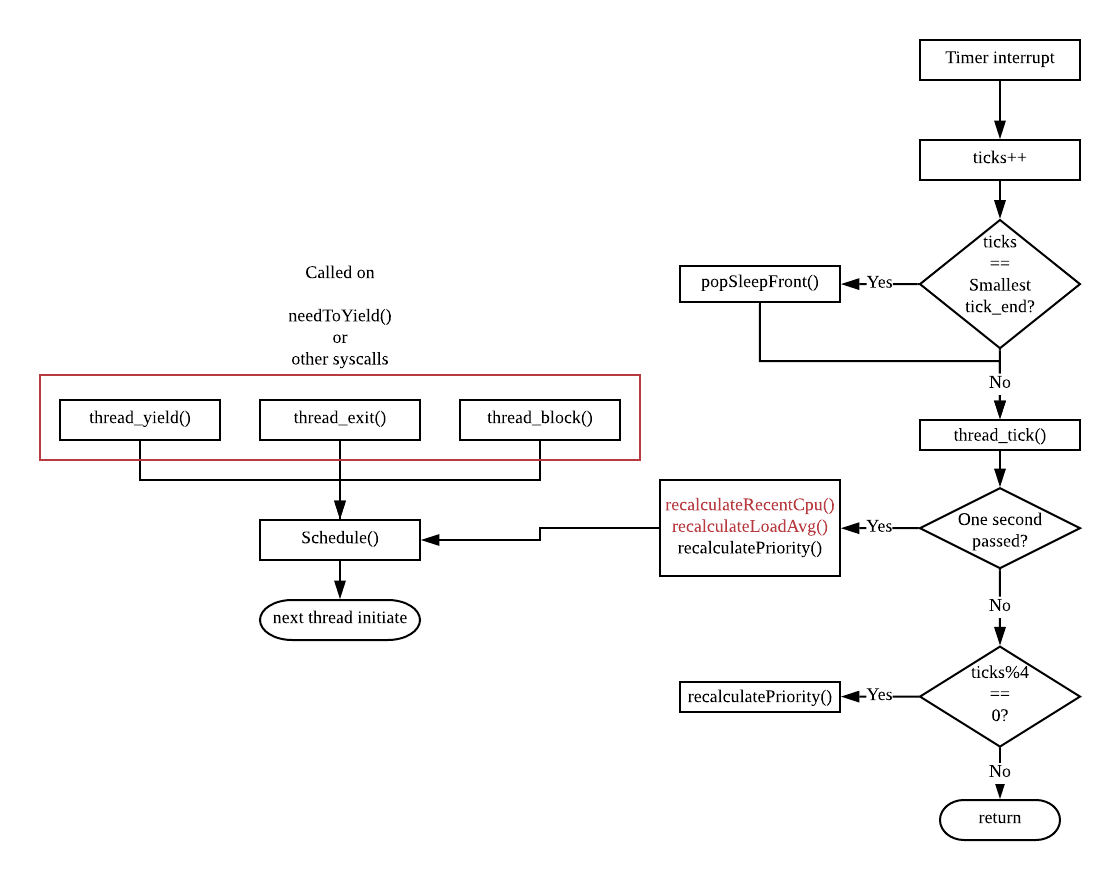
Any thread that doesn’t occupy the CPU decreases its recent\_cpu constantly. Nice value will be added on calculating the priority. recent\_cpu value is also a fixed point value.  
- **Priority** The Final output of priority calculation. Higher priority means, such thread will always be scheduled before lower priority threads. Priority value varies from 0 to 63, and is recalculated every 4 ticks.

**priority = priority\_max(63) - (recent\_cpu / 4) - (nice \* 2)**

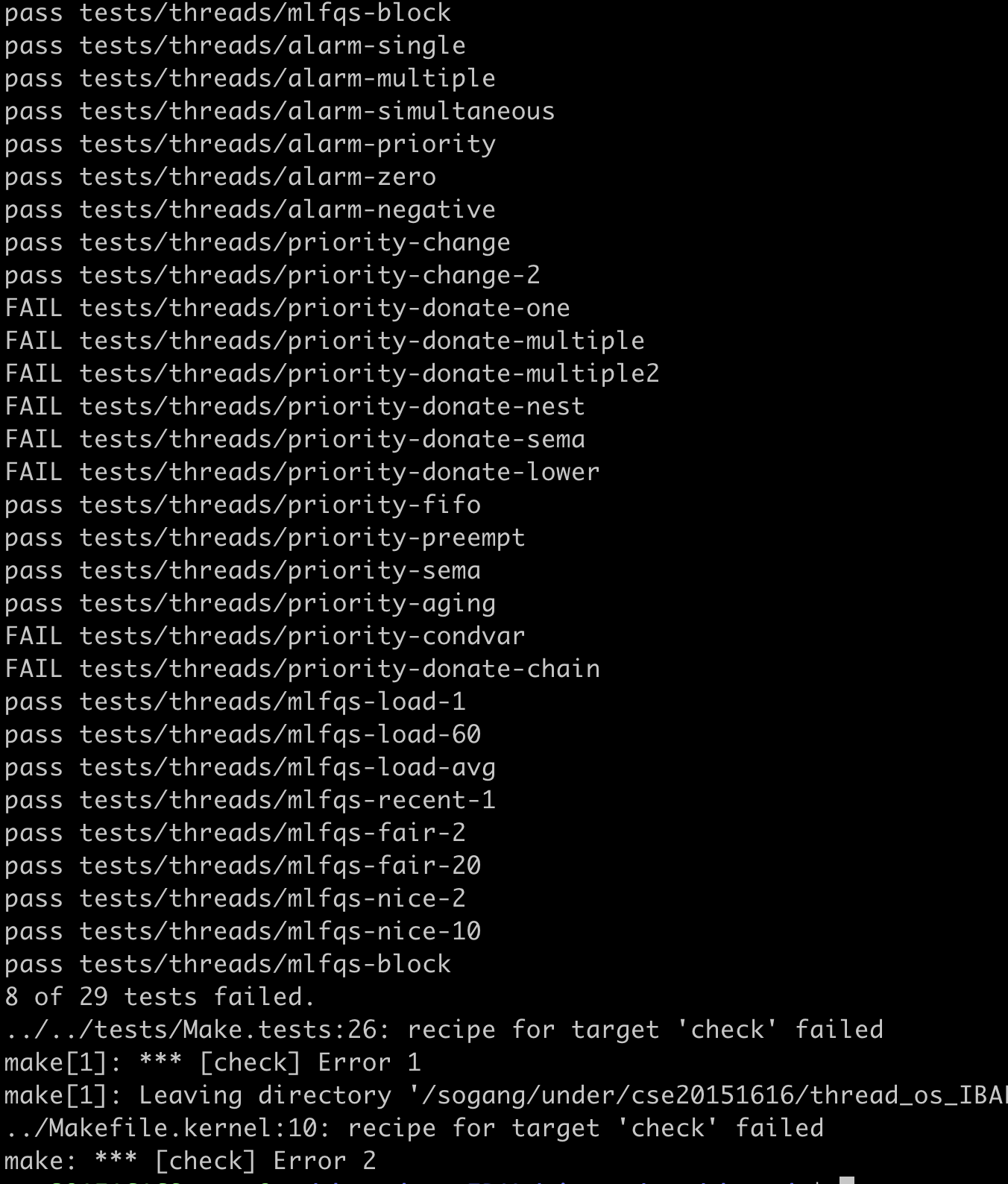
higher recent\_cpu and nice value decreases priority. coefficient of recent\_cpu and nice value is more of a experimental value rather than theoretical value by proof. If recalculatePriority() did not result in the thread’s priority higher than any other ready threads, thread will yield CPU and be pushed into the ready\_list **- Real arithmetic**

* **Github branch management** Github branch management was done in a rather cautious manner since this project was a big, sensitive one. All sub-projects were implemented and tested on an individual branch and merged to each other afterwards.  
   Push on master branch was done only when a module was fully capable of its job. Less merge on branches were needed, complicated division of issues were efficiently managed on two local work directories, and the final push to master branch was also merged without any conflicts.  
  (Though conflicts doesn’t necessarily mean bad project flow, it still is a laboring job, therfore was minimized through all parts of project)  
  1. **Role Sharing**
* **Timer : 최승환**
* **Priority scheduling : 신은우**
* **BSD scheduling : 신은우**
* **Debugging : 최승환, 신은우**
* **Github branch management : 최승환, 신은우**

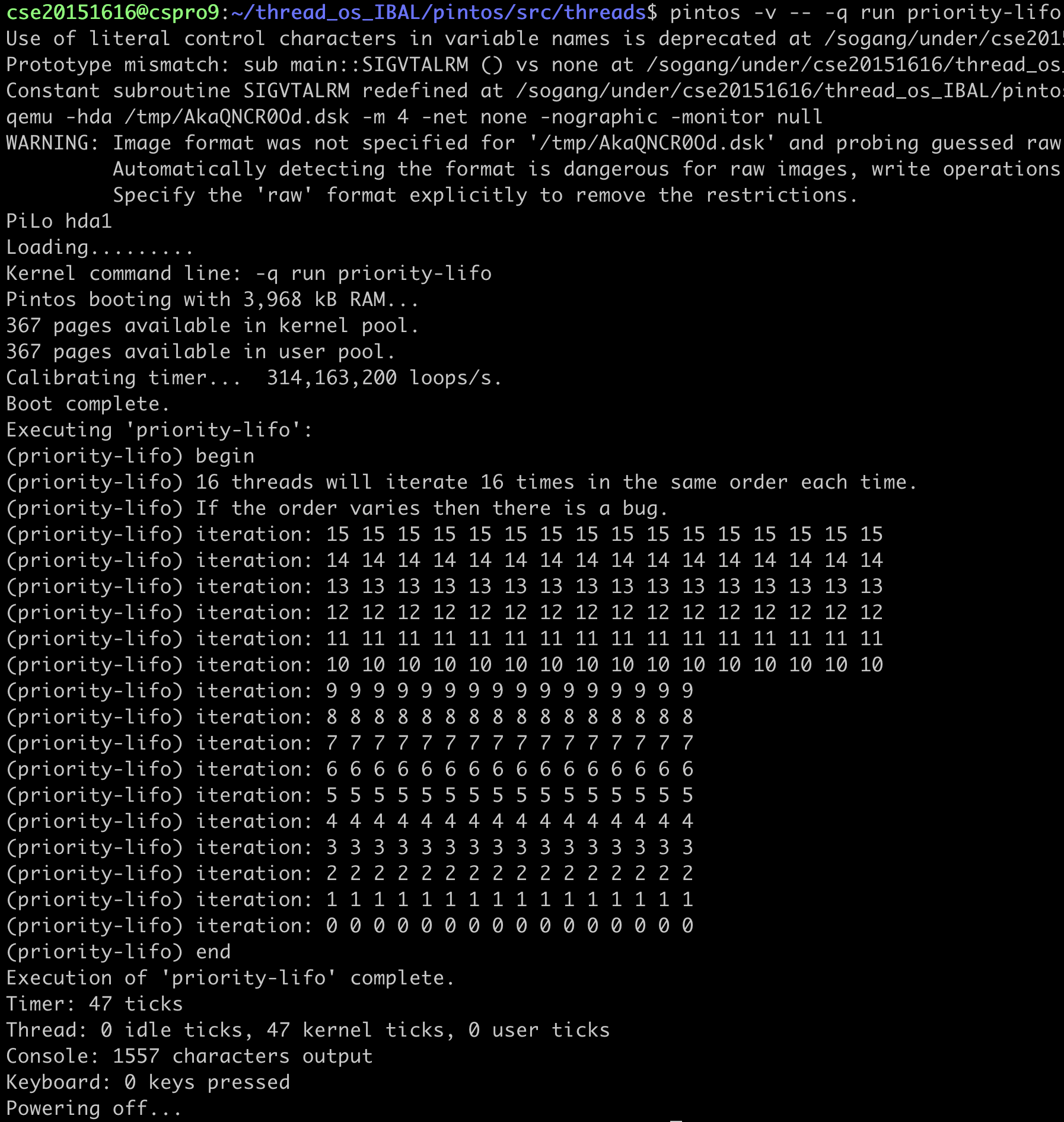
1. **Result of Research**
   1. **Composition of Contents**

****

* 1. **Development of Contents**
* **Timer**  
  - **timer\_interrupt()** It is a external interrupt called every tick. This function cannot be stopped without turning off the interrupts by ‘intr\_disable()’. After this function increase the global variable ‘tick’, it will compare ‘tick’ to the first node of the ‘sleep\_list’ if the first node has to wake up now, it will pop the node from the list and ‘sema\_up()’ the thread. However, there could be multiple threads that needs to wake up at the same time. This is done by popping every node until the first node’s wake-up time does not match ‘ticks’  
  - **timer\_sleep()** This function puts a thread’s sema into the ‘sleep\_list’. Upon pushing to the list, ‘sema\_down()’ the thread’s sema. Another characteristic of this function is that it pushes the sleep thread in a sorted order.
* **Priority scheduling**  
  - **Priority implementation** When a thread is pushed to a ‘ready\_list’, it should be inserted in a decreasing order. This is because a thread will be choosed from the ‘ready\_list’ by ‘list\_pop\_front()’. Threrefore ‘list\_insert\_ordered()’ will be used to maintain the decreasing order on insert. Compare function was also implemented to compare the thread’s two priorities. Afterwards, thread will be pushed into the ready\_list with thread\_yield() and thread\_unblock().(originally implemented with list\_push\_back())  
  - **Aging** Implemented to prevent the starvation of low priority CPU. Each thread in the ‘ready\_list’ increases its priority by 1 every second(100 ticks). After recalculating the priority on aging, sort the threads in the ready\_list again.
* **BSD scheduling**- **Niceness** Because all thread has nice value, added new variable to maintain nice value to thread struct, and each time to create new thread initiate nice value to 0. Also implemented thread\_set\_nice to set nice value of current thread and thread\_get\_nice() to give nice value of current thread.  
  - **Load\_average**Declared new global variable to store load\_average value. And recalculate load\_average value by formula as i wrote above. Calculation is implemented in timer\_intterupt which is function called on each time interrupt.- **Recent\_cpu** Because each thread has recent\_cpu, add new variable to thread struct. every one tick increase recent\_cpu value by one and every four tick recalculate all recent\_cpu value of each ready and running thread. **- Priority** Calculated the priority of all threads every four ticks in ‘timer\_interrupt’ which is called every timer interrupt. Priority order can also change by changing nice value, so at each time recalculate priority of thread which change nice value. After all calculation finished, Check that priority of current running thread is not the highest priority among ready threads. If it is true, highest priority thread will occupy the CPU.
  1. **Test and Evaluation of Contents  
     - Result of ‘make check’**

****

* **Result of ‘priority-lifo’**

****

In the priority-lifo code, threads with 16 priority will be created. However, the thread with smallest priority 0 will be created first, and will create consequently to the highest priority thread of 15.  
 Since, the threads will be scheduled in decreasing order of priority, thread will be scheduled from priority 15, 14, 13, . . . . , 0 . Therefore, the thread will be executed in a LIFO order. Reason why this works for priority scheduling is because right after the yield of thread-creating(main) thread, scheduling policy will determine the next thread to occupy the CPU.   
 However, the priority-lifo tests this same set of test 16 times. Reason for this is because some policies might differ the result on multiple tests. (such as random policy or round-robin with interrupts). 16 times of consequent test would be enough to ensure that the scheduling policy always produce the same result.

The Result of our pintos shows that highest priority always comes first.

Why?  
 These thread names(thread ids) were saved at variable named ‘output’(which is also named as ‘op’) in the simple\_thread\_func() after yielding the main thread. So When print value saved in ‘output’ the faster the thread was scheduled, the faster its name would be printed on the screen. Actually the thread’s id is identical to string of its priority. Since the printing operation is designed to print the output of the thread’s name, output of 15 to 0 means thread with the highest priority always came out first. (if thread name is “15”, its priority is 15) So this is the correct result of priority scheduling.

1. **Etc.**
   1. **Level of Contribution**

* **신은우 (50%)**
* **최승환 (50%)**
  1. **Thoughts for Project**
* **신은우** This is last of all projects. When I started this project, I set my goal as to only referring pintos manual and provided project guide pdf not the code documents on internet. I’m glad to achieve my goal. It was not an easy way, but my knowledge about os advanced by reading manual and pintos code many times. This project was worth devoting my time on solving the issues because I came to know not only about knowledge of os, but also how to debug codes, what I should do on team programming project and a good attitude to face uneasy problem. This will be a important asset when I proceed other projects in future.
* **최승환** Pintos project is always the most representative project of my major course. It must be because it requires the developer two abilities at once. First is a high level implementation of C. Second is the ability to structurize what you want to design. I always had a chance to improve my C implementation skills, but not the second ability.  
   Even though my github history has not been very short, I always found myself using only a few commands to control versions of my own branch. This time it took a lot more than that. I took more time to manage the branches and group multiple jobs in the same module at each branches.  
   Surprisingly it let us save a lot of time debugging and merging our code. Not to mention it actually worked to the standards we set.  
   Job hasn’t been easy through all parts of the project but it was the best experience to construct a systematic workflow with my teammate.