Objective:
1. This homework is designed to analyze the reading comprehension ability of a student in browsing through and understanding the PintOS source code.
2. Team submission is advocated in-order to promote better communication of technical concepts among team members, develop a sense-of-oneness, help understand each other’s ability, and be responsible for each other in a group setting.

Total Points: 100 (+15 points bonus)
Submission deadline: Wednesday- June 20, 2018 (Inclass)
Submission guidelines: One submission per team
Grading criteria: All members of the team receive the same score
2 bonus questions in this homework to challenge you to get more than 100 points.
You can (add/negate) 5 homework buffer days of this semester towards the submission of this homework.
Please note the academic integritry policy at: http://academicintegrity.buffalo.edu/policies

For pintos documentation that would help answer these questions, refer the extensive documentation at: https://web.stanford.edu/class/cs140/projects/pintos/pintos.html. But the most viable way to answer the questions in a time-efficient manner is to read and run the source code and by discussing these questions with your team members. The reading compression ability of the source code is an essential requirement before beginning Project/Checkpoint 1.

1. In PintOS, how do you write your own function and have it invoked on all threads? Write down an example.

   We can call thread_foreach() to invoke our own function on all threads.

   Example of invoking on all threads:
   thread_foreach(self_defined_function, NULL);

   Example of self_defined_function:
   void self_defined_function(struct thread *t, void *aux_arguments) {
     // ...
   }

   Thread_foreach is defined:
   /* Invoke function ‘func’ on all threads, passing along ‘aux’.
   This function must be called with interrupts off. */
   void thread_foreach (thread_action_func *func, void *aux) {
     struct list_elem *e;
     ASSERT (intr_get_level () == INTR_OFF);
     for (e = list_begin (&all_list); e!= list_end(&all_list); e = list_next (e)) {
       struct thread *t = list_entry (e, struct thread, allelem);
       func (t, aux);
     }
   }
2. What is the function of ‘timer’ in PintOS? (5)

Function that tries to simulate a specific amount of time by looping. Time measured by loops, loops per tick, and timer ticks. These are combined in functions that for example can be used to put a thread to sleep for a given number of ticks, or used to create specific timed delays. Obviously, it can also be used for anything else a timer would be useful for, and in an operating system there are lots of uses. Code found in devices/timer.c

3. What are interrupt handlers? What does timer_interrrupt handler do in PintOS? (5)

An interrupt notifies the CPU of some event. Interrupt handlers are executed every period of time to interrupt threads. The main job of interrupt handler is to call the function registered for handling the particular interrupt. (If no function is registered, it dumps some information to the console and panics). It also does some extra processing for external interrupts.

Additionally, as stated in pintOS documentation, if the timer interrupt handler takes too long, then it will take away most of a timer tick from the thread that the timer interrupt preempted. When it returns control to that thread, it therefore won’t get to do much work before the next timer interrupt arrives. That thread will therefore get blamed for a lot more CPU time than it actually got a chance to use.

4. What is the function call for threads in PintOS to block itself from executing? Similarly, what is the call to wake a thread up? (10)

`void thread_block(void)` in thread.c makes a thread to block itself from executing.
`void thread_unblock(struct thread *t)` makes a sleeping thread wake up.

5. Name the different states of a process as discussed in class. Similarly, what are the different states of a thread in PintOS? (5)

Different states of a process are: a) new b) running c) waiting 4) ready, and 5) terminated. Different states of a thread in PintOS are: 1) THREAD_RUNNING, 2) THREAD_READY, 3) THREAD_BLOCKED, 4) THREAD_DYING

6. How would you avoid ‘busy waiting’ in timer_sleep() function in devices/timer.c? Explain your intuition. (15)

Busy waiting in timer_sleep() could be avoided using thread_block() and thread_unblock() instructions. **Intuition:** We disable the interrupts to ensure serialization, block the thread in thread_block() and use a timer_interrupt() for each elapsed tick. On each tick, we call a wake up function on all threads using thread_foreach(), the wake up function checks if the thread is blocked and decrements the sleep tick variable. If the tick count runs out for a thread, we unblock that thread using thread_unblock().
7. **What does alarm-multiple test do?**

Running the command “pintos run alarm-multiple” passes the argument run alarm-multiple to the PintOS-kernel. In these arguments, run instructs the kernel to run a test on alarm-multiple. PintOS boots and run the alarm-multiple test program then the alarm-multiple test outputs a few screen-fulls of text. That text provides information about RAM memory, pages available in kernel pool and user pool. Alarm-multiple test executes and it creates 5 threads to sleep 7 times each. Alarm-multiple test displays that thread 0 sleeps 10 ticks each time, thread 1 sleeps 20 ticks each time, thread 2 sleeps 30 ticks each time, thread 3 sleeps 40 ticks each time, thread 4 sleeps 50 ticks each time. Alarm-multiple test also outputs if execution is successful or not. Alarm-multiple test gives the output for iteration count of products and sleep duration for each thread. Product of iteration count and sleep duration will be shown in non-descending order if execution is successful.

8. **How many tests are currently passing in the threads directory of PintOS?** (You might want to run make check)

20 of 27 tests failed. 7 test passed.

9. **What are external interrupts in PintOS? Give one example of external interrupt in PintOS and the part of the source code where you found it. What are their special characteristics in-terms of pre-emption and nesting?**

External interrupts are interrupts that caused by events outside the CPU. Which basically refers to interrupts that are caused by other hardware. For example, I am causing an external interrupt as I am typing on the keyboard. When I press on a button on the keyboard, an electrical signal is sent to the processor alerting the OS that a button has been processed. In the directory “pintos/src/devices”, you can find multiple external interrupt such as timer, speaker, and input, and kdb (keyboard example). External interrupts are asynchronous and can be invoked at any time that interrupts have not been disabled. External interrupts allow for preemption, in other words, they can temporarily interrupt the task being carried out by the CPU. In PintOS, an external interrupt handler effectively monopolizes the machine and delays all other activities. Nesting is not allowed in PintOS, because only one external interrupt may be processed at a time. “Neither internal nor external interrupt may nest within an external interrupt handler” – PINTOS Manual.

10. **How many interrupts found in PintOS? Name 5 of them. Where did you find them?**

PintOS runs upon simulated 8086 architecture which has interrupts divided between mask-able, non mask-able hardware and software interrupts. We have the NMI pins in 8086 and the INTA for the acknowledge. We can have 0-255 software interrupts.

Some of the interrupts in pintOS are:

- NMI interrupt
- OF Overflow Exception
- DB Degub exception
- DE Divide error
- MF x87 FPU Floating-point error
These were in `interrupt.c` The `intr_init()` sets up the Interrupt Descriptor Table with each of the 256 points of entry to threads.

Below are the 8086 ones for interrupt from the 8086 instruction set.

- **INT** – Used to interrupt the program during execution and calling service specified.
- **INT0 0** – Used to interrupt the program during execution if OF = 1
- **IRET** – Used to return from interrupt service to the main program.

11. **How many processors are simulated in PintOS? Would inhibiting interrupts for critical section execution work in PintOS environment? Explain.**

   There is only 1 processor. Exploring the source code, it seems that the threads are running sequentially. (Additionally, if we try to switch the setting to 2 or more than 2 processors in the virtual box, it will show “invalid setting”)

   Yes, inhibiting interrupts for critical section execution work in PintOS environment. Without interrupts, the critical section will occur but break synchronization.

12. **(Bonus) What are the advantages of spinlock over mutual exclusion?**

   Spinlocks avoid overhead from operating system process, rescheduling, and context switching. Spinlock is more efficient than mutex if threads are likely to be blocked for short period. In short, the advantages of spinlock over mutual exclusion: Spinlocks are useful when you already know that wait time is going to be low.

13. **(Bonus) Is PintOS a pre-emptive or non-preemptive kernel? Explain your understanding.**

   The PintOS kernel is preemptible as it allows kernel threads to be interrupted at any time. This can be changed by disabling interrupts. The preemption process relies on interrupts to “pause” the current thread, and interject a new one of its choosing running the interrupted thread later. If the current thread can’t be paused/interrupted, then preemption can’t be done.