세계 속의 GIST 이공계 대학

Operating System Project 2

Release Date: 2024-03-25

Due Date: 2024-04-08

GIST

Gwangju Institute of Science and Technology

0. Criteria

- A. As mentioned in the first project criteria announcement, HW2 will be graded by using 'make check' command in PintOS(see the HW1 criteria document for more information on 'make check' command). In HW2, we use 'make check' in '/src/threads' folder.
- B. If you fully implemented including the advanced scheduler (problem 1 and 2), you will be able to pass all the tests in the threads folder.
- C. If you did not solve problem 2 and just solve problem 1 perfectly, you would be able to pass all tests except those prefixed as 'mlfqs-'.
- D. One failed test does not mean that you cannot get a project score. Since the score is proportional to the number of tests being passed, try to pass as many tests as possible, but do not have to pass all tests.
- E. Here is the grading rule
    - Problem 1
        - Base point: 2pts
        - Report: 2pts
        - Test result: # of passed tests / # of tests * 2pts
        -
    - Problem 2
        - No base point
        - Report: 2pts
        - Test result: # of passed tests / # of tests * 3pts
- F. If you have any questions, please mail me at **minwook-lee@gm.gist.ac.kr**

1. Introduction

- In this assignment, your job is to extend the functionality of thread system to gain a better understanding of **synchronization** problems.
- You will be working primarily in the threads directory *"~/pintos/src/threads*" for this assignment, with some work in the devices directory "~ */pintos/src/devices*" on the side. Compilation should be done in the threads directory "~ */pintos/src/threads*".
- Before you start this project, you should at least skim the materials about **Loading, Memory Allocation, and Synchronization**. To complete this project you will also need to understand well about **BSD Scheduler.**

Problem 1.

- **Implementing Priority Scheduling** (required, 6pts, base point 2pts)
- Requirement 1:
  - Implement **priority scheduling**.
    - Thread in ready list with the highest priority is always selected to run.
    - If a thread is added to the ready list with a higher priority than the running thread, yield the CPU immediately to the new thread.
    - Threads waiting on a semaphore, lock, or condition variable should have the highest priority waiting thread wake up first.
  - Use source code thread.c and synch.c in “~/pintos/src/threads/”.
  - Thread priorities range from PRI_MIN (0) to PRI_MAX (63). Lower numbers correspond to lower priorities, so that priority 0 is the lowest priority and priority 63 is the highest. The initial thread priority is passed as an argument to thread_create(). If there is no special reason to choose another priority, use PRI_DEFAULT (31).

Problem 1.

- **Implementing Priority Scheduling** (required, 6pts, base 2pts)
- Problem Manual: Current state
  - Check ready_list, to see current state
  - ready_list is called by list_push_back() which just push element to end of the list.
  - Also, in the schedule(), switch_thread() is written in assembly, send the control of CPU from cur thread to next thread and return cur thread address.
  - See where the return value of switch_thread() go.
  - next_thread_to_run do a list_pop_front
  - So we can easily understand that current state is Round Robin

```
static struct thread *
next_thread_to_run (void)
{
  if (list_empty (&ready_list))
    return idle_thread;
  else
    return list_entry (list_pop_front (&ready_list), struct thread, elem);
}
```

Problem 1.

- **Implementing Priority Scheduling** (required, 6pts, base 2pts)
- Problem Manual: How to solve?
  - list.c -> list_insert_ordered() -> list_insert()
  - Is this run exactly how we want?
  - Insertion is okay. How about running thread?

Problem 1.

- **Implementing Priority Scheduling** (required, 6pts, base 2pts)
- Requirement 2:
  - Implement **priority donation**
    - Consider all different situations in which priority donation is required. Be sure to handle multiple donations, in which multiple priorities are donated to a single thread. Also handle nested donation: if H is waiting on a lock that M holds and M is waiting on a lock that L holds, then both M and L should be boosted to H's priority. If necessary, you may impose a reasonable limit on depth of nested priority donation, such as 8 levels.
    - Implement the following functions that allow a thread to examine and modify its own priority. Skeletons for these functions are provided in "*~/pintos/src/threads/thread.c* ".

      (1) Function: void thread_set_priority *(int new_priority)*

      Sets the current thread's priority to new_priority. If the current thread no longer has the highest priority, yields.

      (2)Function: int thread_get_priority *(void)*

      Returns the current thread's priority. In the presence of priority donation, returns the higher (donated) priority.
  - You must implement priority donation for locks and do not need to implement priority donation for the other Pintos synchronization constructs.

Problem 1.

- **Implementing Priority Scheduling** (required, 6pts, base 2pts)
- Problem Manual: Current state
  - lock, semaphore, condition
  - sema_up -> waiters -> which one we should pick?

```
void
sema_down (struct semaphore *sema)
{
  enum intr_level old_level;

  ASSERT (sema != NULL);
  ASSERT (!intr_context ());

  old_level = intr_disable ();
  while (sema->value == 0)
    {
      list_push_back (&sema->waiters, &thread_current ()->elem);
      thread_block ();
    }
  sema->value--;
  intr_set_level (old_level);
}
```

```
void
sema_up (struct semaphore *sema)
{
  enum intr_level old_level;

  ASSERT (sema != NULL);

  old_level = intr_disable ();
  if (!list_empty (&sema->waiters))
    thread_unblock (list_entry (list_pop_front (&sema->waiters),
                                struct thread, elem));
  sema->value++;
  intr_set_level (old_level);
}
```

Problem 1.

- **Implementing Priority Scheduling** (required, 6pts, base 2pts)
- Problem Manual: How to solve?
  - lock is semaphore with holder. We don't need extra steps.
  - condition variables
  - Same?

```
void
cond_wait (struct condition *cond, struct lock *lock)
{
  struct semaphore_elem waiter;

  ASSERT (cond != NULL);
  ASSERT (lock != NULL);
  ASSERT (!intr_context ());
  ASSERT (lock_held_by_current_thread (lock));

  sema_init (&waiter.semaphore, 0);
  list_push_back (&cond->waiters, &waiter.elem);
  lock_release (lock);
  sema_down (&waiter.semaphore);
  lock_acquire (lock);
}
```

```
void
cond_signal (struct condition *cond, struct lock *lock UNUSED)
{
  ASSERT (cond != NULL);
  ASSERT (lock != NULL);
  ASSERT (!intr_context ());
  ASSERT (lock_held_by_current_thread (lock));

  if (!list_empty (&cond->waiters))
    sema_up (&list_entry (list_pop_front (&cond->waiters),
                          struct semaphore_elem, elem)->semaphore);
}
```

Problem 2.

- **Implementing Advanced Scheduler** (Not required, If you solve, it will be considered as an extra point of 5pts. The extra point will be used to supplement your overall score. No base points here.)
- Requirement:
  - Implement a multilevel feedback queue scheduler similar to the BSD scheduler to reduce the average response time for running jobs on your system
    - Like the priority scheduler, the advanced scheduler chooses the thread to run based on priorities. However, the advanced scheduler does not do priority donation. Thus, we recommend that you have the priority scheduler working, except possibly for priority donation, before you start work on the advanced scheduler.
    - Write a code that allows to choose a scheduling algorithm policy at Pintos startup time. By default, the priority scheduler must be active, but we must be able to choose the BSD scheduler with the -mlfqs kernel option. Passing this option sets thread_mlfqs, declared in *threads/thread.h*, to true when the options are parsed by *parse_options()*, which happens early in main().
    - When the BSD scheduler is enabled, threads no longer directly control their own priorities. The priority argument to *thread_create()* should be ignored, as well as any calls to *thread_set_priority()*, and *thread_get_priority()* should return the thread's current priority as set by the scheduler.

4. What to submit?

- Submit a report (5 pages limit) to TA via email (minwook-lee@gm.gist.ac.kr). And commit your source code in github repository.
- The format of report does not matter if it clearly shows how you implemented the project. (please clarify which file of your code has been changed)

- Reference:
  1. https://web.stanford.edu/class/cs140/projects/pintos/pintos.html

5. Appendix

- Current version of pintos system will probably not automatically shutdown qemu emulator
- Add *outw(0x8004, 0x2000);* (or *outw(0x604, 0x0 | 0x2000);*) at pintos/src/devices/shutdown.c
- Do *make clean* and *make* in threads folder then run pintos test again

*shutdown.c

```
  const char *p;

#ifdef FILESYS
  filesys_done ();
#endif

  print_stats ();

  printf ("Powering off...\n");
//  outw (0x604, 0x0 | 0x2000);
  serial_flush ();

  /* This is a special power-off sequence supported by Bochs and
     QEMU, but not by physical hardware. */
  for (p = s; *p != '\0'; p++)
    outw (0xB004, 0x2000);
    outb (0x8900, *p);

  /* This will power off a VMware VM if "gui.exitOnCLIHLT = TRUE"
     is set in its configuration file.  (The "pintos" script does
     that automatically.)  */
  asm volatile ("cli; hlt" : : : "memory");
```

C ▾ Tab Width: 8 ▾ Ln 100, Col 32 INS