Project Threads Report

Group 24

Name	Autograder Login	Email
Sydney Tsai	student134	sydneytsai@gmail.com
Anusha Muley	student143	anusha.muley@berkeley.edu
Cindy Yang	student258	cindycy@berkeley.edu
Qi Li Yang	student85	qiyang2@berkeley.edu

Changes

Efficient Alarm Clock

After our discussion, we took out the locks around the threads list because interrupts are disabled already.

Strict Priority Scheduler

For strict priority scheduler, we did not include a list of all threads per process because we utilize a prio_ready_list within thread.c instead to determine the next highest priority thread.

To determine the next highest priority thread to run, we use the thread_schedule_prio() function to traverse our prio_ready_list and return the thread with the highest effective priority.

Our thread struct now utilizes lock elements instead of thread elements to keep track of the thread that it has donated priority to, as well as threads that have donated priority to it. We traverse these locks and get their holder to determine which thread holds which lock.

For the thread structs, we are also keeping track of more variables. For example, the a list_elem so that we can directly access the list element, the lock the thread needs,

a list of its current locks, and its join status. This helps us better keep track of which threads have been donated to it,

In addition, for strict priority scheduler, we decided not to keep a list of waiting_threads under each lock structure. This is because we utilised the sema→waiters list already defined under the semaphore struct.

Within cond_signal, we don't just iterate through the cond→waiters and get the highest priority thread directly. Instead, we have to access the semaphore within each cond→waiters and find the highest priority within the semaphore→waiters list. We then return the highest priority thread from the semaphore waiters list.

We disable interrupts when we acquire and release lock elements to update our data structures. We also disable it when recomputing the effective priority of our threads.

When donating priority or setting the effective priority, we no longer explicitly join on the receiver_thread. Instead, we just push the lock elements back to the list of current locks under each thread. Similarly, we also no longer use the get_lowest_thread function to loop through and donate priorities. Instead, we just recursively donate the priorities to each other.

Within lock_acquire, we also don't call thread_block() anymore. Instead, after we add the lock element onto the lock holder's current list of locks (to keep track of who priority was donated from), we call sema_down instead. We also update the lock's holder to be the current thread.

User Threads

On our design doc we were unclear about how we were going to keep track of user locks and semas. We ended up making a dictionary with a list of the structs below to keep track of them.

```
  struct list user_lock_dict;     /* list of user level locks
*/
  struct list user_sema_dict;     /* list of user level semaph
ores */
```

```
  struct lock_dict_elem {
    struct lock* lock;
    struct lock_t* user_lock;
    struct list_elem elem;
  };
  struct sema_dict_elem {
    struct semaphore* sema;
    struct sema_t* user_sema;
    struct list_elem elem;
  };
```

We didnt have to redo the thread_create and I think we thought we would have to rewrite that in pthread_create but we were able to just use the existing function to create a new tid and do a lot of the initialization parts.

Added mapping for the page after calling thread using the kpage/upage fields in the thread struct:

```
 uint8_t* upage;             /* pointer to user page */
 uint8_t* kpage;             /* pointer to actual page alloced
for this thread struct*/
```

Also we initially had a lower bound variable in the pcb to keep track of the stack for our install page algorithm but we ended up just looping through pages until finding a page that was available,

We changed the logic for handling threads whose main thread/process has already exited. When a thread calls process_exit we set a bool need_to_kill for each of the threads in the pcb. Then we continue freeing what we can for the process. Once the thread wakes up we trap it in the interrupt handler with the is_trap_from_userspace and call pthread_exit. I think we did not initially have that boolean as part of our design for thread.h

We also added condition variable design in order for the thread calling process exit to wait for all other threads to exit before exiting.

Reflection

Sydney: I worked on priority scheduler, the report doc, and tried to help debug other portions of the project. The priority scheduler was not too bad to understand conceptually, but we were having problems implementing the concepts into the code. It was especially hard to implement the lock priority donation because of the nested donations and computation of effective priority afterwards. Overall, this project was not as bad as the last one because we had learned from the last project how to work more effectively as a team. Although there is still room for improvement, especially with dividing tasks so that we all have enough knowledge to help each other out, we did improve significantly in working as a team during this project.

Cindy: I mainly worked on the priority scheduler, briefly helped set up the syscall table, and the report document. I think the priority scheduler was more straightforward to understand and flowed quite well from the lecture material. This allowed us to have a better time designing the design doc and flushing out our ideas. It was difficult to implement a lot of the nested donations of priorities and recognising when to raise or use condition variables. We also had to create a lot of helper functions in order to make our process smoother. This project was less stressful than the first project because we learned to begin it earlier and make a more thorough design doc. Dividing up the tasks based on what we were comfortable with helped us run through the process a lot easier and improve on working together.

Anusha: I mostly worked on the efficient alarm section and the user threads synchronization methods for locks and semaphores. I think the user threads was very dependent on the low level locking defined through the priority scheduler part and a lot of the threads. I also did some of the syscall table debugging and merging all the different branches we worked on together. I think we were able to understand more of what the other methods everyone was working on this time and share ideas to evolve our design more easily and because we had clearer steps towards passing the tests, building up the codebase and making modifications was a lot easier than trying to fix everything all at once. I think we spent more time on the design and getting conceptual help at office hours so we didnt waste time on poor approaches.

Qi: I worked on all the new pthread syscalls, pthread create, execute, start pthread, setup thread, pthread join, pthread exit, pthread exit main and modifying the process syscalls. I was having trouble with the code flow of pthread create initially because our initial design was inaccurate from what we should have been doing. However, we started the project earlier so I had time to go to office hours and get help. We also made sure we knew how to test incrementally so that we could ensure code was working before building off of it. I think we could've split different parts of the project more evenly so that we have more knowledge of each portion. Overall this project went a lot better than the first one.

Testing

Our test case tests our priority scheduler to see whether priorities are handled correctly when locks, semaphores, and condition variables are all applied to the threads.

```c
void test_priority_donate_sema(void) {
  struct all_synch all;

  /* This test does not work with the MLFQS. */
  ASSERT(active_sched_policy == SCHED_PRIO);

  /* Make sure our priority is the default. */
  ASSERT(thread_get_priority() == PRI_DEFAULT);

  lock_init(&all.lock);
  sema_init(&all.sema, 0);
  cond_init(&all.cond);
  thread_create("low", PRI_DEFAULT + 1, l_thread_func, &all);
  thread_create("med1", PRI_DEFAULT + 3, m_thread_func, &all);
  thread_create("med2", PRI_DEFAULT + 3, m_thread_func, &all);
  thread_create("high", PRI_DEFAULT + 5, h_thread_func, &all);
  sema_up(&all.sema);
  msg("Main thread finished.");
}
```

In our test case, we initialise locks, semaphores, and condition variables. We then create 4 different threads with low, medium, and high priorities. We create two threads with the same priorities to ensure we handle fifo correctly when threads have the exact priority as each other. The expected output of our test is to have the high priority thread run first, then med1, med2, and then the low priority thread. Despite all of the synchronisation variables, it should still run the threads based on its priority and use round robin when they have the priority.

Two potential non-trivial kernel bugs could be race conditions and deadlocks. If there were kernel bugs that incorrectly handled race conditions, then two threads with the same priority could be racing against each other to access the shared resources and modify it. This would affect the output of the test case and cause it to be non-deterministic because there is no way to predict the result of running the program.

In the instance of deadlocks, if the kernel bugs out and does not handle deadlocks correctly, then it could cause multiple threads to wait for the same resource to be released. This could alter the output of the test case because processes could potentially get stuck. Since deadlocks are also non-deterministic, it would also cause the output to be unpredictable.

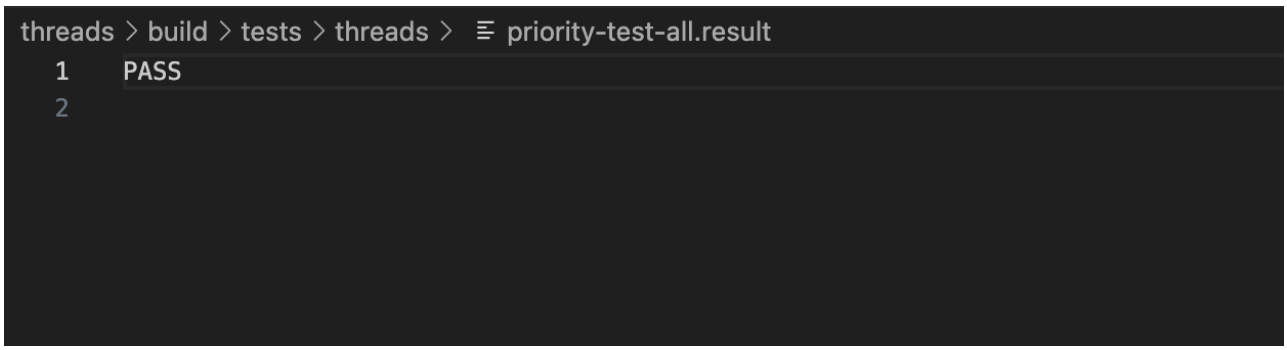
Writing Pintos tests were a little bit unintuitive in the beginning, but we referenced similar priority tests to recognize the format that we needed to follow. From there, we adjusted the test to fit what we were looking for, such as adding in the condition variables and semaphores that we wanted to add. It was also fun thinking about what the test output should be and making sure that our code also accounted for this test. To improve Pintos tests in the future, it would be great to have a master guide to add tests, since it took a while to get the hang of. Through writing test cases, we learned to double check all edge cases of our code and ensure that our schedulers should run properly under any conditions.

threads > build > tests > threads > priority-test-all.output

```
qemu-system-i386 -device isa-debug-exit -hda /tmp/zPVk_T8Ucr.dsk -m 4 -net none -nographic -monitor null
[?7l[2J[0mSeaBIOS (version 1.15.0-1)
Booting from Hard Disk...
PPiiLLoo  hhddaa1
1
LLooaaddiinngg.....................................
Kernel command line: -q -sched=prio -f rtkt priority-test-all
Pintos booting with 3,968 kB RAM...
367 pages available in kernel pool.
367 pages available in user pool.
Calibrating timer...  216,268,800 loops/s.
ide0: unexpected interrupt
hda: 5,040 sectors (2 MB), model "QM00001", serial "QEMU HARDDISK"
hda1: 311 sectors (155 kB), Pintos OS kernel (20)
hda2: 4,096 sectors (2 MB), Pintos file system (21)
ide1: unexpected interrupt
filesys: using hda2
Formatting file system...done.
Boot complete.
Executing 'priority-test-all':
(priority-test-all) begin
(priority-test-all) Thread L acquired lock.
(priority-test-all) Thread L downed semaphore.
(priority-test-all) Signaling...
(priority-test-all) Thread H acquired lock.
(priority-test-all) Thread H upped semaphore.
(priority-test-all) Thread H released lock.
(priority-test-all) Thread H finished.
(priority-test-all) Thread M acquired lock.
(priority-test-all) Thread M upped semaphore.
(priority-test-all) Thread M acquired lock.
(priority-test-all) Thread M upped semaphore.
(priority-test-all) Thread L finished.
(priority-test-all) Main thread finished.
(priority-test-all) end
Execution of 'priority-test-all' complete.
Timer: 62 ticks
Thread: 30 idle ticks, 32 kernel ticks, 0 user ticks
hda2 (filesys): 3 reads, 6 writes
Console: 1285 characters output
Keyboard: 0 keys pressed
Exception: 0 page faults
Powering off...

```

.output file of our custom test



.result file of our custom test