Project Threads Report

Group 79

Name	Autograder Login	Email
Jacob Wu	student303	jacobwu@berkeley.edu
Max Ye	student95	maxye@berkeley.edu
Ryan Nader	student277	berkeleybear22ryan@berkeley.edu
Vedansh Malhotra	student39	vedansh@berkeley.edu

Changes

Efficient Alarm Clock

One minor change was adding `struct list_elem timer_elem` to `struct thread`. We used `timer_elem` to keep track of sleeping threads and remove threads from the `sleeping` list. This was done so as to not interfere with the `struct list_elem elem` in `struct thread` that was used by synch.c (ex. for the `waiters` in `struct semaphore`).

Outside of that, there weren't other changes to our design, since our design document was written after a fully implemented and working solution of Efficient Alarm Clock.

Strict Priority Scheduler

We added a lock priority comparator to compare which lock had a higher priority based on the threads waiting on the lock: `static bool lock_priority_comparator(const struct list_elem* t1, const struct list_elem* t2, void* aux UNUSED)`.

In addition, we added `struct list_elem sched_elem` to `struct thread`. Similar to Efficient Alarm Clock, we did this to not interfere with the `struct list_elem elem` in `struct thread` that was used by synch.c (ex. for the `waiters` in `struct semaphore`). `sched_elem` was used for the list of ready threads.

For the most part, our design stayed the same as outlined in our original design document.

User Threads

For process syscalls, our algorithms did not change much compared to the design doc's specifications (besides adding some sanity checks / asserts, and noticing that the sanitization of arguments was *already* handled in the staff solution.) We added some fields to the metadata of the struct thread and struct process for bookkeeping and to help with debugging (for instance, we made sure that the process kept a list of the join datas of all of its threads, rather than a list of TCBs, to be more memory efficient, and to make it easier to directly access the join data. Hence, the join data struct was also updated to have a ptr to the PCB. Further, the join data now holds the boolean var that tells a thread if it should gracefully exit in a safe space, since the PCB iterates over join datas.) We also decided to keep track of a thread's ESP and EBP, but realized we didn't need to explicitly hold the stack pointer. Per Wilson's suggestions in the review meeting, we also kept track of the palloc'd page that was supposed to be freed in the struct thread. Finally, we also ensured that if the main thread calls `pthread_exit`, then the syscall re-routes it to the `pthread_exit_main` method.

In our design, we noted that the init_user_thread was just attempting to setup a PCB, which was utter tosh, so we removed it. Another big optimization was that we didn't just busy wait (and keep yielding) in process_exit to wait for other threads to exit, instead, we just called join on all of them in our loop! So this was nonsense:

```
while (!list_empty(&pcb->my_threads)) {
  thread_yield(); // Let the others cook!
}
```

Instead: `pthread_join(jd->tid);`

A problem we found was that we could not remove a thread's join data from the PCB list in pthread exit, because it's valid to join on a thread that's already terminated. So, we instead removed it in the join method, once we knew we can't join on the thread legally again. However, this caused some test cases to fail, so we commented it out to maximize points (as of the time of writing this…) A point of confusion we had per the GS rubric is that we thought a process must free all of its *user* locks too, before waiting / thread joining. This was cleared up on an Edstem comment, so we removed this (no need for a *faulty* deadlock prevention algo — be an ostrich.)

Another big concern (thanks sm Wilson for noticing) is that in our design doc, we thought palloc_get_page returns a user (and not a kernel) page! This was wrong, and we had to change the *entire* design for pushing to a stack's memory accordingly. Further, we kept looking for free memory using a while loop instead of a single check, just in case there was some fragmentation in the process' stack (Wilson asked us to do this too.) Further, we did *not* directly dereference the ESP and push arguments to the user page, instead we used the staff solution's push method with the actual kernel page! PS, we also realized we don't need to mark a thread's status as dying in the pthread_exit method since thread_exit handles that. In pthread_exit_main, we also made sure to set the process' exit code accordingly. Another update was that we had to make sure to use different `list_elem`s for each list that a struct's member could be a part of!

A final major change was that we realized using or not using a static kernel lock when trapping into syscalls did not cause any changes to our score. In the end, we defensively commented out the lock in our final submission.

For user-level synchronization syscalls, we added null checks and other checks (ex. in `sys_lock_acquire`, we check if the thread is already holding the lock they are trying to acquire) that weren't accounted for in the original design doc. We also added `char* user_id` to `struct semaphore` and `char* user_id` to `struct lock`. `user_id` would be set to the `typedef char lock_t` that was passed in as an argument in `sys_lock_acquire` for example. Then, for instance, in `sys_lock_release` we would compare `user_id` to the argument.

For pthread library, we tried lots of different changes compared to the initial design document. A lot of these changes had to do with the management of kernel versus user pages, as suggested to us in OH. We also added more synchronization to protect accesses to the page-directory of each process. Besides that, our final submission relied primarily on the design specified in the document.

Reflection

Max - added in staff solution from project 1; implemented all of Efficient Alarm Clock, all of Strict Priority Scheduler, and user-level synchronization syscalls; prepared final report and added test case.
Max - User Threads, in particular the Pthread library, took much more effort than expected. We should have started earlier on the Pthread library so we weren't so rushed at the end. However, we did our best with the time we had and put in serious work, which I appreciated. Overall, I think we did a good job.

Vedansh - Wrote the entire design, answered questions re design as / when needed to help others with the implementation, then debugged the pthread library — got create-simple working and then went ham-and-shotgun with Ryan helping to pass most tests. Updated the report meticulously with all the changes we made to the library in the end.
Vedansh - The Pthread library turned out to be extremely difficult to debug. In hindsight, we should have started much earlier — but I also think there was very limited help in OH so making progress was difficult and slow. When we had an issue, we often just had to keep reading and re-reading our code (as suggested in OH) while making no progress. Still, I wish we had started earlier, but am happy that everyone put in lots of effort to try and make progress. Wilson is a Godsend and was always giving us helpful advice.

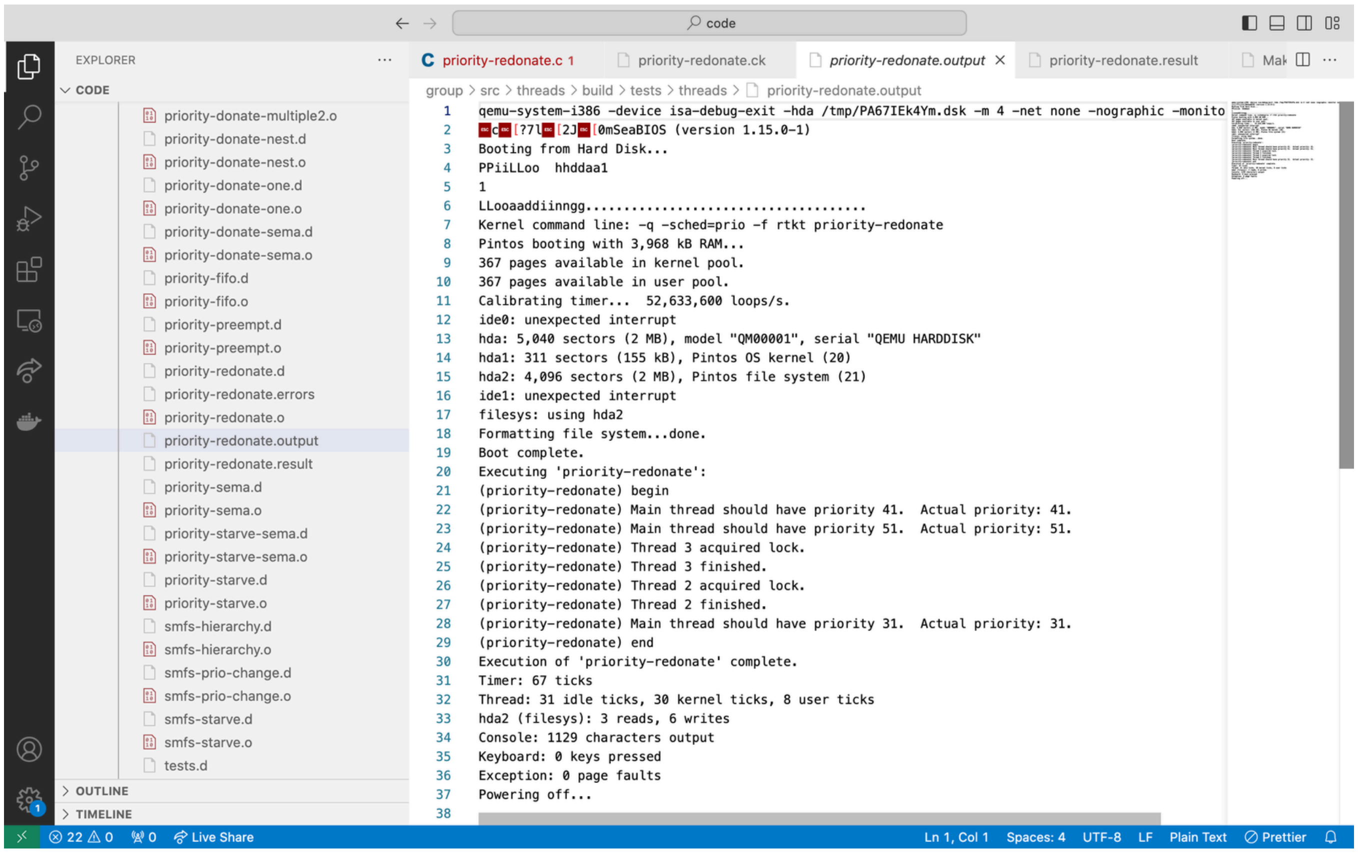
Ryan - Worked on updating the syscalls and pthread library. Working with Vedansh and Jacob and make some changes on Max's user-level synchronization syscalls.
Ryan - Just like everyone has said we ran into issues with `setup_thread` and getting the pthread library working. I think Max did a great job for this project and help minimize the points we lost and I think I could have stopped debugging with gdb and instead just have gone slowly through the instructions again on the stack alignment part as this was the one section I was most unsure about and wasted most time on.

Jacob - Worked on implementation for the pthreads library, specifically the pthreads execute, start pthread, setup stack, pthread join, and other functions in collaboration with teammates, and focused on debugging for the pthreads library, and incrementally made changes as well as tested to ensure correctness and compatibility.
Jacob - We found it extremely challenging to implement the Pthreads library, especially because we do not have specific background in understanding how threads and processes work conceptually, as well as how to align the stack and use the assembly line language to correctly initiate them. As a result, it would have been a lot more helpful if we collaborated on the design document and consider ways in which we can seek alignment between the design decisions and the implementation, making sure whoever implements different components of a project have in-depth knowledge about different design choices, trade-offs, and develop a profound conceptual understanding for each of the components that needs to be implemented.

Testing

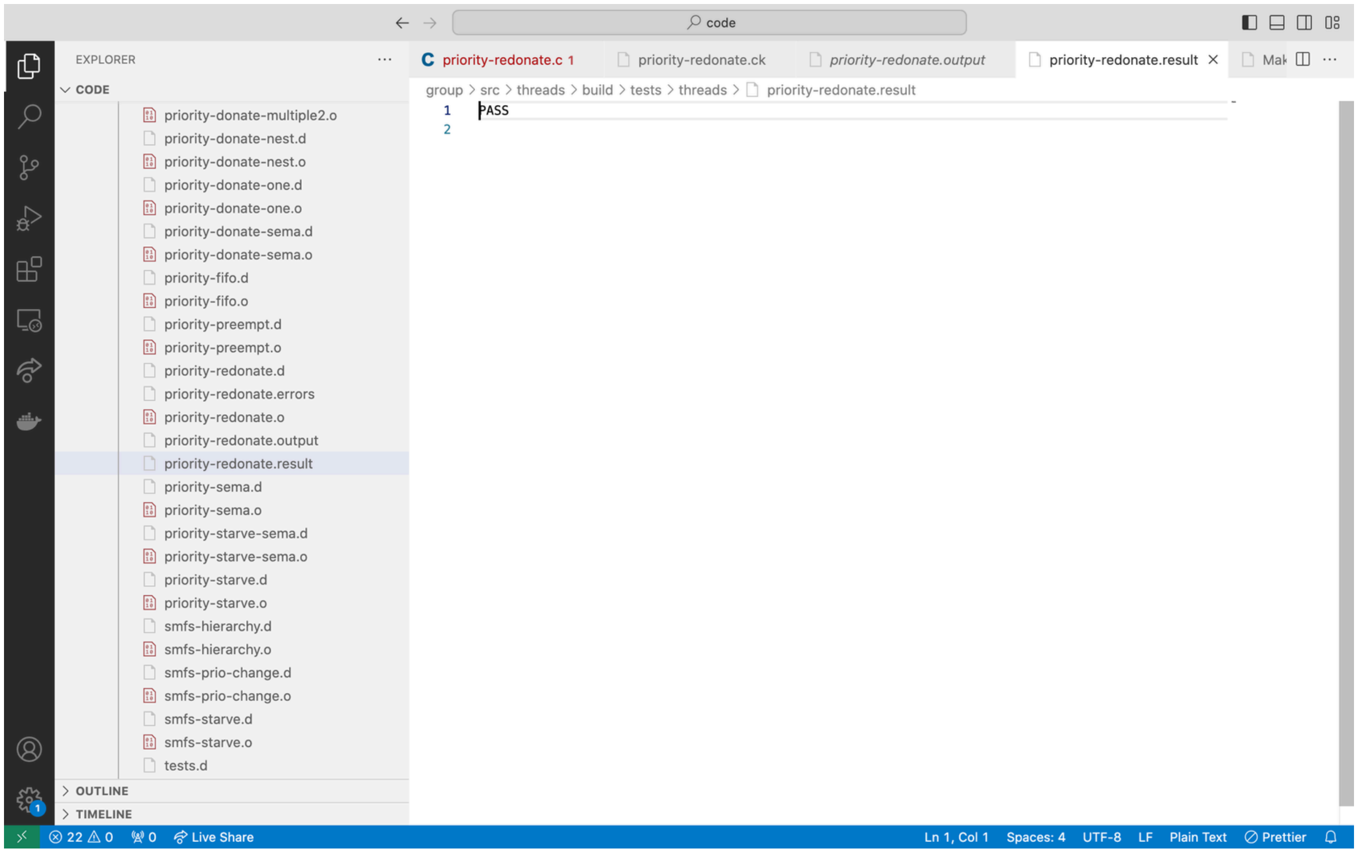
`priority-redonate`

- Description: if a thread has already received a priority donation and then another thread with even higher priority attempts to acquire the same lock, we should test that the system handles re-donation correctly.
- Overview:
  a. The main thread acquires a lock and holds it.
  b. A secondary thread (Thread 2) attempts to acquire the same lock that the main thread holds. Since the lock is occupied, Thread 2 should donate its priority to the main thread.
  c. A third thread (Thread 3) with a higher priority than Thread 2 also tries to acquire the same lock. This should lead to a "priority re-donation," where the main thread should now operate with this new, higher priority that Thread 3 has.
  d. The main thread finally releases the lock. At this point, the priority of the main thread should be restored to its original value, and Thread 3 and then Thread 2 should acquire the lock in that order due to their higher priorities.
- Output and results of Pintos kernel
  - Output:



```
qemu-system-i386 -device isa-debug-exit -hda /tmp/PA67IEk4Ym.dsk -m 4 -net none -nographic -monito
c[?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-redonate
Pintos booting with 3,968 kB RAM...
367 pages available in kernel pool.
367 pages available in user pool.
Calibrating timer...  52,633,600 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-redonate':
(priority-redonate) begin
(priority-redonate) Main thread should have priority 41.  Actual priority: 41.
(priority-redonate) Main thread should have priority 51.  Actual priority: 51.
(priority-redonate) Thread 3 acquired lock.
(priority-redonate) Thread 3 finished.
(priority-redonate) Thread 2 acquired lock.
(priority-redonate) Thread 2 finished.
(priority-redonate) Main thread should have priority 31.  Actual priority: 31.
(priority-redonate) end
Execution of 'priority-redonate' complete.
Timer: 67 ticks
Thread: 31 idle ticks, 30 kernel ticks, 8 user ticks
hda2 (filesys): 3 reads, 6 writes
Console: 1129 characters output
Keyboard: 0 keys pressed
Exception: 0 page faults
Powering off...
```

- Result:



```
PASS
```

- Two non-trivial potential kernel bugs and how they would have affected the output of this test case

- If the kernel had a faulty SCHED_PRIO scheduling policy (ex. randomly scheduling the next ready thread as opposed to scheduling the thread with the highest priority), this would have affected the test output dramatically. For instance, after the main thread releases the lock, Thread 3 should run and then Thread 2 should run (as shown by priority-redonate.output). However, with a faulty SCHED_PRIO policy, Thread 2 could run before Thread 3, giving us an output of "Thread 2 acquired lock. Thread 2 finished. Thread 3 acquired lock. Thread 3 finished." We would have indeterministic outputs with a faulty SCHED_PRIO scheduling policy.
- In `thread_create` in `thread.c`, we add a `thread_yield()` call at the end to defensively yield the CPU in the case the current thread creates a new thread that has a higher priority. If instead we didn't do this, we would create a bug such that a lower-priority thread may run to completion before a higher-priority thread. In our test case, Thread 2 could be created with a higher priority but since the main thread is already running, it continues to run. Thread 2 is queued and doesn't run (even though it has a high priority than the main thread), giving us "Main thread should have priority 41. Actual priority: 31." This is because Thread 2 doesn't run, doesn't try to acquire the lock, and doesn't donate its priority. This could continue with Thread 3 being created with a higher priority, but since the main thread still does not yield, we get "Main thread should have priority 51. Actual priority: 31."

What can be improved about the Pintos testing system? If there was a way Pintos could automatically identify the new tests added, that would be helpful. This would reduce the need of adding the name of the new test to the `tests/threads_TESTS` variable and adding `tests/threads/priority-redonate.c`. to `tests/threads_SRC`.

What did you learn from writing test cases? We learned how to add test cases to the Pintos testing system and to think of the edges cases that could happen for our functions. Though it's hard to write tests that cover every single edge case, it's definitely useful as it helps reduce bugs and clarifies your thinking. We learned that writing tests helps the coding process and is not simply an add-on.