Project 2: Final Report

Group Members

Xinyu Fu (fuxy@berkeley.edu)
James Tang (jamestang@berkeley.edu)
Lanxiang Hu (hlxde1@berkeley.edu)

Changes Made Since Design Doc

Task1

In our original design, we used a variable to keep track of how many remaining ticks a thread has to sleep for. However, this design requires updating this variable for every sleep thread at every time step, which is very costly. After discussion with our GSI William, we decided to keep track of the exact tick at which a thread should be woken up `wakeup_tick`, so that we do not need to update it at every tick.
Another change that we decided to make was that originally we were planning to keep the list of sleeping threads unordered, which means at every tick we have to iterate over all sleeping threads to see if there are threads needed to be woken up (an O(n) operation, n being the number of sleeping threads). This can also be costly once we have a certain number of sleeping threads, since we are performing this check at every tick. In order to minimize work performed in `timer_interrupt`, we decided to maintain an ordered list (by `wakeup_tick`). This way we only need to check the threads in front. This ordering requires some work when putting a thread to sleep, but it is worthwhile to maintain since now we don't have to traverse the entire `sleeping_list` at every single tick.

Task2

Part 1: Priority Scheduling

For this part, we made some extra effort to develop and improve our `thread_yield` scheme in cases when a thread's struct variables are modified, which might therefore lower the current running thread's effective priority. This functionality is integrated into one single function, `check_thread_yield`, which is called in places where immediate yielding could happen due to priority decrementation.
We added functionalities to disable interrupts in multiple places where current thread's struct variables could potentially be modified including `lock_release()`, `thread_set_priority()` and `thread_create()`. Since we would be adding, modifying struct variables from threads or removing threads in the donors' list data structure, disabling interrupts ensures the system won't run into synchronization issues.
We deleted a field which represents a list of waiting threads of a clock in the `lock` struct because we can just use a clock's semaphore to get its waiting threads.

Part 2: Priority Donation

We initialize each thread in `init_thread` with its effective priority and base priority set to the appropriate values.
In our initial design, we kept a list of held locks and a list of locks a thread is waiting for each thread, and then compute the maximum effective priority among the waiters of each thread and take the highest-priority thread among all the held locks as the one to be run next. To make the logic simpler, we simply keep track of a list of donor threads and the donee thread instead.
The donor-donee scheme works as follows:
Everytime a thread calls `lock_acquire()` on a busy lock, priority donation occurs and it gets added to the donors list of the lock holder.
After priority donation, the lock holder becomes the current thread's donee.
To get the maximum effective priority of all threads waiting for the current threads, we just simply iterate through the list of donors to get the highest priority.
Waiters of a lock are removed from the donors list of a lock holder when it calls `lock_release()`.
We looked into `thread_set_priority()` to make sure base priority and effective priority are handled properly in accordance with

the thread's old effective priority so that previous priority donation can stay effective when the new priority doesn't turn out to be as large.
We deleted a field called `highest_p` which is used to keep track of a thread with the highest priority. Then we realized that we can just use a local variable to do that.

Reflection on Group Work

What each of us did:

- Xinyu Fu: implemented parts of task 2.
- James Tang: Task 1 and final debugging.
- Lanxiang Hu: implemented parts of task 2 and debugged thread yield conditions.

Virtual/Asynchronous collaboration strategies:

Since our group members are located in different places, we used texting and zoom as our main ways of communication. We spent as much time as possible together on zoom whenever we were working on the project. In cases when some of us were not available, we tried our best to divide up the work and meet up to put some of our code together.

What went well:

1. Everyone was very involved in the process of completing the project. This time we spent more time together coding, rather than splitting up and code in our own time. Considering that we have a lot less bugs to deal with this time, this way of team work seems to work quite well.

2. All of us contributed a lot in terms of solid coding and creative ideas.

3. We wrote easy-to-understand and clean code with comments.

4. Compared to the previous project, we did not spend too much time debugging this time, which means we had a solid understanding of the project and we were more careful when coding it.

What could be improved:

1. We should git pull and get the up-to-date version of code before implementing the project to avoid git conflicts next time.

2. We should push our code to Github frequently so that other people don't have to wait for the latest version of code to debug the project.

3. We might want to start on project 3 earlier next time. We only had one day left to debug project 2, but it's safer to leave more time for the debugging process because we will never know how many bugs would be.