Alexander La Cour asl247
Raghav Kumar rk524

Laboratory Assignment 4: Locks

I. Introduction

The goal of this lab was to help us understand and implement locks that could guard the critical sections of the concurrently running programs that we had implemented in Lab 3. This lab built on lab 3 and took us a step closer to how concurrency is actually used in modern computers. It required us to leverage our understanding of blocked locks, interrupts, blocking queues and locking mechanisms from class to implement three functions, *lock_initialize, lock* and *unlock*.

From a high level perspective, our design involved a blocked queue for each lock. This queue existed separately from the ready queue. Whenever a process tried acquiring a lock, the *lock* function would check whether the lock was already acquired by some other process. If yes, then the process trying to acquire the lock was put on the blocked queue for that respective lock instead of being put on the ready queue unlike what we were doing in lab 3. In the *unlock* function, we checked whether there were any process that were waiting on that lock, i.e. the lock had processes in its blocked queue. If yes, then we took the head of the blocked queue and put it on the tail of the ready queue. If no, then we just let the lock be up for grabs by setting it's 'acquired' field as 0. We also made helpers like *push_tail_process* and *push_blocked_tail* that helped us push processes at the end of the ready and blocked queue respectively.

II. Design

As mentioned above, our design involved maintaining blocked queues for each lock in addition to the ready queue from lab 3. The blocked queues essentially maintained a record of all the processes waiting on a particular lock in the order that they tried acquiring the lock. All the processes in the ready queue on the other hand were processes that had either acquired the lock or were in their non critical section. As a process exited its critical section and 'unlocked', the process at the head of its blocked queue would acquire the lock and get put on the tail of the ready queue. It's important to note that once a process released its lock, if it tried acquiring it again, it would get put on the tail of the blocked queue. This way we ensured fairness between different processes waiting on the same lock. The concurrency implemented in lab 3 ensured progress because even if a process crashed in its NCS, a context switch would happen eventually, allowing other processes to acquire the lock. The blocking queues also ensured safety because if a process acquired the lock and entered its critical section, all other processes (trying to acquire the same lock) would be put on the blocked queue and would only be allowed to run when the first process released its lock. Below is a schematic of the high level design --

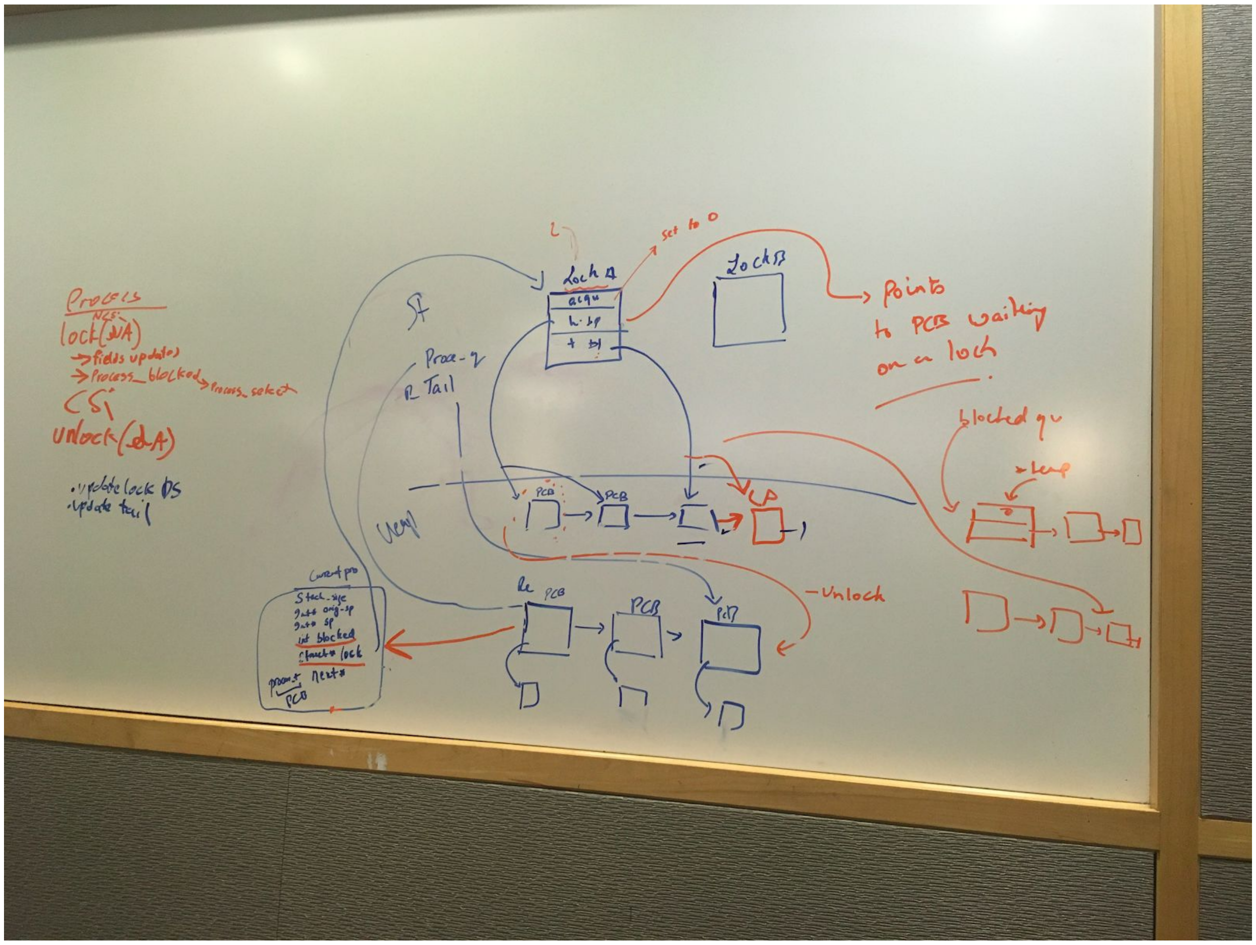

Figure 1 → Although the schematic is not very lucid and clear, one can see how a lock maintained its blocked queue using the *blocked_tail* and *blocked_head*. The PCBs in the lower row represent the ready queue and the orange arrow (from the upper row to the lower row of PCBs) represent what happens when a lock is unlocked and the process is put on the ready queue from the blocked queue.

III. Coding

We first made some changes to the structs that we were provided with. We added fields like *'acquired'* (binary that kept track of whether the lock was acquired or not), *'blocked_queue'* and *'tail_queue'* (pointers to the head and tail of the blocked queue for that lock). Now that we had the lock set up, we implemented the functions as required by the assignment.

A. Lock_init

This was just a simple function that initialized a particular lock. Since a lock could not be acquired on creation, we set the acquired field to 0 and the blocked queue pointers to NULL.

B. l_lock

Since we wanted the process of acquiring a lock to be atomic, we turned off interrupts while in the function. As described in the design, this function would check if the lock that the process is trying to acquire is already acquired. If yes, then the *'blocked'* field of the PCB for that process would be changed to 1 and the process would be added to the block queue for that lock. For this the lock function called would call on the helper '*push_blocked_tail'.* Depending on whether the blocked queue is NULL or not, the helper would add the recently blocked process on the blocked queue. As suggested in the lab handout, we also call the function *process_blocked.* Finally, if there is no process that has acquired the lock, we let the process acquire the lock by putting it on the ready queue and turning the acquired field to 1 for that lock.

C. l_unlock

Similar to the lock function, we disabled interrupts for the unlock function. As per our design, since only one process can acquire the lock at a time, once the *unlock* function is called by a process and the lock is released, the first process in the blocked queue for the lock gets to acquire the lock. If the blocked queue is not empty, we take the head process from the blocked queue and put it on the tail of the ready queue. We do this using the helper function *push_tail_process* that uses the global *process_queue* pointer.

If there are no blocked process in the blocked queue for that lock, we simply make the acquired field for that lock a 0 which would allow any other process to acquire it if need be.

IV. Code Review

All members of the group made sure to spend some time reviewing the code for the lock data structure, the init, lock, and unlock functions as well as changes to our previous functions and data structures to ensure that we all understood what each line of code does. For lock and unlock, we split up the work by first implementing the required C code individually. Once we both had a solution for these functions, we met together to discuss any errors, discrepancies, or misunderstandings. The data structures involved in the blocked queue were more involved, so our original drafts were crafted together by drawing sketches (attached) to ensure we were on the same page. Code review with the addition of peer programming allowed us to get a proper understanding of the C code and the concepts underlying this lab.

V. Testing

When we first finished writing our code for updating process.c and creating lock.c, we used the included test files as a baseline for testing our code and data structures. After this, we began to create many different test cases to test edge cases of our code. These included combinations of two processes that access nested locks, a process with no locks running concurrently with locked processes, and a process that does not unlock at the end of its critical section, effectively deadlocking itself, meaning it could not access its critical section after going into it once. These test cases ensured that we understood exactly how we were manipulating our data structures in our code and that our blocked queue was properly interacting with the current_process and the ready queue so that we could predict the correct output of the LED lights.

In terms of testing our code while we wrote it, we made frequent use of the debugger and stepped through our code line by line in order to draw the data structures as they were initialized to see if the program behavior matched with our predicted behavior of what we drew out.

We were also able to test/verify our program by going to office hours to make sure our code passed lab specifications and proper efficiency with different load values on our interrupt timer.

VI. Documenting

Our main source of documentation for this lab was comments made in the actual code to explain what each line of code was doing. Since this lab involved modifying our code from the previous lab, it was extremely helpful to comment on every change we made as we went along so we could retrace our footsteps. Whenever we came across something that we were unsure about, we would take one approach while documenting the other possible approaches. In addition, because we completed the assignment over the whole week, it was helpful to be able to come back to the code that was written earlier and still understand its functionality with adequate clarity.

VII. Work Distribution

Overall, we met five times to work on this project, with an average meeting length of about 90 minutes. Our first two meetings revolved mostly around reading all the documentation for the lab and reviewing all the files given to us. This allowed us to efficiently setup and design our data structures and related helper functions. Our next two meetings focused on writing the three functions and editing them until we were able to get the included test case to work. The final meeting was where we wrote our own test case files and thought of unique situations, and began writing the report.

Both group members worked on writing code for this assignment, in both the lock functions and our test cases. Whenever we met we used one computer to code and followed pair programming, an agile software development technique in which two programmers work at a single workstation. One of us wrote the code, and the other observed and questioned code to review each line as it was typed in. These roles were frequently alternated between us. Additionally, we verbally explained the purpose of each line of our code to each other to confirm that our logic was consistent. Both partners took the time to explain different concepts to one another, and we often drew sketches on paper of our data structures when we got confused.

We also divided up writing initial sections of the lab report. Raghav worked on the introduction, conclusion, documenting, design. Alex worked on work distribution, coding, testing, and code review.

VIII. Conclusion

Our team is very satisfied with our success in the completion of this laboratory assignment, as both of us were very cooperative, engaging, and attentive during our meetings. Using blocking locks to ensure only one process in the ready queue accessed a critical section at a time was an educational experience as we got to figure out an implementation for the locking mechanisms and concepts we studied in lecture. We also learned a lot about atomic operations, such as when they should be used, and how different locks could be initialized as structs for different critical sections, and using locks in a nested fashion. In addition, we met all the lab requirements and specifications while also effectively testing our design using a diversity of edge test cases.