I started off this lab with tackling each chunk listed in requirements separately. I started with the Basic requirements.

Initial Research: thread.h

After setting up my environment and migrating my lab1 solution (an arduous task), I began reading through *thread.h* for guidance as to where the ready list could be. The first real mention of it was in the large comment block at the top of *thread.h* where is states the purpose of the 'elem' member. I also saw that the thread structure itself contained the 'priority' member and the 'elem' member as promised. You'd think the next step would be to look t the functions listed in *thread.h*, but I got excited and began looking for a run queue in *thread.c*.

Secondary Research: thread.c:

thread.c is a massive file, so searching through it always yielded new surprises. After reading through file/lib inclusions, I saw that there was nothing called run queue but I did see 'ready_list'. I then searched for 'ready_list' in the entirety of *thread.c*. I found in the the following functions, thread_init, thread_tick, thread_unblock, thread_yield, thread_set_priority, and next_thread_to_run. I figured these were all good functions to read through and edit, so I did. After quite a bit of conceptualizing and tracing code, I figured out the magic happening with schedule and next_thread_to_run. I figured that in next_thread_to_run, we'd need to sort our 'ready_list' before we could pop from the front of the 'ready_list'.

Actual Coding Part I-Priority:

At this point, I figured that I'd deal with the priority part of the lab before the preemption part. So I focused explicitly on getting priority right. I made a sort function called, 'sort_ready_list' which just does a bubble sort of our 'ready_list'. This was a bit tedious considering we needed to take the list_elem and pull the thread struct with said list_elem. The swap was also to reason about considering the head and tail inside our list. In hindsight, I likely could have tried using list_max and list_min, but that's for another time. After designing my sort function, I called it inside next_thread_to_run in order to sort our list before we popped from it. Lo and behold, priority done. Now preemption.

Actual Coding Part II-Preemption:

Next was to tackled preempting the Cpu whenever a higher priority level thread became available. A higher priority thread can only become available when a thread is added into the 'ready_list' (whether it be a new thread, or one returning from being blocked), or when the priority of the running thread is lowered by itself. This meant that I was dealing with the functions thread_unblock, thread_yield, thread_create, and thread_set_priority. I dealt with thread_set_priority first. I just pulled from the front of our 'ready_list', compared that elem's thread structure's priority to the priority of the currently running thread. If it was higher the the current thread's priority, I told the current thread to thread_yield(). It turns out that thread_yield did not need to be modified. The test passed. I then went to tackle thread_unblock. I tried a similar solution as with thread_set_priority. It kept timing out on me, so I looked to where thread_unblock was used. Turns out that it was only called in thread_create. So I just implemented my solution after the thread_unblock call there. It worked. I was still doubtful about thread_unblock though. I wondered if it could be called elsewhere. So I looked for a function that interrupted every so often like our timer did to check our sleeping queue. I found that function to be thread_tick. I implemented the same solution there as I did with thread_create and thread_set_priority, except this time, I used intr_yield_on_return since since we're running in an external interrupt context. Thus our preemption was now complete and the tests passed. If you're wondering about the sorting of our 'ready_list', it would never be in an unsorted order when analyzing the front element as anytime we change anything related to the 'ready_list', it's sorted.

Next! I tackled the advanced portion of the requirements. The goal here was to implement priority waiting for locks, semaphores, and condvars. Since locks were covered with semaphores, I didn't touch lock.

Initial Research:
Nothing too noteworthy here, semaphore.c and semaphore.h are small. I saw pretty much immediately that I had to edit semaphore_up somehow. So I did just that.

Actual Coding Part III-semaphore:
This looked fairly similar to next_thread_to_run in thread.c. So I approached the solution the same way. Created bubble sort function called sort_sema_list, this I prototyped in semaphore.h. I then called this function before our list_pop_front function. I then modified the code a little so that there was a needToTest bool that determined if our list was empty or not. If it wasn't empty, it would be set to true. Once intr_set_level was called, I then determined if I needed to test the priority. If so, I analyzed the priority of the thread that was popped from the semaphore waiters list. If it was higher than the current threads priority, I called thread_yield. I had to do this since I couldn't find a way to implement a solution inside thread_unblock. After this, the tests for semaphores passed. Then I began to tackle condvars, which was a bit trickier.

Secondary Research:
Same thing as with semaphore research, the code at first looked simple enough, as if the solution was going to be the same. So I tried that.

Actual Coding Part III- condvar:
I tried implementing the same solution for convar as I did with semaphores. It didn't work. After some digging around, I found out that the waiters list for convar wasn't a list of thread elem's, it was a list of binary semaphores that each held one thread each in their respective waiters lists. Once I figured that out and after I tinkered around with trying to get the priority of the thread struct that contained the elem that was held by the binary semaphore inside an element of our waiters list of convar, the solution was relatively straight forward. Just more sorting. Once I created sort_condvar_list (also prototyped in condvar.h) inside condvar.c, I called it in condvar_signal, right before calling semaphore_up. Since I already tackled semaphores, this worked. All the tests passed.

This was 70% of the grade completed, I would have done more before submission time, but time and all, despite the extension (taking CMPS 142 and CMPS 143 alongside this class). I plan to finish 100% of this though since priority donation is likely good test material.