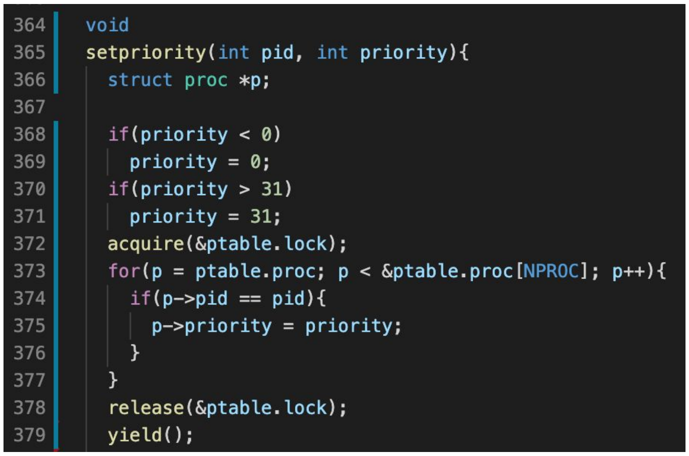
Lab 2 Report

Celyna Su (SID: 862037643) & Wei Xin Koh (SID: 862191103)

For Lab 2, we have implemented the following changes to the code:

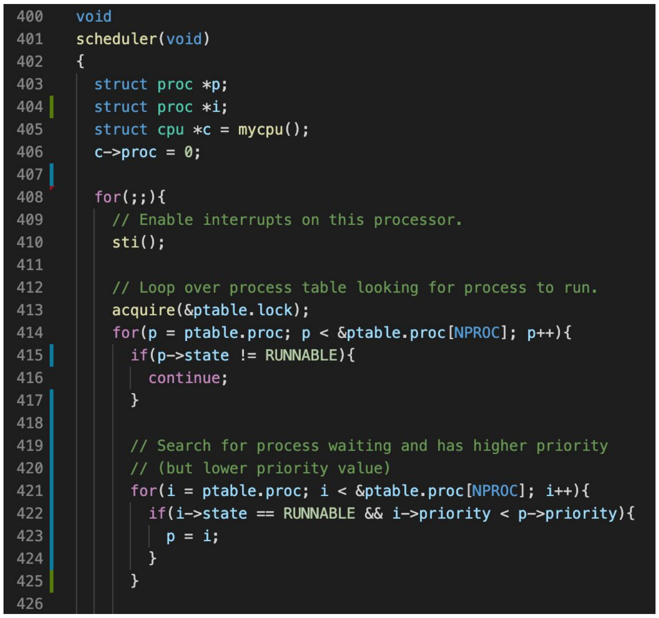
1. Changed the Round Robin scheduler to a priority scheduler. Implemented `void setpriority(int pid, int priority)` function to set the priority value for the process with the pid passed into this function. We also check to ensure that the priority can never fall out of the given [0,31] range. This is done by iterating through the processes in the process table to search for the specific process with the pid passed into the function and assigning the priority set to said process. Most of our changes were in `proc.c`, with some declared in other files such as `syscall.c`, `syscall.h`, etc.



```
void
setpriority(int pid, int priority){
  struct proc *p;

  if(priority < 0)
    priority = 0;
  if(priority > 31)
    priority = 31;
  acquire(&ptable.lock);
  for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
    if(p->pid == pid){
      p->priority = priority;
    }
  }
  release(&ptable.lock);
  yield();
```

2. To implement the aging of the priority, in the `scheduler()` function, within the critical section, we iterate through process table to look for processes to run. We look for a process with the highest priority and is in waiting state and run the process. After it is done running, we look through the process table for any process waiting to run and increase their priority by decrementing their priority values, if it has run, we decrease its priority by incrementing its priority value.



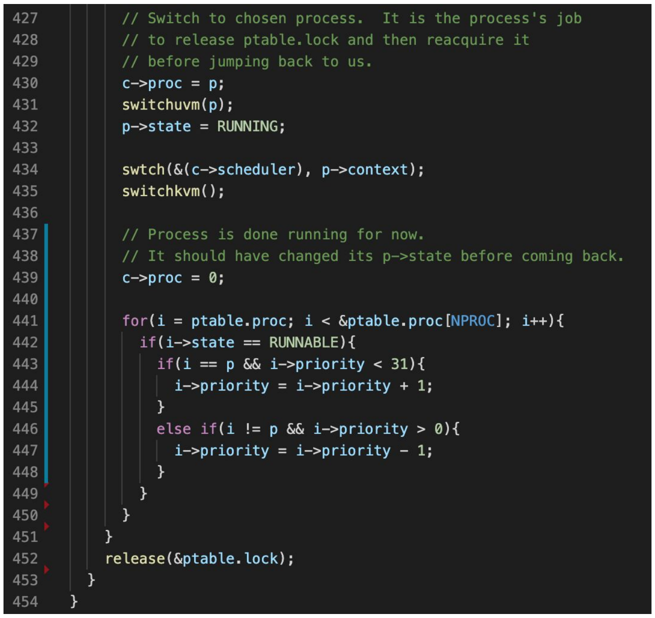
```
void
scheduler(void)
{
  struct proc *p;
  struct proc *i;
  struct cpu *c = mycpu();
  c->proc = 0;

  for(;;){
    // Enable interrupts on this processor.
    sti();

    // Loop over process table looking for process to run.
    acquire(&ptable.lock);
    for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
      if(p->state != RUNNABLE){
        continue;
      }

      // Search for process waiting and has higher priority
      // (but lower priority value)
      for(i = ptable.proc; i < &ptable.proc[NPROC]; i++){
        if(i->state == RUNNABLE && i->priority < p->priority){
          p = i;
        }
      }

```



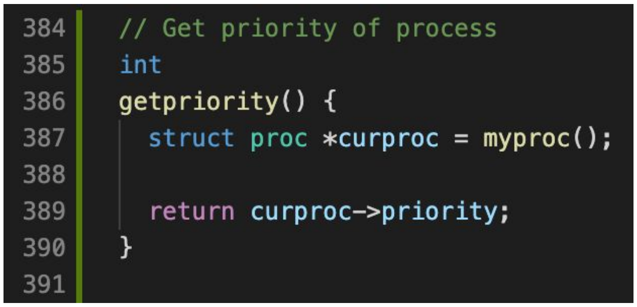
```
      // Switch to chosen process.  It is the process's job
      // to release ptable.lock and then reacquire it
      // before jumping back to us.
      c->proc = p;
      switchuvm(p);
      p->state = RUNNING;

      swtch(&(c->scheduler), p->context);
      switchkvm();

      // Process is done running for now.
      // It should have changed its p->state before coming back.
      c->proc = 0;

      for(i = ptable.proc; i < &ptable.proc[NPROC]; i++){
        if(i->state == RUNNABLE){
          if(i == p && i->priority < 31){
            i->priority = i->priority + 1;
          }
          else if(i != p && i->priority > 0){
            i->priority = i->priority - 1;
          }
        }
      }
    }
    release(&ptable.lock);
  }
}
```

3. Wrote modified `lab2.c` into `lab2_usertest.c` for the user test to trace the priority value of the parent process as it runs and waits for its child process and finally terminates. Implemented `int getpriority()` in `proc.c` to fetch the value of the parent process's priority.



```c
// Get priority of process
int
getpriority() {
  struct proc *curproc = myproc();

  return curproc->priority;
}

```