2022083436 주예지

운영체제 Project 1 Wiki

FCFS and MLFQ Scheduler Implementation

I. Design:

1.) FCFS (First-Come, First-Served) Scheduler

- Selected the process with the earliest PID to simulate creation time
- It ensures non-preemptivity: nonstop behavior until the process yields or terminates.

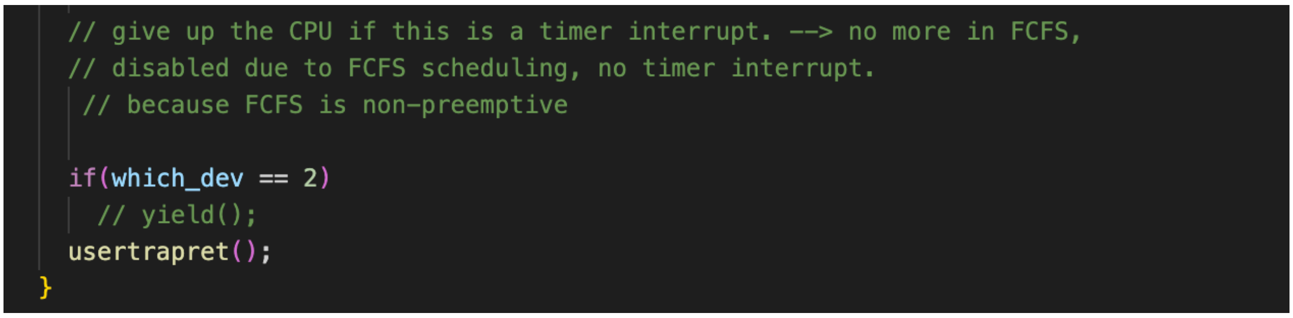
2.) MLFQ (Multi-Level Feedback Queue) Scheduler [the 5 parameters that defines MLFQ]

- Number of queues: 3 (L0, L1 and L2)
- Each queue Li has a time quantum of 2i+1 ticks. (L0: 1tick, L1: 3 ticks, L2: 5 ticks)
- Queue L0 and L1 use Round-Robin scheduling, L2 queue uses priority scheduling.
- Priority boosting is used in every 50 ticks, when L0 uses up its entire time quantum then demotes to L1, and likewise if a process in L1 uses up its entire time quantum then demotes to L2. Within L2 queue, processes with higher priority values are scheduled first. A newly created process has lowest priority 3, and to prevent starvation, priority boosting is implemented so that if the global tick reaches up until 50 then all processes are moved back to L0 queue.
- FCFS and MLFQ mode switching is used by system calls, and the global tick is reset to 0.

II. Implementation

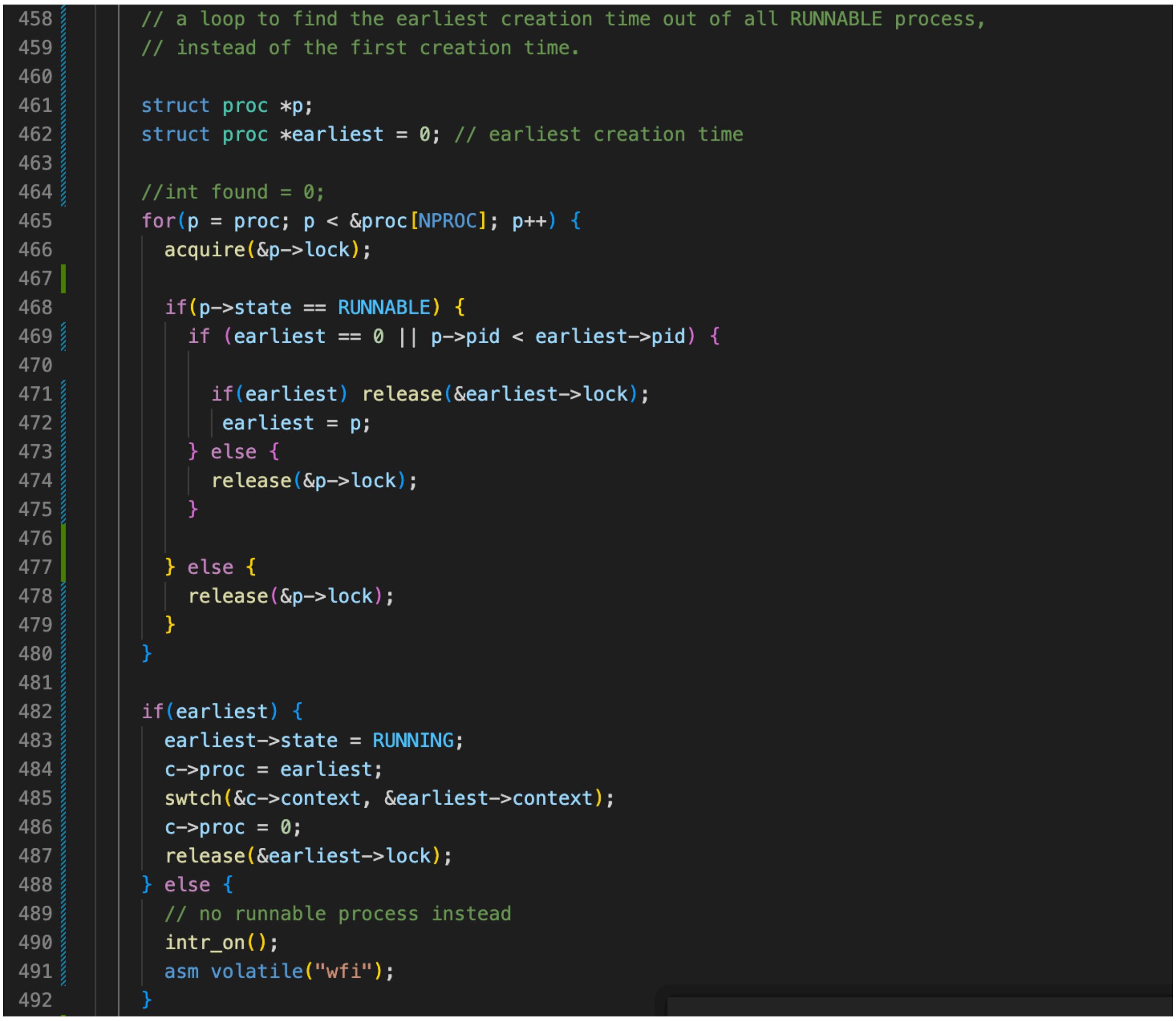
1.) FCFS Implementation

FCFS scheduling is non-preemptive; so inside **int usertrap(void)** of **trap.c**, we turned off the yield() function so that there's no timer interrupt.



```
    // give up the CPU if this is a timer interrupt. --> no more in FCFS,
    // disabled due to FCFS scheduling, no timer interrupt.
     // because FCFS is non-preemptive

    if(which_dev == 2)
      // yield();
    usertrapret();
  }
```

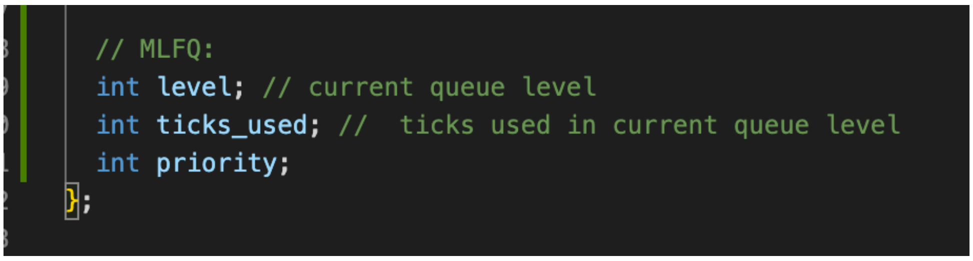
Inside proc.c, I modified the for loop so that I can find the process that spends smallest creation time instead of just the first one.



```
    // a loop to find the earliest creation time out of all RUNNABLE process,
    // instead of the first creation time.

    struct proc *p;
    struct proc *earliest = 0; // earliest creation time

    //int found = 0;
    for(p = proc; p < &proc[NPROC]; p++) {
      acquire(&p->lock);

      if(p->state == RUNNABLE) {
        if (earliest == 0 || p->pid < earliest->pid) {

          if(earliest) release(&earliest->lock);
          earliest = p;
        } else {
          release(&p->lock);
        }

      } else {
        release(&p->lock);
      }
    }

    if(earliest) {
      earliest->state = RUNNING;
      c->proc = earliest;
      swtch(&c->context, &earliest->context);
      c->proc = 0;
      release(&earliest->lock);
    } else {
      // no runnable process instead
      intr_on();
      asm volatile("wfi");
    }
```

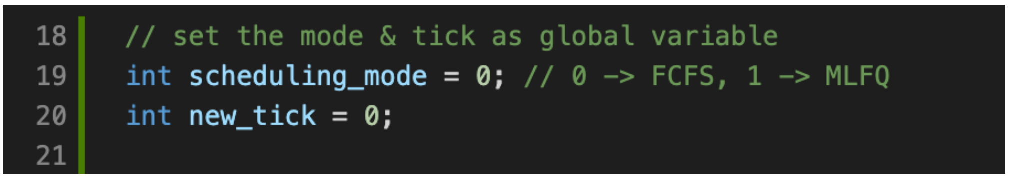
2.) MLFQ Implementation

For MLFQ Scheduling, I defined the prority level, number of ticks used in the current queue level, and the process priority (defined as integers) inside **proc.h**.

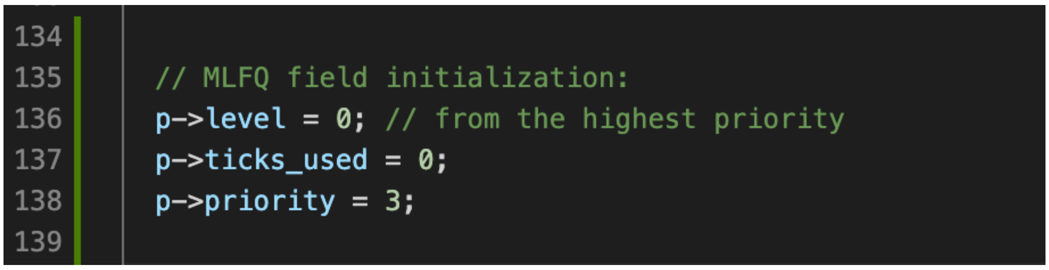


```
  // MLFQ:
  int level; // current queue level
  int ticks_used; //  ticks used in current queue level
  int priority;
};
```

Then I set the global variables for mode switching. For **allocproc**() in **proc.c**, I've initialized the 3 MLFQ fields as such.

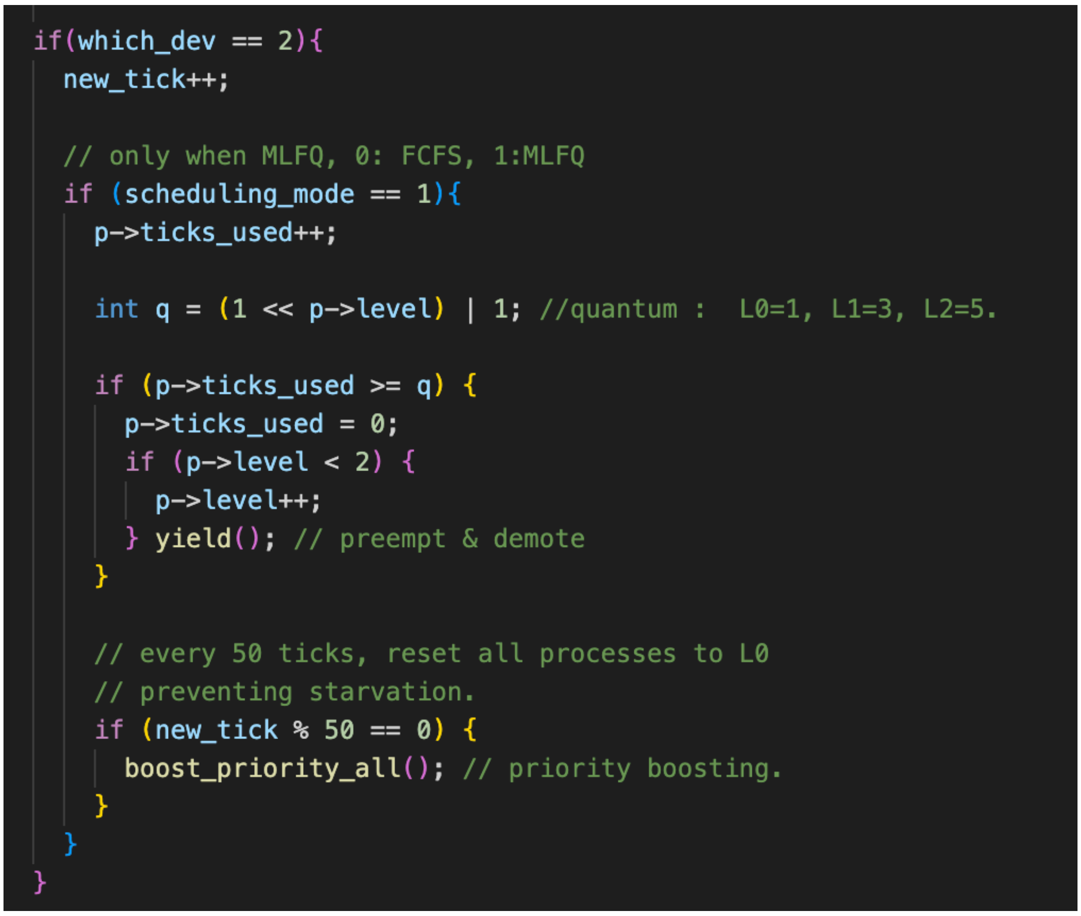


```c
// set the mode & tick as global variable
int scheduling_mode = 0; // 0 -> FCFS, 1 -> MLFQ
int new_tick = 0;

```



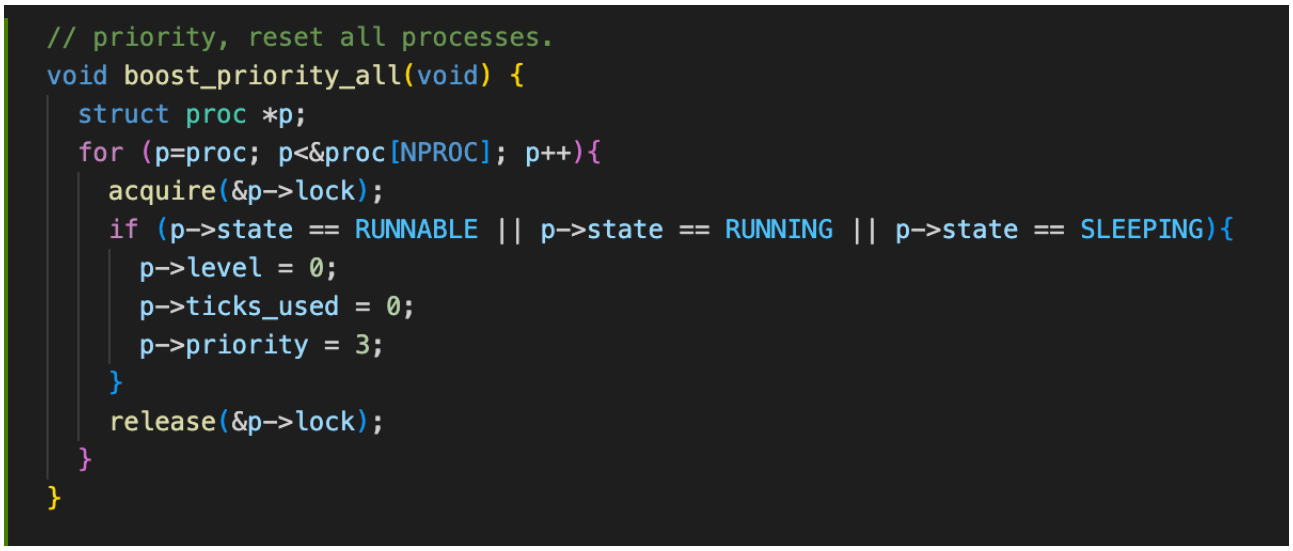
```c

    // MLFQ field initialization:
    p->level = 0; // from the highest priority
    p->ticks_used = 0;
    p->priority = 3;

```

Modification of **usertrap**() in trap.c is as following. This is the timer interrupt section. I've increamented ticks_used and checked for preeption only when the scheduling mode is MLFQ. I've reset and demote the process based on the time quantum. Also I've reset everything in every 50 ticks using "%" function.



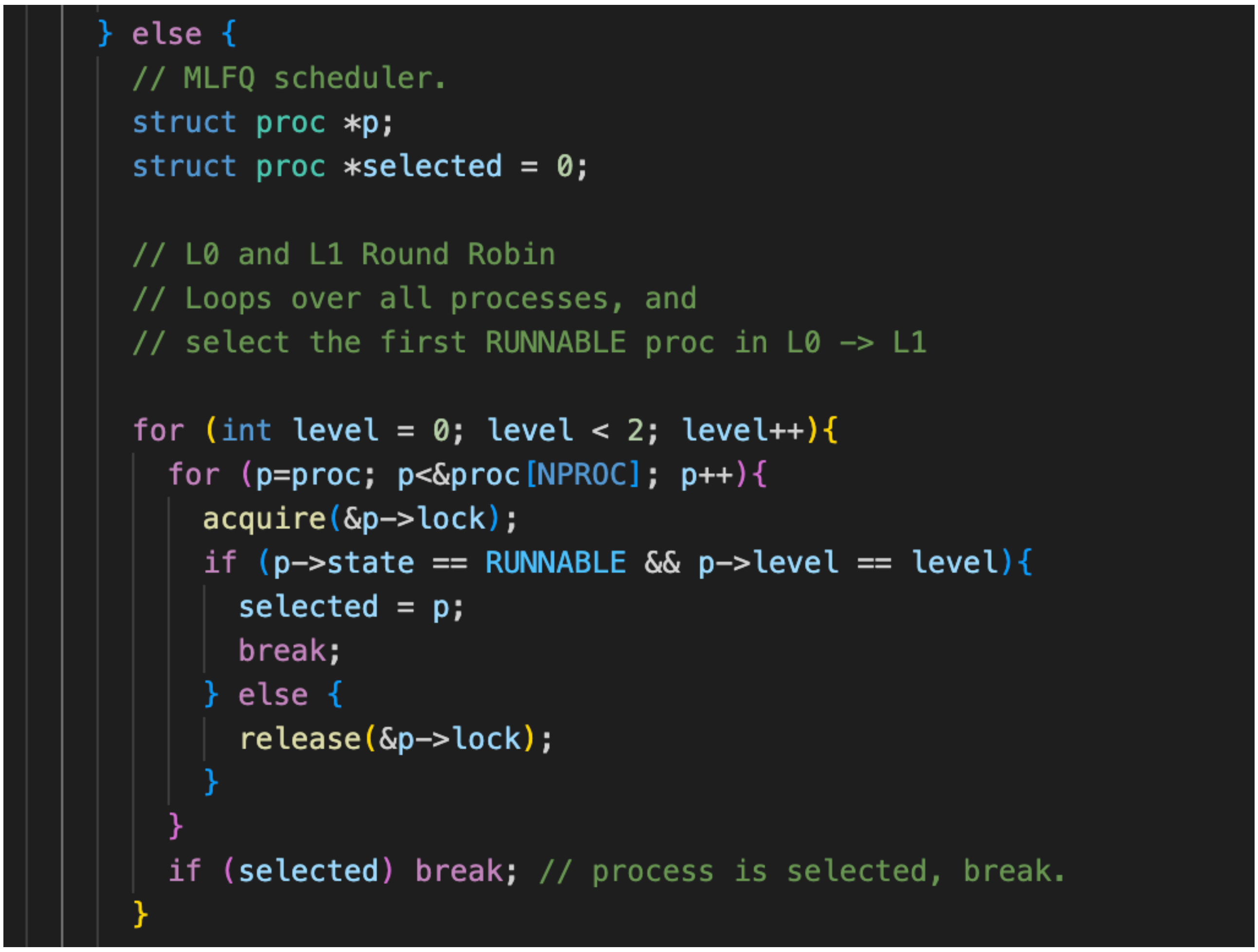
```c
if(which_dev == 2){
  new_tick++;

  // only when MLFQ, 0: FCFS, 1:MLFQ
  if (scheduling_mode == 1){
    p->ticks_used++;

    int q = (1 << p->level) | 1; //quantum :  L0=1, L1=3, L2=5.

    if (p->ticks_used >= q) {
      p->ticks_used = 0;
      if (p->level < 2) {
        p->level++;
      } yield(); // preempt & demote
    }

    // every 50 ticks, reset all processes to L0
    // preventing starvation.
    if (new_tick % 50 == 0) {
      boost_priority_all(); // priority boosting.
    }
  }
}
```

Inside the boost_priority_all() function, I reset all of the processes. I made the queue level into 0, number of ticks to 0 and the priority became 3 by default. Whenever the global tick counts up to 50, this boost_priority_all(void) function is called.



```
// priority, reset all processes.
void boost_priority_all(void) {
  struct proc *p;
  for (p=proc; p<&proc[NPROC]; p++){
    acquire(&p->lock);
    if (p->state == RUNNABLE || p->state == RUNNING || p->state == SLEEPING){
      p->level = 0;
      p->ticks_used = 0;
      p->priority = 3;
    }
    release(&p->lock);
  }
}
```

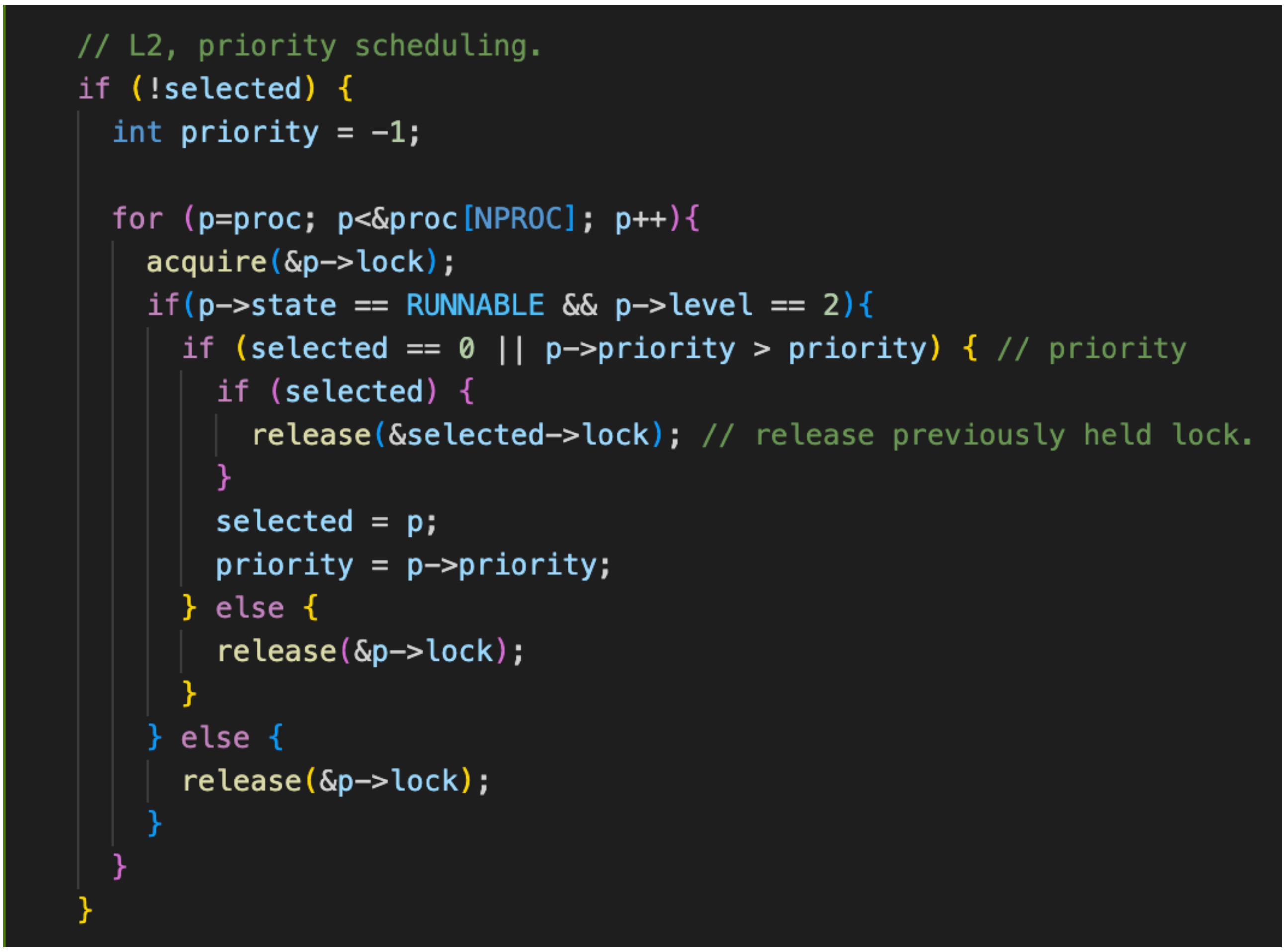
This is the scheduler() in proc.c. The three queues are implemented as follows:



```
      } else {
        // MLFQ scheduler.
        struct proc *p;
        struct proc *selected = 0;

        // L0 and L1 Round Robin
        // Loops over all processes, and
        // select the first RUNNABLE proc in L0 -> L1

        for (int level = 0; level < 2; level++){
          for (p=proc; p<&proc[NPROC]; p++){
            acquire(&p->lock);
            if (p->state == RUNNABLE && p->level == level){
              selected = p;
              break;
            } else {
              release(&p->lock);
            }
          }
          if (selected) break; // process is selected, break.
        }
```

L0 and L1 queues are implemented as Round-Robin (RR) scheduling.

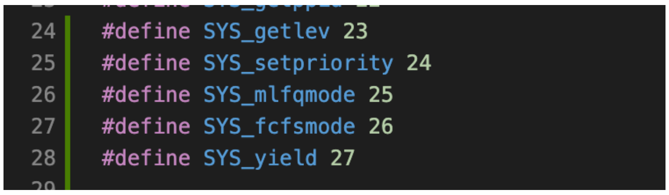
And then L2 queue is based on priority scheduling.



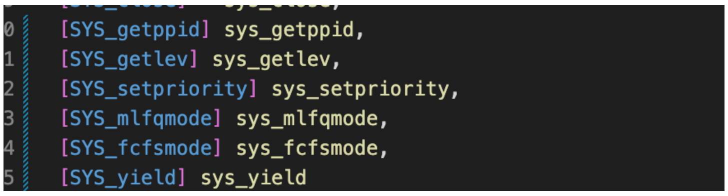
```
// L2, priority scheduling.
if (!selected) {
  int priority = -1;

  for (p=proc; p<&proc[NPROC]; p++){
    acquire(&p->lock);
    if(p->state == RUNNABLE && p->level == 2){
      if (selected == 0 || p->priority > priority) { // priority
        if (selected) {
          release(&selected->lock); // release previously held lock.
        }
        selected = p;
        priority = p->priority;
      } else {
        release(&p->lock);
      }
    } else {
      release(&p->lock);
    }
  }
}
```

I've also added a context switching function, state modification and dealing with no runnable process condition.

And then the rest are the system calls.
These are the defoinition of the required system calls inside **syscall.h** and **syscall.c**.



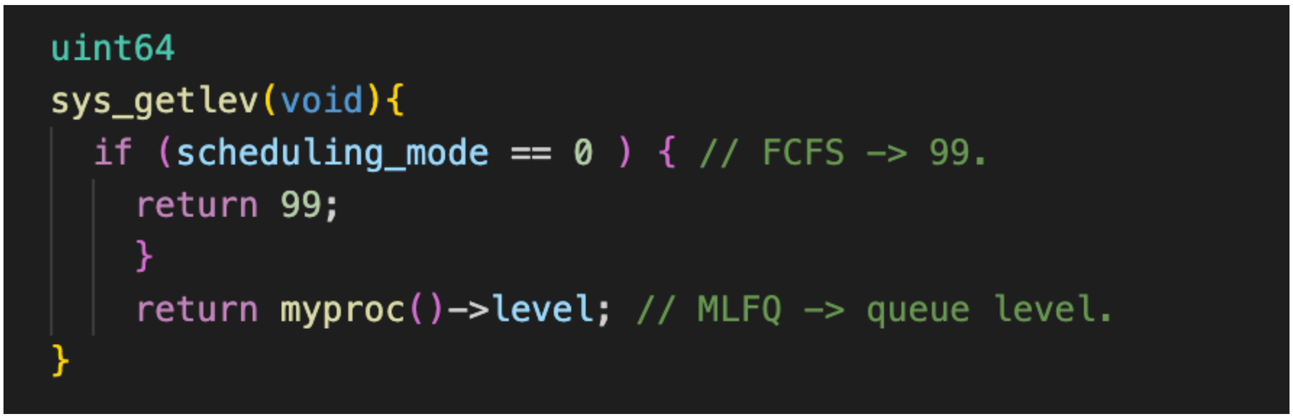
```
#define SYS_getlev 23
#define SYS_setpriority 24
#define SYS_mlfqmode 25
#define SYS_fcfsmode 26
#define SYS_yield 27
```



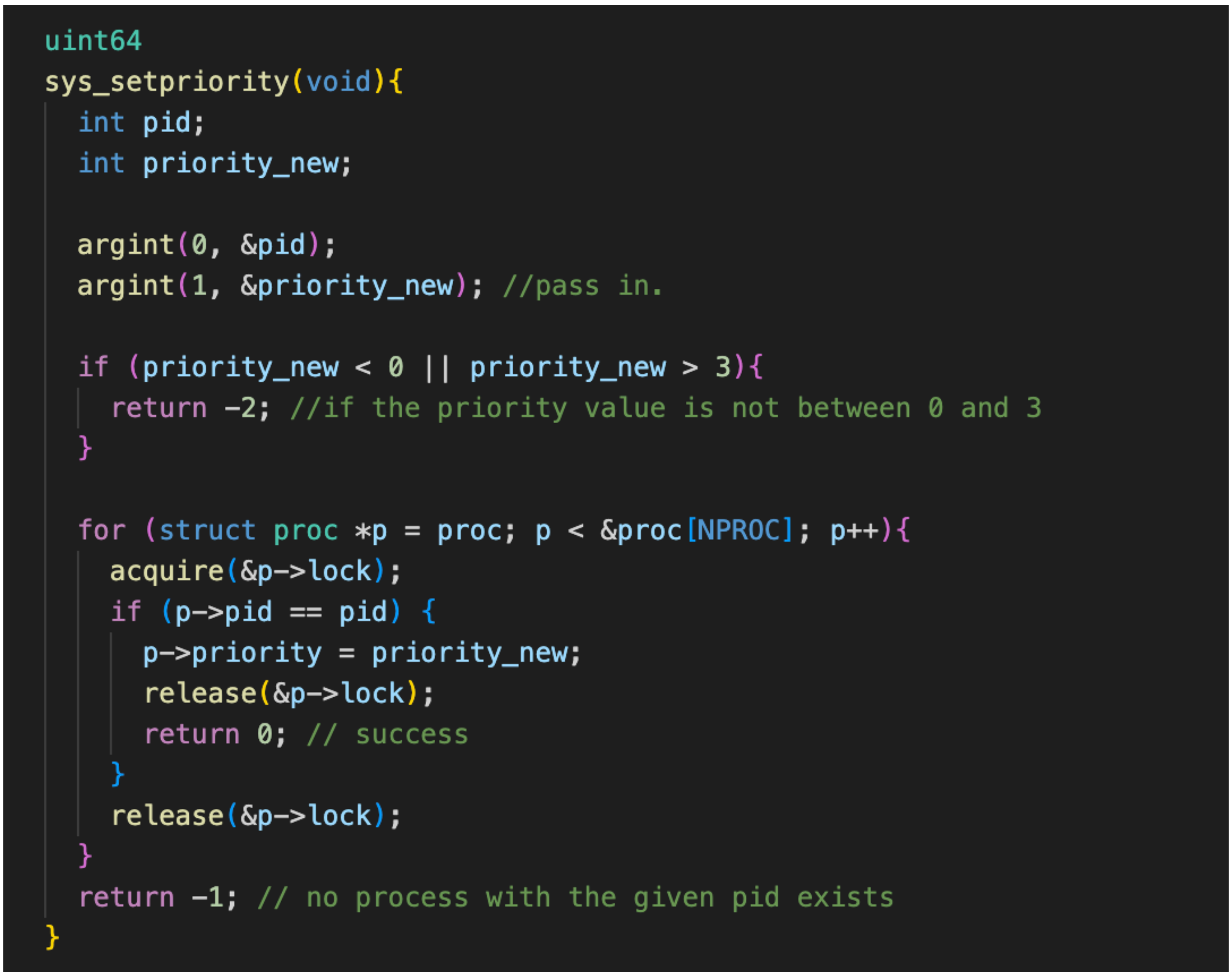
```
[SYS_getppid] sys_getppid,
[SYS_getlev] sys_getlev,
[SYS_setpriority] sys_setpriority,
[SYS_mlfqmode] sys_mlfqmode,
[SYS_fcfsmode] sys_fcfsmode,
[SYS_yield] sys_yield
```

I confused if **yield**() should be redefined, but because the project requirement was to define yield as another system call so I ended up adding the **SYS_yield** at the bottom.

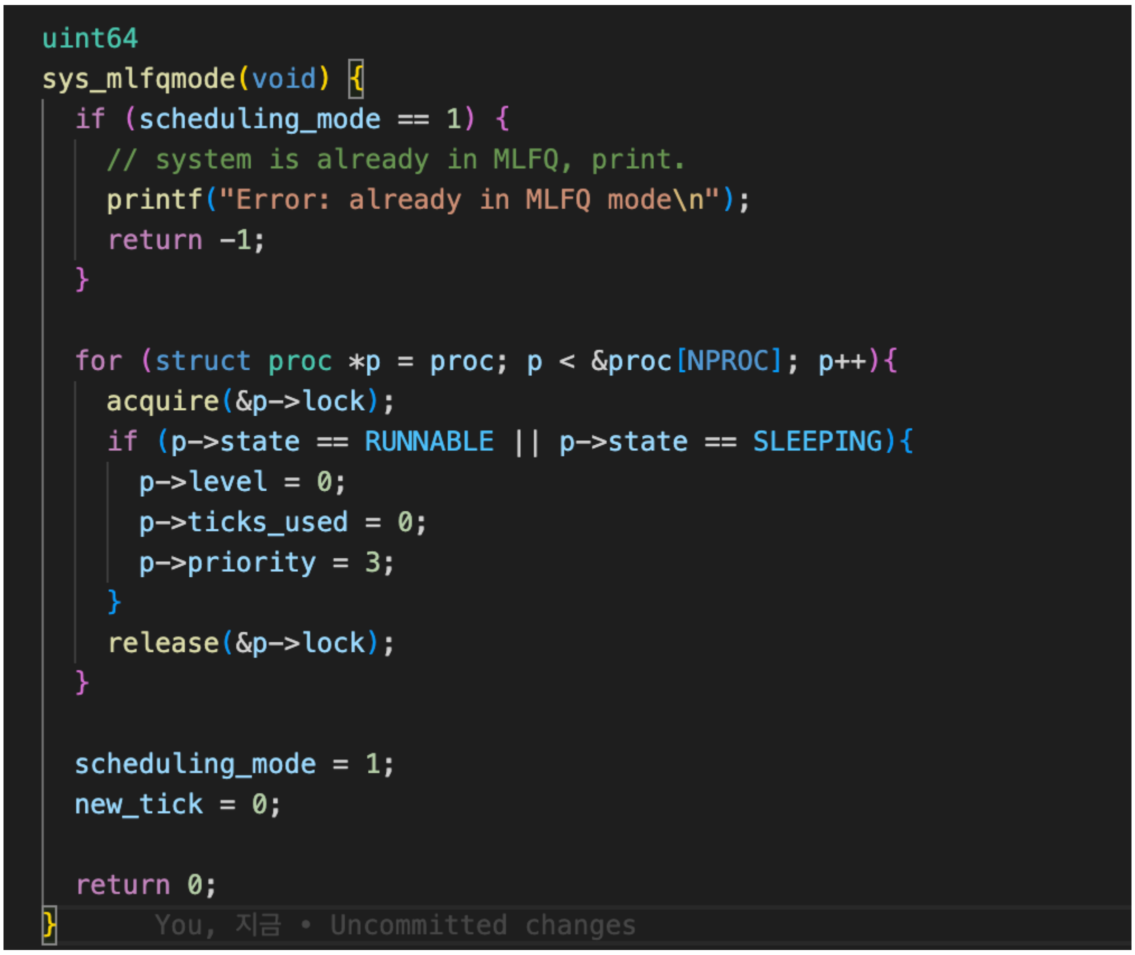
Implementation of 5 system calls in **sysproc.c**:



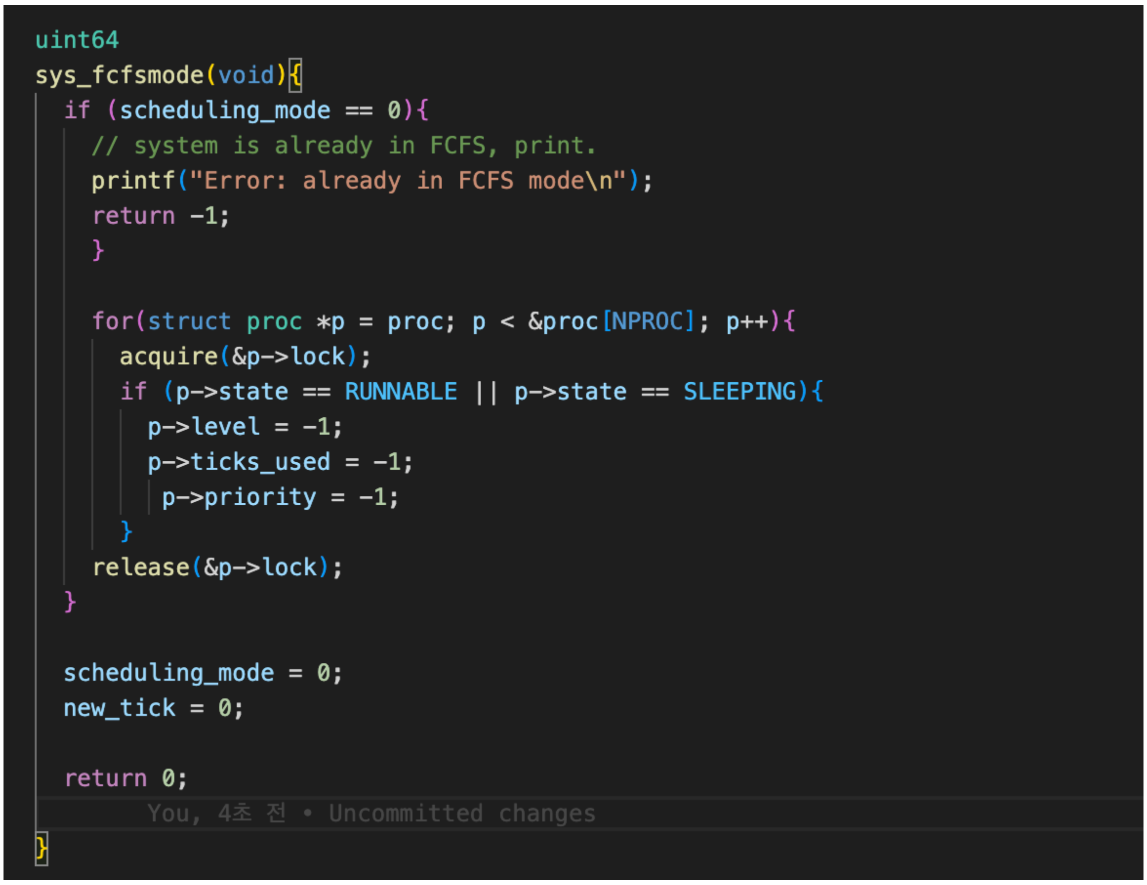
```c
uint64
sys_getlev(void){
  if (scheduling_mode == 0 ) { // FCFS -> 99.
    return 99;
    }
    return myproc()->level; // MLFQ -> queue level.
}
```



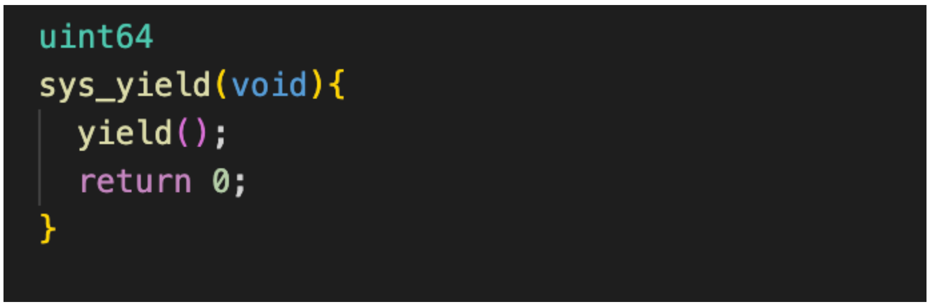
```c
uint64
sys_setpriority(void){
  int pid;
  int priority_new;

  argint(0, &pid);
  argint(1, &priority_new); //pass in.

  if (priority_new < 0 || priority_new > 3){
    return -2; //if the priority value is not between 0 and 3
  }

  for (struct proc *p = proc; p < &proc[NPROC]; p++){
    acquire(&p->lock);
    if (p->pid == pid) {
      p->priority = priority_new;
      release(&p->lock);
      return 0; // success
    }
    release(&p->lock);
  }
  return -1; // no process with the given pid exists
}
```



```
uint64
sys_mlfqmode(void) {
  if (scheduling_mode == 1) {
    // system is already in MLFQ, print.
    printf("Error: already in MLFQ mode\n");
    return -1;
  }

  for (struct proc *p = proc; p < &proc[NPROC]; p++){
    acquire(&p->lock);
    if (p->state == RUNNABLE || p->state == SLEEPING){
      p->level = 0;
      p->ticks_used = 0;
      p->priority = 3;
    }
    release(&p->lock);
  }

  scheduling_mode = 1;
  new_tick = 0;

  return 0;
}       You, 지금 • Uncommitted changes
```

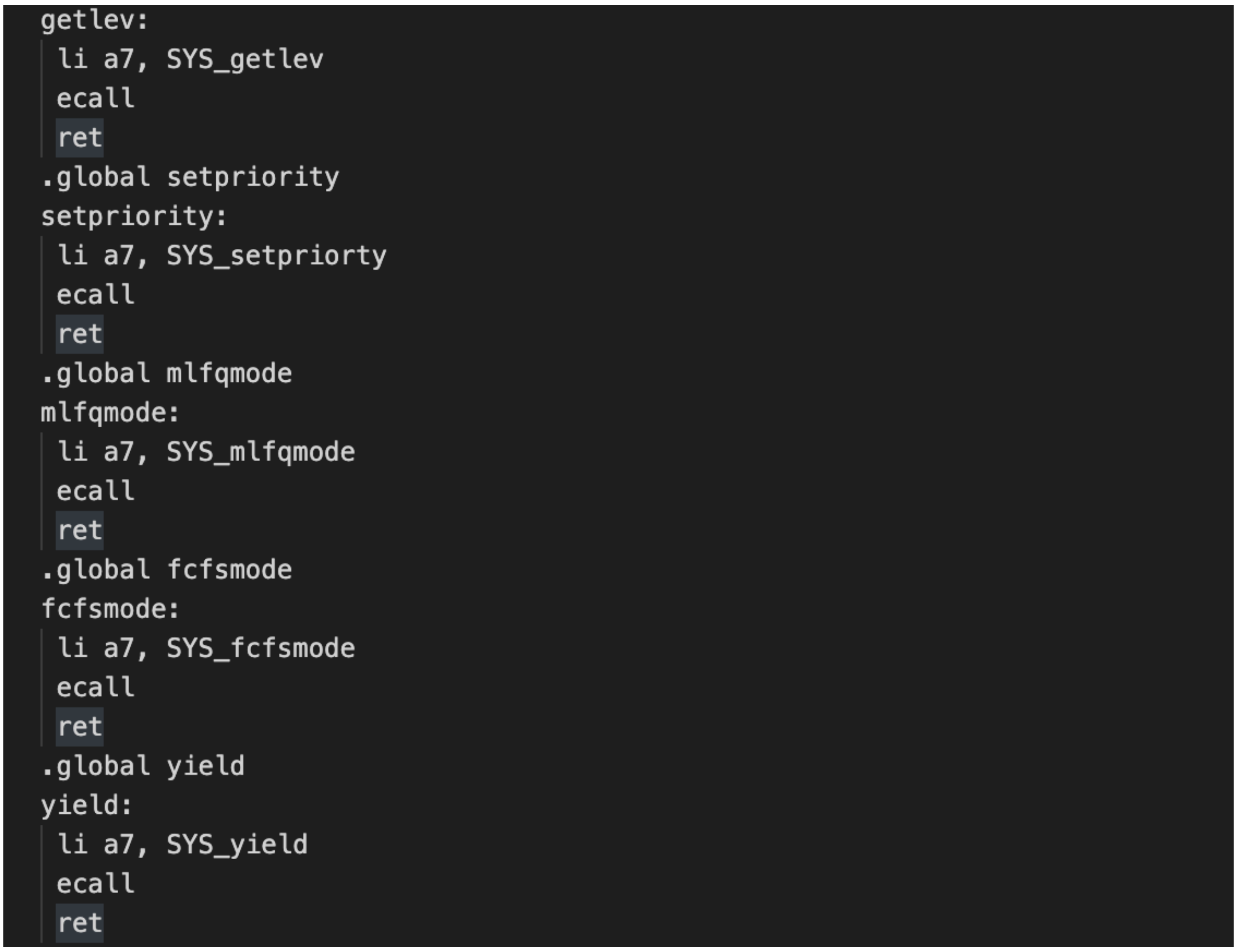


```
uint64
sys_fcfsmode(void){
  if (scheduling_mode == 0){
    // system is already in FCFS, print.
    printf("Error: already in FCFS mode\n");
    return -1;
    }

    for(struct proc *p = proc; p < &proc[NPROC]; p++){
      acquire(&p->lock);
      if (p->state == RUNNABLE || p->state == SLEEPING){
        p->level = -1;
        p->ticks_used = -1;
         p->priority = -1;
      }
    release(&p->lock);
  }

  scheduling_mode = 0;
  new_tick = 0;

  return 0;
        You, 4초 전 • Uncommitted changes
}
```



```
uint64
sys_yield(void){
  yield();
  return 0;
}
```

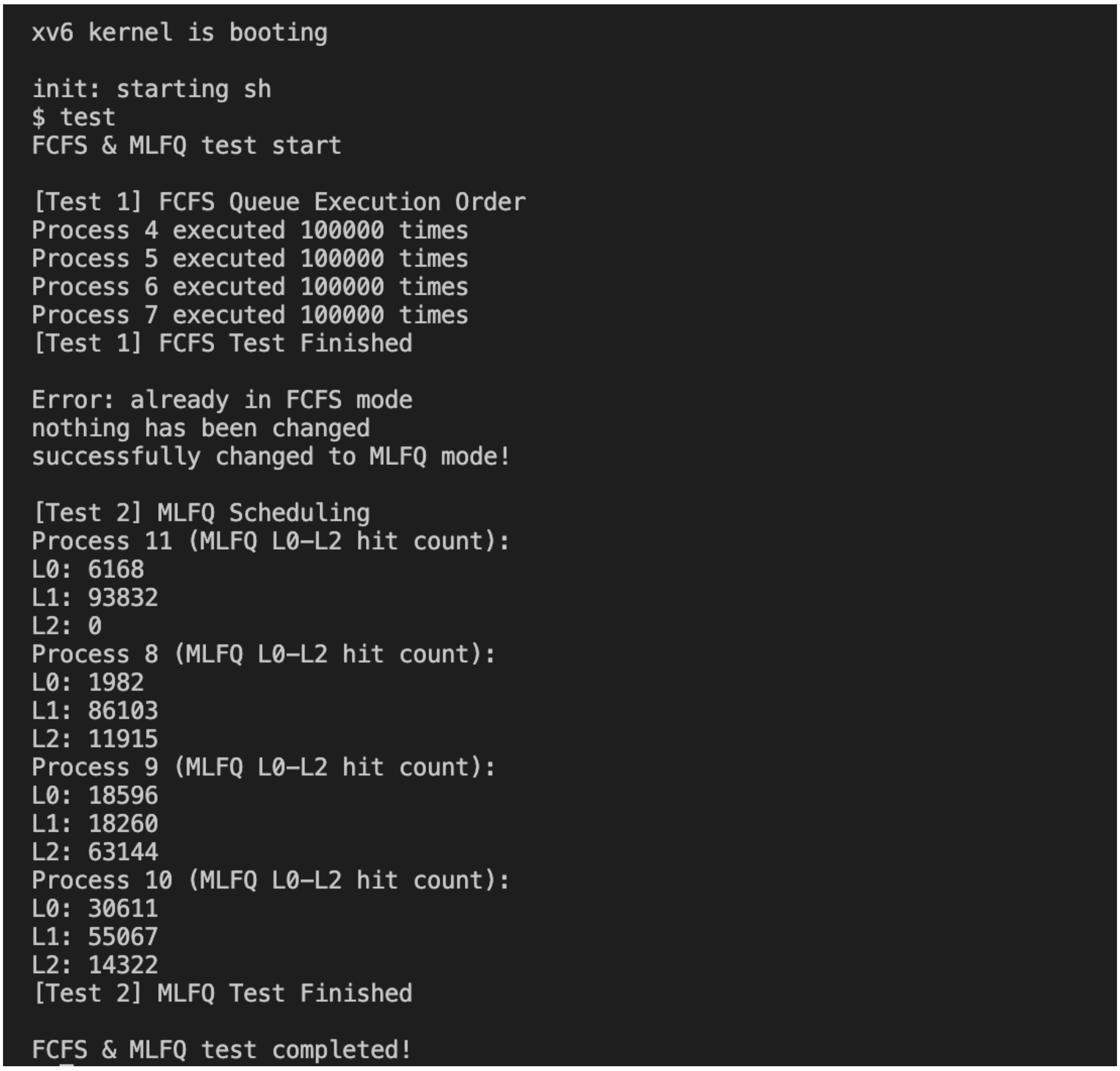
And last, I added the five system calls in usys.S. It is to expose the new system calls to user programs. These are then translated into ecall instructions that are used to enter the computer kernel.



```
getlev:
 li a7, SYS_getlev
 ecall
 ret
.global setpriority
setpriority:
 li a7, SYS_setpriorty
 ecall
 ret
.global mlfqmode
mlfqmode:
 li a7, SYS_mlfqmode
 ecall
 ret
.global fcfsmode
fcfsmode:
 li a7, SYS_fcfsmode
 ecall
 ret
.global yield
yield:
 li a7, SYS_yield
 ecall
 ret
```

III. Result

My code compilation result is as follows:



```
xv6 kernel is booting

init: starting sh
$ test
FCFS & MLFQ test start

[Test 1] FCFS Queue Execution Order
Process 4 executed 100000 times
Process 5 executed 100000 times
Process 6 executed 100000 times
Process 7 executed 100000 times
[Test 1] FCFS Test Finished

Error: already in FCFS mode
nothing has been changed
successfully changed to MLFQ mode!

[Test 2] MLFQ Scheduling
Process 11 (MLFQ L0-L2 hit count):
L0: 6168
L1: 93832
L2: 0
Process 8 (MLFQ L0-L2 hit count):
L0: 1982
L1: 86103
L2: 11915
Process 9 (MLFQ L0-L2 hit count):
L0: 18596
L1: 18260
L2: 63144
Process 10 (MLFQ L0-L2 hit count):
L0: 30611
L1: 55067
L2: 14322
[Test 2] MLFQ Test Finished

FCFS & MLFQ test completed!
```

The program flow is:

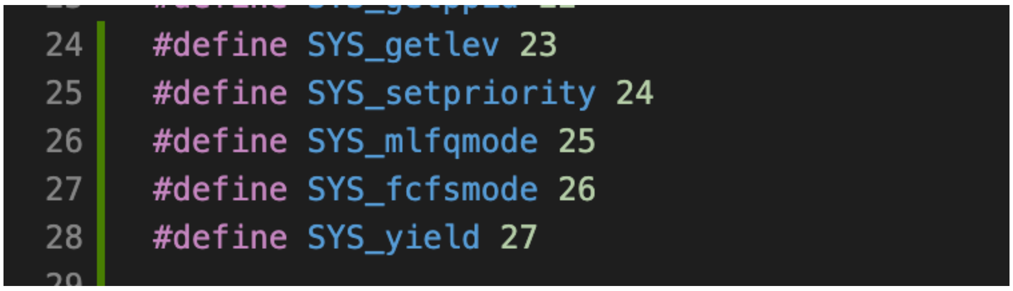
1.) start scheduler()

2.) if the scheduling mode is FCFS, then select earliest-created process (smallest pid) and run until exit.

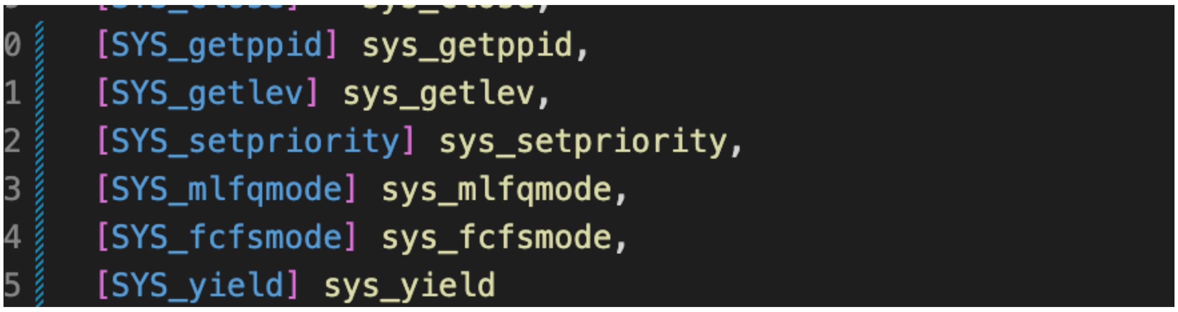
3.) if the scheduling mode is MLFQ, then it checks L0 queue, if L0 is empty then L1, then L2. Applying the demotion rule and scheduling algorithms, if they ended up using all of the time quantum then demotes. After the global tick uses upto 50 ticks then resets everything into default.

IV. Troubleshooting

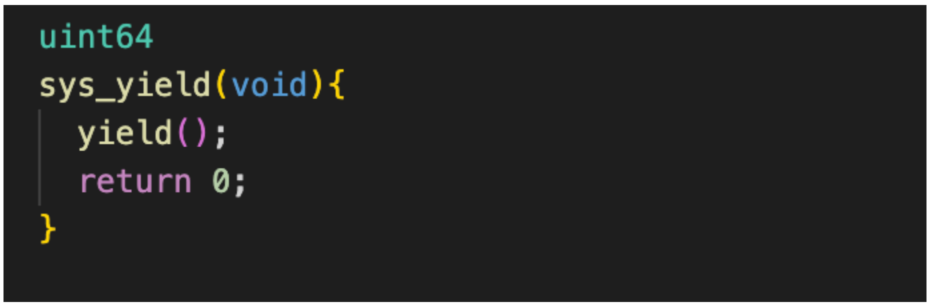
As I mentioned above, I confused if **yield**() should be redefined. Because the project requirement was to define yield as another system call so I ended up adding the **SYS_yield** at the bottom.



```
#define SYS_getlev 23
#define SYS_setpriority 24
#define SYS_mlfqmode 25
#define SYS_fcfsmode 26
#define SYS_yield 27
```

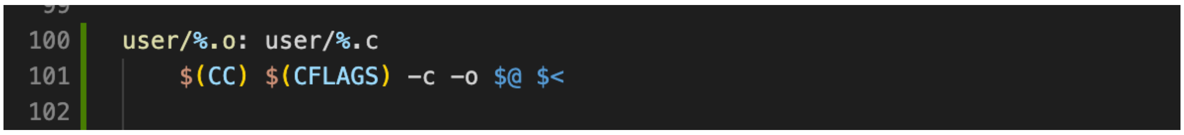


```
[SYS_getppid] sys_getppid,
[SYS_getlev] sys_getlev,
[SYS_setpriority] sys_setpriority,
[SYS_mlfqmode] sys_mlfqmode,
[SYS_fcfsmode] sys_fcfsmode,
[SYS_yield] sys_yield
```

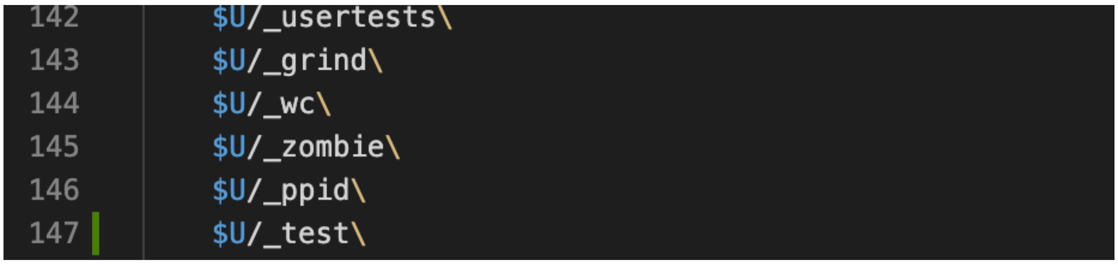


```
uint64
sys_yield(void){
  yield();
  return 0;
}
```

Because the testbench is given later, I was confused how to add the test file into my Makefile, but eventually managed to add.



```
user/%.o: user/%.c
	$(CC) $(CFLAGS) -c -o $@ $<
```



```
	$U/_usertests\
	$U/_grind\
	$U/_wc\
	$U/_zombie\
	$U/_ppid\
	$U/_test\
```