Operating Systems
Project #1 Wiki [유민수 교수님]

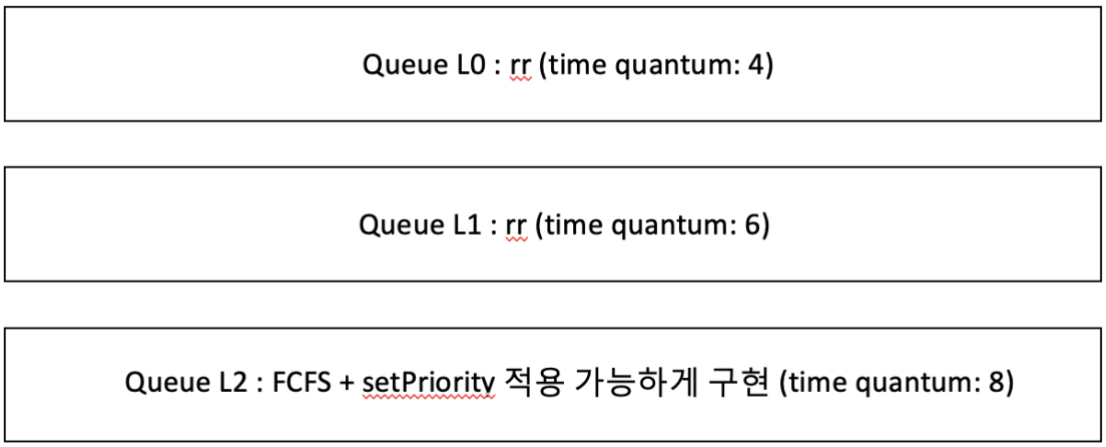
2019087192 이예진

목차

1. Design

1) MLFQ 설계

Circular Queue



Circular queue를 이용하여 L0, L1, L2를 구성하기로 하였습니다. 그 이유는 가장 효율적으로 queue를 조회할 수 있으며, 중간에 빈 공간을 최소화할 수 있기 때문입니다. 사실 단순하게 L[][]방식으로 구현을 했으면 좀 더 과제를 빨리 끝낼 수 있었을 것 같긴 했습니다. 그러나 L2에서 int priority가 낮은 값을 빠르게 찾는 과정에 있어 모든 큐를 도는 것이 아닌 size 만큼만 돌아도 되는 등의 여러 장점이 있기 때문에 circular queue자료구조를 이용하였습니다.

```
struct spinlock lock;
    struct proc proc[NPROC];
} ptable;


typedef struct proc_queue
{
    struct proc *data[NPROC];    //put in processes in data array.
    int front, rear;
    int size;
} proc_queue_t;
```

```
struct
{
    //multi-level
    proc_queue_t queue[MLFQ_NUM];
    int global_executed_ticks; //MFLQ scheduler worked ticks
} mlfq_manager; // There is a global tick in mlfq (3-level feedback queue).
```

[proc.c 상단]

구조는 위와 같습니다. 앞서 언급했듯 process가 담기는 queue는 circuler queue로 구현했기 때문에 front, rear 값 및, queue의 size 를 저장하고 있습니다..
Global tick을 어떻게 저장할지에 대한 고민도 많았습니다. Global tick을 이미 프로그램 내에 내장되어 있는 tick을 이용할까 고민했으나, mlfq_manager로 한 번 더 queue들을 감싼 뒤 이 안에 int global executed ticks를 넣어 global_tick을 관리하기로 결정하였습니다.

과제 명세에 따르면 process안에 기본으로 주어진 값들 외에도 일부 값이 추가적으로 필요함을 확인할 수 있었고, 이를 정의 하였습니다. 그 세부 내용은 아래와 같습니다.

(1) unint level : 해당 프로세스가 L0, L1, L2 중 어떤 level의 queue에 담겨있는지를 나타냅니다.
(2) unint executed_ticks: 얼마나 tick을 사용했는지 나타냅니다.
(3) int priority : L2에서 사용하는 값인 우선순위를 나타낸다. 초기 값은 3으로 담깁니다.
(4) enum lockstate lock: 현재 lock이 걸린 상태인지 나타냅니다. 이는 schedulerLock, schedulerUnlock 부분에서 사용됩니다.
Lockstate는 enum lockstate. { LOCKED, UNLOCKED } 두 가지 상태로 구성되어 있습니다.
(5) arrived_time: FCFS가 setPriority로 인해 혼란이 있을 것 같아 오류를 최소화하기 위해 넣었습니다.

```
enum lockstate {
  LOCKED, UNLOCKED
};
// Per-process state
struct proc {
  uint sz;                     // Size of process memory (bytes)
  pde_t* pgdir;                // Page table
  char *kstack;                // Bottom of kernel stack for this process
  enum procstate state;        // Process state
  int pid;                     // Process ID
  struct proc *parent;         // Parent process
  struct trapframe *tf;        // Trap frame for current syscall
  struct context *context;     // swtch() here to run process
  void *chan;                  // If non-zero, sleeping on chan
  int killed;                  // If non-zero, have been killed
  struct file *ofile[NOFILE];  // Open files
  struct inode *cwd;           // Current directory
  char name[16];               // Process name (debugging)

  //for scheduling (Project 1)
  uint level;  //Queue Level of the process (0, 1, 2 ~Q 1)
  uint executed_ticks; //executed tick count at specific level.
  int priority;
  int arrived_time; //For FCFS check the global time

  enum lockstate lock;
};
```

그림 1 [proc.h] proc에 필요한 내용 추가

Proc.h에 MLFQ안의 Queue 개수를 나타내는 MLFQ_NUM은 3으로 boosting이 일어나는 global tick을 MLFQ_GLOBAL_BOOSTING_TICK_INTERVAL =100 이라는 이름으로 필요한 상수 값을 선언해주었습니다.

2) System call 설정

(1) 모든 함수 및 함수를 감싸는 sys_(function_name) 함수 정의

실습시간에 myfunction을 만들고 호출했던 것처럼, yield()를 제외한 나머지 systemcall들은 모두 따로 파일을 만들어서 함수를 선언해주었습니다. yield를 제외한 이유는 기존의 yield함수에서 크게 달라진 점이 없기 때문입니다.

파일을 따로 만들지 않고, sysproc.c나, proc.c에 함수를 선언해준 뒤 syscall.c, syscall.h등에 등록해주는 방법도 있지만, 최대한 기존의 함수들과 분리해 설계 하고 싶어 모두 다른 파일에 함수를 하나씩 작성하였습니다.

```c
extern struct {
  struct spinlock lock;
  struct proc proc[NPROC];
} ptable;
//getLevel system call

int
getLevel(void)
{
  struct proc *p = myproc();
  acquire(&ptable.lock);
  uint level = p->level; //Fetch level of process
  release(&ptable.lock);
  if(0 <= level && level < MLFQ_NUM)
    return level;
  return -1;
}

int
sys_getLevel(void)
{
  return getLevel();
}
```

그림 2 [pj_one_get_level.c] getLevel()

```c
extern struct {
  struct spinlock lock;
  struct proc proc[NPROC];
} ptable;
//setPriority  system call

void
setPriority(int pid, int priority)
{
  struct proc *p;
  acquire(&ptable.lock);
  //Set priority
  cprintf("Set Priority - %d \n",priority);
  for(p = ptable.proc; p < &ptable.proc[NPROC]; p++) {
    if(p->pid == pid) {
      p->priority = priority;
      break;
    }
  }
  release(&ptable.lock);
  return;
}

int
sys_setPriority(void)
{
  int pid;
  int priority;
  if(argint(0, &pid) <0)
    return -1;
  if(argint(0, &priority) <0)
    return -1;
  setPriority(pid, priority);
  return 0;
}
```

그림 3 [pj_one_set_priority.c] setPriority()

```
void
schedulerLock(int password)
{
        cprintf("SchedulerLock is called...");
        if(password == 2019087192) {
                myproc()->lock = LOCKED;
                set_lockedproc();
                //global_tick_reset to 0.
                set_global_tick_zero();
                __asm__("int $129");
                exit();
        } else {
                cprintf("Password not matched!\n");
                cprintf("PID: %d TIME_QUANTUM: %d CURRENT_LEVEL: %d", myproc()->pid, myproc()
                ->executed_ticks, myproc()->level);
                kill(myproc()->pid);
        }
}

int
sys_schedulerLock(void)
{
        int password;
        if(argint(0, &password) <0)
                return -1;
        schedulerLock(password);
        return 0;
}
```

그림 4 [pj_one_scheduler_lock.c] schedulerLock()

```
//schedulerUnlock  system call
void
schedulerUnlock(int password)
{
        cprintf("SchedulerUnlock is called...\n");
        if(password == 2019087192) {
                myproc()->lock = UNLOCKED;
                withdraw_lock();
                __asm__("int $130");
                exit();
        } else {
                cprintf("Password not matched!\n");
                cprintf("PID: %d TIME_QUANTUM: %d CURRENT_LEVEL: %d", myproc()->pid, myproc()
                ->executed_ticks, myproc()->level);
                kill(myproc()->pid);
        }
}

int
sys_schedulerUnlock(void)
{
        int password;
        if(argint(0, &password) <0)
                return -1;
        schedulerUnlock(password);
        return 0;
}
```

그림 5 [pj_one_scheduler_unlock.c] schedulerUnlock()

(2) Makefile에 source file을 적어주었습니다.

```
prac_syscall.o\
pj_one_get_level.o\
pj_one_set_priority.o\
pj_one_scheduler_lock.o\
pj_one_scheduler_unlock.o\
```

그림 6 [Makefile]

(3) 다른 c files들이 볼 수 있게 defs.h에 위 함수를 선언해주었습니다.

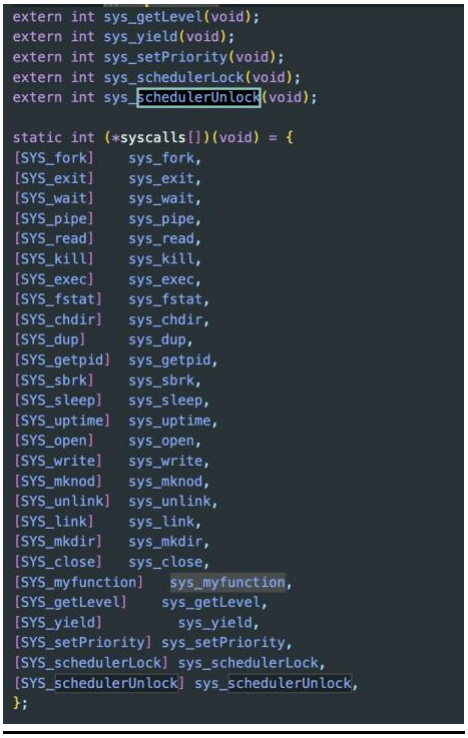
```
//pj_one_get_level.c
int getLevel(void);

//pj_one_set_priority.c
void setPriority(int, int);

//pj_one_scheduler_lock.c
void schedulerLock(int);

//pj_one_scheduler_unlock.c
void schedulerUnlock(int);
```

그림 7 [defs.h]

(4) 새로운 system call을 syscall.h와 syscall.c에 등록해주었습니다.

```
#define SYS_getLevel 23
#define SYS_yield 24
#define SYS_setPriority 25
#define SYS_schedulerLock 26
#define SYS_schedulerUnlock 27
```

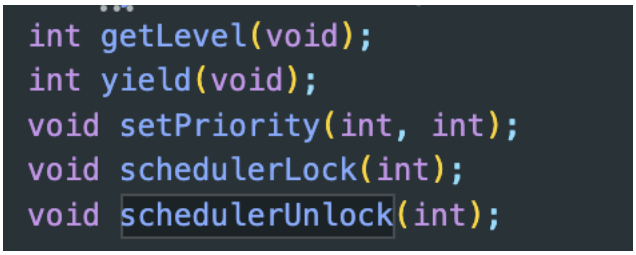
그림 8 syscall.h



```
extern int sys_getLevel(void);
extern int sys_yield(void);
extern int sys_setPriority(void);
extern int sys_schedulerLock(void);
extern int sys_schedulerUnlock(void);

static int (*syscalls[])(void) = {
[SYS_fork]    sys_fork,
[SYS_exit]    sys_exit,
[SYS_wait]    sys_wait,
[SYS_pipe]    sys_pipe,
[SYS_read]    sys_read,
[SYS_kill]    sys_kill,
[SYS_exec]    sys_exec,
[SYS_fstat]   sys_fstat,
[SYS_chdir]   sys_chdir,
[SYS_dup]     sys_dup,
[SYS_getpid]  sys_getpid,
[SYS_sbrk]    sys_sbrk,
[SYS_sleep]   sys_sleep,
[SYS_uptime]  sys_uptime,
[SYS_open]    sys_open,
[SYS_write]   sys_write,
[SYS_mknod]   sys_mknod,
[SYS_unlink]  sys_unlink,
[SYS_link]    sys_link,
[SYS_mkdir]   sys_mkdir,
[SYS_close]   sys_close,
[SYS_myfunction]   sys_myfunction,
[SYS_getLevel]    sys_getLevel,
[SYS_yield]        sys_yield,
[SYS_setPriority] sys_setPriority,
[SYS_schedulerLock] sys_schedulerLock,
[SYS_schedulerUnlock] sys_schedulerUnlock,
};
```

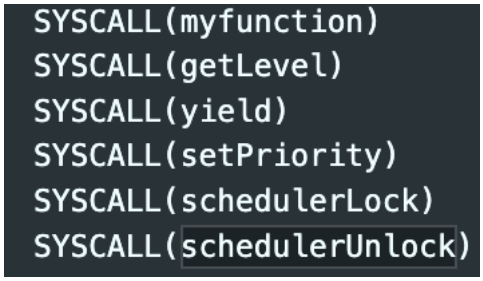
그림 9syscall.c

(5) 유저 모드에서 사용 가능하게 설계 해야 하기 때문에 user.h, usys.S 에 함수들을 정의하였습니다.



```
int getLevel(void);
int yield(void);
void setPriority(int, int);
void schedulerLock(int);
void schedulerUnlock(int);
```

그림 10 [user.h]



```
SYSCALL(myfunction)
SYSCALL(getLevel)
SYSCALL(yield)
SYSCALL(setPriority)
SYSCALL(schedulerLock)
SYSCALL(schedulerUnlock)
```

그림 11 [usys.S]

3) proc.c 설계

mlfq_scheduler_proc.c 등으로 분리하지 않고 proc.c를 수정하는 방식으로 코드를

구현하였습니다.

2) Traps, and Interrupts (chedulerLock() / schedulerUnlock()) 구조 설계

129번 인터럽트 호출 시, schedulerLock()이라는 시스템콜 실행 시키기
130번 인터럽트 호출 시, schedulerUnLock()이라는 시스템콜 실행 시키기
를 구현해야하기 때문에 trap.c에
SETGATE(idt[T_INT129], 1, SEG_KCODE<<3, vectors[T_INT129], DPL_USER);
SETGATE(idt[T_INT130], 1, SEG_KCODE<<3, vectors[T_INT130], DPL_USER); 및 해당 trap이 실행될 때 받아오는 부분을 설정하였습니다..

```
void
tvinit(void)
{
  int i;

  for(i = 0; i < 256; i++)
    SETGATE(idt[i], 0, SEG_KCODE<<3, vectors[i], 0);
  SETGATE(idt[T_SYSCALL], 1, SEG_KCODE<<3, vectors[T_SYSCALL], DPL_USER);
  SETGATE(idt[T_INT128], 1, SEG_KCODE<<3, vectors[T_INT128], DPL_USER);
  SETGATE(idt[T_INT129], 1, SEG_KCODE<<3, vectors[T_INT129], DPL_USER);
  SETGATE(idt[T_INT130], 1, SEG_KCODE<<3, vectors[T_INT130], DPL_USER);

  initlock(&tickslock, "time");
}
```

그림 12 [trap.c]

이 때 lock이 걸렸는지의 여부에 대해 proc.c에서 알 수 있어야 하기 때문에

```
struct proc *lockedproc = 0;
```

그림 13[proc.c]

Proc.c 파일 안에 lockedproc이라는 변수를 선언하고, locked된 process가 있으면 그 process를 담고, lockedproc이 없으면 0이라는 값을 담았습니다.

```
else if(tf->trapno == T_INT129){
  if(myproc()->killed)
    exit();
  myproc()->tf = tf;
  cprintf("user interrupt 129 called!\n");
  if(myproc()->killed)
    exit();
  return;
}
  else if(tf->trapno == T_INT130){
  if(myproc()->killed)
    exit();
  myproc()->tf = tf;
  // myproc()->priority = 3;
  // myproc()->executed_ticks = 0;
  cprintf("user interrupt 130 called!\n");
  //schedulerUnlock
  if(myproc()->killed)
    exit();
  return;
}
```

그림 14 [trap.c]

2. Implement

1) MLFQ 작동 구현

Process가 실행이 시작되면 운영체제는 process의 우선순위에 따라 위에 있는 3개의 Queue중 하나에 Process를 삽입할 수 있습니다. 만약 queue L0에 process가 들어갔다고 했을 때, 4tick이 지나도 완료되지 않으면 우선순위가 감소하고 L1 queue의 process들 뒤로 내려옵니다. L1에서도 L0와 같게 작동하는데 이때의 time quantum은 8입니다. 마지막 queue에서 process들은 기본적으로 FCFS로 작동하며 이 때 priority가 작은 값부터 수행되게 합니다.

이 때 L0, L1에서 스케줄링 될 때는 RR로 작동하기 때문에 예를 들어 L0에 A, B라는 Process가 L0에 있다고 하면, A,B,A,B,A,B,A(큐 이동),B(큐 이동)로 CPU를 할당받습니다. (Time Quantum은 Round Robin에서 특정 프로세스가 최대로 이용할 수 있는 CPU시간이기 때문에 각각 1tick씩 총 4tick을 할당 받게 되고 다음 level 큐로 넘겨줍니다.)

0) pinit()함수를 실행 시 mlfq_init()을 통한 초기화 진행

```
void mlfq_init()
{
  int lev;
  proc_queue_t *queue;

  for (lev = 0; lev < MLFQ_NUM; ++lev)
  {
    queue = &mlfq_manager.queue[lev];
    memset(queue->data, 0, sizeof(struct proc *) * NPROC);
    queue->front = 1;
    queue->rear = 0;
    queue->size = 0;

    for(int i = 0; i < NPROC; i++)
    {
      if(queue->data[i] == 0 && (queue->data[i] == mlfq_manager.queue[lev].data[i]))
        cprintf("[L%d], Successfully allocated at index %d\n", lev, i);
    }
  }

  mlfq_manager.global_executed_ticks = 0;

}
```

그림 15 [proc.c] mlfq_init

설계한 mlfq를 모두 초기화 해줍니다. 이 때 memset을 통해 queue안의 모든 값을 0으로 초기화 하였으며, circuler queue에 필요한 front, rear, size를 모두 초기화 해주었습니다. Global tick 역시 0으로 초기화 해줍니다.

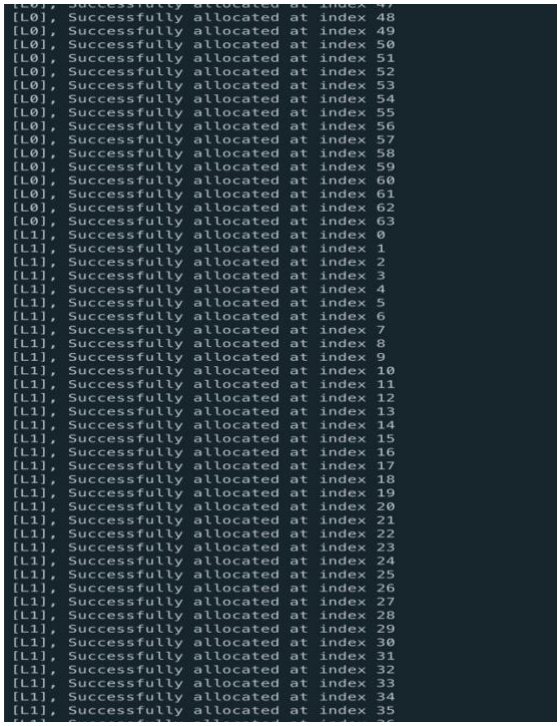


그림 16 잘 할당이 되고 있는 지 확인해본 모습

1) allocproc() 함수

```
static struct proc*
allocproc(void)
{
  struct proc *p;
  char *sp;

  acquire(&ptable.lock);

  for(p = ptable.proc; p < &ptable.proc[NPROC]; p++)
    if(p->state == UNUSED)
      goto found;

  release(&ptable.lock);
  return 0;

found:
  p->state = EMBRYO;
  p->pid = nextpid++;
  p->priority = 3; //if Priority boosting work, prioriy
  reset to 3.
  p->executed_ticks=0;
  p->level=0;
  p->lock = UNLOCKED;
  p->arrived_time = 0;

  cprintf("initialized %d\n", p->pid);
  //first push p in queue
  if (mlfq_enqueue(0, p) != 0)
  {
    cprintf("allocation error %d\n", p->pid );
    release(&ptable.lock);
    return 0;
  }

  release(&ptable.lock);
```

그림 17 [proc.c] 초기화 및 proc값 하나 할당.

추가로 설정해준 값들 역시 모두 기본 값으로 초기화 합니다. 즉 p->executed_ticks=0; p->level=0; p->lock = UNLOCKED; p->arrived_time = 0; 이렇게 값들을 모두 초기화 해주었습니다.

여기서 mlfq_enqueue()라는 queue에 process넣는 함수를 분리하였습니다. (이후에도 다수 동일한 logic이 사용되기 때문에 함수로 처리하였습니다.) 기존의 circular queue의 구조를 가져와 rear값을 1씩 더해주면서 queue 끝에 프로세스를 하나씩 넣어주는 형태를 취하고 있습니다. 이 때 인자로 lev값을 받아와 p->lev값을 업데이트 해줍니다. 이미 queue가 가득 차 있으면 -1을

return 하도록 하였습니다. 즉 allocproc을 실행하면 mlfq_enqueue(0, p)가 실행되어 lev0에 하나의 process가 들어가게 됩니다.

```
int mlfq_enqueue(int lev, struct proc *p)
{
  proc_queue_t *const queue = &mlfq_manager.queue[lev];

  if (queue->size == NPROC)
    return -1; //The queue is full.
  queue->rear = (queue->rear + 1) % NPROC;
  queue->data[queue->rear] = p;
  queue->size = queue->size +1;

  p->level = lev;

  return 0;
}
```

그림 18[proc.c mlfq_enqueue()]

2) scheduler() 함수

계속해서 scheduler를 돌게 될 텐데 이 scheulder 함수는 크게 lock이 걸렸을 때 실행해하는 부분과 lock이 걸리지 않았을 때 실행해야 하는 부분으로 나누었습니다. If(lockedproc !=0)을 기준으로 분기하였습니다. 이 때 mlfq_select라는 함수를 통해 이번에 돌아야하는 값을 찾아옵니다. mlfq_select에서는 실행해야 하는 process를 못 찾으면 0을 리턴해주고, 잘 찾으면 그 process를 리턴해줍니다. 그 이후는 기존의 proc.c에 있던 것과 같습니다. 단, global_excuted_ticks MLFQ_GLOBAL_BOOSTING_TICK_INTERVAL을 넘어가면 priority boost가 필요합니다.

```
    } else {
      SCHEDULER:
        p = mlfq_select(); //select mlfq which to execute.

        if(p != 0)
        {
          // 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;
        }
        if(mlfq_manager.global_executed_ticks >=
        MLFQ_GLOBAL_BOOSTING_TICK_INTERVAL) {
          mlfq_priority_boost();
        }
    }
    release(&ptable.lock);
  }
}
```

3) mlfq_select() 함수

이것은 크게 lev< 2일 때와 lev ==2일 때로 분기해 코드를 구현하였습니다.

L0, L1일 때는 그저 process가 RUNNABLE일 때 값을 찾아나가면 됩니다. 그러나 L2일 때는 priority int가 가장 작으면서도 priority가 낮은 값들 중에는 arrived_time이 작은 값이 먼저 실행되게끔 하였습니다. 이를 구현하던 중, FCFS를 구현할 때도 맨 처음에만 먼저 도착한 process를 실행 시키고 그 이후에는 계속 또 같은 priority를 가진 process끼리는 번갈아가며 사용해야한다고 하셔서, process를 실행할 때마다 arrived_time을 global_ticks로 업데이트 해주었습니다. 그렇게 해서.priority가 같은 그 다음 process도 실행될 수 있게끔 하였습니다.

```
while (1)
{
  //each queue size check
  for (; lev < MLFQ_NUM; ++lev)
  {
    if (mlfq_manager.queue[lev].size > 0)
      break;
  }
  // no process in the mlfq (empty)
  if (lev == MLFQ_NUM)
    return 0;

  size = mlfq_manager.queue[lev].size;
  proc_queue_t *const queue = &mlfq_manager.queue[lev];
  if(lev < 2) {
    for (i = 0; i < size; ++i)
    {
      ret = queue->data[queue->front];
      if(ret->state == RUNNABLE) {
        goto found;
      } else {
        mlfq_dequeue(lev, 0);
        mlfq_enqueue(lev, ret);
      }
    }
  } else if (lev == 2) {
      //find the lowest priority in L2 queue
      int priority = 10000;
      int found_i = 10000;
      int arrived_t = 10000;
      //start front to end rear;
      for(int i = 0; i < size; i++){
        int idx = (queue->front + i) % NPROC;
        ret = queue->data[idx];
        //only ret->state == RUNNABLE -> found_idx can be changed.
        if((ret->priority < priority) || (ret->priority == priority && ret->arrived_time < arrived_t)) {
          found_i = i;
          priority = ret->priority;
          arrived_t = ret->arrived_time;
        }
      }
      if(priority != 10000 && found_i != 10000) {
        //for found_idx and found_idx를 맨 앞으로 보내라.
        int i = 0;
        while(i < 10000) {
          ret =  queue->data[queue->front];
          if (i == found_i) {
            break;
          } else {
            mlfq_dequeue(lev, 0);
            mlfq_enqueue(lev, ret);
          }
          i++;
        }
        // ret = queue->data[found_idx];
        goto found;
      }
  }
  // queue has no runnable process
  // then find candidate at next lower queue
  ++lev;
}
```

```
found: //if runnable process found.
  // cprintf("\n\nFOUND PID: %d\n", ret->pid);
  (ret->executed_ticks)++;
  (mlfq_manager.global_executed_ticks)++;
  //pass the process to lev+1 queue.
  //case L0, L1 && if executed_ticks full.
  if (lev < MLFQ_NUM - 1 && ret->executed_ticks >= MLFQ_TIME_QUANTUM[lev])
  {
    //dequeue ret from current tree
    mlfq_dequeue(lev, 0);
    //enqueue ret to current lev + 1 queue
    mlfq_enqueue(lev + 1, ret);

    //executed_ticks reset to 0
    ret->executed_ticks = 0;
    if (lev == MLFQ_NUM -2) {
      ret->arrived_time = mlfq_manager.global_executed_ticks;
    }
  } else if(lev == MLFQ_NUM -1  && ret->executed_ticks >= MLFQ_TIME_QUANTUM[lev]) {
    ret->executed_ticks = 0;
    //if ret->priority == 0, just keep
    if(ret->priority > 0) {
      ret->priority = ret->priority -1; //prority -
    } else {
      ret->priority = 0;
    }
    ret->arrived_time =  mlfq_manager.global_executed_ticks;
    mlfq_dequeue(lev, 0);
    mlfq_enqueue(lev, ret);
  } else if (ret->executed_ticks % 1 == 0){
    //change the sequence of the queue.
    mlfq_dequeue(lev, 0);
    mlfq_enqueue(lev, ret);
  }
  return ret;
}
```

그림 19 [proc.c] found일 때

위와 같이 한 번씩 실행해야할 process를 찾을 때마다 ret->executed_ticks 및 global_ticks를 올려줬고, lev ==2에서 프로세스를 실행 및 lev=1에서 lev=2로 넘어가는 시점에는 arrived_time을 mlfq_manager.global_excuted_ticks로 변경해주었습니다.

4) mlfq_priority_boost() 함수

Global tick이 100ticks가 될 때 모든 프로세스들은 L0큐로 재조정됩니다.

재조정 될 때, for 문을 돌면서

1) 모든 프로세스들은 L0큐로 재조정

2) 모든 프로세스들의 priority값들은 3으로 재설정

3) 모든 프로세스들의 time quantum은 초기화
를 진행해주었습니다 .이와 더불어 FCFS를 위해 넣은 arrived_time 값도 초기화 해주었습니다.

```
static void
mlfq_priority_boost(void)
{
  struct proc *p;
  int lev;
  for (lev = 1; lev < MLFQ_NUM; ++lev)
  {
    while (mlfq_manager.queue[lev].size) //if each queue is not empty
    {
      //dequeue and then enqueue to L0.
      mlfq_dequeue(lev, &p);
      mlfq_enqueue(0, p);
      p->priority = 3;
      p->executed_ticks = 0; //time quantum reset.
      p->arrived_time = 0;
    }
  }
  mlfq_manager.global_executed_ticks = 0;
}
```

그림 20 [proc.c]

2) Traps, and Interrupts (schedulerLock() / schedulerUnlock()) 구조 설계

129번 인터럽트 호출 시, schedulerLock()이라는 시스템콜 실행 시키기
130번 인터럽트 호출 시, schedulerUnLock()이라는 시스템콜 실행 시키기

Withdraw_lock이라는 함수를 만들어 Locked 된 상태에서 풀어지는 상황을 구현하였다. 이 때 L0의 맨 앞으로 해당 proc을 올려주는 로직을 구현하였습니다. Design단계에서 설정한 lockedproc값에 myproc()을 넣게 되고 withdraw_lock의 마지막에서는 다시 0으로 그 값을 바꿔줍니다. 그 외에도 set_lockproc()이라는 함수와 set_global_tick_zero()라는 함수를 구현해 현재 실행되는 myproc()을 lockedproc에 넣어주는 작업 및 global_executed_ticks 를 0으로 바꿔주는 작업을 진행하였습니다.
;

```c
void
withdraw_lock(void) {
  cprintf("withdraw_lock is called...\n");
  print_mlfq();
  int err = 0;
  if(lockedproc != 0) {
    //remove the lockedproc from the certain queue.
    proc_queue_t *const remove_queue = &mlfq_manager.queue[lockedproc->level];
    struct proc *p;
    // if queue is empty return  -1(error);
    if(remove_queue->size == 0) {
      cprintf("The queue is empty");
      err = 1;
    } else {
      p = remove_queue->data[remove_queue->front];
      //fill data = 0
      remove_queue->data[remove_queue->front] = 0;

      remove_queue->front = (remove_queue->front + 1) % NPROC;
      //front + 1;
      (remove_queue->size)--;
      //size -1
      p->level = 0;
    }
    if(err != 1) {
      /*pass the process to first of L0.*/
      proc_queue_t *const add_queue = &mlfq_manager.queue[0];
      if (add_queue->size == NPROC) {
        cprintf("The queue is full");
      } else {
        if(add_queue->front == 0) {
          add_queue->front = NPROC -1;
          //add_queue->data[63];
        } else {
          add_queue->front = (add_queue->front - 1) % NPROC;
        }
        //reset the certain process's time quantum.
        lockedproc->executed_ticks = 0;
        //reset the process's priority to 3
        lockedproc->priority = 3;
        lockedproc->lock = UNLOCKED;
        add_queue->data[add_queue->front] = lockedproc;
        (add_queue->size)++;
        p->level = 0;
        lockedproc = 0;
      }
    }
  }
  cprintf("withdraw_lock is finished...\n");
  return;
}
```

그림 21 [proc.c] withdraw_lock.c

```
void
schedulerLock(int password)
{
        cprintf("SchedulerLock is called...");
        if(password == 2019087192) {
                myproc()->lock = LOCKED;
                set_lockedproc();
                //global_tick_reset to 0.
                set_global_tick_zero();
                __asm__("int $129");
                exit();
        } else {
                cprintf("Password not matched!\n");
                cprintf("PID: %d TIME_QUANTUM: %d CURRENT_LEVEL: %d", myproc()->pid, myproc()
                ->executed_ticks, myproc()->level);
                kill(myproc()->pid);
        }
}

int
sys_schedulerLock(void)
{
        int password;
        if(argint(0, &password) <0)
                return -1;
        schedulerLock(password);
        return 0;
}
```

그림 22 [pj_one_scheduler_lock] schedulerLock함수

```
//schedulerUnlock  system call
void
schedulerUnlock(int password)
{
        cprintf("SchedulerUnlock is called...\n");
        if(password == 2019087192) {
                myproc()->lock = UNLOCKED;
                withdraw_lock();
                __asm__("int $130");
                exit();
        } else {
                cprintf("Password not matched!\n");
                cprintf("PID: %d TIME_QUANTUM: %d CURRENT_LEVEL: %d", myproc()->pid, myproc()
                ->executed_ticks, myproc()->level);
                kill(myproc()->pid);
        }
}

int
sys_schedulerUnlock(void)
{
        int password;
        if(argint(0, &password) <0)
                return -1;
        schedulerUnlock(password);
        return 0;
}
```

그림 23 [pj_one_scheduler_unlock] schedulerUnlock함수

둘 다 password가 틀릴 때는 에러를 반환합니다. 이 때 PID, TQ, current Level을 출력하도록 구현했으며 kill을 호출해서 실행 proc을 죽입니다.

3. Result

조교님께서 올려주신 테스트 Test Case에 대한 결과 및 제가 구현 과정에서 cprintf를 통해 잘 작동하고 있는지 출력한 결과 입니다.

(1) mlfq 작동 결과

Mlfq_print로 각 큐에 값들이 잘 들어가고 있는지 확인해보았습니다. 이 때 1 tick씩 이동하면서 L0, L1가 RR방식으로 잘 작동하고 있는 것을 확인할 수 있습니다.

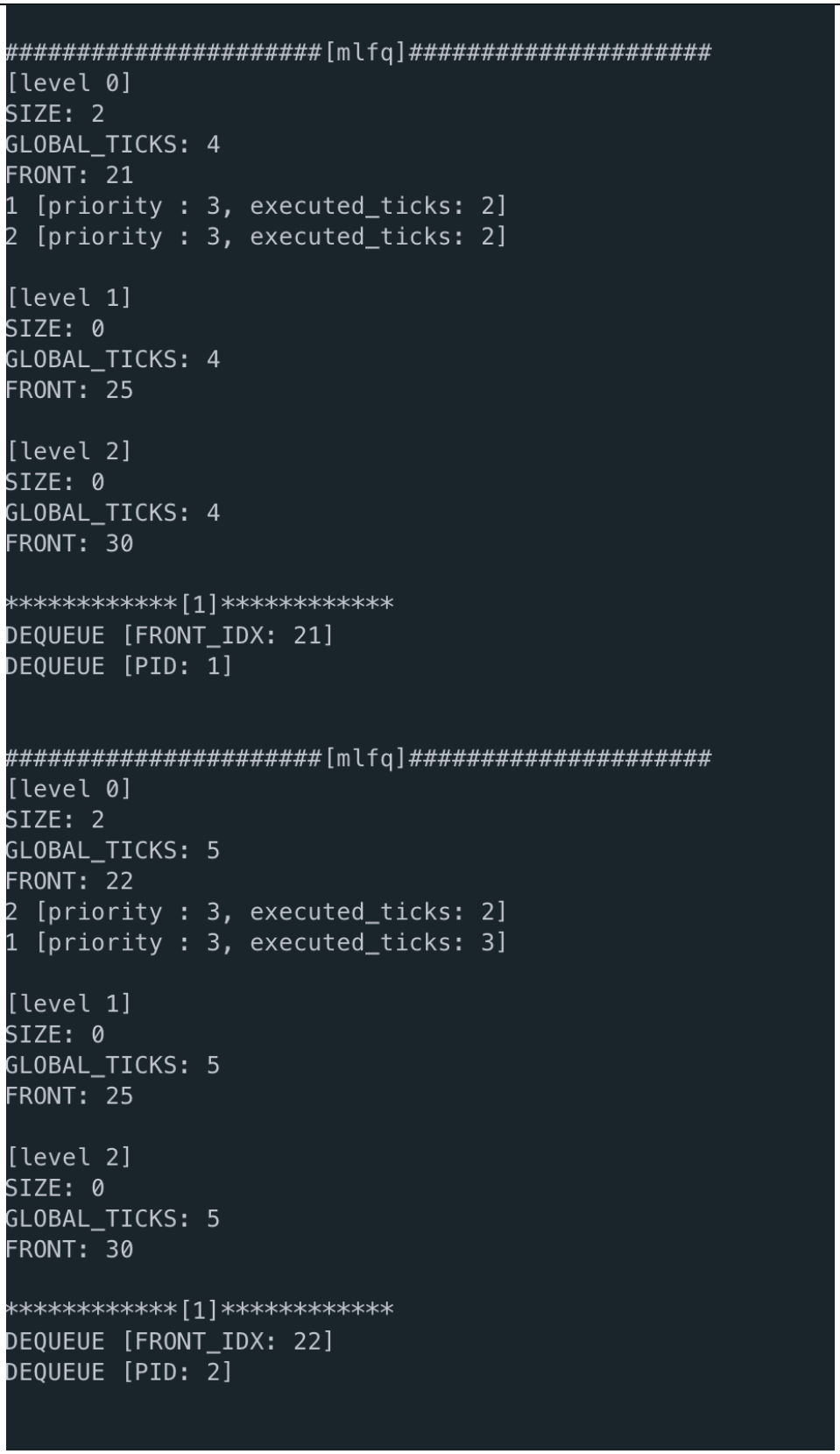


그림 24 1틱씩 ABAB RR방식 작동

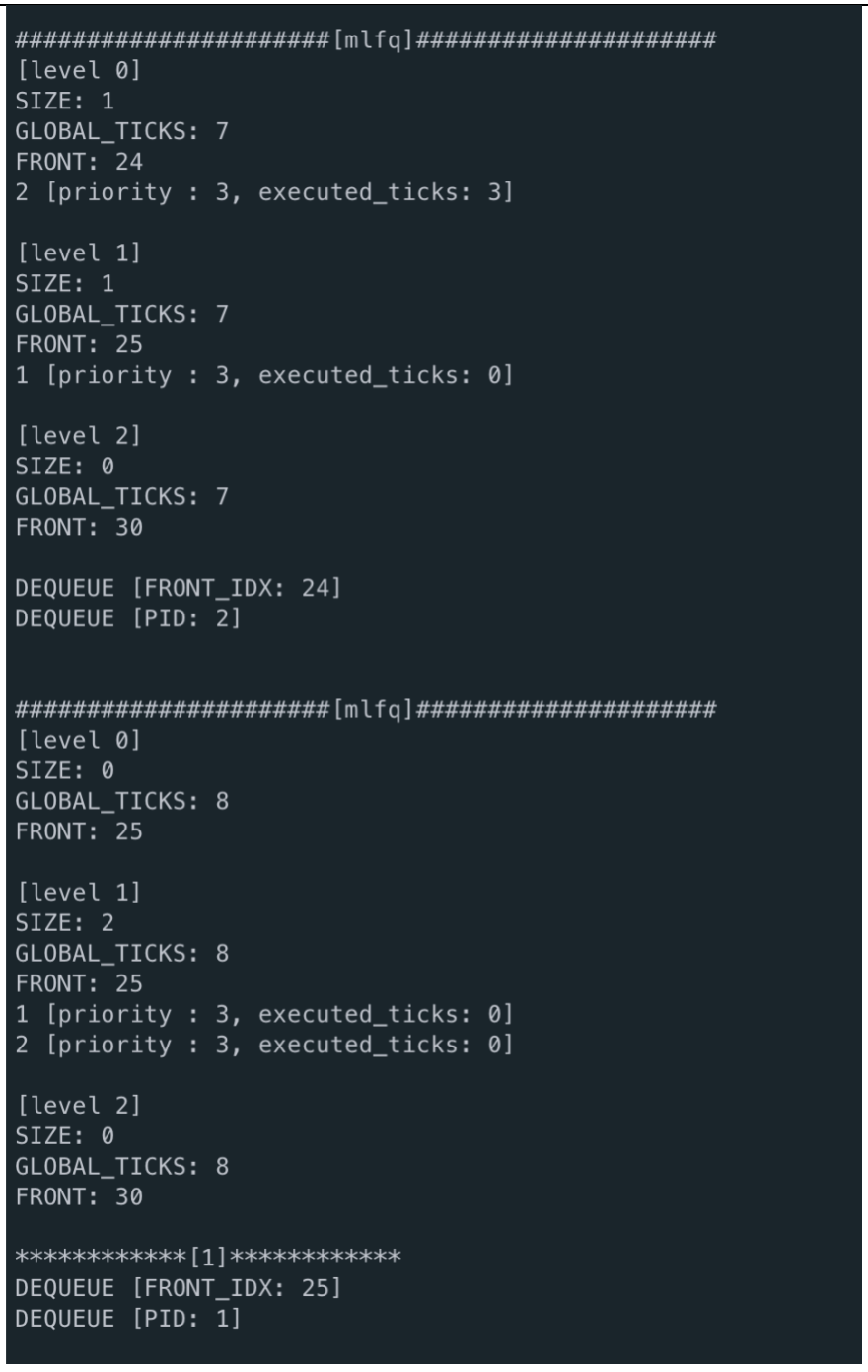
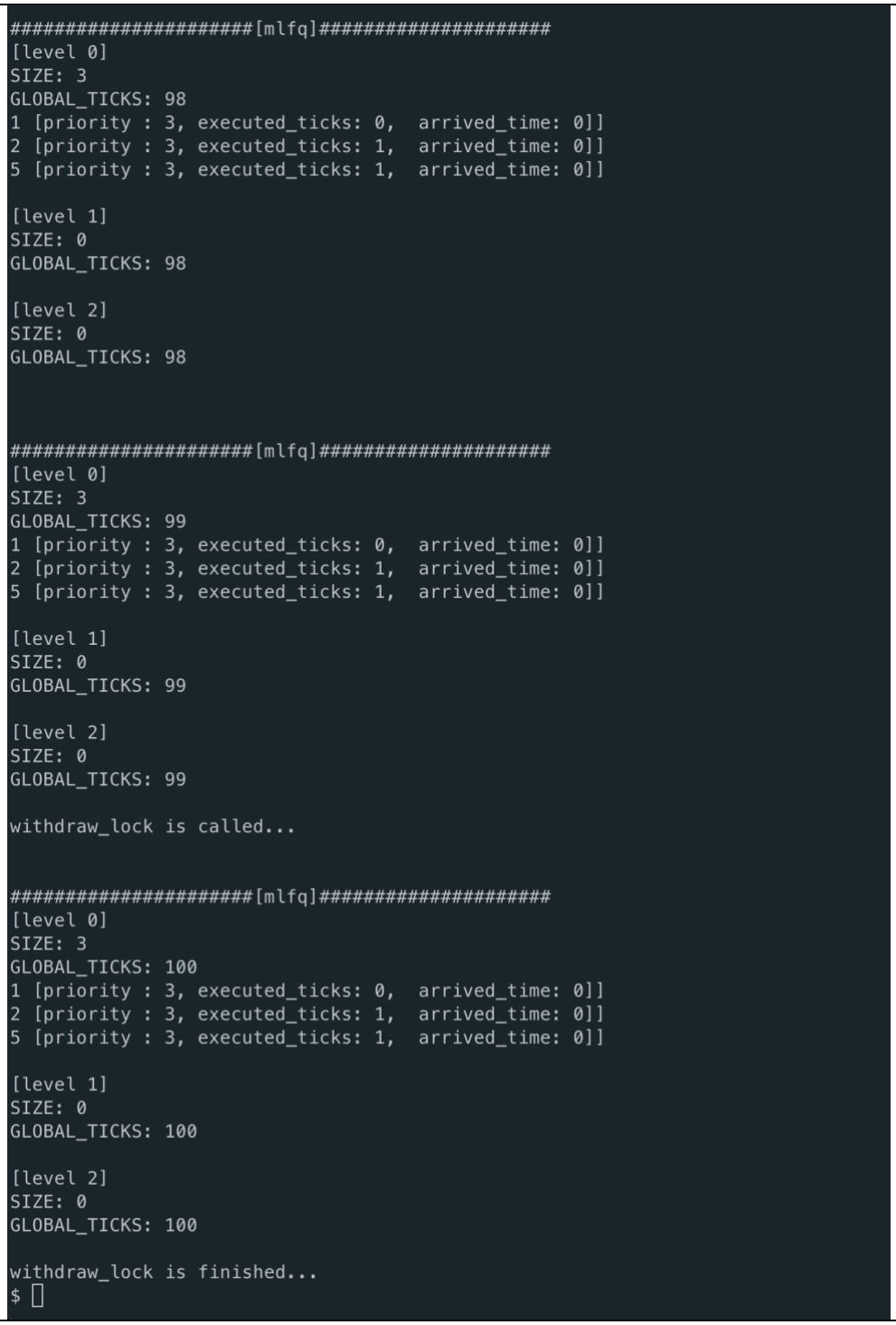


그림 25일정 tick (L0:4, L1: 6, L2: 8)이상이면 다음 큐로 잘 이동

조교님께서 올려주신 mlfq_test 결과
아래와 같이 L0가 가장 작은 값이 나오고 L1이 그 다음 작은 값, L2가 가장 차이가 많이 나게 큰 값이 나오는 것을 확인할 수 있습니다.

```
[Test 1] default
sys_getpid working!
sys_getpid working!
sys_getpid working!
sys_getpid working!
sys_getpid working!
Process 5
L0: 10172
L1: 15459
Process 6
L2: 74369
L3: 0
L4: 0
sys_getpid working!
L0: 9581
L1: 15998
L2: 74421
L3: 0
L4: 0
sys_getpid working!
Process 7
L0: 9998
L1: 20839
L2: 69163
L3: 0
L4: 0
sys_getpid working!
Process 4
L0: 13467
L1: 28043
L2: 58490
L3: 0
L4: 0
sys_getpid working!
[Test 1] finished
done
```

schedulerLockTest: global_ticks가 100 ticks가 되면 자동으로 종료합니다.



schdulerUnlockTest

```
######################[mlfq]#####################
[level 0]
SIZE: 2
GLOBAL_TICKS: 7
13 [priority : 3, executed_ticks: 3,  arrived_time: 0]]
13 [priority : 3, executed_ticks: 3,  arrived_time: 0]]

[level 1]
SIZE: 0
GLOBAL_TICKS: 7

[level 2]
SIZE: 1
GLOBAL_TICKS: 8
2 [priority : 3, executed_ticks: 0,  arrived_time: 4]]

withdraw_lock is finished...
user interrupt 130 called!
$
```

잘못된 학번을 넣었을 때 지금까지 실행된 time quantum, pid, current level을 호출합니다. 이 테스트는 scheduler_lock_test.c와 scheduler_unlock_test.c 파일을 별도로 만들어 진행하였습니다.

```
sb: size 1000 nblocks 941 ninodes 200 nlog 30 logsta
init: starting sh
$ scheduler_lock_test
SchedulerLock is called...Password not matched!
PID: 3 TIME_QUANTUM: 4 CURRENT_LEVEL: 1$
```

그림 26 schedulerlock 잘못 password 입력

```
init: starting sh
$ scheduler_unlock_test
SchedulerUnlock is called...
Password not matched!
PID: 3 TIME_QUANTUM: 3 CURRENT_LEVEL: 1$
```

그림 27 schedulerUnlock password잘못 입력

4. Trouble shooting

(1) 이번 과제에서 저를 가장 힘들고 괴롭게 했던 오류는 trap 13오류였습니다. 이 오류가 proc.c파일에서 난다고 생각하여 처음부터 코드를 뜯어보고 proc.c함구를 다시 구현해보았지만, 원인을 찾을 수 없었습니다. init.c 부터 시작해 프로그램의 맨 처음 시작 부터 계속해서 코드를 리뷰하였습니다. 이 때 저를 가장 슬프게 하였던 점은 오류의 발생 지점이 mlfq가 돌다가 랜덤으로 발생된다는 것이었습니다. 일주일간 이 에러로 하루에 6시간 이상 고생을 하다가 결국에는 xv6를 다시 다운 받고, 순차적으로 코드를 구현하던 중 정말 하나의 단순한 실수로 인해 이 오류가 발생했다는 것을 알게되었습니다.

```
pid 1 initcode: trap 13 err 514 on cpu 0 eip 0x11 addr 0x0--kill proc
lapicid 0: panic: init exiting
 8010488e 801064de 80106294 0 0 0 0 0 0 0
```

그림 28 발생했던 오류

```
void
tvinit(void)
{
  int i;

  for(i = 0; i < 256; i++)
    SETGATE(idt[i], 0, SEG_KCODE<<3, vectors[i], 0);
    //! 오류 원인 ! [T_SYSCALL]
  // SETGATE(idt[T_SYSCALL], 1, SEG_KCODE<<3, vectors[T_SYSCALL], DPL_USER);
  SETGATE(idt[T_INT128], 1, SEG_KCODE<<3, vectors[T_INT128], DPL_USER);
  SETGATE(idt[T_INT129], 1, SEG_KCODE<<3, vectors[T_INT129], DPL_USER);
  SETGATE(idt[T_INT130], 1, SEG_KCODE<<3, vectors[T_INT130], DPL_USER);

  initlock(&tickslock, "time");
}
```

그림 29 [trap.c]

SETGATE를 설정하다가 *SETGATE(idt[T_SYSCALL], 1, SEG_KCODE<<3, vectors[T_SYSCALL], DPL_USER);* 이 라인이 삭제가 되었는데 이를 인지하지 못하고 있었습니다. 이 때문에 제대로 프로그램 상 system call이 제대로 동작하지 못했고 랜덤으로 오류가 발생했던 것이었습니다. 이 부분을 다시 원래대로 추가해주고 나서는 trap 13 에러가 사라졌습니다.

(2) 두 번째로 저를 괴롭게 하던 오류는 zomebie exit이었습니다. 이것도 간헐적으로 나는 오류여서 고치기 매우 힘들었습니다.

```
sys_getpid working!
lapicid 0: panic: zombie exit
 80104998 801061bb 80105599 801066f5 80106384 0 0 0 0 0
```

```
static void
wakeup1(void *chan)
{
  struct proc *p;

  for(p = ptable.proc; p < &ptable.proc[NPROC]; p++)
    if(p->state == SLEEPING && p->chan == chan) {
      p->state = RUNNABLE;
    }
}
```

그림 30 [proc.c] wakeup1

Priority boosting을 테스트 하면서 이 부분을 p->state == SLEEPING && p->chan == chan 이 아닌 p->state == SLEEPING으로 수정했었는데 이로 인해 process가 exit되고 나서 스케줄링 되어 맨 마지막의 panic zombie 가 나타났었던 것이었습니다. 이는 원래대로 수정하여 해결하였습니다.

(3) $가 뜨며 종료되지 않는 오류

잘 구현했다고 생각했고, cprintf를 통해 mlfq를 출력했을 때도 나와야하는 결과 값과 동일하게 잘 출력 되었습니다. 그러나 MakeFile을 실행시켰을 때 $로 입력할 수 있게 나오지 않고 그 전 상태로 대기하는 오류가 등장했습니다.

```
[L2], Successfully allocated at index 61
[L2], Successfully allocated at index 62
[L2], Successfully allocated at index 63
process runnable 1
cpu0: starting 0
sb: size 1000 nblocks 941 ninodes 200 nlog 30 logstart 2 inodestart 32 bmap start 58
init: starting sh
```

이는 proc이 RUNNABLE이 아닐 때는 앞에서 빼고 뒤로 다시 queue에 넣어주는 코드를 구현하여 문제를 해결할 수 있었습니다.

```
    ret = queue->data[queue->front];
    if(ret->state == RUNNABLE) {
      goto found;
    } else {
      mlfq_dequeue(lev, 0);
      mlfq_enqueue(lev, ret);
    }
  }
```

그림 31 [proc.c] mlfq_select()

구현하는데 있어 많은 어려움을 겪었고 조교님들 답변 덕분에 구현할 수 있었지만, 에러를 고칠 때마다 뿌듯하기도 하고 kernel모드와 user모드가 이렇게 소통하는구나 깨달을 수 있었습니다. 다음 과제도 열심히 해보아야 겠다는 생각이 들었습니다. 감사합니다.