

# Project02 WIKI

이름: 문준영

학번: 2022006135

학과: 컴퓨터소프트웨어학부

## Design:

'먼저, 명세서를 읽고 scheduling을 구현하라는 것을 알았습니다. 이를 구현하기 위해 어떤 파일에 무엇을 구현해야 하는지와 그 파일의 함수들이 어떠한 역할을 하는지를 살펴보았습니다.

Proc.c에 가서 보니 여러가지 process 관련 함수와 structure들이 선언되어 있었습니다. 또한, proc.h에 가보니 proc 구조체가 선언되어 있었습니다. 마지막으로 가장 중요하게 trap.c 파일에서 interrupt를 처리한다는 것을 알게 되었습니다. 이를 통해 다음과 같이 저는 본 과제에 대한 설계를 하였습니다.

## 설계:

먼저 proc.h에 L0~L2를 위한 queue 구조체를 선언합니다. 그 구조체 안에는 struct proc \*arr[NPROC]인 process 포인터 배열을 선언합니다. 또한, front와 rear를 사용하여 순환 queue를 구현합니다.

Proc.c에 L0~L2 인스턴스를 만들고 이를 이한 함수들을 만듭니다. 함수로는 Enqueue(), Dequeue, init\_queue()등이 있을 것입니다.

L3는 priority queue이므로 max heap을 구현합니다. Proc 구조체에 priority 변수를 추가하여 L3에서 이를 스케줄링 할 수 있도록 합니다. Proc.c에 L3 인스턴스를 만듭니다. 또한 max heap과 관련된 함수들을 만듭니다. 함수로는 init\_heap(), percolateUp(), insert(), percolateDown(), DeleteMax()가 있을 것입니다.

Proc 구조체에 몇가지 field를 추가합니다. Field로는 int queue\_level, int ticks\_used, int time\_quantum, int priority가 있습니다. 이들은 각각 Queue의 level과 사용한 시간, time quantum, 그리고 L3에서 활용되는 priority입니다.

Trap.c에 있는 trap 함수에 가서 timer interrupt 처리를 해줍니다. P->ticks\_used >= p-

>time\_quantum 일 때 스케줄링이 일어나도록 할 것입니다.

Trap.c에서 타이머 인터럽트에 대한 처리를 해주었으므로, proc.c에 scheduler함수에서는 L0에 있는 프로세스를 먼저 수행하게 할 것이고, L0에 element가 없으면 L1 프로세스를 실행시키고, L1에 프로세스가 없으면 L2에 있는 프로세스, L2에도 프로세스가 없으면 L3의 프로세스를 실행시키는 로직을 구현할 것입니다.

Priority boosting을 위해 trap.c에 trap함수에 가서 ticks가 100의 배수일 때마다 p\_boosting()함수를 호출하게 할 것입니다. P\_boosting()은 제가 proc.c에 구현할 것이고 이는 L1, L2,, L3에 있는 모든 원소를 빼고 각 process의 field를 L0에 적합하게 바꾼후 L0에 Enqueue하는 식으로 구현 할 것입니다.

각 process가 자신의 time\_quantum을 모두 사용하였을 때에 이 process를 queue\_level을 내려주며 time\_quantum을 내려준 queue에 적합하게 만듭니다. 이를 하나의 함수인 Decrease\_queue()함수로 구현할 계획입니다.

Getlev() system call 같은 경우에는 단순히 p->queue\_level을 리턴하는 로직을 구현할 것입니다.

Setpriority() system call의 경우에는 process의 priority를 인자로 들어온 priority로 바꾸어 주는 로직을 구현할 것입니다.

이제 MoQ에 대한 설계를 설명하겠습니다.

먼저 MLFQ 구현 과정에서 만든 구조체인 queue 구조체를 똑같이 활용합니다. 이는 따로 priority boosting이나 decrease\_queue 의 대상이 아니기에 setmonopoly(), monopolize(), unmonopolize()를 구현해줄 것입니다.

먼저, setmonopoly() 함수의 경우 ptable을 모두 돌며 pid가 일치하는 process를 찾습니다 이를 MoQ에 넣어줍니다.

다음으로 monopolize()에서는 현재 진행되고 있는 프로세스를 원래 그 프로세스가 속해 있던 queue에 되돌려 넣어줍니다. 그 후 swtch 함수를 이용해 context switch를 일어나게 할 것입니다.

마지막으로 unmonopolize()에서는 현재 진행되고 있는 MoQ process를 다시 MoQ에 넣어줍니다.

제가 생각해낸 구현 방법 중 가장 중요한 포인트는 바로 “모든 RUNNABLE한 프로세스는 queue나 heap 구조체 안에 존재한다.” 입니다. 이 말인 즉슨, RUNNING 중이거나

SLEEPING중인 여느 다른 process들은 제가 신경을 쓸 필요가 없다는 뜻입니다.

이제 저의 실제 구현 과정을 보도록 하겠습니다.

### Implementation:

Proc.h에 정의한 것들을 살펴보겠습니다.

```
char name[10];
int queue_level;
int ticks_used;
int time_quantum;
int priority;
int in_moq;
```

proc 구조체에 현재 큐 레벨, 사용한 시간, 총 사용 가능한 시간, 우선순위, MoQ에 있는지 여부를 추가하였습니다

Proc.c에 선언한 함수들을 살펴보도록 하겠습니다.

먼저 저는 queue 구조체를 생성하였습니다. Queue 구조체에는 front와 rear, 그리고 이의 최대 크기인 capacity와 이의 크기인 size가 존재합니다. 이와 관련된 함수로 Enqueue,

```
struct queue{
    int capacity;
    int front;
    int rear;
    int size;
    int time_quantum;
    struct proc *arr[NPROC];
};
```

Dequeue, 그리고 queue\_init을 구현 해줬습니다. L0~L2 인스턴스를 proc.c에 생성하고 이를

```
static struct proc *initproc;
static struct queue L0;
static struct queue L1;
static struct queue L2;
static struct queue L3[11];
static struct queue MoQ;
```

```
0 void
1 Enqueue(struct queue *Q, struct proc *p){
2     //if queue is full
3     if(Q->size == Q->capacity){
4         return;
5     }
6     Q->size++;
7     Q->rear = (Q->rear + 1) % (Q->capacity);
8     Q->arr[Q->rear] = p;
9 }
10 struct proc*
11 Dequeue(struct queue *Q){
12     if(Q->size == 0){
13         return NULL;
14     }
15     struct proc *rm_p = Q->arr[Q->front];
16     Q->arr[Q->front] = NULL;
17     Q->front = (Q->front + 1) % (Q->capacity);
18     Q->size--;
19     return rm_p;
20 }
```

```

void
queue_init(struct queue *Q, int time_quantum){
    //Setting lock
    Q->size = 0;
    Q->front = 0;
    Q->rear = -1;
    Q->time_quantum = time_quantum;
    Q->capacity = NPROC;
    for(int i = 0; i < NPROC; i++){
        Q->arr[i] = NULL;
    }
}

```

중요한 점은 queue\_init에서 queue의 rear를 -1부터 시작하게 했다는 것입니다. 이는 순환 큐의 마지막 index를 rear가 가리키도록 하기 위함입니다.

L3를 위해서 queue를 11개의 배열로 만들었습니다. 각각의 queue의 index는 queue의 priority를 의미합니다

```

int find_queue_idx(struct queue *Q, int pid){
    if(Q->size == 0){
        return -1;
    }

    int idx = Q->front;
    int count = 0;

    while (count < Q->size) {
        if (Q->arr[idx]->pid == pid) {
            return idx;
        }
        idx = (idx + 1) % Q->capacity;
        count++;
    }

    return -1;
}

```

```
int removeAt(struct queue *Q, int pid){
    int idx = find_queue_idx(Q, pid);
    if(idx == -1){
        return -1;
    }

    if(Q->size == 1) {
        Q->arr[idx] = NULL;
        Q->front = 0;
        Q->rear = -1;
        Q->size--;
        return 0;
    }

    while(idx != Q->rear){
        int nextidx = (idx + 1) % Q->capacity;
        Q->arr[idx] = Q->arr[nextidx];
        idx = nextidx;
    }
    Q->arr[Q->rear] = NULL;
    Q->rear = (Q->rear - 1 + Q->capacity) % Q->capacity;

    Q->size--;

    if(idx == Q->front) {
        Q->front = (Q->front + 1) % Q->capacity;
    }

    return 0;
}
```

위의 두 함수는 queue에서 중간 원소를 제거할 일이 있을 때 사용하는 함수입니다. 이는 후에 MoQ를 다룰 때 사용합니다.

```

void
Decrease_queue(struct proc *p){
    //already Dequeued when scheduling
    if(p->queue_level == 0){
        if(p->pid % 2 == 1){
            //move process to L1
            p->queue_level = 1;
            p->ticks_used = 0;
            p->time_quantum = L1.time_quantum;
            Enqueue(&L1, p);
        }
        else{
            //move process to L2
            p->queue_level = 2;
            p->ticks_used = 0;
            p->time_quantum = L2.time_quantum;
            Enqueue(&L2,p);
        }
    }
    else if(p->queue_level == 1 || p->queue_level == 2){
        //move process to L3
        p->queue_level = 3;
        p->ticks_used = 0;
        p->time_quantum = L3[p->priorty].time_quantum;
        Enqueue(&L3[p->priorty], p);
    }
    else if(p->queue_level == 3){
        if(p->priorty > 0){
            p->priorty--;
        }
        p->ticks_used = 0;
        p->time_quantum = L3[p->priorty].time_quantum;
        Enqueue(&L3[p->priorty], p);
    }
}

```

이 함수는 현재 진행중인 process가 주어진 시간을 다 사용하였을 때 queue의 level을 낮춰주는 함수입니다.

```
void
p_boosting(){
    if(is_moq){
        return;
    }
    struct proc *p;
    acquire(&ptable.lock);
    //Empty L1 and put all processes into L0
    while(L1.size > 0){
        p = Dequeue(&L1);
        p->queue_level = 0;
        p->ticks_used = 0;
        p->time_quantum = L0.time_quantum;
        Enqueue(&L0, p);
    }
    //Empty L2 and put all processes into L0
    while(L2.size > 0){
        p = Dequeue(&L2);
        p->queue_level = 0;
        p->ticks_used = 0;
        p->time_quantum = L0.time_quantum;
        Enqueue(&L0, p);
    }
    //Empty L3 and put all processes into L0
    for(int i = 10; i >= 0; i--){
        while(L3[i].size > 0){
            p = Dequeue(&L3[i]);
            p->queue_level = 0;
            p->ticks_used = 0;
            p->time_quantum = L0.time_quantum;
            Enqueue(&L0, p);
        }
    }
    release(&ptable.lock);
}
```

```
void
incr_ticks(struct proc *p){
    acquire(&ptable.lock);
    p->ticks_used++;
    release(&ptable.lock);
}
```

이 함수는 global tick이 100이 될 때마다 L0 큐로 프로세스를 재조정하는 함수입니다. 코드에서 볼 수 있듯이 is\_moq가 1이면 실행되지 않습니다. 즉, Monopolize중일 때 priority boosting이 일어나지

이 함수는 trap.c에서 global tick이 증가할 때 현재 진행중인 프로세스의 tick사용량도 증가시켜주는 함수입니다. Ptable lock을 획득하기 위해 proc.c에 선언하였습니다.

```
void
pinit(void)
{
    initlock(&ptable.lock, "ptable");
    queue_init(&L0, 2);
    queue_init(&L1, 4);
    queue_init(&L2, 6);
    for(int i = 10; i >= 0 ; i--){
        queue_init(&L3[i], 8);
    }
    queue_init(&MoQ, 99);

}
```

pinit함수에서 모든 큐를 초기화해 주었습니다.

```
acquire(&ptable.lock);
//initialize the ticks_used
p->ticks_used = 0;
//initialize time_quantum of L0
p->time_quantum = L0.time_quantum;
//set queue_level as 0 representing L0
p->queue_level = 0;
//initialize priority to 0 for later L3
p->priorty = 0;
//process is not in MoQ
p->in_moq = 0;

release(&ptable.lock);
```

allocproc에서 처음 프로세스가 할당 될 때 프로세스에 관한 정보들을 초기화시켜줍니다.

```

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

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

        // Loop over process table looking for process to run.
        acquire(&ptable.lock);
        //If L0 is not empty run L0 process
        if(!is_mq){
            if(L0.size > 0){
                p = Dequeue(&L0);
            }
            //If L1 is empty then run L1 process
            else if(L1.size > 0){
                p = Dequeue(&L1);
            }
            //If L2 is empty then run L2 process
            else if(L2.size > 0){
                p = Dequeue(&L2);
            }
            //If L3 is empty then run L3 process
            else{
                for(int i = 10; i >= 0; i--){
                    if(L3[i].size > 0){
                        p = Dequeue(&L3[i]);
                        break;
                    }
                }
            }
        }
        if(is_mq){
            if(MoQ.size > 0){
                p = Dequeue(&MoQ);
            }
            else{
                unmonopolize();
                release(&ptable.lock);
                continue;
            }
        }

        if(p == NULL || p->state == ZOMBIE){
            release(&ptable.lock);
            continue;
        }
    }
}

```

스케줄러 함수에서 어치파 저의 목표는 큐 안에는 RUNNABLE한 프로세스만 들어있으니 순차적으로 각 큐의 현재 크기를 확인하고 Dequeue를 하여 level 별로 큐에 있는 프로세스를 실행하였습니다

```

5 // Give up the CPU for one scheduling round.
6 void
7 yield(void){
8     acquire(&ptable.lock); //DOC: yieldlock
9     if(is_moq && myproc() && myproc()->in_moq){
10         release(&ptable.lock);
11         return;
12     }
13     if(!is_moq && myproc() && myproc()->in_moq){
14         Enqueue(&MoQ, myproc());
15     }
16     struct proc *p = myproc();
17     Decrease_queue(p);
18     myproc()->state = RUNNABLE;
19     sched();
20     release(&ptable.lock);
21 }

```

습니다.

.yield함수에 moq인지 검사를 하여 현재 RUNNING중인 프로세스가 setmonopoly 함수에 의해 in\_moq 가 추가되었으면 MoQ에 넣어주고 CPU를 양도하도록 했습니다.

또한 monopolize 상태일 때 yield가 일어나지 않도록 설정했

```

int
getlev(void){
    struct proc *p = myproc();
    if(p->in_moq){
        return 99;
    }
    return p->queue_level;
}

```

getlev 함수는 프로세스가 현재 속해있는 큐의 레벨을 리턴하는 함수입니다. 이 함수는 현재 자신의 queue\_level을 리턴하고 MoQ에 속해있으면 99를 리턴합니다

```

int
setpriority(int pid, int priority){
    struct proc *p;
    acquire(&ptable.lock);

    //if priority is not in range 0 <= priority <= 10 return -2
    if(priority < 0 || priority > 10){
        release(&ptable.lock);
        return -2;
    }
    int flag = 0;
    for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
        if(p->pid == pid){
            flag = 1;
            break;
        }
    }
    //Case that pid does not exist
    if(flag == 0){
        release(&ptable.lock);
        return -1;
    }
    if(p->state == RUNNABLE || p->queue_level == 3){
        removeAt(&L3[p->priority], p->pid);
        Enqueue(&L3[priority], p);
    }
    p->priority = priority;
    release(&ptable.lock);
    return 0;
}

```

setpriority함수입니다. 이는 모든 .프로세스를 ptable에서 돌며 pid가 일치하는 프로세스를 찾고 이의 priority를 수정해 줍니다. 만약 이 프로세스가 L3큐에 속해있다면 이를 queue에서 재조정해 줍니다.

```

int
setmonopoly(int pid, int password){
    if(password != 2022006135){
        return -2;
    }
    struct proc *p;
    acquire(&ptable.lock);
    for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
        if(p->pid == pid){
            if(p->state == RUNNABLE){
                p->in_moq = 1;
                Enqueue(&MoQ, p);
            }
            else if(p->state == RUNNING || p->state == SLEEPING){
                p->in_moq = 1;
            }
        }
        release(&ptable.lock);
        return MoQ.size;
    }

    release(&ptable.lock);
    return -1;
}

```

Setmonopoly 함수입니다. 먼저 현재 runnable한 프로세스라면 MLFQ에 존재하는 것이므로 일단 MoQ에 넣어줍니다. 만약 프로세스가 현재 RUNNING되고 있는 상태나 SLEEPING상태에 있다면 in\_moq를 1로 바꾸어 줍니다. 이를 본래 queue에서 제거하는 행동은 exit함수에서 합니다.

```
int
sys_monopolize(){
    monopolize();
    return 0;
}

void
unmonopolize(){
    struct proc *p;
    while(MoQ.size > 0){
        p = Dequeue(&MoQ);
        if(p->state == RUNNABLE){
            if(p->queue_level == 0){
                Enqueue(&L0,p);
            }
            else if(p->queue_level == 1){
                Enqueue(&L1,p);
            }
            else if(p->queue_level == 2){
                Enqueue(&L2,p);
            }
            else if(p->queue_level == 3){
                Enqueue(&L3[p->priorty],p);
            }
        }
    }
    for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
        if(p->in_moq == 1){
            p->in_moq = 0;
        }
    }
    is_moq = 0;
}
```

Monopolize는 전역변수인 is\_moq를 1로 만들어주고 현재 monopolize 상태라는 것을 암시합니다. Unmonopolize의 경우 MoQ에 남아있는 프로세스를 모두 제거하고 is\_moq 전

역변수 값을 0으로 바꾸어 monopolize가 끝남을 암시합니다.

```
1 // ... the ptable lock must be held.
2 static void
3 wakeup1(void *chan)
4 {
5     struct proc *p;
6
7     for(p = ptable.proc; p < &ptable.proc[NPROC]; p++)
8         if(p->state == SLEEPING && p->chan == chan){
9             p->state = RUNNABLE;
10            if(p->in_moq){
11                Enqueue(&MoQ, p);
12            }
13            if(is_moq){
14                continue;
15            }
16            if(p->queue_level == 0){
17                Enqueue(&L0, p);
18            }
19            else if(p->queue_level == 1){
20                Enqueue(&L1, p);
21            }
22            else if(p->queue_level == 2){
23                Enqueue(&L2, p);
24            }
25            else if(p->queue_level == 3){
26                Enqueue(&L3[p->priority], p);
27            }
28        }
29    }
30 }
```

Sleep에서 깨어날 때 본래 큐로 프로세스를 다시 올려주기 위해 p->queue\_level을 참고하여 프로세스를 본래 큐로 Enqueue해줍니다.

```

if(curproc->in_moq){
    if(curproc->queue_level == 0){
        removeAt(&L0, curproc->pid);
    }
    else if(curproc->queue_level == 1){
        removeAt(&L1, curproc->pid);
    }
    else if(curproc->queue_level == 2){
        removeAt(&L2, curproc->pid);
    }
    else if(curproc->queue_level == 3){
        removeAt(&L3[curproc->priority], curproc->pid);
    }
}

```

Exit 함수에서 프로세스가 in\_moq인데 종료할 시 원래 있는 큐에서 제거해줍니다. Monopolize 상태에서 종료된 프로세스는 zombie상태로 본래 MLFQ에 남아 있을 것이기 때문입니다.

### Trap.c에서 수정한 것들

```

case T_IRQ0 + IRQ_TIMER:
    if(cpuid() == 0){
        struct proc *p = myproc();
        acquire(&tickslock);
        ticks++;
        global_tick++;

        if(p && p->state == RUNNING){
            incr_ticks(p);
        }
        if(global_tick % 100 == 0){
            p_boosting();
        }
        wakeup(&ticks);
        release(&tickslock);
    }
    lapiceoi();
    break;

    struct proc *p = myproc();

    if(p->ticks_used >= p->time_quantum){
        yield();
    }
}

```

timer interrupt가 올 때마다 incr\_ticks를 넣어주어 현재 프로세스의 ticks\_used를 증가시킵니다.

Global tick이 100의 배수가 될 때마다 priority boosting을 수행합니다.

프로세스가 자신이 가진 총 시간보다 더 많은 시간을 사용하면 yield를 호출하여 CPU를 양도합니다.

**Result:**

```
HLRQ test start
[Test 1] default
Process 5
L0: 15696
L1: 31702
L2: 0
L3: 52602
MoQ: 0
Process 7
L0: 15645
L1: 31844
L2: 0
L3: 52511
MoQ: 0
Process 9
L0: 16778
L1: 32593
L2: 0
L3: 50629
MoQ: 0
Process 11
L0: 15834
L1: 29688
L2: 0
L3: 54478
MoQ: 0
Process 4
L0: 15629
L1: 0
L2: 40239
L3: 441
Process 6
L0: 13443
L1: 0
L2: 33788
L3: 52769
MoQ: 0
Process 10
L0: 13442
L1: 0
L2: 33811
L3: 52747
MoQ: 0
32
MoQ: 0
ss 8
L0: 15700
L1: 0
L2: 40089
L3: 44211
```

Test1 결과입니다. 모든 프로세스가 정상적으로 큐에서 실행되고 종료되었으며 예상대로 홀수 pid를 가진 것들이 먼저 끝났습니다.

```
[Test 1] finished
[Test 2] priorities
Process 18
L0: 11329
L1: 0
L2: 34103
L3: 54568
MoQ: 0
Process 19
LProcess 16
L0: 13660
L1: 0
L2: 40874
L3: 45466
MoQ: 0
0: 9087
L1: 18126
L2: 0
L3: 72787
MoQ: 0
Process 17
L0: 11385
L1: 22836
L2: 0
L3: 65779
MoQ: 0
Process 15
L0: 12434
L1: 27201
L2: 0
L3: 60365
MoQ: 0
Process 13
L0: 13579
L1: 26388
L2: 0
L3: 60033
MoQ: 0
Process 12
L0: 18199
L1: 0
L2: 54501
L3: 27300
MoQ: 0
Process 14
L0: 10945
L1: 0
```

Test 2의 결과입니다. Pid가 큰 것이 대체로 더 높은 priority를 가지기에 대체로 pid가 큰 것이 먼저 실행되었습니다.

```
MoQ: 0
[Test 2] finished
[Test 3] sleep
Process 20
L0: 500
L1: 0
L2: 0
L3: 0
MoQ: 0
Process 21
L0: 500
L1: 0
L2: 0
L3: 0
MoQ: 0
Process 22
L0: 500
L1: 0
L2: 0
L3: 0
MoQ: 0
Process 23
L0: 500
L1: 0
L2: 0
L3: 0
MoQ: 0
Process 24
L0: 500
L1: 0
L2: 0
L3: 0
MoQ: 0
Process 25
L0: 500
L1: 0
L2: 0
L3: 0
MoQ: 0
Process 26
L0: 500
L1: 0
L2: 0
L3: 0
MoQ: 0
Process 27
L0: 500
L1: 0
L2: 0
```

Test3의 수행 결과입니다. Sleep이 정상적으로 수행된 것이 잘 보이고 아무 문제 없이 정상적으로 수행되었습니다. 이는 제가 wakeup1 함수에서 프로세스가 다시 wakeup할 때 큐에 추가하도록 구현해줬기에 결과가 정상적으로 나왔습니다. 뿐만 아니라, 저의 큐 안에는 오직 RUNNABLE한 프로세스만 있기에 sleep을 수행하는데 별로 어려움을 겪지 않았습니다.

```
[Test 3] finished
[Test 4] MoQ
Number of processes in MoQ: 1
Number of processes in MoQ: 2
Number of processes in MoQ: 3
Number of processes in MoQ: 4
Process 29
L0: 0
L1: 0
L2: 0
L3: 0
MoQ: 100000
Process 31
L0: 0
L1: 0
L2: 0
L3: 0
MoQ: 100000
Process 33
L0: 0
L1: 0
L2: 0
L3: 0
MoQ: 100000
Process 35
L0: 0
L1: 0
L2: 0
L3: 0
MoQ: 100000
Process 32
Process Process 30L0: 4480
L1: 0
L2: 13783
L3: 81737
MoQ: 0

L0: 8323
L1: 0
L2: 20110
L3: 71567
MoQ: 0
34
L0: 9083
L1: 0
L2: 27424
L3: 63493
MoQ: 0
```

Test 4의 수행 결과입니다. MoQ 수행이 잘 되지만 안타깝게도 Test4 finished 문구가 뜨지 않습니다. 이는 디버깅 해본 결과 스케줄러가 무한 루프를 도는 것으로 판단했습니다. 이를 미쳐 수정하지 못하고 과제를 제출하게 되어 마음이 너무 아픕니다.

### **Trouble Shooting:**

어느 때는 실행이 정상적으로 되고 어느 때는 정상적으로 실행 안되는 때가 있었습니다. 저는 이게 제가 어떠한 코드를 건드려서 인 줄 알았지만 디버깅 해본 결과 lock을 온전하게 걸어주지 않아서의 문제였습니다. 또한, 처음에 heap을 구현하여 L3를 구형하였지만 그 결과가 짹수 pid 프로세스가 먼저 수행되는 결과를 가져왔습니다. 이는 저희가 원하는 결과가 아니기에 불가피하게 본래의 heap을 버리고 queue 11개로 대체했습니다.

Lock 수정에 이 과제에 대부분의 시간을 쏟았고 이를 해결하여 기뻤지만 마지막 MoQ에서 Test4 finished가 뜨지 않았습니다. 마감날이기에 어쩔 수 없이 본 코드를 제출하게 되었습니다. 좋게 봐주시면 감사하겠습니다.

본 과제를 통해 xv6에 대해 정말 자세하게 알게 되었고 비록 과제는 제출 하였지만 마지막 MoQ를 꼭 스스로 수정하여 완벽한 스케줄링을 해낼 것입니다.