

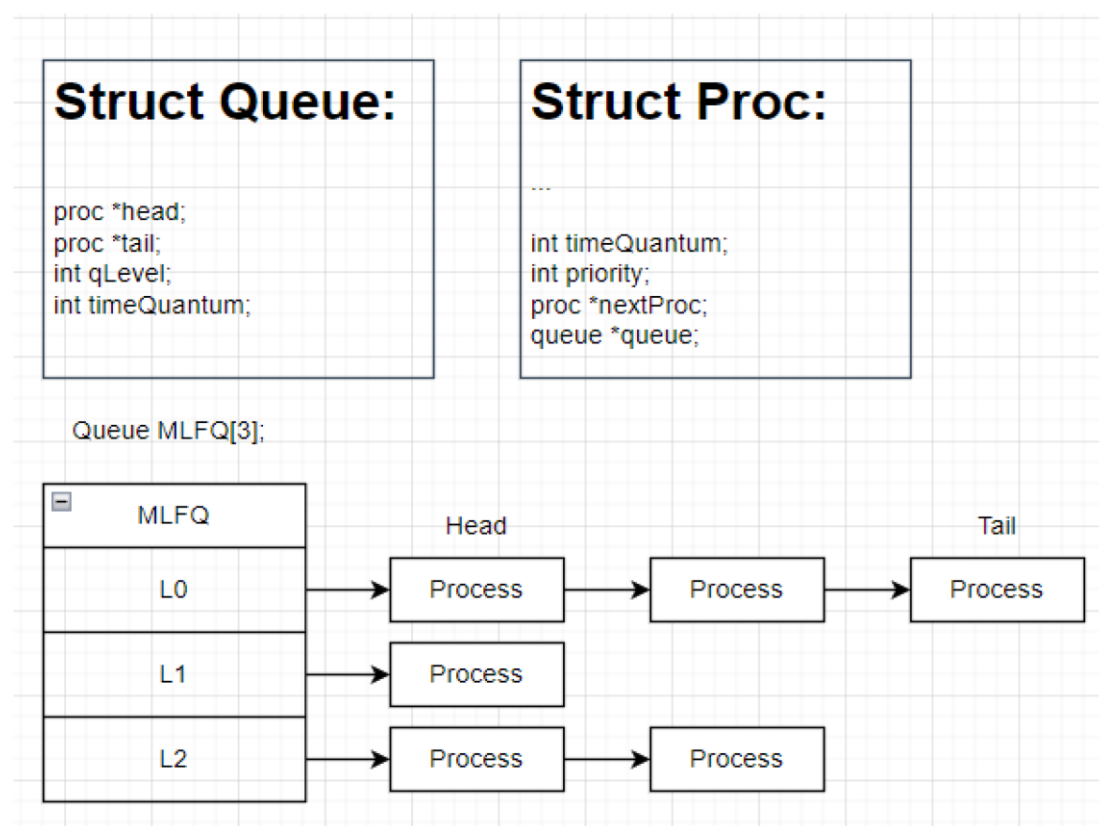
ELE3021 Project #1 Wiki

2020059152 오주영

1. 초기 디자인

최종 구현에 채택된 디자인을 설명드리기 앞서, 구현을 하기 전에 구상했던 디자인을 먼저 설명드리고자 합니다. 따라서 이 디자인들 중 몇몇은 최종 디자인과 다르다는 점 유의 부탁드립니다. 최종 디자인은 다음 섹션에서 설명드리도록 하겠습니다. 또한, 보여드릴 도안에서 나타나는 코드들은 의사코드라는 점 양해 바랍니다.

자료구조 및 구조체



가변길이의 Queue 는, Process 로 이루어진 Linked List 로 구현하는 것이 적합하다고 판단했습니다. 이를 위해서 Process 구조체는, 자신이 현재 속해 있는 Queue 에서 자신 다음 Process 를 가리키는 포인터 변수를 가집니다. Queue 구조체는 가장 앞, 가장 뒤에 있는 Process 를 가리키는 포인터를 각각 가집니다. 또한, Queue 란 Process 모두 timeQuantum 변수를 가집니다. Process 의 timeQuantum 은, 하위 Queue 로 이동하거나 Priority 를 감소시키기 전까지 남은 시간(Demoting 되기 전까지 남은 시간)을 의미하며, Queue 의 timeQuantum 은 어떤 Process 가 해당 Queue 에 처음 들어왔을때 가져야 할 초기 timeQuantum 을 의미합니다. 즉 Process 가 특정 Queue 로 들어갔을때 자신의 timeQuantum 을 이 값으로 초기화시켜줍니다. Process 구조체는 자신이 속해있는 Queue 를 가리키는 포인터를 가집니다. 이 정보는, 이 Process 가 자신의 timeQuantum 을 모두 소진했을 때 다음 어떤 Queue 로 이동해야하는지를 알 수 있기 위함입니다. 마지막으로 Scheduling 을 할 때는, 기존 ptable.proc 배열을 사용하는 것이 아닌, Queue 구조체 3 개를 담은 배열을 사용합니다.

Scheduler 함수

Scheduler():

```
proc p;  
int qLevel = 0;  
  
while (true)  
{  
    while (MLFQ[qLevel].isEmpty == false)  
    {  
        p = MLFQ[qLevel].pop();  
  
        context switch to p;  
  
        qLevel = 0;  
    }  
  
    if (++qLevel == 3) qLevel = 0;  
}
```

Scheduler 함수는 무한 반복문을 돌며, 각 Queue 에 기다리는 Process 가 있는지 확인합니다. 발견될 경우, Dequeue 한뒤 Context Switching 을 통해 이 Process 가 CPU 를 사용하도록 해줍니다. 이렇게 Scheduling 을 성공적으로 할때마다 다시 L0 Queue 로 돌아가서 이 과정을 반복합니다. 예를 들어, L1 에 있는 Process 로 Context Switch 를 한뒤 L2 에서 다음 Process 를 찾는 것이 아니라, 그 동안 L0 에 새로 들어온 Process 가 있는지 다시 확인을 합니다.

Yield 함수

Yield():

```
proc curProc = myproc();

if (--curProc.timeQuantum == 0)
{
    if (curProc.queue == L2)
    {
        if (curProc.priority != 0)
            curProc.priority--;

        Enqueue(curProc, MLFQ[2]);
    }
    else
    {
        Enqueue(curProc, MLFQ[curProc.queue.level + 1]);
    }
    sched();
}
```

매 Timer Interrupt, 즉 매 tick 실행될 Yield 함수는 이렇게 구상해보았습니다. 매 tick 마다 다음 Process 에게 CPU 를 양보하는 기존 xv6 의 디자인과는 달리, 주어진 time quantum 을 모두 소진한 경우에만 sched() 함수를 호출합니다. 뿐만 아니라 time quantum 을 모두 소진하면 Demoting 해주는 작업도 Yield 함수에서 수행합니다.

Enqueue 함수

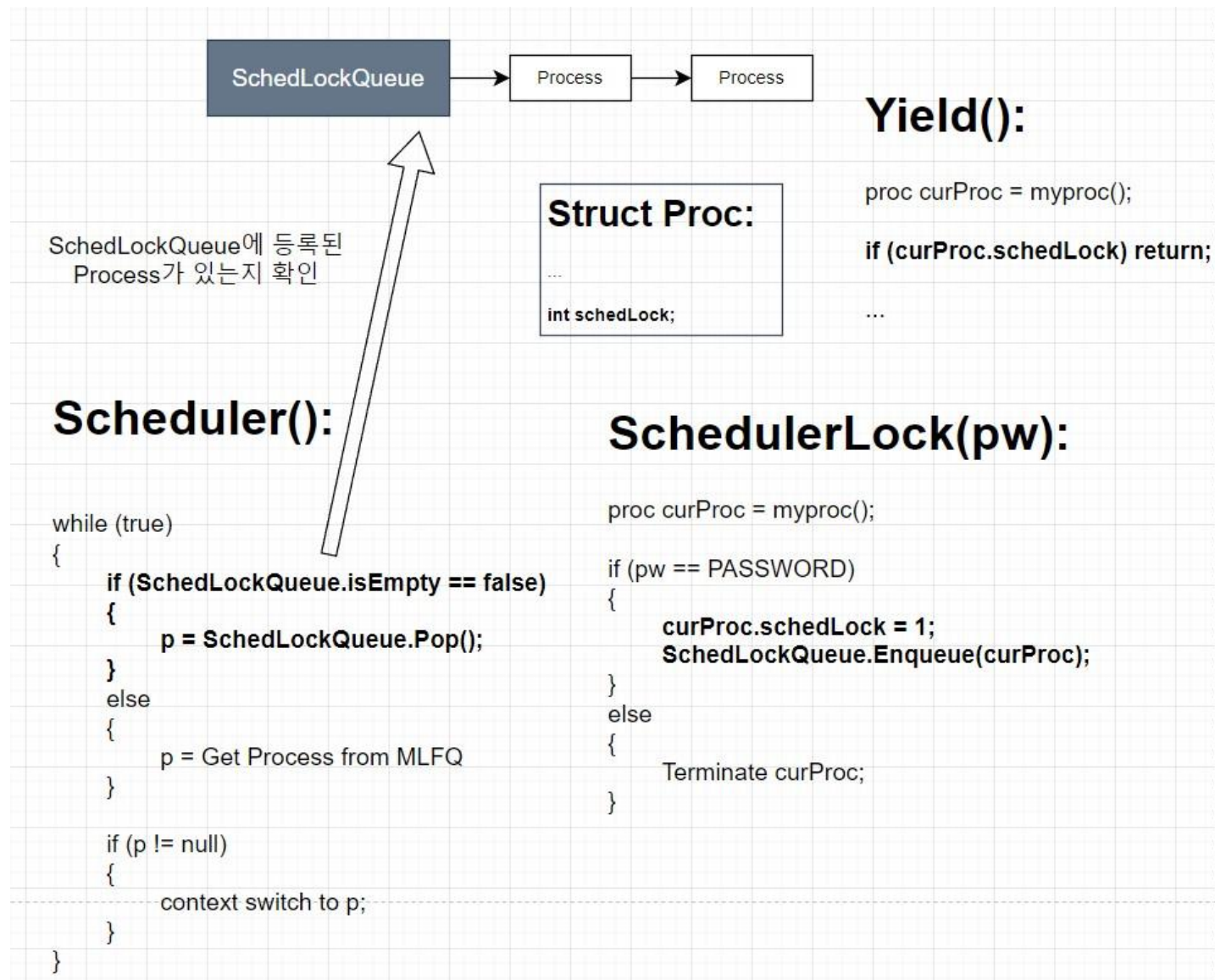
enqueue(proc, queue):

```
proc.tq = queue.tq; // reset tq
proc.q = queue;

if queue.level == 2:
    // 선형탐색으로 enqueue될 위치 찾아서 넣기
else:
    queue.tail.next = proc;
    queue.tail = proc;
```

L0, L1 Queue 에 들어가는 경우 가장 마지막, 즉 tail 의 위치로 넣어줍니다. L2 는 특별히 Priority 값을 비교해 가며 삽입될 위치를 선형적으로 찾습니다. Process 의 Time Quantum 도 이 함수에서 초기화 해줍니다.

Scheduler Lock 시스템



한 시점에 Scheduler Lock 을 필요로 하는 Process 가 여럿 있을 수 있다고 생각해, 이런 Process 들을 위한 Queue 가 따로 필요하다고 판단했습니다. 예를 들어 Process A 가 Scheduler Lock 을 선점하고 있는 동안 Process B 도 Scheduler Lock 을 잡고자 한다면, B 는 이 Queue 에 들어가는 자신의 차례를 기다립니다. A 의 수행이 종료되고 나서야 B 의 수행이 시작됩니다. 이를 위해서 Scheduler 함수에서, MLFQ 에서 Process 를 찾기 전에 이 Queue 에 기다리는 Process 가 있는지 확인해주는 과정이 필요합니다. 또한 Scheduler Lock 을 선점하고 있는 Process 는 Timer Interrupt 이 발생해도 CPU 를 양보하지 않아야 하므로, Yield 함수에서 현재 Process 가 Lock 을 선점하고 있지 않은지를 확인해줘야 합니다. 이를 위해서 Process 구조체에 schedLock 이라는 변수를 뒤 Lock 선점 여부를 저장합니다.

Priority Boosting 함수

PriorityBoost():

```
int qlevel;

for (proc p : MLFQ[0])
{
    p.timeQuantum = MLFQ[0].timeQuantum;
}

for (qlevel = 1; qlevel <= 2; qlevel++)
{
    while (MLFQ[qlevel].isEmpty == false)
    {
        p = MLFQ[qlevel].Pop();
        p.priority = 3;
        MLFQ[0].Enqueue(p);
    }
}

sched();
```

Priority Boosting 이 발생하면, 모든 프로세스의 Time Quantum 과 Priority 를 초기화시켜주고 L0 Queue 로 Enqueue 해줍니다. 다음, sched 함수를 호출해 다시 Scheduling 을 진행합니다.

SetPriority 함수

SetPriority(pid, priority):

```
int qllevel;  
proc p;
```

```
for (qllevel = 0; qllevel <= 1; qllevel++)  
    p = MLFQ[qllevel].head;  
    while (p != null)  
        if (p.pid == pid):  
            p.priority = priority;  
            return;
```

L0, L1 Queue의 Process인 경우

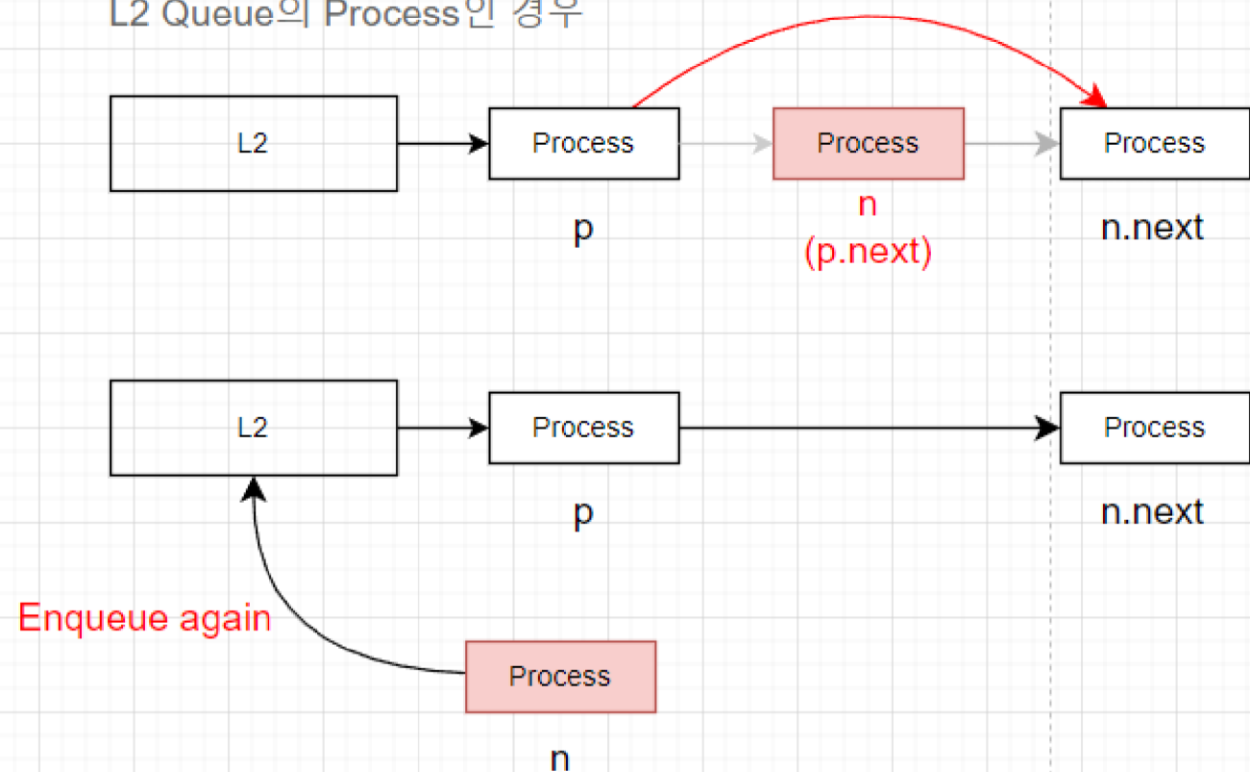
```
p = MLFQ[2].head;
```

```
while ((proc n := p.next) != null)
```

```
    if (n.pid == pid)  
        n.priority = priority;  
        p.next = n.next;  
        MLFQ[2].Enqueue(n);  
        return;
```

```
p = p.next;
```

L2 Queue의 Process인 경우



SetPriority 함수도 Enqueue 함수와 마찬가지로 대상 Process 가 L2 Queue 의 것인지 아닌지에 따라 크게 나뉩니다. L0 와 L1 Queue 의 경 우, Priority 가 Scheduling 순서와 상관이 없으므로 단순히 대상 Process 를 찾아 Priority 값을 바꾸어주면 됩니다. 반면 L2 Queue 에서는 바뀐 Priority 를 기반으로 재정렬이 필요합니다. 따라서 L2 Queue 에서 발견될 경우 Linked List 의 연결고리를 끊어 일종의 Pop 을 하고, Priority 를 바꾼뒤 다시 Enqueue 해줍니다.

2. 최종 구현

이 섹션에서는, 위 작성한 초기 디자인에서 어떻게 달라졌는지를 코드와 함께 설명하도록 하겠습니다. 간결함을 위해, 보여드릴 코드에는 주석을 많이 지웠습니다. 전체 주석은 gitlab 에서 확인하실 수 있습니다. 또한 도우미 함수나 간단한 함수들에 대한 설명은 생략하겠습니다.

자료구조 및 구조체

```
proc.h

struct queue {
    struct proc *head;
    struct proc *tail;
};

struct proc {

    ... // 기존 xv6과 일치

    int tq;                // time quantum
    int qllevel;
    int priority;
    struct proc *next;     // Next proc in linkedlist
};
```

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

struct queue mlfq[3];
```

queue 구조체의 최종 구현에는 qllevel 과 timeQuantum 을 제외시켰습니다. xv6 가 처음 구동될때 head 와 tail 은 0, 즉 null 값을 가지는 것이 당연하지만, qllevel 과 timeQuantum 은 코드상으로 초기화가 필요한데, 구현 당시 이런 초기화를 어디서 해줄지 몰라 결국 제외시켰습니다. 위 키를 작성하는 지금, 다시 생각해보면 proc.c 의 pinit 함수에서 초기화를 진행했으면 되지 않았을까 생각이 듭니다. 이렇게 queue 구조체에서 일종의 메타데이터를 제외시키고 보니, proc 구조체에도 자신이 속한 queue 에 대한 포인터가 불필요하다고 느껴졌습니다. 오히려 qllevel 을 proc 구조체가 갖고, tq (timeQuantum)을 초기화할 때는 과제 명세에 주어진 공식 ($2 * qllevel$) + 4 를 사용하는 방식으로 수정했습니다.

또한, 기존 ptable 을 지우고 queue 배열만을 사용하는 초기 디자인과 달리, ptable 과 queue 배열 모두 활용하는 디자인을 채택했습니다. 이유는 크게 두가지입니다. 첫째, MLFQ 에는 오로지 CPU Scheduling 의 대상이 되는 process 들, 즉 RUNNABLE 상태의 process 만 들어가야하기 때문입니다. 하지만 SLEEPING, EMBRYO, ZOMBIE 상태의 process 들도 OS 의 관리가 필요하므로, MLFQ 외에도 여러 process 들을 저장하는 저장소가 필요합니다. 둘째, 모든 process 의 정보가 담긴, 물리적으로 연속적인 저장소가 필요하다고 판단했습니다. Linked List 구 현은 포인터로 이어진, 논리적으로만 연속적이기 때문에, 어떤 문제로 인해 이런 연속성이 망가지면 복구하기 어려울 것입니다. 예를 들어, P0 → P1 → P2 와 같은 process 의 Linked List 에서, P0 의 next 포인터가 잘못된 주소를 가리키게 되었다면 P1 과 P2 까지 접근 불가능한 상태가 될 것입니다. 물론 이런 문제는 구현을 올바르게 한다면 발생하지 않겠지만, 그럴 자신이 없는 저는 기존의 ptable 을 건드리지 않기로 결정했습니다. 즉, 모든 process 를 물리적으로 연속적으로 ptable 에 저장하고, 논리적으로 연속적으로는 mlfq 배열의 각 Queue 가 저장하는 디자인을 채택했습니다. 기존의 ptable 을 건드리지 않음으로써 동반된 또다른 장점은, 기존 xv6 의 많은 부분들을 수정 없이 활용할 수 있었다는 점입니다. 예를 들어 allocproc, exit, kill, wait, wakeup1 등등 기존의 많은 함수들이 ptable 의 proc 배열을 활용하는데, ptable 을 수정하지 않은 덕분에 이런 함수들도 크게 수정할 필요가 없었습니다.

Scheduler 함수

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

    for(;;){
        sti();
        acquire(&ptable.lock);

        while ((p = dequeue(qllevel))) {

            c->proc = p;
            switchvm(p);
            p->state = RUNNING;

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

            c->proc = 0;
            qllevel = 0;
        }

        if (++qllevel == 3) qllevel = 0;

        release(&ptable.lock);
    }
}
```

최종 구현에서 Scheduler 함수는 변경 없이 초기 디자인을 채택했습니다.

Yield 함수

```
void
yield(void)
{
    struct proc* curproc;

    if (schedlock) {
        return;
    }

    acquire(&ptable.lock);

    curproc = myproc();
    curproc->state = RUNNABLE;

    if (curproc->tq <= 0) {

        if (curproc->qlevel == 2) {
            if (curproc->priority)
                curproc->priority--;

            curproc->tq = 8;
            enqueue(curproc, 2);
        }
        else {
            curproc->tq = 2 * (curproc->qlevel + 1) + 4;
            enqueue(curproc, curproc->qlevel + 1);
        }
    }
    else {
        enqueue(curproc, curproc->qlevel);
    }

    sched();
    release(&ptable.lock);
}
```

Yield 함수의 최종 구현은 초기 디자인에서 크게 달라졌습니다. 우선, 이 과제에서 Time Quantum의 개념이 제가 처음에 이해한 바와 달랐기 때문입니다. 제가 처음에 이해한 Time Quantum은, 한 process가 CPU를 양보하기 전까지 한번에 사용할 수 있는 시간이었습니다. 즉 P0의 Time Quantum이 4 tick이라면, P0는 4 tick 동안 CPU를 사용한 뒤 Scheduler가 다시 다음 process를 찾는 것으로 이해했습니다. 반면, 이 과제에서 Time Quantum은, 한 process가 demoting되기 전까지 CPU를 사용할 수 있는 시간입니다. 또한 이 과제에서 process는 Time Quantum을 전부 소진하지 않았더라도 CPU를 양보해 다른 process가 수행될 수 있어야 합니다.

이런 저의 오해가, 최종 디자인이 초기 디자인과 크게 달라진 유일한 이유는 아닙니다. 이 Yield 함수를 시스템 콜로 만들어, 호출되면 현재 process를 멈추고 다음 process에게 CPU를 양보하도록 구현해야 합니다. 저의 초기 디자인에서는 Time Quantum이 남아있으면 이런 양보가 진행되지 않는다는 문제점이 있습니다. 즉 남아있는 Time Quantum과 무관하게 매 tick마다 sched()를 호출하도록 아래와 같이 수정했습니다. 뿐만 아니라 Time Quantum을 감소시켜주는 작업도 이 함수에서가 아닌 trap.c의 trap() 함수에서 처리해주도록 바꾸었습니다. 실제 1 tick이 지나지 않았어도 Yield 시스템 콜을 호출했을 때 Time Quantum이 감소하는 것은 엉뚱하다고 생각했기 때문입니다.

Enqueue 함수

```
void
enqueue(struct proc *p, int qllevel)
{
    struct queue *q = &mlfq[qllevel];
    struct proc *tmp = q->head;

    p->qllevel = qllevel;

    if (!tmp) {
        q->head = p;
        q->tail = p;
        return;
    }

    if (qllevel == 2) {
        while (tmp->next && p->priority >= tmp->next->priority) {
            tmp = tmp->next;
        }
        p->next = tmp->next;
        if (!p->next)
            q->tail = p;
        tmp->next = p;
    }
    else {
        q->tail->next = p;
        q->tail = p;
    }
}
```

최종 디자인에서는 Process 의 Time Quantum 을 초기화시켜주는 부분은 제외시켰습니다. 한 tick 이 지나면, Time Quantum 을 모두 소진하지 않았어도 Enqueue 를 진행해줘야하기 때문입니다. 따라서 Time Quantum 의 초기화는, yield 함수에서 demoting 이 발생할 시에 해주도록 디자인을 바꾸었습니다.

Scheduler Lock 시스템

최종 구현에서 Scheduler Lock 시스템은 초기 디자인과는 사뭇 다른 디자인을 채택했습니다. 우선, Scheduler Lock Queue 를 둘 필요가 없다고 판단했습니다 - 이 프로젝트에서는 CPU 가 하나임을 가정하므로, 애초에 한 Process 가 Scheduler Lock 을 선점한 동안에 다른 Process 가 SchedulerLock 함수를 호출할 수 없을 것입니다. 이 말은 곧, 한 시점에 Scheduler Lock 을 선점하고 있는 Process 는 최대 하나임이 보장 된다고 생각합니다. 이런 이유로, Scheduler Lock 선점 여부도 Process 구조체에 둘 필요 없이, 하나의 전역변수로 관리하는 것이 적절하다고 판단했습니다.

```
int schedlock = 0;
```

```
void
schedulerLock(int password)
{
    int pid;
    int tq;
    int qlevel;
    struct proc *curproc = myproc();

    if (password == pw) {
        acquire(&tickslock);

        schedlock = 1;
        ticks = 0;

        release(&tickslock);
    }
    else {
        pid = curproc->pid;
        tq = curproc->tq;
        qlevel = curproc->qlevel;

        kill(pid);
        cprintf("Lock failed: incorrect password\n");
        cprintf("pid: %d timequantum: %d queue: L%d\n", pid, 2 * qlevel + 4 - tq, qlevel);
    }
}
```

Scheduler Lock 함수 역시 Enqueue 없이 이 전역변수를 1 로 세팅해줍니다. 또한 trap.c 에 있는 ticks 를 0 으로 초기화시켜줍니다.

```

void
schedulerUnlock(int password)
{
    int pid;
    int tq;
    int qllevel;
    struct proc *curproc = myproc();

    if (!schedlock) return;

    if (password == pw) {

        acquire(&ptable.lock);

        schedlock = 0;

        curproc->tq = 4;
        curproc->priority = 3;
        movefront(curproc);
        curproc->state = RUNNABLE;

        sched();

        release(&ptable.lock);
    }
    else {
        pid = curproc->pid;
        tq = curproc->tq;
        qllevel = curproc->qllevel;

        kill(pid);
        cprintf("Unlock failed: incorrect password\n");
        cprintf("pid: %d timequantum: %d queue: L%d\n", pid, 2 * qllevel + 4 - tq, qllevel);
    }
}

```

Scheduler Unlock 함수에서는 schedLock 을 다시 0 으로 초기화시켜주고 해당 Process 를 L0 Queue 의 가장 앞으로 이동시켜줍니다. 다음, sched 함수를 호출해 다시 MLFQ Scheduling 으로 돌아갑니다.

Scheduler Lock 이 걸려 있지 않은 상태에서 이 SchedulerUnlock 함수를 호출한 상황을 어떻게 처리해주어야 할까 많은 고민을 했습니다. 처음에는, 인자로 받은 비밀번호가 틀렸을 때와 마찬가지로 해당 Process 를 강제종료하는 것을 고려했으나, 결국 아무런 처리를 하지 않기로 결 정했습니다. 그 이유는, 이렇게 Lock 이 걸려있지 않은데 Unlock 을 시도하는 상황이 꼭 부자연스럽지는 않다고 느꼈기 때문입니다. 예를 들어, 한 Process 가 Lock 을 호출하고 다시 Unlock 을 호출하는 시간 동안에 Priority Boosting 이 발생한다면, 즉 100 tick 내로 Unlock 을 호출하지 않는다면 이 과정에서 Unlock 이 이미 진행될 것입니다. 이런 상황에서 Unlock 을 호출하는 것이 Process 의 강제종료로 이어진다는 것은, 개발자가 자신의 프로그램이 100 tick 안에 수행이 될지 안될지를 고려해야한다는 말이나 다름 없습니다.

```

if(!schedlock && myproc() && myproc()->state == RUNNING &&
    tf->trapno == T_IRQ0+IRQ_TIMER) {
    myproc()->tq--;
    yield();
}

```

trap.c 에서는 schedlock 이 걸려있지 않은 경우에만 Time Quantum 을 감소시키고 Yield 를 호출합니다.

```

int
sys_schedulerLock(void)
{
    int pw;
    if (argint(0, &pw) < 0)
        return -1;

    schedulerLock(pw);
    return 0;
}

int
sys_schedulerUnlock(void)
{
    int pw;
    if (argint(0, &pw) < 0)
        return -1;

    schedulerUnlock(pw);
    return 0;
}

```

두 Lock 과 Unlock 함수에 대한 Wrapper Function 입니다.

```

case SCHEDLOCK:
    schedulerLock(2020059152);
    break;
case SCHEDUNLOCK:
    schedulerUnlock(2020059152);
    break;

```

Interrupt 를 통해 호출될 수 있도록 각각에 대한 Interrupt 번호를 등록하고, trap.c 의 trap 함수에서 case 문을 작성해주었습니다. Interrupt 를 통해 호출되는 경우 인자에 미리 올바른 비 번호를 넣어두었기 때문에 성공적으로 수행이 되도록 구현했습니다.

Priority Boosting 함수

```
void
prboost(void)
{
    struct proc *p;
    int qlevel;

    acquire(&ptable.lock);

    p = mlfq[0].head;
    while (p) {
        p->tq = 4;
        p = p->next;
    }
    for (qlevel = 1; qlevel <= 2; qlevel++) {
        while ((p = dequeue(qlevel))) {
            p->tq = 4;
            p->priority = 3;
            enqueue(p, 0);
        }
    }

    p = myproc();
    if (p) {
        if (schedlock) {
            schedlock = 0;
            movefront(p);
        }
        else {
            p->priority = 3;
            p->tq = 1;
            p->qlevel = -1;
        }
    }

    release(&ptable.lock);
}
```

초기 디자인에서 간과한 부분이 있었습니다 - Priority Boosting 이 발생할 때 현재 실행되는 Process 를 처리하는 과정을 추가했습니다. 해당 Process 가 Scheduler Lock 을 선점하고 있었다라면, lock 을 해제해주고 L0 Queue 의 가장 앞으로 이동합니다. 그 외 Process 는 Time Quantum 과 Priority 를 초기화해준 뒤 L0 Queue 의 가장 뒤로 이동하도록 해주었습니다. 위 코드에서 $p \rightarrow tq = 1$; 과 $p \rightarrow qlevel = -1$; 부분은, 인정하기 부끄럽지만 잔피를 부린 것입니다: trap.c 에서 Priority Boost 을 호출한 뒤, Time Quantum 도 감소시켜주며 Yield 함수도 호출되므로, 저의 Enqueue 함수 디자인상 "하위" Queue 로 내려갑니다. 하위 Queue 란 현재 qlevel 보다 1 큰 인덱스의 Queue 로 가는 것이며, 이렇게 qlevel = -1 을 해줌으로써 Yield 함수에서 L0 Queue 로 이동을 하게 됩니다. 한마디로, 제 PriorityBoost 함수는 Yield 와 함께 호출되어야만 정상적으로 작동합니다. 이 Priority Boost 함수 자체에서 sched()나 yield()를 호출해주는 것이 이상적이겠지만, 시도한 결과 에러가 발생했고 함수 로직도 서로 뒤엉켜버려 결국 마음에 드는 방식은 아니지만, 시간과 구현 능력의 부족으로 결국 채택한 방식입니다.

SetPriority 함수

```
void
setPriority(int pid, int priority)
{
    struct queue *q;
    struct proc *t;
    struct proc *t1;
    int found = 0;
    int qllevel = 0;

    if (myproc()->pid == pid) {
        myproc()->priority = priority;
        found = 1;
    } else {
        acquire(&ptable.lock);

        for (; qllevel < 3; qllevel++) {
            q = &mlfq[qllevel];
            t = q->head;
            if (!t) continue;
            if (t->pid == pid) {
                found = 1;
                t->priority = priority;
                break;
            }
            while (t && t->next) {
                if (t->next->pid == pid) {
                    found = 1;
                    t1 = t->next;
                    t1->priority = priority;

                    if (qllevel == 2) {
                        if (q->tail == t1)
                            q->tail = t;

                        t->next = t1->next;
                        enqueue(t1, 2);
                        break;
                    }
                }
                t = t->next;
            }
            if (found) break;
        }
        release(&ptable.lock);
    }

    if (!found) {
        cprintf("setPriority Error: process with pid not found\n");
    }
}
```

기본적인 작동 방식은 초기 디자인에서 설명드린 바와 같지만, 현재 실행되는 Process 를 대상으로 이 함수가 호출되는 경우에 대한 처리를 추가했습니다. 현재 Process 가 L2 Queue 의 것이어도 이미 Scheduler 함수에서 Dequeue 를 해주었기 때문에 복잡한 추가 과정이 필요하지 않습니다.

PrintQueue 함수

다음 설명드릴 PrintQueue 함수는 명세에서 요구한 함수는 아니지만, 이후 실행 결과 화면을 보여드릴 때 사용되므로 간단한 설명을 작성하겠습니다. 기존 proc.c 에 procdump 라는 함수가 있습니다 - procdump 는 디버깅용으로, 현재 모든 Process 에 대한 정보를 나열해줍니다. 이 함수는 console.c 에서 사용자가 Ctrl + P 를 누를 때 호출됩니다. 이 함수에서 영감을 받아, 비슷한 메커니즘으로 작동하는 printqueue 함수를 만들었습니다.

```
void
printqueue(void)
{
    int qlevel = 0;
    struct proc *t = myproc();
    struct queue *q;

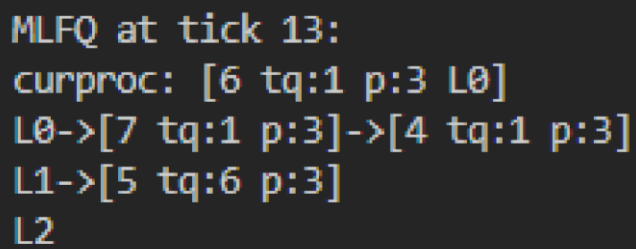
    cprintf("\nMLFQ at tick %d:\n", ticks);

    if (t) {
        cprintf("curproc: [%d tq:%d p:%d L%d]", t->pid, t->tq, t->priority, t->qlevel);
        if (schedlock) cprintf(" LOCKED");
        cprintf("\n");
    }

    for(; qlevel < 3; qlevel++) {
        cprintf("L%d", qlevel);

        q = &mlfq[qlevel];
        t = q->head;

        while (t) {
            cprintf("->[%d tq:%d p:%d]", t->pid, t->tq, t->priority);
            t = t->next;
        }
        cprintf("\n");
    }
    cprintf("\n");
}
```



```
MLFQ at tick 13:
curproc: [6 tq:1 p:3 L0]
L0->[7 tq:1 p:3]->[4 tq:1 p:3]
L1->[5 tq:6 p:3]
L2
```

```
void
consoleintr(int (*getc)(void))
{
    int c, doprocdump = 0;

    acquire(&cons.lock);
    while((c = getc()) >= 0){
        switch(c){
            case C('P'): // Process listing.
                // procdump() locks cons.lock indirectly; invoke later
                doprocdump = 1;
                break;

            case C('Q'): // DEBUG. ctrl Q
                doprintqueue = !doprintqueue;
                break;

            ...
        }
    }
}
```

이 함수도 procdump 와 마찬가지로 console.c 에 등록해 두었습니다. 다만 Ctrl Q 를 눌렀을 때 한번만 출력하는 것이 아닌, 전역변수 doprintqueue 를 toggle 하는 방식이며, 이 doprintqueue 값이 1 이면 trap.c 에서 매 Timer Interrupt 발생마다 출력되도록 했습니다.

최종 구현 디자인 요약

구현을 마치고, 이 위키를 작성하는 중에 뒤늦게 제가 명세에서 오해했던 부분을 몇몇 발견했습니다. 아쉽게도 시간이 얼마 남지 않아 수정을 하지 못했고, 이런 오해점들의 흔적은 최종 구현 코드에 고스란히 잔재합니다. 따라서 이 섹션에서 저의 이런 오해점들, 그리고 추가로 더 강조하고 싶은 내용을 정리해서 작성해보았습니다.

1. "명세에 나와있는 time quantum 은 Process 가 해당 Queue 에서 사용한 시간을 의미합니다."

저의 구현에서 한 Process 의 Time Quantum 은 Demoting 되기 전까지 남은 시간을 의미합니다. 즉 매 tick Time Quantum 이 1 씩 감소하고, 이 값이 0 이 되면 Demoting 이 발생합니다. SchedulerLock / Unlock 함수에 잘못된 비밀번호를 인자로 넘겼을 때의 출력문에는 조교님께서 말씀하신 의미의 Time Quantum 을 출력하도록 수정했으나, 이외 부분에서의 의미는 명세의 요구사항과 다릅니다.

2. Scheduler 는 하나의 Process 를 발견하면 다시 L0 Queue 로 돌아가 다음 Process 를 찾습니다.
3. Scheduler Lock 이 걸려있지 않은 상황에서 Unlock 이 호출되어도 문제가 발생하지 않습니다. 이런 디자인이 적절하다고 생각한 이유는 최종 구현 - Scheduler Lock 시스템 파트에 작성해두었습니다.
4. SchedulerLock 을 선점한 Process 가 Sleeping 상태로 전이할 시 Lock 을 해제합니다.

3. 실행 과정 및 결과

컴파일 및 실행

xv6-public 디렉토리로 들어가서 아래 두 줄을 순서대로 입력합니다.

`./makemake.sh`

`./bootxv6.sh`

```
$ makemake.sh X
xv6-public > $ makemake.sh
1  #!/bin/bash
2
3  make clean
4  make
5  make fs.img
6
$ bootxv6.sh X
xv6-public > $ bootxv6.sh
1  #!/bin/bash
2
3  qemu-system-i386 -nographic -serial mon:stdio -hdb fs.img xv6.img -smp 1 -m 512
4  |
```

실행 결과

위에 설명드린 printqueue 함수를 사용해 작동 결과를 보여드리겠습니다. 앞서 언급한 저의 오해로 인해 아래 사진들에서 printqueue 를 통해 출력되는 tq, 즉 Time Quantum 의 의미는 Demoting 되기 전까지 남은 tick 이라는 점 양해 부탁드립니다.

```
MLFQ at tick 27:
L0
L1
L2

MLFQ at tick 28:
curproc: [3 tq:3 p:3 L0]
L0
L1
L2

MLFQ at tick 29:
curproc: [3 tq:2 p:3 L0]
L0->[4 tq:4 p:3]
L1
L2
```

처음 실행된 프로세스는 가장 높은 레벨의 큐(L0)에 들어갑니다.

```
MLFQ at tick 49:
curproc: [7 tq:1 p:3 L0]
L0->[8 tq:1 p:3]
L1->[4 tq:6 p:3]->[5 tq:6 p:3]->[6 tq:6 p:3]
L2

MLFQ at tick 50:
curproc: [8 tq:1 p:3 L0]
L0
L1->[4 tq:6 p:3]->[5 tq:6 p:3]->[6 tq:6 p:3]->[7 tq:6 p:3]
L2

MLFQ at tick 51:
curproc: [4 tq:6 p:3 L1]
L0
L1->[5 tq:6 p:3]->[6 tq:6 p:3]->[7 tq:6 p:3]->[8 tq:6 p:3]
L2
```

스케줄러는 기본적으로 L0 큐의 RUNNABLE 한 process 를 스케줄링합니다.

L0 큐의 RUNNABLE 한 프로세스가 없을 경우, L1 의 process 를 스케줄링합니다.

L1 큐의 RUNNABLE 한 프로세스가 없을 경우, L2 의 process 를 스케줄링합니다.

```

MLFQ at tick 55:
curproc: [6 tq:2 p:3 L0]
L0->[4 tq:1 p:3]->[7 tq:3 p:3]->[8 tq:3 p:3]
L1->[3 tq:4 p:3]
L2

MLFQ at tick 56:
curproc: [4 tq:1 p:3 L0]
L0->[7 tq:3 p:3]->[8 tq:3 p:3]->[6 tq:1 p:3]
L1->[3 tq:4 p:3]
L2

```

L0 큐와 L1 큐는 기본 Round-Robin 정책을 따릅니다. 매 tick 마다 현재 Process 의 Time Quantum 을 감소시키고 Queue 의 가장 뒤로 이동시킵니다.

```

MLFQ at tick 56:
curproc: [4 tq:1 p:3 L0]
L0->[7 tq:3 p:3]->[8 tq:3 p:3]->[6 tq:1 p:3]
L1->[3 tq:4 p:3]
L2

MLFQ at tick 57:
curproc: [7 tq:3 p:3 L0]
L0->[8 tq:3 p:3]->[6 tq:1 p:3]
L1->[3 tq:4 p:3]->[4 tq:6 p:3]
L2

```

Time Quantum 을 모두 소진한 L0, L1 Process 는 Time Quantum 이 초기화되고 하위 Queue 로 내려갑니다.

```

MLFQ at tick 51:
curproc: [5 tq:1 p:3 L2]
L0
L1
L2->[6 tq:1 p:3]->[4 tq:1 p:3]

MLFQ at tick 52:
curproc: [6 tq:1 p:3 L2]
L0
L1
L2->[5 tq:8 p:2]->[4 tq:1 p:3]

```

L2 큐에서 실행된 프로세스가 L2 에서의 time quantum 을 모두 사용한 경우, 해당 프로세스의 priority 값이 1 감소하고 time quantum 은 초기화됩니다.

```

MLFQ at tick 38:
curproc: [4 tq:1 p:3 L1]
L0
L1->[5 tq:1 p:3]
L2->[6 tq:8 p:3]->[7 tq:8 p:3]

MLFQ at tick 39:
curproc: [5 tq:1 p:3 L1]
L0
L1
L2->[6 tq:8 p:3]->[7 tq:8 p:3]->[4 tq:8 p:3]

```

우선순위가 같은 프로세스끼리는 FCFS 로 스케줄링 됩니다.

```

MLFQ at tick 99:
curproc: [5 tq:2 p:1 L2]
L0
L1
L2->[4 tq:1 p:1]->[6 tq:1 p:3]->[7 tq:1 p:3]

MLFQ at tick 0:
curproc: [4 tq:4 p:3 L0]
L0->[6 tq:4 p:3]->[7 tq:4 p:3]->[5 tq:4 p:3]
L1
L2

```

Global tick 이 100 ticks 가 될 때마다 모든 프로세스들은 L0 큐로 재조정됩니다.

Priority boosting 이 될 때, 모든 프로세스들의 priority 값은 3 으로 재설정됩니다.

Priority boosting 이 될 때, 모든 프로세스들의 time quantum 은 초기화됩니다.

```

MLFQ at tick 63:
L0->[4 tq:4 p:3]->[3 tq:2 p:3]
L1
L2

MLFQ at tick 1:
curproc: [5 tq:4 p:3 L0] LOCKED
L0->[3 tq:2 p:3]->[4 tq:3 p:3]
L1
L2

MLFQ at tick 2:
curproc: [5 tq:4 p:3 L0] LOCKED
L0->[3 tq:2 p:3]->[4 tq:3 p:3]
L1
L2

```

SchedulerLock 시스템 콜이 성공적으로 실행되면, global tick 은 priority boosting 없이 0 으로 초기화 됩니다. 스케줄러를 lock 하는 프로세스가 존재할 경우 MLFQ 스케줄러는 동작하지 않고, 해당 프로세스가 최우선적으로 스케줄링됩니다.

```

MLFQ at tick 8:
curproc: [6 tq:4 p:3 L0] LOCKED
L0->[3 tq:2 p:3]
L1
L2->[4 tq:6 p:3]->[5 tq:6 p:3]

MLFQ at tick 9:
curproc: [6 tq:3 p:3 L0]
L0->[7 tq:4 p:3]
L1
L2->[4 tq:6 p:3]->[5 tq:6 p:3]

```

SchedulerUnlock 시스템 콜이 성공적으로 실행되었다면, 기존의 MLFQ 스케줄러로 돌아갑니다. 해당 프로세스는 L0 큐의 제일 앞으로 이동하고, priority와 time quantum이 초기화됩니다. 위 사진의 경우 L0 Queue의 제일 앞으로 이동해 바로 다시 Scheduling이 된 것을 볼 수 있습니다.

```

MLFQ at tick 29:
curproc: [6 tq:1 p:3 L1]
L0
L1
L2->[5 tq:8 p:3]->[4 tq:8 p:3]

Lock failed: incorrect password
pid: 6 timequantum: 5 queue: L1

MLFQ at tick 29:
curproc: [5 tq:8 p:3 L2]
L0
L1
L2->[4 tq:8 p:3]

```

두 시스템 콜 호출 시, 암호가 일치하지 않으면 해당 프로세스를 강제로 종료합니다. 직후 pid 6의 Process는 강제종료되어 MLFQ에서 사라지는 것을 볼 수 있습니다. 또한 해당 프로세스의 pid, time quantum, 현재 위치한 큐의 level을 출력합니다. 이 출력문의 경우 명세에서 요구하는 time quantum의 의미를 따랐습니다.

```
$ mlfqtest
MLFQ test start
[Test 1] default
Process 12
L0: 17666
L1: 25684
L2: 56650
L3: 0
L4: 0
Process 10
L0: 23646
L1: 31897
L2: 44457
L3: 0
L4: 0
Process 13
L0: 23951
L1: 31543
L2: 44506
L3: 0
L4: 0
Process 11
L0: 23838
L1: 34320
L2: 41842
L3: 0
L4: 0
[Test 1] finished
done
$ █
```

조교님께서 Piazza 에 올려주신 테스트 코드 실행 결과입니다.

4. Trouble Shooting

이번 프로젝트에서 가장 큰 난관은, 초반의 디자인을 구상하는 것과 디버깅의 어려움이었습니다. 특히 디버깅의 어려움은 인해 제 코드의 정당 성을 따지기 어려움으로 이어졌고, "코드는 잘 짠 것 같은데 왜, 어디서 문제가 발생하지?"라는 질문을 여러번 던져야 했습니다. 예를 들어, 구 현 초반에 이런 문제가 있었습니다: Scheduler 함수를 작성 완료하고 실행해보자, sh Process, 즉 \$ 표시를 보여주며 사용자의 입력을 받는 Process 가 정상적으로 실행되지 않았습니다. 제가 입력한 글자들이 화면에 뜨기는 하나, 엔터를 눌러도 반응이 없었습니다. Scheduler 함수 구현을 잘못했다고 생각해 함수 이곳 저곳에 출력문을 작성해 디버깅을 시도했습니다. 그렇게 오랜 시간 고민해도 해결이 되지 않아, 설마하는 마음으로 기존 작성한 모든 출력문을 지워보았더니 정상적으로 실행이 되는 일이 있었습니다.

단순히 저의 어리석은 구현 실수로 인해 며칠을 낭비한 적도 많습니다. 가장 어이없었던 실수는 Priority Boost 함수를 구현할 때였습니다. 모 든 Process 의 Time Quantum 과 Priority 를 초기화시키고 L0 Queue 로 이동해주기 위해, 우선 모든 Queue 를 논리적 순서대로 순회해주어야 합니다. 그러나 저는 어리석게도 L0 ~ L2 순서대로, 각 Queue 가 빌 때까지 Dequeue 를 한뒤 다시 L0 에 Enqueue 하는 코드를 작성했었습니다. L0 Queue 에서 Dequeue 를 한뒤 다시 L0 Queue 로 Enqueue 를 해주니, 애초에 L0 Queue 가 비어있던 상황이 아니라면 이 로직은 무한 반복문이 되어버립니다. 아무런 생각 없이 이 코드를 작성한 저는 Priority Boosting 이 발생할 때마다 멈춰버리는 xv6 를 마주하게 되었고, PriorityBoost 함수와 Yield 함수가 같이 호출되어서 생긴 문제라고 굳건히 믿은 저는 며칠동안 헛발질을 했습니다.

외에도 정말 많은 문제가 있었지만, 아쉽게도 제출 마감시간이 임박함과 다른 시험들을 준비해야 하므로 급하게 글을 마무리하겠습니다. 4 월 2 일부터 위키를 쓰는 오늘 4 월 23 일까지라는 상당한 시간이 걸렸으며 결코 쉽지만은 않았지만, 나름 재밌고 유익한 경험이었습니다. 긴 글 읽어 주셔서 감사합니다.