



Project01

1. Design

이번 프로젝트에서 구현해야 하는 것은 크게 세 가지입니다.

1. FCFS Scheduling
2. MLFQ Scheduling
3. Mode Switch를 포함한 System calls

Scheduling

두 스케줄링을 구현하기 위해서는 Queue 구조체가 필요합니다.

FCFS queue와 I0~I2 queue까지 총 4개의 queue를 필요로 합니다.

Common

1. 먼저 앞서 말한 Queue 구조체를 구현합니다.
2. 두 스케줄링 모두 각자의 queue에서 관리를 해줘야 하므로 프로세스가 새롭게 생겼을 때 적절한 level의 queue에 삽입하는 코드를 작성합니다.
3. timer interrupt가 발생할 시 mode에 따라 CPU를 반납할지 아닐지 결정하는 코드를 작성합니다.
4. CPU를 반납하게 되는 경우는 세 가지입니다. 각 경우에 따라 적절한 조치를 하도록 코드를 작성합니다.
 - a. timer interrupt가 발생하기 전 프로세스가 작업을 끝내 스스로 CPU를 반납하는 경우가 있습니다. 이 경우에는 프로세스를 queue에서 제거해줍니다.
 - b. timer interrupt가 발생하여 프로세스가 작업이 끝나지 않았음에도 CPU를 반납하는 경우가 있습니다. 이 경우에는 프로세스를 MLFQ의 조건에 따라 다음 queue로 이동하거나 우선순위를 변경합니다.
 - c. sleep 또는 yield를 하는 경우가 있습니다. 이 경우에는 아직 프로세스가 작업이 끝나지 않았으므로 아무런 조치를 취하지 않고 다음 프로세스로 스케줄링만을 진행합니다.

5. CPU를 반납하는 경우 외에도 priority_boosting으로 인해 현재 ticks에 따라 모든 프로세스를 이동시키고 우선순위를 변경하는 코드를 작성합니다.

FCFS

1. FCFS는 하나의 FCFS queue만을 사용합니다.
2. timer interrupt가 없고 priority boosting이 없습니다.
3. 프로세스가 생성된 순서대로 queue에 들어오게 하여 FCFS 조건을 만족하도록 코드를 작성합니다.

MLFQ

1. MLFQ는 I0~I2까지 세 개의 queue를 사용합니다.
2. timer interrupt가 존재하여 1틱마다 yield가 되도록 코드를 작성합니다.
3. 이때 프로세스가 속한 queue에 따라 조건에 맞춰 다음 queue로의 이동이나 우선순위 변경이 되도록 코드를 작성합니다.
4. priority boosting이 존재하여 글로벌 틱이 50틱마다 boosting되어 I0 queue로의 이동과 우선순위 초기화가 되도록 코드를 작성합니다.
5. 프로세스가 생성된 순서대로 I0 queue에 들어오게 하여 MLFQ 조건을 만족하도록 코드를 작성합니다.

System calls

총 5개의 System call을 구현해야 합니다.
그 중 Mode Switch가 중요합니다.

mlfqmode

1. FCFS mode에서 MLFQ mode로 변경되도록 코드를 작성합니다.
2. FCFS에서의 순서에 맞춰 I0 queue로 모든 프로세스를 이동시킵니다.
3. 우선순위와 time quantum, level을 조정합니다.
4. 만약 이미 MLFQ mode였을 경우 에러 메시지를 출력합니다.

fcfsmode

1. MLFQ mode에서 FCFS mode로 변경되도록 코드를 작성합니다.
2. 프로세스의 생성 순서에 맞춰 fcfs queue로 이동시킵니다.

3. 우선순위와 time quantum, level을 조정합니다.
4. 만약 이미 FCFS mode였을 경우 에러 메시지를 출력합니다.

Mode Switch 외의 System call들은 명세에 맞춰 적절한 기능이 되도록 코드를 작성합니다.

2. Implementation

Queue

Design에서 말했 듯이 queue 구조체를 구현해야 합니다.

```
//proc.c
struct queue
{
    struct proc *queue[NPROC];
    int time_quantum;
    int level;
    int size;
    int head;
    int tail;
};

void
enqueue(struct queue *q, struct proc *p)
{
    if(q->size == NPROC) {
        return;
    }

    q->queue[q->tail] = p;
    q->tail = q->tail + 1;
    q->size++;
}

void
dequeue(struct queue *q, struct proc *p)
```

```

{
    if(q->size == 0) {
        return;
    }

    for(int i = q->head; i < q->tail; i = i + 1) {
        if(q->queue[i]->pid == p->pid) {
            for(int j = i; j < q->tail; j = j+1) {
                q->queue[j] = q->queue[j+1];
            }
            q->tail = q->tail - 1;
            q->size--;
            break;
        }
    }
}

int
queue_is_empty(struct queue *q)
{
    int count = 0;
    for(int i = q->head; i < q->tail; i = i + 1) {
        if(q->queue[i]->state != RUNNABLE && q->queue[i]->state != RUNNING) {
            count++;
        }
    }
    return (q->size - count) == 0;
}

struct proc*
queue_head(struct queue *q)
{
    if(q->size == 0) {
        return 0;
    }

    return q->queue[q->head];
}

```

```

struct proc*
queue_pick(struct queue *q)
{
    if(queue_is_empty(q)) {
        return 0;
    }
    for(int i = q->head; i < q->tail; i = i + 1) {
        struct proc *p = q->queue[i];
        if(p->state == RUNNABLE) {
            return p;
        }
    }
    return 0;
}

```

```

struct proc*
queue_top(struct queue *q)
{
    if(queue_is_empty(q)) {
        return 0;
    }

```

```

    int idx = 0;
    int max_priority = -1;
    struct proc* p;
    for(int i = q->head; i < q->tail; i = i + 1) {
        p = q->queue[i];
        if(p->state != RUNNABLE) {
            continue;
        }
        if(p->priority > max_priority) {
            idx = i;
            max_priority = p->priority;
        }
    }
    return q->queue[idx];
}

```

위와 같이 queue 구조체를 구현했습니다.

각각의 queue는 time_quantum, level, size, head, tail을 가집니다.

queue에 프로세스를 넣고 빼는 enqueue, dequeue를 구현합니다.

queue가 비었는지 확인하는 queue_is_empty를 구현합니다.

queue의 head 자리에 있는 프로세스를 return하는 queue_head를 구현합니다.

queue의 RUNNABLE 프로세스 중 가장 앞쪽에 있는 프로세스를 return하는 queue_pick을 구현합니다.

queue의 RUNNABLE 프로세스 중 우선순위가 가장 높은 프로세스를 return하는 queue_top을 구현합니다.

뒤에서 자세히 서술하겠지만 제 구현 방식에서는 queue에서 SLEEPING인 프로세스를 제거하지 않습니다.

그렇기에 queue_is_empty에서는 CPU를 잡을 수 있는 프로세스의 개수만을 판별해야 하기에 q->size에서 RUNNABLE 또는 RUNNING이 아닌 프로세스의 개수를 빼주어 판별합니다.

queue_pick와 queue_top 역시 마찬가지로 RUNNABLE이 아닌 경우를 무시한 채 프로세스를 찾게 됩니다.

Proc

앞에서 만든 queue 구조체와 초기화, mode 변경을 위한 변수, 스케줄링을 위해 필요한 프로세스의 추가된 변수입니다.

```
//proc.c
struct queue queues[4];

int mode = 1;//fcfs = 1, mlfq = 0

void
procinit(void)
{
    ...

    for(int i=0;i<4;i++) {
        queues[i].time_quantum = 2*i+1;
        queues[i].level = i;
        queues[i].size = 0;
        queues[i].head = 0;
        queues[i].tail = 0;
```

```

    }
    queues[3].time_quantum = -1;
    queues[3].level = -1;
}

//proc.h
struct proc {
    ...

    int priority;
    int tick;
    int queue_level;
};

```

proc.c

먼저 proc.c에 위에서 만든 queue를 배열로 총 4개 추가해줍니다.

그리고 mode 변경에 필요한 변수도 추가해줍니다.

procinit 함수는 제일 처음 proc 배열을 초기화해줄 때 사용되는 함수입니다.

이때 queue 배열의 초기화도 같이 진행합니다.

FCFS queue는 time_quantum과 level을 -1로 하고 size, head, tail을 0으로 합니다.

I0~I2 queue는 time_quantum을 $2i+1$, level을 i 로 하고 size, head, tail을 0으로 합니다.

proc.h

MLFQ mode에서는 I2 queue에 있을 때 각 프로세스들의 우선순위를 비교하여 스케줄링을 하게 됩니다.

또한 프로세스가 특정 queue의 time quantum을 초과할 경우 다음 queue로 이동하거나 우선순위를 변경하게 됩니다.

그렇기에 각 프로세스들이 어떤 큐에 존재하는지, 현재 얼마나 CPU를 사용했는지, 우선순위가 어떻게 되는지를 저장하는 변수를 추가해줍니다.

Scheduler

새로운 스케줄러는 다음과 같이 구현했습니다.

```

//proc.c
void

```

```

scheduler(void)
{
    struct proc *p = 0;
    struct cpu *c = mycpu();
    c→proc = 0;
    for(;;){
        intr_on();

        int pass = 0;
        if(mode==1) { //fcfs
            if(!queue_is_empty(&queues[3])) {
                p = queue_pick(&queues[3]);
                acquire(&p→lock);
                p→state = RUNNING;
                c→proc = p;
                swtch(&c→context, &p→context);
                c→proc = 0;
                release(&p→lock);
            }
        }
        else if(mode==0) { //mlfq(l0, l1)
            for(int i = 0; i < 2; i++) {
                if(queue_is_empty(&queues[i])) {
                    continue;
                }
                pass = 1;
                p = queue_pick(&queues[i]);
                acquire(&p→lock);

                if(p→tick >= queues[i].time_quantum) {
                    p→queue_level++;
                    p→tick = 0;
                    dequeue(&queues[i],p);
                    enqueue(&queues[p→queue_level],p);
                    release(&p→lock);
                    break;
                }
            }
            else {

```



```

    p→state = RUNNING;
    c→proc = p;
    swtch(&c→context, &p→context);
    c→proc = 0;
    release(&p→lock);
    break;
}
}

if(pass == 1) continue;
if(!queue_is_empty(&queues[2])) { //mlfq(l2)
    p = queue_top(&queues[2]);
    acquire(&p→lock);

    if(p→tick >= queues[2].time_quantum) {
        p→priority--;
        if(p→priority < 0) {
            p→priority = 0;
        }
        p→tick = 0;
        release(&p→lock);
    }
    else {
        p→state = RUNNING;
        c→proc = p;
        swtch(&c→context, &p→context);
        c→proc = 0;
        release(&p→lock);
    }
}
}
}
}
}

```

위 스케줄러는 mode에 따라 FCFS 또는 MLFQ로 스케줄링을 결정합니다.
두 경우로 나눠 설명하겠습니다.

FCFS

1. FCFS queue(queues[3])가 비어있지 않은지 확인합니다.(RUNNABLE한 프로세스가 있는지 확인)
2. 비어있지 않은 경우 queue_pick을 통해 가장 앞쪽의 RUNNABLE한 프로세스를 선택하여 context switch가 진행됩니다.

MLFQ

1. 먼저 MLFQ queue 중 I0와 I1이 비어있지 않은지 순서대로 확인합니다.
2. I0가 비어있지 않은 경우 queue_pick을 통해 가장 앞쪽의 RUNNABLE한 프로세스를 선택하며 pass 변수를 1로 바꿉니다. pass는 I0 또는 I1에서 프로세스를 선택한 경우 I2는 확인하지 않아야 하기에 I2 queue를 무시하기 위한 변수입니다.
3. 만약 선택한 프로세스가 time_quantum을 다 쓴 경우 queue_level을 높이고 프로세스의 tick을 초기화합니다. 그 후 현재 queue에서 제거하고 다음 queue에 삽입해줍니다.
4. time quantum을 다 쓰지 않은 경우 context switch를 진행합니다. 이때 I1의 탐색은 진행하지 않습니다.
5. I0가 비어있고 I1이 비어있지 않은 경우 I0와 동일하게 작동합니다.
6. 만약 I0와 I1 둘 다 비어있는 경우 pass가 그대로 0이기에 I2 queue를 확인하게 됩니다.
7. I2가 비어있지 않은 경우 우선순위에 따라 프로세스를 결정해야 하기에 queue_top을 통해 가장 우선순위가 높은 프로세스를 선택합니다.
8. 이때 프로세스가 time_quantum을 다 쓴 경우 priority를 1 낮추고 tick을 초기화합니다. 만약 priority가 0 미만이 된 경우 0으로 조절해줍니다.

Trap

새로운 trap은 다음과 같이 변경되었습니다.

```
//trap.c
void
usertrap(void)
{
    ...

    if(which_dev == 2) {
        if(p->queue_level != -1) {
```

```

    p->tick++;
    yield();
}
}

...
}

```

기존의 trap과 달리 timer interrupt가 발생했을 때 mode에 따라 yield가 되도록 해야합니다.

현재 실행 중인 프로세스의 queue_level이 -1(FCFS)이 아닌 경우에만 프로세스의 tick을 증가시키고, yield를 하도록 합니다.

usertrap 말고 kerneltrap에도 동일하게 구현했습니다.

trap에서 Priority Boosting 또한 구현했으나 뒤에서 설명하겠습니다.

Process Handling

프로세스들은 필요에 따라 queue에 되거나 삽입되거나 제거되어야 합니다.

Insert

위의 스케줄러는 queue에 프로세스가 잘 보관되어 있는 경우에 작동합니다.

즉 프로세스가 생겨났을 때 queue에 삽입해주는 코드가 필요합니다.

```

//proc.c
static struct proc*
allocproc(void)
{
    ...

found:
    p->pid = allocpid();
    p->state = USED;

    if(mode==1) {
        p->priority = -1;
        p->tick = -1;
        p->queue_level = -1;
        enqueue(&queues[3],p);
    }
}

```

```

    }
    else {
        p->priority = 3;
        p->tick = 0;
        p->queue_level = 0;
        enqueue(&queues[0],p);
    }

    ...
}

```

새로운 프로세스가 만들어지면 userinit 또는 fork가 호출됩니다.

두 함수 모두 프로세스를 초기화해주는 allocproc 함수를 호출합니다.

그렇기에 allocproc 함수에서 새롭게 생겨난 프로세스를 적절한 queue에 넣어주도록 구현합니다.

FCFS mode인 경우 priority, tick, queue_level을 -1로 초기화하고 FCFS queue에 넣어줍니다.

MLFQ mode인 경우 priority는 3, tick은 0, queue_level은 0으로 초기화하고 IO queue에 넣어줍니다.

Delete

프로세스가 작업을 끝냈을 때 queue에서 없애주는 코드가 필요합니다.

없애주지 않을 경우 각 queue에 필요 없는 프로세스들이 같이 보관되어 공간을 차지해버려 필요한 만큼 프로세스들을 보관할 수도 없고 관리에 어려움이 생깁니다.

```

//proc.c
void
sched(void)
{
    int intena;
    struct proc *p = myproc();
    if(!holding(&p->lock))
        panic("sched p->lock");
    if(mycpu()->noff != 1)
        panic("sched locks");
    if(p->state == RUNNING)
        panic("sched running");

```

```

if(intr_get())
    panic("sched interruptible");

int lev = p->queue_level == -1 ? 3 : p->queue_level;
if(p->state == RUNNABLE || p->state == SLEEPING || p->state == RUNNING)
    dequeue(&queues[lev],p);
    enqueue(&queues[lev],p);
}
else {
    dequeue(&queues[lev],p);
}
intena = mycpu()->intena;
swtch(&p->context, &mycpu()->context);
mycpu()->intena = intena;
}

```

sched 함수는 프로세스가 CPU를 놓았을 때 작동합니다.

그렇기에 이곳에서 프로세스가 작업을 끝내서 CPU를 놓은 것인지, 아니면 timer interrupt 나 sleep 등으로 작업이 남았지만 CPU를 놓은 것인지 판별한 후 queue에서 제거합니다.

프로세스가 RUNNABLE 또는 SLEEPING인 경우 아직 작업이 남았지만 CPU를 놓았다는 것이기에 dequeue, enqueue해주어 queue의 맨 뒤쪽으로 이동 시켜 줍니다.

프로세스가 위 두 상태가 아닌 경우 작업이 끝났다는 의미이기에 dequeue하여 queue에서 제거해줍니다.

Priority Boosting

MLFQ mode에서 글로벌 틱이 50틱마다 프로세스들의 우선순위를 높여주고 IO queue로 모든 프로세스를 이동시키는 Priority Boosting 함수가 필요합니다.

```

//proc.c
void
priority_boosting(void)
{
    if(mode==1) {
        return;
    }
    struct proc *p;
    for(p = proc; p < &proc[NPROC]; p++) {

```

```

    acquire(&p->lock);
    if(p->state != RUNNABLE && p->state != RUNNING && p->state != SLEEPING
        dequeue(&queues[p->queue_level],p);
        release(&p->lock);
        continue;
    }
    dequeue(&queues[p->queue_level],p);
    enqueue(&queues[0],p);
    p->priority = 3;
    p->tick = 0;
    p->queue_level = 0;
    release(&p->lock);
}
}

```

이 함수는 다음과 같이 작동합니다.

1. 만약 FCFS mode인 경우 함수를 종료합니다.
2. proc 배열을 탐색하면서 RUNNABLE, RUNNING, SLEEPING인 프로세스들만 원래 있던 queue에서 제거 후 IO queue에 삽입합니다.
3. priority는 3, tick은 0, queue_level은 0으로 초기화합니다.

```

//trap.c
void
clockintr()
{
    if(cpuid() == 0){
        acquire(&tickslock);
        ticks++;
        if(ticks%50 == 0) {
            priority_boosting();
        }
        wakeup(&ticks);
        release(&tickslock);
    }
    w_stimecmp(r_time() + 1000000);
}

```

50틱마다 실행되어야 하므로 trap.c에서 tick을 증가시켜주는 부분에 위와 같이 priority_boosting 함수를 추가합니다.

System calls

아래는 명세에서 요구한 system call의 구현입니다.

yield

명시적으로 CPU를 반납한 후 다음 스케줄링을 진행합니다.

```
//proc.c
void
yield(void)
{
    struct proc *p = myproc();
    acquire(&p->lock);
    p->state = RUNNABLE;
    sched();
    release(&p->lock);
}

//sysproc.c
void
sys_yield(void)
{
    yield();
}
```

getlev

현재 프로세스의 queue_level을 반환합니다.

FCFS mode에서는 99를 반환하도록 합니다.

```
//proc.c
int
getlev(void)
{
    struct proc *p;
```

```

int lev;

p = myproc();
acquire(&p->lock);
lev = p->queue_level == -1 ? 99 : p->queue_level;
release(&p->lock);

return lev;
}

//sysproc.c
uint64
sys_getlev(void)
{
    return getlev();
}

```

setpriority

프로세스의 우선순위를 설정합니다.

priority가 0보다 작거나 3보다 크면 -2를 반환합니다.

주어진 pid의 프로세스가 존재할 경우 우선순위를 변경 후 0을 반환합니다.

없을 경우 -1을 반환합니다.

```

//proc.c
int
setpriority(int pid, int priority)
{
    struct proc *p;

    if(priority < 0 || priority > 3) {
        return -2;
    }

    for(p = proc; p < &proc[NPROC]; p++) {
        if(p->pid == pid) {
            p->priority = priority;

```



```

        return 0;
    }
}
return -1;
}

//sysproc.c
uint64
sys_setpriority(void)
{
    int pid;
    int priority;

    argint(0,&pid);
    argint(1,&priority);
    if(pid < 0 || priority < 0) {
        return -1;
    }
    return setpriority(pid,priority);
}

```

mlfqmode

현재 스케줄링 모드를 MLFQ로 변경합니다.

```

//proc.c
int
mlfqmode(void)
{
    if(mode == 0) {
        printf("already in MLFQ\n");
        return -1;
    }

    mode = 0;
    ticks = 0;

    struct proc *p;

```

```

for(;queues[3].size > 0;) {
    p = queue_head(&queues[3]);
    acquire(&p->lock);
    if(p->state != RUNNABLE && p->state != RUNNING && p->state != SLEEPING)
        dequeue(&queues[3],p);
    release(&p->lock);
    continue;
}
dequeue(&queues[3],p);
enqueue(&queues[0],p);
p->priority = 3;
p->tick = 0;
p->queue_level = 0;
release(&p->lock);
}
temp_yield();
return 0;
}

```

1. 만약 이미 MLFQ mode인 경우 에러 메시지를 출력 후 -1을 반환합니다.
2. FCFS queue를 탐색하며 모든 프로세스를 확인합니다.
3. RUNNABLE, RUNNING, SLEEPING이 아닌 프로세스인 경우 FCFS queue에서 제거합니다.
4. 그렇지 않은 경우 FCFS queue에서 제거 후 IO queue에 삽입합니다.
5. priority는 3, tick은 0, queue_level은 0으로 설정합니다.
6. 그 후 스케줄링을 다시 시작하기 위해 temp_yield를 실행합니다.

```

//proc.c
void
temp_yield(void)
{
    int intena;
    struct proc *p = myproc();
    acquire(&p->lock);
    p->state = RUNNABLE;
    intena = mycpu()->intena;
}

```

```

    swtch(&p→context, &mycpu()→context);
    mycpu()→intena = intena;
    release(&p→lock);
}

```

mode를 변경한 후 변경된 모드에 맞춰 즉각 스케줄링이 되도록 해야합니다.

만약 FCFS mode로 변경되었는데 변경 직전 생성 시간이 가장 빠른 프로세스가 아닌 다른 프로세스가 실행 중이었다면 스케줄링이 의도대로 작동하지 않게 됩니다. 그렇기에 임의로 CPU를 놓게 해야 합니다.

yield 함수를 이용할 경우 sched 함수에서 queue 내부에서의 이동이 일어나 mode 변경 때 정렬된 순서가 깨져버립니다.

그렇기에 queue 내부에서의 이동이 일어나지 않도록 yield 함수와 sched 함수에서 필요한 부분만을 가져와 temp_yield 함수를 만들었습니다.

fcfsmode

현재 스케줄링 모드를 FCFS로 변경합니다.

```

//proc.c
int
fcfsmode(void)
{
    if(mode == 1) {
        printf("already in FCFS\n");
        return -1;
    }

    mode=1;
    ticks = 0;

    struct proc *p;
    int arr[NPROC];
    int i = 0;

    for(p = proc; p < &proc[NPROC]; p++) {
        acquire(&p→lock);
        if(p→state != RUNNABLE && p→state != RUNNING && p→state != SLEEPING
            dequeue(&queues[p→queue_level],p);
    }
}

```

```

        release(&p->lock);
        continue;
    }
    dequeue(&queues[p->queue_level],p);
    arr[i] = p->pid;
    i++;
    release(&p->lock);
}
int temp;
for(int j = 0; j < i; j++) {
    for(int k = 0; k < i; k++) {
        if(arr[j] < arr[k]) {
            temp = arr[j];
            arr[j] = arr[k];
            arr[k] = temp;
        }
    }
}
for(int j=0;j<i;j++) {
    for(p = proc; p < &proc[NPROC]; p++) {
        acquire(&p->lock);
        if(p->pid == arr[j]) {
            enqueue(&queues[3],p);
            p->priority = -1;
            p->tick = -1;
            p->queue_level = -1;
            release(&p->lock);
            break;
        }
        release(&p->lock);
    }
}

temp_yield();
return 0;
}

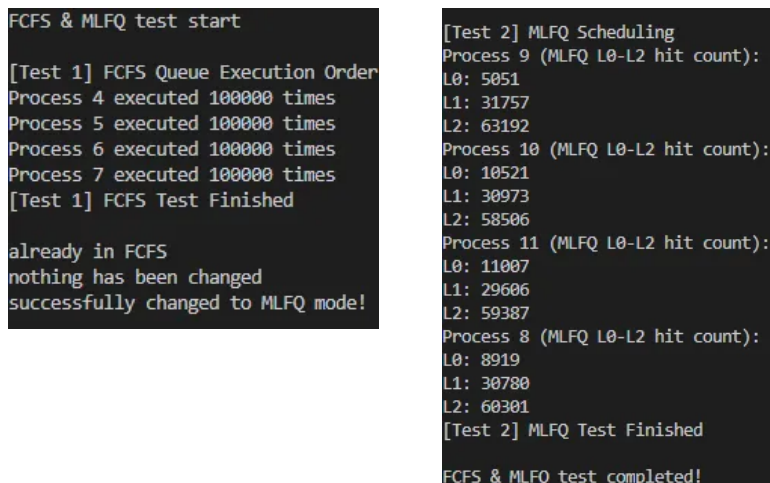
```

1. 만약 이미 FCFS mode인 경우 에러 메시지를 출력 후 -1을 반환합니다.

2. 프로세스의 생성 시간을 기준으로 정렬되어야 하므로 일단 proc 배열을 탐색합니다.
3. 프로세스가 RUNNABLE, RUNNING, SLEEPING이 아닌 경우 각 queue에서 제거합니다.
4. 맞는 경우 queue에서 제거 후 임시 pid 배열에 저장합니다.
5. 그 후 버블 정렬을 통해 생성 시간 순으로 정렬을 진행합니다.
6. 정렬된 pid 순으로 FCFS queue에 순서대로 삽입을 합니다.
7. priority, tick, queue_level을 -1로 설정합니다.
8. 마지막으로 mlfqmode에서처럼 temp_yield를 실행시켜 새롭게 스케줄링을 시작합니다.

3. Results

제공된 테스트 코드를 실행시켜 결과를 확인했습니다.



```

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

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

[Test 2] MLFQ Scheduling
Process 9 (MLFQ L0-L2 hit count):
L0: 5051
L1: 31757
L2: 63192
Process 10 (MLFQ L0-L2 hit count):
L0: 10521
L1: 30973
L2: 58506
Process 11 (MLFQ L0-L2 hit count):
L0: 11007
L1: 29606
L2: 59387
Process 8 (MLFQ L0-L2 hit count):
L0: 8919
L1: 30780
L2: 60301
[Test 2] MLFQ Test Finished

FCFS & MLFQ test completed!

```

1. FCFS mode에서는 4, 5, 6, 7 순서대로 정확히 종료된 것을 알 수 있습니다.
2. 그 후 mode 변경에서 이미 FCFS mode이기에 에러 메세지 출력 후 -1이 반환되어서 테스트 코드의 에러 메세지 또한 출력되었습니다.
3. MLFQ mode로의 변경은 성공적으로 진행되었습니다.
4. MLFQ mode에서는 모든 프로세스들이 I0~I2까지 이동하며 제대로 종료된 것을 알 수 있습니다. 각 프로세스들이 특정 queue에서 실행된 비율이 비슷하므로 제대로 실행되었음을 알 수 있습니다.

4. Troubleshooting

이번 과제는 실질적으로 코드를 작성하는 부분이 엄청 많은 것은 아니었습니다. 단지 xv6의 코드를 쭉 읽어보면서 기존의 코드가 어떻게 동작하는 지를 이해해야 했기에 이 때 시간이 많이 소요되었습니다.

또한 기존의 코드에서는 각 queue에 SLEEPING인 프로세스를 보관하지 않고 wakeup 등에서 RUNNABLE로 바뀔 때 다시 queue에 넣어주는 식으로 구현했었습니다. 하지만 관리에 불편함이 있습니다. 또 어차피 proc 배열도 SLEEPING인 프로세스를 보관하므로 동일한 사이즈의 queue에서도 SLEEPING인 프로세스를 보관하는 것이 문제가 되지 않기에 현재의 구현 방식으로 변경되었습니다.

scheduler 함수에서 l0, l1을 탐색 후 l2의 탐색을 할지 말지 결정을 원래는 goto로 하려고 했었습니다. 그런데 잘 써보지 않았던 것이라 그런지 계속해서 오류가 발생하여 지금과 같이 pass 변수를 이용하여 조절하는 방식으로 변경되었습니다.