

OS_project01_12755_2021079843

Design

CPU Scheduling이란?

| 운영체제가 CPU를 여러 프로세스 사이에서 어떻게 분배할지를 결정하는 메커니즘

여러 프로세스가 동시에 실행을 요구할 때, 어떤 프로세스를 언제 실행할지를 결정하는 것이 바로 이번 프로젝트에서 구현할 CPU Scheduler이다.

xv6에서 scheduling이 일어나는 흐름

1. 프로세스가 실행중(running state)
2. 이 프로세스가 `yield()` 를 호출해서 CPU를 양보하려고 한다
3. `yield()` 안에서 `p->state = RUNNABLE` 하고 `sched()` 를 호출한다
4. `sched()` 가 현재 프로세스의 context를 저장하고, `scheduler()` (CPU context)로 돌아간다
5. `scheduler()` 가 다른 RUNNABLE 프로세스를 찾는다
6. 그 프로세스로 `swtch()` 해서 실행한다

FCFS 구현 계획

- 이번 과제에서 요구하는 FCFS의 경우, 스케줄러가 가장 먼저 생성된 process부터 선택해야한다.
 - 여기서 `earliest creation time` 의 경우, `pid` 로 비교를 할 수 있으므로 기존의 `proc` 구조체를 수정하지 않아도 구현이 가능하였다.
 - `pid`로 비교가 가능한 이유는 xv6에서 `allocproc()` 을 하면 내부적으로 `allocopid()` 가 호출되는데, 새로운 프로세스가 생성될 때마다 `nextpid` 가 1씩 증가하고, 그 값을 프로세스의 `pid`로 받게된다. 이 말은 즉, 먼저 생성된 프로세스는 작은 `pid`를 가지게 되고, 나중에 생성된 프로세스는 더 큰 `pid`를 가지게 되므로 `pid`로 `creation time` 비교하는 방식으로 구현을 하였다.

```
int
allocpid()
{
    int pid;

    acquire(&pid_lock);
    pid = nextpid;
    nextpid = nextpid + 1;
    release(&pid_lock);

    return pid;
}
```

- FCFS `scheduler()`가 동작하는 동안에는 `priority boosting`이 동작하지 않는다.

- FCFS scheduler()는 non-preemptive로, 한 번 프로세스가 실행되면 완료되거나 자발적으로 CPU를 내놓을 때까지 실행된다.

MLFQ 구현 계획

- L0, L1, L2의 3-level feedback queue로 구성되고, level이 낮을수록 더 높은 우선순위를 갖는다.
- 각 level은 $2i+1$ 만큼의 time quantum을 가지고, 새로 생성된 프로세스는 L0에서부터 시작한다.
- L0, L1은 basic Round Robin scheduling 방식을 따르며, 상위 레벨 큐에 RUNNABLE한 프로세스가 없다면 다음 하위 레벨 큐에서 프로세스를 선택한다. 해당 level에서의 time quantum을 모두 사용하면 time quantum을 초기화하고 하위 레벨 큐로 이동한다.
- L2는 priority scheduling 방식을 따라 높은 우선순위를 가지는 프로세스부터 먼저 스케줄링된다.
 - 각 프로세스는 0~3 priority value를 가지며, 높은 숫자가 더 높은 우선순위를 갖는다.
 - 프로세스가 처음 생성되면 default priority는 3이다.
 - 동일한 우선순위를 갖는 프로세스들이 있다면 어떤 프로세스든지 선택되면 된다. 나의 경우, 동일한 priority를 가진 프로세스들 중 proc()배열에서 가장 먼저 순회된 프로세스부터 실행하도록 코드를 구현하였다.
 - L2에선 time quantum을 다 사용하게 되면 priority가 1 감소하고, time quantum이 reset된다.
- starvation을 막기 위해 global tick이 50이 될때마다 priority boosting이 일어나고, priority boosting이 일어나면 모든 프로세스들은 time quantum, priority가 reset되어 L0로 이동한다.

Mode Switch

- mode_switch라는 전역 변수를 두어 mode_switch==1이면 FCFS schedule를 실행하고, mode_switch==0이면 MLFQ scheduler를 실행하는 방식으로 구현하였다.
- 기본적으로 xv6가 처음 실행되었을 때는 FCFS 스케줄러가 실행되도록 한다.
- mode switch가 일어날때는 global tick=0으로 초기화한다.
- FCFS→ FCFS 또는 MLFQ→MLFQ로 mode switch하는 시스템 콜을 호출하면 어떤 변화도 일어나지 않고, 메시지를 출력하도록 한다.
- FCFS → MLFQ로의 mode switch가 일어날 때는 FCFS에서 대기하고 있던 모든 프로세스들이 L0 큐로 이동을 하고, priority=3, time quantum=0, level=0으로 초기화된다.
- MLFQ→FCFS로의 mode switch가 일어날 때는 level, priority, time quantum은 전부 -1로 초기화되고 오로지 creation time에 의해 스케줄링된다.

Implementation

📌 먼저 새로 선언한 구조체, 변수, 함수들에 대해 전부 설명한 후, 제일 마지막으로 `scheduler()` 함수에 설명하고자 한다.

proc.h과 proc.c에 필요한 구조체 및 전역 변수 선언

```
// proc.h
struct proc{
    ...
    int timequantum; //fcfs:-1, mlfq:0 ~ 2i+1
    int level; //fcfs:-1, mlfq:0~2
    int priority; //fcfs:-1, mlfq:0~3 - default:3
};

//proc.c
// MLFQ queue structure
struct mlfq_queue {
    int level; // Queue level (0,1,2)
    int timelimit; // Time limit for this queue
};

struct mlfq_queue queues[3]; // 3 levels for MLFQ

int mode_switch = 1; // 1: FCFS, 0: MLFQ

// Global tick counter for priority boosting
int global_tick_count = 0;
```

`proc` 구조체에는 각 레벨에서 프로세스가 사용한 `time quantum`, 현재 프로세스가 속한 `level`, 프로세스의 `priority`를 저장하는 변수를 추가해주었다.

`process`의 `time quantum`이 프로세스가 속한 `level`의 `time quantum`을 다 사용했는지 비교할 때 사용하기 위한 `mlfq_queue` 구조체를 새로 추가해주었다. L0, L1, L2 3개의 레벨 각각을 의미하는 3개의 `mlfq_queue`를 선언하였다.

`mode_switch`를 컨트롤하는 `mode_switch` 변수를 추가해주었고, 1일때는 FCFS, 0일때는 MLFQ가 실행되는 방식으로 구현하였다. xv6가 처음 켜졌을때 FCFS scheduler가 실행되어야하므로 초기값은 FCFS를 나타내는 1로 해주었다.

priority boosting을 일으키는 global tick을 count하는데 사용할 `global_tick_count` 변수도 선언해주었다.

allocproc() 수정

```
static struct proc*
allocproc(void)
{
    ...
    found:
```

```

p->pid = allocpid();
p->state = USED;
// Initialize MLFQ related fields
if(mode_switch == 1) {
    // FCFS mode
    p->level = -1;
    p->priority = -1;
    p->timequantum = -1;
} else {
    // MLFQ mode
    p->level = 0;    // Start at highest priority queue
    p->priority = 3; // Highest priority
    p->timequantum = 0;
}
...
}

```

process를 생성하는 `allocproc()` 함수에 `proc` 구조체에 새로 추가해준 변수들을 초기화해주는 코드들을 추가해주었다.

`mode_switch==1`, 즉 FCFS mode일때는 `level`, `priority`, `timequantum` 변수가 사용되지 않기 때문에 과제명세서에서 말하는 것처럼 전부 -1로 초기화해준다.

`mode_switch==0`, 즉 MLFQ mode일때는 과제명세서에서 말하는 것처럼 `level=0`, `priority=3`, `timequantum=0`으로 초기화해준다.

trap.c 수정

```

void
usertrap(void)
{
    ...
    // give up the CPU if this is a timer interrupt.
    if(which_dev == 2 && mode_switch==0)
        yield();
    ...
}

void
kerneltrap()
{
    ...
    // give up the CPU if this is a timer interrupt.
    if(which_dev == 2 && myproc() != 0 && mode_switch==0)
        yield();
    ...
}

void
clockintr()

```

```

{
    if(cpuid() == 0){
        acquire(&tickslock);
        ticks++;
        //priority boosting 관련 코드 추가
        global_tick_count++;
        if (global_tick_count >= 50) {
            global_tick_count = 0;
            priority_boost(); //여기서 boosting 수행
        }
        wakeup(&ticks);
        release(&tickslock);
    }
    ...
}

```

`trap.c` 는 모든 trap을 처리하는 파일로, timer interrupt 역시 trap에 해당한다.

`which_dev` 는 trap의 원인을 구분하기 위해 사용되는 변수인데 `which_dev = devintr()` 로 값을 할당받는다. `devintr()` 함수를 살펴보면 timer interrupt가 일어났을때 2를 반환한다.

즉, `usertrap()` 과 `kerneltrap()` 코드에서 `which_dev == 2` 가 timer interrupt 발생을 의미하는 것을 알 수 있고, fcfs mode일때는 non-preemptive이므로 timer interrupt의 영향을 받지 않아야하므로 두 함수 모두 timer interrupt를 처리해주는 부분에 `mode_switich==0` 조건을 추가해주었다. 이로써 MLFQ mode일때만 timer interrupt에 의해 CPU를 놓아줄 수 있도록 하였다.

또한, `clockintr()` 에서는 tick이 증가할 때마다 `global_tick_count` 도 증가시키고, 50이 되면 `priority_boost()` 를 실행하여 priority boosting을 진행한다.

void priority_boost(void)

```

// Handle priority boosting
void
priority_boost(void)
{
    struct proc *p;

    // Move all processes back to L0 queue
    for(p = proc; p < &proc[NPROC]; p++) {
        acquire(&p->lock);
        if(p->state != UNUSED) {
            p->level = 0;          // Move to L0
            p->timequantum = 0;    // Reset quantum (L0 : 2*0+1)
            p->priority = 3;       // Reset priority to highest
        }
        release(&p->lock);
    }
}

```

`priority_boost` 함수는 `global_tick_count` 가 50이 될때마다 실행이 된다.

priority boosting이 일어나면 모든 프로세스는 L0 큐로 이동하고, `level=0, timequantum=0, priority=3` 으로 초기화된다.

void update_proc_level(struct proc *p)

```
// Update process queue level
void
update_proc_level(struct proc *p)
{
    // If process has used its entire quantum
    if (p->timequantum >= queues[p->level].timelimit) {
        p->timequantum = 0;

        // Move to lower priority queue if not already at lowest
        if(p->level < 2) {
            p->level++;
        }
        // For L2, decrease priority if at lowest level
        else if(p->level == 2 && p->priority > 0) {
            p->priority--;
        }
    }
}
```

`update_proc_level` 함수는 MLFQ scheduler에서 사용되는 함수이다.

이 함수는 process가 해당 level에서의 time quantum을 다 사용했을 때, time quantum을 초기화하고, L0, L1에 있었다면 하위 레벨로 내려가고, L2에 있었다면 priority를 1 감소시킨다.

Required System Calls

void yield(void)

```
// Give up the CPU for one scheduling round.
void
yield(void)
{
    struct proc *p = myproc();    // 현재 CPU에서 돌아가고 있는 프로세스 p를 가져옴
    acquire(&p->lock);             // 프로세스 p의 락을 획득 (다른 CPU나 인터럽트가 접근하지 못하게 보호)
    p->state = RUNNABLE;          // 현재 프로세스를 "RUNNABLE" 상태로 바꿈 (다시 스케줄러가 실행할 수 있도록 함)
    sched();                      // 스케줄러를 호출해서 다른 프로세스에게 CPU를 넘김
    release(&p->lock);             // 락을 해제함
}
```

`yield()` 함수의 경우 기존에 만들어져있던 함수를 변형없이 그대로 사용하였다.

int getlev(void)

```
// returns the queue level to which the process belongs
int getlev(void){
    struct proc *p = myproc();
    // 현재 실행중인 프로세스가 없다면 -1 return 가정
    if(p == 0){
        return -1;
    }
    // FCFS mode이면 99 return
    if(mode_switch==1){
        return 99;
    }
    return p->level;
}
```

현재 프로세스가 속해있는 level을 return하는 함수로,
만약 현재 system이 FCFS mode이면 `mode_switch==1` 로 확인해서 `99` 를 return하고,
현재 실행중인 `프로세스가 없는 경우` 엔 -1을 return하고,
그 외에는 `현재 프로세스의 level` 을 return하도록 구현하였다.

int setpriority(int pid, int priority)

```
// sets the priority of process with the given pid
int setpriority(int pid, int priority){
    struct proc *p;

    // Check if priority is valid
    if(priority < 0 || priority > 3)
        return -2;

    // Find process with matching pid
    for(p = proc; p < &proc[NPROC]; p++) {
        acquire(&p->lock);
        if(p->pid == pid) {
            p->priority = priority;
            release(&p->lock);
            return 0; // Success
        }
        release(&p->lock);
    }

    return -1; // Process not found
}
```

주어진 pid를 가진 process에 주어진 priority를 setting하는 함수이다.
만약 성공한다면 0을 return하고, 주어진 pid를 가지는 프로세스가 존재하지 않을때는 -1을 return하고, priority 숫자가 0과 3사이의 숫자가 아닌 경우엔 -2를 return하도록 구현하였다.

int mlfqmode(void)

```
// switches the current scheduling mode from FCFS to MLFQ
int mlfqmode(void){
    if(mode_switch == 0) {
        printf("System is already in MLFQ mode\n");
        return -1;
    }

    // Switch from FCFS to MLFQ
    struct proc *p;
    for(p = proc; p < &proc[NPROC]; p++) {
        acquire(&p->lock);
        if(p->state == RUNNABLE) {
            // Initialize MLFQ parameters for each runnable process
            p->priority = 3;
            p->timequantum = 0;
            p->level = 0; // All processes start in L0 queue
        }
        release(&p->lock);
    }

    mode_switch = 0; // Change to MLFQ mode
    global_tick_count = 0; // Reset global tick count

    return 0;
}
```

FCFS에서 MLFQ로 mode switch하는 함수이다.

만약에 MLFQ에서 MLFQ로 mode switch하려고 한다면 `error message` 와 함께 `-1` 이 return되고 아무 변화도 일어나지 않는다.

FCFS에서 MLFQ로 mode switch를 한다면 FCFS에 있던 모든 process들을 L0 큐로 집어넣기 위해 `level=0, timequantum=0, priority=3` 으로 초기화한다.

또한, mode가 변경되었으므로 `mode_switch=0` (MLFQ)로 바꿔주고, `global_tick_count=0` 으로 초기화해준다.

int fcfsmode(void)

```
// switches the current scheduling mode from MLFQ to FCFS
int fcfsmode(void){
    if(mode_switch == 1){
        printf("System is already in FCFS mode\n");
        return -1;
    }

    // Switch from MLFQ to FCFS
    struct proc *p;
    for(p = proc; p < &proc[NPROC]; p++) {
```



```

    acquire(&p->lock);
    if(p->state == RUNNABLE) {
        // Reset all MLFQ parameters
        p->priority = -1;
        p->level = -1;
        p->timequantum = -1;
    }
    release(&p->lock);
}

mode_switch = 1; // Change to FCFS mode
global_tick_count = 0; // Reset global tick count

return 0;

}

```

MLFQ에서 FCFS로 mode switch하는 함수이다. 전체적인 틀은 `mlfqmode` 함수와 비슷하다.

만약에 FCFS에서 FCFS로 mode switch하려고 한다면 `error message` 와 함께 `-1` 이 return되고 아무 변화도 일어나지 않는다.

MLFQ에서 FCFS로 mode switch를 한다면 MLFQ에 있던 모든 process들의 level과 priority 상관없이 오로지 creation time(pid)으로 스케줄링을 진행할 것이기 때문에 `level=-1, timequantum=-1, priority=-1` 으로 초기화해준다.

또한, mode가 변경되었으므로 `mode_switich=1` (FCFS)로 바꿔주고, `global_tick_count=0` 으로 초기화해준다.

defs.h

```

int      mlfqmode(void);
int      fcfsmode(void);
int      setpriority(int, int);
int      getlev(void);
void     yield(void); //yield는 이미 존재했음

```

sysproc.c

```

// Project 1
void
sys_yield(void)
{
    yield();
}

uint64
sys_mlfqmode(void)
{
    return mlfqmode();
}

```

```

uint64
sys_fcfsmode(void)
{
    return fcfsmode();
}

uint64
sys_getlev(void) {
    return getlev();
}

uint64
sys_setpriority(void) {
    int pid, priority;
    argint(0, &pid);
    argint(1, &priority);
    return setpriority(pid, priority);
}

```

syscall.h

```

#define SYS_yield 22
#define SYS_getlev 23
#define SYS_setpriority 24
#define SYS_mlfqmode 25
#define SYS_fcfsmode 26

```

syscall.c

```

extern uint64 sys_yield(void);
extern uint64 sys_getlev(void);
extern uint64 sys_setpriority(void);
extern uint64 sys_mlfqmode(void);
extern uint64 sys_fcfsmode(void);

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

```

user.h

```

// Project1
void yield(void);
int mlfqmode(void);
int fcfsmode(void);

```

```
int setpriority(int, int);
int getlev(void);
```

user.pl

```
entry("yield");
entry("getlev");
entry("setpriority");
entry("mlfqmode");
entry("fcfsmode");
```

scheduler()

```
void
scheduler(void)
{
    struct proc *p;
    struct cpu *c = mycpu();

    // Initialize queues
    for(int i = 0; i < 3; i++){
        queues[i].level = i;
        queues[i].timelimit = (2*i)+1;
    }

    c->proc = 0; // 이 cpu는 아직 아무 프로세스도 실행 중이 아님
    for(;;){
        // The most recent process to run may have had interrupts
        // turned off; enable them to avoid a deadlock if all
        // processes are waiting.
        intr_on(); // 혹시 인터럽트가 꺼져 있었으면 켜줌

        // FCFS
        if(mode_switch==1){
            struct proc *target = 0;

            // int found = 0;
            for(p = proc; p < &proc[NPROC]; p++) { //모든 프로세스 배열을 돌면서
                acquire(&p->lock); // 프로세스 락을 잡고
                if(p->state == RUNNABLE){
                    if(!target || p->pid < target->pid){
                        if (target) {
                            release(&target->lock); // 이전 target의 락을 해제
                        }
                        target = p;
                    }
                    continue;
                }
            }
        }
    }
}
```

```

    }
    // target이 아닌 p는 lock을 풀어주기
    release(&p->lock);
}
if(target){
    // target 락은 이미 잡혀있는 상태
    target->state = RUNNING; // running state로 변경
    c->proc = target; // 이 CPU가 이 프로세스를 실행한다고 기록
    swtch(&c->context, &target->context); // 커널 스케줄러 context → 프로세스 context로 전환 (실
제 실행)

    // Process is done running for now.
    // It should have changed its p->state before coming back.
    // 프로세스가 돌아오면 (다시 스케줄러로 복귀)
    c->proc = 0;
    // found = 1;
    release(&target->lock);
    continue;
} else{
    // nothing to run; stop running on this core until an interrupt.
    intr_on(); // 인터럽트를 켜고
    asm volatile("wfi"); // "Wait For Interrupt" 명령어로 CPU를 일시 정지
}
}
// MLFQ
else{

    int found = 0;

    // Try to schedule processes from L0 (highest priority)
    for (int level = 0; level < 3 && !found; level++) {

        // For L0 and L1: Round-robin scheduling
        if (level < 2) {
            for(p = proc; p < &proc[NPROC]; p++) {
                acquire(&p->lock);
                if(p->state == RUNNABLE && p->level == level) {
                    // Switch to chosen process.
                    p->state = RUNNING;
                    c->proc = p;
                    swtch(&c->context, &p->context);

                    // Process is done running for now.
                    c->proc = 0;

                    // Increment timequantum used
                    p->timequantum++;
                }
            }
        }
    }
}

```

```

        // Check if process needs to move queue
        update_proc_level(p);

        found = 1;
        release(&p->lock);
        break;
    }
    release(&p->lock);
}
}
// For L2: Priority-based scheduling
else {
    struct proc *highest_priority_proc = 0;
    int highest_priority = -1;

    // Find highest priority process in L2
    for(p = proc; p < &proc[NPROC]; p++) {
        acquire(&p->lock);
        if(p->state == RUNNABLE && p->level == 2 && p->priority > highest_priority) {
            if(highest_priority_proc) {
                release(&highest_priority_proc->lock);
            }
            highest_priority_proc = p;
            highest_priority = p->priority;
        } else {
            release(&p->lock);
        }
    }
}

// Schedule the highest priority process from L2
if(highest_priority_proc) {
    p = highest_priority_proc;
    p->state = RUNNING;
    c->proc = p;
    swtch(&c->context, &p->context);

    // Process is done running for now.
    c->proc = 0;

    // Increment timequantum used
    p->timequantum++;

    // Check if process needs to move queue or decrease priority
    update_proc_level(p);

    found = 1;
    release(&p->lock);
}

```

```

    }

    // If we found a process to run, break out of level loop
    if(found) break;
}
}

}
}

```

📌 FCFS(First-Come, First-Served) Scheduler 동작과정

1. `mode_switch==1` 일때 fcfs scheduling이 진행된다.
2. CPU를 잡아서 실행할 `target` process를 하나 생성한다.
3. 이후 모든 프로세스 배열을 돌면서 프로세스 락을 잡고, 처음에 `target` 프로세스가 아직 없는 경우 또는 `target` process의 pid보다 작은 pid를 가진 프로세스의 경우 if문 안으로 들어간다. 이 과정은 모든 프로세스를 순회 하며 `RUNNABLE` 상태인 프로세스 중 가장 작은 `pid`, earliest creation time을 가지는 프로세스를 선택하는 과정이다.
4. if문 조건에 해당하지 않는 프로세스들은 바로 lock을 풀어주고, `target` process는 새로 변경될 때마다 이전 `target` process의 lock을 풀어주는 과정이 들어간다.
5. 이후 `target` 프로세스가 있는 경우, 해당 프로세스를 `running state` 로 변경하고, CPU에게 해당 프로세스를 실행한다고 전달한다.
6. `swtch(&c->context, &target->context)` 를 통해 선택된 프로세스를 실행하고, 완료되면 다시 스케줄러로 복귀한다.

📌 MLFQ(Multi-Level Feedback Queue) Scheduler 동작과정

1. `mode_switch==0` 일때는 mlfq scheduling이 진행된다.
2. `found` 는 실행할 프로세스를 찾았는지를 나타내는 변수이다. 기본적으로 `0` 으로 설정되어, 프로세스가 찾아지면 `1` 로 변경된다.
3. `highest_priority_proc` 는 가장 높은 우선순위의 프로세스를 추적하는 변수로, L2에서는 이 프로세스가 CPU를 잡아 실행된다.
4. `scheduler()` 함수 제일 초반에 `queues[0] ~ queues[2]` 까지 MLFQ 레벨 0, 1, 2 (우선순위는 0이 가장 높음) 큐를 초기화해 주었다. `timelimit`은 해당 레벨에서의 time quantum으로 $2*i+1$ 이다.
5. 큐 레벨을 `0` 부터 `2` 까지 순차적으로 확인하며, 실행할 프로세스를 찾으면 `found` 가 `1` 로 설정되면 루프를 종료한다.
 - a. 큐 레벨 0부터 순차적으로 확인하기 때문에 L0에 runnable한 프로세스가 있는지 먼저 확인하고, 없다면 L1을 확인하고, L1에도 없다면 L2로 차례대로 확인 가능하다.
6. L0와 L1의 경우 Round Robin 방식으로 스케줄링이 진행되는데, 각 레벨에 대해 `proc[]` 배열을 순차적으로 확인하면서, `RUNNABLE` 상태이고 현재 레벨에 속하는 프로세스를 찾는다.
 - a. 프로세스가 실행 가능한 상태이면 프로세스 상태를 `RUNNING` 으로 변경하고, 해당 프로세스를 `swtch()` 를 통해 실행한다.
 - b. 실행이 끝나면 다시 스케줄러로 돌아오고, 프로세스의 `timequantum` 을 증가시킨다.

- c. `update_proc_level(p)` 를 호출하여 프로세스가 해당 level에서의 time quantum을 초과했는지 확인하고, 초과했다면 큐의 레벨을 업데이트한다.
 - d. 프로세스가 실행되면 `found = 1` 로 설정하고, 더 이상 다른 프로세스를 찾지 않도록 루프를 종료한다.
7. L2의 경우 priority scheduling이 진행되는데, `p->state == RUNNABLE && p->level == 2` 이며 가장 priority가 높은 프로세스를 찾아 실행한다.
- a. 만약 더 높은 우선순위를 가진 프로세스를 찾았다면, 이전에 선택된 프로세스, `highest_priority_proc` 의 락을 해제하고 새로운 프로세스를 선택한다.
8. `highest_priority_proc` 가 있다면, 해당 프로세스를 `RUNNING` 상태로 전환하고, `swtch()` 를 통해 프로세스를 실행하고, 실행이 완료되면 `timequantum` 을 증가시키고, `update_proc_level(p)` 를 통해 해당 level에서의 time quantum을 초과했는지 확인한다.
- a. time quantum을 초과했다면 `update_proc_level(p)` 을 통해 priority가 1감소한다.

Results

FCFS

```
$ test
=== [TEST] FCFS Scheduling Test ===
Child 0 started (pid=4)
Child 1 started (pid=5)
Child 2 started (pid=6)
Child 3 started (pid=7)
Child 4 started (pid=8)
Child 0 finished (pid=4)
Child 1 finished (pid=5)
Child 2 finished (pid=6)
Child 3 finished (pid=7)
Child 4 finished (pid=8)
=== [TEST] FCFS Scheduling Test Complete ===
```

```
$ 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
```

- FCFS scheduler 실행결과와 위의 사진과 같이 earliest creation time, 즉 pid가 작은 process부터 먼저 실행이 되는 것을 볼 수 있다.
- 또한, 모든 프로세스가 본인의 일을 다 끝내고 넘어가는 non-preemptive하게 동작함을 확인할 수 있다.

MLFQ

```
[Test 2] MLFQ Scheduling
Process 9 (MLFQ L0-L2 hit count):
L0: 9413
L1: 20895
L2: 69692
Process 8 (MLFQ L0-L2 hit count):
L0: 7283
L1: 22956
L2: 69761
Process 10 (MLFQ L0-L2 hit count):
L0: 7279
L1: 28649
L2: 64072
Process 11 (MLFQ L0-L2 hit count):
L0: 10344
L1: 31257
L2: 58399
System is already in MLFQ mode
nothing has been changed
[Test 2] MLFQ Test Finished
```

```
[Test 2] MLFQ Scheduling
Process 8 (MLFQ L0-L2 hit count):
L0: 7221
L1: 23842
L2: 68937
Process 9 (MLFQ L0-L2 hit count):
L0: 10245
L1: 20810
L2: 68945
Process 10 (MLFQ L0-L2 hit count):
L0: 3452
L1: 30226
L2: 66322
Process 11 (MLFQ L0-L2 hit count):
L0: 10385
L1: 31064
L2: 58551
[Test 2] MLFQ Test Finished
```

- 모든 프로세스가 처음에 L0에서 시작하고, 이후 순차적으로 L0→L1→L2로 이동하며 잘 실행되는 것을 확인할 수 있다.
- MLFQ에선 대부분의 프로세스들이 각 레벨에서 머문 시간이 비슷하므로 해당 레벨에서의 time quantum을 다 쓰고 내려갔음을 확인할 수 있다.

Mode Switch

```
[Test 1] FCFS Test Finished

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

[Test 2] MLFQ Scheduling
[Test 2] MLFQ Test Finished

FCFS & MLFQ test completed!
System is already in MLFQ mode
nothing has been changed
successfully changed to FCFS mode!
[Test 3] FCFS Queue Execution Order
```

- 첫 번째 사진은 FCFS에서 FCFS로 mode switch를 하고, MLFQ 스케줄러로 모드 변경을 진행한 것이다.
 - FCFS→FCFS로 mode switch를 진행할 때는 error message가 제대로 뜨는 것을 확인할 수 있고, 추후 MLFQ mode로의 switch가 정상적으로 된 것 역시 확인할 수 있다.
- 두 번째 사진은 MLFQ에서 MLFQ로 mode switch를 하고, FCFS 스케줄러로 모드 변경을 진행한 것이다.
 - MLFQ→ MLFQ로 mode switch를 진행할 때는 error message가 제대로 뜨는 것을 확인할 수 있고, 추후 FCFS mode로의 switch가 정상적으로 된 것 역시 확인할 수 있다.

Troubleshooting

make qemu 실행 시 panic: release, acquire

```
xv6 kernel is booting

panic: release
panic: acquire
```

처음에 FCFS 스케줄러 구현 후, `make qemu` 명령어 실행 시 위와 같은 메시지가 뜨며 실행이 되지 않았다.

이는 acquire, release할때마다 printf로 출력하여 확인해본 결과, `scheduler()` 함수에서 `target` 을 반복마다 초기화하지 않고, 이전에 선택된 `pid=1` 인 프로세스만 반복해서 실행되면서 발생하는 문제였다.


```
for(;;){
    // The most recent process to run may have had interrupts
    // turned off; enable them to avoid a deadlock if all
    // processes are waiting.
    intr_on(); // 혹시 인터럽트가 꺼져 있었으면 켜줌

    int pid = 100000; //이전에 실행한 프로세스
    struct proc *target = 0;
```

이는 target 초기화의 위치가 잘못된 것이었으므로, 위와 같이 int pid와 target의 위치를 for문 안으로 옮김으로써 해결하였다.

```
if (target) {
    release(&target->lock); // 이전 target의 락을 해제
}
```

또한, scheduler() 코드에서 프로세스마다 lock을 잡고, 놓는 과정이 반복되는데, 처음에는 위의 코드를 넣지 않았더니, target이 새로 설정될 때 이전 target의 락이 해제 되지 않는 것 때문에 acquire 문제가 뜨는 것을 알고 해당 코드를 추가해주었다.

이는 예를 들어,

- p1 이 RUNNABLE 이고, pid 가 작아서 target = p1 으로 설정된다
- 이후 루프를 돌다가 p2 도 RUNNABLE 이고 pid < p1→pid 일 경우 target = p2 로 덮어쓴다.
- 그런데 p1 의 락을 해제하지 않았기 때문에, p1→lock 이 락을 잡은 채로 방치된다.

다음에 이 프로세스를 스케줄링하거나 접근하려고 하면 acquire() 중 패닉 발생하는 문제가 발생할 수 있었던 것이다.

priority boosting 관련 오류

```
void
scheduler(void)
{
    ...
    for(;;){
        ...
        priority_boost();
```

처음에는 scheduler() 함수가 시작될 때마다 위와 같이 priority_boost() 함수를 넣어서 호출하도록 하고, priority_boost() 함수가 호출될때마다 그 함수 안에서 global_tick_count 를 1 증가시키는 방식으로 구현을 했었는데, Wiki를 작성하다 보니 이게 잘못되었음을 알고 수정하게 되었다.

위의 코드는 global_tick_count 를 오직 scheduler() 루프가 한 바퀴 돌 때마다만 증가시키는 것으로, 이 방식은 엄밀하게는 "tick"이 아니라 "스케줄러 루프 횟수"를 기반으로 동작하는 것이었다.

따라서, tick을 정확히 추적하려면, timer interrupt를 받을 때마다 증가시켜야 정확하므로, scheduler() 루프가 한 번 돌때마다가 아니라 clockintr() 함수 내부에서 global tick을 증가시키는 방식으로 수정한 것이 현재 최종적으로 작성하여 제출한 Wiki의 코드이다.

```

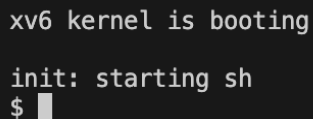
void
clockintr()
{
    if(cpuid() == 0){
        acquire(&tickslock);
        ticks++;
        //priority boosting 관련 코드 추가
        global_tick_count++;
        if (global_tick_count >= 50) {
            global_tick_count = 0;
            priority_boost(); //여기서 boosting 수행
        }
        wakeup(&ticks);
        release(&tickslock);
    }
    ...
}

```

이와 관련하여, `clockintr()` 내부에서 `priority_boost()` 함수를 사용하려고 `#include "proc.c"` 를 했더니 `trap.c` 파일 내부에 있는 다른 함수들에서 그 안에 사용된 함수를 불러오는데 오류가 떠서 결국 기존에 `proc.c`에 있던 `priority_boost()` 함수를 `trap.c` 파일로 코드를 옮겨왔다.

그랬더니 이번에는 `priority_boost()` 함수 내부에서 `proc` 배열을 순회하는데, 이때 `proc`를 인식하지 못하여서 `extern` 으로 가져오는 것으로 해결을 하였다.

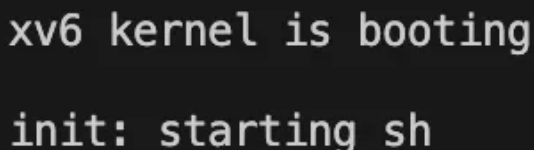
queues 배열 초과된 범위 접근



```

xv6 kernel is booting
init: starting sh
$ 

```



```

xv6 kernel is booting
init: starting sh

```

원래 `make qemu` 를 실행하면 왼쪽 사진과 같이 쉘 프롬프트가 떠야하는데, 아무리 `make clean` 을 하고 여러 방법을 시도해봐도 자꾸 오른쪽 사진처럼 `init: starting sh` 까지만 출력이 되고, 쉘 프롬프트가 뜨지 않아서 `test`를 한참 실행을 하지 못했었다.

처음에는 `size 4` 로 생성을 하고 마지막 `queues[3]` 를 FCFS 모드에서 사용하는 큐로 생각하고 코드를 구현하였다가, 중간에 `queues`는 `mlfq`에서만 사용하는 것으로 코드 방향을 수정하여 `size를 3` 바꿨었는데 `queues[3]` 에 접근하는 코드 지우는 것을 까먹었더니 위와 같은 문제가 발생하였던 것이었다.

해당 부분의 코드를 지워주니 다시 쉘 프롬프트가 잘 떠서 테스트 코드를 수행할 수 있었다.