

Operating System Project1 Wiki

12300

2018079116

정진성

Contents

- 1. Design**
- 2. Implement**
- 3. Result**
- 4. Trouble shooting**

1. Design

xv6의 기본 scheduler는 Round-Robin 방식을 사용하여 스케줄링합니다. 1tick마다 Timer Interrupt를 발생시켜 현재 실행중인 프로세스에서 scheduler로 context switching을 진행하고 scheduler에서는 다음 인덱스의 RUNNABLE 상태를 가진 프로세스를 찾고 그 프로세스로 context switching을 진행하여 모든 프로세스가 균등하게 처리될 수 있도록 합니다.

```
// Force process to give up CPU on clock tick.  
// If interrupts were on while locks held, would need to check nlock.  
if(myproc() && myproc()->state == RUNNING &&  
    tf->trapno == T_IRQ0+IRQ_TIMER)  
    yield();
```

[trap.c]

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

    // 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;
```

[proc.c]

이러한 원래의 간단한 RR 방식에서 3-Level MLFQ(Multi Level Feedback Queue) 알고리즘을 적용하려고 합니다. Level마다 time quantum이 다르고($L_n = 2n + 4$), L0L1은 RR 방식, L2에서는 우선순위를 적용합니다. 이를 위해 원래의 proc 구조체에 현재 속해 있는 queue의 레벨과, 우선순위를 저장할 변수를 추가하고 Global ticks가 100이 될 때마다 모든 프로세스의 level 및 priority, time quantum을 초기화 시켜야하기에 실행시간 및 생성된 시간을 저장할 변수를 추가합니다. 또한 user mode에서 사용할 수 있는 system call들을 추가하고, 이를 구현합니다.

2. Implement

우선 proc 구조체에 현재 level을 저장하기 위한 level, 우선순위를 저장하기 위한 priority, 실행시간을 저장하기 위한 exeticks, 프로세스가 생성된 시간을 비교하기 위한 ctime 변수를 추가하고 password 및 각 level에 맞는 time quantum을 저장합니다.

```
#define PW 2018079116      // PASSWORD
#define TQL0 4                // Level0 time quantum
#define TQL1 6                // Levell time quantum
#define TQL2 8                // Level2 time quantum

// 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[NFILE]; // Open files
    struct inode *cwd;         // Current directory
    char name[16];             // Process name (debugging)

    int level;                // MLFQ queue level
    int priority;              // MLFQ process priority
    uint exeticks;             // Store ticks
    uint ctime;                // Generated time
};
```

[proc.h]

getLevel, setPriority, schedulerLock, schedulerUnlock, yield system call을 추가하고 유저 모드에서 사용하기위해 usys.S, user.h, syscall.h 파일에 추가하고, defs.h파일에 함수의 선언을 알립니다.

```
SYSCALL(fork)
SYSCALL(exit)
SYSCALL(wait)
SYSCALL(pipe)
SYSCALL(read)
SYSCALL(write)
SYSCALL(close)
SYSCALL(kill)
SYSCALL(exec)
SYSCALL(open)
SYSCALL(mknod)
SYSCALL(unlink)
SYSCALL(fstat)
SYSCALL(link)
SYSCALL(mkdir)
SYSCALL(chdir)
SYSCALL(dup)
SYSCALL(getpid)
SYSCALL(sbrk)
SYSCALL(sleep)
SYSCALL(uptime)
SYSCALL(myfunction)
SYSCALL(yield)
SYSCALL(getLevel)
SYSCALL(setPriority)
SYSCALL([schedulerLock])
SYSCALL(schedulerUnlock)
```

```
// system calls
int fork(void);
int exit(void) __attribute__((noreturn));
int wait(void);
int pipe(int*);
int write(int, const void*, int);
int read(int, void*, int);
int close(int);
int kill(int);
int exec(char*, char**);
int open(const char*, int);
int mknod(const char*, short, short);
int unlink(const char*);
int fstat(int fd, struct stat*);
int link(const char*, const char*);
int mkdir(const char*);
int chdir(const char*);
int dup(int);
int getpid(void);
char* sbrk(int);
int sleep(int);
int uptime(void);
int myfunction(char*);
void yield(void);
int getLevel(void);
void setPriority(int pid, int priority);
void schedulerLock(int password);
void schedulerUnlock(int password);
```

[usys.S]

[user.h]

```
// System call numbers
#define SYS_fork    1
#define SYS_exit    2
#define SYS_wait    3
#define SYS_pipe    4
#define SYS_read    5
#define SYS_kill    6
#define SYS_exec    7
#define SYS_fstat   8
#define SYS_chdir   9
#define SYS_dup     10
#define SYS_getpid  11
#define SYS_sbrk    12
#define SYS_sleep   13
#define SYS_uptime  14
#define SYS_open    15
#define SYS_write   16
#define SYS_mknod   17
#define SYS_unlink  18
#define SYS_link    19
#define SYS_mkdir   20
#define SYS_close   21
#define SYS_myfunction 22
#define SYS_yield   23
#define SYS_getLevel 24
#define SYS_setPriority 25
#define SYS_schedulerLock 26
#define SYS_schedulerUnlock 27
```

[syscall.h]

```
// proj_mlfq
void           yield(void);
int            getLevel(void);
void           setPriority(int pid, int priority);
void           schedulerLock(int password);
void           schedulerUnlock(int password);
void           priorityBoosting(void);
```

[defs.h]

getLevel(void)함수는 현재 프로세스가 있는 큐의 레벨을 반환합니다.

```
// Returns the level of the queue to which the process belongs.  
int  
getLevel(void)  
{  
    return myproc()->level;  
}
```

[proc.c]

setPriority(int pid, int priority)함수는 pid와 priority를 인자로 받아 pid가 일치하는 프로세스의 우선순위를 priority 값으로 지정합니다. ptable에서 pid가 일치하는 프로세스가 있는지 검사하여 찾으면 found변수의 값을 1로, 그렇지 않으면 0으로 저장하여 찾지 못하면 찾지 못했다는 메시지를 출력합니다. 찾은 경우라도 priority의 값이 0~3의 값이 아니라면 이 또한 오류 메시지를 출력합니다. 정확한 인자들이라면 pid가 일치하는 프로세스의 priority를 지정한 priority로 바꿉니다. 같은 우선순위인지는 검사하지 않습니다.

```
// Sets the priority of the process for that pid.  
void  
setPriority(int pid, int priority)  
{  
    struct proc *p;  
    int found = 0;  
  
    acquire(&ptable.lock);  
    for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){  
        if(p->pid == pid){  
            found = 1;  
            if (priority < 0 || priority > 3)  
                cprintf("Invalid priority value: %d\n", priority);  
            else p->priority = priority;  
            break;  
        }  
    }  
    if (!found)  
        cprintf("Process with PID %d not found\n", pid);  
    release(&ptable.lock);  
}
```

[proc.c]

schedulerLock(int password) 함수는 password를 인자로 받아 현재 실행중인 프로세스를 L0로 보내고 가장먼저 스케줄링 되도록 합니다. 이미 lock이 되어있다면 함수를 나가고, 만약 password가 proc.h에서 정의한 PW(2018079116)와 일치하지 않으면 함수를 호출한 프로세스의 pid, time quantum, 현재 위치해 있는 level을 출력하고 프로세스를 종료하도록 하였습니다. password가 PW와 일치한다면 L0의 프로세스가 L0에서 L1으로 이동하지 않으며 이는 scheduler() 함수에서 구현이 되어있습니다. L0 level로 보내기 위해 p->level을 0으로 바꾸어 주고 실행시간을 초기화 시켜줍니다. 또한 L0에서 가장 먼저 스케줄링 되도록 위에서 선언한 static struct proc *scheduler_locked_proc 변수에 현재 프로세스의 주소를 넣어주고, scheduler_locked 변수에 1을 넣어 lock 상태임을 알립니다. global ticks 변수 또한 0으로 초기화 시켜줍니다. 이후 yield()를 호출해 scheduler() 함수가 작동하도록 합니다.

```
// Make sure that the process is scheduled first.
// force the process to terminate if the passwords do not match.
// prints the pid of the process, the time quantum, and the level of the currently located queue.
void
schedulerLock(int password)
{
    struct proc *p = myproc();

    // Verify password
    if (PW != password )
    {
        // Print information
        cprintf("Password does not match. pid: %d time quantum: %d Level of the currently located queue: %d\n"
        , p->pid, p->exeticks, p->level);
        exit();
    }

    if (!scheduler_locked){
        acquire(&ptable.lock);
        p->level = 0;
        p->exeticks = 0;
        scheduler_locked_proc = p;
        scheduler_unlocked_proc = 0;
        scheduler_locked = 1;
        ticks = 0;
        release(&ptable.lock);
    }

    // Reschedule
    yield();
}
```

[proc.c]

schedulerUnlock(int password) 함수는 schedulerLock이 된 상태를 해제하여 주는 함수입니다. scheduler_locked 변수를 검사하여 lock이 되어있는 상태인지 검사합니다. lock함수와 같이 password를 검사하고 일치하지 않으면 프로세스의 정보를 출력한 후 프로세스를 종료시킵니다. 이 함수를 호출한 프로세스의 level을 0으로 맞추고, priority = 3, 실행시간을 0으로 초기화 시켜줍니다. scheduler_unlocked_proc 변수에 현재 프로세스의 주소를 넣어 yield()가 호출된 이후에 스케줄링시 scheduler_unlocked_proc가 가리키는 프로세스가 가장먼저 스케줄링 되도록 합니다.

```
// release scheduler lock
void
schedulerUnlock(int password)
{
    struct proc *p = myproc();

    // Verify password
    if (PW != password )
    {
        // Print information
        cprintf("Password does not match. pid: %d time quantum: %d Level of the currently located queue: %d\n",
            p->pid, p->exeticks, p->level);
        exit();
    }

    // Check lock state
    if (scheduler_locked){
        acquire(&ptable.lock);
        p->level = 0;
        p->priority = 3;
        p->exeticks = 0;
        scheduler_locked = 0;
        scheduler_locked_proc = 0;
        scheduler_unlocked_proc = p;
        release(&ptable.lock);
    }

    // Reschedule the current process immediately
    yield();
}
```

[proc.c]

priorityboosting을 구현하여 starvation을 막습니다. boosting은 global tick이 100 ticks가 될 때마다 발생하는데 모든 프로세스들을 L0으로 이동시키며 priority를 3으로 재설정, time quantum을 초기화시킵니다. 또한 global tick이 100이 되면 schedulerLock을 풀어주어야하므로 만약 scheduler_locked 변수가 1이면 Unlock함수를 호출한 것과 같은 효과를 내기 위하여 호출한 함수를 가장먼저 스케줄링하도록 level, priority, exeticks를 초기화 시키고 scheduler_unlocked_proc = p로 가리켜 줍니다.

```
// Reset priority and time quantum for all processes
// to avoid starvation
void
priorityBoosting(void)
{
    struct proc *p;
    struct cpu *c = mycpu();

    acquire(&ptable.lock);

    if (scheduler_locked && c->proc != 0) {
        p = c->proc;
        p->level = 0;
        p->priority = 3;
        p->exeticks = 0;
        scheduler_locked = 0;
        scheduler_locked_proc = 0;
        scheduler_unlocked_proc = p;
    }

    for(p = ptable.proc; p < &ptable.proc[NPROC]; p++) {
        if(p->state != UNUSED && p->state != ZOMBIE) {
            p->level = 0;
            p->exeticks = 0;
            p->priority = 3;
        }
    }

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

[proc.c]

schedulerLock과 Unlock은 129, 130번 인터럽트 호출 시에도 실행할 수 있어야 하기 때문에 traps.h 함수에 각각의 번호를 지정해주고, trap.c 파일에 이에 대한 처리를 진행해 줍니다. 64 인터럽트를 호출할 때와 같은 로직이고, 스케줄러 락, 언락 함수 또한 system call이기 때문에 64번 인터럽트 호출시와 동일하게 처리해 줍니다.

```
// These are arbitrarily chosen, but with care not to overlap
// processor defined exceptions or interrupt vectors.
#define T_SYSCALL      64      // system call
#define T_DEFAULT      500     // catchall

#define T_IRQ0         32      // IRQ 0 corresponds to int T_IRQ

#define IRQ_TIMER      0
#define IRQ_KBD        1
#define IRQ_COM1       4
#define IRQ_IDE        14
#define IRQ_ERROR      19
#define IRQ_SPURIOUS   31

#define T_SCHEDULERLOCK 129
#define T_SCHEDULERUNLOCK 130
```

[traps.h]

```
// Interrupt 129 or 130
if(tf->trapno == T_SCHEDULERLOCK || tf->trapno == T_SCHEDULERUNLOCK){
    if(myproc()->killed)
        | exit();
    myproc()->tf = tf;
    syscall();
    if(myproc()->killed)
        | exit();
    return;
}
```

[trap.c]

allocproc 함수를 수정하여 프로세스가 생성될 때 proc 구조체에서 선언한 변수들을 초기화 시켜 주었습니다. found 레이블을 활용하여 UNUSED 상태인 프로세스가 발견이 되면 goto문으로 found로 이동하여 level, priority, exeticks를 초기상태로 만들어줍니다. ctime은 프로세스가 생성된 시간을 기록하여 FCFS 알고리즘으로 활용하기 위한 변수이므로 pid가 프로세스가 생성될 때마다 높아지므로 p->pid를 저장하여 이를 활용하려고 하였습니다.

```
//PAGEBREAK: 32
// Look in the process table for an UNUSED proc.
// If found, change state to EMBRYO and initialize
// state required to run in the kernel.
// Otherwise return 0.
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++;

    // MLFQ initialize
    p->priority = 3;
    p->level = 0;
    p->exeticks = 0;
    p->ctime = p->pid;

    release(&ptable.lock);
```

[proc.c]

1ticks 마다 Timer interrupt가 발생하며 이에 따라 ticks를 증가시켜 줍니다. 여기에서 ticks를 100으로 나눈 나머지가 0이 될때 boosting을 호출하는데 이는 global ticks가 100이 될때마다 부스팅을 해주는 것과 같은 효과를 만듭니다. 또한 인터럽트가 발생한 상태의 프로세스가 현재 실행 중이라면 exeticks를 증가시켜 프로세스의 실행시간을 증가시켜 이를 이용해 level을 이동하도록 합니다.

```
switch(tf->trapno){
    case T_IRQ0 + IRQ_TIMER:
        if(cpuid() == 0){
            acquire(&tickslock);
            ticks++;

            if (ticks % 100 == 0)
                priorityBoosting();

            // Increase the exeticks of the current process for each increase by 1 tick.
            if(myproc() && myproc()->state == RUNNING)
                myproc()->exeticks++;

            wakeup(&ticks);
            release(&tickslock);
        }
        lapiceoi();
        break;
}
```

[trap.c]

scheduler()함수에는 스케줄링 알고리즘을 구현했습니다. 스케줄러락이 설정되어있는지 확인할 수 있는 scheduler_locked 변수를 선언하고, 락이 설정되면 락을 호출한 프로세스의 주소를 저장할 수 있도록 scheduler_locked_proc 변수, 락을 해제하면 해제한 프로세스의 주소를 저장할 수 있도록 scheduler_unlocked_proc 변수를 선언해주었습니다. for문을 반복적으로 탐색해 L0->L1->L2->L0의 순서가 구현이 되도록 만들었습니다.

```
//PAGEBREAK: 42
// Per-CPU process scheduler.
// Each CPU calls scheduler() after setting itself up.
// Scheduler never returns. It loops, doing:
//   - choose a process to run
//   - swtch to start running that process
//   - eventually that process transfers control
//     via swtch back to the scheduler.
static struct proc *scheduler_locked_proc = 0;
static struct proc *scheduler_unlocked_proc = 0;
static int scheduler_locked = 0;

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

    c->proc = 0;

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

        acquire(&ptable.lock);
```

[proc.c]

전체적으로는 레벨별로 ptable을 탐색하며 실행가능한 프로세스를 찾아 context switching하여 스케줄링합니다. L0 큐에서는 p->level이 0이며 RUNNABLE 상태인 프로세스에게 스케줄러를 할당합니다. 만약 exeticks가 TQL0(4) 보다 크거나 같다면 L1 큐로 이동해야하므로 이를 검사하여 줍니다. level이 0이며 TQL0 보다 작은 프로세스가 할당이 되면 switching을 수행하여 RR방식을 구현하여 줍니다. 하지만 scheduler는 항상 처음부터 시작하지 않고 switching이 된 상태로 돌아가서 시작하므로 L0에서 L1로 바로 탐색하지 않고 L0을 한번 더 탐색하여 실행가능한 프로세스가 있는지 검사하여 L0에 실행가능한 프로세스가 하나도 없을 때만 L1으로 이동합니다. 실행가능한 프로세스가 있다면 goto L0로 L0레이블의 시작부분으로 이동하여 실행가능한 프로세스에 스케줄링합니다. 스케줄러락 함수가 실행되었는지도 검사합니다.

```
if (scheduler_locked && scheduler_locked_proc->state == RUNNABLE)
```

```
p = scheduler_locked_proc;
```

```

if (scheduler_unlocked_proc && scheduler_unlocked_proc->state == RUNNABLE) {

p = scheduler_unlocked_proc; scheduler_unlocked_proc = 0;

```

이 조건문들을 검사하면서 락이 되어있고 scheduler_locked_proc의 상태가 runnable하다면 p에 스케줄러락을 호출한 프로세스의 주소가 들어가 가장 먼저 스케줄링 되게 되며 가장 우선하여 프로세스를 완료하려고 하기 때문에 만약 L0에 A,B 프로세스가 들어있고 A가 락을 호출한 프로세스라면 ABABAB가 아닌 AAAABBB의 방식으로 A는 가장 우선하여 스케줄링 되도록 구현하였습니다. 또한 TQ을 넘어도 L1으로 이동하면 안되기 때문에 **if (p->exeticks >= TQL0 && !scheduler_locked)** 스케줄러락이 되어있는지 같이 검사하여 TQ를 넘더라도 다른 레벨로 가지 않도록 하였습니다.

```

L0:
// Check L0 queue for runnable processes
for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
    if(p->state != RUNNABLE || p->level != 0)
        continue;

    // Allows the function that called the schedulerLock(), schedulerUnlock() to be assigned first.
    if (scheduler_locked && scheduler_locked->state == RUNNABLE)
        p = scheduler_locked_proc;

    if (scheduler_unlocked_proc && scheduler_unlocked_proc->state == RUNNABLE) {
        p = scheduler_unlocked_proc;
        scheduler_unlocked_proc = 0;
    }

    if (p->exeticks >= TQL0 && !scheduler_locked) { // check if time quantum is up and scheduler_locked
        p->level++; // move to next level
        p->exeticks = 0; // reset exeticks for new level
        continue;
    }

    // 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;
    p->exeticks++;
    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;
}

// Rescan L0 for a viable process.
for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
    if(p->state != RUNNABLE || p->level != 0)
        continue;

    if (p->exeticks >= TQL0 && !scheduler_locked) { // check if time quantum is up
        p->level++; // move to next level
        p->exeticks = 0; // reset exeticks for new level
        continue;
    }

    goto L0;
}

```

[proc.c]

L1은 L0와 검사하는 level이 1이고 TQL1이 6임을 제외하고 로직은 동일합니다. 하지만 L0과 다르게 만약 L1의 프로세스가 종료되고 나오는 과정에서 락이 호출되었음을 검사하기 위해 조건문을 추가하고 goto L0로 락이나 락해제가 호출되었다면 L0로 이동하여 L0의 프로세스가 가장먼저 스케줄링되도록 하였습니다. 또한 L0에 프로세스가 있는지 L1의 프로세스를 스케줄링 한후 검사하여 항상 L0의 프로세스가 먼저 스케줄링 되도록 구현하였습니다.

```
L1:
// Check L1 queue for runnable processes
for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
    if(p->state != RUNNABLE || p->level != 1)
        continue;

    if (scheduler_locked && scheduler_locked_proc->state == RUNNABLE)
        goto L0;

    if (scheduler_unlocked_proc && scheduler_unlocked_proc->state == RUNNABLE)
        goto L0;

    if (p->exeticks >= TQL1 ) { // check if time quantum is up
        p->level++; // move to next level
        p->exeticks = 0; // reset exeticks for new level
        continue;
    }

    // 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;
    switchuvmm(p);
    p->state = RUNNING;
    p->exeticks++;
    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;

    // Rescan L0 for a viable process.
    for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
        if(p->state != RUNNABLE || p->level != 0)
            continue;

        if (p->exeticks >= TQL0 && !scheduler_locked) { // check if time quantum is up
            p->level++; // move to next level
            p->exeticks = 0; // reset exeticks for new level
            continue;
        }

        goto L0;
    }
}
```

[proc.c]

L0에서와 동일하게 스케줄링함수는 switching 이후로 돌아오기 때문에 만약 ptable의 마지막 요소를 할당한 후라면 반복문을 빠져나와도 L0의 프로세스를 탐색할 수 있도록 구현하였습니다. 또한 L0에 프로세스가 없다면 L1까지 검사하여 완벽하게 L0, L1에 프로세스가 있는지 검사할 수 있도록 만들었습니다.

```
// Rescan L0 for a viable process.
for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
    if(p->state != RUNNABLE || p->level != 0)
        continue;

    if (p->exeticks >= TQL0 && !scheduler_locked) { // check if time quantum is up
        p->level++; // move to next level
        p->exeticks = 0; // reset exeticks for new level
        continue;
    }

    goto L0;
}

// Rescan L1 for a viable process.
for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
    if(p->state != RUNNABLE || p->level != 1)
        continue;

    if (p->exeticks >= TQL1 ) { // check if time quantum is up
        p->level++; // move to next level
        p->exeticks = 0; // reset exeticks for new level
        continue;
    }

    goto L1;
}
```

[proc.c]

L2는 L0, L1과 다르게 우선순위를 적용하여 스케줄링합니다. L2에서는 프로세스를 찾는 중에 스위칭이 발생하지 않기 때문에 항상 처음부터 끝까지 탐색합니다. 레벨이 2인 프로세스를 탐색하면서 exeticks가 TQL2 보다 크거나 같다면 레벨을 변경하지 않고 priority를 1낮추고 exeticks를 초기화합니다. 가장높은 우선순위값과 그 프로세스의 주소를 저장하여 모두 탐색했을 때 우선순위가 가장낮은 프로세스를 할당해 줍니다. 우선순위가 같다면 ctime을 비교하여 먼저 생성된 프로세스가 먼저 할당되도록 (FCFS) 구현해줍니다. chosen_proc가 없다면 L0을 탐색하는 코드로 다시 이동합니다.

```
// Check L2 queue for runnable processes

struct proc *chosen_proc = 0;
int highest_priority = 4;
for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
    if(p->state != RUNNABLE || p->level != 2)
        continue;

    if (p->exeticks >= TQL2 ) { // check if time quantum is up
        if (p->priority > 0)
            p->priority--; // decrease priority
        p->exeticks = 0; // reset exeticks for new level
        continue;
    }

    if (p->priority < highest_priority) { // found a higher priority process
        highest_priority = p->priority;
        chosen_proc = p;
    } else if (p->priority == highest_priority) { // found a process with same priority
        if (!chosen_proc || p->ctime < chosen_proc->ctime) { // choose one with lower generated time
            chosen_proc = p;
        }
    }
}

// Switch to chosen process. It is the process's job
// to release ptable.lock and then reacquire it
// before jumping back to us.
if (chosen_proc) {
    c->proc = chosen_proc;
    switchuvvm(chosen_proc);
    chosen_proc->state = RUNNING;
    chosen_proc->exeticks++;
    swtch(&(c->scheduler), chosen_proc->context);
    switchkvm();

    // Process is done running for now.
    // It should have changed its p->state before coming back.
    c->proc = 0;
}

release(&ptable.lock);
```

[proc.c]

3. Result

컴파일과정은 이전과 동일합니다. 하지만 편하게 명령어를 입력하기 위해 shell script를 작성했기 때문에 sh makemake.sh, sh start.sh 를 입력하면 편하게 컴파일 및 실행할 수 있습니다.

```
1 #!/bin/sh
2
3 make clean
4 make
5 make fs.img
```

[makemake.sh]

```
1 #!/bin/sh
2
3 qemu-system-i386 -nographic -serial mon:stdio -hdb fs.img xv6.img -smp 1 -m 512
```

[start.sh]

기본적인 mlfq_test의 결과는 이렇습니다. 보통 L2에서 많은 시간을 보냅니다. 이를 통해 getLevel() 함수가 잘 동작함을 알 수 있습니다.

```
$ mlfq_test
MLFQ test start
[Test 1] default
Process 24
L0: 2967
L1: 5327
L2: 91706
L3: 0
L4: 0
Process 25
L0: 6245
L1: 9694
L2: 84061
L3: 0
L4: 0
Process 26
L0: 6386
L1: 9662
L2: 83952
L3: 0
L4: 0
Process 27
L0: 9505
L1: 14431
L2: 76064
L3: 0
L4: 0
[Test 1] finished
done
```

스케줄러락이 잘 작동하는지 테스트 해보기 위해 주어진 테스트 코드를 변경하였습니다. Lock을 걸고 어느 레벨에서 많은 시간을 보내는지 테스트해보려고 합니다.

```
printf(1, "MLFQ test start\n");

printf(1, "[Test 2] schedulerlockunlock\n");
pid = fork_children();

if (pid != parent)
{
    printf(1, "Process %d\n", pid);
    if (pid == 4){
        printf(1, "Lock()\n");
        schedulerLock(2018079116);

        for (i = 0; i < NUM_LOOP; i++) {
            int x = getLevel();
            if (x < 0 || x > 4){
                printf(1, "Wrong level: %d\n", x);
                exit();
            }
            count[x]++;
        }
        for (i = 0; i < MAX_LEVEL; i++)
            printf(1, "L%d: %d\n", i, count[i]);
    }
}
```

락을 실행하면 최대한 우선적으로 스케줄링하기 위해 L0로 실행한 프로세스를 보내고 가장 처음에 실행한 프로세스이기 때문에 기대한 대로 결과가 나옵니다.

```
$ mlfq_test3
MLFQ test start
[Test 2] schedulerlockunlock
Process 4
Lock()
L0: 100000
L1: 0
L2: 0
L3: 0
L4: 0
```

또한 비밀번호를 틀렸을 때의 결과를 확인해 보려고 합니다.

```
if (pid == 7) {
    printf(1,"passwordcheck");
    schedulerLock(20180);

    for (i = 0; i < NUM_LOOP; i++)
{
    int x = getLevel();
    if (x < 0 || x > 4)
    {
        printf(1, "Wrong level: %d\n", x);
        exit();
    }
    count[x]++;
}
for (i = 0; i < MAX_LEVEL; i++)
    printf(1, "L%d: %d\n", i, count[i]);
}
```

비밀번호가 맞지 않다면 명세에 적힌대로 출력해주고 프로세스를 종료함을 확인할 수 있었습니다.

```
Process 7
passwordcheckPassword does not match. pid: 7 time quantum: 1 Level of the currently located queue: 0
```

전체 테스트 결과입니다. 프로세스6은 아예 출력하는 코드가 없었기 때문에 없고, 프로세스 7이 먼저 나온 이유는 스케줄링 하며 실행을 마치기 전에 프로세스가 종료되었기 때문입니다. 마지막에 나오는 레벨에 따른 count가 누구의 것인지 확인하기 위해 5555를 출력하도록 하였습니다.

```
$ mlfq_test3
MLFQ test start
[Test 2] schdulerlockunlock
Process 4
Lock()
L0: 100000
L1: 0
L2: 0
L3: 0
L4: 0
Process 5
Lock()
unlock
unlock()
Process 6
Process 7
passwordcheckPassword does not match. pid: 7 time quantum: 1 Level of the currently located queue: 0
L0: 5016
5555
L1: 6720
5555
L2: 88264
5555
L3: 0
5555
L4: 0
5555
[Test 2] finished
done
$
```

priority boosting 함수를 테스트 하기위해 잠시 부스팅 함수가 실행되면 프로세스들의 pid, level, timequantum을 출력하게 하였습니다.

```
// Reset priority and time quantum for all processes
// to avoid starvation
void
priorityBoosting(void)
{
    struct proc *p;
    struct cpu *c = mycpu();
    cprintf("boosting\n");

    acquire(&ptable.lock);

    if (scheduler_locked && c->proc != 0) {
        p = c->proc;
        p->level = 0;
        p->priority = 3;
        p->exeticks = 0;
        scheduler_locked = 0;
        scheduler_locked_proc = 0;
        scheduler_unlocked_proc = p;
    }

    for(p = ptable.proc; p < &ptable.proc[NPROC]; p++) {
        if(p->state != UNUSED && p->state != ZOMBIE) {
            p->level = 0;
            p->exeticks = 0;
            p->priority = 3;
            cprintf("pid: %d time quantum: %d Level of the currently located queue: %d\n",
                    p->pid, p->exeticks, p->level);
        }
    }

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

xv6의 기본 테스트인 usertests를 실행하고 결과를 확인하였고, 정상 작동함을 확인할 수 있었습니다.

```
mem test
boosting
pid: 1 time quantum: 0 Level of the currently located queue: 0
pid: 2 time quantum: 0 Level of the currently located queue: 0
pid: 4 time quantum: 0 Level of the currently located queue: 0
pid: 426 time quantum: 0 Level of the currently located queue: 0
boosting
pid: 1 time quantum: 0 Level of the currently located queue: 0
pid: 2 time quantum: 0 Level of the currently located queue: 0
pid: 4 time quantum: 0 Level of the currently located queue: 0
pid: 426 time quantum: 0 Level of the currently located queue: 0
allocuvm out of memory
boosting
pid: 1 time quantum: 0 Level of the currently located queue: 0
pid: 2 time quantum: 0 Level of the currently located queue: 0
pid: 4 time quantum: 0 Level of the currently located queue: 0
pid: 426 time quantum: 0 Level of the currently located queue: 0
mem ok
pipe1 ok
preempt: kill... wait... preempt ok
exitwait ok
rmdot test
rmdot ok
fourteen test
fourteen ok
bigfile test
bigfile test ok
subdir test
subdir ok
linktest
linktest ok
unlinkread test
unlinkread ok
dir vs file
dir vs file OK
empty file name
empty file name OK
fork test
fork test OK
bigdir test
boosting
pid: 1 time quantum: 0 Level of the currently located queue: 0
pid: 2 time quantum: 0 Level of the currently located queue: 0
pid: 4 time quantum: 0 Level of the currently located queue: 0
```

4. Trouble shooting

스케줄러 함수가 항상 처음부터 실행하고 끝나는 것이 아니기 때문에 많은 고민을 하게 되었습니다. 어느 부분에서 스위칭 되더라도 현재 상태를 확인해 그에 맞는 행동을 취해야 하기 때문입니다. 그래서 레이블과 goto, L2는 한번에 탐색하고 끝나는 방식. 이 세개를 이용하여 완벽하게 상태를 확인하도록 하였습니다.

Priority Boosting을 구현하려 할때 스케줄러 락이 활성화 된 상태라면 Unlock하도록 명세에 맞게 구현하려고 하였습니다. lock상태를 검사하는 if문을 넣고 그 상태에서 schedulerUnlock()함수를 호출하였는데, 테스트 해본 결과 오류가 발생하여 난항을 겪었습니다. 해결방법은 그냥 계속 확인하는 것이었습니다. acquire(&ptable.lock);을 호출한 상태에서 한번더 acquire(&ptable.lock);를 호출하였기 때문입니다. 따라서 unlock함수를 호출하지 않고 unlock함수의 내용을 boosting함수에 구현하여 해결하였습니다.

또한 boosting 함수를 테스트하는 것에 있어서 100tick마다 일어나기 때문에 boosting함수가 실행될때마다 프로세스들의 pid, level, timequantum을 출력하게 했는데, 매우 빠른 속도이기에 확인하기 힘들었습니다. 그래서 잠시만 500tick으로 바꾸고 결과를 확인할 수 있었습니다.

```
switch(tf->trapno){
    case T_IRQ0 + IRQ_TIMER:
        if(cpuid() == 0){
            acquire(&tickslock);
            ticks++;

            if (ticks % 500 == 0)
                priorityBoosting();

            // Increase the exeticks of the current process for each increase by 1 tick.
            if(myproc() && myproc()->state == RUNNING)
                myproc()->exeticks++;

            wakeup(&ticks);
            release(&tickslock);
        }
        lapiceoi();
        break;
}
```

[trap.c]