

[ELE3021] Operating System
Project #1 Wiki
2019014266 Lim Kyu Min

Contents

I. Introduction

Project #1에 대한 개요이며, 사용된 shell script와 Makefile 수정사항에 대해 서술합니다.

II. MLFQ Design

명세에 나와있는 MLFQ Design을 어떻게 구현했는지 서술합니다.

III. MLFQ Scheduler Implementation

MLFQ Scheduling을 하는 Scheduler의 구현에 대한 자세한 내용을 서술합니다.

IV. Implemented System Calls & Function Calls

명세에 나와있는 필수 구현 system call과
MLFQ Scheduling에 사용되는 function call에 대한 내용을 서술합니다

V. MLFQ Scheduling Workflow & Results

MLFQ Scheduling의 자세한 Workflow를 정리하여 보여주고,
Scheduling, System call, Test case의 결과에 대해 서술합니다.

VI. Trouble Shooting & Dissatisfaction

해당 과제를 구현하며 마주한 문제들과 그 문제들을 어떻게 해결하였는지 서술합니다.

* 코드에서 /* 로 시작하는 주석은 해당 프로젝트 구현 도중 제가 단 주석이며, 이는 코드 구현의 의도와 내용을 이해하는 데에 도움을 줄 것입니다.

* 이하 내용에서도 서술하겠지만, MLFQ의 구현 도중 proc.c와 trap.c를 proc_mlfq.c,
trap_mlfq.c라는 파일로 복사한 후에 작업을 진행하였으며, 이는 Makefile에서도 변경 내용이 적용되어 있습니다.

* MLFQ Scheduling의 구현에 대한 자세한 설명과 사진을 함께 담아 Wiki의 분량이 많습니다. (총 29페이지) 제가 굉장히 열심히 한 과제라 열정이 과도하게 사용된 것 같습니다. 제 MLFQ Scheduler를 자세히 설명해 드리고 싶은 마음에 분량 조절을 실패한 것이니, 너그러운 마음으로 봐주시면 감사하겠습니다! ☺

I. Introduction

제가 구현한 MLFQ Scheduler는 많은 양의 Function Call과 System Call을 사용합니다. 기존 xv6와 MLFQ 버전을 구분하기 위해, 다음과 같은 상황에서 작업을 진행했습니다.

- 기존의 proc.c, trap.c를 복사하여 proc_mlfq.c, trap_mlfq.c라는 동일한 내용의 파일을 만들고, 해당 파일에서 작업을 진행했습니다.
- mlfqsyscall.c라는 파일을 만들어 명세에 나온 내용을 포함한 유저 프로그램에도 사용되어야 하는 system call들의 wrapper function을 정의했습니다. (e.g. sys_yield, sys_getLevel, etc)
- make, make fs.img를 동시에 실행해 주는 build.sh라는 스크립트를 만들었습니다. 해당 파일을 실행함으로서 빌드를 진행할 수 있습니다.

```
#!/bin/bash

set -e #exit when command fails
make
make fs.img
```

Build.sh

- Makefile의 OBJS는 다음과 같은 부분이 수정되었습니다.
 - (proc.o -> proc_mlfq.o, trap.o -> trap_mlfq.o, +mlfqsyscall.o)

```
OBJS = \
    bio.o \
    console.o \
    exec.o \
    file.o \
    fs.o \
    ide.o \
    ioapic.o \
    kalloc.o \
    kbd.o \
    lapic.o \
    log.o \
    main.o \
    mp.o \
    picirq.o \
    pipe.o \
    proc_mlfq.o \
    sleeplock.o \
    spinlock.o \
    string.o \
    switch.o \
    syscall.o \
    sysfile.o \
    sysproc.o \
    trapasm.o \
    trap_mlfq.o \
    uart.o \
    vectors.o \
    vm.o \
    prac_syscall.o \
    mlfqsyscall.o
```

Makefile OBJS

II. MLFQ Design

명세에서 명시된 MLFQ의 조건을 다음과 같이 구현하였습니다.

1. MLFQ는 3-level feedback queue로 이루어져 있습니다.

```
#define MLFQ_LEV 3 /* Define MLFQ Total Level: Used to indicate number of process in each level.
struct proc *L[MLFQ_LEV][NPROC] = {0}; /* Initialize as 0 (NULL)
/** L[0] => L0: Most prioritized process queue, RR, TQ: 4 ticks
/** L[1] => L1: process queue, RR, TQ: 6 ticks
/** L[2] => L2: process queue, Priority Scheduling based on proc()->priority, FCFS for same priority, TQ: 8 tick;

uint arrived = 0; /* arrived - assign current value to process demoted to L2. The value will be increased proportional to number of process demoted to L2.
int lidx[2] = {0, 0}; /* Index of each Queue - index for L0, L1 needs to be memorized.
static struct proc *initproc;

struct proc *lockedproc = 0; /* locked process will be located in here. Initialized as 0 (NULL)
```

Implementation of 3-level feedback queue (Proc_mlfq.c)

- xv6가 진행하는 여러 system call (e.g. yield, exit, kill, etc.)은 ptable 내부의 proc[NPROC]에서 진행을 합니다.
 - struct proc *L[MLFQ_LEV][NPROC]:
 - 프로세스의 scheduling은 전적으로 L[MLFQ_LEV][NPROC]에서 이루어집니다. L은 L[0] L[1] L[2]가 각각 NPROC 만큼의 element를 가지고 있는 2D-Array의 구조를 가지고 있습니다.
 - 모든 L의 element는 기본적으로 0 (NULL)로 초기화를 진행하였습니다. 이하 코드에서는 0으로의 접근은 허용되지 않으며, 오직 해당 queue의 element가 비어있는지 확인하는 용도로 사용됩니다.
 - ◆ 만약 0으로의 포인터 접근을 시도하면 해당 접근은 오류로 판단, panic을 일으켜 커널 데이터가 망가지지 않도록 합니다.
 - MLFQ를 구현한 L은 ptable의 각 process들의 포인터를 참조하여, scheduler에서는 참조된 포인터를 활용하여 scheduling, context switching 등의 작업을 진행합니다.
 - unit arrived:
 - 해당 값은 L2에 오는 프로세스에게 할당이 된 후 1씩 증가하는 counter의 성격을 가진 변수입니다. 이는 L2에서 같은 priority일 시 FCFS policy를 적용하기 위함이며, priority boosting이 일어나면 0으로 초기화가 됩니다.
 - int lidx[2]:
 - L0, L1에서 마지막으로 scheduling된 위치 정보를 저장합니다. 해당 위치를 기억함으로써 L0, L1에서 진행되는 Round-Robin의 scheduling의 순서를 보장할 수 있도록 MLFQ를 구현했습니다.
 - struct proc *lockedproc:
 - schedulerLock() system call을 호출한 프로세스에 대한 정보를 저장하는 포인터 변수입니다. 이후 scheduler의 workflow를 결정하는 요인 중 하나로 사용됩니다.
2. L0, L1은 Round-Robin 정책을 따르고, L2는 priority scheduling을 하며, 같은 priority에 대해서는 FCFS로 scheduling을 진행합니다. (schedulerLock이 되지 않았을 시에만 작동합니다.)
 3. Queue의 우선순위는 L0가 가장 높으며, L0에 scheduling 대상이 없을 시 L1을, L1에 대상이 없을 시 L2를 scheduling 합니다.
 - 해당 내용은 길고 자세한 설명이 필요해 III. MLFQ Scheduler Implementation에 따로 서술했습니다. II.MLFQ Design의 내용을 모두 보신 후에 scheduler에 대한 설명을 보시는 것을 추천드립니다.

4. 각 Queue는 독자적인 Time Quantum ($2n + 4$)이 존재하며, 해당 Time Quantum 안에 프로세스가 끝나지 않으면 우선순위가 낮은 Queue의 scheduling이 됩니다.
5. L2에서는 Time Quantum이 모두 소진될 시, 해당 프로세스의 priority를 높여줍니다. Priority는 최대 0까지 높아질 수 있습니다.

```
if(myproc() && myproc()->state == RUNNING &&
   tf->trapno == T_IRQ0+IRQ_TIMER)
{
    (myproc()->td)--; /* Increase process tick.
    if(myproc()->tq <= 0) /*Time Quantum Expired
    { /* If allowed time quantum elapsed,
       /* cprintf("Time Quantum for process %d had been expired!!(level %d, %d ticks)\n",myproc()->pid, myproc()->level, rettq(myproc()));

       /*L2: Time Quantum Expired -> increase current priority;
       if(myproc()->level == 2)
       {
           incpriority();
       }
       else if(myproc()->level == 1 || myproc()->level == 0) /* Time Quantum Expired -> demote current process.
       {
           demoteproc(); /* Definition: proc_mlfq.c
       }
       else /* ERROR
       {
           panic("Time Quantum!\n");
       }
       //yield();
    }
    yield(); /* Changed yield code due to Piazza implementation direction https://piazza.com/class/lf0nppamy5p2hi/post/58
}
```

Timer Interrupt (trap_mlfq.c)

- proc.h에 존재하는 프로세스 struct에 각 프로세스에 할당된 time quantum을 나타내는 int tq를 정의했습니다. Global tick이 1씩 증가할 때마다 해당 tq는 1씩 감소하며, 0이 되는 순간 해당 프로세스는 time quantum을 다 사용했다고 판단을 합니다.
- 또한 해당 struct에 해당 프로세스가 위치한 레벨을 나타내는 int level을 정의했습니다. 해당 level이 0 혹은 1일 시 (L[0], L[1]) 프로세스를 하위 Queue로 내리는 demoteproc()이 호출되며, level이 2일 시 (L[2]) priority를 올리는 incpriority()를 호출합니다.
- 이외의 경우는 에러로 판단, panic을 호출합니다.
- 처음에는 프로세스에 주어진 time quantum을 다 쓴 후에 yield()를 실행하도록 구현했는데, Piazza에 내용을 보고 매 tick마다 yield()를 호출하도록 변경했습니다. 이제 매 tick마다 context switching이 일어납니다.
- proc.h에 관한 내용은 III. MLFQ Scheduler Implementation에 자세히 서술하겠습니다.
- demoteproc(), incpriority()는 IV. Implemented System Calls & Function Calls에 더욱 자세히 서술되어 있습니다.

6. 프로세스의 Priority 초기값은 3이며, 생성된 후 L0에 들어갑니다.

```
found:
int idx = 0; /* Used to search unused space at queue L0.

p->state = EMBRYO;
p->pid = nextpid++;
p->level = 0; /* Every new process are allocated at level 0.
p->priority = 3; /* Default priority will be 3.
p->arrived = 0; /* Default arrived value will be 0.
p->lock = UNLOCKED; /* Default lock state will be UNLOCKED.

for(idx = 0; idx < NPROC; idx++){
    if(L[0][idx] == 0) { /* If there is no assigned process for current index,
        //cprintf("%s\n", L[0][idx]->state);

        L[0][idx] = p; /* Assign current process to index - Scheduled in L0 initially.
        p->idx = idx; /* Assign Index - current position
        p->tq = 4; /* Assign Time Quantum - Level 0 - 4 ticks
        //cprintf("Allocated Process: %s // PID: %d, Allocated in L0[%d]\n", p->name, p->pid, idx); /* Debug: Comment this line if it is not required.
        break;
    }
}

static struct proc* allocproc(void) - After label found: (proc_mlfq.c)
```

- 프로세스를 새로 만드는 함수 allocproc() 중, found 이후에 코드를 추가했습니다.
- level, priority, arrived 등 MLFQ scheduling에 필요한 proc의 변수들을 모두 초기화를 해주고, 이때 명세에 맞게 Priority는 3으로, level은 0으로 초기화합니다.

- 이후 해당 프로세스를 L[0]의 처음 발견되는 빈자리에 배정을 합니다. 배정된 후 배정된 자리인 index를 프로세스의 idx 변수에 넣어주며, tq 또한 L0의 time quantum인 4로 초기화를 해줍니다.

7. Global Tick이 100이 될 때마다 Priority Boosting이 일어나며, 모든 프로세스의 time quantum, priority는 초기화되고, L0으로 재조정됩니다.

```
case T_IRQ0 + IRQ_TIMER:
    if(cpuid() == 0){
        acquire(&tickslock);
        ticks++;
        /* Apply Priority Boosting based on global ticks.
        if(ticks % 100 == 0) /* For each 100 global ticks,
        {
            /* If there is a lock, nullify it.
            nullifylock();
            /* do priority boosting.
            boostpriority(); /* defined in proc_mlfq.c
        }
        wakeup(&ticks);
        release(&tickslock);
    }
    lapiceoi();
break;
```

Interrupt – case for timer interrupt (trap_mlfq.c)

- 제 MLFQ는 기존의 ticks를 global tick과 같이 사용하고 있습니다. 해당 tick이 100의 배수가 될 때마다 (즉, 100이 되면) schedulerLock()을 호출한 프로세스를 다시 MLFQ에 넣어주는 nullifylock(), Priority Boosting을 해 starvation을 막아주는 boostpriority()가 호출됩니다.
- nullifylock(), boostpriority()는 IV. Implemented System Calls & Function Calls에 자세히 서술하겠습니다.

8. SchedulerLock, SchedulerUnlock과 같은 필수적으로 구현이 되어야 하는 System call이 존재합니다.
- yield, getLevel, setPriority, schedulerLock, schedulerUnlock이 이에 해당합니다. 이에 대한 설명은 IV. Implemented System Calls & Function Calls에 자세히 서술하겠습니다.

III. MLFQ Scheduler Implementation

Scheduler에 대한 설명입니다. Proc.h와 proc_mlfq.c 내부의 scheduler가 어떻게 동작을 하는지 설명을 합니다.

1. proc.h

- proc.h에 정의된 프로세스의 정의 중 MLFQ를 위한 변경점을 서술합니다.

```
/* New Process Structure: Process dedicated for MLFQ scheduling
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;                    // Queue Level: Show Current Process Level ( 0 ~ 2 );
    int idx;                      // Index: Indicates its index in queue (L0 ~ L2);
    int priority;                 // Priority: Used for priority scheduling in L2 <- Higher priority in minimal number.
    int tq;                       // Time Quantum: tq for each process.
    enum lockstate lock;          // Lock: check if current process calls schedulerLock / schedulerUnlock
    uint arrived;                 // Arrived: arrived order of process. Value will be assigned if it comes to L2.
};
```

struct proc (proc.h)

기존 proc에 추가로 다음과 같은 변수들을 새롭게 정의했습니다.

- int level:
 - MLFQ에서 해당 프로세스가 어떤 queue에 있는지를 나타냅니다. 해당 프로세스가 있는 Queue Level에 따라 0, 1, 2의 값 중 하나를 갖습니다.
- int idx:
 - MLFQ에서 해당 프로세스가 queue의 어느 위치 (index)에 있는지 나타냅니다. 값은 $0 \leq idx \leq NPROC$ (64)입니다.
 - 만약 프로세스가 L1의 3번째에 위치해 있으면, level은 1, idx는 3을 갖게 됩니다.
- int priority:
 - 해당 프로세스의 우선순위를 나타내는 값입니다. 0~3 사이의 값을 가지며, 0이 제일 높은 priority이고, 3을 기본값을 가집니다. 해당 값은 L2에서 priority scheduling을 할 시 사용이 됩니다.
- int tq:
 - 해당 프로세스에 할당된 time quantum입니다. 해당 프로세스가 위치해있는 Queue의 level에 따라 4, 6, 8만큼의 값($2n + 4, n == level$)으로 초기화가 되며, 매 tick마다 1씩 감소합니다. Tq 값이 0이 되면 해당 프로세스는 하위 레벨로, 혹은 우선순위 (priority)가 재조정되며, 재조정 후 새롭게 바뀐 위치와 레벨에 따라 새로운 time quantum을 할당받습니다.
- enum lockstate lock:
 - lockstate는 {UNLOCKED, LOCKED} 두개로 이루어져 있는 enumeration이며, 해당 state에 따라 해당 프로세스가 schedulerLock을 호출한 프로세스인지 아닌지를 판단 합니다. Exit()을 호출한 프로세스가 해당 필드에 LOCKED 값을 가지고 있으면 적절 한 조치 후 exit()을 진행 등의 작업에 사용됩니다.
- uint arrived:
 - MLFQ에서 L2에 왔을 때 순차적으로 부여되는 값입니다. L2에 먼저 도착한 프로세스가 더욱 낮은 값을 가지고 있으며, 이는 L2에서 priority scheduling 도중 같은

priority의 경우 FCFS를 적용하기 위해 사용되는 값입니다.

2. Scheduler

- proc_mlfq.c에 정의된 void scheduler(void) 함수의 작동 방식에 대해 자세히 서술합니다.
함수의 길이가 긴 관계로 5부분으로 나누어 설명을 진행하겠습니다.

Workflow A. Scheduler – Entry Phase

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

    c->proc = 0;

    for(;;){
        // Enable interrupts on this processor.
        sti();
        // Loop over process table looking for process to run.
        acquire(&ptable.lock);
```

Scheduler Function [1/5], Entry phase (proc_mlfq.c)

다음은 Scheduler의 도입 부분입니다. For loop에서 정해진 로직과 workflow에 따라 다음 프로세스를 scheduling을 합니다. Scheduler가 호출될 시 가장 먼저 실행되는 Entry Phase에 대한 코드로, 기존 xv6와 동일하게 작동합니다.

Workflow B. Scheduler – Case #1) Scheduler Locked.

```
/* If scheduler is locked, MLFQ will not be in service.
if(lockedproc != 0)
LOCKED:
{ /* SCHEDULER LOCKED!
   /* Check If current process is running state
   if(lockedproc->state != RUNNING)
   { /* If is not running but runnable, make it run again, schedule locked process.
      if(lockedproc->state == RUNNABLE)
      {
          lockedproc->state = RUNNING;
      }
      else /* for SLEEPING and ZOMBIE, nullify current lock (unlock), and go to normal scheduler.
      {
          nullifylock();
          goto SCHEDULER;
      }
   }
   /* Context Switching
   /* It could be a overhead (same process switching)
   /* but it is not preferred to change yield() and sched() function IOT resolve current overhead.
   p = lockedproc;
   c->proc = p;
   switchuvvm(p);
   p->state = RUNNING;

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

   c->proc = 0;
}
```

Scheduler Function [2/5], Workflow for locked scheduler (proc_mlfq.c)

다음은 scheduler의 workflow 중, 어떤 프로세스의 schedulerLock() system call 호출로 인해 scheduler가 lock 되었을 시 작동하는 workflow입니다. 상세 Workflow는 다음과 같습니다.

1. if(lockedproc != 0)
 - 만약 lockedproc이 0이 아니면 (즉, schedulerLock()을 호출한 프로세스가 존재한다면) 해당 workflow로 들어옵니다.
2. if(lockedproc -> state != RUNNING)
 - 만약 schedulerLock()을 호출한 프로세스가 존재하지만, 해당 프로세스가 RUNNING 상태가 아닐 때, 다음과 같은 분기점을 가집니다.
 - A. 만약 해당 프로세스가 RUNNABLE한 경우, 해당 프로세스를 RUNNING으로 바꿉니다. Timer interrupt에 의해 yield된 경우, 다시 runnable로 바꾸어주어 scheduling을 해줍니다.
 - 다음과 같은 이유 때문에 schedulerLock()을 호출한 프로세스를 실행하는 것은 해당 프로세스에서 yield()를 호출하는 것보다 높은 우선순위를 가지고 있습니다. 따라서, scheduleUnlock()을 호출하지 않고 yield()를 호출할 경우, 해당 system call은 무시되고 계속 해당 프로세스를 scheduling할 것입니다.
 - B. 만약 해당 프로세스가 RUNNABLE 이외의 다른 state (SLEEPING, ZOMBIE, etc)일 경우, 해당 프로세스는 scheduler를 lock할 자격이 없다고 판단. Nullifylock()을 호출한 후 일반 MLFQ scheduler workflow로 넘어갑니다 (goto SCHEDULER);
 - 해당 case B workflow에 해당하는 경우는 schedulerUnlock()을 호출하지 않고 exit() 하는 경우가 있겠으며, 해당 프로세스를 곧바로 인식하고 일반적인 scheduler로 넘어 가게 합니다.
 - 3. 이후 일반적인 context switching을 진행합니다.
 - 같은 process를 switching 하는 것이 필요 없는 overhead일 수도 있겠다고 판단했으나, 해당 overhead를 제거하기 위해 yield(), sched()와 같이 여러 서비스에서 사용되는 system call을 수정하는 것은 더욱 큰 overhead라고 판단했습니다. 또한, 엄밀히 말하면 해당 경우도 ‘다음 대상 프로세스’를 새로 scheduling 하는 것인데, 같은 프로세스라고 해서 예외는 없어야 한다고 판단했습니다.

Workflow C. Scheduler – Case #2) L0, L1 (Round Robin)

```

else
{
SCHEDULER:
    int level = retlevel(); /* Level of queue - Initialize, determine the level of queue after the loop is over
    /* MLFO Rule:
    // L0: RR, Mostly Prioritized.
    // L1: RR
    // L2: Priority Scheduling based on process->priority, FCFS for the same priority level.
    if(level < 2) /* L0, L1 - RR
    {
        int *idx = &lidx[level]; /* * Index for Queue.
        *idx = (*idx) % NPROC; /* Prevents overflow, also allows to restart the same queue for new process

        for(; *idx < NPROC; (*idx)++)
        {
            /* Searches for level of queue needed to be executed.

            int new_level = retlevel();
            /*Check if level is different; if new level is smaller(prioritized) than current level, the index must be reset to 0.
            if(new_level == -1) /* Scheduler Locked: jumps to schedulerLock logic.
                goto LOCKED;

            if(new_level == 2) /* L2 - needs different logic - jumps to L2 logic
                goto L2;

            if(new_level < level)
            {
                lidx[new_level] = 0; /*Reset to 0.

            }

            if(new_level != level)
            {
                level = new_level; /* changes to new level
                if(lidx[level] >= NPROC) /* After switch, assume new queue reached the end of the queue, and needs to be reset.
                    lidx[level] = lidx[level] % NPROC;
                idx = &lidx[level];
            }

            if(L[level][*idx] == 0)
                continue; /* empty cell - moves to next cell

            if(L[level][*idx]->state != RUNNABLE)
                continue; /* Not runnable process - moves to next cell

            p = L[level][*idx]; /* Assign current process.
            c->proc = p;
            switchchuvm(p);
            p->state = RUNNING;
            switch(&(c->scheduler), p->context);
            switchchkvm();
            c->proc = 0;
        }
    }
}

```

Scheduler Function [3/5], Workflow for queue L0, L1 (proc_mlfq.c)

다음은 MLFQ scheduler의 workflow 중 L0, L1의 scheduling에 대한 workflow입니다.

Scheduling을 진행하기 전, retlevel()이라는 function call을 통해 현재 몇 번째 queue에서 scheduling을 진행해야 하는지 결정합니다. (int level = retlevel()) 만약 2가 return 되면, L0, L1에 scheduling이 가능한 프로세스가 존재하지 않는다는 의미이므로, L2를 scheduling 하는 파트로 넘어갑니다. 이 workflow는 return value가 0 혹은 1, 즉 L0, L1에 scheduling이 가능한 프로세스가 있을 시에 작동하는 workflow입니다.

1. int *idx = &lidx[level];
- 결정된 level의 마지막으로 scheduling 한 위치를 불러옵니다. L1의 6번째 process까지 scheduling을 했을 때 L0에 새로운 프로세스가 들어올 경우, 해당 L0로 이동하여 다시 scheduling을 합니다. 새로운 프로세스에 대한 작업을 마친 후 다시 L1으로 돌아올 시, 0 번째가 아닌 7번째부터 작업을 시작합니다. 해당 로직을 통해 Round Robin의 Scheduling 순서를 지킬 수 있습니다.
- Modulo 계산을 통해 overflow를 방지합니다.
2. int new_level = retlevel()
- 한 process에 대한 scheduling이 끝난 후, 다음 process를 scheduling 하기 전, 새로운 프로세스의 여부 판단을 통해 다음 scheduling할 queue를 선택합니다. Scheduling할 queue의

level이 달라지는 경우는 다음과 같습니다. (if (new_level != level))

- L0에서 scheduling 도중 retlevel()에서 1이 나오면, L0에서 더 이상 scheduling이 가능한 process가 없다는 의미입니다. 해당 경우 L1의 scheduling을 위해 lidx1의 index 을 불러오고, 해당 위치에서 scheduling을 시작합니다.
- L1 scheduling 도중 retlevel()이 0이 나오면 비어있던 L0에 새로운 프로세스가 도착 했다는 의미입니다. 해당 경우에는 L0의 index를 초기화하고(if(new_level < level)) L0의 첫 번째 index부터 scheduling을 시작합니다.
- 만약 retlevel()이 -1이 나오면, 어떤 프로세스가 schedulerLock()을 호출했다는 의미 입니다. 더 이상의 scheduling은 중지하고, 해당 프로세스를 최우선으로 scheduling 하는 Workflow B로 이동합니다.

3. 해당 Queue의 index의 있는 process 가 비어있거나 ($L[\text{level}][\text{idx}] == 0$), RUNNABLE한 상태가 아니라면 ($L[\text{level}][\text{idx}] \rightarrow \text{state} \neq 0$) 다음 index를 scheduling합니다.
4. Scheduling할 process를 찾았을 시, 일반적인 context switching을 진행합니다.

Workflow D. Scheduler – Case #3) L2 (Priority Queue)

```
else if(level == 2)
{
    /* L2 - Priority Queue based on process priority, FCFS for the same priority (Only executed if there are no runnable process in L0 and L1.) */

L2:
    int tgt_idx = -1; /* target process index;
    int priority = 1000; /* priority; initialized by the LOWEST value that can't be assigned to process.
    int arrived = -1; /* arrived: initialized by -1. It is initialized there is no cap for arrived; it needs to be initialized by the first process.
    int idx = 0; /* no need to memorize index for L2 -> search the whole L2 queue to find process every time.

    for(idx = 0; idx < NPROC; idx++)
    { /* Find the highest priority with highest arrived value.
        if(L[2][idx] == 0)
            continue; /* empty cell - moves to next cell
        if(L[2][idx] > state != RUNNABLE)
            continue; /* Not runnable process - moves to next cell
        if(L[2][idx] > priority)
            || (L[2][idx] > arrived && arrived != -1)) // if arrived haven't be initialized, than arrived will not be included in condition.
            continue; /* Lower Priority or comes late - moves to next cell

        /* FOUND IT
        tgt_idx = idx;
        priority = L[2][idx] > priority; /* Update Value - It must be higher than this priority
        arrived = L[2][idx] > arrived; /* Update Value - It must arrive faster than this arrival time.
    }
    /* Execute the found process
    if(tgt_idx != -1 && priority != 1000)
    { /* Process Found: tgt_idx, priority must be updated for the targeting process

        p = L[2][tgt_idx]; /* Assign current process.
        c->proc = p;
        switchuvmp(p);
        p->state = RUNNING;
        swtch(&(c->scheduler), p->context);
        switchkvm();

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

Scheduler Function [4/5], Workflow for queue L2 (proc_mlfq.c)

다음은 MLFQ scheduler의 workflow 중 L2의 scheduling에 대한 workflow입니다.

Scheduling을 진행하기 전, tgt_idx, priority, arrived 변수에 대한 초기화를 진행해 줍니다. Priority와 arrived는 일반적인 방법으로는 설정될 수 없는 값으로 초기화를 각각 진행해 줍니다. 전체 L2를 탐색한 후, 가장 Priority가 높은 프로세스를 scheduling합니다. 같은 Priority 중에서는 arrived가 높은 값 (즉, L2에 더 빨리 도착한 프로세스)를 scheduling함으로서 FCFS policy를 구현했습니다.

1. 전체 L2를 돌며 대상 프로세스를 찾습니다. 대상 프로세스를 찾으면 해당 위치를 tgt_idx 에 넣고, priority와 arrived를 해당 프로세스의 값으로 변경합니다. 다음 프로세스가 이 프로세스보다 priority가 높거나, priority가 같고 arrived가 더욱 낮지 않은 이상 이 값들은 변경되지 않습니다.
 - A. 비어있거나 ($L[2][\text{idx}] == 0$) RUNNABLE한 상태가 아니라면 ($L[2][\text{idx}] \rightarrow \text{state} \neq 0$) 판단 대상에서 제외됩니다.
 - B. 만약 arrived가 초기값에서 변경되지 않았다면, 지금 검색하고 있는 process가 L2의 첫번째 프로세스이라는 의미입니다. 해당 경우는 arrived여부에 상관 없이 tgt_idx, priority, arrived를 해당 값으로 초기화를 진행해줍니다. (어떠한 경우에도 priority는

1000보다는 높으니, priority에 대해서는 예외처리를 하지 않았습니다.)

2. L2를 모두 search한 후, arrived와 priority값이 초깃값에서 변경되었다면, L2에서 scheduling 할 프로세스를 찾았다는 의미입니다. tgt_idx값을 바탕으로 scheduling할 프로세스를 참조하고 context switching을 진행합니다.

Workflow E. Scheduler – Exit Phase

```
else /* ERROR
{
    panic("MLFQ scheduler()");
}
release(&ptable.lock);
}
```

Scheduler Function [5/5], Exit phase (proc_mlfq.c)

Workflow B, C, D에 해당하지 않는 경우는 오류 케이스입니다. 해당 경우 panic()을 실행함으로써 xv6의 실행을 중단합니다.

이후는 일반적인 xv6의 workflow와 같이, for loop(;;;)가 끝나면 잠시 ptable을 Release 해줍니다.

이상 제 MLFQ Scheduler의 Workflow이었습니다. 해당 Scheduler의 경우 O(n)의 실행 시간을 가지고 있으며, 평균 2~4ms 안에 scheduling과 context switching을 완료합니다. 이는 기본 1 tick의 시간인 10ms보다 낮은 수치이며, 1 tick 안에 scheduling이 결정되는 것을 보장합니다.

IV. Implemented System Calls & Function Calls

제가 구현한 System Calls와 Function Calls에 대한 설명입니다. 명세에서 요구한 필수적인 System calls에 대한 자세한 설명과, MLFQ Scheduling에 필요한 Function Calls들에 대한 간략한 설명을 서술했습니다.

System Calls

1. void yield(void)

```
// Give up the CPU for one scheduling round.
void
yield(void)
{
    acquire(&ptable.lock); //DOC: yieldlock
    myproc()->state = RUNNABLE;
    sched();
    release(&ptable.lock);
}
```

yield() – system call (proc_mlfq.c)

```
/* yield()
int sys_yield (void)
{
    //cprintf("yield() called in proc.c\n");
    yield();
    return 0;
}
```

sys_yield() – wrapper function (mlfqsyscall.c)

기존 xv6의 yield()의 역할과 동일합니다. User mode에서도 사용할 수 있도록 Wrapper function을 추가해준 것 이외의 수정사항은 없습니다.

2. int getLevel(void)

```
/* getLevel()
int
getLevel(void)
{
    if(myproc())
    {
        //cprintf("Current Process Level: %d\n", myproc()->level);
        return myproc()->level; /* Return current process level.
    }
    else /* * Error Case;
    return -1;
    return 0;
}
```

getLevel() – system call (proc_mlfq.c)

```
/* getLevel()
int
sys_getLevel(void)
{
    return getLevel();
}
```

sys_getLevel() – wrapper function (mlfqsyscall.c)

호출될 시, 현재 실행되고 있는 프로세스가 위치한 Queue의 level을 반환합니다. (하단의 return 0은 docker gcc의 compiler가 해당 return 값이 없으면 컴파일 에러를 일으킵니다. 해당 값이 return되는 경우는 존재하지 않습니다.)

3. void setPriority(int pid, int priority)

```
/* setPriority(): set current process priority.
void
setPriority(int pid, int priority)
{
    if(priority < 0 || priority > 3) /* Error Case: wrong priority value
        return;

    struct proc *p;

    acquire(&ptable.lock);

    for(p = ptable.proc; p < &ptable.proc[NPROC]; p++) /* Search process have such pid.
    {
        if(p->pid != pid)
            continue;
        else
        {
            p->priority = priority; /* Update Priority
            //cprintf("Priority Set: Pid: %d, Priority: %d\n", p->pid, p->priority);
            break;
        }
    }
    release(&ptable.lock);
}
```

setPriority() – system call (proc_mlfq.c)

```
/* setPriority()
int
sys_setPriority(void)
{
    int pid, priority;

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

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

    setPriority(pid, priority);
    return 0;
}
```

sys_setPriority() – wrapper function (mlfqsyscall.c)

Pid에 해당하는 process를 찾아 priority의 값을 설정해줍니다. L0~L2를 모두 도는 것보다 ptable의 프로세스를 도는 것이 더욱 효율적이어서 ptable에서 작업을 진행합니다. Wrapper function에서는 인자가 올바르게 있지 않은 경우 -1을 return합니다.

만일 인자 priority의 값이 올바르지 않다면 (priority < 0 || priority > 3) 곧바로 return합니다.

4. void schedulerLock(int password)

```
/* schedulerLock(): lock scheduler, monopolize CPU at 100 ticks maximum.
void
schedulerLock(int password)
{
    //cprintf("schedulerLock() called -> password: %d\n", password);

    if(lockedproc != 0)
    {/** IGNORE: Some process already locked scheduler. It cannot be done.
        cprintf("IGNORE: Scheduler already locked! Ignoring...\n");
    }
    else if(password != LOCK_PW)
    {/** REJECT: Password didn't match
        cprintf("REJECT: Password incorrect, forcing to stop current process...\n");
        cprintf("[REJECTED PROCESS] Pid: %d / Elapsed Time Quantum: %d / Level: %d\n",
               myproc()->pid, myproc()->lock == LOCKED ? 100 - myproc()->tq : ((2* myproc()->level)+4)-myproc()->tq, myproc()->level);
        kill(myproc()->pid);
    }
    else
    { /* Lock scheduler.
        myproc()->lock = LOCKED;
        myproc()->tq = 100; /* Allocate 100 tick;
        lockedproc = myproc();
        L[myproc()->level][myproc()->idx] = 0; /* Remove current process from MLFQ.
        __asm__("int $131"; /* Call interrupt: reset global tick to 0
        cprintf("SCHEDULER LOCKED! - PID: %d\n", myproc()->pid);
    }
    return;
}
```

schedulerLock() – system call (proc_mlfq.c)

scheudulerLock() system call을 호출한 프로세스는 100tick의 time quantum동안 그 어떤 프로세스 보다도 높은 우선순위를 갖게 됩니다. 한 번에 두 프로세스가 schedulerLock()을 호출할 수는 없으며, 해당 time quantum 동안은 yield()마저 무시되고 계속 scheduling됩니다. yield()를 하기 위해서는 반드시 schedulerUnlock()을 호출해야 하며, 프로세스가 종료될 시 자동으로 Lock은 사라지게 됩니다.

- schedulerLock()을 호출할 경우, 먼저 이미 해당 schedulerLock()을 호출한 프로세스가 있는지 확인합니다. 만약 schedulerLock()을 호출한 프로세스가 아직 schedulerUnlock()을 호출하지 않았거나 아직 우선적으로 scheduling되는 프로세스가 존재할 시, 해당 system call은 무시(Ignore)합니다.
- schedulerLock()을 호출 시, 비밀번호가 틀렸을 경우, 해당 system call을 거부(Reject)하며, 해당 프로세스를 종료시킵니다. 이때, 해당 프로세스의 Pid, 해당 queue에서 사용한 Time quantum, level을 출력합니다.
- 성공적으로 schedulerLock()을 호출 시, 해당 프로세스의 lock은 UNLOCKED에서 LOCKED로 변경되며, 100의 time quantum을 할당받게 됩니다. 기존 Queue에서는 제거가 되고 lockedproc에 위치되어 최고 우선순위를 갖게 됩니다. 마지막으로 131번 interrupt를 호출하는데, 이는 global tick을 0으로 초기화를 하는 T_RSTGT interrupt입니다.

5. void schedulerUnlock(int password)

```
/* ( schedulerUnlock(): unlock scheduler, return current process to MLFQ.
void
schedulerUnlock(int password)
{
    if(lockedproc == 0)
    { /** IGNORE: there are no process called schedulerLock. It cannot be done.
        cprintf("IGNORE: Scheduler is not locked! Ignoring...\n");
    }
    else if(password != LOCK_PW)
    {/** REJECT: Password didn't match
        cprintf("REJECT: Password incorrect, forcing to stop current process...\n");
        cprintf("[REJECTED PROCESS] PID: %d / Elapsed Time Quatum: %d / Level: %d\n",
               myproc()->pid, myproc()->lock == LOCKED ? 100 - myproc()->tq : ((2* myproc()->level)+4)-myproc()->tq, myproc()->level);
        kill(myproc()->pid);
    }
    else
    { /* Unlock scheduler
        nullifylock(); /* Nullify current lock, return locked process to MLFQ (to the most front index in L).
        cprintf("SCHEDULER UNLOCKED! - PID: %d\n", myproc()->pid);
    }
    return;
}
```

schedulerUnock() – system call (proc_mlfq.c)

schedulerUnlock()을 호출한 프로세스를 다시 원래 MLFQ의 위치에 돌려놓습니다. 돌아가는 위치는 L0의 가장 앞자리이고, priority는 3으로 초기화됩니다.

- schedulerUnlock() 호출 시, schedulerLock()을 호출한 프로세스가 존재하지 않을 경우, 해당 system call은 무시 (Ignore)합니다.
- schedulerUnlock()을 호출 시, 비밀번호가 틀렸을 경우, 해당 system call을 거부(Reject)하며, 해당 프로세스를 종료시킵니다. 이때 역시 해당 프로세스의 Pid, 해당 queue에서 사용한 Time quantum, level을 출력합니다.
- 성공적으로 schedulerUnlock()을 호출했을 경우, lockedproc에 있던 프로세스를 다시 L0의 가장 앞자리로 옮기고 priority를 초기화해주는 nullifylock() function call을 합니다. 해당 함수의 대한 설명은 이후 Function Call에서 더욱 자세히 서술하겠습니다.

schedulerLock(), schedulerUnlock()의 Wrapper Function 정의입니다.

<pre>/* schedulerLock() int sys_schedulerLock(void) { int password; if(argint(0, &password) < 0){ return -1; } schedulerLock(password); return 0; }</pre>	<pre>/* schedulerUnlock() int sys_schedulerUnlock(void) { int password; if(argint(0, &password) < 0){ return -1; } schedulerUnlock(password); return 0; }</pre>
sys_schedulerLock() – wrapper function (mlfqsyscall.c)	sys_schedulerUnlock() – wrapper function (mlfqsyscall.c)

이후는 기존 system call 중 수정되었지만 위 MLFQ Design과 Scheduler Design에서 설명하지 못한 부분을 서술합니다.

```
for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
    if(p->parent != curproc)
        continue;
    havekids = 1;
    if(p->state == ZOMBIE){
        // Found one.

        /* Remove current process from the queue.
        int level = p->level;
        int idx = p->idx;
        L[level][idx] = 0;
        /* Now, current cell is usable for another new process.
        //cprintf("RELEASED PROCESS -> PID: %d / LEVEL: %d / INDEX: %d / PRIORITY: %d\n", p->pid, p->level, p->idx, p->priority);

        pid = p->pid;
        kfree(p->kstack);
        p->kstack = 0;
        freevm(p->pgdir);
        p->pid = 0;
        p->parent = 0;
        p->name[0] = 0;
        p->killed = 0;
        p->state = UNUSED;
        /* Reset property added for MLFQ.
        p->level = 0;
        p->idx = 0;
        p->priority = 0;
        p->arrived = 0;
        p->tq = 0;
        p->lock = UNLOCKED;

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

wait() – modified for MLFQ scheduling (proc_mlfq.c)

기존 wait() 시스템 콜에서 일부를 수정했습니다. (위 사진은 wait()시 ZOMBIE상태의 프로세스를 UNUSED상태로 만드는 로직이며, 해당 부분 이외에는 모두 동일한 내용입니다.) 해당 프로세스의 level과 index값을 참조하여 MLFQ Queue에서 제거를 하고, 관련된 변수를 모두 초기화 하는 작업을 진행합니다.

Function Calls

MLFQ Scheduling을 도와주는 function calls입니다. System call과 달리 user mode에서 사용이 불가능하고, 오직 MLFQ의 기능의 일부를 효율적으로 실행하기 위한 function들입니다.

1. int rettq(struct proc *p)

```
/* rettq: return time quantum for process
int
rettq(struct proc *p) /* Return time quantum for each queue.
{
    if(p->lock == LOCKED)
    { /*Locked process, gives 100 timequantum as max.
        return 100;
    }
    else
    {
        return (2 * p->level) + 4;
    }
}
```

rettq() – return time quantum for current process (proc_mlfq.c)

프로세스를 인자로 받고, 해당 프로세스의 level에 맞추어 time quantum ($2n + 4$)을 return합니다. 만약 해당 프로세스가 schedulerLock()을 호출한 프로세스라면 ($p->lock == LOCKED$) 레벨과 무관하게 time quantum 100을 return합니다.

2. int retlevel(void)

```
/* retlevel: return level of queue has RUNNABLE process
int
retlevel(void)
{ /* Return level of queue for next process
   /* Searches for the Runnable, returns level where the RUNNABLE process firstly found.

    int level = 0;
    int idx = 0;

    for(level = 0; level < MLFQ.LEV; level++)
    {
        if(level == 2){
            /* If current level is 2, it means there is no RUNNABLE process for 0 and 1. It is reasonable for early return of level
            goto RET;
        }
        for(idx = 0; idx < NPROC; idx++)
        {
            if(L[level][idx] != 0 && L[level][idx]->state == RUNNABLE)
            {
                goto RET; /* If RUNNABLE process found, return current level.
            }
        }
    }

    RET:
    return level;
}
```

retlevel() – return queue level for scheduling (proc_mlfq.c)

MLFQ Scheduling에 사용되는 Queue 전체를 돌며, 가장 먼저 RUNNABLE 프로세스가 발견되는 queue의 level을 return합니다. level==2는 L0, L1에 RUNNABLE한 프로세스가 존재하지 않는다는

의미임으로, 곧바로 2를 return합니다. 이 function은 매 scheduling 전, scheduling을 진행할 queue를 찾을 때에 사용됩니다. (Scheduler Function [3/5] 참고)

3. int demoteproc(void)

```
/*demoteproc - demote process level to lower level.
int
demoteproc(void)
{
    if(myproc()->level < 2) /* level 0 -> level 1, level 1 -> level 2
    {
        int idx = 0; /* new index for demoted process.
        int new_level = myproc()->level + 1;

        for(idx = 0; idx < NPROC; idx++)
        {
            if(L[new_level][idx] == 0) /* Empty cell found
            {
                L[myproc()->level][myproc()->idx] = 0; /* remove current process from previous level.
                L[new_level][idx] = myproc(); /* Assign current process to new level.
                myproc()->idx = idx; /* Gives new index for current queue.
                myproc()->level = new_level; /* Update process level.
                myproc()->tq = rettq(myproc()); /* Update new time quantum

                if(new_level == 2) /* If new level is L2, gives new arrived value.
                    myproc()->arrived = arrived++;

                //cprintf("Demoted Process: %s // PID: %d, Allocated in L%d[%d]\n", myproc()->name, myproc()->pid , myproc()->level, idx);
                /* Debug: Comment this line if it is not required.
                break; /* Exit Loop
            }
        }
    }
    else
    {
        panic("demoteproc()");
    }

    return 0;
}
```

demoteproc() – move current process to the next queue (proc_mlfq.c)

현재 process의 queue level을 낮추는 데에 사용합니다. L0, L1에서 프로세스의 time quantum이 모두 소진되었을 때에 사용되며, L0에서 L1로, L1에서 L2로 프로세스를 옮깁니다. 옮기며 새로운 레벨과 새로운 위치(index), 새로운 time quantum을 할당해주며, L2로 내려온 경우 (new_level == 2), 새로운 arrived 값까지 할당해줍니다. (이 arrived 값으로 더욱 빨리 온 프로세스를 결정합니다.)

4. void incpriority(void)

```
/* incpriority(): increase current priority level.
void
incpriority(void)
{
    myproc()->tq = rettq(myproc()); /* Reset Time Quantum

    if(myproc()-> priority > 0)
        (myproc()->priority)--; /* Increase current priority;

    //cprintf("Increased Priority // PID: %d, Priority became %d\n", myproc()->pid, myproc()->priority);
    /* Debug: Comment this line if it is not required.

    return;
}
```

incpriority() – Increase current process priority (proc_mlfq.c)

현재 프로세스의 priority를 1씩 감소시킬 때 (즉, priority를 1씩 높여줄 때) 사용합니다. Priority가 이미 0인 경우는 더 이상 감소를 하지 않습니다. L2에서 scheduling되던 프로세스가 자신에게 주어진 time quantum을 소진할 경우 사용되며, 명세의 내용을 따라 1단계 높은 priority를 할당해줍니다.

5. void boostpriority(void)

```
/* boostpriority() : do priority boosting
void
boostpriority(void)
{ /* Boost the whole priority if global tick became 100.
   struct proc* p;
   int level = 0;
   int idx = 0;
   int new_idx=0;
   /* Send the whole process in L0 and L2 to L0, reset its time quantum and priority.
   /* relocate process in L0 either to eliminate empty cells between processes.

   for(level = 0; level < MLFQ_LEV; level++)
   {
      for(idx = 0; idx < NPROC; idx++)
      {
         if(L[level][idx] == 0)
            continue;
         if(level == 0 && L[level][idx]->idx == new_idx) /* In level 0, there is no need to move the process to the same place.
         {
            new_idx++;
            continue;
         }

         p = L[level][idx];
         L[level][idx] = 0; /* Detach process from the queue.
         L[0][new_idx] = p; /* relocate current process to new process, increase new_idx value after this instruction.
         p->tq = 4; /* Reset its time quantum. (L0)
         p->level = 0; /* Reset its level.
         p->idx = new_idx; /* Gives new index.
         p->priority = 3; /* Reset its priority.
         p->arrived = 0; /* Reset its arrived

         new_idx++; /* Increase new_idx value.
      }
   }

   /* Reset arrived counter
   arrived = 0;
}
```

boostpriority(void) – boost priority to prevent starvation (proc_mlfq.c)

명세에 나온 Priority Boosting을 구현한 함수입니다. 해당 level의 index에 프로세스를 발견해면 L0의 새로운 위치(new_index)로 옮기고 새로운 time quantum을 제공합니다. 이를 반복하여 모든 Queue의 프로세스를 L0으로 위치를 재조정합니다. arrived 또한 초기화되며, 해당 값은 다시 0부터 할당되기 시작합니다. L0에 있는 프로세스에 한해서, 해당 프로세스의 index와 새로 할당할 new_index가 같다면 해당 프로세스는 priority boosting 이전과 이후 모두 L0의 같은 위치에 있다는 의미이며, 이 경우는 위치 재조정을 하지 않음으로써 workflow의 효율성을 증가시켰습니다.

6. void nullifylock(void)

```
/* nullifylock() - nullify the lock, and relocate locked process to mlfq queue.
void
nullifylock(void)
{
   if(lockedproc != 0)
   {
      int idx = 0; /* Index for search amount of existing element in front of L0.
      int mov = 0; /* Index used for moving

      for(idx = 0; idx < NPROC; idx++)
      {
         /* Make the space for locked process.
         if(L[0][idx] != 0) /* Check amount of element to move
            continue;

         /* Empty element found!
         for(mov = idx; mov > 0; mov--) /* move the element to the empty space one by one.
         {
            L[0][mov] = L[0][mov-1]; /* Moves one right.
            L[0][mov]->idx = mov;
         }

         L[0][0] = lockedproc; /* Located Locked process to the frontmost element in L0.
         lockedproc = 0;
         L[0][0]->level = 0;
         L[0][0]->tq = 4;
         L[0][0]->idx = 0;
         L[0][0]->priority = 3;
         L[0][0]->lock = UNLOCKED;
         break;
      }
   }
   return;
}
```

nullifylock() – remove lockedproc, move the process to front of MLFQ L0. (proc_mlfq.c)

`schedulerUnlock()`이 호출되거나 `global tick`이 1000이 되었을 때, `lockedproc`이 더 이상 실행 가능한 상황이 아닐 때 호출이 됩니다. `lockedproc`에 있는 프로세스를 명세의 내용에 맞춰 MLFQ L0의 첫 번째 자리로 옮겨줍니다. 이때, 첫 번째 자리에 이미 `process`가 있다면, 한 칸씩 뒤로 옮겨줍니다. 이는 마지막으로 옮겨지는 프로세스가 빈 자리에 들어갈 때까지 반복이 됩니다. (예를 들어, `idx = {0,1,2,4}`에 프로세스가 존재할 때 `idx = {1,2,3,4}`가 되도록 0,1,2를 한 칸씩 옮겨줍니다.) 이후 `lockedproc`은 다시 0으로 만들어 비어있음을 나타내고, 해당 프로세스는 L0에 맞추어 다시 초기화를 해줍니다.

V. MLFQ Scheduling Workflow & Test Cases (Result)

제가 구현한 MLFQ는 지금까지 구현된 scheduler, system call, function call을 활용하여 다음과 같이 작동합니다.

MLFQ Workflow
<ol style="list-style-type: none">1. 프로세스가 생성되면, allocproc()에서 프로세스 초기값을 할당받은 후, time quantum 4를 가지고 L0에 할당됩니다. L0의 첫번째로 보이는 빈 공간에 할당을 받습니다.2. 스케줄러가 다음 프로세스를 찾습니다.<ol style="list-style-type: none">A. 만약 lockedproc이 비어있지 않으면 ($\neq 0$), MLFQ의 로직은 작동하지 않고, lockedproc을 다음 scheduling의 대상으로 잡습니다.<ol style="list-style-type: none">a. 만약 해당 lockedproc의 state가 RUNNING하지 않으면,<ol style="list-style-type: none">i. RUNNABLE일 경우는 다시 RUNNING으로 바꾸어 준 후, scheduling의 대상으로 잡습니다.ii. 그 외 state이 경우, 해당 프로세스는 더 이상 scheduler를 lock하고 있을 자격이 없다고 판단, nullifylock()을 호출하여 lockedproc에 있는 프로세스를 제거한 후 (다시 MLFQ에 되돌려 놓은 후,) 기존 MLFQ scheduler를 호출합니다.B. 만약 lockedproc이 비어있으면, MLFQ Scheduler가 작동을 합니다. retlevel()을 호출하여 scheduling을 진행할 queue를 선택합니다.<ol style="list-style-type: none">a. 만약 retlevel()에서 0 혹은 1을 return 하면 Round-Robin policy를 따르는 L0, L1에서 scheduling을 진행합니다.<ol style="list-style-type: none">i. 매 scheduling 전, 한번 더 retlevel()에서 scheduling을 진행할 queue를 선택합니다.ii. retlevel()의 결과와 현재 level이 같다면, 이번 없이 현재 queue를 scheduling합니다.iii. retlevel()의 결과가 현재 level과 다르다면, 다음과 같은 4개의 기점을 같습니다.<ul style="list-style-type: none">- Case #1) 현재 level이 0인데 retlevel()의 결과가 1이라면, L0에는 더 이상 RUNNABLE한 프로세스가 존재하지 않는다는 의미입니다. 이 때, scheduling의 대상은 L0에서 L1으로 변경되고, 마지막으로 scheduling을 진행한 L1의 위치에서부터 scheduling을 진행합니다.- Case #2) 현재 level이 1인데, retlevel()의 결과가 0이라면, 우선순위가 더 높은 L0에 새로운 프로세스가 들어왔다는 의미입니다. 이 때, scheduling의 대상은 L1에서 L0로 변경되게 됩니다. 새로운 프로세스는 비어있는 L0의 첫번째 자리 (첫번째로 발견된 빈 공간)에 할당이 되기에, L0의 첫번째 자리에서부터 scheduling을 진행합니다.- Case #3) 현재 level이 0 혹은 1인데 retlevel()의 결과가 2라면, L0, L1에는 RUNNABLE한 프로세스가 존재하지 않는다는 의미입니다. 이때 L2를 scheduling하는 workflow로 넘어갑니다. (b단계)- Case #4) 현재 level이 0 혹은 1인데 retlevel()의 결과가 -1이라면, 어떠한 프로세스가 schedulerLock()을 호출하여, 해당 프로세스를 최우선순위로 scheduling 해야 한다는 의미입니다. 이때

- lockedproc이 비어있지 않을 때의 workflow로 넘어갑니다 (A단계)
- b. 만약 retlevel()에서 2를 return하면, Priority Queue를 사용하는 L2에서 scheduling을 합니다. 같은 Priority시에는 FCFS Policy를 따릅니다.
 - i. 매 scheduling 전, 전체 L2 queue를 돌며 가장 높은 priority를 찾습니다.
 - ii. 같은 priority인 경우, arrived가 더 적은 (즉, 더욱 L2에 빠르게 온) 프로세스를 대상으로 합니다.
 - iii. 대상 프로세스를 찾을 경우, 해당 프로세스를 scheduling합니다.
3. 1 tick마다 timer interrupt를 발생시켜 yield()를 통해 다음 프로세스가 scheduling되도록 합니다.
 - A. 만약 현재 프로세스가 schedulerLock()을 호출한 프로세스라면, 해당 yield는 무시합니다. Global tick이 100이 되거나, schedulerUnlock()을 호출하지 않는 이상, 해당 프로세스는 자신의 time quantum만큼의 독점적인 CPU사용을 보장합니다.
 4. L2에서의 scheduling이 끝나거나, L0, L1에서 index의 끝까지 도달했을 경우, 다시 2번으로 돌아가 scheduling을 계속합니다. 이때 modulo연산을 통해 index overflow를 방지합니다.
 5. 프로세스의 time quantum (tq)가 모두 소진되었을 경우, 다음과 같은 분기점을 갖습니다.
 - A. 만약 해당 프로세스가 L0, L1에 있을 경우, 다음 큐로 해당 프로세스를 이동시키는 demoteproc()을 호출합니다. (L0->L1, L1->L2) 해당 프로세스는 새로운 level과 index, time quantum을 할당받으며, L2로 이동하는 경우에는 arrived값 또한 받습니다.
 - B. 만약 해당 프로세스가 L2에 있을 경우, priority를 1 감소시켜 더욱 높은 우선순위를 주는 incrpriority()를 호출합니다. Priority가 이미 0인 경우, 더 이상 감소되지 않습니다.
 - C. 만약 해당 프로세스가 schedulerLock()을 호출한 프로세스(lockedproc)일 경우, 해당 프로세스를 다시 MLFQ로 돌려놓는 nullifylock()을 호출합니다. 해당 경우는 priority boosting이 일어나는 조건과 같아 boostPriority()또한 호출됩니다.
 6. Global ticks이 100의 배수가 되는 경우 (100이 되는 경우), priority boosting이 일어납니다.
 - A. 모든 프로세스는 L0으로 재조정되며, time quantum은 4로, priority는 3으로 초기화 됩니다. L2의 프로세스의 경우 arrived 또한 0으로 초기화됩니다.
 7. 프로세스가 schedulerLock()을 호출 할 경우, MLFQ Scheduling은 동작하지 않고, 해당 프로세스를 최우선순위로 scheduling합니다.
 - A. 해당 프로세스는 MLFQ에서 제거가 되고, lockedproc이라는 최우선순위 프로세스를 위한 공간에 할당됩니다.
 - B. Global Tick또한 0으로 초기화가 됩니다.
 - C. 해당 process는 schedulerUnlock()을 호출하거나 RUNNING, RUNNABLE한 상태가 아닐때까지 최대 100tick의 독점적 scheduling을 보장합니다. 이는 해당 프로세스의 yield()호출에서도 마찬가지입니다. schedulerUnlock()을 호출하지 않고 yield()를 호출할 경우, 해당 yield()또한 무시됩니다.
 - D. 이미 schedulerLock을 호출한 프로세스가 존재하고, 해당 프로세스가 아직 독점적 권한을 가지고 있다면 해당 system call은 무시합니다.
 - E. 만약 해당 system call의 비밀번호를 틀렸다면, 해당 프로세스를 종료하고 pid, time quantum, level을 출력합니다.
 8. 프로세스가 schedulerUnlock()을 호출 할 경우, 해당 프로세스를 MLFQ에 돌려놓고 MLFQ Scheduler가 다시 동작하도록 합니다.

- A. lockedproc에 있는 프로세스는 MLFQ L0의 첫번째 자리로 이동, 해당 위치에 맞게 level, index, time quantum등을 초기화 합니다. (nullifylock())
- B. lockedproc을 다시 비움으로서 MLFQ Scheduler가 다시 동작하게 합니다.
- C. 이미 lockedproc이 비워져 있다면 해당 system call은 무시합니다.
- D. 만약 해당 system call의 비밀번호를 틀렸다면, 해당 프로세스를 종료하고 pid, time quantum, level을 출력합니다.
9. 프로세스가 종료될 시, ZOMBIE에서 UNUSED가 되는 workflow에서 모든 MLFQ관련 변수를 초기화합니다. 해당 프로세스는 MLFQ에서 제거됩니다.

Results

제 MLFQ가 잘 동작하는 것을 보여주는 자체 제작 Test Case와 Piazza에서 공유된 Test Case에 대한 결과입니다. 눈에 보이는 결과를 위해 Scheduler 중간중간의 `cprintf()` 구문을 uncomment했으며, 전체 프로세스를 출력하는 `printproc()`, MLFQ의 상태를 보여주는 `printmlfq()`와 같은 system call을 추가 정의 했습니다.

* Makefile에 `proc.o`, `trap.o`를 제거하고, `proc_mlfq.o`, `trap_mlfq.o`, `mlfqsyscall.o`를 추가합니다.

* `./build.sh` shell script를 통해 build합니다.

Compile
<pre>root@ee3468492f57:/OS/xv6-public# ./build.sh gcc -fno-pic -static -fno-builtins -fno-strict-aliasing -O2 -Wall -MD -ggdb -m32 -Werror -fno-omit-frame-pointer -fno-stack-protector -fno-pie -no-pie -c -o proc_mlfq.o proc_mlfq.c ld -m elf_i386 -T kernel.ld -o kernel entry.o bio.o console.o exec.o file.o fs.o ide.o ioapic.o kalloc.o kbd.o lapic.o log.o main.o mp.o picirq.o pipe.o proc_mlfq.o sleeplock.o objdump -S kernel > kernel.asm objdump -t kernel sed '1,/SYMBOL TABLE/d; s/.* /;/; /\$d/ > kernel.sym' dd if=/dev/zero of=xv6.img count=10000 10000+0 records in 10000+0 records out 5120000 bytes (5.1 MB, 4.9 MiB) copied, 0.0220086 s, 233 MB/s dd if=bootblock of=xv6.img conv=notrunc 1+0 records in 1+0 records out 512 bytes copied, 0.001558 s, 329 kB/s dd if=kernel of=xv6.img seek=1 conv=notrunc 429+1 records in 429+1 records out 219844 bytes (220 kB, 215 KiB) copied, 0.00256183 s, 85.8 MB/s make: 'fs.img' is up to date. root@ee3468492f57:/OS/xv6-public#</pre>

Results #1) MLFQ	
<p>A. MLFQ</p> <pre>Booting from Hard Disk..xv6... Allocated Process: // PID: 1, Allocated in L0[0] cpu0: starting 0 580+0 pages 1000 mblocks 941 minodes 200 nlog 30 logstart 2 inodestart 32 bmap start 58 Demoted Process: initcode // PID: 1, Allocated in L1[0] init: starting sh Allocated Process: // PID: 2, Allocated in L0[0] \$ printmlfq Demoted Process: sh // PID: 2, Allocated in L1[0] Allocated Process: // PID: 3, Allocated in L0[0] Demoted Process: printmlfq // PID: 3, Allocated in L1[1] ===== *****CURRENT MLFQ STATUS***** MLFQ STATE: UNLOCKED [PID] level / index / priority / (arrived: for L2) *****L0 - RR, Mostly Prioritized. ***** [1] 0 / 1 / 3 *****L1 - RR, Took a backseat to L0. ***** [2] 1 / 0 / 3 [3] 1 / 3 / 3 *****L2 - Priority Queue. FCFS for same priority ***** RELEASED PROCESS -> PID: 3 / LEVEL: 1 / INDEX: 1 / PRIORITY: 3</pre>	<p>A. MLFQ</p> <ul style="list-style-type: none"> - 프로세스의 Time Quantum이 expired되자 다음 레벨의 Queue에 새로 allocated되는 것을 확인할 수 있습니다. - <code>printmlfq()</code>를 통해 MLFQ를 출력합니다. 도중에 해당 프로세스(PID: 3)가 L0에서 주어진 time quantum을 다 사용해 L1으로 내려갔고, 이를 출력 결과를 통해 확인할 수 있습니다.
<p>B. forktest</p> <pre>\$ forktest Allocated Process: // PID: 3, Allocated in L0[2] fork test Allocated Process: // PID: 4, Allocated in L0[3] Demoted Process: forktest // PID: 3, Allocated in L1[0] Allocated Process: // PID: 5, Allocated in L0[2] Allocated Process: // PID: 6, Allocated in L0[4] Allocated Process: // PID: 7, Allocated in L0[5] Allocated Process: // PID: 8, Allocated in L0[6] Demoted Process: forktest // PID: 3, Allocated in L2[0] Allocated Process: // PID: 9, Allocated in L0[7] Allocated Process: // PID: 10, Allocated in L0[8] Allocated Process: // PID: 11, Allocated in L0[9] Allocated Process: // PID: 12, Allocated in L0[10] Allocated Process: // PID: 13, Allocated in L0[11] Increased Priority // PID: 3, Priority became 2 Allocated Process: // PID: 14, Allocated in L0[12] Allocated Process: // PID: 15, Allocated in L0[14] Allocated Process: // PID: 16, Allocated in L0[15] Demoted Process: forktest // PID: 3, Allocated in L1[0] Allocated Process: // PID: 17, Allocated in L0[13] Allocated Process: // PID: 18, Allocated in L0[16] Allocated Process: // PID: 19, Allocated in L0[17] Allocated Process: // PID: 20, Allocated in L0[18] Demoted Process: forktest // PID: 3, Allocated in L2[0] Allocated Process: // PID: 21, Allocated in L0[19]</pre>	<p>B. forktest</p> <ul style="list-style-type: none"> - forktest와 같은 긴 프로세스의 경우, 해당 프로세스가 다음 레벨로 넘어가는 모습과, L2에서 priority가 상승하는 모습을 볼 수 있습니다.

```

Allocated Process: // PID: 62, Allocated in L0[60]
Increased Priority // PID: 3, Priority became 0
Allocated Process: // PID: 63, Allocated in L0[61]
Allocated Process: // PID: 64, Allocated in L0[62]
RELEASED PROCESS -> PID: 4 / LEVEL: 0 / INDEX: 3 / PRIORITY: 3
RELEASED PROCESS -> PID: 5 / LEVEL: 0 / INDEX: 2 / PRIORITY: 3
RELEASED PROCESS -> PID: 6 / LEVEL: 0 / INDEX: 4 / PRIORITY: 3
RELEASED PROCESS -> PID: 7 / LEVEL: 0 / INDEX: 5 / PRIORITY: 3
RELEASED PROCESS -> PID: 8 / LEVEL: 0 / INDEX: 6 / PRIORITY: 3
RELEASED PROCESS -> PID: 9 / LEVEL: 0 / INDEX: 7 / PRIORITY: 3
Increased Priority // PID: 3, Priority became 0

```

C. Priority Boosting

```

printrPRIORITY BOOSTING!!!
lfq
Allocated Process: // PID: 5, Allocated in L0[2]
Demoted Process: sh // PID: 2, Allocated in L1[0]
exec: fail
exec forprintmlfqDemoted Process: sh // PID: 5, Allocated in L1[1]
failed
RELEASED PROCESS -> PID: 5 / LEVEL: 1 / INDEX: 1 / PRIORITY: 3
$ PRIORITY BOOSTING!!!
PRIORITY BOOSTING!!!
PRIORITY BOOSTING!!!

Allocated Process: // PID: 6, Allocated in L0[2]
RELEASED PROCESS -> PID: 6 / LEVEL: 0 / INDEX: 2 / PRIORITY: 3
$ pPRIORITY BOOSTING!!!
rintmlfq
Allocated Process: // PID: 7, Allocated in L0[2]
Demoted Process: sh // PID: 2, Allocated in L1[0]
CURRENT MLFQ STATUS

```

- 중간에 priority boosting이 발생해 다시 L0 -> L1으로 넘어가는 프로세스(PID:3)를 볼 수 있습니다.

- Priority 0에서 더 이상 우선순위가 증가하는 않는 것을 확인할 수 있습니다.

C. Priority Boosting

- 가독성으로 인해 priority boosting에 대해서는 출력을 하지 않았으나, 만약 할 경우 다음과 같이 꾸준하게 boosting이 일어나는 것을 확인할 수 있습니다.

Result #2) yield(), getLevel(), setPriority()

A. yield(), getLevel() (PID:3)

```

int main(int argc, char* argv[]){
    yield();
    printf(1, "\n====Current Process Level: %d====\n", getLevel());
    printmlfq();
    exit();
}

$ dev
Demoted Process: sh // PID: 2, Allocated in L1[0]
Demoted Process: dev // PID: 3, Allocated in L1[1]
yield() called in proc.c

=====Current Process Level: 1=====
=====CURRENT MLFQ STATUS=====
MLFQ STATE: UNLOCKED
[PID] level / index / priority / (arrived: for L2)
*****L0 - RR, Mostly Prioritized. *****
[1] 0 / 1 / 3
*****L1 - RR, Took a backseat to L0. *****
[2] 1 / 0 / 3
[3] 1 / 1 / 3
*****L2 - Priority Queue, FCFS for same priority *****
Demoted Process: dev // PID: 3, Allocated in L2[0]
[3] 2 / 0 / 3 / 0

```

A. yield(), getLevel()(PID:3)

- yield(), getlevel()을 순차적으로 호출 후, getlevel()은 그 결과를 출력합니다.
- yield()가 호출되는 것을 볼 수 있고, dev (pid:3)가 호출되면 현재 있는 level 이 1이라고 나와있습니다.
- 그리고 중간에 L2를 프린트하기 직전, pid:3이 L1에서 주어진 time quantum을 다 사용하여 L1 -> L2로 내려온 것을 확인할 수 있습니다. 위와 같은 이유로 L2에도 해당 pid:3이 출력되게 됩니다.

B. setPriority() (pid:4)

```

int main(int argc, char* argv[]){
    setPriority(1,1);
    printmlfq();
    exit();
}

$ dev
=====CURRENT MLFQ STATUS=====
Demoted Process: dev // PID: 4, Allocated in L1[0]
MLFQ STATE: UNLOCKED
[PID] level / index / priority / (arrived: for L2)
*****L0 - RR, Mostly Prioritized. *****
[1] 0 / 0 / 1
[2] 0 / 1 / 3
*****L1 - RR, Took a backseat to L0. *****
[4] 1 / 0 / 3
*****L2 - Priority Queue, FCFS for same priority *****

int main(int argc, char *argv[]){
    setPriority(1,5);
    printmlfq();
    exit();
}

$ dev
=====CURRENT MLFQ STATUS=====
MLFQ STATE: UNLOCKED
[PID] level / index / priority / (arrived: for L2)
*****L0 - RR, Mostly Prioritized. *****
[2] 0 / 0 / 3
[1] 0 / 1 / 3
[3] 0 / 2 / 3
*****L1 - RR, Took a backseat to L0. *****
*****L2 - Priority Queue, FCFS for same priority *****

```

B. setPriority() (PID:4)

- dev에서는 pid:1의 priority 를 1로 변경하고 있습니다.
- printmlfq()를 한 결과, 다음과 같이 pid:1의 priority 가 1로 된 것을 확인할 수 있습니다.
- 범위 외의 priority (0보다 작거나 3보다 큼)를 인자로 전달하면, 해당 system call은 무시됩니다.

Result #3) schedulerLock()

A. schedulerLock() – Password Incorrect (PID: 4)

```
int main(int argc, char* argv[]){
    schedulerLock(12341234);
    printmlfq();
    exit();
}
$ dev
REJECT: Password incorrect, forcing to stop current process...
[REJECTED PROCESS] Pid: 4 / Elapsed Time Quantum: 3 / Level: 0
```

B. schedulerLock() – Successful Call (PID: 3)

```
int main(int argc, char* argv[]){
    schedulerLock(2019014266);
    printmlfq();
    exit();
}

int main(int argc, char* argv[]){
    __asm__("int $129");
    printmlfq();
    exit();
}
$ dev
SCHEDULER LOCKED! - PID: 3
=====CURRENT MLFQ STATUS=====
MLFQ STATE: LOCKED [PID: 3]
[PID] level / index / priority / (arrived: for L2)
*****L0 - RR, Mostly Prioritized. *****
[1] 0 / 0 / 3
[2] 0 / 1 / 3
*****L1 - RR, Took a backseat to L0. *****
*****L2 - Priority Queue. FCFS for same priority *****
$ printmlfq
Demoted Process: printmlfq // PID: 4, Allocated in L1[0]
=====CURRENT MLFQ STATUS=====
MLFQ STATE: UNLOCKED
[PID] level / index / priority / (arrived: for L2)
*****L0 - RR, Mostly Prioritized. *****
[1] 0 / 0 / 3
[2] 0 / 1 / 3
*****L1 - RR, Took a backseat to L0. *****
[4] 1 / 0 / 3
*****L2 - Priority Queue. FCFS for same priority *****
Demoted Process: sh // PID: 2, Allocated in L1[0]
```

C. schedulerLock() – Multiple Lock (PID: 3)

```
int main(int argc, char* argv[]){
    schedulerLock(2019014266);
    schedulerLock(2019014266);
    printmlfq();
    exit();
}
$ dev
Demoted Process: sh // PID: 2, Allocated in L1[0]
SCHEDULER LOCKED! - PID: 3
IGNORE: Scheduler already locked! Ignoring...
=====CURRENT MLFQ STATUS=====
MLFQ STATE: LOCKED [PID: 3]
[PID] level / index / priority / (arrived: for L2)
*****L0 - RR, Mostly Prioritized. *****
*****L1 - RR, Took a backseat to L0. *****
[2] 1 / 0 / 3
*****L2 - Priority Queue. FCFS for same priority *****
[1] 2 / 0 / 3 / 0
```

A. schedulerLock() – Password Incorrect (PID: 4)

- schedulerLock()을 잘못된 비밀번호로 호출할 경우, Reject을 합니다. 해당 프로세스를 종료시키고 pid, time quantum, level을 출력합니다.

B. schedulerLock() – Successful Call (PID: 3)

- dev는 schedulerLock()을 알맞은 비밀번호로 호출하거나, 129번 interrupt를 호출하고 있습니다.
- 그 결과, MLFQ Scheduler는 Lock하게 되고, MLFQ에서는 제거되게 됩니다. printmlfq() 호출 시 MLFQ가 잠겼을 시, schedulerLock()을 호출한 pid를 출력합니다.
- 프로세스가 종료된 후, printmlfq() (PID:4)를 통해 다시 MLFQ를 출력했습니다. schedulerUnlock()을 호출하지 않았지만, dev()가 RUNNABLE하지 못하다고 판단하여 scheduler workflow하에 lock을 해제한 모습입니다.

C. schedulerLock() – Multiple Lock (PID: 3)

- 중복 호출의 경우 해당 system call은 Ignore, 무시하고 계속 진행합니다.

Result #4) schedulerUnlock()

A. schedulerUnlock() – Password Incorrect (PID: 3)

```
int main(int argc, char* argv[]){
    schedulerLock(2019014266);
    schedulerUnlock(12341234);
    printmlfq();
    exit();
}
$ dev
SCHEDULER LOCKED! - PID: 3
REJECT: Password incorrect, forcing to stop current process...
[REJECTED PROCESS] Pid: 3 / Elapsed Time Quantum: 1 / Level: 1
```

A. schedulerUnlock() – Password Incorrect (PID: 3)

- schedulerUnlock()을 잘못된 비밀번호로 호출할 경우, Reject을 합니다. 해당 프로세스를 종료시키고 pid, time quantum,

B. schedulerUnlock() – Successful Call (PID: 3)

```

int main(int argc, char* argv[]){
    schedulerLock(2019014266);
    schedulerUnlock(2019014266);
    printmlfq();
    exit();
}

int main(int argc, char* argv[]){
    schedulerLock(2019014266);
    __asm__("int $130");
    printmlfq();
    exit();
}

```

\$ dev
Demoted Process: dev // PID: 3, Allocated in L1[0]
SCHEDULER LOCKED! – PID: 3
=====CURRENT MLFQ STATUS=====
MLFQ STATE: UNLOCKED
[PID] level / index / priority / (arrived: for L2)
*****L0 – RR, Mostly Prioritized. *****
[3] 0 / 0 / 3
[2] 0 / 1 / 3
[1] 0 / 2 / 3
*****L1 – RR, Took a backseat to L0. *****
*****L2 – Priority Queue. FCFS for same priority *****

C. schedulerLock() – Without Lock (PID: 3)

```

int main(int argc, char* argv[]){
    schedulerUnlock(2019014266);
    printmlfq();
    exit();
}

$ dev  
Demoted Process: sh // PID: 2, Allocated in L1[0]  
Demoted Process: dev // PID: 3, Allocated in L1[1]  
IGNORE: Scheduler is not locked! Ignoring...
=====CURRENT MLFQ STATUS=====  
MLFQ STATE: UNLOCKED  
[PID] level / index / priority / (arrived: for L2)  
*****L0 – RR, Mostly Prioritized. *****  
[1] 0 / 1 / 3  
*****L1 – RR, Took a backseat to L0. *****  
[2] 1 / 0 / 3  
[3] 1 / 1 / 3  
*****L2 – Priority Queue. FCFS for same priority *****

```

- level을 출력합니다.
- B. schedulerUnlock() – Successful Call (PID: 3)
- dev는 schedulerUnlock()을 알맞은 비밀번호로 호출하거나, 130번 interrupt를 호출하고 있습니다.
 - 그 결과, schedulerLock()을 호출한 Dev는 다시 해당 scheduler를 Unlock하게 되고, 다시 MLFQ Scheduler가 동작하게 합니다.
 - 해당 프로세스는 L0의 가장 첫번째 자리에 들어가 있는 것을 확인해 볼 수 있습니다.

C. schedulerUnlock() – Without Lock (PID: 3)

- schedulerLock()을 호출한 프로세스가 없을 때 schedulerUnlock()을 호출할 경우, 해당 system call은 Ignore, 무시하고 계속 진행합니다.

Result #5) Piazza Test Case

A. fork children() – Default Test

```

$ dev
MLFQ test start
[Test 1] default
Process 4
L0: 9547
L1: 14875
L2: 75578
L3: 0
L4: 0
Process 5
L0: 10839
L1: 17416
L2: 71745
L3: 0
L4: 0
Process 7
L0: 14146
L1: 21453
L2: 64401
L3: 0
L4: 0
Process 6
L0: 16410
L1: 24422
L2: 59168
L3: 0
L4: 0
[Test 1] finished
done

```

- Piazza에 올라온 Test Case의 결과입니다. 제 환경에 맞게 system call을 변경 후 실행했습니다.
- 해당 코드를 위에서 사용하던 dev.c에 옮겼으며, 깔끔한 결과 확인을 위해 모든 printf는 comment 처리했습니다.
- 수정사항:
 - getlev() => getLevel()
 - setpriority() => setPriority()
 - int setpriority() => void setPriority()
 - r = setpriority(p,i) => setPriority(p,i)

```
B. fork children2() - setPriority
$ dev
MLFQ test start
[Test 2] setPriority
Process 4
L0: 5052
L1: 9810
L2: 85138
L3: 0
L4: 0
Process 7
L0: 9285
L1: 15512
L2: 75203
L3: 0
L4: 0
Process 5
L0: 13352
L1: 20613
L2: 66035
L3: 0
L4: 0
Process 6
L0: 16261
L1: 25922
L2: 57817
L3: 0
L4: 0
[Test 2] finished
done
```

- v. if(r<0) => removed.
- vi. printf(1, "setpriority returned%d\n", r); => printf(1, "setpriority returned \n");

```
C. fork children3() – maxLevel
$ dev
MLFQ test start
[Test 3] maxlevel
Process 4
L0: 4924
L1: 9300
L2: 85776
L3: 0
L4: 0
Process 6
L0: 9528
L1: 16282
L2: 74190
L3: 0
L4: 0
Process 7
L0: 13704
L1: 20809
L2: 65487
L3: 0
L4: 0
Process 5
L0: 15620
L1: 25038
L2: 59342
L3: 0
L4: 0
[Test 3] finished
done
```

VI. Trouble Shooting & Dissatisfaction

Project #1을 진행하며 많은 문제를 직면했었고, 이를 해결하는 과정이 인상적이었거나 기억에 남았던 순간을 서술하였습니다. 완벽하게 해결하지 못한 것은 Solution 대신 Dissatisfaction에 어떤 문제가 있었고, 어떤 경우에 어떤 문제가 발생하는지 서술하였습니다.

Trouble #1) L1 스케줄링 도중 L0에 새로운 프로세스가 도착했을 시에 scheduling 문제.

Explanation	<ul style="list-style-type: none"> - 이전 디자인의 문제점은, L1에서 스케줄링 도중 L0에 새로운 프로세스가 도착했을 시 바로 L0로 넘어가지 못하고 L1를 모두 scheduling한 후에야 L0를 scheduling한다는 문제가 있었습니다.
Solution	<ul style="list-style-type: none"> - 매 scheduling 직전, retlevel()를 사용하여 새로운 프로세스의 유무를 판단합니다. 만약 L0에 새로운 프로세스가 왔으면 level과 L0의 index를 불러와 L0에서의 scheduling을 진행합니다. - 이때 새로운 프로세스는 L0의 첫번째 자리에 위치해 있을 가능성이 높음으로 scheduling 시작 index를 영으로 초기화 후 진행합니다. (모든 프로세스는 각 queue의 첫번째로 발견되는 빈 자리에 들어옵니다.) - 해당 방법을 응용하여 어떤 프로세스가 schedulerLock() 호출했을 시에도 바로 해당 최우선순위 프로세스를 scheduling할 수 있습니다.

Trouble #2) Round-Robin의 Scheduling 순서의 문제 (1)

Explanation	<ul style="list-style-type: none"> - L1의 4번째까지 scheduling한 후, L0에 새로운 프로세스가 들어오면 L0에서의 scheduling이 우선되게 됩니다. - L0에서의 scheduling후, L1으로 다시 돌아왔을 때 첫번째 index에서 시작하면 나중에 온 프로세스가 L1의 5번째 프로세스보다 일찍 실행될 수도 있습니다.
Solution	<ul style="list-style-type: none"> - L0, L1의 경우 index를 항상 기억하며, scheduling시 포인터로 접근하여 index값을 업데이트 합니다. - 이렇게 될 경우, L1으로 다시 돌아왔을 때 index를 불러오면 마지막으로 scheduling했던 4번째 이후의 번호인 5를 불러오게 되면서 5번째부터 계속 scheduling할 수 있게 됩니다. - 위와 같은 로직은 아무리 L1 빈 첫번째 자리에 프로세스가 와도, 마지막으로 scheduling이 된 index보다 늦게 scheduling되는 것을 보장합니다.

Trouble #3) schedulerLock()을 호출한 프로세스의 yield() 호출 문제.

Explanation	<ul style="list-style-type: none"> - schedulerLock()을 호출한 프로세스는 100 tick의 time quantum을 가집니다. 하지만 중간에 state가 RUNNING, RUNNABLE이 아닌 SLEEPING이나 ZOMBIE로 바뀌게 되면 자동으로 Lock을 해제합니다. - 다만, 이 경우 schedulerLock()을 호출한 프로세스 내부에서 yield()를 호출할 시 해당 yield()는 무시된다는 (RUNNABLE일 경우 프로세스를 넘겨주지 않고 계속 점유.) 문제가 발생합니다.
-------------	--

	<ul style="list-style-type: none"> - RUNNABLE인 상태 또한 Lock을 해제하게 하면 매 tick마다 Lock이 해제되기 때문에 (timer interrupt로 인한 yield()) schedulerLock()의 의미가 사라집니다.
Solution	<ul style="list-style-type: none"> - 이는 코드 구현상의 해결방법이 아닌, 제 MLFQ 디자인에 변화를 주었습니다. - schedulerLock()을 호출한 프로세스(lockedproc)는 그 어떤 프로세스보다도 우선순위를 가지고 있으며, yield()와 같은 system call을 사용하기 위해서는 반드시 schedulerUnlock()을 한 이후에 사용을 해야 한다는 규칙을 만들었습니다. - yield() 호출 시 lock이 풀린다는 전제는 이 또한 schedulerUnlock()의 기능을 하는 것과 다름이 없고, schedulerUnlock()을 호출하지 않고 다음 프로세스에게 yield() 요청을 할 경우, 다음 scheduling 대상이 최고 우선순위인 lockedproc이기 때문에, 어차피 같은 프로세스가 scheduling되게 됩니다. - 즉, schedulerUnlock()을 호출하지 않고 yield()을 호출할 경우, 같은 lockedproc이 scheduling되게 되고, 다른 프로세스에게 CPU를 양보하고 싶으면 반드시 schedulerUnlock()을 호출해야 합니다.

Trouble #4) Round-Robin의 Scheduling 순서의 문제 (2)

Explanation	<ul style="list-style-type: none"> - 새로운 프로세스가 형성이 될 때, L0에서 가장 먼저 발견되는 빈 공간에 해당 프로세스가 할당되게 됩니다. 이 경우, 다음과 같은 경우를 생각해볼 수 있습니다. <ol style="list-style-type: none"> a. L0에 0번에서 5번까지 프로세스가 할당되어 있으며, 3번까지는 각자의 작업 일부만을 완료하여 yield()를 호출한 상태입니다. b. 4번 프로세스는 L0에서 time quantum 내에 모든 작업을 완료하여 exit하게 됩니다. 4번 위치는 빈 공간이 됩니다. c. 5번 프로세스는 최근에 들어와 작업의 일부를 수행하고 yield()를 호출했습니다. d. 이때, 새로운 프로세스 6번이 들어옵니다. 해당 프로세스는 L0에서 가장 먼저 발견되는 빈 공간인 4번에 들어오게 됩니다. e. 이 경우, L0에 프로세스가 남아있어 다시 scheduling할 시, 0번에서 3번까지는 이전 순서와 똑같이 scheduling이 되지만, 나중에 들어온 6번 프로세스는 먼저 들어온 5번 프로세스보다 먼저 실행되게 됩니다. - 명세에는 RR는 반드시 FCFS policy를 따라야 한다고 명시는 안되어 있지만, piazza에서 scheduling의 순서를 보장해야 한다는 말이 이 말과 어느정도 맞는 부분이 있어 해결하려고 시도했습니다.
Dissatisfaction	<ul style="list-style-type: none"> - nullifylock()에서와 같이 프로세스가 exit, Unused상태로 돌아갈 때마다 빈 공간을 없애는 방법을 구현하여 해결 시도를 해보았습니다. - 하지만 이 경우, 아주 적은 확률이긴 하지만 프로세스가 복사되어 형성될 때가 있었습니다. 이는 다른 nullifylock()과 달리, exit은 굉장히 자주 호출되는 system call이기 때문이었습니다.

	<p>다.</p> <ul style="list-style-type: none"> - Exit이 연속되어 호출될 경우, 프로세스의 빈칸을 채우려 복사한 후 기존 데이터를 지우는 도중 interrupt가 들어와 지우는 작업을 생략해서 생기는 버그였습니다. 적은 확률이긴 하지만 그 결과가 꽤 치명적이어서 roll-back했습니다. - 과제의 남은 기간과 아직 구현을 못한 중요한 system call등이 급하여 해당 문제는 완벽하게 해결하지 못했습니다. 또한 이 dissatisfaction은 다음과 같은 관점에서 감점의 대상이 되지 않다고 판단하여 넘어간 것도 있습니다. 그 이유는 다음과 같습니다. <ol style="list-style-type: none"> A. Explanation에 나온 경우는 드문 경우에 생기는 일입니다. 자주 일어나는 case가 아닙니다. B. 기존 xv6의 Round-Robin scheduler에서도 해당 문제가 존재합니다. L0, L1의 Round-Robin 작동 방법은 xv6의 것을 그대로 차용한 것입니다. C. 무엇보다도, 명세에 Round-Robin이 FCFS policy를 따라야 한다는 내용이 없었습니다. 기본 Round-Robin policy를 따르면 된다고 명시되어 있으며, Round-Robin은 FCFS와 같은 priority가 없는 scheduling policy입니다. 해당 조건은 잘 만족하고 있기 때문에 명세의 내용을 충실히 이행한 것으로 볼 수 있습니다. - 다만, 제게 조금만 더 여유와 시간이 있었다면 해결을 할 수 있었던 문제 같아 아쉬움이 살짝 남습니다. ☺
--	---

이상입니다. 구현하면서 힘들지 않았다고 하면 거짓말이겠지만, 솔직히 재밌기도 했습니다. 어느새 제 MLFQ에 애정이 생겨 최대한 자세히 설명하느라 분량이 약 30페이지가 되지만, 그래도 제 프로젝트를 잘 설명한 것 같아 마음에 듭니다. ☺ Piazza에 올라온 test case를 모두 잘 통과하는 것으로 보았을 때, 잘 구현한 것으로 판단하여 구현을 마치고 과제를 제출하겠습니다.