Project01 wiki

> 2021011158 김선희

FCFS 구현

xv6의 기본 RR scheduler 동작 방식

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

  c->proc = 0;
  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();

    int found = 0;
    for(p = proc; p < &proc[NPROC]; p++) {
      acquire(&p->lock);
      if(p->state == RUNNABLE) {
        // Switch to chosen process.  It is the process's job
        // to release its lock and then reacquire it
        // before jumping back to us.
        p->state = RUNNING;
        c->proc = p;
        swtch(&c->context, &p->context);

        // Process is done running for now.
        // It should have changed its p->state before coming back
        c->proc = 0;
        found = 1;
      }
      release(&p->lock);
    }
    if(found == 0) {
      // nothing to run; stop running on this core until an interrupt
      intr_on();
      asm volatile("wfi");
    }
  }
}
```

1. process table인 `proc` 배열을 처음부터 끝까지 순회하며, runnable한 프로세스를 찾는다.
1. runnable한 프로세스를 발견하면 바로 `swtch()`를 호출해서 해당 프로세스를 실행한다. runnable한 프로세스들은 순차적으로 돌아가면서 실행된다.

Design

RR scheduler vs. FCFS scheduler

- FCFS는 RR과 다르게 **우선 순위**가 있는 algorithm이다.

  → **earliest creation time을 가지는 프로세스를 가장 먼저 실행**한다.

  ; 따라서 FCFS는 가장 먼저 생성된 프로세스를 가장 먼저 실행하는 로직이 추가되어야 한다. **가장 먼저 생성된 프로세스는 pid가 가장 작은 프로세스!**

  ⇒ ptable에서 runnable한 프로세스 중 pid가 가장 작은 것을 찾아 먼저 실행되도록 하면 될 것이다.

FCFS scheduler 구현 방식

1. runnable한 프로세스 배열 전체를 순회하여 가장 pid가 작은 proces를 선택한다.
2. 선택된 프로세스를 새로 실행하기 위한 context switch를 수행한다.

Implementation

📌 **구현 및 수정한 파일**

```
kernel
|-proc.c
```

proc.c

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

  c->proc = 0;
  for(;;){
    intr_on();

    struct proc *running_p = 0;

    // 가장 작은 PID를 가진 RUNNABLE 프로세스를 선택
    for(p = proc; p < &proc[NPROC]; p++) {
      acquire(&p->lock);
      if(p->state == RUNNABLE) {
        if(running_p == 0 || p->pid < running_p->pid) {
          running_p = p;
        }
      }
      release(&p->lock);
    }

        // 선택된 프로세스를 실행
    if(running_p) {
      acquire(&running_p->lock);
      running_p->state = RUNNING;
      c->proc = running_p;
      swtch(&c->context, &running_p->context);

      // 프로세스 실행 종료 후 처리
      c->proc = 0;
      release(&running_p->lock);
    } else {
      // 실행할 게 없으면 대기
      intr_on();
      asm volatile("wfi");
    }
  }
}
```

Result

주어진 test 파일을 실행한 결과

```
[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
```

MLFQ 구현

Design

MLFQ

Queue는 논리적으로 구현한다

1. `proc` **구조체 구성**

proc.h

```
// proc 구조체 내부
int priority;     // 프로세스의 우선순위 (0~3), L2에서만 사용
int level;        // 현재 프로세스가 속한 MLFQ 레벨 (0~2)
int tq;           // 현재까지 사용한 time quantum
```

어떤 프로세스를 선택할지 결정하는 데 time quantum, queue level, priority에 대한 정보가 필요하므로, proc 구조체에 해당 변수들을 추가하여 논리적으로 queue를 구현했다.

2. **처음 실행된 process의 변수 초기화**

proc.c

```
static struct proc*
allocproc(void)
{
   found:
  ...
  p->pr = 3;
  p->lv = 0;
  p->tq = 0;
  p->time = ticks;
  ...
}
```

`allocproc()` 함수는 process 배열 (process table)에서 `UNUSED` 상태인 process를 찾아 `proc` 구조체와 kernel stack을 할당해주는 역할을 한다.

→ `proc` 구조체 내에 새로 추가한 변수에 대한 초기화 과정도 이 함수 안에 추가해야 한다.

process 결정 방식

: ptable 전체를 한 번 순회하여 가장 우선적으로 처리해야 할 프로세스를 고른다.

ptable을 순회하면서 가장 우선순위가 높은 process를 선택한다.

proc.c

```
// 1-1. lv이 낮은 프로세스부터 선택
if(running_p->lv > p->lv) {
  running_p = p;
}
// 1-2. lv이 같을 때
else if(running_p->lv == p->lv) {
  // L2: pr이 큰 프로세스부터 선택
```

1. `lv` 비교 ⇒ `lv` 이 더 작은 프로세스 선택
   a. 같은 `lv` 이면서 그것이 2가 아니라면, `time` 이 작은 프로세스 선택
2. `pr` 비교 : `lv==2` 로 같다면, `pr` 을 비교한다 ⇒ `pr` 이 더 큰 프로세스 선택
   a. `pr` 마저 같으면, queue에 도착한 시간 (`time`)이 더 이른 프로세스 선택 ⇒ starvation 방지

```
  if(running_p->lv == 2) {
   if(running_p->pr < p->pr) {
    running_p = p;
   } else if((running_p->pr == p->pr) && (running_p->tq > p->time)) {
    running_p = p;
   }
  }
  // L0, L1: 도착한지 오래된 process, 즉 time (도착 당시의 tick 값)이 작은 것을
  else {
   if(running_p->time > p->time) {
    running_p = p;
   }
  }
 }
```

scheduler 구현 방식

1. 위에서 설명한 방식을 따라 process를 찾는다.
2. process를 가지고 context switch를 수행한다.

Timer interrupt

- Timer interrupt를 보는 이유

  : time quantum 처리와 priority boosting이 `ticks`와 관련이 있기 때문이다.

Timer interrupt 기본 동작

> timer interrupt는 `trap.c` 파일에서 처리한다.

user trap 처리

```
//
// handle an interrupt, exception, or system call from user spac
// called from trampoline.S
//
void
usertrap(void)
{
  int which_dev = 0;

  ...
  else if((which_dev = devintr()) != 0){
    // ok
  }
  ...

  // give up the CPU if this is a timer interrupt.
  if(which_dev == 2)
    yield();
}
```

kernel trap 처리

```
// interrupts and exceptions from kernel code go here via kern
// on whatever the current kernel stack is.
void
kerneltrap()
{
  int which_dev = 0;
  ...
  if((which_dev = devintr()) == 0){
    ...
  }

  // give up the CPU if this is a timer interrupt.
  // 기존 코드
  if(which_dev == 2 && myproc() != 0)
    yield();
  ...
}
```

`trap.c`에서는 user space에서 일어난 trap과 kernel space에서 일어나는 trap을 각각 `usertrap()`, `kerneltrap()`에서 따로 처리한다.

두 함수의 timer interrupt 처리 과정은 동일하다. 먼저 `devintr()`라는 함수를 호출하는데, 이것은 발생한 interrupt의 종류가 무엇인지에 따라 return 값을 결정한다. timer interrupt의 경우, `devintr()`의 return값은 2이다.

`devintr()`에서 `clockintr()`을 호출하여 global tick에 해당하는 변수인 `ticks`를 하나 더해주고, 각 trap 함수 안에서 `yield()`하는 방식으로 timer interrupt를 처리하고 있다. `yield()`를 통해 현재 프로세스가 cpu를 포기하고 스케줄러를 다시 호출한다. 따라서 timer interrupt가 발생하고 스케줄러로 돌아가기 전인 바로 이 시점(`if(which_dev == 2)` 내부)에 프로세스의 time quantum, level, priority을 처리하고자 한다.

Timer interrupt 발생 시, time quantum/level/priority 처리

각 trap 처리 함수 (`usertrap()` , `kerneltrap()`)의 timer interrupt 처리 if문 안에 다음 코드를 추가한다.

```
if(which_dev == 2){
  struct proc *p = myproc();

  p->tq++;

  if(p->tq == (p->lv)*2 + 1){
    if(p->lv == 2){
      setpriority(p->pid, (p->pr) - 1);
    }
    else{
      p->lv++;
    }

    p->tq = 0;
  }

  yield();
}
```

`yield()` 전에 time quantum을 하나 증가시키고, 늘어난 time quantum을 기준으로 level과 priority를 처리한다. 현재 프로세스의 time quantum (`tq`)이 해당 레벨의 정해진 time quantum (`(p->lv)*2 + 1`)과 같아졌을 때, 이를 다 소진한 것으로 생각하고 그에 따른 조치를 취한다. `lv==2` 면 priority 값 (`pr`)을 하나 감소시키고, `lv≠2` 면 level 값 (`lv`)을 하나 증가시킨다.

Priority boosting

global tick이 50이 될 때마다 모든 프로세스를 L0로 이동시키고, time quantum과 priority를 각각 초기값인 0과 3으로 세팅해야 한다.

따라서 global tick이 증가하는 부분을 찾아보니, `trap.c` 의 `clockintr()` 에서 처리되고 있었다.

```
clockintr()
{
  if(cpuid() == 0){
    acquire(&tickslock);
    ticks++; // 바로 여기!
    wakeup(&ticks);
    release(&tickslock);
  }

  // ask for the next timer interrupt. this also clears
  // the interrupt request. 1000000 is about a tenth
  // of a second.
  w_stimecmp(r_time() + 1000000);
}
```

Priority boosting 실행 코드

trap.c

```
void
clockintr()
{
...
    if (ticks == 50) { // tick이 50이 될 때마다
      priority_boosting();
      ticks = 0;
    }
...
}
```

proc.c

```
void
priority_boosting(void){
  struct proc *p;

  for(p = proc; p < &proc[NPROC]; p++) {
    p->pr = 3;
    p->lv = 0;
    p->tq = 0;
    p->time = ticks;
  }
}
```

Implement

MLFQ Scheduler

proc.c

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

  c->proc = 0;

  for(;;) {
    intr_on();

    struct proc *running_p = 0;

    // 1. 가장 우선적으로 처리해야 할 프로세스 선택
    for(p = proc; p < &proc[NPROC]; p++) {
      acquire(&p->lock);

      if(p->state == RUNNABLE) {
        if(running_p == 0) {
          running_p = p;
        } else {
          // 1-1. lv이 낮은 프로세스부터 선택
          if(running_p->lv > p->lv) {
            running_p = p;
          }
          // 1-2. lv이 같을 때
          else if(running_p->lv == p->lv) {
            // L2: pr이 큰 프로세스부터 선택
            if(running_p->lv == 2) {
              if(running_p->pr < p->pr) {
                running_p = p;
              } else if((running_p->pr == p->pr) && (running_p->tq > p->time)) {
                running_p = p;
              }
            }
            // L0, L1: 도착한지 오래된 process, 즉 time (도착 당시의 tick 값)이 작은 것을 먼저 실행
            else {
              if(running_p->time > p->time) {
                running_p = p;
              }
            }
          }
        }
      }

      release(&p->lock);
    }

    // 2. 선택된 프로세스를 실행
    if(running_p) {
      //printf("pid: %d | lv: %d | tq: %d | pr: %d | ticks: %d \n", running_p->pid, running_p->lv, running_p->tq, running_p->pr, ticks);

      acquire(&running_p->lock);
      running_p->state = RUNNING;
      c->proc = running_p;
      swtch(&c->context, &running_p->context);
```

```
    // 프로세스 실행 종료 후 처리
    c->proc = 0;
    release(&running_p->lock);
   } else {
    // 실행할 게 없으면 대기
    intr_on();
    asm volatile("wfi");
   }

  }
 }
```

Timer interrupt

trap.c

```
void
usertrap(void)
{
  int which_dev = 0;

  if((r_sstatus() & SSTATUS_SPP) != 0)
    panic("usertrap: not from user mode");

  // send interrupts and exceptions to kerneltrap(),
  // since we're now in the kernel.
  w_stvec((uint64)kernelvec);

  struct proc *p = myproc();

  // save user program counter.
  p->trapframe->epc = r_sepc();

  if(r_scause() == 8){
    // system call

    if(killed(p))
      exit(-1);

    // sepc points to the ecall instruction,
    // but we want to return to the next instruction.
    p->trapframe->epc += 4;

    // an interrupt will change sepc, scause, and sstatus,
    // so enable only now that we're done with those registers.
    intr_on();

    syscall();
  } else if((which_dev = devintr()) != 0){
    // ok
  } else {
    printf("usertrap(): unexpected scause 0x%lx pid=%d\n", r_s
    printf("            sepc=0x%lx stval=0x%lx\n", r_sepc(), r_stval(
    setkilled(p);
  }

  if(killed(p))
    exit(-1);

  // give up the CPU if this is a timer interrupt.
  if(which_dev == 2){
```

```
void
kerneltrap()
{
  int which_dev = 0;
  uint64 sepc = r_sepc();
  uint64 sstatus = r_sstatus();
  uint64 scause = r_scause();

  if((sstatus & SSTATUS_SPP) == 0)
    panic("kerneltrap: not from supervisor mode");
  if(intr_get() != 0)
    panic("kerneltrap: interrupts enabled");

  if((which_dev = devintr()) == 0){
    // interrupt or trap from an unknown source
    printf("scause=0x%lx sepc=0x%lx stval=0x%lx\n", scause,
    panic("kerneltrap");
  }

  // give up the CPU if this is a timer interrup
  if(which_dev == 2  && myproc() != 0){
    struct proc *p = myproc();

    if(sched_mode == 1){
      p->tq++;

      if(p->tq == (p->lv)*2 + 1){
        if(p->lv == 2){
          setpriority(p->pid, (p->pr) - 1);
        }
        else{
          p->lv++;
          p->time = ticks;
        }

        p->tq = 0;
      }
    }

    yield();
  }

  // the yield() may have caused some traps to occur,
  // so restore trap registers for use by kernelvec.S's sepc instr
  w_sepc(sepc);
```

```
    if(sched_mode == 1){
      p->tq++;

      if(p->tq == (p->lv)*2 + 1){
        if(p->lv == 2){
          setpriority(p->pid, (p->pr) - 1);
        }
        else{
          p->lv++;
          p->time = ticks;
        }

        p->tq = 0;
      }
    }

    yield();
  }

  usertrapret();
}
```

```
  w_sstatus(sstatus);
}
```

Priority boosting

trap.c

```
void
clockintr()
{
...
    if (ticks == 50) { // tick이 50이 될 때마다
      priority_boosting();
      ticks = 0;
    }
...
}
```

proc.c

```
void
priority_boosting(void){
  struct proc *p;

  for(p = proc; p < &proc[NPROC]; p++) {
    p->pr = 3;
    p->lv = 0;
    p->tq = 0;
    p->time = 0;
  }
}
```

Result

📌 process 정보 print 시점

: scheduler에서 프로세스가 선택된 후, context switch를 실행하기 직전 시점.

proc.c

```
...
        // 선택된 프로세스를 실행
      if(running_p) {
        printf("pid: %d | lv: %d | tq: %d | pr: %d | ticks: %d \n", running_p->pid, running_p->lv, running_p->tq, running_p->pr, ticl
        acquire(&running_p->lock);
        running_p->state = RUNNING;
        c->proc = running_p;
        swtch(&c->context, &running_p->context);
            ...
        }
    ...
```

MLFQ

- 주어진 test 파일을 실행한 결과:

```
[Test 2] MLFQ Scheduling
Process 8 (MLFQ L0-L2 hit count):
L0: 3337
L1: 17514
L2: 79149
Process 9 (MLFQ L0-L2 hit count):
L0: 5755
L1: 20515
L2: 73730
Process 10 (MLFQ L0-L2 hit count):
L0: 6668
L1: 20305
L2: 73027
Process 11 (MLFQ L0-L2 hit count):
L0: 991
L1: 37499
L2: 61510
[Test 2] MLFQ Test Finished
```

Priority boosting

- 주어진 test 파일을 실행한 결과:

```
pid: 8 | lv: 2 | tq: 4 | pr: 2 | time: 7 | ticks: 40
pid: 9 | lv: 2 | tq: 0 | pr: 2 | time: 10 | ticks: 41
pid: 9 | lv: 2 | tq: 1 | pr: 2 | time: 10 | ticks: 42
pid: 9 | lv: 2 | tq: 2 | pr: 2 | time: 10 | ticks: 43
pid: 9 | lv: 2 | tq: 3 | pr: 2 | time: 10 | ticks: 44
pid: 9 | lv: 2 | tq: 4 | pr: 2 | time: 10 | ticks: 45
pid: 10 | lv: 2 | tq: 0 | pr: 2 | time: 13 | ticks: 46
pid: 10 | lv: 2 | tq: 1 | pr: 2 | time: 13 | ticks: 47
pid: 10 | lv: 2 | tq: 2 | pr: 2 | time: 13 | ticks: 48
pid: 10 | lv: 2 | tq: 3 | pr: 2 | time: 13 | ticks: 49
pid: 8 | lv: 0 | tq: 0 | pr: 3 | time: 0 | ticks: 0
pid: 9 | lv: 0 | tq: 0 | pr: 3 | time: 0 | ticks: 1
```

pid가 8, 9인 프로세스의 level, time quantum, priority 등을 보면, MLFQ mode에서 tick이 50이 될 때마다 priority boosting이 일어나는 것을 확인할 수 있다.

Timer interrupt

- 주어진 test 파일을 실행한 결과:

```
pid: 8 | lv: 2 | tq: 2 | pr: 3 | ticks: 34
pid: 8 | lv: 2 | tq: 3 | pr: 3 | ticks: 35
pid: 8 | lv: 2 | tq: 4 | pr: 3 | ticks: 36
pid: 8 | lv: 2 | tq: 0 | pr: 2 | ticks: 37
pid: 8 | lv: 2 | tq: 1 | pr: 2 | ticks: 38
pid: 9 | lv: 2 | tq: 0 | pr: 2 | ticks: 39
pid: 9 | lv: 2 | tq: 1 | pr: 2 | ticks: 40
```

- process가 L2에 있을 때 주어진 time quantum인 5를 다 쓰면 priority가 하나 감소한다.

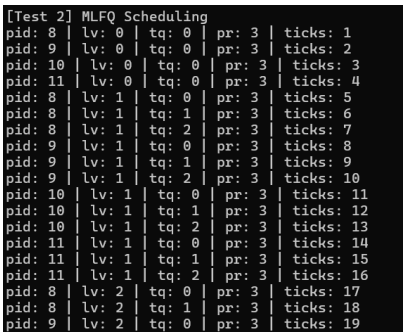
```
pid: 9 | lv: 1 | tq: 0 | pr: 3 | ticks: 8
pid: 9 | lv: 1 | tq: 1 | pr: 3 | ticks: 9
pid: 9 | lv: 1 | tq: 2 | pr: 3 | ticks: 10
```

- 프로세스가 cpu를 한 번 점유할 때마다 (동일 레벨 안에서) time quantum이 증가함을 확인할 수 있다.

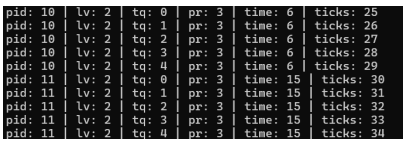
```
pid: 9 | lv: 1 | tq: 1 | pr: 3 | ticks: 9
pid: 9 | lv: 1 | tq: 2 | pr: 3 | ticks: 10
pid: 10 | lv: 1 | tq: 0 | pr: 3 | ticks: 11
pid: 10 | lv: 1 | tq: 1 | pr: 3 | ticks: 12
pid: 10 | lv: 1 | tq: 2 | pr: 3 | ticks: 13
pid: 11 | lv: 1 | tq: 0 | pr: 3 | ticks: 14
pid: 11 | lv: 1 | tq: 1 | pr: 3 | ticks: 15
pid: 11 | lv: 1 | tq: 2 | pr: 3 | ticks: 16
pid: 8 | lv: 2 | tq: 0 | pr: 3 | ticks: 17
pid: 8 | lv: 2 | tq: 1 | pr: 3 | ticks: 18
pid: 9 | lv: 2 | tq: 0 | pr: 3 | ticks: 19
pid: 9 | lv: 2 | tq: 1 | pr: 3 | ticks: 20
```

- pid가 9인 process를 보면, 한 번 cpu를 잡았을 때 time quantum이 주어진 3 (L1의 time quantum = 2 * 1 + 1 = 1)을 모두 소진하면서, L1→L2로 이동했음을 확인할 수 있다.

MLFQ의 동작 확인



- L0에 있는 프로세스를 모두 실행한 후, 남아 있는 프로세스가 없으면 L1, 그 다음으로 L2에 있는 프로세스를 실행하는 것을 확인할 수 있다.



- L2에서 pid가 10인 프로세스를 실행하다가 이것이 time quantum을 모두 소진하여 priority가 2로 감소하면서, priority가 3인 프로세스 (pid: 11)를 먼저 실행하는 것을 볼 수 있다. 이를 통해 L2의 경우, priority가 큰 프로세스가 우선순위를 갖는 것을 확인할 수 있다.

FCFS↔MLFQ switch mode 구현

Design

`sched_mode`라는 변수를 통해 scheduler mode를 표현하고자 했다.

- `sched_mode == 0` : FCFS mode
- `sched_mode == 1` : MLFQ mode

Implement

1. ***mode switch system call***

in proc.c

```
int
mlfqmode(void) {
  if(sched_mode == 1){ // 이미 mlfqmode인 경우
    return -1;
  }
  else{
    sched_mode = 1;

    ticks = 0;

    struct proc *p;

    for(p = proc; p < &proc[NPROC]; p++) {
      p->pr = 3;
      p->lv = 0;
      p->tq = 0;
      p->time = 0;
    }

    return 0;
  }
}
```

```
int
fcfsmode(void) {
  if(sched_mode == 0){ // 이미 fcfsmode인 경우
    return -1;
  }
  else{
    sched_mode = 0;

    ticks = 0;

    struct proc *p;

    for(p = proc; p < &proc[NPROC]; p++) {
      p->pr = -1;
      p->lv = -1;
      p->tq = -1;
      p->time = 0;
    }

    return 0;
  }
}
```

구현이 요구된 두 가지 mode switch system을 통해 sched_mode를 설정한다.

2. ***scheduler mode switch***

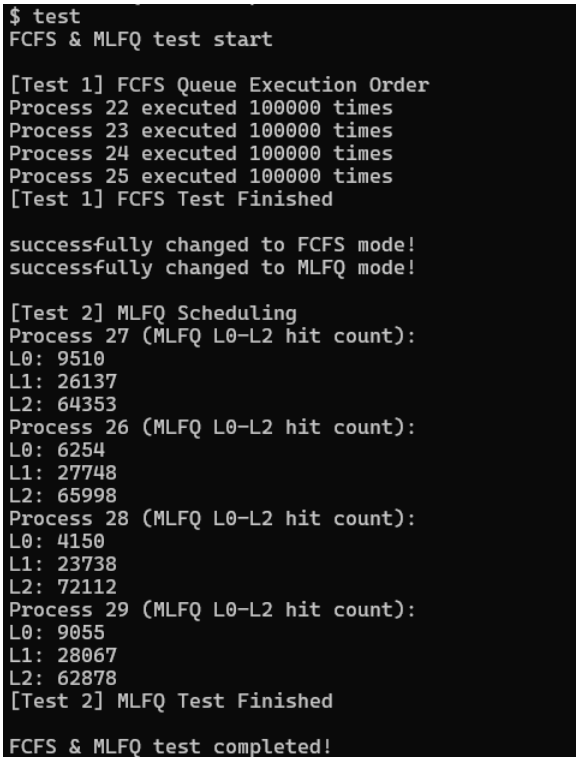
in proc.c: `scheduler()`

```
...
// FCFS mode
   if(sched_mode == 0){
    for(p = proc; p < &proc[NPROC]; p++) {
     acquire(&p->lock);
     if(p->state == RUNNABLE) {
      if(running_p == 0 || p->pid < running_p->pid) {
       running_p = p;
      }
     }
     release(&p->lock);
    }
   }
   // MLFQ mode
   else{
    for(p = proc; p < &proc[NPROC]; p++) {
     acquire(&p->lock);

     if(p->state == RUNNABLE) {
      if(running_p == 0) {
       running_p = p;
      } else {
       // 1-1. lv이 낮은 프로세스부터 선택
       if(running_p->lv > p->lv) {
        running_p = p;
       }
       // 1-2. lv이 같을 때
       else if(running_p->lv == p->lv) {
        // L2: pr이 큰 프로세스부터 선택
        if(running_p->lv == 2) {
         if(running_p->pr < p->pr) {
          running_p = p;
         } else if((running_p->pr == p->pr) && (running_p->tq > p->time)) {
          running_p = p;
         }
        }
        // L0, L1: 도착한지 오래된 process, 즉 time (도착 당시의 tick 값)이 작은 것을 먼저 실행
        else {
         if(running_p->time > p->time) {
          running_p = p;
         }
        }
       }
      }
     }

     release(&p->lock);
    }
   }
   ...
```

`sched_mode` 변수에 따라 프로세스 선택 방식을 결정한다.

Result

주어진 test 파일을 실행한 결과



Trouble Shooting

1. 같은 레벨에 있고, 같은 priority를 가지는 경우에 어떤 프로세스를 선택해야 할까?

   : process 구조체 안에 process가 해당 level에 들어온 시간을 값으로 갖는 time 변수를 추가했다. time 값이 작은, 해당 레벨에 도착한 시간이 이른 프로세스를 선택하도록 한다. ⇒ starvation 방지

   proc.h

   ```
   struct proc {
     ...
     int time         // 같은 queue 안에 있는 프로세스 사이 스케줄링을 돕는 변수
                      // process가 해당 level에 들어온 시간을 값으로 갖는다.
   }
   ```

2. FCFS mode에서는 priority boosting이 실행되지 않는다.

   ⇒ trap.c 내부 priority boosting을 진행하는 조건에 `sched_mode == 1` 을 추가하였다.

   ```
   void
   clockintr()
   {
   ...
     if (ticks == 50 && sched_mode == 1) { // MLFQ mode 안에서 tick이 50이 될 때마다
       priority_boosting();
       ticks = 0;
     }
   ...
   }
   ```

3. mode switch 구현

초기 구현

system call을 통해 scheduler mode를 결정하는 변수인 `sched_mode` 를 바꾸고, 이에 따라 `scheduler()` 함수 안에서 모드를 변경할 수 있게 하자

```
void
scheduler(void) {
 if(sched_mode == 0){ // fcfs mode
  fcfs_scheduler();
 }
 else{ // mlfq mode
  mlfq_scheduler();
 }
}
```

- `fcfs_scheduler()` , `mlfq_scheduler()` 는 각각 'FCFS 구현', 'MLFQ 구현' 파트에서 설명한 대로 구현한 scheduler 함수이다.

초기 구현의 문제점

`scheduler()` 함수는 초기에 main.c를 통해 한 번만 호출된다. 그 뒤에는 scheduler() 함수 호출을 통해 스케줄러가 동작하는 것이 아니라, `swtch()` 를 통해 scheduler로의 context switch를 통해 스케줄러가 동작한다. 그러므로 이 아이디어로는 mode switch가 제대로 일어날 수 없다.

(xv6-riscv book 참고)

The function `swtch` saves and restores registers for a kernel thread switch. `swtch` doesn't directly know about threads; it just saves and restores sets of RISC-V registers, called *contexts*. When it is time for a process to give up the CPU, the process's kernel thread calls `swtch` to save its own context and restore the scheduler's context. Each context is contained in a `struct context` (kernel/proc.h:2), itself contained in a process's `struct proc` or a CPU's `struct cpu`. `swtch` takes two arguments: `struct context *old` and `struct context *new`. It saves the current registers in `old`, loads registers from `new`, and returns.

Let's follow a process through `swtch` into the scheduler. We saw in Chapter 4 that one possibility at the end of an interrupt is that `usertrap` calls `yield`. `yield` in turn calls `sched`, which calls `swtch` to save the current context in `p->context` and switch to the scheduler context previously saved in `cpu->context` (kernel/proc.c:506).

최종 구현

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

 c->proc = 0;
 for(;;){
  intr_on();

  struct proc *running_p = 0;

  // FCFS mode
  if(sched_mode == 0){
   for(p = proc; p < &proc[NPROC]; p++) {
    acquire(&p->lock);
    if(p->state == RUNNABLE) {
     if(running_p == 0 || p->pid < running_p->pid) {
      running_p = p;
     }
    }
    release(&p->lock);
   }
  }
  // MLFQ mode
  else{
   for(p = proc; p < &proc[NPROC]; p++) {
    acquire(&p->lock);

    if(p->state == RUNNABLE) {
     if(running_p == 0) {
      running_p = p;
     } else {
```

```
          // 1-1. lv이 낮은 프로세스부터 선택
          if(running_p->lv > p->lv) {
            running_p = p;
          }
          // 1-2. lv이 같을 때
          else if(running_p->lv == p->lv) {
            // L2: pr이 큰 프로세스부터 선택
            if(running_p->lv == 2) {
              if(running_p->pr < p->pr) {
                running_p = p;
              } else if((running_p->pr == p->pr) && (running_p->tq > p->time)) {
                running_p = p;
              }
            }
            // L0, L1: 도착한지 오래된 process, 즉 time (도착 당시의 tick 값)이 작은 것을 먼저 실행
            else {
              if(running_p->time > p->time) {
                running_p = p;
              }
            }
          }
        }
      }

      release(&p->lock);
    }
  }


    // 선택된 프로세스를 실행
  if(running_p) {
    //printf("pid: %d | lv: %d | tq: %d | pr: %d | time: %d | ticks: %d \n", running_p->pid, running_p->lv, running_p->tq, running_p
    acquire(&running_p->lock);
    running_p->state = RUNNING;
    c->proc = running_p;
    swtch(&c->context, &running_p->context);

    // 프로세스 실행 종료 후 처리
    c->proc = 0;
    release(&running_p->lock);
  } else {
    // 실행할 게 없으면 대기
    intr_on();
    asm volatile("wfi");
  }
 }
}
```

선택된 프로세스를 실행하는 부분은 FCFS와 MLFQ의 공통된 부분이므로, 프로세스를 선택하는 부분이 `sched_mode`에 따라 바뀔 수 있도록 하나의 `scheduler()` 함수로 합치는 방향으로 수정했다.