

# ELE3021\_project01

운영체제 project01 wiki입니다.

## 과제 위치

**Project\_1/xv6-public**

노란색으로 표시된 디렉토리에 Project 1을 구현하였습니다.

master

2023\_ele3021\_2020005269 / +


History

Find file

Web IDE

Download

Clone

 **FIN: project1**  
Kim Sejeong authored 5 minutes ago

6a5ae27e

Auto DevOps enabled

Add README

Add LICENSE

Add CHANGELOG

Add CONTRIBUTING

Add Kubernetes cluster

Name	Last commit	Last update
.vscode	UDT: mlfq 알아엮고 임시 커밋	1 week ago
<b>Project_1/xv6-public</b>	<b>FIN: project1</b>	5 minutes ago
lab4_systemcall	FIX: lab4 - system call - comment fix	4 weeks ago
lab6-sync	CRET: mlfq_test 추가, 시스템콜 이름 변경, 내일 다시 고쳐야함	1 week ago
xv6-public	FIX: proc_ticks 초기화 오류 fix, mlfq 완성(?)	1 week ago

## 실행방법

Project\_1/xv6-public 디렉토리에서

```
make clean
./makemake.sh
./run-xv6.sh
```

를 셸에 입력하여 실행합니다.

컴파일, 실행 방법 예시:



- priority\_boosting이 일어나면 Ln 큐에 있는 모든 프로세스가 L0 큐로 이동하며 local\_tick = 0, priority = 3으로 초기화됩니다.
- 이동할 때는 `for(cur = ptable.proc; cur < *ptable.proc[NPROC]; cur++;)` 으로 ptable의 proc을 완전 탐색하여 L0 큐 안에 L0에 있던 프로세스, L1에 있던 프로세스, L2에 있던 프로세스 순으로 들어가도록 합니다.
- **priority boosting 시에 ptable을 완전 탐색하며 L0 큐로 이동시키므로, 이동된 후 L0 큐 내부의 순서가 이동 전 L0, L1, L2 내부의 순서를 보장하지 않을 수 있습니다.**

ex) priority\_boosting 직전에

L0: A, B

L1: C, D, E

L2: F, G

이러한 순서로 프로세스가 존재했다고 했을 때, priority boosting 후

L0: A, B(L0), **D, C, E(L1), G, F(L2)**

과 같이 순서가 섞이는 경우가 발생할 수 있습니다.

## SchedulerLock

- schedulerLock()이 호출되면, 해당 시스템 콜을 호출한(또는 인터럽트를 일으킨) 프로세스가 locked되어 최대 100tick 동안 cpu를 독점하여 사용합니다.
- ex) schedulerLock()을 호출한 시점의 ticks = 50이었다면 앞으로 50ticks 동안 cpu 독점 사용
- **lock이 걸려 있더라도 1tick 마다 yield()는 계속 발생합니다. 하지만 scheduler는 항상 해당 프로세스를 골라 스케줄 하도록 구현했습니다.**
- struct ptable에 `int locked_table` 이라는 변수를 추가하였습니다.
  - lock 걸리지 않은 상태: `locked_table == -1`
  - lock이 걸린 상태: `locked_table == (lock을 건 프로세스의 pid)`
- mlfq\_scheduler가 실행될 때, 만약 `ptable.locked_table != -1` 이라면, Ln 큐의 process를 고려하지 않고, locked\_table에 저장된 pid와 일치하는 프로세스를 찾아 그 프로세스를 schedule합니다.

## SchedulerUnlock

- schedulerUnlock()이 호출되면 lock이 걸려있던 프로세스가 해제되고, 다시 원래의 mlfq 방식으로 스케줄 됩니다.
- schedulerUnlock()이 호출되면
  - ptable.locked\_proc = -1;로 바꿔줍니다.
  - locked돼있던 프로세스를 찾아서 (ptable 완전 탐색하여 pid가 같은 프로세스를 찾아냄) 해당 프로세스를 L0 큐의 맨 앞(L0 내에서 가장 우선 순위가 높음)으로 보냅니다. 즉 그 프로세스의 q\_num = 0으로 합니다.
  - 해당 프로세스의 local tick = 0으로 해줍니다.

## Other system calls (yield, setPriority, getLevel)

- `yield()`: 기존 xv6의 `yield()` 함수를 그대로 쓰고, system call로만 만들어주었습니다.

다만, 시스템 콜로 `yield()`가 발생하여 cpu가 다시 스케줄러로 넘어갔을 때, 전에 스케줄 되던 프로세스가 다시 선택될 가능성이 있습니다.

큐의 구조를 바꾸는 과정을 타이머 인터럽트가 발생했을 때 하기 때문에, 만약 process A 실행 중에 `yield()` 시스템 콜로 cpu를 뺏으면 큐 내부 구조에 변화가 생기지 않아 process A가 여전히 여러 프로세스 중 가장 우선 순위가 높아 다음 턴에 다시 선택되어 스케줄 될 가능성이 있습니다.

ex)

L0: A B

L1: C

상태에서 A가 `yield()`를 발생 시킨다면

`A(yield())` → A

로 스케줄 됩니다.

- `setPriority(int pid, int priority)`: ptable을 돌면서 해당 pid인 프로세스를 찾아서 해당 프로세스 안의 priority 변수의 값을 변경해줍니다.
- `getLevel()`: 현재 프로세스의 q\_level 값을 리턴합니다.

## Implementation

필요없는 코드들의 삭제로 인해 사진의 줄번호와 실제 코드의 줄번호가 다를 수 있습니다.

`mlfq_scheduler.c`와 `mlfq_scheduler.h` 파일을 만들어, mlfq scheduler의 대부분의 로직과 시스템 콜들은 `mlfq_scheduler.c`에 구현해 놓았습니다.

그 외에 `main.c`, `trap.c`, `proc.c`의 일부와 시스템 콜을 만들기 위해 `sysproc.c`, `syscall.h`, `usys.S` 등등을 수정했습니다.

- `main.c`

`scheduler` 함수를 다음과 같이 변경했습니다. `mlfq_scheduler()` 함수는 `mlfq_scheduler.c`에 정의되어 있습니다.

```
// [Project 1] 기존 스케줄러는 주석처리하고,
// 스케줄러 함수 안에서 제가 구현한 mlfq_scheduler()를 호출하도록 변경했습니다.
void
scheduler(void) {
|   mlfq_scheduler();
}
```

- `proc.c`

- struct ptable (L0\_size, locked\_proc 추가)

```

12  struct ptable {
13      struct spinlock lock;
14      struct proc proc[NPROC];
15
16      int L0_size;
17      int L1_size;
18      int L2_size;
19
20      int locked_proc;
21  } ptable;

```

- allocproc():

```

100  found:
101      p->state = EMBRYO;
102      // p->proc_ticks = 0;
103      p->pid = nextpid++; // pid가 높을 수록 나중에 만들어진 process이다.
104      p->priority = 3; // 프로세스가 처음 만들어졌을 땐 3으로 초기화.
105      p->q_num = ptable.L0_size++;

```

line 104~105를 추가했습니다.

p->q\_num = (L0 큐의 사이즈)++;

p->priority = 3;

- **q\_num 변수와 Queue 내부의 schedule 순서**

**q\_num은 큐 내부에서의 FCFS에 따른 우선 순위를 의미합니다.**

q\_num이 작을 수록 큐 내부에서의 우선 순위가 높습니다. Queue에 새로운 프로세스가 들어오면 그 프로세스의 q\_num은 Ln queue의 기존 사이즈가 되며, queue의 사이즈가 증가합니다.

ex) 비어있는 L0 큐(L0\_size = 0)에 A, B 프로세스가 있다고 하면, 먼저 들어온 A의 q\_num = L0\_size++; 되어 A의 q\_num = 0, L0\_size = 1이 됩니다. 다음에 들어온 B는 q\_num = L0\_size++; 되어 B의 q\_num = 1, L0\_size = 2가 됩니다. A가 CPU를 잡고 1 tick이 지나면 A.q\_num = L0\_size-1;이 되어 큐 내에서 가장 나중 순서로 변경되고, B.q\_num--; 되어 1 감소합니다.

ex) L0 queue에 A, B, C, D 프로세스가 있다면 다음과 같이 스케줄 됩니다.

A,B,C,D,A,B,C,D,A,B,C,D...

하지만 L2 큐에서는 항상

- fork():

```

237     np->state = RUNNABLE;
238     np->priority = 3;
239     np->proc_ticks = 0;
240     np->q_level = 0;
241     np->q_num = ptable.L0_size-1;

```

fork()에 해당 부분을 추가했습니다 (238~241)

priority, proc\_ticks, q\_level, q\_num을 초기화해주는 부분입니다.

- proc.h

```

41  // Per-process state
42  struct proc {
43      uint sz;                // Size of process memory (bytes)
44      pde_t* pgdir;          // Page table
45      char *kstack;           // Bottom of kernel stack for this process
46      enum procstate state;    // Process state
47      int pid;                // Process ID
48      struct proc *parent;     // Parent process
49      struct trapframe *tf;    // Trap frame for current syscall
50      struct context *context; // swtch() here to run process
51      void *chan;              // If non-zero, sleeping on chan
52      int killed;              // If non-zero, have been killed
53      struct file *ofile[NOFILE]; // Open files
54      struct inode *cwd;        // Current directory
55      char name[16];           // Process name (debugging)
56
57      // project #1을 위해서 추가한 것
58      int q_level;              // 현재 위치한 큐의 레벨
59      int priority;             // 0, 1, 2, 3의 값을 가짐. 처음엔 3으로 초기화
60      int proc_ticks;           // 이 프로세스의 Tick. yield, priority boosting
61      int q_num;               // FCFS 구현을 위한 큐 안에서의 순서입니다.
62  };

```

struct proc에 q\_level, priority, proc\_ticks, q\_num 변수를 추가했습니다.

각각의 변수가 어떤 역할을 하는지는 주석을 참고해 주세요.

- mlfq\_scheduler.c
  - mlfq\_scheduler():

```

23 void
24 mlfq_scheduler(void) {
25     #if DEBUG_MODE
26         cprintf("[INFO] mlfq scheduler started.\n");
27     #endif
28     struct cpu *c = mycpu();
29     c->proc = 0;
30
31     struct proc* p = 0;
32     struct proc* cur = 0;
33
34     for(;;) {
35
36         p = 0; cur = 0;
37
38         // 인터럽트 허용
39         sti();
40
41         // custom ptable에 lock 걸기. 인터럽트를 무시
42         acquire(&ptable.lock);
43
44         if(ptable.locked_proc != -1) { // lock 중인 proc이 있다면 개를 먼저 처리
45             for(cur = ptable.proc; cur < &ptable.proc[NPROC]; cur++) {
46                 if(cur->pid == ptable.locked_proc) { // lock중인 프로세스를 찾음
47                     if(cur->state == ZOMBIE) {
48                         // lock을 풀어줍니다.
49                         ptable.locked_proc = -1;
50
51                         int z_q1 = cur->q_level;
52                         int z_qn = cur->q_num;
53                         struct proc* cp = 0;
54                         for(cp = ptable.proc; cp < &ptable.proc[NPROC]; cp++) {
55                             if(cp->q_level == z_q1 && cp->q_num > z_qn) {
56                                 cp->q_num--;
57                             }
58                         }

```

```

59         break;
60     }
61     }
62     p = cur; break;
63 }
64 }
65 } else { // lock 중인 proc이 없다면 -> mlfq로 동작
66     // ptable을 완전 탐색
67     for(cur = ptable.proc; cur < &ptable.proc[NPROC]; cur++) {
68         if(cur->state != RUNNABLE)
69             continue;
70
71         int ql = cur->q_level;
72
73         if(ql == 0) { // cur가 L0에 속한 프로세스라면
74             if(p == 0) {
75                 p = cur;
76             } else if(p->q_level > 0 || p->q_num > cur->q_num) {
77                 // 기존 p가 L1, L2 큐의 것이면 무조건 선택
78                 // p와 cur의 q_num 작은 것을 선택(Fcfs이므로 큐의 앞에 있는 것을 선택)
79                 p = cur;
80             }
81         } else if(ql == 1) { // cur가 L1에 속한 프로세스라면
82             if(p != 0 && p->q_level == 0) continue; // L0에 속한 프로세스는 검사하지 않음
83
84             if(p == 0) {
85                 p = cur;
86             } else if(p->q_level > 1) { // L2에 속한 프로세스라면 L1에 속한 것이 우선이므로 cur
87                 p = cur;
88             } else if(p->q_level == 1 && p->q_num > cur->q_num) { // p와 cur가 둘다 L1에 속한
89                 p = cur;
90             }
91         }
92     }
93 } else if(ql == 2) { // cur가 L2에 속한 프로세스라면
94
95     // min_priority: 현재까지 나온 L2 프로세스의 priority 중 가장 작은 값
96
97     if(p != 0 && p->q_level < 2) continue; // L0, L1에 속한 프로세스는 검사하지 않음
98
99     if(p == 0) {
100         p = cur;
101     } else if(p->priority > cur->priority) { // priority 값이 작은(우선순위가 높은)것을 선택
102         p = cur;
103     } else if(cur->priority == p->priority && p->q_num > cur->q_num) {
104         p = cur; // cur와 p의 priority가 같다면 q_num이 작은 것을 선택
105     } else if(cur->priority == p->priority && p->q_num > cur->q_num && cur->pid < p->pid) {
106         p = cur; // cur와 p의 q_num도 같다면 pid 작은 것을 선택
107     }
108
109 } else { // 나올 수 없는 케이스, 코드가 제대로 동작하는지를 확인하기 위해 넣은 부분입니다.
110     #if DEBUG_MODE
111     cprintf("[DEBUG MODE]: Invalid ql\n");
112     #endif
113 }
114 }
115 }
116 }

```

struct proc에 q\_level, priority 등의 변수를 추가하고, mlfq\_scheduler() 안에서는 `for(;;)` loop를 한 번 돌 때마다 locked된 프로세스가 있는지 확인하고, 있다면 해당 프로세스를 스케줄, 없다면 전체 ptable을 완전 탐색하며 q\_level과 priority를 비교하여 cpu 제어를 넘겨줄 적절한 프로세스 p를 찾을 것입니다.

- Lock이 걸려있는 경우 (line 44 ~ line 64)



ptable.locked\_proc ≠ -1이라면, lock이 걸려있다는 뜻입니다. 따라서 그 프로세스를 찾아서 먼저 스케줄 해야 합니다.

ptable을 완전 탐색하며 해당 pid와 일치하는 프로세스를 찾습니다.

- locked 프로세스가 ZOMBIE 상태인 경우:  
스케줄 하지 않고 lock을 풀어준 뒤 큐에서 제거해주어야 합니다. 따라서 locked\_proc = -1로 바꿔주고, 해당 프로세스와 같은 큐에 있는 프로세스 중, 현재 프로세스보다 q\_num이 높은 프로세스의 q\_num을 하나씩 감소시킵니다. (Zombie process를 큐에서 제거하고 한 칸씩 당겨주는 과정)
- RUNNABLE 상태인 경우:  
해당 프로세스에게 cpu를 넘겨줍니다.

- Lock이 걸려있지 않은 경우 (line 65~ line 116)

초기에 struct proc\* p=0으로 설정하고, struct proc\* cur를 선언하여

`for(cur = ptable.proc; cur < *ptable.proc[NPROC]; cur++;)` 로 완전 탐색을 합니다. 그리고 p보다 cur이 이번에 cpu를 차지할 프로세스로 더 적절하다고 판단되면 `p = cur` (p에 cur을 대입)해줍니다.

다음은 p와 cur을 비교하여 적절한 p를 찾는 flow를 정리한 것입니다.

(ql : 프로세스가 속한 큐의 레벨)

1. ql == 0 일 때
  - a. p==0이면 바꾼다.
  - b. p ≠ 0이면서 p→q\_num > cur→q\_num 이면 바꾼다.
2. ql == 1일 때
  - a. p ≠ 0이면서 p→ql == 0 이면 continue
  - b. p==0이면 바꾼다.
  - c. p ≠ 0 이면서 p→ql > 1이면 바꾼다.
  - d. p ≠ 이면서 p→ql == 1이면서 p→q\_num > cur→q\_num이라면 바꾼다.
3. ql == 2일 때,
  - a. p≠0 이면서 p→ql < 2 라면 continue
  - b. p == 0 이면 바꾼다.
  - c. p ≠ 0 이면서 cur→priority < p→priority 이면 바꾼다.
  - d. p ≠ 0 이면서 cur→priority == p→priority 면서 p→q\_num > cur→q\_num이면 바꾼다.
  - e. p ≠ 0 이면서 cur→priority == p→priority 면서 p→q\_num == cur→q\_num이라면 p와 cur 중 pid가 작은 것을 선택합니다.

위의 과정을 거쳐 최종적으로 선택된 프로세스(p)를 스케줄 합니다.

- getLevel (system call):

```

149 // 현재 cpu가 처리하고 있는 프로세스의 q_level을 반환합니다.
150 int
151 getLevel(void) {
152     int ret;
153     acquire(&ptable.lock);
154     struct proc *p = myproc();
155
156     ret = p->q_level;
157     // cprintf("[DEBUG!!] q level %d\n", p->q_level);
158
159     release(&ptable.lock);
160
161     return ret;
162 }

```

myproc()의 q\_level 값을 반환합니다.

- setPriority (system call):

```

201 // ptable에서 argument로 들어온 pid와 동일한 프로세스가 있다면 그 프로세스의 priority를 변경합니다.
202 void
203 setPriority(int pid, int priority) {
204     // cprintf("[debug] %d %d", pid, priority);
205
206     // exception: priority 값이 0~3이 아닐 경우 -> 콘솔에 에러문을 띄우고 함수를 종료합니다.
207     if(priority < 0 || priority > 3) {
208         cprintf("[Error] setPriority failed (invaild priority %d)\n", priority);
209         return; // 프로세스는 종료하지 않고 return을 하며 나옵니다.
210     }
211     struct proc* cur = 0;
212     int is_found = 0; // pid가 일치하는 프로세스를 찾았는지 검사하는 변수입니다.
213
214     acquire(&ptable.lock);
215     for(cur = ptable.proc; cur < &ptable.proc[NPROC]; cur++) {
216
217         if(cur->state == ZOMBIE || cur->state == UNUSED) continue;
218
219         if(cur->pid == pid) {
220             cur->priority = priority;
221             is_found = 1;
222             break;
223         }
224     }
225     release(&ptable.lock);
226
227     // exception: 해당 pid를 가진 프로세스가 존재하지 않을 경우 -> 콘솔에 에러문을 띄우고 함수를 종료합니다.
228     if(is_found == 0) { //
229         cprintf("[ERROR] setPriority failed (there's no matching pid)\n");
230     }
231
232 }

```

- error handling: 만약 입력된 priority가 0~3의 값이 아니라면 에러문을 띄우고 return합니다.
- ptable을 전체 탐색하며 입력으로 들어온 pid와 일치하는 프로세스를 찾고, 찾았다면 priority를 변경시킵니다.

◦ 찾지 못했다면 에러 문을 띄우고 함수를 빠져나옵니다.

- priority\_boosting()

```
237 void
238 priority_boosting(void) {
239
240     struct proc* cur = 0;
241     // int locked = -1;
242     int i=0;
243     ptable.L0_size = 0;
244     ptable.L1_size = 0;
245     ptable.L2_size = 0;
246
247     if(ptable.locked_proc != -1) { //뭔가 우선적으로 처리되고 있었다면 lock을 풀어줍니다.
248         _schedulerUnlock(2020005269); // locked 된게 있었다면 이 안에서 L0 queue의 맨앞에 locked 됐
249     } else { //lock된 것이 없다면 L0에 있는 프로세스를 먼저 L0에 담습니다.
250         // L0 프로세스를 L0으로 이동시키고, priority = 3으로 설정하고, time quantum을 초기화합니다.
251         for(cur = ptable.proc; cur < &ptable.proc[NPROC]; cur++) {
252             if(cur->state == UNUSED || cur->state == ZOMBIE) continue;
253             if(cur->q_level != 0) continue;
254
255             cur->q_level = 0; // L0로 이동
256             cur->priority = 3; // priority = 3으로 초기화
257             cur->proc_ticks = 0; // time quantum을 초기화
258             cur->q_num = i++;
259             ptable.L0_size++;
260         }
261     }
262 }
263
264 // L1 프로세스를 L0으로 이동시키고, priority = 3으로 설정하고, time quantum을 초기화합니다.
265 for(cur = ptable.proc; cur < &ptable.proc[NPROC]; cur++) {
266     if(cur->state == UNUSED || cur->state == ZOMBIE) continue;
267     if(cur->q_level != 1) continue;
268
269     cur->q_level = 0; // L0로 이동
270     cur->priority = 3; // priority = 3으로 초기화
271     cur->proc_ticks = 0; // time quantum을 초기화
272     cur->q_num = i++;
273     ptable.L0_size++;
```

```

273         ptable.L0_size++;
274     }
275 }
276
277 // L2 프로세스를 L0으로 이동시키고, priority = 3으로 설정하고, time quantum을 초기화함
278 for(cur = ptable.proc; cur < &ptable.proc[NPROC]; cur++) {
279     if(cur->state == UNUSED || cur->state == ZOMBIE) continue;
280     if(cur->q_level != 2) continue;
281
282     cur->q_level = 0; // L0로 이동
283     cur->priority = 3; // priority = 3으로 초기화
284     cur->proc_ticks = 0; // time quantum을 초기화
285     cur->q_num = i++;
286     ptable.L0_size++;
287 }
288
289
290 acquire(&tickslock);
291 ticks = 0; // global ticks를 초기화.
292 release(&tickslock);
293 }

```

- locked 된 프로세스가 있다면 lock을 풀어줍니다.
  - ptable을 3번 전체 탐색하며 L0, L1, L2 순으로 프로세스의 ticks, q\_num, q\_level 등을 적절히 초기화 해준 뒤 L0 큐로 넣어줍니다.
  - Design에서 설명 드렸다 싶이, 이 과정에서 L0 큐에 들어간 결과가, 부스팅이 일어나기 전 Ln 큐 내부의 순서를 보장하지 않을 수 있습니다.
- schedulerLock()

```

296 // 인터럽트로 인한 호출을 위한 ptable.lock을 걸지 않는 버전입니다.
297 void _schedulerLock(int password) {
298
299     if(password != PASSWORD) { // 안전성을 위해 한번 더 검사
300         cprintf("[ERROR]: SchedulerLock failed. (Password is incorrect)\n");
301         return;
302     }
303
304     if(ptable.locked_proc != -1) { // exception: lock을 걸었는데 이미 걸려있다면 그냥 리턴합니다.
305         cprintf("[INFO]: Already locked!\n");
306         return;
307     }
308
309     if(myproc()->state == ZOMBIE) {
310         cprintf("[INFO]: zombie lock.\n");
311         return;
312     }
313     ptable.locked_proc = mycpu()->proc->pid;
314 }
315

```

```

365 void schedulerLock(int password) {
366
367     acquire(&ptable.lock);
368     _schedulerLock(password);
369     release(&ptable.lock);
370 }

```

- 현재 실행 중인 프로세스의 pid를 ptable.locked\_proc에 대입합니다.

다음 스케줄부터는 계속 해당 프로세스가 cpu를 잡게 됩니다.

이 과정에서도 yield()는 1tick마다 일어나지만 해당 프로세스만 스케줄되도록 구현했습니다.

ex) A,A,A,A,A(100ticks), (mlfq로 돌아감)

- Error handling:

- 만약 locked\_table ≠ -1이 아닌데, ptable에 해당 pid를 가진 프로세스가 없다면 에러문을 띄우로 return;하여 나갑니다. (즉, 프린트 문이 뜰 뿐 아무 일도 일어나지 않고 남은 프로세스가 계속 스케줄 됩니다. 코딩이 제대로 되었다면 일어나지 않을 상황입니다.)
- 만약 lock이 이미 걸려있는데 다시 lock을 건다면, 이미 lock되어 있다는 프린트 문을 띄우고 return; 합니다. (즉 아무 일도 일어나지 않고, 남은 프로세스들이 계속 실행됩니다. 이 경우는 test code에 따라 일어날 수 있는 상황입니다.)

- schedulerUnlock():

```

317 void _schedulerUnlock(int password) {
318
319     if(password != PASSWORD) { // 안전성을 위해 한번 더 검사
320         cprintf("[ERROR]: SchedulerUnlock failed. (Password is incorrect)\n");
321         return;
322     }
323
324
325     if(ptable.locked_proc == -1) { // exception: unlock을 걸었는데 lock이 걸려있지 않
326         cprintf("[INFO]: Already Unlocked!\n");
327         return;
328     }
329
330     int pid = ptable.locked_proc;
331     ptable.locked_proc = -1; // lock이 걸려있지 않은 상태로 만들어줍니다.
332
333     // int q_num = -1;
334
335     ptable.L0_size = 0;
336
337     // L0 queue의 맨 앞으로 돌려보내야함
338     struct proc* cur = 0;
339     for(cur = ptable.proc; cur < &ptable.proc[NPROC]; cur++) {
340         if(cur->pid == pid) {
341             ptable.L0_size++;
342             cur->q_level = 0;
343             cur->priority = 3;
344             cur->proc_ticks = 0;
345             cur->q_num = 0;
346
347             break;
348         }
349     }
350 }
351
352     for(cur = ptable.proc; cur < &ptable.proc[NPROC]; cur++) {

```

```

349
350     }
351
352     for(cur = ptable.proc; cur < &ptable.proc[NPROC]; cur++) {
353         if(cur->pid == pid) continue;
354         if(cur->q_level == 0) {
355             ptable.L0_size++;
356             cur->q_num++;
357         }
358     }
359
360     return;
361
362 }

```

```

373 void schedulerUnlock(int password) {
374
375     acquire(&ptable.lock);
376     _schedulerUnlock(password);
377     release(&ptable.lock);
378
379     return;
380
381 }

```

- Error handling: 암호가 틀렸다면 프린트 문을 띄우고 함수를 빠져나갑니다.
  - ptable.locked\_proc = -1로 만들어주고, ptable을 완전 탐색하여 기존에 lock이 걸려있던 프로세스를 찾아낸 후에 그 프로세스의 상태를 적절히 초기화 해주고 L0 큐의 맨 앞으로 보냅니다. (line 338 ~ 360)
  - line 352 ~ line 358은 L0 큐에 있던 프로세스들을 한 칸씩 뒤로 당겨주는 과정입니다.
- update\_proc():

```

348 void
349 update_proc(void) {
350
351
352     int ql = myproc()->q_level;
353
354
355     struct proc* cur = 0;
356
357     // 1 tick 마다 yield는 발생해야하므로 FCFS에 의한 우선 순위를 조정해준다.
358     for(cur = ptable.proc; cur < &ptable.proc[NPROC]; cur++) {
359         if(cur->pid == myproc()->pid) continue;
360         if(cur->q_level == ql && cur->q_num > 0) {
361             cur->q_num--;
362         }
363     }
364
365
366     // 현재 proc의 우선 순위는 맨 나중으로 조정.
367     if(ql == 0) {
368         myproc()->q_num = ptable.L0_size-1;
369     } else if(ql == 1) {
370         myproc()->q_num = ptable.L1_size-1;
371     } else {
372         myproc()->q_num = ptable.L2_size-1;
373     }
374
375
376     /******타임퀀텀 체크***** */
377     if(ql == 0 && myproc()->proc_ticks == L0_TIME_QUANTUM) { // 타임 퀀텀 다 되면
378
379         #if DEBUG_MODE
380             cprintf("[DEBUG MODE] q level down 0 -> 1\n");
381         #endif
382         // L1 큐로 내려주기
383         myproc()->q_level++;
384         ptable.L0_size--;
385         ptable.L1_size++;

```

```

386 // FCFS에 따른 우선순위를 바꿔주기
387 myproc()->q_num = ptable.L1_size-1;
388
389 // local tick 초기화
390 myproc()->proc_ticks = 0;
391
392
393 } else if(q1 == 1 && myproc()->proc_ticks == L1_TIME_QUANTUM) { // L1에 속하고, 타임 쿼텀 다 되면
394
395 #if DEBUG_MODE
396 cprintf("[DEBUG MODE] q level down 1 -> 2\n");
397 #endif
398
399 // L2 큐로 내려주기
400 myproc()->q_level++;
401 ptable.L1_size--;
402 ptable.L2_size++;
403
404
405 // FCFS에 따른 우선순위를 바꿔주기
406 myproc()->q_num = ptable.L2_size-1;
407
408 // local tick 초기화
409 myproc()->proc_ticks = 0;
410
411 } else if(q1 == 2 && myproc()->proc_ticks == L2_TIME_QUANTUM) { // L2에 속하고, 타임 쿼텀 다 되면
412 // local tick 초기화
413 myproc()->proc_ticks = 0;
414
415 if(myproc()->priority >= 1) myproc()->priority--;
416
417 // int q_n = myproc()->q_num;
418
419 myproc()->q_num = ptable.L2_size-1;
420
421
422

```

- line 357~line 362: 같은 큐에 있는 프로세스들의 q\_num(FCFS에 의한 우선 순위)를 하나씩 내려줍니다.
- line 366 ~ line 374: 방금 처리한 프로세스의 q\_num을 Ln\_size-1로 업데이트 해줍니다.
- line ~ line 451: 방금 처리한 프로세스가 타임 쿼텀이 다 되었다면 타임 쿼텀을 초기화해주고, L0,L1에 있었다면 하위 큐로 내려주고, L2에 있었다면 priority를 3으로 바꿔줍니다.

#### • trap.c

```

38 // [Project 1] user mode에서도 인터럽트가 호출될 수 있도록 함
39 SETGATE(idt[T_SCHE_LOCK], 1, SEG_KCODE<<3, vectors[T_SCHE_LOCK], DPL_USER);
40 SETGATE(idt[T_SCHE_UNLOCK], 1, SEG_KCODE<<3, vectors[T_SCHE_UNLOCK], DPL_USER);
41 SETGATE(idt[T_BOOST], 1, SEG_KCODE<<3, vectors[T_BOOST], DPL_USER);

```

129, 130 interrupt가 usermode에서도 호출될 수 있도록 변경해주었습니다.

```

75 if(myproc() && myproc()->state == RUNNING &&
76 tf->trapno == T_IRQ0+IRQ_TIMER) {
77 // 현재 프로세스 proc tick 증가시키기
78 myproc()->proc_ticks++;
79 update_proc();

```



타이머 인터럽트가 발생했을 때, 프로세스의 로컬 틱을 증가시키고, update\_proc() 함수에서 queue 내부의 상태를 변경 시켜줍니다.

```
149 // [Project 1] priority boosting (131 interrupt)
150 if(ticks == 100 && tf->trapno == T_IRQ0+IRQ_TIMER) {
151     priority_boosting();
152 }
```

global tick이 100이 되면, mlfq\_scheduler.c에 구현돼있는 priority\_boosting() 함수가 호출되도록 변경했습니다.

- others

이 외에도 system calls와 interrupt를 구현하기 위해 usys.S, sysproc.c, syscall.h, defs.h 등등을 수정하였으나, 해당 부분은 올려주신 실습 영상과 동일하게 진행했기 때문에 생략하겠습니다.

## Result

piazza에 올려주신 테스트 코드를 약간 수정하여 (프로세스의 개수를 5개로 수정) 실행했을 때의 결과입니다. 다른 시스템 콜들을 호출하지 않고 mlfq scheduler에 따라 스케줄 한 것이기에 pid가 4, 5, 6, 7, 8 순서로 프린트 되었습니다. (이는 프로세스의 개수가 적기 때문에 가능한 결과이며 프로세스의 개수가 이것보다 늘어나면 출력 순서는 달라질 수 있습니다.)

```
SeaBIOS (version 1.15.0-1)
```

```
iPXE (https://ipxe.org) 00:03.0 CA00 PCI2.10 PnP PMM+1FF8B4A0+1FECB4A0 CA00
```

```
Booting from Hard Disk..xv6...
```

```
cpu0: starting 0
```

```
sb: size 1000 nblocks 941 ninodes 200 nlog 30 logstart 2 inodestart 32 bmap start 58
```

```
init: starting sh
```

```
$ mlfq_test
```

```
MLFQ test start
```

```
[Test 1] default
```

```
Process 4
```

```
L0: 15781
```

```
L1: 20158
```

```
L2: 64061
```

```
Process 5
```

```
L0: 17330
```

```
L1: 24969
```

```
L2: 57701
```

```
Process 6
```

```
L0: 19439
```

```
L1: 27615
```

```
L2: 52946
```

```
Process 7
```

```
L0: 22366
```

```
L1: 33125
```

```
L2: 44509
```

```
Process 8
```

```
L0: 22435
```

```
L1: 33793
```

```
L2: 43772
```

```
[Test 1] finished
```

```
done
```

```
$
```

- schedulerLock test: 같은 테스트 코드에 다음 코드를 추가하여 실행한 결과입니다. (process 7이 lock을 계속 잡고 있기 때문에 7이 가장 먼저 출력됩니다.)

```
104         if(i % 10000 == 0) {
105             if(pid % 4 == 3) {
106                 __asm__("int $129");
107             }
108         }
```

```

SeaBIOS (version 1.15.0-1)

ipXE (https://ipxe.org) 00:03.0 CA00 PCI2.10 PnP PMM+1FF8B4A0+1FECB4A0 CA00

Booting from Hard Disk..xv6...
cpu0: starting 0
sb: size 1000 nblocks 941 ninodes 200 nlog 30 logstart 2 inodestart 32 bmap start 58
init: starting sh
$ mlfq_test
MLFQ test start
[Test 1] default
[INFO]: Already locked!
[INFO]: Already locked!
[INFO]: Already locked!
[INFO]: Already locked!
[INFO]: Already locked!
[INFO]: Already locked!
[INFO]: Already locked!
Process 7
L0: 7361
L1: 12495
L2: 80144
Process 4
L0: 11648
L1: 15632
L2: 72720
Process 5
L0: 14569
L1: 18983
L2: 66448
Process 6
L0: 16867
L1: 22503
L2: 60630
Process 8
L0: 18724
L1: 24866
L2: 56410
[Test 1] finished
done
$ 

```

- schedulerUnlock():

line 109~ 114를 추가하였습니다. (lock을 걸고 unlock을 해줌)

lock을 걸고 조금 후에 unlock을 걸어주었기 때문에 4, 5, 7, 6, 8 순서로 출력 되었습니다.

```

103
104     if(i % 10000 == 0) {
105         if(pid % 4 == 3) {
106             __asm__("int $129");
107         }
108     }
109
110     if(i % 10002 == 0) {
111         if(pid % 4 == 3) {
112             __asm__("int $130");
113         }
114     }

```

SeaBIOS (version 1.15.0-1)

iPXE (https://ipxe.org) 00:03.0 CA00 PCI2.10 PnP PMM+1FF8B4A0+1FECB4A0 CA00

Booting from Hard Disk...

xv6...

cpu0: starting 0

sb: size 1000 nblocks 941 ninodes 200 nlog 30 logstart 2 inodestart 32 bmap start 58

init: starting sh

\$ mlfq\_test

MLFQ test start

[Test 1] default

Process 4

L0: 16231

L1: 21879

L2: 61890

Process 5

L0: 20913

L1: 29736

L2: 49351

Process 7

L0: 31791

L1: 36817

L2: 31392

Process 6

L0: 22191

L1: 30435

L2: 47374

Process 8

L0: 23603

L1: 33843

L2: 42554

[Test 1] finished

done

\$ █

## Trouble shooting

- xv6의 부팅이 중간에 멈추던 문제

예전에는 실제로 struct proc\*을 담고 있는 circular queue와 priority queue를 만들어서 L0, L1, L2을 구현하고, 이 큐들의 size가 0보다 크다면 해당 큐에 접근하여 스케줄 하도록 해줬습니다.

그러나 해당 디자인으로 구현을 하던 중, xv6의 부팅이 중간에 멈추고 shell이 나타나지 않는 문제가 발생했습니다.

나중에 문제의 원인을 알게 되었는데,

```
// ...

while(ptable.L0_queue.q_size > 0) { // L0 큐에 프로세스가 1개 이상이라면 -> L0부
    if(c == 0) {
        cprintf("[error]\n");
    }
    #if DEBUG_MODE
        cprintf("line 207\n");
    #endif
    c->q_level = 0;
    p = process_L0(c);
    c->proc = p;
    switchvm(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;
    ptable.L0_queue.start = (ptable.L0_queue.start + 1) % Q_MAX_LE
    ptable.L0_queue.q_size--;
}

// ... 생략
```

다음과 같이 구현하였는데, Ln 큐에 프로세스가 있다면 그 프로세스를 queue에서 pop하고 스케줄을 하고 yield()가 발생하면 다시 큐에 넣어주지 않은 채로 넘어갔습니다.

그렇기 때문에 init process가 다 처리되니 않은 채로 deque되었고, 그 뒤고 해당 프로세스를 스케줄할 수 없어서 shell이 나타나지 않았던 것이었습니다.

해결:

그 당시에는 xv6의 구조를 충분히 파악하지 못했기 때문에 그냥 실제 큐를 구현하지 않고 큐 없이 논리적 multi level queue를 써서 구현하는 방식으로 디자인을 바꾸어 해결하였습니다.

지금 와서 생각해보면 스케줄 할 프로세스를 pop(deque)한 후, cpu를 사용하다가 타이머 인터럽트가 와서 yield()가 실행되면 yield()함수 안에서(또는 yield()함수가 실행되기 직전에) 다시 적절한 큐로 enqueue해주고, 만약 프로세스가 ZOMBIE 상태라면 완전히 deque해주는 방식으로 해주면 될 것 같다는 생각이 듭니다.

- yield()에 대한 디자인을 변경하고 나서, process local tick이 0으로 초기화 되던 문제

원래는  $2n+4$  tick씩 프로세스가 연속으로 cpu를 사용하고 타임퀀텀을 다 소진하면 다른 프로세스로 cpu를 넘겨주는 식으로 구현을 했었는데, 나중에 piazza에 달아주신 답변을 보고 1tick마다 yield()가 발생하도록 바꾸었습니다.

그런데 그렇게 디자인을 바꾼 뒤 process의 proc\_tick이 yield()가 발생할 때마다 0으로 초기화되는 문제가 발생했습니다.

나중에 찾은 원인은 전에  $2n+4$  ticks를 연속으로 사용할 때에는 타임 퀀텀이 다 소진되어야만 context switch가 발생했기 때문에, mlfq\_scheduler() 내부에서 이번에 스케줄 해야하는 프로세스의 proc\_tick을 0으로 초기화하는 코드를 넣었었습니다. 디자인을 바꾸면서 수정해야하는 부분을 다 수정했다고 생각했는데 다 수정하지 않았던 것이었습니다.

해당 코드를 지우니 문제가 해결되었습니다.