

Index

- Index
- Introduction
 - Purpose
 - System Spec
 - 기기 사양
 - xv6 정보
 - Implementation Schedule
- Design
 - 1. Understandings of Basic RR scheduler in xv6
 - 2. FCFS (First-come-First-served) Scheduler Implementation
 - 1. Specification
 - 2. Algorithm
 - 어떻게 FCFS 스케줄러를 구현할까?
 - Modifying the Iterator, `p`
 - System call을 통해 특정 프로세스가 `yield` 한 경우, FCFS 스케줄러는 어떻게 동작해야 할까?
 - 3. MLFQ (Multiple level feedback queue) & Priority Scheduling
 - 1. Specification
 - 2. Algorithm
 - MLFQ 스케줄러를 어떻게 구현해야 할까?
 - Multiple iteration - lock problem
 - Scheduler가 Context switching 할 process를 선택한 경우
 - Scheduler가 Context switching을 할 process를 선택하지 못한 경우
 - 4. Mode Switch between FCFS & MLFQ Mode
 - 1. Specification
 - 2. Algorithm
 - FCFS Scheduler와 MLFQ Scheduler의 통합
- Implementation
 - 1. FCFS Scheduler
 - 전역변수 및 함수 설명
 - `kernel/proc.c`
 - `scheduler()`
 - Related System call
 - `kernel/mysyscall.c`
 - 2. MLFQ Scheduler
 - 전역변수 및 함수 설명
 - `kernel/proc.h`
 - `kernel/proc.c`
 - `kernel/trap.c`
 - `scheduler()`
 - Related System call
 - `kernel/mysyscall.c`
 - 3. Mode change
 - 전역변수 및 함수 설명
 - `kernel/proc.c`
 - `scheduler()`
 - Related System call
 - `kernel/mysyscall.c`
- Results
- Troubleshooting

Introduction

Purpose

Project01의 목적은 서로 다른 유형의 Scheduler를 구현하고, 이들 Scheduler 간의 전환을 가능하게 하는 것이다.

이를 위해 xv6의 프로세스 관리방법, Interrupt 처리 방법, 기존 Scheduler의 구현 사항을 숙지해야 하며, 이에 더하여 FCFS, MLFQ Scheduler의 작동 원리를 이해하고 이를 구현할 수 있어야 한다.

System Spec

기기 사양

1	프로세서	12th Gen Intel(R) Core(TM) i7-1260P	2.10 GHz
2	설치된 RAM	16.0GB(15.7GB 사용 가능)	
3	저장소	477 GB SSD SAMSUNG MZVL2512HCJQ-00BL7	
4	그래픽 카드	Intel(R) UHD Graphics (128 MB)	
5	시스템 종류	64비트 운영 체제, x64 기반 프로세서	

xv6 정보

1	RISC-V 버전 xv6
2	Git classroom에 업로드된 xv6 소프트웨어를 clone 하여 구현하였음.
3	
4	clone address : "https://github.com/splab-ELE3021/project01-2019092824.git"

Implementation Schedule

■ : In progress												
■ : Finished												
	4월	4월	4월	4월	4월	4월	4월	4월	4월	4월	4월	4월
	4일	5일	6일	7일	8일	9일	10일	11일	12일	13일	14일	15일
Task Name												
Context Switch 방법 이해												
Project 01 Repo Setting												
Practice 01 코드 작성 (Yield Syscall)												
Practice 01 코드 테스트												
Practice 02 코드 작성 (Debug Syscall)												
Practice 02 코드 테스트												
FCFS 구현 Blueprint 구상												
FCFS scheduler 구현												
FCFS scheduler <=> Original xv6 RR Switch												
FCFS scheduling test용 User program 작성												
MLFQ 구현 Blueprint 구상												
Priority Scheduling 구현 Blueprint 구상												
MLFQ scheduler 구현												
getlev, setpriority syscall 구상												
MLFQ scheduler <=> FCFS scheduler Switch												
MLFQ scheduling test 용 User program 작성												
Mode switch Syscall 구현 Blueprint 구상												
Mode switch Syscall 구현												
Mode Switch syscall test 용 User program 작성												
Mode Switch System call Refactoring												
Wiki 작성 - FCFS and related Syscall												
Wiki 작성 - MLFQ and related Syscall												
Wiki 작성 - Mode change and related Syscall												
Wiki 작성 - trouble Shooting												

Design

1. Understandings of Basic RR scheduler in xv6

기존 xv6의 scheduler는 다음과 같이 동작한다.

```

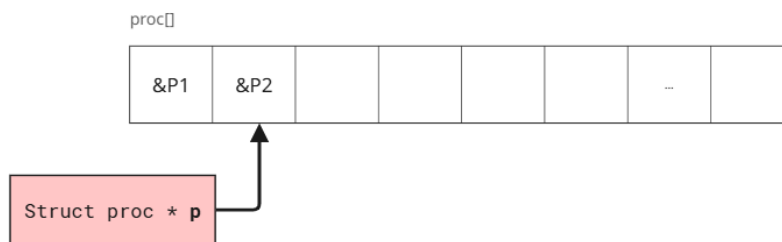
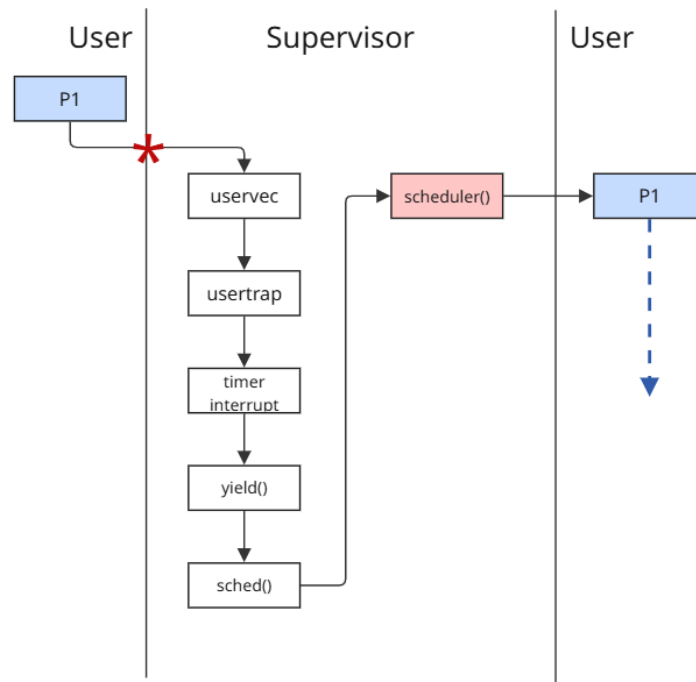
1  for(;;){
2      for(p = proc; p < &proc[NPROC]; p++) {
3          acquire(&p->lock);
4          if(p->state == RUNNABLE) {
5              p->state = RUNNING;
6              c->proc = p;
7              swtch(&c->context, &p->context);
8              // 다음 sched() 호출 시 여기부터 시작
9              c->proc = 0;
10             found = 1;
11         }
12         release(&p->lock);
13     }
14 }

```

proc.c 의 scheduler() 함수는 Timer interrupt로 인해 yield() 가 호출된 후, sched() 를 거쳐 실행된다. 이때 scheduler() 함수의 program counter는 swtch() 가 발생한 바로 다음을 가리킨다.

주목해야 할 것은 for loop 의 iteration 방법인데, proc Array를 iteration 하는 포인터 p 가 증가하는 시점, 즉 p++ 가 행해지는 시점을 명확히 할 필요가 있다. 아래 그림을 보자.

select 변수는 다음에 스케줄링 할 프로세스를 가리키는 포인터 (struct proc *)이다.



If it is RUNNABLE, Switch to that process.
 (if yield is called again)
 p++;

Code block

```

for(p = proc; p < &proc[NPROC]; p++) {
    acquire(&p->lock);
    if(p->state == RUNNABLE) {
        select = p;
        select -> state = RUNNING;
        c -> proc = select;
        selected = 1;
        swtch(&c->context, &select->context);
        // if yield is called again, context will be recovered from here.
    }

    c->proc = 0;
    found = 1;
    release(&p->lock);
    // p++ happens here
}
  
```

따라서, 자연스럽게 기존 xv6의 `scheduler` 는 process의 생성 순서대로 순차적으로 스케줄링되는 Round Robin의 형태를 갖게 된다.

2. FCFS (First-come-First-served) Scheduler Implementation

1. Specification

- 기존 Scheduler의 변형일 것.
- PCB에 어떠한 조작도 가하지 않을 것. (변수 추가 등)
- 생성 순서대로 프로세스를 선택할 것.
- 프로세스가 선택되면, `TERMINATE` 되거나 스스로 `yield` 하지 않는 한 계속 프로세스 실행 권한을 유지할 것.
- 커널 주요 부분을 수정할 것.

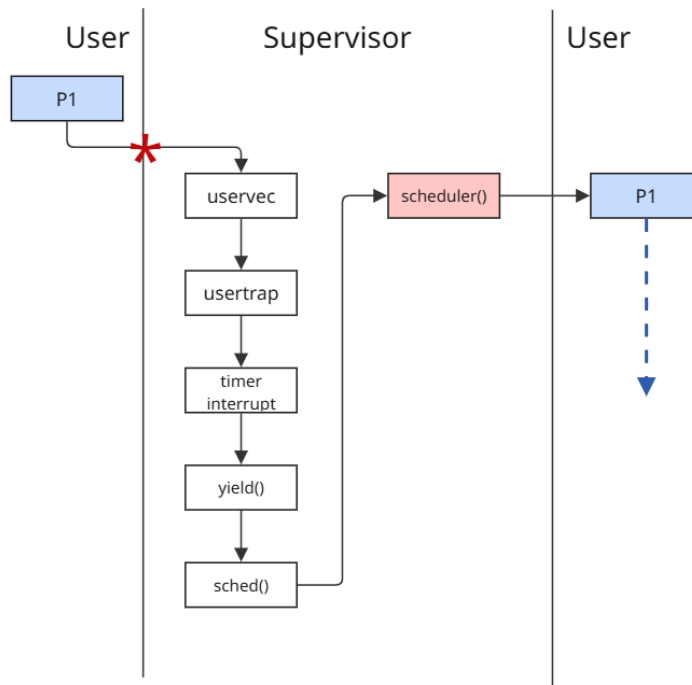
2. Algorithm

어떻게 FCFS 스케줄러를 구현할까?

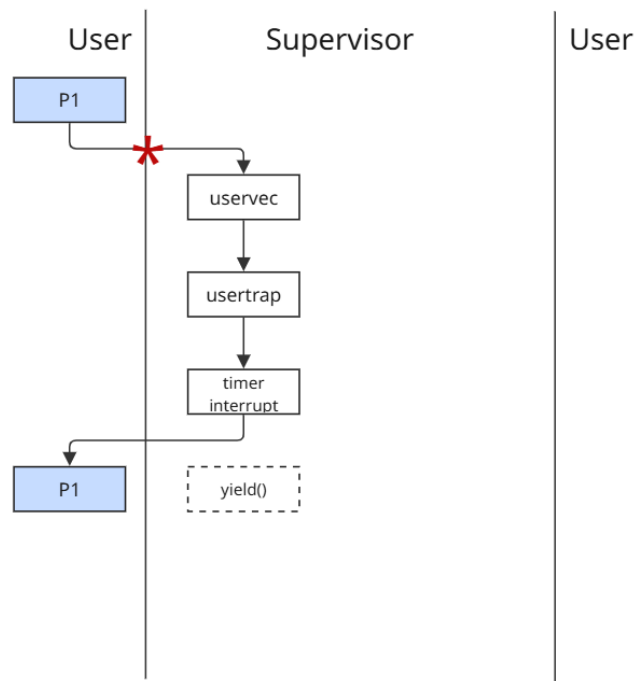
FCFS 스케줄링을 실현할 수 있는 방법은 크게 두가지가 있다.

1. `proc.c` 의 `scheduler` 내부에서 FCFS 처럼 동작하도록 하는 것.
2. 타이머 인터럽트를 꺼버리는 것.

FCFS Scheduler by modifying scheduler()



FCFS Scheduler by disabling timer interrupt



miro

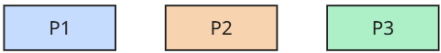
위 삽화는 각각 1번과 2번 방법을 도식화 한 것이다. 이 중 1번 방법을 선택하였으며, 근거는 아래와 같다.

- 타이머 인터럽트를 끄면 FCFS 스케줄링을 간단히 구현할 수 있다.
 - 매 Tick마다 `yield()`가 실행되지 않기 때문이다.
- 그러나 이 방법을 사용해 FCFS 스케줄링을 하게 되면 MLFQ 스케줄러와의 모드 전환이 어려워진다.
 - 예컨대 Process 4와 Process 5가 각각 있다고 하자.
 - P4는 P5의 부모 프로세스다. 즉 `proc` 배열 기준 P4의 index가 P5보다 먼저다.
 - MLFQ로 스케줄링 하는 상황, **P5 프로세스 실행 중에** 모드가 **FCFS로 바뀌면** 그 다음 scheduling round 부터는 정상적인 FCFS 스케줄러였을 때 **P4**로 Context switch가 발생해야 한다.
 - 그러나 타이머 인터럽트를 꺼버리면 그냥 P5가 끝날 때 까지 실행하게 되며, 이는 우리가 원하는 바가 아니다.

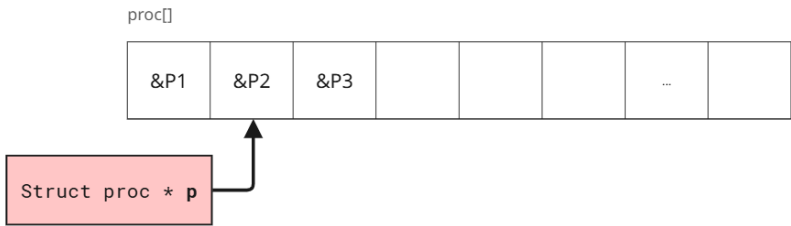
좋다. 1번 방법을 이용해 FCFS 스케줄러를 구현할 것이다. 어떻게 구현할 수 있을까?

앞서 xv6의 원래 scheduler에서의 **Iterator** `p`의 이동 조건에 조작을 가하면, 간단히 구현할 수 있다.

Modifying the Iterator, `p`

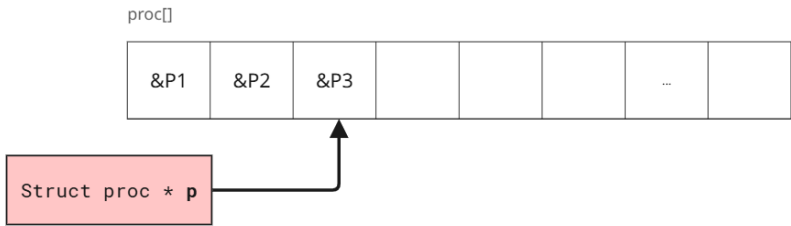


When P1 yield() by timer int



P2 is RUNNABLE, Switch to P2.
`p++;`

When P2 yield() by timer int,
Context **Restored by switch**, so P is at &P2, and then
`p++`, so its &P3 before enter the loop.



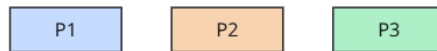
P3 is RUNNABLE, Switch to P3.

miro

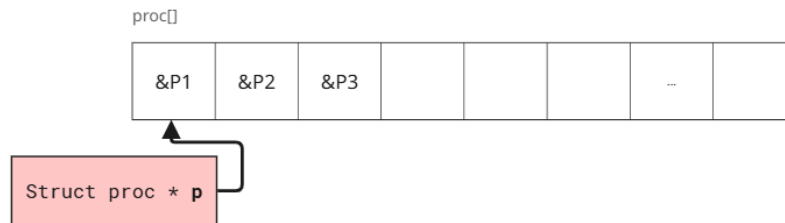
기존 스케줄러의 스케줄링 방법이다.

Iterator `p`가 `sched()` 직후 context restore 시점에 무조건 하나 늘어나기 때문에 (다음 process를 가리키기 때문에), 이 상황에서는 FCFS가 성립되지 않는다.

FCFS가 되려면 아래와 같이 되어야 한다.

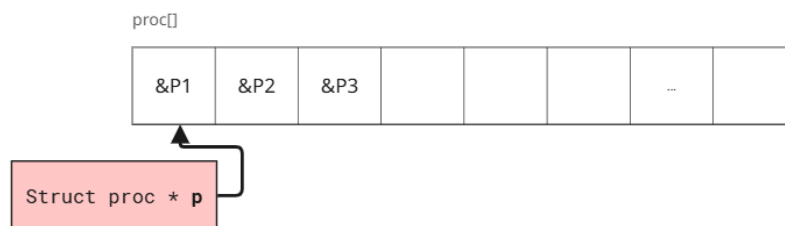


When P1 yield() by timer int



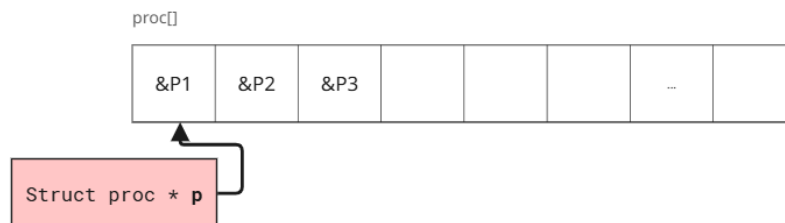
P1 is RUNNABLE, Switch to P1;
Reset p to proc;

When P1 yield() by timer int again.

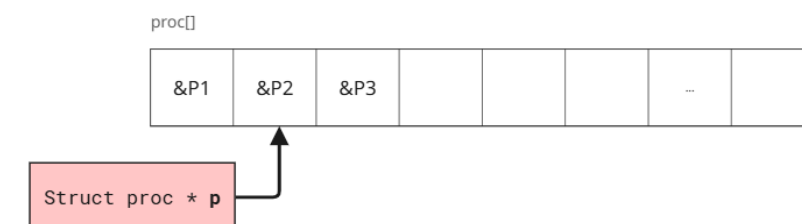


P1 is RUNNABLE, Switch to P1;
Reset p to proc;

When P1 is terminated by exit()

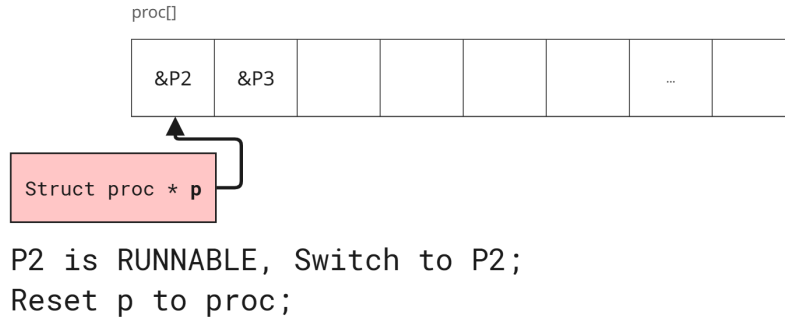


P1 is TERMINATED, Nothing to Switch;
p++;
NEXT LOOP



P2 is RUNNABLE, Switch to P2;
Reset p to proc;

When P2 yield() by timer interrupt



miro

위 Flow를 보자.

context switch 할 프로세스를 선택한 시점에, Iterator `p` 를 `proc` 으로, 즉 프로세스 테이블 배열의 첫번째 인덱스로 초기화하고 있다. 이러면 첫번째 인덱스의 프로세스가 `TERMINATED`, 혹은 `SLEEP` 하기 전 까지 `p` 는 계속 `RUNNABLE` 한 첫번째 프로세스를 가리키게 될 것이다. FCFS의 결과와 정확히 동일하다.

다만 이렇게 구현할 경우 문제가 생긴다. `yield()` 의 호출이 프로세스의 CPU 포기과 무관해진다. P1이 `yield()` 를 해서 CPU를 포기해봤자, 위 scheduling policy 하에선 어차피 P1이 다시 선택되기 때문이다.

System call을 통해 특정 프로세스가 `yield` 한 경우, FCFS 스케줄러는 어떻게 동작해야 할까?

본 문제를 해결하기 위해, 반드시 선행되어야 하는 개념이 있다.

Timer interrupt로 인해 호출되는 `yield()` 와, **System call**에 의해 호출되는 `yield()` 는 다르게 취급되어야 한다. 그 이유는 프로세스가 자신의 cpu 점유를 놓고자 하는 "의도의 유무"와 크게 관련이 있다.

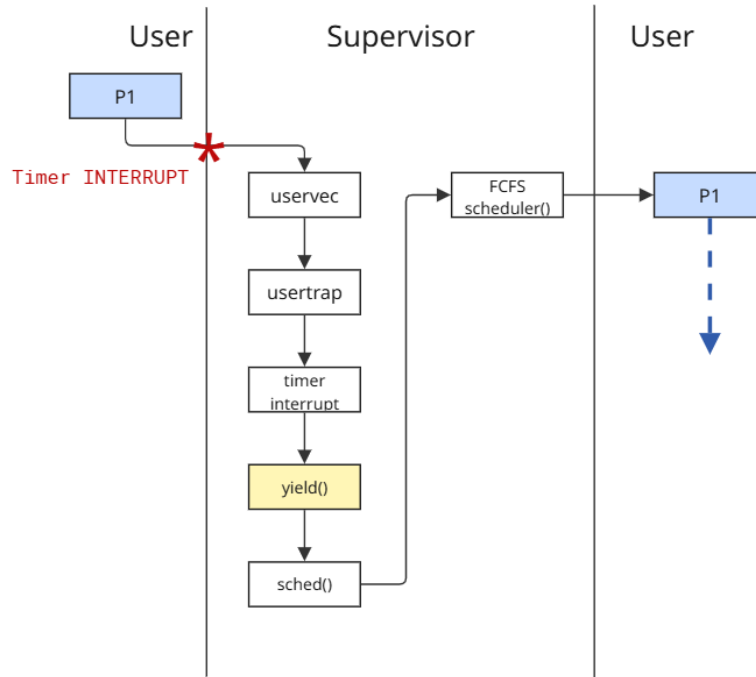
Timer interrupt로 인해 호출되는 `yield()` 는 preemptive 하다. 시간이 되면 반드시 호출되므로 프로세스의 의도와는 관계없이 CPU를 놔야 한다. 기존 코드의 수정으로 scheduler 를 FCFS로 바꿨으니, 그 프로세스가 `RUNNABLE` 해졌다가 다시 `RUNNING` 상태로 전환되는 것은 변함이 없다. 즉 Timer interrupt에 의한 `yield()` 는 별 신경 안써도 된다.

문제는 yield system call에 의한 `yield()` 이다. 이 경우 프로세스는 명확히 **CPU를 놓고자 하는 의도**가 있다. 따라서 이를 존중해야 한다.

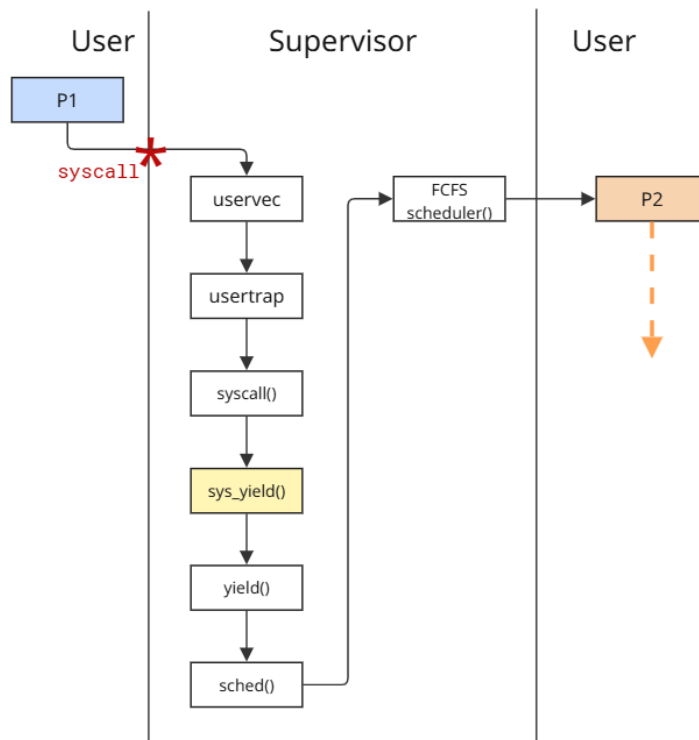
proc[]

&P1	&P2	&P3				...	
-----	-----	-----	--	--	--	-----	--

preemptive yield with timer interrupt - By FCFS, P1 should be scheduled.



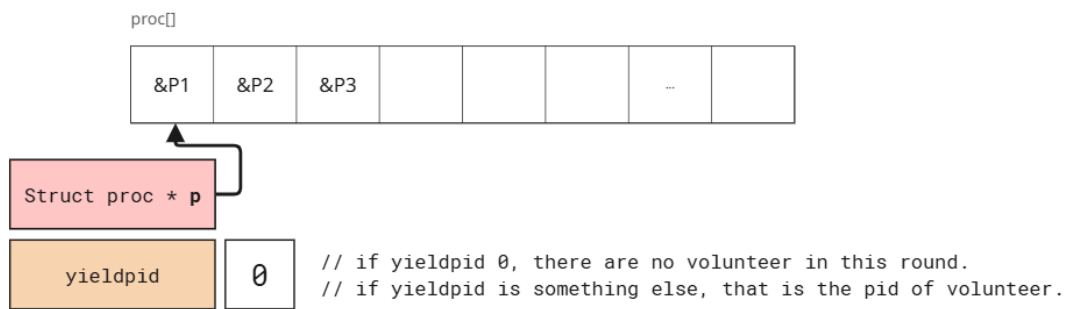
*Non-preemptive yield with **system call** - We have to **Respect** the intention of P1 to yield.*



System call에 의해 특정 프로세스가 `yield()` 를 호출할 경우, 그 프로세스의 `pid` 를 저장하는 변수를 하나 뒀으로써 이 문제를 해결할 수 있다.

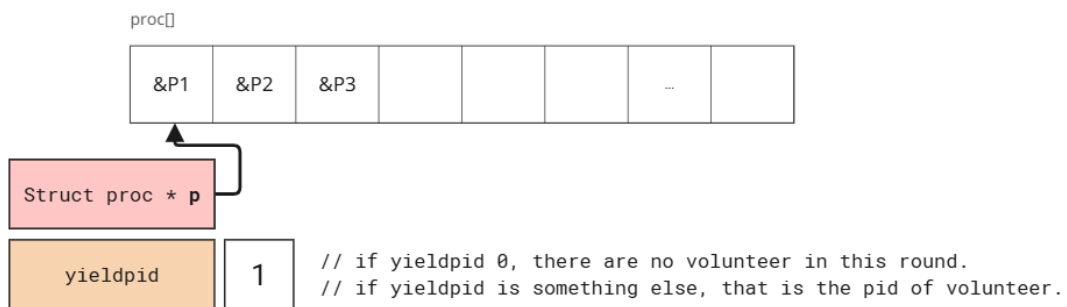
`yieldpid` 가 바로 그 변수이다.

While executing p1, Timer interrupt called yield()

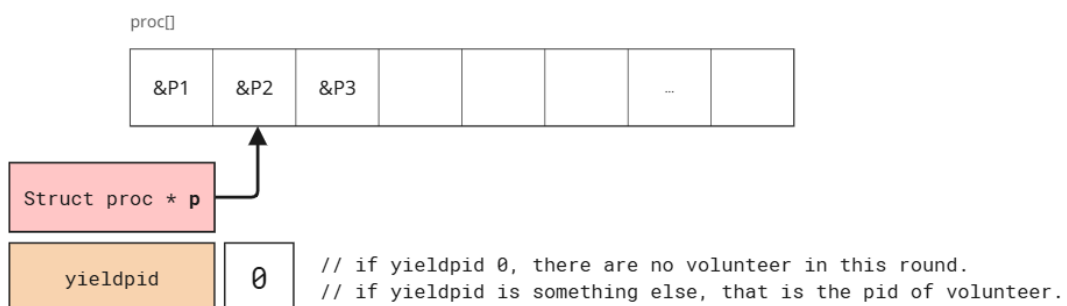


P1 is RUNNABLE;
 P1 is not the volunteer;
 Switch to P1
 reset p to proc;

While executing p1, p1 called yield **system call**

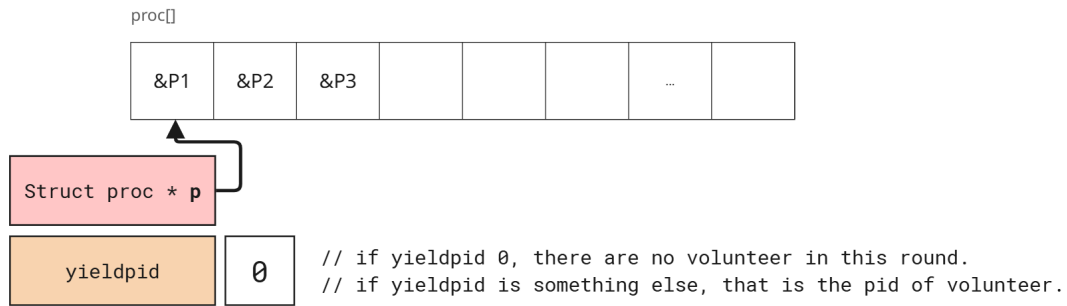


P1 is RUNNABLE;
 P1 **is** the volunteer;
 P1 should be excluded in scheduling this round.
 reset yieldpid;
 p++;
 NEXT LOOP



P2 is RUNNABLE;
 P2 is not the volunteer;
 Switch to P2;
 reset p to proc;

While executing p2, Timer interrupt called yield()



```
P1 is RUNNABLE;  
P1 is not the volunteer;  
Switch to P1  
reset p to proc;
```

miro

위 그림처럼 P1이 yield system call로 yield() 를 호출했다면, 그 Round에는 P1을 **배제해야 한다**. 그 이후에 다시 timer interrupt에 의해 yield() 가 호출됐다면, 그때 다시 P1을 선택하면 된다.

왜냐하면 P1은 yield 를 한 것이지, SLEEP 이나 TERMINATED 된 것이 아니기 때문이다. yield System call이 호출된 시점의 Scheduling Round에서만 배제하면 된다.

3. MLFQ (Multiple level feedback queue) & Priority Scheduling

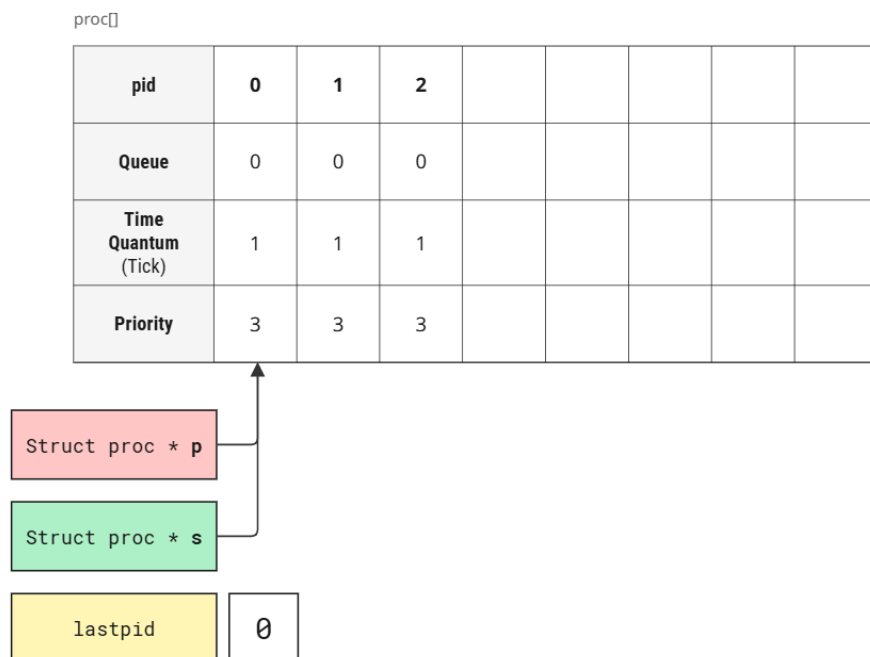
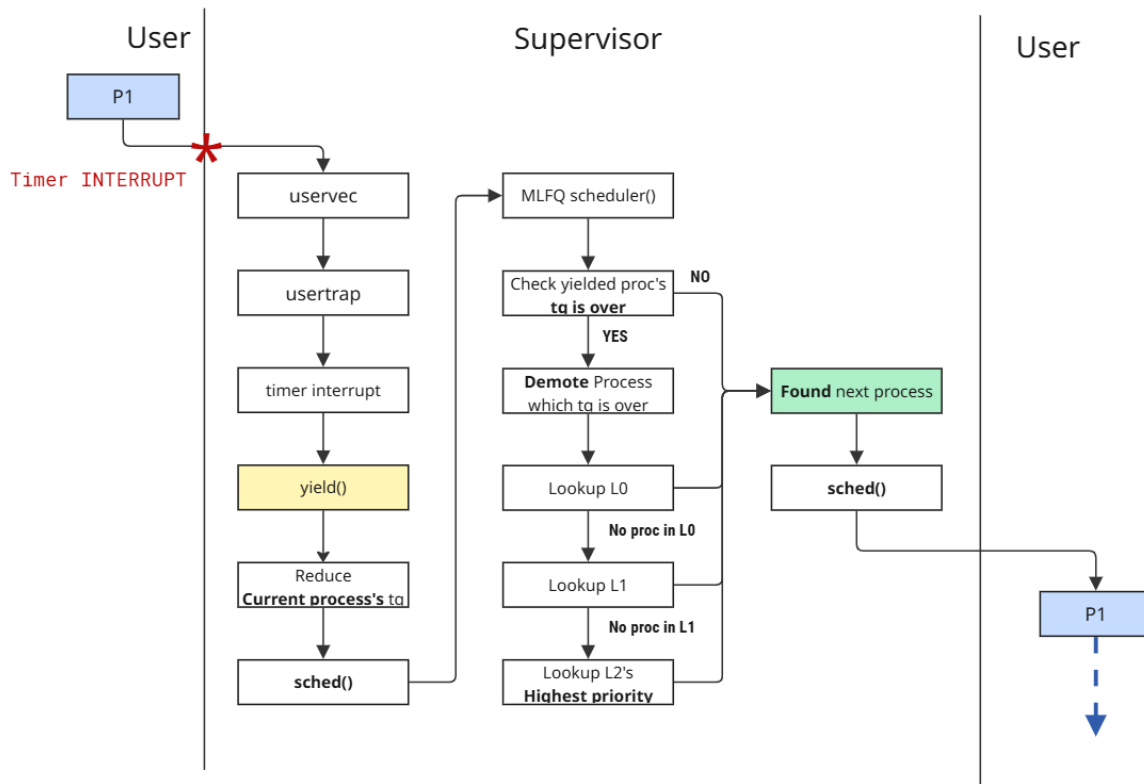
1. Specification

- 각 프로세스는 3개의 Queue(L0, L1, L2)에 속한다.
 - L0 : 1tick의 time quantum
 - L1 : 3tick의 time quantum
 - L2 : 5tick의 time quantum, priority 적용됨.
- 새로운 프로세스 생성시 L0으로 배치된다.
- 각 Queue에서의 RUNNABLE 한 프로세스가 없으면, 하위 큐에서 찾는다.
- 각 Queue에서 time quantum을 전부 소모한 프로세스는 Demotion 한다.
- L2 Queue는 Priority가 적용되며, 높은 Priority를 가진 프로세스가 먼저 실행된다. (3~0 순)
- Global tick이 50이 될 때, Priority Boosting이 발생한다.
 - 모든 프로세스의 Queue를 0으로 초기화하며, Time quantum 역시 초기화 한다.

2. Algorithm

MLFQ 스케줄러를 어떻게 구현해야 할까?

MLFQ는 L0 ~ L3 큐를 순회하며 RUNNABLE 한 프로세스를 찾아야 한다. 따라서 한번의 Iteration 만으로 가능했던 FCFS와는 달리 Iteration이 여러번 일어나야 한다.



miro

위 삽화는 MLFQ 프로세스의 동작을 도식화한 것이다. `s` 는 `p` 를 통한 프로세스 테이블의 탐색이 마무리 된 이후, Context를 switch 할 프로세스를 가리킨다. `lastpid` 는 FCFS의 `yieldpid` 와 유사하나, `lastpid` 의 경우 `yield` system call과 `timer interrupt`로 인해 호출된 `yield()` 를 구분하지 않고, `yield` 한 프로세스의 `pid`를 저장하고 있는 변수다.

크게 3가지 부분을 중점적으로 살펴보자.

- 프로세스의 PCB에는 어떤 데이터가 추가되어야 하는가?
 - 현재 `queue number`, 현재 사용중인 `Time quantum`, L2 큐 스케줄링을 위한 `priority` 가 필요하다.
- 프로세스의 `tq(time quantum)`는 언제 소모되어야 하는가?
 - `Timer interrupt`가 실행되어 `yield()` 가 호출된 시점에, 프로세스를 `RUNNABLE` 로 만들기 전 소모하면 된다.
- Iteration은 몇 번 일어나야 하는가?

- Scheduler 내부에서 최소 **1번**, 많으면 **9번** 일어날 수 있다.
 - 프로세스 전체의 갯수가 64개 이하이고, Queue 갯수가 3개, 그 중 Priority를 쓰는 L2 Queue의 경우 Priority 가 0, 1, 2, 3 4개 이므로
 - 1(priority boosting) + 1 (tq 소모가 다 안됐을 경우) + 1 (tq 소모 다 한 프로세스 Demotion) + 1 (L0) + 1(L1) + 4(L2 priority 4개) = 9
 - 최대 **576 cycle** 이내에 MLFQ 스케줄링은 마무리 된다.
 - 개발 머신이 2.1Ghz frequency를 가지므로, wsl, qemu등의 속도 저해 요소를 배제하고 xv6가 본 머신의 main OS일 경우 100ms 이내에 스케줄링이 끝난다.

위 모형을 바탕으로 MLFQ 스케줄러의 **Pseudo code**는 아래와 같다.


```

MLFQ Scheduler(){
    loop forever...

    struct proc * p // iterator
    struct proc * select // process I want to swtch()
    int selected =FALSE

    Iterating process table with p...
    if (RUNNABLE yielded process's time quantum still left)
        select = p
        selected = TRUE
        break

    if selected = FALSE, Iterating process table with p...
    if (RUNNABLE yielded process's time quantum is over)
        Demote(p)

    if selected = FALSE, Iterating process table with p...
    if (p is L0 queue and RUNNABLE)
        select = p
        selected = TRUE
        break

    if selected = FALSE, Iterating process table with p...
    if (p is L1 queue and RUNNABLE)
        select = p
        selected = TRUE
        break

    if selected = FALSE, Iterating process table with p...
    if (p is L2 queue and Priority is 3 and RUNNABLE)
        select = p
        selected = TRUE
        break

    // same as Priority is 2, 1, 0

    if selected = TRUE
        swtch(selected)
}

```

- Scheduler가 호출되면, 다음에 선택될 프로세스 select 를 선택하기 위해 Process Table을 여러번 iteration하게 된다.
 - 만일 yield() 를 호출한 프로세스의 Time quantum이 남아있다면, 그 프로세스를 그대로 선택하면 된다.
 - 따라서 yield() 를 호출한 Process의 pid를 Scheduler는 인지하고 있어야 한다. 이를 lastpid 로 정의하였으며, FCFS의 yieldpid 와는 다르게 lastpid 는 System call과 Timer Interrupt를 구분하지 않고 해당 프로세스의 pid를 저장한다.
- 본 project01에서는 MLFQ 스케줄러 하에서 System call에 의해 yield() 를 호출한 프로세스의 Time quantum 변화에 대해 기술되어 있지 않다.

- 따라서 MLFQ에서의 yield system call 호출이 일어났을 때에도 일괄적으로 Time quantum을 감소시켰다.

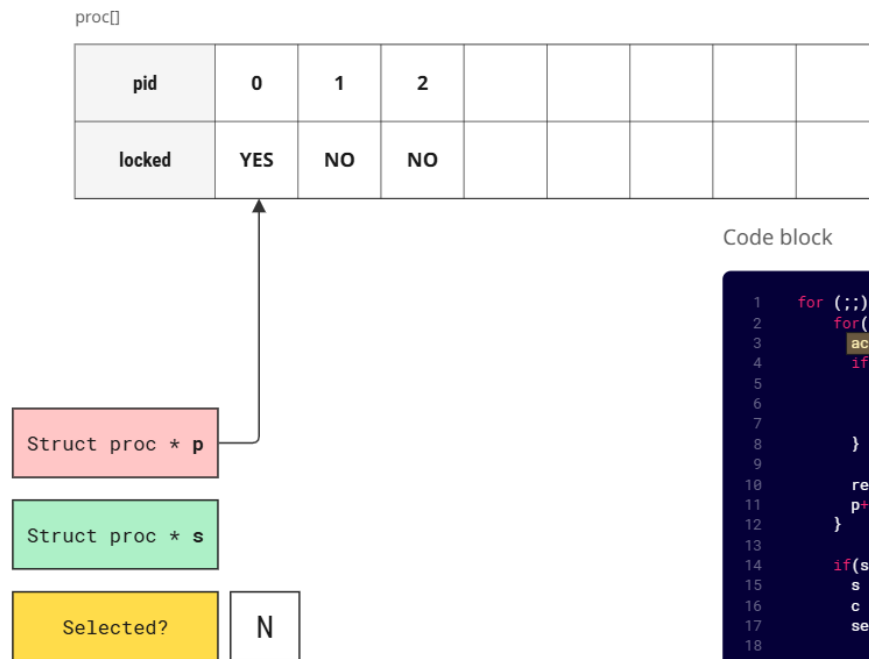
Multiple iteration - lock problem

스케줄러 내부에서 여러번 Iteration을 하려면, lock을 **acquire**하고, **release** 하는 시점에 대한 이해가 필요하다. 우선 iterator p 가 process table을 iterating 하는 도중, lock을 얻는 이유는 PCB 내의 데이터는 **Critical section**이기 때문으로, 해당 값을 변화시키는 것은 **atomic** 하게 동작해야 하기 때문이다.

따라서 Iteration 종료 이후 `swtch()` 를 하려면, 해당 프로세스의 lock이 언제 잠기고 풀리는지 파악할 필요가 있다. 아래 그림은 Iteration 바깥에서 context switch를 하는 중, 해당 프로세스의 lock이 어떻게 다뤄지는지에 대한 과정을 묘사하고 있다.

SCHEDULER가 CONTEXT SWITCHING 할 PROCESS를 선택한 경우

swtch() outside the loop - case 1 (if p0 is runnable)

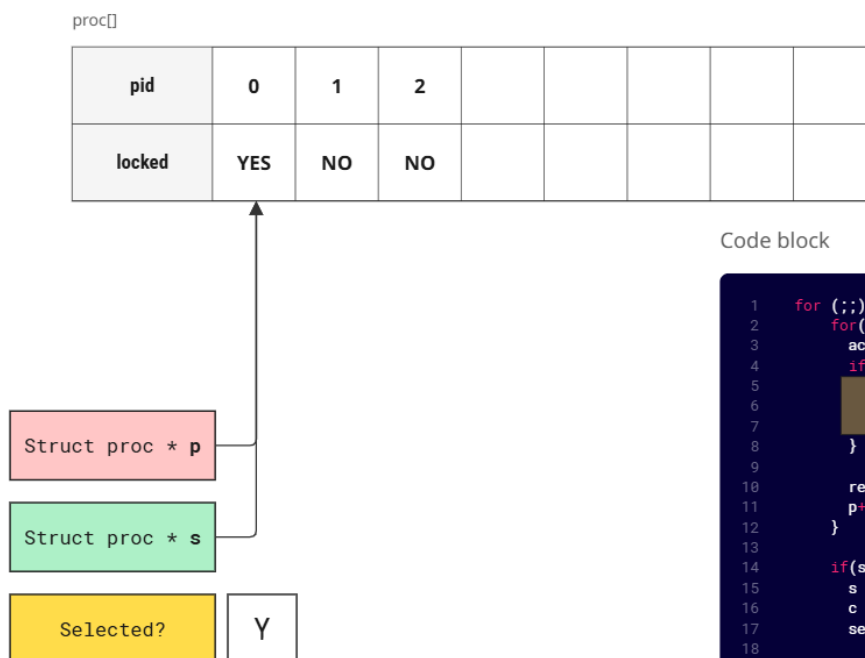


Code block

```

1  for (;;) {
2      for (p = proc; p < &proc[NPROC];) {
3          acquire(&p->lock);
4          if (p->state == RUNNABLE) {
5              s = p;
6              selected = 1;
7              break;
8          }
9
10         release(&p->lock);
11         p++;
12     }
13
14     if (selected == 1) {
15         s->state = RUNNING;
16         c->proc = s;
17         selected = 0;
18
19         swtch(&c->context, &s->context);
20         c->proc = 0;
21         found = 1;
22         release(&s->lock);
23     }
24 }

```

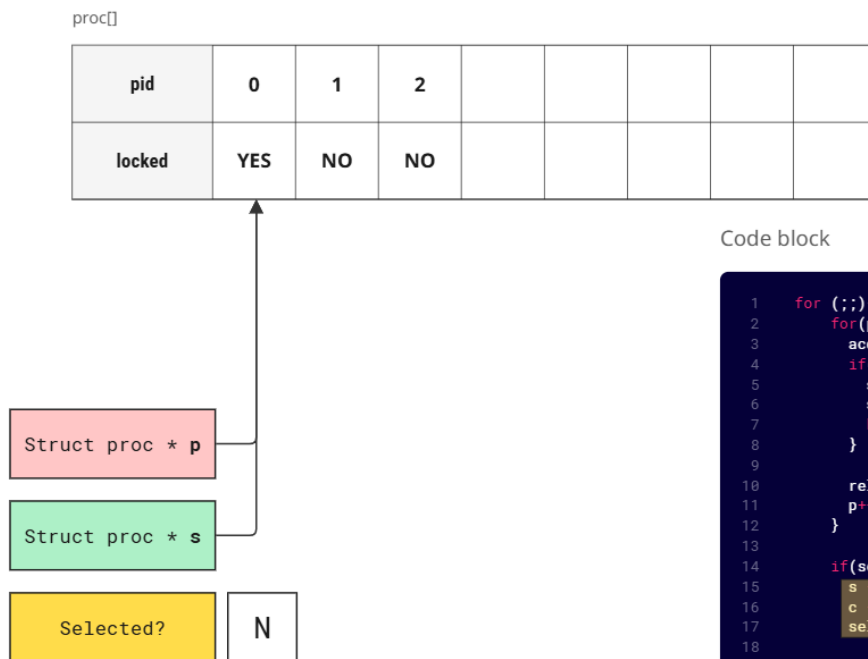


Code block

```

1  for (;;) {
2      for (p = proc; p < &proc[NPROC];) {
3          acquire(&p->lock);
4          if (p->state == RUNNABLE) {
5              s = p;
6              selected = 1;
7              break;
8          }
9
10         release(&p->lock);
11         p++;
12     }
13
14     if (selected == 1) {
15         s->state = RUNNING;
16         c->proc = s;
17         selected = 0;
18
19         swtch(&c->context, &s->context);
20         c->proc = 0;
21         found = 1;
22         release(&s->lock);
23     }
24 }

```

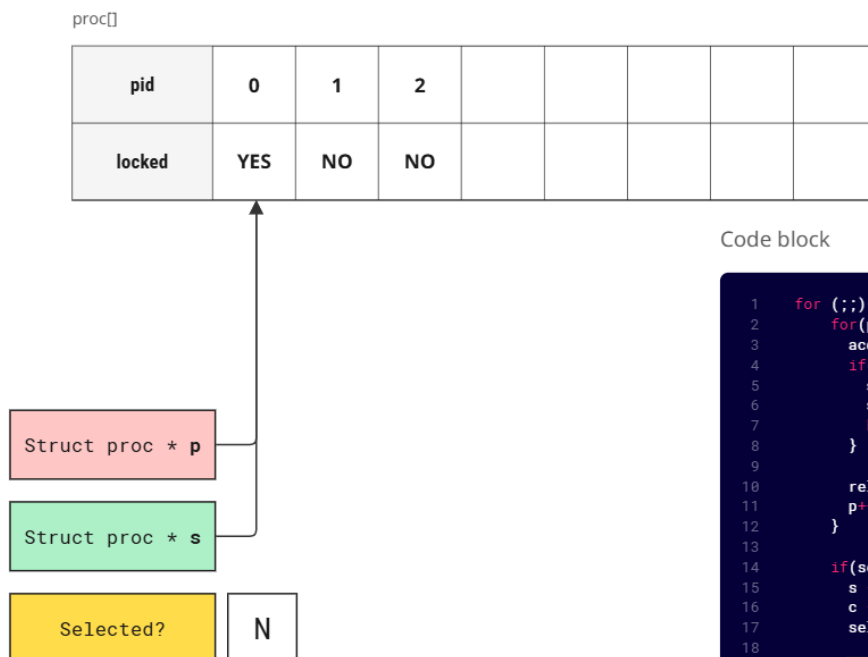


Code block

```

1  for (;;) {
2      for (p = proc; p < &proc[NPROC];) {
3          acquire(&p->lock);
4          if (p->state == RUNNABLE) {
5              s = p;
6              selected = 1;
7              break;
8          }
9      }
10     release(&p->lock);
11     p++;
12 }
13
14 if (selected == 1) {
15     s->state = RUNNING;
16     c->proc = s;
17     selected = 0;
18 }
19
20 switch(&c->context, &s->context);
21 c->proc = 0;
22 found = 1;
23 release(&s->lock);
24 }

```



Code block

```

1  for (;;) {
2      for (p = proc; p < &proc[NPROC];) {
3          acquire(&p->lock);
4          if (p->state == RUNNABLE) {
5              s = p;
6              selected = 1;
7              break;
8          }
9      }
10     release(&p->lock);
11     p++;
12 }
13
14 if (selected == 1) {
15     s->state = RUNNING;
16     c->proc = s;
17     selected = 0;
18 }
19
20 switch(&c->context, &s->context);
21 c->proc = 0;
22 found = 1;
23 release(&s->lock);
24 }

```

proc[]

pid	0	1	2					
locked	NO	NO	NO					

Struct proc * p

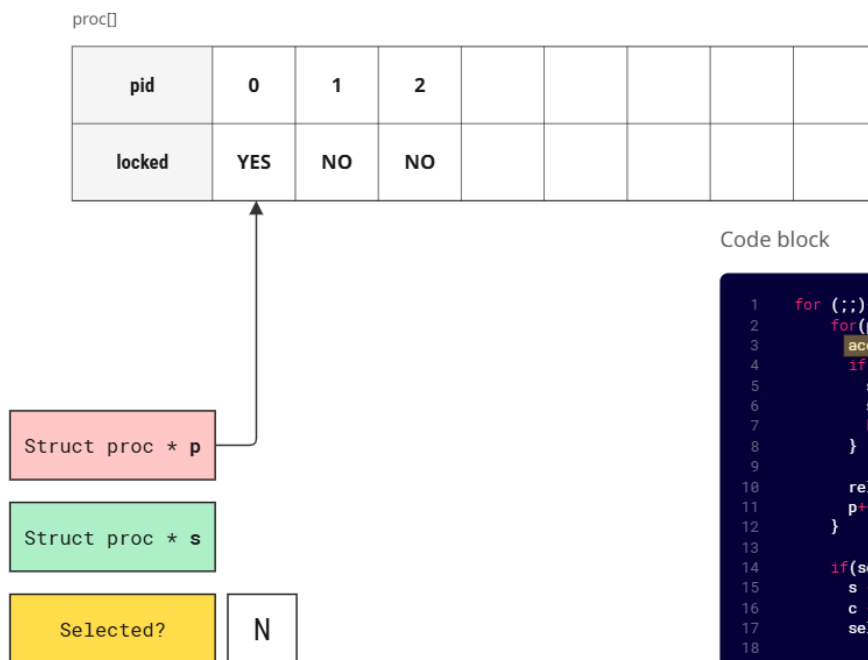
Code block

```
1 void
2 yield(void)
3 {
4
5     struct proc *p = myproc();
6     acquire(&p->lock);
7     p->state = RUNNABLE;
8     sched();
9     release(&p->lock);
10 }
```

miro

SCHEDULER가 CONTEXT SWITCHING을 할 PROCESS를 선택하지 못한 경우

swtch() outside the loop - case 2 (if p0 is not runnable)

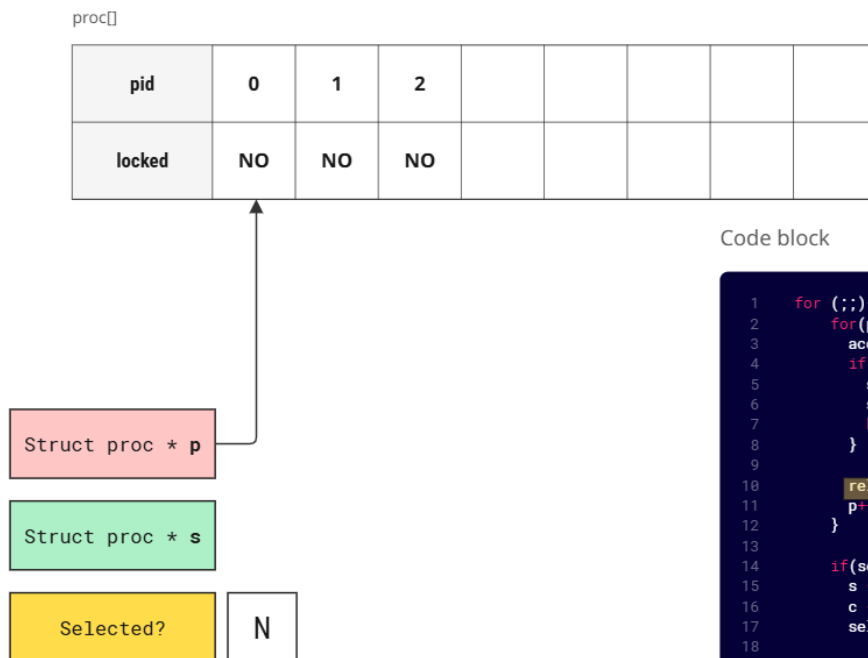


Code block

```

1  for (;;) {
2      for (p = proc; p < &proc[NPROC];) {
3          acquire(&p->lock);
4          if (p->state == RUNNABLE) {
5              s = p;
6              selected = 1;
7              break;
8          }
9
10         release(&p->lock);
11         p++;
12     }
13
14     if (selected == 1) {
15         s->state = RUNNING;
16         c->proc = s;
17         selected = 0;
18
19         swtch(&c->context, &s->context);
20         c->proc = 0;
21         found = 1;
22         release(&s->lock);
23     }
24 }

```

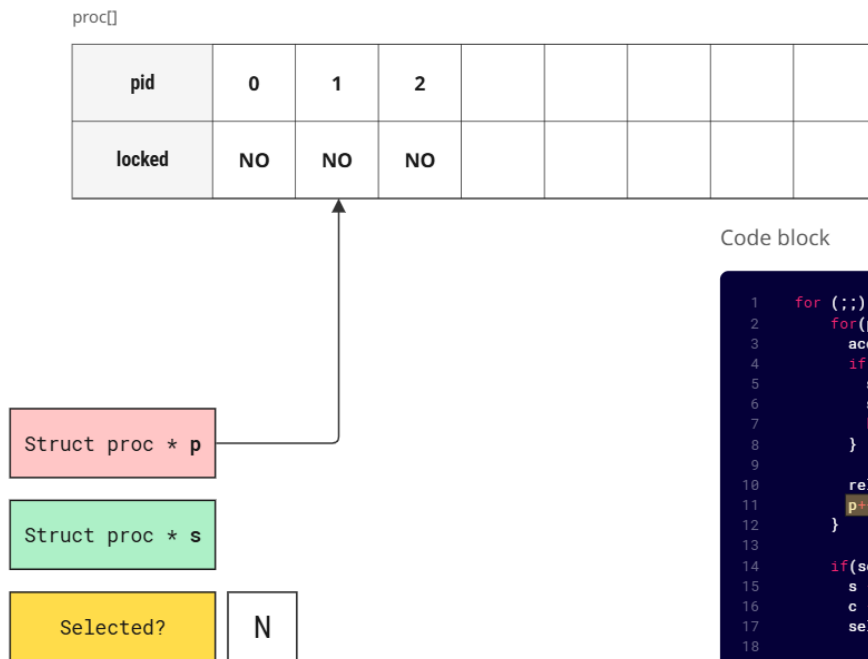


Code block

```

1  for (;;) {
2      for (p = proc; p < &proc[NPROC];) {
3          acquire(&p->lock);
4          if (p->state == RUNNABLE) {
5              s = p;
6              selected = 1;
7              break;
8          }
9
10         release(&p->lock);
11         p++;
12     }
13
14     if (selected == 1) {
15         s->state = RUNNING;
16         c->proc = s;
17         selected = 0;
18
19         swtch(&c->context, &s->context);
20         c->proc = 0;
21         found = 1;
22         release(&s->lock);
23     }
24 }

```

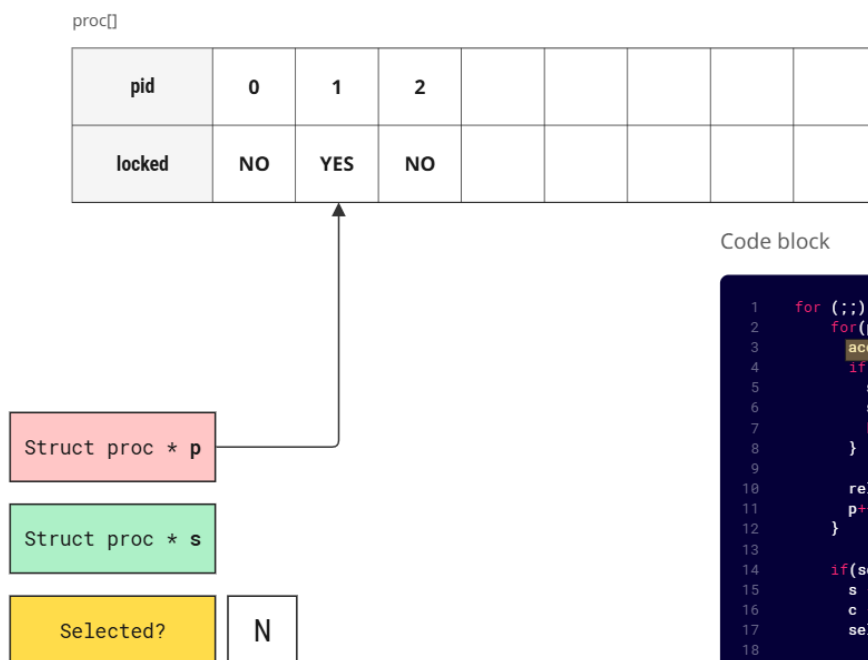


Code block

```

1  for (;;) {
2      for (p = proc; p < &proc[NPROC];) {
3          acquire(&p->lock);
4          if (p->state == RUNNABLE) {
5              s = p;
6              selected = 1;
7              break;
8          }
9
10         release(&p->lock);
11         p++;
12     }
13
14     if (selected == 1) {
15         s->state = RUNNING;
16         c->proc = s;
17         selected = 0;
18
19         switch(&c->context, &s->context);
20         c->proc = 0;
21         found = 1;
22         release(&s->lock);
23     }
24 }

```



Code block

```

1  for (;;) {
2      for (p = proc; p < &proc[NPROC];) {
3          acquire(&p->lock);
4          if (p->state == RUNNABLE) {
5              s = p;
6              selected = 1;
7              break;
8          }
9
10         release(&p->lock);
11         p++;
12     }
13
14     if (selected == 1) {
15         s->state = RUNNING;
16         c->proc = s;
17         selected = 0;
18
19         switch(&c->context, &s->context);
20         c->proc = 0;
21         found = 1;
22         release(&s->lock);
23     }
24 }

```

And So on...

miro

위 코드를 바탕으로, Pseudo code의 **multiple iteration**을 실현할 수 있다.

4. Mode Switch between FCFS & MLFQ Mode

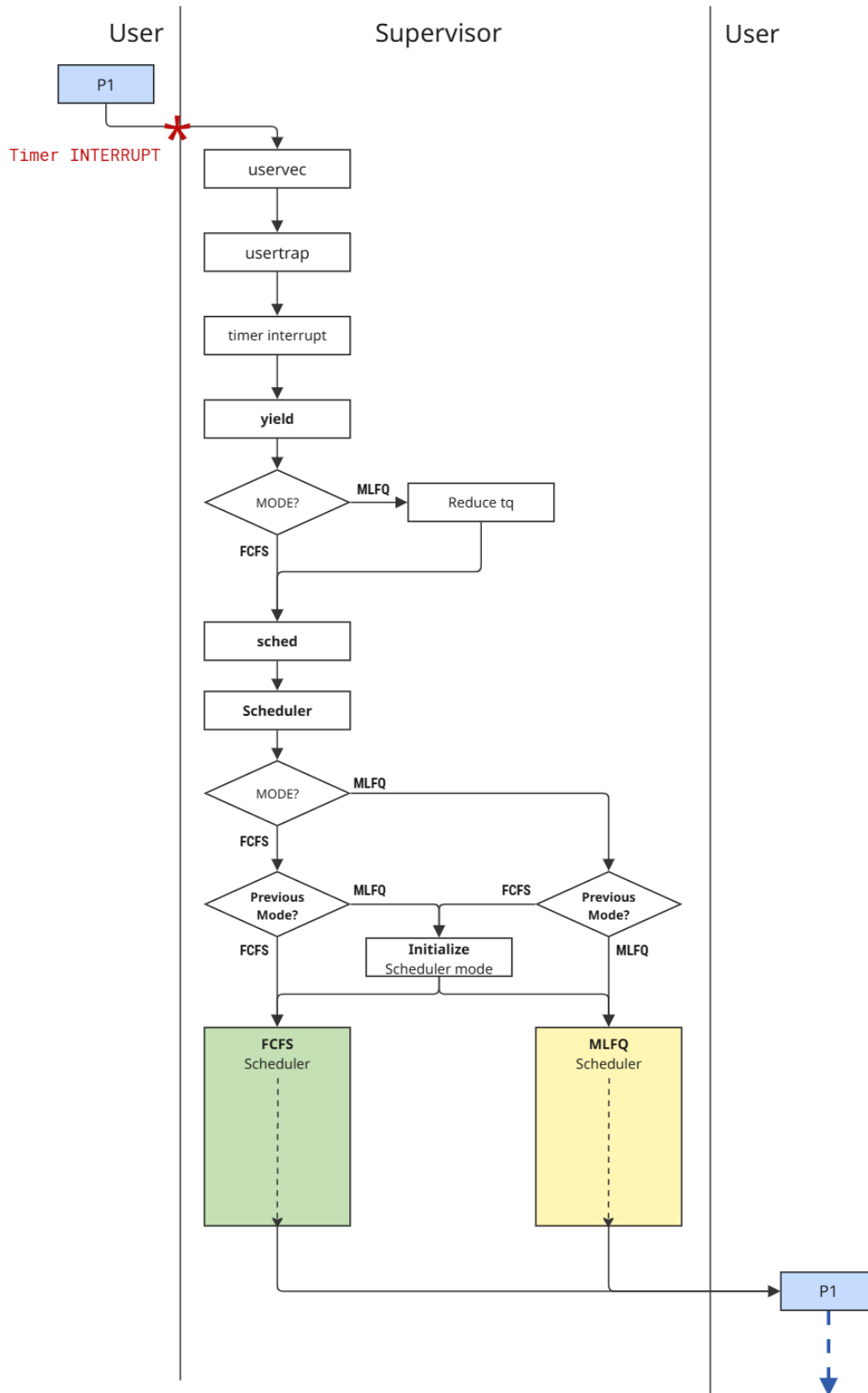
1. Specification

- 부팅 시 초기 스케줄러 모드는 FCFS이다.

- FCFS 모드 하에서 Priority boosting은 발생하지 않는다.
- 모드간의 변화가 발생하면 Global tick count는 0으로 초기화 된다.
- 동일한 모드로의 변화는 에러 메시지를 출력하며, 아무런 변화가 일어나지 않는다.
- MLFQ → FCFS (`fcfsmode()`)
 - 모든 프로세스의 Queue, time quantum, Priority를 -1로 초기화한다.
 - 다음부터 프로세스의 Scheduling은 오직 Creation time에 의존한다(FCFS).
- FCFS → MLFQ (`mlfqmode()`)
 - 모든 프로세스는 L0 Queue로, Priority는 1로 초기화한다.
 - 다음부터 프로세스의 Scheduling은 MLFQ에 따른다.

2. Algorithm

FCFS Scheduler와 MLFQ Scheduler의 통합



miro

위 다이어그램은 FCFS와 MLFQ 스케줄러를 통합한 것이다. 몇가지 살펴볼 점이 있다.


- **Previous mode**는 왜 검사하는가?

- FCFS Scheduler는 **for loop** 내부에서 선택될 프로세스를 정하고, `swtch()` 까지 진행한다.
- 따라서 다음 Scheduling round 가 도달하기 전 모드가 MLFQ로 바뀌더라도, 여전히 Scheduler의 context는 **FCFS의 loop**를 도는 상태로 복귀한다.
 - 이를 예방하기 위해 FCFS에서 MLFQ로 모드 변환이 발생한 경우 의도적으로 FCFS의 loop를 깨고, 다시 Scheduling Mode를 검사하는 절차가 필요하다.

Scheduler with Dual modes - pseudo code

```
Scheduler(){  
    loop forever...  
  
    if mode is FCFS, you have to schedule like this  
    Iterating Process Table ....  
    swtch() loop_FCFS  
  
    if mode is MLFQ, you have to schedule like this  
    Iterating Process Table ... loop_MLFQ  
    swtch()  
  
}
```

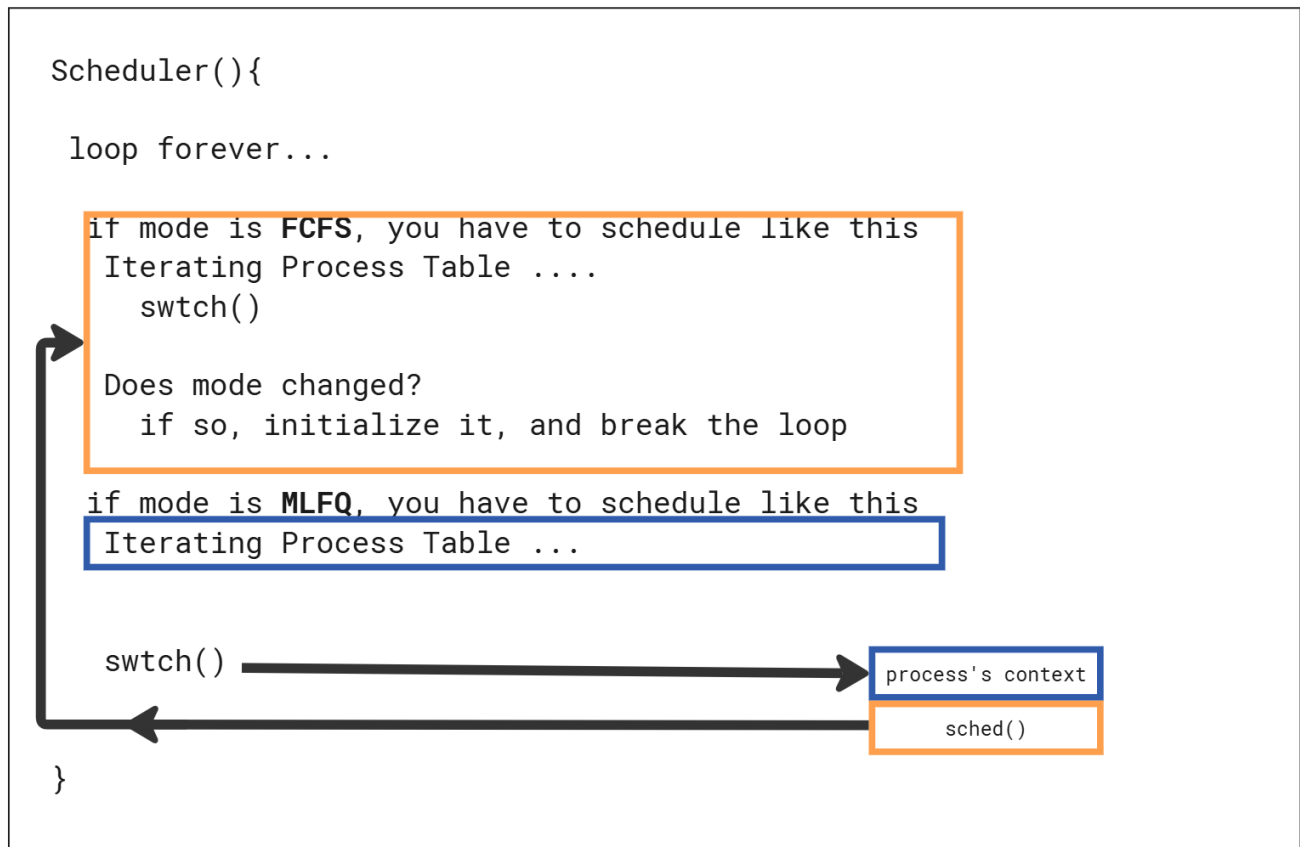
Scheduler with Dual modes - if the mode changes inside of process (FCFS to MLFQ)

```
Scheduler(){  
    loop forever...  
  
    if mode is FCFS, you have to schedule like this  
    Iterating Process Table ....  
    swtch()   
  
    Does mode changed? Yes.  
    initialize it, and break the loop  
  
    if mode is MLFQ, you have to schedule like this  
    Iterating Process Table ...  
    swtch()  
  
}
```

miro

- MLFQ 역시 Scheduling round에 도달하기 전 FCFS로 바뀌었을 경우를 고려해야 하나, MLFQ의 경우에는 `swtch()` 가 loop 바깥에 있기 때문에 명시적으로 loop를 깨라고 지시할 필요가 없다.
 - MLFQ의 경우에는 `swtch()` 가 발생하면 MLFQ 분기를 빠져나오며, 다시 검사가 가능하다.

Scheduler with Dual modes - if the mode changes inside of process (MLFQ to FCFS)



miro

Implementation

1. FCFS Scheduler

전역변수 및 함수 설명

kernel/proc.c

```
1 // =====
2 // 기존 코드 생략..
3 // =====
4
5 // If yield is called by syscall, this is the pid of the process
6 int yieldpid = 0;
7 struct spinlock yieldpid_lock;
8
9 // =====
10 // 기존 코드 생략..
11 // =====
12
13 // initialize the proc table.
14 void
15 procinit(void)
16 {
17     struct proc *p;
18
19     initlock(&pid_lock, "nextpid");
20     initlock(&wait_lock, "wait_lock");
21     initlock(&schedmode_lock, "schedmode_lock");
22     initlock(&yieldpid_lock, "yieldpid_lock");
23     initlock(&lastpid_lock, "lastpid_lock");
24 }
```

```

25     for(p = proc; p < &proc[NPROC]; p++) {
26         initlock(&p->lock, "proc");
27         p->state = UNUSED;
28         p->qnum = FCFSMODE;
29         p->tq = FCFSMODE;
30         p->priority = FCFSMODE;
31         p->kstack = KSTACK((int) (p - proc));
32     }
33 }
34
35 // =====
36 // 기존 코드 생략..
37 // =====
38
39 // 0 : nothing yielded voluntarily
40 // else : pid of the process that yielded on a syscall
41 int
42 yieldp(void)
43 {
44     int pid;
45     acquire(&yieldpid_lock);
46     pid = yieldpid;
47     release(&yieldpid_lock);
48     return pid;
49 }
50
51 // 0 : default(no yield)
52 // else : pid of the process that yielded on a syscall
53 int
54 setyieldpid(int pid)
55 {
56     acquire(&yieldpid_lock);
57     yieldpid = pid;
58     release(&yieldpid_lock);
59     return yieldpid;
60 }

```

- Variables

Type	Name	역할	사용
int	yieldpid	FCFS 모드 중, System call로 yield한 프로세스의 pid를 저장하는 함수	proc.c
struct spinlock	yieldpid_lock	yieldpid 는 한번에 하나의 프로세스만이 설정해야 하므로, 이를 위한 lock	proc.c

- Functions

Return value	Name(args)	역할	사용
int	yieldp(void)	Scheduling round에서 혹시 System call에 의한 yielded process가 있는지 확인하는 함수. 0 : 해당 라운드에 yield Syscall로 yield한 프로세스 없음. 그 외 값 : yield Syscall로 yield한 프로세스의 pid	proc.c
int	setyieldpid(int pid)	Syscall에 의한 yield일 경우, 이 함수를 호출하여 yieldpid 를 Set	mysyscall.c

scheduler()

```

1 void
2 scheduler(void)
3 {
4     struct proc *p;
5     struct cpu *c = mycpu();
6
7     // FCFS only

```

```

8  struct proc *fcfs_select = 0;
9  int fcfs_selected = 0;
10
11  c->proc = 0;
12  for(;;){
13
14      intr_on();
15      int found = 0;
16      for(p = proc; p < &proc[NPROC];) {
17          acquire(&p->lock);
18          if(p->state == RUNNABLE) {
19              if(yieldp() == 0){ // if nothing voluntarily yielded
20                  fcfs_select = p;
21                  fcfs_select -> state = RUNNING;
22                  c -> proc = fcfs_select;
23                  fcfs_selected = 1;
24                  swtch(&c->context, &fcfs_select->context);
25              }
26              else{ // if something voluntarily yielded, you have to respect that intention
27                  if(p->pid != yieldp()){
28                      fcfs_select = p;
29                      fcfs_select -> state = RUNNING;
30                      c -> proc = fcfs_select;
31                      fcfs_selected = 1;
32                      setyieldpid(0); // We handled this, so reset it
33                      swtch(&c->context, &fcfs_select->context);
34                  }
35              }
36              c->proc = 0;
37              found = 1;
38              release(&p->lock);
39
40              if(fcfs_selected == 1){
41                  // if we select a process, p should be reset.
42                  p = proc;
43                  fcfs_selected = 0;
44              }else {
45                  // if we are not able to find a process, we should move to the next one.
46                  p++;
47              }
48          }
49          if(found == 0) {
50              // nothing to run; stop running on this core until an interrupt.
51              intr_on();
52              asm volatile("wfi");
53          }
54      }

```

- Local variables

Type	Name	역할
struct proc *	p	Process table의 iterator.
struct proc *	fcfs_select	Scheduling round 종료 시점에 swtch() 할 프로세스를 가리키는 포인터.
int	fcfs_selected	Iteration 종료 시점에 fcfs_select 가 정해졌는지 아닌지 여부를 나타내는 변수. 0 : 선택되지 않음 (default) 1 : 선택 됨

- Specification 반영사항

Specification	Line Number	Description
기존 Scheduler의 변형일 것	전체	Timer interrupt를 끄는 방향으로 구현하지 않았음
PCB에 어떠한 조작도 가하지 않을 것 (변수 추가 등)	proc.h	FCFS만을 위한 변수 추가는 없었음

Specification	Line Number	Description
생성 순서대로 프로세스 선택할 것	45~52	Process가 생성 순서대로 프로세스 테이블에 들어온다는 성질을 이용, Iterator를 움직이지 말지 결정하며 스케줄링을 구현함
프로세스가 선택되면, TERMINATE 되거나 스스로 yield 하지 않는 한 계속 프로세스 실행 권한을 유지할 것.	22, 30	Syscall에 의한 yield를 구별하였음.

- Design 반영사항

Design	Line Number	Description
어떻게 FCFS 스케줄러를 구현할까?	전체	Scheduler의 변형으로 구현하였음.
Modifying the iterator, p	45~52	다음 프로세스의 선택 여부에 따라 Iterator의 이동 여부가 결정됨.
System call을 통해 특정 프로세스가 yield 한 경우, FCFS 스케줄러는 어떻게 동작해야 할까?	22, 30	System call을 통해 yield한 프로세스를 scheduling round에서 배제.

Related System call

kernel/mysyscall.c

```
1 // =====
2 // 기존 코드 생략..
3 // =====
4
5 void
6 sys_yield(void){
7     setyieldpid(myproc()->pid);
8     yield();
9     return;
10 }
```

- Functions

Return value	Name(args)	역할	사용
void	sys_yield(void)	yield() 의 wrapper function. setyieldpid() 를 통해 호출한 프로세스의 pid를 저장한다.	User program

2. MLFQ Scheduler

전역변수 및 함수 설명

kernel/proc.h

```
1 // =====
2 // 기존 코드 생략..
3 // =====
4
5 // Per-process state
6 struct proc {
7
8     // =====
9     // 기존 코드 생략..
10    // =====
11
12    int tq;                // Time quantum (1,3,5)
13    int qnum;              // Queue num(0-2)
14    int priority;          // Process priority (0-3)
15 }
```

```

16 // =====
17 // 기존 코드 생략..
18 // =====
19 };
20
21 #define TQ_Q0 1
22 #define TQ_Q1 3
23 #define TQ_Q2 5
24 #define FCFSMODE -1
25 #define TIMEQUANTUM(x) ((x) == 0 ? TQ_Q0 : (x) == 1 ? TQ_Q1 : (x) == 2 ? TQ_Q2 : FCFSMODE)

```

- Variables

변수 명	역할	사용
int tq	MLFQ 에서 프로세스 별 사용한 time quantum 값. 0에서 시작하여 점점 증가함.	proc.c
int qnum	MLFQ 에서 프로세스의 큐 위치, 0, 1, 2는 각각 L0, L1, L2 큐를 의미함.	proc.c , mysyscall.c
int priority	MLFQ, L2 큐 스케줄링에서 사용할 우선도 값 , 초기값은 3으로 시작함.	proc.c , mysyscall.c
TQ_QN	MLFQ에서 N번째 큐의 Time quantum limit 값	proc.c
TIMEQUANTUM(x)	MLFQ에서 qnum 을 넣으면 해당하는 TQ_Qn 값으로 치환됨. FCFS 모드일 경우 -1.	proc.c

kernel/proc.c

```

1 // =====
2 // 기존 코드 생략..
3 // =====
4
5 // which process was last scheduled
6 // only used in MLFQ
7 int lastpid = 0;
8 struct spinlock lastpid_lock;
9
10
11 // initialize the proc table.
12 void
13 procinit(void)
14 {
15     struct proc *p;
16
17     initlock(&pid_lock, "nextpid");
18     initlock(&wait_lock, "wait_lock");
19     initlock(&schedmode_lock, "schedmode_lock");
20     initlock(&yieldpid_lock, "yieldpid_lock");
21     initlock(&lastpid_lock, "lastpid_lock");
22
23     for(p = proc; p < &proc[NPROC]; p++) {
24         initlock(&p->lock, "proc");
25         p->state = UNUSED;
26         p->qnum = FCFSMODE;
27         p->tq = FCFSMODE;
28         p->priority = FCFSMODE;
29         p->kstack = KSTACK((int) (p - proc));
30     }
31 }
32
33 // =====
34 // 기존 코드 생략..
35 // =====
36
37 // Create a new process, copying the parent.
38 // Sets up child kernel stack to return as if from fork() system call.
39 int
40 fork(void)
41 {
42     int i, pid;

```

```

43 struct proc *np;
44 struct proc *p = myproc();
45
46 // =====
47 // 기존 코드 생략..
48 // =====
49
50 int mode = schedmode();
51 acquire(&np->lock);
52 if(mode == 0){
53     // FCFS
54     np->qnum = FCFSMODE;
55     np->tq = FCFSMODE;
56     np->priority = FCFSMODE;
57 }
58 else{
59     // MLFQ
60     np->qnum = 0;
61     np->tq = 0;
62     np->priority = 3;
63 }
64 np->state = RUNNABLE;
65 release(&np->lock);
66
67 return pid;
68 }
69
70 // =====
71 // 기존 코드 생략..
72 // =====
73
74 // Give up the CPU for one scheduling round.
75 void
76 yield(void)
77 {
78
79     struct proc *p = myproc();
80     acquire(&p->lock);
81
82     if(schedmode() == 1){
83         // MLFQ ONLY
84         int nexttq = (p -> tq) + 1;
85         if(nexttq >= TIMEQUANTUM(p->qnum)){
86             p -> tq = TIMEQUANTUM(p->qnum);
87         }else{
88             p -> tq = nexttq;
89         }
90         setlastpid(p->pid);
91     }
92
93     p->state = RUNNABLE;
94     sched();
95     release(&p->lock);
96 }
97
98 // =====
99 // 기존 코드 생략..
100 // =====
101
102 // returns lastpid
103 int
104 lastp(void){
105     int pid;
106     acquire(&lastpid_lock);
107     pid = lastpid;
108     release(&lastpid_lock);
109     return pid;
110 }
111
112 // when yield is called, lastpid stores the pid of the process which called yield

```



```

113 int
114 setlastpid(int pid)
115 {
116     acquire(&lastpid_lock);
117     lastpid = pid;
118     release(&lastpid_lock);
119     return lastpid;
120 }
121
122 // demote the process to the next queue
123 // only used in scheduler() because it assumes that the process is already in the lock
124 int
125 demoteproc(struct proc * p){
126     if(p->qnum == 0){
127         p->qnum = 1;
128     }else if(p->qnum == 1){
129         p->qnum = 2;
130         p->priority = 3;
131
132     }else if(p->qnum == 2){
133         p->qnum = 2;
134         int newpriority = (p->priority) - 1;
135
136         if (newpriority < 0){
137             p->priority = 0;
138         }else{
139             p->priority = newpriority;
140         }
141     }else{
142         return -1;
143     }
144     p->tq = 0;
145     return 0;
146 }
147
148 int
149 getlev(struct proc *p){
150     int mode = schedmode();
151     int lev;
152     if(mode == 1){
153         // MLFQ
154         acquire(&p->lock);
155         lev = p->qnum;
156         release(&p->lock);
157         return lev;
158     }
159
160     return 99;
161 }
162
163 int
164 setpriority(int pid, int np){
165     if(np < 0 || np > 3){
166         return -2;
167         // invalid priority
168     }
169
170     struct proc *p;
171     for(p = proc; p < &proc[NPROC];){
172         acquire(&p->lock);
173         if(p->pid == pid){
174             p->priority = np;
175             release(&p->lock);
176             return 0;
177         }
178         release(&p->lock);
179         p++;
180     }
181
182     return -1;

```

```

183     // not found
184
185 }

```

- Variables

Type	Name	역할	사용
int	lastpid	마지막으로 yield한 pid (Syscall, Timer interrupt 무관)	proc.c
struct spinlock	lastpid_lock	lastpid 의 atomicity를 보장하기 위한 lock	proc.c

- Functions

Return value	Name(args)	역할	사용
void	procinit(void)	프로세스 초기화 함수. 각종 lock 및 초기 프로세스의 PCB 구성요소를 초기화한다.	proc.c
int	fork(void)	새로운 프로세스를 만들 때 호출되는 함수. 모드에 따라 qnum, tq, priority 를 다르게 설정한다.	sysproc.c
void	yield(void)	MLFQ 모드일 때, 그 프로세스의 tq 를 하나 증가시킨 뒤, lastpid 를 그 프로세스의 pid 로 설정한다.	trap.c
int	lastp(void)	lastpid 를 atomic하게 return 할 때 쓰는 함수.	proc.c
int	setlastpid(int pid)	yield() 가 호출될 때, 호출한 프로세스의 pid 를 Set하는 함수.	proc.c
int	demoteproc(struct proc *p)	MLFQ 스케줄러 내부에서, tq 를 다 소모한 프로세스를 demote 하는 함수. 프로세스의 정보를 변경하는 만큼, 스케줄러 내부에서 lock이 걸려있다고 가정하고 그 안에서만 사용해야한다.	proc.c
int	getlev(struct proc *p)	프로세스의 qnum 을 return 하는 system call. FCFS 모드라면 99를 return 한다.	proc.c , mysyscall.c
int	setpriority(int pid, int np)	프로세스의 priority 를 변경하는 system call. Invalid priority : -2 not found a process with pid : -1 Successful execution : 0	proc.c , mysyscall.c

kernel/trap.c

```

1  // =====
2  // 기존 코드 생략..
3  // =====
4
5  struct spinlock tickslock;
6  uint ticks;
7
8  // =====
9  // 기존 코드 생략..
10 // =====
11
12 // reset ticks
13 void
14 resetticks(void)
15 {
16     acquire(&tickslock);
17     ticks = 0;
18     release(&tickslock);
19 }

```

- Variables

Type	Name	역할	사용
struct spinlock	tickslock	ticks 의 atomicity를 보장하기 위한 lock	trap.c
uint	ticks	Global tick	proc.c , trap.c

- Functions

Return value	Name(args)	역할	사용
void	resetticks(void)	Global tick을 0으로 초기화 한다.	proc.c

scheduler()

```

1  void
2  scheduler(void)
3  {
4      struct proc *p;
5      struct cpu *c = mycpu();
6
7      // MLFQ only
8      struct proc *mlfq_select = 0;
9      int mlfq_selected = 0;
10
11     c->proc = 0;
12     for(;;){
13
14         intr_on();
15         int found = 0;
16
17
18         // MLFQ
19         // if global tick is 50, reset all processes.
20         if(ticks == 50){
21             for(p = proc; p < &proc[NPROC];) {
22                 acquire(&p->lock);
23                 if(p->state == RUNNABLE){
24                     p->qnum = 0;
25                     p->tq = 0;
26                     p->priority = 3;
27                 }
28                 release(&p->lock);
29                 p++;
30             }
31             resetticks();
32         }
33
34         // Step 1. If yielded process is RUNNABLE and time quantum is not expired, select it.
35         for(p = proc; p < &proc[NPROC];) {
36             acquire(&p->lock);
37             if(p->pid == lastp() && p->pid != yieldp() && p->state == RUNNABLE && p->tq < TIMEQUANTUM(p->qnum)){
38                 mlfq_select = p;
39                 mlfq_selected = 1;
40                 break;
41             }
42
43             release(&p->lock);
44             p++;
45         }
46
47         // Step 2. In scheduler, reorder processes.
48         if(mlfq_selected == 0){
49             for(p = proc; p < &proc[NPROC];) {
50                 acquire(&p->lock);
51                 if(p->pid == lastp() && p->state == RUNNABLE && p->tq >= TIMEQUANTUM(p->qnum)){
52                     // if time quantum is expired, we should demote the process.
53                     demoteproc(p);
54                 }
55
56                 release(&p->lock);
57                 p++;
58             }

```

```

59 }
60
61 // Step 3. Find a process in L0 queue.
62 if(mlfq_selected == 0){
63     for(p = proc; p < &proc[NPROC];) {
64         acquire(&p->lock);
65         if(p->state == RUNNABLE && p->qnum == 0 && p->pid != yieldp()) {
66             mlfq_select = p;
67             mlfq_selected = 1;
68             break;
69         }
70
71         release(&p->lock);
72         p++;
73     }
74 }
75
76 // Step 4. If there are no process to run in L0 queue, Find a process in L1 queue.
77 if(mlfq_selected == 0){
78     for(p = proc; p < &proc[NPROC];) {
79         acquire(&p->lock);
80         if(p->state == RUNNABLE && p->qnum == 1 && p->pid != yieldp()) {
81             mlfq_select = p;
82             mlfq_selected = 1;
83             break;
84         }
85
86         release(&p->lock);
87         p++;
88     }
89 }
90
91 // Step 5. If there are no process to run in L1 queue, Find a process in L2 queue.
92 // L2 queue has a scheduling policy of priority_scheduling.
93 // Since we only have 4 priority levels and the maximum number of process is 64,
94 // we can just brute force it.
95
96 // L2 priority 3
97 if(mlfq_selected == 0){
98     for(p = proc; p < &proc[NPROC];) {
99         acquire(&p->lock);
100         if(p->state == RUNNABLE && p->qnum == 2 && p->priority == 3 && p->pid != yieldp()) {
101             mlfq_select = p;
102             mlfq_selected = 1;
103             break;
104         }
105
106         release(&p->lock);
107         p++;
108     }
109 }
110
111 // L2 priority 2
112 if(mlfq_selected == 0){
113     for(p = proc; p < &proc[NPROC];) {
114         acquire(&p->lock);
115         if(p->state == RUNNABLE && p->qnum == 2 && p->priority == 2 && p->pid != yieldp()) {
116             mlfq_select = p;
117             mlfq_selected = 1;
118             break;
119         }
120
121         release(&p->lock);
122         p++;
123     }
124 }
125
126 // L2 priority 1
127 if(mlfq_selected == 0){
128     for(p = proc; p < &proc[NPROC];) {
129         acquire(&p->lock);

```

```

129     if(p->state == RUNNABLE && p->qnum == 2 && p->priority == 1 && p->pid != yieldp()) {
130         mlfq_select = p;
131         mlfq_selected = 1;
132         break;
133     }
134
135     release(&p->lock);
136     p++;
137 }
138 }
139
140 // L2 priority 0
141 if(mlfq_selected == 0){
142     for(p = proc; p < &proc[NPROC];) {
143         acquire(&p->lock);
144         if(p->state == RUNNABLE && p->qnum == 2 && p->priority == 0 && p->pid != yieldp()) {
145             mlfq_select = p;
146             mlfq_selected = 1;
147             break;
148         }
149
150         release(&p->lock);
151         p++;
152     }
153 }
154
155
156 // Step 6. if selected, swtch it.
157 if(mlfq_selected == 1){
158     mlfq_select -> state = RUNNING;
159     c -> proc = mlfq_select;
160     mlfq_selected = 0;
161
162     swtch(&c->context, &mlfq_select->context);
163     c->proc = 0;
164     found = 1;
165     release(&mlfq_select->lock);
166 }
167
168 if(schedmode() != 1){
169     // if the mode is changed, we dont need to break the loop.
170     // because mlfq scheduling is done outside of the loop.
171 }
172 }
173 if(found == 0) {
174     // nothing to run; stop running on this core until an interrupt.
175     intr_on();
176     asm volatile("wfi");
177 }
178
179 }

```

- Local variables

Type	Name	역할
struct proc *	p	Process table의 iterator.
struct proc *	mlfq_select	Scheduling round 종료 시점에 swtch() 할 프로세스를 가리키는 포인터.
int	mlfq_selected	multiple Iteration 종료 시점에 mlfq_select 가 정해졌는지 아닌지 여부를 나타내는 변수. 0 : 선택되지 않음 (default) 1 : 선택 됨

- Specification 반영사항

Specification	Line Number	Description
3개의 queue	proc.h	Define을 통해 큐와, 큐의 time quantum을 정의함
새로운 프로세스 생성시 L0으로 배치	fork()	MLFQ 모드에서 프로세스 생성시 L0큐로 설정함
L0 → L1 → L2 순의 Search	64, 79, 99	순차적으로 Process table iteration
Time quantum 전부 소모한 Process의 Demotion	56	Time quantum 소모시 demoteproc 호출
L2의 Priority Scheduling	99, 114, 128, 143	Priority 높은 순으로 Iteration 배치
Priority Boosting	22	Global tick이 50이 되면 모든 프로세스를 L0으로 이동

- Design 반영사항

Design	Line Number	Description
MLFQ 스케줄러를 어떻게 구현할까? Multiple Iteration - lock problem	37, 50, 64, 79, 99, 114, 128, 143	Multiple Iteration과 mlfq_select의 lock을 통해 구현하였음.

Related System call

kernel/mysyscall.c

```
1 // =====
2 // 기존 코드 생략..
3 // =====
4
5 int
6 sys_getlev(void){
7     struct proc *p = myproc();
8     int lev = getlev(p);
9     return lev;
10 }
11
12 int
13 sys_setpriority(void){
14     int pid, np;
15     argint(0, &pid);
16     argint(1, &np);
17
18     return setpriority(pid, np);
19 }
20
```

- Functions

Return value	Name(args)	역할	사용
int	sys_getlev(void)	proc.c 의 getlev 를 호출하는 Wrapper function	User program
int	sys_setpriority(void)	User program에서는 int, int 의 argument를 전달함. 이를 받아서 proc.c 의 setpriority 함수를 실행함.	User program

3. Mode change

전역변수 및 함수 설명

kernel/proc.c

```
1 // =====
2 // 기존 코드 생략..
3 // =====
```

```

4 // scheduler mode
5 // 0: FCFS
6 // 1: MLFQ
7 int scheduler_mode = 0;
8 struct spinlock schedmode_lock;
9
10
11
12
13 // =====
14 // 기존 코드 생략..
15 // =====
16
17
18 // 0 : FCFS
19 // 1 : MLFQ
20 int
21 schedmode(void)
22 {
23     int mode;
24     acquire(&schedmode_lock);
25     mode = scheduler_mode;
26     release(&schedmode_lock);
27     return mode;
28 }
29
30 // 0 : FCFS
31 // 1 : MLFQ
32 int
33 mlfqmode(void){
34     acquire(&schedmode_lock);
35     if(scheduler_mode == 1){
36         release(&schedmode_lock);
37         return -1;
38     }
39
40     // FCFS -> MLFQ
41     // Move all processes to the first queue
42     for(struct proc *p = proc; p <= &proc[NPROC];){
43         acquire(&p->lock);
44         p->qnum = 0;
45         p->tq = 0;
46         p->priority = 3;
47         release(&p->lock);
48         p++;
49     }
50     scheduler_mode = 1;
51     resetticks();
52     release(&schedmode_lock);
53     return 0;
54 }
55
56 int
57 fcfsmode(void){
58     acquire(&schedmode_lock);
59     if(scheduler_mode == 0){
60         release(&schedmode_lock);
61         return -1;
62     }
63
64     // MLFQ -> FCFS
65     // Initialize all processes to FCFS
66     for(struct proc *p = proc; p <= &proc[NPROC];){
67         acquire(&p->lock);
68         p->qnum = FCFSMODE;
69         p->tq = FCFSMODE;
70         p->priority = FCFSMODE;
71         release(&p->lock);
72         p++;
73     }

```

```

74
75     scheduler_mode = 0;
76     resetticks();
77     release(&schedmode_lock);
78     return 0;
79 }
80
81 // Deprecated
82 // 0 : FCFS
83 // 1 : MLFQ
84 int
85 setschedmode(int mode)
86 {
87
88     acquire(&schedmode_lock);
89     if(mode == scheduler_mode){
90         release(&schedmode_lock);
91
92         return -1;
93     }else{
94
95         if(mode == 0){
96             // MLFQ -> FCFS
97             // Initialize all processes to FCFS
98             for(struct proc *p = proc; p <= &proc[NPROC];){
99                 acquire(&p->lock);
100                 p->qnum = FCFSMODE;
101                 p->tq = FCFSMODE;
102                 p->priority = FCFSMODE;
103                 release(&p->lock);
104                 p++;
105             }
106         }
107         else{
108             // FCFS -> MLFQ
109             // Move all processes to the first queue
110             for(struct proc *p = proc; p <= &proc[NPROC];){
111                 acquire(&p->lock);
112                 p->qnum = 0;
113                 p->tq = TQ_Q0;
114                 p->priority = 3;
115                 release(&p->lock);
116                 p++;
117             }
118         }
119
120         scheduler_mode = mode;
121         resetticks();
122         release(&schedmode_lock);
123         return 0;
124     }
125 }

```

- Variables

Type	Name	역할	사용
int	schedulermode	현재 Scheduler mode를 확인하기 위한 전역 변수	proc.c
struct spinlock	schedmode_lock	모드 변경 시 atomicity를 유지하기 위한 lock	proc.c

- Functions

Return value	Name(args)	역할	사용
int	schedmode(void)	현재 Scheduler mode를 atomic 하게 return 하는 함수	proc.c

Return value	Name(args)	역할	사용
int	mlfqmode(void)	현재 Scheduler mode를 MLFQ로 만드는 함수	mysyscall.c
int	fcfsmode(void)	현재 Scheduler mode를 FCFS로 만드는 함수	mysyscall.c
int	setschedmode(int mode)	현재 Scheduler mode를 int mode 에 따라 MLFQ, FCFS로 바꾸는 함수	Depracated

scheduler()

```
1 void
2 scheduler(void)
3 {
4     struct proc *p;
5     struct cpu *c = mycpu();
6
7
8     c->proc = 0;
9     for(;;){
10
11         intr_on();
12         int found = 0;
13
14         if(schedmode() == 0){
15             // FCFS
16             for(p = proc; p < &proc[NPROC];) {
17                 // FCFS Scheduling Loop
18                 if(schedmode() != 0){
19                     // if the mode is changed, we should break the loop.
20                     break;
21                 }
22             }
23
24             else{
25                 // MLFQ
26             }
27             if(found == 0) {
28                 // nothing to run; stop running on this core until an interrupt.
29                 intr_on();
30                 asm volatile("wfi");
31             }
32         }
33     }
```

- Specification 반영사항

Specification	Line Number	Description
부팅 시 스케줄러 모드는 FCFS이다	proc.c	scheduler_mode 를 0으로 초기화 함.
모드 전환시 Global tick Reset	mlfqmode() , fcfsmode()	모드 변경 이후에 resettick() 호출
동일한 모드로의 변화는 에러 메시지 출력 이후 아무런 변화를 일으키지 않음	mlfqmode() , fcfsmode()	Wrapper function 단위에서 구현함.
MLFQ → FCFS	fcfsmode() ,18	모든 프로세스의 Queue, time quantum, Priority를 -1로 초기화한다. 또한 기존 FCFS 스케줄링 루프 안에 있었을 경우 Loop를 Break 한다.
FCFS → MLFQ	mlfqmode()	모든 프로세스는 L0 Queue로, Priority는 1로 초기화한다.

- Design 반영사항

Design	Line Number	Description
FCFS 와 MLFQ 스케줄러의 통합	14, 24	동일한 Scheduler 함수 내에서 scheduler_mode 값에 따라 다르게 동작한다.
Previous mode의 검사 (FCFS)	18	MLFQ에서 FCFS로 모드 변환이 발생한 경우, loop를 break 한다.

kernel/mysyscall.c

```
1 // =====
2 // 기존 코드 생략..
3 // =====
4
5 int
6 sys_fcfsmode(void){
7     if(fcfsmode() == -1){
8         printf("The mode is already fcfs\n");
9         return -1;
10    }
11    printf("The mode is now fcfs\n");
12    yield();
13    return 0;
14 }
15
16 int
17 sys_mlfqmode(void){
18     if(mlfqmode() == -1){
19         printf("The mode is already mlfq\n");
20         return -1;
21    }
22    printf("The mode is now mlfq\n");
23    yield();
24    return 0;
25 }
26
27 // =====
28 // 기존 코드 생략..
29 // =====
```

- Functions

Return value	Name(args)	역할	사용
int	sys_fcfsmode(void)	scheduler_mode 를 0으로 바꾸려고 시도한다. 만일 이미 0이라면, 에러메시지를 출력하고 -1을 return 한다.	User program
int	sys_mlfqmode(void)	scheduler_mode 를 1로 바꾸려고 시도한다. 만일 이미 1이라면, 에러메시지를 출력하고 -1을 return 한다.	User program

Results

Troubleshooting