Operating Systems

Team Project
Lab 1 : Scheduler Simulator

Run in Linux

$ git clone https://github.com/zinirun/lab1_sched.git

담당교수	**최종무 교수님**
팀명	**신허지** (시너지)
팀원/학번	**허전진** (32164959) **신창우** (32162436)

목 차

역 할

허전진	1. RR 구현 2. STRIDE 구현	〈 공통 〉 1. Single, Multi Queue 구현 2. header, main UI 설계
신창우	1. FIFO 구현 2. MLFQ 구현	

1. 프로그램 구조

1-1. 프로젝트 폴더의 파일 리스트

```
root@pp-app:/home/ubuntu/os/lab1_sched# ls -l
total 28
drwxr-xr-x 2 root root  4096 Apr 18 19:53 include
-rw-r--r-- 1 root root 14702 Apr 18 19:53 lab1_sched.c
-rw-r--r-- 1 root root  1464 Apr 18 19:53 lab1_sched_test.c
-rw-r--r-- 1 root root   732 Apr 18 19:53 Makefile
```

- include: 스케줄러에 필요한 헤더가 포함된 directory
- lab1_sched.c: 스케줄러 알고리즘을 실질적으로 구현한 c file
- lab1_sched_test.c: 결과를 출력할 main 함수가 포함된 c file
- Makefile: make 커맨드를 사용하기 위해 gcc가 설정된 Makefile

1-2. lab1_sched.c에 작성된 함수

```
/// General functions
void setProcess(PROCESS* pointer);
int getSumST(PROCESS pc[]);
int getQueueSize(PROC_QUEUE* queue);
void processEnd(PROCESS* proc, int index);
void draw(int** arr, PROCESS pc[]);
void insertQueue(PROCESS proc, PROC_QUEUE* queue);
int calcTotalTickets(PROCESS pc[]);
int getGCD(int n, int m);
int getLCM(int n, int m);

/// Scheduler algorithm functions
void FIFO(PROCESS pc[]);
void FIFO_pop(PROC_QUEUE* SQ_pointer, PROCESS* pc, int time, int** arr);
void RR(PROCESS pc[], int tq);
void RR_pop(PROC_QUEUE* SQ_pointer, PROCESS pc[], int time, int** arr, int* flag);
void MLFQ(PROCESS pc[], int MLFQ_cnt);
int MLFQ_pop(PROC_QUEUE* MLFQ_ptr, PROCESS processArr[], int q_index, int time, int** arr, int MLFQ_cnt);
void STRIDE(PROCESS pc[]);
int getPidSmallStride(PROCESS pc[], int SchC[], int index);
void delCand(int* cand, int val);
```

2. 프로그램 설명

2-1. FIFO

FIFO(FCFS)는 먼저 도착하는 순서대로 프로세스를 스케줄링하는 기법이다. 도착한 순서(Arrival Time)대로 single queue에 삽입되고, 스케줄링되는 프로세스의 수행 시간(Service time)이 0이 되면, 다음으로 도착한 프로세스를 수행시킨다.

구현된 모든 스케줄링 알고리즘 (FIFO, RR, MLFQ, STRIDE)는 Queue와 Array(구현 결과가 저장될 이차원 배열)을 초기화시킨 후 시작한다. 프로세스가 도착하면 Queue에 넣고 state(flag)를 RUN 상태로 바꾼다. 시간이 1초씩 증가함에 따라 queue가 비어있지 않으면 FIFO_pop 함수를 호출하여 queue의 가장 앞(head)에 있는 프로세스의 service time의 1초를 수행한다. 이 과정을 반복하다가 해당 프로세스의 service time이 0이 되면 queue의 맨 앞을 플러시(processEnd 함수)하고 queue를 pop한다.

2-2. RR

RR은 도착한 순서대로 single queue에 삽입되고, 시간적 할당(Time slice)를 주어서 time slice만큼 프로세스를 수행하고 service time이 남아있다면 다시 queue에 들어가는 방식이다. queue와 array를 초기화시키고 프로세스의 arrival time에 따라 insertQueue 함수를 통해 queue에 삽입한다. 그 후에 queue에 프로세스가 하나라도 있다면 RR_pop을 호출한다. 또한 프로세스의 상태를 알기 위해 insert_flag를 사용한다. time slice가 지나고 service time이 남아 있는 경우 queue의 가장 뒤(tail)에 삽입한다.

RR_pop 함수에서는 프로세스의 service time이 0이 되면 queue의 인덱스를 초기화시키고 다음 프로세스를 가리킨다. 첫 번째 프로세스의 service time이 남아있는데 두 번째 프로세스가 들어오는 경우 새로운 프로세스를 먼저 수행시킨다. 타임 퀀텀을 1로 고정한다면 queue의 개념 없이도 1초마다의 스케줄링을 할 수 있었는데 유동적인 타임 퀀텀을 사용하기 위해 원형 큐의 개념으로 접근했다.

2-3. MLFQ

MLFQ 본연의 정책에 어긋나지 않으려 노력했다. queue의 개수에 따라 타임 퀀텀(2^i)을 자동으로 할당한다. 시간을 증가시키며 도착하는 프로세스를 queue의 Top에 삽입한다. 그리고 queue를 탐색하여 스케줄링 대상을 결정한다. MLFQ의 기본적인 정책은 우선 순위가 가장 높은 queue부터 아래로 내려가며 제일 먼저 탐색된 프로세스를 수행시키는 것이다. 프로세스가 스케줄링되면 수행 완료 후 다음 queue의 위치를 선택해야 하며, queue에 설정된 time slice를 모두 사용하면 lower priority queue에 삽입하고, 그렇지 않다면 priority는 그대로 유지한다. 그리고 lowest priority queue에 도달하면 해당 queue의 time slice만큼 Round Robin의 방식으로 수행하게 된다.

위에서 설명한 알고리즘이 기본이고, 다른 예외 처리를 많이 했다. 모든 queue에서 프로세스가 하나라면 큐의 time slice를 모두 사용해도 lower priority로 내리지 않는다. 또한, 수행한 프로세스가 다시 queue에 삽입됨과 동시에 새롭게 도착한 프로세스가 있다면 새로운 프로세스에 우선 순위를 주었다.

2-4. STRIDE

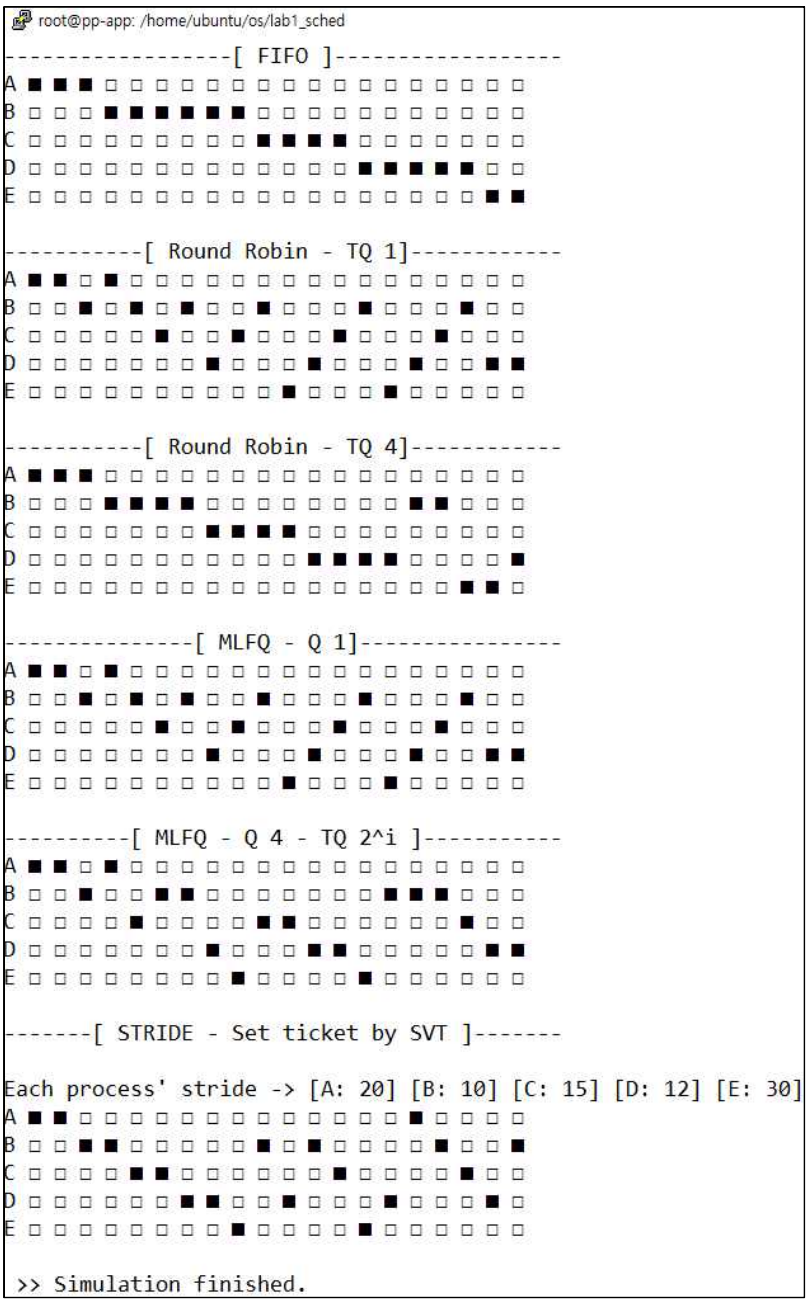
STRIDE 스케줄링은 프로세스마다 Ticket을 부여하여 Stride를 계산하고, time slice마다 가장 낮은 stride의 프로세스를 수행하고 해당 프로세스의 stride를 누적하는 방식이다. ticket은 프로세스의 service time에 따라 자동으로 할당하고, 각 ticket의 LCM(최소공배수)를 구하여 각 프로세스의 stride를 정한다. 1초마다 스케줄링 대상이 되는 프로세스(schCandidate) 중 수행할 프로세스(schTarget)를 결정하기 위해 각 프로세스의 stride를 탐색하고 가장 낮은 stride를 가진 프로세스를 수행한다. 수행한 프로세스의 service time이 0이 되면 스케줄링 후보에서 삭제한다. 이 과정을 반복하다가 time이 width(프로세스 service time의 합)와 같아지면 구현을 종료한다.

3. 시뮬레이션 결과

3-1. Lecture 3 PPT의 Sample로 실행

– input data

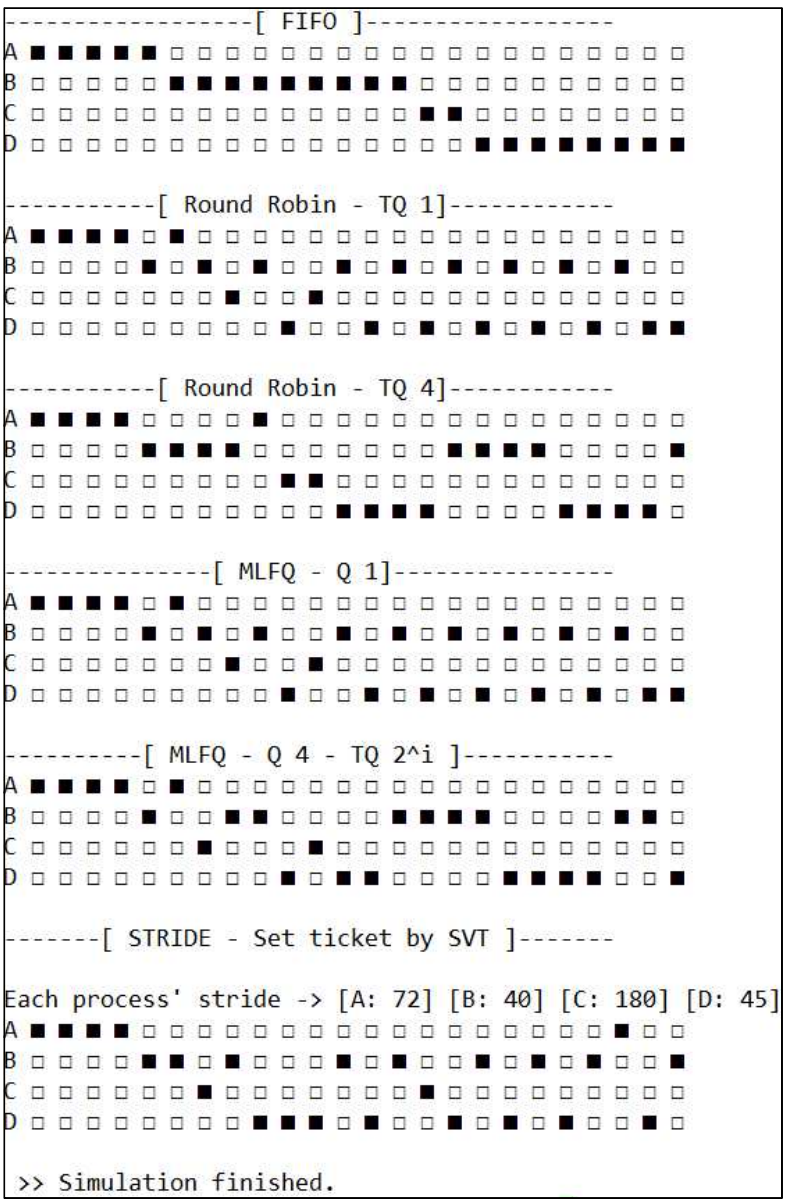
Process	Arrival Time	Service Time
A	0	3
B	2	6
C	4	4
D	6	5
E	8	2



3-2. 임의로 정한 데이터로 실행

- input data

Process	Arrival Time	Service Time
A	0	5
B	4	9
C	6	2
D	8	8



3-3. STRIDE 구현 확인을 위한 PPT의 Stride part sample

- input data

Process	Arrival Time	Service Time
A	0	10
B	0	5
C	0	25

(Ticket을 구하는 알고리즘을 Service Time의 LCM으로 계산)

Pass(A) (stride=100)	Pass(B) (stride=200)	Pass(C) (stride=40)	Who Runs?
0	0	0	A
100	0	0	B
100	200	0	C
100	200	40	C
100	200	80	C
100	200	120	A
200	200	120	C
200	200	160	C
200	200	200	...

```
-------[ STRIDE - Set ticket by SVT ]-------

Each process' stride -> [A: 5] [B: 10] [C: 2]
A ■ □ □ □ □ ■ □ □ ■ □ □ □ □ ■ □ □ ■ □ □ □ □ ■ □ □ ■ □ □ □ □ ■ □ □ ■ □ □ □ □ ■ □ □
B □ ■ □ □ □ □ □ □ □ ■ □ □ □ □ □ □ □ ■ □ □ □ □ □ □ □ ■ □ □ □ □ □ □ □ ■ □ □ □ □ □ □
C □ □ ■ ■ ■ □ ■ ■ □ □ ■ ■ ■ □ ■ ■ □ □ ■ ■ ■ □ ■ ■ □ □ ■ ■ ■ □ ■ ■ □ □ ■ ■ ■ □ ■ ■
```

- Scheduled sequence: A → B → C → C → C → A → C → C ⋯ **Success!**

+ Make 커맨드 사용 화면

```
root@pp-app: /home/ubuntu/os/lab1_sched
root@pp-app:/home/ubuntu/os# git clone https://github.com/zinirun/lab1_sched.git
Cloning into 'lab1_sched'...
remote: Enumerating objects: 65, done.
remote: Counting objects: 100% (65/65), done.
remote: Compressing objects: 100% (40/40), done.
remote: Total 65 (delta 29), reused 50 (delta 16), pack-reused 0
Unpacking objects: 100% (65/65), done.
root@pp-app:/home/ubuntu/os# cd lab1_sched
root@pp-app:/home/ubuntu/os/lab1_sched# ls
include  lab1_sched.c  lab1_sched_test.c  Makefile
root@pp-app:/home/ubuntu/os/lab1_sched# make
Compiling lab1 scheduler simulator lab1_sched.c ...
gcc -c -g -I/home/ubuntu/os/lab1_sched/include/ -o lab1_sched.o lab1_sched.c
Compiling lab1 scheduler simulator lab1_sched_test.c ...
gcc -c -g -I/home/ubuntu/os/lab1_sched/include/ -o lab1_sched_test.o lab1_sched_
test.c
gcc -o lab1_sched lab1_sched.o lab1_sched_test.o
root@pp-app:/home/ubuntu/os/lab1_sched# ./lab1_sched

----------------------------------------
--                2020                --
-- DKU OS LAB 1 - SCHEDULER SIMULATOR --
--             BY HJJ/SCW             --
----------------------------------------

- How many process (1 to 50) ->
```

4. Discussion

4-1. 허전진 Discussion (RR, Stride 구현)

구현을 시작하며

Round Robin과 Stride의 구현을 맡고 어떻게 하면 스케줄링 정책의 기본적인 정책을 지키며 코딩할 것인지 많이 고민했다. 평소 프로그램을 만들 때 sudo code(의사 코드)와 mock up(UI 디자인)을 종이에 그려보고 작업하는 편이라 이번에도 그렇게 했고, 실제로 많은 도움이 됐다.

어떻게 협업할까

팀 프로젝트는 앞으로도 많을 것이라 서로 공유하는 작업 공간을 만들고 싶었다. 또한, 이번 코로나의 여파도 있고, 각자의 스케줄도 있어 면대면으로 만나서 작업하는 데에 어려움이 있었다. 그래서 학교에서 지원받은 토스트 인스턴스를 사용하여 파트너에게 PEM key를 주고, 테스트 폴더와 버전 관리, 백업을 위한 repository를 생성하여 작업했다. 어느 정도 프로그램이 완성됐을 때 git에 커밋하고 CentOS, Ubuntu 인스턴스에서 정상 작동을 확인했다.

Round Robin 구현에 관하여

Round Robin을 구현할 때 처음에는 queue의 개념을 도입하지 않고 스케줄링 대상을 배열로 선언하고 service time이 0이 되면 배열에서 삭제하는 방식으로 구현했는데, 그렇게 되면 타임 퀀텀이 2 이상인 상황에서 새로운 프로세스가 수행 중 난입했을 때 먼저 수행되는 구현이 잘되지 않아서 single queue의 개념을 도입했다. insert flag를 선언하여 queue에 들어갔던 프로세스를 각각 표시했다. 누적 시간은 타임퀀텀에서 한번 돌 때마다 증가시키고, service time은 감소시켜 주었다. 누적 시간을 토대로 타임 퀀텀보다 작으면 결과가 저장되는 array에 1을 저장했다.

자료구조 시간에 배운 queue의 개념이 이렇게 도움이 될 줄 몰랐다. stack, queue의 개념을 처음부터 다시 공부했고, tail에 삽입할 때(push), head에서 삭제할 때(pop) queue의 객체를 어떻게 효율적으로 연결할지 고민했다.

Stride 구현에 관하여

Stride 스케줄링의 구현은 ticket, stride를 프로세스마다 정해주는 것이 핵심이었다. 원래는 사용자에게 각 프로세스의 ticket을 입력받아서 stride를 정했는데, 만약 이 stride 코드가 실제 스케줄러에 들어가 있다면 사용자가 모든 프로세스의 ticket을 정하는 것은 말이 안 되는 일이다. 스케줄러 자체에서 프로세스의 적절한 ticket을 어떻게 부여할지 생각하다가, service time에 따라 ticket을 부여하고 최소공배수를 구하여 적절한 stride를 할당했다. stride가 적절히 할당되었다면 time마다 lowest stride를 찾아 그 프로세스를 수행하고, 누적 stride(runStride)를 증가시키고 그것을 토대로 비교하고 수행시켰다.

코드를 모두 작성하고 어떤 데이터로 테스트할지 찾다가 lecture 3 ppt에 있는 샘플 데이터로 테스트하기로 했다. ppt에 있는 프로세스 3개의 ticket을 10으로 나눈 값을 service time으로 넣으니 성공적으로 작동됨을 확인했다.

더 해보고 싶은 것

기회가 된다면 스케줄러를 웹에서 만들어서 서비스해보고 싶다. 아마 다음 학기의 수강생이 될 후배 학우들에게도 많은 도움이 될 것이고, 결과물이 정말 괜찮다면 교수님이 만드신 lecture의 ppt에 reference에 기여할 수도 있다는 생각을 했다. c언어로 구현된 알고리즘을 nodejs 문법에 맞게 바꾸고, 프론트엔드에서 사용자가 프로세스 정보를 입력하면 서버에서 구현 결과를 클라이언트에서 볼 수 있게 해보고 싶다. 모든 과정(스케줄링 알고리즘)이 동기적으로 수행되어야 하기에 Promise 문법에 대한 공부를 좀 더 하고 나중에 만들어 볼 예정이다.

운영체제 첫 팀 프로젝트를 끝내며

시스템프로그래밍 강의의 과제 중 My Shell 프로그램 구현은 이번 스케줄러 구현에 비교하면 정말 아무것도 아니라는 것을 느꼈다. 첫 번째 과제를 우여곡절 끝에 끝내니 정말 뿌듯했다. 이번에 sudo code를 적기 위해 쓴 종이가 10장이 넘는다. 그만큼 스케줄링 알고리즘에 대한 완벽에 가까운 공부에 한 걸음 다가갔다고 믿는다.

4-2. 신창우 Discussion (FIFO, MLFQ 구현)

구현을 시작하며

FIFO, MLFQ를 구현을 맡았다. 원래의 개발 환경은 시스템프로그래밍 강의에서 사용하던 putty를 이용한 리눅스 환경이었다. 이번 운영체제 시간에서는 별도의 가상 머신에서 우분투를 사용하여 리눅스 환경에서 코딩을 하다가 팀원이 만들어 준 우분투 인스턴스를 통해 협업 개발하였다. 팀원과 프로그램을 설계할 때 단순히 프로세스 정보를 가지고 계산하여 결과만 출력하는 것이 아니라 실제 스케줄러가 가지고 있는 정책들을 최대한 구현하려고 노력하였다.

FIFO

FIFO 알고리즘은 까다로운 조건이 없고 각 프로세스의 도착 시간만 다루면 되는 스케줄링이라 금방 구현할 줄 알았었다. 처음에는 queue의 개념 없이 구현하려고 time-slice에 따라 배열을 선언하여 스케줄링하다가 빈 스케줄링 시간(time-slice에서 수행할 프로세스가 없는 상황)의 구현이 제대로 되지 않아서 근본적인 스케줄러 구현을 위해 single queue 개념을 도입하기로 하고, queue에 대한 개념을 다시 익혔다. queue에 대해 이해는 했지만, 막상 코드로 구현하기는 쉽지 않았다. 자료구조 수업자료를 다시 공부하면서 코드와 개념을 다시 보고, queue에 대한 구현도 가능해졌다. 어려운 스케줄링이 아니라서 single queue의 개념을 도입하고 금방 구현을 해냈다.

MLFQ

MLFQ 알고리즘은 multi queue의 개념을 도입한 후에도 해결해야 할 예외처리가 많았다. 우선, 모든 queue를 통틀어서 프로세스가 하나라면 queue의 time-slice를 만족하여도 다음 queue에 내려 보내지 않는다. 또, 프로세스가 수행되고 다시 ready 상태의 queue에 삽입 될 때 동시에 새로 도착하는 프로세스가 있다면 새로 들어오는 프로세스에게 우선순위를 주었는데, 우선순위를 생각하고 구현하는 것이 까다로웠다. 이번 스케줄러 프로그램에서는 I/O와 boost를 구현하지 못했지만 의도적으로 수행시간을 반납하여 현재 queue에 머무르게 되는 현상을 방지하기 위해 축적 시간을 통하여 time-slice를 비교하게끔 하였다.

FIFO를 구현하면서 생소했던 queue에 대해서 공부를 해보니 multi queue를 이용한 MLFQ를 구현하는데 어려움이 다소 적었다. 하지만 FIFO처럼 간단하지 않고 레벨(priority)에 따른 queue를 생각해서 구현해야 하니 너무 생각해야 될 것도 많고 복잡했다.

이번 과제를 진행하면서 가장 힘들었던 것은 예외처리이다. 분명히 Rule(정책)대로 설계하고 코딩을 했으니 될 것으로 생각했던 것들도 많은 오류, 오차를 내고는 했다. 그리고 이를 추적하면서 예외처리를 해주는 것에 시간이 많이 소모되었다. 가장 대표적으로 MLFQ에서 사용자가 queue를 하나만 쓰기로 하였을 때(single queue를 사용할 때) 스케줄링의 결과는 RR의 결과와 동일해야 하는데, 이를 예외 처리하는데도 많은 고민을 했다.

더 해보고 싶은 것

MLFQ를 구현하고 나니 multi queue에 대한 이해가 깊어져서 다른 것들도 구현하고 싶어졌다. 또한, periodic boost 정책을 적용하여 확장된 MLFQ를 만들어 보고 싶어졌다. 일정 period마다 priority를 top으로 올리면 될 일이라 그리 어려울 것 같지는 않다. 그리고 주기적인 I/O signal를 주어서 I/O period에 맞게 스케줄링되도록 구현하고 싶었지만, 너무 복잡하여 이번에는 구현하지 못하였다. 언젠가 한 번 해보는 것도 좋을 것 같다. 또한, 나만의 새로운 스케줄링 정책을 만들고 싶다. 여유가 생긴다면 MLFQ보다 더 효율적인 스케줄링 정책을 생각해서 구현해보고 싶다.