운영체제 2차 과제

# <가상 CPU & 스케줄링 - 구현과 분석>

2021320011 컴퓨터학과 유지연

제출일: 2025년 6월 10일 (화)

Free days 사용일수: 5일

## 1. 개발환경

- OS (호스트 운영체제): Window 10, 게스트 OS: Ububtu 18.04.2
- 가상화 소프트웨어: Virtual Box 7.0.20
- 커널 버전: Linux 4.20.11
- 개발도구: Visual Studio Code
- 컴파일러: gcc

## 2. CPU 스케줄링과 정책

CPU가 하나라면 한 시점에 동작시킬 수 있는 프로세스는 한 개뿐이다. 따라서 여러 개의 수행 가능한 상태의 프로세스들을 동작시킬 때는 각 프로세스에 CPU를 할당해주는 과정이 필요하고 그것을 CPU 스케줄링이라고 부른다. 어떤 프로세스에 얼만큼의 시간을 할당할 것인지, 또 언제 스케줄링을 결정할 것인지 등의 구체적인 작동 방식은 CPU 스케줄링 정책에 의해 결정된다.

CPU 스케줄링 정책에는 대표적으로 **FCFS, SJFS, SRTF, 우선순위, 라운드 로빈**이 있다. 스케줄링 종류는 크게 선점형과 비선점형으로 나눌 수 있다. 선점형 스케줄링은 현재 수행중인 프로세스가 있더라도 언제든 다른 프로세스가 CPU를 점령할 수 있는 정책이다. 반대로 비선점형 스케줄링은 현재 수행 중인 프로세스가 더 이상 CPU를 사용하지 않는 경우(입출력)에만 다른 프로세스가 CPU를 점령할 수 있는 정책이다.

### 2-1. FCFS (First Come First Served)

FCFS는 준비 큐에 도착하는 프로세스 순서대로 CPU를 할당해주는 방법으로 FIFO 큐를 이용하여 구현할 수 있다. 구현하기 쉽다는 장점이있지만 프로세스의 CPU 버스트를 고려하지 않으므로 큐에 CPU 버스트가 매우 긴 프로세스가 앞 쪽에 존재한다면 뒷 쪽의 다른 프로세스들의 대기 시간이 길어진다.

### 2-2. SJFS (Shortest Job First Scheduling)

SJFS는 준비 큐에 있는 프로세스 중 CPU 버스트가 적은 프로세스들부터 CPU 할당을 해주는 방법이다. 이는 평균 대기 시간이 길어진다는 FCFS의 단점을 보완할 수 있는 스케줄링 정책이다. 세부적으로는 선점형과 비선점형으로 나눌 수 있으며 선점형의 경우 타임퀀텀마다 CPU 버스트를 확인하고 그 시점에서 가장 작은 CPU 버스트를 가진 프로세스를 선택한다. 또한 입출력 작업이 끝나는 경우처럼 준비 큐의 구성 프로세스가 변경될 때마다 프로세스 선택 과정을 반복한다.

하지만 SJFS는 실제 구현은 불가능하다. 프로세스가 수행 전이라면 해당 프로세스의 CPU 버스트를 알 수 없기 때문이다.

### 2-3. Round Robin

Round Robin(이하 RR)은 설정된 타임퀀텀을 기반으로 각 프로세스가 CPU를 균등하게 할당받고, 타임퀀텀 내에 CPU 버스트가 끝나지 않은 경우 준비 큐의 맨 뒤에 해당 프로세스를 놓는 방법이다. 정해진 타임퀀텀이 끝나면 프로세스 수행이 완료되지 않았더라도 다음 프로세스가 CPU를 할당받기 때문에 FCFS의 단점이었던 평균 대기 시간이 길어진다는 부분을 보완할 수 있다. RR은 시분할 방식에서 사용되는 스케줄링으로 리눅스 CFS나 윈도우 커널에서도 사용되고 있다.

### 2-4. 우선순위

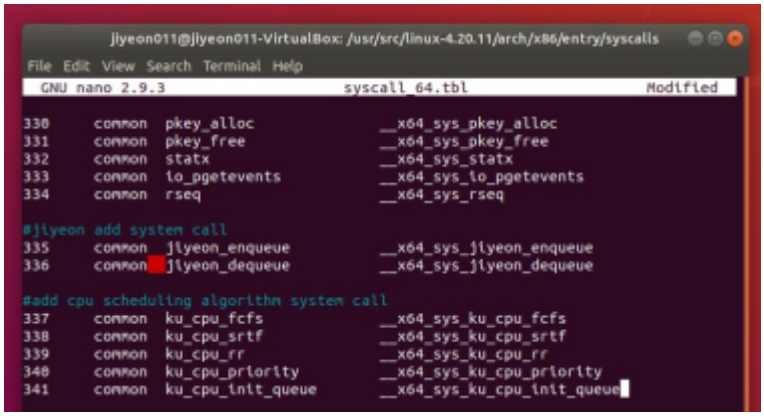
우선순위 스케줄링은 각 프로세스에 우선순위를 두어, 우선순위가 높은 순서대로 CPU를 할당해주는 방법이다. 프로세스에 우선순위를 설정하는 기준은 다양한데 대표적으로 짧은 CPU 버스트를 가진 프로세스나 입출력 작업이 적은 프로세스에 높은 우선순위를 매긴다.

이 때 낮은 우선순위를 받은 프로세스의 경우, 스케줄러가 프로세스를 선택하는 과정에서 선택을 받지 못해 대기 시간이 늘어나는 단점이 있을 수 있다. 이런 단점을 보완하고자 에이징이라는 기법이 사용된다. 에이징이랑 오랫동안 CPU를 할당받지 못한 프로세스의 우선순위를 높여주어 프로세스의 실행을 유도하는 방식이다.

우선순위 스케줄링도 세부적으로 선점형과 비선점형으로 나눌 수 있다. 선점형의 경우 선점형 SJFS와 유사하게 준비 큐의 구성 프로세스가 바뀔 때마다 우선순위를 확인하고 가장 높은 우선순위를 가진 프로세스가 CPU를 할당받는다. 비선점형의 경우에는 한 프로세스의 수행이 끝났을 때 우선순위를 판단하여 다음으로 실행될 프로세스를 선택한다.

## 3. 코드 설명

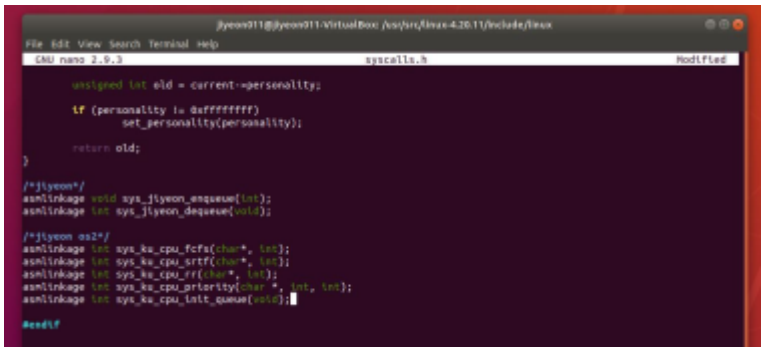
### 3-1. syscall_64.tbl




[그림 1] syscall_64.tbl 수정내용

syscall_64.tbl 에는 4가지 스케줄링 방식에 대한 시스템콜 함수와 준비 큐 구현을 위해 필요한 ku_cpu_init_queue 시스템콜 함수에 대해 337부터 341까지 순서대로 시스템콜 번호를 할당하였다.

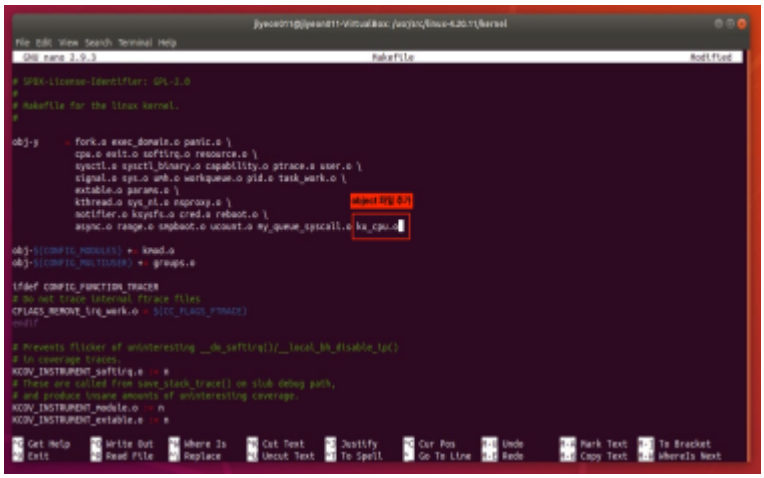
### 3-2. syscalls.h




[그림 2] syscalls.h 수정내용

새롭게 정의할 시스템 콜 함수의 prototype을 정의하는 파일이다. fcfs, srtf, round robin의 경우 char * 타입의 프로세스의 이름과 int 타입의 jobTime으로 구성된 2개의 인자를 받는다. priority의 경우 우선순위 값도 추가로 받아야 하므로 int 타입의 인자를 하나 더 받아 총 3개의 인자를 받는 함수이다. ku_cpu_init_queue의 경우 따로 인자를 받지 않는다.

### 3-3. Makefile



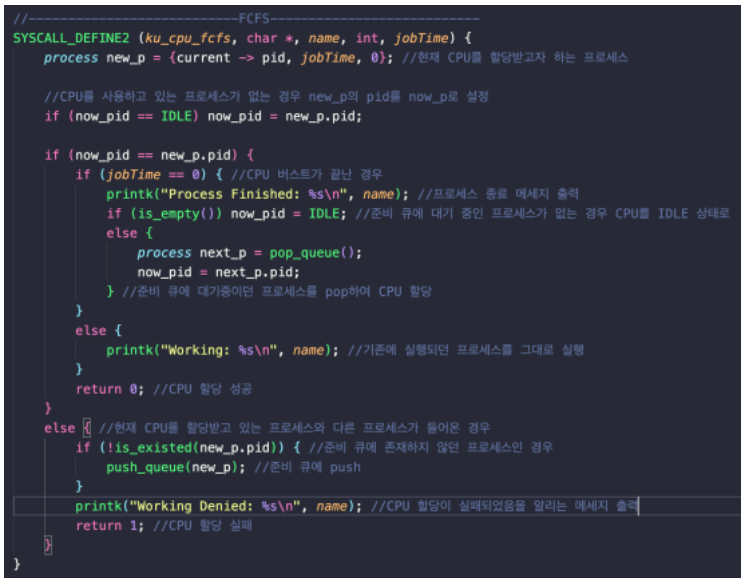
[그림 3] Makefile object 파일 수정내용

kernel make시에 포함되도록 커널 프로그램 c파일의 이름과 동일한 이름의 오브젝트 파일을 Makefile의 obj – y 부분에 추가한다.

### 3-4. ku_cpu.c

시스템콜 함수의 실질적인 구현이 포함되어 있는 파일이다. 4가지 스케줄링 정책에 공통적으로 사용되는 논리는 **CPU 할당을 성공했을 경우 0을 리턴**하고, CPU 할당에 **실패한 경우 1을 리턴**하는 것이다.

### 3-4-1. FCFS



```
//----------------------------FCFS----------------------------
SYSCALL_DEFINE2 (ku_cpu_fcfs, char *, name, int, jobTime) {
    process new_p = {current -> pid, jobTime, 0}; //현재 CPU를 할당받고자 하는 프로세스

    //CPU를 사용하고 있는 프로세스가 없는 경우 new_p의 pid를 now_p로 설정
    if (now_pid == IDLE) now_pid = new_p.pid;

    if (now_pid == new_p.pid) {
        if (jobTime == 0) { //CPU 버스트가 끝난 경우
            printk("Process Finished: %s\n", name); //프로세스 종료 메세지 출력
            if (is_empty()) now_pid = IDLE; //준비 큐에 대기 중인 프로세스가 없는 경우 CPU를 IDLE 상태로
            else {
                process next_p = pop_queue();
                now_pid = next_p.pid;
            } //준비 큐에 대기중이던 프로세스를 pop하여 CPU 할당
        }
        else {
            printk("Working: %s\n", name); //기존에 실행되던 프로세스를 그대로 실행
        }
        return 0; //CPU 할당 성공
    }
    else { //현재 CPU를 할당받고 있는 프로세스와 다른 프로세스가 들어온 경우
        if (!is_existed(new_p.pid)) { //준비 큐에 존재하지 않던 프로세스인 경우
            push_queue(new_p); //준비 큐에 push
        }
        printk("Working Denied: %s\n", name); //CPU 할당이 실패되었음을 알리는 메세지 출력
        return 1; //CPU 할당 실패
    }
}
```

[그림 4] FCFS 시스템콜 코드

FCFS 스케줄링 정책의 처리 과정을 정리하면 다음과 같다.

1) 현재 CPU를 점유하고 있는 프로세스가 있는지 확인한다.
2) CPU가 IDLE 상태인 경우, 현재 요청으로 들어온 프로세스에게 CPU를 할당한다.

3) CPU가 특정 프로세스에 의해 사용되고 있는 경우, 다음을 확인한다.

3-1) 현재 CPU를 점유하고 있는 프로세스가 요청을 보낸 프로세스와 같은 경우

jobTime을 통해 프로세스의 CPU 버스트가 남아있는지 확인한다.

- **jobTime = 0** (프로세스의 시행이 종료되는 경우)

준비 큐에 남아있는 프로세스가 있는지 확인 후 없으면 CPU를 IDLE 상태로 전환하고 작업을 멈춘다. 준비 큐에 CPU 할당을 기다리고 있는 프로세스가 있는 경우 큐의 가장 앞에 있는 프로세스를 pop하여 CPU를 할당한다.

- **jobTime != 0** (프로세스가 계속 시행되어야 하는 경우)

해당 프로세스가 계속 CPU를 사용하도록 한다.

3-2) 현재 CPU를 점유하고 있는 프로세스가 요청을 보낸 프로세스와 다른 경우

준비 큐에 해당 프로세스가 존재하는지 확인한 후, 존재하지 않는 경우만 준비 큐에 push 한다. 해당 프로세스의 요청은 거부되므로 1을 리턴한다.

### 3-4-2. SRTF

SRTF 스케줄링 정책의 처리 과정을 정리하면 다음과 같다.
(아래 코드 이전의 코드는 FCFS의 과정1 ~ 과정 3-1)과 동일하므로 설명을 생략한다.)

```
else { //현재 CPU를 할당받고 있는 프로세스와 다른 프로세스가 들어온 경우
    if (!is_existed(new_p.pid)) {
        //새로 들어오는 프로세스의 jobTime이 더 짧은 경우 -> 그 프로세스로 전환
        if (new_p.job < now_p.job) {
            push_srtf_queue(now_p); //준비 큐에 수행되던 작업을 push
            now_p = new_p;
            now_pid = new_p.pid;
            printk("Working: %s\n", name); //새로 요청받은 프로세스를 실행
            return 0; //CPU 할당 성공
        }
        else { //새로 들어오는 프로세스의 jobTime이 기존에 실행되던 프로세스보다 긴 경우 -> 프로세스 전환 발생 X
            push_srtf_queue(now_p); //요청된 프로세스를 준비 큐에 push
        }
    }
    printk("Working Denied: %s\n", name);
    return 1; //CPU 할당 실패
}
```

[그림 5] SRTF 시스템콜 코드 일부

현재 CPU를 점유하고 있는 프로세스가 요청을 보낸 프로세스와 다른 경우, 현재 CPU를 점유중인 프로세스의 jobTime과 새로 요청을 보낸 프로세스의 **jobTime을 비교하여 더 작은 프로세스에게 CPU를 할당**해준다. 이것이 SRTF의 핵심 작동 방식이다.

3-2) 현재 CPU를 점유하고 있는 프로세스가 요청을 보낸 프로세스와 다른 경우

- **기존 프로세스의 jobTime이 더 작은 경우**

  새로 요청받은 프로세스는 준비 큐에 push하고 1을 리턴한다. (CPU 할당 실패)

- **새로 들어온 프로세스의 jobTime이 더 작은 경우**

  기존 프로세스를 준비 큐에 push하고 새로운 프로세스에 CPU를 할당한다.
  이 경우 요청된 프로세스가 CPU 점령에 성공한 것이므로 0을 리턴한다.

SRTF의 경우 프로세스를 준비 큐에 pop/push하는 과정에서 추가적으로 고려해야 할 것이 있다. 기존 CPU를 사용하던 프로세스의 jobTime이 0이 되어 준비 큐에 있는 새로운 프로세스에 CPU를 할당하는 경우, **준비 큐에 있는 프로세스 중 jobTime이 가장 작은 프로세스가 선택**되어야 한다.

이를 위해 SRTF에서 아용하는 준비 큐의 경우 항상 **jobTime을 기준으로 오름차순 정렬**이 되어있을 수 있도록 코드를 구현하였다.

```
void push_srtf_queue(process p) {
    int i;
    waiting_queue[rear] = p;
    rear = (rear + 1) % MAX;

    //준비 큐를 오름차순 정렬
    for (i = rear - 1; i != front; i = (i - 1) % MAX) {
        if (waiting_queue[i].job < waiting_queue[(i-1) % MAX].job) {
            process temp = waiting_queue[i];
            waiting_queue[i] = waiting_queue[(i-1) % MAX];
            waiting_queue[(i-1) % MAX] = temp;
        }
    }
}
```

[그림 6] SRTF 방식에서 사용하는 push 함수

### 3-4-3. Round Robin

Round Robin은 타임퀀텀(timeslice)를 설정하고 모든 프로세스에 해당 시간만큼의 균등한 CPU 사용 시간을 주고, 주어진 시간 내에 CPU 버스트가 끝나지 않은 경우 (jobTime != 0) 준비 큐의 뒤쪽에 해당 프로세스를 push 하는 방식이다.

```
if (now_pid == new_p.pid) {
    if (jobTime == 0) { //CPU 버스트가 끝난 경우
        printk("Process Finished: %s\n", name);  //프로세스 종료 메세지 출력
        if (is_empty()) now_pid = IDLE; //준비 큐에 대기 중인 프로세스가 없는 경우 CPU를 IDLE 상태로
        else {
            process next_p = pop_queue();
            now_p = next_p;
            now_pid = next_p.pid;
        } //준비 큐에 대기중이던 프로세스를 pop하여 CPU 할당
        timeslice = 20; //타임퀀텀 초기화
        return 0;
    }
    else if (timeslice == 0) { //CPU 버스트는 남았지만 RR 정책에 의해 다른 프로세스로 전환되는 경우
        printk("Turn Over ----> %s\n", name);
        push_queue(new_p); //사용중이던 프로세스를 준비 큐에 push
        now_p = pop_queue(); //준비큐에서 새로운 프로세스 pop
        now_pid = now_p.pid;
        timeslice = 20; //timeslice 초기화
        return 1;
    }
    else { //기존 프로세스가 계속 CPU를 사용하는 경우
        printk("Working: %s\n", name);
        timeslice--;
        return 0;
    }
}
```

[그림 7] RR 시스템콜 함수 중 과정 3-1)에 해당하는 부분

Round Robin 스케줄링 정책의 처리 과정을 정리하면 다음과 같다.

1) 현재 CPU를 점유하고 있는 프로세스가 있는지 확인한다.
2) CPU가 IDLE 상태인 경우, 현재 요청으로 들어온 프로세스에게 CPU를 할당하고 timeslice를 20으로 초기화한다. (임의로 설정한 값)
3) CPU가 특정 프로세스에 의해 사용되고 있는 경우, 다음을 확인한다.

3-1) 현재 CPU를 점유하고 있는 프로세스가 요청을 보낸 프로세스와 같은 경우
jobTime을 통해 프로세스의 CPU 버스트가 남아있는지 확인한다.
- **jobTime = 0** (프로세스의 시행이 종료되는 경우)
준비 큐에 남아있는 프로세스가 있는지 확인 후 없으면 CPU를 IDLE 상태로 전환하고 작업을 멈춘다. 준비 큐에 CPU 할당을 기다리고 있는 프로세스가 있는 경우 큐의 가장 앞에 있는 프로세스를 pop하여 CPU를 할당한다.
- **timeslice = 0** (다른 프로세스로 전환해야 하는 경우)
기존 수행 중이던 프로세스를 준비 큐의 맨 뒤에 삽인한다.
준비 큐의 맨 앞에 있던 프로세스를 pop하고 CPU를 할당해준다.
요청된 프로세스가 CPU 할당에 실패한 경우이므로 1을 리턴해준다.
- **jobTime != 0 && timeslice != 0** (기존 프로세스를 계속 수행하는 경우)
해당 프로세스가 계속 CPU를 사용하도록 한다.

3-2) 현재 CPU를 점유하고 있는 프로세스가 요청을 보낸 프로세스와 다른 경우
준비 큐에 해당 프로세스가 존재하지 않는 경우만 준비 큐에 프로세스를 push 한다.
해당 프로세스의 요청은 거부되므로 1을 리턴한다

```
else {
    if (!is_existed(new_p.pid)) push_queue(new_p);
    printk("Working Denied: %s\n", name);
    return 1; //CPU 할당 실패
}
```

[그림 8] RR 시스템콜 함수 중 과정 3-2)에 해당하는 부분

### 3-4-3. Priority

Priority의 경우 SRTF의 구현방식과 대부분 동일하며, 차이점은 준비 큐를 정렬할 때 **우선순위를 기준으로 오름차순 정렬**하는 것이다. jobTime이 0이 되거나 새로 요청된 프로세스의 우선순위가 기존 실행되던 프로세스보다 높은 경우 (값이 작은 경우) 시행 프로세스 간의 전환이 발생한다.

```
void push_priority_queue(process p) {
    int i;
    waiting_queue[rear] = p;
    rear = (rear + 1) % MAX;

    //준비 큐를 우선순위 값을 기준으로 오른차순 정렬
    for (i = (rear - 1 + MAX) % MAX; i != front; i = (i - 1 + MAX) % MAX) {
        int prev = (i - 1 + MAX) % MAX;
        if (waiting_queue[i].priority < waiting_queue[prev].priority) {
            process temp = waiting_queue[i];
            waiting_queue[i] = waiting_queue[prev];
            waiting_queue[prev] = temp;
        } else {
            break;
        }
    }
}
```

[그림 9] Priority 방식에서 사용하는 push 함수

**3-4-5. 부가적인 함수들 (push, pop, is_empty 등)**
준비큐의 구현 방식으로는 배열을 선언한 후 큐의 front와 rear 값을 가리키는 index 원소를 두고 사용하였다. 따라서 큐의 맨 앞 원소를 **pop 해야하는 경우 waiting_queue[front]를 반환**하였고, 큐의 맨 뒤에 원소를 **push 해야하는 경우 waiting_queue[rear] 위치에 삽입**하였다.

추가적으로 SRTF, Priority의 경우 큐가 특정 값을 기준으로 오름차순 정렬이 되어야하기 때문에 push_srtf_queue, push_prioirty_queue를 별도로 사용하였으며, FCFS와 Round Robin의 경우 동일한 push_queue 함수를 사용하였다.

## 4. 실행 결과 및 분석

### 4-1. 실행 결과 (dmesg 로그)

```
jiyeon011@jiyeon011-VirtualBox:~/test$ ./run
jiyeon011@jiyeon011-VirtualBox:~/test$
Process A : I will use CPU by 7s.

Process B : I will use CPU by 5s.

Process C : I will use CPU by 3s.

Process A: Finish! My response time is 0s and My total wait time is 0s.
Process B: Finish! My response time is 6s and My total wait time is 6s.
Process C: Finish! My response time is 10s and My total wait time is 10s.
```

[그림 10] FCFS 실행결과

```
jiyeon011@jiyeon011-VirtualBox:~/test$ ./run
jiyeon011@jiyeon011-VirtualBox:~/test$
Process A : I will use CPU by 7s.

Process B : I will use CPU by 5s.

Process C : I will use CPU by 3s.

Process C: Finish! My response time is 0s and My total wait time is 0s.
Process B: Finish! My response time is 0s and My total wait time is 3s.
Process A: Finish! My response time is 0s and My total wait time is 8s. ^C
```

[그림 11] SRTF 실행결과

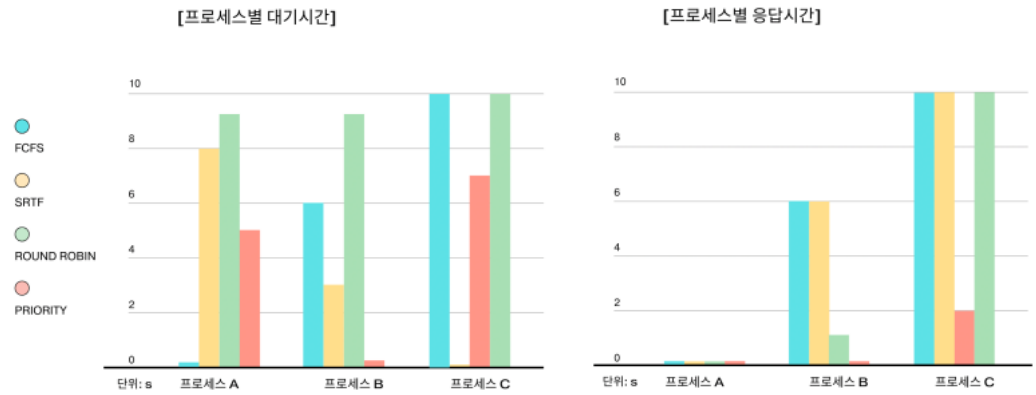
```
jiyeon011@jiyeon011-VirtualBox:~/test$ ./run
jiyeon011@jiyeon011-VirtualBox:~/test$
Process A : I will use CPU by 7s.

Process B : I will use CPU by 5s.

Process C : I will use CPU by 3s.

Process C: Finish! My response time is 2s and My total wait time is 7s.
Process B: Finish! My response time is 1s and My total wait time is 9s.
Process A: Finish! My response time is 0s and My total wait time is 9s. ^C
```

[그림 12] Round Robin 실행결과

```
jiyeon011@jiyeon011-VirtualBox:~/test$ ./run
jiyeon011@jiyeon011-VirtualBox:~/test$
Process A : I will use CPU by 7s.

Process B : I will use CPU by 5s.

Process C : I will use CPU by 3s.

Process B: Finish! My response time is 0s and My total wait time is 0s.
Process A: Finish! My response time is 0s and My total wait time is 5s.
Process C: Finish! My response time is 10s and My total wait time is 10s.
```

[그림 13] Priority 실행결과

### 4-1. 스케줄링 정책 별 대기시간 및 응답시간 비교




각 스케줄링 정책을 적용했을 때 프로세스 별 대기 시간과 응답시간은 위의 그래프와 같다.
평균 대기 시간은 SRTF 방식이 가장 짧고 Round Robin 방식이 가장 길었다. 평균 응답 시간의 경우 Round Robin 방식이 압도적으로 짧고 FCFS, SRTF 방식이 유사한 값으로 가장 길게 측정되었다.

| | FCFS | SJFS | Round Robin | 우선순위 |
|---|---|---|---|---|
| 동작 방식 | - 준비 큐에 들어오는 순서대로 CPU 할당 | - 준비 큐의 프로세스 중 남은 CPU 버스트가 작은 순서대로 CPU 할당 | - 타임퀀텀만큼 각 프로세스에 CPU를 할당<br>- 실행이 끝나지 않은 경우 준비 큐의 맨 뒤에 프로세스 삽입 | - 우선순위가 높은 순서대로 CPU 할당 |
| 장점 | - 구현이 쉬움 (FIFO 큐) | - 프로세스들의 평균 대기 시간이 적음<br>- 이론적으로 CPU 활용성이 가장 높음 | - 타임퀀텀에 따른 균등한 CPU 분배 가능<br>- 프로세스들의 평균 대기 시간 감소 | - 설계 기준에 맞춘 적절한 프로세스 선정 가능 |
| 단점 | - CPU 버스트가 큰 프로세스로 인한 평균 대기 시간 증가 | - 프로세스 실행 전 CPU 버스트를 알 수 없으므로 현실적으로 구현 불가 | - 프로세스간 문맥전환에 의한 오버헤드 증가 | - 우선순위가 낮은 프로세스가 오랫동안 실행되지 못함 |

위는 각 정책별 동작 방식 및 장단점을 정리한 표이다.

각 정책 별 특성을 고려해보았을 때, FCFS는 프로세스 간의 우선순위가 중요하지 않고 응답 시간 보다는 전체 처리량이 중요한 경우 사용될 수 있다. 프로세스간 전환 빈도가 낮기 때문에 전환에 사용되는 오버헤드가 적기 때문이다.

SRTF의 경우 프로세스들의 평균 대기 시간이 작다는 특성을 고려했을 때 짧은 작업이 많고 실시간으로 프로세스들이 수행되어야 하는 경우 가장 적합할 수 있다. Round Robin의 경우 모든 프로세스들이 균등하게 CPU를 사용할 수 있기 때문에 시분할 방식이나 응답 시간이 보장되어야 하는 경우 사용하기 적합하다. 하지만 타임퀀텀이 너무 작게 설정되면 문맥전환에 의한 오버헤드가 증가할 수 있고 반대로 타임퀀텀이 너무 길면 FCFS와 동일한 단점을 가질 수 있기에 적절한 타임퀀텀 설정이 중요하다.

마지막으로 Priority의 경우 프로세스간 우선순위가 확실한 작업을 시행하는 경우 가장 적합하다. 또한 설계자의 의도대로 우선순위를 설정할 수 있기 때문에 여러 상황에 유연하게 대처할 수 있는 스케줄링 방식이 될 수 있다.

## 5. 문제점과 해결방법

과제를 진행하면서 ku_cpu.c를 수정한 후 유저 프로그램을 실행시켰는데도 dmesg를 통해 커널로그를 확인해본 결과, 이전 커널 코드에 포함된 커널로그가 출력되고 있는 문제점을 발견하였다. 원인을 분석하면 다음과 같다.

/usr/src/linux-4.20.11/kernel 디렉토리에 위치한 ku_cpu.c를 수정한 이후 sudo make, sudo make install 명령어를 수행시킨다. 이 때 make 명령어란 수정한 커널 코드를 컴파일 하여 새로운 커널 이미지를 만드는 작업을 하는 것이다. 이 때 만들어진 이미지 파일은 디스크 위에만 존재할 뿐 실제 실행중인 시스템에는 적용되지 않는다.

따라서 새로운 커널을 프로그램에 적용하려면 반드시 해당 커널을 메모리에 올리는 작업이 필요하므로 **“reboot”** 명령어를 실행시켜야한다. reboot 명령어를 수행시킨 이후 다시 터미널에서 유저 프로그램을 실행시킨 후 dmesg를 확인해보니 새롭게 수정한 커널 코드가 적용된 것을 확인할 수 있었다.