

OS Project Report

-Create my own CPU Scheduler-

인하대학교

Operating System 2반

12181667 이지울

Contents

1. 프로젝트의 개요
2. Scheduler 설계
3. 프로세스 Input file 구성 및 결과 예측
4. Scheduler 코드 분석
5. Simulator를 통한 성능 평가 및 결과 분석
6. 개선할 점 및 느낀점

1. 프로젝트의 개요

1) CPU Scheduling이란?

: CPU가 1개인 single-core 환경에서는 CPU가 한 개의 프로세스만 실행할 수 있다. 따라서 여러 프로세스가 동시에 CPU를 차지 할 수 없으므로, 프로세스들 간에 일련의 순서를 정해서 스케줄링하여 혼란 없이 모든 프로세스가 수행될 수 있도록 해야 한다.

이때, CPU scheduler 에서는 메모리에서 ready state에 있는 process를 선택해서, 선택된 process를 CPU에게 할당하는 일을 한다.

CPU Scheduler가 다음 스케줄링될 프로세스를 결정하는 scheduling decision을 내리는데 적용할 수 있는 여러 알고리즘이 존재하고, 대표적인 알고리즘으로는 FCFS, Priority, SJF, RR 알고리즘 등이 있다. 스케줄링의 목적에 맞게 필요한 알고리즘을 선택하여 적용하게 된다.

2) 프로젝트의 목적은?

: 실제 컴퓨터에서 프로세스들을 스케줄링 하는 방법인 Multi-level queue를 구현해보고자 한다. 컴퓨터에서는 프로세스의 우선순위에 따라 n개의 큐에다가 각각 프로세스들을 할당하고, 큐의 우선순위가 높은 순서대로 각 큐의 알고리즘에 따라 CPU에 프로그램을 할당한다.

이번 Project에서는 이러한 Multi-level queue를 다섯 단계의 큐로 구성하고, 각 큐 안의 스케줄링 방식으로는 기존에 있는 SJF, RR, FCFS, SJF 등의 스케줄링 알고리즘들과 직접 구상해본 나만의 스케줄링 알고리즘을 채택하여 실제 C 코드로 구현해보려고 한다.

그 이후에는 적절한 process input file을 입력하고 컴파일 하여 각 큐가 scheduling 된 시뮬레이션 결과가 나오면, 예상했던 결과와 코드로 실행한 결과가 동일한 지 비교하여 c 코드가 잘 구현되었는지 확인하고, CPU Scheduling criteria를 직접 측정한 결과를 통해 각 알고리즘의 성능을 비교하여 각각의 알고리즘이 어떤 점에는 우수하고, 어떤 점에는 개선이 필요한지 분석해 보도록 할 것이다.

3) 사용 프로그램 및 환경

1) C 코드 에디터: Visual studio 2017

2) 컴파일 환경 : Kali Linux의 Terminal에서 gcc 명령어를 통해 컴파일

2. Scheduler 설계

: 우선 실제 컴퓨터에서 사용되는 스케줄링 방식인 Multi-level queue를 구현해볼 것이다.

Queue는 총 다섯 개로 구성하고, Queue간 Scheduling에는 실제로 우선순위가 높은 Task 부터 실행시켜주기 위해서 Priority Scheduling 방식을 채택했다.

=> Process file에서 Class number가 주어지는데, Class number가 작을수록 우선순위가 높은 큐가 될 것이다. 따라서 Queue1-> Queue2-> ... -> Queue5 순서로 스케줄링될 것이다.

: 각각 큐에는 서로 다른 다섯 개의 스케줄링 알고리즘을 구현하여 적용할 것이다.

1. Queue 1: FCFS Scheduling Algorithm

=> 프로세스가 도착하는 순서대로 스케줄링되어 실행될 것이다. 그렇지만 이 프로젝트는 모두 도착 시간이 같다고 가정하므로, 임의로 Process number 순서대로 실행한다.

2. Queue 2: Priority Scheduling Algorithm

=> 프로세스의 우선순위 순서대로 스케줄링되어 실행된다. 프로세스의 우선순위 순서대로 정렬하고, 만약에 프로세스의 우선순위가 같다면 Process number 순서가 빠른 프로세스를 앞으로 스케줄링한다. 그 후에 순서대로 프로세스를 실행한다. 빨리 처리해야 되는 프로세스 부터 스케줄링되는 장점이 있다.

3. Queue 3: Round-Robin Scheduling Algorithm (이하 R-R)

=> 프로세스가 큐에 들어온 순서대로 실행되며, 임의의 time quantum을 정해서 그만큼만 프로세스를 실행하고, 다음 순서로 넘어가서 모든 프로세스가 terminate될때까지 time quantum만큼 모든 프로세스를 실행한다. Response time 측면에서 상당히 우수할 것으로 예측된다.

4. Queue 4: Shortest-Job-First Scheduling Algorithm(이하 SJF)

=> 매번 스케줄링 할때마다 CPU Burst time이 가장 작은 순서대로 실행한다. 따라서 매번 스케줄링 할때마다 shortest remain time을 가진 프로세스를 골라서 실행한다. Average turnaround time 측면에서 상당히 우수할 것으로 예측된다.

5. Queue 5 : 나만의 스케줄러

=> **Absolute-Priority Scheduling (절대 우선순위 스케줄링) 이라고 명명하였다.**

: 위 스케줄링 알고리즘을 자세히 공부해보던 도중, 각 스케줄링 알고리즘 마다 장단점이 있다는 것을 깨달았다. 따라서 몇 개의 스케줄링 알고리즘을 결합하여 사용한다면 많은 scheduling algorithm criteria 측면에서 우수할 것이라고 생각하였다.

우선 Priority Scheduling에서 Priority와, SJF Scheduling에서 CPU-burst time 은 둘 다 작을수록 우선순위가 높고, 더 먼저 스케줄링 된다. 따라서 이 Priority와 burst time를 더하여 두 가지 측면을 모두 고려한 Absolute-Priority라는 절대 우선순위를 부여하여 순서대로 정렬한다. 예를 들어 Priority가 높고, CPU burst time도 짧다면 절대 우선순위도 높아지고, 그 프로세스가 가장 먼저 스케줄링 될 것이다. 그렇지만 CPU burst time이 짧아도, 우선순위가 낮다면 절대 우선순위가 낮아져 스케줄링이 늦게 될 수 있다. 그 후에 Response time 측면에서 좋은 Round-robin scheduling을 도입하여 Time quantum 만큼 모든 프로세스가

terminate 될 때까지 돌아가면서 실행해준다. 이 스케줄러를 도입한다면 priority와 cpu burst time을 모두 고려할 수 있게 되고, Response time면에서는 SJF보다 유리하고, Average turnaround time 측면에서는 Round-Robin보다 나을 것이라고 생각한다. 이후 실제 결과를 통해 성능을 확인해 볼 수 있을 것이다.

또, 사용자의 의도대로 burst time과 priority의 비율을 조정하여 절대 우선순위를 부여할 수도 있을 것이다. 현재는 임의로 5:5 비율로 더하여 Absolute priority를 책정하였지만, Priority에 더 비중을 두고 싶다면 Priority의 비율을 높인 절대 우선순위를 부여하고, Burst time에 더 비중을 두고 싶다면 Burst time의 비율을 높인 절대 우선순위를 부여할 수도 있다.

3. 프로세스 Input file 구성 & 결과 예측

-1) case 1: 성능 분석을 위한 input file

=> 성능 분석을 위한 input 파일을 구성할 때 고려한 점 두 가지가 있다.

- 1) 큐에 고루 들어가는 것을 확인하기 위해서 class number는 무작위로 섞는다.
- 2) arrival time이 같기 때문에 모든 프로세스가 입력된 뒤에 각 큐에 할당한다. 성능을 확인하기 위해서 각 큐에 들어가는 프로세스들의 priority와 burst time을 통일하고, 이 두 조건이 들어가는 순서를 동일하게 하였다. 그래야 각 알고리즘에 들어가는 프로세스들의 조건들은 동일한 상태로 순수한 알고리즘의 성능만을 확인할 수 있게 된다.

=> input file 의 순서는 **Class num, Process num, Burst time, Priority** 순이다.

Process input file 1	각 큐로 들어간 결과	
1 1 7 2	Queue 1	Queue 4
5 2 7 2	1 1 7 2	4 5 7 2
3 3 7 2	1 4 4 1	4 11 4 1
1 4 4 1	1 6 5 3	4 12 5 3
4 5 7 2	Queue 2	Queue 5
1 6 5 3	2 8 7 2	5 2 7 2
3 7 4 1	2 10 4 1	5 13 4 1
2 8 7 2	2 14 5 3	5 15 5 3
3 9 5 3	Queue 3	
2 10 4 1	3 3 7 2	
4 11 4 1	3 7 4 1	
4 12 5 3	3 9 5 3	
5 13 4 1		
2 14 5 3		
5 15 5 3		

=> 각 프로세스들의 burst time, priority 들의 조건 순서가 동일한 것을 확인 할 수 있다.

Input file 1 스케줄링 한 예상 결과

(해당 input file 1은 성능 분석 시뮬레이션을 위해 임의로 구성한 input file 이며, 실제로는 프로세스들의 priority와 cpu time은 모두 다르므로, input file 2에서 process의 우선순위와 burst time(cpu time)을 모두 랜덤하게 구성하여 해당 스케줄러가 맞게 코딩되었는지 재차 확인 해 볼 것이다.)

1. queue 1 (FCFS) : Arrival time이 모두 같으므로 process number대로 정렬하여 스케줄링
순서: Process 1-> Process 4-> Process 6

Process 1 (0~7)	Process 4 (7~11)	Process 6 (11~16)
--------------------	---------------------	----------------------

1 1 1 1 1 1 1
4 4 4 4
6 6 6 6 6

2. Queue 2 (Priority) : 프로세스 우선순위대로 정렬하여 스케줄링
순서 : Process 10(pri: 1) -> Process 8 (Pri: 2)-> Process 14 (Pri: 3)

Process 10 (0~4)	Process 4 (4~11)	Process 6 (11~16)
---------------------	---------------------	----------------------

10 10 10 10
8 8 8 8 8 8 8
14 14 14 14 14

3. Queue 3 (R-R) : queue에 들어온 순서대로 time quantum만큼 돌아가면서 실행한다.
그렇지만, arrival time이 같으므로 process number 순서대로 실행한다.

Quantum: 3 , 순서 : Process 3(스케줄링 후 remain_time: 4)-> Process 7 (rem_t: 1)->
Process 9 (rem_t: 2) -> Process 3 (rem_t :1) ->Process 7 (rem_t: 0)
-> Process 9 (rem_t:0) -> Process 3 (rem_t:0)

Process 3 (0~3)	Process 7 (3~6)	Process 9 (6~9)	Process 3 (9~12)	P7 (12~13)	P9 (13~15)	P3 (15~16)
--------------------	--------------------	--------------------	---------------------	---------------	---------------	---------------

3 3 3
7 7 7
9 9 9
3 3 3
7
9 9
3

4. Queue 4 (SJF)

: 스케줄링 할 때마다 CPU burst time이 가장 짧은 프로세스를 골라 스케줄링한다.

Process 11 (rem_t :4) -> Process 12 (rem_t : 5)-> Process 5(rem_t : 7)

Process 11 (0~4)	Process 12 (4~9)	Process 5 (9~16)
---------------------	---------------------	---------------------

11 11 11 11

12 12 12 12 12

5 5 5 5 5 5 5

5. Queue 5 (Absolute- Priority)

: Remain time과, Priority를 고려한 Absolute_priority 순서대로 정렬하여 스케줄링하고, Quantum 만큼 돌아가면서 프로세스가 수행된다. : Quantum=3

Process 13 (Abs_Pri: 5, 스케줄링 이후 remain_time: 1) -> Process 15 (Abs_Pri : 8, rem_t: 2) -> Process 2 (Abs_Pri: 9,rem_t: 4) -> Process 13 (rem_t:0) -> Process 15(rem_t: 0) -> Process 2(rem_t:1) -> Process 2 (rem_t:0)

Process 13 (0~3)	Process 15 (3~6)	Process 2 (6~9)	P13 (9~10)	P15 (10~12)	P2 (12~15)	P2 (15~16)
---------------------	---------------------	--------------------	---------------	----------------	---------------	---------------

13 13 13

15 15 15

2 2 2

13

15 15

2 2 2

2

-2) Case 2

: cpu time과 우선순위를 모두 랜덤하게 구성한 케이스이다. 한 번 더 input 케이스를 검사하여 본인의 스케줄러가 올바르게 구현되었는지 확인해 보려고 한다.

Process input file 2	각 큐로 들어간 결과
1 1 4 1	Queue 1 Queue 4
2 2 5 2	1 1 4 1 4 10 3 3
1 3 7 2	1 3 7 2 4 12 5 1
2 4 3 3	4 13 2 3
3 5 2 2	
2 6 4 1	Queue 2 Queue 5
3 7 7 2	2 2 5 2 5 8 3 3
5 8 3 3	2 4 3 3 5 11 4 3
3 9 6 3	2 6 4 1 5 14 6 2
4 10 3 3	
5 11 4 3	Queue 3
4 12 5 1	3 5 2 2
4 13 2 3	3 7 7 2
5 14 6 2	3 9 6 3

예상 결과:

1. Queue 1 (FCFS) : P1→ P3 1 1 1 1 3 3 3 3 3 3 3 2. Queue 2 (Priority) : P6(Pri: 1) → P2(2) → P4(3) 6 6 6 6 2 2 2 2 2 4 4 4 3. Queue 3 (R-R) : P5 → P7 → P9 5 5 7 7 7 9 9 9 7 7 7 9 9 9 7	4. Queue 4(SJF) : P13(cpu_t:2) → P10(3) → P12(5) 13 13 10 10 10 12 12 12 12 12 5. Queue 5 (Absolute-Priority) : P8(abs_pri: 6) → P11(7) → P14(8) 8 8 8 11 11 11 14 14 14 11 14 14 14
---	---

-3) 최종 결과 예상

: 큐 간 알고리즘에서는 Priority Scheduling 방식을 채택했으므로,

Queue 1-> Queue 2-> Queue 3-> Queue 4-> Queue 5 순서대로 스케줄링될 것이다.

따라서, Case 1과 Case 2의 최종 예상 결과는 다음과 같다.

Case 1	Case 2
1 1 1 1 1 1 1 4 4 4 4 6 6 6 6 6 10 10 10 10 8 8 8 8 8 8 14 14 14 14 14 3 3 3 7 7 7 9 9 9 3 3 3 7 9 9 3 11 11 11 11 12 12 12 12 12 5 5 5 5 5 5 13 13 13 15 15 15 2 2 2 13 15 15 2 2 2 2	1 1 1 1 3 3 3 3 3 3 3 6 6 6 6 2 2 2 2 2 4 4 4 5 5 7 7 7 9 9 9 7 7 7 9 9 9 7 13 13 10 10 10 12 12 12 12 12 8 8 8 11 11 11 14 14 14 11 14 14 14

=> 추후에 실제로 짜본 스케줄러 코드에 input file을 넣고, 컴파일 한 결과가 예측 결과와 맞는지 확인해볼 것이다. 또 Case 1에서는 어떤 스케줄링 알고리즘이 우수한지 성능을 비교해볼 수 있을 것이다.

4. Scheduler 코드 분석

-1) 전역 변수

1. Struct Process

각 큐들에 받아오는 프로세스는 프로세스 안의 속성들이 많이 있으므로, scheduling criteria에 따라 측정해야 할 변수들을 구조체 안의 멤버 변수로 설정하였다.

큐 클래스 번호, 프로세스 번호, cpu 동작 시간, 대기시간, 우선순위...를 멤버 변수로 설정하여 나중에 simulation 결과로 출력하여 알고리즘 간 성능을 비교할 것이다.

2. ready_queue[MAX], multilevel_queue[MAX]

: 우선 프로세스들이 대기 큐에 한꺼번에 들어온 이후에, 클래스 넘버에 따라 각 큐로 할당된다. 따라서 Ready_queue로 input file을 읽어온 후에, multilevel_queue로 할당해줄 것이다. 각 큐에서 처리할 수 있는 최대 프로세스 개수는 1000개로 임의로 설정했다.

3. sem_t semaphore

: 각 멀티레벨 큐에서 스케줄링을 처리할 때, 각 큐가 모두 스케줄링되면 다음 큐로 넘어가도록 동기화해주기 위해서 semaphore를 각 큐마다 하나씩 설정하였다.

```
typedef struct process {
    int class_num, pro_num, cpu_t, wait_t, pri, ta_t, rem_t, resp_t, abs_pri;
    int terminate_flag;
    //line 1: 큐 클래스 번호, 프로세스 번호, 동작 시간, 대기시간, 우선순위, 전체 동작 시간
    //남은 동작 시간(SJF,RR), 응답시간, 절대 우선순위
    //line 2
}process;

//전체 프로세스를 받아오는 큐
process ready_queue[MAX];
//전체 프로세스 개수
int idx = 0;

//multi level queue 5개
process multilevel_queue1[MAX];
process multilevel_queue2[MAX];
process multilevel_queue3[MAX];
process multilevel_queue4[MAX];
process multilevel_queue5[MAX];

//각 큐에 할당된 프로세스 개수
int c1 = 0;
int c2 = 0;
int c3 = 0;
int c4 = 0;
int c5 = 0;

sem_t semaphore1;
sem_t semaphore2;
sem_t semaphore3;
sem_t semaphore4;
sem_t semaphore5;
```

2. FCFS Scheduling

1) 먼저 도착한 순서대로 스케줄링하여 실행하는 프로그램인데, 일단 Arrival time이 모두 같다고 가정하였으므로, 우선 큐에 들어올 때 Process number순서대로 cpu burst time만큼 출력한다.

2) 해당 프로세스가 언제 처음 스케줄링되는지 response time을 체크하기 위해서 전체 process가 모두 스케줄링되기까지 시간을 더해가며 체크해준다.

3) 해당 프로세스가 terminate되면, terminate_flag를 true로 만들고, process 가 terminate되기까지 걸린 turnaround time이 cpu burst time과 대기 시간을 더한 시간이므로, ta_t 변수에 대입시켜준다.

```
void FCFS_sched(process *pro, int n) {  
    int processing_t = 0;  
    //프로시저 도착 순서가 다 같으므로 process number기준으로 큐에 들어온 순서대로 스케줄링한다.  
    for (int i = 0; i < n; i++) {  
        //응답시간을 체크하기 위해서 processing_time을 0초부터 계산한다.  
        pro[i].resp_t = processing_t;  
  
        for (int p = pro[i].cpu_t; p > 0; p--) {  
            //cpu time 만큼 실행된다.  
            processing_t++;  
            printf("%d ", pro[i].pro_num);  
            pro[i].rem_t--;  
        }  
        printf("\n");  
  
        for (int j = i+1; j < n; j++) {  
            //뒤에 남은 다른 프로세스들에 수행된 시간만큼 wait time을 더해준다.  
            pro[j].wait_t += pro[i].cpu_t;  
        }  
  
        //해당 프로세스가 실행된 total time은 대기 시간+ 실행된 시간이다.  
        pro[i].ta_t = pro[i].cpu_t + pro[i].wait_t;  
        pro[i].terminate_flag = true;  
    }  
}
```

-3) Priority Scheduling

: 우선순위 순서대로, 우선순위가 높은 프로세스 먼저 실행시켜주는 스케줄링 기법이다.

1) 우선 arrival time이 모두 0으로 같으므로, 해당 프로세스들은 큐에 모두 같은시간에 큐에 들어가있다. 따라서 큐에서 프로세스의 우선순위대로 정렬을 한다. 우선순위가 같다면 Process number가 빠른 순서대로 정렬한다.

2) 프로세스들을 순서대로 실행하고, 해당 프로세스가 언제 처음 스케줄링되는지 response time을 체크하기 위해서 전체 process가 모두 스케줄링되기까지 시간을 더해가며 체크한다.

3) 해당 프로세스가 terminate되면, terminate_flag를 true로 만들고, process 가 terminate되기까지 걸린 turnaround time이 cpu burst time과 대기 시간을 더한 시간이므로, ta_t 변수에 대입시켜준다. (이하 모든 스케줄링 에서 turnaround time을 측정하는 법이 같으므로 해당 설명은 생략) 또 이미 terminate된 프로세스와 현재 스케줄링 된 프로세스를 제외하고 남은 프로세스들에 현재 스케줄링된 시간 만큼을 wait time에 더해준다.

```
void priority_sched(process *pro, int n) {
    //모두 같은 시간에 프로세스가 도착했고, non-preemptive 하므로
    //priority 순으로 정렬하여 순서대로 실행한다.
    //priority가 같으면 process number순으로 정렬
    process temp;
    int processing_t = 0;

    for (int i = n - 1; i > 0; i--) {
        for (int j = 0; j < i; j++) {
            if (pro[j].pri > pro[j + 1].pri) {
                temp = pro[j + 1];
                pro[j + 1] = pro[j];
                pro[j] = temp;
            }

            else if (pro[j].pri == pro[j + 1].pri && pro[j].pro_num > pro[j + 1].pro_num) {
                temp = pro[j + 1];
                pro[j + 1] = pro[j];
                pro[j] = temp;
            }
        }
    }

    for (int i = 0; i < n; i++) {
        //우선순위 순서대로 정렬되었으므로, 순서대로 스케줄링하여 실행한다.
        pro[i].resp_t = processing_t;
        for (int p = pro[i].cpu_t; p > 0; p--) {
            printf("%d ", pro[i].pro_num);
            processing_t++;
        }
        printf("\n");

        for (int j = 0; j < n; j++) {
            if (pro[j].terminate_flag == false){
                if (j != i) {
                    pro[j].wait_t += pro[i].cpu_t;
                }
            }
        }
        pro[i].ta_t = pro[i].cpu_t + pro[i].wait_t;
        pro[i].terminate_flag = true;
    }
}
```

-4) RR Scheduling

: 프로세스 사이에 우선순위를 두지 않고, Round-robin 방식으로 큐안의 프로세스가 모두 terminate될때까지 순서대로 Time quantum만큼 프로세스가 돌아가면서 CPU를 선점한다.

1) Queue안의 프로세스가 모두 terminate될때까지 순서대로 큐 안의 프로세스들을 돌면서 스케줄링해야 하기 때문에, Terminate 된 프로세스 개수를 계산하는 terminate_pronum 변수 와, 모든 프로세스가 최초로 한번씩 스케줄링 되는 첫바퀴에만 응답 시간을 체크해야 하기 때문에 몇 바퀴 돌면서 스케줄링되었는지 체크하는 rr 변수를 설정한다.

2) 매 프로세스가 돌아가면서 CPU를 선점하기 때문에, 각 프로세스가 스케줄링 될 때마다 다음에 CPU를 차지할 때 CPU burst time이 얼마나 남았는지 remain time을 계산해 주어야 한다. 이때, Quantum만큼 스케줄링 해주면서 Remain_time을 감소시키면서 출력시켜주고, 만약 Quantum 이전에 remain time이 0이 된다면 바로 해당 프로세스를 terminate시키고 다음 프로세스를 스케줄링해주어야 한다.

+) 또, 프로세스가 몇초만큼 실행되었는지 체크하기 위해서 qtime이라는 변수를 통해 측정해줄 것이다. 프로세스가 Quantum만큼 실행되었거나, 먼저 remain time이 0이 되어서 Quantum만큼 실행되지 못했을 가능성도 있기 때문이다.

```
void RR_sched(process *pro, int n, int Quantum) {
    //프로세스들 사이에 우선순위를 두지 않고 시간단위로 돌아가면서 프로세스를 할당한다.
    int terminate_pronum = 0;
    int rr = 0;
    int processing_t = 0;

    //모든 프로세스가 terminate될때까지 round-robin 돌면서 스케줄링한다.
    while (terminate_pronum != n) {
        rr++;

        for (int i = 0; i < n; i++) {
            int qtime = 0;

            if (pro[i].terminate_flag == false) {
                //terminate 안된 프로세스만 스케줄링한다.
                if (rr == 1) {
                    //response_time은 맨 처음 프로세스가 스케줄링되었을때만 측정되어야 하므로,
                    //round-robin이 한바퀴 돌아왔을때만 측정한다.
                    pro[i].resp_t = processing_t;
                }

                for (int j = 0; j < Quantum; j++) {
                    if (pro[i].rem_t == 0) {
                        //quantum만큼 수행해주되, remain_time이 0이되면 바로 다음 프로세스 스케줄링한다.
                        break;
                    }
                    else {
                        printf("%d ", pro[i].pro_num);
                        processing_t++;
                        pro[i].rem_t = pro[i].rem_t - 1;
                        qtime++; //quantum 만큼 실행이 다 됐으면 qtime=Quantum일것이다.
                    }
                }

                //다음 프로세스가 스케줄링 되면 다음 줄에 출력해야 한다.
                printf("\n");
            }
        }
    }
}
```

3) 현재 스케줄링된 프로세스와 이미 terminate된 프로세스를 제외하고 기다리고 있는 프로세스들에게 현재 프로세스가 실행된 시간(qtime 변수) 만큼 다른 프로세스들의 대기 시간에 더해줘야 한다.

4) remain time이 0이 되었다면, 해당 프로세스가 terminate된 것이므로, terminate_flag와 turnaround time을 측정해주고, terminate_pronum을 증가시켜준다. 이렇게 모든 프로세스가 terminate될때까지 돌아가면서 스케줄링된다.

```
for (int p = 0; p < n; p++) {
    if (pro[p].terminate_flag != true) {
        if (p != i) {
            pro[p].wait_t += qtime;
            //이전 프로세스가 수행된 시간만큼 나머지 프로세스의 wait time에 더해준다.
        }
    }

    if (pro[i].rem_t == 0) {
        //remain_time이 0이 되어 terminate된 프로세스는
        //terminate_flag와 turnaround_time을 측정해준다.
        pro[i].terminate_flag = true;
        pro[i].ta_t = pro[i].cpu_t + pro[i].wait_t;
        terminate_pronum++;
    }
    else {
        //terminate 되었다면 다음 프로세스로 넘어가서 스케줄링한다.
        continue;
    }
}
```

-5) SJF Scheduling

: 스케줄링 할 때 마다 가장 짧은 CPU burst length를 갖는 프로세스에게 우선순위를 높게 준다. 다양한 방법으로 스케줄러 코드를 작성해보기 위해, 이번에는 정렬하여 사용하는 방식 대신, 매번 Scheduling decision을 내릴 때마다 가장 CPU burst time이 짧은 프로세스를 선택하여 스케줄링 하는 방식으로 코드를 만들어 보았다. 해당 프로젝트에서는 arrival time이 모두 같다는 전제를 갖고 있으므로, Non-preemptive 하게 매번 CPU burst time만 비교해서 가장 짧은 burst time을 갖고 있는 프로세스를 스케줄링 하였다.

1) 큐 안에 모든 프로세스가 terminate 될 때까지 큐안에 있는 프로세스를 스케줄링한다. 다음 프로세스를 결정해야 하는 순간마다 cpu burst time이 제일 짧은 프로세스를 골라서 스케줄링한다.

2) 가장 적은 프로세스의 index를 min_index라고 변수를 정하고, 가장 적은 cpu burst time을 찾기 위해서 min_index를 임의로 1000이라고 가정하고 (큐에 들어갈수 있는 최대 프로세스의 개수가 1000개이므로) 더 작은 cpu_burst time을 갖고있는 프로세스의

cpu_time과 min_time 변수를 swap해주고, min_index와 프로세스의 index를 swap 해가면서 가장 cpu burst time이 작은 프로세스를 찾는다.

```
void SJF_sched(process *pro, int n) {
    int terminate_pronum = 0;
    int processing_t = 0;

    //큐 안의 모든 프로세스가 terminate될때까지 스케줄링된다.
    while (terminate_pronum != n) {

        //다음 프로세스를 결정해야 하는 순간마다 cpu burst time이 가장 적은 프로세스를 골라서 스케줄링한다.
        //arrival time이 다 똑같으므로 non-preemptive하다.
        int min_time = 1000;
        int min_index = -1;

        for (int i = 0; i < n; i++) {
            if (pro[i].terminate_flag == false) {
                //이미 terminate된 프로세스는 스케줄링에서 제외한다.

                if (min_time > pro[i].cpu_t) {
                    //cpu burst time이 가장 적은 프로세스를 골라서 스케줄링한다.
                    min_time = pro[i].cpu_t;
                    min_index = i;
                }
                else if (min_time == pro[i].cpu_t && pro[min_index].pro_num > pro[i].pro_num) {
                    //cpu burst time이 같다면, process_number가 빠른 순서로 스케줄링한다.
                    min_time = pro[i].cpu_t;
                    min_index = i;
                }
            }
        }

        }
}
```

3) 해당 프로세스를 cpu time만큼 실행하여 출력한다. 다른 프로세스들에 현재 스케줄링 된 시간만큼 wait time을 더해주고, 스케줄링이 끝난 프로세스는 turnaround time을 계산하여 대입시켜주고, terminate process number를 증가시켜준다. 이렇게 모든 프로세스가 terminate될때까지 모든 프로세스를 비교해가며 가장 cpu burst time이 짧은 프로세스를 찾아 스케줄링 해준다.

```
//해당 프로세스를 cpu time만큼 실행한다
pro[min_index].resp_t = processing_t;
for (int i = 0; i < pro[min_index].cpu_t; i++) {
    processing_t++;
    printf("%d ", pro[min_index].pro_num);
}

//이미 terminate된 프로세스와, 수행중인 프로세스를 제외하고
//아직 수행되지 못한 프로세스들에 현재 수행중인 프로세스의 cpu time(수행시간)만큼 wait time을 더해준다.
for (int i = 0; i < n; i++) {
    if (pro[i].terminate_flag == false) {
        if (i != min_index) {
            pro[i].wait_t += pro[min_index].cpu_t;
        }
    }
}

//terminate된 프로세스는 프로세싱하는데 걸린 시간과, terminate flag를 true로 바꿔준다.
pro[min_index].terminate_flag = true;
pro[min_index].ta_t = pro[min_index].wait_t + pro[min_index].cpu_t;
terminate_pronum++;
printf("\n");
}
```

-6) Absolute Priority Scheduling (나만의 스케줄러)

: CPU burst time과, Priority 모두 숫자가 작을수록, SJF과 Priority 스케줄링에서 우선순위가 높다는 것을 발견하고, 두 개를 더하여, 우선순위와 burst time을 모두 고려한 절대 우선순위 (abs_pri :Absolute priority)를 부여한다. 예를 들어, CPU burst time이 작아도, priority가 높으면 먼저 스케줄링될 수 있다. 급한 프로세스를 먼저 처리하기 위해서 만약 absolute priority가 같다면, 우선순위가 높은 프로세스를 먼저 스케줄링한다.

1) 프로세스가 모두 큐에 들어왔다면, 절대 우선순위 순서대로 정렬해준다. 절대 우선순위가 같다면, 우선순위 가 높은 프로세스를 먼저 정렬해준다.

```
//나만의 스케줄러 하나 짜기
void absolute_priority_sched(process *pro,int n,int Quantum) {
    process temp;
    for (int i = n - 1; i > 0; i--) {
        for (int j = 0; j < i; j++) {
            if (pro[j].abs_pri > pro[j + 1].abs_pri) {
                temp = pro[j + 1];
                pro[j + 1] = pro[j];
                pro[j] = temp;
            }
            else if (pro[j].abs_pri == pro[j + 1].abs_pri && pro[j].pri > pro[j + 1].pri) {
                //절대 우선순위가 같으면 우선순위가 높은 프로세스 먼저 실행한다.
                temp = pro[j + 1];
                pro[j + 1] = pro[j];
                pro[j] = temp;
            }
        }
    }
}
```

2) 다음부터는 Round-robin 방식으로 이미 정렬된 프로세스들을 Time-quantum만큼 돌아가면서 실행한다. 위의 Round-robin scheduler와 방식은 동일하다. 모든 프로세스가 Terminate될 때까지 Time quantum만큼 돌아가면서 프로세스를 실행하고, 큐에 있는 프로세스들이 최초로 한번씩 스케줄링 되었을 때만 response_time을 측정한다.

```
//arrival time이 모두 같으므로,절대 우선순위 순서로 프로세스를 정렬하였다.
//해당 큐의 순서대로 RR scheduling을 진행한다.
int terminate_pronum = 0;
int rr = 0;
int processing_t = 0;

while (terminate_pronum != n) {
    rr++;
    for (int i = 0; i < Quantum; i++) {
        //프로세스가 퀀텀만큼만 동작되어야 한다.
        int qtime = 0;
        if (pro[i].terminate_flag == false) {
            if (rr == 1) {
                // round-robin방식으로 모든 프로세스를 최초로 한번씩만 스케줄링 했을때만 응답 시간을 측정.
                pro[i].resp_t = processing_t;
            }
        }
    }
}
```


3) 이미 terminate된 프로세스를 제외하고, 프로세스를 선택했다면, Time quantum만큼 돌아가면서 실행을 해준다. 이때, quantum보다 적게 실행되고 terminate될수 있으므로, 해당 스케줄링되었을 때 실행된 시간을 qtime으로 측정해준다. quantum만큼 모두 실행이 되었다면, qtime이 quantum과 동일할 것이다.

4) 현재 스케줄링 된 프로세스와 이미 terminate 된 프로세스를 제외한 나머지 프로세스의 wait time 변수에 실행된 시간의 변수인 qtime만큼 더해주고, 해당 스케줄링된 변수의 remain_time이 0이라면, 해당 프로세스를 terminate시키고, turnaround time을 측정해주고, 다음 프로세스를 스케줄링해준다.

```

for (int j = 0; j < Quantum; j++) {
    //quantum만큼 수행해주되, remain_time이 0이되면 바로 terminate 안된 다음 프로세스 스케줄링한다.
    if (pro[i].rem_t == 0) {
        break;
    }
    else {
        printf("%d ", pro[i].pro_num);
        processing_t++;
        pro[i].rem_t = pro[i].rem_t - 1;
        qtime++; //quantum 만큼 실행이 다 됐으면 qtime=Q일것이다.
    }
}

printf("\n");

for (int p = 0; p < n; p++) {
    if (pro[p].terminate_flag != true) {
        if (p != i) {
            pro[p].wait_t += qtime;
            //이전 프로세스가 수행된 시간만큼 나머지 프로세스의 wait time에 더해준다.
        }
    }
}

if (pro[i].rem_t == 0) {
    pro[i].terminate_flag = true;
    pro[i].ta_t = pro[i].cpu_t + pro[i].wait_t;
    terminate_pronum++;
}
else {
    continue;
}
}

```

=> Absolute priority 방식에서의 프로세스는 CPU burst time도 짧고, 우선순위가 높은 프로세스가 가장 먼저 스케줄링된다. 따라서 Average turnaround time에서 기존의 R-R 알고리즘 보다 유리하고, Round-robin 방식으로 돌아가면서 프로세스가 스케줄링되므로, SJF 알고리즘 보다 Response time 측면에서도 우수할 것이라고 예측이 된다. 자세한 사항은 Simulation 부분에서 결과를 분석해 보도록 할 것이다.

또, 우선 임의로 5:5 비율로 더하여 절대 우선순위를 책정하였지만, CPU burst time에 우선순위를 높게 둔다면 burst time의 비율을 높이고, Priority에 우선순위를 높게 둔다면 Priority의 비율을 높여서 사용자의 의도대로 비율을 조정하여서 Absolute Priority를 부여할 수도 있을 것이다. (ex- Absolute_Priority= 0.3 * burst_time+ 0.7 * Priority)

-7) Simulation Result Print

: 각 큐가 스케줄링된 결과를 측정하여 print하는 함수이다.

response time, waiting time, turnaround time, average turnaround time, CPU utilization rate 등의 Scheduling criteria를 한눈에 볼수 있게 table형식으로 출력하여 바로 알고리즘 간의 성능을 비교해 볼 수 있도록 하였다.

=> 각 큐와, 스케줄링된 방식, 큐에 있는 프로세스 개수를 매개변수로 받아와 출력해보면, 한눈에 어떤 알고리즘이 어느 방면에서 우수한지 확인해 볼 수 있다.

```
void simulation_print(process *pro, int n, int schednum) {
    //각 스케줄링 알고리즘들의 시간을 비교하기 위해서, 시뮬레이션을 돌려
    //가장 효율적인 스케줄링 알고리즘이 무엇인지 확인할 수 있다.
    float tat = 0;
    float totalwait = 0;
    float totalwork = 0;
    float cpuutil = 0;

    printf("\n");
    switch (schednum) {
        case 1:
            printf("First come first Serve (FCFS)\n=====FCFS=====\n");
            break;
        case 2:
            printf("Priority \n=====Priority=====\n");
            break;
        case 3:
            printf("SJF(Shortest-Job-First) \n=====SJF=====\n");
            break;
        case 4:
            printf("RR(Round-Robin) \n=====RR=====\n");
            break;
        case 5:
            printf("Absoulte-Priority \n=====Absoulte_Priority=====\n");
            break;
    }
}
```

```
for (int p = 0; p < n; p++) {
    tat = tat + pro[p].ta_t;
    totalwait += pro[p].wait_t;
    totalwork += totalwork + pro[p].cpu_t;
    printf("%d\t%d\t%d\t%d\t%d\t%d\t%d\t%d\n",
        pro[p].class_num, pro[p].pro_num, pro[p].cpu_t, pro[p].pri, pro[p].wait_t, pro[p].resp_t, pro[p].abs_pri, pro[p].ta_t);
}

cpuutil = (totalwork / (totalwork + totalwait)) * 100;

printf("total waiting time:%2f\n", totalwait);
printf("average turnaround time: %.2f\n", tat / n);
printf("CPU Utilization rate: %.2f\n", cpuutil);
printf("=====\n");
}
```

-8). Main 함수

Main 함수에서는 input file에서 전체 프로세스를 받아와 입력하고, 각 프로세스를 큐에 할당할 main thread를 만들어 실행하는 작업을 하였다.

1) input file을 받아오고, 각 input file에서 프로세스의 속성을 받아와 프로세스 구조체의 멤버 변수들에 저장해준다. 여기서 프로세스의 terminate_flag를 false로 바꿔주고, remain time, response time을 0으로 설정하고, absolute_priority를 계산하여 넣어준다. (absolute priority의 계산식은 위에서 언급했듯 비율을 사용자의 의도대로 바꿔도 무관하다.) 그리고 모든 프로세스의 입력이 끝났다고 출력되면, 모든 프로세스가 잘 입력되었는지 class number, process number, CPU burst time, Priority, Remain time등의 리스트를 한번 출력해준다.

2) 그 뒤에 큐 간 스케줄링을 담당할 메인스레드 mainthread를 생성하고, 해당 스레드 함수를 넣어준다. 해당 스레드가 실행될 동안 main 함수는 동기화를 위해 join 함수로 잠시 wait하게 된다. (multiqueue_sched는 바로 뒤에 언급할 예정.) mainthread가 종료되면, "fin.." 문장을 출력하여 모든 출력이 끝났음을 알린다.

```
int main() {
    FILE *fp;
    //알고리즘 성능 분석용 케이스 1
    //fp= fopen("process.txt", "r");

    //랜덤으로 cpu time, priority 할당한 프로세스 케이스 2
    fp = fopen("process2.txt", "r");
    printf("file input start..\n");

    while (!feof(fp)) {
        fscanf(fp, "%d", &ready_queue[idx].class_num);
        fscanf(fp, "%d", &ready_queue[idx].pro_num);
        fscanf(fp, "%d", &ready_queue[idx].cpu_t);
        fscanf(fp, "%d", &ready_queue[idx].pri);
        ready_queue[idx].rem_t = ready_queue[idx].cpu_t;
        ready_queue[idx].terminate_flag = false;
        ready_queue[idx].resp_t = 0;
        ready_queue[idx].abs_pri = (ready_queue[idx].pri) + (ready_queue[idx].cpu_t);

        idx++;
    }
    printf("file input finish...\n\n");

    printf("-----Process List-----\n");
    printf("C#    P#    BT    Pri    RT\n");

    for (int p = 0; p < idx; p++) {
        //모든 프로세스가 제대로 입력되었는지 체크하기 위해서 프로세스 목록을 한번 출력해준다.
        printf("%d    %d    %d    %d    %d\n",
            ready_queue[p].class_num, ready_queue[p].pro_num, ready_queue[p].cpu_t, ready_queue[p].pri, ready_queue[p].rem_t);
    }

    printf("\n\n");
    //스레드를 생성한다.
    //큐간의 스케줄링 담당하는 main thread를 만들어 실행시켜준다.
    pthread_t mainthread;
    pthread_create(&mainthread, NULL, multiqueue_sched, (void *)idx);
    pthread_join(mainthread, NULL);

    print("fin.....");
}
```

-9) Multiqueue_sched 함수

: main 함수에서 큐간 스케줄링을 담당하는 main thread가 생성되었고, 해당 mainthread에서는 multiqueue_sched 함수를 통해, input file에서 들어온 프로세스들을 모두 각 큐에 할당하고, 큐 간 스케줄링 방식으로 채택한 Priority 방식에 따라서 다섯 개의 큐들을 순서대로 실행시킨다. 이때, 확실히 앞 큐의 스케줄링이 끝나야 다음 큐로 넘어가기 위해서 semaphore를 이용해 동기화하여 이미 한 프로세스가 실행중일 때, 다음 프로세스가 실행되지 못하도록 lock을 걸어두었다.

1) 우선 모든 프로세스의 개수를 매개변수 param으로 받아와 int 방식으로 casting하고, 각 process의 class number에 따라 각 multilevel_queue에 할당한다. 또, 프로세스 number 순서대로 큐에 할당하기 때문에, 큐에서는 이미 process number 순서대로 정렬이 되어있다. arrival time이 같기 때문에 해당 스케줄러에서는 process number 순서대로 각 큐에 프로세스들을 할당해주는 채택하였다.

```
void *multiqueue_sched(void *param) {
    //class number대로 multilevel queue로 각 프로세스 할당하고
    //class number 순서에 따라 priority scheduling하여 순서대로 실행함.

    int num = (int)param;
    printf("multi-level queue scheduling in process...");

    //같은 시간에 도착한 프로세스들을 class number에 따라 각 큐에 할당한다.
    //프로세스 number 순서대로 큐에 들어가게 되어 있다.
    for (int i = 0; i < num; i++) {
        int k = ready_queue[i].class_num;
        if (k == 1) {
            multilevel_queue1[c1] = ready_queue[i];
            c1++;
        }

        else if (k == 2) {
            multilevel_queue2[c2] = ready_queue[i];
            c2++;
        }

        else if (k == 3) {
            multilevel_queue3[c3] = ready_queue[i];
            c3++;
        }

        else if (k == 4) {
            multilevel_queue4[c4] = ready_queue[i];
            c4++;
        }

        else if (k == 5) {
            multilevel_queue5[c5] = ready_queue[i];
            c5++;
        }
    }
}
```

2) 또, 해당 스케줄러에서는 각 큐의 스케줄링 또한 각각 스레드로 처리하였는데, 이때 모든 멀티 스레드가 공유자원에 한꺼번에 접근해서는 안 된다. 각 스레드가 priority scheduling 방식에 따라 순서대로 실행되어야 하고, 우선순위 높은 큐의 스케줄링이 모두 끝나야 다음 큐로 넘어갈 수 있기 때문에 반드시 스레드 간에 lock을 걸어주어야 한다. 이때, 스레드들은 semaphore값이 1이면 sem_wait 함수를 통해 1 감소시키면서 공유 자원에 접근할 수 있고, 0보다 작거나 같으면 해당 자원에 접근 할 수 없게 되면서 대기 상태에 빠지게 된다.

따라서, sem_init 함수를 통해 위에서 전역변수로 설정한 semaphore 5개를 초기화 시켜주었다. 가장 먼저 실행될 t[0] thread의 semaphore1은 우선 1로 초기화하여 바로 스레드가 실행을 시작하도록 하였고, 그 다음 스레드부터는 모든 스레드가 0으로 초기화되어있어 대기상태로 있다가, 바로 이전 스레드가 다음 스레드의 semaphore를 sem_post함수를 통해 1로 증가시키면 다음 스레드가 실행되게끔 0으로 초기화해주었다.

3) 그 후로, 각 큐들은 스레드에서 따로 처리하기 위해서 pthread t[5]의 스레드 배열을 선언해주었고, pthread_create() 함수를 통해 각 스레드 변수의 주소, 큐 스케줄러 함수, 각 큐의 프로세스 개수를 인자로 넘겨주었다. 모든 스레드가 종료되면 mainthread함수가 마저 실행되게끔 pthread_join함수를 통해 동기화를 시켜주었다.

```
//class number를 priority로 하여 수행하지만, arrival time이 다 같으므로 큐 순서대로 실행된다.
sem_init(&semaphore1, 0, 1);
sem_init(&semaphore2, 0, 0);
sem_init(&semaphore3, 0, 0);
sem_init(&semaphore4, 0, 0);
sem_init(&semaphore5, 0, 0);

pthread_t t[5];
pthread_create(&t[0], NULL, queue1_sched, (void*)c1);
pthread_create(&t[1], NULL, queue2_sched, (void*)c2);
pthread_create(&t[2], NULL, queue3_sched, (void*)c3);
pthread_create(&t[3], NULL, queue4_sched, (void*)c4);
pthread_create(&t[4], NULL, queue5_sched, (void*)c5);

pthread_join(t[0], NULL);
pthread_join(t[1], NULL);
pthread_join(t[2], NULL);
pthread_join(t[3], NULL);
pthread_join(t[4], NULL);
```

4) 각 큐의 스레드들이 모두 종료되었으면, semaphore를 모두 파괴하고, 모든 멀티레벨 큐들의 simulation 결과를 모두 출력해준 뒤, main thread도 종료시켜준다.

```
// 모든 스레드가 종료되었음을 알리는 문장, 그 후 semaphore와 관련 리소스를 모두 파괴한다.
printf("Every multilevel-queue scheduling is terminated...\n");
sem_destroy(&semaphore1);
sem_destroy(&semaphore2);
sem_destroy(&semaphore3);
sem_destroy(&semaphore4);
sem_destroy(&semaphore5);

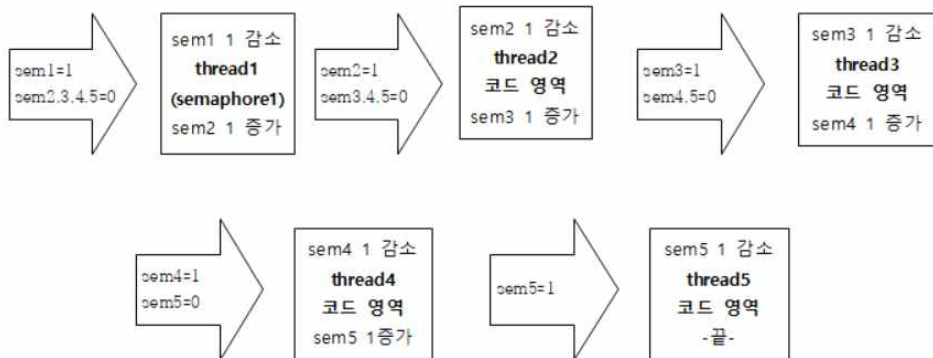
printf("Algorithm simulation result print...\n");
//큐 들 사이 알고리즘 성능을 비교하기 위해서 시뮬레이션 결과 출력함.
simulation_print(multilevel_queue1, c1, FCFS);
simulation_print(multilevel_queue2, c2, Priority);
simulation_print(multilevel_queue3, c3, RR);
simulation_print(multilevel_queue4, c4, SJF);
simulation_print(multilevel_queue5, c4, AP);

pthread_exit(0);
```


-10) 각 큐 간 스케줄링

: 위 함수의 설명 에서 언급했듯, 각 스레드들은 이전 스레드가 실행되기 전까지는 대기 상태에 빠져 있어야 한다. 해당 세마포어가 0이면 공유자원에 접근할 수 없고, 1이면 공유자원에 접근할 수 있으므로, 바로 실행되어야 하는 1번 큐의 스레드를 제외하고는, 모두 semaphore값을 0으로 초기화해서 대기 상태로 만들어주었다.

<그림으로 보는 해당 코드의 원리>



1번 스레드는 semaphore1를 바로 sem_wait함수를 통해 1 감소시키면서 다음 코드 영역을 실행한다. 1번 큐의 코드 영역을 모두 실행하면, 위 그림에서 확인할 수 있듯 2번 큐의 semaphore2를 sem_post 함수를 통해 1 증가시켜서 대기 상태에서 깨운다. 2번 큐도 역시 sem_wait 함수를 통해 semaphore2를 1에서 0으로 1 감소시키면서 다음 코드 영역을 진행하고, 바로 다음 스레드인 3번 큐 스레드의 semaphore3을 또 대기 상태에서 깨운다. 이렇게 모든 스레드가 종료될 때 까지 각 스레드를 atomic하게 실행한 후에 다음 스레드가 실행되도록 멀티스레드를 동기화 할 수 있게 된다.

1) 우선 가장 먼저 실행되는 1번 스레드는 이미 1로 초기화되어있는 semaphore1을 sem_wait함수를 통해 1 감소시키고, 다음 코드 영역을 실행하도록 코드를 작성했다. 각 스레드들이 순서대로 scheduling되는지 확인하기 위해 추가한 "nth queue scheduling in process.." 라는 문장을 출력한 다음, param인자로 해당 큐에 할당된 프로세스의 개수를 받아와 해당 큐의 알고리즘인 FCFS 스케줄링을 실행하고, 다음 스레드의 semaphore를 0에서 1 증가시켜서 대기 상태에서 깨워준 뒤에 스레드를 종료한다.

```

void *queue1_sched(void *param) {
    int num = (int)param;

    //해당 스레드의 semaphore인 semaphore1이 1로 초기화 되어있으므로, 바로 스레드 1이 실행된다.
    //0으로 초기화되어있는 상태인 다른 스레드들은 모두 block된다.
    sem_wait(&semaphore1);
    printf("first queue scheduling in process...\n");

    FCFS_sched(multilevel_queue1, num);
    resort(multilevel_queue1, num);

    //다음 스레드의 semaphore를 1 증가시켜주고, 해당 스레드는 종료처리된다.
    sem_post(&semaphore2);
    pthread_exit(0);
}
  
```

2) 다음 스레드들 또한 이전 큐의 스레드가 해당 큐 스레드의 Semaphore를 1 증가 시켜서 대기 상태에서 깨워주면, sem_wait 함수를 통해 0으로 감소시키면서, 다음 코드 영역을 실행한다. 끝날 때까지 원리는 모두 같게 실행되고, queue 5까지 모두 실행이 끝나면 바로 exit되면서 main thread에서 semaphore가 모두 destroy된다.

```
void *queue2_sched(void *param){
    int num = (int)param;

    //0으로 초기화되어있는 상태였다가 바로 이전 스레드가 종료되기 이전에 값을
    //1로 증가시켜줘서 바로 대기 상태에서 빠져 나오게된다.
    sem_wait(&semaphore2);
    printf("second queue scheduling in process...\n");

    priority_sched(multilevel_queue2, num);
    resort(multilevel_queue2, num);

    //다음스레드의 semaphore를 1 증가시켜주고, 해당 스레드는 종료처리된다.
    sem_post(&semaphore3);
    pthread_exit(0);
}
```

```
void *queue3_sched(void *param) {
    int num = (int)param;

    sem_wait(&semaphore3);
    printf("third queue scheduling in process...\n");

    RR_sched(multilevel_queue3, num, 3);
    resort(multilevel_queue3, num);

    sem_post(&semaphore4);
    pthread_exit(0);
}
```

```
void *queue4_sched(void *param) {

    int num = (int)param;

    sem_wait(&semaphore4);
    printf("forth queue scheduling in process...\n");

    SJF_sched(multilevel_queue4, num);
    resort(multilevel_queue4, num);

    sem_post(&semaphore5);
    pthread_exit(0);
}

void *queue5_sched(void *param) {

    int num = (int)param;

    sem_wait(&semaphore5);
    printf("fifth queue scheduling in process...\n");

    absolute_priority_sched(multilevel_queue5, num, 3);
    resort(multilevel_queue5, num);

    //마지막 스레드이므로 바로 종료시켜준다. 어차피 바로 semaphore가 destroy되게 된다.
    pthread_exit(0);
}
```

5. Simulator를 통한 성능 평가 및 결과 분석

: 본인이 직접 구현해본 코드를 모두 분석해보았으니, 이제 각 case들의 실행 파일과 예측 결과가 맞는지 확인하고, simulation결과를 분석하여 어떤 스케줄링 알고리즘이 어떤 측면에서 성능이 우수한지 비교해 보려고 한다.

1. Case 1, 2: 예측 결과와 실제 결과 비교

-1) Case 1

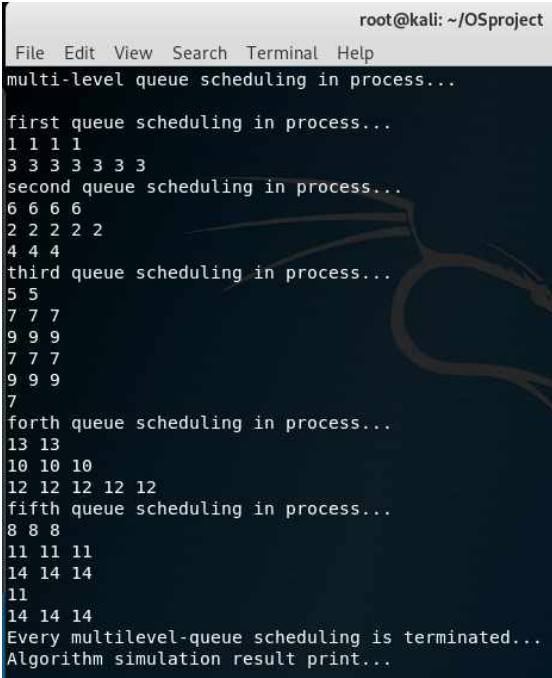
=> linux에서 컴파일했던 결과와, 예측 결과가 정확히 동일한 것을 비교하면서 확인했고, 세마포어 동기화가 정확히 되었는지 확인해 보기위해 "nth queue scheduling in process.."라는 문장을 추가했는데, queue1->queue2->queue3->queue4->queue5 순서대로 동기화가 잘 된 것을 확인할 수 있다.

따라서 스케줄러 코드를 맞게 작성했음을 확인했지만, case 2를 통해 한번 더 코드가 정확한지 확인해볼 것이다.

실제 컴파일한 결과	앞서 예측한 결과
<pre> first queue scheduling in process... 1 1 1 1 1 1 1 4 4 4 4 4 6 6 6 6 6 second queue scheduling in process... 10 10 10 10 8 8 8 8 8 8 14 14 14 14 14 third queue scheduling in process... 3 3 3 7 7 7 9 9 9 3 3 3 7 9 9 3 forth queue scheduling in process... 11 11 11 11 12 12 12 12 12 5 5 5 5 5 5 fifth queue scheduling in process... 13 13 13 15 15 15 2 2 2 13 15 15 2 2 2 2 </pre>	<pre> 1 1 1 1 1 1 1 4 4 4 4 6 6 6 6 6 10 10 10 10 8 8 8 8 8 8 14 14 14 14 14 3 3 3 7 7 7 9 9 9 3 3 3 7 9 9 3 11 11 11 11 12 12 12 12 12 5 5 5 5 5 5 13 13 13 15 15 15 2 2 2 13 15 15 2 2 2 2 </pre>

-2) Case 2

: case 2에서도, 실제 컴파일한 결과와, 앞서 예측한 결과를 비교했을 때 정확히 일치하는 것을 확인할 수 있다. case 1,2를 모두 비교해봤을 때, 앞서 예측했던 스케줄링 결과와 실제 컴파일한 결과가 일치하므로 스케줄러 코드에는 문제가 없음을 확인할 수 있었다.

실제 컴파일한 결과	앞서 예측한 결과
 <pre> root@kali: ~/OSproject File Edit View Search Terminal Help multi-level queue scheduling in process... first queue scheduling in process... 1 1 1 1 3 3 3 3 3 3 second queue scheduling in process... 6 6 6 6 2 2 2 2 2 4 4 4 third queue scheduling in process... 5 5 7 7 7 9 9 9 7 7 7 9 9 9 7 forth queue scheduling in process... 13 13 10 10 10 12 12 12 12 12 fifth queue scheduling in process... 8 8 8 11 11 11 14 14 14 11 14 14 14 Every multilevel-queue scheduling is terminated... Algorithm simulation result print... </pre>	<pre> 1 1 1 1 3 3 3 3 3 3 6 6 6 6 2 2 2 2 2 4 4 4 5 5 7 7 7 9 9 9 7 7 7 9 9 9 7 13 13 10 10 10 12 12 12 12 12 8 8 8 11 11 11 14 14 14 11 14 14 14 </pre>

-2) 스케줄링 알고리즘 성능 분석

1) Scheduling criteria란?

: 스케줄링 알고리즘이 일을 처리하는데 있어서, 어떤 부분에서는 우수하고, 어떤 부분에서는 성능이 떨어지는지 체크하기 위한 일종의 평가 척도이다.

해당 프로젝트에서는 CPU utilization, Average Turnaround time, waiting time, Response time의 네가지 척도를 통해 각 알고리즘의 성능을 비교할 것이다.

1. CPU utilization

: CPU가 idle하지 않고 얼마나 효율적으로 사용되고 있는가를 확인할 수 있는 기준이다.

$$\frac{(TotalCPUbursttime)}{(TotalCPUbursttime) + (Totalwaitingtime)}$$

(실제로 CPU가 일한 시간)/(실제 일한시간 + idle 했던 시간) x 100을 하면 퍼센테이지로 표기할 수 있게 된다.

2. Turnaround time

: 수행을 요청한 다음 프로세스가 Terminate될때까지 걸리는 시간을 측정한 척도이다.

모든 프로세스의 average turnaround time을 통해서 각 알고리즘 간에 turnaround time을 비교할 수 있다.

3. Waiting time

: 프로세스가 실행을 요청은 하였지만, 다른 프로세스가 이미 스케줄링 되어서 대기하고 있는 시간을 계산한 척도이다.

4. Response time

: 프로세스가 실행을 요청한 이후로, 가장 처음 CPU가 응답할 때 까지 걸리는 시간을 계산한 척도이다.

=> 그렇다면, 이제부터 모든 cpu burst time, priority 등의 조건을 모든 큐의 프로세스가 동일하게 하여, 같은 조건 안에서 어떤 스케줄러가 성능이 우수한지 살펴보기 위해 만든 Case 1의 Simulation 결과들을 비교해보면서 각 알고리즘간의 성능을 분석해 보도록 할 것이다.

2) 실제 알고리즘간 성능 비교

- Simulation 결과

=> 알고리즘간 성능을 비교하기 위해 만든 input file 1으로 프로세스들을 스케줄링 해가면서 각 알고리즘의 scheduling criteria 수치를 측정하여 simulation_print 함수를 통해 출력하였고, 이를 바탕으로 각 알고리즘 간의 성능을 비교해 보려고 한다.

1. FCFS Scheduler

```
First come first Serve (FCFS)
=====FCFS=====
C#    P#    BT    Pri    WT    RT    A_PRI    TAT
1      1      4      1      7      7      5      11
1      4      4      1      7      7      5      11
1      6      11     11     8      16
total waiting time:18.000000
average turnaround time:11.33
CPU Utilization rate: 69.49%
=====
```

2. Priority scheduler

```
Priority
=====Priority=====
C#    P#    BT    Pri    WT    RT    A_PRI    TAT
2      8      7      4      9      9      11
2      10     0      0      5      4
2      14      5      3      11     11     8      16
total waiting time:15.000000
average turnaround time:10.33
CPU Utilization rate:73.21%
=====
```

3. RR scheduler

```
RR(Round-Robin)
=====RR=====
C#    P#    BT    Pri    WT    RT    A_PRI    TAT
3      3      7      2      9      0      9      16
3      7      4      1      9      3      5      13
3      9      5      3      10     6      8      15
total waiting time:28.000000
average turnaround time: 14.67
CPU Utilization rate: 59.42%
=====
```

4. SJF scheduler

```

SJF(Shortest-Job-First)
=====SJF=====
C#    P#    BT    Pri    WT    RT    A_PRI    TAT
4      5      7      2      9      9      9      16
4     11     11     4     11     0     5      4
4     12     12     5     12     4     8      9
total waiting time:13.000000
average turnaround time: 9.67
CPU Utilization rate: 75.93%
=====

```

5. Absolute-Priority scheduler

```

Absolute-Priority
=====Absolute_Priority=====
C#    P#    BT    Pri    WT    RT    A_PRI    TAT
5     13     4      1      6      0      5      10
5     15     5      3      7      3      8      12
total waiting time:22.000000
average turnaround time: 12.67
CPU Utilization rate: 65.08%
=====

```

- 모든 알고리즘들의 criteria를 실제로 측정한 결과를 표로 나타내어 보았다.

	FCFS	Priority	R-R	SJF	Abs_Priority
CPU_Util	69.9 (3)	73.21 (2)	59.42 (5)	75.93 (1)	65.08 (4)
Turnaround time	11.33 (3)	10.33 (2)	14.67 (5)	9.67 (1)	12.67 (4)
Waiting	18 (3)	15 (2)	28 (5)	13 (1)	22 (4)
Response	0,7,11 (5)	0,4,11 (4)	0,3,6 (1)	0,9,4 (3)	0,3,6 (1)
Total Grade	D	B	E	A	C

+) 수치 옆에 있는 괄호 안의 숫자 부분은 단순히 성능 비교 측면에서만 각 criteria별로 순위를 매긴 것이고, Total Grade 부분은 각 스케줄링 알고리즘의 각 criteria별 순위의 평균을 내어 비교해본 척도이다. 자세한 분석은 바로 다음 페이지에서 진행할 것이다.

결과 분석

1. FCFS 알고리즘 (Total Grade: D)

: 어떤 프로세스가 먼저 오느냐가 가장 중요한 알고리즘이다. 성능보다는 오는 프로세스를 바로바로 처리해버리는 데 중점을 둔 알고리즘이다. CPU burst time이 짧은 프로세스가 먼저 도착한다면 SJF 스케줄러와 비슷한 효과를 내어 성능이 매우 좋아질 것으로 보인다. 그렇지만 단순 성능 측면에서는 다른 프로세스보다 월등히 나은 점을 발견하지 못했다.

2. Priority (Total Grade: B)

: 우선순위 순서대로 처리하는데 중점을 둔 알고리즘으로, 성능보다는 급한 프로세스를 먼저 처리하는데 중점을 둔 알고리즘이다. 단순 성능 비교 측면에서는 다른 알고리즘에 비해서 고루 고루 모든 점에서 괜찮은 퍼포먼스를 보여준다. 그렇지만 우선순위가 빠른 프로세스의 CPU burst time이 길다면 turnaround time, waiting, response time이 길어질 가능성이 존재한다.

3. R-R (Total Grade: E)

: 역시 성능보다는 계속 돌아가면서 모든 프로세스들을 차례대로 스케줄링하는데 중점을 두었기 때문에, CPU utilization, turnaround time, waiting time 측면에서는 모두 최악의 성능을 보인다. 하지만 모든 프로세스가 $\{(프로세스의\ 개수-1) \times Quantum\}$ 시간 안에 모두 응답하기 때문에, Response time 측면에서는 가장 좋은 성능을 보였다.

4. SJF (Total Grade: A)

: 스케줄러의 성능을 높이기 위함을 중점으로 한 알고리즘이기 때문에, 가장 CPU-burst time이 짧은 프로세스를 우선적으로 처리해버려 우선 waiting time이 아주 적어지고, 따라서 CPU utilization과 Average turnaround time에서 아주 우수한 성능을 보여준다. 하지만 프로세스를 역시 순서대로 non-preemptive하게 처리하기 때문에, Response time 측면에서는 약간 아쉬운 점을 보였다.

5. Absolute-Priority Scheduling (Total Grade : C)

: Round-Robin 보다는 CPU utilization, Average turnaround time에서 나은 성능을 보여주고, SJF 스케줄링 보다는 Response 측면에서 나은 성능을 보여준다.

Round-Robin 방식을 도입한 만큼 매우 빠른 스케줄링 방식이라고 할 수는 없겠지만, 적절하게 모든 알고리즘을 조금씩 보완했고, CPU burst time과 Priority를 모두 고려하면서 프로세스들을 고르게 스케줄링 했다는 점, 그리고 cpu burst time과 priority간에 비중을 조정할 수 있게 하여 사용자의 목적에 따라 비율을 조정하면서 절대 우선순위를 책정할 수 있도록 한 것이 개선점이라고 할 수 있겠다.

#총평

: 모든 프로세스가 각자 설계된 목적이나, 스케줄링하는 방식이 다르기 때문에 사용자의 목적에 맞게 알고리즘을 선택하면 되겠다. 동일한 조건 하에서 정확한 수치를 측정해 단순 성능 측면에서 분석해본 결과, 모든 분야에서 뛰어난 알고리즘은 딱히 찾을 수 없었다.

1. Response time 측면에서 뛰어나게, 그리고 고르게 모든 프로세스를 돌아가면서 스케줄링하고 싶다면? => Round-Robin scheduling

2. Average turnaround time 이나 **CPU utilization** 측면에서 빠르고 효율적으로 스케줄링 하고 싶다면? => SJF Scheduling

3. 우선순위가 높고 급한 일부부터 빠르게 처리하고 싶다면?
=> Priority Scheduling

4. Priority와 CPU time을 모두 고려하여 프로세스들을 고르게 스케줄링 하고 싶다면?
=> Absolute-Priority Scheduling

5. 오는 순서대로 프로세스를 처리하고 싶다면?
=> FCFS Scheduling

을 선택하면 될 것이다. 앞에서 언급했듯 각 스케줄링마다 뛰어난 부분과 사용하는 목적이 다르기 때문에, 본인이 원하는 목적과 기준에 따라서 알고리즘을 선택하여 스케줄링 한다면 그것이 가장 효율적인 스케줄링이 될 것이다.

6. 개선할 점 및 느낀점

-1) 개선해야 할 점

: 우선 전제가 모든 프로세스의 arrival time이 모두 0으로 설정된 후에 개발하게 된 프로젝트라, arrival time이 다를 때 preemptive하게 스케줄링해주는 방면에서는 부족한 점이 분명히 존재한다.

먼저, Priority Scheduler 코드를 작성할 때, 먼저 우선순위 순서대로 정렬을 해주고 순서대로 스케줄링을 하게 되는데, 이 방법은 모든 프로세스가 동시에 도착하는 해당 전제 하에서만 가능하므로, 매번 스케줄링 decision을 할 때마다 가장 우선순위가 높은 프로세스를 골라서 스케줄링 하는 방식으로 약간 개선하여 코드를 새로 작성해 보기도 하였다. 해당 코드는 SJF 스케줄링 방식과 아예 스케줄링 하는 방식이 같고(scheduling decision을 내릴 때마다 SJF에서는 가장 짧은 프로세스를 선택하고, Priority 방식에서는 우선순위가 높은 프로세스를 선택함) 스케줄러 개발 시의 다양성을 위해서 위 코드 설명 시에는 제외하고, 코드 파일에서는 Priority_Sched 함수 안에서 주석처리 하였다.

그렇지만 정렬 방식의 스케줄링과 결과는 100퍼센트 동일하고, 사실 매번 scheduling 할 때마다 우선순위가 높은 프로세스를 골라 decision을 내린다는 점에서 실제로 프로세스들을 스케줄링 하는 방식과는 정렬 방식보다는 훨씬 유사하기 때문에, 개선하여 코드를 작성해보았다는 점에서 해당 코드 또한 보고서에 첨부하도록 하겠다.

```
//다음 프로세스를 결정해야 하는 순간마다 우선순위가 가장 높은(작은) 프로세스를 골라서 스케줄링한다.
//arrival time이 다 똑같으므로 non-preemptive하다.
int min_priority = 1000;
int min_index = -1;

for (int i = 0; i < n; i++) {
    if (pro[i].terminate_flag == false) {
        //이미 terminate된 프로세스는 스케줄링에서 제외한다.

        if (min_priority > pro[i].pri) {
            //우선순위가 높은 프로세스를 골라서 스케줄링한다.
            min_priority = pro[i].pri;
            min_index = i;
        }
        else if (min_priority == pro[i].pri && pro[min_index].pro_num > pro[i].pro_num) {
            //우선순위가 같다면, process_number가 빠른 순서로 스케줄링한다.
            min_priority = pro[i].pri;
            min_index = i;
        }
    }
}

pro[min_index].resp_t = processing_t;
for (int p = 0; p < pro[min_index].cpu_t; p++) {
    printf("%d ", pro[min_index].pro_num);
    processing_t++;
}
printf("\n");

for (int j = 0; j < n; j++) {
    if (pro[j].terminate_flag == false) {
        if (j != min_index) {
            pro[j].wait_t += pro[min_index].cpu_t;
        }
    }
}

pro[min_index].ta_t = pro[min_index].cpu_t + pro[min_index].wait_t;
pro[min_index].terminate_flag = true;
terminate_pronum++;
}
```

또 Round-robin scheduler와, Absolute-priority scheduler를 설계하는 부분에서도, 절대 우선순위 순서대로 먼저 정렬을 해주고 Round-robin scheduling을 하게 되고, Round-robin scheduler도 프로세스가 입력된 process number순서대로 scheduling을 바로 하게 된다.

이 방식도 non-preemptive하게 모두 같은 때에 프로세스가 큐에 들어왔기 때문에 가능한 스케줄링 방식인 듯 싶다. 이때 preemptive 하게 스케줄링 해주게 된다면, Round-robin scheduler에서는 새로 들어온 프로세스를 바로 다음에 스케줄링하는 방식으로 스케줄러를 짜고, Absolute-Priority Scheduling 방식에서는 만약 새로 들어온 프로세스가 현재 실행되고 있는 프로세스보다 절대 우선순위가 높다면, 다음 실행되는 프로세스는 절대 우선순위가 높고, 새로 들어온 프로세스가 되는 식으로 좀 더 스케줄링 방식을 발전시켜줄 수 있을 것 같다.

결론적으로, 나중에는 Preemptive한 방식으로 arrival time이 모두 다른 프로세스를 스케줄링하는 스케줄러 또한 제작해보려고 한다.

-2) 느낀 점

: 각 스케줄링 알고리즘을 분석해보면서 코드를 짜고, 나만의 스케줄러를 설계하면서 이미 기존에 설계된 스케줄링 알고리즘들에서 어떤 점들을 보완할 수 있을까 하면서 생각해 보는 부분이 흥미로웠다. 다음에는 Absolute_Priority를 통해 다른 방식으로 스케줄링을 해보고 싶다. 또 Scheduling criteria 측면에서 모든 수치를 정확하게 계산하여 비교해보니, 각 알고리즘들이 어떤 부분에선 성능이 우수하고, 어떤 부분에선 성능이 다른 알고리즘에 비해 아쉬운지, 또 정확히 어떤 목적을 갖고 설계된 알고리즘들인지 파악할 수 있었다.

그리고 마지막으로 Multilevel-queue를 다섯 레벨로 구성한 탓에 스레드들을 다섯 개 씩이나 동기화시켜야 하다보니, Semaphore를 통해서 어떻게 한 스레드만 공유 자원에 접근해야 하는지 생각해내야 했는데, 이때 하나의 세마포어를 가지고 활용하는 게 아닌, 각 스레드마다 세마포어를 할당해서 공유 자원을 스레드가 사용한 후에 다음 스레드의 세마포어를 증가시켜 대기상태에서 빠져나오게 하는 아이디어를 생각해 내는 부분이 꽤 어려웠던 것 같다.

이후에도 각각의 스케줄러 알고리즘을 시뮬레이션을 통해 실제로 측정해본 결과들과 나의 코드를 분석하면서 발견한 개선점들을 발판 삼아 다른 방식으로 스케줄러를 짜보고 싶다.