운영체제 Term Report

과목명 : 운영체제

교수님 : 유헌창 교수님

이름 : 이호준

학번 : 2014190701

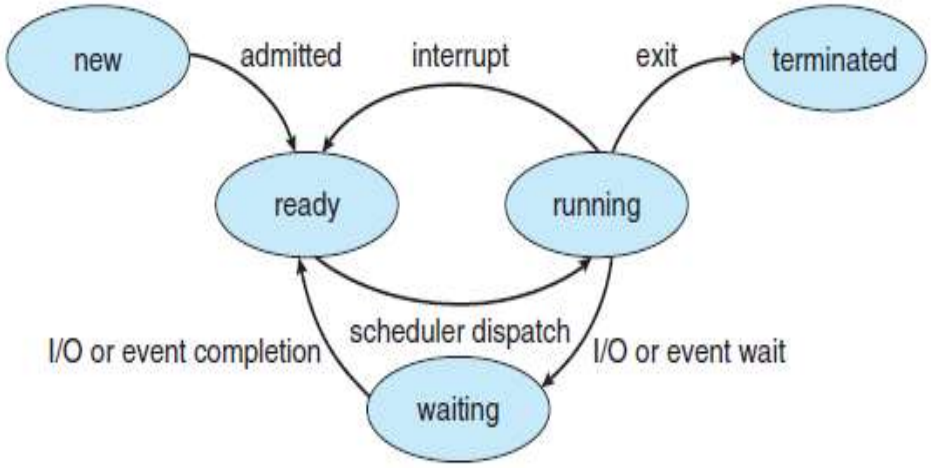
제출일 : 2016. 05. 31

- 목차 -

1장 Process state and Cpu Scheduling

1. Process State

프로세스는 컴퓨터에서 실행되고 있는 컴퓨터 프로그램을 말한다. 운영체제의 핵심 역할중 하나는 효과적인 자원 할당으로, 프로세스들이 효율적으로 실행될 수 있도록 관리한다. 이를 위해서 운영체제의 커널에는 Ready_Queue, Running_Queue, Waiting_Queue등의 자료구조가 있으며, 이것들을 이용해서 프로세스의 상태를 관리하게 된다.



[그림 1] Diagram of process state

(출처 : Operating System Concepts 9th edition p.108 Figure 3.2)

프로세스의 상태는 위와같이 New, Ready, Running, Waiting, Terminated가 존재한다. 프로그램이 job scheduler에 의해 선택받아 메모리에 할당되면 프로세스가 new state가 되고 생성되게 된다. 새롭게 생성된 프로세스는 Ready_Queue에 머물게 된다. CPU Scheduler는 Ready_Queue에 있는 프로세스들 중에 실행될 프로세스를 선택하여 Running_Queue로 보내지고 CPU를 할당받게 된다. CPU를 통해 프로세스가 끝나게 되면 프로세스의 상태는 terminated로 전환되고 그대로 종료된다.

만약, Running상태에 있는 프로세스가 I/O operation이 발생할 경우, 프로세스는 Waiting_Queue로 보내지고 I/O operation을 수행 후 다시 ready_Queue로 돌아오게 된다. 또한, Running__Queue에 있는 프로세스는 interrupt를 받아 ready_Queue로 돌아올 수 있다.

2. CPU scheduling

CPU scheduling(=Short-term Scheduling) 이란 간단히 말해 CPU를 사용하려 하는 프로세스들 간 우선순위를 관리하는 일, 즉 Running_Queue에 어떤 프로세스를 할당할지 관리하는 것이다. CPU Scheduling을 수행하는 방법에 따른 다양한 Scheduling 알고리즘이 존재한다. 또한 프로세스를 할당하는 과정에 있어 Running_Queue에 실행되고 있는 프로세스를 비울 수 있는 지에 따라 preemptive, non_preemptive 방식으로 나눠질 수 있다. 간략히 소개하면 아래와 같다.

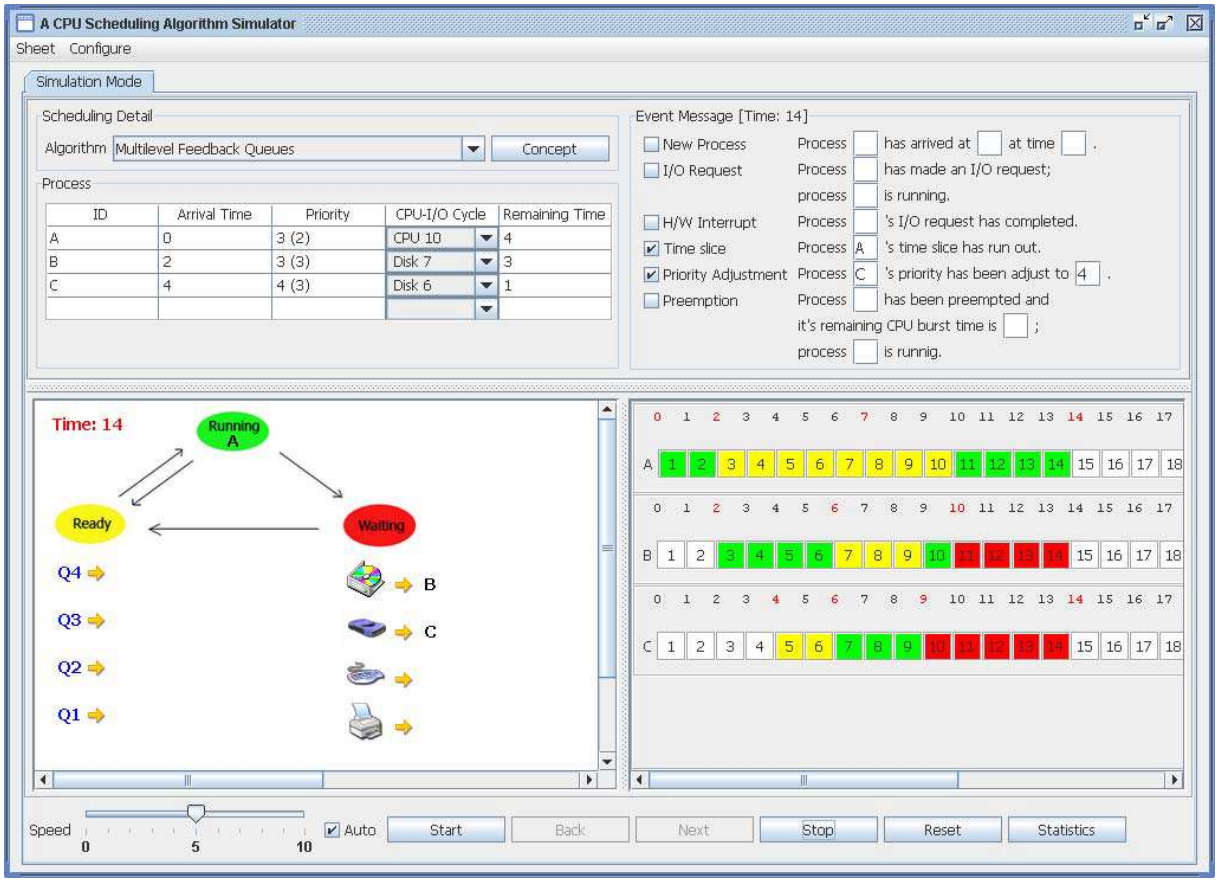
1. FCFS: 프로세스가 들어온 순서대로 처리
2. Shortes Job First: 작업 시간이 짧은 프로세스들을 우선적으로 처리
3. Nonpreemptive Priority: 프로세스들의 우선순위를 바탕으로 처리
4. Round Robin : 프로세스가 들어온 순서대로 처리하되, 특정 시간 이상 수행되면 프로세스가 바뀌는 방식
5. Preemptive Shortest Job First: 작업 시간이 짧은 프로세스들을 우선적으로 처리하되, Running Queue에 있는 프로세스를 비울 수 있다.
6. Preemptive Priority: 우선순위를 바탕으로 프로세스들을 처리하되, Running Queue에 있는 프로세스를 비울 수 있다.
7. Aging Preemptive Priority: Preemptive Priority에서 오래동안 실행되지 못한 프로세스들의 우선순위를 상승시켜준다.
8. Multilevel Queue: 여러 개의 Queue를 만들어 Queue마다 다른 스케줄링을 적용
9. Multilevel Feedback Queue: MLQ에 하위 큐에서 오래동안 실행되지 못한 프로세스들에 대해 상위 큐로 이동시켜 준다.
10. Rate Monotonic: 프로세스의 Period가 낮을수록 우선순위를 높게 처리
11. Earliest Deadline First: 프로세스의 Period가 다가올수록 우선순위를 높게 처리

본문에서는 위와 같은 다양한 스케줄링 기법의 구현을 통해서 이들의 성능을 비교, 분석한다.

2장 CPU scheduling Simulator

1. 다른 CPU Scheduling 시뮬레이터

Sukanya Suranauwarat의 시뮬레이터에서는 FCFS, RR, SJF, SRTF(=Preemptive SJF), MLFQ(multilevel feedback queues)를 구현하였다. 그리고 running 큐, ready 큐, waiting 큐로 이동하는 것과 간트 차느를 그래픽을 통해 구현하였으며, 프로세스의 우선순위, 도착시간등을 임의로 할당할 수 있다.



2. Simulator 시스템 구성도

I. 환경

- 개발 환경: LINUX 환경: https://c9.io/, MS visual studio

- 개발 언어: C언어

- 컴파일 및 실행: GCC, 터미널

II. 구조체

1) PCB

구조체를 사용하여 Process Control Block을 구현하였다. PCB를 이루는 구성요소로 프로세스 id (pid), 프로세스 도착시간(arrive_time), 프로세스 대기시간(waiting_time), 프로세스 실행시간 (execute_time), 프로세스 burst time(burst_time), 남은 burst 시간(remain_time), 프로세스 종료 시간(finish_time), IO operation이 실행되는 시간(io_start_time), IO operation이 수행되야 하는 시간 (io_remain_time), 프로세스 우선순위(priority), Realtime에서의 프로세스 period(period) 를 지정하였다.

2) RESULT

결과를 출력하기 위한 구조체이다. 구성요소에는 프로세스 개수(process), 문맥교환 횟수 (context_switch), 프로세스 대기 시간 총합(sum_wait), 프로세스 burst 시간 총합(sum_burst), 프로세스 turnaround 총합(sum_turn), 프로세스 평균 대기 시간(avg_wait), 프로세스 평균 burst 시간 (avg_burst), 프로세스 평균 turnaround 시간(avg_turn), IO 발생 횟수(io_count)

3) Queue

Ready_Queue, Waiting_Queue등 큐를 구현하기 위한 구조체이다. Queue의 기본적 속성인 front, rear, size가 있으며 Queue에 있는 프로세스의 개수를 나타낸 count, 프로세스들의 정보를 담는 배열을 가르치는 포인터(PCB *buffer)가 있다.

4) Context_switch

스케줄링 알고리즘 별로 context_switch 횟수를 세는 전역 변수

5) Time

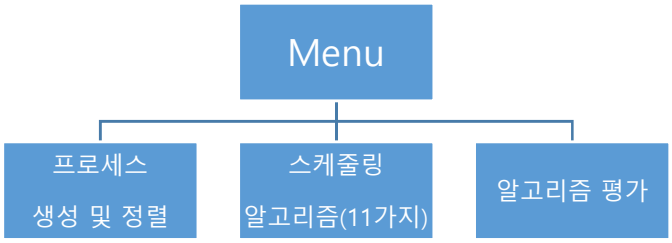
시간의 경과를 나타내기 위한 전역 변수

6) Gantt

Gantt차트를 그리기 위한 배열

3. Simulator 모듈

시뮬레이터를 구성하는 주요 함수들(module 요소들)을 도식화하면 아래와 같다.



1) menu()

menu()에서는 사용자가 몇 개의 프로세스를 생성할지 수를 입력받아 프로세스를 생성하고, 생성된 프로세스를 자신이 선택한 스케줄링 알고리즘의 진행상황을 확인할 수 있다. 또한 생성된 프로세스들에 대해 전체 알고리즘 별로 Average Waiting Time, Average Turnaround Time등으로 비교해 볼 수 있다.

2) process_num()

menu()에서 불려 사용자로부터 원하는 프로세스 개수를 받아온다.

3) init_Queue(Queue *queue, int p_num)

Process_num()을 통해서 입력받은 프로세스 개수를 통해 Queue를 초기화 한다.

4) fill_random_process(Queue *queue)

초기화된 Queue에 랜덤하게 생성된 프로세스들을 채워 넣는다. 프로세스의 arrive_time은 0~MAX_ARRIVE_TIME -1, burst_time은 1~MAX_BURST_TIME으로 초기화되고, priority, period 또한 랜덤하게 초기화 된다. IO는 평균적으로 IO_FREQ번에 한번 씩 프로세스마다 발생하게 된다.

5) sort_by_arrival(Queue *queue)

큐에 있는 프로세스들을 도착시간에 따라서 정렬한다.

6) sort_by_remain(Queue *queue)

큐에 있는 프로세스들을 남은 cpu burst_time에 따라서 정렬한다.

7) sort_by_priority(Queue *queue)

큐에 있는 프로세스들을 우선순위에 따라서 정렬. (우선순위는 값이 낮을수록 우선순위가 높다)

8) FCFS(Queue *queue)

큐(Ready, Waiting, Terminated, Running, FCFS)를 생성한다. 처음에 queue에서 FCFS로 프로세스들을 받은뒤 프로세스들을 도착시간에 따라 정렬한다. 모든 프로세스가 Terminated_Queue에 가득찰 때까지 아래의 과정을 반복 수행하며 Time을 증가시키고, 가득찼을 때 해당 알고리즘의 수행결과에 대한 지표들을 Reulst에 저장한다.

1. FCFS에 들어있는 프로세스의 arrive_time이 Time과 같아질 때 Ready_Queue에 삽입한다.
2. Waiting_Queue에 프로세스가 존재한다면, Waiting_Queue에 가장 첫번째로 있는 프로세스의 io 수행시간을 1만큼 줄이고 만약 io수행시간이 끝났다면 Ready_Queue로 이동시켜주며, cpu_burst_time 또한 끝나있다면 Terminated_Queue로 이동시킨다.
3. Running_Queue에 프로세스가 있다면, 프로세스의 burst_time을 1만큼 줄인다. 프로세스의 cpu_burst_time이 0이되고, 필요한 io 수행시간도 없다면 프로세스를 Terminated_Queue로 이동시킨다. 만약, 프로세스의 io 발생시간이 되었다면 Waiting_Queue로 이동시킨다.
4. Running_Queue가 비어있고, Ready_Queue에서 기다리고 있는 프로세스가 있다면 Ready_Queue에 제일 앞에 있는 프로세스를 Running_Queue로 이동시킨다.

9) Nonpreemptive_SJF(Queue *queue)

큐(Ready, Waiting, Terminated, Running, SJF)를 생성한다. 처음에 queue에서 SJF로 프로세스들을 받은뒤 프로세스들을 도착시간에 따라 정렬한다. 모든 프로세스가 Terminated_Queue에 가득찰 때까지 아래의 과정을 반복 수행하며 Time을 증가시키고, 가득찼을 때 해당 알고리즘의 수행 결과에 대한 지표들을 Result에 저장한다.

1. SJF에 들어있는 프로세스의 arrive_time이 Time과 같아질 때 Ready_Queue에 삽입한다.
2. Ready_Queue에 있는 프로세스들을 남은 시간 순으로 정렬한다.
3. Waiting_Queue에 프로세스가 존재한다면, Waiting_Queue에 가장 첫번째로 있는 프로세스의 io 수행시간을 1만큼 줄이고 만약 io수행시간이 끝났다면 Ready_Queue로 이동시켜주며, cpu_burst_time 또한 끝나있다면 Terminated_Queue로 이동시킨다.
4. Running_Queue에 프로세스가 있다면, 프로세스의 burst_time을 1만큼 줄인다. 프로세스의 cpu_burst_time이 0이되고, 필요한 io 수행시간도 없다면 프로세스를 Terminated_Queue로 이동시킨다. 만약, 프로세스의 io 발생시간이 되었다면 Waiting_Queue로 이동시킨다.
5. Running_Queue가 비어있고, Ready_Queue에서 기다리고 있는 프로세스가 있다면 Ready_Queue에 제일 앞에 있는 프로세스를 Running_Queue로 이동시킨다.

10) Nonpreemptive_Priority(Queue *queue)

큐(Ready, Waiting, Terminated, Running, PRIORITY)를 생성한다. 처음에 queue에서 PRIORITY로 프로세스들을 받은뒤 프로세스들을 도착시간에 따라 정렬한다. 모든 프로세스가 Terminated_Queue에 가득찰 때까지 아래의 과정을 반복 수행하며 Time을 증가시키고, 가득찼을 때 해당 알고리즘의 수행 결과에 대한 지표들을 Result에 저장한다.

1. PRIORITY에 들어있는 프로세스의 arrive_time이 Time과 같아질 때 Ready_Queue에 삽입한다.
2. Ready_Queue에 있는 프로세스들을 우선 순위 순으로 정렬한다.
3. Waiting_Queue에 프로세스가 존재한다면, Waiting_Queue에 가장 첫번째로 있는 프로세스의 io 수행시간을 1만큼 줄이고 만약 io수행시간이 끝났다면 Ready_Queue로 이동시켜주며, cpu_burst_time 또한 끝나있다면 Terminated_Queue로 이동시킨다.
4. Running_Queue에 프로세스가 있다면, 프로세스의 burst_time을 1만큼 줄인다. 프로세스의 cpu_burst_time이 0이되고, 필요한 io 수행시간도 없다면 프로세스를 Terminated_Queue로 이동시킨다. 만약, 프로세스의 io 발생시간이 되었다면 Waiting_Queue로 이동시킨다.
5. Running_Queue가 비어있고, Ready_Queue에서 기다리고 있는 프로세스가 있다면 Ready_Queue에 제일 앞에 있는 프로세스를 Running_Queue로 이동시킨다.

11) Round_Robin(Queue *queue)

큐(Ready, Waiting, Terminated, Running, RR)를 생성한다. 처음에 queue에서 RR로 프로세스들을 받은뒤 프로세스들을 도착시간에 따라 정렬한다. 모든 프로세스가 Terminated_Queue에 가득찰 때까지 아래의 과정을 반복 수행하며 Time을 증가시키고, 가득찼을 때 해당 알고리즘의 수행 결과에 대한 지표들을 Result에 저장한다.

1. RR에 들어있는 프로세스의 arrive_time이 Time과 같아질 때 Ready_Queue에 삽입한다.
2. Waiting_Queue에 프로세스가 존재한다면, Waiting_Queue에 가장 첫번째로 있는 프로세스의 io 수행시간을 1만큼 줄이고 만약 io수행시간이 끝났다면 Ready_Queue로 이동시켜주며, cpu_burst_time 또한 끝나있다면 Terminated_Queue로 이동시킨다.
3. Running_Queue에 프로세스가 있다면, 프로세스의 burst_time을 1만큼 줄인다. 프로세스의 cpu_burst_time이 0이되고, 필요한 io 수행시간도 없다면 프로세스를 Terminated_Queue로 이동시킨다. 만약, 프로세스의 io 발생시간이 되었다면 Waiting_Queue로 이동시킨다.
4. Running_Queue에 있던 프로세스가 time_quantum만큼 수행 되었다면 Ready_Queue로 해당 프로세스를 이동시킨다.
5. Running_Queue가 비어있고, Ready_Queue에서 기다리고 있는 프로세스가 있다면 Ready_Queue에 제일 앞에 있는 프로세스를 Running_Queue로 이동시킨다.

12) Preemptive_SJF(Queue *queue)

큐(Ready, Waiting, Terminated, Running, SJF)를 생성한다. 처음에 queue에서 SJF로 프로세스들을 받은뒤 프로세스들을 도착시간에 따라 정렬한다. 모든 프로세스가 Terminated_Queue에 가득찰 때까지 아래의 과정을 반복 수행하며 Time을 증가시키고, 가득찼을 때 해당 알고리즘의 수행 결과에 대한 지표들을 Result에 저장한다.

1. SJF에 들어있는 프로세스의 arrive_time이 Time과 같아질 때 Ready_Queue에 삽입한다.
2. Ready_Queue에 있는 프로세스들을 남은 시간 순으로 정렬한다.
3. Waiting_Queue에 프로세스가 존재한다면, Waiting_Queue에 가장 첫번째로 있는 프로세스의 io 수행시간을 1만큼 줄이고 만약 io수행시간이 끝났다면 Ready_Queue로 이동시켜주며, cpu_burst_time 또한 끝나있다면 Terminated_Queue로 이동시킨다.
4. Running_Queue에 프로세스가 있다면, 프로세스의 burst_time을 1만큼 줄인다. 프로세스의 cpu_burst_time이 0이되고, 필요한 io 수행시간도 없다면 프로세스를 Terminated_Queue로 이동시킨다. 만약, 프로세스의 io 발생시간이 되었다면 Waiting_Queue로 이동시킨다.
5. 만약 Ready_Queue에 있는 프로세스의 남은 시간이 Running_Queue에 첫번째로 있는 프로세스의 남은 시간보다 많다면 Running_Queue에 있는 프로세스를 Ready_Queue로 이동시켜 Running_Queue를 비운다.
6. Running_Queue가 비어있고, Ready_Queue에서 기다리고 있는 프로세스가 있다면 Ready_Queue에 제일 앞에 있는 프로세스를 Running_Queue로 이동시킨다.

13) Preemptive_Priority(Queue *queue, int aging)

Aging은 Priority 스케줄러에서 우선순위가 낮은 프로세스가 starvation에 시달릴 때, 해당 프로세스의 우선순위를 인위적으로 높여줌으로서 문제를 해결 하는 기법이다. aging의 값이 1일 때 aging기법을 사용하고, 0일 때는 사용하지 않게 된다.

큐(Ready, Waiting, Terminated, Running, PRIORITY)를 생성한다. 처음에 queue에서 PRIORITY로 프로세스들을 받은뒤 프로세스들을 도착시간에 따라 정렬한다. 모든 프로세스가 Terminated_Queue에 가득찰 때까지 아래의 과정을 반복 수행하며 Time을 증가시키고, 가득찼을 때 해당 알고리즘의 수행 결과에 대한 지표들을 Result에 저장한다.

1. PRIORITY에 들어있는 프로세스의 arrive_time이 Time과 같아질 때 Ready_Queue에 삽입한다.
2. Ready_Queue에 있는 프로세스들을 우선 순위 순으로 정렬한다.
3. Waiting_Queue에 프로세스가 존재한다면, Waiting_Queue에 가장 첫번째로 있는 프로세스의

io 수행시간을 1만큼 줄이고 만약 io수행시간이 끝났다면 Ready_Queue로 이동시켜주며, cpu_burst_time 또한 끝나있다면 Terminated_Queue로 이동시킨다.

4. Running_Queue에 프로세스가 있다면, 프로세스의 burst_time을 1만큼 줄인다. 프로세스의 cpu_burst_time이 0이되고, 필요한 io 수행시간도 없다면 프로세스를 Terminated_Queue로 이동시킨다. 만약, 프로세스의 io 발생시간이 되었다면 Waiting_Queue로 이동시킨다.
5. 만약 Ready_Queue에 있는 프로세스의 우선순위가 Running_Queue에 첫번째로 있는 프로세스의 우선순위보다 낮다면 Running_Queue에 있는 프로세스를 Ready_Queue로 이동시켜 Running_Queue를 비운다.
6. Running_Queue가 비어있고, Ready_Queue에서 기다리고 있는 프로세스가 있다면 Ready_Queue에 제일 앞에 있는 프로세스를 Running_Queue로 이동시킨다.
7. Aging값이 1일 때, Ready_Queue에서 starvation에 시달리고 있는 프로세스들의 우선순위를 높여준다.

14) Multilevel_Queue(Queue *queue, int aging)

2개의 레벨의 큐(Ready, Waiting, Terminated, Running, Queue)를 생성한다. 위의 레벨의 큐는 일반적으로 Front 레벨의 프로세스들이기 때문에 Round Robin방식으로, 아래 레벨의 큐는 FCFS방식으로 구현한다. CPU에 높은 레벨의 큐를 더 많이 할당시키기 때문에 아래 레벨의 큐의 프로세스들은 starvation에 시달릴 수 있다. 만약, aging이 1이라면 starvation에 시달리는 프로세스를 위의 레벨의 큐로 승격시켜줌으로서 Multilevel Feedback Queue를 작동시킨다.

처음에 queue에서 우선순위가 높은 프로세스들은 RR Queue로, 낮은 프로세스들은 FCFS Queue로 이동시킨후 RR, FCFS의 프로세스들을 도착시간에 따라 정렬한다. 모든 프로세스가 Terminated_Queue에 가득찰 때까지 아래의 과정을 반복 수행하며 Time을 증가시키고, 가득찼을 때 해당 알고리즘의 수행 결과에 대한 지표들을 Result에 저장한다.

1. RR, FCFS에 들어있는 프로세스의 arrive_time이 Time과 같아질 때 각각 레벨의 Ready_Queue에 삽입한다.
2. Waiting_Queue에 프로세스가 존재한다면, Waiting_Queue에 가장 첫번째로 있는 프로세스의 io 수행시간을 1만큼 줄이고 만약 io수행시간이 끝났다면 Ready_Queue로 이동시켜주며, cpu_burst_time 또한 끝나있다면 Terminated_Queue로 이동시킨다.
3. CPU가 RR방식의 큐에 할당되는 시간일 때,

    i) RR의 Running_Queue에 프로세스가 있다면, 프로세스의 burst_time을 1만큼 줄인다. 프로세스의 cpu_burst_time이 0이되고, 필요한 io 수행시간도 없다면 프로세스를

Terminated_Queue로 이동시킨다. 만약, 프로세스의 io 발생시간이 되었다면 Waiting_Queue로 이동시킨다.

ii) RR의 Running_Queue에 있던 프로세스가 time_quantum만큼 수행 되었다면 Ready_Queue로 해당 프로세스를 이동시킨다.

iii) RR의 Running_Queue가 비어있고, RR의 Ready_Queue에서 기다리고 있는 프로세스가 있다면, RR의 Ready_Queue에 제일 앞에 있는 프로세스를 RR의 Running_Queue로 이동시킨다.이동시킨다. 만약, 프로세스의 io 발생시간이 되었다면 Waiting_Queue로 이동시킨다.

4. CPU가 FCFS방식의 큐에 할당되는 시간일 때,

   i) FCFS의 Running_Queue에 프로세스가 있다면, 프로세스의 burst_time을 1만큼 줄인다. 프로세스의 cpu_burst_time이 0이되고, 필요한 io 수행시간도 없다면 프로세스를 Terminated_Queue로 이동시킨다. 만약, 프로세스의 io 발생시간이 되었다면 Waiting_Queue로 이동시킨다.

   ii) FCFS의 Running_Queue가 비어있고, FCFS의 Ready_Queue에서 기다리고 있는 프로세스가 있다면 Ready_Queue에 제일 앞에 있는 프로세스를 Running_Queue로 이동시킨다.

   iii) aging이 1일 때, FCFS의 Ready_Queue에서 starvation에 시달리는 프로세스가 있다면 우선순위를 높여주고, 우선순위가 충분히 높아졌을 때 FCFS로 승격시켜준다.

15) Realtime(Queue *queue, int edf)

Realtime상황에서의 스케줄링 기법은 크게 Rate monotonic방식과 Earliest Deadline First로 구분되는데, edf값이 0일 때는 Rate monotonic방식으로, 1일 때는 Earliest Deadline First방식으로 구현된다.

큐(Ready, Waiting, Terminated, Running, PRIORITY)를 생성한다. Priority값과 마찬가지로 Period 값이 낮을수록 우선순위가 높음으로 Priority값을 Period값으로 바꾼다. 주어진 queue에서 PRIORITY로 프로세스들을 받은뒤 프로세스들의 도착시간을 0으로 초기화 시켜준다. MAX_PERIOD값만큼 시간이 흐를 때 까지 아래의 과정을 반복 수행하며 Time을 증가시키고, 종료될 때 해당 알고리즘의 수행 결과에 대한 지표들을 Result에 저장한다.

1. PRIORITY에 들어있는 프로세스들의 period값의 배수 값이 Time과 같아질 때 Ready_Queue에 해당 프로세스를 삽입한다. 만약, 아직도 ready_Queue에서 사라지지 않았다면 deadline miss 문구를 띄운다.
2. 만약 Earliest Deadline First방식이라면 Ready_Queue에 있는 프로세스들의 우선순위를 1만큼

씩 높인다(deadline이 다가온다).

3. Ready_Queue에 있는 프로세스들을 우선 순위 순으로 정렬한다.
4. Waiting_Queue에 프로세스가 존재한다면, Waiting_Queue에 가장 첫번째로 있는 프로세스의 io 수행시간을 1만큼 줄이고 만약 io수행시간이 끝났다면 Ready_Queue로 이동시켜주며, cpu_burst_time 또한 끝나있다면 Terminated_Queue로 이동시킨다.
5. Running_Queue에 프로세스가 있다면, 프로세스의 burst_time을 1만큼 줄인다. 프로세스의 cpu_burst_time이 0이되고, 필요한 io 수행시간도 없다면 프로세스를 Terminated_Queue로 이동시킨다. 만약, 프로세스의 io 발생시간이 되었다면 Waiting_Queue로 이동시킨다.
6. 만약 Ready_Queue에 있는 프로세스의 우선순위가 Running_Queue에 첫번째로 있는 프로세스의 우선순위보다 낮다면 Running_Queue에 있는 프로세스를 Ready_Queue로 이동시켜 Running_Queue를 비운다.
7. Running_Queue가 비어있고, Ready_Queue에서 기다리고 있는 프로세스가 있다면 Ready_Queue에 제일 앞에 있는 프로세스를 Running_Queue로 이동시킨다.

16) print_Gantt(int time)

해당 Time시간까지의 Gantt차트를 출력한다.

17) print_all_result()

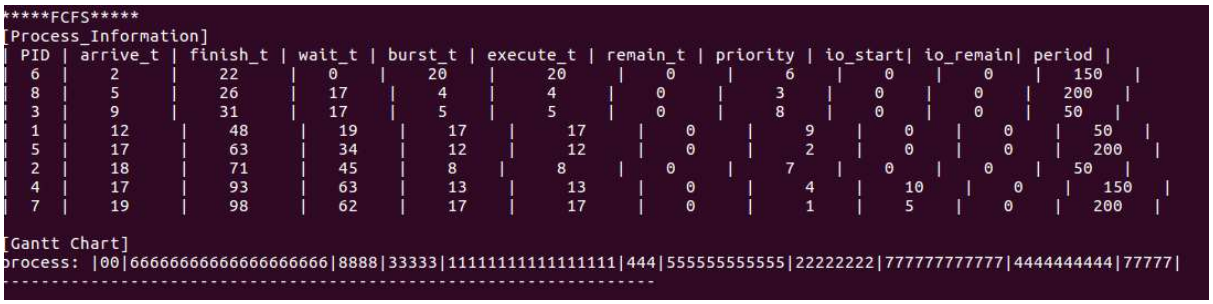
11가지의 CPU 스케줄링 알고리즘들에 대해서 알고리즘별 Context_switch 횟수, Waiting Time, Burst Time, Turnaroud Time을 출력한다.

4. Simulator 실행 결과 화면

[프로세스 생성 및 정렬]

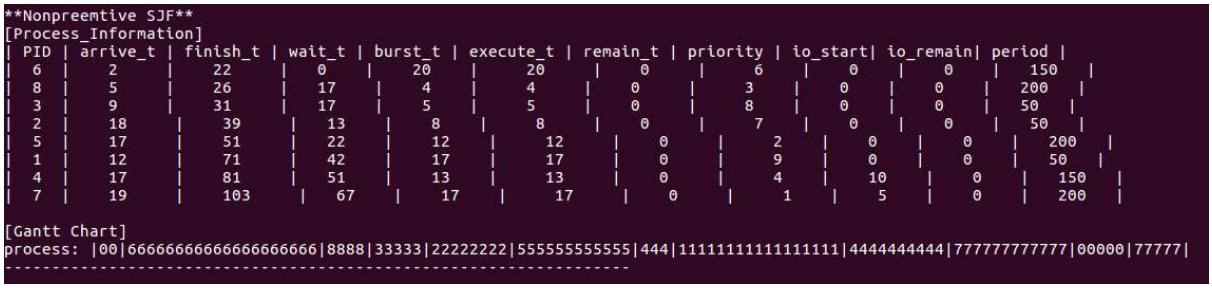
```
[MENU]
[0. 프로세스 생성]
[1. Scheduling 알고리즘]
[2. 전체 알고리즘 평가]
[3. 종료]
[수행 하고 싶은 명령을 선택하세요:]
 0
프로세스 갯수 입력(1-10) : 8
[생성된 큐 출력]
[Process_Information]
| PID | arrive_t | finish_t | wait_t | burst_t | execute_t | remain_t | priority | io_start| io_remain| period |
|  1  |    12    |    0     |   0    |   17    |     0     |    17    |     9    |    0    |    0     |   50   |
|  2  |    18    |    0     |   0    |    8    |     0     |    8     |     7    |    0    |    0     |   50   |
|  3  |    9     |    0     |   0    |    5    |     0     |    5     |     8    |    0    |    0     |   50   |
|  4  |    17    |    0     |   0    |   13    |     0     |    13    |     4    |   10    |    3     |  150   |
|  5  |    17    |    0     |   0    |   12    |     0     |    12    |     2    |    0    |    0     |  200   |
|  6  |    2     |    0     |   0    |   20    |     0     |    20    |     6    |    0    |    0     |  150   |
|  7  |    19    |    0     |   0    |   17    |     0     |    17    |     1    |    5    |    5     |  200   |
|  8  |    5     |    0     |   0    |    4    |     0     |    4     |     3    |    0    |    0     |  200   |

[도착 순서대로 보기]
[Process_Information]
| PID | arrive_t | finish_t | wait_t | burst_t | execute_t | remain_t | priority | io_start| io_remain| period |
|  6  |    2     |    0     |   0    |   20    |     0     |    20    |     6    |    0    |    0     |  150   |
|  8  |    5     |    0     |   0    |    4    |     0     |    4     |     3    |    0    |    0     |  200   |
|  3  |    9     |    0     |   0    |    5    |     0     |    5     |     8    |    0    |    0     |   50   |
|  1  |    12    |    0     |   0    |   17    |     0     |    17    |     9    |    0    |    0     |   50   |
|  4  |    17    |    0     |   0    |   13    |     0     |    13    |     4    |   10    |    3     |  150   |
|  5  |    17    |    0     |   0    |   12    |     0     |    12    |     2    |    0    |    0     |  200   |
|  2  |    18    |    0     |   0    |    8    |     0     |    8     |     7    |    0    |    0     |   50   |
|  7  |    19    |    0     |   0    |   17    |     0     |    17    |     1    |    5    |    5     |  200   |
```

[FCFS]



```
*****FCFS*****
[Process_Information]
| PID | arrive_t | finish_t | wait_t | burst_t | execute_t | remain_t | priority | io_start| io_remain| period |
|  6  |    2     |    22    |   0    |   20    |    20     |    0     |     6    |    0    |    0     |  150   |
|  8  |    5     |    26    |   17   |    4    |     4     |    0     |     3    |    0    |    0     |  200   |
|  3  |    9     |    31    |   17   |    5    |     5     |    0     |     8    |    0    |    0     |   50   |
|  1  |    12    |    48    |   19   |   17    |    17     |    0     |     9    |    0    |    0     |   50   |
|  5  |    17    |    63    |   34   |   12    |    12     |    0     |     2    |    0    |    0     |  200   |
|  2  |    18    |    71    |   45   |    8    |     8     |    0     |     7    |    0    |    0     |   50   |
|  4  |    17    |    93    |   63   |   13    |    13     |    0     |     4    |   10    |    0     |  150   |
|  7  |    19    |    98    |   62   |   17    |    17     |    0     |     1    |    5    |    0     |  200   |

[Gantt Chart]
Process: |00|6666666666666666666|8888|33333|1111111111111111|444|555555555555|22222222|777777777777|4444444444|77777|
-----------------------------------------------------------------
```

[Shortest Job First]



```
**Nonpreemtive SJF**
[Process_Information]
| PID | arrive_t | finish_t | wait_t | burst_t | execute_t | remain_t | priority | io_start| io_remain| period |
|  6  |    2     |    22    |   0    |   20    |    20     |    0     |     6    |    0    |    0     |  150   |
|  8  |    5     |    26    |   17   |    4    |     4     |    0     |     3    |    0    |    0     |  200   |
|  3  |    9     |    31    |   17   |    5    |     5     |    0     |     8    |    0    |    0     |   50   |
|  2  |    18    |    39    |   13   |    8    |     8     |    0     |     7    |    0    |    0     |   50   |
|  5  |    17    |    51    |   22   |   12    |    12     |    0     |     2    |    0    |    0     |  200   |
|  1  |    12    |    71    |   42   |   17    |    17     |    0     |     9    |    0    |    0     |   50   |
|  4  |    17    |    81    |   51   |   13    |    13     |    0     |     4    |   10    |    0     |  150   |
|  7  |    19    |    103   |   67   |   17    |    17     |    0     |     1    |    5    |    0     |  200   |

[Gantt Chart]
process: |00|6666666666666666666|8888|33333|22222222|555555555555|444|1111111111111111|4444444444|777777777777|00000|77777|
-----------------------------------------------------------------
```

[Nonpreemptive Priority]

```
**Nonpreemtive Priority**
[Process_Information]
| PID | arrive_t | finish_t | wait_t | burst_t | execute_t | remain_t | priority | io_start| io_remain| period |
|  6  |    2     |    22    |   0    |    20   |    20     |    0     |    6     |    0    |    0     |   150  |
|  5  |    17    |    46    |   17   |    12   |    12     |    0     |    2     |    0    |    0     |   200  |
|  7  |    19    |    51    |   15   |    17   |    17     |    0     |    1     |    5    |    0     |   200  |
|  8  |    5     |    55    |   46   |    4    |    4      |    0     |    3     |    0    |    0     |   200  |
|  2  |    18    |    66    |   40   |    8    |    8      |    0     |    7     |    0    |    0     |   50   |
|  4  |    17    |    76    |   46   |    13   |    13     |    0     |    4     |    10   |    0     |   150  |
|  3  |    9     |    81    |   67   |    5    |    5      |    0     |    8     |    0    |    0     |   50   |
|  1  |    12    |    98    |   69   |    17   |    17     |    0     |    9     |    0    |    0     |   50   |

[Gantt Chart]
process: |00|66666666666666666666|777777777777|555555555555|77777|8888|444|22222222|4444444444|33333|1111111111111111|
-----------------------------------------------------------------
```

[Round Robin]

```
***Round Robin***
[Process_Information]
| PID | arrive_t | finish_t | wait_t | burst_t | execute_t | remain_t | priority | io_start| io_remain| period |
|  8  |    5     |    12    |   3    |    4    |     4     |    0     |    3     |    0    |    0     |   200  |
|  3  |    9     |    23    |   9    |    5    |     5     |    0     |    8     |    0    |    0     |   50   |
|  2  |    18    |    67    |   41   |    8    |     8     |    0     |    7     |    0    |    0     |   50   |
|  5  |    17    |    82    |   53   |    12   |    12     |    0     |    2     |    0    |    0     |   200  |
|  6  |    2     |    84    |   62   |    20   |    20     |    0     |    6     |    0    |    0     |   150  |
|  4  |    17    |    91    |   61   |    13   |    13     |    0     |    4     |    10   |    0     |   150  |
|  1  |    12    |    93    |   64   |    17   |    17     |    0     |    9     |    0    |    0     |   50   |
|  7  |    19    |    98    |   62   |    17   |    17     |    0     |    1     |    5    |    0     |   200  |

[Gantt Chart]
process: |00|666|888|666|8|333|666|111|33|444|555|222|666|777|111|444|555|222|666|777|111|444|555|22|666|777|111|444|555|66|777|1
11|4|11|77777|
-----------------------------------------------------------------
```

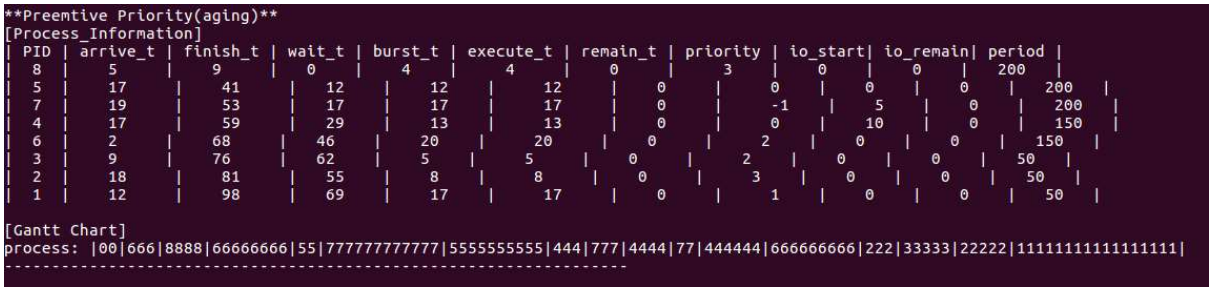
[Preemptive SJF]

```
**Preemtive SJF**
[Process_Information]
| PID | arrive_t | finish_t | wait_t | burst_t | execute_t | remain_t | priority | io_start| io_remain| period |
|  8  |    5     |    9     |   0    |    4    |     4     |    0     |    3     |    0    |    0     |   200  |
|  3  |    9     |    14    |   0    |    5    |     5     |    0     |    8     |    0    |    0     |   50   |
|  2  |    18    |    26    |   0    |    8    |     8     |    0     |    7     |    0    |    0     |   50   |
|  5  |    17    |    37    |   8    |    12   |    12     |    0     |    2     |    0    |    0     |   200  |
|  4  |    17    |    53    |   23   |    13   |    13     |    0     |    4     |    10   |    0     |   150  |
|  1  |    12    |    64    |   35   |    17   |    17     |    0     |    9     |    0    |    0     |   50   |
|  6  |    2     |    81    |   59   |    20   |    20     |    0     |    6     |    0    |    0     |   150  |
|  7  |    19    |    103   |   67   |    17   |    17     |    0     |    1     |    5    |    0     |   200  |

[Gantt Chart]
process: |00|666|8888|33333|111|5|22222222|55555555555|444|111|4444444444|11111111111|66666666666666666|777777777777|00000|77777|
-----------------------------------------------------------------
```

[Preemptive Priority]

```
**Preemtive Priority**
[Process_Information]
| PID | arrive_t | finish_t | wait_t | burst_t | execute_t | remain_t | priority | io_start| io_remain| period |
|  8  |    5     |    9     |   0    |    4    |     4     |    0     |    3     |    0    |    0     |   200  |
|  7  |    19    |    41    |   5    |    17   |    17     |    0     |    1     |    5    |    0     |   200  |
|  5  |    17    |    46    |   17   |    12   |    12     |    0     |    2     |    0    |    0     |   200  |
|  4  |    17    |    62    |   32   |    13   |    13     |    0     |    4     |    10   |    0     |   150  |
|  6  |    2     |    68    |   46   |    20   |    20     |    0     |    6     |    0    |    0     |   150  |
|  2  |    18    |    76    |   50   |    8    |     8     |    0     |    7     |    0    |    0     |   50   |
|  3  |    9     |    81    |   67   |    5    |     5     |    0     |    8     |    0    |    0     |   50   |
|  1  |    12    |    98    |   69   |    17   |    17     |    0     |    9     |    0    |    0     |   50   |

[Gantt Chart]
process: |00|666|8888|66666666|55|777777777777|55555|77777|55555|444|666|4444444444|666666|22222222|33333|11111111111111111|
-----------------------------------------------------------------
```

[Preemptive Priority with Aging]



```
**Preemtive Priority(aging)**
[Process_Information]
| PID | arrive_t | finish_t | wait_t | burst_t | execute_t | remain_t | priority | io_start| io_remain| period |
|  8  |    5     |    9     |   0    |    4    |     4     |    0     |    3     |    0    |    0     |   200  |
|  5  |    17    |    41    |   12   |    12   |     12    |    0     |    0     |    0    |    0     |   200  |
|  7  |    19    |    53    |   17   |    17   |     17    |    0     |    -1    |    5    |    0     |   200  |
|  4  |    17    |    59    |   29   |    13   |     13    |    0     |    0     |    10   |    0     |   150  |
|  6  |    2     |    68    |   46   |    20   |     20    |    0     |    2     |    0    |    0     |   150  |
|  3  |    9     |    76    |   62   |    5    |     5     |    0     |    2     |    0    |    0     |   50   |
|  2  |    18    |    81    |   55   |    8    |     8     |    0     |    3     |    0    |    0     |   50   |
|  1  |    12    |    98    |   69   |    17   |     17    |    0     |    1     |    0    |    0     |   50   |

[Gantt Chart]
process: |00|666|8888|66666666|55|777777777777|5555555555|444|777|4444|77|444444|666666666|222|33333|22222|1111111111111111|
-----------------------------------------------------------------
```

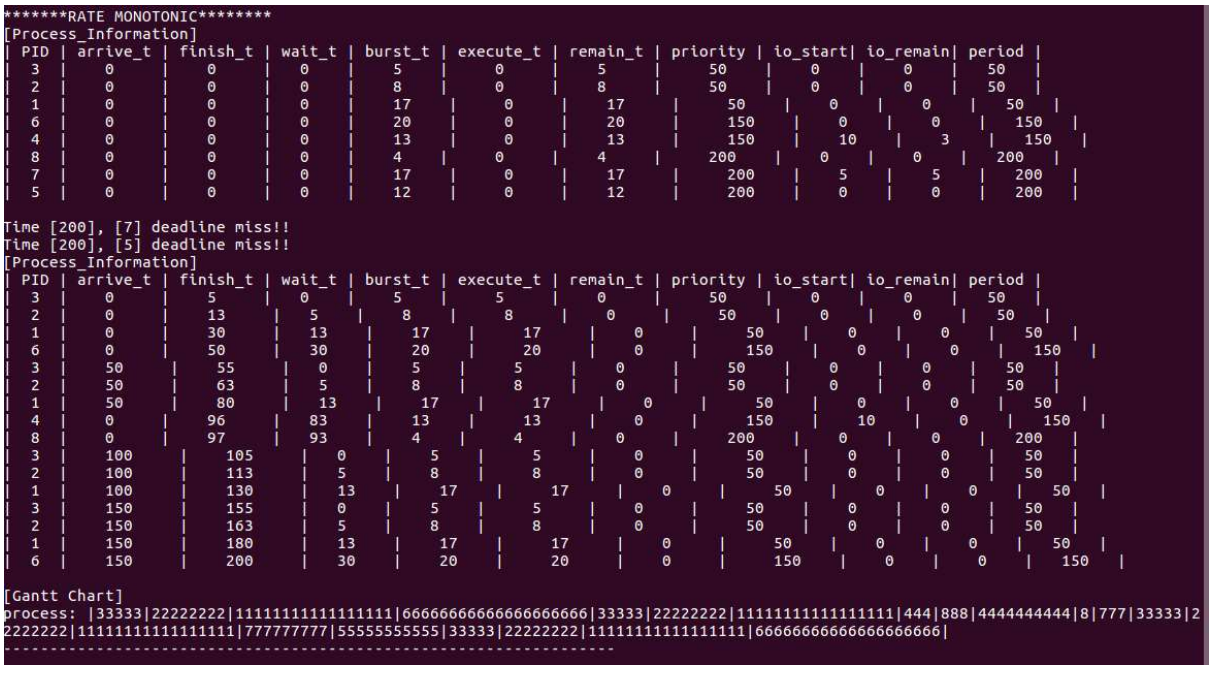
[Multilevel Queue]

```
**Multilevel Queue**
[RR]
front:0 rear:4 count:4
[Process_Information]
| PID | arrive_t | finish_t | wait_t | burst_t | execute_t | remain_t | priority | io_start| io_remain| period |
|  8  |    5     |    0     |   0    |    4    |     0     |    4     |    3     |    0    |    0     |   200  |
|  4  |    17    |    0     |   0    |    13   |     0     |    13    |    4     |    10   |    3     |   150  |
|  5  |    17    |    0     |   0    |    12   |     0     |    12    |    2     |    0    |    0     |   200  |
|  7  |    19    |    0     |   0    |    17   |     0     |    17    |    1     |    5    |    5     |   200  |

[FCFS]
front:0 rear:4 count:4
[Process_Information]
| PID | arrive_t | finish_t | wait_t | burst_t | execute_t | remain_t | priority | io_start| io_remain| period |
|  6  |    2     |    0     |   0    |    20   |     0     |    20    |    6     |    0    |    0     |   150  |
|  3  |    9     |    0     |   0    |    5    |     0     |    5     |    8     |    0    |    0     |   50   |
|  1  |    12    |    0     |   0    |    17   |     0     |    17    |    9     |    0    |    0     |   50   |
|  2  |    18    |    0     |   0    |    8    |     0     |    8     |    7     |    0    |    0     |   50   |

[Process_Information]
| PID | arrive_t | finish_t | wait_t | burst_t | execute_t | remain_t | priority | io_start| io_remain| period |
|  8  |    5     |    12    |   3    |    4    |     4     |    0     |    3     |    0    |    0     |   200  |
|  5  |    17    |    65    |   36   |    12   |     12    |    0     |    2     |    0    |    0     |   200  |
|  6  |    2     |    69    |   47   |    20   |     20    |    0     |    6     |    0    |    0     |   150  |
|  4  |    17    |    72    |   42   |    13   |     13    |    0     |    4     |    10   |    0     |   150  |
|  7  |    19    |    84    |   48   |    17   |     17    |    0     |    1     |    5    |    0     |   200  |
|  3  |    9     |    88    |   74   |    5    |     5     |    0     |    8     |    0    |    0     |   50   |
|  1  |    12    |    147   |   118  |    17   |     17    |    0     |    9     |    0    |    0     |   50   |
|  2  |    18    |    169   |   143  |    8    |     8     |    0     |    7     |    0    |    0     |   50   |

[Gantt Chart]
process: |00000|88|000|88|00000000|444|555|7|000|77|444|55|000|5|777|444|000|555|777|4|000|44|555|77|000|7|4|0000|7|000|7777|0000
0000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000|
-----------------------------------------------------------------
[Gantt Chart]
process: |0000000|666|0000000|666|0000000|666|0000000|666|0000000|666|0000000|666|0000000|66|3|0000000|333|0000000|3|11|0000000|1
11|0000000|111|0000000|111|0000000|111|0000000|111|0000000|222|0000000|222|0000000|22|
-----------------------------------------------------------------
```

[Multilevel Feedback Queue]

```
**Multilevel Feedback Queue**
[RR]
front:0 rear:4 count:4
[Process_Information]
| PID | arrive_t | finish_t | wait_t | burst_t | execute_t | remain_t | priority | io_start| io_remain| period |
|  8  |    5     |    0     |   0    |    4    |     0     |    4     |     3    |    0    |    0     |   200  |
|  4  |    17    |    0     |   0    |    13   |     0     |    13    |     4    |    10   |    3     |   150  |
|  5  |    17    |    0     |   0    |    12   |     0     |    12    |     2    |    0    |    0     |   200  |
|  7  |    19    |    0     |   0    |    17   |     0     |    17    |     1    |    5    |    5     |   200  |

[FCFS]
front:0 rear:4 count:4
[Process_Information]
| PID | arrive_t | finish_t | wait_t | burst_t | execute_t | remain_t | priority | io_start| io_remain| period |
|  6  |    2     |    0     |   0    |    20   |     0     |    20    |     6    |    0    |    0     |   150  |
|  3  |    9     |    0     |   0    |    5    |     0     |    5     |     8    |    0    |    0     |   50   |
|  1  |    12    |    0     |   0    |    17   |     0     |    17    |     9    |    0    |    0     |   50   |
|  2  |    18    |    0     |   0    |    8    |     0     |    8     |     7    |    0    |    0     |   50   |

[Process_Information]
| PID | arrive_t | finish_t | wait_t | burst_t | execute_t | remain_t | priority | io_start| io_remain| period |
|  8  |    5     |    12    |   3    |    4    |     4     |    0     |     3    |    0    |    0     |   200  |
|  3  |    9     |    61    |   47   |    5    |     5     |    0     |     5    |    0    |    0     |   50   |
|  6  |    2     |    69    |   47   |    20   |     20    |    0     |     6    |    0    |    0     |   150  |
|  5  |    17    |    73    |   44   |    12   |     12    |    0     |     2    |    0    |    0     |   200  |
|  4  |    17    |    80    |   50   |    13   |     13    |    0     |     4    |    10   |    0     |   150  |
|  7  |    19    |    86    |   50   |    17   |     17    |    0     |     1    |    5    |    0     |   200  |
|  2  |    18    |    111   |   85   |    8    |     8     |    0     |     4    |    0    |    0     |   50   |
|  1  |    12    |    128   |   99   |    17   |     17    |    0     |     6    |    0    |    0     |   50   |

[Gantt Chart]
process: |00000|88|000|88|00000000|444|555|7|000|77|444|55|000|5|333|777|000|444|555|3|000|3|777|444|000|555|777|4|0000|77777|000
00000000000|2222222|000|2|0000000000000000|
-----------------------------------------------------------------
[Gantt Chart]
process: |0000000|666|0000000|666|0000000|666|0000000|666|0000000|666|0000000|666|0000000|66|1|0000000|111|0000000|111|0000000|11
1|0000000|111|0000000|111|0000000|1|
-----------------------------------------------------------------
```

[Rate Monotonic]



```
*******RATE MONOTONIC********
[Process_Information]
| PID | arrive_t | finish_t | wait_t | burst_t | execute_t | remain_t | priority | io_start| io_remain| period |
|  3  |    0     |    0     |   0    |    5    |     0     |    5     |    50    |    0    |    0     |   50   |
|  2  |    0     |    0     |   0    |    8    |     0     |    8     |    50    |    0    |    0     |   50   |
|  1  |    0     |    0     |   0    |    17   |     0     |    17    |    50    |    0    |    0     |   50   |
|  6  |    0     |    0     |   0    |    20   |     0     |    20    |    150   |    0    |    0     |   150  |
|  4  |    0     |    0     |   0    |    13   |     0     |    13    |    150   |    10   |    3     |   150  |
|  8  |    0     |    0     |   0    |    4    |     0     |    4     |    200   |    0    |    0     |   200  |
|  7  |    0     |    0     |   0    |    17   |     0     |    17    |    200   |    5    |    5     |   200  |
|  5  |    0     |    0     |   0    |    12   |     0     |    12    |    200   |    0    |    0     |   200  |

Time [200], [7] deadline miss!!
Time [200], [5] deadline miss!!
[Process_Information]
| PID | arrive_t | finish_t | wait_t | burst_t | execute_t | remain_t | priority | io_start| io_remain| period |
|  3  |    0     |    5     |   0    |    5    |     5     |    0     |    50    |    0    |    0     |   50   |
|  2  |    0     |    13    |   5    |    8    |     8     |    0     |    50    |    0    |    0     |   50   |
|  1  |    0     |    30    |   13   |    17   |     17    |    0     |    50    |    0    |    0     |   50   |
|  6  |    0     |    50    |   30   |    20   |     20    |    0     |    150   |    0    |    0     |   150  |
|  3  |    50    |    55    |   0    |    5    |     5     |    0     |    50    |    0    |    0     |   50   |
|  2  |    50    |    63    |   5    |    8    |     8     |    0     |    50    |    0    |    0     |   50   |
|  1  |    50    |    80    |   13   |    17   |     17    |    0     |    50    |    0    |    0     |   50   |
|  4  |    0     |    96    |   83   |    13   |     13    |    0     |    150   |    10   |    0     |   150  |
|  8  |    0     |    97    |   93   |    4    |     4     |    0     |    200   |    0    |    0     |   200  |
|  3  |    100   |    105   |   0    |    5    |     5     |    0     |    50    |    0    |    0     |   50   |
|  2  |    100   |    113   |   5    |    8    |     8     |    0     |    50    |    0    |    0     |   50   |
|  1  |    100   |    130   |   13   |    17   |     17    |    0     |    50    |    0    |    0     |   50   |
|  3  |    150   |    155   |   0    |    5    |     5     |    0     |    50    |    0    |    0     |   50   |
|  2  |    150   |    163   |   5    |    8    |     8     |    0     |    50    |    0    |    0     |   50   |
|  1  |    150   |    180   |   13   |    17   |     17    |    0     |    50    |    0    |    0     |   50   |
|  6  |    150   |    200   |   30   |    20   |     20    |    0     |    150   |    0    |    0     |   150  |

[Gantt Chart]
process: |33333|22222222|1111111111111111|6666666666666666666|33333|22222222|1111111111111111|444|888|4444444444|8|777|33333|2
2222222|1111111111111111|777777777|55555555555|33333|22222222|11111111111111111|66666666666666666666|
-----------------------------------------------------------------
```

Deadline을 Miss한 것을 확인 할 수 있다.

[Earliest Deadline First]

```
**EARLIEST DEADLINE FIRST****
[Process_Information]
| PID | arrive_t | finish_t | wait_t | burst_t | execute_t | remain_t | priority | io_start| io_remain| period |
|  3  |    0     |    0     |   0    |    5    |     0     |    5     |    50    |    0    |    0     |   50   |
|  2  |    0     |    0     |   0    |    8    |     0     |    8     |    50    |    0    |    0     |   50   |
|  1  |    0     |    0     |   0    |   17    |     0     |   17     |    50    |    0    |    0     |   50   |
|  6  |    0     |    0     |   0    |   20    |     0     |   20     |   150    |    0    |    0     |  150   |
|  4  |    0     |    0     |   0    |   13    |     0     |   13     |   150    |   10    |    3     |  150   |
|  8  |    0     |    0     |   0    |    4    |     0     |    4     |   200    |    0    |    0     |  200   |
|  7  |    0     |    0     |   0    |   17    |     0     |   17     |   200    |    5    |    5     |  200   |
|  5  |    0     |    0     |   0    |   12    |     0     |   12     |   200    |    0    |    0     |  200   |

[Process_Information]
| PID | arrive_t | finish_t | wait_t | burst_t | execute_t | remain_t | priority | io_start| io_remain| period |
|  3  |    0     |    5     |   0    |    5    |     5     |    0     |    45    |    0    |    0     |   50   |
|  2  |    0     |   13     |   5    |    8    |     8     |    0     |    37    |    0    |    0     |   50   |
|  1  |    0     |   30     |  13    |   17    |    17     |    0     |    20    |    0    |    0     |   50   |
|  6  |    0     |   50     |  30    |   20    |    20     |    0     |   100    |    0    |    0     |  150   |
|  3  |   50     |   55     |   0    |    5    |     5     |    0     |    45    |    0    |    0     |   50   |
|  2  |   50     |   63     |   5    |    8    |     8     |    0     |    37    |    0    |    0     |   50   |
|  1  |   50     |   80     |  13    |   17    |    17     |    0     |    20    |    0    |    0     |   50   |
|  4  |    0     |   96     |  83    |   13    |    13     |    0     |    54    |   10    |    0     |  150   |
|  8  |    0     |   97     |  93    |    4    |     4     |    0     |   103    |    0    |    0     |  200   |
|  3  |  100     |  105     |   0    |    5    |     5     |    0     |    45    |    0    |    0     |   50   |
|  2  |  100     |  113     |   5    |    8    |     8     |    0     |    37    |    0    |    0     |   50   |
|  1  |  100     |  130     |  13    |   17    |    17     |    0     |    20    |    0    |    0     |   50   |
|  5  |    0     |  151     |  139   |   12    |    12     |    0     |    49    |    0    |    0     |  200   |
|  7  |    0     |  156     |  139   |   17    |    17     |    0     |    44    |    5    |    0     |  200   |
|  3  |  150     |  161     |   6    |    5    |     5     |    0     |    39    |    0    |    0     |   50   |
|  2  |  150     |  169     |  11    |    8    |     8     |    0     |    31    |    0    |    0     |   50   |
|  1  |  150     |  186     |  19    |   17    |    17     |    0     |    14    |    0    |    0     |   50   |

[Gantt Chart]
process: |33333|22222222|1111111111111111|66666666666666666|33333|22222222|1111111111111111|444|888|4444444444|8|777|33333|2
2222222|1111111111111111|777777777|555555555555|77777|33333|22222222|1111111111111111|66666666666666|
-----------------------------------------------------------------
```

Waiting Time은 Rate Monotonic에 비해 증가하지만, deadline miss가 발생 하지 않았다.

5. CPU Scheduling 알고리즘 평가

*****Summary of all Results*****

Algorithm	Process Num	Context_Switch
FCFS	8	10
Nonpreemptive SJF	8	10
Nonpreemptive Priority	8	10
Round Robin	8	35
Preemptive SJF	8	14
Preemptive Priority	8	15
Pre_aging Priority	8	16
Multilevel Queue	8	21
Multilevel Feedback	8	24
Rate Monotonic	8	22
Earliest Deadline	8	23

Algorithm	Sum of Waiting Time	Average Waiting Time
FCFS	257	32.125
Nonpreemptive SJF	229	28.625
Nonpreemptive Priority	300	37.500
Round Robin	355	44.375
Preemptive SJF	192	24.000
Preemptive Priority	286	35.750
Pre_aging Priority	290	36.250
Multilevel Queue	511	63.875
Multilevel Feedback	425	53.125
Rate Monotonic	308	38.500
Earliest Deadline	574	71.750

Algorithm	Sum of Burst Time	Average Burst Time
FCFS	96	12.000
Nonpreemptive SJF	96	12.000
Nonpreemptive Priority	96	12.000
Round Robin	96	12.000
Preemptive SJF	96	12.000
Preemptive Priority	96	12.000
Pre_aging Priority	96	12.000
Multilevel Queue	96	12.000
Multilevel Feedback	96	12.000
Rate Monotonic	177	22.125
Earliest Deadline	186	23.250

Algorithm	Sum of Turnaround Time	Average Turnaround Time
FCFS	353	44.125
Nonpreemptive SJF	325	40.625
Nonpreemptive Priority	396	49.500
Round Robin	451	56.375
Preemptive SJF	288	36.000
Preemptive Priority	382	47.750
Pre_aging Priority	386	48.250
Multilevel Queue	607	75.875
Multilevel Feedback	521	65.125
Rate Monotonic	485	60.625
Earliest Deadline	760	95.000

동일 작업임에도 불구하고 Waiting Time과 Turnaround Time이 스케줄링 알고리즘 별로 몹시 상이하게 나왔다. 프로세스의 burst_Time을 알고 수행한다는 Shortest Job First는 사실상 이상적인 이론이기 때문에 가장 적은 Waiting Time과 Turnaround Time을 보여준다. Round Robin은 프로세스들의 responsivness를 높이는데 주력하기 때문에, Turnaround와 Waiting모두 몹시 높게 나왔으며 Context_Switch 또한 몹시 많이 발생하였다. Multilevel Queue에 비해 Multilevel Feedback Queue의 Waiting Time이 적게 나왔는데, 아래 레벨의 큐에서 오랫동안 기다리는 프로세스를 위의 레벨로 상승시켜 주었기 때문에 나타난 결과였다. Rate Monotonic에 비하여 Earliest Deadline First의 Waiting Time과 Turnaround Time이 훨씬 크게 나타났지만 Rate Monotonic과는 다르게 deadline을 지킬 수가 있었다.

3장 결론

1. Simulator 평가 및 개선 방향

정리하면 도입부에서는 CPU의 상태 cycle을 정의하고, CPU utilization을 확대시키기 위한 CPU 스케줄러를 필요성을 논의했다. 이후 본론에서는 각각의 CPU 스케줄링 알고리즘을 설명한 후 구현한 simulator를 바탕으로 CPU 스케줄링 간 알고리즘 성능 비교까지 진행하였다. 11가지에 달하는 다양한 알고리즘들을 비교함으로서 효용성이 높다고 생각한다.

예상과 같이, Shortest Job First는 압도적으로 우월한 성능을 보여주었다. 하지만, CPU burst time을 우리가 확실하게 알지 못하기 때문에 실제로는 예측을 해야하는데 주로 exponential averaging 기법을 사용한다. Exponential averaging기법을 통해 들어오는 프로세스들의 cpu burst time을 예측해보고 이를 기반으로 스케줄링하여 비교 분석 하는 것이 더 합리적일 것이다.

또한, Context Switch Time을 횟수로만 남겨두었는데, 실제 Context Switch Time의 값을 집어넣어 성능 비교에 반영하는 것도 좋은 개선 방향일 것이다.

프로세스들을 랜덤하게 생성되게 하였는데, 사용자가 임의의 원하는 값으로 프로세스들을 생성하고 이를 통해 테스트할 환경을 구축하면 더 좋을 것이다.

2. 프로젝트 수행 소감

이번 term project를 통해 막연히만 여겼던 CPU 스케줄링 알고리즘을 보다 확실히 이해할 수 있었다. OS라는 것이 단순히 CPU와 메모리의 연결고리 역할만 한다고 생각하였는데, 프로세스들을 효과적으로 할당하기 위한 다양한 알고리즘들을 활용한다는 것을 알게되고, 또 이를 구현하며 컴퓨터의 시스템을 보다 잘 이해하게 되었다.

효율적인 공간 활용을 위해 큐를 원형큐로 사용하였는데, 오랜만에 접한 C의 포인터에서 계속 발생되는 메모리 오류로 인해 상당히 고생하였다. 이를 해결하기 위해 수 많은 디버깅을 하고 해결해 내어서 상당한 뿌듯함을 느꼈다.

『참고문헌 및 사이트』

1) Abraham Silberschatz, Peter B.Galvin, Greg Gagne, 『OPERAING SYSTEM CONCEPTS_Ninth Edition』, WILEY, 2013

2) Operating System PDF(Chapter 5: CPU Scheduling),(Chapter 3: Processes), 유헌창 교수님

<부록> - 소스 코드

```
#include <stdlib.h>

#include <stdio.h>

#include <unistd.h>

#include <time.h>

#include <string.h>


#define DELAY 1

#define MAX_PROCESS 10

#define MAX_ARRIVE_TIME 20

#define MAX_BURST_TIME 20

#define MAX_IO_TIME 5

#define IO_FREQ 3

#define MAX_PERIOD 200


int Time;            // 시간 경과 표시

int Gantt[1000]; // 간트 차트

int Gantt2[1000]; // 간트 차트 for multilevelQueue


int Context_switch= 0;
```

```
typedef struct{
        int pid;
        int arrive_time;
        int waiting_time;
        int execute_time; //실행된 시간
        int burst_time;       //CPU burst time
        int remain_time;      //Remaining Burst time
        int finish_time;
        int io_start_time;   //=remain_cpu_time
        int io_remain_time; //I/O연산이 발생할 경우 할당되는 시
        int priority;
        int period;
}PCB;

typedef struct{
        int process;
        int context_switch;
        int sum_wait;
        int sum_burst;
        int sum_turn;
        int io_count;
        float avg_wait;
        float avg_burst;
        float avg_turn;
}RESULT;
```

```
typedef struct{
	int front;
	int rear;
	int size;
	int count;
	PCB*buffer;
}Queue;



PCB running_Q;	// Process on Cpu
RESULT result[10]; // Queue의 결과들을 저장해논 배


/*Get Process Number from User*/
int process_num(){
	int p_num = 0;
	while (1){
		printf("프로세스 갯수 입력(1-%d) : ", MAX_PROCESS);
		scanf("%d", &p_num);
		if (1 <=p_num && p_num <= MAX_PROCESS)
			break;
		else
			printf("1이상 %d 이하의 정수를 입력해주세요.₩n", MAX_PROCESS);
	}
	return p_num;
}
```

```
/* Initialize PCB */
PCB init_PCB(){
        PCB pcb;
        pcb.pid = 0;
        pcb.arrive_time = 0;
        pcb.execute_time = 0;
        pcb.priority = 0;
        pcb.remain_time = 0;
        pcb.waiting_time = 0;
        pcb.io_start_time = 0;
        pcb.io_remain_time = 0;
        pcb.period = 0;
        return pcb;
}

/*Initialize Circular Queue with designated process_number */
void init_Queue(Queue *queue,int p_num){
        p_num += 1;    //n+1 size의 queue에 n개의 process가 들어갈 수 있다.
        queue->buffer = (PCB*)malloc(p_num*sizeof(PCB));
        memset(queue->buffer, 0, p_num * sizeof(PCB));
        queue->size = p_num;
        queue->front = 0;
        queue->rear = 0;
        queue->count = 0;
}
```

```
/*Get process in head of the Queue*/
PCB get_Queue_head(Queue * queue){ //큐 첫번째 항목 반환;
	return queue->buffer[(queue->front+1) % queue->size];
}

/* return True if Queue is full */
int is_Queue_full(Queue *queue){
	if (queue->front == (queue->rear+1)%queue->size)
		return 1;
	else
		return 0;
}

/* Dequeue */
void dequeue(Queue *queue){
	if(queue->count == 0){
		printf("dequeue_큐가 비어있습니다!₩n");
		return;
	}
	queue->count -=1;
	queue->buffer[queue->front] = init_PCB();
	queue->front = (queue->front+1)%queue->size;
}

/* Enqueue: rear를 증가시킨 후 그 자리에 process삽입 */
```

```
void enqueue(Queue *queue, PCB process){
        if((queue->rear+1)%queue->size == queue->front){
                printf("enqueue_큐가 꽉차있습니다!₩n");
                exit(-1);
        }
        queue->count += 1;
        queue->rear = (queue->rear+1) % queue->size;
        queue->buffer[queue->rear] = process;
}

/* Get number of process in Queue */
int get_Queue_count(Queue *queue){
        return queue->count;
}

/* Copy queue to copy_Q */
void copy_Queue(Queue*queue, Queue*copy_Q){
        copy_Q->front = queue->front;
        copy_Q->rear  = queue->rear;
        copy_Q->count = queue->count;
        copy_Q->size  = queue->size;

        for(int i=0; i<queue->size; i++){
                copy_Q->buffer[i] = queue->buffer[i];
        }
}
```

```
/* ready_Q -> running_Q
        running_Q의 작업이 끝나지 않았다면 ready_Q로 */
PCB dispatcher(Queue*ready_Q, PCB running_Q){
        if(running_Q.remain_time != 0){ //작업이 아직 끝나지 않음
                enqueue(ready_Q,running_Q);
        }
        running_Q = get_Queue_head(ready_Q);
        Context_switch += 1;
        return running_Q;
}


/*Sort Queue by arrival time ->for FCFS */
void sort_by_arrival(Queue *queue){
        int i,j,minindex;
        PCB min;
        PCB temp;
        for(int i=(queue->front + 1)%queue->size; (i % queue->size)!=((queue->rear) % queue->size); i++){
                i = i% queue->size;
                minindex = i;
                min = queue->buffer[i];
                for(int j=(i+1)% queue->size; (j % queue->size)!=((queue->rear+1) % queue->size); j++){
```

```
                j = j % queue->size;
                        temp = queue->buffer[j];
                        if(min.arrive_time > temp.arrive_time ||
                                        (min.arrive_time == temp.arrive_time && min.pid > temp.pid)){
                                minindex = j;
                                min = temp;
                        }
                }
                queue->buffer[minindex] = queue->buffer[i];
                queue->buffer[i] = min;
        }
}

/*Sort Queue by remaining time ->for SJF */
void sort_by_remain(Queue *queue){
        int i,j,minindex;
        PCB min;
        PCB temp;
        for(int i=(queue->front + 1)%queue->size; (i % queue->size)!=((queue->rear) % queue->size); i++){
                i = i % queue->size;
                minindex = i;
                min = queue->buffer[i];
                for(int j=(i+1)% queue->size; (j % queue->size)!=((queue->rear+1) % queue->size); j++){
```

```
                        j = j % queue->size;
                        temp = queue->buffer[j];
                        if(min.remain_time > temp.remain_time ||
                                        (min.remain_time == temp.remain_time && min.pid > temp.pid)){
                                minindex = j;
                                min = temp;
                        }
                }
                queue->buffer[minindex] = queue->buffer[i];
                queue->buffer[i] = min;
        }
}

/*Sort Queue by priority time ->for priority first */
void sort_by_priority(Queue *queue){
        int i,j,minindex;
        PCB min;
        PCB temp;
        for(int i=(queue->front + 1)%queue->size; (i % queue->size)!=((queue->rear) % queue->size); i++){
                i = i % queue->size;
                minindex = i;
                min = queue->buffer[i];
                for(int j=(i+1)% queue->size; (j % queue->size)!=((queue->rear+1) % queue->size); j++){
```

```
                        j = j % queue->size;
                        temp = queue->buffer[j];
                        if(min.priority > temp.priority ||
                                (min.priority == temp.priority && min.pid < temp.pid)){
                                minindex = j;
                                min = temp;
                        }
                }
                queue->buffer[minindex] = queue->buffer[i];
                queue->buffer[i] = min;
        }
}

/*random_process()의 random한 priority를 생성해준다*/
int* random_priority_array(int* priority_arr,int size){
        int temp;
        int mix1, mix2;
        int mix_num = 50;
        srand(time(NULL));

        for(int i=0; i<size; i++) //initialize array
                priority_arr[i] = i+1;
        for(int i=0; i<mix_num; i++){ //mix_num만큼 swap
                mix1 = rand() % size;
                mix2 = rand() % size;
                temp = priority_arr[mix1];
```

```
            priority_arr[mix1] = priority_arr[mix2];
            priority_arr[mix2] = temp;
        }
        return priority_arr;
}

/*Initialize가 된 Queue에 대해서 size크기 만큼 random한 process생성*/
void fill_random_process(Queue *queue){
        int temp_p_arr[100];
        int *priority_arr = random_priority_array(temp_p_arr,queue->size);
        srand(time(NULL));
        for(int i=queue->front + 1; i< queue->size; i++){
            queue->buffer[i].pid = i;
            queue->buffer[i].arrive_time = rand() % MAX_ARRIVE_TIME; //
0~MAX_ARRIVE_TIME-1
            queue->buffer[i].burst_time = rand() % MAX_BURST_TIME + 1; //
1~MAX_BURST_TIME
            queue->buffer[i].waiting_time = 0;
            queue->buffer[i].execute_time = 0;
            queue->buffer[i].finish_time = 0;
            queue->buffer[i].remain_time = queue->buffer[i].burst_time; //initialize with
burst_time
            queue->buffer[i].priority = priority_arr[i]; //random priority
            queue->buffer[i].period = ((rand() % 4 +1) * MAX_PERIOD) /4; // 25,50,75,100
            if(rand() % IO_FREQ == 0 && queue->buffer[i].remain_time>1){  //IO_FREQ꼴에
한번 씩 IO연산 할당
                queue->buffer[i].io_remain_time = rand() % MAX_IO_TIME +1;  //1 ~
```

```
MAX_IO_TIME
                        queue->buffer[i].io_start_time = rand() % queue->buffer[i].remain_time;
                        if (queue->buffer[i].io_start_time == 0)
                                queue->buffer[i].io_start_time = 1;
                }
                queue->count += 1;
                queue->rear += 1;
        }
        queue->rear = queue->rear % queue->size;
}

/*Print all the Processes existing in the Queue*/
void print_Queue(Queue*queue){
        PCB temp;

        printf("[Process_Information]₩n");
        printf("| PID | arrive_t | finish_t | wait_t | burst_t | execute_t | remain_t | priority | io_start| io_remain| period |₩n");

        for(int i=(queue->front + 1); (i % queue->size)!=((queue->rear+1) % queue->size); i++){
                i = i % queue->size;
                temp = queue->buffer[i];
                printf("|   %d   |      %d       |      %d       |     %d      |      %d      |       %d       |      %d      |      %d      |     %d     |₩n"

        ,temp.pid,temp.arrive_time,temp.finish_time,temp.waiting_time,temp.burst_time,temp.execute_time
```

```
                    ,temp.remain_time,  temp.priority,temp.io_start_time,temp.io_remain_time,
temp.period);
        }
        printf("₩n");
}

/*Print process in running Queue*/
void print_running_Q(PCB running_Q){
        printf("₩nProcess_Information₩n");
        printf("| PID | arrive_t | wait_t | burst_t | execute_t | remain_t | priority | io_start|
io_remain|₩n");

        printf("| %d | %d | %d | %d | %d | %d | %d
| %d | %d | ₩n"

        ,running_Q.pid,running_Q.arrive_time,running_Q.waiting_time,running_Q.burst_time
                    ,running_Q.execute_time,running_Q.remain_time,running_Q.priority
                    ,running_Q.io_start_time,running_Q.io_remain_time);
}

void init_Gantt(){
        for(int i=0; i<1000; i++)
                Gantt[i] = 0;
}
/*Print Gannt Chart until given time*/
void print_Gantt(int time){
        printf("[Gantt Chart]₩n");
```

```
	printf("process: ");
	if(Gantt[0] == 0)
		printf("|");
	for(int i=0; i < time; i++){
		if(Gantt[i] != Gantt[i-1]){
			if(Gantt[i] >= 10)
				printf("|%d|",Gantt[i]);
			else
				printf("|%d",Gantt[i]);
		}
		else{
			if(Gantt[i] >= 10)
				printf("%d|",Gantt[i]);
			else
				printf("%d",Gantt[i]);
		}
	}
	printf("|₩n");
	printf("----------------------------------------------------------------₩n");
}

void print_algorithm(int a){
	if(a == 0)
		printf("|		FCFS		|");
	else if(a == 1)
		printf("|	Nonpreemptive SJF	|");
```

```
        else if(a == 2)
                printf("|Nonpreemptive Priority|");
        else if(a == 3)
                printf("|          Round Robin        |");
        else if(a == 4)
                printf("|      Preemptive SJF       |");
        else if(a == 5)
                printf("|   Preemptive Priority |");
        else if(a == 6)
                printf("|   Pre_aging Priority   |");
        else if(a == 7)
                printf("|     Multilevel Queue     |");
        else if(a == 8)
                printf("|   Multilevel Feedback |");
        else if(a == 9)
                printf("|        Rate Monotonic     |");
        else if(a == 10)
                printf("|     Earliest Deadline   |");

}

/*Print Result of queue (wait, burst등의 연산이 완료 되어 있어야함!)  a = algorithm_num   */
void print_result(Queue *queue, int a){
        int sum_wait =0, sum_burst = 0, sum_turn = 0;
  //queue->front+1에서 queue->rear까지 출력
        for(int i=(queue->front + 1); (i % queue->size)!=((queue->rear+1) % queue->size); i++){
```

```
                sum_wait    = sum_wait    + queue->buffer[i].waiting_time;
        }
        printf("₩n*****RESULT*****₩n");
        printf("|        Algorithm        |    Number of Process    |    Number of Context Switch    |₩n");
        print_algorithm(a);
        printf("          %8d            | %15d                  | ₩n₩n",queue->count, Context_switch);
        printf("| Sum of Waiting time | Average Waiting time |₩n");
        printf("|                %d                |                %.3f                |₩n",sum_wait,
(float)sum_wait/queue->count);
        printf("|    Sum of Burst time    |    Average Burst time    |₩n");
        printf("|                %d                |                %.3f                |₩n",sum_burst,
(float)sum_burst/queue->count);
        printf("|    Sum of Turn Time    |    Average Turn Time    |₩n");
        printf("|                %d                |                %.3f                |₩n",sum_turn,
(float)sum_turn/queue->count);
}

void initialize_result(){
        for(int i= 0; i<15; i++){
                result[i].sum_wait = 0;
                result[i].sum_burst = 0;
                result[i].sum_turn = 0;
        }
}

//Result에 저장된 알고리즘별 최종 값들을 출력한다.
```

```
void print_all_result(){
	printf("₩n*****Summary of all Results*****₩n");
	printf("-------------------------------------------------------------------------₩n");
	printf("|	Algorithm	|	Process Num	|	Context_Switch	|₩n");
	printf("-------------------------------------------------------------------------₩n");
		for(int i=0; i<11; i++){
			print_algorithm(i);
			printf("	%d	|	%d	| ₩n",result[i].process, result[i].context_switch);
		}
	printf("-------------------------------------------------------------------------₩n");
	printf("|	Algorithm	| Sum of Waiting Time	| Average Waiting Time |₩n");
	printf("-------------------------------------------------------------------------₩n");
		for(int i=0; i<11; i++){
			print_algorithm(i);
			printf("	%d	|	%.3f |₩n",result[i].sum_wait, result[i].avg_wait);
		}

	printf("-------------------------------------------------------------------------₩n");
	printf("|	Algorithm	|	Sum of Burst Time	| Average Burst Time |₩n");
	printf("-------------------------------------------------------------------------₩n");
		for(int i=0; i<11; i++){
			print_algorithm(i);
			printf("	%d	|	%.3f |₩n",result[i].sum_burst, result[i].avg_burst);
```

```
		}

	printf("-------------------------------------------------------------------------₩n");
	printf("	Algorithm	|Sum of Turnaround Time|Average Turnaround Time|₩n");
	printf("-------------------------------------------------------------------------₩n");
		for(int i=0; i<11; i++){
			print_algorithm(i);
			printf("		%d		|		%.3f
|₩n",result[i].sum_turn, result[i].avg_turn);
		}
}

void FCFS(Queue *queue){
	printf("₩n*****FCFS*****₩n");
	Context_switch = 0;
	Time = 0;
	int io_count =0;

	init_Gantt();
	Queue FCFS;	//생성된 프로세스들을 저장한 queue를 받아서 저장한다.
	init_Queue(&FCFS, queue->size-1);
	copy_Queue(queue, &FCFS);

	Queue ready_Q, waiting_Q, terminated_Q;
	init_Queue(&ready_Q, queue->size-1);
	init_Queue(&waiting_Q, queue->size-1);
```

```
init_Queue(&terminated_Q, queue->size-1);


PCB running_Q;
running_Q = init_PCB();


while(is_Queue_full(&terminated_Q) != 1){ //terminated_Q가 꽉찰때 까지 time을 1씩 늘려가며 반복
        if (Time == 200){
                printf("[Ready_Q]₩n");
                printf("front:%d, rear:%d, count:%d₩n",ready_Q.front,ready_Q.rear,ready_Q.count);
                print_Queue(&ready_Q);
                printf("[Waiting_Q]₩n");
                print_Queue(&waiting_Q);
                break;
        }


        for(int i=FCFS.front+1; FCFS.buffer[i].pid!=0; i++){ //TIME에 도달할 때 ready_Q에 들어간다
                if(FCFS.buffer[i].arrive_time != Time)
                        break;
                else{
                        enqueue(&ready_Q, get_Queue_head(&FCFS));
                        dequeue(&FCFS);
                }
```

```
		}
		//sort_by_arrival(&ready_Q);

		if(waiting_Q.buffer[waiting_Q.front+1].pid!=0){ //waiting_Q에 프로세스가 존재한다면 io_time을 1줄인다.
			waiting_Q.buffer[waiting_Q.front+1].io_remain_time -= 1;

			if(waiting_Q.buffer[waiting_Q.front+1].io_remain_time ==0){ //I.O 입출력이 끝난다면, ready_q로 보낸다
				if (waiting_Q.buffer[waiting_Q.front+1].remain_time ==0){
					//waiting_Q.buffer[waiting_Q.front+1].arrive_time = Time;
					waiting_Q.buffer[waiting_Q.front+1].finish_time = Time;
					enqueue(&terminated_Q, waiting_Q.buffer[waiting_Q.front+1]);
					dequeue(&waiting_Q);
				}
				else{
					//waiting_Q.buffer[waiting_Q.front+1].arrive_time = Time; //Time으로 도착시간을 변경한뒤 enqueue
					enqueue(&ready_Q, waiting_Q.buffer[waiting_Q.front+1]);
					dequeue(&waiting_Q);
					}
			}
		}
		//sort_by_arrival(&ready_Q);
```

```
if(running_Q.pid !=0){ //run_Q존재, ready_q도 존재
    running_Q.remain_time -= 1;
    running_Q.execute_time += 1;

    if(running_Q.remain_time <= 0 & running_Q.io_remain_time == 0){ //no more burst and no more i/o
        running_Q.finish_time = Time;
        enqueue(&terminated_Q,running_Q);
        running_Q = init_PCB();
    }

    if(running_Q.io_remain_time !=0 &&
        (running_Q.io_start_time == running_Q.remain_time)){ //만약 IO연산을 수행해야 한다면
        io_count += 1;
        enqueue(&waiting_Q, running_Q); //waiting_Q로 이동시킨다.
        running_Q = init_PCB();          //run_Q를 초기화
    }
}

if(running_Q.pid == 0 && get_Queue_head(&ready_Q).pid!=0){ //run_Q empty, is ready_Q : ready->run
    running_Q = get_Queue_head(&ready_Q);
    dequeue(&ready_Q);
    Context_switch+=1;
```

```
                }

                Gantt[Time] = running_Q.pid;
                Time += 1;
        }
        for(int i=terminated_Q.front+1; i<= terminated_Q.rear; i++){
                PCB temp;
                temp = terminated_Q.buffer[i];
                terminated_Q.buffer[i].waiting_time = temp.finish_time - temp.execute_time - temp.arrive_time;
        }
        print_Queue(&terminated_Q);
        print_Gantt(Time-1);

        result[0].process = queue->size-1;
        result[0].context_switch = Context_switch;
        for(int i=(terminated_Q.front+1); i<=terminated_Q.rear; i++){
                result[0].sum_wait += terminated_Q.buffer[i].waiting_time;
                result[0].sum_burst += terminated_Q.buffer[i].burst_time;
                result[0].sum_turn += terminated_Q.buffer[i].waiting_time + terminated_Q.buffer[i].burst_time;
        }
        result[0].avg_wait = (float)result[0].sum_wait / result[0].process;
        result[0].avg_burst = (float)result[0].sum_burst / result[0].process;
        result[0].avg_turn = (float)result[0].sum_turn / result[0].process;
        result[0].io_count = io_count;
```

```
}


void Nonpreemptive_SJF(Queue *queue){
        printf("₩n**Nonpreemtive SJF**₩n");
        Context_switch = 0;
        Time = 0;
        int io_count =0;

        init_Gantt();
        Queue SJF;      //생성된 프로세스들을 저장한 queue를 받아서 저장한다.
        init_Queue(&SJF, queue->size-1);
        copy_Queue(queue, &SJF);

        Queue ready_Q, waiting_Q, terminated_Q;
        init_Queue(&ready_Q, queue->size-1);
        init_Queue(&waiting_Q, queue->size-1);
        init_Queue(&terminated_Q, queue->size-1);

        PCB running_Q;
        running_Q = init_PCB();

        sort_by_arrival(&SJF); // SJF에 있는 프로세스들을 도착시간 순으로 정렬한다

        while(is_Queue_full(&terminated_Q) != 1){ //terminated_Q가 꽉찰때 까지 time을 1씩 늘려
가며 반복
```

```
for(int i=SJF.front+1; SJF.buffer[i].pid!=0; i++){ //TIME에 도달할 때 ready_Q에 들어간다
    if(SJF.buffer[i].arrive_time != Time)
        break;
    else{
        enqueue(&ready_Q, get_Queue_head(&SJF));
        dequeue(&SJF);
    }
}

if(waiting_Q.buffer[waiting_Q.front+1].pid!=0){ //waiting_Q에 프로세스가 존재한다면 io_time을 1줄인다.
    waiting_Q.buffer[waiting_Q.front+1].io_remain_time -= 1;

    if(waiting_Q.buffer[waiting_Q.front+1].io_remain_time ==0){ //I.O 입출력이 끝난다면, ready_q로 보낸다
        if (waiting_Q.buffer[waiting_Q.front+1].remain_time ==0){
            waiting_Q.buffer[waiting_Q.front+1].finish_time = Time;
            enqueue(&terminated_Q, waiting_Q.buffer[waiting_Q.front+1]);
            dequeue(&waiting_Q);
        }
        else{
            enqueue(&ready_Q, waiting_Q.buffer[waiting_Q.front+1]);
            dequeue(&waiting_Q);
```

```
                    }
                }
            }
            sort_by_remain(&ready_Q);

            if(running_Q.pid !=0){ //run_Q존재, ready_q도 존재
                running_Q.remain_time -= 1;
                running_Q.execute_time += 1;

                if(running_Q.remain_time <= 0 & running_Q.io_remain_time == 0){ //no more burst and no more i/o
                    running_Q.finish_time = Time;
                    enqueue(&terminated_Q,running_Q);
                    running_Q = init_PCB();
                }

                if(running_Q.io_remain_time !=0 &&
                        (running_Q.io_start_time == running_Q.remain_time)){ //만약 IO연산을 수행해야 한다면
                    io_count += 1;
                    enqueue(&waiting_Q, running_Q); //waiting_Q로 이동시킨다.
                    running_Q = init_PCB();          //run_Q를 초기화
                }
            }

            if(running_Q.pid == 0 && get_Queue_head(&ready_Q).pid!=0){ //run_Q empty, is
```

```
ready_Q : ready->run
                        running_Q = get_Queue_head(&ready_Q);
                        dequeue(&ready_Q);
                        Context_switch+=1;
                }

                Gantt[Time] = running_Q.pid;
                Time += 1;
        }
        for(int i=terminated_Q.front+1; i<= terminated_Q.rear; i++){
                PCB temp;
                temp = terminated_Q.buffer[i];
                terminated_Q.buffer[i].waiting_time = temp.finish_time - temp.execute_time - temp.arrive_time;
        }
        print_Queue(&terminated_Q);
        print_Gantt(Time-1);

        result[1].process = queue->size-1;
        result[1].context_switch = Context_switch;
        for(int i=(terminated_Q.front+1); i<=terminated_Q.rear; i++){
                result[1].sum_wait += terminated_Q.buffer[i].waiting_time;
                result[1].sum_burst += terminated_Q.buffer[i].burst_time;
                result[1].sum_turn += terminated_Q.buffer[i].waiting_time + terminated_Q.buffer[i].burst_time;
        }
```

```
        result[1].avg_wait = (float)result[1].sum_wait / result[1].process;
        result[1].avg_burst = (float)result[1].sum_burst / result[1].process;
        result[1].avg_turn = (float)result[1].sum_turn / result[1].process;
        result[1].io_count = io_count;
}


void Nonpreemptive_Priority(Queue *queue){
        printf("₩n**Nonpreemtive Priority**₩n");
        Context_switch = 0;
        Time = 0;
        int io_count =0;

        init_Gantt();
        Queue PRIORITY;      //생성된 프로세스들을 저장한 queue를 받아서 저장한다.
        init_Queue(&PRIORITY,queue->size-1);
        copy_Queue(queue, &PRIORITY);

        Queue ready_Q, waiting_Q, terminated_Q;
        init_Queue(&ready_Q, queue->size-1);
        init_Queue(&waiting_Q, queue->size-1);
        init_Queue(&terminated_Q, queue->size-1);

        PCB running_Q;
        running_Q = init_PCB();
```

sort_by_arrival(&PRIORITY); // SJF에 있는 프로세스들을 도착시간 순으로 정렬한다

while(is_Queue_full(&terminated_Q) != 1){ //terminated_Q가 꽉찰때 까지 time을 1씩 늘려가며 반복

for(int i=PRIORITY.front+1; PRIORITY.buffer[i].pid!=0; i++){ //TIME에 도달할 때 ready_Q에 들어간다

if(PRIORITY.buffer[i].arrive_time != Time)

break;

else{

enqueue(&ready_Q, get_Queue_head(&PRIORITY));

dequeue(&PRIORITY);

}

}

sort_by_priority(&ready_Q);

if(waiting_Q.buffer[waiting_Q.front+1].pid!=0){ //waiting_Q에 프로세스가 존재한다면 io_time을 1줄인다.

waiting_Q.buffer[waiting_Q.front+1].io_remain_time -= 1;

if(waiting_Q.buffer[waiting_Q.front+1].io_remain_time ==0){ //I.O 입출력이 끝난다면, ready_q로 보낸다

if (waiting_Q.buffer[waiting_Q.front+1].remain_time ==0){

waiting_Q.buffer[waiting_Q.front+1].finish_time = Time;

enqueue(&terminated_Q, waiting_Q.buffer[waiting_Q.front+1]);

dequeue(&waiting_Q);

}

```
                    else{
                        enqueue(&ready_Q,
waiting_Q.buffer[waiting_Q.front+1]);
                        dequeue(&waiting_Q);
                        }
                }
            }
            sort_by_priority(&ready_Q);

            if(running_Q.pid !=0){ //run_Q존재, ready_q도 존재
                running_Q.remain_time -= 1;
                running_Q.execute_time += 1;

                if(running_Q.remain_time <= 0 & running_Q.io_remain_time == 0){ //no
more burst and no more i/o
                    running_Q.finish_time = Time;
                    enqueue(&terminated_Q,running_Q);
                    running_Q = init_PCB();
                }

                if(running_Q.io_remain_time !=0 &&
                        (running_Q.io_start_time ==
running_Q.remain_time)){ //만약 IO연산을 수행해야 한다면
                    io_count += 1;
                    enqueue(&waiting_Q, running_Q); //waiting_Q로 이동시킨다.
                    running_Q = init_PCB();          //run_Q를 초기화
```

```
                }
            }

            if(running_Q.pid == 0 && get_Queue_head(&ready_Q).pid!=0){ //run_Q empty, is ready_Q : ready->run
                running_Q = get_Queue_head(&ready_Q);
                dequeue(&ready_Q);
                Context_switch+=1;
            }

            Gantt[Time] = running_Q.pid;
            Time += 1;
        }
        for(int i=terminated_Q.front+1; i<= terminated_Q.rear; i++){
            PCB temp;
            temp = terminated_Q.buffer[i];
            terminated_Q.buffer[i].waiting_time = temp.finish_time - temp.execute_time - temp.arrive_time;
        }
        print_Queue(&terminated_Q);
        print_Gantt(Time-1);

        result[2].process = queue->size-1;
        result[2].context_switch = Context_switch;
        for(int i=(terminated_Q.front+1); i<=terminated_Q.rear; i++){
            result[2].sum_wait += terminated_Q.buffer[i].waiting_time;
```

```
                result[2].sum_burst += terminated_Q.buffer[i].burst_time;
                result[2].sum_turn += terminated_Q.buffer[i].waiting_time + terminated_Q.buffer[i].burst_time;
        }
        result[2].avg_wait = (float)result[2].sum_wait / result[2].process;
        result[2].avg_burst = (float)result[2].sum_burst / result[2].process;
        result[2].avg_turn = (float)result[2].sum_turn / result[2].process;
        result[2].io_count = io_count;
}


void Round_Robin(Queue *queue){
        printf("₩n***Round Robin***₩n");
        Context_switch = 0;
        Time = 0;
        int io_count =0;
        int time_quantum = 3;

        init_Gantt();
        Queue RR;       //생성된 프로세스들을 저장한 queue를 받아서 저장한다.
        init_Queue(&RR, queue->size-1);
        copy_Queue(queue, &RR);

        Queue ready_Q, waiting_Q, terminated_Q;
        init_Queue(&ready_Q, queue->size-1);
        init_Queue(&waiting_Q, queue->size-1);
```

```
init_Queue(&terminated_Q, queue->size-1);

PCB running_Q;
running_Q = init_PCB();

sort_by_arrival(&RR); // FCFS에 있는 프로세스들을 도착시간 순으로 정렬한다

while(is_Queue_full(&terminated_Q) != 1){ //terminated_Q가 꽉찰때 까지 time을 1씩 늘려가며 반복
    for(int i=RR.front+1; RR.buffer[i].pid!=0; i++){ //TIME에 도달할 때 ready_Q에 들어간다
        if(RR.buffer[i].arrive_time != Time)
            break;
        else{
            enqueue(&ready_Q, get_Queue_head(&RR));
            dequeue(&RR);
        }
    }
    //sort_by_arrival(&ready_Q);

    if(waiting_Q.buffer[waiting_Q.front+1].pid!=0){ //waiting_Q에 프로세스가 존재한다면 io_time을 1줄인다.
        waiting_Q.buffer[waiting_Q.front+1].io_remain_time -= 1;

        if(waiting_Q.buffer[waiting_Q.front+1].io_remain_time ==0){ //I.O 입출력이 끝난다면, ready_q로 보낸다
            if (waiting_Q.buffer[waiting_Q.front+1].remain_time ==0){
```

```
                                        //waiting_Q.buffer[waiting_Q.front+1].arrive_time = Time;
                                        waiting_Q.buffer[waiting_Q.front+1].finish_time = Time;
                                        enqueue(&terminated_Q, waiting_Q.buffer[waiting_Q.front+1]);
                                        dequeue(&waiting_Q);
                                    }
                                    else{
                                        //waiting_Q.buffer[waiting_Q.front+1].arrive_time = Time;  //Time으로 도착시간을 변경한뒤 enqueue
                                        enqueue(&ready_Q, waiting_Q.buffer[waiting_Q.front+1]);
                                        dequeue(&waiting_Q);
                                        }
                                }
                            }
                            //sort_by_arrival(&ready_Q);

                            if(running_Q.pid !=0){ //run_Q존재, ready_q도 존재
                                running_Q.remain_time -= 1;
                                running_Q.execute_time += 1;

                                if(running_Q.remain_time <= 0 & running_Q.io_remain_time == 0){ //no more burst and no more i/o
                                    running_Q.finish_time = Time;
                                    enqueue(&terminated_Q,running_Q);
                                    running_Q = init_PCB();
```

```
                }

                if(running_Q.io_remain_time !=0 &&
                        (running_Q.io_start_time ==
running_Q.remain_time)){ //만약 IO연산을 수행해야 한다면
                    io_count += 1;
                    enqueue(&waiting_Q, running_Q); //waiting_Q로 이동시킨다.
                    running_Q = init_PCB();          //run_Q를 초기화
                }

                if(running_Q.pid!=0 && (running_Q.execute_time % time_quantum) ==0){
                    if (running_Q.execute_time>0){
                        enqueue(&ready_Q,running_Q);
                        running_Q = init_PCB();
                    }
                }
            }

            if(running_Q.pid == 0 && get_Queue_head(&ready_Q).pid!=0){ //run_Q empty, is
ready_Q : ready->run
                running_Q = get_Queue_head(&ready_Q);
                dequeue(&ready_Q);
                Context_switch+=1;
            }

            Gantt[Time] = running_Q.pid;
```

```
		Time += 1;
	}
	for(int i=terminated_Q.front+1; i<= terminated_Q.rear; i++){
		PCB temp;
		temp = terminated_Q.buffer[i];
		terminated_Q.buffer[i].waiting_time = temp.finish_time - temp.execute_time - temp.arrive_time;
	}
	print_Queue(&terminated_Q);
	print_Gantt(Time-1);

	result[3].process = queue->size-1;
	result[3].context_switch = Context_switch;
	for(int i=(terminated_Q.front+1); i<=terminated_Q.rear; i++){
		result[3].sum_wait += terminated_Q.buffer[i].waiting_time;
		result[3].sum_burst += terminated_Q.buffer[i].burst_time;
		result[3].sum_turn += terminated_Q.buffer[i].waiting_time + terminated_Q.buffer[i].burst_time;
	}
	result[3].avg_wait = (float)result[3].sum_wait / result[3].process;
	result[3].avg_burst = (float)result[3].sum_burst / result[3].process;
	result[3].avg_turn = (float)result[3].sum_turn / result[3].process;
	result[3].io_count = io_count;
}
```

```
void Preemptive_SJF(Queue *queue){
	printf("₩n**Preemtive SJF**₩n");
	Context_switch = 0;
	Time = 0;
	int io_count =0;

	init_Gantt();
	Queue SJF;	//생성된 프로세스들을 저장한 queue를 받아서 저장한다.
	init_Queue(&SJF, queue->size-1);
	copy_Queue(queue, &SJF);

	Queue ready_Q, waiting_Q, terminated_Q;
	init_Queue(&ready_Q, queue->size-1);
	init_Queue(&waiting_Q, queue->size-1);
	init_Queue(&terminated_Q, queue->size-1);

	PCB running_Q;
	running_Q = init_PCB();

	sort_by_arrival(&SJF); // SJF에 있는 프로세스들을 도착시간 순으로 정렬한다

	while(is_Queue_full(&terminated_Q) != 1){ //terminated_Q가 꽉찰때 까지 time을 1씩 늘려가며 반복
		for(int i=SJF.front+1; SJF.buffer[i].pid!=0; i++){ //TIME에 도달할 때 ready_Q에 들어간다
			if(SJF.buffer[i].arrive_time != Time)
```

```
                    break;
                else{
                    enqueue(&ready_Q, get_Queue_head(&SJF));
                    dequeue(&SJF);
                }
            }
            sort_by_remain(&ready_Q);

            if(waiting_Q.buffer[waiting_Q.front+1].pid!=0){ //waiting_Q에 프로세스가 존재한다면 io_time을 1줄인다.
                waiting_Q.buffer[waiting_Q.front+1].io_remain_time -= 1;

                if(waiting_Q.buffer[waiting_Q.front+1].io_remain_time ==0){ //I.O 입출력이 끝난다면, ready_q로 보낸다
                    if (waiting_Q.buffer[waiting_Q.front+1].remain_time ==0){
                        waiting_Q.buffer[waiting_Q.front+1].finish_time = Time;
                        enqueue(&terminated_Q, waiting_Q.buffer[waiting_Q.front+1]);
                        dequeue(&waiting_Q);
                    }
                    else{
                        enqueue(&ready_Q, waiting_Q.buffer[waiting_Q.front+1]);
                        dequeue(&waiting_Q);
                    }
                }
            }
```

```
sort_by_remain(&ready_Q);

if(running_Q.pid !=0){ //run_Q존재
    running_Q.remain_time -= 1;
    running_Q.execute_time += 1;

    if(running_Q.remain_time <= 0 & running_Q.io_remain_time == 0){ //no more burst and no more i/o
        running_Q.finish_time = Time;
        enqueue(&terminated_Q,running_Q);
        running_Q = init_PCB();
    }

    if(running_Q.io_remain_time !=0 &&
        (running_Q.io_start_time == running_Q.remain_time)){ //만약 IO연산을 수행해야 한다면
        io_count += 1;
        enqueue(&waiting_Q, running_Q); //waiting_Q로 이동시킨다.
        running_Q = init_PCB();          //run_Q를 초기화
    }
    if(running_Q.pid != 0 && get_Queue_head(&ready_Q).pid!=0){ //만약 running_q가 아직 있고, ready_Q도 있다면
        if(running_Q.remain_time > get_Queue_head(&ready_Q).remain_time){ //ready_Q가 더 shortest_job이라면 비운다
            enqueue(&ready_Q,running_Q);
            running_Q = init_PCB();
        }
```

```
                }
            }

            if(running_Q.pid == 0 && get_Queue_head(&ready_Q).pid!=0){ //run_Q empty, is ready_Q : ready->run
                running_Q = get_Queue_head(&ready_Q);
                dequeue(&ready_Q);
                Context_switch+=1;
            }

            Gantt[Time] = running_Q.pid;
            Time += 1;
        }
        for(int i=terminated_Q.front+1; i<= terminated_Q.rear; i++){
            PCB temp;
            temp = terminated_Q.buffer[i];
            terminated_Q.buffer[i].waiting_time = temp.finish_time - temp.execute_time - temp.arrive_time;
        }
        print_Queue(&terminated_Q);
        print_Gantt(Time-1);


        result[4].process = queue->size-1;
        result[4].context_switch = Context_switch;
        for(int i=(terminated_Q.front+1); i<=terminated_Q.rear; i++){
```

```
                result[4].sum_wait += terminated_Q.buffer[i].waiting_time;
                result[4].sum_burst += terminated_Q.buffer[i].burst_time;
                result[4].sum_turn += terminated_Q.buffer[i].waiting_time + terminated_Q.buffer[i].burst_time;
        }
        result[4].avg_wait = (float)result[4].sum_wait / result[4].process;
        result[4].avg_burst = (float)result[4].sum_burst / result[4].process;
        result[4].avg_turn = (float)result[4].sum_turn / result[4].process;
        result[4].io_count = io_count;
}

void Preemptive_Priority(Queue *queue,int aging){
        int result_num;
        if(aging){
                printf("₩n**Preemtive Priority(aging)**₩n");
                result_num = 6;
        }
        else{
                printf("₩n**Preemtive Priority**₩n");
                result_num = 5;
        }
        Context_switch = 0;
        Time = 0;
        int io_count =0;

        init_Gantt();
```

```
Queue PRIORITY;      //생성된 프로세스들을 저장한 queue를 받아서 저장한다.
init_Queue(&PRIORITY, queue->size-1);
copy_Queue(queue, &PRIORITY);

Queue ready_Q, waiting_Q, terminated_Q;
init_Queue(&ready_Q, queue->size-1);
init_Queue(&waiting_Q, queue->size-1);
init_Queue(&terminated_Q, queue->size-1);

PCB running_Q;
running_Q = init_PCB();

sort_by_arrival(&PRIORITY); // SJF에 있는 프로세스들을 도착시간 순으로 정렬한다

while(is_Queue_full(&terminated_Q) != 1){ //terminated_Q가 꽉찰때 까지 time을 1씩 늘려가며 반복
        if (Time == 200){
                printf("TIME:%d",Time);
                printf("[Ready_Q]₩n");
                printf("front:%d, rear:%d, count:%d₩n",ready_Q.front,ready_Q.rear,ready_Q.count);
                print_Queue(&ready_Q);
                printf("[Waiting_Q]₩n");
                print_Queue(&waiting_Q);
        }
```

```
for(int i=PRIORITY.front+1; PRIORITY.buffer[i].pid!=0; i++){ //TIME에 도달할 때 ready_Q에 들어간다
    if(PRIORITY.buffer[i].arrive_time != Time)
        break;
    else{
        enqueue(&ready_Q, get_Queue_head(&PRIORITY));
        dequeue(&PRIORITY);
    }
}
sort_by_priority(&ready_Q);

if(waiting_Q.buffer[waiting_Q.front+1].pid!=0){ //waiting_Q에 프로세스가 존재한다면 io_time을 1줄인다.
    waiting_Q.buffer[waiting_Q.front+1].io_remain_time -= 1;

    if(waiting_Q.buffer[waiting_Q.front+1].io_remain_time ==0){ //I.O 입출력이 끝난다면, ready_q로 보낸다
        if (waiting_Q.buffer[waiting_Q.front+1].remain_time ==0){
            waiting_Q.buffer[waiting_Q.front+1].finish_time = Time;
            enqueue(&terminated_Q, waiting_Q.buffer[waiting_Q.front+1]);
            dequeue(&waiting_Q);
        }
        else{
            enqueue(&ready_Q, waiting_Q.buffer[waiting_Q.front+1]);
            dequeue(&waiting_Q);
```

```
                    }
                }
            }
            sort_by_priority(&ready_Q);

            if(running_Q.pid !=0){ //run_Q존재, ready_q도 존재
                running_Q.remain_time -= 1;
                running_Q.execute_time += 1;

                if(running_Q.remain_time <= 0 & running_Q.io_remain_time == 0){ //no more burst and no more i/o
                    running_Q.finish_time = Time;
                    enqueue(&terminated_Q,running_Q);
                    running_Q = init_PCB();
                }

                if(running_Q.io_remain_time !=0 &&
                        (running_Q.io_start_time == running_Q.remain_time)){ //만약 IO연산을 수행해야 한다면
                    io_count += 1;
                    enqueue(&waiting_Q, running_Q); //waiting_Q로 이동시킨다.
                    running_Q = init_PCB();          //run_Q를 초기화
                }
                if(running_Q.pid != 0 && get_Queue_head(&ready_Q).pid!=0){ //만약 running_q가 아직 있고, ready_Q도 있다면
                    if(running_Q.priority > get_Queue_head(&ready_Q).priority){ //ready_Q가 더 shortest_job이라면 비운다
```

```
                    enqueue(&ready_Q,running_Q);
                    running_Q = init_PCB();
                }
            }
        }

        if(running_Q.pid == 0 && get_Queue_head(&ready_Q).pid!=0){ //run_Q empty, is ready_Q : ready->run
            running_Q = get_Queue_head(&ready_Q);
            dequeue(&ready_Q);
            Context_switch+=1;
        }

        if(aging && Time % 10 == 0){
            for(int i=(ready_Q.front + 1) % ready_Q.size; (i % ready_Q.size)!=((ready_Q.rear+1) % ready_Q.size); i++){
                if(Time - ready_Q.buffer[i].execute_time - ready_Q.buffer[i].arrive_time >=10)
                    ready_Q.buffer[i].priority -= 2;
            }
            for(int i=(waiting_Q.front + 1) % waiting_Q.size; (i % waiting_Q.size)!=((waiting_Q.rear+1) % waiting_Q.size); i++){
                if(Time - waiting_Q.buffer[i].execute_time - waiting_Q.buffer[i].arrive_time >=10)
                    waiting_Q.buffer[i].priority -= 2;
            }
        }
```

```
            Gantt[Time] = running_Q.pid;
            Time += 1;
        }
        for(int i=terminated_Q.front+1; i<= terminated_Q.rear; i++){
            PCB temp;
            temp = terminated_Q.buffer[i];
            terminated_Q.buffer[i].waiting_time = temp.finish_time - temp.execute_time - temp.arrive_time;
        }
        print_Queue(&terminated_Q);
        print_Gantt(Time-1);

        result[result_num].process = queue->size-1;
        result[result_num].context_switch = Context_switch;
        for(int i=(terminated_Q.front+1); i<=terminated_Q.rear; i++){
            result[result_num].sum_wait += terminated_Q.buffer[i].waiting_time;
            result[result_num].sum_burst += terminated_Q.buffer[i].burst_time;
            result[result_num].sum_turn += terminated_Q.buffer[i].waiting_time + terminated_Q.buffer[i].burst_time;
        }

       result[result_num].avg_wait = (float)result[result_num].sum_wait / result[result_num].process;
        result[result_num].avg_burst = (float)result[result_num].sum_burst / result[result_num].process;
       result[result_num].avg_turn = (float)result[result_num].sum_turn / result[result_num].process;
```

```
        result[result_num].io_count = io_count;
}



/* if aging ->Multilevel Feedback Queue*/
void Multilevel_Queue(Queue *queue,int aging){
        int result_num;
        int time_quantum = 3;

        if(aging){
                printf("₩n**Multilevel Feedback Queue**₩n");
                result_num = 8;
        }
        else{
                printf("₩n**Multilevel Queue**₩n");
                result_num = 7;
        }
        Context_switch = 0;
        Time = 0;
        int io_count =0;

        for(int i=0; i<1000; i++){
                Gantt[i] = 0;
                Gantt2[i] = 0;
        }
```

```
Queue RR;      //생성된 프로세스들을 저장한 queue를 받아서 저장한다.
Queue FCFS;
init_Queue(&RR, queue->size-1);
init_Queue(&FCFS, queue->size-1);

for(int i=queue->front+1; i<=queue->rear; i++){
        if(queue->buffer[i].priority < 6)  //priority가 5보다 작을 때
                enqueue(&RR, queue->buffer[i]);
        else
                enqueue(&FCFS, queue->buffer[i]);
}

Queue ready_Q, waiting_Q, terminated_Q;
Queue ready_Q2, waiting_Q2;

init_Queue(&ready_Q, 3*queue->size-1);
init_Queue(&waiting_Q, 3*queue->size-1);
init_Queue(&ready_Q2, 3*queue->size-1);
init_Queue(&waiting_Q2,3*queue->size-1);
init_Queue(&terminated_Q, queue->size-1);

PCB running_Q,running_Q2;
running_Q = init_PCB();
running_Q2 = init_PCB();

sort_by_arrival(&RR);
```

```
sort_by_arrival(&FCFS);
printf("[RR]₩n");
printf("front:%d rear:%d count:%d ₩n",RR.front, RR.rear, RR.count);
print_Queue(&RR);
printf("[FCFS]₩n");
printf("front:%d rear:%d count:%d ₩n",FCFS.front, FCFS.rear, FCFS.count);
print_Queue(&FCFS);

while(is_Queue_full(&terminated_Q) != 1){ //terminated_Q가 꽉찰때 까지 time을 1씩 늘려 가며 반복
        //RR에서 arrive_time에 따라서 ready_Q로 들어간다
        for(int i=RR.front+1; RR.buffer[i].pid!=0; i++){
                if(RR.buffer[i].arrive_time != Time)
                        break;
                else{
                        enqueue(&ready_Q, get_Queue_head(&RR));
                        dequeue(&RR);
                }
        }
        //FCFS에서 arrive_time에 따라서 ready_Q2로 들어간다
        for(int i=FCFS.front+1; FCFS.buffer[i].pid!=0; i++){
                if(FCFS.buffer[i].arrive_time != Time)
                        break;
                else{
                        enqueue(&ready_Q2, get_Queue_head(&FCFS));
                        dequeue(&FCFS);
```

```
                }
            }

            if(waiting_Q.buffer[waiting_Q.front+1].pid!=0){  //waiting_Q에 프로세스가 존재한다면 io_time을 1줄인다.
                waiting_Q.buffer[waiting_Q.front+1].io_remain_time -= 1;

                if(waiting_Q.buffer[waiting_Q.front+1].io_remain_time ==0){ //I.O 입출력이 끝난다면, ready_q로 보낸다
                    if (waiting_Q.buffer[waiting_Q.front+1].remain_time ==0){
                        waiting_Q.buffer[waiting_Q.front+1].finish_time = Time;
                        enqueue(&terminated_Q, waiting_Q.buffer[waiting_Q.front+1]);
                        dequeue(&waiting_Q);
                    }
                    else{
                        enqueue(&ready_Q, waiting_Q.buffer[waiting_Q.front+1]);
                        dequeue(&waiting_Q);
                    }
                }
            }

            /* 70%의 시간이 RR에 할당된다 */
            if(Time % 10 < 7){
```

```
if(running_Q.pid !=0){ //run_Q존재, ready_q도 존재
    running_Q.remain_time -= 1;

    running_Q.execute_time += 1;

    if(running_Q.remain_time <= 0 & running_Q.io_remain_time == 0){ //no more burst and no more i/o
        running_Q.finish_time = Time;
        enqueue(&terminated_Q,running_Q);
        running_Q = init_PCB();
    }

    if(running_Q.io_remain_time !=0 &&
        (running_Q.io_start_time == running_Q.remain_time)){ //만약 IO연산을 수행해야 한다면
        io_count += 1;
        enqueue(&waiting_Q, running_Q); //waiting_Q로 이동시킨다.
        running_Q = init_PCB();          //run_Q를 초기화
    }

    if(running_Q.pid!=0 && (running_Q.execute_time % time_quantum) ==0){
        if (running_Q.execute_time>0){
            enqueue(&ready_Q,running_Q);
            running_Q = init_PCB();
```

```
                    }
                }
            }

            if(running_Q.pid == 0 && get_Queue_head(&ready_Q).pid!=0){ //run_Q empty, is ready_Q : ready->run
                running_Q = get_Queue_head(&ready_Q);
                dequeue(&ready_Q);
                Context_switch+=1;
            }

            Gantt[Time] = running_Q.pid;
        }


        if(waiting_Q2.buffer[waiting_Q2.front+1].pid!=0){ //waiting_Q에 프로세스가 존재한다면 io_time을 1줄인다.
            waiting_Q2.buffer[waiting_Q2.front+1].io_remain_time -= 1;

            if(waiting_Q2.buffer[waiting_Q2.front+1].io_remain_time ==0){ //I.O 입출력이 끝난다면, ready_q로 보낸다
                if (waiting_Q2.buffer[waiting_Q2.front+1].remain_time ==0){
                    //waiting_Q.buffer[waiting_Q.front+1].arrive_time = Time;
                    waiting_Q2.buffer[waiting_Q2.front+1].finish_time = Time;
                    enqueue(&terminated_Q,
```

```
waiting_Q2.buffer[waiting_Q2.front+1]);
                                        dequeue(&waiting_Q2);
                                }
                                else{
                                        //waiting_Q.buffer[waiting_Q.front+1].arrive_time = Time;  //Time으로 도착시간을 변경한뒤 enqueue
                                        enqueue(&ready_Q2, waiting_Q2.buffer[waiting_Q2.front+1]);
                                        dequeue(&waiting_Q2);
                                }
                        }
                }
                /*30%의 시간이 FCFS에 할당*/
                if(Time % 10 >=7){

                        if(running_Q2.pid !=0){ //run_Q존재, ready_q도 존재
                                running_Q2.remain_time -= 1;
                                running_Q2.execute_time += 1;

                                if(running_Q2.remain_time <= 0 & running_Q2.io_remain_time == 0){ //no more burst and no more i/o
                                        running_Q2.finish_time = Time;
                                        enqueue(&terminated_Q,running_Q2);
                                        running_Q2 = init_PCB();
                                }
```

```
                    if(running_Q2.io_remain_time !=0 &&
                            (running_Q2.io_start_time ==
running_Q2.remain_time)){ //만약 IO연산을 수행해야 한다면
                        io_count += 1;
                        enqueue(&waiting_Q2, running_Q2); //waiting_Q로 이
동시킨다.
                        running_Q2 = init_PCB();          //run_Q를 초기화
                    }
                }

                if(running_Q2.pid == 0 &&
get_Queue_head(&ready_Q2).pid!=0){ //run_Q empty, is ready_Q : ready->run
                    running_Q2 = get_Queue_head(&ready_Q2);
                    dequeue(&ready_Q2);
                    Context_switch+=1;
                }

                if(aging && (Time % 9 == 0)){
                    for(int i=(ready_Q2.front + 1) % ready_Q2.size; (i %
ready_Q2.size)!=((ready_Q2.rear+1) % ready_Q2.size); i++){
                        if(Time - ready_Q2.buffer[i].execute_time -
ready_Q2.buffer[i].arrive_time >=10)
                            ready_Q2.buffer[i].priority -= 3;
                            if(ready_Q2.buffer[i].priority < 6){

        enqueue(&ready_Q,ready_Q2.buffer[i]);
                                ready_Q2.buffer[i] =
ready_Q2.buffer[ready_Q2.front+1]; //Change the first and switcher
```

```
                                        dequeue(&ready_Q2);
                                    }
                                }
                                for(int i=(waiting_Q2.front + 1) % waiting_Q2.size;(i % waiting_Q2.size)!=((waiting_Q2.rear+1) % waiting_Q2.size); i++){
                                    if(Time - waiting_Q2.buffer[i].execute_time - waiting_Q2.buffer[i].arrive_time >=10){
                                        waiting_Q2.buffer[i].priority -= 3;
                                        if(waiting_Q2.buffer[i].priority < 6){
        enqueue(&waiting_Q,waiting_Q2.buffer[i]);
                                            dequeue(&waiting_Q2);
                                        }
                                    }
                                }
                            }
                Gantt2[Time] = running_Q2.pid;
                }
                Time += 1;
        }

        for(int i=terminated_Q.front+1; i<= terminated_Q.rear; i++){
                PCB temp;
                temp = terminated_Q.buffer[i];
                terminated_Q.buffer[i].waiting_time = temp.finish_time - temp.execute_time - temp.arrive_time;
        }
```

```
		print_Queue(&terminated_Q);
		print_Gantt(Time-1);
		for(int i=0; i<Time; i++) //Gantt2출력을 위해 대체
			Gantt[i] = Gantt2[i];
		print_Gantt(Time-1);

		result[result_num].process = queue->size-1;
		result[result_num].context_switch = Context_switch;
		for(int i=(terminated_Q.front+1); i<=terminated_Q.rear; i++){
			result[result_num].sum_wait += terminated_Q.buffer[i].waiting_time;
			result[result_num].sum_burst += terminated_Q.buffer[i].burst_time;
			result[result_num].sum_turn += terminated_Q.buffer[i].waiting_time + terminated_Q.buffer[i].burst_time;
		}
		result[result_num].avg_wait = (float)result[result_num].sum_wait / result[result_num].process;
		result[result_num].avg_burst = (float)result[result_num].sum_burst / result[result_num].process;
		result[result_num].avg_turn = (float)result[result_num].sum_turn / result[result_num].process;
		result[result_num].io_count = io_count;
}

void Realtime(Queue *queue, int edf){
		int result_num;

		Context_switch = 0;
```

```
Time = 0;
int io_count =0;

init_Gantt();
Queue PRIORITY;      //생성된 프로세스들을 저장한 queue를 받아서 저장한다.
init_Queue(&PRIORITY, queue->size-1);
copy_Queue(queue, &PRIORITY);

Queue ready_Q, waiting_Q, terminated_Q;
init_Queue(&ready_Q, 5 * (queue->size-1));
init_Queue(&waiting_Q,5 * ( queue->size-1));
init_Queue(&terminated_Q, 5 *(queue->size-1));

PCB running_Q;
running_Q = init_PCB();

if(edf ==0){ //rate_monotonic
            result_num = 9;
            printf("₩n*******RATE MONOTONIC********₩n");
}
else{   //earliest deadline first
            result_num = 10;
            printf("₩n**EARLIEST DEADLINE FIRST****₩n");

}
for(int i=PRIORITY.front+1; PRIORITY.buffer[i].pid!=0; i++){
```

```
            PRIORITY.buffer[i].priority = PRIORITY.buffer[i].period;
            PRIORITY.buffer[i].arrive_time = 0;
        }
        for(int i=PRIORITY.rear+1; i<PRIORITY.size; i++)
            PRIORITY.buffer[i].pid = 0;

        sort_by_priority(&PRIORITY); // SJF에 있는 프로세스들을 도착시간 순으로 정렬한다
        print_Queue(&PRIORITY);

        while(is_Queue_full(&terminated_Q) != 1){ //terminated_Q가 꽉찰때 까지 time을 1씩 늘려가며 반복
            if(edf == 1){
                for(int i=(ready_Q.front+1)%ready_Q.size; i <= ready_Q.rear; i++)
                    ready_Q.buffer[i].priority -= 1;
                running_Q.priority -= 1;
                for(int i=waiting_Q.front+1; i <= waiting_Q.rear; i++)
                    waiting_Q.buffer[i].priority -= 1;
            }

            //25마다 period를 체크해서 period인데 ready_Q에 남아있다면, deadline miss이다
            if(Time == 201)
                break;
            for(int i=(PRIORITY.front+1); i != PRIORITY.rear+1; i++){ //TIME에 도달할 때 ready_Q에 들어간다
                if((Time % PRIORITY.buffer[i].period) == 0) {
```

```
                int cnt =0;
                for(int j=ready_Q.front+1; j<=ready_Q.rear; j++){
                        if(ready_Q.buffer[j].pid == PRIORITY.buffer[i].pid)
                                printf("Time [%d], [%d] deadline miss!!₩n",Time,PRIORITY.buffer[i].pid);
                }
                PRIORITY.buffer[i].arrive_time = Time;
                enqueue(&ready_Q, PRIORITY.buffer[i]);
            }
        }
        sort_by_priority(&ready_Q);

        if(waiting_Q.buffer[waiting_Q.front+1].pid!=0){ //waiting_Q에 프로세스가 존재한다면 io_time을 1줄인다.
            waiting_Q.buffer[waiting_Q.front+1].io_remain_time -= 1;

            if(waiting_Q.buffer[waiting_Q.front+1].io_remain_time ==0){ //I.O 입출력이 끝난다면, ready_q로 보낸다
                if (waiting_Q.buffer[waiting_Q.front+1].remain_time ==0){
                    waiting_Q.buffer[waiting_Q.front+1].finish_time = Time;
                    enqueue(&terminated_Q, waiting_Q.buffer[waiting_Q.front+1]);
                    dequeue(&waiting_Q);
                }
                else{
                    enqueue(&ready_Q, waiting_Q.buffer[waiting_Q.front+1]);
```

```
                    dequeue(&waiting_Q);
                    }
                }
            }
            sort_by_priority(&ready_Q);

            if(running_Q.pid !=0){ //run_Q존재, ready_q도 존재
                running_Q.remain_time -= 1;
                running_Q.execute_time += 1;

                if(running_Q.remain_time <= 0 & running_Q.io_remain_time == 0){ //no more burst and no more i/o
                    running_Q.finish_time = Time;
                    enqueue(&terminated_Q,running_Q);
                    running_Q = init_PCB();
                }

                if(running_Q.io_remain_time !=0 &&
                        (running_Q.io_start_time == running_Q.remain_time)){ //만약 IO연산을 수행해야 한다면
                    io_count += 1;
                    enqueue(&waiting_Q, running_Q); //waiting_Q로 이동시킨다.
                    running_Q = init_PCB();          //run_Q를 초기화
                }
                if(running_Q.pid != 0 && get_Queue_head(&ready_Q).pid!=0){ //만약 running_q가 아직 있고, ready_Q도 있다면
```

```
                    if(running_Q.priority > get_Queue_head(&ready_Q).priority){ //ready_Q가 더 shortest_job이라면 비운다
                        enqueue(&ready_Q,running_Q);
                        running_Q = init_PCB();
                    }
                }
            }

            if(running_Q.pid == 0 && get_Queue_head(&ready_Q).pid!=0){ //run_Q empty, is ready_Q : ready->run
                running_Q = get_Queue_head(&ready_Q);
                dequeue(&ready_Q);
                Context_switch+=1;
            }
            Gantt[Time] = running_Q.pid;
            Time += 1;
        }
        for(int i=terminated_Q.front+1; i<= terminated_Q.rear; i++){
            PCB temp;
            temp = terminated_Q.buffer[i];
            terminated_Q.buffer[i].waiting_time = temp.finish_time - temp.execute_time - temp.arrive_time;
        }
        print_Queue(&terminated_Q);
        print_Gantt(Time-1);
```

```
	result[result_num].process = queue->size-1;
	result[result_num].context_switch = Context_switch;
	for(int i=(terminated_Q.front+1); i<=terminated_Q.rear; i++){
		result[result_num].sum_wait += terminated_Q.buffer[i].waiting_time;
		result[result_num].sum_burst += terminated_Q.buffer[i].burst_time;
		result[result_num].sum_turn += terminated_Q.buffer[i].waiting_time + terminated_Q.buffer[i].burst_time;
	}
	result[result_num].avg_wait = (float)result[result_num].sum_wait / result[result_num].process;
	result[result_num].avg_burst = (float)result[result_num].sum_burst / result[result_num].process;
	result[result_num].avg_turn = (float)result[result_num].sum_turn / result[result_num].process;
	result[result_num].io_count = io_count;
}

void menu(){
	while(1){
		int p_num;
		int num, num2;
		Queue ready_Q;
		printf("₩n[MENU]₩n");
		printf("[0. 프로세스 생성]₩n");
		printf("[1. Scheduling 알고리즘]₩n");
		printf("[2. 전체 알고리즘 평가]₩n");
		printf("[3. 종료]₩n");
		printf("[수행 하고 싶은 명령을 선택하세요:]₩n ");
```

```
scanf("%d",&num);


switch(num){
        case(0):
                initialize_result();
                p_num = process_num();
                init_Queue(&ready_Q,p_num);
                fill_random_process(&ready_Q);
                printf("[생성된 큐 출력]₩n");
                print_Queue(&ready_Q);
                printf("[도착 순서대로 보기]₩n");
                sort_by_arrival(&ready_Q);
                print_Queue(&ready_Q);
                break;
        case(1):
                printf("₩n[1. FCFS 실행]₩n");
                printf("[2. Nonpreemptive SJF 실행]]₩n");
                printf("[3. Nonpreemptive_Priority 실행]₩n");
                printf("[4. Round Robin 실행]₩n");
                printf("[5. Preemptive SJF 실행]₩n");
                printf("[6. Preemptive Priority실행]₩n");
                printf("[7. Preemptive Priority(aging)실행]₩n");
                printf("[8. Multilevel Queue 실행]₩n");
                printf("[9. Multilevel Feedback Queue실행]₩n");
                printf("[10. Rate Monotonic 실행]₩n");
                printf("[11. Earliest Deadline First실행]₩n");
```

```
printf("[수행 하고 싶은 명령을 선택하세요:]₩n");
scanf("%d",&num2);
switch(num2){
        case(1):
                FCFS(&ready_Q);
                break;
        case(2):
                Nonpreemptive_SJF(&ready_Q);
                break;
        case(3):
                Nonpreemptive_Priority(&ready_Q);
                break;
        case(4):
                Round_Robin(&ready_Q);
                break;
        case(5):
                Preemptive_SJF(&ready_Q);
                break;
        case(6):
                Preemptive_Priority(&ready_Q,0);
                break;
        case(7):
                Preemptive_Priority(&ready_Q,1);
                break;
        case(8):
                Multilevel_Queue(&ready_Q,0);
```

```
                    break;
                case(9):
                    Multilevel_Queue(&ready_Q,1);
                    break;
                case(10):
                    Realtime(&ready_Q,0);
                    break;
                case(11):
                    Realtime(&ready_Q,1);
                    break;
            }
            break;
        case(2):
            FCFS(&ready_Q);
            Nonpreemptive_SJF(&ready_Q);
            Nonpreemptive_Priority(&ready_Q);
            Round_Robin(&ready_Q);
            Preemptive_SJF(&ready_Q);
            Preemptive_Priority(&ready_Q,0);
            Preemptive_Priority(&ready_Q,1);
            Multilevel_Queue(&ready_Q,0);
            Multilevel_Queue(&ready_Q,1);
            Realtime(&ready_Q,0);
            Realtime(&ready_Q,1);
            print_all_result();
            break;
```

```
                        case(3):
                                return;
                        default:
                                printf("잘못된 선택입니다. 다시 입력해주세요!₩n");
                                break;
                }
        }
}


int main(void){
        printf("**********2014190701 이호준**********₩n");
        printf("**************CPU SIMULATOR**************");
        menu();
}
```