COSE 341-01

Operating System

The CPU Scheduling

Term Project Report

2020130826Kim Yuna

Table of Contents

1. 서론

CPU 스케줄링

컴퓨터의 여러 프로그램들은 문제 없이 실행되고 있는데, 이는 사실 메모리에 있는 여러 개의 프로세스가 CPU 자원을 번갈아 할당 받으며 실행되고 있기 때문이다. 이처럼 여러 프로그램들이 문제없이 실행되게 하고 전체 시스템의 성능을 높이기 위해서는 OS가 프로세스들에게 공정하고 합리적으로 CPU 자원을 할당해주는 것이 중요하다. 이를 위한 여러 스케줄링 기법들이 존재한다.

실행되기 위해 기다리고 있는 여러 프로세스 중에서 다음으로 누구에게 CPU 자원을 넘겨줄지에 대한 방식이나 규칙에 대한 것이 스케줄링 기법이다. 또한, 이 스케줄링을 수행하는 것을 CPU 스케줄러라고 할 수 있다. 다시 말해서, CPU 스케줄러는 스케줄링 알고리즘에 따라 어떤 프로세스를 다음에 실행시킬지 결정하는 역할을 한다.

CPU 스케줄링의 분류

여러 CPU 스케줄링 기법이 존재하지만, 각 기법은 크게 선점형 스케줄링(preemtive scheduling)과 비선점형 스케줄링(non-preemtive scheduling)으로 분류할 수 있다.

선점형 스케줄링은 현재 CPU를 사용 중인 프로세스로부터 CPU를 빼앗아 다른 프로세스에 할당하는 방식이다. 이는 어느 한 프로세스의 자원 독점을 막고 프로세스들에 골고루 자원을 배분할 수 있다는 장점이 있지만, 그만큼 문맥 교환 과정에서 오버헤드가 발생할 수 있다는 단점도 존재한다.

비선점형 스케줄링은 현재 CPU를 사용 중인 프로세스의 작업이 끝날 때까지 기다리는 방식이다. 선점형 스케줄링에 비해 문맥 교환에서 발생하는 오버헤드가 적다는 장점이 있지만, 모든 프로세스가 골고루 자원을 이용하기 어려워 starvation 등의 문제가 발생할 수 있다.

CPU 스케줄링 성능 척도

스케줄링의 성능을 평가할 수 있는 여러 척도가 존재한다. 스케줄링 기법이 다양하게 존재하기 때문에, 이러한 성능 척도를 통해 각 스케줄링 기법의 성능을 비교 및 분석해볼 수 있다.

✧ 응답 시간(Response Time)

: 응답 시간이란 프로세스의 요청에 대해 시스템이 최초로 출력을 내주기까지 걸린 시간을 말한다. 이 시간이 짧을수록, 사용자 관점에서는 시스템의 성능이 좋다고 느낄 것이다.

✧ 반환 시간(Turnaround Time)

: 응답 시간이 첫 출력까지의 시간이라면, 반환 시간은 특정 프로세스가 시작해서 끝날 때까지 걸린 총 시간으로 볼 수 있다.

✧ 처리량(Throughput)

: 처리량이란 단위 시간 동안 완료된 프로세스의 개수를 말한다.

✧ 활용도(CPU Utilization)

: 활용도는 CPU의 성능을 측정하는 중요한 척도 중 하나로, 전체 시스템 시간 중 CPU가 작업을 처리한 시간의 비율로 계산된 지표이다. CPU가 idle 상태로 돌입하는 것 없이 얼마나 효율적으로 자원을 사용하였는가를 볼 수 있다.

✧ 대기 시간(Waiting Time)

: 어떤 프로세스가 CPU 자원을 할당 받아 수행되고 있는 동안 다른 프로세스들은 준비큐(Ready Queue)에서 기다리고 있어야 하는데, 대기 시간이 길어질수록 작업 시간이 지연되는 것이므로, 비효율적으로 볼 수 있다. 대기 시간은 전체 작업 시간 동안 CPU를 할당 받지 못하고 얼마나 대기했는지를 알 수 있는 값이다.

CPU 스케줄링 알고리즘

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

: FCFS는 선입선처리 스케줄링으로, 준비큐에 먼저 도착한 프로세스부터 CPU를 할당해주는 방식이다. CPU를 할당받은 프로세스는 작업이 끝날 때까지 CPU를 독점하고 다른 프로세스에게 빼앗길 일이 없기 때문에 비선점 방식에 해당한다. 이는 진행중인 프로세스의 작업 시간이 너무 오래 걸리면 다른 프로세스들이 기다리는 시간이 매우 길어질 수 있다는 치명적인 단점이 존재한다. 또한, 먼저 진행중인 프로세스가 끝나야만 자신의 순서가 돌아오기 때문에 프로세스들의 평균 응답 시간이 길어진다.

2. SJF (Shortest Job First)

: SJF는 최단 작업 우선 스케줄링으로, 준비큐에 있는 프로세스 중에서 CPU 사용 시간이 가장 짧은 프로세스부터 처리하는 기법이다. 앞선 FCFS 방식에 비해 평균 응답 시간은 줄어들겠으나, 짧은 실행 시간을 가진 프로세스들이 계속해서 들어온다면 실행 시간이 긴 프로세스는 점점 자신의 순서가 밀려나므로, 특정 프로세스들은 무한 대기 현상이 발생할 수 있다는 단점이 있다.

3. Preemptive SJF (Preemptive Shortest Job First)

: Preemptive SJF는 준비 큐에 있는 프로세스 중에서 남은 CPU 사용 시간이 가장 짧은 프로세스를 먼저 처리하는 방식이라는 점에서 기존의 SJF 방식과 동일하지만, preemptive하다는 점에서 차이가 있다. 새로운 프로세스가 도착하면 현재 실행 중인 프로세스보다 남은 시간이 짧은 경우, CPU

를 빼앗고 새로운 프로세스를 실행한다. 평균 대기 시간을 최소화할 수 있다는 장점이 있지만, 자주 문맥 교환이 발생할 수 있어 오버헤드가 증가할 수 있다.

4. Priority

: Priority는 우선순위 스케줄링으로, 프로세스들에 우선순위를 부여하고 이를 기준으로 실행할 프로세스를 정하는 방식이다. 하지만, 우선순위가 높은 프로세스만 계속해서 실행되고, 우선순위가 낮다면 준비큐에 먼저 삽입되었음에도 불구하고 실행이 연기되는 문제가 발생한다. 이를 방지하기 위해 오랫동안 대기한 프로세스의 우선순위를 점차 높여주는 에이징 기법을 도입하기도 한다.

5. Preemptive Priority

: Preemptive Priority는 우선순위가 높은 프로세스가 도착하면 현재 실행 중인 프로세스의 CPU를 빼앗고 우선순위가 높은 프로세스를 실행하는 방식이다. 중요한 작업을 즉시 처리할 수 있다는 장점이 있지만, 우선순위가 낮은 프로세스는 무한 대기할 수 있다는 단점도 존재한다.

6. RR(Round Robin)

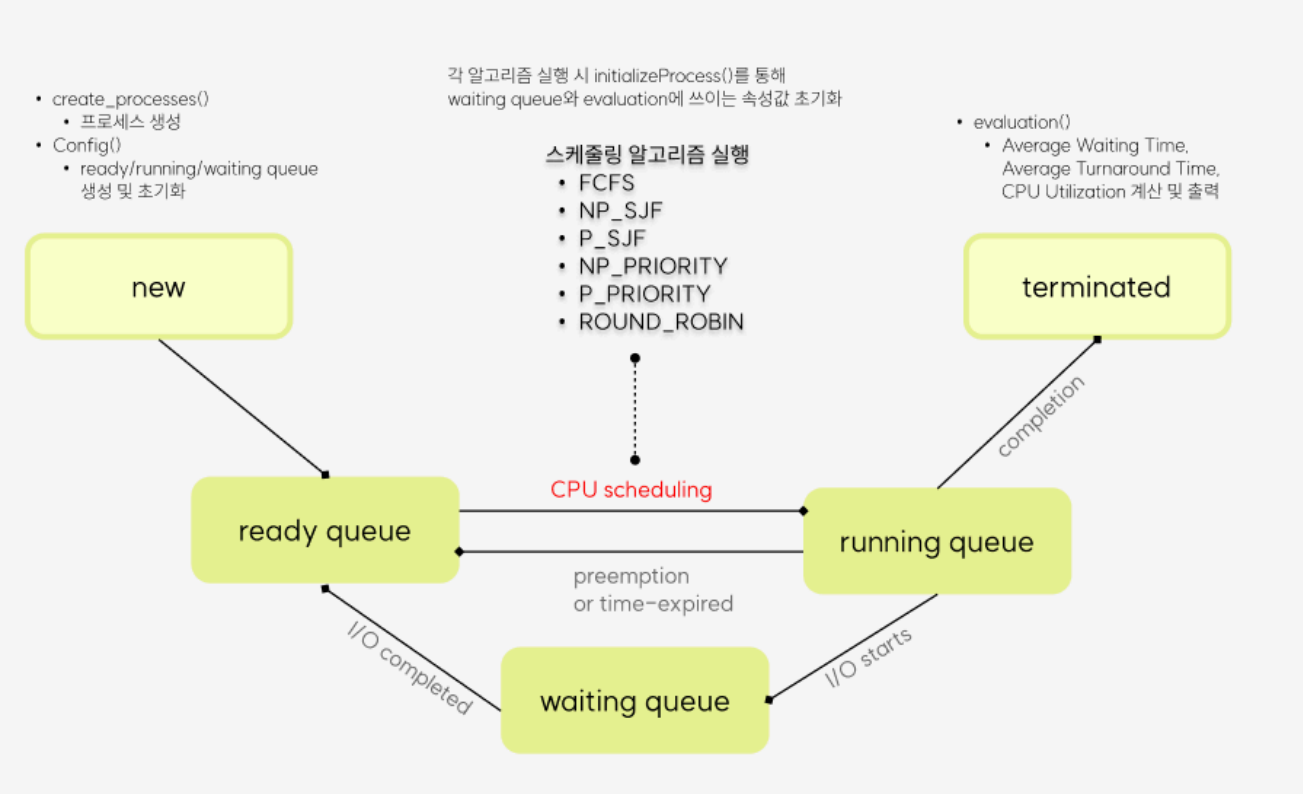
: RR은 라운드 로빈 스케줄링으로, FCFS 스케줄링을 기반으로 하되 각 프로세스가 한 번에 쓸 수 있는 시간을 정해 두어 시간 할당량이 지나면 시간 종료 인터럽트에 의해 CPU를 뺏기는 방식이다. 정해진 시간 할당량 만큼 돌아가며 CPU를 이용하는데, 정해진 시간을 모두 사용했음에도 아직 프로세스가 완료되지 않았다면 다시 준비큐의 맨 뒤에 삽입된다.

이번 프로젝트에서 구현한 CPU 스케줄링

시스템 환경 설정으로 ready_queue, running_queue, waiting_queue 를 생성해주었으며, 프로세스의 개수를 사용자로부터 입력 받아 각 프로세스의 id, burst_time, io_burst_time 등의 속성을 랜덤으로 설정해준다. 생성된 프로세스들을 FCFS, NP_SJF, P_SJF, NP_Priority, P_Priority, RR, Preemptive 스케줄링 기법으로 실행하고, 각 스케줄링 알고리즘들 간의 성능을 비교 및 평가한다.

2. 본론

시뮬레이터의 시스템 구성도



1) new

: create_processes()를 통해 새로운 프로세스들을 생성하고, Config()를 실행하여 CPU simulating에 필요한 ready queue, running queue, waiting queue를 생성하고 초기화한다.

2) ready queue

: 각 프로세스의 arrival_time에 해당되면 ready queue로 이동한다. CPU 작업을 수행하다가 작업이 끝나기 전에 다른 프로세스에 의해 선점된 경우에도 ready queue로 다시 이동하며, I/O 작업이 끝난 프로세스도 ready queue로 옮겨진다.

3) running queue

: CPU 스케줄링 기법에 따라, CPU 스케줄러는 ready queue에 있는 프로세스 중 하나를 running queue로 이동시킨다. 다른 프로세스의 선점이 발생한 경우 기존에 수행되던 프로세스는 running queue에서 ready queue로 이동된다.

4) waiting queue

: 각 프로세스에는 io_burst_time과 io_start_time 값이 존재하는데, I/O 작업 시간에 해당되면

running queue에서 빼낸 후 waiting queue로 이동시키고, waiting queue의 해당 프로세스의 pid에 해당하는 자리에 io_burst_time을 넣어준 후 매 시간마다 감소시켜준다. I/O 작업이 끝나고 아직 burst_time이 남아있는 경우에는 ready queue로 다시 이동시킨다.

5) terminated

: 프로세스의 progress_time(cpu 작업이 진행된 정도)과 burst_time이 같아지면, CPU 작업을 전부 완료했다는 뜻이므로, Terminated 상태로 처리해줘야 한다. 스케줄링 기법의 속성 중 finished_process의 값을 1 증가시켜 완료된 프로세스의 개수를 관리하고, 프로세스의 completed_time에도 끝난 시간을 저장해놓는다. 모든 프로세스의 작업이 종료되었다면, Average Waiting Time과 Average Turnaround Time을 계산해주는 evaluation 작업을 수행한다.

시뮬레이터의 환경 변수 구조체 정의

✧ Process

프로세스의 여러 속성들을 구조체 형식으로 설정해주었다.

```
typedef struct {
    int pid;
    int arrival_time;
    int burst_time;
    int io_burst_time;
    int io_start_time;
    int priority;
    int remaining_time;
    int io_remaining_time;
    int completed_time;
    int waiting_time;
    int turnaround_time;
    int progress_time;
    _Bool entered;
    int order; // np_sjf
    int preemptive; // p_sjf
    int time_quantum; // RR
} Process;
```

✧ Queue & Node

Queue에는 front, rear node를 가리키는 포인터가 정의되어 있으며 각 노드에는 자신의 pid를 담는 data 변수와 큐에서 다음 노드를 알 수 있도록 하는 next 포인터가 정의되어 있다.

```
typedef struct _node {
```

```
  Data data;
  struct _node *next;
} Node;
typedef struct _Queue {
  Node *front;
  Node *rear;
} Queue;
```

✧ Evaluation

성능을 평가할 수 있는 avg_turnaround_time과 avg_waiting_time이 정의되어 있고, 모든 프로세스의 작업을 실행하는 동안 CPU가 idle 상태에 얼마나 있었는지를 알 수 있는 idle_time, 완료된 프로세스 개수를 담는 finished_process, 모든 프로세스가 작업을 완료한 시간인 finished_time이 있다.

```
typedef struct _evaluation {
  float avg_turnaround_time;
  float avg_waiting_time;
  int idle_time;
  int finished_process;
  int finished_time;
} Evaluation;
```

시뮬레이터 모듈

1. create_processes

프로세스의 속성을 무작위로 생성하여 프로세스를 생성하는 역할을 한다. 생성된 프로세스는 실행 시간, I/O 실행 시간, 도착 시간, I/O 시작 시간 및 우선 순위와 같은 속성을 갖는다. 프로세스의 속성이 올바르게 초기화되고, 주어진 한계 내에서 생성되도록 보장해준다.

프로세스 개수 생성은 최대 10개까지 생성 가능하며, process_num 값을 사용자로부터 입력 받는다. 또한, 프로세스 개수만큼 반복문을 실행하면서 각 속성 값을 설정해준다. pid는 해당 index 값으로 부여하였고, cpu_burst_time은 2~10 범위 내로, io_burst_time은 1~6 이내로 설정해주었으며 i/o interrupt는 한 번만 발생하도록 하였다. entered 속성은 프로세스가 CPU 자원을 할당 받았을 때, 즉 context switch 된 시점을 표시하는 역할로 쓰인다.

priority의 경우 중복해서 할당되지 않도록 했고, 적어도 하나의 프로세스는 0초에 도착하는 것을 보장해주도록 처리해주었다.

2. Config

CPU 스케줄링 시뮬레이션에 사용되는 큐를 초기화하는 역할을 한다. ready queue와 running

queue를 QueueInit 함수로 큐의 front와 rear을 NULL로 초기화해준다. waiting queue는 배열로 이루어져 있기 때문에, 값을 -1로 초기화해준다.

3. initialize

이 함수는 각 스케줄링 알고리즘을 시작하기 전에 성능 평가에 쓰이는 변수인 프로세스의 progress_time과 completed_time을 초기화하는 역할을 한다.

4. Queue 관련 함수

큐(Queue)는 데이터 구조로, FIFO(First-In-First-Out) 방식으로 동작한다. 큐의 앞(front)에서 데이터를 제거하고, 뒤(rear)에서 데이터를 추가한다.

IsQueueEmpty 함수는 큐가 비어있는지 확인한다. Enqueue 함수는 큐의 뒤에 데이터를 추가한다. Dequeue 함수는 큐의 앞에서 데이터를 제거하고 반환한다. QPeek는 큐의 앞에 있는 데이터를 확인한다.

5. CPU Scheduling

CPU 스케줄링 기법으로는 FCFS, NP_SJF, P_SJF, NP_PRIORITY, P_PRIORITY, RoundRobin 을 구현하였다. 각 스케줄링 알고리즘에서는 시간의 흐름을 의미하는 time 반복문을 돌게 되는데, 반복문이 종료되는 시점은 완료된 프로세스의 개수와 전체 프로세스 개수가 같아질 때이다.

반복문 내부에서 각 프로세스는 queue에 진입하고 running 되는 등의 작업이 이루어진다. 또한, 각 스케줄링 기법의 속성인 idle_time과 process_progress 변수를 0으로 초기화해주고, initializeProcess 함수를 실행하여 큐를 초기화해준다.

i/o 작업 시작 시간은 절대적인 시간이 아니라 cpu 작업 진행 정도를 기준으로 체크하여 처리한다.

모든 프로세스의 작업이 완료되면 해당 기법의 avg_waiting_time과 avg_turnaround_time을 계산하여 출력한다.

6. evaluation

evaluation함수에서는 모든 스케줄링 기법에 대해 성능 평가 척도인 Average Waiting Time과 Average Turnaround Time, 그리고 CPU Utilization 계산 결과를 출력해준다.

- Average Waiting Time

: 전체 반환 시간에서 전체 실행 시간을 뺀 값을 프로세스 수로 나누어 평균 대기 시간을 구한다. 각 프로세스의 대기 시간(Waiting Time)은 반환 시간에서 실행 시간을 뺀 값이다.

- Average Turnaround Time

: 전체 반환 시간(Turnaround Time) 계산한다. 각 프로세스의 반환 시간(Turnaround Time)은 프

로세스가 완료된 시간(completed_time)에서 도착 시간(arrival_time)을 뺀 값이다. 모든 프로세스의 반환 시간을 더하여 전체 반환 시간을 구하고, 전체 반환 시간을 프로세스 수로 나누어 평균 반환 시간을 구한다.

- CPU Utilization

: 전체 시간에서 CPU가 유휴 상태인 시간을 제외한 비율을 계산하여 CPU 활용도를 구한다. 이를 통해 CPU가 실제로 작업을 수행한 시간을 측정할 수 있다.

스케줄링 알고리즘 구현

1. FCFS

FCFS는 시간 순서에 따라 먼저 들어오는 프로세스를 먼저 처리해주는 방식이며, 기본적으로 non preemption 이기 때문에 특정 프로세스가 수행 중이라면 다른 프로세스들은 프로세스가 CPU 자원을 자진 반납할 때까지 기다려야 한다.

time 반복문을 돌면서 time과 process의 arrival_time이 같은 시점을 체크하여 도착한 프로세스가 있다면 ready queue에 넣어준다. 해당 프로세스의 waiting queue 값이 존재한다면 i/o 작업을 수행하고 있다는 뜻이므로, 남은 i/o 작업을 1 감소시킨다. 이 값이 0이 되는 시점은 i/o 작업이 끝났다는 것이므로 다시 ready queue에 넣어준다.

* running_queue가 비어있는 경우

다음으로, running queue와 ready queue에 프로세스가 들어있는지를 확인한다. 만약, cpu 점유를 기다리는 프로세스는 존재하고 현재 작업중인 프로세스는 없다면 ready queue에 있는 다음 프로세스에게 cpu 자원을 할당해준다.

만약 running queue도 비어 있다면 CPU는 처리해줄 프로세스가 없어서 idle 상태에 놓이게 된다. 이 경우 FCFS 스케줄링 알고리즘의 idle_time 값을 증가시켜준다.

* running_queue에 프로세스가 있는 경우

현재 작업중인 프로세스가 있는 경우, 각 시간마다 cpu 작업 현황에 대한 시간인 progress_time을 증가시켜준다. 만약 i/o 시작 시간에 도달했다면, running queue에서 해당 프로세스를 빼내고 waiting queue에 값을 업데이트해준다.

프로세스의 progress_time과 burst_time이 같아지는 시점은 해당 프로세스의 작업이 완전히 종료되었다는 뜻이므로, running queue에서 빼내고 완료 시간인 completed_time 값을 업데이트해준다.

모든 프로세스의 작업이 완료되면 time에 대한 반복문이 종료되므로, 그 시점의 time을 FCFS 알고리즘의 finished_time으로 업데이트해준다.

2. NP_SJF

NP_SJF는 실행 시간이 가장 짧은 프로세스부터 실행하는 기법이다. 따라서, 해당 알고리즘에서는 각 프로세스의 PID와 CPU burst time을 arr에 저장하고 CPU burst time을 기준으로 arr 배열을 정렬하여 프로세스가 실행될 순서를 결정해주었다. 정렬된 순서에 따라 각 프로세스의 order값이 정해진다.

메인 루프에서 각 타임 슬롯마다 프로세스의 상태를 업데이트 해주고, 프로세스가 도착하면 해당 프로세스를 order 기준으로 ready queue에 넣어준다.

프로세스의 진행 상황을 업데이트하고, I/O 작업이나 프로세스 완료를 체크하는 과정은 Non preemptive이므로 앞선 FCFS 기법과 동일하다.

3. P_SJF

P_SJF는 NP_SJF와는 다르게 실행중인 프로세스가 있더라도 새로 들어온 프로세스와 남은 cpu 작업 시간을 비교하여 선점하거나 선점 당할 수 있다. 따라서, 프로세스의 remaining_time을 이용하여 현재 작업중인 프로세스와 새 프로세스의 remaining_time을 비교하고 새로 들어온 프로세스의 작업시간이 더 작다면 기존에 작업중이던 것은 ready queue로, 새 프로세스는 running queue에 넣어준다. 이때, 작업 중이던 프로세스를 ready queue의 맨 처음 순서로 넣어주는데, 이는 tempQ를 활용하여 구현하였다.

만약 새로 들어온 프로세스가 작업시간이 더 길어 ready queue에 들어가야 할 때, ready queue에서도 남은 시간을 기준으로 정렬해주어야 한다. 이 경우도 tempQ를 활용해 적절한 우선순위의 위치에 넣어주도록 구현하였다.

만약, ready queue에는 프로세스가 존재하고 running queue는 비어 있다면 ready queue를 우선순위에 맞게 정렬해주는 작업이 진행된다.

4. NP_PRIORITY

NP_PRIORITY는 우선순위를 기준으로 프로세스의 작업이 실행되므로, 프로세스의 priority 속성을 이용하여 구현하였다. ready queue는 우선순위를 기준으로 각 프로세스들이 정렬되며, 작업 중인 프로세스가 있으면 ready queue로 정렬되어 들어간다.

5. P_PRIORITY

P_PRIORITY는 앞선 NP_PRIORITY와 동일하게 priority 속성을 기준으로 프로세스들이 실행 및 정렬되는 방식이다. 다만, 다른 점이 있다면 preemptive 하기 때문에, 새로 들어온 프로세스의 우선순위가 기존 프로세스의 우선순위보다 더 높다면 running queue로 들어가고, 기존 프로세스는 ready queue로 넣어준다. 이 과정은 P_SJF 부분과 동일한 방식으로 구현하였다.

6. ROUND_ROBIN

ROUND_ROBIN은 FCFS 방식을 기본으로, time quantum이 추가된 알고리즘 방식이다. 1~5 범

위 내에서 time quantum의 값을 입력 받고, time 루프를 돌면서 running 중인 프로세스의 time_quantum 속성값을 매번 1씩 증가시켜준다.

만약 프로세스의 i/o interrupt가 발생했다면 time_quantum 값을 0으로 초기화해주고 running queue에서 빼낸 후 waiting queue의 값을 업데이트 해준다. 또한, 정해진 time_quantum 시간과 프로세스가 실행된 시간이 동일해지면 cpu 자원을 반납해야 하므로 이 경우에도 time_quantum 값을 0으로 초기화해주고, ready_queue로 보낸다.

스케줄링 알고리즘 성능 평가 및 분석

성능 평가 척도로는 Average Waiting Time, Average Turnaround Time, CPU Utilization를 설정하였고, 이를 기준으로 각 스케줄링 알고리즘의 성능을 분석해보겠다.

I. 프로세스 생성결과

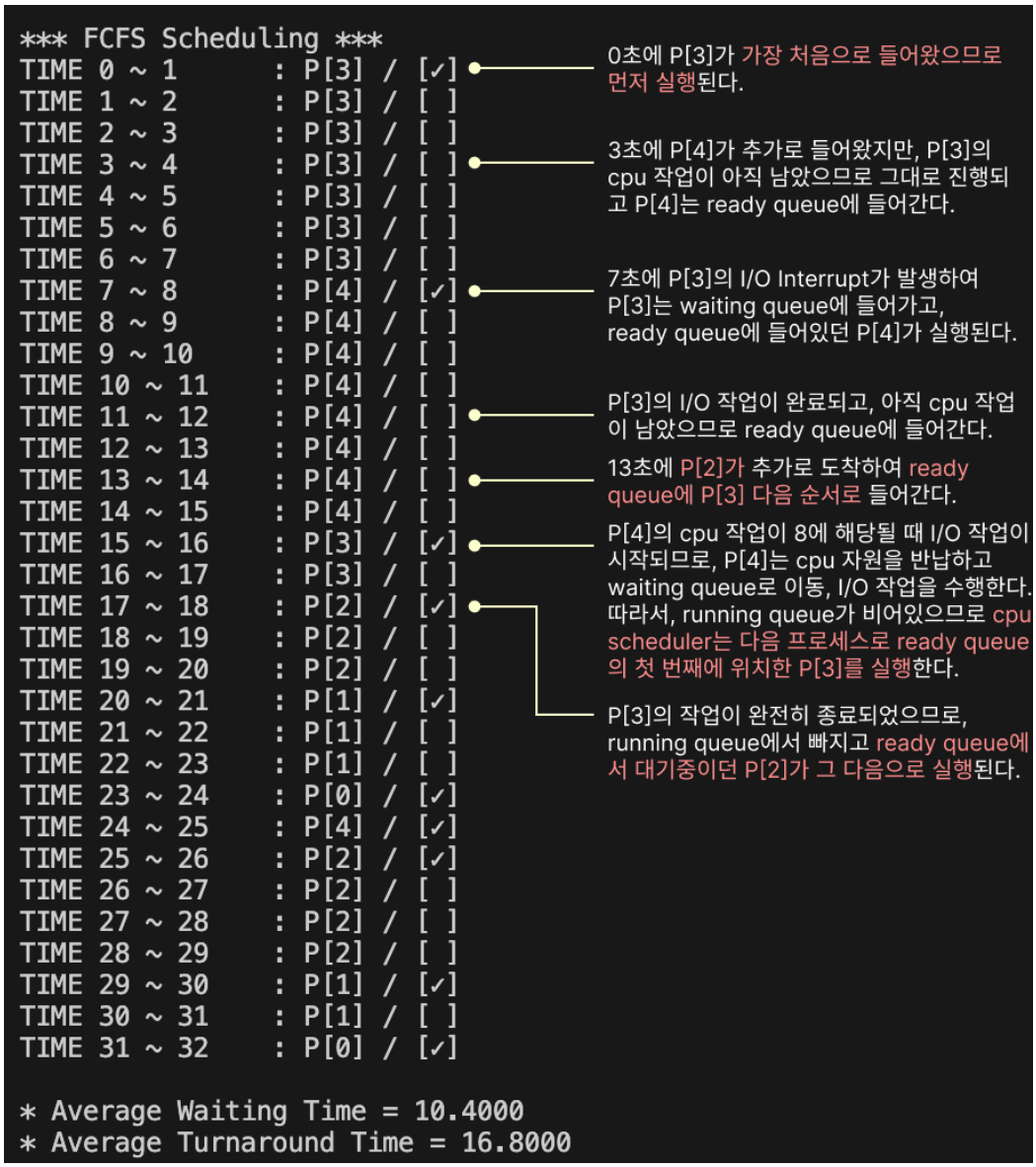
프로세스는 총 4개를 생성하였고 모든 프로세스는 I/O 작업을 포함한다.

```
➜  20241R0136COSE34101-term1 git:(main) ✗ ./scheduler
*** CPU SCHEDULER ***
How many processes do you want to create?: 5
-------------------------------------------------------------------------------
PID             Arrival Time    CPU Burst Time  I/O Burst Time  I/O Start Time  Priority
-------------------------------------------------------------------------------
0               18              2               3               1               1
1               16              5               3               3               2
2               13              7               3               3               4
3               0               9               4               7               5
4               3               9               3               8               3
-------------------------------------------------------------------------------
* Finished creating processes!

* Finished creating queues!
```

II. 각 스케줄링의 실행결과

1. FCFS 스케줄링 실행결과

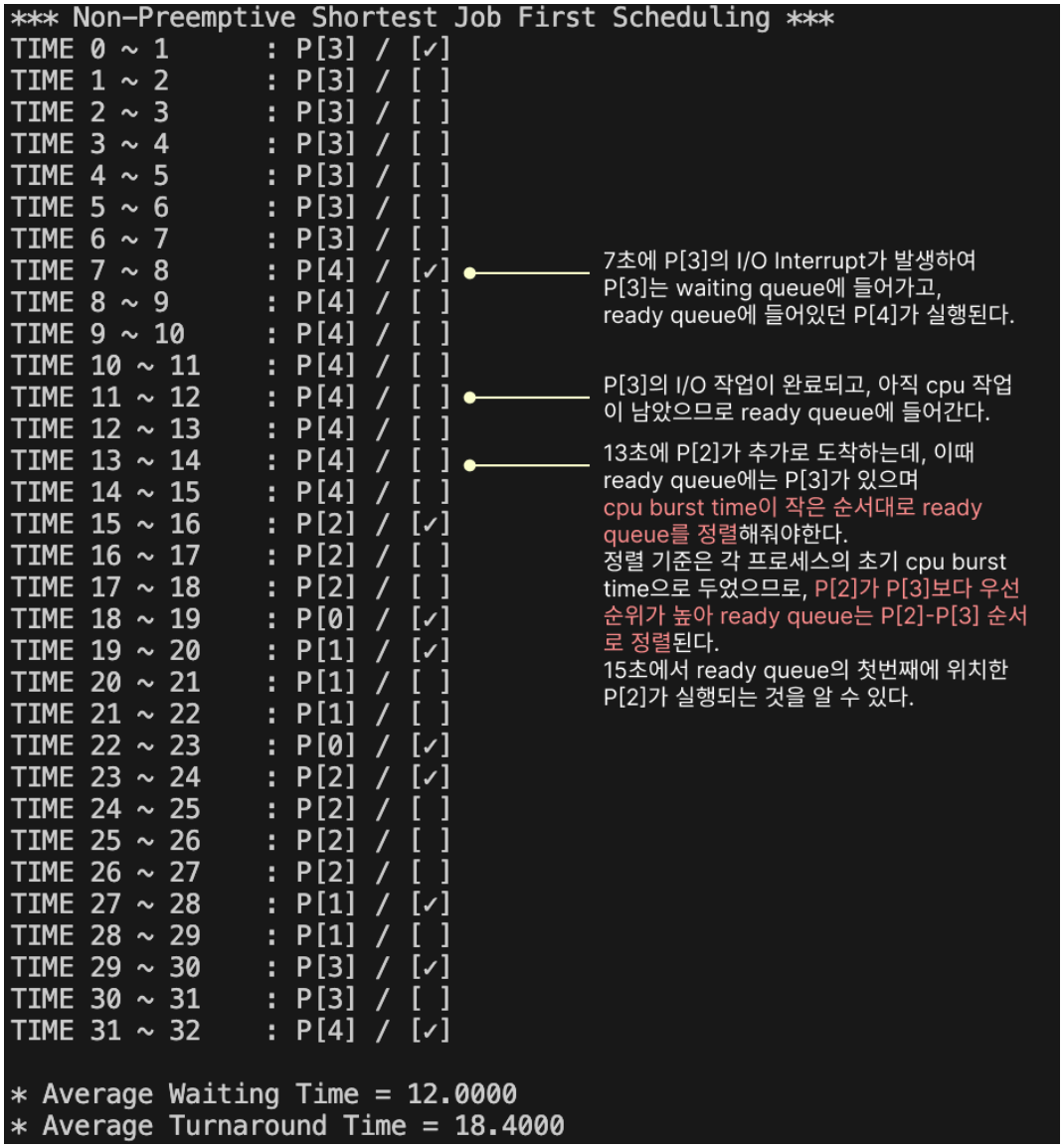


위 결과를 보면, 가장 먼저 들어온 프로세스부터 cpu 자원을 할당받아 작업이 진행되고,프로세스가 도착한 순서대로 ready queue에 들어가는 것을 알 수 있다. Running queue가 비게 되면, ready queue의 맨 처음에 위치한 프로세스부터 순서대로 꺼내서 자원을 할당해준다.

위의 프로세스들에 대해서는, running queue가 빌 때 ready queue에 대기중인 프로세스들이 항상 존재했기 때문에 cpu의 idle 상태는 발생하지 않았다.

프로세스들이 ready queue에서 평균적으로 대기한 시간은 10.4초이며, 각 프로세스가 종료될 때까지 걸린 시간은 평균적으로 16.8초이다.

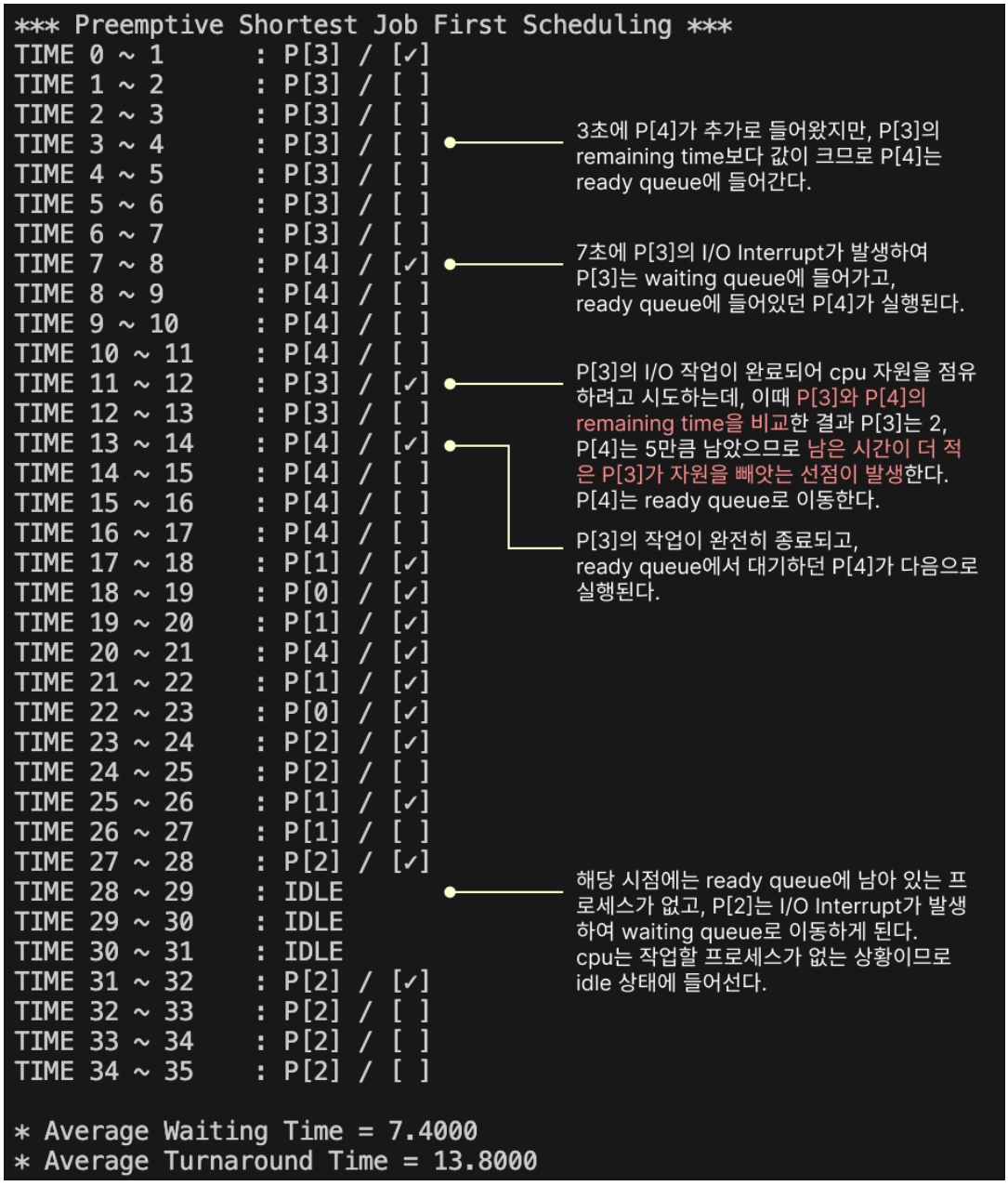
2. NP_SJF 스케줄링 실행결과



NP_SJF에서는 비선점이므로 새로운 프로세스가 도착했을 때 실행중인 프로세스가 있으면 그대로 진행하고, ready queue에 넣는 과정에서 작업 시간이 짧은 순서대로 정렬해준다.

프로세스들이 ready queue에서 평균적으로 대기한 시간은 12.0초이며, 각 프로세스가 종료될 때까지 걸린 시간은 평균적으로 18.4초이다.

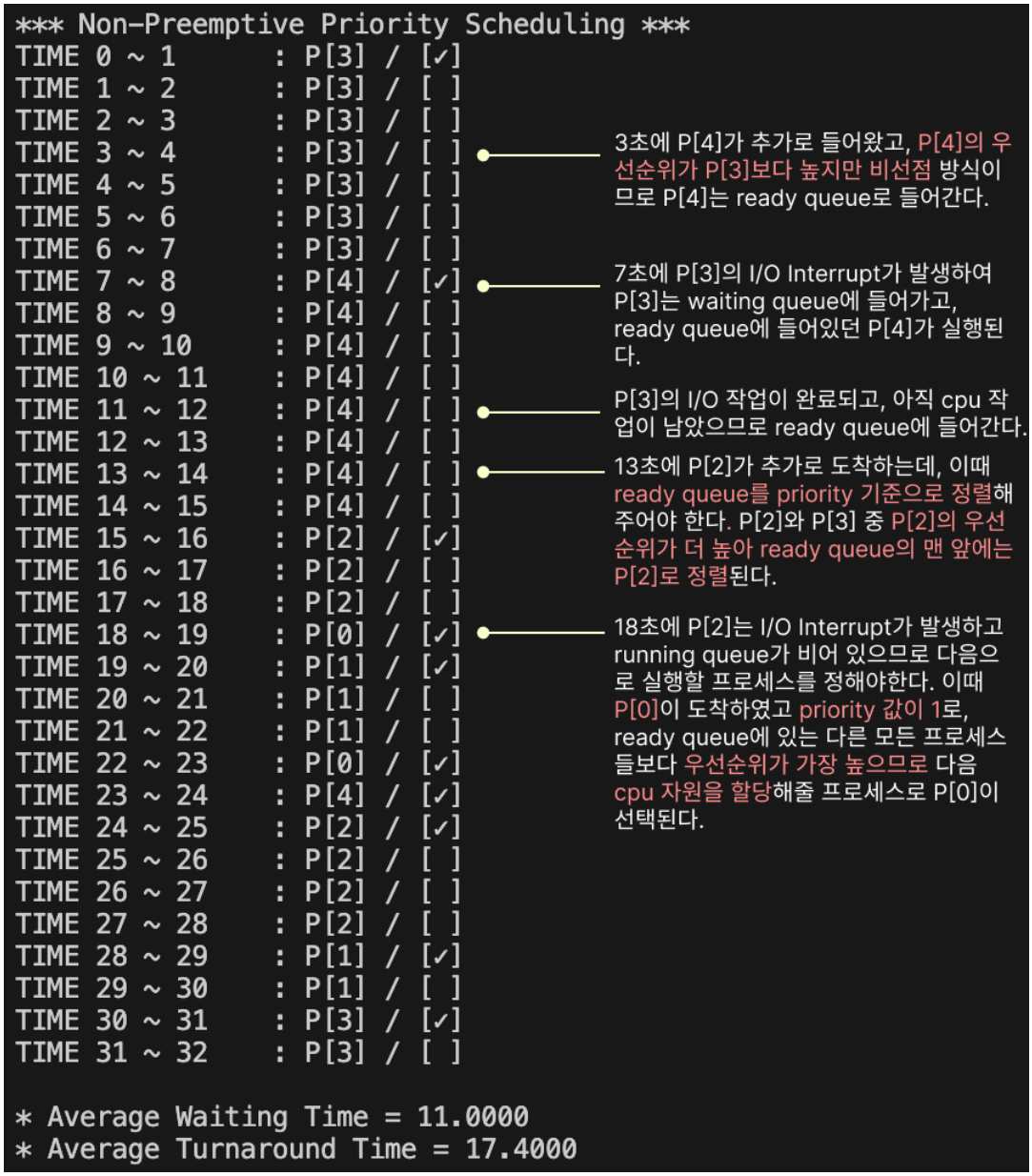
3. P_SJF 스케줄링 실행결과



P_SJF에서는 프로세스의 cpu 작업 remaining time이 작은 순서대로 먼저 실행되며, preemtive 하므로 진행중인 프로세스가 있더라도, 남은 작업 시간이 더 작으면 선점할 수 있다. 위 간트차트에서 10~11초 구간을 보면, P[4]가 cpu 자원을 점유하고 있었지만 새로 들어온 프로세스 P[3]과 비교하였을 때 P[4]가 작업 시간이 더 많이 남아 우선순위에서 밀려나고 결국 cpu 자원을 빼앗긴 것을 볼 수 있다.

프로세스들이 ready queue에서 평균적으로 대기한 시간은 7.4초이며, 각 프로세스가 종료될 때까지 걸린 시간은 평균적으로 13.8초임을 알 수 있다.

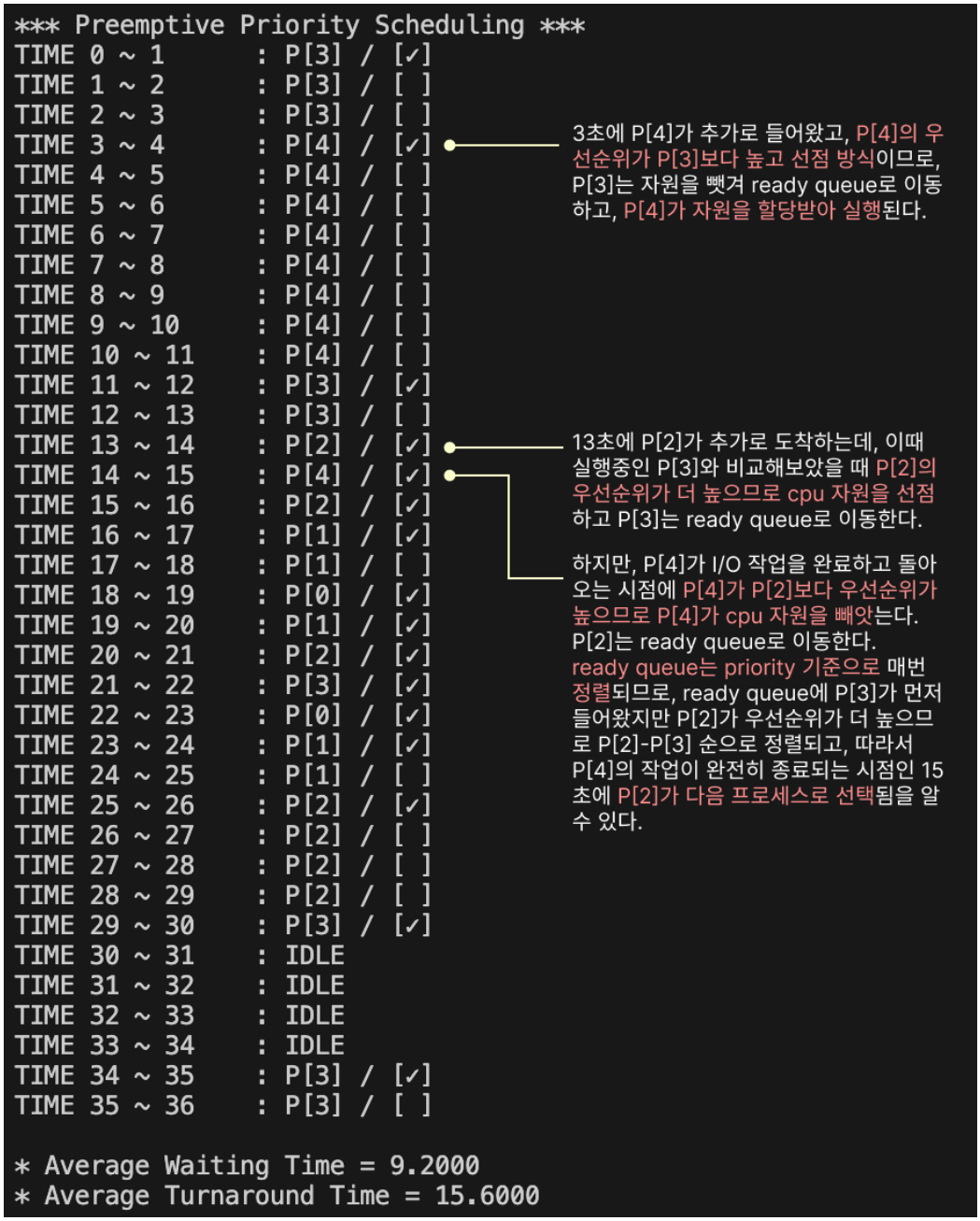
4. NP_PRIORITY 스케줄링 실행결과



NP_PRIORITY는 비선점이므로 새로운 프로세스가 도착했을 때 실행중인 프로세스가 있으면 그대로 진행하고, ready queue에 넣는 과정에서 우선순위가 더 높은 순서대로 정렬해준다. Priority의 값이 작을수록 우선순위가 더 높다고 보았다. 3초 시간대를 보면 우선순위가 더 높은 P[4]가 도착했지만, 비선점 방식이므로 자원을 빼앗지 않고 ready queue로 들어가 대기한다. 13초에서는 P[2]가 도착하였고 P[3]보다 우선순위가 높아 우선순위를 기준으로 ready queue가 정렬되고, 그 결과 15초에 P[2]가 다음 프로세스로 실행되는 것을 알 수 있다. 18초에는 다음으로 실행할 프로세스를 정하는데, P[0]이 우선순위가 가장 높아 그 다음으로 실행된다.

프로세스들이 ready queue에서 평균적으로 대기한 시간은 11초이며, 각 프로세스가 종료될 때까지 걸린 시간은 평균적으로 17.4초이다.

5. P_PRIORITY 스케줄링 실행결과

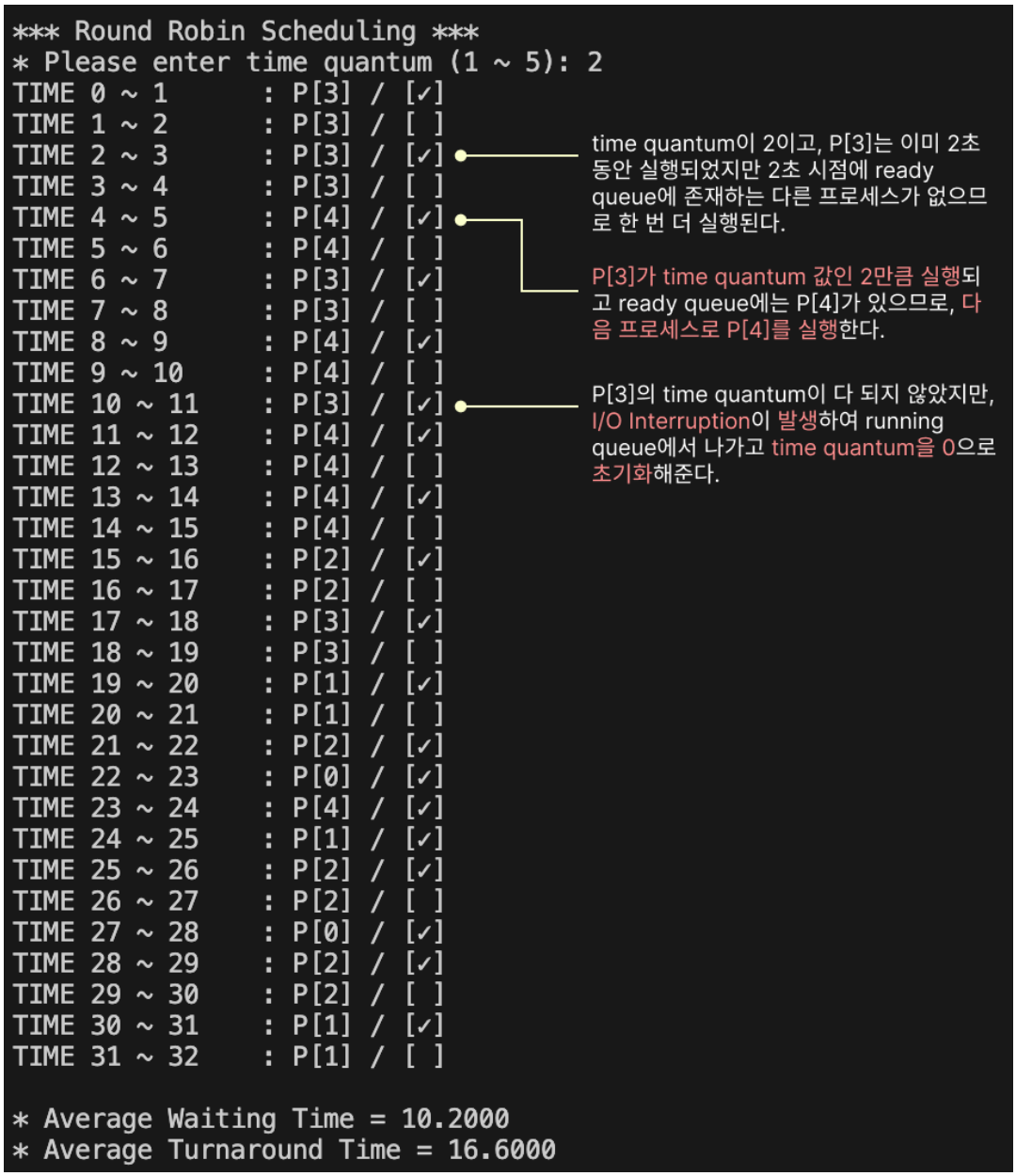


P_PRIORITY는 프로세스의 우선순위가 높은 순서대로 먼저 실행되며, preemtive하므로 진행중인 프로세스가 있더라도, 우선순위가 더 높으며 선점할 수 있다. ready queue에 들어갈 때는 우선순위를 기준으로 다시 정렬된다.

3초에 P[4]가 추가로 들어왔고, P[4]의 우선순위가 P[3]보다 높고 선점 방식이므로, P[3]는 자원을 뺏겨 ready queue로 이동하고, P[4]가 자원을 할당받아 실행되는 것을 볼 수 있다. 특히 12~15초 구간에서, 새로운 프로세스가 들어올 때, I/O 작업이 끝난 후 다시 돌아왔을 때, 작업이 끝났을 때 등 각 경우에 따라 선점 방식으로 다음 프로세스가 우선순위가 높은 것부터 실행되는 것을 잘 볼 수 있다.

프로세스들이 ready queue에서 평균적으로 대기한 시간은 9.2초이며, 각 프로세스가 종료될 때까지 걸린 시간은 평균적으로 15.6초이다.

6. ROUND_ROBIN 스케줄링 실행결과



ROUND_ROBIN은 FCFS 방식을 기본으로 하되, cpu 자원을 할당할 수 있는 고정 시간을 두어 프로세스들에게 자원을 할당해준다. FCFS 방식이 적용되므로 위의 실행결과를 보면 기본적으로 각 프로세스의 첫 실행 순서를 나열 했을 때 도착시간(arrival_time) 순서인 것을 알 수 있다(P[3]-P[4]-P[2]-P[1]-P[0]).

4초 시간대를 보면 기존 프로세스가 time quantum값인 2만큼 실행되고 그 다음 프로세스에게 자

원을 넘겨주는 것을 볼 수 있고, 10초에서는 진행중이던 프로세스가 I/O 작업을 시작하게 되면 time quantum값을 초기화시키고 그 다음 프로세스가 진행된다.

프로세스들이 ready queue에서 평균적으로 대기한 시간은 10.2초이며, 각 프로세스가 종료될 때까지 걸린 시간은 평균적으로 16.6초이다.

III. 전체 비교

```
***Evaluation of each scheduling algorithms***
--------------------------------------------------------------------------------
1. FCFS Scheduling
Average Waiting Time = 10.4000, Average Turnaround Time = 16.8000
CPU Utilization = 100.00%
--------------------------------------------------------------------------------
2. Non-Preemptive SJF Scheduling
Average Waiting Time = 12.0000, Average Turnaround Time = 18.4000
CPU Utilization = 100.00%
--------------------------------------------------------------------------------
3. Preemptive SJF Scheduling
Average Waiting Time = 7.4000, Average Turnaround Time = 13.8000
CPU Utilization = 91.43%
--------------------------------------------------------------------------------
4. Non-Preemptive Priority Scheduling
Average Waiting Time = 11.0000, Average Turnaround Time = 17.4000
CPU Utilization = 100.00%
--------------------------------------------------------------------------------
5. Preemptive Priority Scheduling
Average Waiting Time = 9.2000, Average Turnaround Time = 15.6000
CPU Utilization = 88.89%
--------------------------------------------------------------------------------
6. Round-Robin Scheduling
Average Waiting Time = 10.2000, Average Turnaround Time = 16.6000
CPU Utilization = 100.00%
--------------------------------------------------------------------------------
```

✧ CPU Utilization

CPU 활용도가 100%인 경우는 프로세스가 항상 CPU를 사용하고 있는 상태를 의미한다.

Non preemptive 방식의 경우 모두 cpu utilization이 100%인 것을 볼 수 있는데, 실제 상황에서는 시스템 호출 및 인터럽트 처리 시에 약간의 CPU 비활성화 시간이 발생할 수는 있지만, 현재 구현된 알고리즘 상에서는, ready queue에 항상 프로세스들이 존재하는 상황이라면 CPU가 idle 상태에 돌입하지는 않으므로 cpu utilization은 100%가 나온다.

반면, 주로 preemptive 스케줄링 기법에서는 cpu utilization이 100%에 못 미치는 것을 알 수 있다. preemptive sjf, preemptive priority 기법에서는 우선순위에 의거해 프로세스를 실행하는데, 우선순위가 낮을 경우 실행 중이더라도 자원을 뺏겨 작업 순서가 밀리게 된다. 이 경우 우선순위가 낮은 프로세스만 마지막까지 남아서 실행되므로, 그 프로세스가 i/o 작업이 남아 있고 ready queue에 다른 프로세스들은 존재하지 않는다면 cpu가 idle 상태에 놓이게 되기 때문이다. 일반적으로는, CPU 활용도가 낮은 것은 잦은 컨텍스트 스위칭과 관련이 있다.

✧ Average Waiting Time

최저: Preemptive SJF (7.4)

최고: Non-Preemptive SJF (12.0)

중간값: Preemptive Priority (9.2), FCFS (10.4), Round-Robin (10.2), Non-Preemptive Priority (11.0)

Preemptive SJF가 평균 대기 시간이 가장 낮으며, Non-Preemptive SJF가 가장 높습니다. 이는 Preemptive SJF가 프로세스의 잔여 시간을 기준으로 스케줄링하기 때문에 대기 시간이 최소화되는 반면, Non-Preemptive SJF는 새로운 프로세스가 도착해도 현재 실행 중인 프로세스를 끝까지 실행해야 하므로 대기 시간이 길어질 수 있기 때문이다.

✧ Average Turnaround Time

최저: Preemptive SJF (13.8)

최고: Non-Preemptive SJF (18.4)

중간값: Preemptive Priority (15.6), FCFS (16.8), Round-Robin (16.6), Non-Preemptive Priority (17.4)

평균 반환 시간 역시 Preemptive SJF가 가장 낮고, Non-Preemptive SJF가 가장 높습니다. 이는 앞서 언급한 대기 시간의 이유와 유사합니다. Preemptive SJF는 짧은 작업이 먼저 완료되도록 하여 반환 시간을 최소화한다.

3. 결론

CPU 스케줄링 비교 분석 결과

성능 최적화 측면에서는 Preemptive SJF가 비록 cpu utilization이 100%는 아니지만, 평균 대기 시간과 반환 시간을 최소화하여 성능 최적화에 가장 적합하다고 볼 수 있다. 하지만, 잦은 context-switching도 성능 저하에 영향을 미칠 수 있으므로 상황에 따라 적합하게 사용해야 한다.

공정성 측면에서 보자면, Round-Robin이 모든 프로세스에게 도착한 순서대로 공평한 CPU 시간을 제공해주기 때문에 가장 공정하다. 또한, FCFS는 구현이 가장 단순하며, 평균 대기 시간과 반환 시간이 중간 수준이기 때문에 단순 구현이 필요한 경우 적합하다고 볼 수 있다.

아쉬운 점

✧ I/O 작업이 없는 경우에 대한 고려

: 모든 프로세스에서 I/O 작업이 발생하도록 설정했는데, 실제 상황에서는 I/O 작업이 없는 경우도 많이 발생한다. 이를 고려하지 않은 점이 아쉽다. 특히 I/O가 없는 경우, FCFS 기법에서 기다리는 시간이 전반적으로 증가했을 것으로 예상된다. I/O가 없는 상황을 추가로 고려했으면, 각 스케줄링 기법의 성능을 더욱 다각도로 평가할 수 있었을 것이다.

✧ I/O 작업의 빈도

: I/O 작업을 전체 과정 중 한 번만 발생하도록 설정했는데, 한 번 이상의 I/O 작업이 발생할 수 있도록 설정했다면 더 현실적인 시뮬레이션이 되었을 것이다. 여러 번의 I/O 작업이 발생하는 경우 각 스케줄링 기법의 효율성을 더 잘 평가할 수 있었을 것 같다.

✧ 평가 척도의 다양화

: 성능 평가 척도로 평균 대기 시간, 평균 반환 시간, CPU 활용도만 사용했는데, 추가적으로 응답 시간(response time) 등과 같은 다른 평가 척도들도 포함했다면, 각 기법을 더 다양한 관점에서 비교할 수 있었을 것이다.

✧ 추가적인 스케줄링 알고리즘

: 기본 스케줄링 기법 외에 다른 알고리즘을 추가적으로 구현하지 않은 점이 아쉽다. 예를 들어, Aging 기법을 도입하면 우선순위 스케줄링에서 발생할 수 있는 기아(starvation) 문제를 예방할 수 있었을 것이다.

소감

이번 프로젝트를 통해 컴퓨터 과학 분야에서 중요한 주제인 운영체제, 특히 그 핵심이라 할 수 있는 CPU 스케줄링에 대해 깊이 있게 탐구하고 고민할 기회를 가질 수 있었다. 단순히 이론적으로 알고리즘의 존재와 작동 방식을 공부하는 것과, 직접 코드를 작성하고 구현해보는 것은 큰 차이가 있다고 느꼈다. 실제 구현을 통해 각 알고리즘의 미묘한 차이와 실제 상황에서의 동작을 훨씬 더 깊이 이해할 수 있어서 뜻 깊은 프로젝트였던 것 같다. 이번 프로젝트를 수행하면서 CPU 스케줄링의 중요성을 다시 한 번 느끼게 되었으며, 시스템 성능 최적화에 대해 다시 생각해보는 계기가 되었다.

APPENDIX

scheduler.h

```
#ifndef __SCHEDULER_H__
#define __SCHEDULER_H__

#define TRUE    1
#define FALSE   0

/**
 * Process
*/
typedef struct {
    int pid;
    int arrival_time;
    int burst_time;
    int io_burst_time;
    int io_start_time;
    int priority;
    int remaining_time;
    int io_remaining_time;
    int completed_time;
    int waiting_time;
    int turnaround_time;
    int progress_time;
    _Bool entered;
    int order; // np_sjf
    int preemptive; // p_sjf
    int time_quantum; // RR
} Process;

/**
 * Algorithm Evaluation
*/
typedef struct _evaluation {
  float avg_turnaround_time;
  float avg_waiting_time;
  int idle_time;
  int finished_process;
  int finished_time;
} Evaluation;

/**
 * Queue
*/
typedef int Data;

typedef struct _node {
  Data data;
  struct _node *next;
} Node;
```

```
typedef struct _Queue {
  Node *front;
  Node *rear;
} Queue;

#endif
```

scheduler.c

```
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>
#include <time.h>
#include "scheduler.h"

// Process
#define MAX_PROCESS_NUM 10

int process_num = 0;
Process *processes[MAX_PROCESS_NUM];

// Algorithm Evaluation
Evaluation _FCFS, _NP_SJF, _P_SJF, _NP_PRIORITY, _P_PRIORITY, _ROUND_ROBIN;

// Queue
Queue ready_queue, running_queue;
int waiting_queue[MAX_PROCESS_NUM];

// Functions related to Queue
void QueueInit(Queue *pq) {
  pq->front = NULL;
  pq->rear = NULL;
}

int IsQueueEmpty(Queue *pq) {
  if(pq->front == NULL) return TRUE; // if true, return 1
  else return FALSE; // if false, return 0
}

void Enqueue(Queue *pq, Data data) {
  Node *newNode = (Node*)malloc(sizeof(Node));
  newNode->next = NULL;
  newNode->data = data;

  if(IsQueueEmpty(pq)) {
    pq->front = newNode;
    pq->rear = newNode;
  } else {
    // 기존 맨 마지막 노드의 다음으로 연결
    pq->rear->next = newNode;
    // 맨 마지막 노드로 설정
    pq->rear = newNode;
  }
}
```

```
Data Dequeue(Queue *pq) {
  Node *delNode;
  Data retProcess;

  if(IsQueueEmpty(pq)) {
    printf("Dequeue Memory Error");
    exit(-1);
  }

  delNode = pq->front;
  retProcess = delNode->data;
  pq->front = pq->front->next;

  free(delNode);
  return retProcess;
}

Data QPeek(Queue * pq) {
        if (IsQueueEmpty(pq)) {
                printf("Queue Memory Error![peek]");
                exit(-1);
        }
        return pq->front->data;
}

// Create Processes
void create_processes() {
  printf("*** CPU SCHEDULER ***\n");
  printf("How many processes do you want to create?: ");
  while (1)
  {
    scanf("%d", &process_num);
    if( process_num > MAX_PROCESS_NUM || process_num < 0) printf("[ERROR]
Number of processes must be 1 to 10. Please enter the number of processes
again: ");
    else break;
  }

        srand(time(NULL));

        int found[10]; int pri, flag;

  for(int i = 0; i < process_num; i++) {
    Process *process = (Process*)malloc(sizeof(Process) * 1);
    process->pid = i;
    process->burst_time = (int)(rand() % 9 + 2); // 2 ~ 10
    process->io_burst_time = (int)(rand() % 5 + 1); // 1 ~ 6
    process->arrival_time = (int)(rand() % (4 * process_num));
    process->io_start_time = (int)(rand() % (process->burst_time - 1) + 1); //
1 ~ (cpu burst time - 1)
    process->remaining_time = process->burst_time;
    process->io_remaining_time = process->io_burst_time;
    process->entered = FALSE;
    process->completed_time = 0; // 완료된 시각
    process->progress_time = 0; // 진행 정도
```

```
    // priority
          while (1) {
                        pri = (rand() % process_num + 1);
                        flag = 0;
                        for (int j = 0; j <= i; j++) {
                                if (pri != found[j]) { flag++; }
                        }
                        if (flag == i + 1) { found[i] = pri; break; }
                }
                process->priority = found[i];

    processes[i] = process;
  };

        // 적어도 하나의 프로세스는 0 초에 도착하도록 보장
  processes[(int)(rand() % process_num)]->arrival_time = 0;


        printf("------------------------------------------------------------------
-------------------------\n");
        printf("PID\t\tArrival Time\tCPU Burst Time\tI/O Burst Time\tI/O Start
Time\tPriority\n");
        printf("------------------------------------------------------------------
-------------------------\n");

  for (int i = 0; i < process_num; i++) {
    printf("%d\t\t%d\t\t%d\t\t%d\t\t%d\t\t%d\n",
      processes[i]->pid,
      processes[i]->arrival_time,
      processes[i]->burst_time,
      processes[i]->io_burst_time,
      processes[i]->io_start_time,
      processes[i]->priority);
  }


        printf("------------------------------------------------------------------
-------------------------\n");
        printf("* Finished creating processes!\n\n");
}

// queue init
void Config() {
  QueueInit(&ready_queue);
  QueueInit(&running_queue);
  for(int i = 0; i < process_num; i++) {
    waiting_queue[i] = -1;
  }
        printf("* Finished creating queues!\n\n");
}

// initialize progress_time, completed_time of processes for each algorithm
void initializeProcess() {
  for (int i = 0; i < process_num; i++) {
    waiting_queue[i] = -1;
```

```
    processes[i]->progress_time = 0;
    processes[i]->completed_time = 0;

  }
}

/**
 * FCFS
*/
void FCFS() {
        printf("*** FCFS Scheduling ***\n");
  // init
  int time;
  Data selected;
  _FCFS.idle_time = 0; _FCFS.finished_process = 0;
  initializeProcess();

  for(time = 0; _FCFS.finished_process != process_num; time++) {
    for(int i = 0; i < process_num; i++) {
      // 도착한 프로세스가 있는지 확인 후 enqueue
      if(time == processes[i]->arrival_time) {
        Enqueue(&ready_queue, processes[i]->pid);
      }
      // 대기큐 - io 작업 시간 업데이트
      if(waiting_queue[i] > 0) {
        waiting_queue[i]--;
      }
      // io 작업이 끝나면 준비큐로 넘겨주기
      if(waiting_queue[i] == 0) {
        Enqueue(&ready_queue, i);
        waiting_queue[i]--;
      }
    }
    // ready queue 에 프로세스가 있고, running 중인 것이 없을 때
    if(!IsQueueEmpty(&ready_queue) && IsQueueEmpty(&running_queue)) {
      selected = Dequeue(&ready_queue);
      Enqueue(&running_queue, selected);
      processes[selected]->entered = TRUE;
    }

    /**
     * i/o 상태일 때 ready queue 에 있는 프로세스 실행. ready queue 에 하나도
없으면 cpu 는 idle 상태
    */
    if (IsQueueEmpty(&running_queue)) {
      printf("TIME %d ~ %d\t: IDLE\n", time, time + 1); _FCFS.idle_time++;
    }
    else {
      printf("TIME %d ~ %d\t: P[%d] / [%s] \n", time, time + 1,
processes[selected]->pid, processes[selected]->entered == TRUE ? "✓" : " ");
      if(processes[selected]->entered == TRUE) {
        processes[selected]->entered = FALSE;
      }
      processes[selected]->progress_time++;
```

```
                        if (processes[selected]->progress_time ==
processes[selected]->io_start_time) {
                                int waiting = Dequeue(&running_queue);
                                waiting_queue[waiting] = processes[waiting]-
>io_burst_time + 1;
                        }
                        else if (processes[selected]->progress_time ==
processes[selected]->burst_time) {
                                _FCFS.finished_process++;
                                processes[Dequeue(&running_queue)]-
>completed_time = time + 1;
                        }
    }
  }
  _FCFS.finished_time = time;

  // Evaluation
  int total_turnaround_time = 0, total_burst_time = 0;
  for(int i = 0; i < process_num; i++) {
    total_turnaround_time += processes[i]->completed_time - processes[i]-
>arrival_time;
    total_burst_time += processes[i]->burst_time;
  }
  _FCFS.avg_turnaround_time = (float)total_turnaround_time / process_num;
  _FCFS.avg_waiting_time = (float)(total_turnaround_time - total_burst_time) /
process_num;


        printf("\n* Average Waiting Time = %.4f", _FCFS.avg_waiting_time);
        printf("\n* Average Turnaround Time = %.4f\n",
_FCFS.avg_turnaround_time);
        printf("*****************************************************************
**************\n\n");
}


int compare_burst_time(const void *a, const void *b) {
    Process *p1 = (Process *)a;
    Process *p2 = (Process *)b;
    return p2->burst_time - p1->burst_time;
}


void NP_SJF() {
        printf("*** Non-Preemptive Shortest Job First Scheduling ***\n");
  int selected, time, temp, i, j;
  _NP_SJF.idle_time = 0;
  Queue tempQ;
  initializeProcess();
  QueueInit(&tempQ);

  // 각 프로세스의 PID 와 CPU burst time 을 arr 에 저장
  int arr[10][2] = {0};
  for(i = 0; i < process_num; i++) {
    arr[i][0] = i;
```

```
  arr[i][1] = processes[i]->burst_time;
};

// bust time 순으로 정렬
for (i = 0; i < process_num; i++) {
  for (j = j + 1; j < process_num; j++) {
    if(arr[i][1] > arr[j][1]) {
      temp = arr[i][0]; arr[i][0] = arr[j][0]; arr[j][0] = temp;
      temp = arr[i][1]; arr[i][1] = arr[j][1]; arr[j][1] = temp;
    }
  }
}

// process order
for(i = 0; i < process_num; i++) {
  for (j = 0; j <process_num; j++) {
    if(processes[i]->pid == arr[j][0]) {
      processes[i]->order = j;
    }
  }
}

for(time = 0; _NP_SJF.finished_process != process_num; time++) {
  for(i = 0; i < process_num; i++) {
    if(waiting_queue[i] > 0) {
      waiting_queue[i]--;
    }
    if(waiting_queue[i] == 0 || time == processes[i]->arrival_time) {
      if(waiting_queue[i] == 0) {
        waiting_queue[i]--;
      }
      if(IsQueueEmpty(&ready_queue)) {
        Enqueue(&ready_queue, processes[i]->pid);
      }
      // 임시 큐를 사용하여 적절한 위치에 프로세스를 삽입한 후 다시 원래 큐로 복원
      else {
        while (processes[QPeek(&ready_queue)]->order < processes[i]->order) {
          Enqueue(&tempQ, Dequeue(&ready_queue));
          if(IsQueueEmpty(&ready_queue)) break;
        }
        Enqueue(&tempQ, processes[i]->pid);
        while (!IsQueueEmpty(&ready_queue)) {
          Enqueue(&tempQ, Dequeue(&ready_queue));
        }
        while (!IsQueueEmpty(&tempQ)) {
          Enqueue(&ready_queue, Dequeue(&tempQ));
        }
      }
    }
  }

  if(!IsQueueEmpty(&ready_queue) && IsQueueEmpty(&running_queue)) {
    selected = Dequeue(&ready_queue);
    Enqueue(&running_queue, selected);
    processes[selected]->entered = TRUE;
  }
```

```
    /**
     * i/o 상태일 때 ready queue 에 있는 프로세스 실행. ready queue 에 하나도
없으면 cpu 는 idle 상태
    */
    if (IsQueueEmpty(&running_queue)) {
      printf("TIME %d ~ %d\t: IDLE\n", time, time + 1); _NP_SJF.idle_time++;
    }
    else {
      printf("TIME %d ~ %d\t: P[%d] / [%s] \n", time, time + 1,
processes[selected]->pid, processes[selected]->entered == TRUE ? "✓" : " ");
      if(processes[selected]->entered == TRUE) {
        processes[selected]->entered = FALSE;
      }
      processes[selected]->progress_time++;


                          if (processes[selected]->progress_time ==
processes[selected]->io_start_time) {
                                  int waiting = Dequeue(&running_queue);
                                  waiting_queue[waiting] = processes[waiting]-
>io_burst_time + 1;
                          }
                          else if (processes[selected]->progress_time ==
processes[selected]->burst_time) {
                                  _NP_SJF.finished_process++;
                                  processes[Dequeue(&running_queue)]-
>completed_time = time + 1;
                          }
    }
  }

  _NP_SJF.finished_time = time;

   // Evaluation
  int total_turnaround_time = 0, total_burst_time = 0;
  for(int i = 0; i < process_num; i++) {
    total_turnaround_time += processes[i]->completed_time - processes[i]-
>arrival_time;
    total_burst_time += processes[i]->burst_time;
  }
  _NP_SJF.avg_turnaround_time = (float)total_turnaround_time / process_num;
  _NP_SJF.avg_waiting_time = (float)(total_turnaround_time - total_burst_time)
/ process_num;


        printf("\n* Average Waiting Time = %.4f", _NP_SJF.avg_waiting_time);
        printf("\n* Average Turnaround Time = %.4f\n",
_NP_SJF.avg_turnaround_time);
        printf("*****************************************************************
**************\n\n");
}

void P_SJF() {
  printf("*** Preemptive Shortest Job First Scheduling ***\n");
```

```
  int selected, time, preempted, i, j;
  _P_SJF.idle_time = 0;
  _P_SJF.finished_process = 0;
  Queue tempQ;
  initializeProcess();
  QueueInit(&tempQ);

  for(time = 0; _P_SJF.finished_process != process_num; time++) {
    for(i = 0; i < process_num; i++) {
      if(waiting_queue[i] > 0) {
        waiting_queue[i]--;
      }

      if(waiting_queue[i] == 0 || time == processes[i]->arrival_time) {
        if(waiting_queue[i] == 0) {
          waiting_queue[i]--;
        }

        if(IsQueueEmpty(&ready_queue) && IsQueueEmpty(&running_queue)) {
          Enqueue(&ready_queue, processes[i]->pid);
        }
        // 선점 로직
        else if(!IsQueueEmpty(&running_queue)) {
          // 현재 진행중인 프로세스의 remaining burst time 보다 다른 프로세스의
remaining burst time 이 더 적은 경우 선점
          if(processes[i]->remaining_time < processes[QPeek(&running_queue)]-
>remaining_time) {
            preempted = Dequeue(&running_queue);
            while (!IsQueueEmpty(&ready_queue))
            {
             Enqueue(&tempQ, Dequeue(&ready_queue));
            }
            Enqueue(&ready_queue, preempted);
            while (!IsQueueEmpty(&tempQ))
            {
              Enqueue(&ready_queue, Dequeue(&tempQ));
            }
            // -> 선점 후 ready queue 에 있던 나머지 프로세스들 다시 넣어주는 것
            // 현재 더 우선순위의 프로세스를 running queue 에 넣어줌
            Enqueue(&running_queue, i);
            processes[i]->entered = TRUE;

            if(processes[preempted]->preemptive == 1) {
              processes[preempted]->preemptive = 0;
            }
          }
          else {
            if(IsQueueEmpty(&ready_queue)) {
              Enqueue(&ready_queue, i);
            }
            // 현재 작업중인 프로세스보다는 후순위인데, ready queue 에서 순서 조정이
다시 필요
            else {
              while (processes[QPeek(&ready_queue)]->remaining_time <
processes[i]->remaining_time) {
```

```
              Enqueue(&tempQ, Dequeue(&ready_queue));
              if(IsQueueEmpty(&ready_queue)) {
                break;
              }
            }
            // ready queue 보다 우선순위이면 temp 에 넣어주기
            Enqueue(&tempQ, processes[i]->pid);
            // 나머지 프로세스들 tempQ 로 이동
            while (!IsQueueEmpty(&ready_queue)) {
              Enqueue(&tempQ, Dequeue(&ready_queue));
            }
            // tempQ 에서 다시 ready queue 로 이동
            while(!IsQueueEmpty(&tempQ)) {
              Enqueue(&ready_queue, Dequeue(&tempQ));
            }
          }
        }
      }

      // ready queue 에는 값이 있고 running queue 에는 값이 없을 때
      // ready queue 우선순위로 정렬
      else {
        while(processes[QPeek(&ready_queue)]->remaining_time < processes[i]-
>remaining_time) {
          Enqueue(&tempQ, Dequeue(&ready_queue));
          if(IsQueueEmpty(&ready_queue)) break;
        };
        Enqueue(&tempQ, processes[i]->pid);
        // 나머지 프로세스들 tempQ 로 이동
        while (!IsQueueEmpty(&ready_queue)) {
          Enqueue(&tempQ, Dequeue(&ready_queue));
        }
        // tempQ 에서 다시 ready queue 로 이동
        while(!IsQueueEmpty(&tempQ)) {
          Enqueue(&ready_queue, Dequeue(&tempQ));
        }
      }
    }
  }

  if(!IsQueueEmpty(&running_queue)) {
    selected = QPeek(&running_queue);
  }

  if(!IsQueueEmpty(&ready_queue) && IsQueueEmpty(&running_queue)) {
    selected = Dequeue(&ready_queue);
    Enqueue(&running_queue, selected);
    processes[selected]->entered = TRUE;
    processes[selected]->preemptive = 1;
  }

  /**
   * i/o 상태일 때 ready queue 에 있는 프로세스 실행. ready queue 에 하나도
없으면 cpu 는 idle 상태
  */
```

```
    if (IsQueueEmpty(&running_queue)) {
      printf("TIME %d ~ %d\t: IDLE\n", time, time + 1); _P_SJF.idle_time++;
    }
    else {
      printf("TIME %d ~ %d\t: P[%d] / [%s] \n", time, time + 1,
processes[selected]->pid, processes[selected]->entered == TRUE ? "✓" : " ");
      if(processes[selected]->entered == TRUE) {
        processes[selected]->entered = FALSE;
      }
      processes[selected]->remaining_time--;
      processes[selected]->progress_time++;


                        if (processes[selected]->progress_time ==
processes[selected]->io_start_time) {
                                int waiting = Dequeue(&running_queue);
                                waiting_queue[waiting] = processes[waiting]-
>io_burst_time + 1;
                        }
                        else if (processes[selected]->progress_time ==
processes[selected]->burst_time) {
                                _P_SJF.finished_process++;
                                processes[Dequeue(&running_queue)]-
>completed_time = time + 1;
                        }
    }
  }

  _P_SJF.finished_time = time;

  // Evaluation
  int total_turnaround_time = 0, total_burst_time = 0;
  for(int i = 0; i < process_num; i++) {
    total_turnaround_time += processes[i]->completed_time - processes[i]-
>arrival_time;
    total_burst_time += processes[i]->burst_time;
  }
  _P_SJF.avg_turnaround_time = (float)total_turnaround_time / process_num;
  _P_SJF.avg_waiting_time = (float)(total_turnaround_time - total_burst_time) /
process_num;


        printf("\n* Average Waiting Time = %.4f", _P_SJF.avg_waiting_time);
        printf("\n* Average Turnaround Time = %.4f\n",
_P_SJF.avg_turnaround_time);
        printf("*************************************************************
**************\n\n");
}

/**
 * NP_PRIORITY
*/
void NP_PRIORITY() {
  printf("*** Non-Preemptive Priority Scheduling ***\n");
  int selected, time, i;
  _NP_PRIORITY.idle_time = 0;
```

```
  _NP_PRIORITY.finished_process = 0;
  Queue tempQ;
  initializeProcess();
  QueueInit(&tempQ);

  for(time = 0; _NP_PRIORITY.finished_process != process_num; time++) {
    for(i = 0; i < process_num; i++) {
        if(waiting_queue[i] > 0) {
        waiting_queue[i]--;
      }

      if(waiting_queue[i] == 0 || time == processes[i]->arrival_time) {
        if(waiting_queue[i] == 0) {
          waiting_queue[i]--;
        }

        if(IsQueueEmpty(&ready_queue)) {
          Enqueue(&ready_queue, processes[i]->pid);
        }
        else {
          // ready queue 에서 현재 프로세스보다 우선순위 높은 것들 먼저 넣기
          while (processes[QPeek(&ready_queue)]->priority < processes[i]-
>priority) {
            Enqueue(&tempQ, Dequeue(&ready_queue));
            if(IsQueueEmpty(&ready_queue)) {
              break;
            }
          }
          // 현재 프로세스 넣기
          Enqueue(&tempQ, processes[i]->pid);
          // 현재 프로세스보다 우선순위 낮은 나머지 프로세스들 넣기
          while (!IsQueueEmpty(&ready_queue)) {
            Enqueue(&tempQ, Dequeue(&ready_queue));
          }
          while (!IsQueueEmpty(&tempQ)) {
            Enqueue(&ready_queue, Dequeue(&tempQ));
          }
        }
      }
    }
    if(!IsQueueEmpty(&ready_queue) && IsQueueEmpty(&running_queue)) {
      selected = Dequeue(&ready_queue);
      Enqueue(&running_queue, selected);
      processes[selected]->entered = TRUE;
    }
    if(IsQueueEmpty(&running_queue)) {
      printf("TIME %d ~ %d\t: IDLE\n", time, time + 1);
_NP_PRIORITY.idle_time++;
    }
    else {
      printf("TIME %d ~ %d\t: P[%d] / [%s] \n", time, time + 1,
processes[selected]->pid, processes[selected]->entered == TRUE ? "✓" : " ");
      if(processes[selected]->entered == TRUE) {
        processes[selected]->entered = FALSE;
      }
```

```
      processes[selected]->progress_time++;


      if (processes[selected]->progress_time == processes[selected]-
>io_start_time) {
        int waiting = Dequeue(&running_queue);
        waiting_queue[waiting] = processes[waiting]->io_burst_time + 1;
      }
      else if (processes[selected]->progress_time == processes[selected]-
>burst_time) {
        _NP_PRIORITY.finished_process++;
        processes[Dequeue(&running_queue)]->completed_time = time + 1;
      }
    }
  }
  _NP_PRIORITY.finished_time = time;

  // Evaluation
  int total_turnaround_time = 0, total_burst_time = 0;
  for(int i = 0; i < process_num; i++) {
    total_turnaround_time += processes[i]->completed_time - processes[i]-
>arrival_time;
    total_burst_time += processes[i]->burst_time;
  }
  _NP_PRIORITY.avg_turnaround_time = (float)total_turnaround_time /
process_num;
  _NP_PRIORITY.avg_waiting_time = (float)(total_turnaround_time -
total_burst_time) / process_num;


        printf("\n* Average Waiting Time = %.4f",
_NP_PRIORITY.avg_waiting_time);
        printf("\n* Average Turnaround Time = %.4f\n",
_NP_PRIORITY.avg_turnaround_time);
        printf("*****************************************************************
**************\n\n");
}

void P_PRIORITY() {
  printf("*** Preemptive Priority Scheduling ***\n");
  int selected, time, i, preempted;
  _P_PRIORITY.idle_time = 0;
  _P_PRIORITY.finished_process = 0;
  Queue tempQ;
  initializeProcess();
  QueueInit(&tempQ);

  for(time = 0; _P_PRIORITY.finished_process != process_num; time++) {
    for(i = 0; i < process_num; i++) {
      if(waiting_queue[i] > 0) {
        waiting_queue[i]--;
      }

      if(waiting_queue[i] == 0 || time == processes[i]->arrival_time) {
        if(waiting_queue[i] == 0) {
          waiting_queue[i]--;
```

```
        }

        if(IsQueueEmpty(&ready_queue) && IsQueueEmpty(&running_queue)) {
          Enqueue(&ready_queue, processes[i]->pid);
        }
        // 선점 로직
        else if(!IsQueueEmpty(&running_queue)) {
          // 현재 진행중인 프로세스의 remaining burst time 보다 다른 프로세스의
remaining burst time 이 더 적은 경우 선점
          if(processes[i]->priority < processes[QPeek(&running_queue)]-
>priority) {
            preempted = Dequeue(&running_queue);
            while (!IsQueueEmpty(&ready_queue))
            {
             Enqueue(&tempQ, Dequeue(&ready_queue));
            }
            Enqueue(&ready_queue, preempted);
            while (!IsQueueEmpty(&tempQ))
            {
              Enqueue(&ready_queue, Dequeue(&tempQ));
            }
            // -> 선점 후 ready queue 에 있던 나머지 프로세스들 다시 넣어주는 것
            // 현재 더 우선순위의 프로세스를 running queue 에 넣어줌
            Enqueue(&running_queue, i);
            processes[i]->entered = TRUE;

            if(processes[preempted]->preemptive == 1) {
              processes[preempted]->preemptive = 0;
            }
          }
          else {
            if(IsQueueEmpty(&ready_queue)) {
              Enqueue(&ready_queue, i);
            }
            else {
              while (processes[QPeek(&ready_queue)]->priority < processes[i]-
>priority) {
                Enqueue(&tempQ, Dequeue(&ready_queue));
                if(IsQueueEmpty(&ready_queue)) {
                  break;
                }
              }
              Enqueue(&tempQ, processes[i]->pid);
              // 나머지 프로세스들 tempQ 로 이동
              while (!IsQueueEmpty(&ready_queue)) {
                Enqueue(&tempQ, Dequeue(&ready_queue));
              }
              // tempQ 에서 다시 ready queue 로 이동
              while(!IsQueueEmpty(&tempQ)) {
                Enqueue(&ready_queue, Dequeue(&tempQ));
              }
            }
          }
        }
```

```
        // ready queue 에는 값이 있고 running queue 에는 값이 없을 때
        // ready queue 우선순위로 정렬
        else {
          while(processes[QPeek(&ready_queue)]->priority < processes[i]-
>priority) {
            Enqueue(&tempQ, Dequeue(&ready_queue));
            if(IsQueueEmpty(&ready_queue)) break;
          };
          Enqueue(&tempQ, processes[i]->pid);
          // 나머지 프로세스들 tempQ 로 이동
          while (!IsQueueEmpty(&ready_queue)) {
            Enqueue(&tempQ, Dequeue(&ready_queue));
          }
          // tempQ 에서 다시 ready queue 로 이동
          while(!IsQueueEmpty(&tempQ)) {
            Enqueue(&ready_queue, Dequeue(&tempQ));
          }
        }
      }
    }


    if(!IsQueueEmpty(&running_queue)) {
      selected = QPeek(&running_queue);
    }

    if(!IsQueueEmpty(&ready_queue) && IsQueueEmpty(&running_queue)) {
      selected = Dequeue(&ready_queue);
      Enqueue(&running_queue, selected);
      processes[selected]->entered = TRUE;
      processes[selected]->preemptive = 1;
    }

    /**
     * i/o 상태일 때 ready queue 에 있는 프로세스 실행. ready queue 에 하나도
없으면 cpu 는 idle 상태
    */
    if (IsQueueEmpty(&running_queue)) {
      printf("TIME %d ~ %d\t: IDLE\n", time, time + 1);
_P_PRIORITY.idle_time++;
    }
    else {
      printf("TIME %d ~ %d\t: P[%d] / [%s] \n", time, time + 1,
processes[selected]->pid, processes[selected]->entered == TRUE ? "✓" : " ");
      if(processes[selected]->entered == TRUE) {
        processes[selected]->entered = FALSE;
      }
      processes[selected]->progress_time++;


                          if (processes[selected]->progress_time ==
processes[selected]->io_start_time) {
                                int waiting = Dequeue(&running_queue);
                                waiting_queue[waiting] = processes[waiting]-
>io_burst_time + 1;
```

```
                        }
                        else if (processes[selected]->progress_time ==
processes[selected]->burst_time) {
                                _P_PRIORITY.finished_process++;
                                processes[Dequeue(&running_queue)]-
>completed_time = time + 1;
                        }
    }
  }

  _P_PRIORITY.finished_time = time;

  // Evaluation
  int total_turnaround_time = 0, total_burst_time = 0;
  for(int i = 0; i < process_num; i++) {
    total_turnaround_time += processes[i]->completed_time - processes[i]-
>arrival_time;
    total_burst_time += processes[i]->burst_time;
  }
  _P_PRIORITY.avg_turnaround_time = (float)total_turnaround_time / process_num;
  _P_PRIORITY.avg_waiting_time = (float)(total_turnaround_time -
total_burst_time) / process_num;


        printf("\n* Average Waiting Time = %.4f",
_P_PRIORITY.avg_waiting_time);
        printf("\n* Average Turnaround Time = %.4f\n",
_P_PRIORITY.avg_turnaround_time);
        printf("****************************************************************
**************\n\n");

}

void ROUND_ROBIN() {
  printf("*** Round Robin Scheduling ***\n");
  int selected = -1, time, i;
  _ROUND_ROBIN.idle_time = 0;
  _ROUND_ROBIN.finished_process = 0;
  Queue tempQ;
  initializeProcess();
  QueueInit(&tempQ);

  int time_quantum = 0;
  printf("* Please enter time quantum (1 ~ 5): ");
  while (1) {
    scanf("%d", &time_quantum);
    if (time_quantum > 5 || time_quantum < 1)
      printf("[ERROR] Time Quantum must be 1 to 5. Please enter time quantum
again: ");
    else
      break;
  }

  for (time = 0; _ROUND_ROBIN.finished_process != process_num; time++) {
    for (i = 0; i < process_num; i++) {
      if (time == processes[i]->arrival_time) {
```

```
        Enqueue(&ready_queue, processes[i]->pid);
      }
      if (waiting_queue[i] > 0) {
        waiting_queue[i]--;
        if (waiting_queue[i] == 0) {
          Enqueue(&ready_queue, i);
        }
      }
    }

    if (!IsQueueEmpty(&ready_queue) && IsQueueEmpty(&running_queue)) {
      selected = Dequeue(&ready_queue);
      Enqueue(&running_queue, selected);
      processes[selected]->entered = TRUE;
    }

    if (IsQueueEmpty(&running_queue)) {
      printf("TIME %d ~ %d\t: IDLE\n", time, time + 1);
      _ROUND_ROBIN.idle_time++;
    } else {
      printf("TIME %d ~ %d\t: P[%d] / [%s] \n", time, time + 1,
processes[selected]->pid, processes[selected]->entered == TRUE ? "✓" : " ");
      if (processes[selected]->entered == TRUE) {
        processes[selected]->entered = FALSE;
      }
      processes[selected]->progress_time++;
      processes[selected]->time_quantum++;

      if (processes[selected]->progress_time == processes[selected]-
>io_start_time) {
        int waiting = Dequeue(&running_queue);
        waiting_queue[waiting] = processes[waiting]->io_burst_time + 1;
        processes[waiting]->time_quantum = 0;
        selected = -1;
      } else if (processes[selected]->progress_time == processes[selected]-
>burst_time) {
        _ROUND_ROBIN.finished_process++;
        processes[Dequeue(&running_queue)]->completed_time = time + 1;
        selected = -1;
      } else if (processes[selected]->time_quantum == time_quantum) {
        processes[selected]->time_quantum = 0;
        Enqueue(&ready_queue, Dequeue(&running_queue));
        selected = -1;
      }
    }
  }
  _ROUND_ROBIN.finished_time = time;

  // Evaluation
  int total_turnaround_time = 0, total_burst_time = 0;
  for (int i = 0; i < process_num; i++) {
    total_turnaround_time += processes[i]->completed_time - processes[i]-
>arrival_time;
    total_burst_time += processes[i]->burst_time;
  }
```

```
  _ROUND_ROBIN.avg_turnaround_time = (float)total_turnaround_time /
process_num;
  _ROUND_ROBIN.avg_waiting_time = (float)(total_turnaround_time -
total_burst_time) / process_num;

  printf("\n* Average Waiting Time = %.4f", _ROUND_ROBIN.avg_waiting_time);
  printf("\n* Average Turnaround Time = %.4f\n",
_ROUND_ROBIN.avg_turnaround_time);

printf("**********************************************************************
******\n\n");
}

void evaluation() {
        printf("***Evaluation of each scheduling algorithms***\n");
        printf("----------------------------------------------------------------
--------------\n");
        printf("1. FCFS Scheduling\n");
        printf("Average Waiting Time = %.4f, Average Turnaround Time = %.4f\n",
_FCFS.avg_waiting_time, _FCFS.avg_turnaround_time);
        printf("CPU Utilization = %.2f%%\n", (float)((_FCFS.finished_time -
_FCFS.idle_time) * 100) / _FCFS.finished_time);
        printf("----------------------------------------------------------------
--------------\n");
        printf("2. Non-Preemptive SJF Scheduling\n");
        printf("Average Waiting Time = %.4f, Average Turnaround Time = %.4f\n",
_NP_SJF.avg_waiting_time, _NP_SJF.avg_turnaround_time);
        printf("CPU Utilization = %.2f%%\n", (float)((_NP_SJF.finished_time -
_NP_SJF.idle_time) * 100) / _NP_SJF.finished_time);
        printf("----------------------------------------------------------------
--------------\n");
        printf("3. Preemptive SJF Scheduling\n");
        printf("Average Waiting Time = %.4f, Average Turnaround Time = %.4f\n",
_P_SJF.avg_waiting_time, _P_SJF.avg_turnaround_time);
        printf("CPU Utilization = %.2f%%\n", (float)((_P_SJF.finished_time -
_P_SJF.idle_time) * 100) / _P_SJF.finished_time);
        printf("----------------------------------------------------------------
--------------\n");
        printf("4. Non-Preemptive Priority Scheduling\n");
        printf("Average Waiting Time = %.4f, Average Turnaround Time = %.4f\n",
_NP_PRIORITY.avg_waiting_time, _NP_PRIORITY.avg_turnaround_time);
        printf("CPU Utilization = %.2f%%\n",
(float)((_NP_PRIORITY.finished_time - _NP_PRIORITY.idle_time) * 100) /
_NP_PRIORITY.finished_time);
        printf("----------------------------------------------------------------
--------------\n");
        printf("5. Preemptive Priority Scheduling\n");
        printf("Average Waiting Time = %.4f, Average Turnaround Time = %.4f\n",
_P_PRIORITY.avg_waiting_time, _P_PRIORITY.avg_turnaround_time);
        printf("CPU Utilization = %.2f%%\n", (float)((_P_PRIORITY.finished_time
- _P_PRIORITY.idle_time) * 100) / _P_PRIORITY.finished_time);
        printf("----------------------------------------------------------------
--------------\n");
        printf("6. Round-Robin Scheduling\n");
        printf("Average Waiting Time = %.4f, Average Turnaround Time = %.4f\n",
_ROUND_ROBIN.avg_waiting_time, _ROUND_ROBIN.avg_turnaround_time);
```

```
        printf("CPU Utilization = %.2f%%\n",
(float)((_ROUND_ROBIN.finished_time - _ROUND_ROBIN.idle_time) * 100) /
_ROUND_ROBIN.finished_time);
        printf("------------------------------------------------------------------
--------------\n");
        }

int main() {
    create_processes();
    Config();

    // First-Come First-Served
    FCFS();

    // Non-preemptive SJF
    NP_SJF();
    // Preemptive SJF
    P_SJF();

    // Non-preemptive Priority
    NP_PRIORITY();
    // Preemptive Priority
    P_PRIORITY();

    // Round-Robin
    ROUND_ROBIN();

    // Evaluation
    evaluation();

    return 0;
}
```