COSE341-01

Operating System

Prof. Heonchang Yu

Term Project Report

The CPU Scheduling Simulator

Inho Kim 2012130888

Korea University

English Language and Literature / College of Information and Communication

Table of Contents

1. CPU Scheduling의 정의

1.1 CPU Scheduling이란?

프로세서가 하나인 Single Processor 환경에서는 동시에 최대 한 개의 프로세스만 CPU로부터 리소스를 할당받아서 수행될 수 있다. 따라서 여러 프로세스가 동시에 CPU 작업을 수행하려고 한다면, 일련의 순서를 정해서 혼란이 없도록 해야 한다. 이렇게 순서를 정해주어 각각의 프로세스들이 원활히 수행이 될 수 있도록 하는 것을 CPU Scheduling이라고 한다.

1.2 CPU Scheduling의 평가 지표

CPU Scheduling은 임의적으로 진행되는 것이 아니라 특정 기준에 의해 효율적인 방법을 따라야 한다. Scheduling의 효율성을 평가하는 지표로는 다음과 같은 항목들이 있다.

① CPU Utilization

Scheduling된 결과에 따라 프로세스들이 수행되는 동안 CPU가 유휴 상태 (idle)에 돌입하지 않은 채로 프로세스를 수행하는

② Average Waiting Time

특정 프로세스가 CPU에게 리소스를 할당받아서 수행이 되는 동안 다른 프로세스들은 Scheduling의 결과에 따라 자신이 차례가 될 때까지 기다려야 한다. 프로세스의 기다리는 시간이 길어질수록 비효율적이다. Average Waiting Time은 각각의 프로세스들이 기다린 시간에 대한 평균이다. 따라서 Average Waiting Time이 낮을수록 더 효율적이라고 할 수 있다.

③ Average Turnaround Time

Turnaround Time이란 프로세스가 생성되어서 수행할 준비를 끝마친 순간부터 모든 수행이 완료된 시점까지 소요되는 시간을 의미한다. Average Turnaround Time은 각각의 프로세스들의 Turnaround Time에 대한 평균값으로, 이 값이 낮을수록 더 효율적이다.

④ Average Response Time

Response Time은 특정 프로세스가 수행할 의지를 표현하여 request를 발생시켰을 때, 처음으로

해당 프로세스가 CPU에서 수행이 될 때까지 걸리는 시간을 뜻한다. Average Response Time은 이에 대한 평균값으로, 이 값이 낮을수록 프로세스들의 요청에 대한 응답시간이 빠른 것이기 때문에 더욱 효율적이라고 할 수 있다.

⑤ Throughput

단위 시간 당 수행이 완료된 프로세스의 수를 의미한다.

1.3 CPU Scheduling 방법

어떤 프로세스에게 CPU에서의 수행에 대한 우선권을 줄 것인지를 결정하는 것이 CPU Scheduling의 핵심이다. 이는 다양한 알고리즘들을 통해 성취될 수 있는데, 각각의 알고리즘들은 앞서 언급한 평가 지표들에 대해서 다양한 양상을 보이게 된다. 최적의 알고리즘이란 물론 각각의 평가 지표들에 대해 가장 높은 효율성을 보이는 것이지만, 모든 평가 지표에 대해서 효율성을 만족하는 알고리즘은 사실상 구현하기가 힘들다. 따라서 상황에 따라 각각의 평가 지표들 중 일부에 비중을 두어서 해당 평가 지표에서 높은 효율을 보이는 알고리즘을 선택하여 사용하는 것이 일반적이다. 실제로는 여러 알고리즘들을 융합하여 때에 따라 적절히 사용함으로써 효율성을 극대화하는 방법을 많이 사용한다.

2. CPU Scheduling Algorithms

2.1 FCFS (First Come First Served)

(1) FCFS 알고리즘이란?

FCFS 스케줄링은 먼저 CPU에서의 수행을 요청한 프로세스에게 우선권을 부여하는 스케줄링 방법이다. 이 방식은 Preemptive 방식이기 때문에 일단 한 프로세스가 CPU에서 수행중이라면, 해당 프로세스가 완료되거나 Interrupt를 당하여 CPU가 idle 상태가 되기 전까지는 다른 프로세스가 CPU에서 수행이 될 수 없다.

(2) FCFS 알고리즘의 특징

- 장점

알고리즘의 구현이 매우 쉽다. 그리고 프로세스 사이에서 CPU에서의 수행에 대한 요청을 한 순서대로 수행이 되기때문에 특정 프로세스가 수행에 대한 우선권을 지속적으로 받지 못해서 나타나는 Starvation 문제가 발생하지 않는다.

- 단점

요청 순서를 제외한 다른 사항들은 우선권의 분배에서 고려되지 않기 때문에 Convoy Effect가 발생할 수 있다. Convoy Effect는 먼저 수행중인 프로세스의 수행시간이 길어서 수행시간이 상대적으로 짧은 다른 프로세스들이 수행을 시작하기 전까지 오랜 시간을 기다려야 하는 문제를 일컫는 말이다.

2.2 SJF (Shortest Job First)

(1) SJF 알고리즘이란?

SJF는 남아있는 CPU Burst Time이 작은 프로세스, 다시 말해서 CPU에서 작업을 수행하는데 소요되는 시간이 가장 적게 남은 프로세스에게 우선권을 주는 방법이다. 이 알고리즘은 Preemptive 방식 또는 Non-Preemptive 방식으로 구현될 수 있다. Non-Preemptive 방식일 경우 일단 한 프로세스가 CPU에서 수행중이라면 그 프로세스의 CPU Burst time이 끝나기 전에는 다른 프로세스들이 수행되지 못하게 된다. 반대로, Preemptive 방식이라면 현재 수행중인 프로세스보다 우선권을 가지는 프로세스가 현재 수행중인 프로세스를 일시정지 시키고 먼저 수행될 수 있다.

(2) SJF 알고리즘의 특징

- 장점

CPU Burst Time을 고려하기 때문에 Average Waiting Time과 Average Turnaround Time의 측면에서 효율성을 보인다. 처리 속도에 있어서는 가장 이상적인 알고리즘으로 알려져 있다.

- 단점

이 스케줄링은 프로세스의 CPU Burst Time을 비교함으로서 우선권을 할당하도록 구현되는데, 실제 상황에서는 프로세스의 CPU Burst Time을 예측하는 것이 사실상 불가능하다. 따라서 현실적이지 못한 알고리즘이라고 할 수 있다.

2.3 Priority

(1) Priority 알고리즘이란?

Priority 알고리즘은 각각의 프로세스에 우선순위를 부여하고 가장 높은 우선순위가 부여 된 프로세스에게 CPU 수행의 우선권을 부여하는 방식이다. SJF 알고리즘과 마찬가지로 Preemptive, 혹은 Non-Preemptive 방식으로 구현될 수 있다.

(2) Priority 알고리즘의 특징

- 장점

프로세스들이 수행되는 순서가 우선순위에 따라 결정되기 때문에 이를 활용해서 중요도가 높은 프로세스가 우선적으로 수행될 수 있도록 하는 것이 가능하다.

- 단점

프로세스가 수행에 대한 우선권을 지속적으로 받지 못해서 나타나는 Starvation 문제가 발생할 수 있게 된다. 어떤 프로세스가 상대적으로 낮은 우선순위를 가지고 있을 때, 계속해서 우선순위가 높은 프로세스들이 수행을 요청한다면, 결국 우선순위가 낮은 프로세스는 작업을 수행할 수 없게 된다.

그런데 이 Starvation 문제는 Aging 기법으로 처리할 수 있다. Aging 기법은 어떤 프로세스가 계속해서 수행되지 못하고 대기하게 될 때 시간이 지남에 따라 우선순위를 점차적으로 높여주는 방식이다.

2.4 Round Robin

(1) Round Robin 알고리즘이란?

Round Robin 방식은 각각의 프로세스들이 한 번에 CPU에서 수행될 시간 (Time Quantum)을 미리 정의하고, 이를 각각의 프로세스들에 할당해주는 방식이다. 기본적으로 프로세스가 수행되는 순서는 CPU 수행에 대한 요청을 한 순서, 즉 FCFS 알고리즘과 같은 방식을 따른다. 각각의 프로세스는 자신이 수행될 차례가 되었을 때 Time Quantum만큼의 시간 동안만 수행을 하고 다시 대기상태로 돌아가서 다른 프로세스에게 차례를 넘긴다. 만약 할당된 시간 안에 프로세스가 CPU 수

행이 모두 완료되었다면 그대로 프로세스를 Terminate시킨다.

이 Time Quantum이 무한히 크다면, 결국 FCFS와 동일한 결과를 얻게 되고, Time Quantum이 매우 적다면, 프로세스들의 수행 상태와 대기 상태 사이에서의 회전율이 증가하여 마치 여러 프로세스들이 동시에 수행되는 것과 같은 효과를 볼 수 있게 된다. 하지만, 이렇게 회전율이 높다는 것은 Context Switch가 그만큼 많이 발생하게 되기 때문에 Context Switching Time이 증가함에 의한 Overhead가 많이 발생한다.

(2) Round Robin 알고리즘의 특징

- 장점

각 프로세스들이 최대한으로 CPU를 쓸 수 있는 시간이 Time Quantum 만큼 이므로 Response Time이 다른 알고리즘들에 비해 상대적으로 짧다. 게다가 대기시간도 무한하지 않고, 다른 프로세스들이 Time Quantum만큼의 수행을 마쳤다면 반드시 순서가 돌아올 것임을 보장하기 때문에 Starvation 문제가 발생하지 않는다.

- 2) 단점

앞서 언급했듯이 Time Quantum에 의해서 여러 프로세스들이 수행 상태와 대기 상태를 오가기 때문에 그만큼 Context Switching Time이 증가하여 전체적인 효율성이 하락할 수 있다.

3. CPU Scheduling Simulator의 구현

3.1 시뮬레이터 구현을 위한 기본 설정 사항

- **프로세스의 생성**

본 시뮬레이터는 사용자로부터 프로세스의 개수를 입력 받아서 프로세스들을 생성하도록 하였다. 사용자는 전체 프로세스의 개수 및 I/O 작업을 수행할 프로세스의 개수를 인자로 전달하게 된다. 프로세스는 내부적으로 여러 attribute들을 가지게 되는데, 각각은 프로세스가 처음으로 생성될 때 다음과 같은 값들을 가지도록 설정된다.

Pid = 1 ~ N 범위의 unique한 값

Priority = 1 ~ N 범위의 random한 값

Arrival time = 0 ~ N + 9 범위의 random한 값

CPU burst time = 5 ~ 20 범위의 random한 값

I/O burst time = 0

(단, N = 전체 프로세스의 개수)

총 N개의 프로세스가 생성되고 난 뒤에는, I/O 작업을 수행할 프로세스를 선정한다. 사용자가 입력한 개수만큼 선정하며, 선정된 각각의 프로세스는 1 ~ 10 범위의 random한 값으로 I/O burst time이 초기화된다.

그 외에 Algorithm의 평가를 위해 사용되는 Waiting time, Turnaround time, Response time 등의 정보도 각 프로세스마다 포함하고 있다. 이들은 처음 프로세스가 생성될 때 대부분 다음과 같은 값으로 초기화된다.

Waiting time = 0

Turnaround time = 0

Response time = -1

Waiting time과 Turnaround time은 시뮬레이션이 진행됨에 따라 그 값이 합산되어 점점 증가하게 된다. Response time같은 경우 -1로 초기화된다. 그 의미는 아직 running state에 돌입하지 못했다는 것이다. 시뮬레이션이 진행되면서 프로세스가 처음으로 running state가 될 경우 그 시점을 이 response time에 기록해둔다.

♦ **I/O 작업의 발생 및 수행 과정에 대한 일반화**

앞서 살펴본 프로세스의 생성과정에서 설정된 I/O burst time은 어떤 프로세스가 수행되는 동안 I/O 작업을 수행할 필요가 생겼을 때 실제 그 작업을 수행하면서 waiting queue에 상주하는 시간을 의미한다. 즉, 어떤 프로세스의 I/O burst time이 0인 경우는 I/O 작업을 수행할 필요가 없는 CPU-bound process임을 의미하는 것이다.

본 시뮬레이터는 I/O 작업을 수행할 필요가 있다고 선정된 프로세스들에 대하여 I/O 작업이 특정 시점에 발생하여 수행되도록 구성하였다. CPU Scheduler에 의해 running 상태에 돌입하게 된 프로세스는 우선 CPU operation을 1의 시간단위만큼 수행한 뒤에 바로 I/O 작업이 발생했다고 가정한 뒤 waiting queue로 이동한다. 해당 프로세스는 프로세스 내부적으로 정의된 I/O burst time

만큼의 시간 동안 waiting queue에서 상주한 뒤에, I/O 작업이 끝났음을 알리며 Ready queue로 되돌아오게 된다.

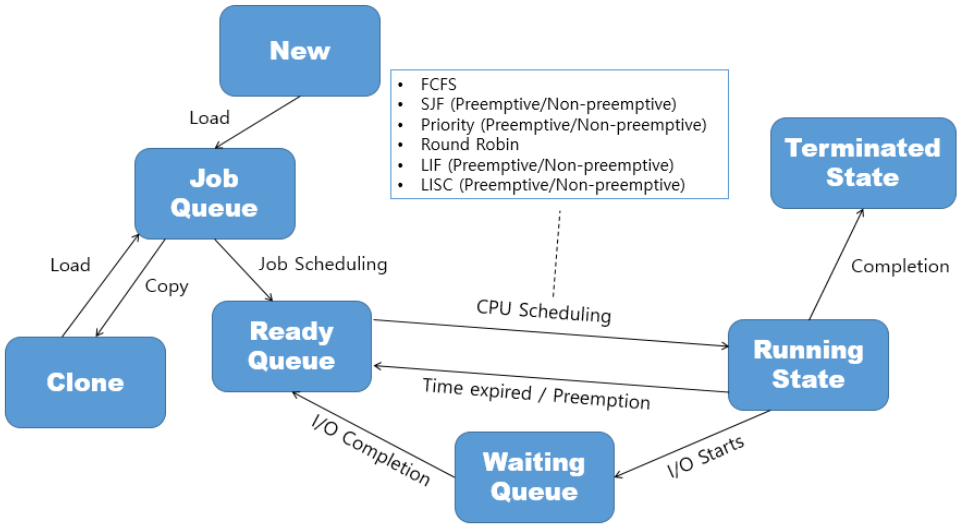
♦ **Round Robin Algorithm에서 사용되는 time quantum 값에 대한 정의**

Time quantum 값은 시뮬레이터 내부에 상수로서 3으로 정의하였다. 필요에 따라 사용자가 이 상수의 값을 수정하여 time quantum을 변경할 수 있도록 하였다. Round Robin 방식에서 프로세스가 running 상태로 time quantum만큼의 시간을 보내게 되면 timer interrupt가 발생하여 ready queue로 이동하게 된다.

♦ **Priority 값에 대한 정의**

Priority 값이 낮을수록 우선순위가 높은 것이라고 정의하였다. 그리고 Priority 값 자체를 설정하는 것은 Scheduler의 몫이 아니라 외부에서 주어진 값을 사용하는 것이라고 가정하였기 때문에, 특정 logic에 의해 priority를 배분하는 것이 아니라 프로세스 생성과정에서 random한 값을 부여하도록 하였다.

3.2 시스템 구성도



① **New**

시뮬레이터가 시작되어 새로운 프로세스들이 생성되는 단계이다. 또한, 시스템에 필요한 Job Queue, Ready Queue, Waiting Queue, Running State, Terminated State 등을 초기화한다.

② **Job Queue**

생성된 프로세스들이 담기는 곳이다. 이는 처리해야할 작업들을 목록 형태로 관리하는 것과새로 유사하다. 새로 생성된 프로세스들은 Job Queue에 load된 다음 pid를 기준으로 오름차순으로 정렬된다. 이는 사용자가 임의적으로 pid를 generate했을 때 시스템 내부적으로 일관성을 유지하기 위해 디버그 차원에서 구현한 것이다.

프로세스들이 pid를 기준으로 정렬된 상태가 되었다면, 여러 Scheduling Algorithm로부터 작업이 수행되기 전 Job Queue의 상태를 reference하기 위해서 프로세스가 생성 뒤에 load된 초기 상태 그대로를 복사해서 Clone에 저장한다.

③ Clone

Job Queue의 초기 정보를 임시적으로 담는 곳으로, 시뮬레이터가 다양한 Algorithm으로 여러 번 수행될 때, 동일한 프로세스 정보를 활용하기 위해 만들어진 것이다. 시뮬레이터가 매번 실행될 때마다 Clone에 저장된 프로세스 정보들을 Job Queue로 불러온 뒤 시뮬레이션을 시작한다.

④ Ready Queue

Arrival time이 되어서 수행할 준비가 된 프로세스들은 Job scheduler에 의해서 Ready Queue로 옮겨진다. Job Scheduler는 매시간단위마다 프로세스들의 Arrival time을 체크하여 Ready Queue로 이동시키기 때문에 Ready Queue 내부의 프로세스들은 자동적으로 Arrival time을 기준으로 정렬되고, 동시에 도착한 프로세스들은 pid를 기준으로 정렬이 된 상태가 된다.

⑤ Running State

시뮬레이션에 사용된 Scheduling Algorithm에 따라 Ready Queue에 있는 프로세스 중 하나의 프로세스가 Running state로 dispatch된다. 본 시뮬레이터는 싱글 프로세서 기반 환경을 가정하였기 때문에 Running State는 다른 Queue들과 달리 size가 1로 설정되었다. 즉, 특정 시간단위에서 수행될 수 있는 프로세스는 최대 한 개이며 동시에 여러 프로세스가 수행될 수 없다. 만약 Preemption이 발생되었다면 Running state에 있던 프로세스는 다시 Ready Queue로 돌아가게 된다.

⑥ Waiting Queue

I/O 작업을 수행할 필요가 있다고 선정된 프로세스의 경우 Running state에서 1만큼의 시간을 보내고 나서 Waiting Queue로 이동하게 된다. 이 Waiting Queue에서 프로세스 내부적으로 정의된 I/O burst time만큼의 시간을 보낸 뒤 다시 Ready Queue로 돌아간다.

⑦ Terminated State

프로세스 내부적으로 정의된 CPU burst time만큼의 시간 동안 Running state에 있었던 프로세스들은 Termination되어 Terminated State로 이동한다. Queue형태로 구현하여서 이곳에 저장된 프로세스의 순서가 곧 프로세스가 종료된 순서를 의미하도록 구성하였다. 시뮬레이션이 끝난 시점에서 이 Terminated State에 저장된 프로세스들의 정보를 통해 evaluation을 수행한다.

3.3 시뮬레이터 모듈

전체 흐름은 시스템 구성도에서 서술한 바와 같이 진행된다. 실제로 구현되어 수행되는 작업은 다음의 네 단계로 요약할 수 있다.

(1) 시스템 자원의 초기화

(2) 프로세스의 생성

(3) 알고리즘의 선택과 시뮬레이션의 시작

(4) 비교분석

(1) 시스템 자원의 초기화

시뮬레이션에 사용될 자원들을 초기화한다. 이 과정에서 Job Queue, Ready Queue, Termination Queue, Waiting Queue, Evaluation Queue의 값을 모두 NULL로 초기화하게 된다. Evaluation Queue 같은 경우는 CPU Scheduling Algorithm 사이에 비교 및 분석을 실행하기 위해 각 알고리즘에 대한 성능 정보를 담아두는 Queue이다.

(2) 프로세스의 생성

3.1에서 설명한대로 사용자가 요구한 개수대로 임의적인 프로세스들을 생성한다. 구조체를 사용하여 프로세스를 정의했기 때문에 프로세스가 생성된다는 것은 프로세스라는 구조체에 메모리를 할당해준 뒤 내부에 정의된 attribute들의 값을 설정해주는 것이다. 일련의 메모리 할당 과정을 진행하면서 생성된 프로세스 구조체들을 Job Queue에 삽입한다.

그 후에는 Job Queue에 삽입된 프로세스들을 pid를 기준으로 정렬한다. 그리고 여러 알고리즘에 대한 시뮬레이션을 하기 위해 현재 상태의 Job Queue를 복사하여 Clone을 만든다.

(3) 알고리즘의 선택과 시뮬레이션의 시작

얼마만큼의 시간단위 동안 시뮬레이션을 진행할 것인지를 amount라는 인자로 받는다. 생성된 모든 프로세스가 다 완료되기까지의 충분한 시간이 주어지지 않을 경우, 완료되지 않은 프로세스는 비교 분석 대상에서 제외되기 때문에 충분한 시간으로 설정할 필요가 있다. 엄청 큰 시간을 인자로 제공하더라도 모든 프로세스가 완료되었다면, 완료된 시점에서 시뮬레이션을 종료하고 평가를 진행하기 때문에 되도록이면 큰 값을 설정하는 것이 좋다.

시뮬레이션 진행 시간과 함께 어떤 알고리즘을 수행할 것인지도 인자로 받는다. 크게는 총 6개

(FCFS, SJF, Priority, Round Robin, LIF, LISC)의 알고리즘이 있으며 일부는 Preemption에 대한 옵션도 설정해주어야 한다.

옵션을 포함하여 인자가 충분히 주어졌을 경우 시뮬레이션을 시작한다. 먼저 프로세스 생성시에 미리 Backup해두었던 Clone의 프로세스들을 Job Queue로 load해온다. 본격적으로 프로세스들을 수행시키기 전에, 우선 알고리즘을 평가하기 위한 여러 준비 사항들을 세팅한다. CPU Utilization을 측정하기 위해 가장 먼저 도착한 프로세스의 Arrival Time을 Computation_start라는 변수에 저장해두고, CPU가 idle한 시간을 저장해두는 Computation_idle이라는 변수를 0으로 초기화한다. 이 두 변수와 모든 프로세스가 완료되었을 때의 시점을 함께 계산하여 CPU Utilization을 측정한다.

준비가 완료가 되었으면 사용자가 인자로 전달한 시간의 양만큼 루프를 돌면서 실제 시뮬레이션을 진행한다. 이 과정을 Pseudo Code로 표현하면 다음과 같다. 루프를 한 번 도는 것은 1만큼의 시간이 흘렀음을 나타낸다.

```
for (i = 0; i < amount_of_time; i ++) {
    while { -> Job Queue에 있는 프로세스들을 하나씩 차례대로 참조한다.
        if p == arrival time이 시간 i인 프로세스
            move(p, Job Queue -> Ready Queue);
    }                                                          Job Scheduling

    nextProcess = 선택한 알고리즘(option);
    -> Running Process 혹은 Ready Queue의 프로세스 중 이번 turn (시간 i)에 수행되어야 할 프로세스를 선정

    if nextProcess != Running Process (선정된 프로세스가 원래 수행중이던 프로세스와 다를 경우)
        RunningStateTime = 0 (동일한 프로세스가 Running State였던 시간)
        if nextProcess.responseTime == -1                      CPU Scheduling
            nextProcess.responseTime = i;
        Running Process = nextProcess;

    elapseTime(Ready Queue);
    -> Ready Queue에 있는 프로세스들의 waitingTime ++, turnaroundTime ++

    elapseTime(Waiting Queue);
    -> Waiting Queue에 있는 프로세스들의 turnaroundTime ++, I/O burstTime --

    check I/O Completion(Wating Queue) {                       I/O Processing
        if p.IOburstTime == 0
            print ("p -> I/O complete");
            move(p, Waiting Queue -> Ready Queue);
    }

    if Running Process != null
        run(Running Process); -> RunningStateTime ++, CPU burstTime --
        if Running Process.CPU burstTime <= 0                  Running
            move(Running Process -> Terminated);
            print("Terminated");
        else
            if Running Process.I/O burstTime > 0               I/O Request
            move(Running Process -> Waiting Queue);
            print("I/O Request");
    else
        print ("idle");
}
```

매 루프마다 우선적으로 Job Queue를 탐색하여 특정 시점 i에 도착한 프로세스가 있는지를 점검한다. 만약 발견 시 해당 프로세스를 Ready Queue로 이동시켜준다.

그리고 나서 현재 Running State에 있는 프로세스와 Ready Queue에 존재하는 프로세스들을 고려하여 시점 i에 수행될 프로세스를 선정해준다. 이때 선택된 알고리즘에 따라 결과가 다르며, 만

약 원래 Running State에 있었던 프로세스가 시점 i에서도 계속 Running State를 유지해야한다면, Scheduling의 결과로 Running State가 그대로 반환된다. 만약 아무 프로세스도 Scheduling되지 않았다면, 수행할 프로세스가 없다고 판단하고 CPU가 idle상태라는 것을 표현한다. CPU가 idle 상태인 것을 감지하면 Computation_idle 변수를 1 증가시켜서 나중에 CPU Utilization을 측정할 때 반영하도록 한다.

따라서 새로 Scheduling된 프로세스가 기존의 Running State였던 프로세스와 다른지 유무를 확인하고, 다를 경우에 RunningStateTime이라고 표기되어 있는 변수의 값을 0으로 초기화한다. 이 변수는 동일한 프로세스가 Running State에 있던 기간을 뜻한다. 이 변수는 Round Robin 알고리즘에서 수행중인 프로세스가 time expired 되었는지의 유무를 판단할 때 사용된다.

이렇게 성공적으로 시점 i에 수행될 프로세스를 scheduling했다면, Ready Queue와 Waiting Queue, 그리고 Running State에 존재하는 프로세스들에게 시간이 흘렀음을 notify해준다. 이는 Waiting Time, Turnaround Time 등을 1씩 증가시켜주거나 CPU Burst Time, I/O Burst Time 등을 1 감소시키는 과정으로 표현된다.

시간이 흘렀음을 프로세스들에게 알린 뒤에는 즉각적으로 해당 프로세스들의 상태를 점검해서 할당된 CPU Burst Time을 모두 소모했다면 Terminated 상태로 만들어주고, 할당된 I/O Burst Time을 모두 소모했다면 Waiting Queue에 있던 해당 프로세스를 Ready Queue로 옮겨주는 등의 작업을 수행한다. 이를 통해 프로세스들이 Progressive하게 여러 State들 사이를 이동할 수 있도록 해주는 것이다.

이러한 루프를 사용자가 제공한 시간 동안 반복하거나, 생성된 프로세스들이 모두 다 완료가 될 때까지 반복한다. 어떤 경우든 간에, 루프가 완료된 시점을 Computation_end라는 변수에 저장하고 CPU Utilization을 측정하는 데 반영한다.

(4) 비교분석

(3)의 과정이 모두 완료되면, 선택된 알고리즘 자체적으로 평가를 진행한다. Termination Queue에 저장되어 있는 프로세스들을 기반으로 Average Waiting Time, Average Turnaround Time, Average Response Time, 그리고 CPU Utilization을 측정한다. 이 정보들을 evaluation이라는 구조체에 저장하고 Evaluation Queue에 삽입하여 다른 알고리즘들과의 비교분석이 가능하도록 한다.

3.4 CPU Scheduling Algorithm의 구현

3.3에서 시점 i에 수행될 프로세스를 Scheduling할 때 사용자가 시뮬레이션을 원하는 알고리즘을 선택할 수 있다. 앞서 언급한 바와 같이 크게 FCFS, SJF, Priority, Round Robin, LIF, LISC로 총 6종류의 알고리즘이 시뮬레이터에 구현되었다. 그런데, 이 알고리즘 중 SJF, Priority, LIF, 그리고

LISC 알고리즘은 Preemptive 옵션을 부여할 수 있기 때문에 총 10가지의 알고리즘을 시뮬레이션에 사용할 수 있다. 각각의 알고리즘들은 다음과 같은 메커니즘으로 구현되었다.

(1) FCFS

알고리즘 상 Preemption은 적용되지 않는 것이 default이며, Job Scheduler가 Job Queue에 상주하는 프로세스들을 Ready Queue로 이동시킬 때 프로세스의 Arrival Time을 기준으로 순차적으로 이동시키기 때문에 프로세스들이 Ready Queue에서 Arrival Time을 기준으로 오름차순으로 정렬되어 있다는 것은 이미 언급하였다. 게다가, 프로세스가 처음 생성되어 Job Queue로 이동한 뒤에 pid를 기준으로 정렬을 한다는 것도 언급하였다. 따라서 이 두 가지 특징에 의해 Ready Queue는 Arrival Time을 기준으로 우선 정렬되고, Arrival Time이 같은 프로세스들 사이에서는 pid를 기준으로 정렬이 되어있다. 따라서 FCFS 알고리즘에서는 별도의 탐색이나 비교 작업을 거치지 않고 간단하게 Ready Queue에서 첫 번째에 위치한 프로세스를 Scheduling해주면 된다.

(2) Preemptive & Non-preemptive SJF (Shortest Job First)

매순간 Ready Queue에서 CPU burst time이 가장 적게 남은 프로세스를 선정해주면 된다. 만약 Non-preemptive mode라면, Running State에 있는 프로세스가 있을 경우 그대로 Running State에 있던 프로세스를 반환해준다.

반대로, Preemptive mode라면 Running State에 있는 프로세스가 있을 경우 우선 새로 선정된 프로세스와 Running 프로세스의 남은 CPU burst time을 비교한 뒤, 만약 새로 선정된 프로세스의 CPU burst time이 더 적게 남았다면, Preemption을 발생시켜 Running State가 되도록 한다. 기존의 Running 프로세스는 Ready Queue로 보내주며, Preempt가 되었음을 출력하여 알린다. 이 과정을 Pseudo Code로 나타내면 다음과 같다.

```
P = PickUp_Min_RemainingCPUburstTime(ReadyQueue);
If (RunningProcess == NULL)    Run(P);
Else {
        If(Preemptive) {
            TEMP = Choose_Smaller_CPUburstTime(RunningProcess, P);
            Print("Preemption is detected!");
            Run(TEMP);
        } Else Run(RunningProcess); //do nothing
}
```

(3) Preemptive & Non-preemptive Priority

SJF 알고리즘과 동일한 구조를 갖는다. 다만 남은 CPU burst time을 기준으로 비교하는 것이 아니라, 프로세스 내부적으로 정의된 Priority 값을 통해 비교한다.

(4) Round Robin

기본구조는 FCFS 알고리즘과 동일하게 구성한다. 매시점마다 FCFS 방식으로 프로세스 한 개를 선정한 뒤 현재 Running State인 프로세스가 존재하는지를 판단하는 것까지는 FCFS 알고리즘과 동일하다. 그런데, 만약 이미 Running State인 프로세스가 존재하는 경우 앞서 언급했던 Running State Time이라는 변수를 체크해서 주어진 time quantum과 비교를 한다. 만약 현재 Running State인 프로세스가 time quantum만큼의 시간 동안 지속되었다면 해당 프로세스를 Ready Queue로 돌려보내고 새로 선정한 프로세스를 Running State로 만들어준다.

이때, time quantum을 모두다 소모하는 그 시점에 프로세스의 CPU burst time이 동시에 모두 소모되어 더 이상 수행할 필요가 없는 프로세스가 Ready Queue로 돌아가지 않느냐는 우려가 나올 수도 있다. 그런데, 3.3의 Pseudo Code에서 명시하였듯이, Running State인 프로세스는 1만큼의 시간 동안 수행이 된 뒤에 루프가 끝나기 전 완료 여부의 상태를 체크받게 되고, 만약 CPU burst time이 모두 소진되었다면 즉각적으로 Terminated 상태로 이동하게 구성되었다. 따라서 이 Round Robin 알고리즘에서 불필요한 State shift가 발생하지 않는다는 것을 보장한다.

(5) Preemptive & Non-preemptive LIF (Longest I/O First)

I/O-bound Process들은 일반적으로 비교적 짧은 다수의 CPU burst를 발생시키고 Waiting Queue에 있는 시간이 많다는 개념에 착안하여 고안한 알고리즘이다. I/O burst time이 길수록 Waiting Queue에 상주하는 시간이 길어져서 그만큼 CPU가 idle 상태가 될 가능성이 높기 때문에 I/O burst time이 긴 프로세스들을 우선적으로 Scheduling해주는 것이다.

기본구조는 SJF 알고리즘과 동일하게 구현하였으며, CPU burst time으로 비교하는 것이 아니라 I/O burst time으로 비교하였다는 것이 유일한 차이점이다.

(6) Preemptive & Non-Preemptive LISC (Longest I/O Shortest CPU First)

(5)의 LIF 알고리즘의 장점은 분명히 있지만, 남아있는 I/O burst time이 같은 프로세스들 사이에서는 FCFS 방식으로 Scheduling이 되기 때문에 FCFS 알고리즘의 단점을 그대로 떠안을 수 밖에 없다. 따라서 LIF 알고리즘의 장점은 살리면서, FCFS 알고리즘의 단점은 최소화하기 위해 우선적으로 LIF 알고리즘의 방식으로 Scheduling을 하되, I/O burst time이 같은 프로세스들 사이에서는 남은 CPU burst time이 가장 적은 프로세스를 선정하도록 구성하였다. 즉, LIF 알고리즘을 수행하는데, 같은 I/O burst time을 갖는 프로세스들은 SJF 알고리즘으로 Scheduling을 하는 것이다. LIF 알

고리즘과 SJF 알고리즘의 융합 형태인 알고리즘이다.

4. Scheduling 알고리즘 성능 평가 및 비교분석

4.1 전제 조건

3.4에서 살펴본 알고리즘들의 성능을 평가 및 비교분석하기 위해서 몇 가지 가정을 전제로 시뮬레이션이 진행되었다. 우선 시뮬레이션에서 알고리즘의 성능을 평가하는 지표로 Average Waiting Time, Average Turnaround Time, Average Response Time, CPU Utilization 이 네 가지를 사용하였다. 각각의 지표들은 다음과 같이 정의되었다.

(1) Average Waiting Time

Waiting Time의 경우는 프로세스가 Waiting Queue에 상주하는 시간은 고려하지 않고 단지 Ready Queue에 상주하는 시간만 고려하였다. 그 이유는 알고리즘간에 비교분석을 할 때, 알고리즘 자체의 특성보다 I/O 작업의 수행시간에 의해 성능이 크게 좌우되는 것을 방지하기 위해서이다.

(2) Average Turnaround Time

Job Scheduler에 의해 프로세스가 처음 Ready Queue에 도착한 순간부터 계산하기 시작한다. 3.3에서 살펴본 시뮬레이터의 모듈에서 루프를 한 번 돌 때마다 1의 시간만큼 Turnaround Time이 증가된다. 해당 프로세스의 모든 CPU burst time이 소진되어 Terminated 상태가 된 순간 Turnaround Time 증가를 멈춘다. 따라서 이는 결과적으로 프로세스가 완료된 시간에서 프로세스가 처음 도착한 시간을 뺀 것과 같게 된다.

(3) Average Response Time

3.3에 언급되었듯이 Job Scheduler에 의해 프로세서가 처음으로 Ready Queue에 도착한 시간을 기록하게 된다.

(4) CPU Utilization

그리고 CPU Utilization의 경우는 시뮬레이션이 시작될 때의 시간부터 고려하는 것이 아니라, 임의의 프로세스가 처음으로 Ready Queue에 도착한 시간부터 고려하였다. 예를 들어 첫 번째로 도

착하는 프로세스의 Arrival Time이 5라면, 시점 5가 될 때까지 CPU는 idle 상태가 된다. 이러한 경우에서 CPU가 idle한 시간은 결과에 반영하지 않는다. 즉, 하나 이상의 프로세스가 도착한 순간부터 CPU의 idle 여부를 체크하기 시작하는 것이다. 이 CPU가 idle 상태인 시간을 측정한 Computation_idle과 함께 처음으로 프로세스가 도착한 시점인 Computation_start, 그리고 모든 프로세스가 완료된 시점인 Computation_end을 활용해 CPU Utilization을 측정한다.

4.2 알고리즘 성능 평가

(1) 프로세스 정보

총 8개의 프로세스를 생성하였으며, 그 중 2개의 프로세스는 I/O 작업을 수행하도록 하였다. 시뮬레이션 과정에서 임의적으로 생성된 프로세스의 정보는 다음과 같다.

총 프로세스 수: 8

pid	priority	arrival_time	CPU burst	IO burst
1	7	13	10	6
2	2	15	5	0
3	6	9	20	2
4	1	0	15	0
5	3	10	13	0
6	2	12	16	0
7	3	12	11	0
8	4	4	7	0

(2) 시뮬레이션 결과

이 프로세스 정보를 바탕으로 10가지의 Scheduling 알고리즘을 적용하여 성능을 측정하였다. Multiprogramming 환경에서 프로세스들이 어떤식으로 수행되는지 정확히 알아보기위해 Gantt Chart를 출력하였다. 맨 좌측에 위치한 숫자는 특정 시점을 의미하며, 해당하는 행마다 프로세스의 상태를 '(pid: x) -> State'와 같은 상태로 나타내었다. 먼저 각각의 알고리즘에 대한 결과를 살펴보고, 종합하여 비교분석을 하도록 하겠다.

① FCFS

<FCFS ALGORITHM>	36: (PID: 6) -> RUNNING	71: (PID: 3) -> RUNNING
0: (PID: 4) -> RUNNING	37: (PID: 6) -> RUNNING	72: (PID: 3) -> RUNNING
1: (PID: 4) -> RUNNING	38: (PID: 6) -> RUNNING	73: (PID: 3) -> RUNNING
2: (PID: 4) -> RUNNING	39: (PID: 6) -> RUNNING	74: (PID: 3) -> RUNNING
3: (PID: 4) -> RUNNING	40: (PID: 6) -> RUNNING	75: (PID: 3) -> RUNNING
4: (PID: 4) -> RUNNING	41: (PID: 6) -> RUNNING	76: (PID: 3) -> RUNNING
5: (PID: 4) -> RUNNING	42: (PID: 6) -> RUNNING	77: (PID: 3) -> RUNNING
6: (PID: 4) -> RUNNING	43: (PID: 6) -> RUNNING	78: (PID: 3) -> RUNNING
7: (PID: 4) -> RUNNING	44: (PID: 6) -> RUNNING	79: (PID: 3) -> RUNNING
8: (PID: 4) -> RUNNING	45: (PID: 6) -> RUNNING	80: (PID: 3) -> RUNNING
9: (PID: 4) -> RUNNING	46: (PID: 6) -> RUNNING	81: (PID: 3) -> RUNNING
10: (PID: 4) -> RUNNING	47: (PID: 6) -> RUNNING	82: (PID: 3) -> RUNNING
11: (PID: 4) -> RUNNING	48: (PID: 6) -> RUNNING	83: (PID: 3) -> RUNNING
12: (PID: 4) -> RUNNING	49: (PID: 6) -> RUNNING	84: (PID: 3) -> RUNNING
13: (PID: 4) -> RUNNING	50: (PID: 6) -> RUNNING	85: (PID: 3) -> RUNNING
14: (PID: 4) -> RUNNING -> TERMINATED	51: (PID: 6) -> RUNNING -> TERMINATED	86: (PID: 3) -> RUNNING
15: (PID: 8) -> RUNNING	52: (PID: 7) -> RUNNING	87: (PID: 3) -> RUNNING -> TERMINATED
16: (PID: 8) -> RUNNING	53: (PID: 7) -> RUNNING	88: (PID: 1) -> RUNNING
17: (PID: 8) -> RUNNING	54: (PID: 7) -> RUNNING	89: (PID: 1) -> RUNNING

```
18: (PID: 8) -> RUNNING
19: (PID: 8) -> RUNNING
20: (PID: 8) -> RUNNING
21: (PID: 8) -> RUNNING -> TERMINATED
22: (PID: 3) -> RUNNING -> IO REQUEST
23: (PID: 5) -> RUNNING
24: (PID: 3) -> IO COMPLETE, (PID: 5) -> RUNNING
25: (PID: 5) -> RUNNING
26: (PID: 5) -> RUNNING
27: (PID: 5) -> RUNNING
28: (PID: 5) -> RUNNING
29: (PID: 5) -> RUNNING
30: (PID: 5) -> RUNNING
31: (PID: 5) -> RUNNING
32: (PID: 5) -> RUNNING
33: (PID: 5) -> RUNNING
34: (PID: 5) -> RUNNING
35: (PID: 5) -> RUNNING -> TERMINATED
55: (PID: 7) -> RUNNING
56: (PID: 7) -> RUNNING
57: (PID: 7) -> RUNNING
58: (PID: 7) -> RUNNING
59: (PID: 7) -> RUNNING
60: (PID: 7) -> RUNNING
61: (PID: 7) -> RUNNING
62: (PID: 7) -> RUNNING -> TERMINATED
63: (PID: 1) -> RUNNING -> IO REQUEST
64: (PID: 2) -> RUNNING
65: (PID: 2) -> RUNNING
66: (PID: 2) -> RUNNING
67: (PID: 2) -> RUNNING
68: (PID: 2) -> RUNNING -> TERMINATED
69: (PID: 1) -> IO COMPLETE, (PID: 3) -> RUNNING
70: (PID: 3) -> RUNNING
90: (PID: 1) -> RUNNING
91: (PID: 1) -> RUNNING
92: (PID: 1) -> RUNNING
93: (PID: 1) -> RUNNING
94: (PID: 1) -> RUNNING
95: (PID: 1) -> RUNNING
96: (PID: 1) -> RUNNING -> TERMINATED
============================================================
(PID: 4)
WAITING TIME = 0, TURNAROUND TIME = 15, RESPONSE TIME = 0
============================================================
(PID: 8)
WAITING TIME = 11, TURNAROUND TIME = 18, RESPONSE TIME = 11
============================================================
(PID: 5)
WAITING TIME = 13, TURNAROUND TIME = 26, RESPONSE TIME = 13
============================================================
(PID: 6)
WAITING TIME = 24, TURNAROUND TIME = 40, RESPONSE TIME = 24
============================================================
(PID: 7)
WAITING TIME = 40, TURNAROUND TIME = 51, RESPONSE TIME = 40
============================================================
(PID: 2)
WAITING TIME = 49, TURNAROUND TIME = 54, RESPONSE TIME = 49
============================================================
(PID: 3)
WAITING TIME = 57, TURNAROUND TIME = 79, RESPONSE TIME = 13
============================================================
(PID: 1)
WAITING TIME = 68, TURNAROUND TIME = 84, RESPONSE TIME = 50
============================================================
START TIME: 0 / END TIME: 96 / CPU UTILIZATION : 100.00%
AVERAGE WAITING TIME: 32
AVERAGE TURNAROUND TIME: 45
AVERAGE RESPONSE TIME: 25
COMPLETED: 8
============================================================
```

프로세스가 도착하는 순서대로 그대로 수행되기 때문에, 이 경우에는 중간에 CPU가 idle 상태가 되는 일 없이 CPU Utilization이 100%로 나타났다. 만약 Ready Queue에 존재하는 프로세스의 수가 하나인 경우에 I/O작업이 수행되었다면 CPU Utilization이 낮아졌을 것이라고 예상할 수 있다.

② Non-Preemptive SJF

```
<NON-PREEMPTIVE SJF ALGORITHM>
0: (PID: 4) -> RUNNING
1: (PID: 4) -> RUNNING
2: (PID: 4) -> RUNNING
3: (PID: 4) -> RUNNING
4: (PID: 4) -> RUNNING
5: (PID: 4) -> RUNNING
6: (PID: 4) -> RUNNING
7: (PID: 4) -> RUNNING
8: (PID: 4) -> RUNNING
9: (PID: 4) -> RUNNING
10: (PID: 4) -> RUNNING
11: (PID: 4) -> RUNNING
12: (PID: 4) -> RUNNING
13: (PID: 4) -> RUNNING
14: (PID: 4) -> RUNNING -> TERMINATED
15: (PID: 2) -> RUNNING
16: (PID: 2) -> RUNNING
17: (PID: 2) -> RUNNING
18: (PID: 2) -> RUNNING
19: (PID: 2) -> RUNNING -> TERMINATED
20: (PID: 8) -> RUNNING
21: (PID: 8) -> RUNNING
22: (PID: 8) -> RUNNING
23: (PID: 8) -> RUNNING
24: (PID: 8) -> RUNNING
25: (PID: 8) -> RUNNING
26: (PID: 8) -> RUNNING -> TERMINATED
27: (PID: 1) -> RUNNING -> IO REQUEST
28: (PID: 7) -> RUNNING
29: (PID: 7) -> RUNNING
30: (PID: 7) -> RUNNING
31: (PID: 7) -> RUNNING
32: (PID: 7) -> RUNNING
36: (PID: 7) -> RUNNING
37: (PID: 7) -> RUNNING
38: (PID: 7) -> RUNNING -> TERMINATED
39: (PID: 1) -> RUNNING
40: (PID: 1) -> RUNNING
41: (PID: 1) -> RUNNING
42: (PID: 1) -> RUNNING
43: (PID: 1) -> RUNNING
44: (PID: 1) -> RUNNING
45: (PID: 1) -> RUNNING
46: (PID: 1) -> RUNNING
47: (PID: 1) -> RUNNING -> TERMINATED
48: (PID: 5) -> RUNNING
49: (PID: 5) -> RUNNING
50: (PID: 5) -> RUNNING
51: (PID: 5) -> RUNNING
52: (PID: 5) -> RUNNING
53: (PID: 5) -> RUNNING
54: (PID: 5) -> RUNNING
55: (PID: 5) -> RUNNING
56: (PID: 5) -> RUNNING
57: (PID: 5) -> RUNNING
58: (PID: 5) -> RUNNING
59: (PID: 5) -> RUNNING
60: (PID: 5) -> RUNNING -> TERMINATED
61: (PID: 6) -> RUNNING
62: (PID: 6) -> RUNNING
63: (PID: 6) -> RUNNING
64: (PID: 6) -> RUNNING
65: (PID: 6) -> RUNNING
66: (PID: 6) -> RUNNING
67: (PID: 6) -> RUNNING
68: (PID: 6) -> RUNNING
69: (PID: 6) -> RUNNING
71: (PID: 6) -> RUNNING
72: (PID: 6) -> RUNNING
73: (PID: 6) -> RUNNING
74: (PID: 6) -> RUNNING
75: (PID: 6) -> RUNNING
76: (PID: 6) -> RUNNING -> TERMINATED
77: (PID: 3) -> RUNNING -> IO REQUEST
78: IDLE
79: (PID: 3) -> IO COMPLETE, IDLE
80: (PID: 3) -> RUNNING
81: (PID: 3) -> RUNNING
82: (PID: 3) -> RUNNING
83: (PID: 3) -> RUNNING
84: (PID: 3) -> RUNNING
85: (PID: 3) -> RUNNING
86: (PID: 3) -> RUNNING
87: (PID: 3) -> RUNNING
88: (PID: 3) -> RUNNING
89: (PID: 3) -> RUNNING
90: (PID: 3) -> RUNNING
91: (PID: 3) -> RUNNING
92: (PID: 3) -> RUNNING
93: (PID: 3) -> RUNNING
94: (PID: 3) -> RUNNING
95: (PID: 3) -> RUNNING
96: (PID: 3) -> RUNNING
97: (PID: 3) -> RUNNING
98: (PID: 3) -> RUNNING -> TERMINATED
```

```
33: (PID: 1) -> IO COMPLETE, (PID: 7) -> RUNNING
34: (PID: 7) -> RUNNING
35: (PID: 7) -> RUNNING
70: (PID: 6) -> RUNNING
============================================================
(PID: 4)
WAITING TIME = 0, TURNAROUND TIME = 15, RESPONSE TIME = 0
============================================================
(PID: 2)
WAITING TIME = 0, TURNAROUND TIME = 5, RESPONSE TIME = 0
============================================================
(PID: 8)
WAITING TIME = 16, TURNAROUND TIME = 23, RESPONSE TIME = 16
============================================================
(PID: 7)
WAITING TIME = 16, TURNAROUND TIME = 27, RESPONSE TIME = 16
============================================================
(PID: 1)
WAITING TIME = 19, TURNAROUND TIME = 35, RESPONSE TIME = 14
============================================================
(PID: 5)
WAITING TIME = 38, TURNAROUND TIME = 51, RESPONSE TIME = 38
============================================================
(PID: 6)
WAITING TIME = 49, TURNAROUND TIME = 65, RESPONSE TIME = 49
============================================================
(PID: 3)
WAITING TIME = 68, TURNAROUND TIME = 90, RESPONSE TIME = 68
============================================================
START TIME: 0 / END TIME: 98 / CPU UTILIZATION : 97.96%
AVERAGE WAITING TIME: 25
AVERAGE TURNAROUND TIME: 38
AVERAGE RESPONSE TIME: 25
COMPLETED: 8
============================================================
```

SJF 알고리즘은 남아있는 CPU Burst Time에 대해서만 고려하고, I/O 작업은 스케줄링 시에 전혀 고려하지 않기 때문에, 이 경우에는 I/O 작업을 수행해야 하는 프로세스가 후반부에 위치하게 되어서 시점 78에서 CPU가 idle 상태가 되는 현상이 발생하였다.

③ Preemptive SJF

```
<PREEMPTIVE SJF ALGORITHM>
0: (PID: 4) -> RUNNING
1: (PID: 4) -> RUNNING
2: (PID: 4) -> RUNNING
3: (PID: 4) -> RUNNING
PREEMPTION IS DETECTED.
4: (PID: 8) -> RUNNING
5: (PID: 8) -> RUNNING
6: (PID: 8) -> RUNNING
7: (PID: 8) -> RUNNING
8: (PID: 8) -> RUNNING
9: (PID: 8) -> RUNNING
10: (PID: 8) -> RUNNING -> TERMINATED
11: (PID: 4) -> RUNNING
12: (PID: 4) -> RUNNING
13: (PID: 4) -> RUNNING
14: (PID: 4) -> RUNNING
PREEMPTION IS DETECTED.
15: (PID: 2) -> RUNNING
16: (PID: 2) -> RUNNING
17: (PID: 2) -> RUNNING
18: (PID: 2) -> RUNNING
19: (PID: 2) -> RUNNING -> TERMINATED
20: (PID: 4) -> RUNNING
21: (PID: 4) -> RUNNING
22: (PID: 4) -> RUNNING
23: (PID: 4) -> RUNNING
24: (PID: 4) -> RUNNING
25: (PID: 4) -> RUNNING
26: (PID: 4) -> RUNNING -> TERMINATED
27: (PID: 1) -> RUNNING -> IO REQUEST
28: (PID: 7) -> RUNNING
29: (PID: 7) -> RUNNING
30: (PID: 7) -> RUNNING
31: (PID: 7) -> RUNNING
32: (PID: 7) -> RUNNING
33: (PID: 1) -> IO COMPLETE, (PID: 7) -> RUNNING
34: (PID: 7) -> RUNNING
35: (PID: 7) -> RUNNING
36: (PID: 7) -> RUNNING
37: (PID: 7) -> RUNNING
38: (PID: 7) -> RUNNING -> TERMINATED
39: (PID: 1) -> RUNNING
40: (PID: 1) -> RUNNING
41: (PID: 1) -> RUNNING
42: (PID: 1) -> RUNNING
43: (PID: 1) -> RUNNING
44: (PID: 1) -> RUNNING
45: (PID: 1) -> RUNNING
46: (PID: 1) -> RUNNING
47: (PID: 1) -> RUNNING -> TERMINATED
48: (PID: 5) -> RUNNING
49: (PID: 5) -> RUNNING
50: (PID: 5) -> RUNNING
51: (PID: 5) -> RUNNING
52: (PID: 5) -> RUNNING
53: (PID: 5) -> RUNNING
54: (PID: 5) -> RUNNING
55: (PID: 5) -> RUNNING
56: (PID: 5) -> RUNNING
57: (PID: 5) -> RUNNING
58: (PID: 5) -> RUNNING
59: (PID: 5) -> RUNNING
60: (PID: 5) -> RUNNING -> TERMINATED
61: (PID: 6) -> RUNNING
62: (PID: 6) -> RUNNING
63: (PID: 6) -> RUNNING
64: (PID: 6) -> RUNNING
65: (PID: 6) -> RUNNING
66: (PID: 6) -> RUNNING
67: (PID: 6) -> RUNNING
68: (PID: 6) -> RUNNING
69: (PID: 6) -> RUNNING
70: (PID: 6) -> RUNNING
71: (PID: 6) -> RUNNING
72: (PID: 6) -> RUNNING
73: (PID: 6) -> RUNNING
74: (PID: 6) -> RUNNING
75: (PID: 6) -> RUNNING
76: (PID: 6) -> RUNNING -> TERMINATED
77: (PID: 3) -> RUNNING -> IO REQUEST
78: IDLE
79: (PID: 3) -> IO COMPLETE, IDLE
80: (PID: 3) -> RUNNING
81: (PID: 3) -> RUNNING
82: (PID: 3) -> RUNNING
83: (PID: 3) -> RUNNING
84: (PID: 3) -> RUNNING
85: (PID: 3) -> RUNNING
86: (PID: 3) -> RUNNING
87: (PID: 3) -> RUNNING
88: (PID: 3) -> RUNNING
89: (PID: 3) -> RUNNING
90: (PID: 3) -> RUNNING
91: (PID: 3) -> RUNNING
92: (PID: 3) -> RUNNING
93: (PID: 3) -> RUNNING
94: (PID: 3) -> RUNNING
95: (PID: 3) -> RUNNING
96: (PID: 3) -> RUNNING
97: (PID: 3) -> RUNNING
98: (PID: 3) -> RUNNING -> TERMINATED
============================================================
(PID: 8)
WAITING TIME = 0, TURNAROUND TIME = 7, RESPONSE TIME = 0
============================================================
(PID: 2)
WAITING TIME = 0, TURNAROUND TIME = 5, RESPONSE TIME = 0
============================================================
(PID: 4)
WAITING TIME = 12, TURNAROUND TIME = 27, RESPONSE TIME = 0
============================================================
(PID: 7)
WAITING TIME = 16, TURNAROUND TIME = 27, RESPONSE TIME = 16
============================================================
```

```
(PID: 1)
WAITING TIME = 19, TURNAROUND TIME = 35, RESPONSE TIME = 14
============================================================
(PID: 5)
WAITING TIME = 38, TURNAROUND TIME = 51, RESPONSE TIME = 38
============================================================
(PID: 6)
WAITING TIME = 49, TURNAROUND TIME = 65, RESPONSE TIME = 49
============================================================
(PID: 3)
WAITING TIME = 68, TURNAROUND TIME = 90, RESPONSE TIME = 68
============================================================
START TIME: 0 / END TIME: 98 / CPU UTILIZATION : 97.96%
AVERAGE WAITING TIME: 25
AVERAGE TURNAROUND TIME: 38
AVERAGE RESPONSE TIME: 23
COMPLETED: 8
============================================================
```

시점 4와 15에서처럼, Preemption이 적용되어 남아있는 CPU Burst Time이 적어서 우선권을 가진 프로세스가 즉각적으로 수행될 수 있도록 하여 Response Time이 향상된 것을 관찰할 수 있다.

④ Non-Preemptive Priority

```
<NON-PREEMPTIVE PRIORITY ALGORITHM>
0: (PID: 4) -> RUNNING
1: (PID: 4) -> RUNNING
2: (PID: 4) -> RUNNING
3: (PID: 4) -> RUNNING
4: (PID: 4) -> RUNNING
5: (PID: 4) -> RUNNING
6: (PID: 4) -> RUNNING
7: (PID: 4) -> RUNNING
8: (PID: 4) -> RUNNING
9: (PID: 4) -> RUNNING
10: (PID: 4) -> RUNNING
11: (PID: 4) -> RUNNING
12: (PID: 4) -> RUNNING
13: (PID: 4) -> RUNNING
14: (PID: 4) -> RUNNING -> TERMINATED
15: (PID: 6) -> RUNNING
16: (PID: 6) -> RUNNING
17: (PID: 6) -> RUNNING
18: (PID: 6) -> RUNNING
19: (PID: 6) -> RUNNING
20: (PID: 6) -> RUNNING
21: (PID: 6) -> RUNNING
22: (PID: 6) -> RUNNING
23: (PID: 6) -> RUNNING
24: (PID: 6) -> RUNNING
25: (PID: 6) -> RUNNING
26: (PID: 6) -> RUNNING
27: (PID: 6) -> RUNNING
28: (PID: 6) -> RUNNING
29: (PID: 6) -> RUNNING
30: (PID: 6) -> RUNNING -> TERMINATED
31: (PID: 2) -> RUNNING
32: (PID: 2) -> RUNNING
33: (PID: 2) -> RUNNING
34: (PID: 2) -> RUNNING
35: (PID: 2) -> RUNNING -> TERMINATED
36: (PID: 5) -> RUNNING
37: (PID: 5) -> RUNNING
38: (PID: 5) -> RUNNING
39: (PID: 5) -> RUNNING
40: (PID: 5) -> RUNNING
41: (PID: 5) -> RUNNING
42: (PID: 5) -> RUNNING
43: (PID: 5) -> RUNNING
44: (PID: 5) -> RUNNING
45: (PID: 5) -> RUNNING
46: (PID: 5) -> RUNNING
47: (PID: 5) -> RUNNING
48: (PID: 5) -> RUNNING -> TERMINATED
49: (PID: 7) -> RUNNING
50: (PID: 7) -> RUNNING
51: (PID: 7) -> RUNNING
52: (PID: 7) -> RUNNING
53: (PID: 7) -> RUNNING
54: (PID: 7) -> RUNNING
55: (PID: 7) -> RUNNING
56: (PID: 7) -> RUNNING
57: (PID: 7) -> RUNNING
58: (PID: 7) -> RUNNING
59: (PID: 7) -> RUNNING -> TERMINATED
60: (PID: 8) -> RUNNING
61: (PID: 8) -> RUNNING
62: (PID: 8) -> RUNNING
63: (PID: 8) -> RUNNING
64: (PID: 8) -> RUNNING
65: (PID: 8) -> RUNNING
66: (PID: 8) -> RUNNING -> TERMINATED
67: (PID: 3) -> RUNNING -> IO REQUEST
68: (PID: 1) -> RUNNING -> IO REQUEST
69: (PID: 3) -> IO COMPLETE, IDLE
70: (PID: 3) -> RUNNING
71: (PID: 3) -> RUNNING
72: (PID: 3) -> RUNNING
73: (PID: 3) -> RUNNING
74: (PID: 1) -> IO COMPLETE, (PID: 3) -> RUNNING
75: (PID: 3) -> RUNNING
76: (PID: 3) -> RUNNING
77: (PID: 3) -> RUNNING
78: (PID: 3) -> RUNNING
79: (PID: 3) -> RUNNING
80: (PID: 3) -> RUNNING
81: (PID: 3) -> RUNNING
82: (PID: 3) -> RUNNING
83: (PID: 3) -> RUNNING
84: (PID: 3) -> RUNNING
85: (PID: 3) -> RUNNING
86: (PID: 3) -> RUNNING
87: (PID: 3) -> RUNNING
88: (PID: 3) -> RUNNING -> TERMINATED
89: (PID: 1) -> RUNNING
90: (PID: 1) -> RUNNING
91: (PID: 1) -> RUNNING
92: (PID: 1) -> RUNNING
93: (PID: 1) -> RUNNING
94: (PID: 1) -> RUNNING
95: (PID: 1) -> RUNNING
96: (PID: 1) -> RUNNING
97: (PID: 1) -> RUNNING -> TERMINATED
============================================================
(PID: 4)
WAITING TIME = 0, TURNAROUND TIME = 15, RESPONSE TIME = 0
============================================================
(PID: 6)
WAITING TIME = 3, TURNAROUND TIME = 19, RESPONSE TIME = 3
============================================================
(PID: 2)
WAITING TIME = 16, TURNAROUND TIME = 21, RESPONSE TIME = 16
============================================================
(PID: 5)
WAITING TIME = 26, TURNAROUND TIME = 39, RESPONSE TIME = 26
============================================================
(PID: 7)
WAITING TIME = 37, TURNAROUND TIME = 48, RESPONSE TIME = 37
============================================================
(PID: 8)
WAITING TIME = 56, TURNAROUND TIME = 63, RESPONSE TIME = 56
============================================================
(PID: 3)
WAITING TIME = 58, TURNAROUND TIME = 80, RESPONSE TIME = 58
============================================================
(PID: 1)
WAITING TIME = 69, TURNAROUND TIME = 85, RESPONSE TIME = 55
============================================================
START TIME: 0 / END TIME: 97 / CPU UTILIZATION : 98.97%
AVERAGE WAITING TIME: 33
AVERAGE TURNAROUND TIME: 46
AVERAGE RESPONSE TIME: 31
COMPLETED: 8
============================================================
```

Priority가 임의의 값으로 주어지기 때문에 성능은 Priority가 어떻게 분배되었는가에 좌우된다.

이 경우 I/O 작업을 수행해야 하는 프로세스들이 우연의 일치로 낮은 우선순위를 얻게 되었는데, I/O Burst Time이 더욱 적은 프로세스 1이 먼저 수행되면서 CPU가 시점 69에서 한 번만 idle한 상태가 되었다.

⑤ Preemptive Priority

```
<Preemptive Priority Algorithm>
0: (pid: 4) -> running
1: (pid: 4) -> running
2: (pid: 4) -> running
3: (pid: 4) -> running
4: (pid: 4) -> running
5: (pid: 4) -> running
6: (pid: 4) -> running
7: (pid: 4) -> running
8: (pid: 4) -> running
9: (pid: 4) -> running
10: (pid: 4) -> running
11: (pid: 4) -> running
12: (pid: 4) -> running
13: (pid: 4) -> running
14: (pid: 4) -> running -> terminated
15: (pid: 6) -> running
16: (pid: 6) -> running
17: (pid: 6) -> running
18: (pid: 6) -> running
19: (pid: 6) -> running
20: (pid: 6) -> running
21: (pid: 6) -> running
22: (pid: 6) -> running
23: (pid: 6) -> running
24: (pid: 6) -> running
25: (pid: 6) -> running
26: (pid: 6) -> running
27: (pid: 6) -> running
28: (pid: 6) -> running
29: (pid: 6) -> running
30: (pid: 6) -> running -> terminated
31: (pid: 2) -> running
32: (pid: 2) -> running
33: (pid: 2) -> running
34: (pid: 2) -> running
35: (pid: 2) -> running -> terminated
36: (pid: 5) -> running
37: (pid: 5) -> running
38: (pid: 5) -> running
39: (pid: 5) -> running
40: (pid: 5) -> running
41: (pid: 5) -> running
42: (pid: 5) -> running
43: (pid: 5) -> running
44: (pid: 5) -> running
45: (pid: 5) -> running
46: (pid: 5) -> running
47: (pid: 5) -> running
48: (pid: 5) -> running -> terminated
49: (pid: 7) -> running
50: (pid: 7) -> running
51: (pid: 7) -> running
52: (pid: 7) -> running
53: (pid: 7) -> running
54: (pid: 7) -> running
55: (pid: 7) -> running
56: (pid: 7) -> running
57: (pid: 7) -> running
58: (pid: 7) -> running
59: (pid: 7) -> running -> terminated
60: (pid: 8) -> running
61: (pid: 8) -> running
62: (pid: 8) -> running
63: (pid: 8) -> running
64: (pid: 8) -> running
65: (pid: 8) -> running
66: (pid: 8) -> running -> terminated
67: (pid: 3) -> running -> io request
68: (pid: 1) -> running -> io request
69: (pid: 3) -> io complete, idle
70: (pid: 3) -> running
71: (pid: 3) -> running
72: (pid: 3) -> running
73: (pid: 3) -> running
74: (pid: 1) -> io complete, (pid: 3) -> running
75: (pid: 3) -> running
76: (pid: 3) -> running
77: (pid: 3) -> running
78: (pid: 3) -> running
79: (pid: 3) -> running
80: (pid: 3) -> running
81: (pid: 3) -> running
82: (pid: 3) -> running
83: (pid: 3) -> running
84: (pid: 3) -> running
85: (pid: 3) -> running
86: (pid: 3) -> running
87: (pid: 3) -> running
88: (pid: 3) -> running -> terminated
89: (pid: 1) -> running
90: (pid: 1) -> running
91: (pid: 1) -> running
92: (pid: 1) -> running
93: (pid: 1) -> running
94: (pid: 1) -> running
95: (pid: 1) -> running
96: (pid: 1) -> running
97: (pid: 1) -> running -> terminated
============================================================
(pid: 4)
waiting time = 0, turnaround time = 15, response time = 0
============================================================
(pid: 6)
waiting time = 3, turnaround time = 19, response time = 3
============================================================
(pid: 2)
waiting time = 16, turnaround time = 21, response time = 16
============================================================
(pid: 5)
waiting time = 26, turnaround time = 39, response time = 26
============================================================
(pid: 7)
waiting time = 37, turnaround time = 48, response time = 37
============================================================
(pid: 8)
waiting time = 56, turnaround time = 63, response time = 56
============================================================
(pid: 3)
waiting time = 58, turnaround time = 80, response time = 58
============================================================
(pid: 1)
waiting time = 69, turnaround time = 85, response time = 55
============================================================
start time: 0 / end time: 97 / cpu utilization : 98.97%
Average waiting time: 33
Average turnaround time: 46
Average response time: 31
Completed: 8
============================================================
```

Priority의 분배가 Preemption이 발생하지 않도록 이루어져서 Preemption이 한 번도 발생하지 않았다. Priority의 분배가 달라지면, Preemption이 발생할 수 있다. 그런데, Average Waiting Time이나 Average Turnaround Time이 줄어든다는 것은 보장할 수 없다. 전적으로 Priority의 분배가 어떻게 이루어지느냐에 따라 성능이 좌우된다.

⑥ Round Robin

```
<Round Robin Algorithm (time quantum: 3)>
0: (pid: 4) -> running
1: (pid: 4) -> running
2: (pid: 4) -> running
3: (pid: 4) -> running
4: (pid: 8) -> running
5: (pid: 8) -> running
6: (pid: 8) -> running
7: (pid: 4) -> running
8: (pid: 4) -> running
9: (pid: 4) -> running
10: (pid: 8) -> running
11: (pid: 8) -> running
12: (pid: 8) -> running
13: (pid: 3) -> running -> IO request
14: (pid: 5) -> running
15: (pid: 3) -> IO complete, (pid: 5) -> running
16: (pid: 5) -> running
17: (pid: 4) -> running
18: (pid: 4) -> running
19: (pid: 4) -> running
20: (pid: 6) -> running
21: (pid: 6) -> running
22: (pid: 6) -> running
23: (pid: 7) -> running
24: (pid: 7) -> running
25: (pid: 7) -> running
26: (pid: 1) -> running -> IO request
27: (pid: 8) -> running -> terminated
28: (pid: 2) -> running
29: (pid: 2) -> running
30: (pid: 2) -> running
31: (pid: 3) -> running
32: (pid: 1) -> IO complete, (pid: 3) -> running
33: (pid: 3) -> running
34: (pid: 5) -> running
35: (pid: 5) -> running
36: (pid: 5) -> running
37: (pid: 4) -> running
38: (pid: 4) -> running
39: (pid: 4) -> running
40: (pid: 6) -> running
41: (pid: 6) -> running
42: (pid: 6) -> running
43: (pid: 7) -> running
44: (pid: 7) -> running
45: (pid: 7) -> running
46: (pid: 2) -> running
47: (pid: 2) -> running -> terminated
48: (pid: 1) -> running
49: (pid: 1) -> running
50: (pid: 1) -> running
51: (pid: 3) -> running
52: (pid: 3) -> running
53: (pid: 3) -> running
54: (pid: 5) -> running
55: (pid: 5) -> running
56: (pid: 5) -> running
57: (pid: 4) -> running
58: (pid: 4) -> running -> terminated
59: (pid: 6) -> running
60: (pid: 6) -> running
61: (pid: 6) -> running
62: (pid: 7) -> running
63: (pid: 7) -> running
64: (pid: 7) -> running
65: (pid: 1) -> running
66: (pid: 1) -> running
67: (pid: 1) -> running
68: (pid: 3) -> running
69: (pid: 3) -> running
70: (pid: 3) -> running
71: (pid: 5) -> running
72: (pid: 5) -> running
73: (pid: 5) -> running
74: (pid: 6) -> running
75: (pid: 6) -> running
76: (pid: 6) -> running
77: (pid: 7) -> running
78: (pid: 7) -> running -> terminated
79: (pid: 1) -> running
80: (pid: 1) -> running
81: (pid: 1) -> running -> terminated
82: (pid: 3) -> running
83: (pid: 3) -> running
84: (pid: 3) -> running
85: (pid: 5) -> running -> terminated
86: (pid: 6) -> running
87: (pid: 6) -> running
88: (pid: 6) -> running
89: (pid: 3) -> running
90: (pid: 3) -> running
91: (pid: 3) -> running
92: (pid: 6) -> running -> terminated
93: (pid: 3) -> running
94: (pid: 3) -> running
95: (pid: 3) -> running
96: (pid: 3) -> running -> terminated
============================================================
(pid: 8)
waiting time = 17, turnaround time = 24, response time = 0
============================================================
(pid: 2)
waiting time = 28, turnaround time = 33, response time = 13
============================================================
(pid: 4)
waiting time = 44, turnaround time = 59, response time = 0
============================================================
(pid: 7)
waiting time = 56, turnaround time = 67, response time = 11
============================================================
(pid: 1)
waiting time = 53, turnaround time = 69, response time = 13
============================================================
(pid: 5)
waiting time = 63, turnaround time = 76, response time = 4
============================================================
(pid: 6)
waiting time = 65, turnaround time = 81, response time = 8
============================================================
(pid: 3)
waiting time = 66, turnaround time = 88, response time = 4
============================================================
start time: 0 / end time: 96 / CPU utilization : 100.00%
Average waiting time: 49
Average turnaround time: 62
Average response time: 6
Completed: 8
============================================================
```

시점 4가 되기 전까지는 Ready Queue에 도착한 프로세스가 4번 프로세스밖에 없기 때문에, 주어진 Time Quantum인 3을 초과하여 4의 시간만큼 수행되었다. 8번 프로세스가 도착한 뒤에는 정상적으로 3의 시간만큼 FCFS의 순서대로 프로세스가 수행되었다. 시점 13 또는 47에서처럼 특정 프로세스가 주어진 Time Quantum을 다 소모하지 못한 상태로 Running State에서 빠져나오게 된다면, CPU를 idle상태로 두지 않고 즉각적으로 다음 프로세스를 수행시켜서 CPU Utilization을 높였다. 그리고 프로세스가 도착한 뒤에 짧은 시간 안에 수행을 할 기회가 주어지게 되므로, Round Robin 방식의 장점 그대로 Average Response Time이 매우 적게 나온 것을 확인할 수 있다.

⑦ Non-Preemptive LIF

```
<Non-preemptive LIF Algorithm>
0: (pid: 4) -> running
1: (pid: 4) -> running
36: (pid: 3) -> running
37: (pid: 3) -> running
38: (pid: 3) -> running
71: (pid: 6) -> running -> terminated
72: (pid: 7) -> running
73: (pid: 7) -> running
```

```
2: (PID: 4) -> RUNNING
3: (PID: 4) -> RUNNING
4: (PID: 4) -> RUNNING
5: (PID: 4) -> RUNNING
6: (PID: 4) -> RUNNING
7: (PID: 4) -> RUNNING
8: (PID: 4) -> RUNNING
9: (PID: 4) -> RUNNING
10: (PID: 4) -> RUNNING
11: (PID: 4) -> RUNNING
12: (PID: 4) -> RUNNING
13: (PID: 4) -> RUNNING
14: (PID: 4) -> RUNNING -> TERMINATED
15: (PID: 1) -> RUNNING -> IO REQUEST
16: (PID: 3) -> RUNNING -> IO REQUEST
17: (PID: 8) -> RUNNING
18: (PID: 3) -> IO COMPLETE, (PID: 8) -> RUNNING
19: (PID: 8) -> RUNNING
20: (PID: 8) -> RUNNING
21: (PID: 1) -> IO COMPLETE, (PID: 8) -> RUNNING
22: (PID: 8) -> RUNNING
23: (PID: 8) -> RUNNING -> TERMINATED
24: (PID: 3) -> RUNNING
25: (PID: 3) -> RUNNING
26: (PID: 3) -> RUNNING
27: (PID: 3) -> RUNNING
28: (PID: 3) -> RUNNING
29: (PID: 3) -> RUNNING
30: (PID: 3) -> RUNNING
31: (PID: 3) -> RUNNING
32: (PID: 3) -> RUNNING
33: (PID: 3) -> RUNNING
34: (PID: 3) -> RUNNING
35: (PID: 3) -> RUNNING
39: (PID: 3) -> RUNNING
40: (PID: 3) -> RUNNING
41: (PID: 3) -> RUNNING
42: (PID: 3) -> RUNNING -> TERMINATED
43: (PID: 5) -> RUNNING
44: (PID: 5) -> RUNNING
45: (PID: 5) -> RUNNING
46: (PID: 5) -> RUNNING
47: (PID: 5) -> RUNNING
48: (PID: 5) -> RUNNING
49: (PID: 5) -> RUNNING
50: (PID: 5) -> RUNNING
51: (PID: 5) -> RUNNING
52: (PID: 5) -> RUNNING
53: (PID: 5) -> RUNNING
54: (PID: 5) -> RUNNING
55: (PID: 5) -> RUNNING -> TERMINATED
56: (PID: 6) -> RUNNING
57: (PID: 6) -> RUNNING
58: (PID: 6) -> RUNNING
59: (PID: 6) -> RUNNING
60: (PID: 6) -> RUNNING
61: (PID: 6) -> RUNNING
62: (PID: 6) -> RUNNING
63: (PID: 6) -> RUNNING
64: (PID: 6) -> RUNNING
65: (PID: 6) -> RUNNING
66: (PID: 6) -> RUNNING
67: (PID: 6) -> RUNNING
68: (PID: 6) -> RUNNING
69: (PID: 6) -> RUNNING
70: (PID: 6) -> RUNNING
74: (PID: 7) -> RUNNING
75: (PID: 7) -> RUNNING
76: (PID: 7) -> RUNNING
77: (PID: 7) -> RUNNING
78: (PID: 7) -> RUNNING
79: (PID: 7) -> RUNNING
80: (PID: 7) -> RUNNING
81: (PID: 7) -> RUNNING
82: (PID: 7) -> RUNNING -> TERMINATED
83: (PID: 1) -> RUNNING
84: (PID: 1) -> RUNNING
85: (PID: 1) -> RUNNING
86: (PID: 1) -> RUNNING
87: (PID: 1) -> RUNNING
88: (PID: 1) -> RUNNING
89: (PID: 1) -> RUNNING
90: (PID: 1) -> RUNNING
91: (PID: 1) -> RUNNING -> TERMINATED
92: (PID: 2) -> RUNNING
93: (PID: 2) -> RUNNING
94: (PID: 2) -> RUNNING
95: (PID: 2) -> RUNNING
96: (PID: 2) -> RUNNING -> TERMINATED
============================================================
(PID: 4)
WAITING TIME = 0, TURNAROUND TIME = 15, RESPONSE TIME = 0
============================================================
(PID: 8)
WAITING TIME = 13, TURNAROUND TIME = 20, RESPONSE TIME = 13
============================================================
(PID: 3)
WAITING TIME = 12, TURNAROUND TIME = 34, RESPONSE TIME = 7
============================================================
(PID: 5)
WAITING TIME = 33, TURNAROUND TIME = 46, RESPONSE TIME = 33
============================================================
(PID: 6)
WAITING TIME = 44, TURNAROUND TIME = 60, RESPONSE TIME = 44
============================================================
(PID: 7)
WAITING TIME = 60, TURNAROUND TIME = 71, RESPONSE TIME = 60
============================================================
(PID: 1)
WAITING TIME = 63, TURNAROUND TIME = 79, RESPONSE TIME = 2
============================================================
(PID: 2)
WAITING TIME = 77, TURNAROUND TIME = 82, RESPONSE TIME = 77
============================================================
START TIME: 0 / END TIME: 96 / CPU UTILIZATION : 100.00%
AVERAGE WAITING TIME: 37
AVERAGE TURNAROUND TIME: 50
AVERAGE RESPONSE TIME: 29
COMPLETED: 8
============================================================
```

CPU Utilization을 극대화하기 위하여 남아있는 I/O Burst Time이 긴 프로세스를 우선적으로 처리하기 때문에 CPU가 idle해질 확률이 매우 적다. 따라서 시뮬레이션 결과에서도 CPU Utilization이 100%로 나타났다. I/O 작업을 수행하는 프로세스를 제외하고는 FCFS 방식으로 Scheduling이 되었다.

⑧ Preemptive LIF

```
<PREEMPTIVE LIF ALGORITHM>
0: (PID: 4) -> RUNNING
1: (PID: 4) -> RUNNING
2: (PID: 4) -> RUNNING
3: (PID: 4) -> RUNNING
4: (PID: 4) -> RUNNING
5: (PID: 4) -> RUNNING
6: (PID: 4) -> RUNNING
7: (PID: 4) -> RUNNING
8: (PID: 4) -> RUNNING
PREEMPTION IS DETECTED.
9: (PID: 3) -> RUNNING -> IO REQUEST
10: (PID: 4) -> RUNNING
11: (PID: 3) -> IO COMPLETE, (PID: 4) -> RUNNING
12: (PID: 4) -> RUNNING
PREEMPTION IS DETECTED.
13: (PID: 1) -> RUNNING -> IO REQUEST
14: (PID: 4) -> RUNNING
35: (PID: 3) -> RUNNING
36: (PID: 3) -> RUNNING
37: (PID: 3) -> RUNNING
38: (PID: 3) -> RUNNING
39: (PID: 3) -> RUNNING
40: (PID: 3) -> RUNNING
41: (PID: 3) -> RUNNING
42: (PID: 3) -> RUNNING -> TERMINATED
43: (PID: 5) -> RUNNING
44: (PID: 5) -> RUNNING
45: (PID: 5) -> RUNNING
46: (PID: 5) -> RUNNING
47: (PID: 5) -> RUNNING
48: (PID: 5) -> RUNNING
49: (PID: 5) -> RUNNING
50: (PID: 5) -> RUNNING
51: (PID: 5) -> RUNNING
52: (PID: 5) -> RUNNING
71: (PID: 6) -> RUNNING -> TERMINATED
72: (PID: 7) -> RUNNING
73: (PID: 7) -> RUNNING
74: (PID: 7) -> RUNNING
75: (PID: 7) -> RUNNING
76: (PID: 7) -> RUNNING
77: (PID: 7) -> RUNNING
78: (PID: 7) -> RUNNING
79: (PID: 7) -> RUNNING
80: (PID: 7) -> RUNNING
81: (PID: 7) -> RUNNING
82: (PID: 7) -> RUNNING -> TERMINATED
83: (PID: 1) -> RUNNING
84: (PID: 1) -> RUNNING
85: (PID: 1) -> RUNNING
86: (PID: 1) -> RUNNING
87: (PID: 1) -> RUNNING
88: (PID: 1) -> RUNNING
```

```
15: (PID: 4) -> RUNNING
16: (PID: 4) -> RUNNING -> TERMINATED
17: (PID: 8) -> RUNNING
18: (PID: 8) -> RUNNING
19: (PID: 1) -> IO COMPLETE, (PID: 8) -> RUNNING
20: (PID: 8) -> RUNNING
21: (PID: 8) -> RUNNING
22: (PID: 8) -> RUNNING
23: (PID: 8) -> RUNNING -> TERMINATED
24: (PID: 3) -> RUNNING
25: (PID: 3) -> RUNNING
26: (PID: 3) -> RUNNING
27: (PID: 3) -> RUNNING
28: (PID: 3) -> RUNNING
29: (PID: 3) -> RUNNING
30: (PID: 3) -> RUNNING
31: (PID: 3) -> RUNNING
32: (PID: 3) -> RUNNING
33: (PID: 3) -> RUNNING
34: (PID: 3) -> RUNNING
35: (PID: 3) -> RUNNING
53: (PID: 5) -> RUNNING
54: (PID: 5) -> RUNNING
55: (PID: 5) -> RUNNING -> TERMINATED
56: (PID: 6) -> RUNNING
57: (PID: 6) -> RUNNING
58: (PID: 6) -> RUNNING
59: (PID: 6) -> RUNNING
60: (PID: 6) -> RUNNING
61: (PID: 6) -> RUNNING
62: (PID: 6) -> RUNNING
63: (PID: 6) -> RUNNING
64: (PID: 6) -> RUNNING
65: (PID: 6) -> RUNNING
66: (PID: 6) -> RUNNING
67: (PID: 6) -> RUNNING
68: (PID: 6) -> RUNNING
69: (PID: 6) -> RUNNING
70: (PID: 6) -> RUNNING
89: (PID: 1) -> RUNNING
90: (PID: 1) -> RUNNING
91: (PID: 1) -> RUNNING -> TERMINATED
92: (PID: 2) -> RUNNING
93: (PID: 2) -> RUNNING
94: (PID: 2) -> RUNNING
95: (PID: 2) -> RUNNING
96: (PID: 2) -> RUNNING -> TERMINATED
============================================================
(PID: 4)
WAITING TIME = 2, TURNAROUND TIME = 17, RESPONSE TIME = 0
============================================================
(PID: 8)
WAITING TIME = 13, TURNAROUND TIME = 20, RESPONSE TIME = 13
============================================================
(PID: 3)
WAITING TIME = 12, TURNAROUND TIME = 34, RESPONSE TIME = 0
============================================================
(PID: 5)
WAITING TIME = 33, TURNAROUND TIME = 46, RESPONSE TIME = 33
============================================================
(PID: 6)
WAITING TIME = 44, TURNAROUND TIME = 60, RESPONSE TIME = 44
============================================================
(PID: 7)
WAITING TIME = 60, TURNAROUND TIME = 71, RESPONSE TIME = 60
============================================================
(PID: 1)
WAITING TIME = 63, TURNAROUND TIME = 79, RESPONSE TIME = 0
============================================================
(PID: 2)
WAITING TIME = 77, TURNAROUND TIME = 82, RESPONSE TIME = 77
============================================================
START TIME: 0 / END TIME: 96 / CPU UTILIZATION : 100.00%
AVERAGE WAITING TIME: 38
AVERAGE TURNAROUND TIME: 51
AVERAGE RESPONSE TIME: 28
COMPLETED: 8
============================================================
```

Preemption을 적용하여 I/O 작업을 수행할 필요가 있는 프로세스들이 즉각적으로 수행될 수 있도록 하여 Response Time을 낮추었다. 하지만, 시점 9에서 13에 이르는 부분에서처럼 잦은 Context Switch가 발생하여서 전체적인 Average Waiting Time과 Average Turnaround Time은 오히려 Non-Preemption의 경우보다 증가할 수 있다는 것을 확인할 수 있다. 이 시뮬레이터에서는 Context Switching Time을 고려하지 않았지만, 실제 환경에서는 Context Switching Time 때문에 오버헤드가 더욱 크게 나타날 것을 예상할 수 있다.

⑨ Non-Preemptive LISC

```
<NON-PREEMPTIVE LISC ALGORITHM>
0: (PID: 4) -> RUNNING
1: (PID: 4) -> RUNNING
2: (PID: 4) -> RUNNING
3: (PID: 4) -> RUNNING
4: (PID: 4) -> RUNNING
5: (PID: 4) -> RUNNING
6: (PID: 4) -> RUNNING
7: (PID: 4) -> RUNNING
8: (PID: 4) -> RUNNING
9: (PID: 4) -> RUNNING
10: (PID: 4) -> RUNNING
11: (PID: 4) -> RUNNING
12: (PID: 4) -> RUNNING
13: (PID: 4) -> RUNNING
14: (PID: 4) -> RUNNING -> TERMINATED
15: (PID: 1) -> RUNNING -> IO REQUEST
16: (PID: 3) -> RUNNING -> IO REQUEST
17: (PID: 2) -> RUNNING
18: (PID: 3) -> IO COMPLETE, (PID: 2) -> RUNNING
19: (PID: 2) -> RUNNING
20: (PID: 2) -> RUNNING
21: (PID: 1) -> IO COMPLETE, (PID: 2) -> RUNNING -> TERMINATED
22: (PID: 8) -> RUNNING
23: (PID: 8) -> RUNNING
24: (PID: 8) -> RUNNING
36: (PID: 1) -> RUNNING
37: (PID: 1) -> RUNNING -> TERMINATED
38: (PID: 7) -> RUNNING
39: (PID: 7) -> RUNNING
40: (PID: 7) -> RUNNING
41: (PID: 7) -> RUNNING
42: (PID: 7) -> RUNNING
43: (PID: 7) -> RUNNING
44: (PID: 7) -> RUNNING
45: (PID: 7) -> RUNNING
46: (PID: 7) -> RUNNING
47: (PID: 7) -> RUNNING
48: (PID: 7) -> RUNNING -> TERMINATED
49: (PID: 5) -> RUNNING
50: (PID: 5) -> RUNNING
51: (PID: 5) -> RUNNING
52: (PID: 5) -> RUNNING
53: (PID: 5) -> RUNNING
54: (PID: 5) -> RUNNING
55: (PID: 5) -> RUNNING
56: (PID: 5) -> RUNNING
57: (PID: 5) -> RUNNING
58: (PID: 5) -> RUNNING
59: (PID: 5) -> RUNNING
60: (PID: 5) -> RUNNING
61: (PID: 5) -> RUNNING -> TERMINATED
71: (PID: 6) -> RUNNING
72: (PID: 6) -> RUNNING
73: (PID: 6) -> RUNNING
74: (PID: 6) -> RUNNING
75: (PID: 6) -> RUNNING
76: (PID: 6) -> RUNNING
77: (PID: 6) -> RUNNING -> TERMINATED
78: (PID: 3) -> RUNNING
79: (PID: 3) -> RUNNING
80: (PID: 3) -> RUNNING
81: (PID: 3) -> RUNNING
82: (PID: 3) -> RUNNING
83: (PID: 3) -> RUNNING
84: (PID: 3) -> RUNNING
85: (PID: 3) -> RUNNING
86: (PID: 3) -> RUNNING
87: (PID: 3) -> RUNNING
88: (PID: 3) -> RUNNING
89: (PID: 3) -> RUNNING
90: (PID: 3) -> RUNNING
91: (PID: 3) -> RUNNING
92: (PID: 3) -> RUNNING
93: (PID: 3) -> RUNNING
94: (PID: 3) -> RUNNING
95: (PID: 3) -> RUNNING
96: (PID: 3) -> RUNNING -> TERMINATED
```

```
25: (PID: 8) -> RUNNING
26: (PID: 8) -> RUNNING
27: (PID: 8) -> RUNNING
28: (PID: 8) -> RUNNING -> TERMINATED
29: (PID: 1) -> RUNNING
30: (PID: 1) -> RUNNING
31: (PID: 1) -> RUNNING
32: (PID: 1) -> RUNNING
33: (PID: 1) -> RUNNING
34: (PID: 1) -> RUNNING
35: (PID: 1) -> RUNNING
62: (PID: 6) -> RUNNING
63: (PID: 6) -> RUNNING
64: (PID: 6) -> RUNNING
65: (PID: 6) -> RUNNING
66: (PID: 6) -> RUNNING
67: (PID: 6) -> RUNNING
68: (PID: 6) -> RUNNING
69: (PID: 6) -> RUNNING
70: (PID: 6) -> RUNNING
============================================================
(PID: 4)
WAITING TIME = 0, TURNAROUND TIME = 15, RESPONSE TIME = 0
============================================================
(PID: 2)
WAITING TIME = 2, TURNAROUND TIME = 7, RESPONSE TIME = 2
============================================================
(PID: 8)
WAITING TIME = 18, TURNAROUND TIME = 25, RESPONSE TIME = 18
============================================================
(PID: 1)
WAITING TIME = 9, TURNAROUND TIME = 25, RESPONSE TIME = 2
============================================================
(PID: 7)
WAITING TIME = 26, TURNAROUND TIME = 37, RESPONSE TIME = 26
============================================================
(PID: 5)
WAITING TIME = 39, TURNAROUND TIME = 52, RESPONSE TIME = 39
============================================================
(PID: 6)
WAITING TIME = 50, TURNAROUND TIME = 66, RESPONSE TIME = 50
============================================================
(PID: 3)
WAITING TIME = 66, TURNAROUND TIME = 88, RESPONSE TIME = 7
============================================================
START TIME: 0 / END TIME: 96 / CPU UTILIZATION : 100.00%
AVERAGE WAITING TIME: 26
AVERAGE TURNAROUND TIME: 39
AVERAGE RESPONSE TIME: 18
COMPLETED: 8
============================================================
```

I/O Burst Time을 우선적으로 고려해주고, CPU Burst Time도 고려하기 때문에, CPU Utilization 측면과 Average Waiting Time 및 Average Turnaround Time 측면에서도 좋은 효율을 보인다.

⑩ Preemptive LISC

```
<PREEMPTIVE LISC ALGORITHM>
0: (PID: 4) -> RUNNING
1: (PID: 4) -> RUNNING
2: (PID: 4) -> RUNNING
3: (PID: 4) -> RUNNING
PREEMPTION IS DETECTED.
4: (PID: 8) -> RUNNING
5: (PID: 8) -> RUNNING
6: (PID: 8) -> RUNNING
7: (PID: 8) -> RUNNING
8: (PID: 8) -> RUNNING
PREEMPTION IS DETECTED.
9: (PID: 3) -> RUNNING -> IO REQUEST
10: (PID: 8) -> RUNNING
11: (PID: 3) -> IO COMPLETE, (PID: 8) -> RUNNING -> TERMINATED
12: (PID: 4) -> RUNNING
PREEMPTION IS DETECTED.
13: (PID: 1) -> RUNNING -> IO REQUEST
14: (PID: 4) -> RUNNING
PREEMPTION IS DETECTED.
15: (PID: 2) -> RUNNING
16: (PID: 2) -> RUNNING
17: (PID: 2) -> RUNNING
18: (PID: 2) -> RUNNING
19: (PID: 1) -> IO COMPLETE, (PID: 2) -> RUNNING -> TERMINATED
20: (PID: 4) -> RUNNING
21: (PID: 4) -> RUNNING
22: (PID: 4) -> RUNNING
23: (PID: 4) -> RUNNING
24: (PID: 4) -> RUNNING
25: (PID: 4) -> RUNNING
26: (PID: 4) -> RUNNING
27: (PID: 4) -> RUNNING
28: (PID: 4) -> RUNNING -> TERMINATED
29: (PID: 1) -> RUNNING
30: (PID: 1) -> RUNNING
31: (PID: 1) -> RUNNING
32: (PID: 1) -> RUNNING
33: (PID: 1) -> RUNNING
34: (PID: 1) -> RUNNING
35: (PID: 1) -> RUNNING
36: (PID: 1) -> RUNNING
37: (PID: 1) -> RUNNING -> TERMINATED
38: (PID: 7) -> RUNNING
39: (PID: 7) -> RUNNING
40: (PID: 7) -> RUNNING
41: (PID: 7) -> RUNNING
42: (PID: 7) -> RUNNING
43: (PID: 7) -> RUNNING
44: (PID: 7) -> RUNNING
45: (PID: 7) -> RUNNING
46: (PID: 7) -> RUNNING
47: (PID: 7) -> RUNNING
48: (PID: 7) -> RUNNING -> TERMINATED
49: (PID: 5) -> RUNNING
50: (PID: 5) -> RUNNING
51: (PID: 5) -> RUNNING
52: (PID: 5) -> RUNNING
53: (PID: 5) -> RUNNING
54: (PID: 5) -> RUNNING
55: (PID: 5) -> RUNNING
56: (PID: 5) -> RUNNING
57: (PID: 5) -> RUNNING
58: (PID: 5) -> RUNNING
59: (PID: 5) -> RUNNING
60: (PID: 5) -> RUNNING
61: (PID: 5) -> RUNNING -> TERMINATED
62: (PID: 6) -> RUNNING
63: (PID: 6) -> RUNNING
64: (PID: 6) -> RUNNING
65: (PID: 6) -> RUNNING
66: (PID: 6) -> RUNNING
67: (PID: 6) -> RUNNING
68: (PID: 6) -> RUNNING
69: (PID: 6) -> RUNNING
70: (PID: 6) -> RUNNING
71: (PID: 6) -> RUNNING
72: (PID: 6) -> RUNNING
73: (PID: 6) -> RUNNING
74: (PID: 6) -> RUNNING
75: (PID: 6) -> RUNNING
76: (PID: 6) -> RUNNING
77: (PID: 6) -> RUNNING -> TERMINATED
78: (PID: 3) -> RUNNING
79: (PID: 3) -> RUNNING
80: (PID: 3) -> RUNNING
81: (PID: 3) -> RUNNING
82: (PID: 3) -> RUNNING
83: (PID: 3) -> RUNNING
84: (PID: 3) -> RUNNING
85: (PID: 3) -> RUNNING
86: (PID: 3) -> RUNNING
87: (PID: 3) -> RUNNING
88: (PID: 3) -> RUNNING
89: (PID: 3) -> RUNNING
90: (PID: 3) -> RUNNING
91: (PID: 3) -> RUNNING
92: (PID: 3) -> RUNNING
93: (PID: 3) -> RUNNING
94: (PID: 3) -> RUNNING
95: (PID: 3) -> RUNNING
96: (PID: 3) -> RUNNING -> TERMINATED
============================================================
(PID: 8)
WAITING TIME = 1, TURNAROUND TIME = 8, RESPONSE TIME = 0
============================================================
(PID: 2)
WAITING TIME = 0, TURNAROUND TIME = 5, RESPONSE TIME = 0
```

```
============================================================
(PID: 4)
WAITING TIME = 14, TURNAROUND TIME = 29, RESPONSE TIME = 0
============================================================
(PID: 1)
WAITING TIME = 9, TURNAROUND TIME = 25, RESPONSE TIME = 0
============================================================
(PID: 7)
WAITING TIME = 26, TURNAROUND TIME = 37, RESPONSE TIME = 26
============================================================
(PID: 5)
WAITING TIME = 39, TURNAROUND TIME = 52, RESPONSE TIME = 39
============================================================
(PID: 6)
WAITING TIME = 50, TURNAROUND TIME = 66, RESPONSE TIME = 50
============================================================
(PID: 3)
WAITING TIME = 66, TURNAROUND TIME = 88, RESPONSE TIME = 0
============================================================
START TIME: 0 / END TIME: 96 / CPU UTILIZATION : 100.00%
AVERAGE WAITING TIME: 25
AVERAGE TURNAROUND TIME: 38
AVERAGE RESPONSE TIME: 14
COMPLETED: 8
============================================================
```

Preemption을 더해서 CPU Utilization, Average Waiting Time, Average Turnaround Time 뿐만 아니라 Average Response Time에서도 좋은 효율을 나타낸다. 단점은 Context Switch가 자주 발생한다는 것이다. 시점 8에서 시점 15까지의 기간 동안 거의 매 시간 단위마다 한 번의 Context Switch가 발생하여 총 6번의 Context Switch가 발생하였다. 실제 환경에서 Context Switching Time을 고려한다면 오버헤드가 다소 크게 나타날 것임을 확인할 수 있다.

(3) 비교분석

Gantt Chart를 제외하고 각 알고리즘들의 성능 측정 결과를 요약하면 다음과 같다.

```
============================================================  ============================================================
<FCFS Algorithm>                                              <Round Robin Algorithm>
------------------------------------------------------------  ------------------------------------------------------------
start time: 0 / end time: 96 / CPU utilization : 100.00%      start time: 0 / end time: 96 / CPU utilization : 100.00%
Average waiting time: 32                                      Average waiting time: 49
Average turnaround time: 45                                   Average turnaround time: 62
Average response time: 25                                     Average response time: 6
Completed: 8                                                  Completed: 8
============================================================  ============================================================
<Non-preemptive SJF Algorithm>                                <Non-preemptive LIF Algorithm>
------------------------------------------------------------  ------------------------------------------------------------
start time: 0 / end time: 98 / CPU utilization : 97.96%       start time: 0 / end time: 96 / CPU utilization : 100.00%
Average waiting time: 25                                      Average waiting time: 37
Average turnaround time: 38                                   Average turnaround time: 50
Average response time: 25                                     Average response time: 29
Completed: 8                                                  Completed: 8
============================================================  ============================================================
<Preemptive SJF Algorithm>                                    <Preemptive LIF Algorithm>
------------------------------------------------------------  ------------------------------------------------------------
start time: 0 / end time: 98 / CPU utilization : 97.96%       start time: 0 / end time: 96 / CPU utilization : 100.00%
Average waiting time: 25                                      Average waiting time: 38
Average turnaround time: 38                                   Average turnaround time: 51
Average response time: 23                                     Average response time: 28
Completed: 8                                                  Completed: 8
============================================================  ============================================================
<Non-preemptive Priority Algorithm>                           <Non-preemptive LISC Algorithm>
------------------------------------------------------------  ------------------------------------------------------------
start time: 0 / end time: 97 / CPU utilization : 98.97%       start time: 0 / end time: 96 / CPU utilization : 100.00%
Average waiting time: 33                                      Average waiting time: 26
Average turnaround time: 46                                   Average turnaround time: 39
Average response time: 31                                     Average response time: 18
Completed: 8                                                  Completed: 8
============================================================  ============================================================
<Preemptive Priority Algorithm>                               <Preemptive LISC Algorithm>
------------------------------------------------------------  ------------------------------------------------------------
start time: 0 / end time: 97 / CPU utilization : 98.97%       start time: 0 / end time: 96 / CPU utilization : 100.00%
Average waiting time: 33                                      Average waiting time: 25
Average turnaround time: 46                                   Average turnaround time: 38
Average response time: 31                                     Average response time: 14
Completed: 8                                                  Completed: 8
============================================================  ============================================================
```

① CPU Utilization

일반적인 경우에, I/O 작업을 수행하는 프로세스들이 우선적으로 Scheduling되는 것이 CPU Utilization을 높이는데 일조한다. 이 시뮬레이션의 경우 주어진 프로세스 중 I/O 작업을 수행하는 1번 프로세스의 Arrival Time이 다른 프로세스들에 비해 다소 늦기 때문에 사실상 가장 기본적인 FCFS 방식으로 Scheduling을 한다면, CPU Utilization이 낮아질 수 있는 위험이 존재한다. 하지만, 전체적인 Arrival Time이 초반부에 몰려있기 때문에 CPU가 idle한 상태가 발생하지 않았다. 만약 1번 프로세스가 다른 프로세스들이 완료되어가는 시점에 충분히 늦게 Ready Queue에 도착을 하게 되었다면, 1번 프로세스가 Waiting Queue에서 I/O 작업을 처리하는 동안 CPU가 idle한 상태가 되어서 CPU Utilization이 현저히 낮아졌을 것이라고 예상할 수 있다. 따라서 LIF와 LISC 방식이 다른 방식들에 비해서 CPU Utilization 면에서 뛰어나다고 할 수 있다. 하지만, Ready Queue에 존재하는 프로세스의 수가 상대적으로 적을 때 I/O 작업을 수행하는 프로세스들을 먼저 Scheduling해 줄 경우 CPU가 idle 상태가 될 확률이 높아지기 때문에 주의해야 한다.

② Average Waiting Time & Average Turnaround Time

FCFS 방식과 비교해보았을 때, SJF 방식이 더욱 좋은 성능을 보인다. CPU Utilization이 100%가 아니어서 모든 프로세스가 완료된 시간이 더 늦춰졌음에도 불구하고, Average Waiting Time과 Average Turnaround Time이 상대적으로 더 낮게 측정되었다.

Priority 방식의 경우 Priority가 어떻게 분배되느냐에 따라 성능이 좌우되기 때문에, 이 경우에는 FCFS 방식보다 오히려 더 성능이 떨어지는 것을 관찰할 수 있다.

Round Robin 방식은 모든 알고리즘들 중에서 가장 Average Waiting Time과 Average Turnaround Time이 높게 측정되었다. 이는 모든 프로세스가 공평하게 time quantum을 부여받아서 수행된다는 특징 때문에 당연한 결과이다.

LIF 방식의 경우 CPU Utilization에만 초점을 두었으며, 기본적인 구조는 FCFS 방식을 따르기 때문에 Average Waiting Time과 Average Turnaround Time은 비교적 큰 값을 가진다. 오히려 FCFS보다 높은 값을 가지게 되었는데, 그 이유는 I/O 작업을 수행하는 프로세스가 우선적으로 Scheduling되어서 1만큼의 시간을 Running State에서 머물다가 Waiting Queue로 이동하기 때문이다. Waiting Queue에 있던 프로세스가 I/O 작업이 끝난 뒤에 다시 Ready Queue로 돌아갈 때는 뒤늦게 도착한 프로세스로 처리가 되기 때문에, 오히려 처음 Ready Queue에 도착한 시간이 자신보다 늦은 프로세스가 Running State일 경우, 수행이 끝날 때까지 기다리게 된다.

LISC 방식의 경우 LIF의 단점을 보완하여 남은 I/O Burst Time이 같은 프로세스들 사이에서는 SJF 방식으로 Scheduling을 하도록 설계가 되어있기 때문에 SJF 방식과 거의 같은 Average Waiting Time과 Average Turnaround Time이 결과로 나타났다.

결론적으로 Average Waiting Time과 Average Turnaround Time의 관점에서는 SJF 방식과 LISC

방식이 가장 효율적이다.

③ **Average Response Time**

전체적으로 Preemption이 적용된 알고리즘들의 Average Response Time이 상대적으로 낮게 나온 것을 확인할 수 있다. 어떤 방식이든 특정 기준에 의해 선정된 프로세스가 즉각적으로 수행될 수 있도록 해주는 것이 Preemption이기 때문에 당연한 결과이다.

FCFS 방식과 비교했을 때, SJF, LISC, 그리고 Round Robin 방식이 상대적으로 낮은 Average Response Time을 나타냈다. CPU Burst Time만을 고려한 SJF 방식보다는 I/O Burst Time도 함께 고려해준 LISC 방식이 더 높은 효율을 보였으며, 독단적으로 가장 짧은 Response Time을 나타낸 방식은 Round Robin 알고리즘이다. 프로세스들이 Ready Queue에 도착하는 대로 빠른 시간 내에 time quantum만큼 수행될 수 있도록 보장을 받기 때문에 당연한 결과이다.

따라서 Average Response Time의 측면에서는 Round Robin 방식이 가장 효율적이며, LISC 알고리즘도 성능이 준수한 편이라고 할 수 있다.

④ **결론**

종합적으로 살펴보았을 때, 모든 면에서 준수한 성능을 내는 것은 LISC 알고리즘이다. SJF 방식은 Average Waiting Time과 Average Turnaround Time이 낮은 편이지만, I/O 작업을 고려해주지 않기 때문에 I/O 작업이 더욱 많이 발생하는 환경에서 효율이 떨어질 것이다. Round Robin 방식은 Average Response Time의 측면에서 최고의 효율을 내지만, Average Waiting Time과 Average Turnaround Time의 측면에서 가장 낮은 효율을 나타내기 때문에 이상적이지 않다.

5. 프로젝트 수행 소감 및 개선해야 할 점

직접 CPU Scheduling Simulator를 구현해보면서 프로세스의 생명주기와 Scheduling Algorithm에 대해 깊이 있는 학습을 하는 좋은 계기였다. 모듈화를 통해 시뮬레이터를 구현하면서 단계적인 과정에 대해 세밀하게 탐구할 수 있었고, 프로세스 Scheduling이 이루어지는 과정을 명확히 이해할 수 있게 되었다.

그런데, 구현한 시뮬레이션 모듈의 구조상 Earliest Deadline First 알고리즘과 같은 Priority Based 알고리즘을 추가적으로 구현하는 것에 어려움이 있었다. 그리고 멀티 프로세서 환경을 제공하지 못한다는 점과 Multilevel Queue 등과 같은 특수한 알고리즘들을 구현하지 않았다는 점에서 한계가 있다. 구조적인 수정을 통해 Deadline이라는 조건을 추가하고, Running State를 Queue형태로 만들어 멀티 프로세서 환경을 표현하고, Ready Queue의 개수를 늘려서 Multilevel Queue 알고리즘

을 구현하는 등의 추가 작업을 통해 한계를 극복할 수 있을 것이라고 본다.

그리고 I/O 작업에 대하여 임의적인 정의를 했기 때문에 실제 환경과 차이가 있다는 한계점도 존재한다. 실제 환경에서 프로세스가 수행될 때 I/O 작업은 보다 임의적이고 산발적으로 발생한다. 그런데, I/O 작업을 실제 환경과 같이 발생시킨다면, 다양한 Scheduling 간에 비교분석을 하는 것이 힘들어진다고 판단하여 한 프로세스가 수행되는 동안 I/O 작업이 한 번만 발생하도록 제한하였다. 그래도 I/O-bound process가 상대적으로 짧은 CPU Burst를 발생시킨다는 점을 반영하기 위하여 프로세스가 처음으로 수행이 되자마자 1의 시간을 Running State로 보내고 바로 I/O 작업을 진행하도록 하였다.

Appendix – Source Code

```
#include <stdio.h>
#include <stdlib.h>
#define MAX_TIME_UNIT 1000
#define MAX_PROCESS_NUM 30
#define MAX_ALGORITHM_NUM 10

#define FCFS 0
#define SJF 1
#define PRIORITY 2
#define RR 3
#define LIF 4
#define LISC 5

#define TRUE 1
#define FALSE 0

#define TIME_QUANTUM 3

//process
typedef struct myProcess* processPointer;
typedef struct myProcess {
	int pid;
	int priority;
	int arrivalTime;
	int CPUburst;
	int IOburst;
	int CPUremainingTime;
	int IOremainingTime;
	int waitingTime;
	int turnaroundTime;
	int responseTime;

}myProcess;

int Computation_start = 0;
int Computation_end = 0;
int Computation_idle = 0;

typedef struct evaluation* evalPointer;
typedef struct evaluation {
  int alg;
  int preemptive;
  int startTime;
  int endTime;
```

```
    int avg_waitingTime;
    int avg_turnaroundTime;
    int avg_responseTime;
    double CPU_util;
    int completed;
}evaluation;

evalPointer evals[MAX_ALGORITHM_NUM];
int cur_eval_num = 0;

void init_evals(){
    cur_eval_num = 0;
    int i;
    for(i=0;i<MAX_ALGORITHM_NUM;i++)
        evals[i]=NULL;
}

void clear_evals() {

    int i;
    for(i=0;i<MAX_ALGORITHM_NUM;i++){
        free(evals[i]);
        evals[i]=NULL;
    }
    cur_eval_num = 0;
}

//Job Queue
processPointer jobQueue[MAX_PROCESS_NUM];
int cur_proc_num_JQ = 0;

void init_JQ () {
    cur_proc_num_JQ = 0;
        int i;
        for (i = 0; i < MAX_PROCESS_NUM; i++)
            jobQueue[i] = NULL;
}

void sort_JQ() { //유저가 pid를 무작위의 순서로 넣는 것을 대비해서 pid를 기준으로 오름차순으로 정렬해준다.
    //같은 시간에 도착한 프로세스들이 pid순서로 정렬되는 효과가 있다.

    //위키피디아의 insertion sort사용 -> 성능 고려 안함
        int i, j;
        processPointer remember;
        for ( i = 1; i < cur_proc_num_JQ; i++ )
        {
            remember = jobQueue[(j=i)];
            while ( --j >= 0 && remember->pid < jobQueue[j]->pid )
                jobQueue[j+1] = jobQueue[j];
            jobQueue[j+1] = remember;
        }
}

int getProcByPid_JQ (int givenPid) { //readyQueue에서 해당 pid를 가지고 있는 process의 index를 리턴한다.
        int result = -1;
        int i;
        for(i = 0; i < cur_proc_num_JQ; i++) {
            int temp = jobQueue[i]->pid;
            if(temp == givenPid)
                return i;
        }
        return result;
}

void insertInto_JQ (processPointer proc) {
        if(cur_proc_num_JQ<MAX_PROCESS_NUM) {
            int temp = getProcByPid_JQ(proc->pid);
```

```
        if (temp != -1) {
            printf("<ERROR> The process with pid: %d already exists in Job Queue₩n", proc->pid);
            return;
        }
        jobQueue[cur_proc_num_JQ++] = proc;
    }
    else {
        puts("<ERROR> Job Queue is full");
        return;
    }
}

processPointer removeFrom_JQ (processPointer proc) { //process 하나를 readyQueue에서 제거하고 빈 공간을 수축을 통해 없앤다.
    if(cur_proc_num_JQ>0) {
        int temp = getProcByPid_JQ(proc->pid);
        if (temp == -1) {
            printf("<ERROR> Cannot find the process with pid: %d₩n", proc->pid);
            return NULL;
        } else {
            processPointer removed = jobQueue[temp];

            int i;
            for(i = temp; i < cur_proc_num_JQ - 1; i++)
                jobQueue[i] = jobQueue[i+1];
            jobQueue[cur_proc_num_JQ - 1] = NULL;

            cur_proc_num_JQ--;
            return removed;
        }

    } else {
        puts("<ERROR> Job Queue is empty");
        return NULL;
    }
}

void clear_JQ() { //메모리 회수용 함수
    int i;
    for(i = 0; i < MAX_PROCESS_NUM; i++) {
        free(jobQueue[i]);
        jobQueue[i] = NULL;
    }
    cur_proc_num_JQ = 0;
}

void print_JQ() { //debug를 위한 print 함수
    //puts("₩nprint_JQ()");
  printf("총 프로세스 수: %d₩n", cur_proc_num_JQ);
  int i;
  puts("pid    priority    arrival_time    CPU burst    IO burst");
  puts("=============================================================");
    for(i = 0; i < cur_proc_num_JQ; i++) {
        printf("%3d    %8d    %12d    %9d    %8d₩n", jobQueue[i]->pid, jobQueue[i]->priority, jobQueue[i]->arrivalTime, jobQueue[i]->CPUburst,
jobQueue[i]->IOburst);
    }
    puts("=============================================================₩n");
}

processPointer cloneJobQueue[MAX_PROCESS_NUM];
int cur_proc_num_clone_JQ = 0;

void clone_JQ() {
  // 여러 시뮬레이션을 처리하기 위해 clone을 만들어준다.

  int i;
  for (i=0; i< MAX_PROCESS_NUM; i++) { //init clone
    cloneJobQueue[i] = NULL;
```

```
    }

    for (i=0; i<cur_proc_num_JQ; i++) {

        processPointer newProcess = (processPointer)malloc(sizeof(struct myProcess));
            newProcess->pid = jobQueue[i]->pid;
            newProcess->priority = jobQueue[i]->priority;
            newProcess->arrivalTime = jobQueue[i]->arrivalTime;
            newProcess->CPUburst = jobQueue[i]->CPUburst;
            newProcess->IOburst = jobQueue[i]->IOburst;
            newProcess->CPUremainingTime = jobQueue[i]->CPUremainingTime;
            newProcess->IOremainingTime = jobQueue[i]->IOremainingTime;
            newProcess->waitingTime = jobQueue[i]->waitingTime;
            newProcess->turnaroundTime = jobQueue[i]->turnaroundTime;
            newProcess->responseTime = jobQueue[i]->responseTime;

            cloneJobQueue[i] = newProcess;
    }

    cur_proc_num_clone_JQ = cur_proc_num_JQ;
}

void loadClone_JQ() {
    // 클론으로부터 JQ에 복사한다.
    clear_JQ(); //clear JQ
    int i;
    for (i=0; i<cur_proc_num_clone_JQ; i++) {

        processPointer newProcess = (processPointer)malloc(sizeof(struct myProcess));
            newProcess->pid = cloneJobQueue[i]->pid;
            newProcess->priority = cloneJobQueue[i]->priority;
            newProcess->arrivalTime = cloneJobQueue[i]->arrivalTime;
            newProcess->CPUburst = cloneJobQueue[i]->CPUburst;
            newProcess->IOburst = cloneJobQueue[i]->IOburst;
            newProcess->CPUremainingTime = cloneJobQueue[i]->CPUremainingTime;
            newProcess->IOremainingTime = cloneJobQueue[i]->IOremainingTime;
            newProcess->waitingTime = cloneJobQueue[i]->waitingTime;
            newProcess->turnaroundTime = cloneJobQueue[i]->turnaroundTime;
            newProcess->responseTime = cloneJobQueue[i]->responseTime;

            jobQueue[i] = newProcess;
    }

    cur_proc_num_JQ = cur_proc_num_clone_JQ;
    //print_JQ();
}

void clearClone_JQ() { //메모리 회수용 함수
        int i;
        for(i = 0; i < MAX_PROCESS_NUM; i++) {
            free(cloneJobQueue[i]);
            cloneJobQueue[i] = NULL;
        }
}

//running state 현재 funning 중인 process
processPointer runningProcess = NULL;
int timeConsumed = 0;

//readyQueue
//arrivalTime이 순서대로 정렬된 채로 process가 create된다고 가정
processPointer readyQueue[MAX_PROCESS_NUM];
int cur_proc_num_RQ = 0; // 현재 process의 수

void init_RQ () {
        cur_proc_num_RQ = 0;
    int i;
```

```
    for (i = 0; i < MAX_PROCESS_NUM; i++)
        readyQueue[i] = NULL;
}

int getProcByPid_RQ (int givenPid) { //readyQueue에서 해당 pid를 가지고 있는 process의 index를 리턴한다.
    int result = -1;
    int i;
    for(i = 0; i < cur_proc_num_RQ; i++) {
        int temp = readyQueue[i]->pid;
        if(temp == givenPid)
            return i;
    }
    return result;
}

void insertInto_RQ (processPointer proc) {
    if(cur_proc_num_RQ<MAX_PROCESS_NUM) {
        int temp = getProcByPid_RQ(proc->pid);
        if (temp != -1) {
            printf("<ERROR> The process with pid: %d already exists in Ready Queue₩n", proc->pid);
            return;
        }
        readyQueue[cur_proc_num_RQ++] = proc;
    }
    else {
        puts("<ERROR> Ready Queue is full");
        return;
    }
}

processPointer removeFrom_RQ (processPointer proc) { //process 하나를 readyQueue에서 제거하고 빈 공간을 수축을 통해 없앤다.
    if(cur_proc_num_RQ>0) {
        int temp = getProcByPid_RQ(proc->pid);
        if (temp == -1) {
            printf("<ERROR> Cannot find the process with pid: %d₩n", proc->pid);
            return NULL;
        } else {
            processPointer removed = readyQueue[temp];

            int i;
            for(i = temp; i < cur_proc_num_RQ - 1; i++)
                readyQueue[i] = readyQueue[i+1];
            readyQueue[cur_proc_num_RQ - 1] = NULL;

            cur_proc_num_RQ--;
            return removed;
        }

    } else {
        puts("<ERROR> Ready Queue is empty");
        return NULL;
    }
}

void clear_RQ() { //메모리 회수용 함수
    int i;
    for(i = 0; i < MAX_PROCESS_NUM; i++) {
        free(readyQueue[i]);
        readyQueue[i]=NULL;
    }
    cur_proc_num_RQ = 0;
}

void print_RQ() { //debug를 위한 print 함수
    puts("₩nprintf_RQ()");
  int i;
    for(i = 0; i < cur_proc_num_RQ; i++) {
```

```
            printf("%d ", readyQueue[i]->pid);
        }
        printf("₩n총 프로세스 수: %d₩n", cur_proc_num_RQ);
}

//waitingQueue
processPointer waitingQueue[MAX_PROCESS_NUM];
int cur_proc_num_WQ = 0;

void init_WQ () {
    cur_proc_num_WQ = 0;
        int i;
        for (i = 0; i < MAX_PROCESS_NUM; i++)
            waitingQueue[i] = NULL;
}

int getProcByPid_WQ (int givenPid) { //waitingQueue에서 해당 pid를 가지고 있는 process의 index를 리턴한다.
        int result = -1;
        int i;
        for(i = 0; i < cur_proc_num_WQ; i++) {
            int temp = waitingQueue[i]->pid;
            if(temp == givenPid)
                return i;
        }
        return result;
}

void insertInto_WQ (processPointer proc) {
        if(cur_proc_num_WQ<MAX_PROCESS_NUM) {
            int temp = getProcByPid_WQ(proc->pid);
            if (temp != -1) {
                printf("<ERROR> The process with pid: %d already exists in Waiting Queue₩n", proc->pid);
                return;
            }
            waitingQueue[cur_proc_num_WQ++] = proc;
        }
        else {
            puts("<ERROR> Waiting Queue is full");
            return;
        }
        //print_WQ();
}

processPointer removeFrom_WQ (processPointer proc) { //process 하나를 waitingQueue에서 제거하고 빈 공간을 수축을 통해 없앤다.
        if(cur_proc_num_WQ>0) {
            int temp = getProcByPid_WQ(proc->pid);
            if (temp == -1) {
                printf("<ERROR> Cannot find the process with pid: %d₩n", proc->pid);
                return NULL;
            } else {

                processPointer removed = waitingQueue[temp];
                int i;
                for(i = temp; i < cur_proc_num_WQ - 1; i++)
                    waitingQueue[i] = waitingQueue[i+1];

                waitingQueue[cur_proc_num_WQ - 1] = NULL;

                cur_proc_num_WQ--;

                return removed;
            }

        } else {
            puts("<ERROR> Waiting Queue is empty");
            return NULL;
        }
```

```
}

void clear_WQ() { //메모리 회수용 함수
	int i;
	for(i = 0; i < MAX_PROCESS_NUM; i++) {
		free(waitingQueue[i]);
		waitingQueue[i] = NULL;
	}
	cur_proc_num_WQ = 0;
}

void print_WQ() { //debug를 위한 print 함수
	puts("₩nprintf_WQ()");
   int i;

	for(i = 0; i < cur_proc_num_WQ; i++) {
		printf("%d ", waitingQueue[i]->pid);
	}
	printf("₩n총 프로세스 수: %d₩n", cur_proc_num_WQ);
}

//terminatedQueue
processPointer terminated[MAX_PROCESS_NUM];
int cur_proc_num_T = 0;

void init_T () {
   cur_proc_num_T = 0;
	int i;
	for (i = 0; i < MAX_PROCESS_NUM; i++)
		terminated[i] = NULL;
}

void clear_T() { //메모리 회수용 함수
	int i;
	for(i = 0; i < MAX_PROCESS_NUM; i++) {
		free(terminated[i]);
		terminated[i] = NULL;
	}
	cur_proc_num_T = 0;
}

void insertInto_T (processPointer proc) {
	if(cur_proc_num_T<MAX_PROCESS_NUM) {
		terminated[cur_proc_num_T++] = proc;
	}
	else {
		puts("<ERROR> Cannot terminate the process");
		return;
	}
}

void print_T() { //debug를 위한 print 함수
	puts("₩nprintf_T()");

   int i;
	for(i = 0; i < cur_proc_num_T; i++) {
		printf("%d ", terminated[i]->pid);
	}
	printf("₩n총 프로세스 수: %d₩n", cur_proc_num_T);
}

processPointer createProcess(int pid, int priority, int arrivalTime, int CPUburst, int IOburst) { //프로세스 하나를 만든다.
	//랜덤으로 생성해서 여러 알고리즘 테스트하는 건 clone을 사용하자

	if (arrivalTime > MAX_TIME_UNIT || arrivalTime < 0) {
		printf("<ERROR> arrivalTime should be in [0..MAX_TIME_UNIT]₩n");
		printf("<USAGE> createProcess(int pid, int priority, int arrivalTime, int CPUburst, int IOburst)₩n");
```

```
        return NULL;
    }

    if (CPUburst <= 0 || IOburst < 0) {
        printf("<ERROR> CPUburst and should be larger than 0 and IOburst cannot be a negative number.₩n");
        printf("<USAGE> createProcess(int pid, int priority, int arrivalTime, int CPUburst, int IOburst)₩n");
        return NULL;
    }
        processPointer newProcess = (processPointer)malloc(sizeof(struct myProcess));
        newProcess->pid = pid;
        newProcess->priority = priority;
        newProcess->arrivalTime = arrivalTime;
        newProcess->CPUburst = CPUburst;
        newProcess->IOburst = IOburst;
        newProcess->CPUremainingTime = CPUburst;
        newProcess->IOremainingTime = IOburst;
        newProcess->waitingTime = 0;
        newProcess->turnaroundTime = 0;
        newProcess->responseTime = -1;

        //job queue에 넣는다.
        insertInto_JQ(newProcess);

    //debug
    //printf("%d %d %d %d created₩n",newProcess.pid ,newProcess.priority, newProcess.arrivalTime, newProcess.CPUburst);
    return newProcess;
}

processPointer FCFS_alg() {

        processPointer earliestProc = readyQueue[0]; //가장 먼저 도착한 process를 찾는다.

        if (earliestProc != NULL){

            if(runningProcess != NULL) { //이미 수행중인 프로세스가 있었다면 preemptive가 아니므로 기다린다.
                /*
        if(runningProcess->arrivalTime > earliestProc->arrivalTime)
          puts("<ERROR> Invalid access."); //오류메세지를 출력한다.
                */
        return runningProcess;
          } else {
        return removeFrom_RQ(earliestProc);
       }

        } else { //readyQueue에 아무것도 없는 경우
            return runningProcess;
        }
}

processPointer SJF_alg(int preemptive) {

  processPointer shortestJob = readyQueue[0];

  if(shortestJob != NULL) {
    int i;
        for(i = 0; i < cur_proc_num_RQ; i++) {
            if (readyQueue[i]->CPUremainingTime <= shortestJob->CPUremainingTime) {

                if(readyQueue[i]->CPUremainingTime == shortestJob->CPUremainingTime) { //남은 시간이 같을 경우먼저 도착한 process가 먼저 수행된다.
                    if (readyQueue[i]->arrivalTime < shortestJob->arrivalTime) shortestJob = readyQueue[i];
                } else {
                    shortestJob = readyQueue[i];
                }
            }
        }

    if(runningProcess != NULL) { //이미 수행중인 프로세스가 있을 때
```

```
        if(preemptive){ //preemptive면

          if(runningProcess->CPUremainingTime >= shortestJob->CPUremainingTime) {
            if(runningProcess->CPUremainingTime == shortestJob->CPUremainingTime) { //남은 시간이 같을 경우먼저 도착한 process가 먼저 수행된다.
                        if (runningProcess->arrivalTime < shortestJob->arrivalTime){
              return runningProcess;
            } else if(runningProcess->arrivalTime == shortestJob->arrivalTime)
              return runningProcess; //arrivalTime까지 같으면 굳이 Context switch overhead를 감수하면서까지 preempt하지 않는다.
            }
            puts("preemption is detected.");
            insertInto_RQ(runningProcess);
            return removeFrom_RQ(shortestJob);
          }

          return runningProcess;
        }
              //non-preemptive면 기다린다.
        return runningProcess;
          } else {
        return removeFrom_RQ(shortestJob);
      }

  }else {
    return runningProcess;
  }
}

processPointer PRIORITY_alg(int preemptive) {

  processPointer importantJob = readyQueue[0];

  if(importantJob != NULL) {
    int i;
        for(i = 0; i < cur_proc_num_RQ; i++) {
            if (readyQueue[i]->priority <= importantJob->priority) {

                if(readyQueue[i]->priority == importantJob->priority) { //priority가 같을 경우먼저 도착한 process가 먼저 수행된다.
                    if (readyQueue[i]->arrivalTime < importantJob->arrivalTime)
            importantJob = readyQueue[i];
                } else {
                    importantJob = readyQueue[i];
                }
            }
        }

    if(runningProcess != NULL) { //이미 수행중인 프로세스가 있을 때
        if(preemptive){ //preemptive면

          if(runningProcess->priority >= importantJob->priority) {
            if(runningProcess->priority == importantJob->priority) { //priority가 같을 경우먼저 도착한 process가 먼저 수행된다.
                        if (runningProcess->arrivalTime < importantJob->arrivalTime){
              return runningProcess;
            } else if(runningProcess->arrivalTime == importantJob->arrivalTime) {
              return runningProcess; //arrivalTime까지 같다면 굳이 preempt안한다 (context - switch overhead 줄이기 위해)

            }
            }
            puts("preemption is detected.");
            insertInto_RQ(runningProcess);
            return removeFrom_RQ(importantJob);
          }

          return runningProcess;
        }
              //non-preemptive면 기다린다.
        return runningProcess;
          } else {
```

```
            return removeFrom_RQ(importantJob);
        }

    }else {
        return runningProcess;
    }
}

processPointer RR_alg(int time_quantum){

    processPointer earliestProc = readyQueue[0]; //가장 먼저 도착한 process를 찾는다.

            if (earliestProc != NULL){

                if(runningProcess != NULL) { //이미 수행중인 프로세스가 있었다면
            //return runningProcess;

            if(timeConsumed >= TIME_QUANTUM){ //이미 수행중이 었던 프로세스가 Time expired되었다면
                insertInto_RQ(runningProcess);
                return removeFrom_RQ(earliestProc);
            } else {
                return runningProcess;
            }

                } else {
            return removeFrom_RQ(earliestProc);
        }

            } else { //readyQueue에 아무것도 없는 경우
                return runningProcess;
            }
}

processPointer LIF_alg(int preemptive) {
    //Non-preemptive의 경우 CPUburst Time을 기준으로 봤을 때는 FCFS와 같다.
    processPointer longestJob = readyQueue[0];
    if(longestJob != NULL) {
        int i;
            for(i = 0; i < cur_proc_num_RQ; i++) {
                if (readyQueue[i]->IOremainingTime >= longestJob->IOremainingTime) {

                    if(readyQueue[i]->IOremainingTime == longestJob->IOremainingTime) { //남은 시간이 같을 경우먼저 도착한 process가 먼저 수행된다.
                        if (readyQueue[i]->arrivalTime < longestJob->arrivalTime) longestJob = readyQueue[i];
                    } else {
                        longestJob = readyQueue[i];
                    }
                }
            }

        if(runningProcess != NULL) { //이미 수행중인 프로세스가 있을 때
            if(preemptive){ //preemptive면

              if(runningProcess->IOremainingTime <= longestJob->IOremainingTime) {
                if(runningProcess->IOremainingTime == longestJob->IOremainingTime) { //남은 시간이 같을 경우먼저 도착한 process가 먼저 수행된다.
                            if (runningProcess->arrivalTime < longestJob->arrivalTime){
                    return runningProcess;
                  } else if(runningProcess->arrivalTime == longestJob->arrivalTime) {
                    return runningProcess; //arrivalTime까지 같다면 굳이 preempt안한다 (context - switch overhead 줄이기 위해)
                  }
                }
                puts("preemption is detected.");
                insertInto_RQ(runningProcess);
                return removeFrom_RQ(longestJob);
              }

              return runningProcess;
            }
```

```
                //non-preemptive면 기다린다.
        return runningProcess;
          } else {
        return removeFrom_RQ(longestJob);
      }

  }else {
    return runningProcess;
  }
}

processPointer LISC_alg(int preemptive) { //Longest IO burst, Shortest CPU burst Algorithm
  processPointer longestJob = readyQueue[0]; //search longest IO burst
  if(longestJob != NULL) {
    int i;
      for(i = 0; i < cur_proc_num_RQ; i++) {
        if (readyQueue[i]->IOremainingTime >= longestJob->IOremainingTime) {

          if(readyQueue[i]->IOremainingTime == longestJob->IOremainingTime) { //남은 IO burst 시간이 같을 경우

          if (readyQueue[i]->CPUremainingTime <= longestJob->CPUremainingTime) { //CPU burst time 을 비교한다.

              if(readyQueue[i]->CPUremainingTime == longestJob->CPUremainingTime) { //CPU burst time마저 같을 경우먼저 도착한 process가 먼저
수행된다.
                if (readyQueue[i]->arrivalTime < longestJob->arrivalTime) longestJob = readyQueue[i];
              } else {
                longestJob = readyQueue[i];
              }
            }

            } else {
              longestJob = readyQueue[i];
            }
        }
      }

    if(runningProcess != NULL) { //이미 수행중인 프로세스가 있을 때
      if(preemptive){ //preemptive면

        if(runningProcess->IOremainingTime <= longestJob->IOremainingTime) {
          if(runningProcess->IOremainingTime == longestJob->IOremainingTime) { //남은 시간이 같을 경우

            if(runningProcess->CPUremainingTime <= longestJob->CPUremainingTime) { //CPU 시간도 고려해준다.
              if(runningProcess->CPUremainingTime == longestJob->CPUremainingTime){
                if (runningProcess->arrivalTime < longestJob->arrivalTime){ //먼저 도착한 process가 먼저 수행된다.
                  return runningProcess;
                } else if(runningProcess->arrivalTime == longestJob->arrivalTime) {
                  return runningProcess; //arrivalTime까지 같다면 굳이 preempt안한다 (context - switch overhead 줄이기 위해)
                }
              } else {
                return runningProcess;
              }
            }
          }
          puts("preemption is detected.");
          insertInto_RQ(runningProcess);
          return removeFrom_RQ(longestJob);
        }

        return runningProcess;
      }
            //non-preemptive면 기다린다.
      return runningProcess;
        } else {
      return removeFrom_RQ(longestJob);
    }
```

```
    }else {
        return runningProcess;
    }
}

processPointer schedule(int alg, int preemptive, int time_quantum) { //timelimit 시간동안 scheduling 알고리즘을 진행한다.
    processPointer selectedProcess = NULL;

    switch(alg) {
        case FCFS:
            selectedProcess = FCFS_alg();
            break;
        case SJF:
            selectedProcess = SJF_alg(preemptive);
            break;
        case RR:
            selectedProcess = RR_alg(time_quantum);
            break;
        case PRIORITY:
            selectedProcess = PRIORITY_alg(preemptive);
            break;
        case LIF:
            selectedProcess = LIF_alg(preemptive);
            break;
        case LISC:
            selectedProcess = LISC_alg(preemptive);
            break;
        default:
        return NULL;
    }

    return selectedProcess;
}

void simulate(int amount, int alg, int preemptive, int time_quantum) { //amount 시점이 흐른 뒤의 상태 -> 반복문에 넣어서 사용
    //우선, Job queue에서 해당 시간에 도착한 프로세스들을 ready queue에 올려준다.
    processPointer tempProcess = NULL;
    int jobNum = cur_proc_num_JQ;
    int i;
    for(i = 0; i < cur_proc_num_JQ; i++) {
        if(jobQueue[i]->arrivalTime == amount) {
            tempProcess = removeFrom_JQ(jobQueue[i--]);
            insertInto_RQ(tempProcess);
        }
    }
    processPointer prevProcess = runningProcess;
    runningProcess = schedule(alg, preemptive, time_quantum); //이번 turn에 수행될 process를 pick up한다.

    printf("%d: ",amount);
    if(prevProcess != runningProcess) { //이전과 다른 프로세스가 running 상태로 되었을 경우
        //printf("다른 프로세스로 바뀌었고, 방금 전 프로세스가 runnig 된 시간은 %d야.₩n",timeConsumed);
        timeConsumed = 0; //running에 소요된 시간을 초기화시켜준다.

        if(runningProcess->responseTime == -1) { //responseTime을 기록해둔다.
            runningProcess->responseTime = amount - runningProcess->arrivalTime;
        }
    }

        for(i = 0; i < cur_proc_num_RQ; i++) { //readyQueue에 있는 process들을 기다리게 한다.

            if(readyQueue[i]) {
                readyQueue[i]->waitingTime++;
                readyQueue[i]->turnaroundTime++;
            }
        }

        for(i = 0; i < cur_proc_num_WQ; i++) { //waitingQueue에 있는 process들이 IO 작업을 수행한다.
```

```
    if(waitingQueue[i]) {
      //waitingQueue[i]->waitingTime++;
      waitingQueue[i]->turnaroundTime++;
      waitingQueue[i]->IOremainingTime--;

      if(waitingQueue[i]->IOremainingTime <= 0 ) { //IO 작업이 완료된 경우
        printf("(pid: %d) -> IO complete, ", waitingQueue[i]->pid);
        insertInto_RQ(removeFrom_WQ(waitingQueue[i--])); //ready queue로 프로세스를 다시 돌려보내준다.
        //print_WQ();
      }
    }
  }

    if(runningProcess != NULL) { //running 중인 프로세스가 있다면 실행시킴
        runningProcess->CPUremainingTime --;
        runningProcess->turnaroundTime ++;
        timeConsumed ++;
        printf("(pid: %d) -> running ",runningProcess->pid);

        if(runningProcess->CPUremainingTime <= 0) { //모두 수행이 된 상태라면, terminated로 보내준다.
      insertInto_T(runningProcess);
      runningProcess = NULL;
      printf("-> terminated");
    } else { //아직 수행할 시간이 남아있을 경우
      if(runningProcess->IOremainingTime > 0) { //IO 작업을 수행해야 한다면, waiting queue로 보내준다.
        insertInto_WQ(runningProcess);
        runningProcess = NULL;
        printf("-> IO request");
      }
    }

        printf("₩n");
    } else { //running 중인 프로세스가 없다면 idle을 출력함
      printf("idle₩n");
      Computation_idle++;
  }
}
void analyize(int alg, int preemptive) {

  int wait_sum = 0;
  int turnaround_sum = 0;
  int response_sum = 0;
  int i;
  processPointer p=NULL;
  puts  ("============================================================");
  for(i=0;i<cur_proc_num_T;i++){
    p = terminated[i];
    printf("(pid: %d)₩n",p->pid);
    printf("waiting time = %d, ",p->waitingTime);
    printf("turnaround time = %d, ",p->turnaroundTime);
    //printf("CPU remaining time = %d₩n",p->CPUremainingTime);
    //printf("IO remaining time = %d₩n",p->IOremainingTime);
    printf("response time = %d₩n",p->responseTime);

    puts  ("============================================================");
    wait_sum += p->waitingTime;
    turnaround_sum += p->turnaroundTime;
    response_sum += p->responseTime;
  }
  printf("start time: %d / end time: %d / CPU utilization : %.2lf%% ₩n",Computation_start, Computation_end,
   (double)(Computation_end - Computation_idle)/(Computation_end - Computation_start)*100);

  if(cur_proc_num_T != 0) {
    printf("Average waiting time: %d₩n",wait_sum/cur_proc_num_T);
    printf("Average turnaround time: %d₩n",turnaround_sum/cur_proc_num_T);
    printf("Average response time: %d₩n",response_sum/cur_proc_num_T);
  }
```

```
        printf("Completed: %d₩n",cur_proc_num_T);

    if(cur_proc_num_T != 0) {
        evalPointer newEval = (evalPointer)malloc(sizeof(struct evaluation));
        newEval->alg = alg;
        newEval->preemptive = preemptive;

        newEval->startTime = Computation_start;
        newEval->endTime = Computation_end;
        newEval->avg_waitingTime = wait_sum/cur_proc_num_T;
        newEval->avg_turnaroundTime = turnaround_sum/cur_proc_num_T;
        newEval->avg_responseTime = response_sum/cur_proc_num_T;
        newEval->CPU_util = (double)(Computation_end - Computation_idle)/(Computation_end - Computation_start)*100;
        newEval->completed = cur_proc_num_T;
        evals[cur_eval_num++] = newEval;
    }
    puts   ("============================================================");

}
void startSimulation(int alg, int preemptive, int time_quantum, int count) {
    loadClone_JQ();

    switch(alg) {
            case FCFS:
                puts("<FCFS Algorithm>");
                break;
            case SJF:
              if(preemptive) printf("<Preemptive ");
              else printf("<Non-preemptive ");
              puts("SJF Algorithm>");
              break;
            case RR:
              printf("<Round Robin Algorithm (time quantum: %d)>₩n",time_quantum);
              break;
            case PRIORITY:
              if(preemptive) printf("<Preemptive ");
              else printf("<Non-preemptive ");
              puts("Priority Algorithm>");
              break;
            case LIF:
              if(preemptive) printf("<Preemptive ");
              else printf("<Non-preemptive ");
              puts("LIF Algorithm>");
              break;
            case LISC:
              if(preemptive) printf("<Preemptive ");
              else printf("<Non-preemptive ");
              puts("LISC Algorithm>");
              break;
            default:
            return;
      }

    int initial_proc_num = cur_proc_num_JQ; //실제 시뮬레이션을 하기 전 프로세스의 수를 저장해둔다.

    int i;
    if(cur_proc_num_JQ <= 0) {
      puts("<ERROR> Simulation failed. Process doesn't exist in the job queue");
      return;
    }

    int minArriv = jobQueue[0]->arrivalTime;
    for(i=0;i<cur_proc_num_JQ;i++) {
      if(minArriv > jobQueue[i]->arrivalTime)
          minArriv = jobQueue[i]->arrivalTime;
    }
    Computation_start = minArriv;
```

```
    Computation_idle = 0;
    for(i=0;i<count;i++) {
        simulate(i,alg, preemptive, TIME_QUANTUM);
        if(cur_proc_num_T == initial_proc_num) {
            i++;
            break;
        }
    }
    Computation_end = i-1;

    analyize(alg, preemptive);
    clear_JQ();
        clear_RQ();
        clear_T();
        clear_WQ();
        free(runningProcess);
        runningProcess = NULL;
        timeConsumed = 0;
        Computation_start = 0;
    Computation_end = 0;
    Computation_idle = 0;
}

void evaluate() {

    puts ("₩n                    <Evaluation>                    ₩n");
    int i;
    for(i=0;i<cur_eval_num;i++) {

        puts ("============================================================");

        int alg = evals[i]->alg;
        int preemptive = evals[i]->preemptive;

        switch (evals[i]->alg) {

        case FCFS:
                puts("<FCFS Algorithm>");
                break;
            case SJF:
                if(preemptive) printf("<Preemptive ");
                else printf("<Non-preemptive ");
                puts("SJF Algorithm>");
                break;
            case RR:
                puts("<Round Robin Algorithm>");
                break;
            case PRIORITY:
                if(preemptive) printf("<Preemptive ");
                else printf("<Non-preemptive ");
                puts("Priority Algorithm>");
                break;
            case LIF:
                if(preemptive) printf("<Preemptive ");
                else printf("<Non-preemptive ");
                puts("LIF Algorithm>");
                break;
            case LISC:
                if(preemptive) printf("<Preemptive ");
                else printf("<Non-preemptive ");
                puts("LISC Algorithm>");
                break;
            default:
            return;
        }
        puts ("------------------------------------------------------------");
        printf("start time: %d / end time: %d / CPU utilization : %.2lf%% ₩n",evals[i]->startTime,evals[i]->endTime,evals[i]->CPU_util);
```

```
        printf("Average waiting time: %d₩n",evals[i]->avg_waitingTime);
        printf("Average turnaround time: %d₩n",evals[i]->avg_turnaroundTime);
        printf("Average response time: %d₩n",evals[i]->avg_responseTime);
        printf("Completed: %d₩n",evals[i]->completed);
    }

    puts   ("============================================================");
}


void createProcesses(int total_num, int io_num) {
    if(io_num > total_num) {
        puts("<ERROR> The number of IO event cannot be higher than the number of processes");
        return;
    }

    srand(time(NULL));

    int i;
    for(i=0;i<total_num; i++) {
        //CPU burst : 5~20
        //IO burst : 1~10
        createProcess(i+1, rand() % total_num + 1, rand() % (total_num + 10), rand() % 16 + 5, 0);
    }

    for(i=0;i<io_num;i++) {

        int randomIndex = rand() % total_num ;
        if(jobQueue[randomIndex]->IOburst ==0) {

        int randomIOburst = rand() % 10 + 1;
        jobQueue[randomIndex]->IOburst = randomIOburst;
        jobQueue[randomIndex]->IOremainingTime = randomIOburst;

        } else {
            i--;
        }

    }
    sort_JQ();
    clone_JQ(); //backup this JQ
    print_JQ();
}

void main(int argc, char **argv) {
    init_RQ();
    init_JQ();
    init_T();
    init_WQ();
    init_evals();

    int totalProcessNum = atoi(argv[1]);
    int totalIOProcessNum = atoi(argv[2]);
    createProcesses(totalProcessNum,totalIOProcessNum);
    int i;
    int amount = 120;
     startSimulation(FCFS,FALSE,TIME_QUANTUM, amount);

    startSimulation(SJF,FALSE,TIME_QUANTUM, amount);
    startSimulation(SJF,TRUE,TIME_QUANTUM, amount);
    startSimulation(PRIORITY,FALSE,TIME_QUANTUM, amount);
    startSimulation(PRIORITY,TRUE,TIME_QUANTUM, amount);
    startSimulation(RR,TRUE,TIME_QUANTUM, amount);
    startSimulation(LIF,FALSE, TIME_QUANTUM, amount);
    startSimulation(LIF,TRUE, TIME_QUANTUM, amount);
    startSimulation(LISC,FALSE, TIME_QUANTUM, amount);
    startSimulation(LISC,TRUE, TIME_QUANTUM, amount);
```

```
    evaluate();

    clear_JQ();
    clear_RQ();
    clear_T();
    clear_WQ();
    clearClone_JQ();

    clear_evals();
}
```