운영체제(Operating Systems)

Term Project Report

CPU Scheduling Simulator 구현

2014210042 컴퓨터학과

한 윤

Table of Contents

1. 프로젝트 개요

1.1 프로젝트 소개 및 진행 목적

이 프로젝트는 수업시간에 배운 CPU Scheduling Algorithm 을 이용하여 CPU Scheduling Simulator 를 구현하는 프로젝트이다. 사용자 인터페이스(UI)는 별다른 GUI 라이브러리를 쓰지 않고, Application Console 에서 최대한 정돈된 출력을 통해 simulating 결과를 보여주도록 노력하였다.

CPU Scheduling algorithm 을 직접 C코드로 작성해보면서 개념적으로만 알고있던 각 스케쥴링 알고리즘에 대해 보다 깊은 이해를 하는 것이 목적이다.

2. 프로젝트 본론

2.1 Scheduling Algorithm 소개

① FCFS (First Come First Served)

FCFS 알고리즘이란 그 이름에서 쉽게 알 수 있듯이, CPU에게 수행을 요청한 순서대로 프로세스에 우선권을 부여하는 스케줄링 방식이다. 이 방식은 non-preemptive 방식이기 때문에 한 프로세스가 CPU를 일단 점유하게 되면 해당 프로세스가 종료를 당하거나 interrupt 되기 전까지는 다른 프로세스가 CPU를 점유할 수 없다.

장점 : 다른 알고리즘에 비해 상대적으로 단순하여 구현이 쉽다. 그리고 수행을 요청한 순서대로 프로세스들이 처리되기 때문에 어느 한 프로세스가 지속적으로 밀리는 현상인 starvation 이 발생하지 않는다.

단점 : 오로지 요청한 순서만 고려하기 때문에 convoy effect가 발생할 수 있다. convoy effect 란 처리 시간(CPU Burst Time)이 긴 프로세스 뒤에서 처리 시간이 짧은 프로세스들이 오랫동안 기다리게 되는 현상을 말한다.

② SJF (Shortest Job First)

SJF 알고리즘이란 remaining CPU burst time 이 가장 작은 프로세스에게 우선권을 주는 방식이다. 이 알고리즘은 Preemptive 버전과 Non-preemptive 버전 두 가지로 구현할 수 있다. Preemptive 방식에서는 Non-preemptive 방식과는 다르게, 이미 CPU를 점유하고 있는 프로세스가 있어도 현재 프로세스보다 남은 CPU Burst time이 작은 프로세스가 들어오면 그 작은 프로세스가 CPU 수행 권한을 받게 된다.

장점 : CPU burst time 이 가장 작은 프로세스를 매번 고려하기 때문에 결과적으로 Average Waiting Time 이나 Average Turnaround Time 의 관점에서 효율적이다. Throughput 관점에서는 가장 이상적인 알고리즘이다.

단점 : 실제 CPU 상황에서는 각 프로세스별 burst time 을 정확히 예측하는 것이 사실상 불가능하므로 현실적이지는 못한 알고리즘이라고 할 수 있다.

③ Priority

Priority 알고리즘이란 단어 뜻 그대로 각 프로세스에 대한 우선 순위 값을 설정하고 그 값에 알맞는 우선 순위대로 프로세스를 처리하는 알고리즘이다. 우선 순위는 정수 값으로 나타내며, 크기가 큰 순서대로 우선순위를 부여할지 작은 순서대로 부여할지는 운영체제마다 다르게 적용된다. SJF와 마찬가지로 Preemptive 방식과 Non-preemptive 방식이 있다.

장점 : 프로세스들의 수행 순서가 우선순위에 따라 결정되므로 중요도가 높은 프로세스를 우선적으로 처리할 수 있다.

단점 : 어떤 프로세스의 우선 순위가 상대적으로 매우 낮아서 더 높은 우선 순위를 가진 프로세스들에게 계속 밀리는 현상인 starvation 이 발생한다. 하지만 이 starvation 현상은 aging 이라는 기법으로 해결이 가능하다. 이 기법은 우선 순위가 낮아서 계속 waiting queue 에 있는 프로세스를 대상으로 시간이 지남에 따라 우선 순위를 점차 높여주는 기법이다.

④ Round Robin

Round Robin 방식은 쉽게 말하면 프로세스들 간에 계속 순서를 돌아가면서 CPU를 사용하는 방식이다. 이 때 각각의 프로세스들이 한 번 CPU를 점유했을 때 사

용할 수 있는 시간을 time quantum 이라고 하며, 이 time quantum 동안 CPU를 사용한 뒤에는 다음 프로세스에게 사용 권한을 넘겨주어야 한다. 프로세스가 처리되는 순서는 FCFS 방식과 같이 CPU에 사용 요청을 한 순서를 따른다. 만약 할당된 time quantum 시간 내에 어떤 프로세스의 CPU burst time 을 모두 소진했다면 해당 프로세스는 그대로 terminate 된다.

Time quantum이 무한히 크다면 결과적으로는 FCFS와 같은 동작 방식을 보이며, Time quantum이 작다면 여러 프로세스를 동시에 처리하는 듯한 효과를 볼 수 있다. 하지만 이 경우에는 여러 프로세스간 전환하는데에 필요한 시간인 context switch가 매우 자주 발생하므로 이에 따른 overhead 가 발생한다.

장점 : 각 프로세스가 대기열에서 대기하는 시간이 무한하지 않고, response time이 상대적으로 빠르다. 각 프로세스의 순서가 빠르게 회전하기 때문에 starvation 문제가 발생하지 않는다.

단점 : Context switch 로 인한 overhead 가 발생하여 처리 효율(efficiency)이 낮아질 수 있다.

2.2 기존 시뮬레이터 소개

기존에 누군가가 만들어 놓은 Simulator가 있나 조사를 해보았다. 구글링 해보니 많은 사람들이 CPU scheduling simulator 를 구현해 놓았는데, 그 중에서 하나만 골라 간략히 소개해 보도록 한다.

출처 : https://calmtot.blogspot.com/2016/04/os-simulator.html

이 블로그에 소개된 simulator 는 총 4가지의 스케줄링 알고리즘을 구현했는데, 그 4가지는 각각 FCFS, SJF, HRN, RoundRobin 이다. FCFS, SJF 그리고 RoundRobin 까지는 나의 이번 프로젝트와 같지만 HRN은 처음 듣는 생소한 알고리즘이었다. 그래서 간략히 알아보자면,

※ HRN 알고리즘

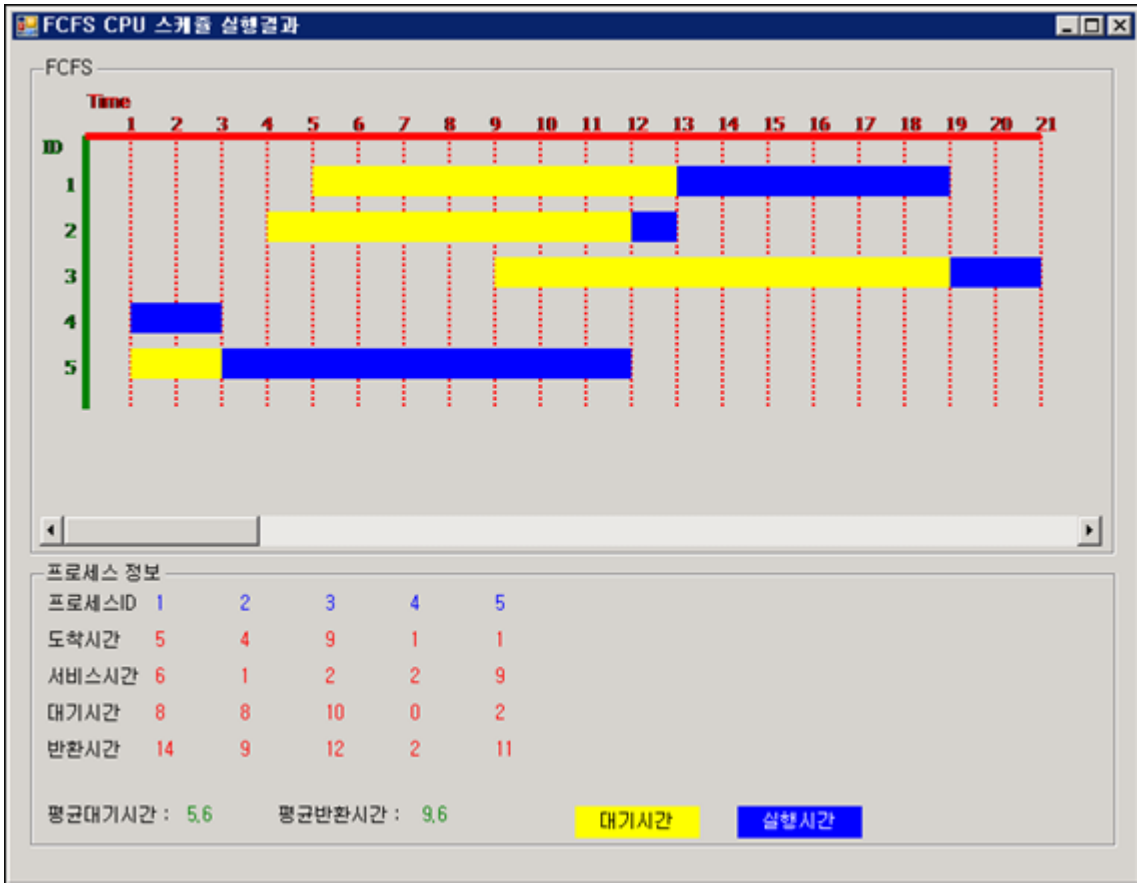
HRN(Highest Response ratio Next)알고리즘은 SJF의 단점을 보완한 알고리즘으로, 처리 시간과 대기 시간 모두를 고려하여 각 프로세스의 우선순위를 정한다. 처리 시간이 긴 프로세스와 짧은 프로세스 간의 지나친 편향을 해결하기 위해 처리 시간과 대기 시간을 고려한 다음과 같은 공식을 따른다.

(Waiting time + Burst time) / Burst time

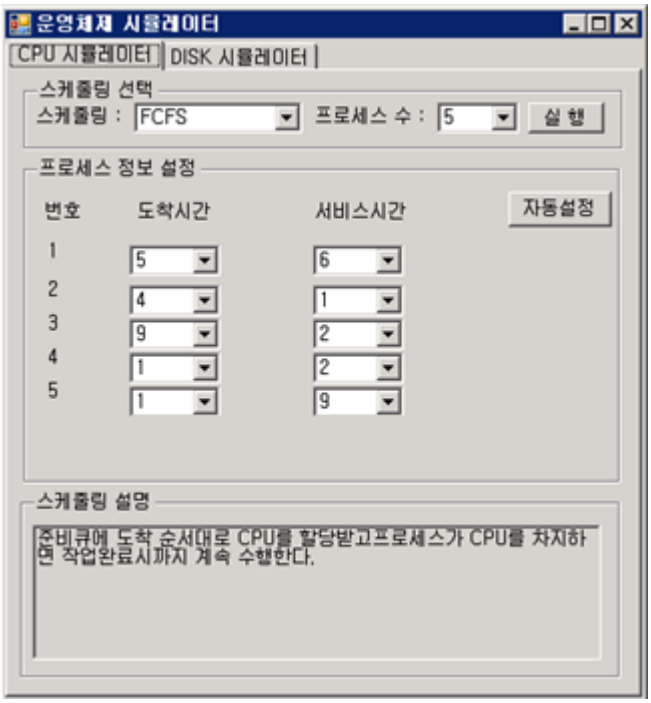
preemptive 방식과 non-preemptive 방식 모두로 구현할 수 있지만, 일반적으로는 non-preemptive 방식으로 구현한다.

이 정도로 HRN 알고리즘을 요약할 수 있겠다.

이 simulator는 GUI 라이브러리를 사용했는데, 각 알고리즘들의 Gantt chart, 프로세스 정보, ATT, AWT 등의 정보를 시각적으로 출력하고 있다.



또한, simulating 전에 여러 설정 값을 받는 부분은 아래 그림과 같이 구성하고 있다.



2.3 시뮬레이터 구현 방식 소개

본 프로젝트의 시뮬레이터는 콘솔 사용자로부터 프로세스의 개수를 직접 입력받아서(IO 프로세스 또한 입력받는다) 프로세스를 생성하도록 했다. 프로세스를 생성할 수 있는 숫자는 상한을 두었으며(15개), 프로세스마다 각 프로세스에 필요한 정보들을 담을 수 있게 구조체로 선언해 주었다.

* N : 프로세스 개수

PID = 1 ~ N 까지의 순차적인 값

Priority = 1 ~ N 까지의 임의적인 값

Arrival time = 0 ~ (N + 4) 까지의 임의적인 값

CPU burst time = 3 ~ 10 범위의 임의적인 값

I/O burst time = 1 ~ 5 범위의 임의적인 값

Turnaround Time

Waiting Time

Remaining Time(CPU, I/O)

이렇게 총 N개의 프로세스가 생성되고 난 후에는 I/O 작업이 수행될 프로세스를 선정한다. 콘솔의 인자로 입력받은 개수 만큼 임의로 선정하고, 선정된 프로세스는 각자의 처리 시간에 25%확률로 I/O interreupt를 발생시켜 I/O작업을 수행하게 된다.

Turnaround time 이나 Waiting Time 은 프로세스가 생성될 때 0으로 초기화 되며 시뮬레이션이 진행됨에 따라 그 값이 계속 누적되는 방식이다.

I/O 작업의 발생 과정에 대한 부가 설명

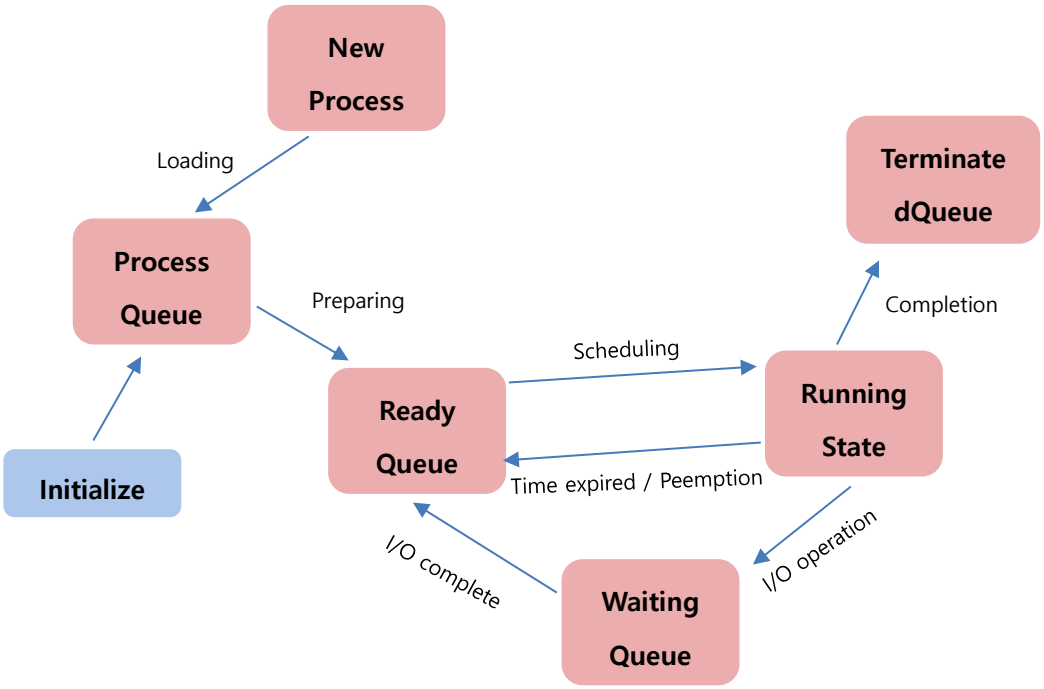
프로세스 생성과정에서 I/O 작업을 하도록 선정된 프로세스들은 1에서 5사이의 I/O burst time 을 가지게 된다. 이 I/O 프로세스들은 CPU를 사용하는 시간 중에 매 time unit 마다 25%의 확률로 I/O 작업을 수행하고 waiting queue 로 들어간다. 한 번 I/O interrupt 가 발생하게 되면 해당 I/O 프로세스는 interrupt 가 발생한 시점에 waiting queue 로 들어가서 I/O burst time 을 모두 소진한 후에 다시 ready queue 로 돌아온다. 따라서 I/O 프로세스는 자신의 turnaround time 까지 단 한번의 I/O 작업만 수행하게 된다.

Time quantum 값과 Priority 값에 대한 설정

Priority 값이 낮을수록 우선순위가 높은 것이라고 가정하였다. 이 priority 값은 CPU schdule 에서 할당해주는것이 아니라 외부에서 받아온 값을 사용한다고 가정했으므로 임의적으로 부여하였다.

Time quantum 값은 프로그램 시작 단계에서 인자로 받도록 했다. Round Robin 알고리즘 수행 시에 여러 time quantum 값에 따른 변화를 보려면 이 인자 값을 바꾸어주면 된다.

시뮬레이터 구성도



① New Process

시뮬레이터가 시작되어서 새로운 프로세스들을 생성하는 단계이다. 이 단계에서는 새로운 프로세스 생성뿐만 아니라 시뮬레이팅이 필요한 여러 queue 들(waiting queue, ready queue, terminated queue)을 초기화하는 작업 또한 이루어진다.

② Process Queue

생성할 프로세스의 수를 입력받고나서 해당 개수만큼 프로세스를 생성하면 만들어질 프로세스를 보관할 공간이 필요하다. 그게 바로 Process Queue 이다. 이 Process Queue 에는 각 프로세스들이 arrival time 을 기준으로 정렬되어있는데 그 이유는 매 time unit 마다 시뮬레이팅을 할 때, 프로세스가 arrival time 기준으로 정렬이 되어있으면 현재 time 에 해당하는 프로세스를 ready queue로 가져오기가 편리하기 때문이다.

③ Initialize

Initialize 부분은 각 알고리즘을 수행한 뒤에 다음 알고리즘을 수행하기 위해 관련된 모든 정보들(waiting time, turnaround time, remaining time ...)을 초기화

해주기 위해 존재한다. 이렇게 알고리즘들 사이에 초기화를 해주지 않으면 이전 단계의 알고리즘 analyzing 결과가 영향을 미쳐 올바른 동작을 보여주지 못하게 된다.

④ **Ready Queue**

현재의 time 과 어떤 프로세스의 arrival time이 같을 때 그 프로세스는 수행할 준비가 된 것이다. 이러한 프로세스들은 Process queue 에서 ready queue로 옮겨지면서 시뮬레이팅될 준비를 한다.

⑤ **Waiting Queue**

Running state 에 있던 프로세스에 I/O interrupt 가 발생했을 때, 해당 프로세스는 running 상태에서 waiting queue로 넘어가게 된다. Waiting queue 로 넘어간 프로세스는 그 프로세스가 가진 I/O burst time 을 모두 소진할 때까지 waiting queue 에 머물며, I/O burst time 을 모두 소진하여 I/O 작업이 끝나면 다시 ready queue 로 돌아오게 된다.

⑥ **Running state**

매 시뮬레이팅 시간마다 해당 time unit 에 맞는 프로세스를 ready queue 에서 가져온다. 가져온 프로세스는 running state 를 가지며, 이 running state 에서 머무른 시간만큼 CPU remaining time 이 줄어든다. 본 시뮬레이터는 싱글 프로세서를 기반으로 하였으므로 running state 가 단일한 값을 가진다. Preemption 이 발생하게 되면 running 되던 프로세스는 ready queue 로 돌아가게 된다.

⑦ **Terminated Queue**

각 프로세스마다 가지고 있던 CPU burst time 만큼의 시간 동안 running state 에 머물렀던 프로세스는 CPU remaining time 이 0 이 되는 시점에 terminated queue 로 들어가게 된다. 프로세스가 종료된 순서는 각 프로세스가 이 terminated queue 에 들어간 순서와 같다. 시뮬레이션이 끝나고나서 알고리즘을 평가할 때에는 terminated queue 에 들어간 프로세스 구조체의 정보를 가지고 평가하게 된다.

시뮬레이터 모듈 구성

전체적인 흐름은 위의 **시뮬레이터 구성도**에서 서술한 바와 같이 진행된다. 시뮬레이팅 단계를 크게 나누어보면 크게 **세** 단계로 나눌 수 있다.

1. 프로세스의 생성

2. 스케줄링 알고리즘 선택 및 시뮬레이팅

3. 선택한 알고리즘의 시뮬레이팅 결과 분석

1. 프로세스의 생성

시뮬레이션을 위해서는 프로세스의 생성이 필요하다. 프로그램 시작 단계에서 사용자로부터 입력받은 인자를 가지고 필요한 수 만큼의 프로세스를 생성하고, I/O 프로세스도 구성한다. C언어의 구조체 문법을 사용해서 프로세스를 정의하였으므로 프로세스를 생성하는 과정은 곧 메모리에 해당 구조체만큼의 크기를 할당받아서 그 내부 메모리 공간에 필요한 멤버 변수들의 값을 저장하는것과 같다. 이렇게 생성된 프로세스들은 process queue 에 저장된다.

2. 스케줄링 알고리즘 선택 및 시뮬레이팅

시뮬레이팅은 생성된 모든 프로세스가 종료되는 시점까지 진행된다. 즉, 얼마만큼 시뮬레이팅을 진행할지 사전에 정해두는 것이 아니라 마지막 프로세스가 종료되는 시점까지 수행을 한다는 의미이다. isDone 이라는 전역 flag 변수를 두어서 시뮬레이팅이 종료된 경우 이 값을 TRUE로 바꾸어서 시뮬레이팅 loop을 빠져나오도록 구성했다.

스케줄링 알고리즘은 switch 문을 통해 1(FCFS)부터 6(Round Robing)까지 선택할 수 있으며, 반복문을 통해서 여러 알고리즘을 사용해 반복적으로 시뮬레이팅 할 수 있도록 구성했다.

필요한 모든 인자값을 정상적으로 받았을 경우 시뮬레이팅을 시작한다. 본격적인 시뮬레이팅 과정을 수행하기 전에 CPU utilisation 을 측정하기 위한 computationStart 라는 변수의 값을 초기화해주고 computationIdle 이라는 변수의 값도 초기화 해준다. 이러한 초기화 과정이 끝난 후에는 시뮬레이션 루프를

시작하여 시뮬레이팅을 한다. 여기에서 1번의 루프는 1만큼의 time 이 지났음을 의미한다. 전체적인 과정을 pseudo-code 로 표현하면 아래와 같다.

```
void startSimulation (algorithm, preemptive, time_quantum ) {
    /* initialisation for simulating */
    while ( ! isDone ) {     // 1번의 while문 실행은 1 time이 흘렀음을 의미
        simulate(current_time, algorithm, preemptive, time_quantum);
        if ( simulating is done ) isDone = TRUE;
        else continue;
    }
    /* analyze algorithm */

    /* initialisation for iteration of this simulating */
```

```
void simulate (current_time, algorithm, preemptive, time_quantum ) {
    // pick a process from ready queue
    prevProcess = getProcessFromReadyQueue();
    // pick one process through scheduling algorithm
    runningProcess = scheduling(algorith, preemptive, time_quantum);
    if ( prevProcess != runningProcess) CPU를 점유하고 있던 시간 = 0;
    /* Ready Queue에 있는 프로세스들의 turnaround time과 waiting time ++ */
    /* Waiting Queue에 있는 프로세스들의 turnaround time++ 및 IO remainging time -- */


    /* runningProcess 의 CPU remaining time--, turnaround time++ */
    if ( runningProcess 존재)
        if ( runningProcess 의 CPU remaining time <= 0 ) runningProcess 를 terminate;
        else
            if ( runningProcess 의 I/O 발생 ) waiting queue로 옮김.
    else   idleTime++;
```

3. 선택한 알고리즘의 시뮬레이팅 결과 분석

2번 과정이 완료되면, while 문을 빠져나오면서 바로 analyzing code 가 실행된다. Terminated queue 에 있는 프로세스들의 멤버 변수 값들을 가지고 Average waiting time, Average turnaround time, CPU utilisation 를 계산한다. 계산이 완료되었으면 적절한 형식(테이블)을 가진 형태로 콘솔창에 출력하는 함수를 호출하여 결과를 사용자에게 보여준다.

2.4 Scheduling Algorithm 구현

2.3 에서의 pseudo-code 안에 scheduling 이라는 함수로 running할 프로세스를 가져오는 부분이 있는데, 이 scheduling 함수안에서는 algorithm 변수에 들어가 있는 값에 따라 switch 문으로 분기하여 어떤 알고리즘을 적용할지 판단해준다.

각 스케줄링 알고리즘(FCFS, SJF(preemptive, non-preemptive), PRIORITY(preemptive, non-preemptive), Round Robin)들이 어떻게 구현되어 있는지 pseudo-code 로 표현하면 아래와 같다.

1) FCFS

이 알고리즘에는 기본적으로 preemption 이 없으므로 이는 제외하고 생각한다.

```
processPointer FCFS_alg() {
    earliestProcess = getProcessFromReadyQueue();
    if ( earliestProcess 존재 )
        if ( runningProcess 가 존재 ) return runningProcess;
        else earliestProcess 를 readyqueue에서 제거하고 earliestProcess return;
    else return runningProcess;
```

2) SJF

매 time unit 마다 ready queue 에서 CPU remaining time 이 가장 작은 프로세스를 선택하면 된다. Non-preemptive 라면 running 되던 프로세스를 그대로 실행하면 되고, preemptive 라면 선정한 프로세스의 remaining time 과 running 프로세스의 remaining time 을 비교하여 더 작은 프로세스를 running 프로세스로 선택한다.

```
processPointer SJF_alg(preemptive)
    processPointer shortestProcess = getProcessFromReadyQueue();
    if ( shortestProcess 존재 )
        for each process in ReadyQueue
            if ( process 의 remaining time < shortestProcess 의 remaining time )
                shortestProcess = process;

        if ( runningProcess 존재 )
            if ( preemptive )
                if ( runninProcess의 remaining time > shortestProcess의 remaining time )
                    return shortestProcess;
                return runningProcess;

            else return runningProcess;
        else return shortestProcess;        // runningProcess가 없으면 선정한 프로세스 반환

    else return runningProcess;      // shortestProcess 존재하지 않을 경우
```

3) Priority

SJF 알고리즘과 동일한 pseudo-code 구조를 갖는다. 유일한 차이점은 remaining time을 비교하는 부분이 priority 를 비교하는 코드로 바뀐 것 뿐이다.

4) Round Robin

기본적인 구조는 FCFS 알고리즘과 동일하다. 매 time unit 마다 FCFS 방식으로 ready queue 에서 프로세스 하나를 선정하는 과정은 동일하나, 이미 running 중인 프로세스가 존재할 경우 해당 running 프로세스의 실행 시간이 time quantum 을 넘었는지 체크하여 preemption 이 일어나도록 하는 점이 다르다. 만약 time quantum 만큼 CPU를 사용한 프로세스가 time quantum 이 끝나는 시점에 CPU burst time 을 모두 소진하게 되면, 이 프로세스는 ready queue 로 돌아가지 않고 바로 terminated queue 로 이동하여 수행을 종료한다.

```
processPointer RR_alg(time_quantum)
    processPointer nextProcess = getProcessFromReadyQueue();
    if ( nextProcess 가 존재 )
        if ( runningProcess 가 존재 )
            if ( 수행시간 >= time_quantum )
                runningProcess를 ready queue로 옮긴다.
                return nextProcess;

            else return runningProcess;
        else return nextProcess;            // runningProcess가 없을 때
    else return runningProcess;             // nextProcess가 없을 때
```

2.4 시뮬레이터 평가 지표

각 알고리즘의 성능을 평가하고 비교 분석하기 위한 지표로는 세 가지를 사용 하였다.

1) Average Waiting Time

프로세스가 단순히 Ready queue 에 상주하는 시간만을 고려했다. Waiting queue 에 있는 시간까지 합하게 되면 I/O 작업으로 인한 수행 시간에 의해 스케줄링 성능이 영향을 받을 수 있기 때문이다.

2) Average Turnaround Time

프로세스가 처음 Ready queue 에 도달한 시간부터 해당 프로세스가 terminated 된 시간까지를 그 프로세스의 turnaround time으로 보았다. 따라서 프로세스가 ready queue에 있을 때, running state에 있을 때 turnaround time을 매 time unit마다 1씩 증가시킴으로써 terminated된 시점의 turnaround time 변수 값으로 이를 측정 할 수 있다.

3) CPU Utilisation

시뮬레이팅을 시작할 때(첫 프로세스가 도착한 시점)부터 시뮬레이팅이 끝날 때(마지막 프로세스가 끝난 시점)까지 CPU 이용률을 계산한다. 시뮬레이션이 끝난

후,

(computationEnd – computationIdle) / (computationEnd – computationStart) * 100

의 식을 통해서 utilisation값을 계산한다.

2.5 실행 결과

① FCFS

프로그램 실행 시 사용자로부터 인자값을 전달 받는 부분.

```
PLEASE ENTER THE NUMBER OF PROCESSES IN TOTAL : 8
PLEASE ENTER THE NUMBER OF IO PROCESSES IN TOTAL : 3
PLEASE ENTER TIME QUANTUM AMOUNT (at least greater than or equal to 3): 4


================ MODE ===============
        1. FCFS
        2. SJF(non-preemptive)
        3. SJF(preemptive)
        4. PRIORITY(non-preemptive)
        5. PRIORITY(preemptive)
        6. ROUND ROBIN
        7. EXIT
====================================
SELECT MODE :
```

매 time unit마다 어떤 프로세스를 선정하고 시뮬레이팅하는지 보여주는 부분.

```
SELECT MODE : 1

<FCFS Algorithm>
0: (pid: 2) -> running
1: (pid: 2) -> running
2: (pid: 2) -> running
3: (pid: 2) -> running
4: (pid: 2) -> running
5: (pid: 2) -> running
6: (pid: 2) -> running
7: (pid: 2) -> running-> terminated
8: (pid: 7) -> running
9: (pid: 7) -> running
10: (pid: 7) -> running
11: (pid: 7) -> running
12: (pid: 7) -> running
13: (pid: 7) -> running
14: (pid: 7) -> running
15: (pid: 7) -> running
16: (pid: 7) -> running-> terminated
17: (pid: 5) -> running
```

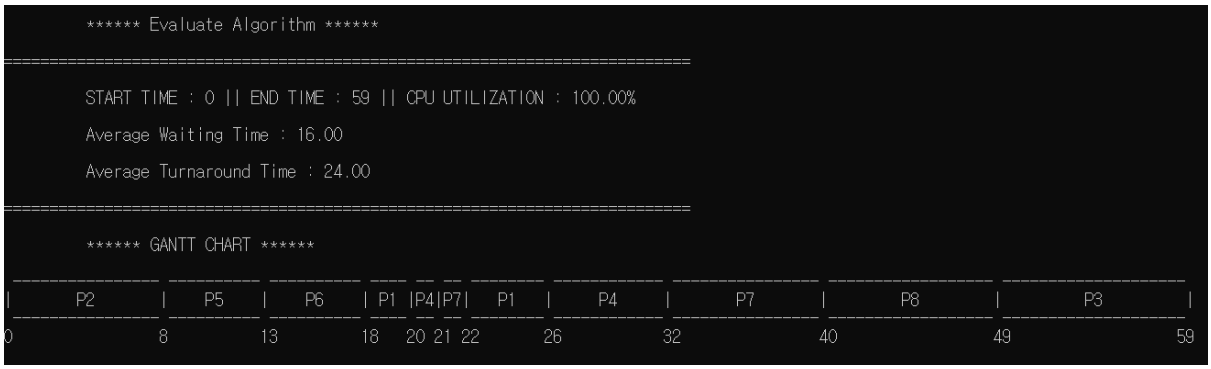
프로세스 정보를 보여주는 테이블 및 알고리즘 평가 출력(FCFS)

```
        ****** Process Table ******

+-----+-----------+---------+-------------+-----------------+--------------+----------+
| PID | Burst Time | IO Time | Waiting Time | Turnaround Time | Arrival Time | Priority |
+-----+-----------+---------+-------------+-----------------+--------------+----------+
|  2  |      8     |    0    |      0      |        8        |      0       |    5     |
+-----+-----------+---------+-------------+-----------------+--------------+----------+
|  7  |      9     |    0    |      6      |       15        |      2       |    6     |
+-----+-----------+---------+-------------+-----------------+--------------+----------+
|  5  |      5     |    0    |     11      |       16        |      6       |    8     |
+-----+-----------+---------+-------------+-----------------+--------------+----------+
|  3  |     10     |    0    |     15      |       25        |      7       |    6     |
+-----+-----------+---------+-------------+-----------------+--------------+----------+
|  8  |      9     |    0    |     24      |       33        |      8       |    7     |
+-----+-----------+---------+-------------+-----------------+--------------+----------+
|  1  |      6     |    2    |     39      |       47        |      9       |    4     |
+-----+-----------+---------+-------------+-----------------+--------------+----------+
|  4  |      7     |    1    |     42      |       50        |      9       |    3     |
+-----+-----------+---------+-------------+-----------------+--------------+----------+
|  6  |      5     |    3    |     39      |       44        |      9       |    7     |
+-----+-----------+---------+-------------+-----------------+--------------+----------+


        ****** Evaluate Algorithm ******

================================================================================

        START TIME : 0 || END TIME : 59 || CPU UTILIZATION : 100.00%

        Average Waiting Time : 22.00

        Average Turnaround Time : 29.00

================================================================================
```

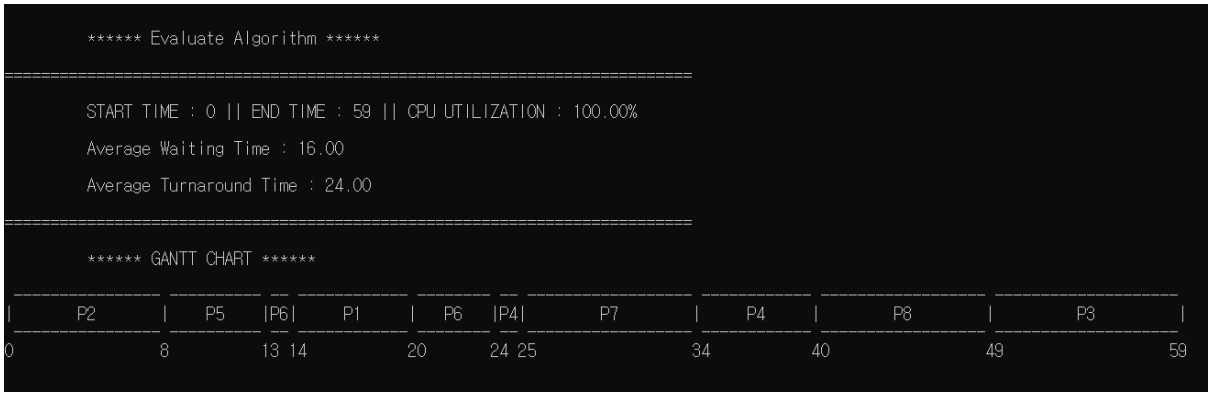
간트 차트 출력(FCFS)

```
        ****** GANTT CHART ******

 ____________ _____________ ________ _____________ _____________ _____ _______ ________ _____ _____
|     P2     |     P7     |   P5   |     P3      |     P8      | P1  |  P4   |   P6   | P1  | P4  |
 ____________ _____________ ________ _____________ _____________ _____ _______ ________ _____ _____
0            8            17       22            32            41    44      48       53    56    59
```

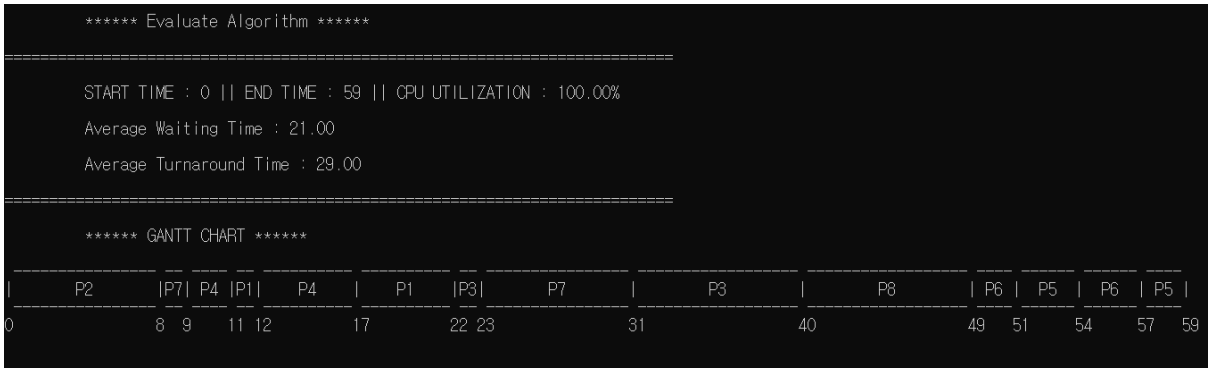
② SJF(preemptive)



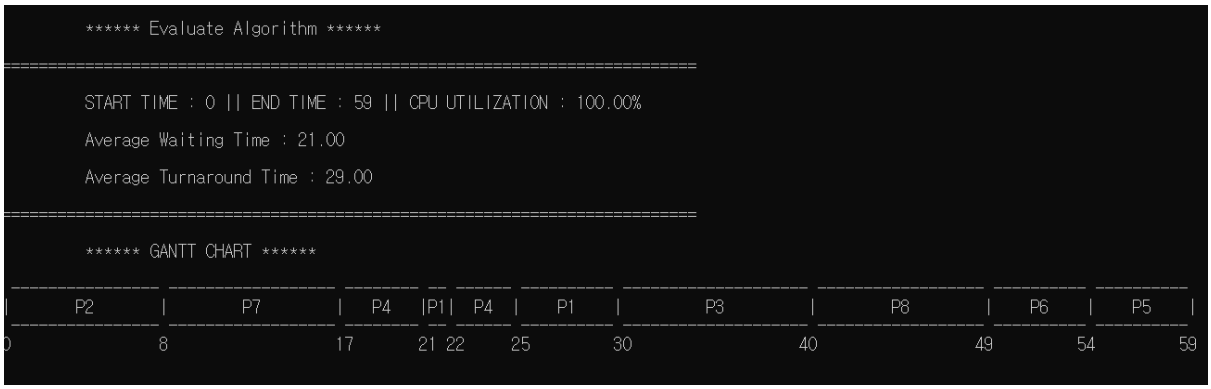
③ SJF(non-preemptive)



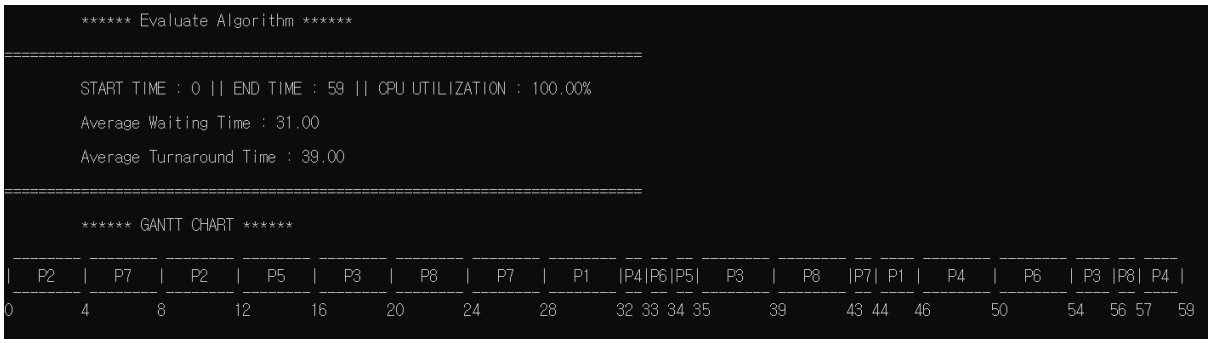
④ Priority(preemptive)



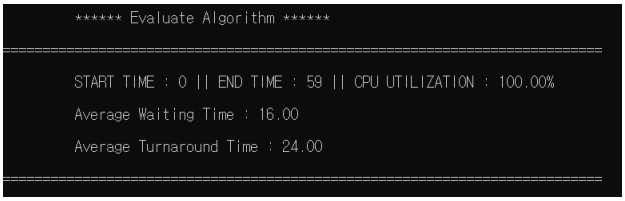
⑤ Priority(non-preemptive)

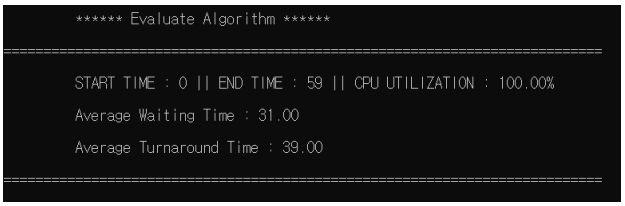


⑥ Round Robin



간트 차트를 제외하고 성능분석 부분만 모아서 보면 다음과 같다.





왼쪽 상단은 FCFS, 오른쪽 상단은 SJF-preemptive, 왼쪽 중간은 SJF-nonpreemptive, 오른쪽 중간은 Priority-preemptive, 왼쪽 하단은 Priority-nonpreemptive, 오른쪽 하단은 Round Robin에 대한 평가 결과이다.

3. 프로젝트 결론

3.1 알고리즘 비교 분석

① CPU Utilisation

Arrival time이 늦은 프로세스들(time 9에 도착)이 전부 I/O프로세스가 되었다. 이러한 상황에서는 이보다 앞서 들어온 프로세스들이 전부 종료되고나서 뒤늦게 들어온 I/O프로세스들이 I/O작업에 들어가게 되면 Idle 구간이 생길 확률이 높아지게된다. 하지만 위의 FCFS 시뮬레이션에서는 운이 좋게도 non I/O프로세스들이 적당히 앞에 몰려 도착했고, I/O프로세스인 P1, P4, P6은 모두다 같은 시간에 들어와서 Ready queue가 비어있는 구간이 없었다. 또한 P6는 자신의 CPU time 내에 I/O 작업을 수행하지 않아서 계속 CPU를 점유하고 있었던 것도 Utilisation 상승에 기여했다고 볼 수 있다.

② Average Waiting Time 과 Average Turnaround Time

모든 스케줄링 알고리즘의 경우에 CPU Utilisation이 100%라 이 지표만으로는 직접적인 비교가 불가능하다. 하지만 ATT와 AWT을 보게되면 확연한 차이가 드러난다. 6개의 알고리즘 중에서 가장 나은 성능을 보인것은 Shortest Job First(SJF) 알고리즘이다. Average Wainting Time = 16, Average Turnaround Time = 24로써 AWT나 ATT측면에서만 본다면 최상의 알고리즘이라고 할 수 있다. 하지만 알고리즘 소개 파트에서 언급했듯이, shortest job이 무엇인지 사전에 파악하기가 현실적으로 불가능하기에, 아쉽지만 실제 세계에서의 사용은 무리가 있을 듯 싶다.

Priority 방식을 보면 SJF에 이어 두 번째로 나은 성능을 보여주는데, 이것은 명확한 이유에서라기 보다는 우연적인 것 같다. 프로세스 테이블을 보고 간트 차트를 추적해 보면, 주어진 프로세스들의 arrival time과 priority 값을 가지고 Priority 알고리즘을 사용할 경우 SJF와 비슷한 방식으로 프로세스들을 처리하는 것을 알 수 있다. 그래서 SJF와 거의 비슷한 성능을 보인 것이다.

Round Robin 방식은 AWT = 31, ATT = 30로써 가장 성능이 낮게 나왔는데, 이는 time quantum이 지날 때 마다 running 프로세스를 바꾸는 데에서 기인한 당연한 결과이다. 프로세스의 어떤 한 특성에 가중치를 주지 않고 모든 프로세스를 공평하게 처리하는 방식의 단점이라고 할 수 있다.

결론적으로, 이상적이긴 하나 SJF 알고리즘이 가장 괜찮은 성능을 보인다는 점을 알 수 있었다.

3.2 수행 소감 및 추후 발전 계획

직접 CPU Scheduling Simulator를 구현해보면서 프로세스가 생성되고 소멸되는 과정에서 어떤 일들이 일어나는지 직접적으로 알 수 있었다. 또한 이론적으로만 알고 있었던 각 Scheduling algorithm들의 동작 원리도 세세하게 알게 되었다. 그리고 Gantt chart를 나타내는 부분을 많이 고민했는데, 이 과정에서 데이터를 어떻게 처리하여 원하는 결과를 출력할지에 대한 입출력 프로세스에 대한 이해가 한 층 높아진 것 같다.

하지만 몇 가지 아쉬운 점도 있다. 하나는 추가적인 알고리즘을 구현하지 않고 기본 알고리즘만 구현했다는 것인데 시간적인 여건이 안되어서 추가적인 스케줄링 알고리즘을 구현하지 못했다. 나중에 여유가 있을 때 도입부에 간략히 소개한 HRN알고리즘을 추가해서 성능을 확인해 보고 싶다.
또 다른 아쉬운 점 하나는 I/O 환경에 대한 가정이 현실과 조금 떨어져 있다는 것이다. 본 시뮬레이터에서는 I/O 작업이 random하게 발생하기는 하지만 그 횟수가 딱 1회로 정해져있다. 하지만 현실세계에서는 I/O작업이 프로세스에서 딱 1회만 일어나라는 법이 없기 때문에 이는 조금 이상적인 가정이라고 할 수 있다. 물론 횟수도 여러번 일어나게끔 코드를 작성할 수도 있었지만, 그렇게하지 않은 이유는 그렇게 했을 경우에 분석과정이 매우 복잡해지기 때문이다. 프로세스의 숫자도 최대 15개로 상당히 많은데, 여기에 I/O 발생이 random하고 반복적이기까지 한다면 간트차트로 프로세스의 실행순서를 추적하기가 상당히 까다로울 것으로 예상했다.

각 Scheduling algorithm에 따라 그에 맞는 Gantt chart가 보기 좋게 출력되었을 때 많이 뿌듯했다. 본 시뮬레이터 구현 프로젝트를 통해 자신감을 많이 얻었고 코딩에 대한 재미를 조금 더 느낄 수 있었다.