OS Term project

- CPU Scheduling Simulator -

과목명: 운영체제

학번: 2017320195

학과: 컴퓨터학과

이름: 김영훈

INTRODUCTION

운영체제는 컴퓨터 시스템을 이루는 중요한 여러가지 요소 중 하나이다. 그 중에서도 컴퓨터의 여러가지 중요 자원(CPU, I/O)를 할당, 제어하는 것은 효율적인 컴퓨터 시스템의 이용에 핵심적인 부분이라 할 수 있다. CPU 는 여러가지 명령어를 처리하는 컴퓨터의 자원 중 중추적 역할을 한다, 다중 프로그래밍 시스템의 개념이 등장하고나서 시스템 내 존재하는 여러 프로세스에 대해 어떤 프로세스에게 얼마나 긴 시간만큼의 CPU 를 할당할 지의 문제, 즉 CPU 스케줄링은 CPU 이용률을 높이기 위해 필수적으로 고민해야할 부분이다.

본 프로젝트는 운영체제의 기본 개념을 바탕으로 Linux 환경에서 C 언어로 단일 CPU 환경에서의 CPU 스케줄링 시뮬레이터를 구현하였다. 이를 통해서 CPU 스케줄러의 역할 및 작동 원리를 이해하고 또한 전반적인 C 언어 개발 능력 향상을 목적으로 하였다. 본 프로젝트에서 구현한 시뮬레이터를 통해 임의로 생성된 여러 프로세스가 처리되는 과정을 살펴보고, 다양한 CPU 스케줄링 알고리즘의 차이와 몇 가지 척도에서의 성능을 비교 분석하였다.

CPU 스케줄러는 여러 프로세스가 존재할 때 각각의 프로세스에게 CPU 를 할당, 제어하는 역할을 한다. CPU 는 특정 시점에서 하나의 프로세스만 처리할 수 있기 때문에 각 프로세스에게 적절하게 CPU 를 할당할 필요가 있다. CPU 를 할당 받은, 즉 CPU 에 의해 처리되고 있는 프로세스는 실행 상태에 있다고 한다. 그 외 CPU 를 할당 받기 위해 프로세스들은 준비완료 큐라는 자료구조에 의해 관리되는데, 이들 프로세스를 준비 상태에 있다고 한다. 마지막으로 I/O 를 처리 중이거나 기타 이벤트를 기다리는 프로세스들은 대기 큐에서 처리중인 작업이 끝나기를 기다리며, 대기 상태에 있다고 한다.

한 프로세스의 상태는 프로세스가 종료되기까지 다양하게 변할 수 있다. 준비 상태에 있는 프로세스는 CPU 스케줄러에 의해 CPU 를 할당 받아 실행 상태로 전환된다. 실행 상태에 있는 프로세스는 다른 프로세스에 의해 선점되어 준비 상태로 전환되거나 I/O 가 처리되기를 기다리기 위해 대기 상태로 전이할 수 있다. 대기 상태에 있는 프로세스들은 처리 중인 I/O 가 끝나면 인터럽트를 발생시키고 준비 상태가 된다. 이렇게 여러 프로세스가 복잡하게 CPU 를 사용하면서 다중 프로그래밍 시스템에서의 동시성을 사용자에게 제공하는 것이다.

앞서 다룬 프로세스간 상태전이 중에서 CPU 스케줄러는 어떤 프로세스가 CPU 를 할당 받는지, 즉 준비 상태에 있는 프로세스 중 어떤 프로세스에게 CPU 를 할당할 것인지를 결정한다. 프로세스는 다양한 기준에 의해 스케줄링 될 수 있으며, 본 프로젝트에서 구현한 알고리즘은 다음과 같다

- First-Come, First-Served(FCFS):

  준비 큐에 먼저 도착한 프로세스가 CPU 를 먼저 할당 받는 알고리즘이다.

- Non-Preemptive Shortest-Job-First:

  준비 큐에 있는 프로세스 중 CPU 처리 시간이 가장 짧은 프로세스에게 CPU 를 할당하는 알고리즘이다.

● Preemptive Shortest-Job-First:

CPU 가 처리중인 프로세스가 없는 경우, 준비 큐에 있는 프로세스 중 CPU 처리 시간이 가장 짧은 프로세스에게 CPU 를 할당한다. CPU 가 처리 중인 프로세스의 남은 CPU 처리 시간보다 CPU 처리 시간이 더 짧은 프로세스가 준비 큐에 도착하면 처리 중인 프로세스는 CPU 처리 시간이 더 짧은 프로세스에 의해 선점된다.

● Non-Preemptive Priority:

준비 큐에 있는 프로세스 중 우선순위가 가장 높은 프로세스에게 CPU 를 할당하는 알고리즘이다.

● Preemptive Priority:

기본적으로 우선순위가 가장 높은 프로세스에게 CPU 를 할당한다. 이후 실행 상태에 있는 프로세스의 우선순위보다 더 높은 우선순위의 프로세스가 준비 상태에 있으면 처리 중이던 프로세스는 선점된다.

● Round-Robin(RR):

프로세스는 알고리즘이 명시한 시간만큼(Time Slice, Time Quantum)만 CPU 를 할당 받을 수 있다. 만약 해당 시간이 되기 전에 프로세스가 종료되거나 I/O 를 처리하기 위해 대기 상태로 전이하면 준비 상태에 있는 다른 프로세스에게 CPU 를 할당한다.

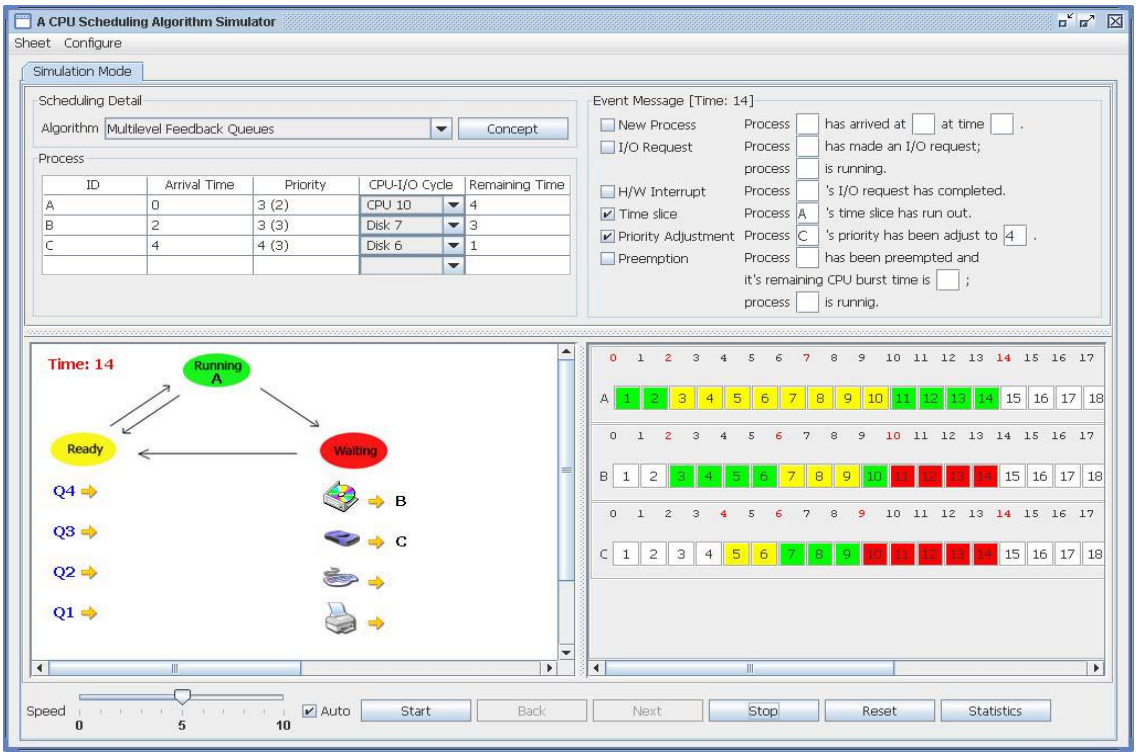

FIGURE 1
SNAPSHOT OF THE SIMULATOR BY SUKANYA SURUNAUWARAT

CPU SCHEDULING SIMULATOR

Sukanya Suranauwarat(2007)의 CPU 스케줄링 시뮬레이터를 살펴보자. Sukanya 에 따르면 그의 CPU 스케줄링 시뮬레이터는 세 가지 특징을 가지고 있다고 한다. 첫 번째, 프로세스를 사용자가 직접 설정할 수 있다. 프로세스가 생성되는 시간, 프로세스의 우선순위, CPU 처리시간 등을 정할 수 있다. 두 번째, 그래픽 사용자 인터페이스(Graphic User Interface)를 통해 시간이 흐름에 따라 스케줄링이 어떻게 이루어지는지 시각적으로 확인할 수 있다. 마지막으로는 사용자가 스케줄링 알고리즘을 잘 이해하고 있는지 확인해볼 수 있다.

Sukanya 의 시뮬레이터에는 두 가지 모드가 있다. 첫 번째는 시뮬레이션 모드로, 이미 구현된 스케줄링 알고리즘과 임의로 생성된 프로세스로 시뮬레이션을 해볼 수 있다. 알고리즘에 따라 필요한 변수의 경우, 예를 들어 Round Robin 알고리즘에서의 Time Slice 같은 변수의 값을 사용자가 직접 설정할 수 있다. 프로세스들은 기본적으로 생성되지만 이 또한 사용자가 임의로 추가, 수정할 수 있다. 시뮬레이션 도중에는 중단, 재개버튼을 통해서 시뮬레이션 진행 과정을 단계별로 확인할 수 있으며, 또한 각 프로세스별로 대기 상태에 있는지, 준비 상태에 있는지 등을 시각적으로 확인할 수 있다. 시뮬레이션이 끝나면 각 프로세스의 응답 시간, 대기 시간, 반환 시간을 알 수 있으며, 모든 프로세스의 평균 대기 시간, 평균 반환 시간 등도 확인할 수 있다.

두 번째 모드는 연습 모드이다. 연습모드는 사용자의 편의를 위해 최대한 시뮬레이션 모드와 비슷한 인터페이스를 제공한다. 시뮬레이션 모드와의 차이는 연습 모드에서는 시작, 중지, 재개 버튼이 없다는 것이다. 대신에 연습해보고자 하는 스케줄링 알고리즘을 선택한다. 그리고 각 프로세스가 특정 시점에 어떤 상태에 있는지를 사용자가 직접 설정한다. 설정 도중, 혹은 과정이 모두 끝나고 나면 확인 버튼을 눌러 사용자가 연습해보고자 했던 알고리즘과 사용자가 입력한 프로세스의 스케줄링 과정이 부합하는지 확인해서 결과를 알려준다. 이와 같이 연습 모드를 활용하면 CPU 스케줄링 알고리즘을 제대로 이해하고 있는지 확인할 수 있다.

본 프로젝트에서는 그래픽 사용자 인터페이스 대신 콘솔에 정보를 출력하는 텍스트 기반 사용자 인터페이스를 사용했다. 구현 환경은 Ubuntu 18.04.01 LTS, gcc 7.4.0 를 사용했다.

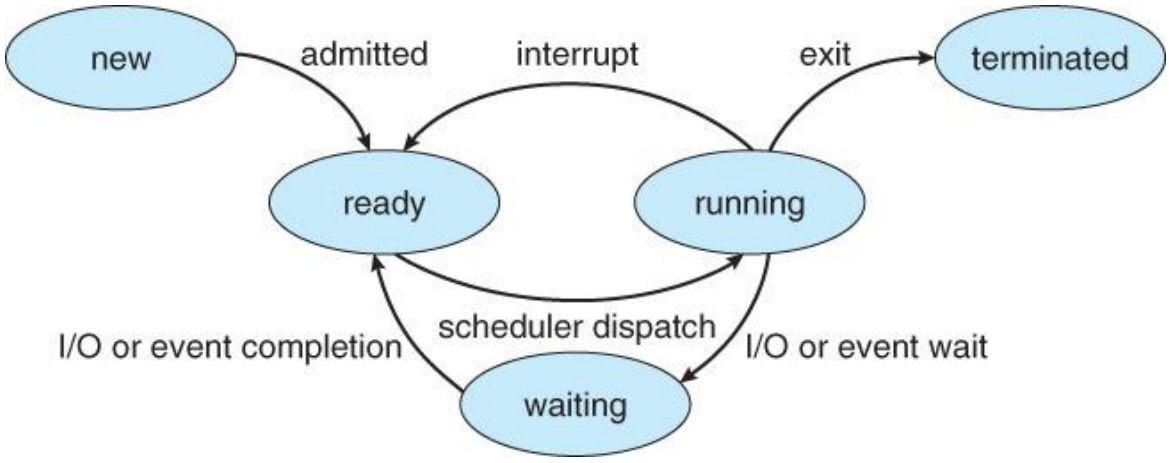

FIGURE 2
DIAGRAM OF PROCESS STATE

프로세스의 생성되고 종료되기까지의 과정은 FIGURE 2 를 따른다. 먼저 프로세스가 생성되면 생성된 프로세스는 준비 큐에 삽입된다. 준비 큐에 있는 프로세스들은 CPU 스케줄러에 따라 CPU 를 할당 받아 실행 상태로 전이한다. CPU 에 의해 처리 중이던 프로세스는 I/O 작업 등의 이벤트가 발생하게 되면 이벤트가 끝나기를 기다리기 위해 대기 큐로 들어가고, 이후 이벤트가 끝나면 인터럽트를 발생시키며 다시 준비 큐에 들어가게 된다.

본 프로젝트의 CPU 스케줄링 시뮬레이터는 대체로 FIGURE 2 의 구조를 따랐다. 실시간으로 생성되는 프로세스와 달리 본 프로젝트의 시뮬레이터는 시뮬레이션을 하기 전에 미리 프로세스를 생성해두기 때문에 생성한 프로세스들을 관리하는 큐가 존재하며 완전히 종료한 프로세스들은 또한 별도의 큐에 의해 관리된다. 두 개의 큐는 도착 시간에 대한 최소 힙(Min heap, 도착 시간이 이른 프로세스가 루트 노드에 위치)으로 구현했다. 또한 대기 큐는 남은 I/O 처리 시간에 대한 최소 힙으로 구현했으며, 준비 큐는 각 스케줄링 알고리즘에 적합한 큐를 사용한다. 예를 들면, SJF 알고리즘에서의 준비 큐는 남은 CPU 처리 시간에 대한 최소 힙을 사용하며, FCFS 알고리즘은 일반적인 선형 큐를 이용한다.

시뮬레이션을 위한 프로세스는 무작위로 생성된다. 각 프로세스는 PID, CPU 처리 시간, I/O 처리 시간, 도착시간, 우선순위 값을 가지며 각각의 값은 무작위로 부여된다. 우선 PID 는 프로세스를 구별하는 고유한 번호로, 전체 프로세스의 수가 N 개라 하면 각 프로세스의 PID 는 1 에서 N 까지의 값을 갖는다. CPU 처리 시간은 1 에서 40 사이의 값을 가진다. 일반적으로 CPU 처리 시간이 짧은 프로세스의 수가 많고 처리 시간이 긴 프로세스의 수는 적다. 이를 위해 CPU 처리 시간은 0.9 의 확률로 1 에서 10 사이의 값을 갖도록 했으며, 0.1 의 확률로 11 에서 40 사이의 값을 갖는다. I/O 처리 시간은 0 보다 크거나 같고 20 보다 작은 값을 가진다. 총 프로세스의 수가 많을수록 전체 시뮬레이션에 걸리는 시간도 평균적으로 길어지는 것을 고려하여 프로세스의 도착 시간의 최댓값은 전체 프로세스의 수에 비례하도록 했다. 마지막으로 프로세스의 우선순위는 -20 보다 크거나 같고 20 보다 작거나 같은 값을 가지며, 값이 작을수록 높은 우선순위를 의미한다.

PID	1 ~ N
CPU 처리 시간	1 ~ 40
I/O 처리 시간	0 ~ 19
도착 시간	0 ~ N*3
우선순위	-20 ~ 20

FIGURE 3
VALUES THAT A PROCESS HAS

CPU SCHEDULING SIMULATOR MODULE

프로세스의 상태 전이를 관리하기 위해 적절한 자료구조를 사용할 필요가 있다. 자료구조는 단일 연결 리스트로 구현한 FIFO 큐와 이진 트리로 구현한 우선순위 큐를 사용했다. 자료구조 구현에 대한 내용은 생략하도록 한다. 시뮬레이터의 구현은 첫 째로 생성된 프로세스를 준비 큐에 넣는 과정이 필요하다. 프로세스는 시뮬레이션 전에 미리 생성해 두기 때문에 시뮬레이션의 경과시간과 프로세스의 도착 시간이 같아질 때 해당 프로세스를 준비 큐에 넣는다.

두 번째, CPU 가 처리 중인 프로세스가 없으며 준비 큐에서 기다리는 프로세스가 있다면 CPU 를 할당하는 작업이 필요하다. 이는 단순히 준비 큐에 있는 프로세스를 CPU 에게 전달해주기만 하면 된다. FCFS 알고리즘과 Round Robin 알고리즘의 경우 먼저 도착한 프로세스에게 CPU 를 할당하기 위해 FIFO 큐를 사용한다. 반면에 SJF 알고리즘이나 Priority 알고리즘의 경우 CPU 처리 시간이 가장 짧은 프로세스, 혹은 우선순위가 가장 높은 프로세스에게 CPU 를 할당한다. 일반적인 연결 리스트를 사용한다면 적합한 프로세스를 찾기 위해 준비 큐를 검색하는 과정이 필요할 것이다. 하지만 항상 스케줄링이 필요할 때마다 검색이 이루어지는 것을 피하기 위해 SJF 알고리즘이나 Priority 알고리즘의 경우 우선순위 큐를 사용했다. 따라서 스케줄링이 필요할 때마다 단순히 우선순위 큐의 루트 노드에 있는 프로세스에게 CPU 를 할당해주면 된다.

세 번째, 처리 중인 프로세스에서 I/O 요청이 발생하면 프로세스는 CPU 를 반납하고 대기 큐에 들어가야 한다. I/O 작업이 무작위로 일어나도록 하기 위해 확률 함수를 만들었다. 확률 함수의 값에 따라 I/O 요청이 무작위로 발생하고, 그에 따라 CPU 가 처리 중이던 프로세스를 대기 큐에 삽입하면 된다. 마지막으로 I/O 작업이 끝난 프로세스는 다시 준비 큐로 이동시켜야 한다. 대기 큐는 남은 I/O 처리 시간에 대한 우선순위 큐를 사용했다. I/O 처리가 끝난 프로세스를 매번 검색하지 않고 우선순위 큐의 루트 노드에 있는 프로세스의 남은 I/O 처리 시간을 확인해 필요시 해당 프로세스를 준비 큐에 넣는다.

```
Function FCFS
      Repeat
            Move processes that are newly generated
               or whose I/O works finished to ready queue.

            If CPU is IDLE:
                  Allocate CPU to the process in ready queue

            Waiting time of processes in ready queue++

            I/O burst time of processes in waiting queue--
            I/O interrupts occur randomly

            CPU burst time of the running process--
            I/O requests occur randomly

            Elapsed time++
      Until all processes terminate
```

FIGURE 4
PSEUDOCODE OF FCFS

FCFS 알고리즘 시뮬레이터의 의사코드는 FIGURE 4 와 같다. 먼저 새로 생성된 프로세스(시뮬레이션 경과 시간과 프로세스의 도착 시간이 같은 프로세스)나 I/O 작업이 끝난 프로세스를 준비 큐로 옮긴다. 생성된 프로세스를 관리하는 큐는 도착 시간이 빠를수록 우선순위가 높은 우선순위 큐로 구현했다. 그리고 대기 큐는 남은 I/O 처리 시간이 적은 프로세스에 대한 우선순위 큐를 사용한다. 따라서 각 큐의 루트 노드를 확인, 준비 큐로 이동해야 할 프로세스가 있는 경우 해당 프로세스를 이동시킨다. 이후 CPU 에게 프로세스를 할당한다. 이미 처리 중인 프로세스가 있는 경우 이 과정은 생략된다. 그 다음엔 준비 큐에서 기다리고 있는 프로세스들의 대기 시간을 증가시키고, 대기 큐에 있는 프로세스들의 남은 I/O 처리 시간을 감소시킨다. 이 과정에서 확률 함수를 호출, 무작위로 I/O 인터럽트를 발생시켜 프로세스들을 준비 큐로 옮긴다. 이어서 실행 상태에 있는 프로세스의 남은 CPU 처리 시간을 감소시키고 확률 함수를 호출하여 반환 값이 1 이라면 처리 중인 프로세스에서 I/O 요청이 발생했다고 가정하고 대기 큐로 옮긴다. 위의 과정을 모든 프로세스가 종료할 때까지 반복한다.

FCFS, Non-Preemptive SJF, Non-Preemptive 알고리즘은 모두 비선점 스케줄링 알고리즘이다. 이 때문에 세 알고리즘의 의사코드는 모두 동일하다. 다만 실제 구현에서는 FCFS 알고리즘에서의 준비 큐는 FIFO 큐, 다른 두 알고리즘의 준비 큐는 우선순위 큐를 사용한다는 차이가 있다. Preemptive SJF, Preemptive Priority 알고리즘의 의사코드는 크게 다르지 않지만 언급된 알고리즘들과 달리 선점 스케줄링 알고리즘이다. 이 때문에 기존에 실행 상태에 있는 프로세스를 선점할 수 있는 프로세스가 존재하는지 검사하고, 있다면 해당 프로세스가 기존 프로세스를 선점하는 과정이 추가된다. Round Robin 알고리즘은 기본적으로 FCFS 알고리즘과 유사하다. 준비 큐에 먼저 도착한 프로세스에게 CPU 를 할당하기 때문이다. 하지만 FCFS 알고리즘과 달리 각 프로세스는 한정적인 시간(Time Quantum)동안만 CPU 를 사용할 수 있다. 따라서 한 프로세스가 Time Quantum 만큼 CPU 를 사용했다면 준비 큐에 있는 다른 프로세스에게 CPU 를 할당하게 된다.

PERFORMANCE EVALUATION

CPU 스케줄링 시뮬레이터를 구현한 후, 6 가지 스케줄링 알고리즘에 대해서 시뮬레이션을 진행했다. 각 스케줄링 알고리즘의 성능을 평가하기 위한 평가 항목은 다음과 같다.

1. CPU 이용률(CPU Utilization)

   다중 프로그래밍 시스템에서는 CPU 가 쉬지 않게끔 CPU 를 최대한 이용하는 것이 효율적이다. 따라서 CPU 이용률을 측정해서 스케줄링 알고리즘의 효율을 평가할 수 있다.

2. 평균 대기 시간(Average Waiting Time)

   한 프로세스의 대기 시간은 프로세스가 준비 큐에서 머무른 시간의 합이다. 일반적으로 평균 대기 시간을 줄이는 것이 효율적이다.

3. 평균 반환 시간(Average Turnaround Time)

   한 프로세스의 반환 시간은 프로세스의 대기 시간과 CPU 를 사용한 시간, I/O 가 처리 되기를 기다린 시간의 합이다.

4. 최대 대기 시간(Maximum Waiting Time)

   모든 프로세스의 대기 시간을 측정한 후 그 중에서 가장 큰 값을 알고리즘 평가에 사용했다.

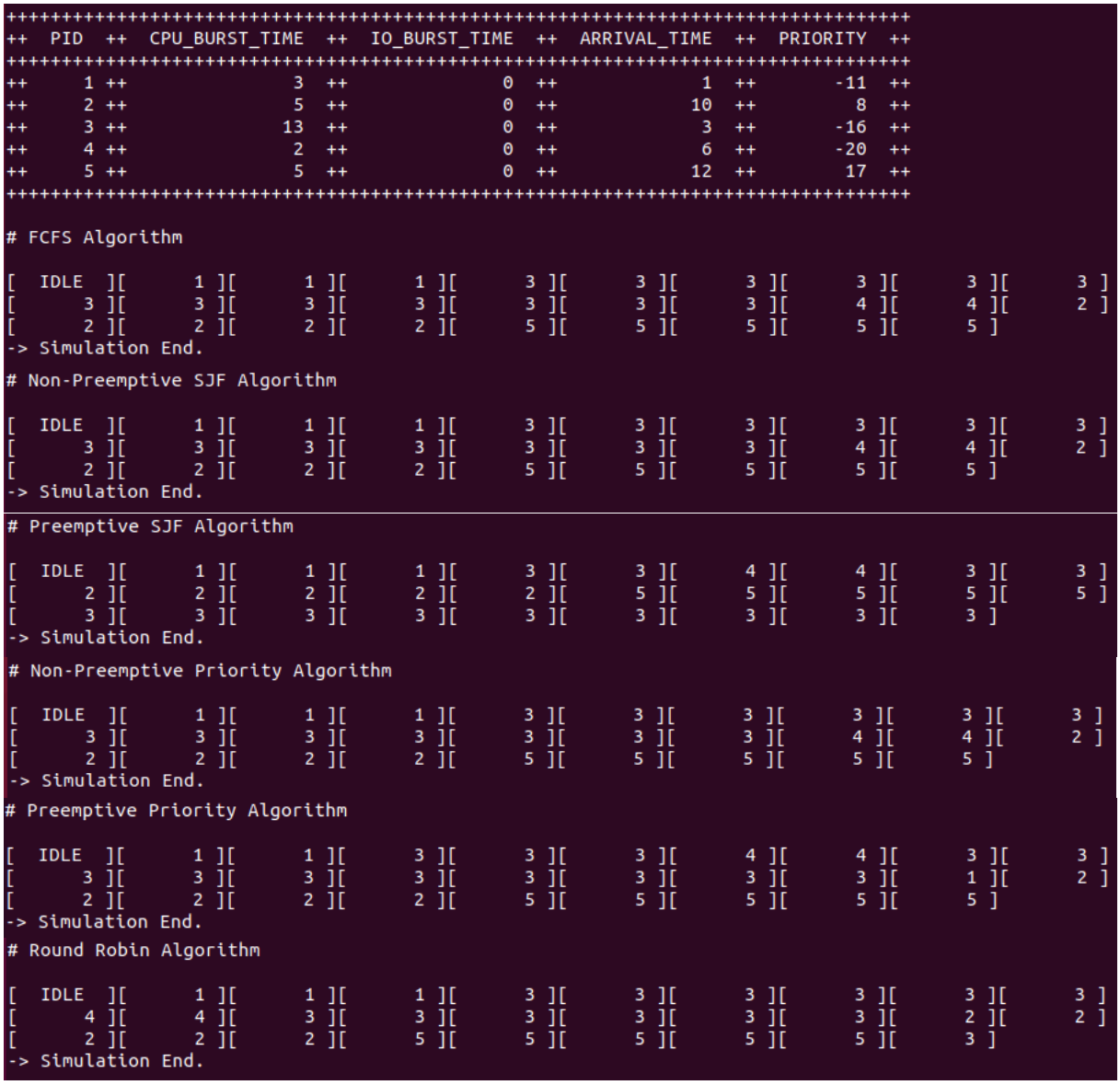

FIGURE 5
GANTT CHART OF ALGORITHMS, WITH NO I/O AND TIME QUANTUM OF 6 IN RR

먼저 I/O 작업이 없다고 가정하고(모든 프로세스의 I/O 처리 시간을 0 으로 고정) 1000 개의 프로세스에 대해서 시뮬레이션을 진행한 후 CPU 이용률, 평균 대기 시간, 평균 반환 시간, 최대 대기 시간을 측정했다. 무작위로 프로세스를 생성하는 시뮬레이터의 특성을 고려하여 100 번의 시뮬레이션에서 측정된 값들의 평균을 다시 계산했다. CPU 이용률은 모든 알고리즘에서 99%라는 수치가 나왔다. 평균 대기 시간, 평균 대기 반환 시간의 경우 Non-Preemptive SJF, Preemptive-SJF 알고리즘을 제외하고는 모두 약 2100 정도로 측정됐고 유의미한 차이는

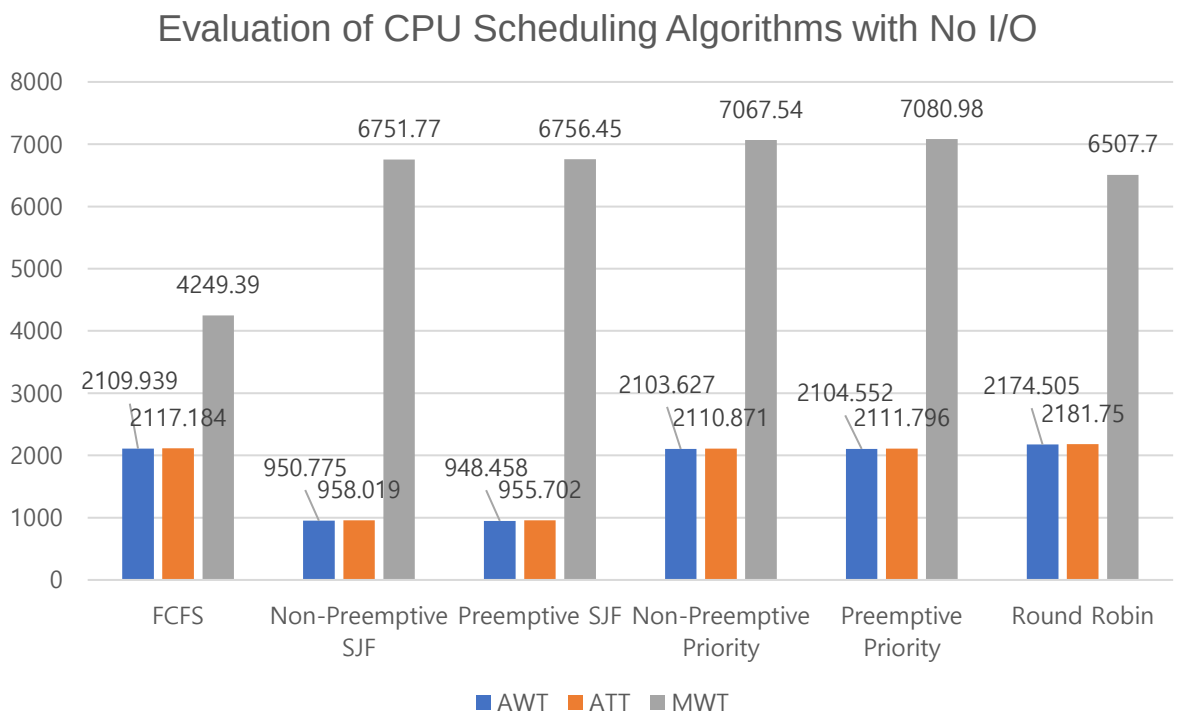

FIGURE 6
EVALUATION OF CPU SCHEDULING ALGORITHMS WITH NO I/O, TIME QUANTUM OF 6 IN RR

없었다. Non-Preemptive SJF, Preemptive-SJF 알고리즘은 다른 알고리즘에 비해 확연히 작은 약 950 정도의 평균 대기 시간, 평균 반환 시간을 보였다. 이는 SJF 알고리즘 자체가 평균 대기 시간에 대해 최적인 알고리즘이기 때문이다. 또한 Preemptive SJF 가 Non-Preemptive SJF 에 비해 평균 대기 시간, 평균 반환 시간이 작지만 그 차이는 2.317 로 무의미한 차이라고 볼 수 있다. 최대 대기 시간은 Preemptive Priority, Non-Preemptive Priority, Preemptive SJF, Non-Preemptive SJF, Round Robin, FCFS 순으로 크게 나타났다. Priority 알고리즘과 SJF 알고리즘 두 알고리즘 모두 스케줄링의 기준으로 우선순위가 고려된다. Priority 알고리즘은 우선순위가 높은 프로세스를, SJF 알고리즘의 경우 남은 CPU 처리 시간이 짧은 프로세스를 우선시한다. 따라서 우선순위가 낮은 프로세스들은 스케줄링이 되지 못하고 계속 준비 큐에서 기다리는 Starvation 이 나타난다. 시뮬레이션의 결과에서도 Priority, SJF 알고리즘의 Starvation 현상을 확인할 수 있다. Round Robin 알고리즘은 프로세스가 Time Quantum 만큼의 CPU 를 사용하면 다른 프로세스에게 CPU 를 할당하기 때문에 CPU 처리 시간이 긴 프로세스는 여러 번 CPU 를 할당 받기 위해 결과적으로 최대 대기 시간이 길어질 수 있다.

I/O 작업이 필요한 프로세스를 생성해서 마찬가지로 1000 개의 프로세스에 대해 100 번의 시뮬레이션을 진행했다. I/O 작업이 필요한 경우 I/O 작업이 없다고 가정했을 때와 다른 결과가 나왔다. CPU 이용률은 모든 알고리즘에서 99%를 보였다. I/O 작업이 없는 경우 SJF 알고리즘을 제외하고는 모든 알고리즘의 평균 대기 시간, 평균 반환 시간이 큰 차이를 보이지 않았다. I/O 작업이 있는 경우에도 Non-Preemptive SJF, Preemptive SJF 알고리즘이 평균 대기 시간, 평균 반환 시간이 약 950 으로 제일 낮았다. Non-Preemptive Priority, Preemptive Priority 알고리즘은 I/O 가 없다고 가정했을 때와 비슷한 약 2100 정도의 평균 대기 시간, 평균 반환 시간을 보였다. 하지만 FCFS, Round Robin 알고리즘은 I/O 가 없다고 가정했을 때보다 약 300 정도 평균 대기 시간, 평균 반환 시간이 증가했다. SJF, Priority 알고리즘에서는 I/O 를 처리하고 준비 큐에서 다시 CPU 를 할당 받기를 기다리더라도 대기 큐에서 기다리는 동안 새로운 프로세스가 생성되지 않았다면 준비 큐로 돌아오자 마자

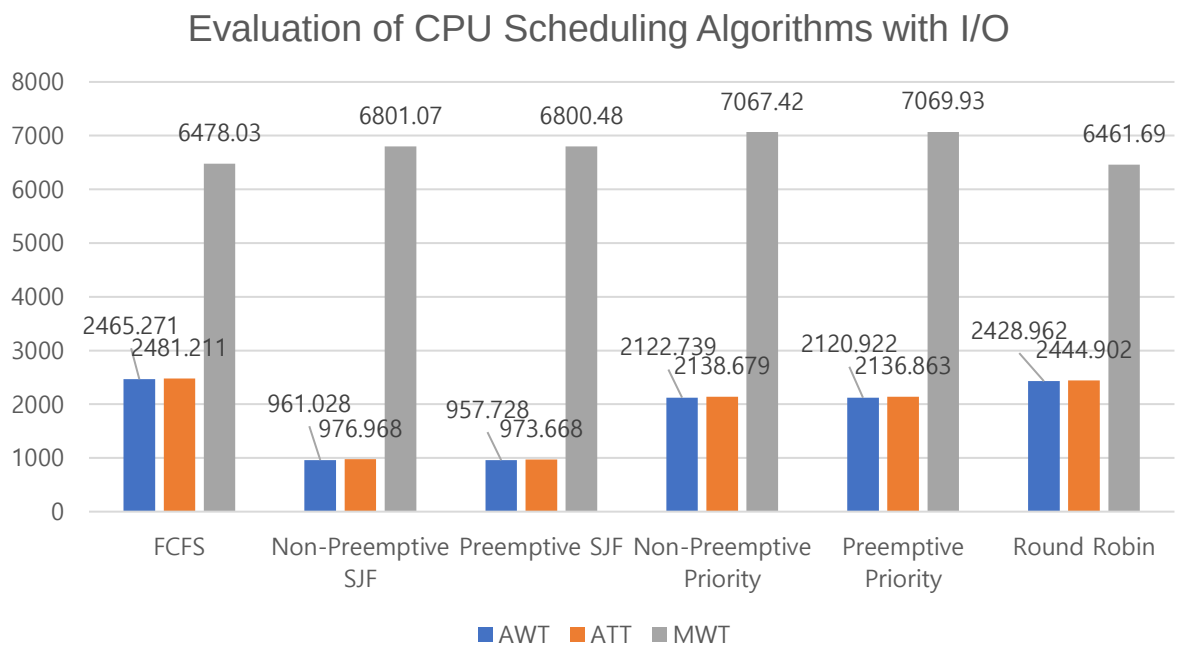

FIGURE 7
EVALUATION OF CPU SCHEDULING ALGORITHMS WITH I/O, TIME QUANTUM OF 6 IN RR

우선순위에 따라 곧바로 CPU 를 할당 받을 가능성이 크다. 해당 프로세스가 CPU 를 사용하고 있었던 것은 준비 큐에 있는 모든 프로세스 중 우선 순위가 가장 높았기 때문이다. 때문에 대기 시간이 크게 증가하지 않았다. 반면에 FCFS, Round Robin 알고리즘에서는 준비 큐에 먼저 도착한 프로세스에게 CPU 를 할당하기 때문에 I/O 를 처리하고 준비 큐로 돌아오면 먼저 도착했던 프로세스들을 기다려야 한다. 그러므로 두 알고리즘은 I/O 가 없었을 때보다 평균 대기 시간, 평균 반환 시간이 증가했다고 볼 수 있다. 최대 대기 시간은 FCFS 알고리즘을 제외하고는 큰 변화가 없었다. FCFS 알고리즘은 I/O 가 없었을 때보다 최대 대기 시간이 약 2200 증가했다. I/O 를 처리하기 위해 대기 큐에 갔다가 준비 큐로 이동하는 것이 Round Robin 알고리즘에서 Time Quantum 동안 CPU 를 사용한 프로세스가 준비 큐로 돌아가는 것과 유사하기 때문이다. 따라서 최대 대기 시간이 Round Robin 알고리즘과 비슷한 수준으로 증가한 것이라고 추측할 수 있다.

CONCLUSION

본 프로젝트를 수행하면서 CPU 스케줄링 알고리즘에 대해서 학습하고 이해하는 시간을 가졌다. 시뮬레이터를 직접 구현을 하는 과정에서 이해가 부족한 부분을 발견하고 보충할 수 있었다. 이번 프로젝트가 스케줄링한 결과를 구하기 위해 알고리즘 문제를 푸는 것이 아니라 시뮬레이터를 구현하는 것이었기 때문에 C 언어지만 모듈별로 객체 지향적인 느낌의 코드를 작성했다. 그리고 프로젝트를 통해서 자료구조를 구현하는 능력도 발전했음을 느꼈다. 모든 자료구조를 동적으로 관리하기 위해 FIFO 큐는 연결 리스트로 구현했고, 힙은 처음으로 이진 트리를 이용해서 구현했는데 새로운 경험이었다. 내용적인 구현과 더불어 많이 신경 쓴 것은 코드의 가독성과 모듈화에 대한 부분이었다. 어떻게 하면 코드를 직관적이고 쉽게 읽을 수 있을지, 중복되는 코드를 피해서 함수로 모듈화를 할 것인지에 대해 구조적인 측면에서 많은 고민이 있었다. 프로젝트 내적인 부분으로는 제시된 여섯 가지 알고리즘 외 다른 스케줄링 알고리즘과 추가적인 기능에도 욕심을 내볼 걸 하는 생각이 들어

아쉬웠다. Linux 커널이 사용하는 스케줄링 알고리즘으로 Completely Fair Scheduling(CFS) 알고리즘이 있다. 준비 큐에서 기다리는 시간, I/O 를 처리하기 위해 대기 큐에서 기다리는 시간을 보상함으로써 모든 프로세스에게 동등한 비율로 CPU 를 할당하는 알고리즘이라고 한다. CFS 알고리즘을 구현하는 것을 고려해봤으나 CFS 는 레드 블랙 트리를 자료구조로 사용하기 때문에 자료구조를 구현하는 데에 추가적인 시간이 많이 소요될 것 같아 시도를 하지 않았다. 하지만 다음에 기회가 된다면 CFS 를 비롯해 다른 스케줄링 알고리즘도 관심을 가져보는 기회가 있었으면 좋겠다.

REFERENCES

S. Suranauwarat, "A CPU Scheduling Algorithm Simulator", In Proc. of the 37th ASEE/IEEE Frontiers in Education Conference, Milwaukee, Wisconsin, USA, Session F2H pp.19-24, Oct.10-13, 2007

APPENDIX – SOURCE CODE

main.c

```
/**
 * @file main.c
 * @brief main function
 */
#include <stdio.h>
#include "cpu_scheduling_simulator.h"

int main() {
  int N;
  Simulator_t *simulator[7];

  Simulator_Init(&simulator[0], FCFS);
  Simulator_Init(&simulator[1], NON_PREEMPTIVE_SJF);
  Simulator_Init(&simulator[2], PREEMPTIVE_SJF);
  Simulator_Init(&simulator[3], NON_PREEMPTIVE_PRIORITY);
  Simulator_Init(&simulator[4], PREEMPTIVE_PRIORITY);
  Simulator_Init(&simulator[5], ROUND_ROBIN);
  simulator[6] = NULL;

  printf("Enter the number of processes: ");
  scanf("%d", &N);

  Simulator_GenerateProcess(simulator, N);

  for (int i = 0; i < 6; i++) Simulator_Start(simulator[i]);

  Simulator_Terminate(simulator);

  return 0;
}
```

cpu_scheduling_simulator.h

```
/**
 * @file cpu_scheduling_simulator.h
 * @brief struct definitions
 *       and function prototypes of simulator, queue, priority queue
 */
#ifndef __CPU_SCHEDULING_SIM_H_
#define __CPU_SCHEDULING_SIM_H_

typedef unsigned int process_id_t;
typedef unsigned int time_len_t;
typedef double avg_time_len_t;

/* Struct that represents a process */
typedef struct Process_t {
  process_id_t pid;         /* unique identifier of a process */
  time_len_t cpu_burst_time; /* time length to be processed by CPU */
  time_len_t io_burst_time;  /* time length to be processed by I/O */
  time_len_t arrival_time;   /* time at which process has created */
  int priority;            /* the smaller value is, the higher priority is */

  time_len_t waiting_time;    /* waiting time of process */
  time_len_t turnaround_time; /* turnaround time of process */
} Process_t, *ProcessPtr_t;
```

```
/* Queue Implementation */
typedef struct ListNode {
  ProcessPtr_t process_ptr; /* pointer of a process */
  struct ListNode *next;   /* pointer used for singly linked list */
} ListNode_t, *ListNodePtr_t;

typedef struct Queue_t {
  int count;
  ListNodePtr_t front, rear;
} Queue_t;

int Queue_Init(Queue_t **queue);
int Queue_Enqueue(Queue_t *queue, ProcessPtr_t process);
int Queue_Dequeue(Queue_t *queue);
ProcessPtr_t Queue_Front(Queue_t *queue);
int Queue_IsEmpty(Queue_t *queue);

/* Priority Queue Implementation */
typedef struct TreeNode {
  ProcessPtr_t process_ptr;             /* pointer of a process */
  struct TreeNode *left, *right, *parent; /* pointer for binary tree */
} TreeNode_t, *TreeNodePtr_t;

typedef struct Priority_Queue_t {
  int count;
  TreeNodePtr_t top;
  int (*compare)(ProcessPtr_t a, ProcessPtr_t b); /* compare function pointer */
} Priority_Queue_t;

int Priority_Init(Priority_Queue_t **queue,
               int (*compare)(ProcessPtr_t a, ProcessPtr_t b));
int Priority_Enqueue(Priority_Queue_t *queue, ProcessPtr_t process);
int Priority_Dequeue(Priority_Queue_t *queue);
ProcessPtr_t Priority_Top(Priority_Queue_t *queue);
int Priority_IsEmpty(Priority_Queue_t *queue);
int SJF_Compare(ProcessPtr_t a, ProcessPtr_t b);
int Priority_Compare(ProcessPtr_t a, ProcessPtr_t b);
int IO_Burst_Compare(ProcessPtr_t a, ProcessPtr_t b);

/* Flag values of scheduling algorithms */
#define FCFS 1
#define NON_PREEMPTIVE_SJF 2
#define PREEMPTIVE_SJF 3
#define NON_PREEMPTIVE_PRIORITY 4
#define PREEMPTIVE_PRIORITY 5
#define ROUND_ROBIN 6

/* Struct that represents a simulator */
typedef struct Simulator {
  unsigned int num_process;           /* the number of processes generated */
  Priority_Queue_t *generated_processes; /* pointer of job queue */
  Priority_Queue_t *terminated_processes; /* queue of terminated processes */

  int flag;                     /* flag of scheduling algorithm */
  ProcessPtr_t cur_cpu_burst;     /* pointer of currently running process */
  void *ready_queue;             /* pointer of ready queue */
  Priority_Queue_t *waiting_queue; /* pointer of waiting queue */

  time_len_t elapsed_time;          /* elapsed time in simulation */
```

```
  time_len_t idle_time;             /* total amount of cpu idle time */
  avg_time_len_t avg_waiting_time;    /* average waiting time */
  avg_time_len_t avg_turnaround_time; /* average turnaround time */
  time_len_t max_waiting_time;
} Simulator_t;

int Simulator_Init(Simulator_t **simulator, int flag);
int Simulator_GenerateProcess(Simulator_t **simulators, int N);
int Simulator_ArrivalTime_Compare(ProcessPtr_t a, ProcessPtr_t b);

void Simulator_Start(Simulator_t *simulator);

void Simulator_LoadReadyQueue(Simulator_t *simulator);
void Simulator_LoadReadyQueue_Priority(Simulator_t *simulator);

void Simulator_LoadProcess(Simulator_t *simulator);
void Simulator_LoadProcess_Priority(Simulator_t *simulator);

int Simulator_CPU_Burst(Simulator_t *simulator);
int Simulator_CPU_Burst_Preemptive(Simulator_t *simulator);
int Simulator_CPU_Burst_RR(Simulator_t *simulator, int *time_quantum);

void Simulator_Queue_Waiting(Queue_t *ready_queue);
void Simulator_Priority_Waiting(TreeNodePtr_t waiting_process);

void Simulator_Process_IO(TreeNodePtr_t io_process);
void Simulator_Process_IO_Queue(Simulator_t *simulator);
void Simulator_Process_IO_Priority(Simulator_t *simulator);

int Simulator_Probability();

void Simulator_FCFS(Simulator_t *simulator);
void Simulator_NonPreemptiveSJF(Simulator_t *simulator);
void Simulator_PreemptiveSJF(Simulator_t *simulator);
void Simulator_NonPreemptivePriority(Simulator_t *simulator);
void Simulator_PreemptivePriority(Simulator_t *simulator);
void Simulator_RoundRobin(Simulator_t *simulator);

void Simulator_Eval(Simulator_t *simulator);
void Simulator_Terminate(Simulator_t **simulators);

#endif
```

 # queue.c

```
/**
 * @file queue.c
 * @brief function implementations of queue
 *       nodes in queue managed by singly linked list.
 */
#include <stdio.h>
#include <stdlib.h>
#include "cpu_scheduling_simulator.h"

/**
 * @fn int Queue_Init(Queue_t **queue)
 * @brief initialize variables in Queue_t
 *
 * @param queue pointer of Queue_t *(double pointer)
 *
```

```
 * @return 1 if initialization successes
 */
int Queue_Init(Queue_t **queue) {
  *queue = (Queue_t *)malloc(sizeof(Queue_t));

  if ((*queue) == NULL) {
    printf("Error log: malloc() in Queue_Init\n");
    exit(-1);
  }

  (*queue)->count = 0;
  (*queue)->front = (*queue)->rear = NULL;

  return 1;
}

/**
 * @fn int Queue_Enqueue(Queue_t *queue, ProcessPtr_t process)
 * @brief insert a new node in queue
 *
 * @param queue pointer of Queue_t
 * @param process ProcessPtr_t of the new node
 *
 * @return 1 if enqueue successes
 */
int Queue_Enqueue(Queue_t *queue, ProcessPtr_t process) {
  ListNodePtr_t new = (ListNodePtr_t)malloc(sizeof(ListNode_t));

  if (new == NULL) {
    printf("Error log: malloc() in Queue_Enqueue\n");
    exit(-1);
  }

  new->process_ptr = process;
  new->next = NULL;

  if (queue->count++ == 0) {
    /* The inserted node is the only node in queue. */
    queue->front = queue->rear = new;
  } else {
    queue->rear->next = new;
    queue->rear = new;
  }

  return 1;
}

/**
 * @fn int Queue_Dequeue(Queue_t *queue)
 * @brief delete a node in queue
 *
 * @param queue pointer of Queue_t
 *
 * @return 1 if dequeue successes
 */
int Queue_Dequeue(Queue_t *queue) {
  if (queue->count == 0) return 0;

  ListNodePtr_t tmp = queue->front;
```

```
  queue->front = tmp->next;
  free(tmp);

  /* There is no node in queue after deletion. */
  if (--queue->count == 0) queue->rear = NULL;

  return 1;
}

/**
 * @fn int Queue_Front(Queue_t *queue)
 * @brief return the pointer of a process that is at the front of the queue
 *
 * @param queue pointer of Queue_t
 *
 * @return the pointer of a process if queue is not empty
 *        NULL                  if queue is empty
 */
ProcessPtr_t Queue_Front(Queue_t *queue) {
  if (queue->count == 0) return NULL;

  return queue->front->process_ptr;
}

/**
 * @fn int Queue_IsEmpty(Queue_t queue)
 * @brief return whether the queue is empty
 *
 * @param queue pointer of Queue_t
 *
 * @return 1 if the queue is empty
 *        0 otherwise
 */
int Queue_IsEmpty(Queue_t *queue) { return queue->count == 0; }
```

 # priority_queue.c

```
/**
 * @file priority_queue.c
 * @brief function implementations of priority queue
 *       nodes in queue managed by binary tree.
 */
#include <stdio.h>
#include <stdlib.h>
#include "cpu_scheduling_simulator.h"

#define SWAP(a, b)                \
  ProcessPtr_t tmp = a->process_ptr; \
  a->process_ptr = b->process_ptr;   \
  b->process_ptr = tmp;
#define parent(x) x->parent

/**
 * @fn int Priority_Init(Priority_Queue_t **queue,
 *                    int (*compare)(ProcessPtr a, ProcessPtr b))
 * @brief initialize variables in Priority_Queue_t
 *
 * @param queue pointer of Priority_Queue_t *(double pointer)
 * @param compare function pointer used for constructing heap
 *
```

```
 * @return 1 if initialization successes
 */
int Priority_Init(Priority_Queue_t **queue,
                  int (*compare)(ProcessPtr_t a, ProcessPtr_t b)) {
  *queue = (Priority_Queue_t *)malloc(sizeof(Priority_Queue_t));

  if ((*queue) == NULL) {
    printf("Error log: malloc() in Priority_Init\n");
    exit(-1);
  }

  (*queue)->count = 0;
  (*queue)->top = NULL;
  (*queue)->compare = compare;

  return 1;
}

/**
 * @fn int Priority_Enqueue(Priority_Queue_t *queue, ProcessPtr_t process)
 * @brief insert a new node in queue
 *
 * @param queue pointer of Priority_Queue_t
 * @param process ProcessPtr_t of the new node
 *
 * @return 1 if enqueue successes
 */
int Priority_Enqueue(Priority_Queue_t *queue, ProcessPtr_t process) {
  int pos; /* varaible used for finding the place at which insert the node */
  TreeNodePtr_t walker;
  TreeNodePtr_t new = (TreeNodePtr_t)malloc(sizeof(TreeNode_t));

  if (new == NULL) {
    printf("Error log: malloc() in Priority_Enqueue\n");
    exit(-1);
  }

  new->process_ptr = process;
  new->left = new->right = new->parent = NULL;

  /* The inserted node is the first node in queue. */
  if (queue->count++ == 0) {
    queue->top = new;

    return 1;
  }

  /* Find the place at which insert the new node */
  walker = queue->top;
  for (pos = 2; !(pos <= queue->count && queue->count < (pos << 1));
       pos = pos << 1)
    ;
  pos = pos >> 1;

  while (pos >= 2) {
    if (pos & queue->count)
      walker = walker->right;
    else
      walker = walker->left;
```

```
    pos = pos >> 1;
  }

  /* Insert the new node in heap */
  if (pos & queue->count) {
    walker->right = new;
    new->parent = walker;
  } else {
    walker->left = new;
    new->parent = walker;
  }

  /* Heap Adjustments */
  walker = new;
  while (walker->parent != NULL) {
    if (queue->compare(walker->process_ptr, parent(walker)->process_ptr)) {
      SWAP(walker, parent(walker));
      walker = walker->parent;
    } else {
      break;
    }
  }

  return 1;
}

/**
 * @fn int Priority_Dequeue(Priority_Queue_t *queue)
 * @brief delete a node in queue
 *
 * @param queue pointer of Queue_t
 *
 * @return 1 if dequeue successes
 */
int Priority_Dequeue(Priority_Queue_t *queue) {
  int pos; /* varaible used for finding the place from which delete the node */
  TreeNodePtr_t walker;

  if (queue->count == 0) return 0;

  if (queue->count == 1) {
    free(queue->top);
    queue->count--;
    queue->top = NULL;

    return 1;
  }

  /* Find the last node in heap */
  walker = queue->top;
  for (pos = 2; !(pos <= queue->count && queue->count < (pos << 1));
       pos = pos << 1)
    ;
  pos = pos >> 1;

  while (pos >= 2) {
    if (queue->count & pos)
      walker = walker->right;
```

```
    else
      walker = walker->left;

    pos = pos >> 1;
  }

  /* Delete the last node */
  if (pos & queue->count) {
    queue->top->process_ptr = walker->right->process_ptr;
    free(walker->right);
    walker->right = NULL;
  } else {
    queue->top->process_ptr = walker->left->process_ptr;
    free(walker->left);
    walker->left = NULL;
  }

  /* Heap Adjustments */
  queue->count--;
  walker = queue->top;
  while (1) {
    /* Walker node is leaf node */
    if (walker->left == NULL && walker->right == NULL) break;

    TreeNodePtr_t child = walker->right
                        ? queue->compare(walker->left->process_ptr,
                                         walker->right->process_ptr)
                              ? walker->left
                              : walker->right
                        : walker->left;
    if (queue->compare(child->process_ptr, walker->process_ptr)) {
      SWAP(child, walker);
      walker = child;
    } else {
      break;
    }
  }

  return 1;
}

/**
 * @fn int Priority_Top(Priority_Queue_t *queue)
 * @brief return the pointer of a process that is at the top of the queue
 *
 * @param queue pointer of Priority_Queue_t
 *
 * @return the pointer of a process if queue is not empty
 *         NULL                  if queue is empty
 */
ProcessPtr_t Priority_Top(Priority_Queue_t *queue) {
  if (queue->count == 0) return NULL;

  return queue->top->process_ptr;
}

/**
 * @fn int Priority_IsEmpty(Priority_Queue_t queue)
 * @brief return whether the queue is empty
```

```
 *
 * @param queue pointer of Priority_Queue_t
 *
 * @return 1 if the queue is empty
 *         0 otherwise
 */
int Priority_IsEmpty(Priority_Queue_t *queue) { return queue->count == 0; }

/**
 * @fn int SJF_Compare(ProcessPtr_t a, ProcessPtr_t b)
 * @brief compare cpu burst time of two processes
 *
 * @param a first process to compare
 * @param b second process to compare
 *
 * @return 1 if cpu burst time of the first process is shorter
 *         0 otherwise
 */
int SJF_Compare(ProcessPtr_t a, ProcessPtr_t b) {
  return a->cpu_burst_time != b->cpu_burst_time
           ? a->cpu_burst_time < b->cpu_burst_time
           : a->pid < b->pid;
}

/**
 * @fn int Priority_Compare(ProcessPtr_t a, ProcessPtr_t b)
 * @brief compare priority of two processes
 *
 * @param a first process to compare
 * @param b second process to compare
 *
 * @return 1 if priority of the first process is higher
 *         0 otherwise
 */
int Priority_Compare(ProcessPtr_t a, ProcessPtr_t b) {
  return a->priority != b->priority ? a->priority < b->priority
                                    : a->pid < b->pid;
}

/**
 * @fn int IO_Burst_Compare(ProcessPtr_t a, ProcessPtr_t b)
 * @brief compare I/O burst time of two processes
 *
 * @param a first process to compare
 * @param b second process to compare
 *
 * @return 1 if I/O burst time of the first process is shorter
 *         0 otherwise
 */
int IO_Burst_Compare(ProcessPtr_t a, ProcessPtr_t b) {
  return a->io_burst_time != b->io_burst_time
           ? a->io_burst_time < b->io_burst_time
           : a->pid < b->pid;
}
```

 # simulator.c

```
/**
 * @file simulator.c
 * @brief function implementations of simulator
```

```
 */
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include "cpu_scheduling_simulator.h"

#define cur_process(x) x->cur_cpu_burst

/* when defined, I/O burst time of all processes are set to 0 */
//#define IO_DEBUG
/* when defined, arrival time of all processes are set to 0 */
// #define ARRIVAL_DEBUG
#define TIME_QUANTUM 10 /* time quantum value used in RR */
#define SIM_SPEED 1    /* speed that displays gantt chart */
#define CUTTER 10      /* the number of blocks in each line of gantt chart */

/**
 * @fn int Simulator_Init(Simulator **simulator, int flag)
 * @brief initialize variables in Simulator_t
 *
 * @param simulator pointer of Simulator *(double pointer)
 * @param flag value that indicates what scheduling algorithm simulator uses
 *
 * @return 1 if initialization successes
 */
int Simulator_Init(Simulator_t **simulator, int flag) {
  *simulator = (Simulator_t *)malloc(sizeof(Simulator_t));

  if (*simulator == NULL) {
    printf("Error log: malloc() in Simulator_Init\n");
    exit(-1);
  }

  /* Initialize variables in simulator */
  (*simulator)->num_process = 0;
  (*simulator)->cur_cpu_burst = NULL;
  (*simulator)->elapsed_time = 0;
  (*simulator)->idle_time = 0;
  (*simulator)->avg_waiting_time = 0;
  (*simulator)->avg_turnaround_time = 0;
  (*simulator)->max_waiting_time = 0;

  /*
   * Allocate data structure of ready queue(normal queue or priority queue)
   * according to given flag parameter
   */
  switch (flag) {
    case FCFS:
    case ROUND_ROBIN:
      Queue_Init((Queue_t **)&((*simulator)->ready_queue));
      break;
    case NON_PREEMPTIVE_SJF:
    case PREEMPTIVE_SJF:
      Priority_Init((Priority_Queue_t **)&((*simulator)->ready_queue),
                    SJF_Compare);
      break;
    case NON_PREEMPTIVE_PRIORITY:
    case PREEMPTIVE_PRIORITY:
      Priority_Init((Priority_Queue_t **)&((*simulator)->ready_queue),
```

```
                Priority_Compare);
      break;
    default:
      printf("Error log: invalid flag parameter value\n");
      exit(-1);
  }

  /* Allocate memory for waiting queue and initialize */
  Priority_Init(&(*simulator)->waiting_queue, IO_Burst_Compare);

  /* Allocate memory for generated processes and initialize */
  Priority_Init(&(*simulator)->generated_processes,
              Simulator_ArrivalTime_Compare);

  /* Allocate memory for queue of terminated processes and initialize */
  Priority_Init(&(*simulator)->terminated_processes,
              Simulator_ArrivalTime_Compare);

  /* Set scheduling algorithm flag */
  (*simulator)->flag = flag;

  return 1;
}

/**
 * @fn int Simulator_GenerateProcess(Simulator_t **simulators, int N)
 * @brief generate random processes and clone them into each simulator
 *       in simulators
 *
 * @param simulators array of Simulator_t *, which ends with NULL pointer
 * @param N the number of generated processes
 *
 * @return 1 if generation successes
 */
int Simulator_GenerateProcess(Simulator_t **simulators, int N) {
  unsigned char data[500]; /* array of random cpu burst time values */
  ProcessPtr_t new;       /* pointer of newly generated process */
  process_id_t pid;
  time_len_t cpu_burst_time;
  time_len_t io_burst_time;
  time_len_t arrival_time;
  int priority;

  srand(time(NULL)); /* seed for rand() */

  /*
   * Generate random pool of cpu burst time values
   * which follows the distribution similar to geometric distribution
   */
  for (int i = 0; i < 500; i++) {
    if (i < 450)
      data[i] = rand() % 10 + 1; /* 1 ~ 10, prob 0.9 */
    else if (i < 475)
      data[i] = rand() % 10 + 11; /* 11 ~ 20, prob 0.05 */
    else
      data[i] = rand() % 20 + 21; /* 21~ 40, prob 0.05 */
  }

  printf(
```

```
    "++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++"
    "++++++++++"
    "\n");
  printf(
    "++  PID  ++  CPU_BURST_TIME  ++  IO_BURST_TIME  ++  ARRIVAL_TIME  ++  "
    "PRIORITY  "
    "++\n");
  printf(
    "++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++"
    "++++++++++"
    "\n");

  /*
   * Generate random processes
   * and clone them into each simulator
   */
  for (int i = 0; simulators[i]; i++) simulators[i]->num_process = N;

  for (int i = 0; i < N; i++) {
    pid = i + 1;
    cpu_burst_time = data[rand() % 500]; /* pick cpu burst time value */

#ifdef IO_DEBUG
    io_burst_time = 0;
#else
    io_burst_time = rand() % 20;
#endif

#ifdef ARRIVAL_DEBUG
    arrival_time = 0;
#else
    arrival_time = rand() % (N * 3);
#endif

    priority = rand() % 41 - 20; /* -20 <= priority <= 20 */

    printf("++ %5d ++  %14d  ++  %13d  ++  %12d  ++  %8d  ++\n", pid,
          cpu_burst_time, io_burst_time, arrival_time, priority);

    for (int j = 0; simulators[j]; j++) {
      new = (ProcessPtr_t)malloc(sizeof(Process_t));

      if (new == NULL) {
        printf("Error log: malloc() in Simulator_GenerateProcess\n");
        exit(-1);
      }

      new->pid = pid;
      new->cpu_burst_time = cpu_burst_time;
      new->io_burst_time = io_burst_time;
      new->arrival_time = arrival_time;
      new->priority = priority;
      new->waiting_time = 0;
      new->turnaround_time = 0;

      Priority_Enqueue(simulators[j]->generated_processes, new);
    }
  }
  printf(
```

```
    "+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++"
    "++++++++++++\n");

  return 1;
}

/**
 * @fn int Simulator_ArrivalTime_Compare
 * @brief compare arrival time of two processes
 *
 * @param a first process to compare
 * @param b second process to compare
 *
 * @return 1 if arrival time of the first process is smaller
 *        0 otherwise
 */
int Simulator_ArrivalTime_Compare(ProcessPtr_t a, ProcessPtr_t b) {
  return a->arrival_time != b->arrival_time ? a->arrival_time < b->arrival_time
                                      : a->pid < b->pid;
}

/**
 * @fn void Simulator_Start(Simulator_t *simulator)
 * @brief call the simulation function
 *
 * @param simulator pointer of Simulator_t
 */
void Simulator_Start(Simulator_t *simulator) {
  switch (simulator->flag) {
    case FCFS:
      Simulator_FCFS(simulator);
      break;
    case NON_PREEMPTIVE_SJF:
      Simulator_NonPreemptiveSJF(simulator);
      break;
    case PREEMPTIVE_SJF:
      Simulator_PreemptiveSJF(simulator);
      break;
    case NON_PREEMPTIVE_PRIORITY:
      Simulator_NonPreemptivePriority(simulator);
      break;
    case PREEMPTIVE_PRIORITY:
      Simulator_PreemptivePriority(simulator);
      break;
    case ROUND_ROBIN:
      Simulator_RoundRobin(simulator);
      break;
    default:
      printf("Error log: invalid flag parameter value\n");
      exit(-1);
  }

  return;
}

/**
 * @fn void Simulator_LoadReadyQueue(Simulator_t *simulator)
 * @brief take processes from job queue(or waiting queue)
 *       and put them in ready queue
```

```
 *
 * @param simulator pointer of Simulator_t
 */
void Simulator_LoadReadyQueue(Simulator_t *simulator) {
  Queue_t *ready_queue = simulator->ready_queue;
  Priority_Queue_t *waiting_queue = simulator->waiting_queue;
  Priority_Queue_t *job_queue = simulator->generated_processes;
  Priority_Queue_t *terminated_processes = simulator->terminated_processes;
  ProcessPtr_t front_process;

  /*
   * Represent when process arrives at ready_queue
   * If elapsed time equals arrival time of the process in job queue,
   * pop the process from job queue and insert it in ready queue
   */
  while (!Priority_IsEmpty(job_queue) &&
        (front_process = Priority_Top(job_queue))->arrival_time ==
           simulator->elapsed_time) {
    Queue_Enqueue(ready_queue, front_process);
    Priority_Dequeue(job_queue);
  }

  /*
   * Represenet I/O of the process completed
   * If I/O burst time of the process in waiting queue is 0,
   * pop the process from waiting queue and insert it in ready queue
   */
  while (!Priority_IsEmpty(waiting_queue) &&
        (front_process = Priority_Top(waiting_queue))->io_burst_time == 0) {
    Priority_Dequeue(waiting_queue);
    Queue_Enqueue(ready_queue, front_process);
  }

  return;
}

/**
 * @fn void Simulator_LoadReadyQueue_Priority(Simulator_t *simulator)
 * @brief take processes from job queue(or waiting queue)
 *       and put them in ready queue
 *
 * @param simulator pointer of Simulator_t
 */
void Simulator_LoadReadyQueue_Priority(Simulator_t *simulator) {
  Priority_Queue_t *ready_queue = simulator->ready_queue;
  Priority_Queue_t *waiting_queue = simulator->waiting_queue;
  Priority_Queue_t *job_queue = simulator->generated_processes;
  Priority_Queue_t *terminated_processes = simulator->terminated_processes;
  ProcessPtr_t front_process;

  /*
   * Represent when process arrives at ready_queue
   * If elapsed time equals arrival time of the process in job queue,
   * pop the process from job queue and insert it in ready queue
   */
  while (!Priority_IsEmpty(job_queue) &&
        (front_process = Priority_Top(job_queue))->arrival_time ==
           simulator->elapsed_time) {
    Priority_Enqueue(ready_queue, front_process);
```

```
    Priority_Dequeue(job_queue);
  }

  /*
   * Represenet I/O of the process completed
   * If I/O burst time of the process in waiting queue is 0,
   * pop the process from waiting queue and insert it in ready queue
   */
  while (!Priority_IsEmpty(waiting_queue) &&
         (front_process = Priority_Top(waiting_queue))->io_burst_time == 0) {
    Priority_Dequeue(waiting_queue);
    Priority_Enqueue(ready_queue, front_process);
  }

  return;
}

/**
 * @fn void Simulator_LoadProcess(Simulator_t *simulator)
 * @brief get a process from ready queue and allocate CPU
 *
 * @param simulator pointer of Simulator_t
 */
void Simulator_LoadProcess(Simulator_t *simulator) {
  Queue_t *ready_queue = simulator->ready_queue;
  Priority_Queue_t *waiting_queue = simulator->waiting_queue;

  /* Get a process from ready queue and allocate CPU to the process */
  if (cur_process(simulator) == NULL) {
    if (!Queue_IsEmpty(ready_queue)) {
      cur_process(simulator) = Queue_Front(ready_queue);
      Queue_Dequeue(ready_queue);
    }
  }

  return;
}

/**
 * @fn void Simulator_LoadProcess_Priority(Simulator_t *simulator)
 * @brief get a process from ready queue and allocate CPU
 *
 * @param simulator pointer of Simulator_t
 */
void Simulator_LoadProcess_Priority(Simulator_t *simulator) {
  Priority_Queue_t *ready_queue = simulator->ready_queue;
  Priority_Queue_t *waiting_queue = simulator->waiting_queue;

  /* Get a process from ready queue and allocate CPU to the process */
  if (cur_process(simulator) == NULL) {
    if (!Priority_IsEmpty(ready_queue)) {
      cur_process(simulator) = Priority_Top(ready_queue);
      Priority_Dequeue(ready_queue);
    }
  }

  return;
}
```

```
/**
 * @fn int Simulator_CPU_Burst(Simulator_t *simulator)
 * @brief running process is processed
 *       and display gantt chart
 *
 * @param simulator pointer of Simulator_t
 *
 * @return 1 if all processes terminated
 *        0 otherwise
 */
int Simulator_CPU_Burst(Simulator_t *simulator) {
  Priority_Queue_t *waiting_queue = simulator->waiting_queue;
  Priority_Queue_t *terminated_processes = simulator->terminated_processes;

  for (int i = 0; i < SIM_SPEED; i++)
    ;

  if (cur_process(simulator)) { /* running process exists */
    cur_process(simulator)->cpu_burst_time--;
    cur_process(simulator)->turnaround_time++;
    printf("[ %6d ]", cur_process(simulator)->pid); /* display gantt chart */

    if (cur_process(simulator)->cpu_burst_time) {
      if (cur_process(simulator)->io_burst_time) {
        if (cur_process(simulator)->cpu_burst_time == 1 ||
            Simulator_Probability()) {
          /*
           * cur_process(simulator) has both CPU and I/O works.
           * I/O request occured randomly
           * move cur_process(simulator) from CPU to waiting queue
           */
          Priority_Enqueue(waiting_queue, cur_process(simulator));
          cur_process(simulator) = NULL;
        }
      }
    } else { /* Process has completely processed, move to terminated_process */
      Priority_Enqueue(terminated_processes, cur_process(simulator));
      cur_process(simulator) = NULL;

      /* All processes terminated */
      if (terminated_processes->count == simulator->num_process) {
        printf("\n-> Simulation End.\n\n");
        Simulator_Eval(simulator);

        return 1;
      }
    }
  } else { /* CPU is in idle */
    simulator->idle_time++;
    printf("[  IDLE  ]");
  }

  /* Limit the number of blocks in each line of gantt chart */
  if ((simulator->elapsed_time + 1) % CUTTER == 0) {
    printf("\n");
  }

  return 0;
}
```

```
/**
 * @fn int Simulator_CPU_Burst_Preemptive(Simulator_t *simulator)
 * @brief running process is processed
 *       and display gantt chart
 *
 * @param simulator pointer of Simulator_t
 *
 * @return 1 if all processes terminated
 *        0 otherwise
 */
int Simulator_CPU_Burst_Preemptive(Simulator_t *simulator) {
  Priority_Queue_t *ready_queue = simulator->ready_queue;
  Priority_Queue_t *waiting_queue = simulator->waiting_queue;
  Priority_Queue_t *terminated_processes = simulator->terminated_processes;
  int (*compare)(ProcessPtr_t a, ProcessPtr_t b) = ready_queue->compare;

  for (int i = 0; i < SIM_SPEED; i++)
    ;

  /*
   * Find whether the process
   * by which currently running process can be preempted exists
   */
  if (cur_process(simulator) && !Priority_IsEmpty(ready_queue) &&
     compare(ready_queue->top->process_ptr, cur_process(simulator))) {
    /* Currently running process is preempted */
    Priority_Enqueue(ready_queue, cur_process(simulator));
    cur_process(simulator) = NULL;

    /* Allocate CPU to another process */
    Simulator_LoadProcess_Priority(simulator);
  }

  if (cur_process(simulator)) { /* running process exists */
    cur_process(simulator)->turnaround_time++;
    cur_process(simulator)->cpu_burst_time--;
    printf("[ %6d ]", cur_process(simulator)->pid); /* display gantt chart */

    if (cur_process(simulator)->cpu_burst_time) {
      if (cur_process(simulator)->io_burst_time) {
        if (cur_process(simulator)->cpu_burst_time == 1 ||
           Simulator_Probability()) {
          /*
           * cur_process(simulator) has both CPU and I/O works.
           * I/O request occured randomly
           * move cur_process(simulator) from CPU to waiting queue
           */
          Priority_Enqueue(waiting_queue, cur_process(simulator));
          cur_process(simulator) = NULL;
        }
      }
    } else { /* Process has completely processed, move to terminated_process */
      Priority_Enqueue(terminated_processes, cur_process(simulator));
      cur_process(simulator) = NULL;

      /* All processes terminated */
      if (terminated_processes->count == simulator->num_process) {
        printf("\n-> Simulation End.\n\n");
```

```
      Simulator_Eval(simulator);

      return 1;
    }
   }
  } else { /* CPU is in idle */
   simulator->idle_time++;
   printf("[  IDLE  ]");
  }

  /* Limit the number of blocks in each line of gantt chart */
  if ((simulator->elapsed_time + 1) % CUTTER == 0) {
   printf("\n");
  }

  return 0;
}

/**
 * @fn int Simulator_CPU_Burst_RR(Simulator_t *simulator)
 * @brief running process is processed
 *       and display gantt chart
 *
 * @param simulator pointer of Simulator_t
 *
 * @return 1 if all processes terminated
 *        0 otherwise
 */
int Simulator_CPU_Burst_RR(Simulator_t *simulator, int *time_quantum) {
 Queue_t *ready_queue = simulator->ready_queue;
 Priority_Queue_t *waiting_queue = simulator->waiting_queue;
 Priority_Queue_t *terminated_processes = simulator->terminated_processes;

 for (int i = 0; i < SIM_SPEED; i++)
   ;

 if (cur_process(simulator)) { /* running process exists */
   cur_process(simulator)->cpu_burst_time--;
   cur_process(simulator)->turnaround_time++;
   printf("[ %6d ]", cur_process(simulator)->pid); /* display gantt chart */
   (*time_quantum)++;

   if (cur_process(simulator)->cpu_burst_time) {
     if (cur_process(simulator)->io_burst_time) {
       if (cur_process(simulator)->cpu_burst_time == 1 ||
          Simulator_Probability()) {
         /*
          * cur_process(simulator) has both CPU and I/O works.
          * I/O request occured randomly
          * move cur_process(simulator) from CPU to waiting queue
          */
         Priority_Enqueue(waiting_queue, cur_process(simulator));
         cur_process(simulator) = NULL;
         *time_quantum = 0;
       }
     }

     if (*time_quantum == TIME_QUANTUM) {
       Queue_Enqueue(ready_queue, cur_process(simulator));
```

```
      cur_process(simulator) = NULL;
      *time_quantum = 0;
    }
   } else { /* Process has completely processed, move to terminated_process */
    Priority_Enqueue(terminated_processes, cur_process(simulator));
    cur_process(simulator) = NULL;
    *time_quantum = 0;

    /* All processes terminated */
    if (terminated_processes->count == simulator->num_process) {
     printf("\n-> Simulation End.\n\n");
     Simulator_Eval(simulator);

     return 1;
    }
   }
  } else { /* CPU is in idle */
   simulator->idle_time++;
   printf("[  IDLE  ]");
  }

  /* Limit the number of blocks in each line of gantt chart */
  if ((simulator->elapsed_time + 1) % CUTTER == 0) {
   printf("\n");
  }

  return 0;
}

/**
 * @fn void Simulator_Queue_Waiting(Queue_t *ready_queue)
 * @brief both waiting time and turnaround time of all processes in ready
 * queue are increased by 1
 *
 * @param ready_queue pointer of Queue_t
 */
void Simulator_Queue_Waiting(Queue_t *ready_queue) {
 ListNodePtr_t tmp = ready_queue->front;

 if (tmp == NULL) return;

 do {
  tmp->process_ptr->waiting_time++;
  tmp->process_ptr->turnaround_time++;
 } while (tmp = tmp->next);

 return;
}

/**
 * @fn void Simulator_Priority_Waiting(TreeNodePtr_t waiting_process)
 * @brief both waiting time and turnaround time of all processes in ready
 * queue are increased by 1
 *
 * @param waiting_process pointer of tree node in ready queue
 */
void Simulator_Priority_Waiting(TreeNodePtr_t waiting_process) {
 if (waiting_process == NULL) return;
```

```
  Simulator_Priority_Waiting(waiting_process->left);
  waiting_process->process_ptr->waiting_time++;
  waiting_process->process_ptr->turnaround_time++;
  Simulator_Priority_Waiting(waiting_process->right);

  return;
}

/**
 * @fn void Simulator_Process_IO(TreeNodePtr_t io_process)
 * @brief I/O burst time of all processes in waiting queue
 *       are decremented by 1
 *       turnaround time of all process in waiting queue
 *       are increased by 1
 *
 * @param io_process pointer of tree node in waiting_queue
 */
void Simulator_Process_IO(TreeNodePtr_t io_process) {
  if (io_process == NULL) return;

  Simulator_Process_IO(io_process->left);
  io_process->process_ptr->io_burst_time--;
  io_process->process_ptr->turnaround_time++;
  Simulator_Process_IO(io_process->right);

  return;
}

/**
 * @fn void Simulator_Process_IO_Queue(Simulator_t *simulator)
 * @brief represents randomly completion of I/O works
 *       all processes in waiting queue randomly make I/O interrupts.
 *
 * @param simulator pointer of Simulator_t
 */
void Simulator_Process_IO_Queue(Simulator_t *simulator) {
  Queue_t *ready_queue = simulator->ready_queue;
  Priority_Queue_t *waiting_queue = simulator->waiting_queue;
  Priority_Queue_t *tmp;
  ProcessPtr_t top_process;

  if (Priority_IsEmpty(waiting_queue)) return;

  Priority_Init(&tmp, IO_Burst_Compare);

  while (top_process = Priority_Top(waiting_queue)) {
    if (Simulator_Probability() && top_process->cpu_burst_time > 1) {
      Priority_Dequeue(waiting_queue);
      Queue_Enqueue(ready_queue, top_process);
    } else {
      Priority_Dequeue(waiting_queue);
      Priority_Enqueue(tmp, top_process);
    }
  }

  simulator->waiting_queue = tmp;
  free(waiting_queue);

  return;
```

```
}

/**
 * @fn void Simulator_Process_IO_Priority(Simulator_t *simulator)
 * @brief represents randomly completion of I/O works
 *       all processes in waiting queue randomly make I/O interrupts.
 *
 * @param simulator pointer of Simulator_t
 */
void Simulator_Process_IO_Priority(Simulator_t *simulator) {
  Priority_Queue_t *ready_queue = simulator->ready_queue;
  Priority_Queue_t *waiting_queue = simulator->waiting_queue;
  Priority_Queue_t *tmp;
  ProcessPtr_t top_process;

  if (Priority_IsEmpty(waiting_queue)) return;

  Priority_Init(&tmp, IO_Burst_Compare);

  while (top_process = Priority_Top(waiting_queue)) {
    if (Simulator_Probability() && top_process->cpu_burst_time > 1) {
      Priority_Dequeue(waiting_queue);
      Priority_Enqueue(ready_queue, top_process);
    } else {
      Priority_Dequeue(waiting_queue);
      Priority_Enqueue(tmp, top_process);
    }
  }

  simulator->waiting_queue = tmp;
  free(waiting_queue);

  return;
}

/**
 * @fn int Simulator_Probability()
 * @brief return 1 or 0
 *
 * @return 1 for probability 0.53
 *        0 for probability 0.47
 */
int Simulator_Probability() {
  int N = 0;

  for (int i = 0; i < 100; i++) {
    if (rand() % 2) {
      N++;
    }
  }

  return N >= 50;
}

/**
 * @fn void Simulator_FCFS(Simulator_t *simulator)
 * @brief simulate CPU scheduling with FCFS algorithm
 *
 * @param simulator pointer of Simulator_t
```

```
 */
void Simulator_FCFS(Simulator_t *simulator) {
  Queue_t *ready_queue = simulator->ready_queue;

  printf("\n# FCFS Algorithm\n\n");
  while (1) {
    Simulator_LoadReadyQueue(simulator);

    Simulator_LoadProcess(simulator);
    Simulator_Queue_Waiting(ready_queue);
    Simulator_Process_IO(simulator->waiting_queue->top);
    Simulator_Process_IO_Queue(simulator);

    if (Simulator_CPU_Burst(simulator)) return;

    simulator->elapsed_time++;
  }
}

/**
 * @fn void Simulator_NonPreemptiveSJF(Simulator_t *simulator)
 * @brief simulate CPU scheduling with Non-Preemptive SJF algorithm
 *
 * @param simulator pointer of Simulator_t
 */
void Simulator_NonPreemptiveSJF(Simulator_t *simulator) {
  Priority_Queue_t *ready_queue = simulator->ready_queue;

  printf("\n# Non-Preemptive SJF Algorithm\n\n");
  while (1) {
    Simulator_LoadReadyQueue_Priority(simulator);

    Simulator_LoadProcess_Priority(simulator);
    Simulator_Priority_Waiting(ready_queue->top);
    Simulator_Process_IO(simulator->waiting_queue->top);
    Simulator_Process_IO_Priority(simulator);

    if (Simulator_CPU_Burst(simulator)) return;

    simulator->elapsed_time++;
  }
  return;
}

/**
 * @fn void Simulator_PreemptiveSJF(Simulator_t *simulator)
 * @brief simulate CPU scheduling with Preemptive SJF algorithm
 *
 * @param simulator pointer of Simulator_t
 */
void Simulator_PreemptiveSJF(Simulator_t *simulator) {
  Priority_Queue_t *ready_queue = simulator->ready_queue;

  printf("\n# Preemptive SJF Algorithm\n\n");
  while (1) {
    Simulator_LoadReadyQueue_Priority(simulator);

    Simulator_LoadProcess_Priority(simulator);
    Simulator_Priority_Waiting(ready_queue->top);
```

```
    Simulator_Process_IO(simulator->waiting_queue->top);
    Simulator_Process_IO_Priority(simulator);

    if (Simulator_CPU_Burst_Preemptive(simulator)) return;

    simulator->elapsed_time++;
  }
  return;
}

/**
 * @fn void Simulator_NonPreemptivePriority(Simulator_t *simulator)
 * @brief simulate CPU scheduling with Non-Preemptive Priority algorithm
 *
 * @param simulator pointer of Simulator_t
 */
void Simulator_NonPreemptivePriority(Simulator_t *simulator) {
  Priority_Queue_t *ready_queue = simulator->ready_queue;

  printf("\n# Non-Preemptive Priority Algorithm\n\n");
  while (1) {
    Simulator_LoadReadyQueue_Priority(simulator);

    Simulator_LoadProcess_Priority(simulator);
    Simulator_Priority_Waiting(ready_queue->top);
    Simulator_Process_IO(simulator->waiting_queue->top);
    Simulator_Process_IO_Priority(simulator);

    if (Simulator_CPU_Burst(simulator)) return;

    simulator->elapsed_time++;
  }
  return;
}

/**
 * @fn void Simulator_PreemptivePriority(Simulator_t *simulator)
 * @brief simulate CPU scheduling with Preemptive Priority algorithm
 *
 * @param simulator pointer of Simulator_t
 */
void Simulator_PreemptivePriority(Simulator_t *simulator) {
  Priority_Queue_t *ready_queue = simulator->ready_queue;

  printf("\n# Preemptive Priority Algorithm\n\n");
  while (1) {
    Simulator_LoadReadyQueue_Priority(simulator);

    Simulator_LoadProcess_Priority(simulator);
    Simulator_Priority_Waiting(ready_queue->top);
    Simulator_Process_IO(simulator->waiting_queue->top);
    Simulator_Process_IO_Priority(simulator);

    if (Simulator_CPU_Burst_Preemptive(simulator)) return;

    simulator->elapsed_time++;
  }

  return;
```

```
}

/**
 * @fn void Simulator_RoundRobin(Simulator_t *simulator)
 * @brief simulate CPU scheduling with Round Robin algorithm
 *
 * @param simulator pointer of Simulator_t
 */
void Simulator_RoundRobin(Simulator_t *simulator) {
  Queue_t *ready_queue = simulator->ready_queue;
  int time_quantum = 0;

  printf("\n# Round Robin Algorithm\n\n");
  while (1) {
    Simulator_LoadReadyQueue(simulator);

    Simulator_LoadProcess(simulator);
    Simulator_Queue_Waiting(ready_queue);
    Simulator_Process_IO(simulator->waiting_queue->top);
    Simulator_Process_IO_Queue(simulator);

    if (Simulator_CPU_Burst_RR(simulator, &time_quantum)) return;

    simulator->elapsed_time++;
  }
  return;
}

/**
 * @fn void Simulator_Eval(Simulator_t *simulator)
 * @brief print total execution time,
 *                 CPU utilization,
 *                 average waiting time,
 *                 and average turnaround time
 *
 * @param simulator pointer of Simulator_t
 */
void Simulator_Eval(Simulator_t *simulator) {
  Priority_Queue_t *termninated_processes = simulator->terminated_processes;
  ProcessPtr_t tmp;
  avg_time_len_t CPU_utilization =
      (double)(simulator->elapsed_time + 1 - simulator->idle_time) /
      simulator->elapsed_time;

  /* Calculate average waiting time and average turnaround time */
  while (tmp = Priority_Top(termninated_processes)) {
    Priority_Dequeue(termninated_processes);

    simulator->avg_waiting_time += tmp->waiting_time;
    simulator->avg_turnaround_time += tmp->turnaround_time;

    if (tmp->waiting_time > simulator->max_waiting_time)
      simulator->max_waiting_time = tmp->waiting_time;
  }

  simulator->avg_waiting_time /= simulator->num_process;
  simulator->avg_turnaround_time /= simulator->num_process;

  printf("-> Execution time: %d\n", simulator->elapsed_time + 1);
```

```
  printf("-> CPU Utilization: %.3f\n", CPU_utilization);
  printf("-> Average waiting time: %.3f\n", simulator->avg_waiting_time);
  printf("-> Average turnaround time: %.3f\n", simulator->avg_turnaround_time);

  return;
}

/**
 * void Simulator_Terminate(Simulator_t **simulators)
 * @brief compare execution time,
 *              CPU utilization,
 *              average waiting time,
 *              average turnaround time of all scheduling algorithms
 *
 * @param simulators array of Simulator_t *, which ends with NULL pointer
 */
void Simulator_Terminate(Simulator_t **simulators) {
  printf("\n# Summary\n\n");
  printf(
      "+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++"
      "++"
      "+++++++++++++++\n");
  printf(
      "++++++++++++++++++++++++++++++  CPU Util  ++   Avg WT   ++   AVG TT  "
      " "
      "++   Max WT   ++\n");

  for (int i = 0; simulators[i]; i++) {
    switch (simulators[i]->flag) {
      case FCFS:
        printf("++          FCFS         ++  ");
        break;
      case NON_PREEMPTIVE_SJF:
        printf("++    Non-Preemptive SJF   ++  ");
        break;
      case PREEMPTIVE_SJF:
        printf("++      Preemptive SJF     ++  ");
        break;
      case NON_PREEMPTIVE_PRIORITY:
        printf("++  Non-Preemptive Priority  ++  ");
        break;
      case PREEMPTIVE_PRIORITY:
        printf("++    Preemptive Priority   ++  ");
        break;
      case ROUND_ROBIN:
        printf("++       Round Robin       ++  ");
        break;
      default:
        printf("Error log: invalid flag parameter value\n");
        exit(-1);
    }

    printf(
        "%8.3f  ++ %10.3f ++ %10.3f ++  %8d "
        " ++\n",
        (double)(simulators[i]->elapsed_time - simulators[i]->idle_time) /
            simulators[i]->elapsed_time,
        simulators[i]->avg_waiting_time, simulators[i]->avg_turnaround_time,
        simulators[i]->max_waiting_time);
```

```
  }

  printf(
      "++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++"
      "++"
      "+++++++++++++++\n");

  return;
}
```