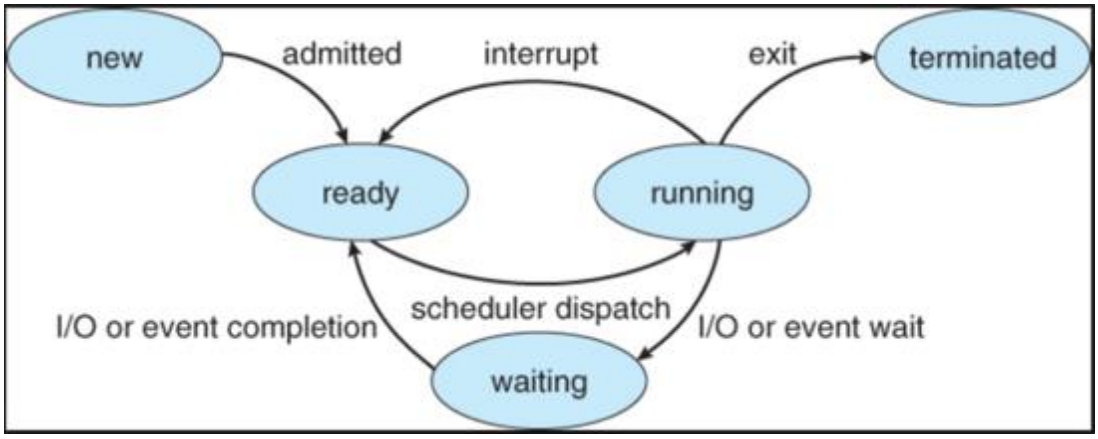
CPU Scheduling Simulator

i. 서론

- Process 란?

  컴퓨터에서 연속적으로 실행되고 있는 프로그램을 프로세스(Process)라고 한다. 프로세스의 상태는 현재 어떤 활동을 하고 있느냐에 따라서 결정되는 데, 아래의 그림과 같이 다른 상태를 보일 수 있다.



  new : 새로 생성된 프로세스의 상태
  ready : Ready queue 에서 대기하고 있는 상태
  running : CPU 를 이용해 명령을 수행하고 있는 상태
  waiting : I/O 완료 또는 어떤 신호를 기다리고 있는 상태
  terminated : 실행 종료된 상태

- CPU Scheduler 란?

  하나의 CPU 는 한 순간에 하나의 프로세스만 실행할 수 있기 때문에 CPU 를 공유하는 프로세스들은 스케줄링을 통해 멀티태스킹을 지원하고, 사용자에게는 병렬 연산이 이루어지는 것과 같은 환경을 제공한다. 이 때, 운영체제에서는 스케줄링을 담당하는 데, 어떤 알고리즘을 사용하는지에 따라 CPU 의 효율이 달라지게 된다. CPU scheduler 는 스케줄링에서 Ready Queue 에 대기하고 있는 프로세스를 CPU 에 할당해서 Running 상태로 바꾸어 주는 역할을 수행한다. 또한, 어떤 프로세스의 IO 처리와 신호 대기를 처리해주는데, IO 가 발생하면 Running 상태에 있던 프로세스를 Waiting Queue 로 보내는 역할도 수행한다. CPU scheduler 가 어떤 알고리즘으로 Running Queue 에서 프로세스를 선택해서 running 상태로 바꾸어 주느냐에 따라 CPU 효율이 많이 달라지므로 CPU scheduler 의 역할이 매우 중요하다.

- CPU Scheduling Simulator

  이번 과제로 구현한 CPU Scheduling Simulator 는 CPU scheduler 가 하는 작업을 수행하는 시뮬레이터로, 입력 받은 알고리즘에 따라 다르게 process 를 선택해서 running 상태로 바꾸어 주는 Scheduler 다. 또한, 입력 받은 알고리즘을 통해 결과로 계산된 Average Waiting Time 과 Average Turnaround Time 를 이용하여 알고리즘마다 성능을 비교할 수 있다.

ii. 본론

➢ 기존 시뮬레이터와 구현한 시뮬레이터의 비교

현재 오픈 소스로, 기존에 이미 구현되어 있는 스케줄러 코드를 많이 찾아 볼 수 있는데, 결국 Process 를 알고리즘에 따라 다르게 선택하고 CPU 할당을 함으로써 가장 이상적인 CPU 효율성을 추구한다는 점에서 비슷하다. 그러나 가장 큰 차이점은 Ready Queue 를 구현하는 방법에서 차이가 발생한다고 생각한다. CPU Scheduler 는 Process 를 Ready Queue 에서 선택해서 Running 상태로 바꾸어 주는데, 이 때 Ready Queue 를 어떤 방법으로 구현 했는지에 따라 스케줄러가 많이 달라진다. 간단한 예시로 Ready Queue 를 배열 또는 원형 큐 등으로 구현한 코드를 많이 볼 수 있었는데, 해당 프로젝트들과는 다르게 이번 프로젝트에서는 Ready Queue 를 Priority Queue 즉, 힙을 이용해 구현했다. Priority Scheduling, Shortest Job First Scheduling 등등 우선순위가 어떤 기준이 되느냐에 따라 가장 높은 우선 순위에 있는 Process 을 Ready Queue 에서 Dequeue 하는 것이므로 해당 구조를 사용해서 Ready Queue 를 구현했다.

➢ 구현한 시뮬레이터

○ 프로세스 초기화

프로세스는 구조체 Process 로 선언하고, 멤버 변수는 아래와 같이 설정했다.

```
int PID;                : 프로세스 ID
int arrivalTime;        : 프로세스 요청 시간
int CPU_burstTime;      : 프로세스의 남아있는 CPU burst time
int IO_startTime;       : 해당 프로세스의 IO 발생 시간
int IO_burstTime;       : 프로세스의 IO burst time
int priority;           : 프로세스의 우선순위(값이 작을수록 우선순위 높음)
int CPU_endTime;        : 프로세스 작업이 끝난 시간
int CPU_runningTime;    : 프로세스가 running 상태로 있었던 시간
int initial_CPU_burstTime; : 초기 CPU burst time
int initial_IO_burstTime;  : 초기 IO burst time
```

해당 멤버 변수들의 값은 PID 는 순차적으로, 이외의 값들은 랜덤으로 초기화 해주었으며, 이는 initProcess(Process * proc)에 구현하였다.

○ Ready Queue , Waiting Queue

해당 스케줄러에서 구현한 알고리즘은 총 8 개로, FCFS, Non-preemptive Shortest-Job-First, Preemptive Shortest-Job-First, Non-preemptive Priority, Preemptive Priority, RoundRobin, LIFO, Longest-Job-First 를 구현하였다. 이 때, 알고리즘에 따라 스케줄러에서 사용하는 Ready Queue 는 2 가지 다른 종류의 구조를 사용하였는데 이는 아래와 같다.

- First-Come-First-Served, RoundRobin : Linked List 를 이용한 Ready Queue 구현
  - QueueList 라는 구조체를 사용했으며 enqueueList 를 이용하여 Queue 의 마지막에 Node 를 추가할 수 있으며 dequeueList 를 이용하여 Queue 의 처음 Node 를 dequeue 한다.

- Non-preemptive Shortest-Job-First, Preemptive Shortest-Job-First, Non-preemptive Priority, Preemptive Priority, LIFO, Longest-Job-First : Priority Queue 즉 Heap 을 이용한 Ready Queue 구현
  - Queue 라는 구조체를 사용했으며 enqueue 를 이용하여 Process 를 enqueue 하게 되면, 힙 구조에서 부모 노드에 있는 Process 와 PriorityComp 를 해서 우선순위가 높으면 부모 노드와 자리를 바꾸며, 낮을 경우 제 자리에 있도록 한다. 반대로 dequeue 를 하게 되면 Heap 에서 가장 높은 자리, 즉 우선순위가 가장 높은 Node 가 dequeue 되게 되며 남은 Node 들을 우선순위에 따라 재정렬하게 된다.

위와 같이 알고리즘마다 다른 자료구조를 사용하여 Ready Queue 를 구현한 이유는, FCFS 와 RR 는 Enqueue 되는 순서에 따라 순서대로 Dequeue 되는 구조여서 일반적인 Queue 를 사용해서 구현하는 것이 용이했다. 반면에 다른 알고리즘들의 경우는 Ready Queue 에서 Dequeue 될 때 각 알고리즘마다 갖고 있는 우선순위에 따라 Process 를 선택해서 Dequeue 하기 때문에, 애초에 Ready Queue 를 Priority Queue 로 구현하게 되면 Deqeue 할 때 가장 높은 Priority 의 노드가 Dequeue 되기 때문에 해당 Queue 에 우선순위 기준만 다르게 정해주면 된다. 예를 들어서, Shortest-Job-First 알고리즘의 경우 Ready Queue 에서 우선순위 기준을 프로세스마다 남은 CPU burst time 을 기준으로 잡되 가장 낮은 burst time 을 가진 프로세스가 순위가 가장 높도록 하면 되었으며, Longest-Job-First 의 경우 반대로 CPU burst time 이 높은 프로세스가 순위가 높도록 PriorityComp 를 설정해주면 되었다.

이와 같이 Ready Queue 의 우선순위에 대한 기준은 PriorityComp(Process p1, Process p2)에 설정해주었다. 따라서, 스케줄링을 시작할 때, 선택한 알고리즘에 따라 Ready Queue 의 초기화를 각각에 맞는 PriorityComp 함수를 이용하여 초기화 시켜주었다.

Ex) Shortest-Job-First :

```
int SJFDataComp(Process p1, Process p2){
        return p2.CPU_burstTime – p1.CPU_burstTime;
}
```

Ready Queue 와 다르게 Waiting Queue 의 경우 배열로 선언해서 각각의 프로세스마다 waiting queue 에서 기다릴 수 있도록 하였다. 즉, 프로세스가 IO 작업 완료를 위해 기다리고 있는 경우 PID 에 맞는 인덱스 칸에서 IO busrt time 이 0 가 될 때까지 기다릴 수 있도록 구현하였다.

- 스케줄링
  - schedule

    해당 스케줄링 함수는 FCFS, RR 알고리즘을 제외한 모든 알고리즘에서 사용하는 스케줄링 함수로, Ready Queue 를 Priority Queue 로 사용한다는 특징이 있다.

```
알고리즘 선택 값에 따라 Ready Queue 초기화
Waiting Queue 초기화
while(시간 < 최대 시간){
        프로세스가 도착할 경우 Ready Queue 에 Enqueue

        // IO 처리
        if(프로세스의 IO burst time 이 남아있고 IO interrupt 가 발생할 시간이 되면){
                IO 처리중인 프로세스 출력

                해당 프로세스가 running 상태였다면 running 을 중단
                Waiting Queue 로 enqueue
                IO burst time 감소
        }

        if(running 상태인 프로세스가 없고, Ready Queue 에 프로세스가 남아있다면)
                Ready Queue 에서 dequeue 해서 해당 프로세스를 running 상태로 전환
        if(running 상태인 프로세스가 있는 경우){
                if(preemp 가 가능할 경우){
                        runnning 이던 프로세스를 다시 Ready Queue 에 enqueue
                        Ready Queue 에서 가장 높은 우선순위의 프로세스를 dequeue
                }

                running 프로세스의 CPU burst time 을 감소
                running 프로세스의 CPU running time 을 증가
                if(프로세스의 CPU burst time 이 끝난 경우){
                        해당 프로세스가 terminate 된 시간을 저장
                        해당 프로세스를 terminate 시키고 running 을 empty 시켜줌
```

```
                }
        }

        // IO waiting 마친 프로세스 다시 Ready Queue 로
        IO burst time 이 0 가 된 프로세스가 있을 경우 해당 프로세스를 Ready Queue 에
Enqueue
}
```

- run_FCFS_RR
  해당 스케줄링 함수는 FCFS, RR 를 지원하는 스케줄러 함수로, 앞서 구현한 schedule 함수와 비슷하다. 단지, Ready Queue 가 Linked List 를 이용한 Queue 로 구현되어 있다는 점이 다르다. 때문에 먼저 enqueue 된 프로세스가 먼저 작업을 수행하도록 되어있다. 또한, Round Robin 을 위해 time 배열을 사용해서 해당 프로세스가 CPU 할당 받아서 running 한 시간을 기록했다. 기록된 time 값이 TIME QUAMTUM 값과 비교했을 때 같을 경우, 다시 Ready Queue 의 마지막으로 돌아가도록 enqueue 됐으며 다음에 수행될 프로세스는 Ready Queue 에 가장 앞에 위치한 프로세스가 진행되도록 하였다.

```
알고리즘 선택 값에 따라 Ready Queue 초기화
Waiting Queue 초기화
while(시간이 최대){
        프로세스가 도착할 경우 Ready Queue 에 Enqueue

        // IO 처리
        if(프로세스의 IO burst time 이 남아있고 IO interrupt 가 발생할 시간이 되면){
                IO 처리중인 프로세스 출력

                해당 프로세스가 running 상태였다면 running 을 중단
                Waiting Queue 로 enqueue
                IO burst time 감소
        }

        // RoundRobin
        if(RR 의 경우, running 중인 프로세스가 있으며 해당 프로세스가 TIME
QUANTUM 만큼 CPU 사용했을 때){
                해당 프로세스를 ReadyQueue 에 다시 enqueue
                해당 프로세스가 CPU 사용한 time 을 0 으로 초기화
                running 프로세스 초기화
        }

        if(running 상태인 프로세스가 없고, Ready Queue 에 프로세스가 남아있다면)
                Ready Queue 에서 dequeue 해서 해당 프로세스를 running 상태로 전환
        if(running 상태인 프로세스가 있는 경우){

                running 프로세스의 CPU burst time 을 감소
                running 프로세스의 CPU running time 을 증가
                running 프로세스의 time 값 증가
                if(프로세스의 CPU burst time 이 끝난 경우){
                        해당 프로세스가 terminate 된 시간을 저장
                        해당 프로세스를 terminate 시키고 running 을 empty 시켜줌
                }
        }

        // IO waiting 마친 프로세스 다시 Ready Queue 로
```

IO burst time 이 0 가 된 프로세스가 있을 경우 해당 프로세스를 Ready Queue 에 Enqueue }

- Average Waiting Time, Average Turnaround Time
  프로세스마다 CPU burst time 이 끝난 시간을 CPU_endTime 에 기록하였으므로 아래와 같이 AWT, ATT 를 구하였다.

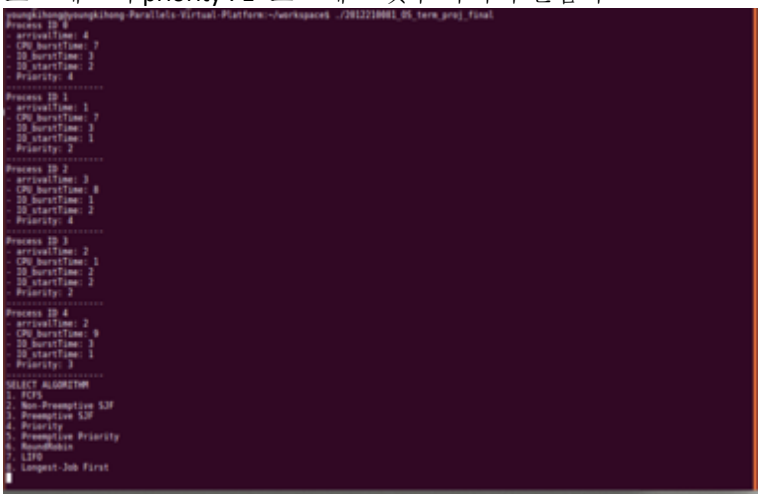
waiting time = process 의 CPU_endTime – 도착시간 – IO burst time – CPU burst time turnaround time = process 의 CPU_endTime – 도착시간 Average waiting time = waiting time 총합 / 프로세스 갯수 Average turnaround time = turnaround time 총합 / 프로세스 갯수

- Main
  스케줄러의 메인 함수로, 어떤 알고리즘을 선택할 지 입력 받고 그에 맞는 알고리즘에 따라 스케줄링을 하고 출력하는 기능을 수행한다.

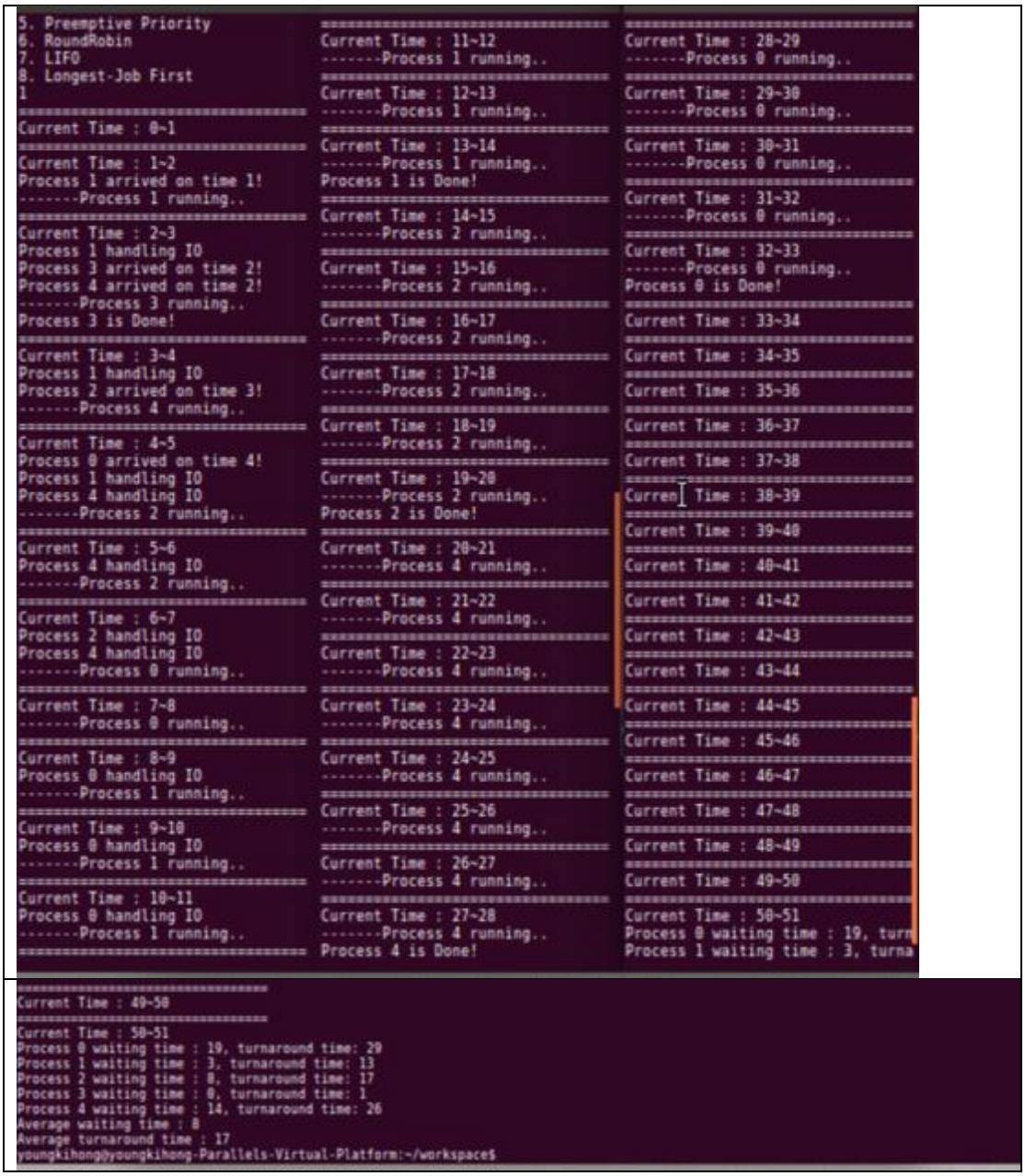
process 초기화 process 출력 알고리즘 선택 메뉴 출력 사용자가 알고리즘 선택 메뉴 선택 알고리즘에 따라 스케줄링 함수 선택 알고리즘 성능 출력

➢ 실행 결과

- 메뉴 화면 및 초기화
  프로세스 수 : 5
  프로세스 도착 시간 : 1~5 사이의 랜덤 수
  프로세스의 CPU burst time : 1~10 사이의 랜덤 수
  프로세스의 IO burst time : 1~5 사이의 랜덤 수
  프로세스의 IO interrupt : 1~3 사이의 랜덤 수
  프로세스의 priority : 1~프로세스 갯수 사이의 랜덤 수

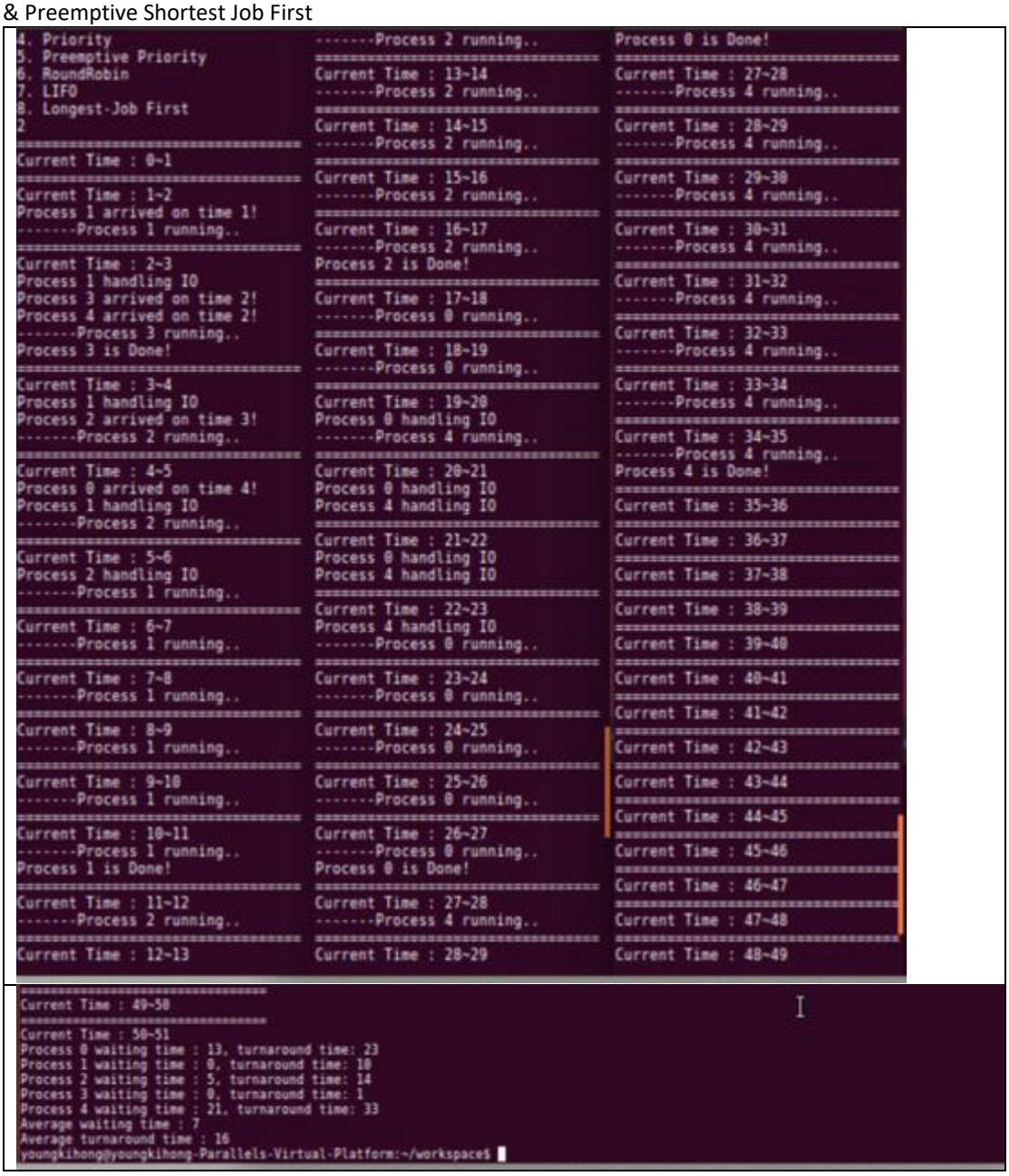


- First-Come First-Served

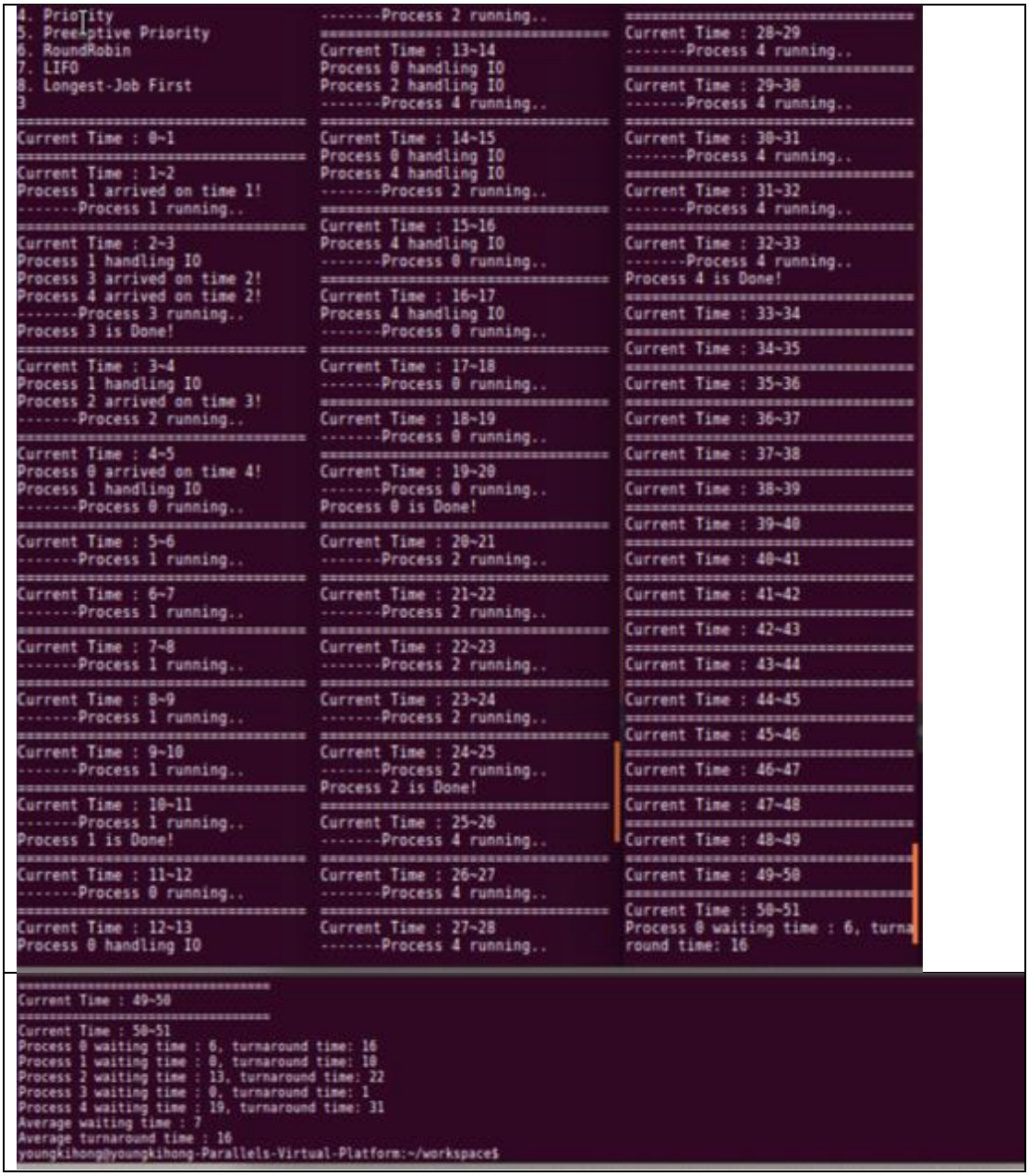


FCFS 는 먼저 도착한 프로세스 먼저 처리하므로, 프로세스 1 이 먼저 CPU 를 할당 받는 것을 볼 수 있다. 2~3 초 구간에서 프로세스 1 은 IO 발생 때문에 Waiting Queue 에 들어가게 되며, 이 때문에 다른 프로세스, 즉 프로세스 3 이 CPU 를 할당 받아서 작업을 수행하게 된다.

- Non-Preemptive Shortest Job First
  & Preemptive Shortest Job First

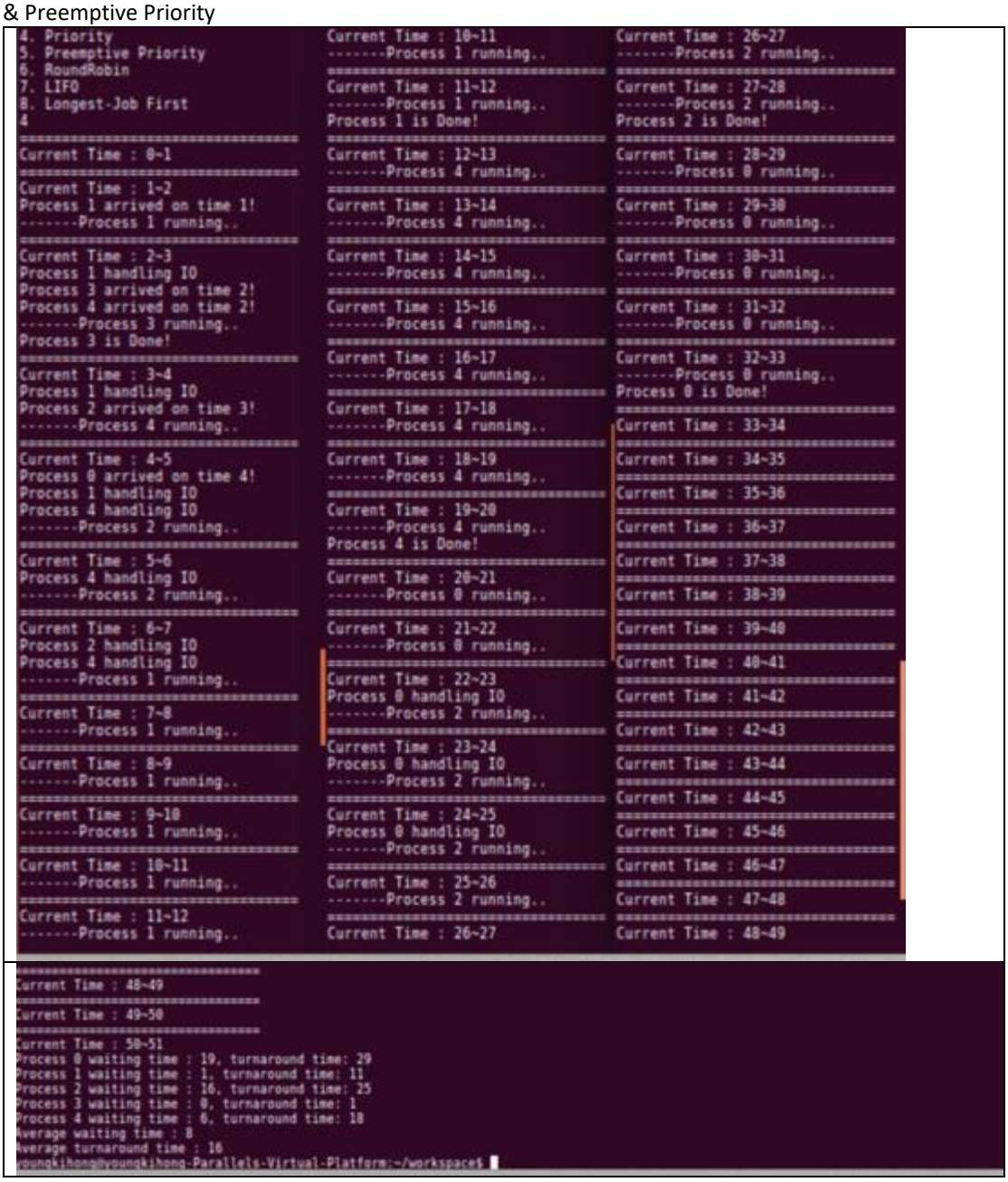


Non-preemptive Shortest Job First 의 경우 non-preemptive 이기 때문에 진행중인 프로세스가 있으면, 해당 프로세스가 IO interrupt 가 발생하지 않는 이상 끝날 때 까지 계속 CPU 를 사용하게 된다. 프로세스를 비교하는 부분은 5~6 구간에서 프로세스 0, 1, 4 가운데 프로세스 1 이 선택되는 이유는 CPU burst time 이 셋 가운데 가장 작기 때문에 선택 되서 CPU 를 할당 받게 된다.

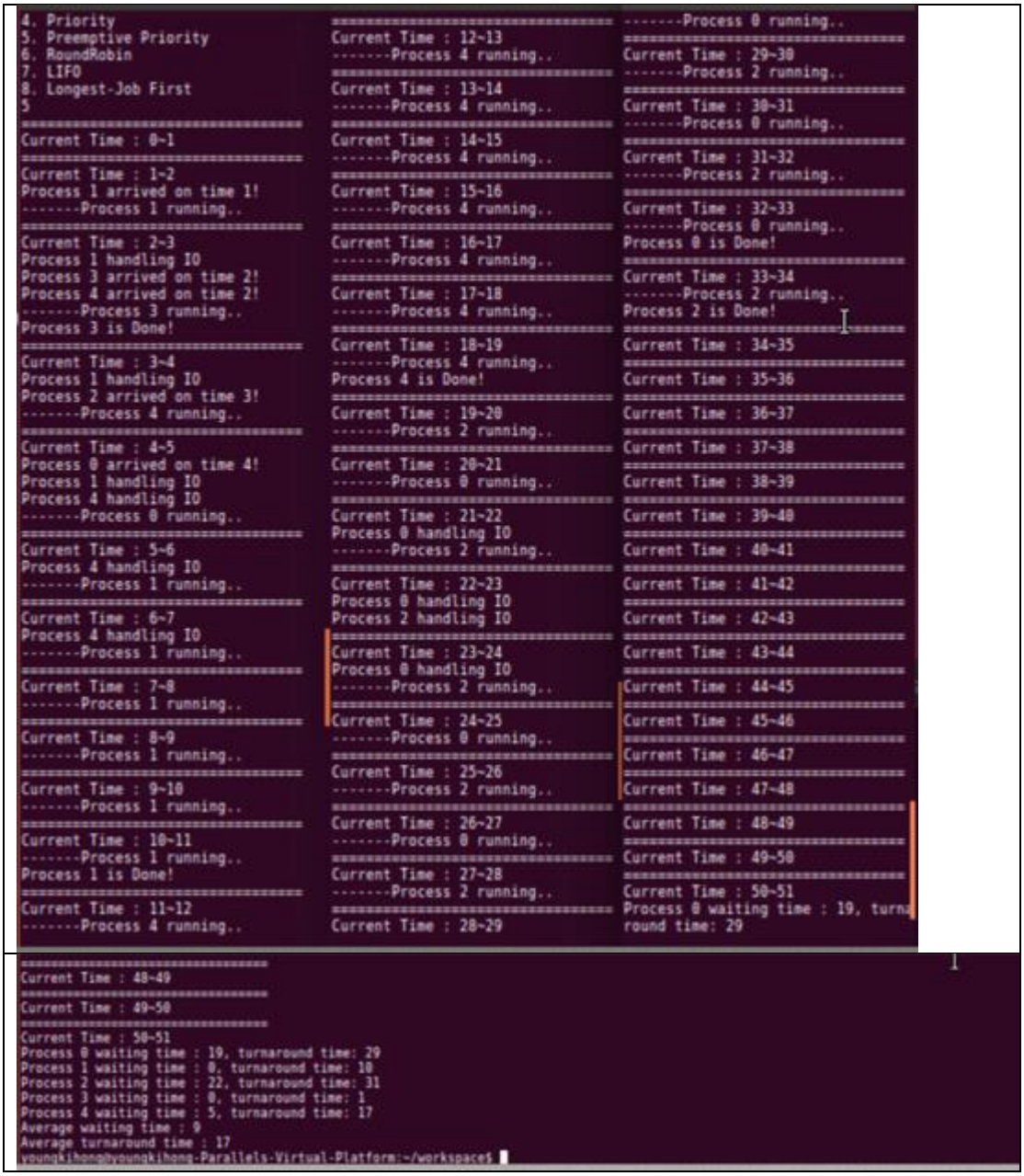


Preemptive SJF 의 경우 진행중인 프로세스가 있더라도, 더 짧은 프로세스가 존재할 경우 해당 프로세스를 우선적으로 처리하는 알고리즘이다. 앞서 살펴보았던 non-preemptive SJF 와는 다르게 4~5 구간에서 CPU burst time 이 더 짧거나 같은 Process 0 가 CPU 를 할당받아서 사용하게 되는 사실을 확인할 수 있다.

- Non-Preemptive Priority
  & Preemptive Priority

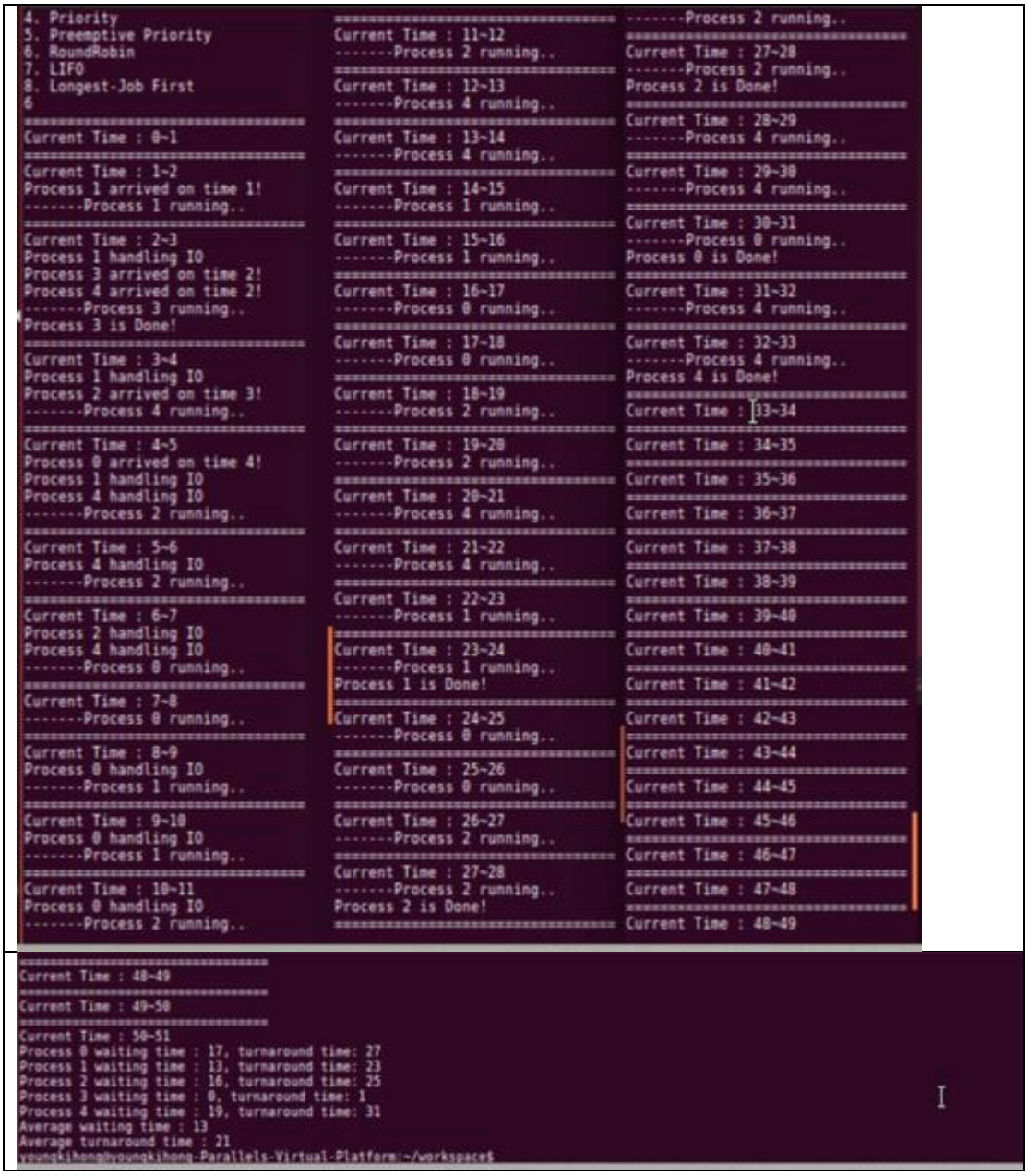


Non-preemptive Priority 스케줄링 기법은 프로세스의 priority 에 따라 스케줄링을 하는 기법으로, 6~7 구간에서 프로세스 0, 1 가운데 Priority 가 프로세스 0 은 4, 프로세스 1 은 2 로 우선순위가 프로세스 1 이 더 높기 때문에 프로세스 1 이 CPU 를 할당받는 것을 확인할 수 있다.



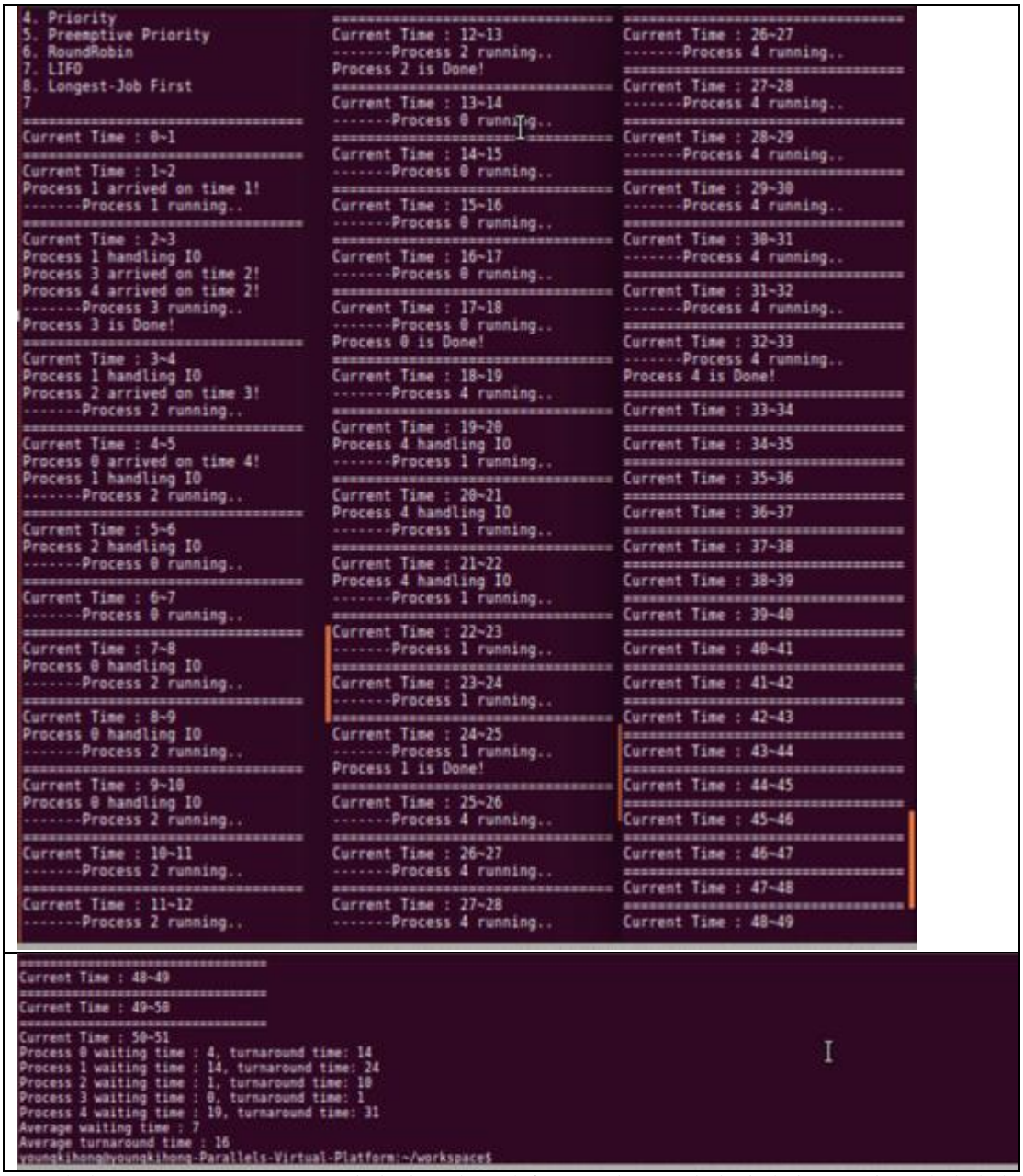
Preemptive Priority 스케줄링은 priority 가 더 높은 프로세스가 있을 경우 진행중이던 프로세스를 멈추고 새로운 프로세스로 바꾸는 기법을 말하는데 , 이는 5~6 구간에서 확인할 수 있다. 4~5 구간에서 프로세스 0 이 진행중이었는데 IO 처리가 끝난 process 1 이 ready queue 에 5 초에 들어오자, priority 가 더 높은 process 1 이 preempt 해서 CPU 를 5~6 구간에서 할당 받는 사실을 확인할 수 있다.

- RoundRobin



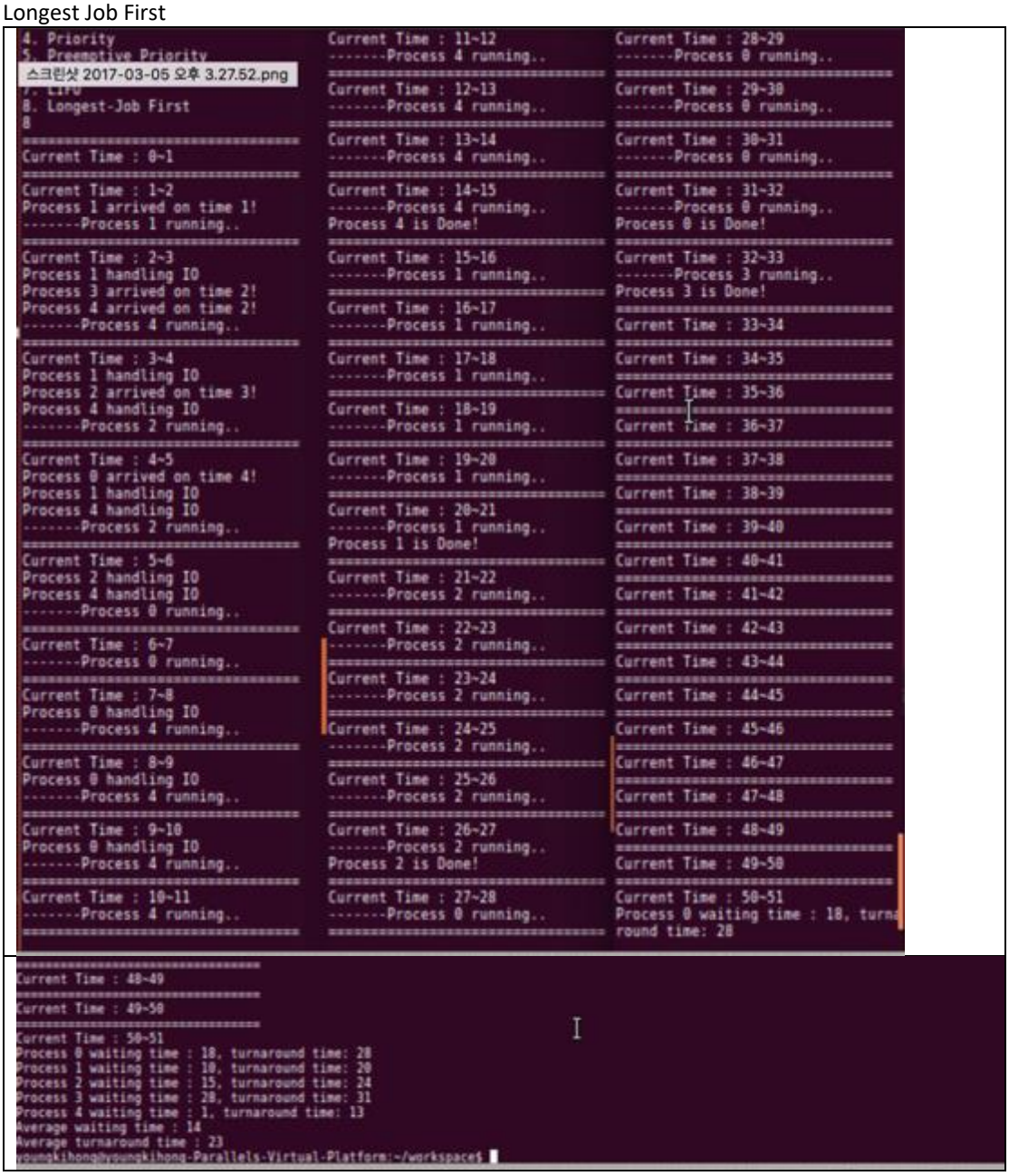
RoundRobin 은 우선 프로세스가 진행중일 때, Time quantum 만큼 진행을 하면 그 다음 프로세스가 진행할 수 있도록 바뀌는 기법이다. 4~5, 5~6 구간을 확인해보면, 프로세스 2 가 계속 진행중인데 Time quantum 2 만큼 진행하자, 이 다음 프로세스 0 이 진행되는 사실을 확인할 수 있다. 다음에 어떤 프로세스가 진행될 지 여부는 FCFS 와 같이 ready queue 에 먼저 들어온 프로세스가 선택되서 CPU 를 할당받게 될 것이다. 이 RoundRobin 의 특징이자 장점은 여러 프로세스가 CPU 할당을 번갈아 가면서 받게 되므로 starving 을 해결할 수 있다는 장점이 있다.

- Last-In-First-Out



LIFO 기법은 Last in first out 으로, 가장 나중에 들어온 프로세스를 먼저 처리하도록 하는 기법이다. 이는 5~6 구간에서 확인할 수 있는데, 프로세스 0, 1, 4 가운데 가장 늦게 ready queue 에 들어온 프로세스 0 이 선택되서 CPU 할당을 받는 것을 확인할 수 있다.

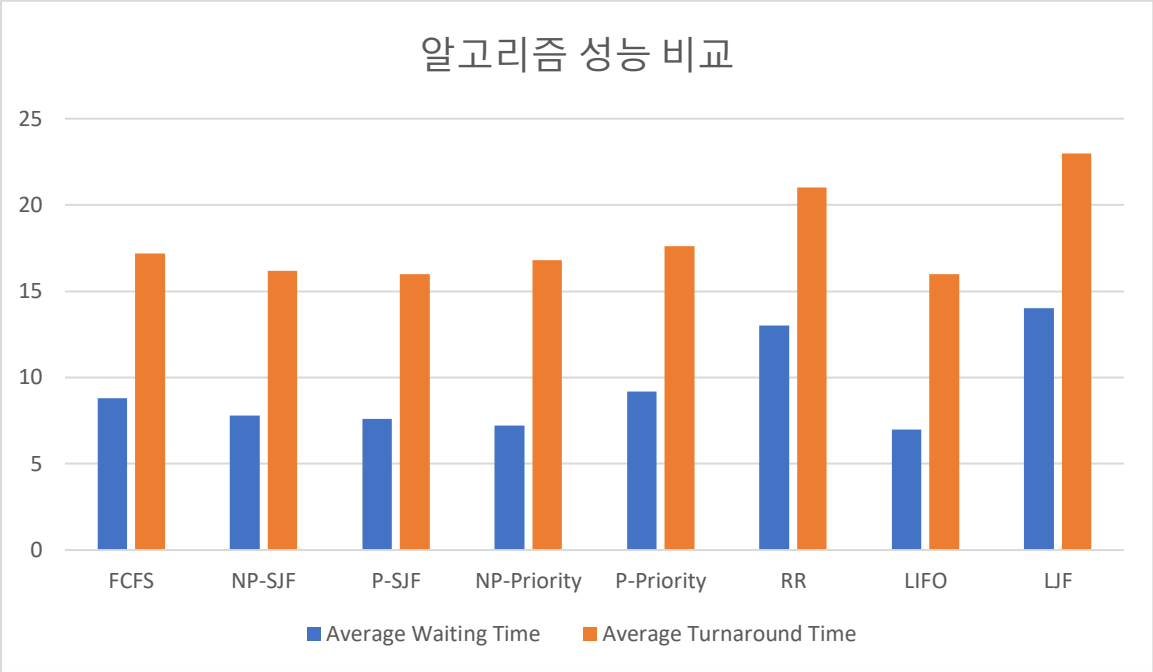
○ Longest Job First



Longest-Job-First 기법은 Shortest-Job-First 와 반대로 가장 CPU burst time 이 많이 남은 프로세스를 우선적으로 처리하는 기법이다. 이는 5~6 구간을 SJF 와 비교했을 때 SJF 에서는 프로세스 1 이 처리되고 있는 반면에 LJF 에서는 프로세스 0 이 처리되고 있는 점을 확인할 수 있다. 5 초에 ready queue 에 있는 프로세스 0, 1 가운데 CPU burst time 이 더 큰 프로세스 0 이 선택되어 CPU 할당 받았다는 사실을 확인할 수 있다.

iii. 결론

➢ 알고리즘들간 비교 분석



구현한 알고리즘마다 발생한 Average Waiting Time, Average Turnaround 을 비교하기 위해 위와 같이 표로 나타났다. 이를 기반으로 알고리즘 간 성능을 비교해보았더니, Shortest-Job-First 알고리즘을 이용한 스케줄링이 대체로 Average Waiting Time 이 작게 나타난 점을 확인할 수 있었다. 이는 CPU burst time 이 작은 프로세스부터 먼저 처리하면서 최대한 waiting time 을 줄임으로써 생긴 결과라고 생각된다. 반면에 RR 와 Longest-Job-First 는 Average waiting time 와 Average turnaround time 이 다른 알고리즘에 비해 월등히 높게 나타난 사실을 확인할 수 있었다. RR 의 경우 번갈아 가면서 CPU 할당을 하고 instruction 을 처리하므로 아무래도 starving 을 해결한다는 장점이 있으므로 그에 따라 생기는 비용이라고 간주된다. 그러나 LJF 의 경우 알고리즘 특성상 가장 CPU burst time 이 긴 작업을 먼저 수행하므로 당연하게 Average waiting time 과 Average turnaround time 이 커질 수 밖에 없고 비효율적이라는 사실을 알 수 있었다. 따라서 위의 표에 나타난 결과로만 확인해보면, 이상적인 알고리즘은 Shortest Job First 인 것을 알 수 있었다.

➢ 프로젝트 소감 및 추후 발전 계획

이번 프로젝트를 통해 CPU 스케줄링에 대해 개념적으로 많이 알 수 있었으며 다양한 알고리즘을 직접 구현하고 결과물을 비교하면서 더 구체적으로 배울 수 있었다. 현재 스케줄러에 구현한 알고리즘들 가운데 starving 을 자체적으로 해결할 수 있었던 RoundRobin 에 비해 다른 알고리즘들은 해결하지 못한 채 잠재적으로 문제 발생 가능성을 지니고 있다. 이를 해결하기 위해 aging 기법을 추가하면 더욱 나은 CPU 스케줄러로 발전시킬 수 있을 것이라고 생각한다.

이와 같이 개념적인 부분 외에도 C 언어로 Linux 환경에서 프로그래밍을 하면서 여러 시행착오를 겪었으며 특히나 Ready Queue 를 구현하는 방법을 고민하는 과정에 있어서 어떤 자료구조를 사용하면 더욱 효율적으로 스케줄러를 구현할 수 있을지 많이 조사하고 공부하게 되었다. 단순히 돌아가는 코드가 아닌 탐색 시간, 복잡도를 줄일 수 있도록 설계를 하면서 프로그래머가 갖추어야할 기본적인 능력을 조금이나마 더 키울 수 있었다. 프로젝트를 마무리하면서 다소 아쉬운 점은, 중복되는 코드를 제거하고 좀 더 효율적이고 가독성이 뛰어나게 개선시키지 못한 점이 아쉽다.

iv. 코드

- https://github.com/youngkih/cpu-scheduling-simulator