REPORT

Project#01 – Simple Scheduling

Freedays: 4 (1 used)

과목명	운영체제(MS)	담당교수	유시환 교수님
학 번	32204292, 32217072	전 공	모바일시스템공학과
이 름	조민혁, 김도익	제 출 일	2024/11/11

목 차

1. Introduction

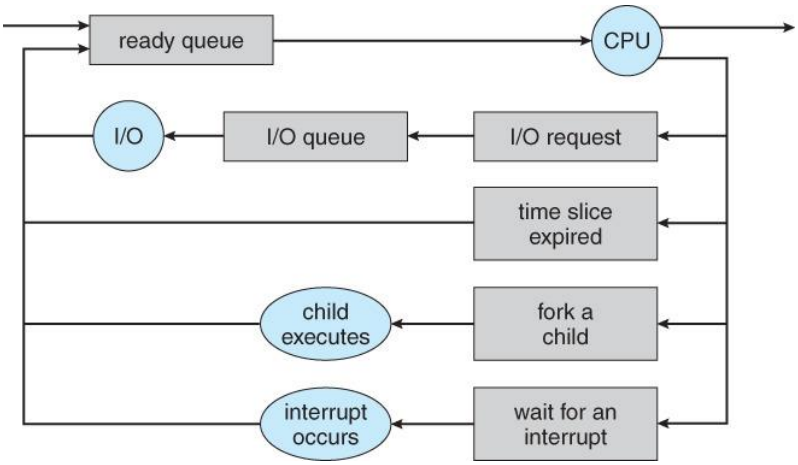


Fig 1. CPU Scheduling

컴퓨터 시스템에서 OS는 컴퓨터의 자원을 효율적으로 프로세스에게 분배하여 CPU와 I/O장치의 이용률을 극대화하는 것을 목표로 하는데, 이러한 자원 관리의 핵심 과정 중 하나가 CPU 스케줄링이다. CPU 스케줄링은 프로세스들이 CPU를 효과적으로 사용하도록 제어하는 과정이다. Fig 1에서 보여주듯이, Ready Queue, I/O Queue와 같은 다양한 큐를 통해 CPU및 I/O자원을 관리하고, I/O 요철, 타임 슬라이스 만료, 자식 프로세스 생성, 인터럽트 발생 등의 이벤트에 따라 프로세스가 상태를 전환함으로써 시스템의 성능을 향상시킬 수 있다.

CPU 스케줄링은 크게 선점 스케줄링과 비선점형 스케줄링이라는 두가지 종류로 나눌 수 있다. 선점 스케줄링의 경우는 이미 CPU를 차지하고 있는 하나의 프로세스를 중단시키고, 우선순위가 높은 다른 프로세스가 CPU를 점유할 수 있는 방식이다. 반면 비선점형 스케줄링은 한 프로세스가 CPU를 점유하고 있으면, 작업을 종료하고 CPU를 반환할 때까지 다른 프로세스가 CPU를 점유하지 못하는 방식이다.

본 레포트에서는 운영체제에서 CPU 스케줄링 기법을 이해하고, 시뮬레이션을 구현한다. 이를 위해 Round Robin 스케줄링 알고리즘을 사용한 CPU 스케줄링 시뮬레이터를 구현한다. 더 나아가, FCFS, SJG 스케줄링을 추가 구현함으로써 Round Robin 스케줄링과의 비교를 진행한다.

본 레포트의 구성은 다음과 같다. 2장에서는 프로그램을 구현하는데 필요한 요구사항들을 제시하고, 3장에서는 이를 구현하기 위한 관련 개념을 살펴본다. 이후 4장에서는 프로그램을 구현하는데 필요한 아이디어를 순서도를 통해 알아보고, 아이디어를 바탕으로 구현한 코드에 대한 설명을 한다. 5장에서는 구현한 프로그램의 실행 결과를 살펴보고, 6장에서는 해당 실행 결과를 평가한다. 마지막으로 7장에서 결론을 작성하며 레포트를 마무리한다.

2. Requirements

Table 1. Requirements Table

Index	Requirements
1	부모 프로세스는 10개의 자식 프로세스를 생성한다.
2	부모 프로세스는 Round-Robin 스케줄링 방식으로 자식 프로세스들을 관리한다.
3	타이머를 설정하여 주기적으로 타이머 인터럽트를 발생시킨다.
4	Ready Queue와 Wait Queue를 통해 큐 관리를 한다.
5	자식 프로세스에게 시간 할당을 하며 남은 시간을 관리한다.
6	자식 프로세스는 cpu burst와 io burst를 무작위로 생성하여 진행한다.
7	자식 프로세스는 cpu burst가 0이 되면 부모 프로세스에게 I/O 요청을 한다.
8	부모 프로세스는 I/O 요청을 통해 자식 프로세스를 Ready Queue -> Wait Queue로 이동시킨다.
9	Wait Queue에서 io burst 값이 감소한다.
10	Io burst 값이 0이 되면, 해당 프로세스를 Wait Queue -> Ready Queue로 이동시킨다.
11	모든 스케줄링 작업을 schedule_dump.txt 파일에 출력한다.
12	Schedule_dump.txt에 (0~10,000) Time Tick을 기록한다.
13	추가 Scheduling Policy를 구현하여 Avg. Wait Time과 Avg. Turnaround Time을 비교한다

Simple Scheduling Program에서 요구되는 요구사항들이 Table 1에 제시 되어있다.

3. Concepts

본 장에서는 해당 프로그램을 구현하는데 있어서 필요한 관련 개념들을 서술한다.

3-1. CPU-I/O Burst cycle

Burst는 한 작업 처리에 필요한 시간 단위를 의미하는데, CPU Burst는 프로세스가 CPU를 집중적으로 실행하는 구간을 의미한다. I/O burst는 프로세스가 I/O요청을 기다리고 작업을 처리하는 구간이다.

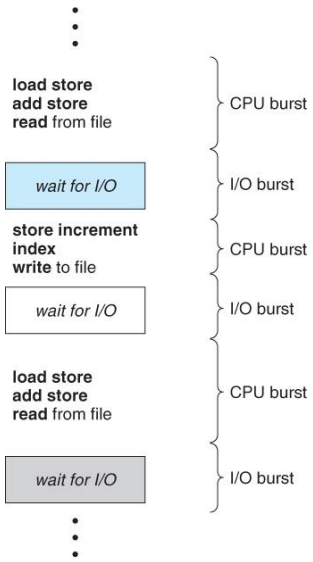


Fig 2. Alternating sequence of CPU and I/O burst

프로세스는 CPU burst와 I/O burst의 사이클로 구성되어 실행되는데, 이 두 상태를 Fig 2에서 볼 수 있듯이, CPU Burst를 시작하고 그 다음으로 I/O burst가 발생하는 것을 반복하다가, 마지막 CPU Burst는 I/O burst가 발생하지 않고, 종료 시스템 콜을 통해 종료된다.

특히, I/O Burst 중심의 프로세스인 경우는 긴 I/O Burst 시간을 갖고, 짧은 CPU Burst 시간을 갖는다. 또, CPU Burst 중심의 프로세스인 경우는 긴 CPU Burst 시간을 갖고, 짧은 I/O Burst 시간을 갖는다. CPU 스케줄러는 프로세스의 CPU 및 I/O burst 패턴을 최적으로 관리하여 CPU를 항상 바쁘게 유지하는 것을 목표로 하기 때문에 프로세스가 CPU 또는 I/O 중심인지에 따라 최적의 스케줄링 전략이 달라질 수 있다.

3-2. CPU scheduling

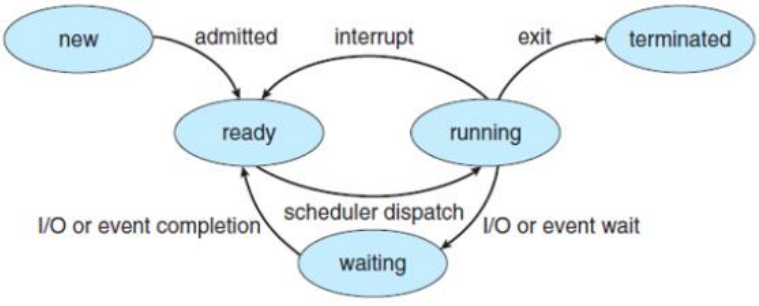


Fig 3. State Diagram

CPU 스케줄링이란 CPU 이용률을 높이고 사용자 응답시간을 최소화하기 위해서 실행을 위해 ready queue 에서 CPU 할당을 기다리는 여러 프로세스 중 언제, 어떤 프로세스를 사용할 것인지 결정하는

과정이다. Fig 3 에서 볼 수 있듯이, CPU 스케줄링은 프로세스가 Ready 상태, Running 상태, Waiting 상태 간에 전이될 때 발생한다. 특히 CPU 스케줄링은 다음과 같은 네 가지 상황에서 이루어진다.

1. 프로세스가 실행상태에서 대기상태로 전환될 때
2. 프로세스가 실행상태에서 준비완료 상태로 전환될 때
3. 프로세스가 대기상태에서 준비완료상태로 전환될 때
4. 프로세스가 종료될 때

위의 1 번과 4 번 상황은 반드시 새로운 프로세스를 선택해야 하는 상황이다. 이를 비선점형 스케줄링이라고 한다.

비선점형 스케줄링이란, 한 프로세스가 CPU 를 점유하면, 해당 프로세스가 종료될 때까지 다른 프로세스로 전환되지 않는 방식이다. 즉, 실행 중인 프로세스를 중단하고, 다른 프로세스를 실행할 수 없다. 커널 구조를 단순하게 유지하고, 구현하기 쉽다는 장점이 있지만, 주어진 시간안에 실행이 완료되어야 하는 환경에서 사용하기에는 적합하지 않다.

반면, 2 번과 3 번의 상황에서는 실행 중인 프로세스를 중단하고 다른 프로세스를 강제적으로 실행시킬 수 있는데, 이를 선점형 스케줄링이라고 한다. 선점형 스케줄링은 프로세스의 우선순위에 따라 CPU 를 재할당하여, 시스템의 응답성을 높이는 데 장점이 있다. 하지만 작업이 얼마나 걸릴지 예측하기가 어렵고, 우선순위가 높은 프로세스들이 계속해서 큐에 추가가 될 경우 오버헤드를 일으킬 수 있다는 단점이 있다.

3-3. Scheduling Algorithms

CPU 스케줄링이란 ready queue 에 있는 여러 프로세스 중 CPU 를 할당할 프로세스를 결정하는 과정이다. 이때 CPU 를 할당할 프로세스를 결정하는 규칙을 스케줄링 알고리즘이라고 한다. 다양한 스케줄링 알고리즘이 있으며, 스케줄링 알고리즘은 각각 고유한 특성과 장단점을 가지고 있다. 스케줄링 알고리즘을 선택할 때 고려해야 할 다섯개의 기준이 있는데, 그 기준은 아래와 같다.

1. CPU 이용률: 우리는 가능한 CPU 를 최대한 바쁘게 유지하기를 원한다.
2. 처리량: CPU 가 프로세스를 수행하느라고 바쁘다면, 작업이 진행되고 있는 것이다. 작업량 측정의 한 방법은 단위 시간당 완료된 프로세스의 개수로, 이것을 처리량이라고 한다.
3. 총 처리 시간: 특정 프로세스의 입장에서 보면, 중요한 기준은 그 프로세스를 실행하는데 소요된 시간일 것이다. 프로세스의 제출 시간의 간격을 총 처리 시간이라고 한다. 총 처리 시간은 waiting queue 에서 대기한 시간, CPU 에서 실행한 시간, 그리고 I/O 시간을 합한 시간이다.
4. 대기 시간: CPU 스케줄링 알고리즘은 프로세스가 실행하거나 I/O 을 하는 시간의 양에 영향을 미치지는 않는다. 스케줄링 알고리즘은 단지 프로세스가 waiting queue 에서 대기하는 시간의 양에만 영향을 준다. 대기 시간은 waiting queue 에서 대기하면서 보낸 시간의 합이다.

5. 응답 시간: 대화식 시스템에서, 총 처리 시간은 최선의 기준이 아닐 수도 있다. 프로세스가 어떤 출력을 매우 일찍 생성하고, 앞서의 결과가 사용자에게 출력되는 사이에 새로운 결과를 얻으려고 연산을 계속하는 경우가 종종 있다. 따라서 또 다른 기준은 하나의 요구를 제출한 후 첫 번째 응답이 나올 때까지의 시간이다. 응답 시간이라고 하는 이 기준은 응답이 시작되는 데까지 걸리는 시간이지, 그 응답을 출력하는 데 걸리는 시간은 아니다.

CPU 이용률과 처리량은 최대한으로 하고, 총 처리 시간, 대기 시간, 그리고 응답 시간을 최소화하는 것이 바람직하다. 각 스케줄링 알고리즘은 이러한 기준들 중 특정한 기준을 더 중요시 여길 수 있기 때문에, 사용 환경과 요구 사항에 따라 적합한 알고리즘을 선택하는 것이 중요하다. 아래에서는 다양한 스케줄링 알고리즘의 주요 특성과 장단점을 설명한다.

3-3-1. FCFS(First-Come-First-Served)

Process	Arrival Time	Burst Time	Waiting Time	Turnaround Time
P1	0	4	0	4
P2	1	10	3	13
P3	2	6	12	18
P4	3	8	17	25

Fig 4. Example of FCFS Scheduling

FCFS 는 먼저 CPU 사용을 요청하는 프로세스 순서대로 프로세스에 CPU 를 할당하는 실행되는 방식이다. 이는 Fig 4 와 같이 FIFO(First In First Out)와 유사한 개념으로, 단순하고 직관적이기 때문에 이해와 구현이 쉽다. FCFS 스케줄링 알고리즘의 작동 방식은 CPU 가 사용할 수 있는 상태일 때, ready queue 의 가장 앞에 있는 프로세스에 할당하고, 프로세스가 종료되면, ready queue 에서 해당 프로세스를 제거한 후, 그 다음에 있는 프로세스에게 CPU 를 할당한다.

FCFS 는 비선점형 스케줄링 방식이므로, 한 프로세스가 CPU 를 점유하면 해당 프로세스가 완료될 때까지 중단되지 않으며, 다른 프로세스가 CPU 를 점유할 수 없다. 이러한 특성 때문에 기아현상이 없다는 장점이 있지만, 긴 작업이 먼저 시작되면 총 처리시간이 길어질 수 있다는 단점이 존재한다.

FCFS 스케줄링은 여러 프로세스의 실행시간이 비슷할 때, 공평하고 간단한 방식으로 순서대로 실행할 수 있어 waiting time 이 균일하게 유지된다. 그러나, 만약 프로세스들의 실행 시간의 차이가 크다면, 긴 실행 시간을 갖는 프로세스가 가장 먼저 CPU 를 할당 받았을 때, 뒤따르는 프로세스들이 오랫동안 대기하는 상황이 발생할 수 있는데, 이것을 호위효과(convey effect)라고 하며, 이로 인해 전체 시스템의 처리 효율이 떨어질 수 있다.

3-3-2. SJF(Shortest-Job-First)

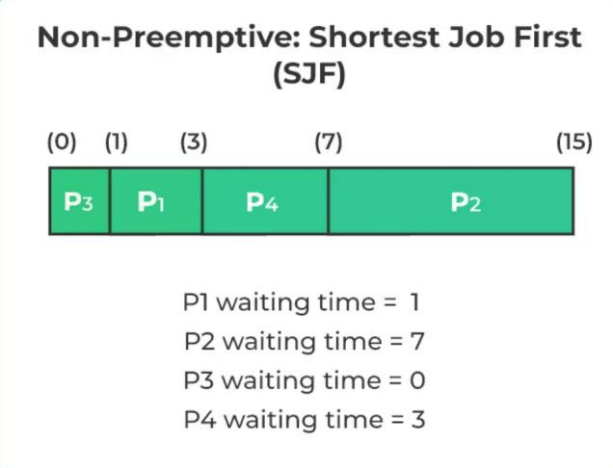


Fig 5. Example of SJF Scheduling

앞서 언급한 FCFS 의 문제를 보완할 수 있는 방법에는 SJF 가 있다. SJF 는 Fig 5 와 같이 여러 프로세스 중 가장 실행시간이 짧은 프로세스 순서대로 CPU 를 할당 받아 실행하는 방식이다. 짧은 작업을 빠르게 처리하여 응답 시간을 줄일 수 있고, 뒤따르는 프로세스들의 오랫동안 대기하지 않아도 된다는 장점이 있다.

그러나 이러한 SJF 는 이론적으로는 가능하지만, 실제 시스템에서는 프로그램이 실행되기전에 프로그램의 실행시간을 미리 알고, 프로그램의 실행순서를 결정하는 것은 현실적으로 어렵기 때문에 적용하기 힘든 방식이다. 하지만 그 근사값을 예측하는 기법들이 존재한다.

예를 들어, 과거의 CPU 사용 패턴을 기반으로 예측하는 방법이 있다. 이전에 실행된 프로세스의 CPU 사용시간 평균값을 기반으로 다음 CPU 사용 시간을 예측하여 우선순위를 결정하는 방식이다. 지수평균법과 같은 기법을 사용하여, 과거 CPU 사용 시간에 가중치를 두어 예측 값을 계산하여 구현할 수 있다.

이와 같은 SJF 스케줄링 알고리즘은 짧은 실행시간의 작업을 우선으로 처리하여 평균 대기 시간과 응답 시간을 감소시키는데 효과적이다. 특히, 짧은 작업이 많은 시스템에서 효율적이다.

3-3-3. Round Robin

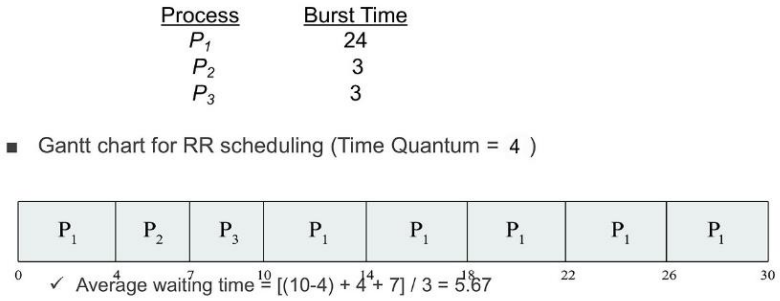


Fig 6. Example of Round Robin Scheduling

선점형 스케줄링 방식 알고리즘 중 하나인 Round Robin 은 시분할 시스템을 위해 설계된 스케줄링 알고리즘으로, 각 프로세스가 정해진 시간 동안 공평하게 CPU 를 사용할 수 있도록 한다. Round Robin 은 선점형 스케줄링 알고리즘으로써 실행중인 프로세스가 시간 단위 따라 중단되고, 다른 프로세스가 CPU 를 할당 받을 수 있다. Fig 6 과 같이 각 프로세스는 타임 슬라이스(time slice) 또는 타임 퀀텀(time quantum)이라고 하는 시간단위동안 CPU 를 사용할 수 있으며, 이후에는 waiting queue 의 뒤로 이동하여 다음 순서를 기다린다.

Round Robin 스케줄링 알고리즘은 waiting queue 가 원형 큐로 구성되어 있다. 타임 퀀텀이 종료된 프로세스는 원형 큐의 뒤로 이동하기 때문에 모든 프로세스가 공평하게 CPU 를 할당 받는 특성을 가지고 있다. 이로 인해 Round Robin 은 짧은 응답 시간이 중요한 시스템에서 유용하다.

Round Robin 스케줄링에서 타임 퀀텀을 조절하는 것은 성능에 매우 큰 영향을 미친다. 퀀텀의 크기가 매우 크다면, FCFS 스케줄링 알고리즘과 같아지게 되어 프로세스가 오래 CPU 를 독점할 수 있다. 반면, 타임 퀀텀의 크기가 너무 작으면 문맥 교환(context switching)이 자주 일어나 스위치 오버헤드가 증가하여, 시스템 자원을 낭비하게 되고, 전체 성능이 저하된다.

타임 퀀텀의 크기는 일반적으로 10ms 에서 100ms 사이로 설정되며, 문맥교환의 시간보다 타임 퀀텀의 크기가 크되, CPU Burst 의 80%는 시간 퀀텀보다 짧은 것이 바람직하다.

Round Robin 은 각각의 프로세스가 공평하게 CPU 를 할당 받기 때문에 응답 시간이 우수하다. 따라서 대화형 운영체제에 유용하다. 그러나 각 프로세스의 CPU 사용 시간의 차이가 큰 경우, 총 처리 시간에서 SJF 스케줄링 알고리즘 보다 성능이 떨어질 수 있다. 또 잦은 문맥교환에 따른 스위치 오버헤드가 증가하여 성능 저하를 유발할 수 있다.

3-4. msg queue

본 스케줄링 시뮬레이터에서는 이러한 메시지 큐의 시스템 콜인 msgget, msgsnd, msgrcv 를 사용하는 방식보다는 직접 정의한 인터페이스를 통해 부모 프로세스로부터 자식 프로세스의 스케줄링을 제어한다. 이를 통해, 단순한 사용자 정의 함수 호출만으로 시뮬레이터의 성능을 더 향상시킬 수 있다. 그러나 해당 인터페이스를 통해서도 스케줄링을 구현할 수 있는 방법이 있기에 본 장에서는 해당 개념에 대해 설명한다.

메시지 큐는 프로세스간 통신 기법 IPC 의 한 종류로, 독립적인 메모리를 가진 프로세스들이 필요한 정보를 주고받을 수 있도록 한다. FIFO(First In First Out) 방식으로 메시지를 처리하며, 먼저 들어온 메시지가 먼저 처리되도록 보장한다. 메시지 큐의 비동기 통신은 프로세스들이 동시에 작동하지 않아도, 필요한 데이터를 주고받을 수 있어 통신 오버헤드를 줄일 수 있고, CPU 를 효율적으로 사용하도록 한다.

메시지 큐의 생성과 접근을 위한 msgget, msgsnd, msgrcv 와 같은 시스템 콜을 사용하기 위해 필요한 헤더 파일은 다음과 같다.

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
```

Fig 7. IPC Message Header

다음은 메시지 큐를 구현하기 위해 사용한 시스템 콜과 그 설명이다.

3-4-1. msgget

```
int msgget(key_t key, int msgflg);
```

Fig 8. Declaration of msgget

msgget 시스템 콜은 메시지 큐를 생성하거나 기존 메시지 큐에 접근할 때 사용하는 함수이다. Fig 8 과 같이 인자로 key 값과 msgflg 를 받아 사용된다.

-Key: 메시지 큐를 식별하는데 사용되는 키로, IPC_PRIVATE 값이거나 ftok 함수를 통해 생성된 키를 지정할 수 있다.

-msgflg: 메시지 큐의 속성을 지정하는 플래그로, 다음과 같은 옵션이 포함된다.

1. IPC_CREAT: 지정된 키에 연관된 ID 가 존재하지 않을 경우 새로운 메시지 큐를 생성한다.
2. IPC_EXCL: 지정된 키에 연관된 ID 가 이미 존재하면 오류를 발생시키며, 새로운 큐가 생성되지 않는다.

Msgget 시스템 콜은 이 두개의 인자를 받아 메시지 큐 식별자를 반환하며, 큐가 존재하지 않을 경우 새로 생성하여 큐의 ID 와 IPC 구조체를 초기화한다.

메시지 큐가 새로 생성되는 조건은 다음과 같다.

1. Key 값이 IPC_PRIVATE 로 설정된 경우 메시지 큐가 생성된다.
2. Key 값이 IPC_PRIVATE 가 아니면서, 기존 메시지 큐가 존재하지 않고 msgflg 가 IPC_CREAT 로 설정된 경우 메시지 큐가 생성된다.

이를 통해 msgget 시스템 콜을 사용하여 기존 큐에 접근하거나 새 메시지 큐를 생성할 수 있다. 이때 msgflg 플래그 설정에 따라, 키에 해당하는 큐가 없을 경우 새로운 큐를 생성할지 여부가 결정된다.

3-4-2. msgsnd

```
int msgsnd(int msgid, const void *msgp, size_t msgsz, int msgflg);
```

Fig 9. Declaration of msgsnd

msgsnd 시스템 콜은 메시지를 메시지 큐에 전송하는 함수이다. 프로세스 간 통신을 위해 필요한 데이터를 msgget 으로 생성한 메시지 큐에 추가할 때 사용된다.

Fig 9 와 같이 인자로 메시지 ID, 메시지를 보낼 버퍼, 메시지 사이즈, msgflg 를 받아 사용된다.

-msgid: msgget 시스템 콜을 통해 생성된 메시지 큐 식별자이다.

-msgp: 메시지 내용을 저장한 버퍼의 주소이다.

-msgsz: 메시지의 크기를 나타낸다. 0 에서부터 시스템에서 설정한 최대값까지 지정 가능하다.

-msgflg: 메시지 전송 방식을 나타낸다. 0 으로 설정하면 블로킹 모드로, 메시지 큐가 가득 차면 공간이 생길 때까지 대기한다. IPC_NOWAIT 으로 설정하면 비블로킹 모드로, 메시지 큐가 가득 찬 경우 대기하지 않고 오류를 반환한다.

msgsnd 시스템콜을 사용하여 메시지 큐에 메시지를 전송할 수 있다. 큐가 가득 찼을 경우, msgflg 설정에 따라 블로킹, 비 블로킹 모드로 동작한다.

3-4-3. msgrcv

```
ssize_t msgrcv(int msqid, void *msgq, size_t msgsz, long msgtyp, int msgflg);
```

Fig 10. Declaration of msgrcv

msgrcv 시스템 콜은 메시지 큐에서 메시지를 수신하는 함수로서, 프로세스 간 통신을 할 때, 메시지 큐에서 데이터를 읽어오기 위해 사용한다.

Fig 10 과 같이 인자로 메시지 ID, 메시지를 받을 버퍼, 메시지 사이즈, 메시지 유형, msgflg 플래그를 받아 사용된다.

-msgid: msgget 시스템콜을 통해 생성된 메시지 큐 식별자이다.

-msgp: 메시지 내용을 저장한 버퍼의 주소이다.

-msgsz: 메시지의 크기를 나타낸다. 0 에서부터 시스템에서 설정한 최대값까지 지정 가능하다.

-msgtyp: 수신 받을 메시지의 유형을 지정한다. 0 으로 설정하면 큐에 존재하는 첫 번째 메시지를 읽고, 다른 값으로 설정하면 해당 값에 맞는 유형의 메시지를 우선으로 읽는다.

-Msgflg: 메시지 수신 방식을 나타낸다. 0 으로 설정하면 블로킹 모드로, 큐에 읽고자 하는 메시지가 없으면 해당 메시지가 도착할 때까지 대기한다. IPC_NOWAIT 으로 설정하면 비블로킹 모드로, 큐에 읽고자 하는 메시지가 없을 경우 대기하지 않고 오류를 반환한다.

msgrcv 시스템콜을 사용하여 메시지 큐에서 메시지를 수신할 수 있다. 원하는 메시지가 큐에 존재하지 않는 경우, msgflg 설정에 따라 블로킹, 비블로킹 모드로 동작한다.

4. Implements

본 장에서는 Simple Scheduling 을 구현하는데 있어서 생각한 아이디어에 대해 제시하고, 완성된 프로그램에 대한 코드를 설명한다.

4-1. Idea for Implementation

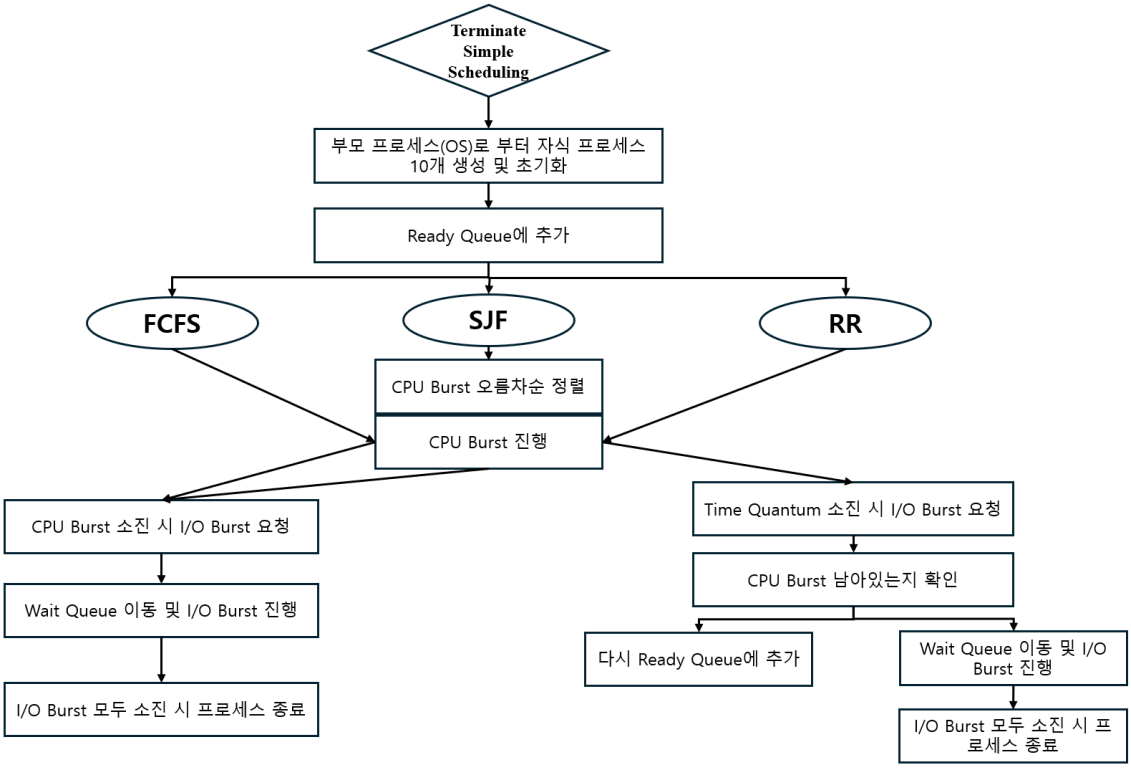


Fig 11. Process of Terminate Scheduling Ver.

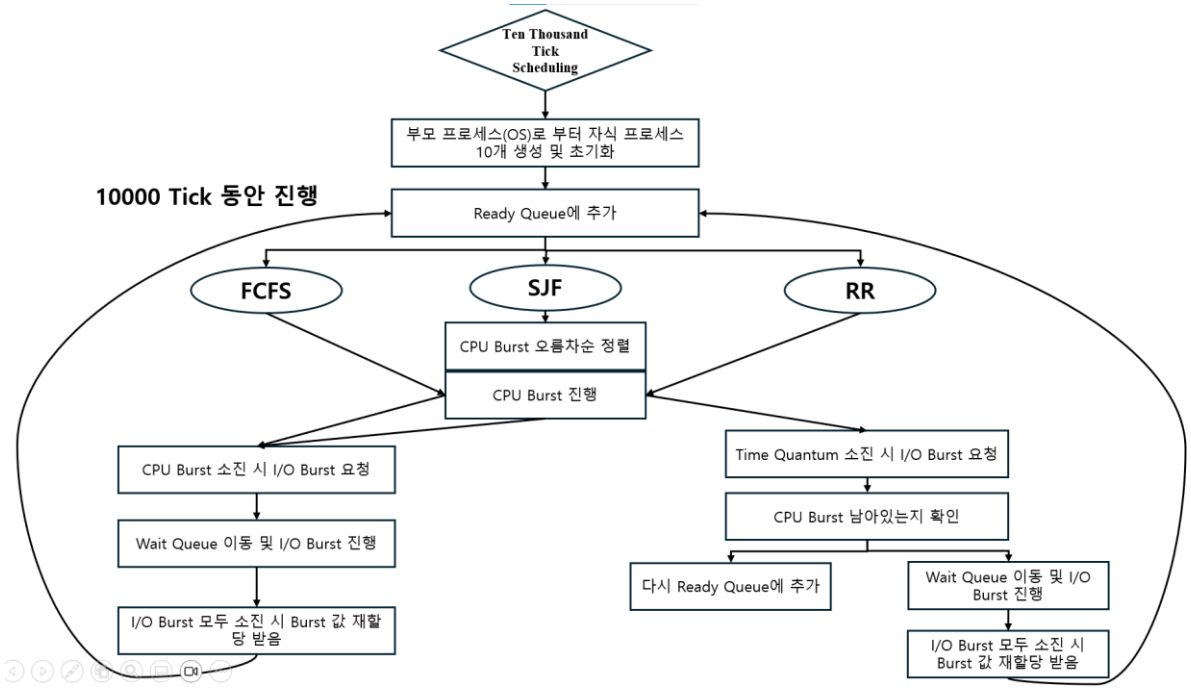


Fig 12. Process of Ten Thousand Tick Scheduling Ver.

본 레포트에서는 두 가지 스케줄링 버전(Fig 11 과 Fig 12)을 구현하였으며, FCFS, SJF, RR 스케줄링 정책을 적용하였다. Fig 11 에 해당하는 Terminate Version 에서는 CPU Burst 와 I/O Burst 를 모두 소진한 후 프로세스를 종료하도록 설계하였다. 이 버전의 순서도에 따르면, 먼저 부모 프로세스는 10 개의 자식 프로세스를 생성하여 Ready Queue 에 추가한다. 이후 각 스케줄링 정책에 따라 프로세스는 서로 다른 방식으로 실행된다.

FCFS(First Come, First Serve)의 경우 Ready Queue 에 도착한 순서대로 CPU Burst 를 진행하며, CPU Burst 가 소진되면 I/O 요청을 하고 Wait Queue 로 이동하여 I/O Burst 를 소진한 후 종료된다. SJF(Shortest Job First)는 먼저 CPU Burst 값이 작은 프로세스를 우선적으로 실행하도록 정렬한 후, FCFS 와 동일하게 CPU Burst 를 소진한 뒤 I/O 작업을 거쳐 종료한다. Round Robin(RR)은 Time Quantum 이 모두 소진되면 CPU Burst 잔여 여부를 확인하여 남아있는 경우 다시 Ready Queue 에 추가하며, CPU Burst 를 모두 소진했을 때에는 I/O 작업을 수행한다. Time Quantum 이 모두 소진되기 전에 CPU Burst 가 끝나면 즉시 I/O 요청을 하도록 하여 프로세스를 효율적으로 관리한다.

Fig 12 는 Ten Thousand Tick Version 을 나타내며, CPU Burst 와 I/O Burst 를 모두 소진하더라도 프로세스를 종료하지 않고 새로운 Burst 값을 할당 받아 다시 스케줄링을 반복하도록 설계되었다. 이를 통해 프로세스가 10,000 Tick 동안 지속적으로 실행될 수 있도록 하며, 반복되는 스케줄링 과정을 통해 각 정책이 긴 시간 동안의 시스템 성능에 어떤 영향을 미치는지 평가할 수 있도록 한다.

4-2. Implement Program

본 절에서는 제시한 아이디어를 바탕으로 구현한 프로그램의 코드를 설명한다.

4-2-1. Build environment

먼저 구현한 코드 설명에 앞서 해당 프로그램을 개발한 IDE, 프로그램 실행하기 위한 환경은 다음과 같다.

- ✔ 개발 환경: Ubuntu 22.04, VSCode
- ✔ 프로그래밍 언어: C
- ✔ 프로그램 실행 방식

  STEP 1) make

  STEP 2) ./scheduler <SCHEDULING POLICY> <TIME TICK(us)> <BURST_LIMIT>

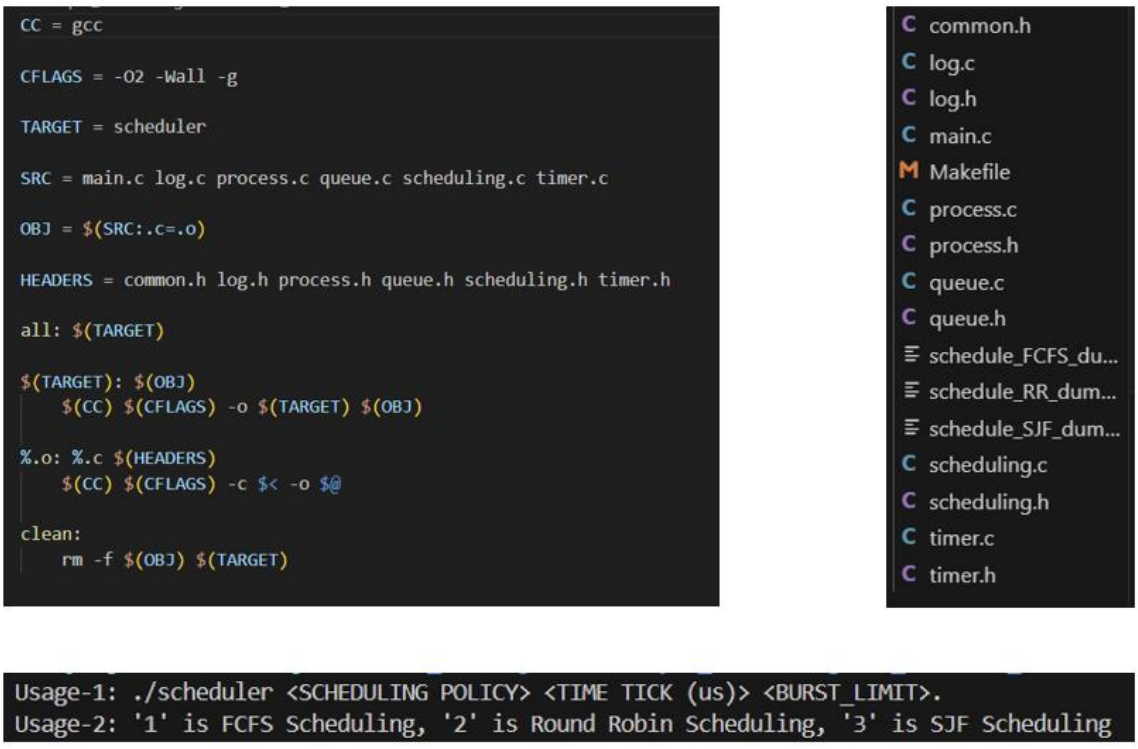


Fig 13. Build Environment

4-2-2. Terminate Version

```c
#pragma once

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <signal.h>
#include <string.h>
#include <time.h>
#include <errno.h>

#include <sys/wait.h>
#include <sys/time.h>
#include <sys/types.h>

#define MAX_PROCESSES 10

extern int counter;
extern int wait_proc_count;
extern int turn_proc_count;
extern int burst_limit;

extern int time_count;

extern int time_tick;
extern int time_quantum;

extern double wait_time;
extern double turnaround_time;
extern double total_wait_time;

extern int schedule_policy;
```

Fig 14. Code of Common.h

Fig 14 에서는 프로그램에서 사용할 공통 라이브러리와 변수들이 선언되어 있으며, 이를 통해 스케줄링 알고리즘의 기본 설정과 작업 처리가 원활히 이루어지도록 설계하였다. 먼저, MAX_PROCESSES 를 10 으로 정의하여 최대 프로세스 수를 설정하였으며, 스케줄링 로그 출력을 위한 counter 변수를 두어 로그 출력을 관리하도록 하였다.

스케줄링 기준(Scheduling Criteria)을 계산하기 위해, 각 프로세스의 대기 시간과 반환 시간을 계산할 수 있도록 wait_proc_count와 turn_proc_count 변수를 선언하고, 이를 통해 wait_time과

turnaround_time 을 누적할 수 있도록 하였다. 이외에도 CPU Burst 와 I/O Burst 의 제한을 설정하기 위해 burst_limit 변수를 두어 각 프로세스의 최대 Burst 시간을 관리하였다.

시간에 따른 작업 처리를 위해 time_count, time_tick, time_quantum 변수를 선언하여, 각 타임 틱(time tick)마다 스케줄링 작업이 적절히 수행되도록 하였다. 특히, time_quantum 을 통해 Round Robin 스케줄링의 타임 퀀텀 값을 설정하여 프로세스들이 고르게 CPU 시간을 할당받을 수 있도록 하였다. 마지막으로, schedule_policy 변수를 통해 현재 실행 중인 스케줄링 정책을 설정하여, 각 스케줄링 기법이 적절히 동작할 수 있게 하였다.

```c
#pragma once

#include "common.h"

typedef enum{
    NEW,
    READY,
    RUNNING,
    WAITING,
    TERMINATED
} ProcessState;

typedef struct{
    int pid;
    int cpu_burst;
    int io_burst;
    int remaining_time;

    double arrival_time;
    double start_time;

    int flag;
    ProcessState state;
} Process;

Process* create_process(int pid, int cpu_burst, int io_burst, int time_tick);
void update_process_state(Process *process, ProcessState new_state);
void terminate_process(Process *process);
```

Fig 15. Code of process.h

Fig 15 에는 프로세스 관리와 관련된 구조체와 함수들이 정의되어 있으며, 이를 통해 스케줄링 과정에서 프로세스의 정보를 효과적으로 관리하고 상태 변화를 처리할 수 있도록 구현하였다. 먼저, 프로세스를 고유하게 식별하기 위해 pid 를 포함한 구조체가 정의되어 있으며, 각 프로세스의 CPU Burst 와 I/O Burst 시간을 관리하기 위한 cpu_burst 와 io_burst 변수가 포함되어 있다. 또한 Round Robin 스케줄링에서 Time Quantum 을 소진할 때까지 남은 시간을 추적하기 위해 remaining_time 변수를 추가하였다.

프로세스의 Ready Queue 도착 시간을 기록하는 arrival_time 과 CPU 점유 시작 시간을 기록하는 start_time 변수를 통해, Wait Time 과 Turnaround Time 을 정확히 계산할 수 있도록 설계하였다. 더불어, 프로세스가 CPU 에서 처음 시작될 때만 Wait TIme 을 계산하기 위해 flag 변수를 추가하고, 프로세스의 현재 상태를 추적하는 state 변수를 통해 Ready, Running, Waiting, Terminated 와 같은 상태를 관리할 수 있게 하였다.

이 구조체와 함께, 세 가지 주요 함수가 정의되어 있다. create_process 함수는 새로운 프로세스를 생성하고 초기화하여 Ready Queue 에 추가하며, 각 프로세스의 pid, cpu_burst, io_burst, arrival_time 등의 초기 값을 설정한다. update_process_state 함수는 프로세스의 상태가 변경될 때 호출되어, state 값을 업데이트하고 상태에 따라 필요한 작업을 수행한다. 예를 들어, CPU Burst 가 소진되었을 때 I/O 작업을 수행하거나 Ready Queue 로 되돌리는 처리를 담당한다. 마지막으로, terminate_process 함수는 프로세스가 모든 작업을 완료했을 때 호출되며, 프로세스를 종료하고 스케줄링에서 제거하는 역할을 한다.

```c
#pragma once

#include "common.h"
#include "process.h"

typedef struct Node{
    Process pcb;
    struct Node* next;
} Node;

typedef struct{
    Node* head;
    Node* tail;
    int count;
} Queue;

Queue* createQueue();
int isEmpty(Queue *q);
void enqueue(Queue *q, int pid, int cpu_burst, int io_burst, double arrival_time, int remaining_time);
Process* dequeue(Queue *q);
void sort_queue(Queue *q);
void removeQueue(Queue *q);
```

Fig 16. Code of queue.h

Fig 16 에서는 프로세스를 효율적으로 관리하기 위해 Ready Queue 와 Wait Queue 를 구현하는 코드가 작성되어 있으며, 이를 통해 스케줄링 과정에서 필요한 프로세스 대기열을 관리할 수 있도록 하였다. 먼저, 각 프로세스를 연결 리스트 형태로 관리하기 위해 Node 구조체를 정의하였고, 이를 활용하여 프로세스를 담는 Queue 구조체를 설계하였다.

Queue 와 관련된 주요 함수들로는 다음과 같은 것들이 있다. 먼저, createQueue 함수는 새로운 Queue 를 생성하여 초기화하며, isEmpty 함수는 Queue 가 비어 있는지 여부를 확인하여 스케줄링 과정에서 대기열의 상태를 쉽게 파악할 수 있도록 한다.

Queue 에 프로세스를 추가하기 위해 enqueue 함수를 두었으며, 이 함수는 새로운 Node 를 Queue 의 끝에 추가하여 Ready Queue 나 Wait Queue 에 프로세스를 배치할 수 있게 한다. 반대로, Queue 의 맨 앞 노드를 제거하는 dequeue 함수는 프로세스가 Ready Queue 나 Wait Queue 에서 나갈 때 호출되어, 해당 노드를 제거하고 다음 프로세스를 스케줄링할 수 있도록 한다. 추가적으로 본 레포트에서는 가장 맨 앞 노드를 현재 CPU 를 점유하거나 I/O 처리를 하는 프로세스로 간주하였다.

또한, SJF(Shortest Job First) 스케줄링 정책에서는 CPU Burst 가 작은 순서대로 프로세스를 실행해야 하기 때문에 sort_queue 함수를 통해 Queue 에 저장된 프로세스들을 버블 정렬 방식으로 정렬하였다. 이로써 SJF 스케줄링의 특성을 반영할 수 있다. 마지막으로, 스케줄링이 완료된 후 Queue 를 완전히 제거하기 위해 removeQueue 함수를 정의하여, 모든 노드를 삭제하고 메모리를 해제하도록 하였다.

```c
static struct itimerval timer;

void initialize_timer(void (*handler)(int)){
    struct sigaction sa;
    sa.sa_handler = handler;
    sa.sa_flags = 0;
    sigemptyset(&sa.sa_mask);

    if(sigaction(SIGALRM, &sa, NULL) == -1){
        perror("Error setting signal handler");
        exit(EXIT_FAILURE);
    }

    timer.it_interval.tv_sec = 0;
    timer.it_interval.tv_usec = time_tick;

    timer.it_value.tv_sec = 0;
    timer.it_value.tv_usec = time_tick;
}

void start_timer(){
    if(setitimer(ITIMER_REAL, &timer, NULL) == -1){
        perror("Error starting timer");
        exit(EXIT_FAILURE);
    }
}

void stop_timer(){
    struct itimerval stop = {0};
    if (setitimer(ITIMER_REAL, &stop, NULL) == -1){
        perror("Error stopping timer");
        exit(EXIT_FAILURE);
    }
}
```

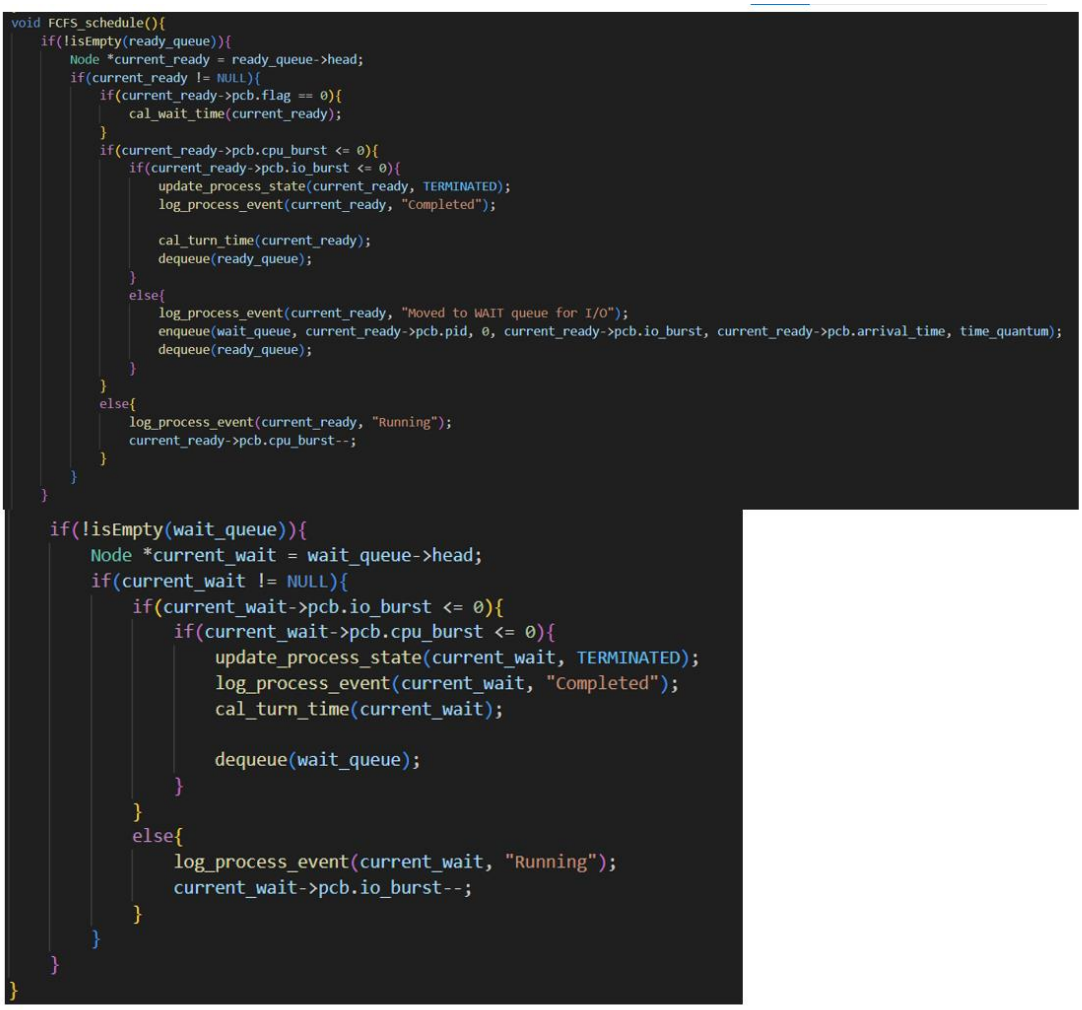
Fig 17. Code of timer.c

Fig 17 에서는 timer.c 파일을 구현하여 스케줄링이 일정한 주기로 신호(Signal)를 보낼 수 있도록 설정하였다. 이를 통해 프로세스의 상태를 주기적으로 업데이트하고, 스케줄링 정책에 따라 프로세스를 관리할 수 있게 하였다.

먼저, 타이머 인터럽트 발생 시 실행될 루틴을 정의하기 위해 sigaction 구조체를 사용하였다. 이 구조체를 통해 타이머 인터럽트가 발생할 때 호출되는 핸들러 함수를 지정하여, 주기적인 스케줄링

처리가 가능하도록 하였다. 적절한 타이머 주기를 설정하기 위해 time_tick 변수를 정의하여, 타이머가 정해진 주기마다 프로세스 스케줄링을 트리거하도록 설정하였다.

start_timer 함수는 타이머를 시작하는 역할을 하며, 타이머가 활성화되면 지정된 time_tick 간격마다 인터럽트가 발생하도록 한다. 이를 통해 스케줄링 정책에 따라 Ready Queue 에서 프로세스를 선택하고, CPU 할당을 조정하는 등의 작업이 자동으로 이루어진다. 반면, stop_timer 함수는 타이머를 종료하여 더 이상 인터럽트가 발생하지 않도록 함으로써 스케줄링을 멈추는 역할을 한다.



```
void FCFS_schedule(){
    if(!isEmpty(ready_queue)){
        Node *current_ready = ready_queue->head;
        if(current_ready != NULL){
            if(current_ready->pcb.flag == 0){
                cal_wait_time(current_ready);
            }
            if(current_ready->pcb.cpu_burst <= 0){
                if(current_ready->pcb.io_burst <= 0){
                    update_process_state(current_ready, TERMINATED);
                    log_process_event(current_ready, "Completed");

                    cal_turn_time(current_ready);
                    dequeue(ready_queue);
                }
                else{
                    log_process_event(current_ready, "Moved to WAIT queue for I/O");
                    enqueue(wait_queue, current_ready->pcb.pid, 0, current_ready->pcb.io_burst, current_ready->pcb.arrival_time, time_quantum);
                    dequeue(ready_queue);
                }
            }
            else{
                log_process_event(current_ready, "Running");
                current_ready->pcb.cpu_burst--;
            }
        }
    }

    if(!isEmpty(wait_queue)){
        Node *current_wait = wait_queue->head;
        if(current_wait != NULL){
            if(current_wait->pcb.io_burst <= 0){
                if(current_wait->pcb.cpu_burst <= 0){
                    update_process_state(current_wait, TERMINATED);
                    log_process_event(current_wait, "Completed");
                    cal_turn_time(current_wait);

                    dequeue(wait_queue);
                }
            }
            else{
                log_process_event(current_wait, "Running");
                current_wait->pcb.io_burst--;
            }
        }
    }
}
```

Fig 18. FCFS Code of Terminate Ver.

Fig 18 에는 FCFS(First-Come, First-Serve) 스케줄링을 구현한 코드가 포함되어 있으며, 이 코드에서는 Ready Queue 와 Wait Queue 를 활용하여 프로세스의 CPU Burst 와 I/O Burst 를 관리한다. 먼저, CPU 를 점유할 프로세스를 Ready Queue 의 head 에서 가져와 스케줄링을 시작한다.

프로세스가 CPU 를 점유하게 되면, 대기 시간 계산을 위해 대기 시간 계산 함수가 호출된다. 이후, CPU Burst 가 모두 소진되면 I/O Burst 를 확인하여 다음 단계로 진행한다.

만약 I/O Burst 도 모두 소진되었다면, 프로세스를 TERMINATED 상태로 업데이트하고, Turnaround Time 을 계산하여 프로세스의 전체 소요 시간을 기록한다. 그런 다음, dequeue 를 호출하여 Ready Queue 에서 해당 프로세스를 제거하고 종료한다. 반면, CPU Burst 가 소진되었으나 I/O Burst 가 남아 있다면, 프로세스를 Wait Queue 로 이동시키고 Ready Queue 에서는 삭제하도록 한다. 만약 CPU Burst 가 아직 남아 있는 경우에는 CPU Burst 값만 감소시키며 프로세스를 유지한다.

I/O 작업은 Wait Queue 에서 처리되며, 여기서 I/O Burst 가 모두 소진된 경우 프로세스는 종료된다. I/O Burst 가 남아 있는 경우에는 단순히 I/O Burst 값을 감소시키며 프로세스의 I/O 작업을 계속 수행한다.

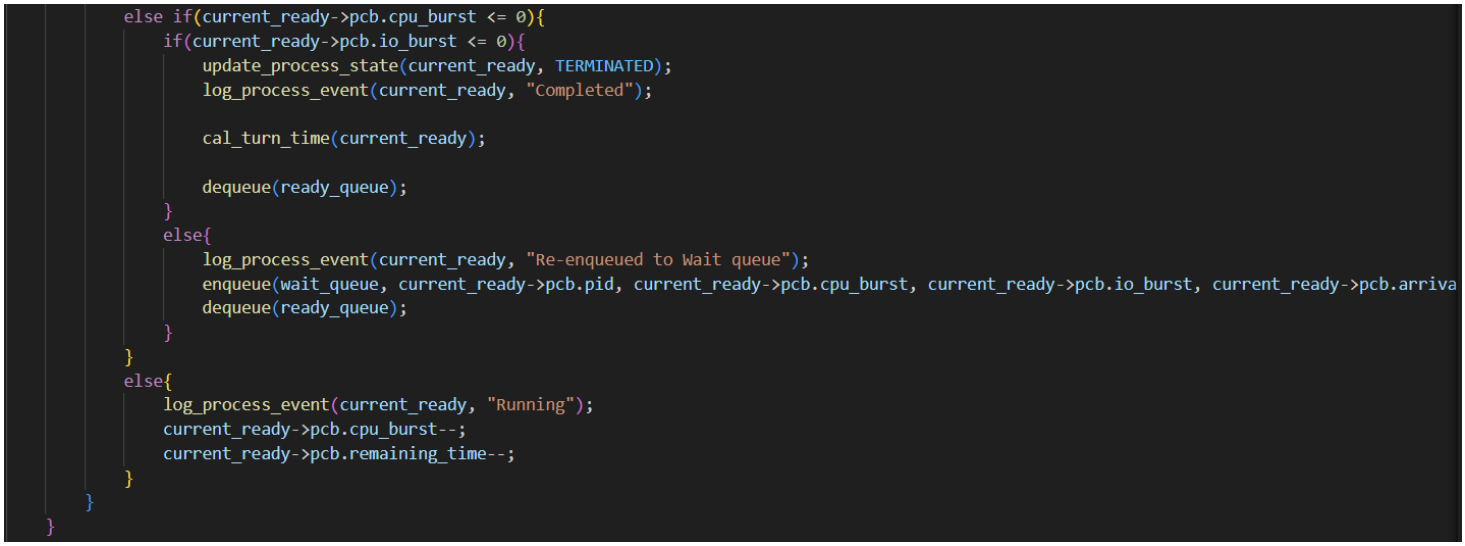
```c
if(schedule_policy == 3){
    sort_queue(ready_queue);
}
```

Fig 19. SJF Code of Terminate Ver.

SJF 스케줄링에 대한 로직은 FCFS 스케줄링과 유사하여 차이점만 설명한다. Fig 19 에 SJF 에서 CPU Burst 값이 작은 순서대로 정렬하기 위한 sort_queue 함수 호출이 이루어진 후, 로직이 진행된다.

```c
void RR_schedule(){
    if(!isEmpty(ready_queue)) {
        Node *current_ready = ready_queue->head;
        if(current_ready != NULL){
            if(current_ready->pcb.flag == 0){
                cal_wait_time(current_ready);
            }

            if(current_ready->pcb.remaining_time == 0){
                if(current_ready->pcb.cpu_burst <= 0){
                    if(current_ready->pcb.io_burst <= 0){
                        update_process_state(current_ready, TERMINATED);
                        log_process_event(current_ready, "Completed");
                        cal_turn_time(current_ready);
                        dequeue(ready_queue);
                    }
                    else{
                        log_process_event(current_ready, "Moved to WAIT queue for I/O");
                        enqueue(wait_queue, current_ready->pcb.pid, 0, current_ready->pcb.io_burst, current_ready->pcb.arrival_time, time_quantum);
                        dequeue(ready_queue);
                    }
                }
                else{
                    log_process_event(current_ready, "Re-enqueued to Ready queue");
                    enqueue(ready_queue, current_ready->pcb.pid, current_ready->pcb.cpu_burst, current_ready->pcb.io_burst, current_ready->pcb.arriv
                    dequeue(ready_queue);
                }
            }
```

Fig 20. RR Code (1) of Terminate Ver.



```
        else if(current_ready->pcb.cpu_burst <= 0){
            if(current_ready->pcb.io_burst <= 0){
                update_process_state(current_ready, TERMINATED);
                log_process_event(current_ready, "Completed");

                cal_turn_time(current_ready);

                dequeue(ready_queue);
            }
            else{
                log_process_event(current_ready, "Re-enqueued to Wait queue");
                enqueue(wait_queue, current_ready->pcb.pid, current_ready->pcb.cpu_burst, current_ready->pcb.io_burst, current_ready->pcb.arriva
                dequeue(ready_queue);
            }
        }
        else{
            log_process_event(current_ready, "Running");
            current_ready->pcb.cpu_burst--;
            current_ready->pcb.remaining_time--;
        }
    }
}
```

Fig 21. RR Code (2) of Terminate Ver.

Fig 20 과 Fig 21 에는 Round Robin 스케줄링을 구현한 코드가 포함되어 있으며, 이 코드에서는 Time Quantum 을 기반으로 각 프로세스에 CPU 를 할당하고, 주어진 Quantum 이 소진되면 프로세스를 재조정하는 방식으로 동작한다. 먼저, 현재 프로세스의 Time Quantum 잔여 시간을 확인하여, 잔여 시간이 0 일 경우 CPU Burst 와 I/O Burst 값을 점검하여 종료 또는 재조정을 결정한다.

Time Quantum 이 모두 소진된 상황에서 CPU Burst 가 남아 있지만 I/O Burst 가 모두 소진되지 않았다면, 프로세스를 Wait Queue 로 이동시켜 I/O 작업을 진행하게 한다. 반면, CPU Burst 가 아직 남아 있다면, Ready Queue 로 프로세스를 다시 이동시켜 Round Robin 방식으로 CPU 할당을 재개하도록 한다.

또한, Time Quantum 이 모두 소진되지 않았더라도 CPU Burst 가 모두 소진된 경우에는 I/O Burst 를 확인하여, I/O 작업이 필요한 경우 Wait Queue 로 이동시키고, 더 이상 처리할 Burst 가 남아 있지 않다면 프로세스를 종료하도록 한다.

모든 조건에 해당되지 않는 경우에는 CPU Burst 값과 Time Quantum 의 잔여 시간을 단순히 감소시켜, CPU 할당 시간을 조금 더 유지하도록 한다.

```
void cal_wait_time(Node *current_node){
    wait_proc_count++;
    current_node->pcb.flag = 1;
    current_node->pcb.start_time = time_count;
    wait_time = current_node->pcb.start_time - current_node->pcb.arrival_time;
    total_wait_time += wait_time;
}

void cal_turn_time(Node *current_node){
    turn_proc_count++;
    turnaround_time += time_count - current_node->pcb.arrival_time;
}
```

Fig 22. Scheduling Criteria Code

Fig 22 에서는 Wait Time 과 Turnaround Time 을 계산하는 코드가 포함되어 있으며, 이를 통해 각 프로세스의 대기 시간과 반환 시간을 정확히 측정할 수 있다.

먼저, Wait Time 계산을 위해 count 값을 증가시키고 flag 를 1 로 설정하여 해당 프로세스의 대기 시간이 중복 계산되지 않도록 한다. 이후, 프로세스가 실제로 CPU 에서 실행된 시간에서 Ready Queue 에 도착한 시간을 뺌으로써 Wait Time 을 계산하게 된다. 이를 통해 프로세스가 CPU 를 점유하기까지 대기한 총 시간을 구할 수 있다.

Turnaround Time 의 경우에도 count 값을 증가시키며 계산을 시작한다. Turnaround Time 은 프로세스가 Ready Queue 에 도착한 시점부터 종료될 때까지의 총 시간을 의미하므로, 전체 종료 시간에서 Ready Queue 도착 시간을 빼 주는 방식으로 계산한다.

이러한 Wait Time 과 Turnaround Time 의 계산 로직은 스케줄링 알고리즘의 성능을 평가할 때 중요한 지표를 제공하며, 각 프로세스가 얼마나 오랫동안 대기하고 실행되었는지를 명확하게 측정할 수 있도록 한다.

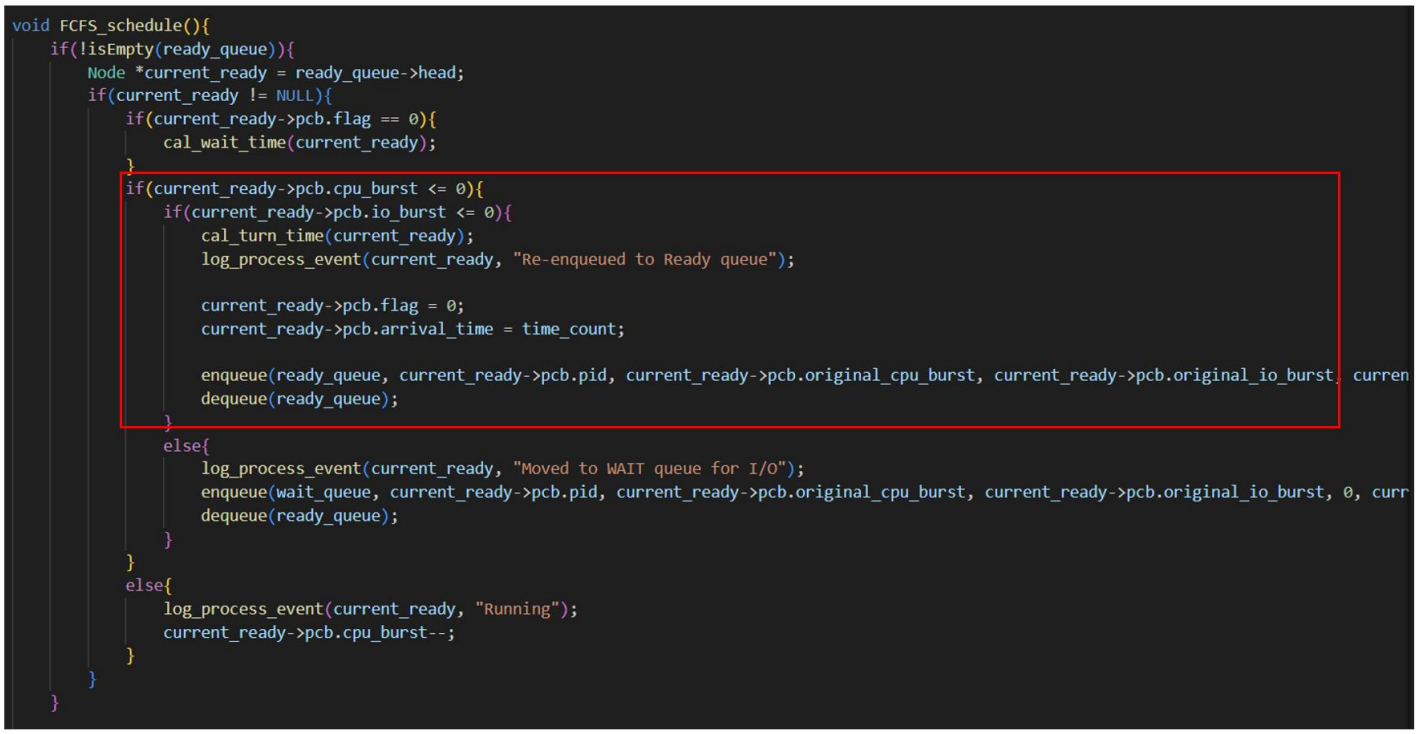
4-2-2. Ten Thousand Tick Version

본 절에서는 전체 코드를 설명하기 보다 Terminate Ver 에서 수정된 부분을 설명하도록 한다.

```c
Process* create_process(int pid, int cpu_burst, int io_burst, int time_tick){
    Process *new_process = (Process*)malloc(sizeof(Process));
    if(new_process == NULL){
        perror("Failed to create process");
        exit(EXIT_FAILURE);
    }

    new_process->pid = pid;

    new_process->original_cpu_burst = cpu_burst;
    new_process->original_io_burst = io_burst;

    new_process->cpu_burst = cpu_burst;
    new_process->io_burst = io_burst;
    new_process->state = NEW;
    new_process->remaining_time = time_tick;


    new_process->arrival_time = 0.0;
    new_process->start_time = 0.0;

    new_process->flag = 0;
    return new_process;
}
```

Fig 23. process.c of Ten Thousand Tick Ver.

Fig 23 에서는 수정된 코드가 강조되어 있으며, 이 코드에서는 Burst 값을 모두 소진했을 때 CPU Burst 와 I/O Burst 값을 다시 할당받도록 설정하여, 프로세스가 반복적으로 스케줄링되는 상황에서도 원래의 Burst 값을 유지할 수 있게 하였다. 이를 위해, 기존의 Burst 값을 보존하는 코드를 추가하여 초기 할당된 CPU Burst 와 I/O Burst 의 값을 저장해 두었다가, 모든 Burst 를 소진한 후에도 이 값들을 재할당하도록 하였다.



```
void FCFS_schedule(){
    if(!isEmpty(ready_queue)){
        Node *current_ready = ready_queue->head;
        if(current_ready != NULL){
            if(current_ready->pcb.flag == 0){
                cal_wait_time(current_ready);
            }
            if(current_ready->pcb.cpu_burst <= 0){
                if(current_ready->pcb.io_burst <= 0){
                    cal_turn_time(current_ready);
                    log_process_event(current_ready, "Re-enqueued to Ready queue");

                    current_ready->pcb.flag = 0;
                    current_ready->pcb.arrival_time = time_count;

                    enqueue(ready_queue, current_ready->pcb.pid, current_ready->pcb.original_cpu_burst, current_ready->pcb.original_io_burst, curren
                    dequeue(ready_queue);
                }
                else{
                    log_process_event(current_ready, "Moved to WAIT queue for I/O");
                    enqueue(wait_queue, current_ready->pcb.pid, current_ready->pcb.original_cpu_burst, current_ready->pcb.original_io_burst, 0, curr
                    dequeue(ready_queue);
                }
            }
            else{
                log_process_event(current_ready, "Running");
                current_ready->pcb.cpu_burst--;
            }
        }
    }
```

Fig 24. FCFS of Ten Thousand Tick Ver.

Fig 24 에서는 FCFS(First-Come, First-Serve) 스케줄링 알고리즘의 수정된 코드가 강조되어 있으며, 이를 통해 10,000 Tick 동안 프로세스가 반복적으로 스케줄링되도록 조정하였다. 기존 코드에서는 모든 CPU Burst 와 I/O Burst 가 소진되면 프로세스를 종료하였으나, 수정된 코드에서는 프로세스를 반복적으로 실행할 수 있도록 flag 와 arrival_time 값을 업데이트하였다. 이를 통해 프로세스가 다시 Ready Queue 로 돌아가도 올바르게 대기 시간과 도착 시간을 유지할 수 있게 하였다.

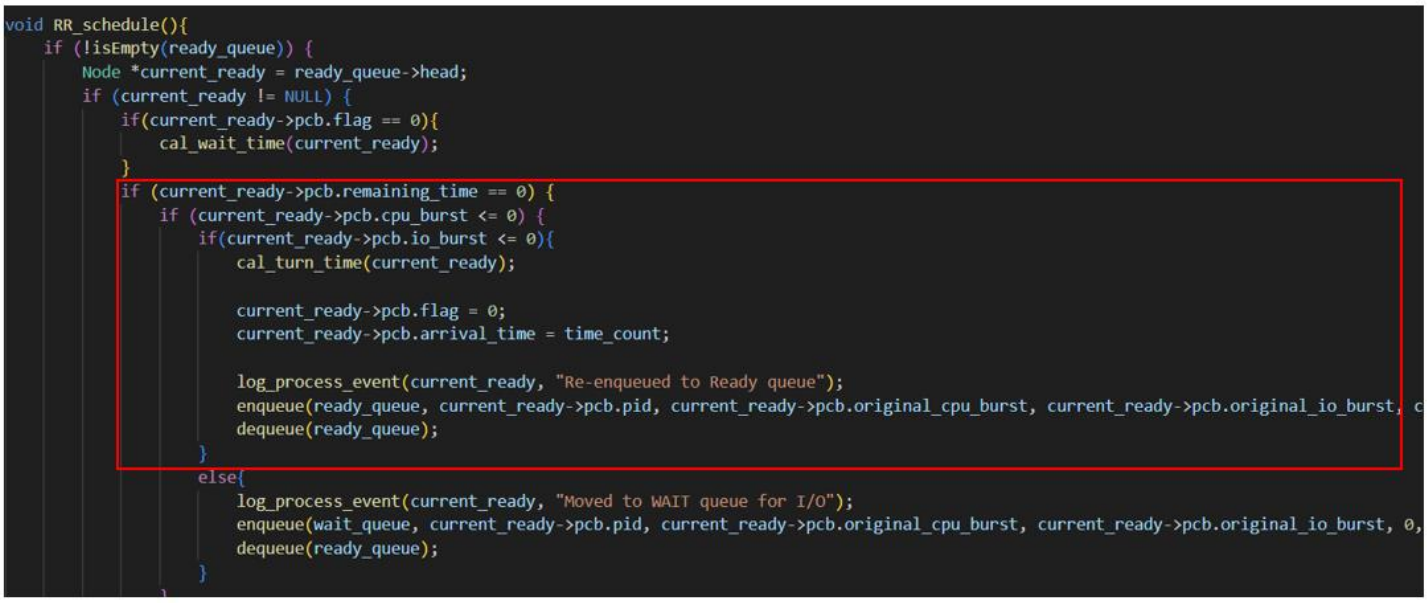
또한, enqueue 시점에 original_cpu_burst 와 original_io_burst 값을 전달하여, 모든 Burst 가 소진된 후에도 초기의 Burst 값을 재사용할 수 있도록 하였다.

```
if(current_ready->pcb.cpu_burst <= 0){
    if(current_ready->pcb.io_burst <= 0){
        cal_turn_time(current_ready);

        log_process_event(current_ready, "Re-enqueued to Ready queue");
        enqueue(ready_queue, current_ready->pcb.pid, current_ready->pcb.original_cpu_burst, current_ready->pcb.original_io_burst, curren
        dequeue(ready_queue);
        sort_queue(ready_queue);
    }
```

Fig 25. SJF of Ten Thousand Tick Ver.

Fig 25 에서는 SJF(Shortest Job First) 스케줄링의 수정된 부분이 강조되어 있으며, 여기서 주목해야 할 점은 sort_queue 함수가 추가되었다는 것이다. 기존 코드에서는 프로세스가 한 번만 실행되고 종료되었기 때문에 Ready Queue 를 초기 상태에서 한 번만 정렬해 주면 충분했다.

그러나 수정된 버전에서는 10,000 Tick 동안 10 개의 프로세스가 CPU 작업과 I/O 작업을 반복적으로 할당받고 소진하는 구조이므로, 각 프로세스의 Burst 값이 모두 소진되면 다시 Burst 값을 할당받게 된다. 이로 인해 Ready Queue 내에서 CPU Burst 값을 기준으로 다시 정렬하는 과정이 필요하게 되었고, 이를 위해 sort_queue 함수가 추가되었다.



```
void RR_schedule(){
    if (!isEmpty(ready_queue)) {
        Node *current_ready = ready_queue->head;
        if (current_ready != NULL) {
            if(current_ready->pcb.flag == 0){
                cal_wait_time(current_ready);
            }
            if (current_ready->pcb.remaining_time == 0) {
                if (current_ready->pcb.cpu_burst <= 0) {
                    if(current_ready->pcb.io_burst <= 0){
                        cal_turn_time(current_ready);

                        current_ready->pcb.flag = 0;
                        current_ready->pcb.arrival_time = time_count;

                        log_process_event(current_ready, "Re-enqueued to Ready queue");
                        enqueue(ready_queue, current_ready->pcb.pid, current_ready->pcb.original_cpu_burst, current_ready->pcb.original_io_burst, c
                        dequeue(ready_queue);
                    }
                    else{
                        log_process_event(current_ready, "Moved to WAIT queue for I/O");
                        enqueue(wait_queue, current_ready->pcb.pid, current_ready->pcb.original_cpu_burst, current_ready->pcb.original_io_burst, 0,
                        dequeue(ready_queue);
                    }
```

Fig 26. RR Code of Ten Thousand Ver.

Fig 26 에서는 Round Robin 스케줄링의 수정된 코드가 강조되어 있으며, 기본적인 로직은 FCFS 와 SJF 와 유사하게 구성되어 있다. Round Robin 스케줄링에서는 Time Quantum 을 기반으로

프로세스의 CPU Burst 와 I/O Burst 를 관리하며, 각 프로세스의 Time Quantum 과 Burst 값이 모두 소진되었을 때 반복적으로 재할당하는 방식으로 처리한다.

수정된 코드에서는 기존 로직의 큰 변경 없이, Time Quantum 의 잔여 시간이 모두 소진되었을 때 CPU Burst 와 I/O Burst 값을 다시 할당받도록 하였다.

5. Results

본 장에서는 4 장에서 설명한 코드를 통해 구현한 프로그램의 결과를 확인한다.

5-1. Result of Terminate Version

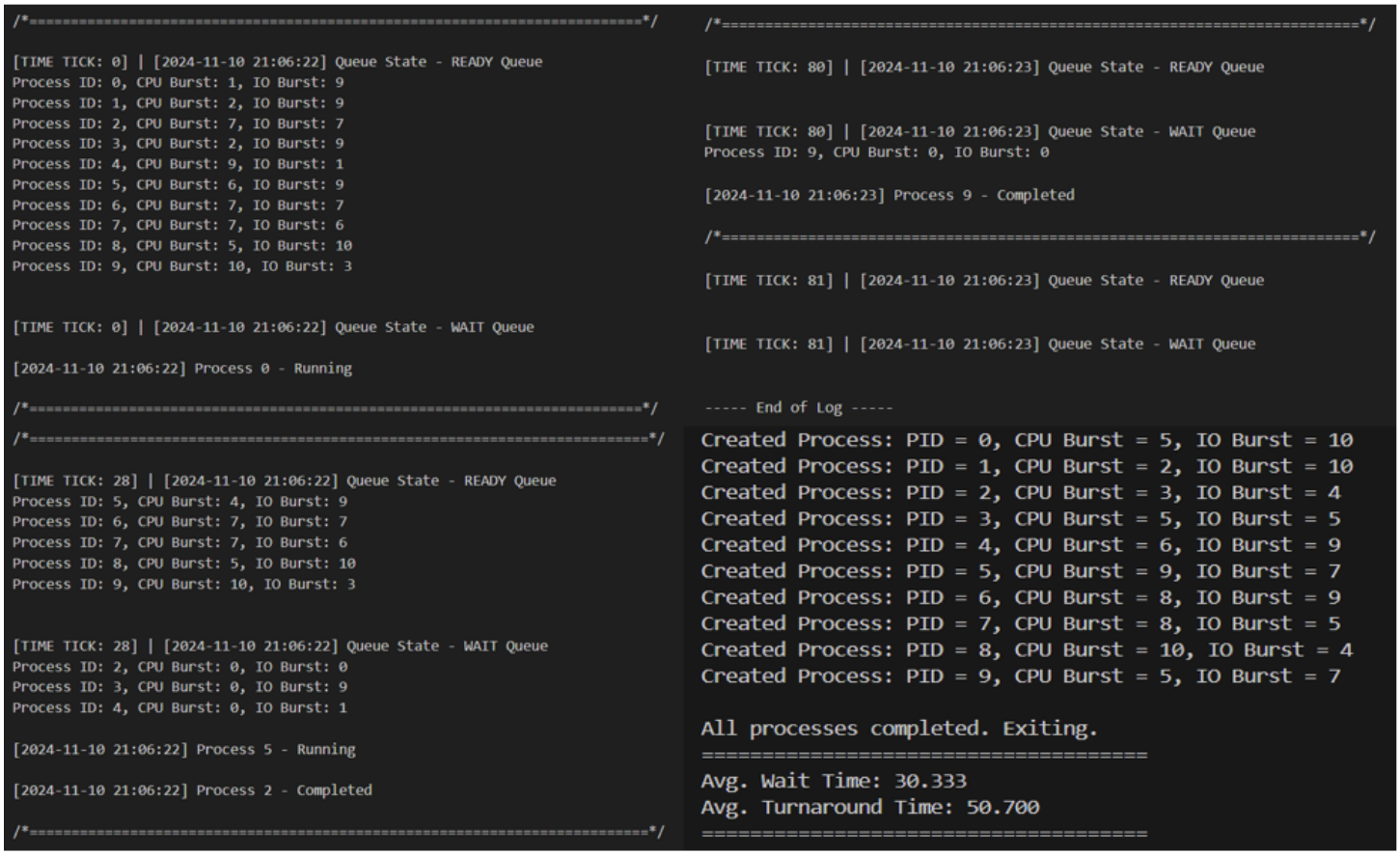


Fig 27. FCFS Results of Terminate Ver.

Fig 27 에 FCFS 스케줄링에 대한 결과가 나타나 있다. Time Tick 순서대로 결과를 살펴보면, 처음에 모든 프로세스들이 Ready Queue 에 할당 되어있고, 이후 작업이 진행되며 마지막에 Queue 가 비워진 것을 확인하면서 스케줄링이 종료되며 Wait Time 과 Turnaround Time 을 확인할 수 있다.

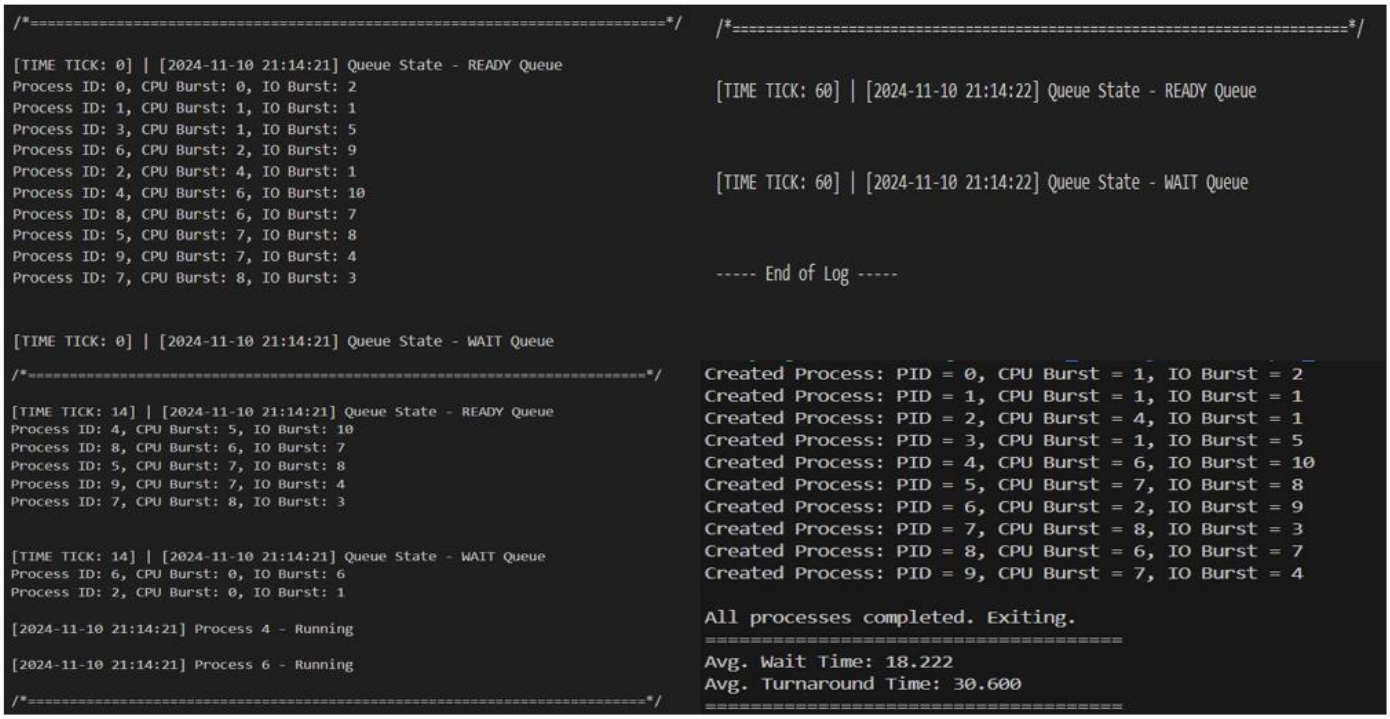


Fig 28. SJF Result of Terminate Ver.

Fig 28 에서 SJF 스케줄링에 대한 결과가 나타나 있다. FCFS 와 비슷한 흐름을 보이지만 주목해야 할 부분은 Wait Time과 Turnaround Time이다. SJF에서는 Wait Time이 줄어들 것으로 기대했지만 해당 결과는 그렇지 못했다.

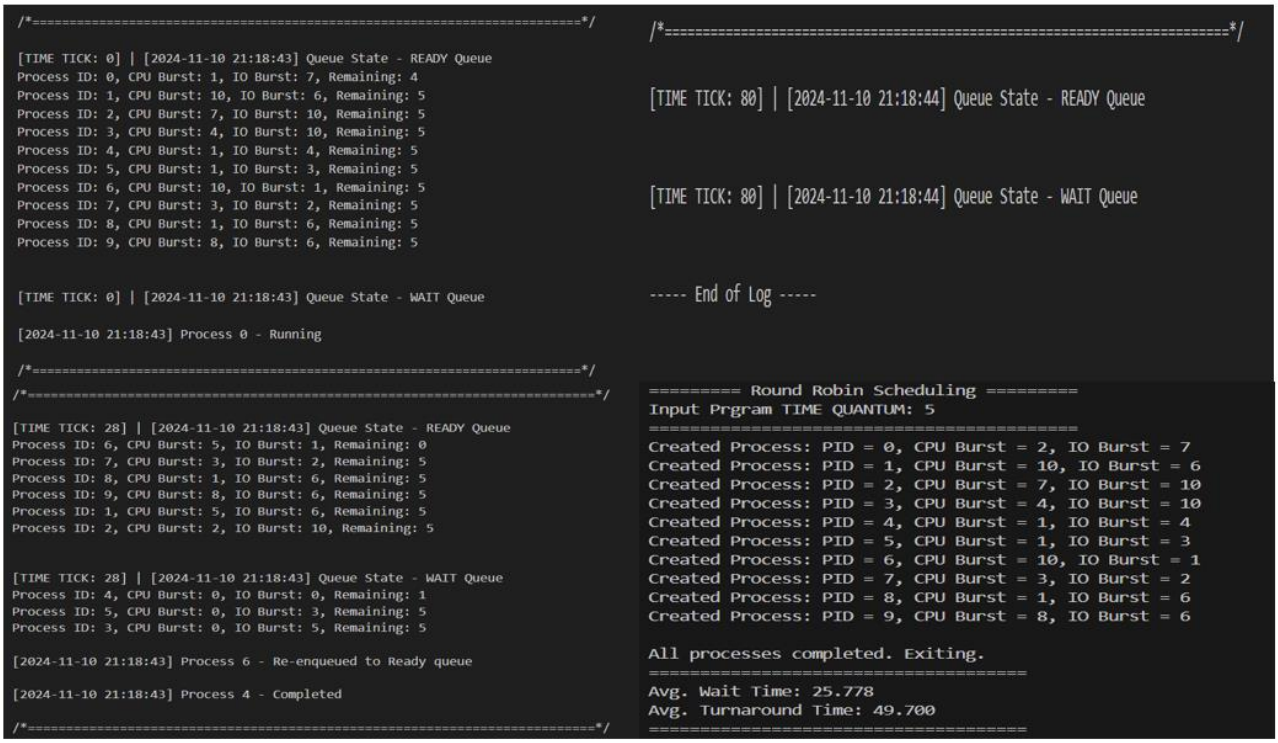


Fig 29. RR Result of Terminate Ver.

Fig 29에 Round Robin 스케줄링에 대한 결과가 나타나 있다. Time Quantum에 따라 프로세스가 Ready Queue와 Wait Queue에서 Burst를 처리하는 것을 확인할 수 있다. 또한 Wait Time과 Turnaround Time이 Fig 28, Fig 29와 유사한 것을 확인할 수 있다.

5-2. Result of Ten Thousand Tick Version

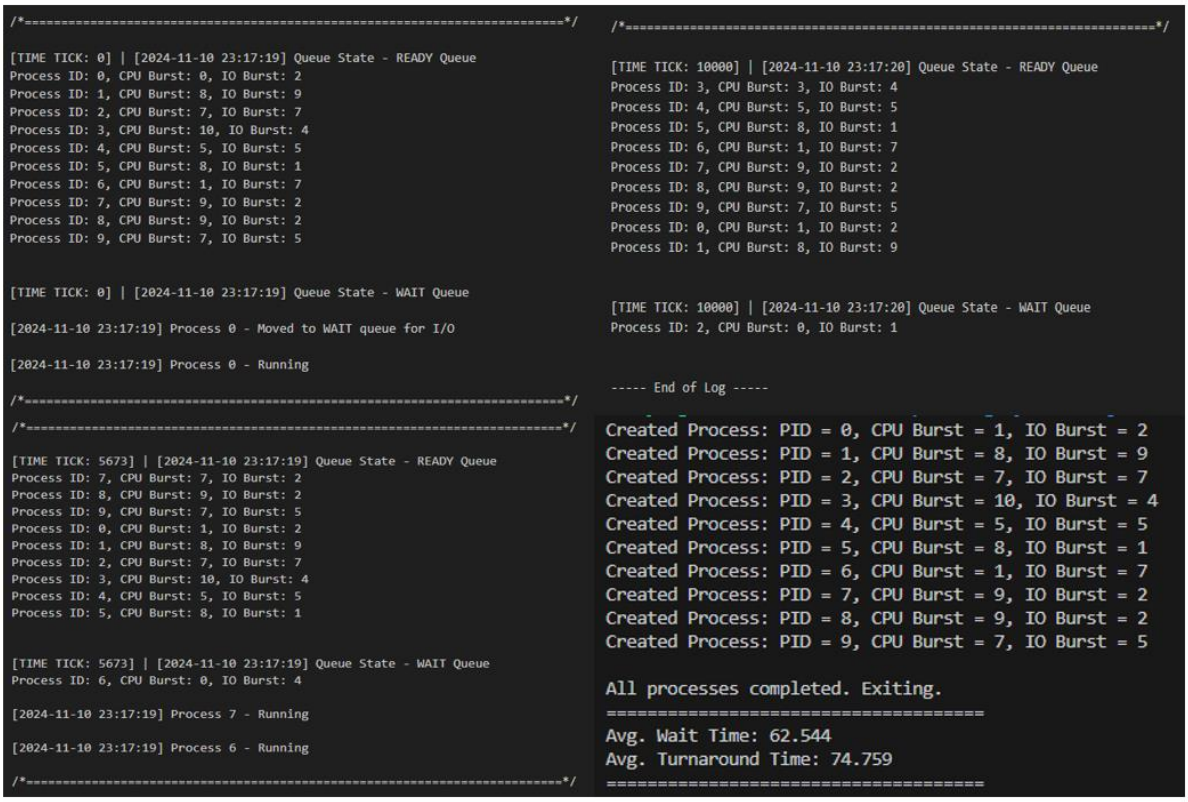


Fig 30. FCFS Result of Ten Thousand Tick Tick Ver.

Fig 30에 FCFS 스케줄링에 대한 결과가 나타나 있다. 먼저 Time Tick이 0일 때 Ready Queue에서 시작하여 Time Tick이 10000이 될 때까지 정상적으로 스케줄링 하는 것을 확인할 수 있다. 이후, 주목해야 할 부분은 Wait Time과 Turnaround Time이다. Wait Time의 경우 2배 이상 증가하였으며, Turnaround Time 또한 증가한 것을 확인할 수 있다.

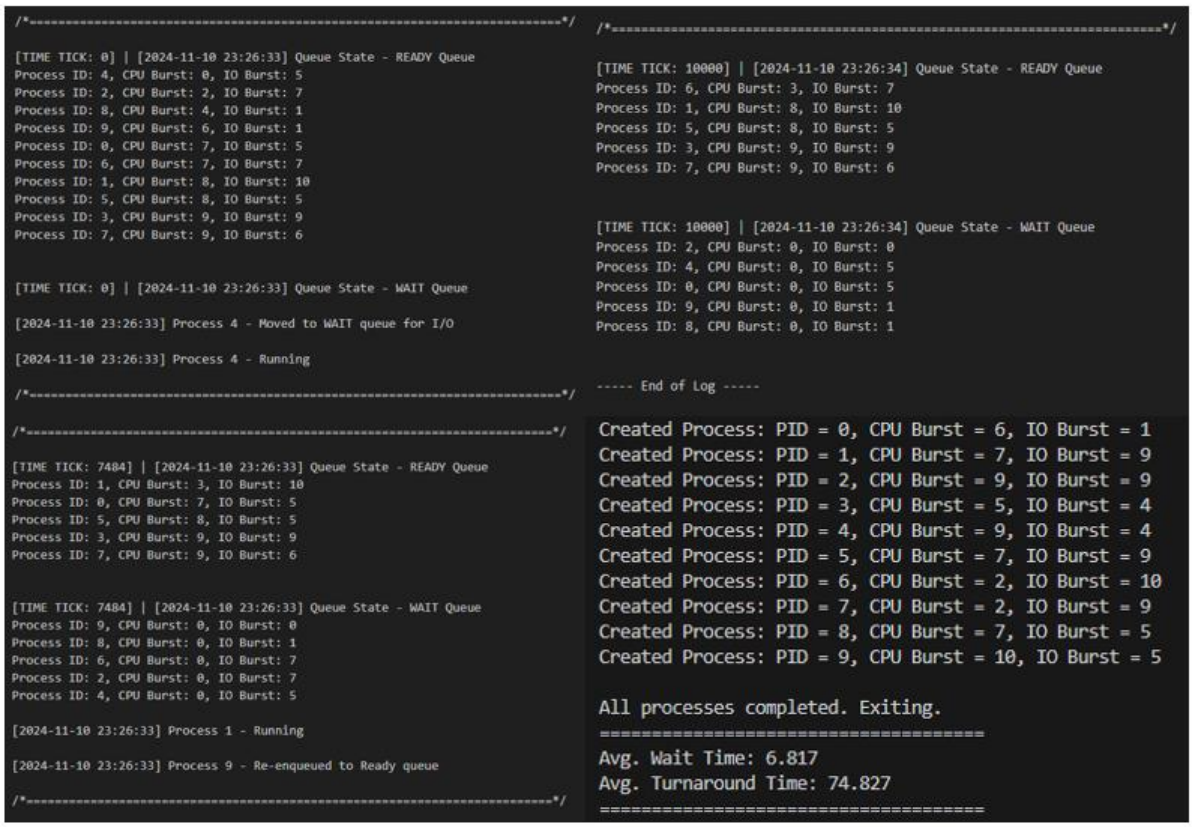


Fig 31. SJF Result of Ten Thousand Tick Ver.

Fig 31에 SJF 스케줄링에 대한 결과가 나타나 있다. Time Tick이 0부터 10000까지 스케줄링 되는 것을 확인할 수 있다. 차이를 보이는 곳은 Wait Time과 Turnaround Time이다. Terminate Version에서는 FCFS와 Wait Time이 큰 차이를 보이지 않았다. 하지만 반복되는 스케줄링에서는 Wait Time이 매우 줄어든 것을 확인할 수 있다.

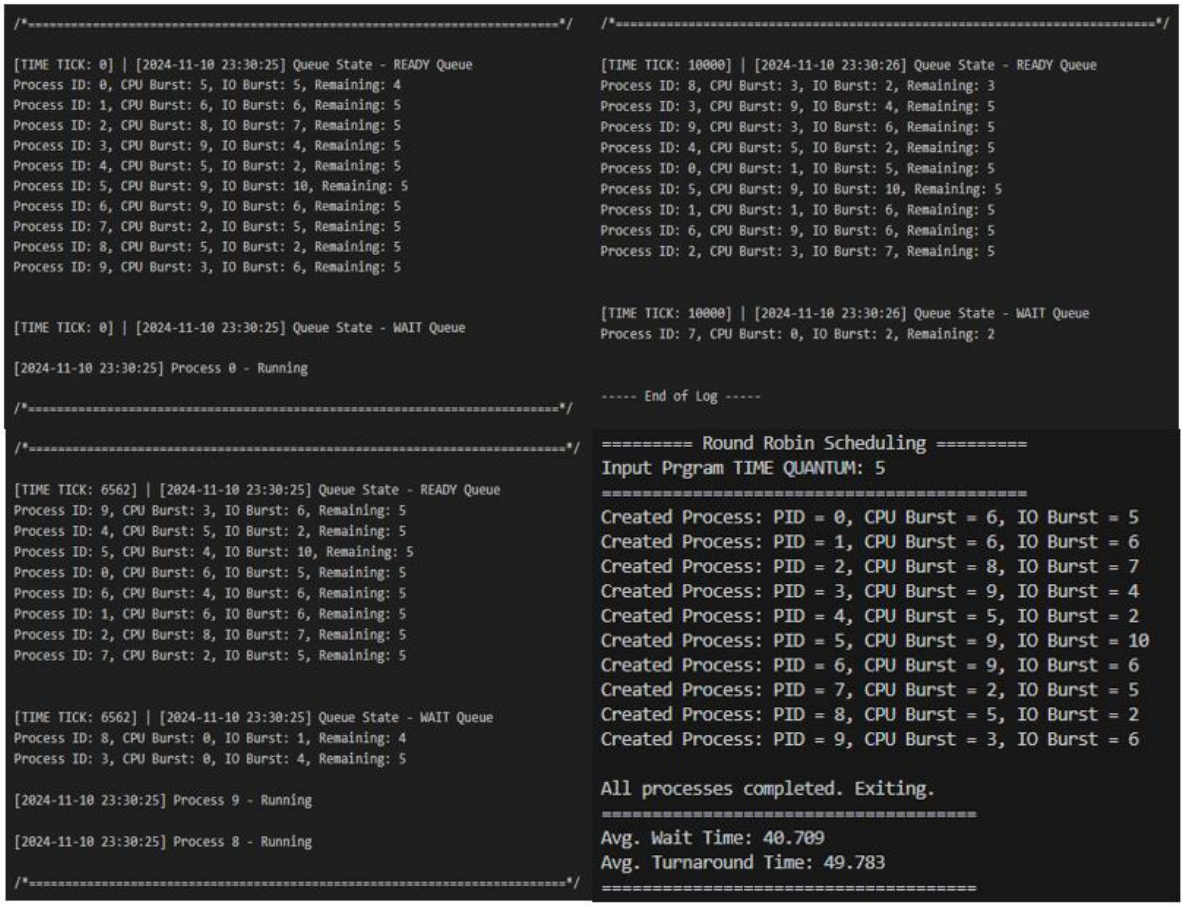


Fig 32. RR Result of Ten Thousand Tick Ver.

Fig 32에 Round Robin 스케줄링에 대한 결과가 나타나 있다. 잔여 시간에 따라 스케줄링이 되는 것을 확인할 수 있고 0부터 10000까지의 Time Tick 동안 스케줄링이 이루어지는 것 또한 확인할 수 있다. Wait Time과 Turnaround Time에서 Wait Time은 증가하였지만 Turnaround Time은 비슷한 시간이 걸렸다는 것을 확인할 수 있다.

6. Evaluation

본 장에서는 5장에서의 실험 결과를 바탕으로 Scheduling Criteria에 대해 평가하도록 한다.

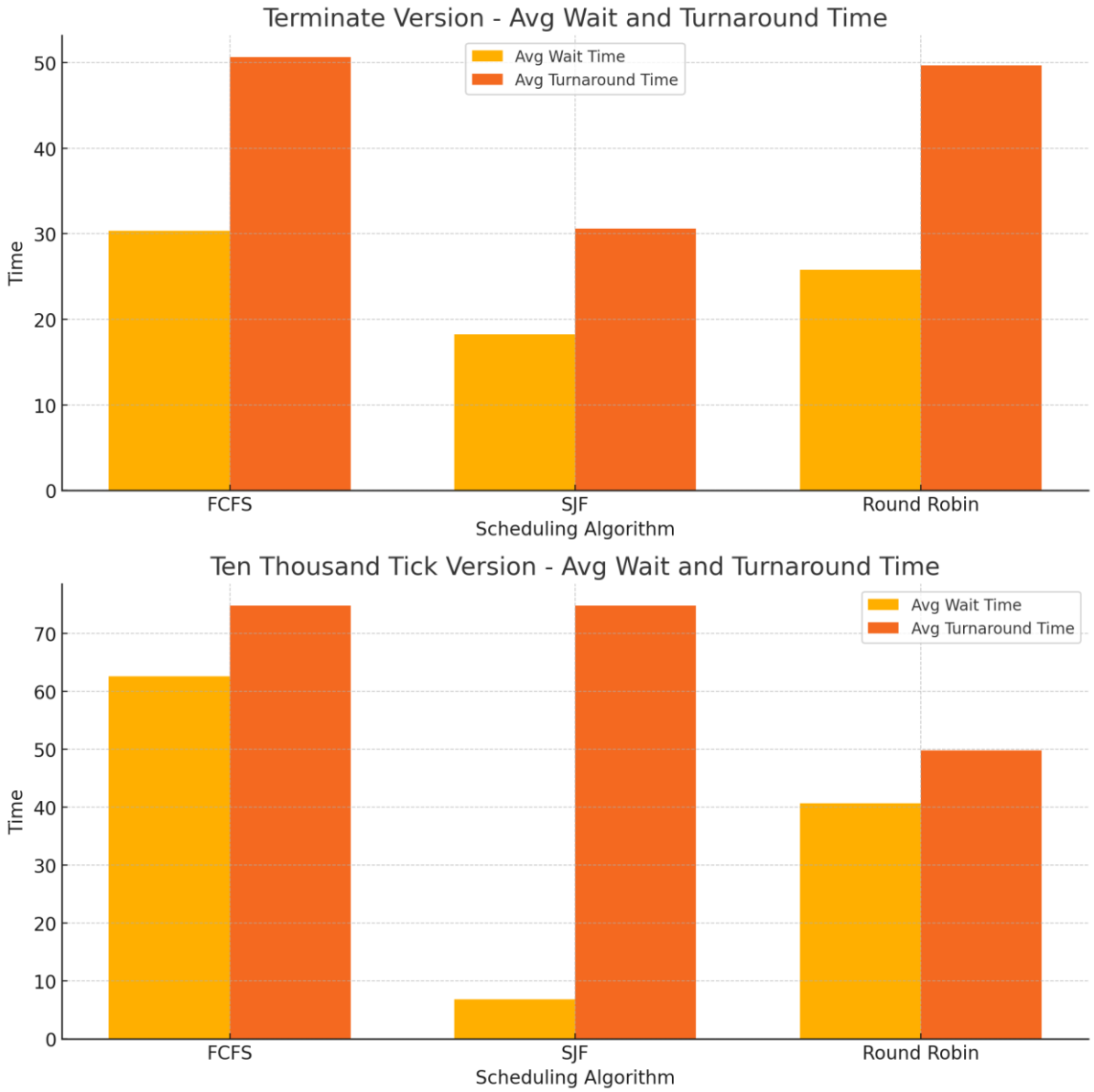


Fig 33. Bar Graph of Both Version's Avg. Wait Time and Avg. Turnaround Time

Fig 33은 5장에서 도출된 결과를 바탕으로 Terminate Version과 Ten Thousand Tick Version에 대한 FCFS, SJF, Round Robin 스케줄링 알고리즘의 성능을 시각적으로 표현한 그래프이다.

우선, Terminate Version에서는 SJF가 가장 짧은 평균 대기 시간을 기록했지만, 기대했던 만큼의 큰 감소는 나타나지 않았다. 이는 해당 버전의 프로그램이 프로세스를 단 한 번만 스케줄링하기 때문이다. 즉, 프로세스는 CPU Burst와 I/O Burst를 한 번 소진한 후 종료되며, Arrival Time은 모두 0으로 고정되어 있어 대기 시간이 크게 감소하지 않았다. FCFS와 Round Robin은 대기 시간과 반환 시

간에서 유사한 결과를 보였는데, 이는 Arrival Time이 0으로 고정되고, Burst가 모두 소진되면 종료되는 구조 때문으로, 성능에 큰 차이를 나타내기 어려운 상황이었다.

반면, Ten Thousand Tick Version에서는 다르게 나타난다. 이 버전에서 프로세스는 Burst가 소진될 때마다 재할당되며 Arrival Time이 변화한다. 이러한 이유로 SJF는 Ready Queue에 들어오는 순서대로 Burst 값을 오름차순으로 정렬하여 평균 대기 시간을 크게 줄일 수 있었다. 반대로, FCFS는 Arrival Time이 고정되어 있는 반면 Time Count는 계속 증가하기 때문에 평균 대기 시간과 반환 시간이 상당히 증가하였다. Round Robin은 Terminate Version과 유사한 성능을 유지했는데, 이는 이 스케줄러가 Time Quantum을 기준으로 프로세스를 관리하기 때문이다. Time Tick이 증가해도 Time Quantum이 일정하게 유지되기 때문에, 대기 시간과 반환 시간에서 균일한 결과를 보인다.

이 **평가**를 통해 얻을 수 있는 결론은 다음과 같다:

1. 단일 스케줄링에서 프로세스가 종료되는 환경에서는 FCFS와 SJF 간의 성능 차이가 미미하다. 이는 프로세스가 한 번의 스케줄링을 통해 종료되는 환경에서 두 스케줄링 방식의 대기 시간과 반환 시간에서 큰 차이가 나기 어렵기 때문이다.

2. 반복 스케줄링 환경에서는 SJF가 FCFS보다 성능이 우수할 수 있다. SJF는 각 프로세스의 Burst 값을 기준으로 정렬하여 대기 시간을 최적화하기 때문에, 프로세스가 반복적으로 스케줄링 될 때 대기 시간 절감에 효과적이다.

3. Round Robin 스케줄러의 성능은 Time Quantum에 크게 좌우된다. 이 실험에서도 확인할 수 있듯이, Time Quantum이 일정하게 유지되는 한 다양한 환경에서도 유사한 성능을 보인다. 따라서, Round Robin의 성능을 최적화하려면 Time Quantum의 값을 설정하는 것이 중요한 요소임을 알 수 있다.

위의 분석을 통해 각 스케줄링 알고리즘이 특정 조건에서 어떻게 동작하는지를 확인할 수 있었으며, 특히 SJF의 성능 향상 조건과 Round Robin의 안정성 조건을 파악할 수 있었다.

7. Conclusion

본 레포트에서는 Simple Scheduling 구현을 목표로 2 장에서 요구 사항을 검토하고, 3 장에서 관련 개념을 이해한 뒤, 4 장에서는 순서도를 통해 구체적인 구현 아이디어를 제시하였다. 이후 구현된 프로그램의 코드 설명을 진행하였다. 5 장에서는 FCFS, SJF, Round Robin 스케줄링 알고리즘을 적용한 두 가지 버전의 결과를 비교하고, 6 장에서 도출된 결론을 기반으로 막대 그래프를 활용해 각 스케줄링 방식의 성능과 특성을 분석하였다.

프로그램의 구현 과정에서 핵심 과제는 부모 프로세스가 자식 프로세스를 효과적으로 관리하고, 프로세스 간 통신을 위한 인터페이스를 구축하는 것이었다. 프로세스 생성에 있어 fork 함수를 단순히 사용하는 대신, 이중 포인터를 활용한 자식 프로세스 관리 방식을 도입하여 부모 프로세스가 자식 프로세스를 더욱 유연하게 관리할 수 있도록 설계하였다. 기존의 메시지 라이브러리 대신 사용자 정의 함수를 사용해 상황에 맞춰 스케줄러를 호출함으로써, 각 스케줄링 알고리즘의 특성에 맞는 동작을 구현할 수 있었다. 이러한 방법으로 CPU Burst 가 모두 소진되었을 때의 로직과 Round Robin 스케줄링에서 Time Quantum 과 Remaining Time 에 따른 분할 처리 로직을 구축하였다.

두 가지 버전의 프로그램, 즉 Terminate Version 과 Ten Thousand Tick Version 을 구현하면서 FCFS, SJF, Round Robin 스케줄링 알고리즘의 성능 차이를 확인할 수 있었다. Terminate Version 에서는 프로세스가 한 번의 CPU Burst 와 I/O Burst 를 소진하고 종료되는 구조로 인해 FCFS 와 SJF 간의 성능 차이가 미미했다. 반면, Ten Thousand Tick Version 에서는 프로세스가 Burst 를 재할당 받으며 Arrival Time 이 변동하는 상황이 발생했으며, 이로 인해 SJF 가 대기 시간을 최소화하여 우수한 성능을 보였다. 이번 프로젝트를 통해 얻은 깨달음은 다음과 같다.

먼저, 스케줄링 알고리즘의 특성에 따른 성능 변화를 실험적으로 확인하였다. 특히, 반복 스케줄링 환경에서 SJF 의 대기 시간 절감 효과와 FCFS 의 상대적 성능 저하를 관찰할 수 있었다. 또한 Round Robin 스케줄링의 Time Quantum 값의 중요성을 확인하였다. Time Quantum 이 유지되는 한, 다양한 상황에서도 안정적인 성능을 제공하는 Round Robin 스케줄링의 특성은 실시간 시스템에서의 응답성을 보장하는 중요한 요소임을 보여준다. 마지막으로, 자식 프로세스 관리 및 통신 인터페이스 설계의 중요성을 깨달았다. 기존 방식에서 벗어난 이중 포인터를 활용한 자식 프로세스 관리와 사용자 정의 함수로 통신을 제어함으로써, 더욱 유연하고 확장 가능한 프로그램을 설계할 수 있었다. 이 레포트의 실험 및 분석 과정은 다양한 스케줄링 알고리즘의 성능과 특징을 비교하고 이해하는 데 기초 자료가 되었으며, 이를 기반으로 향후 더 복잡한 시스템 스케줄링 구현에 필요한 통찰을 제공하였다.

결론적으로, 본 레포트를 통해 스케줄링 알고리즘 구현 및 성능 평가에 대한 이해를 심화할 수 있었으며, 동시성 관리와 효율적인 자원 분배의 중요성을 확인하였다. 이 프로젝트는 더 발전된 스케줄링 설계에 있어 중요한 기초 자료와 성찰을 제공하는 의미 있는 경험이 되었다.