Operating Systems

Proj #1

-Simple Scheduling-

박기태 32161570

Dankook University

Mobile Systems Eng.

2020 Autumn

-Contents-

1. Project Introduction

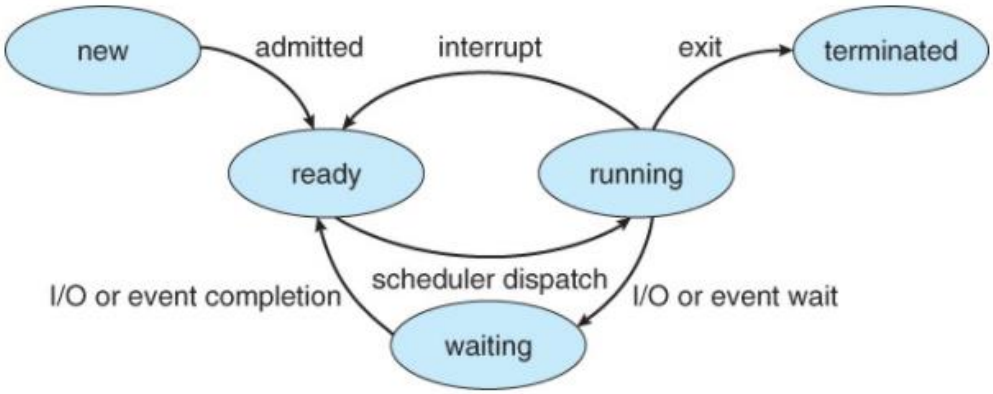
- 운영체제 내에서 프로세스에게 CPU 를 할당해주는 Scheduling 에 대해 소개한다.
- Scheduling 이 어떠한 과정으로 진행이 되는지와 여러 가지 Scheduling 방법들에 대해 소개한다.

2. Project Goals

- 운영체제 내에서 일어나는 Scheduling 과정의 필요성을 이해한다.
- 각 스케줄링 기법의 효율 분석을 하기 위한 척도인 Throughput, Turnaround time, Waiting time, Response time 에 대해 학습한다.
- 선점, 비선점 스케줄링 기법에 대해 학습하고, 각 해당되는 여러가지 Scheduling 기법들에 대한 특성을 이해하고, 각각의 방법들의 장단점을 비교한다.
- 프로세스의 5 가지 상태에 대한 이해를 바탕으로 I/O 작업을 추가하여 실제 스케줄링과 최대한 유사하게 시뮬레이터를 구현하도록 한다.
- 프로세스의 상태 변경 시 linked list 자료구조를 이용하여 프로세스의 삽입, 삭제를 관리하도록 한다.
- SIGALRM, sigaction() 함수를 이용하여 다른 프로세스에게 신호를 보내 시간 경과를 알릴 수 있도록 한다.
- 프로세스 간 데이터를 주고받는 기법인 IPC 에 대한 이해를 바탕으로 일부 기법을 적용해보도록 한다.
- 팀 프로젝트로 진행하여 git 사용에 익숙해지고, 생산적이고 효율적인 분담 작업을 위한 커뮤니케이션 능력 함양을 목표로 한다.

3. Concepts used in Multi-Threads Programming

- **Scheduling**



<그림 1>

- Scheduling 이란 여러 개의 프로세스들이 하나의 CPU 내에서 실행되어야 할 때 운영체제가 CPU 에게 프로세스를 할당하는 과정을 의미한다.
- 프로세스들이 하나의 CPU 에서 처리되는 방법은 위의 <그림 1>과 같은 과정으로 일어난다.
  i. <그림 1>에서는 새롭게 생성되어진 프로세스들은 Ready Queue 로 들어가 대기를 하게 된다.
  ii. Scheduling 을 통해 Ready Queue 내에서 대기하던 프로세스를 CPU 에 할당하여 실행시킨다.
  iii. CPU 에서 실행이 완료된 프로세스는 Waiting Queue 로 이동하여 I/O 작업을 수행한다.
  iv. I/O 작업을 수행한 이후 Ready Queue 로 이동하여 다시 대기한다.
- 위의 과정에서 필요한 과정이 Scheduling 인데, 이는 Ready Queue 에서 대기중인 프로세스를 CPU 에 할당하는 것을 의미한다.
- 하나의 CPU 에서는 한 번에 하나의 프로세스만을 실행할 수 있기에, 여러 개의 프로세스들이 실행되기 위해서는 Scheduling 이라는 과정이 필요하다.

- **Scheduling Metrics**
  - **Waiting Time**
    → CPU 에 할당되기 까지 Ready Queue 에서 대기한 시간을 의미한다
  - **Response Time**
    → 프로세스가 만들어진 후 부터 CPU 에 할당 받을 때까지 대기하는 시간을 의미한다.
  - **Turn-Around Time**
    → Ready-Queue 에 도착해서 부터 프로세스가 종료될 때 까지의 시간을 의미한다
  - **Throughput**
    → 단위 시간동안 완료된 프로세스의 수를 의미한다

- **Scheduling Policy Goals**
  1. Minimize Response time
  2. Maximize Throughput
  3. Fairness

- **Basic criteria of Scheduling**
  1. **Execution Time**
     - 프로세스가 완료될 때까지 CPU 내에서 실행되는 시간을 의미한다.
  2. **I/O Time**
     - 프로세스가 I/O 작업을 처리하는데 필요한 시간을 의미한다.
  3. **CPU bound**

- Execution Time 과 I/O Time 중에서 Execution Time 이 더 큰 프로세스를 의미한다.

4. I/O bound

- Execution Time 과 I/O Time 중에서 I/O Time 이 더 큰 프로세스를 의미한다

5. Time Quantum(Slice)

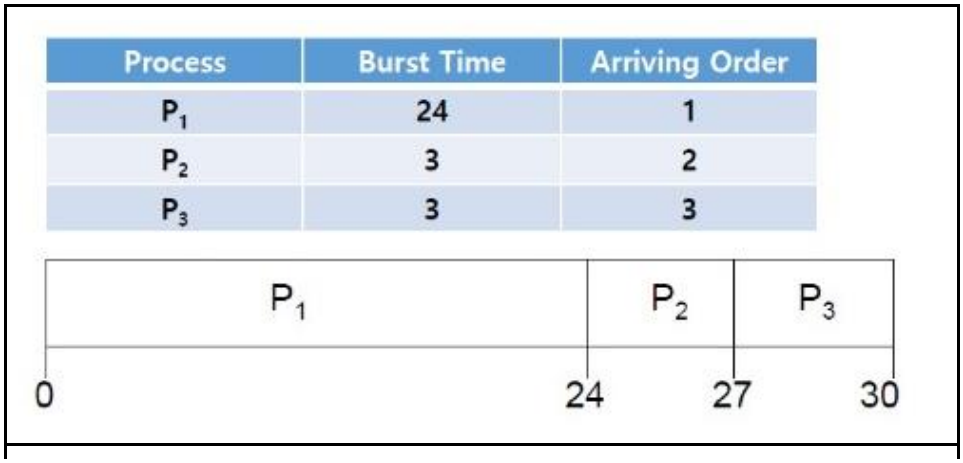
- CPU 에서 한 프로세스에게 할당하는 시간을 의미한다.
- Round-Robin Scheduling 에서 사용한다.

- **Non-Preemptive Scheduling(비선점 스케줄링)**

- 비선점 방식에서는 프로세스가 한 번 CPU 에 할당되면, 할당을 받은 프로세스가 종료될 때 까지 운영체제가 CPU 를 사용할 수 없는 방식을 의미한다.

1. **FCFS(First In First Out)**

- 프로세스가 먼저 들어온 순서대로 CPU 에 할당해주는 방식이다
- 매우 단순하고 기본적인 방법이지만, Execution Time 이 큰 프로세스가 짧은 프로세스 앞에 들어오게 되면 Average Waiting Time 이 길어지는 단점이 있다.

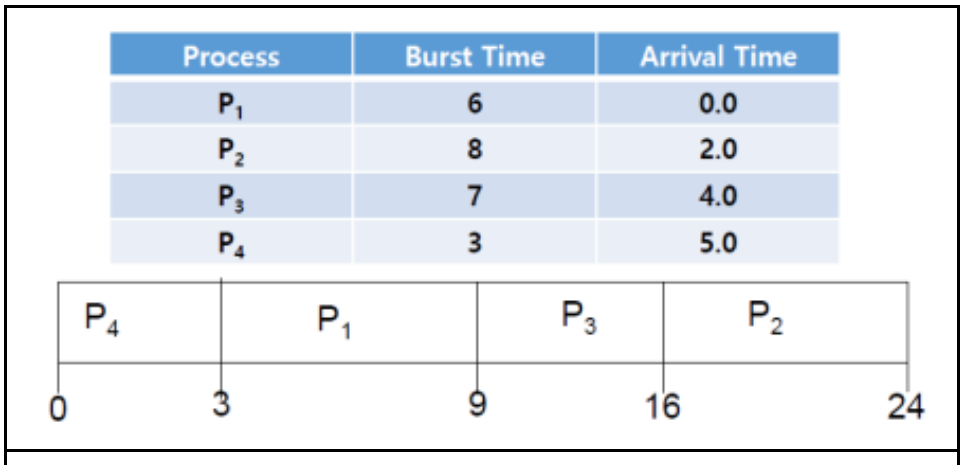


<그림 2>

- 위의 <그림 2>에서 프로세스들이 P1, P2, P3 의 순서로 동시에 들어왔고, 들어온 순서대로 CPU 에 할당되어 프로세스를 수행하게 된다.
- Waiting Time → P1: 0, P2: 24, P3: 27
- Average Waiting Time → (0 + 24 + 27) / 3 = 17
- Execution(Burst) Time 이 가장 큰 P1 뒤에 P2, P3 가 들어왔기 때문에 Average Waiting Time 이 커져 Fairness 의 문제가 발생된다.
- 순서에 따라 CPU 자원 사용에 있어 효율성이 매우 떨어질 수 있다.

2. SJF(Shortest Job First)

- FCFS 방식의 비효율성을 해결하기 위해, Execution(Burst) Time 이 가장 작은 프로세스를 먼저 수행하는 방식이다.
- FCFS 에서 가장 효율성이 좋을 때의 프로세서 순서대로 CPU 할당을 하게 된다.



<그림 3>

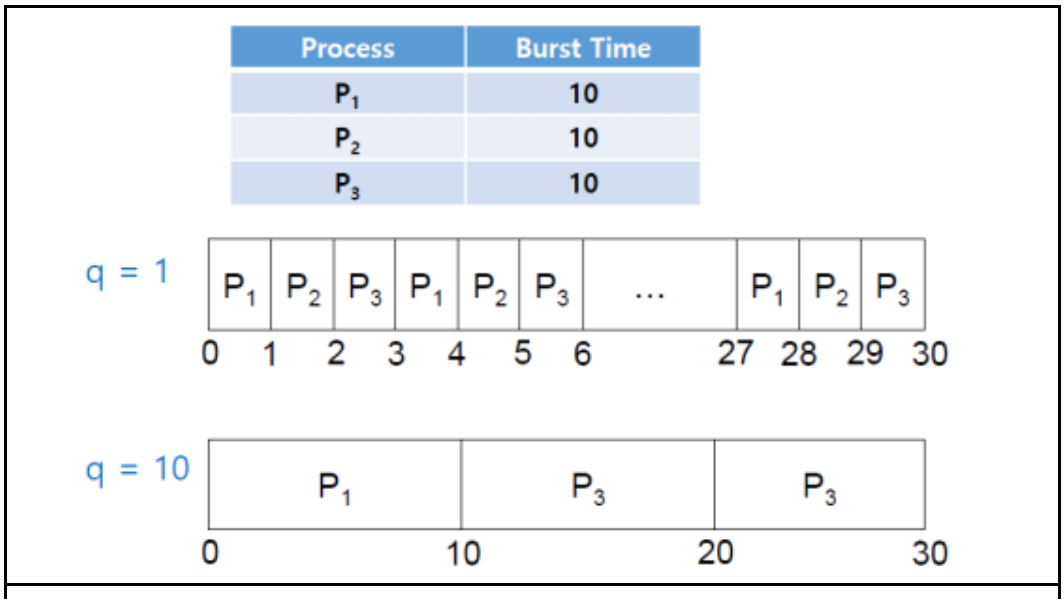
- 위의 <그림 3>에서와 같이, CPU Execution(Burst) Time 이 작은 순서대로 프로세스를 CPU 에 할당하여 연산을 수행하게 된다.
- FCFS 방식에서 Execution(Burst) Time 이 상대적으로 큰 프로세스가 앞에 할당될 경우 Average Waiting Time 이 커져 발생하는 Fairness 와 관련된 문제를 해결할 수 있다.
- SJF 방식을 실제로 구현하기 위해서는 각 프로세스들의 CPU Execution Time 을 모두 예측하여 알고 있어야 하기 때문에, 현실적으로 구현하기 불가능한 이상적인 방식이라고 여겨진다.

- **Preemptive Scheduling(선점 스케줄링)**

- 선점 방식에서는 프로세스가 한 번 CPU 에 할당되면, 할당을 받은 프로세스가 종료되지 않더라도 운영체제가 CPU 자원을 강제로 회수해올 수 있다.
- 대표적인 선점방식으로는 Round-Robin Scheduling 기법이 있는데, 여기서 Time-Quantum(Slice)과 같은 HW 적인 Interrupt 가 필요시 되기에 HW 관점에서의 지원이 필요하다.
- 또한 자원을 회수하는 과정에서 여러 개의 프로세스들 간의 데이터 충돌이 발생할 수 있는 문제점이 존재한다.

1. **RR(Round-Robin)**

- Preemptive Scheduling 의 대표적인 기법으로 여러 프로세스들이 CPU 를 점유할 때 Time-Quantum 을 할당을 받아 일정한 시간을 기준으로 프로세스들이 번갈아 돌아가며 수행되는 방식이다.
- 기존 Non-Preemptive 방식에서의 두 가지 기법의 문제점을 해결하기 위해 고안되어진 방법이다.

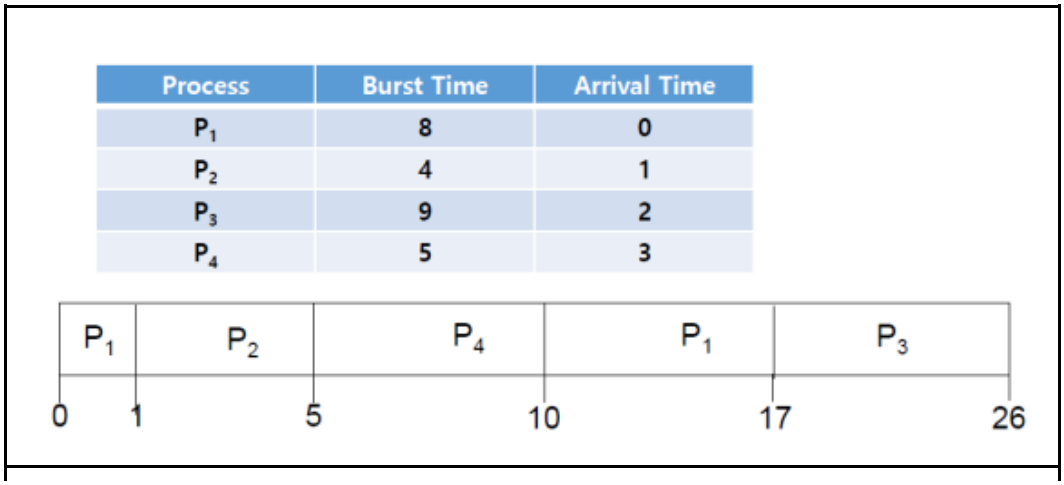


<그림 4>

- 위의 <그림 4>와 같은 방식으로 수행되며, 각 프로세스들에게 Time-Quantum 을 할당하는 방식으로 수행된다.
- Time-Quantum(q)이 1 일 경우 Average Waiting Time 이 29 이고, ime-Quantum(q)이 10 일 경우 Average Waiting Time 이 20 이다.
- 위의 결과에서 볼 수 있듯이 Time-Quantum 이 매우 커지면 FCFS 방식과 차이가 없고, Time-Quantum 이 매우 작아지면 Waiting Time 이 짧아지는 장점이 있지만 프로세스간 Context Switching 시간이 매우 커져 최종 실행 시간은 더 커질 수 있다는 단점이 존재한다.
- 적절한 Time-Quantum 을 할당하여 연산을 수행하는 과정이 필요하며, USER 와 실시간으로 상호작용이 필요할 때에는 Time-Quantum 을 짧게 두는 것이 필요하다.

2. SRT(Shortest Remaining Time)

- SJF 기법을 Preemptive 방식으로 구현한 것이 SRT 방식이다.

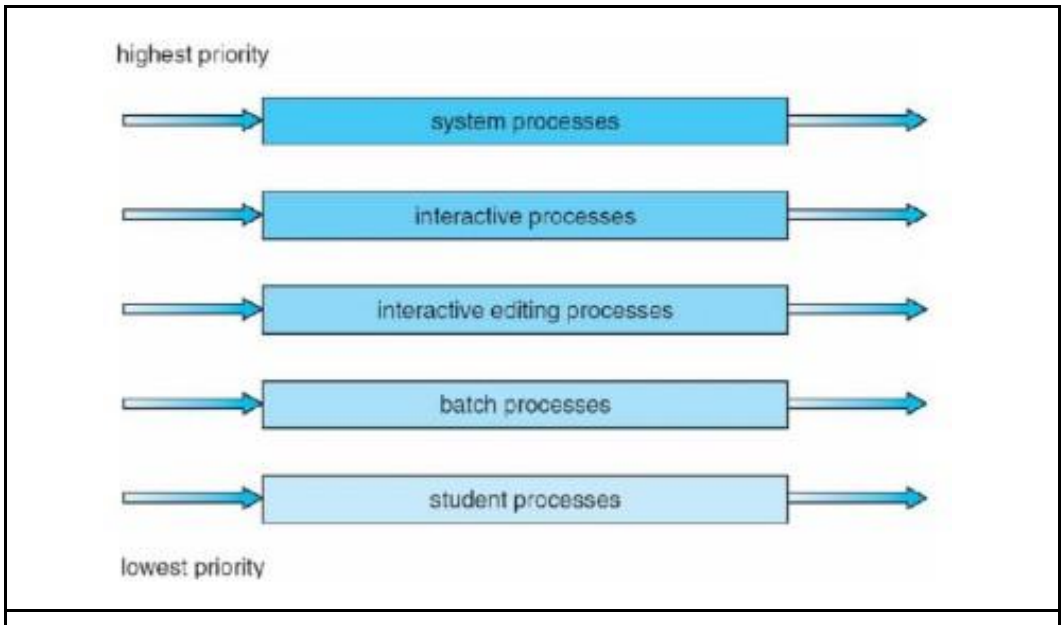


<그림 5>

- 먼저 첫 번째로 들어온 프로세스인 P1 에 CPU 를 할당하고 실행시킨다.
- P1 이 1 초간 실행된 이후, P2 가 들어오게 되는데 이 때 P2 의 Execution Time 이 더 작으므로, P1 프로세스는 CPU 자원을 반납하고 P2 에 할당하게 된다.
- 이런 방식으로 새로운 프로세스가 들어오게 되면, 각 프로세스들 간의 Execution Time 을 비교하여 작은 순서대로 CPU 에 할당하게 된다.
- SRT 방식의 경우, Execution Time 이 큰 프로세스들은 계속 뒤로 밀리게 되어 Fairness 의 문제점이 발생하게 된다.
- 뒤로 밀린 프로세스들은 Starvation 상태가 발생하고, 이러한 문제를 해결하기 위해 Aging 을 이용하여 우선순위를 앞으로 넣어주어 Fairness 의 문제점을 해결할 수 있다.

3. MLQ(Multi-Level Queue)

- MLQ 방식에서는 기존에 설명했던 Scheduling 기법들과는 다르게, 하나만의 Ready-Queue 만을 사용하는 것이 아닌, 다중 Ready-Queue 를 사용하게 된다.
- 서로 다른 시스템들은 서로 다른 스케줄링 목표를 가지고 있다.
- 예를 들어, Multi-User Process, Batch Systems, Interactive Systems, Real-time System 들은 서로 다른 Scheduling 을 목표로 한다.

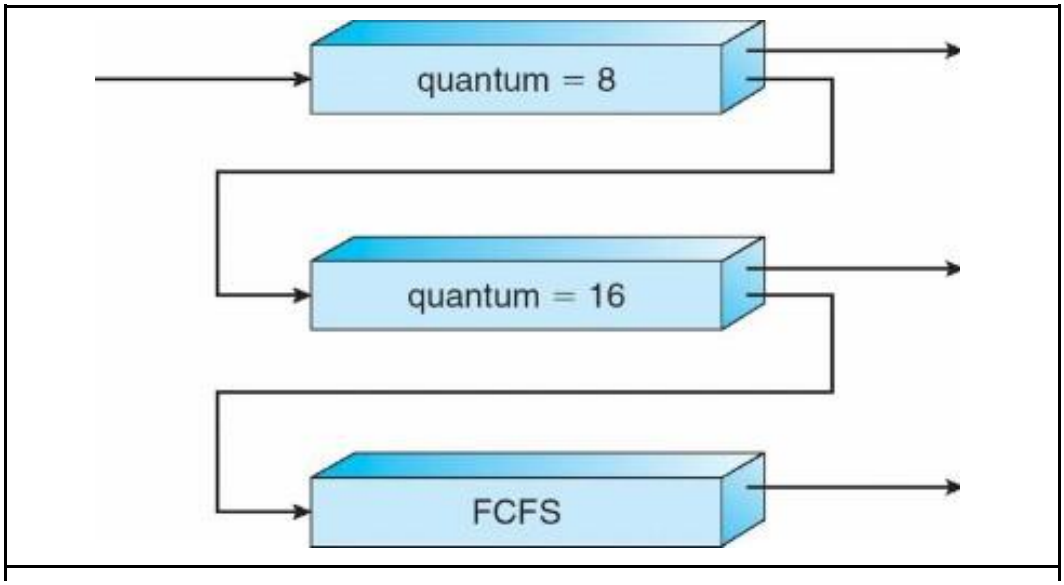


<그림 6>

- <그림 6>과 같이 서로 다른 프로세스들을 각각의 특징에 맞게 여러 개의 Ready-Queue 에 나누어 할당하게 된다.
- MLQ 방식에서는 각각의 Queue 들이 서로 다른 Scheduling 기법을 가지고 수행되고, 각각의 우선순위가 다르게 된다.
- 이로 인하여 특정 Queue 만 수행되고 또 다른 Queue 는 실행되지 않아 Starvation 현상이 발생할 수 있는데, 각 Queue 마다 Time-Quantum 을 다르게 적용하여 독점하는 상황을 방지하고 Aging 을 통해 Fairness 의 문제를 해결해 준다.

4. MLFQ(Multi-level Feedback Queue)

- MLFQ 방식에서는 3 번에서 설명한 MLQ 와 다르게 프로세스들이 서로 다른 Ready-Queue 로 이동할 수 있다.
- MLQ 방식에서 발생할 수 있는 Starvation 을 Feedback 을 통해 예방할 수 있도록 한다.



<그림 7>

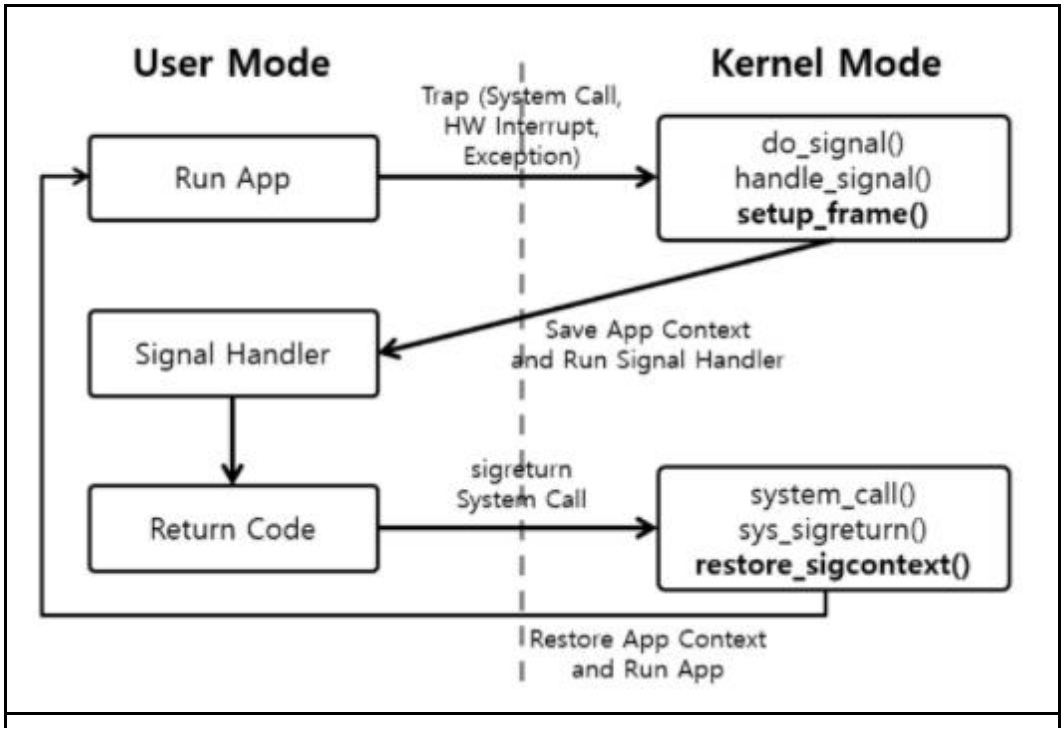
- <그림 7>에서와 같이 프로세스가 생성되면 첫 번째 Queue 에 할당되어진 후 실행된다.
- 이후, 두 번째 Queue 로 이동하여 프로세스가 실행되고 이러한 과정을 거쳐 이후에 FeedBack 을 통해 각 프로세스의 특징에 맞는 Queue 로 할당되어 실행되어진다.
- 이러한 방법을 통하여, MLQ 에서 발생했던 문제들을 효과적으로 해결할 수 있다.

● **IPC(Inter-Process Communication)**

- Parent 프로세스(Kernel)와 연결된 Child 프로세스(User)간 Context Switching 과정이 필요한데, 이 때 필요한 것이 IPC 이다.
- 이번 프로젝트에서 사용한 IPC 기법으로는 SIGNAL, MSG QUEUE 가 있으며 이에 대해 설명한다.

1. SIGNAL

- 시그널이란 프로세스에게 전달되어지는 Interrupt 신호를 의미한다.
- Interrupt 신호를 수신한 프로세스는 신호의 종류에 따라 Handler 를 수행하거나 무시하게 된다.
- 관련된 API 들에는 kill, signal, alarm 등이 존재한다.



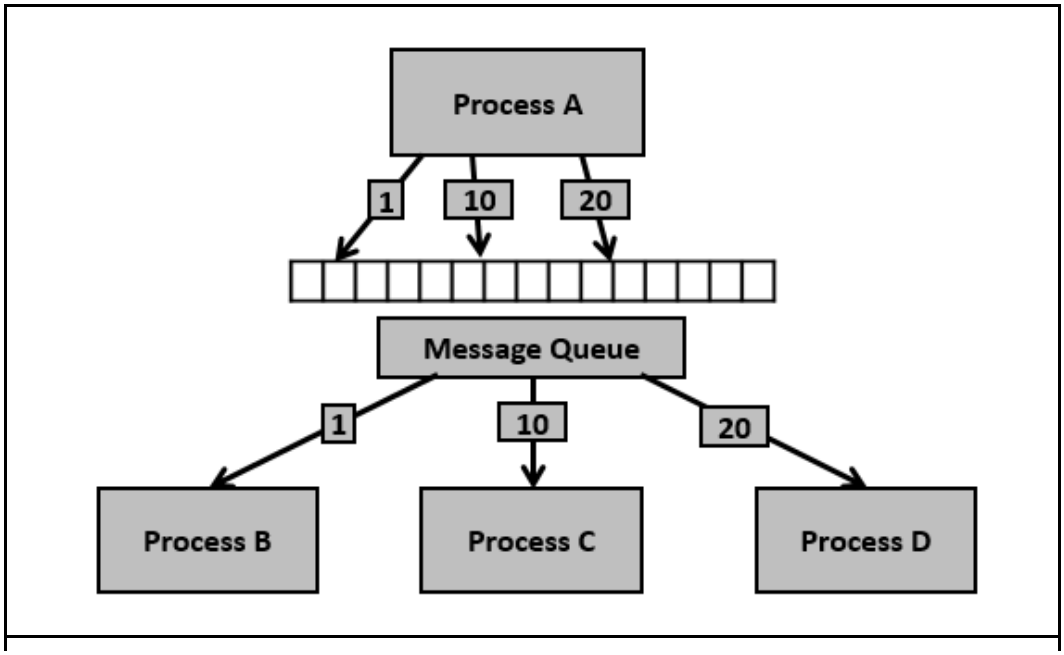
<그림 8>

- 위의 <그림 8>에서는 Signal Handler의 실행 과정을 총체적으로 나타내고 있다.

2. Message Queue

- Message Queue 는 Kernel 에 저장되는 Linked List 이며 전역적으로 관리되고 사용된다.
- 각각의 Message Queue 는 고유한 식별자를 가지고 이 식별자를 통해 어떠한 Process 든 Message Queue 에 접근할 수 있다.

- msgget ( ) : Message Queue 를 생성하는 함수이다.
- msgsnd ( ) : 데이터를 Message Queue 로 전송하는 함수이다.
- msgrcv ( ) : Message Queue 로부터 데이터를 수신하는 함수이다.

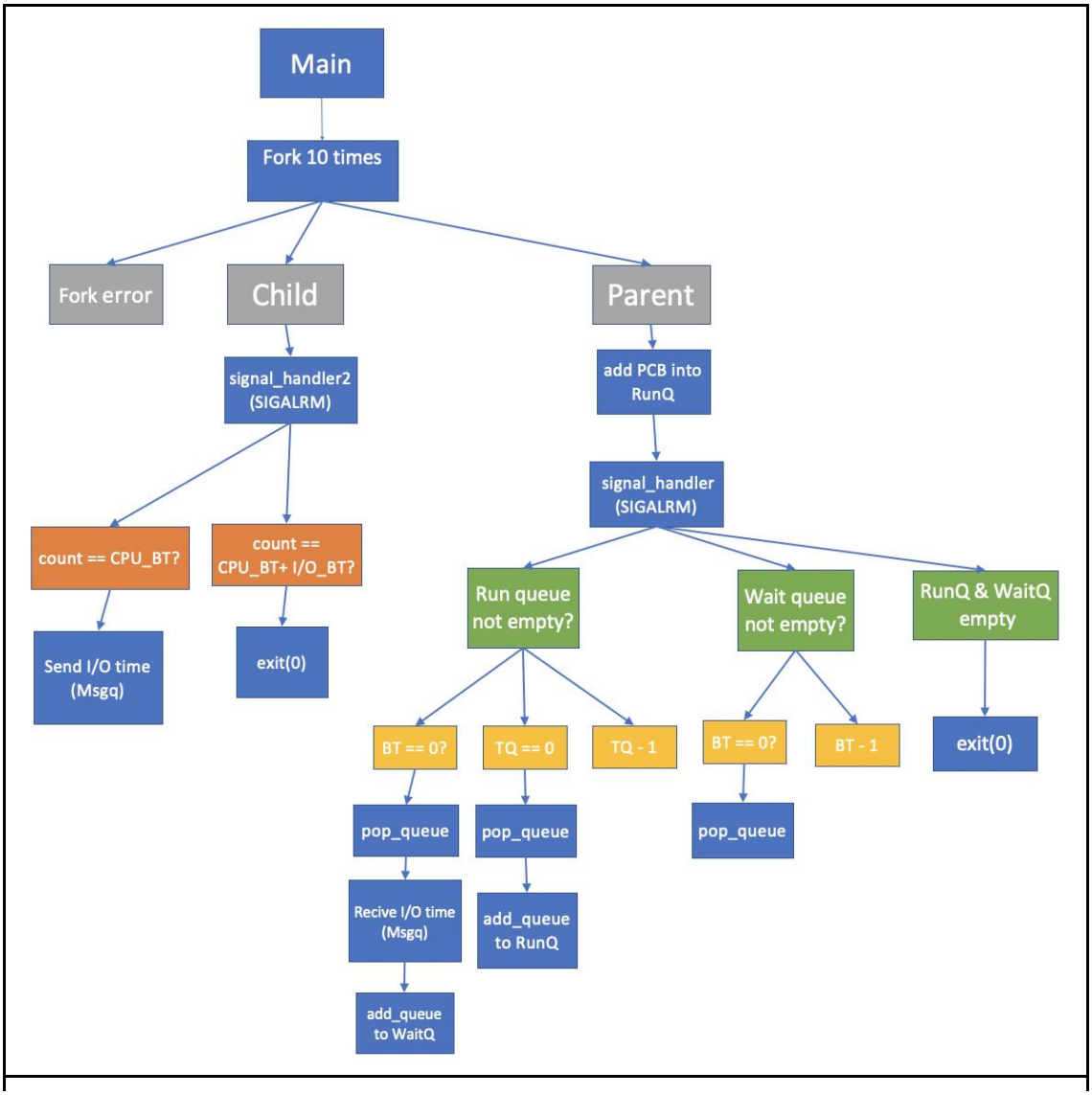


<그림 9>

- Message Queue 에서는 위의 <그림 9> 같이 Global 영역에 저장되어진 공간에 데이터를 전송하여 저장하고, 이를 다른 프로세스들이 수신하여 사용할 수 있다.
- 이번 프로젝트에서 I/O Time 을 전송할 때 Message Queue 를 활용하여 구현하였다.

4. Program Structure

- Proj #1(Simple Scheduling)에서는 운영체제가 프로세스를 Scheduling 하는 Simulator 를 제작하였다.
- 이번 프로젝트에서 제작한 Simulator 에서 **Round-Robin(CPU Execute)**, **FCFS(I/O Execute)**를 구현하였다.
- 프로그램의 전체적인 구조를 한 눈에 살펴보기 쉽도록 아래의 <그림 10>으로 정리하였다.



<그림 10>

- <그림 10>에 대한 내용은 아래의 순서대로 간략히 설명하였다.
    1) 우선 for 문에서 fork() 함수를 이용하여 자식 프로세스를 10 개 생성하도록 하였다.

    2) 자식 프로세스의 경우, signal handler 함수로서 **signal_handler2** 를 등록하고, sigaction() 함수를 통해서 SIGALRM 이 들어올 경우 실행시키도록 하는 구조이다.

    3) 부모 프로세스는 자식 프로세스 10 개를 run queue 에 add 하는 작업을 한다. 이후, 자식 프로세스와 동일하게 **signal_handler** 함수를 등록하고, SIGALRM 이 들어올 경우 실행이 되도록 한다.

● signal_handler

- 매 Tick 마다 count 라는 변수의 값을 1 씩 증가하도록 하였다. 이때 count 의 값이 전체 CPU burst time 보다 크고, run queue, wait queue 가 모두 비어있을 경우 더 이상 진행할 작업이 없기 때문에 exit() 함수를 통해 스케줄링을 종료하도록 하였다.

1) Run Queue part

- Tick 이 발생할 때 마다 burst 를 1 씩 감소시키고 만약 burst time 이 0 일 경우 현재 run queue 의 맨 앞에 있는 프로세스를 pop 하도록 한다. 이후 해당 프로세스가 얼마나 I/O 작업을 할 지 알기 위해 message queue 로부터 값을 가져오도록 하였다. 그리고 난 뒤, wait queue 로 이동시키도록 한다.
- 또한, burst time 을 모두 사용하지 않고 time quantum 만 모두 사용하였을 경우, pop 하고 run queue 에 다시 add 함으로써 round robin 구조를 구현하였다.

- 위의 경우가 모두 부합하지 않는 경우 time quantum 을 1 씩 감소하도록 하였다.

2) Wait Queue part

- 제일 앞에 있는 PCB 의 burst 가 0 이 될 경우, wait queue 에서 pop 하도록 한다. 만약 0 이 아닐 경우 I/O burst 를 1 씩 감소하도록 한다. 이때 I/O 작업은 CPU 작업과 병렬적으로 동시에 이루어지도록 하였다.

● signal_handler2

- 위 함수와 동일하게 Tick 이 발생할 때 마다 count 라는 변수의 값을 1 씩 증가하도록 하였다. 만약 count 의 값이 CPU burst time 과 동일할 경우, CPU 작업이 완료됨을 의미한다. 다음 I/O 작업의 실행시간을 커널에게 알려주어야 한다. random 으로 값을 생성한 뒤 message queue 에 담아 전송하도록 하였다.

- 또한, count 의 값이 CPU burst time + I/O burst time 일 경우, 프로세스의 모든 작업이 마무리 된 것을 나타낸다. exit() 함수를 통해서 프로세스를 종료하도록 하였다.

5. Problems and solutions

- Message Queue 의 관리
  - 이번 프로젝트에서 IPC 를 수행하기 위해 Message Queue 를 사용하는데 있어, 몇 가지의 문제점들을 발생했다.
  - msgget( )함수와 msgsnd( )함수를 활용하여 메시지를 생생하고 커널에 전송하는 과정에서 처음 보는 생소한 문법들을 이해하고 사용하는데 있어 어려움을 겪었다.
  - 메시지들을 Queue 에 저장하고 msgrcv( )함수를 활용하여 메시지를 수신하여 데이터를 읽어오는 과정에서 메시지 찌꺼기들이 남는 상황이 발생했다.
  - 이러한 문제점들을 해결하기 위해 강의, 교안, 예제 코드들을 여러 번 복습하고 여러 방법으로 다시 시도하여, 메시지를 생성하여 송수신 하는 과정에서 메시지 찌꺼기들이 남지 않도록 최적화하여 프로젝트를 완성시켰다.

- Timer 를 통한 Time Quantum 관리
  - 이번 프로젝트에서 Round-Robin 을 구현하기 위해, 어떠한 방법이 효율적인지에 대해 다방면으로 고민하였다.
  - 그 중 하나의 방법으로는 Time Quantum 을 Timer 를 통해 구현하는 것이었다. 예를 들면, Time Quantum 이 4(sec)일 경우 새로운 Signal Handler 를 만들어 하나의 프로세스를 4(sec)마다 관리하는 것이었다.
  - 하지만 이렇게 Round-Robin Scheduling 을 구현하게 된다면, 프로세스가 Time Quantum 을 다 소모하지 않고 종료되었을 경우 Starvation 의 문제가 발생한다는 것을 알게 되었다.
  - 이를 해결하기 위해 TIme Quantum 을 매 Tick 마다 관리하여, 프로세스가 새로 시작하는 경우의 Condition 을 판단하였고, 이 때 Kernel(parent)가 Time Quantum 을 새로 부여하는 방식으로 구현하였다.

- Starvation 의 발생
  - 이번 프로젝트를 수행하기 위해 계획을 세우는 과정에서, 팀원 서로는 먼저 Round-Robin 기법을 완벽히 구현하고, I/O 연산을 추가하여 프로그램을 완성시키기로 하였다.
  - 먼저 I/O 과정이 포함 되어있지 않은 경우에는, process 가 burst time 을 모두 소모하였을 경우 프로세스가 종료되도록 하여, 해당 프로세스가 아직 사용하지 않은 Time Quantum 이 남아있더라도, 바로 다음 프로세스가 Run Queue 에서 실행될 수 있도록 하였다.
  - 하지만 이러한 구조에서 I/O 연산을 포함시키게 되면, Starvation 이 발생하는 경우가 생겼다.
  - 예를 들자면, Time Quantum 이 4 초일 때 프로세스가 해당 Slice 가 실행되고 2 초만에 종료되었다면, 해당 프로세스는 종료되지 않고 Wait Queue 로 이동하게 된다 이 과정에서 남은 2 초동안 CPU 자원에 프로세스가 할당되지 않아 Starvation 이 발생하게 된다.
  - 이러한 문제점을 해결하기 위해 매 Tick 별로 Queue 의 상태를 체크하고, Condition 별로 상황을 나누어 적절한 연산을 수행할 수 있도록 프로그램을 제작하였다.

- sprintf( ), open( ), write( )
  - 이번 프로젝트에서는, 매 Tick 별로 화면에 출력하게 된다면 출력화면이 매우 길어져 비효율적이기에, ***result.txt*** 파일로 출력하도록 하였다.
  - 이를 수행하기 위해 먼저 char array buffer 를 제작하였고, sprintf( )를 통해 Tick 마다 나오는 결과물을 buffer 에 채워 넣고, 이를 write( ) 함수를 이용하여 ***result.txt*** 파일에 저장하도록 하였다.

- 하지만 결과가 파일에 정상적으로 출력되지 않는 현상이 발생하였는데, 이는 buffer size 의 지정에 문제가 있었기 때문이었다. 이를 해결하기 위해 size 를 0x300 으로 지정하였고 Tick 이 끝나면 해당 버퍼를 초기화 시켜 다음 Tick 을 수행할 때 문제가 발생하지 않도록 하였다.

6. Build environment

In this Project #1, Simple Scheduling, we used Termius & Putty

We uploaded our code file on github,

Compilation: Linux Toast Server(133.186.159.169),
with GCC to compile,

To Compile, please type

→ **gcc proj1.c -o proj1**

To run, please type

1. Execute

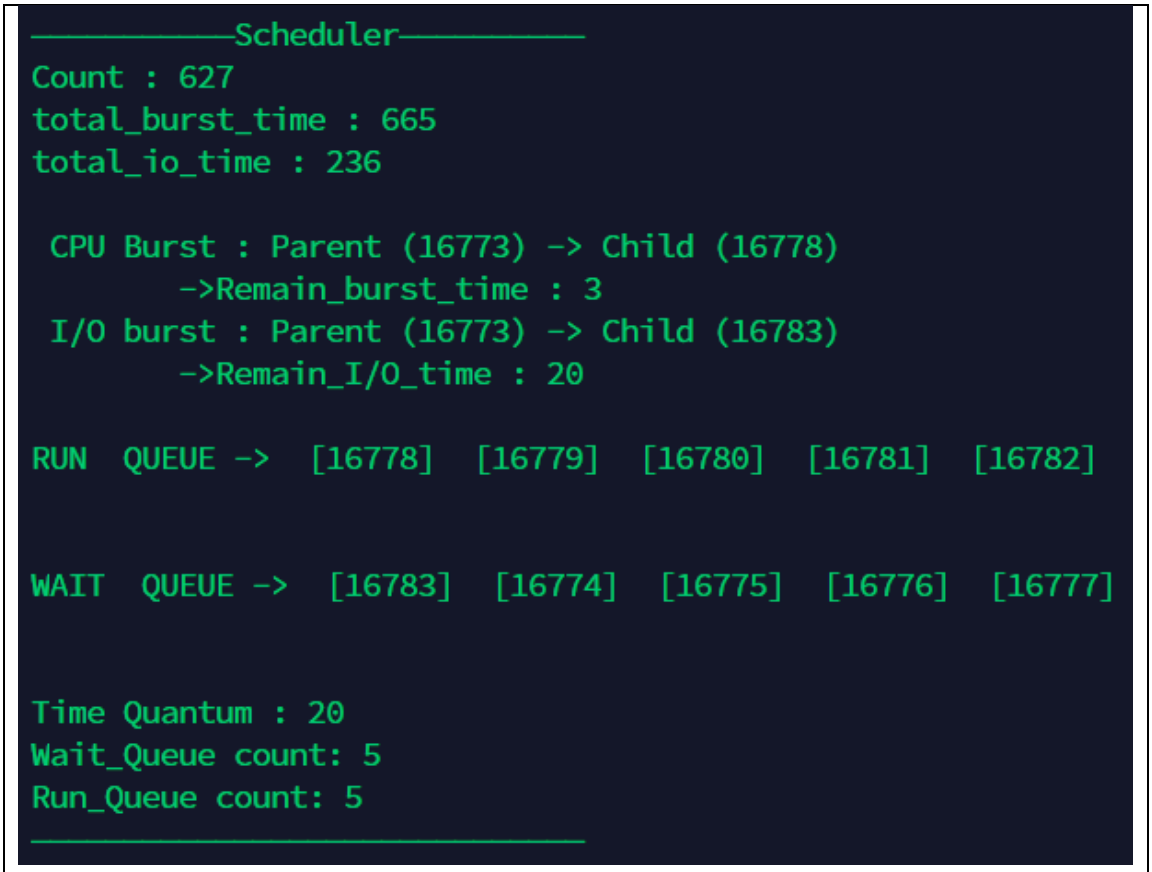
→ **./proj1**

2. View Result

→ **cat result.txt** or **vi result.txt**

7. Screen Capture (Result)

- **Example of Result**



Result Screen of 627 tick

<그림 11>

→ 이번 프로젝트에서 제작한 프로그램은, 프로세스의 스케줄링을 수행하고 **result.txt** 파일을 생성하고 해당 파일에는 위와 같은 구조의 출력물이 계속해서 반복되어 작성 되어있는 것을 볼 수 있다.

→ 위의 <그림 11>에서는 627 Tick 의 순간에 출력된 결과물이다,

→ 출력물에는 Count(현재의 Tick)수, CPU time, I/O time, 현재 CPU 연산이 수행되어지고 있는 프로세스, I/O 를 수행중인 프로세스, 그리고 Wait Queue 와 Run Queue 에 들어가 있는 프로세스 등의 정보를 포함하며 한 눈에 보기 편한 구조로 이루어져 있다.

- **RR Time Quantum 비교 → 20 sec VS 40 sec**

real 18m22.011s user 5m11.965s sys 0m0.179s	real 16m36.008s user 5m22.688s sys 0m0.132s
RR TQ = 20 sec	**RR TQ = 40 sec**

<그림 12>

→ 위의 <그림 12>의 경우는 Time Quantum 이 20 과 40 인 경우를 나타낸다. Process Time 의 경우에는 TIme Quantum 이 변하더라도 영향을 받지 않지만, Time Quantum 이 작은 경우 개별 프로세스들이 더 빠르게 종료되어 I/O 진입을 빠르게 하게 된다. 결론적으로 Time Quantum 이 큰 경우에 I/O 가 더 빠르게 시작하고 빠르게 종료되기 때문에 실제 실행 결과에서는 위와 같은 차이를 보이게 된다.

8. Personal feelings

이번 프로젝트는 여태 해왔던 과제들과 다르게 처음으로 팀원과 함께 진행하는 프로젝트라 남다르게 다가왔던 것 같다. 과제의 난이도도 있지만 코드 관리부터 방향성 설정, 커뮤니케이션 등 익숙하지 않던 점들이 많았다. 하지만 팀으로 진행하며 좋았던 점이 많았다. 우선, 프로젝트를 진행하면서 팀원과 많은 대화를 하다 보니 어떤 부분이 부족한지 자연스레 알게 되고 다시 한번 검색해보고 강의를 듣다 보니 확실하게 정리하고 넘어갈 수 있었다. 방향성에 대해 대화를 나누면서 우선순위를 확실하게 정하였고, 각자의 업무를 분담하여 이전보다 생산적이고 효율적으로 마무리할 수 있었던 것 같다.

저번 멀티 스레드 과제와 동일하게 이번 과제도 기존의 코드를 활용해서 작성하다 보니 초반에 수월하게 진행할 수 있어서 좋았다. 비교적 부담이 적다 보니 스케줄링 개념에 대해 좀 더 검색해볼 수 있었고, 코드로 어떻게 구현할지 충분하게 생각해 볼 수 있었다. 덕분에 Round Robin 을 이용한 기본 구현은 수월하게 진행이 됐다.

하지만 I/O 작업을 추가하는 과정에서 제일 많은 어려움을 느꼈던 것 같다. CPU 작업에서 특정 프로세스가 끝나 Wait queue 로 이동한 뒤 CPU 작업과 I/O 작업이 동시에 실행되는지, I/O 작업에서는 어떤 알고리즘을 사용해야 하는지, 만약 I/O 작업에 Round Robin 알고리즘을 사용한다면 time quantum 을 모두 사용하지 않고 프로세스가 종료되면 이후 어떻게 처리할지, signal handler 를 하나 더 만들어야 할지 등 방향성 부분에서 제일 어려움을 겪었다. 또한, 구현하는 과정에서도 마찬가지였다. I/O 작업 시간을 알려주기 위해 message queue 를 사용하였고, 그 과정에서 메시지 관련된 오류인 “stack smashing” 혹은 메시지 전송을 기다리는 등의 오류가 발생하여 어려움을 겪었다.

이외에도 여태 사용해보지 않았던 sprintf, read, write 와 같은 함수들을 사용하는 과정에서 파일 일부가 잘리는 등 구현에 큰 어려움을 겪었다.

중간에 어려움들도 많았지만 계획했던 내용을 모두 구현하여 만족감과 성취감이 크게 다가온다. 혼자서 진행했으면 아마 모두 구현하지 못하였겠지만 팀원 덕분에 기간 내에 완료한 것 같다. 결과적으로 팀 프로젝트로 진행하며 느꼈던 점들이 많았고, 이번 프로젝트 뿐만 아니라 다음 프로젝트에서도 함께 협업하여 좋은 결과물을 만들어 내고 싶다.