# 현실적인 입출력 환경을 반영한 모듈 기반 CPU 스케줄링 시뮬레이터의 설계와 알고리즘 비교 분석

**2022130802**

**영어영문학과**

**장세린**

**I. 서론**

**II. 본론**

1. 다른 cpu 스케줄링 시뮬레이터
2. 시뮬레이션 시스템의 구성
3. 시뮬레이션의 흐름
4. Non-preemptive 알고리즘 모듈
5. Preemptive 알고리즘 모듈
6. 라운드로빈
7. Preemptive priority with aging
8. Multi-level queue
9. 알고리즘 간의 비교와 Do All and Compare

**III. 결론**

# I. 서론

컴퓨터는 데이터를 입력받아 처리하고, 그 결과를 저장하거나 출력하는 전자적 정보처리 시스템이다. 초기에는 단순한 계산기 형태로 시작했지만, 현재의 컴퓨터는 복잡한 연산, 대용량 데이터 처리, 그리고 창의성이 요구되는 예술 분야까지 아우르며 눈부신 발전을 이뤄냈다. 이러한 발전의 핵심에는 여러 프로그램을 동시에 실행할 수 있는 멀티태스킹 능력이 있으며, 이를 가능케 하는 핵심 기술 중 하나가 바로 **CPU 스케줄링**이다.

만일 하나의 cpu가 하나의 프로그램만 실행한다면, CPU 스케줄링은 필요하지 않다. 단일 프로세스가 cpu를 독점적으로 사용하며, 연산이 필요할 때 마다 cpu에 직접 접근하면 되기 때문이다. 그러나 실제 시스템에서는 하나의 CPU로 수많은 작업을 병렬적으로 처리해야 하는 상황이 빈번하다. 이를 해결하는 방법은 크게 두 가지로 나뉜다. 첫째는 물리적인 프로세서 개수를 늘리는 방식이며, 둘째는 하나의 프로세서를 여러 작업에 **시간을 분할하여** 사용하는 방식이다.

후자의 방식, 즉 하나의 프로세서가 여러 프로세스를 번갈아 실행하는 구조가 바로 **멀티프로그래밍(Multiprogramming)**이다. 대부분의 프로세스는 지속적으로 CPU를 사용하는 것이 아니라, 주기적으로 I/O 작업(예: 사용자 입력, 디스크 접근 등)을 요청하게 된다. 이때, CPU는 해당 I/O 작업이 완료되기를 기다려야 하며, 이는 곧 CPU가 유휴 상태(idle state)에 빠지는 것을 의미한다. 하지만 한정된 자원인 CPU를 I/O 대기 시간 동안 놀게 두는 것은 비효율적이다. 따라서 운영체제가 대기 중인 다른 프로세스들 중 하나를 선택하여 CPU를 할당함으로써 자원의 효율을 극대화하는 비동기 입출력(Asynchronous I/O)이 고안되었다. 이때, 어떤 프로세스를 선택할 것인지, 어떤 순서로 실행할 것인지를 결정하는 정책이 바로 CPU 스케줄링 알고리즘이다. 이는 시스템의 처리량(Throughput), 응답 시간(Response Time), 공정성(Fairness) 등 다양한 성능 지표에 직접적인 영향을 주기 때문에, 적절한 알고리즘 선택은 시스템 성능을 좌우하는 중요한 과제가 된다.

필자는 이와 같은 스케줄링 메커니즘을 직접 체험하고 구조적 원리를 심층적으로 이해하기 위해, 다양한 CPU 스케줄링 알고리즘을 적용할 수 있는 단일 프로세서 기반의 시뮬레이터를 설계 및 구현하였다. 본 시뮬레이터는 interrupt-driven I/O 방식을 기반으로 하며, 임의의 시점에 다양한 I/O 작업이 발생하는 복수의 프로세스를 대상으로 한다. 사용자 입력에 따라 총 8가지 스케줄링 알고리즘(FCFS, non-preemptive SJF, preemptive SJF, non-preemptive priority, preemptive priority, round robin, preemptive priority with aging, multi-level queue) 중 하나를 선택하여 시뮬레이션을 실행할 수 있다. 시뮬레이션 종료 후에는 간트 차트를 포함하여 각 프로세스의 대기 시간(wait time)과 반환 시간(turnaround time)이 출력된다.

본 실습은 단순한 기능 구현을 넘어, 각 알고리즘의 설계 목적과 실제 성능 사이

의 관계를 검토하고, 스케줄링 전략의 선택이 자원 활용도 및 사용자 경험에 어떤 영향을 주는지를 체험할 수 있는 기회가 되었다. 이를 통해 운영체제의 핵심 기능인 CPU 스케줄링을 보다 실용적이고 입체적으로 이해할 수 있었다.

# II. 본론

## 1. 다른 스케줄링 시뮬레이터와의 비교

일반적으로 학습용으로 제작되는 CPU 스케줄링 시뮬레이터는 I/O 작업을 고려하지 않고, ready queue를 중심으로 한 단순한 시간 흐름 기반의 모델에 집중한다. 이러한 방식은 스케줄링 알고리즘의 기본 동작 원리를 학습하는 데는 유용하지만, 실제 운영체제 환경에서의 프로세스 관리와는 다소 거리가 있다. 이에 비해, 본 시뮬레이터는 보다 현실적인 운영체제의 동작을 모사하기 위해 다양한 기능과 구조적 차별성을 반영하였다.

우선, 본 시뮬레이터는 interrupt-driven I/O 모델을 채택하여, 프로세스가 I/O 요청을 하면 wait queue로 이동하고, 일정 시간이 경과한 후 자동으로 ready queue로 복귀하는 구조를 구현하였다. 이를 통해 CPU가 I/O 대기 시간 동안 유휴 상태로 머무는 비효율을 방지하고, 프로세스 간 병렬 실행 환경을 보다 정밀하게 시뮬레이션할 수 있다.

또한, 모듈화된 구조는 본 시뮬레이터의 가장 큰 장점 중 하나이다. 비교 함수 포인터를 인자로 받아 다양한 정렬 기준을 처리할 수 있도록 설계된 non-preemptive scheduling 모듈은 FCFS, SJF, Priority와 같은 알고리즘을 하나의 코드 구조에서 유연하게 지원하며, preemptive scheduling 모듈 역시 preemptive SJF와 priority 알고리즘을 효율적으로 통합 구현할 수 있도록 구성되었다. 여기에, 5가지 이벤트(프로세스 완료, I/O 요청, time quantum 만료, 선점 조건 발생, CPU 유휴 상태)를 통합적으로 처리하는 Context Switch 모듈을 통해, 다양한 알고리즘이 혼합된 Multi-Level Queue 구조도 일관되고 간결하게 구현할 수 있다.

Multi-Level Queue 알고리즘 역시 단순히 큐를 여러 개로 나누는 수준을 넘어서, 각 레벨마다 서로 다른 스케줄링 알고리즘을 적용할 수 있도록 하였으며, 특히 Round Robin 사용 시에는 레벨별로 개별적인 time quantum을 설정할 수 있게 하여, 실제 운영체제의 scheduling class와 유사한 유연성을 갖추었다.

뿐만 아니라, starvation 문제를 해결하기 위한 aging 기법을 직접 구현하였다는 점에서도 본 시뮬레이터는 일반적인 모델과 차별화된다. 우선순위 기반 스케줄링에서 발생할 수 있는 starvation을 방지하기 위해, 대기 시간이 누적될수록 priority를 점진적으로

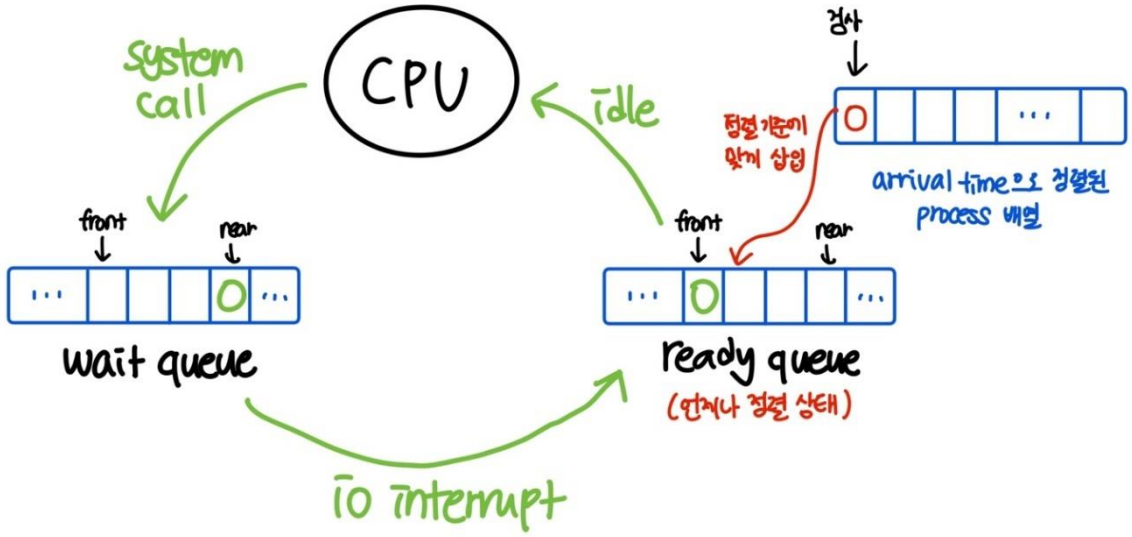
향상시키는 aging 알고리즘을 적용함으로써 공정성을 제고하였다.

내부적으로는 new, quantum_left, io_duration과 같은 상태 추적 변수들을 정교하게 활용하여, 문맥 교환 발생 여부나 라운드로빈의 시간 관리, 정확한 wait time 계산 등이 자동화되어 시뮬레이션 흐름의 정밀도와 안정성이 높아졌다. 특히, 라운드로빈 알고리즘에서는 quantum_left와 new 플래그를 활용해 프로세스의 교체 여부를 감지하고 적절한 시간 할당을 동적으로 갱신할 수 있다.

마지막으로, 평가 모듈(evaluation)의 정밀성과 편의성 또한 큰 장점이다. wait time 계산 시 I/O 대기 시간을 제외하여 실제 ready queue에서의 대기만을 반영하고, 각 프로세스의 turnaround time과 wait time을 arrival time 기준으로 정렬하여 출력하며, 간트 차트를 함께 제공함으로써 결과를 직관적으로 분석할 수 있도록 하였다. 특히 Do all and compare 모듈을 통해 동일한 프로세스 인스턴스에 대해 8가지 알고리즘을 일괄 실행하고, 그 평균 wait time과 turnaround time을 비교함으로써 알고리즘 간 성능을 편리하게 평가할 수 있다.

이처럼 본 시뮬레이터는 단순한 알고리즘 구현을 넘어서, aging이나 multi-level queue와 같은 고급 스케줄링 기법을 실제 운영체제 수준에 가깝게 모사하고자 하였다. 또한, 간트 차트 시각화, 정밀한 시간 측정, 비교 실행 기능 등을 통해 학습 도구를 넘어 스케줄링 전략을 체계적으로 비교·분석할 수 있는 실용적 도구로서도 활용 가능하다.

## 2. 시뮬레이터 시스템의 구성



위 그림은 구현한 시뮬레이터 작동을 단순화한 것이다. 시뮬레이터는 실제 컴퓨터 시스템의 구조를 단순화하여, CPU 스케줄링의 핵심 요소인 CPU, 메모리(ready queue 및 wait queue), 그리고 하나의 I/O 장치만을 모사하였다. 이 중 ready queue와 wait queue는 고정 길이의 원형 큐(circular queue)로 구현되었으며, 각각의 큐는 front, rear 인덱스와 큐에 저장된 프로세스의 개수를 함께 관리한다.

```
typedef struct {
    Process* arr[queue_capacity];
    int front;
    int rear;
    int size;
} Queue;
```

<큐 구조체>

ready queue는 항상 선택된 스케줄링 알고리즘의 정렬 기준(FIFO, 우선순위, 남은 시간 등)에 따라 정렬된 상태로 유지되며, 이 기준은 프로세스를 큐에 삽입할 때마다 적용된다. 반면, wait queue는 단순히 I/O 작업 완료를 대기하는 용도이므로, 정렬되지 않은 상태로 유지된다.

시뮬레이터에서 스케줄링 대상으로 사용되는 프로세스는 pid, arrival time, burst time, remain time, io_duration, priority, finish time 정보를 포함한다. 각 정보에 대한 자세한 설명은 아래 사진에서 확인할 수 있다.

```
typedef struct {
    int pid; //프로세스를 특정한다
    int arrival_time;//프로세스가 발생하는 시점(ready queue에 삽입되는 시점)
    int burst_time;//cpu 작업이 필요한 시간의 합
    int remain_time;//남은 cpu 작업 시간
    int io_duration;//총 io 작업 시간(wait queue에서 대기하는 시간의 합)
    char priority;//우선순위, 작을 수록 좋다
    int finish_time;//프로세스의 작업이 종료된 시점

    IO_operation io[max_io];
    int io_count;
} Process;
```

<프로세스 구조체>

프로세스가 요청하는 I/O 작업은 고정 크기의 배열 형태로 io 배열에 저장되며, 시뮬레이션에서는 각 프로세스에 최대 5개의 I/O 작업이 무작위로 할당된다. 각 I/O 작업은 start_time과 duration을 가지는 구조체로 정의된다.

```c
typedef struct {
    int start_time;
    int duration;
} IO_operation;
```

<io_operation 구조체>

start_time은 프로세스가 io 요청을 발생시키는 시점을 나타내며, 이는 CPU 실행 시간의 누적값으로 계산된다. 예를 들어, 어떤 I/O 작업의 start_time이 n이라면, 해당 프로세스는 CPU에서 n시간 동안 실행된 후, 해당 시점에서 I/O 요청(system call)을 발생시킨다. 이를 구현하기 위해 시뮬레이터는 burst_time과 remaining_time을 별도로 유지하며, 시뮬레이터는 **burst_time - remain_time == io.start_time** 일 때 프로세스가 때 해당 프로세스가 I/O 요청을 발생시킨 것으로 판단한다.

## 3. 시뮬레이션의 흐름

시뮬레이터를 실행하면, 사용자는 생성할 프로세스의 개수와 생성 방식을 선택하게 된다. 생성 방식은 수동 입력과 자동 생성 중 하나를 선택할 수 있으며, 수동 입력의 경우 사용자 정의로 arrival time, burst time, priority를 직접 지정할 수 있다. 단, I/O 작업의 할당은 모든 방식에서 무작위(random)로 이루어지며, 사용자가 직접 설정할 수는 없다.

각 프로세스에는 최소 burst time - 2개에서 최대 5개의 I/O 작업이 할당된다. 이는 프로세스의 시작 직후나 종료 직전에 I/O 요청이 발생하지 않도록 하기 위한 제약 조건이며, 따라서 burst time이 2 이하인 프로세스에는 io 작업이 생성되지 않는다. 생성된 I/O 작업은 start_time 값의 중복을 제거한 뒤, 시작 시점 기준으로 정렬된다. 전체 프로세스는 arrival time 순서로 정렬되어 배열되며, 생성이 완료되면 시뮬레이터는 각 프로세

스를 pid 순으로 출력한다.

```
==== Process 1 ====
PID          : 1
Arrival Time : 9
Burst Time   : 1
priority    : 3
I/O Count    : 0
No I/O operations.

==== Process 2 ====
PID          : 2
Arrival Time : 1
Burst Time   : 6
priority    : 2
I/O Count    : 3
I/O Operations:
  [I/O 0] Start: 1, Duration: 3
  [I/O 1] Start: 2, Duration: 4
  [I/O 2] Start: 5, Duration: 2

==== Process 3 ====
PID          : 3
Arrival Time : 6
Burst Time   : 9
priority    : 0
I/O Count    : 2
I/O Operations:
  [I/O 0] Start: 2, Duration: 2
  [I/O 1] Start: 5, Duration: 2
```

<프로세스 생성 후 출력 예시>

그다음, 사용자는 시뮬레이션에 사용할 스케줄링 알고리즘을 선택한다. 시뮬레이터는 총 8가지 알고리즘(FCFS, non-preemptive SJF, preemptive SJF, non-preemptive Priority, preemptive Priority, Round Robin, preemptive Priority with aging, Multi-Level Queue)을 제공하며, 사용자로부터 알고리즘 번호를 입력받아 해당 알고리즘 함수에 함수 포인터를 연결하는 방식으로 설정된다.

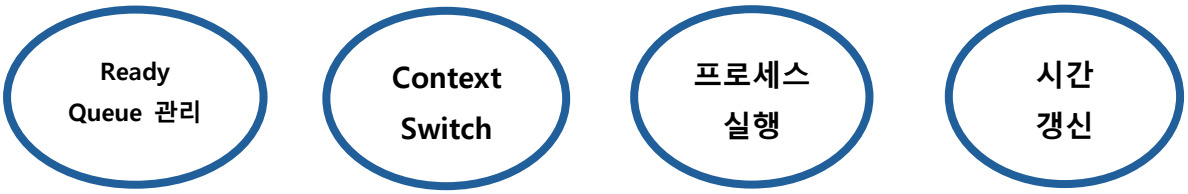
```
어떤 알고리즘으로 CPU sheduling 할까요?

1. FCFS   2. Non-preemptive SJF   3. Preemptive SJF
4. Non-preemptive priority   5. Preemptive priority   6. Round Robin
7. Preemptive priority with aging   8. multi-level   9. Do all and compare

실행할 알고리즘의 번호: 5
```

<알고리즘 선택창 예시>

9번 <do all and compare>을 선택하면, 시뮬레이터는 1번부터 6번까지의 알고리

즘으로 순차적으로 스케줄 사뮬레이션 후 각 알고리즘의 평균 wait time과 turnaround time을 출력한다. 어떤 알고리즘이 생성된 프로세스들을 스케줄링하는 데 유리한지 비교해볼 수 있다.

시뮬레이터는 시간 단위(time unit)로 동작하며, 매 시간마다 다음과 같은 루틴이 반복된다.




이 과정에서 프로세스는 Ready → Running → Waiting 상태로 순환하며, 실행 중 다음과 같은 이벤트가 발생할 수 있다:

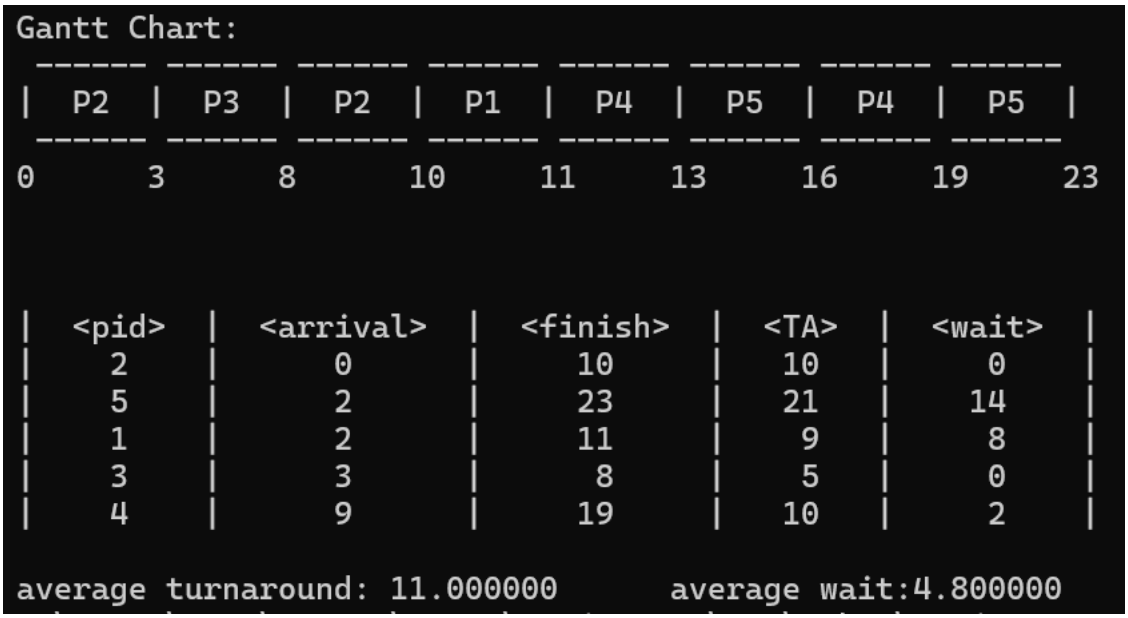
1. CPU가 idle
2. I/O 작업 요청(system call)
3. 프로세스 완료
4. 선점(preemption)
5. 타임 퀀텀 만료(Round Robin인 경우)

이벤트가 발생하면 시뮬레이터는 이벤트 코드와 함께 Context Switch 모듈을 호출한다. context switch 모듈은 각 이벤트 코드를 바탕으로 적절한 상태 전환 및 큐 이동을 수행한다. 모든 Context Switch 활동은 로그(log)로 기록되며, 각 로그 항목에는 이벤트를 발생시킨 프로세스의 pid, 이벤트 종류, 발생 시각, 그리고 이벤트 발생 직전까지 프로세스가 연속적으로 실행된 시간이 함께 저장된다.

```
typedef struct {
    int event_time;
    int event_code;
    int pid;
    int duration;
} TimeLog;
```

<log 구조체>

시뮬레이션이 종료되면, 기록된 로그를 기반으로 간트 차트(Gantt Chart)가 생성되어 출력되며, 각 프로세스의 arrival time, finish time, I/O duration 정보를 바탕으로 turnaround time과 waiting time이 계산된다. 이 값들은 표 형태로 정리되어 출력되며, 마지막으로 전체 프로세스의 평균 turnaround time과 waiting time이 함께 출력되며 시뮬레이터가 완전히 종료된다.




<스케줄링 결과 출력창 예시>

## 4. Non-preemptive 알고리즘

시뮬레이터는 FCFS, SJF, non-preemptive Priority의 세 가지 non-preemptive 스케줄링 알고리즘을 하나의 공통 모듈로 통합하여 구현하였다. 이 모듈은 비교 함수 포인터를 인자로 받아, 각 알고리즘에 맞는 정렬 기준에 따라 ready queue를 정렬하도록 설계되었다. FCFS와 SJF 역시 priority 기반 알고리즘의 한 변형으로 볼 수 있으며, 이들 간의 유일하고 결정적인 차이는 ready queue를 어떤 기준으로 정렬하는가에 있다.

해당 모듈은 다음의 경우마다 입력받은 비교 함수를 사용해 ready queue의 정렬 상태를 유지한다:

- 새 프로세스가 도착하여 ready queue에 삽입될 때
- I/O 작업을 마친 프로세스가 wait queue에서 ready queue로 복귀할 때

다음은 non-preemptive 알고리즘 모듈의 동작을 간략히 표현한 수도 코드이다.

```
Module NonPre_Scheuduler (*compare)
        Do Until (All_Process_Completed) {
                Push_Ready_Queue (ready_queue, compare)
                Check_IO_Interrupt(wait_queue, ready_queue, compare)
                if (CPU_is_idle) current = Context_Switch(START)
                else {
                        if (Process_Complete) current = Context_Switch(COMPLETE)
                        else if (IO_Syscall) current = Context_Switch(IO)
                }
        if (current != NULL) current.remain_time--
        time++
        }
```

시뮬레이션은 프로세스 생성이 완료된 후, 프로세스 배열을 arrival_time 기준으로 정렬한 뒤 시작된다. 시뮬레이터는 시간을 1초 단위로 증가시킨다. 매시간 시뮬레이터의 첫번째 작업은 ready queue에 실행이 준비된 모든 프로세스를 정렬 순서를 지키며 담는 것이다. ready queue에 새로운 프로세스가 담기는 경우는 두 가지이다. 새로운 프로세스가 도착했거나 io 작업 중이던 프로세스가 작업을 완료하고 복귀하는 경우이다.

먼저, 새롭게 도착한 프로세스를 찾기 위해, 프로세스의 배열의 앞에서부터 현재 시간과 arrival time이 같은 프로세스를 찾아 ready queue에 정렬 순서를 지키며 삽입하는 Push_Ready_Queue 모듈을 호출한다. 다음은 복귀하는 프로세스를 찾을 차례이다. Check_IO_Interrupt() 함수를 통해 wait queue를 순회하며 I/O가 완료된 프로세스를 찾아 ready queue에 삽입한다. 이때 해당 프로세스의 io_count를 감소시키고, 완료된 I/O 작업은 배열에서 제거한다. I/O 배열은 항상 start_time 순으로 정렬되어 있으므로, 배열의 맨 앞 원소만 확인하면 다음 I/O 발생 여부를 효율적으로 판단할 수 있다. ready queue 내부는 정렬 기준 상 동등한 우선순위를 갖는다면, ready queue에 삽입된 시간을 기준으로, 삽입된 시간도 같다면, pid의 오름차순으로 정렬된다.

이후, 시뮬레이터는 CPU에서 context switch가 필요한지를 검사한다. Non-

preemptive 구조이므로 선점(preemption)은 없지만, 다음 세 가지 경우에는 context switch가 발생한다:

1. cpu가 유휴상태(idle state)인 경우(current 프로세스 포인터==NULL).

2. 작업 중이던 프로세스가 완료된 경우(프로세스의 remain time==0).

3. 작업 중이던 프로세스가 io 작업을 요청하는 경우
   (burst_time - remain_time == io[0].start_time)

세 번째 조건은 I/O 배열의 첫 번째 원소만 확인함으로써 효율적으로 처리할 수 있다. 위 조건 중 하나라도 만족하면, 시뮬레이터는 context switch 모듈을 이벤트 코드와 함께 호출하여 cpu에 새로운 프로세스의 할당을 시도한다.

Context switch 모듈은 입력받은 이벤트 코드에 맞추어 필요한 작업을 수행한다. 만일 작업 중이던 프로세스가 완료되었음을 가리키는 COMPLETE를 코드로 받았다면, 프로세스의 finish_time을 갱신하고 cpu를 비운다. io 작업 요청 발생을 가리키는 코드 IO를 받았다면 현재 작업 중이던 프로세스를 wait queue에 삽입한다. cpu가 유휴상태임을 가리키는 START를 코드로 입력받았다면, 특별한 작업은 수행되지 않으며, ready queue가 비어있는지 확인하고 비어있지 않다면 ready queue의 front 인덱스가 가리키는 프로세스에 cpu를 할당하는 작업이 공통적으로 수행된다. 이때 ready queue가 비어 있다면, CPU 할당은 실패하며 context switch는 NULL을 반환한다. 문맥 전환에 성공하면, 이벤트 발생 시각, pid, 이벤트 코드, 연속 실행 시간 등이 로그에 기록된다.

마지막으로, CPU에 할당된 프로세스가 있다면 1초간 실행된 것으로 간주하고 remaining_time을 1 감소시킨다. 이후 시뮬레이션 시간을 1초 갱신하고, 모든 프로세스가 완료될 때까지 이 과정을 반복함으로써 스케줄링이 종료된다.

## 5. Preemptive 알고리즘

필자가 구현한 시뮬레이터는 non-preemptive 알고리즘과 마찬가지로, 두 가지 Preemptive 알고리즘(Preemptive SJF, Preemptive priority)를 하나의 통합된 모듈로 구현하였다. 이 이 preemptive 알고리즘 모듈은 전체 구조와 흐름 면에서 non-preemptive 알고리즘 모듈과 거의 동일하나, Context Switch를 발생시키는 조건에 "선점(Preemption)" 여부가 추가된다는 점만 다르다. 다음은 preemptive 모듈의 동작을 간략히 표현한 수도 코드이다. 형광펜 처리된 조건문 하나 외에는 non-preemptive의 수도 코드와 전부 동일하다.

```
Module Pre_Scheuduler (*compare)
    Do Until (All_Process_Completed) {
        Push_Ready_Queue (ready_queue, compare)
        Check_IO_Interrupt(wait_queue, ready_queue, compare)
        if (CPU_is_idle) current = Context_Switch(START)
        else {
            if (Process_Complete) current = Context_Switch(COMPLETE)
            else if (IO_Syscall) current = Context_Switch(IO)
            else if (Check_Preemption(ready_queue, current){
                current = Context_Switch(PREEMPTION)
            }
        }
    if (current != NULL) current.remain_time--
    time++
    }
```

시뮬레이션은 우선 프로세스 배열을 arrival_time 순으로 정렬한 후 시작된다. 매초 시뮬레이터는 먼저 arrival_time 체크와 io_interrupt 체크를 통해 ready queue에 적절한 프로세스를 정렬 순서를 유지한 채로 담는다. 이후, 시뮬레이터는 CPU에서 Context Switch가 필요한지를 다음 네 가지 조건을 기준으로 확인한다:

1. cpu가 유휴상태(idle state)인 경우(current 프로세스 포인터==NULL).
2. 작업 중이던 프로세스가 완료된 경우(프로세스의 remain time==0).
3. 작업 중이던 프로세스가 io 작업을 요청하는 경우
   (burst_time - remain_time == io[0].start_time)
4. 선점이 필요한 경우
   :현재 실행 중인 프로세스와 ready queue의 가장 앞에 있는 프로세스를 비교

했을 때, compare(current, ready_queue[front]) > 0인 경우 (즉, 우선순위가 높은 프로세스가 대기 중일 때)

이 네 번째 조건은 preemptive 모듈에만 존재하며, 모듈 호출 시 전달된 비교 함수 포인터를 통해 두 프로세스를 비교함으로써 선점 여부를 결정한다. 조건들은 순서대로 평가되기 때문에, 먼저 만족되는 조건이 우선적으로 처리된다. 따라서 IO 작업 요청과 선점 조건이 동시에 발생할 경우, 시뮬레이터는 io 작업 요청을 우선 처리한다. 이는 프로세스가 ready queue에서 대기하기보다, wait queue에서 io 작업을 수행할 수 있도록 하여 CPU와 IO 자원의 동시 활용을 극대화하려는 의도적 조건 배치이다.

Context Switch 모듈은 이벤트 코드로 PREEMPTION이 전달되었을 경우, 현재 CPU에서 실행 중이던 프로세스를 ready queue에 다시 삽입하고, ready queue의 첫 번째 프로세스를 새롭게 CPU에 할당한다. 이 과정은 모두 로그에 기록되며, 로그에는 pid, 이벤트 시각, 이벤트 코드, 그리고 연속 실행 시간 등의 정보가 포함된다.

마지막으로, CPU에 프로세스가 정상적으로 할당되어 있다면 해당 프로세스의 remaining_time을 1 감소시키며, 이 루프를 모든 프로세스의 작업이 완료될 때까지 반복하면 시뮬레이션이 종료된다.

## 6. 라운드로빈

Round Robin(RR) 알고리즘은 각 프로세스에 동일한 실행 시간을 순차적으로 부여함으로써, 응답 시간을 줄이고 starvation을 방지하는 데 효과적인 방식이다. 본 시뮬레이터에서의 RR 모듈은 구조적으로는 non-preemptive 알고리즘과 유사하지만, 프로세스에 할당된 time quantum이 만료되었을 때 선점(context switch)을 발생시킨다는 점에서 차이가 있다. 이외의 동작 흐름은 대부분 동일하게 설계되었다. 다음은 라운드 로빈 모듈의 동작을 간략히 나타낸 수도 코드 구조이다.

```
Module Round Robin (*compare)
    Do Until (All_Process_Completed) {
        new = 1
        Push_Ready_Queue (ready_queue, compare)
        Check_IO_Interrupt(wait_queue, ready_queue, compare)
```

```
        if (CPU_is_idle) current = Context_Switch(START)
        else {
                if (Process_Complete) current = Context_Switch(COMPLETE)
                else if (IO_Syscall) current = Context_Switch(IO)
                else if (quantum_left <= 0) {
                        if (!Is_Empty(ready_queue)){
                                current = Context_Switch(EXPIRED)
                        }
                        else new = 0
                }
                else new = 0
        }
        if (current != NULL) {
                current.remain_time--
                if (new == 1) quantum_left = time_quantum
                quantum_left –
        }
time++
}
```

시뮬레이션은 프로세스 배열을 arrival_time 기준으로 정렬한 후 시작된다. 시뮬레이터는 먼저 현재 시점에 도착한 프로세스를 ready queue에 순서대로 삽입하고, wait queue에서 I/O 작업을 완료한 프로세스도 ready queue의 맨 뒤에 삽입한다. 라운드로빈 알고리즘은 도착 순서대로 프로세스를 실행하기 때문에 추가적인 정렬 작업은 필요하지 않다. 프로세스 배열은 arrival time이 같은 경우 pid가 작은 것이 더 앞에 존재하며, arrival time 확인 후 io 작업 완료를 확인하므로, ready queue 내부는 알고리즘의 흐름에 따라 자연히 다음과 같은 우선순위로 정렬되게 된다.

1. ready queue에 삽입된 시간

2. (1이 같을 경우) 처음으로 ready queue에 삽입된 프로세인지 여부

3. (2가 같을 경우) pid 오름차순

이제 시뮬레이터는 CPU에 Context Switch가 필요한지 확인한다. 라운드로빈 알고리즘은 다음 4가지 조건 중 하나가 충족될 경우 Context Switch 모듈을 호출하며, 조건을 순서대로 확인함에 따라 먼저 확인하는 조건을 우선하게 된다.

1. cpu가 유휴상태(idle state)인 경우(current == NULL)

2. 작업 중이던 프로세스가 완료된 경우(프로세스의 remain time == 0)

3. 작업 중이던 프로세스가 io 작업을 요청하는 경우
   (burst_time - remain_time == io[0].start_time)

4. 프로세스에 할당된 시간(quantum_left) 만료 & ready queue가 비어있지 않음

따라서 시뮬레이터는 io 작업 요청을 할당시간 만료보다 우선하여 스케줄링 효율이 증가하게 된다. 또한, 4번 조건에 따라, 시뮬레이터는 time quantum이 만료되었더라도 ready queue가 비어 있는 경우에는 context switch를 호출하지 않는다. 이는 CPU의 불필요한 문맥 교환을 방지하고 작업 효율을 높이기 위한 설계이며, 이를 위해 quantum이 음수가 될 수 있도록 허용하였다.

RR 모듈에서는 프로세스에게 할당된 시간의 소진 여부를 판단하기 위해 quantum_left 변수를 사용하며, 새로운 프로세스가 CPU에 할당되었을 경우 해당 값을 지정된 time_quantum으로 갱신해야 한다. 이를 위해 new 플래그가 사용되며, 이 플래그는 매 시간마다 기본적으로 1로 초기화되고, context switch가 발생하지 않을 경우 0으로 변경된다. 만약 new == 1이고, CPU에 프로세스가 존재한다면, quantum_left는 time_quantum으로 갱신된다.

마지막으로, CPU에 프로세스가 정상적으로 할당되어 있을 경우, remain_time을 1 감소시키고, quantum_left도 1 감소시킨다. 이 모든 과정을 모든 프로세스의 작업이 완료될 때까지 반복하여 시뮬레이션을 종료한다.

## 7. Preemptive priority with aging

이 모듈은 기본 Preemptive Priority 알고리즘의 변형으로, 사용자 지정 속도인

aging rate에 따라 ready queue 내 대기 중인 프로세스들의 priority 값을 주기적으로 감소시킨다. priority 값이 작을수록 높은 우선순위를 의미하므로, 시간이 지날수록 오래 기다린 프로세스의 실행 우선순위가 자연스럽게 높아지게 된다. 이 방식은 preemption을 허용하는 스케줄링 알고리즘에서 발생할 수 있는 starvation 현상, 즉 우선순위가 낮은 프로세스가 계속 밀려 영원히 실행되지 못하는 문제를 완화하기 위해 고안되었다. 다음은 이 모듈의 동작을 간략히 나타낸 수도 코드 구조이다.

```
Module Pre_Priority_Aging (*compare)
    Do Until (All_Process_Completed) {
        Push_Ready_Queue (ready_queue, compare)
        Check_IO_Interrupt(wait_queue, ready_queue, compare)
        if (CPU_is_idle) current = Context_Switch(START)
        else {
            if (Process_Complete) current = Context_Switch(COMPLETE)
            else if (IO_Syscall) current = Context_Switch(IO)
            else if (Check_Preemption(ready_queue, current){
                current = Context_Switch(PREEMPTION)
            }
        }
    if (current != NULL) current.remain_time—
    Aging(ready_queue)
    time++
    }
```

이 모듈의 동작 구조는 기본 preemptive priority 알고리즘과 거의 동일하다. 즉, 다음의 과정은 모두 기존 모듈과 동일하게 수행된다:

- 도착한 프로세스와 I/O 복귀 프로세스를 ready queue에 정렬 순서를 유지하며

삽입

- CPU 내 실행 중인 프로세스에 대해 작업 완료, I/O 요청, 선점 조건을 확인하고 Context Switch 모듈 호출

이 모듈에서의 유일한 차이점은, CPU에 프로세스가 할당된 후 remaining_time을 1 감소시킨 직후, 별도의 Aging 모듈을 호출한다는 점이다. 이 Aging 모듈은 ready queue에 존재하는 모든 프로세스의 priority를 일정한 비율(aging rate, priority/1s)만큼 일괄 감소시키며, 모든 프로세스에 동일한 변동이 적용되므로 ready queue의 정렬 순서를 새로 계산할 필요는 없다.

이후 과정은 다른 모듈과 마찬가지로, CPU에 프로세스가 존재할 경우 remaining_time을 1 감소시키며, 모든 프로세스가 종료될 때까지 이 루프를 반복하여 시뮬레이션을 마무리한다.

## 8. Multi-Level Queue

Multi-Level Queue 알고리즘 모듈은 priority에 따라 여러 개의 ready queue를 생성하고, 각 큐에 서로 다른 알고리즘을 적용할 수 있도록 설계되었다. 실습에서는 priority의 범위를 0~4로 지정하여 총 5개의 ready queue를 사용하였으며, 이는 priority 범위에 따라 유동적으로 조정 가능하다. 이 모듈은 feedback을 통한 레벨 이동을 지원하지 않기 때문에, 프로세스는 생성 당시 부여된 priority에 따라 해당 레벨의 큐에 고정적으로 배치된다

각 레벨에 지정할 수 있는 알고리즘은 nonpreemptive FCFS, nonpreemptive SJF, preemptive FCFS, preemptive SJF, RR로 총 다섯 가지이며, 라운드로빈(RR)을 사용하는 레벨의 경우, 각 레벨마다 서로 다른 time quantum을 개별 설정할 수 있다. 각 priority 레벨에 적용할 알고리즘을 구분하기 위해 preemptive 여부와 time quantum 등의 정보를 Alg 구조체 배열에 저장한다. Alg 구조체는 각 레벨이 사용하는 알고리즘의 선점 여부(preemption), 정렬 기준(compare 함수), 그리고 라운드로빈일 경우 적용할 time quantum을 포함한다.

```
typedef struct {
    char preemption;
    CompareFunc compare;
    int time_quantum;
} Alg;
```

<alg 구조체>

preemption은 해당 큐의 알고리즘이 선점을 허용하는지 나타내는 플래그이다. compare는 정렬 기준 함수 포인터로, FCFS 및 RR은 NULL, SJF는 arrival_time 비교 함수, Priority는 priority 비교 함수가 저장된다. time_quantum은 해당 알고리즘이 RR일 경우 사용자로부터 입력받은 값이 저장되며, RR이 아닐 경우 구분을 위해 99999999와 같은 불가능한 값이 저장된다. 시뮬레이터는 프로세스의 priority를 인덱스로 하여 적절한 알고리즘 설정을 참조한다.

시뮬레이션이 시작되면 도착한 프로세스와 I/O 작업을 마친 프로세스가 자신의 priority에 해당하는 레벨의 ready queue에 삽입된다. 이를 위해 Push_Ready_Queue_M과 Check_IO_Completion_M 모듈이 사용되며, 각 프로세스의 priority를 기준으로 큐를 선택하고, 필요 시 정렬 기준에 따라 큐 내에서 정렬을 수행한다.

이후 시뮬레이터는 다음의 다섯 가지 조건을 기준으로 CPU에 context switch가 필요한지 여부를 확인한다:

1. cpu가 유휴상태(idle state)인 경우(current 프로세스 포인터==NULL)
2. 작업 중이던 프로세스가 완료된 경우(프로세스의 remain time==0)
3. 작업 중이던 프로세스가 io 작업을 요청하는 경우
   (burst_time - remain_time == io[0].start_time)
4. 스케줄링 알고리즘이 선점을 허용 & 선점이 필요
   (Alg[process.priority].preemption == 1 & Check_Preemption_M == 1)
5. 라운드로빈 알고리즘 & 할당된 시간 만료
   (Alg[process.priority].time_quantum<10000 & quantum_left <=0)

4번 조건에서는 먼저 해당 레벨의 알고리즘이 선점을 허용하는지 확인하고, 낮은 레벨의 큐가 모두 비어 있는지를 검사한 뒤, 현재 프로세스와 같은 레벨 큐의 첫 번째 프로세스를 비교하여 선점 여부를 판단한다. 5번 조건에서는 time quantum 값이 99999999가 아닌 경우에만 RR 알고리즘으로 간주하며, quantum이 0 이하로 감소했는지에 따라 context switch 여부를 결정한다.

이 조건 중 하나라도 만족되면 Context_Switch_M() 모듈이 호출된다. 이 모듈은 ready queue에 프로세스를 삽입하거나 CPU에 새로운 프로세스를 할당할 때 priority에 따라 큐를 선택하며, CPU 스케줄링 시 낮은 priority(높은 우선순위) 큐부터 탐색한다는 점에서 기존 모듈과 구분된다.

이제 시뮬레이터는 CPU에 작업 중인 프로세스가 존재하는지 확인 후 remain_time을 감소시키고 프로세스에 할당된 시간을 카운팅하는, quantum_left 값을 조

정해야한다. quantum_left는 Round Robin 스케줄링을 위한 time quantum을 추적하는 변수이며, 상황에 따라 다음과 같이 변동된다:

1. 스케줄링 알고리즘이 라운드로빈에서 다른 알고리즘으로 바뀐 경우
: quantum_left를 불가능한 값(99999999)으로 바꾼다.

2. 라운드로빈의 스케줄링을 받는 **새로운** 프로세스가 CPU에 할당된 경우
: quantum_left를 해당 레벨의 라운드로빈에 지정된 값으로 갱신한다.
(quantum_left = Alg[current.priority].time_quantum)

3. 라운드로빈 스케줄링 프로세스가 CPU에서 실행되고 있는 경우
: quantum_left의 값을 1 감소시킨다.

이 때 프로세스가 새로 CPU에 할당되었는지를 판단하기 위해 new 플래그가 사용된다. new는 매 사이클마다 1로 초기화되며, context switch가 발생하지 않거나 ready queue가 비어 있어 전환이 일어나지 않으면 0으로 갱신된다. new == 1이고 현재 프로세스가 RR 알고리즘에 속할 경우, 해당 time quantum 값으로 quantum_left가 재설정된다. 시뮬레이터는 이러한 과정을 모든 프로세스의 작업이 완료될 때까지 반복하며, 각 레벨의 알고리즘 특성에 따라 시뮬레이션이 유연하게 진행된다. 다음은 위와 같은 multi-level 모듈의 동작을 간략히 정리한 수도코드이다.

```
Module Multi_level (*compare)
    Do Until (All_Process_Completed) {
        new = 1
        Push_Ready_Queue_M (ready_queue, compare)
        Check_IO_Interrupt_M(wait_queue, ready_queue, compare)
        if (CPU_is_idle) current = Context_Switch_M(START)
        else {
            if (Process_Complete) current = Context_Switch_M(COMPLETE)
            else if (IO_Syscall) current = Context_Switch_M(IO)
            else if (Alg[current.priority].preemption & Check_Preemption_M)
                current = Context_Switch_M(PREEMPTION)
            }
            else if (Alg[current.priority].compare==NULL & quantum_left <= 0) {
```

```
                        if (!Is_Empty(ready_queue[current.priority]){
                                current = Context_Switch_M(EXPIRED)
                        else new = 0
                }
                else new = 0
        }
        if (current != NULL) {
                current.remain_time--
                if (new == 1) quantum_left = Alg[current.priority].time_quantum
                if (quantum_left != 99999999) quantum_left --
        }
time++
}
```

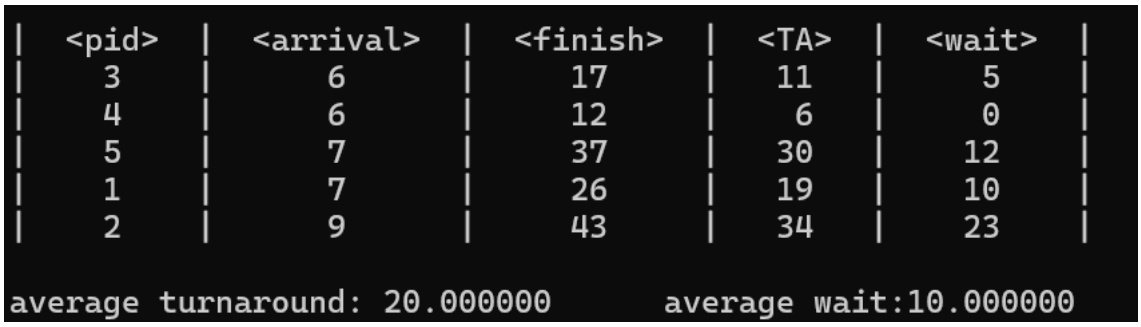
## 9. 알고리즘 간의 비교와 Do All and Compare 모듈

모든 프로세스에 대한 시뮬레이션이 완료되면, 시뮬레이터는 evaluation 모듈을 호출하여 알고리즘의 효율성을 평가한다. 이 모듈은 각 프로세스의 turnaround time과 wait time을 계산하여 결과를 표로 정리한다. Turnaround time은 프로세스가 도착한 시점부터 작업이 완료되기까지의 총 소요 시간을 의미하며, finish_time - arrival_time으로 계산된다. 반면, wait time은 프로세스가 실제로 CPU를 할당받지 못한 채 ready queue에서 대기한 시간을 의미하며, wait queue(입출력 대기)의 대기 시간은 포함되지 않는다.

그러나 초기 실습 데모에서는 이 점을 간과하여 wait time을 turnaround time - burst time으로 잘못 계산하였고, 그로 인해 I/O 작업이 많은 프로세스의 wait time이 과도하게 부풀려지는 오류가 발생하였다. 이후 피드백을 반영하여 프로세스 구조체에 전체 I/O 작업 시간의 합을 저장하는 io_duration 필드를 추가하고, wait time 계산식을 turnaround time - burst time - io_duration으로 수정하였다. 이 개선을 통해 시뮬레이터는 보다 정확하게 CPU 자원에 대한 대기 시간을 평가할 수 있게 되었다.

Evaluation 모듈은 각 프로세스의 turnaround time과 wait time을 도착 순서대로 정렬하여 출력하고, 전체 프로세스에 대한 평균 turnaround time 및 평균 wait time을 함

께 출력하여 알고리즘의 상대적 효율을 정량적으로 비교할 수 있도록 한다.

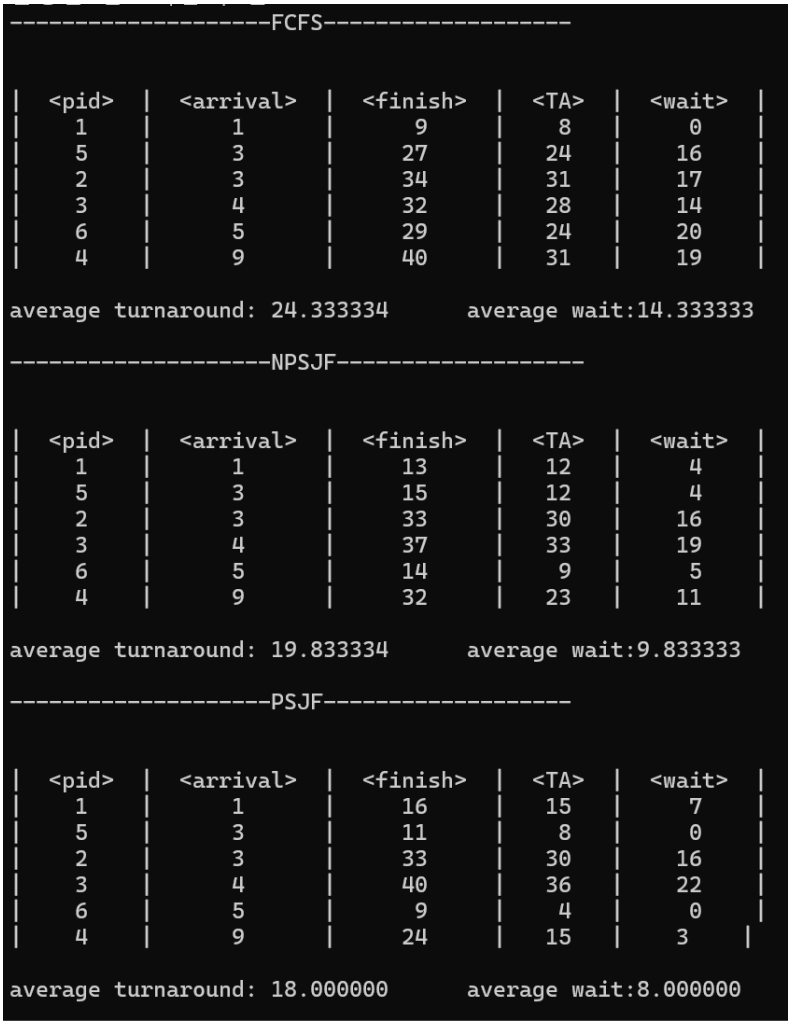


| <pid> | <arrival> | <finish> | <TA> | <wait> |
|---|---|---|---|---|
| 3 | 6 | 17 | 11 | 5 |
| 4 | 6 | 12 | 6 | 0 |
| 5 | 7 | 37 | 30 | 12 |
| 1 | 7 | 26 | 19 | 10 |
| 2 | 9 | 43 | 34 | 23 |

average turnaround: 20.000000 average wait:10.000000

<evaluation 모듈 출력 예시>

시뮬레이터는 알고리즘 선택창에서 유저가 옵션 9번(Do all and compare)을 선택할 경우, 동일한 프로세스 배열에 대해 총 7가지 알고리즘(FIFO, Non-preemptive SJF, Preemptive SJF, Non-preemptive Priority, Preemptive Priority, Round Robin, Preemptive Priority with Aging)을 연속 실행한 뒤, 각 알고리즘의 turnaround time과 wait time을 일괄적으로 출력한다.



----------FCFS----------

| <pid> | <arrival> | <finish> | <TA> | <wait> |
|---|---|---|---|---|
| 1 | 1 | 9 | 8 | 0 |
| 5 | 3 | 27 | 24 | 16 |
| 2 | 3 | 34 | 31 | 17 |
| 3 | 4 | 32 | 28 | 14 |
| 6 | 5 | 29 | 24 | 20 |
| 4 | 9 | 40 | 31 | 19 |

average turnaround: 24.333334 average wait:14.333333

----------NPSJF----------

| <pid> | <arrival> | <finish> | <TA> | <wait> |
|---|---|---|---|---|
| 1 | 1 | 13 | 12 | 4 |
| 5 | 3 | 15 | 12 | 4 |
| 2 | 3 | 33 | 30 | 16 |
| 3 | 4 | 37 | 33 | 19 |
| 6 | 5 | 14 | 9 | 5 |
| 4 | 9 | 32 | 23 | 11 |

average turnaround: 19.833334 average wait:9.833333

----------PSJF----------

| <pid> | <arrival> | <finish> | <TA> | <wait> |
|---|---|---|---|---|
| 1 | 1 | 16 | 15 | 7 |
| 5 | 3 | 11 | 8 | 0 |
| 2 | 3 | 33 | 30 | 16 |
| 3 | 4 | 40 | 36 | 22 |
| 6 | 5 | 9 | 4 | 0 |
| 4 | 9 | 24 | 15 | 3 |

average turnaround: 18.000000 average wait:8.000000

<Do all and compare 실행 예시, PSJF 이하 생략>

Do all and compare 모듈은 대기 시간 관점에서 각 알고리즘의 성능을 일괄적이고 편리하게 비교할 수 있도록 돕는다. 필자는 이 모듈을 이용하여, 각 알고리즘의 성능을 비교하기 위해, 프로세스 6개를 생성하여 성능을 비교하는 작업을 10회 반복하였다. 이때 RR 알고리즘의 time quantum은 2로, priority with aging 알고리즘의 aging rate은 0.5로 설정하였다. 다음은 각 시행에서 알고리즘 별 turnaround time과 wait time을 기록하여 정리한 표이다. AVG는 FCFS 대비 각 알고리즘의 turnaround time과 wait time의 크기의 비율의 10회 시행 동안의 평균을 계산한 것이다.

| | | | | | | | | | | | AVG |
|---|---|---|---|---|---|---|---|---|---|---|---|
| FCFS | turnaround | 15.83 | 24.33 | 15.5 | 17.5 | 13.66 | 24.83 | 31.5 | 29.3 | 14 | 26.33 | 100% |
| | wait | 7.16 | 14.33 | 9.16 | 11 | 6.66 | 15.66 | 21.33 | 18 | 6.83 | 15.6 | 100% |
| | | | | | | | | | | | | |
| NPSJF | turnaround | 13.33 | 19.83 | 14.66 | 13 | 11.83 | 16.83 | 28.16 | 24.33 | 12.66 | 23.66 | 84% |
| | wait | 4.66 | 9.83 | 8.33 | 6.5 | 4.83 | 7.66 | 18 | 13 | 5.5 | 13 | 73% |
| | | | | | | | | | | | | |
| PSJF | turnaround | 13.33 | 18 | 12.33 | 13.16 | 10.66 | 16.5 | 26.16 | 22.5 | 12.66 | 22.83 | 79% |
| | wait | 4.8 | 8 | 6 | 6.66 | 3.66 | 7.3 | 16 | 11.16 | 5.5 | 12.16 | 65% |
| | | | | | | | | | | | | |
| NPPri | turnaround | 21.5 | 21.33 | 15 | 17.33 | 13.5 | 24.33 | 27.5 | 24.66 | 13.66 | 26.16 | 98% |
| | wait | 12.83 | 11.33 | 8.66 | 10.83 | 6.5 | 15.16 | 17.33 | 13.33 | 6.5 | 15.5 | 100% |
| | | | | | | | | | | | | |
| PPri | turnaround | 21.3 | 21 | 14.8 | 17 | 12.5 | 23.5 | 28.16 | 23.16 | 13.66 | 25.83 | 96% |
| | wait | 12.6 | 11 | 8.5 | 10.5 | 5.5 | 14.3 | 18 | 11.83 | 6.5 | 15.16 | 96% |
| | | | | | | | | | | | | |
| RR(quantu | turnaround | 16 | 24.83 | 16.16 | 17.16 | 12.1 | 22.16 | 38 | 30.83 | 15.66 | 29.5 | 103% |
| | wait | 7.3 | 14.83 | 9.83 | 10.66 | 5.16 | 13 | 27.83 | 19.5 | 8.5 | 18.83 | 105% |
| | | | | | | | | | | | | |
| PPaging | turnaround | 16.66 | 25.83 | 15.5 | 18.33 | 13.16 | 25.5 | 31.5 | 26.66 | 13.5 | 25.16 | 100% |
| | wait | 8 | 15.83 | 9.16 | 11.83 | 6.16 | 16.33 | 21.33 | 15.33 | 6.33 | 14.5 | 100% |

예상과 달리, SJF 계열 알고리즘이 다른 방식에 비해 압도적으로 효율적인 결과를 보이진 않았다. 이는 시뮬레이터가 사용하는 프로세스 생성 방식의 특성에서 비롯된 것으로 보인다. 본 시뮬레이터는 burst time이 클수록 더 많은 I/O 작업이 할당될 확률이 높도록 설계되어 있으며, 최대 burst time은 10, I/O 작업은 최대 5개까지 배정 가능하다. 따라서 burst time이 긴 프로세스라고 하더라도 I/O 처리를 위해 빠르게 wait queue로 이동하는 경우가 많다. 그 결과, CPU를 장시간 독점하여 다른 작업들의 실행을 지연시키는 Convoy effect가 상대적으로 적게 발생하였다.

이러한 특성 때문에 알고리즘 간 평균 wait time과 turnaround time의 차이가 크게 벌어지지 않았다. 특히, 실제 컴퓨터 환경에서는 I/O 요청이 없더라도, 다양한 프로그램을 병렬적으로 실행하기 위해 context switching이 자주 발생하는 multi-tasking 구조가 일반적이다. 이런 점을 고려할 때, 단순히 평균 대기 시간을 줄이는 알고리즘이 항상 최적의 선택이라고 단정할 수는 없다.

결국, 스케줄링 알고리즘은 시스템의 목적과 요구사항에 따라 유연하게 선택되어야 하며, 아래는 각 알고리즘의 특성과 장단점을 요약한 내용이다.

1. **FCFS(First-Come, First-Served)는** 구현이 가장 단순하고 도착 순서대로 공정하게 처리된다는 장점이 있지만, 긴 작업이 먼저 도착했을 경우 이후 도착한 짧은 작업들이 길게 대기하는 Convoy effect가 발생하여 평균 대기 시간이 늘어날 수 있다.

2. **Non-preemptive SJF(Shortest Job First)**는 평균 wait time과 turnaround time을 최소화할 수 있는 이론적으로 가장 효율적인 알고리즘이지만, 긴 작업이 계속해서 대기하게 되는 starvation 문제가 존재한다.

3. **Preemptive SJF**는 새롭게 도착한 짧은 작업이 현재 작업보다 짧을 경우 선점하여 응답성을 높일 수 있다는 장점이 있지만, context switch가 빈번히 발생하여 오버헤드가 증가할 수 있다.

4. **Non-preemptive Priority**는 높은 우선순위의 작업을 빠르게 처리할 수 있어 중요한 작업에 집중할 수 있는 장점이 있다. 하지만 낮은 우선순위를 가진 작업은 오랫동안 실행되지 못하는 starvation 문제가 발생한다.

5. **Preemptive Priority**는 우선순위 기반의 선점을 통해 시스템 응답성을 향상시키지만, 이 역시 starvation의 위험이 더 크고 문맥 전환 비용이 증가하는 단점이 있다.

6. **Round Robin**은 모든 프로세스에 균등한 실행 기회를 제공하며 starvation을 방지하고 응답 시간을 줄일 수 있다. 그러나 time quantum이 적절하지 않을 경우 처리량 감소와 문맥 전환 오버헤드가 발생할 수 있다.

7. **Preemptive Priority with Aging**은 priority 기반 선점의 장점에 더해, 대기 시간이 오래된 프로세스의 우선순위를 점진적으로 높이는 방식으로 starvation을 방지하며, 공정성과 응답성 간의 균형을 효과적으로 유지할 수 있다. 실제 운영체제에서 널리 채택되는 전략이기도 하다.

이처럼 각 알고리즘은 실행 환경에 따라 성능과 공정성에서 강점과 약점이 존재하므로, 실제 시스템에서는 요구사항에 맞는 알고리즘 선택 또는 혼합적 전략이 필요하다. 본 시뮬레이터는 이러한 알고리즘들의 동작 원리와 성능 차이를 시각적으로 확인하고 비교할 수 있는 유용한 도구로 기능한다.

## III. 결론

본 보고서는 다양한 CPU 스케줄링 알고리즘을 비교 분석하고 이를 직접 구현함으로써, 운영체제의 핵심 기능 중 하나인 스케줄러의 작동 원리를 심층적으로 이해하는 것을 목표로 하였다. 이를 위해 단일 프로세서 기반 시뮬레이터를 구현하고, 프로세스의

도착 시간(arrival time), 실행 시간(burst time), 우선순위(priority), 그리고 I/O 작업과 같은 요소들을 변수로 하여 실제와 유사한 실행 환경을 구성하였다.

시뮬레이터는 FCFS, SJF, Priority 기반의 non-preemptive 및 preemptive 알고리즘, Round Robin, Priority with Aging, 그리고 Multi-Level Queue까지 총 여덟 가지 알고리즘을 지원하도록 설계되었다. 각 알고리즘은 실제 scheduling 환경에서 발생할 수 있는 다양한 상황(I/O 요청, 프로세스 도착, 선점 조건 등)에 대응할 수 있도록 설계되었고, 시간 단위 시뮬레이션을 통해 각 프로세스의 wait time, turnaround time을 정밀하게 계산하였다.

특히 starvation 문제를 해결하기 위한 aging 기법의 도입, 우선순위 기반의 Multi-Level Queue 구조 설계, 그리고 알고리즘 간 공통 동작을 처리하는 모듈화된 분기 구조는 실용성과 확장성을 동시에 고려한 설계적 성과라 할 수 있다. 각 알고리즘은 조건 충족 시 context switch를 수행하고, 프로세스 상태 전이를 세밀하게 관리하며, 실제 운영체제의 스케줄러와 유사한 흐름을 모사하였다. 이를 통해 다양한 스케줄링 전략의 장단점과 그에 따른 성능 차이를 실험적으로 비교할 수 있었다.

시뮬레이터는 알고리즘 선택창에서 '옵션 9 (Do all and compare)'을 제공하여 동일한 프로세스 배열에 대해 7가지 알고리즘을 자동 실행하고, 각 알고리즘의 평균 turnaround time과 wait time을 비교 출력한다. 이 과정을 통해 시뮬레이션 환경이 알고리즘의 효율성에 큰 영향을 미치며, 단순히 효율성만을 기준으로 알고리즘을 평가하는 것이 아니라, 시스템의 목적(들어 공정성, 응답성, 자원 활용률)에 따라 적절한 알고리즘을 선택해야 한다는 점을 체감할 수 있었다

이번 프로젝트는 크기가 큰 시스템을 설계하고 직접 구현해본 첫 경험이었으며, 많은 도전과 동시에 큰 보람을 느낄 수 있었다. 다양한 스케줄링 알고리즘의 동작 원리를 직접 구현해보고, 이들의 장단점을 실험을 통해 비교해봄으로써, 단순한 이론적 이해를 넘어 실용적인 시스템 관점에서의 판단 능력을 기를 수 있었다. 더 나아가, 프로세스 스케줄링이 단순한 실행 순서 제어를 넘어 전체 시스템의 자원 효율성과 사용자 경험을 결정짓는 핵심 요소임을 깊이 있게 체감할 수 있는 계기가 되었다.

그러나 본 시뮬레이터가 구현한 SJF 알고리즘은, io와 상관없이 남은 CPU 작업 시간인 remain_time을 기준으로 하였기 때문에, io 작업 요청을 고려하면 진정한 의미의 SJF 알고리즘이라 보기 어렵다. SJF 알고리즘 실행 시 유저로부터 CPU burst time, remain time, tentativie CPU time(작업 시작부터 첫 io 작업 요청까지의 연속적인 CPU 작업 시간) 중 판단 기준을 선택 받아 스케줄링하는 방식으로 구현했다면 더 좋았을 것 같다. 다음에 기회가 된다면 SJF 모듈을 이와 같이 수정하고 각 판단 기준에 따른 스케줄링 효율 변화를 비교해보고 싶다. 또한 multi-level queue에서 process들의 후천적인 레벨 이동이

가능한 multi-level feedback queue 알고리즘도 구현해보고 싶다.