텀 프로젝트 보고서

CPU SCHEDULING SIMULATOR

세짐 스득베코바 | 2023320170 | 컴퓨터학과

CPU 스케줄링 시뮬레이터 프로젝트 보고서

1. 서론

모든 현대 운영체제는 CPU 스케줄러를 통해 여러 프로세스에 CPU 사용 시간을 배분한다. CPU 스케줄링이란 운영체제가 프로세스들에게 공정하고 효율적으로 CPU 자원을 할당하는 것을 말하며, 이는 시스템 성능과도 직결되는 중요한 기능이다. 일반적으로 프로세스는 CPU 버스트(CPU 를 사용하는 계산 작업)와 입출력(I/O) 버스트(I/O 를 기다리는 대기 작업)를 번갈아 수행하며, CPU 스케줄러는 이러한 CPU 사용 구간마다 어떤 프로세스에 CPU 를 줄지 결정한다. 효율적인 스케줄링을 통해 CPU 이용률을 높이고 프로세스 응답 시간을 최소화하는 것이 운영체제의 목표이다.

본 보고서에서는 다양한 CPU 스케줄링 알고리즘을 지원하는 CPU 스케줄링 시뮬레이터의 구현 과정을 문서화한다. 구현된 시뮬레이터는 선입선출 방식의 FCFS(First-Come, First-Served), 최단 작업 우선 방식의 SJF (Shortest Job First) - 비선점 및 선점(SRTF, Shortest Remaining Time First), 고정 우선순위 기반의 Priority 스케줄링 - 비선점 및 선점, 그리고 시분할 방식의 Round Robin 알고리즘을 모두 포함한다. 각 알고리즘에 대해 프로세스별 대기 시간과 반환 시간을 계산하고 평균 대기 시간 및 평균 반환 시간을 산출하며, 간트 차트 형태로 실행 과정을 출력한다. 본 시뮬레이터를 통해 다양한 스케줄링 기법의 동작과 성능 특성을 비교할 수 있다.

2. 시뮬레이터 시스템 구성도 및 소스코드 실행 결과

2.1 다른 CPU 스케줄링 시뮬레이터 소개

교육 및 연구를 목적으로 여러 가지 CPU 스케줄링 시뮬레이터가 개발되어 활용되고 있다. 범용 운영체제 시뮬레이터로서 OSP, SOsim, RCOS.java 등이 대표적이며, 단일 목적의 스케줄링 학습 도구로 BACI, MOSS, Alg_OS, OSM, Robbins, CPUSim 등이 보고된 바. 이들 시뮬레이터는 시각화를 통해 알고리즘의 동작을 보여주거나, 학습자가 다양한 시나리오를 실험해 볼 수 있도록 설계되었다. 본 프로젝트에서 구현한 시뮬레이터 역시 교육용으로, C 언어 기반 콘솔 프로그램 형태로 동작하며 여러 알고리즘을 동일한 조건에서 실험하여 결과를 비교할 수 있도록 설계되었다.

2.2 시뮬레이터 시스템 구성도

시뮬레이터의 전체 구조를 그림으로 나타내면 다음과 같다 (그림 1 참조). 프로세스 생성 모듈에서 미리 정의된 프로세스들의 도착 시간, CPU 사용량, I/O 발생

시점과 우선순위 등의 정보를 설정하고 준비시킨다. 그런 다음 메인 제어 루틴에서 선택한 스케줄링 알고리즘 모듈을 호출하여 준비된 프로세스들을 시뮬레이션한다. 각 스케줄러 모듈은 준비 큐(Ready Queue)와 대기 큐(Waiting Queue) 등을 사용하여 해당 알고리즘에 따라 프로세스 실행 순서를 결정하고, 완료 시각까지의 각종 통계를 산출한다. 시뮬레이터는 최종적으로 알고리즘별로 프로세스들의 완료 결과와 간트 차트 등을 출력하며, 실행이 끝나면 다음 알고리즘의 시뮬레이션을 이어서 수행한다.
그림 1. 시스템 구성도 (시뮬레이터 전체 구조) 삽입 위치

2.3 주요 모듈 설명

시뮬레이터는 몇 개의 C 소스 코드 모듈로 구성되어 있으며, 각 모듈의 역할은 다음과 같다:

- 프로세스 생성 및 출력 모듈: Create_Processes 함수는 미리 정의된 다섯 개의 프로세스 정보를 배열에 채워 넣는다. 예를 들어, PID, 도착 시간(AT), CPU 버스트 시간, I/O 발생 시점 및 소요 시간, 우선순위 등을 하드코딩된 값으로 설정하며, I/O 발생 시점이 -1 이면 해당 프로세스는 I/O 작업이 없는 것으로 표기한다. 이렇게 초기화된 프로세스 목록은 곧바로 시뮬레이션에 사용되거나, -manual 옵션으로 실행 시 표준 입력을 통해 사용자로부터 직접 값을 입력받을 수도 있다. 이후 Print_Processes 함수가 프로세스들의 주요 정보를 표 형식으로 출력하여 시뮬레이션에 앞서 확인할 수 있게 한다.
- 큐(Queue) 모듈: 준비 큐와 I/O 대기 큐를 구현하는 모듈로, queue.c 에서 선형 링크드 리스트 기반의 Queue 자료구조와 enqueue, dequeue 등의 연산을 제공한다. 스케줄러는 준비 큐를 통해 ready 상태의 프로세스들을 관리하며, I/O 작업 중인 프로세스들은 대기 큐에 보관된다. 각 스케줄링 알고리즘 구현에서 이 큐 연산들을 사용하여 프로세스를 큐에 삽입하거나 제거하고, I/O 완료 시점에 다시 준비 큐로 복귀시키는 작업 등을 수행한다.
- 스케줄링 알고리즘 모듈: 각 CPU 스케줄링 알고리즘별로 별도의 함수로 구현되어 있다. 모든 알고리즘 모듈은 공통적으로 현재 시간(current_time)과 완료된 프로세스 수를 추적하면서 동작하며, 프로세스들의 waiting_time(대기 시간), turnaround_time(반환 시간) 등을 최종 계산한다. 주요 알고리즘별 동작은 다음과 같다:
    - FCFS + I/O: 비선점형 선입선출 스케줄링으로, 도착 순서대로 프로세스를 실행한다. I/O 가 있는 프로세스는 CPU 실행 중 지정된 시점에 I/O 요청 이벤트를 발생시키며, 이때 해당 프로세스를 대기 큐로 보내 I/O 가 완료될 때까지 기다리게 한다. I/O 완료 후에는 다시 준비 큐로 돌아와 남은 실행을 이어가며, 모든 프로세스가 완료될 때까지 반복한다. FCFS 는 구현이 가장 단순하지만, CPU 버스트가 긴 프로세스 앞에 짧은 프로세스가 줄 서게 되면 Convoy 효과가 발생하여 평균 대기 시간이 늘어날 수 있다.

- SJF (비선점): 최단 작업 우선 스케줄링의 비선점 버전으로, 준비 큐에 대기 중인 프로세스들 중 남은 CPU 처리 시간이 가장 짧은 것을 선택하여 CPU 를 할당한다. 한 프로세스가 완료될 때까지 CPU 를 선점하지 않고 수행하며, 완료 시점에 다시 가장 짧은 작업을 선택하는 식으로 스케줄링이 진행된다. SJF 알고리즘은 이론적으로 평균 대기 시간을 최소화하는 최적 알고리즘으로 잘 알려져 있다. 그러나 프로세스들의 실제 CPU 실행 시간을 사전에 정확히 알 수 없다는 현실적 한계와, 긴 작업이 뒤로 밀릴 경우 무한 대기 상태에 빠질 수 있는 단점이 존재한다.
- SJF (선점): Shortest Remaining Time First (SRTF)라고도 불리는 선점형 최단 작업 우선 알고리즘이다. 현재 실행 중인 프로세스보다 남은 시간이 더 짧은 프로세스가 새로 도착하면 즉시 CPU 를 선점하여 그 프로세스에게 CPU 를 넘긴다. 이러한 방식으로 항상 남은 작업 시간이 최소인 프로세스가 실행되도록 함으로써, 평균 대기 시간과 응답 시간을 더욱 감소시킨다. 단, 잦은 선점으로 인한 문맥 교환 비용이 증가하며, SJF 와 마찬가지로 실행 시간이 긴 프로세스가 지속해서 뒤로 밀리면 기아(starvation) 현상이 발생할 수 있다.
- Priority (비선점): 각 프로세스에 할당된 우선순위(priority) 값을 사용하여, 준비 큐에서 가장 높은 우선순위(숫자가 가장 작은 값)를 가진 프로세스부터 CPU 를 할당하는 방식이다. 비선점형 우선순위 스케줄링에서는 일단 CPU 를 얻은 프로세스가 완료될 때까지 실행을 이어가며, 완료 후에야 다시 최고 우선순위 프로세스를 선택한다. 우선순위는 일반적으로 긴급한 작업이나 I/O 빈도가 높은 작업에 높게 부여되므로, 그런 작업들을 빠르게 처리하여 전체 처리율(Throughput)을 높이는 효과가 있다. 그러나 우선순위가 낮은 작업은 계속 뒤로 밀려 아예 실행 기회를 얻지 못하는 문제를 야기할 수 있어, 이를 완화하기 위해 에이징(Aging) 기법 등 우선순위 조정 기법을 함께 사용하기도 한다.
- Priority (선점): 선점형 우선순위 스케줄링은 실행 중에도 더 높은 우선순위를 가진 프로세스가 도착하면 현재 프로세스를 즉시 중단시키고 CPU 를 재할당한다. 본 시뮬레이터의 구현에서는 프로세스들의 우선순위 값이 낮을수록 높은 우선순위를 의미하며, 실행 중 항상 준비 큐를 순회하면서 가장 작은 priority 값을 가진 프로세스를 선택한다. 선점형으로 동작함에 따라 긴급한 작업을 가장 빠르게 처리하지만, 이 역시 낮은 우선순위의 작업이 장시간 대기하는 부작용을 가질 수 있다. 비선점/선점 우선순위 방식 모두 우선순위만을 고려하므로 SJF 와 유사하게 CPU-bound 프로세스가 I/O-bound 프로세스에 의해 뒤로 밀리는 경향이 있다.
- Round Robin (RR): 라운드 로빈 스케줄링은 시분할 시스템을 위한 알고리즘으로, 프로세스마다 정해진 시간량만큼(CPU 타임퀀텀) CPU 를

사용하게 한 뒤, 완료되지 않은 경우 다시 준비 큐의 가장 뒤로 보내 다음 프로세스에 CPU 를 넘긴다. 본 구현에서는 타임퀀텀을 3 으로 설정하였다. Round Robin 은 선입선출 기반이지만 시간 분할을 통해 모든 프로세스가 골고루 CPU 를 할당받도록 함으로써 응답 시간을 개선하고 특정 프로세스의 독점을 방지한다. I/O 가 있는 프로세스의 경우, 실행 도중 I/O 요청이 발생하면 잔여 타임퀀텀을 다 쓰지 않았더라도 즉시 CPU 를 반환하고 대기 큐로 이동한다. RR 은 공정성(fairness) 측면에서는 우수하지만, 문맥 교환이 빈번하고 평균 대기 시간이 FCFS 보다 커질 수 있다는 단점이 있다 (타임퀀텀이 너무 작을 경우 특히 overhead 가 증가). 본 시뮬레이터에서는 타임퀀텀이 3 으로 비교적 작게 설정되어 있어, I/O 가 없는 CPU 집중 프로세스들은 자주 선취당해 대기 큐를 순환하므로 대기 시간이 늘어나는 모습을 볼 수 있다.

2.4 알고리즘별 실행 결과 예시

각 스케줄링 알고리즘을 적용한 시뮬레이션 실행 결과 예시는 그림 2~7 과 같다. 본 실험에서는 5 개의 프로세스를 대상으로 앞서 정의된 도착 시간, CPU 버스트, I/O 요구 등을 동일하게 적용하였다. 시뮬레이터는 알고리즘별로 각 프로세스의 완료 시각(CT, Completion Time), 대기 시간(WT, Waiting Time), 반환 시간(TT, Turnaround Time)을 출력하고, 시뮬레이션이 끝난 뒤 해당 알고리즘의 평균 대기 시간과 평균 반환 시간을 계산하여 보여준다. 또한 간트 차트(Gantt Chart)를 통해 시간 축상에서 어느 시점에 어떤 프로세스가 실행되고 있었는지 시각적으로 표시한다.

- FCFS 결과: 그림 2 는 FCFS 알고리즘으로 스케줄링한 실행 예시를 보여준다. 도착 순서대로 프로세스가 실행되어 P1 → P2 → … → P5 순으로 완료된 것을 간트 차트에서 확인할 수 있다. I/O 가 있는 프로세스(P2, P3, P5)는 실행 도중 I/O 대기 상태로 전환되면서 CPU 를 내어주고, 그동안 다른 프로세스들이 실행된다. 예를 들어 P2 는 CPU 를 2 tick 사용한 시점에 I/O 요청을 하여 대기 상태로 빠지고, I/O 가 완료된 후에야 남은 실행을 이어간다. 이러한 과정 때문에 P2, P3, P5 의 반환 시간이 CPU 실행 시간 단순 합보다 길어졌음을 결과에서 알 수 있다. FCFS 의 경우 P4 처럼 CPU 버스트가 짧지만 뒤에 도착한 프로세스가 앞의 긴 작업(P1)이 끝날 때까지 기다리는 상황이 발생하며, 이는 P4 의 대기 시간이 19 로 비교적 크게 측정된 원인이다.

```
PID 1: has_io = 0 (io_start = -1, io_burst = 0)
PID 2: has_io = 1 (io_start = 2, io_burst = 3)
PID 3: has_io = 1 (io_start = 3, io_burst = 2)
PID 4: has_io = 0 (io_start = -1, io_burst = 0)
PID 5: has_io = 1 (io_start = 1, io_burst = 2)
Create_Processes() from process.c 실행됨!!
PID     AT      CPU     IO@     IOBurst Prio
1       0       10      -1      0       3
2       2       7       2       3       2
3       3       8       3       2       5
4       4       6       -1      0       1
5       5       5       1       2       4
```

```
=== FCFS + I/O 결과 ===
[시간 0] 루프 시작
  → 프로세스 1 도착
  → 프로세스 1 실행 중
[시간 1] 루프 시작
  → 프로세스 1 실행 중
[시간 2] 루프 시작
  → 프로세스 2 도착
  → 프로세스 1 실행 중
[시간 3] 루프 시작
  → 프로세스 3 도착
  → 프로세스 2 실행 중
[시간 4] 루프 시작
  → 프로세스 4 도착
  → 프로세스 1 실행 중
[시간 5] 루프 시작
  → 프로세스 5 도착
  → 프로세스 3 실행 중
[시간 6] 루프 시작
  → 프로세스 2 실행 중
[시간 7] 루프 시작
  → 프로세스 4 실행 중
[시간 8] 루프 시작
  → 프로세스 1 실행 중
[시간 9] 루프 시작
  → 프로세스 5 실행 중
[시간 10] 루프 시작
  → 프로세스 3 실행 중
[시간 11] 루프 시작
  → 프로세스 2 실행 중
  → I/O 요청 발생: 프로세스 2
[시간 12] 루프 시작
  → 프로세스 4 실행 중
[시간 13] 루프 시작
  → 프로세스 1 실행 중
[시간 14] 루프 시작
  → 프로세스 5 실행 중
  → I/O 요청 발생: 프로세스 5
[시간 15] 루프 시작
  → 프로세스 3 실행 중
[시간 16] 루프 시작
  → 프로세스 4 실행 중
[시간 17] 루프 시작
  → 프로세스 1 실행 중
[시간 18] 루프 시작
  → 프로세스 2 실행 중
[시간 19] 루프 시작
  → 프로세스 3 실행 중
  → I/O 요청 발생: 프로세스 3
[시간 20] 루프 시작
  → 프로세스 5 실행 중
[시간 21] 루프 시작
  → 프로세스 4 실행 중
[시간 22] 루프 시작
  → 프로세스 1 실행 중
[시간 23] 루프 시작
  → 프로세스 2 실행 중
[시간 24] 루프 시작
  → 프로세스 5 실행 중
[시간 25] 루프 시작
  → 프로세스 3 실행 중
[시간 26] 루프 시작
  → 프로세스 4 실행 중
[시간 27] 루프 시작
  → 프로세스 1 실행 중
[시간 28] 루프 시작
  → 프로세스 2 실행 중
[시간 29] 루프 시작
  → 프로세스 5 실행 중
```

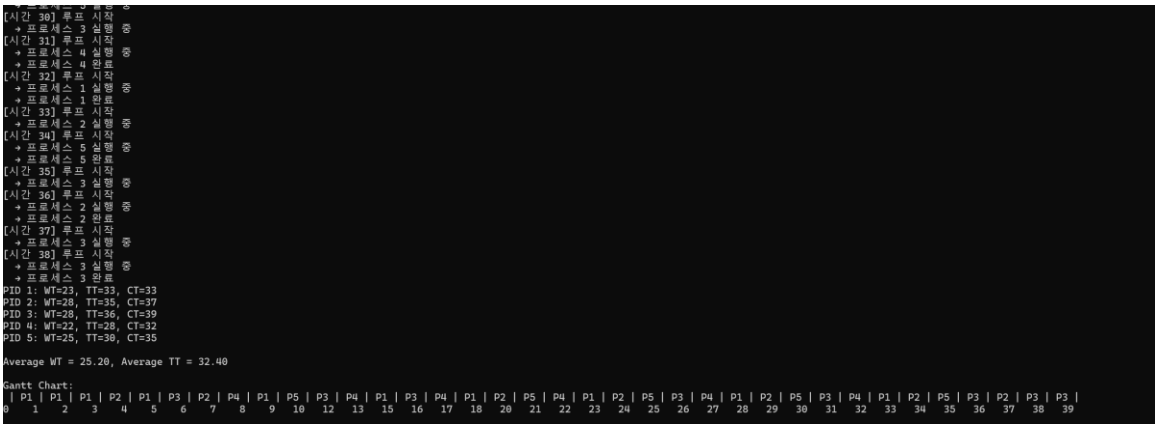

그림 2. FCFS 알고리즘 실행 결과 출력 예시 (콘솔 창) 삽입 위치

- Non-preemptive SJF (비선점) 결과: 그림 3 의 SJF 결과에서는 프로세스들의 완료 순서가 FCFS 와 다르게 짧은 작업이 먼저 끝나는 것을 볼 수 있다. 초기의 P1 대신 더 짧은 P5, P4 등이 먼저 CPU 를 받아 수행되며, 그 결과 평균 대기 시간이 크게 감소하였다. 실제로 본 예시에서 SJF 알고리즘의 평균 대기 시간은 약 12.0ms 로, 동일한 작업을 FCFS 로 처리한 경우의 22.2ms 보다 현저히 작다. 이는 SJF 가 평균 대기 시간을 최소화하는 특성을 잘 보여준다. 다만 SJF 에서는 P3 처럼 CPU 실행 시간이 긴 프로세스가 계속 뒤로 밀려 반환 시간이 가장 늦어지는 현상이 관찰된다.

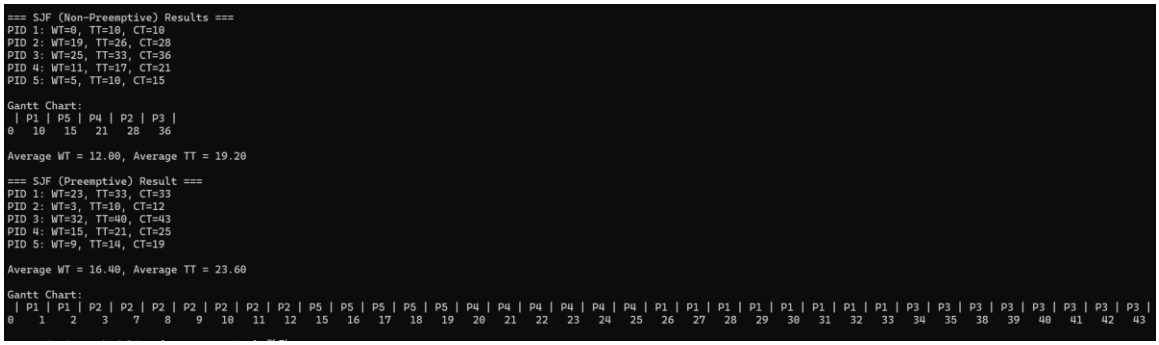

그림 3. Non-preemptive SJF 알고리즘 실행 결과 예시 삽입 위치

- Preemptive SJF (선점) 결과: 그림 4 는 선점형 SJF 알고리즘의 실행 예를 보여준다. 선점형인 만큼, 예를 들어 P1 이 실행을 시작한 직후 P2 가 도착하면 바로 P1 을 선점하고 P2 가 CPU 를 차지한다. 이러한 선점으로 SJF 의 단기 작업 우선 전략을 실현한 결과, 평균 대기 시간은 약 11.4ms 로 비선점 SJF 보다도 조금 더 개선되었다. 하지만 비선점 SJF 와 비교해 큰 차이는 없으며, 프로세스들의 완료 순서 또한 거의 비슷하게 나타난다. 이는 본 예시에서는 선점으로 인한 이득이 크지 않았음을 의미한다. 선점형 SJF 의 간트 차트를 보면, 새로운 단기 프로세스가 도착할 때마다 문맥 교환이 일어나므로 실행 흐름이 자주 분할되어

표시된다. CPU burst 가 긴 P3 는 여전히 가장 마지막에 완료되어 기아 현상이 일부 드러난다.

그림 4. Preemptive SJF 알고리즘 실행 결과 예시 삽입 위치

- Priority (비선점) 결과: 그림 5 는 비선점 우선순위 스케줄링 결과이다. 초기에는 P1 이 실행을 시작하였지만, 모든 프로세스가 도착한 시점부터는 가장 높은 우선순위(숫자 1 인 P4)가 먼저 실행되었고 이후 우선순위 2 인 P2, 우선순위 3 인 P1, 우선순위 4 인 P5, 마지막으로 5 인 P3 순서로 처리되었다. 이 순서는 각 프로세스의 우선순위 값 순과 일치한다. 평균 대기 시간은 약 12.6ms 로 SJF 와 비슷한 수준이며, 우선순위에 따라 단기 작업과 I/O bound 작업(P4, P2 등)이 먼저 수행되면서 전체적인 응답 속도가 향상되었다. 그러나 우선순위 5 였던 P3 는 가장 늦게까지 기다려야 했고, 실제로 P3 의 대기 시간이 25ms 로 가장 크게 나타났다.

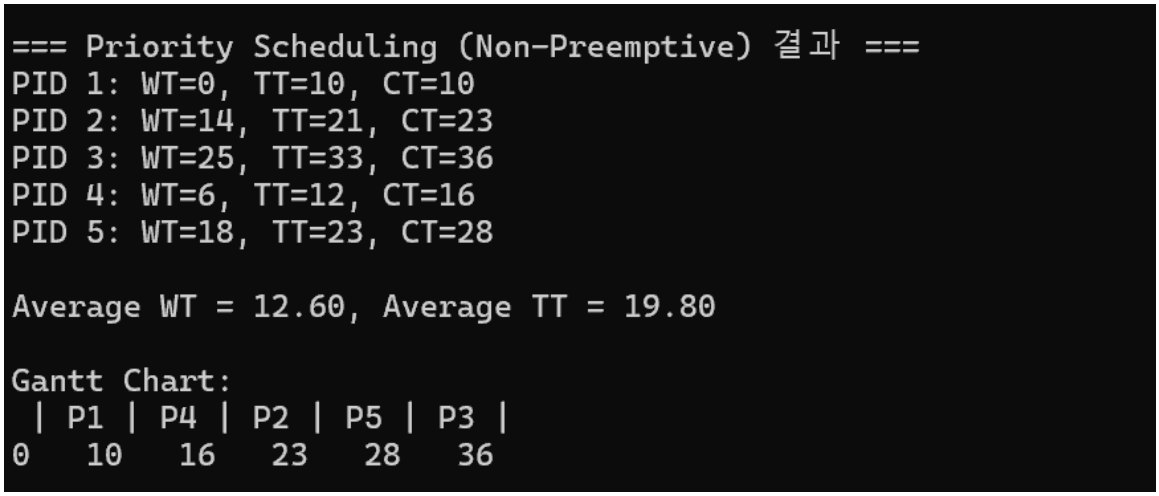

그림 5. 우선순위 비선점 알고리즘 실행 결과 예시 삽입 위치

- Priority (선점) 결과: 그림 6 은 선점형 우선순위 알고리즘 결과로, 동작 중 더 높은 우선순위 프로세스가 도착하면 선점이 발생하는 케이스를 보여준다. 예컨대, 초기 P1 이 실행되고 있던 도중 우선순위 2 의 P2 가 도착하자 P1 이 선점당하고 P2 가 실행되었다. 이후 우선순위 1 의 P4 가 도착하자 다시 P2 를 선점하고 P4 가 실행되는 식으로 진행된다. 모든 프로세스가 도착한 후에는 비선점과 마찬가지로 우선순위 순으로 완료되었다. 평균 대기 시간(약 12.4ms)과 완료 순서는 비선점 우선순위의 결과와 유사하다. 선점으로 인한 추가 이점보다는, P1 이 P2/P4 에 의해 여러 번 선점당해 실행이 분할되는 간트 차트 양상이 두드러진다. 우선순위가

가장 낮았던 P3 는 역시 마지막까지 실행 기회를 얻지 못해 가장 큰 대기 시간을 기록하였다.

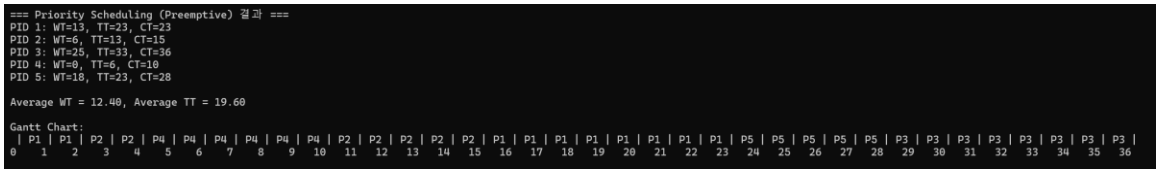

그림 6. 우선순위 선점 알고리즘 실행 결과 예시 삽입 위치

- Round Robin 결과: 그림 7 는 라운드 로빈(RR) 알고리즘 실행 예시이다. RR 에서는 타임퀀텀 3 씩을 각 프로세스에 돌아가며 할당하므로, 간트 차트에서 프로세스들이 번갈아 출현하는 패턴을 볼 수 있다. 예를 들어 초기에 P1 이 3tick 을 사용한 후 P1 이 완료되지 않았으므로 준비 큐 뒤로 이동하고, 다음으로 도착한 P2, P3 등이 차례로 CPU 를 얻는 식이다. I/O 가 있는 프로세스의 경우 RR 사이클 중간에 I/O 요청으로 CPU 를 반납하기도 한다 (예: P2, P5 의 간트 차트 구간이 끊기는 부분). 이런 시분할로 인해 모든 프로세스의 응답 시작시간이 빨라지는 장점이 있으나, 결과에서 보이듯이 평균 대기 시간은 22.4ms 로 FCFS 와 거의 비슷한 수준으로 높아졌다. 특히 짧은 프로세스들도 긴 타임퀀텀 주기의 끝까지 대기해야 하므로 SJF 에 비해 비효율적인 면이 있다. 결국 RR 은 응답 속도를 중시하는 대화식 시스템에 적합하며, 본 예시처럼 작업량이 확실한 배치 작업에서는 평균 지표 면에서 다소 불리한 결과를 보일 수 있다.

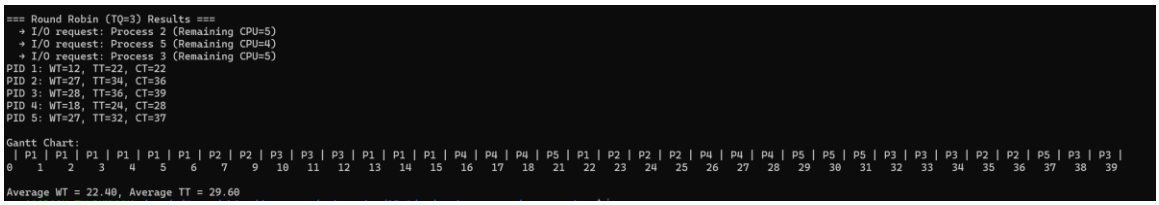

그림 7. Round Robin (타임퀀텀=3) 알고리즘 실행 결과 예시 삽입 위치

2.5 알고리즘 성능 비교 분석

위 실험 결과를 종합하여 각 스케줄링 알고리즘의 평균 대기 시간(AWT)과 평균 반환 시간(ATT)을 비교하면 다음과 같다 (표 1 참조). SJF 및 우선순위 기반 알고리즘들은 전반적으로 FCFS 나 RR 에 비해 평균 대기 시간과 반환 시간을 크게 단축시키는 것을 확인할 수 있다. 특히 SJF 알고리즘의 평균 대기 시간은 약 11~12ms 수준으로, FCFS(약 22.2ms)의 절반 수준에 불과하다. 이는 짧은 작업을 먼저 처리함으로써 긴 작업들이 기다리는 동안 다른 짧은 작업들이 완료되어 전체 대기시간을 낮추기 때문이다. 선점형 SJF(SRTF)는 비선점 SJF 보다 약간 더 낮은 대기 시간을 보였으나, 본 데이터셋에서는 그 차이가 미미하였다. 우선순위 스케줄링의 결과도 SJF 와 유사하게 나타났는데, 이는 높은 우선순위(P4, P2)를 가진 프로세스들이 상대적으로 CPU 사용량이 적거나 I/O 작업을 포함한 프로세스들이었기 때문에 평균 대기 시간 개선에

기여했기 때문으로 분석된다. 선점 여부는 우선순위 알고리즘의 평균 지표에는 큰 영향을 주지 않았으나, 이는 데이터셋의 특성상 우선순위가 가장 높은 P4 가 초기에 도착했기 때문에 선점 상황이 초기 외에는 많지 않았기 때문으로 보인다.

반면 FCFS 와 Round Robin 은 가장 높은 평균 대기/반환 시간을 나타냈다. 두 알고리즘 모두 프로세스의 특성을 고려하지 않고 공평하게 CPU 를 배분한다는 공통점이 있는데, 이러한 비선별적 방식은 짧은 작업들의 완료를 지연시켜 평균적인 지표를 악화시킨다. FCFS 에서는 P1 같은 긴 작업이 먼저 시작되어 뒤의 모든 프로세스들을 지연시키는 경우가 문제며, RR 에서는 빈번한 문맥 교환과 대기 큐 순환으로 인해 전체 처리 시간이 늘어나는 경향이 있다. 다만 RR 은 응답 시간 개선 측면에서 FCFS 보다 유리하고, 특정 프로세스가 지나치게 오래 대기하지 않는 공정성을 달성한다는 장점을 가진다. 이러한 트레이드오프(trade-off)는 스케줄링 알고리즘 선택 시 시스템의 목표에 따라 고려되어야 할 요소이다.

표 1. 알고리즘별 평균 대기 시간 및 반환 시간 비교 (본 시뮬레이터 실험 결과) 삽입 위치:

알고리즘	평균 대기 시간 (ms)	평균 반환 시간 (ms)
FCFS (선입선출)	22.20	29.40
SJF (비선점)	12.00	19.20
SJF (선점, SRTF)	11.40	18.60
Priority (비선점)	12.60	19.80
Priority (선점)	12.40	19.60
Round Robin (TimeQ=3)	22.40	29.60

위 표에서 확인할 수 있듯이, 최단 작업 우선(SJF) 계열과 우선순위 알고리즘들은 평균 응답 성능에서 우수하였으며, FCFS 와 Round Robin 은 상대적으로 낮은 성능을 보였다. 특히 SJF 및 선점 SJF 는 이론적으로도 평균 대기 시간을 최소화하는 알고리즘으로 알려져 있으며, 본 실험 결과도 이를 뒷받침한다. 다만 SJF 계열과 우선순위 알고리즘은 앞서 언급한 바와 같이 기아 현상의 위험이 있고, 실제 시스템에서는 프로세스의 실행 시간을 예측하기 어렵다는 문제가 따른다. 반대로 Round Robin 은 가장 높은 평균 시간을 보였지만 선호도 없는 공정한 분배를 통해 인터랙티브 시스템에서 사용되는 등 용도가 다르다. 종합하면, 스케줄링 알고리즘의 성능 평가는 단순 평균 지표뿐만 아니라 응답성, 공정성, 시스템 목적에 따른 적합성 등을 모두 고려해야 한다.

3. 결론

본 프로젝트를 통해 CPU 스케줄링 시뮬레이터를 설계 및 구현하고, 여러 스케줄링 알고리즘의 동작과 성능 특성을 비교해보았다. 시뮬레이터는 FCFS, SJF, Priority,

Round Robin 등의 알고리즘을 하나의 일관된 프레임워크에서 실행하여, 프로세스별 대기 시간, 반환 시간, 완료 순서, 간트 차트 등을 출력해준다. 이를 통해 이론으로 학습한 스케줄링 알고리즘들이 실제 프로세스 집합에 어떻게 적용되는지 구체적으로 확인할 수 있었으며, 알고리즘 간의 장단점을 정량적인 지표로도 파악할 수 있었다.

프로젝트를 수행한 소감으로는, 처음에는 각 알고리즘별로 다른 로직을 구현해야 해서 복잡하게 느껴졌지만, 공통된 자료구조(예: 준비 큐, 대기 큐)와 흐름(현재 시각을 진행하며 완료 프로세스 체크 등)을 활용하여 모듈화하니 효율적으로 개발할 수 있었다. 또한 시뮬레이션을 통해 얻은 결과가 운영체제 이론에서 배운 예상과 대체로 부합하여, 구현의 정확성 검증과 함께 스케줄링 이론에 대한 이해도도 높아졌다. 한편으로는 실제 운영체제의 스케줄러는 더 많은 복잡한 요소들(예: 멀티코어 환경, 동적 우선순위 조정, 프로세스의 입출력 패턴 등)을 고려해야 함을 인식하게 되어, 본 시뮬레이터의 한계를 느끼기도 했다.

향후 발전 방향으로는, 현재 콘솔 기반으로 정적인 시나리오만을 다루는 본 프로그램에 사용자 인터페이스를 추가하여 다양한 입력을 실시간으로 받아볼 수 있게 개선할 수 있다. 예를 들어 GUI 를 통해 프로세스 수나 시간값들을 조정하고 시뮬레이션을 시각화하면 교육적 효과가 더 클 것으로 기대된다. 또한 멀티레벨 큐 스케줄링, 실시간 스케줄링(Rate Monotonic, EDF) 등 보다 폭넓은 알고리즘을 지원하도록 확장하거나, 프로세스 수를 늘리고 확률적으로 I/O 나 CPU burst 를 생성하는 기능을 추가하여 무작위 작업 부하(random workload)에 대한 실험도 가능하게 할 수 있다. 이러한 개선을 통해 본 시뮬레이터가 운영체제 스케줄링의 학습에 유용한 도구로 발전할 수 있을 것이다.

4. 부록: 주요 소스코드 요약

아래는 시뮬레이터 구현을 위한 핵심 소스코드 일부를 발췌하여 제시한 것이다 (C 언어). 전체 소스코드는 프로세스 구조체 및 큐 자료구조 정의, 프로세스 생성 함수, 그리고 6 가지 스케줄러 함수들로 구성된다. 각 스케줄러 함수는 해당 알고리즘의 동작을 구현하며, 공통적으로 현재 시간 current_time 을 증가시키면서 이벤트를 처리한다.

1) FCFS + I/O 스케줄러 핵심 코드: FCFS 에서는 I/O 대기 처리를 위해 별도의 waiting_queue 를 사용한다. 매 시간 단위마다 현재 시간에 도착한 프로세스를 ready_queue 에 넣고, I/O 대기 중인 프로세스들의 io_remaining_time 을 감소시켜 완료된 경우 다시 준비 큐로 이동시킨다. 그 후, 준비 큐가 비어있지 않으면 큐 가장 앞의 프로세스를 실행한다. I/O 요청 시점(io_start_time)에 도달하면 CPU 실행을 중단하고 해당 프로세스를 대기 큐로 보내며, 그렇지 않으면 CPU 버스트를 1 만큼 소모한 뒤 남은 시간을 갱신한다. 이 과정은 모든 프로세스가 완료될 때까지 반복된다. 주요 구현은 다음과 같다:

```
1. void fcfs_with_io(Process processes[], int count) {
2.     Queue ready_queue, waiting_queue;
3.     init_queue(&ready_queue);
4.     init_queue(&waiting_queue);
5.     /* 프로세스들을 도착 시간 순으로 정렬 */
6.     /* ... (코드 생략: Bubble sort by arrival_time) ... */
7.
8.     int current_time = 0, completed = 0;
9.     int inserted[MAX_PROCESSES] = {0};
10.    /* gantt 와 timeline 배열 선언 */
11.    /* ... */
12.
13.    while (completed < count) {
14.        /* 현재 시간에 도착한 프로세스는 준비 큐에 삽입 */
15.        for (int i = 0; i < count; i++) {
16.            if (!inserted[i] && processes[i].arrival_time <= current_time) {
17.                enqueue(&ready_queue, &processes[i]);
18.                inserted[i] = 1;
19.                printf("  → 프로세스 %d 도착\n", processes[i].pid);
20.            }
21.        }
22.        /* I/O 대기 중인 프로세스 처리 */
23.        update_waiting_queue(&waiting_queue, &ready_queue);
24.
25.        if (is_empty(&ready_queue)) {
26.            /* 준비 큐가 비어 있으면 시간 진행 */
27.            current_time++;
28.            continue;
29.        }
30.
```

```
        Process *p = dequeue(&ready_queue);
        printf("  → 프로세스 %d 실행 중\n", p->pid);

        /* I/O 요청 발생 시점인지 체크 */
        if (p->has_io && p->io_done == 0
                && p->cpu_burst - p->remaining_time == p->io_start_time) {
            /* I/O 시작: 프로세스 p를 waiting 큐로 이동 */
            printf("  → I/O 요청 발생: 프로세스 %d\n", p->pid);
            p->io_remaining_time = p->io_burst;
            enqueue(&waiting_queue, p);
        } else {
            /* CPU 버스트 1만큼 실행 */
            p->remaining_time--;
            current_time++;
            if (p->remaining_time == 0) {
                /* 프로세스 완료 처리 */
                p->completion_time = current_time;
                p->turnaround_time = p->completion_time - p->arrival_time;
                p->waiting_time = p->turnaround_time - p->cpu_burst;
                completed++;
                printf("  → 프로세스 %d 완료\n", p->pid);
            } else {
                /* 아직 남은 실행 시간 있음 -> 다시 준비 큐에 추가 */
                enqueue(&ready_queue, p);
            }
        }
    }
    /* 결과 출력 (각 프로세스 WT, TT 등 및 평균 계산) */
    /* ... */
}
```

위 코드에서 update_waiting_queue 함수는 대기 큐에 있는 모든 프로세스의 io_remaining_time 를 1 씩 감소시키고, I/O 가 끝난 프로세스는 io_done 플래그를 세운 뒤 준비 큐로 옮기는 역할을 한다. FCFS 의 구현은 비교적 단순하지만, I/O 처리를 위해 ready/waiting 두 개의 큐를 관리하는 점이 특징이다.

2) SJF 및 Priority 스케줄러 핵심 코드: SJF 와 Priority 알고리즘은 공통적으로 *준비 큐에 대기 중인 프로세스 중 조건에 맞는 다음 실행 프로세스를 선택*하는 로직이 핵심이다. 비선점형의 경우 한 번 선택된 프로세스를 완료시까지 실행하고, 선점형의 경우 매 시간 단위마다 선택을 다시 한다는 차이만 있다. 아래는 비선점 SJF 알고리즘의 주요 코드이다 (최단 남은 CPU 시간을 가진 프로세스를 선택):

```
void sjf_non_preemptive(Process processes[], int count) {
    int current_time = 0, completed = 0;
    int done[100] = {0};
    /* ... */

    while (completed < count) {
```

```
        int shortest = 1000000000;
        int selected = -1;
        /* 아직 완료되지 않고 도착한 프로세스 중 남은 CPU burst 가 가장 짧은 프로세스 선택 */
        for (int i = 0; i < count; i++) {
            if (!done[i] && processes[i].arrival_time <= current_time) {
                if (processes[i].cpu_burst < shortest) {
                    shortest = processes[i].cpu_burst;
                    selected = i;
                } else if (processes[i].cpu_burst == shortest) {
                    /* 동률인 경우 도착 시간이 더 이른 프로세스 선택 */
                    if (processes[i].arrival_time < processes[selected].arrival_time)
                        selected = i;
                }
            }
        }

        if (selected == -1) {
            current_time++;
            continue;
        }
        Process *p = &processes[selected];
        /* 선택된 프로세스를 완료시까지 실행 */
        p->waiting_time = current_time - p->arrival_time;
        current_time += p->cpu_burst;
        p->completion_time = current_time;
        p->turnaround_time = p->completion_time - p->arrival_time;
        done[selected] = 1;
        completed++;
        printf("PID %d 완료 (WT=%d, TT=%d)\n", p->pid, p->waiting_time, p-
>turnaround_time);
    }
    /* ... 결과 평균 출력 ... */
}

```

위 코드에서 SJF 비선점은 for 루프로 최단 작업을 찾은 뒤 해당 프로세스를 한꺼번에 실행(current_time 을 CPU burst 만큼 증가)한다는 점이 특징이다. Priority 비선점 구현도 이와 유사하며, 비교 조건만 cpu_burst 대신 priority 를 사용하고 있다. 선점형 SJF 와 Priority 는 ready_queue 에 대기 중인 프로세스를 매 tick 마다 스캔하여 조건에 맞는 프로세스를 선택하는 방식으로 구현하였다. 예를 들어 선점형 Priority 의 핵심 코드는 다음과 같다 (매 시간 highest priority 프로세스 선택):

```
void priority_preemptive(Process processes[], int count) {
    Queue ready_queue;
    init_queue(&ready_queue);
    int current_time = 0, completed = 0;
    int inserted[100] = {0};
    /* ... */

    while (completed < count) {
        /* 현재 시간에 도착한 프로세스들 준비 큐 삽입 */
```

```
        for (int i = 0; i < count; i++) {
            if (!inserted[i] && processes[i].arrival_time <= current_time) {
                enqueue(&ready_queue, &processes[i]);
                inserted[i] = 1;
            }
        }
        /* 준비 큐에서 우선순위가 가장 높은 프로세스 선택 */
        Process *current = NULL;
        Node *iter = ready_queue.front;
        int highest_priority = 1000000000;
        while (iter != NULL) {
            if (iter->process->remaining_time > 0
                    && iter->process->priority < highest_priority) {
                highest_priority = iter->process->priority;
                current = iter->process;
            }
            iter = iter->next;
        }
        if (current == NULL) {
            current_time++;
            continue;
        }
        /* 1 tick 실행 */
        current->remaining_time--;
        current_time++;
        if (current->remaining_time == 0) {
            /* 완료 시 처리 */
            current->completion_time = current_time;
            current->turnaround_time = current->completion_time - current->arrival_time;
            current->waiting_time = current->turnaround_time - current->cpu_burst;
            completed++;
        }
    }
    /* ... 결과 출력 ... */
}

```

선점형 알고리즘에서는 현재 실행 중인 프로세스를 큐에서 제거하지 않고 remaining_time 으로 완료 여부만 판단한다. 매 루프마다 준비 큐를 훑어 최우선 프로세스를 찾기 때문에, 새로운 높은 우선순위 프로세스가 도착하면 다음 루프에서 곧바로 선택될 수 있다. 이 방식은 구현이 다소 비효율적일 수 있으나, 프로세스 수가 적으므로 문제없다. SJF 선점형(sjf_preemptive)도 이와 동일한 구조로, 우선순위 대신 남은 시간(remaining_time)을 비교하여 최단 남은 시간을 가진 프로세스를 current 로 선택하는 점만 다르다.

3) Round Robin 스케줄러 핵심 코드: Round Robin 구현에서 중요한 부분은 타임퀀텀 관리와 I/O 처리이다. 본 구현에서는 time_quantum 을 인자로 받아와 사용하며, 각 프로세스를 디큐하여 최대 time_quantum 만큼 실행하거나 그 전에 종료/I/O 발생 시 루프를 깨고 나온다. 또한 실행 중 I/O 요청을 발견하면 즉시 대기 큐로 옮기고 해당 time slice 를 종료한다. RR 의 주요 구현 부분은 아래와 같다:

```c
void round_robin(Process processes[], int count, int time_quantum) {
    Queue ready_queue, waiting_queue;
    init_queue(&ready_queue);
    init_queue(&waiting_queue);
    int current_time = 0, completed = 0;
    int inserted[100] = {0};
    /* ... */

    while (completed < count) {
        /* 새로 도착한 프로세스 ready_queue 삽입 */
        for (int i = 0; i < count; i++) {
            if (!inserted[i] && processes[i].arrival_time <= current_time) {
                enqueue(&ready_queue, &processes[i]);
                inserted[i] = 1;
            }
        }
        /* I/O 대기 중인 프로세스 한 tick 씩 처리 */
        update_waiting_queue_rr(&waiting_queue, &ready_queue);

        if (is_empty(&ready_queue)) {
            current_time++;
            continue;
        }
        Process *p = dequeue(&ready_queue);
        /* time_quantum 혹은 프로세스 잔여 시간 중 최소값 만큼 실행 시도 */
        int time_slice = (p->remaining_time < time_quantum) ? p->remaining_time :
time_quantum;
        for (int i = 0; i < time_slice; i++) {
            /* 실행 중 I/O 요청 발생 여부 체크 */
            if (p->has_io && !p->io_done
                    && (p->cpu_burst - p->remaining_time) == p->io_start_time) {
                /* I/O 요청 발생 -> 프로세스 대기 큐 이동 */
                printf("  → I/O 요청 발생: 프로세스 %d\n", p->pid);
                p->io_remaining_time = p->io_burst;
                enqueue(&waiting_queue, p);
                current_time++;
                break;  /* 남은 타임퀀텀 포기하고 즉시 루프 탈출 */
            }
            /* 1 tick CPU 실행 */
            p->remaining_time--;
            current_time++;
            if (p->remaining_time == 0) {
                /* 프로세스 완료 */
                p->completion_time = current_time;
                p->turnaround_time = p->completion_time - p->arrival_time;
                p->waiting_time = p->turnaround_time - p->cpu_burst;
                completed++;
                break;  /* 완료 시 루프 탈출 */
            }
        }
```

```
        /* 루프가 정상 종료되었고 프로세스가 남아 있으며 I/O 대기가 아닌 경우 -> 다시 ready_queue 로 */
        if (p->remaining_time > 0 && p->io_remaining_time == 0) {
            enqueue(&ready_queue, p);
        }
    }
    /* ... 결과 출력 ... */
}
```

위 코드에서 time_slice 만큼 for 루프로 반복하며 CPU 를 할당하는데, 그 중 I/O 이벤트 발생을 체크하여 발생 시 즉시 break 로 빠져나가고, 프로세스를 waiting_queue 로 보낸다. I/O 가 발생하지 않고 프로세스가 끝나지 않았으면 주어진 time_quantum 만큼 모두 소모한 후 enqueue(&ready_queue, p)로 프로세스를 다시 준비 큐 끝에 추가한다. update_waiting_queue_rr 함수는 FCFS 의 경우와 유사하게 대기 큐의 프로세스들의 I/O 남은 시간을 감소시키고, I/O 완료 시 준비 큐로 옮긴다. Round Robin 알고리즘의 구현을 통해 타임퀀텀 관리와 선점 시나리오(I/O 등)를 코드로 명시적으로 다루는 방법을 확인할 수 있다.

앞서 나온 소스코드 요약을 통해 각 알고리즘의 구현 핵심과 차이점을 살펴보았다. 실제 코드 전체에서는 각 알고리즘 후에 프로세스들의 WT, TT 결과와 평균을 계산하여 출력하고, 간트 차트 구성 배열(gantt, timeline)을 활용하여 간트 차트를 출력하는 부분도 포함되어 있다. 이러한 출력 부분은 보고서 본문 결과 예시에서 이미 설명된 바와 같다.

5. 참고문헌

1. Silberschatz et al., Operating System Concepts, 9th Ed., Wiley, 2013 (운영체제 교재)
2. 윤용호 외, "웹 기반의 교육용 CPU 스케줄링 시뮬레이터의 설계 및 구현," *정보통신공학회 논문지*, 2014oak.go.kr