OPERATING SYSTEM

Term Project Report

CPU Scheduling Simulator

2023320088

조원형

<목차>

1. 서론

2. 본론

1) 다른 CPU 스케줄링 시뮬레이터에 대한 소개

2) 본 시스템 구성도

3) 모듈 별 설명

4) 시뮬레이터 실행 결과

5) 알고리즘 별 성능 비교

3. 결론

1. 서론

1.1 CPU 스케줄러의 개념

운영체제에서 CPU 스케줄러(CPU Scheduler)는 다수의 프로세스들이 한정된 CPU 자원을 효율적으로 공유할 수 있도록 순서를 정해주는 핵심 컴포넌트이다. 일반적으로 현대 운영체제는 다중 프로세스를 동시에 처리하기 위해 선점(preemptive) 또는 비선점(non-preemptive) 스케줄링 알고리즘을 사용한다. 이러한 알고리즘은 각 프로세스의 도착 시간, CPU 요구 시간, 우선순위, 입출력 요청 등을 기준으로 어떤 프로세스를 다음에 실행할지 결정한다.

CPU 스케줄링은 시스템의 응답 시간(Response Time), 대기 시간(Waiting Time), 처리 시간(Turnaround Time), CPU 사용률(CPU Utilization) 등에 큰 영향을 미치기 때문에, 그 알고리즘의 설계와 선택은 운영체제의 전반적인 성능에 중요한 영향을 미친다.

1.2 구현 프로젝트 개요

본 프로젝트에서는 총 6가지의 대표적인 CPU 스케줄링 알고리즘을 구현하고, 다양한 프로세스 조건 하에서 그 성능을 시뮬레이션하고 비교하였다. 구현된 알고리즘은 다음과 같다:

- First Come First Served (FCFS)
- Shortest Job First (SJF)
- Preemptive Shortest Job First (PSJF)
- Priority Scheduling (Non-preemptive)
- Preemptive Priority Scheduling
- Round Robin (RR, 타임 퀀텀 = 2)

본 시뮬레이터는 단일 I/O 요청이 아닌 복수 I/O 요청이 가능하도록 확장되었으며, 각 프로세스는 0~3회의 I/O를 임의로 수행할 수 있도록 설계되었다. 또한 CPU burst와 I/O burst의 시점과 시간이 랜덤으로 주어짐으로써 현실적인 상황을 반영 할 수 있도록 의도하였다.

시뮬레이터는 C 언어를 기반으로 작성되었으며, 전체 시뮬레이션 과정은 단일 프로세서

환경에서 시간 단위로 동작한다. 이를 통해 각 알고리즘의 특성과 실제 적용 시 장단점을 직접 분석해볼 수 있도록 구현하였다.

2. 본론

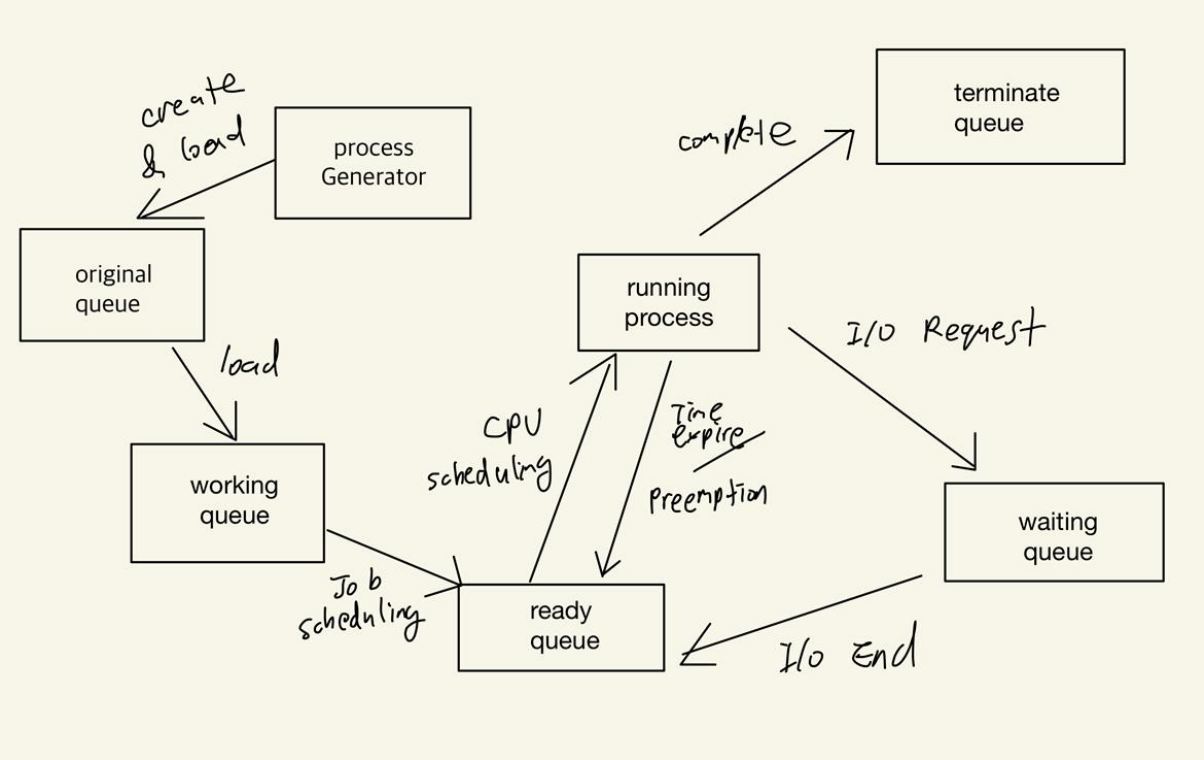
2.1 다른 CPU 스케줄링 시뮬레이터에 대한 소개

CPU 스케줄링 알고리즘은 운영체제 설계에서 핵심적인 부분이며, 다양한 연구 및 교육 목적으로 이를 실험하고 분석할 수 있는 시뮬레이터들이 다수 개발되어 있다. 이러한 시뮬레이터들은 실제 운영체제 수준의 구현 대신, 추상화된 환경에서 스케줄링 알고리즘의 동작 원리와 성능 차이를 관찰하는 것을 목표로 한다.

이러한 시뮬레이터들은 FCFS, SJF등의 다양한 스케줄링 알고리즘을 구현하고 그 성능을 비교할 수 있도록 도와준다. 이는 프로세스의 도착 시간, 버스트 시간 등의 요소를 설정하여 묘사한다.

2.2 시스템 구성도

본 프로젝트에서 구현한 CPU 스케줄링 시뮬레이터는 프로세스의 생성부터 종료까지의 전체 흐름을 시뮬레이션하며, 각 시간 단위(tick)마다 시스템 상태를 갱신하도록 설계되다. 전체 구성은 다음과 같이 나타낼 수 있다.



[구성 요소별 기능 요약]

Process Generator

generate_processes() 함수에서 수행

랜덤한 도착 시간, CPU burst 시간, 우선순위, 최대 3개의 I/O 요청 정보를 갖는 프로세스 생성

Ready Queue

도착 시간에 따라 프로세스가 삽입되며, 스케줄러에 의해 선택 대기

add_ready_queue(), remove_ready_queue()로 조작

Scheduler

FCFS, SJF, PSJF, Priority, Preemptive Priority, RR 등 총 6가지 알고리즘을 지원

schedule() 함수에서 알고리즘에 따라 실행 프로세스를 선택

Running Process

현재 실행 중인 프로세스를 나타내며, CPU burst 및 I/O 요청 여부에 따라 상태가 갱신됨

Waiting Queue

I/O 요청을 수행 중인 프로세스를 보관

I/O 완료 시 다시 ready queue로 복귀

Terminate Queue

실행을 완료한 프로세스 저장

성능 평가 시 통계에 활용됨

이와 같은 구성은 실제 운영체제의 프로세스 상태 전이와 유사한 흐름을 따르며, 각 알고리즘의 특성에 따라 실행 흐름이 동적으로 달라지는 구조를 갖는다.

2.3 알고리즘 및 함수 설명

- 알고리즘

First Come First Served (FCFS)

가장 먼저 도착한 프로세스부터 실행

-선택 기준: Ready Queue에서 도착 순서로 정렬된 첫 번째 프로세스

-구현 포인트:

select_process_fcfs() 함수: ready_queue[0] 반환

schedule() 함수에서 실행 중 프로세스가 없을 경우 호출됩니다.

- Shortest Job First (SJF)

CPU burst가 가장 짧은 프로세스를 먼저 실행

선택 기준: Ready Queue에서 cpu_burst 값이 최소인 프로세스

구현 포인트:

select_process_sjf() 함수: 모든 ready queue를 탐색하여 최소 burst 선택

도착한 순서와 관계없이 burst가 짧은 순으로 실행됩니다.

Preemptive Shortest Job First (PSJF)

현재 실행 중인 프로세스보다 더 짧은 CPU burst를 가진 프로세스가 도착하면 교체

선택 기준: Ready Queue에서 cpu_burst가 가장 작은 프로세스와 비교

구현 포인트:

schedule() 함수 내에서 candidate->cpu_burst < running_process->cpu_burst이면 선점

running_process는 add_ready_queue()로 되돌리고 candidate 실행

Priority Scheduling (Non-preemptive)

우선순위(priority)가 가장 높은(숫자가 작을수록 높음) 프로세스를 선택

선택 기준: Ready Queue에서 가장 낮은 priority 값을 가진 프로세스

구현 포인트:

select_process_priority() 사용

동일한 우선순위라면 FCFS처럼 도착 순서 기준

Preemptive Priority Scheduling

실행 중인 프로세스보다 더 높은 우선순위의 프로세스가 도착하면 선점

선택 기준: Ready Queue에서 가장 낮은 priority를 가진 프로세스를 현재 프로세스와 비교

구현 포인트:

candidate->priority < running_process->priority일 경우 교체

기존 프로세스는 Ready Queue로 복귀

Round Robin (RR)

고정된 Time Quantum 단위로 프로세스를 순환하며 실행

선택 기준: Ready Queue에서 순서대로 프로세스를 실행하되, Time Quantum 경과 시 교체

구현 포인트:

TIME_QUANTUM은 2로 설정

select_process_RR() 함수에서 Time Quantum 경과 확인

교체 시 이전 프로세스를 다시 Ready Queue로 복귀시킵니다.

공통 I/O 처리 로직

I/O 요청 시점: 각 프로세스는 최대 3회까지 I/O 요청 가능 (요청 시점은 CPU 실행 시간 중 무작위로 배정)

진행 흐름:

Tick 단위에서 executed_time == io_request_time 조건 만족 시, io_remaining_time 설정 후 waiting queue로 이동

I/O가 완료되면 다시 ready queue로 복귀

이러한 알고리즘 모듈들은 모두 schedule() 함수 내에서 분기 처리되어 호출되며, 핵심 로직은 매 tick마다 tick() 함수 내에서 프로세스의 상태를 갱신하고 선점 여부를 판단하는 방식으로 작동합니다. 구조적으로는 명확히 모듈화되어 있어, 추가 알고리즘 삽입이나 성능 비교 실험에 유연성을 제공합니다.

-각 함수 별 설명

프로세스 생성 및 초기화 관련

generate_processes(int count)

임의의 수의 프로세스를 생성하며, 각 프로세스에 대해 도착 시간, CPU 버스트, 우선순위, I/O 요청 횟수와 시점, I/O 버스트 등을 무작위로 지정한다.

MAX_IO_REQUESTS 상수에 따라 각 프로세스는 최대 3회의 I/O 요청을 가질 수 있으며, 요청 시점은 CPU burst 내의 랜덤한 시점으로 설정된다.

clone_processes()

생성된 original 프로세스를 working 프로세스 배열로 복제하여, 알고리즘별 실험 시 원

본 손상을 방지한다.

init_simulation()

실행 전 각종 큐(준비 큐, 대기 큐 등)를 초기화하고, 시뮬레이션 관련 변수들을 리셋한다.

큐 관리 함수

add_ready_queue(process* p) / remove_ready_queue(int index)

준비 큐(ready queue)에 프로세스를 추가 또는 삭제한다.

add_to_waiting_queue(process* p) / remove_waiting_queue(int index)

I/O 수행을 위한 대기 큐(waiting queue)를 관리하는 함수들이다.

스케줄링 알고리즘 선택 함수

schedule(SchedulingAlgorithm alg)

현재 선택된 알고리즘에 따라 적절한 프로세스를 선택한다. FCFS, SJF, PSJF, Priority, Preemptive Priority, RR 알고리즘을 모두 지원하며, 선점 여부에 따라 실행 중인 프로세스와 후보 프로세스를 비교해 결정한다.

select_process_fcfs(), select_process_sjf(), select_process_priority()

알고리즘별로 준비 큐에서 다음으로 실행할 프로세스를 선택한다.

select_process_RR(int time_quantum)

라운드 로빈 알고리즘을 위한 선택 로직을 구현한다. 타임 퀀텀이 만료되면 실행 중 프로세스를 준비 큐에 다시 추가한다.

시간 단위 실행 함수

tick(int current_time, SchedulingAlgorithm alg)

실제 시뮬레이션에서 한 틱(1단위 시간)마다 수행되는 핵심 함수이다.

주요 기능:

프로세스 도착 처리 (도착 시간이 현재 시간과 일치하는 프로세스 → ready queue에 추가)

현재 실행 중인 프로세스를 선택 및 실행

I/O 요청 도달 여부 확인 및 처리 (요청 시점에 도달하면 waiting queue로 이동)

실행 완료된 프로세스는 terminated 큐로 이동

각 프로세스의 waiting time, turnaround time, response time 누적

결과 분석 및 출력

evaluate_simulation(SchedulingAlgorithm alg, const char* name, int total_time)

알고리즘 실행이 끝난 후 평균 대기 시간, 평균 반환 시간, 평균 응답 시간, CPU 이용률 등을 계산하고 출력한다.

print_gantt_chart(const char* name, int total_time)

알고리즘의 실행 흐름을 Gantt Chart 형태로 콘솔에 출력한다.

run_simulation(SchedulingAlgorithm alg, const char* name)

시뮬레이션 실행의 총괄 함수로, 알고리즘별 초기화 → 시간 단위 실행 → 결과 분석 과

정을 순차적으로 호출한다.

평가 구조체 및 관리 함수

struct evaluation

알고리즘 실행 결과를 저장하는 구조체로, 평균 시간, CPU 활용률, 시작/종료 시점 등의 정보를 포함한다.

init_evals() / clear_evals()

평가 데이터 구조의 초기화 및 메모리 정리를 위한 함수들이다.

2.4 시뮬레이터 실행 결과 화면

본 프로젝트에서는 6개의 스케줄링 알고리즘(FCFS, SJF, PSJF, Priority, Preemptive Priority, Round Robin)을 동일한 프로세스 집합에 대해 실행하여 결과를 비교하였다. 시뮬레이터는 각 시간 단위마다 CPU 상태를 기록한 Gantt 차트와, 평균 대기 시간(Waiting Time), 평균 반환 시간(Turnaround Time), 평균 응답 시간(Response Time), CPU 이용률, 완료된 프로세스 수 등의 지표를 출력한다.

프로세스 수는 3~8개 중 랜덤이며 이 결과는 6개 프로세스의 경우 결과이다.

1) 프로세스 생성 정보

```
fishingtrap@BOOK-ERP8OOD1BP:~/fishingtrap/test$ ./cpu_scheduling_simulator
PID     Arrival CPU     I/O Requests            Priority
1       13      14      (4@1) (6@4) (2@6)       10
2       12      12      (4@2) (2@5) (6@10)      7
3       1       8       (5@1) (5@2) (5@4)       1
4       6       7       None                    10
5       10      8       None                    8
6       12      12      (3@4) (6@11)            7
```

생성된 프로세스에 대한 정보를 print한 것이다.

IO request에 대해서는 (4@2)의 경우 CPU burst 2일때 4만큼의 IO 처리를 request한다는 뜻이다. 3개의 경우 한 프로세스에서 IO 요청을 3번 발생시킨다.

2) 알고리즘 별 Gantt Chart 및 평가 지표

```
=== First Come First Served ===
Gantt Chart:
| idle | P3 | idle | idle | idle | idle | P4 | P4 | P4 | P4 | P4 | P4 | P4 | P3 | P5 | P5 | P5 | P5 | P5 | P5 | P5 | P5
| P2 | P2 | P6 | P6 | P6 | P6 | P1 | P3 | P3 | P2 | P2 | P2 | P6 | P6 | P6 | P6 | P6 | P6 | P6 | P1 | P1 | P1 | P3 | P3
| P3 | P3 | P2 | P2 | P2 | P2 | P2 | P6 | P1 | P1 | idle | idle | P1 | P1 | P1 | P1 | P1 | P1 | P1 | P1 | P2 | P2 | idle
 | idle | idle | idle | idle | idle | idle |

=== First Come First Served Evaluation ===
Start Time       : 1
End Time         : 68
Avg Waiting Time : 18
Avg Turnaround   : 26
Avg Response     : 6
CPU Utilization  : 91.04%
Completed        : 6

=== Shortest Job First ===
Gantt Chart:
| idle | P3 | idle | idle | idle | idle | P4 | P4 | P4 | P4 | P4 | P4 | P4 | P3 | P5 | P5 | P5 | P5 | P5 | P5 | P5 | P5
| P3 | P3 | P2 | P2 | P6 | P6 | P6 | P6 | P3 | P3 | P3 | P3 | P6 | P6 | P6 | P6 | P6 | P6 | P6 | P2 | P2 | P2 | P1 | idl
e | P2 | P2 | P2 | P2 | P2 | P6 | P1 | P1 | P1 | idle | idle | P2 | P2 | idle | idle | P1 | P1 | idle | idle | P1 | P1 |
 P1 | P1 | P1 | P1 | P1 | P1 | idle | idle |

=== Shortest Job First Evaluation ===
Start Time       : 1
End Time         : 73
Avg Waiting Time : 15
Avg Turnaround   : 23
Avg Response     : 10
CPU Utilization  : 84.72%
Completed        : 6
```

```
=== Preemptive SJF ===
Gantt Chart:
| idle | P3 | idle | idle | idle | idle | P4 | P4 | P4 | P4 | P4 | P4 | P4 | P3 | P5 | P5 | P5 | P5 | P5 | P5 | P5 | P5
| P3 | P3 | P6 | P6 | P6 | P6 | P2 | P3 | P3 | P3 | P3 | P6 | P6 | P6 | P6 | P6 | P6 | P2 | P1 | idle | idle | idle
 | P2 | P6 | P2 | P2 | P1 | P1 | P2 | P2 | P2 | P2 | P2 | P1 | idle | idle | idle | idle | idle | P2 | P2 | P1 | P1 | id
le | idle | P1 | P1 | P1 | P1 | P1 | P1 | P1 |

=== Preemptive SJF Evaluation ===
Start Time       : 1
End Time         : 75
Avg Waiting Time : 15
Avg Turnaround   : 23
Avg Response     : 10
CPU Utilization  : 81.08%
Completed        : 5

=== Priority Scheduling (Non-preemptive) ===
Gantt Chart:
| idle | P3 | idle | idle | idle | idle | P4 | P4 | P4 | P4 | P4 | P4 | P4 | P3 | P2 | P2 | P6 | P6 | P6 | P6 | P3 | P3
| P2 | P2 | P2 | P6 | P6 | P6 | P6 | P6 | P6 | P6 | P3 | P3 | P3 | P3 | P2 | P2 | P2 | P2 | P2 | P6 | P5 | P5 | P5 | P5
| P5 | P5 | P5 | P5 | P2 | P2 | P1 | idle | idle | idle | idle | P1 | P1 | P1 | idle | idle | idle | idle | idle | idle
| P1 | P1 | idle | idle | P1 | P1 | P1 | P1 | P1 |

=== Priority Scheduling (Non-preemptive) Evaluation ===
Start Time       : 1
End Time         : 75
Avg Waiting Time : 18
Avg Turnaround   : 26
Avg Response     : 12
CPU Utilization  : 78.38%
Completed        : 5
```

```
=== Preemptive Priority Scheduling ===
Gantt Chart:
| idle | P3 | idle | idle | idle | idle | P4 | P3 | P4 | P4 | P5 | P5 | P2 | P3 | P3 | P2 | P6 | P6 | P6 | P6 | P3 | P3
| P3 | P3 | P6 | P6 | P6 | P6 | P6 | P6 | P6 | P2 | P2 | P2 | P5 | P5 | P2 | P2 | P2 | P2 | P2 | P6 | P5 | P5 | P5 | P5
| P4 | P2 | P2 | P4 | P4 | P4 | P1 | idle | idle | idle | idle | P1 | P1 | P1 | idle | idle | idle | idle | idle | idle
| P1 | P1 | idle | idle | P1 | P1 | P1 | P1 | P1 |

=== Preemptive Priority Scheduling Evaluation ===
Start Time       : 1
End Time         : 75
Avg Waiting Time : 21
Avg Turnaround   : 29
Avg Response     : 7
CPU Utilization  : 78.38%
Completed        : 5

=== Round Robin (Quantum = 2) ===
Gantt Chart:
| idle | P3 | idle | idle | idle | idle | P4 | P4 | P3 | P4 | P4 | P5 | P5 | P4 | P4 | P2 | P2 | P6 | P6 | P1 | P5 | P5
| P3 | P3 | P4 | P6 | P6 | P2 | P2 | P5 | P5 | P1 | P1 | P3 | P3 | P2 | P6 | P6 | P5 | P5 | P1 | P3 | P3 | P2 | P2 | P6
| P6 | P2 | P2 | P1 | P1 | P6 | P6 | P2 | P1 | P1 | P1 | P6 | P1 | P1 | P1 | P2 | P2 | P1 | P1 | P6 | P1 | idle | idle | idle
 | idle | idle | idle | idle | idle | idle |

=== Round Robin (Quantum = 2) Evaluation ===
Start Time       : 1
End Time         : 66
Avg Waiting Time : 23
Avg Turnaround   : 31
Avg Response     : 2
CPU Utilization  : 93.85%
Completed        : 6
```

<분석>

FCFS는 전통적인 방식으로, 도착 순서대로 실행되기 때문에 짧은 작업이 뒤에 배치될 경우 평균 대기 시간이 증가할 수 있다. 하지만 이번 결과에서는 상대적으로 균형 있는 결과를 보였다.

SJF는 평균 대기 시간과 반환 시간이 가장 낮았으며, 효율적인 처리 순서를 통해 높은 성능을 보였다.

PSJF는 실시간으로 짧은 작업에 우선순위를 부여하지만, I/O 요청과 선점 타이밍이 복잡하게 얽히면서 일부 프로세스가 끝나지 못하는 결과도 나타났다.

Priority (비선점)는 높은 우선순위 작업이 뒤늦게 도착해도 앞선 작업을 기다려야 하므로 응답 시간과 반환 시간이 증가하는 경향이 있었다.

Preemptive Priority는 실시간 대응에는 효과적이지만, 우선순위가 낮은 작업은 지나치게 대기하게 되는 Starvation 현상이 관찰되었다.

Round Robin은 응답 시간이 압도적으로 낮았고, CPU 이용률도 가장 높았다. 그러나 짧은 타임 퀀텀으로 인해 context switching이 많아지고 전체적인 대기 시간이 증가하였다.

2.5 알고리즘들 간의 성능 비교

본 절에서는 앞서 실행한 시뮬레이션 결과를 기반으로, 6가지 CPU 스케줄링 알고리즘 간의 성능을 정량적으로 비교하고 각 특징을 분석한다. 성능 비교 기준은 다음과 같다:

평균 대기 시간 (Avg Waiting Time): CPU를 기다린 총 시간

평균 반환 시간 (Avg Turnaround Time): 프로세스의 완료까지 걸린 총 시간

평균 응답 시간 (Avg Response Time): 처음 CPU를 할당받을 때까지의 시간

CPU 이용률 (CPU Utilization): 유휴 시간 대비 실제 사용 시간의 비율

완료된 프로세스 수 (Completed): 시뮬레이션 내에서 완료된 프로세스의 개수

알고리즘	Avg Waiting	Avg Turnaround	Avg Response	CPU Util (%)	Completed
FCFS	18	26	6	91.04	6
SJF (Non-preemptive)	15	23	10	84.72	6
PSJF (Preemptive SJF)	15	23	10	81.08	5
Priority (Non-preemptive)	18	26	12	78.38	5
Preemptive Priority	21	29	7	78.38	5
Round Robin (Quantum = 2)	23	31	2	93.85	6

FCFS

평균 응답 시간과 대기 시간이 무난하게 나왔다.

걱정되는 점으로는 I/O로 인해 ready queue 뒤로 밀리는 현상이 나오는 것을 볼 수 있다.

SJF/PSJF

평균 대기시간과 반환 시간이 낮아 CPU 효율 측면에서 우수함을 볼 수 있다.

허나 특히 PSJF에서 봤듯이 긴 작업이 뒤로 밀리면서 I/O요청도 같이 밀리면서 모두 complete 안되는 사태가 나오는 것을 볼 수 있다.

Priority, Preemptive Priority

우선순위 기반으로 급한 작업 처리에는 유리하지만 낮은 우선순위의 starvation 현상이

나타나는 것을 볼 수 있다. I/O 요청과도 idle이 나타나는 cpu 비효율적인 모습이 나타났다.

RR

평균 응답 시간이 가장 낮은 것을 볼 수 있다. 대화형 시스템에 어울린다 볼 수 있다.

허나 context swithing이 빈번 해 대기 시간과 반환 시간이 증가하는 것을 볼 수 있다. 그래도 CPU 이용률은 가장 높았다. 다른 이들이 IO요청에 의해 CPU 이용률이 적은 것을 보면 비교가 된다고 할 수 있다.

3. 결론

본 프로젝트에서는 다양한 CPU 스케줄링 알고리즘의 특성을 이해하고, 이를 직접 구현한 시뮬레이터를 통해 실험적으로 비교해보는 경험을 하였다. 구현한 시뮬레이터는 다음과 같은 특징을 갖는다:

6가지 대표적인 알고리즘 구현: FCFS, SJF, PSJF, Priority, Preemptive Priority, Round Robin

I/O 요청 및 처리 시뮬레이션 지원: 각 프로세스는 최대 3회까지 임의의 시점에서 I/O 요청이 가능하며, 해당 요청 시 일시 중단되었다가 복귀한다.

다양한 프로세스 생성: arrival time, cpu burst, priority, I/O request 등 다양한 요소를 무작위로 생성하여 현실성 있는 실험 구성했다,

성능 지표 자동 계산: 평균 대기 시간, 반환 시간, 응답 시간, CPU 이용률 등을 수집하여 각 알고리즘의 장단점을 정량적으로 비교해보았다.

3.1 프로젝트 수행 소감

직접 CPU scheduling simulator를 구현해보면서 프로세스와 그 처리 구조, 스케줄링 알고리즘에 대해 깊이 있게 학습할 수 있었다. 함수를 필요에 따라 나누면서 단계적으로 CPU Scheduling을 알 수 있었다.

처음에는 문제점이 많았었는데 따로 찾아보면서 왜 틀렸고 무엇이 더 필요한지 알아가는 경험은 잊지 못 할 것이다.

구현한 스케줄러에 대해서 현재 IO요청이 굉장히 많은 편인데 시뮬레이션 결과에서 FCFS가 가장 단순하니 성능은 안 좋을 것이라 생각했는데 오히려 starvation의 문제와

IO 처리 시간이 스케줄링에 적용이 안 되어 다른 알고리즘이 성능이 낮게 나와서 놀랐다.

다음 번에 기회가 있다면 aiming 기법 등을 통해 이 문제를 해결하면 더 좋은 simulator를 구현할 수 있을 것이라 생각한다.

그 외로 본 프로젝트에서는 비교적 간단한 알고리즘들을 구현하였지만 실제로는 deadeline요소나 LIF등의 여러 알고리즘의 존재를 생각하면 성능 향상의 가능성이 계속 볼 수 있었다.

Source Code

https://github.com/fishingtrap/251RCOSE34102.git