CPU 스케줄링 알고리즘 성능 분석 시뮬레이터

멀티 I/O 환경에서의 9가지 알고리즘 구현 및 비교

학과명 : 컴퓨터학과
이 름 : 이 성 민
학 번 : 2023320132

Table of Contents

1. 서론

1.1. 프로젝트 목적

본 프로젝트는 운영체제의 핵심 구성 요소인 CPU 스케줄링 알고리즘들을 C언어로 구현하고 성능을 비교 분석하는 것을 목적으로 한다. 기본 6가지의 알고리즘과 실시간 처리 알고리즘 2개를 포함해, 총 9가지의 주요 스케줄링 알고리즘을 구현하여 각각의 특성과 성능을 이해하고, 실시간 시스템에서의 스케줄링 이론을 실제로 적용해보는 것이 주요 목표이다. 또한 Linux 환경에서 CPU Scheduling simulator을 구현함으로써, 실제 Linux 환경에서의 개발 역량을 키우는 것이 목적이다.

1.2. CPU Scheduler

CPU 스케줄링은 OS에서 CPU 자원을 효율적으로 배분하기 위한 핵심 기능 중 하나이다. 여러 프로세스가 동시에 자원 할당을 필요로 할 때, 어떤 프로세스에게 CPU 자원을 할당할지 결정하는 메커니즘이다. 효율적인 CPU 스케줄링은 시스템 전반에 걸쳐, 성능, 사용자 응답성, 그리고 자원 활용률에 직접적인 영향을 미친다.

현대 컴퓨터 시스템에서 CPU 스케줄링은 **다음과 같은 목표를 달성해야 한다:**

- CPU 이용률 최대화: CPU가 유휴 상태 최소화
- 처리량 극대화: 단위 시간당 완료되는 프로세스 수 증가
- 대기 시간 최소화: 프로세스 ready queue 대기 시간 최소화
- 반환 시간 최소화: 프로세스가 도착한 시점부터 완료될 때까지의 시간을 최소화
- 공정성: 모든 프로세스가 적절한 CPU 시간을 할당 받을 수 있도록 보장

1.3. 구현된 스케줄러의 특징

본 프로젝트에서 구현한 CPU 스케줄링 시뮬레이터는 **다음과 같은 주요 특징을 가진다:**

- 다중 알고리즘 지원(FCFS, SJF(P/NP), Priority(P/NP), RR, Priority(Aging))
- 멀티 I/O 지원
- 실시간 스케줄링 특화 기능(RMS, EDF)
- 종합적 성능 분석
- 랜덤 생성 프로세스 정보 저장 기능(process-*.txt 확장자)
- process-*.txt 파일 불러오기 기능
- 실시간 프로세스 정보 입력 기능
- 상세 report 생성 및 저장 기능 (<사용자 정의 이름>.txt)

2. 시스템 설계 및 구현

2.1. 기존 CPU 스케줄링 시뮬레이터 분석

기존의 CPU 스케줄링 시뮬레이터들은 대부분 교육용 목적으로 개발되어 단일 알고리즘

분석이나 제한적인 I/O 모델링에 그치는 경우가 많다. **주요 한계점은 다음과 같다:**

- **제한적 알고리즘 지원**: 대부분 3-4개의 기본 알고리즘만 구현
- **단순한 I/O 모델**: 단일 I/O 버스트 또는 I/O 무시
- **실시간 스케줄링 부재**: 일반 스케줄링에만 집중
- **성능 분석 도구 부족**: 기본 메트릭만 제공, 종합 비교 기능 없음

본 시뮬레이터는 이러한 한계를 극복하여 실무에 가까운 환경을 모델링하고, 다양한 스케줄링 시나리오에서의 종합적 성능 분석을 지원한다.

2.2. Block diagram

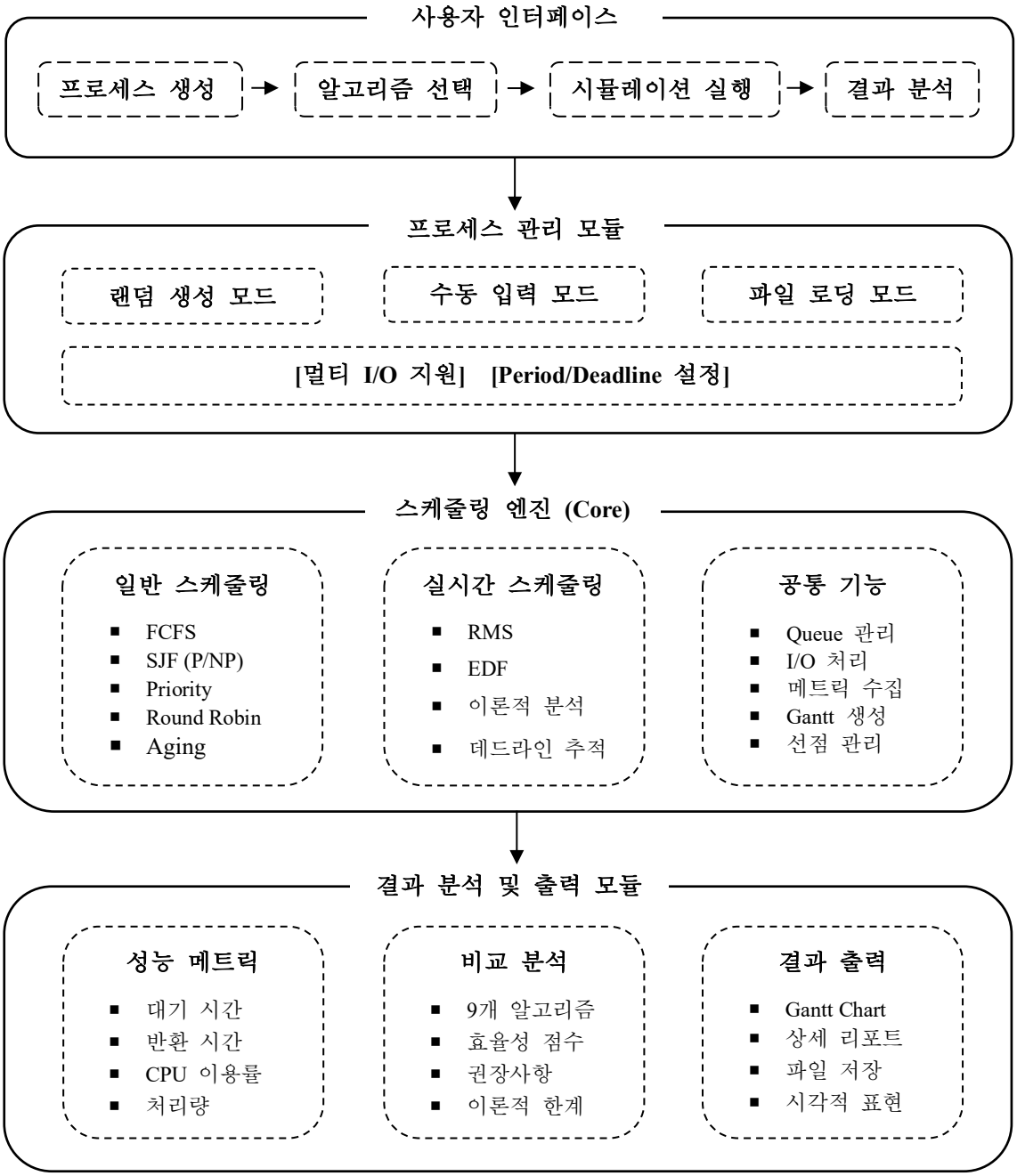

Figure 1. 시뮬레이터의 시스템 구성도

2.3. Muti-I/O Creation

본 시뮬레이터는 프로세스당 최대 3개의 I/O 작업을 지원하며, 다음과 같은 방식으로 처리된다.

I/O 작업 생성:

- 랜덤 I/O 개수: rand() % 4 를 사용하여 0~3개의 I/O 작업이 랜덤하게 생성
- I/O 시작 시간: CPU 버스트 시간 내에서 rand() % (cpu_burst - 1) + 1로 결정
- I/O 지속 시간: rand() % 5 + 1로 1~5 단위 시간 랜덤 생성
- 중복 방지: 동일한 시작 시간을 가진 I/O 작업이 생성되지 않도록 used_start_times 배열로 관리

I/O 생성 조건:

- CPU Burst time >= 2일 때만 I/O 작업 생성 가능
- CPU Burst time < 2인 경우, I/O 작업이 생성되지 않음
- 사용 가능한 시작 시간이 모두 사용된 경우 추가 I/O 생성 중단

I/O 작업 관리:

- init_process_io(): I/O 배열 초기화(시작 시간 -1로 설정)
- add_io_to_process(): 프로세스에 I/O 작업 추가
- sort_io_operations(): I/O 작업을 시작 시간 순으로 정렬

2.4. Data Creation Method

본 시뮬레이터에서는 세 가지 데이터 생성 방식을 지원한다.

2.4.1. 랜덤 모드 (mode = 'y')

- 도착 시간: rand() % 10 으로 0~9 사이 랜덤 생성
- CPU 버스트: rand() % 10 + 1로 1~10 사이 랜덤 생성
- 우선 순위: rand() % 10 + 1로 1~10 사이 랜덤 생성
- Utilization 기반 Period 계산:
  - 프로세스 개수에 따른 목표 개별 utilization 설정
    - ≤2개: 40%, ≤4개: 25%, ≤6개: 18%, >6개: 15%
  - 실제 실행시간 (CPU + I/O 시간) / 목표 utilization으로 Period 계산
  - ±20% 랜덤 변동 적용하여 현실적인 시나리오 구현
- 랜덤 생성 된 프로세스 정보는 사용자의 요구에 따라 test_files/ 폴더에 파일 저장 가능

2.4.2. 수동 입력 모드 (mode = 'n')

- 사용자가 각 프로세스의 모든 파라미터를 직접 입력
- PID, 도착시간, CPU 버스트, 우선순위, 데드라인, Period 입력

- I/O 작업 개수와 각 I/O의 시작시간, 지속시간 개별 입력

2.4.3. 파일 입력 모드 (mode = 'f')

- test_files/ 디렉토리에서 process-*.txt 파일 자동 탐색
- **파일 형식:**

```
[프로세스 개수]
[PID] [도착시간] [CPU 버스트] [우선순위] [데드라인] [Period] [I/O 개수]
[I/O1 시작시간] [I/O1 지속시간]
[I/O2 시작시간] [I/O2 지속시간]
...
```

2.4.4. 프로세스 초기화

모든 모드에서 다음 값들이 초기화 된다:

- remaining_time = cpu_burst
- progress = 0
- comp_time = 0
- waiting_time = 0
- turnaround_time = 0
- missed_deadline = 0

2.5. Circular Queue 구조 및 구현

본 시뮬레이터는 효율적인 프로세스 관리를 위해 원형 큐(Circular Queue) 자료구조를 구현하여 사용한다.

1. **구성 요소**

   데이터 저장 배열: 큐에 저장될 프로세스 ID들을 담는 정수형 배열로, 최대 큐 크기만큼의 요소를 저장할 수 있다.

   - **전단 인덱스(Front Index):** 큐의 맨 앞 요소를 가리키는 인덱스로, 다음에 제거될 요소의 위치를 나타낸다.
   - **후단 인덱스(Rear Index):** 큐의 맨 뒤 요소를 가리키는 인덱스로, 마지막으로 삽입된 요소의 위치를 나타낸다.
   - **요소 개수 카운터:** 현재 큐에 저장된 요소의 총 개수를 추적하는 정수형 변수로, 큐의 공백 상태와 포화 상태를 판단하는 데 사용된다.

2. **설계 특징**

   이러한 구조는 원형 큐(Circular Queue) 방식으로 구현되어 메모리 효율성을 극대화하였다. 전단과 후단 인덱스를 통해 O(1) 시간 복잡도로 삽입과 삭제 연산을 수행할 수 있으며, 별도의 카운터를 유지함으로써 큐의 상태를 효율적으로 관리할 수 있다.

3. **동작 원리**

큐는 FIFO(First In First Out) 원칙에 따라 동작하며, 새로운 프로세스는 후단에서 삽입되고 기존 프로세스는 전단에서 제거된다. 인덱스들은 배열의 크기를 모듈로 연산하여 순환적으로 관리되어 배열 공간을 최대한 활용할 수 있도록 설계되었다.

4. 핵심 기능 구현

- **큐 초기화(init_queue)**
  - front = 0, rear = 0, count = 0으로 초기화
  - 빈 큐 상태로 설정
- **큐 상태 확인**
  - is_empty() : count == 0 으로 빈 큐 여부 확인
  - is_full() : count == MAX_QUEUE_SIZE 로 포화 상태 확인
- **큐 삽입**
  - 큐가 가득 차지 않은 경우에만 삽입 수행
  - rear 위치에 데이터 저장 후 (rear + 1) % MAX_QUEUE_SIZE 로 순환 이동
  - count 증가로 큐 크기 관리
- **큐 제거**
  - 큐가 비어있지 않은 경우에만 제거 수행
  - front 위치의 데이터 반환 후 (front + 1) % MAX_QUEUE_SIZE로 순환 이동
  - count 감소 후 제거된 값 반환
  - 빈 큐일 경우 -1 반환
- **큐 조회**
  - 큐의 맨 앞 요소를 제거하지 않고 조회
  - 빈 큐일 경우 -1 반환

2.6. CPU Scheduling Algorithm

2.6.1. Basic Structure

1. 기본 시뮬레이션 루프 구조

```
시뮬레이션 초기화:
   ready_queue, running_queue, waiting_queue 초기화
   각 프로세스의 상태 변수들 초기화
   time = 0, completed = 0

모든 프로세스가 완료될 때까지 반복:
   ┌─ 1 단계: 알고리즘별 특수 처리 (선택적)
   │   - Round Robin: 타임 퀀텀 만료 검사 (최우선)
   │   - Aging: 나이 증가 및 우선순위 승격 (최우선)
   │
   ├─ 2 단계: 프로세스 도착 처리
   │   현재 시간에 도착하는 모든 프로세스를 ready queue 에 추가
   │   선점형 알고리즘의 경우 즉시 선점 조건 검사
   │
   ├─ 3 단계: I/O 완료 처리
```

```
|   I/O 대기시간이 완료된 프로세스들을 ready queue 로 복귀
|   선점형 알고리즘의 경우 즉시 선점 조건 검사
|
├─ 4 단계: Ready Queue 정렬/관리
|   각 알고리즘의 우선순위 기준에 따라 ready queue 정렬
|   (SJF: 남은시간순, Priority: 우선순위순, RMS: Period 순, EDF: Deadline 순)
|
├─ 5 단계: CPU 스케줄링
|   running queue 가 비어있고 ready queue 에 프로세스가 있으면
|   적절한 프로세스를 선택하여 CPU 할당
|
├─ 6 단계: 프로세스 실행
|   현재 실행 중인 프로세스를 1 time 단위로 실행
|   실행 중이 아니면 유휴시간 기록
|
├─ 7 단계: 상태 변화 처리
|   I/O 시작: 프로세스를 waiting 상태로 전환
|   프로세스 완료: 완료 시간 기록 및 통계 갱신
|
└─ 8 단계: 대기시간 갱신 및 시간 진행
    ready queue 의 모든 프로세스 대기시간 증가
    시간을 1 time 단위로 증가
```

2. 프로세스 상태 전이도

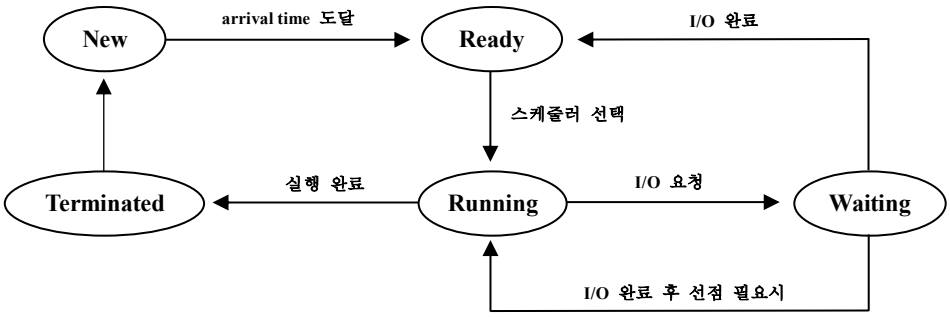

Figure 2. 프로세스 상태 전이도

2.6.2. General Scheduling Algorithms

1. 공통 구조 (Common Framework)

A. 공통 Input/Output

- Input: 프로세스 집합, 프로세스 개수, (알고리즘별 추가 파라미터)
- Output: 성능 메트릭 (총 실행시간, 유휴시간, 평균 대기시간, 평균 반환시간)

B. 공통 초기화

```
Ready queue, Running queue 초기화
I/O 대기 큐 설정 (각 프로세스별 대기시간 추적)
시간 카운터, 완료 프로세스 수, 유휴시간 초기화
알고리즘별 특수 자료구조 초기화
```

C. 공통 메인 루프 구조

```
while 모든 프로세스가 완료되지 않음:
    1) 알고리즘별 특수 이벤트 처리 (선점, 타임퀀텀 등)
    2) 프로세스 도착 처리
    3) I/O 완료 처리
    4) 스케줄링 결정 (알고리즘별 정책 적용)
    5) CPU 할당
    6) 프로세스 실행
    7) 대기시간 관리
    8) 시간 증가
```

D. 공통 프로세스 실행 로직

```
if running queue에 실행 중인 프로세스 존재:
    현재_프로세스 ← running queue의 프로세스
    프로세스 진행도 1 증가

    if 현재 진행도에서 I/O 요청 발생:
        프로세스를 I/O 대기 상태로 전환
        해당 I/O 버스트 시간만큼 대기 설정

    else if 프로세스 완료:
        완료시간, 반환시간, 대기시간 계산
        완료 프로세스 수 증가
        running queue에서 제거

else:
    시스템 유휴상태 기록
    유휴시간 증가
```

E. 공통 대기시간 관리

```
ready queue의 모든 프로세스 대기시간 1 증가
시간 1 증가
```

2. FCFS (First Come First Served) 스케줄링

A. 스케줄링 정책

- **기준**: 도착 순서 (FIFO)
- **선점성**: 비선점

B. 특수 처리 단계

i. 특수 이벤트 처리: 없음 (공통 구조만 사용)

ii. 스케줄링 결정:

```
if running queue가 비어있고 ready queue에 프로세스 존재:
    ready queue의 가장 앞 프로세스 선택 (FIFO 순서)
    선택된 프로세스를 running queue로 이동
```

C. 특징

- 가장 단순한 구현
- Convoy effect 발생 가능
- 비선점이므로 응답시간이 길어질 수 있음

3. Non-Preemptive SJF

A. 추가 자료구조

- 프로세스 구조체의 remaining_time 멤버 변수 활용

B. 스케줄링 정책

- **기준**: 남은 실행시간 (최단 우선)
- **선점성**: 비선점

C. 특수 처리 단계

i. 특수 이벤트 처리: 없음

ii. 스케줄링 결정:

```
if ready queue에 프로세스 존재:
    ready queue를 남은 실행시간 순으로 정렬 (오름차순)

if running queue가 비어있고 ready queue에 프로세스 존재:
    가장 짧은 남은 실행시간을 가진 프로세스 선택
    선택된 프로세스를 running queue로 이동
```

iii. 추가 업데이트:

```
프로세스 실행 시:
    남은 실행시간 = 총 CPU 버스트 - 현재 진행도
```

D. 특징

- 평균 대기시간 최적화
- 긴 프로세스의 기아 상태 가능

4. Preemptive SJF

A. 추가 자료구조

- 프로세스 구조체의 remaining_time 멤버 변수 활용

B. 스케줄링 정책

- **기준**: 남은 실행시간 (최단 우선)
- **선점성**: 선점

C. 특수 처리 단계

i. 선점적 이벤트 처리:

```
프로세스 도착 시:
    if running queue에 프로세스 실행 중:
        새_프로세스_남은시간 vs 실행중_프로세스_남은시간 비교
        if 새 프로세스가 더 짧은 시간:
            실행 중인 프로세스 즉시 선점
            선점된 프로세스를 ready queue로 이동
            새 프로세스를 running queue로 즉시 할당

I/O 완료 시:
    if running queue에 프로세스 실행 중:
        I/O완료_프로세스_남은시간 vs 실행중_프로세스_남은시간 비교
        if I/O 완료 프로세스가 더 짧은 시간:
            실행 중인 프로세스 즉시 선점
            I/O 완료 프로세스를 running queue로 즉시 할당
```

ii. 스케줄링 결정: Non-Preemptive SJF 와 동일

iii. 추가 업데이트: Non-Preemptive SJF 와 동일

D. 특징

- 최적 평균 대기시간 보장
- 빈번한 선점으로 컨텍스트 스위칭 오버헤드 증가

5. Non-Preemptive Priority

A. 스케줄링 정책

- **기준**: 우선순위 (낮은 숫자 = 높은 우선순위)
- **선점성**: 비선점

B. 특수 처리 단계

i. 특수 이벤트 처리: 없음

ii. 스케줄링 결정:

```
if ready queue에 프로세스 존재:
    ready queue를 우선순위 순으로 정렬 (높은 우선순위 우선)

if running queue가 비어있고 ready queue에 프로세스 존재:
    가장 높은 우선순위 프로세스 선택
    선택된 프로세스를 running queue로 이동
```

C. 특징

- 중요도에 따른 처리 가능
- 우선순위 역전 상황 허용 (비선점 특성)
- 낮은 우선순위 프로세스의 기아 상태 가능

6. Preemptive Priority

A. 스케줄링 정책

- **기준**: 우선순위 (낮은 숫자 = 높은 우선순위)
- **선점성**: 선점

B. 특수 처리 단계

i. 선점적 이벤트 처리:

```
프로세스 도착 시:
    if running queue 에 프로세스 실행 중:
        새_프로세스_우선순위 vs 실행중_프로세스_우선순위 비교
        if 새 프로세스가 더 높은 우선순위 (낮은 숫자):
            실행 중인 프로세스 즉시 선점
            새 프로세스를 running queue 로 즉시 할당

I/O 완료 시:
    if running queue 에 프로세스 실행 중:
        I/O 완료_프로세스_우선순위 vs 실행중_프로세스_우선순위 비교
        if I/O 완료 프로세스가 더 높은 우선순위:
            실행 중인 프로세스 즉시 선점
            I/O 완료 프로세스를 running queue 로 즉시 할당
```

ii. 스케줄링 결정: Non-Preemptive Priority 와 동일

C. 특징

- 우선순위 역전 최소화
- 높은 우선순위 프로세스의 빠른 응답시간 보장

7. Round Robin

A. 추가 Input

- 타임 퀀텀 설정값

B. 추가 자료구조

- 각 프로세스별 타임 퀀텀 카운터 배열

C. 스케줄링 정책

- **기준**: FIFO + 타임 퀀텀
- **선점성**: 시간 기반 선점

D. 특수 처리 단계

i. 타임 퀀텀 만료 처리 (최우선):

```
if running queue 에 실행 중인 프로세스 존재:
    if 현재_프로세스_타임퀀텀 == 설정된_타임퀀텀:
        실행 중인 프로세스를 ready queue 맨 뒤로 이동
```

```
    해당 프로세스의 타임 퀀텀 카운터 초기화
```

ii. I/O 완료 시 특수 처리:

```
I/O 완료 시:
    프로세스를 ready queue 로 복귀
    해당 프로세스의 타임 퀀텀 카운터 초기화 (공정성 보장)
```

iii. 스케줄링 결정:

```
if running queue 가 비어있고 ready queue 에 프로세스 존재:
    ready queue 의 가장 앞 프로세스 선택 (FIFO)
    선택된 프로세스를 running queue 로 이동
```

iv. 프로세스 실행 시 추가 처리:

```
프로세스 실행 시:
    해당 프로세스의 타임 퀀텀 카운터 1 증가

I/O 요청 또는 완료 시:
    해당 프로세스의 타임 퀀텀 카운터 초기화
```

E. 특징

- 모든 프로세스 공정한 CPU 할당
- 타임 퀀텀 크기에 따른 성능 트레이드오프

8. Preemptive Priority with Aging

A. 추가 자료구조

```
각 프로세스별 나이 추적 배열
에이징 임계값 (예: 3 시간 단위)
원본 우선순위 백업 복사본(기존 프로세스 정보 유지)
```

B. 스케줄링 정책

- 기준: 동적 우선순위 (에이징 적용)
- 선점성: 선점 + 에이징 기반 추가 선점

C. 특수 처리 단계

i. 에이징 처리 (최우선):

```
에이징_후보[] ← 빈 배열

for ready queue 의 각 프로세스:
```

```
    해당_프로세스_나이 1 증가
    if 프로세스_나이 >= 에이징_임계값 AND 프로세스_우선순위 > 1:
        프로세스를 에이징_후보에 추가

for 각 에이징_후보자:
    기존_우선순위 ← 현재_우선순위
    프로세스_우선순위 1 감소 (우선순위 상승)
    에이징_이벤트_로그_기록(시간, 프로세스 ID, 나이, 기존우선순위, 새우선순위)
    해당_프로세스_나이 초기화

if 에이징_후보가 존재:
    ready queue를 우선순위 순으로 재정렬
```

ii. 에이징 기반 선점 검사:

```
if running queue에 프로세스 실행 중 AND ready queue에 프로세스 존재:
    실행중_프로세스 ← running queue의 프로세스
    최고_대기_프로세스 ← ready queue의 첫 번째 프로세스

    if 최고_대기_프로세스_우선순위 < 실행중_프로세스_우선순위:
        실행 중인 프로세스 선점
        최고 대기 프로세스를 running queue로 즉시 할당
        승진된_프로세스_나이 초기화
        에이징_선점_이벤트_로그_기록
```

iii. 일반 선점 처리:

```
프로세스 도착 및 I/O 완료 시:
    Preemptive Priority와 동일한 선점 로직 적용
```

iv. CPU 할당 시 특수 처리:

```
프로세스가 running queue로 이동 시:
    해당_프로세스_나이 초기화
```

v. 종료 처리:

```
시뮬레이션 완료 후:
    모든 프로세스의 우선순위를 원본값으로 복원
    에이징 분석 결과 출력
```

D. 특징

- 기아 상태 방지 메커니즘
- 동적 우선순위 조정으로 공정성 향상
- 복잡한 다단계 처리 순서

9. 핵심 차이점 요약

A. 이벤트 처리 순서

- 일반 알고리즘: 도착 → I/O 완료 → 스케줄링
- Round Robin: 타임 퀀텀 만료 → 도착 → I/O 완료 → 스케줄링
- Priority with Aging: 에이징 → 에이징 선점 → 도착 → I/O 완료 → 스케줄링

B. 선점 메커니즘

- 비선점형 (FCFS, Non-Preemptive SJF/Priority): CPU 할당 후 완료/I/O까지 실행
- 선점형 (Preemptive SJF/Priority): 도착/I/O 완료 시점에서 즉시 선점 검사
- 시간 기반 선점 (Round Robin): 타임 퀀텀 만료 시 강제 선점
- 복합 선점 (Priority with Aging): 일반 선점 + 에이징 기반 선점

C. 스케줄링 기준

- FCFS: 도착 순서
- SJF: 남은 실행시간
- Priority: 우선순위 (고정 또는 동적)
- Round Robin: FIFO + 시간 분할

2.6.3. Real-time Scheduling Algorithms

1. Rate Monotonic Scheduling (RMS)

A. Input/Output

- **Input**: 프로세스 집합, 프로세스 개수, 최대 시뮬레이션 시간
- **Output**: 성능 메트릭 및 데드라인 미스 정보

B. 알고리즘 단계

i. 스케줄링 가능성 분석 수행

- 각 프로세스의 이용률 계산 (실행시간/주기)
- 전체 시스템 이용률 산출
- RMS 이론적 한계 n(2^(1/n) - 1)와 비교하여 스케줄링 가능성 예측

ii. 동적 프로세스 인스턴스 관리 초기화

- 원본 프로세스 정보 보존
- 동적 확장 가능한 프로세스 배열 할당
- 각 프로세스의 다음 도착 시간 추적 배열 설정
- I/O 대기 큐 및 스케줄링 큐 초기화

iii. 최대 시뮬레이션 시간까지 반복

- **새로운 프로세스 인스턴스 생성:**

```
for 각 프로세스 i:
   if 현재시간 == next_arrival[i]: // 다음 arrival time 값을 지니고 있는 변수(초기엔 처음 도착 시간)
       새 프로세스 인스턴스 생성
```

```
        if 첫 번째 도착:
            deadline = 원본_deadline
        else:
            period_count = (현재시간 - 초기도착시간) / 주기
            deadline = 원본_deadline + period_count * 주기

        주기 기반 선점 검사:
            if 새_프로세스.주기 < 실행중_프로세스.주기:
                실행중 프로세스 선점
                새 프로세스를 running queue 에 삽입
            else:
                새 프로세스를 ready queue 에 삽입

        next_arrival[i] += 주기
```

- **I/O 완료 처리 및 주기 기반 선점 검사:**

```
for 각 프로세스 인스턴스:
    if I/O 완료:
        주기 기반 선점 검사:
            if I/O 완료_프로세스.주기 < 실행중_프로세스.주기:
                실행중 프로세스 선점
                I/O 완료 프로세스를 running queue 에 삽입
            else:
                I/O 완료 프로세스를 ready queue 에 삽입
```

- **가장 짧은 주기를 가진 프로세스 스케줄링:**

```
ready queue 를 주기 기준으로 정렬
if running queue 가 비어있고 ready queue 에 프로세스 존재:
    가장 짧은 주기의 프로세스를 running queue 로 이동
```

- **프로세스 실행 및 데드라인 미스 검사:**

```
if running queue 에 프로세스 존재:
    프로세스 실행 (progress++)

    if I/O 시작점 도달:
        프로세스를 I/O 대기 상태로 전환

    if 프로세스 완료:
        데드라인 미스 검사:
            if 완료시간 > 절대_데드라인:
                데드라인 미스 로그 기록
                미스 카운터 증가
```

2. Earliest Deadline First (EDF)

A. Input/Output

- **Input**: 프로세스 집합, 프로세스 개수, 최대 시뮬레이션 시간
- **Output**: 성능 메트릭 및 데드라인 미스 정보

B. 알고리즘 단계

i. 스케줄링 가능성 분석 수행

- 각 프로세스의 이용률 계산 (실행시간/주기)
- 전체 시스템 이용률 산출
- EDF 이론적 한계 ≤ 1.0과 비교하여 스케줄링 가능성 예측

ii. 동적 프로세스 인스턴스 관리 초기화

- 원본 프로세스 정보 보존
- 동적 확장 가능한 프로세스 배열 할당
- 각 프로세스의 다음 도착 시간 추적 배열 설정
- I/O 대기 큐 및 스케줄링 큐 초기화

iii. 최대 시뮬레이션 시간까지 반복

- **새로운 프로세스 인스턴스 생성:**

```
for 각 프로세스 i:
   if 현재시간 == next_arrival[i]:
       새 프로세스 인스턴스 생성

       if 첫 번째 도착:
           deadline = 원본_deadline
       else:
           period_count = (현재시간 - 초기도착시간) / 주기
           deadline = 원본_deadline + period_count * 주기

       데드라인 기반 선점 검사:
           if 새_프로세스.deadline < 실행중_프로세스.deadline:
               실행중 프로세스 선점
               새 프로세스를 running queue 에 삽입
           else:
               새 프로세스를 ready queue 에 삽입

       next_arrival[i] += 주기
```

- **I/O 완료 처리 및 데드라인 기반 선점 검사:**

```
for 각 프로세스 인스턴스:
   if I/O 완료:
       데드라인 기반 선점 검사:
           if I/O 완료_프로세스.deadline < 실행중_프로세스.deadline:
               실행중 프로세스 선점
               I/O 완료 프로세스를 running queue 에 삽입
           else:
               I/O 완료 프로세스를 ready queue 에 삽입
```

- **가장 빠른 데드라인을 가진 프로세스 스케줄링:**

```
ready queue를 데드라인 기준으로 정렬
if running queue가 비어있고 ready queue에 프로세스 존재:
    가장 빠른 데드라인의 프로세스를 running queue로 이동
```

- **프로세스 실행 및 데드라인 미스 검사:**

```
if running queue에 프로세스 존재:
    프로세스 실행 (progress++)

    if I/O 시작점 도달:
        프로세스를 I/O 대기 상태로 전환

    if 프로세스 완료:
        데드라인 미스 검사:
            if 완료시간 > 절대_데드라인:
                데드라인 미스 로그 기록
                미스 카운터 증가
```

3. 주요 차이점

구분	RMS	EDF
우선순위 기준	주기(Period) - 짧을수록 높음	데드라인(Deadline) - 빠를수록 높음
선점 조건	새 프로세스.주기 < 실행중.주기	새 프로세스.deadline < 실행중.deadline
이론적 한계	$n\left(2^{\frac{1}{n}}-1\right)$ (약 69%~100%)	≤ 1.0 (100%)
우선순위 특성	정적 (주기 기반 고정)	동적 (데드라인 기반 변동)
복잡도	낮음 (정적 우선순위)	높음 (동적 우선순위)

Table 1. 실시간 알고리즘(RMS, EDF)의 주요 차이점

2.7. CPU 스케줄링 알고리즘별 경계 상황 처리 분석

CPU 스케줄링 시뮬레이션에서 경계 상황(Edge Case)은 시뮬레이션의 정확성과 일관성을 결정하는 중요한 요소이다. 본 분석에서는 구현된 9가지 스케줄링 알고리즘에서 발생할 수 있는 주요 경계 상황들과 각 알고리즘의 처리 방식을 체계적으로 분석한다.

2.7.1. 주요 경계 상황 분류

1. 동시성 관련 경계 상황

- **동시 프로세스 도착:** 같은 시간에 여러 프로세스가 도착하는 경우
- **I/O 완료와 프로세스 도착의 동시 발생:** I/O 작업 완료와 새로운 프로세스 도착이 같은 시점에 발생하는 경우

- **동시 I/O 완료:** 여러 프로세스의 I/O 작업이 동시에 완료되는 경우
- **타임 퀀텀 만료와 동시 이벤트:** Round Robin에서 타임 퀀텀 만료와 다른 이벤트가 동시에 발생하는 경우

2. 우선순위 결정 경계 상황

- **동일한 스케줄링 기준값:** CPU 버스트 시간, 우선순위, 데드라인, 주기 등이 동일한 프로세스들의 처리
- **에이징 임계값 동시 도달:** Priority with Aging에서 여러 프로세스가 동시에 에이징 조건을 만족하는 경우
- **실시간 프로세스의 주기적 인스턴스 생성:** RMS/EDF에서 동일한 시점에 여러 프로세스 인스턴스가 생성되는 경우

2.7.2. 알고리즘별 경계 상황 처리 분석

1. FCFS (First Come First Served) 스케줄링

A. 동시 도착 프로세스 처리

FCFS 알고리즘에서 동시에 도착하는 프로세스들은 for 루프의 실행 순서에 따라 처리된다. 프로세스 ID가 작은 순서로 반복문이 실행되므로, 동시 도착 시 PID가 낮은 프로세스가 Ready Queue에 먼저 삽입되어 우선 처리된다.

B. I/O 완료와 프로세스 도착의 동시 처리

동일한 for 루프 내에서 프로세스 도착 처리와 I/O 완료 처리가 순차적으로 이루어진다. 각 프로세스에 대해 먼저 도착 시간을 확인하고, 이후 I/O 완료를 확인하므로, 동시 발생 시 새로 도착하는 프로세스가 먼저 Ready Queue에 삽입되고, I/O 완료 프로세스가 나중에 삽입된다.

C. 동시 I/O 완료 처리

여러 프로세스의 I/O가 동시에 완료되는 경우, PID 순서에 따라 Ready Queue에 순차적으로 삽입된다. FCFS의 특성상 별도의 정렬이 수행되지 않으므로 삽입 순서가 그대로 실행 순서가 된다.

2. Non-Preemptive SJF 스케줄링

A. 동일한 남은 실행시간 처리

동일한 남은 실행시간(remaining_time)을 가진 프로세스들의 경우, 정렬 함수에서 차이가 0으로 반환되어 정렬이 발생하지 않는다. 결과적으로 Ready Queue 내에서의 기존 순서가 유지되며, 이는 궁극적으로 PID 순서에 따른 처리로 귀결된다.

B. I/O 완료 후 Ready Queue 재정렬

I/O 완료 후 Ready Queue로 복귀하는 프로세스는 Queue에 삽입된 후, CPU 할당 직전에 남은 실행시간 기준으로 재정렬된다. 이때 동일한 남은 실행시간을 가진 프로세스들은 PID 순서를 유지한다.

3. Preemptive SJF 스케줄링

A. 동일한 잔여 시간 프로세스의 선점 방지

현재 실행 중인 프로세스와 동일한 잔여 실행 시간을 가진 프로세스가 도착하거나 I/O 완료될 경우, 선점 조건 remaining_i < remaining_current가 false가 되므로 선점이 발생하지 않는다. 이는 불필요한 컨텍스트 스위치를 방지하는 효과적인 구현이다.

B. 즉시 선점 검사 메커니즘

프로세스 도착 시점과 I/O 완료 시점에서 즉시 선점 검사가 수행된다. 새로운 프로세스나 I/O 완료 프로세스의 잔여 시간이 현재 실행 중인 프로세스보다 짧을 경우 즉시 선점이 발생하며, 선점된 프로세스는 Ready Queue로 이동한다.

4. Non-Preemptive Priority 스케줄링

A. 동일한 우선순위 프로세스 처리

동일한 우선순위를 가진 프로세스들은 정렬 과정에서 순서 변경이 발생하지 않으므로, Ready Queue 내에서 PID 순서를 유지한다. 이는 사실상 동일 우선순위 내에서 FCFS 방식으로 처리됨을 의미한다.

B. 우선순위 역전 상황의 자연적 발생

높은 우선순위 프로세스가 I/O 대기 중일 때 낮은 우선순위 프로세스가 실행되는 우선순위 역전 상황이 발생할 수 있다. 비선점형 특성상 실행 중인 낮은 우선순위 프로세스는 완료되거나 I/O 요청할 때까지 계속 실행된다.

5. Preemptive Priority 스케줄링

A. 동일한 우선순위에서의 선점 방지

동일한 우선순위를 가진 프로세스가 도착하거나 I/O 완료될 경우, 선점 조건 processes[i].priority < processes[current].priority가 false가 되므로 선점이 발생하지 않는다.

B. 즉시 선점 검사와 우선순위 역전 방지

프로세스 도착 시점과 I/O 완료 시점에서 즉시 우선순위 기반 선점 검사가 수행된다. 이를 통해 우선순위 역전 상황을 최소화하고 높은 우선순위 프로세스의 빠른 실행을 보장한다.

6. Round Robin 스케줄링

A. 타임 퀀텀 만료 우선 처리

중요한 구현 특징: 타임 퀀텀 만료 처리가 새로운 프로세스 도착 처리보다 먼저 수행된다. 이는 다음과 같은 처리 순서를 보장한다:

1) 타임 퀀텀 만료 검사 및 컨텍스트 스위치
2) 새로운 프로세스 도착 처리
3) I/O 완료 처리

B. I/O 완료 시 타임 퀀텀 리셋

I/O 완료 프로세스는 time_quantum[i] = 0으로 리셋되어 새로운 타임 퀀텀을 할당받

는다. 이는 I/O 대기로 인한 불이익을 방지하는 공정한 처리 방식이다.

C. 동시 도착 프로세스의 순환 순서

동시에 도착한 프로세스들은 PID 순서에 따라 Ready Queue에 삽입되며, 이 순서가 Round Robin의 순환 순서를 결정한다.

7. Priority Scheduling with Aging

A. 에이징 처리의 우선순위

구현 순서의 중요성: 에이징 처리가 모든 다른 이벤트보다 먼저 수행된다:

1) 에이징 대상 식별: Ready Queue의 모든 프로세스 age 증가
2) 우선순위 조정: 임계값 도달 프로세스의 우선순위 향상
3) 에이징 기반 선점: 우선순위 변경으로 인한 선점 검사
4) 새로운 프로세스 도착 처리
5) I/O 완료 처리

B. 에이징 임계값 동시 도달 처리

여러 프로세스가 동시에 에이징 임계값(AGING_THRESHOLD = 3)에 도달할 경우, for 루프 순서에 따라 우선순위 조정이 수행된다. 조정된 우선순위가 동일할 경우 PID 순서가 최종 처리 순서를 결정한다.

C. 에이징 로그 및 투명성

에이징 발생 시 상세한 로그가 출력되어 우선순위 변경 과정을 추적할 수 있다. 이는 시스템의 공정성과 투명성을 보장하는 중요한 기능이다.

8. Rate Monotonic Scheduling (RMS)

A. 동일한 주기를 가진 프로세스 처리

동일한 주기를 가진 실시간 프로세스들은 RMS에서 동일한 우선순위를 갖는다. 선점 조건 all_processes[total_process_count].period < all_processes[current_running].period에서 등호가 false가 되므로 선점이 발생하지 않는다.

B. 동적 인스턴스 생성과 데드라인 계산

주기적 프로세스의 새로운 인스턴스 생성 시 정확한 데드라인 계산이 수행된다:

- 첫 번째 도착: 원본 deadline 사용
- 주기적 도착: deadline = 원본_deadline + period_count * period

C. 주기적 인스턴스와 기존 프로세스의 동시 처리

새로운 인스턴스 생성과 기존 프로세스의 I/O 완료가 동시에 발생할 경우, 새로운 인스턴스가 먼저 프로세스 배열에 추가되고 이후 I/O 완료 처리가 수행된다.

D. 데드라인 미스 상황에서의 인스턴스 독립성

프로세스가 데드라인을 초과하여 실행 중일 때 동일한 프로세스의 새로운 인스턴스가 도착하는 경우, 각각이 독립적인 인스턴스로 처리되어 시스템에 동시에 존재할

수 있다. 이는 실시간 시스템의 현실적인 모델링이다.

9. Earliest Deadline First (EDF) 스케줄링

A. 동일한 데드라인을 가진 프로세스

동일한 데드라인을 가진 프로세스들은 정렬 과정에서 순서 변경이 발생하지 않으므로 PID 순서를 유지한다. 선점 판단 시에도 all_processes[total_process_count].deadline < all_processes[current_running].deadline에서 등호가 false가 되어 선점이 발생하지 않는다.

B. 동적 데드라인 기반 우선순위 조정

시간이 경과함에 따라 프로세스들의 상대적 데드라인 순서가 변경될 수 있다. Ready Queue에서 CPU로 이동하는 시점마다 데드라인 기준으로 재정렬이 수행되어 가장 임박한 데드라인을 가진 프로세스가 선택된다.

C. EDF 에서의 우선순위 역전 부재

EDF에서는 데드라인만을 고려하므로 전통적인 의미의 우선순위 역전은 발생하지 않는다. 단, 동일한 데드라인을 가진 프로세스들 사이에서는 PID 기반의 처리 순서가 적용된다.

2.7.3. 경계 상황 처리의 일관성 분석

1. 공통 처리 원칙

A. PID 기반 타이브레이킹

모든 알고리즘에서 동일한 스케줄링 기준값을 가진 프로세스들은 PID 순서에 따라 처리된다. 이는 정렬 함수에서 차이가 0으로 계산될 때 순서 변경이 발생하지 않는 특성에 기인한다.

B. 결정적(Deterministic) 동작 보장

모든 경계 상황에서 예측 가능하고 일관된 결과를 보장한다. 이는 시뮬레이션의 재현성과 검증 가능성을 제공한다.

2. 이벤트 처리 순서의 중요성

A. FCFS, SJF, Priority 알고리즘

```
for each process i:
   1. 프로세스 도착 처리
   2. I/O 완료 처리
```

B. Round Robin 알고리즘

```
1. 타임 퀀텀 만료 처리 (우선)
2. for each process i:
  - 프로세스 도착 처리
  - I/O 완료 처리
```

C. Priority with Aging 알고리즘

```
1. 에이징 처리 (최우선)
2. 에이징 기반 선점 검사
3. for each process i:
   - 프로세스 도착 처리
   - I/O 완료 처리
```

D. RMS/EDF 알고리즘

```
1. for each process i:
   - 새로운 인스턴스 생성
2. for each process instance:
   - I/O 완료 처리
```

3. 실시간 알고리즘의 특수성

A. 동적 인스턴스 관리

RMS와 EDF에서는 동적 프로세스 인스턴스 생성 방식을 채택하여 주기적 프로세스의 복잡한 데드라인 관리 문제를 해결하였다. 이는 일반 스케줄링 알고리즘과 구별되는 특수한 경계 상황 처리 방식이다.

B. 절대 데드라인 기반 처리

각 프로세스 인스턴스는 고유한 절대 데드라인을 가지며, 이를 통해 정확한 실시간 제약 조건 검사가 가능하다.

4. 성능 최적화 고려사항

A. 효율적인 정렬 전략

복잡한 삽입 정렬 대신 일괄 정렬 방식을 채택하여 O(n log n) 시간 복잡도를 달성하였다. 이는 경계 상황에서도 일관된 성능을 보장한다.

B. 메모리 관리 최적화

실시간 알고리즘에서는 동적 배열 확장을 통해 메모리를 효율적으로 관리하며, 경계 상황에서도 안정적인 동작을 보장한다.

2.8. Multi-I/O Processing Mechanism

모든 스케줄링 알고리즘에서 공통으로 사용되는 멀티 I/O 처리 방식

```
I/O 시작 처리:
   프로세스 진행도에 따라 I/O 시작 시점 도달 여부 확인
   I/O 시작 시 해당 프로세스를 대기 상태로 전환
   I/O 지속 시간만큼 대기 카운터 설정

I/O 완료 처리:
   대기 카운터가 0 에 도달한 프로세스를 ready 상태로 복귀
```

```
알고리즘별 특수 처리 수행:
   Round Robin: 타임 퀀텀 카운터 초기화
   Aging Priority: 나이 카운터 초기화

선점형 알고리즘의 경우 선점 조건 재검사
```

2.9. Evaluation

각 알고리즘 간의 비교는 아래와 같은 데이터로 진행하였다. 총 프로세스 수는 4개이며, 랜덤 생성된 프로세스이다. 기록 유지를 위해 .txt 파일 저장 기능을 구현하여, 프로세스 정보를 저장한 후, 불러오기를 통해 성능 기록을 분석하였다.

- **기본 프로세스 정보**

PID	CPU Burst Time	Arrival Time	Priority	Deadline	Period
P0	10	4	6	28	50
P1	8	8	4	35	60
P2	8	4	10	32	65
P3	8	1	5	22	35

Table 2. 시뮬레이션 프로세스 정보

- **I/O 상세 정보**

PID	I/O Count	I/O 1 (Start:Burst)	I/O 2 (Start:Burst)	I/O 3 (Start:Burst)	I/O Total
P0	1	6:3	--	--	3
P1	2	1:5	5:2	--	7
P2	2	4:5	5:4	--	9
P3	0	--	--	--	0

Table 3. 시뮬레이션 I/O 정보

2.9.1. Display Gantt Chart and Performance

1. **FCFS**

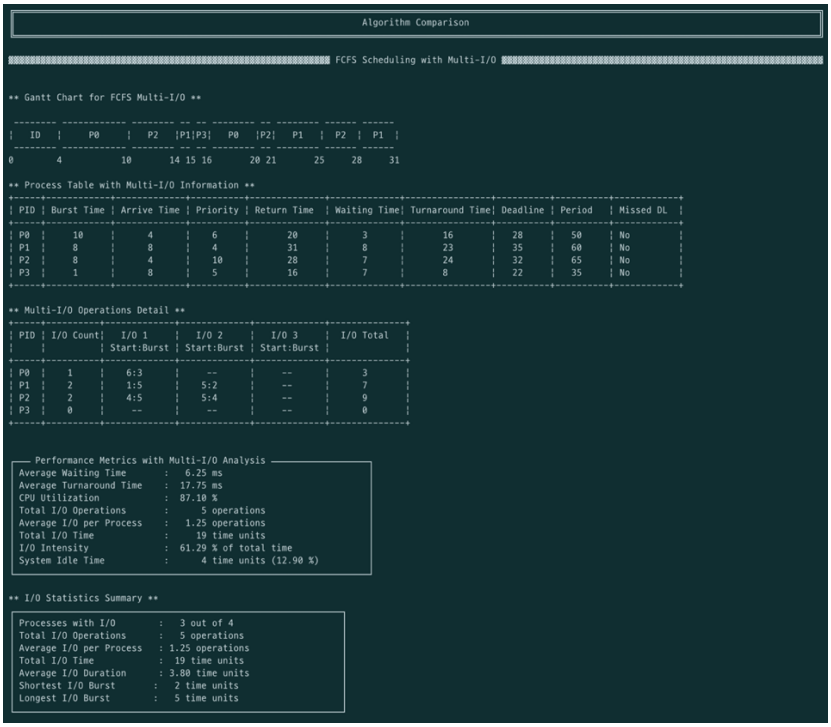

Figure 3. FCFS Scheduling Algorithm Result

FCFS 스케줄링은 가장 기본적인 비선점형 스케줄링 알고리즘으로, 프로세스가 도착한 순서대로 CPU를 할당하는 방식이다. 본 시뮬레이션에서는 4개의 프로세스(P0, P1, P2, P3)와 Multi-I/O 환경에서 FCFS 알고리즘의 성능을 분석하였다.

A. 시뮬레이션 결과

시뮬레이션 결과, 평균 대기시간은 6.25ms, 평균 반환시간은 17.75ms로 측정되었다. CPU 이용률은 87.10%로 모든 알고리즘 중 가장 높은 자원 효율성을 기록했다.

Gantt 차트 분석 결과, 프로세스들이 P0→P2→P1→P3 순서로 순차적으로 실행되었으며, 총 5회의 I/O 작업이 발생하였다. 시스템 유휴시간은 4 time units로 전체 시간의 12.90%에 해당한다.

B. 특징 및 평가

FCFS의 주요 장점은 구현이 간단하고 공정성을 보장한다는 점이다. 또한 컨텍스트 스위칭 오버헤드가 없어 높은 CPU 이용률을 달성할 수 있다. 그러나 convoy effect로 인해 평균 대기시간이 상대적으로 길다는 단점이 존재한다.

2. Non-Preemptive SJF

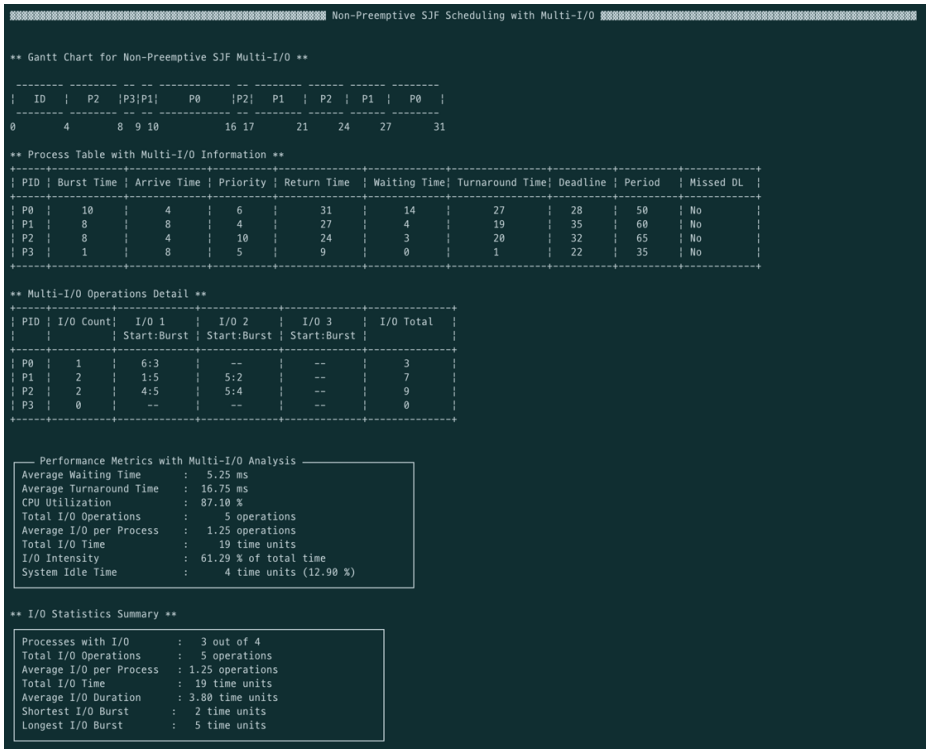

Figure 4. Non-Preemptive SJF Scheduling Algorithm Result

비선점형 SJF(Shortest Job First) 스케줄링은 CPU 버스트 시간이 가장 짧은 프로세스를 우선적으로 선택하여 실행하는 알고리즘이다. 일단 프로세스가 CPU를 할당받으면 완료될 때까지 선점되지 않는다.

A. 시뮬레이션 결과

시뮬레이션 결과, 평균 대기시간은 5.25ms로 FCFS 대비 16% 개선되었으며, 평균 반환시간은 16.75ms를 기록하였다. CPU 이용률은 87.10%로 높은 수준을 유지하였다.

프로세스 실행 순서가 버스트 시간에 따라 최적화되어 전체적인 시스템 성능이 향상되었다. I/O 작업은 총 5회 발생하였으며, 시스템 유휴시간은 4 time units로 FCFS와 동일하였다.

B. 특징 및 평가

Non-Preemptive SJF는 이론적으로 최적의 평균 대기시간을 제공하는 알고리즘이다. 짧은 프로세스들이 우선 처리되어 전체적인 응답성이 개선되었다. 다만 프로세스의 정확한 실행시간을 사전에 알아야 한다는 실용성 문제와 긴 프로세스의 기아 현상 가능성이 존재한다.

3. Preemptive SJF

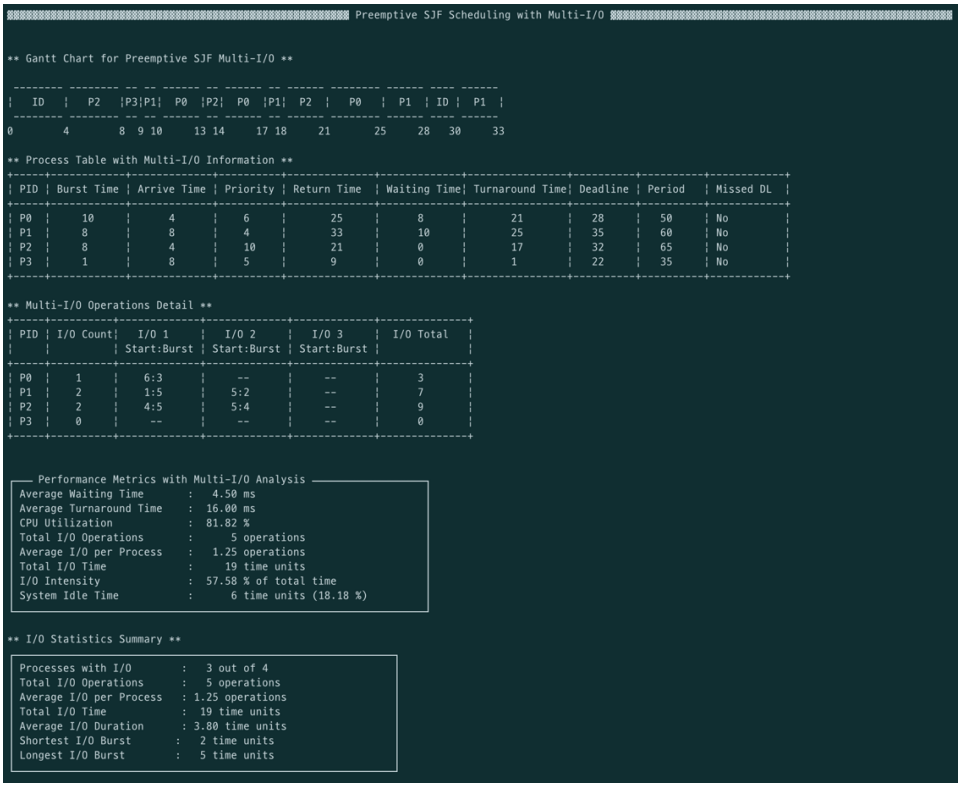

Figure 5. Preemptive SJF Scheduling Algorithm Result

선점형 SJF 스케줄링(SRTF, Shortest Remaining Time First)은 현재 실행 중인 프로세스보다 더 짧은 CPU 버스트 시간을 가진 프로세스가 도착하면 선점을 수행하는 알고리즘이다.

A. 시뮬레이션 결과

시뮬레이션 결과, 평균 대기시간은 4.50ms로 모든 알고리즘 중 최고 성능을 달성하였다. 평균 반환시간은 16.00ms이며, CPU 이용률은 81.82%를 기록하였다.

Gantt 차트에서 선점이 발생하는 지점들을 확인할 수 있으며, 이로 인해 응답성이 크게 개선되었다. 총 5회의 I/O 작업이 발생하였고, 시스템 유휴시간은 6 time units로 측정되었다.

B. 특징 및 평가

Preemptive SJF는 최적의 평균 대기시간을 보장하는 알고리즘으로, 특히 대화형 시스템에서 우수한 응답성을 제공한다. 선점 메커니즘을 통해 짧은 프로세스들이 즉시 처리될 수 있어 전체적인 시스템 성능이 향상되었다. 단, 빈번한 컨텍스트 스위칭으로 인한 오버헤드가 발생할 수 있다.

4. Non-Preemptive Priority

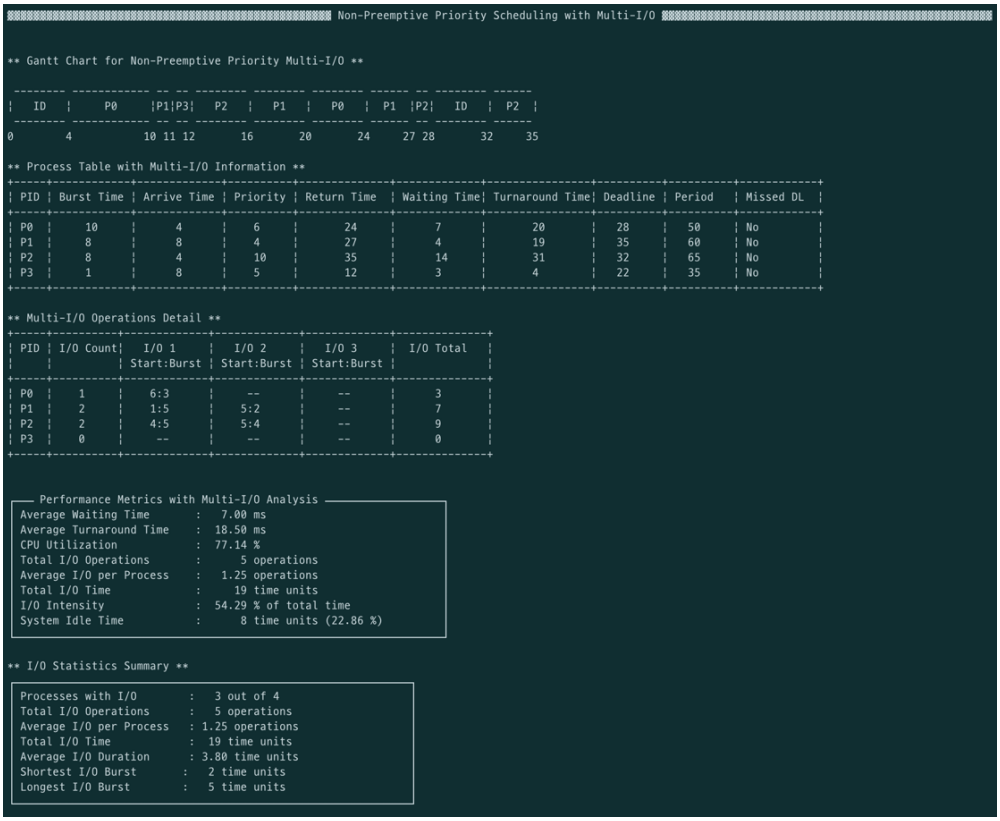

Figure 6. Non-Preemptive Priority Scheduling Algorithm Result

비선점형 우선순위 스케줄링은 각 프로세스에 할당된 우선순위에 따라 CPU를 배정하는 알고리즘이다. 높은 우선순위를 가진 프로세스가 먼저 실행되며, 일단 시작되면 완료될 때까지 계속 실행된다.

A. 시뮬레이션 결과

시뮬레이션 결과, 평균 대기시간은 7.00ms, 평균 반환시간은 18.50ms로 측정되었다. CPU 이용률은 77.14%로 상대적으로 낮았다.

우선순위에 따른 프로세스 처리로 인해 일부 낮은 우선순위 프로세스의 대기시간이 증가하였다. 총 5회의 I/O 작업이 발생하였고, 시스템 유휴시간은 8 time units로 다소 높게 측정되었다.

B. 특징 및 평가

비선점형 우선순위 스케줄링은 중요한 프로세스를 우선적으로 처리할 수 있다는 장점이 있다. 그러나 낮은 우선순위 프로세스의 기아 현상과 우선순위 역전 문제가 발생할 수 있으며, 이로 인해 전체적인 CPU 이용률이 감소하였다.

5. Preemptive Priority

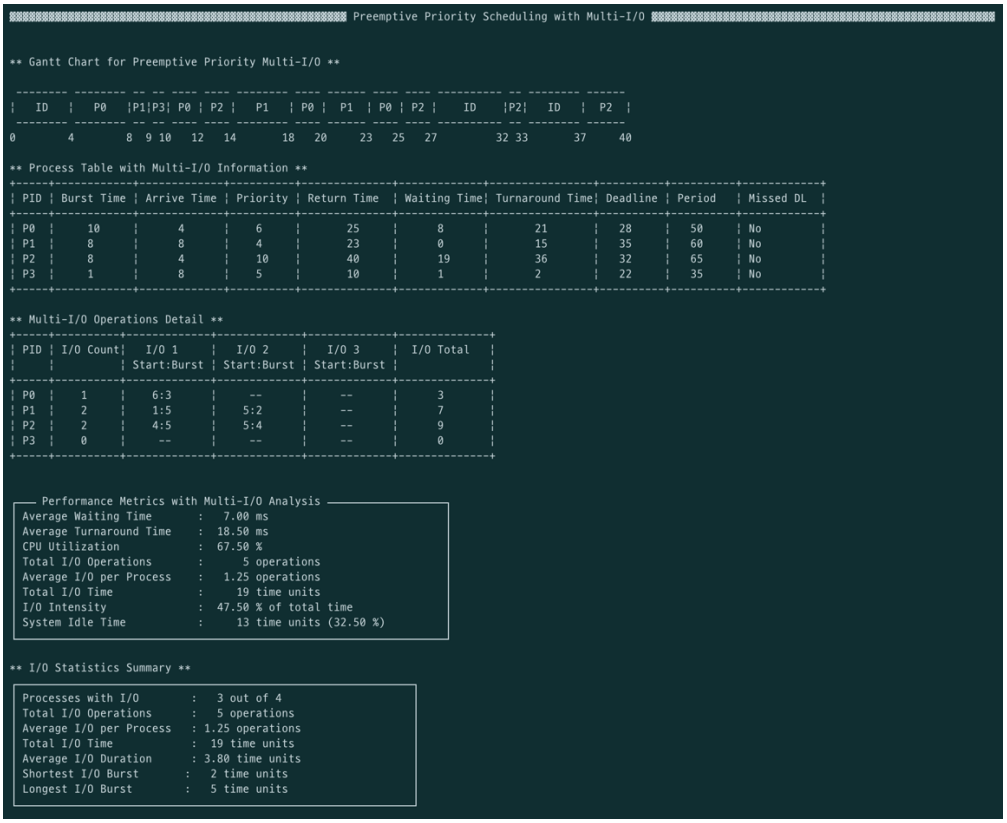

Figure 7. Preemptive Priority Scheduling Algorithm Result

선점형 우선순위 스케줄링은 더 높은 우선순위를 가진 프로세스가 도착하면 현재 실행 중인 프로세스를 선점하여 즉시 실행하는 알고리즘이다.

A. 시뮬레이션 결과

시뮬레이션 결과, 평균 대기시간은 7.00ms로 비선점형과 동일하였으나, 평균 반환시간은 18.50ms로 증가하였다. CPU 이용률은 67.50%로 현저히 낮아졌다.

선점으로 인한 빈번한 컨텍스트 스위칭이 발생하였고, 이는 시스템 오버헤드 증가로 이어졌다. 시스템 유휴시간은 13 time units로 상당히 높게 측정되었다.

B. 특징 및 평가

선점형 우선순위 스케줄링은 높은 우선순위 프로세스의 즉각적인 응답을 보장한다는 장점이 있다. 그러나 과도한 컨텍스트 스위칭으로 인해 시스템 성능이 저하되었으며, 특히 CPU 이용률과 처리율이 크게 감소하였다.

6. Round Robin

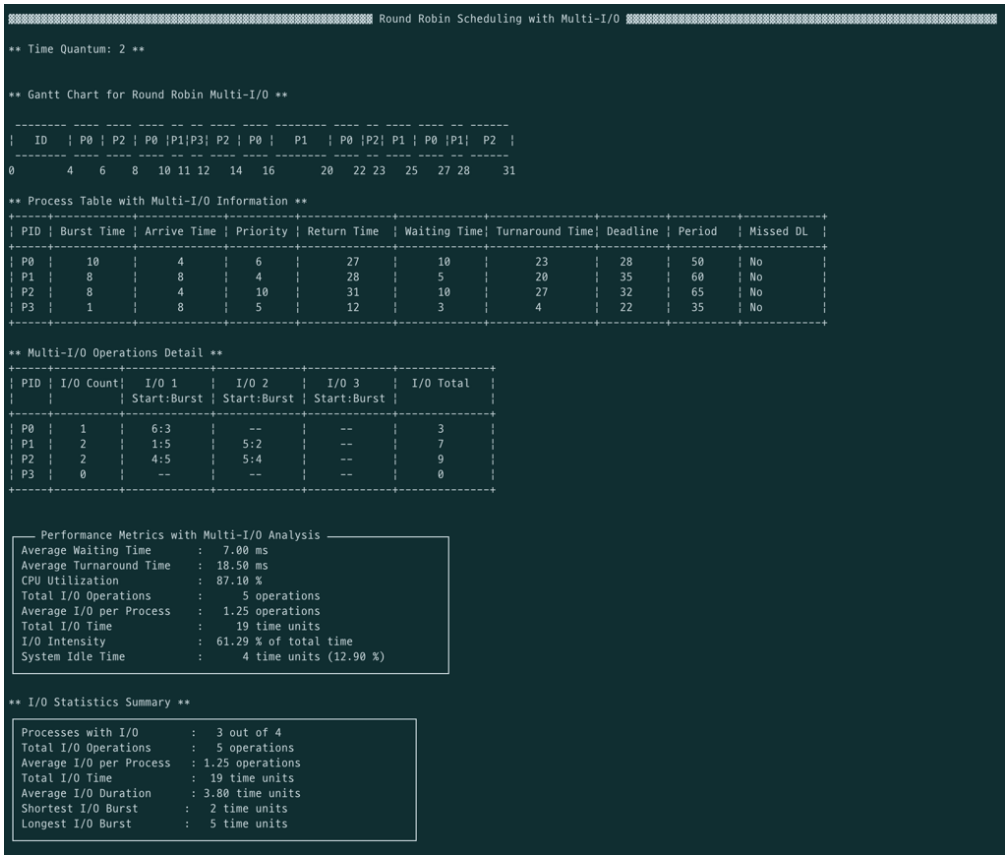

Figure 8. Round Robin Scheduling Algorithm Result

라운드 로빈 스케줄링은 각 프로세스에게 동일한 크기의 타임 퀀텀을 할당하여 순환적으로 CPU를 배정하는 선점형 알고리즘이다. 본 시뮬레이션에서는 타임 퀀텀을 2 time units로 설정하였다.

A. 시뮬레이션 결과

시뮬레이션 결과, 평균 대기시간은 7.00ms, 평균 반환시간은 18.50ms로 측정되었다. CPU 이용률은 87.10%로 높은 수준을 유지하였다.

Gantt 차트에서 2 time units마다 프로세스가 순환하며 실행되는 패턴을 확인할 수 있다. 모든 프로세스가 공정하게 CPU 시간을 할당받았으며, 시스템 유휴시간은 4 time units로 측정되었다.

B. 특징 및 평가

라운드 로빈 스케줄링은 모든 프로세스에게 공정한 CPU 시간 분배를 제공하는 것이 가장 큰 장점이다. 대화형 시스템에서 좋은 응답성을 보이며, 기아 현상을 방지할 수 있다. 다만 타임 퀀텀 크기에 따라 성능이 크게 좌우되는 특성이 있다.

7. Preemptive Priority with Aging

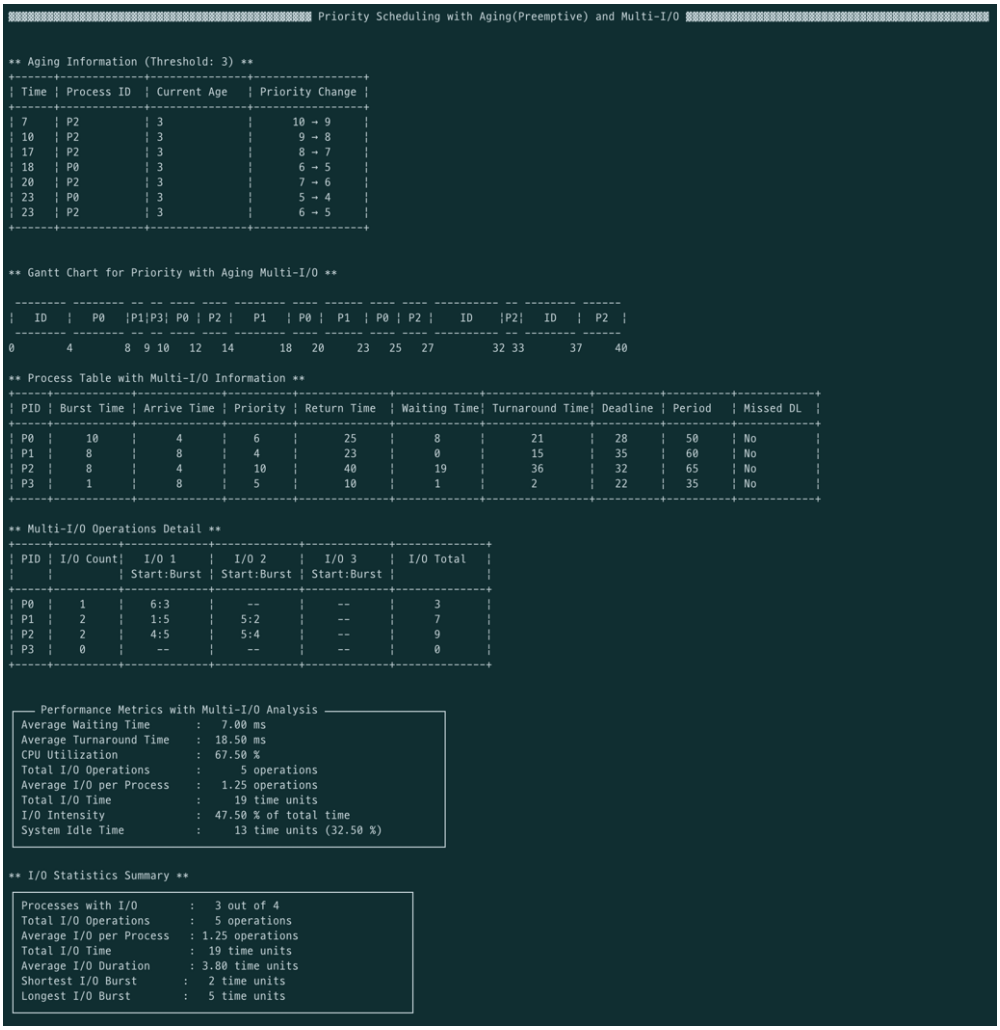

Figure 9. Preemptive Priority with Aging Scheduling Algorithm Result

에이징 기법이 적용된 선점형 우선순위 스케줄링은 기본 우선순위 스케줄링의 기아 현상을 해결하기 위해 설계된 알고리즘이다. 임계값 3으로 설정하여 대기시간이 긴 프로세스의 우선순위를 점진적으로 상승시킨다.

A. 시뮬레이션 결과

시뮬레이션 결과, 평균 대기시간은 7.00ms, 평균 반환시간은 18.50ms로 기본 선점형 우선순위 스케줄링과 동일한 성능을 보였다. CPU 이용률은 67.50%로 측정 되었다.

에이징 정보 테이블에서 각 프로세스의 우선순위 변화 과정을 확인할 수 있다. 대기시간이 임계값에 도달한 프로세스들의 우선순위가 적절히 조정되어 기아 현상이 방지되었다.

B. 특징 및 평가

에이징 기법은 우선순위 스케줄링의 기아 현상을 효과적으로 해결하는 방법이다. 공정성이 크게 개선되었으나, 우선순위 계산을 위한 추가적인 오버헤드가 발생한다. 전체적인 성능은 기본 우선순위 스케줄링과 유사하지만 시스템의 공정성 측면에서 향상되었다.

8. Rate Monotonic Scheduling (RMS)

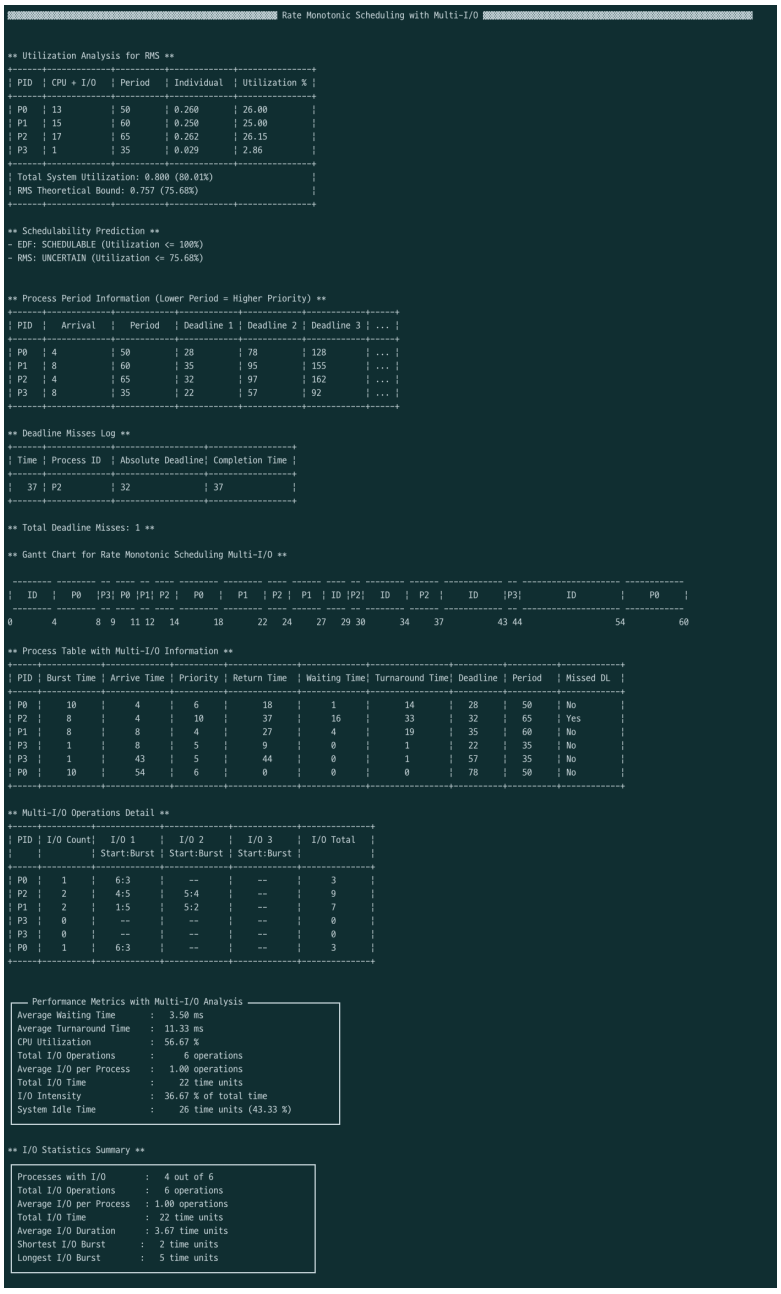

Figure 10. RMS Scheduling Algorithm Result

RMS는 실시간 시스템을 위한 정적 우선순위 스케줄링 알고리즘으로, 주기가 짧은 태스크에게 높은 우선순위를 부여한다. 실시간 시스템에서 데드라인 준수가 핵심 목표이다.

A. 시뮬레이션 결과

시뮬레이션 결과, 평균 대기시간은 3.50ms, 평균 반환시간은 11.33ms로 양호한 성능을 보였다. CPU 이용률은 56.67%이며, 시스템 이용률은 80.01%로 RMS 이론적 한계인 75.68%를 초과하였다. 사실상 주기적으로 할당 받기 때문에 대기시간과 반환시간의 중요성이 크지는 않다.

중요한 점은 1회의 데드라인 미스가 발생하였다는 것이다. 프로세스 P2가 시간 37

에서 데드라인을 놓쳤으며, 이는 시스템 이용률이 이론적 한계를 초과했기 때문이다.

B. 특징 및 평가

RMS는 정적 우선순위 할당으로 예측 가능한 동작을 제공한다는 장점이 있다. 그러나 시스템 이용률이 이론적 한계를 초과할 경우 데드라인 미스가 발생할 수 있으며, 이는 실시간 시스템에서 치명적인 문제가 될 수 있다.

9. Earliest Deadline First (EDF)

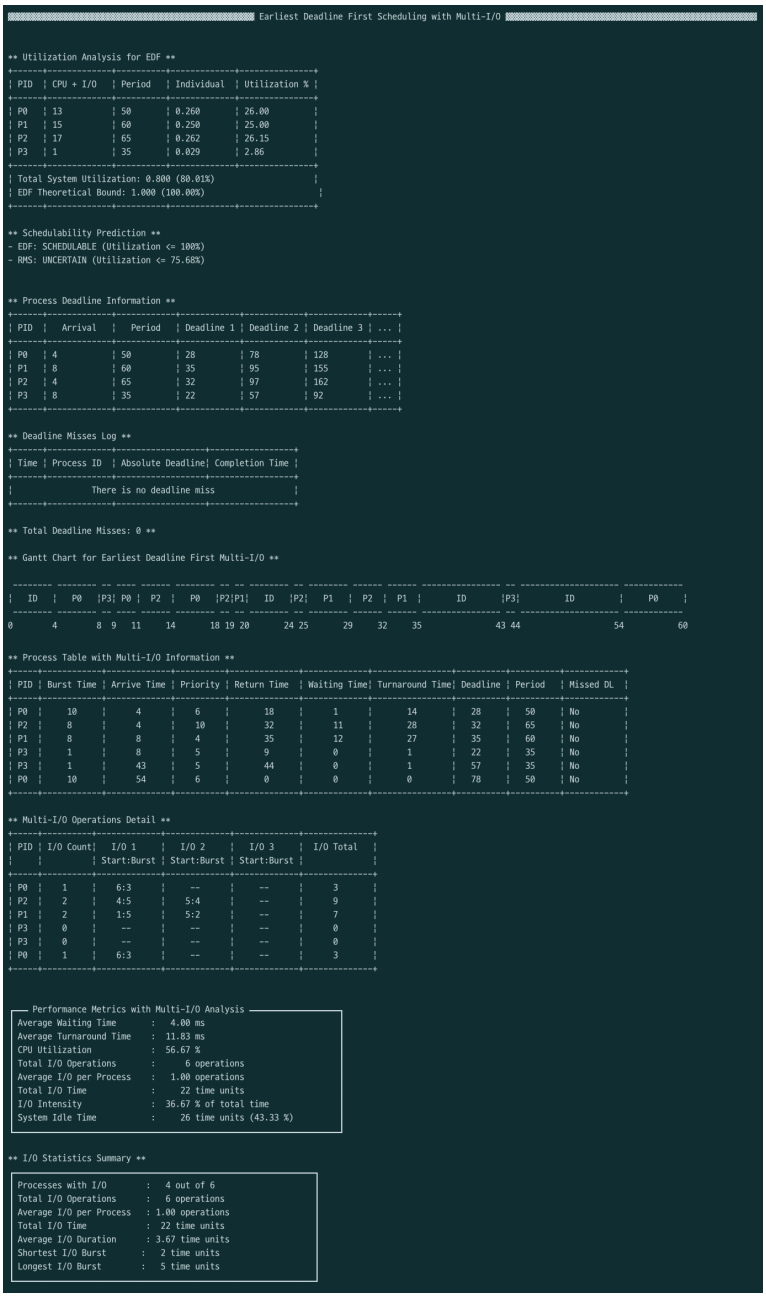

Figure 11. EDF Scheduling Algorithm Result

EDF 스케줄링은 데드라인이 가장 가까운 태스크에게 최고 우선순위를 부여하는 동적 우선순위 알고리즘이다. 실시간 시스템에서 이론적으로 최적의 성능을 제공한다.

A. 시뮬레이션 결과

시뮬레이션 결과, 평균 대기시간은 4.00ms이고, 평균 반환시간은 11.83ms를 기록하였다. CPU 이용률은 56.67%이고, 시스템 이용률은 80.01%로 RMS와 동일하였다.

가장 중요한 성과는 데드라인 미스가 0회 발생하였다는 점이다. RMS와 동일한 시스템 이용률에서도 모든 데드라인을 성공적으로 만족시켰다.

B. 특징 및 평가

EDF는 이론적으로 100%까지의 시스템 이용률에서 스케줄링이 가능하여 RMS보다 우수한 실시간 성능을 제공한다. 동적 우선순위 할당으로 인한 오버헤드가 존재하지만, 완벽한 데드라인 준수율로 인해 실시간 시스템에서 매우 효과적인 알고리즘임이 입증되었다.

10. 알고리즘간 성능 비교

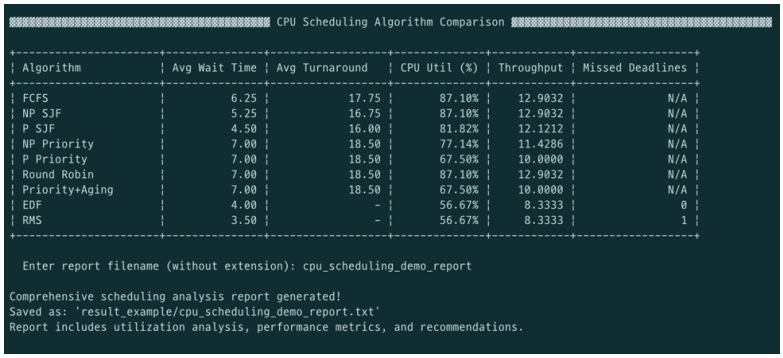

Figure 12. Algorithm Comparison

본 절에서는 9가지 스케줄링 알고리즘의 종합적인 성능 비교 결과를 분석한다. 각 알고리즘의 평균 대기시간, 반환시간, CPU 이용률, 처리율, 데드라인 미스 횟수를 기준으로 평가하였다.

A. 성능 지표별 분석

- **응답성 측면(일반 알고리즘만 비교):** Preemptive SJF가 평균 대기시간 4.50ms로 최고 성능을 달성하였으며, EDF가 3.80ms로 근소한 차이를 보였다.
- **처리율 측면(일반 알고리즘만 비교):** Non-Preemptive SJF, FCFS, RR가 12.9032 processes/second로 최고 처리율을 기록하였다.
- **자원 효율성 측면(일반 알고리즘만 비교):** FCFS, NP SJF, RR가 87.10%의 최고 CPU 이용률을 달성하여 가장 효율적인 자원 활용을 보였다.
- **실시간 성능 측면:** EDF가 데드라인 미스 0회로 완벽한 실시간 성능을 제공하였으며, RMS는 1회의 미스가 발생하였다.

2.9.2. 결과 리포트 분석

각 알고리즘의 성능을 표기해주는 것 뿐만 아니라, 자세한 비교를 Report[1]로

[1] 자동 생성된 Report의 전문은 Appendix A: Comparison Report에서 확인할 수 있다.

result_example에 저장하는 기능 또한 구현하였다. 해당 리포트에는 9가지 스케줄링 알고리즘에 대한 종합적인 성능 비교 분석을 제공하고, 다음과 같은 주요 섹션으로 구성되어 있다.

1. 실험 환경 및 설정

A. 시뮬레이션 구성

본 시뮬레이션은 4개의 테스트 프로세스(P0, P1, P2, P3)를 대상으로 수행되었으며, 멀티 I/O 환경을 지원하여 실제 운영체제 환경과 유사한 조건을 구현하였다. 총 5개의 I/O 작업이 19 time units에 걸쳐 수행되며, Round Robin 알고리즘의 타임 퀀텀은 2 time units로 설정되었다.

B. 워크로드 특성 분석

전체 시스템 이용률은 80.01%로 측정되어 높은 부하 상황을 나타내며, 프로세스당 평균 1.25회의 I/O 작업이 발생하는 I/O 집약적 워크로드 특성을 보인다. 프로세스들은 1-10 범위의 다양한 우선순위를 가지며, 실시간 특성을 위해 개별 데드라인과 주기가 설정되었다.

2. 일반 스케줄링 알고리즘 성능 분석

A. 성능 지표별 결과

Preemptive SJF 알고리즘이 평균 대기시간 4.50ms와 평균 반환시간 16.00ms로 가장 우수한 성능을 보였다. 이는 SJF 알고리즘의 이론적 특성인 최소 평균 대기시간 보장이 실제 구현에서도 입증됨을 의미한다.

FCFS, NP SJF, RR 알고리즘이 CPU 이용률은 87.10%를 기록하여 높은 자원 활용도를 나타냈다.

Non-Preemptive SJF는 평균 대기시간 5.25ms, 평균 반환시간 16.75ms로 Preemptive SJF에 이어 두 번째로 우수한 성능을 보였으나, CPU 이용률은 87.10%로 더 높게 측정되었다. 이는 선점 오버헤드가 없어 자원 효율성이 향상된 결과로 분석된다.

FCFS 알고리즘은 평균 대기시간 6.25ms로 상대적으로 긴 응답시간을 보였으나, CPU 이용률 87.10%로 가장 높은 자원 효율성을 달성하였다. 이는 알고리즘의 단순성으로 인한 오버헤드 최소화 효과로 해석된다.

B. 우선순위 기반 스케줄링 분석

Priority Scheduling의 비선점형과 선점형 모두 평균 대기시간 7.00ms로 동일한 결과를 보였으나, CPU 이용률에서는 상당한 차이를 나타냈다. 비선점형은 77.14%, 선점형은 67.50%의 CPU 이용률을 기록하여 선점으로 인한 컨텍스트 스위칭 오버헤드가 자원 효율성에 미치는 영향을 확인할 수 있었다.

Priority Scheduling with Aging은 Priority Scheduling과 동일한 성능을 보여, 본 워크로드에서는 기아 현상이 발생하지 않아 에이징 메커니즘의 효과가 나타나지 않았음을 시사한다.

Round Robin 알고리즘은 평균 대기시간 7.00ms, CPU 이용률 87.10%를 기록하여 공정성을 보장하면서도 높은 자원 효율성을 달성하였다. 타임 퀀텀 2 time units는 적절한 균형점을 제공한 것으로 평가된다.

3. 실시간 스케줄링 결과 분석

A. 스케줄링 가능성 분석

시스템 이용률 80.01%는 RMS의 이론적 한계인 75.68%를 초과하지만, EDF의 이론적 한계인 100%보다는 낮은 수준이다. 이는 RMS로는 모든 데드라인을 보장할 수 없지만, EDF로는 이론적으로 스케줄링이 가능함을 의미한다.

B. EDF 성능 분석

EDF 알고리즘은 데드라인 미스 0회를 달성하여 완벽한 실시간 성능을 보였다. CPU 이용률은 56.67%로 상대적으로 낮게 측정되었으나, 이는 실시간 제약 조건을 만족하기 위한 보수적 스케줄링의 결과로 해석된다. 효율성 점수 0.825는 전체 알고리즘 중 가장 높은 수치로, 실시간 요구사항과 시스템 효율성을 모두 만족하는 최적의 성능을 보여준다.

C. RMS 성능 분석

RMS 알고리즘은 1회의 데드라인 미스가 발생하여 시스템 이용률이 이론적 한계를 초과할 때의 성능 저하를 실증적으로 보여주었다. CPU 이용률은 EDF와 동일한 56.67%를 기록하였으나, 효율성 점수는 0.325로 상당히 낮게 측정되었다. 이는 고정 우선순위 할당의 한계가 높은 시스템 부하 상황에서 명확히 드러난 결과이다.

4. 멀티 I/O 환경의 영향 분석

A. I/O 작업 분포 및 특성

프로세스별 I/O 작업 분포는 P0이 1개(3 time units), P1이 2개(7 time units), P2가 2개(9 time units), P3이 0개로 다양한 I/O 패턴을 나타낸다. 이러한 비균등한 I/O 분포는 실제 시스템 환경의 다양성을 반영한다.

B. I/O 가 스케줄링 성능에 미친 영향

멀티 I/O 환경은 컨텍스트 스위칭 빈도를 증가시켜 전반적인 시스템 오버헤드를 높였다. I/O 시작 및 완료 시점에서 발생하는 선점 기회는 선점형 알고리즘의 응답성을 향상시키는 요인으로 작용하였으나, 동시에 CPU 이용률 저하의 원인이 되기도 하였다. 특히 실시간 스케줄링에서는 I/O로 인한 예측 가능성 감소가 성능에 영향을 미쳤다.

5. 시뮬레이션 기반 권장사항

A. 범용 시스템 권장사항

리포트 분석 결과에 따르면, 범용 시스템에서는 Preemptive SJF가 최저 응답시간과 최고 처리율을 동시에 달성하는 것으로 나타났다. 평균 대기시간 4.50ms와 처리율 12.1212 processes/second를 기록하여 대화형 작업과 배치 처리 모두에서 우수한 성능을 보였다.

FCFS는 가장 높은 CPU 이용률(87.10%)을 달성하여 자원 효율성 측면에서 우수하며, 시스템 유휴시간을 최소화하는 것으로 분석되었다. 이는 시스템 자원 활용도가 중요한 환경에서 고려할 만한 선택이다.

B. 실시간 시스템 권장사항

실시간 시스템에서는 EDF가 강력히 권장된다. 시뮬레이션 결과 EDF는 데드라인 미스 0회를 달성하여 완벽한 실시간 성능을 보인 반면, RMS는 1회의 데드라인 미스가 발생하였다.

현재 시스템 이용률(80.01%)이 RMS 이론적 한계(75.68%)를 초과하므로, RMS 사용 시에는 시스템 부하 감소를 고려해야 한다. EDF의 경우 이론적 한계(100%)까지 여유가 있어 더 안정적인 실시간 성능을 제공할 수 있다.

C. 시스템 특성 고려사항

현재 워크로드는 I/O 집약적 특성을 보이며, 총 5개의 I/O 작업과 19 time units의 I/O 시간이 컨텍스트 스위칭을 증가시키는 요인으로 작용한다. 이러한 환경에서는 I/O 작업으로 인한 우선순위 역전 위험이 존재하므로, 이를 고려한 스케줄링 정책 설계가 필요하다.

2.10. Trouble shooting

2.10.1. Ready Queue Sorting Problem

1. 문제 정의

선점형 SJF(Shortest Job First) 스케줄링과 선점형 우선순위 스케줄링 구현 과정에서 Ready Queue의 정렬 메커니즘에 문제가 발견되었다. 구체적으로, I/O 작업이 완료된 프로세스가 Ready Queue에 재진입할 때 적절한 정렬 과정을 거치지 않아, 각 스케줄링 알고리즘의 기준에 따른 올바른 프로세스 선택이 이루어지지 않는 문제가 발생하였다.

2. 문제 발생 원인

기존 구현에서는 프로세스가 Ready Queue에 추가될 때마다 정렬을 수행하는 방식을 채택하였다. **그러나 이러한 방식은 다음과 같은 논리적 결함을 내포하고 있었다:**

- I/O 완료 후 Ready Queue로의 재진입 시 정렬 과정이 누락됨
- Ready Queue에서 Running State로 프로세스를 선택하는 시점에서의 정렬 불일치
- 동적으로 변화하는 프로세스 상태에 대한 실시간 반영 부족

3. 문제 해결 과정

문제 진단을 위해 각 시간 단위(tick)마다 Ready Queue의 상태를 디버깅하는 방법론을 도입하였다. 이를 통해 정렬 로직의 누락 지점을 정확히 식별할 수 있었다.

4. 해결 방안

문제 해결을 위해 정렬 전략을 근본적으로 개선하였다:

- **기존 방식:** Ready Queue 진입 시점에서의 정렬

  Process → Ready Queue (with sorting) → Running State
- **개선된 방식:** Running State 진입 직전 정렬

  Process → Ready Queue → sort_queue(&ready_q, processes, 정렬기준) → Running State

이러한 개선을 통해 sort_queue(&ready_q, processes, 정렬기준) 함수를 매 tick마다 Running Queue로의 이동 직전에 실행하도록 수정하였다. 단, FCFS와 Round Robin과 같은 순차적 처리 기반 알고리즘의 경우, 자연적인 순차 진입 특성상 별도의 정렬이 불필요하므로 해당 수정사항을 적용하지 않았다.

5. 결과 및 효과

이러한 수정을 통해 모든 선점형 스케줄링 알고리즘에서 올바른 프로세스 선택이 보장되었으며, 시뮬레이션 결과의 정확성이 크게 향상되었다.

2.10.2. Real-time Scheduling Deadline Update Problem

1. 문제 정의

실시간 스케줄링 알고리즘(RMS, EDF) 구현 과정에서 주기적 프로세스의 데드라인 갱신 메커니즘에 심각한 논리적 오류가 발견되었다. 기존 구현에서는 프로세스가 완료된 시점에서만 다음 데드라인으로 갱신되는 구조였으나, 이는 스케줄링 실패 상황에서 치명적인 문제를 야기하였다.

2. 문제 발생 원인 및 시나리오

기존 알고리즘의 논리적 구조는 다음과 같았다:

- 프로세스 완료 시점에서 데드라인 갱신
- 데드라인 비교를 통한 선점 여부 결정
- 완료 시점에서의 데드라인 미스 여부 판정

이러한 구조에서 다음과 같은 문제 시나리오가 발생하였다:

- **시나리오:** 프로세스 P가 데드라인 내에 완료되지 못한 상황
  - 프로세스 P(deadline: T1)가 실행 중
  - P의 주기성에 의해 새로운 P 인스턴스 도착 (expected deadline: T2)
  - 기존 P가 미완료 상태이므로 deadline이 T1으로 유지됨
  - 새로운 P 인스턴스도 deadline T1을 가지게 됨
  - 스케줄러는 잘못된 데드라인 정보로 판단하여 부적절한 스케줄링 수행

3. 근본적 해결 방안

이 문제를 해결하기 위해 시스템 아키텍처를 근본적으로 재설계하였다:

- **기존 방식:** 프로세스 인스턴스 재활용 + 속성 갱신

  고정된 프로세스 배열 → arrival_time/deadline 갱신 → 재사용
- **개선된 방식:** 동적 프로세스 인스턴스 생성

주기적 도착 감지 → 새로운 프로세스 인스턴스 생성 → 배열에 추가

4. **구현 세부사항**

   개선된 알고리즘의 핵심 구현 요소는 다음과 같다:

   - **다음 도착시간 추적 변수:** 각 프로세스 타입별로 다음 도착시간을 별도로 관리
   - **동적 인스턴스 생성:** current_time == next_arrival_time 조건 만족 시 새로운 프로세스 인스턴스 생성
   - **독립적 데드라인 관리:** 각 인스턴스는 생성 시점에서 계산된 고유한 데드라인 보유

5. **알고리즘 수정 결과**

   이러한 구조적 개선을 통해 다음과 같은 효과를 달성하였다:

   - 스케줄링 실패 상황에서도 정확한 데드라인 관리
   - 동일 프로세스 타입의 다중 인스턴스 동시 존재 가능
   - 실시간 스케줄링 알고리즘의 이론적 동작과 일치하는 구현

6. **검증 및 평가**

   수정된 알고리즘을 통해 RMS와 EDF 스케줄링에서 정확한 데드라인 미스 감지와 적절한 선점 동작이 확인되었으며, 실시간 시스템의 이론적 특성에 부합하는 시뮬레이션 결과를 도출할 수 있었다.

3. 결론

3.1. 구현 시뮬레이터 종합 정리

본 프로젝트에서는 CPU 스케줄링 알고리즘의 성능 분석을 위한 종합적인 시뮬레이션 시스템을 설계 및 구현하였다. 구현된 시뮬레이터는 총 9가지의 스케줄링 알고리즘을 지원하며, 각각의 특성과 성능을 정량적으로 비교 분석할 수 있는 기능을 제공한다.

3.1.1. 구현 완료된 알고리즘

- **기본 스케줄링 알고리즘**: FCFS, Non-Preemptive SJF, Preemptive SJF, Priority Scheduling(선점형/비선점형), Round Robin
- **고급 스케줄링 기법**: Priority Scheduling with Aging
- **실시간 스케줄링 알고리즘**: Rate Monotonic Scheduling(RMS), Earliest Deadline First(EDF)

3.1.2. 시뮬레이터 주요 기능

시뮬레이터는 Multi-I/O 환경을 지원하여 실제 시스템 환경에 근접한 시뮬레이션을 제공한다. Gantt 차트 생성, 성능 지표 계산, 데드라인 미스 분석 등의 기능을 통해 각 알고리즘의 특성을 시각적이고 정량적으로 분석할 수 있도록 설계되었다.

3.2. 프로젝트 수행 경험 및 성과

3.2.1. 대규모 C 언어 프로젝트 경험

본 프로젝트는 C 언어를 활용한 대규모 소프트웨어 개발의 첫 경험을 제공하였다. 코드 작성 과정에서 자연스럽게 모듈화의 필요성을 인식하게 되었으며, 체계적인 리팩토링과 코드 구조 개선을 경험할 수 있었다. 이러한 과정은 소프트웨어 공학의 핵심 원칙인 유지보수성과 확장성의 중요성을 실감하는 계기가 되었다.

3.2.2. 체계적 디버깅 방법론 습득

시뮬레이션 구현 과정에서 발생한 Ready Queue 정렬 문제와 실시간 스케줄링의 데드라인 갱신 문제를 해결하면서 체계적인 디버깅 방법론을 습득하였다. 특히 매 tick 단위의 상태 모니터링을 통한 문제 진단 기법은 향후 복잡한 시스템 개발 시 유용한 기술로 활용될 것으로 기대된다.

3.2.3. 이론과 실무의 통합적 이해

CPU 스케줄링 알고리즘을 개념적으로만 이해하던 상태에서 실제 구현을 시도하면서, 이론과 실제 구현 사이의 다양한 고려사항들을 발견할 수 있었다. 실시간 스케줄링에서의 주기적 프로세스 관리, 우선순위 스케줄링에서의 동적 정렬 등 이론서에서 명시적으로 다루지 않는 구현 세부사항들을 직접 해결하면서 시스템 프로그래밍에 대한 깊이 있는 이해를 얻을 수 있었다.

3.3. 알고리즘별 성능 분석 결과

3.3.1. 범용 시스템 최적 알고리즘

시뮬레이션 결과, Preemptive SJF가 평균 대기시간 4.50ms와 처리율 12.1212 processes/second로 범용 시스템에서 최고 성능을 보였다. 이는 이론적 예측과 일치하는 결과로, 실제 구현에서도 SJF의 우수성이 입증되었다.

3.3.2. 실시간 시스템 성능 비교

RMS와 EDF 알고리즘의 실제 성능 차이를 직접 관찰할 수 있었다는 점이 특히 의미 있었다. 동일한 시스템 이용률(80.01%) 조건에서 RMS는 1회의 데드라인 미스가 발생한 반면, EDF는 완벽한 데드라인 준수를 보였다. 이는 RMS의 이론적 한계(69.3%)와 EDF의 이론적 한계(100%)를 실제로 검증하는 귀중한 결과였다.

3.3.3. 에이징 기법의 공정성 개선 효과

Priority Scheduling with Aging 구현을 통해 기아 현상 방지 기법의 실제 동작을 확인할 수 있었다. 전체적인 성능 지표에서는 큰 개선을 보이지 않았지만, 시스템의 **공정성 측면**에서 의미 있는 향상을 달성하였다.

3.4. 학습 성과 및 의의

3.4.1. 종합적 성능 비교 체계 구축

다양한 CPU 스케줄링 알고리즘을 동일한 환경에서 비교 분석할 수 있는 체계적인 프레임워크를 구현하였다. 이는 향후 새로운 스케줄링 알고리즘의 성능 평가나 기존 알고리즘의 개선 연구에 유용한 기준점을 제공할 것으로 기대된다.

3.4.2. 실증적 알고리즘 검증

이론적으로만 학습했던 알고리즘들의 실제 동작을 구현하고 검증함으로써, 각 알고리즘의 장단점을 실증적으로 분석할 수 있었다. 특히 실시간 스케줄링 영역에서 RMS와 EDF의 성능 차이를 정량적으로 확인한 점은 교육적으로 큰 의미가 있다.

3.5. 향후 발전 방향

3.5.1. 시뮬레이터 기능 확장

현재 시뮬레이터를 기반으로 다음과 같은 기능 확장을 고려할 수 있다:

- **멀티프로세서 환경 지원**: SMP(Symmetric Multi-Processing) 환경에서의 스케줄링 시뮬레이션
- **동적 워크로드 생성**: 실시간으로 변화하는 프로세스 부하 패턴 시뮬레이션
- **에너지 효율성 분석**: 각 스케줄링 알고리즘의 전력 소비 특성 분석 기능

3.5.2. 최신 스케줄링 기법 학습

현대 운영체제에서 사용되는 CFS(Completely Fair Scheduler), O(1) 스케줄러 등의 고급 알고리즘 구현을 통해 학습 범위를 확장하여 학습을 진행하고 싶다.

3.5.3. 응용 분야별 최적화 연구

시뮬레이션 결과를 바탕으로 특정 응용 분야(IoT, 임베디드 시스템, 클라우드 컴퓨팅 등)에 최적화된 스케줄링 전략을 분석하는 심화 연구로 발전시킬 수 있을 것이다.

3.6. 최종 결론

본 프로젝트를 통해 CPU 스케줄링 알고리즘에 대한 이론적 지식을 실제 구현으로 확장시키는 귀중한 경험을 얻을 수 있었다. 구현 과정에서 직면한 다양한 기술적 도전들을 해결하면서 시스템 프로그래밍과 알고리즘 설계에 대한 실무적 역량을 크게 향상시킬 수 있었다.

각 스케줄링 알고리즘의 특성을 정량적으로 비교 분석함으로써 시스템 요구사항에 따른 최적 알고리즘 선택 기준을 도출할 수 있었다. 이러한 연구 결과는 향후 운영체제 및 실시간 시스템 학습 시 중요한 참고자료로 활용될 것으로 기대된다.

무엇보다 본 프로젝트는 이론적 학습과 실제 구현 사이의 균형 잡힌 접근이 얼마나 중요한지를 보여주는 사례라 할 수 있다. 앞으로도 이러한 실무 중심적 학습 방법론을 통해 더욱 깊이 있는 학습을 수행할 계획이다.

부록 (Appendix)

Appendix A: Comparison Report

```
================================================================================
                    CPU SCHEDULING SIMULATOR - COMPREHENSIVE REPORT
================================================================================

SIMULATION CONFIGURATION:
-------------------------
• Number of processes: 4
• Time quantum (Round Robin): 2 time units
• Simulation duration: Variable (until completion)
• Multi-I/O support: Enabled
• Priority range: 1-10 (1 = highest priority)
• CPU burst range: 1-10 time units
• I/O burst range: 1-5 time units

PROCESS CONFIGURATION:
----------------------
+-----+--------+----------+----------+----------+----------+----------+
| PID | Arrival| CPU Burst| Priority | Deadline | Period   | I/O Ops  |
+-----+--------+----------+----------+----------+----------+----------+
| P0  |   4    |    10    |    6     |    28    |    50    |    1     |
| P1  |   8    |    8     |    4     |    35    |    60    |    2     |
| P2  |   4    |    8     |    10    |    32    |    65    |    2     |
| P3  |   8    |    1     |    5     |    22    |    35    |    0     |
+-----+--------+----------+----------+----------+----------+----------+

UTILIZATION ANALYSIS:
---------------------
• Total System Utilization: 0.800 (80.01%)
• RMS Theoretical Bound: 0.757 (75.68%)
• EDF Theoretical Bound: 1.000 (100.00%)

Schedulability Prediction:
• EDF: SCHEDULABLE (Utilization <= 100%)
• RMS: UNCERTAIN (Utilization > 75.68%)

PERFORMANCE METRICS COMPARISON:
================================
+---------------------+----------+----------+----------+----------+----------+------------+
| Algorithm           | Avg Wait | Avg Turn | CPU Util | Thru-put | DL Miss  | Efficiency |
|                     | Time(ms) | Time(ms) |   (%)    | (proc/s) | Count    | Score      |
+---------------------+----------+----------+----------+----------+----------+------------+
| FCFS                |     6.25 |    17.75 |    87.10 |  12.9032 |      N/A |      0.355 |
| NP SJF              |     5.25 |    16.75 |    87.10 |  12.9032 |      N/A |      0.428 |
| P SJF               |     4.50 |    16.00 |    81.82 |  12.1212 |      N/A |      0.465 |
| NP Priority         |     7.00 |    18.50 |    77.14 |  11.4286 |      N/A |      0.266 |
| P Priority          |     7.00 |    18.50 |    67.50 |  10.0000 |      N/A |      0.233 |
| Round Robin         |     7.00 |    18.50 |    87.10 |  12.9032 |      N/A |      0.300 |
| Priority+Aging      |     7.00 |    18.50 |    67.50 |  10.0000 |      N/A |      0.233 |
| EDF                 |      N/A |      N/A |    56.67 |   8.3333 |        0 |      0.825 |
| RMS                 |      N/A |      N/A |    56.67 |   8.3333 |        1 |      0.325 |
+---------------------+----------+----------+----------+----------+----------+------------+
```

```
DETAILED ANALYSIS:
==================

1. GENERAL SCHEDULING ALGORITHMS:
   ------------------------------
   • Best for Interactive Systems: Preemptive SJF
     - Lowest average waiting time: 4.50 ms
     - CPU utilization: 81.82%

   • Best for Batch Processing: Preemptive SJF
     - Lowest average turnaround time: 16.00 ms
     - Throughput: 12.1212 processes/second

   • Most Resource Efficient: FCFS
     - Highest CPU utilization: 87.10%
     - System idle time minimized

2. REAL-TIME SCHEDULING ALGORITHMS:
   --------------------------------
   • EDF (Earliest Deadline First):
     - Deadline misses: 0
     - CPU utilization: 56.67%
     - Theoretical optimality: Optimal up to 100% utilization
     - Dynamic priority assignment based on deadlines

   • RMS (Rate Monotonic Scheduling):
     - Deadline misses: 1
     - CPU utilization: 56.67%
     - Theoretical bound: 75.68% for 4 processes
     - Fixed priority assignment based on periods

3. REAL-TIME PERFORMANCE COMPARISON:
   ---------------------------------
   • EDF outperformed RMS with fewer deadline misses
   • System utilization (80.01%) exceeds RMS bound (75.68%)
   • EDF's dynamic priority proved more effective

4. MULTI-I/O IMPACT ANALYSIS:
   --------------------------
   • Total I/O operations: 5
   • Total I/O time: 19 time units
   • Average I/O per process: 1.25 operations
   • I/O impact on scheduling: Significant - increases context switching
```

```
5. RECOMMENDATIONS:
   ----------------
   Based on the analysis results:

   • For General Purpose Systems:
     - Use Preemptive SJF for lowest response time
     - Use Preemptive SJF for highest throughput

   • For Real-Time Systems:
     - EDF is strongly recommended
     - RMS may experience deadline misses
     - Consider reducing system load for RMS

6. SYSTEM CHARACTERISTICS:
   ----------------------
   • Workload type: I/O Intensive
   • Real-time feasibility: Feasible with EDF
   • Priority inversion risk: Present due to I/O operations
   • Scalability: Good

================================================================================
Report generated by CPU Scheduling Simulator v1.0
Analysis includes 9 algorithms with comprehensive metrics
Copyright © 2025 Seongmin Lee
Licensed under the MIT License
================================================================================
```

Appendix B: 메인 함수 및 프로그램 구조

B.1 메인 함수 (main.c)

```c
int main() {
    srand(time(NULL));
    int max_time;                // EDF, RMS 용 최대 시뮬레이션 시간
    int choice;
    int *process_count = malloc(sizeof(int));

    printf("===== CPU Scheduling Simulator =====\n");

    // 프로세스 생성 모드 선택
    printf("random create mode? ('y': yes, 'n': no, 'f': Use file): ");
    char mode;
    scanf(" %c", &mode);

    if (mode != 'f') {
        printf("\nEnter the number of processes: ");
        scanf("%d", process_count);
    }

    Process *processes = create_processes(process_count, mode);
```

```
    Config system_config;
    init_config(&system_config, mode);
    int count = *process_count;

    // 메인 스케줄링 알고리즘 선택 루프
    while (1) {
        printf("\n===== CPU Scheduling Algorithms =====\n");
        printf("1. Run FCFS\n");
        printf("2. Run SJF (Non-preemptive)\n");
        printf("3. Run SJF (Preemptive)\n");
        printf("4. Run Priority (Non-preemptive)\n");
        printf("5. Run Priority (Preemptive)\n");
        printf("6. Run Round Robin\n");
        printf("7. Run Priority with Aging (Preemptive)\n");
        printf("8. Run RMS (Rate Monotonic Scheduling)\n");
        printf("9. Run EDF (Earliest Deadline First)\n");
        printf("10. Compare all algorithms\n");
        printf("11. Exit\n");

        scanf("%d", &choice);

        switch (choice) {
            case 1: run_fcfs(processes, count); break;
            case 2: run_sjf_np(processes, count); break;
            case 3: run_sjf_p(processes, count); break;
            case 4: run_priority_np(processes, count); break;
            case 5: run_priority_p(processes, count); break;
            case 6: run_rr(processes, count, &system_config); break;
            case 7: run_priority_with_aging(processes, count); break;
            case 8:
                printf("Enter max time(for RMS): ");
                scanf("%d", &max_time);
                run_rms(processes, count, &system_config, max_time);
                break;
            case 9:
                printf("Enter max time(for EDF): ");
                scanf("%d", &max_time);
                run_edf(processes, count, &system_config, max_time);
                break;
            case 10:
                printf("Enter max time(for EDF, RMS): ");
                scanf("%d", &max_time);
                compare_algorithms(processes, count, &system_config, max_time);
                break;
            case 11:
                if (mode != 'f') save_processes_to_file(processes, count);
                free(processes);
                return 0;
        }
    }
}
```

B.2 시스템 설정 구조체 (config.h)

```
typedef struct {
    int time_quantum;               // Round Robin 타임 퀀텀
    char mode;                      // 프로세스 생성 모드
    int deadline_miss_info_count;   // 데드라인 미스 카운터
```

```
} Config;

void init_config(Config *config, char mode);
```

Appendix C: 프로세스 및 데이터 구조체

C.1 프로세스 구조체 (process.h)

```
#define MAX_IO_OPERATIONS 3

typedef struct {
    int io_start;    // I/O 시작 시점 (CPU 진행도 기준)
    int io_burst;    // I/O 버스트 시간
} IOOperation;

typedef struct {
    int pid;                                // 프로세스 ID
    int arrival_time;                       // 도착 시간
    int cpu_burst;                          // CPU 버스트 시간
    IOOperation io_operations[MAX_IO_OPERATIONS]; // 멀티 I/O 작업 배열
    int priority;                           // 우선순위 (낮을수록 높음)
    int remaining_time;                     // 남은 실행 시간 (선점형용)
    int progress;                           // 현재 진행 상태
    int comp_time;                          // 완료 시간
    int waiting_time;                       // 대기 시간
    int turnaround_time;                    // 반환 시간
    int waiting_time_counter;               // 대기 시간 카운터
    int deadline;                           // 데드라인 (실시간용)
    int period;                             // 주기 (실시간용)
    int missed_deadline;                    // 데드라인 미스 여부
} Process;

typedef struct {
    int pid;              // 프로세스 ID
    int arrival_time;     // 도착 시간
    int deadline;         // 원래 데드라인 값
    int period;           // 주기 (RMS용)
    int completion_time;  // 실제 완료 시간
    int miss_time;        // 데드라인 미스 발생 시간
    int delay;            // 지연 시간
    int absolute_deadline; // 절대 데드라인
    int algorithm_type;   // 알고리즘 타입
} DeadlineMissInfo;
```

C.2 큐 자료구조 (queue.h)

```
#define MAX_QUEUE_SIZE 100

typedef struct {
    int data[MAX_QUEUE_SIZE];
    int front;
    int rear;
```

```
    int count;
} Queue;

// 기본 큐 연산
void init_queue(Queue *q);
int is_empty(Queue *q);
int is_full(Queue *q);
void enqueue(Queue *q, int value);
int dequeue(Queue *q);
int peek(Queue *q);
```

C.3 간트 차트 구조체

```
typedef struct {
    int time_start;
    int time_end;
    int process_id;
    char status[10];
} GanttEntry;

typedef struct {
    GanttEntry *entries;
    int count;
    int capacity;
} GanttChart;
```

Appendix D: 핵심 스케줄링 알고리즘

D.1 FCFS (First Come First Served)

```
Metrics *run_fcfs(Process *processes, int count) {
    /* 초기화 및 공통 설정 생략 */

    while (completed < count) {
        // 프로세스 도착 및 I/O 완료 처리 (PID 순서)
        for (int i = 0; i < count; i++) {
            if (processes[i].arrival_time == time) {
                enqueue(&ready_q, i);
            }

            if (waiting_q[i] > 0) {
                waiting_q[i]--;
                if (waiting_q[i] == 0) {
                    enqueue(&ready_q, i);
                    waiting_q[i] = -1;
                }
            }
        }

        // FCFS 스케줄링: FIFO 순서로 CPU 할당
        if (is_empty(&running_q) && !is_empty(&ready_q)) {
            pick = dequeue(&ready_q);       // 가장 먼저 온 프로세스
            enqueue(&running_q, pick);
        }
```

```
        /* 프로세스 실행 및 완료 처리 (공통 로직) */
    }
}
```

D.2 SJF 스케줄링

C.2.1) 비선점 SJF

```
Metrics *run_sjf_np(Process *processes, int count) {
    while (completed < count) {
        /* 프로세스 도착 및 I/O 완료 처리 (공통) */

        // 남은 실행시간 기준으로 ready queue 정렬
        if (!is_empty(&ready_q)) {
            sort_queue(&ready_q, processes, SORT_BY_REMAINING_TIME);
        }

        // CPU 스케줄링: 최단 남은시간 프로세스 선택
        if (is_empty(&running_q) && !is_empty(&ready_q)) {
            pick = dequeue(&ready_q);       // 최단 남은시간
            enqueue(&running_q, pick);
        }

        /* 프로세스 실행 중 남은시간 업데이트 */
        if (!is_empty(&running_q)) {
            processes[pick].remaining_time =
                processes[pick].cpu_burst - processes[pick].progress;
        }
    }
}
```

C.2.2) 선점 SJF

```
Metrics *run_sjf_p(Process *processes, int count) {
    while (completed < count) {
        for (int i = 0; i < count; i++) {
            // 프로세스 도착 시 즉시 선점 검사
            if (processes[i].arrival_time == time) {
                if (!is_empty(&running_q)) {
                    int current = peek(&running_q);
                    int new_remaining = processes[i].cpu_burst - processes[i].progress;
                    int current_remaining = processes[current].cpu_burst -
                                            processes[current].progress;

                    if (new_remaining < current_remaining) {
                        // 선점 발생
                        preempted = dequeue(&running_q);
                        enqueue(&ready_q, preempted);
                        enqueue(&running_q, i);
                    } else {
                        enqueue(&ready_q, i);
                    }
                }
            }
```

```
            // I/O 완료 시에도 동일한 선점 검사 수행
            if (waiting_q[i] > 0) {
                waiting_q[i]--;
                if (waiting_q[i] == 0) {
                    /* I/O 완료 후 선점 검사 로직 */
                }
            }
        }

        /* ready queue 정렬 및 프로세스 실행 */
    }
}
```

D.3 우선순위 스케줄링

C.3.1) 선점 우선순위 핵심 로직

```
// 프로세스 도착 시 선점 검사
if (processes[i].arrival_time == time) {
    if (!is_empty(&running_q)) {
        int current = peek(&running_q);
        if (processes[i].priority < processes[current].priority) {
            // 선점 발생 (낮은 숫자 = 높은 우선순위)
            preempted = dequeue(&running_q);
            enqueue(&ready_q, preempted);
            enqueue(&running_q, i);
        }
    }
}
```

C.3.2) 에이징 메커니즘

```
Metrics *run_priority_with_aging(Process *processes, int count) {
    const int AGING_THRESHOLD = 3;
    int *age = malloc(sizeof(int) * count);

    while (completed < count) {
        // 1 단계: 에이징 처리 (최우선)
        int aging_candidates[100];
        int aging_count = 0;

        if (!is_empty(&ready_q)) {
            for (int i = ready_q.front, cnt = 0; cnt < ready_q.count;
                 cnt++, i = (i + 1) % MAX_QUEUE_SIZE) {
                int pid = ready_q.data[i];
                age[pid]++;

                if (age[pid] >= AGING_THRESHOLD && processes[pid].priority > 1) {
                    aging_candidates[aging_count++] = pid;
                }
            }
        }

        // 우선순위 상승 및 로깅
        for (int i = 0; i < aging_count; i++) {
```

```
            int pid = aging_candidates[i];
            int old_priority = processes[pid].priority;
            processes[pid].priority--;

            printf("| %4d | P%-10d | %-13d | %7d → %-6d|\n",
                  time, pid, age[pid], old_priority, processes[pid].priority);
            age[pid] = 0;
        }

        // 2 단계: 에이징 기반 선점 검사
        if (!is_empty(&running_q) && !is_empty(&ready_q)) {
            int running_pid = peek(&running_q);
            int top_ready_pid = peek(&ready_q);

            if (processes[top_ready_pid].priority <
                processes[running_pid].priority) {
                // 에이징으로 인한 선점 발생
                preempted = dequeue(&running_q);
                enqueue(&ready_q, preempted);
                promoted = dequeue(&ready_q);
                enqueue(&running_q, promoted);
                age[promoted] = 0;
            }
        }

        /* 일반 프로세스 도착 및 I/O 처리 */
    }
}
```

D.4 Round Robin

```
Metrics *run_rr(Process *processes, int count, Config *config) {
    int *time_quantum = malloc(sizeof(int) * count);
    int quantum = config->time_quantum;

    while (completed < count) {
        // 1 단계: 타임 퀀텀 만료 처리 (최우선)
        if (!is_empty(&running_q) &&
            time_quantum[peek(&running_q)] == quantum) {
            int rotated = dequeue(&running_q);
            enqueue(&ready_q, rotated);
            time_quantum[rotated] = 0;
        }

        /* 프로세스 도착 및 I/O 완료 처리 */

        // CPU 스케줄링: FIFO 순서
        if (is_empty(&running_q) && !is_empty(&ready_q)) {
            pick = dequeue(&ready_q);
            enqueue(&running_q, pick);
        }

        // 프로세스 실행 시 타임 퀀텀 증가
        if (!is_empty(&running_q)) {
            pick = peek(&running_q);
            processes[pick].progress++;
            time_quantum[pick]++;
```

```
            // I/O 시작 시 타임 퀀텀 리셋
            if (has_io_at_progress(&processes[pick], processes[pick].progress)) {
                time_quantum[waiting] = 0;
            }
        }
    }
}
```

D.5 실시간 스케줄링 (RMS/EDF)

D.5.1) 동적 프로세스 인스턴스 생성

```
// RMS/EDF 공통 로직
while (time < max_time) {
    // 새로운 프로세스 인스턴스 생성
    for (int i = 0; i < count; i++) {
        if (next_arrival[i] == time) {
            Process new_process;
            memcpy(&new_process, &original_processes[i], sizeof(Process));

            new_process.pid = i;
            new_process.arrival_time = time;

            // 데드라인 계산
            if (time == original_processes[i].arrival_time) {
                new_process.deadline = original_processes[i].deadline;
            } else {
                int period_count = (time - original_processes[i].arrival_time) /
                                original_processes[i].period;
                new_process.deadline = original_processes[i].deadline +
                                    period_count * original_processes[i].period;
            }

            // 스케줄링 기준에 따른 선점 검사
            if (!is_empty(&running_q)) {
                int current_running = peek(&running_q);

                // RMS: period 기준, EDF: deadline 기준
                int should_preempt = (algorithm == RMS) ?
                    (new_process.period < all_processes[current_running].period) :
                    (new_process.deadline < all_processes[current_running].deadline);

                if (should_preempt) {
                    preempted = dequeue(&running_q);
                    enqueue(&ready_q, preempted);
                    enqueue(&running_q, total_process_count);
                }
            }

            next_arrival[i] += original_processes[i].period;
            total_process_count++;
        }
    }

    /* I/O 완료 처리 및 데드라인 미스 검사 */
}
```

D.5.2) 데드라인 미스 처리

```
// 프로세스 완료 시 데드라인 검사
if (all_processes[current_running].progress ==
    all_processes[current_running].cpu_burst) {

    int finished = current_running;
    all_processes[finished].comp_time = time + 1;

    // 데드라인 미스 체크
    if (all_processes[finished].comp_time > all_processes[finished].deadline) {
        deadline_miss_info[config->deadline_miss_info_count].pid =
            all_processes[finished].pid;
        deadline_miss_info[config->deadline_miss_info_count].absolute_deadline =
            all_processes[finished].deadline;
        deadline_miss_info[config->deadline_miss_info_count].completion_time =
            all_processes[finished].comp_time;

        all_processes[finished].missed_deadline = 1;
        config->deadline_miss_info_count++;

        printf("| %4d | P%-10d | %-16d | %-15d |\n", time + 1,
               all_processes[finished].pid,
               all_processes[finished].deadline,
               all_processes[finished].comp_time);
    }

    dequeue(&running_q);
}
```

Appendix E: 유틸리티 함수

E.1 큐 정렬 함수 (utils.c)

```
typedef enum {
    SORT_BY_PRIORITY,
    SORT_BY_REMAINING_TIME,
    SORT_BY_DEADLINE,
    SORT_BY_PERIOD
} SortCriteria;

int compare_processes(Process *processes, int pid1, int pid2, SortCriteria criteria) {
    switch (criteria) {
        case SORT_BY_PRIORITY:
            return processes[pid1].priority - processes[pid2].priority;
        case SORT_BY_REMAINING_TIME:
            return (processes[pid1].cpu_burst - processes[pid1].progress) -
                   (processes[pid2].cpu_burst - processes[pid2].progress);
        case SORT_BY_DEADLINE:
            return processes[pid1].deadline - processes[pid2].deadline;
        case SORT_BY_PERIOD:
            return processes[pid1].period - processes[pid2].period;
        default:
            return 0;
    }
```

```
}

void sort_queue(Queue *queue, Process *processes, SortCriteria criteria) {
    if (is_empty(queue) || queue->count <= 1) return;

    // 큐의 모든 원소를 임시 배열에 복사
    int *temp_array = malloc(sizeof(int) * queue->count);
    int temp_count = 0;

    while (!is_empty(queue)) {
        temp_array[temp_count++] = dequeue(queue);
    }

    // 버블 정렬로 배열 정렬
    for (int i = 0; i < temp_count - 1; i++) {
        for (int j = 0; j < temp_count - i - 1; j++) {
            if (compare_processes(processes, temp_array[j],
                                  temp_array[j + 1], criteria) > 0) {
                int temp = temp_array[j];
                temp_array[j] = temp_array[j + 1];
                temp_array[j + 1] = temp;
            }
        }
    }

    // 정렬된 배열을 다시 큐에 삽입
    for (int i = 0; i < temp_count; i++) {
        enqueue(queue, temp_array[i]);
    }

    free(temp_array);
}
```

E.2 멀티 I/O 처리 함수

```
// 특정 진행도에서 I/O가 시작되는지 확인
int has_io_at_progress(Process *p, int progress) {
    for (int i = 0; i < MAX_IO_OPERATIONS; i++) {
        if (p->io_operations[i].io_start == progress) {
            return 1;
        }
    }
    return 0;
}

// 특정 진행도에서의 I/O 버스트 시간 반환
int get_io_burst_at_progress(Process *p, int progress) {
    for (int i = 0; i < MAX_IO_OPERATIONS; i++) {
        if (p->io_operations[i].io_start == progress) {
            return p->io_operations[i].io_burst;
        }
    }
    return 0;
}

// I/O 작업 개수 반환
int get_io_count(Process *p) {
    int count = 0;
```

```
    for (int i = 0; i < MAX_IO_OPERATIONS; i++) {
        if (p->io_operations[i].io_start != -1) {
            count++;
        }
    }
    return count;
}
```

Appendix F: 성능 평가 및 결과 출력

F.1 간트 차트 생성

```
void display_gantt_chart(GanttChart *gantt, const char *algorithm_name) {
    printf("\n** Gantt Chart for %s **\n\n", algorithm_name);
    GanttChart *consolidated = consolidate_gantt_chart(gantt);

    // 상단 경계선
    printf(" ");
    for (int i = 0; i < consolidated->count; i++) {
        int duration = consolidated->entries[i].time_end -
                    consolidated->entries[i].time_start;
        for (int j = 0; j < duration; j++) {
            printf("--");
        }
        printf(" ");
    }
    printf("\n");

    // 프로세스 ID 표시
    printf("|");
    for (int i = 0; i < consolidated->count; i++) {
        int duration = consolidated->entries[i].time_end -
                    consolidated->entries[i].time_start;

        if (consolidated->entries[i].process_id == -1) {
            // 유휴 상태 표시
            for (int j = 0; j < duration - 1; j++) printf(" ");
            printf("ID");
            for (int j = 0; j < duration - 1; j++) printf(" ");
        } else {
            // 프로세스 ID 표시
            char pid_str[10];
            sprintf(pid_str, "P%d", consolidated->entries[i].process_id);

            for (int j = 0; j < duration - 1; j++) printf(" ");
            printf("%s", pid_str);
            for (int j = 0; j < duration - 1; j++) printf(" ");
        }
        printf("|");
    }
    printf("\n");

    // 시간 축 표시
    printf("0");
    for (int i = 0; i < consolidated->count; i++) {
        int duration = consolidated->entries[i].time_end -
                    consolidated->entries[i].time_start;
        int end_time = consolidated->entries[i].time_end;
```

```
        for (int j = 0; j < duration; j++) {
            printf("  ");
        }

        if (end_time > 9) printf("\b");
        if (end_time > 99) printf("\b");
        printf("%d", end_time);
    }
    printf("\n");
}
```

F.2 성능 메트릭 계산

```
void display_performance_summary(Process *processes, int count,
                                 int total_time, int idle_time) {
    int total_waiting = 0;
    int total_turnaround = 0;
    int total_io_operations = 0;

    for (int i = 0; i < count; i++) {
        total_waiting += processes[i].waiting_time;
        total_turnaround += processes[i].turnaround_time;
        total_io_operations += get_io_count(&processes[i]);
    }

    float avg_waiting = (float)total_waiting / count;
    float avg_turnaround = (float)total_turnaround / count;
    float cpu_utilization = ((float)(total_time - idle_time) / total_time) * 100.0;
    float avg_io_per_process = (float)total_io_operations / count;

    printf("| Average Waiting Time      : %6.2f ms                       |\n",
           avg_waiting);
    printf("| Average Turnaround Time   : %6.2f ms                       |\n",
           avg_turnaround);
    printf("| CPU Utilization           : %6.2f %%                       |\n",
           cpu_utilization);
    printf("| Average I/O per Process   : %6.2f operations                 |\n",
           avg_io_per_process);
}
```

F.3 실시간 시스템 분석

```
void print_utilization_analysis(Process *original_processes, int count,
                                const char *algorithm_name) {
    printf("\n** Utilization Analysis for %s **\n", algorithm_name);

    float total_utilization = 0.0;

    for (int i = 0; i < count; i++) {
        // I/O 시간 포함 실행시간 계산
        int total_io_time = 0;
        for (int j = 0; j < MAX_IO_OPERATIONS; j++) {
            if (original_processes[i].io_operations[j].io_start != -1) {
                total_io_time += original_processes[i].io_operations[j].io_burst;
            }
```

```
        }

        int execution_time = original_processes[i].cpu_burst + total_io_time;
        float individual_util = (float)execution_time / original_processes[i].period;
        total_utilization += individual_util;

        printf("| P%-3d | %-11d | %-8d | %-11.3f | %-13.2f |\n",
               i, execution_time, original_processes[i].period,
               individual_util, individual_util * 100);
    }

    printf("| Total System Utilization: %.3f (%.2f%%)                |\n",
           total_utilization, total_utilization * 100);

    // 스케줄링 가능성 분석
    float rms_bound = count * (pow(2.0, 1.0 / count) - 1);

    printf("- EDF: %s (Utilization <= 100%%)\n",
           total_utilization <= 1.0 ? "SCHEDULABLE" : "NOT SCHEDULABLE");
    printf("- RMS: %s (Utilization <= %.2f%%)\n",
           total_utilization <= rms_bound ? "SCHEDULABLE" : "UNCERTAIN",
           rms_bound * 100);
}
```

Appendix G: 구현 시 고려사항

G.1 메모리 관리

- 동적 메모리 할당 시 반드시 free() 호출
- 프로세스 배열, 큐, 간트 차트 엔트리의 적절한 해제
- 메모리 누수 방지를 위한 체계적인 관리

G.2 경계 조건 처리

- 빈 큐에 대한 dequeue 연산 방지
- 배열 인덱스 범위 초과 방지
- I/O 작업 개수 제한 (MAX_IO_OPERATIONS) 준수

G.3 정확성 보장

- 프로세스 상태 전환의 원자성 보장
- 타임 퀀텀 카운터의 정확한 관리
- 데드라인 미스 검사의 정확한 타이밍

G.4 확장성 고려

- 새로운 스케줄링 알고리즘 추가 용이성
- I/O 작업 수 확장 가능성
- 성능 메트릭 추가 용이성