운영체제 프로젝트 중간보고서

1. **프로젝트 개요 및 설명**
    a. **프로젝트 개요**
        - 운영체제의 핵심 역할 중 하나는 여러 프로세스와 스레드를 효율적으로 관리하고 자원을 할당하는 것
        - 운영체제의 중추적인 부분인 스케줄러를 설계하고 구현함으로써, 실제 시스템에서의 프로세스 관리와 자원 배분에 대한 이해를 깊게 하고자 함
    b. **프로젝트 목표**
        - 다양한 스케줄링 알고리즘(Round Robin, Priority Scheduling, SRTF)의 원리와 장단점을 학습
        - 스케줄링 알고리즘은 C를 사용하여 구현
        - 구현된 스케줄러의 성능을 시뮬레이션을 통해 분석하고, 결과를 통해 각 알고리즘의 효율성을 비교 및 평가

2. **프로젝트 진행 상황**
    a. **Multilevel Queue Scheduling(MLQ)**
        - **학습 내용**
            - 프로세스를 우선순위나 유형에 따라 여러 큐에 분류하여 관리하는 방식
            - 각 큐는 자신의 스케줄링 알고리즘을 가질 수 있으며 우선순위가 높은 큐의 프로세스가 먼저 CPU 시간을 할당 받음
            - 실시간 작업이나 중요한 작업에 더 빠른 응답을 제공할 수 있다
            - 큐 간에 자원 이동이 제한적이여서 유연성이 낮으며 큐의 수와 각 큐의 스케줄링 정책을 적절히 설정하기가 어렵다
        - **구현 및 설명**
            - **executeProcess 함수**

싱글 프로세스를 실행하는 역할, 프로세스의 ID, 우선순위, 버스트 시간을 출력

- **multilevelQueue 함수**

  각 프로세스가 우선순위에 따라 이미 정렬되어 있다고 가정하며 주어진 프로세스 배열을 실행

```
#include <stdio.h>
#include <stdbool.h>
#include <stdlib.h>
#include <limits.h>

#define MAX_PROCESSES 100

typedef struct {
    int pid;
    int arrival_time;
    int burst_time;
    int remaining_time;
    int priority;
} Process;

///multilevel Queue
void executeProcess(Process p) {
    printf("Process %d with priority %d executes for %(
}

void multilevelQueue(Process processes[], int n) {
    // 우선순위 별로 정렬되어있다고 가정함
    for (int i = 0; i < n; i++) {
        executeProcess(processes[i]);
    }
}
```

b. **Round Robin Scheduling**

- **학습 내용**
  - 각 프로세스에 동일한 타임 퀀텀을 주어 순차적으로 CPU시간을 배분하는 방식
  - 모든 프로세스는 준비 큐에서 순서대로 타임 퀀텀 동안 실행되며, 시간이 초과되면 큐의 끝으로 이동한다
  - 모든 프로세스가 동등하게 CPU 시간을 할당받아 공정성을 보장함
  - 타임 퀀텀이 너무 작으면 오버헤드가 증가하고, 너무 크면 FIFO와 유사하게 동작하기 때문에 타임 퀀텀의 크기를 잘 설정해줘야 함
- **구현 및 설명**
  - **roundRobin 함수**

    프로세스의 남은 시간이 타임 퀀텀보다 크면, 타임 퀀텀만큼 시간을 소비하고, 남은 시간을 줄임
    프로세스의 남은 시간이 타임 퀀텀보다 작거나 같으면, 남은 시간만큼만 시간을 소비하고, 프로세스를 완료하며 프로세스가 종료되는 시간을 출력

```
///RoundRobin
void roundRobin(Process processes[], int n, int quantur
    int time = 0, i;
    bool done = false;

    while (!done) {
        done = true;
        for (i = 0; i < n; i++) {
            if (processes[i].remaining_time > 0) {
                done = false;
                if (processes[i].remaining_time > quant
                    time += quantum;
                    processes[i].remaining_time -= quar
                } else {
                    time += processes[i].remaining_time
                    printf("Process %d finishes at time
                    processes[i].remaining_time = 0;
                }
```

```
            }
        }
    }
}
```

c. **SRTF Scheduling**

- **학습 내용**
  - 가장 짧은 남은 실행 시간을 가진 프로세스에게 CPU를 할당함
  - 새 프로세스가 도착하면 현재 실행 중인 프로세스와 남은 실행 시간을 비교하여, 새 프로세스가 더 짧을 경우 즉시 선점한다.
  - 매우 효율적인 스케줄링 방식으로, 짧은 작업을 빠르게 처리한다
  - 선점으로 인해 컨텍스트 스위칭이 자주 발생하여 오버헤드가 증가할 수 있고 실행 시간을 미리 알아야하며 실제 환경에서 구현하기가 어렵다.
- **구현 및 설명**
  - **findNextProcess 함수**

    가장 짧은 남은 실행 시간을 가진 프로세스를 찾아 인덱스를 반환
  - **SRTF 함수**

    findNextProcess 함수를 호출하여 현재 시간에 실행할 프로세스를 찾음

    찾은 프로세스가 있다면, 해당 프로세스의 남은시간을 1 감소시키고, 현재 시간을 1 증가

    프로세스의 남은시간이 0이 되면 프로세스가 완료되었음을 출력하고 completed 수를 증가

    찾은 프로세스가 없다면, 시간을 1 증가시키고 다음 타이밍을 기다림

    모든 프로세스가 완료될 때까지 반복

```
///SRTF
int findNextProcess(Process processes[], int n, int cu
    int min_time = INT_MAX, index = -1;
    for (int i = 0; i < n; i++) {
        if (processes[i].arrival_time <= current_time &
```

```
            if (processes[i].remaining_time < min_time
                min_time = processes[i].remaining_time
                index = i;
            }
        }
    }
    return index;
}

void SRTF(Process processes[], int n) {
    int completed = 0, current_time = 0, i;

    while (completed != n) {
        int index = findNextProcess(processes, n, curre
        if (index != -1) {
            processes[index].remaining_time--;
            if (processes[index].remaining_time == 0) {
                completed++;
                printf("Process %d finishes at time %d\
            }
            current_time++;
        } else {
            current_time++;
        }
    }
}
```

3. **프로젝트 진행 계획**
    a. **싱크코어 프로세서 환경에서의 스케줄링 보안**
        - MLQ 스케줄러에서 각 큐가 각각의 스케줄링 알고리즘을 사용할 수 있도록 구현
        - SRTF에서 동적 사항을 처리하여 프로세스의 도착 시간이 불규칙하거나 실행 시간 추정이 어려운 상황에서의 적응성 높이기
    b. **멀티코어 프로세서 환경에서의 스케줄링 구현**

- 멀티 프로세싱 스케줄링 기법과 그 적용 사례를 학습
- 멀티 프로세서 환경을 위한 스케줄링 알고리즘을 C를 사용하여 구현
- 구현된 스케줄러의 성능을 시뮬레이션을 통해 분석하고, 실제 멀티코어 환경에서의 효율성을 평가

※ 성능 평가는 처리량, 반응 시간, CPU 이용률 등을 측정하여 성능 지표로 활용할 예정

4. **트러블 슈팅**