[프로젝트 **02**] 운영체제 스케쥴러 시뮬레이터

SW 설계서

학과	학번	이름
정보보호학과	2023111740	김묘진
정보보호학과	2023111392	유승리
정보보호학과	2023111399	이시온
정보보호학과	2023111403	이정서

1. 개요

가. 시스템 목적:

본 설계서는 OS 스케쥴러 시뮬레이터의 설계를 설명합니다. 이 시스템의 목적은 여러 시스템(RR, Priority, SJF등)을 구현하고 시뮬레이팅하여 운영체제의 프로세스나 시스템 관리에 사용되는 여러 알고리즘들의 연산 과정과 성능 지표를 확인하는 것에 있습니다.

나. 주요 기능

- 외부 파일 입력 : 입력된 테스트 파일을 바탕으로 시뮬레이터가 자동으로 프로세스를 실행하고 결과를 출력합니다.
- thread scheduling 시뮬레이팅: 시뮬레이터의 타임 슬라이스는 1이며 프로세스마다의 실행 순서, 대기 시간, 반환 시간을 시뮬레이션합니다.
- Gantt 차트 출력: 알고리즘의 시뮬레이션 과정을 추적하고 출력합니다. 각 프로세스가 시작된 때에 프로세스 ID, 실행한 시간을 표시합니다.
- 성능 지표 분석 및 출력: 각 알고리즘의 총 실행 시간, 대기 시간, 반환 시간 등의 성능 지표를 계산합니다.

다. 기술 스택

- 언어: C
- 빌드: gcc
- 입력 방식: 테스트 파일 입력

2. 외부 코드 참조 - 타 코드 및 라이브러리 사용
    - https://velog.io/@strn18/OS-CPU-%EC%8A%A4%EC%BC%80%EC%A4%84%EB%A7%81-%EC%8B%9C%EB%AE%AC%EB%A0%88%EC%9D%B4%ED%84%B0

3. 시스템 설계

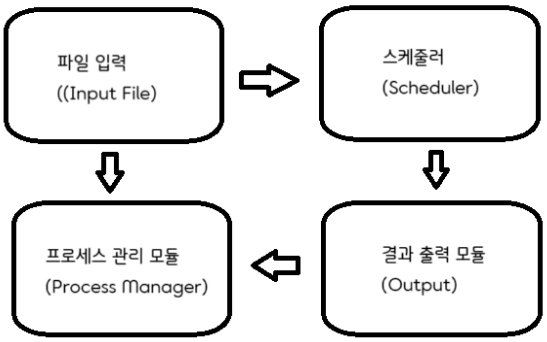
가. 전체 구조

이 시스템의 구조는 크게 파일 입력, 스케쥴러 시뮬레이션 실행, 결과 출력으로 나누어 볼 수 있습니다.

- 파일 입력: 읽기 모드로 파일을 열어 파일을 한 문자씩 읽어옵니다. 만일 해당 파일이 존재하지 않는다면 “파일을 열 수 없습니다”라는 문구가 출력됩니다.
  성공적으로 읽어온 경우 동적 메모리 할당을 위한 배열 포인터 선언을 통해 파일에서 읽어 온 입력을 각 구조체에 할당합니다. 반환시간과 대기 시간을 필요로 하는 LJF 선점형/비선점형은 분리된 구조체로 선언합니다.

- 스케줄러 : 총 8종의 스케줄링 알고리즘을 구현하여 각 프로세스에서 우선순위를 두어 실행하는 기준에 따라 연산합니다. 입력된 파일의 실행 시간과 도착 시간, 우선 순위 등을 통해 완료, 반환, 대기 시간 등을 계산하고 Gantt 차트를 출력합니다.

- 결과 출력 : 스케쥴러 별로 계산한 결과를 파일에 출력하기 전 파일 포인터를 지정, 'cpuschedule_result.txt'에 쓰기 모드로 입력합니다. 만일 탐색되지 않는다면 "파일을 열 수 없습니다"가 출력되고 비정상 오류로 반환됩니다. 지정이 완료된다면 출력과 파일 저장 모두 이행됩니다. 최종 출력 이후 할당했던 동적 메모리를 해제합니다.



나. 담당한 부분

1) 김묘진 : FCFS, RR 알고리즘 작성, 코드 최종 통합 및 자동화 알고리즘 작성

2) 이시온 : SJF Preemptive & Non Preemptive 알고리즘 작성, 코드 최종 통합 및 자동화 알고리즘 작성

3) 유승리 : LJF Preemptive & Non Preemptive 알고리즘 작성, 설계서 작성

4) 이정서 : Priority Preemptive & Non Preemptive 알고리즘 작성, 설계서 작성

다. 설계 상세

1. FCFS

1.1 라이브러리 포함 및 전역 변수 선언

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <limits.h>

#define MAX_THREADS 100  // 최대 스레드 수
#define MAX_OUTPUT 1000 // 출력 저장을 위한 최대 버퍼 크기

typedef struct {
    char id[10];   // 스레드 ID
    int arrival_time;      // 도착 시간
    int burst_time;      // 실행 시간
    int priority;          // 우선순위
    int remaining_time;   // 남은 실행 시간 (선점형 알고리즘에서 사용)
    int completion_time;  // 완료 시간
} Thread;


Thread threads[MAX_THREADS];    // 스레드 배열
int thread_count = 0;    // 스레드 개수
char output_non_preemptive[MAX_OUTPUT] = "";  // 비선점 출력 저장
char output_preemptive[MAX_OUTPUT] = "";      // 선점 출력 저장
double results[2][3];                         // sjf결과 저장: [스케줄링][완료, 턴어라운드, 대기]
double fresults[3];                           //fcfs 결과 저장
double rresults[3];                           // rr 결과 저장
double ljf_results[2][3];       // LJF 결과 저장: [스케줄링 유형][완료 시간, 턴어라운드, 대기]
```

– 라이브러리를 포함하고 스레드 구조체에 파일에서 읽어들여야 할 요소들과 결과 출력을 위해 필요한 요소들을 선언합니다. 출력과 결과를 저장하는 배열은 스레드 배열에 전역 변수로 선언합니다.

1.2 데이터 로드 함수

```c
void load_threads() {
    while (1) {
        Thread t;
        if (fscanf(stdin, "%s", t.id) != 1 || strcmp(t.id, "E") == 0) {
            break;
        }
        fscanf(stdin, "%d %d %d", &t.arrival_time, &t.burst_time, &t.priority);  // 도착 시간, 실행 시간, 우선순위 읽기
        t.remaining_time = t.burst_time;  // 초기 남은 시간 설정
        t.completion_time = 0; // 초기 완료 시간 설정
        threads[thread_count++] = t;  // 스레드 배열에 추가
    }
}
```

- 입력된 파일에서 E까지 스레드 정보를 읽어오고 이를 threads 배열에 저장합니다. E를 읽은 경우 반복구조가 break 되어 파일의 값을 읽어오는 루프가 종료됩니다.

1.3 도착 시간 정렬 함수

```
void sort_by_arrival_time() {
    for (int i = 0; i < thread_count - 1; i++) {
        for (int j = i + 1; j < thread_count; j++) {
            if (threads[i].arrival_time > threads[j].arrival_time) {  // 도착 시간이 빠른 순으로 정렬
                Thread temp = threads[i];
                threads[i] = threads[j];
                threads[j] = temp;
            }
        }
    }
}
```

- FCFS는 'First-Come, First-Served', 즉 먼저 도착한 프로세스가 먼저 실행됩니다. 한 프로세스가 CPU를 할당받으면, 해당 프로세스가 실행을 마칠 수 없는 비선점형이기 때문에 추가적인 비교 없이 도착 시간이 빠른 순으로 레디큐에 정렬하는 그대로 실행 순서가 됩니다. 이중 반복 구조를 통해 도착 시간을 비교하여 정렬합니다.

1.4 FCFS 결과 계산 함수

```
void calculate_results_fcfs() {
    int total_turnaround_time = 0;
    int total_waiting_time = 0;
    int final_completion_time = 0;

    for (int i = 0; i < thread_count; i++) {
        int turnaround_time = threads[i].completion_time - threads[i].arrival_time;
        int waiting_time = turnaround_time - threads[i].burst_time;

        total_turnaround_time += turnaround_time; //총 턴어라운드 시간 누적
        total_waiting_time += waiting_time; // 총 대기 시간 누적

        if (threads[i].completion_time > final_completion_time) {
            final_completion_time = threads[i].completion_time; // 가장 늦게 완료된 시간 갱신
        }
    }

    fresults[0] = final_completion_time;                          // 전체 완료 시간
    fresults[1] = (double)total_turnaround_time / thread_count;   // 평균 Turnaround Time
    fresults[2] = (double)total_waiting_time / thread_count;      // 평균 Waiting Time
}
```

- 결과에서 완료 시간 외엔 Average 값을 요구하기 때문에, 프로세스 별로 완료 시간에서 도착 시간을 뺀 후 총값을 합한 총 반환 시간과 프로세스 별로 반환 시간에서 실행 시간을 뺀 대기 시간에 각각 스레드의 수 만큼을 나누어 평균 값을 도출합니다. 이후 도출된 값은 FCFS의 결과값을 저장하는 배열 fresults[3]에 입력됩니다.

1.5 FCFS 스케줄링 함수

```
void fcfs() {
    int time = 0;

    printf("FCFS Scheduling:₩n");

    for (int i = 0; i < thread_count; i++) {
        // 도착 시간이 현재 시간보다 늦을 경우 대기
        if (threads[i].arrival_time > time) {
            printf("%d: IDLE₩n", time);
            time = threads[i].arrival_time;
        }

        // 스레드 실행
        printf("%d: %s (%d)₩n", time, threads[i].id, threads[i].burst_time);
        time += threads[i].burst_time;

        threads[i].completion_time = time;  // 완료 시간 기록
    }

    printf("%d: E₩n", time);

    // 결과 계산
    calculate_results_fcfs();
}
```

- 각 프로세스가 도착하는 시간에 맞추어 도착하도록 arrival time이 time (현재시간)보다 크면 대기하도록 정리하고, 이전 과정에서 도착 순서대로 정렬된 threads[i]를 실행합니다. 레디큐에서 가장 먼저 도착한 프로세스부터 시작되게 되며, 현재 프로세스가 종료되면 그 다음 프로세스를 실행하며 모든 프로세스가 처리될 때까지 계속해서 큐에서 프로세스를 꺼내어 실행합니다. 프로세스가 종료된 후 time에 누적된 시간을 완료 시간으로 기록합니다.

2. SJF (non preemptive & preemptive)

2.1 라이브러리 포함 및 전역 변수 선언

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <limits.h>

#define MAX_THREADS 100  // 최대 스레드 수
#define MAX_OUTPUT 1000 // 출력 저장을 위한 최대 버퍼 크기

typedef struct {
    char id[10];   // 스레드 ID
    int arrival_time;      // 도착 시간
    int burst_time;      // 실행 시간
    int priority;          // 우선순위
    int remaining_time;   // 남은 실행 시간 (선점형 알고리즘에서 사용)
    int completion_time;  // 완료 시간
} Thread;


Thread threads[MAX_THREADS];    // 스레드 배열
int thread_count = 0;    // 스레드 개수
char output_non_preemptive[MAX_OUTPUT] = "";  // 비선점 출력 저장
char output_preemptive[MAX_OUTPUT] = "";      // 선점 출력 저장
double results[2][3];                         // sjf결과 저장: [스케줄링][완료, 턴어라운드, 대기]
double fresults[3];                           //fcfs 결과 저장
double rresults[3];                           // rr 결과 저장
double ljf_results[2][3];       // LJF 결과 저장: [스케줄링 유형][완료 시간, 턴어라운드, 대기]
```

- 라이브러리를 포함하고 스레드 구조체에 파일에서 읽어들여야 할 요소들과 결과 출력을 위해 필요한 요소들을 선언합니다. 출력과 결과를 저장하는 배열은 스레드 배열에 전역 변수로 선언합니다.

2.2 데이터 로드 함수

```c
void load_threads() {
    while (1) {
        Thread t;
        if (fscanf(stdin, "%s", t.id) != 1 || strcmp(t.id, "E") == 0) {
            break;
        }
        fscanf(stdin, "%d %d %d", &t.arrival_time, &t.burst_time, &t.priority);  // 도착 시간, 실행 시간, 우선순위 읽기
        t.remaining_time = t.burst_time;  // 초기 남은 시간 설정
        t.completion_time = 0; // 초기 완료 시간 설정
        threads[thread_count++] = t;  // 스레드 배열에 추가
    }
}
```

- 입력된 파일에서 E까지 값을 읽어와 스레드 배열에 추가합니다. E를 읽은 경우 반복구조가 break 되어 파일의 값을 읽어오는 루프가 종료됩니다.

2.3 도착 시간 정렬 함수

```c
void sort_by_arrival_time() {
    for (int i = 0; i < thread_count - 1; i++) {
        for (int j = i + 1; j < thread_count; j++) {
            if (threads[i].arrival_time > threads[j].arrival_time) {  // 도착 시간이 빠른 순으로 정렬
                Thread temp = threads[i];
                threads[i] = threads[j];
                threads[j] = temp;
            }
        }
    }
}
```

- 이중 반복구조를 통해 도착 시간을 비교, 정렬하여 시간에 혼선이 생기는 것을 방지합니다.

2.4 sjf 결과 요약 계산 함수

```
void calculate_results_sjf(int type) {
    int total_turnaround_time = 0;
    int total_waiting_time = 0;

    int final_completion_time = 0;

    for (int i = 0; i < thread_count; i++) {
        int turnaround_time = threads[i].completion_time - threads[i].arrival_time;
        int waiting_time = turnaround_time - threads[i].burst_time;

        total_turnaround_time += turnaround_time;
        total_waiting_time += waiting_time;

        if (threads[i].completion_time > final_completion_time) {
            final_completion_time = threads[i].completion_time;  // 가장 늦은 종료 시간
        }
    }

    results[type][0] = final_completion_time;  // 전체 완료 시간
    results[type][1] = (double)total_turnaround_time / thread_count;  // 평균 Turnaround Time
    results[type][2] = (double)total_waiting_time / thread_count;    // 평균 Waiting Time
}
```

- 결과에서 완료 시간 외엔 Average 값을 요구하기 때문에, 프로세스 별로 완료 시간에서 도착 시간을 뺀 후 총값을 합한 총 반환 시간과 프로세스 별로 반환 시간에서 실행 시간을 뺀 대기 시간에 각각 스레드의 수 만큼을 나누어 평균 값을 도출합니다. 이후 도출된 값은 SJF의 결과값을 저장하는 배열 results[2][3]에 입력됩니다.

2.5 SJF 비선점

```
void sjf_non_preemptive() {
    int time = 0;
    int completed = 0;
    int visited[MAX_THREADS] = { 0 };

    sprintf(output_non_preemptive, "₩nSJF (non-preemptive):₩n");
    while (completed < thread_count) {
        int shortest = -1;
        for (int i = 0; i < thread_count; i++) {
            if (!visited[i] && threads[i].arrival_time <= time) {
                if (shortest == -1 || threads[i].burst_time < threads[shortest].burst_time) {
                    shortest = i;
                }
            }
        }
        if (shortest != -1) {
            visited[shortest] = 1;
            sprintf(output_non_preemptive + strlen(output_non_preemptive),
                "%d: %s (%d)₩n", time, threads[shortest].id, threads[shortest].burst_time);
            time += threads[shortest].burst_time;
            threads[shortest].completion_time = time;  // 완료 시간 기록
            completed++;
        }
        else {
            time++;
        }
    }
    sprintf(output_non_preemptive + strlen(output_non_preemptive), "%d: E₩n", time);

    // 결과 계산
    calculate_results_sjf(0);
}
```

- time은 현재 시간으로 초기값이 0이며 completed는 완료된 스레드의 수 , output_non_preemptive는 출력 문자열을 저장하는 변수, 즉 시뮬레이션 결과를 나타냅니다. SJF (Shortest Job First) 스케줄러 중에서도 비선점은 가장 짧은 작업을 먼저 완료한 후 , 다음 빠른 작업을 순차적으로 처리해주는 알고리즘이기 때문에 thread_count가 0이 되기 전까지 반복하는 while문 내부에서 반복적으로 비교, 실행, 종료, 비교의 순서를 거치게 됩니다. 위 코드에서는 shortest가 -1일때, 즉 실행하고 있는 프로세서가 없을 때 현재 시간(time)에 도착해 있는 프로세스들 사이에서 비교하여 실행합니다. 실행 중이던 프로세스가 종료될 때 시작시간과 ID, 실행한 시간이 출력됩니다. 종료되지 않은 상황에서는 time에 시간이 누적됩니다.

2.6 SJF 선점

```
void sjf_preemptive() {
    int time = 0;
    int completed = 0;
    int prev_thread = -1;
    int start_time = 0; // 현재 스레드 시작 시간

    sprintf(output_preemptive, "SJF (preemptive):₩n");

    while (completed < thread_count) {
        int shortest = -1;

        // 현재 시간에서 실행 가능한 스레드 중 남은 실행 시간이 가장 짧은 스레드 선택
        for (int i = 0; i < thread_count; i++) {
            if (threads[i].remaining_time > 0 && threads[i].arrival_time <= time) {
                if (shortest == -1 || threads[i].remaining_time < threads[shortest].remaining_time) {
                    shortest = i;
                }
            }
        }

        if (shortest != -1) {
            // 새로운 스레드가 선택된 경우 출력
            if (prev_thread != shortest) {
                if (prev_thread != -1) {
                    // 이전 스레드가 종료된 시간을 기반으로 실행 시간 출력
                    sprintf(output_preemptive + strlen(output_preemptive),
                        "%d: %s (%d)₩n", start_time, threads[prev_thread].id, time - start_time);
                }
                prev_thread = shortest;
                start_time = time; // 새로운 스레드 시작 시간 설정
            }

            // 남은 시간 감소 및 시간 증가
            threads[shortest].remaining_time--;
            time++;

            // 스레드 완료 처리
            if (threads[shortest].remaining_time == 0) {
                threads[shortest].completion_time = time; // 완료 시간 기록
                completed++;
                sprintf(output_preemptive + strlen(output_preemptive),
                    "%d: %s (%d)₩n", start_time, threads[shortest].id, time - start_time);
                prev_thread = -1; // 스레드 초기화
            }
        }
        else {
            // 실행 가능한 스레드가 없을 경우 IDLE 출력
            if (prev_thread != -1) {
                sprintf(output_preemptive + strlen(output_preemptive),
                    "%d: %s (%d)₩n", start_time, threads[prev_thread].id, time - start_time);
                prev_thread = -1; // IDLE 상태로 전환
            }
            time++;
        }
    }

    // 종료 시간 출력
    sprintf(output_preemptive + strlen(output_preemptive), "%d: E₩n", time);

    // 결과 계산
    calculate_results_sjf(1);
}
```

- SJF (Shortest Job First) 스케줄러 중에서 선점형은 가장 짧은 작업을 먼저 실행하지만 도중에 도착한 프로세스와 비교, 더 짧다면 해당 스케쥴러가 실행되는 구조를 가지고 있습니다.
time은 현재 시간으로 초기값이 0이며 completed는 완료된 스레드 수,output_preemptive는 출력 문자열을 저장하는 변수, 즉 시뮬레이션 결과를 나타냅니다.
while구조 내에서 해당 알고리즘은 스레드가 모두 완료될 때까지 시간을 증가시키며 각 스레드를 실행합니다. 각 스레드를 실행하고, 실행 중인 스레드와 도중에 도착한 스레드를 비교하여 실행할 스레드를 결정합니다. 실행되고 있지 않을 때에는 남은 레디 큐에서 remaining_time을 비교하여 더 짧은 대상을 찾아 실행합니다. 이전에 실행되었던 스레드와 현재 실행 중인 스레드가 다르다면 실행되었던 스레드의 ID와 실행시간을 출력하고 start_time을 갱신합니다. 스레드가 완료된 경우(remaining_time==0)에도 시작시간과 ID, 실행한 시간을 출력하고 스레드 완료처리를 위해 completed에 1을 증가시킵니다. 실행 가능한 스레드가 없는 경우에는 대기 상태로 전환하고 time에 1을 증가시켜 시스템의 시간만 흐르게 합니다.

3. LJF (non preemptive & preemptive)

3.1 라이브러리 포함 및 전역 변수 선언

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <limits.h>

#define MAX_THREADS 100  // 최대 스레드 수
#define MAX_OUTPUT 1000 // 출력 저장을 위한 최대 버퍼 크기
```

```c
Thread threads[MAX_THREADS];    // 스레드 배열
int thread_count = 0;    // 스레드 개수
char output_non_preemptive[MAX_OUTPUT] = "";  // 비선점 출력 저장
char output_preemptive[MAX_OUTPUT] = "";      // 선점 출력 저장
double results[2][3];                         // sjf결과 저장: [스케줄링][완료, 턴어라운드, 대기]
double fresults[3];                           //fcfs 결과 저장
double rresults[3];                           // rr 결과 저장
double ljf_results[2][3];       // LJF 결과 저장: [스케줄링 유형][완료 시간, 턴어라운드, 대기]
// LJF 스케줄링 관련 구조체와 데이터
typedef struct {
    char id[10];       // 프로세스 ID
    int arrival_time;    // 도착 시간
    int burst_time;   // 실행 시간
    int remaining_time;    // 남은 실행 시간
    int completion_time;   // 완료 시간
    int turnaround_time;     // 턴어라운드 시간
    int waiting_time;    // 턴어라운드 시간
} LJFProcess;

// 전역 변수 선언
LJFProcess ljf_processes[MAX_THREADS]; // LJF 프로세스 배열
int ljf_process_count = 0;    // LJF 프로세스 개수
```

- 라이브러리를 포함하고 스레드 구조체에 파일에서 읽어들여야 할 요소들과 결과 출력을 위해 필요한 요소들을 선언합니다. 출력과 결과를 저장하는 배열은 스레드 배열에 전역 변수로 선언합니다. 우선 순위 데이터는 사용하지 않기 때문에 읽어오지 않습니다.

3.2 데이터 로드 함수

```
void load_ljf_processes() {
    ljf_process_count = 0;
    while (1) {
        LJFProcess p;
        if (fscanf(stdin, "%s", p.id) != 1 || strcmp(p.id, "E") == 0)
            break;
        fscanf(stdin, "%d %d", &p.arrival_time, &p.burst_time);
        p.remaining_time = p.burst_time;
        p.completion_time = 0;
        ljf_processes[ljf_process_count++] = p;
    }
}
```

- 입력된 파일에서 E까지 스레드 정보를 읽어오고 이를 threads 배열에 저장합니다. E를 읽은 경우 반복구조가 break 되어 파일의 값을 읽어오는 루프가 종료됩니다. 우선 순위를 제외한 값들을 파일에서 읽어옵니다.

3.3 LJF Non-Preemptive 스케줄링

```
void ljf_non_preemptive() {
    int time = 0, completed = 0;
    printf("₩nLJF (non-preemptive):₩n");

    while (completed < ljf_process_count) {
        int longest = -1;
        for (int i = 0; i < ljf_process_count; i++) {
            if (ljf_processes[i].remaining_time > 0 && ljf_processes[i].arrival_time <= time) {
                if (longest == -1 || ljf_processes[i].burst_time > ljf_processes[longest].burst_time) {
                    longest = i;
                }
            }
        }

        if (longest != -1) {
            printf("%d: %s (%d)₩n", time, ljf_processes[longest].id, ljf_processes[longest].burst_time);
            time += ljf_processes[longest].burst_time;
            ljf_processes[longest].remaining_time = 0;
            ljf_processes[longest].completion_time = time;
            ljf_processes[longest].turnaround_time = time - ljf_processes[longest].arrival_time;
            ljf_processes[longest].waiting_time = ljf_processes[longest].turnaround_time - ljf_processes[longest].burst_time;
            completed++;
        }
        else {
            printf("%d: IDLE₩n", time);
            time++;
        }
    }
    printf("%d: E₩n", time);
}
```

- LJF(Long Job First) 스케쥴러는 가장 긴 작업을 우선적으로 실행하는 스케줄링 알고리즘으로 가장 긴 프로세스를 찾아 실행하는 것을 모든 프로세스가 완료될 때까지 반복합니다.

completed는 완료된 프로세스의 수를 추적합니다. 모든 프로세스가 완료되면 이 값은 process_count와 동일해집니다. 메인 루프에서는 completed < ljf_process_count 조건이 참일 때까지 진행되고, longest가 가장 긴 실행 시간이 남은 프로세스의 인덱스를 저장합니다. 모든 프로세스를 순차적으로 확인하면서

레디 큐에 남은 프로세스 중 가장 긴 실행 시간이 남은 프로세스를 선택합니다. longest != -1은 실행할 프로세스가 있음을 알려주는데, 있다면 실행 시키고, 해당 프로세스가 완료된 것으로 표시합니다. 그후 프로세스의 변수를 계산하여 업데이트 합니다. 더 이상 실행할 프로세스가 없다면 대기 상태로 전환하고, 모든 프로세스가 완료되었다면 time(종료시간)을 출력합니다.

3.4 LJF Preemptive 스케줄링

```
void ljf_preemptive() {
    int time = 0, completed = 0;
    printf("₩nLJF (preemptive):₩n");

    while (completed < ljf_process_count) {
        int longest = -1;
        for (int i = 0; i < ljf_process_count; i++) {
            if (ljf_processes[i].remaining_time > 0 && ljf_processes[i].arrival_time <= time) {
                if (longest == -1 || ljf_processes[i].remaining_time > ljf_processes[longest].remaining_time) {
                    longest = i;
                }
            }
        }

        if (longest != -1) {
            printf("%d: %s (1)₩n", time, ljf_processes[longest].id);
            ljf_processes[longest].remaining_time--;
            time++;

            if (ljf_processes[longest].remaining_time == 0) {
                ljf_processes[longest].completion_time = time;
                ljf_processes[longest].turnaround_time = time - ljf_processes[longest].arrival_time;
                ljf_processes[longest].waiting_time = ljf_processes[longest].turnaround_time - ljf_processes[longest].burst_time;
                completed++;
            }
        }
        else {
            printf("%d: IDLE₩n", time);
            time++;
        }
    }
    printf("%d: E₩n", time);
}
```

- 현재 시간을 나타내는 time 변수를 0으로 초기화하고 완료된 프로세스 수를 나타내는 completed 변수를 0으로 초기화하고 스케줄링 알고리즘의 제목을 출력합니다. 완료된 프로세스 수가 전체 프로세스 수보다 적을 때까지 반복합니다. longest는 가장 긴 남은 실행 시간을 가진 프로세스의 인덱스를 나타내며 초기값은 -1로 설정합니다. 모든 프로세스를 검사하여 남은 실행 시간이 0보다 크고 도착 시간이 현재 시간 이하인 프로세스를 선택합니다. 선택 기준은 남은 실행 시간이 가장 긴 프로세스입니다. 선택된 프로세스를 실행하여 간트 차트에 표시하고프로세스의 남은 실행 시간을 감소시키고 현재 시간을 증가시킵니다. 프로세스가 완료 시간, 반환 시간, 대기 시간을 계산하고 완료된 프로세스 수를 증가시킵니다. 선택된 프로세스가 없는 경우 IDLE 상태로 표시하고 현재 시간을 증가시킵니다. 모든 프로세스가 완료되면 종료 시간을 출력합니다.

4. priority (non preemptive&preemptive)

4.1 라이브러리 포함 및 변수 선언

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

// priority프로세스 구조체
struct proc {
    char pid[3];
    int parrival, pCPU_burst, ppriority, pwaiting_time, pturnaroud_time, premaining_time, pcompleted;
};
// 동적 메모리 할당을 위한 구조체 배열 포인터 선언
struct proc* p;
int process_num;
```

- 프로그램에서 필요한 헤더 파일을 포함하고 프로세스 구조체에서 프로세스의 ID, 도착 시간, 실행 시간, 우선순위, 대기한 시간, 반환 시간, 남은 실행 시간, 완료 여부를 저장합니다.

4.2 프로세스 정보를 파일에서 읽어오는 함수

```c
void read_processes(void) {
    process_num = 0;
    p = malloc(sizeof(struct proc) * 10); // 초기 크기로 메모리 할당
    int capacity = 10;

    while (scanf("%s", p[process_num].pid) != EOF && strcmp(p[process_num].pid, "E") != 0) {
        scanf("%d", &p[process_num].parrival);
        scanf("%d", &p[process_num].pCPU_burst);
        if (scanf("%d", &p[process_num].ppriority) != 1) {
            p[process_num].ppriority = 0;  // 우선순위가 없으면 0으로 설정
        }
        p[process_num].premaining_time = p[process_num].pCPU_burst;
        p[process_num].pcompleted = 0;
        process_num++;

        if (process_num >= capacity) {
            capacity *= 2;
            p = realloc(p, sizeof(struct proc) * capacity); // 필요 시 메모리 크기 확장
        }
    }
}
```

- process_num 변수를 0으로 초기화하고 프로세스 정보를 저장할 동적 메모리 배열 p를 초기 크기 10으로 할당하고 capacity 변수에 배열의 현재 용량을 저장합니다. scanf를 사용하여 stdin에서 프로세스 ID를 읽어 입력이 EOF이거나 ID가 “E” 이면 루프를 종료합니다. 도착시간, 실행 시간, 우선순위를 순차적으로 읽어오고 우선순위를 읽지 못한 경우 우선순위를 0으로 설정합니다. 남은 시간과 완료 상태를 초기화하고 읽어온 프로세스를 배열에 추가하고 process_num을 증가시킵니다. 프로세스 수가 현재 용량을 초과하면 용량을 두 배로 늘립니다. realloc을 사용하여 배열의 크기를 확장합니다.

4.3 Gantt 차트를 출력하는 함수

```
void print_Gantt(int start, int end, int idx) {
    if (idx == -1) {
        printf("%d: IDLE₩n", start);
    }
    else {
        printf("%d: %s (%d)₩n", start, p[idx].pid, end - start);
    }
}
```

- 주어진 시간 구간 동안 특정 프로세스의 실행 상태를 출력하고 프로세스가 없으면 “IDLE” 상태를 출력합니다.

4.4 모든 프로세스의 우선순위가 같은지 확인하는 함수

```
int all_priorities_equal(void) {
    for (int i = 1; i < process_num; i++) {
        if (p[i].ppriority != p[0].ppriority) {
            return 0;
        }
    }
    return 1;
}
```

- 함수는 반환형이 int이며 입력 매개변수가 없습니다. 반환 값은 모든 프로세스의 우선순위가 같으면 1 그렇지 않다면 0입니다. 인덱스 1부터 시작하여 process_num보다 작은 동안 반복합니다. 인덱스를 1부터 시작하는 이유는 첫 번째 프로세스를 기준으로 다른 프로세스와 우선순위를 비교하기 위해서입니다. 현재 프로세스(p[i])의 우선순위가 첫 번째 프로세스(p[0])의 우선순위와 다른 경우 함수는 0을 반환하고 종료합니다. 이는 우선순위가 동일하지 않음을 의미합니다. 루프가 종료될 때까지 모든 프로세스의 우선순위가 같다면, 함수는 1을 반환합니다.

4.5 비선점형 우선순위 스케줄링 함수

```
void Priority_NonPreemptive(int* total_pcompleted_time, int* total_pturnaroud_time, int* total_pwaiting_time) {
    int time = 0;
    int finished = 0;

    printf("₩nPriority (non preemptive)₩n");
    while (finished < process_num) {
        int idx = -1;

        for (int i = 0; i < process_num; i++) {
            if (p[i].premaining_time == 0) continue;

            if (p[i].parrival <= time) {
                if (idx == -1 || (all_priorities_equal() && p[i].parrival < p[idx].parrival) ||
                    (!all_priorities_equal() && (p[i].ppriority > p[idx].ppriority ||
                        (p[i].ppriority == p[idx].ppriority && p[i].parrival < p[idx].parrival)))) {
                    idx = i;
                }
            }
        }

        if (idx == -1) {
            time++;
            continue;
        }

        print_Gantt(time, time + p[idx].premaining_time, idx);

        p[idx].pwaiting_time = time - p[idx].parrival;
        p[idx].pturnaroud_time = time - p[idx].parrival + p[idx].premaining_time;
        time += p[idx].premaining_time;
        p[idx].premaining_time = 0;
        p[idx].pcompleted = 1;
        finished++;
    }
    printf("%d: E₩n", time);

    // 전체 완료 시간 설정
    *total_pcompleted_time = time;

    for (int i = 0; i < process_num; i++) {
        *total_pturnaroud_time += p[i].pturnaroud_time;
        *total_pwaiting_time += p[i].pwaiting_time;
    }
}
```

- 함수는 세 개의 포인터 매개변수를 받고 현재 시간을 나타내는 time을 0으로 초기화하고 완료된 프로세스 수를 나타내는 finished를 0으로 초기화하고 스케줄링 알고리즘의 제목을 출력합니다. 완료된 프로세스 수가 전체 프로세스 수보다 적을 때까지 반복합니다. idx는 현재 실행할 프로세스의 인덱스를 나타내며, 초기값은 -1로 설정합니다. 모든 프로세스를 검사하여 아직 완료되지 않았고 도착 시간이 현재 시간 이하인 프로세스를 선택합니다. 선택 기준은 우선순위가 높은 프로세스이거나 우선순위가 같을 경우 도착 시간이 빠른 프로세스입니다. 선택된 프로세스가 없는 경우 시간을 1증가시키고 다음 루프로 이동합니다. 선택된 프로세스를 실행하여 간트 차트에 표시합니다. 대기 시간과 반환 시간을 계산하고 실행 시간을 업데이트합니다. 프로세스 완료되었음을 표시하고 완료된 프로세스 수를 증가시킵니다. 모든 프로세스가 완료되면 현재 시간을 출력하여 전체 완료 시간을 설정합니다. 각 프로세스의 반환 시간과 대기 시간을 합산하여 총 반환 시간과 총 대기 시간을 계산합니다.

4.6 선점형 우선순위 스케줄링 함수

```
void Priority_Preemptive(int* total_pcompleted_time, int* total_pturnaroud_time, int* total_pwaiting_time) {
    int time = 0;
    int finished = 0;
    int prev_idx = -1;
    int start_time = 0;

    printf("₩nPriority (preemptive)₩n");
    while (finished < process_num) {
        int idx = -1;

        for (int i = 0; i < process_num; i++) {
            if (!p[i].premaining_time) continue;

            if (p[i].parrival <= time) {
                if (idx == -1 || (all_priorities_equal() && p[i].parrival < p[idx].parrival) ||
                    (!all_priorities_equal() && (p[i].ppriority > p[idx].ppriority ||
                        (p[i].ppriority == p[idx].ppriority && p[i].parrival < p[idx].parrival)))) {
                    idx = i;
                }
            // 모든 우선순위가 동일한지 체크 후 그에 따른 조치
            if (all_priorities_equal()) {
                if (idx == -1) {
                    time++;
                    continue;
                }

                if (prev_idx != idx) {
                    if (prev_idx != -1) {
                        print_Gantt(start_time, time, prev_idx);
                    }
                    start_time = time;
                    prev_idx = idx;
                }

                p[idx].premaining_time--;
                time++;

                if (p[idx].premaining_time == 0) {
                    print_Gantt(start_time, time, idx);
                    p[idx].pturnaroud_time = time - p[idx].parrival;
                    p[idx].pwaiting_time = p[idx].pturnaroud_time - p[idx].pCPU_burst;
                    p[idx].pcompleted = 1;
                    prev_idx = -1;
                    start_time = time;
                    finished++;
                }
            }
            else {
                if (idx == -1) {
                    time++;
                    continue;
                }

                if (prev_idx != idx) {
                    if (prev_idx != -1) {
                        print_Gantt(start_time, time, prev_idx);
                    }
                    start_time = time;
                    prev_idx = idx;
                }

                p[idx].premaining_time--;
                time++;

                if (p[idx].premaining_time == 0) {
                    print_Gantt(start_time, time, idx);
                    p[idx].pturnaroud_time = time - p[idx].parrival;
                    p[idx].pwaiting_time = p[idx].pturnaroud_time - p[idx].pCPU_burst;
                    p[idx].pcompleted = 1;
                    prev_idx = -1;
                    start_time = time;
                    finished++;
                }
            }
        }
        printf("%d: E₩n", time);
        // 전체 완료 시간 설정
        *total_pcompleted_time = time;

        for (int i = 0; i < process_num; i++) {
            *total_pturnaroud_time += p[i].pturnaroud_time;
            *total_pwaiting_time += p[i].pwaiting_time;
        }
    }
```

- 함수는 세 개의 포인터 매개변수를 받고 현재 시간을 나타내는 time을 0으로 초기화하고 완료된 프로세스 수를 나타내는 finished를 0으로 초기화하고 이전에 실행된 프로세스의 인덱스를 나타내는 prev_idx를 -1로 초기화하고 프로세스 실행 시작 시간을 나타내는 start_time을 0으로 초기화하고 스케줄링 알고리즘의 제목을 출력합니다. 완료된 프로세스 수가 전체 프로세스 수보다 적을 때까지 반복합니다. idx는 현재 실행할 프로세스의 인덱스를 나타내며 초기값을 -1로 설정합니다. 모든 프로세스를 검사하여 아직 완료되지 않았고 도착 시간이 현재 시간 이하인 프로세스를 선택합니다. 선택 기준은 우선순위가 높은 프로세스이거나 우선순위가 같을 경우 도착 시간이 빠른 프로세스입니다. 우선순위가 동일한 경우와 그렇지 않은 경우에 따라 조건을 검사하고 프로세스를 선점적으로 실행합니다. 이전에 실행된 프로세스와 현재 프로세스가 다를 경우 간트 차트를 업데이트합니다. 실행 중인 프로세스의 남은 시간을 감소시키고 완료된 경우 대기 시간과 반환 시간을 계산합니다. 프로세스가 완료되었음을 표시하고 완료된 프로세스 수를 증가시킵니다. 모든 프로세스가 완료되면 현재 시간을 출력하여 전체 완료 시간을 설정합니다 각 프로세스의 반환 시간과 대기 시간을 합산하여 총 반환 시간과 총 대기 시간을 산합니다.

5. RR

5.1 라이브러리 포함 및 전역 변수 선언

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <limits.h>

#define MAX_THREADS 100  // 최대 스레드 수
#define MAX_OUTPUT 1000 // 출력 저장을 위한 최대 버퍼 크기

typedef struct {
    char id[10];   // 스레드 ID
    int arrival_time;      // 도착 시간
    int burst_time;      // 실행 시간
    int priority;         // 우선순위
    int remaining_time;   // 남은 실행 시간 (선점형 알고리즘에서 사용)
    int completion_time;  // 완료 시간
} Thread;


Thread threads[MAX_THREADS];    // 스레드 배열
int thread_count = 0;    // 스레드 개수
char output_non_preemptive[MAX_OUTPUT] = "";  // 비선점 출력 저장
char output_preemptive[MAX_OUTPUT] = "";      // 선점 출력 저장
double results[2][3];                         // sjf결과 저장: [스케줄링][완료, 턴어라운드, 대기]
double fresults[3];                           //fcfs 결과 저장
double rresults[3];                           // rr 결과 저장
double ljf_results[2][3];       // LJF 결과 저장: [스케줄링 유형][완료 시간, 턴어라운드, 대기]
```

- 프로그램에서 필요한 표준 라이브러리를 포함하고 Thread 구조체에 스레드의 ID, 도착 시간, 실행 시간, 우선순위, 남은 시간 및 완료 시간을 저장하고 전역 변수는 스레드 배열, 스레드 개수, 결과를 저장하는 배열을 선언합니다.

5.2 데이터 로드 함수

```c
void load_threads() {
    while (1) {
        Thread t;
        if (fscanf(stdin, "%s", t.id) != 1 || strcmp(t.id, "E") == 0) {
            break;
        }
        fscanf(stdin, "%d %d %d", &t.arrival_time, &t.burst_time, &t.priority);  // 도착 시간, 실행 시간, 우선순위 읽기
        t.remaining_time = t.burst_time;  // 초기 남은 시간 설정
        t.completion_time = 0; // 초기 완료 시간 설정
        threads[thread_count++] = t;  // 스레드 배열에 추가
    }
}
```

- stdin에서 데이터를 읽어와 스레드 정보를 로드함수입니다. 스레드 ID를 읽고 도착 시간, 실행 시간, 우선순위를 읽어와 스레드 배열에 저장합니다. 스레드의 초기 남은 시간과 완료 시간을 설정합니다. “E”가 입력되면 루프를 종료합니다.

5.3 도착 시간 정렬 함수

```
void sort_by_arrival_time() {
    for (int i = 0; i < thread_count - 1; i++) {
        for (int j = i + 1; j < thread_count; j++) {
            if (threads[i].arrival_time > threads[j].arrival_time) {  // 도착 시간이 빠른 순으로 정렬
                Thread temp = threads[i];
                threads[i] = threads[j];
                threads[j] = temp;
            }
        }
    }
}
```

- 스레드 배열을 도착 시간 기준으로 정렬합니다.

5.4 결과 계산 함수

```
void calculate_results_rr() {
    int total_turnaround_time = 0;
    int total_waiting_time = 0;
    int final_completion_time = 0;
    for (int i = 0; i < thread_count; i++) {
        int turnaround_time = threads[i].completion_time - threads[i].arrival_time;
        int waiting_time = turnaround_time - threads[i].burst_time;
        total_turnaround_time += turnaround_time;
        total_waiting_time += waiting_time;
        if (threads[i].completion_time > final_completion_time) {
            final_completion_time = threads[i].completion_time;
        }
    }
    rresults[0] = final_completion_time;                          // 전체 완료 시간
    rresults[1] = (double)total_turnaround_time / thread_count;   // 평균 Turnaround Time
    rresults[2] = (double)total_waiting_time / thread_count;      // 평균 Waiting Time
}
```

- 전체 완료시간과 각 스레드의 Turnaround Time, Waiting Time을 계산하여 결과 배열에 저장합니다.

5.5 Round Robin 스케줄링 함수

```
void round_robin() {
    int quantum = 1;  // 타임 슬라이스를 1로 고정
    int time = 0;          // 현재 시간
    int completed = 0;     // 완료된 프로세스 수
    int progress = 0;      // 실행 여부 플래그
    printf("₩nRR(quantum=%d):₩n", quantum);
    while (completed < thread_count) {
        progress = 0;
        for (int i = 0; i < thread_count; i++) {
            if (threads[i].remaining_time > 0 && threads[i].arrival_time <= time) {
                progress = 1;
                // 실행 시간 결정
                int execution_time = (threads[i].remaining_time > quantum) ? quantum : threads[i].remaining_time;
                // 실행 로그 출력
                printf("%d: %s (%d)₩n", time, threads[i].id, execution_time);
                // 시간 증가 및 남은 실행 시간 감소
                threads[i].remaining_time -= execution_time;
                time += execution_time;
                // 프로세스 완료 시 처리
                if (threads[i].remaining_time == 0) {
                    threads[i].completion_time = time;
                    completed++;
                }
            }
        }
        // 실행 가능한 프로세스가 없으면 IDLE 상태
        if (!progress) {
            printf("%d: IDLE₩n", time);
            time++;
        }
    }
    printf("%d: E₩n", time);
    // 결과 계산
    calculate_results_rr();
}
```

- 현재 시간, 완료된 프로세스수, 실행 여부 플래그 초기화를 통해 Round Robin 스케줄링을 위한 변수들을 초기화합니다. 그리고 주어진 quantum 시간만큼 각 스레드를 실행하고, 남은 실행 시간을 감소시킵니다. 프로세스가 완료되면 완료 시간을 기록하고 완료된 프로세스 수를 증가시킵니다. 실행 가능한 프로세스가 없으면 IDLE 상태로 시간을 증가시킵니다. 스케줄링이 완료된 후 결과를 계산합니다.

7. 메인함수

7.1 메인 함수의 시작 부분

```
int main() {

    int nonpreemptive_pcompleted = 0, nonpreemptive_turnaround = 0, nonpreemptive_waiting = 0;
    int preemptive_pcompleted = 0, preemptive_turnaround = 0, preemptive_waiting = 0;

    load_threads();
    sort_by_arrival_time();
```

- 비선점형 및 선점형 스케줄링의 완료 시간, 반환 시간, 대기 시간을 저장할 변수를 선언합니다.
- 스레드 데이터를 로드하고 도착 시간 기준으로 정렬합니다.

7.2 파일 출력 설정

```
    // 파일에 출력하기 전에 파일 포인터를 지정
    FILE* file = fopen("cpuschedule_result.txt", "w");
    if (file == NULL) {
        fprintf(stderr, "파일을 열 수 없습니다.₩n");
        return 1;
    }

    // 화면 출력도 그대로 두고, 파일로도 저장되도록 설정
    FILE* original_stdout = stdout; // 원래의 stdout을 저장합니다.
    stdout = file; // stdout을 파일로 리디렉션
```

- 파일 포인터 지정하여 결과를 저장할 파일을 엽니다. 화면 출력도 그대로 두고, 파일로도 저장되록 설정하고 원래의 stdout을 저장하고 stdout을 파일로 리디렉션합니다.

6. 파일 열기

```c
void print_file_contents(const char* filename) {
    FILE* file = fopen(filename, "r");  // 읽기 모드로 파일 열기
    if (file == NULL) {
        fprintf(stderr, "파일을 열 수 없습니다.₩n");
        return;
    }

    char ch;
    while ((ch = fgetc(file)) != EOF) {  // 파일 끝까지 한 문자씩 읽기
        putchar(ch);  // 읽은 문자를 화면에 출력
    }

    fclose(file);  // 파일 닫기
}
```

.- fopen 함수로 filename 에 해당하는 파일을 읽기 모드로 엽니다. 파일을 여는 데 실패하는 경우 오류메시지를 출력하고 함수를 종료합니다. fgetc함수로 파일에서 한 문자씩 읽어들입니다. EOF을 만날 때까지 계속 읽습니다. 읽어들인 문자를 putchar 함수로 화면에 출력합니다. fclose 함수로 파일을 닫아 리소스를 해제합2023111399니다..

7.3 FCFS, SJF(non-preemptive & preemptive), Round Robin 실행

```
    fcfs();

    // SJF Non-Preemptive
    for (int i = 0; i < thread_count; i++) {
        threads[i].remaining_time = threads[i].burst_time;
        threads[i].completion_time = 0;
    }
    sjf_non_preemptive();

    // 초기화 후 SJF Preemptive
    for (int i = 0; i < thread_count; i++) {
        threads[i].remaining_time = threads[i].burst_time;
        threads[i].completion_time = 0;
    }
    sjf_preemptive();

    for (int i = 0; i < thread_count; i++) {
        threads[i].remaining_time = threads[i].burst_time;
        threads[i].completion_time = 0;
    }
    // Round Robin 실행
    round_robin();
```

- FCFS 실행, SJF(non-preemptive)실행, 초기화 후 SJF(preemptive)실행, Round Robin을 실행합니다.

7.4 priority (non preemptive & preemptive), LJF (non preemptive & preemptive) 실행

```
freopen(NULL, "r", stdin);

// stdin에서 데이터를 읽어오기
read_processes();
Priority_NonPreemptive(&nonpreemptive_pcompleted, &nonpreemptive_turnaround, &nonpreemptive_waiting);

// stdin에서 다시 데이터를 읽어오기
freopen(NULL, "r", stdin);  // stdin을 다시 읽기 위해 초기화
read_processes();
Priority_Preemptive(&preemptive_pcompleted, &preemptive_turnaround, &preemptive_waiting);

// LJF 스케줄링 추가 실행
freopen(NULL, "r", stdin);  // LJF 데이터 재로딩
load_ljf_processes();
ljf_non_preemptive();
ljf_preemptive();
```

- stdin 에서 데이터를 읽어오고 priority (non preemptive)실행하고 stdin에서 다시 데이터를 읽어오고 stdin을 다시 읽기 위해 초기화하고 priority (preemptive)실행, stdin을 다시 읽기 위해 초기화 후 프로세스 데이터를 로드하고 LJF (non preemptive) 실행, LJF (preemptive)를 실행합니다.

7.5 결과 출력

```
printf("%s₩n", output_non_preemptive);
printf("%s₩n", output_preemptive);

printf("Results               Total Completion Time   Turnaround Time   Waiting Time₩n");
printf("-------------------------------------------------------------------------------₩n");
printf("FCFS                %.2f                %.2f             %.2f₩n",
    fresults[0], fresults[1], fresults[2]);
printf("SJF (non-preemptive)   %.2f                %.2f             %.2f₩n",
    results[0][0], results[0][1], results[0][2]);
printf("SJF (preemptive)      %.2f                %.2f             %.2f₩n",
    results[1][0], results[1][1], results[1][2]);
printf("priority (non preemptive)₩t%.2f₩t₩t%.2f₩t₩t%.2f₩n",
    (float)nonpreemptive_pcompleted,
    (float)nonpreemptive_turnaround / process_num,
    (float)nonpreemptive_waiting / process_num);
printf("priority (preemptive)₩t₩t%.2f₩t₩t%.2f₩t₩t%.2f₩n",
    (float)preemptive_pcompleted,
    (float)preemptive_turnaround / process_num,
    (float)preemptive_waiting / process_num);
printf("RR          %.2f                %.2f             %.2f₩n",
    rresults[0], rresults[1], rresults[2]);
printf("LJF (non-preemptive)   %.2f                %.2f             %.2f₩n",
    ljf_results[0][0], ljf_results[0][1], ljf_results[0][2]);
printf("LJF (preemptive)      %.2f                %.2f             %.2f₩n",
    ljf_results[1][0], ljf_results[1][1], ljf_results[1][2]);
```

- 각 스케줄링 알고리즘의 결과, 총 완료 시간, 평균 반환 시간, 평균 대기 시간을 출력합니다.

7.6 stdout 복원 및 메모리 해제

```
    // stdout을 원래 화면으로 복원
    stdout = original_stdout;
    // 파일로도 저장이 되었고 화면에도 출력되는 결과 확인
    fclose(file); // 파일 닫기

    free(p); // 동적 메모리 해제

    print_file_contents("cpuschedule_result.txt");



    return 0;
}
```

- stdout을 원래 화면으로 복원하여 파일로도 저장이 되었고 화면에도 출력되는 결과를 확인하고 파일을 닫고 동적 메모리를 해제합니다. 결과 파일의 내용 출력합니다.