CPU Scheduling Simulator Report

제출일	2023-06-05	전공	컴퓨터소프트웨어공학과
과목	운영체제	학번	20194024
담당 교수님	김대영 교수님	이름	김민욱

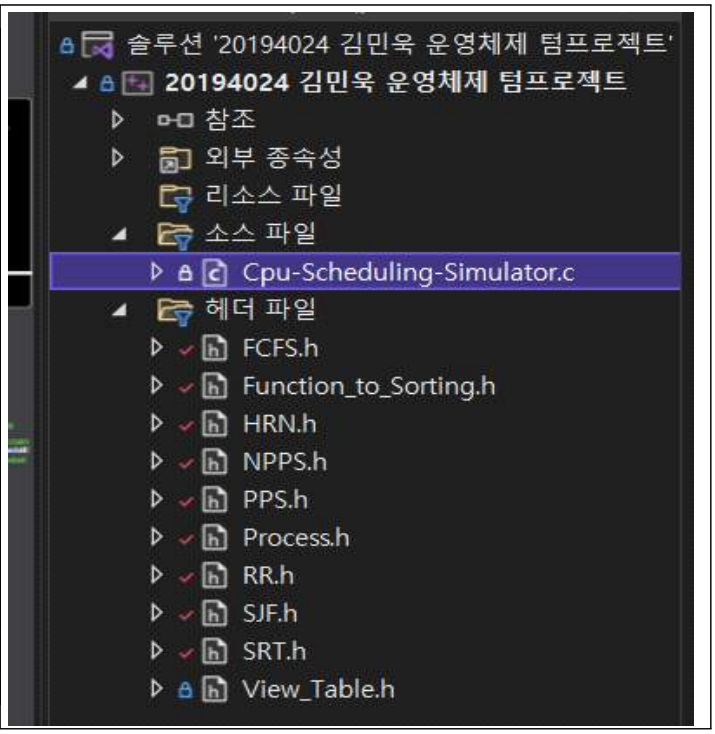
| 목 차 |

1. 프로젝트 기본 소스 코드

1.1 Cpu-Scheduling-Simulator.c

헤더 선언 및 소스 파일 구성

```c
#include <stdio.h>
#include <stdlib.h>
#include "Process.h"
#include "FCFS.h"
#include "SJF.h"
#include "NPPS.h"
#include "HRN.h"
#include "PPS.h"
#include "RR.h"
#include "SRT.h"
#include "Function_to_Sorting.h"

#pragma warning(disable:4996)
```



메인 함수

```c
int main() {
    int process_count = 0; // 프로세스 갯수를 저장할 변수

    Quantum quantum; // 퀀텀 변수 선언

    FILE* fp = fopen("Process.txt", "r"); // 파일 읽기 모드로 열기

    /* 파일 열기에 실패한 경우 */
    if (fp == NULL) {
        fprintf(stderr, "File Open Fail₩n");
        exit(1);
    }

    fscanf(fp, "%d", &process_count); // 파일에서 읽은 프로세스 갯수 저장
```

파일 읽기에 실패한 경우 오류 처리를 통해 파일을 열지 못했다는 문구 출력

```
Process* process = (Process*)malloc(sizeof(Process) * process_count); // 프로세스 개수 만큼 동적할당

if (process == NULL) {
    fprintf(stderr, "메모리 할당 실패₩n");
    exit(1);
}

for (int i = 0; i < process_count; i++) { // 파일에서 프로세스id, 도착시간, 반환시간, 우선순위를 받아 구조체에 저장
    fscanf(fp, "%s %d %d %d", process[i].id, &process[i].arrival_time, &process[i].run_time, &process[i].priority);
    // printf("%s %d %d %d₩n", process[i].id, process[i].arrival_time, process[i].run_time, process[i].priority);
}

fscanf(fp, "%d", &quantum); // 파일에서 읽은 퀀텀 개수 저장
// printf("%d", quantum);
```

Process 구조체가 헤더에 선언되어 있으므로 Process 구조체에 있는 멤버 사용가능하고 멤버에 대하여 파일에서 읽은 프로세스 개수만큼 동적 할당한다. 할당된 메모리 만큼 파일에 있는 데이터를 구조체 배열에 저장한다. 마지막으로 파일에서 읽은 정수형 데이터(퀀텀)를 퀀텀 개수로 저장한다.

각 스케줄링에 대하여 함수 호출

```
FCFS_Scheduling(process, process_count);

SJF_Scheduling(process, process_count);

NPPS_Scheduling(process, process_count);

HRN_Scheduling(process, process_count);

PPS_Scheduling(process, process_count);

RR_Scheduling(process, process_count,quantum);

SRT_Scheduling(process, process_count);

free(process);
```

각 스케줄링에 대하여 함수 호출하고 동적 할당한 process 구조체 배열에 대하여 메모리를 해제시킨다.

1.2 Process.h

프로세스 구조체 선언

```
typedef struct _process { // 프로세스 구조체 선언
    char id[20]; // 프로세스 id - File
    int waiting_time; // 프로세스 대기시간
    int arrival_time; // 프로세스 도착시간 - File
    int run_time; // 프로세스 실행시간 - File
    int priority; // 프로세스 우선순위 - File
    int response_time; // 프로세스 응답시간
    int return_time; // 프로세스 반환시간
    int turnAround_time; // 프로세스 소요시간
    bool completed;
}Process;
```

파일에서 받아와 데이터를 저장할 변수는 프로세스 id, 도착시간, 실행시간, 우선순위이다. 구조체에 필요한 변수는 대기 시간, 응답시간, 반환시간, 소요 시간 그리고 프로세스 완료를 확인할 Bool 변수(True & False)이다.

Quantum 키워드 선언

```
typedef int Quantum; // 시간 할당량 Quantum 키워드 사용
```

프로세스 초기화 함수

```
void Process_init(Process p[], int n) // 프로세스 초기화 함수
{
    for (int i = 0; i < n; i++) { // 프로세스 갯수 만큼 반복
        p[i].waiting_time = 0; // 대기 시간 초기화
        p[i].response_time = 0; // 응답 시간 초기화
        p[i].turnAround_time = 0; // 반환 시간 초기화
        p[i].completed = false;
    }
}
```

1.3 Funtion_To_Sorting

모든 정렬에는 stdlib.h 라이브러리 함수에 들어있는 퀵 정렬 함수를 사용하였고 원형은 다음과 같다.

```
void qsort(void *base, int num, int width, int (*compare)(const void *, const void *));
```

base : 시작 주소

num : 배열 요소의 개수

width : 배열 요소 하나의 크기

compare : 비교함수

- 포인터를 통하여 두 개의 요소를 비교하여 비교 결과를 정수로 변환

반환 값

<0 : 요소1이 요소2보다 작으면

0 : 요소1과 요소2가 같으면

>0 : 요소1이 요소2보다 크면

● 반환시간 퀵 정렬 함수

```
// 반환시간 퀵정렬 매개변수로 들어갈 함수 - 오름차순
int compare_return_time(const void* a, const void* b){
    Process p1 = *(Process*)a;
    Process p2 = *(Process*)b;

    if (p1.return_time < p2.return_time)
        return -1;

    if (p1.return_time > p2.return_time)
        return 1;

    return 0;
}
// 반환시간 기준으로 (퀵)정렬할 함수
void qsort_return_time(Process* p, int n) {
    qsort(p, n, sizeof(Process), compare_return_time);
}
```

프로세스의 반환시간을 기준으로 오름차순 정렬하는 퀵정렬 함수를 재정의 하였다.

● 우선순위 퀵 정렬 함수

```
// 우선순위 퀵정렬 매개변수로 들어갈 함수 - 오름차순
int compare_priority_time(const void* a, const void* b){
    Process p1 = *(Process*)a;
    Process p2 = *(Process*)b;

    if (p1.priority < p2.priority)
        return -1;

    if (p1.priority > p2.priority)
        return 1;

    if (p1.priority == p2.priority) {
        if (p1.arrival_time < p2.arrival_time)
            return -1;
        if (p1.arrival_time > p2.arrival_time)
            return 1;
    }
    else
        return 0;
}
// 우선순위를 기준으로 (퀵)정렬할 함수
void qsort_priority_time(Process* p, int n) {
    qsort(p, n, sizeof(Process), compare_priority_time);
}
```

우선순위를 기준으로 퀵 정렬 함수를 재정의하였다. 우선순위가 같은 경우 도착시간을 기준으로 먼저 도착하면 음수를 반환하도록 재정의하였다.

● 실행시간 퀵 정렬 함수

```
// 실행시간 퀵정렬 매개변수로 들어갈 함수 - 오름차순
int compare_run_time(const void* a, const void* b) {
    Process p1 = *(Process*)a;
    Process p2 = *(Process*)b;

    if (p1.run_time < p2.run_time)
        return -1;

    if (p1.run_time > p2.run_time)
        return 1;

    return 0;
}
// 실행시간을 기준으로 (퀵)정렬할 함수
void qsort_run_time(Process* p, int n) {
    qsort(p, n, sizeof(Process), compare_run_time);
}
```

실행시간을 기준으로 퀵 정렬 함수를 재정의하였다.

● 도착시간 퀵 정렬 함수

```
// 도착시간 퀵정렬 매개변수로 들어갈 함수 - 오름차순
int compare_arrival_time(const void* a, const void* b) {
    Process p1 = *(Process*)a;
    Process p2 = *(Process*)b;

    if (p1.arrival_time < p2.arrival_time)
        return -1;

    if (p1.arrival_time > p2.arrival_time)
        return 1;

    return 0;
}

// 도착시간을 기준으로 (퀵)정렬할 함수
void qsort_arrival_time(Process* p, int n) {
    qsort(p, n, sizeof(Process), compare_arrival_time);
}
```

도착시간을 기준으로 퀵 정렬을 재정의 하였다.

2. First Come First Served Scheduling (선입선출 스케줄링)

FCFS 스케줄링은 준비 큐에 도착한 순서대로 CPU를 할당하는 비선점형 방식이다. 한 번 실행되면 그 프로세스가 끝나야 다음 프로세스를 실행할 수 있다. 모든 우선순위는 동일하다. 비선점 스케줄링을 사용하고 있어서 오버헤드가 낮다는 장점이 있지만, 긴 수행시간을 가지고 있는 프로세스가 자원을 점유하고 있는 경우 다른 프로세스들이 그만큼 대기 시간을 갖게 된다. 이러한 현상을 Convey Effect라 한다.

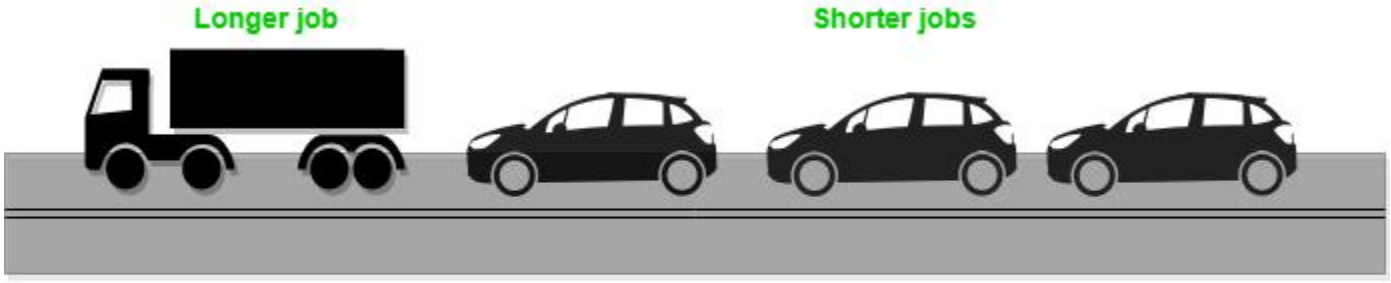


Figure - The Convey Effect, Visualized

[사진 출처]

https://velog.io/@leeesangheee/%EC%9A%B4%EC%98%81%EC%B2%B4%EC%A0%9C.-CPU-%EC%8A%A4%EC%BC%80%EC%A4%84%EB%9F%AC

FCFS 스케줄링 함수

```
void FCFS_Scheduling(Process* p, int pc) {
    int total_waiting_time = 0; // 총 프로세스 대기 시간
    int total_turnAround_time = 0; // 총 프로세스 소요 시간
    int total_response_time = 0; // 총 프로세스 응답 시간
    int total_return_time = 0; // 총 프로세스 반환 시간

    Process_init(p, pc);

    qsort_arrival_time(p, pc);

    /* 맨 처음 들어온 프로세스 실행 */
    p[0].return_time = p[0].run_time; // 파일에서 받은 run_time을 return_time에 대입
    p[0].turnAround_time = p[0].return_time - p[0].arrival_time; // 반환시간 = 실행시간 - 대기시간
    p[0].response_time = 0; // 응답시간 초기화
    p[0].waiting_time = 0; // 대기시간 초기화

    /* 실행된 프로세스 만큼 사용률 증가 */
    total_waiting_time += p[0].waiting_time; // 총 대기시간 증가
    total_turnAround_time += p[0].turnAround_time; // 총 소요시간 증가
    total_response_time += p[0].response_time; // 총 응답시간 증가
    total_return_time += p[0].run_time; // 총 실행시간 증가
```

매개변수로는 프로세스 포인터와 프로세스 개수를 받아와 프로세스 초기화 함수를 통해 프로세스를 초기화한다. FCFS는 도착한 순서대로 작업하기 때문에 도착시간 순서로 정렬한 후 프로세스를 실행한 결과를 계산하면 된다.

```
// 선입 선출 구조로 들어오는 순서대로 프로세스 계산
for (int i = 1; i < pc; i++)
{
    /* 각 프로세스 계산 */
    p[i].waiting_time = total_return_time - p[i].arrival_time; // 각 프로세스 대기시간 = 총 반환시간 - 도착시간
    p[i].return_time = total_return_time + p[i].run_time; // 각 프로세스 반환시간 = 총 반환시간 + 실행시간
    p[i].turnAround_time = p[i].return_time - p[i].arrival_time; // 각 프로세스 소요시간 = 각 프로세스 반환시간 - 도착시간
    p[i].response_time += p[i].waiting_time; // 반응시간 += 대기시간

    /* 실행된 프로세스 만큼 사용률 증가 */
    total_return_time += p[i].run_time; // 총 반환시간 증가
    total_waiting_time += p[i].waiting_time; // 총 대기시간 증가
    total_turnAround_time += p[i].turnAround_time; // 총 소요시간 증가
    total_response_time += p[i].response_time; // 총 응답시간 증가
}
```

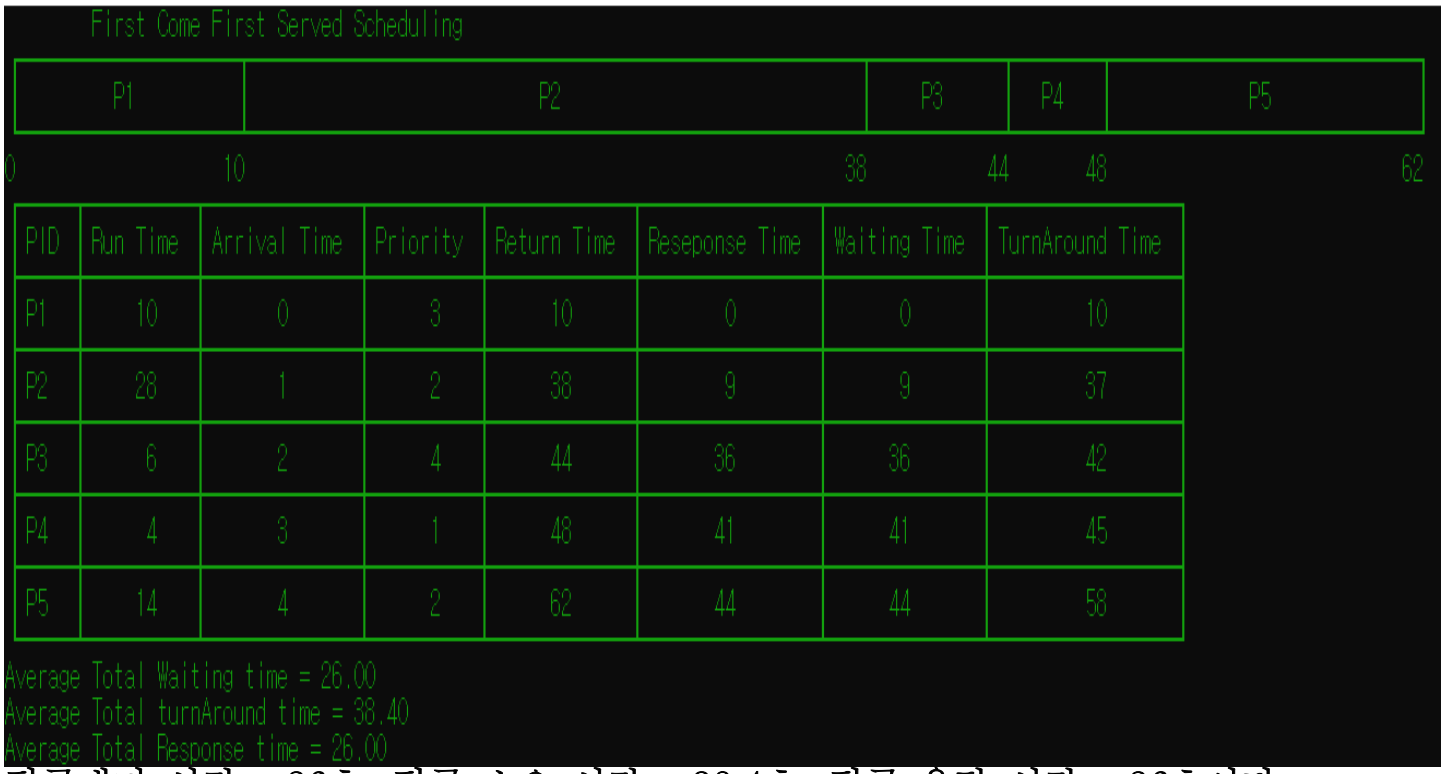
선입 선출 구조이므로 처음 실행한 프로세스를 제외한 나머지 프로세스들에 대하여 계산한다. 각 프로세스의 대기 시간은 증가한 총 프로세스의 반환시간에서 해당 프로세스의 도착시간을 뺀 값이다. 각 프로세스 반환시간은 증가한 총 프로세스의 반환시간에서 각 프로세스의 실행시간을 더한 값이다. 각 프로세스의 소요 시간은 각 프로세스 반환시간에서 각 프로세스 도착시간을 뺀 값이다. 각 프로세스 응답시간은 계산된 각 프로세스의 대기 시간을 더한 값이다.

스케줄링에서 구현하려고 하는 프로세스들의 정보는 아래와 같다.

Quantum = 2

PID	Arrival Time	Run Time	Priority
P1	0	10	3
P2	1	28	2
P3	2	6	4
P4	3	4	1
P5	4	14	2

FCFS 구조이므로 P1 → P2 → P3 → P4 → P5 순으로 작업이 진행된다.
간트 차트를 그려보면 다음과 같다.

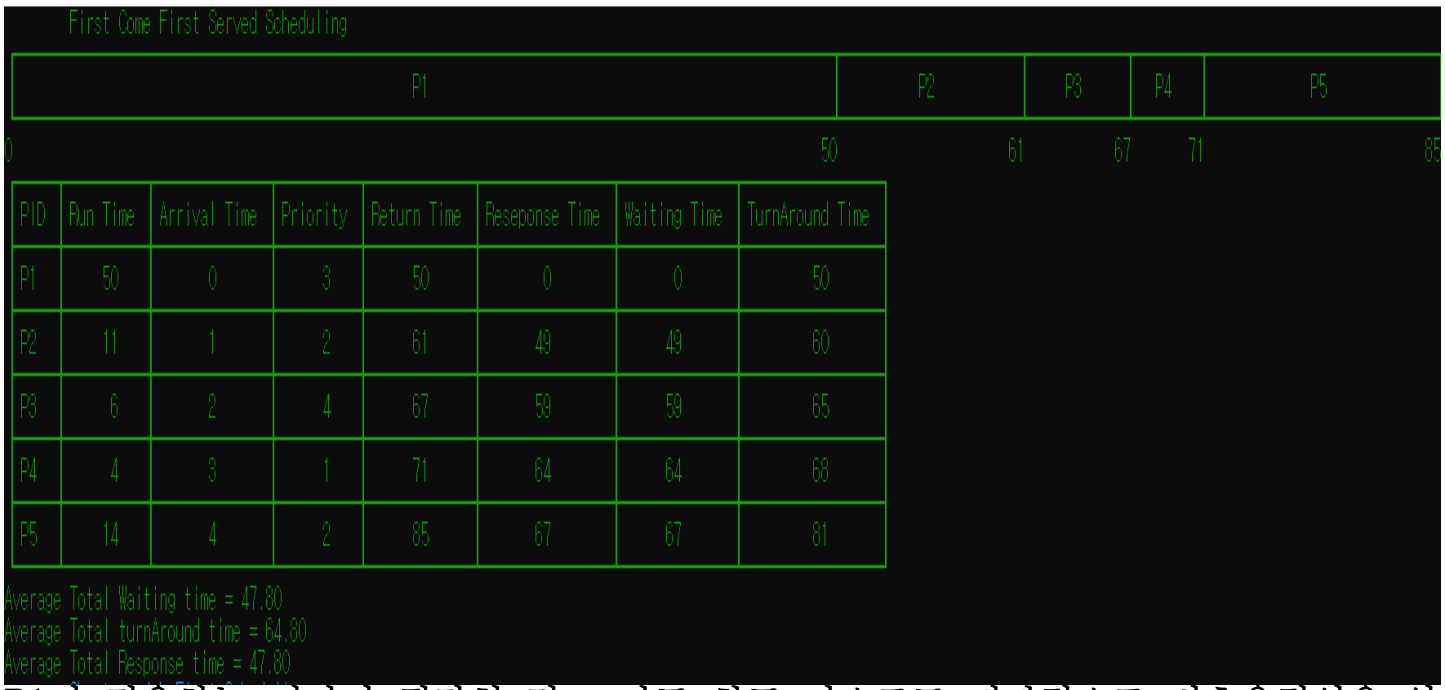


PID	Run Time	Arrival Time	Priority	Return Time	Reseponse Time	Waiting Time	TurnAround Time
P1	10	0	3	10	0	0	10
P2	28	1	2	38	9	9	37
P3	6	2	4	44	36	36	42
P4	4	3	1	48	41	41	45
P5	14	4	2	62	44	44	58

Average Total Waiting time = 26.00
Average Total turnAround time = 38.40
Average Total Response time = 26.00

평균대기 시간 : 26초, 평균 소요 시간 : 38.4초, 평균 응답 시간 : 26초이다.

Convey Effect를 직접 마주해 보기 위해서 도착시간이 가장 빠른 P1의 실행시간을 50, 다음 프로세스인 P2의 실행시간을 11로 수정하고 실행해 보았다.



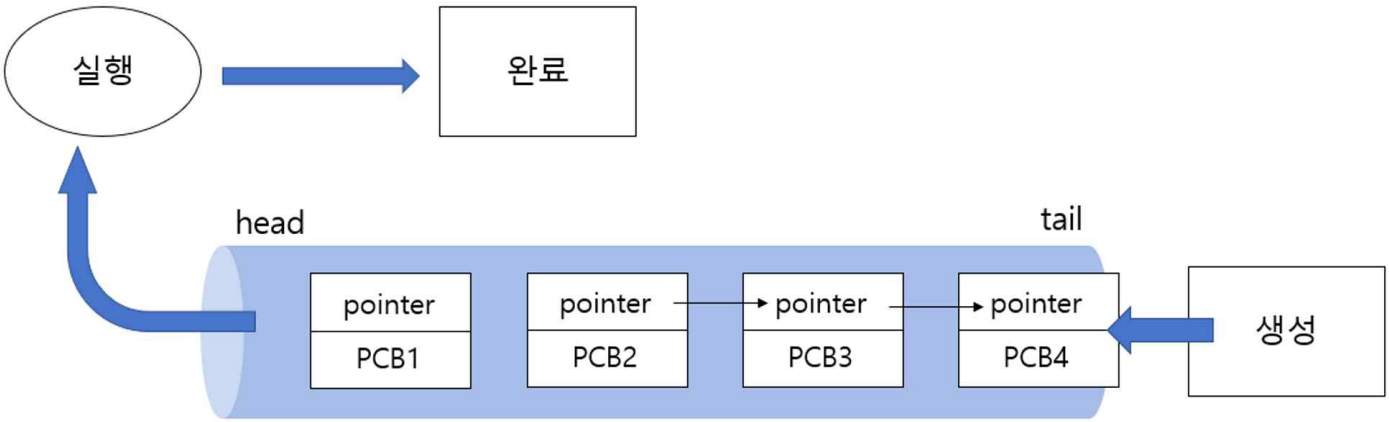
PID	Run Time	Arrival Time	Priority	Return Time	Reseponse Time	Waiting Time	TurnAround Time
P1	50	0	3	50	0	0	50
P2	11	1	2	61	49	49	60
P3	6	2	4	67	59	59	65
P4	4	3	1	71	64	64	68
P5	14	4	2	85	67	67	81

Average Total Waiting time = 47.80
Average Total turnAround time = 64.80
Average Total Response time = 47.80

P1이 점유하는 기간이 굉장히 길고 간트 찬트 만으로도 직관적으로 비효율적임을 알 수 있다. 또한 평균대기, 소요, 응답시간이 기하급수적으로 증가한 것을 볼 수 있다.

3. Shortest Job First Scheduling (최단 작업 우선 스케줄링)

SJF 스케줄링은 준비 큐에 있는 프로세스 중에서 실행시간이 가장 짧은 작업부터 CPU를 할당하는 비선점형 방식이다. Convey Effect를 완화하여 시스템의 효율성을 높인 스케줄링이다.



[사진 출처]

https://80000coding.oopy.io/fc28d9ac-e072-492c-a390-40ec8ebaac96

SJF 스케줄링 함수

● SJF_Process_Time(Process *p,int n);

```
void SJF_Process_Time(Process* p, int n) { // 프로세스 별 시간 계산 함수
    int current_time = 0; // 현재까지 누적 시간을 저장할 변수

    int min_index = 0; // 최소 시간을 가지는 인덱스 변수

    /* 도착시간이 가장 빠른 프로세스 먼저 실행 */
    p[0].return_time = p[0].run_time; // 반환시간 저장
    p[0].turnAround_time = p[0].run_time - p[0].arrival_time; // 소요시간 = 실행시간 - 도착시간
    p[0].waiting_time = 0; // 대기시간 0
    p[0].completed = true; // 완료 표시

    current_time = p[0].run_time; // 현재 시간 증가
```

각 프로세스의 시간을 계산하기 위한 함수이다. current_time 변수로 현재까지 누적된 실행시간을 저장한다. 실행시간이 가장 짧은 프로세스를 찾기 위한 min_index 변수를 선언한다.

최소 실행시간을 가지는 프로세스 인덱스를 찾는 과정이다. 가장 처음 들어온 프로세스는 작업이 진행되었으므로 제외하고 알고리즘을 수행한다. 이미 완료된 프로세스라면 탐색을 진행하고 아직 실행되지 않은 프로세스로 우선 업데이트한다. 업데이트된

```
/* p[0]는 진행하였으므로 인덱스 1부터 시작 */
for (int i = 1; i < n; i++) {
    for (int j = 1; j < n; j++) {
        if (p[j].completed == true) // 이미 완료된 프로세스
            continue;
        else {
            min_index = j; // 아직 실행하지 않은 프로세스
            break;
        }
    }

    for (int k = 1; k < n; k++) {
        /* 완료되지 않은 프로세스이고 도착시간이 현재 누적된 시간보다 적어야 한다.(프로세스 연기 되는 현상 해결) !
        if (p[k].completed == false && p[k].arrival_time < current_time && p[k].run_time < p[min_index].run_time)
            min_index = k;
        }
    }
```

프로세스로 다시 한번 탐색을 진행하는데 아직 완료되지 않았고 도착시간이 실행 중인 시간보다 작고 업데이트된 프로세스보다 실행시간이 작다면 최소 실행시간을 가지는 인덱스를 업데이트한다. 최종적으로 모든 조건을 만족하는 인덱스가 결정된다.

```
p[min_index].waiting_time = current_time - p[min_index].arrival_time; // 대기 시간 계산(=응답시간) : FCF
p[min_index].completed = true; // 완료 상태 갱신

current_time += p[min_index].run_time; // 누적 시간 갱신
p[min_index].return_time = current_time; // 반환 시간 갱신
p[min_index].turnAround_time = p[min_index].return_time - p[min_index].arrival_time; // 소요 시간 갱신
```

최소 실행시간을 가지는 프로세스에 대하여 각 프로세스의 상태를 갱신한다.

작업이 완료된 후 간트 차트를 그려보면 아래와 같다.
구현하려는 프로세스들의 정보는 항상 동일하다.

Shortest Job First Scheduling

P1	P4	P3	P5	P2

0 10 14 20 34 62

PID	Run Time	Arrival Time	Priority	Return Time	Reseponse Time	Waiting Time	TurnAround Time
P1	10	0	3	10	0	0	10
P4	4	3	1	14	7	7	11
P3	6	2	4	20	12	12	18
P5	14	4	2	34	16	16	30
P2	28	1	2	62	33	33	61

Average Total Waiting time = 13.60
Average Total turnAround time = 26.00
Average Total Response time = 13.60

평균대기 시간 : 13.6초, 평균 소요 시간 : 26초, 평균 응답 시간 : 13.6초이다.

FCFS 알고리즘보다 효율적인 것을 볼 수 있다. 하지만 프로세스와 프로세스 간 도착 시간에 공백이 있다면 실행시간 간에 공백이 생겨 CPU가 아무것도 할당하지 않을 경우가 존재한다. 또한 가장 최소 실행시간을 가지는 프로세스가 아직 도착하지 않았고 다른 프로세스가 도착하여 CPU를 점유하고 있는 경우에 도착한 프로세스의 작업을 마친 후 다음 작업을 해야 하므로 비선점형의 단점이 나타난다.

따라서 프로세스의 종료 시간을 정확하게 예측하기 어렵고 작업시간이 긴 프로세스는 계속해서 뒤로 밀리게 된다. 이는 아사 현상이라고 불린다.

아사 현상을 해결하기 위해 에이징 기법을 사용한다. 에이징 기법이란 프로세스가 양보할 수 있는 상한선을 정하는 방식이다. 한 프로세스가 다른 프로세스에게 CPU 점유권을 양보하면 양보한 프로세스의 응답시간을 증가시키면 에이징 기법을 구현 할 수 있을 것이라 생각된다. 레포트를 마친 후 시간이 남는다면 에이징 기법을 구현해 보겠다.

4. None Preemptive Priority Scheduling
(비선점형 우선순위 스케줄링)

비선점형 스케줄링은 한 프로세스가 CPU를 할당받으면 작업 종료 후 CPU 반환 시까지 다른 프로세스는 CPU점유가 불가능한 스케줄링 방식이다. 비선점형 우선순위는 우선순위를 기준으로 CPU가 프로세스의 작업을 점유한다.

● NPP 스케줄링 함수

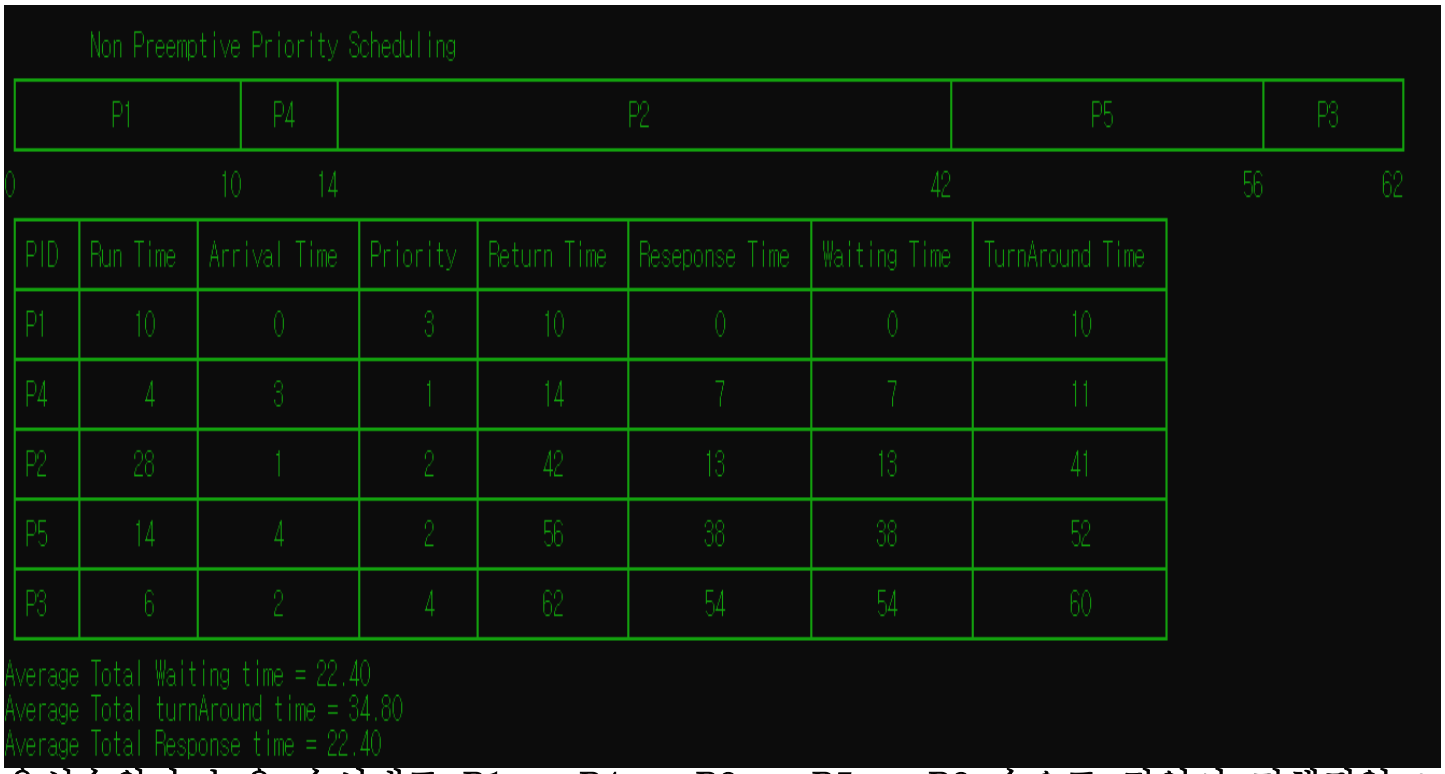
```
void NPPS_Process_Time(Process* p, int n, char* check_id) {
    int current_time = 0; // 현재까지 누적 시간을 저장할 변수

    /* 가장 먼저 들어온 프로세스 실행 및 시간 계산 */
    p[0].return_time = p[0].run_time;
    p[0].turnAround_time = p[0].return_time - p[0].arrival_time;
    p[0].response_time = 0;
    p[0].completed = true;

    current_time = p[0].run_time;
```

매개변수로 프로세스 포인터와 프로세스 개수, 가장 처음 들어온 프로세스를 확인할 문자열 포인터를 선언한다. 가장 먼저 들어온 프로세스의 작업을 진행하고 현재 시간을 갱신한다.

```
qsort_priority_time(p, n);

for (int i = 0; i < n; i++) {
    if (strcmp(check_id, p[i].id) == 0) {
        i++;
    }

    p[i].response_time = current_time - p[i].arrival_time;
    p[i].return_time = current_time + p[i].run_time;
    p[i].turnAround_time = p[i].return_time - p[i].arrival_time;
    p[i].waiting_time = current_time - p[i].arrival_time;
    p[i].completed = true;

    current_time += p[i].run_time;
}
```

도착시간을 기준으로 프로세스를 정렬하고 가장 처음 각 프로세스의 시간을 계산한다. 우선순위 기준으로 정렬된 알고리즘이므로 간단하게 구현할 수 있었다.

비선점형 우선순위 스케줄링의 간트차트는 아래와 같다.



PID	Run Time	Arrival Time	Priority	Return Time	Reseponse Time	Waiting Time	TurnAround Time
P1	10	0	3	10	0	0	10
P4	4	3	1	14	7	7	11
P2	28	1	2	42	13	13	41
P5	14	4	2	56	38	38	52
P3	6	2	4	62	54	54	60

Average Total Waiting time = 22.40
Average Total turnAround time = 34.80
Average Total Response time = 22.40

우선순위가 높은 순서대로 P1 → P4 → P2 → P5 → P3 순으로 작업이 진행되었고, 우선순위가 같은 P2와 P5의 경우 도착한 시간이 빠른 P2가 먼저 실행되었다.

평균대기 시간 : 22.4초, 평균 소요 시간 : 34.8초, 평균 응답 시간 : 22.4초이다.

우선순위를 사용하지만, 원하는 프로세스를 먼저 실행할 수 있는 장점이 있지만 비선점형의 단점을 가지고 있는 비효율적인 스케줄링이다. 우선순위가 높은 프로세스가 진입하더라도 해당 프로세스의 작업을 완료한 후 진입 요청이 들어온 프로세스의 작업을 미루기 때문에 좋은 방법은 아닌 것 같다.

5. High Response Ratio Next Scheduling
(최고 응답률 우선 스케줄링)

HRN 스케줄링은 SJF 스케줄링에서 발생할 수 있는 아사 현상을 해결하기 위해 만들어진 비선점형 알고리즘이다. HRN 스케줄링은 서비스를 받기 위해 기다린 시간과 CPU 사용 시간을 고려하여 스케줄링하는 알고리즘이다.
최고 응답률을 구하는 식은 아래와 같다.

$$\text{우선순위} = \frac{\text{대기 시간} + \text{CPU 사용 시간}}{\text{CPU 사용 시간}}$$

[출처]
https://velog.io/@chappi/OS%EB%8A%94-%ED%95%A0%EA%BB%80%EB%8D%B0-%ED%95%B5%EC%8B%AC%EB%A7%8C-%ED%95%A9%EB%8B%88%EB%8B%A4.-5%ED%8E%B8-%EC%8A%A4%EC%BC%80%EC%A4%84%EB%A7%812-%EB%B9%84%EC%84%A0%EC%A0%90%ED%98%95-%EC%8A%A4%EC%BC%80%EC%A4%84%EB%A7%81-%EC%95%8C%EA%B3%A0%EB%A6%AC%EC%A6%98FCFS-SJF-HRN
우선순위를 결정할 때 대기 시간을 고려함으로써 아사 현상을 완화한다.

● HRN 스케줄링 함수

```
void HRN_Scheduling(Process* p, int n) {
    int current_time = 0; // 현재시간 저장 할 변수
    int priority_index; // 우선순위를 가지는 인덱스
    int total_waiting_time = 0; // 총 프로세스 대기 시간
    int total_turnAround_time = 0; // 총 프로세스 소요 시간
    int total_response_time = 0; // 총 프로세스 응답 시간
    int total_run_time = 0; // 총 프로세스 실행 시간

    double hrr, temp; // 우선순위 저장 변수, 소숫점 가능
```

HRN 스케줄링에 필요한 변수들을 선언하였다. 우선순위 저장 변수는 우선순위 계산에서 int형 간 나눗셈 계산이 필요하기 때문에 double형으로 선언하였다.

```
Process_init(p, n);

for (int i = 0; i < n; i++) {
    total_run_time += p[i].run_time; // 총 실행 시간을 계산
}

qsort_arrival_time(p, n);
```

프로세스를 초기화하고 총 실행 시간을 계산한 후 도착 시간을 기준으로 정렬한다.

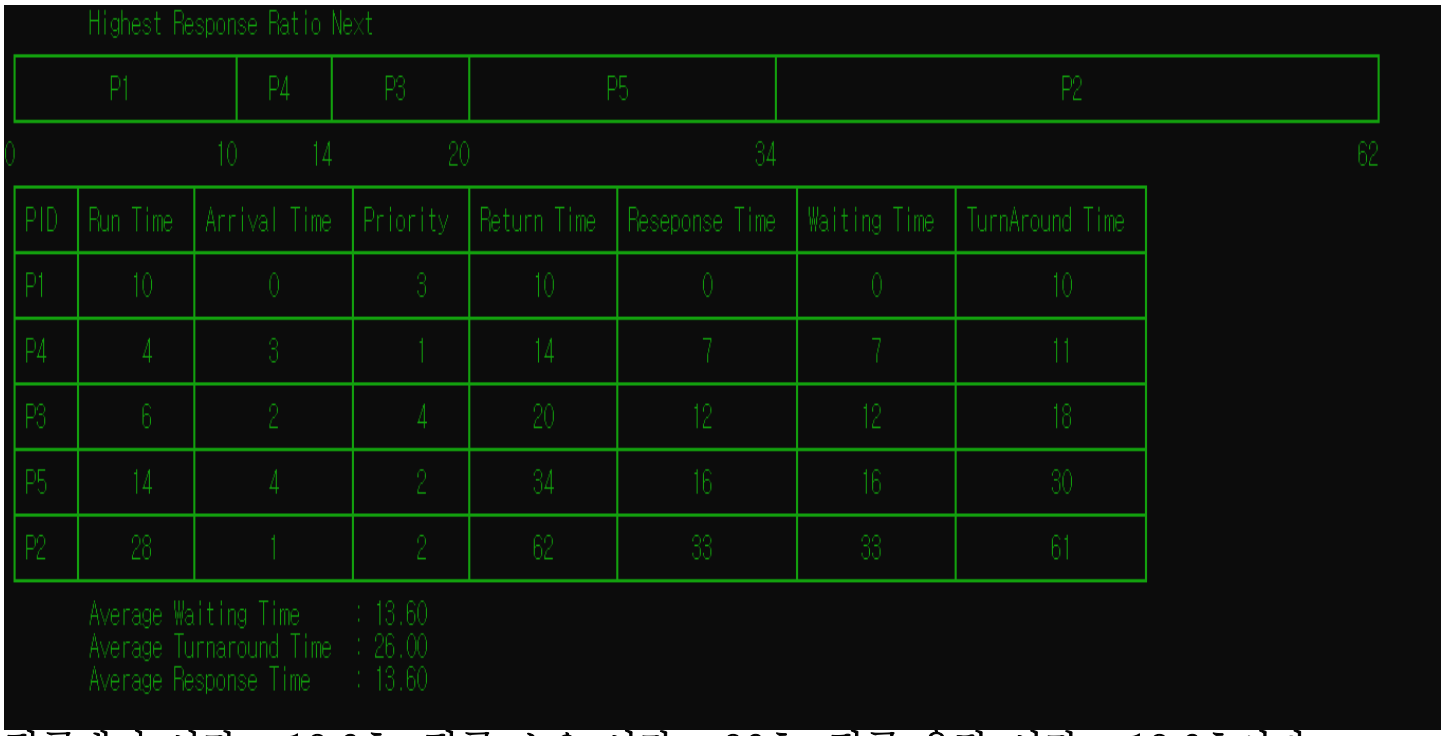
```
current_time = p[0].arrival_time; // 현재 시간은 첫 프로세스가 도착한 시간

while(current_time < total_run_time){ // 증가된 현재 시간이 총 프로세스 시간보다 크다면 모든 프로세스를 돌았음.
    hrr = -9999.0;
    for (int i = 0; i < n; i++){
        if ((p[i].arrival_time <= current_time) && (p[i].completed == false)) {// 가장 처음 들어온 프로세스 부터 실행
            temp = ((double)p[i].run_time + ((double)current_time - (double)p[i].arrival_time)) / (double)p[i].run_time;
            // (실행시간 + 대기시간) / 실행시간 으로 우선순위 계산
            if (hrr < temp){ // 우선순위 갱신
                hrr = temp;
                priority_index = i; // 해당 인덱스로 갱신
            }
        }
    }

    current_time += p[priority_index].run_time; // 서비스 시간 증가
```

current_time 변수는 반복문이 실행되면서 증가하고 증가한 현재 시간이 총 프로세스 실행 시간 변수보다 클 때까지 증가한다. 우선순위 계산을 통해 우선순위를 계산하고 가장 큰 우선순위를 가지는 인덱스로 갱신한다. 단, 완료되지 않았고 도착시간이 실행 중인 프로세스 시간보다 작아야 한다.

```
    p[priority_index].waiting_time = current_time - p[priority_index].arrival_time - p[priority_index].run_time;
    p[priority_index].turnAround_time = current_time - p[priority_index].arrival_time;
    p[priority_index].return_time = p[priority_index].turnAround_time + p[priority_index].arrival_time;
    p[priority_index].response_time = p[priority_index].waiting_time;
    p[priority_index].completed = true;

    total_waiting_time += p[priority_index].waiting_time;
    total_turnAround_time += p[priority_index].turnAround_time;
    total_response_time += p[priority_index].response_time;
```

갱신된 인덱스에 대하여 각 프로세스의 정보들을 계산한다.

계산된 프로세스들의 정보로 간트 차트를 표현하면 아래와 같다.



PID	Run Time	Arrival Time	Priority	Return Time	Reseponse Time	Waiting Time	TurnAround Time
P1	10	0	3	10	0	0	10
P4	4	3	1	14	7	7	11
P3	6	2	4	20	12	12	18
P5	14	4	2	34	16	16	30
P2	28	1	2	62	33	33	61

Average Waiting Time : 13.60
Average Turnaround Time : 26.00
Average Response Time : 13.60

평균대기 시간 : 13.6초, 평균 소요 시간 : 26초, 평균 응답 시간 : 13.6초이다.

HRN 스케줄링과 SJF 스케줄링을 비교하면 대기 시간이 긴 프로세스의 우선순위를 높임으로써 CPU가 점유할 수 있는 확률을 높인다. 하지만 이는 여전히 공평성에 위배된다.

6. Preemptive Priority Scheduling (선점형 우선순위 스케줄링)

선점형 스케줄링은 어떤 프로세스가 CPU를 할당받아 실행 중이라도 CPU의 점유권을 빼앗을 수 있다. 선점형 우선순위 스케줄링은 프로세스 작업 중 우선순위가 높은 프로세스가 진입한다면 높은 우선순위를 가지는 프로세스를 실행하는 알고리즘이다.

● PP 스케줄링 함수

```
void PPS_Process_Time(Process* p, int n) {
    int current_time = 0; // 현재까지 누적 시간을 저장할 변수

    int priority; // 우선순위를 저장할 변수
    int total_run_time = 0; // 총 실행 시간을 저장할 변수
    int k = 0;// 현재 실행할 프로세스 번호


    int* remain_run_time = NULL;
    remain_run_time = (int*)malloc(sizeof(int) * n); // 남은시간을 저장할 배열 동적 할당

    if (remain_run_time == NULL) {      // 메모리 할당 실패
        printf("Not enough memory!");     // 오류 상황 알림
        return -1;                        // 함수 종료
    }

    int* response = NULL;
    response = (int*)malloc(sizeof(int) * n); // 응답 시간을 확인할 배열 동적 할당

    if (response == NULL) {      // 메모리 할당 실패
        printf("Not enough memory!");     // 오류 상황 알림
        return -1;                        // 함수 종료
    }
```

CPU 점유권이 다른 프로세스로 넘어간 경우 남은 실행시간을 계산하기 위한 remain_run_time 변수를 동적 할당한다. 또한 응답시간을 확인하기 위한 response 배열을 동적 할당한다.

```
for (int i = 0; i < n; i++){
    response[i] = 0;
    remain_run_time[i] = p[i].run_time;
    total_run_time += p[i].run_time;
}
```

PP 스케줄링 전 초기화하는 과정이다. response의 경우 true 혹은 false를 판별하는 배열이고 bool 형으로 선언해도 무관했을 것 같다. 각각의 프로세스의 남은 실행시간을 저장하고 총 실행시간을 계산한다.

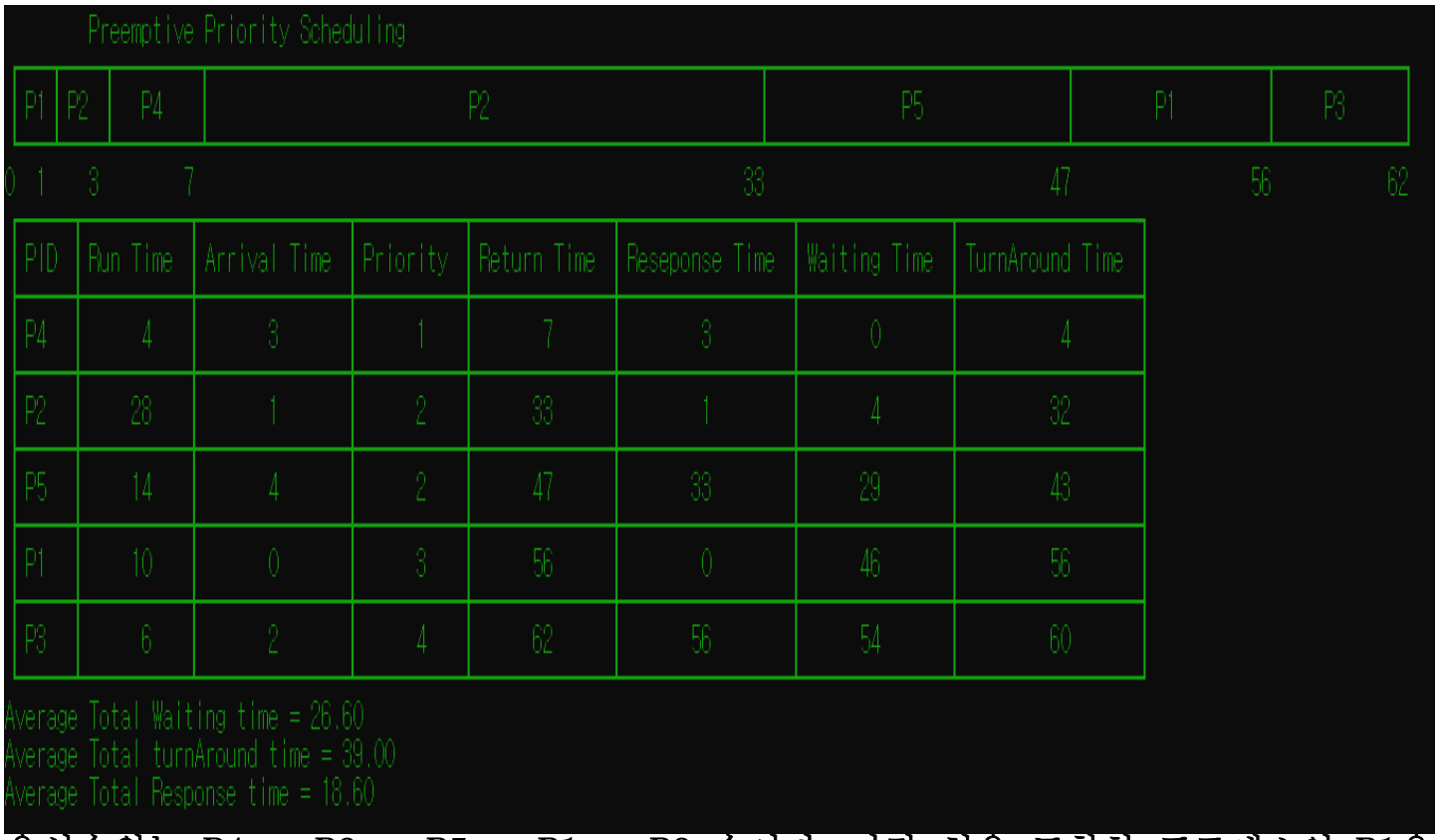
```
/* 현재 시간이 총 실행 시간이 되기 전까지 반복 */
while (current_time < total_run_time){ // 현재 시간이 총 실행시간보다 작은 경우 반복

    priority = INT_MAX; // 우선순위 초기화

    if (current_time < p[n - i].arrival_time) { // 프로세스 도착 검사
        for (int i = 0; i < n; i++) {
            /* 완료되지 않았으며 도착시간이 현재시간보다 작거나 같으며 현재 우선순위보다 우선순위가 작을 경우 */
            if ((p[i].completed == false) && (p[i].arrival_time <= current_time) && (priority > p[i].priority)) {
                priority = p[i].priority; // 우선순위 갱신
                k = i;
            }
        }
    }else { // 모든 프로세스가 도착한 경우
        for (int i = 0; i < n; i++){
            /* 완료되지 않았으며 현재 우선순위보다 우선순위가 작을 경우 */
            if ((p[i].completed == false) && (priority > p[i].priority)){
                priority = p[i].priority;
                k = i;
            }
        }
    }
```

우선순위가 가장 높은 프로세스를 찾는 과정이다. 현재 시간이 총 실행시간과 같거나 커지면 반복을 종료한다. 프로세스가 도착하지 않았고 아직 프로세스가 완료되지 않았으며 현재 우선순위보다 해당 프로세스의 우선순위가 작은 경우 우선순위를 갱신한다. 모든 프로세스가 도착한 경우, 도착시간을 제외한 나머지 조건을 고려하여 우선순위를 갱신한다.

```
if (response[k] == 0){ // 선택된 프로세스가 처음 시작될 경우
    response[k]++; // 초기 실행이 아님을 표시
    p[k].response_time = current_time; // 실행중인 프로세스의 응답시간 저장
}

remain_run_time[k]--; // k 프로세스 남은시간 감소
current_time++; // 현재 시간 증가


if (remain_run_time[k] == 0){ // k 프로세스가 완료된 경우
    p[k].completed = true; // 완료 상태로 변경
    p[k].waiting_time = current_time - p[k].run_time - p[k].arrival_time; // 대기 시간 계산
    p[k].return_time = current_time; // 반환 시간 계산

}
```

초기 실행인 경우엔 응답시간을 증가하여 초기 실행이 아님을 표시하고 해당 프로세스에 응답시간을 저장한다. 갱신된 프로세스의 남은 시간을 감소하고 프로세스의 현재 시간을 증가한다. 해당 프로세스가 완료됐을 경우 프로세스 상태를 변경하고 프로세스의 정보를 계산한다.

계산된 프로세스들의 정보와 간트차트는 아래와 같다.



우선순위는 P4 > P2 > P5 > P1 > P3 순이다. 가장 처음 도착한 프로세스인 P1을 실행하고 P2가 도착한 순간 우선순위를 계산하여 우선순위가 더 높은 프로세스로 작업한다. 다음 우선순위가 높은 P4가 도착하면 P4의 작업을 진행한다. P4의 작업이 끝나고 다음 우선순위에 따라 우선순위가 높았던 P2를 작업한다. P5가 P1과 P3보다 우선순위가 높으므로 P5의 작업이 이루어졌다.

평균대기 시간 : 26.6초, 평균 소요 시간 : 39초, 평균 응답 시간 : 18.6초이다.

현재 작업 중인 프로세스를 무시하고 우선순위에 따라 프로세스의 작업을 진행한다는 것은 공평성을 위배하고 아사 현상을 일으킬 수 있다. 또한 매번 우선순위를 계산해야 하기 때문에 많은 양의 프로세스가 진입한 경우 오버헤드가 일어나 시스템의 효율이 떨어진다.

7. Round Robin Scheduling (라운드 로빈 스케줄링)

라운드 로빈 스케줄링은 선점형 스케줄링으로, 프로세스들 사이에 우선순위를 두지 않고, 순서대로 시간 단위(타임 슬라이스/퀀텀)로 CPU를 할당하는 방식이다. 라운드 로빈 스케줄링은 FCFS와 유사하나 타임 슬라이스를 가진다는 점에서 다르다.

● RR 스케줄링 대기 시간 계산 함수

```
void RR_Process_Waiting_Time(Process* p, int n, Quantum quantum) { // 대기 시간 계산 함수
    int current_time = 0; // 현재시간 초기화
    int total_run_time = 0; // 총 실행시간 초기화
    int* remain_run_time = NULL;
    remain_run_time = (int*)malloc(sizeof(int) * n); // 남은 시간 저장 배열 할당

    if (remain_run_time == NULL) {
        fprintf(stderr, "메모리 할당 실패₩n");
        exit(1);
    }

    bool* check_response_time = NULL;
    check_response_time = (bool*)malloc(sizeof(bool) * n); // 응답 시간 처리 확인 배열 할당

    if (check_response_time == NULL) {
        fprintf(stderr, "메모리 할당 실패₩n");
        exit(1);
    }
```

매개변수로 프로세스 포인터와 프로세스 개수 그리고 퀀텀 개수를 받아온다. 남은 시간을 저장할 배열을 동적 할당하고 프로세스가 응답 하였는지 확인할 배열을 동적 할당 한다.

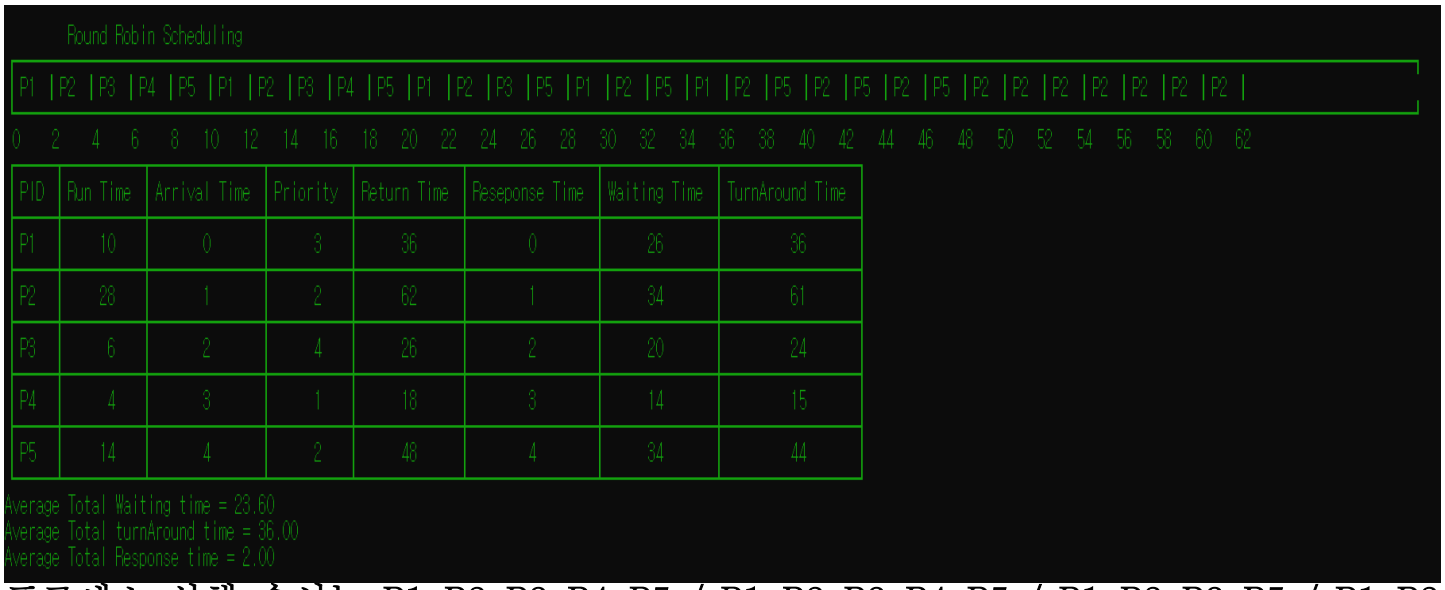
```
for (int i = 0; i < n;i++) {
    remain_run_time[i] = p[i].run_time; // 각 프로세스별 남은시간 저장
    check_response_time[i] = false; // 프로세스 응답 초기화
    total_run_time += p[i].run_time; // 총 실행시간 계산
}
```

프로세스의 남은 시간을 저장하고 동적 할당한 배열을 초기화한다.

```
while (true) { // 모든 프로세스가 완료될 때 까지 반복
    bool check = true;
    for (int i = 0; i < n; i++) {
        if (remain_run_time[i] > 0) { // 실행 시간이 남아 있는 경우
            check = false;

            if (check_response_time[i] == false) { // 응답 시간 처리 안한 경우
                p[i].response_time = current_time - p[i].arrival_time; // 응답시간 계산
                check_response_time[i] = true;
            }

            if (remain_run_time[i] > quantum) { // 남은 시간이 타임 슬라이스 보다 클 경우
                current_time += quantum; // 현재 시간 증가
                remain_run_time[i] -= quantum; // 현재 실행중인 프로세스 남은 시간 감소
            }
            else { // 남은 시간이 타임 슬라이스 보다 작은 경우
                current_time += remain_run_time[i]; // 현재시간 증가
                p[i].waiting_time = current_time - p[i].run_time; // 대기시간 계산
                remain_run_time[i] = 0; // 작업 끝난 프로세스 남은 시간 0으로 초기화
            }
        }
    }

    if (check == true) // 남아 있는 프로세스가 없는 경우
        break;
}
```

무한 루프를 통하여 모든 프로세스가 완료될 때까지 계산한다. bool check 변수를 통하여 남아있는 프로세스가 없도록 하였다. 실행시간이 남아있는 경우 프로세스의 작업이 진행되는데 응답시간을 처리한 후 타임 슬라이스를 고려하여 작업을 진행한다. 남은 시간이 타임 슬라이스 보다 큰 경우 현재 시간은 타임 슬라이스만큼 증가하고 남은 실행시간을 타임 슬라이스 만큼 감소시킨다. 남은 시간이 타임 슬라이스보다 작은 경우 현재 시간을 증가시키고 작업이 끝난 프로세스의 남은 시간은 초기화한다. 또한 해당 프로세스에 대한 대기 시간을 계산한다.

계산된 프로세스들의 정보와 간트 차트는 아래와 같다.



프로세스 실행 순서는 P1 P2 P3 P4 P5 / P1 P2 P3 P4 P5 / P1 P2 P3 P5 / P1 P2 P5 / P1 P2 P5 / P2 P5 / P2 P5 / P2 P2 P2 P2 P2 P2 P2 이다. P4의 작업시간은 4 이므로 P1 P2 P3 P4 P5 2번 반복후 P1 P2 P3 P5가 진행된다. 타임 슬라이스 만큼 작업이 잘 진행된 것을 볼 수 있다.

평균대기 시간 : 23.6초, 평균 소요 시간 : 36초, 평균 응답 시간 : 2초이다.

라운드 로빈 스케줄링은 긴 작업을 무한히 기다리는 Convey Effect를 효과적으로 줄일 수 있다는 장점이 있다. 만약 타임 슬라이스가 커진다면 FCFS와 같은 방식이기 때문에 코드만 복잡해질 뿐 알고리즘의 효율성은 떨어진다. 또한 선점형 방식으로 인한 문맥 교환 시간이 추가되어 알고리즘의 작업시간이 더 길어질 수 있다.

8. Shortest Remaining Time Scheduling
(최소 잔류 시간 우선 스케줄링)

비선점 스케줄링인 SJF 스케줄링 기법을 선점 형태로 변경한 기법이다. SJF 스케줄링처럼 CPU 점유 시간이 가장 짧은 프로세스에 CPU를 먼저 할당하는 방식으로, 단지 차이점은 선점형으로 바뀌어 중요한 프로세스가 있으면 점유 시간이 길어도 먼저 실행시킬 수 있는 권한이 생긴다는 것이다.

● SRT 스케줄링 함수

```
void SRT_Process_System(Process* p, int n) {
    int current_time = 0;

    int total_run_time = 0;

    int shortest_remain_time;

    int k = 0;

    int* remain_run_time = NULL;
    remain_run_time = (int*)malloc(sizeof(int) * n);
    if (remain_run_time == NULL) {
        fprintf(stderr, "메모리 할당 실패₩n");
        exit(1);
    }

    int* check_response = (int*)malloc(sizeof(int) * n);
    check_response = (int*)malloc(sizeof(int) * n);
    if (check_response == NULL) {
        fprintf(stderr, "메모리 할당 실패₩n");
        exit(1);
    }
```

SRT 스케줄링에 필요한 변수와 배열 들을 동적 할당한다. k는 갱신될 프로세스 인덱스이다.

```
for (int i = 0; i < n; i++){
    check_response[i] = 0; //
    remain_run_time[i] = p[i].run_time;
    total_run_time += p[i].run_time;
}
```

동적 할당한 응답 처리 배열과 남은 시간을 저장할 배열을 초기화하고 총 실행시간을 계산한다.

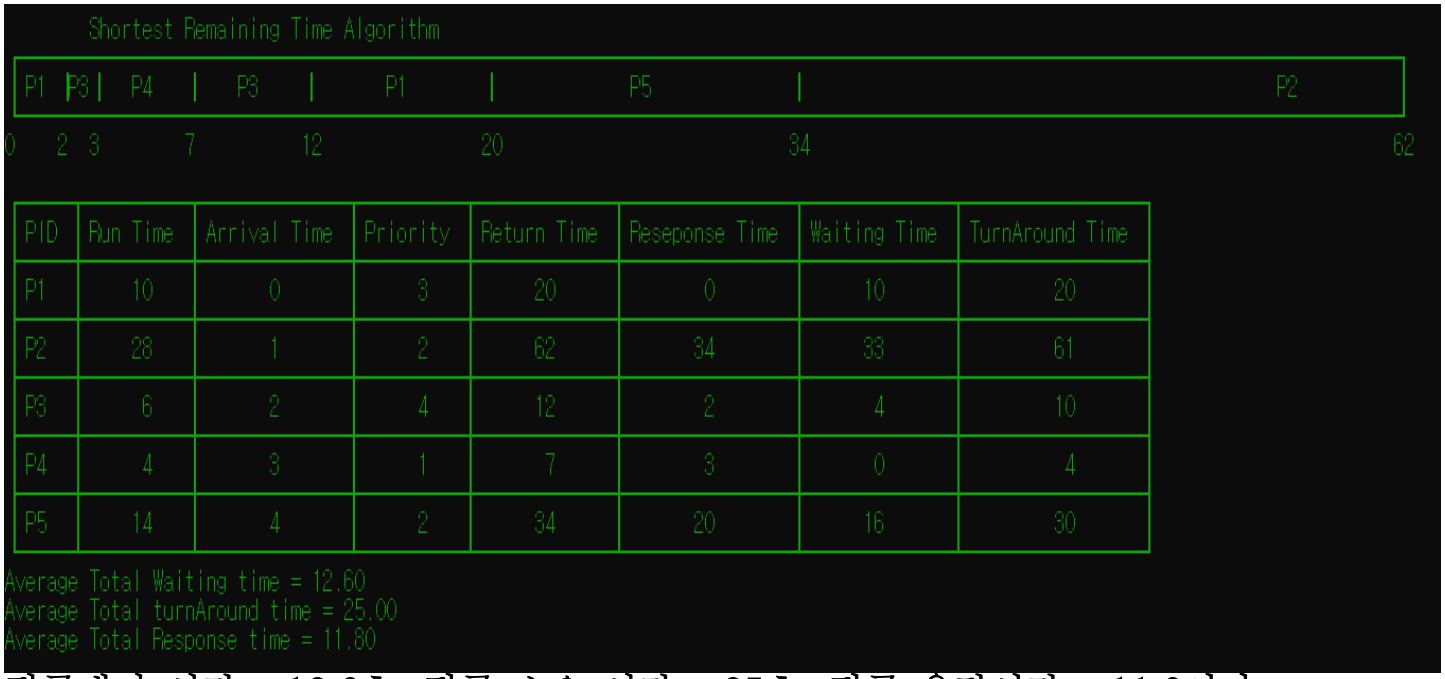
```
/* 현재 시간이 총 실행 시간이 되기 전까지 반복 */
while (current_time < total_run_time)  {
    shortest_remain_time = INT_MAX; // 최소작업 인덱스를 INT_MAX로 초기화

    /* 가장 마지막에 들어온 프로세스의 도착시간 보다 작을 경우 */
    if (current_time <= p[n - 1].arrival_time) {
        for (int i = 0; i < n; i++) {
            /* 완료되지 않았으며 도착시간이 현재시간보다 작거나 같으며 현재 최소작업 시간보다 남은 실행시간이 작을 경우 */
            if ((p[i].completed == false) && (p[i].arrival_time <= current_time) && (shortest_remain_time > remain_run_time[i])){
                shortest_remain_time = remain_run_time[i]; // 최소 작업 시간 갱신
                k = i; // 최소 작업 프로세스 인덱스 갱신
            }
        }
    }
    else
    {
        for (int i = 0; i < n; i++){ // 완료되지 않았으며 현재 최소작업 시간보다  남은 실행시간이 작을 경우
            if ((p[i].completed == false) && (shortest_remain_time > remain_run_time[i]))  {
                shortest_remain_time = remain_run_time[i]; // 최소 작업 시간 갱신b
                k = i; // 최소 작업 프로세스 인덱스 갱신
            }
        }
    }
```

현재 시간이 총 실행시간이 되기 전까지 반복한다. 아직 도착하지 않은 경우와 도착이 완료된 경우로 최소 작업시간을 갱신한다. 완료되지 않았고 도착시간이 현재 시간보다 작거나 같으며 현재 최소 작업시간 보다 남은 실행시간이 작은 경우 갱신한다.

```
/* 선택된 프로세스가 처음 시작될 경우 */
if (check_response[k] == 0){
    check_response[k] = 1; // 응답 표시
    p[k].response_time = current_time; // 실행중인 프로세스의 응답시간 저장
}

remain_run_time[k]--;
// 실행된 프로세스의 남은 시간 감소
current_time++;
// 현재 시간 증가

/* 프로세스의 남은 실행 시간이 0이될 경우 */
if (remain_run_time[k] == 0){
    p[k].completed = true; // 완료 상태로 변경
    p[k].waiting_time = current_time - p[k].run_time - p[k].arrival_time; // 대기 시간 계산
    p[k].return_time = current_time; // 반환 시간 계산
}
```

프로세스가 응답한 경우를 표시하고 실행 중인 프로세스의 응답시간을 저장한다. 실행한 프로세스의 남은 시간을 감소하고 현재 시간을 증가시킨다. 프로세스가 종료된 경우 해당 프로세스의 정보를 계산한다.

계산된 프로세스들의 정보와 간트 차트는 아래와 같다.



PID	Run Time	Arrival Time	Priority	Return Time	Reseponse Time	Waiting Time	TurnAround Time
P1	10	0	3	20	0	10	20
P2	28	1	2	62	34	33	61
P3	6	2	4	12	2	4	10
P4	4	3	1	7	3	0	4
P5	14	4	2	34	20	16	30

Average Total Waiting time = 12.60
Average Total turnAround time = 25.00
Average Total Response time = 11.80

평균대기 시간 : 12.6초, 평균 소요 시간 : 25초, 평균 응답시간 : 11.8이다.

SJF 스케줄링과 비교하였을 때 평균 대기 시간이 짧아지는 것을 볼 수 있다. 하지만 SJF 스케줄링과 마친가지로 운영체제가 프로세스가 종료된 시간을 예측하기 어렵고 아사 현상이 일어날 수 있기 때문에 잘 사용하지 않는다.

9. 전체 소스 코드

Cpu-Scheduling-Simulator.c

```
/*
	Operating System Term Project
	프로젝트 명 : CPU-Schduling-Simulator
	이름 : 김민욱
	학번 : 20194024
	사용 언어 : C Language
	프로젝트 내용 : 7가지 CPU 스케줄링 시뮬레이터 개발
					(FCFS, SJF, 비선점 Priority, 선점 Priority, RR, SRT, HRN)
					간트 차트, 각 프로세스별 대기시간, 평균 대기 시간, 각 프로세스별 응답시
간,
					평균 응답시간, 각 프로세스별 반환 시간, 평균 반환 시간 출력
	[Repository]
	https://github.com/MinWook6457/cpu-scheduling-simulator
*/
/*
	Turnaround Time (반환 시간)
	Turnaround Time = Completion Time - Arrival Time (각 프로세스 완료 시간 - 각 프로세스 도착 시간)

	Response Time (응답 시간)
	Response Time = FirstRun Time - Arrival Time (첫 번째 실행 시간 - 첫 번째 도착 시간)
*/
#include <stdio.h>
#include <stdlib.h>
#include "Process.h"
#include "FCFS.h"
#include "SJF.h"
#include "NPPS.h"
#include "HRN.h"
#include "PPS.h"
#include "RR.h"
#include "SRT.h"
#include "Function_to_Sorting.h"

#pragma warning(disable:4996)


int main() {
	int process_count = 0; // 프로세스 갯수를 저장할 변수

	Quantum quantum; // 퀀텀 변수 선언

	FILE* fp = fopen("Process.txt", "r"); // 파일 읽기 모드로 열기

	/* 파일 열기에 실패한 경우 */
	if (fp == NULL) {
		fprintf(stderr, "File Open Fail\n");
		exit(1);
	}
```

```
	fscanf(fp, "%d", &process_count); // 파일에서 읽은 프로세스 갯수 저장

	Process* process = (Process*)malloc(sizeof(Process) * process_count); // 프로세스 개수 만큼 동적할당

	if (process == NULL) {
		fprintf(stderr, "메모리 할당 실패\n");
		exit(1);
	}

	for (int i = 0; i < process_count; i++) { // 파일에서 프로세스id, 도착시간, 반환시간, 우선순위를 받아 구조체에 저장
		fscanf(fp, "%s %d %d %d", process[i].id, &process[i].arrival_time, &process[i].run_time, &process[i].priority);
		// printf("%s %d %d %d\n", process[i].id, process[i].arrival_time, process[i].run_time, process[i].priority);
	}

	fscanf(fp, "%d", &quantum); // 파일에서 읽은 퀀텀 개수 저장
	// printf("%d", quantum);

	FCFS_Scheduling(process, process_count);
	printf("\n");
	SJF_Scheduling(process, process_count);
	printf("\n");
	NPPS_Scheduling(process, process_count);
	printf("\n");
	HRN_Scheduling(process, process_count);
	printf("\n");
	PPS_Scheduling(process, process_count);
	printf("\n");
	RR_Scheduling(process, process_count,quantum);
	printf("\n");
	SRT_Scheduling(process, process_count);

	free(process);


	return 0;
}
```

Process.h

```
#ifndef _process_
#define _process_

#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>
#include <malloc.h>
#include <Windows.h>
/*
	헤더명 : FCFS.h(First Come First Served)
	내용 : 프로세스 구조체 선언, 퀀텀 키워드 선언, 프로세스 함수 초기화
*/

typedef struct _process { // 프로세스 구조체 선언
	char id[20]; // 프로세스 id - File
	int waiting_time; // 프로세스 대기시간
	int arrival_time; // 프로세스 도착시간 - File
	int run_time; // 프로세스 실행시간 - File
	int priority; // 프로세스 우선순위 - File
	int response_time; // 프로세스 응답시간
	int return_time; // 프로세스 반환시간
	int turnAround_time; // 프로세스 소요시간
	bool completed;
}Process;

typedef int Quantum; // 시간 할당량 Quantum 키워드 사용

void Process_init(Process p[], int n) // 프로세스 초기화 함수
{
	for (int i = 0; i < n; i++) { // 프로세스 갯수 만큼 반복
		p[i].waiting_time = 0; // 대기 시간 초기화
		p[i].response_time = 0; // 응답 시간 초기화
		p[i].turnAround_time = 0; // 반환 시간 초기화
		p[i].completed = false;
	}
}

#endif
```

Function_to_Sorting.h

```
#ifndef __Function_to_Sorting__
#define __Function_to_Sorting__

#include "Process.h"
#include <stdio.h>

// 반환시간 퀵정렬 매개변수로 들어갈 함수 - 오름차순
int compare_return_time(const void* a, const void* b){
        Process p1 = *(Process*)a;
        Process p2 = *(Process*)b;

        if (p1.return_time < p2.return_time)
                return -1;

        if (p1.return_time > p2.return_time)
                return 1;

        return 0;
}
// 반환시간 기준으로 (퀵)정렬할 함수
void qsort_return_time(Process* p, int n) {
        qsort(p, n, sizeof(Process), compare_return_time);
}

// 우선순위 퀵정렬 매개변수로 들어갈 함수 - 오름차순
int compare_priority_time(const void* a, const void* b){
        Process p1 = *(Process*)a;
        Process p2 = *(Process*)b;

        if (p1.priority < p2.priority)
                return -1;

        if (p1.priority > p2.priority)
                return 1;

        if (p1.priority == p2.priority) {
                if (p1.arrival_time < p2.arrival_time)
                        return -1;
                if (p1.arrival_time > p2.arrival_time)
                        return 1;
        }
        else
                return 0;
}
// 우선순위를 기준으로 (퀵)정렬할 함수
void qsort_priority_time(Process* p, int n) {
        qsort(p, n, sizeof(Process), compare_priority_time);
}
```

```
// 실행시간 퀵정렬 매개변수로 들어갈 함수 - 오름차순
int compare_run_time(const void* a, const void* b) {
	Process p1 = *(Process*)a;
	Process p2 = *(Process*)b;

	if (p1.run_time < p2.run_time)
		return -1;

	if (p1.run_time > p2.run_time)
		return 1;

	return 0;
}
// 실행시간을 기준으로 (퀵)정렬할 함수
void qsort_run_time(Process* p, int n) {
	qsort(p, n, sizeof(Process), compare_run_time);
}


// 도착시간 퀵정렬 매개변수로 들어갈 함수 - 오름차순
int compare_arrival_time(const void* a, const void* b) {
	Process p1 = *(Process*)a;
	Process p2 = *(Process*)b;

	if (p1.arrival_time < p2.arrival_time)
		return -1;

	if (p1.arrival_time > p2.arrival_time)
		return 1;

	return 0;
}

// 도착시간을 기준으로 (퀵)정렬할 함수
void qsort_arrival_time(Process* p, int n) {
	qsort(p, n, sizeof(Process), compare_arrival_time);
}

#endif
```

View_Table.h

```
#ifndef __View_Table__
#define __View_Table__


#include "Process.h"
#include <stdio.h>
void view_table(Process p[], int n) {
        printf("┌─────┬──────────┬──────────────┬──────────┬─────────────┬────────────────┬──────────────┬──────────────────┐\n");
        printf("| PID | Run Time | Arrival Time | Priority | Return Time | Reseponse Time | Waiting Time | TurnAround Time |");
        printf("\n├─────┼──────────┼──────────────┼──────────┼─────────────┼────────────────┼──────────────┼──────────────────┤");
        printf("\n");
        for (int i = 0; i < n - 1; i++) {
                printf("| %s  |    %3d   |      %3d     |    %3d   |     %3d     |       %3d      |      %3d     |        %3d       |\n",
                                        p[i].id,    p[i].run_time,    p[i].arrival_time,    p[i].priority,    p[i].return_time, p[i].response_time, p[i].waiting_time, p[i].turnAround_time);
                printf("├─────┼──────────┼──────────────┼──────────┼─────────────┼────────────────┼──────────────┼──────────────────┤\n");
        }
        printf("| %s  |    %3d   |      %3d     |    %3d   |     %3d     |       %3d      |      %3d     |        %3d       |\n",
                p[n - 1].id, p[n - 1].run_time, p[n - 1].arrival_time, p[n - 1].priority, p[n - 1].return_time, p[n - 1].response_time, p[n - 1].waiting_time, p[n - 1].turnAround_time);
        printf("└─────┴──────────┴──────────────┴──────────┴─────────────┴────────────────┴──────────────┴──────────────────┘");
}


#endif __View_Table__
```

FCFS.h

```
#ifndef _FCFS_
#define _FCFS_

/*
	헤더명 : FCFS.h(First Come First Served Scheduling)
	내용 : 준비 큐에 도착한 순서대로 CPU를 할당하는 비선점형방식
	입력 : 프로세스 포인터, 프로세스 갯수
	출력 : FCFS 스케쥴링으로 인한
	각 프로세스별 대기시간, 평균 대기 시간,
	각 프로세스별 응답시간, 평균 응답시간,
	각 프로세스별 반환 시간, 평균 반환 시간
*/

#include "Process.h"
#include "Function_to_Sorting.h"
#include "View_Table.h"
#include <Windows.h>
#include <stdbool.h>

void textcolor(int colorNum) {
	SetConsoleTextAttribute(GetStdHandle(STD_OUTPUT_HANDLE), colorNum);
}

void FCFS_print_gantt_chart(Process* p, int n)
{	// 반복문에서 사용할 변수 선언

	textcolor(2);
	printf("\tFirst Come First Served Scheduling\n");
	printf("┌");
	/* 상단 바 출력 */
	for (int i = 0; i < n; i++)
	{
		for (int j = 0; j < p[i].run_time; j++)
			printf("─");

		if (i == n - 1)
			printf("┐");
		else
			printf("┬");

	}

	printf("\n|");

	/* 프로세스 이름 출력 */
	for (int i = 0; i < n; i++)
	{
		for (int j = 0; j < p[i].run_time - 1; j++)
			printf(" ");
```

```
                printf("%s", p[i].id);

                for (int j = 0; j < p[i].run_time - 1; j++)
                        printf(" ");

                printf("|");
        }

        printf("\n");
        printf("└");
        /* 하단 바 출력 */
        for (int i = 0; i < n; i++)
        {
                for (int j = 0; j < p[i].run_time; j++)
                        printf("—");

                if (i == n - 1)
                        printf("┘");
                else
                        printf("┴");
        }

        printf("\n");

        /* 프로세스 시간 출력 */
        printf("0");

        for (int i = 0; i < n; i++)
        {
                for (int j = 0; j < p[i].run_time; j++)
                        printf("  ");

                if (p[i].return_time > 9)
                        printf("\b");

                printf("%d", p[i].return_time);

        }

        printf("\n");
}

void FCFS_Scheduling(Process* p, int pc) {
        int total_waiting_time = 0; // 총 프로세스 대기 시간
        int total_turnAround_time = 0; // 총 프로세스 소요 시간
        int total_response_time = 0; // 총 프로세스 응답 시간
        int total_return_time = 0; // 총 프로세스 반환 시간

        Process_init(p, pc);

        qsort_arrival_time(p, pc);
```

```
	/* 맨 처음 들어온 프로세스 실행 */
	p[0].return_time = p[0].run_time; // 파일에서 받은 run_time을 return_time에 대입
	p[0].turnAround_time = p[0].return_time - p[0].arrival_time; // 반환시간 = 실행시간 - 대기시간
	p[0].response_time = 0; // 응답시간 초기화
	p[0].waiting_time = 0; // 대기시간 초기화

	/* 실행된 프로세스 만큼 사용률 증가 */
	total_waiting_time += p[0].waiting_time; // 총 대기시간 증가
	total_turnAround_time += p[0].turnAround_time; // 총 소요시간 증가
	total_response_time += p[0].response_time; // 총 응답시간 증가
	total_return_time += p[0].run_time; // 총 실행시간 증가

	// 선입 선출 구조로 들어오는 순서대로 프로세스 계산
	for (int i = 1; i < pc; i++)
	{
		/* 각 프로세스 계산 */
		p[i].waiting_time = total_return_time - p[i].arrival_time; // 각 프로세스 대기시간 = 총 반환시간 - 도착시간
		p[i].return_time = total_return_time + p[i].run_time; // 각 프로세스 반환시간 = 총 반환시간 + 실행시간
		p[i].turnAround_time = p[i].return_time - p[i].arrival_time; // 각 프로세스 소요시간 = 각 프로세스 반환시간 - 도착시간
		p[i].response_time += p[i].waiting_time; // 반응시간 += 대기시간

		/* 실행된 프로세스 만큼 사용률 증가 */
		total_return_time += p[i].run_time; // 총 반환시간 증가
		total_waiting_time += p[i].waiting_time; // 총 대기시간 증가
		total_turnAround_time += p[i].turnAround_time; // 총 소요시간 증가
		total_response_time += p[i].response_time; // 총 응답시간 증가
	}
	FCFS_print_gantt_chart(p, pc); // 간트차트 그리기
	view_table(p, pc); // 테이블 그리기
	printf("\n");
	printf("Average Total Waiting time = %.2lf\n", (double)total_waiting_time / (double)pc); // 평균 대기 시간
	printf("Average Total turnAround time = %.2lf\n", (double)total_turnAround_time / (double)pc); // 평균 반환 시간
	printf("Average Total Response time = %.2lf\n", (double)total_response_time / (double)pc); // 평균 응답 시간


}

#endif
```

SJF.h

```
#ifndef __SJF__
#define __SJF__

/*
	헤더명 : SJF.h(Shortest Job First Scheduling)
	내용 : 준비 큐에 있는 프로세스 중에서 실행 시간이 가장 짧은 작업부터 CPU를 할당하는 비선점형 방식
	입력 : 프로세스 포인터, 프로세스 갯수
	출력 : SJF 스케쥴링으로 인한
	각 프로세스별 대기시간, 평균 대기 시간,
	각 프로세스별 응답시간, 평균 응답시간,
	각 프로세스별 반환 시간, 평균 반환 시간
*/
#pragma warning(disable:4996)
#include "Process.h"
#include "View_Table.h"
#include "Function_to_Sorting.h"
#include <string.h>

Process tmp[100];
void SJF_print_gantt_chart(Process* p, int n, char* check_id)
{
	printf("\tShortest Job First Scheduling\n");
	int check_num = 0; // 처음 실행된 PID를 추출하기 위한 변수

	for (int i = 0; i < n; i++){
		if (strcmp(check_id, p[i].id) == 0) { // 처음 실행된 프로세스 id를 받아 현재 구조체에 있는 id 와 같은 인덱스 추출
			break;
		}
		check_num++;
	}
	// Process* tmp = (Process*)malloc(sizeof(Process) * (check_num)); // 구조체 교환을 위한 동적할당)
	Process_init(tmp, check_num);
	/*
	for (int i = 0; i < check_num ; i++) {
		tmp[i].arrival_time = 0;
		tmp[i].return_time = 0;
		tmp[i].run_time = 0;
		tmp[i].priority = 0;
		strcpy(tmp[i].id, "");
	}
	*/
	for (int i = 0; i <= check_num; i++) {
		tmp[i] = p[i]; // 구조체 복사
	}

	Process Temp = tmp[check_num]; // Temp에 추출된 인덱스가 해당하는 프로세스 저장
	for (int i = 0; i < check_num; i++) { // 가장 먼저 실행된 프로세스를 맨 앞으로 보내기 위해서 구조체 배열을 뒤로 민다.
		p[i + 1] = tmp[i];
```

```
}

p[0] = Temp; // 저장해두었던 프로세스를 다시 구조체 첫 번째 인덱스로 저장

// free(tmp);


printf("┌");
/* 상단 출력 */
for (int i = 0; i < n; i++){
        for (int j = 0; j < p[i].run_time; j++)
                printf("─");
        if (i == n - 1)
                printf("┐");
        else
                printf("┬");
}

printf("\n|");
/* 프로세스 이름 출력 */
for (int i = 0; i < n; i++){
        for (int j = 0; j < p[i].run_time - 1; j++)
                printf(" ");

        printf("%s", p[i].id);

        for (int j = 0; j < p[i].run_time - 1; j++)
                printf(" ");

        printf("|");
}

printf("\n");
printf("└");

/* 하단 바 출력 */
for (int i = 0; i < n; i++){
        for (int j = 0; j < p[i].run_time; j++)
                printf("─");
        if (i == n - 1)
                printf("┘");
        else
                printf("┴");
}

printf("\n");

/* 프로세스 시간 출력 */
printf("0");
for (int i = 0; i < n; i++){
        for (int j = 0; j < p[i].run_time; j++)
                printf("  ");
```

```
                if (p[i].return_time > 9)
                        printf("\b");
                printf("%d", p[i].return_time);
        }
        printf("\n");
}

void SJF_Process_Time(Process* p, int n) { // 프로세스 별 시간 계산 함수
        int current_time = 0; // 현재까지 누적 시간을 저장할 변수

        int min_index = 0; // 최소 시간을 가지는 인덱스 변수

        /* 도착시간이 가장 빠른 프로세스 먼저 실행 */
        p[0].return_time = p[0].run_time; // 반환시간 저장
        p[0].turnAround_time = p[0].run_time - p[0].arrival_time; // 소요시간 = 실행시간 - 도착시간
        p[0].waiting_time = 0; // 대기시간 0
        p[0].completed = true; // 완료 표시

        current_time = p[0].run_time; // 현재 시간 증가

        /* p[0]는 진행하였으므로 인덱스 1부터 시작 */
        for (int i = 1; i < n; i++) {
                for (int j = 1; j < n; j++) {
                        if (p[j].completed == true) // 이미 완료된 프로세스
                                continue;
                        else {
                                min_index = j; // 아직 실행하지 않은 프로세스
                                break;
                        }
                }

                for (int k = 1; k < n; k++) {
                        /* 완료되지 않은 프로세스이고 도착시간이 현재 누적된 시간보다 적어야 한다.(프로세스 연기되는 현상 해결) 또한 해당 인덱스의 최소 인덱스 다시 검색 */
                        if (p[k].completed == false && p[k].arrival_time < current_time && p[k].run_time < p[min_index].run_time) { // 최소 작업 시간 조건에 맞는 프로세스 찾기
                                min_index = k;
                        }
                }

                p[min_index].waiting_time = current_time - p[min_index].arrival_time; // 대기 시간 계산(=응답시간) : FCFS 형식을 이루고 있으므로 대기시간 = 응답시간
                p[min_index].completed = true; // 완료 상태 갱신

                current_time += p[min_index].run_time; // 누적 시간 갱신
                p[min_index].return_time = current_time; // 반환 시간 갱신
                p[min_index].turnAround_time = p[min_index].return_time - p[min_index].arrival_time; // 소요 시간 갱신
        }
}

void SJF_Scheduling(Process* p, int pc) {
```

```
	int total_waiting_time = 0; // 총 프로세스 대기 시간
	int total_turnAround_time = 0; // 총 프로세스 소요 시간
	int total_response_time = 0; // 총 프로세스 응답 시간

	Process_init(p, pc);

	qsort_arrival_time(p, pc); // 도착시간 기준으로 정렬

	SJF_Process_Time(p, pc); // 도착시간 기준으로 프로세스 계산

	char check_id[10];
	strcpy(check_id, p[0].id); // 가장 처음으로 들어온 PID 저장

	for (int i = 0; i < pc; i++){
		p[i].return_time = p[i].turnAround_time + p[i].arrival_time;
		p[i].response_time = p[i].waiting_time;

		total_waiting_time += p[i].waiting_time;
		total_turnAround_time += p[i].turnAround_time;
		total_response_time += p[i].response_time;
	}

	qsort_run_time(p, pc); // 실행 시간 기준 정렬

	SJF_print_gantt_chart(p, pc, check_id); // 실행 시간 기준으로 출력

	view_table(p, pc);

	printf("\n");
	printf("Average Total Waiting time = %.2lf\n", (double)total_waiting_time / (double)pc); // 평균 대기 시간
	printf("Average Total turnAround time = %.2lf\n", (double)total_turnAround_time / (double)pc); // 평균 반환
시간
	printf("Average Total Response time = %.2lf\n", (double)total_response_time / (double)pc); // 평균 응답 시
간

}

#endif __SJF__
```

NPPS.h

```
#ifndef __NPPS__
#define __NPPS__

/*
	헤더명 : NPPS.h(None Preemptive Priority Scheduling)
	내용 : 각각의 프로세스의 우선순위를 정하여 순위가 높은 작업 순으로 처리하는 방식
	입력 : 프로세스 포인터, 프로세스 갯수
	출력 : NPPS 스케쥴링으로 인한
	각 프로세스별 대기시간, 평균 대기 시간,
	각 프로세스별 응답시간, 평균 응답시간,
	각 프로세스별 반환 시간, 평균 반환 시간

*/

#include "Process.h"
#include <Windows.h>
#include "View_Table.h"
#include "Function_to_Sorting.h"
#pragma warning(disable:4996)

void NPPS_Process_Time(Process* p, int n, char* check_id) {
	int current_time = 0; // 현재까지 누적 시간을 저장할 변수

	/* 가장 먼저 들어온 프로세스 실행 및 시간 계산 */
	p[0].return_time = p[0].run_time;
	p[0].turnAround_time = p[0].return_time - p[0].arrival_time;
	p[0].response_time = 0;
	p[0].completed = true;

	current_time = p[0].run_time;

	qsort_priority_time(p, n);

	for (int i = 0; i < n; i++) {
		if (strcmp(check_id, p[i].id) == 0) {
			i++;
		}

		p[i].response_time = current_time - p[i].arrival_time;
		p[i].return_time = current_time + p[i].run_time;
		p[i].turnAround_time = p[i].return_time - p[i].arrival_time;
		p[i].waiting_time = current_time - p[i].arrival_time;
		p[i].completed = true;

		current_time += p[i].run_time;
	}
}

void NPPS_print_gantt_chart(Process* p, int n, char* check_id)
{
```

```
printf("\tNon Preemptive Priority Scheduling\n");

printf("┌");
/* 상단 출력 */
for (int i = 0; i < n; i++)
{
	for (int j = 0; j < p[i].run_time; j++)
		printf("─");

	if (i == n - 1)
		printf("┐");
	else
		printf("┬");
}

printf("\n|");
/* 프로세스 이름 출력 */
for (int i = 0; i < n; i++)
{
	for (int j = 0; j < p[i].run_time - 1; j++)
		printf(" ");

	printf("%s", p[i].id);

	for (int j = 0; j < p[i].run_time - 1; j++)
		printf(" ");

	printf("|");
}

printf("\n");
printf("└");

/* 하단 바 출력 */
for (int i = 0; i < n; i++)
{
	for (int j = 0; j < p[i].run_time; j++)
		printf("─");


	if (i == n - 1)
		printf("┘");
	else
		printf("┴");
}

printf("\n");

/* 프로세스 시간 출력 */
printf("0");
// qsort_return_time(p, n);
for (int i = 0; i < n; i++)
```

```
	{
		for (int j = 0; j < p[i].run_time; j++)
			printf("  ");

		if (p[i].return_time > 9)
			printf("\b");
		printf("%d", p[i].return_time);

	}

	printf("\n");
}



void NPPS_Scheduling(Process* p, int pc) {
	int total_waiting_time = 0; // 총 프로세스 대기 시간
	int total_turnAround_time = 0; // 총 프로세스 소요 시간
	int total_response_time = 0; // 총 프로세스 응답 시간

	Process_init(p, pc);

	qsort_arrival_time(p, pc);

	char check_id[10];
	strcpy(check_id, p[0].id); // 가장 처음으로 들어온 PID 저장


	NPPS_Process_Time(p, pc, check_id);

	for (int i = 0; i < pc; i++) {
		total_response_time += p[i].response_time;
		total_waiting_time += p[i].waiting_time;
		total_turnAround_time += p[i].turnAround_time;
	}
	qsort_return_time(p, pc);

	NPPS_print_gantt_chart(p, pc, check_id);

	view_table(p, pc);

	printf("\n");
	printf("Average Total Waiting time = %.2lf\n", (double)total_waiting_time / (double)pc); // 평균 대기 시간
	printf("Average Total turnAround time = %.2lf\n", (double)total_turnAround_time / (double)pc); // 평균 반환
시간
	printf("Average Total Response time = %.2lf\n", (double)total_response_time / (double)pc); // 평균 응답 시
간
}

#endif __NPPS__
```

HRN.h

```
#ifndef __HRN__
#define __HRN__
/*
	헤더명 : HRN.h (Highest Response Ratio Next Scheduling)
	내용 : 각 작업의 우선순위로 스케쥴링
		우선순위 = (대기시간 + CPU 사용시간) / (CPU 사용시간)
		SJF에서 발생할 수 있는 아사 현상 해결
	입력 : 프로세스 포인터, 프로세스 갯수
	출력 : HRN 스케쥴링으로 인한
	각 프로세스별 대기시간, 평균 대기 시간,
	각 프로세스별 응답시간, 평균 응답시간,
	각 프로세스별 반환 시간, 평균 반환 시간
*/
#include "Process.h"
#include "View_Table.h"
#include "Function_to_Sorting.h"
#include <Windows.h>
#include <string.h>


void HRN_print_gantt_chart(Process* p, int n,char *check_id)
{
	printf("\tHighest Response Ratio Next\n");

	int check_num = 0; // 처음 실행된 PID를 추출하기 위한 변수

	for (int i = 0; i < n; i++)
	{
		if (strcmp(check_id, p[i].id) == 0) { // 처음 실행된 프로세스 id를 받아 현재 구조체에 있는 id 와 같은 인덱스 추출
			break;
		}
		check_num++;
	}

	Process_init(tmp, check_num);

	for (int i = 0; i < check_num; i++) {
		tmp[i].arrival_time = 0;
		tmp[i].return_time = 0;
		tmp[i].run_time = 0;
		tmp[i].priority = 0;
		strcpy(tmp[i].id, "");
	}

	for (int i = 0; i < check_num; i++) {
		tmp[i] = p[i]; // 구조체 복사
	}

	Process Temp = tmp[check_num]; // Temp에 추출된 인덱스가 해당하는 프로세스 저장
```

```
	for (int i = 0; i < check_num; i++) { // 가장 먼저 실행된 프로세스를 맨 앞으로 보내기 위해서 구조체 배열을 뒤로 민다.
		p[i + 1] = tmp[i];
	}

	p[0] = Temp; // 저장해두었던 프로세스를 다시 구조체 첫 번째 인덱스로 저장


	printf("┌");
	/* 상단 바 출력 */
	for (int i = 0; i < n; i++){
		for (int j = 0; j < p[i].run_time; j++)
			printf("─");

		if (i == n - 1)
			printf("┐");
		else
			printf("┬");

	}

	printf("\n|");

	/* 프로세스 이름 출력 */
	for (int i = 0; i < n; i++){
		for (int j = 0; j < p[i].run_time - 1; j++)
			printf(" ");

		printf("%s", p[i].id);

		for (int j = 0; j < p[i].run_time - 1; j++)
			printf(" ");

		printf("|");
	}

	printf("\n");
	printf("└");
	/* 하단 바 출력 */
	for (int i = 0; i < n; i++){
		for (int j = 0; j < p[i].run_time; j++)
			printf("─");

		if (i == n - 1)
			printf("┘");
		else
			printf("┴");
	}

	printf("\n");
```

```
	/* 프로세스 시간 출력 */
	printf("0");

	for (int i = 0; i < n; i++){
		for (int j = 0; j < p[i].run_time; j++)
			printf("  ");

		if (p[i].return_time > 9)
			printf("\b");

		printf("%d", p[i].return_time);

	}

	printf("\n");
}

void HRN_Scheduling(Process* p, int n) {
	int current_time = 0; // 현재시간 저장 할 변수
	int priority_index; // 우선순위를 가지는 인덱스
	int total_waiting_time = 0; // 총 프로세스 대기 시간
	int total_turnAround_time = 0; // 총 프로세스 소요 시간
	int total_response_time = 0; // 총 프로세스 응답 시간
	int total_run_time = 0; // 총 프로세스 실행 시간

	double hrr, temp; // 우선순위 저장 변수, 소숫점 가능

	Process_init(p, n);

	for (int i = 0; i < n; i++)	{
		total_run_time += p[i].run_time; // 총 실행 시간을 계산
	}

	qsort_arrival_time(p, n);

	char check_id[10];
	strcpy(check_id, p[0].id); // 가장 처음으로 들어온 PID 저장

	current_time = p[0].arrival_time; // 현재 시간은 첫 프로세스가 도착한 시간

	while(current_time < total_run_time){ // 증가된 현재 시간이 총 프로세스 시간보다 크다면 모든 프로세스를 돌았음.
		hrr = -9999.0;
		for (int i = 0; i < n; i++){
			if ((p[i].arrival_time <= current_time) && (p[i].completed == false)) {// 가장 처음 들어온 프로세스 부터 실행
				temp = ((double)p[i].run_time + ((double)current_time - (double)p[i].arrival_time)) / (double)p[i].run_time;
				// (실행시간 + 대기시간) / 실행시간 으로 우선순위 계산
				if (hrr < temp){ // 우선순위 갱신
					hrr = temp;
					priority_index = i; // 해당 인덱스로 갱신
```

```
                        }
                    }
                }

                current_time += p[priority_index].run_time; // 서비스 시간 증가

                p[priority_index].waiting_time = current_time - p[priority_index].arrival_time -
p[priority_index].run_time;
                p[priority_index].turnAround_time = current_time - p[priority_index].arrival_time;
                p[priority_index].return_time = p[priority_index].turnAround_time + p[priority_index].arrival_time;
                p[priority_index].response_time = p[priority_index].waiting_time;
                p[priority_index].completed = true;

                total_waiting_time += p[priority_index].waiting_time;
                total_turnAround_time += p[priority_index].turnAround_time;
                total_response_time += p[priority_index].response_time;

        }

        qsort_run_time(p, n);

        HRN_print_gantt_chart(p, n, check_id);

        view_table(p, n);

        printf("\n\tAverage Waiting Time      : %-2.2lf\n", (double)total_waiting_time / (double)n);
        printf("\tAverage Turnaround Time  : %-2.2lf\n", (double)total_turnAround_time / (double)n);
        printf("\tAverage Response Time    : %-2.2lf\n\n", (double)total_response_time / (double)n);
}

#endif __HRN__
```

PPS.h

```
#ifndef __PPS__
#define __PPS__

/*
	헤더명 : PPS.h(Preemptive Priority Scheduling)
	내용 : 현재 실행중인 프로세스보다 높은 우선순위의 프로세스가 등장하면 스케줄러에 의한 실행순서 조정이 즉각적으로 가해짐(선점형)
	입력 : 프로세스 포인터, 프로세스 갯수
	출력 : PPS 스케쥴링으로 인한
	각 프로세스별 대기시간, 평균 대기 시간,
	각 프로세스별 응답시간, 평균 응답시간,
	각 프로세스별 반환 시간, 평균 반환 시간
*/

/*
if priority(current_process) > priority(prior_process) :- then execute the current process.
if priority(current_process) < priority(prior_process) :- then execute the prior process.
if priority(current_process) = priority(prior_process) :- then execute the process which arrives
*/

#include "Process.h"
#include <Windows.h>
#include "View_Table.h"
#include "Function_to_Sorting.h"

void PPS_print_gantt_chart(Process* p, int n) {
	printf("\tPreemptive Priority Scheduling\n");
	int current_time = 0, previous_time = 0; // 현재 프로세스 시간과 이전 프로세스 시간 변수
	int total_run_time = 0; // 총 실행 시간을 저장할 변수
	int k = 0, pre_k= 0;// 현재와 이전 프로세스 번호

	int priority, distance; // 우선순위가 두 프로세스간 거리를 나타낼 변수

	int* remain_run_time = NULL;
	remain_run_time = (int*)malloc(sizeof(int) * n); // 남은시간을 저장할 배열 동적 할당

	if (remain_run_time == NULL) {		// 메모리 할당 실패
		printf("Not enough memory!");	// 오류 상황 알림
		return -1;			// 함수 종료
	}

	int* response = NULL;
	response = (int*)malloc(sizeof(int) * n); // 응답 시간을 확인할 배열 동적 할당

	if (response == NULL) {		// 메모리 할당 실패
		printf("Not enough memory!");	// 오류 상황 알림
		return -1;			// 함수 종료
	}

	for (int i = 0; i < n; i++) {
```

```
                response[i] = 0;
                remain_run_time[i] = p[i].run_time;
                total_run_time += p[i].run_time;
                p[i].completed = false; // 완료상태 초기화
        }
        /* 상단 바 출력 */
        printf("┌");
        while (current_time < total_run_time) // 현재시간이 총 실행시간보다 작을때 까지 반복
        {
                priority = INT_MAX; // 우선순위 초기화

                if (current_time <= p[n - 1].arrival_time){ // 모든 프로세스가 도착할 때 까지 계산
                        for (int i = 0; i < n; i++){
                                if ((p[i].completed == false) && (p[i].arrival_time <= current_time)) { // 완료
되지 않았고 도착시간이 현재시간보다 작은경우
                                        if (priority > p[i].priority){ // 우선순위 갱신
                                                priority = p[i].priority;
                                                k = i; // 해당 인덱스 저장
                                        }
                                }
                        }
                }else{ // 모든 프로세스 도착한 경우
                        for (int i = 0; i < n; i++){
                                if (p[i].completed == false){ // 완료되지 않은 프로세스에 대하여
                                        if (priority > p[i].priority){ // 우선순위 갱신
                                                priority = p[i].priority;
                                                k = i;
                                        }
                                }
                        }
                }

                if (pre_k != k) { // 이전 프로세스와 현재 프로세스가 같지 않은경우
                        if (current_time == 0) {
                                printf(" ");
                        }
                        else {
                                printf("┬");
                        }
                }

                printf("─");

                remain_run_time[k]--; // 남은 실행시간 감소
                current_time++; // 현재시간 증가
                pre_k = k; // 이전 프로세스로 저장(연결리스트 처럼 활용)

                if (remain_run_time[k] == 0) // 남은 시간이 없는경우
                        p[k].completed = true; // 완료표시
        }

        /* 출력을 위한 초기화 과정 진행 */
```

```
for (int i = 0; i < n; i++){
        remain_run_time[i] = p[i].run_time;
        p[i].completed = false;
}
printf("┌");
printf("\n| ");
current_time = 0;

/* 프로세스 ID 출력 (상단 출력 알고리즘과 동일 )*/
while (current_time < total_run_time){
        priority = INT_MAX;
        if (current_time <= p[n - 1].arrival_time){
                for (int i = 0; i < n; i++){
                        if ((p[i].completed == false) && (p[i].arrival_time <= current_time)){
                                if (priority > p[i].priority){
                                        priority = p[i].priority;
                                        k = i;
                                }
                        }
                }
        }else{
                for (int i = 0; i < n; i++){
                        if (p[i].completed == false){
                                if (priority > p[i].priority){
                                        priority = p[i].priority;
                                        k = i;
                                }
                        }
                }
        }
        if (current_time == 0){
                response[k]++;
                printf("  ");
        }else{
                if (pre_k != k){ // 다른 프로세스 인 경우
                        distance = response[pre_k] + 1; // 두 프로세스 시간 차이 저장
                        response[pre_k] = 0; // 이전 프로세스 카운트 초기화
                        response[k]++; // 현재 프로세스 카운트 증가

                        for (int i = 0; i < distance ; i++) // 간격을 맞추어 공백 출력
                                printf("\b");

                        printf("%2s", p[pre_k].id); // 이전 프로세스 ID 출력

                        for (int i = 0; i < distance - 2; i++) // 간격을 맞추어 공백 출력
                                printf(" ");

                        printf("|  ");
                }else // 같은 프로세스일 경우
                {
                        response[k]++; // 현재 프로세스 증가
                        printf("  "); // 공백 출력
```

```
                        /* 마지막 프로세스 실행일 경우 */
                        if (current_time == total_run_time - 1){
                                distance = response[pre_k] + 1;
                                response[pre_k] = 0;
                                response[k]++;

                                for (int i = 0; i < distance; i++) {
                                        printf("\b");
                                }

                                printf("%2s", p[pre_k].id);

                                for (int i = 0; i < distance - 2; i++) {
                                        printf(" ");
                                }
                        }
                }
        }

        pre_k = k;
        remain_run_time[k]--;
        current_time++;

        if (remain_run_time[k] == 0)
                p[k].completed = true;
}

/* 출력을 위한 초기화 과정 진행 */
for (int i = 0; i < n; i++){
        remain_run_time[i] = p[i].run_time;
        p[i].completed = false;
}
current_time = 0;
printf("|");
printf("\n");
printf("└");

/* 하단 출력 (상단 출력 알고리즘과 동일)*/
while (current_time < total_run_time){
        priority = INT_MAX;

        if (current_time <= p[n - 1].arrival_time){
                for (int i = 0; i < n; i++){
                        if ((p[i].completed == false) && (p[i].arrival_time <= current_time)){
                                if (priority > p[i].priority){
                                        priority = p[i].priority;
                                        k = i;
                                }
                        }
                }
        }else{
                for (int i = 0; i < n; i++){
```

```
                        if (p[i].completed == false){
                                if (priority > p[i].priority){
                                        priority = p[i].priority;
                                        k = i;
                                }
                        }
                }
        }

        if (pre_k != k) {
                if (current_time == 0) {
                        printf(" ");
                }
                else {
                        printf("┴");
                }
        }
        printf("—");

        remain_run_time[k]--;
        current_time++;
        pre_k = k;

        if (remain_run_time[k] == 0)
                p[k].completed = true;
}

for (int i = 0; i < n; i++){
        remain_run_time[i] = p[i].run_time;
        p[i].completed = false;
}

current_time = 0;
distance = 0;
printf("┘");
printf("\n");

/* 프로세스 ID 출력과 같은 방법으로 실행하며 시간 출력 */
while (current_time <= total_run_time){
        if (total_run_time != current_time){
                priority = INT_MAX;
                if (current_time <= p[n - 1].arrival_time){
                        for (int i = 0; i < n; i++){
                                if ((p[i].completed == false) && (p[i].arrival_time <= current_time)){
                                        if (priority > p[i].priority){
                                                priority = p[i].priority;
                                                k = i;
                                        }
                                }
                        }
                }else{
                        for (int i = 0; i < n; i++){
```

```
                                        if ((p[i].completed == false) && (priority > p[i].priority)){
                                                priority = p[i].priority;
                                                k = i;
                                        }
                                }
                        }

                        if (pre_k != k){
                                for (int i = 0; i < distance && current_time != 0; i++)
                                        printf("  ");

                                if (current_time != 0)
                                        printf(" ");

                                printf("%-2d", current_time);
                                distance = 0;

                                previous_time = current_time;
                        }
                        else {
                                distance++;
                        }
                        remain_run_time[k]--;
                        current_time++;
                        pre_k = k;

                        if (remain_run_time[k] == 0)
                                p[k].completed = true;
                }
                else
                {
                        for (int i = 0; i < current_time - previous_time - 1; i++)
                                printf("  ");
                        printf(" ");

                        printf("%2d", current_time);

                        break;
                }
        }

        printf("\n");
        if (response != NULL)
                free(response);

        if (remain_run_time != NULL)
                free(remain_run_time);

}

void PPS_Process_Time(Process* p, int n) {
        int current_time = 0; // 현재까지 누적 시간을 저장할 변수
```

```
        int priority; // 우선순위를 저장할 변수
        int total_run_time = 0; // 총 실행 시간을 저장할 변수
        int k = 0;// 현재 실행할 프로세스 번호


        int* remain_run_time = NULL;
        remain_run_time = (int*)malloc(sizeof(int) * n); // 남은시간을 저장할 배열 동적 할당

        if (remain_run_time == NULL) {       // 메모리 할당 실패
                printf("Not enough memory!");     // 오류 상황 알림
                return -1;                        // 함수 종료
        }

        int* response = NULL;
        response = (int*)malloc(sizeof(int) * n); // 응답 시간을 확인할 배열 동적 할당

        if (response == NULL) {       // 메모리 할당 실패
                printf("Not enough memory!");     // 오류 상황 알림
                return -1;                        // 함수 종료
        }
        // qsort_arrival_time(p, n);

        for (int i = 0; i < n; i++){
                response[i] = 0;
                remain_run_time[i] = p[i].run_time;
                total_run_time += p[i].run_time;
        }

        /* 현재 시간이 총 실행 시간이 되기 전까지 반복 */
        while (current_time < total_run_time){ // 현재 시간이 총 실행시간보다 작은 경우 반복

                priority = INT_MAX; // 우선순위 초기화

                if (current_time < p[n - 1].arrival_time) { // 프로세스 도착 검사
                        for (int i = 0; i < n; i++) {
                                /* 완료되지 않았으며 도착시간이 현재시간보다 작거나 같으며 현재 우선순위보다 우
선순위가 작을 경우 */
                                if ((p[i].completed == false) && (p[i].arrival_time <= current_time) &&
(priority > p[i].priority)) {
                                        priority = p[i].priority; // 우선순위 갱신
                                        k = i;
                                }
                        }
                }else { // 모든 프로세스가 도착한 경우
                        for (int i = 0; i < n; i++){
                                /* 완료되지 않았으며 현재 우선순위보다 우선순위가 작을 경우 */
                                if ((p[i].completed == false) && (priority > p[i].priority)){
                                        priority = p[i].priority;
                                        k = i;
                                }
                        }
```

```
                }

                // k는 알고리즘을 거쳐 선택된 프로세스


                if (response[k] == 0){ // 선택된 프로세스가 처음 시작될 경우
                        response[k]++; // 초기 실행이 아님을 표시
                        p[k].response_time = current_time; // 실행중인 프로세스의 응답시간 저장
                }

                remain_run_time[k]--; // k 프로세스 남은시간 감소
                current_time++; // 현재 시간 증가


                if (remain_run_time[k] == 0){ // k 프로세스가 완료된 경우
                        p[k].completed = true; // 완료 상태로 변경
                        p[k].waiting_time = current_time - p[k].run_time - p[k].arrival_time; // 대기 시간 계산

                        p[k].return_time = current_time; // 반환 시간 계산

                }
        }

        /* 동적 할당한 배열의 메모리 할당 해제 */
        free(remain_run_time);
        free(response);
}


void PPS_Scheduling(Process* p, int pc) {
        int total_waiting_time = 0; // 총 프로세스 대기 시간
        int total_turnAround_time = 0; // 총 프로세스 소요 시간
        int total_response_time = 0; // 총 프로세스 응답 시간

        Process_init(p, pc);

        qsort_arrival_time(p, pc);

        PPS_Process_Time(p, pc);

        for (int i = 0; i < pc; i++) {
                p[i].turnAround_time = p[i].return_time - p[i].arrival_time; // 각 프로세스 소요시간 계산
                total_waiting_time += p[i].waiting_time; // 총 대기시간 저장
                total_turnAround_time += p[i].turnAround_time; // 총 소요시간 저장
                total_response_time += p[i].response_time; // 총 응답시간 저장
        }

        qsort_return_time(p, pc);

        PPS_print_gantt_chart(p, pc);

        view_table(p, pc);
```

```
        printf("\n");
        printf("Average Total Waiting time = %.2lf\n", (double)total_waiting_time / (double)pc); // 평균 대기 시간
        printf("Average Total turnAround time = %.2lf\n", (double)total_turnAround_time / (double)pc); // 평균 반환
시간
        printf("Average Total Response time = %.2lf\n", (double)total_response_time / (double)pc); // 평균 응답 시
간
}

#endif __PPS__
```

RR.h

```
#ifndef __RR__
#define __RR__
/*
	헤더명 : RR.h(Round Robin Scheduling)
	내용 : 한 프로세스가 할당받은 시간(타임 슬라이스Time Slice/Time Quantum)
		동안작업을 하다가 작업을 완료하지 못하면 준비 큐의 맨 뒤로 가서 자기 차례를 기다리는 방식
	입력 : 프로세스 포인터, 프로세스 갯수 , 타임 슬라이스(퀀텀)
	출력 : RR 스케쥴링으로 인한
	각 프로세스별 대기시간, 평균 대기 시간,
	각 프로세스별 응답시간, 평균 응답시간,
	각 프로세스별 반환 시간, 평균 반환 시간
*/

#include "Process.h"
#include <Windows.h>
#include "View_Table.h"
#include "Function_to_Sorting.h"

void RR_Process_Waiting_Time(Process* p, int n, Quantum quantum) { // 대기 시간 계산 함수
	int current_time = 0; // 현재시간 초기화
	int total_run_time = 0; // 총 실행시간 초기화
	int* remain_run_time = NULL;
	remain_run_time = (int*)malloc(sizeof(int) * n); // 남은 시간 저장 배열 할당

	if (remain_run_time == NULL) {
		fprintf(stderr, "메모리 할당 실패\n");
		exit(1);
	}

	bool* check_response_time = NULL;
	check_response_time = (bool*)malloc(sizeof(bool) * n); // 응답 시간 처리 확인 배열 할당

	if (check_response_time == NULL) {
		fprintf(stderr, "메모리 할당 실패\n");
		exit(1);
	}


	for (int i = 0; i < n;i++) {
		remain_run_time[i] = p[i].run_time; // 각 프로세스별 남은시간 저장
		check_response_time[i] = false; // 프로세스 응답 초기화
		total_run_time += p[i].run_time; // 총 실행시간 계산
	}
	while (true) { // 모든 프로세스가 완료될 때 까지 반복
		bool check = true;
		for (int i = 0; i < n; i++) {
			if (remain_run_time[i] > 0) { // 실행 시간이 남아 있는 경우
				check = false;

				if (check_response_time[i] == false) { // 응답 시간 처리 안한 경우
```

```
                                        p[i].response_time = current_time - p[i].arrival_time; // 응답시간 계산
                                        check_response_time[i] = true;
                                }

                                if (remain_run_time[i] > quantum) { // 남은 시간이 타임 슬라이스 보다 클 경우
                                        current_time += quantum; // 현재 시간 증가
                                        remain_run_time[i] -= quantum; // 현재 실행중인 프로세스 남은 시간 감
소
                                }
                                else { // 남은 시간이 타임 슬라이스 보다 작은 경우
                                        current_time += remain_run_time[i]; // 현재시간 증가
                                        p[i].waiting_time = current_time - p[i].run_time; // 대기시간 계산
                                        remain_run_time[i] = 0; // 작업 끝난 프로세스 남은 시간 0으로 초기화
                                }
                        }
                }

                if (check == true) // 남아 있는 프로세스가 없는 경우
                        break;
        }
        /* 동적할당한 메모리 해제 */
        free(remain_run_time);
        free(check_response_time);
}

void RR_Process_TurnAround_Time(Process* p, int n) { // 소요 시간 계산 함수
        for (int i = 0; i < n; i++) {
                p[i].turnAround_time = p[i].run_time + p[i].waiting_time - p[i].arrival_time;
        }
}

        /* 대기시간 계산 함수 알고리즘과 동일 */
void RR_print_gantt_chart(Process* p, int n, Quantum quantum)
{

        int current_time = 0, total_run_time = 0; // 현재 시간과 총 실행 시간을 저장할 변수

        int* remain_run_time = NULL;
        remain_run_time = (int*)malloc(sizeof(int) * n);

        if (remain_run_time == NULL) {
                fprintf(stderr, "메모리 할당 실패\n");
                exit(1);
        }

        for (int i = 0; i < n; i++){
                remain_run_time[i] = p[i].run_time;
                total_run_time += p[i].run_time;
        }

        printf("┌");
```

```
/* 상단 바 출력 */
while (true){
        bool check = true;
        for (int i = 0; i < n; i++){
                if (remain_run_time[i] > 0){
                        check = false;
                        if (remain_run_time[i] < quantum){
                                for (int j = 0; j < remain_run_time[i]; j++)
                                        printf("");
                        }else{
                                for (int j = 0; j < quantum ; j++)
                                        printf("—");
                        }

                        if (remain_run_time[i] > quantum){
                                current_time += quantum;
                                remain_run_time[i] -= quantum;
                                printf("—");
                        }else{
                                current_time += remain_run_time[i];
                                p[i].waiting_time = current_time - p[i].run_time;
                                remain_run_time[i] = 0;
                        }
                }
        }
        if (check == true)
                break;
}

printf("┐\n");

for (int i = 0; i < n; i++){
        remain_run_time[i] = p[i].run_time;
}

/* 프로세스 아이디 출력 */
while (true){
        int check = true;
        for (int i = 0; i < n; i++){
                if (remain_run_time[i] > 0){
                        check = false;
                        if (remain_run_time[i] < quantum){
                                printf(" | ");
                                if (remain_run_time[i] != 1){
                                        printf(" %s ", p[i].id);
                                }
                                else {
                                        printf(" %s ", p[i].id);
                                }
                        }else{
                                printf("|");
                                printf(" %s ", p[i].id);
```

```
                        }

                        if (remain_run_time[i] > quantum){
                                current_time += quantum;
                                remain_run_time[i] -= quantum;
                        }else
                        {
                                current_time += remain_run_time[i];
                                p[i].waiting_time = current_time - p[i].run_time;
                                remain_run_time[i] = 0;
                        }
                }
        }

        if (check == true)
                break;
}

printf("┃\n");

for (int i = 0; i < n; i++){
        remain_run_time[i] = p[i].run_time;
}


printf("┗");

/* 하단 바 출력 */
while (true){
        bool check = true;
        for (int i = 0; i < n; i++){
                if (remain_run_time[i] > 0){
                        check = false;

                        if (remain_run_time[i] < quantum){
                                for (int j = 0; j < remain_run_time[i]; j++) {
                                        printf("1");
                                }
                        }else{
                                for (int j = 0; j < quantum; j++) {
                                        printf("━");
                                }
                        }

                        if (remain_run_time[i] > quantum){
                                current_time += quantum;
                                remain_run_time[i] -= quantum;
                                printf("━");
                        }else{
                                current_time += remain_run_time[i];
                                p[i].waiting_time = current_time - p[i].run_time;
                                remain_run_time[i] = 0;
```

```
                        }
                }
        }

        if (check == true)
                break;
}

printf("┘\n");
for (int i = 0; i < n; i++)
        remain_run_time[i] = p[i].run_time;

current_time = 0;

/* 프로세스 시간 출력 */
while (true){
        int check = true;

        for (int i = 0; i < n; i++){
                if (remain_run_time[i] > 0){
                        check = false;

                        if (remain_run_time[i] < quantum){
                                printf("%2d", current_time);
                        }else{
                                printf("%2d", current_time);

                                for (int j = 0; j < quantum / 2; j++) {
                                        printf("   ");
                                }
                        }
                        if (remain_run_time[i] > quantum){
                                current_time += quantum;
                                remain_run_time[i] -= quantum;
                        }else{
                                current_time += remain_run_time[i];
                                p[i].waiting_time = current_time - p[i].run_time;
                                remain_run_time[i] = 0;
                        }
                }
        }

        if (check == true)
                break;
}

printf("%d", total_run_time);

printf("\n");

free(remain_run_time);
// 동적 할당한 배열의 메모리 할당 해제
```

```
}

void RR_Scheduling(Process* p, int pc, Quantum quantum) {

	int total_waiting_time = 0; // 총 프로세스 대기 시간
	int total_turnAround_time = 0; // 총 프로세스 소요 시간
	int total_response_time = 0; // 총 프로세스 응답 시간

	Process_init(p, pc);

	qsort_arrival_time(p, pc);

	RR_Process_Waiting_Time(p, pc, quantum);

	RR_Process_TurnAround_Time(p, pc);

	for (int i = 0; i < pc; i++){
		p[i].waiting_time = p[i].turnAround_time - p[i].run_time; // 대기 시간 계산 후 저장
		p[i].return_time = p[i].arrival_time + p[i].run_time + p[i].waiting_time; // 반환 시간 계산 후 저장

		total_waiting_time += p[i].waiting_time; // 총 대기 시간 증가
		total_turnAround_time += p[i].turnAround_time; // 총 소요 시간  증가
		total_response_time += p[i].response_time; // 총 응답 시간 증가
	}

	printf("\tRound Robin Scheduling\n");

	RR_print_gantt_chart(p, pc, quantum);

	view_table(p, pc);
	printf("\n");

	printf("Average Total Waiting time = %.2lf\n", (double)total_waiting_time / (double)pc); // 평균 대기 시간
	printf("Average Total turnAround time = %.2lf\n", (double)total_turnAround_time / (double)pc); // 평균 반환
시간
	printf("Average Total Response time = %.2lf\n", (double)total_response_time / (double)pc); // 평균 응답 시
간

}
#endif
```

SRT.h

```
#ifndef __SRT__
#define __SRT__

/*
	헤더명 : SRT.h(Shortest Remaining Time Scheduling)
	내용 : CPU를 할당받을 프로세스를 선택할 때 남아 있는 작업 시간이 가장 적은 프로세스를 선택
	입력 : 프로세스 포인터, 프로세스 갯수
	출력 : SJF 스케쥴링으로 인한
	각 프로세스별 대기시간, 평균 대기 시간,
	각 프로세스별 응답시간, 평균 응답시간,
	각 프로세스별 반환 시간, 평균 반환 시간
*/

#include "Process.h"
#include "View_Table.h"
#include "Function_to_Sorting.h"

void SRT_Process_System(Process* p, int n) {
	int current_time = 0;

	int total_run_time = 0;

	int shortest_remain_time;

	int k = 0;

	int* remain_run_time = NULL;
	remain_run_time = (int*)malloc(sizeof(int) * n);
	if (remain_run_time == NULL) {
		fprintf(stderr, "메모리 할당 실패\n");
		exit(1);
	}

	int* check_response = NULL;
	check_response = (int*)malloc(sizeof(int) * n);
	if (check_response == NULL) {
		fprintf(stderr, "메모리 할당 실패\n");
		exit(1);
	}

	for (int i = 0; i < n; i++){
		check_response[i] = 0; //
		remain_run_time[i] = p[i].run_time;
		total_run_time += p[i].run_time;
	}

	/* 현재 시간이 총 실행 시간이 되기 전까지 반복 */
	while (current_time < total_run_time)	{
		shortest_remain_time = INT_MAX; // 최소작업 인덱스를 INT_MAX로 초기화
```

```
                /* 가장 마지막에 들어온 프로세스의 도착시간 보다 작을 경우 */
                if (current_time <= p[n - 1].arrival_time) {
                        for (int i = 0; i < n; i++) {
                                /* 완료되지 않았으며 도착시간이 현재시간보다 작거나 같으며 현재 최소작업 시간보
다 남은 실행시간이 작을 경우 */
                                if ((p[i].completed == false) && (p[i].arrival_time <= current_time) &&
(shortest_remain_time > remain_run_time[i])){
                                        shortest_remain_time = remain_run_time[i]; // 최소 작업 시간 갱신
                                        k = i; // 최소 작업 프로세스 인덱스 갱신
                                }
                        }
                }
                else
                {
                        for (int i = 0; i < n; i++){ // 완료되지 않았으며 현재 최소작업 시간보다  남은 실행시간
이 작을 경우
                                if ((p[i].completed == false) && (shortest_remain_time > remain_run_time[i]))
{
                                        shortest_remain_time = remain_run_time[i]; // 최소 작업 시간 갱신b
                                        k = i; // 최소 작업 프로세스 인덱스 갱신
                                }
                        }
                }

                /* 선택된 프로세스가 처음 시작될 경우 */
                if (check_response[k] == 0){
                        check_response[k] = 1; // 응답 표시
                        p[k].response_time = current_time; // 실행중인 프로세스의 응답시간 저장
                }

                remain_run_time[k]--;
                // 실행된 프로세스의 남은 시간 감소
                current_time++;
                // 현재 시간 증가

                /* 프로세스의 남은 실행 시간이 0이될 경우 */
                if (remain_run_time[k] == 0){
                        p[k].completed = true; // 완료 상태로 변경
                        p[k].waiting_time = current_time - p[k].run_time - p[k].arrival_time; // 대기 시간 계산

                        p[k].return_time = current_time; // 반환 시간 계산
                }
        }

        /* 동적 할당한 배열의 메모리 할당 해제 */
        free(check_response);
        free(remain_run_time);
}

void SRT_print_gantt_chart(Process* p, int n) {
        int current_time = 0, previous_time = 0;
```

```
	int total_run_time = 0;

	int shortest_remain_time, distance;

	int k, pre_k = 0;

	int* remain_run_time = (int*)malloc(sizeof(int) * n);

	int* check_response = (int*)malloc(sizeof(int) * n);

	for (int i = 0; i < n; i++) {
		check_response[i] = 0;
		remain_run_time[i] = p[i].run_time;
		total_run_time += p[i].run_time;
		p[i].completed = false;
	}

	printf("┌");
	while (current_time < total_run_time) {
		shortest_remain_time = INT_MAX;
		if (current_time <= p[n - 1].arrival_time) { // if - a
			for (int i = 0; i < n; i++) {
				if ((p[i].completed == false) && (p[i].arrival_time <= current_time)){
					if (shortest_remain_time > remain_run_time[i]) {
						shortest_remain_time = remain_run_time[i]; // 최소 작업 시간 갱
신

						k = i; // 최소 작업 프로세스 인덱스 갱신
					}
				}
			}
		}
		else { // else - a
			for (int i = 0; i < n; i++) {
				if (p[i].completed == false){
					if (shortest_remain_time > remain_run_time[i]) {
						shortest_remain_time = remain_run_time[i];
						k = i;
					}
				}
			}
		}
		if (pre_k != k)
			printf(" ");


		printf("—");


		remain_run_time[k]--;
		current_time++;
		pre_k = k;
```

```
            if (remain_run_time[k] == 0) {
                    for (int i = 0; i < n / 2; i++)
                            printf("—");
                    p[k].completed == true;
            }
    }
    printf("┐");

    for (int i = 0; i < n; i++)
    {
            remain_run_time[i] = p[i].run_time;
            p[i].completed = false;
    }

    current_time = 0;
    printf("\n|");
    while (current_time <= total_run_time){
            if (current_time != total_run_time){ // 현재 시간이 총 실행시간과 다를 경우
                    shortest_remain_time = INT_MAX;
                    if (current_time <= p[n - 1].arrival_time){ // if - a
                            for (int i = 0; i < n; i++){
                                    if ((p[i].completed == false) && (p[i].arrival_time <= current_time
)&& (shortest_remain_time > remain_run_time[i])){
                                            shortest_remain_time = remain_run_time[i];
                                            k = i;
                                    }
                            }
                    }
                    else // end - a
                    {
                            for (int i = 0; i < n; i++){
                                    if ((p[i].completed == false) && (shortest_remain_time >
remain_run_time[i])){
                                            shortest_remain_time = remain_run_time[i];
                                            k = i;
                                    }
                            }
                    }

                    if (current_time == 0)
                    {
                            check_response[k]++;
                            printf("  ");
                    }
                    else
                    {
                            if (pre_k != k){ // 이전 프로세스와 다른 경우
                                    distance = check_response[pre_k] + 1; // 두 프로세스 시간 차이 저장

                                    check_response[pre_k] = 0; // 이전 프로세스 카운트 초기화

                                    check_response[k]++; // 현재 프로세스 카운트 증가
```

```
                                for (int i = 0; i < distance; i++)
                                        printf("\b");

                                printf("%s", p[pre_k].id);

                                for (int i = 0; i < distance - 2; i++)
                                        printf(" ");

                                printf("|  ");
                        }

                        /* 같은 프로세스일 경우 */
                        else
                        {
                                check_response[k]++; // 현재 프로세스 카운트 증가
                                printf("  ");

                        }
                }

                pre_k = k;
                remain_run_time[k]--;
                current_time++;

                if (remain_run_time[k] == 0)
                        p[k].completed = true;
        }
        else // 현재 실행시간이 총 실행시간과 같을 경우
        {
                for (int i = 0; i < check_response[pre_k] + n; i++)
                        printf("\b");
                printf("%s", p[k].id);
                for (int i = 0; i < check_response[pre_k] - 1; i++)
                        printf(" ");

                break;
        }
}
for (int i = 0; i < check_response[pre_k] - (check_response[pre_k] / 2 ); i++)
        printf("\b");
for (int i = 0; i < n - 1; i++)
        printf("\b");


for (int i = 0; i < n; i++){
        remain_run_time[i] = p[i].run_time;
        p[i].completed = false;
}

current_time = 0;
printf("|\n");
```

```
        printf("└");

        /* 동일 알고리즘을 사용하여 하단 바 출력 */
        while (current_time < total_run_time){
                shortest_remain_time = INT_MAX;
                if (current_time <= p[n - 1].arrival_time){
                        for (int i = 0; i < n; i++){
                                if ((p[i].completed == false) && (p[i].arrival_time <= current_time) &&
(shortest_remain_time > remain_run_time[i])){
                                        shortest_remain_time = remain_run_time[i];
                                        k = i;
                                }
                        }
                }
                else
                {
                        for (int i = 0 ; i < n; i++){
                                if ((p[i].completed == false) && (shortest_remain_time > remain_run_time[i])){
                                        shortest_remain_time = remain_run_time[i];
                                        k = i;
                                }
                        }
                }

                if (pre_k != k)
                        printf("-");

                printf("—");

                remain_run_time[k]--;
                current_time++;
                pre_k = k;

                if (remain_run_time[k] == 0)
                        p[k].completed = true;
        }

        for (int i = 0; i < n; i++){
                remain_run_time[i] = p[i].run_time;
                p[i].completed = false;
        }

        current_time = 0;
        printf("\b┘\n");

        /* 프로세스 ID 출력과 같은 알고리즘으로 실행하며 시간 출력 */
        while (current_time <= total_run_time){
                if (total_run_time != current_time){
                        shortest_remain_time = INT_MAX;

                        if (current_time <= p[n - 1].arrival_time){
                                for (int i = 0; i < n; i++){
```

```
                                        if ((p[i].completed == false) && (p[i].arrival_time <= current_time)
&& (shortest_remain_time > remain_run_time[i])){
                                                shortest_remain_time = remain_run_time[i];
                                                k = i;
                                        }
                                }
                        }
                        else
                        {
                                for (int i = 0; i < n; i++){
                                        if    ((p[i].completed    ==    false)    &&    (shortest_remain_time    >
remain_run_time[i])){
                                                shortest_remain_time = remain_run_time[i];
                                                k = i;
                                        }
                                }
                        }


                        if (pre_k != k)
                        {
                                for (int i = 0; i < distance && current_time != 0; i++)
                                        printf("  ");

                                if (current_time != 0)
                                        printf(" ");

                                printf("%-2d", current_time);
                                distance = 0;

                                previous_time = current_time;
                        }

                        else
                                distance++;

                        remain_run_time[k]--;
                        current_time++;
                        pre_k = k;

                        if (remain_run_time[k] == 0)
                                p[k].completed = true;
                }

                else
                {
                        for (int i = 0; i < current_time - previous_time - 1; i++)
                                printf("  ");
                        printf(" ");

                        printf("%-2d", current_time);
```

```
                        break;
                }
        }

        printf("\n");

        /* 동적 할당한 배열 메모리 할당 해제 */
        free(check_response);
        free(remain_run_time);
}

void SRT_Scheduling(Process* p, int pc) {
        int total_waiting_time = 0; // 총 프로세스 대기 시간
        int total_turnAround_time = 0; // 총 프로세스 소요 시간
        int total_response_time = 0; // 총 프로세스 응답 시간

        Process_init(p, pc);

        qsort_arrival_time(p, pc);

        SRT_Process_System(p, pc);

        for (int i = 0; i < pc ; i++) {
                p[i].turnAround_time = p[i].return_time - p[i].arrival_time; // 각 프로세스 소요 시간 계산
                total_waiting_time += p[i].waiting_time; // 총 대기시간 저장
                total_turnAround_time += p[i].turnAround_time; // 총 소요시간
                total_response_time += p[i].response_time;
        }
        printf("\tShortest Remaining Time Algorithm\n");
        SRT_print_gantt_chart(p, pc);

        printf("\n");
        view_table(p, pc);
        printf("\n");

        printf("Average Total Waiting time = %.2lf\n", (double)total_waiting_time / (double)pc); // 평균 대기 시간
        printf("Average Total turnAround time = %.2lf\n", (double)total_turnAround_time / (double)pc); // 평균 반환
시간
        printf("Average Total Response time = %.2lf\n", (double)total_response_time / (double)pc); // 평균 응답 시
간
}

#endif __SRT__
```

10. 느낀 점

운영체제의 다양한 스케줄링 알고리즘에 대해 직접 구현해 보면서 가장 좋은 성능을 가질 수 있는 알고리즘이 무엇인가에 대해 생각할 수 있는 좋은 과제였다고 생각합니다. 컴퓨터의 하드웨어가 진화해오고 그에 따른 소프트웨어도 진화해 가면서 하드웨어의 성능에 따라가다 보니 비효율적인 알고리즘이 등장했다고도 느껴졌습니다.

현재 레포트를 작성하는 중에도 컴퓨터에서는 한 타임마다 프로세스의 작업이 진행되고 있다는 것을 깨달은 후 미처 알지 못했던 부분에서 컴퓨터의 작업이 이루어지고 있었다는 것에 놀라웠습니다. 아직 운영체제의 알고리즘을 작성하고 구현하기엔 부족한 실력이지만, 개발자로서의 실력을 쌓고 본인만의 운영체제 알고리즘을 구현해 보고 싶습니다.

개발 언어가 C언어여서 코드 라인이 많이 늘어난 감이 있지만 다룰 수 있는 개발 언어 중에 C언어만큼 직관적이고 가벼운 언어가 가장 본인에게 맞았기 때문에 C언어를 선택하였습니다. 개발하는 과정에서 “자바로 한다면 편하지 않을까.”라는 생각도 자주 했었지만 자바 보다 C언어가 재밌기 때문에 선택했던 것도 있습니다.

처음 프로젝트를 시작할 때 막막하였지만 헤더를 나누고 구조체를 선언하고 포인터를 사용함으로써 조금씩 실마리가 보이기 시작했으며 알고리즘을 작성하기 전에 노트나 머릿속으로 알고리즘을 구현한 후에 개발을 시작하니 크게 막힌 부분은 없던 것 같습니다.

기회가 된다면, 프로세스가 진입하는 다양한 경우를 추가하여 여러 알고리즘을 구현해 보고 싶습니다. 이번 프로젝트를 통해서 CPU 스케줄링에 대해 깊게 공부할 수 있었고 컴퓨터 구조에 대해서도 다시 한번 생각해보게 되는 계기가 되었습니다.