

# 운영체제



제출일	2023.05.30	전 공	컴퓨터소프트웨어공학과
과 목	운영체제	학 번	20194018
담당교수	김대영 교수님	이 름	최 안 용

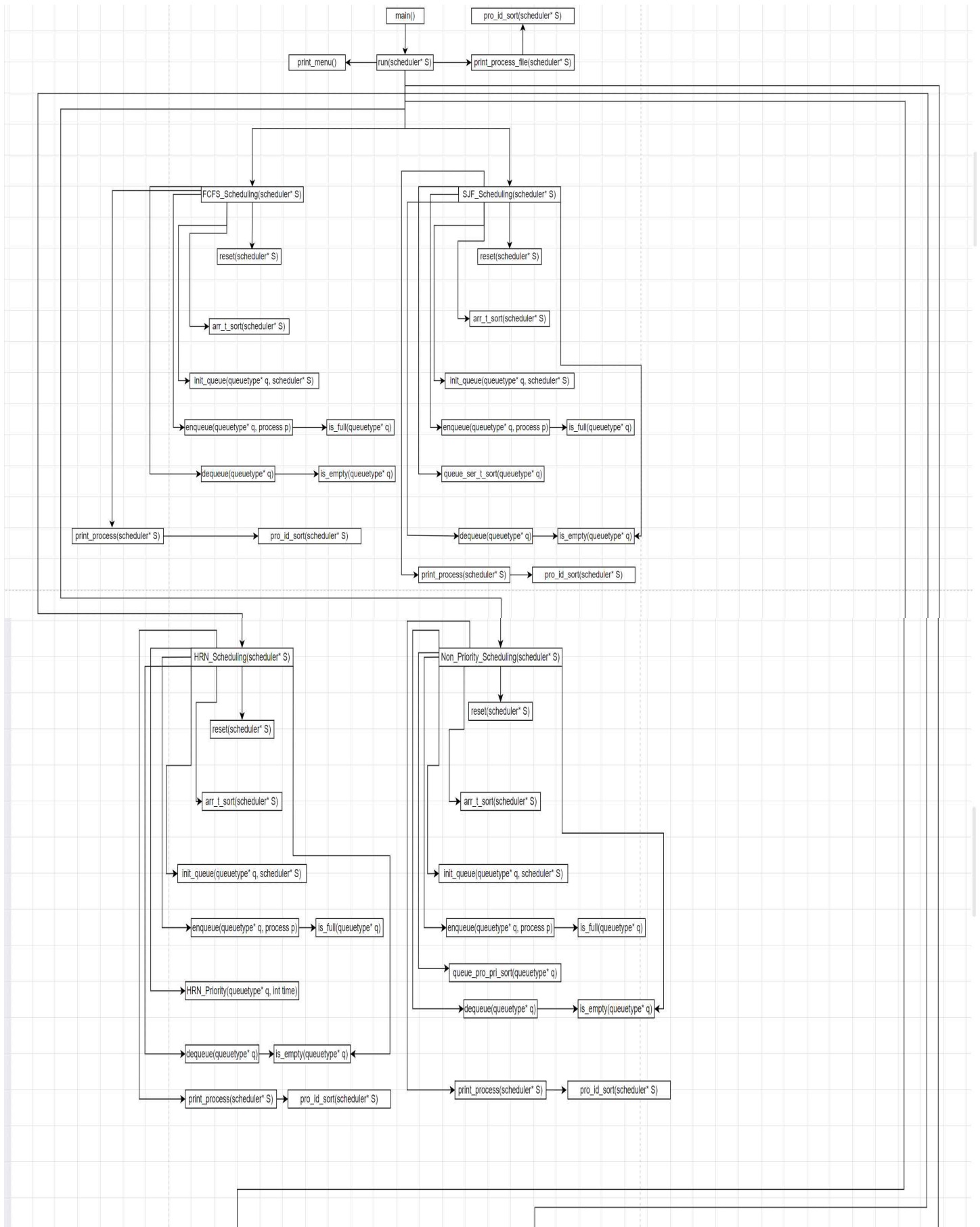
# 목 차

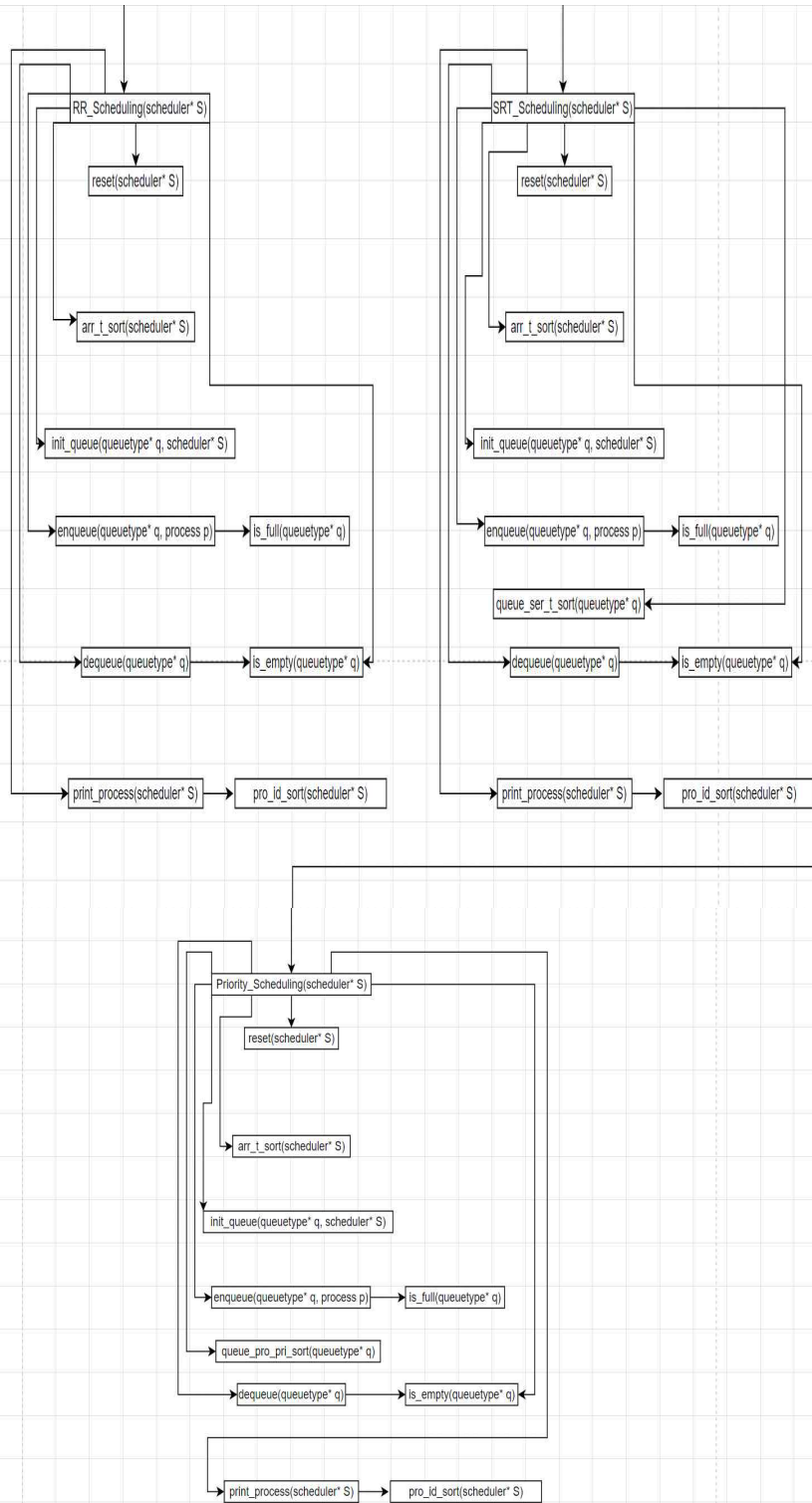
• 함수 구성도	3
• 자료 구조 및 주요 변수	5
• 각 알고리즘별 코드 설명	8
• UI 특징	21
• 실행창	24
• 느낀점	29

# • 함수 구성도

언어는 C언어를 이용하여 개발하였습니다.

아래 사진은 함수 구성도입니다.





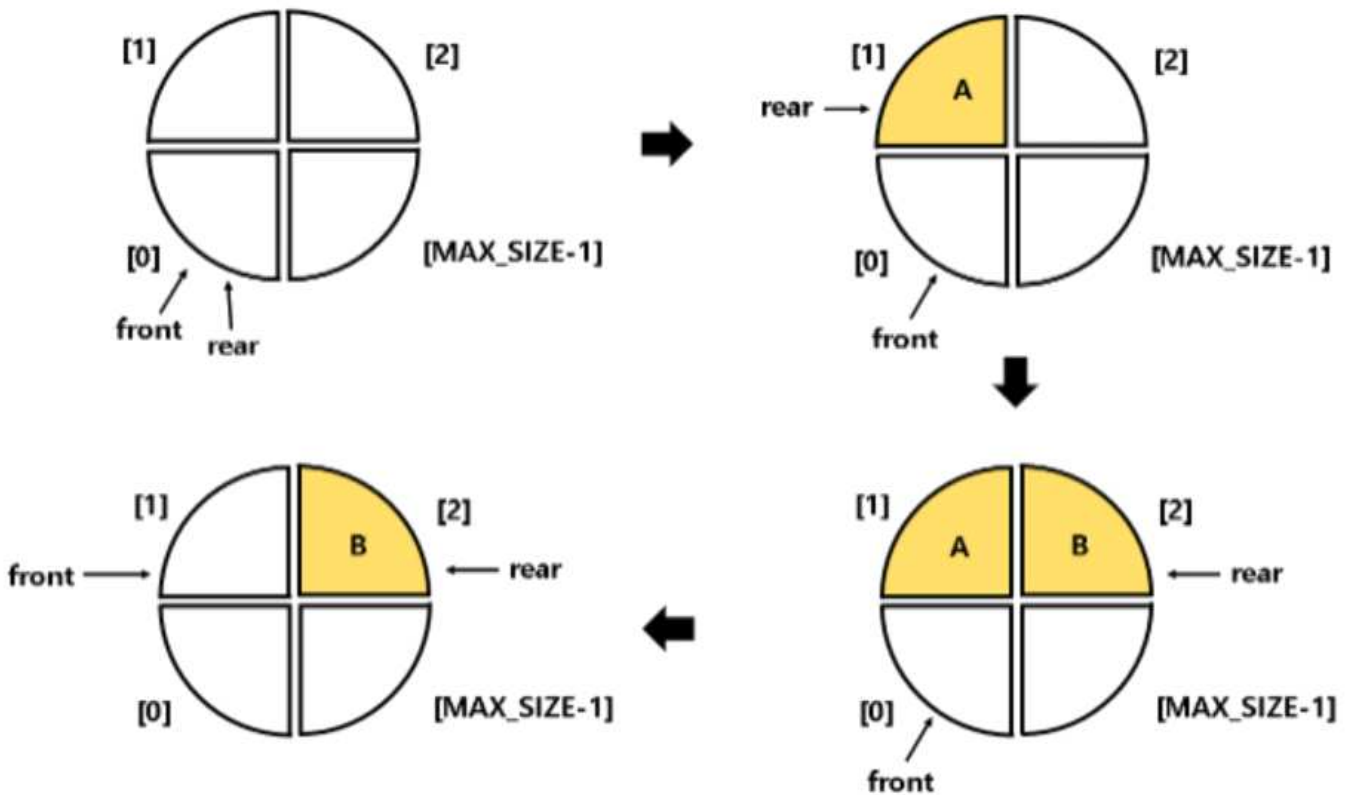
# • 자료 구조 및 주요 변수

-CPU 스케줄링 시뮬레이터 개발에 사용한 자료 구조



## 1. 메모리 동적 할당된 배열

메모리 동적 할당된 배열의 경우는 파일에 프로세스가 몇 개가 있는지를 모르기 때문에 파일에 따라 알맞은 배열을 할당하기 위해 사용하였습니다. 동적 할당된 배열의 경우 먼저 포인터 변수를 선언하고 Heap에 동적으로 만들어놓은 배열의 시작 주소를 저장해주면 포인터 변수를 통해 동적 할당된 메모리에 접근이 가능하며 동적으로 할당받은 메모리도 배열처럼 [index]로 접근이 가능합니다.



## 2. 원형 큐

원형 큐의 경우는 준비 큐를 구현하기 위해 사용하였습니다. 준비 큐의 경우 삽입 삭제가 빈번히 일어나기 때문에 rear이 가리키는 포인터가 배열의 마지막 인덱스를 가리키고 있을 때 앞쪽에서 Dequeue로 발생한 배열의 빈 공간을 활용할 수 없다는 문제점을 가진 선형 큐는 알맞지 않다고 생각하였습니다. 따라서 선형 큐의 문제점을 보완하기 위한 자료 구조인 원형 큐를 사용하였고 원형 큐는 포인터 증가 방식이  $(front+1)\%MAX\_SIZE$ ,  $(rear+1)\%MAX\_SIZE$  형식으로 변환하기 때문에 배열의 첫 인덱스부터 다시 데이터의 삽입이 가능해지도록 하였습니다.

# -CPU 스케줄링 시뮬레이터 개발에 사용한 주요 변수

## 1. process 구조체

```
typedef struct process {  
    int pro_id, arr_t, ser_t, pro_pri, res_t, wait_t, turn_a_t, index, count;  
    double HRN_pro_pri;  
}process;
```

process 구조체는 파일에서 읽어온 프로세스의 정보와 스케줄링 과정에서 나온 정보를 저장하기 위한 구조체입니다. int형 변수로는 pro\_id, arr\_t, ser\_t, pro\_pri, res\_t, wait\_t, turn\_a\_t, index, count를 가지며 double형 변수로는 HRN\_pro\_pri를 가집니다.

pro\_id: 프로세스의 ID를 저장할 변수

arr\_t: 프로세스의 도착 시간을 저장할 변수

ser\_t: 프로세스의 서비스 시간을 저장할 변수

pro\_pri: 프로세스의 우선순위를 저장할 변수

res\_t: 프로세스의 응답 시간을 저장할 변수

wait\_t: 프로세스의 대기 시간을 저장할 변수

turn\_a\_t: 프로세스의 반환 시간을 저장할 변수

index: 현재 프로세스가 저장된 배열의 인덱스를 저장할 변수

count: 현재 프로세스가 수행한 시간을 저장할 변수

HRN\_pro\_pri: HRN에서 계산된 우선순위를 저장할 변수, 실수형으로 나올 수 있으므로 double형으로 선언

## 2. scheduler 구조체

```
typedef struct scheduler {  
    int pro_count, time_slice;  
    double awt, art, att;  
    process* list;  
}scheduler;
```

scheduler 구조체는 현재 스케줄러 정보를 저장하기 위한 구조체입니다. int형 변수로는 pro\_count, time\_slice를 가지며 double형 변수로는 awt, art, att를 가집니다. 또한 process들을 저장하기 위해 process 구조체 배열 list를 가집니다.

pro\_count: 프로세스의 개수를 저장할 변수

time\_slice: 타임슬라이스를 저장할 변수

awt: 평균 대기 시간을 저장할 변수

art: 평균 응답 시간을 저장할 변수

att: 평균 반환 시간을 저장할 변수

list: 프로세스 개수에 따라 동적 할당하여 프로세스 정보를 저장할 구조체 배열

## 3. queue type 구조체

```
typedef struct {  
    int size;  
    int rear;  
    int front;  
    int count;
```

```
process* list;
}queuetype;
```

queuetype 구조체는 준비 큐에 정보를 저장하기 위한 구조체입니다. int형 변수로는 size, rear, front, count를 가지며 큐에 들어온 프로세스의 정보를 저장하기 위한 process 구조체 배열 list를 가집니다.

size: 큐의 크기를 저장할 변수

rear: 큐의 마지막 인덱스를 저장할 변수

front: 큐의 첫 번째 인덱스(원형 큐이므로 큐의 첫 번째 원소 인덱스는 아닙니다.)를 저장할 변수

count: 현재 큐에 있는 프로세스의 개수를 저장할 변수

#### 4. 상수 RT

```
const int RT = 1; //응답시간
```

상수 RT는 응답 시간을 임의로 지정하기 편하게 const를 이용하여 상수로 선언하였습니다.

# • 각 알고리즘별 코드 설명

## 1. FCFS 스케줄링

```
void FCFS_Scheduling(scheduler* S) {
    printf("<FCFS 스케줄링>\n[간트차트]\n");
    reset(S); //스케줄러 정보 초기화(ID, 도착시간, 서비스 시간, 우선순위, 프로세스 개수, 타임슬라이스, 인덱스 제외)
    arr_t_sort(S); //도착 순서로 정렬
    queue_t q; //큐 생성
    process temp; //큐에서 나온 프로세스의 정보를 저장할 변수
    int run_time = 0, wt_sum = 0, tt_sum = 0, rt_sum = 0; //현재 시간, 대기시간 합, 반환시간 합, 응답시간 합
    //을 저장할 변수

    init_queue(&q, S); //큐 초기화
    for (int i = 0; i < S->pro_count; i++) {
        enqueue(&q, S->list[i]); //도착순으로 큐에 삽입
    }

    for (int i = 0; i < S->pro_count; i++) {
        temp = dequeue(&q); //하나씩 디큐
        S->list[temp.index].wait_t = run_time - temp.arr_t; //현재 프로세스의 대기시간 계산 후 변경
        S->list[temp.index].res_t = S->list[temp.index].wait_t + RT; //현재 프로세스의 반응시간 계산 후 변경
        S->list[temp.index].turn_a_t = S->list[temp.index].wait_t + temp.ser_t; //현재 프로세스의 반환시간
        //계산 후 변경

        for (int i = 0; i < temp.ser_t; i++) //현재 프로세스의 서비스 시간만큼 간트차트 출력
            printf("P%d", temp.pro_id);
        printf("[%d] | ", temp.ser_t); //다음 프로세스와 구분하고 서비스 시간을 나타내기 위한 출력문
        run_time += temp.ser_t; //현재시간을 현재 프로세스의 서비스 시간만큼 증가
        wt_sum += S->list[temp.index].wait_t; //프로세스들의 대기시간 합을 구하기 위해 현재 프로세스의 대기
        //시간 더하기
        tt_sum += S->list[temp.index].turn_a_t; //프로세스들의 반환시간 합을 구하기 위해 현재 프로세스의 반
        //환시간 더하기
        rt_sum += S->list[temp.index].res_t; //프로세스들의 응답시간 합을 구하기 위해 현재 프로세스의 응답
        //시간 더하기
    }
    S->awt = (double)wt_sum / S->pro_count; //모든 프로세스가 끝났으므로 평균을 구해서 스케줄러에 저장
    S->att = (double)tt_sum / S->pro_count; //모든 프로세스가 끝났으므로 평균을 구해서 스케줄러에 저장
    S->art = (double)rt_sum / S->pro_count; //모든 프로세스가 끝났으므로 평균을 구해서 스케줄러에 저장
    print_scheduler(S); //스케줄러 정보 출력
    free(q.list); //q에서 프로세스 정보 저장을 위해 동적할당 한 배열의 메모리 해제
}
```

FCFS 스케줄링은 파일에서 불러온 프로세스를 도착 시간순으로 정렬하여 큐에 넣었다가 차례로 꺼내서 대기시간과 응답시간, 반환시간을 구하였습니다. 모든 프로세스가 나오면 평균 대기시간과 평균 응답시간, 평균 반환시간을 구한 후 출력하였고 마지막으로 큐의 list 배열의 메모리를 해제해주었습니다. 도착시간이 같은 경우는 프로세스 ID 순으로 정렬하였습니다.



## 2. SJF 스케줄링

```
void SJF_Scheduling(scheduler* S) {
    printf("<SJF 스케줄링>\n[간트차트]\n");
    reset(S); //프로세스 ID, 도착시간, 서비스시간, 우선순위를 제외한 값들을 초기화
    arr_t_sort(S); //도착 시간 기준으로 정렬
    queue_t q;
    process temp;
    int wt_sum = 0, tt_sum = 0, rt_sum = 0;
    int* check = (int*)malloc(sizeof(int) * S->pro_count); //준비큐에 있거나 있었던 프로세스의 아이디를 저장할
배열
    init_queue(&q, S); //큐 초기화
    int end_pro = 0, cur_time=0; // end_pro 종료된 프로세스의 수를 저장할 변수, cur_time 현재 시간을 가리키는
변수로 0으로 초기화
    int dup = 0; // 중복된 프로세스가 큐에 있는 경우 1 없는 경우 0을 저장할 변수
    int check_count = 0; //큐에 있거나 있었던 프로세스의 총 개수를 저장할 변수
    int ganttChart = 0; //프로세스의 서비스 시간을 간트차트에 출력하기 위한 변수

    while (S->pro_count != end_pro) { //스케줄러에 있는 프로세스의 개수와 종료된 프로세스의 개수가 같지 않을
때까지 반복
        if (check_count != S->pro_count) { //큐에 들어온 적이 없는 프로세스가 존재하는 경우
            for (int i = 0; i < S->pro_count; i++) { //스케줄러에 있는 프로세스의 개수만큼 반복
                if (S->list[i].arr_t <= cur_time) { //스케줄러에 있는 프로세스 중 선택된 프로세스의
도착시간이 현재 시간보다 작거나 같은 경우
                    for (int j = 0; j < S->pro_count; j++) { //이미 큐에 들어왔던 프로세스인지
검사하기 위한 반복문
                        if (S->list[i].pro_id == check[j]) dup = 1; //id가 같은 프로세스가
있다면 dup를 1로 설정
                        if (dup == 0) { //dup가 0인 경우는 현재 프로세스가 큐에 들어온 적 없는 프
로세스인 경우
                            enqueue(&q, S->list[i]); //큐에 삽입
                            check_count++;
                            check[S->list[i].index] = S->list[i].pro_id; //현재 프로세스의 아이
디를 저장
                            if (q.count > 1) queue_ser_t_sort(&q); //큐에 2개 이상 있을 경우
SJF이므로 서비스 시간순으로 정렬
                        }
                    }
                    else { //현재 프로세스가 큐에 들어왔었던 프로세스인 경우
                        dup = 0;
                    }
                }
            }
            if (cur_time == 0) temp = dequeue(&q); //현재 시간이 0일 때 첫번째 프로세스를 디큐
            else if (temp.ser_t == 0) { //현재 프로세스가 종료된 상태로 스케줄러에 해당 프로세스의 정보를 계산해
서 저장
                printf("[%d] | ", ganttChart); //간트차트에 출력된 서비스 시간 출력
                ganttChart = 0; //끝났으므로 0으로 초기화
                end_pro++;
                S->list[temp.index].wait_t = cur_time - S->list[temp.index].ser_t - temp.arr_t;
                S->list[temp.index].res_t = S->list[temp.index].wait_t + RT;
                S->list[temp.index].turn_a_t = S->list[temp.index].wait_t + S->list[temp.index].ser_t;
                wt_sum += S->list[temp.index].wait_t;
            }
        }
        cur_time++;
    }
}
```

```

        tt_sum += S->list[temp.index].turn_a_t;
        rt_sum += S->list[temp.index].res_t;
        if (!is_empty(&q))//큐가 비어있지 않은 경우 디큐
            temp = dequeue(&q);
    }
    if (S->pro_count != end_pro) {//모든 프로세스가 종료된 경우에도 출력되는 것을 방지하기 위한 조건문
        printf("P%d", temp.pro_id);//현재 프로세스의 간트차트 출력
        ganttChart++;//간트차트에 출력된 서비스 시간 증가
    }
    cur_time++;//현재 시간 증가
    temp.ser_t--;//현재 프로세스의 서비스 시간 감소
}
S->awt = (double)wt_sum / S->pro_count;
S->att = (double)tt_sum / S->pro_count;
S->art = (double)rt_sum / S->pro_count;
print_scheduler(S);
free(q.list);//동적할당된 배열 메모리 반환
free(check);//프로세스 아이디를 저장하기 위해 동적할당된 배열의 메모리 반환
}

```

SJF 스케줄링은 현재시간을 나타내는 변수 cur\_time를 사용하여 cur\_time가 증가할 때마다 프로세스의 도착시간을 비교하여 도착하였으면 큐에 삽입하고 큐에 2개 이상의 프로세스가 있는 경우 서비스 시간순으로 큐를 정렬해주었습니다. 이때 서비스 시간이 같은 경우는 프로세스 ID 순으로 다시 정렬하였습니다. 그리고 현재 프로세스의 서비스 시간이 0이 되면 프로세스가 종료된 경우로 큐에서 새로운 프로세스를 꺼내 모든 프로세스가 종료될 때까지 반복하였습니다. 하나의 프로세스가 종료되면 해당 프로세스의 대기시간, 응답시간, 반환시간을 저장하고 모든 프로세스가 종료되면 평균값을 저장하였습니다. 마지막으로 큐의 list배열과 check 배열의 메모리를 반환해주었습니다.

### 3. HRN 스케줄링

```
void HRN_Scheduling(scheduler* S) {
    printf("<HRN 스케줄링>\n[간트차트]\n");
    reset(S); //스케줄러 정보 초기화(ID, 도착시간, 서비스 시간, 우선순위, 프로세스 개수, 타임슬라이스, 인덱스 제외)
    arr_t_sort(S); //도착 순서로 정렬
    queue_t q; //큐 생성
    process temp; //큐에서 나온 프로세스의 정보를 저장할 변수
    int wt_sum = 0, tt_sum = 0, rt_sum = 0, end_pro = 0, cur_time = 0, dup = 0, check_count = 0,
    ganttChart = 0; //위 함수에 있는 변수와 같은 역할
    int* check = (int*)malloc(sizeof(int) * S->pro_count); //준비큐에 있거나 있었던 프로세스의 아이디를 저장할 배열
    init_queue(&q, S); //큐 초기화

    while (S->pro_count != end_pro) { //스케줄러에 있는 프로세스의 개수와 종료된 프로세스의 개수가 같지 않을 때까지 반복
        if (check_count != S->pro_count) { //큐에 들어온 적이 없는 프로세스가 존재하는 경우
            for (int i = 0; i < S->pro_count; i++) { //스케줄러에 있는 프로세스의 개수만큼 반복
                if (S->list[i].arr_t <= cur_time) { //스케줄러에 있는 프로세스 중 선택된 프로세스의 도착시간이 현재 시간보다 작거나 같은 경우
                    for (int j = 0; j < S->pro_count; j++) { //이미 큐에 들어왔던 프로세스인지
                        if (S->list[i].pro_id == check[j]) dup = 1; //id가 같은 프로세스가 있다면 dup를 1로 설정
                    }
                    if (dup == 0) { //dup가 0인 경우는 현재 프로세스가 큐에 들어온 적 없는 프로세스인 경우
                        enqueue(&q, S->list[i]); //큐에 삽입
                        if (q.count > 1) { //큐에 2개 이상의 프로세스가 있다면 정렬
                            HRN_Priority(&q, cur_time); //HRN_Priority 함수를 이용하여 정렬
                        }
                        check_count++;
                        check[S->list[i].index] = S->list[i].pro_id; //현재 프로세스의 아이디를 저장
                    }
                    else { //현재 프로세스가 큐에 들어왔었던 프로세스인 경우
                        dup = 0;
                    }
                }
            }
        }

        if (cur_time == 0) temp = dequeue(&q); //현재 시간이 0일 때 첫번째 프로세스를 디큐
        else if (temp.ser_t == 0) { //현재 프로세스가 종료된 상태로 스케줄러에 해당 프로세스의 정보를 계산해서 저장
            printf("[%d] | ", ganttChart); //간트차트에 출력된 서비스 시간 출력
            ganttChart = 0; //끝났으므로 0으로 초기화
            end_pro++;
            S->list[temp.index].wait_t = cur_time - S->list[temp.index].ser_t - temp.arr_t;
            S->list[temp.index].res_t = S->list[temp.index].wait_t + RT;
            S->list[temp.index].turn_a_t = S->list[temp.index].wait_t + S->list[temp.index].ser_t;
            wt_sum += S->list[temp.index].wait_t;
            tt_sum += S->list[temp.index].turn_a_t;
            rt_sum += S->list[temp.index].res_t;
            if (q.count > 1) { //큐에 2개 이상의 프로세스가 있다면 정렬
                HRN_Priority(&q, cur_time);
            }
        }
        cur_time++;
    }
}
```

```

        HRN_Priority(&q, cur_time); //HRN_Priority 함수를 이용하여 정렬
        if (!is_empty(&q)) //큐가 공백이 아닌 경우
            temp = dequeue(&q); //디큐
    }
    if (S->pro_count != end_pro) { //모든 프로세스가 종료된 경우에도 출력되는 것을 방지하기 위한 조건문
        ganttChart++; //간트차트에 출력된 서비스 시간 증가
        printf("P%d", temp.pro_id); //현재 프로세스의 간트차트 출력
    }
    cur_time++; //현재 시간 증가
    temp.ser_t--; //현재 프로세스의 서비스 시간 감소
}
S->awt = (double)wt_sum / S->pro_count;
S->att = (double)tt_sum / S->pro_count;
S->art = (double)rt_sum / S->pro_count;
print_scheduler(S);
free(q.list); //메모리 반환
free(check); //메모리 반환
}

```

HRN 스케줄링은 SJF 스케줄링의 알고리즘 틀과 비슷하지만 HRN\_Priority 함수를 사용한다는 차이가 있습니다. HRN\_Priority 함수는 현재 큐에 있는 프로세스들을 기준으로 HRN에서 사용하는 우선순위를 계산하고 그 결과값에 따라 정렬해주는 함수입니다. 이 함수 또한 우선순위가 같은 경우는 프로세스 ID를 기준으로 다시 정렬해주었습니다.

#### 4. 비선점형 우선순위 스케줄링

```
void Non_Priority_Scheduling(scheduler* S) {
    printf("<비선점형 우선순위 스케줄링>\n[간트차트]\n");
    reset(S); //스케줄러 정보 초기화(ID, 도착시간, 서비스 시간, 우선순위, 프로세스 개수, 타임슬라이스, 인덱스 제외)
    arr_t_sort(S); //도착 순서로 정렬
    queue_t q; //큐 생성
    process temp; //큐에서 나온 프로세스의 정보를 저장할 변수
    int wt_sum = 0, tt_sum = 0, rt_sum = 0, cur_time = 0, check_count = 0, dup = 0, end_pro = 0,
    ganttChart = 0; //위 함수에 있는 변수와 같은 역할
    int* check = (int*)malloc(sizeof(int) * S->pro_count); //준비큐에 있거나 있었던 프로세스의 아이디를 저장할
    배열
    init_queue(&q, S); //큐 초기화

    while (S->pro_count != end_pro) { //스케줄러에 있는 프로세스의 개수와 종료된 프로세스의 개수가 같지 않을
    때까지 반복
        if (check_count != S->pro_count) { //큐에 들어온 적이 없는 프로세스가 존재하는 경우
            for (int i = 0; i < S->pro_count; i++) { //스케줄러에 있는 프로세스의 개수만큼 반복
                if (S->list[i].arr_t <= cur_time) { //스케줄러에 있는 프로세스 중 선택된 프로세스의
                도착시간이 현재 시간보다 작거나 같은 경우
                    for (int j = 0; j < S->pro_count; j++) { //이미 큐에 들어왔던 프로세스인지
                    검사하기 위한 반복문
                        if (S->list[i].pro_id == check[j]) dup = 1; //id가 같은 프로세스가
                        있다면 dup를 1로 설정
                    if (dup == 0) { //dup가 0인 경우는 현재 프로세스가 큐에 들어온 적 없는 프
                    로세스인 경우
                        enqueue(&q, S->list[i]); //큐에 삽입
                        check_count++;
                        check[S->list[i].index] = S->list[i].pro_id; // 현재 프로세스의 아
                        이드를 저장
                        if (q.count > 1) queue_pro_pri_sort(&q); //큐에 2개 이상의 프로
                        세스가 있다면 우선순위를 기준으로 정렬
                    }
                    else { //현재 프로세스가 큐에 들어왔었던 프로세스인 경우
                        dup = 0;
                    }
                }
            }
        }
        if (cur_time == 0) temp = dequeue(&q); //현재 시간이 0일 때 첫번째 프로세스를 디큐
        else if (temp.ser_t == 0) { //현재 프로세스가 종료된 상태로 스케줄러에 해당 프로세스의 정보를 계산해
        서 저장
            printf("[%d] | ", ganttChart); //간트차트에 출력된 서비스 시간 출력
            ganttChart = 0; //끝났으므로 0으로 초기화
            end_pro++;
            S->list[temp.index].wait_t = cur_time - S->list[temp.index].ser_t - temp.arr_t;
            S->list[temp.index].res_t = S->list[temp.index].wait_t + RT; //응답 시간 1로 설정
            S->list[temp.index].turn_a_t = S->list[temp.index].wait_t + S->list[temp.index].ser_t;
            wt_sum += S->list[temp.index].wait_t;
            tt_sum += S->list[temp.index].turn_a_t;
            rt_sum += S->list[temp.index].res_t;
            if (!is_empty(&q)) { //큐가 공백이 아닌 경우
                temp = dequeue(&q); //디큐
            }
        }
    }
}
```

```

    }
    if (S->pro_count != end_pro) { // 모든 프로세스가 종료된 경우에도 출력되는 것을 방지하기 위한 조건문
        printf("P%d", temp.pro_id); // 현재 프로세스의 간트차트 출력
        ganttChart++; // 간트차트에 출력된 서비스 시간 증가
    }
    cur_time++; // 현재 시간 증가
    temp.ser_t--; // 현재 프로세스의 서비스 시간 감소
}
S->awt = (double)wt_sum / S->pro_count;
S->att = (double)tt_sum / S->pro_count;
S->art = (double)rt_sum / S->pro_count;
print_scheduler(S); // 스케줄러 정보 출력
free(q.list); // 메모리 반환
free(check); // 메모리 반환
}

```

비선점형 스케줄링은 현재시간을 나타내는 변수 cur\_time를 사용하여 cur\_time가 증가할 때마다 프로세스의 도착시간을 비교하여 도착하였으면 큐에 삽입하고 큐에 2개 이상의 프로세스가 있는 경우 우선순위 순으로 큐를 정렬해주었습니다. 이때 서비스 시간이 같은 경우는 프로세스 ID 순으로 다시 정렬하였습니다. 그리고 현재 프로세스의 서비스 시간이 0이 되면 프로세스가 종료된 경우로 큐에서 새로운 프로세스를 꺼내 모든 프로세스가 종료될 때까지 반복하였습니다. 하나의 프로세스가 종료되면 해당 프로세스의 대기시간, 응답시간, 반환시간을 저장하고 모든 프로세스가 종료되면 평균값을 저장하였습니다. 마지막으로 큐의 list배열과 check 배열의 메모리를 반환해주었습니다.

## 5. 라운드로빈 스케줄링

```
void RR_Scheduling(scheduler* S) {
    printf("<Round-Robin 스케줄링>\n[간트차트]\n");
    reset(S); //스케줄러 정보 초기화(ID, 도착시간, 서비스 시간, 우선순위, 프로세스 개수, 타임슬라이스, 인덱스 제외)
    arr_t_sort(S); //도착 순서로 정렬
    queue_t q; //큐 생성
    process temp; //큐에서 나온 프로세스의 정보를 저장할 변수
    int wt_sum = 0, tt_sum = 0, rt_sum = 0, cur_time = 0, time_slice = 0, check_count = 0, dup = 0,
    end_pro = 0, ganttChart = 0;
    //time_slice는 주어진 타임슬라이스 시간과 비교하기 위해 현재 프로세스가 사용한 시간을 저장할 변수, 나머지 변수는 위 함수에 있는 변수와 동일한 기능
    int* check = (int*)malloc(sizeof(int) * S->pro_count); //준비큐에 있거나 있었던 프로세스의 아이디를 저장할 배열
    init_queue(&q, S); //큐 초기화

    while (S->pro_count != end_pro) { //스케줄러에 있는 프로세스의 개수와 종료된 프로세스의 개수가 같지 않을 때까지 반복
        if (check_count != S->pro_count) { //큐에 들어온 적이 없는 프로세스가 존재하는 경우
            for (int i = 0; i < S->pro_count; i++) { //스케줄러에 있는 프로세스의 개수만큼 반복
                if (S->list[i].arr_t <= cur_time) { //스케줄러에 있는 프로세스 중 선택된 프로세스의 도착시간이 현재 시간보다 작거나 같은 경우
                    for (int j = 0; j < S->pro_count; j++) { //이미 큐에 들어왔던 프로세스인지 검사하기 위한 반복문
                        if (S->list[i].pro_id == check[j]) dup = 1; //id가 같은 프로세스가 있다면 dup를 1로 설정
                    }
                    if (dup == 0) { //dup가 0인 경우는 현재 프로세스가 큐에 들어온 적 없는 프로세스인 경우
                        enqueue(&q, S->list[i]); //큐에 해당 프로세스 삽입
                        check_count++;
                        check[S->list[i].index] = S->list[i].pro_id; //현재 프로세스의 아이디를 저장
                    }
                }
                else { //현재 프로세스가 큐에 들어왔었던 프로세스인 경우
                    dup = 0;
                }
            }
        }
        if (cur_time == 0) temp = dequeue(&q); //현재 시간이 0일 때 첫번째 프로세스를 디큐

        if (time_slice == S->time_slice && temp.ser_t != 0) { //현재 프로세스가 주어진 타임슬라이스를 다 썼지만 작업이 끝나지 않은 상태
            printf("[%d] | ", ganttChart); //간트차트에 출력된 서비스 시간 출력
            ganttChart = 0; //끝났으므로 0으로 초기화
            time_slice = 0; //다음 프로세스의 시간을 저장하기 위해 0으로 초기화
            enqueue(&q, temp); //작업이 남았으므로 다시 큐에 삽입
            temp = dequeue(&q); //디큐
        }
        else if (temp.ser_t == 0) { //현재 프로세스가 종료된 상태로 스케줄러에 해당 프로세스의 정보를 계산해서 저장
            printf("[%d] | ", ganttChart); //간트차트에 출력된 서비스 시간 출력
            ganttChart = 0; //끝났으므로 0으로 초기화
        }
    }
}
```

```

        end_pro++;
        time_slice = 0; //다음 프로세스의 시간을 저장하기 위해 0으로 초기화
        S->list[temp.index].wait_t = cur_time - S->list[temp.index].ser_t - temp.arr_t;
        S->list[temp.index].turn_a_t = S->list[temp.index].wait_t + S->list[temp.index].ser_t;
        wt_sum += S->list[temp.index].wait_t;
        tt_sum += S->list[temp.index].turn_a_t;
        rt_sum += S->list[temp.index].res_t;
        if (!is_empty(&q))//큐가 공백이 아닌 경우
            temp = dequeue(&q);//디큐
    }
    if (S->pro_count != end_pro) {//모든 프로세스가 종료된 경우에도 출력되는 것을 방지하기 위한 조건문
        printf("P%d", temp.pro_id);//현재 프로세스의 간트차트 출력
        ganttChart++;//간트차트에 출력된 서비스 시간 증가
    }
    temp.count++;//해당 프로세스의 현재 수행시간 증가
    cur_time++;//현재 시간 증가
    if (temp.count == RT)S->list[temp.index].res_t = cur_time - temp.arr_t;//해당 프로세스의 현재 수
    행시간이 주어진 반응시간과 같으므로 반응시간 입력
    time_slice++;//해당 프로세스의 사용시간 증가
    temp.ser_t--;//해당 프로세스의 서비스 시간 감소

}
S->awt = (double)wt_sum / S->pro_count;
S->att = (double)tt_sum / S->pro_count;
S->art = (double)rt_sum / S->pro_count;
print_scheduler(S);//스케줄러의 정보 출력
free(q.list);//메모리 반환
free(check);//메모리 반환
}

```

라운드로빈 스케줄링은 현재시간을 나타내는 변수 cur\_time를 사용하여 cur\_time가 증가할 때마다 프로세스의 도착시간을 비교하여 도착하였으면 큐에 삽입하고 큐에 2개 이상의 프로세스가 있는 경우 도착시간 순으로 큐를 정렬해주었습니다. 이때 도착시간이 같은 경우는 프로세스 ID 순으로 다시 정렬하였습니다. 그리고 현재 타임슬라이스가 주어진 타임슬라이스와 같고 서비스 시간이 남은 경우가 추가 되었습니다. 이 경우는 다시 큐에 삽입하고 큐 제일 앞에 있는 프로세스를 꺼내서 같은 방식으로 진행하게 됩니다. 현재 프로세스의 서비스 시간이 0이 되면 프로세스가 종료된 경우로 큐에서 새로운 프로세스를 꺼내 모든 프로세스가 종료될 때까지 반복하였습니다. 하나의 프로세스가 종료되면 해당 프로세스의 대기시간, 반환시간을 저장하고 모든 프로세스가 종료되면 평균값을 저장하였습니다. 마지막으로 큐의 list배열과 check 배열의 메모리를 반환해주었습니다. 또한 temp.count를 사용하여 현재 프로세스가 동작 시간이 반응시간과 같은 경우 반응시간을 계산하여 저장해주었습니다.



## 6. SRT 스케줄링

```
void SRT_Scheduling(scheduler* S) {
    printf("<SRT스케줄링>\n[간트차트]\n");
    reset(S); //스케줄러 정보 초기화(ID, 도착시간, 서비스 시간, 우선순위, 프로세스 개수, 타임슬라이스, 인덱스 제외)
    arr_t_sort(S); //도착 순서로 정렬
    queue_t q; //큐 생성
    process temp; //큐에서 나온 프로세스의 정보를 저장할 변수
    int wt_sum = 0, tt_sum = 0, rt_sum = 0, cur_time = 0, end_pro = 0, dup = 0, check_count = 0,
    time_slice = 0, ganttChart = 0; //라운드로빈 함수의 변수와 같은 역할
    int* check = (int*)malloc(sizeof(int) * S->pro_count); //준비큐에 있거나 있었던 프로세스의 아이디를 저장할
    배열
    init_queue(&q, S); //큐 초기화

    while (S->pro_count != end_pro) { //스케줄러에 있는 프로세스의 개수와 종료된 프로세스의 개수가 같지 않을
    때까지 반복
        if (check_count != S->pro_count) { //큐에 들어온 적이 없는 프로세스가 존재하는 경우
            for (int i = 0; i < S->pro_count; i++) { //스케줄러에 있는 프로세스의 개수만큼 반복
                if (S->list[i].arr_t <= cur_time) { //스케줄러에 있는 프로세스 중 선택된 프로세스의
                도착시간이 현재 시간보다 작거나 같은 경우
                    for (int j = 0; j < S->pro_count; j++) { //이미 큐에 들어왔던 프로세스인지
                    검사하기 위한 반복문
                        if (S->list[i].pro_id == check[j]) dup = 1; //id가 같은 프로세스가
                        있다면 dup를 1로 설정
                    if (dup == 0) { //dup가 0인 경우는 현재 프로세스가 큐에 들어온 적 없는 프
                    로세스인 경우
                        enqueue(&q, S->list[i]); //큐에 해당 프로세스 삽입
                        if (q.count > 1) { //큐에 2개 이상의 프로세스가 있는 경우 서비스 시
                        간 기준으로 정렬
                            queue_ser_t_sort(&q);
                            check_count++;
                            check[S->list[i].index] = S->list[i].pro_id; //현재 프로세스의 아이
                            디를 저장
                        }
                        else { //현재 프로세스가 큐에 들어왔었던 프로세스인 경우
                            dup = 0;
                        }
                    }
                }
            }
            if (cur_time == 0) { //현재 시간이 0일 때 첫번째 프로세스를 디큐
                temp = dequeue(&q);

                if (time_slice == S->time_slice && temp.ser_t != 0) { //현재 프로세스가 주어진 타임슬라이스를 다 썼
                지만 작업이 끝나지 않은 상태
                    printf("[%d] | ", ganttChart); //간트차트에 출력된 서비스 시간 출력
                    ganttChart = 0; //끝났으므로 0으로 초기화
                    time_slice = 0; //다음 프로세스의 시간을 저장하기 위해 0으로 초기화
                    enqueue(&q, temp); //작업이 남았으므로 다시 큐에 삽입
                    if (q.count > 1) { //큐에 2개 이상의 프로세스가 있는 경우 서비스 시간 기준으로 정렬
                        queue_ser_t_sort(&q);
                        temp = dequeue(&q);
                    }
                }
            }
        }
    }
```

```

else if (temp.ser_t == 0) { //현재 프로세스가 종료된 상태로 스케줄러에 해당 프로세스의 정보를 계산해
서 저장

    printf("[%d] | ", ganttChart); //간트차트에 출력된 서비스 시간 출력
    ganttChart = 0; //끝났으므로 0으로 초기화
    end_pro++;
    time_slice = 0; //다음 프로세스의 시간을 저장하기 위해 0으로 초기화
    S->list[temp.index].wait_t = cur_time - S->list[temp.index].ser_t - temp.arr_t;
    S->list[temp.index].turn_a_t = S->list[temp.index].wait_t + S->list[temp.index].ser_t;
    wt_sum += S->list[temp.index].wait_t;
    tt_sum += S->list[temp.index].turn_a_t;
    rt_sum += S->list[temp.index].res_t;
    if (!is_empty(&q)) //큐가 공백이 아닌 경우 디큐
        temp = dequeue(&q);
    }
    if (S->pro_count != end_pro) { //모든 프로세스가 종료된 경우에도 출력되는 것을 방지하기 위한 조건문
        printf("P%d", temp.pro_id); //현재 프로세스의 간트차트 출력
        ganttChart++; //간트차트에 출력된 서비스 시간 증가
    }
    temp.count++; //해당 프로세스의 현재 수행시간 증가
    cur_time++; //현재 시간 증가
    if (temp.count == RT) S->list[temp.index].res_t = cur_time - temp.arr_t; //해당 프로세스의 현재 수
행시간이 주어진 반응시간과 같으므로 반응시간 입력
    time_slice++; //해당 프로세스의 사용시간 증가
    temp.ser_t--; //해당 프로세스의 서비스 시간 감소
}
S->awt = (double)wt_sum / S->pro_count;
S->att = (double)tt_sum / S->pro_count;
S->art = (double)rt_sum / S->pro_count;
print_scheduler(S); //스케줄러의 정보 출력
free(q.list); //메모리 반환
free(check); //메모리 반환
}

```

SRT 스케줄링은 라운드로빈 스케줄링과 알고리즘이 똑같지만 큐에 있는 프로세스를 서비스 시간을 기준으로 정렬한다는 점만 다릅니다. SRT 스케줄링 또한 서비스 시간이 같은 경우에는 ID 순으로 다시 정렬해주었습니다.

## 7. 선점형 우선순위 스케줄링

```
void Priority_Scheduling(scheduler* S) {
    printf("<선점형 우선순위 스케줄링>\n[간트차트]\n");
    reset(S); //스케줄러 정보 초기화(ID, 도착시간, 서비스 시간, 우선순위, 프로세스 개수, 타임슬라이스, 인덱스 제외)
    arr_t_sort(S); //도착 순서로 정렬
    queue_t q; //큐 생성
    process temp; //큐에서 나온 프로세스의 정보를 저장할 변수
    int wt_sum = 0, tt_sum = 0, rt_sum = 0, cur_time = 0, end_pro = 0, dup = 0, check_count = 0,
    ganttChart = 0; //라운드로빈의 변수들과 같은 역할
    int* check = (int*)malloc(sizeof(int) * S->pro_count); //준비큐에 있거나 있었던 프로세스의 아이디를 저장할
    배열
    init_queue(&q, S); //큐 초기화

    while (S->pro_count != end_pro) { //스케줄러에 있는 프로세스의 개수와 종료된 프로세스의 개수가 같지 않을
    때까지 반복
        if (check_count != S->pro_count) { //큐에 들어온 적이 없는 프로세스가 존재하는 경우
            for (int i = 0; i < S->pro_count; i++) { //스케줄러에 있는 프로세스의 개수만큼 반복
                if (S->list[i].arr_t <= cur_time) { //스케줄러에 있는 프로세스 중 선택된 프로세스의
                도착시간이 현재 시간보다 작거나 같은 경우
                    for (int j = 0; j < S->pro_count; j++) { //이미 큐에 들어왔던 프로세스인지
                    검사하기 위한 반복문
                        if (S->list[i].pro_id == check[j]) dup = 1; //id가 같은 프로세스가
                        있다면 dup를 1로 설정
                    if (dup == 0) { //dup가 0인 경우는 현재 프로세스가 큐에 들어온 적 없는 프
                    로세스인 경우
                        enqueue(&q, S->list[i]); //큐에 해당 프로세스 삽입
                        if (q.count > 1) { //큐에 프로세스가 2개 이상인 경우 우선순위 기준
                        으로 정렬
                            queue_pro_pri_sort(&q);
                            check_count++;
                            check[S->list[i].index] = S->list[i].pro_id; //현재 프로세스의 아이
                            디를 저장
                        }
                        else { //현재 프로세스가 큐에 들어왔었던 프로세스인 경우
                            dup = 0;
                        }
                    }
                }
            }
            if (cur_time == 0) { //현재 시간이 0일 때 첫번째 프로세스를 디큐
                temp = dequeue(&q);

                if (q.count != 0 && temp.ser_t != 0 && temp.pro_pri > q.list[q.front + 1].pro_pri) { //현재 프로세스
                의 작업 시간이 남아있으며 큐에 프로세스가 1개 이상 있고 제일 앞에 있는 프로세스의 우선 순위가 현재 프로세스의 우선순
                위보다 높은 경우
                    printf("[%d] | ", ganttChart); //간트차트에 출력된 서비스 시간 출력
                    ganttChart = 0; //끝났으므로 0으로 초기화
                    enqueue(&q, temp); //작업이 남아있으므로 다시 큐에 삽입
                    if (q.count > 1) { //큐에 프로세스가 2개 이상 있는 경우
                        queue_pro_pri_sort(&q); //우선순위 기준으로 정렬
                        temp = dequeue(&q); //디큐
                    }
                }
            }
        }
    }
```

```

else if (temp.ser_t == 0) { //현재 프로세스가 종료된 상태로 스케줄러에 해당 프로세스의 정보를 계산해
서 저장

    printf("[%d] | ", ganttChart); //간트차트에 출력된 서비스 시간 출력
    ganttChart = 0; //끝났으므로 0으로 초기화
    end_pro++;
    S->list[temp.index].wait_t = cur_time - S->list[temp.index].ser_t - temp.arr_t;
    S->list[temp.index].turn_a_t = S->list[temp.index].wait_t + S->list[temp.index].ser_t;
    wt_sum += S->list[temp.index].wait_t;
    tt_sum += S->list[temp.index].turn_a_t;
    rt_sum += S->list[temp.index].res_t;
    if (lis_empty(&q)) //큐가 공백이 아닌 경우 디큐
        temp = dequeue(&q);
    }
    if (S->pro_count != end_pro) { //모든 프로세스가 종료된 경우에도 출력되는 것을 방지하기 위한 조건문
        printf("P%d", temp.pro_id); //현재 프로세스의 간트차트 출력
        ganttChart++; //간트차트에 출력된 서비스 시간 증가
    }

    temp.count++; //해당 프로세스의 현재 수행시간 증가
    cur_time++; //현재 시간 증가
    if (temp.count == RT) S->list[temp.index].res_t = cur_time - temp.arr_t; //해당 프로세스의 현재 수
행시간이 주어진 반응시간과 같으므로 반응시간 입력
    temp.ser_t--; //해당 프로세스의 서비스 시간 감소
}
S->awt = (double)wt_sum / S->pro_count;
S->att = (double)tt_sum / S->pro_count;
S->art = (double)rt_sum / S->pro_count;
print_scheduler(S); //스케줄러의 정보 출력
free(q.list); //메모리 반환
free(check); //메모리 반환
}

```

선점형 우선순위 스케줄링은 현재시간을 나타내는 변수 cur\_time를 사용하여 cur\_time가 증가할 때마다 프로세스의 도착시간을 비교하여 도착하였으면 큐에 삽입하고 큐에 2개 이상의 프로세스가 있는 경우 우선 순위 순으로 큐를 정렬해주었습니다. 이때 우선순위가 같은 경우는 프로세스 ID 순으로 다시 정렬하였습니다. cur\_time가 증가할 때마다 큐의 첫 번째 프로세스의 우선순위가 현재 프로세스의 우선순위보다 높은 경우는 현재 프로세스를 종료한 후 큐에 넣고 큐에 첫 번째 프로세스를 꺼내 진행하였습니다. 현재 프로세스의 서비스 시간이 0이 되면 프로세스가 종료된 경우로 큐에서 새로운 프로세스를 꺼내 모든 프로세스가 종료될 때까지 반복하였습니다. 하나의 프로세스가 종료되면 해당 프로세스의 대기시간, 반환시간을 저장하고 모든 프로세스가 종료되면 평균값을 저장하였습니다. 마지막으로 큐의 list배열과 check 배열의 메모리를 반환해주었습니다. 또한 temp.count를 사용하여 현재 프로세스가 동작 시간이 반응시간과 같은 경우 반응 시간을 계산하여 저장해주었습니다.

# • UI 특징

저는 CPU 스케줄링 시뮬레이터를 사용자 입력에 따라 알고리즘이 실행되도록 하였습니다. 처음 화면에는 프로세스 정보가 출력된 후 메뉴가 출력됩니다. 그다음 1~7번 중 하나를 선택하면 현재 콘솔창을 지우고 해당하는 알고리즘 결과가 출력되도록 하였습니다. 결과가 출력된 후에는 메뉴가 출력되어 다른 메뉴를 선택할 수 있도록 하였습니다. 8번을 입력하면 메모리를 해제한 후 프로그램이 종료되도록 하였습니다. 또한 1~8이 아닌 숫자를 입력하면 콘솔창이 지워지고 초기화면이 다시 출력되도록 하였습니다. 간트 차트의 경우는 따로 함수가 존재하지 않고 각각의 알고리즘 내부에 출력되도록 하였습니다.

## 1-1. 파일에서 읽어온 프로세스 정보 출력 함수

```
void print_process_file(scheduler* S) {
    pro_id_sort(S);
    printf("\t <프로세스 정보>");
    printf("\n |-----| \n");
    printf(" | ID | 도착시간 | 서비스시간 | 우선순위 | \n");
    printf(" |-----| \n");
    for (int i = 0; i < S->pro_count; i++)
        printf(" | %-2d | %3d | %3d | %3d | \n", S->list[i].pro_id, S->list[i].arr_t,
S->list[i].ser_t, S->list[i].pro_pri);
    printf(" |-----| \n");
    printf("\t [타임 슬라이스 %d]\n", S->time_slice);
}
```

## 1-2. 실행창

텍스트 파일

data


×

+

—

□

×

파일   편집   보기   

5  
P1 0 10 3  
P2 1 28 2  
P3 2 6 4  
P4 3 4 1  
P5 4 14 2  
2

줄 6, 열 10

100%

Windows (CRLF)

UTF-8

실행창

<프로세스 정보>

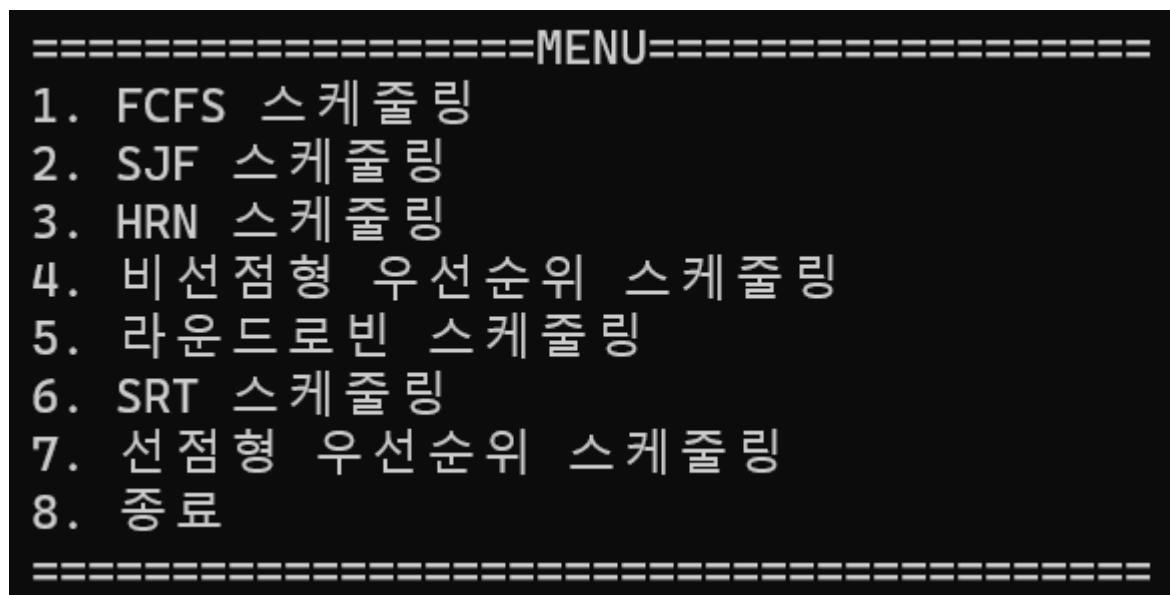
ID	도착시간	서비스시간	우선순위
P1	0	10	3
P2	1	28	2
P3	2	6	4
P4	3	4	1
P5	4	14	2

[타임 슬라이스 2]

## 2-1. 메뉴 출력 함수

```
void print_menu() {  
    printf("=====MENU=====\\n");  
    printf("1. FCFS 스케줄링\\n");  
    printf("2. SJF 스케줄링\\n");  
    printf("3. HRN 스케줄링\\n");  
    printf("4. 비선점형 우선순위 스케줄링\\n");  
    printf("5. 라운드로빈 스케줄링\\n");  
    printf("6. SRT 스케줄링\\n");  
    printf("7. 선점형 우선순위 스케줄링\\n");  
    printf("8. 종료\\n");  
    printf("=====\\n");  
}
```

## 2-1. 실행창



```
=====MENU=====  
1. FCFS 스케줄링  
2. SJF 스케줄링  
3. HRN 스케줄링  
4. 비선점형 우선순위 스케줄링  
5. 라운드로빈 스케줄링  
6. SRT 스케줄링  
7. 선점형 우선순위 스케줄링  
8. 종료  
=====
```

### 3-1. 스케줄러 정보 출력 함수

```
void print_scheduler(scheduler* S) {
    //id순으로 출력하기 위해 pro_id_sort함수 호출
    pro_id_sort(S); //스케줄러에 있는 프로세스를 id순으로 정렬
    printf("\n |-----| \n");
    printf(" | ID |   대기시간   | \n");
    printf(" |-----| \n");
    for (int i = 0; i < S->pro_count; i++)
        printf(" | P%-2d |      %3d      | \n", S->list[i].pro_id, S->list[i].wait_t);
    printf(" |-----| \n");
    printf(" | 평균 |      ");
    if ((int)S->awt / 10 == 0) printf(" "); //평균이 한자리 나와 표가 깨지는 경우를 방지하기 위한 조건문
    printf("%.2f | \n", S->awt); //소수점 2자리까지만 출력
    printf(" |-----| ");
    printf("\n |-----| \n");
    printf(" | ID |   응답시간   | \n");
    printf(" |-----| \n");
    for (int i = 0; i < S->pro_count; i++)
        printf(" | P%-2d |      %3d      | \n", S->list[i].pro_id, S->list[i].res_t);
    printf(" |-----| \n");
    printf(" | 평균 |      ");
    if ((int)S->art / 10 == 0) printf(" "); //평균이 한자리 나와 표가 깨지는 경우를 방지하기 위한 조건문
    printf("%.2f | \n", S->art); //소수점 2자리까지만 출력
    printf(" |-----| ");
    printf("\n |-----| \n");
    printf(" | ID |   반환시간   | \n");
    printf(" |-----| \n");
    for (int i = 0; i < S->pro_count; i++)
        printf(" | P%-2d |      %3d      | \n", S->list[i].pro_id, S->list[i].turn_a_t);
    printf(" |-----| \n");
    printf(" | 평균 |      ");
    if ((int)S->att / 10 == 0) printf(" "); //평균이 한자리 나와 표가 깨지는 경우를 방지하기 위한 조건문
    printf("%.2f | \n", S->att); //소수점 2자리까지만 출력
    printf(" |-----| \n\n");
}
```

### 3-2. 실행창

ID	대기시간	ID	응답시간	ID	반환시간
P1	0	P1	1	P1	10
P2	9	P2	10	P2	37
P3	36	P3	37	P3	42
P4	41	P4	42	P4	45
P5	44	P5	45	P5	58
평균	26.00	평균	27.00	평균	38.40

- 실행창

## 초기 화면

```
C:\Users\Wksdy\Source\Wrep x + v
<프로세스 정보>

ID   도착시간   서비스시간   우선순위
P1   0           10           3
P2   1           28           2
P3   2           6            4
P4   3           4            1
P5   4           14           2

[타임 슬라이스 2]
=====MENU=====
1. FCFS 스케줄링
2. SJF 스케줄링
3. HRN 스케줄링
4. 비선점형 우선순위 스케줄링
5. 라운드로빈 스케줄링
6. SRT 스케줄링
7. 선점형 우선순위 스케줄링
8. 종료
=====
메뉴를 선택해주세요: 1
```

## 메뉴 1번 선택

[illegible]



## 메뉴 2번 선택

```
C:\Users\Wskdy\source\Wrep x + v
<SJF 스케줄링>
[간트차트]
P1P1P1P1P1P1P1P1[10] | P4P4P4P4[4] | P3P3P3P3P3P3[6] | P5P5P5P5P5P5P5P5P5P5P5[14] | P2P2P2P2P2P2P2P2P2P2P2P2P2P2P2P2P2P2P2P2P2P2P2P2[28] |
```

ID	대기시간
P1	0
P2	33
P3	12
P4	7
P5	16
평균	13.60

ID	응답시간
P1	1
P2	34
P3	13
P4	8
P5	17
평균	14.60

ID	반환시간
P1	10
P2	61
P3	18
P4	11
P5	30
평균	26.00

```
=====MENU=====
1. FCFS 스케줄링
2. SJF 스케줄링
3. HRN 스케줄링
4. 비선점형 우선순위 스케줄링
5. 라운드로빈 스케줄링
6. SRT 스케줄링
7. 선점형 우선순위 스케줄링
8. 종료
=====
메뉴를 선택해주세요: 3
```

## 메뉴 3번 선택

```
C:\Users\Wskdy\source\Wrep x + v
<HRN 스케줄링>
[간트차트]
P1P1P1P1P1P1P1P1[10] | P4P4P4P4[4] | P3P3P3P3P3P3[6] | P5P5P5P5P5P5P5P5P5P5P5[14] | P2P2P2P2P2P2P2P2P2P2P2P2P2P2P2P2P2P2P2P2P2P2P2P2[28] |
```

ID	대기시간
P1	0
P2	33
P3	12
P4	7
P5	16
평균	13.60

ID	응답시간
P1	1
P2	34
P3	13
P4	8
P5	17
평균	14.60

ID	반환시간
P1	10
P2	61
P3	18
P4	11
P5	30
평균	26.00

```
=====MENU=====
1. FCFS 스케줄링
2. SJF 스케줄링
3. HRN 스케줄링
4. 비선점형 우선순위 스케줄링
5. 라운드로빈 스케줄링
6. SRT 스케줄링
7. 선점형 우선순위 스케줄링
8. 종료
=====
메뉴를 선택해주세요: 4
```

## 메뉴 4번 선택

```
C:\Users\Wksdy\Source\Wrep x + v
<비선점형 우선순위 스케줄링>
[간트차트]
P1P1P1P1P1P1P1P1P1[10] | P4P4P4P4[4] | P2P2P2P2P2P2P2P2P2P2P2P2P2P2P2P2P2P2P2P2[28] | P5P5P5P5P5P5P5P5P5P5P5P5[14] | P3P3P3P3P3P3[6] |



| ID | 대기 시간 |
|----|-------|
| P1 | 0     |
| P2 | 13    |
| P3 | 54    |
| P4 | 7     |
| P5 | 38    |
| 평균 | 22.40 |



| ID | 응답 시간 |
|----|-------|
| P1 | 1     |
| P2 | 14    |
| P3 | 55    |
| P4 | 8     |
| P5 | 39    |
| 평균 | 23.40 |



| ID | 반환 시간 |
|----|-------|
| P1 | 10    |
| P2 | 41    |
| P3 | 60    |
| P4 | 11    |
| P5 | 52    |
| 평균 | 34.80 |



=====MENU=====
1. FCFS 스케줄링
2. SJF 스케줄링
3. HRN 스케줄링
4. 비선점형 우선순위 스케줄링
5. 라운드로빈 스케줄링
6. SRT 스케줄링
7. 선점형 우선순위 스케줄링
8. 종료
=====
메뉴를 선택해주세요: 5
```

## 메뉴 5번 선택

```
C:\Users\Wksdy\Source\Wrep x + v
<Round-Robin 스케줄링>
[간트차트]
P1P1[2] | P2P2[2] | P3P3[2] | P1P1[2] | P4P4[2] | P5P5[2] | P2P2[2] | P3P3[2] | P1P1[2] | P4P4[2] | P5P5[2] | P2P2[2] | P3P3[2] | P1P1[2] | P5P5[2] | P2P2[2] | P5P5[2] | P2P2[2] | P5P5[2] | P2P2[2] | P5P5[2] | P2P2[2] | P2P2[2] | P2P2[2] | P2P2[2] | P2P2[2] | P2P2[2] | P2P2[2] |



| ID | 대기 시간 |
|----|-------|
| P1 | 24    |
| P2 | 33    |
| P3 | 18    |
| P4 | 13    |
| P5 | 30    |
| 평균 | 23.60 |



| ID | 응답 시간 |
|----|-------|
| P1 | 1     |
| P2 | 2     |
| P3 | 3     |
| P4 | 6     |
| P5 | 7     |
| 평균 | 3.80  |



| ID | 반환 시간 |
|----|-------|
| P1 | 34    |
| P2 | 61    |
| P3 | 24    |
| P4 | 17    |
| P5 | 44    |
| 평균 | 36.00 |



=====MENU=====
1. FCFS 스케줄링
2. SJF 스케줄링
3. HRN 스케줄링
4. 비선점형 우선순위 스케줄링
5. 라운드로빈 스케줄링
6. SRT 스케줄링
7. 선점형 우선순위 스케줄링
8. 종료
=====
메뉴를 선택해주세요: 6
```

## 메뉴 6번 선택

```

C:\Users\Wksdy\source\Wrep  X  +  v
<SRT 스케줄링>
[간트차트]
P1P1[2] | P3P3[2] | P3P3[2] | P4P4[2] | P4P4[2] | P1P1[2] | P1P1[2] | P1P1[2] | P1P1[2] | P5P5[2] | P5P5[2] | P5P5[2] | P5P5[2] | P5P5[2] | P5P5[2] | P5P5[2] | P2P2[2] |
P2P2[2] | P2P2[2] | P2P2[2] | P2P2[2] | P2P2[2] | P2P2[2] | P2P2[2] | P2P2[2] | P2P2[2] | P2P2[2] | P2P2[2] | P2P2[2] |
ID 대기 시간
P1 10
P2 33
P3 0
P4 5
P5 16
평균 12.80

ID 응답 시간
P1 1
P2 34
P3 1
P4 6
P5 17
평균 11.80

ID 반환 시간
P1 20
P2 61
P3 6
P4 9
P5 30
평균 25.20

=====MENU=====
1. FCFS 스케줄링
2. SJF 스케줄링
3. HRN 스케줄링
4. 비선점형 우선순위 스케줄링
5. 라운드로빈 스케줄링
6. SRT 스케줄링
7. 선점형 우선순위 스케줄링
8. 종료

메뉴를 선택해주세요: 7
=====

```

## 메뉴 7번 선택

```
C:\Users\Wdksdy\SourceWrep  ×  +  ▾
<선점형 우선순위 스케줄링>
[간트차트]
P1[1] | P2P2[2] | P4P4P4P4[4] | P2P2P2P2P2P2P2P2P2P2P2P2P2P2P2P2P2[26] | P5P5P5P5P5P5P5P5P5P5P5P5P5[14] | P1P1P1P1P1P1P1P1[9] | P3P3P3P3P3P3[6] |

ID   대기 시간
P1   46
P2   4
P3   54
P4   0
P5   29
평균 26.60

ID   응답 시간
P1   1
P2   1
P3   55
P4   1
P5   30
평균 17.60

ID   반환 시간
P1   56
P2   32
P3   60
P4   4
P5   43
평균 39.00

=====MENU=====
1. FCFS 스케줄링
2. SJF 스케줄링
3. HRN 스케줄링
4. 비선점형 우선순위 스케줄링
5. 라운드로빈 스케줄링
6. SRT 스케줄링
7. 선점형 우선순위 스케줄링
8. 종료

=====
메뉴를 선택해주세요: 12
```

## 다른 숫자 입력

```
C:\Users\Wksdy\source\Wrep x + v
<프로세스 정보>


| ID | 도착시간 | 서비스시간 | 우선순위 |
|----|------|-------|------|
| P1 | 0    | 10    | 3    |
| P2 | 1    | 28    | 2    |
| P3 | 2    | 6     | 4    |
| P4 | 3    | 4     | 1    |
| P5 | 4    | 14    | 2    |


[타임 슬라이스 2]
=====MENU=====
1. FCFS 스케줄링
2. SJF 스케줄링
3. HRN 스케줄링
4. 비선점형 우선순위 스케줄링
5. 라운드로빈 스케줄링
6. SRT 스케줄링
7. 선점형 우선순위 스케줄링
8. 종료
=====
메뉴를 선택해주세요: 8
```

## 메뉴 8번 선택

```
Microsoft Visual Studio 디버그 x + v
C:\Users\dksdy\source\repos\Project1\x64\Debug\Project1.exe(프로세스 17212개)이(가) 종료되었습니다(코드: 1개).
디버깅이 중지될 때 콘솔을 자동으로 닫으려면 [도구] -> [옵션] -> [디버깅] > [디버깅이 중지되면 자동으로 콘솔 닫기]를 사용하도록 설정합니다.
이 창을 닫으려면 아무 키나 누르세요...
```

## • 느낀점

이번 과제를 통해 CPU 스케줄링에 대한 이해도가 높아진 거 같습니다. 책에 있던 문제를 풀고 이해했던 것만으로 코드를 작성하는 것은 생각보다 어려웠습니다. 그래서 자세히 각각의 방법이 어떻게 작동하는지 다시 공부한 후에 코드를 작성하였습니다. 그 결과 구현을 할 수 있었습니다. 하지만 생각보다 예외가 많아 어려움을 겪었습니다. 서비스 시간이나 우선순위, 도착시간 등이 같은 경우는 갈아질 수 없는 프로세스 ID를 기준으로 다시 정렬하여 해결하였습니다. 현재 코드는 복잡하고 최적화하지 못하여 아쉬움이 많이 남았습니다. 학기 중이 아니라 방학 중에 조금 더 좋은 알고리즘으로 해보고 싶다는 생각을 하였고 이번 과제를 계기로 코딩 실력이 조금이나마 향상된 거 같아 뿌듯한 느낌이 들었습니다. 공부할 때 이론만 공부하는 것이 아니라 직접 코드로 작성해보는 것이 더 도움이 된다는 것을 깨달았고 앞으로는 이론 수업도 조금씩 시간을 더 투자하여 코드로 작성해봐야겠다는 것다는 다짐을 하게되었습니다.