운영체제 텀 프로젝트 CPU Scheduling Simulator 보고서

2021350011 김준우

서론

CPU 스케줄러는 여러 개의 프로세스가 대기 중일 때, 어느 프로세스가 우선적으로 처리되어야 하는지를 정해주어 각각의 프로세스들이 원활히 수행되도록 하는 것을 의미한다. 우선순위를 결정하는 요소는 각 알고리즘마다 상이하지만, 기본적으로 프로세스들의 waiting time과 turnaround time을 최소화 하는 것을 효율적으로 본다. Turnaround time이란 프로세스가 ready state에서부터 프로세스 진행 완료까지 걸리는 총 시간을 의미하며, 간단히 exit time에서 arrival time을 뺀 것으로 구할 수가 있다. Waiting time은 프로세스가 도착한 이후부터 cpu 자원을 받을 때까지 기다리는 시간을 의미하며 간단히 turnaround time에서 burst time을 뺀 것을 의미한다.

CPU 스케줄러에 사용되는 알고리즘은 방법에 따라 여러가지가 있지만, 텀 프로젝트 데모 기간에 구현된 알고리즘은 6개로, FCFS, SJF, Priority, Round Robin, Non-preemptive SJF, Non-preemptive Priority였다. 이는 가장 대표적인 알고리즘으로 요구 명세서에 명시되어 있는 알고리즘이다.

본인이 구현한 스케줄러에서는, 원래 프로세스마다 랜덤한 burst time, arrival time, priority를 가지도록 했어야 하는데, 프로세스를 여러 요소별로 sorting하는 알고리즘을 구현하지 못해서 그냥 특정 값을 가지도록 설정한 이후에 알고리즘을 구현했다. 만약 sorting 알고리즘을 구현한다면 최대 프로세스 수가 5를 넘지 못했기 때문에 가장 간단한 selection sort로 sorting을 할 것이다.

그리고 구현해 놓은 간트 차트를 통해 각 알고리즘 별로 어떤 프로세스가 얼만큼의 turnaround time과 waiting time을 갖는지를 확인하도록 했다.

본론

본인이 만든 스케줄링 시뮬레이터에 대해서 설명하기 전에, 다른 cpu 스케줄링 시뮬레이터에 대해 보자면, preemptive priority scheduling 알고리즘을 사용할 때 aging 기법을 사용한 예시를 볼 수 있다. Aging 기법은 낮은 우선순위를 가진 프로세스가 높은 우선순위의 프로세스에 의해 계속해서 cpu를 빼앗기며 실행되지 못하는 기아 문제를 해결하기 위해 사용된다. Aging 기법은 시간이 지날수록 우선순위를 점차 높여주는 것으로 우선순위가 낮다 하더라도 유동적으로 대처할 수 있도록 해준다.

```
for (int i = 0; i < cur_proc_num_RQ; i++) {
    if (readyQueue[i]->priority != 1)
        // priority upgrade
        readyQueue[i]->priority -= (amount - readyQueue[i]->arrivalTime) / AGING_PERIOD;
}
```

cpu 스케줄링 시뮬레이터 우수 예시에서 aging 기법을 사용한 부분은 다음과 같은데, amount-arrivaltime을 통해 프로세스가 ready queue에서 얼마나 오래 기다렸는지를 파악하고, 이 대기 시간을 aging period로 나누어 대기한 시간에 비례하여 priority를 낮춰준다. 숫자를 줄이면 우선순위가 높아지므로, 대기 시간이 길수록 우선순위가 점점 높아지는 효과를 보여 단순히 프로세스마다 부여된 우선순위에 맞게 진행하지 않고 유동적으로 스케줄을 조절할 수 있게 해준다.

내가 구현한 시뮬레이터는 매우 단순한데,

```
#define MAX_PROCESS 10
#define MAX_BURST 20
#define MAX_PRIORITY 5
```

를 통해 프로세스의 개수와 cpu burst 시간, priority등을 정의한 이후

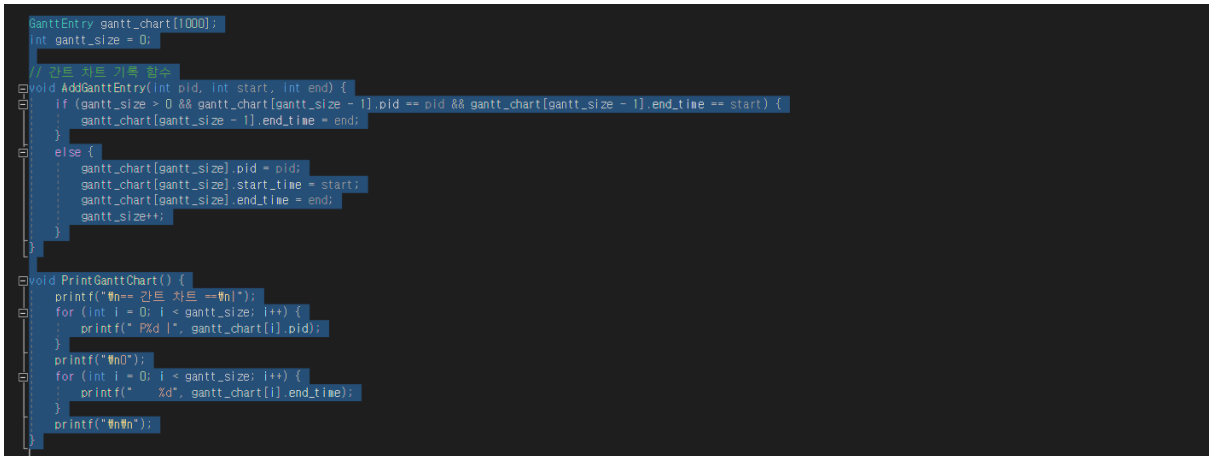
```
typedef struct {
    int pid;
```

```
    int arrival_time;
    int total_cpu_burst;
    int priority;

    int waiting_time;
    int turnaround_time;
} Process;

typedef struct {
    int pid;
    int start_time;
    int end_time;
```

를 통해 arrival time, total cpu burst, waiting time, turnaround time, pid등을 한 묶음으로 하는 구조체를 통해 프로세스의 특징을 정의하고,

간트 차트를 start time과 end time을 이용해 계산하는 함수와 만든 간트 차트를 출력하는 함수를 구현했다. 그 이후 상술한 6개의 알고리즘 중 라운드 로빈 알고리즘을 제외한 5개를 간단하게 구현했다. 소팅 알고리즘을 미구현 했으므로 5개의 프로세스의 priority, arrival time, cpu burst time 등을 임의로 설정한 후 돌아갈 수 있게 하였다.

모듈 설명 1: 간트 차트 구현



```
GanttEntry gantt_chart[1000];
int gantt_size = 0;

// 간트 차트 기록 함수
void AddGanttEntry(int pid, int start, int end) {
    if (gantt_size > 0 && gantt_chart[gantt_size - 1].pid == pid && gantt_chart[gantt_size - 1].end_time == start) {
        gantt_chart[gantt_size - 1].end_time = end;
    }
    else {
        gantt_chart[gantt_size].pid = pid;
        gantt_chart[gantt_size].start_time = start;
        gantt_chart[gantt_size].end_time = end;
        gantt_size++;
    }
}

void PrintGanttChart() {
    printf("\n== 간트 차트 ==\n");
    for (int i = 0; i < gantt_size; i++) {
        printf(" P%d |", gantt_chart[i].pid);
    }
    printf("\n0");
    for (int i = 0; i < gantt_size; i++) {
        printf("    %d", gantt_chart[i].end_time);
    }
    printf("\n\n");
}
```

설명: Addgantentry 함수에서 기존에 실행되던 프로세스가 지속될 경우 end_time만 늘려주고, 프로세스가 바뀔 경우 새 프로세스의 start_time과 end_time을 기록하는 식으로 간트 차트를 기록했다. 그리고 printganttchart 함수를 통해 addgantentry 함수에 기록된 결과물을 프린트했다.

2. 원본 프로세스 및fcfs

```
// 원본 프로세스(초기화 용도)
void InitProcesses() {
    for (int i = 0; i < process_count; i++) {
        processes[i] = original_processes[i];
        processes[i].waiting_time = 0;
        processes[i].turnaround_time = 0;
    }
    gantt_size = 0;
}


// FCFS
void FCFS() {
    InitProcesses();
    int time = 0;
    for (int i = 0; i < process_count; i++) {
        if (time < processes[i].arrival_time) time = processes[i].arrival_time;
        processes[i].waiting_time = time - processes[i].arrival_time;
        AddGanttEntry(processes[i].pid, time, time + processes[i].total_cpu_burst);
        time += processes[i].total_cpu_burst;
        processes[i].turnaround_time = time - processes[i].arrival_time;
    }
    printf("FCFS 스케줄링 \n");
    PrintGanttChart();
}
```

특정 알고리즘이 실행된 다음 구조체의 요소를 초기화 시키는 initprocesses와 fcfs를 구현했다. Fcfs를 하려면 sorting을 해야 하지만, sorting을 미구현해서 예시 프로세스의 arrival time을 오름차순으로 배치했다. Waiting time을 arrival time, total cpu burst와의 덧셈과 뺄셈으로 연산해 결과물을 간트차트에 적게 하고 있다.

3. Non-preemptive SJF

```
// Non-preemptive SJF
void NonPreemptiveSJF() {
    InitProcesses();
    int completed = 0, time = 0;
    int finished[MAX_PROCESS] = { 0 };

    while (completed < process_count) {
        int index = -1;
        int min_burst = 100;
        for (int i = 0; i < process_count; i++) {
            if (!finished[i] && processes[i].arrival_time <= time && processes[i].total_cpu_burst < min_burst) {
                min_burst = processes[i].total_cpu_burst;
                index = i;
            }
        }
        if (index == -1) {
            time++;
            continue;
        }
        processes[index].waiting_time = time - processes[index].arrival_time;
        AddGanttEntry(processes[index].pid, time, time + processes[index].total_cpu_burst);
        time += processes[index].total_cpu_burst;
        processes[index].turnaround_time = time - processes[index].arrival_time;
        finished[index] = 1;
        completed++;
    }
    printf("Non-preemptive SJF 스케줄링₩n");
    PrintGanttChart();
}
```

비선점형 sjf를 구현했다. Finished[max_process]를 돌면서 process가 끝났는지 안 끝났는지를 매번 확인하는 것은 비효율적이므로 completed 변수를 준비해 completed가 process_count와 같아질때까지 while문을 돌리는 방식으로 모든 process가 완료될때까지 돌리려고 했다. 새로운 process가 cpu 자원을 먹을 경우 간트차트의 starttime과 endtime을 바꿔주고 그만큼 time 변수를 앞뒤로 땡기는 식으로 arrivaltime과 turnaroundtime을 계산했다

4. Non-preemptive priority

```
// Non-preemptive Priority
void NonPreemptivePriority() {
    InitProcesses();
    int completed = 0, time = 0;
    int finished[MAX_PROCESS] = { 0 };

    while (completed < process_count) {
        int index = -1;
        int highest_priority = 100;
        for (int i = 0; i < process_count; i++) {
            if (!finished[i] && processes[i].arrival_time <= time && processes[i].priority < highest_priority) {
                highest_priority = processes[i].priority;
                index = i;
            }
        }
        if (index == -1) {
            time++;
            continue;
        }
        processes[index].waiting_time = time - processes[index].arrival_time;
        AddGanttEntry(processes[index].pid, time, time + processes[index].total_cpu_burst);
        time += processes[index].total_cpu_burst;
        processes[index].turnaround_time = time - processes[index].arrival_time;
        finished[index] = 1;
        completed++;
    }
    printf("Non-preemptive Priority 스케줄링₩n");
    PrintGanttChart();
}
```

비선점형 priority를 구현했다. Arrival_time이 priority로 바뀐 것을 제외하면 똑같은 메커니즘을 가진다.

5. preemptive priority

```
// Preemptive Priority
void PreemptivePriority() {
    InitProcesses();
    int completed = 0, time = 0;
    int remaining[MAX_PROCESS];
    int last_pid = -1;

    for (int i = 0; i < process_count; i++) {
        remaining[i] = processes[i].total_cpu_burst;
    }

    while (completed < process_count) {
        int index = -1;
        int highest_priority = 100;
        for (int i = 0; i < process_count; i++) {
            if (processes[i].arrival_time <= time && remaining[i] > 0 && processes[i].priority < highest_priority) {
                highest_priority = processes[i].priority;
                index = i;
            }
        }
        if (index == -1) {
            time++;
            continue;
        }

        if (last_pid != index) {
            if (last_pid != -1)
                AddGanttEntry(last_pid, time - 1, time);
            AddGanttEntry(index, time, time + 1);
        }
        else {
            if (gantt_size > 0 && gantt_chart[gantt_size - 1].pid == index && gantt_chart[gantt_size - 1].end_time == time)
                gantt_chart[gantt_size - 1].end_time++;
            else
                AddGanttEntry(index, time, time + 1);
        }

        remaining[index]--;
        time++;
        last_pid = index;

        if (remaining[index] == 0) {
            processes[index].turnaround_time = time - processes[index].arrival_time;
            processes[index].waiting_time = processes[index].turnaround_time - processes[index].total_cpu_burst;
            completed++;
        }
    }
```

선점형 priority를 구현했다. 한번 프로세스가 걸리면 완료할 때 멈추지 않는 비선점형 방식과 다르게 실시간으로 어느 priority가 높은지를 비교해야 했다. 따라서 기존 비선점형 방식들이 completed와 remaining[]을 통해 프로세스가 끝났는지만 확인하면서 맡은 프로세스의 burst time을 time에 더해주는 식으로 time이 1씩 증가할 수도, 프로세스의 cpu burst time만큼 증가할 수도 있었지만, 선점형은 매초 priority가 바뀌므로 time을 1씩 늘려주면서 highest_priority 변수를 통해 매초 가장 높은 priority를 관리했다.

6. preemptive sjf

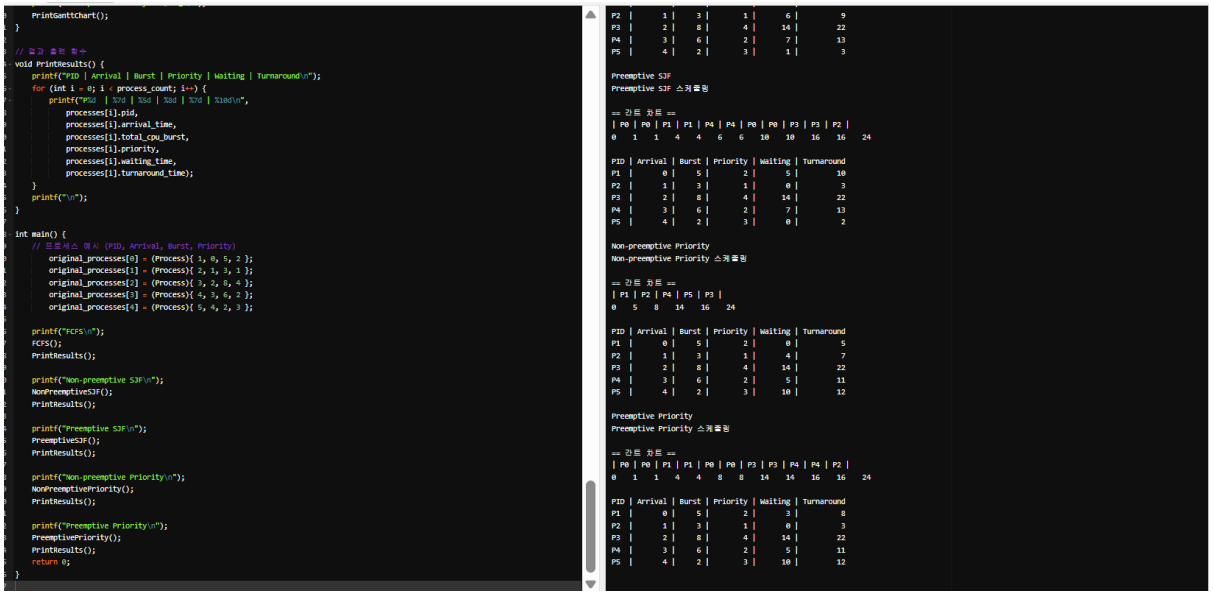
```c
// Preemptive SJF
void PreemptiveSJF() {
    InitProcesses();
    int completed = 0, time = 0;
    int remaining[MAX_PROCESS];
    int last_pid = -1;

    for (int i = 0; i < process_count; i++) {
        remaining[i] = processes[i].total_cpu_burst;
    }

    while (completed < process_count) {
        int index = -1;
        int min_remain = 100;
        for (int i = 0; i < process_count; i++) {
            if (processes[i].arrival_time <= time && remaining[i] > 0 && remaining[i] < min_remain) {
                min_remain = remaining[i];
                index = i;
            }
        }
        if (index == -1) {
            time++;
            continue;
        }

        if (last_pid != index) {
            if (last_pid != -1)
                AddGanttEntry(last_pid, time - 1, time);
            AddGanttEntry(index, time, time + 1);
        }
        else {
            if (gantt_size > 0 && gantt_chart[gantt_size - 1].pid == index && gantt_chart[gantt_size - 1].end_time == time)
                gantt_chart[gantt_size - 1].end_time++;
            else
                AddGanttEntry(index, time, time + 1);
        }

        remaining[index]--;
        time++;
        last_pid = index;

        if (remaining[index] == 0) {
            processes[index].turnaround_time = time - processes[index].arrival_time;
            processes[index].waiting_time = processes[index].turnaround_time - processes[index].total_cpu_burst;
            completed++;
```

선점형 sjf 방식이다. Highest priority 대신 min_remain을 통해 관리하는 것을 빼면 선점형 priority 방식과 같은 메커니즘이다.

실행 결과

```c
    PrintGanttChart();
}

// 결과 출력 함수
void PrintResults() {
    printf("PID | Arrival | Burst | Priority | Waiting | Turnaround\n");
    for (int i = 0; i < process_count; i++) {
        printf("P%d  | %7d | %5d | %8d | %7d | %10d\n",
            processes[i].pid,
            processes[i].arrival_time,
            processes[i].total_cpu_burst,
            processes[i].priority,
            processes[i].waiting_time,
            processes[i].turnaround_time);
    }
    printf("\n");
}

int main() {
    // 프로세스 예시 (PID, Arrival, Burst, Priority)
    original_processes[0] = (Process){ 1, 0, 5, 2 };
    original_processes[1] = (Process){ 2, 1, 3, 1 };
    original_processes[2] = (Process){ 3, 2, 8, 4 };
    original_processes[3] = (Process){ 4, 3, 6, 2 };
    original_processes[4] = (Process){ 5, 4, 2, 3 };

    printf("FCFS\n");
    FCFS();
    PrintResults();

    printf("Non-preemptive SJF\n");
    NonPreemptiveSJF();
    PrintResults();

    printf("Preemptive SJF\n");
    PreemptiveSJF();
    PrintResults();

    printf("Non-preemptive Priority\n");
    NonPreemptivePriority();
    PrintResults();

    printf("Preemptive Priority\n");
    PreemptivePriority();
    PrintResults();
    return 0;
}
```

```
FCFS
FCFS 스케줄링

== 간트 차트 ==
| P1 | P2 | P3 | P4 | P5 |
0   5   8   16   22   24

PID | Arrival | Burst | Priority | Waiting | Turnaround
P1  |       0 |     5 |        2 |       0 |          5
P2  |       1 |     3 |        1 |       4 |          7
P3  |       2 |     8 |        4 |       6 |         14
P4  |       3 |     6 |        2 |      13 |         19
P5  |       4 |     2 |        3 |      18 |         20

Non-preemptive SJF
Non-preemptive SJF 스케줄링

== 간트 차트 ==
| P1 | P5 | P2 | P4 | P3 |
0   5   7   10   16   24

PID | Arrival | Burst | Priority | Waiting | Turnaround
P1  |       0 |     5 |        2 |       0 |          5
P2  |       1 |     3 |        1 |       6 |          9
P3  |       2 |     8 |        4 |      14 |         22
P4  |       3 |     6 |        2 |       7 |         13
P5  |       4 |     2 |        3 |       1 |          3

Preemptive SJF
Preemptive SJF 스케줄링

== 간트 차트 ==
| P0 | P0 | P1 | P1 | P4 | P4 | P0 | P0 | P3 | P3 | P2 |
0   1   1   4   4   6   6   10   10   16   16   24

PID | Arrival | Burst | Priority | Waiting | Turnaround
P1  |       0 |     5 |        2 |       5 |         10
P2  |       1 |     3 |        1 |       0 |          3
P3  |       2 |     8 |        4 |      14 |         22
P4  |       3 |     6 |        2 |       7 |         13
P5  |       4 |     2 |        3 |       0 |          2

Non-preemptive Priority
Non-preemptive Priority 스케줄링

== 간트 차트 ==
| P1 | P2 | P4 | P5 | P3 |
0   5   8   14   16   24
```



```
    PrintGanttChart();
}

// 결과 출력 함수
void PrintResults() {
    printf("PID | Arrival | Burst | Priority | Waiting | Turnaround\n");
    for (int i = 0; i < process_count; i++) {
        printf("P%d  | %7d | %5d | %8d | %7d | %10d\n",
            processes[i].pid,
            processes[i].arrival_time,
            processes[i].total_cpu_burst,
            processes[i].priority,
            processes[i].waiting_time,
            processes[i].turnaround_time);
    }
    printf("\n");
}

int main() {
    // 프로세스 예시 (PID, Arrival, Burst, Priority)
        original_processes[0] = (Process){ 1, 0, 5, 2 };
        original_processes[1] = (Process){ 2, 1, 3, 1 };
        original_processes[2] = (Process){ 3, 2, 8, 4 };
        original_processes[3] = (Process){ 4, 3, 6, 2 };
        original_processes[4] = (Process){ 5, 4, 2, 3 };

    printf("FCFS\n");
    FCFS();
    PrintResults();

    printf("Non-preemptive SJF\n");
    NonPreemptiveSJF();
    PrintResults();

    printf("Preemptive SJF\n");
    PreemptiveSJF();
    PrintResults();

    printf("Non-preemptive Priority\n");
    NonPreemptivePriority();
    PrintResults();

    printf("Preemptive Priority\n");
    PreemptivePriority();
    PrintResults();
    return 0;
}
```

```
P2  |      1 |     3 |        1 |      6 |         9
P3  |      2 |     8 |        4 |     14 |        22
P4  |      3 |     6 |        2 |      7 |        13
P5  |      4 |     2 |        3 |      1 |         3

Preemptive SJF
Preemptive SJF 스케줄링

== 간트 차트 ==
| P0 | P0 | P1 | P1 | P4 | P4 | P0 | P0 | P3 | P3 | P2 |
0    1    1    4    4    6    6    10   10   16   16   24

PID | Arrival | Burst | Priority | Waiting | Turnaround
P1  |      0 |     5 |        2 |       5 |         10
P2  |      1 |     3 |        1 |       0 |          3
P3  |      2 |     8 |        4 |      14 |         22
P4  |      3 |     6 |        2 |       7 |         13
P5  |      4 |     2 |        3 |       0 |          2

Non-preemptive Priority
Non-preemptive Priority 스케줄링

== 간트 차트 ==
| P1 | P2 | P4 | P5 | P3 |
0    5    8    14   16   24

PID | Arrival | Burst | Priority | Waiting | Turnaround
P1  |      0 |     5 |        2 |       0 |          5
P2  |      1 |     3 |        1 |       4 |          7
P3  |      2 |     8 |        4 |      14 |         22
P4  |      3 |     6 |        2 |       5 |         11
P5  |      4 |     2 |        3 |      10 |         12

Preemptive Priority
Preemptive Priority 스케줄링

== 간트 차트 ==
| P0 | P0 | P1 | P1 | P0 | P0 | P3 | P3 | P4 | P4 | P2 |
0    1    1    4    4    8    8    14   14   16   16   24

PID | Arrival | Burst | Priority | Waiting | Turnaround
P1  |      0 |     5 |        2 |       3 |          8
P2  |      1 |     3 |        1 |       0 |          3
P3  |      2 |     8 |        4 |      14 |         22
P4  |      3 |     6 |        2 |       5 |         11
P5  |      4 |     2 |        3 |      10 |         12
```

알고리즘들간의 성능 비교를 위한 함수는 구현하지 않았지만, 매 초마다 연산을 반복해야 하는 선점 방식이 컴퓨팅 자원을 많이 잡아먹을 것으로 예상된다. 예시에서 arrival time을 오름차순으로 정리해 sorting하는 컴퓨팅 자원이 없어 자원을 제일 덜 먹는 것은 fcfs이지만, process의 개수가 증가함에 따라 sorting에 있어 자원 소모량도 따로 봐야 할 것 같다.

결론

구현한 시뮬레이터에 대한 정리해보자면 간단하게 round robin을 제외한 명세에 있는 알고리즘이 실행되는지 정도만 확인해 볼 수 있는 수준의 코드를 제출해 냈다. 부족한 부분이 매우 많아 데모 시연 과정에서 round robin을 구현 못했지만 사후적으로 구현한 코드는 다음과 같다.

```
// Round Robin
void RoundRobin(int time_quantum) {
    InitProcesses();
    int remaining[MAX_PROCESS];
    int time = 0;
    int completed = 0;
    int last_pid = -1;
    int queue[MAX_PROCESS];
    int front = 0, rear = 0;
    int in_queue[MAX_PROCESS] = { 0 };

    for (int i = 0; i < process_count; i++) {
        remaining[i] = processes[i].total_cpu_burst;
    }

    while (1) {
        for (int i = 0; i < process_count; i++) {
            if (!in_queue[i] && processes[i].arrival_time <= time) {
                queue[rear++] = i;
                in_queue[i] = 1;
            }
        }
        if (rear > 0) break;
        time++;
    }

    while (completed < process_count) {
        int index = queue[front++];

        int exec_time = (remaining[index] > time_quantum) ? time_quantum : remaining[index];
        AddGanttEntry(processes[index].pid, time, time + exec_time);
        time += exec_time;
        remaining[index] -= exec_time;


        for (int i = 0; i < process_count; i++) {
            if (!in_queue[i] && processes[i].arrival_time <= time) {
                queue[rear++] = i;
                in_queue[i] = 1;
            }
        }

        if (remaining[index] > 0) {
            queue[rear++] = index;
        }
        else {
            processes[index].turnaround_time = time - processes[index].arrival_time;
            processes[index].waiting_time = processes[index].turnaround_time - processes[index].total_cpu_burst;
            completed++;
        }
```

```
        }
        if (remaining[index] > 0) {
            queue[rear++] = index;
        }
        else {
            processes[index].turnaround_time = time - processes[index].arrival_time;
            processes[index].waiting_time = processes[index].turnaround_time - processes[index].total_cpu_burst;
            completed++;
        }


        if (front == rear && completed < process_count) {
            for (int i = 0; i < process_count; i++) {
                if (!in_queue[i]) {
                    queue[rear++] = i;
                    in_queue[i] = 1;
                    if (time < processes[i].arrival_time)
                        time = processes[i].arrival_time;
                    break;
                }
            }
        }
    }

    printf("Round Robin (Time Quantum = %d) 스케줄링₩n", time_quantum);
    PrintGanttChart();
}
```

간단히 설명하자면 위에 있던 비선점형 알고리즘을 구현할 때와 마찬가지로 remaining[]와 completed 변수를 가지고 알고리즘 시작과 종료를 제어했고, 타임 퀀텀이 끝났을때 남은 프로세스를 queue의 맨 뒤로 보내는 것을 구현하기 위해 queue를 만들었다.

프로젝트 수행 소감은 처음엔 매우 막막했는데 preemptive priority algorithm을 구현한 이후부터 그나마 감을 좀 잡은 것 같다. 시간 부족으로 round robin을 구현하지 못했는데 어차피 process 5개 임의로 arrival time, burst time, priority time 까지 일일이 지정해서 실행시킨 거 조금만 시간 더 투자했으면 되었을 텐데 안타깝다.

다음 번에 다른 텀 프로젝트를 한다면 더 잘할 수 있을 것 같다는 자신감이 생겨서 그건 감사하다는 생각이 들었다.