목차

1. Implement FCFS & Round Robin Scheduling Algorithms

FCFS (Core logic)

```
for (int i = 0; i < n; i++) {
    // 프로세스 시작 시간, 완료 시간, 반환 시간, 대기 시간, 응답 시간을 계산
    p[i].start_time = (i == 0) ? p[i].arrival_time : max(p[i - 1].completion_time,
p[i].arrival_time);
    p[i].completion_time = p[i].start_time + p[i].burst_time;
    p[i].turnaround_time = p[i].completion_time - p[i].arrival_time;
    p[i].waiting_time = p[i].turnaround_time - p[i].burst_time;
    p[i].response_time = p[i].start_time - p[i].arrival_time;

    // 총 시간들을 계산
    total_turnaround_time += p[i].turnaround_time;
    total_waiting_time += p[i].waiting_time;
    total_response_time += p[i].response_time;
    total_idle_time += (i == 0) ? (p[i].arrival_time) : (p[i].start_time - p[i -
1].completion_time);
}
```

FCFS 스케줄링의 동작에 맞게 프로세스들을 선입선출 방식으로 스케줄링하고 그에 따른 시작 시간, 완료 시간, 반환 시간, 대기 시간, 응답 시간을 구하는 로직을 구현하였다.

Round Robin (Core logic)

```
// 프로세스 큐
queue<int> q;
// 현재 시간
int current_time = 0;
// 첫 번째 프로세스를 큐에 추가
q.push(0);
// 완료된 프로세스 수
int completed = 0;
// 프로세스의 상태 표시 배열
int mark[100];
memset(mark, 0, sizeof(mark));
mark[0] = 1; // 첫 번째 프로세스 표시 초기화

// 모든 프로세스가 완료될 때 까지
while (completed != n) {
    // 큐의 첫번째 프로세스를 인출
    idx = q.front();
    q.pop();

    // 프로세스가 처음 실행되는 경우 초기화
    if (burst_remaining[idx] == p[idx].burst_time) {
        p[idx].start_time = max(current_time, p[idx].arrival_time);
        total_idle_time += p[idx].start_time - current_time;
        current_time = p[idx].start_time;
    }

    // 타임 퀀텀보다 남은 시간이 많으면
    if (burst_remaining[idx] - tq > 0) {
        // 타임 퀀텀만큼 남은 시간 감소 후 현재 시간 갱신
        burst_remaining[idx] -= tq;
        current_time += tq;
    } else {
        // 타임 퀀텀보다 남은 시간이 적으면
        // 남은 시간만큼 현재 시간 증가
        current_time += burst_remaining[idx];
        // 남은 시간을 0 으로 설정
        burst_remaining[idx] = 0;
        // 완료된 프로세스 수 증가
        completed++;

        // 프로세스 완료 시간, 반환 시간, 대기 시간, 응답 시간을 계산
        p[idx].completion_time = current_time;
        p[idx].turnaround_time = p[idx].completion_time - p[idx].arrival_time;
        p[idx].waiting_time = p[idx].turnaround_time - p[idx].burst_time;
        p[idx].response_time = p[idx].start_time - p[idx].arrival_time;

        // 총 시간들을 계산
        total_turnaround_time += p[idx].turnaround_time;
        total_waiting_time += p[idx].waiting_time;
        total_response_time += p[idx].response_time;
    }

    // 다음에 실행할 프로세스를 큐에 추가
```

```
    for (int i = 1; i < n; i++) {
        if (burst_remaining[i] > 0 && p[i].arrival_time <= current_time && mark[i] == 0) {
            q.push(i);
            mark[i] = 1;
        }
    }

    // 남은 시간이 있는 경우, 큐에 프로세스를 다시 추가
    if (burst_remaining[idx] > 0) {
        q.push(idx);
    }

    // 큐가 빈 경우, 다음 프로세스를 찾아 추가
    if (q.empty()) {
        for (int i = 1; i < n; i++) {
            if (burst_remaining[i] > 0) {
                q.push(i);
                mark[i] = 1;
                break;
            }
        }
    }
}
```

FCFS 스케줄링의 동작에 맞게 프로세스들을 주어진 타임 퀀텀에 따라 분할하여 스케줄링하고 그에 따른 시작 시간, 완료 시간, 반환 시간, 대기 시간, 응답 시간을 구하는 로직을 구현하였다.

<원래의 프로세스 정보 입력 처리 부분>

```
for (int i = 0; i < n; i++) {
    cout << "Enter arrival time of process " << i + 1 << ": ";
    cin >> p[i].arrival_time;
    cout << "Enter burst time of process " << i + 1 << ": ";
    cin >> p[i].burst_time;
    burst_remaining[i] = p[i].burst_time;
    p[i].pid = i + 1;
    cout << endl;
}

// 도착 시간에 따라 프로세스를 정렬
sort(p, p + n, compare1);
```

<수정된 프로세스 정보 입력 처리 부분>

```
for (int i = 0; i < n; i++) {
    cout << "Enter arrival time of process " << i + 1 << ": ";
    cin >> p[i].arrival_time;
    cout << "Enter burst time of process " << i + 1 << ": ";
    cin >> p[i].burst_time;
    p[i].pid = i + 1;
    cout << endl;
}

// 도착 시간에 따라 프로세스를 정렬
sort(p, p + n, compare1);

// 실행 시간 초기화
for (int i = 0; i < n; i++) {
    burst_remaining[i] = p[i].burst_time;
}
```

주어진 참조 코드에서 남은 실행 시간 배열 (burst_remaining)을 프로세스 배열 (p)과 동시에 초기화하고 난 뒤, 도착 시간에 따라 프로세스를 정렬하는데, 이렇게 초기화가 진행될 경우

같은 i 에 대해서 초기화된 p 배열의 프로세스 정보와, 남은 실행 시간 배열의 실행 시간이 동일한 프로세스 색인에 대해서 서로 맞지 않게 되어, 제대로 된 시간 계산이 이루어지지

않는 버그가 발생한다. 따라서 해당 부분을 정렬 이후 따로 초기화함으로써, 올바르게 출력이 나올 수 있게 수정하였다.

2. Test it for the “SJF_Scheduling_Notes” in the attachment. (Test results)

FCFS

```
#P      AT      BT      ST      CT      TAT     WT      RT

1       3       1       13      14      11      10      10

2       1       4       6       10      9       5       5

3       4       2       14      16      12      10      10

4       0       6       0       6       6       0       0

5       2       3       10      13      11      8       8


Average Turnaround Time = 9.80
Average Waiting Time = 6.60
Average Response Time = 6.60
CPU Utilization = 100.00%
Throughput = 0.31 process/unit time
```

FCFS 방식은 프로세스가 도착한 순서대로 처리된다. 이 결과에서 보듯이, 평균 대기 시간과 응답 시간이 상당히 높다. 이는 FCFS가 긴 처리 시간을 가진 프로세스 때문에 다른 프로세스들이 오랫동안 기다려야 하는 '컨베이어 벨트 효과'를 가질 수 있기 때문으로 보인다.

Round Robin

```
                        [TQ: 1]
#P      AT      BT      ST      CT      TAT     WT      RT

1       3       1       5       6       3       2       2

2       1       4       1       14      13      9       0

3       4       2       7       12      8       6       3
```

```
4       0       6       0       16      16      10      0

5       2       3       3       13      11      8       1


Average Turnaround Time = 10.20
Average Waiting Time = 7.00
Average Response Time = 1.20
CPU Utilization = 100.00%
Throughput = 0.42 process/unit time
```

```
                          [TQ: 2]
#P      AT      BT      ST      CT      TAT     WT      RT

1       3       1       8       9       6       5       5

2       1       4       2       13      12      8       1

3       4       2       9       11      7       5       5

4       0       6       0       16      16      10      0

5       2       3       4       14      12      9       2


Average Turnaround Time = 10.60
Average Waiting Time = 7.40
Average Response Time = 2.60
CPU Utilization = 100.00%
Throughput = 0.45 process/unit time
```

```
                          [TQ: 3]
#P      AT      BT      ST      CT      TAT     WT      RT

1       3       1       9       10      7       6       6

2       1       4       3       16      15      11      2

3       4       2       13      15      11      9       9

4       0       6       0       13      13      7       0

5       2       3       6       9       7       4       4


Average Turnaround Time = 10.60
Average Waiting Time = 7.40
Average Response Time = 4.20
CPU Utilization = 100.00%
Throughput = 0.33 process/unit time
```

[TQ: 4]

#P	AT	BT	ST	CT	TAT	WT	RT
1	3	1	11	12	9	8	8
2	1	4	4	8	7	3	3
3	4	2	12	14	10	8	8
4	0	6	0	16	16	10	0
5	2	3	8	11	9	6	6

Average Turnaround Time = 10.20
Average Waiting Time = 7.00
Average Response Time = 5.00
CPU Utilization = 100.00%
Throughput = 0.36 process/unit time

[TQ: 5]

#P	AT	BT	ST	CT	TAT	WT	RT
1	3	1	12	13	10	9	9
2	1	4	5	9	8	4	4
3	4	2	13	15	11	9	9
4	0	6	0	16	16	10	0
5	2	3	9	12	10	7	7

Average Turnaround Time = 11.00
Average Waiting Time = 7.80
Average Response Time = 5.80
CPU Utilization = 100.00%
Throughput = 0.33 process/unit time

[TQ: 6]

#P	AT	BT	ST	CT	TAT	WT	RT
1	3	1	13	14	11	10	10
2	1	4	6	10	9	5	5

```
3       4       2       14      16      12      10      10

4       0       6       0       6       6       0       0

5       2       3       10      13      11      8       8


Average Turnaround Time = 9.80
Average Waiting Time = 6.60
Average Response Time = 6.60
CPU Utilization = 100.00%
Throughput = 0.31 process/unit time
```

- TQ가 1부터 6까지 변화하면서 평균 대기 시간, 평균 응답 시간, 평균 반환 시간, 처리량의 수치가 변화함을 알 수 있었다.
- Round Robin은 FCFS에 비해 더 빈번한 컨텍스트 스위칭을 유발한다. 이로 인해 평균 응답 시간은 FCFS보다 대체로 낮지만, 평균 대기 시간과 평균 반환 시간은 더 높은 경향을 보였다.
- TQ가 커지면 평균 응답 시간이 증가하지만, 컨텍스트 스위칭이 줄어들어 처리량이 늘어나는 경향을 보였다.

3. Impressions

이번 과제를 통해 FCFS 및 Round Robin 스케줄링 알고리즘을 구현하고 분석하면서, 운영 체제에서의 프로세스 관리와 스케줄링의 중요성에 대해 깊이 이해할 수 있었다. 특히, 다양한 타임 퀀텀 값이 Round Robin 알고리즘의 성능에 미치는 영향을 실제로 측정하고 분석하면서, 이론적 지식과 실제 적용 사이의 괴리를 좁힐 수 있었다.