C 언어를 이용한 CPU 스케줄러 구현 및 분석

데이터과학과
2024320366 최도은

< 목 차 >

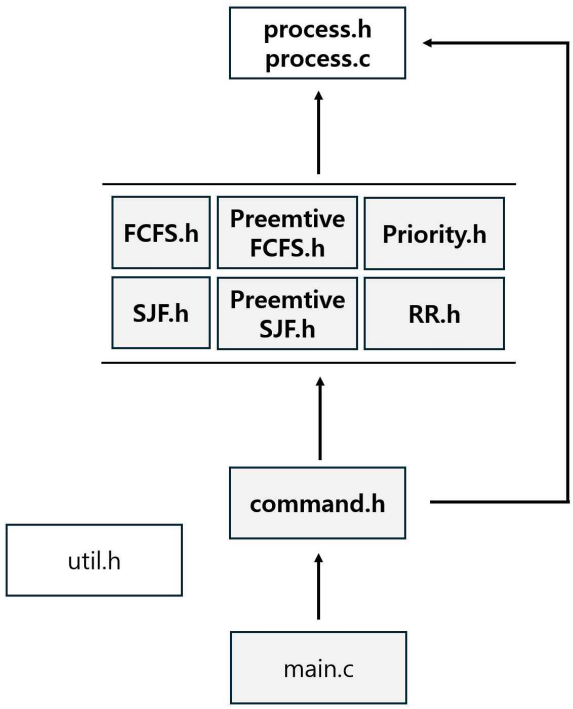
I. 서론

CPU Scheduler란 운영 체제 (OS) 내부의 소프트웨어 모듈로, 여러 개의 프로세스 혹은 스레드가 실행되어야 할 때 작업 순서를 정하는 역할을 한다. 만약 작업 순서를 잘 정하지 못하면 한정된 자원인 CPU를 고르게 분배하지 못하게 되고, 이에 따라 실행되어야 할 시기를 놓친 프로세스들이 생길 수 있다. 따라서 CPU Scheduler들은 프로세스들의 특성과 공정성 등을 고려하여 적절한 알고리즘을 사용해 프로세스들의 실행 순서를 결정해야 한다. 그 방법으로 FCFS, SJF, Priority, Round Robin 등의 스케줄링 알고리즘이 존재하며, 다양한 상황에서 장점과 단점을 갖는다.

CPU Scheduling Simulator란 이러한 알고리즘들의 동작을 실험하고 성능을 비교해 볼 수 있는 도구로, 프로세스의 처리 과정을 Gantt Chart 등으로 시각화하여 각 알고리즘이 가지는 장단점에 대한 이해를 돕는다.

본 보고서에서 구현한 CPU Scheduling Simulator는 FCFS, SJF, Priority와 Round Robin 알고리즘을 포함하며, SJF와 Priority의 경우 preemptive(선점) 옵션을 제공한다. 구성은 여타 CLI (Command Line Interface) 기반 프로그램과 유사하다. 커맨드를 통한 프로세스의 생성과 삭제를 지원하며, 동일한 케이스에 대한 여러 알고리즘 간의 성능 비교 등을 지원한다.

II. 시스템 구성



위 그림은 프로그램을 구성하는 파일 사이의 기능적 관계를 나타낸 것이다.
회색으로 표현된 파일은 독립적인 로직을 구현함을 나타낸다. 반면 흰색으로 표현된 파일은 독립적 기능보다는 자주 사용되는 보조적 연산(helper function)만을 정의해 두었음을 나타낸다.

볼드체로 표현된 파일끼리는 전역 변수들을 공유하며 프로그램에서 핵심적 역할을 담당한다.
화살표는 그 시작점에 있는 파일이 화살표가 가리키는 파일의 함수에 의존한다는 의미이다. 예컨대 command.h는 process.h의 함수를 사용하지만, 반대는 그렇지 않다.

도식을 구성하는 각 파일에 대한 설명은 아래와 같다.

1. main.c
command.h의 기능을 실행하기 위한 시작점이다.

2. command.h
사용자의 명령에 따라 각 알고리즘들의 실행과 프로세스 메모리 수정, 프로그램 종료 등 프로그램의 전체 흐름을 제어한다.

3. FCFS.h, SJF.h, ..., RR.h
6개의 헤더 파일을 통해 각 알고리즘을 구현한다.

4. process.h, process.c
실제 알고리즘을 구현하는 헤더 파일들에서 쓰이는 큐들과 변수들, 보조 연산들을 정의한다. 이외에도 프로세스 현황 및 Gantt Chart 출력 등 process 구조체에 관한 연산을 처리한다.

5. util.h
문자열의 형변환 함수 또는 split 함수와 같이, C에 기본적으로 정의되어 있지 않으나 구현할 필요성이 있는 함수들을 모아 두었다.

II.1 초기화 및 시작 과정

시뮬레이터에서 알고리즘을 실행하고 분석하기 위해서는 프로그램의 메모리에 프로세스를 추가해야 한다. 프로세스는 다음과 같은 구조체의 형식을 하고 있다.

```
struct Process {
    int PID;
    int arrivalTime;
    int CPUBurstTime;
    int IOBurstTime[6][2];
    int IOBurstTimeNumber;
    int priority;
    ...
};
```

IOBurstTime[6][2]의 2차원 배열은 CPUBurstTime이 얼마만큼 실행되었을 때 얼마만큼의 IO burst가 실행되어야 하는지를 나타낸다. 가령 {{10, 10}}의 경우 CPU burst를 10만큼 시행한 후 IO burst를 10만큼 시행한다는 의미이다.

구조체의 속성들을 초기화하고 프로그램의 메모리에 프로세스를 추가하기 위해 다음과 같이 **add 명령**을 사용할 수 있다.

```
>add
Number of processes:3
Random for 1, else 0:0
-- 1th Process Config --
PID (0 ~ 99) :1
Arrival Time:0
CPU Burst Time:8
```

[이후 생략]

add 명령으로 생성된 프로세스들은 process.h에 정의된 전역 변수 struct Process *processes[]에 저장되고 **arrival time을 기준으로 오름차순으로 정렬**된다. 이 정렬 작업은 모든 스케줄링 알고리즘에서 공통적으로 요구하는 사항이므로, 효율적인 구현을 위해 프로세스 생성 단계에서 처리해준다.

```
+-----+-----+-----+--------------------+-----+
| PID | AT  | CBT | I/O Burst Times    | Pri |
+-----+-----+-----+--------------------+-----+
| 1   | 0   | 8   | (6,4)              | 2   |
| 2   | 1   | 7   | (2,3)              | 1   |
| 3   | 2   | 7   | (1,2)              | 3   |
+-----+-----+-----+--------------------+-----+
```

[Arrival time을 기준으로 정렬된 모습]

메모리에 프로세스를 추가한 후 다음과 같이 **run 명령**을 시행하여 알고리즘을 실행할 수 있다.

```
>run fcfs
============
FCFS
============

[Gantt Chart]

1_____2_____3_____      1_____3_____2_____
0     6     8     9     10    12    18    23

Average Turnaround : 16.67
Average Waiting : 6.33
Maximum Waiting : 7.00
```

III. FCFS

FCFS(First Come, First Served) 알고리즘은 ready queue에 먼저 도착한 프로세스일수록 먼저 실행되도록 하는 알고리즘이다. CPU burst time이 큰 프로세스 뒤에 실행되는 프로세스는 waiting time이 커진다는 단점이 있지만, **모든 프로세스의 실행이 보장되고 context switching overhead가 최소**라는 장점이 있다. 이러한 장점을 살려, 비교적 종료 기한을 엄격하게 지켜야 할 필요가 없는 백그라운드 배치 작업 처리와 같은 경우에 사용된다.

본 프로젝트에서는 FCFS의 동작을 다음 의사 코드와 같이 구현하였다.

Algorithm 1: FCFS

```
   Input: struct Process *processes[] (sorted by arrival time)
   Output: queue[][3] = {{PID, endTime, ArrivalTime}, ...}
 1 readyQueue ← processes; // minheap
 2 readyQueueLen ← len(processes);
 3 endTime ← 0;
 4 i ← 0;
 5 while readyQueueLen > 0 do
 6     proc ← peek(readyQueue);
 7     if proc→arrivalTime > endTime then
           // Idle process
 8         PID, endTime ← -1, (proc→arrivalTime);
 9         goto parse_next;
10     pop(readyQueue);
11     PID ← (proc→PID);
12     endTime ← endTime + nextCPUBurstTime(proc);
13     if proc has completed IO then
14         readyQueueLen ← readyQueueLen - 1;
15     else
16         (proc→arrivalTime) ←
17           endTime + (proc→nextIOBurstTime);
18         insert proc to minheap readyQueue
19     parse_next:
20     update queue[i][3] using PID, endTime, arrivalTime
21     i ← i + 1;
```

Input | 상기한 방법으로 오름차순으로 정렬된 process 구조체 배열

Output | 프로세스들의 arrival time과 종료 시점을 저장하는 2차원 int 배열. 가령 PID 2가 10에 도착하여 20에 끝났다면, queue[i] = {2, 20, 10}와 같이 저장한다.

위 Input과 Output의 양식은 통일성을 위해 모든 스케줄링 알고리즘이 동일한 양식을 사용하도록 했다.

Line 1 | readyQueue에 processes를 copy한 후에 arrival time을 기준으로 **최소 힙 화** (min-heapify) 한다. (배열을 힙처럼 다루기 위한 함수는 process.h에 정의되어 있다.)

Line 7 | readyQueue의 제일 위에 있는 프로세스는 arrival time이 제일 작은 프로세스다. 이 arrival time이 바로 이전 프로세스의 종료 시점 (endTime)보다 크다면, 현재 실행되고 있는 프로세스는 peek(readyQueue)가 아닌 **Idle 프로세스**이다.

Line 8-9 | 따라서 Idle 상태를 나타내도록 PID ← -1을 지정한다. Idle process의 종료 시점은 바로 다음 process의 arrival time과 동일하므로, 이 점을 고려하여 endTime을 설정해 준다. 라벨 parse_next로 점프하여 현재 상태를 기록하고 readyQueue의 다음 원소를 처리한다.

Line 10-12 | 현재 실행되고 있는 프로세스가 Idle이 아닌 readyQueue의 첫 번째 원소임이 확정되면, 이에 맞춰 현재 PID와 종료 시점 endTime을 업데이트 해 준다.

Line 13 | 만약 해당 프로세스의 I/O가 모두 끝났다면, 남은 CPU burst time을 실행하고 프로세스가 끝난다. 즉 readyQueueLen을 1 감소한다.

Line 16-17 | 해당 프로세스에 아직 예정된 I/O가 있다면, I/O 시행 후 다시 돌아올 시점을 업데이트해 준다. 그리고 프로세스를 **readyQueue에 넣는다**.

여기서 readyQueue는 arrival time에 대한 최소 힙이고 FCFS는 arrival time만을 기준으로 스케줄링하기 때문에, waitingQueue를 실제로 거치지 않고도 프로세스들의 I/O를 구현할 수 있다.

II.1에서 제시된 케이스에 대한 실행 결과는 아래와 같다.

PID	AT	CBT	I/O Burst Times	Pri
1	0	8	(6,4)	2
2	1	7	(2,3)	1
3	2	7	(1,2)	3

[**II.1**에서 제시된 케이스]

```
>run fcfs
============
FCFS
============

[Gantt Chart]

1_____2_____3_____      1_____3_____2_____
0     6     8     9     10    12    18    23

Average Turnaround : 16.67
Average Waiting : 6.33
Maximum Waiting : 7.00
```

IV. SJF

SJF(Shortest Job First) 알고리즘은 ready queue에 존재하는 모든 프로세스 중 다음으로 시행될 CPU burst time이 짧은 프로세스일수록 먼저 실행되도록 하는 알고리즘이다. 평균 waiting time이 가장 낮은 알고리즘 중 하나이지만, CPU burst time이 긴 작업들은 **다른 작업에 밀려 수행되지 못할 가능성**이 존재한다 (starvation)는 단점이 있다.

Preemtive (선점형) SJF 알고리즘은 새로운 프로세스가 ready queue에 들어오거나 혹은 waiting queue에서 프로세스가 복귀할 때, 현재 실행 중인 프로세스의 잔여 시간보다 짧은 CPU burst time을 가졌다면 실행 중인 프로세스를 갈음할 수 있는 권한을 가지는 방식이다. 이런 점에서 SRJF (Shortest Remaining Job First)라고 불리기도 한다. 평균 waiting time은 이론상 최소치이지만 context switching overhead가 커지고 starvation의 가능성이 여전하다는 단점이 있다.

Ready queue에 있는 프로세스들의 실행이 보장되지 않는다는 단점이 커서 해당 알고리즘이 단독으로 사용되는 경우는 드물다. 작업을 최대한 빨리 처리해야 하는 경우 등에 제한적으로 사용되거나, queue에 오래 머무를수록 우선순위가 증가하는 aging과 같은 보완 방법을 접목하는 경우가 많다.

IV.I Non-preemtive SJF

Algorithm 2: SJF

```
   Input: struct Process *processes[] (sorted by arrival time)
   Output: queue[][3] = {{PID, endTime, ArrivalTime}, ...}
1  endTime ← 0;
2  i ← 0;
3  while runningProcesses > 0 do
4      for struct Process *p : processes ∪ waitingQueue do
5          if p→arrivalTime ≤ endTime then
6              insert p to minheap readyQueue
7      if readyQueue is empty then
           // Idle process
8          PID ← -1;
9          endTime ← min(peek(processes)→arrivalTime,
            peek(waitingQueue)→arrivalTime);
10         goto parse_next;
11     proc ← pop(readyQueue);
12     PID ← (proc→PID);
13     endTime ← endTime + nextCPUBurstTime(proc);
14     if proc has completed IO then
15         runningProcesses ← runningProcesses - 1;
16     else
17         (proc→arrivalTime) ←
18           endTime + (proc→nextIOBurstTime);
19         insert proc to minheap waitingQueue
20     parse_next:
21     update queue[i][3] using PID, endTime, arrivalTime
22     i ← i + 1;
```

waitingQueue | Arrival time을 기준으로 하는 최소 힙이다.

readyQueue | 다음 CPU burst time을 기준으로 하는 최소 힙이다.

Line 4-6 | 루프의 시작 부분에서, 현재 시간(endTime)보다 더 일찍 readyQueue에 도착했어야 하는 프로세스들을 readyQueue에 넣어 준다.
readyQueue에 프로세스가 도착하는 경우는 processes 배열에서 새로 들어오는 경우와 waitingQueue에서 복귀하는 경우가 있는데, 두 경우 모두 검사해 준다.

Line 7-10 | readyQueue가 비어 실행할 프로세스가 없으면 Idle 프로세스가 실행된다. Idle 프로세스의 종료 시간(endTime)은 이 이후 (processes 또는 waitingQueue에서) 제일 먼저 readyQueue에 도착할 프로세스의 arrrival time과 같다.

Line 11~ | 현재 프로세스가 Idle이 아님이 확정된다. FCFS에서와 유사하게 I/O 처리를 해 준다. I/O를 모두 수행한 프로세스에 대해서는 runningProcesses를 1 감소, I/O를 수행해야 할 프로세스는 waitingQueue에 삽입한다.

II.1에서 제시된 케이스에 대한 실행 결과는 아래와 같다.

```
>run sjf
============
SJF
============

[Gantt Chart]

1_____3_____2_____3_____1_____2_____
0     6     7     9     15    17    22

Average Turnaround : 17.00
Average Waiting : 6.67
Maximum Waiting : 6.00
```

IV.2 Preemtive SJF

Algorithm 3: Preemtive SJF

Input: struct Process *processes[] (sorted by arrival time)
Output: queue[][3] = {{PID, endTime, ArrivalTime}, ...}

```
1  preempted ← NULL;
2  waitingQueue ← processes;
3  endTime ← 0;
4  i ← 0;
5  while runningProcesses > 0 do
6      if readyQueue is empty then
           // Idle process
7          PID ← -1;
8          firstArrival ← (peek(waitingQueue)→arrivalTime);
9          while peek(waitingQueue)→arrivalTime == firstArrival
            do
10             insert pop(waitingQueue) to readyQueue
11         if i ≠ 0 then
12             goto parse_next;
13     proc ← preempted ? preempted : pop(readyQueue);
14     PID ← (proc→PID);
15     expectedEnd ← (end→nextCPUBurst(proc));
16     while waitingQueue is not empty do
17         candidateArrival ← (peek(waitingQueue)→arrivalTime);
18         if candidateArrival > expectedEnd then
19             break;
20         if nextCPUBurst(peek(waitingQueue)) < expectedEnd -
            candidateArrival then
               // Preempt
21             preempted←pop(waitingQueue);
22             (proc→arrivalTime)←candidateArrival;
23             insert proc into readyQueue;
24         else
25             insert pop(waitingQueue) into readyQueue;
26     if preempted then
27         goto parse_next;
28     if proc has completed IO then
29         runningProcesses ← runningProcesses - 1;
30     else
31         (proc→arrivalTime) ←
32           endTime + (proc→nextIOBurstTime);
33         insert proc to minheap waitingQueue
34     parse_next:
35     update queue[i][3] using PID, endTime, arrivalTime
36     i ← i + 1;
```

preemmpted | 바로 전 루프에서 어떤 값으로 preempt가 되었는지 나타낸다. preempt되지 않았으면 NULL값을 가진다.

readyQueue | 다음 CPU burst time을 기준으로 하는 최소 힙이다.

waitingQueue | Arrival time을 기준으로 하는 최소 힙이다.

Line 1 | waitingQueue에 processes를 복사한다. 프로세스가 processes로부터 readyQueue에 도착하는지 waitingQueue로부터 도착하는지는 알고리즘의 시뮬레이션에 영향을 주지 않고, 오히려 양쪽 배열을 모두 고려하는 경우가 번거로우므로 waitingQueue 쪽만 고려할 수 있도록 해 주었다.
waitingQueue는 arrival time 기준의 최소 힙이고 processes는 arrival time 기준 오름차순 정렬이 되어 있으므로, 복사 후 별도의 과정을 거치지 않아도 waitingQueue는 최소 힙으로서 정상적으로 동작한다.

Line 6-10 | readyQueue가 비었을 때 Idle process를 생성하고 readyQueue를 채우는 과정이다. waitingQueue에서 arrival time이 최소인 원소들을 모두 readyQueue에 넣어 준다.

Line 11-12 | 첫 실행 시에는 readyQueue가 비어 있을 수밖에 없으므로, 이때는 Idle process를 만들지 않는다.

Line 13 | 실행될 프로세스를 정하는 과정이다. 직전 루프에서 현재 실행되는 프로세스를 preempt하기로 결정한 경우, 그 프로세스가 실행된다. 그렇지 않은 경우, 다음 CPU burst가 가장 짧은 프로세스가 실행된다.

Line 15 | 변수 expectedEnd는 실행 중인 프로세스가 중간에 선점당하지 않고 현재 CPU burst를 마쳤을 때의 시간을 나타낸다.

Line 16-19 | peek(waitingQueue) 요소가 현재 작업을 선점 가능한지 검사한다. 해당 요소의 도착 시간이 현재 작업이 끝난 후라면 선점이 불가능하다. 이때 waitingQueue는 arrival time에 대한 최소 힙이므로, waitingQueue의 첫 번째 요소가 현재 작업 종료 이후에 도착한다면 나머지 모든 요소들도 그러하다. 즉 이후 경우는 더 고려할 필요가 없으므로 break 문을

사용해 반복을 빠져나온다.

Line 20-25 | peek(waitingQueue) 요소가 현재 작업 종료 전에 도착하는 상황에서, 선점 가능한지 검사한다. 해당 요소의 다음 CPU burst가 현재 작업의 잔여 CPU burst보다 작다면 선점 가능하다.

만약 선점한다면, preempted 변수에 pop(waitingQueue)를 저장하여 다음 루프를 돌 때 무조건 해당 프로세스가 실행되도록 한다. 선점당한 현재 프로세스는 readyQueue에서 대기한다.

만약 선점하지 못한다면, **해당 요소는 readyQueue로 옮겨간다**. line 16-19에 의해 해당 요소가 현재 작업 종료 이전에 도착함이 보장되기 때문이다.

Line 26-27 | 만약 선점당했으면 실행 중이던 프로세스는 실행이 중단되고 곧바로 다음 프로세스가 실행된다.

Line 28-33 | SJF에서와 같은 방법으로 프로세스들의 I/O를 처리해 준다.

II.1에서 제시된 케이스에 대한 실행 결과는 아래와 같다.

```
>run sjf -p
============
Preemtive SJF
============

[Gantt Chart]

1_____2_____3_____1_____2_____1_____3_____
0     1     3     4     9     14    16    22

Average Turnaround : 16.33
Average Waiting : 6.00
Maximum Waiting : 10.00
```

V. Priority

Priority 스케줄링 알고리즘은 사용자가 직접 작업들의 우선순위를 설정할 수 있는 알고리즘이다. 상황에 따라 유연하게 우선순위를 조정할 수 있지만, 우선순위가 낮은 프로세스들은 우선순위가 높은 프로세스들에 밀려 실행되지 못할 수도 있다 (starvation). 단점 보완을 위해 aging 또는 round robin 등과 접목하여 쓰이기도 한다.

V.1 Non-preemtive Priority

Algorithm 4: Priority

Input: struct Process *processes[] (sorted by arrival time)
Output: queue[][3] = {{PID, endTime, ArrivalTime}, ...}

```
1  endTime ← 0;
2  i ← 0;
3  while runningProcesses > 0 do
4      for struct Process *p : processes ∪ waitingQueue do
5          if p→arrivalTime ≤ endTime then
6              insert p to minheap readyQueue
7      if readyQueue is empty then
           // Idle process
8          PID ← -1;
9          endTime ← min(peek(processes)→arrivalTime,
            peek(waitingQueue)→arrivalTime);
10         goto parse_next;
11     proc ← pop(readyQueue);
12     PID ← (proc→PID);
13     endTime ← endTime + nextCPUBurstTime(proc);
14     if proc has completed IO then
15         runningProcesses ← runningProcesses - 1;
16     else
17         (proc→arrivalTime) ←
18           endTime + (proc→nextIOBurstTime);
19         insert proc to minheap waitingQueue
20     parse_next:
21     update queue[i][3] using PID, endTime, arrivalTime
22     i ← i + 1;
```

의사 코드는 SJF와 동일하다. 이것은 SJF가 남은 시간을 우선순위로 하는 Priority 스케줄링과 동일하기 때문이다.

따라서 우선순위를 조정하는 **최소 힙 readyQueue의 구성 방식**이 SJF와 Priority 스케줄링의 차이점이다. SJF에서는 다음 CPU burst time을 기준으로 했지만, Priority 스케줄링에서는 Priority를 기준으로 한다.

II.1에서 제시된 케이스에 대한 실행 결과는 아래와 같다.

```
>run priority
============
Priority
============

[Gantt Chart]

1_____2_____3_____      1_____2_____3_____
0     6     8     9     10    12    17    23

Average Turnaround : 16.33
Average Waiting : 6.00
Maximum Waiting : 6.00
```

V.2 Preemtive Priority

Algorithm 5: Preemtive Priority

Input: struct Process *processes[] (sorted by arrival time)
Output: queue[][3] = {{PID, endTime, ArrivalTime}, ...}

```
1  preempted ← NULL;
2  waitingQueue ← processes;
3  endTime ← 0;
4  i ← 0;
5  while runningProcesses > 0 do
6      if readyQueue is empty then
           // Idle process
7          PID ← -1;
8          firstArrival ← (peek(waitingQueue)→arrivalTime);
9          while peek(waitingQueue)→arrivalTime == firstArrival
            do
10             insert pop(waitingQueue) to readyQueue
11         if i ≠ 0 then
12             goto parse_next;
13     proc ← preempted ? preempted : pop(readyQueue);
14     PID ← (proc→PID);
15     expectedEnd ← (end→nextCPUBurst(proc));
16     while waitingQueue is not empty do
17         candidateArrival ← (peek(waitingQueue)→arrivalTime);
18         if candidateArrival > expectedEnd then
19             break;
20         if peek(waitingQueue)→priority < proc→priority then
               // Preempt
21             preempted←pop(waitingQueue);
22             (proc→)←candidateArrival;
23             insert proc into readyQueue;
24         else
25             insert pop(waitingQueue) into readyQueue;
26     if preempted then
27         goto parse_next;
28     if proc has completed IO then
29         runningProcesses ← runningProcesses - 1;
30     else
31         (proc→arrivalTime) ←
32           endTime + (proc→nextIOBurstTime);
33         insert proc to minheap waitingQueue
34     parse_next:
35     update queue[i][3] using PID, endTime, arrivalTime
36     i ← i + 1;
```

위와 마찬가지로 의사 코드는 Preemtive SJF와 거의 동일하다. Preemtive SJF는 남은 시간이 우선순위인 Preemtive Priority와 같기 때문이다. 한편 이 경우에서는 readyQueue의 구성 방식뿐 아니라 **프로세스가 선점이 가능할 조건**을 정의하는 부분 역시 차이가 있다.

Line 20 | Preemtive SJF가 선점 허용 조건을 '선점하고자 하는 프로세스의 다음 CPU burst time은 실행되고 있는 프로세스의 남은 CPU burst time보다 작을 것'으로 규정하고 있다면, Preemtive Priority는 '선점하고자 하는 프로세스의 priority가 현재 실행 중인 프로세스의 priority보다 높을 것'으로 규정하고 있음을 알 수 있다.

II.1에서 제시된 케이스에 대한 실행 결과는 아래와 같다.

```
>run priority -p
============
Preemtive Priority
============

[Gantt Chart]

1_____2_____1_____2_____1_____3_____      3_____1_____3_____
0     1     3     6     11    13    14    16    17    19    24

Average Turnaround : 17.00
Average Waiting : 6.67
Maximum Waiting : 11.00
```

VI. Round Robin

Round Robin 알고리즘은 프로세스들 간의 우선순위를 두지 않는 선점형 스케줄링 방식이다. Ready queue의 첫 번째 원소에 일정 시간 할당량(time quantum)을 제공하고, 해당 시간 내에 프로세스가 종료되지 못할 경우 ready queue의 제일 끝으로 보내는 순환 큐(round queue)를 이용한다.

모든 프로세스들이 돌아가면서 공정하게 CPU를 할당받을 수 있으므로, 평균 응답 시간 측면에서 효율적이며 starvation 문제가 해결된다. 알고리즘의 효율은 time quantum의 설정에 큰 영향을 받는데, 너무 작게 설정하면 context switching overhead가 커지고 너무 크게 설정하면 알고리즘은 FCFS와 동일해진다. 따라서 프로세스의 특성에 따라 time quantum을 적절하게 조절하는 Multilevel Feedback Queue 등과 같은 방법들이 접목되고 있다.

아래 의사 코드의 구현에서, Round Robin 알고리즘은 processes 배열과 함께 timeQuantum을 input으로 제공받는다.

Line 3-4 | 작업을 시작하기 전, 시간 0에 도착하는 프로세스들을 readyQueue에 넣어 준다.

Line 8-11 | readyQueue가 비었을 때 Idle 프로세스를 만들어 처리해 준다.

Line 14-17 | 만약 현재 실행하려고 하는 프로세스의 다음 CPU burst time이 timeQuantum을 넘는다면,

Algorithm 6: Round Robin

Input: struct Process *processes[] (sorted by arrival time), **timeQuantum**
Output: queue[][3] = {{PID, endTime, ArrivalTime}, ...}

```
1  endTime ← 0;
2  i ← 0;
3  while processes && peek(processes)→arrivalTime == 0 do
4      insert pop(processes) into readyQueue
5  while runningProcesses > 0 do
6      done ← False;
7      requestIO ← False;
8      if readyQueue is empty then
           // Idle process
9          PID ← -1;
10         endTime ← min(peek(processes)→arrivalTime,
            peek(waitingQueue)→arrivalTime);
11         goto parse_next;
12     proc ← pop(readyQueue);
13     PID ← (proc→PID);
14     if nextCPUBurst(proc) > timeQuantum then
15         endTime ← endTime + timeQuantum;
16         (proc→arrivalTime) ← endTIme;
17         goto parse_next;
18     endTime ← nextCPUBurstTime(proc);
19     if proc has completed IO then
20         runningProcesses ← runningProcesses - 1;
21         done←True;
22     else
23         (proc→arrivalTime) ←
24           endTime + (proc→nextIOBurstTime);
25         requestIO←True;
26     parse_next:
27     for struct Process *p : processes ∪ waitingQueue do
28         if p→arrivalTime ≤ endTime then
29             insert p to minheap readyQueue
30     if requestIO then
31         insert proc to minheap waitingQueue
32     else
33         if !done && PID ≠ -1 then
34             insert proc to readyQueue to the last
35     update queue[i][3] using PID, endTime, arrivalTime
36     i ← i + 1;
```

프로세스는 timeQuantum만큼만 실행되고 readyQueue로 돌아가며 다른 프로세스에게 CPU를 내준다.

Line 18-25 | 만약 현재 실행하려고 하는 프로세스의 다음 CPU burst time이 timeQuantum을 넘지 않는다면, I/O 작업을 처리해 준다.
만약 프로세스가 모든 I/O를 마쳤다면 done 플래그를 이용해 프로세스를 더 이상 readyQueue에 추가하지 않을 것을 표시한다.
만약 프로세스가 남은 I/O 작업이 있다면 requestIO 플래그를 이용해 프로세스가 waitingQueue에 추가되어야 함을 표시한다.

Line 26-29 | 현재 프로세스가 실행되는 동안 readyQueue에 도착한 프로세스들을 업데이트해 준다.

Line 30- | requestIO와 done 플래그를 처리해 주고, 현재 프로세스 실행 정보를 기록한다.

Round Robin의 구현에서는 위에서 구현한 다른 알고리즘들과 달리, I/O request가 발생하는 경우와 프로세스가 종료되었을 때의 경우 등을 감지 즉시 바로 처리하지 않고 **플래그를 만들어 루프의 끝에서 처리**해 주었다. 다른 알고리즘들은 readyQueue가 힙의 형태로 구현되어 삽입 순서가 중요하지 않지만, Round Robin의 경우에는 순서가 있는 linear한 구조로 구현되어야 하기 때문이다.

II.1에서 제시된 케이스에 대한 실행 결과는 아래와 같다.

```
>run rr 3
============
RR (T = 3)
============

[Gantt Chart]

1_____2_____3_____1_____2_____3_____2_____1_____3_____
0     3     5     6     9     12    15    17    19    22

Average Turnaround : 18.33
Average Waiting : 8.00
Maximum Waiting : 4.00
```

VII. 알고리즘 간 성능 비교

아래 표는 **compare 명령**을 사용하여 각 알고리즘에 대한 **평균** turnaround time, waiting time과 max waiting time을 비교한 결과이다. 알고리즘이 실행할 프로세스들은 add 명령을 통해 무작위 생성하였으며, 개수를 N=10 부터 N=40 까지 10씩 늘려 가며 각 경우를 테스트하였다.

스케줄링 알고리즘의 성능을 비교할 때, turnaround time이 짧은 정도를 나타내는 **효율성**과 max waiting time이 짧은 정도를 나타내는 **공정성**을 고려할 수 있다. 이때 모든 경우에서

$$\text{FCFS} \fallingdotseq \text{RR} < \text{SJF} \lesssim \text{Preemtive SJF}$$

의 양상을 보이며, 공정성은

$$\text{Preemtive SJF} \lesssim \text{SJF} < \text{FCFS} << \text{RR}$$

의 양상을 보임을 확인할 수 있다.

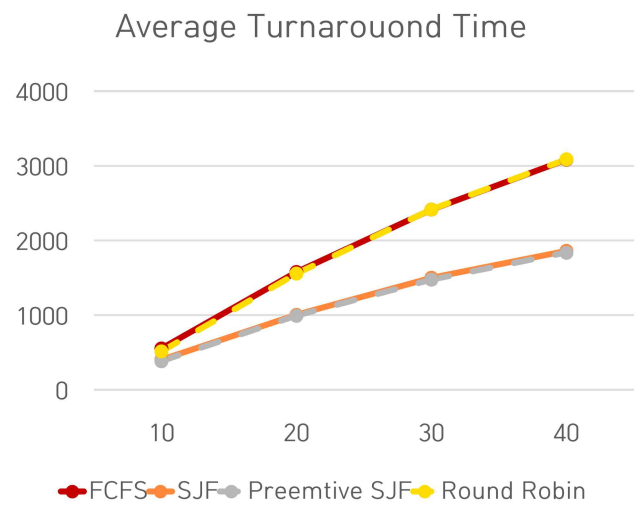
[Algorithm Compare View]

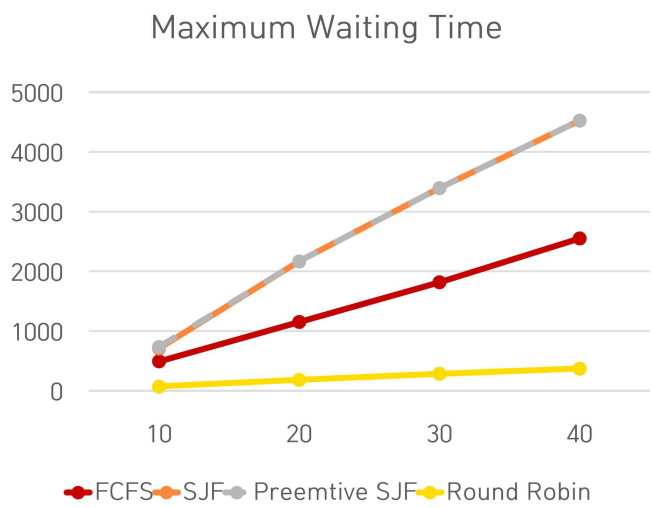
Algorithm	Turnaround	Waiting	Max. Waiting	Turnaround	Waiting	Max. Waiting
FCFS	556.00	368.00	495.00	1580.50	1338.50	1155.00
SJF	404.00	216.00	710.00	1007.00	765.00	2170.00
Preemtive SJF	385.00	197.00	735.00	994.25	752.25	2170.00
Priority	528.50	340.50	475.00	1268.75	1026.75	2225.00
Preemtive Priority	512.50	324.50	525.00	1248.00	1006.00	2225.00
Round Robin (T = 10)	517.50	329.50	75.00	1558.25	1316.25	185.00

[N = 10] [N = 20]

Algorithm	Turnaround	Waiting	Max. Waiting	Turnaround	Waiting	Max. Waiting
FCFS	2416.50	2183.00	1820.00	3085.62	2865.75	2555.00
SJF	1501.33	1267.83	3395.00	1862.50	1642.62	4525.00
Preemtive SJF	1476.50	1243.00	3395.00	1839.12	1619.25	4525.00
Priority	1907.50	1674.00	3465.00	2392.12	2172.25	4550.00
Preemtive Priority	1866.33	1632.83	3490.00	2460.38	2240.50	4550.00
Round Robin (T = 10)	2418.83	2185.33	285.00	3088.25	2868.38	375.00

[N = 30] [N = 40]





프로세스의 개수에 따른 **효율성과 공정성의 변화 폭**도 주목할 만하다. FCFS와 RR은 SJF 계열보다 프로세스 개수 증가에 따른 효율성의 감소 폭이 크다. 프로세스가 많아질수록 사이에 CPU burst가 긴 작업이 섞여 있을 확률이 증가하고, 따라서 convoy effect의 영향을 더 받을 확률이 증가하기 때문이다.

한편 공정성의 변화 측면에서는 RR이 가장 변화 폭이 작고 FCFS가 그 다음이며, SJF 계열이 가장 큰 폭으로 변화한다. RR의 max waiting time은 오로지 time quantum과 process의 개수에 비례하므로 합리적인 time quantum을 정의한다면 최대의 공정성이 보장된다. 한편 SJF 계열의 경우, 프로세스 개수가 증가할수록 초반에 CPU burst time이 큰 프로세스가 많이 할당될 가능성이 높아지고 starvation의 발생 확률과 강도 역시 높아진다. 따라서 starvation이 발생하지 않음이 보장되는 FCFS보다 공정성의 감소 폭이 큼을 알 수 있다.

Priority 알고리즘과 같이 스케줄링 과정에서 유저에 따른 임의성이 포함되는 알고리즘은 분석에서 제외하였다.

VIII. 결론

CPU Scheduling Simulator의 목적은 사용자로 하여금 다양한 상황에서 다양한 알고리즘을 모의로 실행, 분석, 비교할 수 있게 하는 것이다. 본 시뮬레이터에서는 위와 같이 총 6개의 스케줄링 알고리즘(FCFS, SJF[Preemtive], Priority[Preemtive], RR)과 그 분석을 구현하였으며, 명령어를 지원하는 CLI 인터페이스를 통해 실험 상황의 관리를 용이하게 했다.

한편 본문에서 각 알고리즘들의 단점과 그 보완법(Aging, Multilevel Queue)을 소개하였으나, 해당 기능은 프로젝트에 구현되어 있지 않다. 단점 보완 전후의 성능을 비교할 수 있는 기능을 추가하는 방향으로의 발전을 고려해 볼 수 있다.

본론에서 소개한 각 알고리즘들의 단점을 보완하는 방법 (Aging이나 Multilevel Queue 등)을 실제로 구현하여, 보완 전후의 성능 차이를 비교하는 기능을 지원할 수 있다.

개인적으로 Python과 같은 고수준 언어에 익숙해져 있어서, C와 같이 비교적 저수준이고 절차적인 프로그래밍 언어를 사용해 긴 코드를 짜 본 경험이 드물었다. 그래서 이번 프로젝트를 진행하면서 특히 파일 구조나 함수의 input/output 디자인 등을 어떻게 구성해야 할지 확신이 서지 못해 유사한 경우를 많이 찾아봤어야 했다. 그러면서 소프트웨어 디자인이나 인터페이스 설계에 관한 내용들을 많이 알아갈 수 있었다.

부록 1 - 명령어 일람

명령어 일람과 소스 코드는 아래 GItHub repository에 접속하여 확인할 수 있다.
https://github.com/choidoeun2005/251RCOSE34102/tree/master/CPUSchedulingSimulator

command	subcommand	arguments	options	description
add	-			Adds a new process to memory through an interactive dialogue sequence. Type -1 any time to suspend.
remove	-	<PID>	-	Removes the specified process from memory.
	-	-	-a	Removes all processes from the memory.
run	-	<FCFS|SJF|PRIORITY>	[-p]	Runs the CPU scheduler with the chosen algorithm. -p enables preemptive mode.
		RR, [time_quantum]	-	Runs the CPU scheduler with the Round Robin algorithm with the specified time quantum (default 10).
	-		-a	Runs the CPU scheduler with all possible algorithms (FCFS, SJF, PRIORITY, RR). Runs the preemtive mode as well if possible.
compare	add	<FCFS|SJF|PRIORITY>	[-p]	Adds or removes an algorithm to/from the compare view. [-p] option to enable preemtive mode (if possible).
		RR, [time_quantum]		Adds removes Round Robin algorithm to the compare view. Time quantum can be set by the argument. (Default 10)
		-	-a	Adds all algorithms to the compare view. Round Robin is added with the default time quantum 10.
	clear	-		Removes all algorithms from the compare view.
	-			Displays compare view (up to 10 algorithms).
list	-			Show all the processes in memory.
exit	-			Exit the CPU scheduler.
help	-			Displays command list.