

# **2019 봄학기 운영체제**

## **CPU Scheduling Simulator 보고서**

컴퓨터학과 2017320233 김채령

제출일: 2019.06.10.월

교수님: 유현창 교수님

## <목차>

### I . CPU scheduler와 CPU scheduling algorithms 소개

1장. CPU scheduling과 CPU scheduler

### II. CPU scheduling simulator의 구현

2장. 기존 CPU scheduling simulator

3장. 직접 설계한 CPU scheduling simulator

4장. CPU scheduling simulator 실행 결과

### III. 알고리즘 간 비교 분석 및 소감

5장. 알고리즘 간의 evaluation 비교 분석

6장. 프로젝트 소감 및 추후 발전 계획

### 부록

- 참고 문헌

# I . CPU scheduler와 CPU scheduling algorithms 소개

## 1장. CPU scheduling과 CPU scheduler

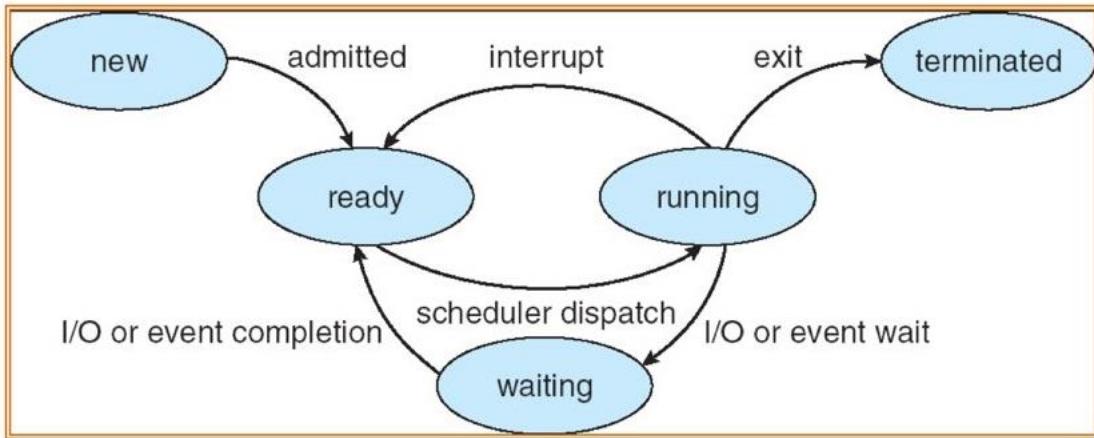
여러 프로세스에 의해 공유되는 컴퓨팅 자원에 대한 Scheduling은 OS의 기본적인 기능이다. 그리고 CPU는 주요한 컴퓨팅 자원이므로 CPU를 어떤 프로세스가 사용할 것인가에 대한 할당이 이루어져야 한다. Single processor의 경우, CPU 자원을 사용하고자 하는 프로세스들 중 하나를 선택해서 CPU를 할당해주는 것을 CPU scheduling이라고 하며 이를 담당하는 모듈을 CPU scheduler라고 한다. Multi 혹은 many processor 환경이라면 processor간의, 또 한 processor 내부에서 프로세스들에 대한 scheduling 등 추가적으로 고려해줘야 할 부분들이 많다. 하지만 CPU scheduling simulator를 본 과제에서 구현할 때 single processor 환경을 가정했기 때문에 보고서 역시 single processor 환경을 기반으로 한다. 1장은 CPU scheduling과 scheduler에 대한 기본적인 개념을 포괄한다.

### 1-1. Multi-programmed OS와 CPU scheduling

Multi-programming은 한 메모리에 여러 개의 프로세스를 로딩하는 것을 지원한다. 이것은 CPU 활용을 최대화하기 위한 목적을 지니며, 최대한 CPU가 idle하지 않고 실행 중인 프로세스를 가지도록 한다. 예를 들어, CPU를 할당 받아 실행 중이던 프로세스가 IO를 처리하는 동안 가지고 있던 CPU를 놓게 되면 CPU는 idle하게 되는데, 이 공백을 활용하기 위해 여러 개의 프로세스가 있는 메모리에서 CPU scheduling algorithm에 따라 한 프로세스를 선택해서 CPU를 할당한다. 이때 CPU scheduling이 필요하게 되는데, 즉 한 메모리에 있는 여러 개의 실행 준비가 된 프로세스들 중에서 하나의 프로세스를 골라 CPU를 그 프로세스에게 할당해주는 것을 CPU scheduling이라고 하고 CPU scheduler가 이를 담당한다.

### 1-2. CPU scheduling이 일어날 수 있는 시점

CPU scheduling이 언제 일어나야 하는지는 프로세스의 상태의 관점에서 설명할 수 있는데 아래의 도식은 프로세스의 상태에 대한 도표이다.



간단히 프로세스의 상태에 대해서 설명하자면, 먼저 new는 대기하고 있는 프로세스의 상태이며 위에서 언급한 multi-programmed OS의 job scheduler 모듈이 new queue의 프로세스들 중 일부를 메모리(ready queue)로 로딩해오면 CPU를 할당 받아 실행할 준비가 된 ready 상태가 된다. 이때 job scheduler는 본 보고서의 중점이 되는 CPU scheduler와 다른 것으로, 상대적으로 scheduling이 가끔 발생한다고 해서 long term scheduler라고도 부른다. Ready queue에 있는 프로세스들 중 한 프로세스가 CPU를 할당 받으면 running, 즉 CPU 작업을 수행하는 상태가 된다. 이때, 여러가지 interrupt에 의해서 작업을 완료하지 못하고 다시 ready queue로 돌아갈 수도 있으며, IO 처리를 해줘야 하는 경우는 waiting queue로 갔다가 IO가 끝나면 다시 ready queue로 돌아가서 남은 작업을 처리하기 위해 CPU 할당을 기다릴 수도 있다. CPU를 할당 받아 CPU 작업을 다 처리하게 되면 그 프로세스는 종료를 뜻하는 terminated 상태가 된다.

프로세스의 상태가 변함에 따라 CPU scheduling 결정이 필요할 때가 있는데, 다음의 4가지 경우가 있다.

- 프로세스가 running 상태에서 waiting 상태로 바뀔 때
- 프로세스가 running 상태에서 ready 상태로 바뀔 때
- 프로세스가 waiting 상태에서 ready 상태로 바뀔 때
- 프로세스가 terminate(종료)할 때

프로세스의 상태가 running에서 waiting으로 가는 경우와 프로세스가 종료하는 경우에는 ready queue에는 변화가 없고, CPU가 idle해지기 때문에 CPU를 다른 프로세스에게 할당해주기 위해 프로세스를 선택하는 CPU scheduling이 필요해진다. Ready queue에 변화를 주는 두 경우, 프로세스가 running 상태에서 ready 상태로 혹은 waiting 상태에서 ready 상태로 변할 때에는 CPU를 할당 받고자 기다리는 프로세스 리스트가 달라지기 때문에 CPU scheduling algorithm의 정책에 따라 현재 수행 중인 프로세스가 preempt(CPU를 뺏김)될 수도 있다.

### 1-3. CPU scheduling algorithms

CPU scheduler는 어떤 프로세스에게 언제, 얼마동안 CPU를 할당할 지 결정하기 위해 CPU scheduling algorithm을 사용한다. 그리고 각 알고리즘 내부에서도 특정한 조건이나 상황에 대한 정책이 상이할 수 있다. 다음은 본 과제에서 구현한 CPU scheduling algorithm들이며 구현할 때 고려한 자세한 정책들은 3장으로 미뤄두고, 여기서는 알고리즘별로 일반적인 작동 원리와 특징을 다루기로 한다.

#### (1) First-Come, First-Served (FCFS)

선착순과 비슷한 개념으로, FIFO(First-In, First-Out) 방식의 알고리즘이다. 각 프로세스들은 그들이 request를 제출한, 본 과제에서는 new에서 ready queue에 도달한 arrival time 순서로 CPU를 배정받는다. 기본적으로 non-preemptive 방식으로 한 프로세스가 CPU를 사용하고 있다면, 이 프로세스가 waiting 상태가 되거나 작업을 완료할 때까지 다른 프로세스가 CPU를 할당 받을 수 없다.

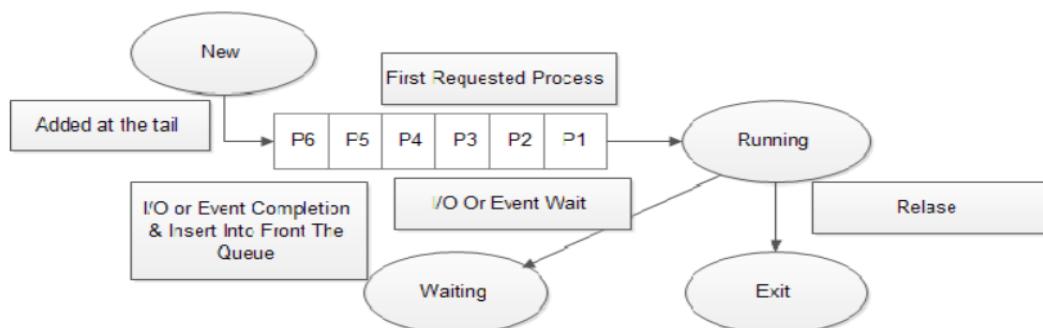
##### (1)-① FCFS 알고리즘의 장점

CPU를 사용하여 수행하겠다는 request를 제출한 순서대로 프로세스 간의 순서를 정해주면 되기 때문에 구현이 매우 간단하다며, 특정 프로세스가 자신보다 priority가 높은 프로세스들에 의해서 우선순위가 밀려서 CPU를 할당 받지 못하고 계속 대기해야 하는 starvation (indefinite blocking) 문제가 발생하지 않는다.

##### (1)-② FCFS 알고리즘의 단점

프로세스가 CPU 사용에 대한 request를 제출한 시점(순서)만을 CPU scheduling에서 고려하기 때문에 프로세스들의 작업 시간 등의 다른 요소는 FCFS 알고리즘에서 배제된다. 그래서 CPU를 사용하여 작업을 수행하는 시간이 짧은 프로세스들의 앞에 먼저 수행 중인 긴 프로세스가 있을 경우 convoy effect 문제가 발생할 수 있다. 그렇게 되면 짧은 프로세스들의 waiting time이 늘어나서 AWT가 커질 수 있다.

##### (1)-③ FCFS 알고리즘의 flow chart



## (2) Shortest-Job-First (SJF)

가장 적은 next CPU burst time이 남은 프로세스에게 우선권을 주어 CPU를 할당하는 알고리즘으로 CPU를 사용하여 작업을 수행하는데 걸리는 시간이 가장 적게 남은 프로세스를 선택하므로 shortest-next-CPU-burst 알고리즘이라고도 한다. Non-preemptive와 preemptive로 2 가지 방식에 따라 구현이 달라지는데 전자는 비선점 방식, 후자는 선점 방식으로 이해할 수 있다. Non-preemptive 방식의 경우 현재 CPU를 사용하여 수행하고 있는 프로세스가 있으면, 이 프로세스가 자신의 CPU burst를 끝내도록 보장해주는데, 즉 이 프로세스가 수행 중일 때 다른 프로세스가 next CPU burst time이 더 적게 남았더라도 수행할 수 없다. 수행 중인 프로세스가 작업을 완료하거나 IO를 처리하려 waiting 상태가 되어서 CPU가 idle해질 때까지 해당 프로세스가 CPU를 사용하도록 보장해 준다. 반면에 preemptive 방식은 현재 수행 중인 프로세스보다 next CPU burst time이 더 적게 남아서 더 높은 우선권을 가지는 프로세스가 있다면, 현재 수행 중인 프로세스를 ready 상태로 바꿔주고 우선권이 높은 프로세스에게 CPU를 할당한다. 따라서 shortest-remaining-time-first 알고리즘이라고 하기도 한다.

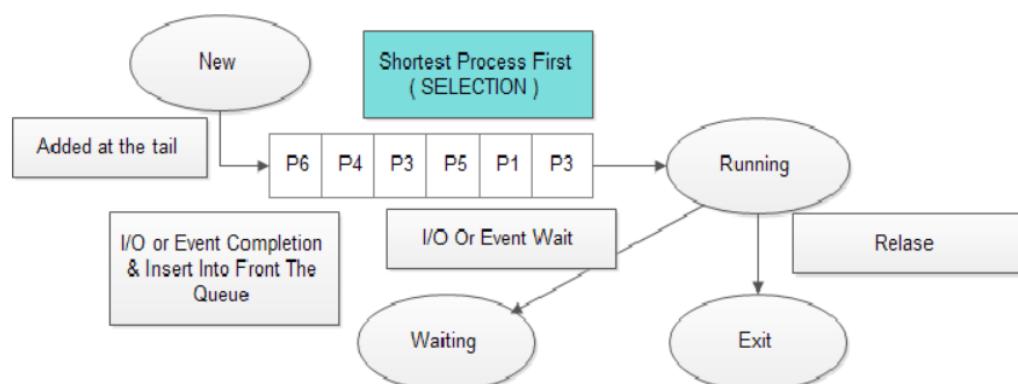
### (2)-① SJF 알고리즘의 장점

주어진 프로세스들에 대해서 최소한의 AWT를 가진다는 점에서 optimal한 알고리즘으로 알려져 있다.

### (2)-② SJF 알고리즘의 단점

보통 SJF 알고리즘은 이론적으로 simulation에서만 가능하다고 하는데 그 이유는 future knowledge를 필요로 하기 때문이다. 다시 말해서, next CPU request의 길이를 아는 것이 어렵기 때문이다. 따라서 부정확하지만 각 프로세스마다 recent history와 past history의 exponential average를 구해서 next CPU burst 값을 예측하여 사용할 수도 있다.

### (2)-③ SJF 알고리즘의 flow chart



위의 flow chart에서 preemptive 방식의 경우에는 ready queue에 변화가 있는지 매 시점에 확인을 해서 running 중인 프로세스가 있더라도 그 프로세스보다 next CPU burst time이

적게 남은 프로세스가 있다면, 해당 프로세스에게 CPU를 할당하고 원래 수행 중이던 프로세스를 ready queue로 넣어주는 것으로 해석하면 된다. Non-preemptive의 경우 수행 중인 즉, running 상태의 프로세스가 없을 경우 next CPU burst time이 가장 적게 남은 프로세스를 ready queue에서 골라 CPU를 할당하는 것으로 생각할 수 있다.

### (3) Priority

Priority는 말 그대로 우선순위가 가장 높은 프로세스에게 CPU를 할당하는 방식이며, 여기서 priority는 각 프로세스에게 수치화해서 보통 정수로 부여된다. 이때 가장 높은 우선순위를 가진 프로세스가 2개 이상이라면, 여러가지 기준을 통해 tie break할 수 있는데 통상적으로 FCFS 순서대로 arrival time을 기준으로 schedule된다. 여기서 각 프로세스에게 할당되는 priority는 숫자가 클수록 우선순위가 높을 수도 있고 작을수록 우선순위가 높다고 볼 수도 있는데, 이는 시스템에 따라 상이하다. 또 priority는 내부적으로 혹은 외부적으로 정의될 수 있는데, 내부적인 priority는 우선순위를 계산하기 위해서 요구되는 메모리 크기, average IO burst와 average CPU burst의 비율, time limit등의 측정가능한 수량을 사용한다. 외부적인 priority는 OS 외부에서 설정된 기준을 따르는데 프로세스의 중요도 등을 고려할 수 있다. Priority 알고리즘도 preemptive, non-preemptive 방식 모두 구현이 가능하며 앞서 언급한 SJF 알고리즘의 경우 next CPU burst time이 짧을수록 높은 priority를 가지는 것으로 볼 수도 있다.

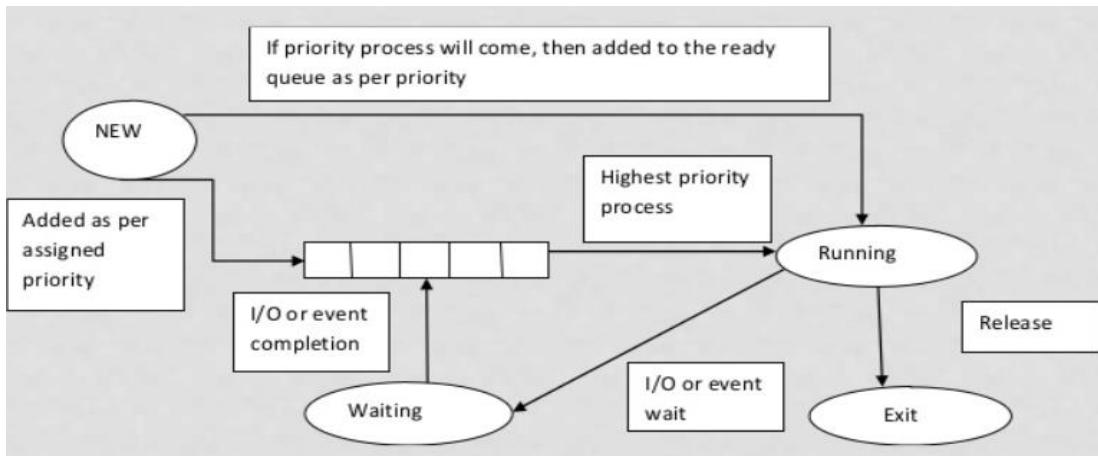
#### (3)-① Priority 알고리즘의 장점

프로세스의 내부적, 혹은 외부적인 priority 기준에 따라 우선순위가 높은 프로세스에게 CPU 사용 우선 권한을 주기 때문에, 이를 활용해서 중요도가 높은 프로세스를 먼저 수행하도록 할 수 있다.

#### (3)-② Priority 알고리즘의 단점

Indefinite blocking이라고도 알려진 starvation 문제는 priority 알고리즘의 중요한 이슘이다. 이 문제는 priority가 낮은 프로세스의 관점에서 계속 자신보다 높은 priority를 가진 프로세스가 ready queue로 들어와서 CPU를 할당 받지 못하고 우선순위가 계속 밀려서 작업을 수행할 수 없는 상황을 일컫는다. 그러나 Starvation 문제는 aging 기법으로 해결할 수 있는데, aging은 정책에 따라 오랜 시간 CPU 할당을 기다린 프로세스의 priority를 점차적으로 높여주어 definite blocking으로 개선시켜주는 기법이다.

#### (3)-③ Priority 알고리즘의 flow chart



Preemptive priority algorithm의 경우 running 중인 프로세스가 없을 경우 ready queue에서 가장 높은 priority를 가진 프로세스를 골라 CPU를 할당해주는 것으로, non-preemptive 방식의 경우 ready queue에 더 높은 priority를 가진 프로세스가 있으면 running 중인 프로세스가 있더라도 CPU를 더 높은 우선순위의 프로세스에게 할당해주고 이미 수행 중이지만 우선순위가 낮은 프로세스는 다시 ready queue로 돌아가도록 해주는 방법으로 이해할 수 있다.

#### (4) Round Robin (RR)

FCFS 알고리즘과 비슷하지만 여기에 preemption 기법과 time quantum이라는 CPU 사용 시간의 작은 단위의 개념을 도입한 것이 Round Robin 알고리즘이다. FCFS 알고리즘처럼 먼저 ready queue에 도착한 즉, 먼저 CPU 사용에 대한 request를 보낸 프로세스 순서로 CPU를 할당 받아서 작업을 수행하는데 time quantum 혹은 time slice라고 하는 보통 10-100 milliseconds의 시간이 지나면 수행 중이던 프로세스는 ready queue의 가장 앞에 있는 프로세스에 의해 preempt되고 ready queue의 가장 마지막으로 들어간다. 만약 time quantum 내에 프로세스의 수행이 모두 끝나면 그 프로세스를 terminate하고 남은 time quantum을 기다리지 않고 다른 프로세스에게 CPU를 할당해서 수행하도록 한다.

RR 알고리즘에서는 time quantum을 정하는 것이 중요한 이슈가 되는데, 만약 time quantum이 매우 크다면 결국 FCFS 알고리즘과 같게 되고 time quantum이 매우 적다면 context switch가 자주 빨리 일어나서 마치 여러 프로세스들이 동시에 수행되는 것처럼 보이는 processor sharing한 상황에 비유할 수 있게 된다. Context switch time은 running할 프로세스를 바꾸기 위해서 CPU를 release하고 다른 프로세스에게 CPU를 할당해주는 데, 수행 중인 프로세스의 정보를 해당 프로세스의 process control block에 저장하고 수행할 프로세스의 정보를 그 프로세스의 process control block에서 읽어오는 시간들을 말한다. Time quantum은 이 context switch time에 비해서 커야 하는데 그렇지 않으면 context switch time의 overhead가 너무 커져서 scheduling의 효율에 큰 영향을 줄 수 있기 때문이다.

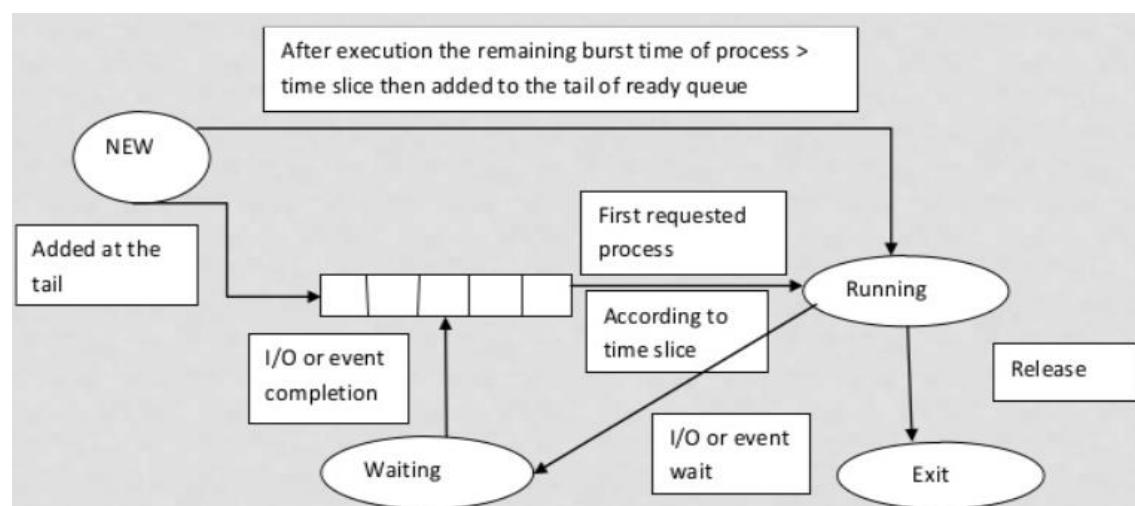
##### (4)-① RR 알고리즘의 장점

앞선 priority 알고리즘과 비교했을 때 RR의 장점은 starvation 문제가 발생하지 않는다는 것인데, 이것은 한 프로세스가 time quantum만큼의 시간만 한번에 최대로 CPU를 사용할 수 있고 circular order로 여러 프로세스들이 순서를 돌아가면서 수행을 하는 것이 보장되어 있기 때문이다. 만약 n개의 프로세스가 ready queue에 있고 time quantum이 q만큼이라면, 각 프로세스는 CPU 시간의  $1/n$ 을 한번에 최대로 q만큼을 가지므로 각 프로세스는  $(n-1)*q$  시간 이내에 자신의 다음 CPU 사용 차례가 돌아오게 된다. 따라서 보통 response time이 다른 알고리즈다 뛰어나다.

#### (4)-② RR 알고리즘의 단점

Time quantum에 따라 프로세스들이 한번에 수행할 수 있는 시간이 정해져 있기 때문에 상대적으로 context switch가 많이 발생하여 전체적인 효율성이 하락할 수 있다.

#### (4)-③ RR 알고리즘의 flow chart



#### (5) Lottery

각 프로세스에게 lottery ticket을 줘서 CPU scheduling이 필요할 때마다 랜덤하게 lottery ticket을 추첨해서 당첨된 lottery ticket을 가진 프로세스에게 CPU를 할당하는 방식이다. 추첨을 1 second 당 몇 번을 할 지 등은 정책에 따라 상이하다. Lottery 알고리즘에도 특정 프로세스에게 우선권을 부여할 수 있는데, 해당 프로세스에게 lottery ticket을 다른 프로세스들보다 더 많이 부여하면 추첨에서 당첨될 확률이 높아지므로 CPU를 할당 받을 확률이 높아지니까 우선권이 생긴다고 할 수 있다.

#### (5)-① Lottery 알고리즘의 장점

확률적으로 starvation이 일어나지 않는다. 특정 프로세스에게 lottery ticket을 많이 부여하더라도, 또 특별히 우선권을 부여하지 않더라도 프로세스의 정보에 따른 scheduling 방식이

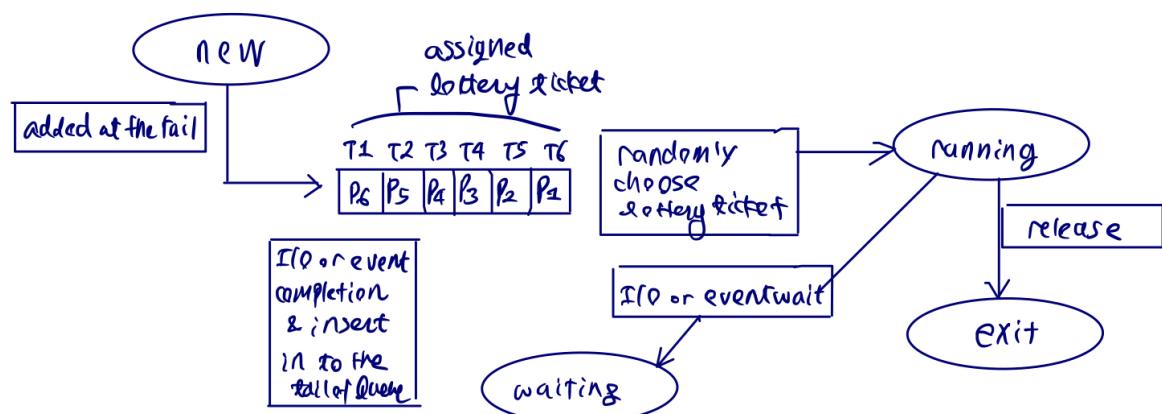
아니기 때문에 결국 확률에 따라서 모든 프로세스들이 (공평하게) 추첨에 당첨될 확률을 갖기 때문이다.

### (5)-② Lottery 알고리즘의 단점

추첨 방식이다 보니 프로세스 내부의 정보를 이용해서 scheduling 하지 않는다. 따라서 어떤 정책에 따라서 한 프로세스가 수행을 이어 나갈 수 없기 때문에 context switch time에 따른 overhead와 한 프로세스의 관점에서 효율성이 낮을 수 있다.

### (5)-③ Lottery 알고리즘의 flow chart

적당한 flow chart를 찾을 수 없어서, lottery algorithm에 대한 개념을 기반으로 직접 flow chart를 그려보았다.



## 1-4. CPU scheduling algorithms의 evaluation criteria

여러 CPU scheduling algorithm을 선택하거나 평가할 때 다음의 잣대들로 효율성을 측정한다. 결국 CPU scheduling은 CPU utilization, throughput을 최대화하고 ATT, AWT, response time을 최소화하는 방향으로 이루어져야 한다.

### (1) CPU utilization

CPU scheduling에 따라 프로세스들이 수행되는 총 시간에 대한 CPU가 idle하지 않고 busy한 채로 프로세스를 수행하는 시간의 비율이다. 이 비율이 높을수록 더 효율적인데, CPU가 idle한 시간이 적을수록 즉, CPU가 최대한 busy할수록 효율적이기 때문이다.

### (2) Average turnaround time (ATT)

Turnaround time은 한 프로세스가 수행하는 데 걸린 전체 시간이며, 프로세스가 ready queue에 들어오는 시점 혹은 submit되는 시점인 arrival time으로부터 수행이 완료되는 시점 까지의 간격이다. 각 프로세스의 turnaround time의 평균값이 ATT가 된다. 이 값은 작을수록 더 효율적인데, 각 프로세스들이 최대한 짧은 간격 내에 수행을 마치는 것이 더 효율적이기 때문이다.

#### (3) Average waiting time (AWT)

Waiting time은 한 프로세스가 ready queue에서 대기하는 기간을 뜻하고 각 프로세스의 waiting time을 평균 내면 AWT가 된다. 이 값은 각 프로세스들이 CPU를 할당 받기까지 오래 기다릴수록 커지고 대기 시간이 작을수록 줄어들기 때문에 작을수록 효율적이다.

#### (4) Average response time

Response time은 한 프로세스가 수행 의지 request를 submit한 시점으로부터 첫 응답 즉, 처음으로 CPU를 할당 받는 시점까지의 시간을 의미하며, 각 프로세스의 response time의 평균을 구한 것이 average response time이다. 이는 simulation을 통해서 측정하기 어렵지만, 본 과제 구현에서는 프로세스가 ready queue에 도달하는 arrival time과 처음으로 cpu를 할당 받는 시점 사이의 간격으로 측정하였다. Average response time은 값이 작을수록 효율적인데, 프로세스들이 평균적으로 짧은 시간 내에 요청에 대한 응답을 받은 것이기 때문이다.

#### (5) Throughput

단위 시간 당 완료되는 프로세스의 개수이며, 많은 프로세스가 완료될수록 효율적이다.

### 1-5. 팀 프로젝트 과제 개요

이번 팀 프로젝트는 CPU scheduling simulator를 구현하는 것으로 여러가지의 알고리즘에 따라 CPU scheduling을 진행하고 위의 evaluation criteria에 따라 평가한다. 이때, 각 알고리즘에 주어지는 데이터인 프로세스에 대한 정보는 모든 알고리즘에게 동일하게 주어져서 알고리즘 간 비교 분석이 가능하도록 구현했는데 simulator를 실행시키면 프로세스 id, CPU burst time, IO burst time, arrival time, priority 등을 포함한 프로세스에 대한 랜덤한 데이터가 생성된다. CPU scheduling이 프로세스들의 상태 변화에 따라 일어나는 경우를 위에서 설명했는데, 이것을 토대로 구현하기 위해서 new, ready, running, waiting, terminated 5가지의 각 상태에 해당하는 queue를 만들어주어 프로세스의 상태 변화를 표현했다. CPU scheduling algorithm을

평가하고 프로세스의 상태 변화를 위한 조건을 확인하는 등의 용도로 여러가지 count 기능의 변수를 도입하여 구현하였다. 추가적으로, aging 기법을 적용한 multi-level feedback queue도 구현해보았으며, 요약하자면 랜덤한 수의 프로세스를 생성해서 이들 간의 CPU scheduling을 여러가지 알고리즘에 따라 수행해서 간트 차트로 출력하고 평가하는 CPU scheduling simulator를 디자인하고 구현했다.

## II. CPU scheduling simulator의 구현

### 2장. 기존 CPU scheduling simulator

이번 2장에서는 3장에서 소개되는 직접 CPU scheduling simulator를 구현하기 위해서 참고한 기존의 CPU scheduling simulator 2가지를 소개하고 몇가지의 기준에 따라 비교 분석해 보았다. 비교 분석한 기준에 따라 기존 simulator의 장점은 차용해서 3장의 simulator를 구현하는데 참고했으며, 아쉬운 점은 개선했다. 2가지 simulator를 비교 분석하기 위한 기준들을 몇 가지 정해보았는데, 다음과 같다.

- IO 구현: IO 발생 횟수, IO burst time, IO start time
- Gantt chart의 제시
- Schedule되어야 할 프로세스들의 생성 방법
- Evaluation 구현: CPU utilization , AWT, ATT
- 추가 기능 여부: 추가적인 algorithm, multi-level feedback queues with aging

위의 5가지의 기준은 2가지의 기존 simulator에 공통적으로 적용되어 비교 분석을 해보았고 위 기준 외의 특징들도 언급하였다. 첫번째로 소개할 CPU scheduling simulator는 몇 해 전에 운영체제 수업을 수강하신 분이 open source로 공개하신 source code로 부록에 url을 첨부했다. 두번째 simulator는 과제를 위해 참고용으로 추천 받은 paper인데 언급된 url을 통해 simulator를 실행시켜 볼 수 있다고 소개되어 있지만 보고서를 작성할 때는 6월 7일 3시 42분 마지막으로 시도해 본 결과 해당 주소로 사이트를 찾을 수 없었다. 따라서 아쉽지만, paper 자체의 내용만을 가지고 분석을 해보았다.

#### 2-1. 첫번째 CPU scheduling simulator 소개

Single CPU 기반의 simulator이며, Source code가 모두 공개되어 있어서 구현 방법의 자세한 사항까지 확인할 수 있었고, simulator 구현에 필요한 모듈 및 전반적인 구조를 참고하면

서 3장의 simulator 구현을 위한 기본적인 구현 계획을 세울 수 있었다.

### (1) Simulator의 간단한 작동 방식 소개

본 simulator는 C언어로 작성되었으며, linux 환경에서 실행을 시킬 때 user로부터 schedule 할 프로세스의 개수와 발생시킬 IO 횟수를 입력받는 점이 특징이다.

전체적인 작동 방식에 대한 흐름은 시스템 자원의 초기화, 프로세스 생성, 알고리즘 별 시뮬레이션 시작, evaluation으로 볼 수 있다.

#### ① 시스템 자원의 초기화

Simulation에 사용되는 자원들을 초기화하는데, 이때 자원들은 Job(new) queue, ready queue, termination queue, waiting queue, 그리고 각 알고리즘 별로 simulation 결과 및 효율성 분석에 대한 정보를 담고 있는 evaluation queue를 모두 NULL로 초기화한다.

#### ② 프로세스 생성

이 simulator는 linux환경에서 실행을 시킬 때 user가 직접 프로세스의 개수와 발생시킬 IO 횟수를 입력 받고 이 정보를 활용하여 프로세스를 생성한다. 이때 IO를 수행할 프로세스를 랜덤으로 골라서 IO burst time을 랜덤으로 설정한다. IO 횟수를 최대 프로세스의 개수만큼만 받을 수 있도록 구현해서 프로세스 자체가 생성될 때 IO를 수행할 지의 여부를 랜덤으로 설정하는 것이 아니라, 프로세스를 생성하고 나서 IO를 수행할 프로세스를 user가 준 입력에 기반해서 고르는 방식이다. 프로세스는 구조체로 정의되어서 메모리 할당 후 임의의 값을 각 프로세스 구조체의 attribute 값에 설정해주는 것으로 생성되는데, 프로세스 구조체는 다음과 같다.

```
typedef struct myProcess {
    int pid;
    int priority;
    int arrivalTime;
    int CPUburst;
    int IOburst;
    int CPUremainingTime;
    int IOremainingTime;
    int waitingTime;
    int turnaroundTime;
    int responseTime;
}myProcess;
```

각 프로세스의 pid는 1부터 1씩 증가하며 할당되고 priority, arrival time, CPU burst, IO burst는 설정된 범위 내의 랜덤 값으로 설정된다. 이때 CPU burst 시간은 5이상의 임의의 값으로 주었다.

프로세스를 생성하고 나면 Job queue에 삽입하고 Job queue 출력 시 편의를 위해 pid를 기준으로 정렬한다. user가 입력한 개수만큼 프로세스를 생성하고 나면 초기 모든 프로세스들이 삽입된 Job queue를 복사해 두는데, 이는 알고리즘 별로 비교 분석을 하기 위해서 동일한 데이터를 부여하기 위함이다.

### ③ 알고리즘 별 시뮬레이션 시작

프로세스를 다 생성하고 나서 초기에 복사해둔 프로세스들을 매 알고리즘마다 Job queue에 로딩해온다. amount라는 인자를 통해서 simulation을 진행할 최대 시간을, 그리고 어떤 알고리즘을 수행할 것인지, 일부 알고리즘에 대해서 preemption 기법을 적용할 것인지에 대한 정보를 simulator 내부적으로 전달함으로써 CPU scheduling simulation이 본격적으로 시작된다. Simulation을 시작할 때, 후의 evaluation을 위해서 두 변수를 먼저 세팅하는데 computation\_start라는 변수에 가장 먼저 ready queue에 도착하는 프로세스의 arrival time을, computation\_idle이라는 변수에는 0을 대입한다. 두 변수는 알고리즘 별로 프로세스들이 모두 완료되었을 때의 시점과 함께 CPU utilization을 측정하기 위함이다.

최대 amount만큼의 시간만큼 루프를 돌면서 프로세스들이 모두 완료되면 루프를 끝내고 evaluation을 측정해 출력한다. 루프를 한번 도는 것은 simulation의 시간 상 1만큼의 시간이 흐르는 것이고, 매 1단위의 시간 간격으로 scheduling 알고리즘이 작동한다. 이 과정에 대해서 simulator를 구현한 분이 제시한 pseudo code는 다음과 같다.

```

for (i = 0; i < amount_of_time; i++) {
    while { -> Job Queue에 있는 프로세스들을 하나씩 차례대로 참조한다.
        if p == arrival time이 시간 i인 프로세스
            move(p, Job Queue -> Ready Queue);
    }

    nextProcess = 선택한 알고리즘(option);
    -> Running Process 혹은 Ready Queue의 프로세스 중 이번 turn (시간 i)에 수행되어야 할 프로세스를 선정

    if nextProcess != Running Process (선정된 프로세스가 원래 수행중이던 프로세스와 다른 경우)
        RunningStateTime = 0 (종일한 프로세스가 Running State였던 시간)
        if nextProcess.responseTime == -1
            nextProcess.responseTime = i;
        Running Process = nextProcess;

    elapseTime(Ready Queue);
    -> Ready Queue에 있는 프로세스들의 waitingTime ++, turnaroundTime ++

    elapseTime(Waiting Queue);
    -> Waiting Queue에 있는 프로세스들의 turnaroundTime ++, I/O burstTime --

    check I/O Completion(Wating Queue) {
        if p.IOburstTime == 0
            print ("p -> I/O complete");
            move(p, Waiting Queue -> Ready Queue);
    }

    if Running Process != null
        run(Running Process); -> RunningStateTime ++, CPU burstTime --
        if Running Process.CPU burstTime <= 0
            move(Running Process -> Terminated);
            print("Terminated");
    else
        if Running Process.I/O burstTime > 0
            move(Running Process -> Waiting Queue);
            print("I/O Request");
    else
        print ("idle");
}

```

Job Scheduling

CPU Scheduling

I/O Processing

Running

I/O Request

매 시간(루프)마다 먼저 job scheduling을 해주는데, 그 시점이 arrival time이 되어서 new queue에서 ready queue로 이동해야 할 프로세스들이 있으면 ready queue로 삽입해준다.

그 다음에는 cpu scheduling이 각 알고리즘의 구현에 따라서 일어나는데, 이 시점에서 CPU를 할당 받는 프로세스를 선택해준다. 이전에 수행 중인 프로세스와 이 시점에서 CPU를 할당 받도록 선택된 프로세스가 다를 경우에는 각 알고리즘 모듈에서 수행 중이었던 프로세스가 한번에 연속적으로 수행한 시간을 0으로 초기화 시켜주는데, 이는 RR 알고리즘에서 활용하기 위함이다. 만약 새로 CPU를 할당 받아 수행하게 될 프로세스가 처음으로 CPU를 할당 받았다면, 현 시점과 arrival time의 간격 차를 통해서 response time을 측정한다.

그리고 나서는 ready queue, waiting queue, running 중인 프로세스에 대한 처리를 해주는데 먼저 ready queue에서 대기 중인 프로세스들의 waiting time과 turn around time을 1만큼 증가시켜 준다. Waiting queue의 프로세스들에 대해서는 turn around time은 1을 증가시키고, IO burst time은 1을 감소시켜서 만약 이 시점에서 프로세스의 IO 처리가 끝났다면 이 프로세스를 waiting queue에서 다시 ready queue로 이동해 준다. 수행 중인 프로세스가 없다면 CPU가 idle함을 출력해주고 CPU utilization 측정을 위해서 idle한 시간을 기록해주는 computation\_idle 변수에 1을 더해준다. 수행 중인 프로세스가 있으면 RR 알고리즘에서 time quantum을 지났는지를 체크해주기 위해서 한번에 수행 중인 시간에 대한 변수를 1만큼 더 해주고, CPU burst time은 1만큼 감소시켜서 남은 CPU burst time을 기록한다. 이 루프의 시점에서 수행 중인 프로세스의 작업이 완료되면 이 프로세스를 termination queue로 이동해준다.

프로세스가 IO를 수행할 프로세스로 임의로 선정되어 IO burst time이 양수의 값으로 존재하면 waiting queue로 삽입하고 IO 처리 과정에 들어갔음을 출력해 주는데, 이 simulator는 CPU burst time을 5 이상으로 설정해서 한 프로세스가 처음으로 CPU를 할당 받아 1만큼의 시간동안 CPU 작업을 하면, 바로 IO 작업을 하도록 waiting queue로 넣어주도록 설계되었음을 확인할 수 있다.

이렇게 모든 프로세스가 작업을 완료하고 termination queue로 들어갈 때까지 혹은 주어진 amount 시간까지 매 1단위의 시간 간격으로 루프를 반복하고 완료된 시점을 Computation\_end라는 변수에 CPU utilization 측정을 위해서 기록한다.

#### ④ Evaluation

각 알고리즘 별로 모든 프로세스가 완료되면, evaluation queue에 위에서 언급된 변수들을 기반으로 AWT, ATT, response time, CPU utilization을 계산해서 저장한다.

### (2) 기준에 따른 분석

#### ① IO 구현: IO 발생 횟수, IO burst time, IO start time

앞서 언급했듯이, 이 simulator는 프로세스를 다 생성하고 나서 user가 입력한 IO 횟수만큼의 프로세스를 임의적으로 선택해서 IO를 수행할 프로세스를 정했다. 이렇게 정해진 프로세스에는 설정된 범위 내의 임의적인 값이 IO burst time으로 설정되었다. 하지만 IO start time은 IO를 수행해야 하는 모든 프로세스에 대해서 static하게 구현되었는데, 각 프로세스가 CPU를 할당 받고 수행을 한 첫 시점의 끝에서 바로 IO를 시작하도록 설계했다.

#### ② Gantt chart의 제시

1단위의 시간으로 0부터 설정된 최대 simulation 시간인 amount만큼의 시간까지, 혹은 모든 프로세스들이 완료하는 시점까지의 간트 차트가 출력된다. 매 시점에서 수행 중인 프로세스가 있으면 그 프로세스의 pid를, 없으면 idle함을 보여주고, 추가적으로 IO 수행을 시작한 프로세스와 끝낸 프로세스, 알고리즘에 따라 preemption이 발생함을 제시했다.

#### ③ Schedule되어야 할 프로세스들의 생성 방법

앞서 작동 방식 소개의 프로세스 생성 부분에서 자세하게 설명했지만, 간단히 요약하자면 프로세스 구조체가 선언되어 있어서 프로세스 id, arrival time, CPU burst time, IO burst time을 대입함으로써 프로세스가 생성되었다. 이때 프로세스 id는 순차적으로 할당되었고 arrival time과 CPU burst time은 임의로 설정했다. IO burst time은 해당 프로세스가 IO를 수행할 프로세스로 선정된 경우 0 이상의 특정 범위 내 값을 대입해준다.

#### ④ Evaluation 구현: CPU utilization, AWT, ATT

각 알고리즘을 수행하기 전과 수행 중에 computation이 시작되는 시점, CPU가 idle한 시간, computation이 끝나는 시간을 측정하여 CPU utilization을 측정한다. 그리고 각 프로세스 별로 waiting time과 turn around time을 측정하여 평균을 계산해서 AWT와 ATT를 구한다. 추가적으로 각 프로세스 별로 arrival time과 처음으로 CPU를 할당 받는 시점의 간격 차이를 구해서 response time을 구하고 average response time을 측정한다.

#### ⑤ 추가 기능 여부: 추가적인 algorithm, multi-level feedback queues with aging

FCFS, SJF, Priority, RR 알고리즘 외에도 2가지 CPU scheduling 알고리즘이 구현되어 있었는데, Longest-IO-First 알고리즘과 Longest-IO-Shortest-CPU-First 알고리즘이다. 두 알고리즘 모두 preemptive와 non-preemptive 방식 모두 설계되어 있다. 먼저 Longest-IO-First 알고리즘은 IO burst time이 길수록 우선적으로 CPU를 할당해주는 방법이며, Longest-IO-Shortest-CPU-First 알고리즘은 Longest-IO-First를 보완하여 IO burst time이 긴 순서대로 CPU를 할당하지만, 프로세스 간 IO burst time이 같은 경우를 고려해주었다. 즉, IO burst time이 같은 프로세스들이 2개 이상이라면, 이들 사이에서 CPU burst time이 가장 적은 프로세스를 선정하도록 한 것이다.

추가적인 알고리즘 구현 외에 우선순위가 낮은 프로세스들의 starvation을 방지하기 위해서

aging 기법을 적용한 multi-level feedback queues는 구현되어 있지 않았다.

## ⑥ 기타

이 simulator의 구현은 프로세스의 상태에 따른 queue를 잘 구분했고, scheduling의 과정에서 프로세스의 상태 변화가 있으면 queue별로 프로세스 간의 이동을 적절히 해준 것과 전체적인 simulation의 흐름 구조가 잘 설계되어 있었다.

## 2-2. 두번째 CPU scheduling simulator 소개

이 simulator 역시 single CPU를 바탕으로 한 CPU scheduling simulator이며, Paper 내용만을 참고할 수 있고 simulator를 직접 실행시켜서 체험해보거나 source code를 열람할 수는 없어 아쉽지만, IO와 관련해서 좀 더 현실적인 설정을 바탕으로 설계되었다. 앞서 소개된 simulator와 달리 Java로 작성되었다고 소개되어 있다.

### (1) Simulator의 간단한 작동 방식 소개

CPU scheduling simulator 프로그램을 실행시키는 user가 원하는 알고리즘을 선택하면 자동으로 저장되어 있던 프로세스 정보들이 로딩되면서 simulation은 시작된다. 프로세스에 대한 정보를 configure 창에서 저장할 수 있기 때문에, 각 알고리즘 별로 동일한 데이터에 대한 simulation이 가능하다. User가 next라는 버튼을 누름으로써 1만큼의 시간이 흐르고 각 시점에 대해서 해당 시점까지의 간트 차트, 프로세스들의 상태, running queue, waiting queue, ready queue의 정보, 그리고 새로 ready queue에 삽입된 프로세스나 IO를 시작한 프로세스 등에 대한 event 정보를 보여준다. 모든 프로세스들이 작업을 완료할 때까지 user가 next 버튼을 누르거나 auto 옵션을 통해 자동으로 모든 프로세스들이 수행을 끝난 시점으로 이동하게 되면 statistics 버튼이 활성화된다. 이 버튼을 통해 이동한 창에서는 해당 알고리즘에 대한 evaluation 정보가 열람 가능하다.

### (2) 기준에 따른 분석

#### ① IO 구현: IO 발생 횟수, IO burst time, IO start time

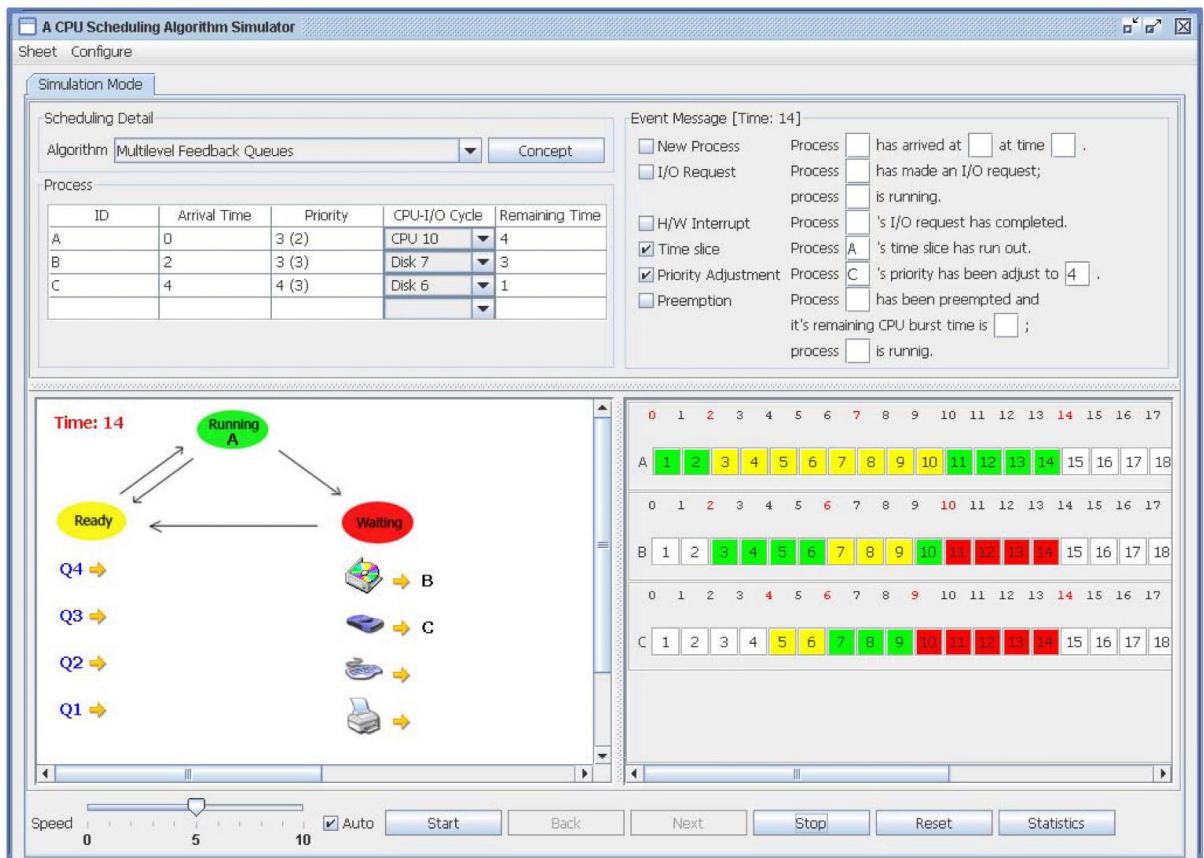
랜덤하게 프로세스를 생성하는 옵션을 선택할 경우 IO 발생 횟수 및 IO burst time은 임의의 값으로 설정이 되고 user가 입력하는 경우에는 IO 발생 횟수 및 IO burst time은 user의 입력대로 설정이 된다. 하지만 IO start time에 대한 정보는 paper를 통해 얻을 수 없었다.

이 simulator에서 흥미로운 점은 IO device를 고려했다는 점인데, disk나 printer 등의 여러 가지 IO device를 임의적으로 혹은 user 입력으로 프로세스에 설정함으로써 IO device 내에서

프로세스들 간의 IO scheduling도 구현되었다. 하지만 아쉽게도 IO device들에 대한 scheduling 알고리즘은 paper에 소개되어 있지 않아 확인할 수 없었다.

### ② Gantt chart의 제시

이 simulator의 가장 큰 장점은 gantt chart와 running, waiting, ready queue의 상태 및 프로세스의 상태 변화를 user를 위해 그래픽 애니메이션으로 다음의 결과창으로 보여준다는 점이다.



프로세스들이 모두 완료될 때까지의 간트 차트와 running, ready queue의 프로세스들과 waiting queue의 프로세스 및 처리 중인 IO 장치를 이미지 기반으로 매 시점 확인할 수 있다. 또 job scheduling에 의해 ready queue로 새로 들어온 프로세스, IO 작업을 시작한 프로세스, RR 알고리즘의 경우 time quantum이 만료된 프로세스, aging 기법에 따른 프로세스의 priority 및 ready queue 간 이동을 event message 부분을 통해 확인할 수 있다.

### ③ Schedule되어야 할 프로세스들의 생성 방법

이 simulator는 크게 3가지 방법으로 프로세스들을 생성한다. 첫번째는 미리 입력되어 있는 프로세스들에 대한 정보를 사용하는 것이고, 두번째는 user가 직접 프로세스를 등록하는 것이다. 마지막으로는 random하게 프로세스들을 생성하는 것이다. 세가지 방법 모두 생성되는 프로세스들에 대한 정보는 프로세스 ID, arrival time, priority, CPU time, IO device, IO time을 포

함한다. 이때 user에게 프로세스 ID는 A, B, C와 같은 알파벳 대문자로 보여지며 이 simulator가 먼저 소개된 simulator보다 더 현실적인 부분은 IO 설정에 있어서 IO device에 대한 개념을 도입했다는 것이다. Printer, disk 등의 IO device들이 있다. 또 user가 설정할 수 있는 부분은 RR 알고리즘의 time quantum, multi-level feedback queues의 aging 기법에서 언제 얼만큼 priority를 높여주고 낮게 할 것인지 등이 있다.

#### ④ Evaluation 구현: CPU utilization, AWT, ATT

Source code가 공개되어 있지 않으므로 내부적으로 어떤 변수를 통해서 효율성 측정을 위한 기준들이 계산되는지는 알 수 없다. 단지 statistic이라는 창을 simulation이 끝난 후에 띄워서 각 프로세스의 response time, waiting time, turnaround time과 모든 프로세스에 대해서 average response time, average waiting time, average turnaround time 및 CPU utilization을 출력해준다고 설명되어 있다.

하지만 각 알고리즘 별로 이러한 evaluation 정보를 확인할 수 있고, 하나의 창에서 여러 알고리즘의 evaluation 정보를 볼 수는 없어서 비교하는데 어려움이 있을 수 있다고 생각했다.

#### ⑤ 추가 기능 여부: 추가적인 algorithm, multi-level feedback queues with aging

알고리즘은 FCFS, RR, SJF를 구현했으며 추가적인 알고리즘은 없다.

Multi-level feedback queues(MLFQ)가 구현되어 있는데 이것은 priority에 기반해서 몇 개의 ready queue가 있고, 일정 기간 이상 CPU를 점유하거나 점유하지 못한 프로세스의 priority를 낮추고 높임으로써 ready queue들 사이의 프로세스 이동이 일어난다고 설명되어 있다. Aging 기법에 해당해서 priority increase와 decrease가 어느 조건에 얼만큼 일어나는지는 configure 창에서 user가 설정할 수 있다.

하지만 queue 안에서는 RR 알고리즘을 쓴다고만 명시되어 있을 뿐, 모든 ready queue에서 RR 알고리즘을 사용하는지, 그렇다면 각 ready queue에서 동일한 time quantum을 사용하는지, 그리고 기본적으로 몇 개의 ready queue를 사용하는지와 ready queue들 사이의 우선순위(CPU 시간 차지 비율)는 어떻게 되는가에 대해서는 전혀 설명되어 있지 않다.

#### ⑥ 기타

프로세스 정보를 랜덤하게 생성할 때 CPU time의 범위를 어떻게 설정했는지, user가 입력할 때 CPU time이 0이하이거나 2미만일 때 예외 처리를 해주었는지에 대한 정보는 아쉽게도 paper에 소개되지 않았다. 이부분에 대해 아쉬운 점은, IO를 처리하는 프로세스의 경우 CPU 작업을 수행하는 시간 범위 내에 IO 작업이 시작되고 끝나야 하는데 CPU time 범위에 대한 설명이 없어서 CPU time이 부적절하게 입력되거나, IO가 언제 시작되는지를 이해할 수 없었다. 또 IO device가 여러 개 있는데, 같은 IO device에 2개 이상의 프로세스들이 IO 작업을 원

할 때, 이 프로세스들 간의 IO scheduling은 어떻게 작동하는 지에 대한 정보도 얻을 수 없었다.

### 2-3. 두 가지 기존의 simulator 비교 분석 및 차용, 개선할 점

소개한 두가지 기존의 simulator에 대해서 5가지의 비교 분석 기준을 적용해서 소개하였다. 이 기준들을 바탕으로 직접 CPU scheduling simulator를 구현함에 있어서 차용할 점과 개선할 점을 정리해보았다.

#### ① IO 구현: IO 발생 횟수, IO burst time, IO start time

IO의 발생 횟수와 관련해서는 첫번째 simulator처럼 프로세스가 다 만들어진 후, user가 입력한 횟수 정보에 따라 IO를 수행할 프로세스를 선택하는 방식보다는 simulator 내부에서 각 프로세스 별로 IO burst time을 임의의 값으로 부여해서 그 값이 0이면 IO를 수행하지 않는 것으로 개선할 수 있다. IO start time과 관련해서는 두번째 simulator에서는 정보를 얻을 수 없었고 첫번째 simulator의 IO start time이 IO를 처리해야 하는 모든 프로세스들에 static하게 적용된 점을 고려했을 때, IO 시작 시간을 임의로 설정해주는 설계를 도입해서 좀 더 현실적인 simulation을 구현할 수 있겠다고 생각했다.

#### ② Gantt chart의 제시

간트 차트 display와 관련해서는 두번째 simulator가 훨씬 이해하기 쉽게 정보를 보여준다는 것을 알지만, 첫번째 simulator처럼 텍스트 기반으로 0 시간부터 모든 프로세스가 작업을 완료하는 시점까지 혹은 최대 시점까지 간트 차트를 출력하기로 계획했다. 그 이유로 첫번째는 시간 상의 여유가 없었다. 그리고 두번째로 C 언어로 구현해야 한다는 점을 고려했을 때 Java가 Graphic User Interface에 좀 더 최적화되어 있는 프로그래밍 언어이고 C 언어로 그래픽 애니메이션을 다룰 줄 모른다는 한계점이 있었다.

#### ③ Schedule되어야 할 프로세스들의 생성 방법

첫번째 simulator의 프로세스 구조체를 차용하여 필요한 attribute를 추가해서 구현하기로 했다. 단, 프로세스를 생성할 때 입력되는 attribute의 값 중 IO start time이라는 값을 도입해서 random하게 IO 작업을 시작하는 시간을 설정했다. 이와 관련해서 IO start time이라는 변수의 정확한 의미와 어떻게 IO 시작 시간을 구현했는지는 3장에서 설명하기로 한다.

#### ④ Evaluation 구현: CPU utilization, AWT, ATT

두 simulator에서 제시한 것처럼 각 프로세스 별로 waiting time, turnaround time, response time을 측정하고 이를 모든 프로세스에 대하여 평균 내어 average response time, AWT, ATT를

계산하여 출력하도록 계획했다. CPU utilization을 측정하는 것에 있어서 필요한 전체 CPU 시간 및 CPU가 idle한 시간에 대한 변수 설정은 첫번째 simulator에서 사용하여 구현했다.

#### ⑤ 추가 기능 여부: 추가적인 algorithm, multi-level feedback queues with aging

두 simulator 모두 다루지 않은 Lottery 방식의 scheduling 알고리즘을 구현하고 FCFS 알고리즘도 preemptive 기법을 시도해보기로 계획했다. 또 두번째 simulator에서 설명된 MLFQ 알고리즘을 바탕으로 ready queue가 2개인 MLFQ도 설계해서 구현해보았다.

#### ⑥ 기타

Simulation의 전반적인 흐름과 결과 출력에 있어서 첫번째 simulator의 구현이 open source로 제공되어 있었기 때문에 이를 참고하여 3장의 simulator를 구현하기로 했다. 앞서 정리해본 개선점과 여러가지 변화를 주었고, 수차례의 디버깅과 간트 차트를 여러 번 그려서 확인해 봄으로써 잘못된 알고리즘에 대해서는 수정 작업도 동반하였다.

## 3장. 직접 설계한 CPU scheduling simulator

2장의 두 가지 기존 CPU scheduling simulator에 착안해서 비교 분석한 내용을 바탕으로 직접 CPU scheduling simulator를 설계하고 구현해 보았다. 구현한 simulator를 요약하자면, 랜덤한 수와 정보를 담고 있는 프로세스들을 생성해서 preemptive & non-preemptive FCFS, preemptive & non-preemptive SJF, preemptive & non-preemptive Priority, RR (time quantum = 3), Lottery 알고리즘에 따라 CPU scheduling simulation을 진행하고 결과로 gantt chart, evaluation을 출력한다. 또 Multi-level feedback queue를 구현하여 ready queue가 2개인 aging 기법을 적용한 simulator도 결과로 gantt chart와 evaluation을 출력한다.

### 3-1. simulator 기본 사항 및 시스템 구조 소개

본 simulator의 모듈을 이해하는 데에 있어서 필요한 시스템의 기본적인 설정 사항들을 소개한다.

#### (1) 프로세스와 evaluation 구조체 소개

본 simulator 프로그램에서는 CPU scheduling할 프로세스의 개수와 내부 정보들을 모두 프로그램 내에서 임의의 값으로 설정하기 때문에 사용자가 인자로 어떠한 값도 넘겨주지 않는다. 알고리즘 역시 simulator 내에서 구현되어 있는 각 알고리즘에 대해서 동일한 프로세스 데이터로 자동적으로 순차적인 실행이 되기 때문에 한 알고리즘에 대해서 실행이 완료되면 이 알고리즘의 효율성 정보를 evaluation 구조체의 배열에 저장하고 다음 알고리즘에 대한

scheduling을 실시한다.

다음은 simulator에서 구현한 프로세스들이 가지는 값으로 프로세스 구조체의 attribute에 해당한다. Simulator 내부에서 각 값을 할당함으로써 프로세스가 생성된다.

Process (총 N개의 프로세스)	
<b>Pid</b>	프로세스 아이디, 1~N 범위의 unique한 값
<b>Priority</b>	우선순위, 1~N 범위의 random 값
<b>Arrival time</b>	Ready queue로 삽입 시간, 0~(N+10) 범위의 random 값
<b>CPU burst time</b>	요구되는 CPU 작업 처리 시간, 2~12 범위의 random 값
<b>IO burst time</b>	요구되는 IO 작업 처리 시간, 0~5 범위의 random 값
<b>CPU remaining time</b>	처리하고 남은 CPU 작업 시간, CPU burst time으로 초기화
<b>IO remaining time</b>	처리하고 남은 IO 작업 시간, IO burst time으로 초기화
<b>IO start time</b>	IO 작업을 시작하는 상대적인 시간, (1~CPU burst-1) 범위의 random 값
<b>Progress</b>	IO 시작 시간 count를 위한 값, 0으로 초기화
<b>Waiting time</b>	Ready queue에서 대기 중인 시간, 0으로 초기화
<b>Turnaround time</b>	Ready queue에 도착해서부터 작업을 완료할 때까지 걸리는 시간, 0으로 초기화
<b>Response time</b>	Ready queue에 삽입된 후 처음 CPU를 할당 받은 시간 간격, -1로 초기화

Simulator를 실행시키면, 임의의 프로세스 개수가 정해지는데, 이 개수만큼 위의 프로세스 구조체에 값을 채워 넣고 Job(new) queue에 삽입하는 것을 반복한다.

- pid

pid는 프로세스의 id를 뜻하며, 정수 1부터 1씩 증가하며 각 프로세스에게 부여된다.

- Priority

priority 알고리즘으로 scheduling할 때 사용되는 attribute이며 각 프로세스의 우선순위를 뜻하는데, 이를 수치화해서 구현한 것이다. Priority 값으로는 1에서 총 생성되는 프로세스의 개수 사이의 임의의 값을 부여한다.

- arrival time

프로세스가 생성되고 Job(new) queue에 있다가 ready queue로 옮겨지는 시간을 말한다. 시간은 0부터 매 1을 단위로 흘러가는데, 매시간마다 simulation 루프가 돈다. 이 루프의 초기 부분에서 각 시점에 ready queue로 들어온 프로세스를 골라서 job queue에서 ready

queue로 옮겨 주는데, 이때 기준이 각 시점과 arrival time이 일치할 때이다. 즉, 시뮬레이션을 시작하고 각 프로세스의 arrival time만큼의 시간이 지나면 비로소 CPU를 할당 받을 준비가 된 프로세스가 되는 것이다.

이 arrival time 역시 simulator 내에서 임의의 값으로 설정되는데 다수의 프로세스가 생성될 때 너무 몰리는 것을 방지하기 위해서 0에서 전체 프로세스의 개수+10 사이의 값이 배정된다.

- cpu burst time

CPU burst time을 뜻하며 각 프로세스가 작업을 완료하기까지 CPU를 할당 받아서 얼마큼의 시간동안 처리해야하는지를 나타낸다. 매시점의 scheduling에서 수행하도록 CPU가 할당된 프로세스는 다음 시점의 scheduling까지 1만큼의 시간동안 CPU를 수행하는데, 시간을 1단위로 쪼개 보았을 때, 한 프로세스가 작업을 끝내기 위해서는 총 cpu burst만큼의 CPU를 할당 받아야 한다.

이 CPU burst는 2부터 12 사이의 값이 배정되는데 여기서 lower bound인 2는 단순히 정한 값이 아니라 의미를 지닌 값이다. 한 프로세스가 IO 작업을 시작하고 끝마칠 때는 반드시 그 프로세스의 CPU 시간 범위 내에서 모든 처리가 완료되어야 하는데, 만약 CPU burst가 1이라면, 해당 프로세스는 IO를 처리할 수 없다. 왜냐하면 CPU 시간 내에 IO가 시작되어야 하니까 최소한 1만큼의 CPU 연산을 해주고 IO를 시작해야 하는데, CPU burst가 1이면 CPU 시간이 끝난 후에 IO를 시작할 수 밖에 없기 때문이다.

- io burst time

프로세스가 얼마 동안 IO 작업을 수행할 지에 대한 정보이며, 해당 simulator에서는 IO device에 대한 개념이 도입되지 않았기 때문에 IO를 수행할 시간이 되면 주어진 io burst 만큼의 시간을 연속적으로 waiting queue에서 보낸다.

IO burst는 각 프로세스 당 0에서 5 사이의 값으로 임의 할당되는데, 0이 할당되면 그 프로세스는 IO 작업을 필요로 하지 않는다고 해석할 수 있다.

- cpu remaining time & io remaining time

cpu remaining time과 io remaining time은 각 프로세스가 작업을 완료해서 termination queue로 이동하는 것과 IO 처리를 완료해서 다시 ready queue로 돌아가는 것에 대한 조건을 확인하기 위한 attribute이다. 전자는 프로세스가 CPU를 할당 받아 running 중일 때 1씩 감소되며 후자는 waiting queue에서 대기하는 매 시점에 1만큼 감소된다. 두 변수 모두 0이 되면 프로세스의 각 상태 변화가 일어난다.

Cpu remaining time은 초기에 설정된 cpu burst로 초기화되고, io remaining time은 io burst로 초기화된다.

- io start time & progress

io start time과 progress는 각 프로세스의 IO 처리 시작 시간을 random하게 구현하기 위해 도입되었다. io start time은 attribute로 프로세스 별로 상대적인 범위 내의 값이 할당되는데, 이는 앞서 설명했듯이 IO 처리의 시작과 끝이 CPU 시간의 범위 내에 포함되도록 하기 위해서다. 각 프로세스의 CPU burst가 정해지면 io start time은 1부터 CPU burst-1 사이의 값으로 임의 설정된다.

Progress는 각 프로세스 별로 io 시작 시점을 체크하기 위함이며, 0으로 초기화된다. 각 프로세스가 CPU를 할당 받아서 수행을 할 때마다 1씩 증가시켜서, 이 progress 값과 io start time의 값이 같아질 때 io burst가 양수의 값이라면 해당 프로세스는 waiting queue로 옮겨져 IO 작업을 수행한다.

- waiting time & turnaround time & response time

waiting time과 turnaround time은 0으로 초기화되어 simulation이 진행됨에 따라서 각 프로세스가 ready queue에서 대기할 때 두 값이 모두 증가하고, 수행할 때의 경우 turnaround time의 값이 증가하도록 구현되었다. 따라서 각 프로세스 별로 측정해서 후에 모든 프로세스에 대한 average를 구하기 위해서 설정된 attribute다. Response time은 -1로 초기화되어서 CPU를 처음 할당 받은 프로세스의 경우 그 시점을 response time에 update한다.

다음은 각 알고리즘 별로 evaluation을 계산하고 출력하기 위한 evaluation 구조체의 attribute 값이다.

Evaluation	
Algorithm	매크로로 선언된 해당 알고리즘 번호
Preemptive	0 or 1
Start time	Computation start time(가장 먼저 도착한 프로세스의 arrival time)
End time	Computation end time(모든 프로세스가 종료되는 시점)
AWT	Average waiting time
ATT	Average turnaround time
ART	Average response time
CPU utilization	전체 CPU 시간 중 idle하지 않은 시간 비율
Completed	완료된 프로세스의 개수

각 알고리즘에 따라 preemptive, non-preemptive 방식이 구분되어 있으므로 preemptive 값이 1이면 해당 알고리즘을 preemptive 방식으로 작동시킴을, 0이면 그렇지 않음을 표기한다. Start time과 end time은 CPU utilization을 측정하기 위해서 최초로 CPU가 할당되어 CPU 작업이 시작되는 시점과 모든 프로세스가 작업을 완료하게 되는 시점을 기록한다. AWT, ATT,

ART은 해당 알고리즘에 대한 simulation이 끝난 후에 각 프로세스에 저장해둔 waiting time, turnaround time, response time의 평균을 구해서 전체 프로세스에 대한 효율성의 지표로 기능한다. 마지막으로 completed라는 값은 0으로 초기화되어서, 해당 알고리즘에 대해 simulation을 하면서 수행이 완료된 프로세스가 termination queue로 이동할 때마다 1씩 증가시켜준다. 이는 simulator를 실행해서 결과를 확인할 때, 모든 프로세스가 작업을 종료했는지를 쉽게 파악하기 위함이다.

## (2) IO 작업의 발생 및 수행 과정

본 simulator에서 각 프로세스가 생성될 때 임의로 설정된 IO burst 시간은 한 프로세스가 수행되는 동안 IO 작업을 수행해야 할 때, waiting queue에 있는 시간을 의미한다. 이때 IO burst가 0이면 IO 작업이 필요 없는 CPU-bound process임을 뜻하고, 그 외의 모든 양수 값에 대해서는 IO 작업을 IO burst 만큼 waiting queue에 상주하며 처리하는 것으로 의미를 두었다.

따라서 IO를 수행할 지에 대한 여부는 각 프로세스가 생성될 때 임의로 정해지며, IO 작업이 요구되는 프로세스에 대해서는 임의의 상대적인 시간에 IO 작업을 시작하도록 구현하였다. 각 프로세스가 io start time만큼의 CPU 연산을 처리하면 해당 프로세스를 running queue에서 waiting queue로 이동시키고 io burst만큼의 시간을 waiting queue에서 보내면, 다시 ready queue로 이동시켜 남은 CPU 작업을 완료할 수 있도록 하였다.

정리하자면, 한 프로세스는 임의로 IO 작업을 수행할지 말지가 결정되며, IO burst 시간도 임의로 할당된다. 또한 IO 작업을 시작하는 시점도 임의로 해당 프로세스의 CPU 시간 범위 내에서 설정되며, io start time이라는 변수가 상대적인 시작 시간을 의미한다. 결국 본 simulator에서 io 시작 시간은 각 프로세스 별로 상대적인 값이며, 이는 그 프로세스가 단위 시간 1을 기준으로 몇번의 CPU 작업 수행 후에 io를 시작할지 간격을 의미한다.

## (3) RR에서 time quantum 정의

RR에서 한 프로세스가 한번에 최대로 연속적으로 수행 가능한 시간을 의미하는 time quantum은 simulator 내부에서 3으로 정의하였다. 필요 시 사용자가 이 상수 값을 변경할 수 있다. 프로세스가 CPU를 할당 받아 최대 time quantum만큼의 시간동안 수행을 하고 나면 다른 ready queue의 프로세스에 의해서 preempt되어 ready queue로 이동한다.

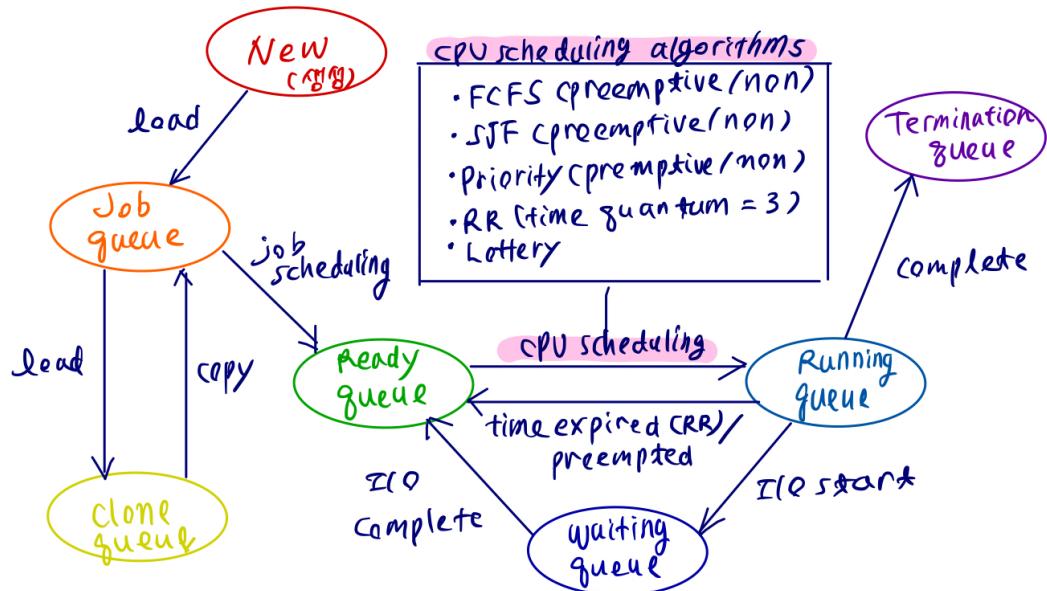
## (4) Priority 정의

각 프로세스가 지니고 있는 정보 중 하나인 priority는 해당 프로세스가 priority 알고리즘을 기반으로 scheduling될 때의 우선순위를 수치화하여 표현한 값이다. 본 시스템에서는 이 priority값을 정수로 1에서 총 프로세스 개수 사이의 값을 임의로 할당했으며, 값이 작을수록

높은 우선순위를 가지는 것으로 설계했다.

### (5) 시스템 구조 – queue 소개

Simulator를 통해 구현된 CPU scheduling simulation의 전체적인 시스템 구성은 다음의 도식처럼 각 프로세스의 상태 queue로 표현할 수 있다.



#### ① New

simulator가 시작되어 새로운 프로세스들을 임의로 생성하고 시스템에 필요한 job queue, ready queue, waiting queue, running queue, termination queue 등을 초기화한다.

#### ② Job queue

생성된 프로세스들이 simulation 시작을 위해 삽입되는 queue로 처리해야 할 프로세스들의 목록 정도의 의미를 지닌다. 새로 생성된 프로세스들은 job queue에 로딩된 다음 pid를 기준으로 오름차순으로 정렬하는데 이는 사용자가 임의적으로 pid를 생성했을 때와 job queue를 출력했을 때 사용자의 편의를 위해서이다.

정렬 후, 프로세스 생성 직후의 초기 job queue의 상태를 복사해서 clone queue에 저장해 두고 매 알고리즘의 수행 전 job queue를 초기화하고 clone의 상태를 job queue로 로딩해온다.

#### ③ Clone queue

프로세스 생성 직후 job queue의 초기 정보를 임시적으로 담는 곳으로, 여러 algorithm으로 scheduling simulation이 여러 번 수행될 때, 동일한 프로세스 정보를 활용하기 위해 도입되었다. 시뮬레이터가 매번 각 알고리즘 별로 실행될 때마다 clone queue에 저장된 프로세스 정

보들을 job queue로 로딩한 후 시뮬레이션을 시작해서 simulator 실행이 끝난 뒤, 같은 프로세스들에 대해서 각 알고리즘을 비교할 수 있도록 했다.

#### ④ Ready queue

Arrival time이 되어서 수행할 준비가 된 프로세스들은 job scheduling을 통해 ready queue로 옮겨지는데 이때 Job Scheduler는 매시간 단위의 제일 처음에 job queue의 프로세스들의 arrival time을 체크하여 ready queue로 이동시킨다. 따라서 ready queue는 running에서 ready로, waiting에서 ready로 삽입되는 프로세스들의 영향을 제외한다면, 자동적으로 arrival time을 기준으로 정렬된다. 동시에 도착한 프로세스들은 pid를 기준으로 정렬된다.

#### ⑤ Running queue

본 simulator는 single 프로세서 기반이기 때문에 각 scheduling algorithm에 따라 ready queue의 프로세스 중 하나가 running queue로 삽입된다. 따라서 running queue는 크기가 1인 배열로 구현되었고, 다른 모든 queue들은 최대 프로세스의 개수를 크기로 한다. 다시 말해서, 특정 시점에서 최대 하나의 프로세스만이 수행이 가능하다. 만약 preemption이 발생하면 running queue에 있던 프로세스는 다시 ready queue로 돌아가고 CPU를 선점한 다른 프로세스가 running queue에 삽입된다.

#### ⑥ Waiting queue

IO 작업을 수행해야 하는 프로세스가 io start time만큼 CPU를 할당 받아 수행을 하고 나면 Waiting Queue로 이동하게 되는데 여기서 해당 프로세스의 io burst time만큼의 시간을 보내고 다시 ready queue로 들어간다.

#### ⑦ Termination queue

프로세스들은 임의로 생성 시 할당된 CPU burst time만큼의 시간 동안 수행하고 그 시간 범위 내에 필요하다면 IO 작업을 처리한다. 이 과정의 끝에서 프로세스는 마지막으로 CPU를 할당 받아 수행하고 CPU remaining time이 0이 되어 작업을 완료하므로 termination queue에 들어간다. 배열로 Queue를 구현해 termination queue에 저장된 프로세스의 순서와 개수가 곧 종료된 프로세스의 순서 및 개수를 의미하도록 구성하였다. 각 알고리즘 별로 모든 프로세스의 작업이 끝난 시점에 termination queue에 저장된 프로세스들의 정보를 통해 evaluation을 측정한다.

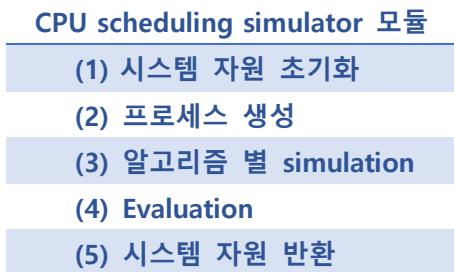
### (6) Simulation의 시간 단위

본 simulator에서 scheduling이 일어나는 시간 단위는 1이며 이 시간은 한번의 scheduling을 통해서 각 queue간의 변화와 각 프로세스의 정보 변화를 모두 고려해준 후 1을 더해줌으로써 다음 시점으로 넘어간다. 0 시간에서 시작하며 gantt chart의 가시성을 위해 최대 시간은 충분히 120으로 두었고, 모든 프로세스가 120 시간 전에 완료되면 simulation은 그 시점에서

종료된다.

### 3-2. CPU scheduling Simulator 모듈 구현 소개

Simulator 모듈은 각 프로세스의 생성부터 종료까지 앞서 언급한 시스템 흐름에 따라 simulation이 진행되는데, main 함수에 따라 전체적으로 아래의 단계로 구분할 수 있다.



#### (1) 시스템 자원 초기화

simulation에 사용될 job queue, ready queue, running queue, waiting queue, termination queue, evaluation queue의 값을 모두 NULL로 초기화한다. Evaluation queue는 각 CPU scheduling algorithm의 성능 정보를 담아둬서 알고리즘 간의 비교 분석을 하기 위한 것이다.

#### (2) 프로세스 생성

다음은 프로세스 생성과 관련된 간략한 pseudo code이다.

```
total_num_process = 1~10 범위의 랜덤 값;

for (i = 0; i < total_num_process; i++) {
    CPU burst time = 2~12 범위의 랜덤 값;
    IO burst time = 0~5 범위의 랜덤 값
    IO start time = 1~(CPU burst time-1) 범위의 랜덤 값;

    (i+1) 번째 process 메모리 할당;
    { // (i+1) 번째 프로세스 생성(attribute 대입)
        pid = i+1;
        priority = 1~total_num_process 범위의 랜덤 값;
        arrival time = 0~(total_num_process+10) 범위의 랜덤 값;
        CPU burst time = CPU remaining time = CPU burst time;
        IO burst time = IO remaining time = IO burst time;
        IO start time = IO start time;
        waiting time = turnaround time = progress = 0;
        response time = -1;
    } -> job queue에 insert
}

sort_jobQ();
clone_jobQ();
```

3-1.(1)에서 자세히 설명한 것처럼, 프로세스를 생성하는 것은 임의의 1개 이상의 개수의

프로세스 구조체에 attribute 값을 할당하는 것인데, 이 과정에서 필요한 메모리 할당과 값의 attribute에 임의 값을 넣거나 초기화한다. 필요한 개수만큼의 프로세스가 모두 만들어지면 프로세스들을 job queue에 넣어주고 pid 기준 오름차순으로 프로세스를 job queue 내에서 sorting해준 후 이 정보를 여러 알고리즘에 적용하기 위해서 clone queue로 복사해서 넣어둔다.

### (3) 알고리즘 별 simulation

각 알고리즘 별로 simulation 모듈은 0시간부터 모든 프로세스가 완료될 때까지 혹은 최대 120시간까지 매 1시간을 단위로 루프를 돌게 된다. Simulation 시작 전, clone queue에서 프로세스 정보를 job queue로 로딩해온다. CPU utilization 측정을 위해 프로세스 중 가장 이른 arrival time을 computation 시작 시간으로, CPU가 idle한 computation 시간을 측정하는 변수에는 0으로 초기화 시켜준다. Simulation 모듈은 루프를 돌 때 현재 시간, 알고리즘, 알고리즘 별 preemption 여부, 그리고 RR의 경우 time quantum을 인자로 넘겨받음으로써 작동한다.

다음은 simulation 모듈에 대한 간략한 pseudo code이다.

```
for (now_time = 0; now_time<max_time OR 모든 프로세스 완료; now_time++)
{
    1. job scheduling
    for(job queue의 처음부터 끝까지 모든 프로세스에 대해){
        if(arrival time이 now_time인 프로세스)
            move 프로세스 from job queue to ready queue;
    }

    prevProcess = 수행 중이던 프로세스;

    2. CPU scheduling
    thisProcess = 알고리즘에 따라 이번 시점에서 프로세스 선택;

    if(prevProcess != thisProcess){
        연속수행시간 = 0;
        remove prevProcess from running queue;
        insert thisProcess to running queue;
        if(prevProcess가 처음으로 CPU를 할당 받음){
            response_time = now_time - arrival time;
        }
    }

    3. ready queue 내 프로세스 처리
    for(ready queue의 처음부터 끝까지 모든 프로세스에 대해){
        waiting_time++; turnaround_time++;
    }

    4. waiting queue 내 프로세스 처리
    for(waiting queue의 처음부터 끝까지 모든 프로세스에 대해){
        IO_remaining_time--; turnaround_time++;
        if(IO_remaining_time==0인 프로세스)
            move 프로세스 from waiting queue to ready queue;
    }
}
```

```

5. running queue 프로세스 처리
if(CPU not idle){ //thisProcess != NULL
    CPU_remaining_time--; turnaround_time++; 연속수행시간++;
    progress++;

    6. terminated 프로세스 처리
    if(CPU_remaining_time==0)
        move 프로세스 from running queue to termination queue;

    7. IO 시작되는 프로세스 처리
    if(progress==IO_start_time AND IO_burst_time>0){
        move 프로세스 from running queue to waiting queue;
    }
}
else{//CPU idle
    idle한 시간++;
}
}

```

이 전체의 simulation 루프가 한번 돈다는 것은 1만큼의 시간 경과를 뜻한다. 매 시점에서 가장 먼저 일어나는 것은 job scheduling인데, job queue를 탐색하면서 arrival time이 현 시점이 된 프로세스가 있으면 해당 프로세스를 ready queue로 옮겨 준다.

두번째로는, 이번 시점에 대한 CPU scheduling이 일어나는데 이때 각 알고리즘 내부에서 현재 수행 중인 프로세스와 ready queue의 프로세스들을 모두 고려해서 이번에 수행될 프로세스를 선택한다. 직전 시점에 수행된 프로세스와 이번 시점에 선택된 프로세스가 같다면, running queue에 변화가 없고 다른 queue에 대한 처리로 넘어가면 된다. 하지만 다르다면, RR 알고리즘을 위해 연속해서 수행한 시간을 0으로 다시 초기화해주고, running queue에서 이전의 프로세스를 제거하고 새로 선택된 프로세스를 넣어준다. Ready queue에 대한 삽입과 제거는 각 알고리즘에서 다룬다. 이때 running queue의 프로세스가 response time이 -1이라면 처음으로 CPU를 할당 받은 것이고, 이 시점과 해당 프로세스의 arrival time의 차이를 구해서 response time을 기록해준다.

CPU scheduling까지 끝나게 되면, 각 상태 queue의 프로세스에 대한 처리를 해준다.

먼저 ready queue 내의 프로세스들은 다음 scheduling의 기회를 노려야 하고 이번 시점은 1만큼 기다려야하기 때문에 waiting time과 turnaround time을 모두 1만큼 증가시켜 준다.

Waiting queue의 프로세스들은 현재 IO를 진행 중인 프로세스들이므로, 1만큼의 시간이 경과하고 있으니 IO remaining time을 1만큼 감소해주고 turnaround time을 1만큼 더해주는데, 이때 IO remaining time이 0이 되면 이번 시점의 끝에 IO 작업을 완료하고 다음 시점 전에 ready queue로 이동해야 하므로 waiting queue에서 ready queue로 옮겨 준다.

마지막으로 현재 수행 중인 프로세스 즉, running queue에 있는 프로세스에 대한 처리를 해주는데, running queue가 비어 있다면 현재 CPU가 idle하므로 idle한 computation 시간을 1만큼 증가해준다. 그렇지 않다면 수행 중인 프로세스의 CPU remaining time을 1만큼 감소해주고, turnaround time, 연속해서 수행한 시간, progress를 모두 1만큼 증가시켜 준다. 여기서

CPU remaining time이 0이 되면 이 프로세스는 작업을 완료하고 종료되는 것이므로 termination queue에 넣어주고 혹은 progress 값이 IO start time과 동일해지고 IO burst time이 양수로 IO를 수행해야할 작업이라면, IO 작업을 수행할 시간이 다가온 것으로 수행 중인 프로세스를 waiting queue에 넣어준다.

이렇게 job scheduling, CPU scheduling과 ready queue, waiting queue, running queue에 대한 처리가 모두 끝나면 비로소 1만큼의 시간이 경과한 것이고 다음 시점에서의 simulation 루프가 다시 돌게 된다. 그러나 모든 프로세스들이 작업을 완료하고 termination queue에 들어가거나, 최대로 설정한 시간인 120 시간을 넘게 되면 루프를 벗어나서 computation 종료 시간을 기록하고 알고리즘의 성능을 evaluation queue에 기록한다. 그리고 후에 simulation 할 알고리즘이 있으면 다시 queue들을 초기화하고 job queue에 clone queue의 프로세스들을 복사해옴으로써 다시 simulation 루프가 시작된다.

#### (4) Evaluation

한 알고리즘의 simulation 과정이 모두 완료되면, 효율성에 대한 평가를 진행하는데 termination queue의 프로세스들을 기반으로 Average Waiting Time, Average Turnaround Time, Average Response Time, 그리고 CPU Utilization을 측정한다. 이 정보들을 evaluation이라는 구조체에 저장하고 evaluation queue에 삽입하여 다른 알고리즘과의 비교분석이 가능하도록 한다.

#### (5) 시스템 자원 반납

구현한 모든 알고리즘에 대해서 동등한 프로세스 정보의 simulation이 끝나면 evaluation queue에 저장되어 있던 각 알고리즘 별 효율성 정보를 출력한 뒤, 사용한 자원을 시스템에 반납하는 과정을 진행한다. Job queue, ready queue, running queue, termination queue, waiting queue, clone queue, evaluation queue의 메모리를 모두 release함으로써 이 simulator의 실행은 끝이 나게 된다.

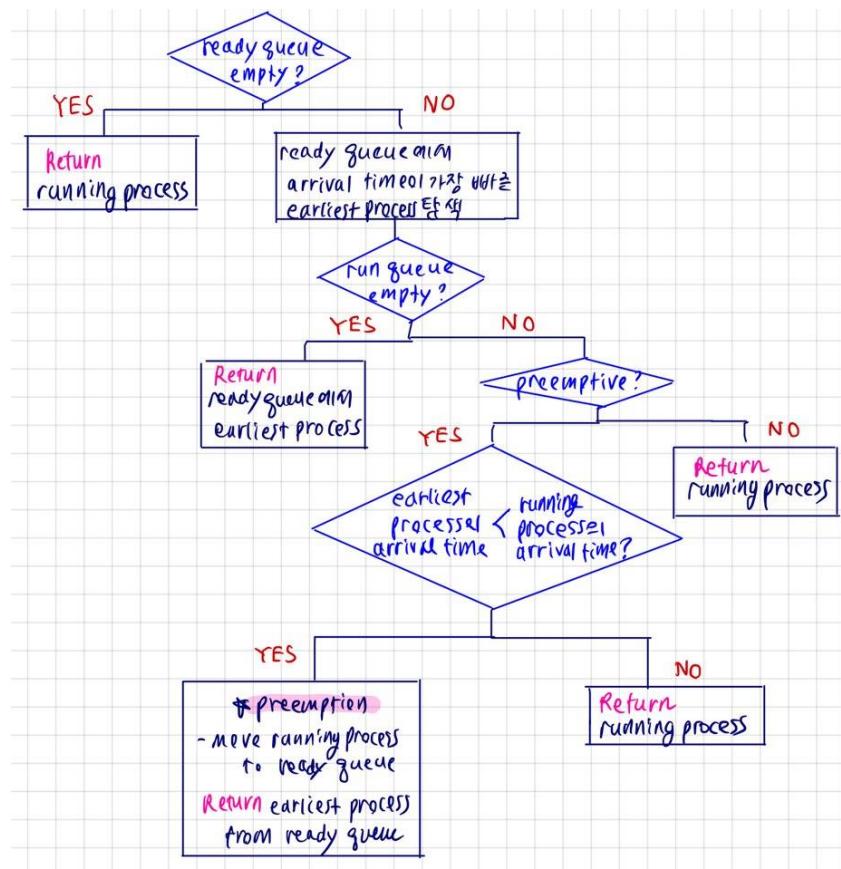
### 3-3. 알고리즘 구현 소개

각 구현된 알고리즘은 정해진 순서대로 같은 프로세스 데이터를 가지고 simulation을 진행하게 되는데, 이때 각 알고리즘의 작동 방식과 정책에 따라 매번 CPU를 할당 받는 프로세스는 다르게 선택된다. 이때 선택된 프로세스는 각 알고리즘에서 반환되는데, simulation 모듈에서 CPU scheduling을 할 때 이 프로세스를 해당 시점에 수행할 프로세스로 받는다. 본 simulator에서 구현한 알고리즘은 preemptive & non-preemptive FCFS, preemptive & non-

preemptive SJF, preemptive & non-preemptive Priority, RR, 그리고 Lottery 방식으로 preemptive FCFS와 Lottery 방식을 추가 기능으로 구현했다. Preemption이 적용된 알고리즘 기법의 경우, 확실한 기준에 부합할 때 preemption을 발생시키고 그렇지 않고 두 프로세스가 동등한 상황일 경우에는 굳이 context switch를 감수하지 않기 위해서 수행 중인 프로세스를 다시 선택하는 정책을 사용했다. 앞서 1-3에서 각 알고리즘에 대해서 자세히 설명했으므로, 이번 장에서는 간략한 소개와 세부 정책을 위주로 다루겠다. 각 알고리즘 별로 이해하기 쉽게 flow chart를 그려보았다.

### (1) FCFS

기본적으로 FCFS는 non-preemption 기반이나 다른 알고리즘처럼 preemption 기법을 추가해서 구현해 보았다. FCFS는 프로세스의 arrival time을 기준으로 먼저 도착한 프로세스를 선택하는 알고리즘이다.

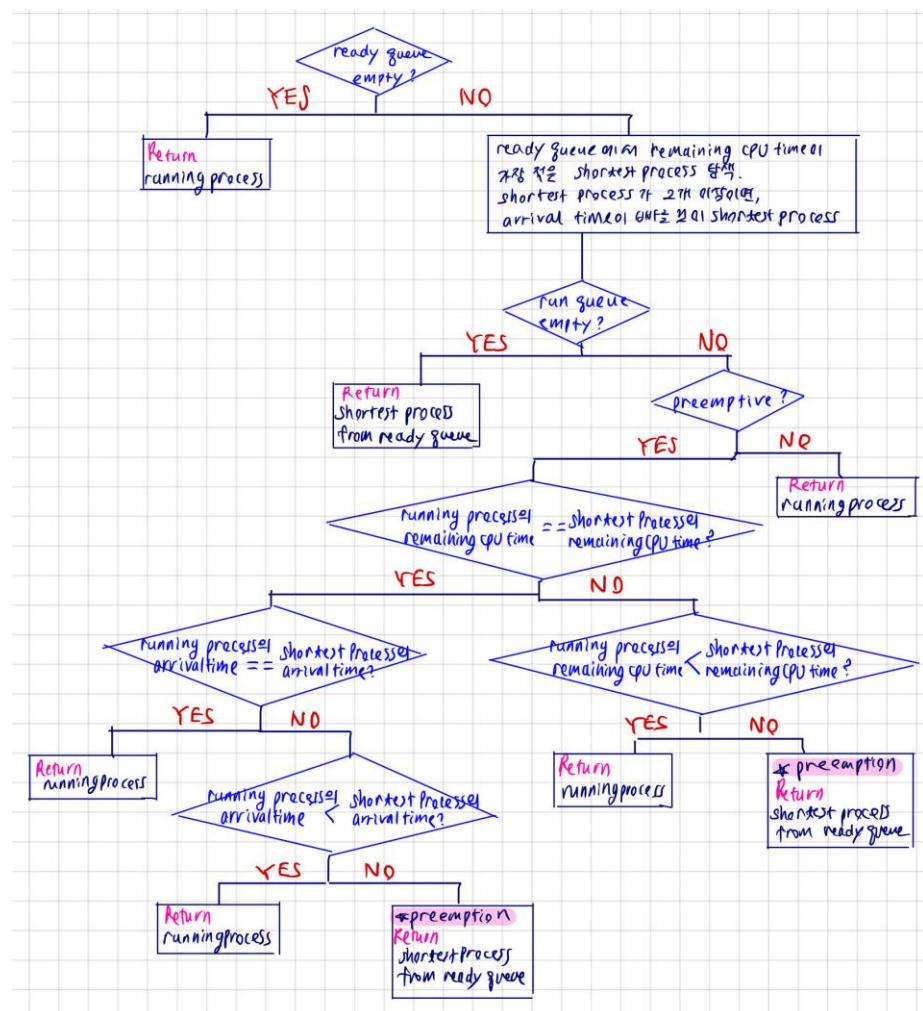


먼저 ready queue가 비어 있다면 CPU를 할당 받기를 대기하는 프로세스가 없다는 것이니까 원래 수행 중이던 프로세스를 선택한다. Ready queue에 프로세스가 존재한다면, ready queue 중에서 가장 arrival time이 빠른 프로세스를 선택하는데, 이때 run queue가 비어있다면 즉, 수행 중인 프로세스가 없다면 탐색한 가장 빨리 도착한 프로세스를 ready queue에서 삭제하고 반환하면 된다. 여기서 만약 non-preemptive라면 이미 수행 중인 프로세스가 있는데

굳이 preemption을 해서 context switch를 발생시키지 않지만 preemptive라면 현지 수행 중인 프로세스와 ready queue에서 찾은 가장 빨리 도착한 프로세스의 arrival time을 비교하여 후자의 것이 더 작다면 수행 중인 프로세스를 ready queue로 옮기고 해당 프로세스를 반환함으로써 preemption을 발생시키고 아니라면 현재 수행 중인 프로세스를 선택한다.

## (2) SJF

Next CPU time 즉, 처리하고 남은 CPU burst time을 기준으로 더 적은 시간이 남은 프로세스를 선택하는 알고리즘이다. 이때, 남은 CPU burst time이 동일한 프로세스가 2개 이상 존재한다면 통상적으로 FCFS의 원칙에 따라 arrival time을 기준으로 먼저 온 프로세스에게 우선권을 준다.

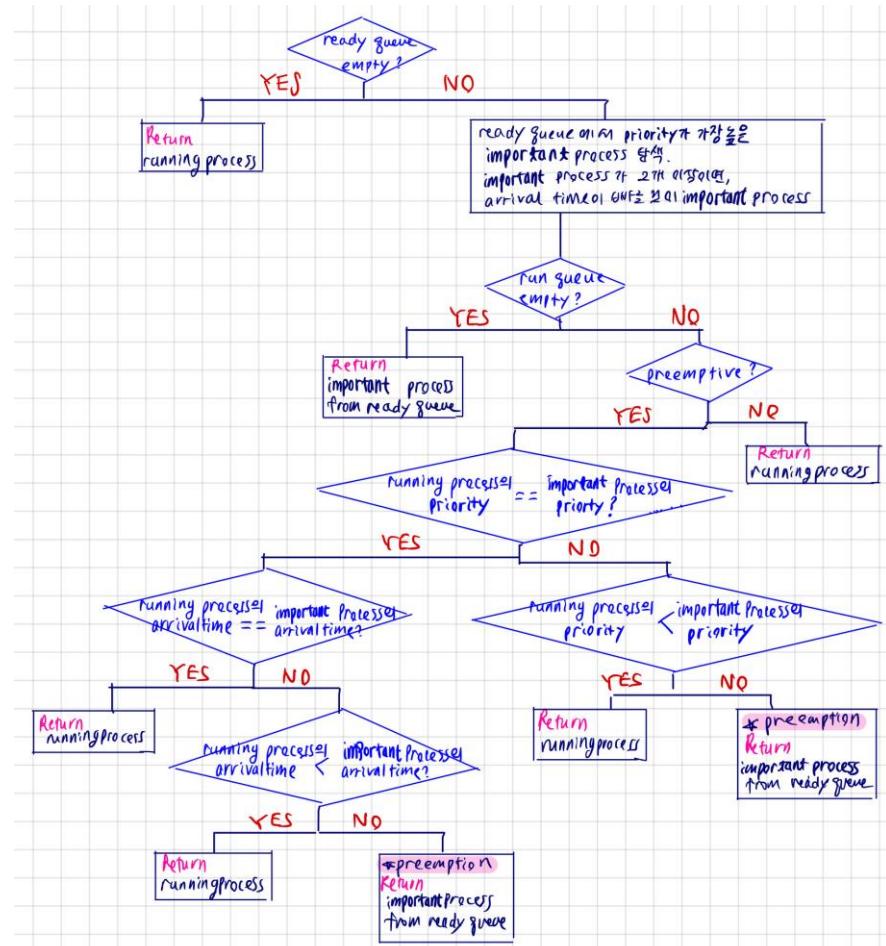


이 역시 만약 ready queue가 비어 있다면 수행 중인 프로세스를 반환하고 그렇지 않다면 remaining CPU time이 가장 적게 남은 프로세스를, 그리고 이러한 프로세스가 2개 이상이라면 arrival time을 기준으로 가장 빨리 도착한 프로세스를 shortest process로 선택한다. 이때 수행 중인 프로세스가 없다면 shortest process를 반환하고 있다면 preemptive 방식의 경우

수행 중인 프로세스의 next CPU burst와 shortest process의 next CPU burst를 비교해서 더 짧은 것을 반환한다. 여기서 비교가 그치지 않을 경우, 두 값이 같은 때인데 이때는 역시 arrival time을 기준으로 shortest process의 arrival time이 더 짧은 경우만 preemption을 발생시켜주고 그렇지 않은 경우는 수행 중인 프로세스를 택한다. Preemption이 발생하는 경우는 ready queue의 프로세스가 수행 중인 프로세스의 CPU를 선점하는 것인데, 이때 수행 중인 프로세스는 ready queue로 넣어주고 선점하는 프로세스는 ready queue에서 삭제해주어야 한다.

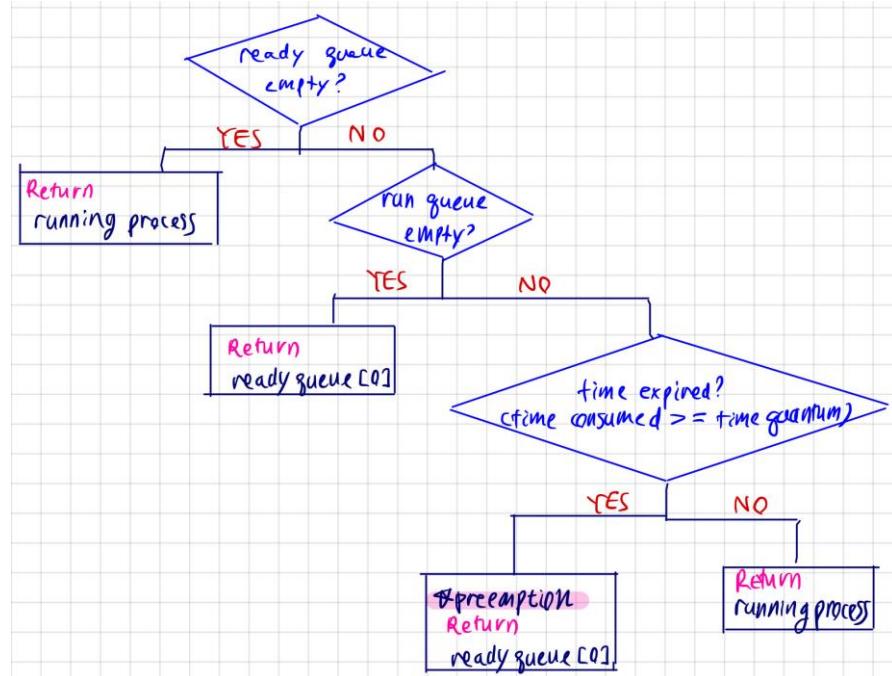
### (3) Priority

Priority는 프로세스의 우선순위를 수치화해서 이 값을 토대로 더 높은 우선권을 지니는 프로세스에게 CPU를 할당하는 방법이다. 본 simulator 시스템에서는 priority 값이 더 작을수록 더 높은 우선순위를 의미한다고 설정했다. Priority는 SJF 알고리즘과 비슷하게, 같은 우선순위를 지니는 프로세스들이 있으면, arrival time을 기준으로 더 높은 우선순위를 주었다.



#### (4) RR

RR 알고리즘은 simulation 모듈에서 연속적으로 수행하는 시간을 의미하는 time consumed 변수를 초기화하거나 증가시켜서 알고리즘 모듈에서 time quantum을 다 썼을 때와 그렇지 않을 때를 구분해주었다.



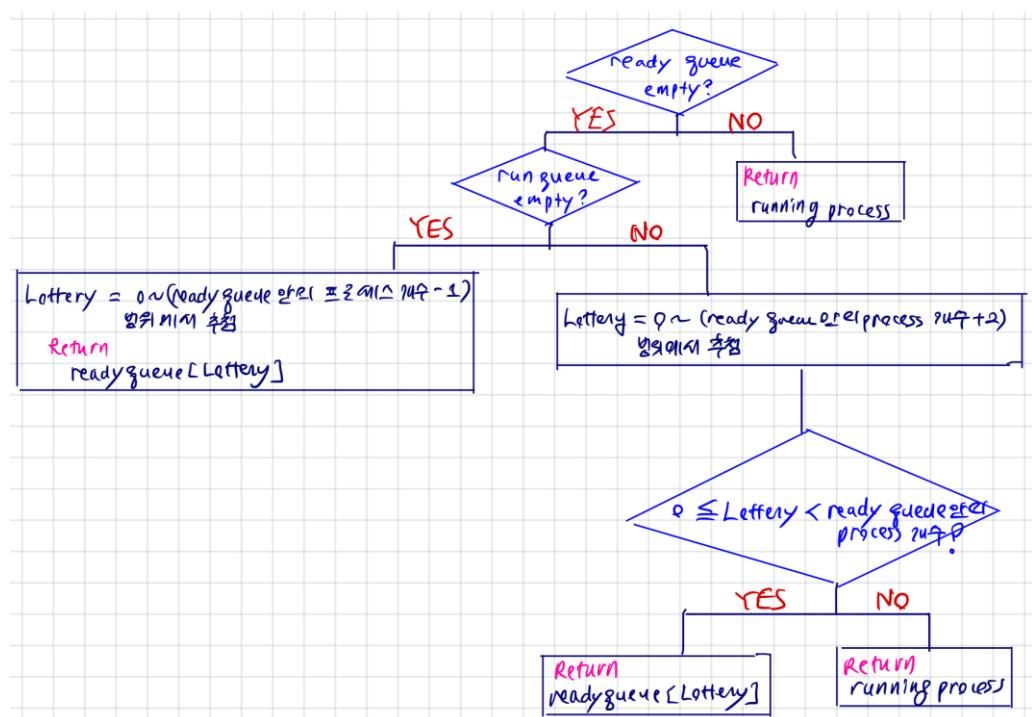
RR 알고리즘에서 추가적으로 고민한 것은, circular order로 ready queue에 누적되는 대로 즉, 처음에는 FCFS에 따라서 ready queue의 가장 앞에 있는 프로세스를 선택하지만, ready queue에는 IO를 끝내고 돌아온 프로세스와 새롭게 job schedule되어 들어온 프로세스, 그리고 time quantum을 다 써서 time expired interrupt에 의해 running queue에서 다시 되돌아온 프로세스들 간의 삽입 순서의 영향을 받게 된다. 즉, RR 알고리즘은 ready queue에 삽입되는 프로세스들의 순서 정책이 영향을 준다는 것이다. 정리하자면, ready queue로 프로세스가 들어오는 세 경우는 다음과 같다.

- Job queue -> ready queue
- Running queue -> ready queue
- Waiting queue -> ready queue

우선 본 simulator의 특정 시점이 시작되면 먼저 job scheduling을 함으로 job queue에서 새로 ready queue로 들어오는 프로세스를 가장 먼저 삽입해주었고, 그 후에 CPU scheduling 알고리즘에서 프로세스를 선택할 때 time quantum을 다 쓴 수행 중이었던 프로세스를 ready queue에 삽입해주었다. 마지막으로 IO 작업을 끝내고 ready queue로 돌아오는 프로세스를 삽입해주었다. 한 시점에서 이 세가지가 모두 발생하면 언급된 순서에 따라 ready queue에 삽입을 하고 ready queue의 가장 앞에 놓인 프로세스를 선택하도록 구현했다.

## (5) Lottery

1장에서 언급했듯이, lottery 알고리즘은 프로세스들에게 로또 티켓을 배부해서 추첨한 뒤 추첨 결과와 같은 티켓을 가지고 있는 프로세스에게 CPU를 할당한다. 이때 한 프로세스에게 우선권을 주려면 더 많은 티켓을 줘서 당첨될 확률을 높여주는 것이다. Lottery 알고리즘을 구현할 때 가장 신경 쓴 부분이 바로 이 부분인데, 너무 자주 context switch가 발생할 수 있으므로, 이미 수행 중인 프로세스에게는 3장의 티켓을 부여해서 ready queue의 프로세스들보다 당첨 확률을 3배 높게 설정해주었다. 또 lottery ticket을 매번 부여하는 것은 결국 거의 매 시점에서 ready queue의 각 인덱스를 지닌 프로세스가 달라지므로 ready queue의 인덱스를 티켓으로 사용하고 ready queue 인덱스 밖의 범위의 세가지 숫자는 수행 중인 프로세스의 티켓으로 사용하였다.



## 3-4. Multi-level queues with aging 구현 소개

### (1) 전반적인 구현 설명

본 CPU scheduling simulator에서는 2 level의 ready queue와 aging 기법을 도입해서 MLFQ를 구현했다. Level 1 ready queue 내에서는 non-preemptive priority 알고리즘을, Level 2 ready queue에서는 기본적인 non-preemptive 방식의 FCFS 알고리즘을 사용한다. 두 ready queue 사이의 scheduling은 매번 0부터 9 범위의 임의의 값을 선택해서 8이나 9가 선택되면 L2의 ready queue를, 이외의 숫자들에 대해서는 L1의 ready queue를 선택하도록 해서 Level 1 ready queue가 CPU 시간의 80%를 사용하고 Level 2 ready queue가 CPU 시간의 20%를 사용

하도록 구현했다.

Aging 기법은 CPU를 연속적으로 4번의 시간동안 사용한 프로세스의 priority를 2씩 낮춰주고, ready queue에서 5 이상의 시간동안 대기한 프로세스의 priority를 3 증가시켜주었다. 매 simulation의 순간에, priority가 5이하인 프로세스는 L2 ready queue로, 6이상인 프로세스는 L1 ready queue로 재배정해서 ready queue 사이의 upgrade와 degrade가 자유롭도록 구현했다.

참고로 priority는 1에서 10 범위 내의 숫자로 부여되며, 숫자가 작을수록 높은 우선순위를 의미하고, 모든 프로세스가 생성된 직후 ready queue의 초기 배정에도 위와 같은 기준을 적용하여 priority가 5이하인 프로세스와 6이상인 프로세스를 나눠 각 L2, L1 ready queue에 넣어 주었다.

## (2) 새로 도입한 변수

프로세스의 내부 정보와 관련하여 starvation time과 sequential time, 그리고 origin이라는 변수를 추가해주었다.

Starvation time은 0으로 초기화되며, 프로세스가 ready queue에서 대기한 시간을 측정해서 이 시간이 5가 되면 해당 프로세스의 priority를 증가시켜주고 다시 이 변수를 초기화한다.

Sequential time은 degrade를 위한 변수이며 한번에 CPU를 얼마나 오랫동안 차지하는지를 측정하는 변수로, 마찬가지로 0으로 초기화되고 그 값이 4 이상이 되면 해당 프로세스의 priority를 낮춰주고 다시 이 변수를 초기화한다.

Origin은 해당 프로세스가 running 혹은 waiting queue에서 다시 ready queue로 돌아올 때, 직전에 있었던 ready queue로 돌아오도록 구현하기 위해서 도입했으며, 각 프로세스는 ready queue를 처음 배정받을 때와, 옮겨 다닐 때 이 origin 변수에 1 또는 2를 기록하게 된다.

## (3) Simulation 모듈의 Pseudo code

매 시점을 인자로 simulation 모듈에 넘겨주며 이는 0시간으로부터 이 인자만큼의 시간이 지난 시점에서의 simulation 루프를 의미한다. Computation 시작 시간과 종료 시간을 초기화 해준다.

```
for(모든 프로세스가 종료 OR 최대 시간){
```

```
    1. job scheduling
    for(job queue의 모든 프로세스){
        if(arrival time이 된 프로세스){
            if(priority>5){
                origin = 1;
                insert into L1 ReadyQueue
            }
            else{
                origin = 2;
                insert into L2 ReadyQueue
            }
        }
    }

    2. 재배치 upgrade & degrade
    for(L1 Ready queue의 모든 프로세스){
        if(priority<=5){
            origin = 2;
            move 프로세스 from L1 to L2 ReadyQueue
        }
    }
    for(L2 Ready queue의 모든 프로세스){
        if(priority>5){
            origin = 1;
            move 프로세스 from L2 to L1 ReadyQueue
        }
    }
```

```
3. Ready queue 간 scheduling
0~9 범위의 임의의 값 생성
if(0<=값<=7) L1 ReadyQueue에서 select
else L2 ReadyQueue에서 select
```

```
4. 선택된 ready queue에서 CPU scheduling
prevProcess = 수행 중이던 프로세스
thisProcess = 새롭게 선택된 프로세스

thisProcess의 starvation_time = 0
if(prevProcess != thisProcess){
    prevProcess의 sequential_time = 0
    move prevProcess from running queue to origin ready queue
    insert this Process to running queue
    if(thisProcess가 CPU를 처음 할당 받음)
        response_time = 현 시점 - arrival_time
}
```

```
5. ready queue 내 프로세스 처리
for(L1과 L2의 모든 프로세스){
    starvation_time++; waiting_time++; turnaround_time++;
    // aging - up
    if(starvation_time>=5)
        priority에 +3;
}
```

```

6. waiting queue 내 프로세스 처리
for(waiting queue의 모든 프로세스){
    turnaround_time++; IO_remaining_time--;
    if(IO_remaining_time==0)
        move to origin ReadyQueue
}

7. running queue 프로세스 처리
if(CPU idle){
    computation_idle++;
}
else {
    CPU_remaining_time--; progress++;
    turnaround_time++; sequential_time++;

    // aging-down
    if(sequential_time>4)
        priority를 -2;

    8. terminated 프로세스 처리
    if(CPU_remaining_time==0){
        move 프로세스 to termination queue
    }
    else if(progress==IO_start_time){
        move 프로세스 to waiting queue
    }
}
}

```

## 4장. CPU scheduling simulator 실행 결과

구현된 Simulator를 실행하면 생성된 프로세스 정보를 job queue를 출력함으로써 제시한다. 그리고 preemptive, non-preemptive를 포함해서 총 8개의 알고리즘에 대한 간트 차트와 evaluation 정보를 출력한다.

### 4-1. 생성된 프로세스 정보

다음은 실행 시 임의로 생성된 프로세스들이며, 8개의 알고리즘 모두 동일한 아래의 프로세스 정보를 job queue에 로딩해서 scheduling 한다.

총 프로세스 수: 5					
pid	priority	arrival_time	CPU burst	IO burst	IOStartTime
1	5	6	7	2	4
2	3	2	7	3	2
3	2	0	5	5	2
4	3	3	12	4	1
5	5	4	5	5	3

## 4-2. 알고리즘 별 simulation 결과

4-1의 프로세스 정보를 바탕으로 8가지의 CPU scheduling 알고리즘을 적용하여 성능을 측정하였다. Multiprogramming 환경에서 프로세스들이 어떤 식으로 수행되는지 알아보기 위해 Gantt Chart를 출력했는데, Tx는 0시간으로부터 x만큼의 시간이 흐른 x 시점을 의미하며 해당 시간마다 idle 혹은 running, IO start, IO complete, terminated 등의 프로세스의 상태를 '(pid: x) -> State'와 같은 상태로 보여준다. 먼저 각각의 알고리즘에 대한 출력 결과를 살펴보고, 종합하여 evaluation에 대한 비교분석은 5장에서 다루도록 한다.

### (1) Non-preemptive FCFS

```
<Non-preemptive FCFS Algorithm>
T0: (pid: 3) -> running
T1: (pid: 3) -> running -> IO request
T2: (pid: 2) -> running
T3: (pid: 2) -> running -> IO request
T4: (pid: 4) -> running -> IO request
T5: (pid: 5) -> running
T6: (pid: 3) -> IO complete, (pid: 2) -> IO complete, (pid: 5) -> running
T7: (pid: 5) -> running -> IO request
T8: (pid: 4) -> IO complete, (pid: 3) -> running
T9: (pid: 3) -> running
T10: (pid: 3) -> running -> terminated
T11: (pid: 2) -> running
T12: (pid: 5) -> IO complete, (pid: 2) -> running
T13: (pid: 2) -> running
T14: (pid: 2) -> running
T15: (pid: 2) -> running -> terminated
T16: (pid: 4) -> running
T17: (pid: 4) -> running
T18: (pid: 4) -> running
T19: (pid: 4) -> running
T20: (pid: 4) -> running
T21: (pid: 4) -> running
T22: (pid: 4) -> running
T23: (pid: 4) -> running
T24: (pid: 4) -> running
T25: (pid: 4) -> running
T26: (pid: 4) -> running -> terminated
T27: (pid: 5) -> running
T28: (pid: 5) -> running -> terminated
T29: (pid: 1) -> running
T30: (pid: 1) -> running
T31: (pid: 1) -> running

T32: (pid: 1) -> running -> IO request
T33: idle
T34: (pid: 1) -> IO complete, idle
T35: (pid: 1) -> running
T36: (pid: 1) -> running
T37: (pid: 1) -> running -> terminated
=====
(pid: 3)
waiting time = 1, turnaround time = 11, response time = 0
=====
(pid: 2)
waiting time = 4, turnaround time = 14, response time = 0
=====
(pid: 4)
waiting time = 8, turnaround time = 24, response time = 1
=====
(pid: 5)
waiting time = 15, turnaround time = 25, response time = 1
=====
(pid: 1)
waiting time = 23, turnaround time = 32, response time = 23
=====
start time: 0 / end time: 37 / CPU utilization : 94.59%
Average waiting time: 10.20
Average turnaround time: 21.20
Average response time: 5.00
Completed: 5
=====
```

프로세스들이 ready queue에 먼저 오는 arrival time 순서대로 잘 수행됨을 확인할 수 있다.

### (2) Preemptive FCFS

```

<Preemptive FCFS Algorithm>
T0: (pid: 3) -> running
T1: (pid: 3) -> running -> IO request
T2: (pid: 2) -> running
T3: (pid: 2) -> running -> IO request
T4: (pid: 4) -> running -> IO request
T5: (pid: 5) -> running
T6: (pid: 3) -> IO complete, (pid: 2) -> IO complete, (pid: 5) -> running
!preemption is detected!
T7: (pid: 3) -> running
T8: (pid: 4) -> IO complete, (pid: 3) -> running
T9: (pid: 3) -> running -> terminated
T10: (pid: 2) -> running
T11: (pid: 2) -> running
T12: (pid: 2) -> running
T13: (pid: 2) -> running
T14: (pid: 2) -> running -> terminated
T15: (pid: 4) -> running
T16: (pid: 4) -> running
T17: (pid: 4) -> running
T18: (pid: 4) -> running
T19: (pid: 4) -> running
T20: (pid: 4) -> running
T21: (pid: 4) -> running
T22: (pid: 4) -> running
T23: (pid: 4) -> running

T24: (pid: 4) -> running
T25: (pid: 4) -> running -> terminated
T26: (pid: 5) -> running -> IO request
T27: (pid: 1) -> running
T28: (pid: 1) -> running
T29: (pid: 1) -> running -> IO request
T30: (pid: 1) -> running -> IO request
T31: (pid: 5) -> IO complete, idle
T32: (pid: 1) -> IO complete, (pid: 5) -> running
T33: (pid: 5) -> running -> terminated
T34: (pid: 1) -> running
T35: (pid: 1) -> running
T36: (pid: 1) -> running -> terminated
=====
(pid: 3)
waiting time = 0, turnaround time = 10, response time = 0
=====
(pid: 2)
waiting time = 3, turnaround time = 13, response time = 0
=====
(pid: 4)
waiting time = 7, turnaround time = 23, response time = 1
=====
(pid: 5)
waiting time = 20, turnaround time = 30, response time = 1
=====
(pid: 1)
waiting time = 22, turnaround time = 31, response time = 21
=====
start time: 0 / end time: 36 / CPU utilization : 97.22%
Average waiting time: 10.40
Average turnaround time: 21.40
Average response time: 4.60
Completed: 5

```

FCFS에 preemptive 기법이 잘 적용됨을 확인할 수 있는데, non-preemptive FCFS에서 수행 중인 프로세스가 바뀌지 않은 부분에 preemptioin이 감지되었다고 출력이 되었고, 실제로 확인을 해보면 preemptioin이 되어야 할 상황으로 context switch가 이뤄졌음을 알 수 있다.

### (3) Non-Preemptive SJF

```

<Non-preemptive SJF Algorithm>
T0: (pid: 3) -> running
T1: (pid: 3) -> running -> IO request
T2: (pid: 2) -> running
T3: (pid: 2) -> running -> IO request
T4: (pid: 5) -> running
T5: (pid: 5) -> running
T6: (pid: 3) -> IO complete, (pid: 2) -> IO complete, (pid: 5) -> running -> IO request
T7: (pid: 3) -> running
T8: (pid: 3) -> running
T9: (pid: 3) -> running -> terminated
T10: (pid: 2) -> running
T11: (pid: 5) -> IO complete, (pid: 2) -> running
T12: (pid: 2) -> running
T13: (pid: 2) -> running
T14: (pid: 2) -> running -> terminated
T15: (pid: 5) -> running
T16: (pid: 5) -> running -> terminated
T17: (pid: 1) -> running
T18: (pid: 1) -> running
T19: (pid: 1) -> running
T20: (pid: 1) -> running -> IO request
T21: (pid: 4) -> running -> IO request
T22: (pid: 1) -> IO complete, idle
T23: (pid: 1) -> running
T24: (pid: 1) -> running
T25: (pid: 4) -> IO complete, (pid: 1) -> running -> terminated
T26: (pid: 4) -> running
T27: (pid: 4) -> running
T28: (pid: 4) -> running
T29: (pid: 4) -> running
T30: (pid: 4) -> running
T31: (pid: 4) -> running

T32: (pid: 4) -> running
T33: (pid: 4) -> running
T34: (pid: 4) -> running
T35: (pid: 4) -> running
T36: (pid: 4) -> running -> terminated
=====
(pid: 3)
waiting time = 0, turnaround time = 10, response time = 0
=====
(pid: 2)
waiting time = 3, turnaround time = 13, response time = 0
=====
(pid: 5)
waiting time = 3, turnaround time = 13, response time = 0
=====
(pid: 1)
waiting time = 11, turnaround time = 20, response time = 11
=====
(pid: 4)
waiting time = 18, turnaround time = 34, response time = 18
=====
start time: 0 / end time: 36 / CPU utilization : 97.22%
Average waiting time: 7.00
Average turnaround time: 18.00
Average response time: 5.80
Completed: 5
=====
```

Next CPU burst time 즉, remaining CPU time에 근거해서 잘 수행되고 있음을 확인할 수 있다.

#### (4) Preemptive SJF

```
<Preemptive SJF Algorithm>
T0: (pid: 3) -> running
T1: (pid: 3) -> running -> IO request
T2: (pid: 2) -> running
T3: (pid: 2) -> running -> IO request
T4: (pid: 5) -> running
T5: (pid: 5) -> running
T6: (pid: 3) -> IO complete, (pid: 2) -> IO complete, (pid: 5) -> running -> IO request
T7: (pid: 3) -> running
T8: (pid: 3) -> running
T9: (pid: 3) -> running -> terminated
T10: (pid: 2) -> running
T11: (pid: 5) -> IO complete, (pid: 2) -> running
'preemption is detected!
T12: (pid: 5) -> running
T13: (pid: 5) -> running -> terminated
T14: (pid: 2) -> running
T15: (pid: 2) -> running
T16: (pid: 2) -> running -> terminated
T17: (pid: 1) -> running
T18: (pid: 1) -> running
T19: (pid: 1) -> running
T20: (pid: 1) -> running -> IO request
T21: (pid: 4) -> running -> IO request
T22: (pid: 1) -> IO complete, idle
T23: (pid: 1) -> running
T24: (pid: 1) -> running
T25: (pid: 4) -> IO complete, (pid: 1) -> running -> terminated
T26: (pid: 4) -> running
T27: (pid: 4) -> running
T28: (pid: 4) -> running
T29: (pid: 4) -> running
T30: (pid: 4) -> running

T31: (pid: 4) -> running
T32: (pid: 4) -> running
T33: (pid: 4) -> running
T34: (pid: 4) -> running
T35: (pid: 4) -> running
T36: (pid: 4) -> running -> terminated
=====
(pid: 3)
waiting time = 0, turnaround time = 10, response time = 0
=====
(pid: 5)
waiting time = 0, turnaround time = 10, response time = 0
=====
(pid: 2)
waiting time = 5, turnaround time = 15, response time = 0
=====
(pid: 1)
waiting time = 11, turnaround time = 20, response time = 11
=====
(pid: 4)
waiting time = 18, turnaround time = 34, response time = 18
=====
start time: 0 / end time: 36 / CPU utilization : 97.22%
Average waiting time: 6.80
Average turnaround time: 17.80
Average response time: 5.80
Completed: 5
=====
```

Preemption이 잘 적용되어 remaining CPU burst time을 기준으로 또, 두번째로는 FCFS의 기준으로 더 적절한 프로세스가 있을 경우, 이미 실행 중인 프로세스에서 CPU를 선점해서 preemption이 나타나는 것을 확인할 수 있다.

#### (5) Non-preemptive Priority

```
<Non-preemptive Priority Algorithm>
T0: (pid: 3) -> running
T1: (pid: 3) -> running -> IO request
T2: (pid: 2) -> running
T3: (pid: 2) -> running -> IO request
T4: (pid: 4) -> running -> IO request
T5: (pid: 5) -> running
T6: (pid: 3) -> IO complete, (pid: 2) -> IO complete, (pid: 5) -> running
T7: (pid: 5) -> running -> IO request
T8: (pid: 4) -> IO complete, (pid: 3) -> running
T9: (pid: 3) -> running
T10: (pid: 3) -> running -> terminated
T11: (pid: 2) -> running
T12: (pid: 5) -> IO complete, (pid: 2) -> running
T13: (pid: 2) -> running
T14: (pid: 2) -> running
T15: (pid: 2) -> running -> terminated
T16: (pid: 4) -> running
T17: (pid: 4) -> running
T18: (pid: 4) -> running
T19: (pid: 4) -> running
T20: (pid: 4) -> running
T21: (pid: 4) -> running
T22: (pid: 4) -> running
T23: (pid: 4) -> running
T24: (pid: 4) -> running
T25: (pid: 4) -> running
T26: (pid: 4) -> running -> terminated
T27: (pid: 5) -> running
T28: (pid: 5) -> running -> terminated
T29: (pid: 1) -> running
T30: (pid: 1) -> running
T31: (pid: 1) -> running

T32: (pid: 1) -> running -> IO request
T33: idle
T34: (pid: 1) -> IO complete, idle
T35: (pid: 1) -> running
T36: (pid: 1) -> running
T37: (pid: 1) -> running -> terminated
=====
(pid: 3)
waiting time = 1, turnaround time = 11, response time = 0
=====
(pid: 2)
waiting time = 4, turnaround time = 14, response time = 0
=====
(pid: 4)
waiting time = 8, turnaround time = 24, response time = 1
=====
(pid: 5)
waiting time = 15, turnaround time = 25, response time = 1
=====
(pid: 1)
waiting time = 23, turnaround time = 32, response time = 23
=====
start time: 0 / end time: 37 / CPU utilization : 94.59%
Average waiting time: 10.20
Average turnaround time: 21.20
Average response time: 5.00
Completed: 5
=====
```

이 알고리즘의 성능은 임의의 priority 배정에 영향을 많이 받는다. 위의 간트 차트를 확인해본 결과, priority 값이 낮은 순서대로 높은 우선순위로 해석하여 scheduling이 잘 작동하고 있다.

## (6) Preemptive Priority

```
<Preemptive Priority Algorithm>
T0: (pid: 3) -> running
T1: (pid: 3) -> running -> IO request
T2: (pid: 2) -> running
T3: (pid: 2) -> running -> IO request
T4: (pid: 4) -> running -> IO request
T5: (pid: 5) -> running
T6: (pid: 3) -> IO complete, (pid: 2) -> IO complete, (pid: 5) -> running
!preemption is detected!
T7: (pid: 3) -> running
T8: (pid: 4) -> IO complete, (pid: 3) -> running
T9: (pid: 3) -> running -> terminated
T10: (pid: 2) -> running
T11: (pid: 2) -> running
T12: (pid: 2) -> running
T13: (pid: 2) -> running
T14: (pid: 2) -> running -> terminated
T15: (pid: 4) -> running
T16: (pid: 4) -> running
T17: (pid: 4) -> running
T18: (pid: 4) -> running
T19: (pid: 4) -> running
T20: (pid: 4) -> running
T21: (pid: 4) -> running
T22: (pid: 4) -> running
T23: (pid: 4) -> running
T24: (pid: 4) -> running
T25: (pid: 4) -> running -> terminated
T26: (pid: 5) -> running -> IO request
T27: (pid: 1) -> running
T28: (pid: 1) -> running
T29: (pid: 1) -> running
T30: (pid: 1) -> running -> IO request

T31: (pid: 5) -> IO complete, idle
T32: (pid: 1) -> IO complete, (pid: 5) -> running
T33: (pid: 5) -> running -> terminated
T34: (pid: 1) -> running
T35: (pid: 1) -> running
T36: (pid: 1) -> running -> terminated
=====
(pid: 3)
waiting time = 0, turnaround time = 10, response time = 0
=====
(pid: 2)
waiting time = 3, turnaround time = 13, response time = 0
=====
(pid: 4)
waiting time = 7, turnaround time = 23, response time = 1
=====
(pid: 5)
waiting time = 20, turnaround time = 30, response time = 1
=====
(pid: 1)
waiting time = 22, turnaround time = 31, response time = 21
=====
start time: 0 / end time: 36 / CPU utilization : 97.22%
Average waiting time: 10.40
Average turnaround time: 21.40
Average response time: 4.60
Completed: 5
=====
```

앞서 non-preemptive priority에서 발생하지 않았던 preemption이 적절히 발생하고 있으며 해당 프로세스 정보에서는 non-preemptive할 때에 비해 CPU utilization이 향상된 것을 확인할 수 있다.

## (7) RR (time quantum=3)

```

<Round Robin Algorithm (time quantum: 3)>
T0: (pid: 3) -> running
T1: (pid: 3) -> running -> IO request
T2: (pid: 2) -> running
T3: (pid: 2) -> running -> IO request
T4: (pid: 4) -> running -> IO request
T5: (pid: 5) -> running
T6: (pid: 3) -> IO complete, (pid: 2) -> IO complete, (pid: 5) -> running
T7: (pid: 5) -> running -> IO request
T8: (pid: 4) -> IO complete, (pid: 1) -> running
T9: (pid: 1) -> running
T10: (pid: 1) -> running
T11: (pid: 3) -> running
T12: (pid: 5) -> IO complete, (pid: 3) -> running
T13: (pid: 3) -> running -> terminated
T14: (pid: 2) -> running
T15: (pid: 2) -> running
T16: (pid: 2) -> running
T17: (pid: 4) -> running
T18: (pid: 4) -> running
T19: (pid: 4) -> running
T20: (pid: 1) -> running -> IO request
T21: (pid: 5) -> running
T22: (pid: 1) -> IO complete, (pid: 5) -> running -> terminated
T23: (pid: 2) -> running
T24: (pid: 2) -> running -> terminated
T25: (pid: 4) -> running
T26: (pid: 4) -> running
T27: (pid: 4) -> running
T28: (pid: 1) -> running
T29: (pid: 1) -> running
T30: (pid: 1) -> running -> terminated
T31: (pid: 4) -> running

T32: (pid: 4) -> running
T33: (pid: 4) -> running
T34: (pid: 4) -> running
T35: (pid: 4) -> running -> terminated
=====
(pid: 3)
waiting time = 4, turnaround time = 14, response time = 0
=====
(pid: 5)
waiting time = 9, turnaround time = 19, response time = 1
=====
(pid: 2)
waiting time = 13, turnaround time = 23, response time = 0
=====
(pid: 1)
waiting time = 16, turnaround time = 25, response time = 2
=====
(pid: 4)
waiting time = 17, turnaround time = 33, response time = 1
=====
start time: 0 / end time: 35 / CPU utilization : 100.00%
Average waiting time: 11.80
Average turnaround time: 22.80
Average response time: 0.80
Completed: 5
=====
```

Ready queue에 아무런 프로세스도 있지 않는 이상, time quantum을 설정된 3보다 많은 시간 연속해서 수행하는 경우가 없고, 3번 수행을 연속으로 하면 다른 프로세스가 CPU를 차지하는 것을 확인할 수 있다. 또 앞서 알고리즘의 구현에서 설명한 바와 같이 FCFS 알고리즘에 기반하여 new->ready, running->ready, waiting->ready로 한 시점에서 ready queue에 삽입이 되도록 설계했는데, 이 순서에 따라서 ready queue 내에서 circular하게 순서가 잘 유지되어 프로세스들이 돌아가면서 CPU를 할당 받음을 확인할 수 있다. 또 이 임의의 프로세스 정보에 대해서는 CPU utilization과 ART이 매우 좋게 측정됨을 볼 수 있다.

## (8) Lottery

```

<Lottery Algorithm>
Congratulations! Process w/ pid 3 won!
T0: (pid: 3) -> running
Process w/ pid 3 won AGAIN!
T1: (pid: 3) -> running -> IO request
Congratulations! Process w/ pid 2 won!
T2: (pid: 2) -> running
Congratulations! Process w/ pid 4 won!
T3: (pid: 4) -> running -> IO request
Congratulations! Process w/ pid 5 won!
T4: (pid: 5) -> running
Process w/ pid 5 won AGAIN!
T5: (pid: 5) -> running
Congratulations! Process w/ pid 2 won!
T6: (pid: 3) -> IO complete, (pid: 2) -> running -> IO request
Congratulations! Process w/ pid 1 won!
T7: (pid: 4) -> IO complete, (pid: 1) -> running
Process w/ pid 1 won AGAIN!
T8: (pid: 1) -> running
Process w/ pid 1 won AGAIN!
T9: (pid: 2) -> IO complete, (pid: 1) -> running
Congratulations! Process w/ pid 5 won!
T10: (pid: 5) -> running -> IO request
Congratulations! Process w/ pid 1 won!
T11: (pid: 1) -> running -> IO request
Congratulations! Process w/ pid 4 won!
T12: (pid: 4) -> running
Congratulations! Process w/ pid 2 won!
T13: (pid: 1) -> IO complete, (pid: 2) -> running
Congratulations! Process w/ pid 4 won!
T14: (pid: 4) -> running
Process w/ pid 4 won AGAIN!
T15: (pid: 5) -> IO complete, (pid: 4) -> running

Process w/ pid 4 won AGAIN!
T16: (pid: 4) -> running
Congratulations! Process w/ pid 2 won!
T17: (pid: 2) -> running
Process w/ pid 2 won AGAIN!
T18: (pid: 2) -> running
Process w/ pid 2 won AGAIN!
T19: (pid: 2) -> running
Congratulations! Process w/ pid 4 won!
T20: (pid: 4) -> running
Process w/ pid 4 won AGAIN!
T21: (pid: 4) -> running
Process w/ pid 4 won AGAIN!
T22: (pid: 4) -> running
Process w/ pid 4 won AGAIN!
T23: (pid: 4) -> running
Process w/ pid 4 won AGAIN!
T24: (pid: 4) -> running
Process w/ pid 4 won AGAIN!
T25: (pid: 4) -> running
Process w/ pid 4 won AGAIN!
T26: (pid: 4) -> running -> terminated
Congratulations! Process w/ pid 2 won!
T27: (pid: 2) -> running -> terminated
Congratulations! Process w/ pid 5 won!
T28: (pid: 5) -> running
Process w/ pid 5 won AGAIN!
T29: (pid: 5) -> running -> terminated
Congratulations! Process w/ pid 3 won!
T30: (pid: 3) -> running
Congratulations! Process w/ pid 1 won!
T31: (pid: 1) -> running
Process w/ pid 1 won AGAIN!

T32: (pid: 1) -> running
Process w/ pid 1 won AGAIN!
T33: (pid: 1) -> running -> terminated
Congratulations! Process w/ pid 3 won!
T34: (pid: 3) -> running
Process w/ pid 3 won AGAIN!
T35: (pid: 3) -> running -> terminated
=====
(pid: 4)
waiting time = 8, turnaround time = 24, response time = 0
=====
(pid: 2)
waiting time = 16, turnaround time = 26, response time = 0
=====
(pid: 5)
waiting time = 16, turnaround time = 26, response time = 0
=====
(pid: 1)
waiting time = 19, turnaround time = 28, response time = 1
=====
(pid: 3)
waiting time = 26, turnaround time = 36, response time = 0
=====
start time: 0 / end time: 35 / CPU utilization : 100.00%
Average waiting time: 17.00
Average turnaround time: 28.00
Average response time: 0.20
Completed: 5
=====
```

Starvation을 막는 lottery 알고리즘은 역시 다른 알고리즘에 비해서 ART이 월등히 좋게 확인되었다. CPU utilization도 최대치임을 확인할 수 있다. 이는 CPU 활용률을 높이기 위해서 scheduling을 할 때, 티켓 배부 시 ready 및 running queue가 비는 경우의 추첨 티켓 배부를 달리 설계해서 최대한 CPU가 busy하도록 구현한 결과다.

#### .4-3. MLFQ의 simulation 결과

##### (1) 생성된 process

총 프로세스 수: 9					
pid	priority	arrival_time	CPU burst	IO burst	IOStartTime
1	7	0	11	5	9
2	6	7	6	1	5
3	3	7	6	0	2
4	1	5	12	1	9
5	2	5	3	5	2
6	5	4	7	4	4
7	6	6	4	5	3
8	4	12	4	0	1
9	1	2	9	5	4

## (2) Simulation 결과

T0, L1 : (pid: 1) -> running	T34, L1 : (pid: 4) -> running
T1, L1 : (pid: 1) -> running	T35, L1 : (pid: 4) -> running
T2, L1 : (pid: 1) -> running	T36, L1 : (pid: 4) -> running
T3, L1 : (pid: 1) -> running	T37, L1 : (pid: 4) -> running
T4, L2 : (pid: 1) -> running	T38, L1 : (pid: 4) -> running -> IO request
T5, L1 : (pid: 1) -> running	T39, L1 : (pid: 4) -> IO complete, idle
T6, L2 : (pid: 1) -> running	T40, L2 : (pid: 9) -> running
T7, L1 : (pid: 1) -> running	T41, L1 : (pid: 9) -> running
T8, L1 : (pid: 1) -> running -> IO request	T42, L2 : (pid: 9) -> running
T9, L1 : (pid: 7) -> running	T43, L2 : (pid: 9) -> running
T10, L1 : (pid: 7) -> running	T44, L1 : (pid: 9) -> running -> terminated
T11, L1 : (pid: 7) -> running -> IO request	T45, L2 : (pid: 6) -> running
T12, L1 : (pid: 2) -> running	T46, L1 : (pid: 6) -> running
T13, L1 : (pid: 1) -> IO complete, (pid: 2) -> running	T47, L1 : (pid: 6) -> running -> terminated
T14, L1 : (pid: 2) -> running	T48, L1 : idle
T15, L1 : (pid: 2) -> running	T49, L1 : idle
T16, L2 : (pid: 7) -> IO complete, (pid: 2) -> running -> IO request	T50, L1 : idle
T17, L1 : (pid: 2) -> IO complete, (pid: 7) -> running -> terminated	T51, L1 : idle
T18, L2 : (pid: 1) -> running	T52, L1 : idle
T19, L1 : (pid: 1) -> running -> terminated	T53, L1 : idle
T20, L1 : idle	T54, L1 : idle
T21, L1 : idle	T55, L1 : idle
T22, L2 : (pid: 9) -> running	T56, L1 : idle
T23, L1 : (pid: 9) -> running	T57, L2 : (pid: 5) -> running
T24, L1 : (pid: 9) -> running	T58, L1 : (pid: 5) -> running -> IO request
T25, L1 : (pid: 9) -> running -> IO request	T59, L2 : (pid: 4) -> running
T26, L2 : (pid: 6) -> running	T60, L1 : (pid: 4) -> running
T27, L2 : (pid: 6) -> running	T61, L1 : (pid: 4) -> running -> terminated
T28, L1 : (pid: 6) -> running	T62, L1 : idle
T29, L1 : (pid: 6) -> running -> IO request	T63, L1 : (pid: 5) -> IO complete, idle
T30, L2 : (pid: 9) -> IO complete, (pid: 4) -> running	T64, L1 : idle
T31, L2 : (pid: 4) -> running	T65, L2 : (pid: 5) -> running -> terminated
T32, L1 : (pid: 4) -> running	T66, L1 : idle
T33, L1 : (pid: 6) -> IO complete, (pid: 4) -> running	T67, L1 : idle

T68, L1 : idle	=====
T69, L2 : (pid: 3) -> running	(pid: 7)
T70, L2 : (pid: 3) -> running	waiting time = 3, turnaround time = 12, response time = 3
T71, L1 : (pid: 3) -> running	=====
T72, L1 : (pid: 3) -> running	(pid: 1)
T73, L1 : (pid: 3) -> running	waiting time = 4, turnaround time = 20, response time = 0
T74, L2 : (pid: 3) -> running -> terminated	=====
T75, L1 : idle	(pid: 9)
T76, L1 : idle	waiting time = 29, turnaround time = 43, response time = 20
T77, L1 : idle	=====
T78, L1 : idle	(pid: 6)
T79, L1 : idle	waiting time = 33, turnaround time = 44, response time = 22
T80, L1 : idle	=====
T81, L2 : (pid: 2) -> running -> terminated	(pid: 4)
T82, L1 : idle	waiting time = 44, turnaround time = 57, response time = 25
T83, L1 : idle	=====
T84, L1 : idle	(pid: 5)
T85, L1 : idle	waiting time = 53, turnaround time = 61, response time = 52
T86, L1 : idle	=====
T87, L1 : idle	(pid: 3)
T88, L1 : idle	waiting time = 62, turnaround time = 68, response time = 62
T89, L1 : idle	=====
T90, L1 : idle	(pid: 2)
T91, L2 : (pid: 8) -> running	waiting time = 68, turnaround time = 75, response time = 5
T92, L2 : (pid: 8) -> running	=====
T93, L1 : (pid: 8) -> running	
T94, L2 : (pid: 8) -> running -> terminated	

```

(pid: 8)
waiting time = 79, turnaround time = 83, response time = 79
=====
start time: 0 / end time: 94 / CPU utilization : 64.89%
Average waiting time: 41.67
Average turnaround time: 51.44
Average response time: 29.78
Completed: 9
=====
```

실제 간트 차트를 통해서 일정시간 이상 CPU를 점유한 프로세스는 priority가 하락해서 낮은 레벨의 ready queue로 이동했음을 확인할 수 있다. 출력된 시간 옆의 L1 혹은 L2는 해당 프로세스가 어느 ready queue에 속했는지 가장 최근의 정보를 보여주는 것이다. MLFQ의 경우 starvation을 막기 위한 방법이지만, CPU utilization이 매우 낮음을 확인할 수 있다. 이는 ready queue 간 scheduling 시 각 ready queue가 비어 있는지 여부에 대한 조건 확인이 없었고 또, 해당 프로세스 정보에서 많은 프로세스가 IO를 처리하는 시점이 겹치는 영향도 있다.

## 5장. 알고리즘 간의 evaluation 비교 분석

앞서서 simulator의 구현에서 알고리즘 별로 evaluation을 측정하기 위해서 여러 변수를 도입했었다. 결과적으로 계산된 evaluation criteria로는 AWT, ATT, ART, CPU utilization이 4가지를 사용한다.

### 5-1. evaluation criteria에 대한 간단한 소개

#### (1) Average Waiting Time (AWT)

Waiting Time은 프로세스가 Waiting Queue에 상주하는 시간은 고려하지 않고 단지 Ready Queue에 상주하는 시간만을 포함한다.

#### (2) Average Turnaround Time (ATT)

Job Scheduler에 의해 프로세스가 처음 Ready Queue에 도착한 순간부터 해당 프로세스의 CPU burst time이 소진되어 Terminated 상태가 된 순간 Turnaround Time 증가를 멈춘다. 따라서 이는 결과적으로 프로세스가 완료된 시간에서 프로세스가 처음 도착한 시간을 뺀 것과 같고 CPU burst time + IO burst time + waiting time이라고 할 수 있다.

#### (3) Average Response Time (ART)

각 프로세스 별 response time을 평균을 취한 것인데, response time은 본 simulation에서

CPU를 한 프로세스가 처음으로 할당 받은 시점과 ready queue에 도착한 시점 즉, arrival time의 차이로 계산한다.

#### (4) CPU utilization

하나의 프로세스라도 ready queue에 도착한 순간부터 측정해서 모든 프로세스가 작업을 완료하는 시점까지의 시간 간격을 분모로 하고, 그 기간 중 CPU가 busy한 기간을 분자로 해서 CPU가 작업 시간 내에 얼마나 활용되는 가에 대한 지표이다. 이때 CPU가 busy한 기간은 분모에서 CPU가 idle할 때마다 1씩 증가해준 computation\_idle이라는 변수를 빼서 구한다.

### 5-2. 알고리즘 별 성능 평가 비교 분석

각 알고리즘 별 비교 분석을 위해 사용한 임의의 프로세스 정보는 다음과 같다. 이는 앞서 4장에서 실행 결과를 제시하기 위해 실행한 때의 프로세스 정보이다.

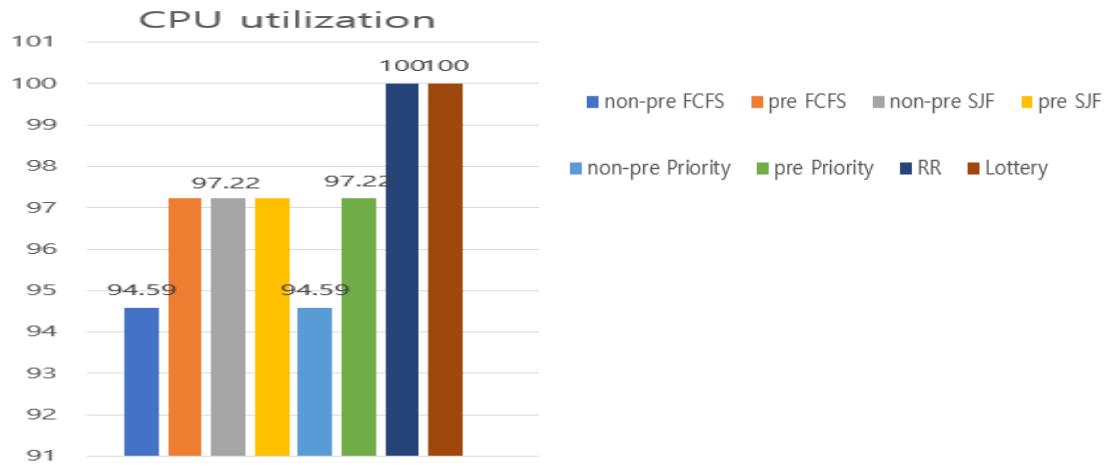
총 프로세스 수: 5					
pid	priority	arrival_time	CPU burst	IO burst	IStartTime
1	5	6	7	2	4
2	3	2	7	3	2
3	2	0	5	5	2
4	3	3	12	4	1
5	5	4	5	5	3

다음은 simulator를 실행시키면 가장 아래에 출력되는 알고리즘 별 evaluation 정보이다.

```
=====
<Non-preemptive FCFS Algorithm>
=====
start time: 0 / end time: 37 / CPU utilization : 94.59%
Average waiting time: 10.00
Average turnaround time: 21.00
Average response time: 5.00
Completed: 5
=====
<Preemptive FCFS Algorithm>
=====
start time: 0 / end time: 36 / CPU utilization : 97.22%
Average waiting time: 10.00
Average turnaround time: 21.00
Average response time: 4.00
Completed: 5
=====
<Non-preemptive SJF Algorithm>
=====
start time: 0 / end time: 36 / CPU utilization : 97.22%
Average waiting time: 7.00
Average turnaround time: 18.00
Average response time: 5.00
Completed: 5
=====
<Preemptive SJF Algorithm>
=====
start time: 0 / end time: 36 / CPU utilization : 97.22%
Average waiting time: 6.00
Average turnaround time: 17.00
Average response time: 5.00
Completed: 5
=====
<Non-preemptive Priority Algorithm>
=====
start time: 0 / end time: 37 / CPU utilization : 94.59%
Average waiting time: 10.00
Average turnaround time: 21.00
Average response time: 5.00
Completed: 5
=====
<Preemptive Priority Algorithm>
=====
start time: 0 / end time: 36 / CPU utilization : 97.22%
Average waiting time: 10.00
Average turnaround time: 21.00
Average response time: 4.00
Completed: 5
=====
<Round Robin Algorithm>
=====
start time: 0 / end time: 35 / CPU utilization : 100.00%
Average waiting time: 11.00
Average turnaround time: 22.00
Average response time: 0.00
Completed: 5
=====
<Lottery Algorithm>
=====
start time: 0 / end time: 35 / CPU utilization : 100.00%
Average waiting time: 17.00
Average turnaround time: 28.00
Average response time: 0.00
Completed: 5
=====
```

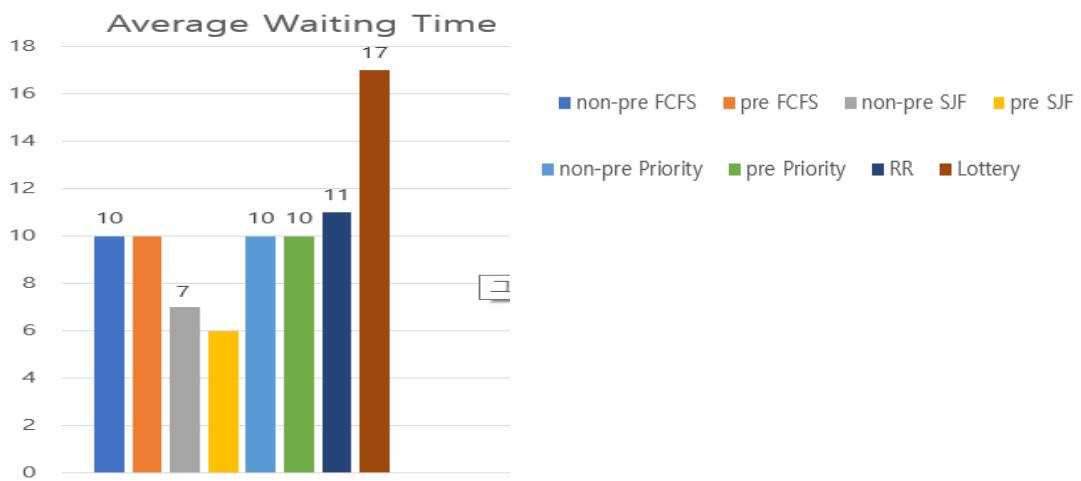
Evaluation criteria를 기준을 비교 분석한 아래의 내용은 임의로 생성된 프로세스에 대한 scheduling이므로 다른 프로세스 정보로 simulation을 하면, 다른 결과가 나올 수 있다.

### (1) CPU utilization



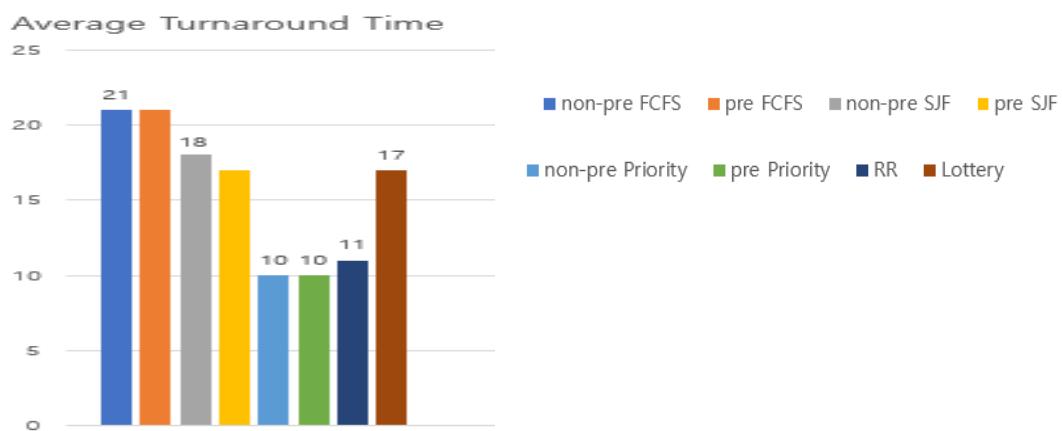
이 데이터에 대한 simulation의 결과를 CPU utilization을 기준으로 알고리즘 간의 비교를 하자면, 우선 RR과 Lottery방식은 CPU utilization이 100%로 가장 좋은 효율을 보였다. 이때, Lottery는 ready queue와 run queue가 비었을 때를 각각 고려하여 추첨 범위를 정해줬기 때문에 효율적인 결과로 이어졌다고 볼 수 있다. 전반적으로 non-preemptive보다 preemptive한 방식의 알고리즘이 더 높은 효율성을 결과로 보여줌을 확인할 수 있고, SJF의 경우 먼저 도착한 프로세스가 IO 작업을 처리하는 동안 다른 프로세스가 ready queue에 도착하지 않았기 때문에 효율이 낮은 결과를 보임을 생각할 수 있다.

### (2) AWT



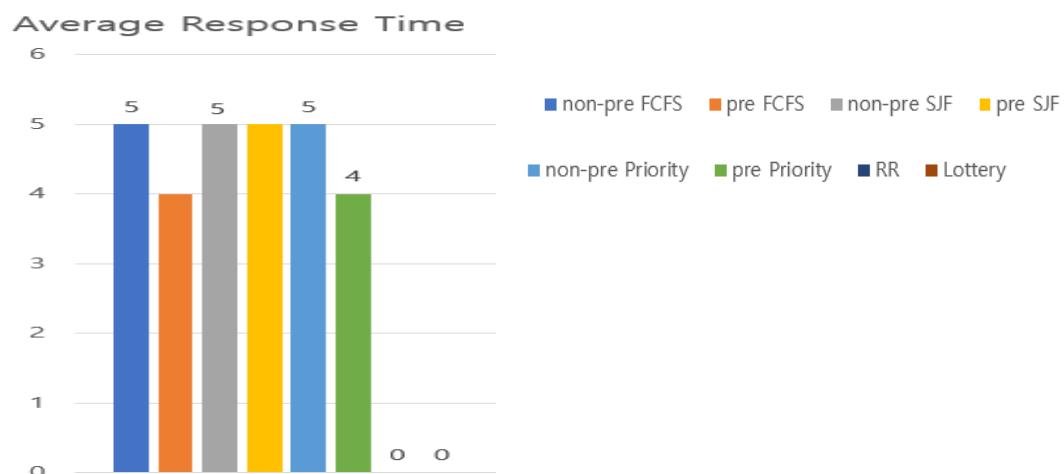
AWT는 각 프로세스들이 ready queue에서 대기하는 시간을 평균 낸 것인데, 이 값은 적을 수록 더 좋은 효율성의 결과라고 해석해야 한다. 이 프로세스 데이터에 대한 결과는 preemptive SJF, non-preemptive SJF 알고리즘이 가장 최적화되어 있음을 확인할 수 있다. 실제로, 강의 자료의 chapter 5를 참고하면 SJF 알고리즘이 AWT이 가장 적다는 점에서 optimal하다고 소개되어 있다. 가장 AWT의 관점에서 효율성이 낮은 알고리즘은 Lottery 알고리즘이다. 이는 당첨의 개념으로 확률이 동일하게 주어졌더라도, 운에 의해 당첨이 안된 프로세스가 있을 수 있기 때문에 AWT이 다른 알고리즘에 비해 큰 값을 얻는다고 볼 수 있다.

### (3) ATT



ATT는 각 프로세스가 ready queue에 들어온 시점부터 작업을 종료할 때까지의 각 프로세스의 turnaround time의 평균값이다. 이 데이터의 경우 priority 알고리즘이 ATT가 가장 적으므로 좋은 효율을 보여준다. 그리고 FCFS가 가장 낮은 효율을 보여주는데 이는 CPU utilization과 관련해서 idle한 시간이 많은 영향이 있을 것이다.

### (4) ART



다른 evaluation metric에 비해서 알고리즘 간의 차이가 평이하다. ready queue에 들어온 시점으로부터 처음으로 CPU를 할당 받기까지의 시간 간격을 평균 낸 값이다. 특이한 점은 RR과 Lottery 알고리즘의 경우 그 값이 0이라는 것인데, 프로세스가 ready queue로 진입하고 나서 바로 CPU를 할당 받았다고 해석할 수 있으며, 이는 이 데이터의 경우 프로세스들의 arrival time에 어느정도 간격이 있기에 가능한 결과인 것 같다.

### 5-3. 결론

위의 4가지의 evaluation metric을 종합했을 때, 해당 임의의 프로세스 데이터에 대해서는 RR (time quantum = 3)인 알고리즘이 가장 우수한 효율성을 보여주었다. 우선 RR 알고리즘의 특성이 각 프로세스의 starvation을 막기 위해서 FCFS의 방식과 preemption, time quantum 개념을 도입해서 circular order로 프로세스를 수행시키기 때문에, 모든 프로세스에 대해서 평균적으로 좋은 evaluation 결과를 낸 것 같다.

## 6장. 프로젝트 소감 및 추후 발전 계획

직접 개념으로만 배운 CPU scheduling simulator를 구현해보면서, 가장 크게 느낀 점은 구현을 할 때 모듈 간의 구성이 가장 중요하다는 것이다. Simulator를 구현하기 위해서는 초기에 자원을 초기화하고, 프로세스 데이터를 생성하고, simulation을 돌리고, 평가 내역을 출력하는 등의 모듈이 필요했는데, 이 모듈 간의 순서가 매우 짜임새 있어야 모든 실행이 오류 없이 돌아가기 때문이다. 또한 예외 처리에 신경을 써야 원하는 방식대로 작동한다는 것을 알게 되었다.

구현 측면에서 아쉬운 점은 크게 gantt chart와 IO 구현 측면에서 있다. Gantt chart를 참고한 논문처럼 가시적으로 graphic UI를 통해서 표현하지 못했기 때문에 user 입장에서 가독성이 높은 gantt chart는 출력되지 않았다. IO 구현과 관련해서는 2가지의 한계점이 있는데, 우선 IO 작업이 한 프로세스 내에서도 여러 번 발생할 수 있는데 이를 고려하지 않고 발생하지 않거나 1번만 발생하는 경우만을 고려해 주었다는 것이다. 이를 보완하기 위해서 여러 번의 IO 작업이 한 프로세스에 발생하는 경우를 생각해보았는데, 그럴 경우 다른 IO device에 대한 작업이 겹치는지, 같은 IO device로 처리를 해주고자 하는지, IO device의 개수와 종류 및 scheduling까지 고려해 주어야 한다는 것을 알게 되었다.

이렇게 gantt chart와 IO 처리에 대한 한계점을 보완하면 더 현실적인 CPU scheduling simulator를 구현할 수 있다. 구현한 simulator를 추후에 더 보완해서 이러한 부족한 점들을 극복해볼 것이다.

## <부록 - 참고 문헌>

A.Silberschatz, P.Galvin, G.Gagne (2013). Operating System Concepts[9<sup>th</sup> Ed.]

Suranauwarat, S. (2007, October). A CPU scheduling algorithm simulator. In *2007 37th annual frontiers in education conference-global engineering: knowledge without borders, opportunities without passports* (pp. F2H-19). IEEE.

1. 알고리즘 flow chart 이미지, 검색일 2019-06-07

[https://www.semanticscholar.org/paper/Simulation-of-First-Come-First-Served-\(FCFS\)-and-Xoxa-Zotaj/7cc4cb63d7ef6f605ebf9ed74b69d78b64effff6](https://www.semanticscholar.org/paper/Simulation-of-First-Come-First-Served-(FCFS)-and-Xoxa-Zotaj/7cc4cb63d7ef6f605ebf9ed74b69d78b64effff6)

[https://www.slideshare.net/Shreya\\_lovi/process-scheduling-in-light-weight-weight-and-heavy-weight-processes](https://www.slideshare.net/Shreya_lovi/process-scheduling-in-light-weight-weight-and-heavy-weight-processes)

2. 기존 simulator 구현 코드, 검색일 2019-06-07

<https://github.com/arkainoh/CPU-Scheduling-Simulator/blob/master/CPUScheduler.c>