CPU Scheduling Simulator

: OS Term Project

제출일	2023.06.11	전공	컴퓨터소프트웨어공학과
과목	운영체제	학번	20204009
담당교수	김대영 교수님	이름	김민아

목 차

1. 개요

1.1 프로그램 개요

작성자 : 김민아
학번 : 20204009
프로그램명 : CPU Schedular Simulator
GitHub : https://github.com/Ogu1208/OS-CPU-Schedular-Simulator.git

1.2 개발 환경

프로그램 동작 환경 : Windows
사용 컴파일러 : IDE : IntelliJ
사용 언어 : Java 11
GUI 환경 : Java Swing

1.3 요구 분석

- **FCFS, SJF, 비선점 Priority, 선점 Priority, RR, SRT, HRN**
- **입력** : (프로세스 수), 프로세스 ID, 도착시간, 서비스 시간, 우선 순위, 시간 할당량
- 우선순위 숫자가 작을수록 우선순위 높게
- **출력** : 간트 차트, 각 프로세스 별 대기 시간, 평균 대기 시간, 각 프로세스 별 응답 시간, 평균 응답 시간, 각 프로세스 별 반환 시간, 평균 반환 시간

1.4 파일 설명 및 실행법

<파일설명>

압축파일의 구성은 아래와 같다.

OS-CPU-Schedular-Simulator -> CPU 스케줄러의 프로젝트 폴더

➔ src 폴더 : java소스파일

➔ out > artifacts > OS_CPU_Schedular_Simulator_jar > OS-CPU-Schedular-Simulator.jar : 실행파일

OS-CPU-Schedular-Simulator.jar - 바로 가기 -> jar 실행파일의 바로가기 링크이다.

Term Project 최종 보고서.docx--> 본 파일

<실행법>

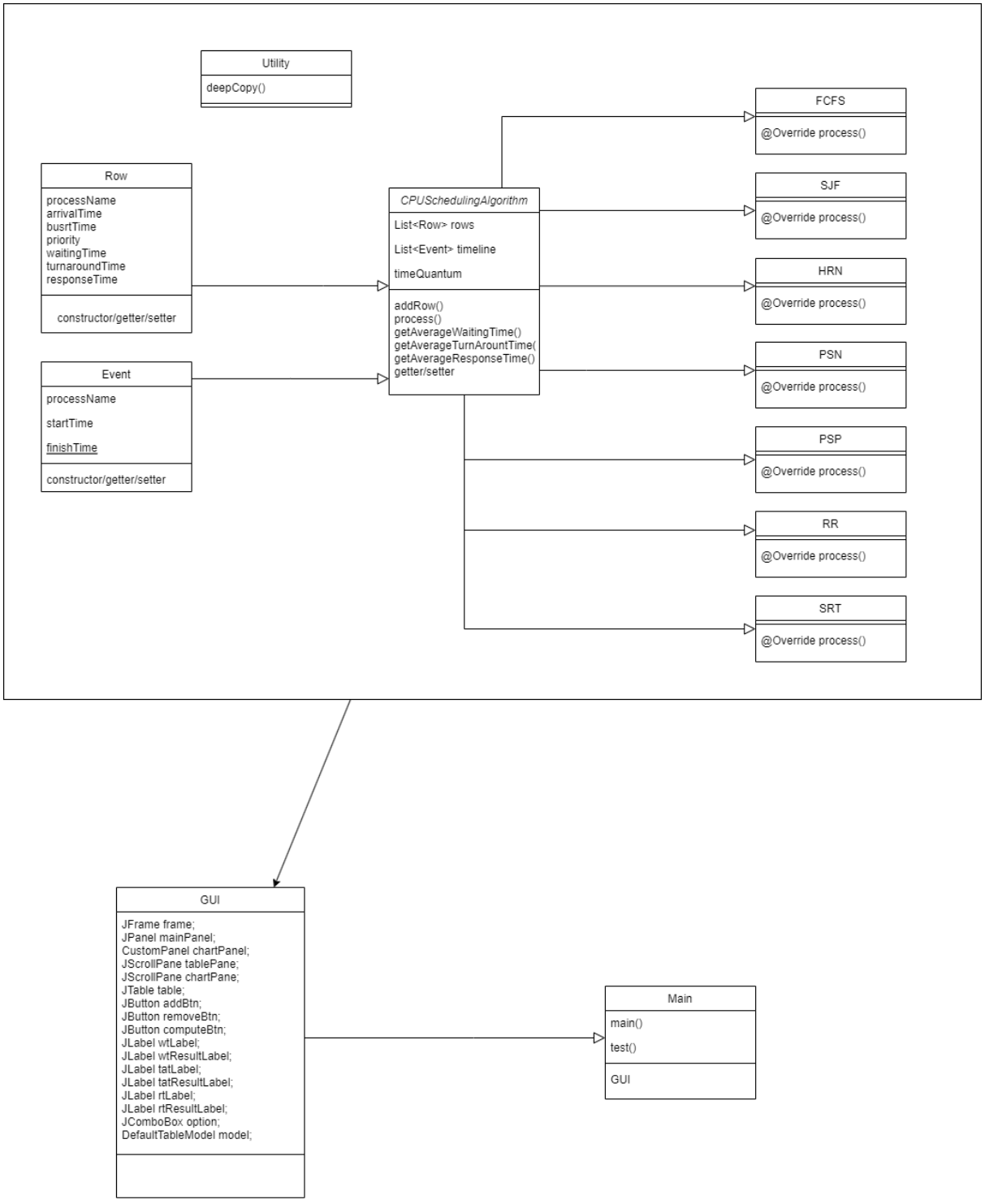
해당 프로젝트는 Java swing을 이용하여 구현 하였다.

JDK11 이상으로 압축 파일 안의 `OS-CPU-Schedular-Simulator.jar - 바로 가기` 를 실행한다.

2. 프로젝트 구성

2.1 함수, 클래스 구성도

각 스케줄링 알고리즘은 CPUSchedulingAlgorithm 이라는 추상 클래스를 상속하여 process()라는 추상 메서드를 구현하는 형태이다.



- Utility : List<Row>를 깊은 복사를 하기 위한 Uitility 성 함수이다.
- Row : Process | AT | BT | Priority | WT | TAT | RT 로 구성되어 있는 한 프로세스 행을 위한 클래스이다. 이를 List 형태로 사용한다.
- Event : processName, startTime, finishTime 을 포함한 실제 알고리즘의 수행 타임라인을 기록하기 위한 클래스이다. 역시 List 형태로 사용한다.
- CPUSchedulingAlgorithm : 알고리즘들의 추상 메서드이다. 모든 알고리즘은 이 클래스를 상속하여 추상 메서드 process()를 Override 하여 구현하는 형태이다.
- FCFS : First Come First Served 알고리즘 이다.
- SJF : Shortest Job First 알고리즘이다.
- HRN : Highest Response Ratio Next 알고리즘 이다.
- PSN : Priority Non Preemptive(우선순위 비선점) 알고리즘 이다.
- PSP : Priority Preemptive(우선순위 선점) 알고리즘 이다.
- RR : Round Robin 알고리즘 이다.
- SRT : Shortest Remaining Time 알고리즘 이다. 기본적으로 RR 스케줄링을 사용한다.
- GUI : Java Swing 을 이용한 GUI 클래스이다.
- Main : GUI 클래스를 불러와 실행하는 main() 함수와 콘솔 실행을 테스트하는 test() 함수가 있다.

2.2 각 알고리즘 핵심 코드

1. FCFS

FCFS 는 준비 큐에 도착한 순서대로 CPU 를 할당하는 비선점형 방식이다. 한번 실행 되면 그 프로세스가 끝나야만 다음 프로세스를 실행할 수 있으며, 큐가 하나이기 때문에 모든 프로세스의 우선 순위가 동일하다.

추상 메서드 process를 구현한 함수는 다음과 같다.

우선 도착한 순서대로 프로세스들을 정렬한다.

```
3개 사용 위치  Ogu1208 *
public class FCFS extends CPUSchedulingAlgorithm {
    Ogu1208 *
    @Override
    public void process() {

        // Sort list of objects using Collection.sort() with lambdas only
        // 도착시간(ArrivalTime)을 기준으로 빠른순으로 정렬한다. (도착한 순서대로)
        Collections.sort(this.getRows(), (Object o1, Object o2) -> {
            if (((Row) o1).getArrivalTime() == ((Row) o2).getArrivalTime()) {
                return 0;
            } else if (((Row) o1).getArrivalTime() < ((Row) o2).getArrivalTime()) {
                return -1;
            } else {
                return 1;
            }
        });   Ogu1208, 2023-05-22 오후 5:06 • [feat]: create FCFS class
```

이후 Event 의 List 인 timeline 에 for 문을 돌며 각 프로세스의 Event 값들을 Event List 에 추가한다. 만약 timeline 이 비어 있었다면, 첫 번째 프로세스를 add 한다. 이때 첫번째로 추가되는 프로세스의 startTime 은 그 프로세스의 arrivalTime 이다. 이후에 add 하는 프로세스들은 timeline 에 마지막에 들어있는 프로세스의 finishtime 이 startTime 이 된다.

```
20      List<Event> timeline = this.getTimeline();
21
22      // for문을 돌며 각 프로세스의 Event 값들을 Event List에 add
23      for (Row row : this.getRows()) {
24          if (timeline.isEmpty()) {   // Event 리스트가 비어있으면 첫번째 process Event 추가
25              timeline.add(new Event(row.getProcessName(), row.getArrivalTime(),  finishTime: row.getArrivalTime() + row.getBurstTime()));
26          } else {   // 비어있지 않으면 (2번째 프로세스부터) Event 추가
27              Event event = timeline.get(timeline.size() - 1);  // 마지막 프로세스를 가져옴
28              // startTime : 마지막에 진행된 프로세스의 끝난시간
29              // finishTime : 마지막에 진행된 프로세스의 끝난시간 + row의 burstTime
30              timeline.add(new Event(row.getProcessName(), event.getFinishTime(),  finishTime: event.getFinishTime() + row.getBurstTime()));
31          }
32      }
```

for 문을 돌며 waitingTime, turnAroundTime, responseTime 을 설정한다.

```
33
34          // for문을 돌며 waitingTime, turnAroundTime, responseTime set
35          for (Row row : this.getRows()) {
36              // waitingTime = 시작 시작 - 도착 시간
37              row.setWaitingTime(this.getEvent(row).getStartTime() - row.getArrivalTime());
38              // turnAroundTime = 대기 시간 + 실행 시간(burst time)
39              row.setTurnaroundTime(row.getWaitingTime() + row.getBurstTime());
40              // responseTime = 첫 작업을 시작한 후 첫 번째 출력(반응)이 나오기 전까지 시간
41              row.setResponseTime(row.getWaitingTime() + 1);
42          }
43      }   Ogu1208, 2023-05-22 오후 5:06 • [feat]: create FCFS class
44  }
```

2. SJF

SJF 는 준비큐에 있는 프로세스 중에서 실행 시간이 가장 짧은 작업부터 CPU 를 할당하는 비선점형 방식이다. 콘보이 효과를 완화하여 시스템의 효율성을 높인다.

추상 메서드 process를 구현한 함수는 다음과 같다.

우선 도착한 순서대로 프로세스들을 정렬한다.

```
3개 사용 위치  Ogu1208 *
public class SJF extends CPUSchedulingAlgorithm {
    Ogu1208 *
    @Override
    public void process() {
        // Sort list of objects using Collection.sort() with lambdas only
        // 도착시간(ArrivalTime)을 기준으로 빠른순으로 정렬한다. (도착한 순서대로)
        Collections.sort(this.getRows(), (Object o1, Object o2) -> {
            if (((Row) o1).getArrivalTime() == ((Row) o2).getArrivalTime()) {
                return 0;
            } else if (((Row) o1).getArrivalTime() < ((Row) o2).getArrivalTime()) {
                return -1;
            } else {
                return 1;
            }
        });
```

Utility 의 deepCopy 함수를 이용해 프로세스들을 깊은 복사 한다.
또한 time 을 첫번째 프로세스의 도착시간으로 설정한다.

```
        // rows = this.getRows를 깊은 복사
        List<Row> rows = Utility.deepCopy(this.getRows());
        int time = rows.get(0).getArrivalTime();
```

while 문을 돌며 rows 가 비어 있지 않은 동안 반복한다.
우선 매번 availableRows 를 만들어 실행 가능한 프로세스들의 List 를 만든다.

```
        while (!rows.isEmpty()) {
            // available Row의 List를 만든다.
            List<Row> availableRows = new ArrayList();

            // for문을 돌며 time보다 row의 도착시간이 작거나 같으면 available Row에 추가한다.
            for (Row row : rows) {
                if (row.getArrivalTime() <= time) {
                    availableRows.add(row);
                }
            }
```

availableRows 를 실행 시간이 가장 짧은 순서대로 정렬한다.

```
// 실행시간(burst time)이 짧은 순으로 정렬한다. (SJF)
Collections.sort(availableRows, (Object o1, Object o2) -> {
    if (((Row) o1).getBurstTime() == ((Row) o2).getBurstTime()) {
        return 0;
    } else if (((Row) o1).getBurstTime() < ((Row) o2).getBurstTime()) {
        return -1;
    } else {
        return 1;
    }
});
```

정렬한 avilableRows 에서 첫번째 프로세스(실행시간이 가장 짧은)를 가져온다.
그 후 timeLine(List<Event>)에 해당 프로세스의 Evnet 를 추가하고 startTime 과 finishTime 을 세팅한다.
그 후 time 에 방금 실행한 프로세스의 burstTime 만큼 더한다.

```
// 가능한 rows들 중 실행시간이 가장 짧은 row(process)를 구한다.
Row row = availableRows.get(0);

// timeline(List<Event>)에 실행시간이 가장 짧은 row(process)의 Event를 추가
// startTime : time
// finishTime : time + row의 burstTime(실행시간)
this.getTimeline().add(new Event(row.getProcessName(), time,  finishTime: time + row.getBurstTime()));

// time에 burstTime을 더함으로써 다음 실행되는 프로세스의 startTime이 되도록 함.
time += row.getBurstTime();
```

for 문을 돌며 깊은 복사한 rows 에서 실행된 row 를 찾아 삭제한다.(remove)

```
        // for문을 돌며 깊은복사한 rows에서 실행된 row를 찾아 삭제한다.(remove)
        for (int i = 0; i < rows.size(); i++) {
            if (rows.get(i).getProcessName().equals(row.getProcessName())) {
                rows.remove(i);
                break;
            }
        }
    }
```

마찬가지로 for 문을 돌며 waitingTime, turnAroundTime, responseTime 을 설정한다.

```
        // for문을 돌며 waitingTime, turnAroundTime, responseTime set
        for (Row row : this.getRows()) {
            // waitingTime = 시작 시작 - 도착 시간
            row.setWaitingTime(this.getEvent(row).getStartTime() - row.getArrivalTime());
            // turnAroundTime = 대기 시간 + 실행 시간(burst time)
            row.setTurnaroundTime(row.getWaitingTime() + row.getBurstTime());
            // responseTime = 첫 작업을 시작한 후 첫 번째 출력(반응)이 나오기 전까지 시간
            row.setResponseTime(row.getWaitingTime() + 1);

        }   Ogu1208, 2023-05-23 오전 2:55 • [feat]: create SJF Algorithm
    }
}
```

3. HRN

HRN 스케줄링은 SJF 스케줄링에서 발생할 수 있는 아사 현상을 해결하기 위해 만들어진 비선점형 알고리즘이다. 최고 응답률 우선 스케줄링이라고도 하며 서비스를 받기 위해 기다린 시간과 CPU 사용 시간을 고려하여 스케줄링을 하는 방식이다. 프로세스의 우선 순위를 결정하는 기준은 아래와 같다.

우선순위 = (대기시간 + CPU 사용 시간) / CPU 사용 시간

추상 메서드 process를 구현한 함수는 다음과 같다.

우선 도착한 순서대로 프로세스들을 정렬한다.

```
public class HRN extends CPUSchedulingAlgorithm {

    Ogu1208
    @Override
    public void process() {
        // Sort list of objects using Collection.sort() with lambdas only
        // 도착시간(ArrivalTime)을 기준으로 빠른순으로 정렬한다. (도착한 순서대로)
        Collections.sort(this.getRows(), (Object o1, Object o2) -> {
            if (((Row) o1).getArrivalTime() == ((Row) o2).getArrivalTime()) {
                return 0;
            } else if (((Row) o1).getArrivalTime() < ((Row) o2).getArrivalTime()) {
                return -1;
            } else {
                return 1;
            }
        });
```

Utility 의 deepCopy 함수를 이용해 프로세스들을 깊은 복사 한다.

또한 time 을 첫번째 프로세스의 도착시간으로 설정한다.

```
        // rows = this.getRows를 깊은 복사
        List<Row> rows = Utility.deepCopy(this.getRows());
        int time = rows.get(0).getArrivalTime();  // 도착 시간
```

while 문을 돌며 rows 가 비어 있지 않은 동안 반복한다.

우선 매번 availableRows 를 만들어 실행 가능한 프로세스들의 List 를 만든다.

```
        while (!rows.isEmpty()) {
            // available Row의 List를 만든다.
            List<Row> availableRows = new ArrayList();

            // for문을 돌며 time보다 row의 도착시간이 작거나 같으면 available Row에 추가한다.
            for (Row row : rows) {
                if (row.getArrivalTime() <= time) {
                    availableRows.add(row);
                }
            }
```

for 문을 돌며 각 프로세스의 현재 기준 WaitingTime(대기시간)과 그에 따른 우선 순위를 세팅한다. 이 때, 숫자가 클수록 우선순위가 높음에 주의한다. (대기시간에 비례하기 때문)

```
// for문을 돌며 우선순위를 계산한다
// priority = (대기 시간 + CPU 사용 시간) / CPU 사용시간
for (Row row : availableRows) {
    row.setWaitingTime(time - row.getArrivalTime());
    row.setPriority((row.getWaitingTime() + row.getBurstTime()) / row.getBurstTime());
}
```

우선순위가 높은 순(숫자가 큰 순 = 에이징 순)으로 정렬한다.

```
// 우선순위가 높은 순(숫자가 큰 순 = 에이징 순)으로 정렬한다. (Priority)
Collections.sort(availableRows, (Object o1, Object o2) -> {
    if (((Row) o1).getPriority() == ((Row) o2).getPriority()) {
        return 0;
    } else if (((Row) o1).getPriority() > ((Row) o2).getPriority()) {
        return -1;
    } else {
        return 1;
    }
});
```

정렬한 avilableRows 에서 첫번째 프로세스(우선순위가 가장 높은 = 우선순위 숫자가 가장 큰)를 가져온다.
그 후 timeLine(List<Event>)에 해당 프로세스의 Evnet 를 추가하고 startTime 과 finishTime 을 세팅한다.
그 후 time 에 방금 실행한 프로세스의 burstTime 만큼 더한다.

```
63          // 가능한 rows들 중 우선순위가 가장 높은 row(process)를 구한다.
64          Row row = availableRows.get(0);
65
66          // timeline(List<Event>)에 우선순위가 가장 높은 row(process)의 Event를 추가
67          // startTime : time
68          // finishTime : time + row의 burstTime(실행시간)
69          this.getTimeline().add(new Event(row.getProcessName(), time,  finishTime: time + row.getBurstTime()));
70          time += row.getBurstTime();
71
```

for 문을 돌며 깊은 복사한 rows 에서 실행된 row 를 찾아 삭제한다. (remove)

```
72          // for문을 돌며 깊은복사한 rows에서 실행된 row를 찾아 삭제한다.(remove)
73          for (int i = 0; i < rows.size(); i++) {
74              if (rows.get(i).getProcessName().equals(row.getProcessName())) {
75                  rows.remove(i);
76                  break;
77              }
78          }
79      }
```

마찬가지로 for 문을 돌며 waitingTime, turnAroundTime, responseTime 을 설정한다.

```
81          // for문을 돌며 waitingTime, turnAroundTime, responseTime set
82          for (Row row : this.getRows()) {
83              // waitingTime = 시작 시작 - 도착 시간
84              row.setWaitingTime(this.getEvent(row).getStartTime() - row.getArrivalTime());
85              // turnAroundTime = 대기 시간 + 실행 시간(burst time)
86              row.setTurnaroundTime(row.getWaitingTime() + row.getBurstTime());
87              // responseTime = 첫 작업을 시작한 후 첫 번째 출력(반응)이 나오기 전까지 시간
88              row.setResponseTime(row.getWaitingTime() + 1);
89
90          }   Ogu1208, 2023-05-24 오전 2:54 • [feat]: create HRN"
91      }
92  }
```

4. PriorityNonPreemptive (우선순위 비선점)

우선순위 스케줄링 중 비선점 방식입니다. 프로세스의 중요도에 따른 우선순위를 반영한 스케줄링 알고리즘으로, 숫자가 작을수록 우선순위가 높다. 비선점은 작업이 완료될 때까지 뺏을 수 없다.

추상 메서드 process를 구현한 함수는 다음과 같다.

우선 도착한 순서대로 프로세스들을 정렬한다.

```
public class PriorityNonPreemptive extends CPUSchedulingAlgorithm {

    Ogu1208
    @Override
    public void process() {
        // Sort list of objects using Collection.sort() with lambdas only
        // 도착시간(ArrivalTime)을 기준으로 빠른순으로 정렬한다. (도착한 순서대로)
        Collections.sort(this.getRows(), (Object o1, Object o2) -> {
            if (((Row) o1).getArrivalTime() == ((Row) o2).getArrivalTime()) {
                return 0;
            } else if (((Row) o1).getArrivalTime() < ((Row) o2).getArrivalTime()) {
                return -1;
            } else {
                return 1;
            }
        });
```

앞선 과정과 같이 rows 를 깊은 복사 하고 time 을 설정한다.

```
        // rows = this.getRows를 깊은 복사
        List<Row> rows = Utility.deepCopy(this.getRows());
        int time = rows.get(0).getArrivalTime();  // 도착 시간
```

while 문을 돌며 rows 가 비어 있지 않은 동안 반복한다.
우선 매번 availableRows 를 만들어 실행 가능한 프로세스들의 List 를 만든다.

```
        while (!rows.isEmpty()) {
            // available Row의 List를 만든다.
            List<Row> availableRows = new ArrayList();

            // for문을 돌며 time보다 row의 도착시간이 작거나 같으면 available Row에 추가한다.
            for (Row row : rows) {
                if (row.getArrivalTime() <= time) {
                    availableRows.add(row);
                }
            }
```

우선순위가 높은 순(숫자가 작은 순)으로 정렬한다.

```
        // 우선순위가 높은 순(숫자가 작은 순)으로 정렬한다. (Priority)
        Collections.sort(availableRows, (Object o1, Object o2) -> {
            if (((Row) o1).getPriority() == ((Row) o2).getPriority()) {
                return 0;
            } else if (((Row) o1).getPriority() < ((Row) o2).getPriority()) {
                return -1;
            } else {
                return 1;
            }
        });
```

정렬한 avilableRows 에서 첫번째 프로세스(우선순위가 가장 높은)를 가져온다.
그 후 timeLine(List<Event>)에 해당 프로세스의 Evnet 를 추가하고 startTime 과 finishTime 을 세팅한다.
그 후 time 에 방금 실행한 프로세스의 burstTime 만큼 더한다.

```
        // 가능한 rows들 중 우선순위가 가장 높은 row(process)를 구한다.
        Row row = availableRows.get(0);

        // timeline(List<Event>)에 우선순위가 가장 높은 row(process)의 Event를 추가
        // startTime : time
        // finishTime : time + row의 burstTime(실행시간)
        this.getTimeline().add(new Event(row.getProcessName(), time,  finishTime: time + row.getBurstTime()));
        time += row.getBurstTime();
```

for 문을 돌며 깊은 복사 한 rows 에서 실행된 row 를 찾아 삭제한다. (remove)

```
        // for문을 돌며 깊은복사한 rows에서 실행된 row를 찾아 삭제한다.(remove)
        for (int i = 0; i < rows.size(); i++) {
            if (rows.get(i).getProcessName().equals(row.getProcessName())) {
                rows.remove(i);
                break;
            }
        }
    }
```

마찬가지로 for 문을 돌며 waitingTime, turnAroundTime, responseTime 을 설정한다.

```
        // for문을 돌며 waitingTime, turnAroundTime, responseTime set
        for (Row row : this.getRows()) {
            // waitingTime = 시작 시작 - 도착 시간
            row.setWaitingTime(this.getEvent(row).getStartTime() - row.getArrivalTime());
            // turnAroundTime = 대기 시간 + 실행 시간(burst time)
            row.setTurnaroundTime(row.getWaitingTime() + row.getBurstTime());
            // responseTime = 첫 작업을 시작한 후 첫 번째 출력(반응)이 나오기 전까지 시간
            row.setResponseTime(row.getWaitingTime() + 1);   0gu1208, 2023-05-24 오전 1:32 • [feat

        }
    }
}
```

5. Priority Preemptive (우선순위 선점)

프로세스의 중요도에 따른 우선순위를 반영한 스케줄링 알고리즘으로, 숫자가 작을수록 우선순위가 높으며 변동 우선순위 알고리즘이다. 선점형은 작업 중간에 뺏을 수 있음을 의미한다.
추상 메서드 process를 구현한 함수는 다음과 같다.

우선 도착한 순서대로 프로세스들을 정렬한다.

```
public class PriorityPreemtive extends CPUSchedulingAlgorithm {
    Ogu1208
    @Override
    public void process() {
        // Sort list of objects using Collection.sort() with lambdas only
        // 도착시간(ArrivalTime)을 기준으로 빠른순으로 정렬한다. (도착한 순서대로)
        Collections.sort(this.getRows(), (Object o1, Object o2) -> {
            if (((Row) o1).getArrivalTime() == ((Row) o2).getArrivalTime()) {
                return 0;
            } else if (((Row) o1).getArrivalTime() < ((Row) o2).getArrivalTime()) {
                return -1;
            } else {
                return 1;
            }
        });
```

앞선 과정과 같이 rows 를 깊은 복사 하고 time 을 설정한다.

```
        // rows = this.getRows를 깊은 복사
        List<Row> rows = Utility.deepCopy(this.getRows());
        int time = rows.get(0).getArrivalTime();  // 도착 시간

```

while 문을 돌며 rows 가 비어 있지 않은 동안 반복한다.
우선 매번 availableRows 를 만들어 실행 가능한 프로세스들의 List 를 만든다.

```
        while (!rows.isEmpty()) {
            List<Row> availableRows = new ArrayList();
            // available Row의 List를 만든다.

            // for문을 돌며 time보다 row의 도착시간이 작거나 같으면 available Row에 추가한다.
            for (Row row : rows) {
                if (row.getArrivalTime() <= time) {
                    availableRows.add(row);
                }
            }
```

우선순위가 높은 순(숫자가 작은 순)으로 정렬한다.

```
// 우선순위가 높은 순(숫자가 작은 순)으로 정렬한다. (Priority)
Collections.sort(availableRows, (Object o1, Object o2) -> {
    if (((Row) o1).getPriority() == ((Row) o2).getPriority()) {
        return 0;
    } else if (((Row) o1).getPriority() < ((Row) o2).getPriority()) {
        return -1;
    } else {
        return 1;   Ogu1208, Today • [feat]: create Priority Preemtive
    }
});
```

정렬한 avilableRows 에서 첫번째 프로세스(우선순위가 가장 높은)를 가져온다.
그 후 timeLine(List<Event>)에 해당 프로세스의 Evnet 를 추가하고 startTime 과 finishTime 을 세팅한다. 이때 선점형 방식은 중간에 뺏을 수 있으므로 1초씩 실행시킨다. 따라서 finishTime 은 ++time 이 도고, burstTime 도 1초씩 줄어든다.

```
// 가능한 rows들 중 우선순위가 가장 높은 row(process)를 구한다.
Row row = availableRows.get(0);

// timeline(List<Event>)에 우선순위가 가장 높은 row(process)의 Event를 추가
// startTime : time
// finishTime : ++time (1초씩 증가)
this.getTimeline().add(new Event(row.getProcessName(), time, ++time));
row.setBurstTime(row.getBurstTime() - 1);
```

만약 해당 프로세스의 burstTIme 이 끝났다면(0이라면) 같은 이름의 프로세스를 찾아 rows 행에서 remove 시킨다.

```
// 만약 해당 process의 burstTime이 끝났다면, 같은 이름의 프로세스를 찾아 rows행에서 remove시킨다.
if (row.getBurstTime() == 0) {
    for (int i = 0; i < rows.size(); i++) {
        if (rows.get(i).getProcessName().equals(row.getProcessName())) {
            rows.remove(i);
            break;
        }
    }
}
}
```

선점형 방식이어서 1초씩 실행시켰으므로 Event List 에도 역시 1초씩 add 되었을 것이다. 따라서 한 프로세스가 몇초동안 실행된다면 같은 프로세스가 1초씩 계속 add 될 것이므로 for 문으로 Event 의 List 를 뒤에서부터 거꾸로 찾으면서 timeline 의 마지막 프로세스와 그 바로 앞의 프로세스가 같다면 마지막-1의 프로세스의 finishTime 을 마지막 프로세스의 finishTime 으로 업데이트하고 마지막 프로세스를 삭제한다.

```
        // for문으로 Event의 List를 뒤에서부터 거꾸로 찾으면서
        for (int i = this.getTimeline().size() - 1; i > 0; i--) {
            List<Event> timeline = this.getTimeline();

            // timeline의 마지막 프로세스와 마지막-1의 프로세스가 같다면
            if (timeline.get(i - 1).getProcessName().equals(timeline.get(i).getProcessName())) {
                // 마지막-1의 프로세스의 finishTime을 마지막 프로세스의 finishTime으로 업데이트하고
                timeline.get(i - 1).setFinishTime(timeline.get(i).getFinishTime());
                // 마지막 프로세스를 삭제한다.
                timeline.remove(i);
            }   Ogu1208, Today • [feat]: create Priority Preemtive
        }
```

HashMap 을 생성하고 for 문을 돌며 row 와 evnet 를 비교하여 waitingTime, turnAroundTime, responseTime 을 설정한다.

```
        Map map = new HashMap();

        // for문을 돌며 row와 evnet 비교
        for (Row row : this.getRows()) {
            map.clear();

            // waitingTime = 시작 시작 - 도착 시간
            for (Event event : this.getTimeline()) {
                // row와 evnet의 프로세스 이름이 같으면
                if (event.getProcessName().equals(row.getProcessName())) {
                    // map에 프로세스 이름의 key가 있다면 (이미 작업중인 경우가 있었다면)
                    if (map.containsKey(event.getProcessName())) {
                        // 기존  waitingTime에 (이전에 작업했을 때의 finishTime - 이번 작업의 startTime)을 더함.
                        int w = event.getStartTime() - (int) map.get(event.getProcessName());
                        row.setWaitingTime(row.getWaitingTime() + w);
                    }
                    // 아직 map에 프로세스 이름의 key가 없다면
                    else {  // row의 waitingTime을 startTime - ArrivalTime으로 초기화
                        row.setWaitingTime(event.getStartTime() - row.getArrivalTime());
                    }
                    // map에 key : processName, value: finishTime을 삽입
                    map.put(event.getProcessName(), event.getFinishTime());
                }
            }

            // turnAroundTime = 대기 시간 + 실행 시간(burst time)
            row.setTurnaroundTime(row.getWaitingTime() + row.getBurstTime());
            // responseTime = 첫 작업을 시작한 후 첫 번째 출력(반응)이 나오기 전까지 시간
            row.setResponseTime(row.getWaitingTime() + 1);
        }
    }
}
```

6. Round Robin

라운드 로빈은 한 프로세스가 할당받은 시간(타임 슬라이스/타임 퀀텀)동안 작업을 하다가 작업을 완료하지 못하면 준비 큐의 맨 뒤로 가서 자기 차례를 기다리는 방식이다. 선점형 알고리즘 중 가장 단순하고 대표적인 방식으로 프로세스들이 작업을 완료할 때까지 계속 순환하면서 실행한다.

추상 메서드 process를 구현한 함수는 다음과 같다.

우선 도착한 순서대로 프로세스들을 정렬한다.

```
public class RoundRobin extends CPUSchedulingAlgorithm {

    Ogu1208
    @Override
    public void process() {
        // Sort list of objects using Collection.sort() with lambdas only
        // 도착시간(ArrivalTime)을 기준으로 빠른순으로 정렬한다. (도착한 순서대로)
        Collections.sort(this.getRows(), (Object o1, Object o2) -> {
            if (((Row) o1).getArrivalTime() == ((Row) o2).getArrivalTime()) {
                return 0;
            } else if (((Row) o1).getArrivalTime() < ((Row) o2).getArrivalTime()) {
                return -1;
            } else {
                return 1;
            }
        });
```

앞선 과정과 같이 rows 를 깊은 복사 하고 time 을 설정한다.
또한 timeQuantum(타임 슬라이스)를 설정한다.

```
        // rows = this.getRows를 깊은 복사
        List<Row> rows = Utility.deepCopy(this.getRows());
        int time = rows.get(0).getArrivalTime();

        // time Quantum 설정
        int timeQuantum = this.getTimeQuantum();
```

while 문을 돌며 rows 가 비어 있지 않은 동안 반복한다.
첫번째 프로세스(가장 먼저 도착한)를 꺼내 한 타임 슬라이스동안 작업할 burstTime 을 계산한다.
timeQuantum 보다 burstTime 이 작으면 burstTime 만큼, 아니라면 timeQuantum 만큼 실행한다.

```
        while (!rows.isEmpty()) {
            Row row = rows.get(0);
            // 한 텀의  burst time 계산
            // (timeQuantum보다 burstTime이 작으면 burstTime만큼, 아니라면 timeQuantum만큼 실행)
            int bt = (row.getBurstTime() < timeQuantum ? row.getBurstTime() : timeQuantum);
```

그 후 timeLine(List<Event>)에 해당 프로세스의 Evnet 를 추가하고 startTime 과 finishTime 을 세팅한다.
그 후 time 에 방금 실행한 프로세스의 burstTime 만큼 더한다. 그 후 rows 에서 해당 프로세스(가장 앞자리)를 remove 시킨다.

```
35        // timeline(List<Event>)에 가장 먼저 도착한 row(process)의 Event를 추가
36        // startTime : time
37        // finishTime : time + row의 bt
38        this.getTimeline().add(new Event(row.getProcessName(), time,  finishTime: time + bt));
39        time += bt;  // time 갱신 (timme+=bt)
40        rows.remove( index: 0);
```

만약 프로세스의 burstTime 이 timeQuantum 보다 커서 burstTime 이 남았다면 burstTime 을 방금 bt 만큼 뺀 값으로 다시 갱신하고, 현재 기준 큐의 맨 뒤에 추가한다.

```
42        // 만약 프로세스의 burstTime이 timeQuantum보다 크다면
43        if (row.getBurstTime() > timeQuantum) {
44            // burtTime을 burstTime - timeQuantum으로 갱신
45            row.setBurstTime(row.getBurstTime() - timeQuantum);
46
47            // timequantum만큼 일한 프로세스 row를 큐의 뒤에 추가
48            for (int i = 0; i < rows.size(); i++) {
49                // 만약 rows의 행이 방금 작업이 끝난 시간 time보다 늦게 도착한다면
50                if (rows.get(i).getArrivalTime() > time) {
51                    // 해당 자리에 방금 끝난 프로세스 row를 추가
52                    rows.add(i, row);
53                    break;
54                }
55                // 아니라면 큐의 마지막에 row 추가
56                else if (i == rows.size() - 1) {
57                    rows.add(row);
58                    break;
59                }
60            }
61        }   Ogu1208, Yesterday • [feat]: create RoundRobin
62    }
```

HashMap 을 생성하고 for 문을 돌며 row 와 evnet 를 비교하여 waitingTime, turnAroundTime, responseTime 을 설정한다.

```
        Map map = new HashMap();

        // for문을 돌며 row와 evnet 비교   Ogu1208, Yesterday • [feat]: create RoundRobin
        for (Row row : this.getRows()) {
            map.clear();

            // waitingTime = 시작 시작 - 도착 시간
            for (Event event : this.getTimeline()) {
                // row와 evnet의 프로세스 이름이 같으면
                if (event.getProcessName().equals(row.getProcessName())) {
                    // map에 프로세스 이름의 key가 있다면 (이미 한번 이상 burstTime만큼 작업했다면)
                    if (map.containsKey(event.getProcessName())) {
                        // 기존  waitingTime에 (이전에 작업했을 때의 finishTime - 이번 작업의 startTime)을 더함.
                        int w = event.getStartTime() - (int) map.get(event.getProcessName());
                        row.setWaitingTime(row.getWaitingTime() + w);
                    }
                    // 아직 map에 프로세스 이름의 key가 없다면
                    else {   // row의 waitingTime을 startTime - ArrivalTime으로 초기화
                        row.setWaitingTime(event.getStartTime() - row.getArrivalTime());
                    }

                    // map에 key : processName, value: finishTime을 삽입
                    map.put(event.getProcessName(), event.getFinishTime());
                }
            }

            // turnAroundTime = 대기 시간 + 실행 시간(burst time)
            row.setTurnaroundTime(row.getWaitingTime() + row.getBurstTime());
            // responseTime = 첫 작업을 시작한 후 첫 번째 출력(반응)이 나오기 전까지 시간
            row.setResponseTime(row.getWaitingTime() + 1);
        }
    }
}
```

7. SRT (Shortest Remaining Time)

SRT 는 기본적으로 라운드 로빈 스케줄링을 사용하지만, CPU 를 할당 받을 프로세스를 선택할 때 남아 있는 작업 시간이 가장 적은 프로세스를 선택한다.

추상 메서드 process를 구현한 함수는 다음과 같다.

우선 도착한 순서대로 프로세스들을 정렬한다.

```
public class SRT extends CPUSchedulingAlgorithm {
    Ogu1208
    @Override
    public void process() {
        // Sort list of objects using Collection.sort() with lambdas only
        // 도착시간(ArrivalTime)을 기준으로 빠른순으로 정렬한다. (도착한 순서대로)
        Collections.sort(this.getRows(), (Object o1, Object o2) -> {
            if (((Row) o1).getArrivalTime() == ((Row) o2).getArrivalTime()) {
                return 0;
            } else if (((Row) o1).getArrivalTime() < ((Row) o2).getArrivalTime()) {
                return -1;
            } else {
                return 1;
            }
        });
```

앞선 과정과 같이 rows 를 깊은 복사 하고 time 을 설정한다.

또한 timeQuantum(타임 슬라이스)를 설정한다.

```
        // rows = this.getRows를 깊은 복사
        List<Row> rows = Utility.deepCopy(this.getRows());
        int time = rows.get(0).getArrivalTime();  // 첫 번째 프로세스의 도착시간

        // time Quantum 설정
        int timeQuantum = this.getTimeQuantum();
```

while 문을 돌며 rows 가 비어 있지 않은 동안 반복한다.

우선 매번 availableRows 를 만들어 실행 가능한 프로세스들의 List 를 만든다.

```
        while (!rows.isEmpty()) {
            // available Row의 List를 만든다.
            List<Row> availableRows = new ArrayList();

            // for문을 돌며 time보다 row의 도착시간이 작거나 같으면 available Row에 추가한다.
            for (Row row : rows) {
                if (row.getArrivalTime() <= time) {
                    availableRows.add(row);
                }
            }
```

burstTime 이 작은(남아있는 작업 시간이 가장 적은)순으로 프로세스들을 정렬한다.

```
36          // burstTime이 작은(남아있는 작업 시간이 가장 적은 순)으로 정렬한다. (Shortest Remaining Time)
37          Collections.sort(availableRows, (Object o1, Object o2) -> {
38              if (((Row) o1).getBurstTime() == ((Row) o2).getBurstTime()) {
39                  return 0;   Ogu1208, Today • [feat]: create SRT
40              } else if (((Row) o1).getBurstTime() < ((Row) o2).getBurstTime()) {
41                  return -1;
42              } else {
43                  return 1;
44              }
45          });
```

while 문을 돌며 rows 가 비어 있지 않은 동안 반복한다.
첫번째 프로세스(현재 기준 가장 남아있는 작업 시간이 적은)를 꺼내 한 타임 슬라이스동안 작업할 burstTime 을 계산한다.
timeQuantum 보다 burstTime 이 작으면 burstTime 만큼, 아니라면 timeQuantum 만큼 실행한다.

```
47          // 가능한 rows들 중 burstTime이 가장 적은 row(process)를 구한다.
48          Row row = availableRows.get(0);
49
50          // 한 턤의  burst time 계산
51          // (timeQuantum보다 burstTime이 작으면 burstTime만큼, 아니라면 timeQuantum만큼 실행)
52          int bt = (row.getBurstTime() < timeQuantum ? row.getBurstTime() : timeQuantum);
53
```

그 후 timeLine(List<Event>)에 해당 프로세스의 Evnet 를 추가하고 startTime 과 finishTime 을 세팅한다.
그 후 time 에 방금 실행한 프로세스의 burstTime 만큼 더한다.

```
54          // timeline(List<Event>)에 burstTime이 가장 작은(남아있는 작업 시간이 가장 적은) row(process)의 Event를 추가
55          // startTime : time
56          // finishTime : time + row의 bt
57          this.getTimeline().add(new Event(row.getProcessName(), time,  finishTime: time + bt));
58          time += bt;  // time 갱신 (timme+=bt)
```

Row 의 burstTIme 을 burstTime – bt 로 갱신하고, 해당 process 의 burstTime 이 끝났다면 같은 이름의 프로세스를 찾아 rows 행에서 remove 시킨다.

```
60          // row의 burstTime = burstTime - bt 로 업데이트
61          row.setBurstTime(row.getBurstTime() - bt);
62
63          // 만약 해당 process의 burstTime이 끝났다면, 같은 이름의 프로세스를 찾아 rows행에서 remove시킨다.
64          if (row.getBurstTime() == 0) {
65              for (int i = 0; i < rows.size(); i++) {
66                  if (rows.get(i).getProcessName().equals(row.getProcessName())) {
67                      rows.remove(i);
68                      break;
69                  }
70              }
71          }
72      }
```

for 문으로 Event 의 List 를 뒤에서부터 거꾸로 찾으면서 timeline 의 마지막 프로세스와 그 바로 앞의 프로세스가 같다면 마지막-1의 프로세스의 finishTime 을 마지막 프로세스의 finishTime 으로 업데이트하고 마지막 프로세스를 삭제한다.

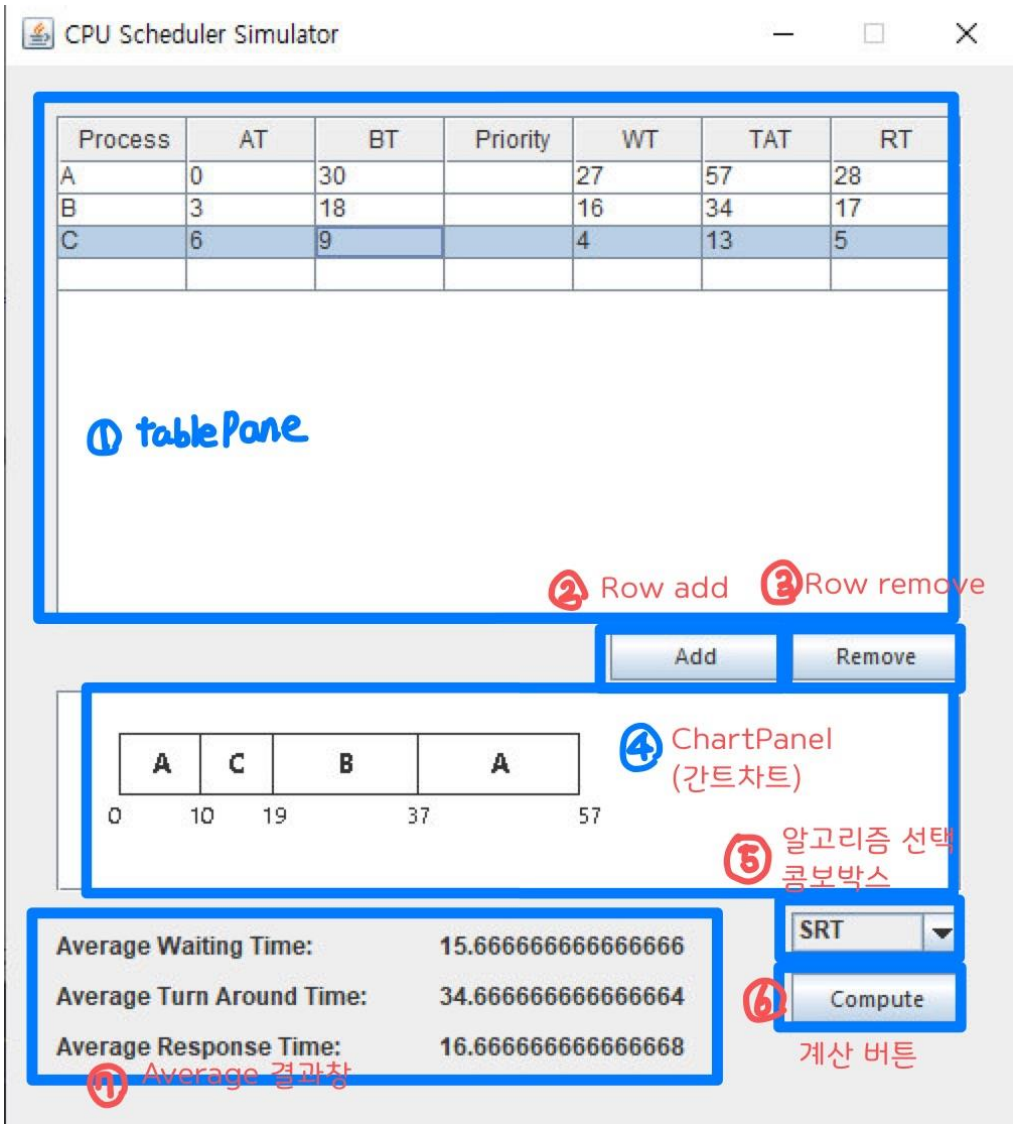
```
74          // for문으로 Event의 List를 뒤에서부터 거꾸로 찾으면서
75          for (int i = this.getTimeline().size() - 1; i > 0; i--) {
76              List<Event> timeline = this.getTimeline();
77
78              // timeline의 마지막 프로세스와 마지막-1의 프로세스가 같다면
79              if (timeline.get(i - 1).getProcessName().equals(timeline.get(i).getProcessName())) {
80                  // 마지막-1의 프로세스의 finishTime을 마지막 프로세스의 finishTime으로 업데이트하고   Ogu1208, To
81                  timeline.get(i - 1).setFinishTime(timeline.get(i).getFinishTime());
82                  // 마지막 프로세스를 삭제한다.
83                  timeline.remove(i);
84              }
85          }
```

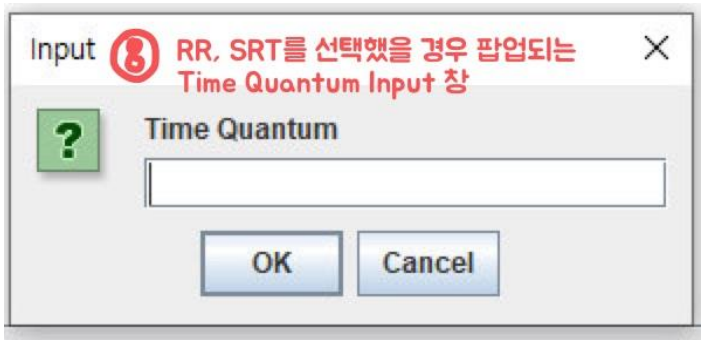
HashMap 을 생성하고 for 문을 돌며 row 와 evnet 를 비교하여 waitingTime, turnAroundTime, responseTime 을 설정한다.

```
87          Map map = new HashMap();
88
89          // for문을 돌며 row와 evnet 비교
90          for (Row row : this.getRows()) {
91              map.clear();
92
93              // waitingTime = 시작 시작 - 도착 시간
94              for (Event event : this.getTimeline()) {
95                  // row와 evnet의 프로세스 이름이 같으면   Ogu1208, Today • [feat]: create SRT
96                  if (event.getProcessName().equals(row.getProcessName())) {
97                      // map에 프로세스 이름의 key가 있다면 (이미 한번 이상 burstTime만큼 작업했다면)
98                      if (map.containsKey(event.getProcessName())) {
99                          // 기존  waitingTime에 (이전에 작업했을 때의 finishTime - 이번 작업의 startTime)을 더함.
100                         int w = event.getStartTime() - (int) map.get(event.getProcessName());
101                         row.setWaitingTime(row.getWaitingTime() + w);
102                     }
103                     // 아직 map에 프로세스 이름의 key가 없다면
104                     else {   // row의 waitingTime을 startTime - ArrivalTime으로 초기화
105                         row.setWaitingTime(event.getStartTime() - row.getArrivalTime());
106                     }
107
108                     // map에 key : processName, value: finishTime을 삽입
109                     map.put(event.getProcessName(), event.getFinishTime());
110                 }
111             }
112
113             // turnAroundTime = 대기 시간 + 실행 시간(burst time)
114             row.setTurnaroundTime(row.getWaitingTime() + row.getBurstTime());
115             // responseTime = 첫 작업을 시작한 후 첫 번째 출력(반응)이 나오기 전까지 시간
116             row.setResponseTime(row.getWaitingTime() + 1);
117         }
118     }
119 }
```

2.2 개발한 UI 특징

AT : ArrivaTime(도착시간), BT : BurstTime (서비스 시간), Priority : 우선 순위, WT : 대기 시간, TAT : 반환 시간, RT : 응답 시간

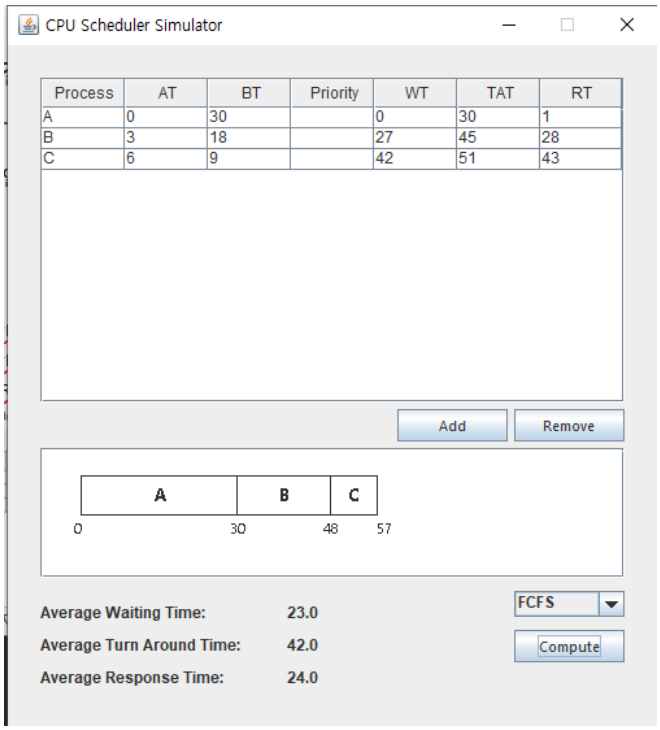




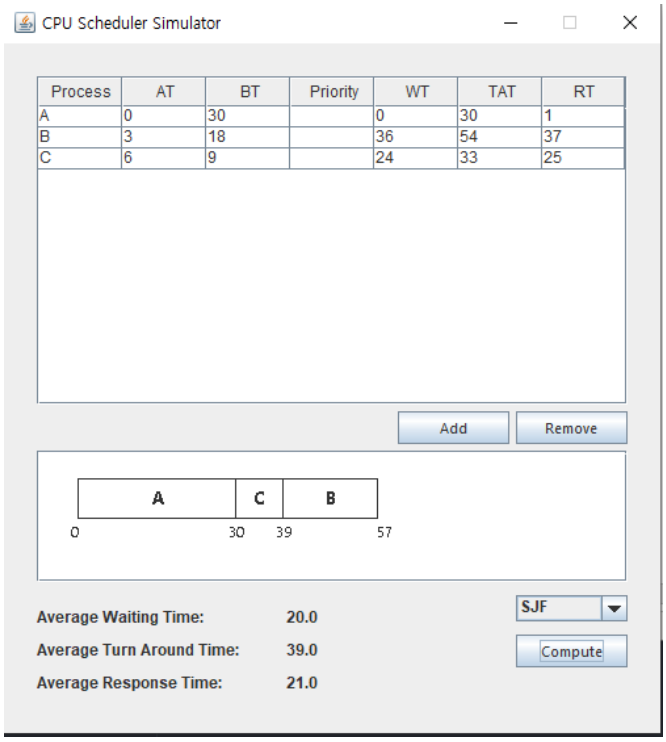
3. 결과

3.1 각 알고리즘 별 결과 창

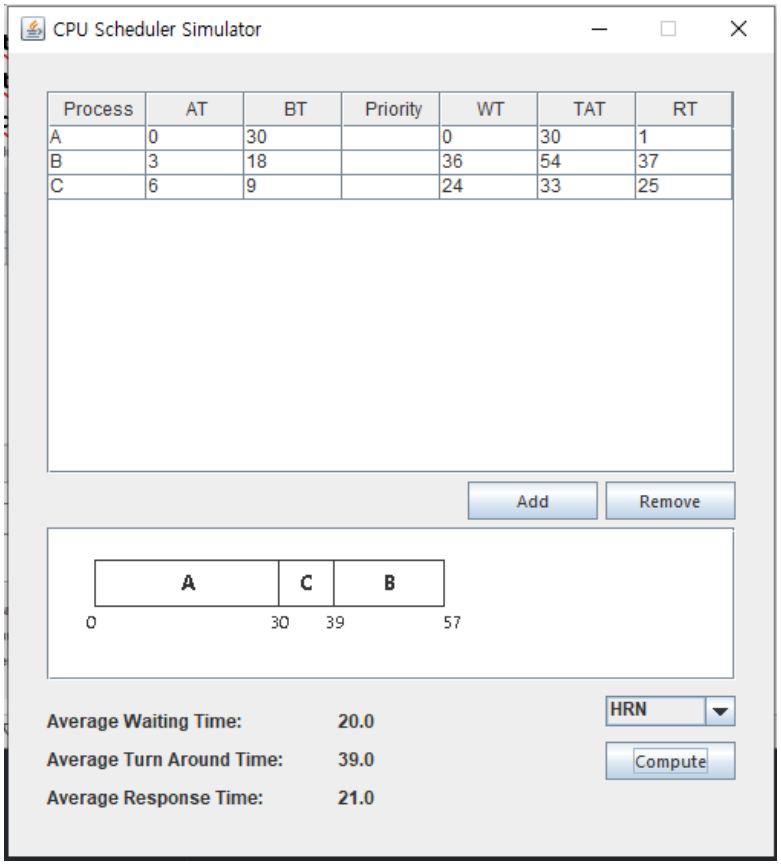
1. FCFS



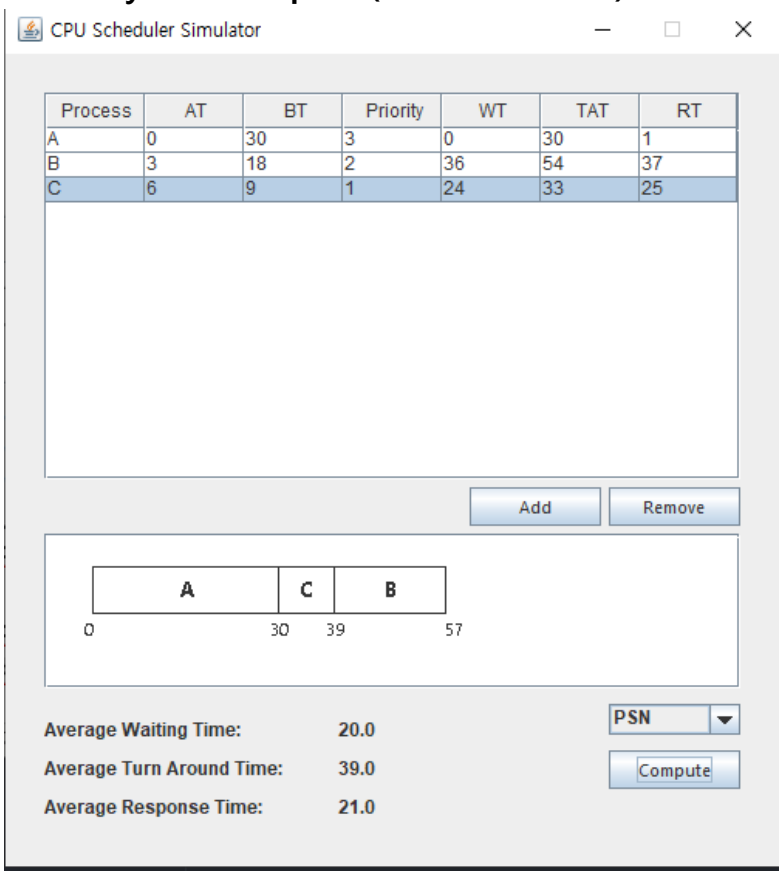
2. SJF



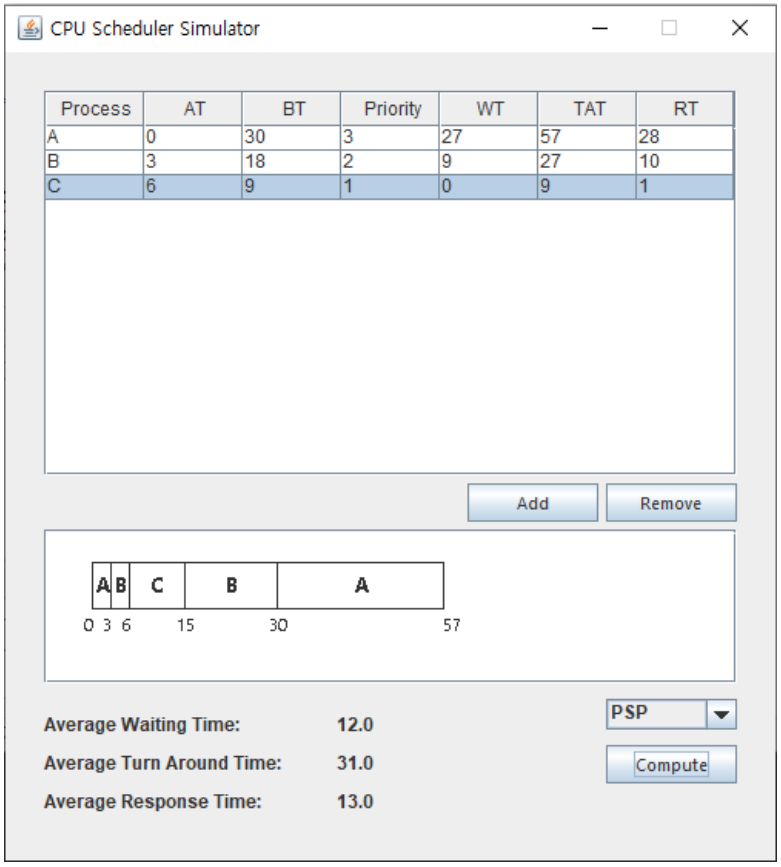
3. HRN



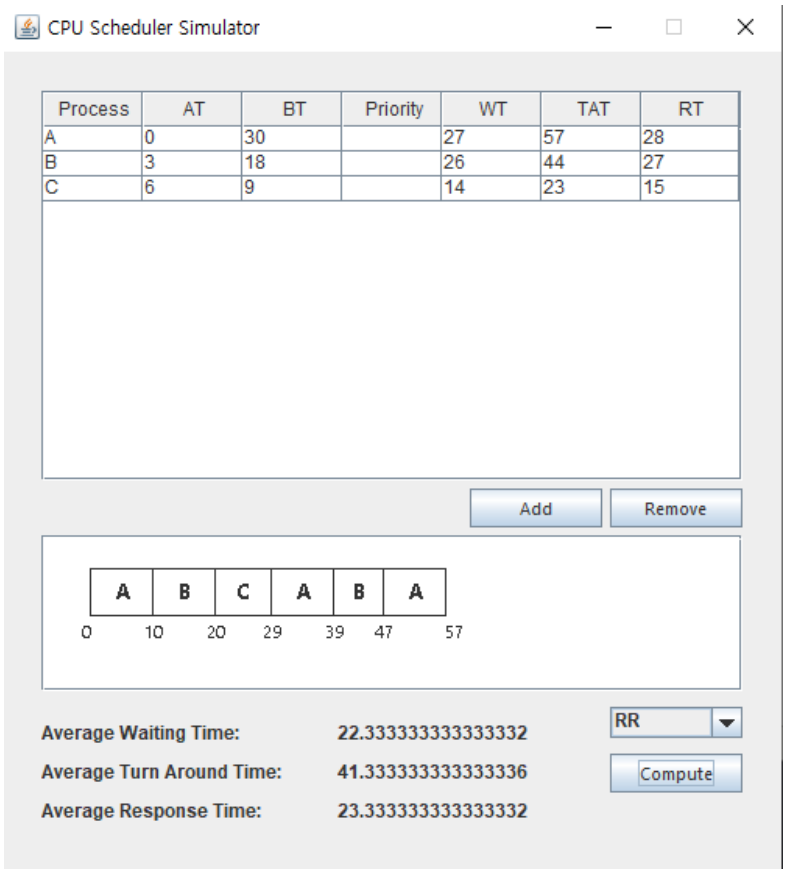
4. PriorityNonPreemptive (우선 순위 비선점)



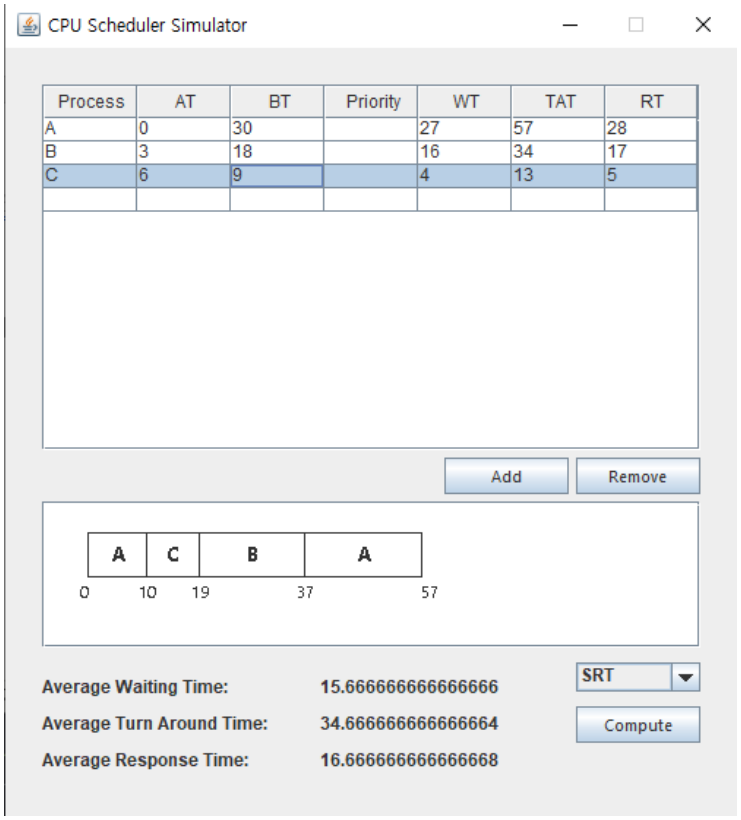
5. PriorityPreemtive (우선 순위 선점)



6. RoundRobin



7. SRT



3.2 느낀점

스케줄링 시뮬레이터를 구현하면서 다양한 스케줄링 알고리즘에 대해 깊이 이해할 수 있었습니다. 각 알고리즘의 작동 방식, 장단점, 성능 특성 등을 파악하고 구현에 반영하는 과정은 알고리즘 이론을 공부하는 것과는 차원이 다른 난이도였습니다. 사람이 생각하는 것과 이를 실제로 적용해보는 것은 정말 많은 시간을 생각에 잠기게 했습니다.

각 프로세스를 도착 시간 순으로 정렬하는 과정, 각 시간에서 실행 가능한 프로세스를 선별하여 해당 알고리즘에 맞게 정렬하는 과정, 원래의 프로세스 List 를 건드리지 않기 위한 deepCopy, 각 프로세스가 어떻게 진행되는지 기록하는 과정 등등을 고려해야 했습니다. 특히 선점형 방식의 경우 실행 도중 빼앗을 수 있기 때문에, timeQuantum 이 있는 알고리즘들은 해당 Quantum 동안 어떻게 작업을 마쳤는지, 프로세스의 작업 상태를 기록해야 했고, Quantum 을 사용하지 않는 선점형 알고리즘의 경우 1초마다 계산하였습니다.

또한 Java Swing 을 이용해 GUI 환경을 구성하여 시뮬레이션을 실행하고 나온 결과를 분석하는 과정을 시각화하였습니다. 각 스케줄링 알고리즘의 성능을 비교하고, 프로세스들의 대기 시간, 반환 시간, 응답 시간 등과 같은 메트릭을 평가할 수 있도록 하였습니다. GUI 로 꼭 구현하고 싶어, 외국 유튜브를 이것 저것 강의를 참고하여 Java Swing 을 공부한 과정을 GitHub 에 기록하였습니다.

그리고 본 과제를 수행하며 Git 과 GitHub 를 적극적으로 활용하며 feature branch 를 따로 만들어 각 프로세스의 브랜치를 만들어 개발하고, 이를 main 브런치에 pull request 를 보내는 방식으로

개발하였습니다. 이 과정에서 git revert 가 되어 발생하는 오류들이 발생하여 해결하는 과정을 거쳤습니다. 길다면 길고 짧다면 짧은 텀 프로젝트 기간동안, 각 알고리즘과 GUI 를 어떻게 구성할지 깊은 고민을 할 수 있었고 시뮬레이터를 개발하는 과정에서 여러 시나리오를 시뮬레이션하고 결과를 분석하면서 디버깅 스킬도 향상시킬 수 있었습니다. 예상치 못한 문제점을 찾아내고 수정하는 과정은 프로그래밍 능력과 문제 해결 능력을 향상시키고, 자신감을 찾는데 도움이 되었습니다.

OS 는 딱딱하고 리눅스와 같은 운영체제를 공부하는 과목이라고 생각했는데, 전반적인 컴퓨터 구조와 프로세스들을 어떻게 관리하고 실행시키는지, 메모리 관리와 입출력 및 파일 시스템은 어떻게 동작되는지에 대해 한학기동안 정말 전념하여 관심을 가지고 열심히 공부했습니다.

앞으로 이러한 경험이 앞으로의 발전에 있어 도움이 될 것이라고 확신합니다.