운영체제

CPU스케줄링 시뮬레이터 레포트

제출일	22.06.12
전공	컴퓨터소프트웨어공학과
과목	운영체제
학번	20184612
담당교수	김대영 교수님
이름	김동민

목차

1. 개발 언어

2. 함수/클래스 구성도/다이어그램

3. 자료구조/주요 변수설명

4. 각 알고리즘 핵심코드 설명

4-1. FCFS스케줄링

4-2. SJF스케줄링

4-3. 고정 우선순위 스케줄링

4-4. 변동 우선순위 스케줄링

4-5. RoundRobin스케줄링

4-6. SRT스케줄링

4-7. HRN스케줄링

5.개발한 UI특징

6.각 알고리즘 실행 결과

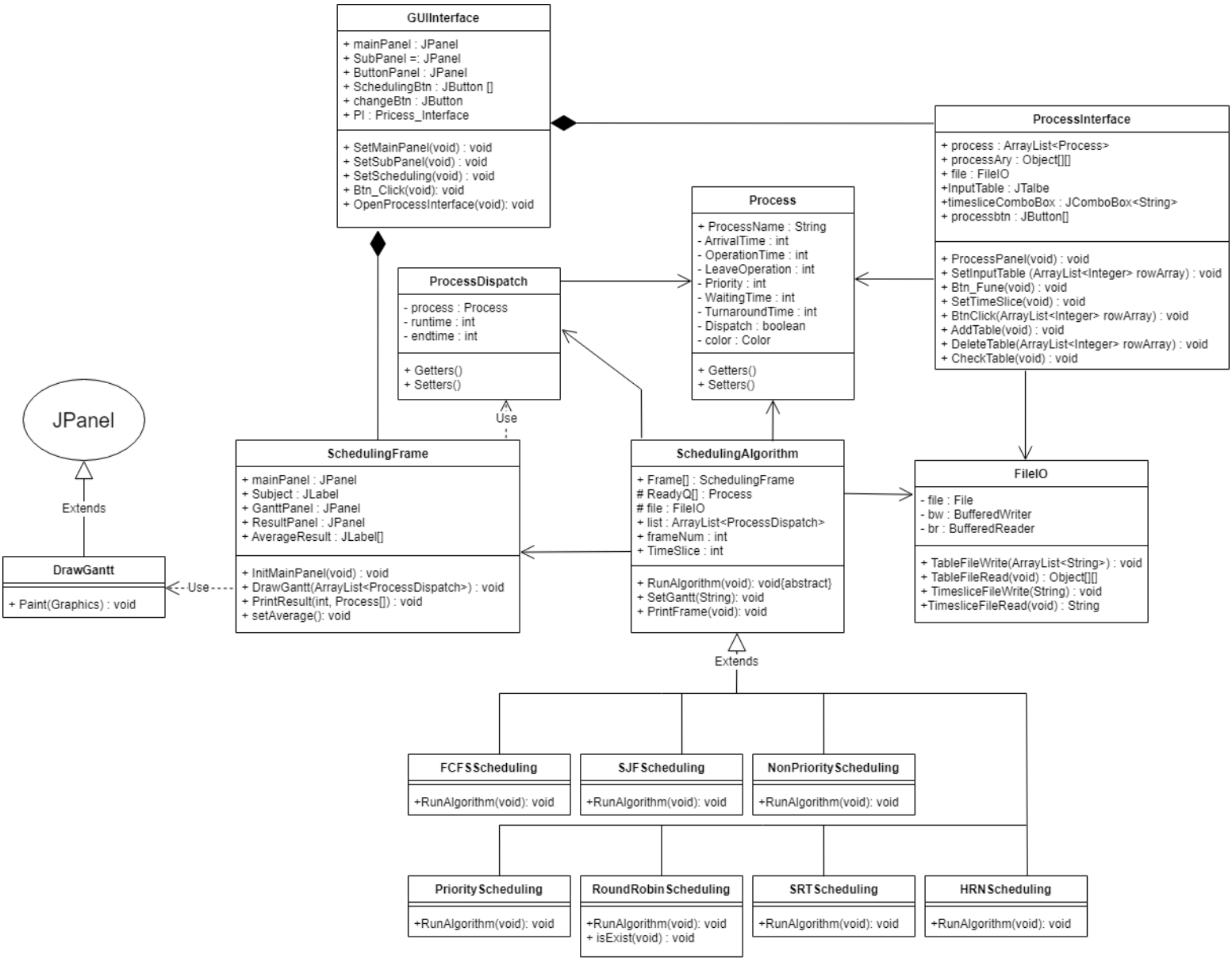
7. 느낀 점

1. 개발언어

개발언어: JAVA

JAVA를 사용한 이유: 가장 먼저 콘솔 창보다는 GUI가 더 사용자에게 직관적으로 보이기 때문에 GUI를 이용하여 개발하기로 했고, 작년 JAVA프로그래밍 시간에 GUI를 사용하여 프로그래밍을 해본 경험이 있기 때문에 JAVA를 이용하여 개발을 했다.

2. 함수/클래스 구성도/다이어그램



사용한 프로그램: diagrams.net

GUIInterface클래스: 프로세스 수정 또는 스케줄링 프레임을 실행시킨다.

ProcessInterface클래스: 프로세스를 추가, 삭제하고 파일에 작성한다.

SchedulingFrame클래스: 스케줄링 결과를 화면에 표시한다.

Process클래스: 프로세스에 대한 정보를 저장한다.

ProcessDispatch클래스: 프로세스 실행 순서에 대한 정보를 담고있다.

FileIO클래스: 프로세스, 타임슬라이스를 파일에 쓰고 읽어오는 클래스이다.

SchedulingAlgorithm클래스: 모든 스케줄링 클래스의 부모클래스이다. 알고리즘 구현 함수가 추상클래스로 작성되어있다.

FCFSScheduling, SJFScheduling, NonPriorityScheduling, PriorityScheduling, RoundRobinScheduling, SRTScheduling, HRNScheduling 클래스: 각 알고리즘을 구현한 클래스이다.

3. 자료구조/주요변수 설명

```
String ProcessName;
int ArrivalTime;    //도착시간
int OperationTime;  //작업시간
int LeaveOperation; //남은시간

int Priority;       //우선순위
int WaitingTime;    //대기시간
int ResponseTime;   //응답시간

int TurnaroundTime; //반환시간
boolean Dispatch;

Color color;
```

Process 클래스

Process클래스에는 이름을 저장할 ProcessName과 도착시간(ArrivalTime), 작업시간(OperationTime), 남은시간(LeaveOperation), 우선순위(Priority), 대기시간(WaitingTime), 응답시간(ResponseTime), 반환시간(TurnaroundTime)을 저장할 변수가 있다.

Dispatch변수는 이 프로세스가 디스패치 되었는지 확인하는 boolean타입 변수이고 color변수는 간트차트를 그릴 때 프로세스의 색상을 저장하는 변수이다.

```
public class ProcessDispatch {
//  String processName;
    Process process;
    int runtime;
    int endtime;
```

ProcessDispatch 클래스

ProcessDispatch클래스는 프로세스를 간트차트에 그릴 때 사용하는 클래스이다. 스케줄링 도중 디스패치나 타임아웃이 발생하면 ProcessDispatch형으로 선언된 리스트에 이를 저장하고, 이 리스트를 바탕으로 간트차트에 표를 그린다.

```
FCFSScheduling(){
    frameNum = 0;
    ShowFrame();

    System.out.println("FCFS");
    SetGantt("FCFS");
    RunAlgorithm();
}
public void RunAlgorithm() {
    System.out.println("Running");

    int n = ReadyQ.length;
    int Pcnt = 0, time = 0;      //Pc
```

FCFS스케줄링 클래스의 예시

FCFS의 스케줄링 프레임 번호는 0번이고 SetGantt()함수로 프레임을 초기화한다. 그리고 부모클래스의 RunAlgorithm() 추상클래스를 구현한다.

Pcnt는 실행한 프로세스의 개수를 나타내고 time은 시간을 나타내는 변수로 프로세스가 동작하면 실행시간이나 타임슬라이스만큼 증가한다.

```
public abstract class SchedulingAlgorithm {
//  protected Process [] process;
    protected FileIO file = new FileIO();

    static SchedulingFrame []frame= new SchedulingFrame[7];

    protected Process []ReadyQ;
//  protected ArrayList<Process>ProcessPID = new ArrayList<>();
    int frameNum = -1;

    int TimeSlice;

    int operationSum = 0;
    int AvgWaiting = 0;
    int AvgResponse = 0;
    int AvgTurnAround = 0;

    ArrayList<ProcessDispatch> list = new ArrayList<>();
    SchedulingAlgorithm(){
        SetProcess();


    }

    public void SetProcess() {
        Object [][] temp;


        temp = file.TableFileRead();
        TimeSlice = Integer.parseInt(file.TimesliceFileRead());
    //  process = new Process[temp.length];
        ReadyQ = new Process[temp.length];

        for(int i=0; i<temp.length; i++) {
            ReadyQ[i] = new Process(temp[i][0], (temp[i][1]),  (temp[i][2]), (temp[i][3]));
            operationSum+=Integer.parseInt((String)temp[i][2]);
            ReadyQ[i].setLeaveOperation(ReadyQ[i].getOperationTime());
        }
        RandomColor();
        System.out.println(operationSum);

    }
    abstract public void RunAlgorithm();
```

SchedulingAlgorithm클래스의 변수와 생성자

7개의 알고리즘 클래스의 부모 클래스SchedulingAlgorithm이다. 스케줄링 버튼을 누르면 다른 스케줄링 창을 모두 닫아야 하기 때문에 Frame을 static으로 7개 생성한다.

frameNum은 프레임 번호를 나타내는 변수로 0은 FCFS, 1은 SJF, 3은 고정 우선순위, 4는 변동 우선순위, 5는 라운드로빈, 6은 SRT, 7은 HRN프레임을 나타낸다. 각 자식 클래스의 생성자에서 프레임 번호를 지정한다.

ReadyQ는 프로세스를 삽입하는 배열로 선언하고 파일에 저장되어 있는 행의 개수로 배열의 크기를 지정한다. TimeSlice는 타임슬라이스를 저장하는 변수이고 AvgWaiting, AvgResponse, AvgTurnAround는 각각 평균 대기시간, 평균 응답시간. 평균 반환시간을 저장하는 변수이다.

ArrayList자료형의 list변수는 ProcessDispatch클래스를 저장하고 프로세스의 동작이 일어나는 순서를 저장하는 변수이다. list에 저장된 element로 간트차트를 그릴 때 활용할 수 있다.

SetProcess함수에서는 파일 데이터를 읽어오는 file클래스를 활용하여 ReadyQ와 TimeSlice를 읽어오고 RandomColor()함수로 프로세스의 색깔을 지정한다.

추상클래스 abstract void RunAlgorithm()클래스이다. SchedulingAlgorithm을 상속받는 자식 클래스는 RunAlgorithm() 추상클래스에 알고리즘을 작성하여 구현을 한다.

4. 각 알고리즘 핵심부분 코드 설명

4-1. FCFS 스케줄링 알고리즘

```
while(true) {
    int c = n, min = 999;
    if(Pcnt == n)
        break;
    for(int i=0; i<n; i++) {
        if((ReadyQ[i].getArrivalTime()<=time)&& (!ReadyQ[i].getDispatch())&&(ReadyQ[i].getArrivalTime()<min) ) {
            min = ReadyQ[i].getArrivalTime();

            c = i;
        }
    }

//  if(tot==0) st = ReadyQ[c].getArrivalTime();
    if(c==n) {
    //  System.out.println(c+", "+ min);
        System.out.println("none");
        Process tmp = new Process("none");
    //  tmp.setColor(Color.gray);
    //  System.out.println(tmp.getColor());
        if(list.size()>0) {
            if(list.get(list.size()-1).getProcessName().equals("none")) {
                list.get(list.size()-1).runtime+=1;
            }
            else list.add(new ProcessDispatch(tmp, 1));
        }
        else list.add(new ProcessDispatch(tmp, 1));
        time++;
    }
    else {
        ReadyQ[c].setTurnaroundTime(time+ReadyQ[c].getOperationTime() - ReadyQ[c].getArrivalTime());
        ReadyQ[c].setResponseTime(ReadyQ[c].getTurnaroundTime()-ReadyQ[c].OperationTime);
        ReadyQ[c].setWaitingTime(ReadyQ[c].getTurnaroundTime()-ReadyQ[c].OperationTime);
        ReadyQ[c].setDispatch(true);
    //  ReadyQ[tot[]
        System.out.println(ReadyQ[c].ProcessName);
    //  System.out.println(ReadyQ[c].getTurnaroundTime());
        time+=ReadyQ[c].getOperationTime();
        list.add(new ProcessDispatch(ReadyQ[c], ReadyQ[c].getLeaveOperation()));

        Pcnt++;

    }

}
GanttColor();

for(int i = 0; i<n; i++) {
    AvgWaiting += ReadyQ[i].getWaitingTime();
    AvgResponse +=ReadyQ[i].getResponseTime();
    AvgTurnAround+=ReadyQ[i].getTurnaroundTime();
}
PrintFrame();
```

1. n은 프로세스의 개수를 나타내고 min은 도착시간이 가장 작은 값을 저장한다.

2. Pcnt는 종료된 프로세스의 개수를 나타내고 time은 현재 시간을 나타낸다.

3. FCFS스케줄링은 프로세스가 먼저 도착한 순서대로 디스패치된다. 때문에 가장 먼저 프로세스 개수만큼 반복문을 돌며 디스패치 되지 않은 것 중에 가장 도착 순서가 빠른 것을 min을 통해 찾아낸다. c에는 프로세스 인덱스

를 저장한다.

4. C를 처음에 n으로 초기화했기 때문에 만약 c와 n이 같다면 도착한 프로세스가 없다는 의미이다. 때문에 프로세스가 없다는 none프로세스를 리스트에 삽입한다.
5. Else문은 프로세스가 도착했다는 의미이므로 CPU를 할당한다. 응답시간, 대기시간, 반환시간을 set함수를 통해 저장하고 setDispatch함수로 할당되었음을 표시한다.
6. 간트차트에 표를 그리기위해 list에 프로세스가 진행되는 순서를 저장한다.
7. GanttColor()함수를 통해 프로세스의 색을 지정한다.
8. AvgWaiting, AvgResponse, AvgTurnAround변수에 각 프로세스의 대기, 응답, 반환시간을 더하고 PrintFrame에서 평균을 구해 화면에 출력한다.

4-2. SJF 스케줄링 알고리즘

```
while(true) {
    int c = n, min = 999;
    if(Pcnt == ReadyQ.length)
        break;
    for(int i=0; i<ReadyQ.length; i++) {
        if((ReadyQ[i].getArrivalTime()<=time)&& !ReadyQ[i].getDispatch()&&ReadyQ[i].getOperationTime()<min ) {
            min = ReadyQ[i].getOperationTime();
            System.out.println(ReadyQ[i].ProcessName);
            c = i;
        }
    }
    //if(tot==0) st = ReadyQ[c].getArrivalTime();
    if(c==n) {
        System.out.println(c+", "+ min);
        Process tmp = new Process("none");
        //  tmp.setColor(Color.gray);
        //  System.out.println(tmp.getColor());
            if(list.size()>0) {
                if(list.get(list.size()-1).getProcessName().equals("none")) {
                    list.get(list.size()-1).runtime+=1;
                }
                else list.add(new ProcessDispatch(tmp, 1));
            }
        else list.add(new ProcessDispatch(tmp, 1));
            time++;
    }
    else {
        ReadyQ[c].setWaitingTime(time+ReadyQ[c].getOperationTime());
        time+=ReadyQ[c].getOperationTime();
    //  ReadyQ[tot].
        ReadyQ[c].setTurnaroundTime(ReadyQ[c].getWaitingTime()-ReadyQ[c].getArrivalTime());
        ReadyQ[c].setResponseTime(ReadyQ[c].getTurnaroundTime()-ReadyQ[c].getOperationTime());
        ReadyQ[c].setWaitingTime(ReadyQ[c].getTurnaroundTime()-ReadyQ[c].getOperationTime());
        ReadyQ[c].setDispatch(true);
        Pcnt++;

        list.add(new ProcessDispatch(ReadyQ[c], ReadyQ[c].getLeaveOperation()));
    }
}

GanttColor();

for(int i = 0; i<n; i++) {
    AvgWaiting += ReadyQ[i].getWaitingTime();
    AvgResponse +=ReadyQ[i].getResponseTime();
    AvgTurnAround+=ReadyQ[i].getTurnaroundTime();
}
PrintFrame();
```

1. SJF는 실행시간에 따라 CPU를 할당받는 순서가 달라진다. 따라서 현재 도착한 프로세스에서 getOperationTime으로 실행시간이 가장 짧은 프로세스를 찾아 c에 인덱스를 저장한다.
2. C를 n으로 초기화했기 때문에 만약 c가 n과 같다면 프로세스가 없다는 의미이기 때문에 none프로세스를 list에 삽입하고, time을 증가시킨다.
3. else이면 프로세스를 CPU에 할당했다는 의미이므로 대기, 반환, 응답시간을 구하여 저장하고 setDispatch로 프로세스가 디스패치 되었음을 알린다.
4. Time에는 실행시간을 추가로 더하고 list에 프로세스 진행상황을 저장한다.

4-3. 고정 우선순위 스케줄링 알고리즘

```
while(true) {
    int c = n, min = 999;
    if(Pcnt == ReadyQ.length)
        break;
    for(int i=0; i<ReadyQ.length; i++) {
        if((ReadyQ[i].getArrivalTime()<=time)&& !ReadyQ[i].getDispatch()&&ReadyQ[i].getPriority()<min   ) {
            min = ReadyQ[i].getPriority();

            c = i;
        }
    }
//  if(tot==0) st = ReadyQ[c].getArrivalTime();
     if (c==n) {
         Process tmp = new Process("none");
            //  tmp.setColor(Color.gray);
            //  System.out.println(tmp.getColor());
            if(list.size()>0) {
                if(list.get(list.size()-1).getProcessName().equals("none")) {
                    list.get(list.size()-1).runtime+=1;
                }
                else list.add(new ProcessDispatch(tmp, 1));
            }
            else list.add(new ProcessDispatch(tmp, 1));
            time++;
     }
        else
        {
        ReadyQ[c].setResponseTime(time+ReadyQ[c].getOperationTime());
    //  ReadyQ[c].setWaitingTime(st+ReadyQ[c].getOperationTime());
        time+=ReadyQ[c].getOperationTime();
    //  ReadyQ[tot].
        ReadyQ[c].setTurnaroundTime(ReadyQ[c].getResponseTime()-ReadyQ[c].getArrivalTime());
        ReadyQ[c].setResponseTime(ReadyQ[c].getTurnaroundTime()-ReadyQ[c].getOperationTime());
        ReadyQ[c].setWaitingTime(ReadyQ[c].getTurnaroundTime()-ReadyQ[c].getOperationTime());
        ReadyQ[c].setDispatch(true);
        System.out.println(ReadyQ[c].ProcessName);
        Pcnt++;
        list.add(new ProcessDispatch(ReadyQ[c], ReadyQ[c].getLeaveOperation()));

       }
}
GanttColor();
for(int i = 0; i<n; i++) {
    AvgWaiting += ReadyQ[i].getWaitingTime();
    AvgResponse += ReadyQ[i].getResponseTime();
    AvgTurnAround+=ReadyQ[i].getTurnaroundTime();
}
PrintFrame();
```

1. 고정 우선순위 알고리즘은 우선순위를 부여받으면 종료될 때까지 우선순위가 고정된다. 따라서 비교대상은 우선순위가 되어야 하므로 현재 도착한 프로세스 중에서 getPriority()함수를 통해 가장 높은 우선순위를 찾는다.
2. 우선순위가 작은 것을 우선순위가 높은 것으로 설정했기 때문에 min에는 가장 작은 우선순위를 저장한다.
3. C가 n과 같다면 프로세스가 없다는 의미이기 때문에 list에 none프로세스를 삽입하고 time을 증가시켜 다음에 들어올 프로세스를 찾는다.
4. else문에서는 프로세스가 할당되었으므로 응답시간, 대기시간, 반환시간을

구하고 setDispatch로 프로세스가 할당되었음을 저장한다. 한번 디스패치 되면 프로세스의 실행시간 전까지는 할당이 끝나지 않기 때문에 time에는 프로세스의 실행시간을 더한다.

5. 간트차트를 그리기 위해 list에 프로세스 진행상황을 저장한다.
6. GanttColor()함수를 통해 프로세스의 색을 지정한다.
7. AvgWaiting, AvgResponse, AvgTurnAround변수에 각 프로세스의 대기, 응답, 반환시간을 더하고 PrintFrame에서 평균을 구해 화면에 출력한다.
8. PrintFrame에서는 스케줄링 창에 레디큐의 도착시간, 실행시간, 우선순위와 대기, 응답, 반환시간이 출력될 수 있도록 하고 평균 대기시간, 평균 응답시간, 평균 반환시간 또한 출력한다.

4-4. 변동 우선순위 스케줄링 알고리즘

```
for(time = 0; Pcnt!=n;) {
     smallest=n-1; min=9999;
     for(int i=0;i<n;i++) {
        //System.out.println("min "+min);
        if( ReadyQ[i].getArrivalTime() <= time &&   ReadyQ[i].getLeaveOperation()>0&& ReadyQ[i].getPriority() < min ) {
           smallest=i;
        //   System.out.println(smallest);
           min= (int)ReadyQ[smallest].getPriority();
        }

     }
     if( min==9999) {
         System.out.println(1234);
         list.add(new ProcessDispatch(new Process("none"),1));
         time++;
         continue;
     }
     else {
//     if(st==0) st=ReadyQ[smallest].getArrivalTime();
         if(!ReadyQ[smallest].getDispatch()) {
             ReadyQ[smallest].setResponseTime(time-ReadyQ[smallest].getArrivalTime());
         //   System.out.println(ReadyQ[smallest].ProcessName);
             ReadyQ[smallest].setDispatch(true);
         }
         ReadyQ[smallest].setLeaveOperation(ReadyQ[smallest].getLeaveOperation()-1);
         if(ReadyQ[smallest].getLeaveOperation()<=0) {
             Pcnt++;
             end = time+1;

         //   list.add(new ProcessDispatch(1, ReadyQ[smallest]));
             System.out.println(1+ReadyQ[smallest].ProcessName +","+   ReadyQ[smallest].getLeaveOperation());
             ReadyQ[smallest].setWaitingTime(end-ReadyQ[smallest].getArrivalTime()-ReadyQ[smallest].getOperationTime());
             ReadyQ[smallest].setTurnaroundTime(end - ReadyQ[smallest].getArrivalTime());
         //   ReadyQ[smallest].setLeaveOperation(ReadyQ[smallest].getLeaveOperation()-ReadyQ[smallest].getOperationTime());
             list.add(new ProcessDispatch(ReadyQ[smallest],1));
         }
         else {
             System.out.println(ReadyQ[smallest].ProcessName +","+  ReadyQ[smallest].getLeaveOperation());
             list.add(new ProcessDispatch(ReadyQ[smallest],1 ));
         }
         time++;
     }
}
for(int i=1; i<list.size();) {
    if(list.get(i).getProcessName().equals(list.get(i-1).getProcessName())) {
        list.get(i-1).setRuntime(list.get(i-1).getRuntime()+1);
        list.remove(i);
    }
    else i++;
}
```

1. 변동 우선순위 알고리즘은 일정시간마다 우선순위가 변한다. 따라서 위에서의 알고리즘과 같이 time이 프로세스 실행 시간만큼 늘어나지 않고 1씩 늘어나며 더 높은 우선순위가 있는지 확인하며 알고리즘이 진행된다.

2. 우선순위를 구하는 방법은 고정 우선순위와 같다. 현재 time에 도착한 프로세스 중에서 우선순위가 가장 높은 프로세스를 찾고 smallest변수에 프로세스의 인덱스를 저장한다.

3. 만약 min에 값이 없다면 프로세스가 없다는 의미이므로 list에 none프로세스를 삽입한다.

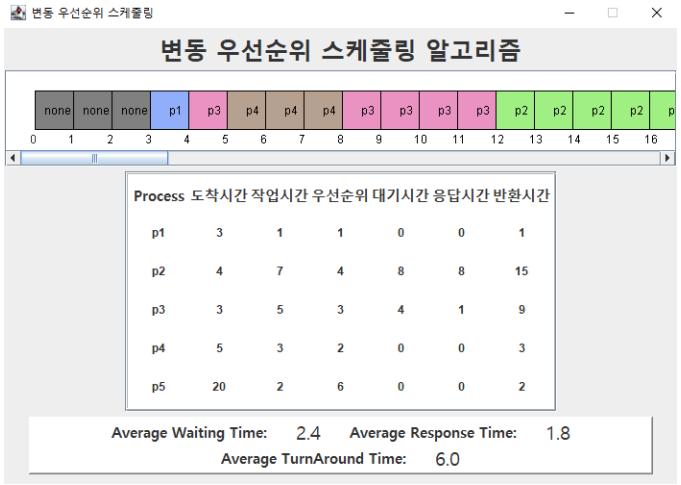
4. 만약 프로세스가 있다면 먼저 프로세스가 첫 실행인지, 한번 이상 실행이 된 적이 있는지 getDispatch함수를 사용하여 확인한다. 만약 한 번도 실행

되지 않은 프로세스라면 응답시간을 구하여 저장하고, 디스패치 되었다는 표시를 setDispatch로 남긴다.

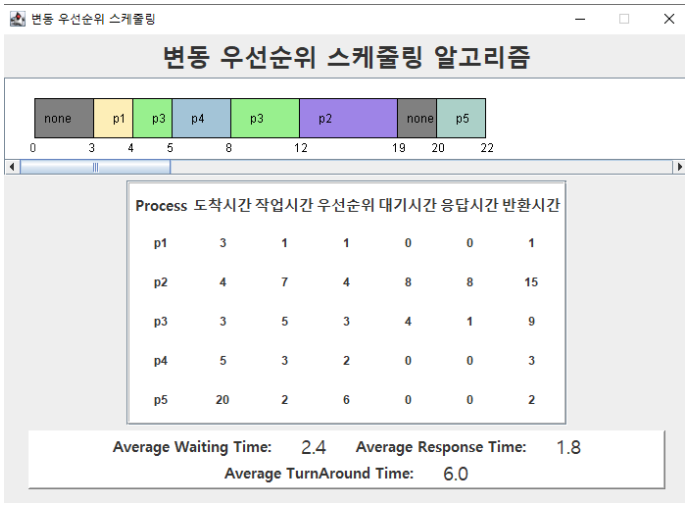
5. setLeaveOperation함수로 프로세스의 남은시간에서 1을 뺀다. 그리고 남은시간이 0보다 작거나 같다면 프로세스가 끝났다는 의미이므로 대기시간, 반환시간을 구하여 저장하고 list에 실행시간 1씩 저장한다.

6. 만약 남은시간이 1이상이라면 아직 프로세스에 남은 시간이 있으므로 list에 실행시간 1로 저장한다.

7. 마지막 반복문은 자신의 앞에 있는 프로세스가 자신과 같다면 간트차트를 하나로 합치는 코드이다. 실행시간을 1씩 list에 삽입했기 때문에 만약 이 코드가 없으면 아래 사진과 같이 간트차트에 1마다 칸이 생기게 될 것이다.



코드를 추가-> 앞 프로세스가 같다면 묶기 때문에 한눈에 알아보기가 쉽다.



4-5. Round-Robin 스케줄링 알고리즘

```
else {
    if(!isExist(ReadyQ[c].ProcessName)) queue.add(ReadyQ[c]);
    if(queue.size()>0) temp = queue.get(0);
    for(int i=0; i<n; i++) {
        if(temp.ProcessName.equals(ReadyQ[i].ProcessName) && !calc_response[i]) {
            if(cur_time>ReadyQ[i].getArrivalTime()) {
                ReadyQ[i].setResponseTime(cur_time - ReadyQ[i].getArrivalTime());
            }
               calc_response[i] = true;
        }
    }

    if(temp.getLeaveOperation()>TimeSlice) {
        cur_time+=TimeSlice;
        temp.setLeaveOperation(temp.getLeaveOperation()-TimeSlice);
        for(int i=0; i<=cur_time; i++) {
            for(int j=0; j<n; j++) {
                if(ReadyQ[j].getArrivalTime()==i  && !isExist(ReadyQ[j].ProcessName) && ReadyQ[j].getLeaveOperation()>0 ) {
                    queue.add(ReadyQ[j]);
                }
            }
        }
        queue.removeFirst();
        System.out.println(temp.ProcessName);
        list.add(new ProcessDispatch(temp,TimeSlice));
    }
    else {
        System.out.println(temp.ProcessName);
        cur_time += temp.getLeaveOperation();
        temp.setWaitingTime(cur_time-temp.getOperationTime());
        for(int i=0; i<n; i++) {
            if(temp.ProcessName.equals(ReadyQ[i].ProcessName)) {
                ReadyQ[i] = temp;
            }
            for(int j=0; j<=cur_time; j++) {
            //  for(int j=0; j<n; j++) {
                if(ReadyQ[i].getArrivalTime()==j  && !isExist(ReadyQ[i].ProcessName) && ReadyQ[i].getLeaveOperation()>0) {
                    queue.add(ReadyQ[i]);
                }

            }
        }
        queue.removeFirst();
    //  System.out.println(temp.ProcessName + ", "+ temp.getLeaveOperation());
        list.add(new ProcessDispatch(temp,temp.getLeaveOperation()));
        temp.setLeaveOperation(0);
        temp.setDispatch(true);
        Pcnt++;
        System.out.println(Pcnt);
    }
}
```

1. 라운드로빈의 경우도 앞의 알고리즘과 같이 현재 cur_time과 도착시간을 비교하고 가장 먼저 도착한 프로세스를 CPU할당한다.

2. 라운드 로빈에서는 큐를 사용한다. 큐는 링크리스트를 이용하여 구현했고 Process자료형을 사용한다. 만약 도착한 프로세스가 있고 그 프로세스가 큐에 존재하지 않는다면 큐에 삽입한다.

3. 큐에 가장 앞에 있는 프로세스를 temp에 삽입하고 처음 할당 되었는지 calc_respone로 확인한다. 만약 첫 할당이면 응답시간을 구하고 저장한다.

4. Temp의 남은 시간이 타임슬라이스보다 크다면 현재 시간을 타임슬라이만큼 증가시키고 프로세스의 남은시간에서 타임슬라이스를 뺀다. 그리고 현재 시간만큼 for문을 반복하며 그 사이에 도착한 프로세스들을 큐에 삽입

한다. 그리고 removefirst함수로 큐의 가장 앞을 삭제한다. 간트차트 list에는 TimeSlice만큼의 runtime으로 삽입을 한다.

5. 만약 temp의 남은시간이 타임슬라이보다 작다면 현재 시간에 남은 시간만큼 증가시키고 대기시간과 반환시간을 구한다. 그리고 레디큐를 변화한 temp값으로 업데이트한다. 또한 위 처럼 for문을 반복하며 프로세스들을 큐에 삽입한다.
6. 프로세스의 남은 시간은 0이므로 0으로 설정하고 간트차트를 그리기 위해 list에 실행시간만큼의 runtime으로 삽입한다.

4-6. SRT 스케줄링 알고리즘

```
for(time=0; Pcnt!=n;) {
        min = 999;
        s = 0;
        for(int i=0; i<n; i++) {
            if(ReadyQ[i].getArrivalTime()<=time&&ReadyQ[i].getLeaveOperation()<=min&& ReadyQ[i].getLeaveOperation()>0) {
                s = i;
            //  System.out.println(ReadyQ[smallest].ProcessName+","+ReadyQ[smallest].getLeaveOperation()+"min: "+min);
                min = ReadyQ[s].getLeaveOperation();

            }
        }

        if(min==999) {
             Process tmp = new Process("none");
            if(list.size()>0) {
                if(list.get(list.size()-1).getProcessName().equals("none")) {
                    list.get(list.size()-1).runtime+=1;
                }
                else list.add(new ProcessDispatch(tmp, 1));
            }
            else list.add(new ProcessDispatch(tmp, 1));
            time++;
        }
        else {
            if(!ReadyQ[s].getDispatch()) {
                 ReadyQ[s].setResponseTime(time-ReadyQ[s].getArrivalTime());
                    //   System.out.println(ReadyQ[smallest].ProcessName);
                 ReadyQ[s].setDispatch(true);
            }
            if(ReadyQ[s].getLeaveOperation()>TimeSlice) {
                ReadyQ[s].setLeaveOperation(ReadyQ[s].getLeaveOperation()-TimeSlice);
                list.add(new ProcessDispatch(ReadyQ[s],TimeSlice));
                time+=TimeSlice-1;
            }
            else {
                list.add(new ProcessDispatch(ReadyQ[s],ReadyQ[s].getLeaveOperation()));
                time+=ReadyQ[s].getLeaveOperation()-1;
                ReadyQ[s].setLeaveOperation(0);
                ReadyQ[s].setDispatch(true);

            }

            if(ReadyQ[s].getLeaveOperation()<=0) {
                Pcnt++;
                end = time+1;

                ReadyQ[s].setTurnaroundTime(end-ReadyQ[s].getArrivalTime());
                ReadyQ[s].setWaitingTime(ReadyQ[s].getTurnaroundTime()-ReadyQ[s].getOperationTime());

            }
            time++;
        }
```

1. SRT알고리즘은 현재 남은 시간을 기준으로 프로세스를 할당한다. 따라서 현재 도착한 프로세스 중에 남은시간을 반환하는 getLeaveOperation함수를 사용하여 가장 높은 우선순위의 인덱스를 구하여 s에 저장한다.
2. 만약 min이 초기값 그대로 999라면 현재 프로세스가 없다는 의미이므로 none프로세스를 list에 삽입한다.
3. 만약 프로세스를 할당했다면 getDispatch로 첫 할당이면 응답시간을 구하여 저장한다.
4. 프로세스의 남은 시간이 타임슬라이스보다 크다면 현재 남은시간에서 타임슬라이스를 빼고, list에 runtime이 타임슬라이스가 되도록 저장한다.

5. 만약 남은시간이 타임슬라이보다 작다면 리스트에는 runtime이 남은 시간이 되도록 저장한다. 만약 남은 시간이 0이라면 종료된 프로세스를 나타내는 Pcnt를 증가시키고 대기시간과 반환시간을 구하여 저장한다.

4-7. HRN 스케줄링 알고리즘

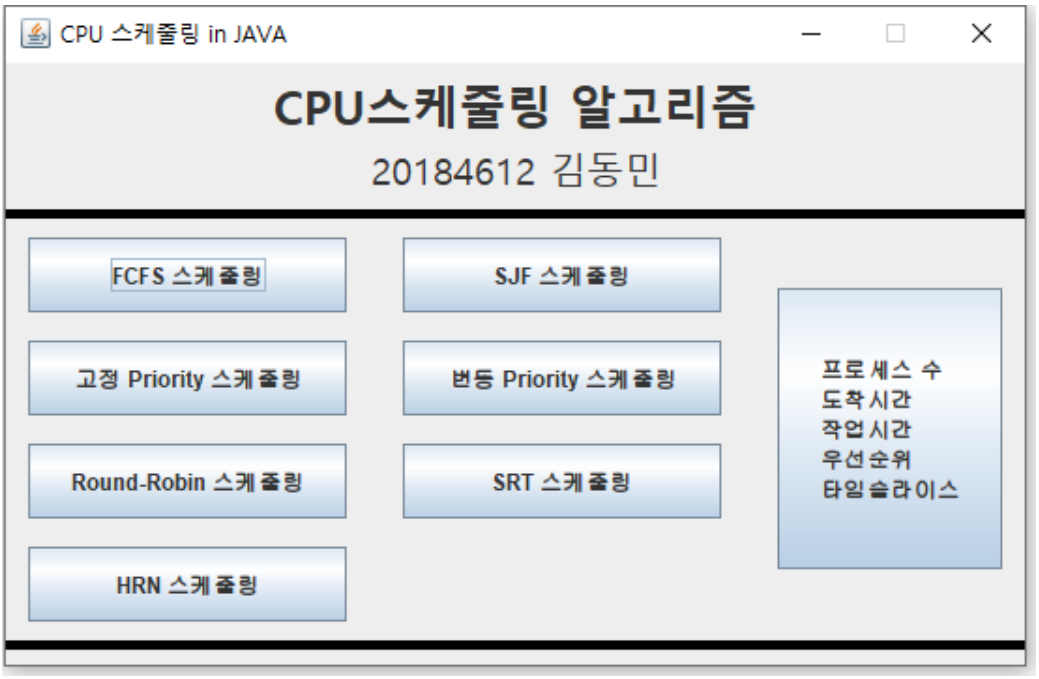
```
for(int t = 0; Pcnt<n;) {
    float hrr = -999;
    for(int i=0; i<n; i++) {
        if(ReadyQ[i].getArrivalTime()<=t && !ReadyQ[i].getDispatch()) {
            priority = ((float)ReadyQ[i].getOperationTime()+(float)(t-ReadyQ[i].getArrivalTime()))/(float)ReadyQ[i].getOperationTime();
            if(hrr<priority) {
                hrr = priority;
                c = i;
            }
        }
    }
    if (hrr==-999) {
         Process tmp = new Process("none");
            //  tmp.setColor(Color.gray);
            //  System.out.println(tmp.getColor());
            if(list.size()>0) {
                if(list.get(list.size()-1).getProcessName().equals("none")) {
                    list.get(list.size()-1).runtime+=1;
                }
                else list.add(new ProcessDispatch(tmp, 1));
            }
            else list.add(new ProcessDispatch(tmp, 1));
       t++;
     }
    else {
    t+=ReadyQ[c].getOperationTime();
    ReadyQ[c].setDispatch(true);
    list.add(new ProcessDispatch(ReadyQ[c]));
    ReadyQ[c].setTurnaroundTime(t-ReadyQ[c].getArrivalTime());
    ReadyQ[c].setResponseTime(t-ReadyQ[c].getArrivalTime()-ReadyQ[c].getOperationTime());
    ReadyQ[c].setWaitingTime(t-ReadyQ[c].getArrivalTime()-ReadyQ[c].getOperationTime());
    Pcnt++;
    }
//  ReadyQ[loc]

}
Arrays.sort(ReadyQ, new Comparator<Process>() {
    @Override
    public int compare(Process o1, Process o2) {
        // TODO Auto-generated method stub
        return o1.ProcessName.compareTo(o2.ProcessName);
    }

});


GanttColor();

for(int i = 0; i<n; i++) {
    AvgWaiting += ReadyQ[i].getWaitingTime();
    AvgResponse += ReadyQ[i].getResponseTime();
    AvgTurnAround+=ReadyQ[i].getTurnaroundTime();
```

1. HRN은 다른 스케줄링 알고리즘과는 달리 HRN만의 우선순위 공식으로 우선순위를 구한다. 따라서 어떤 프로세스를 할당할지 선택하는 알고리즘에서 HRN우선순위 공식을 사용하여 구한다.
2. (대기시간+CPU사용시간)/CPU사용시간으로 구한 값을 priority에 저장하고 hrr에 가장 높은 우선순위를 구하여 저장한다.
3. 만약 hrr이 초기값이라면 프로세스가 없다는 의미이므로 none프로세를

list에 삽입하고 현재 시간 t를 증가시킨다.

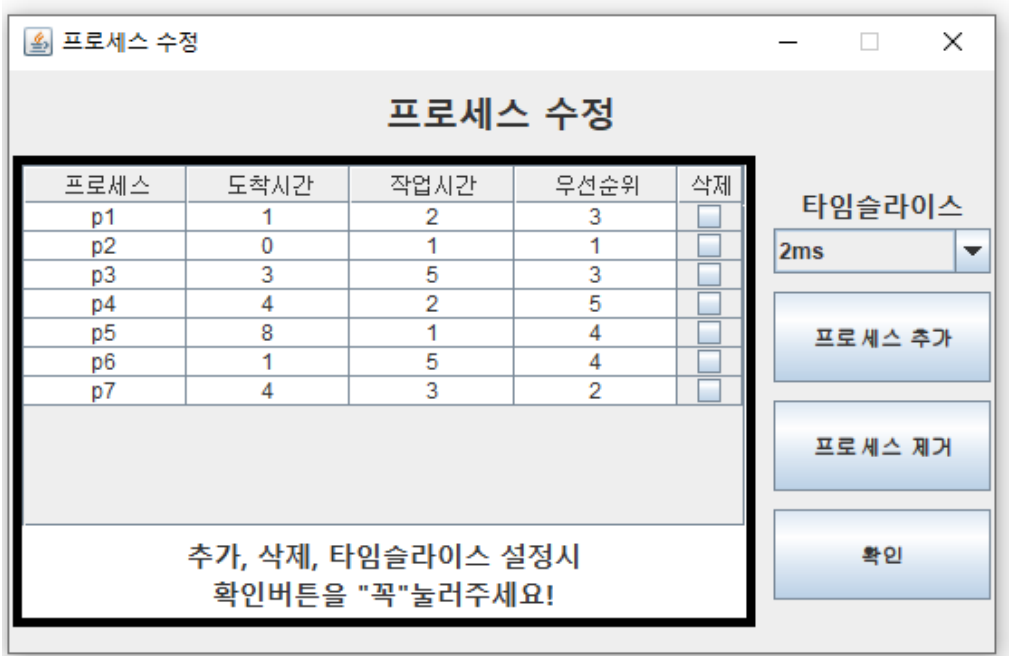
4. Else는 프로세스를 할당한다는 의미이므로 t에 프로세스의 실행시간을 더하고 list에 할당한 프로세스를 삽입한다. 마지막으로 대기시간, 응답시간, 반환시간을 구하여 저장한다.
5. 맨 처음 프로세스 도착 순서대로 프로세스를 정렬했기 때문에 다시 레디큐를 이름 순서대로 p1부터 시작되게 정렬한다.
6. GanttColor함수로 프로세스의 색깔을 지정하고 AvgWaiting, AvgResponse, AvgTurnAround에 각 대기, 응답 반환시간을 저장하고 PrintFrame함수로 스케줄링 화면에 출력하도록 한다.

5. 개발한 UI 특징



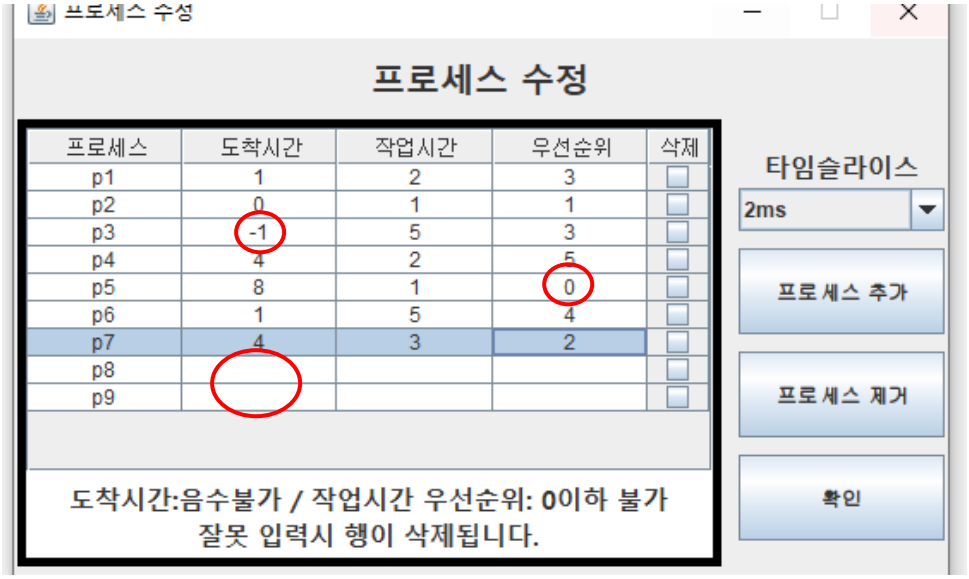
메인 인터페이스

실행파일을 실행시키면 가장 먼저 보이는 메인 인터페이스이다. 학번과 이름이 출력 되어있다. 각 스케줄링 버튼을 누르면 새로운 창이 열리면서 선택한 스케줄링 알고리즘결과를 출력한다. 오른쪽의 프로세스 버튼을 누르면 프로세스를 추가, 삭제할 수 있고 타임슬라이스를 조정할 수 있다.



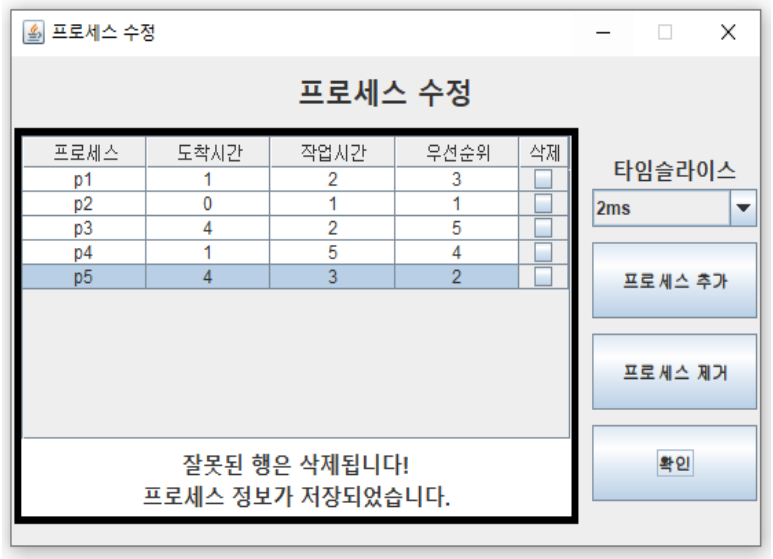
프로세스 수정 인터페이스

프로세스를 추가, 삭제할 수 있는 인터페이스이다. 프로세스 이름은 수정이 불가하고 자동으로 추가, 삭제할 때마다 이름이 변경된다.



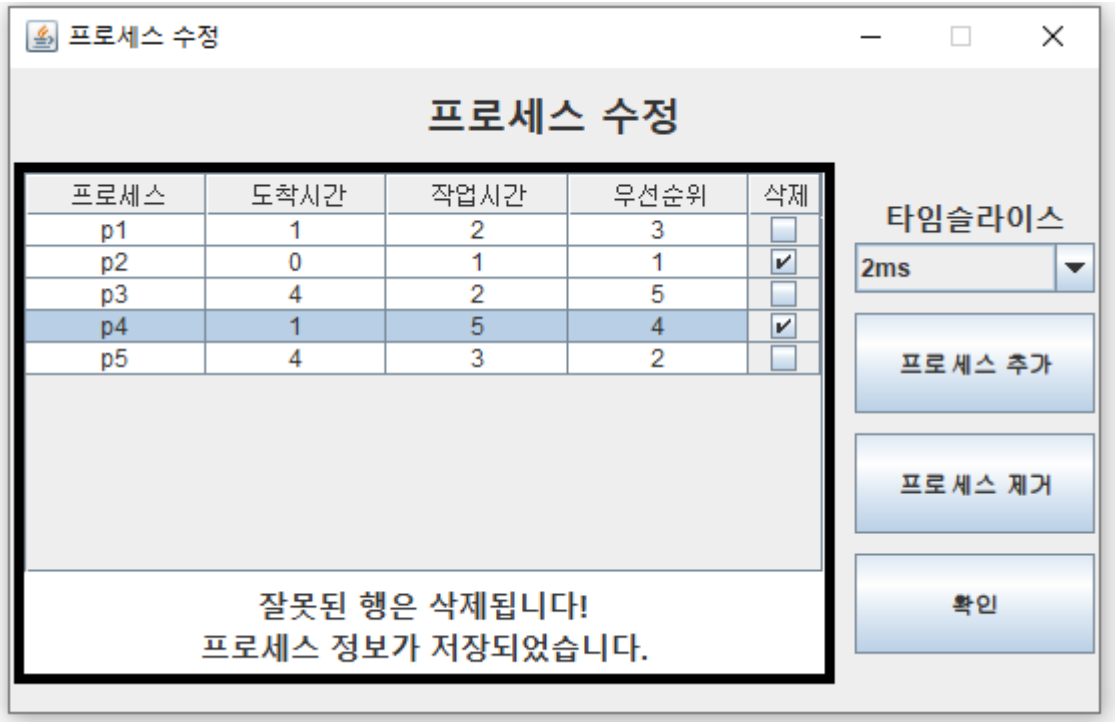
프로세스 추가버튼 클릭 시

프로세스 이름은 자동으로 입력되고 도착시간, 작업시간, 우선순위를 입력해야 한다. 도착시간은 음수 입력이 불가하고, 작업시간과 우선순위는 1이상의 값이 입력되어야 정상적인 저장이 가능하다. 만약 잘못된 입력이 있다면 확인버튼을 눌렀을 때 행이 자동으로 삭제된다.



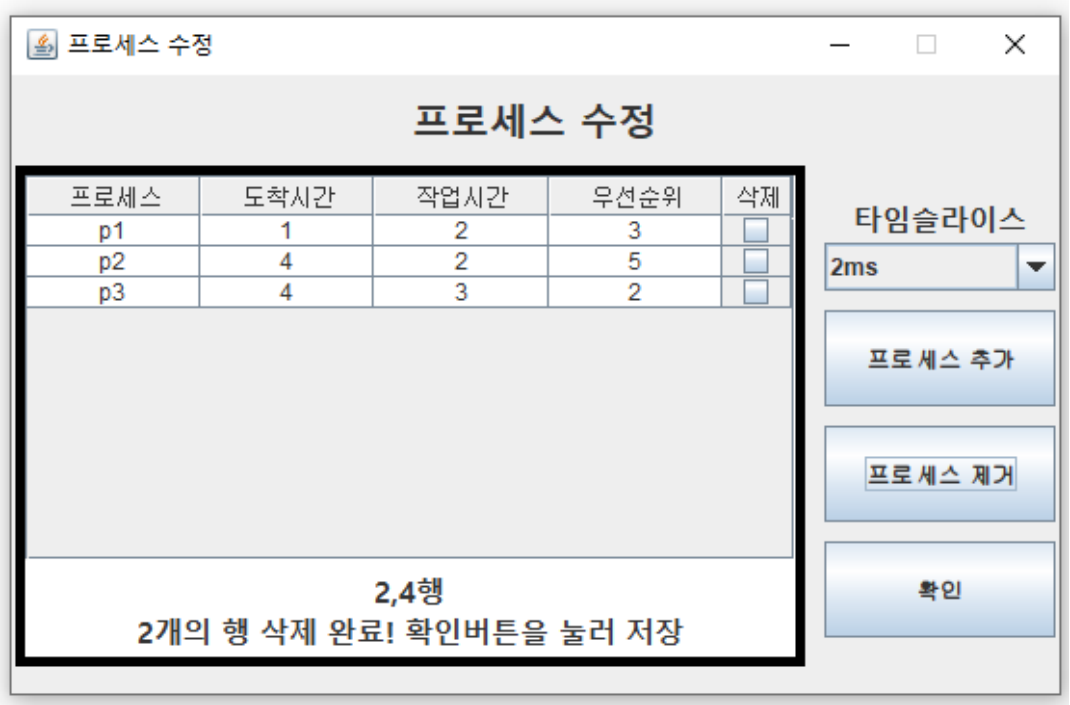
확인 버튼 클릭 시

위의 프로세스 추가창에서 p3의 도착시간이 -1로 되어있고, p5의 우선순위가 0으로 되어있다. 그리고 p8과 p9는 아예 입력이 안되어 있다. 확인 버튼을 누르면 이 잘못 입력된 행들을 삭제한다. 프로세스 이름은 다시 행 번호 순서대로 매겨진다.



프로세스 제거 시

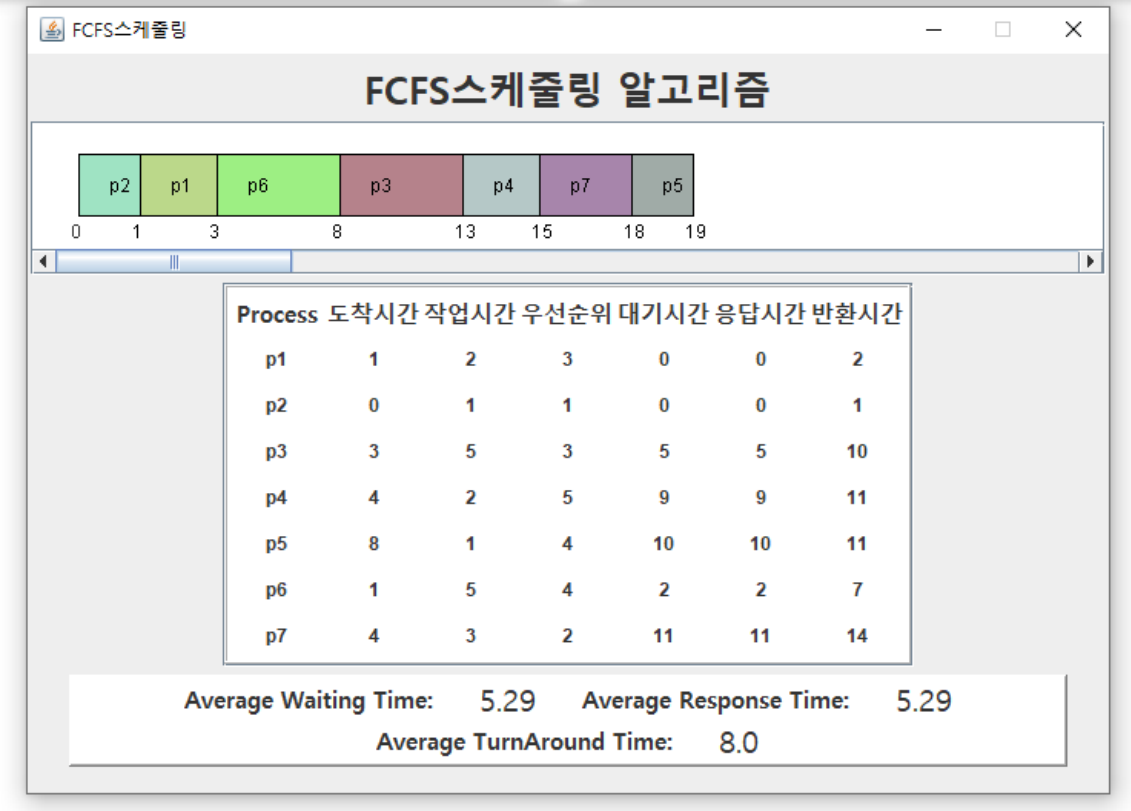
원하는 행의 버튼을 클릭한다. 여러 개의 행도 제거가 가능하다.



프로세스 제거 버튼을 누르면 어느 행이 제거되었는지 출력한다. 만약 확인 버튼을 클릭하지 않는다면 파일에 저장이 되지 않으므로 확인버튼을 눌러 저장하도록 한다.

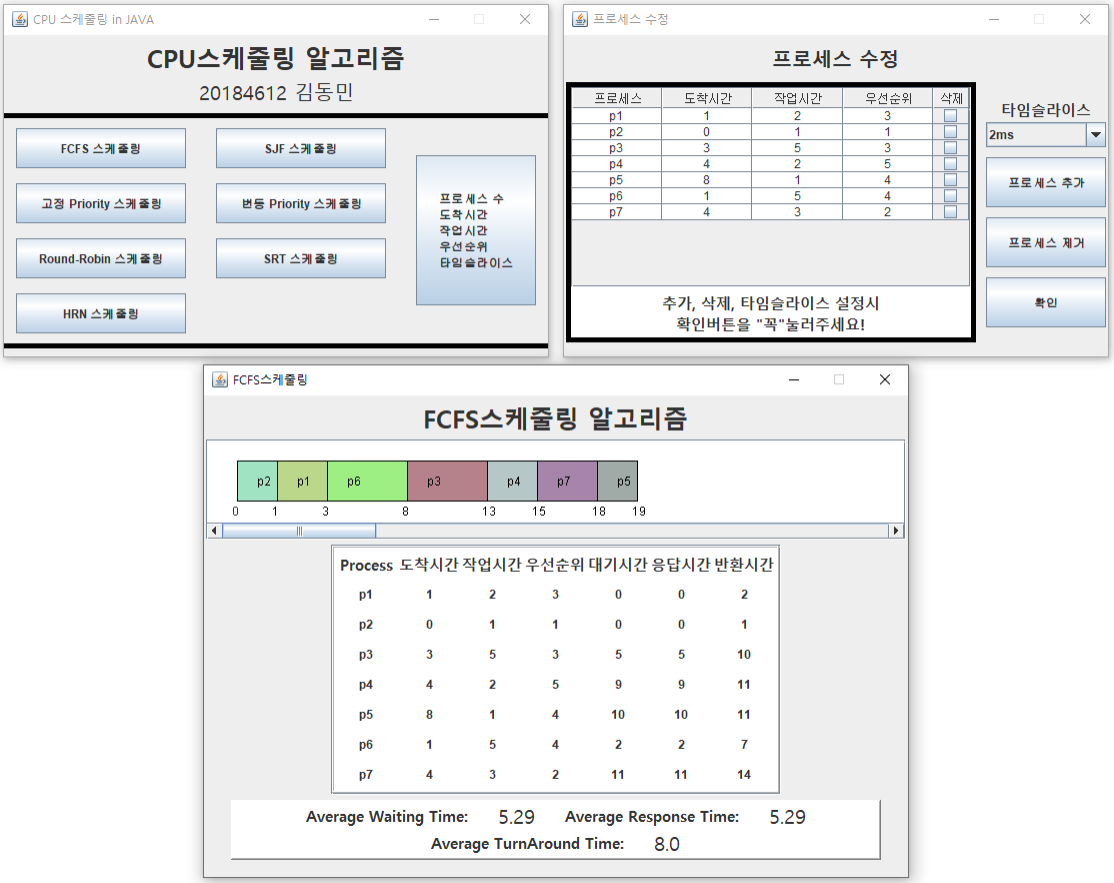
타임슬라이스 수정 시

타임슬라이스를 바꾸면 아래의 출력 창에 바꾼 타임슬라이스 값을 표시한다. 타임슬라이스 또한 확인버튼을 눌러야 파일에 저장이 가능하다.



스케줄링 버튼 클릭 시

클릭한 스케줄링에 맞는 프레임을 출력한다. 프레임은 여러 번 누른다고 여러 개가 생기지 않고 딱 하나의 프레임만 생성된다. 다른 알고리즘을 선택하면 원래 화면에 띄워진 프레임은 사라지고 새로운 프레임이 생긴다.

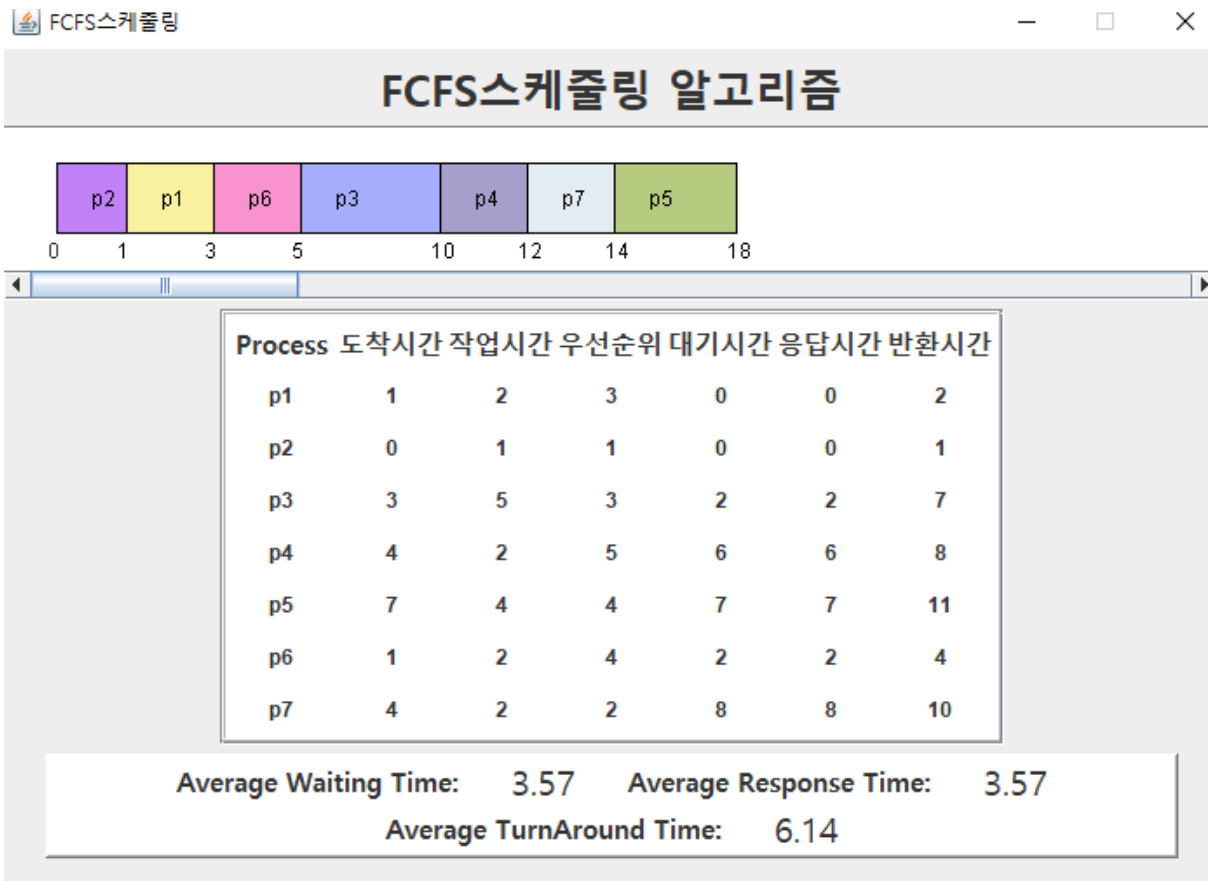


전체 창

모든 버튼을 클릭하게 되면 사용자가 한 눈에 볼 수 있도록 프레임을 배치했다. 만약 다른 프로세스 예시를 테스트해보고 싶다면 프로세스 수정 창에서 프로세스를 바꾼 후 확인 버튼을 눌러 파일에 저장하고 스케줄링 버튼을 누른다. 그러면 사용자는 새로운 프로세스로 알고리즘의 동작을 확인할 수 있다.

6. 각 알고리즘 실행 결과 캡쳐 및 설명

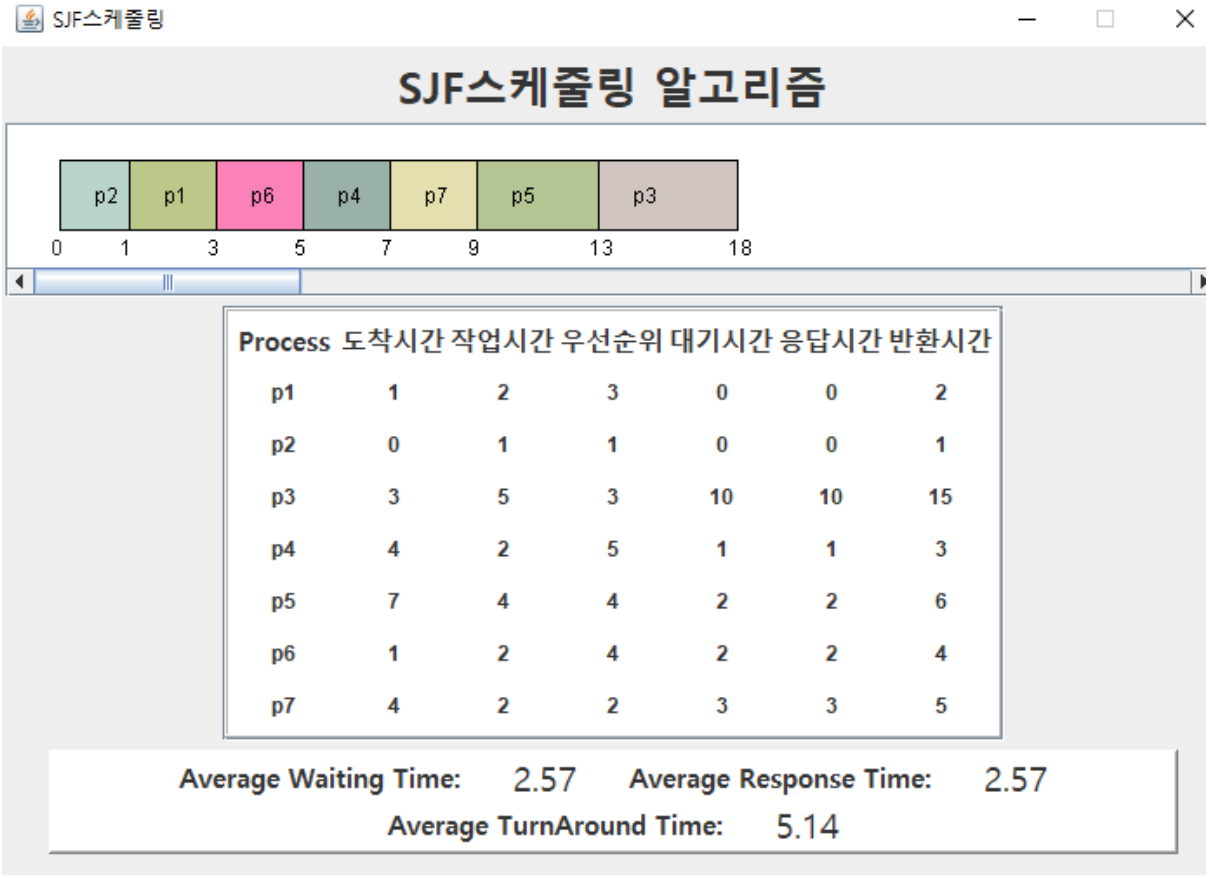
4-1. FCFS 스케줄링 알고리즘



FCFS스케줄링: 준비 큐에 도착한 순서대로 CPU를 할당하는 비선점형 스케줄링 방식. 처리시간이 긴 프로세스가 CPU를 차지하면 다른 프로세스들은 하염없이 기다려 효율성이 떨어지는 콘보이 효과가 발생한다.

p2의 도착시간이 0으로 가장 먼저 CPU를 할당 받고, 그 뒤로 도착하는 순서대로 CPU에 할당된다. FCFS스케줄링이라는 것을 제목과 프레임의 타이틀을 통해 알 수 있고, 그 밑의 간트차트를 통해 어떤 프로세스가 CPU에 할당되었는지 바로 확인이 가능하다. 프로세스가 순서대로 정렬되어 있고, 대기시간, 응답시간, 반환시간과 평균 대기시간, 평균 응답시간, 평균 반환시간이 한눈에 확인 가능하다.

4-2. SJF 스케줄링 알고리즘

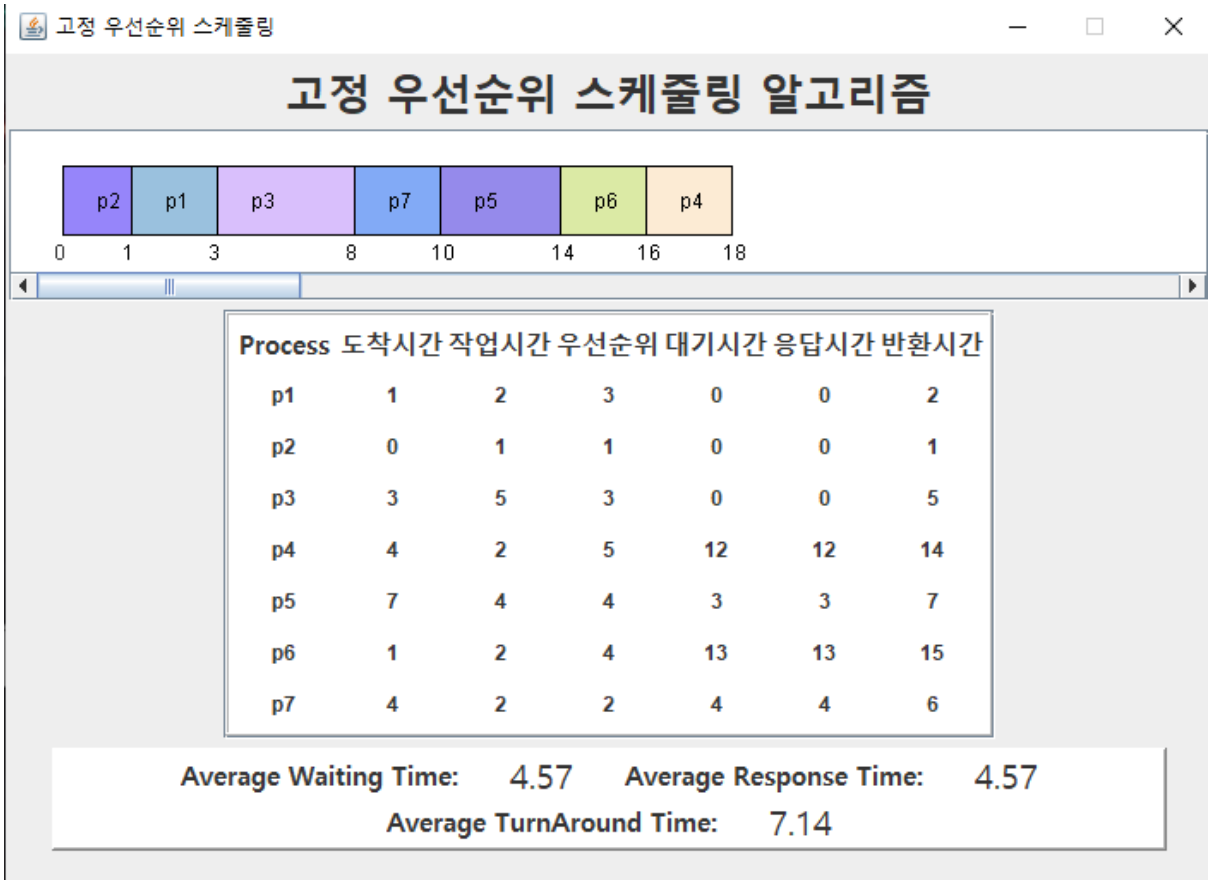


SJF스케줄링: 준비 큐에 있는 프로세스 중에서 실행시간이 가장 짧은 작업부터 CPU를 할당하는 비선점형 방식 FCFS의 콘보이 효과를 완화하여 시스템의 효율성을 높인다.

P2프로세스는 도착시간이 0이기 때문에 가장 먼저 CPU를 할당 받고, 시간이 1이 되면 p1과 p6가 준비 큐에 들어온다. P1과 p6는 작업시간이 같으므로 먼저 들어온 p1이 먼저 할당되는 모습을 볼 수 있다.

여기서 p3프로세스의 도착시간은 3ms로 다른 프로세스보다 빨리 준비 큐에 들어온 모습을 볼 수 있다. 하지만 작업시간이 5로 그 뒤에 들어오는 p4, p5, p7의 프로세스보다 실행시간이 길다. 때문에 가장 늦게 실행되어 아사현상이 일어나는 모습을 볼 수 있다.

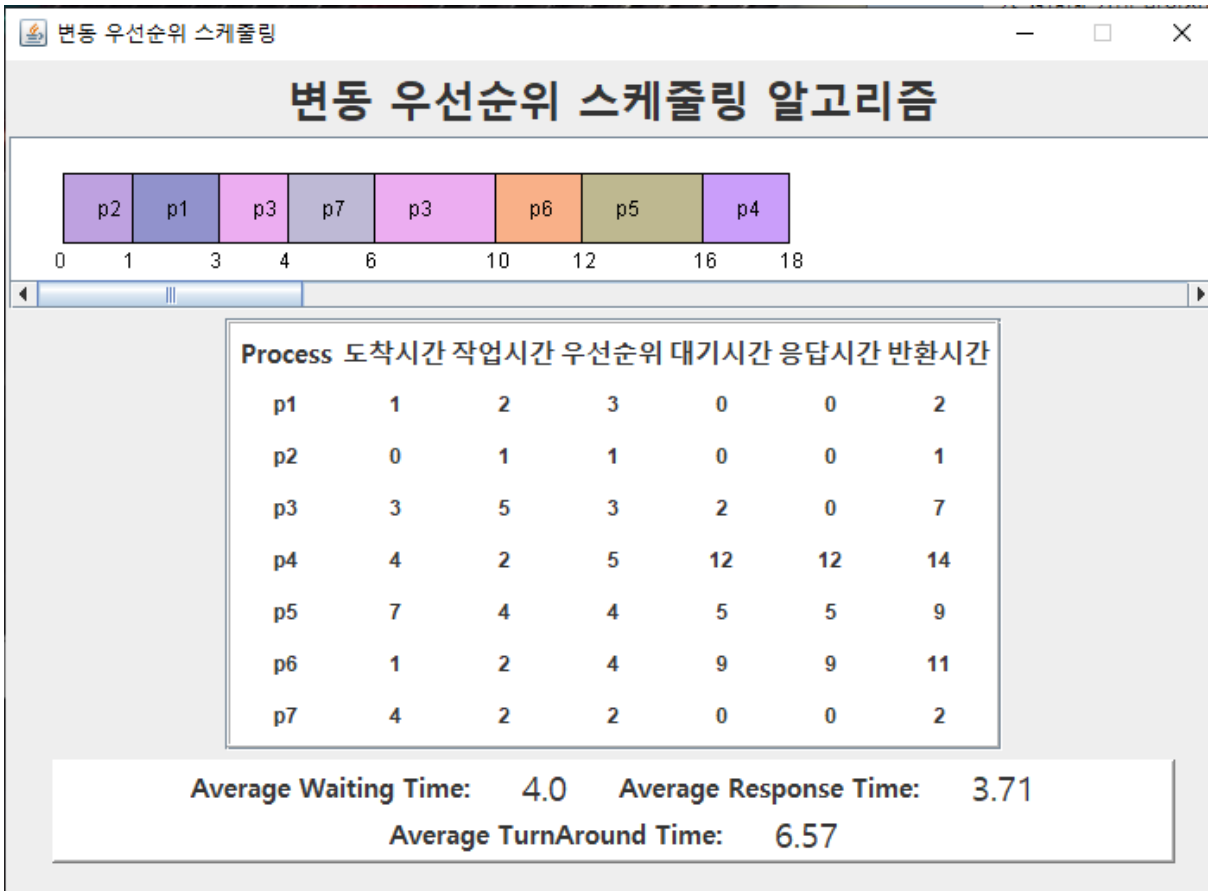
4-3. 고정 우선순위 스케줄링 알고리즘



고정 우선순위 스케줄링: 한번 우선순위를 부여받으면 종료될 때까지 우선순위가 고정된다.

P1과 p6의 도착시간은 1ms로 같은 시간에 도착했지만 p1의 우선순위는 3, p6의 우선순위는 4로 p1이 먼저 실행된다. 그리고 뒤에 들어오는 p7, p5가 우선순위가 더 높기 때문에 먼저 CPU를 할당 받게 된다. P4의 경우 우선순위가 5로 가장 낮기 때문에 4ms에 도착했지만 가장 늦게 실행되는 모습을 볼 수 있다.

4-4. 변동 우선순위 스케줄링 알고리즘

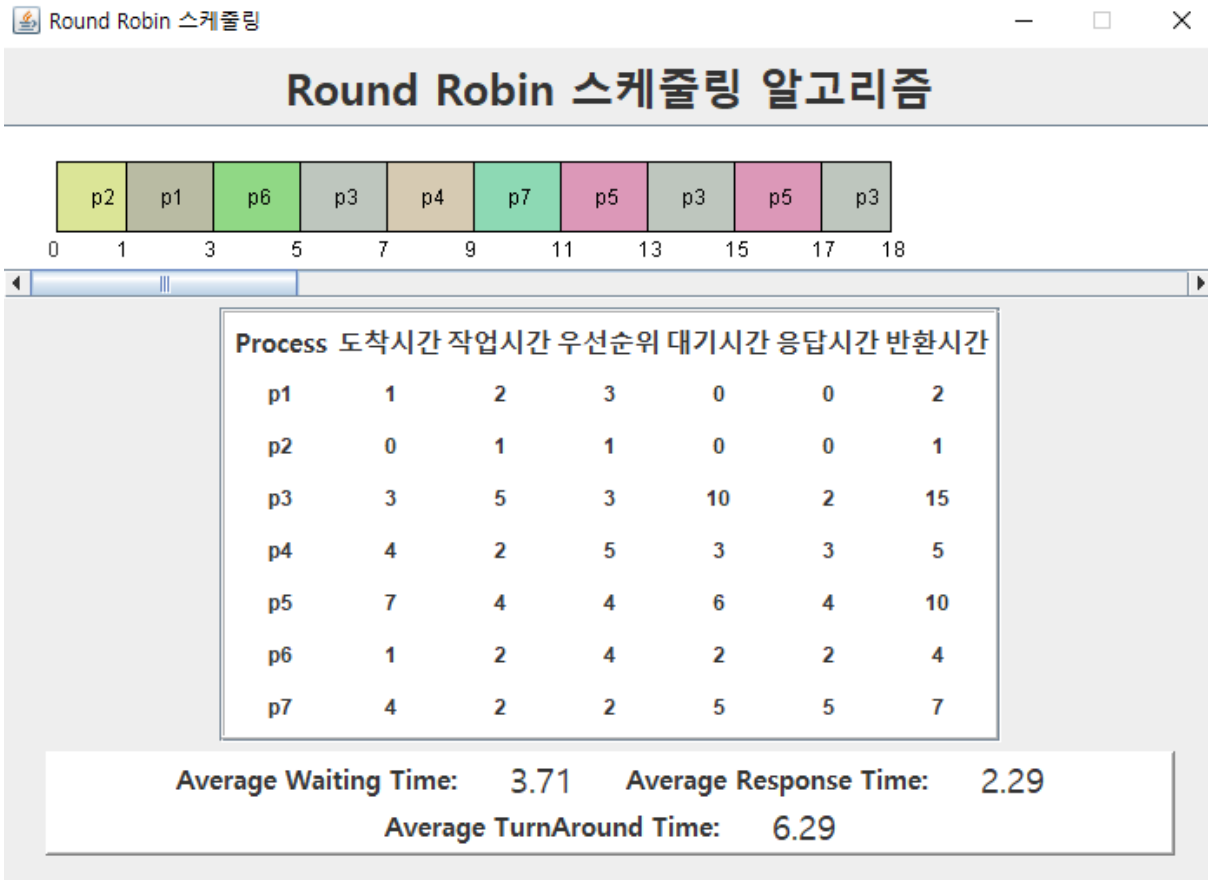


변동 우선순위 스케줄링: 일정 시간마다 우선순위가 변한다.

P3프로세스는 3ms에 도착하여 바로 실행된다. 하지만 실행 도중, 4ms때 우선순위가 2인 p7프로세스가 들어오게 된다. 그러면 p3는 타임아웃으로 나가고 p7이 새롭게 할당된다. P7프로세스가 완료되었다면 준비 큐에 남아있는 프로세스 중 우선순위가 가장 높은 프로세스가 p3이기 때문에 다시 p3를 할당하게 된다.

P6와 p5는 우선순위가 같지만 p6가 먼저 실행된 이유는 p6의 도착시간이 1ms로 p5 도착시간의 7ms보다 먼저 준비 큐에 먼저 들어왔기 때문이다.

4-5. Round-Robin 스케줄링 알고리즘

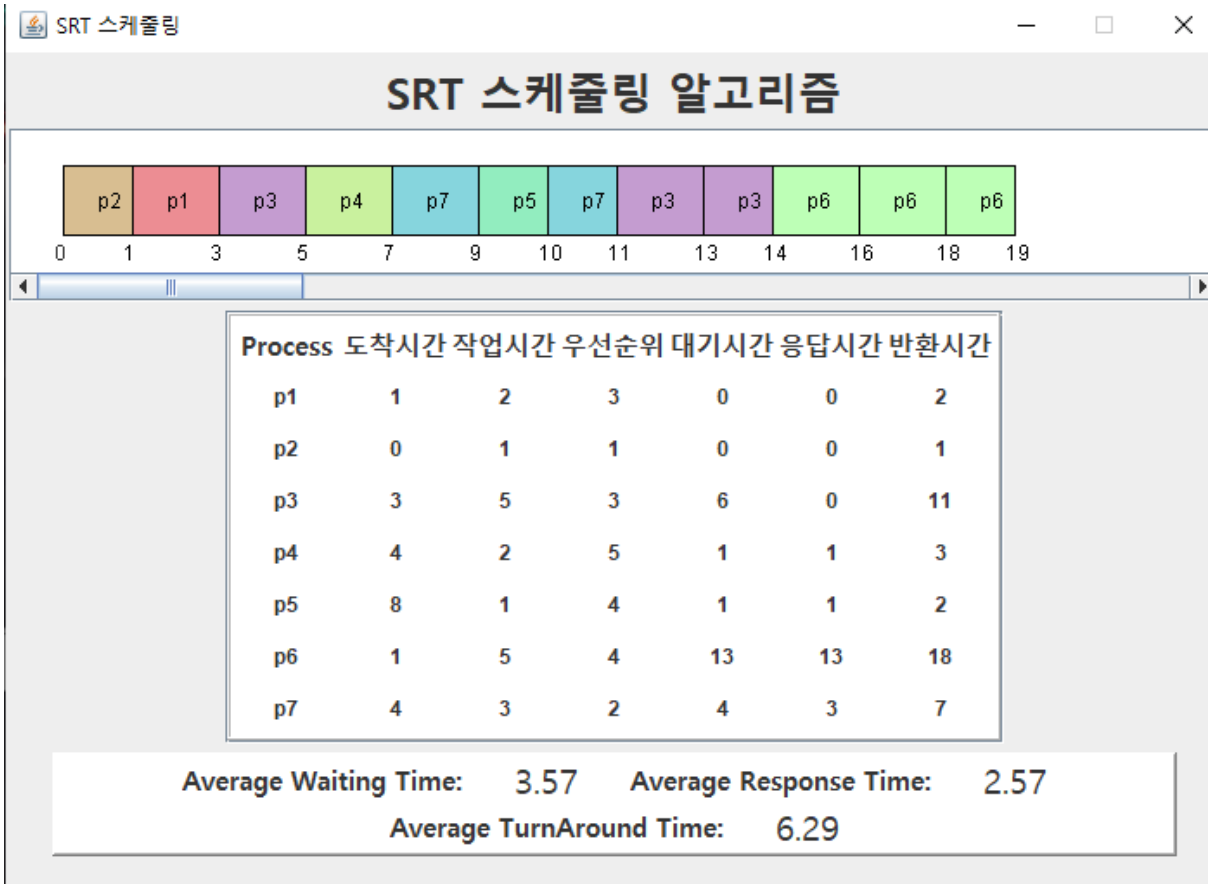


Round Robin스케줄링: 한 프로세스가 타임슬라이스 동안 작업을 하다가 완료하지 못하면 큐의 맨 뒤로 가서 자기 차례를 기다리는 방식

이 프로세스 예제의 경우 타임슬라이스는 2로 만약 프로세스의 실행시간이 2이상이면 중간에 타임아웃이 된다. P3프로세스의 경우 5ms때 실행이 되었지만 실행시간이 5로 3이 남게 되어 준비 큐의 마지막으로 가게 된다. 그리고 11ms때 p5프로세스도 실행되지만 작업시간이 4이기 때문에 2가 남아 준비큐의 마지막으로 가고, p3가 실행되지만 실행시간이 1이 남아 다시 준비 큐로 들어가게 된다.

마지막 p3는 실행시간이 1ms가 남아있기 때문에 1ms만 실행하고 모든 할당이 끝나게 된다.

4-6. SRT 스케줄링 알고리즘



SRT스케줄링: SJF와 Round-Robin을 혼합한 방식. 기본적으로 Round-Robin스케줄링을 사용하지만 CPU를 할당받을 프로세스를 선택할 때 남아있는 작업시간이 가장 적은 프로세스를 선택한다.

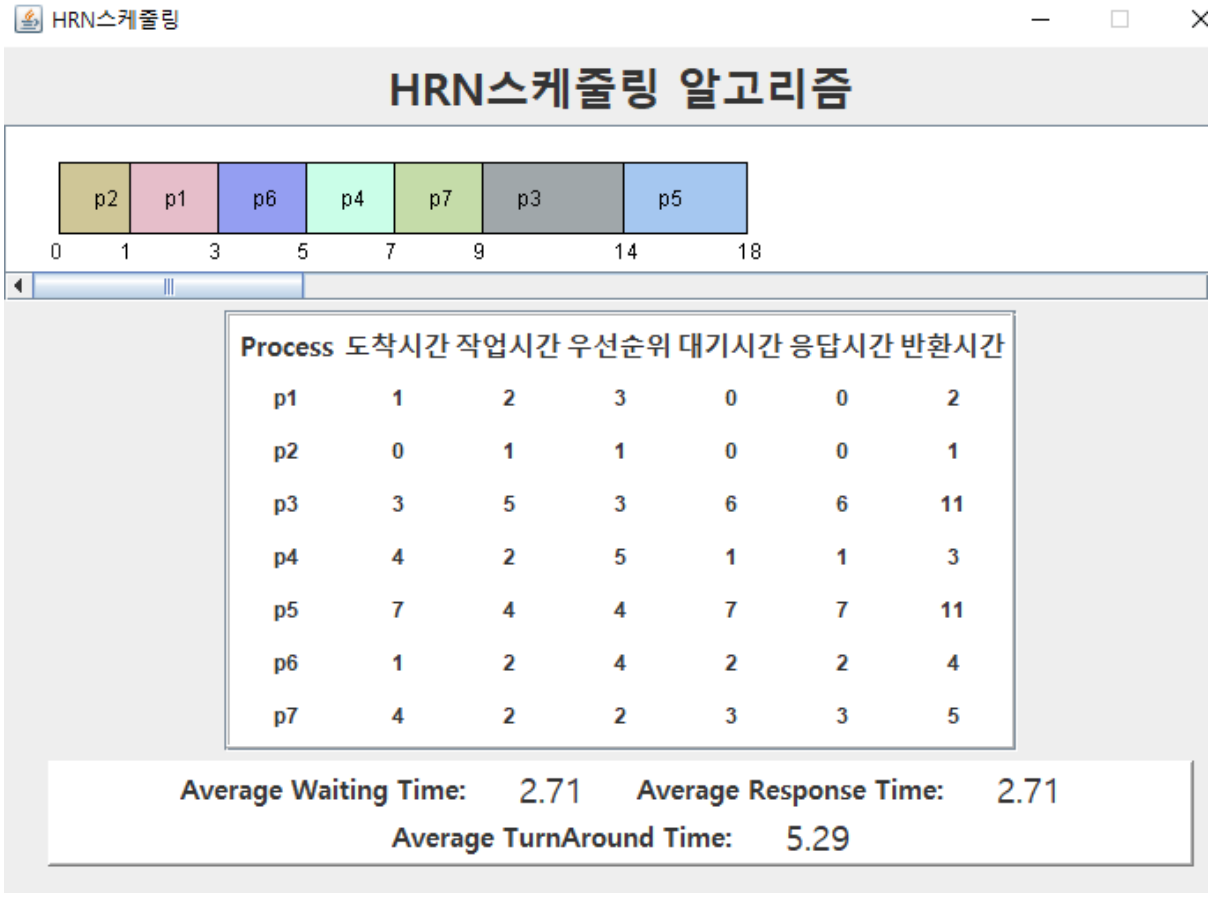
이번 경우는 SRT알고리즘이 잘 보일 수 있도록 프로세스를 약간 수정하여 실행했다. 타임슬라이스는 2로 설정했다.

3ms때 도착한 p3는 준비 큐에 비교할 대상이 없기 때문에 일단 실행되고 타임슬라이스가 2이기 때문에 작업시간이 3이 남게 된다. 그리고 뒤에 들어오는 p4가 작업시간이 2이기 때문에 작업시간이 더 작은 p4를 먼저 할당하게 된다.

P7의 경우도 작업시간 3에서 2를 사용했기 때문에 1이 남게 되고, 준비 큐에 다시 들어가고 작업시간 1인 p5를 먼저 할당하고 p7를 할당하게 된다.

11ms가 되면 P3는 p6보다 작업시간이 더 적게 남았기 때문에 p3를 먼저 할당한다.

4-7. HRN 스케줄링 알고리즘

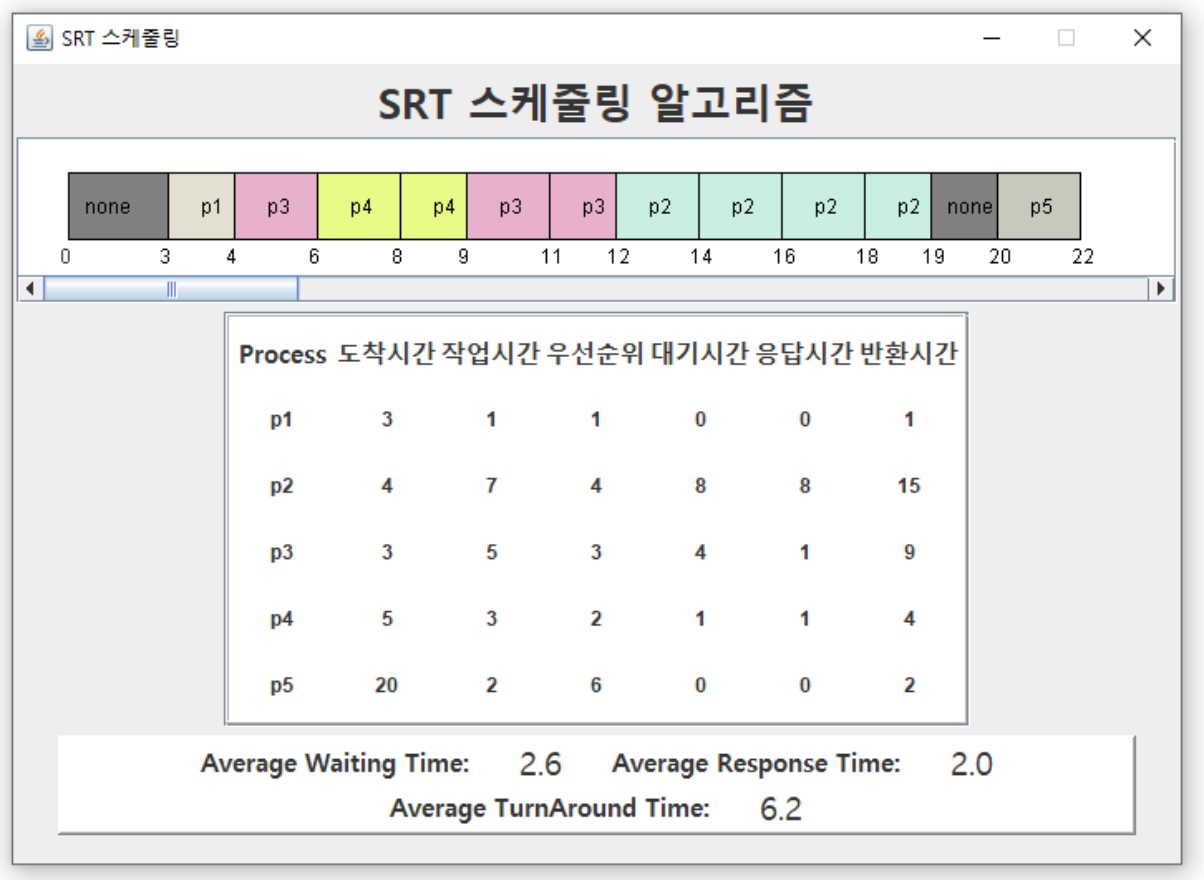


HRN스케줄링: SJF스케줄링에서 발생할 수 있는 아사현상을 해결한다. 실행시간이 짧은 프로세스의 우선순위를 높게 설정하면서도 대기시간을 고려하여 아사현상을 완화한다.

HRN스케줄링에서 프로세스의 우선순위를 결정한다. 우선순위 = (대기시간+CPU사용시간)/CPU사용시간

9ms때 p3의 우선순위는 (대기시간(6)+CPU사용시간(5))/CPU사용시간(5) = 1.2이고 p5의 우선순위는 (대기시간(2)+CPU사용시간(4))/CPU사용시간(4) = 0.5로 HRN스케줄링에서는 숫자가 클수록 우선순위가 높기 때문에 p3를 먼저 실행하고 p5를 실행한 모습을 볼 수 있다.

4-8 중간에 프로세스가 끊길 때



프로세스 예시를 테스트하던 중 만약 중간에 프로세스가 끊긴다면 어떻게 동작할지 의문이 들었다.

그래서 그에 맞는 프로세스 예시를 넣어보았고, 역시나 오류가 발생했다. 그래서 이 부분을 해결하기 위해 만약 프로세스가 입력되지 않은 상황이라면 간트차트에 none을 표시할 수 있도록 약간의 코드를 추가했고 알맞게 동작했다.

7. 느낀 점

이번 프로젝트를 진행할 땐 작년에 자바를 써본 경험으로 자바 GUI를 이용하여 프로그래밍하기로 생각했습니다. 저번에 프로그래밍을 했을 때는 코드를 한 함수에 몰아서 작성하여 가독성이 너무 떨어져서 다시 봤을 때 불편한 점들이 많이 있었습니다. 이번 과제에서는 이 부분을 줄여보려고 노력했고, 처음에는 잘 되는가 싶다가도 코드의 양이 많아지자 다시 가독성이 떨어지는 부분들이 생겼습니다. 이런 부분들에 대해서는 아직 많이 부족하다는 것을 느꼈고 함수형 프로그래밍에 대해 조금 더 배워야겠다는 생각을 했습니다.

처음 스케줄링을 배울 땐 7가지나 되는 알고리즘을 어떻게 이해하고 프로그래밍을 해야할지 막막함이 먼저 들었습니다. 스케줄링 알고리즘을 작성할 때는 먼저 어떤 프로세스가 먼저 할당되고, 대기시간, 응답시간, 반환시간이 어떻게 출력되는지 직접 손으로 써가며 알고리즘의 동작을 확인하고 코드를 작성했습니다. 먼저 어떤 값이 출력되어야 하는지 확인을 하고 코드를 짜니 더 쉽고 편하게 코드를 짤 수 있었던 것 같습니다.

스케줄링 알고리즘을 모두 완성하고 함께 수업을 듣는 동기들과 프로세스 예시를 만들고 서로 공유해보며 많은 테스트를 진행했습니다. 이를 통해 저 혼자 예시를 만들어 보는 것보다 더 많은 프로세스들을 테스트할 수 있었습니다, 그 중에 잘못 출력되는 부분이나 오류를 찾을 수 있었고 이를 빠르게 고칠 수 있는 기회가 되었습니다. 이를 통해 협력의 중요성을 깨달을 수 있었습니다.

이번 스케줄링 프로젝트를 통해 평소에 제가 부족했던 부분을 깨닫고 방학을 이용해 어떤 것들을 해야할지 알 수 있는 계기가 되었습니다. 방학이 되었다고 해서 놀지 않고 저만의 프로젝트를 작성해보며 제 프로그래밍 실력을 늘릴 수 있도록 노력하겠습니다.

한 학기동안 감사했습니다! 다음 학기 때 뵙겠습니다!!