제출일	2023.05.29.	전 공	컴퓨터소프트웨어공학과
과 목	운영체제	학 번	20194009
담당 교수	김 대 영 교수님	이 름	이 준 석

<목차>

<1-1. FCFS스케줄링 코드>

```java
import java.util.*;
public class FCFS_Scheduling extends ProcessSort{
    public void run() {
        process= FileOpen();
        Deque<String> q = new LinkedList<>();
        ArriveTimeSort(); //도착시간대로 정렬

        for (int i = 1; i <= ProcessCount; i++) {
            for (int k = 1; k <= ProcessCount; k++) {
                StringTokenizer ProcessToken = new StringTokenizer(process[k]);
                    ProcessId = ProcessToken.nextToken();
                    ArriveTime = Integer.parseInt(ProcessToken.nextToken());
                    ServiceTime =Integer.parseInt(ProcessToken.nextToken());
                    ProcessToken.nextToken();ProcessToken.nextToken();
                    if (tmp_time[i] == ArriveTime){ //도착시간 순으로 q에 삽입
                        q.add(process[k]);
                    }
            }
        }
        System.out.println("비선점형 - FCFS스케줄링");
        Nonpreemptive_Print_Process print_process=new Nonpreemptive_Print_Process();
        print_process.print(q, ProcessCount);
    }
}
```

<1-2. SJF스케줄링 코드>

```
import java.util.*;
public class SJF_Scheduling extends ProcessSort{
    public void run(){
        process= FileOpen();
        ArriveTimeSort(); //도착시간대로 정렬

        for(int i = 1; i<= ProcessCount; i++){
            for(int k = 1; k<= ProcessCount; k++){
                int count=0;
                StringTokenizer processToken=new StringTokenizer(process[k]);
                while(processToken.hasMoreTokens()){
                    ProcessId =processToken.nextToken();
                    ArriveTime =Integer.parseInt(processToken.nextToken());
                    ServiceTime =Integer.parseInt(processToken.nextToken());
                    processToken.nextToken();processToken.nextToken();
                    /*
                    이미 덱 안에 동일한 프로세스가 들어가 있는지 확인
                    */
                    Iterator it=deque.iterator();
                    while(it.hasNext()){
                        String s= (String) it.next();
                        StringTokenizer QToken=new StringTokenizer(s);
                        while(QToken.hasMoreTokens()){
                            String Id=QToken.nextToken();
                            if(Id.equals(ProcessId)) //덱에 동일한 프로세스가 있는지 확인
                                count++;
                        }
                    }

                if(count==0) { //동일한 프로세스가 없는 경우
                    if (ServiceTime == tmp_servicetime[i]) { //정렬된 tmp_servicetime에 의해 가장 짧은 실행시간 프로세스를 덱에 삽입
                        deque.add(process[k]);
                    }
                }
            }
        }
    }
    System.out.println("비선점형 - SJF스케줄링");
    Nonpreemptive_Print_Process print_process=new Nonpreemptive_Print_Process();
    print_process.print(deque, ProcessCount);
}
}
```

<1-3. HRN스케줄링 코드>

```java
import java.util.*;
public class HRN_Scheduling extends ProcessSort{
    public void run(){
        process= FileOpen();
        ArriveTimeSort(); //도착시간대로 정렬
        /*
        덱에서 하나씩 꺼내서 HRN우선순위 판별
         */
        String tmp = null;
        for(int i = 0; i< ProcessCount; i++){
            String que= deque.peekLast(); //덱의 가장 나중에 들어온 프로세스 정보
            StringTokenizer DequeToken=new StringTokenizer(que);
            double max_priority=0;
                DequeToken.nextToken();DequeToken.nextToken();
                ServiceTime =Double.parseDouble(DequeToken.nextToken());
                ServiceTimeSum += ServiceTime; //실행시간 총합
                DequeToken.nextToken();DequeToken.nextToken();

            for(int k = 1; k<= ProcessCount; k++){
                int count=0;
                StringTokenizer processToken=new StringTokenizer(process[k]);
                    ProcessId =processToken.nextToken();
                    ArriveTime =Double.parseDouble(processToken.nextToken());
                    ServiceTime =Double.parseDouble(processToken.nextToken());
                    /*
                    이미 덱 안에 동일한 프로세스가 들어가 있는지 확인
                     */
                    Iterator it= deque.iterator();
                    while(it.hasNext()){
                        String s= (String) it.next();
                    StringTokenizer QueueToken=new StringTokenizer(s);
                    while(QueueToken.hasMoreTokens()){
                        String Id=QueueToken.nextToken();
                        if(Id.equals(ProcessId)) //덱 안에 동일한 프로세스가 있다면 count증가
                            count++;
                    }
                }
                if(count==0) { //덱에 동일한 프로세스가 없는 경우, 실제 우선순위를 구하는 작업을 하는 조건문
                    double m=(ServiceTime +(ServiceTimeSum- ArriveTime))/ ServiceTime; //(대기시간+CPU사용률)/CPU사용률
                    if(max_priority<m){
                        max_priority=Math.max(max_priority,m);
                        tmp=process[k]; //우선순위가 가장 빠른 프로세스를 tmp에 저장
                    }
                }
            }
            deque.add(tmp); //우선순위가 가장 높은 프로세스를 덱에 삽입
        }
        deque.pollLast(); //마지막 요소 삭제
        System.out.println("비선점형 - HRN스케줄링");
        Nonpreemptive_Print_Process print_process=new Nonpreemptive_Print_Process();
        print_process.print(deque, ProcessCount);
    }
}
```

<1-4. 비선점 우선순위 스케줄링 코드>

```java
import java.util.*;
public class NonpreemptivePriority_Scheduling extends ProcessSort{
    public void run(){
        process= FileOpen();
        Deque<String> q = new LinkedList<>();
        Queue<String> tmp_q = new LinkedList<>(); //동일한 우선순위일 경우 도착시간에 따른 정렬까지 마침
        ArriveTimeSort(); //도착시간대로 정렬
        TimeSort(); //정렬된 도착시간에 맞게 프로세스들의 각종 정보들 정렬

        String[] tmp_arr = new String[ProcessCount -1];
        String[] arr = new String[ProcessCount -1];
        int[] tmp_priority1 = new int[ProcessCount -1]; //P1프로세스를 제외한 나머지 프로세스들 우선순위 배열
        int[] tmp_servicetime1 = new int[ProcessCount -1]; //P1프로세스를 제외한 나머지 프로세스들 실행시간 배열
        int[] tmp_arrivetime1 = new int[ProcessCount -1]; //P1프로세스를 제외한 나머지 프로세스들 도착시간 배열

        for (int i = 0; i < ProcessCount -1; i++) {
            tmp_arr[i] = NonpreemrtiveProcess[i+2]; //P2 P3 P4 P5 프로세스 정렬
        }

        for (int i = 0; i < ProcessCount -1; i++) {
            StringTokenizer processToken = new StringTokenizer(tmp_arr[i]);
                ProcessId = processToken.nextToken(); //프로세스 ID
                tmp_arrivetime1[i] = Integer.parseInt(processToken.nextToken()); //도착시간
                tmp_servicetime1[i] = Integer.parseInt(processToken.nextToken()); //실행시간
                tmp_priority1[i] = Integer.parseInt(processToken.nextToken()); //우선순위
                processToken.nextToken();
        }
        Arrays.sort(tmp_priority1); //1 2 2 4(P4 P2 P5 P3순으로 정렬)

         for(int i = 0; i< ProcessCount -1; i++){
             int PriorityCount=0;
             for(int j = 0; j< ProcessCount -1; j++){
                 StringTokenizer processToken=new StringTokenizer(tmp_arr[j]);
                 int SameProcessCount=0;
                     ProcessId =processToken.nextToken();
                     ArriveTime =Integer.parseInt(processToken.nextToken());
                     ServiceTime =Integer.parseInt(processToken.nextToken());
                     priority=Integer.parseInt(processToken.nextToken());
                     processToken.nextToken();
                 Iterator iterator=q.iterator(); //덱에 동일 프로세스가 있는지 확인
                 while(iterator.hasNext()){
                     StringTokenizer QToken=new StringTokenizer((String) iterator.next());
                     String proId=QToken.nextToken();
                     QToken.nextToken();QToken.nextToken();QToken.nextToken();QToken.nextToken();
                     if(ProcessId.equals(proId)) //동일한 프로세스가 있다면 SameProcessCount증가
                         SameProcessCount++;
                 }
                 if(tmp_priority1[i]==priority&& SameProcessCount==0){ //덱에 동일한 프로세스가 없는 경우
                     PriorityCount++; //동일한 우선순위가 있는지 확인
                     tmp_q.add(tmp_arr[j]); //임시저장소인 큐에 저장
                 }
             }
             if(PriorityCount==1){ //PriorityCount가 1인 경우는 동일한 우선순위가 없는 것을 의미
                 while(!tmp_q.isEmpty()){
                     q.add(tmp_q.poll()); //덱에 삽입
                 }
             }
             else if(PriorityCount>1){ //동일한 우선순위를 가진 프로세스가 있는 경우
                 String str1=tmp_q.poll();
```

```
                String str2=tmp_q.poll();
                StringTokenizer str1Token=new StringTokenizer(str1);
                StringTokenizer str2Token=new StringTokenizer(str2);
                    str1Token.nextToken();
                    int arrtime1=Integer.parseInt(str1Token.nextToken());
                    str1Token.nextToken();str1Token.nextToken();str1Token.nextToken();
                    str2Token.nextToken();
                    int arrtime2=Integer.parseInt(str2Token.nextToken());
                    str2Token.nextToken();str2Token.nextToken();str2Token.nextToken();
                    if(arrtime1>arrtime2) //도착시간이 더 빠른 프로세스를 덱에 저장
                        q.add(str2);
                    else
                        q.add(str1);
            }
        }
        q.addFirst(NonpreemrtiveProcess[1]); //덱에 도착시간이 0인 프로세스를 삽입
        System.out.println("비선점형 - 비선점 우선순위 스케줄링");
        Nonpreemptive_Print_Process print_process=new Nonpreemptive_Print_Process();
        print_process.print(q, ProcessCount);
    }
}
```

<1-5. RR스케줄링 코드>

```java
import java.util.*;
public class RR_Scheduling extends ProcessSort{
    public void run() {
        process = FileOpen();
        int c = 0;
        int total_servicetime = 0;
        ArriveTimeSort(); //도착시간대로 정렬
        TimeSort(); //도착시간에 맞는 프로세스들의 각종 정보들 정렬
        String[] ganttchatt = new String[ServiceTimeSum];

        while (total_servicetime != ServiceTimeSum) { //현재까지의 실행시간이 전체 실행시간이 되기 전까지 반복
            String str = q.poll(); //큐에 들어있는 프로세스를 str에 저장
            StringTokenizer strToken = new StringTokenizer(str);
            ProcessId = strToken.nextToken();
            ArriveTime = Integer.parseInt(strToken.nextToken());
            ServiceTime = Integer.parseInt(strToken.nextToken());
            strToken.nextToken();
            ResponseTime = Integer.parseInt(strToken.nextToken());

            for (int i = 1; i <= ProcessCount; i++) {
                if (tmp_processId[i].equals(ProcessId)) { //정렬된 프로세스ID와 큐에서 꺼낸 프로세스ID가 동일한 경우
                    if (total_servicetime >= ArriveTime) { //현재 실행시간안에 도착한 프로세스인 경우
                        check[i] = 0; //한번 검사했다는 걸 표시

                        if (SaveServiceTime[i] >= TimeQuantum) { //해당 프로세스의 실행시간이 타임슬라이스보다 큰 경우
                            for (int j = 1; j <= TimeQuantum; j++) {
                                wait_time[i] += (total_servicetime - tmp_arrivetime[i]); //대기시간 저장
                                SaveServiceTime[i] -= 1; //해당 프로세스의 남은 실행시간 1초 감소
                                total_servicetime++; //전체 실행시간 1초 증가

                    for (int m = 1; m <= ProcessCount; m++) { //큐에 새로운 실행시간 안에 도착한 프로세스 삽입
                        StringTokenizer processToken = new StringTokenizer(process[m]);
                        String processId1 = processToken.nextToken();
                        int arriveTime1 = Integer.parseInt(processToken.nextToken());
                        int serviceTime1 = Integer.parseInt(processToken.nextToken());
                        processToken.nextToken();
                        int responseTime1 = Integer.parseInt(processToken.nextToken());
                        /*
                        현재 실행시간 안에 도착한 프로세스와 한번도 검사하지 않은 프로세스인 경우
                         */
                        if (total_servicetime >= arriveTime1 && check[m] != 0) {
                            q.add(process[m]); //큐에 해당 프로세스 정보 삽입
                            check[m] = 0; //한번 검사했다는 것을 표시
                        }
                    }

                    if (response_time[i] == 0) //응답시간을 한번만 계산
                        response_time[i] = (int) ((total_servicetime + ResponseTime) - ArriveTime);
                    tmp_arrivetime[i] = total_servicetime;
                    ganttchatt[c++] = ProcessId;
                    if (SaveServiceTime[i] == 0) { //프로세스가 주어진 작업을 다한 경우
                        return_time[i] = (int) (total_servicetime - ArriveTime); //반환시간 계산
                        break;
                    }
                }
                q.add(process[i]); //타임슬라이스만큼 작업하고 다시 큐에 삽입
            }
```

```
58          else if (SaveServiceTime[i] != 0 && SaveServiceTime[i] < TimeQuantum) { //프로세스 남은 작업시간이 0이 아니고 타임슬라이스보다 적은경우
59              for (int j = 1; j <= TimeQuantum; j++) {
60                  wait_time[i] += (total_servicetime - tmp_arrivetime[i]); //대기시간 저장
61                  SaveServiceTime[i] -= 1; //남은 실행시간 1초 감소
62                  total_servicetime++; //전체 실행시간 1초 증가
63
64                  for (int m = 1; m <= ProcessCount; m++) { //큐에 새로운 실행시간 안에 도착한 프로세스 삽입
65                      StringTokenizer processToken = new StringTokenizer(process[m]);
66                      String processId1 = processToken.nextToken();
67                      int arriveTime1 = Integer.parseInt(processToken.nextToken());
68                      int serviceTime1 = Integer.parseInt(processToken.nextToken());
69                      processToken.nextToken();
70                      int responseTime1 = Integer.parseInt(processToken.nextToken());
71                      /*
72                      현재 실행시간 안에 도착한 프로세스와 한번도 검사하지 않은 프로세스인 경우
73                       */
74                      if (total_servicetime >= arriveTime1 && check[m] != 0) {
75                          q.add(process[m]); //큐에 해당 프로세스 정보 삽입
76                          check[m] = 0; //한번 검사했다는 것을 표시
77                      }
78                  }
79
80                  if (response_time[i] == 0)
81                      response_time[i] = (int) ((total_servicetime + ResponseTime) - ArriveTime);
82                  tmp_arrivetime[i] = total_servicetime;
83                  ganttchatt[c++] = ProcessId;
84                  if (SaveServiceTime[i] == 0) {
85                      return_time[i] = (int) (total_servicetime - ArriveTime);
86                      break;
87                  }
88                              }
89                              q.add(process[i]);
90                          }
91                      }
92                  }
93              }
94              }
95              System.out.println("선점형 - 라운드로빈(RR)스케줄링");
96              Preemptive_Print_Process print_process = new Preemptive_Print_Process();
97              print_process.print(ProcessCount, wait_time, tmp_processId, return_time, ganttchatt, response_time);
98          }
99      }
```

<1-6. SRT스케줄링 코드>

```
import java.util.*;
public class SRT_Scheduling extends ProcessSort{
    public void run(){
        process= FileOpen();
        int c=0,gc=0,total_servicetime=0,tmp_index=1,max=0;
        ArriveTimeSort(); //도착시간대로 정렬
        TimeSort(); //정렬된 도착시간에 맞게 프로세스들의 각종 정보들 정렬
        String[] ganttchatt=new String[ServiceTimeSum];

        StringTokenizer processrealToken=new StringTokenizer(SrtProcess[1]); //도착시간이 0인 프로세스 정보
            ProcessId = processrealToken.nextToken();
            ArriveTime = Integer.parseInt(processrealToken.nextToken());
            ServiceTime = Integer.parseInt(processrealToken.nextToken());
            processrealToken.nextToken();processrealToken.nextToken();
            if (SaveServiceTime[1] >= TimeQuantum) { //도착시간이 0인 프로세스의 실행시간이 타임슬라이스보다 큰 경우
                wait_time[1] += (total_servicetime - tmp_arrivetime[1]); //대기시간 저장
                SaveServiceTime[1] -= TimeQuantum;
                response_time[1]= (int) ((restime[1]+total_servicetime)- ArriveTime); //응답시간 계산
                total_servicetime += TimeQuantum;
                tmp_arrivetime[1] = total_servicetime;
                for(int m = 0; m< TimeQuantum; m++)
                    ganttchatt[gc++]= ProcessId;
                if (SaveServiceTime[1] == 0) { //프로세스가 주어진 작업을 다한 경우
                    return_time[1] = (int) (total_servicetime - ArriveTime); //반환시간 계산
                    process[tmp_index++]= SrtProcess[1]; //process배열에 작업을 다한 프로세스 정보 저장
                }
            }
    else if (SaveServiceTime[1] != 0 && SaveServiceTime[1] < TimeQuantum) { //도착시간이 0인 프로세스의 실행시간이 타임슬라이스보다 작은경우
        wait_time[1] += (total_servicetime - tmp_arrivetime[1]); //대기시간 저장
        response_time[1]= (int) ((restime[1]+total_servicetime)- ArriveTime); //반응시간 계산
        total_servicetime += SaveServiceTime[1];
        for(int m = 0; m< SaveServiceTime[1]; m++)
            ganttchatt[gc++]= ProcessId;
    }

while(total_servicetime!= ServiceTimeSum) { //현재까지의 실행시간이 전체 실행시간이 되기 전까지 반복
    int index=0;
    if (c!= ProcessCount) {
        int min=10000; //최소 실행시간을 매우 큰 수로 임의지정
        for(int i = 1; i<= ProcessCount; i++){
            /*
            최단 작업시간이면서 수행할 작업이 남아있고 현재까지의 실행시간안에 도착한 프로세스와 남은 작업시간이 같은 경우 먼저 도착한 프로세스 먼저 수행하는 조건
             */
                if(min>= SaveServiceTime[i]&& SaveServiceTime[i]!=0&&tmp_arrivetime[i]<=total_servicetime&&min!= SaveServiceTime[i]){
                        min = SaveServiceTime[i];
                        index = i; //조건에 맞는 프로세스의 인덱스를 index에 저장
                    }
                }
```

```
51          if (SaveServiceTime[index] >= TimeQuantum) { //프로세스의 실행시간이 타임슬라이스보다 큰 경우
52              wait_time[index] += (total_servicetime - tmp_arrivetime[index]); //대기시간 저장
53              SaveServiceTime[index] -= TimeQuantum;
54              if(response_time[index]==0) //응답시간을 한번만 계산
55                  response_time[index]=(restime[index]+total_servicetime)-tmp_arrivetime[index];
56              total_servicetime += TimeQuantum;
57              tmp_arrivetime[index] = total_servicetime;
58              for(int m = 0; m< TimeQuantum; m++)
59                  ganttchatt[gc++]=tmp_processId[index];
60              if (SaveServiceTime[index] == 0) { //주어진 작업을 프로세스가 다 수행한 경우
61                  StringTokenizer processToken=new StringTokenizer(SrtProcess[index]);
62                  processToken.nextToken();
63                  ArriveTime =Integer.parseInt(processToken.nextToken());
64                  processToken.nextToken();processToken.nextToken();
65                  return_time[index] = (int) (total_servicetime - ArriveTime); //반환시간 계산
66                  c++;
67                  continue;
68              }
69          }
70          else if (SaveServiceTime[index] != 0 && SaveServiceTime[index] < TimeQuantum) { //프로세스 남은 작업시간이 0이 아니고 타임슬라이스보다 적은경우
71              wait_time[index] += (total_servicetime - tmp_arrivetime[index]); //대기시간 저장
72              if(response_time[index]==0)
73                  response_time[index]=(restime[index]+total_servicetime)-tmp_arrivetime[index];
74              total_servicetime += SaveServiceTime[index];
75              for(int m = 0; m< SaveServiceTime[index]; m++)
76                  ganttchatt[gc++]=tmp_processId[index];
77          }
78          }
79      }
80          System.out.println("선점형 - SRT스케줄링");
81          Preemptive_Print_Process print_process=new Preemptive_Print_Process();
82          print_process.print(ProcessCount,wait_time,tmp_processId,return_time,ganttchatt,response_time);
83      }
84  }
```

<1-7. 선점 우선순위 스케줄링 코드>

```java
import java.util.*;
public class PreemptivePriority_Scheduling extends ProcessSort{
    public void run() {
        process= FileOpen();
        ArriveTimeSort(); //도착시간대로 정렬
        Vector<Integer>v=new Vector<>();
        Vector<Integer>sertime_v=new Vector<>();
        Vector<Integer>arrtime_v=new Vector<>();
        String[] cpu_process = new String[ProcessCount +1];
        int index = 0,c=1,gc=0;
        String[] GanttChatt=new String[ServiceTimeSum]; //간트차트 배열
/*
프로세스 정렬
 */
        for(int i = 1; i<= ProcessCount; i++){
            int count=0;
            for(int j = 1; j<= ProcessCount; j++){
                StringTokenizer processToken=new StringTokenizer(process[j]);
                    ProcessId = processToken.nextToken();
                    ArriveTime = Integer.parseInt(processToken.nextToken());
                    ServiceTime = Integer.parseInt(processToken.nextToken());
                    priority = Integer.parseInt(processToken.nextToken());
                    if(tmp_priority[i]==priority&&tmp_processId[j]!="0"){ //우선순위대로 정렬, 벡터에 동일한 프로세스가 있는지 검사
                        count++;
                        v.add(j);//인덱스를 담음
                        sertime_v.add((int) ServiceTime);
                        arrtime_v.add((int) ArriveTime);
                    }
            }
            if(count==1){ //동일한 우선순위 프로세스가 없을 때
                cpu_process[c] = process[v.get(0)]; //우선순위에 맞는 프로세스를 정렬된 cpu_process에 담음
                SaveServiceTime[c]=sertime_v.get(0); //해당 프로세스의 실행시간을 각각의 프로세스 실행시간을 저장하는 배열에 담음
                tmp_processId[v.get(0)]="0"; //한번 배열에 들어간 프로세스의 아이디를 "0"으로 바꿈
                v.clear();
                sertime_v.clear();
                arrtime_v.clear();
                c++; //cpu_process의 인덱스를 하나 증가시킴
            }
            else if(count>=1) { //동일한 우선순위를 가진 프로세스가 있는 경우
                while (!v.isEmpty()) {
                    int min = arrtime_v.get(0); //도착시간이 빠른 프로세스를 먼저 cpu_process배열에 담음
                    for (int m = 0; m < v.size(); m++) {
                        if (min >= arrtime_v.get(m)) {
                            min = arrtime_v.get(m);
                            index = m; //최소 도착시간을 가진 프로세스의 인덱스값을 index배열에 저장
                        }
                    }
                    cpu_process[c] = process[v.get(index)]; //우선순위에 맞는 프로세스를 정렬된 cpu_process에 담음
                    SaveServiceTime[c] = sertime_v.get(index); //해당 프로세스의 실행시간을 각각의 프로세스 실행시간을 저장하는 배열에 담음
                    tmp_processId[v.get(index)] = "0"; //한번 배열에 들어간 프로세스의 아이디를 "0"으로 바꿈
                    v.remove(index);
                    sertime_v.remove(index);
                    arrtime_v.remove(index);
                    c++; //cpu_process의 인덱스를 하나 증가시킴
                }
            }
        }
```

```
59          for(int i = 1; i<= ProcessCount; i++){ //정렬된 프로세스들의 정보를 각각의 정보에 맞는 배열에 저장
60              StringTokenizer CpuprocessToken=new StringTokenizer(cpu_process[i]);
61              tmp_processId[i] = CpuprocessToken.nextToken(); //프로세스 ID
62              tmp_arrivetime[i] = Integer.parseInt(CpuprocessToken.nextToken()); //도착시간
63              tmp_servicetime[i] = Integer.parseInt(CpuprocessToken.nextToken()); //실행시간
64              tmp_priority[i] = Integer.parseInt(CpuprocessToken.nextToken()); //우선순위
65              restime[i]=Integer.parseInt(CpuprocessToken.nextToken()); //응답시간
66          }
67
68          int total_servicetime=0;
69          while(total_servicetime!= ServiceTimeSum){ //1초 단위로 현재 실행시간이 전체 프로세스들의 실행시간이 되기 전까지 반복
70              for(int i = 1; i<= ProcessCount; i++){
71                  StringTokenizer CpuprocessToken=new StringTokenizer(cpu_process[i]);
72                  ProcessId =CpuprocessToken.nextToken();
73                  ArriveTime = Integer.parseInt(CpuprocessToken.nextToken());
74                  ServiceTime = Integer.parseInt(CpuprocessToken.nextToken());
75                  priority = Integer.parseInt(CpuprocessToken.nextToken());
76                  if(total_servicetime>= ArriveTime && SaveServiceTime[i]!=0){ //프로세스의 실행시간이 남아있고, 현재작업시간안에 프로세스가 도착한 경우
77                      wait_time[i]+=(total_servicetime-tmp_arrivetime[i]); //대기시간 계산
78                      if(response_time[i]==0) //한번만 응답시간을 계산
79                          response_time[i]= (int) ((total_servicetime+restime[i])- ArriveTime);
80                      total_servicetime++; //현재까지의 작업시간을 1초 증가
81                      SaveServiceTime[i]--; //한번 작업한 프로세스의 실행시간을 1초 감소
82                      tmp_arrivetime[i]=total_servicetime; //해당 인덱스 프로세스 도착시간을 현재까지 실행한 작업시간으로 초기화
83                      GanttChatt[gc++]= ProcessId; //간트차트에 작업을 수행한 프로세스아이디 저장
84                      return_time[i]= (int) (total_servicetime- ArriveTime); //반환시간 계산
85                      break;
86                  }
87              }
88          }
89          System.out.println("선점형 - 선점형 우선순위 스케줄링");
90          Preemptive_Print_Process print_process=new Preemptive_Print_Process();
91          print_process.print(ProcessCount,wait_time,tmp_processId,return_time,GanttChatt,response_time);
92      }
93  }
```

<1-8. 정렬(processSort)메소드 코드>

```java
import java.util.Arrays;
import java.util.StringTokenizer;
public class ProcessSort extends Process_Variable{
    public  void ArriveTimeSort(){ //도착시간을 정렬해주는 메소드
        for (int i = 1; i <= ProcessCount; i++) {
            StringTokenizer processToken = new StringTokenizer(process[i]);
            ProcessId = processToken.nextToken(); //프로세스 ID
            ArriveTime = Integer.parseInt(processToken.nextToken()); //도착시간
            ServiceTime = Integer.parseInt(processToken.nextToken()); //실행시간
            tmp_priority[i] = Integer.parseInt(processToken.nextToken()); //우선순위
            processToken.nextToken(); //응답시간
            tmp_time[i] = (int) ArriveTime; //도착시간 모음배열
            ServiceTimeSum += ServiceTime;
            if (ArriveTime == 0){ //도착시간이 0인 프로세스를 검사
                q.add(process[i]); //도착시간이 0인 프로세스를 큐에 저장
                deque.addLast(process[i]); //도착시간이 0인 프로세스를 덱에 저장
            }
            tmp_servicetime[i]= (int) ServiceTime; //실행시간 저장
            tmp_arrivetime[i]= (int) ArriveTime; //도착시간 저장
            tmp_processId[i]= ProcessId; //프로세스 ID저장
        }
        Arrays.sort(tmp_time); //도착시간 정렬
        Arrays.sort(tmp_priority); //우선순위 정렬
        Arrays.sort(tmp_servicetime); //실행시간 정렬
    }
    public void TimeSort(){ //정렬된 도착시간배열에 맞춰 다른 프로세스 정보들도 정렬(srt, 비선점우선순위, rr사용)
        for(int i = 1; i<= ProcessCount; i++){
            for(int j = 1; j<= ProcessCount; j++){
                StringTokenizer processToken=new StringTokenizer(process[j]);
                ProcessId =processToken.nextToken();
                ArriveTime =Integer.parseInt(processToken.nextToken());
                ServiceTime =Integer.parseInt(processToken.nextToken());
                processToken.nextToken();//우선순위
                ResponseTime =Integer.parseInt(processToken.nextToken());
                if(tmp_time[i]== ArriveTime){ //정렬된 도착시간배열에 프로세스 정보를 정렬
                    SrtProcess[i]=process[j]; //SRT에서 사용하는 정렬된 프로세스 배열
                    NonpreemrtiveProcess[i] = process[j]; //비선점우선순위에서 사용하는 정렬된 프로세스 배열
                    process[i] = tmp_process[i]; //정렬된 프로세스 배열
                    SaveServiceTime[i]= (int) ServiceTime; //각각 프로세스의 남은 실행시간 저장
                    tmp_arrivetime[i]= (int) ArriveTime; //각 프로세스 별 정렬된 도착시간 저장
                    tmp_processId[i]= ProcessId; //프로세스 ID저장
                    tmp_servicetime[i] = (int) ServiceTime; //각 프로세스별 정렬된 실행시간 저장
                    check[i] = 1; //RR스케줄링에서 한번 검사한 프로세스인지 확인하는 배열
                    restime[i]= (int) ResponseTime; //각 프로세스별 정렬된 반응시간 저장
                }
            }
        }
    }
}
```

<1-9. 변수(ProcessVariable)메소드 코드>

```java
import java.io.*;
import java.util.*;
public class Process_Variable {
    static int num; //파일 라인수
    public String[] FileOpen(){
        File f = new File( pathname: "c:\\Temp\\OS.txt");
        FileReader fin = null;
        BufferedReader br = null;
        String[] tmp_process = new String[100]; //파일 전체의 내용을 저장하는 배열
        num=0;
        try {
            br = new BufferedReader(new FileReader(f));
            while (true) {
                String line = br.readLine();
                if (line == null)
                    break;
                tmp_process[num++] = line;
            }
        } catch (IOException e) {
            e.printStackTrace();
        }
        return tmp_process;
    }
    Queue<String> q = new LinkedList<>(); //프로세스 정보를 저장하는 Queue
    Deque<String> deque =new LinkedList<>(); //임시로 프로세스 정보를 저장하는 Deque
    String process[]= FileOpen();
    double ArriveTime, ServiceTime, ResponseTime;
    String ProcessId;
    int ProcessCount =Integer.parseInt(process[0]); //전체 프로세스 개수
    int priority=0, ServiceTimeSum =0; //우선순위, 실행시간 총합
    String[] tmp_processId = new String[ProcessCount +1]; //프로세스ID 임시 저장 배열
    String[] SrtProcess =new String[ProcessCount +1]; // srt스케줄링 정렬된 프로세스 배열
    String[] NonpreemrtiveProcess = new String[ProcessCount +1]; //비선점 우선순위 스케줄링 정렬된 프로세스 배열
    String[] tmp_process=process;
    int[] tmp_servicetime = new int[ProcessCount +1]; //실행시간 임시 저장 배열
    int[] tmp_arrivetime = new int[ProcessCount +1]; //도착시간 임시 저장 배열
    int[] tmp_priority = new int[ProcessCount +1]; //우선순위 임시 저장 배열
    int[] tmp_time = new int[ProcessCount +1]; //시간 임시 저장 배열
    int[] wait_time=new int[ProcessCount +1]; //프로세스 대기시간 배열
    int[] return_time=new int[ProcessCount +1]; //프로세스 반환시간 배열
    int[] SaveServiceTime =new int[ProcessCount +1]; //각 프로세스 별 남은 실행시간 저장 배열
    int[] response_time=new int[ProcessCount +1]; //각 프로세스 응답시간값 저장 배열
    int[] restime=new int[ProcessCount +1]; //각 프로세스 응답시간
    int TimeQuantum =Integer.parseInt(process[num-1]); //타임슬라이스
    int[] check=new int[ProcessCount +1]; //검사한 프로세스인지 확인하는 배열
}
```

<1-10. 비선점 출력 메소드 코드>

```java
import java.util.*;
public class Nonpreemptive_Print_Process extends Process_Variable{
    public void print(Deque<String> q, int process_count){
        int TotalWaitTime = 0,retime; //프로세스 대기시간 총합, 응답시간
        int TotalResponseTime=0,TotalServiceTime=0,j=0; //반응시간, 실행시간 총합
        double TotalReturnTime = 0; //반환시간 총합
        int[] time = new int[process_count]; //프로세스 별 반환시간 저장배열
        System.out.println("<간트차트>");

        Iterator t=q.iterator();
        while(t.hasNext()){
            String str= (String) t.next();
            StringTokenizer strToken=new StringTokenizer(str);
            ProcessId =strToken.nextToken();
            strToken.nextToken();
            ServiceTime =Integer.parseInt(strToken.nextToken());
            strToken.nextToken();
            for(int i = 0; i< ServiceTime; i++)
                System.out.print("["+ ProcessId +"]");
        }
        System.out.println();
      Iterator it = q.iterator();
      while (it.hasNext()) {
          String str = (String) it.next();
          StringTokenizer strToken = new StringTokenizer(str);
              ProcessId =strToken.nextToken();
              ArriveTime = Integer.parseInt(strToken.nextToken()); //도착시간
              ServiceTime = Integer.parseInt(strToken.nextToken()); //실행시간
              System.out.print(ProcessId + "의 대기시간: "); //프로세스 ID
              System.out.println(ServiceTimeSum - ArriveTime); //프로세스별 대기시간 출력
              TotalWaitTime += (ServiceTimeSum - ArriveTime); // 대기시간 총합
              ServiceTimeSum += ServiceTime;
              time[j] = (int) (ServiceTimeSum - ArriveTime); //반환시간 저장
              j++;
              strToken.nextToken();strToken.nextToken();
      }
      System.out.println("평균 대기 시간(AWT): " + (double)TotalWaitTime / process_count);
    it=q.iterator();
    while(it.hasNext()){
        String str= (String) it.next();
        StringTokenizer strToken=new StringTokenizer(str);
        ProcessId = strToken.nextToken();
        ArriveTime =Integer.parseInt(strToken.nextToken());
        ServiceTime =Integer.parseInt(strToken.nextToken());
        strToken.nextToken();
        retime=Integer.parseInt(strToken.nextToken());
        ResponseTime =(TotalServiceTime+retime) - ArriveTime; //응답시간 계산
        System.out.println(ProcessId + "의 응답시간: " + ResponseTime); //프로세스 ID
        TotalServiceTime+= ServiceTime;
        TotalResponseTime+= ResponseTime;
    }

    System.out.println("평균응답시간(ART): "+(double)TotalResponseTime/process_count);

    j = 0;
    it = q.iterator();
    while (it.hasNext()) {
        String str = (String) it.next();
        StringTokenizer strToken = new StringTokenizer(str);
            System.out.println(strToken.nextToken() + "의 반환시간: " + time[j]); //프로세스 ID
            strToken.nextToken(); strToken.nextToken(); strToken.nextToken(); strToken.nextToken();
        TotalReturnTime += time[j++];
    }
    System.out.println("평균 반환 시간(ATT): "+TotalReturnTime /process_count);
  }
}
```

<1-11. 선점 출력 메소드 코드>

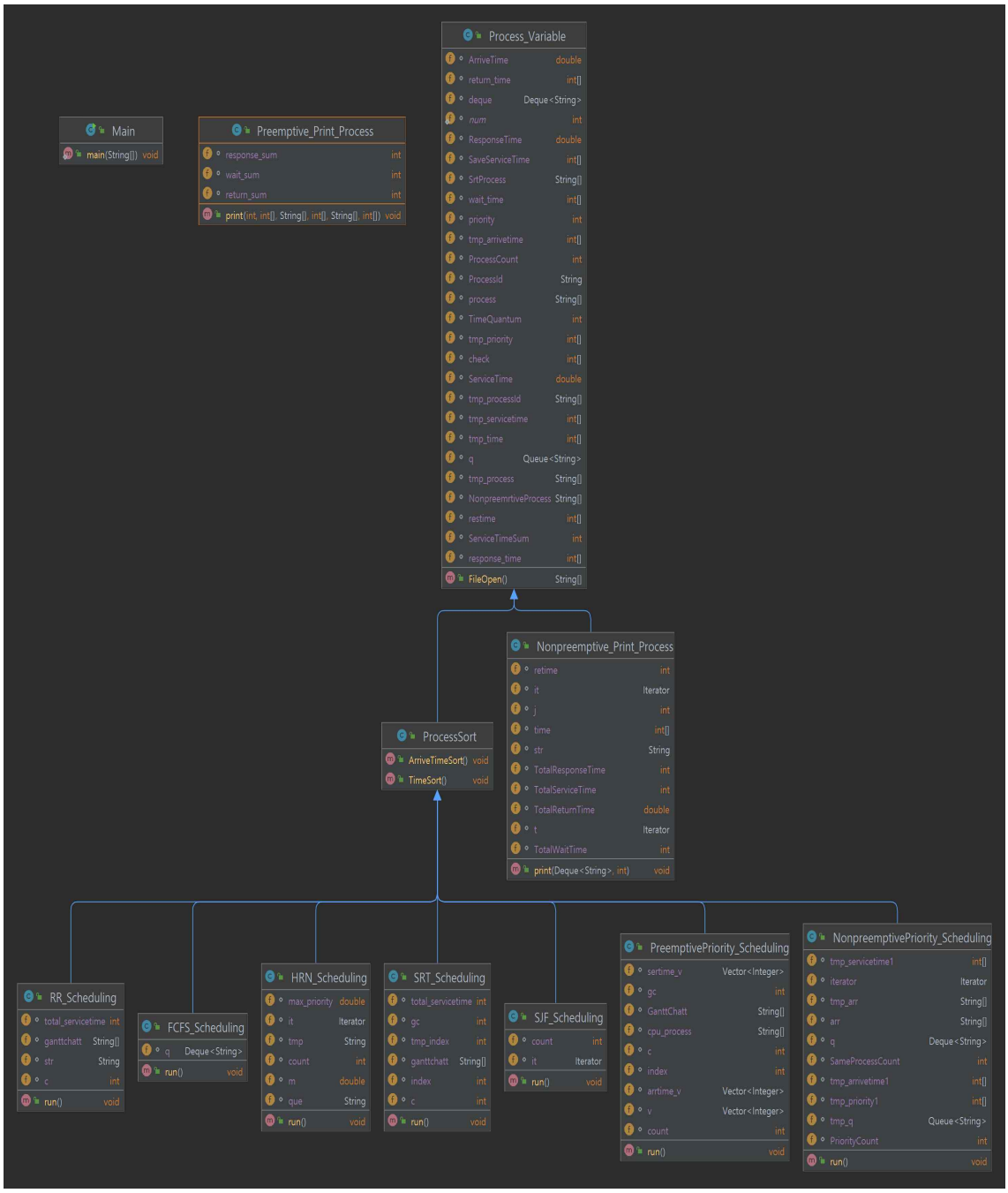
```java
public class Preemptive_Print_Process {
    public void print(int process_count,int wait_time[],String tmp_processId[],int return_time[],String gant[],int responsetime[]){
        int wait_sum=0;
        int return_sum=0;
        int response_sum=0;
        System.out.println("<간트차트>");
        for(int i=0;i< gant.length;i++){
            System.out.print("["+gant[i]+"]");
        }

        System.out.println();
        for(int i=1;i<=process_count;i++){
            System.out.println(tmp_processId[i]+"의 대기시간: "+wait_time[i]);
            wait_sum+=wait_time[i];
        }
        System.out.println("평균 대기 시간(AWT): "+(double)wait_sum/process_count);

        for(int i=1;i<=process_count;i++){
            System.out.println(tmp_processId[i]+"의 응답시간: "+responsetime[i]);
            response_sum+=responsetime[i];
        }
        System.out.println("평균 응답 시간(ART): "+(double)response_sum/process_count);

        for(int i=1;i<=process_count;i++){
            System.out.println(tmp_processId[i]+"의 반환시간: "+return_time[i]);
            return_sum+=return_time[i];
        }
        System.out.println("평균 반환 시간(ATT): "+(double)return_sum/process_count);
    }
}
```

<2. 클래스 다이어그램 & 클래스 구성도>



<3. 주요 변수>

```
double ArriveTime, ServiceTime, ResponseTime;
String ProcessId;
```

1. 프로세스의 도착시간, 실행시간, 응답시간, 프로세스 ID를 저장해주는 변수이다.

```
int ProcessCount =Integer.parseInt(process[0]); //전체 프로세스 개수
```

2. 파일에 존재하는 전체 프로세스의 개수를 저장해주는 변수이고, ProcessCount만큼 반복을 진행한다.

```
int[] wait_time=new int[ProcessCount +1]; //프로세스 대기시간 배열
int[] return_time=new int[ProcessCount +1]; //프로세스 반환시간 배열
int[] response_time=new int[ProcessCount +1]; //각 프로세스 응답시간값 저장 배열
```

3. 각 프로세스들의 대기시간, 반환시간, 응답시간을 저장해주는 배열이다.

```
int[] check=new int[ProcessCount +1]; //검사한 프로세스인지 확인하는 배열
```

4. RR스케줄링에서 한번 검사한 프로세스인지 판별해주는 배열으로써 배열에 동일한 프로세스가 저장되는 것을 막아준다.

```
int TimeQuantum =Integer.parseInt(process[num-1]); //타임슬라이스
```

5. 타임슬라이스를 저장해주는 변수이다.

```
String process[]= FileOpen();
```

6. txt파일에서 읽어온 데이터들을 저장해주는 중요한 배열이다.

```
Deque<String> q = new LinkedList<>();
```

7. 비선점형 스케줄링에서 정렬된 프로세스들을 저장해주는 덱 자료구조다.

```
Queue<String> q = new LinkedList<>();
```

8. 선점형 스케줄링에서 프로세스들을 정렬하면서 작업을 수행하도록 도와주는 큐 자료구조이다.

```
String[] ganttchatt = new String[ServiceTimeSum];
```

9. 스케줄링의 간트차트를 저장해주는 배열이다.

<3. 사용한 자료구조>

저는 일단 총 4가지의 자료구조를 사용하였습니다.

우선 비선점형 스케줄링(FCFS, SJF, HRN,비선점 우선순위)에서 Deque(덱)자료구조를 사용하였고, 추가적으로 비선점 우선순위에서는 Deque(덱)과 Queue(큐) 자료구조를 이용하였습니다.

```
Deque<String> q = new LinkedList<>();
```

1) FCFS스케줄링 -> Deque(덱) 자료구조사용

```
Iterator it=deque.iterator();
```

2) SJF스케줄링 -> Deque(덱) 자료구조사용

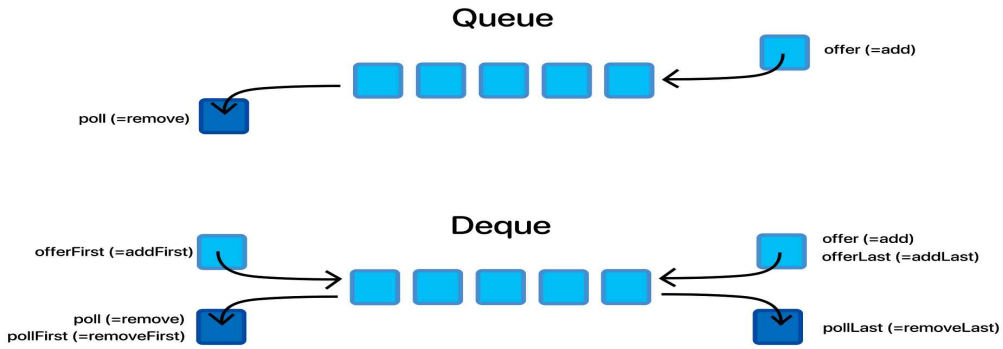
```
Iterator it=deque.iterator();
```

3) HRN스케줄링 -> Deque(덱) 자료구조사용

```
Deque<String> q = new LinkedList<>();
Queue<String> tmp_q = new LinkedList<>();
```

4) 비선점 우선순위 스케줄링 -> Deque(덱) & Queue(큐) 자료구조사용

저는 비선점스케줄링을 구현할 때 덱과 큐의 자료구조를 이용하여 구현을 하였는데 덱과 큐의 차이는 큐는 무조건 맨 뒤에 삽입이 되지만 덱은 처음과 끝 둘 다 개발자가 지정한대로 삽입을 할 수 있다는 장점이 있습니다.



※ Deque(덱) vs Queue(큐)

위의 사진처럼 덱과 큐는 비슷하지만 삽입하고 poll할때도 큐는 FIFO의 구조이지만 덱은 FIFO+FILO의 구조를 합한 것과 같은 자료구조입니다.
FCFS스케줄링에서의 Deque(덱)은 도착한 시간의 프로세스 순서대로 덱에 삽입을 한 다음 FIFO구조를 이용하여 순차적으로 출력하기 위하여 사용하였습니다.

SJF스케줄링과 HRN스케줄링에서의 Deque(덱)은 처음에 도착시간이 0인 프로세스들을 먼저 Deque(덱)에 삽입을 한 다음 Iterator연산자를 이용하여 덱에 들어있는 정보들을 꺼내어 덱에 동일한 프로세스가 있는지 확인하여서 동일한 프로세스가 없다면 정렬 기준에 따라 덱에 새로운 프로세스를 삽입하는 용도로 사용하였습니다.

```
Iterator it=deque.iterator();
while(it.hasNext()){
    String s= (String) it.next();
    StringTokenizer QToken=new StringTokenizer(s);
    while(QToken.hasMoreTokens()){
        String Id=QToken.nextToken();
        if(Id.equals(ProcessId)) //덱에 동일한 프로세스가 있는지 확인
            count++;
    }
}
```

※ SJF와 HRN스케줄링에서 덱을 이용한 알고리즘 구현

비선점 우선순위 스케줄링에서는 덱과 큐의 자료구조를 이용하였다
비선점 우선순위에서의 덱은 비선점 우선순위의 정렬기준에 맞춰 프로세스를 저장하는 핵심 자료구조이고, 큐는 덱에 동일한 프로세스가 있는지 검사할 때 임시적으로 프로세스의 정보를 담아두는 용도의 자료구조로 이용하였다.

```
Iterator iterator=d.iterator(); //덱에 동일 프로세스가 있는지 확인
while(iterator.hasNext()){
    StringTokenizer QToken=new StringTokenizer((String) iterator.next());
    String proId=QToken.nextToken();
    QToken.nextToken();QToken.nextToken();QToken.nextToken();QToken.nextToken();
    if(ProcessId.equals(proId)) //동일한 프로세스가 있다면 SameProcessCount증가
        SameProcessCount++;
}
```

※ 비선점 우선순위 스케줄링에서 덱을 이용한 알고리즘 구현

```
if(PriorityCount==1){ //PriorityCount가 1인 경우는 동일한 우선순위가 없는 것을 의미
    while(!tmp_q.isEmpty()){
        d.add(tmp_q.poll()); //덱에 삽입
    }
}
```

※ 비선점 우선순위 스케줄링에서 덱에 동일한 프로세스가 없는 경우 삽입

```
    if(tmp_priority1[i]==priority&& SameProcessCount==0){ //덱에 동일한 프로세스가 없는 경우
        PriorityCount++; //동일한 우선순위가 있는지 확인
        tmp_q.add(tmp_arr[j]); //임시저장소인 큐에 저장
    }
```

※ 비선점 우선순위 스케줄링에서 덱에 동일한 프로세스가 있는지 확인할 때 임시저장장소로 큐를 이용

선점형 스케줄링(RR, SRT, 선점 우선순위)에서는 큐, 배열, 벡터의 자료구조를 이용하였습니다.

```
String str = q.poll(); //큐에 들어있는 프로세스를 str에 저장
```

1) RR스케줄링 -> Queue(큐) 자료구조 사용

```
wait_time[1] += (total_servicetime - tmp_arrivetime[1]); //대기시간 저장
SaveServiceTime[1] -= TimeQuantum;
response_time[1]= (int) ((restime[1]+total_servicetime)- ArriveTime); //응답시간 계산
total_servicetime += TimeQuantum;
tmp_arrivetime[1] = total_servicetime;
for(int m = 0; m< TimeQuantum; m++)
    ganttchatt[gc++]= ProcessId;
if (SaveServiceTime[1] == 0) { //프로세스가 주어진 작업을 다한 경우
    return_time[1] = (int) (total_servicetime - ArriveTime); //반환시간 계산
}
```

2) SRT스케줄링 -> 배열 자료구조 사용

```
Vector<Integer>v=new Vector<>();
Vector<Integer>sertime_v=new Vector<>();
Vector<Integer>arrtime_v=new Vector<>();
String[] cpu_process = new String[ProcessCount +1];
```

3) 선점 우선순위 스케줄링 -> 배열 & Vector(벡터) 자료구조 사용

RR(라운드로빈)스케줄링에서는 Queue(큐)자료구조를 이용하였는데 큐에는 우선 도착시간이 0인 프로세스가 들어가 있고, 큐에 있는 프로세스들을 앞에서부터 꺼내서 동일한 프로세스 정보가 있는지 검사하고

```
String str = q.poll(); //큐에 들어있는 프로세스를 str에 저장
StringTokenizer strToken = new StringTokenizer(str);
ProcessId = strToken.nextToken();
ArriveTime = Integer.parseInt(strToken.nextToken());
ServiceTime = Integer.parseInt(strToken.nextToken());
strToken.nextToken();
ResponseTime = Integer.parseInt(strToken.nextToken());

for (int i = 1; i <= ProcessCount; i++) {
    if (tmp_processId[i].equals(ProcessId)) { //정렬된 프로세스ID와 큐에서 꺼낸 프로세스ID가 동일한 경우
```

※ RR스케줄링에서 큐에 동일한 프로세스가 있는지 확인

만약 동일한 프로세스가 없는 경우 해당 프로세스의 작업을 수행하고 아래의 사진처럼 현재까지의 실행시간안에 도착한 프로세스가 있는지 확인하고 큐에 새로운 프로세스를 삽입할 때 큐를 이용하였다.

```
if (total_servicetime >= arriveTime1 && check[m] != 0) {
    q.add(process[m]); //큐에 해당 프로세스 정보 삽입
    check[m] = 0; //한번 검사했다는 것을 표시
}
```

※ 현재까지의 실행시간안에 도착한 프로세스가 있는 경우 큐에 삽입

SRT스케줄링에서는 배열 자료구조를 이용하였는데 process정보들이 담겨있

는 process배열을 이용하는 것이 아니라 프로세스의 정보들을 담는

```
int[] wait_time=new int[ProcessCount +1]; //프로세스 대기시간 배열
int[] return_time=new int[ProcessCount +1]; //프로세스 반환시간 배열
int[] response_time=new int[ProcessCount +1]; //각 프로세스 응답시간값 저장 배열
```

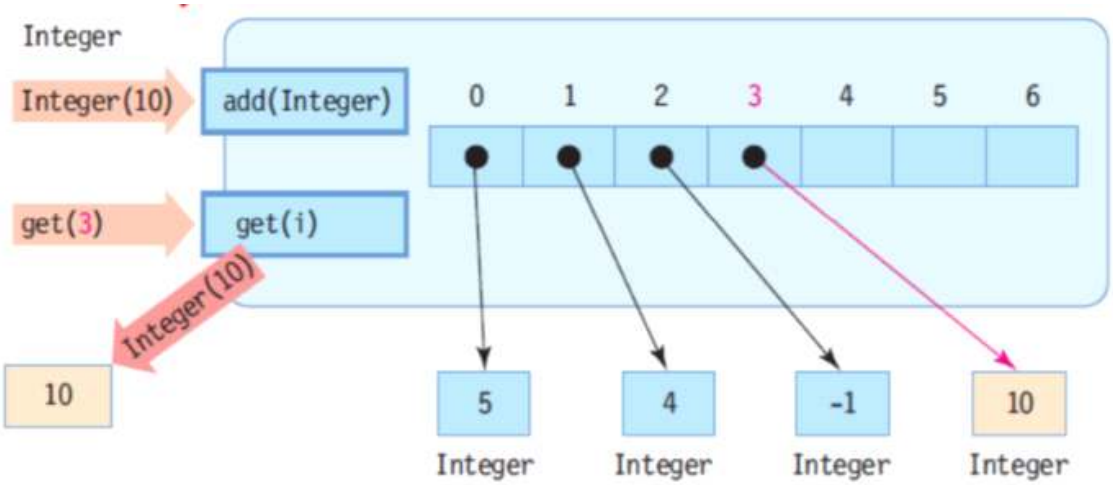
※ SRT스케줄링에서 사용하는 프로세스의 정보를 담는 배열

배열들에 1초마다 프로세스들의 작업에 따라 변하는 내용을 저장하는 용도로 배열 자료구조를 이용하였다.

선점 우선순위 스케줄링에서는 배열과 벡터 자료구조를 이용하였다.
배열은 SRT스케줄링에서처럼 각각의 배열을 이용하여 프로세스의 정보들을 저장하고 있고 Vector(벡터)는 프로세스들 간의 동일 우선순위가 있는지 파악할 때 프로세스들을 가리키는 인덱스값, 실행시간, 도착시간을 담는 용도로 Vector(벡터)자료구조를 이용하였다.

```
if(tmp_priority[i]==priority&&tmp_processId[j]!="0"){ //우선순위대로 정렬, 벡터에 동일한 프로세스가 있는지 검사
    count++;
    v.add(j);//인덱스를 담음
    sertime_v.add((int) ServiceTime);
    arrtime_v.add((int) ArriveTime);
}
```

※ 선점 우선순위 스케줄링에서 프로세스 정보를 담기 위해 사용한 Vector(벡터) 자료구조



※ Vector(벡터) 내부 구조

<4. 주요 알고리즘>

※ArriveTimeSort() 메소드

```
4       public  void ArriveTimeSort(){ //도착시간을 정렬해주는 메소드
5           for (int i = 1; i <= ProcessCount; i++) {
6               StringTokenizer processToken = new StringTokenizer(process[i]);
7               ProcessId = processToken.nextToken(); //프로세스 ID
8               ArriveTime = Integer.parseInt(processToken.nextToken()); //도착시간
9               ServiceTime = Integer.parseInt(processToken.nextToken()); //실행시간
10              tmp_priority[i] = Integer.parseInt(processToken.nextToken()); //우선순위
11              processToken.nextToken(); //응답시간
12              tmp_time[i] = (int) ArriveTime; //도착시간 모음배열
13              ServiceTimeSum += ServiceTime;
14              if (ArriveTime == 0){ //도착시간이 0인 프로세스를 검사
15                  q.add(process[i]); //도착시간이 0인 프로세스를 큐에 저장
16                  deque.addLast(process[i]); //도착시간이 0인 프로세스를 덱에 저장
17              }
18              tmp_servicetime[i]= (int) ServiceTime; //실행시간 저장
19              tmp_arrivetime[i]= (int) ArriveTime; //도착시간 저장
20              tmp_processId[i]= ProcessId; //프로세스 ID저장
21          }
22          Arrays.sort(tmp_time); //도착시간 정렬
23          Arrays.sort(tmp_priority); //우선순위 정렬
24          Arrays.sort(tmp_servicetime); //실행시간 정렬
25      }
```

※ FCFS스케줄링이 상속하는 ArriveTimeSort()메소드

위 메소드는 process의 정보들을 도착시간 순서대로 정렬을 해주는 저의 모든 스케줄링에서 공통적으로 사용되는 핵심 알고리즘이 담겨있는 기본적인 메소드이다.

4-1) FCFS스케줄링

```
11          for (int i = 1; i <= ProcessCount; i++) {
12              for (int k = 1; k <= ProcessCount; k++) {
13                  StringTokenizer ProcessToken = new StringTokenizer(process[k]);
14                  ProcessId = ProcessToken.nextToken();
15                  ArriveTime = Integer.parseInt(ProcessToken.nextToken());
16                  ServiceTime = Integer.parseInt(ProcessToken.nextToken());
17                  ProcessToken.nextToken();
18                  ProcessToken.nextToken();
19                  if (tmp_time[i] == ArriveTime) { //도착시간 순으로 q에 삽입
20                      q.add(process[k]);
21                  }
22              }
23          }
```

※ FCFS스케줄링 핵심 알고리즘

위 코드가 FCFS의 핵심 알고리즘인데 정렬된 tmp_time배열에 맞춰서 도착시간 순서대로 Deque(덱)에 삽입을 하여서 출력하는 알고리즘이다.

```
print_process.print(q, ProcessCount);
```

덱에 순차적으로 들어가있는 프로세스정보를 비선점형 스케줄링 출력 메소드에 파라미터로 전달해 주어서 간트차트, 각 프로세스별 대기시간, 평균대기시간, 각 프로세스별 응답시간, 평균응답시간, 각 프로세스별 반환시간, 평균반환시간을 출력해준다.

4-2) SJF스케줄링

```
7         for(int i = 1; i<= ProcessCount; i++){
8             for(int k = 1; k<= ProcessCount; k++){
9                 int count=0;
10                StringTokenizer processToken=new StringTokenizer(process[k]);
11                while(processToken.hasMoreTokens()){
12                    ProcessId =processToken.nextToken();
13                    ArriveTime =Integer.parseInt(processToken.nextToken());
14                    ServiceTime =Integer.parseInt(processToken.nextToken());
15                    processToken.nextToken();processToken.nextToken();
```

이중 반복문을 이용하여 작은 for문에서 프로세스정보가 들어있는 배열을 순차적으로 검사하여서 분리하는 작업을 하는 알고리즘이다.
이 부분에서 각 프로세스의 ID, 도착시간, 실행시간, 우선순위, 응답시간을 확인할 수 있다.

```
19                Iterator it=deque.iterator();
20                while(it.hasNext()){
21                    String s= (String) it.next();
22                    StringTokenizer QToken=new StringTokenizer(s);
23                    while(QToken.hasMoreTokens()){
24                        String Id=QToken.nextToken();
25                        if(Id.equals(ProcessId)) //덱에 동일한 프로세스가 있는지 확인
26                            count++;
27                    }
28                }
```

Iterator연산자를 이용하여 덱에 들어있는 프로세스 ID를 확인하여 10번라인에서 StringTokenizer를 이용하여 분리한 프로세스 ID와 비교하여 동일한 프로세스가 덱에 존재한다면 count변수를 1증가 시켜주는 알고리즘이다.
만약 동일한 프로세스ID가 존재하지 않는다면 count변수의 값은 초기값인 0이다.

```
30            if(count==0) { //동일한 프로세스가 없는 경우
31                if (ServiceTime == tmp_servicetime[i]) { //정렬된 tmp_servicetime에 의해 가장 짧은 실행시간 프로세스를 덱에 삽입
32                    deque.add(process[k]);
33                }
34            }
```

20번라인의 while문을 수행하고 count값이 0이면 덱에 동일한 프로세스가 없다는 의미이며 SJF는 최단 작업시간을 우선적으로 하기 때문에 31번라인의 조건문 즉, 작업시간이 짧은 순대로 정렬되어 있는 tmp_servicetime[i] 배열에 맞춰서 덱에 프로세스를 삽입한다.

```
print_process.print(q, ProcessCount);
```

덱에 순차적으로 들어가있는 프로세스정보를 비선점형 스케줄링 출력 메소드에 파라미터로 전달해 주어서 간트차트, 각 프로세스별 대기시간, 평균대기시간, 각 프로세스별 응답시간, 평균응답시간, 각 프로세스별 반환시간, 평균반환시간을 출력해준다.

4-3) HRN스케줄링

```
10          for(int i = 0; i< ProcessCount; i++){
11              String que= deque.peekLast(); //덱의 가장 나중에 들어온 프로세스 정보
12              StringTokenizer DequeToken=new StringTokenizer(que);
13              double max_priority=0;
14                  DequeToken.nextToken();DequeToken.nextToken();
15                  ServiceTime =Double.parseDouble(DequeToken.nextToken());
16                  ServiceTimeSum += ServiceTime; //실행시간 총합
17                  DequeToken.nextToken();DequeToken.nextToken();
```

16번라인에서 현재 도착시간이 0인 프로세스가 덱에 존재하고 덱에 마지막에 있는 프로세스를 꺼내서 ServiceTimeSum변수에 덱에 있는 프로세스 실행시간을 더해준다. 이는 나중에 HRN스케줄링에서 우선순위를 구하는 **<(대기시간+CPU사용률)/CPU사용률>을 이용하기 위하여 선언하였다.**

```
19          for(int k = 1; k<= ProcessCount; k++){
20              int count=0;
21              StringTokenizer processToken=new StringTokenizer(process[k]);
22                  ProcessId =processToken.nextToken();
23                  ArriveTime =Double.parseDouble(processToken.nextToken());
24                  ServiceTime =Double.parseDouble(processToken.nextToken());
25                  /*
26                  이미 덱 안에 동일한 프로세스가 들어가 있는지 확인
27                   */
28                  Iterator it= deque.iterator();
29                  while(it.hasNext()){
30                      String s= (String) it.next();
31                      StringTokenizer QueueToken=new StringTokenizer(s);
32                      while(QueueToken.hasMoreTokens()){
33                          String Id=QueueToken.nextToken();
34                          if(Id.equals(ProcessId)) //덱 안에 동일한 프로세스가 있다면 count증가
35                              count++;
36                      }
37                  }
```

작은 반복문을 수행하면서 process배열에 들어있는 프로세스 정보를 꺼내서 각각의 정보를 StringTokenizer로 분리한 다음 28번 라인의 Iterator연산자를 이용하여 덱에 있는 프로세스들의 프로세스 ID와 34번 라인을 통해 동일한 프로세스 ID가 있는지 검사하여서 동일한 프로세스가 존재한다면 count변수를 1증가시켜준다.

```
38              if(count==0) { //덱에 동일한 프로세스가 없는 경우, 실제 우선순위를 구하는 작업을 하는 조건문
39                  double m=(ServiceTime +(ServiceTimeSum- ArriveTime))/ ServiceTime; //(대기시간+CPU사용률)/CPU사용률
40                  if(max_priority<m){
41                      max_priority=Math.max(max_priority,m);
42                      tmp=process[k]; //우선순위가 가장 빠른 프로세스를 tmp에 저장
43                  }
44              }
45          }
46          deque.add(tmp); //우선순위가 가장 높은 프로세스를 덱에 삽입
47      }
```

만약 동일한 프로세스가 없다면 count는 0이니까 38번라인의 조건문에 들어와서 m은 현재 실행시간 상태에서 HRN스케줄링의 우선순위 공식을 이용하여 가장 우선순위가 빠른 프로세스 정보를 String타입의 tmp변수에 초기화를 해주고 19번라인의 반복문을 다 반복해서 최종적으로 가장 높은 우선순

위를 가진 프로세스가 tmp에 저장되어 있는 상태니까 tmp에 저장되어있는 프로세스 정보를 덱에 삽입하고, 10번라인의 큰 반복문으로 돌아가서 위에 설명한 반복을 I가 ProcessCount보다 작을 때 까지 반복한다.

```
print_process.print(deque, ProcessCount);
```

덱에 순차적으로 들어가있는 프로세스정보를 비선점형 스케줄링 출력 메소드에 파라미터로 전달해 주어서 간트차트, 각 프로세스별 대기시간, 평균대기시간, 각 프로세스별 응답시간, 평균응답시간, 각 프로세스별 반환시간, 평균반환시간을 출력해준다.

4-4) 비선점 우선순위 스케줄링

```
16          for (int i = 0; i < ProcessCount -1; i++) {
17              tmp_arr[i] = NonpreemrtiveProcess[i+2]; //P2 P3 P4 P5 프로세스 정렬
18          }
```

NonpreemrtiveProcess배열에는 도착시간순서대로 정렬이 되어있는 상태고, 도착시간이 0인 프로세스는 1번인덱스에 저장되어있고 나머지 프로세스들을 우선순위를 기준으로 정렬하기 위하여 tmp_arr배열에 담아둔다.

```
20          for (int i = 0; i < ProcessCount -1; i++) {
21              StringTokenizer processToken = new StringTokenizer(tmp_arr[i]);
22                  ProcessId = processToken.nextToken(); //프로세스 ID
23                  tmp_arrivetime1[i] = Integer.parseInt(processToken.nextToken()); //도착시간
24                  tmp_servicetime1[i] = Integer.parseInt(processToken.nextToken()); //실행시간
25                  tmp_priority1[i] = Integer.parseInt(processToken.nextToken()); //우선순위
26                  processToken.nextToken();
27          }
28          Arrays.sort(tmp_priority1); //1 2 2 4(P4 P2 P5 P3순으로 정렬)
```

tmp_arr배열에 있는 프로세스정보를 StringTokenizer를 이용하여 분리한다음 도착시간, 실행시간, 우선순위를 담는 배열에 저장한다음 28번라인처럼 우선순위대로 정렬해준다.

```
30          for(int i = 0; i< ProcessCount -1; i++){
31              int PriorityCount=0;
32              for(int j = 0; j< ProcessCount -1; j++){
33                  StringTokenizer processToken=new StringTokenizer(tmp_arr[j]);
34                  int SameProcessCount=0;
35                  ProcessId =processToken.nextToken();
36                  ArriveTime =Integer.parseInt(processToken.nextToken());
37                  ServiceTime =Integer.parseInt(processToken.nextToken());
38                  priority=Integer.parseInt(processToken.nextToken());
39                  processToken.nextToken();
40                  Iterator iterator=d.iterator(); //덱에 동일 프로세스가 있는지 확인
41                  while(iterator.hasNext()){
42                      StringTokenizer QToken=new StringTokenizer((String) iterator.next());
43                      String proId=QToken.nextToken();
44                      QToken.nextToken();QToken.nextToken();QToken.nextToken();QToken.nextToken();
45                      if(ProcessId.equals(proId)) //동일한 프로세스가 있다면 SameProcessCount증가
46                          SameProcessCount++;
47                  }
```

tmp_arr배열에 들어있는 정보를 프로세스ID, 도착시간, 실행시간, 우선순위대로 분리를 한 후, Iterator을 이용하여 덱에 동일한 프로세스가 있는지 45번라인을 통해 검사하여서 동일한 프로세스가 있는 경우 SameProcessCount값을 1 증가시킨다.

```
48                  if(tmp_priority1[i]==priority&& SameProcessCount==0){ //덱에 동일한 프로세스가 없는 경우
49                      PriorityCount++; //동일한 우선순위가 있는지 확인
50                      tmp_q.add(tmp_arr[j]); //임시저장소인 큐에 저장
51                  }
```

정렬한 우선순위와 덱에 동일한 프로세스가 있는지 48번라인에서 검사를 한 후 참이면 PriorityCount값을 증가시켜서 동일한 우선순위가 있는지 확인한 후, 임시저장소인 큐에 해당 조건에 맞는 프로세스 정보를 저장한다.

```
53          if(PriorityCount==1){ //PriorityCount가 1인 경우는 동일한 우선순위가 없는 것을 의미
54              while(!tmp_q.isEmpty()){
55                  d.add(tmp_q.poll()); //덱에 삽입
56              }
```

48번 라인의 조건을 통과해서 만약 PriorityCount값이 1이면 동일한 우선순위를 가진 프로세스가 없다는 의미이므로 큐에 있는 값을 비선점 우선순위 정렬조건에 맞게끔 덱에 삽입을 한다.

```
58          else if(PriorityCount>1){ //동일한 우선순위를 가진 프로세스가 있는 경우
59              String str1=tmp_q.poll();
60              String str2=tmp_q.poll();
61              StringTokenizer str1Token=new StringTokenizer(str1);
62              StringTokenizer str2Token=new StringTokenizer(str2);
63                  str1Token.nextToken();
64                  int arrtime1=Integer.parseInt(str1Token.nextToken());
65                  str1Token.nextToken();str1Token.nextToken();str1Token.nextToken();
66                  str2Token.nextToken();
67                  int arrtime2=Integer.parseInt(str2Token.nextToken());
68                  str2Token.nextToken();str2Token.nextToken();str2Token.nextToken();
69                  if(arrtime1>arrtime2) //도착시간이 더 빠른 프로세스를 덱에 저장
70                      d.add(str2);
71                  else
72                      d.add(str1);
73          }
```

PriorityCount값이 1보다 크면 동일한 우선순위를 가진 프로세스가 존재한다는 의미이고 큐에 최소 2개이상의 프로세스가 들어가 있으니 poll()을 이용하여 빼낸 다음 각각의 프로세스정보를 StringTokenizer로 분리한다음 두 프로세스의 도착시간을 비교해서 도착시간이 빠른 프로세스를 덱에 삽입을 한다. 그런 다음 다시 30번라인의 큰 반복문의 I값을 1 증가시켜 다음 반복을 이와 같이 진행한다.

```
print_process.print(d, ProcessCount);
```

덱에 순차적으로 들어가있는 프로세스정보를 비선점형 스케줄링 출력 메소드에 파라미터로 전달해 주어서 간트차트, 각 프로세스별 대기시간, 평균대기시간, 각 프로세스별 응답시간, 평균응답시간, 각 프로세스별 반환시간, 평균반환시간을 출력해준다.

4-5) RR스케줄링

```
11        while (total_servicetime != ServiceTimeSum) { //현재까지의 실행시간이 전체 실행시간이 되기 전까지 반복
12            String str = q.poll(); //큐에 들어있는 프로세스를 str에 저장
13            StringTokenizer strToken = new StringTokenizer(str);
14            ProcessId = strToken.nextToken();
15            ArriveTime = Integer.parseInt(strToken.nextToken());
16            ServiceTime = Integer.parseInt(strToken.nextToken());
17            strToken.nextToken();
18            ResponseTime = Integer.parseInt(strToken.nextToken());
```

while문의 조건이 의미하는 것은 스케줄링과정을 1초 단위로 확인하는 것을 의미하고 ServiceTimeSum은 스케줄링의 전체 실행시간을 초기화해놨고, total_servicetime은 프로세스가 한번 작업할 때마다 1씩 증가시켜서 total_servicetime이 ServiceTimeSum과 같아질 때 까지 반복을 진행한다.
그리고 큐에 가장 앞부분에 있는 프로세스를 꺼내어서 작업을 진행한다.

```
20            for (int i = 1; i <= ProcessCount; i++) {
21                if (tmp_processId[i].equals(ProcessId)) { //정렬된 프로세스ID와 큐에서 꺼낸 프로세스ID가 동일한 경우
22                    if (total_servicetime >= ArriveTime) { //현재 실행시간안에 도착한 프로세스인 경우
23                        check[i] = 0; //한번 검사했다는 걸 표시
```

프로세스 개수만큼 반복을 돌리면서 큐에서 꺼내 프로세스 ID와 도착시간순으로 정렬된 프로세스 ID배열과 값이 같다면 21번라인의 조건에 맞으니 22번 라인으로 들어와서 현재까지의 실행시간안에 프로세스가 도착했다는 했다면 23번 라인으로와서 해당 프로세스의 인덱스를 한번 검사했다는 의미로 check배열의 값을 1에서 0으로 초기화를 해준다.

```
25                    if (SaveServiceTime[i] >= TimeQuantum) { //해당 프로세스의 실행시간이 타임슬라이스보다 큰 경우
26                        for (int j = 1; j <= TimeQuantum; j++) {
27                            wait_time[i] += (total_servicetime - tmp_arrivetime[i]); //대기시간 저장
28                            SaveServiceTime[i] -= 1; //해당 프로세스의 남은 실행시간 1초 감소
29                            total_servicetime++; //전체 실행시간 1초 증가
```

21번,22번 라인의 조건에 부합한 경우 25번 라인 if문이 현재 프로세스의 남은 실행시간이 타임슬라이스보다 큰 경우를 의미한다.
만약 남은 실행시간이 타임슬라이스보다 큰 경우는 26번 라인과 같이 타임슬라이스만큼 반복을 하면서 해당 프로세스의 대기시간을 계산하고, 해당 프로세의 남은 실행시간을 1씩 감소하고 현재까지의 전체 실행시간을 1씩 증가시킨다.

```
31                    for (int m = 1; m <= ProcessCount; m++) { //큐에 새로운 실행시간 안에 도착한 프로세스 삽입
32                        StringTokenizer processToken = new StringTokenizer(process[m]);
33                        String processId1 = processToken.nextToken();
34                        int arriveTime1 = Integer.parseInt(processToken.nextToken());
35                        int serviceTime1 = Integer.parseInt(processToken.nextToken());
36                        processToken.nextToken();
37                        int responseTime1 = Integer.parseInt(processToken.nextToken());
38                        /*
39                        현재 실행시간 안에 도착한 프로세스와 한번도 검사하지 않은 프로세스인 경우
40                         */
41                        if (total_servicetime >= arriveTime1 && check[m] != 0) {
42                            q.add(process[m]); //큐에 해당 프로세스 정보 삽입
43                            check[m] = 0; //한번 검사했다는 것을 표시
44                        }
```

26번 라인의 반복 즉, cpu를 할당받은 프로세스가 타임슬라이스 만큼 작업

을 하는 동안 현재까지의 실행시간안에 도착한 프로세스가 있는지 검사하는 알고리즘이다. 32번 라인에서 프로세스 정보를 담고 있는 process배열에서 하나씩 프로세스를 꺼내어 각각의 정보대로 StringTokenizer로 분해한 다음, 41번 라인의 조건을 통해 현재까지의 실행시간안에 도착한 프로세스가 있다면 큐에 프로세스 정보를 삽입하고 해당 프로세스를 한번 검사했다는 의미로 check배열의 값을 1에서 0으로 초기화해준다.

```
47                          if (response_time[i] == 0) //응답시간을 한번만 계산
48                              response_time[i] = (int) ((total_servicetime + ResponseTime) - ArriveTime);
```

응답시간배열의 값이 0인 경우는 응답시간은 실행되고 첫 반응이 있는 시간을 구하는 것이니 딱 한번만 계산을 하면 되는 것이다.
즉 21, 22번 라인의 조건에 부합하여 cpu를 할당받은 프로세스가 처음 실행된 프로세스이면 응답시간을 한번만 계산을 해주고 실행한 프로세스에 대한 응답시간을 48번 라인에서 계산을 해준다.

```
51                          if (SaveServiceTime[i] == 0) { //프로세스가 주어진 작업을 다한 경우
52                              return_time[i] = (int) (total_servicetime - ArriveTime); //반환시간 계산
53                              break;
54                          }
```

만약 타임슬라이스 만큼 실행한 프로세스가 주어진 작업을 다 한 경우 51번 라인의 조건에 들어와서 52번 라인에서 반환시간을 계산해준다.
또한 더 이상 할 작업이 없다면 26번라인의 for문을 벗어나게 된다.
반환시간은 프로세스의 작업이 마무리된 시간에서 도착시간을 빼주면 반환시간이 나오게 된다.

```
q.add(process[i]); //타임슬라이스만큼 작업하고 다시 큐에 삽입
```

프로세스가 타임슬라이스만큼 작업을 수행하면 다시 큐의 맨 뒤에 해당 프로세스정보를 삽입한다.

```
58              else if (SaveServiceTime[i] != 0 && SaveServiceTime[i] < TimeQuantum) { //프로세스 남은 작업시간이 0이 아니고 타임슬라이스보다 적은경우
59                  for (int j = 1; j <= TimeQuantum; j++) {
60                      wait_time[i] += (total_servicetime - tmp_arrivetime[i]); //대기시간 저장
61                      SaveServiceTime[i] -= 1; //남은 실행시간 1초 감소
62                      total_servicetime++; //전체 실행시간 1초 증가
```

58번 라인의 if문이 의미하는 것은 프로세스의 남은 작업시간이 타임슬라이스보다 작은 경우를 의미하고 위의 if문에 들어오면 위의 25번라인의 조건문 안에서 한 방식과 똑같이 프로세스의 작업을 수행하고, 대기시간, 응답시간, 반환시간을 계산하여서 각각의 배열에 값을 저장한다.

```
q.add(process[i]); //타임슬라이스만큼 작업하고 다시 큐에 삽입
```

위의 프로세스의 남은 작업시간이 타임슬라이스보다 큰 경우처럼 프로세스가 타임슬라이스만큼 작업을 수행하면 다시 큐의 맨 뒤에 해당 프로세스정보를 삽입한다.

```
print_process.print(ProcessCount, wait_time, tmp_processId, return_time, ganttchatt, response_time);
```

11번라인의 while문이 종료되면 계산된 대기시간, 반환시간, 응답시간, 간트차트를 선점형 스케줄링의 결과를 출력해주는 함수의 파라미터로 전달해준다.

4-6) SRT스케줄링

```
10          StringTokenizer processrealToken=new StringTokenizer(SrtProcess[1]); //도착시간이 0인 프로세스 정보
11              ProcessId = processrealToken.nextToken();
12              ArriveTime = Integer.parseInt(processrealToken.nextToken());
13              ServiceTime = Integer.parseInt(processrealToken.nextToken());
14              processrealToken.nextToken();processrealToken.nextToken();
15              if (SaveServiceTime[1] >= TimeQuantum) { //도착시간이 0인 프로세스의 실행시간이 타임슬라이스보다 큰 경우
16                  wait_time[1] += (total_servicetime - tmp_arrivetime[1]); //대기시간 저장
17                  SaveServiceTime[1] -= TimeQuantum;
18                  response_time[1]= (int) ((restime[1]+total_servicetime)- ArriveTime); //응답시간 계산
19                  total_servicetime += TimeQuantum;
20                  tmp_arrivetime[1] = total_servicetime;
21                  for(int m = 0; m< TimeQuantum; m++)
22                      ganttchatt[gc++]= ProcessId;
23                  if (SaveServiceTime[1] == 0) { //프로세스가 주어진 작업을 다한 경우
24                      return_time[1] = (int) (total_servicetime - ArriveTime); //반환시간 계산
25                  }
26              }
```

SrtProcess[1]에는 도착시간이 0인 프로세스 정보가 들어있다.
우선 스케줄링에서 가장 먼저 도착한 프로세스 먼저 작업을 수행하니까 먼저 StringTokenizer를 이용하여 각각의 정보대로 분리한다음 15번 라인에서 해당 프로세스의 남은 실행시간이 타임슬라이스보다 큰 경우 조건부에 들어온다. 다른 선점형 스케줄링과 같이 대기시간, 응답시간을 계산하고 해당 프로세스의 남은 실행시간을 타임슬라이스만큼 감소시킨다.
23번라인은 만약 해당 프로세스가 주어진 작업을 완료하면 반환시간을 계산한다.

```
28          else if (SaveServiceTime[1] != 0 && SaveServiceTime[1] < TimeQuantum) { //도착시간이 0인 프로세스의 실행시간이 타임슬라이스보다 작은경우
29              wait_time[1] += (total_servicetime - tmp_arrivetime[1]); //대기시간 저장
30              response_time[1]= (int) ((restime[1]+total_servicetime)- ArriveTime); //반응시간 계산
31              total_servicetime += SaveServiceTime[1];
32              for(int m = 0; m< SaveServiceTime[1]; m++)
33                  ganttchatt[gc++]= ProcessId;
34          }
```

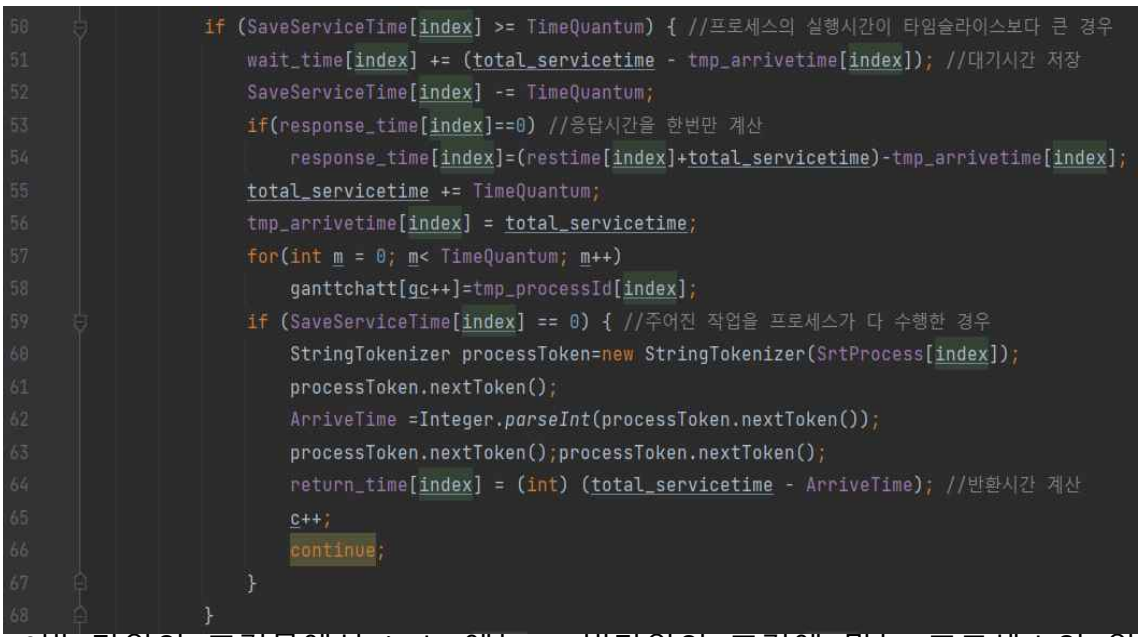
28번 라인은 도착시간이 가장 빠른 프로세스의 남은 실행시간이 타임슬라이스보다 작은 경우를 의미하고, 조건에 맞다면 15번라인의 조건내부와 동일하게 대기시간, 응답시간등을 계산한다.

```
36      while(total_servicetime!= ServiceTimeSum) { //현재까지의 실행시간이 전체 실행시간이 되기 전까지 반복
37          int index=0;
38          if (c!= ProcessCount) {
39              int min=10000; //최소 실행시간을 매우 큰 수로 임의지정
40              for(int i = 1; i<= ProcessCount; i++){
41                  /*
42                  최단 작업시간이면서 수행할 작업이 남아있고 현재까지의 실행시간안에 도착한 프로세스와 남은 작업시간이 같은 경우 먼저 도착한 프로세스 먼저 수행하는 조건
43                   */
44                      if(min>= SaveServiceTime[i]&& SaveServiceTime[i]!=0&&tmp_arrivetime[i]<=total_servicetime&&min!= SaveServiceTime[i]){
45                              min = SaveServiceTime[i];
46                              index = i; //조건에 맞는 프로세스의 인덱스를 index에 저장
47                          }
48                      }
```

36번 라인의 while문의 조건은 현재까지의 실행시간이 전체 실행시간이 되

기전까지 반복하는 의미고, 1초간격으로 스케줄링을 확인하는 의미이다.
38번 라인의 if문이 의미하는 것은 c는 프로세스가 주어진 작업을 완료한 프로세스가 있는 경우에만 1씩 증가하므로 전체 프로세스가 주어진 작업을 완료하면 조건문에 들어오지 않아 로직을 수행하지 않는다.

이번 스케줄링 코드에서 가장 중요한 부분입니다.
SRT의 우선순위를 구하는 기준은 남은 작업시간이 가장 적은 프로세스가 cpu할당을 먼저 받는 스케줄링이다.
44라인의 조건이 의미하는 것은 가장 최단 작업시간을 가지며, 남은 작업이 있어야하고 현재까지의 실행시간안에 도착한 프로세스이고, 만약 남은 작업이 같다면 우선적으로 도착한 프로세스를 먼저 수행하는 조건문이다.
index변수에는 조건에 맞는 프로세스의 인덱스값을 저장한다.

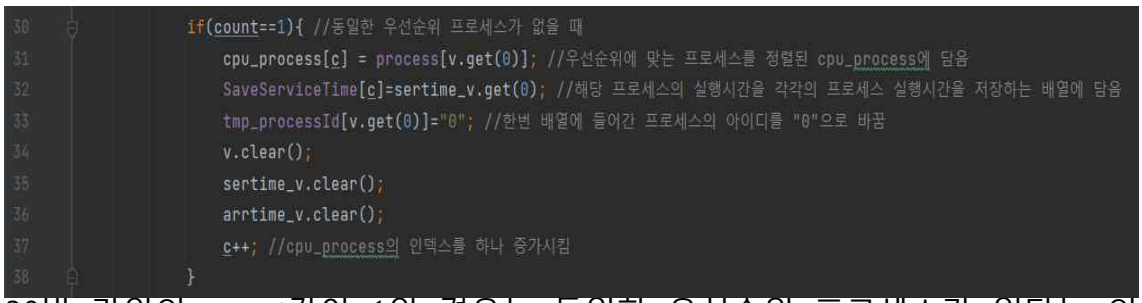


50번 라인의 조건문에서 index에는 44번라인의 조건에 맞는 프로세스의 인덱스가 저장되어 있고, 해당 프로세스의 남은 실행시간이 타임슬라이스보다 큰 경우 조건에 들어와서 다른 선점형 스케줄링과 똑같이 대기시간, 응답시간을 구한다. 여기서 만약 프로세스가 주어진 작업을 다 수행하면 c값을 증가시키며 38번라인의 조건에 충족하는지 확인한다.

4-7) 선점 우선순위 스케줄링

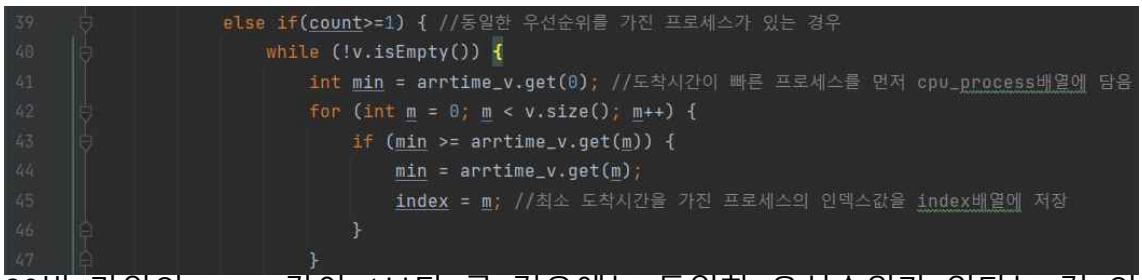
```
15      for(int i = 1; i<= ProcessCount; i++){
16          int count=0;
17          for(int j = 1; j<= ProcessCount; j++){
18              StringTokenizer processToken=new StringTokenizer(process[j]);
19                  ProcessId = processToken.nextToken();
20                  ArriveTime = Integer.parseInt(processToken.nextToken());
21                  ServiceTime = Integer.parseInt(processToken.nextToken());
22                  priority = Integer.parseInt(processToken.nextToken());
23                 if(tmp_priority[i]==priority&&tmp_processId[j]!="0"){ //우선순위대로 정렬, 벡터에 동일한 프로세스가 있는지 검사
24                      count++;
25                      v.add(j);//인덱스를 담음
26                      sertime_v.add((int) ServiceTime);
27                      arrtime_v.add((int) ArriveTime);
28                  }
29          }
```

23번라인의 if문에서 tmp_priority배열에는 도착시간 순으로 정렬이되어있기 때문에 정렬된 우선순위와 같은 프로세스이고 cpu_process배열에 동일한 프로세스가 저장되는 걸 막기 위해 tmp_processId배열의 값이 0이 아니어야한다. 해당 조건에 맞는 프로세스가 있다면 count값을 증가하고 벡터를 이용하여 해당 인덱스와 실행시간, 도착시간을 각각의 벡터에 삽입한다.



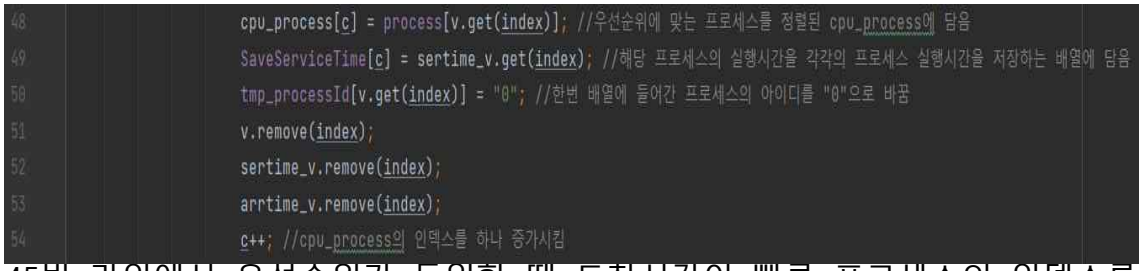
```
30          if(count==1){ //동일한 우선순위 프로세스가 없을 때
31              cpu_process[c] = process[v.get(0)]; //우선순위에 맞는 프로세스를 정렬된 cpu_process에 담음
32              SaveServiceTime[c]=sertime_v.get(0); //해당 프로세스의 실행시간을 각각의 프로세스 실행시간을 저장하는 배열에 담음
33              tmp_processId[v.get(0)]="0"; //한번 배열에 들어간 프로세스의 아이디를 "0"으로 바꿈
34              v.clear();
35              sertime_v.clear();
36              arrtime_v.clear();
37              c++; //cpu_process의 인덱스를 하나 증가시킴
38          }
```

30번 라인의 count값이 1인 경우는 동일한 우선순위 프로세스가 없다는 의미이다. 그래서 스케줄링 정렬 조건에 맞게 정렬된 cpu_process배열에 23번 라인에서 받아온 프로세스의 정보를 저장하고 해당 프로세스의 실행시간을 받고, 한번 검사를 했으니 프로세스ID를 0으로 변경해준다.
그리고 마지막으로 c값을 증가시키면서 cpu_process의 인덱스를 하나 증가시킨다.



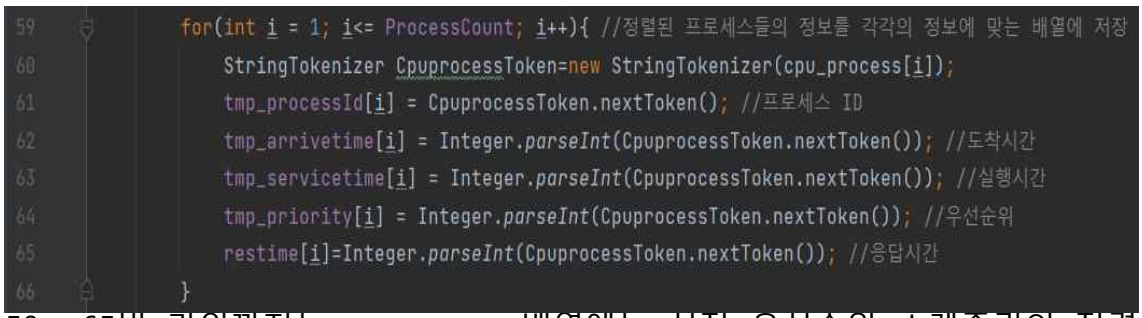
```
39          else if(count>=1) { //동일한 우선순위를 가진 프로세스가 있는 경우
40              while (!v.isEmpty()) {
41                  int min = arrtime_v.get(0); //도착시간이 빠른 프로세스를 먼저 cpu_process배열에 담음
42                  for (int m = 0; m < v.size(); m++) {
43                      if (min >= arrtime_v.get(m)) {
44                          min = arrtime_v.get(m);
45                          index = m; //최소 도착시간을 가진 프로세스의 인덱스값을 index배열에 저장
46                      }
47                  }
```

39번 라인의 count값이 1보다 큰 경우에는 동일한 우선순위가 있다는 걸 의미한다. 그러므로 인덱스와 실행시간, 도착시간을 저장하는 벡터에 여러값이 들어가있으므로 40번 라인의 whiile문을 반복하면서 42번 라인의 for문을 이용하여 우선순위가 같을 때 먼저 도착한 프로세스의 인덱스값을 구한다.



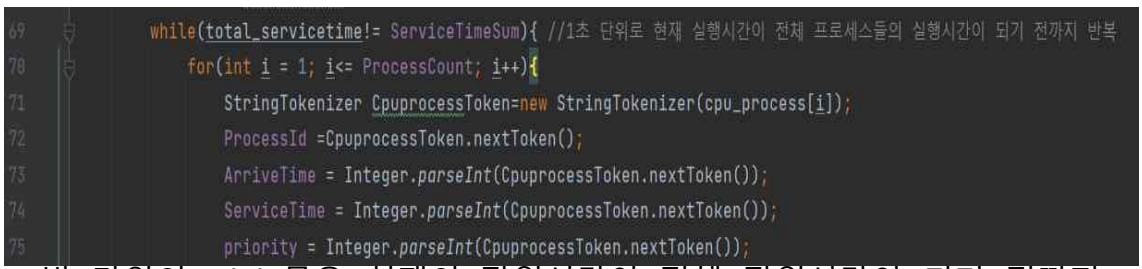
```
48                  cpu_process[c] = process[v.get(index)]; //우선순위에 맞는 프로세스를 정렬된 cpu_process에 담음
49                  SaveServiceTime[c] = sertime_v.get(index); //해당 프로세스의 실행시간을 각각의 프로세스 실행시간을 저장하는 배열에 담음
50                  tmp_processId[v.get(index)] = "0"; //한번 배열에 들어간 프로세스의 아이디를 "0"으로 바꿈
51                  v.remove(index);
52                  sertime_v.remove(index);
53                  arrtime_v.remove(index);
54                  c++; //cpu_process의 인덱스를 하나 증가시킴
```

45번 라인에서 우선순위가 동일할 때 도착시간이 빠른 프로세스의 인덱스를 index에 초기화를 하였기 때문에 48~50번 라인을 통해 각각의 배열에 해당 프로세스의 정보를 저장하고 51~53번 라인을 통해 해당 index값을 벡터에서 삭제한다.
마지막으로 54번 라인을 통해 cpu_process배열의 인덱스를 하나 증가시킨다.



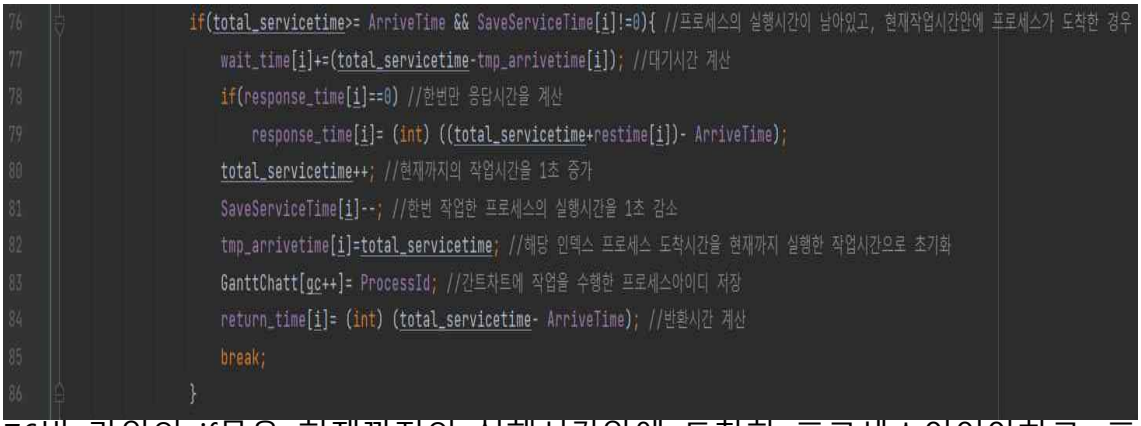
```
59          for(int i = 1; i<= ProcessCount; i++){ //정렬된 프로세스들의 정보를 각각의 정보에 맞는 배열에 저장
60              StringTokenizer CpuprocessToken=new StringTokenizer(cpu_process[i]);
61              tmp_processId[i] = CpuprocessToken.nextToken(); //프로세스 ID
62              tmp_arrivetime[i] = Integer.parseInt(CpuprocessToken.nextToken()); //도착시간
63              tmp_servicetime[i] = Integer.parseInt(CpuprocessToken.nextToken()); //실행시간
64              tmp_priority[i] = Integer.parseInt(CpuprocessToken.nextToken()); //우선순위
65              restime[i]=Integer.parseInt(CpuprocessToken.nextToken()); //응답시간
66          }
```

59~ 65번 라인까지는 cpu_process배열에는 선점 우선순위 스케줄링의 정렬 조건에 맞는 순서로 프로세스 정보가 들어가 있는 상태이기 때문에 반복을 돌면서 프로세스ID, 도착시간, 실행시간, 우선순위, 응답시간의 배열에 각각의 프로세스 정보를 저장한다.



```
69          while(total_servicetime!= ServiceTimeSum){ //1초 단위로 현재 실행시간이 전체 프로세스들의 실행시간이 되기 전까지 반복
70              for(int i = 1; i<= ProcessCount; i++){
71                  StringTokenizer CpuprocessToken=new StringTokenizer(cpu_process[i]);
72                  ProcessId =CpuprocessToken.nextToken();
73                  ArriveTime = Integer.parseInt(CpuprocessToken.nextToken());
74                  ServiceTime = Integer.parseInt(CpuprocessToken.nextToken());
75                  priority = Integer.parseInt(CpuprocessToken.nextToken());
```

69번 라인의 while문은 현재의 작업시간이 전체 작업시간이 되기 전까지 1초 간격으로 확인하는 반복조건이다.
cpu_process에 들어있는 프로세스 정보를 StringTokenizer을 사용하여 분리한다.

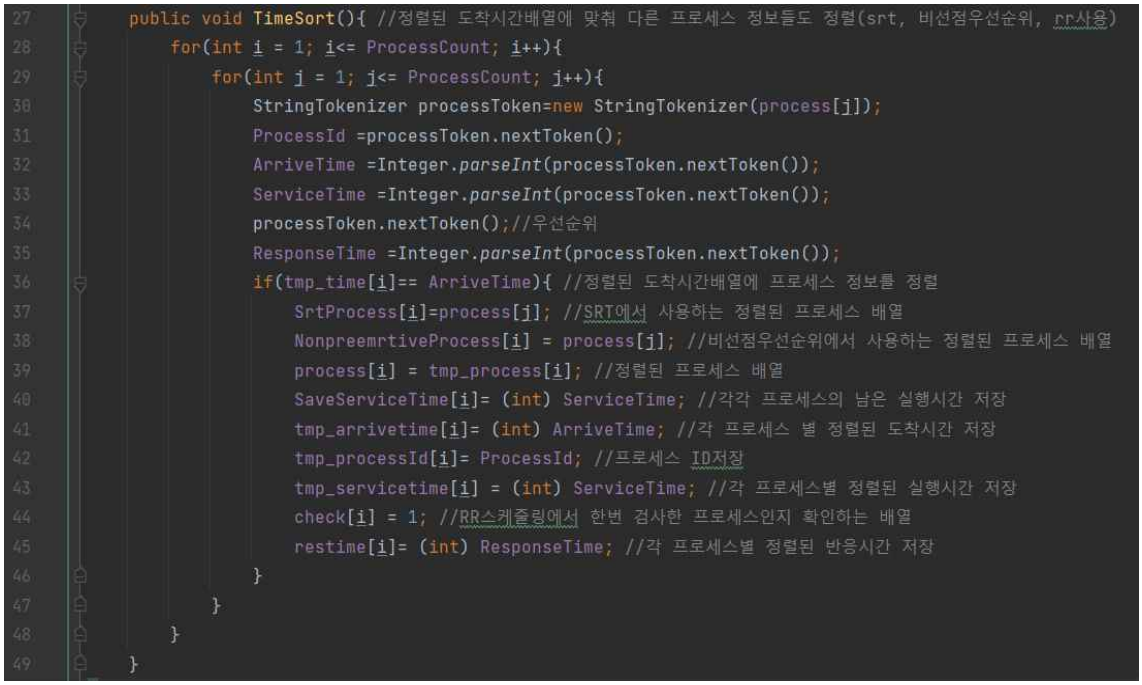


76번 라인의 if문은 현재까지의 실행시간안에 도착한 프로세스이어야하고, 프로세스가 주어진 작업을 다 완료하지 않은 상태이어야 하는 것을 의미한다. 76번 라인의 조건에 맞는 프로세스의 인덱스를 이용하여 대기시간, 응답시간, 반환시간을 계산하고, 현재 실행시간을 증가하고 프로세스의 남은 작업시간을 1초 감소하고 break를 통해 70번라인의 for문으로 돌아가서 똑같이 1초마다 작업을 수행한다.

4-8) 정렬(Sort) 메소드

```
4    public  void ArriveTimeSort(){ //도착시간을 정렬해주는 메소드
5        for (int i = 1; i <= ProcessCount; i++) {
6            StringTokenizer processToken = new StringTokenizer(process[i]);
7            ProcessId = processToken.nextToken(); //프로세스 ID
8            ArriveTime = Integer.parseInt(processToken.nextToken()); //도착시간
9            ServiceTime = Integer.parseInt(processToken.nextToken()); //실행시간
10           tmp_priority[i] = Integer.parseInt(processToken.nextToken()); //우선순위
11           processToken.nextToken(); //응답시간
12           tmp_time[i] = (int) ArriveTime; //도착시간 모음배열
13           ServiceTimeSum += ServiceTime;
14           if (ArriveTime == 0){ //도착시간이 0인 프로세스를 검사
15               q.add(process[i]); //도착시간이 0인 프로세스를 큐에 저장
16               deque.addLast(process[i]); //도착시간이 0인 프로세스를 덱에 저장
17           }
18           tmp_servicetime[i]= (int) ServiceTime; //실행시간 저장
19           tmp_arrivetime[i]= (int) ArriveTime; //도착시간 저장
20           tmp_processId[i]= ProcessId; //프로세스 ID저장
21       }
22       Arrays.sort(tmp_time); //도착시간 정렬
23       Arrays.sort(tmp_priority); //우선순위 정렬
24       Arrays.sort(tmp_servicetime); //실행시간 정렬
25   }
```

ArriveTimeSort()메소드는 파일에서 읽어온 프로세스 정보가 저장되어 있는 process배열을 StringTokenizer을 통해 각각의 정보를 분리하고 14번 라인에서는 도착시간이 0인 프로세스 정보를 큐와 덱에 각각 삽입을 하고, 5번 라인의 for문 끝나고 22~24번 라인을 통해 프로세스의 도착시간, 우선순위, 실행시간을 sort함수를 통해 오름차순으로 정렬을 한다.



```
27      public void TimeSort(){ //정렬된 도착시간배열에 맞춰 다른 프로세스 정보들도 정렬(srt, 비선점우선순위, rr사용)
28          for(int i = 1; i<= ProcessCount; i++){
29              for(int j = 1; j<= ProcessCount; j++){
30                  StringTokenizer processToken=new StringTokenizer(process[j]);
31                  ProcessId =processToken.nextToken();
32                  ArriveTime =Integer.parseInt(processToken.nextToken());
33                  ServiceTime =Integer.parseInt(processToken.nextToken());
34                  processToken.nextToken();//우선순위
35                  ResponseTime =Integer.parseInt(processToken.nextToken());
36                  if(tmp_time[i]== ArriveTime){ //정렬된 도착시간배열에 프로세스 정보를 정렬
37                      SrtProcess[i]=process[j]; //SRT에서 사용하는 정렬된 프로세스 배열
38                      NonpreemrtiveProcess[i] = process[j]; //비선점우선순위에서 사용하는 정렬된 프로세스 배열
39                      process[i] = tmp_process[i]; //정렬된 프로세스 배열
40                      SaveServiceTime[i]= (int) ServiceTime; //각각 프로세스의 남은 실행시간 저장
41                      tmp_arrivetime[i]= (int) ArriveTime; //각 프로세스 별 정렬된 도착시간 저장
42                      tmp_processId[i]= ProcessId; //프로세스 ID저장
43                      tmp_servicetime[i] = (int) ServiceTime; //각 프로세스별 정렬된 실행시간 저장
44                      check[i] = 1; //RR스케줄링에서 한번 검사한 프로세스인지 확인하는 배열
45                      restime[i]= (int) ResponseTime; //각 프로세스별 정렬된 반응시간 저장
46                  }
47              }
48          }
49      }
```

TimeSort()메소드는 위의 ArriveTimeSort()메소드에서 도착시간대로 정렬한 tmp_time배열을 이용하여 29번 라인의 작은 내부 for문을 돌리면서 36번 라인을 통해 37~45번 라인까지 각각의 프로세스 정보를 나타내는 배열에 도착시간순을 기준으로 정렬시키는 과정이다.

<5. 실행 결과>

1) FCFS스케줄링

```
비선점형 - FCFS스케줄링
<간트차트>
[P1][P1][P1][P1][P1][P1][P1][P1][P1][P1][P2][P2][P2][P2][P2][P2][P2][P2][P2][P2][P2][P2][P2][P2][P2][P2][P2]
[P2][P2][P2][P2][P2][P2][P2][P2][P2][P3][P3][P3][P3][P3][P3][P4][P4][P4][P4][P5][P5][P5][P5][P5][P5][P5][P5][P5][P5][P5][P5][P5]
P1의 대기시간: 0.0
P2의 대기시간: 9.0
P3의 대기시간: 36.0
P4의 대기시간: 41.0
P5의 대기시간: 44.0
평균 대기 시간(AWT): 26.0
P1의 응답시간: 2.0
P2의 응답시간: 11.0
P3의 응답시간: 38.0
P4의 응답시간: 43.0
P5의 응답시간: 46.0
평균응답시간(ART): 28.0
P1의 반환시간: 10
P2의 반환시간: 37
P3의 반환시간: 42
P4의 반환시간: 45
P5의 반환시간: 58
평균 반환 시간(ATT): 38.4
```

2) SJF스케줄링

```
비선점형 - SJF스케줄링
<간트차트>
[P1][P1][P1][P1][P1][P1][P1][P1][P1][P1][P4][P4][P4][P4][P3][P3][P3][P3][P3][P5][P5][P5][P5][P5][P5][P5][P5][P5]
[P5][P5][P5][P5][P5][P2][P2][P2][P2][P2][P2][P2][P2][P2][P2][P2][P2][P2][P2][P2][P2][P2][P2][P2][P2][P2][P2][P2][P2][P2][P2][P2]
P1의 대기시간: 0.0
P4의 대기시간: 7.0
P3의 대기시간: 12.0
P5의 대기시간: 16.0
P2의 대기시간: 33.0
평균 대기 시간(AWT): 13.6
P1의 응답시간: 2.0
P4의 응답시간: 9.0
P3의 응답시간: 14.0
P5의 응답시간: 18.0
P2의 응답시간: 35.0
평균응답시간(ART): 15.6
P1의 반환시간: 10
P4의 반환시간: 11
P3의 반환시간: 18
P5의 반환시간: 30
P2의 반환시간: 61
평균 반환 시간(ATT): 26.0
```

3) HRN스케줄링

```
비선점형 - HRN스케줄링
<간트차트>
[P1][P1][P1][P1][P1][P1][P1][P1][P1][P1][P4][P4][P4][P4][P3][P3][P3][P3][P3][P5][P5][P5][P5][P5][P5][P5][P5][P5]
[P5][P5][P5][P5][P5][P2][P2][P2][P2][P2][P2][P2][P2][P2][P2][P2][P2][P2][P2][P2][P2][P2][P2][P2][P2][P2][P2][P2][P2][P2][P2][P2][P2][P2][P2][P2][P2][P2]
P1의 대기시간: 0.0
P4의 대기시간: 7.0
P3의 대기시간: 12.0
P5의 대기시간: 16.0
P2의 대기시간: 33.0
평균 대기 시간(AWT): 13.6
P1의 응답시간: 2.0
P4의 응답시간: 9.0
P3의 응답시간: 14.0
P5의 응답시간: 18.0
P2의 응답시간: 35.0
평균응답시간(ART): 15.6
P1의 반환시간: 10
P4의 반환시간: 11
P3의 반환시간: 18
P5의 반환시간: 30
P2의 반환시간: 61
평균 반환 시간(ATT): 26.0
```

4) 비선점 우선순위 스케줄링

```
비선점형 - 비선점 우선순위 스케줄링
<간트차트>
[P1][P1][P1][P1][P1][P1][P1][P1][P1][P1][P4][P4][P4][P4][P2][P2][P2][P2][P2][P2][P2][P2][P2][P2][P2][P2][P2][P2][P2][P2][P2][P2][P2][P2][P2][P2][P2][P2]
[P2][P2][P2][P2][P2][P2][P2][P2][P2][P2][P2][P2][P2][P2][P5][P5][P5][P5][P5][P5][P5][P5][P5][P5][P5][P5][P5][P5][P5][P5][P5][P5][P5][P5][P5][P5][P5][P5]
P1의 대기시간: 0.0
P4의 대기시간: 7.0
P2의 대기시간: 13.0
P5의 대기시간: 38.0
P3의 대기시간: 54.0
평균 대기 시간(AWT): 22.4
P1의 응답시간: 2.0
P4의 응답시간: 9.0
P2의 응답시간: 15.0
P5의 응답시간: 40.0
P3의 응답시간: 56.0
평균응답시간(ART): 24.4
P1의 반환시간: 10
P4의 반환시간: 11
P2의 반환시간: 41
P5의 반환시간: 52
P3의 반환시간: 60
평균 반환 시간(ATT): 34.8
```

5) RR스케줄링

```
선점형 - 라운드로빈(RR)스케줄링
<간트차트>
[P1][P1][P2][P2][P3][P3][P1][P1][P4][P4][P5][P5][P2][P2][P3][P3][P1][P1][P4][P4][P5][P5][P2][P2][P3][P3][P1][P1][P5][P5]
[P2][P2][P1][P1][P5][P5][P2][P2][P5][P5][P2][P2][P5][P5][P2][P2][P5][P5][P2][P2][P2][P2][P2][P2][P2][P2][P2][P2][P2][P2][P2]
P1의 대기시간: 24
P2의 대기시간: 33
P3의 대기시간: 18
P4의 대기시간: 13
P5의 대기시간: 30
평균 대기 시간(AWT): 23.6
P1의 응답시간: 3
P2의 응답시간: 4
P3의 응답시간: 5
P4의 응답시간: 8
P5의 응답시간: 9
평균 응답 시간(ART): 5.8
P1의 반환시간: 34
P2의 반환시간: 61
P3의 반환시간: 24
P4의 반환시간: 17
P5의 반환시간: 44
평균 반환 시간(ATT): 36.0
```

6) SRT스케줄링

```
선점형 - SRT스케줄링
<간트차트>
[P1][P1][P3][P3][P3][P3][P3][P3][P4][P4][P4][P4][P1][P1][P1][P1][P1][P1][P1][P5][P5][P5][P5][P5][P5][P5][P5][P5]
[P5][P5][P5][P5][P2][P2][P2][P2][P2][P2][P2][P2][P2][P2][P2][P2][P2][P2][P2][P2][P2][P2][P2][P2][P2][P2][P2][P2][P2][P2]
P1의 대기시간: 10
P2의 대기시간: 33
P3의 대기시간: 0
P4의 대기시간: 5
P5의 대기시간: 16
평균 대기 시간(AWT): 12.8
P1의 응답시간: 2
P2의 응답시간: 35
P3의 응답시간: 2
P4의 응답시간: 7
P5의 응답시간: 18
평균 응답 시간(ART): 12.8
P1의 반환시간: 20
P2의 반환시간: 61
P3의 반환시간: 6
P4의 반환시간: 9
P5의 반환시간: 30
평균 반환 시간(ATT): 25.2
```

7) 선점 우선순위 스케줄링

```
선점형 - 선점형 우선순위 스케줄링
<간트차트>
[P1][P2][P2][P4][P4][P4][P4][P2][P2][P2][P2][P2][P2][P2][P2][P2][P2][P2][P2][P2][P2][P2][P2][P2][P2][P2][P2][P2][P2][P2]
[P2][P2][P2][P5][P5][P5][P5][P5][P5][P5][P5][P5][P5][P5][P5][P5][P5][P1][P1][P1][P1][P1][P1][P1][P1][P3][P3][P3][P3][P3]
P4의 대기시간: 0
P2의 대기시간: 4
P5의 대기시간: 29
P1의 대기시간: 46
P3의 대기시간: 54
평균 대기 시간(AWT): 26.6
P4의 응답시간: 2
P2의 응답시간: 2
P5의 응답시간: 31
P1의 응답시간: 2
P3의 응답시간: 56
평균 응답 시간(ART): 18.6
P4의 반환시간: 4
P2의 반환시간: 32
P5의 반환시간: 43
P1의 반환시간: 56
P3의 반환시간: 60
평균 반환 시간(ATT): 39.0
```

<6. 느낀 점>

이번 스케줄링 과제를 처음 받았을 때 감도 잡히지 않고 막막했었다. 이론적으로만 배웠던 스케줄링을 과연 코드로 어떻게 구현해야할까에 대한 생각이 많았다.
그래서 우선 스케줄링에 대해 다시 한번 pdf파일을 보며 원리에 대해 복습하고 노트에 직접 과정을 적어가며 코드를 작성해나갔다. 처음 FCFS스케줄링 코드를 작성할 때 시간도 오래걸렸지만 한번 코드를 완성하고 구현이 성공적으로되자 자신감이 붙고 비선점형은 비슷한 알고리즘이라 금방 구현에 성공하였다. 하지만 선점으로 들어와서 알고리즘 변경의 필요성을 느끼고 처음에 알고리즘을 구상할 때 잘 짤걸이라는 후회가 들었다.
그리고 구상을 끝내고 마무리르 할려고 했을 때 쯤 각종 예외상황 혹은, 우선순위가 동일한 경우에 대한 코드를 작성하지 않은 걸 알고 다시 짜기 시작하였다.
이번 스케줄링 알고리즘을 구현하면서 전에는 예외 처리에 대한 고려를 하지 않고 코드를 작성했는데 이번에 코드를 작성하면서 예외처리에 대한 중요성을 다시 한번 깨달았다.
또한 알고리즘을 구현할 때 처음 베이스가 굉장히 중요하고 잘 짜야 나중에 뒤에 가서 상속을 사용하고 알고리즘을 구현할 때 오류발생도 적고 코드가 뒤죽박죽되는 것을 막을 수 있다는 것을 알게 되었다.
이번 기회에 복잡한 코드를 작성하면서 시작이 중요하다는 걸 다시 한번 깨달았고, 접근 방식이 여러 가지가 있다는 걸 알게 되었습니다.
뿐만 아니라 평소에는 배열 자료구조만 자주 사용했었는데 이번에는 평소 자주 사용하지 않는 큐와 덱, 벡터의 자료구조에 Iterator연산자도 사용하면서 까먹었던 자료구조를 이번 기회에 다시 복습을 하며 사용하는 방법을 익히는 유익한 시간이 되었습니다.
다음 기회에 시간이 된다면 stream연산자를 이용하여 파일에서 읽어온 후 이번에 작성한 코드에 비해 더 간결하고 누구나 주석이 없이도 저의 코드를 이해할 수 있는 가독성이 좋은 그런 알고리즘을 구현해보고 싶다는 생각이 들었습니다.
항상 형식이 정해져 있는 과제들만 하여서 사용할 언어도 정해져 있어 불편함이 있었는데 이번 스케줄링 과제에서는 본인이 원하는 언어를 선택할 수 있는 기회를 주셔서 앞으로 취업을 위해 사용할 언어도 정하는 계기가 되기도 하여서 더욱 더 유익한 과제였고 시간이었던 것 같습니다.
또한 앞으로도 쭉 사용할 언어라고 생각하니까 겉핥기만 하고 가는 느낌이 아니라 좀 더 언어에 대해 깊숙이 알려고 하니까 생각보다 재미도 있었고 성장해나가는 느낌이 들어서 좋았던 것 같습니다.