Term Project 최종보고서

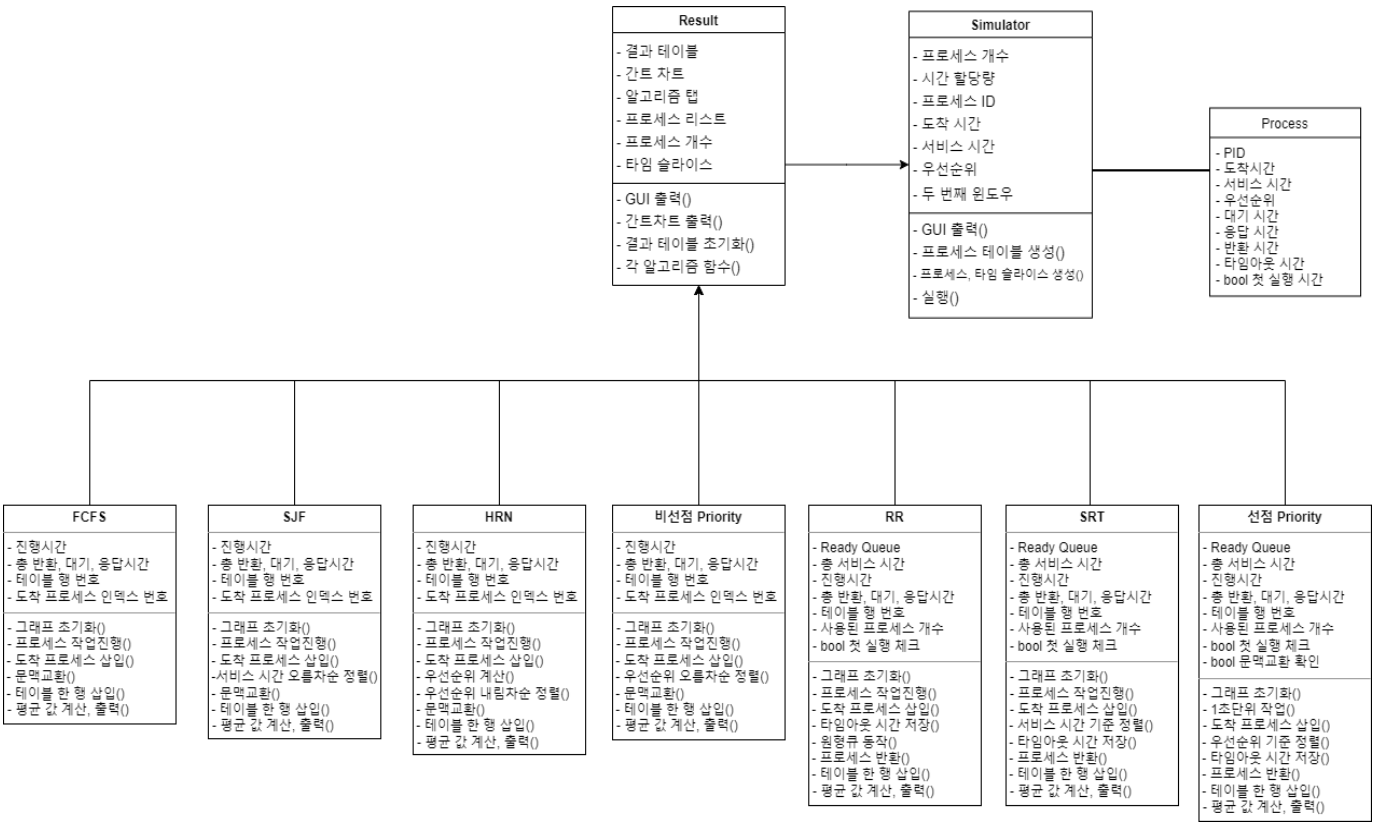
작성일	2022.06.11	전공	컴퓨터소프트웨어공학
작성자	김주원	학번	20184009
강의명	운영체제	강의교수	김대영 교수님

| 목차 |

1. 개발언어

개발언어는 파이썬이다. 선택의 이유는 우선 GUI 의 구현이 깔끔하고 단순하여 스케줄러 시뮬레이터와 같이 단순한 프로그램에 알맞다고 생각하였다. 그리고 7 가지의 크고 많은 알고리즘을 만들기 때문에 가독성이 나빠질 수 있어, 비교적 가독성이 좋은 파이썬을 선택하게 되었다.

2. 함수/클래스 다이어그램



함수, 클래스 다이어그램이다. 프로세스의 정보를 담는 Process 클래스, 첫번째 윈도우인 Simulator 클래스 그리고 결과를 보여주는 두번째 윈도우인 Result 클래스가 있다. Process 클래스의 객체 리스트를 Simulator 클래스의 객체 윈도우에서 생성하고, Simulator 의 객체에서 실행 버튼을 누르게 되면 두번째 윈도우 객체인 Result 가 생성된다. 만들어진 Process 객체 리스트를 복사하고 각각의 알고리즘 함수에서 다른 형태로 사용한다.

3. 자료구조, 주요 변수

먼저 각 알고리즘에 대체적으로 공통적으로 사용되는 변수가 있다. P_num 은 프로세스의 개수 변수이다. ax 는 각 알고리즘의 간트차트 객체이다. Wait_S, Resp_S, Return_S 는 대기시간, 응답시간 반환시간의 합계 변수이다. Exec_T 라는 변수는 작업이 진행된 시간의 변수이다. FCFS_r, SJF_r, HRN_r, NP_r, RR_r, SRT_r, P_r 은 각 알고리즘의 결과 테이블이고, 뒤에 r 이 빠진 FCFS_C 와 같은 변수는 각 알고리즘의 GUI 의 그래프 객체이다.

각 알고리즘에서 다르게 반복돼 사용되는 변수도 있다. PCB 는 비선점형 알고리즘에서는 처음 만들어진 Process 객체 리스트의 복사본으로 사용되고, 선점형 알고리즘 에서는 Ready Queue 로 사용된다. 비선점형 알고리즘에서 사용되는 arrive 는 도착한 마지막 프로세스의 인덱스 번호 변수이다. 선점형 알고리즘에서 사용되는 변수 total_T 는 프로세스의 총 서비스 시간이고, used 는 프로세스 리스트에 사용된 프로세스의 인덱스 번호이며 row 는 결과 테이블의 입력할 행의 번호이다. Process 클래스의 First 변수는 프로세스가 CPU 가 처음 올라왔는지 확인하기 위한 변수이다.

각각의 알고리즘에 따라 자료구조가 달라지게 된다. FCFS 알고리즘은 단순한 큐의 구조를 사용한다. SJF, HRN, 비선점 우선순위, SRT, 선점 우선순위의 알고리즘에선 히프트리 자료구조의 형태가 사용된다. 라운드 로빈의 알고리즘에서는 원형 큐의 자료구조가 사용된다.

4. 핵심코드

```
self.Btn_Assign.clicked.connect(self.Assign_event)        #할당 버튼 커맨드 연결
self.Btn_Creation.clicked.connect(self.Creation_event)    #생성 버튼 커맨드 연결
self.Btn_Execution.clicked.connect(self.Execution_event) #실행 버튼 커맨드 연결
```

각각 할당버튼, 생성버튼, 실행버튼의 커맨드를 연결하였다.

```
def Assign_event(self):      #할당 버튼 커맨드 함수
    self.Process_num = int(self.Process_n.text())    #입력값 프로세스값 변수에 정수형 저장
    self.Process_Table.setRowCount(self.Process_num) #저장된 값 만큼 행 생성
```

할당버튼의 커맨드 함수이다. 입력된 텍스트를 읽어와 정수형으로 바꾸고 값만큼 프로세스 정보 입력 테이블의 행을 생성한다.

```
def Creation_event(self):   #실행 버튼 커맨드 함수
    self.Time_slice = int(self.T_slice.text())      #입력값 타임슬라이스 변수에 정수형 저장
    for row in range(self.Process_num):     #프로세스 개수만큼 행에 입력된 값 반복하여 읽어옴
        PID = (int(self.Process_Table.item(row, 0).text()))
        Arrive_T = (int(self.Process_Table.item(row, 1).text()))
        Service_T = (int(self.Process_Table.item(row, 2).text()))
        Priority = (float(self.Process_Table.item(row, 3).text()))
        self.process.append(Process(PID, Arrive_T, Service_T, Priority))    #읽어온 값으로 프로세스 생성
```

생성 버튼의 커맨드 함수이다. 타임 슬라이스 텍스트에 입력된 타임 슬라이스의 값을 읽어 정수형 변수로 저장하고, 각 열에 입력한 값들을 한 행씩 읽어 프로세스의 객체를 생성한다.

비선점형 알고리즘과 선점형 알고리즘의 작동방식이 크게 다르기 때문에 두 분류로 나누어 설명을 할 것이다.

```
PCB = self.init_P()      #프로세스에 필요한 초기화 호출
PCB.sort(key=lambda x: (x.Arrive_T, x.PID))      #도착순, 이름순으로 정렬
```

```
for i in range(self.P_num): #FCFS알고리즘 동작
```

비선점형 알고리즘에서는 다음과같이 PCB 변수에 값을 복사한 후에 도착시간을 기준으로 정렬하여 인덱스 번호를 사용하여 작업을 시뮬레이팅한다.

```
arrive = i + 1
while arrive < self.P_num and PCB[arrive].Arrive_T <= Exec_T:
    arrive += 1      #프로세스 실행동안 도착한 마지막 프로세스의 인덱스값 arrive

if arrive <= self.P_num:
    PCB = copy(self.SJF_sort(PCB, i+1, arrive))  #도착한 프로세스들 서비스 시간 순으로 정렬
```

```
def SJF_sort(self, orignal, first, last):    #리스트 일정 범위 서비스 시간 기준 오름차순 정렬
    new_list = orignal[first:last]
    new_list.sort(key=lambda x: (x.Service_T, x.Arrive_T, x.PID))
    orignal[first:last] = new_list
    return orignal
```

SJF 알고리즘에 사용되는 부분이다. 진행시간보다 도착시간이 빠른 프로세스, 즉 도착한 마지막 프로세스의 인덱스 번호를 알아내고 리스트에서 해당 번호까지의 프로세스를 서비스 시간을 기준으로 정렬한다.

```
ax.barh('p'+str(PCB[i].PID), left = Exec_T, width = PCB[i].Service_T, color = 'gray')
Exec_T += PCB[i].Service_T
```

작업과정을 그리는 동작은 위와 같이 프로세스가 처음 CPU 에 도착한 시간(Exec_T)을 기준으로 해당 CPU 의 서비스 시간을 만큼을 그래프로 그린다. 그리고 서비스 시간을 작업진행 시간에 더해준다.

```
arrive = i + 1
while arrive < self.P_num and PCB[arrive].Arrive_T <= Exec_T:     #도착한 프로세스를 알아냄
    Wait_T = Exec_T - PCB[arrive].Arrive_T   #도착한 프로세스 대기시간 계산
    PCB[arrive].Priority = self.HRN_P(Wait_T, PCB[arrive].Service_T) #우선순위 계산
    arrive += 1

if i + 1 < self.P_num and PCB[i + 1].Arrive_T <= Exec_T:
    PCB = self.Descending_Psort(PCB, i+1, arrive)    #도착한 프로세스들 우선순위 내림차순으로 정렬
```

```
def HRN_P(self, wait_T, serv_T):    #우선순위 계산
    return (wait_T + serv_T)/serv_T
def Descending_Psort(self, orignal, first, last):    #리스트 일정 범위 우선순위로 내림차순 정렬
    new_list = orignal[first:last]
    new_list.sort(key=lambda x: (-x.Priority))
    orignal[first:last] = new_list
    return orignal
```

HRN 알고리즘 부분이다. HRN 알고리즘은 대기시간과 서비스 시간으로 우선순위를 계산하고 이 우선순위로 CPU 를 할당한다. 따라서 다음과 같이 도착한 프로세스들의 우선순위를 다음과 같이 함수로 계산한 값으로 매기고 우선순위의 내림차순(Descending)으로 정렬한다.

```
if i + 1 < self.P_num and PCB[i + 1].Arrive_T <= Exec_T:
    PCB = self.Ascending_Psort(PCB, i+1, arrive)  #우선순위 오름차순 정렬
```

비선점형 우선순위 알고리즘은 프로세스에 매겨진 우선순위를 기준으로 실행하기 때문에 처음에 입력한 우선순위의 값을 기준으로 오름차순(Ascending)으로 정렬한다.

비선점형 알고리즘은 위와 같이 동작하고 다음은 선점형 알고리즘의 설명이다.

```
process = self.init_P()
process.sort(key=lambda x: x.Arrive_T)

PCB = []
PCB.append(process[0])  #대기 큐에 첫 프로세스 도착
```

선점형 알고리즘은 Ready Queue 와 같이 구현하였다. 프로세스의 리스트를 가져오고 도착 순서로 정렬을 시킨다. 그리고 맨 처음으로 도착한 프로세스를 PCB 리스트에 삽입한다.

```
total_T = 0
for i in range(self.P_num):
    total_T += process[i].Service_T
```

모든 프로세스의 서비스 시간을 합하여 총 서비스 시간을 계산했다. 선점형 프로세스는 총 서비스 시간이 0 이 될 때까지 반복하여 작업을 완료한다.

```
if PCB[pre].First:
    PCB[pre].Resp_T = Exec_T - PCB[pre].Arrive_T    #CPU에 처음 올라온 프로세스 응답시간 계산
    PCB[pre].Wait_T += Exec_T - PCB[pre].Arrive_T   #첫 대기시간 계산
    PCB[pre].First = False    #처음이 False(아니다)

elif not PCB[pre].First:  #처음이 아니므로 전에 타임아웃 된 시간으로 대기시간 계산
    PCB[pre].Wait_T += Exec_T - PCB[pre].Stop_T
```

선점형 알고리즘은 타임아웃이 일어나므로 First 라는 bool 멤버변수로 첫 실행인지 확인하여 대기시간과 응답시간을 계속하여 계산한다. pre 변수는 0 이다. Ready Queue 는 문맥교환이 일어나서 다음에 작업할 프로세스를 맨 처음으로 가져오기 때문에 리스트의 맨 처음 요소만 건든다.

```
if PCB[pre].Service_T <= self.T_slice:    #남은 서비스시간이 타임슬라이스보다 작다면
    ax.barh('p'+str(PCB[pre].PID), left = Exec_T, width = PCB[pre].Service_T, color = 'gray')
    Exec_T += PCB[pre].Service_T
    total_T -= PCB[pre].Service_T     #남은 시간으로 실행시간 누적, 남은 실행시간 차감
    PCB[pre].Service_T = 0    #프로세스의 남은 서비스시간 0
    PCB[pre].Return_T = Exec_T - PCB[pre].Arrive_T
else:       #서비스시간이 타임슬라이스보다 크다면
    ax.barh('p'+str(PCB[pre].PID), left = Exec_T, width = self.T_slice, color = 'gray')
    Exec_T += self.T_slice
    total_T -= self.T_slice     #타임 슬라이스로 실행시간 누적, 남은 실행시간 차감
    PCB[pre].Stop_T = Exec_T      #타임아웃 된 시간
    PCB[pre].Service_T -= self.T_slice    #프로세스의 남은 서비스시간 - 타임 슬라이스
```

라운드 로빈과 SRT 알고리즘은 타임 슬라이스를 사용하여 작업하기 때문에 남은 서비스 시간이 타임 슬라이스보다 크다면 타임 슬라이스로 남은 총 작업시간을 빼고, 현재 진행 시간을 더하고, 해당 프로세스의 서비스 시간을 뺀다. 타임 슬라이스보다 서비스 시간이 작다면 남은 서비스 시간으로 계산을 해준다.

```
while used < self.P_num and process[used].Arrive_T <= Exec_T:
    PCB.append(process[used])   #도착한 프로세스 대기 큐에 삽입
    used += 1   #사용된 프로세스 개수(사용될 프로세스 인덱스)
```

```
if not PCB[i].Service_T:    #서비스 시간이 끝난 경우 해당 프로세스 제거
    del PCB[i]  #프로세스 반환
```

```
elif len(PCB) > 1 and PCB[pre].Service_T:  #남아 있다면 문맥교환
    PCB.append(PCB[pre])    #가장 뒤로 이동
    del PCB[pre]

if used < self.P_num and not len(PCB):
    PCB.append(process[used])   #대기큐가 비었다면, 실행된 시간보다 도착시간이 늦는 프로세스를 가져옴
    used += 1
```

위의 부분은 라운드 로빈의 프로세스 도착과 문맥교환시의 동작 부분이다. 맨 위의 부분은 현재 작업 진행시간보다 도착시간이 빠르거나 같은 프로세스가 있다면 Ready Queue 인 PCB 리스트에 삽입한다.

라운드 로빈의 Ready Queue 는 타임아웃이 일어난다면 현재 프로세스를 맨 뒤로 옮긴 뒤, 큐의 다음 프로세스를 작업하는 원형 큐이다. 이것은 두번째 부분의 동작과 같다.

만약 프로세스 리스트의 다음 프로세스 도착시간이 작업진행 시간보다 늦다면 Ready Queue 가 비기 때문에 다음 프로세스를 가져온다.

SRT 알고리즘은 라운드 로빈과 다르다. 무조건 적으로 문맥교환이 일어나지 않고, 남은 서비스 시간이 적은 순으로 작동하기 때문에 아래와 같이 현재 Ready Queue 에 들어온 프로세스들을 서비스 시간의 오름차순으로 정렬하였다.

```
PCB.sort(key=lambda x: (x.Service_T))   #남은 서비스 시간으로 정렬
```

선점형 우선순위 알고리즘은 서비스 시간을 1 초씩 감소시키는 방식으로 구현했다.

```
while not change and PCB[pre].Service_T:   #서비스 시간이 남았고, 문맥교환 불필요시에 반복
    ax.barh('p'+str(PCB[pre].PID), left = Exec_T, width = 1, color = 'gray')

    PCB[pre].Service_T -= 1     #서비스 시간 1초 진행
    Exec_T += 1
    total_T -= 1

    while used < self.P_num and process[used].Arrive_T == Exec_T:
        PCB.append(process[used])   #1초가 지났을때 도착한 프로세스 추가
        if PCB[pre].Service_T and process[used].Priority < PCB[pre].Priority:
            PCB[pre].Stop_T = Exec_T    #새로운 PCB가 실행중인 PCB보다 우선순위가 큰(숫자가 작은)경우 문맥교환 발생
            change = True   #문맥교환의 필요를 표시
        used += 1
```

서비스 시간과 총 실행시간을 -1, 작업 진행시간을 +1 해주며 그래프를 그리고 프로세스 리스트에서 도착한 프로세스가 있는지 계속하여 확인하고 Ready Queue 에 넣어준다. 그리고 새롭게 들어온 프로세스의 우선순위가 더 크다면 문맥교환의 필요를 알리는 bool 자료형인 change 변수를 True 로 바꾸어 작업을 멈추게 한다.

```
PCB.sort(key=lambda x: (x.Priority, x.Arrive_T))    #우선순위 오름차순 정렬
```

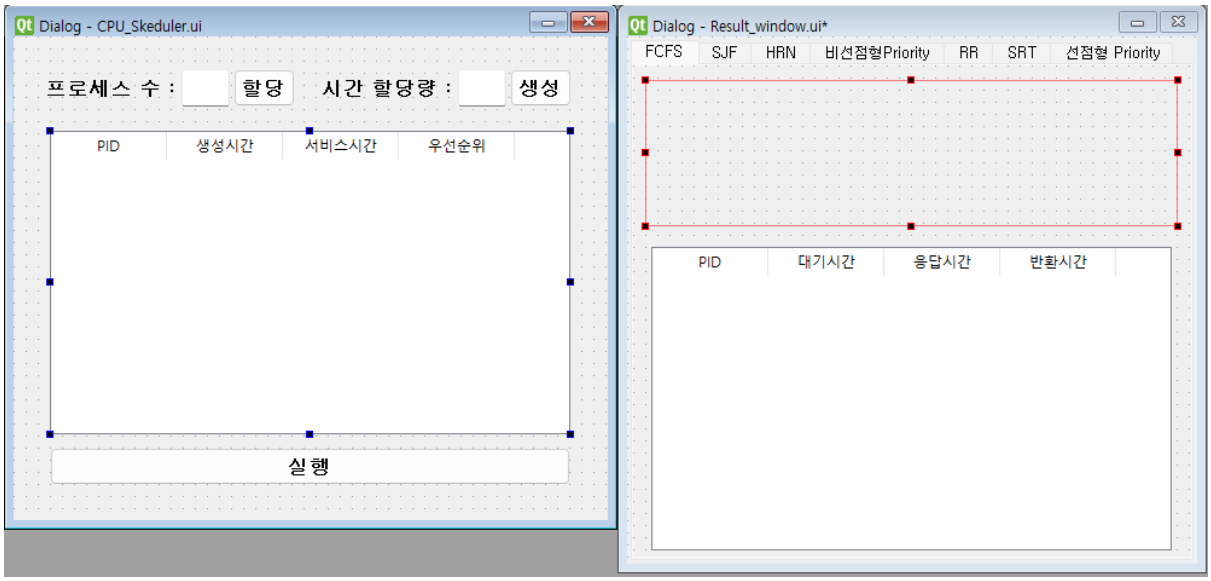
작업을 멈추고 우선순위로 Ready Queue 를 정렬하는 방식으로 문맥교환이 일어난다.

```
def empty_T(self, ax, empty_T): #프로세스 도착 전, 빈 시간 그래프
    if empty_T:
        ax.barh(str('Empty'), left = 0, width = empty_T, color = 'w')
    return empty_T
```

```
Exec_T = self.empty_T(ax, PCB[0].Arrive_T)
```

모든 알고리즘이 공통적으로 실행하는 함수이다. 프로세스 리스트의 첫 요소의 도착시간을 확인하여 0 이 아니라면 해당 시간을 그래프로 그리고 Exec_T 에 반환한다. 비어 있는 시간을 흰색 그래프로 그려 내기 위해 동작하는 함수이다.

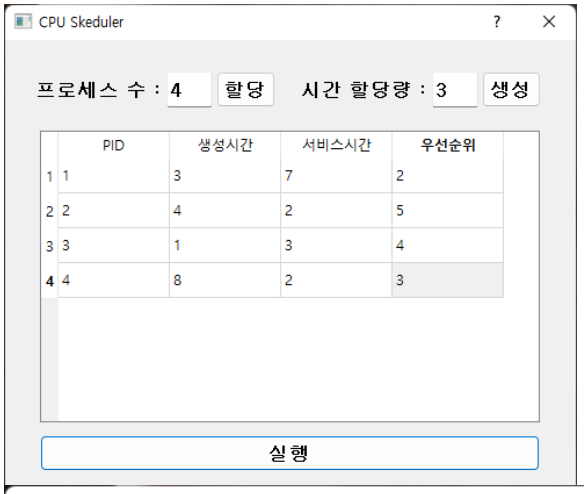
5. UI 설계



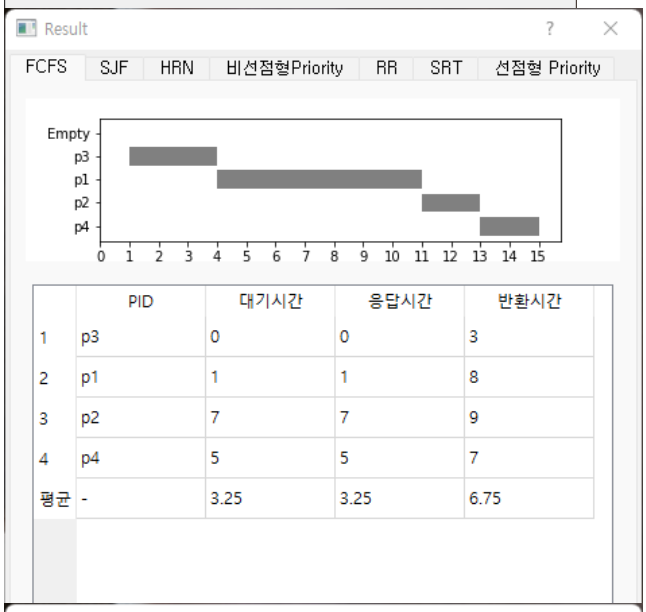
첫번째 윈도우는 왼쪽이다. 프로세스 수를 입력하고 할당을 누르면 해당 값만큼 프로세스 값 입력 테이블의 행이 생성된다. 생성된 테이블에 들어갈 수 있는 값은 정수형 값 밖에 없다. 값을 입력하고 시간 할당량의 텍스트 박스에 값을 입력하고 생성을 누르면 프로세스의 객체 리스트와 시간 할당량이 저장된다. 마지막으로 실행을 누르면 두번째 윈도우가 실행된다.

두번째 윈도우는 결과창이다. 맨 위에 각 프로세스의 이름이 적혀 있고, 원하는 알고리즘의 탭을 눌러 각 알고리즘의 시뮬레이팅 결과를 볼 수 있다. 탭들은 전부 그래프가 그려지는 박스 레이아웃과 테이블 위젯으로 구성 돼있다. 박스 레이아웃은 간트차트가 출력되고, 테이블 위젯에는 각 프로세스의 대기시간, 응답시간, 반환시간과 각각의 평균값이 출력된다.

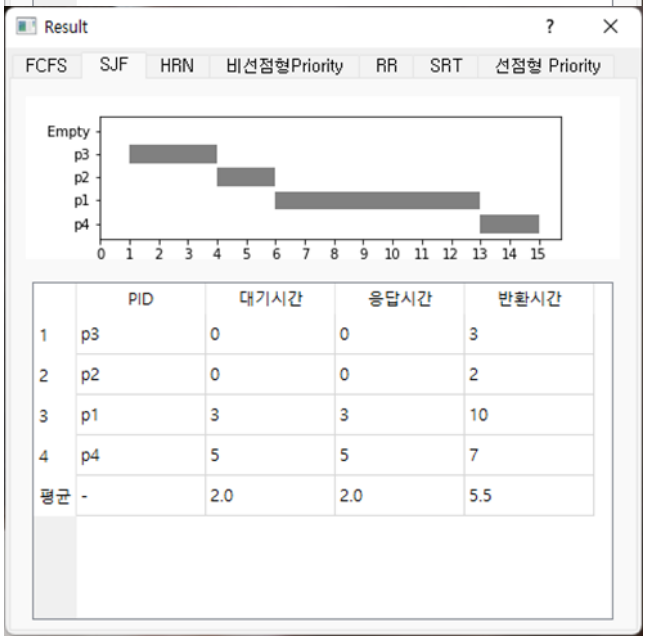
6. 실행결과



첫 실행 화면이다. 프로세스 수를 입력한 후에 생성된 테이블에 프로세스의 정보를 넣고 시간 할당량 값을 넣어준 뒤에 생성버튼을 눌러 프로세스를 생성해 준 후에 실행을 눌러 작업을 진행시켰다.

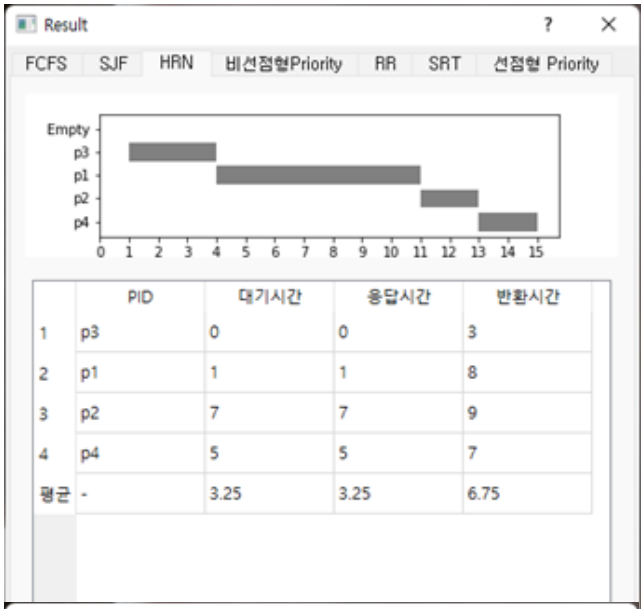


FCFS 스케줄링 실행 결과이다.
첫 프로세스의 도착시간이 1 이므로 1 초의 빈 시간이 흰색으로 그려졌다.
처음으로 도착한 프로세스 p3 가 일괄작업을 끝내고 다음으로 도착한 p1, p2, p4 순서대로 작업이 끝나게 되었다.



SJF 스케줄링의 실행 결과이다.

가장 먼저 도착한 p3 의 실행 도중 도착시간이 3 인 p1 과 4 인 p2 가 도착하였다.
p3 이 작업을 마치고 문맥교환이 발생하는데, p1 의 서비스 시간이 7 이고 p2 가 2 이므로 p2 가 먼저 작업을 진행하게 된다.
그리고 이어서 p1 이 작업중에 8 초에 도착한 p4 가 p1 이 작업을 마친 13 초에 CPU 에 올라와 작업을 진행하게 된다.



	PID	대기시간	응답시간	반환시간
1	p3	0	0	3
2	p1	1	1	8
3	p2	7	7	9
4	p4	5	5	7
평균	-	3.25	3.25	6.75

다음으로 HRN 스케줄링이다.
마찬가지로 p3 의 작업도중 p1 과 p2 가 도착한다. 둘의 우선순위를 계산한다.

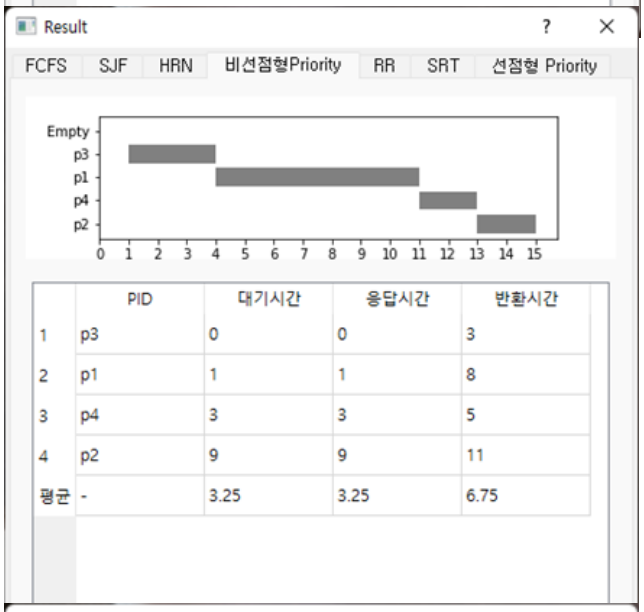
p1 => (1(대기) + 7(서비스)) / 7 = 1.14
p2 => (0(대기) + 2(서비스)) / 2 = 1

우선순위가 p1>p2 이므로 p1 이 실행된다.

p1 의 작업동안 p4 가 도착하게 된다. p2 와 p4 의 우선순위를 비교해야 한다.

p2 => ((11-4)(대기) + 2(서비스)) / 2 = 4.5
p4 => ((11-8)(대기) + 2(서비스)) / 7 = 1.5

p2>p4 이므로 p2->p4 로 작업을 한다.



	PID	대기시간	응답시간	반환시간
1	p3	0	0	3
2	p1	1	1	8
3	p4	3	3	5
4	p2	9	9	11
평균	-	3.25	3.25	6.75

비선점형 우선순위 스케줄링이다.
처음으로 도착한 p3 가 작업을 마친다.
작업도중 도착한 p1 과 p2 의 우선순위를 비교한다.

p1(2) > p2(5) (우선순위 값이 작을수록 더 우선순위가 높음

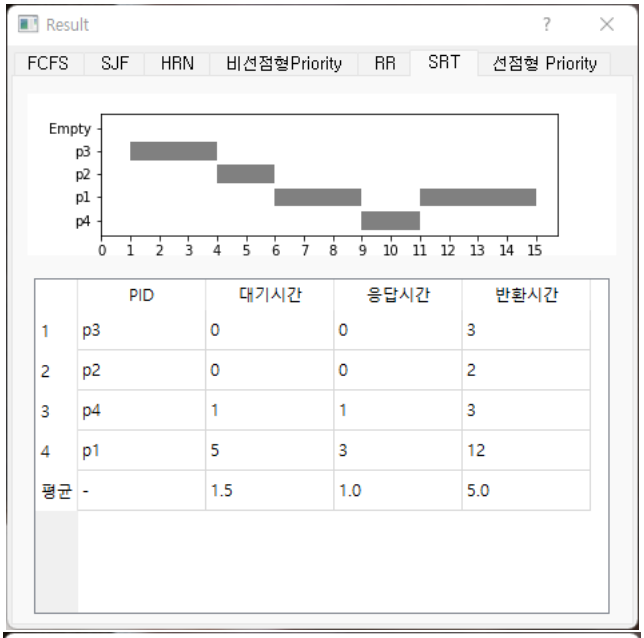
따라서 p1 이 작업을 마치고 새로 들어온 p4 와 p2 를 비교한다.

p2(5) > p4(3)

따라서 p2->p4 순서대로 작업을 마치게 된다.

	PID	대기시간	응답시간	반환시간
1	p3	0	0	3
2	p2	3	3	5
3	p4	4	4	6
4	p1	5	1	12
평균	-	3.0	2.0	6.5

라운드 로빈 스케줄링이다. 처음으로 도착한 p3 이 3 초(타임 슬라이스)동안 작업을 끝내고 반환된다.
p1(3)와 p2(3)이 들어왔다. 들어온 순서대로 작업을 진행한다.
p1 이 3 초의 작업을 마치고 큐의 맨 끝으로 이동한다.
다음 프로세스인 p2 가 2 초의 서비스 시간을 끝내므로 작업이 9 초에 반환된다.
그사이 p4(8)가 도착 큐에 들어온다.
큐의 맨 앞인 p1 이 3 초의 작업을 진행하고 큐의 맨 끝으로 이동한다.
다음 프로세스인 p4 가 2 초의 서비스 시간을 끝내고 이어서 p1 이 작업을 끝낸다.



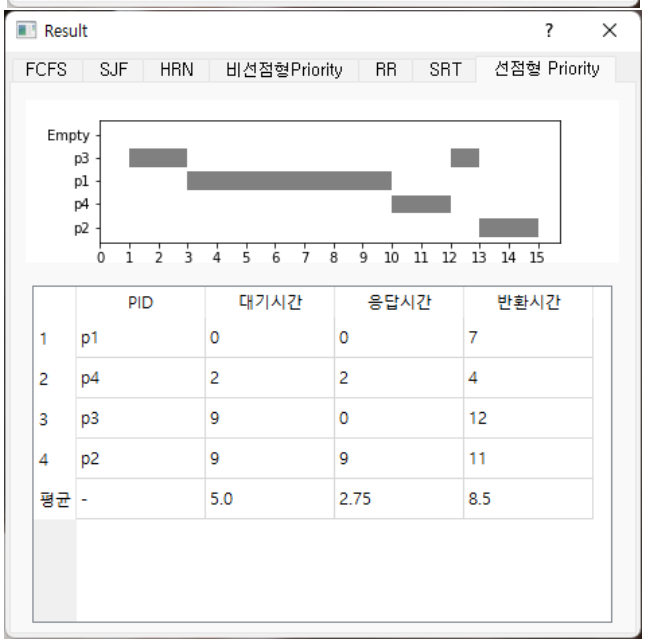
	PID	대기시간	응답시간	반환시간
1	p3	0	0	3
2	p2	0	0	2
3	p4	1	1	3
4	p1	5	3	12
평균	-	1.5	1.0	5.0

SRT 스케줄링이다. 처음 p3 이 3 초간의 작업을 진행한다.

p1(3), p2(4)가 도착하였다. p1 의 서비스 시간 7, p2 는 2 이므로 p2 가 큐의 맨 앞에 온다.

p2 의 작업이 끝나고(6), 큐에 남은 p1 만이 작업을 진행한다(3). 작업중에 p4 가 큐에 들어온다(8).

p1 의 남은 서비스 시간 4, p4 는 2 이므로 p4 가 먼저 실행되고 이어서 p1 이 실행되어 작업이 끝나게 된다.



	PID	대기시간	응답시간	반환시간
1	p1	0	0	7
2	p4	2	2	4
3	p3	9	0	12
4	p2	9	9	11
평균	-	5.0	2.75	8.5

선점형 우선순위 스케줄링이다. 처음 p3 이 작업을 진행하는데, 3 초에 p1 이 큐에 들어오게 된다.

p1(2)이 p3(4)의 우선순위보다 더 크므로 문맥교환이 일어난다.

4 초가 돼서 p2(5)가 새로 들어왔지만, p1 보다 우선순위가 더 작으므로 p1 의 작업이 계속된다.

8 초에 p4(3)가 도착하지만 우선순위가 더 작으므로 p1 의 작업이 계속되고, 서비스 시간이 끝나 p1 이 반환된다.

현재 큐에 남은 프로세스들끼리 우선순위를 비교하여 정렬을 한다. p4(3), p3(4), p2(5)의 순으로 정렬된다.

결국 새로운 프로세스는 없으므로 일괄작업의 형태로 작업이 끝난 모습이 보인다.

7. 느낀 점

파이썬을 배우지 않아 독학을 했는데, 시간이 예상보다 더 소모가 됐고, 급하게 배운 언어로 꽤 난이도 있는 코드를 구현하려다 보니 생각대로 되지 않아 많이 헤매게 되었다. 처음 계획했던 작업기간이 좀 부족했던 것 같다.

선점형 스케줄링의 작동 과정은 실제 스케줄러의 작업동작과 비슷하게 구현한 것 같다는 느낌이 든다. 하지만 비선점형 스케줄링의 경우에는 실행결과만을 고려해서 급하게 만들었다.

그래서 코드가 지저분하게 느껴진다. 그리고 반복되는 동작을 함수의 형태로 만들어야 의미가 클래스와 객체의 의미가 있는데, 조급해진 마음에 결과만 완벽하자 라는 생각으로 구현을 하게 되어 많이 아쉬운 코드가 만들어지게 되었다. 실제 스케줄러의 동작형태와 유사하게 만드는 게 더 시간이 덜 소모되고 코드가 유연해질 것 같다는 생각이 든다.