운영체제-Scheduling

1. 구현/평가의 목적
2. 사용 알고리즘 개요
3. 결과 및 결론

휴먼지능정보공학과

201910798

노희재

1. 구현/평가의 목적

스케줄링 알고리즘을 직접 구현하면서 각 알고리즘의 특성에 대해 깊이 있게 알게 되었다. 어떤 상황에 어떤 알고리즘이 최적인지, 알고리즘별 장점을 통합한 좋은 알고리즘이란 무엇인지, 운영체제의 입장에서 어떤 알고리즘을 선택할 것인가에 대한 고민을 했다. 각 알고리즘마다 좋은 특성을 가지고 있고 평가를 통해 결과로써 그것을 확인할 수 있었다.

2. 사용 알고리즘 개요

2-1. 프로세스 집합 생성

리스트나 딕셔너리 자료형을 사용할 수 있지만 Pandas의 DataFrame으로 프로세스의 집합을 만드는 것이 가독성이 좋다고 생각하였다.

```
In [ ]:  import pandas as pd

         data1 = {'Process': ['P1','P2','P3','P4','P5'],
                  'BurstTime': [10,29,3,7,12],
                  'Priority': [5,1,3,4,2]
                 }
         dataset1 = pd.DataFrame(data1)
         dataset1.set_index('Process', inplace=True)

         data2 = {'Process': ['P1','P2','P3','P4','P5'],
                  'BurstTime': [2,1,8,4,5],
                  'Priority': [2,1,4,2,3]
                 }
         dataset2 = pd.DataFrame(data2)
         dataset2.set_index('Process', inplace=True)
```

1. Dataset1 (프로세스 집합)

Process	BurstTime	Priority
P1	10	5
P2	29	1
P3	3	3
P4	7	4
P5	12	2

2. Dataset2 (프로세스 집합)

Process	BurstTime	Priority
P1	2	2
P2	1	1
P3	8	4
P4	4	2
P5	5	3

2-2. FCFS 구현

```
def FCFS(dataset):
    avg_WT=0
    WT = [0 for _ in range(len(dataset))]
    BT = [i for i in dataset['BurstTime']]
    time=0

    for i in range(len(dataset)):
        print("|", " "*BT[i], dataset.index[i], end=' ')

    print("|")
    print(0, end=" ")

    for i in range(len(dataset)):
        time += BT[i]
        WT[i] = time - BT[i]
        print(" "*(BT[i]+2), time, end=" ")

    print("₩n")
    for i in range(len(dataset)):
        print("Waiting Time P{0}: {1}".format(i+1, WT[i]))
        avg_WT += WT[i]

    print("₩n")
    print("Average Waiting Time: {}".format(avg_WT/len(WT)))
```

변수설정

```
avg_WT=0
WT = [0 for _ in range(len(dataset))]
BT = [i for i in dataset['BurstTime']]
time=0
```

avg_WT 는 평균대기시간을 저장하기 위한 배열이다.

WT[] 는 각 프로세스별 대기시간을 저장하기 위한 배열이다. 모든 값을 0으로 초기화하였다.

BT[] 는 각 프로세스별 버스트타임을 저장하는 배열이다.

time 은 지나간 시간을 표기하고 저장하는 변수로 gantt차트 밑에 시간을 표기하기 위한 변수이다.

Gantt 차트를 그리기 위한 반복문

```
for i in range(len(dataset)):
    print("|", " "*BT[i], dataset.index[i], end=' ')

print("|")
```

len(dataset) 으로 프로세스 집합의 길이만큼 반복하면서 gantt 차트를 출력한다. 버스트타임만큼 공백을 생성하고 프로세스명을 표기한다. 파이썬의 print문은 "end='₩n'을 생략하고 있음으로 줄바꿈이 되지 않게 하기 위해 " end=' ' " 를 써준다.

Gantt 차트 밑에 시간을 표기하고 WT[]에 대기시간을 저장하는 반복문

```
print(0, end=" ")

for i in range(len(dataset)):
    time += BT[i]
    WT[i] = time - BT[i]
    print(" "*(BT[i]+2), time, end=" ")
```

처음에 시간시간 0을 써주고 줄바꿈을 하지 않는다. 반복문을 돌면서 time에 버스트타임을 더하고 대기시간에 (time – BT[i])를 한다. 지난 시간에서 자신의 버스트타임을 뺀 것이 대기시간이기 때문이다. 그리고 마찬가지로 공백을 생성하고 time을 표기하는데 공백 생성 시 gantt차트에서 프로세스명을 표기할 때 공간을 차지하므로 줄을 맞춰주기 위해 버스트타임+2 만큼 공백을 생성한다.

대기시간을 출력하는 반복문

```
for i in range(len(dataset)):
    print("Waiting Time P{0}: {1}".format(i+1, WT[i]))
    avg_WT += WT[i]

print("₩n")
print("Average Waiting Time: {}".format(avg_WT/len(WT)))
```

format을 사용하여 프로세스별 대기시간을 출력한다. avg_WT에 대기시간을 합한다.

avg_WT/len(WT) 를 하여 평균대기시간을 출력한다.

결과

1번 프로세스 집합

```
FCFS(dataset1)

 |            P1 |                                P2 |      P3 |          P4 |              P5 |
 0              10                                  39        42            49                61

 Waiting Time P1: 0
 Waiting Time P2: 10
 Waiting Time P3: 39
 Waiting Time P4: 42
 Waiting Time P5: 49


 Average Waiting Time: 28.0
```

2번 프로세스 집합

```
FCFS(dataset2)

 |    P1 |   P2 |          P3 |       P4 |        P5 |
 0       2      3            11         15          20

 Waiting Time P1: 0
 Waiting Time P2: 2
 Waiting Time P3: 3
 Waiting Time P4: 11
 Waiting Time P5: 15


 Average Waiting Time: 6.2
```

2-3. Round Robin 구현

```
def RR(dataset, quantum):
    quantum = quantum
    avg_WT = 0
    WT = [0 for _ in range(len(dataset))]
    BT = [i for i in dataset['BurstTime']]
    time=0
    i=0
    while(1):
        if(BT[i]== -100):
            pass
        elif(BT[i]>quantum):
            print("|", " "*quantum, dataset.index[i], end=" ")
            BT[i] -= quantum
        else:
            print("|", " "*BT[i], dataset.index[i], end=" ")
            BT[i] = -100
        i+=1
        i = i%(len(dataset))
        if(all([(BT[i]==-100) for i in range(len(BT))])):
            break

    print("|")
    print(0, end=" ")
    BT = [i for i in dataset['BurstTime']]
    i=0
```

```
    while(1):
        if(BT[i]== -100):
            pass
        elif(BT[i]>quantum):
            time += quantum
            WT[i] -= quantum
            print(" "*(quantum+2), time, end=" ")
            BT[i] -= quantum
        else:
            time += BT[i]
            WT[i] += time - BT[i]
            print(" "*(BT[i]+2), time, end=" ")
            BT[i] = -100
        i+=1
        i = i%(len(dataset))
        if(all([(BT[i]==-100) for i in range(len(BT))])):
            break

    print("₩n")
    for i in range(len(WT)):
        print("Waiting Time P{0}: {1}".format(i+1,WT[i]))
        avg_WT += WT[i]

    print("₩n")
    print("Average Waiting Time: {}".format(avg_WT/len(BT)))
```

변수설정

```
RR(dataset, quantum):
quantum = quantum
avg_WT = 0
WT = [0 for _ in range(len(dataset))]
BT = [i for i in dataset['BurstTime']]
time=0
i=0
```

RR함수의 매개변수로 quantum을 받아서 quantum 직접 설정할 수 있도록 구현했다.

avg_WT 는 평균대기시간을 저장하기 위한 배열이다.

WT[] 는 각 프로세스별 대기시간을 저장하기 위한 배열이다. 모든 값을 0으로 초기화하였다.

BT[] 는 각 프로세스별 버스트타임을 저장하는 배열이다.

time 은 지나간 시간을 표기하고 저장하는 변수로 gantt차트 밑에 시간을 표기하기 위한 변수이다. i는 반복문 안에서 각 프로세스를 가리키고 배열에 접근하기 위한 인덱스 변수이다.

Gantt 차트를 그리기 위한 반복문

```
while(1):
    if(BT[i]== -100):
        pass
    elif(BT[i]>quantum):
        print("|", " "*quantum, dataset.index[i], end=" ")
        BT[i] -= quantum
    else:
        print("|", " "*BT[i], dataset.index[i], end=" ")
        BT[i] = -100
    i+=1
    i = i%(len(dataset))
    if(all([(BT[i]==-100) for i in range(len(BT))])):
        break

print("|")
```

무한루프를 돌면서 모든 프로세스가 소진될 때까지 반복한다. 각 프로세스가 모두 소진되면 버스트타임 배열에 -100을 넣어 모두 소진되었음을 표기했다. 첫 번째 if 조건문에 -100일 경우에 pass명령어를 넣어 다음 조건으로 넘어가는 것을 방지했다. elif 조건인 버스트타임이 quantum보

다 크다면 quantum만큼 공백을 생성하고 프로세스명을 표기한다. 그리고 버스트타임에서 quantum만큼 빼준다. else조건은 버스트타임이 quantum보다 작거나 같은 경우로 버스트타임 만큼의 공백을 생성하고 프로세스명을 표기한다. 그리고 버스트타임 배열에 -100을 넣어서 프로세스가 모두 소진되었음을 명시한다. i 인덱스를 하나 증가시키면서 다음 프로세스를 가리킬 수 있도록 한다. 모든 프로세스가 한 번씩 돌고 다시 첫 번재 프로세스를 가리키기 위해 모듈러 연산자를 사용하였다. 무한루프의 종료조건인 버스트타임 배열에 모든 값이 -100인 것을 확인하기 위해 all()을 사용하였다.

Gantt 차트 밑에 시간을 표기하고 WT[]에 대기시간을 저장하는 반복문

```
print(0, end=" ")
BT = [i for i in dataset['BurstTime']]
i=0
while(1):
    if(BT[i]== -100):
        pass
    elif(BT[i]>quantum):
        time += quantum
        WT[i] -= quantum
        print(" "*(quantum+2), time, end=" ")
        BT[i] -= quantum
    else:
        time += BT[i]
        WT[i] += time - BT[i]
        print(" "*(BT[i]+2), time, end=" ")
        BT[i] = -100
    i+=1
    i = i%(len(dataset))
    if(all([(BT[i]==-100) for i in range(len(BT))])):
        break
```

버스트타임 배열과 인덱스를 초기화해준다. 마찬가지로 무한루프를 돌고 종료조건은 버스트타임 배열의 모든 값이 -100일 때이다. elif조건인 버스트타임이 quantum보다 크다면 time에 quantum을 더하고 대기시간에서 quantum을 빼준다. 실행시간은 대기시간에 포함되지 않기 때문이다. quantum+2 만큼 공백을 채우고 지나간 시간(time)을 표기한다. else조건인 버스트타임이 quantum보다 작거나 같으면 time에 버스트타임을 더하고 대기시간에 time을 더하고 버스트타임을 빼준다. 마찬가지로 실행시간은 대기시간에 포함되지 않기 때문이다. 버스트타임+2 만큼 공백

을 채우고 time을 표기하고 버스트타임에 -100을 넣는다. 인덱스를 증가하고 인덱스 범위를 확인한 다음 종료조건을 확인한 뒤 루프를 돈다.

대기시간을 출력하는 반복문

```
for i in range(len(WT)):
    print("Waiting Time P{0}: {1}".format(i+1,WT[i]))
    avg_WT += WT[i]

print("₩n")
print("Average Waiting Time: {}".format(avg_WT/len(BT)))
```

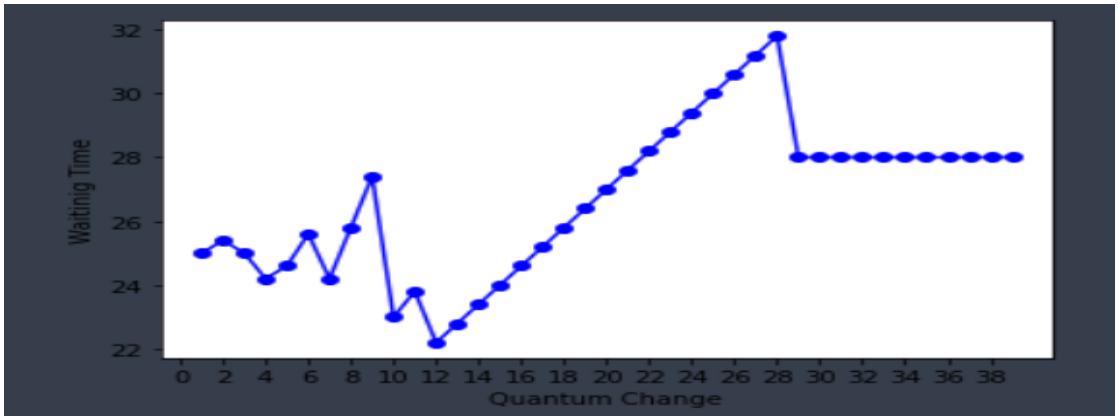
format을 사용하여 프로세스별 대기시간을 출력한다. avg_WT에 대기시간을 합한다.

avg_WT/len(WT) 를 하여 평균대기시간을 출력한다.

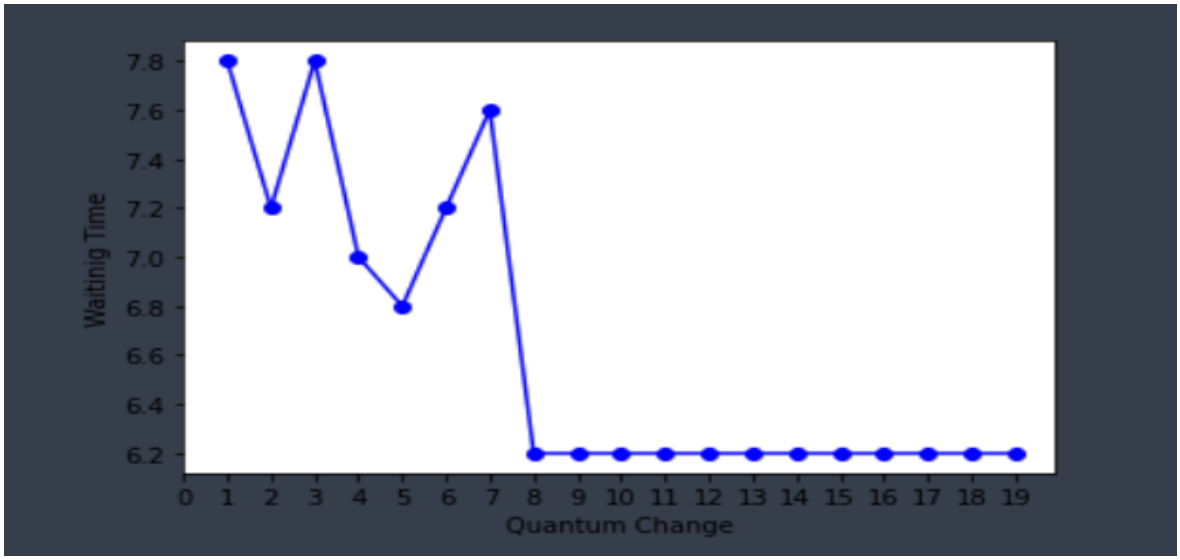
결과

```
from matplotlib import pyplot as plt

quan=[]
avg_WT=[]
for i in range(1,40):
    print("quantum이 {0}일 때 평균대기시간: {1}".format(i,avg_WT_RR(dataset1,i)))
    quan.append(i)
    avg_WT.insert(i,avg_WT_RR(dataset1,i))

plt.plot(quan, avg_WT,color='blue', marker='o', linestyle='solid')
plt.xticks([2*i for i in range(20)])
```

초기에 quantum 설정 시 프로세스 버스트타임의 평균으로 하면 좋을 것 같다고 생각했다. quantum의 변화에 따른 대기시간 변화를 알아보기 위해 위와 같은 코드를 작성해서 확인했다.



1번 프로세스 집합의 경우 버스트타임의 평균이 12.2이고 결과 또한 quantum이 12일 때 평균대기시간이 가장 짧은 것을 알 수 있다.



2번 프로세스 집합의 경우 버스트타임의 평균이 5이고 결과 또한 quantum이 5일 때 평균대기시간이 가장 짧은 것을 알 수 있다. 하지만 2번 프로세스 집합의 경우 RR방식을 사용하지 않고 FCFS방식(quantum이 가장 긴 버스트타임보다 큰 quantum을 사용하는 경우)이 평균대기시간이 더 짧은 것을 알 수 있다. 적절한 quantum을 설정한다면 RR방식이 FCFS방식보다는 항상 좋을 것이라 생각했지만 그렇지 않은 경우도 존재한다.

1번 프로세스 집합 결과(quantum=10)

```
RR(dataset1, 12)

|          P1 |          P2 |     P3 |          P4 |          P5 |          P2 |       P2 |
0            10            22       25            32            44            56         61

Waiting Time P1: 0
Waiting Time P2: 32
Waiting Time P3: 22
Waiting Time P4: 25
Waiting Time P5: 32


Average Waiting Time: 22.2
```

2번 프로세스 집합 결과(quantum=5)

```
RR(dataset2, 5)

|    P1 |   P2 |       P3 |      P4 |       P5 |     P3 |
0      2     3         8        12         17       20

Waiting Time P1: 0
Waiting Time P2: 2
Waiting Time P3: 12
Waiting Time P4: 8
Waiting Time P5: 12


Average Waiting Time: 6.8
```

2-4. Priority 구현

```
def Priority(dataset):
    avg_WT=0
    WT = [0 for _ in range(len(dataset))]
    sorted_set = dataset.sort_values(by=['Priority','BurstTime'])
    BT = [i for i in sorted_set['BurstTime']]

    for i in range(len(sorted_set)):
        print("|", " "*BT[i], sorted_set.index[i], end=' ')

    print("|")
    print(0, end=" ")
    time=0

    for i in range(len(sorted_set)):
        time += BT[i]
        WT[i] = time - BT[i]
        print(" "*(BT[i]+2), time, end=" ")

    print("₩n")
    for i in range(len(sorted_set)):
        print("Waiting Time {0}: {1}".format(sorted_set.index[i], WT[i]))
        avg_WT += WT[i]

    print("₩n")
    print("Average Waiting Time: {}".format(avg_WT/len(WT)))
```

변수설정

```
    avg_WT=0
    WT = [0 for _ in range(len(dataset))]
    sorted_set = dataset.sort_values(by=['Priority','BurstTime'])
    BT = [i for i in sorted_set['BurstTime']]
```

sorted_set에 우선순위를 기준으로 정렬된 DataFrame을 저장한다. 우선순위가 같을 경우 버스트 타임을 기준으로 정렬한다.

Gantt 차트 출력 및 밑에 시간을 적는 반복문, 대기시간을 출력하는 반복문

```
for i in range(len(sorted_set)):
    print("|", " "*BT[i], sorted_set.index[i], end=' ')

print("|")
print(0, end=" ")

time=0
for i in range(len(sorted_set)):
    time += BT[i]
    WT[i] = time - BT[i]
    print(" "*(BT[i]+2), time, end=" ")

print("₩n")
for i in range(len(sorted_set)):
    print("Waiting Time {0}: {1}".format(sorted_set.index[i], WT[i]))
    avg_WT += WT[i]

print("₩n")
print("Average Waiting Time: {}".format(avg_WT/len(WT)))
```

위 코드의 경우 기존 프로세스 집합인 dataset을 sorted_set으로 바꿔주면 FCFS와 같은 메커니즘으로 작동한다.

결과

1번 프로세스 집합

```
Priority(dataset1)

 |                          P2 |            P5 |     P3 |         P4 |           P1 |
 0                           29             41       44           51             61

 Waiting Time P2: 0
 Waiting Time P5: 29
 Waiting Time P3: 41
 Waiting Time P4: 44
 Waiting Time P1: 51


 Average Waiting Time: 33.0
```

2번 프로세스 집합

```
Priority(dataset2)

 |   P2 |    P1 |      P4 |       P5 |          P3 |
 0     1      3        7        12           20

 Waiting Time P2: 0
 Waiting Time P1: 1
 Waiting Time P4: 3
 Waiting Time P5: 7
 Waiting Time P3: 12


 Average Waiting Time: 4.6
```

2-5. SJF 구현

```
def SJF(dataset):
    avg_WT=0
    WT = [0 for _ in range(len(dataset))]
    sorted_set = dataset.sort_values(by=['BurstTime','Priority'])
    BT = [i for i in sorted_set['BurstTime']]

    for i in range(len(sorted_set)):
        print("|", " "*BT[i], sorted_set.index[i], end=' ')

    print("|")
    print(0, end=" ")
    time=0

    for i in range(len(sorted_set)):
        time += BT[i]
        WT[i] = time - BT[i]
        print(" "*(BT[i]+2), time, end=" ")

    print("₩n")
    for i in range(len(sorted_set)):
        print("Waiting Time {0}: {1}".format(sorted_set.index[i], WT[i]))
        avg_WT += WT[i]

    print("₩n")
    print("Average Waiting Time: {}".format(avg_WT/len(WT)))
```

위의 코드는 앞서 Priority알고리즘과 sorted_set의 정렬 방법만 다르고 같은 메커니즘으로 작동한다. 정렬 시 버스트타임으로 정렬하고 버스트타임이 같다면 우선순위를 기준으로 정렬한다.

결과

1번 프로세스 집합

```
SJF(dataset1)
```

```
  |     P3 |         P4 |           P1 |             P5 |                              P2 |
  0        3           10             20               32                              61

  Waiting Time P3: 0
  Waiting Time P4: 3
  Waiting Time P1: 10
  Waiting Time P5: 20
  Waiting Time P2: 32


  Average Waiting Time: 13.0
```

2번 프로세스 집합

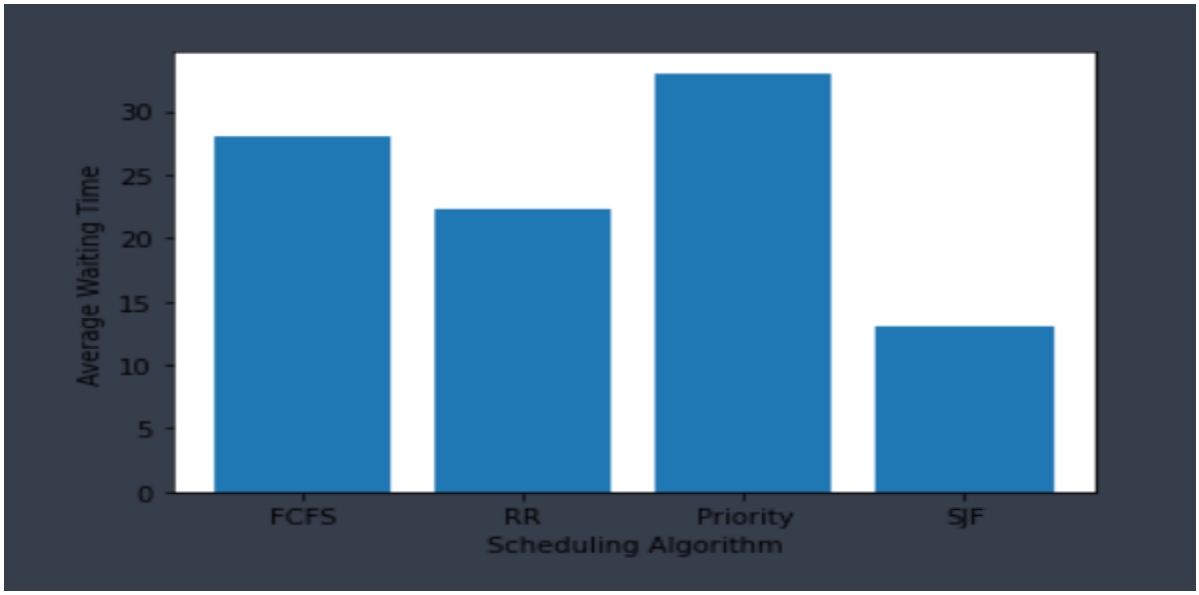
```
SJF(dataset2)

  |   P2 |    P1 |      P4 |       P5 |          P3 |
  0      1       3         7         12           20

  Waiting Time P2: 0
  Waiting Time P1: 1
  Waiting Time P4: 3
  Waiting Time P5: 7
  Waiting Time P3: 12


  Average Waiting Time: 4.6
```

3. 결과 및 결론

각 알고리즘 별로 평균대기시간을 반환하는 avg_WT_[AlgorName]() 함수(소스코드에 포함)를 만들고 바그래프를 출력해주는 BarPlot() 함수를 만들었다.

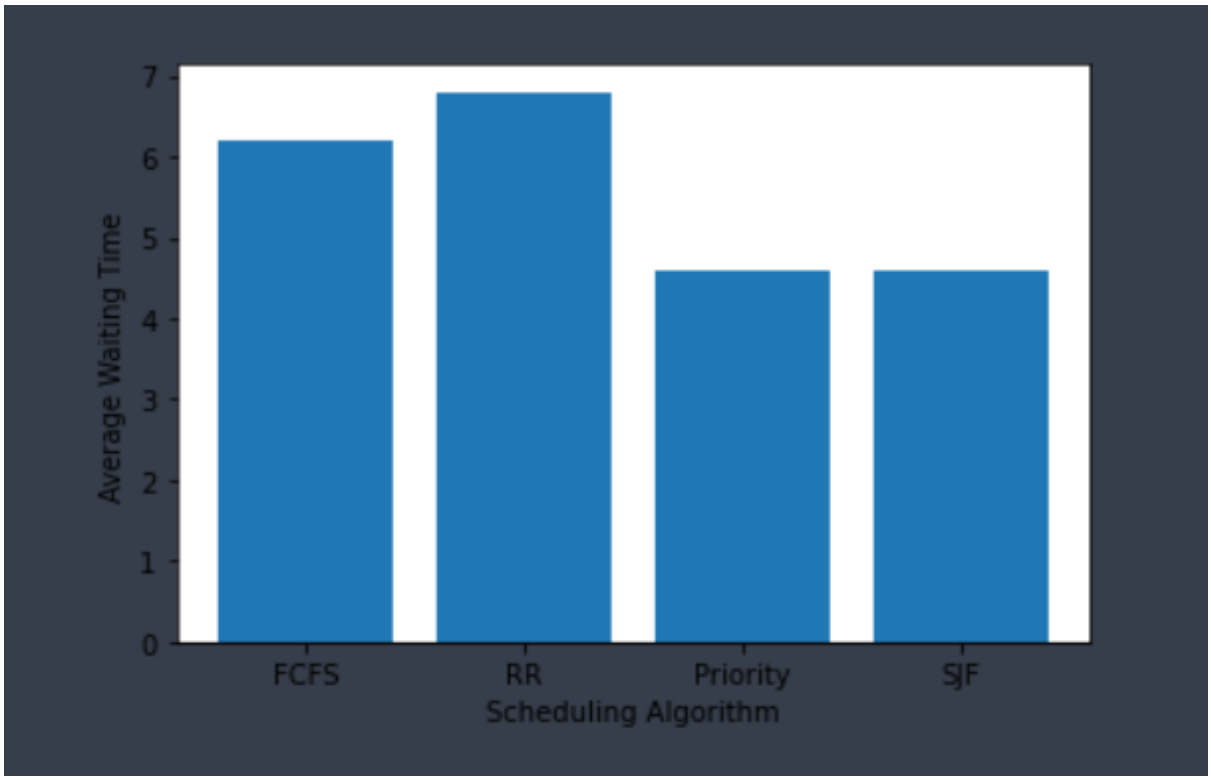
```
import matplotlib.pyplot as plt

def BarPlot(dataset):
    avg_WT = [avg_WT_FCFS(dataset), avg_WT_RR(dataset,12), avg_WT_Priority(dataset), avg_WT_SJF(dataset)]
    Schedule_Name = ['FCFS', 'RR', 'Priority', 'SJF']
    plt.bar(Schedule_Name,avg_WT)
    plt.xlabel("Scheduling Algorithm")
    plt.ylabel("Average Waiting Time")
    plt.show()
```

1번 프로세스 집합의 결과



1번 프로세스 집합의 경우 SJF알고리즘의 평균대기시간이 가장 짧고 Priority알고리즘의 평균대기시간이 가장 긴 것을 알 수 있다. 1번의 경우 우선순위에 따라 프로세스를 처리했을 때의 문제를 나타내고 있으며 RR이 FCFS보다 평균대기시간이 짧은 평균의 경우를 나타내고 있다고 볼 수 있다.

2번 프로세스 집합의 결과



2번 프로세스 집합의 경우 SJF와 Priority알고리즘의 평균대기시간이 가장 짧고 RR알고리즘의 평균대기시간이 가장 긴 것을 알 수 있다. 2번의 경우 운이 좋게 우선순위가 높은 것이 프로세스의 버스트타임도 짧아 Priority알고리즘에서도 좋은 성능을 보인다. RR알고리즘이 FCFS보다 평균대기시간이 긴 것을 보아 특수한 경우임을 알 수 있다.

결론

평균대기시간만을 고려한다면 위의 결과처럼 SJF알고리즘이 가장 짧은 대기시간을 갖는다는 것은 명확하다. 하지만 대부분의 작업에는 우선순위가 분명히 존재한다. 그렇기에 단순히 가장 짧은 작업을 먼저 하는 알고리즘은 옳지 않다. 우선순위만을 고려하는 것 또한 1번 프로세스의 결과를 통해 평균대기시간이 길어질 수 있음을 확인했다. 공정하다고 생각이 되는 FCFS알고리즘이지만 실제 컴퓨터 상에서는 우선적으로 처리되어야 하는 작업들이 있기 때문에 오류를 일으킬 수 있다. RR알고리즘은 먼저 온 프로세스를 고려하며 모든 프로세스에게 순서가 돌아오기 때문에 좋은 알고리즘이라 생각했다. 하지만 2번 프로세스의 결과를 보면 RR알고리즘의 평균대기시간이 가장 길다. 이처럼 가장 효율적이고 공정한 스케줄링 알고리즘을 찾는 것은 매우 어렵다. 나 또한 그렇다. 해야 할 과제, 친구들과의 만남, 가족들과의 시간, 아르바이트, 내가 하고 싶은 공부 등 수 많은 작업이 있고 나는 그것의 우선순위와 걸리는 시간 등을 고려해서 지금 무엇을 할 지 정한다. 사람도 스케줄링하는 것이 쉽지 않은데 컴퓨터 알고리즘이 많은 변수들을 모두 고려해서 하는 것은 어려운 일이다. 하지만 처리되야 하는 프로세스의 평균적인 버스트타임이나 우선순위(중요도)를 알고 있다면 그 상황에 맞는 스케줄링 알고리즘을 적용할 수 있다. 따라서 하나의 좋은 알고리즘이 아닌 각 상황에 맞춰 다양한 알고리즘을 택하는 것이 그것을 사용하는 운영체제에게 가장 좋은 방법이라고 생각한다.