

# OS Assignment3 결과 보고서

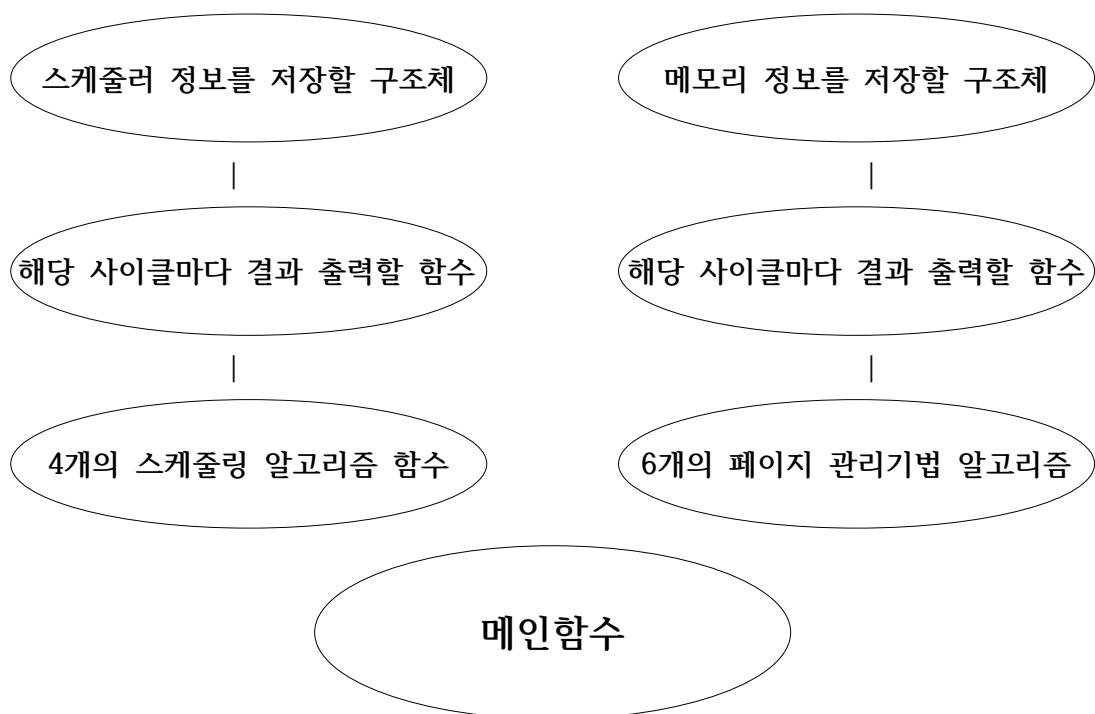
컴퓨터과학과  
2015147533 유현석

## 1. 작성한 프로그램의 동작 과정과 구현 방법

스케줄러와 메모리 관리 기법이 결합 된 시뮬레이터 만들기 프로젝트

### 1-1 main 프로그램

먼저 main에서 크게 3가지의 정보를 가져와야 합니다. 그것들은 바로 스케줄러 알고리즘과 메모리관리 알고리즘인 페이지 교체 알고리즘을 선택해줘야 합니다. 또한, 인풋 파일이 위치하는 디렉토리 및 출력 파일이 위치할 디렉토리의 정보가 필요합니다. 이러한 큰 3가지 정보를 받아와서 정해준 후 해당 알고리즘 함수를 불러와 완성 시켜 줍니다. 그리고 각각의 process의 정보를 저장할 구조체가 스케줄과 메모리에 각각 필요합니다. 또한, 그 중간에는 각각의 사이클마다 결과를 파일에 적어줄 각각의 함수 역시 필요합니다. 따라서 전체적인 구성은 아래의 그림과 같습니다.



## 1-2 스케줄러 프로그램

그다음은 스케줄러 프로그램을 구현하는데 기본적으로 들어가는 공통의 코드부터 설명하도록 하겠습니다. 공통적인 부분으로는 먼저 인풋으로 들어오는 각종 프로세스와 INPUT (I/O)에 대한 정보를 저장해줍니다. 그 후, 그 명령어가 프로세스를 불러오는 것인지 혹은 INPUT을 읽어오는지에 따라 구분 지어 앞서 말한 프로세스 정보를 가지고 있는 구조체에 저장해줍니다. 또한, 각각의 프로세스가 실행해야 하는 op, arg 정보 역시 vector로 하여금 다 읽어 저장해줍니다. 그 후에는 각각에 알고리즘에 맞는 while이 실행됩니다.

<각각의 process의 정보를 해당 구조체에 저장하는 과정>

```
for(int b=0;b<nun;b++){
    if(readprocess[b][2] == "-1"){
        ppid++;
        processif[ppid].pid=ppid;
        string readingprocessing=dir+readprocess[b][1];
        readfile.open(readingprocessing.c_str());
        processif[ppid].name=readprocess[b][1];

        readfile>>read;
        processif[ppid].time=stoi(readprocess[b][0]);
        starttime[ppid]=stoi(readprocess[b][0]);
        processif[ppid].linenum=stoi(read);
        processif[ppid].readline=0;
        processif[ppid].iowait=0;
        processif[ppid].sleeptime=0;
        processif[ppid].end=0;
        for(int a=0;a<(processif[ppid].linenum)*2;a++){
            readfile>>read;
            processif[ppid].alldocu.push_back(read);
        }
        readfile.close();
    } //when is the process
    else{
        iopid++;
        processio[iopid].name=readprocess[b][1];
        processio[iopid].time=stoi(readprocess[b][0]);
        processio[iopid].pid=stoi(readprocess[b][2]);
    } //when is the IO input
} //reading the all code file
```

if(readprocess[b][2] == "-1")의 의미는 해당 읽은 줄이 I/O인지 process인지 구분하는 과정입니다.

readfile.open(readingprocessing.c\_str())의 의미는 process명으로 된 파일을 열어 정보를 읽어오는 과정입니다.

또한 스케줄러 while문 안에는 크게 3가지 경우로 나눴습니다.

1. process가 돌아가고 있는 중

2. 실행되는 process가 없고, 새로운 runtime에 도달한 경우

3. 실행되는 process가 없고, 새로운 runtime에 도달하지 않는 경우

또한 3가지 경우를 3가지 경우 중에서 실행하기 전에 먼저 체크를 해주었습니다.

1. process가 런닝타임에 도달한 경우

-> 해당 process를 runQ에 넣고 실행되고 있는 process가 없다면 위에 2번 경우 있다면 1번 경우에 해당됩니다.

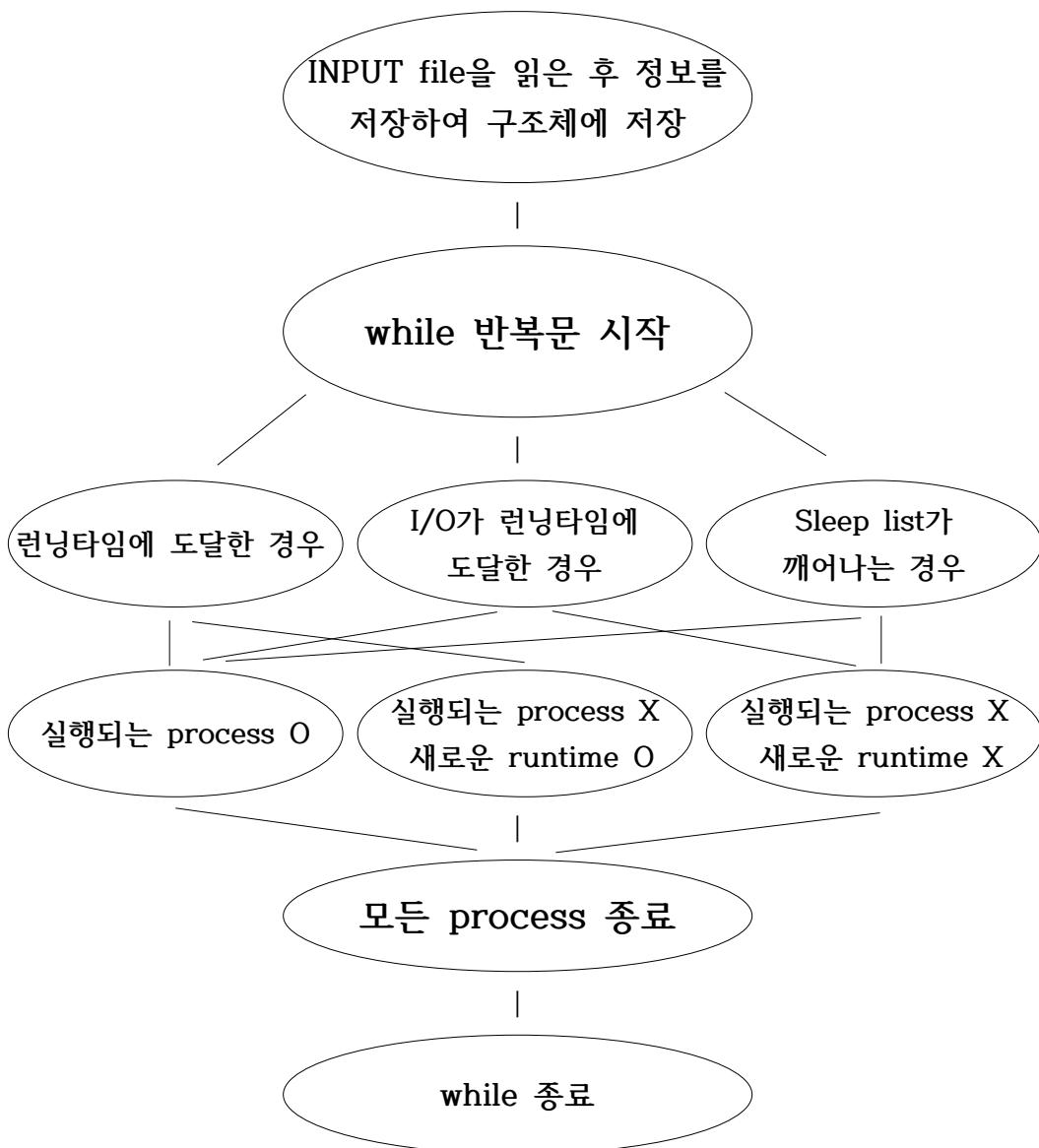
## 2. I/O가 런닝타임에 도달한 경우

-> I/O wait에 있는 process를 깨워준 후 runQ에 넣어줍니다. 이 경우에는 2번에 해당하지 않고 3번에 해당합니다.

## 3. Sleep list에 있는 process가 깨어나는 경우

-> Sleep list에 있는 process를 runQ로 옮겨줍니다. 이 경우 역시 2번에 해당하지 않고 3번에 해당 됩니다.

앞선 3가지 경우인 (process가 돌아가고 있는 중, 실행되는 process가 없고, 새로운 runtime에 도달한 경우, 실행되는 process가 없고, 새로운 runtime에 도달하지 않는 경우) 가 끝났다면 이제 존재하는 모든 process가 끝났는지 확인을 한 후 모든 process가 끝난 것이 맞다면 while문을 종료시켜줍니다.



## ① FCFS 알고리즘

FSFS 알고리즘이란 run queue에 들어오는 순서대로 CPU가 선택하는 알고리즘입니다. 그렇다면 이 알고리즘을 완성하기 위해서는 순서대로 run Queue에 들어가는 것이 중요합니다. run Queue에만 순서대로 잘 들어가면 앞에서부터 차근차근히 실행시켜 주면 됩니다. 위에 3가지 경우가 존재할 때 각각 어떻게 동작하는지 알아보도록 하겠습니다.

### 1. process가 돌아가고 있는 중

-> 새로운 process가 runtime에 도달한 여부와 상관없이 기존에 돌아가고 있는 process를 계속 진행 시켜주면 됩니다. 만약 진행하던 process가 모든 명령어를 읽거나 I/O wait, sleep이 되는 경우가 아니면 지속이 되고 그 경우라면 I/O wait는 I/O queue에 sleep은 sleeplist에 저장을 해주고 process를 종료시켜줍니다.

### 2. 실행되는 process가 없고, 새로운 runtime에 도달한 경우

-> 새로운 runtime에 도달한 경우라면 무조건 run Queue에 하나 이상의 process 가 존재한다는 의미이기 때문에 run Queue에 있는 맨 앞 첫 번째 process를 실행 시켜줍니다. 이 역시 마찬가지로 모든 명령어를 읽거나 I/O wait, sleep이 되는 경우가 아니면 지속이 되고 그 경우라면 I/O wait는 I/O queue에 sleep은 sleeplist에 저장을 해주고 process를 종료시켜줍니다.

### 3. 실행되는 process가 없고, 새로운 runtime에 도달하지 않는 경우

-> 새로운 runtime에 어떠한 새로운 process가 도달하지 않았기 때문에 run Queue에 process가 존재하는지 아닌지 알 수 없습니다. 그렇기 때문에 먼저 2가지 경우로 나누어 줍니다.

#### 1. run Queue에 없는 경우

-> 이 경우에는 실행시킬 수 있는 process가 존재하지 않기 때문에 NO-OP를 실행 시켜 주면 됩니다.

#### 2. run Queue에 있는 경우

-> run Queue에 process가 존재한다면 맨 앞 첫 번째 process를 실행시켜줍니다. 이 역시 마찬가지로 모든 명령어를 읽거나 I/O wait, sleep이 되는 경우가 아니면 지속이 되고 그 경우라면 I/O wait는 I/O queue에 sleep은 sleeplist에 저장을 해주고

process를 종료시켜줍니다.

## ② RR

RR 알고리즘이란 I/O나 sleep 명령어가 나오지 않는다면 최대 10 cycle까지 실행될 수 있는 알고리즘입니다. 기본적으로 run queue에 들어오는 순서대로 CPU가 선택하는 알고리즘입니다. 그렇다면 이 알고리즘을 완성하기 위해서는 순서대로 run Queue에 들어가는 것과 실행되는 cycle의 숫자를 알아야 합니다. 위에 3가지 경우가 존재할 때 각각 어떻게 동작하는지 알아보도록 하겠습니다.

### 1. process가 돌아가고 있는 중

-> 새로운 process가 runtime에 도달한 여부와 상관없이 기존에 돌아가고 있는 process를 계속 진행 시켜주면 됩니다. 만약 진행하던 process가 모든 명령어를 읽거나 I/O wait, sleep이 되는 경우가 아니면 지속이 되고 그 경우라면 I/O wait는 I/O queue에 sleep은 sleeplist에 저장을 해주고 cycle을 0으로 만들어준 후 process를 종료시켜줍니다. 실행되는 중에도 10 cylce에 도달하면 run Queue로 넣어줍니다.

### 2. 실행되는 process가 없고, 새로운 runtime에 도달한 경우

-> 새로운 runtime에 도달한 경우라면 무조건 run Queue에 하나 이상의 process가 존재한다는 의미이기 때문에 run Queue에 있는 맨 앞 첫 번째 process를 실행시켜줍니다. 또한 cycle을 1 증가시켜줍니다. 이 역시 마찬가지로 모든 명령어를 읽거나 I/O wait, sleep이 되는 경우가 아니면 지속이 되고 그 경우라면 I/O wait는 I/O queue에 sleep은 sleeplist에 저장을 해주고 cycle을 0으로 만들어준 후 process를 종료시켜줍니다.

### 3. 실행되는 process가 없고, 새로운 runtime에 도달하지 않는 경우

-> 새로운 runtime에 어떠한 새로운 process가 도달하지 않았기 때문에 run Queue에 process가 존재하는지 아닌지 알 수 없습니다. 그렇기 때문에 먼저 2가지 경우로 나누어 줍니다.

#### 1. run Queue에 없는 경우

-> 이 경우에는 실행시킬 수 있는 process가 존재하지 않기 때문에 NO-OP를 실행

시켜 주면 됩니다.

## 2. run Queue에 있는 경우

run Queue에 있는 맨 앞 첫 번째 process를 실행 시켜줍니다. 또한 cycle을 1 증가시켜줍니다. 이 역시 마찬가지로 모든 명령어를 읽거나 I/O wait, sleep이 되는 경우가 아니면 지속이 되고 그 경우라면 I/O wait는 I/O queue에 sleep은 sleeplist에 저장을 해주고 cycle을 0으로 만들어준 후 process를 종료시켜줍니다.

### ③ SJF - SIMPLE

SJF 알고리즘이란 예측된 값을 가지고 가장 먼저 끝낼 수 있는 process부터 끝내는 과정입니다. 이 예측된 값을 구하기 위한 식은 다음과 같습니다.

- $S[n+1] = (1/n) T[n] + ((n-1)/n) S[n]$
- $S[n+1] = (1/n) \sum_{i=1}^n T[i]$  (simple averaging)

이 두 가지 중 저는 아래의 Exponential과 적용이 쉬운 첫 번째 식을 사용하였습니다. 아래의 그림은 해당 식을 구하는 과정에서 사용한 코드입니다.

<run Queue에 있는 process 중에서 예측되는 값이 제일 적은 것을 골라주는 과정>

```
for(int b=0;b<runQ.size();b=b+2){  
    if(b==0){  
        tempid=stoi(runQ[b]);  
        minid=tempid;  
        min=processif[minid].S.back();  
        mindex=0;  
    }  
    else{  
        tempid=stoi(runQ[b]);  
        if(min>processif[tempid].S.back()){  
            min=processif[runpid].S.back();  
            minid=tempid;  
            mindex=b;  
        }  
    }  
}
```

위의 코드는 runQ에 저장되어있는 process들 중에서 예측되는 값이 제일 적은 process를 골라주는 과정입니다. 위와 같이 제일 작은 값을 고르는 과정은 위에 설명 했다 3가지 경우인 (process가 돌아가고 있는 중, 실행되는 process가 없고, 새로운 runtime에 도달한 경우, 실행되는 process가 없고, 새로운 runtime에 도달하지 않는 경우)에서 모두 확인합니다.

## 1. process가 돌아가고 있는 중

-> 진행되고 있는 process와 만약 새로운 process가 sleep이나 I/O wait에서 run queue로 들어왔다면 비교를 해줍니다. 이때 1. 진행되고 있던 process의 예측값이 더 작은 경우 2. 진행되고 있던 process의 예측값이 더 큰 경우로 나눌 수 있습니다.

### 1. 진행되고 있던 process의 예측값이 더 작은 경우

-> 진행하던 process를 계속 진행하면 됩니다.

### 2. 진행되고 있던 process의 예측값이 더 큰 경우

-> 진행되고 있던 process의 cpu burst 값을 저장한 후 run Queue로 넣어주고 최솟값의 process를 대신 실행 시켜줍니다.

## 2. 실행되는 process가 없고, 새로운 runtime에 도달한 경우

-> 가장 작은 예측값인 마지막 S값을 비교하여 해당 process를 실행시켜줍니다.

## 3. 실행되는 process가 없고, 새로운 runtime에 도달하지 않는 경우

-> 새로운 runtime에 어떠한 새로운 process가 도달하지 않았기 때문에 run Queue에 process가 존재하는지 아닌지 알 수 없습니다. 그렇기 때문에 먼저 2가지 경우로 나누어 줍니다.

### 1. run Queue에 없는 경우

-> 이 경우에는 실행시킬 수 있는 process가 존재하지 않기 때문에 NO-OP를 실행 시켜 주면 됩니다.

### 2. run Queue에 있는 경우

-> 가장 작은 예측값인 마지막 S값을 비교하여 해당 process를 실행시켜 줍니다.

<sleep이거나 I/O wait일 때 S와 T를 업데이트 하는 과정>

```

if(codenum=="4"){
-----update T,S-----
processif[runpid].T.push_back(processif[runpid].cpubust);
min=0;
SSS=(processif[runpid].cpubust/processif[runpid].T.size())+((processif[runpid].S.back()/processif[runpid].T.size())*(processif[runpid].T.size()-1));
processif[runpid].S.push_back(SSS);
processif[runpid].cpubust=0;
-----update T,S-----
sleepL.push_back(to_string(processif[runpid].pid));
sleepL.push_back(processif[runpid].name);
printscheduler(rung,sleepL,10W,processif[runpid],time00,boolrun,boolsche);
processif[runpid].sleepetime=stoi(codetime);
boolrun=0;
}
else if(codenum=="5"){
-----update T,S-----
processif[runpid].T.push_back(processif[runpid].cpubust);
min=0;
SSS=(processif[runpid].cpubust/processif[runpid].T.size())+((processif[runpid].S.back()/processif[runpid].T.size())*(processif[runpid].T.size()-1));
processif[runpid].S.push_back(SSS);
processif[runpid].cpubust=0;
-----update T,S-----
}

```

위에 •  $S[n+1] = (1/n) T[n] + ((n-1)/n) S[n]$ 을 이용하여 명령어가 sleep이거나 I/O wait일 때 업데이트 시켜줍니다. ( $T[n] = CPU BURST$ ,  $S[n]=$  마지막  $S$  값)

#### ④ SJF - EXPONENTIAL

SJF - EXPONENTIAL 알고리즘은 앞선 SJF 함수와 모든 것이 비슷하지만 현재의 수치와 과거의 수치들을  $1/n : (n-1)/n$ 이 아닌  $a : (1-a)$ 의 비율로 적용하는 것이 차이점입니다. 그 식은 아래와 같습니다. simple과 비교해서  $a$ 값과  $(1-a)$ 값이 곱해진 것이 차이점입니다.

$$S[n+1] = a T[n] + (1-a) S[n] ; 0 < a < 1$$

-> 이번 과제에서는  $a$ 값이 0.6으로 설정하였습니다.

```

//-----update T,S-----
processif[runpid].T.push_back(processif[runpid].cpubust);
min=0;
if(processif[runpid].T.size()==1){
    SSS=processif[runpid].cpubust;
}
else{
    SSS=(0.6*(processif[runpid].cpubust/processif[runpid].T.size()))+(0.4*((processif[runpid].S.back()/
processif[runpid].T.size())*(processif[runpid].T.size()-1)));
}
processif[runpid].S.push_back(SSS);
processif[runpid].cpubust=0;
//-----update T,S-----

```

위의 코드를 보시면 simple과 다르게 0.6과 0.4가 각각  $T[n]$ 과  $S[n]$ 에 해당하는 값에 곱해진 것을 확인할 수 있습니다.

### 1-3 메모리 관리 기법

메모리 관리 알고리즘을 짜는 과정에서 공통으로 들어가는 내용을 설명하도록 하겠습니다. 먼저 저는 optimal 알고리즘에 맞춰서 해당 cycle에 실행되는 명령어들을 order라는 vector에 저장한 후에 모든 cycle이 끝날 때까지 while문을 통해 진행되었습니다. op와 arg에 따라 실행하기 전에 4가지 조건을 먼저 확인하였습니다.

#### 1. 먼저 종료된 process가 있는지 확인해주었습니다.

-> 있다면 종료된 process가 가지고 있던 모든 data를 가상메모리와 물리메모리에서 삭제해주었습니다.

#### 2. 새롭게 시작된 process가 있는지 확인해주었습니다.

-> 새로 시작된 process가 존재한다면 가상메모리를 할당해주었습니다.

#### 3. No-op인 cycle인지 확인해주었습니다.

-> No-op인 cycle이었다면 op를 -1로 변경해주었습니다.

#### 4. aid가 할당된 여부를 확인해주었습니다.

-> 할당되어있지 않은 pid라면 pid 값을 저장해주었습니다.

그 후에 op에 따라 4가지 경우를 고려해주었습니다.

#### 1. op가 0인 경우

-> op가 0이라면 해당 arg에 해당하는 값을 가상메모리에 할당시켜주었습니다.

#### 2. op가 1인 경우

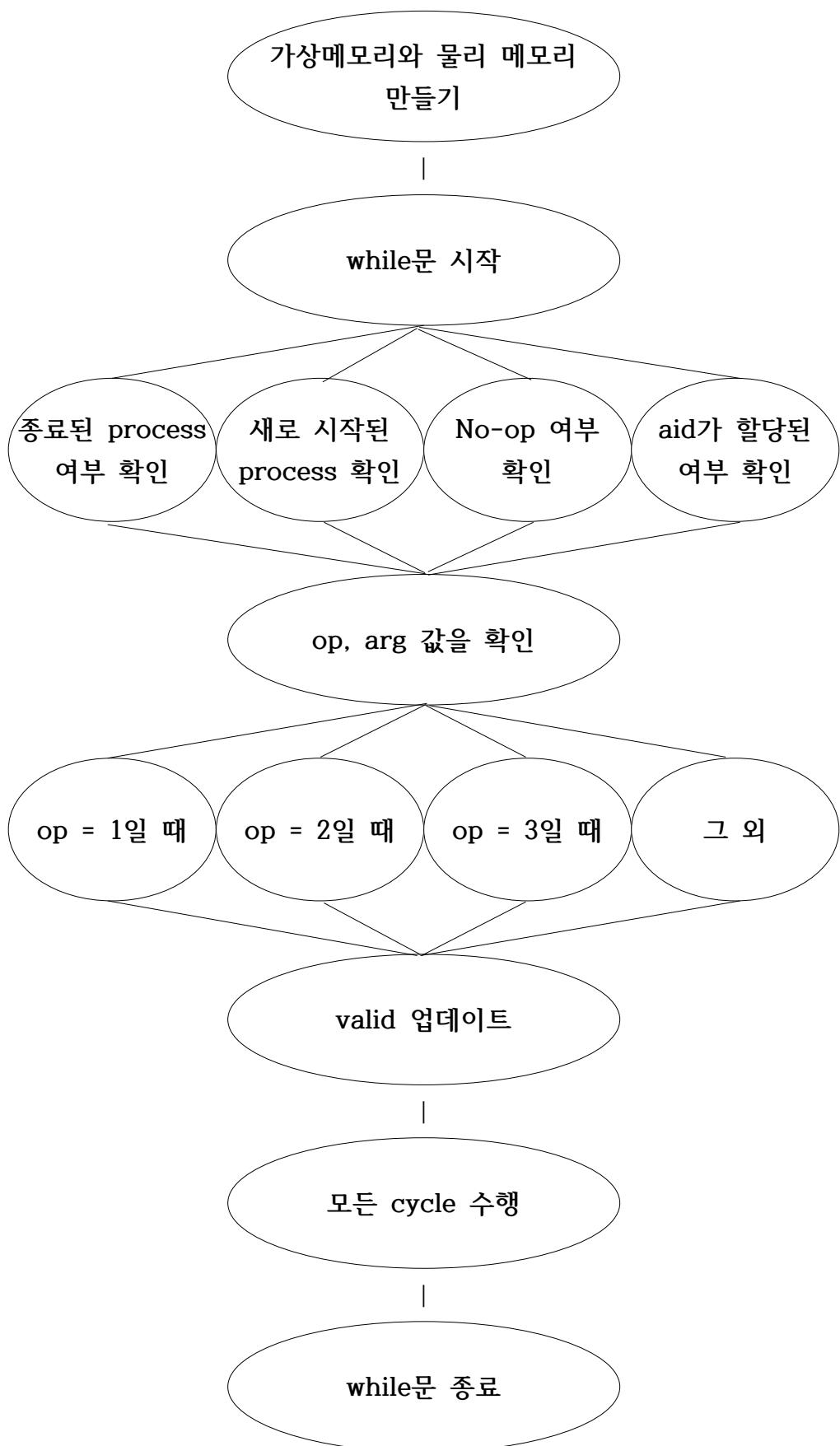
-> op가 1인 경우에는 실질적으로 물리 메모리에 할당되는 과정이라 page fault가 발생하고 교체해줘야 하는 경우가 발생합니다. 이는 해당 기법에 따라 다를것입니다.

#### 3. op가 2인 경우

-> op가 2인 경우에는 할당을 해제하는 과정이라 물리 메모리 및 가상 메모리에서 모두 삭제해주었습니다.

#### 4. op가 그 외인 경우

그 후 Valid 테이블 업데이트 시켜준 후, 만약 모든 cycle을 실행해줬다면 while문을 끝냈습니다.



## ① FIFO

FIFO 알고리즘은 물리 메모리를 교체할 때 가장 오래된 메모리를 바꿔 줘야 합니다. 따라서 물리 메모리에 들어갈 때 1로 만들어주고 만약 나오게 되면 0으로 만들어 줍니다. 그리고 매 사이클마다 각각의 물리 메모리에 있는 값을 +1씩 해줍니다. 아래의 사진은 그러한 시간의 값들 중에서 가장 큰 값을 찾아 교체해주는 과정입니다. 원하는 aid값의 크기가 access가 될 때까지 가장 큰 값을 찾아 교체해줍니다.

<가장 오래된 시간의 값을 찾아 교체해주는 과정>

```
for(int h=0;h<physize;h=h+4){
    if(physical[h]=="-"){
        for(int m=h;m<h+4;m++){
            physical[m]=to_string(arg);
        }
        pagefault++;
        timeaid[arg]=1;
        break;
    }
}
if(timeaid[arg]==0){
    pagefault++;
    for(int n=0;n<aid+1;n++){
        if(n==0){
            max=timeaid[n];
            min=n;
        }
        else if(max<timeaid[n]){
            max=timeaid[n];
            min=n;
        }
    }
    for(int s=0;s<physize;s++){
        if(physical[s]==to_string(min)){
            physical[s]=-";
        }
    }
    timeaid[min]=0;
    for(int b=0;b<physize;b=b+4){
        if(physical[b]=="-"){
            for(int m=b;m<b+4;m++){
                physical[m]=to_string(arg);
            }
            timeaid[arg]=1;
            break;
        }
    }
}
```

## ② LRU

LRU 알고리즘은 물리 메모리를 교체할 때 가장 오랫동안 사용하지 않은 메모리를 바꿔 줘야 합니다. 따라서 물리 메모리에 들어갈 때 1로 만들어주고 만약 나오게 되면 0으로 만들어 줍니다. 그리고 매 사이클마다 각각의 물리 메모리에 있는 값을 +1씩 해줍니다. 여기까지는 fifo 알고리즘과 같습니다.

차이점(fifo): 기존에 있던 물리 메모리에 access 하면 시간은 1로 초기화시켜줍니다.

아래의 사진은 마지막 access 된 후에 지난 시간의 값들 중에서 가장 큰 값을 찾아

교체해주는 과정입니다. 원하는 aid값의 크기가 access가 될 때까지 가장 큰 값을 찾아 교체해줍니다.

<access 후 가장 오래된 시간의 값을 찾아 교체해주는 과정>

```

    for(int h=0;h<physize;h=h+4){
        if(physical[h]=="."){
            for(int m=h;m<h+4;m++){
                physical[m]=to_string(arg);
            }
            pagefault++;
            timeaid[arg]=1;
            break;
        }
    }
    if(timeaid[arg]==0){
        pagefault++;
        for(int n=0;n<aid+1;n++){
            if(n==0){
                max=timeaid[n];
                min=n;
            }
            else if(max<timeaid[n]){
                max=timeaid[n];
                min=n;
            }
        }
        for(int s=0;s<physize;s++){
            if(physical[s]==to_string(min)){
                physical[s](".");
            }
        }
        timeaid[min]=0;
        for(int b=0;b<physize;b=b+4){
            if(physical[b]=="."){
                for(int m=b;m<b+4;m++){
                    physical[m]=to_string(arg);
                }
                timeaid[arg]=1;
                break;
            }
        }
    }
}

```

### ③ LRU-SAMPLED

LRU-SAMPLED 알고리즘은 기존의 LRU와 비슷한 원리지만 time interval과 reference byte가 존재합니다. 이 과제에서는 각각 8의 크기로 지정되었습니다. 따라서 reference byte를 8 cycle마다 2진법으로 계산해서 업데이트시켜주었습니다. 그 전에 먼저 R bit를 가지고 최상위의 reference byte를 업데이트시켜주었습니다. 또한 R bit가 우선순위를 가지기 때문에 R bit가 1이 되는 순간 256을 더해 reference byte보다 우선순위를 주었습니다. 그러면 값 중에서 가장 작은 값을 제일 안 사용될 가능성이 큰 것이기 때문에 교체해줍니다.

아래의 첫 번째 사진은 reference byte를 가지고 각각의 물리 메모리에 access 되어 있는 값을 구해주는 과정입니다. 2진법을 토대로 구했고 8 cycle마다 갱신됩니다. 또한 reference byte도 마찬가지로 Rbit를 최상위로 업데이트됩니다.

아래의 두 번째 사진은 그러한 값을 중에서 가장 작은 값을 찾아서 원하는 aid값의 크기가 access가 될 때까지 교체해주는 과정입니다.

<reference byte 업데이트하는 과정>

```

if(timekk%8==0){
    for(int y=0;y<aid+1;y++){
        for(int v=0;v<1;v--){
            refer[y][v+1]=refer[y][v];
        }
        refer[y][0]=timeaid[y];
    }
    for(int x=0;x<aid+1;x++){
        timeaid[x]=0;
    }
    for(int f=0;f<aid+1;f++){
        for(int g=0;g<8;g++){
            if(refer[f][g]==1){
                if(g==0){
                    score=score+128;
                }
                else if(g==1){
                    score=score+64;
                }
                else if(g==2){
                    score=score+32;
                }
                else if(g==3){
                    score=score+16;
                }
                else if(g==4){
                    score=score+8;
                }
                else if(g==5){
                    score=score+4;
                }
                else if(g==6){
                    score=score+2;
                }
                else if(g==7){
                    score=score+1;
                }
            }
        }
        totalnum[f]=score;
        score=0;
    }
}

```

<가장 작은 reference byte를 찾는 과정>

```

for(int h=0;h<physize;h=h+4){
    if(physical[h]=="-"){
        for(int m=h;m<h+4;m++){
            physical[m]=to_string(arg);
        }
        pagefault++;
        if(timeaid[arg]==0){
            timeaid[arg]=1;
            totalnum[arg]=totalnum[arg]+256;
        }
        timeaid[arg]=1;
        break;
    }
}
if(timeaid[arg]==0){
    pagefault++;
    min=512;
    for(int n=0;n<aid+1;n++){
        for(int v=0;v<physize;v++){
            if(physical[v]==to_string(n)){
                if(min>totalnum[n]){
                    min=totalnum[n];
                    max=n;
                }
            }
        }
    }
    for(int s=0;s<physize;s++){
        if(physical[s]==to_string(max)){
            physical[s]="-";
        }
    }
    for(int b=0;b<physize;b=b+4){
        if(physical[b]=="-"){
            for(int m=b;m<b+4;m++){
                physical[m]=to_string(arg);
            }
            if(timeaid[arg]==0){
                timeaid[arg]=1;
                totalnum[arg]=totalnum[arg]+256;
            }
            timeaid[arg]=1;
            break;
        }
    }
}

```

#### ④ LFU

LFU 알고리즘은 가장 적게 사용된 aid값이 교체되는 과정입니다. 그렇기 때문에 해당 aid 값을 가진 op가 1일 때 count를 +1 해줍니다. 그 후 가장 작은 count를 가진 aid 값을 원하는 aid값의 크기가 access가 될 때까지 교체해주는 과정입니다.

아래의 그림은 가장 작은 count를 가진 aid를 찾아 교체해주는 과정입니다.

<가장 작은 count를 찾는 과정>

```

for(int h=0;h<physize;h=h+4){
    if(physical[h]=="-"){
        for(int m=h;m<h+4;m++){
            physical[m]=to_string(arg);
        }
        pagefault++;
        timeaid[arg]=1;
        break;
    }
}
if(timeaid[arg]==0){
    pagefault++;
    for(int n=0;n<aid+1;n++){
        if(n==0){
            min=9999;
            if(timeaid[n] !=0){
                min=timeaid[n];
            }
            max=n;
        }
        else if(timeaid[n] !=0 && min>timeaid[n]){
            min=timeaid[n];
            max=n;
        }
    }
    for(int s=0;s<physize;s++){
        if(physical[s]==to_string(max)){
            physical[s]="-";
        }
    }
    timeaid[max]=0;
    for(int b=0;b<physize;b=b+4){
        if(physical[b]=="-"){
            for(int m=b;m<b+4;m++){
                physical[m]=to_string(arg);
            }
            timeaid[arg]=1;
            break;
        }
    }
}

```

## ⑤ MFU

MFU 알고리즘은 LFU와 비슷하면서 반대의 개념을 가진 알고리즘입니다. 가장 적게 사용된 aid값이 교체되는 과정이 아닌 가장 많이 사용된 aid값이 교체되는 과정입니다. 그렇기 때문에 해당 aid 값을 가진 op가 1일 때 count를 +1 해줍니다. 그 후 가장 큰 count를 가진 aid 값을 원하는 aid값의 크기가 access가 될 때까지 교체해주는 과정입니다.

아래의 그림은 가장 큰 count를 가진 aid를 찾아 교체해주는 과정입니다.

<가장 큰 count를 찾는 과정>

```
if((max-min)<5){  
    for(int h=0;h<physize;h=h+4){  
        if(physical[h]==".") {  
            for(int m=h;m<h+4;m++){  
                physical[m]=to_string(arg);  
            }  
            pagefault++;  
            timeaid[arg]=1;  
            break;  
        }  
        if(timeaid[arg]==0){  
            pagefault++;  
            for(int n=0;n<aid+1;n++){  
                if(n==0){  
                    max=timeaid[n];  
                    min=n;  
                }  
                else if(max<timeaid[n]){  
                    max=timeaid[n];  
                    min=n;  
                }  
            }  
            for(int s=0;s<physize;s++){  
                if(physical[s]==to_string(min)){  
                    physical[s]=".";  
                }  
            }  
            timeaid[min]=0;  
            for(int b=0;b<physize;b=b+4){  
                if(physical[b]==".") {  
                    for(int m=b;m<b+4;m++){  
                        physical[m]=to_string(arg);  
                    }  
                    timeaid[arg]=1;  
                    break;  
                }  
            }  
        }  
    }  
}
```

## ⑥ OPTIMAL

OPTIMAL 알고리즘은 미리 미래의 cycle을 알아야지만 사용할 수 있는 알고리즘입니다. 그 이유는 애초에 평소에는 OPTIMAL을 구현할 수 없습니다. 언제 어떻게 명령어가 들어오는지 알 수 없기 때문입니다. 그렇기 때문에 미리 해당 스케줄러가 이미 모든 결과를 수행한 값을 저장한 후 가장 오랫동안 사용되지 않을 aid값을 해당 aid값이 access 될 수 있을 때까지 변경해줍니다.

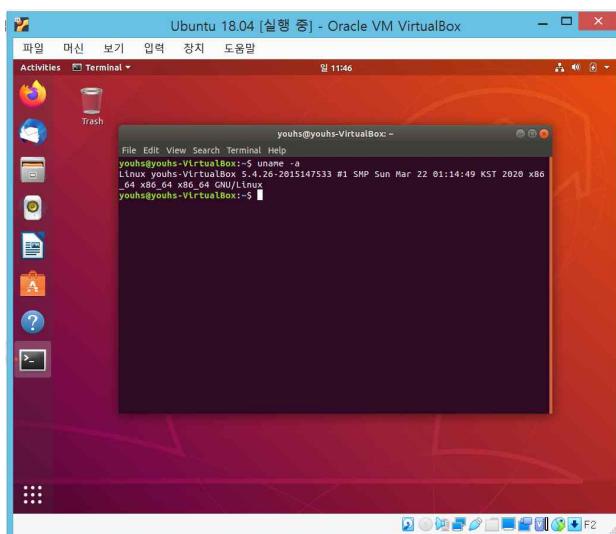
아래의 그림은 가장 가깝게 access 될 cycle의 간격을 구하는 것입니다. 만약 끝까지 해당 aid 값이 access 되지 않는다면 cycle의 값은 최대로 설정하였습니다.

<가장 가까운 cycle내에 access 될 간격을 구하는 과정>

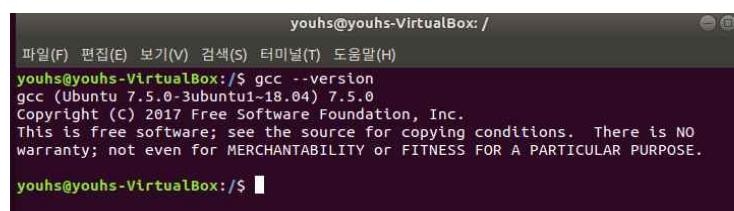
```
for(int v=0;v<aid+1;v++){
    booloptimal=0;
    for(int z=0;z<physize;z++){
        if(physical[z]==to_string(v)){
            for(int n=0;n<order.size();n=n+3){
                if(order[n+1]=="1"){
                    if(order[n+2]==to_string(v)){
                        optimal[v]=n+2;
                        booloptimal=1;
                        break;
                    }
                }
            }
        }
    }
    if(booloptimal==0){
        optimal[v]=999999;
    }
}
```

## 2. 개발 환경 명시

### 2-1 uname -a 실행 결과



### 2-2 사용한 컴파일 버전, CPU, 메모리정보



```

youhs@youhs-VirtualBox: /proc
파일(F) 편집(E) 보기(V) 검색(S) 터미널(T) 도움말(H)
boot      home      lib64      opt      sbin      sys      vmlinuz
cdrom    initrd.img  lost+found  proc      snap     tmp      vmlinuz.old
youhs@youhs-VirtualBox:/proc$ cd proc
youhs@youhs-VirtualBox:/proc$ cat cpuinfo
processor       : 0
vendor_id      : GenuineIntel
cpu family     : 6
model          : 61
model name     : Intel(R) Core(TM) i3-5005U CPU @ 2.00GHz
stepping        : 4
cpu MHz        : 1995.380
cache size     : 3072 KB
physical id    : 0
siblings        : 1
core id        : 0
cpu cores      : 1
apicid         : 0
initial apicid : 0
fpu             : yes
fpu_exception   : yes
cpuid level    : 20
wp              : yes
flags           : fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge mca cmov
pat pse36 clflush mmx fxsr sse sse2 ht syscall nx rdtscp lm constant_tsc rep_goo

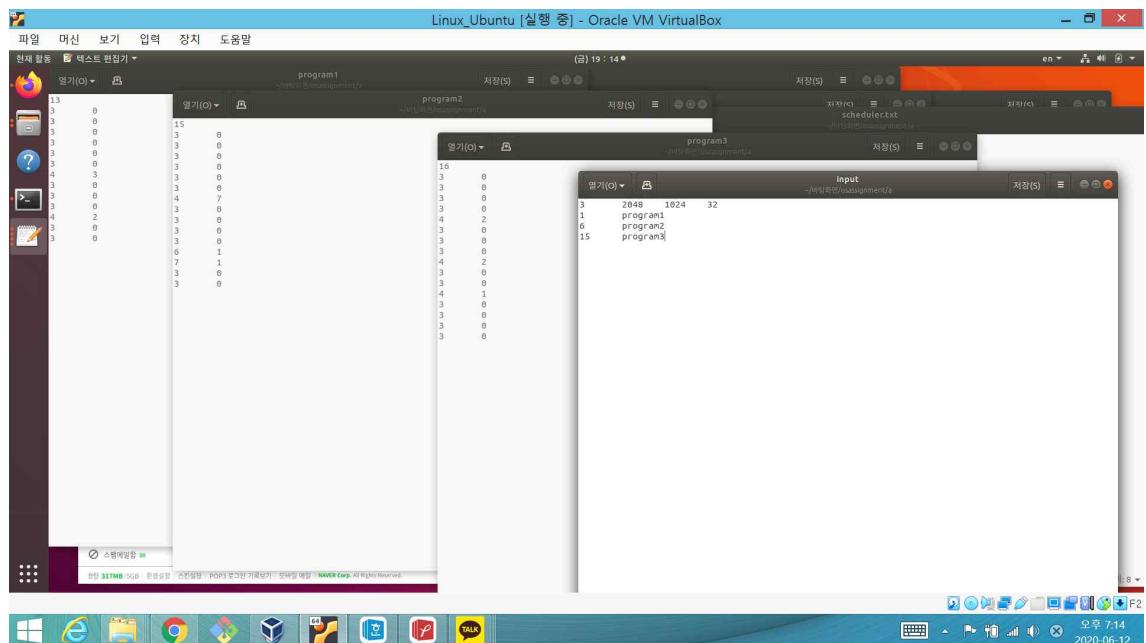
youhs@youhs-VirtualBox: /
파일(F) 편집(E) 보기(V) 검색(S) 터미널(T) 도움말(H)
youhs@youhs-VirtualBox:$ cat /proc/meminfo | grep MemTotal
MemTotal:      4030904 kB
youhs@youhs-VirtualBox:$

```

### 3. 결과 화면 스크린샷과 그 것에 대한 토의 내용

#### 3-1 스케줄링 알고리즘 비교

##### <INPUT1 (a)>



## 1-<FCFS>



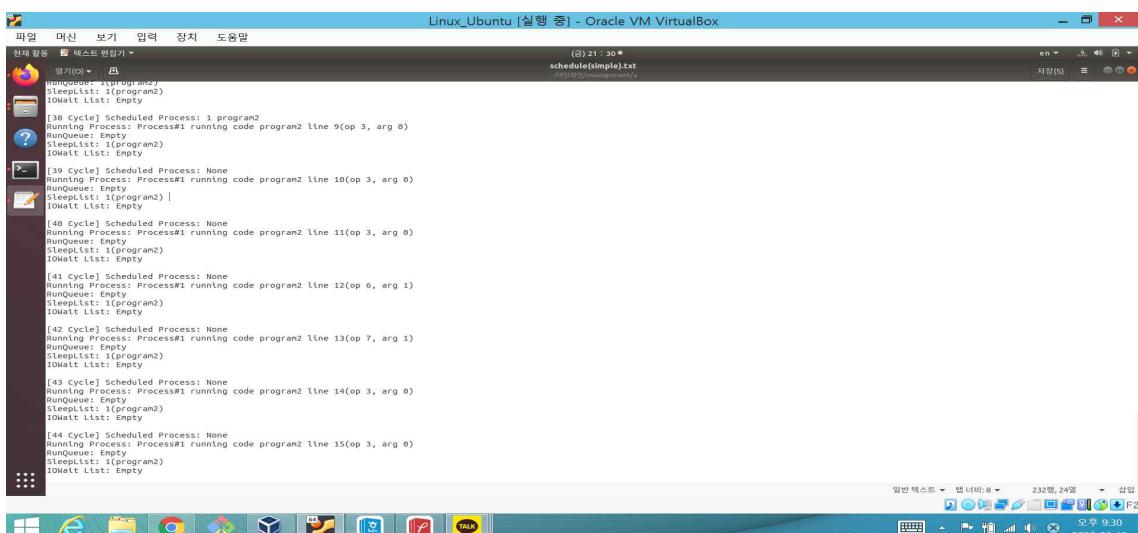
```
(일) 21 : 29 ● schedule(fcfs).txt
schedule(fcfs).txt
[39 Cycle] Scheduled Process: 2 program
Running Process: Process#2 running code program3 line 10(op 3, arg 0)
RunQueue: Empty
SleepList: 2(program3)
IOWait List: Empty
[40 Cycle] Scheduled Process: None
Running Process: Process#2 running code program3 line 11(op 3, arg 0)
RunQueue: Empty
SleepList: Empty
IOWait List: Empty
[41 Cycle] Scheduled Process: None
Running Process: Process#2 running code program3 line 12(op 4, arg 1)
RunQueue: Empty
SleepList: Empty
IOWait List: Empty
[42 Cycle] Scheduled Process: 2 program
Running Process: Process#2 running code program3 line 13(op 3, arg 0)
RunQueue: Empty
SleepList: Empty
IOWait List: Empty
[43 Cycle] Scheduled Process: None
Running Process: Process#2 running code program3 line 14(op 3, arg 0)
RunQueue: Empty
SleepList: Empty
IOWait List: Empty
[44 Cycle] Scheduled Process: None
Running Process: Process#2 running code program3 line 15(op 3, arg 0)
RunQueue: Empty
SleepList: Empty
IOWait List: Empty
[45 Cycle] Scheduled Process: None
Running Process: Process#2 running code program3 line 16(op 3, arg 0)
RunQueue: Empty
SleepList: Empty
IOWait List: Empty
```

## 1-<RR>



```
(일) 21 : 30 ● schedule(rr).txt
schedule(rr).txt
[39 Cycle] Scheduled Process: 2 program
Running Process: Process#2 running code program3 line 10(op 3, arg 0)
RunQueue: Empty
SleepList: 2(program3)
IOWait List: Empty
[40 Cycle] Scheduled Process: None
Running Process: Process#2 running code program3 line 11(op 3, arg 0)
RunQueue: Empty
SleepList: Empty
IOWait List: Empty
[41 Cycle] Scheduled Process: None
Running Process: Process#2 running code program3 line 12(op 4, arg 1)
RunQueue: Empty
SleepList: Empty
IOWait List: Empty
[42 Cycle] Scheduled Process: 2 program
Running Process: Process#2 running code program3 line 13(op 3, arg 0)
RunQueue: Empty
SleepList: Empty
IOWait List: Empty
[43 Cycle] Scheduled Process: None
Running Process: Process#2 running code program3 line 14(op 3, arg 0)
RunQueue: Empty
SleepList: Empty
IOWait List: Empty
[44 Cycle] Scheduled Process: None
Running Process: Process#2 running code program3 line 15(op 3, arg 0)
RunQueue: Empty
SleepList: Empty
IOWait List: Empty
[45 Cycle] Scheduled Process: None
Running Process: Process#2 running code program3 line 16(op 3, arg 0)
RunQueue: Empty
SleepList: Empty
IOWait List: Empty
```

## 1-<SJF-SIMPLE>



```
(일) 21 : 30 ● schedule(simple).txt
schedule(simple).txt
[39 Cycle] Scheduled Process: 1 program
Running Process: Process#1 running code program2 line 9(op 3, arg 0)
RunQueue: Empty
SleepList: 1(program2)
IOWait List: Empty
[40 Cycle] Scheduled Process: None
Running Process: Process#1 running code program2 line 10(op 3, arg 0)
RunQueue: Empty
SleepList: 1(program2)
IOWait List: Empty
[41 Cycle] Scheduled Process: None
Running Process: Process#1 running code program2 line 11(op 3, arg 0)
RunQueue: Empty
SleepList: 1(program2)
IOWait List: Empty
[42 Cycle] Scheduled Process: None
Running Process: Process#1 running code program2 line 12(op 6, arg 0)
RunQueue: Empty
SleepList: 1(program2)
IOWait List: Empty
[43 Cycle] Scheduled Process: None
Running Process: Process#1 running code program2 line 13(op 7, arg 1)
RunQueue: Empty
SleepList: 1(program2)
IOWait List: Empty
[44 Cycle] Scheduled Process: None
Running Process: Process#1 running code program2 line 14(op 3, arg 0)
RunQueue: Empty
SleepList: 1(program2)
IOWait List: Empty
[45 Cycle] Scheduled Process: None
Running Process: Process#1 running code program2 line 15(op 3, arg 0)
RunQueue: Empty
SleepList: 1(program2)
IOWait List: Empty
```

## 1-<SJF-EXPONENTIAL>

```
([38 Cycle] Scheduled Process: 1 program2
Running Process: Process#1 running code program2 line 9(op 3, arg 0)
RunQueue: Empty
SleepList: 1(program2)
IOWait List: Empty
[39 Cycle] Scheduled Process: None
Running Process: Process#1 running code program2 line 10(op 3, arg 0)
RunQueue: Empty
SleepList: 1(program2)
IOWait List: Empty
[40 Cycle] Scheduled Process: None
Running Process: Process#1 running code program2 line 11(op 3, arg 0)
RunQueue: Empty
SleepList: 1(program2)
IOWait List: Empty
[41 Cycle] Scheduled Process: None
Running Process: Process#1 running code program2 line 12(op 6, arg 1)
RunQueue: Empty
SleepList: 1(program2)
IOWait List: Empty
[42 Cycle] Scheduled Process: None
Running Process: Process#1 running code program2 line 13(op 7, arg 1)
RunQueue: Empty
SleepList: 1(program2)
IOWait List: Empty
[43 Cycle] Scheduled Process: None
Running Process: Process#1 running code program2 line 14(op 3, arg 0)
RunQueue: Empty
SleepList: 1(program2)
IOWait List: Empty
[44 Cycle] Scheduled Process: None
Running Process: Process#1 running code program2 line 15(op 3, arg 0)
RunQueue: Empty
SleepList: 1(program2)
IOWait List: Empty
```

## <INPUT2 (input\_sJf)>

```
3 2048 1024 32
sJFProc1
sJFProc2
sJFProc3
13 0 16 0 15 0
3 0 3 0 3 0
3 0 3 0 3 0
3 0 4 3 3 0
3 0 3 0 3 0
3 0 3 0 3 0
3 0 3 0 3 0
3 0 4 2 3 0
3 0 3 0 3 0
4 2 3 0 3 0
3 0 3 0 3 0
3 0 3 0 3 0
3 0 3 0 3 0
3 0 4 1 3 0
3 0 3 0 0|
```

## 2-<FCFS>

```
([38 Cycle] Scheduled Process: 1 sJFProc2
Running Process: Process#1 running code sJFProc2 line 10(op 3, arg 0)
RunQueue: Empty
SleepList: Empty
IOWait List: Empty
[39 Cycle] Scheduled Process: None
Running Process: Process#1 running code sJFProc2 line 11(op 3, arg 0)
RunQueue: Empty
SleepList: Empty
IOWait List: Empty
[40 Cycle] Scheduled Process: None
Running Process: Process#1 running code sJFProc2 line 12(op 3, arg 0)
RunQueue: Empty
SleepList: Empty
IOWait List: Empty
[41 Cycle] Scheduled Process: None
Running Process: Process#1 running code sJFProc2 line 13(op 3, arg 0)
RunQueue: Empty
SleepList: Empty
IOWait List: Empty
[42 Cycle] Scheduled Process: None
Running Process: Process#1 running code sJFProc2 line 14(op 3, arg 0)
RunQueue: Empty
SleepList: Empty
IOWait List: Empty
[43 Cycle] Scheduled Process: None
Running Process: Process#1 running code sJFProc2 line 15(op 4, arg 1)
RunQueue: Empty
SleepList: 1(sJFProc2)
IOWait List: Empty
[44 Cycle] Scheduled Process: 1 sJFProc2
Running Process: Process#1 running code sJFProc2 line 16(op 3, arg 0)
RunQueue: Empty
SleepList: Empty
IOWait List: Empty
```

2-<RR>

```
Linux_Ubuntu [실행 중] - Oracle VM VirtualBox
파일 마신 보기 입력 장치 도움말
현재 활동 텍스트 편집기 ▾
열기(O) ▾
newfile(0) sjfProc3
SleepList: Empty
IOWait List: Empty
[38 Cycle] Scheduled Process: None
Running Process: Process#1 running code sjfProc2 line 12(op 3, arg 0)
RunQueue: 2(sjfProc3)
SleepList: Empty
IOWait List: Empty
[39 Cycle] Scheduled Process: None
Running Process: Process#1 running code sjfProc2 line 13(op 3, arg 0)
RunQueue: 2(sjfProc3)
SleepList: Empty
IOWait List: Empty
[40 Cycle] Scheduled Process: None
Running Process: Process#1 running code sjfProc2 line 14(op 3, arg 0)
RunQueue: 2(sjfProc3)
SleepList: Empty
IOWait List: Empty
[41 Cycle] Scheduled Process: None
Running Process: Process#1 running code sjfProc2 line 15(op 4, arg 1)
RunQueue: 2(sjfProc3)
SleepList: 3(sjfProc2)
IOWait List: Empty
[42 Cycle] Scheduled Process: 2 sjfProc3
Running Process: Process#2 running code sjfProc3 line 14(op 3, arg 0)
RunQueue: 1(sjfProc2)
SleepList: Empty
IOWait List: Empty
[43 Cycle] Scheduled Process: None
Running Process: Process#2 running code sjfProc3 line 15(op 3, arg 0)
RunQueue: 1(sjfProc2)
SleepList: Empty
IOWait List: Empty
[44 Cycle] Scheduled Process: 1 sjfProc2
Running Process: Process#1 running code sjfProc2 line 16(op 3, arg 0)
RunQueue: Empty
SleepList: Empty
IOWait List: Empty
```

2-<SJF-SIMPLE>

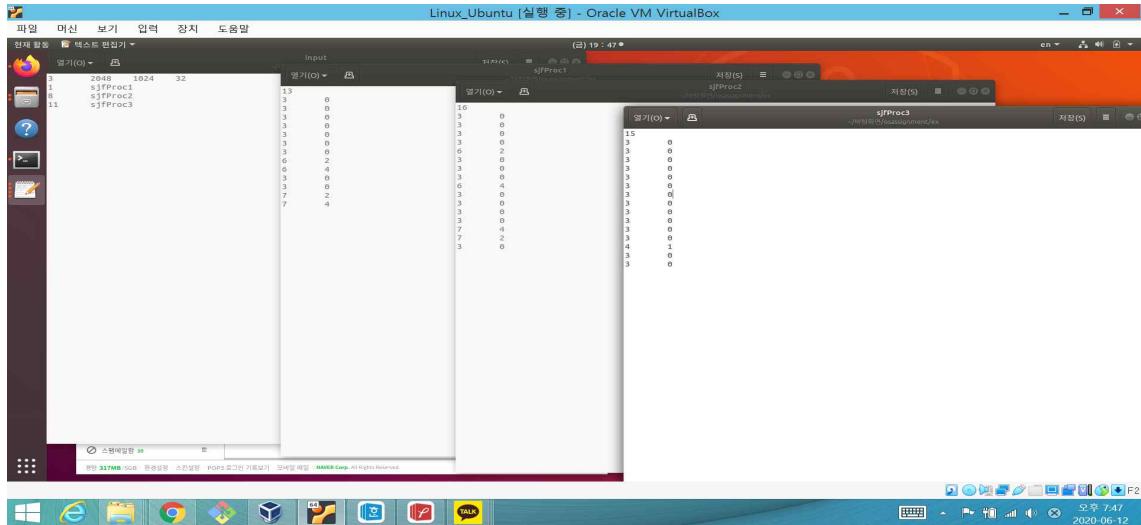
```
Linux_Ubuntu [실행 중] - Oracle VM VirtualBox
[ 01 ] 파일 마신 보기 앱력 장치 도움말
[ 02 ] 현재 활동 텍스트 편집기 ▾
[ 03 ] 웹(아이오) ▾
[ 04 ] SleepList: Empty
[ 05 ] IOWait List: Empty
[ 06 ] [38 Cycle] Scheduled Process: 0 sjfProc1
[ 07 ] Running Process: Process@0 running code sjfProc1 line 9(op 3, arg 0)
[ 08 ] RunQueue: 2(sjfProc3)
[ 09 ] SleepList: Empty
[ 10 ] IOWait List: Empty
[ 11 ] [39 Cycle] Scheduled Process: None
[ 12 ] Running Process: Process@0 running code sjfProc1 line 10(op 3, arg 0)
[ 13 ] RunQueue: 2(sjfProc3)
[ 14 ] SleepList: Empty
[ 15 ] IOWait List: Empty
[ 16 ] [40 Cycle] Scheduled Process: None
[ 17 ] Running Process: Process@0 running code sjfProc1 line 11(op 4, arg 2)
[ 18 ] RunQueue: 2(sjfProc3)
[ 19 ] SleepList: 0(sjfProc1)
[ 20 ] IOWait List: Empty
[ 21 ] [41 Cycle] Scheduled Process: 2 sjfProc3
[ 22 ] Running Process: Process@2 running code sjfProc3 line 14(op 3, arg 0)
[ 23 ] RunQueue: Empty
[ 24 ] SleepList: 0(sjfProc1)
[ 25 ] IOWait List: Empty
[ 26 ] [42 Cycle] Scheduled Process: 0 sjfProc1
[ 27 ] Running Process: Process@0 running code sjfProc1 line 12(op 3, arg 0)
[ 28 ] RunQueue: 2(sjfProc3)
[ 29 ] SleepList: Empty
[ 30 ] IOWait List: Empty
[ 31 ] [43 Cycle] Scheduled Process: None
[ 32 ] Running Process: Process@0 running code sjfProc1 line 13(op 3, arg 0)
[ 33 ] RunQueue: 2(sjfProc3)
[ 34 ] SleepList: Empty
[ 35 ] IOWait List: Empty
[ 36 ] [44 Cycle] Scheduled Process: 2 sjfProc3
[ 37 ] Running Process: Process@2 running code sjfProc3 line 15(op 3, arg 0)
[ 38 ] RunQueue: Empty
[ 39 ] SleepList: Empty
[ 40 ] IOWait List: Empty
[ 41 ] "/home/youba/비방학연/osassignment/input_sjf/scheduler(simple).txt" 파일을 읽어들이는 중입니다...
[ 42 ] (금) 21:34 scheduler(simple).txt
[ 43 ] /home/youba/비방학연/osassignment/input_sjf/scheduler(simple).txt
[ 44 ] 설정(G) ▾
[ 45 ] 일정 텍스트 ▾ 웹 뉴스: 8 ▾ 222행, 1열 ▾ 살펴보기
[ 46 ] 오류: 9:34 F2
```

## 2-<SIE-EXPONENTIAL>

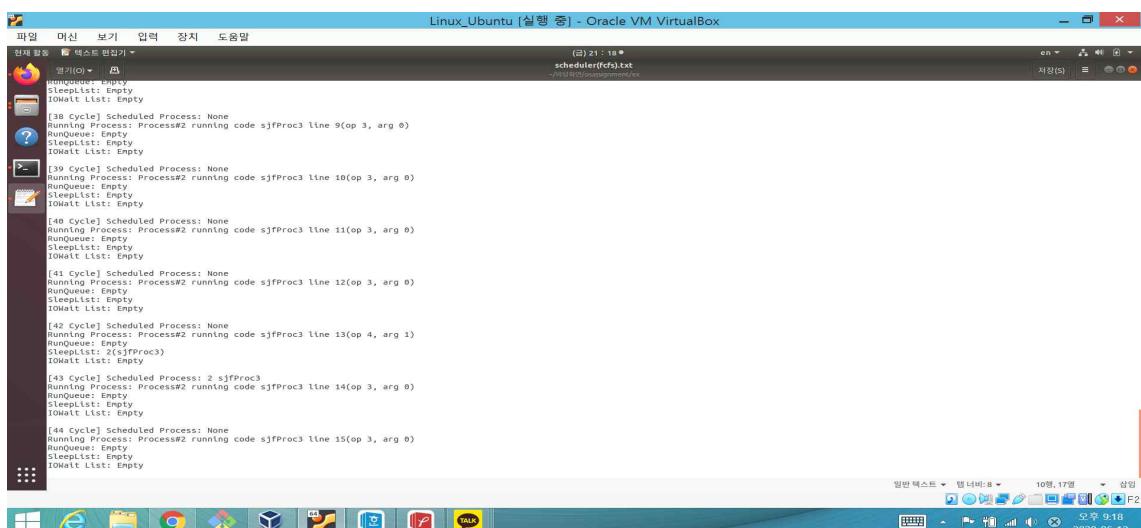
```
Linux_Ubuntu [실행 중] - Oracle VM VirtualBox

파일 마신 보기 업력 장치 도움말
전체 할정 텍스트 편집기 ▾
모드 ▾ B
scheduler(exponential).txt
(화) 21 : 35
调度策略: 指数调度
调度器状态: 空闲
睡眠列表: 空
IO等待列表: 空
[38 周期] 调度进程: 0 sjfProc1
运行队列: 2(sjfProc0)
RunQueue: 2(sjfProc0)
睡眠列表: 空
IO等待列表: 空
[39 周期] 调度进程: None
运行进程: Process#0 运行代码 sjfProc1 线程 10(op 3, arg 0)
RunQueue: 2(sjfProc3)
睡眠列表: 空
IO等待列表: 空
[40 周期] 调度进程: None
运行进程: Process#0 运行代码 sjfProc1 线程 11(op 4, arg 2)
RunQueue: 2(sjfProc3)
睡眠列表: 0(sjfProc1)
IO等待列表: 空
[41 周期] 调度进程: 2 sjfProc3
运行进程: Process#2 运行代码 sjfProc3 线程 14(op 3, arg 0)
RunQueue: 空
睡眠列表: 空
IO等待列表: 空
[42 周期] 调度进程: 0 sjfProc1
运行进程: Process#0 运行代码 sjfProc1 线程 12(op 3, arg 0)
RunQueue: 2(sjfProc3)
睡眠列表: 2(sjfProc1)
IO等待列表: 空
[43 周期] 调度进程: None
运行进程: Process#0 运行代码 sjfProc1 线程 13(op 3, arg 0)
RunQueue: 2(sjfProc3)
睡眠列表: 空
IO等待列表: 空
[44 周期] 调度进程: 2 sjfProc3
运行进程: Process#2 运行代码 sjfProc3 线程 15(op 3, arg 0)
RunQueue: 空
睡眠列表: 空
IO等待列表: 空
```

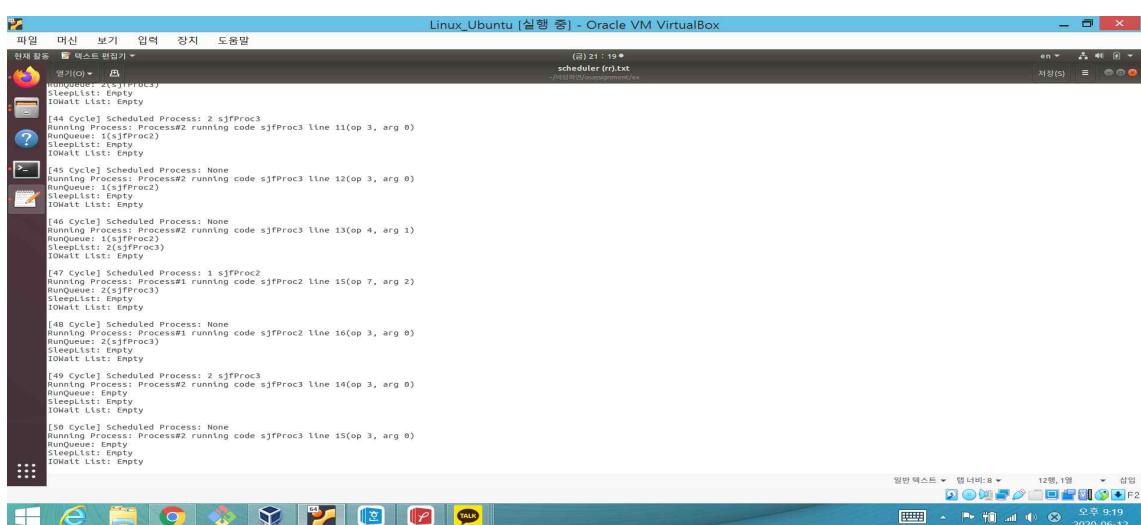
### <INPUT3 (ex)>



### 3-<FCFS>



### 3-<RR>



### 3-<SJF-SIMPLE>

### 3-<SJF-EXPONENTIAL>

```
파일 마신 보기 입력 장치 도움말
Linux_Ubuntu [실행 중] - Oracle VM VirtualBox
(금) 21:22 *
scheduler(exponential).txt
[00 Cycle] Scheduled Process: None
Running Process: None
Runqueue: Empty
SleepList: Empty
IOWait List: Empty

[01 Cycle] Scheduled Process: None
Running Process: Process#1 running code sjfProc1 line 9(op 3, arg 0)
Runqueue: Empty
SleepList: Empty
IOWait List: Empty

[02 Cycle] Scheduled Process: None
Running Process: Process#2 running code sjfProc2 line 10(op 3, arg 0)
Runqueue: Empty
SleepList: Empty
IOWait List: Empty

[03 Cycle] Scheduled Process: None
Running Process: Process#2 running code sjfProc3 line 11(op 3, arg 0)
Runqueue: Empty
SleepList: Empty
IOWait List: Empty

[04 Cycle] Scheduled Process: None
Running Process: Process#2 running code sjfProc3 line 12(op 3, arg 0)
Runqueue: Empty
SleepList: Empty
IOWait List: Empty

[05 Cycle] Scheduled Process: None
Running Process: Process#2 running code sjfProc3 line 13(op 4, arg 1)
Runqueue: Empty
SleepList: 2(sjfProc3)
IOWait List: Empty

[06 Cycle] Scheduled Process: 2 sjfProc3
Running Process: Process#2 running code sjfProc3 line 14(op 3, arg 0)
Runqueue: Empty
SleepList: Empty
IOWait List: Empty

[07 Cycle] Scheduled Process: None
Running Process: Process#2 running code sjfProc3 line 15(op 3, arg 0)
Runqueue: Empty
SleepList: Empty
IOWait List: Empty
```

	기본	input1	input2	input3
FCFS	33	45	44	44
RR	33	45	44	50
SJF-SIMPLE	32	44	44	44
SJF-EXPONENTIAL	32	44	44	44

<기본이라고 적혀 인풋은 과제설명에 나와 있는 예시 인풋입니다.>

기본적으로 스케줄 되는 전체 사이클의 숫자를 보면 그렇게 눈에 띠는 차이를 볼 수 없습니다. 하지만 전체 사이클의 숫자보다는 얼마나 효율적으로 사용되는 여부를 따졌을 때는 큰 차이가 있을 것이라고 생각합니다. 특히 뒤에 나올 메모리의 교체되는 숫자뿐만 아니라 사이클의 전체 숫자가 적은 일부 프로세스들은 FCFS가 아닌 다른 방법들은 사용하게 되면 금방 끝날 수가 있습니다.

또한 input3은 dead lock이 걸리게 만든 예시입니다. 그 결과를 보시면 오히려 RR 일 때 dead lock이 걸려 전체 사이클의 숫자가 늘어나는 것을 확인 할 수 있습니다. 그리하여 기본적인 FCFS 보다는 락이 두 번 연속 걸려 busy waiting 걸리는 경우가 아니라면 RR이 전체적인 process의 평균 소요 시간으로 봤을 때는 효율적인 것 같습니다. FCFS 같은 경우에는 마지막 남은 프로세스에 I/O wait 나 sleep이 뒤쪽에 많이 남아있을 경우 낭비되는 사이클이 많아질 수 있는 문제가 생깁니다.

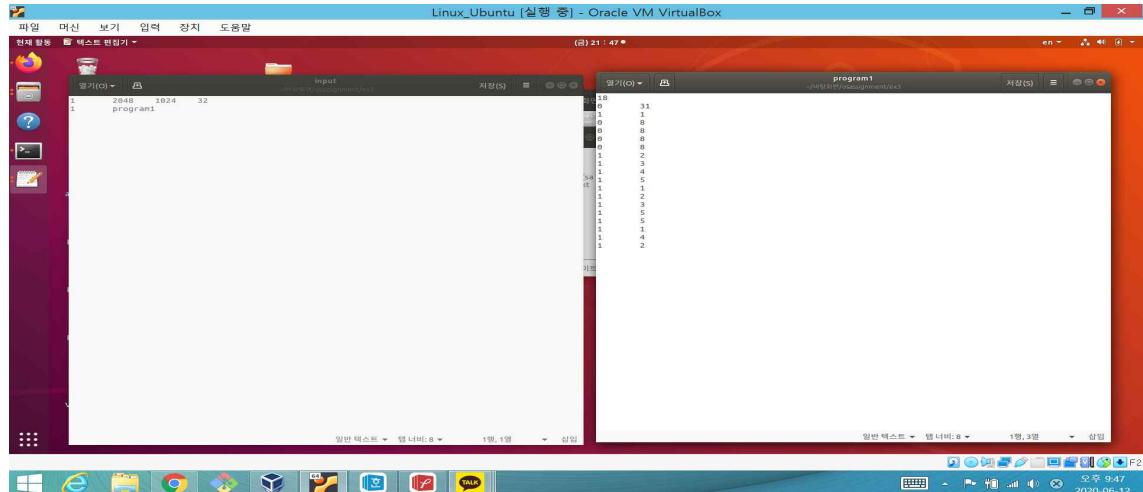
그렇다면 SJF를 보면 SJF 같은 경우에는 자동으로 busy waiting이 풀릴 수 있는 RR과는 다르게 영원히 풀리지 않는 busy waiting이 생길 수도 있습니다. 락이 걸린 순간 S값은 변하지 않는데 이 때 I/O 혹은 sleep에서 넘어오는 process들이 오히려 S값이 더 크거나 발생하지 않는다면 그 process는 자동으로 영원히 풀리지 않게 됩니다.

SJF-Exponential 같은 경우에는 a 값에 따라 예측되는 결과 값이 다르게 치중되고 SJF-simple과는 다른 결과값이 도출 되어질 수 있습니다. 그 결과의 보다 효율적인 것은 누가 더 순간순간의 예측값이 효율적이냐에 따라 다를거 같습니다. 물론 많은 데이터가 쌓이면 보다 올바른 예측값이 구해질 수 있을거라 예상합니다.

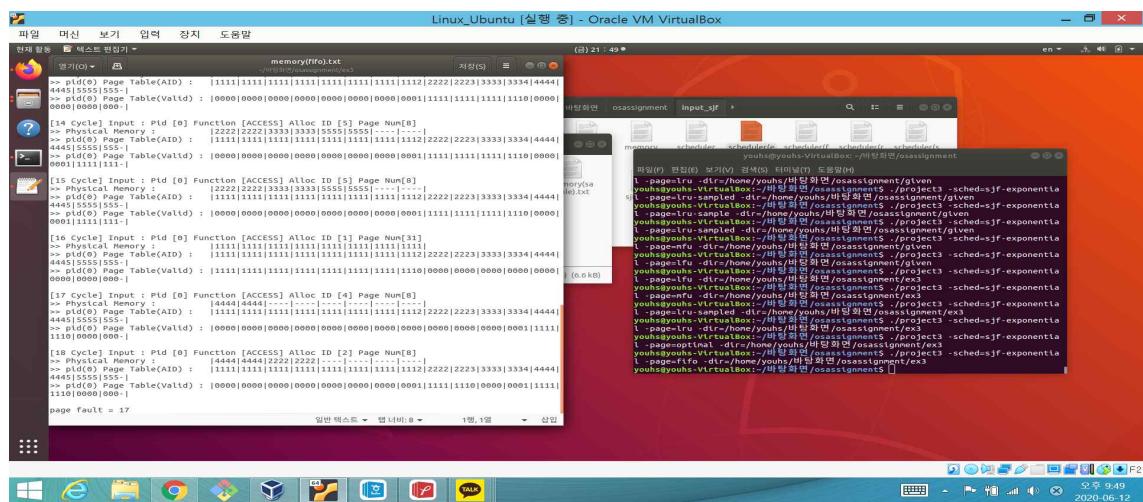
이처럼 각각의 스케줄하는 알고리즘에는 장단점이 있고 각각 상황에 맞는 보다 적합한 알고리즘이 존재할 것이라고 생각합니다.

### 3-2 페이지 교체 알고리즘

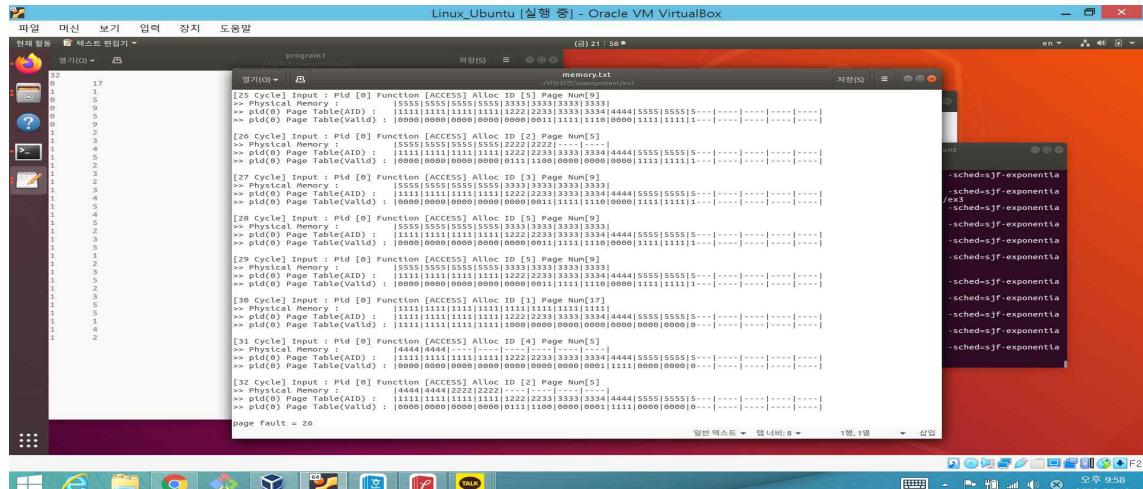
#### <INPUT1 (EX3)>



#### 1-<FIFO>



#### 1-<LFU>



1-<LRU>

## 1-<LRU-SAMPLED>

1-<MFU>

1-<OPTIMAL>

<INPUT2 (INPUT\_MEMTEST>

A screenshot of a Linux desktop environment, likely Ubuntu, running in Oracle VM VirtualBox. The desktop has a red-orange theme. Two terminal windows are open. The left terminal window shows a program's output with the following text:

```
32
17
1
5
9
5
9
2
3
4
5
2
3
2
3
4
5
4
5
2
3
5
1
2
3
2
5
2
3
5
1
4
2
```

At the bottom of this window, the status bar says "page fault = 28". The right terminal window shows a file named "input" with the following content:

```
1 2048 1024 32
program]
Nu
33
33
33
111
```

The taskbar at the bottom contains icons for various applications, including a web browser, file manager, and system tools. The system tray shows the date as "오늘 9:52".

2-<FIFO>

2-<LFU>

2-<LRU>

```
Linux_Ubuntu [실행 중] - Oracle VM VirtualBox
파일 마신 보기 장치 도움말
현재 활동 텍스트 편집기 (官司) 21 : 00 ●
모든 파일
[> Physical Memory : [1111|1111|1111|2222|2222|0505|5555|3333|3333]
--> pid(e) Page Table(AID) : [1111|1111|1111|2222|2222|3333|3333|4444|4444|5555|5555|-----|-----|-----|-----|-----]
--> pid(e) Page Table(Valid) : [1111|1111|1111|1111|1111|1111|0000|0000|1111|1111|-----|-----|-----|-----|-----]

[19 Cycle] Input : Pid [0] Function [ACCESS] Alloc ID [4] Page Num[8]
--> Physical Memory : [1111|1111|1111|2222|2222|4444|4444|3333|3333]
--> pid(e) Page Table(AID) : [1111|1111|1111|2222|2222|3333|3333|4444|4444|5555|5555|-----|-----|-----|-----|-----]
--> pid(e) Page Table(Valid) : [1111|1111|1111|1111|1111|1111|0000|0000|0000|0000|-----|-----|-----|-----|-----]

[20 Cycle] Input : Pid [0] Function [ACCESS] Alloc ID [3] Page Num[8]
--> Physical Memory : [1111|1111|2222|2222|4444|4444|3333|3333]
--> pid(e) Page Table(AID) : [1111|1111|2222|2222|3333|3333|4444|4444|5555|5555|-----|-----|-----|-----|-----]
--> pid(e) Page Table(Valid) : [1111|1111|1111|1111|1111|1111|0000|0000|0000|0000|-----|-----|-----|-----|-----]

[21 Cycle] Input : Pid [0] Function [ALLOCATION] Alloc ID [0] Page Num[12]
--> Physical Memory : [1111|1111|2222|2222|4444|4444|3333|3333]
--> pid(e) Page Table(AID) : [1111|1111|2222|2222|3333|3333|4444|4444|5555|5555|6666|6666|6666|6666|-----|-----]
--> pid(e) Page Table(Valid) : [1111|1111|1111|1111|1111|1111|0000|0000|0000|0000|0000|0000|0000|0000|-----|-----]

[22 Cycle] Input : Pid [0] Function [RELEASE] Alloc ID [2] Page Num[0]
--> Physical Memory : [1111|1111|-----|-----|4444|4444|3333|3333]
--> pid(e) Page Table(AID) : [1111|1111|-----|-----|3333|3333|4444|4444|5555|5555|6666|6666|6666|6666|-----|-----]
--> pid(e) Page Table(Valid) : [1111|1111|1111|1111|1111|1111|0000|0000|0000|0000|0000|0000|0000|0000|-----|-----]

[23 Cycle] Input : Pid [0] Function [ACCESS] Alloc ID [4] Page Num[12]
--> Physical Memory : [1111|1111|1111|2222|2222|4444|4444|3333|3333]
--> pid(e) Page Table(AID) : [1111|1111|1111|2222|2222|3333|3333|4444|4444|5555|5555|6666|6666|6666|6666|-----|-----]
--> pid(e) Page Table(Valid) : [0000|0000|-----|-----|1111|1111|1111|1111|0000|0000|0000|0000|-----|-----|-----|-----|-----]

[24 Cycle] Input : Pid [0] Function [RELEASE] Alloc ID [6] Page Num[0]
--> Physical Memory : [1111|1111|1111|-----|-----|4444|4444|3333|3333]
--> pid(e) Page Table(AID) : [1111|1111|1111|-----|-----|3333|3333|4444|4444|5555|5555|-----|-----|-----|-----|-----]
--> pid(e) Page Table(Valid) : [0000|0000|-----|-----|1111|1111|1111|1111|0000|0000|0000|0000|-----|-----|-----|-----|-----]

[25 Cycle] Input : Pid [0] Function [ACCESS] Alloc ID [3] Page Num[8]
--> Physical Memory : [1111|1111|-----|-----|4444|4444|3333|3333]
--> pid(e) Page Table(AID) : [1111|1111|-----|-----|3333|3333|4444|4444|5555|5555|-----|-----|-----|-----|-----]
--> pid(e) Page Table(Valid) : [0000|0000|-----|-----|1111|1111|1111|1111|0000|0000|0000|0000|-----|-----|-----|-----|-----]

[26 Cycle] Input : Pid [0] Function [ACCESS] Alloc ID [1] Page Num[8]
--> Physical Memory : [1111|1111|-----|-----|4444|4444|3333|3333]
--> pid(e) Page Table(AID) : [1111|1111|-----|-----|3333|3333|4444|4444|5555|5555|-----|-----|-----|-----|-----]
--> pid(e) Page Table(Valid) : [0000|0000|-----|-----|1111|1111|1111|1111|0000|0000|0000|0000|-----|-----|-----|-----|-----]

page Fault = 9
```

## 2-<LRU-SAMPLED>

2-<MFU>

```
Linux_Ubuntu [실행 중] - Oracle VM VirtualBox  
파일 마신 보기 압력 장치 도움말  
현재 활동 텍스트 편집기 ▾  
문서 (1) memory(mfuj).txt  
[18 Cycle] Input : Pfd [0] Function [ACCESS] Alloc ID [8] Page Num[8]  
[Physical Memory] : [1111|1111|1111|1111|1111|1111|1111|1111]  
[pid(0)] Page Table (AID) : [1111|1111|1111|2222|2222|3333|3333]  
[pid(0)] Page Table (Valid) : [1111|1111|1111|1111|0000|0000|1111|1111]  
[19 Cycle] Input : Pfd [0] Function [ACCESS] Alloc ID [4] Page Num[8]  
[Physical Memory] : [1111|1111|1111|1111|1111|1111|1111|1111]  
[pid(0)] Page Table (AID) : [1111|1111|1111|2222|2222|3333|3333]  
[pid(0)] Page Table (Valid) : [0000|0000|1111|1111|1111|1111|1111|1111]  
[20 Cycle] Input : Pfd [0] Function [ACCESS] Alloc ID [3] Page Num[8]  
[Physical Memory] : [1111|1111|1111|1111|1111|1111|1111|1111]  
[pid(0)] Page Table (AID) : [1111|1111|1111|2222|2222|3333|3333]  
[pid(0)] Page Table (Valid) : [0000|0000|1111|1111|1111|1111|1111|1111]  
[21 Cycle] Input : Pfd [0] Function [ALLOCATION] Alloc ID [6] Page Num[12]  
[Physical Memory] : [1111|1111|1111|1111|1111|1111|1111|1111]  
[pid(0)] Page Table (AID) : [1111|1111|1111|2222|2222|3333|3333]  
[pid(0)] Page Table (Valid) : [0000|0000|1111|1111|1111|1111|1111|1111]  
[22 Cycle] Input : Pfd [0] Function [RELEASE] Alloc ID [2] Page Num[0]  
[Physical Memory] : [1111|1111|1111|1111|1111|1111|1111|1111]  
[pid(0)] Page Table (AID) : [1111|1111|1111|2222|2222|3333|3333]  
[pid(0)] Page Table (Valid) : [0000|0000|1111|1111|1111|1111|1111|1111]  
[23 Cycle] Input : Pfd [0] Function [RELEASE] Alloc ID [0] Page Num[12]  
[Physical Memory] : [1111|1111|1111|1111|1111|1111|1111|1111]  
[pid(0)] Page Table (AID) : [1111|1111|1111|2222|2222|3333|3333]  
[pid(0)] Page Table (Valid) : [0000|0000|1111|1111|1111|1111|1111|1111]  
[24 Cycle] Input : Pfd [0] Function [RELEASE] Alloc ID [0] Page Num[0]  
[Physical Memory] : [1111|1111|1111|1111|1111|1111|1111|1111]  
[pid(0)] Page Table (AID) : [1111|1111|1111|2222|2222|3333|3333]  
[pid(0)] Page Table (Valid) : [0000|0000|1111|1111|1111|1111|1111|1111]  
[25 Cycle] Input : Pfd [0] Function [ACCESS] Alloc ID [3] Page Num[8]  
[Physical Memory] : [1111|1111|1111|1111|1111|1111|1111|1111]  
[pid(0)] Page Table (AID) : [1111|1111|1111|2222|2222|3333|3333]  
[pid(0)] Page Table (Valid) : [0000|0000|1111|1111|1111|1111|1111|1111]  
[26 Cycle] Input : Pfd [0] Function [ACCESS] Alloc ID [3] Page Num[8]  
[Physical Memory] : [1111|1111|1111|1111|1111|1111|1111|1111]  
[pid(0)] Page Table (AID) : [1111|1111|1111|2222|2222|3333|3333]  
[pid(0)] Page Table (Valid) : [1111|1111|1111|1111|1111|1111|1111|1111]  
page fault = 12
```

2-<OPTIMAL>

<INPUT3 (ex2)>

The screenshot shows a terminal window titled "Input" with the command "cat /proc/meminfo | grep MemFree" run. The output is as follows:

```
MemFree: 2848 1024 32  
program1
```

Below this, there is a large block of memory dump output from "/proc/kcore" with the following header:

```
(일) 19:56 * [MemFree] Input : Pfd [0] Function [ACCESS] Alloc ID [3] Page Num[8]  
MemFree: 17 31  
1 1  
2 1  
6 6  
8 8  
0 8  
0 8  
0 8  
1 2  
1 3  
1 4  
1 5  
2 2  
2 3  
2 4  
2 5  
0 30  
0 0  
|
```

3-<FIFO>

3-<LFU>

3-<LRU>

```
Linux_Ubuntu [실행 중] - Oracle VM VirtualBox

파일 마신 보기 앱력 장치 도움말

현재 활동 텍스트 편집기 □

[0] (21) 21:14 •
memory1.txt
memory1segment.ec2
저장(S)

[0] Cycle] Input : Pid [0] Function [ACCESS] Alloc ID [5] Page Num[8]
--> Physical Memory : [2222|2222|3333|3333|4444|4444|5555|5555]-----[----|----|----|----|----|----|----|----]
--> pid(0) Page Table(AID) : [2222|2222|3333|3333|4444|4444|5555|5555]-----[----|----|----|----|----|----|----|----]
--> pid(0) Page Table(Valid) : [1111|1111|1111|1111|0000|0000|0000|0000]-----[----|----|----|----|----|----|----|----]

[10] Cycle] Input : Pid [0] Function [ACCESS] Alloc ID [4] Page Num[8]
--> Physical Memory : [2222|2222|3333|3333|4444|4444|5555|5555]-----[----|----|----|----|----|----|----|----]
--> pid(0) Page Table(AID) : [2222|2222|3333|3333|4444|4444|5555|5555]-----[----|----|----|----|----|----|----|----]
--> pid(0) Page Table(Valid) : [1111|1111|1111|1111|0000|0000|0000|0000]-----[----|----|----|----|----|----|----|----]

[11] Cycle] Input : Pid [0] Function [ACCESS] Alloc ID [3] Page Num[8]
--> Physical Memory : [2222|2222|3333|3333|4444|4444|5555|5555]-----[----|----|----|----|----|----|----|----]
--> pid(0) Page Table(AID) : [2222|2222|3333|3333|4444|4444|5555|5555]-----[----|----|----|----|----|----|----|----]
--> pid(0) Page Table(Valid) : [1111|1111|1111|1111|0000|0000|0000|0000]-----[----|----|----|----|----|----|----|----]

[13] Cycle] Input : Pid [0] Function [RELEASE] Alloc ID [3] Page Num[8]
--> Physical Memory : [1111|1111|1111|1111|0000|0000|0000|0000]-----[----|----|----|----|----|----|----|----]
--> pid(0) Page Table(AID) : [1111|1111|1111|1111|0000|0000|0000|0000]-----[----|----|----|----|----|----|----|----]
--> pid(0) Page Table(Valid) : [1111|1111|1111|1111|0000|0000|0000|0000]-----[----|----|----|----|----|----|----|----]

[14] Cycle] Input : Pid [0] Function [RELEASE] Alloc ID [4] Page Num[8]
--> Physical Memory : [1111|1111|1111|1111|0000|0000|0000|0000]-----[----|----|----|----|----|----|----|----]
--> pid(0) Page Table(AID) : [1111|1111|1111|1111|0000|0000|0000|0000]-----[----|----|----|----|----|----|----|----]
--> pid(0) Page Table(Valid) : [1111|1111|1111|1111|0000|0000|0000|0000]-----[----|----|----|----|----|----|----|----]

[15] Cycle] Input : Pid [0] Function [RELEASE] Alloc ID [5] Page Num[8]
--> Physical Memory : [1111|1111|1111|1111|0000|0000|0000|0000]-----[----|----|----|----|----|----|----|----]
--> pid(0) Page Table(AID) : [1111|1111|1111|1111|0000|0000|0000|0000]-----[----|----|----|----|----|----|----|----]
--> pid(0) Page Table(Valid) : [0000|0000|0000|0000|0000|0000|0000|0000]-----[----|----|----|----|----|----|----|----]

[16] Cycle] Input : Pid [0] Function [ALLOCATION] Alloc ID [6] Page Num[30]
--> Physical Memory : [1111|1111|1111|1111|0000|0000|0000|0000]-----[----|----|----|----|----|----|----|----]
--> pid(0) Page Table(AID) : [0000|0000|0000|0000|0000|0000|0000|0000]-----[----|----|----|----|----|----|----|----]
--> pid(0) Page Table(Valid) : [0000|0000|0000|0000|0000|0000|0000|0000]-----[----|----|----|----|----|----|----|----]

[17] Cycle] Input : Pid [0] Function [ALLOCATION] Alloc ID [7] Page Num[30]
--> Physical Memory : [1111|1111|1111|1111|0000|0000|0000|0000]-----[----|----|----|----|----|----|----|----]
--> pid(0) Page Table(AID) : [0000|0000|0000|0000|0000|0000|0000|0000]-----[----|----|----|----|----|----|----|----]
--> pid(0) Page Table(Valid) : [0000|0000|0000|0000|0000|0000|0000|0000]-----[----|----|----|----|----|----|----|----]

page Fault = 5
```

### 3-<LRU-SAMPLED>

3-<MFU>

3-<OPTIMAL>

(EXPONENTIAL)	기본	input1	input2	input3
FIFO	9	28	13	5
LRU	9	28	8	5
SAMPLE-LRU	7	24	9	5
LFU	9	26	8	5
MFU	8	27	12	5
OPTIMAL	7	21	10	5
CYCLE 숫자	33	32	26	17

<기본이라고 적혀 인풋은 과제설명에 나와 있는 예시 인풋입니다.>

스케줄링 알고리즘에 따라 명령어 수행 순서가 달라질 수 있어서 저는 한가지 스케줄링 방식인 sjf-exponential을 기준으로 수행시켜보았습니다.

메모리 관리기법을 보시면 인풋에 따라 변화하는 정도가 많이 차이가 있습니다. 하나도 변하지 않는 인풋이 존재하기도 하고 많이 변하는 인풋도 역시 존재합니다. 그렇다면 이러한 결과의 차이가 나는 이유로는 어떤 것이 영향을 주었을지 찾아보았습니다. 전체 cycle의 숫자, 혹은 평균적으로 일어나는 page fault의 숫자가 있었습니다. 전체 cycle의 숫자는 어떻게 보면 영향이 있어보이기도 하고 아니기도 합니다. 33개가 있을 때 일어나는 변화율과 26개가 있을 때 일어나는 변화율이 오히려 26에서 많기 때문입니다. 또한 cycle의 전체 숫자 역시 input1과 input2를 비교했을 때는 많은 차이가 있어보이지만 기본과 input2를 비교했을 때는 그렇지 않기 때문입니다.

즉 결과적으로 정확한 전체 cycle의 숫자, 혹은 평균적으로 일어나는 page fault의 숫자와 각각의 페이지 교체 알고리즘에 따른 page fault의 숫자는 정확한 상관관계를 알아내는 것이 어려웠습니다. 또한 스케줄러를 어떤 것을 쓰느냐에 따라도 영향을 받을 것이고 각각의 프로세스의 op와 org에 따라 차이가 있을 것이기 때문입니다.

하지만 optimal의 구현 방식인 모든 op와 org를 알고 나서 하는 페이지 교체 알고리즘은 거의 대부분 좋은 성적을 보여주었습니다. 이처럼 많은 요소들이 결과에 영향을 주기 때문에 그때 그때에 따라 알맞은 페이지 교체 알고리즘이 있을 것이고 그것을 정하는 것이 어려울거 같습니다.

## 4. 과제 수행 시 겪었던 어려움과 해결 방법

### 4-1 segmentation fault와 core dump

주로 발생했던 수많은 문제들의 대부분은 아무래도 segmentation fault와 core dump가 아니었을까 생각합니다. 이유로는 데이터값들을 다루고 배열이나 벡터에 저장하고 불러오고 하는 과정에서 잘못된 주소 값을 인용하거나 데이터를 넣는 경우에 발생하였습니다. 이러한 문제점들은 하나하나 출력을 해보면서 디버깅을 통해서 고쳤습니다.

### 4-2 파일 입출력 문제

파일 입출력 문제는 해당 주어진 파일을 읽고 그 적혀있는 글자의 이름으로 되는 파일을 열어야 하는데 계속 같다고 인식하지 못하는 문제였습니다. 이 문제는 출력했을 때는 원래 열어야 하는 파일과 같게 보였지만 길이를 출력해봤을 때 보이지 않는 글자들이 양옆으로 숨어있었습니다. 그러면 이유로 각각의 string에서 양옆으로 한글자씩을 지워 문제를 해결하였습니다.