

# Assignment 3

2015198005

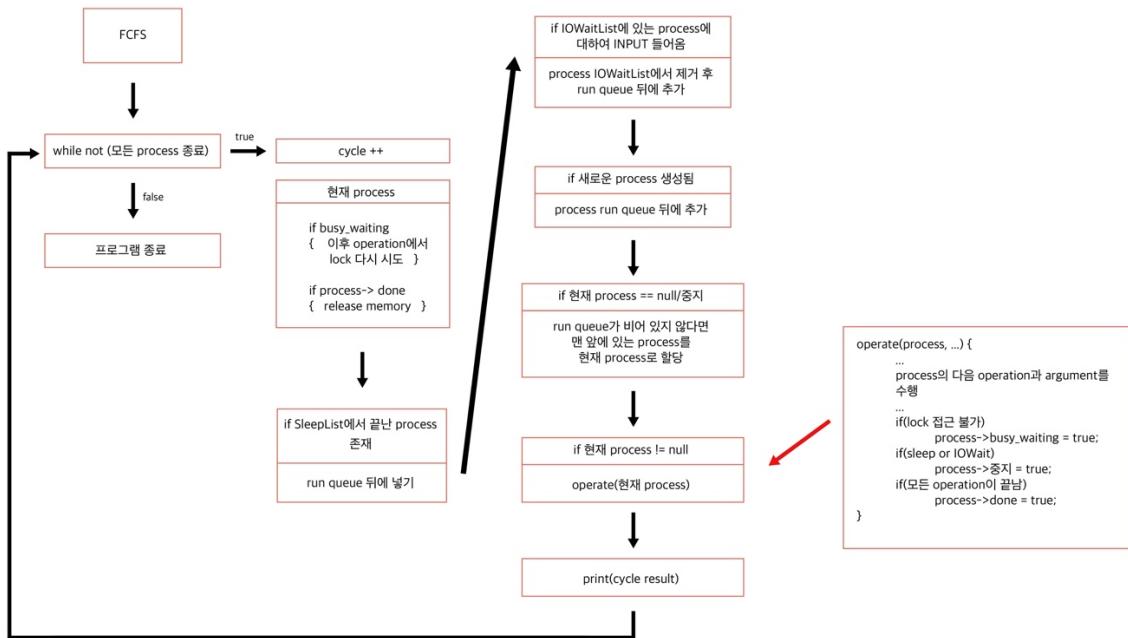
허진욱

## 1. 과제 수행 결과 보고서

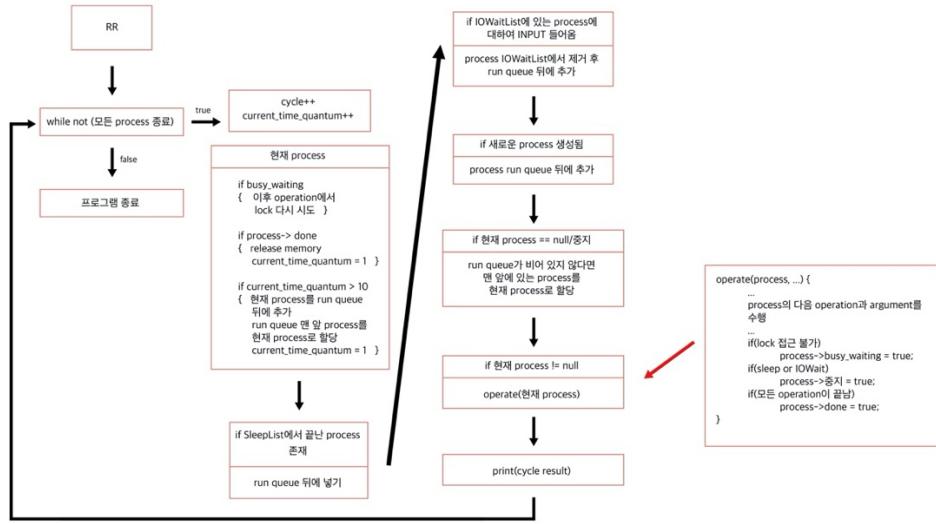
### 1) 작성한 프로그램의 동작 과정과 구현 방법

#### a) Schedulers

##### i) FCFS(First-come First-served)

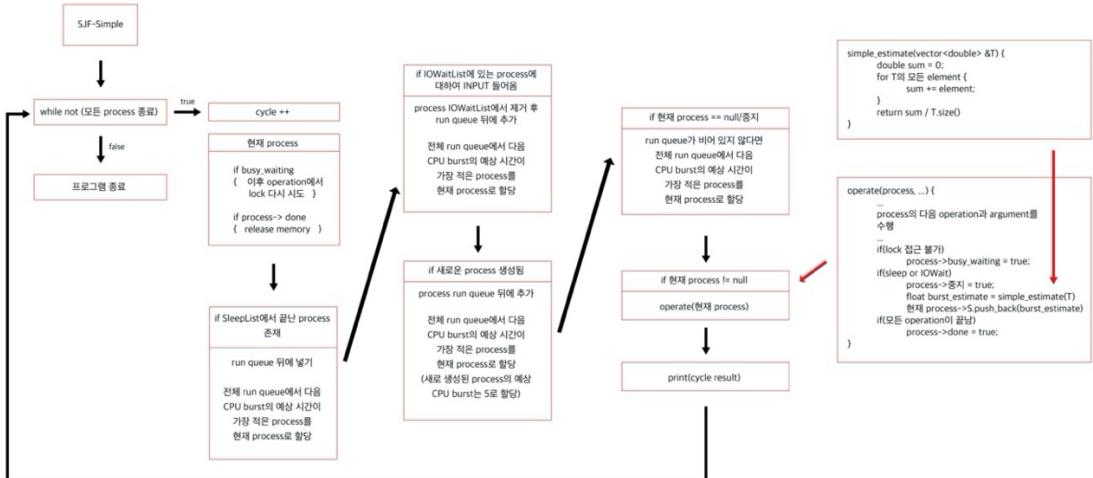


## ii) RR(Round Robin)



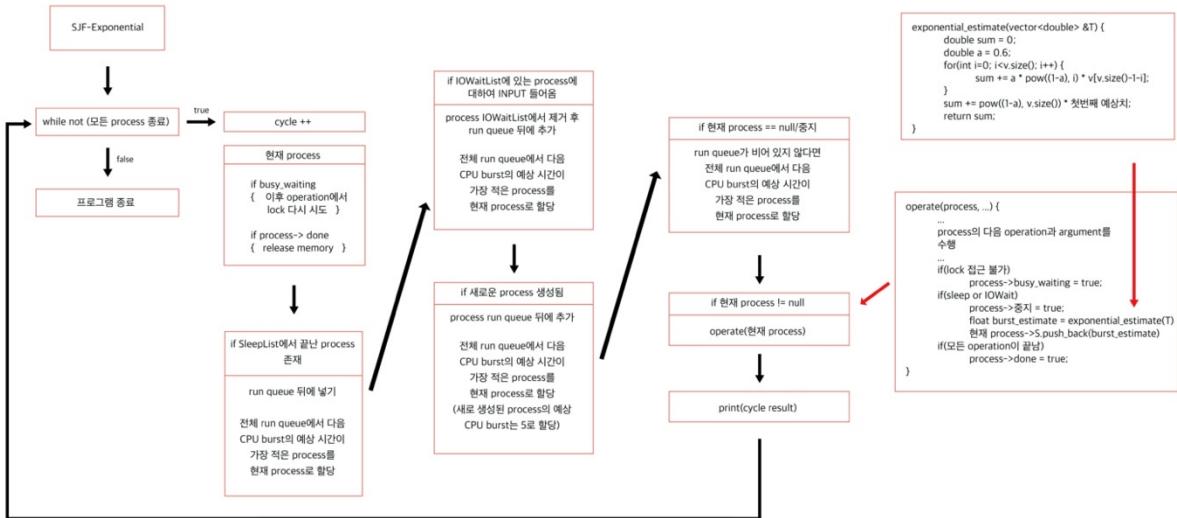
RR(Round Robin)방식의 스케줄링은 run queue 에 먼저 들어온 순서대로 CPU 에 할당 된다는 점에서 FCFS 와 유사하지만 한 process 가 최대 time quantum 인 10 cycle 동안 CPU 에 있었으면 다시 run queue 뒤로 간다는 점에서 preemptive 한 방식이다. 한 cycle 이 시작하면 current\_quantum 이 1 씩 증가하고, 프로세스 교체가 일어나는 순간이 오면 다시 1 로 초기화한다.

## iii) SJF(Shortest Job First)-Simple average



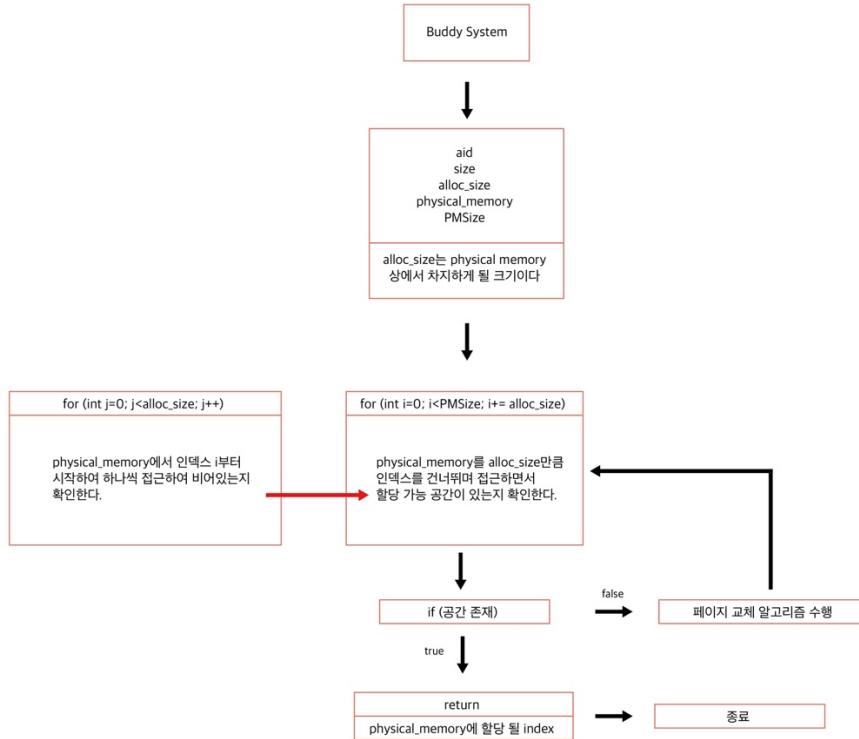
새로운 process 를 run queue 에서 선택할 때, 예상 CPU burst 가 가장 짧은 process 를 선택하는 방식이다. Cycle 이 시작하고 SleepList 나 I/O WaitList 에서 run queue 로 돌아온 process 가 있으면 그 즉시 run queue 내에서 새 process 를 선택한다. operate() 함수 내에서 sleep 이나 i/o wait operation 이 발생하면 이전까지 발생했던 모든 CPU burst 를 단순 평균 내어 새로운 예상 시간을 계산 한다.

#### iv) SJF(Shortest Job First)-Exponential average



앞선 방식과 마찬가지로 새로운 process 를 run queue 에서 선택할 때, 예상 CPU burst 가 가장 짧은 process 를 선택하는 방식이다. Cycle 이 시작하고 SleepList 나 I/O WaitList 에서 run queue 로 돌아온 process 가 있으면 그 즉시 run queue 내에서 새 process 를 선택한다. operate() 함수 내에서 sleep 이나 i/o wait operation 이 발생하면 이전까지 발생했던 모든 CPU burst 를 토대로 새로운 예상 시간을 계산 한다. 이때 Simple average 와는 달리, 최근에 일어난 CPU burst 에 가중치를 부여하는 방식으로 계산을 한다.

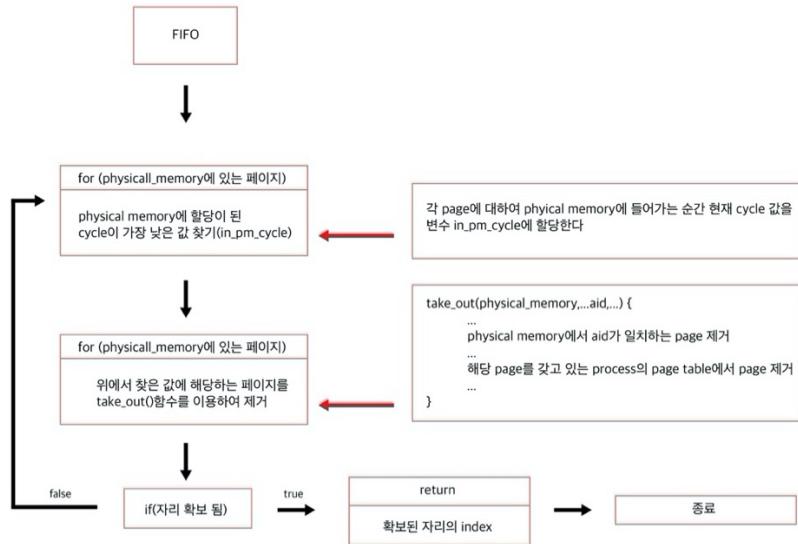
## b) Buddy system



physical memory에 페이지를 올려놓기 전에 할 우선 총 페이지 크기보다 크면서 가장 가까운 2의 지수값을 alloc\_size에 할당한다. 이후 페이지를 physical memory에 올릴 때 alloc\_size 크기만큼 인덱스를 건너 뛰며 자리가 있는지 확인한다. 자리가 있다면 해당 자리에 첫번째 인덱스를 리턴하고, 만약 없다면 페이지 교체 알고리즘을 수행 한 후 다시 자리 확인을 한다.

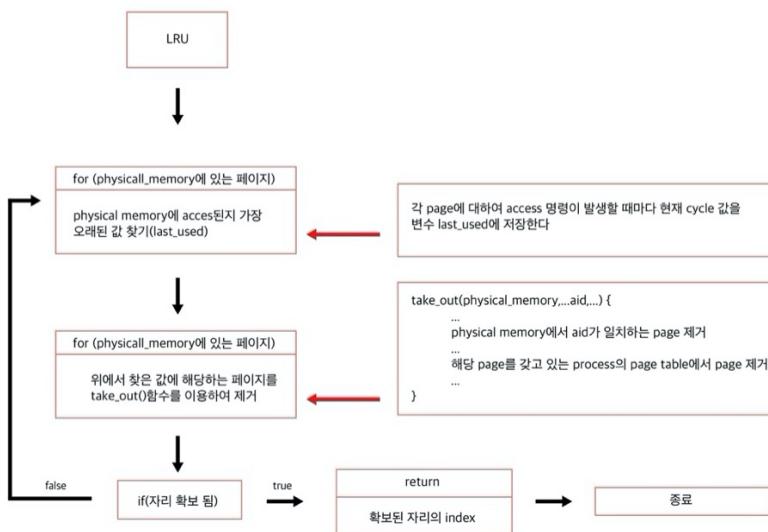
### c) Page Replacement Algorithms

i) FIFO



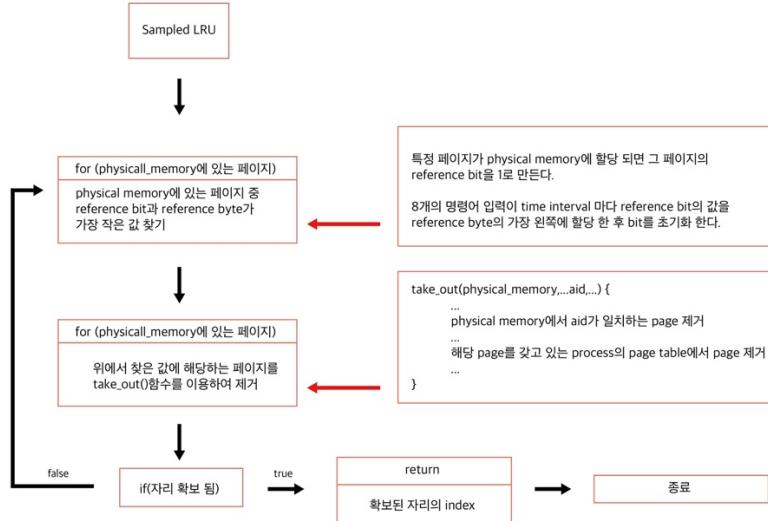
FIFO 알고리즘은 physical memory에 가장 먼저 들어온 페이지가 빠지는 방식이다. 각 페이지가 physical memory에 할당된 시점을 변수 `im_pm_cycle`에 저장한 후, 이후 교체하는 시점에 이 변수 값이 가장 작은 페이지가 교체 된다.

ii) LRU



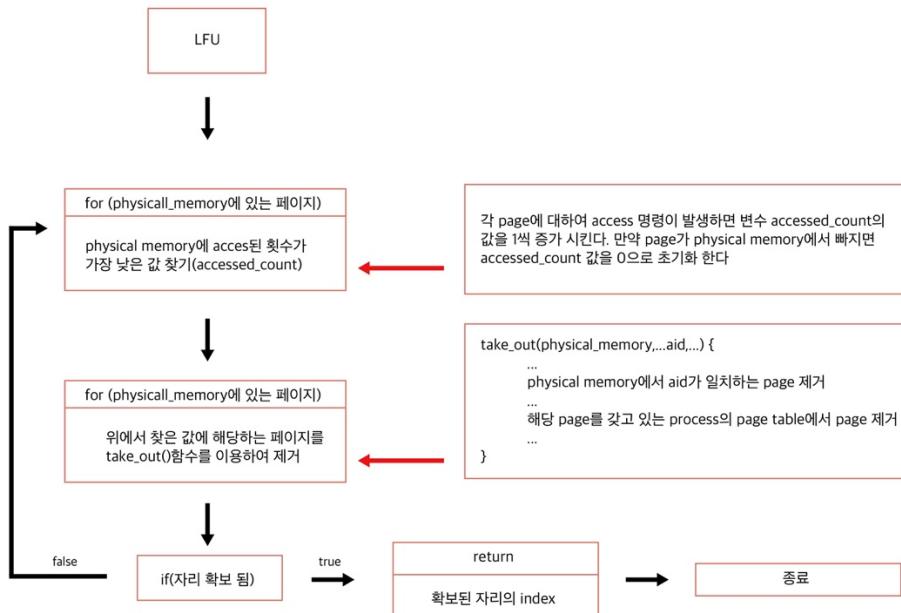
마지막으로 참조된 시점이 가장 먼 페이지가 교체 되는 알고리즘이다. 페이지가 참조 될 때마다 해당 시점을 변수 `last_used`에 업데이트 해준다.

### iii) Sampled LRU



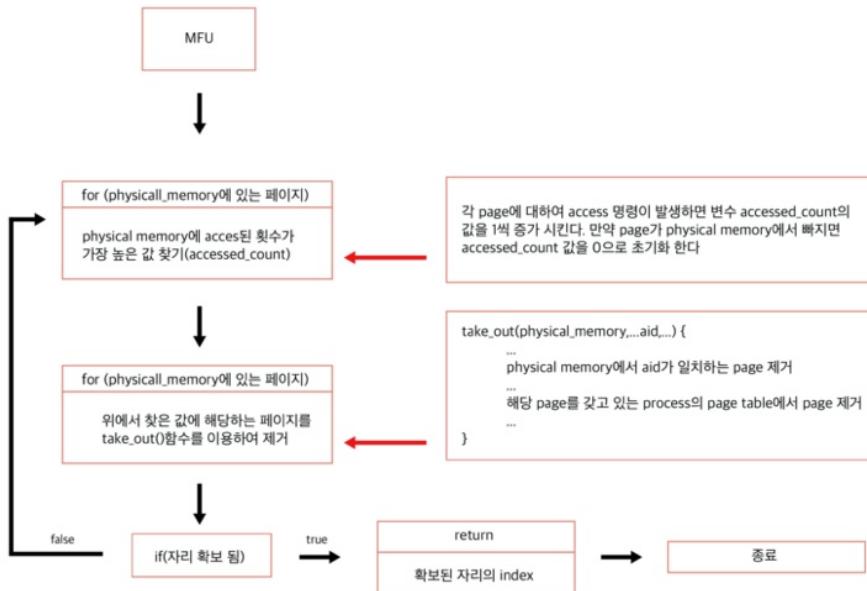
Overhead 가 많이 발생한다는 기존 LRU 의 문제점을 보완한 알고리즘이다. 특정 페이지가 physical memory 에 할당 되면 해당 페이지의 reference bit 을 1 로 만든다. Time interval 이 되면 physical memory 에 있는 페이지들의 reference bit 을 각각의 reference byte 의 맨 왼쪽에 추가한다. 페이지 교체가 일어날 때, reference bit 과 reference byte 의 값이 가장 작은 페이지가 교체 된다.

### iv) LFU



참조된 횟수가 가장 적은 페이지가 교체되는 알고리즘이다. 각 페이지에 대하여 access 명령이 발생할 때마다, 변수 accessed\_count 을 1 씩 증가시킨다. 이후 이 값이 가장 작은 페이지가 교체된다.

### v) MFU



참조된 횟수가 가장 많은 페이지가 교체되는 알고리즘이다. 각 페이지에 대하여 access 명령이 발생할 때마다, 변수 accessed\_count 을 1 씩 증가시킨다. 이후 이 값이 가장 큰 페이지가 교체된다.

## 2) 결과 화면과 토의 내용

## a) scheduler.txt

## Input 1

input

```
20    4096   2048   32
1     program1
3     program2
5     INPUT    0
10    program3
12    program4
20    program5
22    INPUT    2
40    program6
70    program7
80    INPUT    1
100   program8
110   program9
142   program10
240   INPUT    9
250   INPUT    8
1000  INPUT    2
1200  INPUT    9
1300  INPUT    0
1400  INPUT    1
1500  INPUT    8
```

## program1

## program2

## program3

## program4

## program5

## program6

33

## program7

## program8

## program9

## program10

### i) FCFS 결과

```
[1502 Cycle] Scheduled Process: None
Running Process: Process#8 running code program9 line 155(op 3, arg 0)
RunQueue: Empty
SleepList: Empty
IOWait List: Empty

Average Waiting Time: 832.000000
```

### ii) RR 결과

```
[1502 Cycle] Scheduled Process: None
Running Process: Process#8 running code program9 line 155(op 3, arg 0)
RunQueue: Empty
SleepList: Empty
IOWait List: Empty

Average Waiting Time: 878.000000
```

### iii) SJF-Simple average 결과

```
[1502 Cycle] Scheduled Process: None
Running Process: Process#8 running code program9 line 155(op 3, arg 0)
RunQueue: Empty
SleepList: Empty
IOWait List: Empty

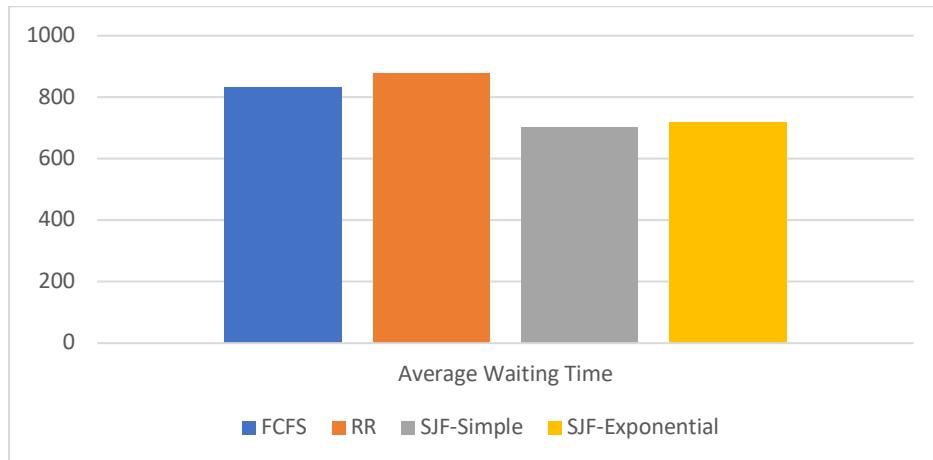
Average Waiting Time: 703.000000
```

### iv) SJF-Exponential average 결과

```
[1502 Cycle] Scheduled Process: None
Running Process: Process#8 running code program9 line 155(op 3, arg 0)
RunQueue: Empty
SleepList: Empty
IOWait List: Empty

Average Waiting Time: 718.000000
```

### v) Result



프로그램 실행 결과 Average Waiting Time 이 가장 낮은 스케줄링 방식은 SJF-Simple Average 방식이다. Simple average 방식에 비해 Exponential average 방식이 미미하게 성능이 안좋았는데 그 이유는 다음 CPU burst 를 예측하는 과정에서 최근 burst 에 가중치를 부여한 것이 오히려 오차가 큰 예측값을 도출하게 만들었기 때문이다. 더 나아가 인풋 값으로 주어진 프로세스들이 일관성 없이 block 이 되게 만들어졌다는 사실 또한 추론할 수 있다.

### Input 2

input

```
4      2048    1024     32
1      program1
2      program2
10     program3
50     INPUT    0
```

## program1

## program2

## program3

### i) FCFS 결과

```
[188 Cycle] Scheduled Process: None
Running Process: Process#1 running code program2 line 60(op 3, arg 0)
RunQueue: Empty
SleepList: Empty
IOWait List: Empty

Average Waiting Time: 80.000000
```

## ii) RR 결과

```
[188 Cycle] Scheduled Process: None  
Running Process: Process#0 running code program1 line 62(op 3, arg 0)  
RunQueue: Empty  
SleepList: Empty  
IOWait List: Empty  
  
Average Waiting Time: 90.000000
```

### iii) SJF-Simple average 결과

```
[193 Cycle] Scheduled Process: None
Running Process: Process#2 running code program3 line 66(op 3, arg 0)
RunQueue: Empty
SleepList: Empty
IOWait List: Empty

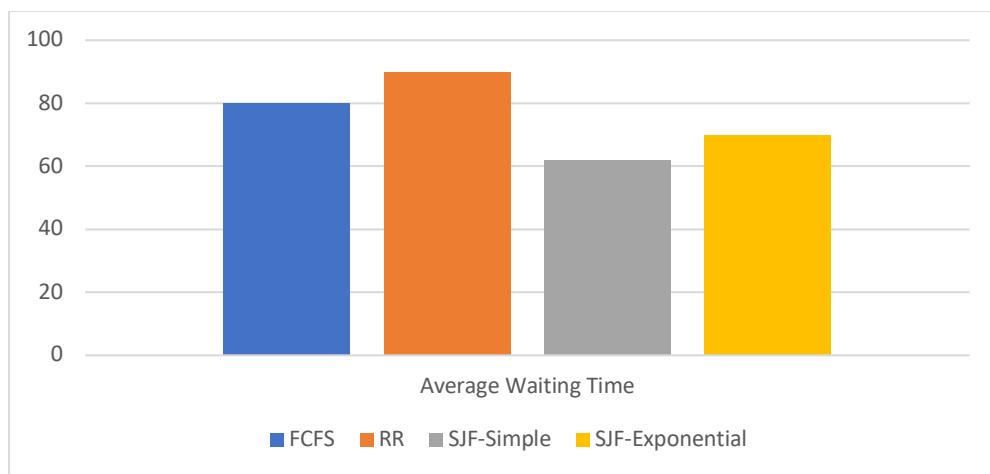
Average Waiting Time: 62.000000
```

### iv) SJF-Exponential average 결과

```
[188 Cycle] Scheduled Process: None
Running Process: Process#2 running code program3 line 66(op 3, arg 0)
RunQueue: Empty
SleepList: Empty
IOWait List: Empty

Average Waiting Time: 70.000000
```

### v) Result



프로그램 실행 결과 input 1 과 비교하여 거의 비슷한 결과가 나왔다. input 1에 비하여 FCFS 와 RR 의 차이가 더 많이 발생하였다. RR 방식을 사용하면 특정 프로세스의 CPU 독점을 막을 수 있지만, FCFS 방식이라면 일찍 실행하고 종료될 프로세스들이 늦게까지 run queue 에 남아 있게 되어 이러한 결과가 나온 것으로 보인다.

## Input 3

input

6	2048	1024	32
1	program1		
10	program2		
15	program3		
50	program4		
150	INPUT 1		
420	INPUT 2		

## program1

## program2

### program3

## program4

### i) FCFS 결과

```
[425 Cycle] Scheduled Process: None  
Running Process: Process#2 running code program3 line 107(op 3, arg 0)  
RunQueue: Empty  
SleepList: Empty  
IOWait List: Empty  
  
Average Waiting Time: 205.000000
```

## ii) RR 결과

```
[425 Cycle] Scheduled Process: None
Running Process: Process#2 running code program3 line 107(op 3, arg 0)
RunQueue: Empty
SleepList: Empty
IOWait List: Empty

Average Waiting Time: 240.000000
```

## iii) SJF-Simple average 결과

```
[425 Cycle] Scheduled Process: None
Running Process: Process#2 running code program3 line 107(op 3, arg 0)
RunQueue: Empty
SleepList: Empty
IOWait List: Empty

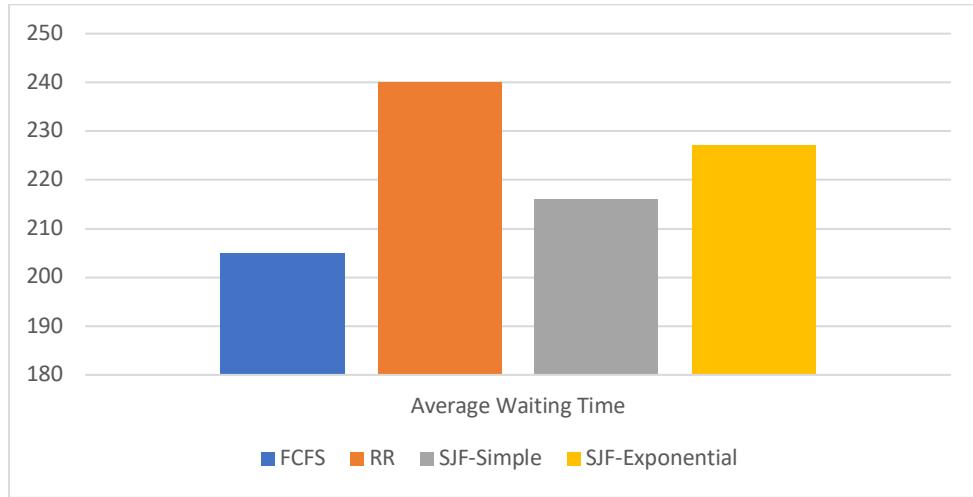
Average Waiting Time: 216.000000
```

## iv) SJF-Exponential average 결과

```
[425 Cycle] Scheduled Process: None
Running Process: Process#2 running code program3 line 107(op 3, arg 0)
RunQueue: Empty
SleepList: Empty
IOWait List: Empty

Average Waiting Time: 227.000000
```

### v) Result



앞선 두 인풋과는 달리 FCFS 의 성능이 가장 좋았다. 이러한 결과가 나온 가장 큰 이유는 제일 먼저 생성된 program1 매우 늦은 시간까지 종료되지 않고 run queue에 남아있기 때문이다. program1은 sleep 과 i/o wait 명령을 수행하는 횟수가 비교적 적다. 때문에 예상 CPU burst 시간이 매우 크게 계산되고 그로 인해 CPU 가 다음 프로세스를 선택할 때 우선순위에서 밀리게 된다. 이를 통해, 절대적으로 우수한 스케줄링 방식은 존재하지 않고, 프로그램의 특성에 따라 가장 적합한 방식을 선택해야 한다는 것을 알 수 있다.

### b) memory.txt

#### Input 1

input

```
3      2048    1024    32
1      program1
3      program2
10     program3
```

### program1

71				
0	16	3	0	
0	8	3	0	
0	2	3	0	
0	4	1	4	
0	10	1	5	
4	2	1	2	
1	3	1	1	
1	4	1	2	
1	2	1	4	
1	4	4	2	
1	2	1	3	
1	5	1	3	
1	3	1	1	
1	1	1	3	
3	0	1	3	
3	0	1	2	
1	4	1	1	
1	1	3	0	
1	3	3	0	
1	4	3	0	
1	4	3	0	
1	2	3	0	
3	0	4	2	
3	0	3	0	
1	1	3	0	
1	3	3	0	
1	5	1	4	
3	0	1	5	

### program2

58				
0	5	1	6	
0	10	1	7	
4	1	1	7	
1	6	1	7	
1	7	1	6	
1	7	1	7	
1	6	1	7	
1	7	1	7	
3	0	1	6	
3	0	4	1	
3	0	1	6	
3	0	1	7	
3	0	1	6	
1	6	1	7	
1	7	1	6	
1	6	1	7	
1	6	3	0	
1	7	3	0	
1	7	3	0	
1	6	3	0	
3	0	3	0	
3	0	1	7	
3	0	1	6	
1	6	1	7	
4	2	2	7	
1	6	2	6	

### program3

5	7	10	1	9
0	0	4	1	9
0	4	5	1	8
4	1	8	4	2
1	1	9	1	8
1	1	8	1	9
1	1	8	1	9
3	3	0	1	9
3	3	0	1	9
1	1	9	1	8
1	1	9	1	9
4	1	1	3	0
1	1	8	3	0
1	1	9	3	0
1	1	9	3	0
1	1	9	3	0
1	1	9	3	0
3	3	8	1	8
3	3	0	1	8
3	3	0	1	8
1	1	9	1	9
1	1	9	1	8
1	1	8	1	8
1	1	9	1	9
3	3	0	1	8
3	3	0	4	1
3	3	0	1	8
3	3	0	1	9

### i) FIFO 결과

## ii) LRU 결과

### iii) Sampled LRU 결과

#### iv) LFU 결과

```
[186 Cycle] Input : PID [2] Function [ACCESS] Alloc ID [9] Page Num [4]
>> Physical Memory :      |9999|---|---|---|8888|8888|8888|8888|
>> pid(2) Page Table(AID) :  |8888|8888|8899|99--|---|---|---|---|---|---|---|---|---|
>> pid(2) Page Table(VALID) : |1111|1111|1111|11--|---|---|---|---|---|---|---|---|---|
page fault = 58
```

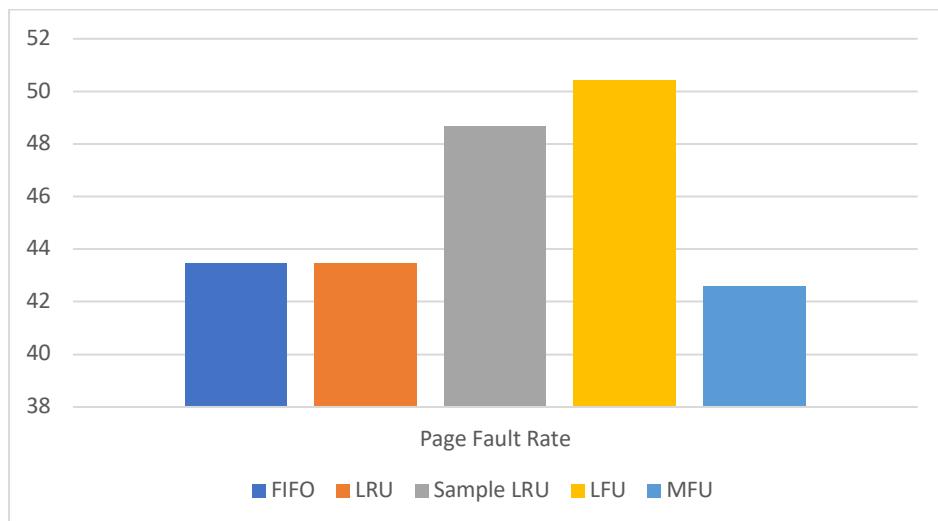
#### v) MFU 결과

```
[186 Cycle] Input : PID [2] Function [ACCESS] Alloc ID [9] Page Num [4]
>> Physical Memory :      |---|---|9999|---|8888|8888|8888|8888|
>> pid(2) Page Table(AID) :  |8888|8888|8899|99--|---|---|---|---|---|---|---|---|---|
>> pid(2) Page Table(VALID) : |1111|1111|1111|11--|---|---|---|---|---|---|---|---|---|
page fault = 49
```

#### vi) Result

**Page Fault Rate** = page fault / total number of access operations

Algorithm	Page Fault Rate
FIFO	43.47%
LRU	43.47%
Sampled LRU	48.69%
LFU	50.43%
MFU	42.60%



MFU 와 LRU 가 가장 높은 성능을 보이고 LFU 가 가장 안좋은 성능을 보인다. MFU 가 LFU 에 비해 성능이 월등히 좋은 이유는 각 페이지에 대한 access 횟수가 비교적 고르게 발생하기 때문이다. 이미 접근 횟수가 많은 페이지는 앞으로 접근 될 확률이 적으므로 MFU 알고리즘이 높은 성능을 보인다. LRU 와 Sampled LRU 의 차이가 확연하게 크기 때문에, overhead 가 많더라도 LRU 알고리즘을 사용하는 것이 바람직하다.

## Input 2

input

```
3      2048    1024     32
1      program1
3      program2
10     program3
```

program1

```
36
0      16
0      8
0      4
4      1
1      2
1      3
1      1
1      2
1      3
1      3
1      2
1      3
1      1
1      3
1      1
1      2
1      2
1      3
1      1
4      1      1      1
1      2      1      2
1      3      1      3
1      2      1      1
1      2      1      3
1      3      1      1
1      1      1      3
1      3      1      1
```

program2

43			
0	5		
0	10		
4	1		
1	4		
1	5		
1	5		
1	4		
1	4		
1	5		
1	5		
1	4		
1	4		
1	4		
1	5		
1	5		
1	4		
1	5	1	4
1	4	1	4
1	4	1	4
1	5	1	5
1	4	1	5
1	4	1	4
1	5	1	5
1	4	1	4
1	4	1	4
4	2	1	4
1	5	1	5
1	4	1	4
1	5	1	5
1	4	1	4
1	5	1	4

## program3

37	10		
0	4		
0	1		
4	6		
1	7		
1	6		
1	6		
1	7		
1	6		
1	7		
1	7		
1	6		
1	7		
1	7		
1	6		
1	6		
1	7		
1	6		
4	2		
1	6		
1	7		
1	7		
1	6		
1	7		
1	6		
1	7		
1	7		
1	6		
1	7		
1	7		
1	6		
1	4		
1	1		
1	2		
1	6		
1	6		
1	7		
1	1		
1	1		
1	7		
1	6		
1	7		
1	6		
1	6		

### i) FIFO 결과

## ii) LRU 결과

```
[117 Cycle] Input : PID [2] Function [ACCESS] Alloc ID [6] Page Num [10]
>> Physical Memory :      |7777|----|---|---|6666|6666|6666|6666|
>> pid(2) Page Table(AID) : |6666|6666|6677|77--|---|---|---|---|---|---|---|
>> pid(2) Page Table(VALID) : |1111|1111|1111|11--|---|---|---|---|---|---|---|
page fault = 26
```

## iii) Sampled LRU 결과

```
[117 Cycle] Input : PID [2] Function [ACCESS] Alloc ID [6] Page Num [10]
>> Physical Memory :      |6666|6666|6666|6666|7777|----|---|---|
>> pid(2) Page Table(AID) : |6666|6666|6677|77--|---|---|---|---|---|---|
>> pid(2) Page Table(VALID) : |1111|1111|1111|11--|---|---|---|---|---|---|
page fault = 35
```

## iv) LFU 결과

```
[117 Cycle] Input : PID [2] Function [ACCESS] Alloc ID [6] Page Num [10]
>> Physical Memory :      |----|---|---|7777|6666|6666|6666|6666|
>> pid(2) Page Table(AID) : |6666|6666|6677|77--|---|---|---|---|---|---|
>> pid(2) Page Table(VALID) : |1111|1111|1111|11--|---|---|---|---|---|---|
page fault = 25
```

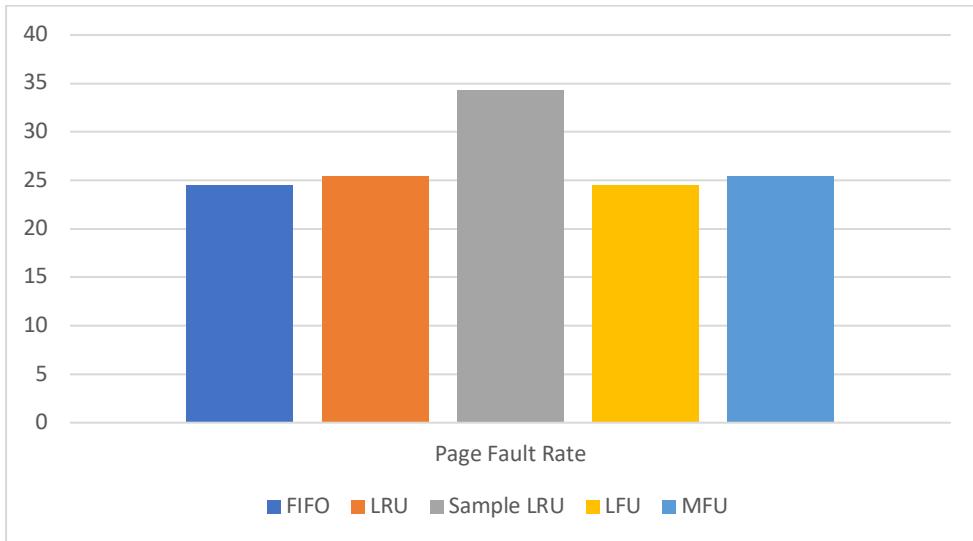
## v) MFU 결과

```
[117 Cycle] Input : PID [2] Function [ACCESS] Alloc ID [6] Page Num [10]
>> Physical Memory :      |6666|6666|6666|6666|7777|----|---|---|
>> pid(2) Page Table(AID) : |6666|6666|6677|77--|---|---|---|---|---|---|
>> pid(2) Page Table(VALID) : |1111|1111|1111|11--|---|---|---|---|---|---|
page fault = 26
```

### vi) Result

**Page Fault Rate** = page fault / total number of access operations

Algorithm	Page Fault Rate
FIFO	24.50%
LRU	25.49%
Sampled LRU	34.31%
LFU	24.50%
MFU	25.49%



Sampled LRU 를 제외하고는 대부분의 알고리즘이 비슷한 성능을 보인다. 근소하지만 더 나은 성능을 보이는 FIFO 나 LFU 알고리즘을 사용하는 것이 바람직해 보인다.

### Input 3

input

```
3      2048    1024     32
1      program1
3      program2
10     program3
```

## program1

34			
0	16		
0	8		
0	4		
4	1		
1	3		
3	0		
3	0		
3	0		
3	0		
3	0		
3	0		
3	0		
3	0		
1	2		
3	0		
3	0		
3	0		
3	0		
3	0		
3	0		
3	0		
3	0		
3	0		
3	0		
3	0		
1	1		
3	0		
3	0		
3	0		
3	0		
3	0		
3	0		
4			1
3			0
3			0
3			0
3			0
3			0

## program2

26	5
0	10
0	1
4	4
1	0
3	0
3	0
3	0
3	0
3	0
3	0
1	5
3	0
3	0
3	0
3	0
3	0
3	0
3	0
4	2
3	0
3	0
3	0
3	0
3	0

## program3

9  
0  
0  
0  
0  
0  
4  
1  
1  
1  
1

10  
4  
8  
5  
5  
6  
7  
8  
9

### i) FIFO 결과

```
[69 Cycle] Input : PID [0] Function [NONMEMORY]
>> Physical Memory :      |1111|1111|1111|1111|----|----|----|----|
>> pid(0) Page Table(AID) : |1111|1111|1111|1111|2222|2222|3333|----|----|----|----|----|----|
>> pid(0) Page Table(VALID) : |1111|1111|1111|1111|0000|0000|0000|----|----|----|----|----|----|
page fault = 9
```

### ii) LRU 결과

```
[69 Cycle] Input : PID [0] Function [NONMEMORY]
>> Physical Memory :      |1111|1111|1111|1111|----|----|----|----|
>> pid(0) Page Table(AID) : |1111|1111|1111|1111|2222|2222|3333|----|----|----|----|----|----|
>> pid(0) Page Table(VALID) : |1111|1111|1111|1111|0000|0000|0000|----|----|----|----|----|----|
page fault = 9
```

### iii) Sampled LRU 결과

```
[69 Cycle] Input : PID [0] Function [NONMEMORY]
>> Physical Memory :      |1111|1111|1111|1111|----|----|----|----|
>> pid(0) Page Table(AID) : |1111|1111|1111|1111|2222|2222|3333|----|----|----|----|----|----|
>> pid(0) Page Table(VALID) : |1111|1111|1111|1111|0000|0000|0000|----|----|----|----|----|----|
page fault = 9
```

### iv) LFU 결과

```
[69 Cycle] Input : PID [0] Function [NONMEMORY]
>> Physical Memory :      |1111|1111|1111|1111|----|----|----|----|
>> pid(0) Page Table(AID) : |1111|1111|1111|1111|2222|2222|3333|----|----|----|----|----|----|
>> pid(0) Page Table(VALID) : |1111|1111|1111|1111|0000|0000|0000|----|----|----|----|----|----|
page fault = 9
```

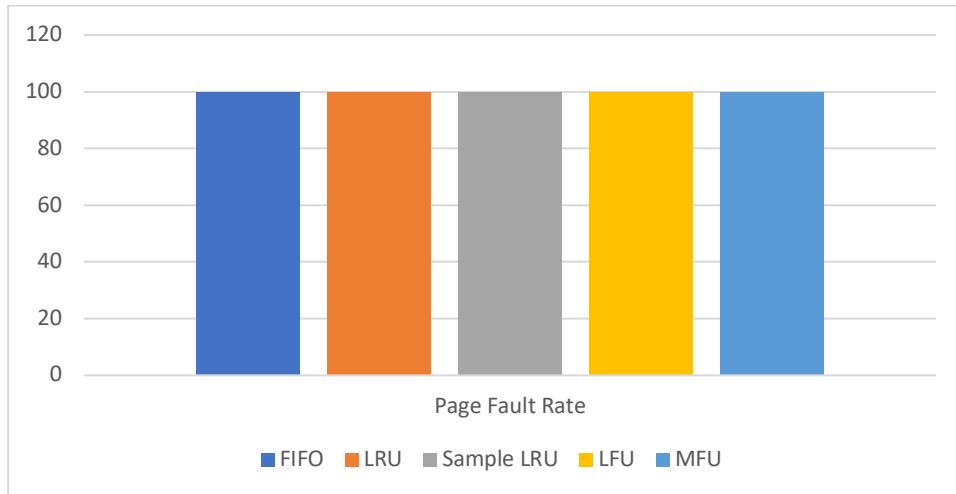
### v) MFU 결과

```
[69 Cycle] Input : PID [0] Function [NONMEMORY]
>> Physical Memory :      |1111|1111|1111|1111|----|----|----|----|
>> pid(0) Page Table(AID) : |1111|1111|1111|1111|2222|2222|3333|----|----|----|----|----|----|
>> pid(0) Page Table(VALID) : |1111|1111|1111|1111|0000|0000|0000|----|----|----|----|----|----|
page fault = 9
```

## vi) Result

**Page Fault Rate** = page fault / total number of access operations

Algorithm	Page Fault Rate
FIFO	100%
LRU	100%
Sampled LRU	100%
LFU	100%
MFU	100%



모든 페이지가 동일하게 한번씩 access 되었기 때문에 Page Fault Rate 가 모두 100%다.  
이런 경우에는 어떠한 알고리즘을 사용하여도 무방하다.

### 3) 개발 환경 명시

i) uname -a 실행 결과

```
jinwook@jinwook-ubuntu:~$ uname -a
Linux jinwook-ubuntu 5.4.33-2015198005 #1 SMP Tue Apr 21 03:29:54 KST 2020 x86_64
x86_64 x86_64 GNU/Linux
```

ii) 사용한 컴파일러 버전

```
gcc version 7.5.0 (Ubuntu 7.5.0-3ubuntu1~18.04)
```

iii) CPU 코어 수

```
jinwook@jinwook-ubuntu:~$ grep -c processor /proc/cpuinfo
4
```

iv) 운영체제 버전

```
jinwook@jinwook-ubuntu:~$ cat /etc/issue
Ubuntu 18.04.4 LTS \n \l
```

v) 메모리 정보

```
jinwook@jinwook-ubuntu:~$ cat /proc/meminfo
MemTotal:       2034964 kB
MemFree:        126004 kB
MemAvailable:   836692 kB
Buffers:        152484 kB
Cached:         635900 kB
SwapCached:     1212 kB
Active:          925428 kB
Inactive:       706352 kB
Active(anon):   461028 kB
Inactive(anon): 397584 kB
Active(file):   464400 kB
Inactive(file): 308768 kB
Unevictable:    16 kB
Mlocked:        16 kB
SwapTotal:      1942896 kB
SwapFree:       1918400 kB
Dirty:           0 kB
Writeback:       0 kB
AnonPages:      842864 kB
Mapped:          149988 kB
Shmem:          15216 kB
KReclaimable:   125416 kB
Slab:            171024 kB
SReclaimable:   125416 kB
SUnreclaim:     45608 kB
KernelStack:    9056 kB
PageTables:     38736 kB
NFS_Unstable:   0 kB
Bounce:          0 kB
WritebackTmp:   0 kB
CommitLimit:    2960376 kB
Committed_AS:   4529480 kB
VmallocTotal:   34359738367 kB
VmallocUsed:    45980 kB
VmallocChunk:   0 kB
Percpu:          1296 kB
HardwareCorrupted: 0 kB
AnonHugePages:   0 kB
ShmemHugePages:  0 kB
ShmemPmdMapped: 0 kB
FileHugePages:   0 kB
FilePmdMapped:  0 kB
CmaTotal:        0 kB
CmaFree:         0 kB
HugePages_Total: 0
HugePages_Free:  0
HugePages_Rsvd:  0
HugePages_Surp:  0
Hugepagesize:    2048 kB
Hugetlb:         0 kB
DirectMap4k:    163776 kB
DirectMap2M:    1933312 kB
```

## 4) 과제 수행 중 발생한 애로사항 및 해결방법

### i) Input file 만들기

지난 과제와는 달리 프로그램의 테스트를 위한 인풋을 만드는 과정이 굉장히 까다로웠다. 일정한 규칙의 랜덤 숫자들을 인풋 값으로 하면, faulty input 으로 인해 프로그램이 에러가 발생하게 된다. 때문에 직접 수작업으로 정교하게 인풋을 만드는 과정이 필요했고 이 단계에서 많은 시간이 소요되었다.

### ii) RR 스케줄링

RR 스케줄링에서 프로세스가 Time quantum 이 모두 끝났을 때, 해당 cycle 의 명령이 sleep이나 I/O wait 인 경우 run queue 에도 들어가고 sleep / IO wait list 에도 들어가게 되었다. 또한 print 를 하는 과정에서도 미리 run queue 에 넣고 다음 프로세스를 선택하면 현재 running process 에서 해당 프로세스가 사라졌다.  
이를 해결하기 위해 time quantum 이 다 되어 새로운 프로세스를 선택하는 과정을 한 사이클의 마지막 그리고 처음, 둘 단계로 나누어 구현하여 해결하였다.

Cycle 뒷부분

```
if(next_p != nullptr) {  
    if(current_quantum>9 && !next_p->end) {      //time quantum finished  
        run_queue.push_back(next_p);      //push to run queue  
    }  
}
```

다음 Cycle 의 앞부분

```
if(next_p != nullptr) {  
    if(current_quantum>10 && !next_p->end) {      //time quantum finished  
        next_p = nullptr;  
    }  
}
```