

운영체제 실습

[Assignment4]

Class : [B]

Professor : [최상호 교수님]

Student ID : [2021202089]

Name : [오나균]

Introduction

Assignment4 는 Assignment 4-1 과 Assignment 4-2 두 부분으로 구성된다.

먼저 Assignment 4-1 에서는 프로세스의 PID 를 입력값으로 받아 해당 프로세스의 메모리 영역 정보를 출력하는 Kernel Module 을 구현한다. 이를 위해 2 차 과제에서 구현하였던 ftrace 시스템 콜(번호 336)을 기반으로 새로운 시스템콜을 wrapping 하며, hooking 함수의 이름은 file_varea 로 정의한다. 출력해야 하는 정보는 다음과 같다.

- 프로세스 이름과 PID
- 해당 프로세스의 메모리 정보가 위치하는 가상 메모리 주소
- 데이터 영역 주소, 코드 영역 주소, 힙(Heap) 영역 주소
- 해당 메모리 정보의 원본이 되는 파일의 전체 경로

Assignment 4-2 에서는 Page Replacement 알고리즘 시뮬레이터를 User-level program 으로 구현한다.

프로그램은 입력 파일로부터 사용 가능한 Page Frame 수와 Page reference string 을 읽어온 뒤, 다음의 네 가지 page replacement 알고리즘을 수행한다.

- Optimal (OPT)
- FIFO (First In, First Out)
- LRU (Least Recently Used)
- Clock (Second-Chance) Algorithm

각 알고리즘은 동일한 Page reference string 과 Page Fault 의 발생 횟수에 대해서 Page Fault Rate 를 계산하여 비교한다.

결과화면

Assignment4-1

Assignment4-1 디렉토리 생성 및 이동

```
os2021202089@ubuntu:~$ mkdir Assignment4-1
os2021202089@ubuntu:~$ cd Assignment4-1
```

file_varea.c 작성

```
os2021202089@ubuntu:~/Assignment4-1$ vi file_varea.c
```

```
os2021202089@ubuntu: ~/Assignment4-1
#include <asm/ptrace.h> // struct pt_regs
#include <linux/sched.h>
#include <linux/module.h>
#include <linux/highmem.h>
#include <linux/sched/mm.h> // get_task_mm()
#include <linux/mm_types.h>
#include <linux/kallsyms.h>
#include <linux/init_task.h>
#include <linux/slab.h> // kmalloc , kfree
#include <linux/fs.h>
#include <linux/dcache.h> // d_path()
#include <linux/path.h>
#include <linux/limits.h> // PATH_MAX
#include <linux/mm.h> // vm_area_struct

void **syscall_table;
void *real_ftrace;
```

Makefile 작성

```
os2021202089@ubuntu:~/Assignment4-1$ vi Makefile
```

```
os2021202089@ubuntu: ~/Assignment4-1
obj-m += file_varea.o # 커널 모듈로 컴파일할 오브젝트 파일

KDIR := /lib/modules/$(shell uname -r)/build # 현재 시스템의 커널 빌드 시스템 위치
PWD := $(shell pwd) # 현재 모듈을 컴파일하는 디렉토리 위치

# 빌드
all:
    $(MAKE) -C$(KDIR) M=$(PWD) modules
# 빌드 생성물 제거
clean:
    $(MAKE) -C$(KDIR) M=$(PWD) clean

~
```

make 과정을 통해서 file_varea Module 을 빌드하여 , file_varea.ko 파일을 생성한다.

```
os2021202089@ubuntu:~/Assignment4-1$ make
make -C /lib/modules/5.4.282-os2021202089/build M=/home/os2021202089/Assignment4-1 modules
make[1]: Entering directory '/usr/src/linux-5.4.282'
CC [M] /home/os2021202089/Assignment4-1/file_varea.o
/home/os2021202089/Assignment4-1/file_varea.c: In function 'file_varea':
/home/os2021202089/Assignment4-1/file_varea.c:78:4: warning: ISO C90 forbids mixed declarations and code [-Wdeclaration-after-statement]
   78 |     char*path = d_path(&vma->vm_file->f_path,buf,PATH_MAX);
      |     ^~~~~
Building modules, stage 2.
MODPOST 1 modules
CC [M] /home/os2021202089/Assignment4-1/file_varea.mod.o
LD [M] /home/os2021202089/Assignment4-1/file_varea.ko
make[1]: Leaving directory '/usr/src/linux-5.4.282'
```

생성된 file_varea.ko 모듈을 삽입합니다.

```
os2021202089@ubuntu:~/Assignment4-1$ sudo insmod file_varea.ko
```

test.c 작성

```
os2021202089@ubuntu:~/Assignment4-1$ vi test.c
```

```
os2021202089@ubuntu: ~/Assignment4-1
#include <unistd.h>
#include <sys/types.h>
#include <linux/unistd.h>

#define __NR_ftrace 336
int main(void)
{
    syscall(__NR_ftrace,getpid());
    return 0;
}
```

test.c 컴파일 및 실행

```
os2021202089@ubuntu:~/Assignment4-1$ gcc test.c
os2021202089@ubuntu:~/Assignment4-1$ ./a.out
```

커널 버퍼 출력 및 확인

```
os2021202089@ubuntu:~/Assignment4-1$ dmesg
```

```
0512.017589] ##### Loaded files of a process 'a.out(11165)' in VM #####
0512.017513] code(56320e09b000-56320e09c215) data(56320e09edb0-56320e09f010) heap(56320e25f000-56320e25f000) /home/os2021202089/Assignment4-1/a.out
0512.017519] nen(56320e09b000-56320e09c000) code(56320e09b000-56320e09c215) data(56320e09edb0-56320e09f010) heap(56320e25f000-56320e25f000) /usr/lib/x86_64-linux-gnu/libc-2.31.so
0512.017522] nen(56320e09c000-56320e09d000) code(56320e09b000-56320e09c215) data(56320e09edb0-56320e09f010) heap(56320e25f000-56320e25f000) /home/os2021202089/Assignment4-1/a.out
0512.017525] nen(56320e09d000-56320e09e000) code(56320e09b000-56320e09c215) data(56320e09edb0-56320e09f010) heap(56320e25f000-56320e25f000) /usr/lib/x86_64-linux-gnu/libc-2.31.so
0512.017527] nen(56320e09e000-56320e09f000) code(56320e09b000-56320e09c215) data(56320e09edb0-56320e09f010) heap(56320e25f000-56320e25f000) /home/os2021202089/Assignment4-1/a.out
0512.017529] nen(56320e09f000-56320e0a0000) code(56320e09b000-56320e09c215) data(56320e09edb0-56320e09f010) heap(56320e25f000-56320e25f000) /home/os2021202089/Assignment4-1/a.out
0512.017532] nen(7f6911b25000-7f6911b47000) code(56320e09b000-56320e09c215) data(56320e09edb0-56320e09f010) heap(56320e25f000-56320e25f000) /usr/lib/x86_64-linux-gnu/libc-2.31.so
0512.017534] nen(7f6911b47000-7f6911cbf000) code(56320e09b000-56320e09c215) data(56320e09edb0-56320e09f010) heap(56320e25f000-56320e25f000) /usr/lib/x86_64-linux-gnu/libc-2.31.so
0512.017537] nen(7f6911cbf000-7f6911d0d000) code(56320e09b000-56320e09c215) data(56320e09edb0-56320e09f010) heap(56320e25f000-56320e25f000) /usr/lib/x86_64-linux-gnu/libc-2.31.so
0512.017539] nen(7f6911d0d000-7f6911d11000) code(56320e09b000-56320e09c215) data(56320e09edb0-56320e09f010) heap(56320e25f000-56320e25f000) /usr/lib/x86_64-linux-gnu/libc-2.31.so
0512.017542] nen(7f6911d11000-7f6911d13000) code(56320e09b000-56320e09c215) data(56320e09edb0-56320e09f010) heap(56320e25f000-56320e25f000) /usr/lib/x86_64-linux-gnu/libc-2.31.so
0512.017544] nen(7f6911d29000-7f6911d2a000) code(56320e09b000-56320e09c215) data(56320e09edb0-56320e09f010) heap(56320e25f000-56320e25f000) /usr/lib/x86_64-linux-gnu/libc-2.31.so
0512.017547] nen(7f6911d2a000-7f6911d4d000) code(56320e09b000-56320e09c215) data(56320e09edb0-56320e09f010) heap(56320e25f000-56320e25f000) /usr/lib/x86_64-linux-gnu/libc-2.31.so
0512.017549] nen(7f6911d4d000-7f6911d55000) code(56320e09b000-56320e09c215) data(56320e09edb0-56320e09f010) heap(56320e25f000-56320e25f000) /usr/lib/x86_64-linux-gnu/libc-2.31.so
0512.017551] nen(7f6911d55000-7f6911d57000) code(56320e09b000-56320e09c215) data(56320e09edb0-56320e09f010) heap(56320e25f000-56320e25f000) /usr/lib/x86_64-linux-gnu/libc-2.31.so
0512.017554] nen(7f6911d57000-7f6911d58000) code(56320e09b000-56320e09c215) data(56320e09edb0-56320e09f010) heap(56320e25f000-56320e25f000) /usr/lib/x86_64-linux-gnu/libc-2.31.so
```

file_varea 모듈을 통해 실행한 결과, PID 를 바탕으로 과제 요구사항에 맞게 VMA(Virtual Memory Area) 정보가 정상적으로 출력되었다.

Assignment 4-2

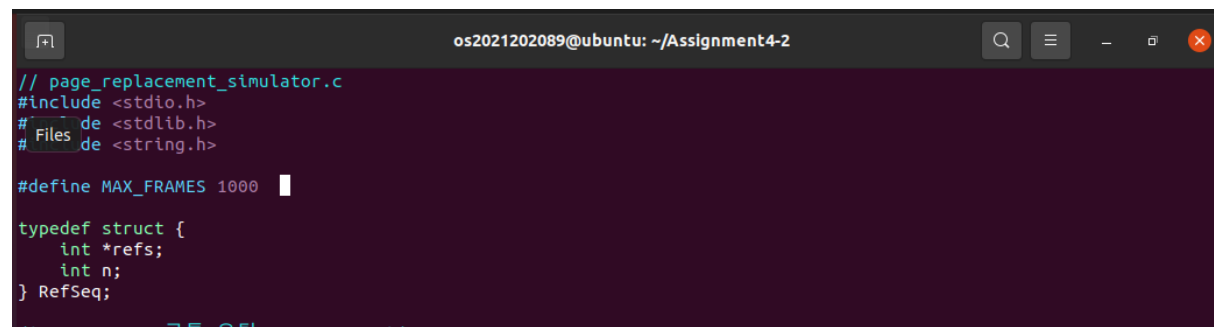
Assignment4-2 디렉토리 생성 및 이동

```
os2021202089@ubuntu:~$ mkdir Assignment4-2
```

```
os2021202089@ubuntu:~$ cd Assignment4-2
```

Page_replacement_simulator.c 작성

```
os2021202089@ubuntu:~/Assignment4-2$ vi page_replacement_simulator.c
```

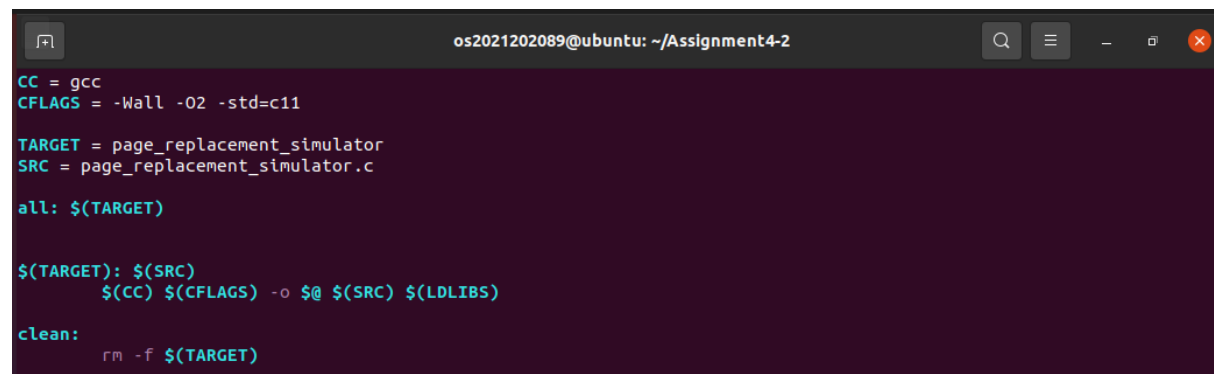


```
// page_replacement_simulator.c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define MAX_FRAMES 1000

typedef struct {
    int *refs;
    int n;
} RefSeq;
```

Makefile 작성



```
CC = gcc
CFLAGS = -Wall -O2 -std=c11

TARGET = page_replacement_simulator
SRC = page_replacement_simulator.c

all: $(TARGET)

$(TARGET): $(SRC)
    $(CC) $(CFLAGS) -o $@ $(SRC) $(LDLIBS)

clean:
    rm -f $(TARGET)
```

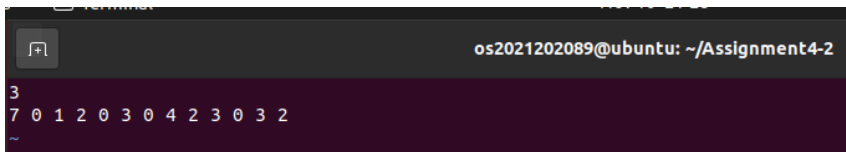
make

```
os2021202089@ubuntu:~/Assignment4-2$ make clean && make
rm -f page_replacement_simulator
gcc -Wall -O2 -std=c11 -o page_replacement_simulator page_replacement_simulator.c
```

Test case 수행

a. Test case 1 & 각 알고리즘 성능 분석

```
os2021202089@ubuntu:~/Assignment4-2$ vi input.1
```



```
3
7 0 1 2 0 3 0 4 2 3 0 3 2
```

```
os2021202089@ubuntu:~/Assignment4-2$ cat input.1
3
7 0 1 2 0 3 0 4 2 3 0 3 2
os2021202089@ubuntu:~/Assignment4-2$ ./page_replacement_simulator input.1
Optimal Algorithm:
Number of Page Faults: 7
Page Fault Rate: 53.85%

FIFO Algorithm:
Number of Page Faults: 10
Page Fault Rate: 76.92%

LRU Algorithm:
Number of Page Faults: 9
Page Fault Rate: 69.23%

Clock Algorithm:
Number of Page Faults: 9
Page Fault Rate: 69.23%
```

1. OPT 가 가장 낮은 Page Faults (7 회)를 기록하였다.

⇒ 이는 미래 참조 정보를 통해, 정확히 가장 손해가 적은 페이지를 교체하기 때문이다.

2. FIFO 의 성능이 가장 낮음

⇒ 해당 테스트 케이스는 초기 페이지들이 자주 재사용되지 않아서, Page Faults 가 10 회로 가장 높게 나옴.

3. LRU 와 Clock 은 동일한 성능

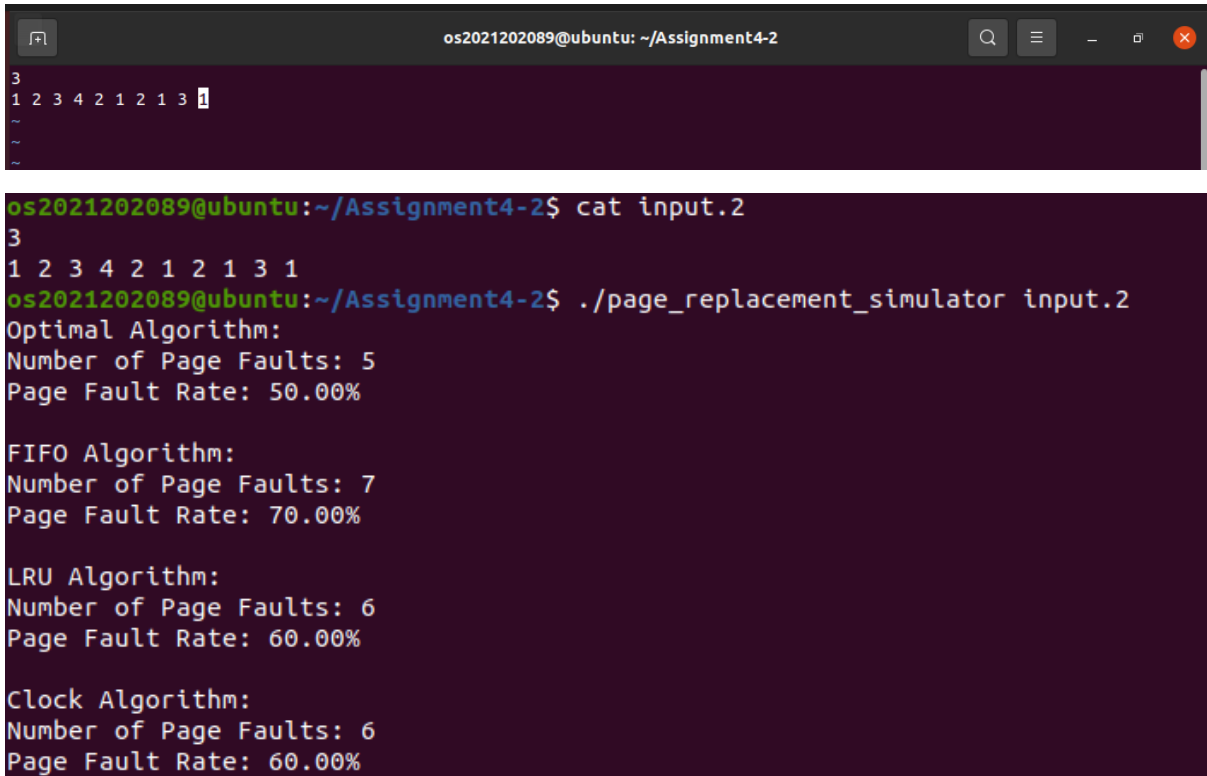
⇒ Reference bit 기반 Clock 이 LRU 와 같은 결과, 이는 LRU 가 오래된 페이지를 선택하는 알고리즘에서 차이를 만들지 못함을 의미한다.

➔ 정리하면, Input.1 에서는 $OPT < LRU = CLOCK < FIFO$ 순으로 우수한 성능을 보였다.

➔ 이는, STRIDE 가 넓고 다양한 페이지가 등장하는 케이스라, FIFO 가 가장 취약함을 의미한다.

b. Test case 2

```
os2021202089@ubuntu:~/Assignment4-2$ vi input.2
```



```
os2021202089@ubuntu:~/Assignment4-2$ cat input.2
3
1 2 3 4 2 1 2 1 3 1
os2021202089@ubuntu:~/Assignment4-2$ ./page_replacement_simulator input.2
Optimal Algorithm:
Number of Page Faults: 5
Page Fault Rate: 50.00%

FIFO Algorithm:
Number of Page Faults: 7
Page Fault Rate: 70.00%

LRU Algorithm:
Number of Page Faults: 6
Page Fault Rate: 60.00%

Clock Algorithm:
Number of Page Faults: 6
Page Fault Rate: 60.00%
```

1. OPT 는 Page Fault 가 5 번만 발생하고 가장 효율적이다.

- 초기 구간 1,2,3,4 를 제외하고, 이후 패턴은 반복성이 높아서 OPT 가 성능이 다른 모델에 비해 좋게 나옴을 알 수 있다.

2. FIFO 는 7 회로 성능이 가장 떨어진다.

- 초반 1,2,3, 상태에서 4 가 들어오며 교체가 되고, 이후 반복 패턴과도 잘 맞지 않음을 알 수 있다.

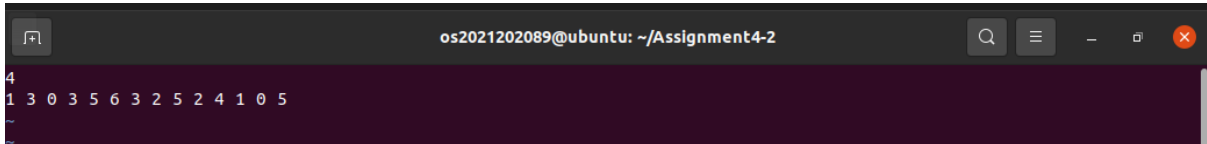
3. LRU 와 Clock 는 동일한 성능(6 회)

- 해당 입력은 반복되는 부분이 많아서, 최근에 자주 쓰인 페이지가 명확히 구분되어서, LRU 모델이 효과적이다.
- Clcok 도 참조 비트덕분에 LRU 와 같은 성능을 보인다.

➔ 반복되는 참조 패턴이 나타나면서 OPT 가 유리하고, LRU/Clock 성능이 잘 나옴을 알 수 있다. 하지만 FIFO 는 패턴을 고려하지 못해서, 가장 높은 Page Fault 가 발생한다.

c. Test case 3

```
os2021202089@ubuntu:~/Assignment4-2$ vi input.3
```



```
4
1 3 0 3 5 6 3 2 5 2 4 1 0 5
```

```
os2021202089@ubuntu:~/Assignment4-2$ cat input.3
4
1 3 0 3 5 6 3 2 5 2 4 1 0 5
os2021202089@ubuntu:~/Assignment4-2$ ./page_replacement_simulator input.3
Optimal Algorithm:
Number of Page Faults: 8
Page Fault Rate: 57.14%

FIFO Algorithm:
Number of Page Faults: 10
Page Fault Rate: 71.43%

LRU Algorithm:
Number of Page Faults: 10
Page Fault Rate: 71.43%

Clock Algorithm:
Number of Page Faults: 10
Page Fault Rate: 71.43%
```

1. OPT 가 가장 적은 Page Fault(8 회)

- 참조열이 길고 다양하지만, 미래 참조 기반 선택을 하는 OPT 가 가장 효율적으로 결과가 나옴을 알 수 있다.

2. FIFO / LRU / Clock 모두 동일한 Page Fault 수 (10 회)

- 이는 참조 패턴이 랜덤에 가까워서 Recency(최근성) 기반 전략이 크게 이득을 보지 못함을 알 수 있다.
- LRU 조차 오랫동안 활용된 페이지가 일정하지 않아서, victim 선택이 FIFO 와 비슷해짐을 알 수 있다.
- Clock도 Reference bit 만으로는 OPT 만큼의 성능이 나오지 않음을 알 수 있다.

➔ 페이지 사용 패턴이 불규칙적일 때는 FIFO, LRU, Clock 모두 비슷한 성능을 보인다.
이때 OPT 는 미래 정보를 알고 있어서, 가장 성능이 좋게 나옴을 알 수 있다.

각 알고리즘 종합 분석

1. 항상 OPT 가 가장 좋은 성능을 가진다.

- OPT 는 미래 참조를 완전히 알고 있으므로, 이론적으로 항상 최소의 Page Fault 값을 가진다.
- SSI(Page replacement theory)의 기준선(baseline) 역할을 함.

2. FIFO 는 대부분의 입력에서 최악의 성능을 가진다.

- 페이지의 실제 활용도를 고려하지 않는다.
- 오래 들어온 페이지부터 제거하는 방식은 비효율적임을 알 수 있다.
- Belady anomaly 가 나타날 수 있는 대표 알고리즘
- 세가지 테스트 케이스 모두 FIFO 가 가장 높은 Page Fault 를 기록하였다.

3. LRU 는 현실적인 대안으로 일정하게 좋은 성능을 가진다.

- 최근 사용된 페이지는 곧 다시 사용될 가능성이 높다는 locality 에 근거한다.
- 대부분의 경우 FIFO 보다 좋고, OPT 에 근접한 결과를 보인다.

4. Clock 은 LRU 보다 단순하지만 성능은 거의 동일하다.

- 실제 OS 는 LRU 를 직접 구현하기 어렵기 때문에 Clock 을 사용한다.
- 해당 과제의 테스트 케이스에서도 LRU 와 동일한 결과를 유지한다.
- 성능 대비 구현 복잡도가 LRU 보다는 낮다.

5. 테스트 케이스별 성능 패턴

입력	패턴특징	Page Faults
input.1	다양한 페이지 분포	OPT < LRU=Clock < FIFO
Input.2	반복되는 참조 패턴	OPT < LRU=Clock < FIFO
Input.3	불규칙/랜덤 패턴	OPT < FIFO=LRU=Clock

➔ Locality 가 강한 입력에서는 LRU/Clock 가 성능이 좋게 나옴을 알 수 있고,

➔ Locality 가 약한 랜덤형 입력에서는 FIFO/LRU/Clock 모두 유사한 결과

종합적인 결론

- OPT 는 비교 기준이 되는 이상적 알고리즘으로 항상 최고의 성능을 보인다.
- FIFO 는 Locality 를 고려하지 않으므로, 모든 케이스에서 가장 낮은 성능을 기록하였다.
- LRU 는 현실적인 알고리즘으로 안정적으로 높은 성능을 보여주었다.
- Clock 도 LRU 가 유사한 알고리즘으로 LRU 와 동일한 성능을 기록하였다.
-

고찰

이번 과제를 하면서 운영체제에서 메모리가 어떻게 관리되는지, 그리고 프로세스가 어떤 구조로 돌아가는지를 훨씬 더 이해할 수 있었다.

Assignment 4-1 에서는 커널 내부 주소들을 직접 출력해보면서, 이론시간에 개념으로만 배워서 실제로 수치를 확인 해보지 못했던 영역들(코드, 데이터, 힙)이 실제로 어디에 존재하는지 확인할 수 있어서 의미가 있었던 것 같다.

Assignment 4-2 에서는 시뮬레이터를 만들면서 각 페이지 교체 알고리즘이 얼마나 다르게 동작하는지 실험적으로 확인할 수 있었다.

같은 입력인데도 알고리즘마다 page fault 가 크게 달라지는 걸 보면서, 운영체제가 왜 locality 를 중요하게 여기는지 이해할 수 있었던 것 같다.

특히 Clock 알고리즘에서는 이동 규칙 하나 때문에 결과가 달라져서, 구현할 때 정확한 이해가 더 중요하다는 걸 알게된 것 같다.

중간중간 헛갈리는 부분도 많았지만, 직접 알고리즘을 시각화해보니 훨씬 쉽게 이해할 수 있었다.

이번 과제를 통해 단순히 이론을 외우는 것보다, 직접 구현하고 테스트해보는 경험이 훨씬 큰 도움이 된다는 걸 느꼈다.

무엇보다도, 운영체제의 내부 동작을 직접 코드를 통해 재현해봤다는 점이 가장 의미 있었다. 앞으로 다른 시스템 코드나 커널 관련 작업을 하게 되어도 이번 경험이 큰 밑바탕이 될 것 같다.

Reference

<https://chaaany.tistory.com/378> Belady Anomaly (페이지 교체시 FIFO 의 성능이 저하되는 이유)