운영체제

Assignment #4

Class : A

Professor : 김태석 교수님

Student ID : 2022202065

Name : 박나림

Introduction

이번 과제는 2 가지로 나누어서 진행하게 된다.

먼저 첫번째로, 기존 os_ftrace 시스템콜(336)을 hooking 하여 새로운 함수로 file_varea 이름으로 모듈을 작성하도록 한다. 해당 모듈은 pid 를 인자로 받아서 해당 프로세스의 정보가 위치하는 가상 메모리 주소, 데이터 주소, 코드 주소, 힙 주소, 원본 파일의 전체 경로를 출력한다. 이를 통해 가상 메모리와 물리 메모리에 대해 더 잘 이해할 수 있도록 한다.

두번째로는 4 개의 page replacement algorithms simulator 를 만들도록 한다. 각각의 알고리즘들을 구현하면서 페이지 교체와 page fault rate 에 대해 자세히 공부해보고, 각 알고리즘 별 성능 결과를 비교 분석하는 것이 목표이다.

결과화면

Assignment 4-1

Pid 를 인자로 받아서 해당 프로세스의 정보를 출력하는 module 을 작성한다. 해당 모듈은 기존의 os_ftrace 336 번 시스템콜을 wrapping 하여 file varea 로 hooking 하는 함수이다. 주소 정보를 출력하기 위해 task_struct, mm_struct, vm_area_struct 구조체를 사용하여 가상 메모리 영역 별로 순회하며 주소와 파일 경로를 출력하도록 한다.

```
asmlinkage pid_t file_varea(const struct pt_regs *regs) {
        struct task_struct *task;
        struct mm_struct *mm;
        struct vm_area_struct *vm;

        pid_t trace_task = (pid_t)regs->di; //get pid
        task = pid_task(find_vpid(trace_task), PIDTYPE_PID); //Find task by pid
        if (!task) { //can't find it
                return -1;
        }
        mm = get_task_mm(task); //get the memory management structure of the process


        //=================print process information================//
        printk(KERN_INFO "######## Loaded files of a process '%s(%d)' in VM ########\n", task->comm, task->pid); //process name, pid

        //print address where information is located
        for (vm = mm->mmap; vm; vm = vm->vm_next) { //start:mmap ~ each area
                if (vm->vm_file) { //When the actual file exists
                        char buf[256];
                        char *path = d_path(&vm->vm_file->f_path, buf, sizeof(buf));
                        if (!IS_ERR(path)) {
                                printk(KERN_INFO "mem(%lx-%lx) code(%lx-%lx) data(%lx-%lx) heap(%lx-%lx) %s\n", vm->vm_start, vm->vm_end, mm->start_code, mm->end_code, mm->start_data, mm->end_data
mm->start_brk, mm->brk, path);
                        }
                }
        }
        printk(KERN_INFO "#########################################################\n");

        mmput(mm); //mm structure reference end
        return trace_task; //Returns the entered pid
}
```

Makefile 과 test.c 파일도 작성한다.

```
obj-m += file_varea.o

PWD = $(CURDIR)

all:
        make -C /lib/modules/$(shell uname -r)/build M=$(PWD) modules

clean:
        make -C /lib/modules/$(shell uname -r)/build M=$(PWD) clean
```

```
1 #include <unistd.h>
2 #include <sys/types.h>
3 #include <linux/unistd.h>
4
5 #define __NR_os_ftrace 336
6
7 int main(void) {
8         syscall(__NR_os_ftrace, getpid());
9         return 0;
10 }
```

코드를 make 한 뒤 insmod 로 모듈을 적재한다.

```
os2022202065@ubuntu:~/Assignment4/Assignment4-1$ make
make -C /lib/modules/5.4.282-os2022202065/build M=/home/os2022202065/Assignm
make[1]: Entering directory '/usr/src/linux-5.4.282'
  CC [M]  /home/os2022202065/Assignment4/Assignment4-1/file_varea.o
  Building modules, stage 2.
  MODPOST 1 modules
  CC [M]  /home/os2022202065/Assignment4/Assignment4-1/file_varea.mod.o
  LD [M]  /home/os2022202065/Assignment4/Assignment4-1/file_varea.ko
make[1]: Leaving directory '/usr/src/linux-5.4.282'
os2022202065@ubuntu:~/Assignment4/Assignment4-1$ sudo insmod file_varea.ko
```

gcc test.c 로 테스트 파일을 make 하고 ./a.out 으로 실행한 다음 dmesg 로 결과를 확인한다.

```
os2022202065@ubuntu:~/Assignment4/Assignment4-1$ ./a.out
os2022202065@ubuntu:~/Assignment4/Assignment4-1$ dmesg
```

dmesg 로 확인해보면 아래와 같이 현재 실행되는 a.out 파일 이름과 pid 번호가 출력되고, 해당 프로세스의 정보가 위치하는 가상 메모리 주소들이 영역별로 경로와 함께 출력되는 것을 확인할 수 있다.

```
[ 4452.013766] ######## Loaded files of a process 'a.out(7471)' in VM ########
[ 4452.013773] mem(55991e139000~55991e13a000) code(55991e139000~55991e13a215) data(55991e13cdb0~55991e13d010) heap(55991f9a6000~55991f9a6000) /home/os2022202065/Assignment4/Assignment4-1/a.out
[ 4452.013776] mem(55991e13a000~55991e13b000) code(55991e139000~55991e13a215) data(55991e13cdb0~55991e13d010) heap(55991f9a6000~55991f9a6000) /home/os2022202065/Assignment4/Assignment4-1/a.out
[ 4452.013779] mem(55991e13b000~55991e13c000) code(55991e139000~55991e13a215) data(55991e13cdb0~55991e13d010) heap(55991f9a6000~55991f9a6000) /home/os2022202065/Assignment4/Assignment4-1/a.out
[ 4452.013781] mem(55991e13c000~55991e13d000) code(55991e139000~55991e13a215) data(55991e13cdb0~55991e13d010) heap(55991f9a6000~55991f9a6000) /home/os2022202065/Assignment4/Assignment4-1/a.out
[ 4452.013784] mem(55991e13d000~55991e13e000) code(55991e139000~55991e13a215) data(55991e13cdb0~55991e13d010) heap(55991f9a6000~55991f9a6000) /home/os2022202065/Assignment4/Assignment4-1/a.out
[ 4452.013787] mem(7fee999c2000~7fee999e4000) code(55991e139000~55991e13a215) data(55991e13cdb0~55991e13d010) heap(55991f9a6000~55991f9a6000) /usr/lib/x86_64-linux-gnu/libc-2.31.so
[ 4452.013790] mem(7fee999e4000~7fee99b5c000) code(55991e139000~55991e13a215) data(55991e13cdb0~55991e13d010) heap(55991f9a6000~55991f9a6000) /usr/lib/x86_64-linux-gnu/libc-2.31.so
[ 4452.013792] mem(7fee99b5c000~7fee99baa000) code(55991e139000~55991e13a215) data(55991e13cdb0~55991e13d010) heap(55991f9a6000~55991f9a6000) /usr/lib/x86_64-linux-gnu/libc-2.31.so
[ 4452.013795] mem(7fee99baa000~7fee99bae000) code(55991e139000~55991e13a215) data(55991e13cdb0~55991e13d010) heap(55991f9a6000~55991f9a6000) /usr/lib/x86_64-linux-gnu/libc-2.31.so
[ 4452.013797] mem(7fee99bae000~7fee99bb0000) code(55991e139000~55991e13a215) data(55991e13cdb0~55991e13d010) heap(55991f9a6000~55991f9a6000) /usr/lib/x86_64-linux-gnu/libc-2.31.so
[ 4452.013800] mem(7fee99bc6000~7fee99bc7000) code(55991e139000~55991e13a215) data(55991e13cdb0~55991e13d010) heap(55991f9a6000~55991f9a6000) /usr/lib/x86_64-linux-gnu/ld-2.31.so
[ 4452.013803] mem(7fee99bc7000~7fee99bea000) code(55991e139000~55991e13a215) data(55991e13cdb0~55991e13d010) heap(55991f9a6000~55991f9a6000) /usr/lib/x86_64-linux-gnu/ld-2.31.so
[ 4452.013805] mem(7fee99bea000~7fee99bf2000) code(55991e139000~55991e13a215) data(55991e13cdb0~55991e13d010) heap(55991f9a6000~55991f9a6000) /usr/lib/x86_64-linux-gnu/ld-2.31.so
[ 4452.013808] mem(7fee99bf3000~7fee99bf4000) code(55991e139000~55991e13a215) data(55991e13cdb0~55991e13d010) heap(55991f9a6000~55991f9a6000) /usr/lib/x86_64-linux-gnu/ld-2.31.so
[ 4452.013811] mem(7fee99bf4000~7fee99bf5000) code(55991e139000~55991e13a215) data(55991e13cdb0~55991e13d010) heap(55991f9a6000~55991f9a6000) /usr/lib/x86_64-linux-gnu/ld-2.31.so
[ 4452.013812] ############################################################
```

Assignment 4-2

4 종류의 Page replacement 알고리즘을 시뮬레이션하는 c 코드를 작성한다. main 에서 input 파일을 read 하여 frame 수와 page reference string 을 저장한 뒤, 각 알고리즘 함수로 전달한다. 전달받은 인자로 알고리즘을 동작시킨 뒤, page fault 수를 return 하여 main 에서 page fault rate 를 계산하여 출력한다.

1.Optimal (OPT)

page fault 이고 frame 이 가득 찼을 때, Optimal 방법으로 페이지를 교체한다. 주어진 reference string 배열을 봄으로써 미래를 예측, 즉 앞으로 가장 오랫동안 쓰이지 않을 page 를 선택하여 교체한다. 배열을 순회하며 furthestPage 를 update 하는 식으로 한다.

```
// Page fault
if (!pageExist) {
        optimalFault++;

        // empty frame
        if (rear < frame) {
                frameArr[rear++] = currPage;
        }
        else { // Optimal Policy
                int replacePage = 0; // Page number to be replaced
                int furthestPage = i+1; // furthest page number

                for (int j=0; j<frame; j++) { // j==frame
                        int next = 0; // Next Reference Position Distance
                        for (next=i+1; next<page; next++) {
                                if (frameArr[j]==referStr[next])
                                        break;
                        }

                        if (next == page) { // Not future referenced
                                replacePage = j;
                                break;
                        }

                        if (next > furthestPage) { // update to farthest
                                furthestPage = next;
                                replacePage = j;
                        }
                }
                frameArr[replacePage] = currPage;
```

2. FIFO (First In First Out)

가장 단순한 방법으로, page fault 시 원형 큐 방식으로 페이지를 추가한다. frame 이 비어있다면 그대로 들어가고, frame 이 비어있지 않다면 해당 자리에 page 가 교체된다. 가장 오래된 page 가 선입선출 방식으로 교체되는 방법이다.

```
// Page fault
if (!pageExist) {
        fifoFault++;
        // Circular Queue
        frameArr[rear] = currPage;
        rear = (rear+1) % frame;
```

3. LRU (Least Recently Used)

Page fault 이고 frame 이 가득 찼을 시, reference string 배열의 과거를 봄으로써 최근에 가장 오랫동안 참조되지 않은 page 를 교체시킨다. 이는 FIFO 와 반대로 stack 으로 구현될 수 있다. 처음에 bottom 부터 page 가 들어와서 top 에는 가장 최근 page 가 있게 된 후, 새로운 page 가 들어오려고 하면 기존 배열을 아래로 한 칸씩 밀고 새 page 를 top 에 저장한다. 또한 page hit 이면 해당 page 를 top 으로 옮기고 나머지는 다시 아래로 밀어준다. 이렇게 하면 가장 최근 참조된 page 는 top 에 있게 되며, 가장 오랫동안 참조되지 않은 page 는 맨아래에 있어서 빠져나가게 된다.

```
// Page fault
if (!pageExist) {
        lruFault++;

        // empty frame
        if (top < frame) {
                frameArr[top++] = currPage;
        }
        else { // full frame
                for (int j=0; j<top-1; j++) {
                        frameArr[j] = frameArr[j+1];
                }
                frameArr[top-1] = currPage;
        }
        /*
        printf("Page Fault %d: ", currPage);
        for (int k = 0; k < frame; k++) {
                printf("%d ", frameArr[k]);
        }
        printf("\n");
        */
}
else { // currPage hit
        for (int j=hitPage; j<top-1; j++) {
                frameArr[j] = frameArr[j+1];
        }
        frameArr[top-1] = currPage;
```

4. Clock

LRU 알고리즘을 변형시킨 방법으로, 최근 참조됨을 알리는 use bit 와 clock 한바퀴를 돌아가는 포인터인 clock hand 가 추가된다. page 를 검사할 때 hit 이면 pageExist 를 true 로 바꾸는건 다른 알고리즘과 동일하나, 여기서 추가로 useBit 을 1 로 하여 최근 참조됨을 알린다. page fault 일때만 clock hand 가 움직이면서 들어갈 frame 을 찾는다. frame 이 비었을 땐 바로 추가되며 새로 추가되는 page 의 useBit 는 0 이다. 이때 hand 는 다음 칸을 가리키게 된다. frame 이 가득 찼을 시엔 hand 가 useBit 이 0 인 page 를 가리킬 때까지 순회한다. 이때 useBit 이 1 이면 0 으로 바꾼다. (2 번의 기회를 준 셈) 0 이면 해당 page 를 교체하고 마찬가지로 hand 를 다음 칸으로 옮긴다. LRU 처럼 완전히 과거 전체를 보는 건 아니지만 주변에서 최근 참조됐는지 정도는 구분할 수 있는 알고리즘이다.

```
// Check page existence
for (int j=0; j<frame; j++) {
        if (currPage == frameArr[j].pageNum) {
                pageExist = true;
                frameArr[j].useBit = 1;

                printf("Page hit   %d: ", currPage);
                for (int k = 0; k < frame; k++) {
                        printf("%d(%d) ", frameArr[k].pageNum, frameArr[k].useBit);
                }
                printf("\n");

                break;
        }
}

// Page fault
if (!pageExist) {
        clockFault++;

        // empty frame
        if (tail < frame) {
                frameArr[tail].pageNum = currPage;
                frameArr[tail].useBit = 0;
                tail++;
        }
        else { // full frame
                while(1) { // Find and replace pages with use bit of 0
                        if (frameArr[hand].useBit == 1) {
                                frameArr[hand].useBit = 0;
                                hand = (hand+1) % frame; //circular
                        }
                        else {
                                frameArr[hand].pageNum = currPage;
                                frameArr[hand].useBit = 0;
                                hand = (hand+1) % frame;
                                break;
```

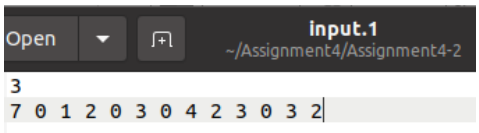
코드 작성을 끝낸 뒤, Makefile 을 만든다.

```
all: page_replacement_simulator

page_replacement_simulator: page_replacement_simulator.c
        gcc -o page_replacement_simulator page_replacement_simulator.c
```

#test case 1

3 개의 frame 과 아래 reference string 을 사용했을 때 각 알고리즘 별로 page fault 수와 page fault rate 를 출력한 결과이다.




```
os2022202065@ubuntu:~/Assignment4/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: 8
Page Fault Rate: 61.54%
```

각 알고리즘 별로 frame 의 단계 별 과정을 출력한 결과이다.

Optimal

```
Page Fault 7: 7 -1 -1
Page Fault 0: 7 0 -1
Page Fault 1: 7 0 1
Page Fault 2: 2 0 1
Page hit   0: 2 0 1
Page Fault 3: 2 0 3
Page hit   0: 2 0 3
Page Fault 4: 2 4 3
Page hit   2: 2 4 3
Page hit   3: 2 4 3
Page Fault 0: 2 0 3
Page hit   3: 2 0 3
Page hit   2: 2 0 3
```

FIFO

```
Page Fault 7: 7 -1 -1
Page Fault 0: 7 0 -1
Page Fault 1: 7 0 1
Page Fault 2: 2 0 1
Page hit   0: 2 0 1
Page Fault 3: 2 3 1
Page Fault 0: 2 3 0
Page Fault 4: 4 3 0
Page Fault 2: 4 2 0
Page Fault 3: 4 2 3
Page Fault 0: 0 2 3
Page hit   3: 0 2 3
Page hit   2: 0 2 3
```

LRU

```
Page Fault 7: 7 -1 -1
Page Fault 0: 7 0 -1
Page Fault 1: 7 0 1
Page Fault 2: 0 1 2
Page hit   0: 1 2 0
Page Fault 3: 2 0 3
Page hit   0: 2 3 0
Page Fault 4: 3 0 4
Page Fault 2: 0 4 2
Page Fault 3: 4 2 3
Page Fault 0: 2 3 0
Page hit   3: 2 0 3
Page hit   2: 0 3 2
```

Clock

```
Page Fault 7: 7(0) -1(0) -1(0)
Page Fault 0: 7(0) 0(0) -1(0)
Page Fault 1: 7(0) 0(0) 1(0)
Page Fault 2: 2(0) 0(0) 1(0)
Page hit   0: 2(0) 0(1) 1(0)
Page Fault 3: 2(0) 0(0) 3(0)
Page hit   0: 2(0) 0(1) 3(0)
Page Fault 4: 4(0) 0(1) 3(0)
Page Fault 2: 4(0) 0(0) 2(0)
Page Fault 3: 3(0) 0(0) 2(0)
Page hit   0: 3(0) 0(1) 2(0)
Page hit   3: 3(1) 0(1) 2(0)
Page hit   2: 3(1) 0(1) 2(1)
```

#test case 2

3 개의 frame 과 아래 reference string 을 사용했을 때 각 알고리즘 별로 page fault 수와 page fault rate 를 출력한 결과이다.

```
3
1 2 3 4 2 1 2 1 3 1
os2022202065@ubuntu: ~/Assignment4/Assignment4-2
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%
```

각 알고리즘 별로 frame 의 단계 별 과정을 출력한 결과이다.

Optimal

```
Page Fault 1: 1 -1 -1
Page Fault 2: 1 2 -1
Page Fault 3: 1 2 3
Page Fault 4: 1 2 4
Page hit   2: 1 2 4
Page hit   1: 1 2 4
Page hit   2: 1 2 4
Page hit   1: 1 2 4
Page Fault 3: 1 3 4
Page hit   1: 1 3 4
```

FIFO

```
Page Fault 1: 1 -1 -1
Page Fault 2: 1 2 -1
Page Fault 3: 1 2 3
Page Fault 4: 4 2 3
Page hit   2: 4 2 3
Page Fault 1: 4 1 3
Page Fault 2: 4 1 2
Page hit   1: 4 1 2
Page Fault 3: 3 1 2
Page hit   1: 3 1 2
```

LRU

```
Page Fault 1: 1 -1 -1
Page Fault 2: 1 2 -1
Page Fault 3: 1 2 3
Page Fault 4: 2 3 4
Page hit   2: 3 4 2
Page Fault 1: 4 2 1
Page hit   2: 4 1 2
Page hit   1: 4 2 1
Page Fault 3: 2 1 3
Page hit   1: 2 3 1
```

Clock

```
Page Fault 1: 1(0) -1(0) -1(0)
Page Fault 2: 1(0) 2(0) -1(0)
Page Fault 3: 1(0) 2(0) 3(0)
Page Fault 4: 4(0) 2(0) 3(0)
Page hit   2: 4(0) 2(1) 3(0)
Page Fault 1: 4(0) 2(0) 1(0)
Page hit   2: 4(0) 2(1) 1(0)
Page hit   1: 4(0) 2(1) 1(1)
Page Fault 3: 3(0) 2(1) 1(1)
Page hit   1: 3(0) 2(1) 1(1)
```

#test case 3

4 개의 frame 과 아래 reference string 을 사용했을 때 각 알고리즘 별로 page fault 수와 page fault rate 를 출력한 결과이다.

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

```
os2022202065@ubuntu: ~/Assignment4/Assignment4-2
os2022202065@ubuntu:~/Assignment4/Assignment4-2$ ./page_replacement_simulator input.1
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%
```

각 알고리즘 별로 frame 의 단계 별 과정을 출력한 결과이다.

Optimal

```
Page Fault 1: 1 -1 -1 -1
Page Fault 3: 1 3 -1 -1
Page Fault 0: 1 3 0 -1
Page hit   3: 1 3 0 -1
Page Fault 5: 1 3 0 5
Page Fault 6: 1 3 6 5
Page hit   3: 1 3 6 5
Page Fault 2: 1 2 6 5
Page hit   5: 1 2 6 5
Page hit   2: 1 2 6 5
Page Fault 4: 1 4 6 5
Page hit   1: 1 4 6 5
Page Fault 0: 0 4 6 5
Page hit   5: 0 4 6 5
```

FIFO

```
Page Fault 1: 1 -1 -1 -1
Page Fault 3: 1 3 -1 -1
Page Fault 0: 1 3 0 -1
Page hit   3: 1 3 0 -1
Page Fault 5: 1 3 0 5
Page Fault 6: 6 3 0 5
Page hit   3: 6 3 0 5
Page Fault 2: 6 2 0 5
Page hit   5: 6 2 0 5
Page hit   2: 6 2 0 5
Page Fault 4: 6 2 4 5
Page Fault 1: 6 2 4 1
Page Fault 0: 0 2 4 1
Page Fault 5: 0 5 4 1
```

LRU

```
Page Fault 1: 1 -1 -1 -1
Page Fault 3: 1 3 -1 -1
Page Fault 0: 1 3 0 -1
Page hit   3: 1 0 3 -1
Page Fault 5: 1 0 3 5
Page Fault 6: 0 3 5 6
Page hit   3: 0 5 6 3
Page Fault 2: 5 6 3 2
Page hit   5: 6 3 2 5
Page hit   2: 6 3 5 2
Page Fault 4: 3 5 2 4
Page Fault 1: 5 2 4 1
Page Fault 0: 2 4 1 0
Page Fault 5: 4 1 0 5
```

Clock

```
Page Fault 1: 1(0) -1(0) -1(0) -1(0)
Page Fault 3: 1(0) 3(0) -1(0) -1(0)
Page Fault 0: 1(0) 3(0) 0(0) -1(0)
Page hit   3: 1(0) 3(1) 0(0) -1(0)
Page Fault 5: 1(0) 3(1) 0(0) 5(0)
Page Fault 6: 6(0) 3(1) 0(0) 5(0)
Page hit   3: 6(0) 3(1) 0(0) 5(0)
Page Fault 2: 6(0) 3(0) 2(0) 5(0)
Page hit   5: 6(0) 3(0) 2(0) 5(1)
Page hit   2: 6(0) 3(0) 2(1) 5(1)
Page Fault 4: 4(0) 3(0) 2(1) 5(0)
Page Fault 1: 4(0) 1(0) 2(1) 5(0)
Page Fault 0: 4(0) 1(0) 2(0) 0(0)
Page Fault 5: 5(0) 1(0) 2(0) 0(0)
```

-알고리즘 별 성능결과 비교 분석

각 test case 마다 출력 결과를 보면, 항상 Optimal 이 제일 page fault 가 낮아 좋은 성능을 보인다는 것을 알 수 있다. Optimal 은 실제로는 구현이 불가능하지만 이론적으로는 제일 좋은 알고리즘이기 때문이다.

반대로 FIFO 는 언제나 제일 높은 Page fault 가 나온다. 구현이 간단하지만 그만큼 성능이 좋지 않고, frame 이 증가해도 page fault 가 증가하는 Belady's Anomaly 이상현상이 발생할 수 있다.

따라서 test case 에서도 중간 정도의 page fault 를 가지는 LRU 와 Clock 이 실제로 쓰일 수 있는데, 대체로 Optimal 보단 떨어지지만 FIFO 보다는 좋은 성능이 나오는 것을 볼 수 있다. LRU 는 Locality 를 다 반영하여 성능이 좋지만 실제 구현이 복잡하기 때문에 그와 비슷한 Clock 을 널리 사용하는데, 실제로 출력 결과에서도 LRU 와 Clock 이 비슷한 page fault 를 가지는 것을 볼 수 있다.

고찰

4-1

가상 메모리 영역을 for 문을 통해 순회하는 과정에서, 단순히 시작 주소인 mmap 부터 vm_next 로 이동하는 코드만 작성하고 실행했다가 killed 되는 상황이 발생하였다. 이는 파일의 경로를 얻는 과정에서 만약 vm_file 이 NULL 일 경우에 NULL 을 참조하게 되기 때문에 오류가 발생하는 것이었다. vm_file 은 vm_area_struct 가 특정 파일과 연관되어 있는지, 실제로 그 파일이 존재하는지 여부를 나타내므로 먼저 이 조건을 검사해야 한다는 것을 알 수 있었다. 또한 파일의 경로를 얻었더라도 거기에 에러가 없는지 검사해야 정상적으로 출력될 수 있었다.

4-2

각 알고리즘을 구현하는 과정에서, page fault 가 예상 값과 다르게 나오기에 frame 배열의 단계별 진행상황을 출력하는 디버깅 코드를 추가하였다. (이후 주석 처리된 부분) FIFO 를 구현하는 건 원형 큐 방식을 이용하여 구현하였는데, Optimal 은 가장 먼 거리를 측정해야 하니 변수를 정해두고 Max 값으로 업데이트하는 방식을 취하였다. 비슷한 원리인 LRU 는 그 반대로 측정하도록 구현하려 했다가, 배열 값들이 다 다르게 나오는 등 더 어려워지는 것 같아 stack 원리를 이용하도록 수정하였다. 처음엔 stack 의 기능을 다 구현하여서 push 하고 remove 하는 걸 생각하였으나, 이렇게 하면 원래 생각했던 거리 측정이랑 복잡도가 크게 달라지지 않을 것 같아 더 간단한 방법을 생각해냈다. 어차피 중간에서 삭제되고 비워져 있을 일이 없으니 시간은 조금 더 걸리더라도 page 를 교체할 때마다 frame 의 page 들을 한 칸씩 밀어버려서 자동 삭제되는 형식으로 구현하였다. Clock 은 FIFO 처럼 원형 큐로 구현하였는데, 처음에는 원리를 잘 이해하지 못하여 page hit 을 검사하는 과정에서 다 clock hand 를 움직였다. 그렇게 하니 원하던 결과가 안 나왔고, 이후 다시 공부해보니 hand 는 page fault 시에만 들어갈 자리를 찾기 위해 움직인다는 것을 깨달았다. 해당 방법으로 수정하니 정상적으로 출력될 수 있었다.

Reference

강의자료 참고