운영체제실습

assignment 4

담당교수 : 김태석 교수님

학 번 : 2021202058

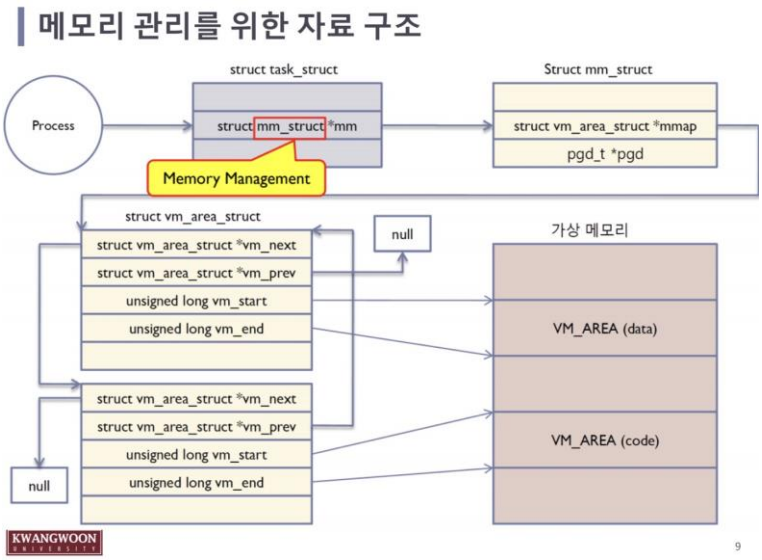
성 명 : 송채영

1. Introduction

이번 과제에서는 프로세스의 가상메모리로부터 pid 를 바탕으로 프로세스의 이름, 정보가 위치하는 가상 메모리 주소, 프로세스의 데이터 주소, 코드 주소, 힙 주소, 정보의 원본 파일의 전체 경로를 출력해본다. 또한 Dynamic Recompilation, 동적 재컴파일에 대해 이해하고, 코드를 최적화해보며 코드의 작동시간을 비교해본다.

2. Conclusion & Analysis

4-1



우선 메모리 관리를 위한 자료구조를 살펴보기 위해, 12 주차 강의자료를 살펴보았다. Task 는 메모리관리를 사용하는 주체로, task 가 사용하는 memory 에 대한 정보를 유지하기 위한 자료구조가 필요하며 task_struct 구조체 안에 struct mm_struct *mm 으로 선언되어 있다. 이때 mm_struct 를 memory management 라고 한다.

```
struct mm_struct                *mm;
```

task_sturct 에서 mm_sturct 가 선언되어 있으며, mm_struct 의 포인터는 mm 인 것을 확인하였다.

```
struct mm_struct {
        struct {
                struct vm_area_struct *mmap;            /* list of VMAs */
                struct rb_root mm_rb;
                u64 vmacache_seqnum;                    /* per-thread vmacache */
#ifdef CONFIG_MMU
```

mm_struct 내부에는 struct vm_area_struct 가 선언되어 있으며 포인터는 mmap 인 것을 확인하였다. 가상 메모리 영역을 관리하기 위해 struct vm_area_struct 를 사용하는데 이들은 연결 리스트로 이루어져 있으며 mmap 변수는 리스트의 시작을 나타낸다.

```
struct vm_area_struct {
        /* The first cache line has the info for VMA tree walking. */

        unsigned long vm_start;         /* Our start address within vm_mm. */
        unsigned long vm_end;           /* The first byte after our end address
                                           within vm_mm. */

        /* linked list of VM areas per task, sorted by address */
        struct vm_area_struct *vm_next, *vm_prev;
```

vm_area_struct 의 멤버변수로 unsigned long vm_start, unsigned long vm_end, struct vm_area_struct *vm_next 가 존재하는데, 각각 영역의 시작과 끝 주소, 다음 가상 메모리블록과 이전 가상 메모리블록을 의미한다.

```
spinlock_t arg_lock; /* protect the below fields */
unsigned long start_code, end_code, start_data, end_data;
unsigned long start_brk, brk, start_stack;
unsigned long arg_start, arg_end, env_start, env_end;
```

코드 영역의 시작과 끝, 데이터 영역의 시작과 끝, 힙 영역의 시작과 끝을 나타내는 변수이다.

```
struct file * vm_file;          /* File we map to (can be NULL). */
```

추가적으로 원본 파일의 전체 경로를 출력하기 위해 찾은 부분이다.

ftrace 시스템콜(336)을 file_varea 함수로 후킹하여 프로세스의 메모리 매핑 정보를 출력하는 함수를 구현하였으며, 구현방식에 대해 간단하게 설명해보겠다. 우선 주어진 PID 를 활용해 해당하는 task_struct 의 프로세스 정보를 얻었다. 이후 프로세스의 mm_struct 를 얻어 그 안에 있는 vm_area_struct 를 순회하며 정보를 출력해주었다.

결과화면에 대해 설명하겠다.

```
os2021202058@ubuntu:~/Assignment4/4-1$ make
make -C /lib/modules/4.19.67-2021202058/build SUBDIRS=/home/os2021202058/Assignment4/4-1 modules
make[1]: Entering directory '/home/os2021202058/Downloads/linux-4.19.67'
  CC [M]  /home/os2021202058/Assignment4/4-1/file_varea.o
  Building modules, stage 2.
  MODPOST 1 modules
  CC      /home/os2021202058/Assignment4/4-1/file_varea.mod.o
  LD [M]  /home/os2021202058/Assignment4/4-1/file_varea.ko
make[1]: Leaving directory '/home/os2021202058/Downloads/linux-4.19.67'
gcc -o test test.c
os2021202058@ubuntu:~/Assignment4/4-1$ sudo insmod file_varea.ko
[sudo] password for os2021202058:
os2021202058@ubuntu:~/Assignment4/4-1$ ./test
os2021202058@ubuntu:~/Assignment4/4-1$ sudo rmmod file_varea.ko
os2021202058@ubuntu:~/Assignment4/4-1$ dmesg
```

주어진 test 파일을 이용해 테스트해보았으며 실행결과는 아래와 같다.

```
[79898.223884] ######## Loaded files of a process 'test(14850)' in VM ########
[79898.223887] mem[400000~401000] code[400000~40074c] data[600e10~601040] heap[12eb000~12eb000] /home/os2021202058/Assignment4/4-1/test
[79898.223888] mem[600000~601000] code[400000~40074c] data[600e10~601040] heap[12eb000~12eb000] /home/os2021202058/Assignment4/4-1/test
[79898.223889] mem[601000~602000] code[400000~40074c] data[600e10~601040] heap[12eb000~12eb000] /home/os2021202058/Assignment4/4-1/test
[79898.223890] mem[7ff7cb324000~7ff7cb4e4000] code[400000~40074c] data[600e10~601040] heap[12eb000~12eb000] /lib/x86_64-linux-gnu/libc-2.23.so
[79898.223891] mem[7ff7cb4e4000~7ff7cb6e4000] code[400000~40074c] data[600e10~601040] heap[12eb000~12eb000] /lib/x86_64-linux-gnu/libc-2.23.so
[79898.223891] mem[7ff7cb6e4000~7ff7cb6e8000] code[400000~40074c] data[600e10~601040] heap[12eb000~12eb000] /lib/x86_64-linux-gnu/libc-2.23.so
[79898.223892] mem[7ff7cb6e8000~7ff7cb6ea000] code[400000~40074c] data[600e10~601040] heap[12eb000~12eb000] /lib/x86_64-linux-gnu/libc-2.23.so
[79898.223893] mem[7ff7cb6ee000~7ff7cb714000] code[400000~40074c] data[600e10~601040] heap[12eb000~12eb000] /lib/x86_64-linux-gnu/ld-2.23.so
[79898.223894] mem[7ff7cb913000~7ff7cb914000] code[400000~40074c] data[600e10~601040] heap[12eb000~12eb000] /lib/x86_64-linux-gnu/ld-2.23.so
[79898.223895] mem[7ff7cb914000~7ff7cb915000] code[400000~40074c] data[600e10~601040] heap[12eb000~12eb000] /lib/x86_64-linux-gnu/ld-2.23.so
[79898.223895] ####################################################################
os2021202058@ubuntu:~/Assignment4/4-1$
```

프로세스 이름과 pid, 정보가 위치하는 가상 메모리 주소, 데이터주소, 코드 주소, 힙 주소, 원본파일의 전체 경로가 잘 출력되는 것을 확인하였다.

4-2

Terminal

```
EXEC = D_recompile
cc = gcc

default:
	$(CC) -c D_recompile_test.c
	$(CC) -o test2 D_recompile_test.c
	$(CC) -o drecompile D_recompile.c -lrt
	objdump -d D_recompile_test.o > test
	./test2
	./drecompile

dynamic:
	$(CC) -c D_recompile_test.c
	$(CC) -o test2 D_recompile_test.c
	$(CC) -Ddynamic -o drecompile D_recompile.c -lrt
	objdump -d D_recompile_test.o > test
	./test2
	./drecompile

clean:
	sync
	echo 3 | sudo tee /proc/sys/vm/drop_caches
	$(RM) D_recompile_test D_recompile_test.o
```

4-2 의 makefile 이다. 제안서 내 makefile 형식 예시처럼 default 와 dynamic 으로 나누어 recompile 하지 않는 부분과 recompile 하는 부분으로 나누었으며, test2 와 drecompile 실행 코드 그리고 Objdump 명령어도 각각에 넣어주었다. 또한 매 실험 전 캐시 및 버퍼를 비워주는 명령어도 추가해주었다.

Objdump 뜨는 과정은 $(CC) -c D_recompile_test.c, objdump -d D_recompile_test.o > test 이 두부분에 해당한다. Objdump 는 바이너리에 있는 기계어를 어셈블리어로 변환해주는 역할을 한다.

```
os2021202058@ubuntu:~/Assignment4/4-2$ ls
drecompile  D_recompile.c  D_recompile_test.c  D_recompile_test.o  Makefile  test  test2
os2021202058@ubuntu:~/Assignment4/4-2$ cat test

D_recompile_test.o:     file format elf64-x86-64


Disassembly of section .text:

0000000000000000 <Operation>:
   0:   55                      push   %rbp
   1:   48 89 e5                mov    %rsp,%rbp
   4:   89 7d fc                mov    %edi,-0x4(%rbp)
   7:   8b 55 fc                mov    -0x4(%rbp),%edx
   a:   89 d0                   mov    %edx,%eax
   c:   b2 02                   mov    $0x2,%dl
   e:   83 c0 01                add    $0x1,%eax
  11:   83 c0 01                add    $0x1,%eax
  14:   83 c0 01                add    $0x1,%eax
  17:   83 c0 01                add    $0x1,%eax
  1a:   83 c0 02                add    $0x2,%eax
  1d:   83 c0 03                add    $0x3,%eax
  20:   83 c0 01                add    $0x1,%eax
  23:   83 c0 02                add    $0x2,%eax
  26:   83 c0 01                add    $0x1,%eax
  29:   83 c0 01                add    $0x1,%eax
  2c:   6b c0 02                imul   $0x2,%eax,%eax
  2f:   6b c0 02                imul   $0x2,%eax,%eax
  32:   6b c0 02                imul   $0x2,%eax,%eax
  35:   83 c0 01                add    $0x1,%eax
  38:   83 c0 01                add    $0x1,%eax
  3b:   83 c0 03                add    $0x3,%eax
  3e:   83 c0 01                add    $0x1,%eax
  41:   83 c0 01                add    $0x1,%eax
  44:   83 c0 01                add    $0x1,%eax
  47:   83 c0 03                add    $0x3,%eax
  4a:   83 c0 01                add    $0x1,%eax
  4d:   83 c0 01                add    $0x1,%eax
  50:   83 c0 02                add    $0x2,%eax
  53:   83 c0 01                add    $0x1,%eax
  56:   83 c0 01                add    $0x1,%eax
  59:   83 c0 01                add    $0x1,%eax
  5c:   83 c0 01                add    $0x1,%eax
  5f:   83 c0 01                add    $0x1,%eax
  62:   f6 f2                   div    %dl
  64:   f6 f2                   div    %dl
  66:   83 e8 01                sub    $0x1,%eax
  69:   83 e8 01                sub    $0x1,%eax
  6c:   83 e8 03                sub    $0x3,%eax
  6f:   83 e8 01                sub    $0x1,%eax
```

```
 c9e:   48 01 c2                add    %rax,%rdx
 ca1:   48 8b 45 f0             mov    -0x10(%rbp),%rax
 ca5:   0f b6 00                movzbl (%rax),%eax
 ca8:   88 02                   mov    %al,(%rdx)
 caa:   48 8b 45 f0             mov    -0x10(%rbp),%rax
 cae:   48 8d 50 01             lea    0x1(%rax),%rdx
 cb2:   48 89 55 f0             mov    %rdx,-0x10(%rbp)
 cb6:   0f b6 00                movzbl (%rax),%eax
 cb9:   3c c3                   cmp    $0xc3,%al
 cbb:   75 d1                   jne    c8e <main+0x46>
 cbd:   bf 01 00 00 00          mov    $0x1,%edi
 cc2:   e8 00 00 00 00          callq  cc7 <main+0x7f>
 cc7:   89 c6                   mov    %eax,%esi
 cc9:   bf 00 00 00 00          mov    $0x0,%edi
 cce:   b8 00 00 00 00          mov    $0x0,%eax
 cd3:   e8 00 00 00 00          callq  cd8 <main+0x90>
 cd8:   48 8b 45 f8             mov    -0x8(%rbp),%rax
 cdc:   48 89 c7                mov    %rax,%rdi
 cdf:   e8 00 00 00 00          callq  ce4 <main+0x9c>
 ce4:   bf 00 00 00 00          mov    $0x0,%edi
 ce9:   e8 00 00 00 00          callq  cee <main+0xa6>
 cee:   b8 00 00 00 00          mov    $0x0,%eax
 cf3:   c9                      leaveq
 cf4:   c3                      retq
os2021202058@ubuntu:~/Assignment4/4-2$
```

cat 명령어를 통해 test 를 출력해보았다. Dump 파일로부터 코드를 어떻게 구현해보아야 할지 고민을 해보았는데, 먼저 각 operation 의 규칙을 살펴보았다. add 명령어의 경우 첫 번째 값이 0x83 두 번째 값이 0xc0, 세 번째 값은 상수에 해당한다. sub 명령어의 경우 첫 번째 값이 0x83 두 번째 값이 0xe8, 세 번째 값은 상수에 해당한다. imul 명령어의 경우 첫 번째 값이 0x6b 두 번째 값이 0xc0, 세 번째 값은 상수에 해당한다. div 명령어의 경우 첫 번째 값이 0xf6 두 번째 값이 0xf2 에 해당한다. 또한 type 이 unit8_t 을

사용하며 이는 부호가 없는 8 비트를 의미한다. 이를 통해 같은 부분이 반복해서 출력되는 것을 확인했고, 배열로 접근하며 operation 을 구분한 후 나오는 숫자들을 모두 합쳐서 연산하는 방식으로 구현하면 최적화가 될 것 같다는 생각이 들었다.

구현한 코드에 대해 간단하게 설명해보겠다.

우선 sharedmem_init 부분은 shmget 함수를 통해 커널에 shared memory segment 를 위한 공간 요청을 수행하였다. 또한 shmat 함수를 활용해 현재 프로세스가 생성된 shared memory 를 사용할 수 있도록 id 값을 이용해 attach 를 수행하였다. 이렇게 함으로써 D_recompile_test.c 코드에서 shared memory 에 존재하는 operation 부분을 얻을 수 있다.

decompile_init 부분은 write 권한이 있는 영역 즉 compiled_code 를 mmap 을 통해 할당하고 operation 함수를 복사하는 작업을 수행하였다.

decompile_exit 부분은 msync 를 사용하여 메모리에 매핑된 파일이나 장치를 unmap 시켰고, munmap 함수를 사용하여 메모리 매핑 후 변경된 사항을 파일에 반영하였다.

sharedmem_exit 부분은 shmdt 함수를 사용하여 프로세스와 shared memory 를 분리하였고 shmctl 함수를 사용하여 shared memory 의 권한을 삭제 하였다.

최적화를 하는 drecompile 함수에서는 최적화를 진행하지 않을 때 memcpy 를 사용해 값을 복사해줌으로써 두 포인터가 동일한 메모리를 가리키도록 하였으며, 최적화를 진행할 때는 ifdef 와 endif 를 사용하여 구현해주었다. 우선 변수 i 는 func 배열의 인덱스 값에 해당하며 변수 j 는 compiled_code 배열의 인덱스 값에 해당한다.

func 배열을 순회하며 종료조건인 0xc3 을 만날 때 까지 반복하였고, add 명령어와 sub 명령어의 첫 부분이 0x83 이므로 0xc0 인 경우와 0xe8 인 경우로 나누어 명령어를 구분해주었다. 이후 연속된 ADD or SUB 명령어를 찾아 그 합을 구해준 후 compiled 배열에 복사하였다. imul 명령어도 위와 동일하게 진행하였고 마지막으로 div 명령어의 경우에는 mov di 부분이 추가적으로 필요했다. Mov 명령어는 di 값에 2 를 로드하기 때문에 해당 부분을 이용하여 배열에 복사한 후 다시 원래대로 돌려놓는 과정을 추가적으로 해주었다.

마지막으로 execute 권한을 부여하기 위해 mprotect 함수를 사용하였다.

Make clean 을 통해 캐시 및 버퍼를 지우고 최적화 하기 전 결과를 실행하고, 다시 캐시 및 버퍼를 지우고 최적화 후 결과를 출력한 사진이다.

```
os2021202058@ubuntu:~/Assignment4/4-2$ make clean
sync
echo 3 | sudo tee /proc/sys/vm/drop_caches
3
rm -f D_recompile_test D_recompile_test.o
rm -f D_recompile D_recompile.o
rm -f test2 test
rm -f drecompile
os2021202058@ubuntu:~/Assignment4/4-2$ make
cc -c D_recompile_test.c
cc -o test2 D_recompile_test.c
cc -o drecompile D_recompile.c -lrt
mobjdump -d D_recompile_test.o > test
./test2
result: 15
Data was filled to shared memory.
./drecompile
result: 15
total execution time: 0.000000890 sec
os2021202058@ubuntu:~/Assignment4/4-2$ make clean
sync
echo 3 | sudo tee /proc/sys/vm/drop_caches
3
rm -f D_recompile_test D_recompile_test.o
rm -f D_recompile D_recompile.o
rm -f test2 test
rm -f drecompile
os2021202058@ubuntu:~/Assignment4/4-2$ make dynamic
cc -c D_recompile_test.c
cc -o test2 D_recompile_test.c
cc -Ddynamic -o drecompile D_recompile.c -lrt
objdump -d D_recompile_test.o > test
./test2
result: 15
Data was filled to shared memory.
./drecompile
result: 15
total execution time: 0.000000306 sec
```

해당 set 를 50 번 반복한 결과이다.

번호	최적화 전	최적화 후
1	0.000000767	0.000000553
2	0.000000927	0.000000351
3	0.000000911	0.000000306
4	0.000000879	0.000000459
5	0.000000729	0.000000452
6	0.000000818	0.000000303
7	0.000000858	0.000000305
8	0.000000774	0.000000259
9	0.000000866	0.000000262
10	0.000000963	0.000000354
11	0.000000875	0.000000461
12	0.000000732	0.000000355

13	0.000000941	0.00000036
14	0.000000832	0.000000556
15	0.000000825	0.000000457
16	0.00000077	0.000000511
17	0.000000775	0.000000312
18	0.000000811	0.00000029
19	0.000001013	0.000000302
20	0.000000825	0.000000455
21	0.000000779	0.000000356
22	0.000000827	0.000000352
23	0.000000816	0.000000502
24	0.000000855	0.000000354
25	0.000000875	0.000000501
26	0.000000715	0.000000357
27	0.000000867	0.000000366
28	0.000000669	0.000000353
29	0.000000872	0.000000352
30	0.000000723	0.000000315
31	0.000000824	0.000000552
32	0.000000819	0.000000354
33	0.000000764	0.000000358
34	0.000000764	0.000000362
35	0.000001088	0.000000353
36	0.000000865	0.000000453
37	0.000000966	0.00000035
38	0.00000072	0.000000351
39	0.000000815	0.000000306
40	0.000000825	0.000000453
41	0.000000801	0.000000353
42	0.00000077	0.000000354
43	0.00000093	0.000000357
44	0.000000819	0.000000308
45	0.000000817	0.000000306
46	0.000001081	0.000000354
47	0.000000766	0.000000306

48	0.00000098	0.000000351
49	0.000000826	0.000000373
50	0.00000089	0.000000306
평균	0.000000840	0.000000375

50 번을 실행한 후 평균을 낸 결과로 약 0.000000465 sec 차이가 난다.

3. 고찰

4-1 과제는 3-3 차 과제를 바탕으로 진행했기 때문에 큰 어려움은 없었다. 강의자료를 바탕으로 cscope 를 사용했는데, 정보의 원본 파일의 전체 경로를 출력하는 부분이 막막했다. 구글링을 통해 dpath 를 활용하여 출력할 수 있다는 것을 확인하여 사용했는데 vm_file 원형은 찾을 수 있었으나, f_path 원형은 찾지 못하여 이 부분 역시 구글링을 통해 해결하였다. dpath 이외에도 dentry_path_raw 를 사용하여 출력할 수 있었으나 dentry 내부를 확인하여 구현해야 했기에 더 간단한 방법을 택하였다. 4-2 과제에서는 objdump 파일을 보고 어떤 식으로 구현하면 좋을지 생각해 내는 부분에서 시간이 많이 걸렸다. 또한 drecompile 부분을 구현하면서 segmentation fault 도 자주 떠서 애를 많이 먹었던 것 같다. seg 오류가 떴던 결론적인 이유는 continue 를 해주지 않았기 때문이다. 또한 최적화 전에 해당하는 부분에도 while 문으로 코드를 끝까지 돌며 복사하는 과정을 넣어주었다. 단순 복사 과정임에도 불구하고 복사했을 때와 안 했을 때 실행시간이 같지 않고 차이가 발생하므로 같은 상황을 만들어 주기 위해 필요하다고 생각했다. 또한 실행 시간 측정에 있어서도 공평한 조건을 만들어주기 위해 func(1) 앞뒤로 실행 시간을 측정하도록 하였다. 마지막으로 drecompile_init 부분에서 fd 를 0 으로 넣어 사용하여 mapping 이 제대로 되지 않는다는 것을 확인하고 MAP_ANONYMOUS 를 사용하여 fd 대신 파일로 사용하지 않음을 뜻하며 -1 을 넣어주었다. assert 함수를 통해 확인해본 결과 mapping 이 잘됐음을 확인할 수 있었다. 그 전에 구현했던 내용은 shared memory 에 mapping 하도록 한 것이기 때문에 mapping 이 제대로 이루어지지 않아 해결해주었다. 또한 처음에는 compiled_code = func 으로 해줌으로써 같은 메모리를 참조하게끔 구현하였는데 이렇게 할 경우, func 의 부분이 망가질 것이라고 생각해 memcpy 를 사용하도록 수정하였다. 또한 그렇게 할 경우 mmap 부분을 주석 처리하고 실행해도 코드가 돌아가므로 mmaping 을 제대로 한 것을 사용하는 것이 아니라 shared memory 를 사용하는 것이기 때문에 고쳐주어 올바르게 동작하도록 해결하였다.

4. Reference

- 운영체제실습 강의자료 참조
- mm_struct / vm_area_struct 구조체 / https://showx123.tistory.com/92
- 파일명, 파일 경로 출력 / https://beausty23.tistory.com/109
- dpath / https://github.com/iamroot11c/kernel/blob/master/mm/memory.c#L4289
- fpath / https://github.com/iamroot11c/kernel/blob/master/mm/memory.c#L4289
- MAP_ANONYMOUS / https://decdream.tistory.com/602