운영체제실습

assignment 4

학　　과: 컴퓨터정보공학부

담당교수: 김태석 교수님

학　　번: 2019202050

성　　명: 이강현

1. Introduction

pid 를 바탕으로 프로세스의 정보를 출력해보는 과제를 통해 프로세스내의 여러 구조체가 가지는 정보에 어떤 것이 있는지 알아보고 프로세스가 가상메모리에 어떻게 저장이 되어있는지를 안다. 다른 프로그램의 코드를 shared memory 를 이용하여 얻고 이를 최적화 해보는 Dynamic Recompilation 을 진행해 본다. 메모리 매핑에 대한 개념과 활용 방법에 대해 배운다.

2. Conclusion & Analysis

<assignment 4-1>

먼저 정보들을 출력하기 전 구조체내에 어떻게 정보들이 저장되어있는지 cscope 를 이용하여 원형을 찾아보았다.

```
struct mm_struct {
        struct {
                struct vm_area_struct *mmap;
                struct rb_root mm_rb;
                u64 vmacache_seqnum;
*/
#ifdef CONFIG MMU
```

먼저 task_struct 안에 존재하는 mm_struct 이다. 내부에 vm_area_struct 포인터인 mmap 이 존재하는 것을 확인할 수 있다. mm_struct 에 추가적으로 code, data, heap 가상메모리 주소의 시작과 끝을 담고 있는 변수들이 있는데 아래와 같다.

```
        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;
```

프로세스가 할당 받은 가상메모리에서 블록단위로 정보들을 출력하기 때문에 vm_area_struct 내부의 블록의 시작과 끝의 주소를 담고있는 변수들을 확인해본다.

```
struct vm_area_struct {
        /* The first cache line has t

        unsigned long vm_start;
        unsigned long vm_end;
s
```

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

vm_start 와 vm_end 로 블록의 시작주소와 끝주소를 알아낼 수 있고 다음 블록으로의 접근을 위해 vm_next 변수를 사용하였다.

다음은 이 블록을 사용하는 원본 파일의 전체 경로를 출력해야하는데 이에 대한 정보는 vm_area_struct 의 vm_file 이라는 변수를 참조하여 얻어낼 수 있었다.

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

```
struct dentry {
        /* RCU lookup touched fields */
        unsigned int d_flags;           /* pr
        seqcount_t d_seq;               /* pe
        struct hlist_bl_node d_hash;    /* lo
        struct dentry *d_parent;        /* pa
        struct qstr d_name;
        struct inode *d_inode;          /* Wh
```

```
struct qstr {
        union {
                struct {
                        HASH_LEN_DECLARE;
                };
                u64 hash_len;
        };
        const unsigned char *name;
};
```

vm_file 내의 dentry, qstr 을 이용하여 파일의 이름을 알아낼 수도 있으나 이번에는 전체 경로가 필요했기에 vm_file 의 file 이라는 struct 를 찾아 f_path 라는 변수를 통해 전체 경로를 얻어냈다.

```
struct file {
        union {
                struct llist_node       fu_llist;
                struct rcu_head         fu_rcuhead;
        } f_u;
        struct path             f_path;
```

f_path 를 d_path 라는 함수를 통해 char* 형식으로 변환하여 출력하였다.

모듈을 완성하고 다음과 같이 적재하고 실행하였다. test 파일은 전역변수를 선언하고 지역변수를 선언하고 동적할당도 하여 데이터를 채운 후 시스템콜 336 을 호출하는 코드를 직접 작성하여 테스트하였다. 결과는 아래와 같다.

```
os2019202050@ubuntu:~/assignment4-1$ make
make -C /lib/modules/4.19.67-2019202050/build SUBDIRS=/home/os2019202050/assignment4-1 modules
make[1]: Entering directory '/home/os2019202050/Downloads/linux-4.19.67'
  CC [M]  /home/os2019202050/assignment4-1/file_varea.o
  Building modules, stage 2.
  MODPOST 1 modules
  CC      /home/os2019202050/assignment4-1/file_varea.mod.o
  LD [M]  /home/os2019202050/assignment4-1/file_varea.ko
make[1]: Leaving directory '/home/os2019202050/Downloads/linux-4.19.67'
cc -o test test.c
os2019202050@ubuntu:~/assignment4-1$ sudo insmod file_varea.ko
[sudo] password for os2019202050:
os2019202050@ubuntu:~/assignment4-1$ lsmod | grep file
file_varea             16384  0
os2019202050@ubuntu:~/assignment4-1$ ./test
os2019202050@ubuntu:~/assignment4-1$ dmesg
```

```
[16355.517895] ######## Loaded files of a process 'test(13102)' in VM ########
[16355.517897] mem[400000~401000] code[400000~4007fc] data[600e10~601054] heap[743000~764000] /home/os2019202050/assignment4-1/test
[16355.517898] mem[600000~601000] code[400000~4007fc] data[600e10~601054] heap[743000~764000] /home/os2019202050/assignment4-1/test
[16355.517899] mem[601000~602000] code[400000~4007fc] data[600e10~601054] heap[743000~764000] /home/os2019202050/assignment4-1/test
[16355.517899] mem[743000~764000] associated file is not exist
[16355.517900] mem[7f2dc9636000~7f2dc97f6000] code[400000~4007fc] data[600e10~601054] heap[743000~764000] /lib/x86_64-linux-gnu/libc-2.23.so
[16355.517901] mem[7f2dc97f6000~7f2dc99f6000] code[400000~4007fc] data[600e10~601054] heap[743000~764000] /lib/x86_64-linux-gnu/libc-2.23.so
[16355.517901] mem[7f2dc99f6000~7f2dc99fa000] code[400000~4007fc] data[600e10~601054] heap[743000~764000] /lib/x86_64-linux-gnu/libc-2.23.so
[16355.517902] mem[7f2dc99fa000~7f2dc99fc000] code[400000~4007fc] data[600e10~601054] heap[743000~764000] /lib/x86_64-linux-gnu/libc-2.23.so
[16355.517902] mem[7f2dc99fc000~7f2dc9a00000] associated file is not exist
[16355.517903] mem[7f2dc9a00000~7f2dc9a26000] code[400000~4007fc] data[600e10~601054] heap[743000~764000] /lib/x86_64-linux-gnu/ld-2.23.so
[16355.517903] mem[7f2dc9c0c000~7f2dc9c0f000] associated file is not exist
[16355.517904] mem[7f2dc9c25000~7f2dc9c26000] code[400000~4007fc] data[600e10~601054] heap[743000~764000] /lib/x86_64-linux-gnu/ld-2.23.so
[16355.517905] mem[7f2dc9c26000~7f2dc9c27000] code[400000~4007fc] data[600e10~601054] heap[743000~764000] /lib/x86_64-linux-gnu/ld-2.23.so
[16355.517905] mem[7f2dc9c27000~7f2dc9c28000] associated file is not exist
[16355.517905] mem[7ffc9dc1e000~7ffc9dc3f000] associated file is not exist
[16355.517906] mem[7ffc9dcc7000~7ffc9dcca000] associated file is not exist
[16355.517906] mem[7ffc9dcca000~7ffc9dccc000] associated file is not exist
[16355.517907] ############################################################
```

이때 file 과 연관없는 블록이 존재하기도 해서 이와 같은 블록은 경로를 출력하지 않고 예외처리를 하였다.

```
os2019202050@ubuntu:~/assignment4-1$ sudo rmmod file_varea
[sudo] password for os2019202050:
os2019202050@ubuntu:~/assignment4-1$ lsmod | grep file
os2019202050@ubuntu:~/assignment4-1$ make clean
make -C /lib/modules/4.19.67-2019202050/build SUBDIRS=/home/os2019202050/assignm
ent4-1 clean
make[1]: Entering directory '/home/os2019202050/Downloads/linux-4.19.67'
  CLEAN   /home/os2019202050/assignment4-1/.tmp_versions
  CLEAN   /home/os2019202050/assignment4-1/Module.symvers
make[1]: Leaving directory '/home/os2019202050/Downloads/linux-4.19.67'
rm -f test
```

모듈 해제도 잘 되는 모습이다.

<assignment 4-2>

```
os2019202050@ubuntu:~/assignment4-2$ make
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
os2019202050@ubuntu:~/assignment4-2$ 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 03                add    $0x3,%eax
  1a:   83 c0 01                add    $0x1,%eax
  1d:   83 c0 01                add    $0x1,%eax
  20:   83 c0 01                add    $0x1,%eax
  23:   83 c0 03                add    $0x3,%eax
  26:   83 c0 01                add    $0x1,%eax
  29:   83 c0 01                add    $0x1,%eax
  2c:   83 c0 02                add    $0x2,%eax
  2f:   83 c0 01                add    $0x1,%eax
  32:   83 c0 01                add    $0x1,%eax
  35:   83 c0 01                add    $0x1,%eax
  38:   83 c0 01                add    $0x1,%eax
  3b:   83 c0 01                add    $0x1,%eax
  3e:   83 e8 01                sub    $0x1,%eax
  41:   83 e8 01                sub    $0x1,%eax
  44:   83 e8 03                sub    $0x3,%eax
  47:   83 e8 01                sub    $0x1,%eax
  4a:   83 e8 01                sub    $0x1,%eax
  4d:   83 e8 01                sub    $0x1,%eax
  50:   83 e8 03                sub    $0x3,%eax
  53:   83 e8 01                sub    $0x1,%eax
```

D_recompile_test.c 와 D_recompile.c 를 컴파일하고 D_recompile_test.o 파일을 objdump 를 이용하고 결과를 test 에 redirection 하여 test 를 확인해 보았다. c 파일을 컴파일하여 어셈블리 코드로 변환이 되어있었는데 Operation 함수의 어셈블리 코드를 확인할 수 있었다.

```
83 c0 01                add    $0x1,%eax
83 c0 01                add    $0x1,%eax
83 c0 01                add    $0x1,%eax
83 c0 01                add    $0x1,%eax
83 c0 01                add    $0x1,%eax
83 c0 01                add    $0x1,%eax
```

add 연산자는 공통적으로 0x83, 0xc0 를 가지고 그 다음은 상수였다. 이때 이 상수는 eax 레지스터값과 더해져 다시 eax 레지스터에 저장된다.

```
83 e8 01                sub    $0x1,%eax
83 e8 01                sub    $0x1,%eax
83 e8 03                sub    $0x3,%eax
83 e8 01                sub    $0x1,%eax
83 e8 01                sub    $0x1,%eax
```

sub 연산자는 0x83, 0xe8 를 가졌는데 add 와 0x83 은 동일하나 0xe8 이

달라 구분할 수 있었다. 그 뒤는 상수인데 eax 레지스터값에서 상수를 빼고 다시 eax 레지스터에 저장한다.

```
6b:   6b c0 02             imul   $0x2,%eax,%eax
6e:   6b c0 02             imul   $0x2,%eax,%eax
71:   6b c0 02             imul   $0x2,%eax,%eax
74:   6b c0 02             imul   $0x2,%eax,%eax
77:   6b c0 02             imul   $0x2,%eax,%eax
7a:   6b c0 02             imul   $0x2,%eax,%eax
```

imul 은 0x6b, 0xc0 이고 뒤는 eax 값과 곱할 상수이다.

```
f6 f2                    div    %dl
f6 f2                    div    %dl
f6 f2                    div    %dl
f6 f2                    div    %dl
f6 f2                    div    %dl
```

div 는 0xf6 0xf2 로 eax 레지스터를 dl 로 나누어 eax 에 저장한다.

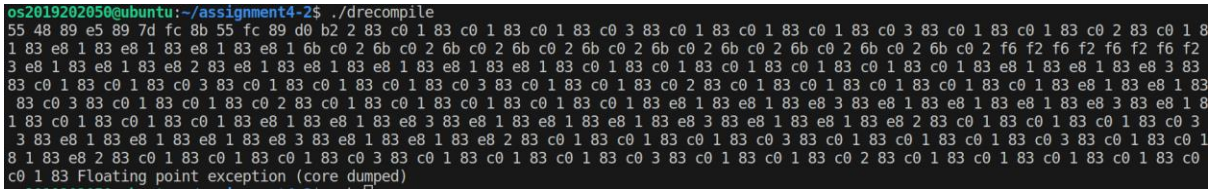
따라서 add,sub,imul,div 일 때 공통되는 값을 가지기 때문에 조건문으로 나누고 뒤의 상수를 하나로 모아 최적화하면 되겠다라는 생각을 하였다. add 와 sub 은 상수값들을 모두 하나의 변수로 합하여 명령줄을 수정하고 imul 과 div 는 상수들을 모두 곱해서 연산을 진행하면 되겠다는 아이디어를 얻었다.

objdump 를 확인하고 중복되는 연산자를 합치는 최적화를 해야했는데 D_recompile_test.c 를 이용하여 Operation 함수를 shared memory 에 저장하고 같은 key 값으로 D_recompile.c 를 이용하여 접근한 후 이를 복사하고 최적화해야 한다는 것은 알고 있었지만 0x6b,0xf6...이런 값들에 어떻게 접근하는지가 이해가 되지 않았다 이에 대한 아이디어는 uint8_t 라는 함수 포인터의 자료형을 보고 알게 되었는데 shared memory 에서 uint8_t 라는 변수를 가리키고 uint8_t 는 결과적으로 unsigned 이고 8bit 라는 뜻이기 때문에 16 진수로 표현된 값 두자리를 담고 있다. 따라서 배열처럼 인덱스를 증가하며 접근하면 값들을 확인할 수 있었다.

반복문과 조건문을 사용해서 add,sub,imul,div 의 명령어에 해당하는 16 진수값들이 나온다면 그 이후에 추가적으로 같은 명령어가 나오는지를 반복문으로 검사하여 나오지 않을때까지 상수값을 합하여 변수에 저장하고 반복문을 탈출하면 새로운 명령줄을 compiled_code 에 넣어주는 것으로 구현하였다.

따라서 전체적인 흐름은 D_recompiled.c 에서는 D_recompiled_test.c 에서 사용한 공유메모리와 동일한 키인 1234 로 공유메모리를 받아오고 프로세스에 attach 한 후 이를 참조하여 함수포인터를 값을 미리 mmap 을 통해 메모리 매핑한 compiled_code 에 넣은 후 최적화한 후 mprotect 로 compiled_code 에 실행권한을 부여해 실행결과를 확인한 후 공유메모리를 detach 하고 해제하는 것으로 구성되어 있다.

결과적으로 코드를 최적화하고 3 이라는 값을 넣고 실행시켜 보았는데 다음과 같은 오류가 발생하였다.



코드를 실행하던 도중 floating point exception 이 발생하였다. 0 이라는 값이 없기 때문에 너무 많은 나누기와 곱셈연산을 진행하다보니 3 이라는 초기값에서 너무 값이 커져버려 표현 범위를 넘었던 것 같다. 따라서 함수에 1 을 넣어서 실행하였다.

전체 결과이다. Makefile 에 c 파일들의 컴파일과 objdump 를 사용하는 것을 모두 담아 make 한 후 ./test2 를 이용하여 shared memory 에 operation 함수의 코드를 채운다. 그 후 ./drecompile 하여 확인하면 최적화되지 않은 것을 확인할 수 있다.

최적화되지 않은 함수를 이용하여 초기값 1 을 넣고 연산을 진행하였을 때 result 는 15 가 나왔고 실행시간은 0.000008 초 가 나온 것을 확인할 수 있다. 그 후 make clean 을 이용해 캐시 및 버퍼를 지우고 파일들을 모두 지운 후 이번엔 make dynamic 을 이용해 최적화 하게끔 하였다. 그 후 ./test2 로 shared memory 를 채우고 ./drecompile 하면 코드의 수가 눈에 띄게 줄어든 것을 확인할 수 있다. 또한 result 도 동일하게 나오는 것을 확인할 수 있고 실행시간도 0.000002 초로 줄었다. 한번의 실행으로는 최적화의 성능을 확인하기 어려우니

50 번의 실행을 진행하여 평균을 내었다. 또한 많은 실행을 위해서 make 파일에 컴파일과 ./test2 ./drecompile 의 실행을 모두 넣어 make 만 입력하면 컴파일과 실행결과가 나오게 끔 하였다. 따라서 make, make clean 으로 최적화 전 set 하나를 구성하였고 make dynamic, make clean 으로 최적화 후 set 하나를 구성하였다.

	A	B	C	D
1	Set	최적화 전	최적화 후	
2	1	0.000009	0.000002	
3	2	0.000009	0.000004	
4	3	0.000003	0.000002	
5	4	0.000016	0.000002	
6	5	0.000002	0.000002	
7	6	0.000002	0.000002	
8	7	0.000009	0.000002	
9	8	0.000014	0.000002	
10	9	0.000002	0.000002	
11	10	0.000008	0.000002	
12	11	0.00001	0.000002	
13	12	0.000002	0.000002	
14	13	0.000002	0.000002	
15	14	0.000002	0.000002	
16	15	0.000002	0.000002	
17	16	0.000009	0.000002	
18	17	0.000002	0.000002	
19	18	0.000009	0.000002	
20	19	0.000009	0.000002	
21	20	0.000002	0.000002	
22	21	0.000002	0.000002	
23	22	0.000009	0.000002	
24	23	0.000002	0.000002	
25	24	0.000003	0.000002	
26	25	0.000002	0.000002	
27	26	0.000002	0.000002	
28	27	0.000009	0.000003	
29	28	0.000002	0.000002	
30	29	0.000002	0.000002	
31	30	0.000003	0.000002	
32	31	0.000002	0.000002	
33	32	0.000002	0.000016	
34	33	0.000009	0.000002	
35	34	0.000009	0.000002	
36	35	0.000015	0.000002	
37	36	0.000002	0.000002	
38	37	0.000002	0.000002	
39	38	0.00001	0.000002	
40	39	0.000009	0.000002	
41	40	0.000002	0.000002	
42	41	0.000009	0.000002	
43	42	0.000002	0.000003	
44	43	0.000002	0.000002	
45	44	0.000002	0.000002	
46	45	0.000002	0.000002	
47	46	0.000002	0.000002	
48	47	0.000008	0.000002	
49	48	0.000002	0.000002	
50	49	0.000002	0.000002	
51	50	0.000013	0.000003	
52	평균	0.00000530	0.00000238	

평균적으로 약 0.00000292 초 정도 빨라진 것을 확인하였다.

3. 고찰

프로세스의 내부에 많은 정보들이 존재한다는 것을 알게 되었고 가상메모리가 어떻게 나누어져 있는지 또 그 주소는 어떻게 존재하는지를 알 수 있었다. 또한 컴퓨터가 다른데 각 페이지들의 주소가 결과 예시와 동일하게 나오는 것을 보고 가상주소라는 것을 확인할 수 있었다. 출력을 해보면서 한가지 알아낸 것은 모든 블록들이 실제 파일과 연관되어 있지는 않는다는 점이었다. vm_file 의 f_path 를 이용하여 확인할 때 null 인 값이 존재하는 경우가 있었다. 두번째로 dynamic recompile 에서는 어떻게 진행해야하는지는 이해했지만 어떻게 코드를 최적화해야 하는지에 대한 방법론을 찾는게 어려웠다. objdump 를 잘 확인하고 함수 포인터의 자료형이 핵심 힌트였던 것 같다. 또 많은 연산을 진행하다 보니 너무 연산의 중간에서 값이 너무 커져버려 오류가 발생했는데 이를 코드를 수정해서 범위가 넘어가지 않게끔 수정하였기에 결과를 얻을 수 있었다. 최적화에는 각 명령어 모두 비슷한 방법을 사용하였으나 div 의 최적화를 위해서 dl 레지스터의 값이 고정적이므로 div 명령어 수만큼 곱하여 저장해두고 이를 mov 명령어를 추가해서 dl 레지스터의 값을 바꾼 후 div 를 한번만 실행하게끔 한다. 그 이후에 dl 를 또 mov 명령어를 이용해 초기값으로 초기화 해주는데 div 의 최적화를 위해서 mov 명령어를 두개 넣었기 때문에 div 가 최소 3 개 이상은 반복되는 코드에 적용했을 때는 최적화가 성능을 낼 수 있으나 1 개나 2 개일때는 오히려 더 많은 시간을 소요할 것으로 예상된다. 이처럼 상황에 따라 최적화 방법이 달라질 수 있고 그 상황에 맞게 최적화 알고리즘을 구현하고 선택하는 것이 좋을 것 같다.

4. Reference

운영체제실습 강의자료 참조