2022학년도 2학기

운영체제

-REPORT-

수업명	운영체제
과제 이름	Assignment4
담당 교수님	최상호 교수님
학번	2018204058
이름	김민교

1. Introduction :

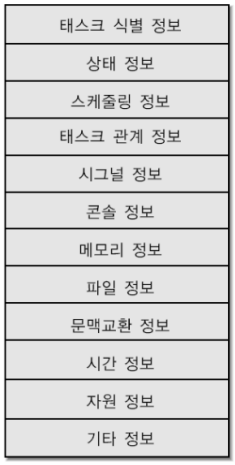
IPC란 Inter-Process Communication의 약자다. 다중 thread나 process간 데이터를 주고 받기 위한 방법을 의미한다. 우리는 IPC에서 Shared memory에 관해 응용해보고 프로세스 간 코드를 공유해본다. 공유 메모리를 활용함과 mmap()를 활용함으로써 Dynamic Compile을 진행한다. 또 task_struct를 분석함으로써 운영체제 이론 수업에서 배웠던 가상메모리를 실제 코드로 공부해본다. 프로세스 당 가상메모리가 정말 할당되는 것을 알고, page 개념을 코드에서 접근한다. 그리고 프로세스의 가상메모리가 각각 어떻게 할당되는지 확인한다.

2. Conclusion & Analysis :

A. assignment4-1

- struct task_struct 분석

task_struct는 프로세스당 하나씩 가지고 있다. Process Descriptor는 프로세스의 모든 정보를 담고 있는 자료구조이다. 운영체제 이론수업에서는 이것을 PCB(Process Control Block)이라 불렀다. 리눅스 에서는 task_struct라는 자료구조를 이용해서 PCB를 표현한다. 커널은 프로세스가 무엇을 하고 있는지 명확히 알아야 한다.



 # task_struct 구조체

먼저 프로세스가 생성되면 task_struct 구조체를 통해 프로세스의 모든 정보를 저장하고 관리한다.

태스크는 자신의 고유한 가상 메모리를 갖는다. 따라서 커널은 태스크의 가상 메모리가 어디에 존재하는지 관리를 해야한다. 즉, 어디에 text 영역이 있고 어디에 data 영역이 있는지, 그리고 어느 영역이 사용 중이며 어느 영역이 사용 가능한지 등등의 정보를 알고 있어야 한다. 가상 메모리 관련된 정보도 역시 이 자료구조에서 관리된다. task_struct에서 태스크의 메모리와 관련된 내용은 mm이라는 이름의 필드에 담겨져 있다. 이 필드는 mm_struct(~/include/linux/mm_types.h)라는 구조체를 가리킨다.

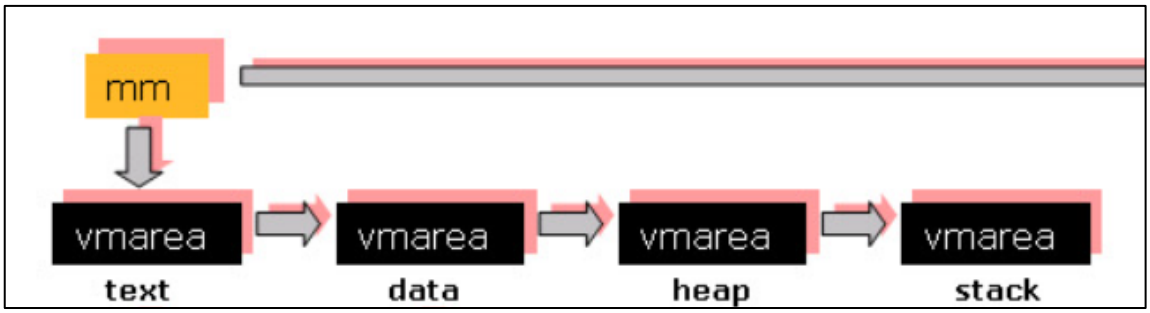
```
struct mm_struct                *mm;
struct mm_struct                *active_mm;
```

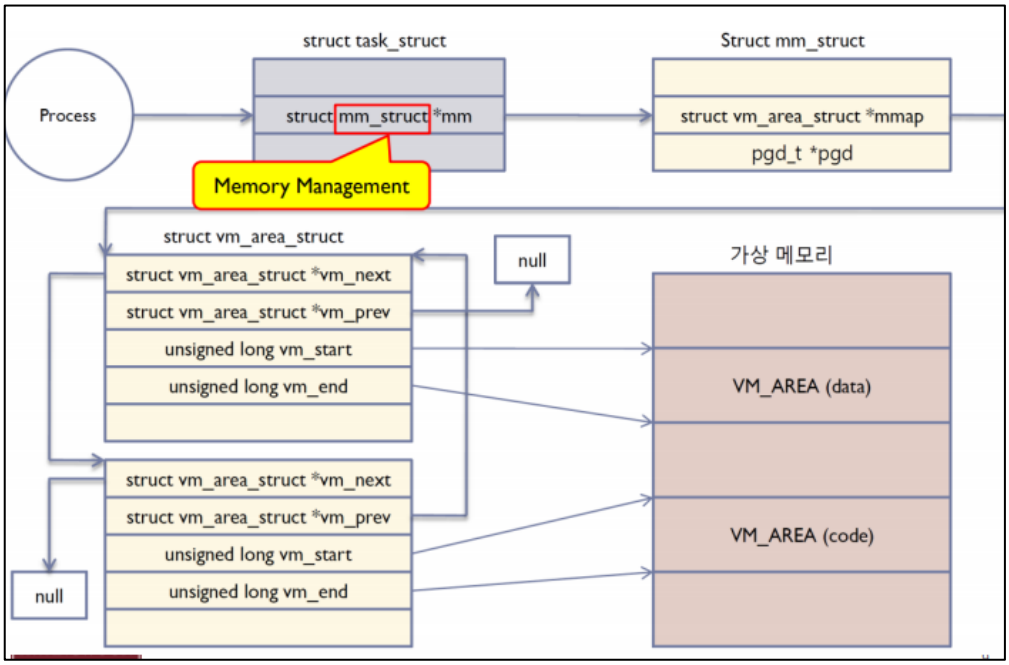
~/include/linux/sched.h 파일에 있는 taks_struct의 mm 필드이다.

```
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
                unsigned long (*get_unmapped_area) (struct file *filp,
                                unsigned long addr, unsigned long len,
                                unsigned long pgoff, unsigned long flags);
#endif
                unsigned long mmap_base;        /* base of mmap area */
                unsigned long mmap_legacy_base; /* base of mmap area in bottom-up allocations */
#ifdef CONFIG_HAVE_ARCH_COMPAT_MMAP_BASES
                /* Base adresses for compatible mmap() */
                unsigned long mmap_compat_base;
                unsigned long mmap_compat_legacy_base;
#endif
                unsigned long task_size;        /* size of task vm space */
                unsigned long highest_vm_end;   /* highest vma end address */
                pgd_t * pgd;

                /**
                 * @mm_users: The number of users including userspace.
                 *
                 * Use mmget()/mmget_not_zero()/mmput() to modify. When this
                 * drops to 0 (i.e. when the task exits and there are no other
                 * temporary reference holders), we also release a reference on
                 * @mm_count (which may then free the &struct mm_struct if
                 * @mm_count also drops to 0).
                 */
                atomic_t mm_users;
```

mm_struct는 vm_area_struct 구조체 멤버를 가진다. VMA 리스트를 의미한다. 리눅스 커널은 가상 메모리 공간 중 같은 속성을 가지며 연속인 영역을 region이라는 이름으로 부른다. 즉, 일반적인 용어 '세그먼트'를 리눅스 용어로 번역하면 'region'인 것이다. 리눅스는 각각의 region을 vm_area_struct라는 자료구조를 통해 관리한다.





```
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_struct에는 vm_start, vm_end가 있다. vm_start는 세그먼트의 시작 주소고 vm_end는 끝 주소이다. vm_next와 vm_prev는 리스트를 연결하기 위한 것이다. 다음 vm_area와 이전 vm_area를 의미한다.

```
        struct file * vm_file;          /* File we map to (can be NULL). */
        void * vm_private_data;          /* was vm_pte (shared mem) */
```

vm_area_struct에는 vm_file도 존재한다. vm_file은 이 세그먼트가 실제 실행 파일의 어느 위치에 있는지에 대한 정보이다.

```
        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;

        unsigned long saved_auxv[AT_VECTOR_SIZE]; /* for /proc/PID/auxv */
```

mm_struct에는 코드 세그먼트 영역, 데이터 세그먼트 영역, 힙의 시작과 끝 주소, 스택의 시작 위치등을 가리키는 변수들이 존재한다.

- ftrace Hooking

```
19 void make_rw(void *addr){
20         unsigned int level;
21         pte_t *pte = lookup_address((u64)addr, &level);
22
23         if(pte->pte &~ _PAGE_RW)
24                 pte->pte |= _PAGE_RW;
25 }
26
27 void make_ro(void *addr){
28         unsigned int level;
29         pte_t *pte = lookup_address((u64)addr, &level);
30
31         pte->pte = pte->pte &~ _PAGE_RW;
32 }
```

make_rw, make_ro는 addr이 속해 있는 페이지의 읽기 및 쓰기 권한을 부여하는 함수이다. 기본적으로. system call table은 쓰기 권한이 존재하지 않는다. make_rw를 호출하여 쓰기 권한이 없는 system call table에 쓰기 권한을 부여하고 make_ro를 호출하여 쓰기 권한을 회수한다

```
34 static asmlinkage void file_varea(pid_t trace_task){
35
36         if(current==NULL)
37                 return;
38         //print process name and pid
39         printk(KERN_INFO "####### Loaded files of a process '%s(%ld)' in VM ######'\n",current->comm, current->pid);
40
41         struct mm_struct *mm;
42         struct vm_area_struct *mmap;
43         struct file *file;
44
45         char * filename;
46         char buf[1024];
47         mm = current->mm;
48         mmap = mm->mmap;
```

line 36~37: 현재 프로세스에 오류가 있으면 종료한다.

line 39 : 처음 출력

line 41 : 프로세스마다 가지는 task_struct의 mm 멤버를 저장하는 변수이다.

line 42 : mm_struct의 vm_area_struct 리스트의 시작 주소를 저장하는 변수이다.

line 43 : 파일 정보를 저장할 변수다.

line 45~46 : 파일 이름을 출력하기 위한 변수

line 47 ~ 48 : 현재 프로세스의 정보를 통해 mm과 mmap에 정보를 할당한다.

```
50         while(mmap){
51                 unsigned long vm_start = mmap->vm_start;
52                 unsigned long vm_end = mmap->vm_end;
53                 unsigned long start_code = mm-> start_code;
54                 unsigned long end_code = mm-> end_code;
55                 unsigned long start_data = mm->start_data;
56                 unsigned long end_data = mm->end_data;
57                 unsigned long start_heap = mm->start_brk;
58                 unsigned long end_heap= mm->brk;
59
60                 file = mmap->vm_file;
61                 if(file){
62                         filename = d_path(&file->f_path,buf,1024);
63                         printk(KERN_INFO "mem[%ld ~ %ld] code[%ld ~ %ld] data[%ld ~ %ld
   ] heap[%ld ~ %ld] %s\n",vm_start,vm_end,start_code,end_code,start_data,end_data,start_h
   eap,end_heap,filename);
64                 }
65                 mmap = mmap->vm_next;
```

mmap는 리스트 형태이다. 반복문을 통해 모든 요소들을 조회한다.

line 51 ~ 59 : 현재 mmap의 가상메모리 시작 주소를 알아낸다. code영역, heap 영역, stack, data 영역 등의 주소를 저장한다.

line 60 : mmap->vm_file을 통해 file 정보를 얻어낸다.

line 61 : file이 유효하면 파일 이름과 함께 출력한다.

line 65 : 다음 vm_area 요소로 넘어간다.

[출력 결과]

```
[ 1808.823038] ####### Loaded files of a process 'assign4(3727)' in VM ######'
[ 1808.823041] mem[4194304 ~ 4198400] code[4194304 ~ 4196172] data[6295056 ~ 6295616] heap[15720448 ~ 15720448] /home/os2018204058/assign4/assign4
[ 1808.823042] mem[6291456 ~ 6295552] code[4194304 ~ 4196172] data[6295056 ~ 6295616] heap[15720448 ~ 15720448] /home/os2018204058/assign4/assign4
[ 1808.823043] mem[6295552 ~ 6299648] code[4194304 ~ 4196172] data[6295056 ~ 6295616] heap[15720448 ~ 15720448] /home/os2018204058/assign4/assign4
[ 1808.823045] mem[140510983221248 ~ 140510985056256] code[4194304 ~ 4196172] data[6295056 ~ 6295616] heap[15720448 ~ 15720448] /lib/x86_64-linux-gnu/libc-2.23.so
[ 1808.823046] mem[140510985056256 ~ 140510987153408] code[4194304 ~ 4196172] data[6295056 ~ 6295616] heap[15720448 ~ 15720448] /lib/x86_64-linux-gnu/libc-2.23.so
[ 1808.823047] mem[140510987153408 ~ 140510987169792] code[4194304 ~ 4196172] data[6295056 ~ 6295616] heap[15720448 ~ 15720448] /lib/x86_64-linux-gnu/libc-2.23.so
[ 1808.823048] mem[140510987169792 ~ 140510987177984] code[4194304 ~ 4196172] data[6295056 ~ 6295616] heap[15720448 ~ 15720448] /lib/x86_64-linux-gnu/libc-2.23.so
[ 1808.823049] mem[140510987194368 ~ 140510987350016] code[4194304 ~ 4196172] data[6295056 ~ 6295616] heap[15720448 ~ 15720448] /lib/x86_64-linux-gnu/ld-2.23.so
[ 1808.823050] mem[140510989443072 ~ 140510989447168] code[4194304 ~ 4196172] data[6295056 ~ 6295616] heap[15720448 ~ 15720448] /lib/x86_64-linux-gnu/ld-2.23.so
[ 1808.823052] mem[140510989447168 ~ 140510989451264] code[4194304 ~ 4196172] data[6295056 ~ 6295616] heap[15720448 ~ 15720448] /lib/x86_64-linux-gnu/ld-2.23.so
[ 1808.823052] ########################################################
```

B. Assignment4-2

Dynamic Recompliation : 프로그램 수행 중에 시스템이 컴파일을 다시 하는 것이다.

이를 수행함으로써, 프로그램의 런타임 환경에 맞춤화 하고 더 효율적인 코드를 생성할 수 있다.

ojbdump : 바이너리에 있는 기계어를 어셈블리 코드로 변환해주는 옵션

Shared Memory란?

메모리 상의 특정 공간에 여러 프로그램이 동시에 접근하여 데이터를 주고 받을 수 있는 IPC 기법 중 하나. IPC 기법 중 가장 빠른 수행 속도를 보인다. 불필요한 memory copy가 발생하지 않는다. Shared memory에는 한 번에 하나의 프로세스가 접근하고 있음을 보증해야 한다.

프로세스가 shared memory를 요청한다. 커널이 공간 할당을 해준다. 커널이 내부 자료구조 shmid_ds를 이용해 직접 관리한다.

[D_recompile_test.c 분석]

```
 7 int Operation(int a)
 8 {
 9         __asm__(
10         ".intel_syntax;"
11         "mov %%eax, %1;"
12         "mov %%dl, 2;"
13         "add %%eax, 1;"
14         "add %%eax, 1;"
15         "add %%eax, 1;"
16         "add %%eax, 1;"
17         "add %%eax, 2;"
18         "add %%eax, 3;"
19         "add %%eax, 1;"
20         "add %%eax, 2;"
21         "add %%eax, 1;"
22         "add %%eax, 1;"
23         "imul %%eax, 2;"
```

```
1079 int main(void)
1080 {
1081         uint8_t* func = (uint8_t*)Operation;
1082         int i = 0;
1083         int segment_id;
1084         uint8_t* shared_memory;
1085         segment_id = shmget(1234, PAGE_SIZE, IPC_CREAT | S_IRUSR | S_IWUSR);
1086         shared_memory = (uint8_t*)shmat(segment_id, NULL, 0);
1087         do
1088         {
1089         shared_memory[i++] = *func;
1090         } while (*func++ != 0xC3);
1091
1092         shmdt(shared_memory);
1093         printf("Data was filled to shared memory.\n");
1094         return 0;
1095 }
```

line 1081 : 함수 포인터 선언. 함수 포인터라 함은 함수를 가리키는 포인터다. 함수에도 주소가 있다. (어셈블리에서 symbol!)

line 1083 : shared memory segemnt를 식별하는 integer형 변수를 선언한다.

line 1084 : shared memory에서 uint8_t* 형을 가리킬 포인터를 선언한다.

line 1085 : shmget()을 사용하여 커널에게 shared memory 공간을 요청한다. 1234는 shared memory에 접근할 수 있는 키다. PAGE_SIZE, shared memory의 최소 크기다. 접근 권한은 IPC_CREAT | S_IRUSR | S_IWUSR이다.

IPC_CREAT : 새로운 영역을 할당한다. 이미 키에 해당하는 메모리 존재하면 그것을 반환한다.

S_IRUSR : owner에 대한 read 권한 허용

S_IWUSR :owner에 대한 write 권한 허용

line 1086 : 현재 프로세스가 만들어진 shared memory를 사용할 수 있게 한다.

line 1087 ~ 1090 : Operation 함수의 내용을 shared memory에 올린다.

line 1092 : 공유 메모리를 dettach한다.

```
os2018204058@ubuntu:~/assign4-2$ gcc D_recompile_test.c
os2018204058@ubuntu:~/assign4-2$ ls
a.out  D_recomile.c  D_recompile_test.c  D_recompile_test.o  test
os2018204058@ubuntu:~/assign4-2$ ./a.out
Data was filled to shared memory.
os2018204058@ubuntu:~/assign4-2$ ipcs -m

------ Shared Memory Segments --------
key        shmid      owner      perms      bytes      nattch     status
0x00000000 294912     os20182040 600        524288     2          dest
0x00000000 1310721    os20182040 600        524288     2          dest
0x00000000 425986     os20182040 600        524288     2          dest
0x00000000 688131     os20182040 600        524288     2          dest
0x00000000 786436     os20182040 600        524288     2          dest
0x00000000 950277     os20182040 600        524288     2          dest
0x00000000 1048582    os20182040 600        524288     2          dest
0x00000000 1212423    os20182040 600        524288     2          dest
0x00000000 1114120    os20182040 600        16777216   2          dest
0x00000000 1409033    os20182040 600        524288     2          dest
0x00000000 1441802    os20182040 600        33554432   2          dest
0x000004d2 1474571    os20182040 600        4096       0
```

맨 마지막줄에 1234를 키로 가지는 shared memory가 생긴 것을 볼 수 있다.

[D_recompile.c 분석]

```
19 int main(void)
20 {
21         int (*func)(int a);
22         int i;
23
24         // time start
25         struct timespec begin, end;
26         clock_gettime(CLOCK_MONOTONIC, &begin);
27
28         sharedmem_init();
29         drecompile_init(Operation);
30
31         func = (int (*)(int a))drecompile(Operation);
32
33         //int result =func(4);
34         //printf("result : %d\n",result);
35         drecompile_exit();
36         sharedmem_exit();
37
38         //time end
39         clock_gettime(CLOCK_MONOTONIC,&end);
40
41         struct timespec temp;
42         long msec = end.tv_nsec - begin.tv_nsec;
43         if(msec<0)
44                 msec = 1000000000 + msec;
45         printf("total execution time : %lf\n", (double)msec/100000);
46         return 0;
47 }
48
```

line 21 : 함수 포인터 선언. 반환형은 int, 매개변수로 int를 가짐.

line 25~26 : 시간을 측정하기 위해 변수를 선언하고 측정한다.

line 28 : sharedmem_init()을 통해 D_recompile_test.c에서 공유한 메모리에 접근할 것이다.

line 29: drecompile_init() - 매모리 맵핑을 한다. Operation에 담긴 공유메모리의 원본 코드를

compiled_code 안에 카피한다.

line 31 : func를 통해서 계산된 결과를 확인할 수 있다.

line 35 ~ 36 : 공유 메모리를 해제하고 unmap을 한다.

line 39 : 끝나는 시간을 측정한다.

line 41 ~ 46 : 시간을 출력한다.

mmap란?

mmap는 메모리의 내용을 파일이나 디바이스에 mapping하기 위해서 사용하는 시스템 호출이다.

메모리 관리와 mmap

각각의 프로세스는 프로세스마다 다른 프로세스와 중복되지 않는 주소공간을 가진다. 주소 공간은 3개의 세그먼트로 분할된다. 코드, 데이터와 스택이 그것이다. 코드 세그먼트는 읽기 전용으로 프로그램의 명령을 포함하고 있다. 데이터와 스택 세그먼트는 읽기, 쓰기가 모두 가능한 영역이다.

프로세스 메모리는 기본적으로 다른 프로세스 메모리와 공유되지 않는다. 이것은 프로세스의 데이터를 보호하기 위해서 반드시 필요한 기능이다. 하지만 다른 프로세스와 특정 데이터를 공유해야 할 때에도 있다. 이 때문에 IPC를 사용하게 된다. mmap는 메모리의 특정 영역을 파일로 대응시킬 수 있도록 도와준다. 파일은 시스템 전역적인 객체이므로 다른 프로세스에서 접근가능하도록 할 수 있으며, 이러한 mmap 특징 때문에 IPC 용도로 사용가능하다.

* mmap는 프로세스의 주소공간을 파일에 대응 시킨다. 파일은 운영체제 전역적인 자원이므로 당연히 어렵잖게 다른 프로세스와 공유해서 사용할 수 있다.

mmap의 활용 용도:

메모리의 내용을 파일에 대응시켜 얻을 수 있는 이익은?

1. 메모리의 내용을 파일에 대응시킬 수 있다면 프로세스간 데이터 교환을 위한 용도로 사용 가능하다. 프로세스간 공유하고자 하는 데이터를 파일에 대응시키고 이것을 읽고 쓰면 된다. 접근 제어 필요
2. 메모리의 내용을 파일에 직접 대응시킨다면 성능 향상을 생각할 수 있을 것이다. 고전적인 방법은 파일 지정자를 얻어서 직접 입출력하는 방식으로 open, read, write, lseek과 같은 함수를 이용한다. 이러한 함수의 사용은 당연하지만 상당한 비용을 지불해야 하는데, mmap를 이용하면 비용을 줄일 수 있다.

mmap 함수는 start부터 length까지의 메모리 영역을 열린 파일 fd에 대응한다. 대응 할 때 파일의 위치를 지정할 수 있는데 이것은 offset을 통해서 이루어진다

PROT_EXEC : 페이지(page)는 실행될 수 있다.

PROT_READ : 페이지는 읽혀질 수 있다.

PROT_WRITE : 페이지는 쓸 수 있다.

PROT_NONE : 페이지를 접근할 수 없다.

mmap를 사용할 때 반드시 MAP_PRIVATE와 MAP_SHARED 둘 중 하나를 사용해야 한다.

- sharedmem_init()

```
void sharedmem_init()
{
        // D_recompile_test.c , 1234 is the shared memory key
        int size = PAGE_SIZE;
        int segment_id=shmget(1234,size,0);
        Operation = (uint8_t*)shmat(segment_id,NULL,0);
        return;
}
```

Operation 변수에 공유 메모리에 올려놓은 함수의 주소를 가리키게한다. 이제 Operation로 D_recompile_test.c에서 공유메모리에 올린 Operation 함수를 접근할 수 있다.

- sharedmem_exit()

```
void sharedmem_exit()
{
        shmdt(Operation);
}
```

프로세스가 더 이상 shared memory를 사용 할 필요가 없으니 프로세스에서 공유 메모리를 분리한다.

- drecompile_init()

```
void drecompile_init(uint8_t *func)
{
        int pagesize;
        int fd;
        pagesize = getpagesize();
        compiled_code = mmap(0,pagesize, PROT_READ | PROT_WRITE, MAP_ANONYMOUS | MAP_PRIVAT
E, fd,0);
        compiled_code=(uint8_t*)Operation;

//      printf("Hello!\n");
//      printf("%d\n",*(compiled_code+20));
//      msync(compiled_code, pagesize, MS_ASYNC);
}
```

mmap를 통해서 프로세스의 주소공간을 파일에 대응 시킨다. 이 때, 임의의 파일포인터 fd를 선언했다. MAP_ANONYMOUS, MAP_PRIVATE 설정을 했다.

- MAP_ANONYMOUS : 익명페이지란 의미이다. 익명페이지는 커널로부터 프로세스에게 할당된 일반적인 메모리 페이지이다. 힙을 거치지 않고 할당 받는다. '익명'이라는 뜻은 파일에 기반하지 않고 있지 않은 (파일로부터 맵핑되지 않은) 페이지라는 듯이다. 파일에 맵핑되어 있다면 파일의 내용이 들어가겠지만, 파일에 매핑되어있지 않으니 0으로 초기화 되어있다.
- MAP_PRIVATE : 데이터의 변경 내용을 공유하지 않는다.

compile_code에 Operation이 가리키는 함수를 복사한다. 이제 compile_code에는 공유메모리에 올라와 있던 함수가 저장되어 있다! 이 함수를 읽고 수정할 수 있을 것이다.

- drecompile_exit()

```
void drecompile_exit()
{       int pagesize;
        pagesize = getpagesize();
        munmap(compiled_code,pagesize);
}
```

메모리에 매핑되어 있던 것을 unmap한다.

- objdump 확인하기

```
os2018204058@ubuntu:~/assign4-2$ gcc -c D_recompile_test.c
os2018204058@ubuntu:~/assign4-2$ objdump -d D_recompile_test.o > test
os2018204058@ubuntu:~/assign4-2$ ls
D_recomile.c  D_recompile_test.c  D_recompile_test.o  test
```

```
0000000000000c48 <main>:
 c48:   55                      push   %rbp
 c49:   48 89 e5                mov    %rsp,%rbp
 c4c:   48 83 ec 20             sub    $0x20,%rsp
 c50:   48 c7 45 f0 00 00 00    movq   $0x0,-0x10(%rbp)
 c57:   00
 c58:   c7 45 e8 00 00 00 00    movl   $0x0,-0x18(%rbp)
 c5f:   ba 80 03 00 00          mov    $0x380,%edx
 c64:   be 00 10 00 00          mov    $0x1000,%esi
 c69:   bf d2 04 00 00          mov    $0x4d2,%edi
 c6e:   e8 00 00 00 00          callq  c73 <main+0x2b>
 c73:   89 45 ec                mov    %eax,-0x14(%rbp)
 c76:   8b 45 ec                mov    -0x14(%rbp),%eax
 c79:   ba 00 00 00 00          mov    $0x0,%edx
 c7e:   be 00 00 00 00          mov    $0x0,%esi
 c83:   89 c7                   mov    %eax,%edi
 c85:   e8 00 00 00 00          callq  c8a <main+0x42>
 c8a:   48 89 45 f8             mov    %rax,-0x8(%rbp)
 c8e:   8b 45 e8                mov    -0x18(%rbp),%eax
 c91:   8d 50 01                lea    0x1(%rax),%edx
 c94:   89 55 e8                mov    %edx,-0x18(%rbp)
 c97:   48 63 d0                movslq %eax,%rdx
 c9a:   48 8b 45 f8             mov    -0x8(%rbp),%rax
```

main함수의 어셈블리 코드 내용이다

```
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
"test" 1120L, 50304C
```

```
c11:   83 e8 01                sub    $0x1,%eax
c14:   83 e8 01                sub    $0x1,%eax
c17:   83 e8 03                sub    $0x3,%eax
c1a:   83 e8 01                sub    $0x1,%eax
c1d:   83 e8 01                sub    $0x1,%eax
c20:   83 e8 01                sub    $0x1,%eax
c23:   83 e8 03                sub    $0x3,%eax
c26:   83 e8 01                sub    $0x1,%eax
c29:   83 e8 01                sub    $0x1,%eax
c2c:   83 e8 02                sub    $0x2,%eax
c2f:   83 e8 01                sub    $0x1,%eax
c32:   83 e8 01                sub    $0x1,%eax
c35:   83 e8 01                sub    $0x1,%eax
c38:   83 e8 01                sub    $0x1,%eax
c3b:   83 e8 01                sub    $0x1,%eax
c3e:   89 c2                   mov    %eax,%edx
c40:   89 55 fc                mov    %edx,-0x4(%rbp)
c43:   8b 45 fc                mov    -0x4(%rbp),%eax
c46:   5d                      pop    %rbp
c47:   c3                      retq
```

Operation 함수의 어셈블리 내용이다

주소 c에는 나눗셈 할 때 사용할 dl값을 채우고 있다. dl을 2로 설정한 것 같다.

주소 14부터 최적화할 instruction이 시작된다. add 명령어가 여기서부터 나왔기 때문이다.

add : 83 c0 더해지는 숫자

sub : 83 e8 빼지는 숫자

imul : 6b c0 곱해지는 수

div : f6 f2

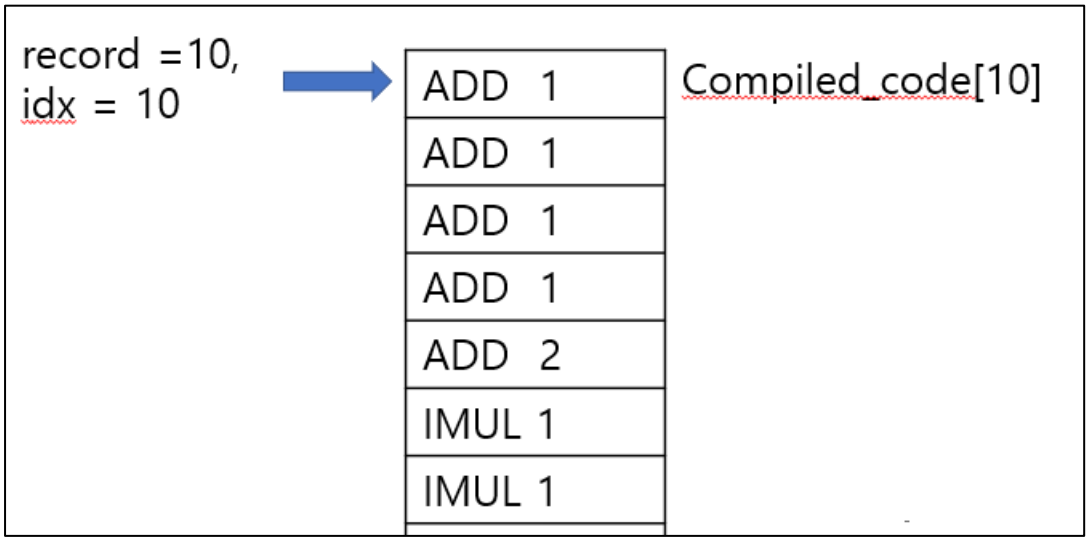
add와 sub를 구분 할 때에는 일단 첫번째 코드는 83으로 같지만 두번째 코드가 c0과 e8로 달라지는 걸 이용하려 했다. 나머지 imul과 div는 첫번째 코드의 수가 유니크해서 구분할 수 있었다.

```
c46:   5d                      pop    %rbp
c47:   c3                      retq
```

retq를 만나면 함수의 종료이다. c3값을 만나면 최적화를 종료할 계획이었다.

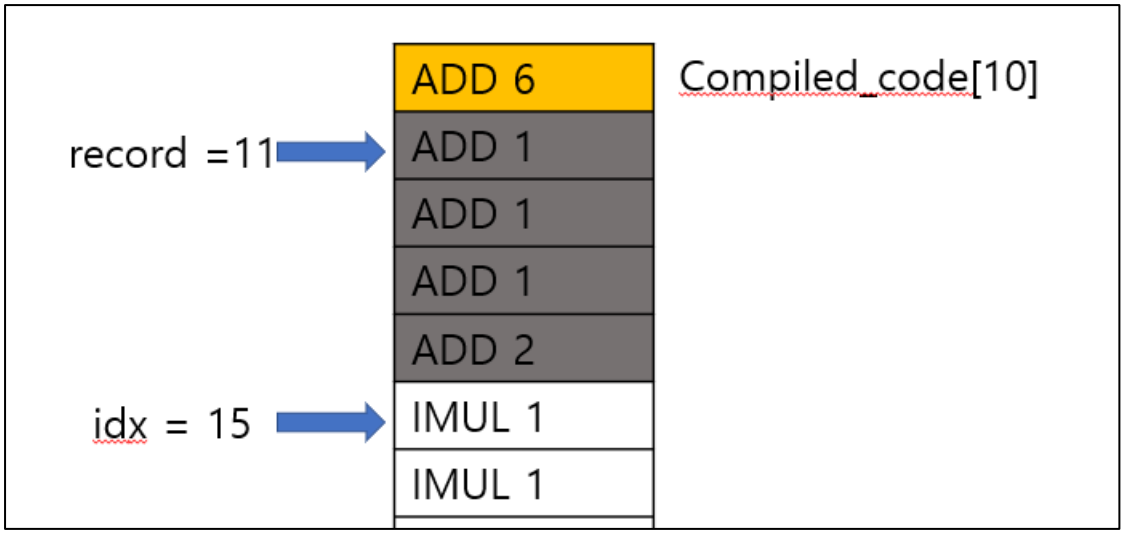
- drecompile()

record, idx 변수 설명



초기 record 값은 10, idx 값은 10이다.

record는 최적화할 코드를 적을 때 필요한 인덱스이고, idx는 Operation 코드를 읽을 때 필요한 인덱스이다.



ADD 명령어를 최적화 하고 난 후, record의 값을 +1 한다. 다음 최적화 할 코드를 적을 곳은 현재 코드 다음이다. 이제 Operation 코드를 읽을 곳은 15이다.

```
66 void* drecompile(uint8_t* func)
67 {
68         int start = compiled_code[10];
69         int record_point = 10;//compiled_code[10];
70         int ist = start;
71         int idx = 10;
72         int num = 0;
73         int prev=-1;
74
75         enum Instruction {ADD, SUB, IMUL, DIV};
```

line 68 : 최적화 할 부분의 시작 주소이다. 처음 add instruction이 나오는 곳이다.

line 69 : 최적화한 코드를 적을 주소이다.

line 70 : ist는 instruction을 의미한다. 명령어라는 의미다.

line 71 : idx는 compile_code에서 읽을 주소를 가리키는 index이다.

line 72 : num 변수는 최적화 하기위해 숫자들을 누적하는 것이다. ex) add++ , add++ -> num에는 2가 들어감.

line 73 : prev는 이전에 나온 명령어를 저장하는 것이다.

line 75 : enum type으로 명령어의 종류를 선언했다.

```
76         while(1){
77         ist = compiled_code[idx];
78         if( ist ==0x83 ){
79                 if( compiled_code[idx+1] == 0xc0){
80                         // it is add instruction
81                         if(prev != ADD){
82                                 //different instruction
83                                 compiled_code[record_point]=0x83;
84                                 compiled_code[record_point+1]=0xc0;
85                                 compiled_code[record_point+2]=num;
86                                 record_point=+3;
87                                 num=0;
88                         }
89                         num +=compiled_code[idx+2];
90                         prev = ADD;
91                         idx+=3;
92                 }
```

line 76 : is가 return을 의미하는 0xc3이면 종료할 것이다. 그렇지 않으면 아직 코드를 다 안봤다는 의미이니 무한 반복문으로 선언한다.

line 78 : ist가 0x83이면 ADD나 SUB 명령어이다.

line 79 : ist 다음 자리가 0xc0이면 ADD 명령어란 의미이다.

line 81 ~ 88 : 만약 이전 명령어가 ADD명령어가 아니면, 그동안 합쳐놓은 num으로 한번에 명령을 수행한다. 83부터 85줄은 compiled_code에 최적화한 명령어를 적는 코드이다. record_point는 기록할 주소인데 이것 또한 +3을 해준다. 또 누적값을 저장하는 num을 0으로 초기화 한다.

idx는 compiled_code에서 읽을 값이다. +3을 해주어서 다음 명령어를 읽는다.

```
93                  else if (compiled_code[idx+1] == 0xe8){
94                  // it is sub instruction
95                          if(prev != SUB){
96                                  //different instruction
97                                  compiled_code[record_point]=0xe8;
98                                  compiled_code[record_point+1]=0xe8;
99                                  compiled_code[record_point+2]=num;
100                                 record_point+=3;
101                                 num=0;
102                         }
103                         num += compiled_code[idx+2];
104                         prev = SUB;
105                         idx+=3;
106                 }
107         }
```

line 93: ist 다음 자리가 0xe8이면 SUB 명령어란 의미이다.

line 95 ~ 102 : 만약 이전 명령어가 SUB명령어가 아니면, 그동안 합쳐놓은 num으로 한번에 명령을 수행한다. 97부터 99줄은 compiled_code에 최적화한 명령어를 적는 코드이다. record_point는 기록할 주소인데 이것 또한 +3을 해준다. 또 누적값을 저장하는 num을 0으로 초기화 한다.

idx는 compiled_code에서 읽을 값이다. +3을 해주어서 다음 명령어를 읽는다.

```
108         else if( ist == 0x6b){
109                 // it is imul
110                 if(prev != IMUL){
111                         //different instruction
112                         compiled_code[record_point]=0x6b;
113                         compiled_code[record_point+1]=0xc0;
114                         compiled_code[record_point+2]=num;
115                         record_point+=3;
116                         num=0;
117                 }
118                 num+=compiled_code[idx+2];
119                 prev=IMUL;
120                 idx+=3;
121         }
```

line 108: ist가 0x6b면 IMUL 명령어란 의미이다.

line 110 ~ 116 : 만약 이전 명령어가 IMUL명령어가 아니면, 그동안 합쳐놓은 num으로 한번에 명령을 수행한다. 112부터 115줄은 compiled_code에 최적화한 명령어를 적는 코드이다. record_point는 기록할 주소인데 이것 또한 +3을 해준다. 또 누적값을 저장하는 num을 0으로 초기화 한다.

idx는 compiled_code에서 읽을 값이다. +3을 해주어서 다음 명령어를 읽는다.

```
121         }
122         else if (ist == 0xf6){
123                 // it is div
124                 if(prev !=DIV){
125                         //different instruction
126                         compiled_code[record_point]=0xf6;
127                         compiled_code[record_point+1]=num;
128                         record_point+=2;
129                         num=0;
130                 }
131                 num+=0x02;
132                 prev=DIV;
133                 idx+=2;
134         }
```

line 122: ist가 0xf6이면 DIV 명령어란 의미이다.

line 124 ~ 130 : 만약 이전 명령어가 DIV명령어가 아니면, 그동안 합쳐놓은 num으로 한번에 명령을 수행한다. 126부터 129줄은 compiled_code에 최적화한 명령어를 적는 코드이다. record_point는 기록할 주소인데 이것 또한 +2을 해준다. 또 누적값을 저장하는 num을 0으로 초기화 한다.

idx는 compiled_code에서 읽을 값이다. +2을 해주어서 다음 명령어를 읽는다.

```
135         else{
136                 compiled_code[record_point] = ist;
137                 record_point+=1;
138                 idx+=1;
139         }
140         if(ist == 0xc3)//this means return
141                 break;
142         }
```

명령어가 ADD, SUB, IMUL, DIV가 아니라면 읽었던 ist를 compiled_code 에 적는다. 다음에 적을 주소를 증가하기 위해 record_point를 +1해준다. 또 다음에 읽을 주소를 증가하기 위해 idx를 +1해준다.

ist가 0xc3이면 return을 의미하므로 함수의 종료이다. break를 해준다.

```
143         return compiled_code;
144 }
```

drecompile함수에서 compiled_code를 리턴한다.

- 결과

```
os2018204058@ubuntu:~/assign4-2$ ./a.out
Segmentation fault (core dumped)
```

오류가 났다…

[Reference]

https://selfish-developer.com/entry/objdump-%EB%A5%BC-%EC%9D%B4%EC%9A%A9%ED%95%9C-%EB%B0%94%EC%9D%B4%EB%84%88%EB%A6%AC-%EA%B9%A8%EB%B3%B4%EA%B8%B0

https://seyun99.tistory.com/entry/mmap%EC%9D%B4%EB%9E%80

https://showx123.tistory.com/70

https://showx123.tistory.com/92

https://hackereyes.tistory.com/entry/%EB%B0%94%EB%9E%8C%EC%9D%B4-mmstruct-%EC%97%90-%EB%8C%80%ED%95%9C-%EA%B0%9C%EC%9A%94

https://beausty23.tistory.com/109

https://velog.io/@mythos/%EC%BB%A4%EB%84%90-%EC%8A%A4%ED%84%B0%EB%94%94iamroot-18%EA%B8%B0-4%EC%A3%BC%EC%B0%A8-%EB%82%B4%EC%9A%A9-%EC%A0%95%EB%A6%AC-2

https://hodev.tistory.com/117

https://stackoverflow.com/questions/15114233/linux-is-it-possible-to-share-code-between-processes