시스템 프로그래밍 과제 #2 보고서

2019147542 하동현

1. 사전 조사 보고서

a. 2.6.29 버전 / 6.2.0 버전의 메모리 관리 기법 커널 코드 비교 분석

사전 조사에서는 커널 2.6.29 버전과 6.2.0 버전의 메모리 관리 기법 코드의 공통점과 차이점을 중심으로 조사해보려고 한다.

1. Multi-level paging을 위한 각 버전별 자료 구조

2.6.29 버전에서는 PGD, PUD, PMD, PTE의 4-level paging 기법을 사용했다. 하지만 6.2.0 버전에서는 PGD와 PUD 사이에 P4D라는 새로운 level이 추가되어 5-level paging을 사용하고 있다.

Virtual address를 physical address로 변환할 때 사용하는 자료 구조, 매크로, 함수들은 include/linux/mm_types.h 와 include/linux/pgtable.h에 대부분 정의되어 있으므로 각 버전별로 저 두 헤더파일의 코드를 비교하고 분석할 것이다.

(1) 2.6.29 버전

in include/asm/page.h:

```
typedef struct { pgdval_t pgd; } pgd_t;

typedef struct { pudval_t pud; } pud_t;

typedef struct { pmdval_t pmd; } pmd_t;

typedef unsigned long pte_t;
```

pgd_t, pud_t, pmd_t, pte_t 등의 자료형을 통해 각 페이지 테이블 엔트리를 나타내는 자료형임을 확인할 수 있었다.

(2) 6.2.0 버전

6.2.0 버전에서도 위와 같은 pgd_t, pud_t, pmd_t, pte_t 등의 자료형이 선언되어 있다. 하지만 6.2.0 버전에서는 pgd와 pud 사이에 p4d라는 새로운 테이블이 생겼다. 커널이 발전하고 컴퓨터의 정보량이 많아짐에 따라, multi-level paging에서도 level이 늘어나야 했다고 판단한 것 같다.

in include/asm/pagetable_types.h:

```
typedef struct { p4dval_t p4d; } p4d_t;
```

2.6.29 버전에는 존재하지 않던 p4d_t 자료형이 새로 생겼음을 확인할 수 있었다.

2. PGD, (P4D), PUD, PMD, PTE의 관계 분석

(1) 2.6.29 버전

강의안을 참조하면, 2.6.11 버전 이후의 리눅스 paging은 아래와 같은 방식을 따르고 있다.

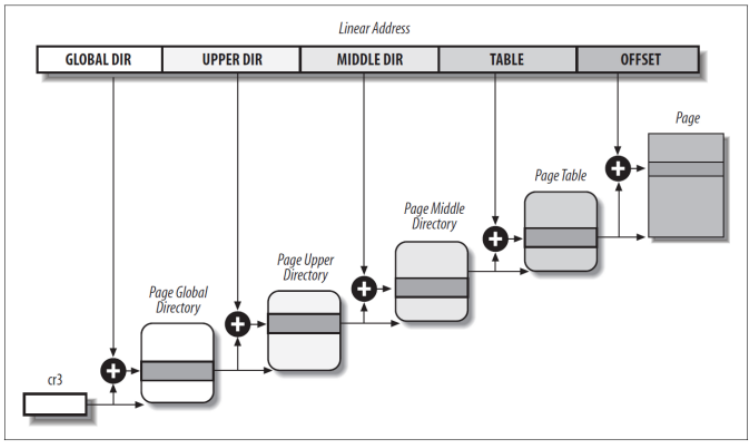
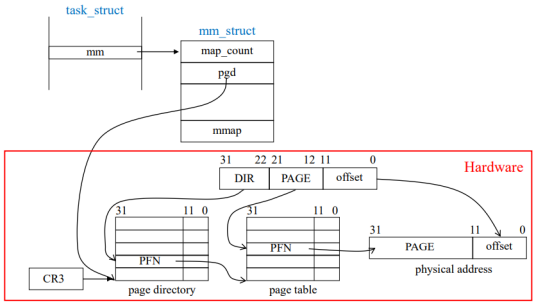


Figure 2-12. The Linux paging model

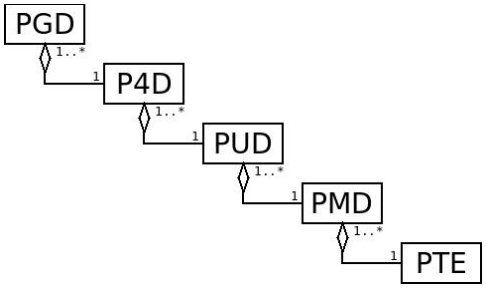
먼저 cr3 레지스터나 task_struct의 mm_struct에 있는 pgd 변수를 통해 page global directory의 base address를 찾는다. 리눅스의 페이징은 기본적으로 이 base address와 주어진 Linear address를 통해, 커널이 구동되는 아키텍처가 사용하는 구체적인 multi-level paging 기법에 따라 설정된 오프셋을 테이블마다 가져와 합치며 하위 테이블을 찾는다. 다시 말해, 각 테이블의 오프셋에 해당하는 비트만큼을 상위 테이블의 페이지 디렉토리에서 찾은 값에 나눠 더해서 하위 테이블로 내려가 찾고자 하는 page를 가져온다.



위 설명을 간단하게 2-level paging이라고 가정하면 다음과 같이 나타낼 수 있다.

(2) 6.2.0 버전

6.2.0 버전에서는 2.6.11 버전 이후의 4-level paging에서 pgd와 pud 사이에 p4d라는 새로운 테이블이 도입됨을 알 수 있었다.

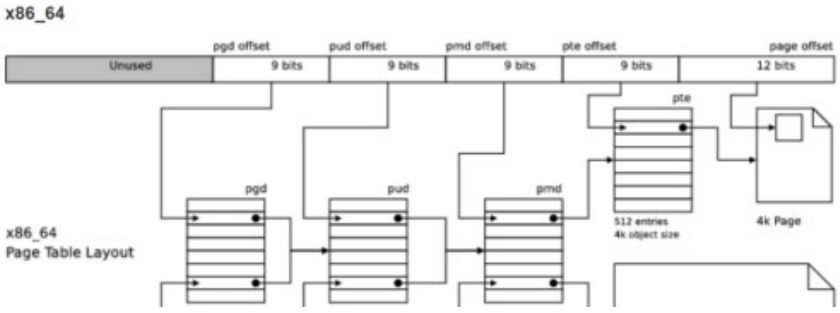


다음 도식을 보면 pgd의 하위 테이블이 p4d가 되고, p4d의 하위 테이블이 pud라는 것을 알 수 있었다.

3. Process별로 할당된 page 정보와 매크로, 함수를 통해 물리 메모리를 참조하는 과정

위에서 개략적으로 가상 메모리 주소가 페이징을 통해 물리 메모리를 참조하는 과정을 알아보았다. 그럼 실제로 매크로와 함수를 통해 virtual adrress가 pyshical address로 변환되는 과정을 조사해 보겠다.

(1) 2.6.29 버전



2.6.29 버전, 그리고 intel x86_64 아키텍처 기반 커널일 때, 주소의 계산 방법까지 기록된 강의안의 4-level-paging 구조도를 참조하여, 실제 매크로와 함수를 통한 변환 과정을 알아보려고 한다.

기본적으로 커널에서는 pgd_offset, pud_offset, pmd_offset, pte_offset_kernel 등의 매크로를 이용해 하위 테이블의 엔트리 주소를 계산한다.(base address와 각 table의 offset을 통해 하위 테이블의 엔트리 주소를 계산.)

(a) pgd 테이블에서 찾아야 할 주소 계산

in x86/include/asem/pgtable.h:

```
/*
 * pgd_offset() returns a (pgd_t *)
 * pgd_index() is used get the offset into the pgd page's array of pgd_t's;
 */
#define pgd_offset(mm, address) ((mm)->pgd + pgd_index((address)))
```

pgd_offset이라는 매크로는 mm 구조체와 virtual address를 받아, mm에서 pgd base address(cr3 레지스터 값)을 찾아낸다. 또 pgd_index라는 매크로를 사용해 virtual address에서 pgd part에 해당하는 offset을 계산한다.

```
/*
 * the pgd page can be thought of an array like this: pgd_t[PTRS_PER_PGD]
 *
 * this macro returns the index of the entry in the pgd page which would
 * control the given virtual address
 */
#define pgd_index(address) (((address) >> PGDIR_SHIFT) & (PTRS_PER_PGD - 1))
```

pgd_index 매크로를 관찰해 보면, 주소를 PGDIR_SHIFT 매크로 값만큼 right shift하고 bitwise and 연산을 통해 마스크해서 virtual address에서 pgd 의 offset 비트만을 빼낸다. 위에 있는 강의안의 자료(x86_64에서의 offset)와 동일하다면, 이는 virtual address의 중간 9비트만큼을 가져오게 될 것이다.

그 후 pgd base address와 pgd index를 더해 최종적으로 다음 하위 테이블인 pud(6.2.0 버전에선 p4d)에서 어떤 값을 찾아가야 할 지 알려주는 주소를 얻을 수 있게 된다.

(b) pud 테이블에서 찾아야 할 주소 계산

다음은 pud_offset 매크로를 관찰해 보자.

in x86/include/asm/pgtable_64.h:

```
#define pud_index(address) (((address) >> PUD_SHIFT) & (PTRS_PER_PUD - 1))
#define pud_offset(pgd, address)                                        \
        ((pud_t *)pgd_page_vaddr(*(pgd)) + pud_index((address)))
```

pud_offset도 비슷한 매크로 구조를 가지고 있다. 하지만 차이점은, pud_offset은 첫번째 인자로 pgd base address를 가져오는 것이 아닌, 위에서 계산한 pgd에서 받아온 주소(포인터 값) pgd를 가져온다.

```
#define pgd_page_vaddr(pgd)                                             \
        ((unsigned long)__va((unsigned long)pgd_val((pgd)) & PTE_PFN_MASK))
```

또한 pgd 주소를 base address처럼 바로 사용하여 offset과 더하는 것이 아니라, pgd_page_vaddr 매크로를 사용해 한번의 계산을 거친다. 다시 말해, PTE_PFN_MASK 매크로와 and 연산을 통해 뒤의 pud_offset이 더해져야 할 비트들을 0으로 만든다. 즉, pud의 page frame number에 pud_index 매크로를

통해 virtual address에서 pud의 offset 부분만을 가져와 더해서 다음 pmd table에서 찾아가야 할 주소를 얻을 수 있게 되는 것이다.

pud_offset 매크로도 pud에 맞는 PUD_SHIFT 값, PTRS_PER_PUD 등의 bitmask 연산을 위한 매크로를 따로 정의하고 있다.

(c) pmd, pte 테이블 주소 계산

pmd, pte 테이블까지 계속해서 하위 테이블로 이동하는 매크로도 거의 동일한 방식을 가지고 있다.

```
#define pmd_page_vaddr(pmd) ((unsigned long) __va(pmd_val((pmd)) & PTE_PFN_MASK))
#define pmd_page(pmd)       (pfn_to_page(pmd_val((pmd)) >> PAGE_SHIFT))

#define pmd_index(address) (((address) >> PMD_SHIFT) & (PTRS_PER_PMD - 1))
#define pmd_offset(dir, address) ((pmd_t *)pud_page_vaddr(*(dir)) + ₩
                                  pmd_index((address)))

#define pmd_page_vaddr(pmd) ((unsigned long) __va(pmd_val((pmd)) & PTE_PFN_MASK))

#define pte_index(address) (((address) >> PAGE_SHIFT) & (PTRS_PER_PTE - 1))
#define pte_offset_kernel(dir, address) ((pte_t *) pmd_page_vaddr(*(dir)) + ₩
                                         pte_index((address)))
```

위에 있는 pmd_offset 매크로와 pte_offset_kernel 매크로를 통해 최종적으로 pte 테이블에서 page 주소를 찾을 수 있게 될 것이다.

(d) 마지막: pte_value를 통해 physical address 계산

in include/asm/page.h:

```
#define pte_val(x)      ((x).pte)
```

먼저 pte_val 매크로는 주소 x가 가리키는 pte 값을 가져온다.

in /x86/include/asm/pgtable.h:

```
static inline unsigned long pte_pfn(pte_t pte)
{
        return (pte_val(pte) & PTE_PFN_MASK) >> PAGE_SHIFT;
}
```

pte_pfn 매크로는 찾은 pte 테이블의 주소를 통해 page 주소의 값을 찾고, PTE_PFN_MASK를 통한 masking을 통해 뒤의 12비트(이는 2^12bit로, 4KB, 즉 page=frame 한칸의 크기가 된다.) 값을 지운 후 12번 left shift 한다. 즉 다시 말해, 이는 page의 frame number를 구하는 매크로인 것이다. 이 pfn 주소 뒤에 마지막 pte_index, 즉 virtual address의 맨 오른쪽 12자리 offset을 붙힌 값이 결국 우리가 찾는 physical address가 될 것이다. 내 모듈에서는 좀 더 간단하게 pte_value 값의 맨 뒷 12비트를 0으로 바꾸고 그 자리에 offset을 더하는 방식으로 계산하였다. 결과적으로 둘은 같은 값이 나온다는 것을 검증하

였다.

pgd의 시작 주소(pgd base address)와 virtual address를 이용하여, 복잡한 multi-level paging을 통해 결국 physical address를 구하게 된다는 것을 커널 코드를 관찰하며 알 수 있었다. 더불어서 나는 x86 시스템의 paging만 조사했지만, 이 과정이 기반 아키텍처, n-bit 시스템인지에 따라 모두 다른 방식을 사용해야 하기 때문에, 엄청나게 많은 메모리 관리 구현 코드가 존재하며 이들이 매우 정교하게 디자인되어 있는 이유를 깨닫게 되었다.

(2) 6.2.0 버전

6.2.0 버전의 페이징도 2.6.29 버전과 매커니즘 상으로는 거의 동일하나, 위에서 말했듯이 pgd와 pud 사이에 새로운 페이지 테이블이 생겨났다. 메모리 크기, 동시의 처리하는 비트의 양이 과거에 비해 늘어나서 이를 효과적으로 처리하기 위해 새로운 p4d라는 테이블을 만든 것이라고 생각이 되었다.

(a) 새로운 테이블: p4d

```
#define p4d_offset p4d_offset
static inline p4d_t *p4d_offset(pgd_t *pgd, unsigned long address)
{
        if (pgtable_l5_enabled)
                return pgd_pgtable(*pgd) + p4d_index(address);

        return (p4d_t *)pgd;
}
```

2.6.29 버전과 동일하게 p4d_offset이라는 매크로와 함수가 정의되어 있다. 인자로 pgd를 받는 것을 알 수 있다. 상위 테이블이 pgd라는 것을 의미할 것이다.

(b) p4d가 사이에 생성됨으로 변화한 pud 관련 매크로

```
#define pud_index(addr) (((addr) >> PUD_SHIFT) & (PTRS_PER_PUD - 1))

#define pud_offset pud_offset
static inline pud_t *pud_offset(p4d_t *p4d, unsigned long address)
{
        if (pgtable_l4_enabled)
                return p4d_pgtable(*p4d) + pud_index(address);

        return (pud_t *)p4d;
}
```

pud의 상위 테이블이 pgd에서 p4d로 변화했기 때문에, 6.2.0 버전 커널에서의 pud_offset 매크로는 2.6.29에서의 매크로와 차이가 있을 것이다. 위에서는 pud가 pgd의 값을 가져왔지만, 여기서는 p4d를 인자로 받아 바로 윗 상위 테이블인 p4d에서 값을 가져온다는 것을 관찰할 수 있었다.

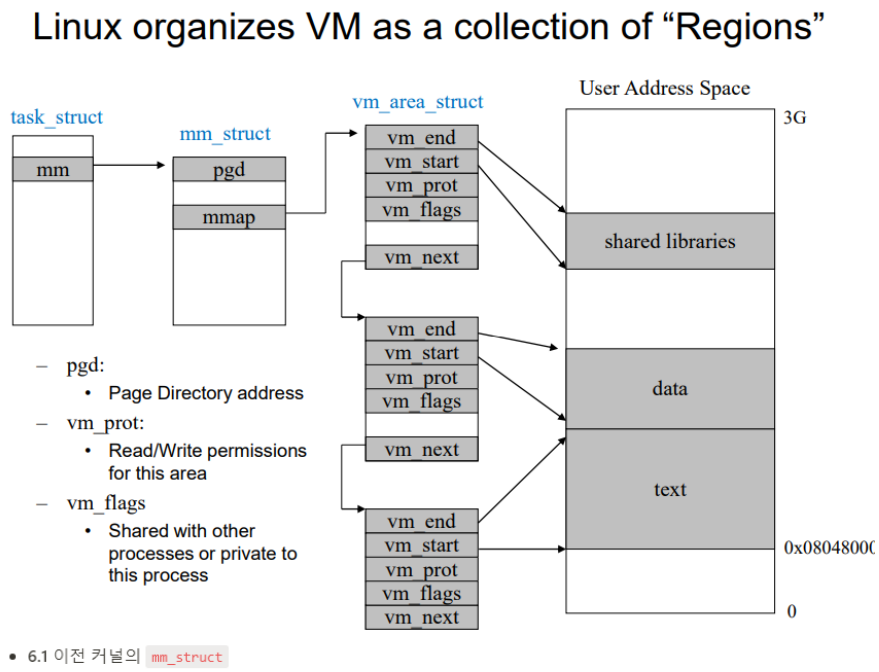
4. 상위 Level Paging 기법이 도입된 커널이 하위 Level Paging을 지원하는 방법

   리눅스 커널 코드에서는, 상위 레벨 페이지 테이블의 값에 하위 레벨 페이지 테이블 정보를 기록하여 상위 레벨 페이지가 하위 레벨 페이지의 존재 유무를 파악하고 지원할 수 있게 한다.

5. 추가적인 2.6.29 버전과 6.2.0 버전의 차이

   task_struct 의 mm 구조 안의 mmap 포인터는 vm_area_struct를 가리키고 있다.

   vm_area_struct는 user address space 상에서 각 region들(.code area, .data area, .stack area, .heap area 등..) 의 시작 주소와 끝 주소 정보를 가지고 있는 구조체이다.



- 6.1 이전 커널의 `mm_struct`

```
struct mm_struct {
    struct {
        struct vm_area_struct *mmap; /* list of VMAs */
        struct rb_root mm_rb; // 레드블랙트리의 루트
        ...
    }
...
```

m_area_struct는 6.1 버전 이전에는 rb 트리로 추적되었고 서로 pointer를 통해 link 되었었다.

```
struct mm_struct {
    struct {
        ...
        struct maple_tree mm_mt;
    }
...
```

하지만 6.2.0 버전부터, m_area_struct의 link 방식이 maple_tree로 변경된 것을 볼 수 있다.

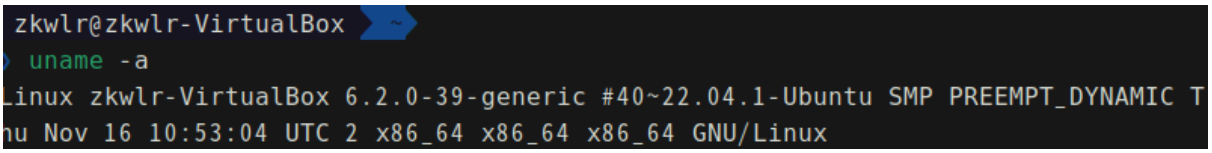
이유는, 기존의 rb tree 추적 방식, 즉 숫자 순서로 노드를 탐색하는 것이 효율적이지 않고, 락에 대한 문제가 존재하기 때문이라고 한다. 이를 대체하기 위해 maple tree가 도입된 것이다. 비교적 매우 최신에 바뀐 커널 구조여서, 몇십년동안 커널이 업그레이드 되어도 더 효율적으로 변화할 수 있는 부분이 있다는 것이 신기했다.

이외에도 효율성 증가, 가독성 증가, 헤더의 구체화/세분화 등 메모리 관리 기법 코드에서 많은 변경점을 찾을 수 있었다. 오랜 기간동안 많은 리눅스 커널 개발자들이 더 빠르고 안정적인 커널을 위해 코드를 수정해 왔는지 알 수 있었다.

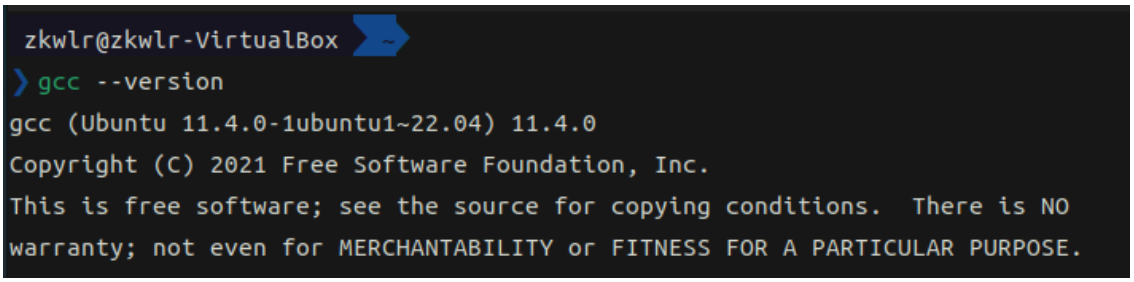
2. 실습 과제 보고서

a. 개발 환경

A. uname -a 결과



B. 컴파일러 정보



C. 운영체제 및 CPU 정보

```
zkwlr@zkwlr-VirtualBox ~
❯ cat /etc/issue
Ubuntu 22.04.3 LTS \n \l

zkwlr@zkwlr-VirtualBox ~
❯ cat /proc/cpuinfo
processor       : 0
vendor_id       : AuthenticAMD
cpu family      : 25
model           : 33
model name      : AMD Ryzen 5 5600 6-Core Processor
stepping        : 2
cpu MHz         : 4294.964
cache size      : 512 KB
physical id     : 0
siblings        : 6
core id         : 0
cpu cores       : 6
apicid          : 0
initial apicid  : 0
fpu             : yes
fpu_exception   : yes
cpuid level     : 16
wp              : yes
flags           : fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge mca cmov pat pse36 clflush m
mx fxsr sse sse2 ht syscall nx mmxext fxsr_opt rdtscp lm constant_tsc rep_good nopl nonstop_tsc cpu
id extd_apicid tsc_known_freq pni pclmulqdq ssse3 cx16 sse4_1 sse4_2 x2apic movbe popcnt aes xsave
avx rdrand hypervisor lahf_lm cmp_legacy cr8_legacy abm sse4a misalignsse 3dnowprefetch vmmcall fsg
sbase bmi1 avx2 bmi2 invpcid rdseed clflushopt arat
bugs            : fxsave_leak sysret_ss_attrs null_seg spectre_v1 spectre_v2
bogomips        : 8589.92
TLB size        : 2560 4K pages
clflush size    : 64
cache_alignment : 64
```

b. 커널 모듈 작성 보고서

A. 커널 모듈 코드

2번 과제에서는 커널 코드를 직접 수정하지 않고 모듈 코드만 수정하였다. 예시로 주어진 모듈 코드에서 아랫 부분들을 수정하였다.

```c
#include <linux/module.h>
#include <linux/init.h>
#include <linux/kernel.h>
#include <linux/proc_fs.h>
#include <linux/seq_file.h>
#include <linux/sched.h>
#include <linux/time.h>
#include <linux/interrupt.h>
#include <linux/timer.h>
#include <linux/jiffies.h>
#include <linux/pgtable.h>
```

먼저 모듈 프로그래밍 및 메모리 추적에 필요한 헤더파일들을 불러왔다.

```c
struct task_info {
    bool is_info_exist; // 정보가 저장됐는지 확인하는 flag, false로 기본 설정
    char name[TASK_COMM_LEN];
    pid_t pid;
    unsigned long long up_time;
    unsigned long long start_time;
    // unsigned long flags;
    // Page grobal directory의 시작 주소(base address)를 저장하는 포인터 변수(포인터면서 그 자체로 우리가 원하는 주
    pgd_t *pgd; // PGD base address
    // Code Area의 start, end 주소를 저장하는 포인터 변수
    unsigned long code_start_vaddr, code_end_vaddr;
    pgd_t *code_start_pgd, *code_end_pgd;
    p4d_t *code_start_p4d, *code_end_p4d;
    pud_t *code_start_pud, *code_end_pud;
    pmd_t *code_start_pmd, *code_end_pmd;
    pte_t *code_start_pte, *code_end_pte;
    // 출력용 val 저장 변수 - 포인터 값은 계속 변하므로 정보를 scan하는 시점의 값을 저장해서 출력해야 한다.
    // 바로 val 함수에 포인터로 넘겨 출력하면 그 시점의 변화한 값이 출력되므로
    unsigned long start_pgd_val, end_pgd_val;
    unsigned long start_pud_val, end_pud_val;
    unsigned long start_pmd_val, end_pmd_val;
    unsigned long start_pte_val, end_pte_val;
    unsigned long code_start_paddr, code_end_paddr;
    // unsigned long code_start_paddr1;
    // Data Area의 가상, 물리 start, end 주소를 저장하는 포인터 변수
    unsigned long data_start_vaddr, data_end_vaddr;
    unsigned long data_start_paddr, data_end_paddr;
    // Heap Area의 가상, 물리 start, end 주소를 저장하는 포인터 변수
    unsigned long heap_start_vaddr, heap_end_vaddr;
    unsigned long heap_start_paddr, heap_end_paddr;
    // Stack Area의 가상, 물리 start, end 주소를 저장하는 포인터 변수
    unsigned long stack_start_vaddr, stack_end_vaddr;
    unsigned long stack_start_paddr, stack_end_paddr;
};
```

그 이후명세서에서 출력하라고 한 정보들을 저장하는 구조체 task_info를 정의하였다. 여러 자료형의 정보들을 저장할 수 있게 만들었다.

```c
struct task_info task_info_list[5];
int task_index = 0;

static struct timer_list my_timer;
```

그 다음으로는 5번의 trace를 기록할 수 있게 5 크기의 task_info를 담는 task_info_list를 만들었다. 또한 가장 최신 정보가 아래로 갈 수 있게 하도록 task_index 전역변수를 미리 선언하였다.

또한 10초마다 정보를 가져오는 함수를 tasklet을 통해 실행하기 위해 my_timer라는 타이머도 생성했다.

```c
void my_tasklet_function(unsigned long data) {

    // printk(KERN_INFO "my_tasklet_function was called.\n");

    //현재 task 정보를 담을 구조체 변수 선언해 정보 저장 후 task_info_list에 저장
    struct task_info latest_task_info;
    struct task_struct *task;
    struct task_struct *latest_task;
    unsigned long long latest_start_time = 0;

    // 가장 최근에 실행된 task를 찾아 latest_task에 저장 후 정보를 찾는다.
    for_each_process(task) {
        // 탐색하는 task의 flag의 PF_KTHREAD field가 0010이면 kthread이므로
        // flags와 0x00200000을 bitwise and 연산해 kthread field를 확인한다.
        // 결과 0이면 flags의 PF_KTHREAD field가 0이므로 결과가 0이 나오므로 if문을 실행한다.(kthread가 아님)
        if (!(task->flags & PF_KTHREAD)) {
            // start_time이 현재 latest_start_time보다 크면
            if ((task->start_time > latest_start_time)) {
                latest_start_time = task->start_time;
                latest_task = task;
            }
        }
    }
```

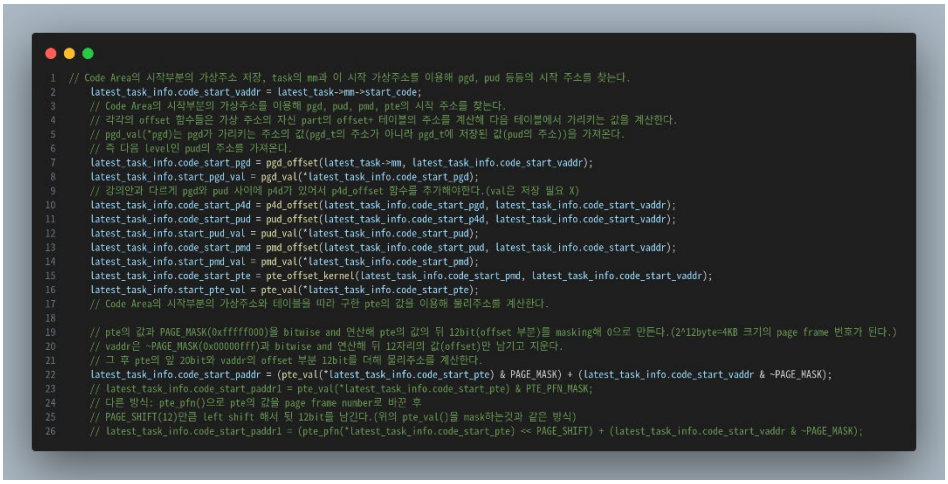
my_tasklet_function은 tasklet 실행시 사용될, task의 메모리 정보를 가져오는 가장 중요한 함수이다. 먼저, task 정보를 담을 task_info 변수를 하나 선언에 그곳에 정보를 모두 저장, 그 후 list에 넣는 식으로 설계하였다. 그 외에도 필요한 여러 변수를 선언하였다.

for_each_process 매크로를 통해 현재 실행중인 task를 모두 탐색하였다. 먼저 task의 flags 항목을 체크해 명세서의 조건대로 kernel thread라면 수집하지 않게 조건을 걸었다. 그 후 kthread가 아니라면 가장 최근에 실행된 task를 찾고 그 task를 latest_task로 설정하게 하였다. 또한 실행된 시각도 저장하였다.

```c
// latest_task의 정보를 latest_task_info에 저장
    latest_task_info.pid = latest_task->pid;
    strcpy(latest_task_info.name, latest_task->comm);
    latest_task_info.up_time = ktime_get_ns() / 1000000000; // 명령 실행된 시각
    latest_task_info.start_time = latest_start_time / 1000000000;
    // latest_task_info.flags = latest_task->flags;

    latest_task_info.pgd = latest_task->mm->pgd; // Page global directory의 시작 주소(포인터) 저장
```

먼저 task의 기본 정보를 저장하였다. 또한 pgd base address를 task의 mm에서 가져와 저장하였다.



```
// Code Area의 시작부분의 가상주소 저장, task의 mm과 이 시작 가상주소를 이용해 pgd, pud 등등의 시작 주소를 찾는다.
    latest_task_info.code_start_vaddr = latest_task->mm->start_code;
    // Code Area의 시작부분의 가상주소를 이용해 pgd, pud, pmd, pte의 시작 주소를 찾는다.
    // 각각의 offset 함수들은 가상 주소의 자신 part의 offset+ 테이블의 주소를 계산해 다음 테이블에서 가리키는 값을 계산한다.
    // pgd_val(*pgd)는 pgd가 가리키는 주소의 값(pgd_t의 주소가 아니라 pgd_t에 저장된 값(pud의 주소))을 가져온다.
    // 즉 다음 level인 pud의 주소를 가져온다.
    latest_task_info.code_start_pgd = pgd_offset(latest_task->mm, latest_task_info.code_start_vaddr);
    latest_task_info.start_pgd_val = pgd_val(*latest_task_info.code_start_pgd);
    // 강의안과 다르게 pgd와 pud 사이에 p4d가 있어서 p4d_offset 함수를 추가해야한다.(val은 저장 필요 X)
    latest_task_info.code_start_p4d = p4d_offset(latest_task_info.code_start_pgd, latest_task_info.code_start_vaddr);
    latest_task_info.code_start_pud = pud_offset(latest_task_info.code_start_p4d, latest_task_info.code_start_vaddr);
    latest_task_info.start_pud_val = pud_val(*latest_task_info.code_start_pud);
    latest_task_info.code_start_pmd = pmd_offset(latest_task_info.code_start_pud, latest_task_info.code_start_vaddr);
    latest_task_info.start_pmd_val = pmd_val(*latest_task_info.code_start_pmd);
    latest_task_info.code_start_pte = pte_offset_kernel(latest_task_info.code_start_pmd, latest_task_info.code_start_vaddr);
    latest_task_info.start_pte_val = pte_val(*latest_task_info.code_start_pte);
    // Code Area의 시작부분의 가상주소와 테이블을 따라 구한 pte의 값을 이용해 물리주소를 계산한다.

    // pte의 값과 PAGE_MASK(0xfffff000)을 bitwise and 연산해 pte의 값의 뒤 12bit(offset 부분)를 masking해 0으로 만든다.(2^12byte=4KB 크기의 page frame 번호가 된다.)
    // vaddr은 ~PAGE_MASK(0x00000fff)과 bitwise and 연산해 뒤 12자리의 값(offset)만 남기고 지운다.
    // 그 후 pte의 앞 20bit와 vaddr의 offset 부분 12bit를 더해 물리주소를 계산한다.
    latest_task_info.code_start_paddr = (pte_val(*latest_task_info.code_start_pte) & PAGE_MASK) + (latest_task_info.code_start_vaddr & ~PAGE_MASK);
    // latest_task_info.code_start_paddr1 = pte_val(*latest_task_info.code_start_pte) & PTE_PFN_MASK;
    // 다른 방식: pte_pfn()으로 pte의 값을 page frame number로 바꾼 후
    // PAGE_SHIFT(12)만큼 left shift 해서 뒷 12bit를 남긴다.(위의 pte_val()을 mask하는것과 같은 방식)
    // latest_task_info.code_start_paddr1 = (pte_pfn(*latest_task_info.code_start_pte) << PAGE_SHIFT) + (latest_task_info.code_start_vaddr & ~PAGE_MASK);
```

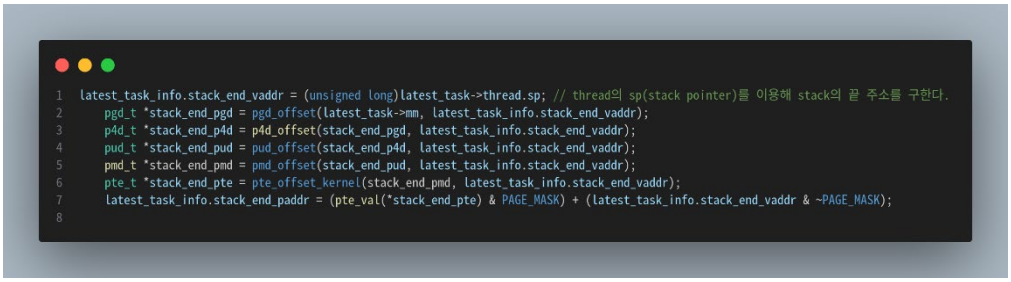
이후 code_area의 여러 메모리 정보를 저장하였다. mm_struct의 start_code 변수를 통해 시작 주소를 찾을 수 있었다. 시작 주소(virtual address)와 pgd base address를 이용해, 선행 보고서에서 알아본 매크로와 함수들을 사용하여 단계적으로 pte의 값까지 찾아내었다. 또한 주소와 값을 찾아내는 즉시 task_info에 저장해 정보입 변경이나 손실이 없게 하였다.

마지막으로 pte의 값과 offset을 각자에 맞게 mask 후 연산하여 시작 부분의 physical address를 계산하여 task_info에 저장하였다. 사전 조사 보고서에 설명했듯 주석 처리된 함수를 통해 여러 방식 모두 같은 주소가 도출됨을 관찰했다.

code_area의 end 부분도 같은 방식으로 task_info에 저장하였다.

```
// Data Area의 가상주소, 물리주소
latest_task_info.data_start_vaddr = latest_task->mm->start_data;
pgd_t *data_start_pgd = pgd_offset(latest_task->mm, latest_task_info.data_start_vaddr);
p4d_t *data_start_p4d = p4d_offset(data_start_pgd, latest_task_info.data_start_vaddr);
pud_t *data_start_pud = pud_offset(data_start_p4d, latest_task_info.data_start_vaddr);
pmd_t *data_start_pmd = pmd_offset(data_start_pud, latest_task_info.data_start_vaddr);
pte_t *data_start_pte = pte_offset_kernel(data_start_pmd, latest_task_info.data_start_vaddr);
latest_task_info.data_start_paddr = (pte_val(*data_start_pte) & PAGE_MASK) + (latest_task_info.data_start_vaddr & ~PAGE_MAS
latest_task_info.data_end_vaddr = latest_task->mm->end_data;
pgd_t *data_end_pgd = pgd_offset(latest_task->mm, latest_task_info.data_end_vaddr);
p4d_t *data_end_p4d = p4d_offset(data_end_pgd, latest_task_info.data_end_vaddr);
pud_t *data_end_pud = pud_offset(data_end_p4d, latest_task_info.data_end_vaddr);
pmd_t *data_end_pmd = pmd_offset(data_end_pud, latest_task_info.data_end_vaddr);
pte_t *data_end_pte = pte_offset_kernel(data_end_pmd, latest_task_info.data_end_vaddr);
latest_task_info.data_end_paddr = (pte_val(*data_end_pte) & PAGE_MASK) + (latest_task_info.data_end_vaddr & ~PAGE_MASK);
// Heap Area의 가상주소, 물리주소
...생략
```

Data Area도 같은 방식으로 pgd base address와 virtual address를 통해 physical address를 계산하였다. code 영역을 제외하고는 변환 중간 과정의 값(pgd_t, p4d_t 등)을 저장할 필요는 없기에, 바로 지역변수를 선언해 중간 계산의 용도로만 쓰이게 하였다. Stack area, Heap area도 같은 식으로 정보를 저장하였다.



```
latest_task_info.stack_end_vaddr = (unsigned long)latest_task->thread.sp; // thread의 sp(stack pointer)를 이용해 stack의 끝 주소를 구한다.
    pgd_t *stack_end_pgd = pgd_offset(latest_task->mm, latest_task_info.stack_end_vaddr);
    p4d_t *stack_end_p4d = p4d_offset(stack_end_pgd, latest_task_info.stack_end_vaddr);
    pud_t *stack_end_pud = pud_offset(stack_end_p4d, latest_task_info.stack_end_vaddr);
    pmd_t *stack_end_pmd = pmd_offset(stack_end_pud, latest_task_info.stack_end_vaddr);
    pte_t *stack_end_pte = pte_offset_kernel(stack_end_pmd, latest_task_info.stack_end_vaddr);
    latest_task_info.stack_end_paddr = (pte_val(*stack_end_pte) & PAGE_MASK) + (latest_task_info.stack_end_vaddr & ~PAGE_MASK);

```

특이 사항으로, 다른 region들의 start 값과 end 값은 mm에 변수로 저장되어 있지만, 유일하게 stack의 end 주소는 존재하지 않았다. 그래서 다른 방식으로 주소를 가져왔다. stack의 끝 주소는 항상 $esp, 즉 스택 포인터 레지스터에 저장되어 있으므로, task의 sp 정보에 접근해 주소를 가져오게 하였다.

```
// latest_start_time이 0이면 유효한 정보가 아니므로 flag를 false로 설정
    if (latest_start_time != 0) {
        latest_task_info.is_info_exist = true; // 정보가 저장됐음을 flag에 기록
    }

    // task_info_list에 최근 정보가 마지막 index로 가게 저장
    if (task_index < 5) { // 배열이 아직 차기 전이면
        task_info_list[task_index] = latest_task_info;
        task_index++;
    }
    else { // 배열이 가득 차 있으면
        for (int i = 0; i < 4; i++) {
            task_info_list[i] = task_info_list[i + 1]; // 앞으로 한칸씩 이동
        }
        task_info_list[4] = latest_task_info; // 마지막 index에 latest_task_info 저장
    }
}
```

모든 정보를 저장한 후, flag를 통해 유효한 정보인지를 확인할 수 있게 하였다. 만약 start_time이 0이라면, 유효한 정보가 아니므로 flag를 false로 설정하고, 아니라면 true로 설정해 정보의 저장 유무를 확인할 수 있게 하였다.

그 아래는 task_info_list에 순차적으로 수집된 정보가 저장될 수 있게 구현하였다. 여기까지가 tasklet을 통해 실행될 my_tasklet_function 함수 구현이다.

```
DECLARE_TASKLET_OLD(my_tasklet, my_tasklet_function);

// timer_list *을 인수로 넘기는 callback 함수를 만들어 타이머가 만료되면 실행될수 있게 함
// tasklet 실행 후 타이머를 10초 뒤로 다시 설정해 callback 함수를 10초 후 다시 실행
static void callback_timer(struct timer_list *timer) {
    tasklet_schedule(&my_tasklet);
    mod_timer(timer, jiffies + msecs_to_jiffies(10000));
}
```

그 후 my_tasklet이라는 tasklet을 정의하고, tasklet_function을 실행할 수 있도록

하였다. 그후 callback 함수를 만들어 타이머를 통해 10초마다 계속해서 timer를 세팅하고 tasklet(메모리 정보를 수집)을 동작할 수 있게 하였다.

```c
static int __init hello_init(void) {
    struct proc_dir_entry *proc_file_entry;
    proc_file_entry = proc_create(PROC_NAME, 0, NULL, &hello_proc_ops);

    // my timer 초기화(시간이 되면 callback_timer 통해 tasklet_schedule 함수 호출되게 my timer 설정)
    timer_setup(&my_timer, callback_timer, 0);
    // 모듈 삽입 0.1초 후에 callback_timer 함수 호출해 tasklet_schedule 함수 호출
    // 그 후 10초마다 callback_timer 함수 안에서 다시 tasklet_schedule 함수 호출
    mod_timer(&my_timer, jiffies + msecs_to_jiffies(100));

    return 0;
}
```

모듈 삽입 후 타이머를 초기화하고 세팅해 설계한 대로 정보를 수집할 수 있게 하였다.

```c
static int hello_seq_show(struct seq_file *s, void *v)
{
    loff_t *spos = (loff_t *) v;
    // 기본 정보 출력 - 학번, 이름, 현재 시스템의 Uptime (초)
    seq_printf(s, "[System Programming Assignment #2]\n");
    seq_printf(s, "ID: 2019147542\n");
    seq_printf(s, "Name: Ha, Donghyun\n");
    seq_printf(s, "Uptime (s): %llu\n", ktime_get_ns() / 1000000000);
    seq_printf(s, "-------------------------------------------------\n");

    // 가장 마지막 기록 5번 출력 (5번 미만으로 정보가 수집된 경우 수집된 횟수만큼만 출력)

    for (int i = 0; i < 5; i++) { // 배열 탐색
        if (task_info_list[i].is_info_exist) { // 정보가 있으면 출력
            seq_printf(s, "[Trace #%d]\n", i);
            seq_printf(s, "Uptime (s): %llu\n", task_info_list[i].up_time);
            seq_printf(s, "Command: %s\n", task_info_list[i].name);
            seq_printf(s, "PID: %d\n", task_info_list[i].pid);
            seq_printf(s, "Start time (s): %llu\n", task_info_list[i].start_time);
            seq_printf(s, "PGD base address: 0x%lx\n", task_info_list[i].pgd); //시작 주소(포인터값) 출력
            // seq_printf(s, "Flags: 0x%lx\n", task_info_list[i].flags);
            // Code Area
            seq_printf(s, "Code Area\n");
            seq_printf(s, "- start (virtual): 0x%lx\n", task_info_list[i].code_start_vaddr); // unsigned long, 16진수로 출력
            seq_printf(s, "- start (PGD): 0x%lx, 0x%lx\n", task_info_list[i].code_start_pgd, task_info_list[i].start_pgd_val); // pgd_t, pgd_val(*pgd)로 pgd가 가리키는 주소의 값(pud의 주소)을 가져온다.
            seq_printf(s, "- start (PUD): 0x%lx, 0x%lx\n", task_info_list[i].code_start_pud, task_info_list[i].start_pud_val);
            seq_printf(s, "- start (PMD): 0x%lx, 0x%lx\n", task_info_list[i].code_start_pmd, task_info_list[i].start_pmd_val);
            seq_printf(s, "- start (PTE): 0x%lx, 0x%lx\n", task_info_list[i].code_start_pte, task_info_list[i].start_pte_val);
            seq_printf(s, "- start (physical): 0x%lx\n", task_info_list[i].code_start_paddr);
            // seq_printf(s, "- start (physical1): 0x%lx\n", task_info_list[i].code_start_paddr1);
            ...생략
            // Data Area
            seq_printf(s, "Data Area\n");
            seq_printf(s, "- start (virtual): 0x%lx\n", task_info_list[i].data_start_vaddr);
            seq_printf(s, "- start (physical): 0x%lx\n", task_info_list[i].data_start_paddr);
            ...생략
            // Heap Area
            seq_printf(s, "Heap Area\n");
            ...생략
            // Stack Area
            seq_printf(s, "Stack Area\n");
            ...생략
        }
    }
    return 0;
}
```

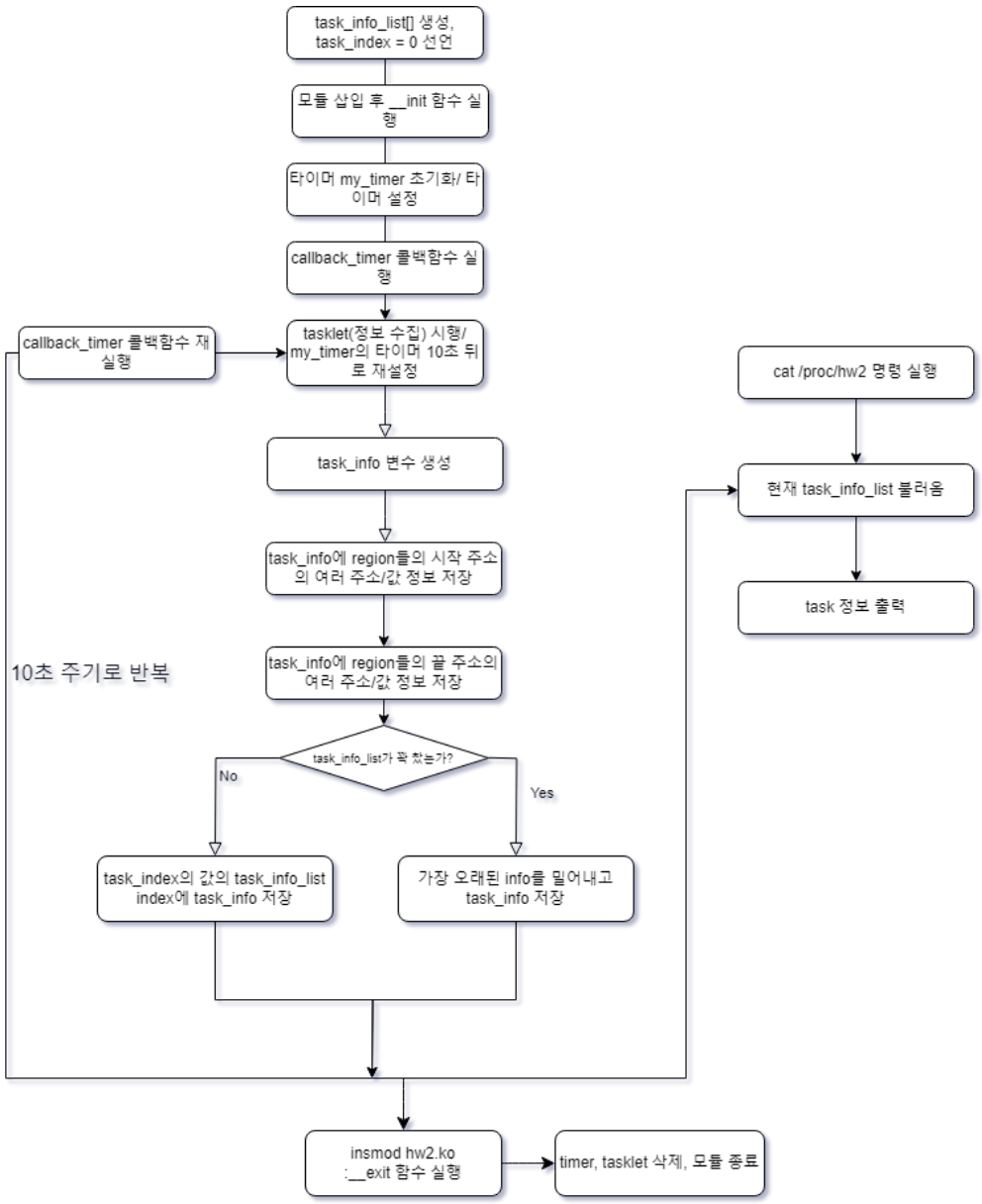
이후, 과제 명세대로 출력되도록 task_info_list에서 정보를 가져와 포맷팅하여 출력하였다.

```c
static void __exit hello_exit(void) {
    remove_proc_entry(PROC_NAME, NULL);
    del_timer_sync(&my_timer);
    tasklet_kill(&my_tasklet);
}
```

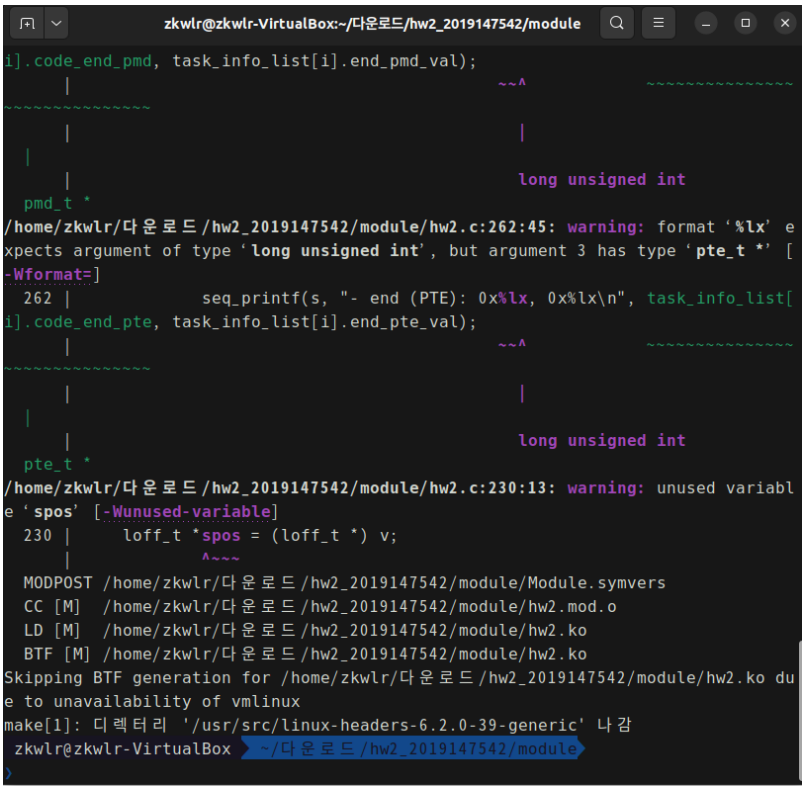
모듈이 삭제되면 동시에 타이머와 tasklet도 삭제되게 하였다.

B. 동작 과정

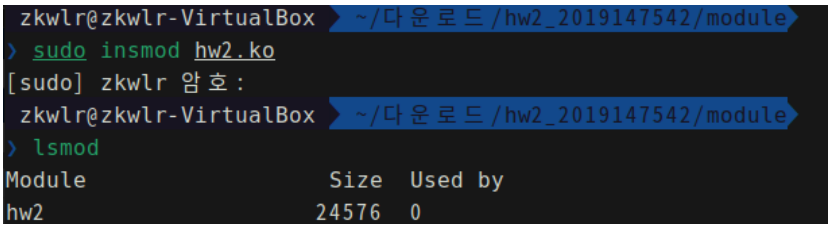
동작 과정은 위에 코드 분석을 통해 설명하였으므로 순서도를 통해 다시 한번 설명하려고 한다.



c. 실습 과제 결과 검증 및 분석



먼저 make를 통해 hw2.ko가 잘 컴파일 되었다.



sudo insmod hw2.ko 명령을 통해 커널에 모듈을 등록 후, lsmod를 통해 hw2 모듈이 등록된 것을 확인할 수 있었다.

```
❯ cat /proc/hw2
[System Programming Assignment #2]
ID: 2019147542
Name: Ha, Donghyun
Uptime (s): 15399
-------------------------------------------------
[Trace #0]
Uptime (s): 15349
Command: zsh
PID: 11457
Start time (s): 15238
PGD base address: 0xffff9d13da23c000
Code Area
- start (virtual): 0x55b38fff6000
- start (PGD): 0xffff9d13da23c558, 0x8e4b8067
- start (PUD): 0xffff9d13ce4b8670, 0x8e4b9067
- start (PMD): 0xffff9d13ce4b93f8, 0xdc30067
- start (PTE): 0xffff9d134dc30fb0, 0xaf843025
- start (physical): 0xaf843000
- end (virtual): 0x55b3900b3386
- end (PGD): 0xffff9d13da23c558, 0x8e4b8067
- end (PUD): 0xffff9d13ce4b8670, 0x8e4b9067
- end (PMD): 0xffff9d13ce4b9400, 0x96b6c067
- end (PTE): 0xffff9d13d6b6c598, 0x8fe01025
- end (physical): 0x8fe01386
Data Area
- start (virtual): 0x55b3900cfae0
- start (physical): 0x8000000076128ae0
- end (virtual): 0x55b3900d6cec
- end (physical): 0x800000000a8aecec
Heap Area
```

이후 cat /proc/hw2 명령을 입력하여 명세서의 예시대로 가장 최근 수집한 정보 5개가 출력됨을 확인하였다. task의 기본 정보, 메모리의 시작/끝 가상주소, 물리 주소 등이 제대로 계산되어 출력됨을 코드에 주석 처리된 여러 함수들로 교차검증 하였다.

```
zkwlr@zkwlr-VirtualBox ~/다운로드/hw2_2019147542/module
❯ sudo rmmod hw2.ko
zkwlr@zkwlr-VirtualBox ~/다운로드/hw2_2019147542/module
❯ lsmod
Module                  Size  Used by
vboxsf                 98304  1
vboxvideo              57344  0
binfmt_misc            24576  1
nls_iso8859_1          16384  1
snd_intel8x0           53248  2
intel_rapl_msr         20480  0
snd_ac97_codec        200704  1 snd_intel8x0
intel_rapl_common      40960  1 intel_rapl_msr
ac97_bus               16384  1 snd_ac97_codec
snd_pcm               192512  2 snd_intel8x0,snd_ac97_codec
snd_seq_midi           20480  0
crct10dif_pclmul       16384  1
snd_seq_midi_event     16384  1 snd_seq_midi
snd_rawmidi            53248  1 snd_seq_midi
polyval_clmulni        16384  0
polyval_generic        16384  1 polyval_clmulni
snd_seq                94208  2 snd_seq_midi,snd_seq_midi_event
```

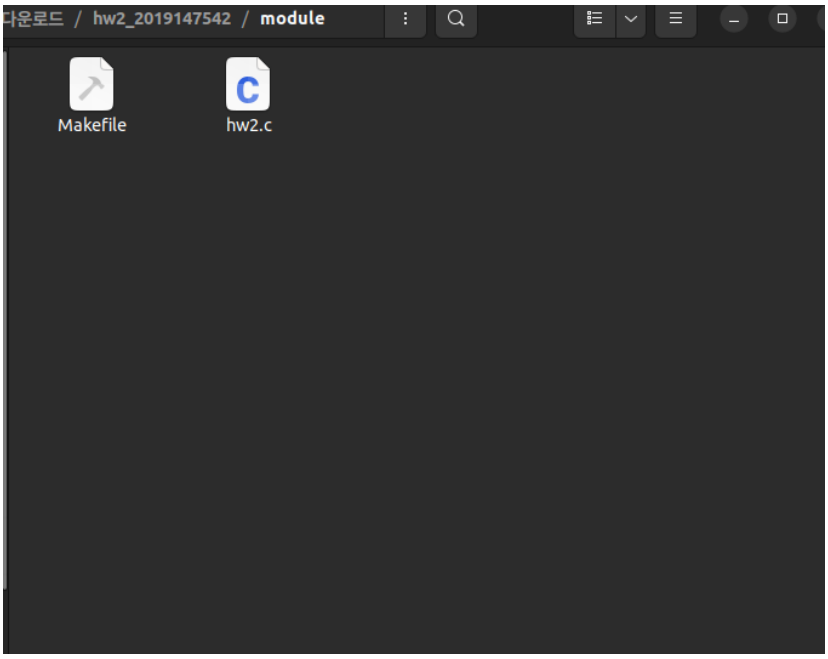
sudo rmmod hw2를 통해 모듈을 제거하였고, 이후 lsmod 입력 결과 hw1 모듈이 삭

제됨을 확인하였다.

```
autofs4                57344  2
hid_generic            16384  0
usbhid                 73728  0
hid                   176128  2 usbhid,hid_generic
crc32_pclmul           16384  0
psmouse               212992  0
ahci                   49152  2
libahci                57344  1 ahci
i2c_piix4              28672  0
e1000                 180224  0
pata_acpi              16384  0
video                  73728  0
wmi                    40960  1 video
 zkwlr@zkwlr-VirtualBox  ~/다운로드/hw2_2019147542/module
❯ cat /proc/hw2
cat: /proc/hw2: 그런 파일이나 디렉터리가 없습니다
 ✘ zkwlr@zkwlr-VirtualBox  ~/다운로드/hw2_2019147542/module
❯
```

제거 후 cat /proc/hw2 명령을 사용하면 작동하지 않는 것을 확인하였다.

```
 ✘ zkwlr@zkwlr-VirtualBox  ~/다운로드/hw2_2019147542/module
❯ make clean
rm -rf *.ko *.mod *.mod.* .*.cmd *.o *.symvers *.order
 zkwlr@zkwlr-VirtualBox  ~/다운로드/hw2_2019147542/module
❯
```



make clean 명령 후 Makefile과 모듈 프로그래밍 구현물 등을 제외한 나머지 파일이 삭제되었다.

d. 과제를 수행하면서 있었던 문제 및 해결방법

이번 과제에서는 커널 코드를 직접 수정하지 않았기 때문에 커널 빌드 시간에 대한 부담은 없었다. 하지만 시스템에서 복잡한 부분인 메모리 정보를 건드리고, 심지어 직접 내 코드를 가상 주소를 물리 주소로 변환해야 하였기 때문에 상당히 까다로웠던 것 같다. 먼저 모듈 pgtable.h를 불러올 때, linux/pgtable.h가 아니라 asm/pgtable.h에서 불러오려고 해서 메모리 정보를 제대로 가져오지 못하는 문제가 있었다. linux/pgtable.h를 불러오면 알아서 내 아키텍처에 맞는 pgtable 헤더를 가져온다는 것을 깨닫고 문제를 해결하였다.

또한 tasklet과 timer 등 리눅스 커널 디자인용 도구들을 처음 사용하게 되었는데, 처음 사용하다 보니 익숙하지 않아 구현하는데 시간이 좀 걸렸다. 특히 callback 함수와 timer를 통해 함수 실행에 delay를 거는 방식은 이번 과제를 통해 처음 알게 되었고 상당히 간결하고 직관적이지만 효율적인 구현 방식이라는 것을 깨닫게 되었다.

pte_val 값과 virtual address의 offset을 통해 마지막 물리 주소로 변환하는 코드를 만드는 것에도 시간이 소요되었다. 미리 정의된 여러 매크로 MASK, SHIFT 값들을 커널 코드를 따라가며 찾아보고, bit masking/ bitwise and or 연산을 통해 물리 주소를 계산한다는 것을 알게 되었다.

또한 정보를 출력할때, pte_val(*pte) 등으로 value 값을 포인터로 직접 넘겼더니, 그 당시의 value가 출력되는 게 아니라 출력 당시의 주소의 value가 출력되는 오류가 발생했다. 포인터 값은 정보를 로그하는 당시에만 의미가 있는 주소 값이므로 출력 당시의 포인터가 가리키는 값은 쓰레기 값이거나 의미없는 값일 수 있다. 그래서 포인터로 넘겨 출력하지 않고, task_info에 각 value를 저장할 수 있게 하여 정보를 수집하는 즉시 포인터가 가리키는 값을 저장할 수 있게 하였다. 이를 통해 정상적인 value 값을 출력할 수 있었다.

마지막으로 rmmod 후 insmod를 바로 하는 경우 가끔 kernel crash가 발생하여 가상환경이 먹통이 되는 경우가 발생했다. insmod를 하는 동시에 tasklet_schedule을 통해 첫번째 trace를 수집한다. 그 순간 task가 충돌되면서 kernel crash가 발생하는 것 같다고 추측하였다. 그래서, insmod 직후 tasklet을 실행하지 않고 초기 타이머를 0.1초로 세팅해 insmod 후 0.1초 후부터 10초 간격으로 tasklet을 실행하여 정보를 수집하도록 하였다. 이렇게 수정하니 kernel crash가 전혀 작동하지 않고 정상적으로 작동함을 확인할 수 있었다.

e. 참고한 문헌/사이트

https://elixir.bootlin.com/linux/v2.6.29/source/arch/x86/include/asm/page.h#L119
bootlin – Elixir Cross linux Referencer – v 2.6.29

https://elixir.bootlin.com/linux/v6.2/source/arch/riscv/include/asm/pgtable-64.h#L368 bootlin – Elixir Cross linux Referencer – v 6.2

시스템 프로그래밍 L03-interrruptsexception 강의노트

시스템 프로그래밍 L04-timingmeasurements 강의노트

시스템 프로그래밍 L08-processscheduling 강의노트

시스템 프로그래밍 L09-memorymanagement 강의노트

시스템 프로그래밍 L10-processaddressspace 강의노트

https://dataonair.or.kr/db-tech-reference/d-lounge/technical-data/?mod=document&uid=236882 지연 가능 함수, 커널 태스크릿 및 작업 큐

https://velog.io/@jinh2352/series/Linux [Linux] Task, Interrupt, Module programming

https://damduc.tistory.com/160 리눅스 커널 모듈 프로그래밍 – 2.4

https://wikidocs.net/196806 테스크 스케줄링

https://badayak.com/entry/C%EC%96%B8%EC%96%B4-%ED%8F%AC%EC%9D%B8%ED%84%B0-%ED%95%A8%EC%88%98 C언어의 포인터 함수와 콜백함수

https://hyeyoo.com/90 Linux - 시간과 타이머

https://austindhkim.tistory.com/38 [리눅스커널] 태스크 디스크립터(struct task_struct) 세부 필드 분석

https://brewagebear.github.io/linux-kernel-internal-2/ [Kernel] 리눅스 메모리 관리 훑어보기

https://people.kernel.org/linusw/arm32-page-tables ARM32 Page Tables

https://jeongzero.oopy.io/359eaa11-b6e6-466c-a066-9ae582c886d4 [LInux Kernel] 메모리 관리 : 가상 메모리

https://www.cnblogs.com/ronnydm/p/5756158.html【深入理解Linux内核架构】3.3 页表

hw2에서 다시 참조한 hw1 참고문헌

https://velog.io/@mythos/Linux-Tutorial-9-%EC%BB%A4%EB%84%90-%EB%AA%A8%EB%93%88-%ED%94%84%EB%A1%9C%EA%B7%B8%EB%9E%98%EB%B0%8D 커널 모듈 프로그래밍

https://jhnyang.tistory.com/299 C 전처리기 지시어

https://codingcoding.tistory.com/193 리눅스 proc 파일 시스템 이해하기

https://waterfogsw.tistory.com/16 Linux kernel Structure

https://blog.naver.com/jeix2/80007580165 task_struct 구조체

https://darkengineer.tistory.com/119 테스크 디스크립터 자료구조

https://poplinux.tistory.com/105 extern : 개발한 함수를 외부에 공개하는법

https://blog.naver.com/cre8tor/90193630398 EXPORT_SYMBOL

https://velog.io/@mythos/Linux-Tutorial-6-%EB%8D%B0%EC%9D%B4%ED%84%B0-%ED%83%80%EC%9E%85 리눅스 커널의 다양한 데이터 타입