System Programming 2nd Assignment

목차

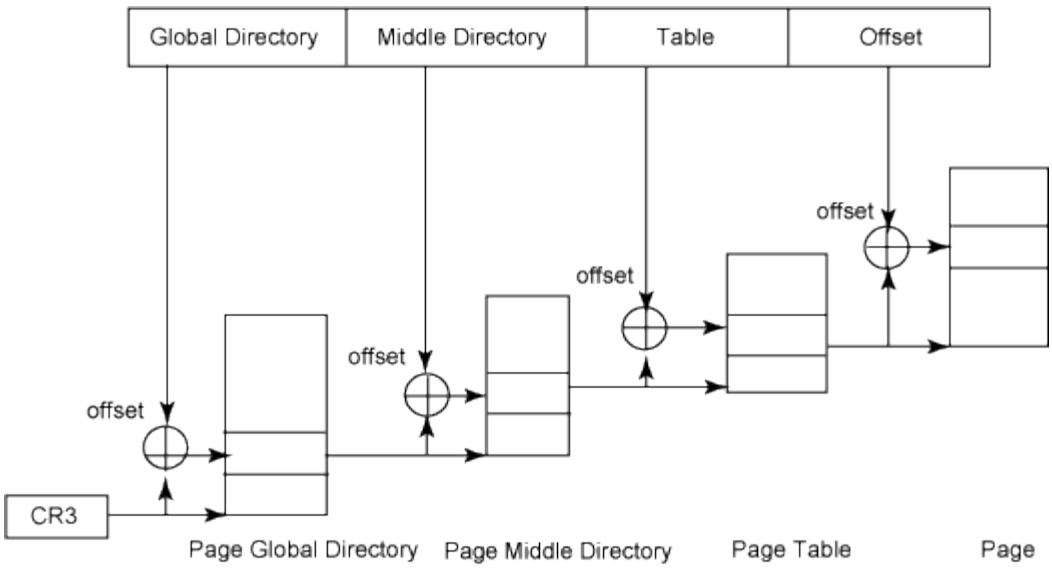
1. 사전 조사 보고서
   가. 3 Level Paging
   나. 4 Level Paging
   다. 3 Level Paging, 4 Level Paging의 차이점
   라. User application에서의 Memory Allocation
   마. Tasklet
2. 실습 과제 수행 보고서
   가. 프로그램 개관
   나. phys_to_virt
   다. 소결

2014147550 컴퓨터과학과
강효림

1. 사전 조사 보고서

가. 3 Level Paging

1) 개요



위 그림과 같이 pgd, pmd, pte의 세 페이지 테이블을 이용하여 virtual address를 physical address로 변환한다.

2) follow_page()

```
struct page *
follow_page(struct mm_struct *mm, unsigned long address, int write)
{
    pgd_t *pgd;
    pmd_t *pmd;
    pte_t *ptep, pte;
    unsigned long pfn;
    struct page *page;

    page = follow_huge_addr(mm, address, write);
    if (! IS_ERR(page))
        return page;

    pgd = pgd_offset(mm, address);
    if (pgd_none(*pgd) || unlikely(pgd_bad(*pgd)))
        goto out;

    pmd = pmd_offset(pgd, address);
    if (pmd_none(*pmd))
        goto out;
    if (pmd_huge(*pmd))
        return follow_huge_pmd(mm, address, pmd, write);
    if (unlikely(pmd_bad(*pmd)))
        goto out;

    ptep = pte_offset_map(pmd, address);
    if (!ptep)
        goto out;

    pte = *ptep;
```

address로 주어진 가상 주소를, 여러 페이지 테이블을 거치면서 물리주소로 변환하는 함수이다. 여기서 쓰이는 pgd_offset()과 같은 커널 매크로/함수들을 중심으로 가상 주소->물리주소에 쓰이는 커널 코드들을 분석한다.

3) mm_struct->pgd

```
struct mm_struct {
    struct vm_area_struct * mmap;       /* list of VMAs */
    struct rb_root mm_rb;
    struct vm_area_struct * mmap_cache; /* last find_vma result */
    unsigned long (*get_unmapped_area) (struct file *filp,
                unsigned long addr, unsigned long len,
                unsigned long pgoff, unsigned long flags);
    void (*unmap_area) (struct vm_area_struct *area);
    unsigned long mmap_base;        /* base of mmap area */
    unsigned long free_area_cache;      /* first hole */
    pgd_t * pgd;
```

pgd_t* type이며, 당해 프로세스의 pgd의 주소값을 저장하고 있다. pgd_t는 구조체로, 그 원소로 unsigned long 형태의 pgd를 갖는데, 이는 그 주소값에 저장되어있는 pgd entry의 값, 즉 pmd의 주소값을 나타내게 된다.

4) pgd_offset()

#define pgd_offset(mm, addr) ((mm)->pgd + pgd_index(addr)

pgd의 주소값에 pgd_index(addr)를 통하여 pgd offset을 더해주어 pgd_offset의 주소를 반환하는 커널 매크로이다.

5) pgd_index()

#define pgd_index(addr) ((addr) >> PGDIR_SHIFT)

addr로 주어진 virtual address에서 pgd의 offset을 추출하기 위해 PGDIR_SHIFT만큼 우향 shift 연산을 수행한다.

6) pmd_offset()

```
/* Find an entry in the second-level page table.. */
#define pmd_offset(dir, address) ((pmd_t *) pgd_page(*(dir)) + \
            pmd_index(address))
```

위에서 살핀 pgd_offset과 거의 비슷한 역할을 한다. dir을 pmd_t의 pointer 변수로 캐스팅 한 뒤 index를 더하여 반환하는 것을 볼 수 있다.

7) pte_offset_map()

```
#define pte_offset_map(dir,addr)        (pmd_page_kernel(*(dir)) + __pte_index(addr))
```

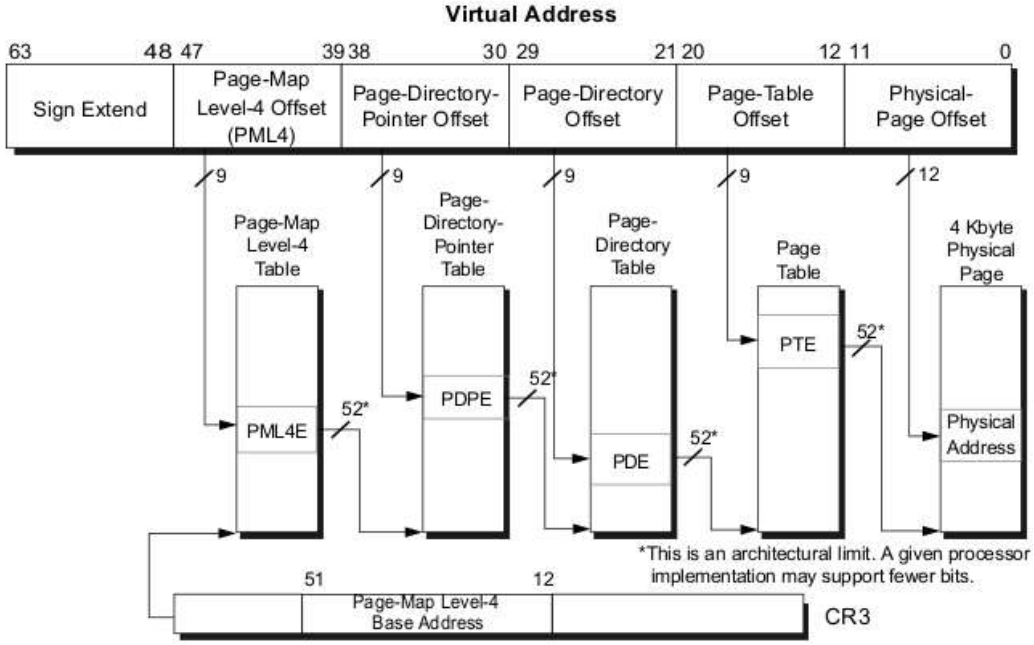
위의 pmd_offset()과 마찬가지 역할을 수행한다. 다만 연산은 pmd_t의 value에 대하여 이루어진다.

8) pte = *ptep

ptep은 pte_offset_map()의 반환값이며, 페이지의 주소값이다. 따라서 *연산을 통하여 그 주소에 있는 값을 참조하면 물리주소 pte가 얻어지게 된다.

나. 4 Level Paging

1) 개요



3 Level Paging과 매우 유사한 구조로 virtual address->physical address의 변환을 수행하며, 커널 매크로/함수도 거의 같다. 그러나 위와 같이 하나의 페이지 테이블이 더 추가되는데, 이는 pud이다. pud와 관련된 함수를 중심으로 kernel code를 분석한다.

2) follow_page()

```
struct page *follow_page_mask(struct vm_area_struct *vma,
                              unsigned long address, unsigned int foll_flags,
                              unsigned int *page_mask);

static inline struct page *follow_page(struct vm_area_struct *vma,
                unsigned long address, unsigned int foll_flags)
{
        unsigned int unused_page_mask;
        return follow_page_mask(vma, address, foll_flags, &unused_page_mask);
}
```

follow_page()는 follow_page_mask()를 그 내부에서 사용하므로, follow_page_mask()를 살핀다.

3) follow_page_mask()

```
pud = pud_offset(pgd, address);
if (pud_none(*pud))
        return no_page_table(vma, flags);
if (pud_huge(*pud) && vma->vm_flags & VM_HUGETLB) {
        page = follow_huge_pud(mm, address, pud, flags);
        if (page)
                return page;
        return no_page_table(vma, flags);
}
if (unlikely(pud_bad(*pud)))
        return no_page_table(vma, flags);
```

follow_page_mask()에서 pud를 다루는 부분이다. pgd에는 pgd_offset() 매크로로 얻어진 값이 존재한다. 따라서 pud_offset()함수를 살핀다.

4) pud_offset()

```
static inline pud_t *pud_offset(pgd_t *pgd, unsigned long addr)
{
        return (pud_t *)pgd_page_vaddr(*pgd) + pud_index(addr);
}
```

위의 3단계 페이징에서 살핀 xxx_offset() 매크로들과 유사하게 작동하고 있는 것을 확인할 수 있다.

5) * 참 고 자 료 ->pgd_t, pud_t 등

```
typedef struct { pteval_t pte; } pte_t;
#define pte_val(x)      ((x).pte)
#define __pte(x)        ((pte_t) { (x) } )

#if CONFIG_PGTABLE_LEVELS > 2
typedef struct { pmdval_t pmd; } pmd_t;
#define pmd_val(x)      ((x).pmd)
#define __pmd(x)        ((pmd_t) { (x) } )
#endif

#if CONFIG_PGTABLE_LEVELS > 3
typedef struct { pudval_t pud; } pud_t;
#define pud_val(x)      ((x).pud)
#define __pud(x)        ((pud_t) { (x) } )
#endif

typedef struct { pgdval_t pgd; } pgd_t;
#define pgd_val(x)      ((x).pgd)
#define __pgd(x)        ((pgd_t) { (x) } )
```

페이지 테이블의 엔트리들을 저장하는 구조체이다. #if 절로 page level에 따라 구조체들을 선별적으로 define하는 것을 볼 수 있다. 앞서 여러번 살피었듯, 구조체의 주소값은 그 전 레벨의 구조체 엔트리의 'value'에 virtual address의 offset을 더하여 얻어진다.

다. 3 Level Paging, 4 Level Paging의 차이점

1) 차이점

Linux kernel 2.6.10버전까지 사용되어진 3 Level Paging과 달리, 4 Level Paging은 새로운 page table인 pud를 사용한다. 또한 32bit를 linear address로 사용하던 3 Level paging과 달리, 48bit를 사용한다.

2) Compatibility

가) 2 level paging의 경우

pud와 pmd의 entry number를 1로 하고, 이를 적당한(비어있는) pgd의 entry에 연결시키며, 페이징 과정에서 pud와 pmd의 참여를 배제하고 진행한다. (0 bit를 담고 있다 선언)

나) 3 level paging의 경우

위와 비슷하나, pmd는 남겨두고, pud만 eliminate한다.

라. User application에서의 Memory Allocation

1) Kernel Memory Allocation과의 차이점

User mode의 program들은 non-urgent하며 trustworthy하지 않기 때문에, memory allocation이 요구되었을 때 즉시 page frame을 할당받지 않는다. 다만 **virtual address가 확장되면서, 그 확장된 부분만큼의 주소를 사용할 권리를 얻음**에 그친다.

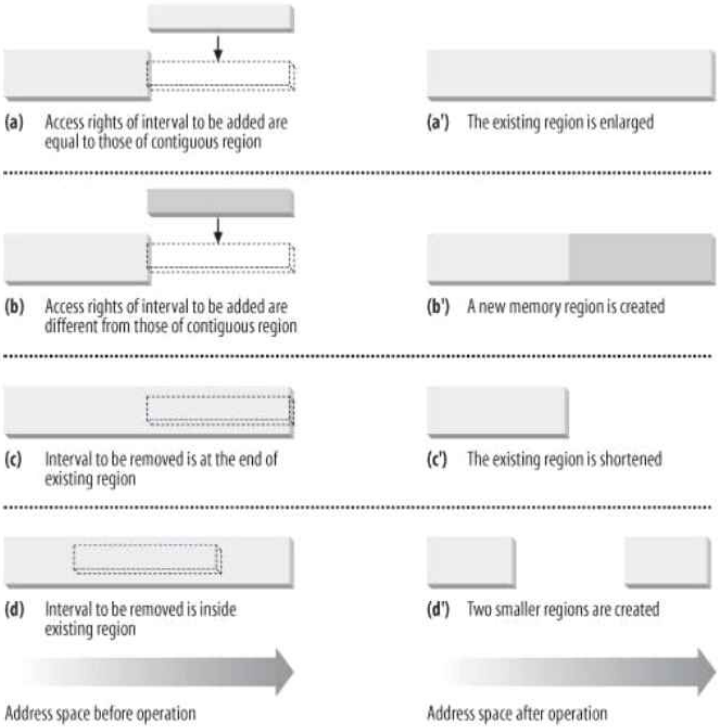
2) local variable의 경우

stack memory region이 expand 된다.

3) dynamic allocation

heap memory region이 expand되며, free 되면 줄어든다.

4) 기타의 경우 -> 도식(수업 ppt 中)



마. Tasklet

1) 자료 구조

```
struct tasklet_struct
{
        struct tasklet_struct *next;
        unsigned long state;
        atomic_t count;
        void (*func)(unsigned long);
        unsigned long data;
};
```

next 포인터로 다음 tasklet_struct에 접근할 수 있으며, 함수 포인터인 func에 기능을 등록하여 사용한다.

2) 특징

가) 한번만 실행됨이 보장됨

나) strictly serialized

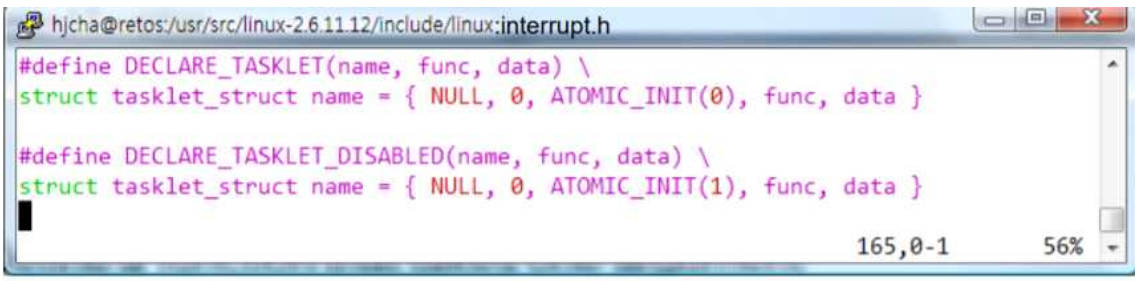
다) 같은 type의 tasklet은 多 CPU환경에서 concurrent한 실행이 허용되지 않음

라) interrupt context에서 실행되기에 sleep, 선점 등은 허용되지 않음

3) 작동 메커니즘

tasklet은 softirq의 한 부분으로 동작하며, DECLARE_TASKLET으로 선언되고, tasklet_schedule()로 커널에 스케쥴링된다. 이를 이하 과제에서 활용하였다.

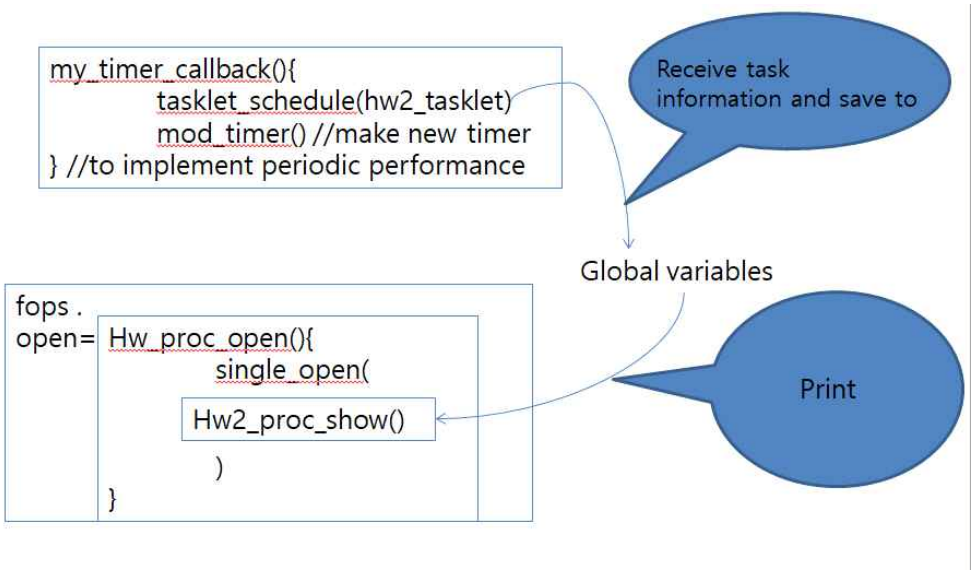
4) Declaring Tasklet 예시 - 수업 ppt



2. 실습 과제 수행 보고서

가. 프로그램 개관

1) 도식



2) 설명

가) 타이머 부분

```
static struct timer_list my_timer;

void my_timer_callback (unsigned long data)
{
  call_cnt++;
  tasklet_schedule(&hw2_tasklet);
  mod_timer(&my_timer, jiffies + msecs_to_jiffies(period*1000) );
}

int init_module( void )
{
  int ret;
  proc_create("hw2", 0, NULL, &fops);
  setup_timer( &my_timer, my_timer_callback, 0 );
  ret = mod_timer( &my_timer, jiffies + msecs_to_jiffies(period*1000) );
  if (ret) printk("Error in mod_timer\n");

  return 0;
}
```

setup_timer로 timer을 setup하고, mod_timer로 타이머의 expire time을 정한다, 그 expire time이 되면 my_timer_callback 함수가 실행되는데, 그 함수에서 다시 mod_timer를 call하여 다시 같은 주기로 expire time을 정해준다. 이로써 periodic performace를 구현한다.

나) tasklet 부분

```
DECLARE_TASKLET(hw2_tasklet, hw2_tasklet_function, (unsigned long) &hw2_tasklet_data);
```

```
void hw2_tasklet_function(unsigned long data){
    int constant = 1000/HZ;
    system_uptime = (jiffies-INITIAL_JIFFIES)*constant;
    struct task_struct *task;
    struct task_struct *target = NULL;
    struct mm_struct *mm;
    struct vm_area_struct *mmap;
```

DECLARE_TASKLET으로 tasklet을 선언하면서, tasklet의 function으로는 위와 같은 hw2_tasklet_function을 지정해 준다. 한편 위 timer code에서 볼 수 있듯 timer callback 함수에서 이 tasklet이 schedule되므로 이 hw2_tasklet_function은 주기적으로 실행된다.

다) hw2_tasklet_function

(1) select random process

```
for_each_process(task){
    if (task->mm != NULL){
        target = task;
        if (cnt > randNum){
            break;
        }
        cnt++;
    }
}
```

이때 randNum은 linux/random.h의 get_random_bytes()로 얻은 수를 100으로 나눈 나머지이다.

(2) code, data, heap, stack virtual address

```
unsigned long total_vm;         /* Total pages mapped */
unsigned long locked_vm;        /* Pages that have PG_mlocked set */
unsigned long pinned_vm;        /* Refcount permanently increased */
unsigned long shared_vm;        /* Shared pages (files) */
unsigned long exec_vm;          /* VM_EXEC & ~VM_WRITE */
unsigned long stack_vm;         /* VM_GROWSUP/DOWN */
unsigned long def_flags;
unsigned long nr_ptes;          /* Page table pages */
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;
```

mm_struct의 위와 같은 정보를 이용하여 code, data, heap 등의 주소값들을 얻어온다. 이 때, bss는 나와있지 않으므로, vm_area_struct 중 그 start address가 end_data와 start_brk 사이에 존재하는 것을 bss area로 간주한다.

```
do {
    if (mmap->vm_start < start_heap && mmap->vm_start > end_data){
        bss_flag = 1;
        start_bss = mmap->vm_start;
        end_bss = mmap->vm_end;
    } else if (mmap->vm_start > end_heap){
        start_shared = mmap->vm_start;
        end_shared = start_shared + (PAGE_SIZE * shared_size);
        break;
    }
} while(mmap=mmap->vm_next);
if (!bss_flag){
    start_bss = 0;
    end_bss = 0;
}
```

(3) paging info

```
target_vaddress = start_code;
_pgd_offset = pgd_offset(target->mm, target_vaddress);
_pgd_base = target->mm->pgd;
_pgd_val = pgd_val(*_pgd_offset);
if (_pgd_val&_PAGE_PRESENT){
    _pgd_present = 1;
} else {
    _pgd_present = 0;
}
if (_pgd_val&_PAGE_PCD){
    strcpy(_pgd_cache_disable, "true");
} else {
    strcpy(_pgd_cache_disable, "false");
}
if (_pgd_val&_PAGE_ACCESSED){
    _pgd_accessed = 1;
} else {
    _pgd_accessed = 0;
}
if (_pgd_val&_PAGE_USER){
    strcpy(_pgd_user, "user");
} else {
    strcpy(_pgd_user, "supervisor");
}
if (_pgd_val&_PAGE_PWT){
    strcpy(_pgd_write_through, "write-through");
} else {
    strcpy(_pgd_write_through, "write-back");
}
if (_pgd_val&_PAGE_RW){
    strcpy(_pgd_read_write, "read-write");
} else {
    strcpy(_pgd_read_write, "read-only");
}
_pud_offset = pud_offset(_pgd_offset, target_vaddress);
_pud_val = pud_val(*_pud_offset);
_pmd_offset = pmd_offset(_pud_offset, target_vaddress);
_pmd_val = pmd_val(*_pmd_offset);
_pte_offset = pte_offset_kernel(_pmd_offset, target_vaddress);
_pte_val = pte_val(*_pte_offset);
_real_phy = (_pte_val / PAGE_SIZE)*PAGE_SIZE;
_vad = phys_to_virt(_real_phy);
```

사전 보고서에서 살핀 커널 메크로/함수(xxx_offset()) 등을 이용하여 paging information을 추출하여 이를 global variable에 저장한다. 추가적으로는 flag에 대하여 AND mask 연산을 수행하여 additional paging information을 파악한다.

(4) phys_to_virt()

마지막은 이런 식으로 구한 physical address를 phys_to_virt()로 virtual address로 바꾼 후 출력해 본다.

나. phys_to_virt()

1) 분석

```
static inline void *phys_to_virt(phys_addr_t address)
{
        return __va(address);
}

#define __va(x)                 ((void *)((unsigned long)(x)+PAGE_OFFSET))

#define __PAGE_OFFSET           _AC(0xffff880000000000, UL)
```

커널 코드를 보면, phys_to_virt() 메크로는 인자로 들어온 physical address에 PAGE_OFFSET을 더해주는 코드임을 알 수 있다.

2) 실제 virtual address와의 차이, 및 그 이유

실행해 보면 실제 virtual address와는 전혀 다른 값이 출력되는 것을 볼 수 있다. (뒤에 결과 화면 첨부) 이는 앞서 살피었듯 phys_to_virt() 메크로는 현 physical address에 PAGE_OFFSET을 더해주는, 즉 (~896MB) kernel memory와 같이 direct mapping을 전제한 함수이기 때문이다. 그러나 랜덤으로 select된 process는 user process이고(task->mm != NULL), process memory는 direct mapping이 되지 않으므로 이 매크로로 physical address를 역산해서 virtual address를 구할 수는 없는 것이다.

다. 소결

1) uname -a

```
aya@aya-VirtualBox:~$ uname -a
Linux aya-VirtualBox 4.4.21 #1 SMP Mon Nov 26 17:31:05 KST 2018 x86_64 x86_64 x86_64 GNU/Linux
aya@aya-VirtualBox:~$
```

2) 결과화면

가) 실행 명령

```
root@aya-VirtualBox:~/Desktop/hw2# insmod hw2.ko period=3
root@aya-VirtualBox:~/Desktop/hw2#
```

나) 결과 화면

```
Student ID: 2014147550  Name: Kang Hyo Lim
Virtual Memory Address Information
Process (indicator-messa:1381)
Last update time 33431728 ms
********************************************************************
0x00400000 - 0x0041845c : Code Area, 25 page(s)
0x00619140 - 0x0061b8e8 : Data Area, 3 page(s)
None : BSS Area, 0 page(s)
0x00a28000 - 0x00a8b000 : Heap Area, 99 page(s)
0x7f1e0c000000 - 0x7f1e0f0d9000 : Shared Libraries Area, 12505 page(s)
0x7fff06d752f0 - 0x7fff06d972f0 : Stack Area, 34 page(s)
********************************************************************
1 Level Paging: Page Directory Entry Information
********************************************************************
PGD     Base Address            : 0xffff8800d7fa9000
code    PGD Address             : 0xffff8800d7fa9000
        PGD Value               : 0xd5df6067
        +PFN Address            : 0x000d5df6
        +Page Size              : 4KB
        +Accessed Bit           : 1
        +Cache Disable Bit      : false
        +Page Write-Through     : write-back
        +User/Supervisor Bit    : user
        +Read/Write Bit         : read-write
        +Page Present Bit       : 1
********************************************************************
2 Level Paging: Page Upper Directory Entry Information
********************************************************************
code    PUD Address             : 0xffff8800d5df6000
        PUD Value               : 0xd610c067
        +PFN Address            : 0x000d610c
********************************************************************
3 Level Paging: Page Middle Directory Entry Information
********************************************************************
code    PMD Address             : 0xffff8800d610c010
        PMD Value               : 0xdaece067
        +PFN Address            : 0x000daece
********************************************************************
4 Level Paging: Page Table Entry Information
********************************************************************
code    PTE Address             : 0xffff8800daece000
        PTE Value               : 0xc86dc025
        +Page Base Address      : 0x000c86dc
        +Dirty Bit              : 0
        +Accessed Bit           : 1
        +Cache Disable Bit      : false
        +Page Write-Through     : write-back
        +User/Supervisor Bit    : user
        +Read/Write Bit         : read-only
        +Page Present Bit       : 1
********************************************************************
Start of Physical Address       : 0xc86dc000
********************************************************************
Start of Virtual Address        : 0xffff8800c86dc000
```

앞서 언급하였듯 역산하여 구한 virtual address와 실제 virtual address(위에서는 Code Area의 start address)가 크게 다름을 알 수 있다. 또한 꽤 많은 process에서 bss area가 발견되지 않는 것을 볼 수 있었다.(다만 bss가 나타나는 프로세스도 많음)

다) Reference

http://jake.dothome.co.kr

http://pobimoon.tistory.com

수업 자료

리눅스 커널 내부구조 (백승재, 최종무 著)