시스템 프로그래밍(CSI3107-01)

Assignment 2

Linux Kernel Memory Management & Interrupt Handler 보고서

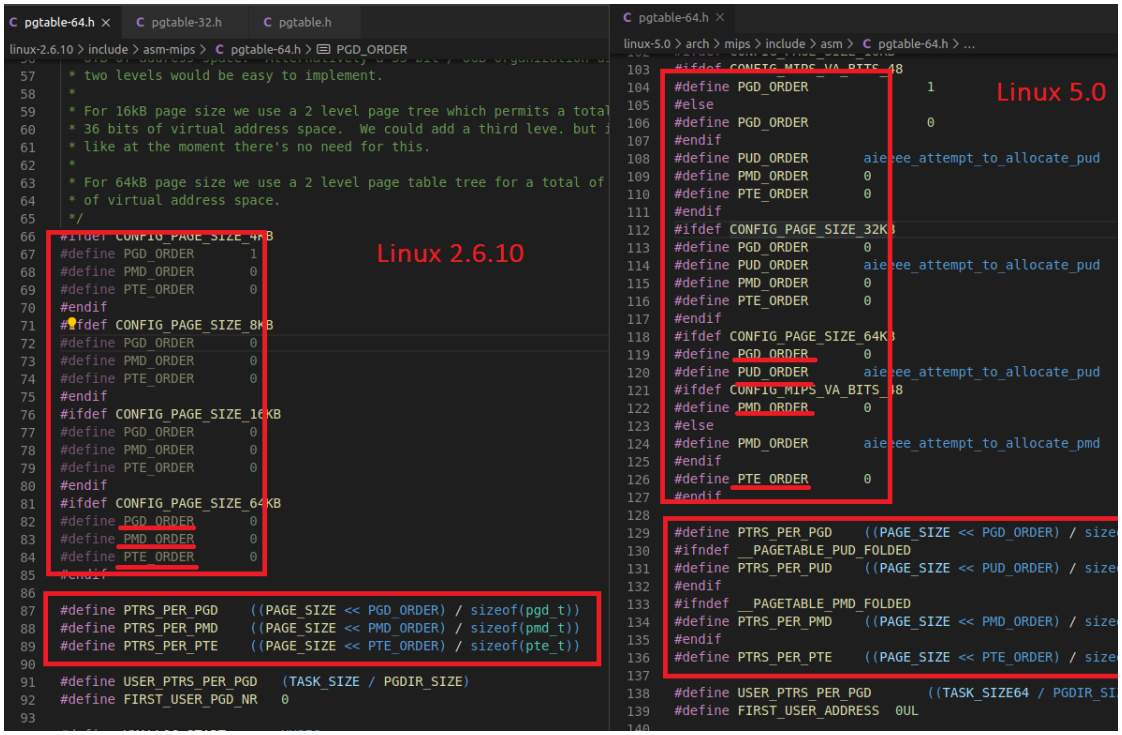
컴퓨터과학과

2013147551 김지현

사전 조사 보고서

▣ 리눅스 커널 2.6.10 버전과 5.0.0 버전의 커널 소스 분석(메모리 페이징 기법 중심)

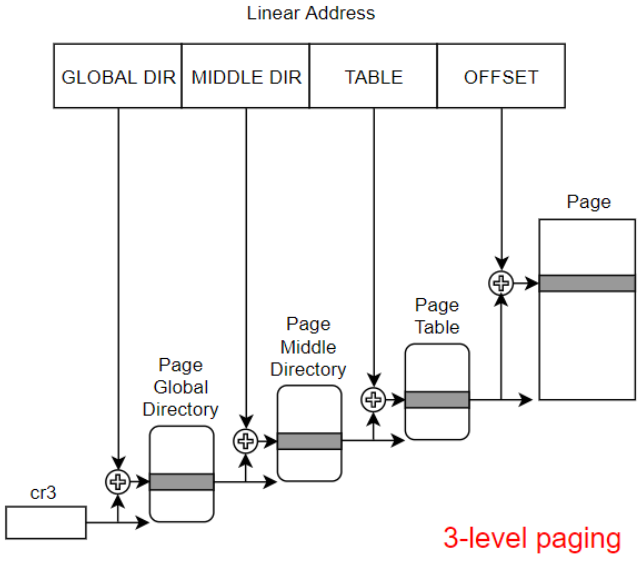
- 2.6.10 버전과 5.0.0 버전 코드에서의 페이징 레벨 비교



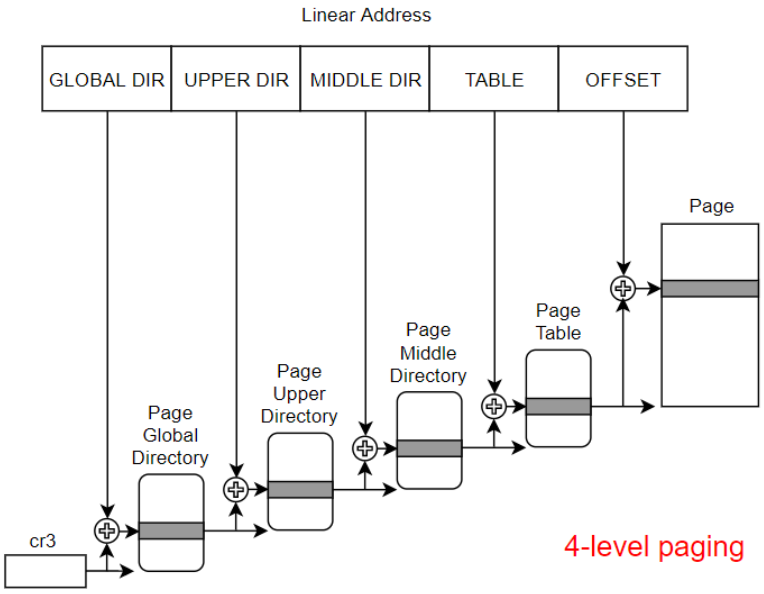
위 코드는 각각의 버전의 pgtable 헤더에서 페이지 크기에 따른 ORDER를 정의하는 부분입니다. 네모 안의 부분에서 확인할 수 있듯이, 2.6.10 버전에서는 PGD, PMD, PTE 3개의 level이 존재하는 반면, 5.0 버전에서는 PGD와 PMD 사이에 PUD level이 추가되어 총 4개의 level이 구현된 것을 확인할 수 있습니다.

- 3-level paging / 4-level paging 비교 분석 및 리눅스의 하위 level paging 지원

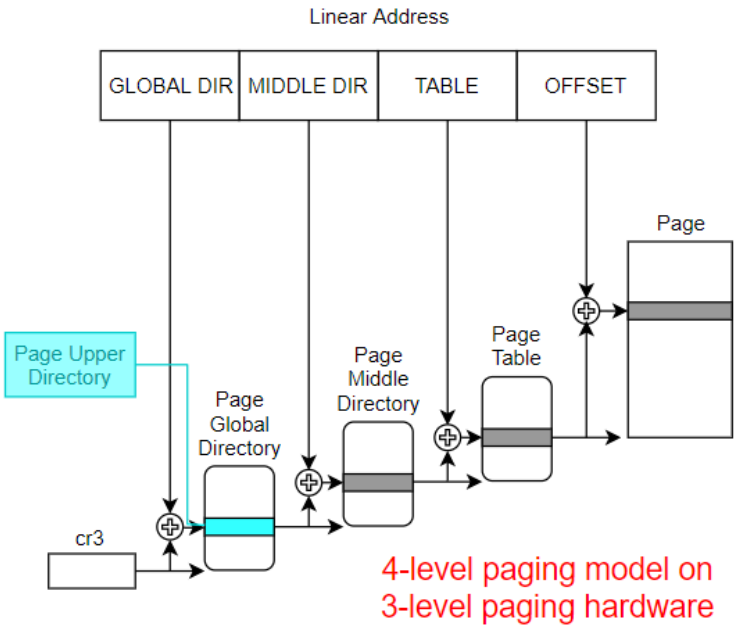
우선, 3-level paging 구조에 대한 그림은 다음과 같습니다.



다음으로, 4-level paging 구조에 대한 그림은 다음과 같습니다.



두 그림을 비교해보면 4-level paging에서는 3-level paging에서 없는 Page Upper Directory 단계가 존재합니다. 이에 따라 Linear Address 또한 세분화되어 구분됩니다. Linux의 경우 2.6.11 버전 커널부터 64비트 물리 메모리를 지원하기 위해 기존의 3-level paging 기법을 4-level paging으로 대체하였습니다. 이렇게 페이징 기법을 바꾸는 경우 기존의 하드웨어에 대한 지원 또한 이루어져야 하는데, Linux에서는 이러한 문제를 다음과 같이 해결하였습니다.



우선, Linear Address 내에서 Page Upper Directory를 제거(0비트를 할당)합니다. 포인터에서 Page Upper Directory의 위치는 변하지 않으므로 하드웨어의 비트 수에 상관없이 같은 코드를 사용할 수 있습니다. 그리고, 위의 그림과 같이 커널은 3-level paging hardware에서 사용하지 않는 Page Upper Directory의 엔트리 개수를 1개로 설정하고, 이 1개의 엔트리는 적절한 Page Global Directory의 엔트리로 매핑합니다.

- PGD, PUD, PMD, PTE의 관계와 Linear Address, Physical Address의 관계

  ■ 자료구조

4-level paging에 사용되는 4가지 테이블 엔트리는 각각의 type이 존재하며, 각각 pgd_t, pud_t, pmd_t, pte_t로 정의됩니다. 각각의 자료구조에는 각 엔트리의 value가 존재합니다.

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

  ■ 매크로

각각의 엔트리에 대해 인덱스와 오프셋 등을 계산하는 매크로가 존재하며, 인덱스의 경우 자신의 테이블 관련 정보만을 참고하며, 오프셋의 경우 자신의 상위 테이블에 대한 자료구조를 참조하여 계산합니다.

```
 */
#define pgd_offset_k(address) pgd_offset(&init_mm, address)

/*
 * the pmd page can be thought of an array like this: pmd_t[
 *
 * this macro returns the index of the entry in the pmd page
 * control the given virtual address
 */
#define pmd_index(address) \
        (((address) >> PMD_SHIFT) & (PTRS_PER_PMD-1))

/*
 * the pte page can be thought of an array like this: pte_t[
 *
 * this macro returns the index of the entry in the pte page
 * control the given virtual address
 */
#define pte_index(address) \
        (((address) >> PAGE_SHIFT) & (PTRS_PER_PTE - 1))
```

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

#define pud_offset(pgdp, addr)  \
    (((pud_t *) pgd_page_vaddr(*(pgdp))) + pud_index(addr))
#define pmd_offset(pudp,addr) \
    (((pmd_t *) pud_page_vaddr(*(pudp))) + pmd_index(addr))
#define pte_offset_kernel(dir,addr) \
    (((pte_t *) pmd_page_vaddr(*(dir))) + pte_index(addr))
```

Offset의 경우 pgd_offset은 mm, pud_offset은 pgd, pmd_offset은 pud를 참조하는 것을 확인할 수 있습니다.

  ■ 함수

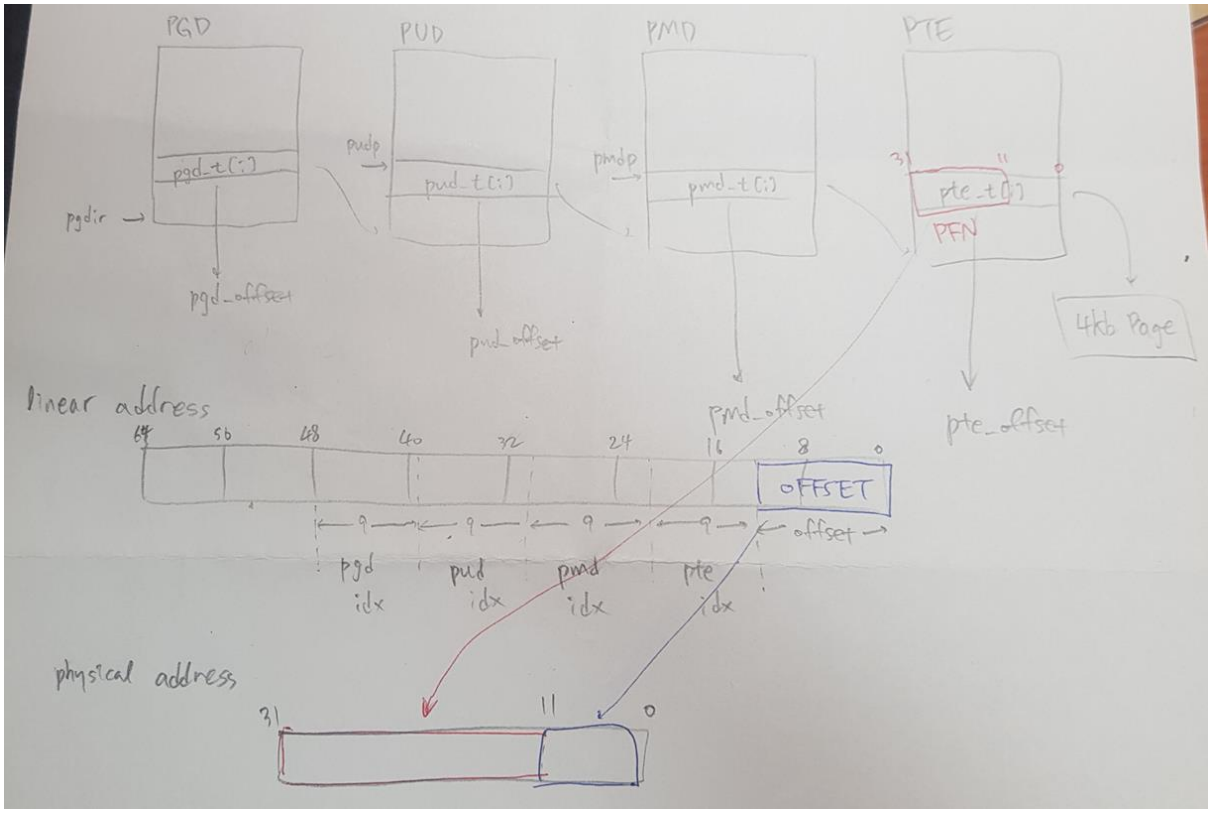
위에서 언급한 자료구조와 이들 각각에 대한 매크로를 이용하여 다양한 페이지 테이블 엔트리 함수가 구현되어 있는데, 가장 간단한 예를 들자면 pud_t 자료형을 pte_t 자료형으로 변환하는 pud_pte 함수가 있습니다.

```
909
910   extern struct page *pud_page(pud_t pud);
911   extern struct page *pmd_page(pmd_t pmd);
912   static inline pte_t pud_pte(pud_t pud)
913   {
914       return __pte_raw(pud_raw(pud));
915   }
```

이 외에도 메모리에 접근하기 위해 페이지 테이블 엔트리를 이용한 다양한 함수가 제공됩니다.

- 매커니즘의 흐름

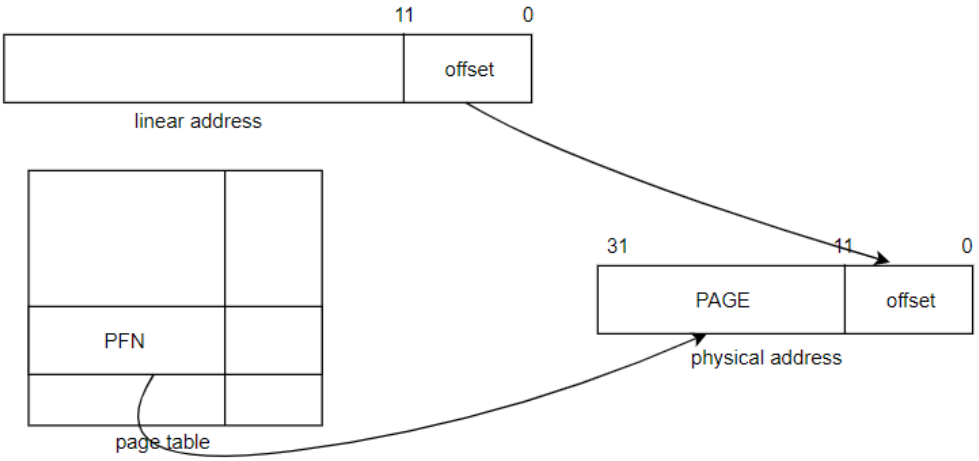
위의 자료구조, 매크로 및 함수를 이용하여 각 페이지 테이블 간의 매커니즘을 그림으로 나타내면 다음과 같습니다.



총 8바이트의 linear address에서 각 페이지 테이블 엔트리의 인덱스 정보가 9비트씩 담기고, 가장 마지막 12비트는 페이지의 offset이 됩니다. 그리고 이에 대한 physical address를 구할 때는 해당 페이지 테이블 엔트리의 PFN 주소를 앞 20비트로 사용하고, 페이지의 offset을 뒤 12비트로 사용하면 됩니다.

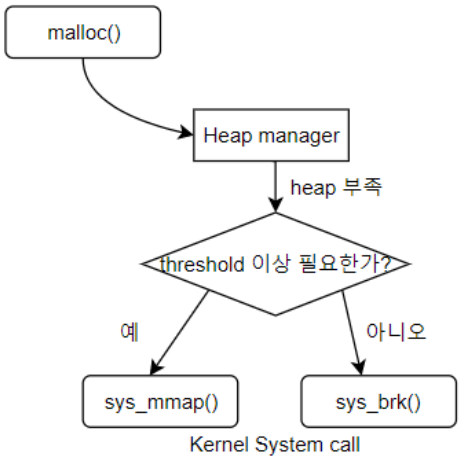
- Process 별로 할당된 page 정보를 이용하여 물리 메모리를 참조하는 과정

위에서 언급하였듯이, 각 자료구조마다 할당된 엔트리를 따라가면 Page table entry에 도달하는데, 이의 PFN address를 앞의 20비트로 사용하고, linear address의 offset을 뒤의 12비트로 사용하면 총 32비트의 physical address를 구할 수 있습니다. 이에 대한 개략적 그림은 다음과 같습니다.



▣ 사용자 프로그램에서 일정 크기의 메모리를 할당했다가 해제할 시 커널 내부에서 일어나는 동작

사용자의 프로그램이 malloc() 함수를 사용하여 일정 크기의 메모리를 할당하려 할 경우 Heap manager가 이를 처리합니다. 만약 heap memory가 부족한 경우 heap manager는 커널에게 posix system call을 통해 user anonymous memory request를 하게 되고, 이 때 커널에서 sys_brk() 또는 sys_mmap() 함수가 호출됩니다. 일반적으로 threshold 이하의 heap을 확장하려는 경우 brk를 호출하며, threshold 이상의 heap을 확장하려는 경우 mmap 함수를 호출합니다. 반대로, heap 영역을 축소할 경우 해당 영역을 unmapping하여 공간을 확보합니다.



▣ Linux에서 인터럽트 지연 처리를 위한 **Tasklet의 자료구조** 및 수행 메커니즘 분석

Linux에서는 tasklet을 사용하기 위해 tasklet에 대한 자료구조 tasklet_struct를 사용합니다.

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

각각의 필드는 다음 tasklet에 대한 포인터, 현재 tasklet의 상태, enable count, tasklet handler

function, handler function parameter를 담고 있습니다. 이 자료구조를 이용해서 tasklet을 선언하고 스케줄링하는 코드는 다음과 같습니다.

```c
#include <linux/module.h> // module_init, module_exit
#include <linux/init.h>
#include <linux/interrupt.h> // tasklet

static void hw2_tasklet_handler(unsigned long flag);
DECLARE_TASKLET(hw2_tasklet, hw2_tasklet_handler, 0);

static void hw2_tasklet_handler(unsigned long flag)
{
    tasklet_disable(&hw2_tasklet);
    printk("Tasklet running! \n");
    tasklet_enable(&hw2_tasklet);
}

static int hw2_tasklet_init(void)
{
    printk("Tasklet Scheduled!\n");
    tasklet_schedule(&hw2_tasklet);
    return 0;
}

static void hw2_tasklet_exit(void)
{
    tasklet_kill(&hw2_tasklet);
    printk("Tasklet Killed.\n");
}

module_init(hw2_tasklet_init);
module_exit(hw2_tasklet_exit);
```

우선, DECLARE_TASKLET 매크로를 이용하여 hw2_tasklet이라는 새로운 tasklet 자료구조를 선언할 수 있으며, 반대로 DECLARE_TASKLET_DISABLED 매크로를 이용하여 비활성화할 수도 있습니다. 선언 시 tasklet 수행 시 호출될 callback function을 함께 정의할 수 있으며, 이 callback function에서 tasklet을 활성화/비활성화할 수도 있습니다. 이렇게 선언한 tasklet은 tasklet_schedule 함수를 통해 스케줄링할 수 있으며, 이 tasklet를 죽일 때는 tasklet_kill 함수를 사용합니다.

▣ 참고 문헌

「Understanding the Linux Kernel」 Daniel P. Bovet and Marco Cesati, 3rd edition, 2006

수업 PPT

https://developer.ibm.com/tutorials/l-tasklets/

https://goodtogreate.tistory.com/entry/9%EC%9E%A5-%ED%94%84%EB%A1%9C%EC%84%B8%EC%8A%A4-%EC%A3%BC%EC%86%8C-%EA%B3%B5%EA%B0%84

http://jake.dothome.co.kr/pt64-api/

실습 과제 보고서

▣ 과제 수행 시스템 환경

- OS

- CPU

```
kjh@kjh-VirtualBox:~$ lscpu
Architecture:        x86_64
CPU op-mode(s):      32-bit, 64-bit
Byte Order:          Little Endian
CPU(s):              4
On-line CPU(s) list: 0-3
Thread(s) per core:  1
Core(s) per socket:  4
Socket(s):           1
NUMA node(s):        1
Vendor ID:           GenuineIntel
CPU family:          6
Model:               142
Model name:          Intel(R) Core(TM) i7-8550U CPU @ 1.80GHz
Stepping:            10
CPU MHz:             1992.004
BogoMIPS:            3984.00
Hypervisor vendor:   KVM
Virtualization type: full
L1d cache:           32K
L1i cache:           32K
L2 cache:            256K
L3 cache:            8192K
NUMA node0 CPU(s):   0-3
Flags:               fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge mca cm
ov pat pse36 clflush mmx fxsr sse sse2 ht syscall nx rdtscp lm constant_tsc rep_
good nopl xtopology nonstop_tsc cpuid tsc_known_freq pni pclmulqdq ssse3 cx16 pc
id sse4_1 sse4_2 x2apic movbe popcnt aes xsave avx rdrand hypervisor lahf_lm abm
 3dnowprefetch invpcid_single pti fsgsbase avx2 invpcid rdseed clflushopt flush_
l1d
```

- 메모리 정보

```
kjh@kjh-VirtualBox:~$ cat /proc/meminfo
MemTotal:        4036704 kB
MemFree:         1488288 kB
MemAvailable:    2434896 kB
Buffers:          119992 kB
Cached:          1009960 kB
SwapCached:            0 kB
Active:          1592904 kB
Inactive:         728232 kB
Active(anon):    1155952 kB
Inactive(anon):    44296 kB
Active(file):     436952 kB
Inactive(file):   683936 kB
Unevictable:          16 kB
Mlocked:              16 kB
SwapTotal:       2097148 kB
SwapFree:        2097148 kB
Dirty:              2776 kB
Writeback:             0 kB
AnonPages:       1191260 kB
Mapped:           464632 kB
Shmem:             49132 kB
KReclaimable:      57664 kB
Slab:             100412 kB
SReclaimable:      57664 kB
SUnreclaim:        42748 kB
KernelStack:       11456 kB
PageTables:        52396 kB
NFS_Unstable:          0 kB
Bounce:                0 kB
WritebackTmp:          0 kB
CommitLimit:     4115500 kB
Committed_AS:    6205260 kB
VmallocTotal:   34359738367 kB
VmallocUsed:           0 kB
VmallocChunk:          0 kB
Percpu:             1168 kB
HardwareCorrupted:     0 kB
AnonHugePages:         0 kB
ShmemHugePages:        0 kB
ShmemPmdMapped:        0 kB
CmaTotal:              0 kB
CmaFree:               0 kB
HugePages_Total:       0
HugePages_Free:        0
HugePages_Rsvd:        0
HugePages_Surp:        0
Hugepagesize:       2048 kB
Hugetlb:               0 kB
DirectMap4k:      171968 kB
DirectMap2M:     4022272 kB
```

- uname -a

```
kjh@kjh-VirtualBox:~$ uname -a
Linux kjh-VirtualBox 5.0.0-37-generic #40~18.04.1-Ubuntu SMP Thu Nov 14 12:06:39
 UTC 2019 x86_64 x86_64 x86_64 GNU/Linux
```

SMP 모드로 컴파일된 커널로 과제를 수행하였습니다.

▣ 커널 모듈 작성

- 프로그램 구현에 사용한 커널 함수 및 자료구조(메모리 관련 함수, Tasklet 관련 자료구조)

실행 중인 프로세스의 정보를 5초마다 수집하기 위해 5초마다 동작하는 타이머를 구현하고, 타이머가 5초마다 작동할 때마다 프로세스 정보를 가져오기 위해 tasklet을 schedule하여 구현합니다.

```
static struct timer_list timer;

void tasklet_handler(unsigned long);
DECLARE_TASKLET(ts, tasklet_handler, 0); //declare tasklet
```

여기서 tasklet_struct 자료형인 ts의 구조는 다음과 같습니다.

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

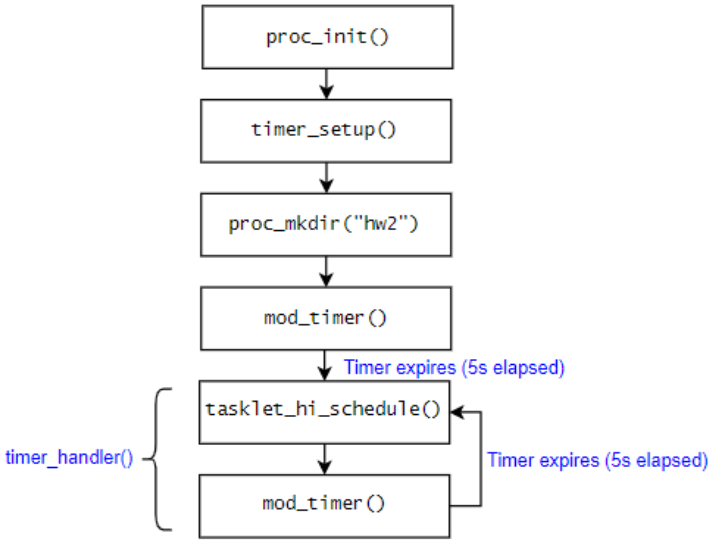
*next는 다음 tasklet을 가리키고, state는 현재 tasklet의 상태를 나타냅니다. 이 외에도, tasklet handler 함수와 이에 전달될 parameter를 담고 있습니다.

프로세스 정보를 수집하고 출력하는 과정에서 사용된 함수들은 다음과 같습니다.

```
// 1 level paging (PGD Info)
unsigned long msb_clear;
vmai[i].pgd_ba = m->pgd;
vmai[i].pgd_a = pgd_offset(m, vmai[i].ca_s);
vmai[i].pgd_v = (unsigned) pgd_val(*pgd_offset(m, vmai[i].ca_s));
msb_clear = (vmai[i].pgd_v << 1) >> 1;
vmai[i].pgd_pfna = msb_clear >> PAGE_SHIFT;                    :ist

// 2 level paging (PUD Info)
vmai[i].pud_a = pud_offset(p4d_offset(vmai[i].pgd_a, vmai[i].ca_s), vmai[
vmai[i].pud_v = (unsigned) pud_val(*pud_offset(p4d_offset(vmai[i].pgd_a,
msb_clear = (vmai[i].pud_v << 1) >> 1;
vmai[i].pud_pfna = msb_clear >> PAGE_SHIFT;

// 3 level paging (PMD Info)
vmai[i].pmd_a = pmd_offset(vmai[i].pud_a, vmai[i].ca_s);
vmai[i].pmd_v = (unsigned) pmd_val(*pmd_offset(vmai[i].pud_a, vmai[i].ca_
msb_clear = (vmai[i].pmd_v << 1) >> 1;
vmai[i].pmd_pfna = msb_clear >> PAGE_SHIFT;

// 4 level paging (PTE Info)
vmai[i].pte_a = pte_offset_kernel(vmai[i].pmd_a, vmai[i].ca_s);
vmai[i].pte_v = (unsigned) pte_val(*pte_offset_kernel(vmai[i].pmd_a, vmai
msb_clear = (vmai[i].pte_v << 1) >> 1;
vmai[i].pte_pba = msb_clear >> PAGE_SHIFT;
```

"*_offset()" 함수를 사용하여 해당 memory struct 또는 엔트리의 하위 디렉토리에서의 offset을 구하고, "*_val()" 함수를 사용하여 해당 값을 불러옵니다. 특히, intel x86 구조에서는 기술의 발전에 따른 대비를 위해 5-level paging 형태로 구성되어 있으므로 Page Global Directory와 Page Upper Directory 사이에 Page 4th Directory가 존재합니다. 즉 pud offset 함수에 대한 parameter로 pgd_t가 아닌 p4d_t가 들어가는 것에 주의하여 p4d offset을 먼저 구해야 합니다.

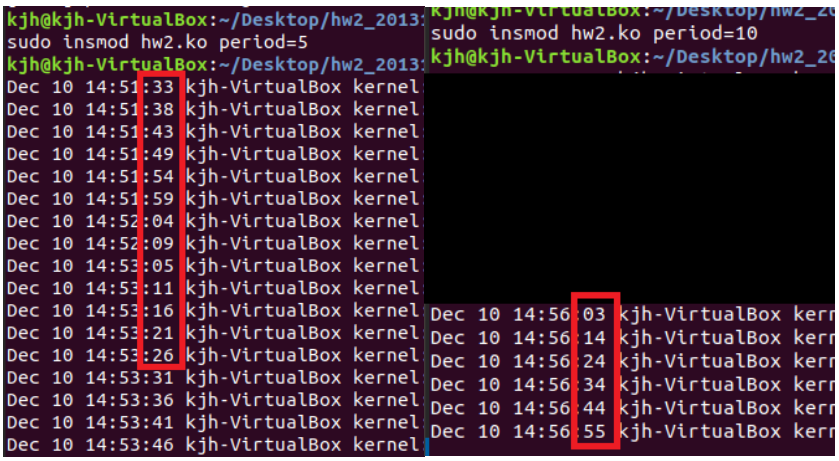
- 작성한 프로그램의 전체 구조 및 동작 과정



```c
static int period = 5; // initialization
module_param(period, int, 0);
static struct timer_list timer;

void tasklet_handler(unsigned long);
DECLARE_TASKLET(ts, tasklet_handler, 0); //declare tasklet      a);

277
278   void timer_handler(struct timer_list *t){
279       tasklet_hi_schedule(&ts);
280
281       // reschedule timer
282       mod_timer(&timer, jiffies + msecs_to_jiffies(period*1000));
283   }
284
285   int hw2_proc_init(void){
286       // initialize timer
287       timer_setup(&timer, timer_handler, 0);
288
289       hw2_proc_dir = proc_mkdir("hw2", NULL);
290
291       // schedule timer. when expires, run handler function(timer_handler)
292       mod_timer(&timer, jiffies + msecs_to_jiffies(period*1000));
293
294       return 0;
295   }
296
297   void hw2_proc_exit(void){
298       tasklet_kill(&ts);
299       del_timer(&timer);
300       proc_remove(hw2_proc_dir);
301   }
302
303   module_init(hw2_proc_init);
304   module_exit(hw2_proc_exit);
```

위 순서도는 구현한 코드의 전체적인 흐름을 나타낸 것이며, 위 코드는 그에 대한 부분을 코드로 구현한 것입니다. 우선, module_param을 통해 사용자로부터 직접 정의된 period을 받아와 저장하고, 주기적 실행에 사용될 timer를 선언합니다. 또, 프로세스의 값을 받아올 때 사용할 tasklet을 선언하고, tasklet에서 실행할 tasklet_handler 함수도 미리 prototype으로 선언합니다. 이 함수에서 폴더 생성 및 프로세스별 정보 수집을 담당할 예정이고, 이렇게 선언한 tasklet은 timer handler에서 tasklet_hi_schedule() 함수를 통해 스케줄합니다.

Tasklet 스케줄링에 대한 설계가 끝났으니 이제 timer를 설계합니다. 5초마다 특정 작업을 수행하기 위해서는 타이머가 5초 뒤에 expire될 때의 handler function에 해당 타이머를 다시 실행하면 됩니다. 타이머의 스케줄링은 timer_setup, 타이머의 작동은 mod_timer를 사용합니다.



위 두 로그는 각각 period가 5일 때, 그리고 period가 10일 때의 커널 로그입니다. 입력한 period값에 맞게 타이머가 작동하는 것을 확인할 수 있습니다.

끝으로 모듈의 생성 및 삭제입니다. insmod를 통해 모듈을 등록할 때 timer의 스케줄링과 함께 /proc/hw2 폴더를 생성하며, rmmod를 통해 모듈을 삭제할 때는 tasklet을 죽이고, timer를 제거한 뒤 사용한 /proc/hw2 폴더를 제거합니다.

```
static struct VMAI {
    char name[TASK_COMM_LEN];
    pid_t pid;
    int lut; // jiffies_to_msecs(jiffies)

    // info about each area
    unsigned long ca_s, ca_e, da_s, da_e, ba_s, ba_e, ha_s, ha_e, sla_s, sla_e, sa_s, sa_e;
    unsigned long ca_p, da_p, ba_p, ha_p, sla_p, sa_p;

    // 1 level paging (PGD Info)
    unsigned long pgd_ba, pgd_a, pgd_v, pgd_pfna;
    char pgd_ps[4], pgd_ab[2], pgd_cdb[6], pgd_pwt[14], pgd_usb[11], pgd_rwb[11], pgd_ppb[2];

    // 2 level paging (PUD Info)
    unsigned long pud_a, pud_v, pud_pfna;

    // 3 level paging (PMD Info)
    unsigned long pmd_a, pmd_v, pmd_pfna;

    // 4 level paging (PTE info)
    unsigned long pte_a, pte_v, pte_pba;
    char pte_db[2], pte_ab[2], pte_cdb[6], pte_pwt[14], pte_usb[11], pte_rwb[11], pte_ppb[2];

    // Physical address
    unsigned long phy_a;
} vmai[32769];
```

다음은 프로세스별 출력 정보를 담을 구조체 VMAI에 대한 정의입니다. 주소 등의 정보는 unsigned long, 그리고 flag에 대한 정보는 char[]로 선언하였습니다. 각 정보는 pid에 해당하는 index에 저장합니다.

```
void tasklet_handler(unsigned long data){
    // remove and make directory /proc/hw2
    proc_remove(hw2_proc_dir);
    hw2_proc_dir = proc_mkdir("hw2", NULL);
    struct task_struct* p;
    struct mm_struct* m;
    struct vm_area_struct* vm;
    int a=0;
    int i;
    int last_update_time = jiffies_to_msecs(jiffies);
    for_each_process(p) {
        m = p->active_mm;
        vm = m->mmap;
        // for kernel threads, mm is always NULL
        if (!m) continue;

        i = p->pid;

        strncpy(vmai[i].name, p->comm, TASK_COMM_LEN);

        // pid, last update time
        vmai[i].pid = i;
        vmai[i].lut = last_update_time;

        // Each area info
        vmai[i].ca_s = m->start_code;
        vmai[i].ca_e = m->end_code;

        vmai[i].da_s = m->start_data;
        vmai[i].da_e = m->end_data;

        vmai[i].ha_s = m->start_brk;
        vmai[i].ha_e = m->brk;

        vmai[i].sa_s = m->start_stack;
```

위에서 선언한 tasklet_handler의 정보입니다. Handler 함수가 실행되면 우선 기존에 존재하는 /proc/hw2 디렉토리를 삭제하고, /proc/hw2 디렉토리를 새로 생성합니다. 만약 프로세스가 커널 스레드인 경우 mm이 null이므로 제외하고 진행합니다. 그 후 pid와 이름을 저장하고, last update time의 경우 tasklet_handler가 실행되었을 때의 시간을 저장합니다. 위의 코드는 mm_struct에 존재하는 area 정보만을 우선 저장합니다.

```
// finding bss area, sharedlib area, stack area end
while (vm){
    // bss area may not exist
    if (vm->vm_start <= vmai[i].da_e && vmai[i].da_e <= vm->vm_next->vm_start){ // vm
        vmai[i].ba_s = vm->vm_end;
    }
    else if (vm->vm_end <= vmai[i].ha_s){ // vm = end of bss area
        vmai[i].ba_e = vm->vm_end;
    }

    // stack area has only 1 vm_area_struct
    if (vm->vm_start <= vmai[i].sa_s && vmai[i].sa_s <= vm->vm_end){ // vm = stack
        vmai[i].sa_e = vm->vm_end;
    }

    // sharedlib is not the start or end of the vm_area_structs
    if (vm->vm_prev != NULL && vm->vm_next != NULL){
        if (vm->vm_prev->vm_end == vmai[i].ha_e){ // vm = start of sharedlib
            vmai[i].sla_s = vm->vm_start;
        }
        else if (vm->vm_next->vm_end == vmai[i].sa_e){ // vm = end of sharedlib
            vmai[i].sla_e = vm->vm_end;
        }
    }

    vm = vm->vm_next;
}

vmai[i].ca_p = (vmai[i].ca_e-vmai[i].ca_s);
vmai[i].da_p = (vmai[i].da_e-vmai[i].da_s);
vmai[i].ba_p = (vmai[i].ba_e-vmai[i].ba_s);
vmai[i].ha_p = (vmai[i].ha_e-vmai[i].ha_s);
vmai[i].sla_p = (vmai[i].sla_e-vmai[i].sla_s);
vmai[i].sa_p = (vmai[i].sa_e-vmai[i].sa_s);
```

나머지 area의 경우 위와 같이 vm_area_struct를 이용하여 찾습니다. Bss area는 data area와 heap area의 사이에 있고, shared library area는 heap area와 stack area 사이에 있고, stack area의 경우 1개의 vm_area_struct만을 가지므로 해당 개체를 찾으면 그에 따른 end value도 구할 수 있습니다. 이렇게 얻은 각각의 start, end 값을 토대로 page 개수를 계산합니다. Page 개수의 경우 page size에 따라 달라지므로 우선은 두 end start 값의 차만 기록합니다.

```
strncpy(vmai[i].pgd_ps, ((vmai[i].pgd_v >> 7)%2 == 0) ? "4KB" : "4MB", 4);
strncpy(vmai[i].pgd_ab, ((vmai[i].pgd_v >> 5)%2 == 0) ? "0" : "1", 2);
strncpy(vmai[i].pgd_cdb, ((vmai[i].pgd_v >> 4)%2 == 0) ? "false" : "true", 6);
strncpy(vmai[i].pgd_pwt, ((vmai[i].pgd_v >> 3)%2 == 0) ? "write-back" : "write-through
strncpy(vmai[i].pgd_usb, ((vmai[i].pgd_v >> 2)%2 == 0) ? "supervisor" : "user", 11);
strncpy(vmai[i].pgd_rwb, ((vmai[i].pgd_v >> 1)%2 == 0) ? "read-only" : "read-write", 1
strncpy(vmai[i].pgd_ppb, (vmai[i].pgd_v%2 == 0) ? "0" : "1", 2);

// page number calculation
// 0: 0
// 1 ~ ps: 1
// ps+1 ~ 2ps: 2
int page_size = 4096;
if ((vmai[i].pgd_v >> 7)%2 == 1){ // if page size is 4MB
    page_size *= 1024;
}
vmai[i].ca_p = (vmai[i].ca_p - 1) / page_size + 1;
vmai[i].da_p = (vmai[i].da_p - 1) / page_size + 1;
vmai[i].ba_p = (vmai[i].ba_p - 1) / page_size + 1;
vmai[i].ha_p = (vmai[i].ha_p - 1) / page_size + 1;
vmai[i].sla_p = (vmai[i].sla_p - 1) / page_size + 1;
vmai[i].sa_p = (vmai[i].sa_p - 1) / page_size + 1;

if (vmai[i].ba_e < vmai[i].ba_s) vmai[i].ba_p = 0; // if bss not exist
if (vmai[i].ha_e == vmai[i].ha_s) vmai[i].ha_p = 0; // if heap not exist
```

다음은 각 레벨 페이징 정보입니다. Pgd info를 대표로 설명하면 우선 pgd_t 타입의 엔트리를 받아온 뒤, 이에 대한 offset과 value를 받아옵니다. 이 과정에서 msb가 set될 수 있으므로 shift를 통해 제거해줍니다. PFN address의 경우 이 value에서 12개의 lsb를 제외한 값이므로 PAGE_SHIFT(=12)만큼 right shift해준 값을 저장합니다. 각 플래그 정보의 경우 이 value에서 각 해당하는 플래그 비트를 확인하고 그에 따른 출력값을 저장합니다. 이렇게 모든 플래그 정보를 확인하고, page size 정보도 확인하였다면 이를 토대로 page size를 4kb(=4096)으로 할 지, 4mb(=4096*1024)로 할 지 정한 뒤 앞에서 구한 값에서 나눠줍니다. 추가적으로, bss area 또는 heap area의 경우 프로세스 내에 존재하지 않을 수 있는데 이 경우 앞의 계산식에서 start가 end보다 크게 나오거나 같게 나오므로 이 경우 page size를 0으로 처리해줍니다.

```
// Physical address
vmai[i].phy_a = (vmai[i].pte_pba << PAGE_SHIFT) + (vmai[i].ca_s & 0xfff); // 12 lsb

char name[6];
snprintf(name, sizeof(name), "%d", i);

// create /proc/hw2/[pid]
proc_create(name, 644, hw2_proc_dir, &hw2_proc_ops);
```

끝으로, 물리 주소의 경우 page table entry의 base address를 12만큼 left shift한 값에 logical address의 offset(12비트)을 더해줍니다. 이렇게 프로세스에 대한 모든 정보를 저장한 뒤, pid를 이름으로 하는 proc entry를 생성합니다. 그럼 for_each_process 매크로에 의해 현재 실행 중인 모든 프로세스에 대한 proc entry file이 생성됩니다.

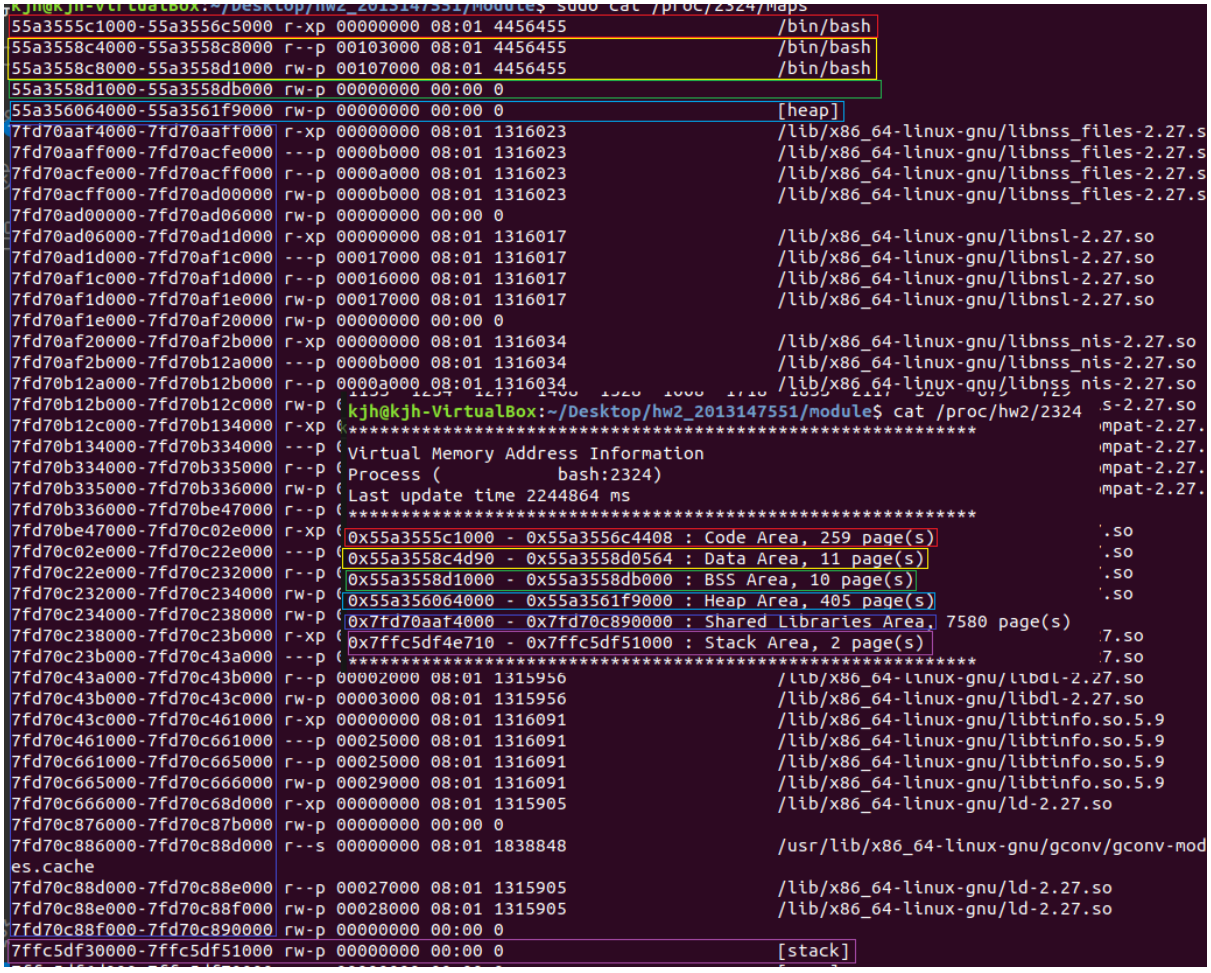
```c
static int hw2_single_show(struct seq_file *s, void *unused){

    // use file name to get pid
    const unsigned char *c = s->file->f_path.dentry->d_name.name;
    char **ep;
    int i = simple_strtol(c, ep, 10);

    seq_printf(s, "*************************************************************\n");
    seq_printf(s, "Virtual Memory Address Information\n");
    seq_printf(s, "Process (%15s:%lu)\n", vmai[i].name, vmai[i].pid);
    seq_printf(s, "Last update time %llu ms\n", vmai[i].lut);
    seq_printf(s, "*************************************************************\n");

    // print info about each area
    seq_printf(s, "0x%08lx - 0x%08lx : Code Area, %lu page(s)\n",
    vmai[i].ca_s, vmai[i].ca_e, vmai[i].ca_p);
    seq_printf(s, "0x%08lx - 0x%08lx : Data Area
    vmai[i].da_s, vmai[i].da_e, vmai[i].da_p);
    seq_printf(s, "0x%08lx - 0x%08lx : BSS Area, %lu page(s)\n",
```

```
(char [43])"0x%08lx - 0x%08lx : BSS Ar
\n"
```

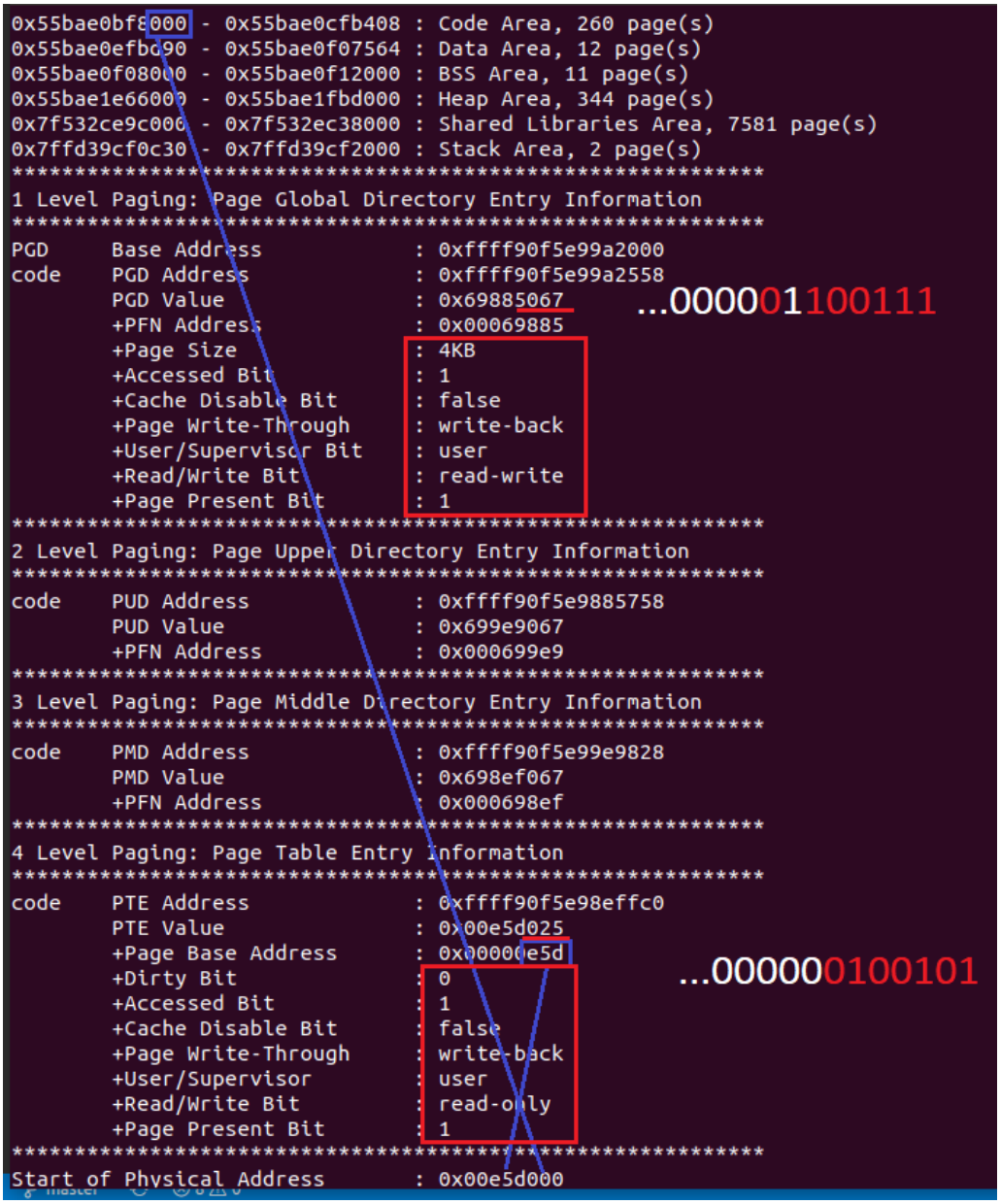
마지막으로 저장한 값들을 출력하는 부분입니다. Operation 중 open을 사용하였기 때문에 cat /proc/hw2/[pid]를 실행하는 경우 기준의 index가 반영되므로 index를 global variable로 사용하지 않는 대신에, open하는 파일의 이름을 다시 숫자로 변환하여 index로 사용합니다. 이렇게 모든 값을 출력하면 실제로 실행하여 얻게 되는 화면은 다음과 같습니다.

```
*************************************************************
Virtual Memory Address Information
Process (           bash:2672)
Last update time 8552448 ms
*************************************************************
0x55a4de5cd000 - 0x55a4de6d0408 : Code Area, 260 page(s)
0x55a4de8d0d90 - 0x55a4de8dc564 : Data Area, 12 page(s)
0x55a4de8dd000 - 0x55a4de8e7000 : BSS Area, 10 page(s)
0x55a4df6e1000 - 0x55a4df859000 : Heap Area, 376 page(s)
0x7fbc7f91a000 - 0x7fbc816b6000 : Shared Libraries Area, 7580 page(s)
0x7ffc07e90220 - 0x7ffc07e92000 : Stack Area, 2 page(s)
*************************************************************
1 Level Paging: Page Global Directory Entry Information
*************************************************************
PGD     Base Address            : 0xffff99d539916000
code    PGD Address             : 0xffff99d539916558
        PGD Value               : 0x79885067
        +PFN Address            : 0x00079885
        +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             : 0xffff99d539885498
        PUD Value               : 0x6fe5e067
        +PFN Address            : 0x0006fe5e
*************************************************************
3 Level Paging: Page Middle Directory Entry Information
*************************************************************
code    PMD Address             : 0xffff99d52fe5e790
        PMD Value               : 0x79925067
        +PFN Address            : 0x00079925
*************************************************************
4 Level Paging: Page Table Entry Information
*************************************************************
code    PTE Address             : 0xffff99d539925e68
        PTE Value               : 0x0dc56025
        +Page Base Address      : 0x0000dc56
        +Dirty Bit              : 0
        +Accessed Bit           : 1
        +Cache Disable Bit      : false
        +Page Write-Through     : write-back
        +User/Supervisor        : user
        +Read/Write Bit         : read-only
        +Page Present Bit       : 1
*************************************************************
Start of Physical Address       : 0x0dc56000
*************************************************************
```

■ 검증



위 실행 결과는 모듈 등록 후 5초가 지난 뒤 pid가 2324 인 프로세스에 대한 정보를 모듈을 통해 출력한 결과(/proc/hw2/2324)와 실제 컴퓨터에 저장되는 프로세스의 메모리 공간을 나타내는 결과(/proc/2324/maps)에 대한 비교입니다. 보이는 바와 같이 maps에서 구분할 수 있는 각 공간 안에 구해진 값들이 올바르게 포함되는 것을 확인할 수 있습니다.



다음은 각 단계 별 address와 value, 그리고 flag와 physical address에 대한 정보입니다. Flag의 경우 해당 엔트리의 value의 마지막 12비트를 참조하며, 해당 값의 의미와 실제 출력 값이 올바르게 매치되는 것을 확인할 수 있으며, 프로세스의 실제 physical address 또한 앞 20비트는 PTE value, 그리고 뒤 12비트는 virtual memory의 시작점, 즉 code area의 시작점으로 이루어지는 것을 확인할 수 있습니다.

▣ 문제점 및 해결 방법, 애로사항

과제에 필요한 값을 출력하는 과정에서 virtual memory의 각 area를 구하는 데 문제점이 있었습니다. 일반적으로 code – data – BSS – heap – shared library – stack 순서로 존재하기 때문에 이에 맞게 값을 구하도록 설계하였으나, 일부 프로세스의 경우 BSS area가 존재하지 않거나 heap area가 존재하지 않아 다른 area와 관련된 값이 엉뚱한 수치로 출력되는 문제가 발생하였습니다. 이를 해결하기 위해 BSS area의 경우 end의 위치가 start의 위치보다 앞에 있다고 구해진 경우 페이지 수를 0개로 고정하였고, heap area 또한 존재하지 않을 경우 start와 end의 위치가 같으므로 페이지 수가 0개로 출력되도록 구현하였습니다. 그리고, 과제를 수행하는 중 가장 큰 애로사항으로 커널 모듈을 등록하는 과정에서 무한루프 또는 null

pointer 예외가 발생하였을 때 우분투가 멈춰버리는 문제가 있었습니다. 이 경우 virtualbox를 끄고 다시 실행해야 하므로 사소한 디버깅 오류를 수정하는 과정에서도 많은 시간이 소모되었습니다.

▣ 참고 문헌

http://lxr.free-electrons.com

http://kldp.org/KoreanDoc/html/Kernel-KLDP

http://tldp.org/LDP/lkmpg/2.6/lkmpg.pdf

http://jake.dothome.co.kr/proc/

https://devarea.com/linux-kernel-development-creating-a-proc-file-and-interfacing-with-user-space/

https://elixir.bootlin.com/