시스템 프로그래밍 과제 #2

2016147571 김조현

1. 사전 조사 보고서

페이징 관련 자료구조

2.6.10

먼저 include/asm-i386/page.h에서 몇가지 정의된 부분들을 볼 수 있었다.

```
#define PAGE_SHIFT 12
#define PAGE_SIZE (1UL << PAGE_SHIFT)
#define PAGE_MASK (~(PAGE_SIZE-1))
```

page size는 페이징에 사용할 페이지의 크기를 정의하고, 이 값은 12로 정의된 page_shift를 사용해서, 1을 12번 shift 연산 해준 결과값이 된다. 그러므로 페이지의 크기는 4096으로 정의된 것이다. page_mask는 page_size가 차지한 12비트를 제외한 부분을 의미한다. 이 page_shift값을 이용해서 페이지 미들 디렉터리의 시작위치도 정의가 된다.

include/asm-i386/pgtable-3level-defs.h 에서 3단계 페이징이 정의된 것을 확인할 수 있었다.

```
#define PMD_SHIFT 21
#define PTRS_PER_PMD 512
```

페이지 미들디렉터리(PMD)의 시작 위치는 PMD_SHIFT로 정의한다. PMD_SHIFT 값 만큼 비트를 이동하면 페이지 미들 디렉터리가 된다. 3단계 페이징의 경우에는, 앞서 PAGE_SHIFT 값이 12비트였고, PMD_SHIFT값이 21인것으로 보아, 페이지 미들 디렉터리의 범위는 9비트인 것을 알 수 있다. PTRS_PER_PMD는 PMD 하나에 할당된 항목의 개수를 의미하고, 2의 9승인 512로 정의되어 있다.

```
#define PGDIR_SHIFT 30
#define PTRS_PER_PGD 4
```

3level paging에서 PGDIR_SHIFT는 30으로 정의 되어있고, PMD_SHIFT는 21로 정의되어 있는데, 그러므로 PGD에 9비트, PMD에 9비트, 페이지 테이블에 9비트, 페이지에 12비트를 사용해서 총 38비트를 사용하는 것을 알 수 있다.

4.4.21

include/asm-generic/page.h에서 page_shift값을 찾을 수 있었는데 다음과 같다.

```
#define PAGE_SHIFT  12
#define PAGE_SIZE   (1UL << PAGE_SHIFT)
#define PAGE_MASK   (~(PAGE_SIZE-1))
```

앞서 3-level paging과 코드가 변화가 없음을 확인할 수 있다.

include/asm-generic/pgtable-nopmd.h에서 4단계 페이징에서의 pmd에 대한 정의를 볼 수 있었다.

```
#define PMD_SHIFT   PUD_SHIFT
#define PTRS_PER_PMD    1
#define PMD_SIZE    (1UL << PMD_SHIFT)
#define PMD_MASK    (~(PMD_SIZE-1))
```

PTRS_PER_PMD는 페이지 미들 디렉터리 하나에 할당된 항목의 개수를 의미한다. 3level paging의 경우와는 다르게, 페이지 상위 디렉토리(PUD)에 대한 변수가 생긴 것을 확인할 수 있다. 여기서 PUD_SHIFT값을 통해서 PMD_SHIFT값이 정의되므로 PUD에 대한 정의 부분을 확인해 보았다.

해당 정의부분은 include/asm-generic/pgtable-nopud.h에서 찾을 수 있었다.

```
#define PUD_SHIFT   PGDIR_SHIFT
#define PTRS_PER_PUD    1
#define PUD_SIZE    (1UL << PUD_SHIFT)
#define PUD_MASK    (~(PUD_SIZE-1))
```

PUD는 4-level paging에만 존재한다. PUD_SHIFT는 PMD_SHIFT와 마찬가지로 몇비트를 이동해야 PUD인지를 나타낸다. PTRS_PER_PUD의 값은 마찬가지로 PUD 하나에 할당된 항목의 개수를 의미한다. PUD_SIZE는 PUD의 최대 크기를, PUD_MASK는 PUD영역을 제외한 부분을 의미한다.

페이지 글로벌 디렉터리 (PGD)의 정의부분은 arc/x86/include/asm/pgtable_64_types.h 에서 찾을 수 있었다.

```
#define PGDIR_SHIFT 39
#define PTRS_PER_PGD    512
#define PGDIR_SIZE  (_AC(1, UL) << PGDIR_SHIFT)
#define PGDIR_MASK  (~(PGDIR_SIZE - 1))
```

4level paging에서는 PGD도 9비트가 할당된다. PGDIR_SHIFT의 값이 39인 것을 확인할 수 있다. 따라서 PUD_SHIFT와 PMD_SHIF의 값또한 39가되고, PTRS_PER_PGD의 값은 마찬가지로 PUD 하나에 할당된 항목의 개수를 의미한다. 이 값은 2의 9승인 512인 것을 확인할 수 있다.

페이징 영역

2.6.10

페이징 영역에 대한 정보는 include/asm-alpha/page.h에서 찾을 수 있었다.

```
typedef struct { unsigned long pte; } pte_t;
typedef struct { unsigned long pmd; } pmd_t;
typedef struct { unsigned long pgd; } pgd_t;
typedef struct { unsigned long pgprot; } pgprot_t;
```

각 페이징 영역은 pte_t, pmd_t, pgd_t로 정의되어 있다. 2.6.10버전은 3-level paging이기 때문에 pud에 관한 구조체는 없는 것을 확인할 수 있다. pgprot_t는 page protection을 위한 변수인데, 페이지가 읽을 수 있는지, 쓸 수 있는지 등을 표현할때 사용하는 변수이다.

```
#define pte_val(x)      ((x).pte)
#define pmd_val(x)      ((x).pmd)
#define pgd_val(x)      ((x).pgd)
#define pgprot_val(x)   ((x).pgprot)
```

위 함수들은 type을 맞춰주기위해서 사용되는 함수들이다. 메모리 주소값을 unsigned long으로 표현하기 위해 그 type으로 맞춰준다.

```
#define __pte(x)        ((pte_t) { (x) } )
#define __pmd(x)        ((pmd_t) { (x) } )
#define __pgd(x)        ((pgd_t) { (x) } )
#define __pgprot(x)     ((pgprot_t) { (x) } )
```

위 함수들은 반대로 unsinged long 형태의 값을 각 구조체로 바꾸어주는 역할을 한다.

4.4.21

페이징 영역에 대한 정보는 include/asm-generic/page.h에서 찾을 수 있었다.

```
typedef struct { pteval_t pte; } pte_t;
typedef struct { pmdval_t pmd; } pmd_t;
typedef struct { pudval_t pud; } pud_t;
typedef struct { pgdval_t pgd; } pgd_t;
typedef struct { pteval_t pgprot; } pgprot_t;
```

각 페이징 영역은 pte_t, pmd_t, pud_t, pgd_t로 정의되어 있다. 4.4.21 버전은 4-level paging이기 때문에 pud_t가 있는 것을 확인할 수 있다. pgprot_t는 page protection을 위한 변수인데, 페이지가 읽을 수 있는지, 쓸 수 있는지 등을 표현할때 사용하는 변수이다.

페이징 을 위한 함수에 대한 정보는 arch/arm64/include/asm/pgtable-types.h에서 찾을 수 있었다.

```
#define pte_val(x)  ((x).pte)
#define pmd_val(x)  ((x).pmd)
#define pud_val(x)  ((x).pud)
#define pgd_val(x)  ((x).pgd)
#define pgprot_val(x)   ((x).pgprot)
```

위 함수들은 type을 맞춰주기위해서 사용되는 함수들이다. 메모리 주소값을 unsigned long으로 표현하기 위해 그 type으로 맞춰준다.

```
#define __pte(x)    ((pte_t) { (x) } )
#define __pmd(x)    ((pmd_t) { (x) } )
#define __pud(x)    ((pud_t) { (x) } )
#define __pgd(x)    ((pgd_t) { (x) } )
#define __pgprot(x) ((pgprot_t) { (x) } )
```

위 함수들은 반대로 unsinged long 형태의 값을 각 구조체로 바꾸어주는 역할을 한다.

page 구조체

2.6.10

page 구조체는 include/linux/mm.h에서 찾을 수 있었다.

```
struct page {
    page_flags_t flags;
    atomic_t _count;
    atomic_t _mapcount;
    unsigned long private;
    struct address_space *mapping;
    pgoff_t index;
    struct list_head lru;
    void virtual;
}
```

먼저 page_flags_t는 페이지의 상태를 나타내는 옵션값이다. 이 값은 페이지가 잠겨있는 상태인지, 해당 페이지의 변경사항이 디스크에 아직 기록되지 않은 상태인지, 최신상태인지등을 나타낸다. _count는 페이지를 참조한 횟수를 나타내고, _mapcount는 페이지 프레임을 참조하는 페이지 테이블 항목의 수를 의미한다. index는 현재 페이지의 어느 위치가 사용되고 있는지를 나타낸다. lru는 least recently used의 약자로 최근데 가장 적게 사용된 목록을 위한 이중연결리스트이다. virtual은 가상주소를 나타낸다. page 구조체는 이외에도 private, mapping등의 멤버들을 가지고 있다.

4.4.21

page 구조체는 include/linux/mm_types.h에서 찾을 수 있었다.

```
struct page {
    unsigned long flags;
    struct address_space *mapping;
    pgoff_t index;
    struct list_head lru;
    unsigned long private;
    struct mem_cgroup *mem_cgroup;
    void *virtual;
    void *shadow;
    int _last_cpupid;
}
```

2.6.10버전과 flags, mapping, index, private, lru, virtual등의 부분은 동일하다. mem_cgroup은 모든 램 공간이 kernel address sapce에 맵핑되어있을때, 그 주소들을 가지고 있는 공간이다.

mm_struct 구조체

4.4.21

```
struct mm_struct {
    struct vm_area_struct *mmap;        /* list of VMAs */
    pgd_t * pgd;
...

    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 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;
```

4.4.21 버전에서의 mm_struct 구조체의 모습은 위와 같았다. mmap 변수는 vm_area_struct 구조체를 담고 있고, start_code, end_code, start_data, end_data 등의 변수들은 각각 코드, data의 시작과 끝 주소들을 가지고 있었다. 위 변수들은 process의 각 부분의 주소들을 담고있어, 이를 통해 process의 정보를 알 수 있었다. 그리고 pgd의 값을 통해 base address의 값도 알 수 있었다.

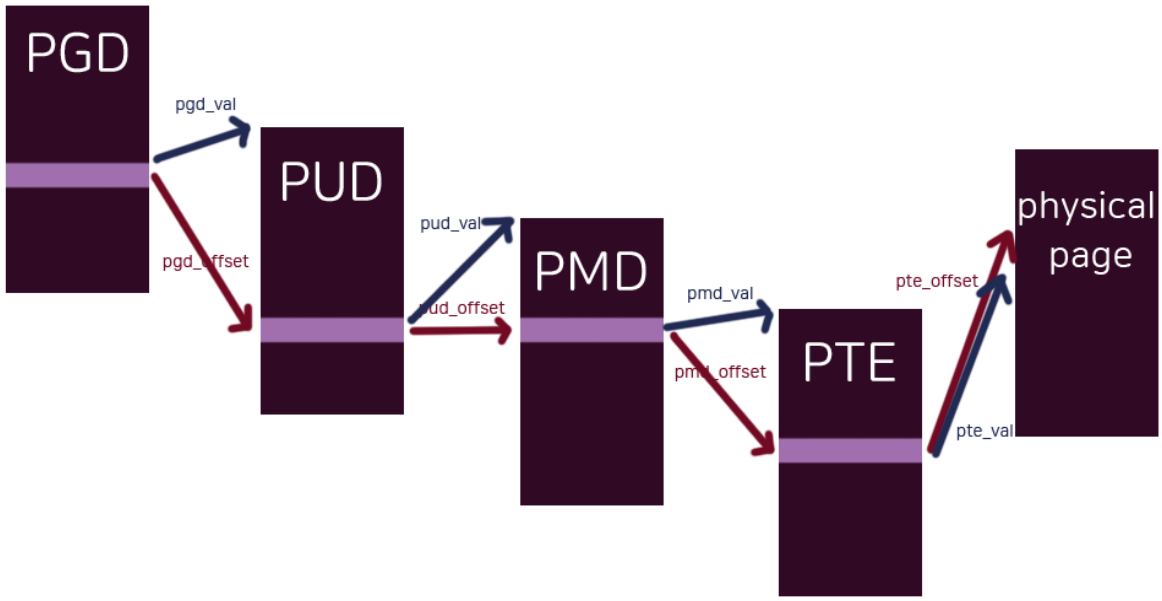
2.6.10

```
struct mm_struct {
struct vm_area_struct * mmap;       /* list of VMAs */
unsigned long addr, unsigned long len,
pgd_t * pgd;
    ...

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 rss, anon_rss, total_vm, locked_vm, shared_vm;
unsigned long exec_vm, stack_vm, reserved_vm, def_flags, nr_ptes;
```

4.4.21 버전을 먼저 봤는데, 2.6.10 버전을 보면 큰 차이점은 없어보인다. 2.6.10에서는 rss, anon_rss등의 변수가 있었는데, 4버전으로 가면서 사라지고, 4버전에서는 2에서 없었던 arg_start등의 변수가 생긴것을 확인할 수 있다.

물리메모리 참조 과정



4-level paging에서 페이지를 사용하여 physical page에 접근하는 것은 위 그림과 같다.

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

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

pgd_offset과 pud_offset 함수의 코드를 가져와 봤다. pud는 pgd address를 사용하여 값을 구할 수 있고, pmd는 pud address를, pte는 pmd address를 사용하여 함수의 값을 구할 수 있는 방식이다.

각 _val 함수들은 각 address값을 넣어주면 구할 수 있다.

3-level vs 4-level

3level paging과 4level paging의 차이점은 먼저, 3level-paging에는 없는 pud table이 4-level paging에는 있다는 점이다.

2.6.10버전의 커널과, 4.4.21 버전의 커널의 paging관련 코드들을 분석하면서 세세한 차이점까지 알 수 있었다.

그리고 4level paging을 기반으로 구현되어있는 4.4.21버전에서 3level을 사용해야 하는 경우에는, 리눅스는 아무것도 하지 않는 추가적인 페이지 (이 경우에 pud)를 에뮬레이션 하여 적용 가능하도록 한다고 한다.

메모리 할당 /해제

먼저 어떤 메모리가 할당되면 처음 프로그램이 시작될때 stack의 맨뒤를 가리키던 esp는 감소되면서 지역변수 공간이 할당된다.

mmap() 함수는 메모리를 임의 주소에 할당한 후 시작점을 알려주는 함수이고, 앞서 mm_struct에서는 brk를 보았었다. malloc은 mmap과, brk를 사용한다. brk 콜로 heap 영역을 뒤쪽으로 밀고, malloc을 통해서 할당을 해준다.

tasklet

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

4.4.21 버전에서 tasklet 구조체인 tasklet_struct의 코드는 위와 같았다. tasklet_struct 안에 해야할 일들을 담아두고 next를 사용해서 다음 일로 넘어갈 수 있었다. 이 tasklet은 1번 실행될 것이 보장되는 특징이 있다.

```
static inline void tasklet_schedule(struct tasklet_struct *t)
{
    if (!test_and_set_bit(TASKLET_STATE_SCHED, &t->state))
        __tasklet_schedule(t);
}
```

tasklet은 위의 tasklet_schedule 함수를 통해 쓸 수 있고, 위 함수가 사용되고 나면, 무조건 한번은 그 작업이 실행될 것이라는 보장을 가지게 된다.

```
static inline void tasklet_disable(struct tasklet_struct *t)
{
    tasklet_disable_nosync(t);
    tasklet_unlock_wait(t);
    smp_mb();
}

static inline void tasklet_enable(struct tasklet_struct *t)
{
    smp_mb__before_atomic();
    atomic_dec(&t->count);
}
```

위와 같이 tasklet_enable, tasklet_disable을 통해서 tasklet을 할당, 해제할 수 있다.

실습 과제 수행 보고서

개발환경

```
cc@cc-VirtualBox:~/Desktop/sys$ uname -a
Linux cc-VirtualBox 4.4.21 #1 SMP Tue Dec 4 05:36:21 KST 2018 x86_64 x86_64 x86_
64 GNU/Linux
```

개발환경은 위와 같다. 리눅스 16.04.01 버전의 가상 머신을 virtual box에서 실행하였고, 4.4.21버전의 커널 빌드를 한 후 과제를 수행하였다.

주요 자료구조

이번 실습과제를 하는데에는 먼저 mm_struct가 가장 크게 사용되었다. 그 외에 task_struct, mm_area_struct등이 사용되었으며, 이 구조체들을 사용하여 각 프로세스의 정보를 가져올 수 있었다.

코드 분석

```
module_param(period, int, S_IRUSR | S_IWUSR | S_IRGRP | S_IROTH);
MODULE_PARM_DESC(period, "An integer"); //get period parameter
```

period를 모듈을 등록할때 parameter으로 넘겨줄 수 있어야 하기 때문에, 위와같은 코드를 사용해서 period라는 변수에 parameter가 잘 전달 될 수 있도록 해주었다.

```
void my_tasklet_function(int data) // tasklet function
{
    printk("tasklet1");
 return ;
}

DECLARE_TASKLET( my_tasklet, my_tasklet_function, 2);
```

tasklet을 사용해서 process를 선택해야했으나, 해당 함수를 통해 process를 선택하게 하는 것에 실패하였다.

```
int task_num = 0;
for_each_process(task_list)
    // count process number (excluding kernel thread process)
{
    if(task_list->parent->pid == 1 || task_list->pid == 1){

        task_num+=1 ;
    }
}
//uptime.c
int random_num = idle.tv_sec % task_num;
```

tasklet을 사용하여 선택하는것을 실패하였기 때문에, 모듈을 등록할 당시 실행중인 커널 쓰레드 프로세스가 아닌 프로세스들 중에서 랜덤으로 한개의 프로세스를 뽑기 위해 해당되는 프로세스들의 개수를 세서 task_num 변수에 넣어주었다.

그리고 proc/uptime.c에서 uptime을 출력하기 위해 사용되는 코드 부분을 인용하여 idle.tv_sec 으로 현재 uptime을 초단위로 받아올 수 있었고, 이를 task_num으로 나누는 방식으로 랜덤 숫자를 얻게 되었다.

```
while(1){ // get random task_struct
    p = next_task(p);
    if(p->parent->pid == 1 || p->pid == 1){
        temp+=1;
        printk("pid : %d\n", p->pid);
        if(temp == random_num-2){
            break;
        }
    }
}
task_list = p;
```

위 코드는 커널 쓰레드 프로세스가 아닌 프로세스들 중에서, random_num만큼의 순서의 프로세스를 뽑아서 task_list 변수에 넣는다.

이후 출력하는 부분들은 이 task_list 변수를 통해서 할 수 있었다.

```
seq_printf(s, "Process (%15s:%lu)\n", "vi", task_list->pid);


seq_printf(s, "Last update time %lu%03lu ms\n", // print uptime
           (unsigned long) idle.tv_sec,
           (idle.tv_nsec / (NSEC_PER_SEC / 1000)));
printf_bar(s);

// print info about each area

seq_printf(s, "0x%08lx - 0x%08lx : Code Area, %lu page(s)\n",task_list->mm->mmap-
>vm_start, task_list->mm->mmap->vm_end, (task_list->mm->mmap->vm_end - task_list->mm-
>mmap->vm_start) / PAGE_SIZE);
seq_printf(s, "0x%08lx - 0x%08lx : Data Area, %lu page(s)\n",task_list->mm->mmap-
>vm_next->vm_start, task_list->mm->mmap->vm_next->vm_end, (task_list->mm->mmap-
>vm_next->vm_end - task_list->mm->mmap->vm_next->vm_start)/ PAGE_SIZE);
seq_printf(s, "0x%08lx - 0x%08lx : BSS Area, %lu page(s)\n",task_list->mm->mmap-
>vm_next->vm_next->vm_start, task_list->mm->mmap->vm_next->vm_next->vm_end, (
task_list->mm->mmap->vm_next->vm_next->vm_end - task_list->mm->mmap->vm_next->vm_next-
>vm_start)/ PAGE_SIZE);
seq_printf(s, "0x%08lx - 0x%08lx : Heap Area, %lu page(s)\n",task_list->mm->start_brk,
task_list->mm->brk, ( task_list->mm->brk - task_list->mm->start_brk)/PAGE_SIZE);
```

먼저 task_struct 구조체 안의 pid 값을 통해 pid값을 출력할 수 있었고, uptime의 경우 위에서 구한 idle변수를 이용하였다. 그리고 code Area, data area, bss area, heap area들은 mm_struct 구조체내의 vm_area_struct 구조체내의 변수들을 사용해서 구할 수 있었다. code area는 start_code, end_code를 이용하면 되었고, data area는 vm_area_struct를 한번 vm_next 시킨것의 vm_start, vm_end를 사용하였다. bss는 vm_next를 두번 시킨 후 vm_start, vm_end를사용하였다. heap area는 start_brk, brk을 사용하면 되었다. 모두 주소공간을 표현하기 위하여 PAGE_SIZE로 나누어서 출력해 주었다.

```
    struct vm_area_struct* ss;
    ss = task_list->mm->mmap->vm_next->vm_next->vm_next->vm_next;
    unsigned long sum = 0, sum_s, sum_e;
    sum_s = ss->vm_start;
    sum_e = ss->vm_start;

    while(ss->vm_end <= task_list->mm->mmap_base){ // to get end of the shared library

        sum += (ss->vm_end - ss->vm_start);
        sum_e = ss->vm_end;
        ss = ss->vm_next;
    }
```

shared libraries area를 구하기 위해서는 vm_area_struct 구조체를 4번 vm_next 시켜준 후, 계속적으로 vm_next 시켜주면서 vm_end를 mmap_base와 비교해서 그 처음과 끝부분을 구해주었다. 처음 부분의 주소값이 sum_s에, 끝이 sum_e에 들어갔다.

```
seq_printf(s, "0x%08lx - 0x%08lx : Stack Area, %lu page(s)\n",
            task_list->mm->start_stack, task_list->mm->start_stack - task_list->mm-
>stack_vm * PAGE_SIZE, task_list->mm->stack_vm);
```

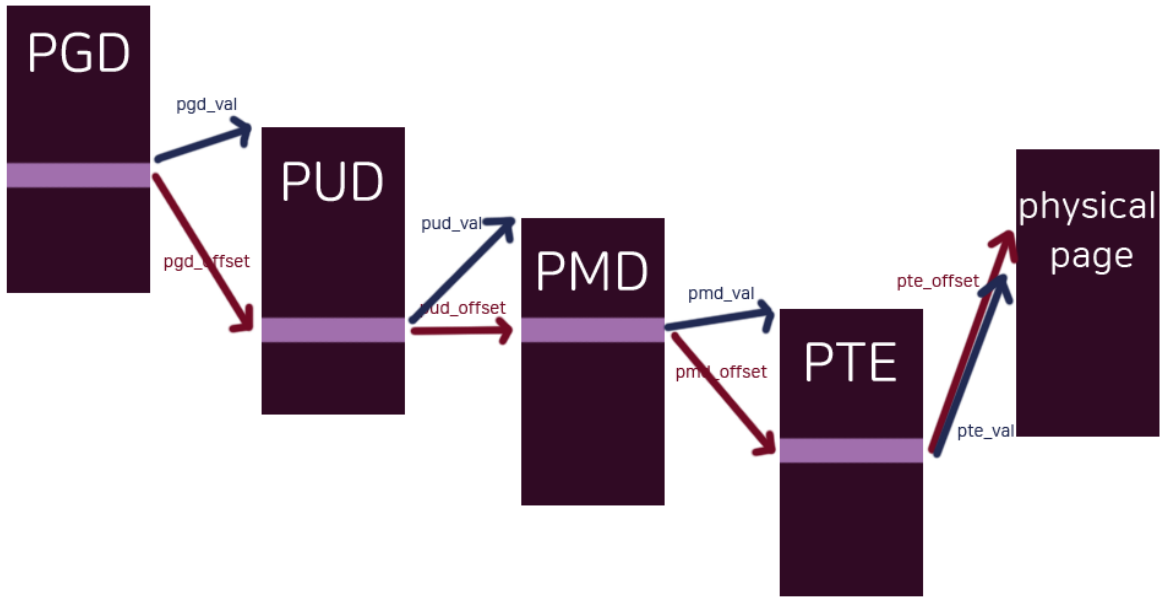
stack area를 구하기 위해서는 vm_area_struct 구조체인 mm의 start_stack과 stack_vm을 사용하였다. stack_vm은 스택이 차지하고 있는 공간을 나타내주는 변수이기 때문에, 스택이 끝나는 부분은 start_stack에 stack_vm과 페이지 사이즈를 곱해준 값이 되도록 해주었다.

```
seq_printf(s, "PGD     Base Address            : 0x%08lx\n", task_list->mm->pgd);
unsigned long baseAddr = task_list->mm->pgd;

seq_printf(s, "code    PGD Address             : 0x%08lx \n", pgd_offset(task_list->mm,
task_list->mm->mmap->vm_start));
seq_printf(s, "        PGD Value               : 0x%08lx \n",
pgd_val(*pgd_offset(task_list->mm, task_list->mm->mmap->vm_start)));
seq_printf(s, "        +PFN Address            : 0x%08lx \n",
pgd_val(*pgd_offset(task_list->mm, task_list->mm->mmap->vm_start)) / 0x1000);

seq_printf(s, "        +Page Size              : %dKB\n", PAGE_SIZE/1024); /// PRINT BY
STRING?
```

pgd의 base address는 task_struct구조체 내의 mm_struct 구조체의 pgd 값으로 구할 수 있었다. pgd address값은 pgd_offset 함수를 사용하여 구할 수 있었고, pgd value값은 pgd_val 함수를 사용하여 구할 수 있었다.



4 -level paging에서 *val, *offset 함수들의 모습을 그림으로 그려보았다. 위와같이 _offset으로 끝나는 함수들의 결과값은 각 페이지의 address값이 되고, _val로 끝나는 함수들은 각 페이지의 value 값을 나타내게 된다. 그리고 _offset으로 끝나는 함수들은 address라는 변수도 넣어주어야 하는데, 그 값은 mm_area_struct의 vm_start 값을 넣어주었다. 유의할 점은 pte 페이지의 경우에만 pte_offset이 아닌 pte_offset_kernel이 함수이름이었다. _value 함수들에는 각 페이지의 address 즉, _offset 함수의 결과값을 넣어주었다.

```
seq_printf(s, "        +PFN Address            : 0x%08lx \n",
pgd_val(*pgd_offset(task_list->mm, task_list->mm->mmap->vm_start)) / 0x1000);
seq_printf(s, "        +PFN Address            : 0x%08lx\n",
pud_val(*pud_offset(pgd_offset(task_list->mm, task_list->mm->mmap->vm_start),
task_list->mm->mmap->vm_start))/0x1000);
    seq_printf(s, "        +PFN Address            : 0x%08lx\n",
pmd_val(*pmd_offset(pud_offset(pgd_offset(task_list->mm, task_list->mm->mmap-
>vm_start), task_list->mm->mmap->vm_start), task_list->mm->mmap->vm_start))/0x1000);
seq_printf(s, "        +Page Base Address      : 0x%08lx\n",
pte_val(*pte_offset_kernel(pmd_offset(pud_offset(pgd_offset(task_list->mm, task_list-
>mm->mmap->vm_start), task_list->mm->mmap->vm_start), task_list->mm->mmap->vm_start),
task_list->mm->mmap->vm_start))/0x1000);
```

pgd, pud, pmd의 pfn address와, pte의 page base address는 16진수 기준으로 맨뒤 세자리를없애고 그 앞의 숫자들만 놓은 값이다. 그러므로 16진수 0x1000으로 나누어주어 출력했다.

pgd와 pte의 경우에 각 비트를 분석해서 cache disable bit 등을 표현해 주어야 했는데, 이는 pte 코드를 기준으로 보겠다.

```
ptee = pte_val(*pte_offset_kernel(pmd_offset(pud_offset(pgd_offset(task_list->mm,
baseAddr), baseAddr), baseAddr), baseAddr));
    ptee_2 = (ptee-(ptee/0x1000*0x1000));
    ptee_2 = 0x025;
    ptee_1 = (ptee_2 - (ptee_2/0x100 * 0x100));
    ptee_0 = ptee_1 - (ptee_1/0x10 * 0x10);
    p_2 = ptee_2/0x100;
    p_1 = ptee_1/0x10;
    p_0 = ptee_0;
```

먼저 ptee 변수에 pte_val의 결과값을 넣어주었고, 계산을 통해 p_2, p_1, p_0에 각각 16진수 기준 마지막 3개 숫자들의 값을 넣어주었다.

그리고 이 3자리의 16진수를 총 12자리의 2진수로 변환하는 과정을 거쳐서,

```
seq_printf(s,  "        +Dirty Bit              : %lx \n", (p_1-(p_1/8)*8) /4 );
seq_printf(s, "        +Accessed Bit           : %lx\n", (p_1-(p_1/4)*4) /2);
if((p_1-(p_1/2)*2) == 1){
seq_printf(s,  "        +Cache Disable Bit      : true\n" );
}else{
seq_printf(s,  "        +Cache Disable Bit      : false\n" );
}
if( (p_0/8) ==1){
seq_printf(s, "        +Page Write-Through     : write-through\n");
}else{
seq_printf(s, "        +Page Write-Through     : write-back\n");
}
if((p_0-(p_0/8)*8) /4 ==1){
seq_printf(s, "        +User/Supervisor        : user\n");
}else{
seq_printf(s, "        +User/Supervisor        : supervisor\n");
}
if((p_0-(p_0/4)*4) /2 ==1){
seq_printf(s, "        +Read/Write Bit         : read/write \n");
}else{
seq_printf(s, "        +Read/Write Bit         : read-only\n");
}
seq_printf(s, "        +Page Present Bit       : %d \n", (p_0-(p_0/2)*2) );
```

위와같이 각 비트들을 이용해서 if문을 사용해서 process의 information을 출력할 수 있었다.

```
seq_printf(s, "Start of Physical Address       : 0x%08lx\n",(
pte_val(*pte_offset_kernel(pmd_offset(pud_offset(pgd_offset(task_list->mm, task_list-
>mm->mmap->vm_start), task_list->mm->mmap->vm_start), task_list->mm->mmap->vm_start),
task_list->mm->mmap->vm_start))/0x1000)*0x1000);
unsigned long virt =
phys_to_virt(pte_val(*pte_offset_kernel(pmd_offset(pud_offset(pgd_offset(task_list->mm,
task_list->mm->mmap->vm_start), task_list->mm->mmap->vm_start), task_list->mm->mmap-
>vm_start), task_list->mm->mmap->vm_start)));
seq_printf(s, "Start of Virtual Address        : 0x%08lx\n", virt -
(virt/0x100000000)*0x100000000);
```

마지막 physical address와 virtual address의 값은 먼저 physical address의 경우 pte_val 값을 넣어주었고, physical address는 해당 값을 phys_to_virt 함수로 가상 주소로 변환하여 출력해주었다.

기타 코드들은 1차 과제에서 사용한 것들을 인용하였다.

phys_to_virt

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


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

물리주소를 가상 주소로 변환해주는 함수인 phys_to_virt의 반환값을 보면, physical address 값을 __va()에 넣은 결과인 것을 확인할 수 있다.

그리고 __va()가 정의된 부분을 보면, unsigned long 값(이경우에는 physical 주소)를 받아서, 그 값에 page offset을 더한 값을 리턴해 주는 것을 확인할 수 있다.

```
*************************************************************************
4 Level Paging: Page Table Entry Information
*************************************************************************
code    PTE Address             : 0xffff880079fabb20
        PTE Value               : 0x3288e025
        +Page Base Address      : 0x0003288e
        +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       : 0x3288e000
*************************************************************************
Start of Virtual Address        : 0x3288e025
*************************************************************************
```

결과 화면의 4level paging pte부분을 보면 위와 같다. pte_value의 값과 physical address를 phys_to_virt 함수를 통해서 역산해서 나온 결과의 값이 정확히 일치하는 것을 확인할 수 있다.

결과 화면

```
cc@cc-VirtualBox:~/Desktop/sys$ cat /proc/hw2
**************************************************************************
Student ID: 2016147571    Name: Kim Jo Hyun
Virtual Memory Address Information
Process (             vi:1719)
Last update time 6344504 ms
**************************************************************************
0x557393764000 - 0x557393782000 : Code Area, 30 page(s)
0x557393981000 - 0x557393983000 : Data Area, 2 page(s)
0x557393983000 - 0x557393984000 : BSS Area, 1 page(s)
0x557395009000 - 0x55739660f000 : Heap Area, 5638 page(s)
0x7f1780000000 - 0x7f17ac22d000 : Shared Libraries Area, 180781 page(s)
0x7ffee33ef220 - 0x7ffee33cd220 : Stack Area, 34 page(s)
**************************************************************************
1 Level Paging: Page Directory Entry Information
**************************************************************************
PGD     Base Address             : 0xffff880079fa0000
code    PGD Address              : 0xffff880079fa0550
        PGD Value                : 0x79f79067
        +PFN Address             : 0x00079f79
        +Page Size               : 4KB
        +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
**************************************************************************
2 Level Paging: Page Upper Directory Entry Information
**************************************************************************
code    PUD Address              : 0xffff880079f79e70
        PUD Value                : 0x79f7a067
        +PFN Address             : 0x00079f7a
**************************************************************************
3 Level Paging: Page Middle Directory Entry Information
**************************************************************************
code    PMD Address              : 0xffff880079f7a4d8
        PMD Value                : 0x79fab067
        +PFN Address             : 0x00079fab
**************************************************************************
4 Level Paging: Page Table Entry Information
**************************************************************************
code    PTE Address              : 0xffff880079fabb20
        PTE Value                : 0x3288e025
        +Page Base Address       : 0x0003288e
        +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        : 0x3288e000
**************************************************************************
Start of Virtual Address         : 0x3288e025
**************************************************************************
```

출력화면은 위와같다. 먼저 각 페이지의 value의 마지막 3자리수를 사용해서 프로세스의 정보를 출력하는 부분이 잘 된것을 확인할 수 있고, 커널 쓰레드 프로세스가 아닌 pid1719인 프로세스가 선택되어 잘 출력된 것을 확인할 수 있다.

아쉬운점

pte value는 pte_val 함수를 통해 얻을 수 있는데, 여기에는 pte address가 들어가게 된다. pte address는 pmd address를, pmd address는 pud address가 사용되는 형식으로 구했는데, 이때 pte_offset ()함수내의 두번째 인자인 address를 어떤 값을 넣어주어야 할지 명확하게 판단하지 못해서 그점이 조금 아쉽다.

참고자료

리눅스 커널 프로그래밍 - 한동훈 한빛미디어

리눅스 페이징

https://en.wikipedia.org/wiki/Intel_5-level_paging

리눅스에서의 메모리 할당

https://www.codentalks.com/t/topic/2279