LINUX MEMORY MANAGEMENT

System Programming 2019_2 HW2

2019 DECEMBER 2

2014190150 JUNGWOO LEE

Linux Memory Management

2019-2 Assignment2

2014190150

Jung Woo Lee

1. Purpose

To understand how Linux kernel performs memory management

To learn how to use tasklet in bottom-half with module programming

2. Background

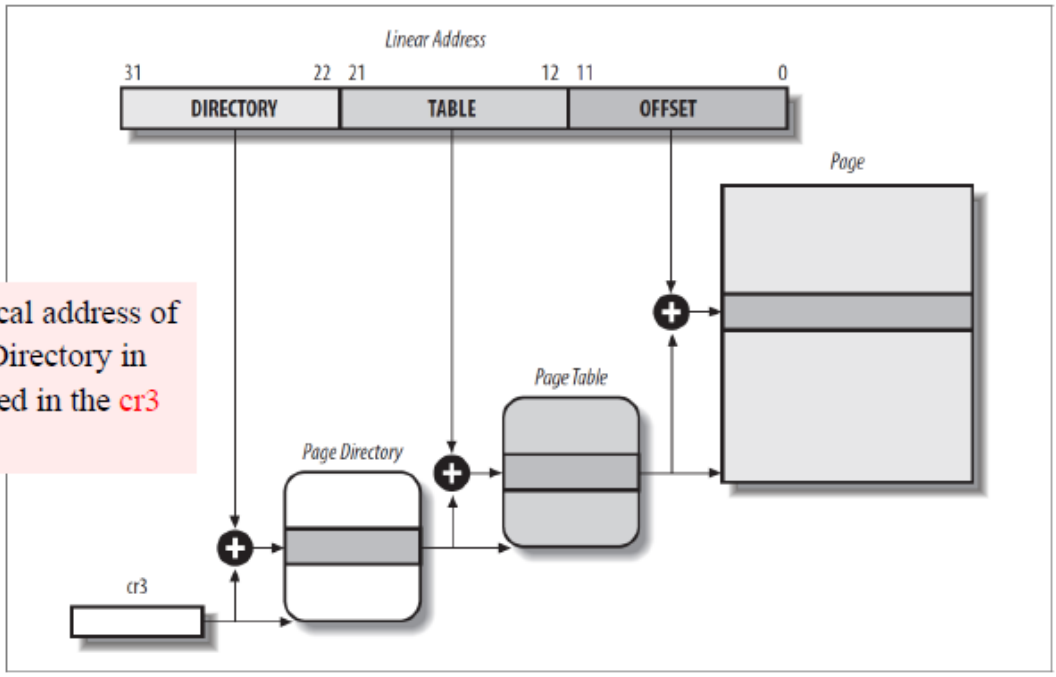
a) Paging

Linux paging is a minimal approach of segmentation. Linux decided to use paging instead of full segmentation since paging was a simpler memory management and was more portable to apply to another kinds of architectures. The concept of paging is that all the Linux processes running in User or Kernel mode use the same pair of segments to address instructions and data. Also, under this design, it is possible to store all segment descriptors in the global descriptor table. With these techniques, the main purpose of paging is to translate linear address into physical address. And thus, page defines a continuous linear address that grouped in fixed length intervals and mapped into contiguous physical address frame.

Generally, in page model, continuous linear addresses are grouped into page. Each page mapped to same size of continuous physical addresses that are grouped as frame. The data structure that represents these mapping is called page table. Each entry of table allows to contact to associated physical address with given linear addresses. The page table is saved in main memory and initialized by kernel.

In x86(32bit) architecture, usually 2 level paging is used. In linear address, 3 fields are necessary to translate into physical address. First 10 bits of segmentation is called directory that represents the corresponding entry of page directory. The starting address of page directory is saved in cr3 register and thus directory bits means the distance of entry from starting address. Second 10bits field is called table that represents the corresponding entry of page table. The selected entry of page directory has the starting address of desired page table. Therefore, by adding table field to the address in the entry of page directory, it is possible to select the wanted entry of page table. The last 12bits field is called offset that represent the actual address in the page. The selected entry of page table contains the starting address of specific page. With the given offset, we can contact to specific address in the page that holds actual physical address of

data. Since the lengths of directory and table are 10bits, the limit of address representation is 1024*1024 while offset has limits of 2^12 = 4096bits. Thus, 1024*1024*4096 = 2^32 = 4GB is the limited total address. Also, since the length of offset is 12 bits, each page can hold 4KB size.



<Figure 1. Regular Paging(x86)>

i) 3-level-paging

In Linux, the paging technique is similar to regular paging approach that is explained above. The difference of Linux is the use of 3 level tables instead of 2 level. Although the x86(32bits) architecture only need to use 2 level tables in paging, Linux created 3 level tables to prepare for the future portability. If 64bits architecture created, Kernel developers must change the memory management systems. Thus, Linux developers implemented 3 levels of tables ahead into Linux 2.6.10 version to use Linux in any system including x86 and Alpha CPU.

The mechanism of translating linear address to physical address is same to the regular paging. The data structures Linux used in 3-level paging are Page Global Directory, Page Middle Directory and Page Table Entry. The linear address of 3-level-paging system contains 4 fields. The first field of 10 bits is a page global directory, pgd_t. The pgd_t data represent the location of entry in page global directory from the starting address of pgd. The entry contains the starting address of page middle directory. The second field of 10bits called middle directory, pmd_t represents the location of entry in page middle directory. The entry of page middle directory contains the starting address of page table. The third field of 10bits is called page table, pte_t. In pte_t field, pte_offset represent offset from the starting address of selected page table. The specific position selected by 3 fields represent the starting address of page frame in physical memory. With the last LSB 13bits field, offset, the kernel is able to contact to actual desired position.

```
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;
        atomic_t mm_users;                      /* How many users with user space? */
```

<Figure 2. mm_struct data structure included in PCB>

```
#ifdef CONFIG_X86_PAE
extern unsigned long long __supported_pte_mask;
typedef struct { unsigned long pte_low, pte_high; } pte_t;
typedef struct { unsigned long long pmd; } pmd_t;
typedef struct { unsigned long long pgd; } pgd_t;
typedef struct { unsigned long long pgprot; } pgprot_t;
#define pte_val(x)      ((x).pte_low | ((unsigned long long)(x).pte_high << 32))
#define HPAGE_SHIFT     21
#else
typedef struct { unsigned long pte_low; } pte_t;
typedef struct { unsigned long pmd; } pmd_t;
typedef struct { unsigned long pgd; } pgd_t;
typedef struct { unsigned long pgprot; } pgprot_t;
```

<Figure 3. Page.h where pgd_t, pmd_t, pte_t, are defined>

In the kernel code, the information of address of process is located in task_struct. The mm_struct in task_struct has a data structure of pgd_t which is a data structure represent the starting address of page global directory. Then, the pgd_offset(mm, address) macro can be used to determine the actual position of selecting entry. The entry pointed by pgd_offset, again point to the position of pmd_offset() that returns pmd_t data. Then, entry returned by pmd_offset() also point to pte_offset(). Unlike pmd and pgd, pte has 2 variables, pte_low and pte_high. Pte_high is only used data for 3-level paging. In the page data structure, address space that points to physical address space exists. Also, if virtual memory is used, there is pointer that points to the kernel virtual address. The data structure of pgd_t, pmd_t, and pte_t is defined in page.h below include/asm-i386 directory. Also, pmd_offset is differently used when 2-level or 3-level paging is used.

```
#define pgd_index(address) (((address) >> PGDIR_SHIFT) & (PTRS_PER_PGD-1))

/*
 * 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))
```

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

<Figure 4. Pgd_offset macro and pte_offset macro in pgtable.h>

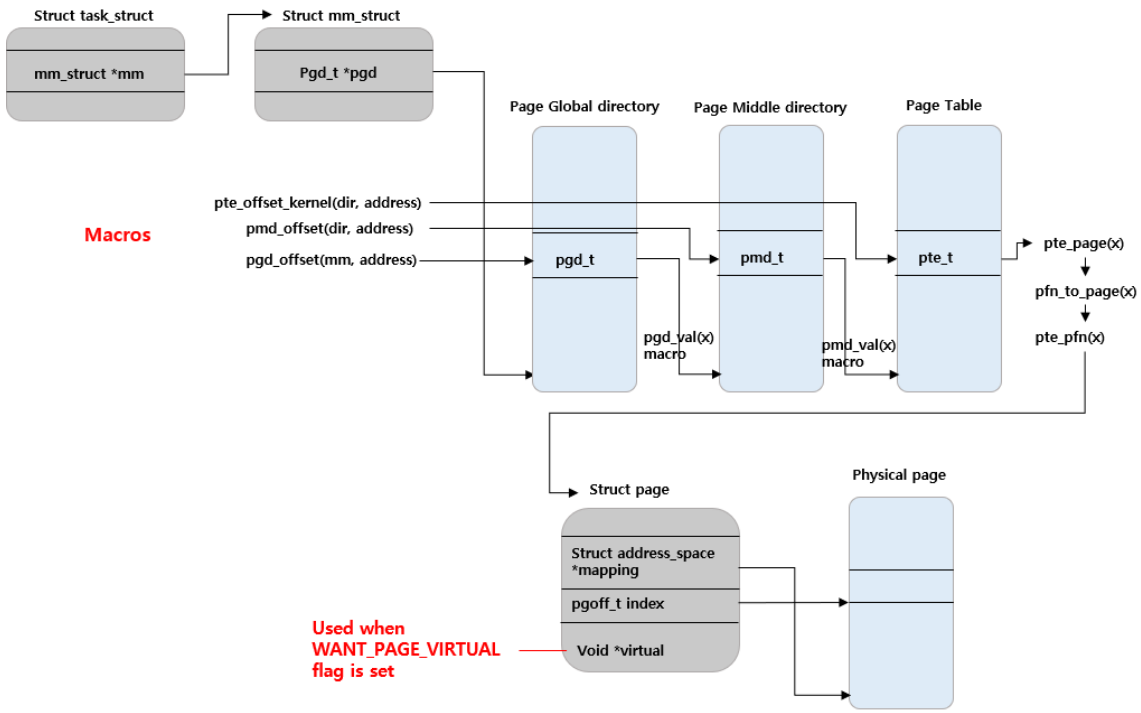
```c
static inline pmd_t * pmd_offset(pgd_t * dir, unsigned long address)
{
        return (pmd_t *) dir;
}
```

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

<Figure 5. Pmd_offset function and macro used in 2-level(up), and 3-level(down)>

```c
struct page {
        page_flags_t flags;             /* Atomic flags, some possibly
                                         * updated asynchronously */
        atomic_t _count;                /* Usage count, see below. */
        atomic_t _mapcount;             /* Count of ptes mapped in mms,
                                         * to show when page is mapped
                                         * & limit reverse map searches.
                                         */
        unsigned long private;          /* Mapping-private opaque data:
                                         * usually used for buffer_heads
                                         * if PagePrivate set; used for
                                         * swp_entry_t if PageSwapCache
                                         */
        struct address_space *mapping;  /* If low bit clear, points to
                                         * inode address_space, or NULL.
                                         * If page mapped as anonymous
                                         * memory, low bit is set, and
                                         * it points to anon_vma object:
                                         * see PAGE_MAPPING_ANON below.
                                         */
        pgoff_t index;                  /* Our offset within mapping. */
        struct list_head lru;           /* Pageout list, eg. active_list
                                         * protected by zone->lru_lock !
                                         */
        /*
         * On machines where all RAM is mapped into kernel address space,
         * we can simply calculate the virtual address. On machines with
         * highmem some memory is mapped into kernel virtual memory
         * dynamically, so we need a place to store that address.
         * Note that this field could be 16 bits on x86 ... ;)
         *
         * Architectures with slow multiplication can define
         * WANT_PAGE_VIRTUAL in asm/page.h
         */
#if defined(WANT_PAGE_VIRTUAL)
        void *virtual;                  /* Kernel virtual address (NULL if
                                           not kmapped, ie. highmem) */
#endif /* WANT_PAGE_VIRTUAL */
};
```

<Figure 6. Page data structure of kernel 2.6.10>



<Figure 7. Flow chart of 3-level-paging>

ii) 4-level-paging

The Linux updated paging technique from 3 level into 4 level to prepare for the 128 bits system in future. It was implemented since the 2.6.11 version to 5.0 version. The page upper table was newly introduced between page middle table and page global directory. Most of mechanism of paging is similar to regular paging and 3-level paging. However, the biggest difference is a data structure of page and the simplified but more separated header files in the kernel.

To begin, mm_struct is located in mm_types.h instead of sched.h. Also, pgb pointer still exists and the pgtables_bytes that represent offset of pte table is implemented. Additionally, the page data structure is also defined in mm_types.h. The struct page changed a lot compared to that of 3-level paging. The page data structure contains many unions and inner structs for better portability with variables for different swapping algorithms such as LRU and RCU.

```
struct page {
        unsigned long flags;            /* Atomic flags, some possibly
                                         * updated asynchronously */
        /*
         * Five words (20/40 bytes) are available in this union.
         * WARNING: bit 0 of the first word is used for PageTail(). That
         * means the other users of this union MUST NOT use the bit to
         * avoid collision and false-positive PageTail().
         */
        union {
                struct {        /* Page cache and anonymous pages */
                        /**
                         * @lru: Pageout list, eg. active_list protected by
                         * zone_lru_lock.  Sometimes used as a generic list
                         * by the page owner.
                         */
                        struct list_head lru;
                        /* See page-flags.h for PAGE_MAPPING_FLAGS */
                        struct address_space *mapping;
                        pgoff_t index;          /* Our offset within mapping. */
                        /**
                         * @private: Mapping-private opaque data.
                         * Usually used for buffer_heads if PagePrivate.
                         * Used for swp_entry_t if PageSwapCache.
                         * Indicates order in the buddy system if PageBuddy.
                         */
                        unsigned long private;
                };
```

<Figure 8. Part of Struct page in kernel 5.0>

Moreover, in page.h, the data structures of each page tables addresses are more simplified by deleting pte_low and pte_high. Another different point is that pmd_t represents the array of 16 size. Moreover, it is not able to observe the newly added table for 4-level paging. The reason is that in the version 5.0 kernel, all the pgtable.h file is separated by the architectures. Only general structures and xxx_var() macro exists in general pgtable.h in kernel. All the xxx_offset() macro are implemented in the architecture based pgtable.h and even changed into inline function except pgd_offset(). Additionally, all the flag checking functions are located into pgtable.h. Also, every progress must check the presence of the table first.

```
static inline int pmd_present(pmd_t pmd)
{
        /*
         * Checking for _PAGE_PSE is needed too because
         * split_huge_page will temporarily clear the present bit (but
         * the _PAGE_PSE flag will remain set at all times while the
         * _PAGE_PRESENT bit is clear).
         */
        return pmd_flags(pmd) & (_PAGE_PRESENT | _PAGE_PROTNONE | _PAGE_PSE);
}
```

<Figure 9. Present flag checking of pmd. (other tables have same structures>

However, the calculation to transform the linear address into physical address is still similar. First, the kernel starts from the *pgd in the mm_struct that represents the starting address of page global directory. Then with the pgd_offset(), select the entry of PGD that contains the starting address of page upper directory. Similarly, with the pud_offset(), contact to the entry of PUD that has the starting address of page middle directory and keep going to the page table entry. At last, by using pte_page() to get the page frame descriptor of desired address.

```
typedef struct {
        unsigned long pte;
} pte_t;
typedef struct {
        unsigned long pmd[16];
} pmd_t;
typedef struct {
        unsigned long pgd;
} pgd_t;
typedef struct {
        unsigned long pgprot;
} pgprot_t;
typedef struct page *pgtable_t;
```

```
#if CONFIG_PGTABLE_LEVELS > 3
typedef struct { pudval_t pud; } pud_t;

static inline pud_t native_make_pud(pmdval_t val)
{
        return (pud_t) { val };
}

static inline pudval_t native_pud_val(pud_t pud)
{
        return pud.pud;
}
```

```
/* Find an entry in the third-level page table.. */
static inline pud_t *pud_offset(p4d_t *p4d, unsigned long address)
{
        return (pud_t *)p4d_page_vaddr(*p4d) + pud_index(address);
}
```

<Figure 10. (1) defined table values in generic / (2) x86_pud define / (3) xxx_offset() function>

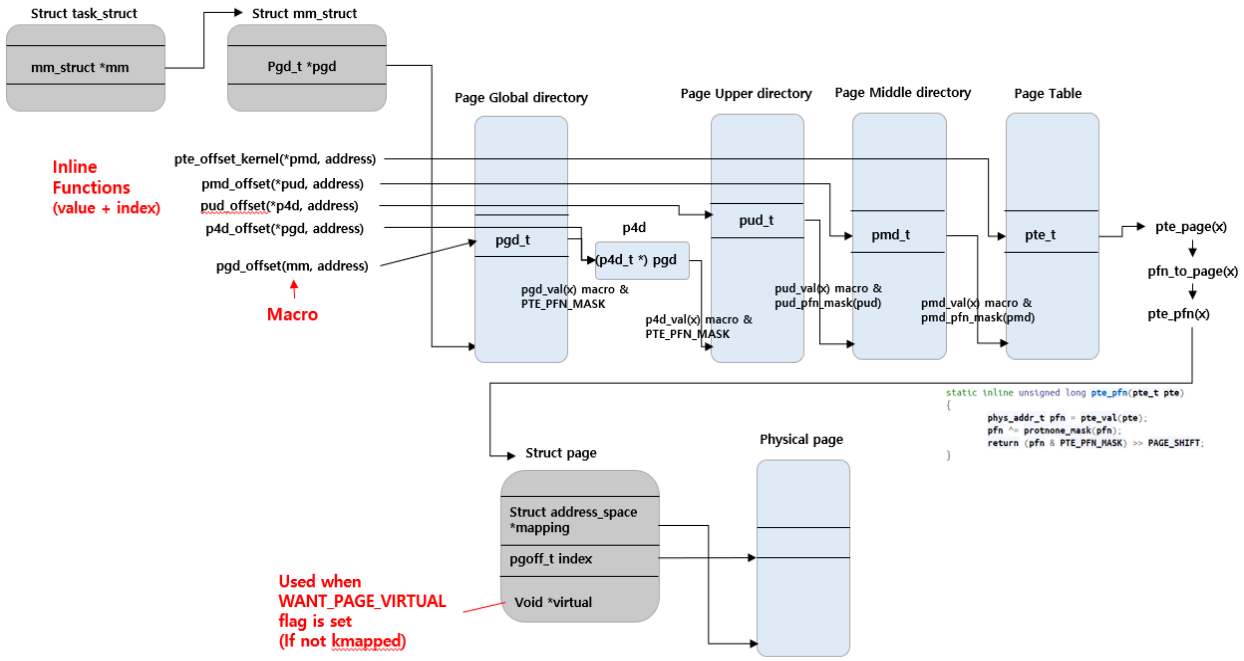
In figure 10.(3), the p4d page table is observed instead of pgd. In kernel version 5.0, the Linux developers actually implemented systems for 5-level table of paging. However, the 4 level paging system do not use p4d by setting the directory with entry size 1 and set the value as same as pgd_t value. Also, if the architecture is 32 bits or 2-level supporting system, pud and pmd can be skipped like p4d by applying 1 entry size table with same value of previous table's entry value. In the 5.0 kernel, the applied portability can be found in the pgtable-no4d.h or pgtable-nopud.h files in asm-generic.

```
#if CONFIG_PGTABLE_LEVELS > 4
typedef struct { p4dval_t p4d; } p4d_t;

static inline p4d_t native_make_p4d(pudval_t val)
{
        return (p4d_t) { val };
}

static inline p4dval_t native_p4d_val(p4d_t p4d)
{
        return p4d.p4d;
}
#else
#include <asm-generic/pgtable-nop4d.h>

static inline p4d_t native_make_p4d(pudval_t val)
{
        return (p4d_t) { .pgd = native_make_pgd((pgdval_t)val) };
}

static inline p4dval_t native_p4d_val(p4d_t p4d)
{
        return native_pgd_val(p4d.pgd);
}
#endif
```

<Figure 11. The value setting according to the CONFIG_PGTABLE_LEVELS in x86 specific>

```
static inline p4d_t *p4d_offset(pgd_t *pgd, unsigned long address)
{
        return (p4d_t *)pgd;
}
```

<Figure 12. The generic version in nop4d.h file>



<Figure 13. Flow chart of 4-level paging in kernel v5.0>

b) Memory Allocation and Deallocation in User Process

Unlike Kernel Memory allocation, the memory allocation by user process cannot be trusted. Thus, the kernel must prepare to catch all addressing errors caused by user mode processes. Therefore, in general case, process requests for dynamic memory allocation is considered as non-urgent. Since the kernel tries to defer allocating page frame to user processes, it does not get additional page frames but, gets the authority to use a new memory region. The kernel can modify a user address space dynamically by removing or adding range of linear addresses that is the multiple of 4KB.

Process gets new memory regions when a process is newly created, becomes a different process by exec() functions, requests the dynamic memory mapping by mmap() system call, has to add data on the user stack, shares data with other process via shared memory and expands the heap.

When user application process requests memory allocation, kernel first gives the rights to have certain size of memory. Then, kernel must check whether the request of memory regions or address is in good area. Then the kernel can allocate a linear address interval in high level by using do_mmap(). In the do_mmap function, a new address interval is created. This memory region can be represented as vm_area_struct in mm_struct. At last step of allocation, the memory region described by vm_area_struct can be allocated in frame by handle_mm_falut() that checks the page fault and performs demand paging. When the user application request to release the linear address interval, do_munmap() function is used. It invokes unmap_region() function to release the page frames belonging to the interval. Also, the vm_area_struct that describe certain region should be deleted and data structures such as mm_struct and page tables must be updated.

c) Tasklet

A tasklet is one type of bottom half interrupt that run on software interrupt context which cannot be blocked and not-preemptable in most case. It's main concept is simple and easy use of softirq and thus, it is flexible and can be dynamically created. There are two categories of tasklet exist according to the priority of interrupts. These categories are defined as TASKLET_SOFTIRQ and HI_SOFTIRQ in enum numbering.

Since tasklet is running in interrupt context, it cannot block or locked. Also, once tasklet is scheduled, the tasklet is guaranteed at least one execution. Tasklet does not support nested executions. Moreover, tasklet can be considered as simpler use since simultaneous running on different processors is restricted and thus, lock is not necessary. Therefore, tasklet is preferred to implement deferrable functions such as device drivers.

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

```
#define DECLARE_TASKLET(name, func, data) \
struct tasklet_struct name = { NULL, 0, ATOMIC_INIT(0), func, data }

#define DECLARE_TASKLET_DISABLED(name, func, data) \
struct tasklet_struct name = { NULL, 0, ATOMIC_INIT(1), func, data }
```

<Figure 14. tasklet data structure(left) and macro to declare(right)>

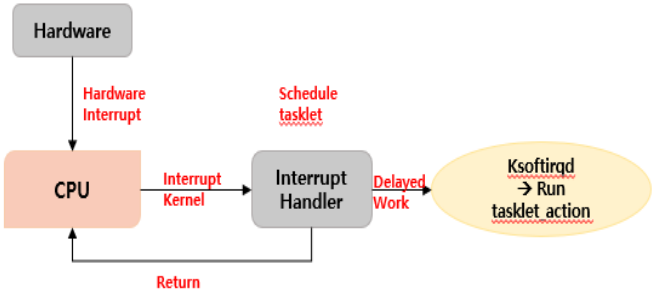
A data structure of tasklet is located in the include/linux/interrupt.h file. As we can see in the first field of tasklet, it is constructed as a single linked list. A *next points to the next tasklet occurred. Also, the state part represents that whether tasklet is scheduled for execution or running. As we can observe in declaring macro, the state of tasklet is always 0 at the beginning. The third field count represent the enabled state of tasklet. When count is 0, tasklet is enabled to executed and when count is 1, tasklet is not able to execute until it changed to the enabled state again. The 4th field is a pointer to handler function that would be executed when the tasklet is executed. The last field data represents to the arguments that would be used in function executions. A tasklet is simple to use since user only have to provide the functions and data to create for the specific interrupts.

```
static DEFINE_PER_CPU(struct tasklet_head, tasklet_vec);
static DEFINE_PER_CPU(struct tasklet_head, tasklet_hi_vec);

void __tasklet_schedule(struct tasklet_struct *t)
{
        __tasklet_schedule_common(t, &tasklet_vec,
                                  TASKLET_SOFTIRQ);
}
EXPORT_SYMBOL(__tasklet_schedule);
```

<Figure 15. Define and schedule of tasklet in kernel>

To begin, kernel defines 2 lists of tasklet in each CPU. Additionally, the drivers declare the corresponding tasklet in source file with linked handling functions and data. Then, whenever drivers' soft interrupt raised, the tasklet changes its count into enable and scheduled into corresponding lists between tasklet_vec and tasklet_hi_vec.

```
static void kbd_bh(unsigned long dummy)
{
        struct list_head * node;
        unsigned char leds = getleds();
        ...
}

DECLARE_TASKLET_DISABLED(keyboard_tasklet, kbd_bh, 0);

...

int __init kbd_init(void)
{
        int i;
        ...

        for (i = 0 ; i < MAX_NR_CONSOLES ; i++)
                kbd_table[i] = kbd0;

        input_register_handler(&kbd_handler);

        tasklet_enable(&keyboard_tasklet);
        tasklet_schedule(&keyboard_tasklet);

        return 0;
}
```



<Figure 16. Keyboard driver tasklet(left) and flow chart of tasklet(right)>

When the kernel initialized, tasklet is linked with given softirq numbering corresponds to tasklet or hi_tasklet with corresponding action functions by softirq_init(). The tasklet_action function is declared in interrup.h file. Both tasklet lists are actually positioned in ksoftirqd that is a group of deferred softirqs in pending list. When we observe the tasklet action function, the while loop never ends until when the tasklets in the list no more exists. Therefore, once tasklet is executed in ksoftirqd, all the tasklet in the list will be performed. In the while loop, kernel checks whether tasklet is able to lock, which means changing the state of tasklet into running to prevent simultaneous running in the other CPU. Then, checks the count field to determine whether tasklet is enabled. If these checks are passed, then tasklet invoke the provided functions with the provided data as an argument.

```
static void tasklet_action_common(struct softirq_action *a,
                                  struct tasklet_head *tl_head,
                                  unsigned int softirq_nr)
{
        struct tasklet_struct *list;

        local_irq_disable();
        list = tl_head->head;
        tl_head->head = NULL;
        tl_head->tail = &tl_head->head;
        local_irq_enable();

        while (list) {
                struct tasklet_struct *t = list;

                list = list->next;

                if (tasklet_trylock(t)) {
                        if (!atomic_read(&t->count)) {
                                if (!test_and_clear_bit(TASKLET_STATE_SCHED,
                                                        &t->state))
                                        BUG();
                                t->func(t->data);
                                tasklet_unlock(t);
                                continue;
                        }
                        tasklet_unlock(t);
                }

                local_irq_disable();
                t->next = NULL;
                *tl_head->tail = t;
                tl_head->tail = &t->next;
                __raise_softirq_irqoff(softirq_nr);
                local_irq_enable();
        }
}
```


<Figure 17. Comments to the tasklet action function that mapped to tasklet softirq>

3. Environment

The assignment was performed on virtual machine environment of VM ware. The operating system was Ubuntu 18.04 version with 5.0 Linux kernel to customize. The system had 2 core CPU with 4GB of main memory.

```
jwlee@2014190150:~$ uname -a
Linux 2014190150 5.0.0-23-generic #24~18.04.1-Ubuntu SMP Mon Jul 29 16:12:28 UTC
 2019 x86_64 x86_64 x86_64 GNU/Linux
```

```
jwlee@2014190150:~$ cat /proc/cpuinfo
processor       : 0
vendor_id       : GenuineIntel
cpu family      : 6
model           : 94
model name      : Intel(R) Core(TM) i5-6300HQ CPU @ 2.30GHz
stepping        : 3
microcode       : 0xc6
cpu MHz         : 2304.002
cache size      : 6144 KB
```

```
processor       : 1
vendor_id       : GenuineIntel
cpu family      : 6
model           : 94
model name      : Intel(R) Core(TM) i5-6300HQ CPU @ 2.30GHz
stepping        : 3
microcode       : 0xc6
cpu MHz         : 2304.002
cache size      : 6144 KB
```

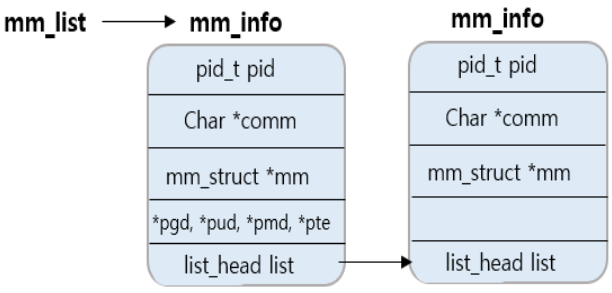
<Figure 18. Environment information>

4. Implementation

i) Data structures and Kernel functions

The purpose of project was to update memory information of user process and show by using proc file system. To maintain these data I created mm_info data structure that holds all the data that will be used to print in proc file. Also, to form a list, I included list_head data structure from linux/list.h.

```
struct mm_info {

    pid_t pid;
    char *comm;
    unsigned long up_time;
    struct mm_struct *mm;
    pgd_t *pgd;
    pud_t *pud;
    pmd_t *pmd;
    pte_t *pte;

    struct list_head list;
};

extern struct list_head mm_list;
```



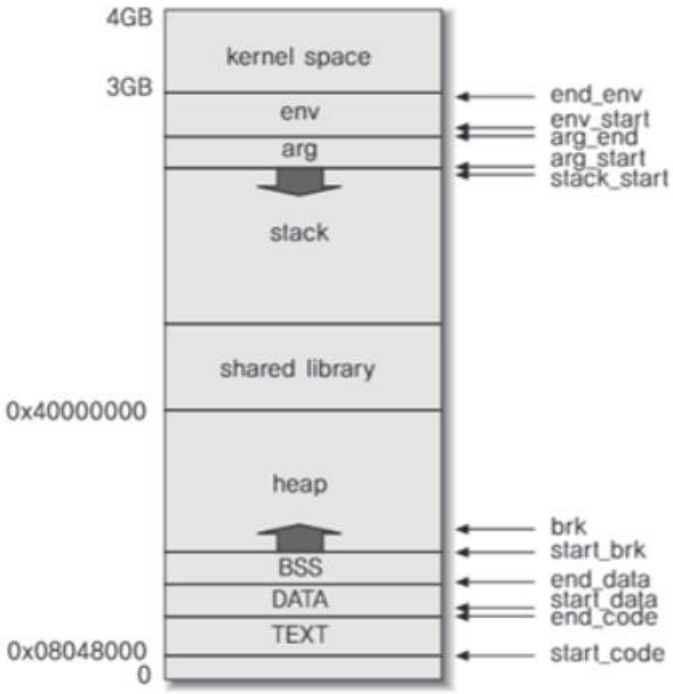
<Figure 19. Mm_info Data structure>

Also, to use tasklet, I must declare tasklet data structure with specific handler function. It is one type of bottom half irq, however, it cannot be preempted and easy to make and schedule. To make tasklet, DECLARE_TASKLET is used with name of tasklet and the name of handler function. Tasklet has state and by using tasklet_schedule, we are able to change tasklet state to enable to allow tasklet to perform its handling function. In the tasklet handler function, I implemented remove of data, file and directory. Then, I update the information of tasks to mm_list by using add_data function. At last step of tasklet function, it creates directory hw2 and all the proc files with name of each process id.

The tasklet is only scheduled when the module initiated and when the given time period is expired. The timer is implemented by using timer_setup and mod_timer. With the timer_setup function, we can initiate the dynamic timer and assign certain function to handle a timer interrupt. In my implementation, when the timer expires given time, tasklet will be scheduled to update data in the timer callback function. Then, by using mod_timer, I assigned next expire time with HZ constant that defined in kernel.

The add_data function simply get pointer of task_struct as a parameter to fill the mm_info data structure with given data and add mm_info data structure to the mm_list. Since add_data used kzalloc to maintain data, remove_data function to kfree the memory space is required.

With the maintained list data structure, I used seq_file data structure to show the data in proc file system. Seq_file gets a list of data and automatically iterate the list with given operations. In mm_start, the seq_file iteration starts when we request the cat or vi command in terminal. Then, with mm_next operation, the next entry of list will be provided. Then with mm_show operation, I was able to check whether pointed entry contains requested information by comparing pid of entry and requested file name. This requested file name can be obtained by global variable. If the selected entry is what we are looking for, mm_show will show the wanted data on proc file. However, if the pid does not match, mm_next operation will occur again to get next entry of the list that contains information. If the iteration ends, the mm_stop operation is called. These operations that I implemented will be the component of seq_operations data structure.

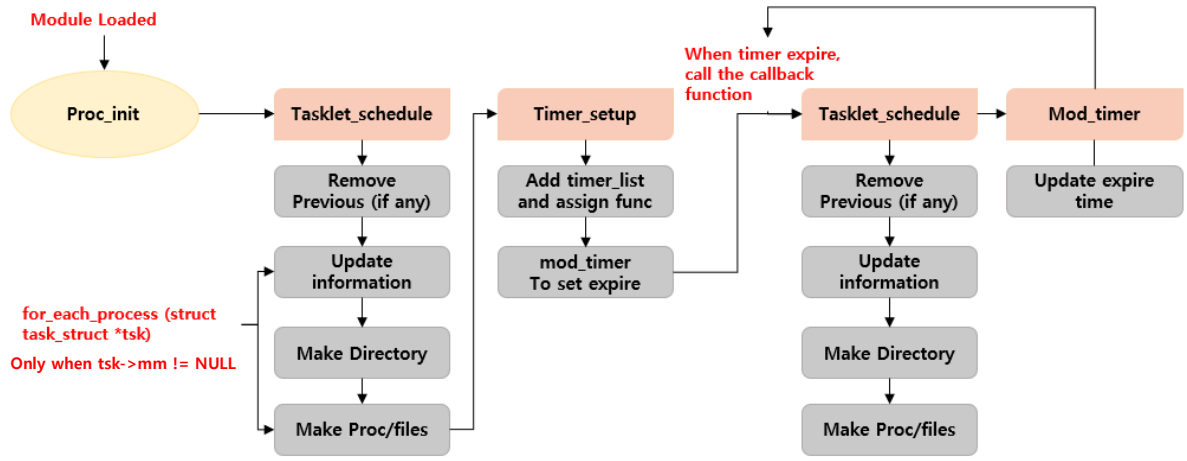


<Figure 20. Virtual Memory Structure>

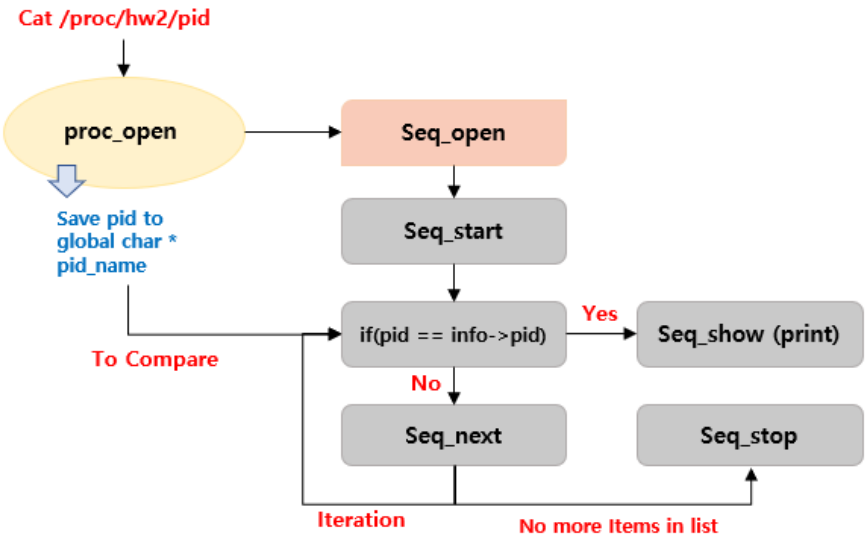
In the mm_show function., I implemented the given format to print the memory address data. General memory structure data is shown in figure20. As shown in the figure, the data contained in mm_struct allow module to approach certain part of memory. Also, mm->pgd has the base

address of Page Global Directory that is the value of er4 register. Also, for every page tables and directories, ***_offset can give the address of the entry and ***_val macro provide the value of entry. Moreover, the ***_flag macro allow module to obtain the flag parts that contained in the entry. For the easy use of flag, linux already defined the flag bits into literals with the name of flag. At last, the start physical address can be obtained by bit masking the 12LSB since the 12LSB of pte entry is a field of flag bits.

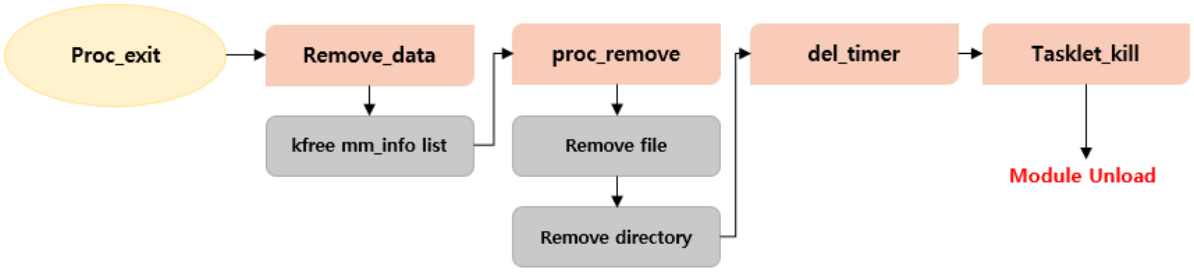
One of the most helpful kernel macro was for_each_process macro. It automatically iterates through all the tasks that is performed in current time period. However, since the macro iterates without distinguishing user process or kernel process, we must check whether mm_struct of task has reasonable value. The kernel process' mm_struct points to NULL. To use for_each_process macro, module must include linux/sched/signal.h file.



<Figure 21. Flow chart of when module is loaded>



<Figure 22. Flow chart of when cat /proc/hw2/pid called>



<Figure 23. Flow chart of when module unloaded>

5. Result & Errors

```
jwlee@2014190150:~/Desktop/hw2_2014190150/module$ sudo insmod hw2.ko period=5
[sudo] password for jwlee:
jwlee@2014190150:~/Desktop/hw2_2014190150/module$ ls /proc/hw2
1     1136  1178  1211  1224  1239  1279  1333  1445  1500  1527  1563  1582  1611  1622  1720  2142  404   4430  4614  597  612  640  648  718  869
1025  1139  1181  1212  1227  1245  1304  1335  1454  1504  1536  1570  1588  1612  1636  1729  2144  4129  4504  4644  598  617  641  649  729  891
1032  1145  1184  1213  1228  1247  1311  1339  1459  1506  1544  1575  1590  1615  1671  1735  2169  4135  4534  4656  599  621  642  650  750  901
1042  1157  1186  1217  1233  1251  1315  1342  1462  1511  1548  1576  1593  1617  1682  1749  2268  422   4553  548   602  624  643  651  775  908
1128  1164  1197  1219  1235  1256  1316  1437  1478  1514  1552  1578  1600  1618  1698  1761  2430  4403  4565  549   604  638  644  689  777  928
1131  1166  1204  1223  1236  1260  1329  1440  1487  1523  1561  1581  1610  1619  1716  1777  2470  4407  4586  591   607  639  645  692  787  930
```

<Figure 24. Load module of ls /proc>

As a result, the proc files for all the user process is successfully created. Each file will show the requested data of assignment. The possible error might be occurred when the update the information. The update cannot be exactly performed at specific period due to iterating tasks consume times.

The most time I consumed to implement is to create many proc files. Since I tried to update info and create proc file at once at loading module, kernel shocked frequently. However, by separating updating part and creation of files solved the problem. Also, at first, I tried to use mask to PGD value since it has more bits than I learned, however, deleted the masking part after read QnAs in website. Another problem I faced was how to get pid of user tried to see data. I was able to solve the problem by using file data structure that is used as parameter in proc_open. File name when we typed with cat command, was maintained at file->f_path.dentry->d_name.name. By comparing this file name with pid of info list, the module is able to select specific task that is requested. Obtaining system information is pretty straight forward in the Linux since developers made a lots macros and data structures for easier use. However, if one wants to use certain API, one must look through bunch of files that are inter-connected to each other. Moreover, such abstractions for portability caused more difficulty to analyze the kernel data structures and functions.

```
jwlee@2014190150:~/Desktop/hw2_2014190150/module$ cat /proc/hw2/1025
**********************************************************************
Virtual Memory Address Information
Process (    dbus-daemon:1025)
Last update time 6672384 ms
**********************************************************************
0x56239cf7a000 - 0x56239cfb13e8 : Code Area, 55 page(s)
0x56239d1b1a70 - 0x56239d1b33f0 : Data Area, 1 page(s)
0x56239d1b33f0 - 0x56239d3ca000 : BSS Area, 535 page(s)
0x56239d3ca000 - 0x56239d471000 : Heap Area, 167 page(s)
0x56239cf7a000 - 0x56239cfb2000 : Shared Libraries Area, 56 page(s)
0x7fff5cc69280 - 0x7fff5cc6ad12 : Stack Area, 1 page(s)
**********************************************************************
1 Level Paging: Page Global Directory Entry Information
**********************************************************************
PGD     Base Address            : 0xffffa040b2e2c000
code    PGD  Address            : 0xffffa040b2e2c560
        PGD  Value              : 0x8000000132fc7067
        +PFN Addrerss           : 0x00132fc7
        +Page Size              : 4KB
        +Accessed Bit           : 1
        +Cache Disable Bit      : true
        +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            : 0xffffa040b2fc7470
        PUD  Value              : 0x132f9b067
        +PFN Address            : 0x00132f9b
**********************************************************************
3 Level Paging: Page Middle Directory Entry Information
**********************************************************************
code    PMD  Address            : 0xffffa040b2f9b738
        PMD  Value              : 0x132f7b067
        +PFN Address            : 0x00132f7b
**********************************************************************
4 Level Paging: Page Table Entry Information
**********************************************************************
code    PTE  Address            : 0xffffa040b2f7bbd0
        PTE  Value              : 0x129163025
        +Page Base Address      : 0x00129163
        +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       :0x129163025000
**********************************************************************
```

<Figure 25. Result Page>

6. Reference

a. Lecture PPT

b. http://egloos.zum.com/dojeun/v/317480

c. http://bigs.egloos.com/

d. https://elixir.bootlin.com/linux/v5.0/

e. http://esos.hanyang.ac.kr/tc/2015gradproject2/i/entry/2 (figure 20)

f. http://forum.falinux.com/zbxe/index.php?mid=lecture_tip&search_target=nick_name&search_keyword=%EC%9D%B4%EC%9A%B0%EC%98%81&page=5&document_srl=556041

g. https://onecellboy.tistory.com/52 Tasklet

h. https://julrams.tistory.com/4

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

j. https://stackoverflow.com/questions/6252063/how-to-get-the-physical-address-from-the-logical-one-in-a-linux-kernel-module/6262158#6262158