CS 3502
Operating Systems

Project 3 Lab

Kun Suo

Computer Science, Kennesaw State University

https://kevinsuo.github.io/

Assignment 1

- Write a system call *sys_getmemInfo (*e.g., follow the steps in Project 1) to report statistics of a process's virtual address space. You can use *printk* in your system call and collect the information in *dmesg* command. The system call should take a process ID as input and outputs the following information about the process:

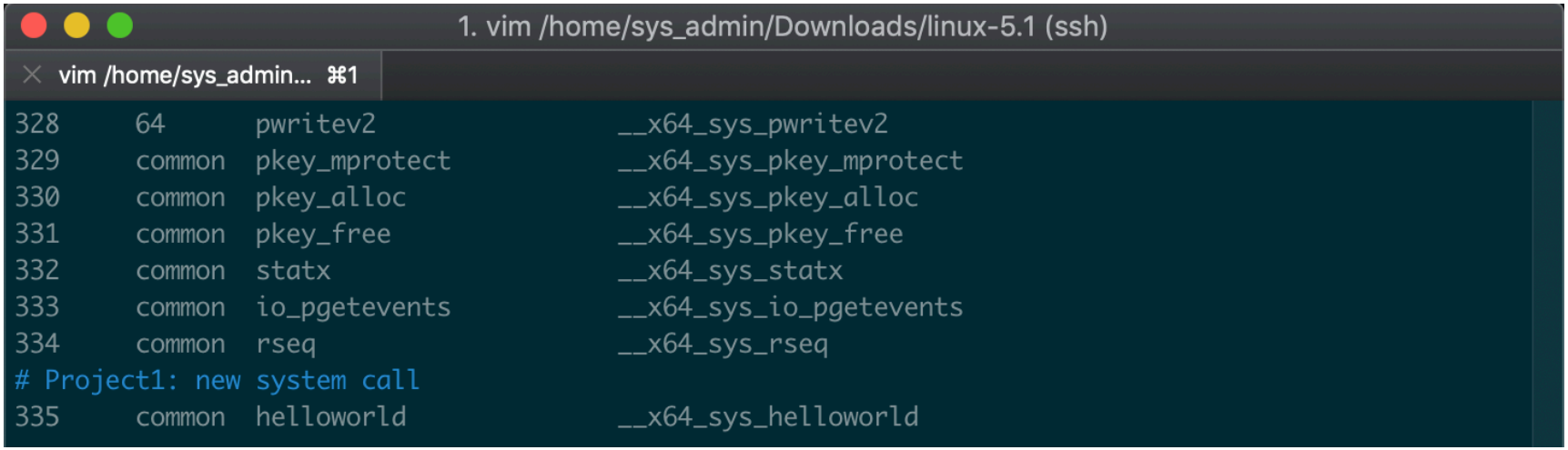
1. Each virtual memory area's access permissions.
2. The names of files mapped to these virtual memory areas.
3. The total size of the process's virtual address space.

Review how to add system calls

- Step 1: register your system call
- Step 2: declare your system call in the header file
- Step 3: implement your system call
- Step 4: write user level app to call it

Step 1: register your system call

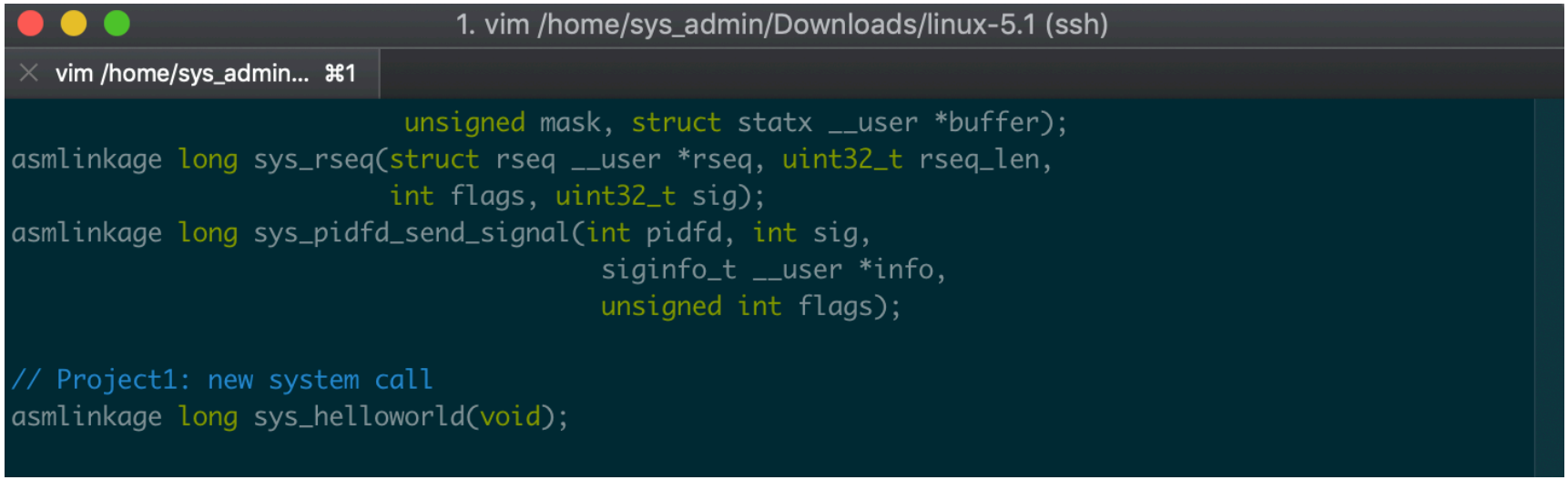
arch/x86/entry/syscalls/syscall_64.tbl



https://elixir.bootlin.com/linux/v5.0/source/arch/x86/entry/syscalls/syscall_64.tbl#L346

Step 2: declare your system call in the header file

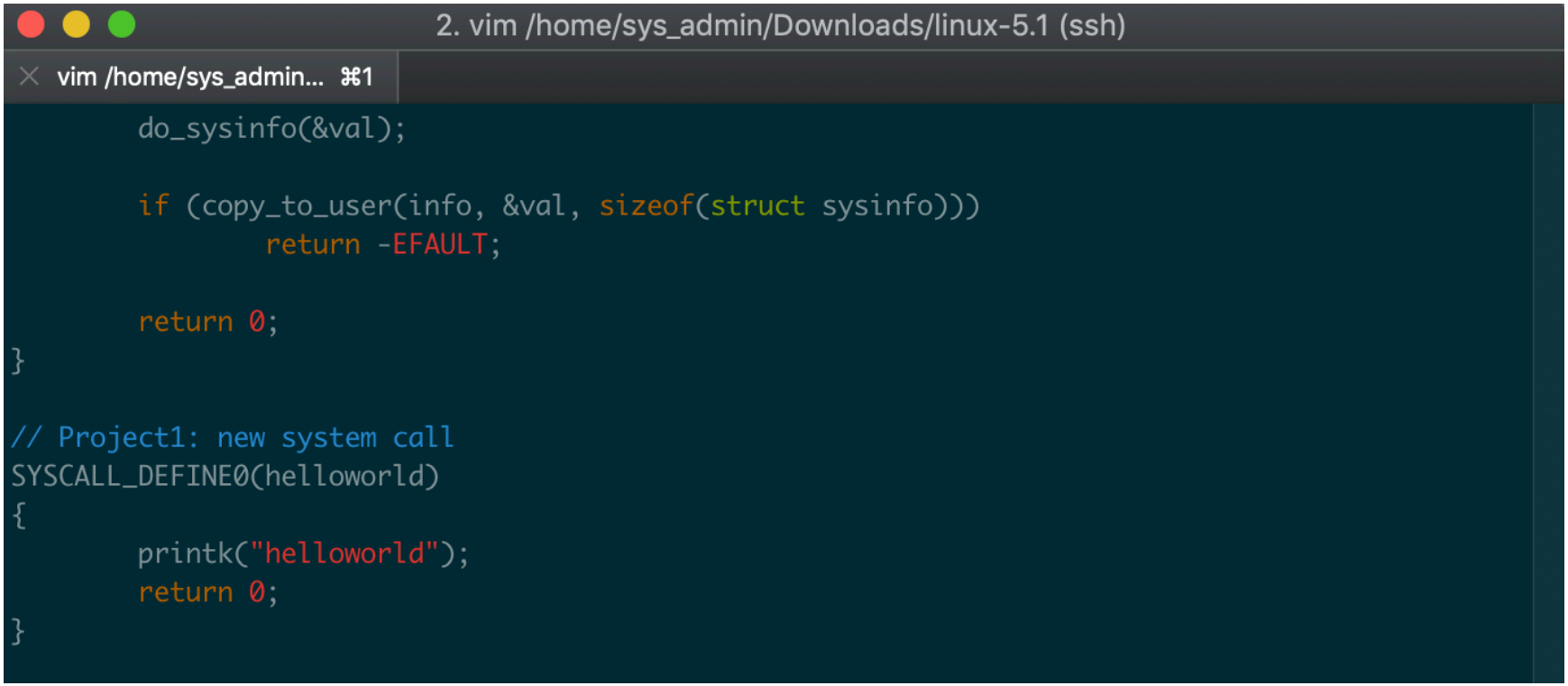
include/linux/syscalls.h



https://elixir.bootlin.com/linux/v5.0/source/include/linux/syscalls.h

Step 3: implement your system call

kernel/sys.c



https://elixir.bootlin.com/linux/v5.0/source/kernel/sys.c#L402

Step 4: write user level app to call it

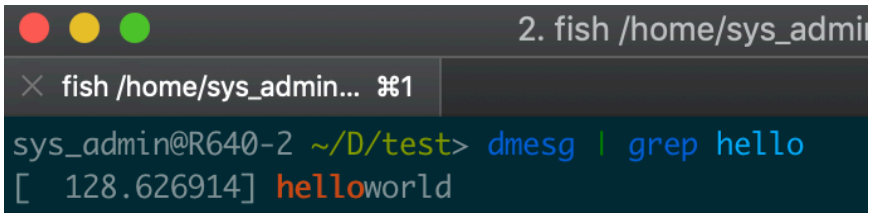
test_syscall.c：

```
****************************************
#include <linux/unistd .h>
#include <sys/syscall .h>
#include <sys/types .h>
#include <stdio .h>
#define __NR_helloworld 335

int main(int argc, char *argv[])
{
    syscall (__NR_helloworld) ;
    return 0 ;
}
****************************************

//If syscall needs parameter, then：
//syscall (__NR_helloworld，a，b，c) ;
```

Compile and execute：

```
$ gcc test_syscall.c -o test_syscall
$ ./test_syscall
```

The test program will call the new system call and output a helloworld message at the tail of the output of dmesg (system log).



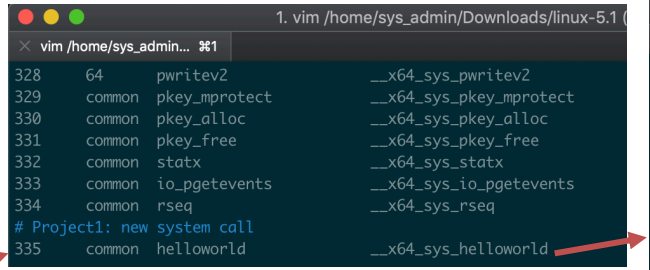
Put it all together

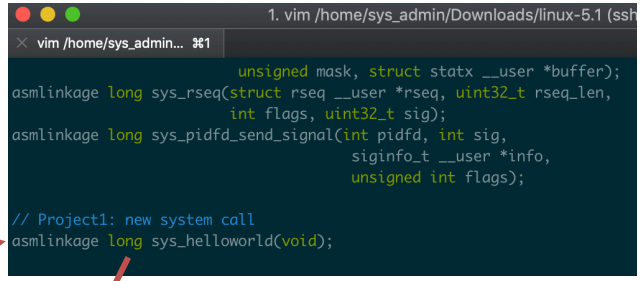
User space

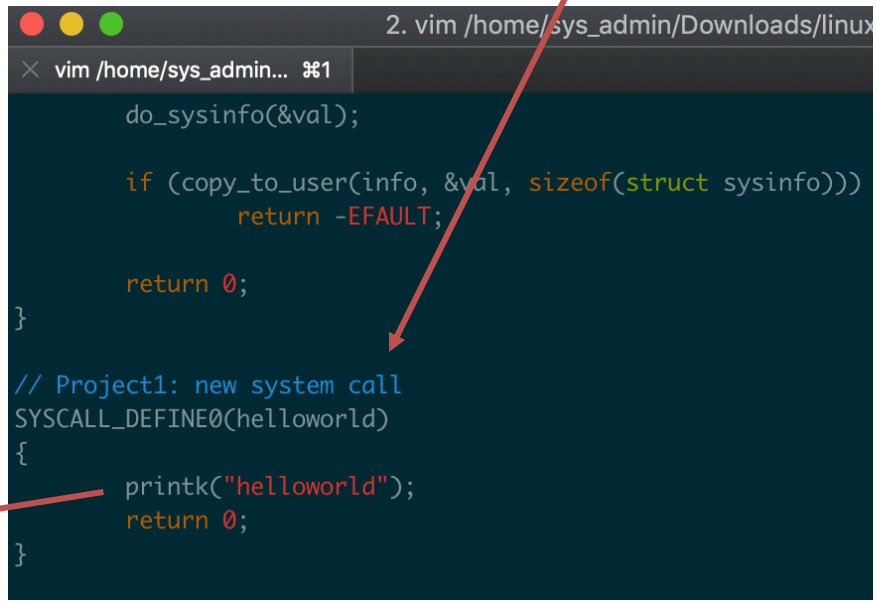
Kernel space

test_syscall.c:

```
**********************************************
#include <linux/unistd .h>
#include <sys/syscall .h>
#include <sys/types .h>
#include <stdio .h>
#define __NR_helloworld 335

int main(int argc, char *argv[])
{
    syscall (__NR_helloworld) ;
    return 0 ;
}
**********************************************
```







Assignment 1

- Write a system call *sys_getmemInfo (*e.g., follow the steps in Project 1) to report **statistics of a process's virtual address space**. You can use *printk* in your system call and collect the information in *dmesg* command. The system call should take a process ID as input and outputs the following information about the process:

1. Each virtual memory area's access permissions.

2. The names of files mapped to these virtual memory areas.

3. The total size of the process's virtual address space.

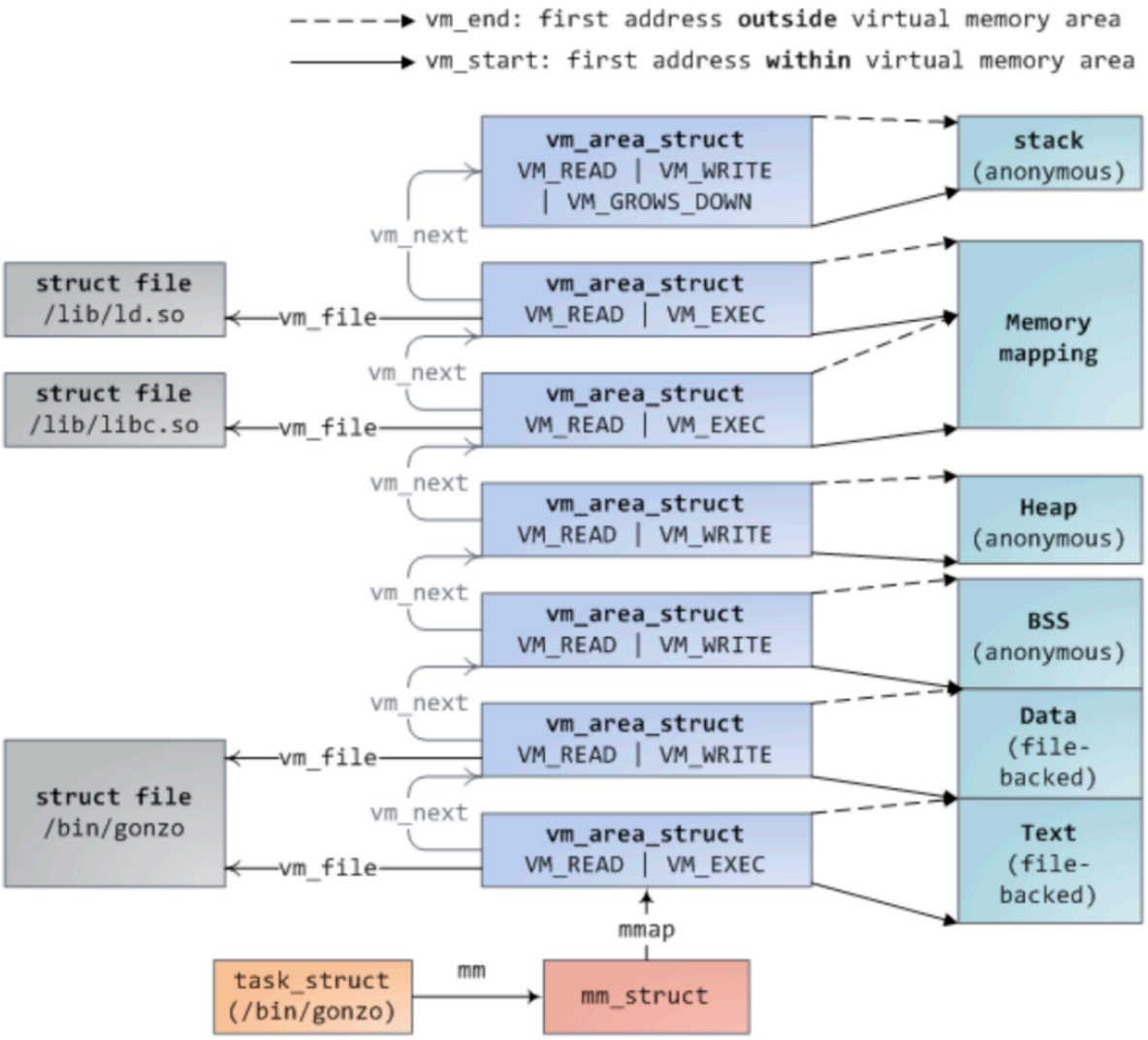
Revisit Process Data Structure

- Process Control Block (PCB)
  - State
  - Identifiers
  - Scheduling info
  - File system
  - Virtual memory
  - Process specific context
  - …

task_struct
https://elixir.bootlin.com/linux/latest/source/include/linux/sched.h#L637

https://elixir.bootlin.com/linux/latest/source/include/linux/sched.h#L740

Revisit Process Data Structure

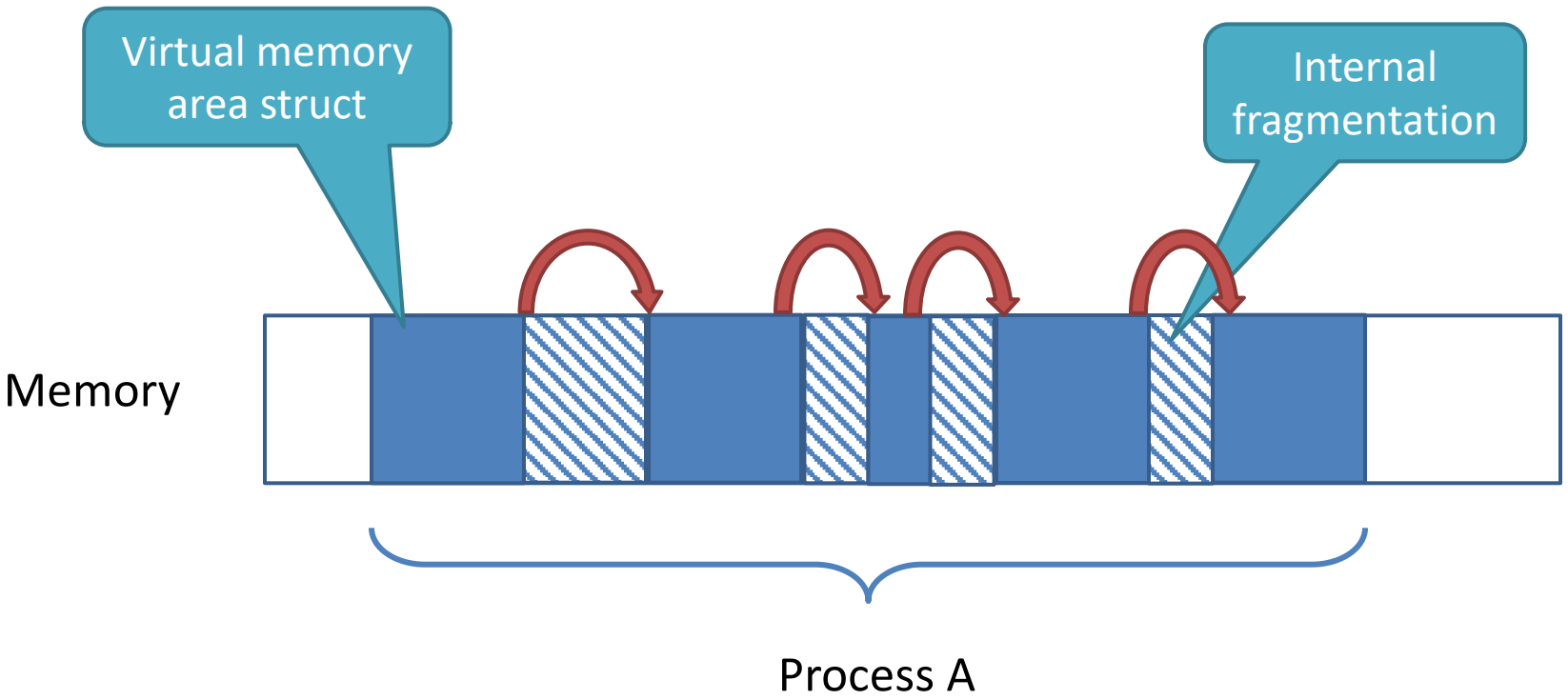


Revisit Process Data Structure

```
struct task_struct {
#ifdef CONFIG_THREAD_INFO_IN_TASK
        /*
         * For reasons of header soup (see current_thr
         * must be the first element of task_struct.
         */
        struct thread_info              thread_info;
#endif
        /* -1 unrunnable, 0 runnable, >0 stopped: */
        volatile long                   state;


        struct sched_info               sched_info;

        struct list_head                tasks;
#ifdef CONFIG_SMP
        struct plist_node               pushable_tasks;
        struct rb_node                  pushable_dl_tasks;
#endif

        struct mm_struct                *mm;
        struct mm_struct                *active_mm;
```

```
struct mm_struct {
        struct vm_area_struct *mmap;
        struct rb_root mm_rb;
        u32 vmacache_seqnum;
#ifdef CONFIG_MMU
        unsigned long (*get_unmapped_area) (st
                                    unsigned long
                                    unsigned long
#endif
        unsigned long mmap_base;
        unsigned long mmap_legacy_base;
        unsigned long task_size;
        unsigned long highest_vm_end;
        pgd_t * pgd;
        atomic_t mm_users;
        atomic_t mm_count;
        atomic_long_t nr_ptes;
#if CONFIG_PGTABLE_LEVELS > 2
```

Revisit Process Data Structure



Revisit Process Data Structure

```
struct mm_struct {
        struct vm_area_struct *mmap;
        struct rb_root mm_rb;
        u32 vmacache_seqnum;
#ifdef CONFIG_MMU
        unsigned long (*get_unmapped_area) (st
                                unsigned long
                                unsigned long
#endif
        unsigned long mmap_base;
        unsigned long mmap_legacy_base;
        unsigned long task_size;
        unsigned long highest_vm_end;
        pgd_t * pgd;
        atomic_t mm_users;
        atomic_t mm_count;
        atomic_long_t nr_ptes;
#if CONFIG_PGTABLE_LEVELS > 2
```

```
struct vm_area_struct {
        /* The first cache line has the info for VMA tree walking. */

        unsigned long vm_start;         /* Our start address within vm_mm. */
        unsigned long vm_end;           /* The first byte after our end address
                                           within vm_mm. */

        /* linked list of VM areas per task, sorted by address */
        struct vm_area_struct *vm_next, *vm_prev;

        struct rb_node vm_rb;

        /*
         * Largest free memory gap in bytes to the left of this VMA.
         * Either between this VMA and vma->vm_prev, or between one of the
         * VMAs below us in the VMA rbtree and its ->vm_prev. This helps
         * get_unmapped_area find a free area of the right size.
         */
        unsigned long rb_subtree_gap;

        /* Second cache line starts here. */

        struct mm_struct *vm_mm;        /* The address space we belong to. */
        pgprot_t vm_page_prot;          /* Access permissions of this VMA. */
        unsigned long vm_flags;         /* Flags, see mm.h. */
```

1. Each virtual memory area's access permissions

```
struct vm_area_struct {
        /* The first cache line has the info for VMA tree walking. */

        unsigned long vm_start;         /* Our start address within vm_mm. */
        unsigned long vm_end;           /* The first byte after our end address
                                           within vm_mm. */

        /* linked list of VM areas per task, sorted by address */
        struct vm_area_struct *vm_next, *vm_prev;

        struct rb_node vm_rb;

        /*
         * Largest free memory gap in bytes to the left of this VMA.
         * Either between this VMA and vma->vm_prev, or between one of the
         * VMAs below us in the VMA rbtree and its ->vm_prev. This helps
         * get_unmapped_area find a free area of the right size.
         */
        unsigned long rb_subtree_gap;

        /* Second cache line starts here. */

        struct mm_struct *vm_mm;        /* The address space we belong to. */
        pgprot_t vm_page_prot;          /* Access permissions of this VMA. */
        unsigned long vm_flags;         /* Flags, see mm.h. */
```

```
typedef struct pgprot { pgprotval_t pgprot; } pgprot_t;
```

```
typedef unsigned long   pgprotval_t;
```

printk("%d", permission);

printk("%lu", permission);

2. The names of files mapped to these virtual memory areas

```
struct vm_area_struct {
        /* The first cache line has the info for VMA tree walking. */

        unsigned long vm_start;         /* Our start address within vm_mm. */
        unsigned long vm_end;           /* The first byte after our end address
                                           within vm_mm. */

        /* linked list of VM areas per task, sorted by address */
        struct vm_area_struct *vm_next, *vm_prev;

        struct rb_node vm_rb;



        /* Information about our backing store: */
        unsigned long vm_pgoff;         /* Offset (within vm_file) in PAGE_SIZE
                                           units, *not* PAGE_CACHE_SIZE */
        struct file * vm_file;          /* File we map to (can be NULL). */
        void * vm_private_data;         /* was vm_pte (shared mem) */

#ifndef CONFIG_MMU
        struct vm_region *vm_region;    /* NOMMU mapping region */
#endif
#ifdef CONFIG_NUMA
        struct mempolicy *vm_policy;    /* NUMA policy for the VMA */
#endif
};
```

```
struct file {
        union {
                struct llist_node       fu_llist;
                struct rcu_head         fu_rcuhead;
        } f_u;
        struct path             f_path;
        struct inode            *f_inode;       /*
```

```
struct path {
        struct vfsmount *mnt;
        struct dentry *dentry;
};
```

```
struct dentry {
        /* RCU lookup touched fields */
        unsigned int d_flags;
        seqcount_t d_seq;
        struct hlist_bl_node d_hash;
        struct dentry *d_parent;
        struct qstr d_name;
        struct inode *d_inode;
```

```
struct qstr {
        union {
                struct {
                        HASH_LEN_DECLARE;
                };
                u64 hash_len;
        };
        const unsigned char *name;
};
```

```
printk("%s", vas->vm_file->
        f_path.dentry->d_name.name);
```

2. The names of files mapped to these virtual memory areas

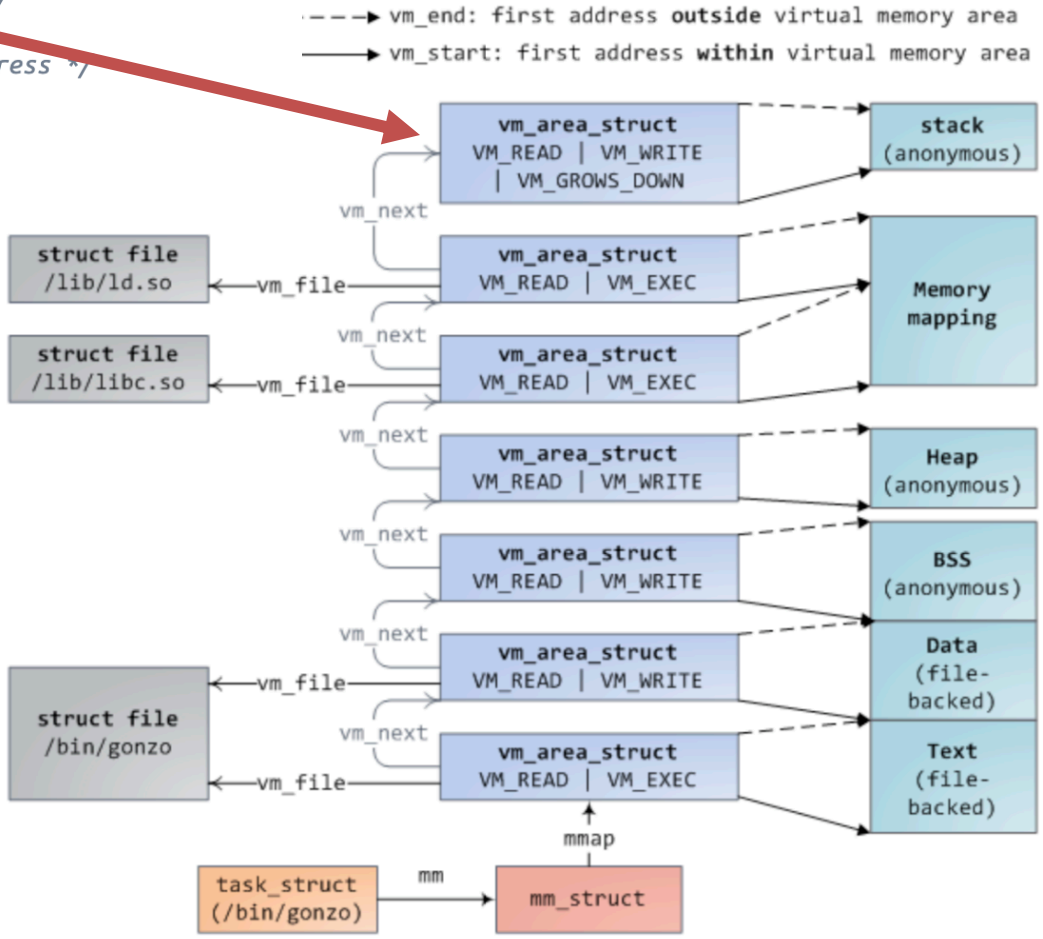
```
struct vm_area_struct {
        /* The first cache line has the info for VMA tree walking. */

        unsigned long vm_start;         /* Our start address within vm_mm. */
        unsigned long vm_end;           /* The first byte after our end address
                                           within vm_mm. */

        /* linked list of VM areas per task, sorted by address */
        struct vm_area_struct *vm_next, *vm_prev;

        struct rb_node vm_rb;



        /* Information about our backing store: */
        unsigned long vm_pgoff;         /* Offset (within vm_file) in PAGE_SIZE
                                           units, *not* PAGE_CACHE_SIZE */
        struct file * vm_file;          /* File we map to (can be NULL). */
        void * vm_private_data;         /* was vm_pte (shared mem) */

#ifndef CONFIG_MMU
        struct vm_region *vm_region;    /
#endif
#ifdef CONFIG_NUMA
        struct mempolicy *vm_policy;    /* NUMA policy for the VMA */
#endif
};
```

It could be null!

```
fs = vas->vm_file;
if(fs != NULL){
        printk("The file name mapped to
        this vma is %s\n", fs-
        >f_path.dentry->d_name.name);
} else {
        printk("The file name mapped to
        this vma is NULL\n");
}
```

3. The total size of the process's virtual address space

```
struct vm_area_struct {
        /* The first cache line has the info for VMA tree walking. */

        unsigned long vm_start;         /* Our start address within vm_mm. */
        unsigned long vm_end;           /* The first byte after our end address
                                           within vm_mm. */

        /* linked list of VM areas per task, sorted by address */
        struct vm_area_struct *vm_next, *vm_prev;

        struct rb_node vm_rb;
```



Assignment 1

- Write a system call *sys_getmemInfo (*e.g., follow the steps in Project 1) to report statistics of a process's virtual address space. You can use *printk* in your system call and collect the information in *dmesg* command.

- Input: pid
- Output:
  - Each virtual memory area's access permissions.
  - The names of files mapped to these virtual memory areas.
  - The total size of the process's virtual address space.

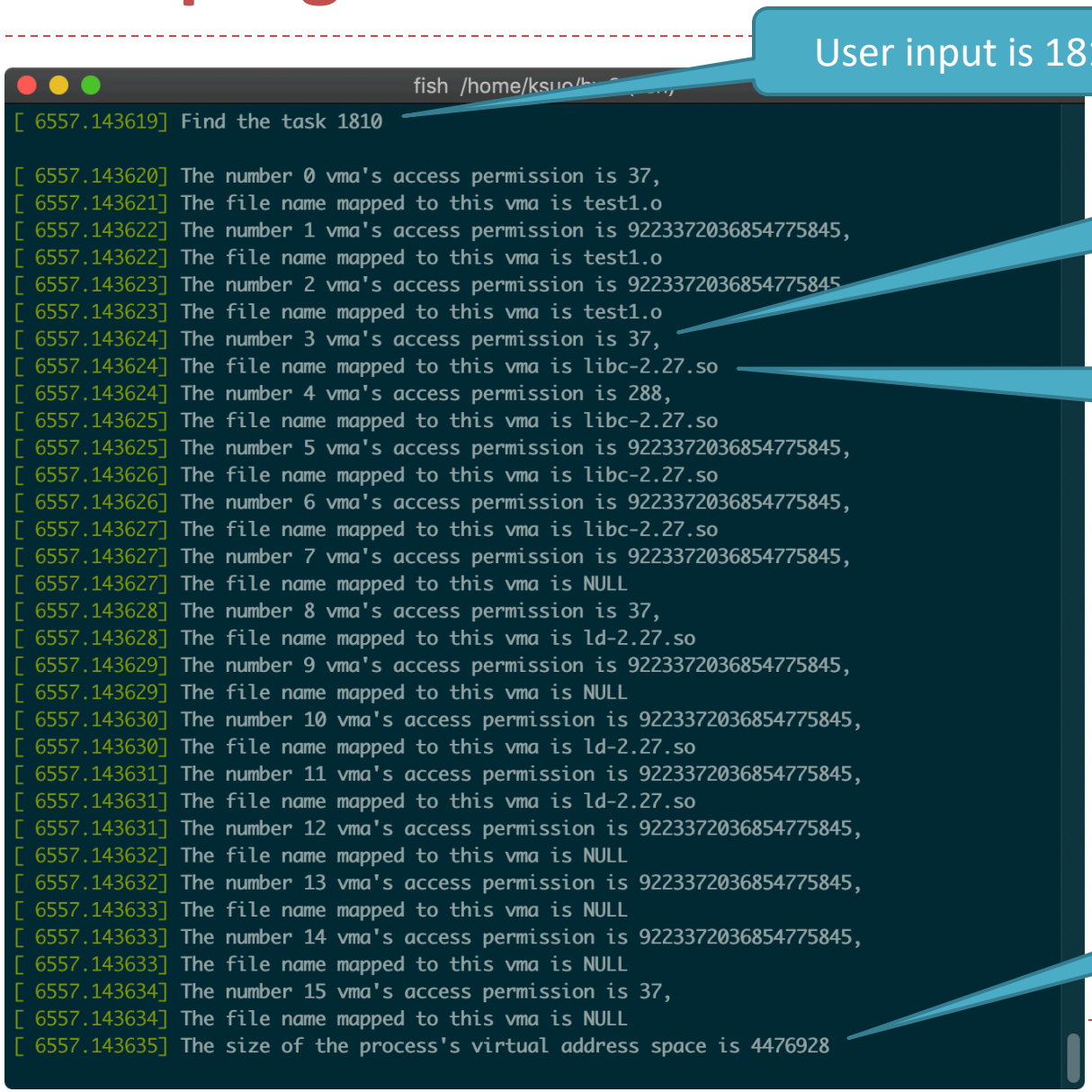
Assignment 1: user test program 1

```c
#include <linux/unistd.h>
#include <sys/syscall.h>
#include <sys/types.h>
#include <stdio.h>
#include <unistd.h>

#define __NR_getmemInfo 338


int main(int argc, char *argv[])
{
    int no = getpid();
    syscall (__NR_getmemInfo, no);
    return 0;
}
```

System call getpid() will return the pid of current process.

- One test program just calls the new system call and report the calling process's statistics.

Assignment 1: expected output of user test program 1



Assignment 1: user test program 2

```c
#include <stdio.h>
#include <pthread.h>
#include <sys/types.h>
#include <sys/syscall.h>
#include <unistd.h>

#define __NR_getmemInfo 338
#define NUM_THREADS 3

void *myThread(void *threadid)
{
    int no = syscall(SYS_gettid);
    printf("This is a pthread. Pid is %d\n", no);
    syscall(__NR_getmemInfo, no);
}


int main(int argc, char *argv[])
{
    pthread_t threads[NUM_THREADS];
    int t, rc;

    printf("This is a process. Pid is %d\n", getpid());

    for(t=0; t<NUM_THREADS; t++){
        rc = pthread_create(&threads[t], NULL, (void*)myThread, NULL);

        if(rc){
            printf("Error\n");
            return 1;
        }
    }

    for(t=0; t<NUM_THREADS; t++){
        pthread_join(threads[t], NULL);
    }

    printf("%s", "program is over.\n");
    pthread_exit(NULL);
    return 0;
}
```

System call SYS_gettid() returns the pid of each thread

- The other test program should create multiple threads and report information about individual threads.

Assignment 1: expected output of user test program 2

```
ksuo@ksuo-VirtualBox ~/hw3> ./test2.o
This is a process. Pid is 2007
This is a pthread. Pid is 2008
This is a pthread. Pid is 2010
This is a pthread. Pid is 2009
program is over.
```

User side

Kernel dmesg

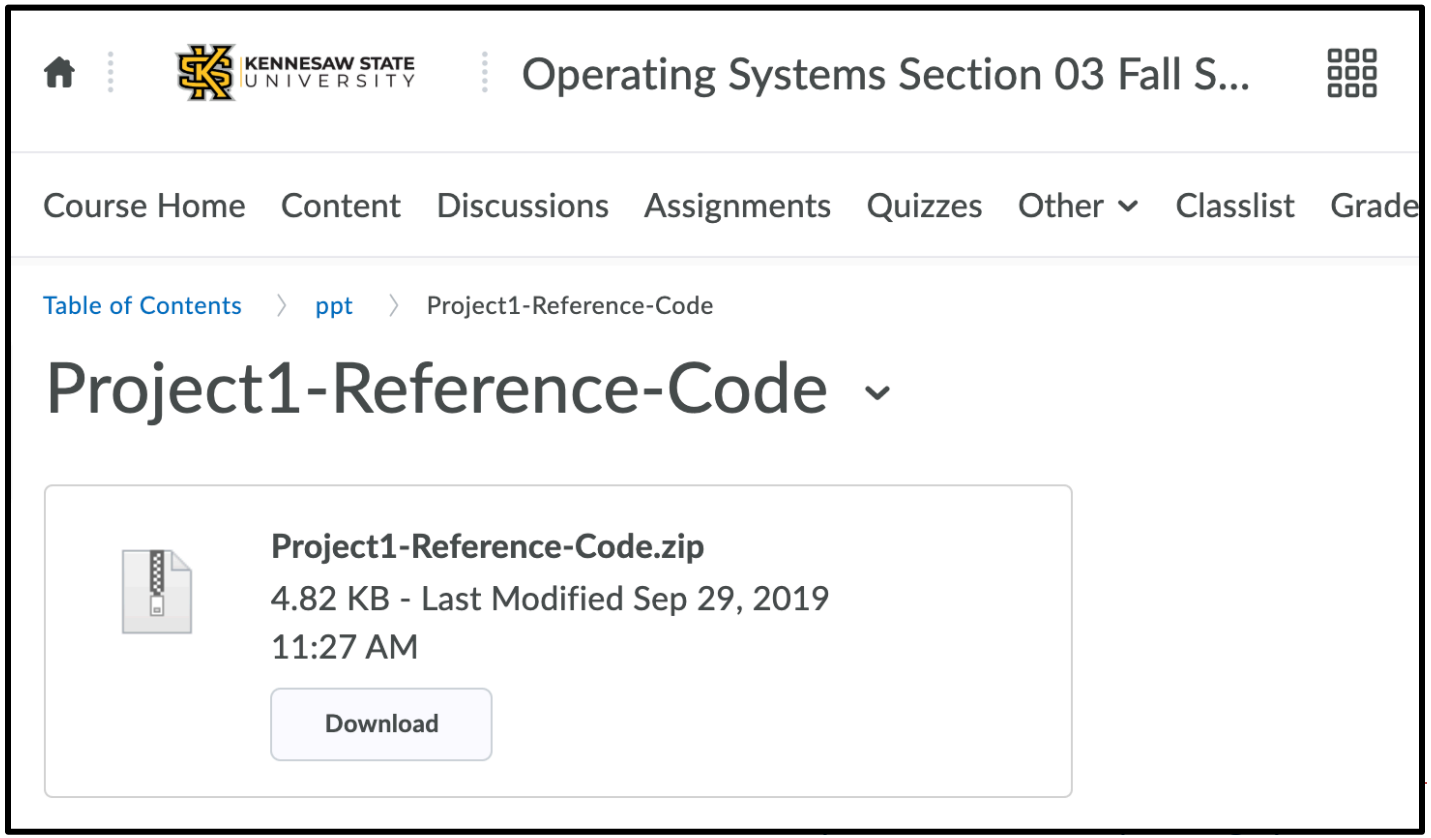
```
fish  /home/ksuo (ssh)
[ 6859.138227] Find the task 2009
[ 6859.138229] The nur
[ 6859.138230] The fil
[ 6859.138231] The nur
[ 6859.138231] The fil
[ 6859.138232] The nur
[ 6859.138232] The fil
[ 6859.138233] The nur
[ 6859.138234] The fil
[ 6859.138234] The nur
[ 6859.138235] The fil
[ 6859.138235] The nur
[ 6859.138236] The fil
[ 6859.138236] The nur
[ 6859.138237] The fil
[ 6859.138237] The nur
[ 6859.138238] The fil
[ 6859.138238] The nur
[ 6859.138239] The fil
[ 6859.138240] The nur
[ 6859.138240] The fil
[ 6859.138241] The nur
[ 6859.138241] The fil
[ 6859.138242] The nur
[ 6859.138242] The fil
[ 6859.138243] The nur
[ 6859.138244] The fil
[ 6859.138244] The nur
[ 6859.138245] The fil
[ 6859.138245] The nur
[ 6859.138245] The fil
[ 6859.138246] The nur
[ 6859.138246] The fil
[ 6859.138247] The nur
[ 6859.138247] The fil
[ 6859.138248] The nur
[ 6859.138322] The fil
[ 6859.138323] The nur
[ 6859.138323] The fil
[ 6859.138324] The nur
[ 6859.138324] The fil
[ 6859.138325] The nur
[ 6859.138325] The fil
```

```
fish /ho  (ssh)  ⌥⌘2
[ 6859.138085] Find the task 2010
[ 6859.138086] The nu
[ 6859.138087] The fi
[ 6859.138087] The nu
[ 6859.138087] The fi
[ 6859.138088] The nu
[ 6859.138088] The fi
[ 6859.138089] The nu
[ 6859.138089] The fi
[ 6859.138090] The nu
[ 6859.138090] The fi
[ 6859.138090] The nu
[ 6859.138091] The fi
[ 6859.138091] The nu
[ 6859.138091] The fi
[ 6859.138092] The nu
[ 6859.138092] The fi
[ 6859.138093] The nu
[ 6859.138093] The fi
[ 6859.138093] The nu
[ 6859.138094] The fi
[ 6859.138094] The nu
[ 6859.138095] The fi
[ 6859.138095] The nu
[ 6859.138096] The fi
[ 6859.138096] The nu
[ 6859.138096] The fi
[ 6859.138097] The nu
[ 6859.138097] The fi
[ 6859.138098] The nu
[ 6859.138098] The fi
[ 6859.138099] The nu
[ 6859.138099] The fi
[ 6859.138099] The nu
[ 6859.138100] The fi
[ 6859.138100] The nu
[ 6859.138101] The fi
[ 6859.138101] The nu
[ 6859.138102] The fi
[ 6859.138102] The nu
[ 6859.138102] The fi
[ 6859.138103] The nu
[ 6859.138103] The fi
```

```
fish  /home/ksuo (ssh)  ⌥⌘2
[ 6859.137954] Find the task 2008
[ 6859.137955] The number 0 vma's access permission is 37,
[ 6859.137956] The file name mapped to this vma is test2.o
[ 6859.137957] The number 1 vma's access permission is 9223372036854775845,
[ 6859.137957] The file name mapped to this vma is test2.o
[ 6859.137957] The number 2 vma's access permission is 9223372036854775845,
[ 6859.137958] The file name mapped to this vma is test2.o
[ 6859.137958] The number 3 vma's access permission is 9223372036854775845,
[ 6859.137959] The file name mapped to this vma is NULL
[ 6859.137959] The number 4 vma's access permission is 288,
[ 6859.137960] The file name mapped to this vma is NULL
[ 6859.137960] The number 5 vma's access permission is 9223372036854775845,
[ 6859.137960] The file name mapped to this vma is NULL
[ 6859.137961] The number 6 vma's access permission is 288,
[ 6859.137961] The file name mapped to this vma is NULL
[ 6859.137962] The number 7 vma's access permission is 9223372036854775845,
[ 6859.137962] The file name mapped to this vma is NULL
[ 6859.137962] The number 8 vma's access permission is 37,
[ 6859.137963] The file name mapped to this vma is libc-2.27.so
[ 6859.137963] The number 9 vma's access permission is 288,
[ 6859.137964] The file name mapped to this vma is libc-2.27.so
[ 6859.137964] The number 10 vma's access permission is 9223372036854775845,
[ 6859.137965] The file name mapped to this vma is libc-2.27.so
[ 6859.137965] The number 11 vma's access permission is 9223372036854775845,
[ 6859.137966] The file name mapped to this vma is libc-2.27.so
[ 6859.137966] The number 12 vma's access permission is 9223372036854775845,
[ 6859.137966] The file name mapped to this vma is NULL
[ 6859.137967] The number 13 vma's access permission is 37,
[ 6859.137967] The file name mapped to this vma is libpthread-2.27.so
[ 6859.137968] The number 14 vma's access permission is 288,
[ 6859.137968] The file name mapped to this vma is libpthread-2.27.so
[ 6859.137969] The number 15 vma's access permission is 9223372036854775845,
[ 6859.137969] The file name mapped to this vma is libpthread-2.27.so
[ 6859.137970] The number 16 vma's access permission is 9223372036854775845,
[ 6859.137970] The file name mapped to this vma is libpthread-2.27.so
[ 6859.137971] The number 17 vma's access permission is 9223372036854775845,
[ 6859.137971] The file name mapped to this vma is NULL
[ 6859.137971] The number 18 vma's access permission is 37,
[ 6859.137972] The file name mapped to this vma is ld-2.27.so
[ 6859.137972] The number 19 vma's access permission is 9223372036854775845,
[ 6859.137973] The file name mapped to this vma is NULL
[ 6859.137973] The number 20 vma's access permission is 9223372036854775845,
[ 6859.137973] The file name mapped to this vma is ld-2.27.so
```

Reference

- If you forget how to print out process related information, please review our project 1



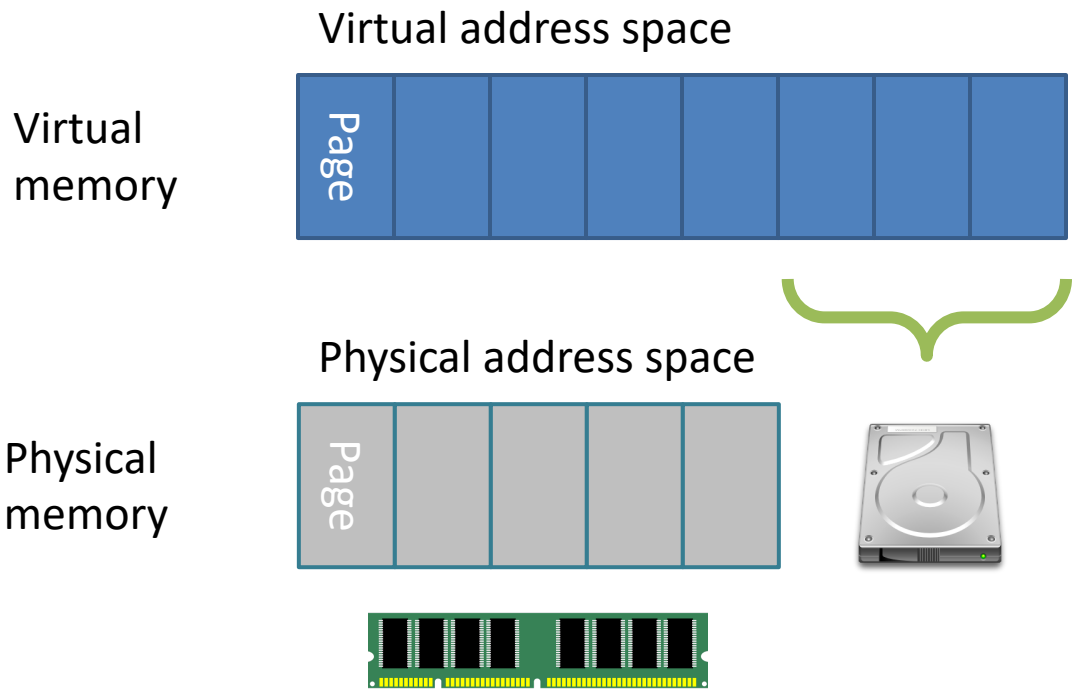
Assignment 2

- Given the above virtual memory areas used by a process, write a system call *sys_getPageInfo* to report the current status of specific addresses in these virtual memory areas. The system call takes the pid of a process as input and outputs the following information of start address *vm_start* in each *vm_area_struct* that the process has:

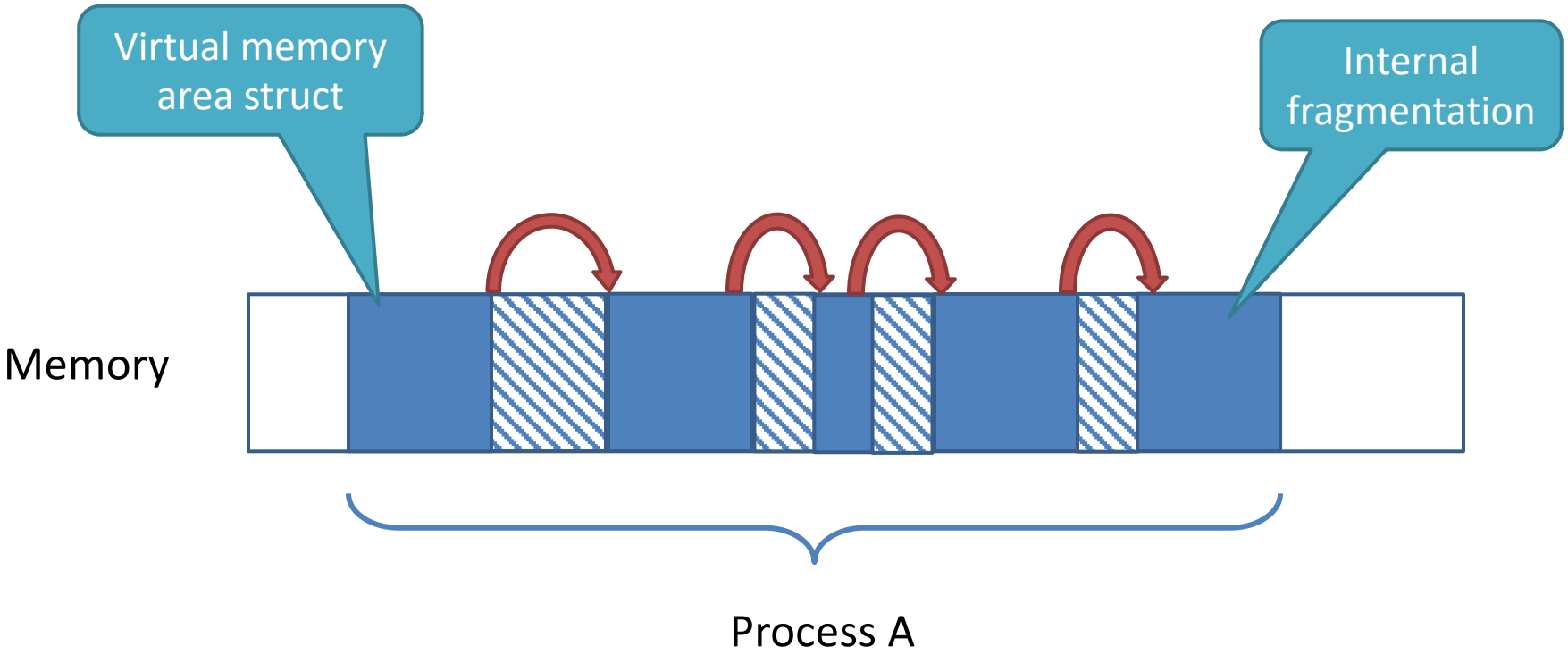
1. If the data in this address is in memory or on disk.
2. If the page which this address belongs to has been referenced or not.
3. If the page which this address belongs to is dirty or not.

Assignment 2

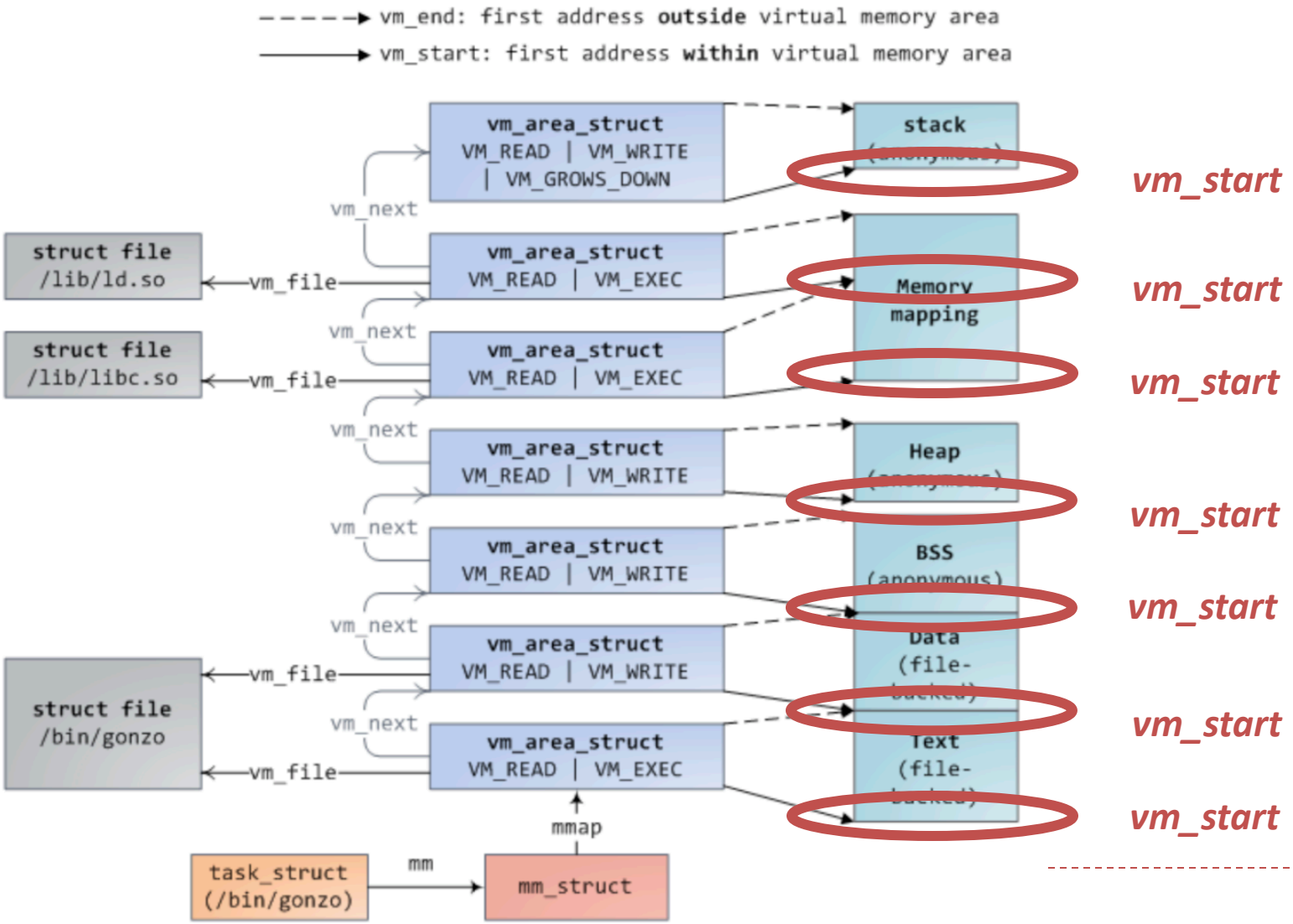
- Virtual memory address could be in memory or on disk.



Revisit Process Data Structure



Assignment 2



Assignment 2

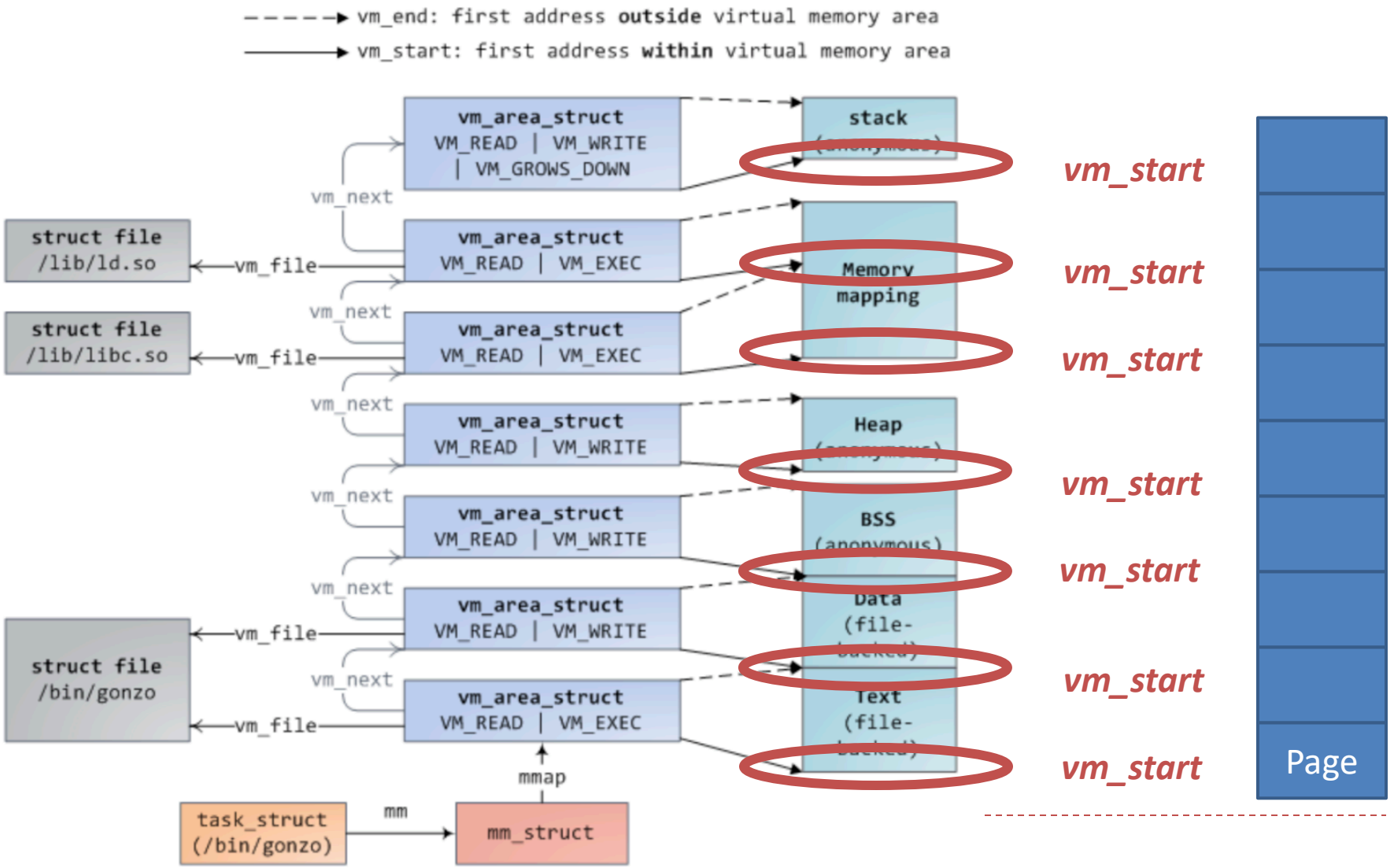
- Given the above virtual memory areas used by a process, write a system call *sys_getPageInfo* to report the current status of specific addresses in these virtual memory areas. The system call takes the **pid of a process as input** and **outputs** the following information of **start address *vm_start* in each *vm_area_struct*** that the process has:

1. If the data in this address is in memory or on disk.

2. If the page which this address belongs to has been referenced or not.

3. If the page which this address belongs to is dirty or not.

Assignment 2



Assignment 2

Table: Page Table Entry Status Bits

Bit	Function
_PAGE_PRESENT	Page is resident in memory and not swapped out.
_PAGE_ACCESSED	Set if the page is referenced.
_PAGE_DIRTY	Set if the page is written to.

```
static inline int pte_present(pte_t a)
{
        return pte_flags(a) & (_PAGE_PRESENT | _PAGE_PROTNONE);
}

static inline int pte_young(pte_t pte)
{
        return pte_flags(pte) & _PAGE_ACCESSED;
}

static inline int pte_dirty(pte_t pte)
{
        return pte_flags(pte) & _PAGE_DIRTY;
}
```

https://elixir.bootlin.com/linux/latest/source/arch/x86/include/asm/pgtable.h

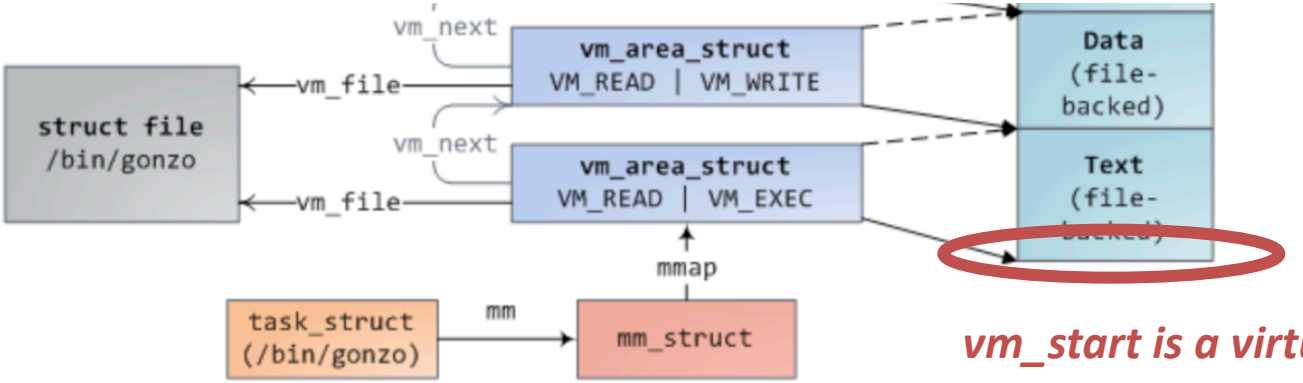
Assignment 2

Table: Page Table Entry Status Bits

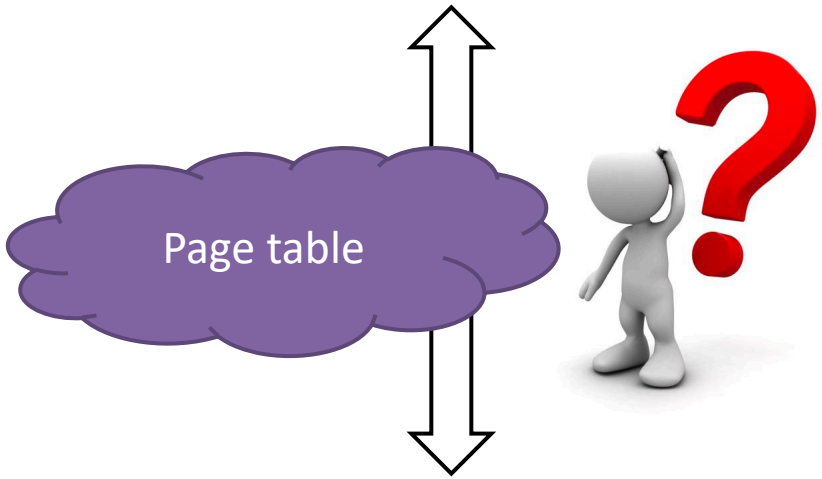
Bit	Function
_PAGE_PRESENT	Page is resident in memory and not swapped out.
_PAGE_ACCESSED	Set if the page is referenced.
_PAGE_DIRTY	Set if the page is written to.

- The function *pte_present* is used for judging a page is in memory or disk. If the return value is 1, the page is in the memory, otherwise it is in the disk.
- The function *pte_young* is used for judging a page is referenced or not. If the return value is 1, the page has been referenced, otherwise it has not been referenced yet.
- The function *pte_dirty* is used for judging a page is dirty or not. If the return value is 1, the page is dirty, otherwise it is not dirty.

Assignment 2



vm_start is a virtual memory address

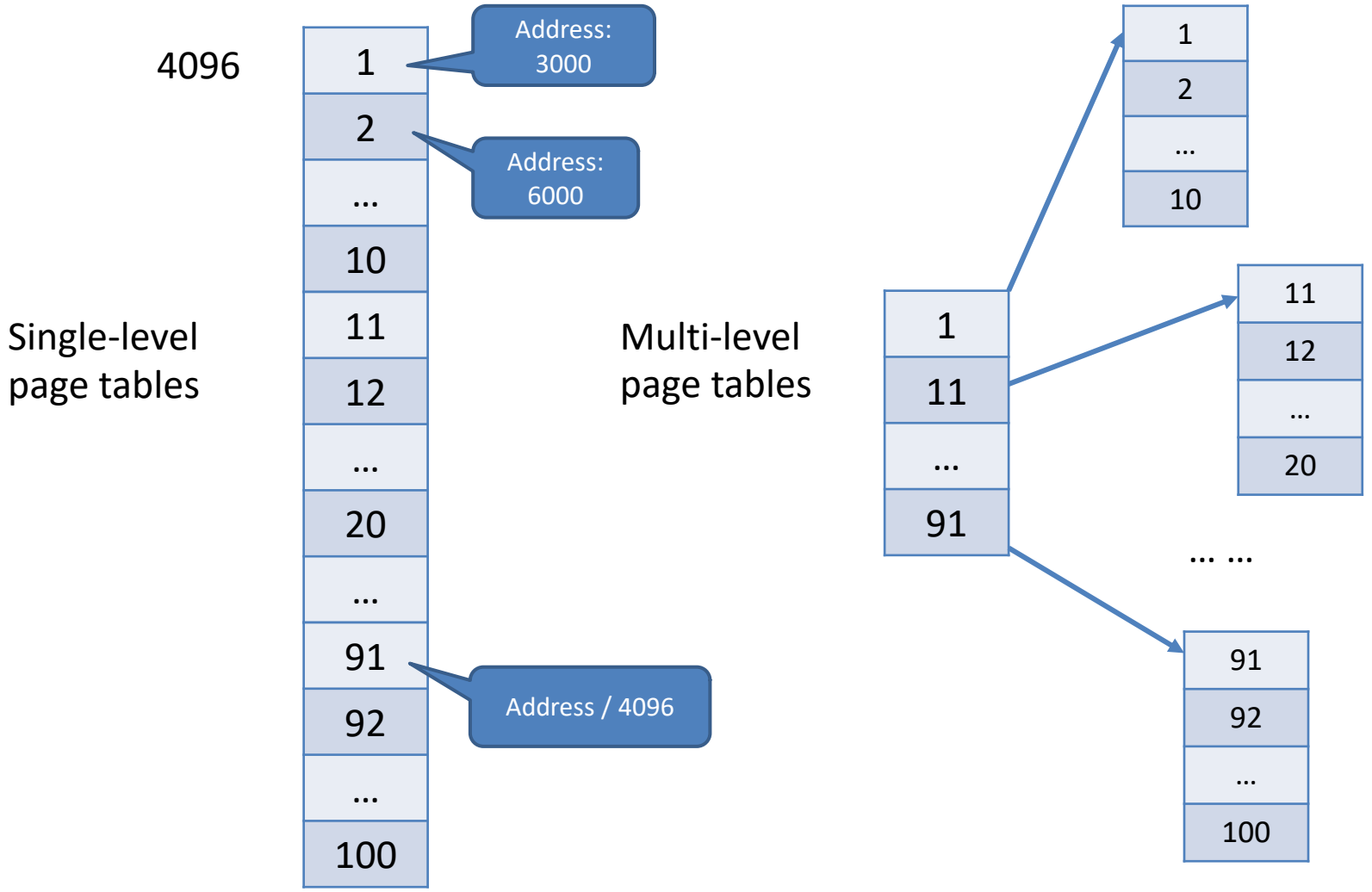



```
static inline int pte_present(pte_t a)
{
        return pte_flags(a) & (_PAGE_PRESENT | _
}

static inline int pte_young(pte_t pte)
{
        return pte_flags(pte) & _PAGE_ACCESSED;
}

static inline int pte_dirty(pte_t pte)
{
        return pte_flags(pte) & _PAGE_DIRTY;
}
```

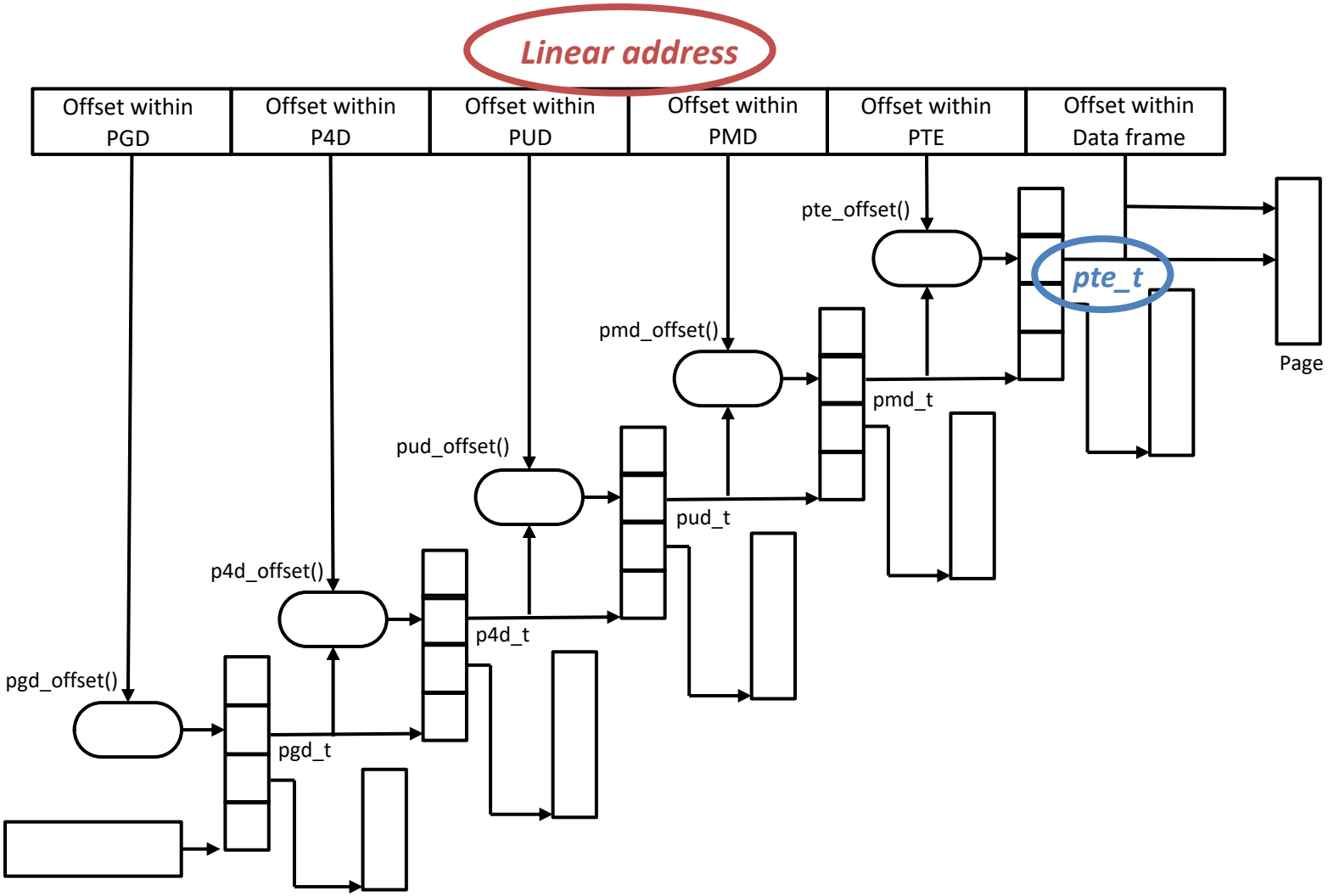
```
typedef struct { pteval_t pte; } pte_t;
```

pte_t is a memory page entry

Assignment 2: revisit memory address



Assignment 2: revisit memory address



Virtual address

```c
/*
 * a shortcut to get a pgd_t in a given mm
 */
#define pgd_offset(mm, address) pgd_offset_pgd((mm)->pgd, (address))
```

```c
/* to find an entry in a page-table-directory. */
static inline p4d_t *p4d_offset(pgd_t *pgd, unsigned long address)
{
        if (!pgtable_l5_enabled())
                return (p4d_t *)pgd;
        return (p4d_t *)pgd_page_vaddr(*pgd) + p4d_index(address);
}
```

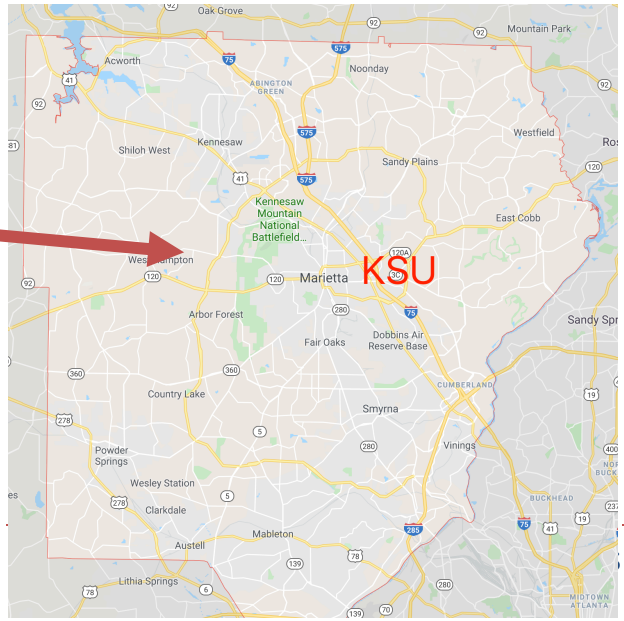
https://elixir.bootlin.com/linux/latest/source/arch/x86/include/asm/pgtable.h

```c
/* 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);
}
```

```c
/* Find an entry in the second-level page table.. */
static inline pmd_t *pmd_offset(pud_t *pud, unsigned long address)
{
        return (pmd_t *)pud_page_vaddr(*pud) + pmd_index(address);
}
```

```c
static inline pte_t *pte_offset_kernel(pmd_t *pmd, unsigned long address)
{
        return (pte_t *)pmd_page_vaddr(*pmd) + pte_index(address);
}
```

Just like we search on maps



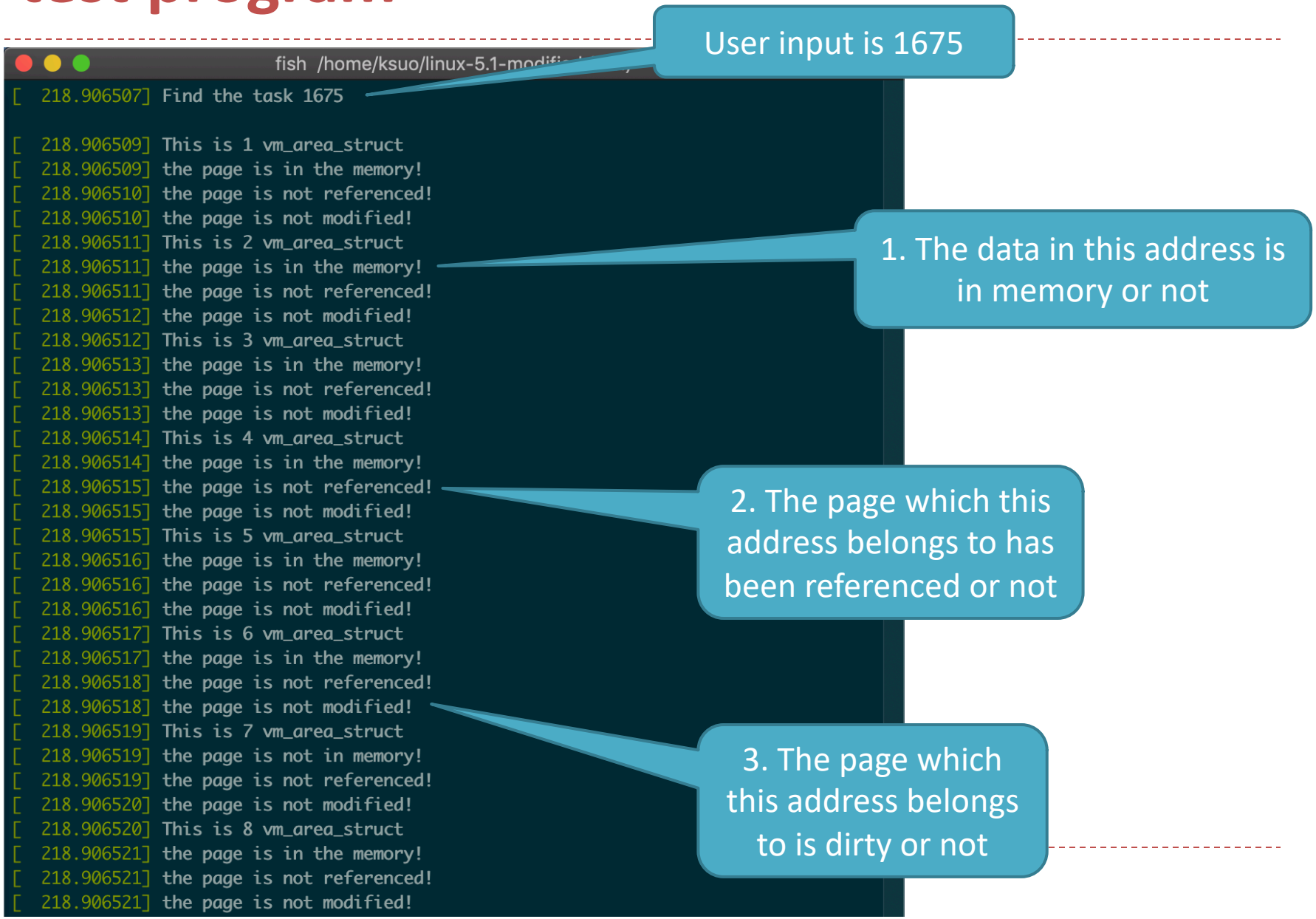
Assignment 2: user test program

```c
#include <linux/unistd.h>
#include <sys/syscall.h>
#include <sys/types.h>
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>

#define __NR_getPageInfo 339

/*
 * Execution: test using process pid=123
 * $ ./assignment2-user1.o  123
 */

int main(int argc, char *argv[])
{
    int pid = 0;

    pid = atoi(argv[1]);   //pid the proces
    syscall(__NR_getPageInfo, pid);

    return 0;
}
```

Input is the user process id

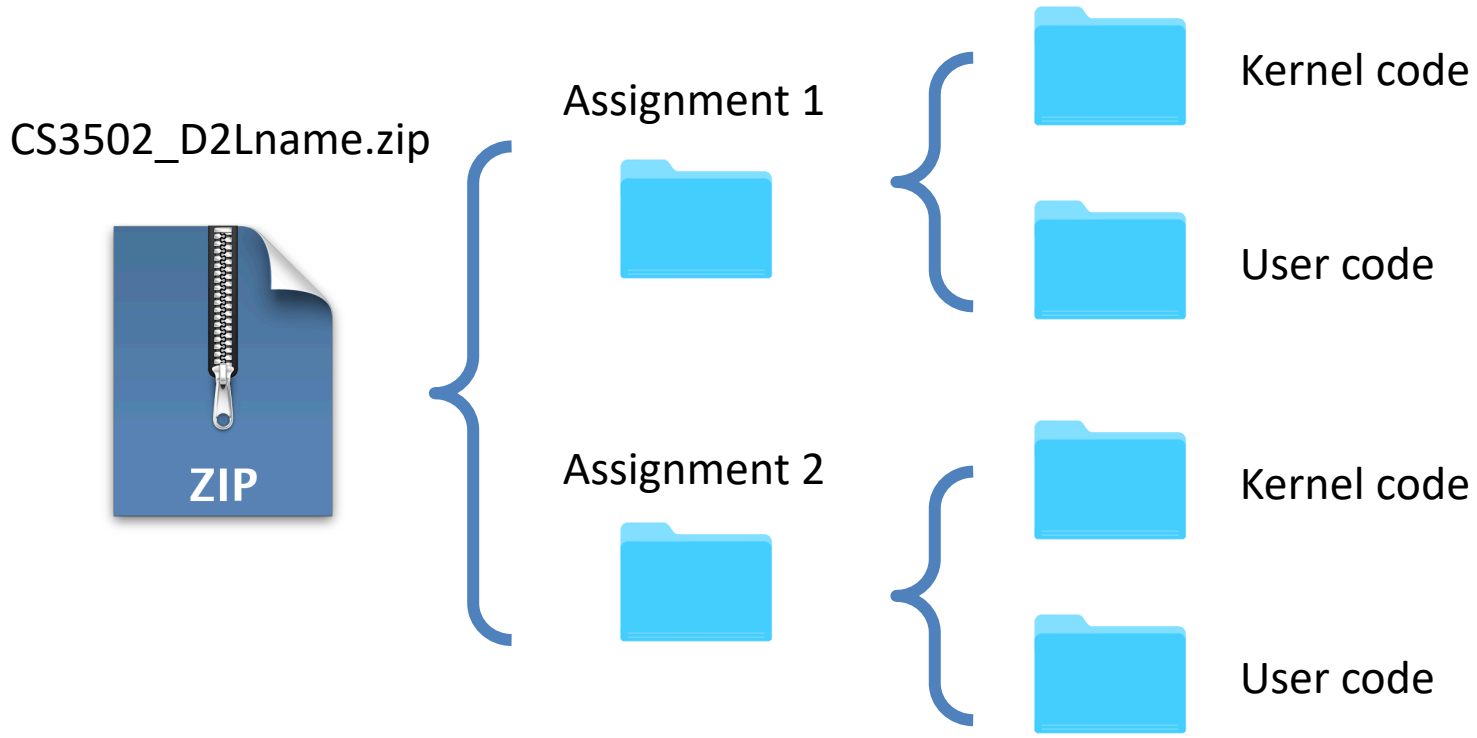
- The system call takes the pid of a process as input and outputs the following information of start address *vm_start* in each *vm_area_struct* that the process has:
  - Memory or disk?
  - Referenced or not?
  - Dirty or not?

Assignment 2: expected output of user test program



Submission

Submit your assignment zip file through D2L using the appropriate assignment link.



Questions

- T/Th 2-3pm, J-318
- Skype ID: *suokun.nju*, share home screen, only voice, no video, no remote control and privacy issue. No time or location limitation.