Arda Kıray
21302072
Section 2

CS 342 - Operating Systems
Project 3

Step 1: We are required to find a process' PCB by its given process id number and display some memory management informations about the process in this step. So after taking the processid from arguments, I traversed over "task_struct"s which are Linux's process control block implementations. I started from init_task which is the first process in the doubly linked list of task_structs and found the according process. If the process id is not valid program terminated with a message. After I reached the process, the mm_struct of the process is reached from its task_struct. If process is not an anonymous process and have an address space in the user context, processes mm_struct will be pointed mm value. Otherwise, it means that process is an anonymous process and uses the another finished processes mm_struct and it is pointed at the active_mm pointer by:

```
if(fprocess->mm != NULL){
    processmm = fprocess->mm;
  }
 else if(fprocess->active_mm != NULL){
    processmm = fprocess->active_mm;
 }
 else{
    printk(KERN_INFO "No virtual memory information is found...\n");
    return 0;
 }
```

After accessing the mm_struct, by using it vm_area_struct of the process is accessed. Mmap pointer points to the vm_area_struct in task_struct. vm_area_struct holds the virtual memory portions and sizes of those portions of the process. By using it, each virtual memory portions' starting address and its size are logged into the log file by:

```
 vm = processmm->mmap;
 while(vm->vm_next != NULL){
    printk(KERN_INFO "VM area start: 0x%lx VM area size: %lu\n", vm->vm_start, (vm-
>vm_end - vm->vm_start));
    vm = vm->vm_next;
 }
```

At the end additional informations about the virtual memory layout of the process are provided as beginning and end addresses of code, stack, enviromental variables, heap etc. mm_struct holds much of the informations about those partitions. In mm_struct the start and the end addresses of most of the segments are held as:

For code segment : start_code and end_code hold start and end addresses
For data segment : start_data and end_data hold start and end addresses
For heap segment : start_brk and brk hold start and end addresses
For main arguments: arg_start and arg_end hold start and end addresses
For enviromental variables : env_start and env_end hold start and end addresses

Also total number of used frames is held at hiwater_rss and total number of used virtual memory is held at total_vm parameters. Therefore the most of the informations are fetched by using these parameters. For calculating the sizes start addresses are subtracted from end addresses and displayed. One exception is happened when finding the stack informations. The end address of the stack is not provided at mm_struct. Therefore, virtual memory areas of the processes are investigated by using their flags and the flag for stack is VM_GROWSDOWN. Below function illustrates how begging and end address of the stack is found :

```
// Stack is traced via finding virtual memory area of it because no stack_end entry
// exists in mm_struct. The stack flag is VM_GROWSDOWN.
struct vm_area_struct *vmarea = processmm->mmap;
  int terminate = 0;
  do{
    if((vmarea->vm_flags|VM_GROWSDOWN) == vmarea->vm_flags){
        printk(KERN_INFO "The start address of the stack segment: 0x%lx\n", vmarea->vm_start);
        printk(KERN_INFO "The end address of the stack segment: 0x%lx\n", vmarea->vm_end);
        printk(KERN_INFO "The size of the stack segment: %lu\n", (vmarea->vm_end - vmarea->vm_start));
        terminate = 1;
    }
    vmarea = vmarea->vm_next;
 }while(terminate == 0);
```

Step 2: Linux uses 4 level hierarchical paging but they added 5 level paging support in their software in order to meet the needs of the future. Before 5 level support pages of the process could be accessed with this schema:

- mm_struct of the process hold a pgd_t struct pgd which mean page global directory which is the first page structure of the process.
- Each entry in the pgd has a pdu_t struct pdu which is the second level page table directory and addresses the further third level page table.
- By using this pdu_t struct, we can access pmd_t struct which means page middle directory that holds the 3rd level page table.
- At last by using pmd_t struct we can access pte_t struct that holds the last level page table.

However, after the 5 level paging support is added after kernel versions 4.12.x, now there is an extra p4d struct which changes the addressing schema as this:

pgd—>p4d—>pdu—>pmd—>pte—>physical frame

I read a lot of linux kernel codes to find out the paging scheme and some resources are below :

* **https://elixir.bootlin.com/linux/v4.15.9/source/arch/x86/include/asm/pgtable_types.h#L286**
* **https://github.com/lorenzo-stoakes/linux-vm-notes/blob/master/sections/page-tables.md**

* **https://elixir.bootlin.com/linux/v4.15.9/source/arch/x86/include/asm/pgtable_types.h#L286**
* **https://docs.huihoo.com/doxygen/linux/kernel/3.7/structpgd__t.html**

Although not being yet certainly true, as I understood from Linux paging mechanism of my kernel version which is 4.15.0-36 generic, Linux holds a pgtable_l5_enabled flag which indicates 5 level page table is enabled or not. Then when offsetting the p4d_t struct according to p4d_offset function it returns the pgd value if 5 level paging is not enabled. The code for p4d_offset is below:

```
/* 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);
}
```

Source : https://elixir.bootlin.com/linux/latest/source/arch/x86/include/asm/pgtable.h#L947

Since there is no processor around to support five level paging, I assumed that it is by default disabled and saved for future supports. Therefore, while accessing the page tables I assumed that this function just adds one useless step which returns the old value by type casting it to p4d_t structure. It still is important because while offsetting the addresses that I gathered from page table entries, I used the offset function that :

pgd_offset( ), p4d_offset( ), pud_offset( ), pmd_offset ( ) and pte_offset_kernel( ).

The sequence must be as above because after the addition of the p4d_t structure, now pud_offset( ) function cannot take a pgd_t struct but take p4d_t instead.

I started from the pgd entry and get the first tables entries. Empty entries are just logged with 0 value. To indicate the structure of the used entries, I masked corresponding bits to get the relevant portions of each entry portions. I used the provided Intel x86_64 entry structure to determine the fields and the code is below :

```
int a1 = (pgd[i].pgd)&(0x1); // Valid bit
int a2 = (pgd[i].pgd >> 1)&(0x1); // Read/Write bit
int a3 = (pgd[i].pgd >> 2)&(0x1); // User/Supervisor bit
int a4 = (pgd[i].pgd >> 3)&(0x1); // Page-level write through bit
int a5 = (pgd[i].pgd >> 4)&(0x1); // Page-level cache disabled bit
int a6 = (pgd[i].pgd >> 5)&(0x1); // Access bit
int a7 = (pgd[i].pgd >> 6)&(0x1); // Ignored bit
int a8 = (pgd[i].pgd >> 7)&(0x1); // Reserved bit
int a9 = (pgd[i].pgd >> 8)&(0xF); // Ignored bits (4 bits)
int a10 = (pgd[i].pgd >> 12)&(0xFFFFFF); // Physical frame base address (24 bits)
int a11 = (pgd[i].pgd >> 36)&(0xFFFF); // Reserved bits (16 bits)
int a12 = (pgd[i].pgd >> 52)&(0x7FF); // Ignored bits (11 bits)
int a13 = (pgd[i].pgd >> 63)&(0x1); // Execute disabled bit
```

Then I used p4d_offset(pgd, address) to get p4d entries. I considered them as one to one mapping not a page table entries since 5 level paging is not supported by Intel architecture. Then I used pdu_offset(p4d, address) to get physical address of the pdu elements and iterated like that. I logged the entries' structures in the way to the log file.

Below are the complete module codes namely partA and partB which are corresponds to the kernel modules in Step1 and Step2 :

Part A :

```
#include <linux/module.h>
#include <linux/kernel.h>
#include <linux/proc_fs.h>
#include <linux/sched/signal.h>
#include <linux/sched.h>
#include <linux/mm.h>

MODULE_LICENSE("GPL");
MODULE_AUTHOR("Arda Kıray");

static int processid = -1;
struct pid *pidstruct;
struct task_struct *processList = &init_task; // used to loop around task_struct list
struct task_struct *fprocess; // founded process struct pointer
struct mm_struct *processmm = NULL;
struct vm_area_struct *vm;
int found = 0; // boolean value for the result of process search


module_param(processid, int, S_IRUSR | S_IWUSR | S_IRGRP | S_IROTH);
MODULE_PARM_DESC(myint, "Process ID");

static int __init partA_init(void)
{
  do{
    if(processList->pid == processid){
      printk(KERN_INFO "Process found. The name is : %s and pid : %d\n", processList-
>comm, processList->pid);
      fprocess = processList;
      found = 1;
    }
  }while((processList = next_task(processList)) != &init_task && found == 0);


  if(found == 0){
    printk(KERN_INFO "There is no such process with a pid: %d\n", processid);
    return 0;
  }

  // If the process is not a kernel process and therefore have an address space in the user
context
```

```
  // mm_struct mm is assigned to processmm or else if it is an anonymous process the
previous processes'
  // mm_struct which is active_mm is assigned to processmm.
  if(fprocess->mm != NULL){
    processmm = fprocess->mm;
  }
  else if(fprocess->active_mm != NULL){
    processmm = fprocess->active_mm;
  }
  else{
    printk(KERN_INFO "No virtual memory information is found...\n");
    return 0;
  }

  // Below loop writes all virtual memory area start adress and size pairs
  vm = processmm->mmap;
  while(vm->vm_next != NULL){
    printk(KERN_INFO "VM area start: 0x%lx VM area size: %lu\n", vm->vm_start, (vm-
>vm_end - vm->vm_start));
    vm = vm->vm_next;
  }

  // Below code gives detailed virtual memory informations about the given process
  if(processmm != NULL){
    printk(KERN_INFO "Below are some virtual memory informations of process %s\n",
fprocess->comm);

    printk(KERN_INFO "The start address of the code segment: 0x%lx\n", processmm-
>start_code);
    printk(KERN_INFO "The end address of the code segment: 0x%lx\n", processmm-
>end_code);
    printk(KERN_INFO "The size of the code segment: %lu\n", (processmm->end_code -
processmm->start_code));

    printk(KERN_INFO "The start address of the data segment: 0x%lx\n", processmm-
>start_data);
    printk(KERN_INFO "The end address of the data segment: 0x%lx\n", processmm-
>end_data);
    printk(KERN_INFO "The size of the data segment: %lu\n", (processmm->end_data -
processmm->start_data));

    // Stack is traced via finding virtual memory area of it because no stack_end entry
    // exists in mm_struct. The stack flag is VM_GROWSDOWN.
    struct vm_area_struct *vmarea = processmm->mmap;
    int terminate = 0;
    do{
      if((vmarea->vm_flags|VM_GROWSDOWN) == vmarea->vm_flags){
          printk(KERN_INFO "The start address of the stack segment: 0x%lx\n", vmarea-
>vm_start);
          printk(KERN_INFO "The end address of the stack segment: 0x%lx\n", vmarea-
>vm_end);
```

```
        printk(KERN_INFO "The size of the stack segment: %lu\n", (vmarea->vm_end -
vmarea->vm_start));
        terminate = 1;
     }
     vmarea = vmarea->vm_next;
   }while(terminate == 0);

   printk(KERN_INFO "The start address of the heap segment: 0x%lx\n", processmm-
>start_brk);
   printk(KERN_INFO "The end address of the heap segment: 0x%lx\n", processmm-
>brk);
   printk(KERN_INFO "The size of the heap segment: %lu\n", (processmm->brk -
processmm->start_brk));

   printk(KERN_INFO "The start address of the main arguments: 0x%lx\n", processmm-
>arg_start);
   printk(KERN_INFO "The end address of the main arguments: 0x%lx\n", processmm-
>arg_end);
   printk(KERN_INFO "The size of the main arguments: %lu\n", (processmm->arg_end -
processmm->arg_start));

   printk(KERN_INFO "The start address of the enviroment variables: 0x%lx\n",
processmm->env_start);
   printk(KERN_INFO "The end address of the enviroment variables: 0x%lx\n",
processmm->env_end);
   printk(KERN_INFO "The size of the enviroment variables: %lu\n", (processmm-
>env_end - processmm->env_start));

   printk(KERN_INFO "The number of the used frames: %lu\n", processmm-
>hiwater_rss);
   printk(KERN_INFO "Total virtual memory used: %lu\n", processmm->total_vm);

  }

  return 0;
}

static void __exit partA_exit(void)
{
  printk(KERN_INFO "Module is removed succesfully...\n");
}

module_init(partA_init);
module_exit(partA_exit);
```

——

Part B :

```
#include <linux/module.h>
#include <linux/kernel.h>
```

```
#include <linux/proc_fs.h>
#include <linux/sched/signal.h>
#include <linux/sched.h>
#include <linux/mm.h>

MODULE_LICENSE("GPL");
MODULE_AUTHOR("Arda Kıray");

static int processid = -1;
struct pid *pidstruct;
struct task_struct *processList = &init_task; // used to loop around task_struct list
struct task_struct *fprocess; // founded process struct pointer
struct mm_struct *processmm = NULL;
struct vm_area_struct *vm;
int found = 0; // boolean value for the result of process search


module_param(processid, int, S_IRUSR | S_IWUSR | S_IRGRP | S_IROTH);
MODULE_PARM_DESC(myint, "Process ID");

static int __init partB_init(void)
{
  do{
    if(processList->pid == processid){
      printk(KERN_INFO "Process found. The name is : %s and pid : %d\n", processList-
>comm, processList->pid);
      fprocess = processList;
      found = 1;
    }
  }while((processList = next_task(processList)) != &init_task && found == 0);


  if(found == 0){
    printk(KERN_INFO "There is no such process with a pid: %d\n", processid);
    return 0;
  }

  // If the process is not a kernel process and therefore have an address space in the user
context
  // mm_struct mm is assigned to processmm or else if it is an anonymous process the
previous processes'
  // mm_struct which is active_mm is assigned to processmm.
  if(fprocess->mm != NULL){
    processmm = fprocess->mm;
  }
  else if(fprocess->active_mm != NULL){
    processmm = fprocess->active_mm;
  }
  else{
    printk(KERN_INFO "No virtual memory information is found...\n");
    return 0;
  }
```

```
 // FIRST LEVEL PAGE TABLE TRAVELSAL :

 pgd_t *pgd;

 struct vm_area_struct *pvm = processmm->mmap;

 unsigned long address = (pvm->vm_start);// >> 12) << 12; // process virtual address
without offset

 // Below code traverses outmost page table by using its page number index pgd. If the
pgd enrty
 // is empty or unsuitable loop returns for that entry by using none() and bad(). Else the
entry
 // with corresponding parsed informations are printed.
 pgd = pgd_offset(fprocess->mm, address);
 int i;
 printk(KERN_INFO "First level table entries with their structures are below:\n");

 for(i = 0; i < 512; i++){

  if(processmm->pgd[i].pgd == 0){
   //printk(KERN_INFO "Entry no. %d : %lu\n", i, processmm->pgd[i].pgd);
   return 0;
  }

  if (pgd_none(*pgd) || unlikely(pgd_bad(*pgd)))
   return 0;

  if(processmm->pgd[i].pgd != 0){

   int a1 = (pgd[i].pgd)&(0x1); // Valid bit
   int a2 = (pgd[i].pgd >> 1)&(0x1); // Read/Write bit
   int a3 = (pgd[i].pgd >> 2)&(0x1); // User/Supervisor bit
   int a4 = (pgd[i].pgd >> 3)&(0x1); // Page-level write through bit
   int a5 = (pgd[i].pgd >> 4)&(0x1); // Page-level cache disabled bit
   int a6 = (pgd[i].pgd >> 5)&(0x1); // Access bit
   int a7 = (pgd[i].pgd >> 6)&(0x1); // Ignored bit
   int a8 = (pgd[i].pgd >> 7)&(0x1); // Reserved bit
   int a9 = (pgd[i].pgd >> 8)&(0xF); // Ignored bits (4 bits)
   int a10 = (pgd[i].pgd >> 12)&(0xFFFFFF); // Physical frame base address (24 bits)
   int a11 = (pgd[i].pgd >> 36)&(0xFFFF); // Reserved bits (16 bits)
   int a12 = (pgd[i].pgd >> 52)&(0x7FF); // Ignored bits (11 bits)
   int a13 = (pgd[i].pgd >> 63)&(0x1); // Execute disabled bit

   printk(KERN_INFO "Entry no. %d : %lu\n", i, pgd[i].pgd);
   printk(KERN_INFO "   Valid bit: %lx\n", a1);
   printk(KERN_INFO "   Read/Write bit: %lx\n", a2);
   printk(KERN_INFO "   User/Supervisor bit: %lx\n", a3);
   printk(KERN_INFO "   Page-level write through bit: %lx\n", a4);
   printk(KERN_INFO "   Page-level cache disabled bit: %lx\n", a5);
   printk(KERN_INFO "   Access bit: %lx\n", a6);
```

```
    printk(KERN_INFO "   Ignored bit: %lx\n", a7);
    printk(KERN_INFO "   Reserved bit: %lx\n", a8);
    printk(KERN_INFO "   Ignored bits(4): %lx\n", a9);
    printk(KERN_INFO "   Physical frame base address(24): %lx\n", a10);
    printk(KERN_INFO "   Reserved bits(16): %lx\n", a11);
    printk(KERN_INFO "   Ignored bits(11): %lx\n", a12);
    printk(KERN_INFO "   Execute disabled bit: %lx\n", a13);
    printk(KERN_INFO "-----------------------------------------");

  }
}

// SECOND LEVEL PAGE TABLES TRAVELSAL :

printk(KERN_INFO "Second level table entries with their structures are below:\n");

p4d_t *p4d;
pud_t *pud;
p4d = p4d_offset(pgd, address);
pud = pud_offset(p4d, address);

for(i = 0; i < 262144; i++){

  if(pud[i].pud == 0){
    //printk(KERN_INFO "Entry no. %d : %lu\n", i, pud[i]);
    return 0;
  }

  if (pud_none(*pud) || unlikely(pud_bad(*pud)))
    return 0;

  if(pud[i].pud != 0){

    int a1 = (pud[i].pud)&(0x1); // Valid bit
    int a2 = (pud[i].pud >> 1)&(0x1); // Read/Write bit
    int a3 = (pud[i].pud >> 2)&(0x1); // User/Supervisor bit
    int a4 = (pud[i].pud >> 3)&(0x1); // Page-level write through bit
    int a5 = (pud[i].pud >> 4)&(0x1); // Page-level cache disabled bit
    int a6 = (pud[i].pud >> 5)&(0x1); // Access bit
    int a7 = (pud[i].pud >> 6)&(0x1); // Ignored bit
    int a8 = (pud[i].pud >> 7)&(0x1); // Reserved bit
    int a9 = (pud[i].pud >> 8)&(0xF); // Ignored bits (4 bits)
    int a10 = (pud[i].pud >> 12)&(0xFFFFFF); // Physical frame base address (24 bits)
    int a11 = (pud[i].pud >> 36)&(0xFFFF); // Reserved bits (16 bits)
    int a12 = (pud[i].pud >> 52)&(0x7FF); // Ignored bits (11 bits)
    int a13 = (pud[i].pud >> 63)&(0x1); // Execute disabled bit

    printk(KERN_INFO "Entry no. %d : %lu\n", i, pud[i]);
    printk(KERN_INFO "   Valid bit: %lx\n", a1);
    printk(KERN_INFO "   Read/Write bit: %lx\n", a2);
    printk(KERN_INFO "   User/Supervisor bit: %lx\n", a3);
    printk(KERN_INFO "   Page-level write through bit: %lx\n", a4);
```

```
    printk(KERN_INFO "   Page-level cache disabled bit: %lx\n", a5);
    printk(KERN_INFO "   Access bit: %lx\n", a6);
    printk(KERN_INFO "   Ignored bit: %lx\n", a7);
    printk(KERN_INFO "   Reserved bit: %lx\n", a8);
    printk(KERN_INFO "   Ignored bits(4): %lx\n", a9);
    printk(KERN_INFO "   Physical frame base address(24): %lx\n", a10);
    printk(KERN_INFO "   Reserved bits(16): %lx\n", a11);
    printk(KERN_INFO "   Ignored bits(11): %lx\n", a12);
    printk(KERN_INFO "   Execute disabled bit: %lx\n", a13);
    printk(KERN_INFO "-------------------------------------------------\n");
  }
}


// THIRD LEVEL PAGE TABLES TRAVELSAL :

  printk(KERN_INFO "Third level table entries with their structures are below:\n");

  pmd_t *pmd;
  pmd = pmd_offset(pud, address);

  for(i = 0; i < 134217728; i++){
    if(pmd[i].pmd == 0){
        //printk(KERN_INFO "Entry no. %d : %lu\n", i, pmd[i]);
        return 0;
    }

    if (pmd_none(*pmd) || unlikely(pmd_bad(*pmd)))
        return 0;

    if(pmd[i].pmd != 0){

        int a1 = (pmd[i].pmd)&(0x1); // Valid bit
        int a2 = (pmd[i].pmd >> 1)&(0x1); // Read/Write bit
        int a3 = (pmd[i].pmd >> 2)&(0x1); // User/Supervisor bit
        int a4 = (pmd[i].pmd >> 3)&(0x1); // Page-level write through bit
        int a5 = (pmd[i].pmd >> 4)&(0x1); // Page-level cache disabled bit
        int a6 = (pmd[i].pmd >> 5)&(0x1); // Access bit
        int a7 = (pmd[i].pmd >> 6)&(0x1); // Ignored bit
        int a8 = (pmd[i].pmd >> 7)&(0x1); // Reserved bit
        int a9 = (pmd[i].pmd >> 8)&(0xF); // Ignored bits (4 bits)
        int a10 = (pmd[i].pmd >> 12)&(0xFFFFFF); // Physical frame base address (24 bits)
        int a11 = (pmd[i].pmd >> 36)&(0xFFFF); // Reserved bits (16 bits)
        int a12 = (pmd[i].pmd >> 52)&(0x7FF); // Ignored bits (11 bits)
        int a13 = (pmd[i].pmd >> 63)&(0x1); // Execute disabled bit

        printk(KERN_INFO "Entry no. %d : %lu\n", i, pmd[i]);
        printk(KERN_INFO "   Valid bit: %lx\n", a1);
        printk(KERN_INFO "   Read/Write bit: %lx\n", a2);
        printk(KERN_INFO "   User/Supervisor bit: %lx\n", a3);
        printk(KERN_INFO "   Page-level write through bit: %lx\n", a4);
        printk(KERN_INFO "   Page-level cache disabled bit: %lx\n", a5);
```

```
        printk(KERN_INFO "   Access bit: %lx\n", a6);
        printk(KERN_INFO "   Ignored bit: %lx\n", a7);
        printk(KERN_INFO "   Reserved bit: %lx\n", a8);
        printk(KERN_INFO "   Ignored bits(4): %lx\n", a9);
        printk(KERN_INFO "   Physical frame base address(24): %lx\n", a10);
        printk(KERN_INFO "   Reserved bits(16): %lx\n", a11);
        printk(KERN_INFO "   Ignored bits(11): %lx\n", a12);
        printk(KERN_INFO "   Execute disabled bit: %lx\n", a13);
        printk(KERN_INFO "-------------------------------------------------\n");

    }
  }

// FORTH LEVEL PAGE TABLES TRAVELSAL :

  printk(KERN_INFO "Forth level table entries with their structures are below:\n");

  pte_t *pte;
  pte = pte_offset_kernel(pmd, address);

  for(i = 0; i < 68719476736; i++){
    if(pte[i].pte == 0){
        //printk(KERN_INFO "Entry no. %d : %lu\n", i, pte[i]);
        return 0;
    }

    if (pte_none(*pte) || unlikely(pte_accessible(processmm, *pte)))
        return 0;

    if(pte[i].pte != 0){

        int a1 = (pte[i].pte)&(0x1); // Valid bit
        int a2 = (pte[i].pte >> 1)&(0x1); // Read/Write bit
        int a3 = (pte[i].pte >> 2)&(0x1); // User/Supervisor bit
        int a4 = (pte[i].pte >> 3)&(0x1); // Page-level write through bit
        int a5 = (pte[i].pte >> 4)&(0x1); // Page-level cache disabled bit
        int a6 = (pte[i].pte >> 5)&(0x1); // Access bit
        int a7 = (pte[i].pte >> 6)&(0x1); // Ignored bit
        int a8 = (pte[i].pte >> 7)&(0x1); // Reserved bit
        int a9 = (pte[i].pte >> 8)&(0xF); // Ignored bits (4 bits)
        int a10 = (pte[i].pte >> 12)&(0xFFFFFF); // Physical frame base address (24 bits)
        int a11 = (pte[i].pte >> 36)&(0xFFFF); // Reserved bits (16 bits)
        int a12 = (pte[i].pte >> 52)&(0x7FF); // Ignored bits (11 bits)
        int a13 = (pte[i].pte >> 63)&(0x1); // Execute disabled bit

        printk(KERN_INFO "Entry no. %d : %lu\n", i, pte[i]);
        printk(KERN_INFO "   Valid bit: %lx\n", a1);
        printk(KERN_INFO "   Read/Write bit: %lx\n", a2);
        printk(KERN_INFO "   User/Supervisor bit: %lx\n", a3);
        printk(KERN_INFO "   Page-level write through bit: %lx\n", a4);
        printk(KERN_INFO "   Page-level cache disabled bit: %lx\n", a5);
        printk(KERN_INFO "   Access bit: %lx\n", a6);
```

```
        printk(KERN_INFO "   Ignored bit: %lx\n", a7);
        printk(KERN_INFO "   Reserved bit: %lx\n", a8);
        printk(KERN_INFO "   Ignored bits(4): %lx\n", a9);
        printk(KERN_INFO "   Physical frame base address(24): %lx\n", a10);
        printk(KERN_INFO "   Reserved bits(16): %lx\n", a11);
        printk(KERN_INFO "   Ignored bits(11): %lx\n", a12);
        printk(KERN_INFO "   Execute disabled bit: %lx\n", a13);
        printk(KERN_INFO "-------------------------------------------------\n");

     }
    }

  return 0;
}

static void __exit partB_exit(void)
{
  printk(KERN_INFO "Module is removed succesfully...\n");
}

module_init(partB_init);
module_exit(partB_exit);
```