CS - 342

Operating Systems

Project # 3

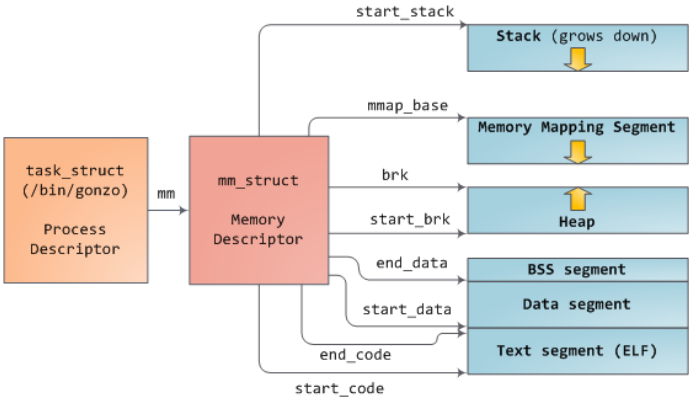
Kerem Ayöz - 21501569
Cansu Yıldırım – 21502891

Project 3 - Part B

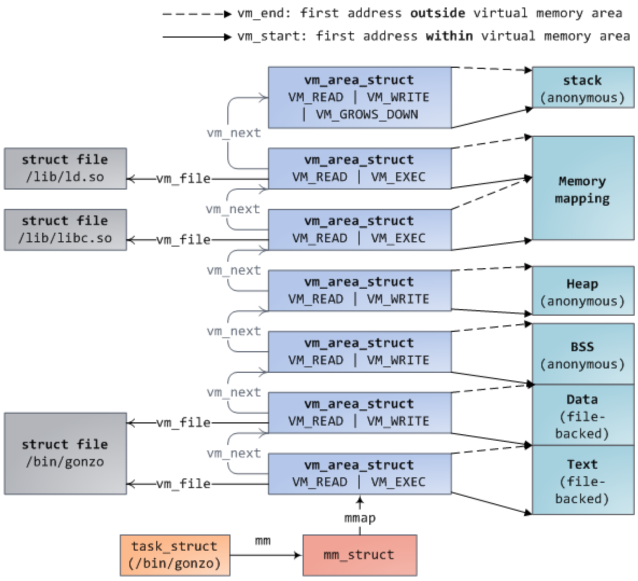
We used ubuntu 16.04.01. Our kernel module version is 4.4.164

Step-1. Develop a Kernel Module for VM Information

We created a hello.c file inside the kernel module. Our module takes an argument named processid. task_struct *cur_task shows the processid of the currently working process. Our module searches all the working processes and it stops when it finds the process with the given id. After it finds the desired process, it goes to vm_area_information() function in our code to print its virtual memory usage. In the mm_struct of Linux , the start and end addresses can be found. Thus, we used cur_task->mm to reach its fields. We found the start, end and size of the code, data, heap, main arguments, environment variables, rss and total vm.



However, for stack, there is only the start address of the stack. To find its end address, we used vm_area_struct *cur_vm.



As it is in above, cur_vm is cur_task->mm->mmap. By moving with vm_next, we reached the end of the stack (cur_vm->vm_end). Then, we found the size of the stack. We checked our results by using cat /proc/pid/maps and sudo pmap <pid> commands.

Step-2. Multi-Level Page Table Content

After finding the process that we want, we access its Page Global Director which is called "pgd" from the process's mm field(**task->mm->pgd**). This pointer points to the beginning address of the 1st level page table of process. It has 512 entries, each entry points to different second level tables if they are invalid. We traverse pgd table by following code:

```
static void pgd_table_lookup(struct task_struct *cur_task, unsigned long addr, unsigned long end) {
    pgd_t *pgd;
    unsigned long i = 0;
    unsigned long next;

    pgd = pgd_offset(cur_task->mm, addr);
    do {
        next = pgd_addr_end(addr, end);
        if (pgd_none(*pgd) || pgd_bad(*pgd)) {
            printk("Invalid Entry: %d\n",i);
        }
        else {
            printk("Valid PGD Entry: %lu\n",i++);
            pud_table_lookup(pgd, addr, next);
        }
        addr = next;
        pgd++;
    } while (addr < end);
}
```

We take current task pointer, starting address and ending address for traversal process. Starting address and ending address is the starting and ending virtual addresses for the pgd table, since first table can represent all of the virtual address space of process. Then we use the **pgd_offset()** function that calculates the pgd entry which represent the starting address. In a loop, we iterate through the pgd entries. If they are invalid, we print a error message. If they are not, we start to traverse the second level table which is Page Upper Directory called "pud". As we go deeper in the page tables, their address representation capabilities gets lower. For instance pgd table could represent all of the virtual address space of the process while pud table-which is an entry of the pgd table- could only represent the *(total address space / 512)* bytes. As we go deeper, their address representation intervals shrink.

In order to provide the proper addresses, we use a function called **pgd_addr_end()** which basically divides the address space that pgd could represent into 512 equal parts and returns the ending address of the first part. So addr points the beginning and next points to the *(beginning + address space that one pud table could represent)*. By iterating the addr and next pointer by same amount, we can traverse each pud table and could give the proper address interval. After that we traverse the pud table with the same logic with the following code:

```
static void pud_table_lookup(pgd_t *pgd, unsigned long addr, unsigned long end) {
    pud_t *pud;
    unsigned long j = 0;
    unsigned long next;

    pud = pud_offset(pgd, addr);
    do {
        next = pud_addr_end(addr, end);
        if (pud_none(*pud) || pud_bad(*pud)) {
            printk("Invalid Entry: %d\n",j);
        }
        else {
            printk("Valid PUD Entry: %lu\n",j++);
            pmd_table_lookup(pud, addr, next);
        }
        addr = next;
        pud++;
    } while(addr < end);
}
```

As we can see that this code is something symmetric with the previous one. We iterate through the table and as we find valid entries, we go for the upper level table Page Middle Directory which is called as “pmd”. Same logic is applied here; since a pmd table could represent smaller address space, we divide the address space that a pud table could represent into 512 equal intervals. As we iterate through the pud table, we also iterate the addr and next addresses. For instance if the second entry of the pud table is valid, then we need to search in the second equal piece that we found by addresses next and addr.

After that we traverse the pmd table by following code:

```
static void pmd_table_lookup(pud_t *pud, unsigned long addr, unsigned long end) {
    pmd_t *pmd;
    unsigned long k = 0;
    unsigned long next;

    pmd = pmd_offset(pud, addr);
    do {
        next = pmd_addr_end(addr, end);
        if (pmd_none(*pmd) || pmd_bad(*pmd)) {
            printk("Invalid Entry: %d\n",k++);
        }
        else {
            printk("Valid PMD Entry: %lu\n",k++);
            pte_table_lookup(pmd ,addr, next);
        }
        addr = next;
        pmd++;
    } while (addr < end);
}
```

Same logic is still valid in the traversal of that table. We search for the valid entries, if we found that we pass the proper address space for function that traverses next page table Page Table Entry which is called “pte”. This is the table that contains the physical frame numbers inside its entries. Each pte table represent an address space with size **PAGE_SIZE.** We iterate in that list by 4 byte steps and print the physical address and physical frame number by functions **pte_page()** and **pte_pfn().** We also check the invalid bit for accessing the valid entries only.

```
static void pte_table_lookup(pmd_t *pmd, unsigned long addr, unsigned long end) {
    pte_t *pte;
    unsigned long m = 0;
    unsigned long pfn;
    struct page *page = NULL;

    pte = pte_offset_map(pmd, addr);
    do {
        if (pte_none(*pte)) {
            printk("Invalid Entry: %d\n",m++);
        }
        else {
            printk("Valid PTE entry: %lu\n", m++);
            pfn = pte_pfn(*pte);
            count++;
            page = pte_page(*pte);
            printk(KERN_INFO "Frame number is: # %lu\n", pfn);
            printk(KERN_INFO "Physical address is : 0x%lu\n", page);
        }
        addr += PAGE_SIZE;
        pte++;
        m++;
    } while(addr < end);
    pte_unmap(pte);
}
```

With that program, we demonstrated the Intel x86 64-bit 4-level paging architecture. We saw that it uses 48-bit logical addresses and 52-bit physical addresses. It has 4-level hierarchical page tables.

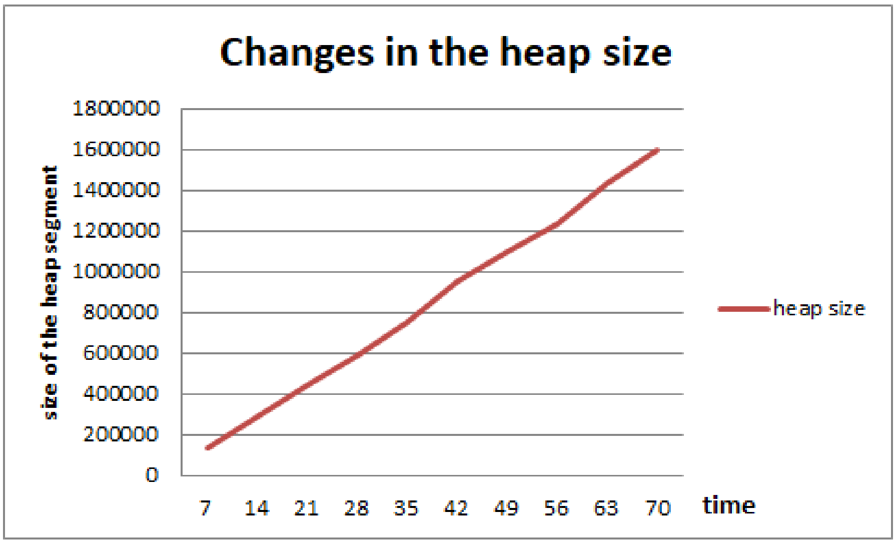
Sample outputs of our program are below. The process **init** is examined.

```
Dec 12 12:01:11 ubuntu kernel: [ 7295.116634] Invalid Entry: 126
Dec 12 12:01:11 ubuntu kernel: [ 7295.116635] Invalid Entry: 128
Dec 12 12:01:11 ubuntu kernel: [ 7295.116635] Invalid Entry: 130
Dec 12 12:01:11 ubuntu kernel: [ 7295.116636] Invalid Entry: 132
Dec 12 12:01:11 ubuntu kernel: [ 7295.116636] Invalid Entry: 134
Dec 12 12:01:11 ubuntu kernel: [ 7295.116637] Invalid Entry: 136
Dec 12 12:01:11 ubuntu kernel: [ 7295.116637] Invalid Entry: 138
Dec 12 12:01:11 ubuntu kernel: [ 7295.116637] Invalid Entry: 140
Dec 12 12:01:11 ubuntu kernel: [ 7295.116638] Valid PTE entry: 142
Dec 12 12:01:11 ubuntu kernel: [ 7295.116638] Frame number is: # 115604
Dec 12 12:01:11 ubuntu kernel: [ 7295.116639] Physical address is : 0x18446719884461139200
Dec 12 12:01:11 ubuntu kernel: [ 7295.116639] Valid PTE entry: 144
Dec 12 12:01:11 ubuntu kernel: [ 7295.116640] Frame number is: # 115599
Dec 12 12:01:11 ubuntu kernel: [ 7295.116640] Physical address is : 0x18446719884461138880
Dec 12 12:01:11 ubuntu kernel: [ 7295.116640] Valid PTE entry: 146
Dec 12 12:01:11 ubuntu kernel: [ 7295.116641] Frame number is: # 115592
Dec 12 12:01:11 ubuntu kernel: [ 7295.116641] Physical address is : 0x18446719884461138432
Dec 12 12:01:11 ubuntu kernel: [ 7295.116642] Valid PTE entry: 148
Dec 12 12:01:11 ubuntu kernel: [ 7295.116642] Frame number is: # 127977
Dec 12 12:01:11 ubuntu kernel: [ 7295.116642] Physical address is : 0x18446719884461931072
Dec 12 12:01:11 ubuntu kernel: [ 7295.116643] Valid PTE entry: 150
Dec 12 12:01:11 ubuntu kernel: [ 7295.116643] Frame number is: # 127978
Dec 12 12:01:11 ubuntu kernel: [ 7295.116644] Physical address is : 0x18446719884461931136
Dec 12 12:01:11 ubuntu kernel: [ 7295.116644] Valid PTE entry: 152
Dec 12 12:01:11 ubuntu kernel: [ 7295.116645] Frame number is: # 1757413
Dec 12 12:01:11 ubuntu kernel: [ 7295.116645] Physical address is : 0x18446719884566214976
Dec 12 12:01:11 ubuntu kernel: [ 7295.116645] Valid PTE entry: 154
Dec 12 12:01:11 ubuntu kernel: [ 7295.116646] Frame number is: # 127973
Dec 12 12:01:11 ubuntu kernel: [ 7295.116646] Physical address is : 0x18446719884461930816
Dec 12 12:01:11 ubuntu kernel: [ 7295.116647] Valid PTE entry: 156
Dec 12 12:01:11 ubuntu kernel: [ 7295.116647] Frame number is: # 1765876
Dec 12 12:01:11 ubuntu kernel: [ 7295.116648] Physical address is : 0x18446719884566756608
Dec 12 12:01:11 ubuntu kernel: [ 7295.116648] Valid PTE entry: 158
Dec 12 12:01:11 ubuntu kernel: [ 7295.116648] Frame number is: # 379243
Dec 12 12:01:11 ubuntu kernel: [ 7295.116649] Physical address is : 0x18446719884478012096
Dec 12 12:01:11 ubuntu kernel: [ 7295.116649] Valid PTE entry: 160
Dec 12 12:01:11 ubuntu kernel: [ 7295.116650] Frame number is: # 387536
Dec 12 12:01:11 ubuntu kernel: [ 7295.116650] Physical address is : 0x18446719884478542848
Dec 12 12:01:11 ubuntu kernel: [ 7295.116651] Valid PTE entry: 162
Dec 12 12:01:11 ubuntu kernel: [ 7295.116651] Frame number is: # 277864
Dec 12 12:01:11 ubuntu kernel: [ 7295.116651] Physical address is : 0x18446719884471523840
Dec 12 12:01:11 ubuntu kernel: [ 7295.116652] Valid PTE entry: 164
Dec 12 12:01:11 ubuntu kernel: [ 7295.116652] Frame number is: # 285974
Dec 12 12:01:11 ubuntu kernel: [ 7295.116653] Physical address is : 0x18446719884472042880
Dec 12 12:01:11 ubuntu kernel: [ 7295.116653] Valid PTE entry: 166
Dec 12 12:01:11 ubuntu kernel: [ 7295.116654] Frame number is: # 8756
Dec 12 12:01:11 ubuntu kernel: [ 7295.116654] Physical address is : 0x18446719884454300928
Dec 12 12:01:11 ubuntu kernel: [ 7295.116654] Valid PTE entry: 168
Dec 12 12:01:11 ubuntu kernel: [ 7295.116655] Frame number is: # 7660
Dec 12 12:01:11 ubuntu kernel: [ 7295.116655] Physical address is : 0x18446719884454230784
```

Step 3. Write an Application Allocating Memory Dynamically

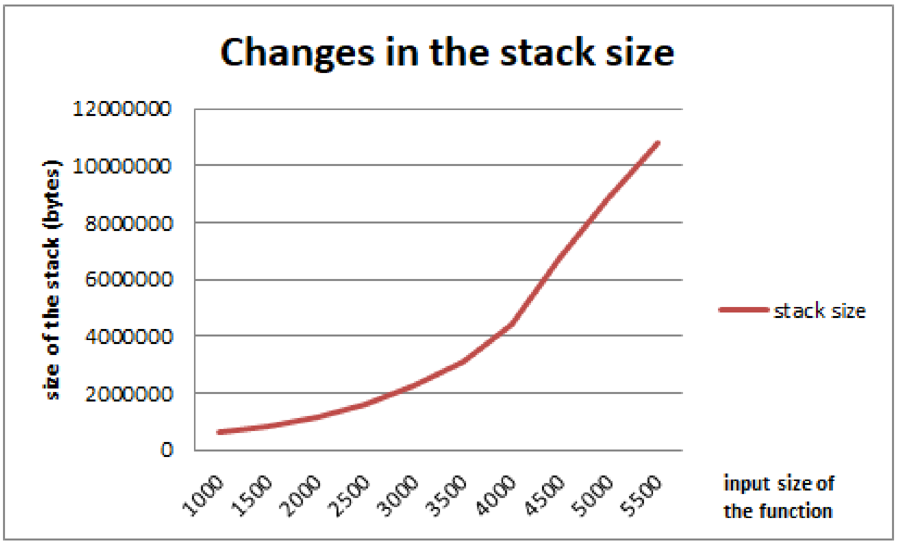
We wrote an application named app.c. In every 7 seconds, in main function we allocate and deallocate memory by using **malloc()**. We allocate memory with the size of 12800. When its second turn, it allocates another 12800 bytes and it continues like this. Then, we load our module and see the changes in the heap size. Our results are below:

sec	7	14	21	28	35	42	49	56	63	70
bytes	135168	286720	442368	593920	749768	956324	1095653	1234297	1436259	1598735



Thus, every time it allocates memory, heap segment extends in the virtual memory. To observe the stack size, we used recursive **fibonacci()** function. We give the input size of a function which are 1000, 1500, 2000, 2500... The size of a stack in the virtual memory is changed as it is shown below.

input size	1000	1500	2000	2500	3000	3500	4000	4500	5000	5500
bytes	632256	821932	1135616	1589862	2257604	3138069	4393296	6754329	8843178	10768569



Step 4. Address Translation

We get the process id and virtual address as a parameter for this step. After we find the cur_task with the given process id, we send this cur_task and given virtual address to the address_translation() function.

In this function, we first find the pgd by using **pgd_offset(cur_task->mm,virtaddr)**. We check whether the present bit of this entry is valid or not. If it is valid, we find its pud by using **pud_offset(pgd, virtaddr)**. Then, we check the present bit again. If it is valid we find the pmd by using **pmd_offset(pud,virtaddr).** Then, again we check the present bit. If it is valid we find the pte with **pte_offset_map(pmd,virtaddr)**. Lastly, we check the pte entry present bit. If it is valid with the function **pte_page(*pte)**, we print the physical address of the given virtual address.

Pagewalk Kernel Module Code

pagewalk.c

```
#include <linux/init.h>
#include <linux/module.h>
#include <linux/kernel.h>
#include <linux/errno.h>
#include <linux/sched.h>
#include <asm/pgtable.h>
#include <asm/page.h>
#include <linux/proc_fs.h>
#include <linux/mm.h>

MODULE_LICENSE("GPL");
MODULE_AUTHOR("Kerem & Cansu");
MODULE_DESCRIPTION("Module that does page walk through the given processid.");
MODULE_VERSION("3.0");
static int processid = 0;
module_param(processid, int, 0);

static unsigned long virtaddr = 0;
module_param(virtaddr, long, 0);


int count = 0; // Page counter

static void pte_table_lookup(pmd_t *pmd, unsigned long addr, unsigned long end) {
  pte_t *pte;
  unsigned long m = 0;
  unsigned long pfn;
  struct page *page = NULL;

  pte = pte_offset_map(pmd, addr);
  do {
    if (pte_none(*pte)) {
      //printk("Invalid Entry: %d\n",m++);
    }
    else {
      printk("Valid PTE entry: %lu\n", m++);
      pfn = pte_pfn(*pte);
      page = pte_page(*pte);
      count++;
      printk("Frame number is:# %lu\n", pfn);
      printk("Physical address is : 0x%p\n", page);
    }
    addr += PAGE_SIZE;
    pte++;
  } while(addr < end);
  pte_unmap(pte);
}

static void pmd_table_lookup(pud_t *pud, unsigned long addr, unsigned long end) {
  pmd_t *pmd;
  unsigned long k = 0;
  unsigned long next;

  pmd = pmd_offset(pud, addr);
  do {
    next = pmd_addr_end(addr, end);
    if (pmd_none(*pmd) || pmd_bad(*pmd)) {
```

```
      //printk("Invalid Entry: %d\n",k++);
    }
    else {
      printk("Valid PMD Entry: %lu\n",k++);
      pte_table_lookup(pmd ,addr, next);
    }
    addr = next;
    pmd++;
  } while (addr < end);
}

static void pud_table_lookup(pgd_t *pgd, unsigned long addr, unsigned long end) {
  pud_t *pud;
  unsigned long j = 0;
  unsigned long next;

  pud = pud_offset(pgd, addr);
  do {
    next = pud_addr_end(addr, end);
    if (pud_none(*pud) || pud_bad(*pud)) {
      //printk("Invalid Entry: %d\n",j++);
    }
    else {
      printk("Valid PUD Entry: %lu\n",j++);
      pmd_table_lookup(pud, addr, next);
    }
    addr = next;
    pud++;
  } while(addr < end);
}

static void pgd_table_lookup(struct task_struct *cur_task, unsigned long addr,
    unsigned long end) {
  pgd_t *pgd;
  unsigned long i = 0;
  unsigned long next;

  pgd = pgd_offset(cur_task->mm, addr);
  do {
    next = pgd_addr_end(addr, end);
    if (pgd_none(*pgd) || pgd_bad(*pgd)) {
      //printk("Invalid Entry: %d\n",i++);
    }
    else {
      printk("Valid PGD Entry: %lu\n",i++);
      pud_table_lookup(pgd, addr, next);
    }
    addr = next;
    pgd++;
  } while (addr < end);
}

static void page_walk(struct task_struct *cur_task) {
  unsigned long start, end;
  struct vm_area_struct *cur_vm;

  cur_vm = cur_task->mm->mmap;
  start = cur_vm->vm_start;

  while(cur_vm->vm_next)
```

```
        cur_vm = cur_vm->vm_next;

  end = cur_vm->vm_end;

  pgd_table_lookup(cur_task, start, end);
  printk("Page count: %d\n",count);
}

static void vm_area_information(struct task_struct *cur_task) {
  struct vm_area_struct *cur_vm;
  cur_vm = cur_task->mm->mmap;
  while(cur_vm->vm_next){
        cur_vm = cur_vm->vm_next;
  }

  printk("Code_start: 0x%lu Code_end: 0x%lu Code_size: %lu bytes\n", cur_task->mm->start_code,cur_task->mm->end_code,cur_task->mm->end_code-cur_task->mm->start_code);
  printk("Data_start: 0x%lu Data_end: 0x%lu Data_size: %lu bytes\n", cur_task->mm->start_data,cur_task->mm->end_data,cur_task->mm->end_data-cur_task->mm->start_data);
  printk("Stack_start: 0x%lu Stack_end: 0x%lu Stack_size: %lu bytes\n", cur_task->mm->start_stack,cur_vm->vm_end,cur_vm->vm_end-cur_task->mm->start_stack);
  printk("Heap_start: 0x%lu Heap_end: 0x%lu Heap_size: %lu bytes\n", cur_task->mm->start_brk,cur_task->mm->brk,cur_task->mm->brk-cur_task->mm->start_brk);
  printk("Argument_start: 0x%lu Argument_end: 0x%lu Argument_size: %lu bytes\n", cur_task->mm->arg_start,cur_task->mm->arg_end,cur_task->mm->arg_end-cur_task->mm->arg_start);
  printk("EV_start: 0x%lu EV_end: 0x%lu EV_size: %lu bytes\n", cur_task->mm->env_start,cur_task->mm->env_end,cur_task->mm->env_end-cur_task->mm->env_start);
  printk("RSS: %lu\n", cur_task->mm->hiwater_rss);
  printk("Total_VM: %lu pages\n", cur_task->mm->total_vm);
}

static void address_translation(struct task_struct *cur_task,unsigned long virtaddr) {
  pgd_t *pgd;
    pud_t *pud;
  pmd_t *pmd;
  pte_t *pte;

  struct page *page = NULL;

  pgd = pgd_offset(cur_task->mm, virtaddr);
  if (pgd_none(*pgd) || pgd_bad(*pgd))
    printk("Invalid Virtual Address");
  else {
    pud = pud_offset(pgd, virtaddr);
    if (pud_none(*pud) || pud_bad(*pud))
      printk("Invalid Virtual Address");
    else {
      pmd = pmd_offset(pud,virtaddr);
      if (pmd_none(*pmd) || pmd_bad(*pmd))
        printk("Invalid Virtual Address");
      else {
        pte = pte_offset_map(pmd,virtaddr);
        if (pte_none(*pte))
          printk("Invalid Virtual Address");
        else {
          page = pte_page(*pte);
```

```c
          printk("Physical Address: 0x%p\n", page);
        } // else pte
      } // else pmd
    } // else pud
  } // else pgd
}

static int __init hello_init(void){
  struct task_struct *task = current;
    struct task_struct *cur_task;

    printk("Current process = %s, Current pid = %d\n",task->comm, task->pid);

    for_each_process(cur_task) {
      if (cur_task->pid == processid)
        break;
    }
  printk("Given process: %s [%d]\n",cur_task->comm , cur_task->pid);

  // STEP 1
  vm_area_information(cur_task);

  // STEP 2
  page_walk(cur_task);

    // STEP 3

    // STEP 4
    address_translation(cur_task, cur_task->mm->start_code);

  // Done
    printk("Hello by Kerem & Cansu %d\n", processid);
    return 0;
}

static void __exit hello_exit(void){
  printk("Goodbye by Kerem & Cansu\n");
}

module_init(hello_init);
module_exit(hello_exit);
```

Application Code

app.c

```
#include<stdio.h>
#include<stdlib.h>
#include <time.h>

int *heap_arr;

void delay(int number_of_seconds)  {
    int milli_seconds = 1000 * number_of_seconds;
    clock_t start_time = clock();
    while (clock() < start_time + milli_seconds);
}

int fibonacci(int n) {
  if (n == 0 || n == 1)
    return n;
  else {
    int f1 = fibonacci(n-1);
    int f2 = fibonacci(n-2);
    return f1 + f2;
  }
}


int main() {

// Observe Heap Size
  while(1) {
    printf("Allocating!\n");
    heap_arr = malloc(sizeof(int)*12800);
    delay(7000);
    printf("De-allocating!\n");
    //free(heap_arr);
    delay(7000);
   }
/*
   // Observe Stack Size
  int i = 0;
  int total = 0;
  for (i = 1; i <= 100; i++) {
    total += fibonacci(10*i);
    printf("Total: %d\n", total);
  }
*/
   return 0;
}
```