# LAB 2 Document

07302010063 何柯君

This document includes three parts: difficult parts in this lab and how did I solve these puzzles and some exercises answers.

**Part 1**

The difficult parts in this lab:

In most of the functions which I should fill in ,the coding job is not so hard if you understand the comments given by TAs , and what you should do is writing the code just step by step . But if you do not know how the virtual management working and how the whole process finish its job , you will get some error finally , and the debug job is very hard because the GCC complier do not give enough information .

**Part 2**

How I solve my puzzles:

Think hardly before coding. Because there are many new computer system concepts here , try to find it on the Internet. When error happens ,debug carefully and patiently and correct it finally. Also get some tips from my classmates.

**Part3**

Some exercises answers:

Exercise 1.

Assuming that the following JOS kernel code is correct, what type should variable x have, uintptr_t or physaddr_t?

```
1.      mystery_t x;
2.      char* value = return_a_pointer();
3.      *value = 10;
    x = (mystery_t) value;
```

Answer: Obviously ,this is a c function code . When running this function , we get the address of value , and it's a virtual address . The address of the variable defined in C programs can not be a physical address.

Exercise 2

1. What entries (rows) in the page directory have been filled in at this point? What addresses do they map and where do they point? In other words, fill out this table as much as possible:

Answer:

| Entry | Base Virtual Address | Points to (logically): |
|---|---|---|
| 1023 | 0xffc00000 | 0x0fc00000 |
| 1022 | 0xff800000 | 0x0f800000 |
| . | ? | ? |
| . | ? | ? |
| . | ? | ? |
| 2 | 0x00800000 | ? |
| 1 | 0x00400000 | ? |
| 0 | 0x00000000 | 0 |

2. After check_boot_pgdir(), i386_vm_init() maps the first four MB of virtual address space to the first four MB of physical memory, then deletes this mapping at the end of the function. Why is this mapping necessary? What would happen if it were omitted? Does this actually limit our kernel to be 4MB? What must be true if our kernel were larger than 4MB?

   Answer:

   The mapping is very import to the kernel address translation. Before the kernel turn on the paging translation ,the kernel uses the segment translations like it in the Question1. If we do not set up 4 MB space and delete the map , there will be a lot of problems. As a PDE can only map to 4MB physical memory space , the kernel space is limit to 4M. More PDEs will be used if our kernel was larger than 4MB.

3. (From Lecture 4) We have placed the kernel and user environment in the same address space. Why will user programs not be able to read or write the kernel's memory? What specific mechanisms protect the kernel memory?

   Answer:

   If the user programs are able to read or write the kernel’s memory , the kernel may be modified by the user when it is running . The operate system will break down. The U/S ,R/W in the PDE and PTE can protect the kernel memory.

4. What is the maximum amount of physical memory that this operating system can support? Why?

   In the kernel part code , we can use function i386_detect_memory(void) to get the maximum amount of physical memory . The variable” maxpa” represent it and its value is 32MB. But the maximum amount of physical memory can support 256MB for the map is from KERNBSE~2^32, the size of the physical space is 256MB.

5. How much space overhead is there for managing memory, if we actually had the maximum amount of physical memory? How is this overhead broken down?

Answer:

The maximum amount of physical memory is 256MB, so the number of the 4K page is 64K. Each of the entry structure is 4 byte , we need 256KB to store it.

**That is all my document here. Thank you for your kind reading!**