OS Lab2 Answers

Author: Chengzu Ou (14302010037)

Q1: Assuming that the following JOS kernel code is correct, what type should variable `x` have, `uintptr_t` or `physaddr_t`?

```
mystery_t x;
char* value = return_a_pointer();
*value = 10;
x = (mystery_t) value;
```

The type of variable `x` should be `uintptr_r` because it uses `*` operator on it.

Q2: 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:

Entry	Base Virtual Address	Points to (logically):
1023	0xffc00000	Page table for top 4MB of phys memory
1022	0xff800000	4MB kernel space
...	...	...
960	0xf0000000	All physical memory mapped at this address. (KERNBASE)
959	0xefc00000	Kernel stack. (KSTACKTOP)
958	0xef800000	The limit of user programming. (ULIM)
957	0xef400000	Intented for user to access VPT. (UVPT)
956	0xef000000	Mapping of struct Page in memory. (UPAGES)
955	0xeec00000	The top of user exception stack, and the limit of user to write. (UTOP, UENVS, UXSTACKTOP)
.	0xeebfe000	The top of user stack area. (USTACKTOP)
...	...	...
2	0x00800000	User program space. (UTEXT)
1	0x00400000	User temporary space to copy temporary data. (UTEMP)
0	0x00000000	Page table for bottom 4MB of physical memory(empty memory of user space).

Q3: 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?

We can not let user program to access (read or write) kernel space because that may disrupt kernel and cause system crashes.

We use paging translation to protect kernel space. By setting the permission of page entry in kernel space to kernel only, user will not be able to access this page.

Q4: What is the maximum amount of physical memory that this operating system can support? Why?

The operating system uses one page directory entry, `UPAGES`, with size 4MB to store all `PageInfo` structures. Every structure takes 8B space. Thus there can be in total 512K `PageInfo` structures, which means 512K physical pages. Every physical page is 4KB. So the maximum amount of physical memory that this OS can support is 2GB.

Q5: 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?

To manage the mapping between virtual memory and physical memory. First we need to store all `PageInfo` structures on memory, which costs 4MB. Then we need a 4KB `kern_pgdir`. And we also need to store current page table which is 2MB. Thus the total cost is 6MB + 4KB.

Q6: Revisit the page table setup in `kern/entry.S` and `kern/entrypgdir.c`. Immediately after we turn on paging, EIP is still a low number (a little over 1MB). At what point do we transition to running at an EIP above KERNBASE? What makes it possible for us to continue executing at a low EIP between when we enable paging and when we begin running at an EIP above KERNBASE? Why is this transition necessary?

In `entry.s`, there is an instruction `jmp *%eax`. This will set `eip` to the value stored in `eax` which is larger than KERNBASE. This completes the transition from low number to above KERNBASE.

The reason why we could continue executing at a low `eip` after enabling paging and before runing at an `eip` above KERNBASE is that we also map the lowest address(from 0 to 4MB) in virtual memory to the lowest address in physical memory in `entry_pgdir` table. Thus when access virtual address within [0, 4MB], it will mapping to the same address in physical memory.