CSL373/CSL633 Minor 1 Exam
Operating Systems
Sem II, 2013-14

Answer all 6 questions Max. Marks: 32

Unix System Calls

1. What is the total number of processes at the end of the execution of the following program? Assume there is one process in the beginning that starts running at main. Also, assume that all system calls succeed.

```
main() {
  fork();
  fork();
  fork();
}
```

Explain. [3]

Ans: 2^3 = 8.

fork#	#processes
1	2
2	4
3	8

First fork is called by the first process, it generates one more process (or doubles). Both these processes call second fork and doubles the number of processes to 4...and so on.

2. Consider the following program:

```
main() {
    int fd;
    fd = open(“outfile”, O_RDWR)
    fork();
    write(fd, “hello”, 5);
    exit();
}
```

Assume all system calls finish successfully on a uniprocessor system. Also, assume that a system call cannot be interrupted in the middle of its execution. What will be the contents of the “outfile” file, after all processes have successfully exited? Explain briefly. [3]

hellohello

Fork will create two processes with the same *fd* pointing to the given file.
Two “hello” writes will be observed by the same *fd* (seek value or pos will be shared). And because of the given assumption, interleaving/mixing of these two writes is not possible.

3. Now, consider the following program:

```
main() {
    int fd;

    fork();
    fd = open(“outfile”, O_RDWR)
    write(fd, “hello”, 5);
    exit();
}
```

Assume all system calls finish successfully on a uniprocessor system. Also, assume that a system call cannot be interrupted in the middle of its execution.

Notice that open is now called after fork (not before fork as in the previous question). What will be the contents of the “outfile” file, after all processes have successfully exited? Explain briefly. [3]

hello

Value of fd will be same for both the process.
But open file entry corresponding to both this fd are different. ie. there are two copies of write offset.
After open system call, write offset in each of the opened file entry is initialized to zero.
And second write will over write the first one.

Virtual Memory

4. Consider the following program header of an ELF executable file a.out:

LOAD: offset 0x00001000
vaddr 0x40100000
paddr 0x00100000//removed this line
align 2**12
filesize 0x0000b596
memsize 0x000126fc
flags rwx

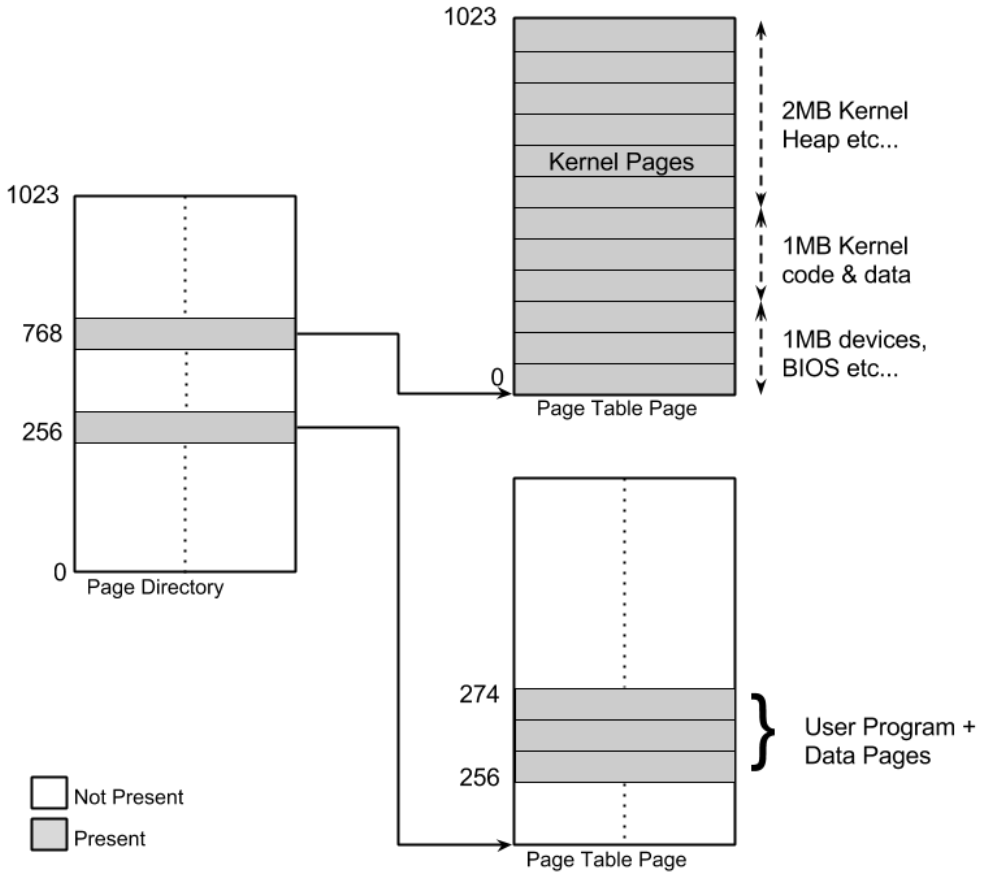
4a. Assume that this executable is loaded using the exec("a.out", …) system call on 32-bit Linux. Also, assume that the Linux kernel is mapped starting at virtual address 0xc0000000. Draw the layout of the virtual address space of the process just after successful completion of the exec() system call. Indicate the sizes, and the contents of the memory regions, wherever possible. [6]

Answer:

0xffffffff
... Kernel (Access: kernel only)
0xc0000000
... Heap(why?)
...
... Heap(why?)
0x401126fc
… zero filled memory (user)
0x4010b596
... Program (user)
0x40100000
… (Either unmapped: or Devices/VGA/Bios mapped but access to kernel only).
…
… Unmapped (atleast will help in catching null pointer dereference at kernel level) or access to kernel only.
0x00000000

4b. Assume that the operating system is using paging to map the pages of the executable on

x86 using a two-level page table. Also assume that it is *not* using large pages --- i.e., it is only using 4KB pages to map the process and kernel's address space. Assume that the size of the physical memory is 4MB and it is entirely mapped in the kernel address space (starting at 0xc0000000). Also, assume that the kernel's code and data takes 1MB of physical memory space (start at physical address 0x100000). Draw the page table and indicate the values stored in them. Especially, say which entries will be present and where they will be mapped (what are the likely values of these entries). Assume all space is mapped with rwx privileges (but of course, differing in user/kernel privileges). [10]



One page table has 4KB/4Bytes = 1024 Page table entries (PTE)

- 0x40100000 -> [0100 0000 00] [01 0000 0000] [0000 0000 0000]
  PD Index PT Index Last 12 bits
  256th entry 256th entry
- 0x401126fc -> 256th Page directory entry, 274 (0x112) Page table entry index
- 0xc0000000 = 768th entry

Assembly Code / Calling Conventions

5. Consider the code in bootasm.S (for the bootsector). At line 8468, the code uses the “call” instruction to branch into the C function called “bootmain” (defined at line 8517 in bootmain.c). Would it have been okay to instead use the “jmp” instruction like this:

```
8467    movl    $start, %esp
8468    jmp     bootmain
```

If this is okay, explain why. If not, explain why not? [3]
Yes: bootmain shouldn’t return. So it is okay to assume bootmain as noreturn function and optimize out call statement by replacing with jmp

No, It makes it difficult to debug if there exists a bug: bootmain returns.

xv6 paging

6. Consider lines 1049-1051 that enable paging in the hardware. As soon as paging is enabled (at line 1051), the address space changes. i.e., some addresses that were valid previously are no longer valid. Similarly, some (virtual) addresses that were invalid previously are now valid. Assume that the size of the physical memory is 64MB.

```
pde_t entrypgdir[NPDENTRIES] = {
  // Map VA's [0, 4MB) to PA's [0, 4MB)
  [0] = (0) | PTE_P | PTE_W | PTE_PS,
  // Map VA's [KERNBASE, KERNBASE+4MB) to PA's [0, 4MB)
  [KERNBASE>>PDXSHIFT] = (0) | PTE_P | PTE_W | PTE_PS,
};
```

KERNBASE is 0x80000000.
V2P(X) = X - KERNBASE.

Address range	Before paging	After paging	Example symbols
[0,4MB)	VALID	VALID	_start
[4MB,physmem)	VALID	INVALID	
[KERNBASE, KERNBASE+4MB)	INVALID	VALID	entry,stack,main,entrypgdir

a. Give an example of a (virtual) address that was invalid before enabling paging at line 1051 (i.e., accessing it would have generated an exception), but would be valid after enabling paging. [2]

Ans: Any location between [KERNBASE, KERNBASE+4MB)
Wrong answer: all other values are wrong.

Example symbols located between [KERNBASE,KERNBASE+4MB): entry,stack,main,entrypgdir

Wrong symbols: _start
Why you will get exceptions before: Accessing value outside physical memory.

Why you won't get exceptions after: Page table mapping exists.

b. Give an example of a (virtual) address that was valid before enabling paging at line 1051 (i.e., accessing it would have resulted in some valid data), but would be invalid after enabling paging i.e., accessing that address will generate an exception). [2]

Ans: Any valid physical address other than [0,4MB) and [KERNBASE,KERNBASE+4MB)

Wrong ans: symbols between [KERNBASE,KERNBASE+4MB): entry,stack,main
Wrong ans: symbols between [0,4MB): _start

Why you wont get exceptions before:
valid physical memory location(assuming physical memory is more than 4MB)

Why you will get exceptions after:
Only [0,4MB) and [KERNBASE,KERNBASE+4MB) are valid.