

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]

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]

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]

Virtual Memory

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

```
LOAD: offset 0x00001000
      vaddr 0x40100000
      paddr 0x00100000
      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]

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]

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]

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.

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]

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]

