Department of Electrical Engineering and Computer Science

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

6.828 Fall 2008

Quiz I

All problems are open-ended questions. In order to receive credit you must answer the question as precisely as possible. You have 80 minutes to finish this quiz.

Write your name on this cover sheet AND at the bottom of each page of this booklet.

Some questions may be harder than others. Read them all through first and attack them in the order that allows you to make the most progress. If you find a question ambiguous, be sure to write down any assumptions you make. Be neat. If we can't understand your answer, we can't give you credit!

THIS IS AN OPEN BOOK, OPEN NOTES EXAM.

Please do not write in the boxes below.

I (xx/12)	II (xx/18)	III (xx/16)	IV (xx/12)	V (xx/8)
VI (xx/16)	**VII (xx/8)**	**VIII (xx/4)**	**IX (xx/6)**	**Total (xx/100)**

Name:

I System Call Arguments

1. [6 points]: Given a user-provided pointer to a buffer in user memory (such as the argument of a system call), explain what checks and translations the JOS kernel must do to ensure that it can safely read or write the buffer memory. Do not assume or rely on the existence of a function in the kernel that does these checks and translations for you.

2. [6 points]: Given a user-provided pointer to a buffer in user memory (such as the argument of a system call), explain what checks and translations the xv6 kernel must do to ensure that it can safely read or write the buffer memory. Do not assume or rely on the existence of a function in the kernel that does these checks and translations for you.

II File System

3. [8 points]:

Describe two situations in which a power failure during a system call could leave different parts of the xv6 file system metadata inconsistent with each other. By "metadata" we mean the file system's on-disk data structures: the superblock, inodes, indirect blocks, the block in-use bitmap, and directory contents. A power failure halts the CPU and stops the disk after completion of the current sector read/write request (if any). Be sure to indicate the line number after which you assume the power failure occurs for each of the two situations, and the nature of the resulting inconsistencies.

4. [10 points]: Alice is worried that her xv6 kernel might panic at line 3596 because it runs out of struct bufs, of which there are only 10 (line 3529). Since she is mostly concerned with reliability, and does not care about the performance benefits of using these buffers as a cache, she decides to replace bget() and brelse() as follows:

```
static struct buf* bget(uint dev, uint sector) {
    struct buf *b = kalloc(sizeof(*b));
    memset(b, 0, sizeof(*b));
    b->flags |= B_BUSY;
    b->dev = dev;
    b->sector = sector;
    return b;
}

void brelse(struct buf *b) {
    kfree(b, sizeof(*b));
}
```

She modifies kalloc() to allow any size (rather than just multiples of 4096). Assuming that kalloc() never fails, describe a concrete correctness problem she will run into with this code.

III Paging vs. segmentation

JOS sets up the x86 page table to contain PTE entries for all kernel code and data structures (at a high virtual address, KERNBASE = 0xF00000000), even when running a user-space environment. The code and data for the user-space environment is mapped at low virtual memory addresses.

5. [4 points]: Explain why it is safe to keep sensitive kernel data structures mapped at 0xF0000000 while executing user-level code, which should not be able to access those kernel data structures.

6. [6 points]: Explain why JOS needs the kernel mapped while executing user-level code.

The xv6 kernel lives at low addresses, starting at 0x00100000. xv6 user-level code resides at memory addresses starting from zero, so a program with more than 1MB of instructions will use addresses that overlap with the kernel's addresses.

7. [6 points]: Explain how the processor allows both the xv6 kernel and user-level processes to store instructions at the same addresses.

IV Traps and interrupts

Recall that gcc calling conventions allow any function to trash the "caller-saved" registers %eax, %ecx, %edx, so that they may be different upon return.

8. [6 points]: xv6's system call stubs (sheet 68) are called by user-space C code as if they were ordinary C functions, so all callers are prepared for the stubs to not preserve caller-saved registers. The stubs invoke the INT instruction, which vectors into kernel code that shortly jumps to alltraps (line 2456). alltraps and trapret save and restore all registers to and from a struct trapframe (line 0477), including %eax, %ecx, and %edx, using pushal and popal, even though gcc does not expect any of these registers to remain the same after return from the system call stub.

Suppose we wanted to reduce the size of struct trapframe by not saving %ecx and %edx in alltraps. It turns out that this would cause problems. Why? Give an example of something that would break.

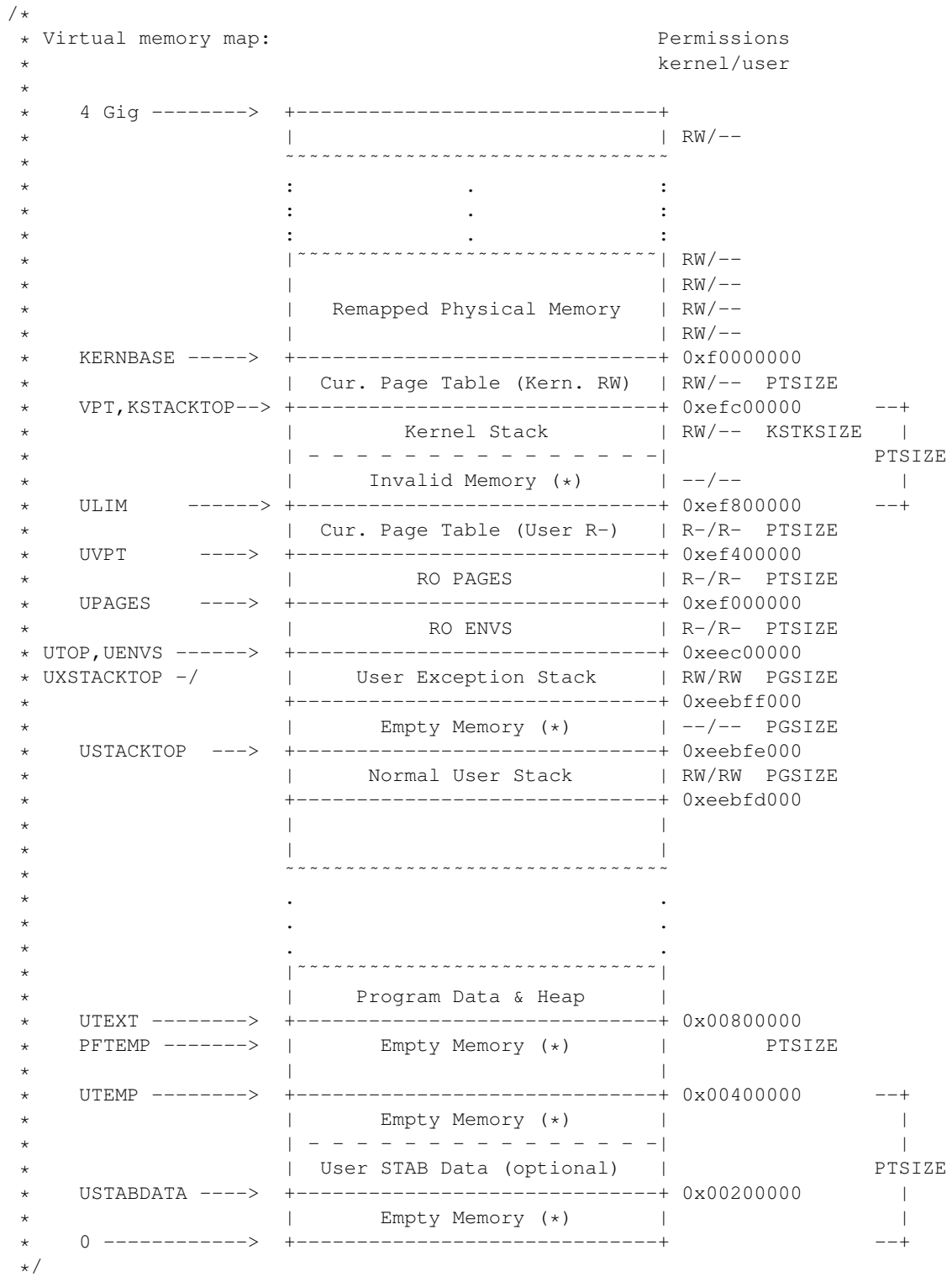
9. [6 points]: In lab 3, the JOS kernel treats user-mode page faults and errors in system calls differently. For page faults, the kernel destroys the user environment, but for system calls that encounter an error, the kernel sets an error code in the %eax register and resumes the environment at the instruction following the `INT`. Explain what would happen if the JOS kernel, when handling a page fault from user space, were to set an error code in %eax and resume the environment at the instruction following the one that caused a page fault.

V Virtual vs. physical memory

Ben wants to add support for a graphics card in JOS. The graphics card provides a 1024-by-1024-pixel display, which appears as 1MB of memory at physical address 0xF0000000, with each byte of that memory range corresponding to one pixel on the screen. The graphics card memory is not part of the physical memory detected by i386_detect_memory().

Assuming pixel value 0 corresponds to black, provide C code for clearing the screen to all-black, to be executed just after JOS turns on paging (i.e. in i386_init() just after i386_vm_init returns). The next page includes a copy of `inc/memlayout.h` for your reference.

10. [8 points]: Your code:


/*
* Virtual memory map:
Permissions
kernel/user
4 Gig
RW/--
Remapped Physical Memory
RW/--
KERNBASE
0xf0000000
Cur. Page Table (Kern. RW)
RW/-- PTSIZE
VPT,KSTACKTOP
0xefc00000
Kernel Stack
RW/-- KSTKSIZE
Invalid Memory (*)
--/--
ULIM
0xef800000
Cur. Page Table (User R-)
R-/R- PTSIZE
UVPT
0xef400000
RO PAGES
R-/R- PTSIZE
UPAGES
0xef000000
RO ENVS
R-/R- PTSIZE
UTOP,UENVS
0xeec00000
UXSTACKTOP -/
User Exception Stack
RW/RW PGSIZE
0xeebff000
Empty Memory (*)
--/-- PGSIZE
USTACKTOP
0xeebfe000
Normal User Stack
RW/RW PGSIZE
0xeebfd000
Program Data & Heap
UTEXT
0x00800000
PFTEMP
Empty Memory (*)
PTSIZE
UTEMP
0x00400000
Empty Memory (*)
User STAB Data (optional)
USTABDATA
0x00200000
Empty Memory (*)
0
*/

Name:

VI Kill

Here is process X running on xv6:

```
main()
{
        while(1) {
        }
}
```

As you can see, process X is in an infinite loop. Another process Y uses the kill() system call to terminate process X.

11. [8 points]: For this question, assume the xv6 system has a single CPU. Is it possible for process X to execute any more user-space instructions after process Y's kill() returns? Explain. If "yes," what will finally cause X to stop executing user-space instructions?

12. [8 points]: For this question, assume the xv6 system has two CPUs. Is it possible for process X to execute any more user-space instructions after process Y's kill() returns? Explain. If "yes," what will finally cause X to stop executing user-space instructions?

VII CLI

13. [8 points]: What would go wrong if you replaced pushcli()'s implementation (xv6 sheet 14) with just cli(), and popcli()'s implementation with just sti()?

VIII Fork

Here's a copy of sys_fork() from sheet 28:

```
int
sys_fork(void)
{
  int pid;
  struct proc *np;

  if((np = copyproc(cp)) == 0)
    return -1;
  pid = np->pid;
  np->state = RUNNABLE;
  return pid;
}
```

Explain what could go wrong if the code looked like this:

```
int
sys_fork(void)
{
  struct proc *np;

  if((np = copyproc(cp)) == 0)
    return -1;
  np->state = RUNNABLE;
  return np->pid;
}
```

14. [4 points]: Your answer:

IX 6.828

We'd like to hear your opinions about 6.828, so please answer the following questions. (Any answer, except no answer, will receive full credit.)

15. [2 points]: How could we make the ideas in the course easier to understand?

16. [2 points]: What is the best aspect of 6.828?

17. [2 points]: What is the worst aspect of 6.828?

End of Quiz

Name:

6.828 Operating System Engineering

Fall 2012