Department of Electrical Engineering and Computer Science

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

6.828 Fall 2011

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/28)	II (xx/20)	III (xx/44)	IV (xx/8)	Total (xx/100)
25	10	26	8	69

mean 67
median 66

Name:

I General questions

1. [8 points]: Unix's API is carefully designed so that programs compose easily. As an example, `write` and `read` don't take an offset as argument, but instead the kernel maintains an offset for each file descriptor. Give a shell command that illustrates how this unusual API simplifies composing programs and explain briefly why.

cat some_file | wc -l => # lines in some_file

(5) cat always writes to std out, wc always reads from stdin (if no file arg given)
The two programs keep their original behavior when combined because the Unix API (pipes) takes care of linking cat's stdout to wc's stdin

2. [12 points]: In xv6, `wakeup` must scan the entire `ptable` to find the processes that are sleeping on the specified channel. Ben proposes to change xv6 to use condition variables to avoid this scan. His condition variables have the following structure:

```
struct condvar {
  struct proc *waiters;
};
```

For each channel there is a separate condition variable, which holds the list of processes waiting on that condition variable. Wakeup wakes up only those processes:

```
void
cv_wakeup(struct condvar *cv)
{
  acquire(&ptable.lock);
  struct proc *p = cv->waiters;
  while (p) {
    struct proc *next = p->cv_next;
    p->cv_next = 0;
    p->state = RUNNABLE;
    p = next;
  }
  cv->waiters = 0;
  release(&ptable.lock);
}
```

(continued on next page)

Name:

Complete the implementation of `cv_sleep`. Your implementation should not call `sleep`.

```
void
cv_sleep(struct condvar *cv, struct spinlock *lk)
{
  if(proc == 0)
    panic("sleep");
  if(lk == 0)
    panic("sleep without lk");
  if (lk != &ptable.lock) {
    acquire(&ptable.lock);
    release(lk);
  }
  p->state = SLEEPING;
  p->cv_next = cv->waiters;
  cv->waiters = p;
  sched();

  if (lk != &ptable.lock) {
    release(&ptable.lock);
    acquire(lk);
  }
}
```

(12)

3. [8 points]: `swtch` in xv6 doesn't explicitly save and restore all fields of **`struct`** `context`. Why is it okay that `swtch` doesn't contain any code that saves `%eip`?

Because eip is implicitly saved on the stack before the call to swtch() and restored by the last instruction of swtch(), namely ret.

(8)

Name:

II File systems

4. [10 points]: Albert has a computer running xv6. He has written his own application, which forks many processes, all of which write large files. Every once in a while, Albert's computer panics with "bget: no buffers" (line 3946). He thinks to himself that, since buffers are only used for short periods of time, it would be better for `bget()` to sleep until a buffer is free. So he changes the panic at the very end of `bget()` to:

```
sleep(&bcache, &bcache.lock); // new code
goto loop;                    // new code
```

And adds a wakeup before the release on the last line of `brelse()`:

```
wakeup(&bcache);          // new code
release(&bcache.lock);    // existing code
```

After this change, Albert's computer occasionally stops working: no panic, but no forward progress either. He calls `procdump()` from `gdb` and sees that every process (other than process 1) is waiting in his new sleep in `bget()`; the complete stack for each process is:

```
Function  Line
sleep     2525
bget      the new call to sleep
bread     3956
readsb    4282
balloc    4311
bmap      4629
writei    4769
filewrite 5176
sys_write 5285
syscall   3283
```

Process 1 is sleeping in `wait()`. Albert's computer has only one core. Albert is running a modified version of xv6 with logging disabled: `log_write()` just calls `bwrite()`, and `begin_trans()` and `end_trans()` do nothing.

Explain why the processes never wake up from the sleep in `bget()`. (Hint: look at `bmap`.)

bmap has to allocate blocks on demand so if many processes are writing large files then soon the kernel will run out of buffer space at which point everyone waits for blocks to be freed but no one is freeing them ✓

(10)

Name:

5. [10 points]: Albert decides to take a different approach: in order to avoid running out of buffers, he'll allocate them with `kalloc()` as needed. Here's his new `bget()` and `brelse()`:

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

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

Again, Albert's computer has only one core, and has logging disabled.

Albert's computer seems to work for a while, but eventually panics with "freeing free block" from `bfree()` at line 4342.

Explain what is going on.

By using kalloc(), bget() is bypassing balloc() whose job it is to mark a disk block as being in use in the bitmap block. This way all blocks are marked free in the bitmap block (unless explicitly allocated with balloc) and when bfree() is called on such a block a panic occurs. X

0

grading scheme gives different answer but I think mine still has merit

Name:

III JOS memory layout and traps

Currently, the JOS kernel reserves part of every user environment's address space for itself. Ben Bitdiddle (who fights for the user environments) decides to modify his lab 3 implementation so user environments have control over (nearly) their entire 4GB virtual address space.

6. [8 points]: In regular JOS, where the kernel and user code share the same address space, what prevents user code from reading or writing sensitive kernel data?

The permission bits (lowest 12 bits) of each PTE and PDE are interpreted by the MMU hardware and the entries for the kernel mappings don't have PTE_U set. ✓

8

Ben separates the user and kernel address spaces. He constructs the kernel page directory, `kern_pgdir`, like usual, but modifies `env_setup_vm` to create a nearly empty page directory for each new environment instead of mapping the kernel in to each environment.

Ben's plan is to switch to `kern_pgdir` whenever there's a trap. Unfortunately, the x86 provides no convenient way to switch address spaces when taking a trap. Ben solves this with a *trap trampoline*. He reserves a few pages for kernel use at the top of each environment's address space. He puts all of his `trapentry.S` code (`TRAPHANDLER` declarations and `alltraps`) on one of these pages so that the CPU can jump to the trap handler entry point without switching address spaces.

7. [8 points]: Ben also maps the kernel stack in to each user environment. Why is this necessary?

The int instruction will load a new ~~task~~ stack from the TSS and start pushing the trapframe on that stack. This happens before any additional instructions (trap handler) can be executed. This means you won't have the opportunity to switch page directories. ✓

8

Name:

Ben modifies `alltraps` (which runs from the trampoline page) to switch to the kernel page directory before calling the `trap` (up to the `lcr3`, this should be equivalent to the `alltraps` you wrote):

```
_alltraps:
  pushl %ds                 # Build trap frame
  pushl %es
  pushal
  movl $GD_KD, %eax         # Load kernel segments
  movw %ax, %ds
  movw %ax, %es
  pushl %esp
  lcr3 PADDR(kern_pgdir)    # NEW: Switch to kernel address space
  mov $trap, %eax           # Call trap
  call *%eax
```

8. [10 points]: Louis Reasoner thinks doing `lcr3 PADDR(kern_pgdir)` from `alltraps` won't work because, in Ben's design, `kern_pgdir` isn't accessible in the user environment's address space. In fact, it *will* load the kernel page directory. Why?

When kern_pgdir gets setup, it is mapped at UPAGES which is readable by user evironments (has PTE_U set)

0 X

9. [8 points]: When Ben runs his code, he finds that it executes the `lcr3`, but fails to execute the next instruction (it never sets `%eax` to the address of `trap`). Why and how can he fix this?

The trampoline page is not mapped in kern_pgdir so he needs to ~~#~~ map it at the same location

8

10. [10 points]: Ben fixes this problem and then discovers that he needs to rewrite `sys_cputs` because it can no longer read its string argument directly from the current address space. Describe how Ben can fix `sys_cputs`.

The pages where those arguments are stored, ~~dp~~ ~~not mapped~~ are not mapped in kern_pgdir

2

Name:

IV 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.)

11. [2 points]: This year we posted the complete draft of the xv6 commentary at the beginning of the semester. Did you find the chapters useful? What should we do to improve them?

The xv6 commentary was very helpful, especially in the context of self study. To improve the commentary, please include answers to the questions posed at the end of every chapter

12. [2 points]: This year we introduced code reviews. Did you find them useful? What should we do to improve them?

CRs are very useful

13. [2 points]: What is the best aspect of 6.828?

The labs are very interesting because you get to build out a real kernel

8

14. [2 points]: What is the worst aspect of 6.828?

It's hard and requires lots of work :-

End of Quiz

Name: