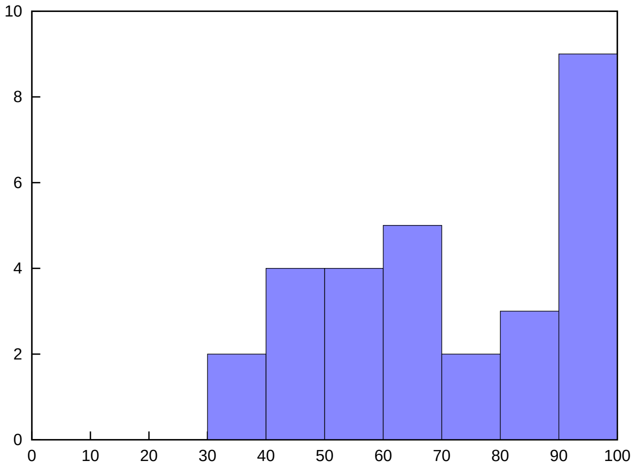
Department of Electrical Engineering and Computer Science

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

6.828 Fall 2011

Quiz I Solutions

Mean 71 Median 69 Std. dev. 20.2



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.

Answer: `(echo Your files: ; ls) > out`
The `ls` program does not have to do any work to figure out where in file `out` to write; it would if `write()` took an offset argument.

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)

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);
  }

  proc->cv_next = cv->waiters;
  cv->waiters = proc;
  proc->state = SLEEPING;
  sched();

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

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`?

Answer: The call to `swtch()` saves `%eip` on the stack, which is constructed in a way such that it follows the layout of **`struct`** `context`. The return from `swtch()` restores `%eip`.

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`.)

Answer:

`bmap()` needs to use two buffers at the same time if it is adding a new block to a large file, one to hold the file's indirect block and one (in `balloc()`) to hold the superblock while allocating a new block. Suppose there are only 10 slots in the buffer cache, and ten processes (writing ten distinct files) have each read their file's indirect block. At that point there is no free buffer into which any of them can read the superblock, so they will all wait in Albert's new sleep.

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.

Answer:
Albert's new scheme lets multiple processes use the same block at the same time (it has nothing equivalent to `B_BUSY`). This allows two processes to allocate the same block or the same inode. If two processes allocate the same block for different files, xv6 will first notice the problem when both are deleted: the second delete will free a freed block.

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?

Answer: The page table entries for the kernel portion of the address space do not have the 'U' bit set, so the paging hardware disallows access to these pages when the CPU is running user code.

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?

Answer: The CPU needs somewhere to push the trap-time state (EIP, CS, etc) when it takes the interrupt. This happens before Ben has a chance to switch to the kernel address space, hence the kernel stack needs to be mapped in to each user environment.

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?

Answer: `lcr3` takes a physical address so it doesn't matter what the current address space is.

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?

Answer: Ben didn't map the trampoline page in the kernel address space, so as soon as he switches to `kern_pgdir`, the instruction at the EIP will no longer be the `mov` at the end of `alltraps`. Ben can fix this by mapping the trampoline pages at the same location in `kern_pgdir`.

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`.

Answer: For each page that the argument string spans, `sys_cputs` needs to translate that page to a physical address using the user environment's page directory, and then read from the page that maps that physical page in the kernel's address space.

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?

Answer: Yes, they were very useful. They could use more figures.

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

Answer: Yes. Reading other people's code was as helpful as getting reviews.

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

Answer: Labs.

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

Answer: Bugs. Too much code is given.

End of Quiz