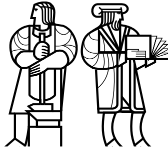


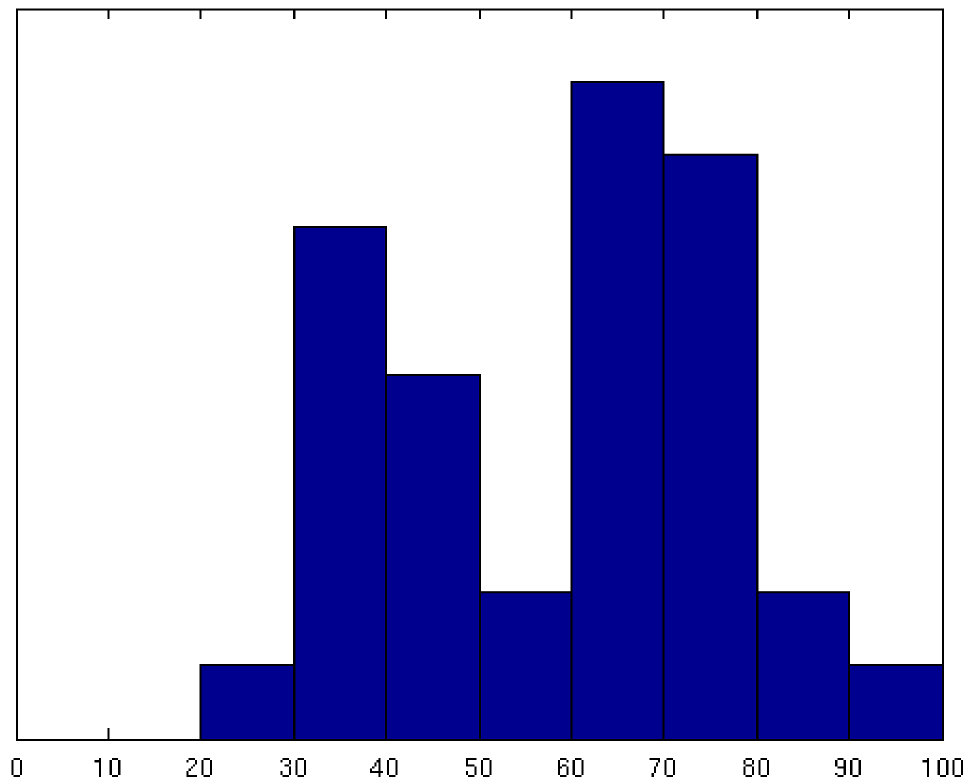
Department of Electrical Engineering and Computer Science

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

6.828 Fall 2007

Quiz I Solutions

The mean score was 59, and the median was 64.



I Boot

1. [4 points]: Explain how xv6 ensures that the boot loader (bootasm.S and bootmain.c) copies the kernel from disk to memory in a way that avoids overwriting the boot loader's own instructions or stack.

Answer: The xv6 kernel is linked at 0x100000, so the ELF header information will cause bootmain() to only overwrite memory at 0x100000 and above. The boot loader's instructions are at 0x7c00, and its stack is just below that, so they won't be overwritten by the kernel.

2. [5 points]: Bugs Bunny, after reading xv6's `userinit()` (sheet 17), claims that the start-up of the first process could be made more efficient by arranging for `swtch()` to directly enter the process. Specifically he suggests changing line 1774 to read

```
p->context.eip = (uint) p->mem; // swtch() directly to start of initcode
```

Why is this change a bad idea?

Answer: The change affects what swtch() will do when scheduler() eventually calls it: swtch() will cause the CPU to execute initcode's instructions, but with the kernel's privileges and in the kernel's address space. Perhaps the most immediate problem will be that proc_table_lock won't be released (ordinarily that swtch() jumps to forkret(), which releases the lock).

II Processes and Address Spaces

3. [5 points]: JOS has a single kernel stack, while xv6 has one kernel stack per process. Explain how this restricts what the JOS kernel can do, and mention a specific feature of xv6 that would be difficult in the JOS kernel because it has only the one stack.

Answer: xv6 can have system calls that block in the middle to wait for I/O, for example to wait for the various disk reads required to look up a file pathname. The process's kernel stack saves the intermediate state of the system call while it blocks. A JOS system call cannot block in the middle, it can only block at the very end.

The Intel x86 allows the stack segment (SS) and data segment (DS) to differ. This separation allows an operating system to permit a process to expand both segments independently as needed, without worrying about the stack running into the heap. xv6, however, always makes the two segments the same (for example, see `userinit()` on sheet 17). As a result, an xv6 process's stack cannot expand beyond the memory initially allocated for it.

4. [5 points]: Explain why it would be awkward for xv6 to give a process different data and stack segments (i.e. have DS and SS refer to descriptors with different BASE fields).

Answer: It's common for C code to create pointers to data on the stack, as well as pointers to data in the heap. Functions such as strlen() that operate on pointers would need to know whether to use the pointers with DS or SS. So either the compiler would have to be modified to use 48-bit pointers (32 bits of offset plus a segment selector), or the programmer would have to keep track of this information by hand for each pointer.

III Kernel Entry

5. [5 points]: When a user-space process executes INT, INT switches to a stack specified in the TSS. JOS sets up the TSS to point to a stack in kernel memory. What might go wrong in JOS if the INT instruction *didn't* change stacks?

Answer: The environment might set its ESP to an invalid address before the INT, which would cause the kernel to fault. The environment might set its ESP to point to a protected kernel data structure, so that the INT would save EFLAGS &c on the kernel's data. If JOS had system calls like read() that copied data to user space, then the environment might be able to trick the kernel into executing arbitrary code by read()ing data onto the stack: the data could overwrite one of the kernel's saved return EIPs with a pointer to instructions in the environment's memory.

6. [12 points]: Explain why the `pushl` at xv6 line 6610 is needed.

Answer: Ordinarily that space on the stack would hold the return program counter from the call to the C library system call routine. The kernel expects something to be there on the stack when it is calculating the address from which to fetch an argument, for example on line 2696. If the pushl were deleted, argint() would return the wrong argument.

7. [12 points]: xv6 defines two structures that hold saved registers for a process: `struct trapframe` on sheet 05, and `struct context` on sheet 15. Explain a situation in which a single process will have *three* sets of saved registers.

Answer: A process makes a system call (saving user registers in a trapframe), a clock interrupt happens just as trap() is returning (saving kernel registers for the same process in another trapframe lower on the same stack), and this second call to trap() calls yield() and thus swtch() (saving kernel registers for the same process in a struct context).

IV Locking

8. [5 points]: xv6's `acquire()` turns off interrupts at line 1431 only if `nlock` is zero. `release()` turns them back on only if `nlock` (after decrementing) is zero (at line 1466). What could go wrong if `acquire()` *always* turned off interrupts, and `release()` *always* turned interrupts on?

Answer: If a process acquires two locks, and then releases one of them, this modification would cause interrupts to be turned on. In that case there is a risk that an interrupt routine (such as ide_intr()) could try to acquire the lock that is still held. That would produce a deadlock (actually a panic in acquire()).

Suppose you swapped xv6 lines 1869 and 1870, so that `yield()` looked like this:

```
void
yield(void)
{
  cp->state = RUNNABLE;
  acquire(&proc_table_lock);
  sched();
  release(&proc_table_lock);
}
```

9. [8 points]: Would anything go wrong? Explain why or why not.

Answer: The scheduler() on a different CPU might see that the process is RUNNABLE and run it. Then two CPUs would be executing on the process's kernel stack, overwriting each others' variables and return EIPs, which is likely to cause trouble.

V Context Switch

When one xv6 process switches to another, two calls to `swtch()` are involved: one from `sched()` to `scheduler()`, and one from `scheduler()` to `sched()`. Bugs Bunny is obsessed with `swtch()` and suggests eliminating one of the calls: `sched()` could call a variant of `scheduler()` directly. The new code:

```
// caller must hold proc_table_lock.
void
sched(void)
{
  scheduler2();
}

// caller must hold proc_table_lock.
void
scheduler2(void)
{
  struct proc *p;
  struct proc *from = cp; // remember who we are
  int i;

  for(;;){
    for(i = 0; i < NPROC; i++){
      p = &proc[i];
      if(p->state != RUNNABLE)
        continue;

      cp = p;
      setupsegs(p);
      p->state = RUNNING;
      swtch(&from->context, &p->context);

      // a return from swtch() means some other call to
      // scheduler2() decided to run us.
      cp = from;
      setupsegs(cp);
      return;
    }

    release(&proc_table_lock);
    acquire(&proc_table_lock);
  }
}
```

10. [12 points]: This new `swtch()`-less `sched()` works most of the time, but not always. What's likely to go wrong?

Answer: Suppose a process sleep()s waiting for an IDE interrupt. The CPU on which it was running will use the process's kernel stack to execute scheduler2(). When the IDE interrupt occurs and marks the process RUNNABLE, the scheduler on a different CPU may run the process. Then there will be two CPUs using the same kernel stack, which is sure to cause trouble.

11. [5 points]: What's the reason for the `release()` and immediate re-`acquire()` at the end of Bugs's `scheduler2()`?

Answer: Without the release() and acquire(), interrupts would always be off when there are no RUNNABLE processes, permanently preventing delivery of (for example) keyboard and IDE interrupts. Another problem is that, if there is just one process RUNNING, it may spin forever next time it tries to acquire process_table_lock since the lock may be permanently held by an idle CPU.

VI Sleep/Wakeup

In `exit()` (sheet 20), suppose you swapped lines 2023 and 2026 and changed the `wakeup1()` to `wakeup()`, so that the resulting code looked like:

```
// Parent might be sleeping in proc_wait.
wakeup(cp->parent);
acquire(&proc_table_lock);
```

12. [8 points]: What is likely to go wrong as a result of this change to `exit()`?

Answer: Suppose the parent is in wait() just before the sleep(), and the child's exit() calls wakeup(). Then the parent will miss the wakeup and sleep forever.

VII File System

The policy that `bget()` line 3576 and `brelse()` enforce is that at most one process at a time can use a buffer. One reason is to prevent other processes from using a buffer between the time it is allocated in `bget()` and when the IDE driver finishes reading valid data from the disk into the buffer (and sets `B_VALID`).

13. [8 points]: Suppose `bget()` and `brelse()` were modified to let multiple processes use a `B_VALID` buffer at the same time, while ensuring that a buffer is only re-used for a different sector when the buffer is not in use. Please describe a specific problem that would arise.

Answer: The file system code uses the one-at-a-time rule to provide mutual exclusion for code that reads and then updates the file system's own on-disk data structures. For example, if two processes call balloc() at the same time and bget() allowed them to use the same buffer at the same time, they might both allocate the same free block. It would be a serious error for the same disk block to be used by two different files.

VIII 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.)

14. [2 points]: What ideas in the course have been the hardest to understand?

Answer: The difference between virtual and physical memory.

15. [2 points]: What is the best aspect of 6.828?

Answer: The labs.

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

Answer: Debugging the labs.

End of Quiz

MIT OpenCourseWare
http://ocw.mit.edu

6.828 Operating System Engineering
Fall 2012

For information about citing these materials or our Terms of Use, visit: http://ocw.mit.edu/terms.