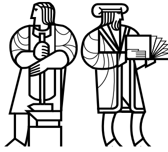


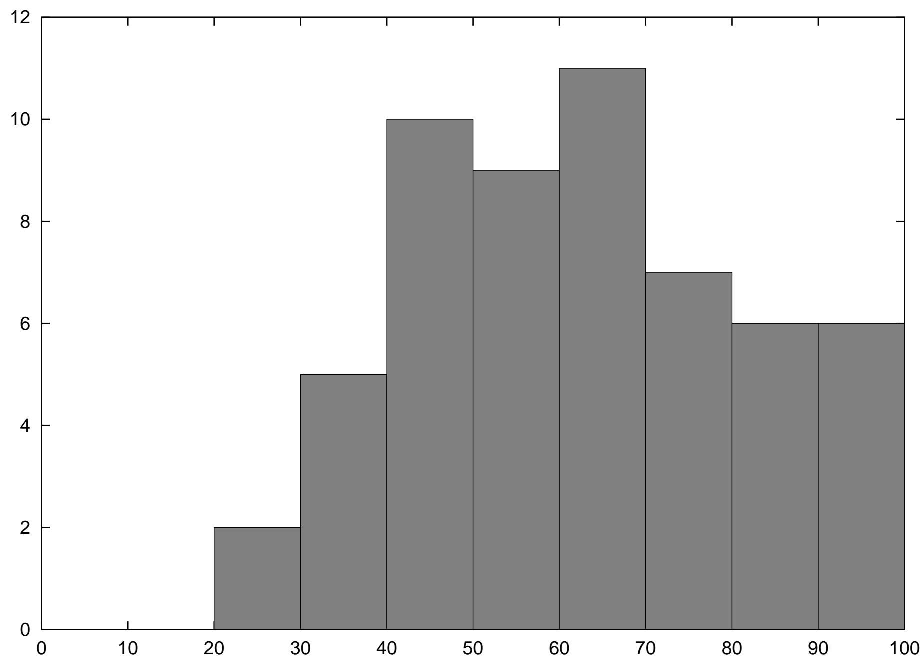
Department of Electrical Engineering and Computer Science

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

6.828 Fall 2008

Quiz I Solutions

The mean score was 62, median was 61, and standard deviation 18.8.



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.

Answer: The JOS kernel does not need to translate the address, since the environment's memory is mapped into the kernel at the same addresses that the environment itself uses. JOS needs to make four safety checks before it can dereference a pointer supplied by the environment without fear of incurring a page fault or reading or writing outside the environment's memory. 1) The PTE_U flag must be set in the corresponding PTE (equivalently, the address must be less than UTOP). 2) The PTE_P bit must be set in the PTE. 3) If JOS needs to write through the pointer, the PTE_W bit must be set. 4) The above three conditions need to hold for each page of the environment's buffer.

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.

Answer: xv6 must check that the address a of each byte in the buffer is inside the process's memory, by checking that $0 \leq a <$ `p->sz`. xv6 must translate the address by adding `p->mem`.

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.

Answer: There are many possibilities. Some lead to a block or inode not being marked free, but also not being used. Some lead to user-visible incorrectness.

Suppose a process is writing to a file, and `bmap()` decides it must allocate a new block to hold the data (line 4019), and calls `balloc()`. If a crash were to occur after line 3718 in `balloc()`, the block would be marked as in-use in the on-disk bitmap, but no inode would refer to it. This would not cause any immediate damage if the file system were to be used after a reboot, but it would decrease the amount of usable space in the file system.

As a second example, suppose a process is creating a new directory with the `mkdir()` system call, which calls `create()`, and the system crashed after line 4836. The new directory would be visible in its parent directory (due to the `dirlink()` at line 4831) but the new directory would not have “.” or “..” entries. If the system rebooted and a user process looked at the new directory it would look odd.

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.

Answer: This code allows the kernel threads of multiple processes to acquire a `struct buf` for the same disk block at the same time via `bread()`. Then, for example, if `balloc()` were invoked concurrently from two threads, `bread()` would return copies of the same bitmap block to both `balloc()`s, which would return the same free block to both threads. This could result in two different files using the same disk block for their data. The real `bget()` only lets one thread at a time use any given disk block, which serializes modifications to each block. Many functions rely on this serialization to avoid races or to avoid overwriting unrelated modifications to the same block: `balloc()`, `bfree()`, `ialloc()`, `ilock()`, `iupdate()`, and perhaps others.

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.

Answer: JOS only sets the PTE_U bit in the PTEs of pages that the current environment is allowed to use, and in particular, does not set PTE_U for kernel memory mapped above 0xF0000000. The x86 paging hardware doesn't allow access to pages without PTE_U when CPL is 3.

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

Answer: Both user and kernel memory must be mapped in the page table in order to cross the user/kernel boundary in either direction. For example, in order for user-level code to make a system call with INT, the IDT, GDT, task state segment, kernel stack, and kernel trap handlers must be at addresses that are present in the page table.

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.

Answer: The kernel and user-level processes use different segments in xv6, and the x86 hardware switches CS and SS segments automatically when crossing between user and kernel (and the kernel trap code switches DS). The user and kernel segments have different bases (`p->mem` and 0, respectively), so a given address refers to different physical memory depending on when user or kernel code is executing.

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.

Answer: Interrupts go through alltraps. If alltraps did not save %ecx and %edx, then the interrupted code would see unexpected changes in those registers.

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.

Answer: One problem is that user-level instructions that load memory into registers would sometimes not actually do the load. Another problem is that any user-level memory reference instruction might have the side effect of modifying %eax. `gcc` generates code that assumes neither of these things ever happens.

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]:

Answer: The idea is to map the display memory at an unused part of the virtual address space, write zeroes, and then unmap. This code zeroes the display memory one page at a time, mapping each page at UTEMP:

```
pte_t *pte = pgdir_walk(pgdir, (void *) UTEMP, 1);
for(pa = 0xf0000000; pa < 0xf0000000 + 1024*1024; pa += PGSIZE){
  *pte = pa | PTE_P | PTE_W;
  tlb_invalidate(boot_pgdir, UTEMP);
  memset(UTEMP, 0, PGSIZE);
}
*pte = 0;
```

```
/*
 * Virtual memory map:                                Permissions
 *                                                    kernel/user
 *
 *    4 Gig -------->  +------------------------------+
 *                     |                              | RW/--
 *                     ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
 *                     :              .               :
 *                     :              .               :
 *                     :              .               :
 *                     |~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~| RW/--
 *                     |                              | RW/--
 *                     |   Remapped Physical Memory   | RW/--
 *                     |                              | RW/--
 *    KERNBASE ----->  +------------------------------+ 0xf0000000
 *                     |  Cur. Page Table (Kern. RW)  | RW/--  PTSIZE
 *    VPT,KSTACKTOP--> +------------------------------+ 0xefc00000      --+
 *                     |         Kernel Stack         | RW/--  KSTKSIZE   |
 *                     | - - - - - - - - - - - - - - -|                 PTSIZE
 *                     |      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)   |                 PTSIZE
 *    USTABDATA ---->  +------------------------------+ 0x00200000        |
 *                     |       Empty Memory (*)       |                   |
 *    0 ------------>  +------------------------------+                 --+
 */
```

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?

Answer: Yes. If process Y is running, process X cannot be running. The only way process X can not be running is if it were interrupted by a timer interrupt, causing `trap()` to call `yield()` at line 2592. When `kill()` marks process X `RUNNABLE`, X's `yield()` will return, `trap()` will return to user space, and X will execute in user space. Eventually another interrupt (like the timer interrupt) will occur and `trap()` will see that X's `p->killable` flag is set and call `exit()` at 2587.

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?

Answer: Yes. Process X is running in user space on a different CPU than Y, and will continue to execute until the next hardware interrupt (like the timer interrupt) on its CPU.

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()?

Answer: If a kernel thread acquires two locks and then releases one, `release()` would cause interrupts to be turned on. Then an interrupt could occur, and if the interrupt handling code tried to acquire the one lock that is still held, `acquire()` would panic.

VIII Fork

14. [4 points]: 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;
}
```

Answer: Nothing would go wrong. The `scheduler()` on a different CPU could run the new process just after `sys_fork()` marks the new process `RUNNABLE`, and the new process might immediately call `exit()`. It turns out this does no harm, because the process table slot won't be re-used (i.e. `np->pid` won't be overwritten) until the process calling `sys_fork()` calls `wait()`.

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