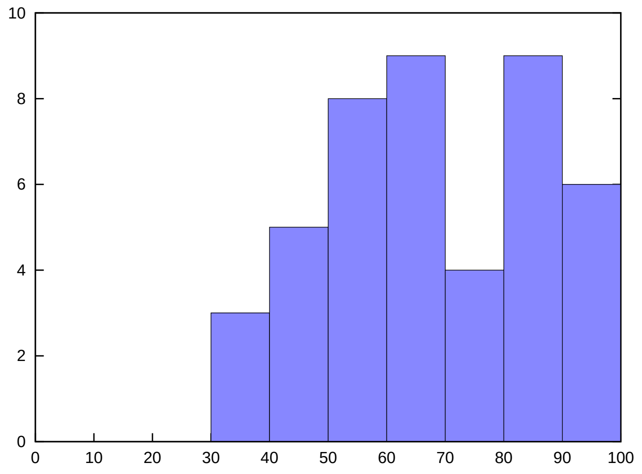
Department of Electrical Engineering and Computer Science

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

6.828 Fall 2010

Quiz I Solutions

Mean 67 Median 66 Std. dev. 18.3



I Concurrency

1. [8 points]: Alyssa runs xv6 on a machine with 8 processors and 8 processes. Each process calls `sbrk` (3351) continuously, growing and shrinking its address space. Alyssa measures the number of `sbrks` per second and notices that 8 processes achieve the same total throughput as 1 process, even though each process runs on a different processor. She profiles the xv6 kernel while running her processes and notices that most execution time is spent in `kalloc` (2426) and `kfree` (2405), though little is spent in `memset`. Why is the throughput of 8 processes the same as that of 1 process?

Answer: Both `kalloc` and `kfree` hold `kmem.lock` during their whole execution, which will serialize all of the processes so that only one is making progress at a time.

2. [8 points]: Does the JOS kernel have a mechanism equivalent to xv6's `swtch` (2308)? If yes, what? If not, explain why xv6 needs it but JOS does not.

Answer: No. xv6 must switch between the stacks of kernel threads, while JOS has only one kernel stack.

3. [8 points]: xv6 enables interrupts in the kernel during system calls and device interrupts, which adds some complexity since xv6 has to carefully disable and enable interrupts when locking. In contrast, JOS (as you will find in Lab 4 Part C) only enables interrupts in user space, and arranges for the hardware to automatically disable interrupts when entering the kernel. Would anything go wrong if xv6 also disabled interrupts in the kernel? Support your claim.

Answer: Yes. Suppose there is only one running or runnable process and it performs a disk read. The kernel will issue the IDE request and that kernel thread will go to sleep to wait for the IDE interrupt. Since that was the only runnable process, the kernel will enter the idle loop and never wake up because it will never receive the IDE interrupt.
(It is *not* true that blocking system calls simply stop working. A blocking system call will still go to sleep, which will schedule another runnable user process and return to user space, where interrupts will be delivered. The danger is if there are *no* runnable user processes, in which case you stay in the kernel's idle loop with interrupts disabled.)

II File systems

Alyssa add the statement:

```
cprintf("bwrite sector %d\n", b->sector);
```

to `bwrite` (4014), as in one of the homeworks. She then types the following two commands and observes their disk writes:

```
$ echo > x
bwrite sector 4      (inode for x)
bwrite sector 4
bwrite sector 29     (directory data for /)
$ echo ab > x
bwrite sector 28     (bitmap)
bwrite sector 508    (file data for x)
bwrite sector 4
bwrite sector 508
bwrite sector 4
bwrite sector 508
bwrite sector 4
```

Alyssa observes that her commands run slowly because each `bwrite` is synchronous (that is, it waits until the disk finishes writing). She recalls from lecture that this is necessary to help `fsck` be able to repair any inconsistencies after a crash or power failure, but naturally wonders if any of the writes could be made asynchronous. An asynchronous write updates the in-memory buffer for the block and marks it dirty, but does not send the buffer to the IDE driver. Instead, a dirty buffered block is written to the disk on a later synchronous write to that block, after a delay, or when buffer space is needed for some other block. Dirty buffered blocks may be written to disk in any order.

4. [10 points]: Assume xv6 does *no* repair of a file system after a reboot (it has no `fsck`). Alyssa starts by thinking about a single aspect of consistency: preventing the same block from being referenced by two different files; she will worry about other aspects later. Which writes during `echo ab > x` above must be synchronous to ensure this does not happen? Provide a brief explanation.

Answer: Only the write of sector 28, which marks the file data block in-use, must be synchronous. This ensures that the block allocation write cannot occur after the first write to the inode, which creates the reference to the block. If the inode write were to happen first, followed by a crash, then after the system rebooted and read the bitmap back in, it could allocate that block for another purpose, even though it's actually in use. The remaining writes simply update file data and the file length.

Alyssa goes on to wonder how to extend xv6 to support *named pipes*. A named pipe allows the following shell script:

```
$ mkfifo mypipe
$ cat file > mypipe &
$ wc < mypipe
$ rm mypipe
```

This shell code behaves much like `cat file | wc`, except that the pipe is given a regular file name that is stored in the file system, even though the contents of the pipe are not. Just like a regular pipe, the kernel passes all data from the writer to the reader internally, via in-memory buffers (without writing to the disk). Unlike a regular pipe, a named pipe appears in the file system, as a "special file." As a result, the processes communicating over the pipe don't have to have a parent-child relationship, since they can open the pipe by its file name (as in the example).

5. [12 points]: Describe what modifications you would make to xv6 to support named pipes so that you can run the above shell script. You don't have to provide code; just sketch what modifications you would make in which files. (Hint: exploit the existing pipes.)

Answer: mkfifo would create a file with the new inode type `T_PIPE`. The first open of such a file should create a `struct pipe` using `pipealloc`, record the resulting pair of `struct file`'s in the `struct inode`, and associate the `struct file` with the returned FD. From then on `read()`s and `write()`s would go to the `pipe.c` code. Later opens should simply fetch the appropriate `struct file` from the inode. Various other changes are also necessary to add an argument to `sys_mknod`, handle closing of named pipes, etc.

III Virtual Memory

Here's a simplified version of JOS's virtual memory map from `memlayout.h`.

```
   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
                    |       ...                    |
   0 ------------>  +------------------------------+
#define KERNBASE 0xF0000000
#define VPT (KERNBASE - PTSIZE)
#define KSTACKTOP VPT
#define ULIM (KSTACKTOP - PTSIZE)
#define UVPT (ULIM - PTSIZE)
#define UPAGES (UVPT - PTSIZE)
#define UENVS (UPAGES - PTSIZE)
#define UTOP UENVS
#define USTACKTOP (UTOP - 2*PGSIZE)
```

6. [8 points]: You have a machine with 4 GB of DRAM (and 32-bit addresses). What new value should you give KERNBASE to allow a single environment to allocate and address the largest number of distinct user memory pages possible? It's okay to be off by a few dozen megabytes. Explain why a KERNBASE higher or lower than your answer would reduce the maximum allocatable environment size.

Answer: About 0x80000000 (2 gigabytes). A higher KERNBASE would reduce the available physical memory; JOS can only use physical pages that it can access above KERNBASE. A lower KERNBASE would reduce user virtual address space available to the environment.

7. [12 points]: You have the following user-level code in JOS:

```
sys_page_alloc(0, 0x1000000, PTE_P|PTE_W|PTE_U);
*(int*)0x1000000 = 111;
sys_page_alloc(0, 0x2000000, PTE_P|PTE_W|PTE_U);
*(int*)0x2000000 = 222;

// your call(s) to sys_page_map() here

if(*(int*)0x1000000 == 222 &&
   *(int*)0x2000000 == 111)
  cprintf("OK\n");
```

Show how to make this code print `OK` using just one or more calls to

```
int sys_page_map(envid_t srcenvid, void *srcva,
                 envid_t dstenvid, void *dstva,
                 int perm);
```

Before the above code, nothing is mapped between 0x1000000 and the user stack.

Answer:

```
sys_page_map(0, 0x1000000, 0, UTEMP, PTE_P|PTE_W|PTE_U);
sys_page_map(0, 0x2000000, 0, 0x10000000, PTE_P|PTE_W|PTE_U);
sys_page_map(0, UTEMP,     0, 0x20000000, PTE_P|PTE_W|PTE_U);
// (the temporary address doesn't have to be UTEMP)
```

IV JOS

Ben Bitdiddle decides to experiment with a new and potentially more flexible system call convention. Rather than passing the syscall number and arguments in registers, he instead writes the syscall number and arguments to a “syscall page” at a new, fixed location `USYSCALL` before invoking `int 0x30`. He defines a `struct SysArgs` for storing the arguments, adds `struct Page *env_syspage` to `struct Env` to record the physical location of an environment’s syscall page, and adds the following code to allocate and map the syscall page for each environment:

```
static int
env_setup_vm(struct Env *e) {
  // ...
  page_alloc(&e->env_syspage);
  page_insert(e->env_pgdir, e->env_syspage,
              (void*)USYSCALL, PTE_U|PTE_W);
  return 0;
}
```

8. [10 points]: Ben begins updating his system call implementations to read their arguments from `USYSCALL`. His new convention is working great until he updates his inter-environment message passing syscalls. In his implementation, an environment sends a message by writing the message and its destination environment ID to its syscall page and invoking the kernel, which validates arguments, switches to the destination environment’s address space, copies the message from `(struct SysArgs*)USYSCALL`, then switches back to the calling environment’s address space. However, Ben finds that this writes the wrong data to the destination environment. Alyssa suggests that Ben should instead read the message from
`(struct SysArgs*)page2kva(curenv->env_syspage)`. Explain what was wrong with Ben’s approach and why Alyssa’s approach fixes Ben’s problem.

Answer: `USYSCALL` is a virtual address that will map to a different phyiscal page in each environment’s page table. Ben’s code is reading the message data from the *destination* environment’s address space, instead of the source environment’s, which contains the data he wants to copy. Alyssa’s approach works because it reads the message via the source environment’s physical `USYSCALL` page.

9. [8 points]: Both the original calling convention and Ben's new scheme use an interrupt to invoke the syscall. What prevents him from following the usual C calling convention from the user environment and simply calling the kernel function that implements the desired syscall?

Answer: Because the kernel text is mapped without the `PTE_U` bit set, the CPU will page fault when it tries to read an instruction from a function in the kernel text. (Alternatively, the kernel must execute at a higher privilege level than user code, and a simple function call does not switch privilege levels on the x86.)

10. [8 points]: Ben wants to take his syscall convention to a new level, so he adds *syscall batching*. He modifies his user code to store a *sequence* of system calls in the syscall page and modifies the `syscall` function in his kernel to iterate over this page and invoke each function in the sequence before returning to `trap_dispatch`. Louis Reasoner wants to take this for a spin, so he modifies `user/yield.c` to write its sequence of `sys_yield`, `sys_cputs`, `sys_yield`... syscalls to the syscall page before entering the kernel, as follows

```
for (i = 0; i < 5; i++) {
  batch_add(SYS_yield);
  batch_add(SYS_cputs, "Back in environment\n");
}
sys_execute_batch();
cprintf("All done.\n");
```

Much to his surprise, none of the "Back in environment" messages get printed, even though "All done" does. What happened?

Answer: JOS's `sys_yield` enters the scheduler. When the yielding environment gets scheduled again, JOS will resume execution from the user-space EIP in the environment's saved trap frame, which will be the EIP of the instruction following the `sys_execute_batch int 0x30` (roughly, the "All done." `cprintf`). Abstractly, the position in the syscall batch is transient state kept on the kernel stack by the loop in syscall, which is lost when switching environments because JOS only has one kernel stack.

V 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 (mostly) ahead of lecture draft chapters that provide a commentary on xv6. Did you read them? If so, did you find the chapters useful? What should we do to improve them?

Answer: They were fantastic, except that the line numbers didn't match the code (please don't wait until the quiz feedback to point out bugs like this to us!)

12. [2 points]: Are the labs too time consuming, too short, or are they about right?

Answer: About right.

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

Answer: The students.

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

Answer: Debugging little mistakes.

End of Quiz