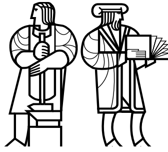


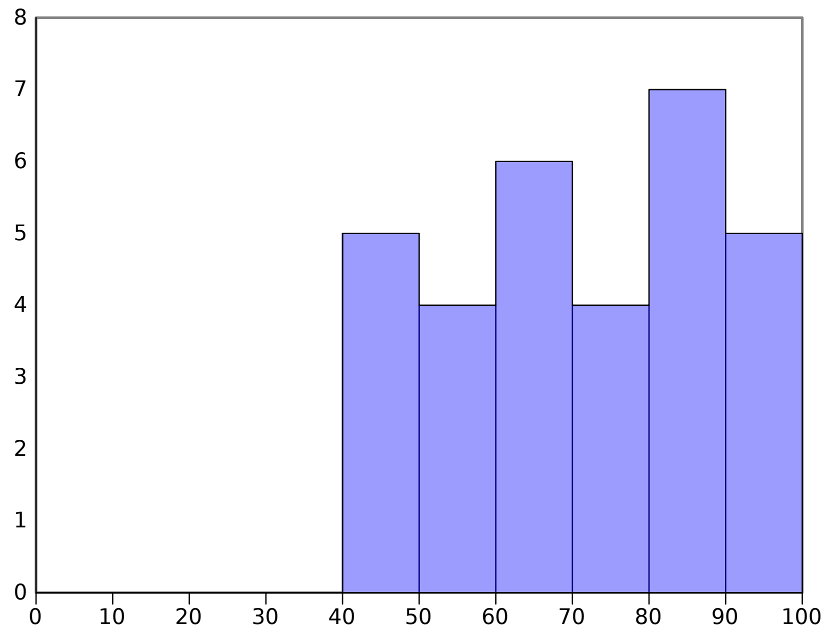
Department of Electrical Engineering and Computer Science

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

6.828 Fall 2009

Quiz I Solutions

Mean 70.6 Median 73 Std. dev. 17.9



I Concurrency

1. [8 points]: In xv6, the interrupt handler `ideintr` uses the lock `idelock` to ensure that it executes atomically. For example, if processor 0 runs `iderw` and processor 1 runs the interrupt handler `ideintr`, then the lock `idelock` ensures that their critical sections are appropriately ordered. Why is it important that `acquire(&idelock)` also turns off interrupts, in addition to marking the lock as acquired?

Answer: Suppose an IDE interrupt occurred on processor 0 while `iderw` held the `idelock` on processor 0. If `acquire` did not disable interrupts, this interrupt would be serviced immediately, resulting in a call to `ideintr`, which would try to acquire the `idelock`. This would panic because the lock is already held on processor 0. (If `acquire` did not check for this situation, it would simply deadlock, waiting for the lock to be released while blocking the very processor holding the lock.)

2. [8 points]: `bget` (and thus `bread`) returns a locked block to its caller. Why is it important that xv6 locks blocks? Consider `ialloc` (line 3802). What could go wrong from line 3811 through 3817 if the block returned by `bread` wasn't locked?

Answer: Without this, simultaneous calls to `ialloc` might allocate the same inode for two different files.

3. [8 points]: `bget` (line 3566) uses an integer `flags` stored in the returned block to indicate whether a thread is holding the lock on the block, instead of a spinlock. What could go wrong if we replaced the per-block `B_BUSY` flag used by `bget` with a per-block spinlock?

Answer: `sched` would panic. Specifically, when `bread` calls `iderw`, `iderw` puts the I/O in the IDE queue and puts the current process to sleep while waiting for an IDE completion interrupt. However, holding the block's spinlock disables interrupts, so it will never receive this interrupt. `sched` (called by `sleep`), will panic if any spinlocks are held, precisely to catch such mistakes.
Note that `bget` can check whether or not a per-block spinlock is held just as readily as it can check the `B_BUSY` flag, so it is *not* the case that `bget` will block while trying to allocate a fresh block.

4. [4 points]: What ensures that setting `flags` in `bget` (line 3566) is an atomic operation?

Answer: `bget` holds the buffer cache lock, `bcache.lock`, which protects modifications to `flags`.

II File systems

5. [8 points]: Ben wants to speed up the rate at which xv6 can write to the disk. He notices that `iderw` can have only one outstanding disk write per user process; xv6 puts the caller of `iderw` to sleep until the dirty bit is cleared by `ideintr`. Ben proposes to modify `iderw` to not call sleep, but instead return immediately. If a process issues multiple writes, all these writes will end up in the IDE queue, and the driver and disk controller can optimize performing a batch of writes. Ben makes all the required changes to xv6; he observes that write throughput improves for large files and that the system works correctly when there are no failures. Unfortunately, when Ben runs `fsck` after a system failure, he finds he cannot repair the disk into a consistent state anymore. What has gone wrong?

Answer: Ben's change allows metadata writes to be reordered. The consistency of the file system's on-disk structures depends on specific ordering of metadata writes. For example, suppose you deleted a file and then created a new file. If the new file's inode is written to disk before the removal is written to disk, and the system failed between these two writes, then the two files could potentially share the same data blocks.

6. [6 points]: Ben obsesses about performance and proposes a design to speed up directory listings. He stores the inode for each file in the `struct dirent`, a scheme called *embedded inodes*. When a user types "`ls -l`" to get a detailed directory listing, xv6 can avoid reading the inode for each file in the directory, because it is stored with the directory. Ben makes the change, and observes that xv6 is faster for running "`ls -l`". He also notices that hard links don't work correctly any more. Give a sequence of shell commands that will work incorrectly with Ben's embedded inodes and briefly describe what will go wrong.

Answer: Here's one of many possible solutions

```
echo a > a
ln a b
echo bbb > b
ls -l
```

will yield a directory listing where `a` has size 2 and `b` has size 4 (including the newlines produced by `echo`), even though they should be the same file and thus the same size.

7. [6 points]: Suggest a solution to the link problem that achieves the performance benefits of embedded inodes without sacrificing correctness.

Answer: If a file has a link count greater than 1, don't embed the inode. Alternative solutions include tracking the entire set of directories where a given file is linked and updating all copies of the inode when any one of them changes, but this would be very slow and poses serious challenges for file system recovery. Simply using symbol links instead of hard links is *not* a solution, since the semantics of hard links and symbolic links differ significantly.

III Stacks

8. [8 points]: Suppose you wanted to change the system call interface in JOS so that, instead of returning the system call result in EAX, the kernel pushed the result on to the user space stack. Fill in the code below to implement this. For the purposes of this question, you can assume that the user stack pointer points to valid memory.

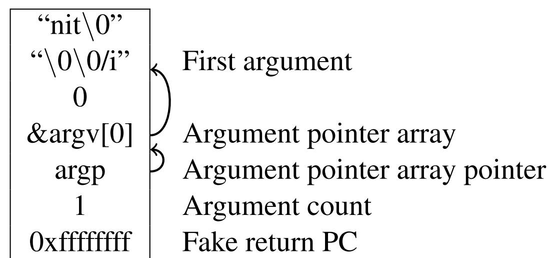
```
static void
trap_dispatch(struct Trapframe *tf)
{
        // ...
        if (tf->tf_trapno == T_SYSCALL) {
                r = syscall(tf->tf_regs.reg_eax, ...);
                // QUIZ: Your code here
```

Answer:
```
                tf->tf_esp -= 4;
                *(int*)tf->tf_esp = r;
```

```
        }
        // ...
}
```

9. [8 points]: xv6's `initcode` invokes the `exec` system call to replace itself with `/init`. `exec` allocates a stack for `/init` and carefully initializes it, starting from line 5080. Draw the stack it builds, giving the values that are on the stack and a brief explanation of each value. For pointers, simply draw an arrow or describe where they point to.

Answer:



10. [2 points]: While debugging xv6, Alyssa notices the value `0x7b70` in the ESP register. Which kernel thread is running on the processor?

Answer: The scheduler thread (on processor 0, to be exact).

IV Virtual Memory

Ben Bitdiddle wants to add a debugger for user programs to his implementation of JOS. He decides to write this debugger as a regular user program, and realizes that he's going to need to add some system calls to support debugging. He starts with a system call that lets the debugger copy a page of memory between its own environment and any other environment.

Ben first implements his new syscall, `sys_copy_page`, by switching to the source environment's address space and copying the data into a scratch buffer, then switching to the destination environment's address space and copying the data back out of the scratch buffer. Ben's code is as follows:

```
static char scratch[PGSIZE];

static int
sys_copy_page(envid_t srcenvid, void *srcva,
              envid_t dstenvid, void *dstva)
{
        struct Env *srcenv, *dstenv;
        int r;

        if ((r = envid2env(srcenvid, &srcenv, 0)) < 0)
                return r;
        if ((r = envid2env(dstenvid, &dstenv, 0)) < 0)
                return r;
        if (srcenv != curenv && dstenv != curenv)
                return -E_INVAL;

        lcr3(srcenv->env_cr3);
        memmove(scratch, srcva, PGSIZE);
        lcr3(dstenv->env_cr3);
        memmove(dstva, scratch, PGSIZE);

        lcr3(curenv->env_cr3);
        return 0;
}
```

11. [8 points]: Louis Reasoner is troubled by Ben's approach. He points out that Ben is copying data in to the scratch buffer while running in one environment's address space, then expecting to read that data back *after* switching to a different address space. Ben assures Louis that this is alright. Why is it okay for Ben's implementation to switch between address spaces like this?

Answer: `scratch` is stored in the kernel BSS, and thus above KERNBASE. All mappings above ULIM are identical in every environment's address space, so switching address spaces does not affect the mapping of `scratch`.

12. [6 points]: Having addressed Louis' concerns, Ben starts boasting about his understanding of address spaces and protection. Alyssa decides to put his kernel to the test and discovers that, not only can she crash Ben's kernel by passing the virtual address of an unmapped page to `sys_copy_page`, but she can write a user program that writes to arbitrary kernel memory. How can Ben fix these two problems?

Answer: Ben should check that `srcva` is mapped with permissions `PTE_P|PTE_U` in `srcenv->env_pgdir` and that `dstva` is mapped with permissions `PTE_P|PTE_U|PTE_W` in `dstenv->env_pgdir`. A convenient way to do this is with `user_mem_assert`:

```
user_mem_assert(srcenv, srcva, PGSIZE, 0);
user_mem_assert(dstenv, dstva, PGSIZE, PTE_W);
```

13. [6 points]: Ben decides to change his strategy to copy data directly from the source page to the destination page without switching address spaces. His new implementation of `sys_copy_page` starts by finding the `struct Page*` of the source page and the physical address of the destination page. Why does this approach let Ben avoid switching address spaces in `sys_copy_page`?

Answer: All physical memory is directly mapped starting at KERNBASE in every environment's address space. Thus, the kernel can convert a physical address to a virtual address that is valid regardless of the current address space.

14. [8 points]: Ben uses a helper function to copy data between the source and destination pages that he looked up in his new `sys_copy_page`. Implement this function below.

```
void
copy_physical_page(physaddr_t dst, struct Page* src)
{
```

Answer: `memmove(KADDR(dst), page2kva(src), PGSIZE);`

```
}
```

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.)

15. [2 points]: This year we posted 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?

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