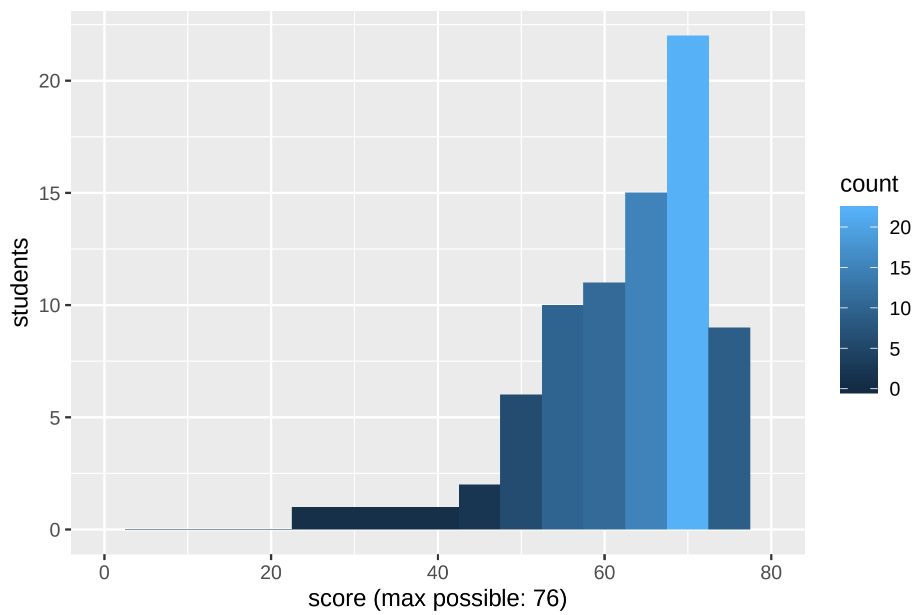
Department of Electrical Engineering and Computer Science

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

6.S081/6.828 Fall 2019

Quiz II Solutions

Mean 62.1 Median 65.0 Standard deviation 10.4



I Xv6 file system

Alyssa adds the statement:

```
printf("bwrite %d\n", b->blockno);
```

to xv6's `bwrite` in `bio.c`. She then makes a fresh `fs.img`, boots xv6, and runs the following command:

```
$ cat README > z
bwrite 3
bwrite 4
bwrite 5
bwrite 2
bwrite 33
bwrite 46
bwrite 32
bwrite 2
bwrite 3
bwrite 4
bwrite 5
bwrite 2
bwrite 45
bwrite 801
bwrite 33
bwrite 2
bwrite 3
bwrite 4
bwrite 2
bwrite 801
bwrite 33
bwrite 2
bwrite 3
bwrite 4
bwrite 5
bwrite 2
bwrite 45
bwrite 802
bwrite 33
bwrite 2
bwrite 3
bwrite 4
bwrite 2
bwrite 802
bwrite 33
bwrite 2  // XXX
$
```

Alyssa is surprised by the large number of blocks written. She looks at the source code of `cat.c` and observes that `cat` writes 512 bytes at the time. `ls` reports that `README` is 1982 bytes large. So, `cat README > z` results in 4 write system calls. The block size of the xv6 file system is 1024 bytes, so the content of `z` fits in 2 file system blocks.

1. [5 points]: Explain what block 801 contains.

Answer: The first 1024 bytes of `z` (which is equal to the first 1024 bytes of `README`)

2. [5 points]: Explain briefly what block 33 contains.

Answer: It is a block of inodes, including the inode for file z.

3. [5 points]: Explain briefly the purpose of the write to block 2 at the line marked `XXX`.

Answer: Block 2 is the log header block. The write at XXX is setting n in the header block to zero to indicate that the log no longer holds a valid transaction, since the transaction has been installed.

II Lab Lock

Ben is working on parallelizing xv6's memory allocator for 6.828's lock lab. He modifies the kernel's page allocator to use per-CPU free lists, using the `cpuid()` function to determine which CPU the code is running on. Following the hint in the lab text, Ben sees that `cpuid()`'s documentation (in `proc.c`) says "Must be called with interrupts disabled, to prevent a race with the process being moved to a different CPU." He writes his `kalloc()` implementation as follows:

```
1  void *
2  kalloc(void)
3  {
4    struct run *r;
5
6    push_off();
7    int me = cpuid();
8    pop_off();
9    acquire(&kmem[me].lock);
10   r = kmem[me].freelist;
11   if(r)
12     kmem[me].freelist = r->next;
13   release(&kmem[me].lock);
14
15   // if r == NULL, search other CPUs' free lists
16   // ...
17
18   // clear page and return it
19   // ...
20 }
```

4. [5 points]: Ben expects line 10, `r = kmem[me].freelist;`, to access the free list of the CPU that the code is currently executing on. It turns out that this is **not** always the case. Describe a concrete sequence of events that violate Ben's expectation.

Answer: Line 7 gets cpuid, then after interrupts are re-enabled (line 8), there's a clock interrupt and the process is later scheduled to a different CPU. It then uses the cpuid it got earlier, which no longer corresponds to the current CPU.

5. [5 points]: How could Ben change his code so that his expectation holds, i.e. `r = kmem[me].freelist;` is guaranteed to access the free list of the CPU that the code is executing on?

Answer: Move `pop_off()` to after the `release` on line 13.

6. [5 points]: Alyssa points out that Ben can remove the calls to `push_off()` (line 6) and `pop_off()` (line 8), even though that violates the `cpuid()` function's specification. Ben modifies xv6 and it passes all the usertests, despite the missing `push_off()` and `pop_off()`. Explain why deleting these two lines doesn't break xv6.

Answer: It's okay to use the "wrong" cpuid because using the "right" free list is just a performance optimization; the locking guarantees correct behavior when accessing any of the free lists.

III Lab Syscall

The two parts of the 6.828 user-level threads and alarm lab both involve saving and restoring contexts. For user-level threading, this happens in `uthread_switch`, and for the alarm system call, saving the context happens in `usertrap()` and restoring the context happens in `sys_sigreturn()`. While all registers are saved/restored for the alarm system call, this is not necessary for `uthread_switch`, which only needs to save `sp`, `s0` through `s11`, and `ra`.

7. [5 points]: Explain why `uthread_switch` can get away with not saving and restoring certain general-purpose registers.

Answer: The main difference is that the timer interrupt for alarm is preemptive, while switching uthreads is done cooperatively. `uthread_switch` is called via a function call, so the compiler will emit the code to save caller-saved registers, and so `uthread_switch` only needs to save callee-saved registers.

IV xv6 i-node counts

Alyssa is implementing symbolic links in xv6, as part of the 6.828 file system lab. She observes that each in-memory inode (`struct inode` in `file.h`) contains two similar fields: `nlink` and `ref`.

8. [5 points]: Suppose a bug accidentally changed a file's `nlink` field from 2 to 1. What bad thing(s) would happen as a result?

Answer: The fact that the file's correct `nlink` is two means that it has two names. If `nlink` is incorrectly set to one, then if one of the two names is removed, xv6 will delete the file's content and mark the i-node as free. At that point the file's other name refers to a freed i-node, which will cause a panic if the name is ever used. If a new unrelated file is created and is allocated the i-node, then that other name will incorrectly refer to the unrelated new file.

9. [5 points]: Suppose a bug accidentally changed a file's `ref` field from 2 to 1. What bad thing(s) would happen as a result?

Answer: The correct value of two means that there are two open file descriptors (or current directories) that refer to the in-memory i-node. If the value is incorrectly set to one, and then one of the file descriptors is closed, the `struct inode` will be marked free, and possibly re-used for a different file. Then that other file descriptor will refer to nothing, or to the wrong file.

V EXT3

Recall the Linux EXT3 journaling file system from *Journaling the Linux ext2fs Filesystem* and Lecture 14. The paper's "ext2fs" is the same as EXT3.

Suppose you run the following program on Linux with EXT3:

```
int
main()
{
  int fd;

  fd = open("a", O_CREAT|O_WRONLY, 0666); // create a
  if(fd < 0) exit(1);
  close(fd);

  fd = open("b", O_CREAT|O_WRONLY, 0666); // create b
  if(fd < 0) exit(1);
  close(fd);

  printf("done\n");
}
```

The two `open()` system calls create files. Before you run the program, the two files did not exist.

The program prints `done`, so you know the file creations succeeded. Moments after the program finishes, there's a power failure and your computer (which has no battery) stops executing. After a while the power comes back on, and your computer reboots and runs EXT3's recovery code. You look for files a and b.

10. [4 points]: Which of the following situations could exist at this point?

(Circle True or False for each choice.)

- **True / False** Neither a nor b exists.
- **True / False** File a exists, but not b.
- **True / False** File b exists, but not a.
- **True / False** Both a and b exist.

Answer: True, True, False, True. The answer depends on when the power failure occurs relative to when EXT3 writes data from its in-memory block cache to its log on disk. One possibility is that EXT3 has written nothing at all; in that case, neither a nor b will exist. Another possibility is that EXT3 decides to commit its log to disk after the program creates file a (but before it creates b), and all the log writes for the commit finish; and then the power fails; the result will be that a exists but not b. In this scenario, the program's system call to create b finishes but only modifies cached blocks in RAM. Another possibility is that the power failure occurs after EXT3 commits both a and b to its log on disk; then both will be visible. It cannot be the case that b exists but not a, because when EXT3 decides to commit, it commits all completed system calls; if the creation of file b has completed, so must the creation of file a.

VI RCU

Consider *RCU Usage In the Linux Kernel: One Decade Later*, by McKenney *et al*.

Suppose Figure 6's `setsockopt()` were modified to copy the new options into the socket structure, rather than changing a pointer, like this:

```
if (opt == IP_OPTIONS) {
  memmove(sock->opts, arg, ...the correct size...);
  return;
}
```

This modification would require that `sock->opts` be a buffer of the appropriate size. `udp_sendmsg()` remains the same.

11. [5 points]: Explain why this modification would break RCU. What kinds of problems would it cause to `udp_sendmsg()`?

Answer: `udp_sendmsg` might observe a partially written `sock->opts`, because `memmove` is not atomic.

Ben Bitdiddle is thinking about adding RCU to his xv6 kernel, which he runs on real RISC-V hardware. His RISC-V hardware has a timer that ticks every 10 milliseconds, and Ben sees that xv6's `kerneltrap()` causes a context switch if a timer interrupt arrives while a kernel thread is running. He reasons that since the point of `synchronize_rcu()` is to wait for a context switch on each CPU, he could change `synchronize_rcu()` to simply wait 10 milliseconds rather than schedule itself on each CPU in turn (as in Figure 2).

12. [5 points]: Explain why Ben's idea of waiting 10 milliseconds would break RCU.

Answer: An RCU read critical section might last more than 10 milliseconds. As a result it might still be using an RCU-protected pointer after `synchronize_rcu()` returns.

Alyssa is excited about Biscuit as described in *The benefits and costs of writing a POSIX kernel in a high-level language* by Cutler et al. She notices that Biscuit uses RCU for its directory cache. In studying the Biscuit code, she notices that there are no calls to `synchronize_rcu()` (or to any similar function).

13. [5 points]: Explain why Biscuit has no need for `synchronize_rcu()`.

Answer: `synchronize_rcu()` is used to delay freeing until nobody else has a pointer to the object. Go's garbage collector does this automatically.

VII Networking lecture/reading

Consider *Eliminating Receive Livelock in an Interrupt-driven Kernel*, by Mogul *et al*, as well as Lecture 17.

Ben notices that a networking stack is failing to make much progress sending packets when it is receiving packets at a high rate. To solve this problem, he configures the network interface card (NIC) to **not** generate receive interrupts, and instead polls the receive descriptor ring during each timer tick (every millisecond). More progress is now made in sending packets, but a new problem has appeared where some incoming packets are dropped even at load below saturation.

14. [5 points]: Explain why packets are dropped because of this change.

Answer: Even if the CPU isn't saturated (fully busy), the packet rate could be high enough to fill the receive descriptor ring entirely before the next polling interval. When the ring becomes full, the NIC drops the packets.

VIII Virtualization

Consider *A Comparison of Software and Hardware Techniques for x86 Virtualization*, by Adams *et al*.

The designers of a new RISC-V processor want to more easily debug the state of the running CPU, so they allow reads to the `sstatus` (supervisor status) register to be performed from user mode without trapping. Normally making `sstatus` readable doesn't create any problems, but suppose a trap-and-emulate hypervisor (in supervisor mode) is running xv6 as a guest (in user mode). After running for a short time the guest xv6 kernel panics and prints "kerneltrap: not from supervisor mode".

Note: `sret` does not modify `SSTATUS.SPP`.

15. [5 points]: Explain why allowing `sstatus` to be read in user mode without trapping results in this specific panic in the xv6 guest kernel.

Answer: `kerneltrap()` reads sstatus and checks SPP as a sanity check. If the guest is executing in the kernel (in virtual supervisor mode) and an interrupt occurs, SPP should be 1. A VMM would ordinarily trap and emulate a read to sstatus and make the SPP be 1 (the virtual CPU state). On this incorrectly virtualized machine, the SPP bit would be 0 if the previous physical trap originated from user mode. This is an example of violating the fidelity property from the Popek and Goldberg virtualization requirements, as discussed in lecture.

IX 6.828

16. [1 points]: Please indicate which of the labs you found to be the most helpful, and which the least.

- Alarm / uthreads **Answer: 18 helpful / 13 unhelpful**
- Locking **Answer: 11 helpful / 26 unhelpful**
- File system **Answer: 20 helpful / 7 unhelpful**
- mmap **Answer: 21 helpful / 10 unhelpful**
- Networking (if you did this lab) **Answer: 8 helpful / 5 unhelpful**

17. [1 points]: Which paper is the best candidate for deletion in future years?

Answer:

- ext2fs: 9
- VM primitives for user programs: 16
- Exokernel: 9
- Receive livelock: 9
- Biscuit: 10
- Scalable locks: 1
- RCU: 1
- Virtualization: 20
- None: 1

End of Quiz II — Happy holidays!