Department of Electrical Engineering and Computer Science

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

6.1810 Fall 2025

Quiz II Solutions

Mean 81.65 Median 82 Standard deviation 14.34 Maximum 113 (out of 113)

I Threads

Ben Bitdiddle discovers that the C compiler he uses to compile the xv6 kernel generates code that never uses the `s11` register. He decides to optimize `swtch` by removing the lines that save and restore `s11` from `swtch.S`. However, Ben compiles the user-space applications running on top of his xv6 kernel with a different compiler that does make use of `s11`.

1. [10 points]: Would this be a safe change to make, assuming that the kernel itself is always compiled with Ben's compiler that never uses `s11`?

Answer: Yes, this will not affect any kernel code execution, since it does not depend on `s11`, and user-space code that might use `s11` will also not be affected because the user-space `s11` register is saved and restored by `uservec` and `userret`.

II Coordination

Ben is extending the xv6 kernel to support a new pipe-like abstraction, the *bipe*. A bipe has just one reading and one writing process, which are known in advance when the bipe is created. Only a single byte is ever written to a bipe.

Ben would like a bipe's reading process to wait for the one byte if the writer hasn't written it yet, and he'd like the reader to wait without spinning. He realizes that xv6's sleep and wakeup could be used here, but he believes he can use simpler code that's specialized to his use case.

On the next page is part of Ben's kernel bipe implementation (what's missing is initialization, freeing when no longer in use, and glue code to allow user system calls to use the functions). In Ben's application there's no need for a bipe reader to worry about being killed while waiting.

```
struct bipe {
  struct spinlock lock;
  char data;        // just one character
  int ready;        // has data been written?
  struct proc *dst; // the one reader
};

int biperead(struct bipe *b) {
  struct proc *p = myproc();
  while(1){
    acquire(&b->lock);
    if(b->ready){
      char c = b->data;
      release(&b->lock);
      return c;
    }
    release(&b->lock);

    // AAA

    acquire(&p->lock);
    p->state = SLEEPING;
    sched();
    release(&p->lock);
  }
}

void bipewrite(struct bipe *b, char c) {
  acquire(&b->lock);
  b->data = c;
  b->ready = 1;
  release(&b->lock);

  // BBB

  acquire(&b->dst->lock);
  if(b->dst->state == SLEEPING)
    b->dst->state = RUNNABLE;
  release(&b->dst->lock);
}
```

2. [10 points]: Ben's bipe implementation has a serious correctness problem. What is it?
(Circle the one best choice; we subtract points for incorrect answers.)

A. xv6 forbids kernel code from holding locks when calling `sched()`, but `biperead()` holds `p->lock`.

B. if `biperead()` executes entirely while `bipewrite()` is at line `BBB`, the reader may never see the data.

C. `biperead()` should hold `b->lock` for the entire time, acquiring it at the start of the function, and only releasing it just before returning.

D. if `bipewrite()` executes entirely while `biperead()` is at line `AAA`, the reader may never see the data.

E. `bipewrite()` acquires `b->dst->lock`, but it should instead acquire its own lock (i.e. `myproc()->lock`).

Answer: D is correct: in this situation, the reader would put itself to sleep after the writer had decided that it didn't need to wake up the reader. **A** is wrong because xv6 allows (indeed requires) a process to hold `p->lock` when it puts itself to sleep. **B** is wrong because the reader in that situation will see the data, and won't put itself to sleep. **C** is wrong because then the writer may never be able to acquire `b->lock` and set `b->data` and `b->ready`. **E** is wrong.

III Networking lecture/reading

Consider *Eliminating Receive Livelock in an Interrupt-driven Kernel*, by Mogul *et al.*, and Lecture 15.

Ben Bitdiddle runs xv6 just like in the net lab (with 3 CPUs), but configures the hardware to deliver the network card interrupt only to CPU 0. Ben's xv6 network stack does not use transmit interrupts.

3. [10 points]: Could Ben's system experience receive livelock? Briefly explain your answer.

Answer: No, because other CPUs can make progress, even if CPU 0 spends all of its time processing network receive interrupts.

IV Shenango

Which of the following are true statements about Shenango as described in "Shenango: achieving high CPU efficiency for latency-sensitive datacenter workloads"?

(Circle True or False for each choice; we subtract points for incorrect answers.)

4. [12 points]:

A. True / False To get high performance, Shenango allows the IOKernel to read and write descriptors in the NIC queues from user space.

B. True / False Shenango may run an application on cores guaranteed to another application.

C. True / False A 99.9% tail latency of 37μsec means 999 out of 1000 requests take more than 37μsec.

D. True / False Even when memcached is under high load from clients Shenango manages to run a batch processing application occasionally.

Answer: A and D are correct. B is wrong because Shenango allows bursting onto extra burstable cores, not onto guaranteed ones. C is wrong because tail latency is the opposite.

V File System Layout in xv6

Alyssa is using xv6 with triply-indirect blocks, and her file system has 10 million 1024-byte blocks (`FSSIZE` in `param.h` is 10000000). In one directory, Alyssa creates one file whose size is nearly 10 million blocks (so it almost fills the disk, but not quite). In another directory, that starts out empty, Alyssa creates a new file called `x` with this shell command:

```
echo hello > x
```

No other file-system-related actions occur.

5. [10 points]: Approximately how many blocks will Alice's `echo` shell command cause the kernel's file-system code to **read** from the disk?

(Circle the one best choice; we subtract points for incorrect answers.)

A. 2

B. 10

C. 600

D. 1300

E. 2600

Answer: The correct answer is **D**, 1300. The command requires the allocation of a block to hold "hello". Since most blocks were already allocated when Alyssa created the initial huge file, and no block has been freed since then, `balloc()` will need to scan nearly all of the bitmap blocks before it finds a free block. 10,000,000 bits require 1220 1024-byte blocks, so `balloc()` will need to read roughly that many blocks. Plus a few more to read the directory and allocate the i-node.

VI EXT3

Recall the Linux EXT3 journaling file system from *Journaling the Linux ext2fs Filesystem* and Lecture 19. The paper's "ext2fs" is the same as EXT3, and we are referring to the file system running with ordered data (as opposed to journal data) mode.

6. **[10 points]:** Which of the following are true statements about EXT3.
(Circle True or False for each choice; we subtract points for incorrect answers.)

   A. **True / False** EXT3 can combine changes from multiple system calls in a single log/journal commit.

   B. **True / False** When writing 1MB of data to a file, EXT3 will write roughly 2MB of data to disk (writing 1MB to the log/journal, and then installing that 1MB to the file's data blocks).

   C. **True / False** After a crash, EXT3 might be missing the last few file changes.

   D. **True / False** In EXT3, a system call can be modifying a directory `/d` while EXT3 is in the process of writing previous changes to directory `/d` to its on-disk journal.

   E. **True / False** EXT3 can evict cached disk blocks that have been committed to the journal but not yet installed.

Answer: **A** is true.

B is not true: EXT3 does not journal data writes.

C is true.

D is true: EXT3 will use copy-on-write to keep track of changes to `/d`'s blocks if they are modified before the previous dirty blocks are written to the journal.

E is not true. The blocks will eventually be needed in order to install them. In principle perhaps EXT3 could read them from the log, but it doesn't do that, since it would be slow. Instead EXT3 takes the more efficient approach of always leaving dirty blocks in the cache until it has installed them.

Suppose you run the following application on EXT3, where, before this program ran, the files `/a` and `/b` never existed, and `ccc` never appeared anywhere on disk:

```
int main() {
  int fd;

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

  for (int i = 0; i < 3; i++)
    write(fd, "a", 1);
  close(fd);

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

  for (int i = 0; i < 3; i++)
    write(fd, "b", 1);
  close(fd);

  fd = open("/a", O_RDWR, 0666);
  if (fd < 0)
    exit(1);

  for (int i = 0; i < 3; i++)
    write(fd, "c", 1);
  close(fd);

  exit(0);
}
```

Assume the program exits with exit status 0, and the system crashes after that.

7. [12 points]: What could you see in the file system after the system boots up and recovers?
(Circle True or False for each choice; we subtract points for incorrect answers.)

A. True / False You could see that `/a` and `/b` both exist and are both empty.

Answer: This is not possible: EXT3 commits meta-data updates to the journal in the order that the system calls executed, so if /b exists, the operation that extended /a must have also committed.

B. True / False You could see that neither `/a` nor `/b` exist, but `ccc` appears somewhere on disk.

Answer: This is possible: EXT3 writes file content blocks to disk before committing the associated meta-data to the on-disk log, so `ccc` could be on the disk before any of the log has been committed.

C. True / False You could see that `/a` contains `aaa` and `/b` is empty.

Answer: This is possible: the system could have committed the journal through the operation creating `/b`, but before the write to `/b`, and then crashed before committing the rest of the journal to disk.

D. True / False You could see that `/a` does not exist, and `/b` contains `bbb`.

Answer: This is not possible: if the transaction creating `/b` committed, it must have also committed the creation of `/a`.

VII Multicore scalability and RCU

Ben implements his read-write lock for the lock lab as follows:

```
struct rwspinlock {
 int32_t n;
};

void rwlock_r_acquire(struct rwspinlock *rwlk) {
 while (1) {
  int32_t v = __atomic_load_n(&rwlk->n, __ATOMIC_ACQUIRE);
  if (v < 0) { // does a writer have the lock?
   continue;
  }
  // __atomic_compare_exchange_n returns true if the value at &rwlk->n
  // matched v and the value was successfully updated to v+1.
  // It returns false otherwise.  You can ignore the last 3 arguments.
  if (__atomic_compare_exchange_n(&rwlk->n, &v, v+1,
                                  0, __ATOMIC_ACQUIRE, __ATOMIC_RELAXED)) {
     break;
  }
 }
}

void rwlock_r_release(struct rwspinlock *rwlk) {
  // atomically subtract 1 from rwlk->n
 __atomic_sub_fetch(&rwlk->n, 1, __ATOMIC_RELEASE);
}

void rwlock_w_acquire(struct rwspinlock *rwlk) {
 while (1) {
  // if rwlk->n is 0, set it to -1, and return true; otherwise
  // return false
  int32_t v = 0;
  if (__atomic_compare_exchange_n(&rwlk->n, &v, -1,
                                  0, __ATOMIC_ACQUIRE, __ATOMIC_RELAXED)) {
     break;
  }
 }
}

void rwlock_w_release(struct rwspinlock *rwlk) {
 __atomic_store_n(&rwlk->n, 0, __ATOMIC_RELEASE);
}
```

8. [8 points]: Which of the following statements are true about Ben's implementation?
(Circle True or False for each choice; we subtract points for incorrect answers.)

- **A. True / False** The lock can never be held by a reader and a writer at the same time
- **B. True / False** The lock can be held by many readers
- **C. True / False** A writer may never be able to acquire the lock
- **D. True / False** The lock can be acquired in read mode $2^{32} - 1$ times without releasing the lock

Answer: A, B, and C. C is correct because even if every reader holds the lock for a finite time and then releases, it could be that a collection of reading threads overlap their critical sections and cause n to be continuously greater than zero. D is wrong, because a signed-integer can count up to $2^{31} - 1$ readers; the next one overflows the signed integer. (Some real-world read-write lock implementations must handle overflow.)

Consider a multithreaded program in which n threads acquire a read-write lock in read mode, compute for some time t, and then release the lock. None of the threads acquire the lock in write mode. The program is running on a 16-CPU x86 processor as used in the RCU paper "RCU Usage in the Linux kernel: one decade later".

9. [8 points]: Which of the following are true statements?
(Circle True or False for each choice; we subtract points for incorrect answers.)

- **A. True / False** If n is large and t is 1s, the lock will be a performance bottleneck.
- **B. True / False** If n is large and t is 0s, the lock will be a performance bottleneck.

Answer: B is correct: the atomic operations in `rwlock_r_acquire` become a bottleneck, because all cores will continuously invoke them and the hardware will execute them serially. A is wrong because if the critical section is one second, there will be very little contention on the read lock.

VIII Containers and VMs

Consider the reading "Blending Containers and Virtual Machines: a study of Firecracker and gVisor (2020)". Suppose an adversary discovers that calling `close(-1)` causes the Linux kernel to crash, and writes a program to invoke `close(-1)`. Assume there are no other bugs.

10. [9 points]: Which of the following are true statements?
(Circle True or False for each choice; we subtract points for incorrect answers.)

A. True / False If the program is running on gVisor, the program causes a crash in the host kernel shown in Figure 2, if the host kernel is the buggy Linux kernel.

B. True / False If the program is running on Firecracker, the program causes a crash in the Linux kernel shown at the bottom of Figure 3 (assuming it is a buggy Linux kernel).

C. True / False If the program is running in a LXC container on top of a Linux kernel, the program causes a crash in the Linux kernel (assuming `close` is on the allowed syscall list).

Answer: C is correct, because an application inside an LXC container directly invokes the system calls of the buggy Linux kernel. B is false, because the application cannot directly invoke a system call of the host kernel: it can only invoke system calls of the guest kernel. A is false because the Sentry implements the `close` system call and doesn't relay it to the host kernel.

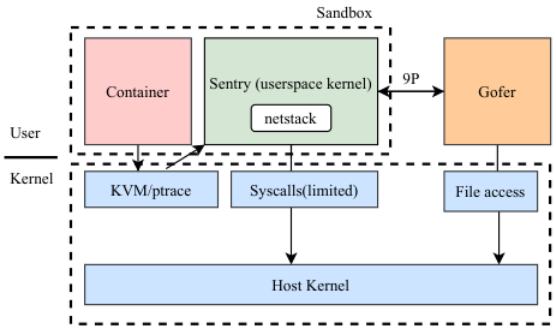

Figure 2. gVisor architecture

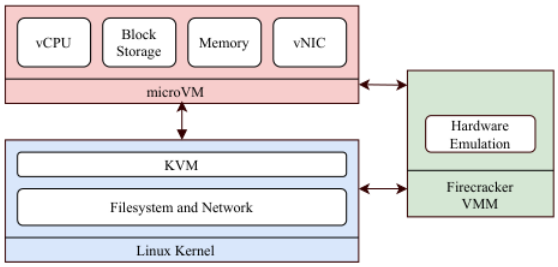

Figure 3. Firecracker architecture

IX Meltdown

The paper *Meltdown: Reading Kernel Memory from User Space*, by Lipp *et al.*, mentions that KAISER is an effective defense against Meltdown.

11. [10 points]: Why does KAISER prevent Meltdown from working?
(Circle the one best choice; we subtract points for incorrect answers.)

A. KAISER marks each page of physical RAM as inaccessible to user code.

B. KAISER clears the PTE_U bits of PTEs for kernel virtual addresses in user page tables.

C. KAISER causes kernel virtual addresses to all be mapped to physical page zero, so that Meltdown loads the wrong data.

D. KAISER causes user memory not be be mapped when the kernel is executing, so that kernel activity cannot affect user space.

E. KAISER enables kernel address space layout randomization (KASLR), which makes Meltdown impossible.

F. None of the above.

Answer: **F** is the correct answer. What KAISER does is set up user page tables so that they have no PTEs for kernel addresses. Pre-KAISER kernels mapped the kernel into user page tables but with PTE_U clear; the mere presence of the mappings, even without permissions, is what Meltdown exploits. **A** is wrong because even if there were a mechanism to protect physical pages, that would prevent the program from accessing its own memory. **B** is wrong because the kernel clears those PTE_U bits even without KAISER: and much of the point of Meltdown is that it can extract kernel data despite the PTE_U bits being clear. **C** is wrong because it's not what KAISER does (though it would prevent Meltdown from working). **D** is wrong because, though KAISER does indeed omit user mappings from the kernel's page table, that doesn't have any effect on Meltdown. **E** is wrong because, as the paper points out, KASLR does not prevent Meltdown from working (also KAISER doesn't enable KASLR).

X 6.1810

12. [2 points]: What lab did you learn the least from (net, lock, fs, or mmap), and why?

Answer: net (x62): more about hardware than xv6, want more guidance, poor alignment with lectures. lock (x49): simple, didn't dig into code, overlap with other classes. fs (x27): straightforward, didn't touch the hard parts about logging. mmap (x26): just more VM stuff that was already covered in many labs, lots of debugging.

13. [2 points]: Which of the papers (Livelock, Shenango, ext3, RCU, containers, BPF, Meltdown) do you think we should delete next year, and why?

Answer: Containers (x44). Shenango (x42). BPF (x35). Meltdown (x23): already read the paper in other classes. ext3 (x11). RCU (x8). Livelock (x2).

Livelock and Shenango are a bit redundant because they're both networking (x3).

End of Quiz II — Happy holidays!