*Department of Electrical Engineering and Computer Science*

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

**6.1810 Fall 2025**

# Quiz I Solutions

Mean 39.69 Median 41.5 Standard deviation 8.51 (out of 52)

# I System calls

The following UNIX program is executed:

```
int main() {
  int fds[2];
  pipe(fds);
  fork();
  write(fds[1], "x", 1);
  close(fds[1]);
  char buf[32];
  int n = read(fds[0], buf, sizeof(buf));
  printf("%d\n", n);
  exit(0);
}
```

None of the system calls return failure (i.e. none of them return -1).

**1. [5 points]:** The program prints two numbers. Given the description of UNIX system calls in Chapter 1 of the xv6 book, what outputs are possible? Circle all that apply.

**A.** 2 0

**B.** 0 0

**C.** 1 1

**D.** 2 2

**E.** 0 1

**Answer: A** and **C**. Two bytes are written to the pipe, so the numbers cannot sum to more than two; thus **D** can't be right. **B** can't be correct because if one process didn't read any bytes, the other must have read at least one byte (the one it wrote). **E** can't be right because a process can't read zero bytes from a pipe until all write FDs to the pipe have been closed; this can only happen after both processes write to the pipe; if one process read 0, the other must read 2.

# II Sandboxing

Consider the xv6 system call sandboxing implementation in lab `syscall`, where `interpose()` takes an allowed pathname argument.

**2. [5 points]:** Suppose that process A, sandboxed by calling `interpose(1 << SYS_open, "foo")`, executes the following code:

```
strcpy(0x4000, "foo");
open(0x4000, 0);
```

After the `syscall()` function checks that the pathname is allowed, but before the `sys_open()` function runs, process B running on another core executes the following code:

```
strcpy(0x4000, "bar");
```

Can this cause the sandboxed process A to open the file `"bar"`? Explain why or why not.

**Answer:** No, the other process has a separate address space that does not affect the contents of 0x4000 in the sandboxed process.

**3. [5 points]:** Suppose that Ben Bitdiddle sandboxes a process, preventing it from invoking `sys_open()`, but does not prevent it from calling `sys_read()` and `sys_write()`. What files can the sandboxed process read and write, if any?

**Answer:** The sandboxed process can read and write files for which it receives open file descriptors.

# III Page tables

Figure 1 shows how RISC-V translates virtual addresses to physical addresses and the format of a Page Table Entry (PTE).

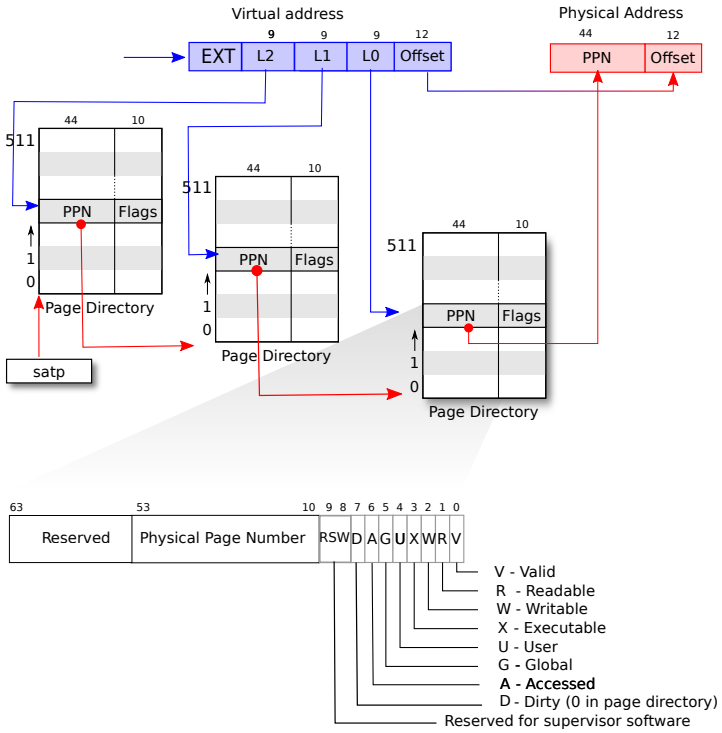


Figure 1: RISC-V address translation details.

*(continued on the next page.)*

**4. [4 points]:** Circle the virtual addresses below such that, if code running on the CPU accesses a byte of memory at that address, it would cause the processor to load index 256 from the L0 page directory (the rightmost out of the 3 shown in the figure above). You can assume the virtual address has a valid mapping in the page table. Circle all that apply.

**A.** `0xfffff100`

**B.** `0x00100000`

**C.** `0xfff00fff`

**D.** `0x10101010`

**Answer:** 0x00100000 and 0xFFF00FFF. Accessing index 256 requires the L0 bits in the VA to be 256=0x100, which can be combined with an arbitrary offset (zero in our first answer), and arbitrary L2 and L1 bits (again zero in our first answer; we assume it's mapped in those page directories).

**5. [4 points]:** Circle the virtual addresses below such that, if code running on the CPU accesses a 64-bit integer at that one address, it would cause the processor to load indexes 510 *and* 511 from the L1 page directory (the middle out of the 3 shown in the figure above). You can assume the memory accessed is present in the page table. Circle all that apply.

**A.** `0x3fdffff0`

**B.** `0x3fdfffff`

**C.** `0x3fdfeffc`

**D.** `0x3fffeffe`

**Answer:** 0x3fdfffff. Accessing the first byte at this address gives a VA that has L1 bits of 0x1fe (index 510), and accessing the next 7 bytes to load the rest of the 64-bit integer leads to L1 bits of 0x1ff (index 511). The L0 and offset bits have to be high enough (in our example, all ones) such that the first and last bytes of the 64-bit (8-byte) load have different L1 bits.

# IV Traps

The trampoline code in `uservec` in trampoline.S saves all registers, including ones that are caller-saved (e.g., `t0`). Ben observes that system calls in user code take the form of function calls to stubs in `usys.S`, and the compiler always generates user code for those function calls that saves and restores caller-saved registers (if those registers hold anything useful). Since user code has already saved caller-saved registers before each system call's `ecall`, Ben wonders if he could make traps faster by not saving and restoring caller-saved registers in `uservec` and `userret`, respectively.

Ben modifies `uservec` and `userret` to not save and restore caller-saved registers. To test his modified xv6, he runs this program multiple times:

```
int main(int ac, char **av) {
  if(fork() == 0){
    char buf[512];
    int fd = open("README", 0);
    read(fd, buf, sizeof(buf));
    close(fd);
    exit(0);
  } else {
    printf("5 * 5 = %d\n", 5 * 5);
  }
  exit(0);
}
```

The program often malfunctions, printing a value other than 25, or printing gibberish.

**6. [5 points]:** Explain why Ben's modification to xv6 causes the program to malfunction.

**Answer:** The user code for `printf()` might be interrupted at any point by a disk or timer device interrupt. With Ben's modification, interrupts at those points will return to user code with altered caller-saved registers, which are likely to cause `printf()` to execute incorrectly.

# V Lazy page allocation

Ben deletes the following code from `copyin` in kernel/vm.c:

```
if(pa0 == 0) {
  if((pa0 = vmfault(pagetable, va0, 0)) == 0) {
    return -1;
  }
}
```

**7. [5 points]:** Write a sequence of system calls, as user code in C, that would ordinarily cause no problems, but that would cause a kernel with the above modification to panic. Briefly explain your answer.

**Answer:**

```
char *p = sbrklazy(4096);
write(1, p, 1);
```

This code passes a pointer to lazily-allocated memory for which there won't be a mapping in the user page table. The `write()` system call will call `copyin()` for this address; `walkaddr()` will return zero; and the subsequent `memmove()` will fault.

# VI Page Size

Alyssa P. Hacker is interested in the effect of page size on performance. She has just implemented copy-on-write fork in xv6, so she decides to investigate the effect of page size on COW fork. She finds a variant of the RISC-V CPU that allows selection of one of a range of page sizes (512 bytes, 1024, 2048, 4096, 8192, ..., up to 2 megabytes). She further modifies her xv6 to support a configurable page size. At any given time there is only one page size, but Alyssa can change the page size by modifying the kernel source and rebooting.

Alyssa times how long the following program takes to run, as her performance benchmark. As you can see, the program executes a loop 50 times; each iteration of the loop forks a child, which modifies one byte of memory and then exits. Alyssa's computer has more than enough physical memory for this program.

```
char buf[10*1024*1024];

int main() {
  for(int iterations = 0; iterations < 50; iterations++){
    int pid = fork();
    if(pid == 0){
      buf[sizeof(buf) / 2] = 99;
      exit(0);
    }
    int st;
    wait(&st);
  }
  exit(0);
}
```

Alyssa times this program on her COW-fork-enabled kernel using each available page size.

**8. [5 points]:** What trend does Alyssa see, and why, as she varies page size? Circle the one best answer.

- **A.** The program takes more time as the page size increases, because each page fault takes a longer amount of time for the kernel to handle.
- **B.** The program takes less time as the page size increases, because there are fewer page faults.
- **C.** The program takes more time as the page size increases, because there are more page faults.
- **D.** The program takes less time as the page size increases, because the page free list is shorter.

**Answer:** The correct answer is **A**. With COW fork, when the child writes memory, it will incur a page fault, and the kernel will have to allocate a new page for it and copy the old contents to that new page. As the page size increases, the time required for the copy grows.

Alyssa modifies just the indicated line of her benchmark:

```
char buf[10*1024*1024];

int main() {
  for(int iterations = 0; iterations < 50; iterations++){
    int pid = fork();
    if(pid == 0){
      for(int i = 0; i < sizeof(buf); i++) buf[i] = 99; // THIS LINE
      exit(0);
    }
    int st;
    wait(&st);
  }
  exit(0);
}
```

**9. [5 points]:** Now what trend does Alyssa see as she varies page size? Circle the one best answer.

**A.** The program takes more time as the page size increases, because each page fault takes a longer amount of time for the kernel to handle.

**B.** The program takes less time as the page size increases, because there are fewer page faults.

**C.** The program takes more time as the page size increases, because there are more page faults.

**D.** The program takes less time as the page size increases, because the page free list is shorter.

**Answer:** The correct answer is **B**. The same number of bytes in total have to be allocated and copies as a result of COW faults, regardless of page size. But the number of page faults will go down as the page size increases.

# VII Locking

Ben Bitdiddle is implementing a bank-like system that allows transferring credits between accounts. Ben is worried about concurrency, so he protects each account with a spinlock, as follows:

```
struct account {
  uint balance;
  struct spinlock lock;
};

void
transfer(struct account *sender, struct account *recipient, uint amount)
{
  acquire(&sender->lock);
  acquire(&recipient->lock);
  if (sender->balance >= amount) {
    sender->balance = sender->balance - amount;
    recipient->balance = recipient->balance + amount;
  }
  release(&sender->lock);
  release(&recipient->lock);
}
```

You can assume that the total number of credits across all accounts is small enough that balance never overflows.

**10. [5 points]:** What concurrency bug is present in Ben's code? How should he change his code to fix it?

**Answer:** Deadlock. Order lock acquisition by the memory address of the struct account. Or protect the entire system with one global lock, at the cost of reduced concurrency.

# VIII 6.1810

**11. [1 points]:** What lab so far have you found to be the worst for you, and why?

**Answer:** pgtbl (x134). cow (x14). syscall (x6). trap (x6). util (x3). net (x1).

Many specifically said superpages part of pgtbl was the worst. Textbook doesn't mention superpages. Many moving parts involved in superpages. More guidance/hints/unit-tests on superpages; not clear how to get started. Superpages requires changes in many places; not clear what's OK to modify or not. Superpage promotion/demotion was hard. Debugging is too time-consuming. Many said superpages was the worst but also fun, satisfying, learned a lot. COW: hard to track down bugs, grader didn't catch locking bugs, didn't expect that locking mattered yet. Net: reading documentation. Util: didn't understand FDs despite the lab.

**12. [1 points]:** Which of the papers so far (Microkernels, Superpages, Syscall interposition, VM primitives for user programs) do you think we should delete next year, and why?

**Answer:** Microkernels: not used in practice, not in xv6 (x65). VM primitives: old, does not connect to class or labs, confusing (x46). Interposition: not in xv6, lab is enough (x28). Superpages: lab was enough, more about security, dated, hard to read (x20).

Liked interposition (x1). Liked VM primitives (x1).

**13. [1 points]:** What has been the most fruitful way that you've found to use ChatGPT-like tools to help you in this class?

**Answer:** Understanding what some part of xv6 code does, why not do it another way, design questions, faster than reading code myself, quickly find relevant code (x37). Expand/summarize/re-explain/reformat lecture notes, textbook (x36). Summarize/explain papers, how they relate to class (x27). Summarize/explain concepts (x15). C syntax (x10). Like an online TA, 24/7 office-hours, faster / more direct answer than going to office hours, faster and more interactive than Piazza (x7). RISC-V details (x6). Sanity-check understanding of material or labs (x5). Quizzing self to check understanding / generate study guide / generate practice exam (x5). Debugging, sanity-checking code for simple errors (x4). Search through reference material (x2). Asking questions/clarifications (x2). Asking it questions during lecture (x2). Understanding homework questions (x2). Help make sense of debug output (x2). Using GDB (x2). Understand design concepts before diving into xv6 code. Synthesize counter-examples to understand why something is necessary. Answer "what if" questions. Asking hypotheticals about why something has to be done (caller-saved reg, syscall/intr trapframes, etc). Explain answers to past exams. Preparing for exam. Unix shell commands. Discover adjacent ideas. Sounding board for ideas. Generate diagrams to explain page tables.

Have not used (x13). Also: have gotten bogus answers from ChatGPT.

**14. [1 points]:** Since ChatGPT-like tools can help answer many questions related to this class, in what ways do you think the course staff can be most useful in helping you learn the class material?

**Answer:** More office hours / TAs, especially when too confused to formulate ChatGPT question (x26). Debugging, GDB help, lab guidance, hints (x21). More intuition, structure, design rationale, motivation, connect to applications (x18). Recitations: diving into details, repeating concepts, problem-solving sessions, quiz review (x11). More detailed / nicer formatted lecture notes (x9). Check-offs (x9). Draw more diagrams, visualizing concepts (x9). More examples / demos, make demos available (x7). Use ChatGPT in office hours, show how students can best use ChatGPT (x7). Lecture videos (x6). More hands-on homework questions separate from labs (x5). More labs, open-ended lab options, harder labs (x4). More practice quiz problems (x3). Feedback on submitted homework (x3). Discuss trade-offs (x2). More thought-provoking brain-teaser questions (x2). Walk through and discuss solutions after each lab (x2). Allow ChatGPT on exam. Code review. Something like Python Tutor. Challenges/extensions of course material. Make reading questions more applied. Critiquing student's understanding. Guiding questions. Better summaries of tricky code. Give problem sets designed for ChatGPT. Discuss challenge questions at the end of homeworks/labs. More interactive lectures. More case studies. Lab FAQs like 6.5840. Recommend other papers to read. Generate GPT summaries of lectures. Correct conceptual explanations unlike ChatGPT. Connect xv6 principles to real OSes. ChatGPT gives too much info.

# End of Quiz I