Department of Electrical Engineering and Computer Science

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

6.1810 Fall 2025

Quiz I

All problems are open-ended questions. In order to receive credit you must answer each question as precisely as possible. You have 80 minutes to finish this quiz.

Write your name on this cover sheet AND at the bottom of each page of this booklet.

Some questions may be harder than others. Read them all first and attack them in the order that allows you to make the most progress. If you find a question ambiguous, write down any assumptions you make. Be neat. If we can't understand your answer, we can't give you credit!

THIS EXAM IS OPEN BOOK AND OPEN LAPTOP, but CLOSED NETWORK.

Gradescope E-Mail Address:

Name:

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

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.

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?

III Page tables

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

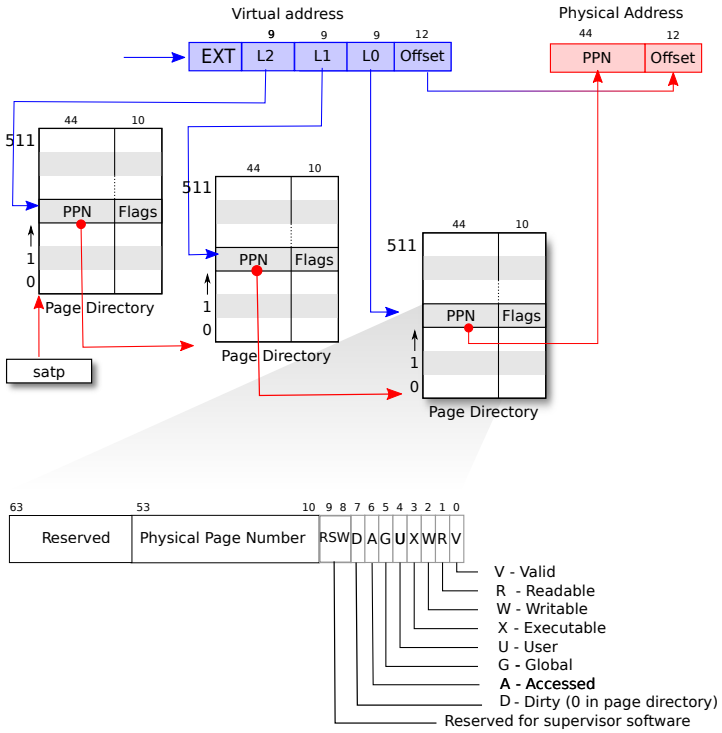

Figure 1: RISC-V address translation details.

(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`

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`

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.

Name:

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.

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.

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.

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?

VIII 6.1810

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

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?

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?

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?

End of Quiz I