Department of Electrical Engineering and Computer Science

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

6.1810 Fall 2024

Quiz I Solutions

Mean 42.75 Median 42 Standard deviation 9.35 (out of 61)

I System Calls

Suppose that you run the following program on xv6. (See Chapter 1 of the xv6 book for a description of the system calls.)

```
int main() {
  if(fork() == 0){
    write(1, "x", 1);
  }
  if(fork() == 0){
    write(1, "y", 1);
  }
  wait(0);
  wait(0);
}
```

None of the system calls fail.

1. [3 points]: Which of the following outputs could this program produce? Circle all that apply.

A. `xy`

B. `yx`

C. `xyy`

D. `yxy`

E. `yyx`

Answer: **C** and **D**. After the first fork, the parent and child execute in parallel, so the x could be printed before or after the first y. Two y's will be printed, because both the first child and the parent will call the second fork.

Now consider running this xv6 program:

```
char aaa = 'w';
char bbb;

int
main() {
  int fds[2];

  aaa = 'x';

  pipe(fds);
  if(fork() == 0){
    write(fds[1], &aaa, 1);
    bbb = 'y';
    while(1){
      sleep(1);
    }
  } else {
    aaa = 'z';
    read(fds[0], &bbb, 1);
    write(1, &bbb, 1);
  }
}
```

No system calls fail.

2. [5 points]: What are the possible outputs? Circle all that apply.

A. Nothing

B. x

C. y

D. z

Answer: B. Only x will be printed. `fork` gives the child process an exact copy of the calling parent process's memory, so local updates to variables will not be reflected in other processes.

II Page tables

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

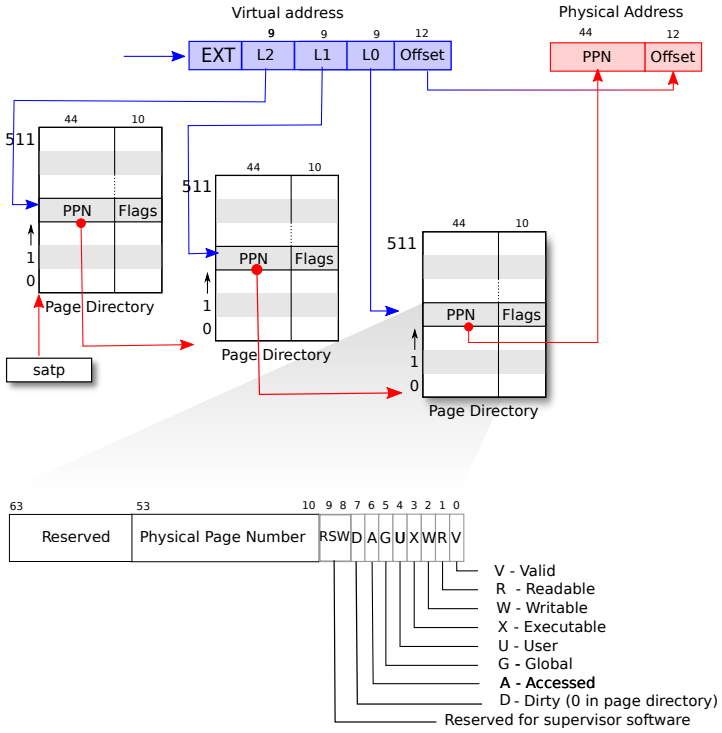

Figure 1: RISC-V address translation details.

3. [5 points]: For the virtual address 0xFFFFD000 what is the index into the L0 page directory? Please write down a hex number.

Answer: 0x1FD. The L0 index is in bits 12 to 21 of the virtual address: (0xFFFFD000 >> 12) & 0x1FF.

4. [5 points]: Ben creates a page table that has PTE 0x21FD10D7 for virtual address 0 and PTE 0x21FD10D6 for virtual address 4096. If Ben stores the value 1 to virtual address 0 and then loads from virtual address 4096, does the load return the value 1? Circle the one best answer.

A. Yes

B. No

Answer: No, because PTE 0x21FD10D6 doesn't have the valid bit set.

III Superpages

Ben observes that KERNBASE is 0x80000000 and PHYSTOP is 0x88000000 in the two calls to kvmmap shown below on line 20 and 23.

```
1  // Make a direct-map page table for the kernel.
2  pagetable_t
3  kvmmake(void)
4  {
5    pagetable_t kpgtbl;
6
7    kpgtbl = (pagetable_t) kalloc();
8    memset(kpgtbl, 0, PGSIZE);
9
10   // uart registers
11   kvmmap(kpgtbl, UART0, UART0, PGSIZE, PTE_R | PTE_W);
12
13   // virtio mmio disk interface
14   kvmmap(kpgtbl, VIRTIO0, VIRTIO0, PGSIZE, PTE_R | PTE_W);
15
16   // PLIC
17   kvmmap(kpgtbl, PLIC, PLIC, 0x4000000, PTE_R | PTE_W);
18
19   // map kernel text executable and read-only.
20   kvmmap(kpgtbl, KERNBASE, KERNBASE, (uint64)etext-KERNBASE, PTE_R | PTE_X);
21
22   // map kernel data and the physical RAM we'll make use of.
23   kvmmap(kpgtbl, (uint64)etext, (uint64)etext, PHYSTOP-(uint64)etext, PTE_R | PTE_W);
24
25   // map the trampoline for trap entry/exit to
26   // the highest virtual address in the kernel.
27   kvmmap(kpgtbl, TRAMPOLINE, (uint64)trampoline, PGSIZE, PTE_R | PTE_X);
28
29   // allocate and map a kernel stack for each process.
30   proc_mapstacks(kpgtbl);
31
32   return kpgtbl;
33 }
```

5. [5 points]: How many physical pages of memory do these two kvmmap calls allocate for page directories to map this region of memory? Circle the one best answer.

A. 65

B. 32768

Answer: A (65). 0x8000000 bytes is 32768 pages, so 32768 PTEs are required. Each PTE uses 8 bytes, totaling 262144 bytes, or 64 4096-byte pages of PTEs. Plus one PTE to hold the L1 page directory.

Eve observes that that these two calls map physical memory contiguously and modifies xv6 to use superpages for this region of memory.

6. [5 points]: How many physical pages of memory does Eve's modified xv6 allocate for page directories to map this memory? Circle the one best answer.

A. 1

B. 65

Answer: A (1). Given that KERNBASE is a 2MB-aligned address, one L1 page directory can map this region with 64 2MB superpages.

IV Traps

xv6 user code uses the RISC-V `ecall` instruction to start a system call. Among other things, `ecall` jumps to the start of the trampoline code.

Ben wonders whether it would work to replace `ecall` with `jal 0x3ffffff000`, a RISC-V subroutine call instruction that jumps to the start of the trampoline. (In fact, Ben would need a few more instructions to load the 64-bit 0x3ffffff000 into a register, followed by `jalr`.)

7. **[5 points]:** Why won't Ben's idea work? Circle all that apply.

   A. ecall is needed because ecall changes `satp` to point to the kernel page table.

   B. ecall is needed because ecall saves the 32 user registers.

   C. ecall is needed because the trampoline's address has no PTE in the user page table.

   D. ecall is needed because the address of the trampoline page is different for different processes.

   E. none of the above.

Answer: E.

When a device interrupt occurs while user code is executing, `usertrap()` in `kernel/trap.c` saves the RISC-V `sepc` register, which holds the program counter value at which the interrupt occured:

```
// save user program counter.
p->trapframe->epc = r_sepc();
```

When returning to user space, `usertrapret()` copies that saved value back to `sepc`:

```
// set S Exception Program Counter to the saved user pc.
w_sepc(p->trapframe->epc);
```

In many cases the value in the `sepc` register doesn't change during interrupt handling, so that this save and restore doesn't actually change the value in `sepc`.

8. [5 points]: There is a situation in which, after a process traps into the kernel due to a device interrupt, the `sepc` register will have changed by the time of that process's return via `usertrapret()`. Please briefly explain what that situation is.

Answer: If the device interrupt in question is a timer interrupt, then `usertrap()` will call `yield()`, and other processes will run. Those processes will change the `sepc` register if they return to user space or encounter interrupts while in the kernel. When the scheduler finally switches back to the original process, `sepc` will differ from `p->trapframe->epc`.

V Threads

`scheduler()` in `kernel/proc.c` context-switches with this call:

```
swtch(&c->context, &p->context);
```

Ben accidentally changes `c` to `p`:

```
swtch(&p->context, &p->context);
```

so that now (ignoring comments) the `scheduler()` loop looks like this:

```
for(;;){
  intr_on();
  for(p = &proc[0]; p < &proc[NPROC]; p++) {
    acquire(&p->lock);
    if(p->state == RUNNABLE) {
      p->state = RUNNING;
      c->proc = p;
      swtch(&p->context, &p->context);
      c->proc = 0;
    }
    release(&p->lock);
  }
}
```

What will go wrong when Ben runs xv6? Circle the one best answer.

9. [5 points]:

A. `scheduler()`'s call to `swtch()` will context-switch to a different kernel thread, but that thread will soon crash or panic because the register contents are incorrect.

B. `scheduler()` will loop forever without ever context-switching to a different kernel thread.

C. `swtch()` will context-switch to a different kernel thread, which will execute correctly until it tries to `swtch()` back to the scheduler thread, at which point the fact that `c->context` was never initialized will cause a crash or panic.

D. `scheduler()`'s call to `swtch()` will run the process `p` twice, instead of just once, giving it an unfair advantage.

E. `scheduler()` will likely `swtch()` to a kernel thread that is already running on a different CPU.

Answer: B. The `swtch()` call first saves the *scheduler's* registers (including `ra`, which refers to `scheduler()`) into `p->context`, then restores those same registers. As a result, `swtch()` returns to `scheduler()` rather than switching to process `p`.

An xv6 process that is giving up the CPU calls `sched()`, which calls `swtch()`. Alyssa wants to know what address that `swtch()` call will return to. She would like to add a `printf()` just before the `swtch()` in `sched()` to print that address, so that `sched()` looks like this:

```
printf("%p\n", (void *) ???);
swtch(&p->context, &c->context);
```

10. [5 points]: What value should Alyssa print in order to find out where `sched()`'s call to `swtch()` will return? Circle the one best answer.

A. `p->context.ra`

B. `&scheduler`

C. `r_sepc()`

D. `c->context.ra`

E. `p->trapframe->ra`

F. `p->trapframe->epc`

Answer: D. `swtch()` will load registers from its second argument, `c->context`, so it will return to `c->context.ra`.

Ben is using xv6 on a computer with just a single CPU (i.e. a single RISC-V hart). He thinks that, because threads cannot ever run truly in parallel with only one CPU, that therefore `acquire()` and `release()` are not needed. Ben modifies his kernel's `acquire()` and `release()` functions to simply return without doing anything, and he also eliminates the three checks and panics in `sched()` (since they are checking things to do with locks).

Ben has overlooked something important about the effects of xv6's spinlocks even when there is just one CPU!

11. [5 points]: Explain briefly why Ben's modifications will result in a broken kernel even though he only runs it on machines with a single CPU.

Answer: `acquire()` disables interrupts as well as locking. There are places in xv6 where correct operation requires interrupts to be off even with a single CPU, for example to prevent a timer interrupt (and consequent `yield()` and `swtch()`) from happening during `swtch()`.

VI Copy-on-write fork

Ben's solution for COW-fork marks in `fork` all the pages below `p->sz` in the parent and child as read only. On a page fault, Ben's solution checks if the faulting address is below `p->sz`, and, if so, allocates a new page, copies the faulting page into the new page, and inserts the new page with read-write permissions in the process's page table. Ben's solution passes `cowtest`.

12. [5 points]: Briefly explain why Ben's solution is incorrect despite passing `cowtest`.

Answer: Ben's solution would result in an initially read-only page having read and write permissions after a store page fault to that page. For example, if a process after a `fork` writes to a text page, which initially had read-only permissions, the write succeeds and modifies the text page, which is undesirable. Similarly, the guard page would get write permissions and wouldn't protect against stack overflow.

VII 6.1810

13. [1 points]: Which parts of the xv6 textbook were most helpful? Which parts did you think were most confusing?

Answer:
Helpful:

- Diagrams - 47
- Pagetables - 12
- Locks - 7
- Traps - 6
- Code snippets and explanations - 6
- Scheduling - 5

Confusing

- Scheduling - 17
- Long explanations with no diagrams/code - 16
- Code snippets - 15
- Locking - 10
- Devices - 10
- Traps - 6

14. [1 points]: Please indicate which of the labs you found to be the most helpful in learning the material, and which the least.

A. Utilities

Answer: Helpful: 2; Unhelpful: 34

B. Sys calls

Answer: Helpful: 5; Unhelpful: 17

C. Page tables

Answer: Helpful: 30; Unhelpful: 14

D. Traps

Answer: Helpful: 13; Unhelpful: 8

E. Copy-on-write fork

Answer: Helpful: 43; Unhelpful: 5

15. [1 points]: What's the most important thing we could fix about 6.1810 to make it better?

Answer:

- More TAs/OH - 18
- More explanations/walkthroughs of the xv6 codebase - 13
- More relevant exam questions/exam review sessions - 12
- More details/formatting in lecture notes - 10
- More visual components in lectures - 7

End of Quiz I