CS 134 Midterm Solution March 11, 2019

Name: _____

Answer the questions in the spaces provided on the question sheets. If you run out of room for an answer, continue on the back of the page.

You may use any books or notes you wish. You may also use a computer to access the class website (any of the pages hosted at `https://www.cs.hmc.edu/~rhodes/cs134`, plus any reachable in one click from any of those pages) and your github Jos repository. No other use of the computer is allowed. (For example, no browsing source code locally, no use of calculators, no playing music or checking email, no browsing of any other websites.)
You may bring a laptop to the in-class midterm, or you may choose to take the midterm in the B105 lab and use the computers there. If the latter, arrive to class at 1PM to receive the exam. Begin the exam at 1:15, Finish at 2:30, and then return the exam to the classroom.

If you think something about a question is open to interpretation, ask. If necessary, write any assumptions you've made as part of answering the question.

Be concise in your answers; you need not try to fill in all or even most of the lines provided for an answer.

| Question: | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | Total |
|---|---|---|---|---|---|---|---|---|---|---|
| Points: | 12 | 12 | 15 | 6 | 6 | 6 | 6 | 6 | 6 | 75 |
| Score: | | | | | | | | | | |

1. Assume an xv6 file system with the following layout (note the following sizes—blocks: 512 bytes, inodes: 64 bytes, directory entries: 16 bytes , inode numbers: 2 bytes, and block numbers: 4 bytes):

| block num | usage |
|---|---|
| 0 | unused (usually boot block) |
| 1 | super block |
| 2...31 | log for transactions |
| 32...57 | array of inodes, packed into blocks |
| 58 | block in-use bitmap (0=free, 1=used) |
| 59... | file or directory content blocks |

Assume only a single directory (the root).

3 points (a) What is the maximum size a file can be?

**Solution:** There are $12$ block references in an inode, and one single-indirect block that refers to another $512/4 = 128$ blocks. So there are a total of $140$ blocks. With $512$ bytes per block, the maximum file size is $140 \times 512 = 7168$ bytes.

3 points (b) How many such maximum-size files can there be (and what is the limiting factor?)

**Solution:** Each file takes $141$ blocks ($140$ data blocks + one indirect block). There are a maximum of $512 \times 8 = 4096$ blocks addressable by the block in-use bitmap. One block is used by the root directory. So, we could support $\lfloor 4096/141 \rfloor = 29$ maximum-size files (whose directory entries could fit in the single root directory disk block).

The limiting factor is the block in-use bitmap.

3 points (c) What is the minimum size a file can be?

**Solution:** A file can by 0 bytes long

3 points (d) How many such minimum-size files can there be (and what is the limiting factor?)

**Solution:** There would be no file content blocks, and only a single directory content block. There are a maximum of $512 \times 8 = 2048$ blocks addressable by the block in-use bitmap. Each such block could support $512/16 = 32$ directory entries. So, an upper-bound on the number of directory entries is $2048 \times 32$. We'd have to subtract the two directory entries for `.` and `..`

However, a file/directory can't be 2048 blocks big. From part (a) we know the directory can contain at most 140 blocks. With $32$ directory entries per block, we'd have a total of $140x32 - 2 = 4478$ files ($-2$ for `.` and `..` entries).

A third limitation is the inodes. Each file would consume a single inode. There are $512/64 = 8$ inodes per block and 26 inode blocks for a total of $26 \times 8 = 208$ inodes. One of the inodes is used by the directory, so there are $207$ left for files. Thus, we could support 207 minimum-length files and the number of inodes would be the limiting factor.

However, if we did a hard-link so that every file had the same inode (possible since they all have the same contents), the number of inodes would no longer be a limiting factor. Instead, the size of the root directory would, leading to a final answer of $4478$.

2. Imagine a simplified x86 architecture with a single-level page table, 4-bit page number, and 4-bit offset. There are still the standard segments: code, data, stack, and other. We have 256 bytes of physical memory

   An 8-bit virtual address consists of:

   | 4-bit page # | 4-bit offset |
   |---|---|

   Each page table entry (PTE) consists of:

   | 4-bit frame # | 2-bits unused | PTE_W | PTE_P |
   |---|---|---|---|

   Here are the relevant tables for a specific process:

   Segment Table:

   | Segment | Base address | Length |
   |---|---|---|
   | code (CS) | 0x10 | 0x8 |
   | data (DS) | 0xF0 | 0x10 |
   | stack (SS) | 0x30 | 0x40 |

   Page Table:

   | Page #: | Frame # | PTE_W | PTE_P |
   |---|---|---|---|
   | 0 | `0xA` | 1 | 1 |
   | 1 | `0x3` | 1 | 1 |
   | 2 | `0x8` | 1 | 1 |
   | 3 | `0x2` | 1 | 1 |
   | 4 | `0x1` | 1 | 1 |
   | 5 | `0x0` | 0 | 0 |
   | 6 | `0x6` | 1 | 1 |
   | 7...14 | `0x0` | 0 | 0 |
   | 15 | `0xB` | 1 | 1 |

   For the following, find the physical address (or `INVALID`):

3 points (a) If the instruction pointer (IP) were 0x06, what physical address would it refer to?

**Solution:** The segment register used for the IP is the code segment (CS). The max length is 0x8 bytes, and the IP doesn't exceed that. The base address is 0x10, so we construct the linear address by adding together the base address with the VA: 0x10 + 0x06=0x16.

We convert a linear address to a physical address using the page table. The high nibble (half-byte) of the linear address is 1, so we use the entry at location 1 of the page table and find that the entry is present. The frame number from that entry is 3, so we concatenate the frame number of 3 with the offset of 6 to obtain a PA of 0x36.

3 points (b) If the instruction pointer (IP) were 0x16, what physical address would it refer to?

**Solution:** The segment register used for the IP is the code segment (CS). The max length is 0x8 bytes, but the IP is greater than that. So, this is an INVALID address.

3 points (c) If the stack pointer (SP) were 0x18, what physical address would it refer to?

**Solution:** The segment register used for the SP is the stack segment (SS). The max length of SS is 0x40 bytes, and the SP doesn't exceed that. The base address is 0x30, so we construct the linear address by adding together the base address with the VA: 0x30 + 0x18=0x48.

We convert a linear address to a physical address using the page table. The high nibble of the linear address is 4, so we use the entry at location 4 of the page table and find that the entry is present. The frame number from that entry is 1, so we concatenate the frame number of 1 with the offset of 8 to obtain a PA of 0x18.

3 points (d) If the stack pointer (SP) were 0x28, what physical address would it refer to?

**Solution:** The segment register used for the SP is the stack segment (SS). The max length of SS is 0x40 bytes, and the SP doesn't exceed that. The base address is 0x30, so we construct the linear address by adding together the base address with the VA: 0x30 + 0x28=0x58.

We convert a linear address to a physical address using the page table. The high nibble of the linear address is 5, so we use the entry at location 5 of the page table and find that PTE_P is not set. Thus, the address is INVALID

3. This question looks at different processes running on xv6, and what happens to those processes after a specified event occurs.

   Here is the code for process X running on xv6:

```
main()
{
  while(1) {
  }
  printf(1, here);
  exit();
}
```

   As you can see, process X is in an infinite loop. The event is that another process Y calls (and completes) the `kill()` system call to terminate process X.

   Is it possible for process X to execute any more user-space instructions after the event completes? Explain. If `yes`, what will finally cause X to stop executing user-space instructions, and will X print anything?

3 points (a) If the xv6 system has 1 CPU?

**Solution:**

No. If process Y is running, process X cannot be running.

The only way process X can not be running is if it had been:

- interrupted by a timer interrupt, causing `trap()` to call `yield()` at line 3475, or
- sleeping as a result of some system call.

In the first case, when the scheduler tries to run X, X's kernel thread will resume in the `yield`, and `trap()` will then check in line 3478 whether the process has been killed, Since it has, line 479 will exit the process, never returning.

In the second case, when `kill()` marks the sleeping process X `RUNNABLE`, X will wakeup from sleep and return the the caller of sleep. Some callers may exit the sleep loop; others (like `iderw`) may not immediately return, but only return once the needed I/O is done. Eventually, though the system call that was underway will be finished. At that point, line 3407 will return, and `trap` will find that the process has been killed and will exit, never returning.

3 points (b) If the xv6 system has 2 CPUs?

**Solution:** Yes. If Process X is running in user space on a different CPU than Y, it will continue to execute until the next hardware interrupt (like the timer interrupt) on its CPU. At that point,`trap()` will see that Xs `p->killable` flag is set and call `exit()` at 3469.

Process X will not print anything because it'll never leave the infinite loop.

Here is the code for process U running on xv6:

```
main()
{
  kill(getpid());     // The completion of the kill
                      // call is the event
  printf(1, "here");
  exit();
}
```

3 points (c) Is it possible for process U to execute any more user-space instructions after the event completes? Explain. If `yes`: what will finally cause U to stop executing user-space instructions, and will U print anything?

**Solution:** No. After process U calls kill on its own PID, `exit` will never return. Line 2663 sets the state of the process to `ZOMBIE`, and then calls the scheduler, which'll never run a `ZOMBIE` process.

Here is the code for process V running on xv6:

```
main()
{
  * (char *)  0xf0000000 = 'a';  // The assignment is the event
  printf(1, "here");
  exit();
}
```

3 points (d) Is it possible for process V to execute any more user-space instructions after the event completes? Explain. If `yes`: what will finally cause V to stop executing user-space instructions, and will V print anything?

**Solution:** No. The assignment wil cause a page fault (because V is running in user-mode, and the given address doesn't have `PTE_U` set). Line 3458 of `trap` will set the process to killed. Lines 3468 and 3469 will then exit the newly-killed process.

Here is the code for process W running on xv6 (assume `foo` is a file of length 10):

```
main()
{
  int fd = open("foo", O_RDONLY);
  read(fd, (char *) 0xf0000000, 5); // The completion of the
                                    // read call is the event
  printf(1, "here");
  exit();
}
```

3 points

(e) Is it possible for process W to execute any more user-space instructions after the event completes? Explain. If `yes`: what will finally cause W to stop executing user-space instructions, and will W print anything?

**Solution:** Yes. Passing an invalid address to the `read` system call will cause `read` to return an error (lines 6138-6139); it *w*on't exit the process. W will print `here`.
W will stop executing user-space instructions after its call to `exit()`.

6 points 4. What would go wrong if you replaced `pushcli()`'s implementation (xv6 line 1667) with just `cli()`, and `popcli()`'s implementation (xv6 line 1679) with just `sti()`?

**Solution:** If a kernel thread acquires two locks and then releases one, `release()` would cause interrupts to be turned on. Then an interrupt could occur, and if the interrupt handling code tried to acquire the one lock that is still held, `acquire()` would panic.

6 points 5. Given a user-provided pointer to a buffer in user memory (such as the argument of a system call), explain what checks and translations the JOS kernel must do to ensure that it can safely read or write the buffer memory. Do not assume or rely on the existence of a function in the kernel that does these checks and translations for you.

**Solution:** The JOS kernel does not need to translate the address, since the environment's memory is mapped into the kernel at the same addresses that the environment itself uses. JOS needs to make three safety checks before it can dereference a pointer supplied by the environment without fear of incurring a page fault or reading or writing outside the environment's memory.

1. The `PTE_U` flag must be set in the corresponding PTE (equivalently, the address must be less than `UTOP`).
2. The `PTE_P` bit must be set in the PTE.
3. If JOS needs to write through the pointer, the `PTE_W` bit must be set.

The above three conditions need to hold for each page of the environment's buffer.

6 points 6. Does the JOS kernel have a mechanism equivalent to xv6's `swtch` (xv6 line 3058)? If yes, what? If not, explain why xv6 needs it but JOS does not.

**Solution:** Answer: No. xv6 must switch between the stacks of kernel threads, while JOS has only one kernel stack.

6 points 7. xv6 enables interrupts in the kernel during system calls and device interrupts, which adds some complexity since xv6 has to carefully disable and enable interrupts when locking. In contrast, JOS (as you will find in Lab 4 Part C) only enables interrupts in user space, and arranges for the hardware to automatically disable interrupts when entering the kernel. Would anything go wrong if xv6 also disabled interrupts in the kernel? Support your claim.

**Solution:** Answer: Yes. Suppose we have a single-CPU system, there is only one running or runnable process and it performs a disk read. The kernel will issue the IDE request and that kernel thread will go to sleep to wait for the IDE interrupt. Since that was the only runnable process, the kernel will enter the idle loop and never wake up because it will never receive the IDE interrupt.

(It is not true that blocking system calls simply stop working. A blocking system call will still go to sleep, which will schedule another runnable user process and return to user space, where interrupts will be delivered. The danger is if there are no runnable user processes, in which case you stay in the kernel's idle loop with interrupts disabled.)

6 points 8. Suppose you wanted to modify xv6 so that a user process can address more than 2GB of virtual memory. `KERNBASE` is mapped at 2GB, and you could increase it. However, the macros `V2P()` and `P2V()` wouldn't work correctly anymore, because they assume that the kernel has all of physical memory mapped at `KERNBASE`. Assuming you can't port xv6 to a 64-bit architecture, how could you modify xv6 to support user processes with more than 2GB of memory while still allowing the kernel to to address all of physical memory? What's the least amount of address space in the kernel that would be needed to address all of physical memory?

**Solution:** xv6 could temporarily map each physical address at the time the kernel needs to access it. This could be done in a dynamically managed region of kernel virtual memory that is smaller than `KERNBASE`. At a minimum, the kernel would need a single page (4Kb) of virtual address that could be remapped to the desired physical page. Two pages might be easier (so that copying could be done between the two physical pages without having to use an intermediate buffer).

9. This question is a survey for which I pay midterm points:)

2 points (a) Describe the most memorable error you have made so far in one of the labs. (Provide enough detail so I can understand your answer.)

**Solution:**

I would like to hear your opinions of CS 134 so far. Please answer the following two questions (any answer except no answer at all will receive full credit).

2 points (b) What is the best part of CS 134?

**Solution:**

2 points (c) What is the worst part of CS 134?

**Solution:**