Department of Electrical Engineering and Computer Science

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

6.828 Fall 2016

Quiz I

All problems are open-ended questions. In order to receive credit you must answer the 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 through first and attack them in the order that allows you to make the most progress. If you find a question ambiguous, be sure to 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.

Please do not write in the boxes below.

I (xx/5)	II (xx/10)	III (xx/10)	IV (xx/15)	V (xx/5)	VI (xx/5)	VII (xx/15)	Total (xx/65)

Name:

I Context Switch

Homework 8 (user-level threads) uses this structure to hold information about each thread:

```
struct thread {
  int        sp;                /* curent stack pointer */
  char stack[STACK_SIZE];       /* the thread's stack */
  int        state;             /* FREE, RUNNING, RUNNABLE */
};
```

Ben Bitdiddle wonders why the `sp` element of the struct is needed. He plans to delete `sp` from `struct thread`:

```
struct thread {
  char stack[STACK_SIZE];       /* the thread's stack */
  int        state;             /* FREE, RUNNING, RUNNABLE */
};
```

He plans to have `thread_switch()` push the `%esp` register onto the "current" stack and pop it from the "next" stack. And he plans to modify `thread_create()` appropriately.

1. **[5 points]:** Explain why Ben will find it difficult or impossible to make this idea work.

II xv6 Shell, Sleep, and Wakeup

Recall from Homework 2 that the shell uses the `wait()` system call to wait for each command to exit, at which point the shell prints another prompt. Look at the implementation of xv6's `wait()` system call in `proc.c`. Here's an abbreviated version of the xv6 code (with lots of lines omitted):

```
wait(void) {
  for(;;){
    if there is an exited child {
      clean it up ...;
      return pid; // return the child pid
    }

    if(this process has no children || proc->killed){
       ...
       return -1; // error
    }

    // wait for a child to exit.
    sleep(proc, ...);
  }
}
```

2. [5 points]: Is it possible for the `sleep()` to return even if there is no exited child? If yes, how can that happen?

Suppose we re-arranged the code in the `for` loop to first check if the process has children, then sleep, and then check if there is an exited child:

```
wait(void) {
  for(;;){
    if(this process has no children || proc->killed){
       ...
       return -1; // error
    }

    // wait for a child to exit.
    sleep(proc, ...);

    if there is an exited child {
      clean it up ...;
      return pid; // return the child pid
    }
  }
}
```

This modification will cause `wait()` to act incorrectly in some circumstances.

3. **[5 points]:** Explain what can go wrong as a result of the modification.

III xv6 Locking

Ben Bitdiddle thinks that xv6 kernel threads should be able to yield the CPU while holding locks. To test whether this works, he places an `acquire`/`yield`/`release` at the beginning of `syscall()` where it will be called a lot:

```
void
syscall(void)
{
  int num;

  acquire(&test_lock);
  yield();
  release(&test_lock);

  // the rest of syscall()...
}
```

No other code uses `test_lock`. Ben also deletes the panic calls in `sched()` (in `proc.c`), and deletes the `holding()` panics in `acquire()` and `release()`.

Ben boots his modified xv6 on qemu with a single core (`CPUS:=1` in the Makefile). xv6 prints `init: starting sh` but doesn't print a shell prompt; it does nothing, as if in a deadlock. When Ben looks at a backtrace with gdb, he sees the CPU is looping in the `acquire()` call he added to `syscall()`.

4. [5 points]: Explain a sequence of events in Ben's modified xv6 that could cause this to happen.

Now Ben tries his modified kernel on a two-core qemu (`CPUS:=2`). He gets a prompt this time and types a few commands.

5. [5 points]: Can Ben's modification cause a deadlock situation with two cores? Or does using a two-core machine make his modification safe? Explain your answer.

IV JOS Paging (1)

Ben Bitdiddle is finding himself thoroughly confused by page tables and how they work in JOS. He finds that much of his confusion stems from all the bit fiddling that goes on with the `pde_t` and `pte_t` types. Ben decides to rewrite the code to make `pde_t` and `pte_t` (defined in `inc/memlayout.h`) structs with separate `uint32_t` fields for the PDE index and the PTE index, as well as individual fields for the permission bits. He changes all the bit manipulation macros so that they instead modify the appropriate struct fields. Here is Ben's PTE struct definition; his PDE definition is similar:

```
struct pte {
  uint32_t physical_page_number;
  uint32_t available;
  uint32_t dirty;
  uint32_t accessed;
  uint32_t user;
  uint32_t writeable;
  uint32_t present;
};
```

Satisfied with this much more readable code, Ben compiles his kernel and tries to run it. It immediately crashes.

6. [5 points]: Why is it not okay for Ben to modify the `pde_t` and `pte_t` types?

Ben has taken 6.858 (Computer Security), and finds it suspicious that `pgdir_walk` allocates new PDEs with all permissions set (`PTE_W|PTE_U`). He worries that this would mean that all pages will be writeable by user processes.

7. [5 points]: Why is this not the case?

Ben is having unexpected issues with lab 3. His `load_icode` contains the following code

```
elf = (struct Elf *)binary;
ph = (struct Proghdr *) ((uint8_t *) elf + elf->e_phoff);
eph = ph + elf->e_phnum;
for (; ph < eph; ph++) {
  if (ph->p_type != ELF_PROG_LOAD) continue;
  region_alloc(e, (void *) ph->p_va, ph->p_memsz);
  memcpy((void *) ph->p_va, binary + ph->p_offset, ph->p_filesz);
  memset((void *) ph->p_va + ph->p_filesz, 0, ph->p_memsz-ph->p_filesz);
}
```

Ben has carefully traced through his code, and finds that a triple fault occurs in `memcpy`, despite the write going to the address that was just allocated with `region_alloc`.

8. [5 points]: Why is Ben's call to `memcpy` not doing what he expects? Assume that his implementation of `region_alloc` is correct.

V JOS Paging (2)

Ben has a bug in his JOS Lab 3. He boots his kernel and runs a userspace process with the following code at the start:

```
_start:
  800020:  cmpl $USTACKTOP, %esp
  800026:  jne args_exist
  800028:  pushl $0
  80002a:  pushl $0
args_exist:
  80002c:  call libmain
  800031:  jmp 800031
```

Ben gets a page fault in his program, with the trap-frame looking like this:

```
TRAP frame at 0xf01c0000
  edi  0x00000000
  esi  0x00000000
  ebp  0x00000000
  oesp 0xeffffdc
  ebx  0x00000000
  edx  0x00000000
  ecx  0x00000000
  eax  0x00000000
  es   0x----0023
  ds   0x----0023
  trap 0x0000000e Page Fault
  cr2  0xeebfdffc
  err  0x00000006 [user, write, not-present]
  eip  0x00800028
  cs   0x----001b
  flag 0x00000046
  esp  0xeebfe000
  ss   0x----0023
```

9. [5 points]:

What is Ben's bug? What evidence supports your answer?

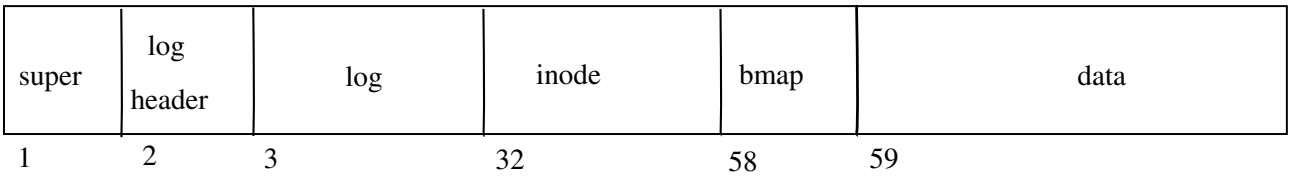
VI Booting

Ben wants to set a break-point inside kern/entry.s because he is unsure if his bootloader is loading and jumping into the kernel correctly. He opens up obj/kern/kernel.asm and sees the address of entry is f010000c, so he sets a breakpoint at 0xf010000c right when GDB starts. However, the breakpoint never hits and Ben is stumped.

10. [5 points]: Why doesn't the breakpoint at 0xf010000c ever trigger?

VII Xv6 file system

Xv6 lays out the file system on disk as follows:



Block 1 contains the super block. Blocks 2 through 31 contain the log header and the log. Blocks 32 through 57 contain inodes. Block 58 contains the bitmap of free blocks. Blocks 59 through the end of the disk contain data blocks.

Ben modifies the function `bwrite` in `bio.c` to print the block number of each block written. He boots xv6 with a fresh `fs.img` and types in the command `echo > x`. This command creates the file `x` but does not write anything into `x`. This command produces the following trace:

```
$ echo > x
write 3
write 4
write 2
write 34
write 59
write 2
$
```

11. [5 points]: Briefly explain what block 59 contains in the above trace.

12. [5 points]: Briefly explain what block 4 contains in the above trace.

13. [5 points]: Consider the first 3 writes in the trace. Could the correctness of the xv6 file system be violated if the disk driver sent the writes to the disk hardware in the following order? (Briefly explain your answer)

```
write 4
write 2
write 3
```

End of Quiz I