*Department of Electrical Engineering and Computer Science*

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

## 6.828 Fall 2017

# 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/14) | II (xx/12) | III (xx/7) | IV (xx/7) | V (xx/14) | VI (xx/24) | VII (xx/21) | VIII (xx/1) | (xx/100) |
|---|---|---|---|---|---|---|---|---|
| | | | | | | | | |

**Name:**

# I Stack frame layout

Ben Bitdiddle is debugging his JOS implementation and observes the following state after pausing execution.

The bottom of the stack:

```
0xf011efe4:     ...
0xf011efe0:     0x00001234
0xf011efdc:     0xf010012b
0xf011efd8:     0xf011eff8
0xf011efd4:     0xf0100064
0xf011efd0:     0xf011efd8
0xf011efcc:     0x00043110
0xf011efc8:     0xf0100059
0xf011efc4:     0xf011efd0     <-- ebp, esp
```

**1. [7 points]:** Which of the values in the stack above are return addresses?

With execution still paused at the same point, Ben notices that the `eip` register contains the address of an instruction near the beginning of the following C function:

```
void hexprint(int v)
{
        cprintf("%x\n", v);
}
```

**2. [7 points]:** If Ben were to continue execution until `hexprint` returned, what would the `cprintf` call print?

## II UNIX system call API

Here's a program that uses the UNIX system call API, as described in Chapter 0 of the xv6 book:

```
main() {
  char *message = "aaa\n";
  int pid = fork();
  if(pid != 0){
    // parent process
    message = "bbb\n";
    close(1);
  }
  write(1, message, 4);
  exit(0);
}
```

Assume that fork() succeeds, that file descriptor 1 is connected to the terminal when the program starts, and that write() does nothing when called on a file descriptor that isn't valid.

**3. [5 points]:** Which of the following best describes the output(s) that this program can produce? Circle just one.

- It always prints just aaa.
- It always prints just bbb.
- It prints one or the other of aaa and bbb (but not both).
- It prints aaa, or bbb, or nothing (but not both aaa and bbb).
- It prints both aaa and bbb, in one order or the other.
- It prints nothing at all.

Looking at the xv6 source, Ben Bitdiddle notices that there can be at most one process or kernel thread executing at a time on each core. He thinks that xv6's kernel thread stack per process is wasteful, since only a few of those stacks will be actively used by cores at any give time. Ben suggests modifying xv6 to get rid of the separate kernel thread stack per process, and instead have just a single stack per core, a "core stack." Ben intends that system calls and interrupts on core $i$ execute on core $i$'s core stack.

**4. [7 points]:** Explain to Ben why a separate kernel stack per process is very convenient for xv6's system call implementations.

# III xv6 traps

Recall that the C function calling convention on the x86 divides the eight general-purpose registers into caller-saved and callee-saved. The machine code for a C function must preserve the content of callee-saved registers, either by not modifying them at all, or by saving them on entry and restoring them on exit (typically on the stack). If C code is calling a function, and the machine code for the caller will need the content of a register to still be there after the function returns, and the register is caller-saved, the caller must save and restore the register (again, typically by pushing onto the stack before the call). EAX, ECX, and EDX are caller-saved; EBX, EBP, ESI, and EDI are callee-saved.

Ben is thinking about xv6's struct trapframe and alltraps/trapret in trapasm.S. He observes that user-space C code makes system calls by calling C library functions (generated by usys.S), and that these C calls save caller-saved registers if needed. Ben claims that this means it would be OK to modify alltraps, trapret, and struct trapframe to not save the caller-saved registers. Ben starts by modifying alltraps, trapret, and struct trapframe to get rid of ECX, which is caller-saved: he replaces the pushal/popal in trapasm.S with separate pushes/pops of the other seven registers, and he deletes the `uint ecx` line from struct trapframe in x86.h. Ben notes that system calls seem to work fine after this modification.

Ben's modification means that only user programs that obey the C calling convention are supported, but you should assume that is OK.

**5. [7 points]:** Explain to Ben why, even though his modification works for system calls, it may cause programs to compute incorrectly if there are device interrupts.

# IV Sleep and wakeup

You would like to add xv6 system calls that give lock-like functionality to user-level programs. You start simple with a pair of system calls `uacquire()` and `urelease()`, that implement a single global lock that all processes can use. These system calls take no arguments. `uacquire()` should wait until it acquires the lock, and then return. `urelease()` should release the lock. Multiple processes might be waiting for the lock at any given time.

**6. [7 points]:** Here are incomplete versions of xv6 kernel code implementing the two system calls. Please add code to complete them. Use `sleep()` to wait for the lock, in order that processes not waste too much CPU time while waiting.

```
int taken;           // does a process hold the lock?
struct spinlock mu; // protect taken (mu for mutual exclusion)

int
sys_uacquire(void)
{
  acquire(&mu);
  while(taken != 0){
    sleep(&taken, &mu);
  }
  taken = 1;
  release(&mu);
  return 0;
}

int
sys_urelease(void)
{
  acquire(&mu);
  taken = 0;
  wakeup(&taken);
  release(&mu);
  return 0;
}
```

**Name:**

# V JOS Lab 2

Ben Bitdiddle is looking at this code from his Lab 2 `kern/pmap.c`:

```
void
mem_init(void)
{
  ...
  // create initial page directory.
  kern_pgdir = (pde_t *) boot_alloc(PGSIZE);
  memset(kern_pgdir, 0, PGSIZE);

  kern_pgdir[PDX(UVPT)] = PADDR(kern_pgdir) | PTE_U | PTE_P;

  ...
}
```

UVPT is 0xef400000.

**7. [7 points]:** Ben claims that the UVPT mapping allows any user program to discover the physical address of the page directory. Is Ben right? Why or why not?

Ben recalls from lecture that the MMU checks the permission bits in both the page directory and the page table, so it is safe to leave permissions in the page directory more permissive. He decides to change the code to the following:

```
void
mem_init(void)
{
  ...
  // create initial page directory.
  kern_pgdir = (pde_t *) boot_alloc(PGSIZE);
  memset(kern_pgdir, 0, PGSIZE);

  kern_pgdir[PDX(UVPT)] = PADDR(kern_pgdir) | PTE_W | PTE_U | PTE_P;

  ...
}
```

**8. [7 points]:** What is wrong with Ben's change?

# VI Locking

Recall in homework 6 that some keys went missing after being inserted into the hash table. The problem occurred because of a race condition between parallel threads executing the `put()` method. The solution was to add synchronization to `put()` using `pthread_mutex_lock()`, resulting in this code:

```
static void
insert(int key, int value, struct entry **p, struct entry *n)
{
  struct entry *e = malloc(sizeof(struct entry));
  e->key = key;
  e->value = value;
  e->next = n;
  *p = e;
}

static
void put(int key, int value)
{
  int i = key % NBUCKET;
  pthread_mutex_lock(&lock);
  insert(key, value, &table[i], table[i]);
  pthread_mutex_unlock(&lock);
}

static struct entry*
get(int key)
{
  struct entry *e = 0;
  for (e = table[key % NBUCKET]; e != 0; e = e->next) {
    if (e->key == key) break;
  }
  return e;
}
```

In homework 6, all keys were inserted at the start of execution, before any calls to `get()`. Ben modifies the homework to instead call `get()` and `put()` in parallel, from different threads. While testing the hash table on a multicore mobile phone with a different memory model from x86, he notices that once in a while the `struct entry` returned by `get()` contains the wrong `value`.

**9. [7 points]:** Explain why parallel execution of `get()` and `put()` is failing.

**10. [5 points]:** Indicate where in the above code calls to `pthread_mutex_lock()` and `pthread_mutex_unlock()` should be added to fix the problem.

After fixing the code, Ben notices a drop in performance. Alyssa diagnoses the problem as lock contention and observes that Ben's code calls `get()` much more often than `put()`. She suggests using a different kind of synchronization primitive called a read-write lock. A read-write lock is a lock that allows multiple readers to execute the critical section in parallel. However, it forces writers to run the critical section in serial with respect to both other readers and writers.

Ben looks up the man pages, and finds the following functions are available to support read-write locks:

- **pthread_rwlock_rdlock()**: lock a read-write lock object for reading.
- **pthread_rwlock_wrlock()**: lock a read-write lock object for writing.
- **pthread_rwlock_unlock()**: unlock a read-write lock object.

**11. [5 points]:** Ben removes all calls to `pthread_mutex_lock()` and `pthread_mutex_unlock()`. Indicate where in the above code calls to `pthread_rwlock_rdlock()`, `pthread_rwlock_wrlock()`, and `pthread_rwlock_unlock()` should be added for it to be both correct and more efficient.

Unfortunately, the pthread library on Ben's mobile phone platform does not include support for read-write locks. Read-write locks can be implemented using atomic instructions. The compiler provides, among others, the following portable functions that generate the appropriate atomic instructions on each platform.

- **int __sync_bool_compare_and_swap(int *ptr, int oldval, int newval)**: Atomically compares the value of ptr with oldval and replaces oldval with newval if they are equal. Returns true if the comparison was successful.
- **int __sync_fetch_and_add(int *ptr, int val)**: Returns the current value of ptr and atomically replaces it with ptr + val.

Ben's plan is to represent the state of the read-write lock using an int, with $x = 0$ indicating unlocked, $x < 0$ indicating write-locked and $x > 0$ indicating read-locked.

Ben's code so far is below. The `volatile ... ptr` declarations tell the compiler to not cache the data that `ptr` points to in a register, and instead execute a memory load instruction every time `ptr` needs to be dereferenced.

```
typedef rwlock_t int;

void pthread_rwlock_rdlock(volatile rwlock_t *ptr)
{
  int x;
  while (1) {
    x = *ptr;
    if (x < 0) // is writer holding lock?
      continue;
    if(__sync_bool_compare_and_swap(ptr, x, x + 1))
      break;
  }
}

void pthread_rwlock_unlock(volatile rwlock_t *ptr)
{
  int x = *ptr;
  __sync_fetch_and_add(ptr, x < 0 ? 1 : -1);
}
```

**12. [7 points]:** Fill in the missing code for `pthread_rwlock_wrlock()` below.

```
void pthread_rwlock_wrlock(volatile rwlock_t *ptr)
{
  while (1) {
    if (__sync_bool_compare_and_swap(ptr, 0, -1))
      break;
  }
}
```

# VII Using Virtual Memory

Normally, xv6 allocates heap pages at the time the `brk` system call is executed. However, some memory mappings in the heap are never touched. Recall in homework 4, you were tasked with modifying xv6 to support lazy page allocation. Consider the following proposed solution to the homework. Page allocation and insertion into the page table is removed from `sys_brk`, and instead the following code is added to the `trap()` handler.

```
if(tf->trapno == T_PGFLT){
  uint va = PGROUNDDOWN(rcr2());
  char *mem = kalloc();
  if(mem == 0){
    exit();
    return;
  }
  memset(mem, 0, PGSIZE);
  mappages(proc->pgdir, (char*)va, PGSIZE, v2p(mem), PTE_W|PTE_U);
  return;
}
```

For your reference, xv6 makes use of the following PTE flags values:

```
#define PTE_P           0x001   // Present
#define PTE_W           0x002   // Writeable
#define PTE_U           0x004   // User
```

**13. [7 points]:** Ben traces the changes to the trap handler with gdb and finds during one successful invocation of `trap()` that `kalloc()` returns `0x81000000`. Moreover, `v2p(mem)` returns `0x00110000`. Recall that a PTE consists of a page number (upper 20 bits) and a set of flag bits (lower 12 bits). Using one or more of the flag bits above, what is the precise value of the PTE inserted by `mappages()`?

Alice performs a security audit on Ben's code and discovers some issues. As just one example, the following simple test program causes the xv6 kernel to panic inside `mappages()`.

```
int main(void)
{
  volatile int *bar = KERNBASE;
  *bar = 0xDEADBEEF; //crash happens here
  return 0;
}
```

**14. [7 points]:** Explain why Ben's `trap()` changes are insecure. How would you fix them?

Another possible heap optimization might be to remove a page mapping and free the underlying physical page after the user-level memory allocator is done using it. Ben creates a new system call called `dont_need(void *va)` to facilitate this interaction. `dont_need` reclaims the heap page located at va. If va is touched afterward, a new page will be demand faulted at the location. Ben does nothing to invalidate or flush TLB entries in his system call handler.

**15. [7 points]:** While testing his kernel changes, Ben observes that occasionally memory is corrupted in seemingly random locations. Explain why the code is failing.

## VIII 6.828

**16. [1 points]:** What's the most important thing we could fix about 6.828 to make it better?

# End of Quiz I