Department of Electrical Engineering and Computer Science

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

6.828 Fall 2018

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/15)	II (xx/10)	III (xx/5)	IV (xx/5)	V (xx/5)	VI (xx/10)	VII (xx/15)	VIII (xx/1)	(xx/66)

Name:

I UNIX system call API

Consider the program below. The goal is to use the UNIX system call API (as described in Chapter 0 of the xv6 book) so that multiple processes each write a single, unique character to the write-end of a pipe. A single process reads their characters from the pipe and prints them.

```
// The process reading from the pipe
void reader(int fds[]) {
   int n;
   char buf[1];

   close(fds[1]);
   while((n = read(fds[0], buf, 1)) > 0){
     printf(1, "%c", buf[0]);
   }
}

void writer(int fds[], char c) {
   write(fds[1], &c, 1);
}

// Make n writer processes, each calling writer()
void makewriters(int n, int fds[]) {
  char c = 'a';
  int pid;

  close(fds[0]);
  for (int i = 0; i < n; i++) {
    // YOUR CODE HERE
    c += 1;
  }
  for (int i = 0; i < n; i++)
    wait();
  exit();
}

int
main() {
  int fds[2], pid;

  pid = fork();
  if(pid != 0){
    // parent process
    reader(fds);
  } else {
    // child process
    makewriters(4, fds);
  }
  wait();
  exit();
}
```

Name:

1. [5 points]: Mark with "*" where in the code you would insert the following snippet to make a shared pipe.

```
if(pipe(fds) != 0){
  printf(1, "pipe()␣failed\n");
  exit();
}
```

2. [5 points]: Write down the missing code snippet in `makewriters()`.

Assume that your `makewriters` is correct, that `fork()` succeeds, that file descriptor 1 is connected to the terminal when the program starts, etc.

3. [5 points]: Which of the following output(s) could the `reader` print? (Circle *all* that apply)

- abcd
- cbad
- aabd
- abc

II Sequence coordination and scheduling

Ben replaces the `while` statement in `pipewrite` in xv6 with an `if` statement, as marked below:

```
int
pipewrite(struct pipe *p, char *addr, int n)
{
  int i;

  acquire(&p->lock);
  for(i = 0; i < n; i++){
    if(p->nwrite == p->nread + PIPESIZE){       // THIS LINE IS MODIFIED
      if(p->readopen == 0 || myproc()->killed){
        release(&p->lock);
        return -1;
      }
      wakeup(&p->nread);
      sleep(&p->nwrite, &p->lock);
    }
    p->data[p->nwrite++ % PIPESIZE] = addr[i];
  }
  wakeup(&p->nread);  //DOC: pipewrite-wakeup1
  release(&p->lock);
  return n;
}
```

4. [5 points]: For the program with multiple writers in the previous question could this change break that program? If yes, describe an incorrect behavior that could be observed. If not, explain why.

5. [5 points]: The xv6 scheduler switches from a process stack to a scheduler stack before running `schedule()` to select the next process. Explain why it can't switch to from one process stack to the next process stack directly, using the current process stack to run `schedule()`.

Name:

III JOS Traps and exceptions

Recall that in Lab 3, in `trap()`, JOS checks that interrupts are disabled:

```
  assert(!(read_eflags() & FL_IF));
```

In addition, a comment warns: "if this assertion fails, DO NOT be tempted to fix it by inserting a `cli` in the interrupt path".

6. [5 points]: What could go wrong if Ben, instead of specifying istrap=0 when setting up the trap gate (via the `SETGATE` macro), Ben specified istrap=1 (setting up an exception gate rather than an interrupt gate) and added a `cli` as the first instruction in `_alltraps`? (Explain your answer briefly.)

IV Exofork

You just completed the user-level implementation of fork() using `sys_exofork()` for Lab 4. Your friend who is also taking 6.828 is advertising her idea for even faster COW page allocation; *only duplicate the writable pages during COW*. This is her implementation:

```
envid_t
fork(void)
{
    envid_t envid;
    set_pgfault_handler(pgfault);

    // Create a child.
    envid = sys_exofork();
    if (envid < 0)
        return envid;
    if (envid == 0) {
        thisenv = &envs[ENVX(sys_getenvid())];
        return 0;
    }

    // My COW opt starts here!
    int pn, end_pn, r;

    // Copy the address space.
    for (pn = 0; pn < PGNUM(UTOP); ) {
        if (!(uvpd[pn >> 10] & PTE_P)) {
            pn += NPTENTRIES;
            continue;
        }
        for (end_pn = pn + NPTENTRIES; pn < end_pn; pn++) {
            if ((uvpt[pn] & (PTE_P|PTE_U|PTE_W)) != (PTE_P|PTE_U|PTE_W))
                continue;
            if (pn == PGNUM(UXSTACKTOP - 1))
                continue;
            duppage(envid, pn);
        }
    }
```

7. [5 points]: Briefly explain what problems you see with her solution and how you would address them.

Name:

V xv6 file system

Alyssa has modified `bmap` to support a double-indirect block, as asked in the homework to support big files. Alyssa notices, however, that when she removes the big file ``big.file'' created by `big` not all blocks are freed.

8. [5 points]: Complete the code below that must be added to `itrunc` to handle the double-indirect block of a big file such as ``big.file'' so that all blocks of the file are freed.

```
if(ip->addrs[NDIRECT+1]) {      // is there a double-indirect block?
  bp = bread(ip->dev, ip->addrs[NDIRECT+1]);  // read double-indirect block
  a = (uint*)bp->data;
  for(j = 0; j < NINDIRECT; j++){
    if(a[j]) {
      // YOUR CODE HERE






    }
  }
  brelse(bp);
  bfree(ip->dev, ip->addrs[NDIRECT+1]);
  ip->addrs[NDIRECT+1] = 0;
}
```

Name:

VI Synchronization

A downside of xv6's spinlocks are that they disable interrupts during a critical section. A long-running critical section will delay interrupts for the complete duration of that critical section. Alyssa mentions to Ben that xv6 could support a variant of spinlocks that do not disable interrupts. She suggests adding a new counter, called `nlock`, to `struct cpu` (the struct returned by `mycpu()`). Her plan is to increment the counter each time a lock is acquired and decrement the counter each time a lock is released. Alyssa then uses the counter to prevent rescheduling inside locking critical sections, modifying `trap()` as follows:

```
void
trap(struct trapframe *tf)
{
  // SAME CODE AS BEFORE

  // Force process to give up CPU on clock tick.
  // Alyssa: Added a new check to ensure nlock is zero.
  if(myproc() && myproc()->state == RUNNING &&
     tf->trapno == T_IRQ0+IRQ_TIMER && mycpu()->nlock == 0)
    yield();

  // SAME CODE AS BEFORE
}
```

Meanwhile, Ben is working on implementing Alyssa's new variant of spinlocks that don't disable interrupts:

```
// Acquire the lock. This version allows interrupts to be left enabled inside
// the critical section.
void
acquireirqon(struct spinlock *lk)
{
  // YOUR CODE HERE







  // The xchg is atomic.
  while(xchg(&lk->locked, 1) != 0)
    ;

  // Tell the C compiler and the processor to not move loads or stores
  // past this point, to ensure that the critical section's memory
  // references happen after the lock is acquired.
  __sync_synchronize();
}

// Releases a lock acquired with acquireirqon().
void
releaseirqon(struct spinlock *lk)
{
  // Tell the C compiler and the processor to not move loads or stores
  // past this point, to ensure that all the stores in the critical
  // section are visible to other cores before the lock is released.
  // Both the C compiler and the hardware may re-order loads and
  // stores; __sync_synchronize() tells them both not to.
  __sync_synchronize();

  // Release the lock, equivalent to lk->locked = 0.
  // This code can't use a C assignment, since it might
  // not be atomic. A real OS would use C atomics here.
  asm volatile("movl␣$0,␣%0" : "+m" (lk->locked) : );

  // YOUR CODE HERE






}
```

9. [5 points]: Fill in the two missing pieces of code on this page. For simplicity, ignore any potential lock debugging features.

Now that Ben has finished his implementation of the new spinlocks, he wants to determine which existing calls to `acquire()` and `release()` can be replaced with `acquireirqon()` and `releaseirqon()`.

10. [5 points]: Circle the best guideline for when it is correct to use this new spinlock API.

A. The new API can be used anywhere.

B. The new API can only be used when xv6 is running on a uniprocessor (one core).

C. The new API can be used on any lock that isn't acquired inside an interrupt handler.

D. The new API can only be used when locks are acquired inside interrupt handlers.

Justify your answer.

Name:

VII Virtual Memory

11. [5 points]: Ben Bitdiddle decides that he doesn't like the 4 KB page size used in JOS, so he designs an alternative page table structure with 1 KB pages. He changes `PGSIZE`, `pgdir_walk`, and the rest of his kernel to use the new page size. Unfortunately, when he attempts to run his OS in QEMU he gets a triple fault. What is going wrong, and is there anything that Ben can do to fix this?

12. [5 points]: Suppose Ben wants to modify xv6 so that a user process can address more than 2GB of virtual memory. He notices that KERNBASE is mapped at 2GB and he increases it. However, he then notices that the macros `v2p()` and `p2v()` don'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 allowing the kernel to to address all of physical memory?

13. [5 points]: Ben is working on a new security feature for xv6 that allows existing parts of the user heap (the region below `myproc()->sz`) to be marked read-only. After changing the permission flags of one mapping in the page table from `PTE_P|PTE_W|PTE_U` to `PTE_P|PTE_U`, Ben realizes that the kernel sometimes but does not always receive a page fault when the user writes to the virtual address marked read-only. Explain why the CPU fails to detect the changed mapping. How could you fix it?

Name:

VIII 6.828

14. [1 points]: What's the most important thing we could fix about 6.828 to make it better?

End of Quiz I