Department of Electrical Engineering and Computer Science

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

6.828 Fall 2012

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/11)	II (xx/20)	III (xx/16)	IV (xx/24)	V (xx/8)	VI (xx/16)	VII (xx/5)	Total (xx/100)

Name:

I Shell

The xv6 shell creates new processes to run a command as follows:

```
if(fork1() == 0)
  runcmd(parsecmd(buf));
wait();
```

1. [5 points]: It is possible that the child process that runs the command exits before the parent calls `wait`. Give a scenario in which that can happen.

2. [6 points]: How does xv6 ensure that `wait` returns correctly even if the child already has exited before the parent called `wait`?

II Locks

In the in-class exercises on locking you modified a hash table to work correctly under concurrent lookup and insert by adding `acquire` and `release` statements for a global lock. The code we handed out for `put` is as follows:

```
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)
{
  struct entry *n, **p;
  for (p = &table[key%NBUCKET], n = table[key % NBUCKET];
                       n != 0; p = &n->next, n = n->next) {
    if (n->key > key) {
      insert(key, value, p, n);
      goto done;
    }
  }
  insert(key, value, p, n);
 done:
  return;
}
```

3. [4 points]: In your solutions for the in-class exercise, you added in the above code `acquire` and `release` calls to ensure correct operation of concurrent `put` invocations. Indicate where in the above code they should be added.

To avoid the overhead of locks, Alyssa suggests use of atomic operations, which are guaranteed to perform their multiple step operation as a single atomic operation. In particular, Alyssa suggests use of the following atomic operation:

```
bool __sync_bool_compare_and_swap (type *ptr, type oldval, type newval)
```

This builtin function performs an atomic compare and swap. If the current value of `*ptr` is `oldval`, it writes `newval` into `*ptr`, and returns `true`. Otherwise it returns `false`. The compiler turns the builtin function into the atomic instruction sequence that's appropriate for the target architecture; on the x86, it might use `lock cmpxchg`.

An example:

```
char *p;
  ...
  if(__sync_bool_compare_and_swap(&p, 0, 1))
    ....  // success: p was set to 1
  else
    ...   // failure: p was not set to 1, because p wasn't equal to 0
```

This code will set `p` to `1` only if `p` previously held `0`. If two cores execute the code at the same time, and `p` held `0`, only one of the two cores will see a `true` return value from the call.

4. [8 points]: Complete the `put` below using `__sync_bool_compare_and_swap` so that concurrent invocations will run correctly without using locks:

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




  for (p = &table[key%NBUCKET], n = table[key % NBUCKET]; n != 0;
                                        p = &n->next, n = n->next) {



    if (n->key > key) {









    }
  }




 done:
  return;
}
```

5. [8 points]: For the lock-free implementation, give an example of a workload for which `put`'s throughput scales linearly with an increasing number of cores.

III Logical Logging

Alyssa is unhappy that xv6 logging writes an entire sector to the log even if the file system only needs to modify a few bytes in the sector. She observes that log records would be much shorter if they described the operation to be performed, rather than containing the entire modified sector. Alyssa modifies the xv6 logging system to use this idea, called "logical logging." For example, when her `balloc()` allocates a data block, it logs the setting of the bit in the block allocation bitmap with a "set bit" record. A "set bit" record contains a constant indicating the record type, and the block number that was allocated. Alyssa's recovery code for this type of log record sets the relevant bit in the block allocation bitmap on disk. The "set bit" record takes up only a few bytes in the log, rather than an entire sector.

While the "set bit" record works, it turns out that correct recovery from logical log records is not possible for some operations.

6. [8 points]: Alyssa defines a "create file" log record containing the parent directory i-number and the new file's name. Recovery for this kind of record calls a version of create() (sysfile.c) that does everything but the initial nameiparent(), and that uses bwrite for all modifications. Explain why use of this "create file" log record will incorrectly handle some failure scenarios that the current xv6 handles correctly.

7. [8 points]: Consider a "delete directory entry" log record that contains the parent i-number and the entry's name. Recovery finds the named entry in the parent directory and overwrites it with nulls. Explain if this works; if not, briefly explain why not.

IV Page Tables

Consider the following 32-bit x86 page table setup.

`%cr3` holds 0x00001000.

The Page Directory Page at physical address 0x00001000:

```
PDE 0: PPN=0x00002, PTE_P, PTE_U, PTE_W
PDE 1: PPN=0x00003, PTE_P, PTE_U, PTE_W
PDE 2: PPN=0x00002, PTE_P, PTE_U, PTE_W
... all other PDEs are zero
```

The Page Table Page at physical address 0x00002000 (which is PPN 0x00002):

```
PTE 0: PPN=0x00005, PTE_P, PTE_U, PTE_W
PTE 1: PPN=0x00006, PTE_P, PTE_U, PTE_W
... all other PTEs are zero
```

The Page Table Page at physical address 0x00003000:

```
PTE 0: PPN=0x00005, PTE_P, PTE_U, PTE_W
PTE 1: PPN=0x00005, PTE_P, PTE_U, PTE_W
... all other PTEs are zero
```

8. [8 points]: List all virtual addresses that map to physical address 0x00005555.

9. [8 points]: Given the above page table, what virtual address (if any) should the kernel store to in order to modify the mapping for virtual address 0x402000 (that's 4*1024*1024 + 8192)?

10. [8 points]: Explain exactly what the above page table should contain in order to to allow the kernel to write to the page directory page. There is more than one answer; you only have to explain one.

.

V JOS memory mapping and fork

Here is a code snippet from Ben Bitdiddle's lab 4. Ben is struggling to get user-mode `fork` to work. In his `fork`, Ben invokes `duppage` for every virtual page number (`vpn`) below `UTOP` that is present in the current environment's address space, except for the exception stack.

```
static int duppage(envid_t envid, unsigned vpn) {
    void *addr = (void *)(vpn << PGSHIFT);
    pte_t pte = uvpt[vpn];
    int r;
    // if the page is just read-only, just map it in.
    if (!(pte & (PTE_W|PTE_COW))) {
        if ((r = sys_page_map(0, addr, envid, addr, pte & PTE_SYSCALL)) < 0)
            panic("sys_page_map␣in␣child␣failed:␣%e", r);
        return 0;
    }
    if ((r = sys_page_map(0, addr, 0, addr, PTE_P|PTE_U|PTE_COW)) < 0)
        panic("sys_page_map␣in␣parent␣failed:␣%e", r);
    if ((r = sys_page_map(0, addr, envid, addr, PTE_P|PTE_U|PTE_COW)) < 0)
        panic("sys_page_map␣in␣child␣failed:␣%e", r);
    return r;
}
```

11. [8 points]: Ben Bitdiddle could not pass the `forktree` test because the above code has a bug. While debugging, he printed out the value of `uvpt[vpn]` at the end of the function, and found that the parent environment's user stack was not marked copy-on-write. Explain why this is happening, and describe how to fix the bug. You should assume that his other code is free of bugs.

VI Exokernel `exec`

Ben Bitdiddle has chosen to implement UNIX-style `exec` as his challenge exercise for lab 5. `exec` replaces the current environment's memory with the desired executable, sets up a stack, and jumps to the entry point of the executable.

Alyssa P. Hacker proposes the following design for `exec` in the user-space library operating system:

1. Unmap all pages in the current environment except for the stack and the code for `exec`.
2. Map in the pages (loaded from the ELF file on disk) for the new environment.
3. Unmap the code page for `exec`.
4. Jump to the new environment's entry point.

12. [8 points]: Ben Bitdiddle sees a problem with Alyssa's first and second steps. What might go wrong? Assume the code for `exec` fits in a single page and doesn't make any function calls (other than system calls).

13. [8 points]: There are a few other problems with Alyssa's plan. Please describe a new plan for a working `exec` in user space. Your solution shouldn't modify the kernel.

VII 6.828

We'd like to hear your opinions about 6.828, so please answer the following questions. (Any answer, except no answer, will receive full credit.)

14. [2 points]: This year we posted an improved draft of the xv6 commentary at the beginning of the semester. Did you find the chapters useful? What should we do to improve them?

15. [2 points]: This year we continued code reviews and added rebuttals. Did you find them useful? What should we do to improve them?

16. [1 points]: Did you find the in-class exercises useful? How would you suggest improving them?

End of Quiz

Name: