Department of Electrical Engineering and Computer Science

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

6.828 Fall 2016

Quiz II

All problems are open-ended questions. In order to receive credit you must answer the question as precisely as possible. You have 120 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/10)	II (xx/10)	III (xx/5)	IV (xx/5)	V (xx/10)	VI (xx/10)	VII (xx/15)	VIII (xx/15)	IX (xx/5)	Total (xx/85)

Name:

I Logging

```
1   inMemoryLog = {}
2
3   def log_flush():
4       i = 0
5       for (a, v) in inMemoryLog.iteritems():
6           if i >= LOGSIZE: return None
7           disk_write(LOGSTART + i, v)
8           logHeader.sector[i] = a
9           i = i + 1
10      logHeader.len = len(inMemoryLog)
11      return logHeader
12
13  def log_apply():
14      for (a, v) in inMemoryLog.iteritems():
15          disk_write(a, v)
16
17  def log_commit():
18      logHeader = log_flush()
19      if logHeader is None: return False
20      disk_sync()
21      disk_write(COMMITBLOCK, logHeader)
22      disk_sync()
23      log_apply()
24      disk_sync()
25      logHeader.len = 0
26      disk_write(COMMITBLOCK, logHeader)
27      disk_sync()
28      inMemoryLog = {}
29      return True
30
31  def log_recover():
32      logHeader = disk_read(COMMITBLOCK)
33      for i in range(0, logHeader.len):
34          v = disk_read(LOGSTART + i)
35          disk_write(logHeader.sector[i], v)
36      disk_sync()
37      logHeader.len = 0
38      disk_write(COMMITBLOCK, logHeader)
39      disk_sync()
```

The pseudo-code above replicates the key functions of the logging system from Figure 14 in the paper *Using Crash Hoare Logic for Certifying the FSCQ File System*. The logging system guarantees that if a transaction successfully commits, then all the blocks modified by the transaction will be installed or none.

The intended use is that every system call is wrapped in a transaction, starting with a call to log_begin and ending with a call to log_commit. disk_write writes a block to the disk's non-persistent buffer. disk_sync tells the disk to flush its internal buffer to the persistent storage medium; when disk_sync returns, the disk has finished writing its internal buffer to persistent storage.

Name:

1. [5 points]: The Linux ext3 file system as described in the paper *Journaling the Linux ext2fs Filesystem* also wraps system calls inside a transaction, but provides different persistence guarantees. Briefly explain the difference between the persistence guarantees of the pseudo-code above and ext3. You should assume ext3 journals both file contents and meta-data.

Ben Bitdiddle observes that `disk_sync` takes a long time, so he deletes the `disk_sync` at line 27. He reasons that erasing the commit record on disk isn't really necessary after the transaction has already completed.

2. [5 points]: Give an example scenario that demonstrates that Ben's change will break the correctness of the logging system. (Briefly explain your answer)

Name:

II Scalable locks

After reading the paper *Non-scalable locks are dangerous*, Ben is excited by scalable locks. But studying the lecture notes and 6.828 quizzes from previous years, he is intimidated by the complexity of MCS locks. He considers a simpler scalable lock design:

```
1   struct lock {
2     struct alock {
3       int x;
4     } __attribute__ ((aligned (CACHELINE)));
5     struct alock has_lock[NCORE];
6     unsigned int next_slot;
7   }
8
9   void init(lock *l) {
10    for (int i = 0; i < NCORE; i++)
11      l->has_lock[i].x = 0;
12    l->has_lock[0].x = 1;
13    next_slot = 0;
14  }
15
16  int acquire(lock *l)
17  {
18    int me = fetch_and_add(&l->next_slot, 1);
19    me = me % NCORE;
20    while(l->has_lock[me].x == 0)
21      ;
22    l->has_lock[me].x = 0;
23    return me;
24  }
25
26  void release(lock *l, int me)
27  {
28    l->has_lock[(me + 1) % NCORE].x = 1;
29  }
```

Ben's implementation assumes there is one thread per core. Each lock has as many `alock` structures as there are cores. Each `alock` is on a separate cache line. `fetch_and_add` is an atomic instruction that atomically increments `next_slot` and returns the previous value.

A core is supposed to use the lock as follows:

```
1     struct lock l;
2     int me;
3
4     init(&l);  // call once for a lock
5
6     me = acquire(&l);
7     // critical section
8     release(&l, me);
```

Name:

You should assume that the compiler and the processor do not eliminate or re-order memory references. You can also assume that `next_slot` is on a separate cache line.

3. [5 points]:   Ben runs a sequence of experiments with increasing numbers of cores that all continuously acquire and release the same lock. Ben observes that his lock doesn't experience a performance collapse. Instead, the lock is scalable: the number of acquires (and releases) completed per second stays roughly constant regardless of the number of cores. Explain why, focusing on cache line transfers.

4. [5 points]:   What is a downside of Ben's lock compared to a single spinlock? Explain your answer.

Name:

III   Refcache

Figure 2 in the *RadixVM: Scalable address spaces for multithreaded applications* paper describes the ref-cache algorithm to implement scalable reference counters. Ben Bittdiddle finds it difficult to understand this code and implements a different plan. The following code gives his scalable implementation for a distributed reference counter for a single object.

```
1    struct obj *p;
2    int counter[NCORE];
3
4    void allocref() {
5      p = malloc(sizeof(*p));
6      counter[mycore()] = 1;
7      return p;
8    }
9
10   struct obj *getref() {
11     counter[mycore()]++;
12     return p;
13   }
14
15   void releaseref() {
16     counter[mycore()]--;
17     if (counter[mycore()] == 0 && isZero()) {
18       free(p);
19     }
20   }
21
22   bool isZero() {
23     int sum = 0;
24
25     for (int i = 0; i < NCORE; i++) {
26       sum += counter[i];
27     }
28     return sum == 0;
29   }
```

You can assume that there is exactly one thread per core. Each core maintains a local counter for the object referenced by p in the array counter. A single core calls allocref to initialize the counter. Before another core uses p, it calls getref, increasing its local counter. If a core is done with the object, it calls releaseref, decreasing its local counter. If the local counter is 0, releaseref checks if the object can be freed using isZero, which sums up all the per-core counters and checks if the sum is 0.

Name: 6

5. [5 points]: Can Ben's reference counter code cause the object to be freed even though there are still references to it? Explain your answer.

IV   Dune

Suppose an application running as a Dune process has virtual address v_1 mapped to a page of memory allocated from the kernel.

6. [5 points]:   What does the application have to do to create a second virtual address v_2 that refers to the same memory?

V   Singularity

Ben is implementing a file server as a SIP in Singularity. It will talk to client SIPs on the same machine through Singularity channels. Ben would like his file server to maintain a cache of file blocks. Because some files need to be read by many applications, Ben would like each cached block to be useable by multiple clients without having to copy it.

7. [5 points]:   Explain why this is difficult to achieve in Singularity.

Consider the bar marked "Add Ring 3" in Figure 5 of the Singularity paper. This bar reflects the run-time of the benchmark with the client SIP running at CPL=3. The run-time is noticeably longer than if the client SIP runs at CPL=0 (the "Add Separate Domain" bar).

8. [5 points]:   Explain why the system is slower when the client SIP runs with CPL=3 rather than CPL=0.

Name: 9

VI   IX

Consider the paper *IX: A Protected Dataplane Operating System for High Throughput and Low Latency*, by Belay *et al.*

In the experiment in the paper's Section 5.2 (Figure 2), for a message size of 20 kilobytes, the client sends 20 kilobytes, and then the server sends a 20-kilobyte reply. The experiments use 1500-byte packets, so a 20-kilobyte message consists of roughly 14 packets.

9. [5 points]:   How many transitions between IX and the application (ring 0 and ring 3 in Figure 1) does the server make in receiving and replying to one 20-kilobyte message? Briefly explain your answer.

10. [5 points]:   IX requires use of a NIC that supports multiple independendent hardware DMA queues (called Receive Side Scaling (RSS)). Explain how IX benefits from these multiple queues.

VII   Lab 4

Alyssa P. Hacker is working her way through the Copy-on-Write fork implementation in Lab 4. When she gets to the part on setting up user-level pagefault handling, she feels that it is silly to have two separate stacks—a normal stack and a user exception stack—for every environment. So she decides to have a single stack instead. Her implementation uses this single stack for both application code and exception handling. Her fork implementation maps the page of this single stack COW.

11. [5 points]:   Explain what will go wrong with Alyssa's single stack scheme.

Ben Bitdiddle decides to change the success return value of sys_ipc_recv() from zero to 0x6828. So he modifies his kernel to look like this:

```
static int
sys_ipc_recv(void *dstva)
{
        if ((dstva < (void*)UTOP) && ((uint32_t)dstva % PGSIZE != 0))
                return -E_INVAL;
        curenv->env_ipc_recving = 1;
        curenv->env_ipc_dstva = dstva;
        curenv->env_status = ENV_NOT_RUNNABLE;
        sched_yield();
        return 0x6828; // success!
}
```

Name: 11

12. [5 points]: Explain why this change will **not** cause 0x6828 to be returned from a successful call to sys_ipc_recv().

Ben modifies the sys_ipc_try_send() system call implementation to keep trying until the send succeeds, instead of returning an error if the target environment isn't ready. Here's his code:

```
static int
sys_ipc_try_send(envid_t envid, uint32_t value, void *srcva, unsigned perm)
{
  while (do_ipc_try_send(envid, value, srcva, perm) == -E_IPC_NOT_RECV)
    ;

  return 0;
}
```

`do_ipc_try_send` tries to deliver the IPC to the destination environment. When he runs the pingpong program, however, it hangs.

13. [5 points]: Explain why Ben's modification causes the program to hang.

VIII   Lab 5

14. [5 points]:   Ben Bitdiddle is working on exercise 7 of lab 5, and is writing up his implementation of the `sys_env_set_trapframe` syscall. He proudly shows his code to Alyssa. Alyssa promptly points out that his code is wildly insecure since he doesn't do any checks on the values set in the user trapframe. Ben is skeptical that this is actually a problem, and stubbornly challenges Alyssa to find a way to use `sys_env_set_trapframe` to give an environment privileges it wouldn't normally have. Describe how Alyssa might construct a malicious trapframe, and what privileges this trapframe would give to an environment.

Ben is on exercise 3, and is having issues with his implementation of alloc_block. Here is the full code of Ben's implementation:

```c
int
alloc_block(void)
{
  uint32_t i, blockno;
  for (i = 0; i < BLKSIZE; i++) {
    if (bitmap[i] != 0) {
      // assume this correctly identifies the first free block
      blockno = 32*i + (32 - __builtin_clzl(bitmap[i])-1);
      break;
    }
  }
  // assume this correctly sets the bit corresponding to blockno
  bitmap[blockno/32] ^= (1<<(blockno%32));
  return blockno;
}
```

15. [5 points]:   When he runs the tests, he observes that the disk never seems to fill (i.e., alloc_block always returns a block number). Not even for the tests that are designed to write enough to fill the disk. What check is Ben's code missing?

16. [5 points]:   When Ben runs the persistence tests with the code above, he observes that after a reboot, certain file system operations corrupt data in entirely unrelated files. In particular, appends to an existing file, or creating and/or writing to a new file, exhibit this behavior, but modifying the blocks of an existing file does not. What is causing this mysterious issue?

Name: 14

IX 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.)

17. [1 points]: Grade 6.828 on a scale of 0 (worst) to 10 (best)?

18. [2 points]: Any suggestions for how to improve 6.828?

19. [1 points]: What is the best aspect of 6.828?

20. [1 points]: What is the worst aspect of 6.828?

End of Quiz II

Name: