



*Department of Electrical Engineering and Computer Science*

**MASSACHUSETTS INSTITUTE OF TECHNOLOGY**

**6.828 Fall 2011**

## **Quiz II**

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 IS AN OPEN BOOK, OPEN NOTES EXAM.**

*Please do not write in the boxes below.*

<b>I (xx/40)</b>	<b>II (xx/18)</b>	<b>III (xx/18)</b>	<b>IV (xx/16)</b>	<b>V (xx/8)</b>	<b>Total (xx/100)</b>

**Name:**

## I Short paper questions

1. [8 points]: In Singularity (“Singularity: rethinking the software stack” by Hunt and Larus) a system call is run on the same stack as user code. As the authors point out, this has the potential to entangle user and kernel garbage collection. Why can a malicious SIP *not* trick the kernel into examining a user stack frame when the kernel performs garbage collection? (The paper doesn’t give a precise answer; you’ll have to deduce it as best as you can.)

2. [8 points]: The paper on Linux scalability uses a standard trick with generation numbers to avoid a scalability bottleneck in the `dentry` cache (section 4.4). The pseudo code is as follows:

```
lookup(inode, name):
    dentry = hash(inode, name)
    g = dentry.gen
    if g == 0:
        # some core is modifying dentry
        fall back to locking protocol
    # copy relevant fields
    i = dentry.inode
    n = dentry.name
    g1 = dentry.gen
    if g == g1 and inode == i and name == n:
        ok = atomic_inc_if_not_zero(&dentry.refcnt);
        if (!ok) fall back to locking protocol
    else:
        fall back to locking protocol
    return dentry

modify(dentry):
    spin lock dentry
    g = dentry.gen
    dentry.gen = 0 // DELETE THIS LINE
    dentry.inode = ...
    dentry.name = ...
    dentry.gen = g + 1
    unlock dentry
```

(continued on next page)

Name:

Give an example of something that could go wrong if you deleted the line marked “DELETE THIS LINE”.

**3. [8 points]:** The ballooning technique described in “Memory Resource Management in VMware ESX Server” by Waldspurger avoids double paging by monitor and guests. Why does ESX use ballooning instead of a scheme in which the guest kernel explicitly tells the VMM about how it uses physical pages?

MCS locks don't suffer performance collapse like ticket spin locks do, as described in unpublished document assigned for the lecture 16 (paper #243). For reference in the following two questions, here is an implementation of MCS locks in C (which follows the description of MCS locks in the unpublished document):

```

1 // Each core has a qnode for each lock
2 struct qnode {
3     volatile struct qnode *next;
4     volatile char waiting;
5     char __pad[0] __attribute__((aligned(CACHELINE)));
6 };
7
8 typedef struct {
9     struct qnode *tail __attribute__((aligned(CACHELINE)));
10 } mcslock;
11
12 void
13 mcs_lock(mcslock *L, struct qnode *mynode)
14 {
15     mynode->next = NULL;
16     struct qnode *predecessor = fetch_and_store(&L->tail, mynode);
17     if (predecessor) {
18         mynode->waiting = 1;
19         predecessor->next = mynode;
20         while (mynode->waiting)
21             /* spin */;
22     }
23 }
24
25 void
26 mcs_unlock(mcslock *L, struct qnode *mynode)
27 {
28     if (!mynode->next) {
29         if (compare_and_swap(&L->tail, mynode, NULL))
30             return;
31         while (!mynode->next)
32             /* spin */;
33     }
34     mynode->next->waiting = 0;
35 }

```

Each core invokes lock and unlock with its own private qnode.

Name:

**4. [8 points]:** Use the above code to explain why MCS locks don't suffer performance collapse as ticket locks do when several cores are contending for the same lock.

**5. [8 points]:** As the paper reports, a disadvantage of MCS locks is that acquiring and releasing an uncontended MCS lock takes more time than acquiring and releasing an uncontended ticket lock (see Figure 11). Using the above MCS code and the ticket code from the paper (Figure 1), explain why MCS locks are more expensive in the non-contended case.

## II Rethink the Sync

Tom the TA writes the following grading program. The idea is that per-assignment scores are stored in a set of input files, and the program's job is to compute a final course score for each student and write it to file `studentname.final`.

```
main():
  remove all files ending with ".final"
  for each student name s in alphabetical order:
    read assignment scores for student s
    calculate final grade
    filename = s + ".final"
    fd = creat(filename)
    write(fd, final grade, ...)
    close(fd)
    printf("done with %s\n", s)
```

Tom uses a Linux laptop with an EXT3 journaling file system in “journaled data” mode (so that the journal contains file content, in addition to metadata). He runs the following:

```
program | cat
```

Tom sees “done with x” for all students with names up through “m”, and then his laptop crashes. Tom restarts his laptop.

- 6. [6 points]:** Tom thinks he may have to re-run the grading program for some or all students. Explain what guarantees he can count on about which final grades will be on disk after the restart.

The next year, Tom does the same thing, but this time he uses the xsyncfs file system described in the Rethink the Sync paper. Again, Tom's laptop crashes just after he sees the "done with x" for all students with names up through "m", and he restarts the laptop.

**7. [6 points]:** Explain what guarantees Tom can count on about which final grades will be on disk after the restart.

The third year, Tom types these commands on his xsyncfs laptop:

```
program > out  
cat out
```

Again, he sees "done with x" for all students with names up through "m", his laptop crashes, and he restarts it.

**8. [6 points]:** Explain what guarantees Tom can count on about which final grades will be on disk after the restart.

**Name:**

### III KLEE

Consider this C program:

```
int
square(int x)
{
    int table[] = { 0, 1, 4, 9, 0xb00b00, 25, 36 };
    return table[x];
}

void
main(void)
{
    printf("%d\n", square(4));
}
```

You run KLEE (see the paper by Cadar *et al.*) on this program. Assume that the C libraries have no bugs.

**9. [6 points]:** Would KLEE find any bugs in the above program? If yes, describe the full set of test cases that KLEE could generate to illustrate the bug(s).

**Name:**



Now suppose you run KLEE on this program; `square()` is the same but `main()` is different:

```
int
square(int x)
{
    int table[] = { 0, 1, 4, 9, 0xb00b00, 25, 36 };
    return table[x];
}

void
main(void)
{
    int x = 0;
    scanf("%d", &x);
    printf("%d\n", square(x));
}
```

The call to `scanf()` reads an integer from the standard input and places it in `x`.

**10. [6 points]:** Would KLEE find any bugs in the above program? If yes, describe the full set of test cases that KLEE could generate to illustrate the bug(s).

**Name:**

Now this program; again, `square()` is the same but `main()` is different:

```
int
square(int x)
{
    int table[] = { 0, 1, 4, 9, 0xb00b00, 25, 36 };
    return table[x];
}

void
main(void)
{
    int x = 0, y;
    scanf("%d", &x);
    y = square(x);
    assert(y == x*x);
}
```

**11. [6 points]:** Would KLEE find any bugs in the above program? If yes, describe the full set of test cases that KLEE could generate to illustrate the bug(s).

**Name:**

## IV E1000

Ben Bitdiddle has just finished implementing his E1000 receive function, shown below.

```
int
e1000_receive(char *buf)
{
    int tail = read_reg(RDT);

    // Check for and clear "Descriptor Done" bit
    if (!(rx_descriptors[tail].status & E1000_RXD_STAT_DD))
        return 0;
    rx_descriptors[tail].status = 0;

    // Receive the packet
    write_reg(RDT, (tail + 1) % RX_RING_SIZE);
    memmove(buf, rx_buffers[tail], rx_descriptors[tail].length);
    return rx_descriptors[tail].length;
}
```

Ben tests his receive function by sending JOS a hundred packets and, between each packet, checking that it received the right data. Confident in his work, he shows it off to Alyssa P. Hacker. When Alyssa tries sending a hundred packets in a short burst, everything falls apart; Ben's JOS drops several packets, duplicates some, and corrupts others.

**12. [8 points]:** Why does Ben's implementation drop, duplicate, and corrupt packets and how can he fix it?

Name:

Louis Reasoner is having similar problems receiving packets. However, he knows that his E1000 driver is correctly receiving all packets. For Louis, packets are getting dropped or duplicated before they are processed by the network server environment. He narrows the problem to his input environment, which looks like

```
void
input(envid_t ns_envid)
{
    while (1) {
        // Receive a packet
        nsipcbuf.pkt.jp_len = sys_net_rx(nsipcbuf.pkt.jp_data, 2048);
        if (nsipcbuf.pkt.jp_len > 0) {
            // Pass packet to network server environment
            ipc_send(ns_envid, NSREQ_INPUT, &nsipcbuf, PTE_P|PTE_U);
            // Wait for NS environment to process the packet
            for (int i = 0; i < 8; i++)
                sys_yield();
        } else
            sys_yield();
    }
}
```

**13. [4 points]:** Explain the race in Louis' implementation that causes the network server environment to see dropped and duplicated packets.

**14. [4 points]:** Modify Louis' code to fix this race. (You can mark up the code above if you find it easier.)

**Name:**

## V 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.)

15. [2 points]: Grade 6.828 on a scale of 0 (worst) to 10 (best)?

16. [2 points]: Any suggestions for how to improve 6.828?

17. [2 points]: What is the best aspect of 6.828?

18. [2 points]: What is the worst aspect of 6.828?

# End of Quiz

Name:

MIT OpenCourseWare  
<http://ocw.mit.edu>

6.828 Operating System Engineering  
Fall 2012

For information about citing these materials or our Terms of Use, visit: <http://ocw.mit.edu/terms>.