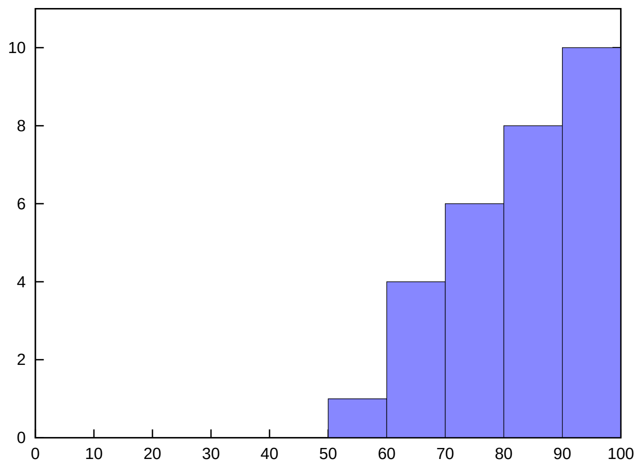
Department of Electrical Engineering and Computer Science

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

6.828 Fall 2011

Quiz II Solutions

Mean 82 Median 85 Std. dev. 10.9



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.)

Answer: Crossing the user/kernel boundary pushes a barrier on the stack, which the garbage collector knows not to pass when scanning the stack. This, combined with the fact that object references cannot be passed through the kernel ABI and the language is typesafe, means that the kernel garbage collector will never traverse into a user stack frame.

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)

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

Answer: If `lookup` were to run while a dentry is being modified, lookup could return a dentry that isn't what the caller asked for. (Even with the indicated line included in the code, this is a problem because `modify` may alter a dentry with live references. This can be fixed in `modify` by checking that refcnt is zero after setting gen to 0. With this modification, if the indicated line is deleted and `lookup` runs between the assignment of `inmode` and `name` in `modify`, `lookup` can return a dentry the caller didn't request.)

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?

Answer: This requires minimal guest modifications and keeps the page eviction policy in the guest memory manager, which has the most accurate information about memory usage.

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):

```c
// Each core has a qnode for each lock
struct qnode {
    volatile struct qnode *next;
    volatile char waiting;
    char __pad[0] __attribute__((aligned(CACHELINE)));
};

typedef struct {
    struct qnode *tail __attribute__((aligned(CACHELINE)));
} mcslock;

void
mcs_lock(mcslock *L, struct qnode *mynode)
{
    mynode->next = NULL;
    struct qnode *predecessor = fetch_and_store(&L->tail, mynode);
    if (predecessor) {
        mynode->waiting = 1;
        predecessor->next = mynode;
        while (mynode->waiting)
            /* spin */;
    }
}

void
mcs_unlock(mcslock *L, struct qnode *mynode)
{
    if (!mynode->next) {
        if (compare_and_swap(&L->tail, mynode, NULL))
            return;
        while (!mynode->next)
            /* spin */;
    }
    mynode->next->waiting = 0;
}
```

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

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.

Answer: Each core waiting on a contended MCS lock spins on a separate cache line (line 21). Hence, when the lock is released, only the cache line belonging to the next waiter needs to be transferred (line 36). In contrast, for a ticket lock, every core spins on the same cache line, so release must transfer $O(n)$ cache lines, where n in the number of waiting cores.

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.

Answer: An uncontended MCS lock requires an atomic operation in both acquire and release, while a ticket lock only requires an atomic operation in acquire.

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.

Answer: If x.final exists on the disk, then all files that precede x in alphabetical order are guaranteed to be on disk and complete. The last file might be before or after m and might be truncated.

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.

Answer: All final grade files before the one starting with m will be on disk and complete. Other grade files might also be on disk and those files may be truncated.

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.

Answer: All grade files will be on disk and complete.

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).

Answer: No, KLEE would not find any bugs.

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).

Answer: Yes. Stdin would consist of any number < 0 or > 6, optionally followed by anything starting with a non-digit (because scanf stops reading at the first non-digit).

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).

Answer: Yes. Stdin would consist of any number < 0, 4, or > 6, optionally followed by anything starting with a non-digit.

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?

Before copying the packet buffer, he updates the receive tail pointer, which allows the E1000 to reuse that buffer immediately. He should change the end of his function to

```
    memmove(buf, rx_buffers[tail], rx_descriptors[tail].length);
    int len = rx_descriptors[tail].length;
    write_reg(RDT, (tail + 1) % RX_RING_SIZE);
    return len;
```

(Note it's just as important to read the length before updating the tail pointer as it is to copy the packet contents.)

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.

Answer: Louis is reusing the page passed to the NS environment. Since the NS environment is running in parallel with the input environment, it may still be using that page when Louis reads the next packet into it. Delaying between receiving packets doesn't help, since Louis can't guarantee a bound on how long it takes the NS environment to process a packet.

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

Answer: Replace the `sys_yield` loop with `sys_page_alloc(0, &nsipcbuf, PTE_P|PTE_U|PTE_W);`

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)?

Answer: 11

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

Answer: More time/emphasis on the project. Make a debugging cheat sheet and hold a GDB tutorial. Post homework solutions. More final project suggestions. More coherent lectures and varied lecture styles. Let students write the shell. Slower introduction to xv6 and C. Make a list of common OS C programming pitfalls. More pictures. More memory checking. Choose final project earlier. Cover x86-64 or ARM. Fewer readings but more depth. More familiarization with provided code.

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

Answer: Labs.

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

Answer: Bugs. Quizzes.

End of Quiz