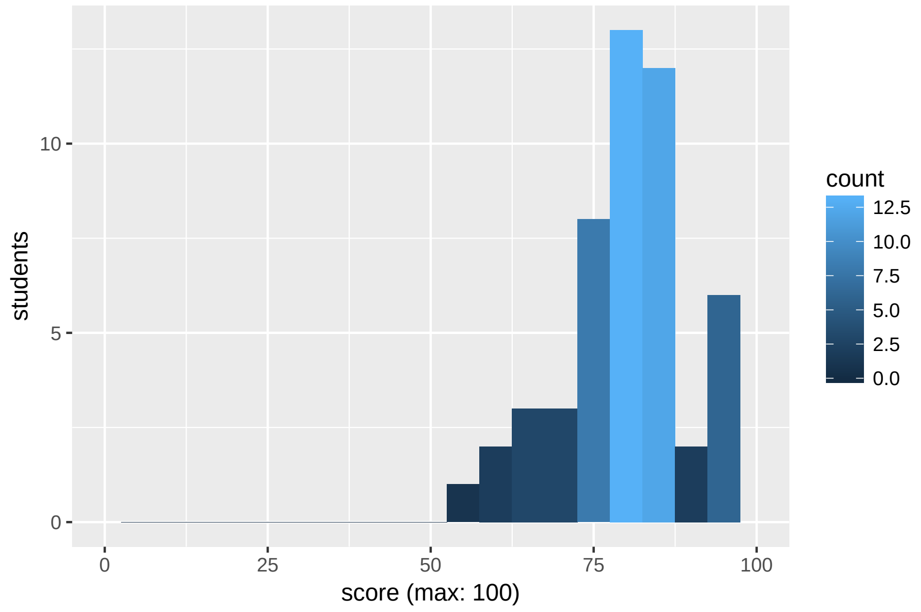
Department of Electrical Engineering and Computer Science

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

6.828 Fall 2017

Quiz II Solutions

Mean 81 Median 81 Standard deviation 10.0



I xv6 Logging

Ben Bitdiddle is looking at commit() in xv6's log.c:

```
commit()
{
  if (log.lh.n > 0) {
    write_log();      // (AAA) Write modified blocks from cache to log
    write_head();     // (BBB) Write header to disk -- the real commit
    install_trans(); // Now install writes to home locations
    log.lh.n = 0;
    write_head();     // (CCC) Erase the transaction from the log
  }
}
```

write_head() writes the log head to the disk and waits for the write to complete. Ben observes that write_head() takes a long time, and he wonders why commit() calls write_head twice. He modifies his commit() to eliminate the second call to write_head() (marked CCC).

1. [6 points]: Describe a sequence of events in which Ben's modification will lead to incorrect file system operation.

Answer: A first system call does some writes and they commit. After commit() returns, the log head still indicates that there is a complete committed transaction on the disk. A second system call starts to commit. After it has written just one of its blocks to the log, the power fails. On reboot, xv6's recover_from_log() will think there is a complete committed transaction in the log, since the n field of the on-disk log header is non-zero. So it will replay all the blocks in the log. But that means just one of the second system calls's writes will be replayed, which breaks atomicity. Just as bad, the recovery code will write the second system call's block to the wrong location: the block address specified in the first system call's log header.

Ben restores the second call to write_head().

Now he notices that write_log() and write_head() end up writing the disk in an order that leads to low performance. On the disk, the log header block is just before the log data blocks, so commit() ends up having to wait for a whole rotation of the disk between the write_log() (AAA) and the write_head() (BBB). Ben swaps lines AAA and BBB, hoping to avoid the extra rotation.

2. [6 points]: Describe a sequence of events in which Ben's modification will lead to incorrect file system operation.

Answer: Suppose call to commit() executes write_head() (BBB) and starts write_log(), but there's a power failure after write_log() has written just one of the system call's blocks to the on-disk log. Recovery will see the non-zero n in the log header and assume the log contains a complete transaction. But some of the transaction's writes are missing from the log, so recovery will fail to preserve atomicity. In addition, recovery will likely write block contents from previous transactions to the disk addresses specified in the interrupted transaction's log header (i.e. to the wrong disk locations).

II EXT3 Logging

Recall Linux's ext3 file system, described in Lecture 14 and in Tweedie's paper *Journaling the Linux ext2fs Filesystem*.

Ben observes that ext3 writes meta-data blocks (i-nodes, directory blocks, free-block bitmaps, indirect blocks) to the disk twice, once to the journal and once to their home locations. Ben is considering speeding up ext3 by writing modified meta-data blocks to their home locations as soon as system calls modify them, and later writing the blocks to the journal.

3. [6 points]: Describe a sequence of events in which Ben's modification will lead to incorrect file system operation.

Answer: Suppose a system needs to update multiple meta-data blocks, for example to initialize an i-node and to add an entry to a directory that refers to that i-node. The system could suffer a power failure after writing one of these blocks to its home location, but before writing the other block. Recovery would not see this transaction in the log, so it would not perform the missing write, and as a result the system call's effects would not be atomic. For example, the directory entry might refer to an un-initialized i-node containing garbage. At a higher level, Ben's idea breaks the "write-ahead" in write-ahead logging.

III Lab 4

Ben is having trouble implementing the user page fault handler for copy-on-write fork in Lab 4: he is confused about how to get the right environment id to pass to the system calls.

4. [6 points]: Ben tries passing the global pointer thisenv to system calls. However, his child page fault handler incorrectly issues system calls to the parent's environment id. Assuming the rest of Ben's code is correct, what could be causing this bad behavior?

Answer: It sounds like the child is using thisenv before updating it to refer to the child. Better would be for the page fault handler to pass zero as the environment ID to system calls.

Ben's lab4 is still broken and `forktree` crashes with unexpected page faults. After much writing, re-writing, and debugging, he ends up with the following:

```
envid_t
start_fork(void)
{
        // set parent's page fault handler...
        return sys_exofork();
}

envid_t
fork(void)
{
        envid_t child = start_fork();
        if (child == 0) {
                // child stuff...
        } else {
                // parent: call duppage() to map the parent's pages in the
                // child, etc...
        }
        // finish fork and return
}
```

After tinkering with his code, Ben is astonished to find that his lab4 works perfectly if he moves the call to `sys_exofork` from the `start_fork` function to the `fork` function as follows:

```
void
start_fork(void)
{
        // set parent's page fault handler...
}

envid_t
fork(void)
{
        start_fork();
        envid_t child = sys_exofork();
        if (child == 0) {
                // child stuff...
        } else {
                // parent: call duppage() to map the parent's pages in the
                // child, etc...
        }
        // finish fork and return
}
```

5. [6 points]: Explain why Ben's lab4 crashes when `sys_exofork` is called from `start_fork`. Assume that all other code is correct.

Answer: The parent gives the child a copy of its memory, including the stack. However, the point at which the parent copies its stack is in fork(), after its call to start_fork() returns and after the parent has made other function calls that overwrite the stack frame created by its start_fork() call. As a result, the child receives a stack in which there is no longer a valid stack frame for the call to start_fork(). However, the child's initial EIP and ESP are copied from the parent at the time of the sys_exofork(), and thus don't make sense if there is no stack frame for start_fork(). When the child starts to run, it will try to return from start_fork(), but pop garbage off its stack instead of a valid return program counter.

IV Virtual Machines

For these questions, refer to Adams and Agesen's *A Comparison of Software and Hardware Techniques for x86 Virtualization*.

6. [6 points]: The x86 `CLI` and `STI` instructions disable and enable interrupts respectively. They each take a dozen or so cycles to execute — the reasons for these times are not documented but probably have something to do with serializing the delivery of interrupts. Surprisingly, with VMware's software VMM, a guest kernel can execute CLI and STI instructions faster than it could on a native machine. Explain how VMware binary translation handles a guest's CLI and STI instructions.

Answer: The binary translator replaces the guest kernel's CLI and STI instructions with memory references (loads and stores) to a "shadow" EFLAGS register. Real interrupts are left enabled, and the VMM uses the shadow EFLAGS to determine whether it is safe to forward interrupts to the guest kernel.

7. [6 points]: For what purpose does code generated by VMware's binary translator use the x86 segmentation registers?

Answer: The VMM hides its code and data in memory only accessible through the `%gs` segment register, and not through any other segment. The binary translator rewrites guest uses of `%gs` to use a shadow register instead of the real `%gs`. Since most guest kernels only occasionally use `%gs`, this is an efficient way to prevent the guest from using the VMM's memory.

8. [7 points]: Suppose you wanted to port the VMware software VMM (and associated binary translator) to a machine without segmentation but otherwise like the x86 (for example, to a 64-bit Intel CPU). What could your binary translator do in place of using segmentation registers? Explain your answer.

Answer: The VMM must allow the guest to use any virtual address, but must also store its own state (e.g. a shadow EFLAGS) somewhere in memory. One possibility is for the binary translator to insert instructions before every guest memory reference to check if the reference is using the VMM's memory, and (if so) redirect the reference somewhere else. These checks would reduce guest performance. Or the VMM could arrange to use a different page table for the guest instructions and for the VMM's own code; only the VMM's page table would contain mappings to the VMM's state memory. The binary translator would need to insert code to change `%cr3` around instructions it injects that use VMM memory.

V Virtual Memory

For these questions, refer to Appel and Li's *Virtual Memory Primitives for User Programs* and Belay et al. *Dune: Safe User-level Access to Privileged CPU Features*.

9. [7 points]: Ben Bitdiddle is working on a new programming language that uses a concurrent version of Baker's algorithm to perform garbage collection (outlined in Section 3 of the Appel and Li paper and the slide entitled "Solution: Use virtual memory!" in the Virtual Memory 2 lecture). In an effort to improve performance, Ben devises a scheme where he maps unscanned pages in the to-space as read-only instead of disabling boths reads and writes. Unfortunately, the program crashes as a result of this change. Assuming Ben makes no other changes, explain which Baker's algorithm invariant is being violated.

Answer: An object in the unscanned to-space could contain a pointer to the from-space. Because the mutator won't trap on read, one of these pointers could get loaded into its registers. However, Baker's algorithm requires that the mutator sees only to-space pointers. Ben's change breaks this invariant, and as a result the mutator can see old versions of objects in the from-space whose correct versions have already been moved to the to-space.

10. [7 points]: Instead of making changes to the garbage collection algorithm, Alyssa P. Hacker suggests that Ben could use Dune to handle page fault exceptions directly from userspace. After porting the language runtime to Dune, Ben observes a large speedup in garbage collection performance. However, programs that compute a lot but don't involve much garbage collection run a bit slower. What is the most likely explanation for this slowdown?

Answer: Dune uses hardware virtualization's Extended Page Tables (EPTs). Looking up a virtual-to-physical transaction in an EPT requires more memory references than looking up a mapping in an ordinary x86 page table. Thus programs that miss a lot in the TLB will be slower when run under Dune. System call overhead is less likely to be a culprit because workloads that mostly compute don't make many I/O calls to the kernel.

VI Synchronization

For these questions, refer to Boyd-Wickizer et al. *Non-scalable locks are dangerous* and McKenny et al. *RCU Usage In The Linux Kernel: One Decade Later*.

Ben is working on a version of XV6 with the goal of supporting efficient, flexible networking. So far he has implemented the following code to allow for routing parameters to be reconfigured while transmitting packets. Normally, a network stack would support several different routes, but in Ben's prototype only a single route is supported so far.

```
spinlock_t route_lock;
// information about the next router hop.
struct route {
        uint32_t hostip;
        uint32_t gatewayip;
        uint32_t subnetmask;
};
struct route *route;

// Transmit a packet.
void transmit_pkt(struct pkt *p)
{
        struct route r;
        acquire(&route_lock);
        r = *route; // make a copy
        release(&route_lock);
        transmit_on_route(p, &r);
}

// Update the routing information with new parameters.
void change_route(struct route *nroute)
{
        struct route *oroute;
        acquire(&route_lock);
        oroute = route;
        route = nroute;
        release(&route_lock);
        free(oroute);
}
```

11. [7 points]: XV6 normally uses test-and-set (TAS) locks, which are known to be susceptible to congestion collapse. Ben tries replacing XV6's locking scheme with MCS locks. He then tests the system's performance using a single-threaded network application. To his surprise, the CPU time spent in `transmit_pkt()` increased slightly after changing the locking implementation. Why are MCS locks slower in this situation?

Answer: An MCS lock is only beneficial when multiple cores try to acquire the lock at the same time. Because the application is single-threaded, it is unlikely there will be any lock contention. MCS locks require extra instructions on release, so, when uncontended, they have higher overhead than TAS locks.

Recall the Linux RCU API you learned about in lecture. A summary is as follows:

- **rcu_read_lock()** Begin an RCU critical section.
- **rcu_read_unlock()** End an RCU critical section.
- **synchronize_rcu()** Wait for pending RCU critical sections to finish.
- **rcu_dereference()** Signal the intent to dereference a pointer inside an RCU critical section.
- **rcu_assign_pointer()** Assign a value to a pointer that is read in RCU critical sections.

Using RCU, Ben reimplements `transmit_pkt()` and `change_route()` as follows.

```
spinlock_t route_lock;
// information about the next router hop.
struct route {
        uint32_t hostip;
        uint32_t gatewayip;
        uint32_t subnetmask;
};
__rcu struct route *route;

// Transmit a packet.
void transmit_pkt(struct pkt *p)
{
        struct route r;
        rcu_read_lock();
        r = *rcu_dereference(route); // make a copy
        rcu_read_unlock();

        synchronize_rcu();
        transmit_on_route(p, &r);
}

// Update the routing information with new parameters.
void change_route(struct route *nroute)
{
        struct route *oroute;
        acquire(&route_lock);
        oroute = route;
        rcu_assign_pointer(route, nroute);
        release(&route_lock);

        free(oroute);
}
```

12. [7 points]: Ben's code has a bug in the way it uses RCU. Clearly explain the bug.

Answer: The call to `sychronize_rcu()` is in the wrong place. It should be in `change_route()`, just before freeing `oroute`. Ben's misplacement of `synchronize_rcu()` may cause `transmit_pkt()` to use a `struct route` after `change_route()` frees it.

13. [7 points]: How will Ben's RCU bug cause his code to malfunction?

Answer: `oroute` could get overwritten when its reallocated, so `transmit_on_route()` could observe potentially any value (likely an incorrect one).

VII IX

You would like to use IX on your heavily loaded memcached server. Your server has a 40-gigabit/second Ethernet NIC, which supports a single pair of DMA queues (one incoming queue, one outgoing queue). Your server has eight cores, and you'd like to use all of them for memcached. You are thinking of modifying IX so that application threads on all eight cores share the single pair of NIC DMA queues.

14. [7 points]: Explain why this arrangement would likely deliver lower performance than if you used a 40-gigabit/second NIC that supported multiple pairs of DMA queues. Be specific about what would go wrong, and why.

Answer: A single NIC DMA queue used by multiple cores would have to be locked, and the cores would likely have to spend time waiting for the lock. They might also have to spend time waiting to move the cache lines associated with the DMA queue between cores. As a result the server would be able to process fewer requests per second than if each core had a dedicated pair of NIC queues.

Ben Bitdiddle has a multi-core memcached server that uses IX. The server has a NIC that supports many pairs of DMA queues. He is thinking of modifying the way his NIC chooses the queue on which to place each incoming packet. Instead of having the NIC hash the packet's port numbers and IP addresses in order to choose the queue, he wants to modify his NIC to place each incoming packet on the queue that currently has the fewest packets on it. You should assume that this modification is possible, and that it does not slow down the NIC. Ben does not modify IX. His memcached uses TCP.

15. [7 points]: Explain what will go wrong and why.

Answer: One problem is that IX assumes that all the packets for a given connection arrive on the same NIC DMA queue, and that therefor IX need not lock TCP connection state. But Ben's modification will cause multiple cores to process packets from a given TCP connection, and the lack of locking will likely cause TCP to malfunction. If IX were modified to lock TCP connection state, there would be a decrease in performance due to lock contention.

VIII 6.828

16. [1 points]: What was your favorite topic in 6.828?

Answer: Virtual memory

17. [1 points]: Which 6.828 topic should we discard next year?

Answer: Singularity

End of Quiz II