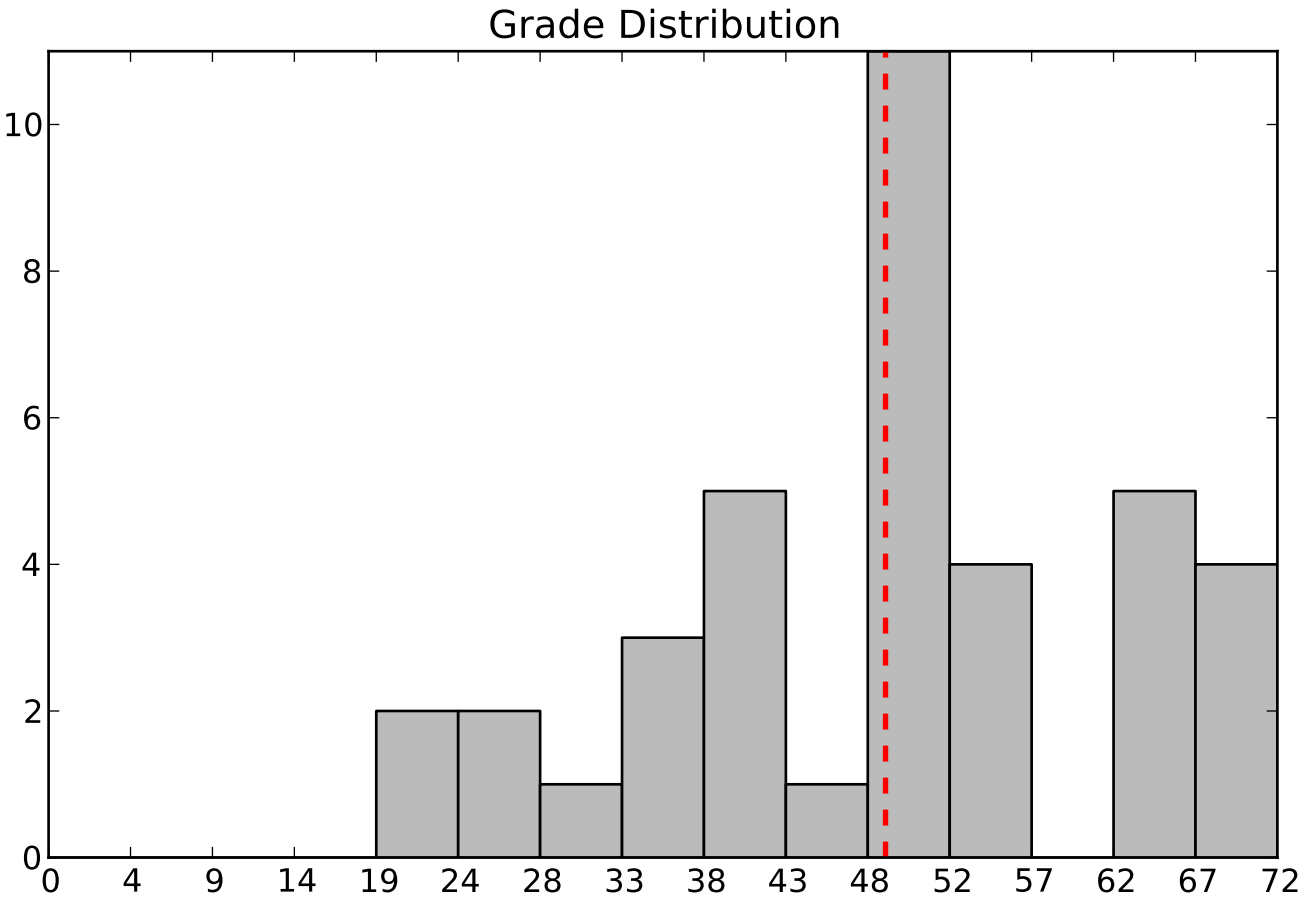

Mean: 49.04 | Median: 50.75 | Std. Dev: 13.09

CS 240 - Midterm Exam

Stanford University
Computer Science Department

May 5, 2015

!!!!! SKIP 20 POINTS WORTH OF QUESTIONS. !!!!!

This is an open-book (but closed-laptop) exam. You have 75 minutes. Cross out the questions you skip. Write all of your answers directly on the paper. Make your answers as concise as possible. Sentence fragments ok. **NOTE: We will deduct points if a correct answer includes incorrect or irrelevant information. (I.e., don't put in everything you know in hopes of saying the correct buzzword.)**

Question	Points	Score
1	10	
2	5	
3	15	
4	5	
5	5	
6	10	
7	10	
8	15	
9	15	
10	10	
Total:	100	

Stanford University Honor Code In Accordance with both the letter and the spirit of the Honor Code, I did not cheat on this exam nor will I assist someone else in cheating.

Name, SUNetID, and Stanford ID number:

Signature:

1. (10 points) Eraser

   (a) (5 points) You program uses lock-unlock functions Eraser does not know about: what happens? Your program uses a alloc-free functions eraser does not know about: what happens?

   Solution: Lock-unlock: You will get false positives everywhere, since locksets will be incomplete. Alloc-free: false positives. The main problem is that eraser must know about re-allocation of memory so that it can reset the locksets associated with the re-allocated memory (since no thread that was using the memory before `free` should be reusing it and thus intersecting the previous locksets would give false positives).

   (b) (5 points) You run the code in Section `4.1` in the Boehm paper using Eraser. Assume the program has no other uses (reads or writes) of `x` or `y`. Will Eraser detect an error? If not, why not? If so, give the exact series of events that lead to the error, including which Eraser state machine transitions happen.

   Solution: Will detect if a scheduling switch happens after the initial `++x` or `++y`. Consider the `x` case: initially it's in the virgin state. The pre-increment `++x` takes it to the Exclusive state. If a switch happens here, the read `x != 1` will transition to Shared with the (empty) candidate set of the current thread. When the system eventually switches back to the first thread, we will then transition to Shared-modified and give an error (since the candidate set is empty). Without this transiton, the writes `++x` and `--x` will be counted as initialization, leaving us in the exclusive state. The subsequent read `x != 0` will just put us in Exclusive.

2. (5 points) The C standards people get sick of listening to Boehm complain, so decree that any `volatile` variable `v` has the following two guarantees: (1) no additional loads or stores can be done to `v` other than what appear in the program text and (2) an access to `v` cannot be reordered with any other volatile access or lock call. Which problems (if any) in Section 4 would this fix?

Solution: It should preclude speculation in loads and stores, which fixes both 4.1 and 4.3. It arguably fixes the global variable problem in 4.2 but we didn't require you say so.

3. (15 points) ESX

   (a) (5 points) 0000 tells you that an implicit page fault where ESX has invisibly evicted a page and must later fetch it from disk in response to a guest OS reference hurts performance much more than an explicit page fault where a guest OS has explicitly evicted a page to disk and later has to fetch it back into main memory. 1111 tells you that this is complete nonsense because Carl, who stood to make money from this fact, never mentioned it. Who is right?

   Solution: You expect invisible page faults to hurt throughput much more than visible ones. For invisible faults, ESX will have to block the guest OS completely since it has no visibility inside of it to run a different kernel thread or process. In contrast, with explicit faults, the guest OS can just switch to another process. (If you just mentioned double paging from the paper, you were given partial credit.)

   (b) (5 points) figure 6: give two ways that this figure could change if ESX did a poor job for this experiment. Figure 7: why do both alloc lines converge to about 179MB rather than some other number?

   Solution: Figure 6: from the guest OS's perspective, good = one of the lines is above toucher at all times since that means the guest OS will get enough memory. From the systems' perspective: the lines should not be dramatically higher than toucher since that means the system would allocate more memory than it should. Figure 7: Both processes are fully active and so get allocated half of available memory. While the machine has 512MB, only 360MB is available for use (since each VMM has a 32MB of overhead). Thus both guest OSes get about 180MB.

(c) (5 points) Recall that the guest OS's page tables map VPNs to PPNs and that ESX's page tables map VPN's to MPNs. What does it mean if there is a VPN mapped to an MPN in ESX, but there is no equivalent VPN in the guest OS page table? What happens if there is VPN mapped to a PPN in the guest OS, but no equivalent mapping in ESX?

Solution: If ESX maps a VMP to anything but the guest OS does not, that is a likely bug —- on the real machine, an access to that VPN would fault, but on ESX it will silently succeed. If there is no mapping in ESX this is safe: when the fault happens ESX can forward the page fault to the guest OSes page fault handler (similar to how the real hardware would).

4. (5 points) Figure 8 in the LFS paper compares LFS's performance to FFS. You redo this experiment, measuring the performance of running one NFS server on top of LFS and one NFS server running on top of FFS (SunOS). Roughly speaking what do you expect to happen to the relative performance difference between NFS+LFS and NFS+FFS?

Solution: We would expect the relative performance difference to significantly decrease. LFS's big win was from large writes (512K - 1MB in the paper) which can't happen with the NFS file system since writes are broken up and synchronously forced to disk before reply. I.e., each FS will have to write to disk before sending an ACK back to the client. LFS can collapse a bunch of these writes, but it can only do one file operation at a time, which will be much less data than a 512K or 1MB segment size. However, LFS may still perform better b/c while it will have one sync write, FFS can have many more because it gives stronger meta data guarantees.

5. (5 points) Leases+NFS: sketch two ways you could use leases in the original NFS paper to significantly improve performance or consistency.

Solution: Attribute cache; file caching in general.

6. (10 points) LFS

   (a) (5 points) Segment S contains a live inode 5 that points to data block 10. Must 10 be live? Segment S contains a dead inode 5 that points to data block 10. Must 10 be dead?

   Solution: It must be live. If block 10 was dead, the inode would have been modified to reflect that and would thus be written to a new location. If inode 5 is dead, data block 10 may or may not be dead. A write could have happened to a different data block the inode pointed to, changing it's location and causing the inode to move.

   (b) (5 points) Give three *clear, obvious* examples drawn from different papers where a system has exploited the fact that it is supposed to do X but did Y instead (for speed, reliability, space, etc) because no observer should be able to tell the difference. These examples can be incidental to the actual contributions of the paper themselves.

   Solution: Concurrency: critical sections. VMware: lying that you are running on the underlying machine. The notion that a program runs sequentially when in fact it's interlaced. compiler optimizations: code reordering, etc. (If all three of your examples were examples from section 4.1, 4.2, 4.3 of Boehm paper, you were given partial credit.)

7. (10 points) Linux Scalability

   (a) (5 points) Imagine a resource in an operating system kernel that is garbage collected using a global reference count. The `increment` and `decrement` operations on the reference count are implemented using locks: a lock is taken before/after the increment/decrement. It is observed that the global reference count is leading to poor scalability in the operating system. Will removing the lock and using atomic instructions to perform the increment/decrement result in perfect scaling as far as the reference count goes?

   Solution: No. The central issue is that the shared resource causes cache coherence to run when the reference count is modified. Even if an atomic is used, the cache line corresponding to the reference count will continue to be shared amongst the cores.

   (b) (5 points) Multiple cores are attempting to lock a variable. The lock is a traditional ticket lock. Explain how the lock causes high levels of cache coherence traffic when multiple cores are waiting for the lock and the core holding the lock releases it.

   Solution: Right before the core holding the lock releases it, every core has the lock's cache line in a shared state. When the core releases the lock by writing to a field inside the lock's allocated memory, the CPU must invalidate every core's cache line and give exclusive access to the unlocking core, causing heavy traffic. Every core continues to spin on the lock, and so immediately, the CPU returns the cache line state of every core to shared, again causing heavy traffic.

8. (15 points) The Hoard authors decide to change their superblock implementation so that there is a single block size class. That is, all superblocks are divided into the same number of blocks, and all blocks are the same size. Each superblock is of size S and each block is of size B. Allocations follow the following algorithm:

```
void *alloc(size_t num_bytes) {
    if (num_bytes <= B) {
        return free_block();
    }

    if (num_bytes <= S) {
        return free_superblock();
    }

    return mmap(NULL, num_bytes, ...);
}
```

Any allocation less than or equal to the size of a block is returned a free block from the per-processor heap. Any allocation less than or equal to the size of a super block is returned a free super block from the per-processor heap. Any allocation greater than the size of a superblock is fulfilled using mmap(). The `free_block` and `free_superblock` functions obtain a free block or superblock in the same way the original Hoard did.

(a) (5 points) In the original Hoard, internal fragmentation was bounded to a factor of b. What design decisions and characteristics of Hoard led to this bound?

Solution: The use of block class sizes $b^1, b^2, ..., b^n$ for some n led to this bound.

(b) (5 points) Assume $B <<< S$. Also assume $B \approx b$. Does the new algorithm increase or decrease internal fragmentation? What about external fragmentation? Why?

Solution: Internal fragmentation is increased but external fragmentation remains roughly the same.

Internal fragmentation is increased because an allocation greater than size B but less than S wastes at most $S - (B + 1) \approx S$ bytes since $B <<< S$. The internal fragmentation of the original Hoard was b. Since $B \approx b$ and $B <<< S$, $b <<< S$.

External fragmentation tracks how many superblocks remain *fully* unused. Because there are no significant differences in how *any* portion of a superblock is used or *full* superblocks are freed, external fragmentation remains the same.

(c) (5 points) How does the new algorithm for allocation affect blowup in the producer-consumer pattern where a producer on one thread allocates an object that is then freed by a consumer on a different thread?

Solution: It doesn't. The bounds on blowup are still constant. Blowup was bounded by moving superblocks between per-core and global heaps. The allocation scheme had nothing to do with bounding blowup.

9. (15 points) Alyssa recently read the *The Scalable Commutativity Rule* and is excited to apply the commutativity rule to her new email sending library, "BitMail". So far, her interface for sending mail is as follows:

```
/**
 * Sends a message with text `msg` to the email address
 * `recipient`. Messages are sent to each recipient in the
 * order this function is called.
 */
void send_message(char *recipient, char *msg);
```

   (a) (5 points) She analyzed the `send_message` API using ANALYZER and determined that `send_message` does not commute with itself when the `recipient` parameter is the same. Why?

   Solution: In short, the state of which messages have been sent directly reflects the order in which `send_message` is called. Because we can tell the order of operations by examining the state, they clearly don't commute.

   (b) (5 points) How can the `send_message` interface be changed so that two sends to the same recipient commute?

   Solution: Remove the requirement that messages are sent in the order the function is called.

Alyssa took your advice and ANALYZER now reports that `send_message` commutes with itself when the `recipient` parameter is the same. Alyssa then asked Ben to implement the function in a scalable way. Unfortunately Ben's code does not pass the TESTGEN generated tests:

```
// per-core message queues, none sharing cache lines
// add messages to a queue to have the message be sent
queue *per_core_qs[NUM_CORES];

void send_message(char *recipient, char *text) {
    // msg_t is a queue node for the message
    msg_t *msg = create_msg(recipient, text);
    long msg_hash = hash(msg);

    queue *q = per_core_qs[msg_hash % NUM_CORES];

    lock(q);
    q->tail->next = msg;
    q->tail = msg;
    unlock(q);
}
```

(c) (5 points) When (under what conditions) and why do the TESTGEN tests fail? What is Ben doing wrong, and how can he fix his problem?

Solution: The TESTGEN tests will fail when `msg_hash % NUM_CORES` evaluate to the same value on two different cores.

Ben is improperly using the per core queue array. In his implementation, he's essentially selecting a per core queue pseudorandomly, so two `send_message` calls on two different cores will share memory when the hash modulo the number of cores evaluates to the same value on both cores. To fix this, he could use the current core's number as an index into the per core queue.

10. (10 points) Lab 1

(a) (5 points) Ben argues that Lab 1 is an example showing that threads *can* be implemented in C and that the *Threads Cannot be Implemented as a Library* is therefore wrong, but Alyssa argues against him. Who's right and why? *Hint: Think about what the word "thread" means in each implementation.*

Solution: Without the extra credit, threads are cooperating. Because the compiler will see a `yield` call as an opaque function, no unknown races, like those demonstrated in Boehm's paper, can occur.

With the extra credit, preemption occurs, so the same issues as presented in the paper apply to the lab 1 threads.

In either case, Boehm's paper was discussing preemptive threads, so it could be said that Alyssa is correct because even if Ben was arguing for cooperative threads, which *can* be implemented without fearing the issues presented in the paper, this is not what Boehm was arguing against. *(Note: Depending on your viewpoint, either person could be right. As long as the explanation was valid, full points were awarded.)*

(b) (5 points) Ben Bitdiddle wants to extend lab 1 so that the initial function of a thread is passed in a `void *` parameter containing a pointer to any data its parent wants it to have access to. Will Ben have to modify his `spawn` stack layout code to accomplish this? Why or why not?

Solution: He will not have to modify the stack layout code. As per the System-V x64 ABI, the first parameter is passed inside register `%rdi`.