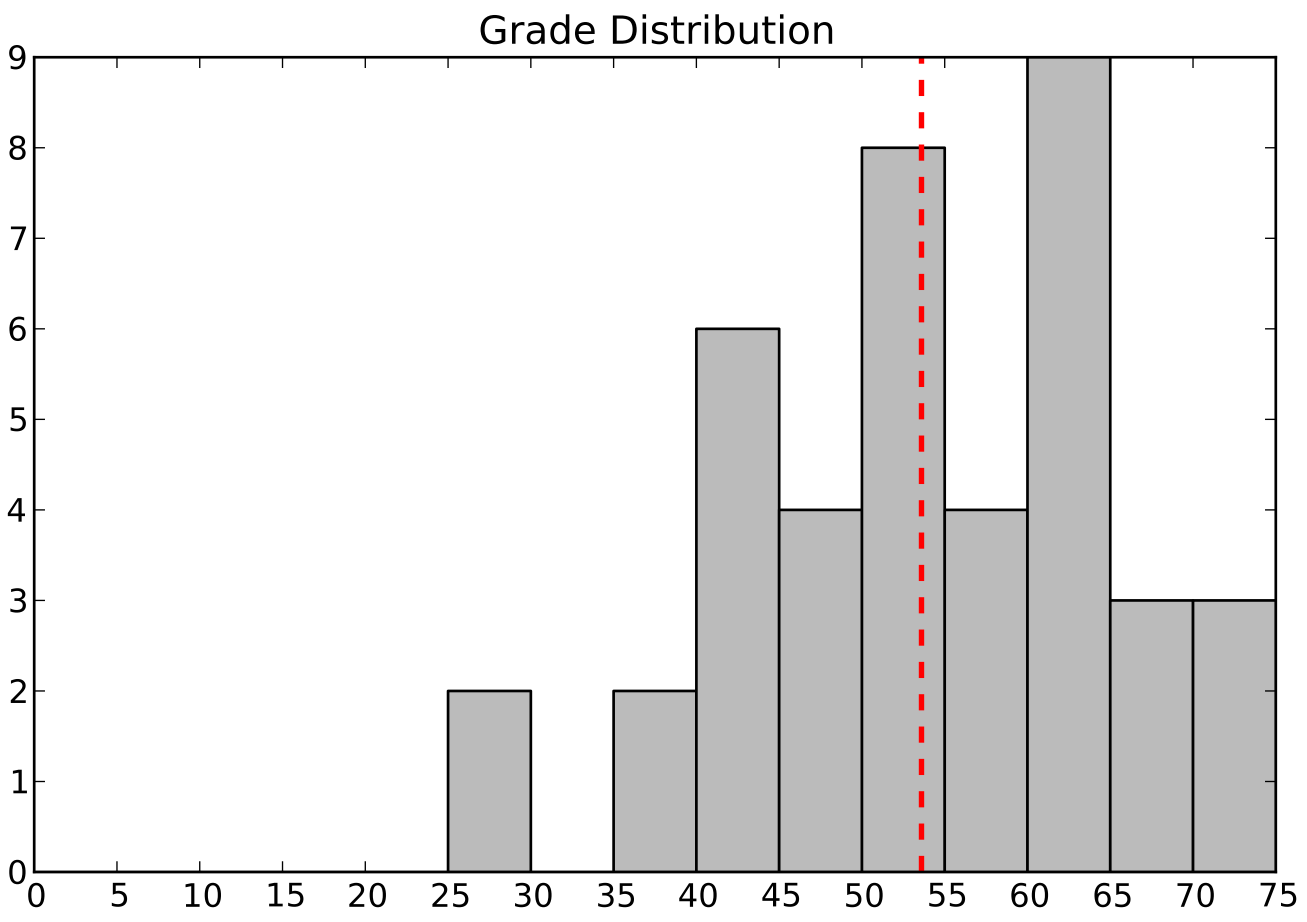

Mean: 53.60 | Median: 54.00 | Std. Dev: 11.23

CS 240 Final Exam

Stanford University
Computer Science Department

June 2, 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 write in everything you know in hopes of saying the correct buzzword.)**

Question	Points	Score
1	15	
2	10	
3	20	
4	10	
5	10	
6	10	
7	10	
8	15	
Total:	100	

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

Name, SUNetID, and Stanford ID number:

SOLUTIONS

Signature:

1. (15 points) Battle of the Xs: EXO vs ESX

   (a) (5 points) Carl says that since any unprivileged user can write a guest OS it's easy for ESX to export an exokernel disk interface: just give each guest OS a raw disk partition. State the main way this approach arguably satisfies exokernel principles; give the main way it arguably violates them.

   Solution: For: it's low level, physical names, could have complete control over the partition (depending). Against: coarse-grain, hard (or impossible) to share. Exokernel wants fine-grained control.

   (b) (5 points) The exokernel people say that one trivial disk interface would be to associate a capability with each disk block using an on-disk table. That way if you had a capability you could read or write blocks as you wanted. Carl says this will perform much more slowly than a raw partition and, plus, be vulnerable to crashes. Is he right? Why or why not?

   Solution: Really bad idea. On-disk table = in worst case two disk reads to access a block (the block itself and the block that holds the needed capabilities), also two writes. Caching could mitigate this, but it's burning more memory than a non-exo system. Also, two locations means you can crash after writing the first but before the second, which can leave you in a bad state (e.g., access to blocks you should not have, or not having access to blocks you should have).

(c) (5 points) Give an intuitive explanation for how to use an exokernel's fast page fault handling to reimplenment Eraser-like race detection without rewriting the binary (you may assume you can relink applications).

Solution: Protect the heap. Every load or store would trap: you can do lockset calculations here. You can relink to change malloc/free/lock/unlock etc to be wrappers that call into your system so you can track things just like Eraser does. Given how slow eraser was, this approach may not in fact be too bad in comparision.

2. (10 points) Rethink this.

   (a) (5 points) Ignore cleaner overhead for this question. Mendel says that with a 30 second flush, LFS should be at least as fast as xsyncfs. What is the intuition for why this *could* be true? If we assume it is true, is there any good reason to prefer xsynfs over LFS?

   Solution: It could be faster: it will group commit everything into one place and since it flushes every 30 sec it doesn't pause otherwise. The lack of pause + group commit was xsyncfs's big win, so LFS should at least be in the same ballpark (possibly better). On the flip side, if the user does sync() it will be slower (since it will block then) and it also has weaker guarantees — i.e., it can return from a system call, the program could then print or send a network message implying/stating the operation completed, but a crash would kill the data.

   (b) (5 points) In the midterm, running NFS on top of LFS removed much of the performance gain of LFS. If we run an NFS server on top of an xsyncfs system, does something similar happen? Why or why not?

   Solution: Observability requires xsyncfs writes to disk before replying to a network message, which obviates most of it's benefit. This is similar to how LFS had to write to disk before replying, removing most of the benefit since it could no longer do large sequential log writes. xsynfs still can improve when there are multiple clients.

3. (20 points) Deviant stuff.

   (a) (5 points) There are a variety of procedures you cannot call from a Linux kernel interrupt handler. Assuming you have a list of interrupt handlers, and by construction, non-interrupt handlers, explain how to do a MAY belief checker to catch such mistakes, ranking them from most to least likely.

   Solution:

   1. Count the number of times E a function is called from call chains that (transitively) start from an interrupt handler.
   2. Count the number of times S a function is called from call chains that (transitively) do not start from an interrupt handler (e.g., system calls, the boot code).
   3. Flag calls in (1) based on the test statistic value based on E and S —- functions with high values for S and low values for E are likely mistakes.

   (b) (5 points) You have a preexisting checker that can flag when code uses network data without first sanitizing it. The checker is having a hard time flagging code on a new kernel that does not clearly indicate data coming from the network. You notice the kernel frequently calls `ntohl` and `ntohs` which convert 32-bit and 16-bit integers from "network byte order" to "host order." How can the checker use these calls to find errors?

   Solution: Passing a value `x` to `ntohl(x)` or `ntohs(x)` implies it came from a network message. You can mark the buffer `x` came from as network data and `x` as tainted.

(c) (10 points) Sketch, by providing pseudocode or intuition, a brand new belief-style checker (either MUST or MAY) for some important bug type of your choice. *Hint: It may help to think about common bugs you find in your own programs.*

4. (10 points) Native client: consider the code in figure 3

   (a) (2.5 points) A code modification makes `inst_is_disallowed` a nop (no op). What is an externally visible action (in the sense of what xsyncfs would consider externally visible) that an attacker could now perform that native client is trying to prevent?

   Solution: Issue a system call instruction to send a network packet or write to disk.

   (b) (2.5 points) Does this code force direct jumps to be aligned to 32-bytes? Why or why not?

   Solution: You know where direct jumps go, so they don't have to be aligned; therefore they don't have to end a 32-byte block.

   (c) (2.5 points) For the check `Block(StartAddr[icount-2] != Block(IP))`, what is the opcode for the instruction `IP` points to?

   Solution: Put the first instruction of a nacl jump at the end of one block (B1) and the indirect jump in the next (B2). Jumping to B2 will skip the alignment instruction.

   (d) (2.5 points) What is an attack if you delete the `else` branch?

   Solution: Similar to previous question: JumpTargets will contain indirect jumps as valid targets. A direct jump can jump right to them.

5. (10 points) For each statement about Singularity below, choose **true** if you believe the statement is true. If you believe it is false, choose **false** and provide brief justification.

   (a) (2.5 points) **true — false**
   Singularity would remain just as secure if SIPs were not compiled in the kernel.

   Solution: False. SIPs needs to be compiled at install time in the kernel to ensure that all of the language invariants are indeed maintained. If the SIP is compiled outside of the kernel, the necessary type and memory information would be lost.

   (b) (2.5 points) **true — false**
   As long as the references point to different memory locations, it is possible for two SIPs to hold memory references to the same exchange heap.

   Solution: True.

   (c) (2.5 points) **true — false**
   A SIP uses channels to communicate with other SIPs and the kernel.

   Solution: False. SIPs communicate with the kernel through ABI calls.

   (d) (2.5 points) **true — false**
   Singularity is based on a microkernel-like architecture.

   Solution: True.

6. (10 points) Singularity

   (a) (5 points) The system-wide state isolation invariant allows Singularity to garbage collect each SIP independently. Explain why a SIP not maintaining the SI invariant would not be able to be garbage collected independently.

   Solution: One possible violation of the SI invariant involves a SIP holding a reference to an object in another SIP. In order for the garbage collector to free that object, the GC would need to inspect both SIPs and thus would not be able to garbage collect each SIP independently.

   (b) (5 points) How do Singularity's kernel calls (ABI calls) differ from typical Unix kernel calls (system calls)?

   Solution: 1) Singularity's ABI calls are function calls; the calls do not cross protection domains and no address space switch is made. Typical Unix kernel calls use the `syscall` instruction to switch address spaces and protection rings. That is, Singularity's ABI calls don't involve crossing hardware protection barriers.

   2) Singularity's ABI is strongly versioned. A SIP must describe explicitly in its manifest which version of the ABI to use.

   3) Singularity's ABI cannot be used to alter the state of any other SIP but the calling SIP. This contrasts Unix where calles like `signal` change the state of other processes.

7. (10 points) Ben Bitdiddle is attempting to use your Lab 2 SNFS server to write to a file located at path `/a/b/c.txt`. His client has mounted and obtained the root file handle `0xF00D`; his client hasn't done anything else.

   (a) (5 points) Describe the series of messages Ben's client must send to the server to write "Hello, world!" to the beginning of `/a/b/c.txt`. *Hint: Ben's client must send at least 4 messages to the server.*

   Solution:
   LOOKUP(0xF00D, "a") -> a_fh
   LOOKUP(a_fh, "b") -> b_fh
   LOOKUP(b_fh, "c.txt") -> c_fh
   WRITE(c_fh, 0, 13, "Hello, world!") -> 13

   (b) (5 points) Ben is trying to extend the server to support a `REMOVE(dir, name)` operation, which, given a file handle `dir` for a directory and a file name `name`, unlinks the file named `name` from the directory pointed to by the handle. Besides unlinking the file in the file system, what else must the SNFS server do to ensure successful removal?

   Solution: The entries (name -> fh and fh -> name) in the file handle database must also be removed. If they are not removed, the server will be internally inconsistent.

8. (15 points) Assume the following MapReduce interface has been provided to you:

```
map(Iterator it) -> List<Pair<Key, Value>>
reduce(Key k, List<Value> vals) -> List<Pair<Key, Value>>
```

That is, your `map` method takes as its parameter an iterator and returns (by calling `emit`) a list of key value pairs. Your `reduce` function takes in a key and all of its associated values and returns a list (by calling `emit`) of values. Using this interface, a word count MapReduce program could be written in pseudocode as follows:

```
def map(it):
  for word in it:
    emit(word, 1)

def reduce(k, vals):
  emit(k, len(vals))
```

(a) (5 points) Complete the pseudocode below for a MapReduce program that inverts an index. An index is a mapping from word to location; an inverted index is a mapping from location to word.

Solution:

```
def map(it):
  for (word, location) in it:
    emit(location , word)

def reduce(k, vals):
  emit(k, vals)
```

(b) (5 points) Complete the pseudocode below for a MapReduce program that filters the corpus of text to only those words that appear more than 'N' number of times. You may use 'N' in your program.

Solution:

```
def map(it):
  for word in it:
    emit(word, 1)

def reduce(k, vals):
  if len(vals) > N:
    emit(k)
```

(c) (5 points) With *The Scalable Commutativity Rule* paper in mind, do two `map` operations in the MapReduce interface always commute? What about two `reduce` operations? Why or why not?

Solution: Yes, two map operations (almost) always commute and two reduce operations (almost) always commute. Regardless of the order they are run, `map` and `reduce` operations will always return the same results. **Note:** It is not the case that map operations always commute with reduce operations. **Note:** Successful arguments can be made that non-deterministic map or reduce tasks do not commute. **Note:** An argument can be made that in the case of two redundant reduce tasks, only the output of the first reduce task to finish will be visible thus exposing an ordering.