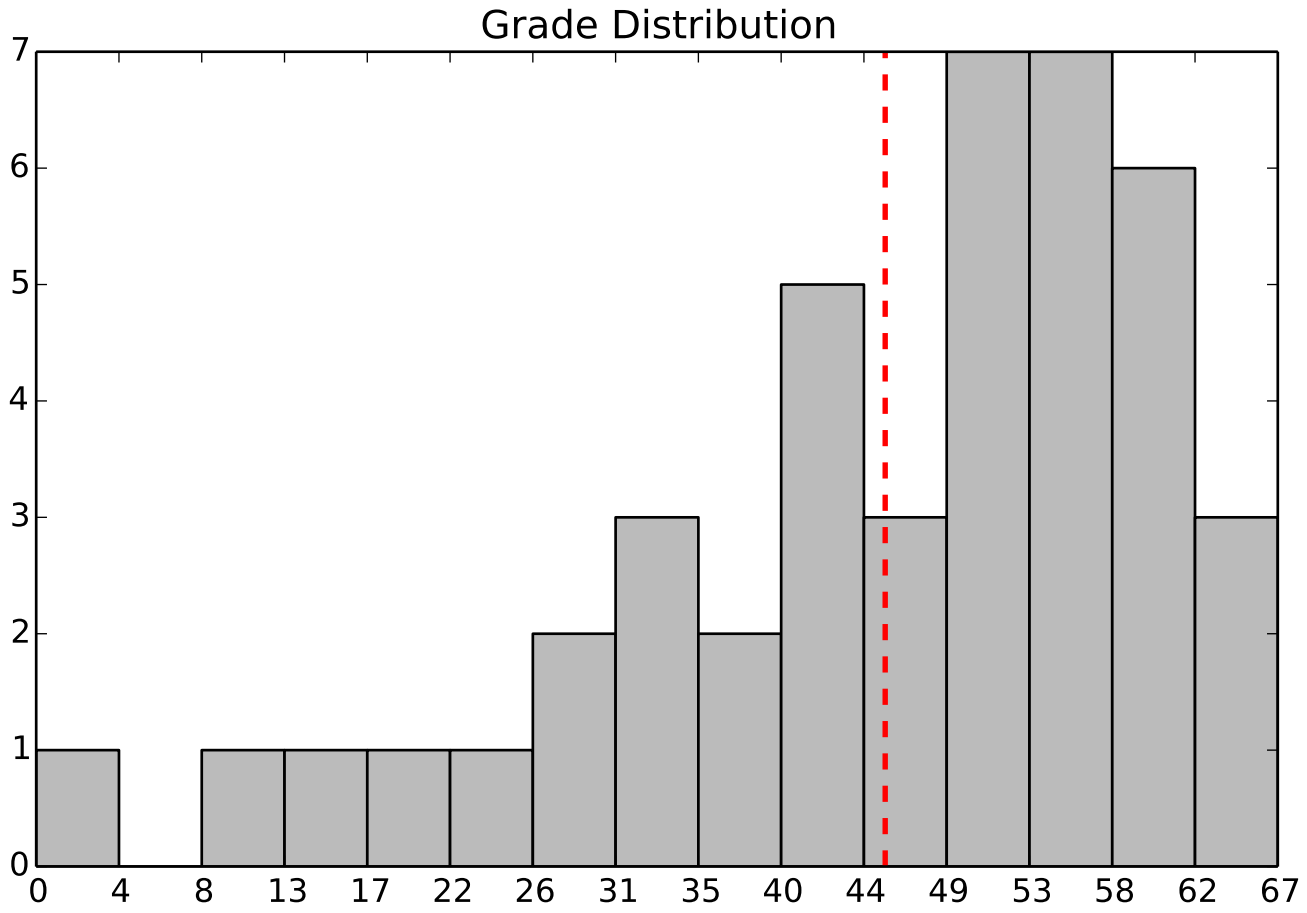

Mean: 45.81 | Median: 50.00 | Std. Dev: 14.58

CS 240 Midterm

Stanford University
Computer Science Department

April 28, 2016

!!!!! SKIP 15 POINTS WORTH OF QUESTIONS. !!!!!

This is an open-book (but closed-laptop) exam. You have 80 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	10	
4	5	
5	10	
6	10	
7	15	
8	5	
9	15	
Total:	95	

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) Ben Bitdiddle just finished reading §3 of Hoard. After reading, "In the implementation we actually use 2P heaps [...] in order to decrease the probability that concurrently-executing threads use the same heap," Ben decides to take this idea to the *max* and forks Hoard on Github. He modifies Hoard to have per-thread heaps instead of per-processor heaps. As such, a thread's `malloc()` will be handled by the per-thread heap and `free()`'d memory will be returned to the heap where the memory allocation occurred. Ben doesn't change anything else about Hoard. Ben names his modified version of Hoard, *Stockpile*.

   (a) (5 points) Describe why Stockpile might perform allocations *faster* than Hoard.

   Solution: Concurrently executing threads can no longer block on trying to aquire the per-processor heap lock.

   (b) (5 points) Recall that threads can migrate between processors. In comparison to Hoard, does thread migration on a system running Stockpile lead to more, less, or about the same amount of false sharing? Explain your answer.

   Solution: Less. Say you have a thread A and thread B on processor P1 and P2, respectively. On Hoard, if A migrates to P2 and B migrates to P1, allocations from A and B may result in cache line sharing. This can't happen in Stockpile since it gives each thread a different heap.

(c) (5 points) Describe a scenario where Stockpile allocates significantly more memory from the operating system than Hoard.

Solution: Say we have one processor and N threads, each of which makes a single allocation of the same size s. This requires Stockpile to allocate N superblocks. Hoard would have only allocated $(N * s)/S$ superblocks. If s is small and S is somewhat large, then Stockpile allocates N times more memory from the OS than Hoard.

2. (10 points) Trust

   (a) (5 points) Consider the string declaration "`char s[]`" in Thompson's first Figure. Where does the beginning `'\t', '0', ...` correspond to in the code? What is the last character of the deleted 213 lines?

   Solution: The last part of the array. '`{`' or possibly '`\n`'.

   (b) (5 points) **In at most 10 lines**, write the pseudo-code for "bug 2" in figure 3.3 using ideas from the paper; you can refer to other figures in the paper. What is the intuition for why you need self-replication?

   Solution: Basic idea: you need an array that contains the code for the if-statements and two bugs. You print this out a character at a time and then print the whole thing as with the original self-replicating code. If you just inject the attack, then if the compiler is used to compile itself the attack will be lost.

3. (10 points) After developing Goroutines (user-level threads) for the Go programming language, developers at Google encounter some of the same issues raised by the Scheduler Activations paper. In particular, they're concerned about the time it takes to context switch between kernel-level threads, as well as with the inability to control which user-level threads are scheduled when.

   To solve these issues, the engineers abandon user-level threads and instead map each Goroutine to a lightweight kernel-level thread. Then, they propose (they really did, in 2013!) and implement a *switch_to(tid)* system call in Linux. The specification is below:

```
/**
 * Set the current thread's state to WAITING and the thread with
 * 'tid''s state to RUNNABLE. The current thread will not resume
 * automatically; it must be 'switch_to'd. The thread with 'tid'
 * will be scheduled as soon as possible.
 *
 * @param tid The thread id of the thread to switch to.
 */
void switch_to(int tid);
```

   (a) (5 points) The Google engineers benchmark `sched_yield()`, a syscall that asks Linux to find a new thread and run it, against `switch_to` and find that `switch_to` is 20x faster. After profiling, they determine that much of the cost is due to what was happening inside the kernel and not due to the cost of context-switching in and out of kernel-space. What could the kernel be doing in one case that it doesn't have to do in the other that could lead to such a drastic performance difference?

   Solution: The operating system is no longer making scheduling decisions as it was with `sched_yield()`. As such, it really only need to change the state of the threads to handle `switch_to()`.

(b) (2 ½ points) Is a user-level threading library that uses `switch_to` susceptible to the spin-lock problem in §2.2 of Scheduler Activations? Why or why not?

Solution: Yes. These are just preemptive kernel-level threads with no notification of preemption. Thus, the library can't run critical sections to completion when a thread in a critical section is preempted.

Alternatively, one could successfully argue that an implementation of locking (that first checks if the lock is held and `switch_to()`s the thread holding the lock if it is not already running) ameliorates the situation. This means that a user would have to always use the library's lock implementation. In a language like Go, this is achievable.

(c) (2 ½ points) What are two clear advantages that this approach has when compared to the solution presented in the Scheduler Activations paper?

Solution:

- Now longer have the large cost of scheduler activations.
- The API is way simpler.
- All the tools should work just fine since we're doing 1:1.

4. (5 points) Eraser

   (a) (5 points) Draw the state diagram if you deleted their "initialization hack" but kept the reasonable restriction that you should not give error messages for memory only accessed by one thread. Then, give a short (3 line or less) code snippet with two threads that contains a legitimate bug that Eraser will now flag and explain why.

Solution: This is a little tricky if we want to make sure that a r/w on private memory does not give an error. A common mistake was to have an exclusive state where R/W by the first thread stays, and a subsequent read by another thread goes to a ReadOnly state — this would not flag an error if the first thread wrote and a second thread read (i.e., the init hack they did). V goes to WriteExclusive and stays there for all writes by the first thread, intersecting the lockset. A write by a new thread goes to SharedModified; an error is flagged if the lockset is empty. V on read goes to ReadExclusive, stays there for all subsequent reads by the same thread, intersecting the lockset. A write by the same thread goes to WriteExclusive. A write by another thread goes to SharedModified (and gives an error for an empty lockset). A read by another thread goes to ReadOnly and stays there for subsequent reads (lockset is being intersected). Write goes to shared modified; error on empty lockset. Note you'll have to add some memory to hold the TID.

```
T1                  T2
 x=1;
                    x++;      // no concurrency control.
could happen in either order.
```

5. (10 points) Boehm is looking for something new to whine about. He starts complaining that file locks can't be implemented in an isolated lock server, but rather that they must cooperate at some level with NFS (despite the NFS paper's suggestion to the contrary). As an example of a typical problem, he points to the following code that acquires a lock "`l`" from a remote lock server, does a sequence of operations to a remote file served by an NSF server:

```
fd = open("/remote/a");   // remote file --- served over NFS
lock_server_lock("l");
read(fd, ...);
...
write(fd, ...);
...
lock_server_unlock("l");
close(fd);
```

   (a) (5 points) Using reasoning similar to that in the Boehm paper (where everything not explicitly forbidden is permitted), give two examples of how the code could have it's intended mutual exclusion violated. Which code fragment in the Boehm paper has the most similar problem to the above code and why?

   Solution: Writes can be pushed below the unlock. Reads could be prefetched before the lock (e.g., at the open). Seems most similar to the last loop example where writes are pushed outside of the loop's critical section.

(b) (5 points) What is the simple, but perhaps inefficient change to **only** the NFS client code that fixes these problems? How can you solve these problems more efficiently by changing both the lock server client code and NFS client code?

Solution: The simple, inefficient solution is to do all operations synchronously so they occur in program order — force all writes to the server when they occur and fetch all reads when (and only when) they occur. A bit more clever is to do the analogue of a memory barrier — flush all outstanding operations to the server at each lock and unlock. Generally we took points off if your solution didn't work, or went if a client crashed.

6. (10 points) Livelock

Your cs140 partner says that livelock can be solved by wrapping the network processing code in a monitor and having the NIC simply use "the well-known wakeup waiting switch" to signal when packets arrive. Assume the network monitor provides two routines, "`msg *receive(void)`" which returns a received packet and "`void send(msg *m)`" which sends one. With this interface, the kernel packet forwarding example in the paper can be written roughly as:

```
ENTRY forward() {
        message *m;
        while ((m = receive()))
                send(m);
}
```

(a) (5 points) Will this implementation livelock? Explain why or why not. If it livelocks, what mistakes does it make that are similar to the livelock-prone in-kernel forwarding code in the livelock paper? If it does *not* livelock, how does it prevent doing so in ways similar to the paper's fixed version?

Solution: Won't livelock. The key problem in the livelock paper was that packet handling code was always scheduled when packets arrived (on interrupt). This meant that transmit and anything else would not run if packets arrived quickly enough. In a sense it was a "push" architecture where packets were shoved in the system whether it was able to process them or not. The code above is a "pull" architecture where there are no interrupts and the code pulls packets only when it is able to process them.

(b) (5 points) Assume that you either fix the issues in the previous part or that it had none. You have a routine `display()` for displaying running statistics about the packet forwarding that works by repeatedly sleeping for a second and displaying the forwarding statistics. You simply fork the forwarding routine and the display procedure so they run concurrently:

```
fork forward;
fork display;
```

Based on your knowledge of MESA (assume it never preempts threads at the same priority level): Will this always, sometimes, or never have the problems described in the livelock paper? Why or why not?

Solution: Since `forward` spins in a tight loop and `receive` will only yield the processor when it needs to wait for a packet, then as long as packets arrive quickly enough, `forward` will always run, starving `display`.

7. (15 points) Scalable Commutativity

Consider the following interface specification, which is similar to the specification in the Scalable Commutativity reading question:

```
/*
 * After this call, the global value of 'GLOBAL' is 0.
 */
void reset();

/*
 * Returns the global value of 'GLOBAL'.
 */
int get();

/*
 * After this call, the global value of 'GLOBAL' is 'x'.
 */
void set(int x);

/*
 * Increments the global value of 'GLOBAL' by 1.
 */
void inc();
```

Also consider the following histories:

```
1. T1: set(x)  reset
   T2:                  reset

2. T1: inc              set(x)
   T2:         reset

3. T1: get
   T2:         set(y)
```

We copy the corrected[1] version of the definition of SIM-commutativity below:

Definition. An action sequence Y SIM-commutes in a history $H = X \,||\, Y$ when for any prefix P of any reordering of Y (including $P = Y$), P SI-commutes in $X \,||\, P$.

(a) (3 points) Which histories are not **SI**-commutative under all possible states and parameters? You needn't explain why.

Solution: 2, 3

(b) (6 points) Which histories *are* **SI**-commutative under certain or all states and/or invocation parameters? For each answer, say whether a certain state is required and explain the state or parameters that would be required, if any.

Solution:
1. (any x, any state)
2. (x = 0)
3. (y = GLOBAL)

(c) (6 points) Which histories are **SIM**-commutative under certain or all states and/or invocation parameters? For each answer, say whether a certain state is required and explain the state or parameters that would be required, if any.

Solution:
1. (x = 0)
3. (y = GLOBAL)

[1] The paper uses the word *some* instead of *any* in "of any reordering of Y". This wording was later changed in Austin's thesis.

8. (5 points) Lab 1

Imagine that the `grn_thread` structure was modified to be:

```
typedef struct grn_thread_struct {
  ...
  grn_context *context;
  ...
} grn_thread;
```

...where `grn_context` continues to be defined as:

```
typedef struct grn_context_struct {
  uint64_t rsp;
  ...
  uint64_t rbp;
} grn_context;
```

Note that `context` is now a pointer in `grn_thread` where it was previously a value. The definition for `context_switch` is changed accordingly:

```
extern void grn_context_switch(grn_context **, grn_context **);
```

(a) (5 points) Finish the implementation of `grn_context_switch` below by inserting code between the comments that makes `grn_context_switch` correct.(Hint: You can do this in 2 instructions.)

```
grn_context_switch:
  push %rbp
  push %rbx
  push %r12
  push %r13
  push %r14
  push %r15
  push %rsp
  /* your code below  */



  /* end your code  */
  pop %rsp
  pop %r15
  pop %r14
  pop %r13
  pop %r12
  pop %rbx
  pop %rbp

  ret
```

Solution:

```
grn_context_switch:
  push %rbp
  push %rbx
  push %r12
  push %r13
  push %r14
  push %r15
  push %rsp

  /* your code below  */

  mov %rsp, (%rdi)
  mov (%rsi), %rsp

  /* end your code  */

  pop %rsp
  pop %r15
  pop %r14
  pop %r13
  pop %r12
  pop %rbx
  pop %rbp

  ret
```

9. (15 points) NFS

   (a) (5 points) As discussed in class, NFS clients defer flushing data to the server until file close. What happens if another client reads the file while close is occurring? Sketch how to fix this problem if NFS provides an atomic rename operation. (Hint: we discussed this method in class on tuesday.)

   Solution: It does not flush the data atomically, so a client that reads at the same time can see a mishmash of old and new data. The "easy" fix is to write all the data back to the NFS server in a temporary file, flush, and then atomically rename it.

(b) (10 points) You can view file read and write as loads and stores. Assume the NFS guys build an EraserNFS that runs on an NFS server and uses logic identical to the Eraser paper to flag potential races in file data accesses of NFS clients by monitoring their read and write calls. Sketch how to handle the following two issues. One, since NFS does not have locks, what problem does this cause, and what's a intuition for how to partially address the problem without adding them? Two, Eraser must know about `malloc` and `free` — what is the analogous problem here and how could you fix it?

Solution: We took a variety of answers for this question. One argument is that without locks there is no way for multiple processes to write to the same file, even if these writes were separated logically across time. One partial hack to would be track open and close and only warn when multiple clients wrote to the same open file. A second argument is that most files have a single writer and many readers of the given copy (e.g., emacs can write a file, gcc will read it, wc might count it, grep might be used to look for an identifier in it). In this case eraser's initialization hack will solve the issue until the next "complete write" of the file (e.g., when emacs writes out a new version). An easy hack would be to have a complete overwrite set the blocks to virgin. Two: you must have some notion of when data blocks are reused for a new file. One possible hack is the code that increments the generation number for an inode on free also goes and resets all the data blocks to the virgin state (you also do this on truncate.)