Stanford University
Computer Science Department
CS 140 Final Exam
Dawson Engler
Fall 1999

This is an open-book exam. You have 180 minutes to answer as many questions as possible. The number in parenthesis at the beginning of each question indicates the number of points given to the question. Write all of your answers directly on the paper. Make your answers as concise as possible. Sentence fragments ok.

Question	Points	Score
1	40	
2	15	
3	15	
4	25	
5	25	
6	30	
total	150	
bonus	5	

Stanford University Honor Code

In accordance with both the letter and the spirit of the Honor Code, I did not cheat on this exam.

Name:

Signature:

1. Short-attention-span questions (40 points)
Answer each of the following questions, and give a sentence or two justification for your answer (5 points each).

1. (5 points) Does a fully utilized disk imply the scheduler is doing a good job interleaving I/O and CPU bound jobs?

2. (5 points) If we can store persistent data on a local disk, we can also store it on a remote one. In this case, if you cache file blocks rather than reading them from the remote disk on every access you notice that decreasing the block size results in less network traffic when you evict a cached block. What's a likely reason for this?

3. (5 points) Would you expect an extent-based file system to have more or less space overhead than FFS as the disk becomes increasingly full? (Recall: an extent-based file system simply adds a counter to each disk pointer, which tracks the number of blocks the pointer refers to.)

4. (5 points) Give a reasonable disadvantage to using hierarchy in the systems covered during lecture.

5. (5 points) On an exokernel operating system, does the code that decides how many disk blocks to prefetch on file read reside in the library operating system or in the exokernel itself?

6. (5 points) What speed trick can we do to optimize context switching for exiting or newly created threads?

7. (5 points) When a phone is disconnected, phone companies try to keep a FIFO list for reassignment to minimize the chance that the new owner will get calls intended for the old recipient. Briefly explain what similar problem TCP sequence numbers are used to solve (and how).

8. (5 points) While we primarily use caches for speed, how can they increase reliability for name translation in a distributed system? (E.g., to cache the binding between DNS name and IP address, between IP address and Ethernet address, between DNS suffixes and name servers, etc.)

2. Memory allocation (15 points)

Spastic after cramming for exams, your partner blurts out that free really serves two functions: (1) reusing virtual address space, and (2) reusing storage. She declares that as a result free is obsolete on new 64-bit address space computers and we only need to implement malloc. Her reasons are that (1) reclaiming virtual addresses is unimportant since you can allocate memory until the sun dies before running out of bits and (2) the operating system reclaims physical memory via paging, making any reclamation the memory allocator does redundant. Assuming we have plenty of disk space, is her conclusion right or wrong? (And, of course, justify your answer.)

3. Concurrency fun (15 points)
The following code attempts to implement an atomic stack. Give a short, intuitive correctness argument for why it works, or state how it is vulnerable to race conditions. (Assume the stack is initialized correctly.) You may rip out this page.

```
typedef struct stack {
    lock_t head_lock;   /* used to lock head */
    struct elem *head;
    lock_t count_lock;  /* used to lock count */
    int count;
} stack;

typedef struct elem {
    struct elem *next;
    /* stuff */
} elem;

void push(stack *s, elem *e) {
      lock(s->head_lock);
      e->next = s->head;
      s->head = e;
      unlock(s->head_lock);

      lock(s->count_lock);
      s->count++;
      unlock(s->count_lock);
}

elem *pop(stack *s) {
      elem *e;
      int acquired;

      for(acquired = 0; !acquired; ) {
              lock(s->count_lock);
              if(s->count) {
                      s->count--;
                      acquired = 1;
              }
              unlock(s->count_lock);
      }
      lock(s->head_lock);
      e = s->head;
      s->head = e->next;
      unlock(s->head_lock);
      return e;
}
```

4. Stupid data structure tricks (25 points)
Fresh from getting .rich, Mr. V. A. Pid hires you on for his next project. His current obsession is to make systems simple by eliminating duplicate functionality.

1. (15 points) He notices that page tables and file system meta data are both used to map integers to integers and decides to replace the meta data on his BSD-style file system with a simple on-disk "direct" page table structure (i.e., each file is mapped by a single, fixed-sized linear array). State whether you would expect the following to get better, worse, or remain the same as compared to using standard Unix style meta data and give a few words of justification.

   (a) The space overhead for small files.

   (b) The space overhead for large files.

   (c) The performance of the Unix command "ls -l."

   (d) Reading the last block in a large file.

   (e) Fragmentation of disk.

2. (10 points) Drunk on power, Mr. P hijacks an ITF meeting and forces this notoriously passive group of people to eliminate IP addresses, and just use DNS-style symbolic names ("stanford.edu", "hotmail.com", etc.). (After all, we always resolve a DNS name to IP address, so why not eliminate the the translation?) In a few sentences, say what changes we need to make to IP packets to make this work, along with a concrete advantage and disadvantage of this approach.

5. Sort of logging (25 points)

The sort-of-logging file system (SOL-FS) implement a file system that supports the standard Unix file operations: create, unlink, link, mkdir, and move. It preserves consistency across crashes by having a single entry "log" that works as follows:

1. Before doing any operation, it writes a description of the operation and its operands to the on-disk log.

2. It then performs the operation, flushes the modified meta data out to disk, and deletes the on-disk log.

For example, to create a new file "a" in directory "/foo" it will write the entry "create /foo/a" to disk, do the meta data updates needed to insert a into foo, flush them to disk, and then delete the log.

After a crash, it checks to see if a log exists and, if so, does the operation it describes (as detailed above) and deletes the log. If the log's entry conflicts with existing state it skips the log action. For our example, if the file /foo/a already exists, it skips the log entry.

1. (15 points) Describe how to implement move in this system. In particular, indicate (1) what meta data has to be modified, (2) what has to be flushed (and what order), and (3) and how to recover after a crash (taking care to handle the case where you crash during recovery).

2. (10 points) What problem does the given scheme pose if we want to extend it to allow the log to hold an arbitrary number of entries?

6 Faking what you don't got (30 points)

V. A. Pid isn't particularly smart, and has brought a segmentation-based VM system (cheap!) for his page-based virtual memory hardware. Fortunately, you notice that the segmentation system requires that (1) a segment's base be aligned to a 1K byte boundary, and (2) the number of bytes in a segment be a multiple of 4K. Since your virtual memory hardware supports 10-bit pages and software TLB exception handling you decide to emulate segmentation on top of paging.

More concretely, the hardware's addresses use the high 22-bits to specify the virtual or physical page number and the low 10-bits to specify the offset within the page:

```
+----------------------------+-----------------+
|   22-bit page number       |    10-bit offset |
+----------------------------+-----------------+
```

While the segmentation scheme assumes the the high 16-bits of a virtual address hold the segment number and the low 16-bits hold the segment offset:

```
+-------------------------+-----------------------+
|   16-bit segment number |   16-bit offset       |
+-------------------------+-----------------------+
```

The segment table is an array of active segment descriptors, where each descriptor has the following fields:

```
struct seg {
        unsigned segnum:16; /* segment number */
        unsigned nbytes:16; /* number of bytes in segment */
        unsigned base:32; /* physical address of segment base */
        unsigned prot:2; /* no access = 0, readable = 1, writeable = 2 */
};
```

Your job is to write a TLB fault handler (in C-ish pseudo code) that has the following signature:

```
/*
 * Inputs:
 *    badva - the faulting virtual address.
 *    write - was the access a memory read or a write?
 *    st    - pointer to the segment table.
 *    ns    - number of segments in segment table.
 *
 * Output is either:
 *   1. The physical page number to insert in the TLB to handle this fault.
 *   2. A call to a routine, 'fatal', on illegal accesses.
 */
unsigned fault_handler(unsigned badva, int write, struct seg *st, int nsegs);
```

You do not have to handle paging. Please state any assumptions you make about how the hardware or segmentation system works. *And please comment your code so that it can be understood!.* You may rip out this page.

7 Sometimes profs aren't real smart (5 bonus points)
State one error or misleading pronouncement on the lecture slides.