Stanford University
Computer Science Department
CS 140 Final
Dawson Engler
Winter 2000

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	45	
2	12	
3	28	
4	15	
5	20	
total	120	
bonus	3	

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 cheating.

Name:

Signature:

1. Short-attention-span questions (45 points)
Answer each of the following questions and, in a sentence or two, say *why* your answer holds. (5 points each).

1. (5 points) Assume we eliminate the use of inodes by putting the information they contain in their associated directory entry. Ignoring the issue of hard links, what is a performance drawback of doing so? (And please give an example Unix command that illustrates this problem.)

   Information becomes more diffuse. 'ls' can now run slower since names are scattered.

2. (5 points) After a long period of no dating, your cs140 partner gets really excited about "signal" and "wait" and wants to use them throughout your nachos code, despite it not being implemented as a monitor (i.e., multiple threads can run "related" code concurrently). What general problem will you probably run into?

   You'll probably run into the "lost wakeup" problem, where one thread decides to sleep on a condition, but before it blocks, another thread signals that condition. This problem doesn't happen in monitors since only one thread at a time can execute monitor code.

3. (5 points) On hardware that uses a TLB and a fixed page size, what feature of the TLB will place a hard upper bound on the amount of physical memory your system can use?

   The number of bits in the physical page number.

4. (5 points) To impress your friend XY you change his file system to always prefetch the next 64K of disk memory on every disk access. He's enthusiastic about the speedup and throws a party. You then try the same trick for your friend XX. She's unimpressed by the negligible improvement and is irritated that you've wasted her afternoon. Who would you expect to be using the more stupid file system? (And why?)

   You'd expect XY's file system to be doing more intelligent block allocation since prefetching helped his performance. Two other acceptable answers: (1) XX's file system is

smarter since it could already be doing prefetching; (2) both could have smart file systems but XX's is full, so couldn't allocate contiguous blocks.

5. (5 points) Ethernets use statistical multiplexing to increase link utilization. Please give an example of where we've used similar ideas to increase resource utilization, and give a reasonable, non-statistical multiplexing alternative.

   For VM we allocate pages based on need (using metrics such as the working set). A non-statistical multiplexing scheme would be to allocate fixed portions of memory to processes. This has the disadvantage that we'd usually be giving them too little or too much memory. Similarly: for CPU, give fixed percentage of CPU.

6. (5 points) You are using a reliable byte stream protocol (such as TCP) that acknowledges the number of bytes it has received so far (e.g., "the last byte received was byte 1023"). For the case where a sender has multiple, unacknowledged packets outstanding what might be a better acknowledgement message for the receiver to send?

   A better ACK would be to say explicitly what messages you've received. That way a lost message in a sequence wouldn't force the later ones to be retransmitted. We also accepted: sending a NACKs back for lost messages, but this can be worse, since you don't always know you've lost something.

7. (5 points) You and I want to agree on a time to eat lunch. I send you an email, saying "let's meet at 1pm." You reply "ok, we'll meet at 1pm." Do you know for sure that we have agreed upon when to go to lunch? If so, say why; if not give a concrete example of failure that could mess things up. (You may assume our mail readers attempt to handle packet loss using techniques discussed in class.)

   You cannot know that we've reached consensus. For example, the link between the two machines could have been severed right before you sent your message. We also accepted the possiblity that I didn't read my mail (though presumably since I'd sent you a message asking a question I would read my mail.)

8. (5 points) Roughly order stack, code, and heap segments as to how well you would expect them to perform under a paged VM system that uses LRU page replacement. (And give the intuition(s) for your ordering.)

The best is stack, since stack access = LIFO and will have dense memory usage, which is perfect for LRU. Code will be next since it is sort-of LIFO (with possible duplicates in the call chain) but will likely be more diffuse than stack. Finally, heap will tend to be random access with more diffuse usage, which will be the worse of the three.

9. (5 points) What virtual memory paging and eviction strategy should you use for virtual address range that exhibits good spatial locality but poor temporal locality? Additionally, please give an realistic example of when such a situation can arise.

   Read-ahead + MRU will work best: spatial locality says we'll be using the subsequent blocks next (making read-ahead profitable), non-temporal locality means that the blocks we've just used won't be touched for a while so can be evicted (MRU). An example is a large vector that you are linearly traversing.

2. One name. (12 points)

Disgruntled with overwork, your 140 partner decides to simplify your nachos implementation by having all processes live in a single address space. Now, instead of having a page table per process, there is one for the entire system, and each process has a protection table that tracks the virtual pages it is allowed to access.

1. (6 points) At a high level sketch how to do linking/loading on this system.

   Our problem is that we cannot know where a process is going to live. The easiest solution is to do load time relocation.

2. (6 points) What correctness challenge does the use of a single address space create for a Unix fork implementation? (I.e., where a new process is created by cloning an existing one.)

   You won't be able create private copies of the data segment, since that part of the address space is in use. You could try relocating the data segment, but that won't work since you don't necessarily know where pointers to the old data are, and can't update them to point to the new location. (You could do this on the fly by marking the old data region as unaccessible, and catching pointer references to it, but we didn't require answers to be so fancy.)

3. Sort-of RAID (28 points)

To help pay Stanford tuition, you buy a cut rate disk for your computer. Unfortunately, you notice that over the course of use one data block per file goes bad.

1. (10 points) Ignoring system crashes, explain how to compensate for a bad data block using error correction ideas from RAID. (Note: you only have a single disk and you should only need a single extra "parity" block per file.) At a high level please state (1) what you must do on a disk write and (2) what you must do when a block goes bad. (You may assume that the disk controller will give you a "bad block" error when you try to read or write a bad block.)

   The basic approach: have an extra parity block per file which is computed by xoring all data blocks together. On a write you have to read the block you're modifying and the parity block (or get them from the cache), subtract it out of the parity block (by xor'ing it), then xor the new block with the parity, and write them both to disk. When a data block goes bad you have to allocate a new block for it's data and recreate it by xor'ing all remaining data blocks with the parity block. Of course if we get a second bad block, we're out of luck; similarly, if meta data goes bad, we won't necessarily know where the parity block is, so won't be able to can be in a bad state as well.

2. (4 points) When modifying cached blocks, you can either recompute the parity block when you evict a block, or when you modify it (and thus still have the unmodified version of the block around). What is the main (significant) advantage of recomputing it on every modification?

   If you do on every modification: easy to subtract out the old block, then xor in the new block. Otherwise, you'll have compute the parity block by xor'ing all the data blocks with the new copy. (Also: for reliability, if we recompute parity later, then we cannot recover from an error.)

3. (10 points) Assume we do not ignore crashes, but do ignore the problem of blocks going bad during the period of crash to the end of recovery. (I.e., the set of bad blocks right before a crash is exactly equal to the set of bad blocks after recovery.) What order should you write out disk blocks to eliminate the chance that your file system will be in an unrecoverable state? (And why?) What does an "fsck" file system recovery program now have to do on after a crash?

   Either order can be made to work, the only requirement is that you are consistent. If you write out the disk block first, then parity, crash recovery consists of fsck xoring all data blocks together to recompute parity (for every file). If you write out the parity first, then you can always recreate it by xoring everything together (although this means that the data block will be the "old" unmodified version, which is acceptable.) Note that if you write out parity first you cannot reliably recreate the new data block since you don't know which one it was, so don't know which subset of data blocks to xor with the parity block.

4. (4 points) Assume blocks can go bad during the period of crash to recovery: what problem does this pose?

   If a block goes bad for a partially written file, you won't be able to recover since there will be two blocks out of synch with the parity block: the one you were writing, and the one that went bad. For example, if you were writing parity first, the data block used to compute parity is different from its on disk version, which means that we cannot xor it with the parity block to recreate the value of the bad block. If a data block goes bad for a fully written file, we'll run into trouble, since fsck recomputes the parity block for every file and won't be able to disambiguate this case from that of a partially written file. (We could go ahead and try to recover and hope for the best, which is not unreasonable.)

4. Stacks-R-Fun (15 points)

(10 points) Assume you are building a user-level (*not* kernel-level) threads system that manages multiple threads in the same address space. Each thread has their own stack. For ease of use, you want your system to transparently grow thread stacks dynamically. Please give a high level sketch of what virtual memory support the OS should provide so that you can (1) detect when a thread's stack has been exceeded and (2) grow the thread's stack. (And, of course, how you'd use this functionality to do these two acts.) Additionally, please briefly discuss the tradeoffs around where you decide to place thread stacks in virtual memory.

Basic idea: Mark the page (or pages) after each thread stack as unaccessible. Any reference will give a segmentation fault, which the threads package must be able to catch. Then it either remaps that memory as accessible and the next page as inaccessible or, if there is no space left, moves the stack to a place that fits. You can either allocate large chunks space using malloc or have some way of requesting ranges from the OS (such as "mmap"). The main tradeoff: you'd like to allocate stacks far apart to eliminate the chance that an overflow in one is a valid reference in another, plus you'll have to relocate less often, but this will make the address space less dense, which will likely degrade performance.

(5 points) What do you have to be prepared to do to dynamically grow a thread's stack? What complication does taking the address of a stack variable create?

You may have to copy the stack to another location to find space. At this point, if the program has stored a pointer the stack, this value will be invalid (since it points to the old location). To fix this you'd have to be able to find and then update pointers.

5. Weird concurrency (20 points)

Given a list of runnable threads, the sort-of TERA machine executes them round-robin by running one statement in the first thread, always immediately context switching to the next thread, running one statement in it, etc.

You have been presented with the following two "push" implementations that operate on a single shared stack. The code was intended to run on a TERA machine that supports a maximum of two threads: state whether it is free of race conditions (and provide a succinct intuition for why) or is broken (and if so, give a specific execution sequence that will cause it to fail).

Note, the lack of a "pop" implementation is not accidental: you may ignore it and any possible interference with it. Additionally, to simplify reasoning, each statement in the implementation has a number preceding it: *the machine will run this statement in its entirety, then always(!) context switch to the next thread, run a statement in it, etc.*

1. **(10 points)** The first (partial) stack implementation is linked list based.

```
struct elem {
        struct elem *next;
        /* ..stuff.. */
};

/* stack head */
struct elem *head;

void push(struct elem *e) {
          struct elem *old;

/* 1 */   old = head;
/* 2 */   e->next = head;
/* 3 */   head = e;
/* 4 */   if(head != e) {
/* 5 */        head = e;
/* 6 */        e->next = old;
          }
}
```

Assume they start at the same time. A bad sequence that lose's T2's update: T1:1, T2:1, T1:2, T2:2, T1:3, T2:3, T1:4, T2:4, T1:5, T2:done, T1:6 (e-¿next now points to 'old', which is the original head, which means you lose T2's update).

2. **(10 points)** The second partial stack implementation holds elements in an "infinite" array.

```
elem stack[]; /* an "infinite" stack */
int n = 0;    /* position of last stack element */

void push(struct elem *e) {
          int i;

/* 1 */    i = n;
/* 2 */    n = n + 1;
/* 3 */    stack[i] = e;
/* 4 */    if(stack[i] != e)
/* 5 */         stack[i+1] = e;
}
```

Assuming we always push unique elements, this code works correctly. The only possible race condition: two concurrent updates, which will cause the first update to be lost (i.e., it assigns 'e' to the stack, T2 comes in and overwrites it). T1 will catch this case (from statement 4), and patch in it's element appropriately. This can result in not strict LIFO, which is acceptable (intuition: if 'push' was wrapped in a lock, whoever grabs the lock first will get to push their element — this is not necessarily the first caller of push.)

6 Sometimes profs aren't real smart (3 bonus points)

State one error or misleading pronouncement on the lecture slides.