Stanford University
Computer Science Department
CS 140 Final
Dawson Engler
Autumn 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.

NOTE: We will take off points if a correct answer also includes incorrect or irrelevant information. (I.e., don't put in everything you know in hopes of saying the correct buzz word.)

Question	Points	Score
1	30	
2	12	
3	12	
4	16	
5	18	
6	16	
7	16	
total	120	

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 (30 points)
Answer each of the following questions and, in a sentence or two, say *why* your answer holds. (5 points each).

1. Assume we have n threads at different priority levels and that they all use a lock l, which schedules waiting threads in FIFO order. Describe a plausible steady state behavior of this system.

   One scenario: we will lose all parallelism. The lowest priority thread grabs l and gets preempted or blocked. Then the system will run all other threads until they block on the queue. It will then run the low priority thread, which will release the lock, run until it reacquires the lock and then get placed on the queue. This will repeat for all other threads on the queue.

2. Assume every file records the last two times it was used, and whether these uses were sequential. How could you use this information to control caching?

   Close together = cache, far apart = preferentially flush.

3. It's the year 2010: CPU's do trillions of instructions a second, disks are petabytes in size, and disk access times are still (surprise) about 10ms. Your old cs140 partner still stalks you. Their latest scheme is exploit the fact that CPU instructions are essentially free to compress all on-disk data, despite the fact that disks are not, typically, anywhere near full. What is a likely reason that read workloads could actually see much better performance in this scheme?

   We can pack more data close together, which allows us to grab more data quickly. E.g., if we can fit 2x more data in a track, we are happy since we can read without a seek.

4. Unsurprisingly, it's common to have multiple connections open between the same two machines. Give a plausibly-useful optimization that could be done to the reliable transmission protocol we talked in class about to exploit this situation.

   We could potentially merge ACKs and messages across connections, or also share window space for flow control.

5. Assume we have an extent-based FS (such as the Linux ext2 FS) that tracks extents of file data using a block pointer and a block count. What is a bad file access pattern for this system compared to a FS that just uses simple block pointers?

   Reading bytes randomly will be bad. To find out what block holds byte b in a file we have to traverse all previous extents adding up the block counts.

6. Assume our TLB can map arbitrary-sized, contiguous byte ranges. Explain how a paged-VM system and a malloc implementation could cooperate to eliminate heap fragmentation. Is this a good idea?

   malloc/free run into fragmentation because since they cannot find pointers to allocated data, they cannot move the data once it has been allocated. The above TLB would be used to do such relocation using aliasing.

2. Matrix fun (12 points)
You are given a bunch of scientific code that contains loops of the form:

```
for(i = 0; i < n; i++)
        for(j = 0; j < n; j++)
                a[j][i] += b[j][i];
```

The matrices a and b are allocated using code such as:

```
a = malloc(n * sizeof *a);
for(i = 0; i < n; i++)
        a[i] = malloc(n * sizeof a[i]);
```

You notice that the addition loop trashes the TLB much more than it should. Say why, and give a simple fix. In general, what do you expect to happen to paging performance?
The inner addition loop will stride through memory in n sized pieces, which will really hurt performance. E.g., if $n >= page - size$ then every access will touch a different page. The fix is to just swap the loops. You'd expect this to also help paging performance.

- *-2 got it right, but also said something wrong (told you)*
- *-3 didn't comment on paging performance/got that wrong*
- *-5 didn't recognize the problem of the stride, but did something to improve spatial locality*
- *-5 wrong fix*
- *-8 didn't recognize the problem of the stride and fix wouldn't improve spatial locality*
- *-10 mentioned something that could improve paging when n is large*

3. Stupid scheduling tricks (12 points)

Your partner hacks up a proportional share scheduler that attempts to allocate each process a specific share of the processor. It works as follows:

- A newly arriving process is assigned the minimum of all priorities on the system, or 0 if there are no other processes.
- Every time a process runs, its priority is incremented by $priority = priority + 1/weight$. (Note, you may ignore roundoff error.)
- At each scheduling point the system runs the smallest priority process. It never leaves the CPU idle if there is a runnable process.

For example, assume there are no processes on the system, and we start two processes P1 and P2 with weights 10 and 1 respectively (intending to give P1 10x more of the CPU than P2). The scheduler will run P1, give it a priority of 1/10, run P2, and give it a priority of 1. P1 will then be run 9 more times until it also has a priority of 1, then P2 will run once, etc.

Assume we run the job mix above for "a while" and then introduce a third process. Will the above scheduler give processes the correct proportions on a two-processor system? Please either state your intuition, or give a small, concrete example where it breaks. (Note, it's ok if the scheduler does not preserve ratios when there is a surplus — i.e., running P1 and P2 on the system above will give them both 100% of a CPU, but will not penalize P1 since it cannot use more than 100% of the CPU.)

The scheduling algorith breaks badly. Assume we run P1 and P2 for 1000 time slices. At the end, P1 will have a priority of 100 *(*1000/10*), and P2 will have a priority of 1000 (*1000/1*). If we add P3, we'll give it a priority of* 100*. Then P1 and P3 will run constantly until P3 gets a priority of* 1000 *(900 more slices). Effectively starving P2.*

Points:

- *-1 minor mistake*
- *-6 got a little bit of the explanation right*
- *-6 said that at steady state scheduler would work as intended*
- *-8 wrong explanation*
- *-8 the intuition is sort of right*

4. Something for nothing. (16 points)
Assume you have a set of tasks that must read every block of data on disk (e.g., a virus scan or backup) but are low priority compared to normal processes. Assume further that your disk's seek time and rotational delay are roughly similar in cost.

Part A (10 points) Explain how and when you can interleave background requests without hurting the performance of normal disk writes.

Assume we have a normal read of sector s which requires seeking to track t and waiting for the sector to rotate underneath the disk head. The disk can be used for other work during the rotation delay, which allows us to potentially read sector s' and still get the head to s in the same amount of time.

Points:

- *-4: If you missed the fact that you can get work done because of the rotational delay.*
- *-2 to -6 if your algorith hurt performance or was mostly wrong.*

Part B (6 points) To do this interleaving, you will need the disk driver to tell you how much various accesses cost. Give the C prototype of the function you would like, and describe what it does. Note, that the driver only has a limited view of the universe (i.e., the disk drive) so this function must be something realistic for it to implement.

```
/* assuming the disk head is at s0, how many cycles will
   it take to get to s1?   Assume s0 of -1 means from the
   current disk head position. */
int access(int s0, int s1);
```

From above example: if access(-1, s) == (access(-1, s') + access(s', s)) *we can do the read. If there are multiple sectors, we would add in these access times. -2 points if you didn't talk about reading from current position, -3 if didn't talk about how to use it.*

5. Redundancy (18 points)
You notice that many blocks on the same system have the same value. (E.g., there are many copies of nachos running around, many slightly-edited files have the same prefix, etc.) To exploit this you modify a Vanilla Unix file system to share blocks across files by adding a reference counter to the start of every disk block.

Part A (14 points) Assume you use write ordering to guarantee file system consistency. In a few sentences, say how to order writes for block deallocation and what you will do if the system crashes. Be careful that there are no non-recoverable failures in your scheme. (Hint: consider what could happen if someone allocates a block you just deallocated.)

Simplest approach (this problem degenerated into something we covered in lecture): (1) persistently nullify pointer, (2) decrement reference count, (3) if refcnt == 0, put on free list and write back. Note that (2) could be omitted if you knew refcnt was going to zero. Recovery consisted of doing a mark and sweep that calculated how many pointers there were to each block and used this to (1) put blocks on free list and (2) patch reference counts.

Points:

- *-4 if you reversed (1) and (2) (this leads to a potential race condition involving two processes decrementing a block's count and another process allocating the same block.*
- *-3 if no discussion of freelist*
- *-3-6 if no good discussion of recovery.*

Part B (4 points) Assume you implement your scheme with minimal overhead. When would you expect it to have poor read performance?
It will have bad read performance for files that are interlaced with shared blocks, since you'll have to wander over disk picking them up.

6. Concurrency fun (16 points)

Part A. (8 points) Assume that every word-sized memory location has an associated version number, which is incremented on every write. Given the function version to access this value

```
int version(int *p);
```

is the following a correct implementation of our usual spin lock?

```
int lock(int *l) {
        while(1) {
                int old = version(l);
                if(!*l) {
                        *l = 1;
                        if((old+1) == version(l))
                              break;
                }
        }
}
void unlock(int *l) {
        *l = 0;
}
```

If so, give a short intuition, if not, give a counter example.

Part B (8 points) Assume we have a critical section that only writes to a shared variable, but does not read any shared state:

```
shared int x; /* shared variable */
lock_t x_l; /* lock for x */

void foo(int y) {
          lock(x_l);
          x = y;
          unlock(x_l);
}
```

What is the intuition for why we can eliminate locking in foo? Is this also true for critical sections that have writes to multiple pieces of shared state?

7. The essence of naming. (16 points)

A "naming system" maps names to values (which in turn can be other names). Many of our abstractions do some amount of naming. For example, VM maps virtual addresses (names) to physical addresses (values); a Unix FS uses meta data to map file offsets and file names to disk blocks. Using these systems and others as a basis, discuss three issues common to different naming systems, giving several examples to make your case. Possible properties include: what influences the structure used to do this mapping, what happens when names cannot be resolved, whether names can be synonyms, if the space of names is flat, etc. Please discuss at least one property not on this list. Feel free to use short bulleted lists rather than full sentences. Your answer should easily fit on this page.

We mostly docked points if (1) you didn't have much discussion of tradeoffs or (2) your examples had incorrect statements.