Systems Software Comprehensive Exam

Fall 2005

Solutions by 2006 First Years

1. Write the shortest non-reentrant legal C function you can and why you think it is non-reentrant.

   Solution:

```
int foo() {
     static int x;
     x++;
     return x;
}
```

   This function is not reentrant because it modifies a global (static) variable before returning it. Calling it at different times from different threads will alter the variable x and return different results.

2. Your machine can only do 32-bit loads and stores. What is the locking-related problem in the following code? How would you fix it?

```
struct foo {
     lock_t a_lock; // always held before touching a
     lock_t b_lock; // always held before touching b

     char a;
     char b;
};
...

void update_ab(struct foo *f) {
     lock(f->a_lock);
     f->a++;
     unlock(f->a_lock);

     lock(f->b_lock);
     f->b++;
     unlock(f->b-lock);
}
```

   Solution: The fields `a` and `b` in `struct foo` are both `char` variables that are 1-byte in length, so the compiler will place them together within one 4-byte (32-bit) chunk in memory. Because we can only read/write 32-bits at a time, when we do `f->a++`, we actually have to read the 32-bit chunk containing both the values of `a` and `b`,

modify the 1 byte corresponding to `a` by incrementing it, and then write back the entire 32-bit chunk. If another thread attempts to modify `f->b` and that operation finishes before the original thread's call to `f->a++`, then when the new value of `a` is written back to memory, the ORIGINAL value of `b` is also written back, thus clobbering the new `b` value. This problem can be solved by adding padding in the `struct` to ensure that `a` and `b` are in separate 32-bit chunks. e.g.,

```
struct foo {
     lock_t a_lock;
     lock_t b_lock;
     char a;
     char padding[3];
     char b;
}
```

3. Your threaded code has no race conditions. It does have this routine:

```
foo() {
     lock(a);
     lock(b);
     ...
     unlock(a);
     ...
     lock(a);
     ...
     unlock(a);
     unlock(b);
}
```

Ignore performance: what is wrong here?

Solution: This code can deadlock as follows.

Thread 1:	Thread 2:
lock(a)	-
lock(b)	-
unlock(a)	-
	lock(a)
lock(a) WAITING	-
-	lock(b) WAITING

Deadlock!

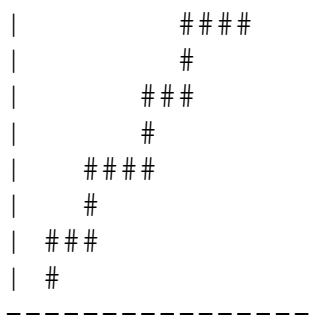
4. In what way is a good proportional-share process scheduling algorithm essentially equivalent to a good graphics line-drawing algorithm? (Use a picture in your answer and label it.)

Solution: In a proportional-share process scheduling algorithm (otherwise known as a lottery system), each process has a priority specified by a percent, where all

currently-active processes have percents that add up to 100%. Each process gets a number of tickets proportional to its priority (say tickets numbered between 1 to 100, so if a process has 20% priority, it might receive 20 tickets). During each scheduling quantum, the scheduler picks a random number, and whichever process owns the ticket for the picked number gets to run. Thus, on average, processes with more tickets (higher priorities) run more than those with less tickets (lower priorities).

In a graphics line-drawing algorithm, the challenge is to draw a continuous line (in x dimensions, let's say 2 for simplicity) by drawing discrete pixels in a pattern such that when one steps back, the pixels form a line with a particular slope.

What the two have in common is taking a continuous process and mapping it into the discrete domain.



Imagine that there are only 2 processes. On one axis is the amount of time that one process runs and on the other axis is the amount of time that the other process runs. Given enough samples (and a really long run), this will look like a STRAIGHT LINE with slope determined by the relative priorities of the 2 processes (a slope of 1 if each process has 50% chance of running at any given scheduling quantum). However, the scheduling world is discrete, so there will be chunks of time (demonstrated by the '#' in my pathetic ASCII art) where one process will run and then the other process (or maybe the same process) will run. This sort of draws a line, albeit stochastically.

5. You run program A using kernel threads, and then re-run it using user-level threads. How could these two runs behave differently with respect to load and store instructions?

Solution: If you're running your program on one processor, then all the loads/stores across different threads are guaranteed to be atomic/sequential. However, kernel threads can run simultaneously on multiple processors, so loads/stores do not have this sequential consistency guarantees. User-level threads can only run on one processor, so you still have the sequential consistency guarantees.

Another answer we came up with: Loads/stores can can cause page faults, so with kernel threads, the kernel knows about page faults and only blocks the 1 thread with the faulting instruction, but for user-level threads, if one thread blocks on a page fault, the entire process blocks because from the kernel's point-of-view, there is only one thread.

6. Draw the structure of a standard 32-bit virtual address with 4K pages. What bad things happen if you switch the order of the two components?

   Solution:
   [20 bits for virtual page number (VPN)][12 bits for page offset]
   Backwards:
   [12 bits for page offset][20 bits for virtual page number (VPN)]

   The reason why the least significant bits are assigned to the page offset is that you often access nearby bytes together (spatial locality), so you want those to map to the same page. If you get the order backwards and instead assign the least significant bits to the VPN, then you lose spatial locality (locality of reference) and you get horrendous performance because every time you access nearby bytes in memory, you end up accessing completely different pages, thus killing your TLB hit rate and causing other slowdowns.

7. Your (ancient) machine has an 8K, direct mapped, physical cache with 4K pages. Program A is using an 8K, page-aligned array, whose first page maps to physical page 31. Give the set of physical pages that the second page of the array should be mapped to and why.

   Solution: Any even-numbered page would work.
   There are two parts to this problem:

   - First, you have an 8K direct-mapped cache reference by physical addresses. 8K = 2^{13}, so here's how 32-bit addresses map to the cache:
   `[ 19 bits (cache tag) ][ 13 bits (cache index) ]`

   - Second, you have a virtual memory system with 4K pages, so here's how 32-bit addresses map to the virtual memory (recall that $4K=2^{12}$)
   `[ 20 bits (page number) ][ 12 bits (page offset) ]`

   You have an 8K, page-aligned array whose first page maps to physical page 31. This first page contains the first half of the array, and the second half of the array fits entirely on some other page. So where does this array reside in physical memory? Let's dissect the page number 31:

   This is the address range of the first half of the array:

```
[ 20 bits (page number) ][ 12 bits (page offset) ]
0000 0000 0000 0001 1111  0000 0000 0000
0000 0000 0000 0001 1111  FFFF FFFF FFFF
```

Now where does this array fit into our 8K direct-mapped cache? Well, let's see:

```
[ 19 bits (cache tag)  ][ 13 bits (cache index) ]
0000 0000 0000 0001 111 1 0000 0000 0000
0000 0000 0000 0001 111 1 FFFF FFFF FFFF
```

Notice that the cache tag is `0b1111`, and the cache indices range from: `1 0000 0000 0000` to `1 FFFF FFFF FFFF`. This is the 4K that comprises the upper-half of the 8K cache. Now, in order to maximize cache hits, we want to put the second-half of the array in the LOWER-HALF of the 8K cache. How can we do that? By ensuring that the MSB (most significant bit) of the cache index is a 0. What does that mean in terms of the physical page number? It must end in 0.

```
Cache:  [ 19 bits (cache tag) ][ 13 bits (cache index) ]
        ???? ???? ???? ???? ??? 0 0000 0000 0000
        ???? ???? ???? ???? ??? 0 FFFF FFFF FFFF
VM:     [ 20 bits (page number)  ][12 bits (page offset)]
```

It doesn't matter what the rest of the physical page number is, as long as it ends in a 0, which means that it must be EVEN.

8. What address space layout will be the best for a linear page table as compared to a hashed page table and vice versa? (Make sure to say why.)

   Solution: Linear page table is better for densely-filled memory because a hashed page table will have lots of collisions with a densely-filled memory. Hashed page table is good for sparsely-filled memory because a linear page table will waste more space with a sparsely-filled memory.

9. Give the simplest example of a chunk of data and a predictable access pattern that LRU will perform optimally bad on. In the absence of prefetching, what is the best *realistic* algorithm to use?

   Solution: LRU is terrible whenever you have a working set that is 1 larger than the size of your cache, and you're accessing the elements repeatedly in a loop.

   Example, 3 addresses (0, 1, 2) and size-2 cache:
   0, 1, 2, 0, 1, 2, 0, 1, 2, 0, 1, 2, etc... Every single access is a cache miss!

   Random replacement is the best realistic algorithm to use.

10. Joebob, your mortal enemy, gives you a USB stick that you want to mount as a file system on your computer. Give the type of checks that the file system should do before treating the USB data as a valid file system.

    Solution: You should check that the FS is well-formed - e.g., no cycles in directory structure, the . and .. entries point to right places, link counts are correct, etc.

11. You create a file on a Unix file system (such as FFS). Roughly: what meta data do you modify, what errors could you get, what order do you write the metadata out in?

    Solution: Metadata modified: inodes, inode reference count, timestamp, pointers to data blocks, directory entry

    Possible errors: out of disk space, out of inodes, permission errors, hardware errors

    Order to write metadata out to disk:
    1.) Create and initialize data blocks
    2.) Allocate and initialize inode to point to data blocks
    3.) Add the filename to the directory entry and point it to the appropriate inode

    You want to do things in this order because if you crash sometime in the middle, you won't be left with a pointer to garbage. General rule: always initialize something before setting a pointer to it.

12. Among NFS's operations are:

```
//write nbytes from buf into offset of the file named by fh
write(fh, offset, buf, nbytes)

//make directory "name" in directory named by dh
mkdir(dh, name)
```

    It sends these requests across a lossy network, so may obviously have to retransmit them. Discuss: what problems could occur because of retransmission for these two operations and a (partial) fix.

    Solution: If you do 2 successive writes of different contents to the same file, the two write commands could arrive out-of-order if the network is slow and retransimission is necessary. This could be a problem if, say, you did write(file, "FOO") and then write(file, "BAR"). After the second write, you might expect that the file contains "BAR", but if they arrive out-of-order, then the "FOO" write might come after the "BAR" write. Similar problem with mkdir.

    One partial fix is to put some sort of sequence numbers to ensure in-order delivery.

    (We're not sure whether NFS guarantees in-order delivery even in the presence of

retramissions, etc.)

13. You have a distributed file system. What is the perfect guarantee it could give for cache consistency? What would you have to do to implement this? Is there any way an application running on top of this perfect consistency could see stale data?

    Solution: Perfect cache consistency guarantee is called 'write-to-read' consistency, which means that when one client reads a file, it should have the latest contents of the last thing that anybody else ever wrote to the file. You would need to lock files a lot in order to implement this and update all local clients' caches whenever there is a write to a file by one client.

    We were unsure about if there was any way that an app running on top of perfect consistency could see stale data:

    Yes - If the network is slow, then the clients' cache updates might not happen fast enough so that clients can still see stale data.

    No - by definition, it's perfect.

14. Many distributed file systems implement close-to-open consistency. Give an intuitive statement of what this is, and two reasons you might prefer it over perfect consistency (including one non-performance reason).

    Solution: Close-to-open consistency means that whenever a client opens a file, he is guaranteed to at least see the data on the file at the time when it was last closed. This is a weaker guarantee than 'write-to-read' consistency because another client may have that same file opened and be writing to it.

    Advantages:
    Performance - You no longer have to invalidate/update everyone's cache whenever you do a write or use that many locks, etc. You just need to do updates when you close a file, which is much less frequent.

    Robustness - If you're writing to a file and then crash or have your data corrupted without closing the file, other clients don't see your corrupted version. (This isn't a great reason, so somebody could probably think of a better one ...)

15. Let's say you have a network interrupt handler that looks something like:

```
interrupt_handler() {
     while(there are packets to receive)
          pull pkt off network interface
          enqueue(pkt);
     while(there are packets to transmit)
```

```
            dequeue from transmit queue
            give to network interface
}
```

You notice that sometimes no packets come out of the system for awhile, in situations where they should. Similarly, you notice that sometimes applications do not run, but should. What problems in this code could cause this behavior? Give a sketch of how to fix it.

Solution: No packets coming out of system for a while, when there are packets to transmit - This is a case of starvation and can occur whenever there are a constant stream of packets to receive. The interrupt handler keeps on handling received packets and never gets to send packets.

Applications don't run - The applications can suffer from starvation when all the time is spent in the interrupt handler. This can occur whenever there is a constant stream of packets being sent or received.

The problem is that the interrupt handler does not give up control as long as there are packets to receive and/or transmit, and there is no way to interrupt the interrupt handler (because interrupts are run with interrupts disabled).

One solution might be to modify the code so that 2 things happen:
1.) The transmit loop has a chance of running even if lots of packets are being received
2.) The application has a chance of running even if lots of packets are being sent or received

One possibility is a lottery system where, in the interrupt handler, the receive handler has 1/3 chance of running, the transmit handler has 1/3 chance of running, and the application itself has 1/3 chance of running. Then as the program progresses, dynamically change those percentages to adapt to the rate at which the different handlers and application execute, providing a kind of negative feedback. A simpler scheme would be to put a limit of how many packets to process in a particular time range.