CS4410: Homework 1. Due 9am ~~Feb. 2~~ Feb 3 via CMS

Q1. Suppose that you plug a new disk into your laptop and boot the machine. But when you try to access the disk, the whole machine freezes, including the power button – the only way to restart it is to physically remove the battery, after which it finally comes back to life. Explain in one or two lines if each of these is a possible culprit:

1. A hardware problem with the disk controller.

A malfunctioning disk controller could, in effect, jam the computer's bus. The bus connects the CPU to main memory, the special flash memory where the BIOS lives, and with devices… if the bus is jammed, the entire system will freeze up because nothing, not even the BIOS, can be executed.

2. A bug in the file system browser, which runs in user-mode.

This is much less likely because the O/S kernel protects itself against anything a user-mode application could possibly do. For the machine to wedge up so firmly, whatever went wrong was much worse than a simple user-mode software bug.

Q2. You are using FireFox. Everything was working properly yesterday, but then your machine crashed. Now FireFox hangs when you run it. With "process explorer" you can see that it is rapidly issuing file I/O operations (very rapidly: about 10,000 per second!). The only open file is called "firefox.cnf." This file exists, and can definitely be accessed by the application. On the other hand, the file looks a bit strange (it is a small text file, the line of which seems to be truncated).

3. What would be the likely consequence of changing the scheduler priority of FireFox to the maximum value supported by the operating system? (You may assume that this would be higher than the priority of any other process, including the command interface).

If FireFox is in an infinite loop (and apparently it is), that would be a terrible idea: the system would hang and become completely unresponsive. You would then have no option except to kill FireFox with alt-F4 (^C if you use Linux), or to hold the power-key down until the BIOS notices and restarts the machine.

4. Presumably, the file was damaged by the crash and this is the root cause of the problem. But what's your best guess as to why FireFox is performing so many system calls

It seems likely that FireFox has gotten stuck somehow trying to read the last line of that file, looking for the newline but not checking for EOF. This is a good example of how people who write O/S services need to be extra careful in ways that normal application designers might not.

Here's an example of how you can easily make this mistake. This code fragment, written in C, would behave precisely as described (getchar() calls the read() system call, which would account for those I/O requests):

```
readline(string_buffer)
char string_buffer[];
{
        char *sp, c;
        sp = string_buffer;
```

```
        while((c = getchar()) != '\n')
                if(isalnum(c) || c == '=' || c == ';')
                        *sp++ = c;
        *sp = 0;
}
```

The code is "trying" to read one line of text into string_buffer, but is only keeping alpha-numeric characters, equal signs, and semicolons. Programs like FireFox do this all the time, to clean up pretty-printing in their input file. The bug here is that this particular code snippet doesn't check for EOF, which comes back from getchar() as a null (0) byte. So if a line doesn't end with a newline character, it loops forever, and getchar() keeps calling the read system call, explaining the high rate of I/O requests. Notice that the program won't run off the end of the string buffer because a null doesn't satisfy the if condition.

*This code illustrates a second kind of common mistake, too: it doesn't check to see if the string is too long – it has no way to know whether *sp has run off the end of the string buffer. We can't even tell how long the string buffer argument is allowed to be! Probably our truncated file won't trigger this bug, but imagine a file with a really long (screwed up) last line. The program will just copy and copy, overwriting the stuff beyond the end of the string buffer argument, until it sees a newline character. You are probably guessing that this would cause the application to crash… but it isn't so simple.*

Very often, the string buffer argument is just a local variable allocated by the caller of readline, and this means it might live on the stack… right next to the return PC address that the caller of readline() will use when it finishes looking at the string readline() copied in. A virus could potentially take advantage of this kind of mistake to trick such a program into copying the virus instructions into memory – the virus writer would study the application, realize that a bad argument string can be used to overwrite the stack and the return PC address. They would design virus code that can live on the stack and pick a return PC address that will jump right into the virus code at the "main" procedure of their virus. In effect, like an alien that takes over the body of an innocent actor, the virus is able to fool the service into loading its code and launching it! This is called a "buffer overrun exploit" and is surprisingly easy to do if people write sloppy code like the fragment shown, and then give out the source code. Most computer viruses spread through these sorts of tricks.

Sure, it wouldn't be easy (the virus writer needs to be sure he can guess what the stack will look like, and where it will be in memory). But believe it or not, many programs are so predictable that you can actually study them with a debugging tool and figure things like this out! Virus writers take that extra time and this is why they are so good at building viruses that can spread and cause such havoc. Of course if everyone took cs4410, they would write code that simply checks for EOF and makes sure to copy incoming strings within the allocated space, and that would completely fix the problem here.

Q3. FireFox is a "multi-threaded" program.

5. Is the number of threads FireFox can create limited by the number of cores on your computer?

Nope. Threads are a logical, not a physical thing. You can have as many threads as you like, provided that there is enough memory for their stack areas.

6. Describe a situation in which, if it were up to you, you would have coded FireFox to create an additional thread. What would be the benefit of doing so?

Let's imagine a new feature whereby FireFox would watch the web pages on the "favorites" list and change the shortcut color to red when one of them gets updated (signaling me that I should maybe check it out to see what changed). You would want this work to happen in a new, extra, thread so that it wouldn't disrupt normal web surfing.

7. FireFox is an Open Source application. You download the source code and, just for fun, decide to add a new feature. Your new code tries to change the color of a button on the screen from green to red. It compiles, but when tested, throws the following exception: "Windows control can only be updated by the thread that created it." (NB: a button is a "Windows control.") Make a guess as to why the Windows O/S imposes this restriction.

Presumably the Windows system was having too many problems associated with concurrent access to controls and added this rule as a clumsy form of mutual exclusion lock. The main insight is just that a control is like any other kind of shared object and concurrent access to it (even to change the color) can result in crashes or other problems.

8. Suppose that thread A creates a button, and thread B later needs to recolor it. How can thread B do so, in light of the restriction mentioned in (3)?

The quick and dirty solution, which works only if you are certain your code is safe, is to just disable this exception by setting the value of the CheckForIllegalCrossThreadCalls property to false.

But there is actually a .NET help page on this specific topic (title is "How to: Make Thread-Safe Calls to Windows Forms Controls"). It shows a few ways to solve this problem. The one they recommend works by having thread B ask the control (the button) to perform an action <u>while running within its owner's thread</u>. Operations such as the control-recolor request need to be coded as procedures that can be accessed from a call-back. When thread A next runs, as soon as it wakes up, it will do the callback. The web page for that help topic shows a code snippet that you can cut and paste for this purpose. The specific way it works involves a built-in method called obj.invoke, built into the C# object system. In our case, obj would be the button control object. Obj.invoke is always safe to call. The messy part of the example is the stuff involved in setting up the callback itself (called a "delegate" in C#). But the bottom line is that you can cut-and-paste the magic ritual and it lets thread B ask thread A (or whatever thread really owns the object) to perform the recolor operation.

Q4. Linux and Windows both have system calls for sharing memory between distinct processes.

9. Why is a system call needed for this purpose?

When a process is created, it has its own memory region and other processes can't see the contents. If you do want to share memory, you need to ask the O/S to help out.

10. Does the same issue arise when two threads share memory, in a single process?

No: threads created by a single process live in that process' address space. They can see the same memory (global variables, for example, or data allocated within the heap).