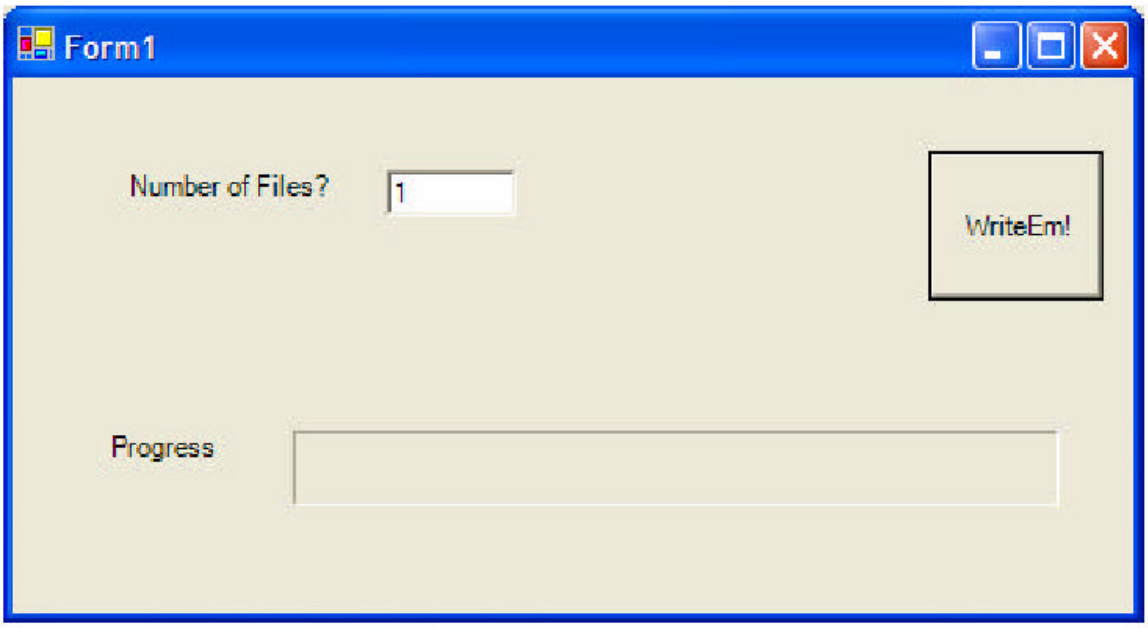
Our Last Programming Assignment (CS414, HW6. Due November 18)

The purpose of this assignment is to do some file-I/O within a program, and to measure the performance of a multi- threaded application. Along the way, we'll also enlarge our experience with some of the basic Windows controls available in C#/Visual Studio.

There are two steps to this programming assignment.

1) First, build a small application that we'll call FileWriter. The program should display a screen like the one shown below:



As seen here, your program will have a textbox into which a user can put an integer, a button the user can press, and a progress bar to show status of the execution. While executing, the "WriteEm" button should turn red and the progress bar should show progress; when finished the screen should go back to the way we see it above.

What the program should do when the button is pressed is to create NFiles (whatever number is in the text box) files containing random numbers written out 100 bytes at a time. Use the File.Open() system call to create a new Stream for each of these files (use filemode FileMode.Create); named DataFile1.dat through DataFileN.dat, (e.g. 1 to 4) and then loop, filling a 100- byte buffer with 100 random numbers in the range 0 to 127 and then writing the buffer contents to the file. The total (over all files) should be 2^17 records -- around 13MB in size on the disk. Thus with 2 files, each would be 2^16 records in size, etc.

Mystery for you to solve: If you don't do anything special, where will these files be created? Ideally, your program should just put them in the "My Documents" folder.

To change the color of a button, you'll want to make note of the old color and then set the backgroundcolor property to the new color. Later, set it back to the old value so that the color red is a sort of visual signal that the program is still writing. As for the progress bar, once you set the minimum and maximum value, changing the "Value" property will automatically refresh the bar; setting the Value to zero clears it. (Don't do this too often or the program may run slowly; most people try to arrange to update a progress bar about 20 times, in total, while a program like this is running).

In your code, be careful to check for exceptions. For example, when parsing the input integer, check for and catch integer parsing exceptions. File I/O can also trigger exceptions. Good code *always* checks for exceptions. Handle exceptions appropriately, for example using MessageBox.Show() to display a suitable complaint if an integer can't be parsed properly.

2) OK, by now you're a Windows Wizard, or at least a Wizard- in-training! Next, we're going to write a second program to sort files. Once you have your program working, you'll compile it with "optimization" (up to now you've been working in debugging mode, but that creates a slower program), then measure its performance as we vary a few parameters.

Our specific goal is to understand how threading can be used when doing file I/O or some other kind of I/O in which your application needs to wait for the I/O to be completed. As it happens, Windows file I/O is a bit too clever for our purposes, so the specific mechanism we're use below tosses a wrench into the mix to force the Windows I/O operations to run slower than would otherwise be the case[1]. Look at how the performance of a file sorting program changes as a function of how many threads are involved. For example, we could sort a single file using one thread, ten threads, or hundreds of threads. And the same can be done when sorting multiple files. The benefit of doing this would be visible mostly if the program manages to keep the machine more busy. But there are also disadvantages: threading overhead, scheduling overhead, and possibly reduced buffer hit rates in the file system buffer pool. Are threads useful for file I/O? Your job is to find out!

We'll be sorting the files *in place*. This entails lots of file I/O: reading records from the file, then writing them back out. Do not create temporary files.

---

[1] For those who are curious: no matter what we did to try and force disk I/O to block the calling thread, Windows somehow maps that I/O to in-memory access to the Windows file system buffer pool, and this can be done without blocking the caller… so the caller doesn't block. Of course in many other settings the caller *would* block, for example when doing requests on a remote Web Services server. We hacked the desired delay into the assignment by calling the thread sleep primitive at the key point in the execution – obviously not something you would normally do, but it does have the desired effect. One take-away from this experience we had is the realization that normally, there actually isn't much benefit to multi-threaded I/O on a Windows file system. In fact, if you remove the thread.sleep call, you'll see that your program gives identical performance no matter how many threads it uses, until you get to some huge thread count, at which point overheads start to be an issue. With Web Services, though, you would see real delays (due to the network). So, a Web Services application talking to remote servers might get a *big* win from using multiple threads. On the other hand, asking you to code a Web Services application was a bit more than we wanted to tackle in this homework assignment! We'll save that for CS514, next spring.

One way to sort a file is to recursively subdivide the file, sort small chunks, then merge them (this is called merge-sort). For example, let's focus on the case of a single file containing 2^17 records, each record 100 bytes in length. One could subdivide such a file into two pieces, each of length 2^16. Having sorted each piece, we could then merge them to end up with a single sorted file. But each of those pieces can be treated as a sub-file of length 2^16, and we could apply the same approach to each subfile. In the limit, this sort of binary sub division will eventually get us down to a file of size 1 record, which we will sort using C#'s *Array.Sort()* method. So sorting becomes a merge operation: merging two single-record "subfiles" to obtain a new subfile of length 2 records, then merging two of these to end up with one of length 4, etc. At the end the file is a fully sorted stream of bytes. Note that the *Array.Sort()* method is to be used only when the recursion level goes down to a file-size of one record each, and not earlier. And, the bytes required are to be read from the file, and the sorted bytes are to be written back to the file (using *Read()* and *Write()*) at each level, i.e., no in-memory sorting is allowed, except at the very last level.

Now we get to the issue mentioned earlier about delaying writes. Your program will end up doing streams of record merges. If a merge is to be done involving a total of 64 or more records, *first* we delay the merge operation by 50 ms by calling *Thread.Sleep(50) (just call it once, then do the merge)*. The effect of introducing this delay is to make the threads repeatedly flip between the blocked and ready states, which is a tailor-made scenario for us to verify the benefits of using multiple threads. As mentioned earlier, this is a bit fake, but in many other settings (settings a bit too complex to set up in a homework due just a few weeks from now) the delays are built into system calls you perform and threads give a benefit from parallelism.

Now, our application uses multiple files. So, having sorted one file, the application would sort the next one, until all have been sorted.

With just one thread, what we've described is just a recursive merge-sort on a series of files. But if we are allowed to create multiple threads, new options emerge. For example, we could ask one thread to sort the first file, a second to sort the second file, etc – and we could do that sorting in parallel. Or we could use our threads to sort individual files in parallel: one thread might sort first half of a given file, while another thread simultaneously sorts the second half, and then one thread could terminate while the other merges the two halves.

We would like your program to work in the following modes: (1) with a single thread, (2) with a single thread per file, and (3) with some power of 2 of threads per-file. You don't need to deal with the case of having multiple threads, but not enough to allocate one to each file.

However, even though we want you to get the solution working for these cases, we're only going to ask that you evaluate the performance of the solution in one of them. Specifically, we want you to produce an optimized copy of your program (see the options

in the “build” dialog box). Then we want you to undertake some simple measurements of your program with a *single* file and with powers of 2 of threads: 1 thread, 2, etc. Run your file writer program from part (1) to create a random file. Now sort it with a single thread and time the run accurately (you can get time information accurate to the millisecond from the Windows clock). Next, recreate the random file and sort it again, this time with two threads, and time it again. Do it again with four threads, then eight, sixteen, and so on. (Test with successively larger number of threads till you hit the memory limit).

Graph the results, with the x-axis showing the number of threads in use and the y-axis showing the running time. We recommend that you make this graph using Microsoft excel or a similar package. Hand-drawn graphs are not acceptable.

Hand in: (a) A printout for your “random file creator” program and (b) for your “file sorter” program. (c) The graph of how performance changes as a function of the number of threads. (d) A short discussion of the performance “factors” at work here. What seems to be the best way to use threads when doing this kind of I/O intensive application?