Operating Systems
Assignment 2

Objectives:

Synchronization mechanisms in multithreading and multiprocessing. Preventing race conditions, locks, busy waiting (spinlocks), data protection in the critical region.

Task: Synchronize simultaneous file writes (A version of the 'readers-writers' problem).

For this assignment, we will use multiple worker processes - each of which needs to add updates to a data structure. This mimics common scenarios where multiple workers will be adding data to a central repository from multiple sources at the same time (OS kernel data structures, database updates, etc.).

We will use a file for the data structure for this assignment - because it is persisted to disk, easy to manage, and easy to verify. But in real life, this could be a file, a linked list, a database table, an array, tree, graph, or any other structure which receives data from multiple processes simultaneously.

Each thread must merge their separate new data with the main data file, keeping the main file in sorted order. So each process must read the main file, read the new file, merge the two, and then write the merged, sorted results back to the main file. This takes a little time, so if there is no locking mechanism, two processes can read the same file virtually instantaneously, merge their separate data sets with the main file, and then try to write their new version of the main file back to disk.

At the very least, we have a race condition where we have two (or more) conflicting versions of the data. This will cause an accidental data deletion when the processes try to write their data, and the second version overwrites the first. A demo will be given in-class as to what can go wrong if we don't use a lock for this type of operation.

What we need is for one process to acquire a lock, open the main file, merge its data, write the main file, and then release the lock so that other processes can proceed. In this way, each process is guaranteed to load updated, correct data to merge with, and no process has 'stale' data to merge with, which would cause the data anomaly (accidental deletion).

What you need to do:

Implement a program which starts three threads of execution, each of which attempts to open a data file and merge it with the main file, keeping the final results in sorted order. Data for the main file and the three data files have been provided.

You can choose to do this via multithreading, multiprocessing (generated from the same parent process), or by deploying three completely separate processes which are initiated by the operating system.

Implement a locking system, so that only one thread of execution can access the main file (critical region) at a time. This locking system should include some simple mechanism that allows the processes that do not hold the lock to obtain it once it is free. Busy waiting will probably be the easiest approach, and it will be sufficient for this assignment.

Your choices for locking could be mutexes, semaphores, a simple variable flag (like Peterson's solution), a monitor, or file locking (available on both UNIX and Windows).

Check your results against the answers given, to make sure they are correct.

Notes:

Your threads must run simultaneously, and make use of the locking mechanism. If you write one program which merges the files one after the other sequentially, you will receive very little credit (this completely misses the point, and does not cure the race condition in real life).

You can write the program in any language that you want, and use any method you like to start the three threads of execution, and to implement the lock. But keep it simple, and test your results against the answers file.

What to turn in:

1. A (very brief) summary PDF, explaining your approach and the choices you made
   This should be one page or less - just quickly summarize what you did, and why you made your choices.

2. Your source code - complete enough that it can be executed and tested.

3. Any compile / make / run instructions you used, so your code can be executed and tested as you intended.
   This can be a make file, shell script, or a simple series of instructions in a text file -
   just include whatever you did to compile & run your code.

4. The init mechanism you used to start your three processes simultaneously.
   If you used multithreading or process-forking, this will be a part of your source code.
   If you started three separate processes via a shell script, or a batch file, submit as well.

Zip files and other compressed folders will receive zero credit. Just attach your files to the email.