# **Operating Systems Project Report (Spring 2015)**

**Topic:**

“File Access (read/write) Synchronization between Processes/Threads.”

**Team Members:**

* Daniyal Kashif **(13K-2095)**, Section B.
* Muhammad Waqas Maqsood **(13K-2185)**, Section B.

**Project Description:**

On any operating system, everything visible and invisible to the user and/or system, in other words everything that exists within the bounds of the system’s world, has its existence realized through a file. Be it a process, a module, a program, an entity, a setting, an option, anything. Everything ‘exists’ at the base level as some semblance of instructions/properties written within a general container that we call a file.

When programs have to execute, they take the forms of processes, and whenever processes need something done, they require instructions on what information or object they require to manipulate in order to accomplish their task. Since all information about anything on a system is somewhere in a file, processes generally access various files during execution.

It is understood that a Process may either simply need information from a file (reading), or it may need to change information in a file (writing), because in modern computing systems, we may have multi-programmed or multi-threaded environments, wherein more than one process may be executing in the system and accessing files, it is important for the host operating system, to implement procedures which insure that process do not interfere with each other and continue to run smoothly, either concurrently or in parallel.

This was our project, to analyze ways in which modern operating systems (we performed all practical work on Linux) make sure that processes on the system run in a synchronized manner, and read/write access of files by processes occurs smoothly and avoids the classic “Readers-Writers Problem”.

**The Readers-Writers Problem:** In a nutshell the readers-writers problem is simply the process synchronization problem which presents a situation where the two kinds of processes, readers and writers compete for the access to the same resource, leading to a race condition, if either is allowed to interrupt the other the resource might be corrupted.

**How We Started:**

Since the beginning we were completely aware that the only common-sense solution to such a process synchronization problem would be that the processes simply play nice with each other and take turns whenever they want to access the same resource.

But since processes never know how to play nice on their own, we would need some sort of locking or exclusive access mechanism to make sure a single process can work with the resource uninterrupted for a given time.

Readers by definition don’t make changes to the resource, so it was considered safe to allow multiple readers concurrent access to the resource, however, we decided to make writer processes wait for their turn, that is while there were readers reading/waiting on a particular resource, no writer may access it.

Basically, our project is an analysis of all the different methods to synchronize processes for file access, the core of any operating systems operations, our project generates large text files, and then a random number of threads race for read/write access to the file. Every time that a thread acquires/releases a lock, or access the input file, it posts time stamps and event descriptions, along with its unique thread ID, to a separate output file.

**Progress Details and Problems Encountered:**

The first real decision we had to make for this project was what lock method to use, since Linux gives us both file locks and process locks, we decided to test out both approaches.

We tested out process synchronization on the following three methods, performing rigorous testing and analysis for time, efficiency and reliability.

1. The first problem we encountered was that while exploring the native File Locks libraries for Linux, we found that most File Locking procedures were actually broken and could not be depended on to handle race conditions.  
    This is because the struct for the flock in fcntl.h (default linux), does not enforce atomic instruction, furthermore, advisory locks do not block outside/uncooperative processes, that is, if some program does not check for file locks, file locks do not stop it.

We were able to make a readers preferred solution using normal fcntl code, however the writers were not blocking other writers. So, to solve the broken Linux file locking, we made use of the pid variable in the lock struct, by having processes guarantee exclusive access to the lock, by copying their own pid into the lock struct.

1. The second approach we tried was synchronizing processes through semaphores. The only problem we encountered with semaphores was that they are slower than file locks and spinlocks.
2. The third way we implemented a reader’s preferred solution to the Readers-Writers problem was through monitors. No problems were encountered during this part of the project.

**Explanation of Source Code:**

Our project is comprised of the following files:-

1. “Help.h” - Contains functions to get the current system time, generate random numbers and other constant/static variables.
2. “fcntl.h” - Contains fcntl(), takes an input file, creates n threads, has them race for the file, uses the <fcntl> flock struct to enforce file locking. Outputs results to fcntl.out.
3. “fcntl\_fix.h” - Contains fcntl\_fix(), takes an input file, creates n threads, has them race for the file, also uses the <fcntl> flock struct but fixes its race condition problems.
4. “normal.h” - Contains normal(), just has threads race for access to an input file, doesn’t enforce any sort of locking/mutual exclusion mechanism to ‘see’ the problem at its barest .
5. “Sema.h” - Same working, but uses Semaphores and locks are held through processes instead of for files.
6. “Monitors.h” - A readers-writers solution using Monitors.

**References & Sources:**

Our most helpful source during this project was undoubtedly Miss Atika, our Operating Systems Course Instructor, her clear directions and stern guidance lent us a precise and straightforward idea of how to begin, and eventually complete this project.

Apart from her invaluable support, the following websites proved to be of crucial help to us during the course of this project:

* <http://en.wikipedia.org/wiki/File_locking>  
   A description of file lock mechanisms in modern operating systems.
* <http://0pointer.de/blog/projects/locking.html>  
   A blogpost detailing the flaws in Linux File Locks.
* <https://www.kernel.org/doc/Documentation/filesystems/mandatory-locking.txt>

A system call tracing of Linux File Locks.

* <http://www.gnu.org/software/libc/manual/html_node/File-Locks.html>

The file lock struct description.

* For all help with Semaphores and Monitors, the assigned text was used.