LOUISIANA STATE UNIVERSITY
DEPARTMENT OF COMPUTER SCIENCE

CSC 4103: Operating Systems
Fall 2020
Programming Assignment # 1: Safe System Calls
Prof. Golden G. Richard III
Due Date: TBA @ Class Time

As discussed in lecture, system calls provide a well-defined interface between applications running in user mode and the operating system kernel. Essentially, system calls "beg" the operating system to perform some operation, such as reading or writing a file, which must be tightly controlled for reasons of fairness, security, etc.

In this assignment, you'll evaluate the implementation of a fictitious set of system calls that provide file I/O. These system calls allow opening, closing, reading, and writing files. One interesting aspect of this set of system calls is that the position in a file for individual read and write operations is always specified (although the natural "current position" can be specified using the anchor CURRENT_POSITION--read on for more details). Another twist is that the system call for writing data into a file must prevent the string "MZ" from appearing in the first two bytes of the file. The current implementation of the system calls appears in **`fileio.c`** and **`fileio.h`** (available from Moodle) and includes:

```
// open or create a file with pathname 'name' and return a File
// handle.  The file is always opened with read/write access. If the
// open operation fails, the global 'fserror' is set to OPEN_FAILED,
// otherwise to NONE.
File open_file(char *name);

// close a 'file'.  If the close operation fails, the global 'fserror'
// is set to CLOSE_FAILED, otherwise to NONE.
void close_file(File file);

// read at most 'num_bytes' bytes from 'file' into the buffer 'data',
// starting 'offset' bytes from the 'start' position.  The starting
// position is BEGINNING_OF_FILE, CURRENT_POSITION, or END_OF_FILE. If
// the read operation fails, the global 'fserror' is set to
// READ_FAILED, otherwise to NONE.
unsigned long read_file_from(File file, void *data, unsigned long
num_bytes, SeekAnchor start, long offset);
```

```
// write 'num_bytes' to 'file' from the buffer 'data', starting
// 'offset' bytes from the 'start' position.  The starting position
// is BEGINNING_OF_FILE, CURRENT_POSITION, or END_OF_FILE.  If an
// attempt is made to modify a file such that "MZ" appears in the
// first two bytes of the file, the write operation fails and
// ILLEGAL_MZ is stored in the global 'fserror'.  If the write fails
// for any other reason, the global 'fserror' is set to
// WRITE_FAILED, otherwise to NONE.
unsigned long write_file_at(File file, void *data, unsigned long
num_bytes, SeekAnchor start, long offset);
```

There's also a helper function that displays a string explanation of any error code that's raised by the system calls. That function is called **`fs_print_error()`.**

Your job is to evaluate (and improve) the implementation of the special check that's performed in the **`write_file_at()`** system call to prevent "MZ" from being stored in the first two bytes of a file. You should first study the implementations of the system calls, then carefully write some C programs to stress test both the current implementation and your modifications.

Your goals:

- Don't change the names or parameters of the system calls. You must improve the implementation of "MZ" checking without breaking the existing programming interface.
- Any write that doesn't result in "MZ" appearing in the first two bytes of the file is legal and **must** be allowed.
- Any write that results in "MZ" appearing in the first two bytes of the file is illegal and **must** result in an error, without modifying the file. You must **never** allow "MZ" to appear in the first two bytes of the file, regardless of how clever a programmer using the system calls might be!
- The implementation must be efficient, e.g., **an implementation of the check for "MZ" that reads the file on every write operation is not permitted**. Similarly, consuming large amounts of memory is also unacceptable.

Do your work on the **classes.csc.lsu.edu** server via `ssh`. Usernames and passwords for this server are sent via email. Please talk to me or the TA if you don't have one.

When you create a test program, e.g., `readwrite.c`, compile and link your test program with the system call implementation with a command line like this:

```
$ gcc -o readwrite readwrite.c fileio.c
```

For example, one of my (inadequate!) test programs, called `readwrite.c`, attempts a bunch of different file operations and interrogates the value of `fserror` after each operation via a call to `fs_print_error()`:

```
$ gcc -o readwrite readwrite.c fileio.c
$ ./readwrite
```

```
Creating new file called "important.dat"...
FS ERROR: NONE
Writing MAZ to beginning of file...
FS ERROR: NONE
Writing MZ to beginning of file...
FS ERROR: ILLEGAL_MZ: SHAME ON YOU!
Closing file...
FS ERROR: NONE
$
```

`readwrite.c` is available on Moodle and can be used as a starting point for developing your test programs.

When you're ready to submit, organize your solution to this assignment in a directory called **`prog1`** in your 4103 account on **classes.csc.lsu.edu**. Please put the following files in that directory:

- Your modified version of `fileio.c`.
- A copy of `fileio.h`, whether you modified it or not.
- The test programs you've developed.

Then issue the following command to turn in your solution:

```
$ ~cs4103_ric/bin/p_copy 1
```