Lab Session 3: Synchronisation threads
24296 - Operating systems

1 Objective

In this laboratory assignment, you will parallelize the CRC verification of several files, the data paths of which will be passed as command line arguments. The path to the CRC will be, as in the previous practice, obtained by appending ".crc" to the data file path. The code is given almost complete, but it has data races. You must add the relevant synchronization tools in the *fileManager.c* and *main.c* to avoid them.

The parallelization is done using threads, each of them will continuously choose a file from which no other thread is currently reading, read a 256-byte block of the data file and its CRC, and verify it. In case there is an inconsistency, the thread should log it in the standard output in the standard - specifying the file in which the inconsistency was found. Notice that different blocks of the same file can be read by different threads - but not simultaneously. To help you with this, the code already uses a File Manager structure (you can take a look at its code, or). You'll need to complete their functions (markFileAsFinished and getAndReserveFile). Also, the threads will have to exit only when there are no more files to process.

2 Delivery

Deliver a zip file (ONLY zip, don't use RAR) and include the previous multithreaded program (call it main.c), the rest of the files and headers required to compile your program, and a document answering the questions below:

1. Imagine that you have a data source which allows N simultaneous reads. What would you change, in order to implement it?
2. Why there is no need for synchronisation in the function *unreserveFile*? Would it be in the case of the previous question?
3. Compare the execution time between a version of 1 thread, and 4 threads, using the timer seen in P1. Create a table when processing a number of files ranging from 1 to 10 (you can make copies of large_file.txt).

3 Programming hints

To compile with pthreads, remember to use the adequate options:

```
gcc main.c myutils.c fileManager.c crc.c -lpthread -o main    # Linux
gcc main.c myutils.c fileManager.c crc.c-o main               # MAC
```

Remember the usage of the pthread functions to create and wait for a thread, mutex locks:

```
pthread_create(pthread_t *tid, NULL, fthread, void* p);
int pthread_join(pthread_t thread, NULL);

pthread_mutex_t lock;
pthread_mutex_lock(pthread_mutex_t* lock);
pthread_mutex_unlock(pthread_mutex_t* lock);

pthread_cond_t cond; //Condition associated to a monitor
pthread_cond_init(pthread_cond_t * cond, NULL);
pthread_cond_wait(pthread_cond_t * cond, pthread_mutex_t * lock);
pthread_cond_signal(pthread_cond_t * cond);
```

And our implementation of semaphores using monitors, which can be found in myutils.c:

```
typedef struct semaphore_struct {
        int i;
        pthread_mutex_t lock;
        pthread_cond_t cond;
} my_semaphore;

void my_sem_init(my_semaphore* sem, int i);
void my_sem_wait(my_semaphore* sem);
void my_sem_signal(my_semaphore* sem);
```

4 Explanation of the File Manager

To aid you assigning files to threads, you are a *FileManager* structure, (files fileManager.c, and its header fileManager.h), which has functions that searches for free files, and marks them as busy. This data structure contains the file descriptor of the input files, the number of files that have been already finished verifying, as well as information on whether they are being currently read by a thread. The fields are as follows:

- *nFilesTotal*: The number of total files were given to be processed.
- *nFilesRemaining*: The number of files which haven't still finished processing.
- *fdData* and *fdCRC*: these arrays will store the file descriptors of the data and CRC files.
- *fileFinished*: this array will be 1 if the i-th file has been completely read, and 0 otherwise.
- *fileAvailable*: this array will contain whether the i-th file is being currently read by a thread, and 0 otherwise.

The *FileManager* has the following functionalities. When considering the race conditions, remember that each thread can call this structure at any point, and that its content is shared by all threads.

- *void initialiseFdProvider(FileManager * fm, int argc, char **argv)* **You need to complete its implementation.** This function will receive a non-initialised FileManager struct, and initialise it (including any synchronisation tool that you might need), open all the data and CRC files, and store their fd. It will also need to allocate the memory needed for its array using malloc.
- *void destroyFdProvider(FileManager * fm)*: Frees all allocated memory and closes the open files. Has to be called at the end of the program. Already implemented.
- *int getAndReserveFile(FileManager * fm, DataEntry * de)*: **You need to complete its implementation.** This function will search in the FileManager for a file that is neither finished nor being currently read by another thread, and return its information (fds, index) in the DataEntry (see below), so the thread will be able to read a datablock and a crc of this fail. To avoid two threads to read from the same file, it will mark the file as not available. If there is no file available, it will return 1, otherwise it will return 0. Warning: what can happen if two threads call this function at the same time?
- *void unreserveFile(FileManager *fm, DataEntry * d)*: This function is to be called when the thread has finished reading the block in the file specified in the dataEntry struct, and updates the FileManager to mark the file as ready to be read.
- *void markFileAsFinished(FileManager *fm,dataEntry * d)*: This function is to be called when a file has been completely read, to mark it as finished in the FileManager struct.

To store the output result of these calls, a struct called DataEntry is given, which will store the file descriptors of the returned CRC and data files, and the file's index inside the FileManager structure, so it can be later identified by the FileManager (in order to mark the file as available when finishing). You will have to complete the *getAndReserve* and *initialiseFdProvider*. WARNING: the given implementation contains data races. Add semaphores and/or locks to avoid them.

The main will initialise the FileManager with the list of files to open, and create exactly 4 worker threads. It will then wait until all the threads have finished. In this exercise, you are asked to keep track of the available files in the FileManager using semaphores, such that when you call *getAndReserveFile*, you always obtain a valid file. Use semaphores to achieve this without busy waiting.