Javier Bellón u214002
Raúl Díaz u214409

Report Lab 3

1. Imagine that you have a data source which allows N simultaneous reads. What would you change, in order to implement it?

In fileManager.h we first have to change the definition to allow for a new parameter which will store the number of threads reading from each file.

```
typedef struct {
    int * fdData;
    int * fdCRC;
    int * fileFinished;
    int * fileAvailable;
    int * fileReading;
    char** fileName;

    int nFilesRemaining; //
    int nFilesTotal; // Aqu
} FileManager;
```

In fileManager.c we define the N specified for the number of threads that will be able to read at the same time.

```
#define N 2
```

During initialization we allocate memory for the array of ints and assign 0 to every position.

```
void  initialiseFdProvider(FileManager * fm, int argc, char **argv) {
    /* Your rest of the initialisation comes here*/
    my_sem_init(&sem, 1);
    fm->nFilesTotal = argc -1;
    fm->nFilesRemaining = fm->nFilesTotal;
    // Initialise enough memory to  store the arrays
    fm->fdData = malloc(sizeof(int) * fm->nFilesTotal);
    fm->fdCRC= malloc(sizeof(int) * fm->nFilesTotal);
    fm->fileFinished = malloc(sizeof(int) * fm->nFilesTotal);
    fm->fileAvailable = malloc(sizeof(int) * fm->nFilesTotal);
    fm->fileName = malloc(sizeof(char)*fm->nFilesTotal);
    fm->fileReading = malloc(sizeof(int) * fm->nFilesTotal);
    for (int i = 0; i < fm->nFilesTotal; i++){
        fm->fileName[i] = malloc(sizeof(char)*50);
    }
```

```
for (int i = 1; i < fm->nFilesTotal +1; ++i) {
    char path[100];
    strcpy(path, argv[i]);
    strcat(path, ".crc");
    fm->fdData[i-1] = open(argv[i], O_RDONLY);
    fm->fdCRC[i-1] = open(path, O_RDONLY);
    fm->fileReading = 0;
    strcpy(fm->fileName[i-1], argv[i]);

    if (fm->fdData[i-1] < 0 || fm->fdCRC[i-1] < 0){
        printf("Couldn't open at least one of the files\n");
        exit(1);
    }
```

We also free the allocated memory during the destruction of the provider.

```
void  destroyFdProvider(FileManager * fm) {
    int i;
    for (i = 0; i < fm->nFilesTotal; i++) {
        close(fm->fdData[i]);
        close(fm->fdCRC[i]);
        free(fm->fileName[i]);
    }
    free(fm->fdData);
    free(fm->fdCRC);
    free(fm->fileFinished);
    free(fm->fileName);
    free(fm->fileReading);
}
```

When we reserve a file, we update the counter of threads reading that file, if the file is now being read N times we mark it as unavailable.

```
int getAndReserveFile(FileManager *fm, dataEntry * d) {
    // This function needs to be implemented by the students
    int i;
    for (i = 0; i < fm->nFilesTotal; ++i) {
        if (fm->fileAvailable[i] && !fm->fileFinished[i]) {
            d->fdcrc = fm->fdCRC[i];
            d->fddata = fm->fdData[i];
            d->index = i;
            d->filename = fm->fileName[i];

            // You should mark that the file is not available
            my_sem_wait(&sem);
            fm->fileReading[i] += 1;
            if (fm->fileReading[i] >= N){
                fm->fileAvailable[i] = 0;
            }
            my_sem_signal(&sem);
            return 0;
        }
    }
```

When we unreserve the file we just decrease the counter of threads reading the file and mark the file as available in case it wasn't before because not it has N-1 threads reading from it.

```
void unreserveFile(FileManager *fm,dataEntry * d) {
    my_sem_wait(&sem);
    fm->fileReading--;
    fm->fileAvailable[d->index] = 1;
    my_sem_signal(&sem);
}
```

The rest remains unchanged.

2. Why there is no need for synchronisation in the function unreserveFile? Would it be in the case of the previous question?

Because only one thread can read from a file at the same time, that means that if a file is reserved, no other thread can access that file (they will see it's reserved and will skip it) so the file's position in the array of files available won't be accessed from outside the thread.

Because there is no way to unreserve the same file more than once there is no need to account for synchronization. In the previous question we had to add synchronization tools because now multiple threads are accessing the same position in the array and we must take that into consideration in order to avoid unexpected behaviour.

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).

We copìed large_file into 10 files and computed their crc to fill the table.

Rows: Threads
Columns: Files

	1	2	3	4	5	6	7	8	9	10
1	84	165	246	328	409	491	572	655	736	820
4	84	85	93	101	177	180	192	212	287	293

For one file the time is the same because the file is only being read from one thread at the same time regardless of the total of threads available. From then on the bigger the number of files the greater the difference in times.