CSCE 613: Project 6 Design

Student: Caio Duarte Diniz Monteiro

Date: 05/02/16

Bonus points options done:

- Option 0 (10 points) – Early submission of Part I
- Option 2 (10 points) – Design document of a thread-safe disk and file system
- Option 3 (10 points) – Implementation of a thread-safe disk and fle system

Blocking disk class

```
class BlockingDisk : public SimpleDisk{
        BOOLEAN busy;
    protected:
        //OVERRIDE
        void wait_until_ready();

    public:
        BlockingDisk(DISK_ID _disk_id, unsigned int _size);
        /* Creates a SimpleDisk device with the given size connected to the MASTER
           or SLAVE slot of the primary ATA controller. */

        /*DISK OPERATIONS */
        void read(unsigned long _block_no, unsigned char * _buf);
        /* Reads 512 Bytes from the given block of the disk and copies them
            to the given buffer. No error check! */
        void write(unsigned long _block_no, unsigned char * _buf);
        /* Writes 512 Bytes from the buffer to the given block on the disk. */

};
```

The Blocking disk class extends from the Simple Disk and makes it possible to yield the CPU while waiting for a disk operation to complete, this will avoid wasting CPU time.

Constructor

The constructor simply passes the call to the Simple Disk class constructor, nothing needed to change here.

Read and Write functions

Read and Write functions were overridden just to implement the thread safety (described in document - design - Bonus option 2.docx). If it was not because of that, they would not be overridden and the calls for those functions would just use the Simple Disk implementation.

Wait until ready function

Wait until ready function was overridden to pass the CPU to another process while waiting for the current disk operation. Whenever the disk operation is not ready, it will put the current thread awaiting on a blocked threads queue, and pass the CPU to another thread. Then, every time that a resume is performed

on the Scheduler, the blocked threads queue is checked to see if there is any threads waiting for the disk. If that is the case, the CPU is yielded to that thread, and it checks again if the disk is ready, continue with its expected execution if ready or going to the blocked threads queue again if not ready.

File class

File class represents the data of a file and its metadata. On this design, I have limited the files to be contained on a single block (discussed this with Dr. da Silva in class), simplifying the metadata overhead on the file system. Each file now just has to contain: The file id, which identifies the file within the file system; the block number, which indicates in which block of the disk the file is physically stored; Current position, which indicates in which byte of data any read or write operations are going to start.

```
class File {
    friend class FileSystem;
    /* -- your file data structures here ... */
     unsigned int   file_id;
     unsigned int   block_num; //number of the block containing the file
     unsigned int   current_position; //current position in the block
     BOOLEAN busy; //variable used for thread-safe file access
```

Constructor

```
File();
File(unsigned int fileID);
/* Constructor for the file handle. Set the 'current position'
* to be at the beginning of the file.
```

There are two constructor signatures for a file, one doesn't have any parameters and the other one receives a fileID. Both of them starts the current position at 0. The one with the parameter, checks if a file with that ID already exist and return it if so. Otherwise, it will create the file with the specified ID.

Read function

```
unsigned int Read(unsigned int _n, char * _buf);
```

Read function receives the number of bytes to read and a buffer to where the read data should be stored. It will then read the file content until _n bytes are read or reach the end of file, whatever happens first. If not all the expected bytes are read, it will display a warning message and return the number of bytes read.

Write function

```
void Write(unsigned int _n, char * _buf);
```

Write function receives the same parameters as the read function, but here the _n means the number of bytes that you want to write to file, and the buffer contains the data to be written. It will first read the file from disk, and start writing the buff to the local copy of it, if it writes all the expected _n bytes, then the changes are saved to the disk. If it reaches the end of file before writing everything, an error message is displayed and the changes are discarded.

Reset, Rewrite, and EoF functions

```
void Reset();
/* Set the 'current position' at the beginning of the file.
*/
void Rewrite();
/* Erase the content of the file. Return any freed blocks.
* Note: This function does not delete the file!
* It just erases its content.
*/
BOOLEAN EoF();
/* Is the current location for the file at the end of the file? */
```

These three functions are sort of auxiliary functions for the file manipulation. Reset function puts the current position variable back at 0, so you can read and write from the beginning of it. Rewrite function is similar to the reset function, because it will also return the current position to 0, but besides that, it will erase all the data content of the file, making it a blank file with no content. End of file function checks to see if the current position is equals to the last byte of the data array, meaning it reached the end of file.

File System class

```
class FileSystem {
    friend class File;
    /* your file system data structures here ... */
    SimpleDisk * disk; //disk accessed by the file system
    static unsigned int size; //size of the file system
    File* files; //keeps track of the files in the file system
    unsigned int files_size; //number of files in the file system
    static BOOLEAN busy; //variable used for thread-safe file system access
```

File system class is responsible for managing the files on the disk, keeping track of them and how many files there are. Since it directly manipulates the disk, it needs to store a pointer to the disk. It has a size, so you know how many files you can store on the file system. And it has a list of files and the number of files on the file system.

Constructor

File system constructor, initializes the number of files in the file system to 0, and the array of files to be null. It also clears the disk_buffer array to make sure it is proper to use (without any garbage values) in further operations.

Mount and Format Functions

```
BOOLEAN Mount(SimpleDisk * _disk);
/* Associates the file system with a disk.
* Limit to at most one file system per disk.
* Returns TRUE if operation successful (i.e. there is indeed
* a file system on the disk.
*/
static BOOLEAN Format(SimpleDisk * _disk, unsigned int _size);
/* Wipes any file system from the given disk and installs an
* empty file system of given size.
*/
```

Format function is responsible to create a file system with the desired size on the specified disk. It will simply erase all the all the contents on disk until the desired size is reached.

The Mount function will mount the file system on top of the disk. It will read the disk until the size defined on the format, adding existing files to the files array and updating the number of files present in the file system.

LookupFile function

```
BOOLEAN LookupFile(int _file_id, File * _file);
```

Lookup function receives a file ID as parameter and looks up for a file with that ID on the files array. If such file is found, it stores the reference to the file on the _file pointer and return true, indicating the search was successful. Otherwise, it will return false, and will not set the _file pointer.

Create File and Delete File functions

```
BOOLEAN CreateFile(int _file_id);
/* Create file with given id in the file system.
* If file exists already, abort and return FALSE.
* Otherwise, return TRUE.
*/
BOOLEAN DeleteFile(int _file_id);
/* Delete file with given id in the file system and free any
* disk block occupied by the file.
```

Create File function receives the file ID as parameter and tries to create a new file with that ID. It will first make sure that no other file on the file system has this ID (file IDs are unique). Then, it will allocate a block to hold the newly created file, it will set the file metadata with the file id name and allocated block number, add the file to the files array, write it to disk, and return true. If another file with the same ID already existed, if there was no free blocks to allocate, or any other problem occur, the function would return false.

Delete File function also receives the file ID as parameter. It will search for a file with that ID on the file system, and return false if no file is found. If the file is found it will then free the block that contains the file and remove it from the files array on the file system.

Allocate Block and Free Block functions

```
int AllocateBlock();
/* Allocates a block from the available ones, for use in file */
void FreeBlock(unsigned int block_index);
/* Deallocates a used block */
```

Allocate and Free block functions are the auxiliary functions used by Create and Delete files respectively. The allocate function traverse the disk blocks of the file system looking for an available block, setting it to used and returning its number if found. Otherwise, it will return -1. Free block function will set the block identified by the parameter block_index to be free, making it available to be used by the file system.

Exercise file system function

```
void exercise_file_system(FileSystem * _file_system, SimpleDisk * _simple_disk)
```

The exercise file system function is the function in the Kernel responsible for testing the file system of our OS. It receives a pointer to the OS file system object and another one to the hard disk. It will then perform the following operations on the file system and files:

- Format the disk using a size of 2048, making the file system to have 4 blocks.
- Mount the file system on the disk.
- Create two files on the file system, with IDs 1 and two.
- Look up for file with ID 1 in the file system, should be able to find it.
- Look up for file with ID 2 in the file system, should be able to find it also.
- It will then perform a series of operations on the file with ID 2:
    - Write 504 bytes of data to file (the entire data available on the file)
    - Reset the current position and read the same 504 bytes just written
    - Compare the written and read bytes to make sure they are the same
    - Test if file is on EoF, should be true
    - Try to read and write on the file, both should display error messages because we are already on the EoF
    - Rewrite the file
    - Test EoF again, now it should be false, because we just rewritten the file
    - Write to the file again, now it should be ok to write
    - Delete the file
    - Lookup for the file with ID 2, should not find it because we have just deleted
    - Lookup for the file with ID 1, should find it because we did not delete it
- Perform the same set of operations mentioned above on the file with ID 1. At the end neither file with ID 1 or 2 should be found by the Lookup calls.

With this testing routine we tested all the functions of both File and File System and checked the output with the expected behavior.