CSCE 613: Project 6 Bonus Option 2 design

Student: Caio Duarte Diniz Monteiro

Date: 05/02/16

Binary semaphore using Locks

```
//methods to implement a lock to be used on the thread-safe disk and file system bonus option 3
static void Lock(BOOLEAN& lock)
{
    while(lock){
        Console::puts("Waiting for another disk/file system operation... yield CPU to another thread!\n");
        SYSTEM_SCHEDULER->resume(Thread::CurrentThread());
        SYSTEM_SCHEDULER->yield(); //pass the cpu to the next thread on the queue
    }

    lock = TRUE;
}

static void Unlock(BOOLEAN& lock)
{
    lock = FALSE;
}
```

In order to implement a thread-safe file system and disk access, we need to guarantee that concurrent access to those resources are not going to create any problems. There are several approaches possible for that, ranging from very simple and limited solutions to more robust ones. My approach was to use something in between, where I would have time to implement and test it and would not severely limit the system. Considering this I decided to use binary semaphores to handle thread-safety, the semaphore will guarantee just a single access at a time is made on the shared resources.

I implemented the semaphore in two global functions, a Lock function and an Unlock function. Each of these two functions receives the “lock” variable that controls the mutual exclusion for a particular resource. The Lock function should be called whenever mutual exclusion is required over an resource, on the Lock function, a while loop prevents the code to continue executing until the lock variable is released by the thread currently using the desired resource. Inside the while loop the blocked thread is resumed and the CPU is yielded to another process, avoiding to waste any CPU time. Once the resource is available, the blocked thread will get out of the while loop and hold the lock for the resource. The Unlock function should be called right after performing the desired operations on the shared resource, this will remove the lock control from the current thread and making it available to other threads.

Disk access

The disk implementation used on this project uses a simple programmed I/O, capable to dealing with just one disk operation at a time. So, to guarantee the thread safety, the blocking disk class have a Boolean variable busy, that controls the lock on the resource, and I overridden the read and write functions to always call the Lock function at the beginning and the Unlock at the end.

File access

The file class also has a Boolean variable to control the lock to access it, this prevent mutual changes on the same file. Then, the Lock and Unlock function are called on this variable at the beginning and end of the Read, Write, and Rewrite functions.

File System access

File System class has another Boolean variable to control the mutual exclusion on it. For the file system we need to guarantee that no concurrent changes happen at the same time on it, avoiding conflicting changes. Because of that, the Lock and Unlock functions are then called on the beginning and end of the functions that modify the file system: Mount, Format, CreateFile, and DeleteFile.