Project 4

Due June 3, 2020 at 11:59 PM

You will be working alone for this project. This specification is subject to change at anytime for additional clarification. For this project, you will be implementing a virtual machine threading API in either C or C++. **Your virtual machine will be tested on the CSIF machines.** You must submit the provided source code (including apps), your source files, readme and Makefile in a tgz file to Canvas prior to the deadline. Submit your project on Canvas.

You will be continuing the development of your virtual machine. You will be adding the ability to mount and manipulate a FAT file system image. Except were specified in this description the API will remain as stated in the Project 2 & 3 descriptions. Files that are opened in the virtual machine will now be opened on the FAT image instead of in the host file system. The reading and writing of the files will affect the FAT image with the exception of file descriptor 0 – 2, which will still affect the standard in, out, and error files. The fatgen103.pdf in the resources is the Microsoft white paper on FAT. You do not need to fully implement all possible FAT file systems. You may make the following assumptions:

1. All images have 512 byte sectors
2. All images are FAT 16
3. Long file name entries will contiguously proceed the corresponding short file name entry

You must implement the ability to create, read and write files that are in the root directory and to open and read the root directory. You must be able to read all of the short file name entries (long file name entries can be skipped) in the root directory.

A working example of the vm and apps can be found in /home/cjnitta/ecs150. The vm syntax is vm [options] appname [appargs]. The possible options for vm are -t, -s and -f; -t specifies the tick time in millisecond, -s specifies the size of the shared memory used between the virtual machine and machine, and -f specifies the FAT image to be mounted. By default the tick time is set to 100ms, for debugging purposes you can increase this value to slow the running of the vm. The size of the shared memory is 16KiB. The default FAT image is fat.ima. When specifying the application name the ./ should be prepended otherwise vm may fail to load the shared object file. In addition you will find several programs that will help create and manipulate the fat images in the /home/cjnitta/ecs150 directory.

Several functions have been provided for you in the VirtualMachineUtils.c that you might find useful, not all will be necessary: VMDateTime, VMFileSystemValidPathName, VMFileSystemIsRelativePath, VMFileSystemIsAbsolutePath, VMFileSystemGetAbsolutePath, VMFileSystemPathIsOnMount, VMFileSystemDirectoryFromFullPath,

VMFileSystemFileFromFullPath, VMFileSystemConsolidatePath, VMFileSystemSimplifyPath, VMFileSystemRelativePath.

The function specifications for both the virtual machine and machine are provided in the subsequent pages.

You should avoid using existing source code as a primer that is currently available on the Internet. You **must** specify in your readme file any sources of code that you or your partner have viewed to help you complete this project. All class projects will be submitted to MOSS to determine if pairs of students have excessively collaborated with other pairs. Excessive collaboration, or failure to list external code sources will result in the matter being transferred to Student Judicial Affairs.

Extra Credit:
There are two extra credit opportunities.

1. Add support for subdirectories, this includes creating, deleting, and modifying files and directories anywhere in the directory structure. In addition, implement the VMDirectoryUnlink and VMDirectoryCreate functions.
2. Add support for long file names, the long file names should be both read correctly from directories as well as created when new files are added to the mounted file system.

Helpful Hints

- You may want to keep your working VMFile functions and rename them so they are only used internally. You can then use them to communicate with the machine for the first three file descriptors and to manipulate the fat image.
- Your new VMFile functions should pass through to the machine if the file descriptor is between 0 and 2 (you can call your new internal versions, or just modify your VMFile functions). If the value is not between 0 and 2 it should be a file operation on in the FAT file system.
- Open the image file with O_RDWR flag and 0600 mode using MachineFileOpen in VMStart before calling VMMain. (Or use your internal VMFileOpen)
- If you have implemented a mutex, you may want to use one for the FAT image to avoid race conditions.
- You will want to write functions to read and write full sectors from/to the image.
- Write a function to read in the volume information and verify that this matches the information provided using the fatvol.
- You may want to read in the entire FAT Table and hold it in memory as it will be accessed a lot. You should verify that it matches the values output with fatfatdump. Being able to find the next cluster in the chain or allocating a value from the table will likely be very helpful. You may also want to be able to find a cluster number from a beginning cluster and a byte offset.
- You will probably want to write functions to read and write full clusters from/to the image. This will most likely use the sector functions.

- Caching the FAT table and data clusters may be helpful in implementing the file system.
- You will probably want to write class to cache that caches full clusters. The interface could have a read and write functions that take in a cluster number, data pointer, and amount to read/write. If the cluster isn't in the cache it can first be loaded before the read or write. If the cluster is marked dirty, then it can be written back out during the dismount.
- Write a function that can parse the directory entries. You will want to verify that the information read in matches the root directory entries. You may find fatdirdump helpful.
- Use the shell.so app to test reading of files. Verify correct reading of the file system before attempting to write the file system.

Name

VMStart – Start the virtual machine.

Synopsys

```
#include "VirtualMachine.h"

TVMStatus VMStart(int tickms, TVMMemorySize sharedsize,
const char *mount, int argc, char *argv[]);
```

Description

VMStart() starts the virtual machine by loading the module specified by *argv*[0]. The *argc* and *argv* are passed directly into the VMMain() function that exists in the loaded module. The time in milliseconds of the virtual machine tick is specified by the *tickms* parameter. The size of the shared memory space between the virtual machine and the machine is specified by the *sharedsize*. The name of the FAT file system image is specified by *mount*.

Return Value

Upon successful loading and running of the VMMain() function, VMStart() will return VM_STATUS_SUCCESS after VMMain() returns. If the module fails to load, or the module does not contain a VMMain() function, or if the *mount* FAT file system image cannot be loaded, VM_STATUS_FAILURE is returned.

Name

VMFileOpen – Opens and possibly creates a file in the mounted FAT file system.

Synopsys

```
#include "VirtualMachine.h"

TVMStatus VMFileOpen(const char *filename, int flags, int mode,
int *filedescriptor);
```

Description

VMFileOpen() attempts to open the file specified by *filename*, using the flags specified by *flags* parameter, and mode specified by *mode* parameter. The file descriptor of the newly opened file will be placed in the location specified by *filedescriptor*. The flags and mode values follow the same format as that of open system call. The filedescriptor returned can be used in subsequent calls to VMFileClose(), VMFileRead(), VMFileWrite(), and VMFileSeek(). When a thread calls VMFileOpen() it blocks in the wait state VM_THREAD_STATE_WAITING until the either successful or unsuccessful opening of the file is completed. In order to prevent confusion with standard in, out and error, the file descriptor returned in the location specified by *filedescriptor* will be 3 or larger.

Return Value

Upon successful opening of the file, VMFileOpen() returns VM_STATUS_SUCCESS, upon failure VMFileOpen() returns VM_STATUS_FAILURE. If either *filename* or *filedescriptor* are NULL, VMFileOpen() returns VM_STATUS_ERROR_INVALID_PARAMETER.

Name

VMFileClose – Closes a file that was previously opened in the mounted FAT file system.

Synopsys

```
#include "VirtualMachine.h"

TVMStatus VMFileClose(int filedescriptor);
```

Description

VMFileClose() closes a file previously opened with a call to VMFileOpen().When a thread calls VMFileClose() it blocks in the wait state VM_THREAD_STATE_WAITING until the either successful or unsuccessful closing of the file is completed.

Return Value

Upon successful closing of the file VMFileClose() returns VM_STATUS_SUCCESS, upon failure VMFileClose() returns VM_STATUS_FAILURE.

Name

VMFileRead – Reads data from a file.

Synopsys

```
#include "VirtualMachine.h"

TVMStatus VMFileRead(int filedescriptor, void *data, int *length);
```

Description

VMFileRead() attempts to read the number of bytes specified in the integer referenced by *length* into the location specified by *data* from the file specified by *filedescriptor*. The *filedescriptor* should have been obtained by a previous call to VMFileOpen(). The actual number of bytes transferred by the read will be updated in the *length* location. When a thread calls VMFileRead() it blocks in the wait state VM_THREAD_STATE_WAITING until the either successful or unsuccessful reading of the file is completed. If the filedescriptor is less than 3 the reading will be attempted on the main file system, if the value of filedescriptor is 3 or larger the reading will be attempted on the mounted FAT file system file.

Return Value

Upon successful reading from the file, VMFileRead() returns VM_STATUS_SUCCESS, upon failure VMFileRead() returns VM_STATUS_FAILURE. If *data* or *length* parameters are NULL, VMFileRead() returns VM_STATUS_ERROR_INVALID_PARAMETER.

Name

VMFileWrite – Writes data to a file.

Synopsys

```
#include "VirtualMachine.h"

TVMStatus VMFileWrite(int filedescriptor, void *data, int *length);
```

Description

VMFileWrite() attempts to write the number of bytes specified in the integer referenced by *length* from the location specified by *data* to the file specified by *filedescriptor*. The *filedescriptor* should have been obtained by a previous call to VMFileOpen(). The actual number of bytes transferred by the write will be updated in the *length* location. When a thread calls VMFileWrite() it blocks in the wait state VM_THREAD_STATE_WAITING until the either successful or unsuccessful writing of the file is completed. If the *filedescriptor* is less than 3 the writing will be attempted on the main file system, if the value of *filedescriptor* is 3 or larger the writing will be attempted on the mounted FAT file system file.

Return Value

Upon successful writing from the file, VMFileWrite() returns VM_STATUS_SUCCESS, upon failure VMFileWrite() returns VM_STATUS_FAILURE. If *data* or *length* parameters are NULL, VMFileWrite() returns VM_STATUS_ERROR_INVALID_PARAMETER.

Name

VMFileSeek – Seeks within a file.

Synopsys

```
#include "VirtualMachine.h"

TVMStatus VMFileSeek(int filedescriptor, int offset, int whence,
int *newoffset);
```

Description

VMFileSeek() attempts to seek the number of bytes specified by *offset* from the location specified by *whence* in the file specified by *filedescriptor*. The *filedescriptor* should have been obtained by a previous call to VMFileOpen(). The new offset placed in the *newoffset* location if the parameter is not NULL. When a thread calls VMFileSeek() it blocks in the wait state VM_THREAD_STATE_WAITING until the either successful or unsuccessful seeking in the file is completed. If the *filedescriptor* is less than 3 the seeking will be attempted on the main file system, if the value of *filedescriptor* is 3 or larger the seeking will be attempted on the mounted FAT file system file.

Return Value

Upon successful seeking in the file, VMFileSeek () returns VM_STATUS_SUCCESS, upon failure VMFileSeek() returns VM_STATUS_FAILURE.

Name

VMDirectoryOpen – Opens a directory for reading in the mounted FAT file system.

Synopsys

```
#include "VirtualMachine.h"

TVMStatus VMDirectoryOpen(const char *dirname, int *dirdescriptor);
```

Description

VMDirectoryOpen() attempts to open the directory specified by *dirname*. The directory descriptor of the newly opened directory will be placed in the location specified by *dirdescriptor*. The directory descriptor returned can be used in subsequent calls to VMDirectoryClose(), VMDirectoryRead(), and VMDirectoryRewind(). When a thread calls VMDirectoryOpen() it must block in the wait state VM_THREAD_STATE_WAITING if the opening of the directory cannot be completed immediately. In order to prevent confusion with standard in, out and error, the file descriptor returned in the location specified by *dirdescriptor* will be 3 or larger.

Return Value

Upon successful opening of the directory, VMDirectoryOpen() returns VM_STATUS_SUCCESS, upon failure VMDirectoryOpen() returns VM_STATUS_FAILURE. If either *dirname* or *dirdescriptor* are NULL, VMDirectoryOpen() returns VM_STATUS_ERROR_INVALID_PARAMETER.

Name

VMDirectoryClose – Closes a directory that was previously opened in the mounted FAT file system.

Synopsys

```
#include "VirtualMachine.h"

TVMStatus VMDirectoryClose(int dirdescriptor);
```

Description

VMDirectoryClose() closes a directory previously opened with a call to VMDirectoryOpen().When a thread calls VMDirectoryClose() it blocks in the wait state VM_THREAD_STATE_WAITING if the closing of the directory cannot be completed immediately.

Return Value

Upon successful closing of the directory VMDirectoryClose() returns VM_STATUS_SUCCESS, upon failure VMDirectoryClose() returns VM_STATUS_FAILURE.

Name

VMDirectoryRead – Reads a directory entry from a directory in the mounted FAT file system.

Synopsys

```
#include "VirtualMachine.h"

TVMStatus VMDirectoryRead(int dirdescriptor,
SVMDirectoryEntryRef dirent);
```

Description

VMDirectoryRead() attempts to read the next directory entry into the location specified by *dirent* from the file specified by *dirdescriptor*. The *dirdescriptor* should have been obtained by a previous call to VMDirectoryOpen(). When a thread calls VMDirectoryRead() it blocks in the wait state VM_THREAD_STATE_WAITING if the reading of the directory entry cannot be completed immediately.

Return Value

Upon successful reading from the directory, VMDirectoryRead() returns VM_STATUS_SUCCESS, upon failure VMDirectoryRead() returns VM_STATUS_FAILURE. If *dirent* parameter is NULL, VMDirectoryRead() returns VM_STATUS_ERROR_INVALID_PARAMETER.

Name

VMDirectoryRewind – Rewinds a previously opened directory to the beginning.

Synopsys

```
#include "VirtualMachine.h"

TVMStatus VMDirectoryRewind(int dirdescriptor);
```

Description

VMDirectoryRewind() attempts to rewind the directory specified by *dirdescriptor* to the beginning. The *dirdescriptor* should have been obtained by a previous call to VMDirectoryOpen(). When a thread calls VMDirectoryRewind() it should block in the wait state VM_THREAD_STATE_WAITING if it is not possible to complete the rewind operation immediately.

Return Value

Upon successful rewinding of the directory, VMDirectoryRewind() returns VM_STATUS_SUCCESS, upon failure VMDirectoryRewind() returns VM_STATUS_FAILURE.

Name

VMDirectoryCurrent – Fill the absolute path with the current working directory in the mounted FAT filesystem.

Synopsys

```
#include "VirtualMachine.h"

TVMStatus VMDirectoryCurrent(char *abspath);
```

Description

VMDirectoryCurrent() attempts to place the absolute path of the current working directory in the location specified by *abspath*.

Return Value

Upon successful copying of the directory name into *abspath*, VMDirectoryCurrent() returns VM_STATUS_SUCCESS, if *abspath* is NULL then VMDirectoryCurrent() returns VM_STATUS_INVALID_PARAMETER.

Name

VMDirectoryChange – Changes the current working directory of the mounted FAT file system.

Synopsys

```
#include "VirtualMachine.h"

TVMStatus VMDirectoryChange(const char *path);
```

Description

VMDirectoryChange() attempts to change the current working directory of the mounted FAT file system to the name specified by *path*.

Return Value

Upon successful changing of the directory to *path*, VMDirectoryChange() returns VM_STATUS_SUCCESS, if path specified by *path* does not exist, VMDirectoryChange() returns VM_STATUS_FAILURE. If *path* is NULL then VMDirectoryChange() returns VM_STATUS_INVALID_PARAMETER.

Name

VMDirectoryCreate – Attempts to create a directory in the mounted FAT file system. (EXTRA CREDIT ONLY)

Synopsys

```
#include "VirtualMachine.h"

TVMStatus VMDirectoryCreate(const char *dirname);
```

Description

VMDirectoryCreate() attempts to create a directory in the mounted FAT file system specified by *dirname*.

Return Value

Upon successful creation of the directory, VMDirectoryCreate() returns VM_STATUS_SUCCESS, if *dirname* is NULL then VMDirectoryCreate() returns VM_STATUS_INVALID_PARAMETER. If the directory is unable to be created, VMDirectoryCreate() returns VM_STATUS_FAILURE.

Name

VMDirectoryUnlink – Removes a directory or file from the mounted FAT file system. (EXTRA CREDIT ONLY)

Synopsys

```
#include "VirtualMachine.h"

TVMStatus VMDirectoryUnlink(const char *path);
```

Description

VMDirectoryUnlink() attempts to remove the file or directory specified by *path* from the mounted FAT file system. VMDirectoryUnlink() will fail if the file or directory is current opened, or if the directory attempting to be unlinked is a parent of an open file or directory.

Return Value

Upon successful removing of the directory or file, VMDirectoryUnlink() returns VM_STATUS_SUCCESS, upon failure VMDirectoryUnlink() returns VM_STATUS_FAILURE. If *path* is NULL then VMDirectoryUnlink() returns VM_STATUS_INVALID_PARAMETER.