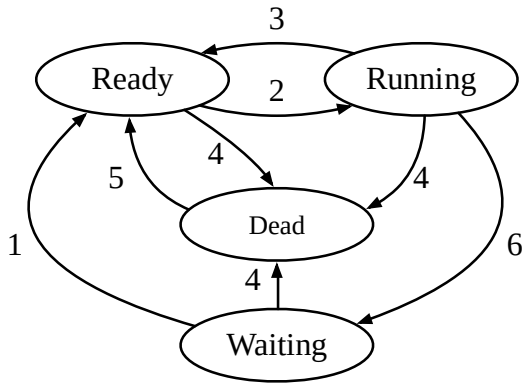
Project 2

Due May 4, 2020 at 11:59 PM

You will be working alone on 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 your source files, readme and Makefile in a tgz file to Canvas prior to the deadline.

You will be provided a machine abstraction upon which you will be building the thread scheduler. The virtual machine will load the "applications" from shared objects that implement the VMMain function. The virtual machine will need to support multiple user space preemptive threads. Threads at the same priority level will time share with a quantum of one tick. The virtual machine file access is provided through the file API.

Threads have three priority levels low, medium, and high. Threads are created in the dead state and have and have state transitions as shown below.



1. I/O, acquire mutex or timeout
2. Scheduler selects process
3. Process quantum up
4. Process terminates
5. Process activated
6. Process blocks

A makefile has been provided that compiles the virtual machine as long as you provide your code as VirtualMachine.c or VirtualMachine.cpp. Do not modify any of the other files, just create your VirtualMachine.c or VirtualMachine.cpp file. It will create the virtual machine call **vm**. The applications can be built by making the apps with **make apps**. New apps can be built by adding a C or C++ file in the apps directory and adding $(BINDIR)/**filename**.so to the Makefile apps line dependencies.

A working example of the vm and apps can be found in /home/cjnitta/ecs150. The vm syntax is vm [option] appname [appargs]. The possible option for vm is -t; -t specifies the tick time in millisecond. By default this is set to 100ms, for debugging purposes you can increase these values to slow the running of the vm. When specifying the application name the ./ should be prepended otherwise vm may fail to load the shared object file.

The machine layer is implemented using a cooperative process that communicates using the System V message queues. As such during your development your program may crash prior to the closing of the message queues. In order to determine the message queue id, you can use the

ipcs command from the shell. You can remove the message queue with a call to ipcrm. You can read more about the System V IPC commands at:
http://man7.org/linux/man-pages/man1/ipcs.1.html
http://man7.org/linux/man-pages/man1/ipcrm.1.html

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 have viewed to help you complete this project. If you do not have any sources, you must specify that as well. All class projects will be submitted to MOSS to determine if students have excessively collaborated with others. Excessive collaboration, or failure to list external code sources will result in the matter being transferred to Student Judicial Affairs.

Helpful Hints

- Create a VirtualMachine.c or VirtualMachine.cpp file and include VirtualMachine.h
- If your code is in VirtualMachine.cpp, you need to enclose all of your functions in `extern "C" {}`. For example it might look like:

```
#include "VirtualMachine.h"
extern "C"{

TVMStatus VMStart(int tickms, int argc, char *argv[]){
...
}
...
TVMStatus VMFileWrite(int filedescriptor, void *data, int
*length){
...
}

}
```

- You should probably get the applications working in the following order: `hello.so`, `sleep.so`, `thread.so`, `file.so`, and then `preempt.so`. File might be able to be done sooner but may need to be revisited once threads are added if done after `hello.so`.
- The `VMStart` and `VMFileWrite` functions will be the first you will want to write.
- You will want to use a skeleton function to be the initial entry point for the thread, and for it to call the entry from the `VMThreadCreate`. This is necessary in case the thread doesn't explicitly call `VMThreadTerminate`.
- Disabling signals with `MachineSuspendSignals` and resuming them with `MachineResumeSignals` is similar to disabling and resuming interrupts and can provide mutual exclusion during the execution of critical code. This pair of functions can be safely nested without loss of mutual exclusion in the outer set.

- Don't forget to enable signals with `MachineEnableSignals` before calling the `VMMain` app entry point.
- You will need to use the `volatile` keyword for variables that may get modified during a signal handler. The `volatile` keyword guarantees that the compiler will generate code to go to memory for every access of the variable.
- You will likely want an idle thread that will execute when all other threads are blocked. Conveniently the thread priorities are set to HIGH, NORMAL, and LOW as 3, 2, 1, so a lower priority could be used for IDLE.
- It may seem intuitive to initialize the `SMachineContext` in `VMThreadCreate` with `MachineContextCreate`; however, you will run into difficulties with reactivating dead threads if not done in `VMThreadActivate`.
- Remember that when `MachineContextSwitch` is called the context is switched, so the function will not return until the context is switched back. You will need to update global and local variables before switching contexts.

Beginning VMStart

At the very beginning to get `hello.so` working you will want your `VMStart` to do the following in order:

1. Load the module with `VMLoad` that is specified by `argv[0]`.
2. Initialize the machine with `MachineInitialize`.
3. Enable signals with `MachineEnableSignals`.
4. Call the `VMMain` entry point.
5. Terminate the machine with `MachineTerminate`.
6. Unload the module with `VMUnloadModule`.
7. Return from `VMStart`.

As you add more functionality you will be inserting more code at various points in `VMStart`.

Name

VMStart – Start the virtual machine.

Synopsys

```
#include "VirtualMachine.h"

TVMStatus VMStart(int tickms, 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.

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, VM_STATUS_FAILURE is returned.

Name

VMLoadModule – Loads the module and returns a reference to VMMain function.

Synopsys

```
#include "VirtualMachine.h"
typedef void (*TVMMainEntry)(int, char*[]);

TVMMainEntry VMLoadModule(const char *module);
```

Description

VMLoadModule() loads the shared object module (or application) specified by the *module* filename. Once the module has been loaded a reference to VMMain function obtained. The source for VMLoadModule is provided in VirtualMachineUtils.c

Return Value

Upon successful loading of the module specified by *module* filename, a reference to the VMMain function is returned, upon failure NULL is returned.

Name

VMUnloadModule – Unloads the previously loaded module.

Synopsys

```
#include "VirtualMachine.h"

void VMUnloadModule(void);
```

Description

VMUnloadModule() unloads the previously loaded module. The source for VMUnloadModule is provided in VirtualMachineUtils.c

Return Value

N/A

Name

VMTickMS – Retrieves milliseconds between ticks of the virtual machine.

Synopsys

```
#include "VirtualMachine.h"

TVMStatus VMTickMS(int *tickmsref);
```

Description

VMTickMS() puts tick time interval in milliseconds in the location specified by *tickmsref*. This is the value *tickms* from the previous call to VMStart().

Return Value

Upon successful retrieval of the tick interval from the virtual machine, VMTickMS() returns VM_STATUS_SUCCESS. If the parameter *tickmsref* is NULL, VM_STATUS_ERROR_INVALID_PARAMETER is returned.

Name

VMTickCount – Retrieves number of ticks that have occurred since the start of the virtual machine.

Synopsys

```
#include "VirtualMachine.h"

TVMStatus VMTickCount(TVMTickRef tickref);
```

Description

VMTickCount() puts the number of ticks that have occurred since the start of the virtual machine in the location specified by *tickref*.

Return Value

Upon successful retrieval of the number of elapsed ticks, VMTickCount() returns VM_STATUS_SUCCESS. If the parameter *tickref* is NULL, VM_STATUS_ERROR_INVALID_PARAMETER is returned.

Name

VMThreadCreate – Creates a thread in the virtual machine.

Synopsys

```
#include "VirtualMachine.h"
typedef void (*TVMThreadEntry)(void *);

TVMStatus VMThreadCreate(TVMThreadEntry entry, void *param, TVMMemorySize
memsize, TVMThreadPriority prio, TVMThreadIDRef tid);
```

Description

VMThreadCreate() creates a thread in the virtual machine. Once created the thread is in the dead state VM_THREAD_STATE_DEAD. The *entry* parameter specifies the function of the thread, and *param* specifies the parameter that is passed to the function. The size of the threads stack is specified by *memsize*, and the priority is specified by *prio*. The thread identifier is put into the location specified by the *tid* parameter.

Return Value

Upon successful creation of the thread VMThreadCreate() returns VM_STATUS_SUCCESS. VMThreadCreate() returns VM_STATUS_ERROR_INVALID_PARAMETER if either *entry* or *tid* is NULL.

Name

VMThreadDelete – Deletes a dead thread from the virtual machine.

Synopsys

```
#include "VirtualMachine.h"

TVMStatus VMThreadDelete(TVMThreadID thread);
```

Description

VMThreadDelete() deletes the dead thread specified by *thread* parameter from the virtual machine.

Return Value

Upon successful deletion of the thread from the virtual machine, VMThreadDelete() returns VM_STATUS_SUCCESS. If the thread specified by the thread identifier *thread* does not exist, VM_STATUS_ERROR_INVALID_ID is returned. If the thread does exist, but is not in the dead state VM_THREAD_STATE_DEAD, VM_STATUS_ERROR_INVALID_STATE is returned.

Name

VMThreadActivate – Activates a dead thread in the virtual machine.

Synopsys

```
#include "VirtualMachine.h"

TVMStatus VMThreadActivate(TVMThreadID thread);
```

Description

VMThreadActivate() activates the dead thread specified by *thread* parameter in the virtual machine. After activation the thread enters the ready state VM_THREAD_STATE_READY, and must begin at the *entry* function specified.

Return Value

Upon successful activation of the thread in the virtual machine, VMThreadActivate() returns VM_STATUS_SUCCESS. If the thread specified by the thread identifier *thread* does not exist, VM_STATUS_ERROR_INVALID_ID is returned. If the thread does exist, but is not in the dead state VM_THREAD_STATE_DEAD, VM_STATUS_ERROR_INVALID_STATE is returned.

Name

VMThreadTerminate– Terminates a thread in the virtual machine.

Synopsys

```
#include "VirtualMachine.h"

TVMStatus VMThreadTerminate(TVMThreadID thread);
```

Description

VMThreadTerminate() terminates the thread specified by *thread* parameter in the virtual machine. After termination the thread enters the state VM_THREAD_STATE_DEAD. The termination of a thread can trigger another thread to be scheduled.

Return Value

Upon successful termination of the thread in the virtual machine, VMThreadTerminate() returns VM_STATUS_SUCCESS. If the thread specified by the thread identifier *thread* does not exist, VM_STATUS_ERROR_INVALID_ID is returned. If the thread does exist, but is in the dead state VM_THREAD_STATE_DEAD, VM_STATUS_ERROR_INVALID_STATE is returned.

Name

VMThreadID – Retrieves thread identifier of the current operating thread.

Synopsys

```
#include "VirtualMachine.h"

TVMStatus VMThreadID(TVMThreadIDRef threadref);
```

Description

VMThreadID() puts the thread identifier of the currently running thread in the location specified by *threadref*.

Return Value

Upon successful retrieval of the thread identifier from the virtual machine, VMThreadID() returns VM_STATUS_SUCCESS. If the parameter *threadref* is NULL, VM_STATUS_ERROR_INVALID_PARAMETER is returned.

Name

VMThreadState – Retrieves the state of a thread in the virtual machine.

Synopsys

```
#include "VirtualMachine.h"
#define VM_THREAD_STATE_DEAD       ((TVMThreadState)0x00)
#define VM_THREAD_STATE_RUNNING    ((TVMThreadState)0x01)
#define VM_THREAD_STATE_READY      ((TVMThreadState)0x02)
#define VM_THREAD_STATE_WAITING    ((TVMThreadState)0x03)

TVMStatus VMThreadState(TVMThreadID thread, TVMThreadStateRef state);
```

Description

VMThreadState() retrieves the state of the thread specified by *thread* and places the state in the location specified by *state*.

Return Value

Upon successful retrieval of the thread state from the virtual machine, VMThreadState() returns VM_STATUS_SUCCESS. If the thread specified by the thread identifier *thread* does not exist, VM_STATUS_ERROR_INVALID_ID is returned. If the parameter *stateref* is NULL, VM_STATUS_ERROR_INVALID_PARAMETER is returned.

Name

VMThreadSleep– Puts the current thread in the virtual machine to sleep.

Synopsys

```
#include "VirtualMachine.h"
#define VM_TIMEOUT_INFINITE  ((TVMTick)0)
#define VM_TIMEOUT_IMMEDIATE ((TVMTick)-1)

TVMStatus VMThreadSleep(TVMTick tick);
```

Description

VMThreadSleep() puts the currently running thread to sleep for *tick* ticks. If tick is specified as VM_TIMEOUT_IMMEDIATE the current process yields the remainder of its processing quantum to the next ready process of equal priority.

Return Value

Upon successful sleep of the currently running thread, VMThreadSleep() returns VM_STATUS_SUCCESS. If the sleep duration *tick* specified is VM_TIMEOUT_INFINITE, VM_STATUS_ERROR_INVALID_PARAMETER is returned.

Name

VMPrint, VMPrintError, and VMFilePrint – Prints out to a file.

Synopsys

```
#include "VirtualMachine.h"

#define VMPrint(format, ...)
    VMFilePrint ( 1,  format, ##__VA_ARGS__)
#define VMPrintError(format, ...)
    VMFilePrint ( 2,  format, ##__VA_ARGS__)
TVMStatus VMFilePrint(int filedescriptor, const char *format, ...);
```

Description

VMFilePrint() writes the C string pointed by *format* to the file specified by *filedescriptor*. If format includes format specifiers (subsequences beginning with %), the additional arguments following format are formatted and inserted in the resulting string replacing their respective specifiers. The VMPrint and VMPrintError macros have been provided as a convenience for calling VMFilePrint. The source code for VMFilePrint is provided in VirtualMachineUtils.c

Return Value

Upon successful writing out of the *format* string to the file VM_STATUS_SUCCESS is returned, upon failure VM_STATUS_FAILURE is returned.

Name

VMFileOpen – Opens and possibly creates a file in the 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.

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.

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.

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.

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.

Return Value

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

Name

MachineContextSave – Saves a machine context.

Synopsys

```
#include "Machine.h"
typedef struct{
    jmp_buf DJumpBuffer;
} SMachineContext, *SMachineContextRef;

#define MachineContextSave(mcntx) setjmp((mcntx)->DJumpBuffer)
```

Description

MachineContextSave() saves the machine context that is specified by the parameter *mcntx*.

Return Value

Upon successful saving of the context, MachineContextSave () returns 0.

Name

MachineContextRestore – Restores a machine context.

Synopsys

```
#include "Machine.h"
typedef struct{
    jmp_buf DJumpBuffer;
} SMachineContext, *SMachineContextRef;

#define MachineContextRestore(mcntx) longjmp((mcntx)->DJumpBuffer, 1)
```

Description

MachineContextRestore() restores a previously saved the machine context that is specified by the parameter *mcntx*.

Return Value

Upon successful restoring of the context, MachineContextRestore() should not return.

Name

MachineContextSwitch – Switches machine context.

Synopsys

```
#include "Machine.h"
typedef struct{
    jmp_buf DJumpBuffer;
} SMachineContext, *SMachineContextRef;

#define MachineContextSwitch (mcntxold,mcntxnew)
    if(setjmp((mcntxold)->DJumpBuffer) == 0)
       longjmp((mcntxnew)->DJumpBuffer, 1)
```

Description

MachineContextSwitch() switches context to a previously saved the machine context that is specified by the parameter *mcntxnew*, and stores the current context in the parameter specified by *mctxold*.

Return Value

Upon successful switching of the context, MachineContextRestore() should not return until the original context is restored.

Name

MachineContextCreate – Creates a machine context.

Synopsys

```
#include "Machine.h"
typedef struct{
    jmp_buf DJumpBuffer;
} SMachineContext, *SMachineContextRef;

void MachineContextCreate(SMachineContextRef mcntxref,
void (*entry)(void *), void *param, void *stackaddr, size_t stacksize);
```

Description

MachineContextCreate() creates a context that will enter in the function specified by *entry* and passing it the parameter *param*. The contexts stack of size *stacksize* must be specified by the *stackaddr* parameter. The newly created context will be stored in the *mcntxref* parameter, this context can be used in subsequent calls to MachineContextRestore(), or MachineContextSwitch().

Return Value

N/A

Name

MachineInitialize – Initializes the machine abstraction layer.

Synopsys

```
#include "Machine.h"

void MachineInitialize(void);
```

Description

MachineInitialize() initializes the machine abstraction layer.

Return Value

N/A

Name

MachineTerminate – Terminates the machine abstraction layer.

Synopsys

```
#include "Machine.h"

void MachineTerminate(void);
```

Description

MachineTerminate() terminates the machine abstraction layer. This closes down the cooperative process that is executing the machine abstraction.

Return Value

N/A

Name

MachineEnableSignals – Enables all signals.

Synopsys

```
#include "Machine.h"

void MachineEnableSignals(void);
```

Description

MachineEnableSignals() enables all signals so that the virtual machine may be “interrupted” asynchronously.

Return Value

N/A

Name

MachineSuspendSignals – Suspends all signals.

Synopsys

```
#include "Machine.h"
typedef sigset_t TMachineSignalState, *TMachineSignalStateRef;

void MachineSuspendSignals(TMachineSignalStateRef sigstate);
```

Description

MachineSuspendSignals() suspends all signals so that the virtual machine will not be "interrupted" asynchronously. The current state of the signal mask will be placed in the location specified by the parameter *sigstate*. This signal state can be restored by a call to MachineResumeSignals().

Return Value

N/A

Name

MachineResumeSignals – Resumes signal state.

Synopsys

```
#include "Machine.h"
typedef sigset_t TMachineSignalState, *TMachineSignalStateRef;

void MachineResumeSignals(TMachineSignalStateRef sigstate);
```

Description

MachineResumeSignals() resumes all signals that were enabled when previous call to MachineSuspendSignals() was called so that the virtual machine will my be “interrupted” asynchronously. The signal mask in the location specified by the parameter *sigstate* will be restored to the virtual machine. This signal state should have been initialized by a previous call to MachineSuspendSignals().

Return Value

N/A

Name

MachineRequestAlarm – Requests periodic alarm callback.

Synopsys

```
#include "Machine.h"
typedef void (*TMachineAlarmCallback)(void *calldata);

void MachineRequestAlarm(useconds_t usec,
TMachineAlarmCallback callback, void *calldata);
```

Description

MachineRequestAlarm() requests periodic alarm callback from the machine abstraction layer. The callback function specified by the *callback* parameter will be called at a period of *usec* microseconds being passed the parameter specified by *calldata*. The alarm callback can be canceled by calling MachineRequestAlarm() with a parameter of 0 *usec*.

Return Value

N/A

Name

MachineFileOpen – Opens a file with the machine abstraction layer.

Synopsys

```
#include "Machine.h"
typedef void (*TMachineFileCallback)(void *calldata, int result);

void MachineFileOpen(const char *filename, int flags, int mode,
TMachineFileCallback callback, void *calldata);
```

Description

MachineFileOpen() 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 passed in to the *callback* function as the *result*. The *calldata* parameter will also be passed into the *callback* function upon completion of the open file request. The flags and mode values follow the same format as that of open system call. The *result* returned can be used in subsequent calls to MachineFileClose(), MachineFileRead(), MachineFileWrite(), and MachineFileSeek(). MachineFileOpen() should return immediately, but will call the *callback* function asynchronously when completed. Upon failure the *result* will be less than zero.

Return Value

N/A

Name

MachineFileRead – Reads from a file in the machine abstraction.

Synopsys

```
#include "Machine.h"
typedef void (*TMachineFileCallback)(void *calldata, int result);

void MachineFileRead(int fd, void *data, int length, TMachineFileCallback
callback, void *calldata);
```

Description

MachineFileRead() attempts to read the number of bytes specified in by *length* into the location specified by *data* from the file specified by *fd*. The *fd* should have been obtained by a previous call to MachineFileOpen(). The actual number of bytes transferred will be returned in the *result* parameter when the *callback* function is called. Upon failure the *result* will be less than zero. The *calldata* parameter will also be passed into the *callback* function upon completion of the read file request. MachineFileRead () should return immediately, but will call the *callback* function asynchronously when completed.

Return Value

N/A

Name

MachineFileWrite – Writes to a file in the machine abstraction.

Synopsys

```
#include "Machine.h"
typedef void (*TMachineFileCallback)(void *calldata, int result);

void MachineFileWrite(int fd, void *data, int length, TMachineFileCallback
callback, void *calldata);
```

Description

MachineFileWrite() attempts to write the number of bytes specified in by *length* into the location specified by *data* to the file specified by *fd*. The *fd* should have been obtained by a previous call to MachineFileOpen(). The actual number of bytes transferred will be returned in the *result* parameter when the *callback* function is called. Upon failure the *result* will be less than zero. The *calldata* parameter will also be passed into the *callback* function upon completion of the write file request. MachineFileWrite() should return immediately, but will call the *callback* function asynchronously when completed.

Return Value

N/A

Name

MachineFileSeek – Seeks in a file in the machine abstraction.

Synopsys

```
#include "Machine.h"
typedef void (*TMachineFileCallback)(void *calldata, int result);

void MachineFileSeek(int fd, int offset, int whence, TMachineFileCallback
callback, void *calldata);
```

Description

MachineFileSeek() attempts to seek the number of bytes specified in by *offset* from the location specified by *whence* in the file specified by *fd*. The *fd* should have been obtained by a previous call to MachineFileOpen(). The actual offset in the file will be returned in the *result* parameter when the *callback* function is called. Upon failure the *result* will be less than zero. The *calldata* parameter will also be passed into the *callback* function upon completion of the seek file request. MachineFileSeek() should return immediately, but will call the *callback* function asynchronously when completed.

Return Value

N/A

Name

MachineFileClose – Closes a file in the machine abstraction layer.

Synopsys

```
#include "Machine.h"
typedef void (*TMachineFileCallback)(void *calldata, int result);

void MachineFileClose(int fd, TMachineFileCallback callback,
void *calldata);
```

Description

MachineFileClose() attempts to close the file specified by *fd*. The *fd* should have been obtained by a previous call to MachineFileOpen(). The *result* parameter when the *callback* function is called will be zero upon success; upon failure the *result* will be less than zero. The *calldata* parameter will also be passed into the *callback* function upon completion of the file closure request. MachineFileClose() should return immediately, but will call the *callback* function asynchronously when completed.

Return Value

N/A