Project 3

Due November 13, 2018 at 11:59 PM

You will be working with a partner for this project, you specify your partner in your readme file. This specification is subject to change at anytime for additional clarification. For this project, you and a partner 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. Submit only one copy of your project per pair on Canvas.

You will be continuing the development of your virtual machine. You will be adding to the virtual machine a mutex API which provides a synchronization mechanism. You will also be adding the ability to create, allocate, deallocate, query, and delete memory pools. Except were specified in this description the API will remain as stated in the Project 2 description. The memory pools allow for dynamic memory allocation from specified pools of memory, there is a main system memory pool (VM_MEMORY_POOL_ID_SYSTEM) that is dynamically allocated by the virtual machine at the beginning of execution. The stack space for each of the threads must be allocated from this memory pool. Space is allocated from the memory pools using a first fit algorithm.

The communication between the virtual machine and machine has changed. The MachineFileRead and MachineFileWrite functions require that shared memory locations be used to transfer data to/from the machine. In addition the maximum amount of data transferred must be limited to 512 bytes. VMFileRead and VMFileWrite **must** still be able to transfer up any number of bytes specified.

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, -h, and -s; -t specifies the tick time in millisecond, -h specifies the size of the heap this is the size of the VM_MEMORY_POOL_ID_SYSTEM memory pool, and -s specifies the size of the shared memory used between the virtual machine and machine. By default the tick time is set to 10ms, for debugging purposes you can increase these values to slow the running of the vm. The size of the heap by default is 16MiB, and the shared memory is 16KiB. When specifying the application name the ./ should be prepended otherwise vm may fail to load the shared object file.

The function specifications for both the virtual machine and machine are provided in the subsequent pages, those that did not change from project 2 are not listed.

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.

Name

VMStart – Start the virtual machine.

Synopsys

```
#include "VirtualMachine.h"

TVMStatus VMStart(int tickms, TVMMemorySize heapsize, TVMMemorySize
sharedsize, 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 heap size of the virtual machine is specified by *heapsize*. The heap is accessed by the applications through the VM_MEMORY_POOL_ID_SYSTEM memory pool. The size of the shared memory space between the virtual machine and the machine is specified by the *sharedsize*.

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

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, **and must release any mutexes that it currently holds**. 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

VMMutexCreate– Creates a mutex in the virtual machine.

Synopsys

```
#include "VirtualMachine.h"

TVMStatus VMMutexCreate(TVMMutexIDRef mutexref);
```

Description

VMMutexCreate() creates a mutex in the virtual machine. Once created the mutex is in the unlocked state. The mutex identifier is put into the location specified by the *mutexref* parameter.

Return Value

Upon successful creation of the thread VMMutexCreate() returns VM_STATUS_SUCCESS. VMMutexCreate() returns VM_STATUS_ERROR_INVALID_PARAMETER if either *mutexref* is NULL.

Name

VMMutexDelete – Deletes a mutex from the virtual machine.

Synopsys

```
#include "VirtualMachine.h"

TVMStatus VMMutexDelete(TVMMutexID mutex);
```

Description

VMMutexDelete() deletes the unlocked mutex specified by *mutex* parameter from the virtual machine.

Return Value

Upon successful deletion of the thread from the virtual machine, VMMutexDelete() returns VM_STATUS_SUCCESS. If the mutex specified by the thread identifier *mutex* does not exist, VM_STATUS_ERROR_INVALID_ID is returned. If the mutex does exist, but is currently held by a thread, VM_STATUS_ERROR_INVALID_STATE is returned.

Name

VMMutexQuery– Queries the owner of a mutex in the virtual machine.

Synopsys

```
#include "VirtualMachine.h"

TVMStatus VMMutexQuery(TVMMutexID mutex, TVMThreadIDRef ownerref);
```

Description

VMMutexQuery() retrieves the owner of the mutex specified by *mutex* and places the thread identifier of owner in the location specified by *ownerref*. If the mutex is currently unlocked, VM_THREAD_ID_INVALID returned as the owner.

Return Value

Upon successful querying of the mutex owner from the virtual machine, VMMutexQuery() returns VM_STATUS_SUCCESS. If the mutex specified by the mutex identifier *mutex* does not exist, VM_STATUS_ERROR_INVALID_ID is returned. If the parameter *ownerref* is NULL, VM_STATUS_ERROR_INVALID_PARAMETER is returned.

Name

VMMutexAcquire – Locks the mutex.

Synopsys

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

TVMStatus VMMutexAcquire(TVMMutexID mutex, TVMTick timeout);
```

Description

VMMutexAcquire() attempts to lock the mutex specified by *mutex* waiting up to *timeout* ticks. If *timeout* is specified as VM_TIMEOUT_IMMEDIATE the current returns immediately if the mutex is already locked. If *timeout* is specified as VM_TIMEOUT_INFINITE the thread will block until the *mutex* is acquired.

Return Value

Upon successful acquisition of the currently running thread, VMMutexAcquire() returns VM_STATUS_SUCCESS. If the *timeout* expires prior to the acquisition of the mutex, VM_STATUS_FAILURE is returned. If the mutex specified by the mutex identifier *mutex* does not exist, VM_STATUS_ERROR_INVALID_ID is returned.

Name

VMMutexRelease – Releases a mutex held by the currently running thread.

Synopsys

```
#include "VirtualMachine.h"

TVMStatus VMMutexRelease(TVMMutexID mutex);
```

Description

VMMutexRelease() releases the mutex specified by the *mutex* parameter that is currently held by the running thread. Release of the mutex may cause another higher priority thread to be scheduled if it acquires the newly released mutex.

Return Value

Upon successful release of the mutex, VMMutexRelease() returns VM_STATUS_SUCCESS. If the mutex specified by the mutex identifier *mutex* does not exist, VM_STATUS_ERROR_INVALID_ID is returned. If the mutex specified by the mutex identifier *mutex* does exist, but is not currently held by the running thread, VM_STATUS_ERROR_INVALID_STATE is returned.

Name

VMMemoryPoolCreate – Creates a memory pool from an array of memory.

Synopsys

```
#include "VirtualMachine.h"

TVMStatus VMMemoryPoolCreate(void *base, TVMMemorySize size,
TVMMemoryPoolIDRef memory);
```

Description

VMMemoryPoolCreate() creates a memory pool from an array of memory. The base and size of the memory array are specified by *base* and *size* parameters respectively. The memory pool identifier is put into the location specified by the *memory* parameter.

Return Value

Upon successful creation of the memory pool, VMMemoryPoolCreate() will return VM_STATUS_SUCCESS. If the *base* or *memory* are NULL, or *size* is zero VM_STATUS_ERROR_INVALID_PARAMETER is returned.

Name

VMMemoryPoolDelete – Deletes a memory pool from the virtual machine.

Synopsys

```
#include "VirtualMachine.h"

TVMStatus VMMemoryPoolDelete(TVMMemoryPoolID memory);
```

Description

VMMemoryPoolDelete() deletes a memory pool that has no memory allocated from the pool. The memory pool to delete is specified by the *memory* parameter.

Return Value

Upon successful deletion of the memory pool, VMMemoryPoolDelete() will return VM_STATUS_SUCCESS. If the memory pool specified by *memory* is not a valid memory pool, VM_STATUS_ERROR_INVALID_PARAMETER is returned. If any memory has been allocated from the pool and not deallocated, then VM_STATUS_ERROR_INVALID_STATE is returned.

Name

VMMemoryPoolQuery – Queries the available space in the memory pool.

Synopsys

```
#include "VirtualMachine.h"

TVMStatus VMMemoryPoolQuery(TVMMemoryPoolID memory,
TVMMemorySizeRef bytesleft);
```

Description

VMMemoryPoolQuery() queries a memory pool for the available memory left in the pool. The memory pool to query is specified by the *memory* parameter. The space left unallocated in the memory pool is placed in the location specified by *bytesleft*.

Return Value

Upon successful querying of the memory pool, VMMemoryPoolQuery() will return VM_STATUS_SUCCESS. If the memory pool specified by *memory* is not a valid memory pool or *bytesleft* is NULL, VM_STATUS_ERROR_INVALID_PARAMETER is returned.

Name

VMMemoryPoolAllocate – Allocates memory from the memory pool.

Synopsys

```
#include "VirtualMachine.h"

TVMStatus VMMemoryPoolAllocate(TVMMemoryPoolID memory,
TVMMemorySize size, void **pointer);
```

Description

VMMemoryPoolAllocate() allocates memory from the memory pool. The memory pool to allocate from is specified by the *memory* parameter. The size of the allocation is specified by *size* and the base of the allocated array is put in the location specified by *pointer*. The allocated size will be rounded to the next multiple of 64 bytes that is greater than or equal to the *size* parameter.

Return Value

Upon successful allocation from the memory pool, VMMemoryPoolAllocate() will return VM_STATUS_SUCCESS. If the memory pool specified by *memory* is not a valid memory pool, *size* is zero, or *pointer* is NULL, VM_STATUS_ERROR_INVALID_PARAMETER is returned. If the memory pool does not have sufficient memory to allocate the array of *size* bytes, VM_STATUS_ERROR_INSUFFICIENT_RESOURCES is returned.

Name

VMMemoryPoolDeallocate – Deallocates memory to the memory pool.

Synopsys

```
#include "VirtualMachine.h"

TVMStatus VMMemoryPoolDeallocate(TVMMemoryPoolID memory,
void *pointer);
```

Description

VMMemoryPoolDeallocate() deallocates memory to the memory pool. The memory pool to deallocate to is specified by the *memory* parameter. The base of the previously allocated array is specified by *pointer*.

Return Value

Upon successful deallocation from the memory pool, VMMemoryPoolDeallocate() will return VM_STATUS_SUCCESS. If the memory pool specified by *memory* is not a valid memory pool, or *pointer* is NULL, VM_STATUS_ERROR_INVALID_PARAMETER is returned. If *pointer* does not specify a memory location that was previously allocated from the memory pool, VM_STATUS_ERROR_INVALID_PARAMETER is returned.

Name

MachineInitialize – Initializes the machine abstraction layer.

Synopsys

```
#include "Machine.h"

void *MachineInitialize(size_t sharesize);
```

Description

MachineInitialize() initializes the machine abstraction layer. The *sharesize* parameter specifies the size of the shared memory location to be used by the machine. The size of the shared memory will be set to an integral number of pages (4096 bytes) that covers the size of *sharesize*.

Return Value

Upon successful initialization MachineInitialize returns the base address of the shared memory. NULL is returned if the machine has already been initialized. If the memory queues or shared memory fail to be allocated the program exits.

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*. If the *data* value is not a location in the shared memory, MachineFileRead will fail; in addition if length is greater than 512 bytes MachineFileRead will also fail. 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*. If the *data* value is not a location in the shared memory, MachineFileWrite will fail; in addition if length is greater than 512 bytes MachineFileWrite will also fail. 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