CSC 501 Program *Threading library*

Assignment:

Implement a *non pre-emptive* user-level threading library.

Due:

See home page.

Description:

Implement a non pre-emptive user-level threads library *mythread.a* with the following routines.

Thread routines.

MyThread ***MyThreadCreate*** *(void(*start_funct)(void *), void *args)*
: This routine creates a new *MyThread*. The parameter *start_func* is the function in which the new thread starts executing. The parameter *args* is passed to the start function. This routine does *not* pre-empt the invoking thread. In others words the parent (invoking) thread will continue to run; the child thread will sit in the ready queue.

void ***MyThreadYield****(void)*
: Suspends execution of invoking thread and yield to another thread. The invoking thread remains ready to execute—it is not blocked. Thus, if there is no other ready thread, the invoking thread will continue to execute.

int ***MyThreadJoin****(MyThread thread)*
: Joins the invoking function with the specified child thread. If the child has already terminated, do not block. Note: A child may have terminated without the parent having joined with it. Returns 0 on success (after any necessary blocking). It returns -1 on failure. Failure occurs if specified thread is not an immediate child of invoking thread.

void ***MyThreadJoinAll****(void)*
: Waits until all children have terminated. Returns immediately if there are no *active* children.

void ***MyThreadExit****(void)*
: Terminates the invoking thread. ***Note:*** all *MyThread*s are required to invoke this function. Do not allow functions to "fall out" of the start function.

Semaphore routines.

MySemaphore ***MySemaphoreInit****(int initialValue)*
: Create a semaphore. Set the initial value to *initialValue*, which must be non-negative. A positive initial value has the same effect as invoking

MySemaphoreSignal the same number of times.

void ***MySemaphoreSignal****(MySemaphore sem)*
Signal semaphore *sem*. The invoking thread is not pre-empted.

void ***MySemaphoreWait****(MySemaphore sem)*
Wait on semaphore *sem*.

int ***MySemaphoreDestroy****(MySemaphore sem)*
Destroy semaphore *sem*. Do not destroy semaphore if any threads are blocked on the queue. Return 0 on success, -1 on failure.

Unix process routines.

The Unix process in which the user-level threads run is not a *MyThread*. Therefore, it will not be placed on the queue of *MyThreads*. Instead it will create the first *MyThread* and relinquish the processor to the *MyThread* engine.

The following routine may be executed only by the Unix process.

void ***MyThreadInit*** *(void(*start_funct)(void *), void *args)*
This routine is called before any other *MyThread* call. It is invoked only by the Unix process. It is similar to invoking *MyThreadCreate* immediately followed by *MyThreadJoinAll*. The *MyThread* created is the oldest ancestor of all *MyThread*s—it is the "main" *MyThread*. This routine can only be invoked once. It returns when there are no threads available to run (i.e., the thread ready queue is empty.

Notes:

- Use a FIFO (first-in, first-out) scheduling policy for threads on the ready queue.
- Make all the routines available via a library. Use the *ar* command to make a library. It will be used similar to this:

```
ar rcs mythread.a file1.o file2.o file3.o
```

- All programs that use this library will include a header file (*mythread.h*) that defines all the interfaces and data structures that are needed—but no more.
- This library does not have to be *thread-safe* and will only execute in a single-threaded (OS) process. Assume that an internal thread operation cannot be interrupted by another thread operation. (E.g., *MyThreadExit* will not be interrupted by *MyThreadYield*.) That means that the library does not have to acquire locks to protect the **internal** data of the thread library. (A user may still have to acquire semaphores to protect user data.)
- The interface only allows one parameter (a **void ***) to be passed to a *MyThread*. This is sufficient. One can build wrapper functions that *pack* and *unpack* an arbitrary parameter list into an object pointed to by a **void ***. This program gives an example of that. It is not a complete program—some of the logic is absent. It is only provided to illustrate how to use the interface.
- Allocate at least 8KB for each thread stack.

Turn In:

Be sure to turnin **all** the files needed. Include a `Makefile` that creates a library (a ".a" file) called `mythread.a`. An appropriate penalty will be assessed if this is not so.

Also, denote any resources used--including other students--in a file named **REFERENCES**.

Resources:

Here is the header file for the library. This file is the *de facto* interface for the library. **Do not edit it.**

Additionally, here is a program that uses the library. **The program comes with no warranty at all.** It is provided as an illustration only.

Some slides describing the assignment.

Testing:

This assignment will be compiled and tested on a VCL Linux machine.

Notes:

1. **Do not** use the *pthread* library or any similar threading package.
2. Instead use the "context" routines: *getcontext(2), setcontext(2), makecontext(3)*, and *swapcontext(3)*
3. During testing many processes (or threads) will be created on machines used by others. Take care when testing.
4. Test programs Here are four example test cases. We think they are correct. However, there is no warranty. Alert us if you find any mistakes. We will use many more test cases during grading; do not let success on these codes lull you in to complacency.
5. The code will be tested on the VCL image named here.

Grading:

The weighting of this assignment is given in policies.

Extra Credit

Extra credit of 5% is available. Two restrictions were designed into the assignment to keep it simple: (1) Each thread must end in *MyThreadExit* and (2) The initial "UNIX" process context is not used as a thread context. To receive the extra credit, students must provide the following work-arounds to those restrictions. Specifically, the library must do the following.

1. Work correctly when threads do not end with *MyThreadExit*.
2. Create a rountine *int MyThreadInitExtra(void)* that initializes the threads package **and** converts the UNIX process context into a MyThread context.

```
...

MyThreadInitExtra(...);
// code here runs in thread context.
// it is the root or "oldest ancestor" thread
MyThreadCreate(...); // Creates a new thread
MyThreadYield(...);  // Yield to new thread just created

...
```

3. Submit a second header file (mythreadextra.h) that contains everything in the original header plus the definition of MyThreadInitExtra().