Lab2 – Simple thread library

This laboratory requires some amount of low-level debugging which is very time consuming. So better get started.

Objective:

- To enhance your understanding of threads and their implementation.
- To enhance your low-level programming skills and expose you to some of the commonly used techniques in doing so.
- To expose you to calling semantics of C language on x86_64.

Design goals:

Your implementation should have the following features.

- You should be able to create any number of threads. In other words, data structure used to store information about threads should **not** be an array.
- When a new thread is created, you may assume that the environments is already is setup. In other words you can execute the given function directly without any initialization code.
- When a thread exit, it should be removed from the system. However, note that all the threads in this particular implementation is multiplexed on the same kernel level thread. So the exiting thread should not inform the kernel that it is doing so (which is done via the *exit* system call). If you do so the kernel unaware of user-level threads will kill all of them.
- You may use the given *malloc_stack* function without any modifications to allocate memory for the stack region of a thread. If you are modifying do it at your own risk.

Tasks:

There are three tasks. **Task 1 is not marked**, but you should do that during the lab itself; it essentially contains warmup exercise for what is to come. Tasks 2 should be completed and the coding should be submitted before the deadline.

Task 1: Answer the following questions looking at the provided code (*l4.tgz available via FEeLS*)

1. What is the purpose of following C flags: -I./ -Wall -Wstrict-prototypes -Werror (see *Makefile*)
2. Explain the behavior of #ifdef and #define statements (in *threadlib.h*)
3. Explain how a function written in assembly can be called by a function written in C.
4. What is meant by size aligned?
5. Look at the use of count variable in the main.c file. It is shared between the user-level threads. Do you need to look that before accessing it? If not how is mutual exclusion guaranteed?
6. Can thread1 cause harm to thread2 by accessing its memory? Explain.
7. What is the purpose of *assert* function?

Task 2:

The *main.c* file contains two main functions – one is used for testing your code and once you are confident about the implementation you can use the other one. You can do this by removing the “define TESTING 1” line from the main.

When it comes to the thread switching you have two main parts – one figuring out what thread to run and second making that thread runnable. These two operations are done by two functions; *schedule* picks up the next thread to run and *switch_threads* and *machine_switch* does the actual switching. The main idea is to do much work as possible in the *switch_thread C* function and let the *machine_switch* do the remaining machine level switching (that is saving and restoring registers).

The *schedule* uses a simple policy – round-robin. You can implement other policies here as well. However not required for this laboratory. You may try it to improve your knowledge.

Once a thread has done its job it calls *delete_thread* function which should remove the thread from the queue of threads, free memory the thread used (for example stack space and memory for the TCB). It should notify the OS (via the *exit* system call) to reclaim all the resources only when the last thread in your application calls *delete_thread.*

The *stop_main* function is somewhat tricky to understand. When we run this application the kernel will create a kernel level thread and main will start running as this thread. Then main creates other threads by invoking functions from our library. Now unlike other threads we created (using the library function we have) we do not have enough information about the main thread and in fact we are going to steal the CPU time given to this thread and multiplex all other threads on it. We should drop this thread from the system and switch to a thread we created. You should be able to use the *machine_switch* (and may be *schedule*) with some hacks to achieve this.

On top of what is said here, the code contains lots of comments. You should first read the code (assuming you have read this document) and understand what is done in it. Then do a design and then start coding.

Ones your code works with the given (t*esting-)*main function test it with the (*production-)*main function. **If it works submit your work by creating a tarball containing all the files.**