EC440: Project 4 - Thread Local Storage

Project Goals

- To understand basic concepts of memory management
- To provide protected memory regions for threads

Collaboration policy

- You are encouraged to discuss this project with your classmates/instructors but are required to turn in your own solution.
- You must be able to fully explain your solution during oral examination.

Deadline

It is due on Tuesday, November 28th, 23:59 ET (no deadline extensions or late submissions).

Project Description

The goal of this project is to implement a library that provides protected memory regions for threads, which they can safely use as local storage. As you remember, all threads share the same memory address space. While it can be beneficial since it allows threads to easily share information, it can also be a problem when one thread accidentally modifies values that another thread has stored in a variable. To protect data from being overwritten by other threads, it would be convenient for each thread to possess a protected storage area that only this thread can read from and write to. We call this protected storage area *thread local storage*. Your task is to implement support for thread local storage, either on top of your existing user mode thread library or on top of Linux' pthread implementation. In both cases, your code will be a library that implements the functions and functionality introduced below.

To provide support for thread local storage (TLS), you are supposed to implement the following functions in a library:

```
int tls_create(unsigned int size)
```

This function creates a local storage area (LSA) for the currently executing thread that can hold (at least) *size* bytes. It returns 0 on success or -1 (error) if a thread already has a local storage which size is larger than 0 bytes. After a LSA has been created, it can be read and written by the following two functions.

```
int tls_write(unsigned int offset, unsigned int length, char *buffer)
```

This function reads *length* bytes, starting from the memory location pointed to by *buffer*, and writes them into the local storage area of the currently executing thread, starting at position *offset*. It returns 0 on success and -1 (error) if the function is asked to write more data than the LSA can hold (i.e., *offset* + *length* > *size of LSA*) or if the current thread has no LSA. Finally,

the function trusts that the buffer from which data is read holds at least *length* bytes. If not, the result of the call is undefined.

`int tls_read(unsigned int offset, unsigned int length, char *buffer)`

This function reads *length* bytes from the local storage area of the currently executing thread, starting at position *offset*, and writes them into the memory location pointed to by *buffer*. The function returns 0 on success or -1 (error) if it is asked to read past the end of the LSA (i.e., *offset* + *length* > *size of LSA*) or if the current thread has no LSA. Finally, the function trusts that the buffer into which data is written is large enough to hold at least *length* bytes. If not, the result of the call is undefined.

`int tls_destroy()`

This function frees a previously allocated local storage area of the currently executing thread. It returns 0 on success and -1(error) when the thread does not have a local storage area.

`int tls_clone(pthread_t tid)`

This function clones the local storage area of a target thread identified by *tid*. When a thread local storage is cloned, the content is **not** simply copied. Instead, the storage areas of both threads initially refer to the same memory location. Only when one thread writes to its own LSA (using the **`tls_write`** function), then the TLS library creates a private copy of the region that is written. Note, though, that the remaining, untouched areas still remain shared. This approach is called CoW (copy-on-write), and it is done to save memory space and to avoid unnecessary copy operations. The function returns 0 on success. It is an error when the target thread has no LSA, or when the currently executing thread already has a LSA. In both cases, the function returns -1.

Whenever a thread attempts to read from or write to any thread local storage area, including its own, *without* using the appropriate **`tls_read`** and **`tls_write`** functions, then this thread should be terminated (by calling **`pthread_exit`** on its behalf). The remaining threads continue to run unaffected.

Since we have to implement TLS in user space and cannot modify the operating system, we introduce the following two simplifications to make our lives easier:

First, whenever a thread calls **`tls_read`** or **`tls_write`**, you can temporarily unprotect this thread's local storage area. That is, when a thread A is executing one of these two functions, it would be possible for another thread B (that happens to interrupt thread A) to access A's local storage (and only that of thread A) without being terminated.

Second, we will see that most memory protection operations do not work with byte granularity but with page granularity. Thus, we relax the sharing requirement for **`tls_clone`**. Assume that thread B clones the local storage of thread A (which has a size of 2**page-size* -- where *page-size* is typically 4096 bytes and can be determined by calling the library routine **`getpagesize()`**). Now, let's assume that thread A writes one byte at the beginning of its

own local storage. Originally, we required that only this bytes is copied. For convenience, we relax this requirement and allow the entire first page (i.e., the entire first 4096 of the local storage) to be copied. The second page, however, still remains shared between thread A and thread B.

Note that it is possible that more than two threads share the same local storage. That is, multiple threads can **`tls_clone`** the LSA of the same target thread, and all these threads would then point to the same memory region (pages). When one thread write to its storage, only this thread gets its own copy. The remaining threads would still share the same region.

Implementation Hints

First, you need a way to create a local storage that cannot be directly accessed by a thread. To this end, we suggest that you use the library function **`mmap`**. **`mmap`** has two advantages: First, it allows one to obtain memory that is aligned with the start of a page, and the function allocates memory in multiples of the page size. Second, **`mmap`** allows you to create pages that have no read/write permissions, and thus, cannot be accessed arbitrarily.

Now that we have pages that are properly aligned and that cannot be accessed by any thread, the next question is how we can implement the **`tls_read`** and **`tls_write`** functions. For this, the library routine **`mprotect`** is very handy, which allows us to "unprotect" memory regions at the beginning of a read or write operation, and later "reprotect" it when we are done. Note that mprotect can only assign coarse-grain permissions at the level of individual pages. This is another reason why it is convenient to create the local storage area as multiples of memory pages.

It is important to observe that the local storage area of a thread can contain both shared pages and pages that are private copies. Hence, it is clear that these pages are *not always contiguous* in memory. As a result, when you perform read and write operations that span multiple pages of the local storage, you need to break up these operations into accesses to the individual pages.

Finally, the question arises what happens when a thread directly accesses a memory page (a LSA) that is protected. In this case, the operating system sends a signal (**`SIGSEGV`**) to the offending thread. Thus, you could install a signal handler for **`SIGSEGV`**, and whenever such a signal is caught, you simply terminate the currently running thread (by calling **`pthread_exit`**). Unfortunately, this is not correct, because there could be other reasons for a segmentation fault (a normal programming error). In this case, you do not want to only terminate the currently running thread, but kill the entire process and dump the core. Thus, your signal handler must be able to **distinguish** between a case in which a segmentation fault is caused by a thread that incorrectly accesses a local storage area, or a regular fault where no LSA is involved. To this end, we suggest that you look closely at the manual page for **`sigaction`** and try to find how the **`struct siginfo_t`** might help you to achieve your goal.

Submission Guidelines

- Your tls library must be written in C and run on Linux. It must compile without any errors when compiled with -Wall -Werror and run on ec440.bu.edu.
- Your makefile should compile your source files into an object tls.o file. If you want to use lpthread library, compile it with

gcc -Wall -Werror -c -lpthread -o tls.o tls.c

- In your home directory create a folder *project4* and place *README*, *makefile* and all of your source files there. Switch to the *project4* directory and execute **`submit4`**
- A confirmation mail of your submission is sent to your account on ec440.bu.edu. You can read this mail by executing **`mail`**.
- In the *README* file explain what you did. If you had problems, tell us why and what.
- Create a file called TIME and store in it the number of hours you invested into the homework (only needs to be updated for your last submission)
- You are allowed to resubmit your files. The latest submission before the deadline will be graded.

Oral Examination

- You are required to meet with one member of the course staff (excluding Prof. Egele) during office hours to explain your solution.