EC440: Project 4 – Copy On Write

Project Goals:

To understand basic concepts of memory management by implementing your own implementation of copy on write (key functionality for fork), handle page faults, and implement something very much like what kernel's do on copy to/from user space.

Collaboration Policy:

You are encouraged to discuss this project with your classmates and instructors. You **must** develop your own solution. While you **must not** share your thread-local storage code with other students, you **are encouraged** to share test case code that you developed to test your solution for this assignment.

Deadline:

Project 4 is due April 11 at 4:30 PM EDT.

Project Description:

In this project, we are implementing a library that provides protected memory regions for threads, which can be used for thread local storage. Recall that normally all threads share the same memory address space. While this sharing is helpful for programs that need to share information across threads and don't want to repeatedly copy every modification back and forth, it can be problematic when one thread modifies information that another thread assumed would remain unmodified.

Wouldn't it be great if you could share data across threads, but also ensure that any modifications to the data made by one thread do not impact another thread? This is where *copy-on-write (COW) thread local storage (TLS)* comes in!

- **Thread-local storage** (TLS) means that we provide a way for each thread to have a reserved part of memory. Other threads will not be able to successfully access that memory unless it is explicitly shared by a *clone* operation. Each thread has its own Local Storage Area (LSA).
- **Copy-on-write** (COW) means that cloned data will generally reference the same location in memory. *But* once there is any write to the cloned data, a copy of the data is used for the write. Furthermore, subsequent reads of the data from the writer will see the modified data, and references by other readers of the data will still reference the unmodified memory.

For this assignment, you will implement interfaces to interact with copy-on-write thread-local storage. Your TLS implementation should be agnostic of the current pthread implementation (see the hints section!). That is, it should not rely on the internal state of your threads.c project.

Required Functions to Implement

You will implement functions to support TLS with CoW.

Mutex Functions

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

The `tls_create()` creates a local storage area (LSA) for the currently-executing thread. This LSA must hold at least `size` bytes.

This function returns 0 on success, and -1 for errors. It is an error if a thread already has more than 0 bytes of local storage.

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

The `tls_write()` writes data to the current thread's local storage. The written data comes from `length` bytes starting at `buffer`. The data is written to a location in the local storage area `offset` bytes from the start of the LSA.

This function returns 0 on success and -1 when an error occurs. It is an 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.

This function can assume that `buffer` 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);
```

The `tls_read()` function reads data from the current thread's local storage. The read data must be copied to `length` bytes starting at `buffer`. The data is read from a location in the local storage area `offset` bytes from the start of the LSA.

This function returns 0 on success and -1 when an error occurs. It is an error if the function is asked to read more data than the LSA can hold (i.e., offset+length>size of LSA) or if the current thread has no LSA.

This function can assume that `buffer` holds at least `length` bytes. If not, the result of the call is undefined.

`int tls_destroy();`

The `tls_destroy()` function frees a previously-allocated LSA of the current thread. It returns 0 on success and -1 when an error occurs. It is an error if the thread does not have an LSA.

`int tls_clone(pthread_t` *`tid`***`);`**

The `tls_clone()` function clones the local storage area of a target thread, `tid`. When a LSA is cloned, the content is **not** simply copied. Instead, the storage areas of both threads initially refer to the same memory location. When one thread writes to its own LSA (using `tls_write`), the TLS library creates a private copy of the region that is written. Note that the remaining unmodified regions of the TLS remain shared. This approach is called copy on write (CoW), and saves memory space by avoiding unnecessary copy operations.

The function returns 0 on success, and -1 when an error occurs. It is an error when the target thread has no LSA, or when the current thread already has an LSA.

Whenever a thread attempts to read from or write to any LSA (even its own) without using the appropriate tls_read and tls_write functions, then this thread should be terminated (by calling `pthread_exit`). Remaining threads should 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:

1. Whenever a thread calls `tls_read` or `tls_write`, you can temporarily unprotect this thread's local storage area. That is, when 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. This is not typically desirable behavior for this kind of library, but it is allowed behavior to significantly simplify your implementation.
2. As we have learned, memory protection operations do not work with byte granularity but with page granularity. Thus, we relax the sharing requirement for `tls_clone`, to work on page granularity. 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 TLS. Originally, we required that only those bytes were copied. For simplicity, 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.

To reiterate: Both of the above relaxations are expected to reduce the amount of work required to solve this assignment. Also, they force you to write your code using memory management techniques. So make sure you keep them in consideration while designing your solution.

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 writes to its storage, only this thread gets its own copy. The remaining threads would still share the same region.

Given Code:

A minimal makefile and header file are provided along with a stub .c file in your github repo for this assignment when it is set up. You can modify them (or ignore them) as long as your default make rule generates a tls.o file that defines the functions outlined above. Some more code snippets will be discussed in class. The provided code is also available in the Github examples repo under challenges/ch4.

If you think of any tests you want to share with your classmates, please post them to piazza posting test cases to piazza.

Hints:

1) **Test early and test often.** “The program does not crash” does not necessarily imply “the program works.” Determine expected behavior ahead of time, then verify that it actually occurs. If you are running a test case shared by someone else, study that test case and understand what it is supposed to do. Otherwise, you won’t be able to know when unexpected things are happening in the program.
2) Note that we require that your TLS implementation does not depend on the internal state of your threading library. This gives you the benefit that you can also test with `-pthread` instead of `threads.o` (and the autograder will use -pthread). Why is that beneficial? Because now you can go back to using some very helpful tools that worked in homework 1 but not in 2 or 3: valgrind and sanitizers. Also continue to use the static analysis tools (e.g., clang-tidy) that were introduced in the last assignment. Those were able to detect some very confusing bugs in homework 2 and 3.
    a) How do you specify that you want to use the provided pthread implementation? There are two changes in your makefile. First, instead of linking with threads.o, you can add `-pthread` to your `LDLIBS`. Second, you can add the same `-pthread` to your `CFLAGS`.
3) 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:
    a) 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.
    b) `mmap` allows you to create pages that have no read/write permissions, and thus, cannot be accessed arbitrarily.
4) Now 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.

5) 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.
6) 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 CoW TLS library must be written in C and run on the Linux Gradescope environment. Submit your ***source code and makefile*** to Gradescope. When we run `make` in your submission, a tls.o object file must be produced, containing the compiled definitions of the required functions listed in this homework prompt. This file must not define a main function.

You may optionally make your Gradescope submission run with your own thread implementation instead of the system pthread implementation. To do so, just ensure that your default make rule produces both tls.o (as defined in this assignment) and threads.o (as defined in HW2-HW3).

Your final submission must also include a `README` file (README.md is also okay). If you relied on any **external resources** to help complete this assignment, note the resource and the challenge it helped resolve. **If not**, say so. Use this file to provide any **additional notes** you would like to share with instructors about your submission. Aside from that, it just needs a **short description** of the purpose of the project. But you are permitted to include additional details that *you* want in your README.

Oral Exams

When the submission deadline is reached, we will start oral exam sessions over Zoom. In these sessions, we will ask you to explain how some parts of your program work. Details about how to schedule an oral exam time will be posted in the week leading up to the exams.