ECS 150: Project #3 - User-level thread library (part 2)

Joël Porquet

UC Davis, Spring Quarter 2017, version 1

Changelog

- v1: Fix sentence in cloning example
- v0: First publication

General information

Due before **11:59 PM, Friday, May 19th, 2017**.

You will be working with a partner for this project. Remember, you cannot keep the same partner for more than 2 projects over the course of the quarter.

The reference work environment is the CSIF.

Specifications

Note that the specifications for this project are subject to change at anytime for additional clarification. Make sure to always refer to the latest version.

Introduction

The goal of this project is to extend your understanding of threads, by implementing two independent parts:

1. Semaphores, for efficient thread synchronization by using waiting queues.
2. Per-thread protected memory regions. Threads will be able to create their own private storage (a.k.a. TPS or Thread Private Storage), transparently accessible by an API and protected from other threads.

Concerning the thread management, and in order for this project to be as independent as possible from the previous project, we will use the POSIX `pthreads` library.

For both parts, you are welcome to re-use the queue API that you developed for project 2.

Constraints

Your library must be written in C, be compiled with GCC and only use the standard functions provided by the GNU C Library (aka `libc`). *All* the functions provided by the `libc` can be used, but your program cannot be linked to any other external libraries.

Your source code should follow the relevant parts of the Linux kernel coding style and be properly commented.

Skeleton code

The skeleton code that you are expected to complete is available in `/home/jporquet/ecs150/`. This code already defines most of the prototypes for the functions you must implement, as explained in the following sections.

```
$ cd /home/jporquet/ecs150/
$ tree
.
├── libuthread
│   ├── Makefile*
│   ├── sem.c*
│   ├── sem.h
│   ├── thread.h
│   ├── thread.o
│   ├── tps.c*
│   └── tps.h
├── Makefile
├── sem_buffer.c
├── sem_count.c
├── sem_prime.c
└── tps.c
```

The code is organized in two parts. At the root of the directory, there are a couple of test applications which make use of the thread library. You can compile these applications and run them.

In the subdirectory `libuthread`, there are the files composing the thread library that you must complete. The files to complete are marked with a star (you should have **no** reason to touch any of the headers which are not marked with a star, even if you think you do...).

Part 1: semaphore API

The interface of the semaphore API is defined in `libuthread/sem.h` and your implementation should go in `libuthread/sem.c`.

Semaphores are a way to control the access to common resources by multiple threads. Internally, a semaphore has a certain count, that represent the number of threads able to share a common resource at the same time. This count is determined when initializing the semaphore for the first time.

Threads can then ask to grab a resource (known as "down" or "P" operation) or release a resource (known as "up" or "V" operation).

Trying to grab a resource when the count of a semaphore is down to 0 adds the requesting thread to the list of threads that are waiting for this resource. The thread is put in a blocked state and shouldn't be eligible to scheduling.

When a thread releases a semaphore which count was 0, it checks whether some other threads were currently waiting on it. In such case, the first thread of the waiting list can be unblocked, and be eligible to run later.

The semaphore implementation should make use of the functions defined in the thread API, as described in the `thread.h` header. The implementation of this API is provided to you already compiled, as an object file, in `thread.o`. You will probably need to slightly modify your Makefile to not try to recreate this object file when compiling, or remove it when cleaning.

Testing

Three testing programs are available in order to test your semaphore implementation:

- `sem_count`: simple test with two threads and two semaphores
- `sem_buffer`: producer/consumer exchanging data in a buffer
- `sem_primer`: prime sieve implemented with a growing pipeline of threads

Part 2: TPS API

As you know, threads of a same process all share the same memory address space. In general, this is a great behavior because it allows threads to easily share information. However, it can also be an issue when, typically due to programming bugs, one thread accidentally modifies values that another thread was using. 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 could read from and write to. For this project, we call such memory area, *Thread Private Storage*.

The goal of the TPS API is to provide a private and protected memory page (i.e. 4096 bytes) to each thread that requires it.

The interface of the TPS API is defined in `libuthread/tps.h` and your implementation should go in `libuthread/tps.c`.

Note that your TPS implementation should make use of the critical section helpers defined in the thread API, as described in the `thread.h` header, when manipulating shared variables.

Phase 2.1: Unprotected TPS with naive cloning

For this phase, you will need to implement a first version of the API. Below is a few indications to help you started.

In `tps_init()`, you can initialize your API the way you want. Depending on your implementation, it is not necessary that this function contains any code at this point. But if you need some internal objects to be initialized before any TPS can be created, this function is where it should go. For this phase, you can ignore the parameter `segv`.

In `tps_create()`, you must create a TPS for the thread that requires it. As you can notice, this function doesn't return any object. It means that the TPS object must be kept inside the library and shall not be exposed to the client. Each time the client will subsequently interact with its TPS, via `tps_read()` or `tps_write()`, it is up to the library to find the corresponding TPS and operate on it.

> Hint: you can identify client threads by getting their Thread ID with `pthread_self()`.

The page of memory associated to a TPS should be allocated using the C library function `mmap()`. Because `mmap()` allocates memory at the granularity of pages, it will be possible in the next phase to protect these pages by setting no read/write permissions. `mmap()` is very versatile and you will need to study the documentation in order to understand how to allocate a memory page that is private and anonymous, and can be accessed in reading and writing.

In `tps_clone()`, the calling thread is requesting its own TPS whose content must be a copy of the target thread's TPS. For this first phase, you can create a new memory page and simply copy the content from the target thread's TPS with `memcpy()`.

Useful resources for this phase:

- https://www.gnu.org/software/libc/manual/html_mono/libc.html#Memory_002dmapped-I_002fO

Testing

One simple testing program, `tps.c`, is available in order to test your first implementation.

Phase 2.2: Protected TPS

In this phase, you need to modify the previous phase by adding TPS protection.

In `tps_create()`, the memory page should have no read/write permission by default. It's only in `tps_read()` that the page should temporarily become readable, and in `tps_write()` that the page should temporarily become writable.

At this stage, you might now encounter two types of segmentation faults. In addition to the usual programming errors (e.g. dereferencing a NULL pointer), accessing a protected TPS area will also cause a segmentation fault.

A good way to distinguish between these two types of faults is to set up our own segmentation fault handler. Modify `tls_init()` in order to associate a function handler to the signals of type `SIGSEGV` and `SIGBUS`:

```
int tps_init(int segv)
{
    ...
    if (segv) {
        struct sigaction sa;

        sigemptyset(&sa.sa_mask);
        sa.sa_flags = SA_SIGINFO;
        sa.sa_sigaction = segv_handler;
        sigaction(SIGBUS, &sa, NULL);
        sigaction(SIGSEGV, &sa, NULL);
    }
    ...
}
```

Write the signal handler that will distinguish between real segmentation faults and TPS protection faults. Here is a skeleton that needs to be completed:

```
static void segv_handler(int sig, siginfo_t *si, void *context)
{
    /*
     * Get the address corresponding to the beginning of the page where the
     * fault occurred
     */
    void *p_fault = (void*)((uintptr_t)si->si_addr & ~(TPS_SIZE - 1));

    /*
     * Iterate through all the TPS areas and find if p_fault matches one of them
     */
    ...
    if (/* There is a match */)
        /* Printf the following error message */
        fprintf(stderr, "TPS protection error!\n");

    /* In any case, restore the default signal handlers */
    signal(SIGSEGV, SIG_DFL);
    signal(SIGBUS, SIG_DFL);
    /* And transmit the signal again in order to cause the program to crash */
    raise(sig);
}
```

Useful resources for this phase:

- https://linux.die.net/man/2/mprotect
- http://pubs.opengroup.org/onlinepubs/7908799/xsh/sigaction.html

Phase 2.3: Copy-on-Write cloning

With the naive cloning strategy adopted so far, the memory page of the target thread is directly copied when being cloned. But assuming that the new TPS won't be modified right away, it would be possible to delay this copy operation as long as the shared memory page is only read from by either one of the threads sharing it. This approach is called CoW (Copy-on-Write), which saves memory space and avoid unnecessary copy operations until required.

The cloning operation should therefore only create a TLS object that refers to the same page. And it's only when one of the threads sharing the same page actually calls `tls_write()` that a new page is created and the content is copied from the original memory page.

```
- After Thread 2 clones Thread 1's TPS:

+----------+       +-------+       +----------------+       +--------------+
| Thread 1 | +---> | TLS 1 | +---> | Page address X | +---> | Hello world! |
+----------+       +-------+       +----------------+       |              |
                             +-----^                        | Other content|
                             |                              | ...          |
+----------+       +-------+ |                              |              |
| Thread 2 | +---> | TLS 2 | +                              |              |
+----------+       +-------+                                |              |
                                                            |              |
                                                            |              |
                                                            |              |
                                                            |              |
                                                            |              |
                                                            |              |
                                                            +--------------+

- After Thread 2 writes to its TPS and causes a Copy-on-Write:

+----------+       +-------+       +----------------+       +--------------+
| Thread 1 | +---> | TLS 1 | +---> | Page address X | +---> | Hello world! |
+----------+       +-------+       +----------------+       |              |
                                                            | Other content|
                                                            | ...          |
                                                            |              |
                                                            |              |
                                                            |              |
                                                            |              |
                                                            |              |
                                                            |              |
                                                            |              |
                                                            |              |
                                                            |              |
                                                            +--------------+

+----------+       +-------+       +----------------+       +--------------+
| Thread 2 | +---> | TLS 2 | +---> | Page address Y | +---> | Hello davis! |
+----------+       +-------+       +----------------+       |              |
                                                            | Other content|
                                                            | ...          |
                                                            |              |
                                                            |              |
                                                            |              |
                                                            |              |
                                                            |              |
                                                            |              |
                                                            |              |
                                                            |              |
                                                            |              |
                                                            +--------------+
```

In order to implement such cloning strategy, you will need to perform a few modifications to your existing implementation:

1. You need to dissociate the TLS object from the memory page. Before you probably had the address of the TLS's memory page as part of the TLS object, now you need an extra level of indirection. You need a page structure containing the memory page's address, and TLS can only point to such page structures. This way, two or more TLSes can point to the same page structure. Then, in order to keep track of how many TLSes are currently "sharing" the same memory page, the struct page must also contain a reference counter.
2. In `tls_clone()`, you need to allocate a new TLS object for the calling thread, but have the page structure be shared with the target thread and the reference counter updated accordingly.
3. In `tls_write()`, writing to a page that has a reference count superior to 1 should causes a new memory page to be created first. This memory page should be the identical copy of the original one, and should now become the private copy of the calling thread, while the original page's reference counter should be decremented. Also, protection just be properly adjusted for both pages: the original page should become protected, while the new page should become temporarily writable during the ongoing writing operation.

Deliverable

Content

Your submission should contain, besides your source code, the following files:

- `AUTHORS`: first name, last name, student ID, CSIF username and email of each partner, one entry per line formatted in CSV (fields are separated with commas). For example:

```
$ cat AUTHORS
Homer,Simpson,00010001,simpson32,hsimpson@ucdavis.edu
Robert David,Martin,00010002,rdm,rdmartin@ucdavis.edu
```

- `REPORT.md`: a description of your submission. Your report must respect the following rules:
    - It must be formatted in markdown language as described in this Markdown-Cheatsheet.
    - It should contain no more than 300 lines and the maximum width for each line should be 80 characters (check your editor's settings to configure that).
    - It should explain your high-level design choices, details about the relevant parts of your implementation, how you tested your project, the sources that you may have used to complete this project, and any other information that can help understanding your code.
    - Keep in mind that the goal of this report is not to paraphrase the assignment, but to explain how you implemented it.
- `libuthread/Makefile`: a Makefile that compiles your source code without any errors or warnings (on the CSIF computers), and builds a static library named `libuthread.a`.

  The compiler should be run with the options `-Wall -Werror`.

  There should also be a `clean` rule that removes generated files and puts the directory back in its original state.

Your submission should be empty of any clutter files such as executable files, core dumps, etc.

Git

Your submission must be under the shape of a Git bundle. In your git repository, type in the following command (your work must be in the branch `master`):

```
$ git bundle create synctps.bundle master
```

It should create the file `synctps.bundle` that you will submit via `handin`.

You can make sure that your bundle has properly been packaged by extracting it in another directory and verifying the log:

```
$ cd /path/to/tmp/dir
$ git clone /path/to/synctps.bundle -b master synctps
$ cd synctps
$ ls -l
...
$ git log
...
```

Handin

Your Git bundle, as created above, is to be submitted with `handin` from one of the CSIF computers:

```
$ handin cs150 p3 synctps.bundle
Submitting synctps.bundle... ok
$
```

You can verify that the bundle has been properly submitted:

```
$ handin cs150 p3
The following input files have been received:
...
```

```
$
```

Academic integrity

You are expected to write this project from scratch, thus avoiding to use any existing source code available on the Internet. You must specify in your `REPORT.md` 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, or have used the work of past students.

Excessive collaboration, or failure to list external code sources will result in the matter being transferred to Student Judicial Affairs.