ECS 150: Project #2 - User-level thread library

Joël Porquet

UC Davis, Winter Quarter 2018

1 Changelog

- v3: Make the thread unblocking clearer in terms of scheduling
- v2: Fix confusion around paths (nothing should be in the root directory, added test programs should be in directory test/)
- v1: First publication

2 General information

- Due before **11:59 PM, Friday, February 9th, 2017**.
- You will be working **with a partner** for this project.
- The reference work environment is the CSIF.

3 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.*

3.1 Introduction

The goal of this project is to understand the idea of threads, by implementing a basic user-level thread library for Linux. Your library will provide a complete interface for applications to create and run independent threads concurrently.

Similar to existing lightweight user-level thread libraries, your library must be able to:

1. Create new threads
2. Schedule the execution of threads in a round-robin fashion
3. Provide a synchronization mechanism for threads to join other threads
4. Be preemptive, that is to provide an interrupt-based scheduler

3.1.1 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.

3.1.2 Skeleton code

The skeleton code that you are expected to complete is available in `/home/cs150/public/p2`. This code already defines most of the prototypes for the functions you must implement, as explained in the following sections.

```
$ cd /home/cs150/public/p2
$ tree
.
├── libuthread
│   ├── context.c
│   ├── context.h
│   ├── Makefile*
│   ├── preempt.c*
│   ├── preempt.h
│   ├── queue.c*
│   ├── queue.h
│   ├── uthread.c*
│   └── uthread.h
└── test
    ├── Makefile
    ├── uthread_hello.c
    └── uthread_yield.c
```

The code is organized in two parts. In the subdirectory `test`, there are a couple of test applications which make use of the thread library. You can compile these applications and run them (you need Phase 2 to make them running).

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…).

3.2 Phase 1: queue API

In this first phase, you must implement a simple FIFO queue. The interface to this queue is defined in `libuthread/queue.h` and your code should be added into `libuthread/queue.c`.

You will find all the API documentation within the header file.

The constraint for this exercise is that all operations (apart from the iterate and delete operation) must be *O(1)*. This implies that you must choose the underlying data structure for your queue implementation carefully.

3.2.1 1.1 Makefile

Complete the file `libuthread/Makefile` in order to generate a *static library archive* named `libuthread/libuthread.a`.

This library archive must be the default target of your Makefile, because your Makefile is called from the Makefile in the test directory without any argument.

Note that at first, only the file `libuthread/queue.c` should be included in your library. You will add the other C files as you start implementing them in order to expand the API provided by your library.

Useful resources for this phase:

- http://tldp.org/HOWTO/Program-Library-HOWTO/static-libraries.html

3.2.2 1.2 Testing

Add a new test program in the test directory, called `test_queue.c`, which tests your queue implementation. It is important that your test program is as comprehensive as possible in order to ensure that your queue implementation is resistant. It will ensure that you don't encounter bugs when using your queue later on.

Many tests can be to create queues, enqueue some items, make sure that these items are dequeued in the same order, delete some items, test the length of the queue, etc.

For example, a first basic step would be to make sure that your implementation doesn't crash when receiving NULL pointers as arguments:

```
assert(queue_destroy(NULL) == -1);
assert(queue_enqueue(NULL, NULL) == -1);
```

3.2.3 1.3 Hints

Most of the functions of this API should look very familiar if you have ever coded a FIFO queue (e.g. create, destroy, enqueue, dequeue, etc.). However, one function of the API stands out from typical interfaces: `queue_iterate()`. This function provides a generic way to call a custom function (i.e. a function provided by the caller) on each item currently enqueued in the queue.

For example, the following snippet of code shows you how a certain operation can be applied to every item of a queue, or how you can find a certain item in the queue and return it:

```c
/* Callback function that increments items by a certain value */
static int inc_item(queue_t q, void *data, void *arg)
{
    int *a = (int*)data;
    int inc = (intptr_t)arg;

    *a += inc;

    return 0;
}

/* Callback function that finds a certain item according to its value */
static int find_item(queue_t q, void *data, void *arg)
{
    int *a = (int*)data;
    int match = (intptr_t)arg;

    if (*a == match)
        return 1;

    return 0;
}

void test_iterator(void)
{
    queue_t q;
    int data[] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
    int i;
    int *ptr;

    /* Initialize the queue and enqueue items */
    q = queue_create();
    for (i = 0; i < 10; i++)
        queue_enqueue(q, &data[i]);

    /* Add value '1' to every item of the queue */
    queue_iterate(q, inc_item, (void*)1, NULL);
    assert(data[0] == 2);

    /* Find and get the item which is equal to value '5' */
    queue_iterate(q, find_item, (void*)5, &ptr);
    assert(*ptr == 5);
}
```

Hopefully, you will find that this function can be used in various ways when implementing the uthread API. One interesting usage can be, for example, to debug your queue(s) of threads! But other usages are possible too…

3.3 Phase 2: uthread API

In this second phase, you must implement most of the thread management (some is provided to you for free). The interface to this thread API is defined in `libuthread/uthread.h` and your code should be added into `libuthread/uthread.c`.

You will find all the API documentation within the header file.

3.3.1 Thread definition

Threads are independent execution flows that run concurrently in the address space of a single process (and thus, share the same global variables, heap memory, open files, process identifier, etc.). Each thread has its own execution context, which mainly consists of:

1. an identifier, know as TID (Thread IDentifier)
2. a state (running, ready, blocked, etc.)
3. a backup of the CPU registers (for saving the thread upon de-scheduling and restoring it later)
4. a stack

The goal of a thread library is to provide applications that want to use multithreading an interface (i.e. a set of library functions) that the application can use to create and start new threads, terminate threads, or manipulate threads in different ways.

For example, the most well-known and wide-spread standard that defines the interface for threads on Unix-style operating systems is called *POSIX thread* (or `pthread`). The pthread API defines a set of functions, a subset of which we want to implement for this project. Of course, there are various ways in which the pthread API can be realized, and existing libraries have implemented pthread both in the OS kernel and in user mode. For this project, we aim to implement a few pthread functions at user level on Linux.

3.3.2 Public API

The API of the uthread library defines the set of functions that applications and the threads they create can call in order to interact with the library.

The first function an application has to call in order to kick off the uthread library and create the first user thread is `uthread_create()`.

When called for the first time, this function must first initialize the uthread library by registering the so-far single execution flow of the application as the *main* thread that the library can later schedule for execution like any other thread.

Hint: you should put this initialization code in a separate function for clarity.

`uthread_create()` then creates the first user thread as specified by the arguments, and registers it to the library so that it can be scheduled for execution later. In this function, you will need to use the `context` API as discussed below.

This first user thread can in turn call `uthread_create()` in order to create more user threads, and so on.

For this step, we expect the scheduler to be non-preemptive. Threads must call the function `uthread_yield()` in order to ask the library's scheduler to pick and run the next available thread. In non-preemptive mode, a non-compliant thread that never yields can keep the processing resource for itself.

For this first step, you also don't have to write `uthread_join()` the way it is defined by the documentation. At this point, we assume that `uthread_join()` is only called by the main function, mostly in order to wait for the threading system to terminate executing threads. You can therefore implement `uthread_join()` as follows:

- Execute an infinite loop in which
  - If there are no more threads which are ready to run in the system, break the loop and return
  - Otherwise simply yield to next available thread

3.3.3 Private data structure

In order to deal with the creation and scheduling of threads, you will need a data structure that can store information about a single thread.

This data structure will likely need to hold, at least, information mentioned above such as the TID, the context of the thread (its set of registers), information about its stack (e.g., a pointer to the thread's stack area), and information about the state of the thread (whether it is running, ready to run, or has exited).

It will also need to contain more information when you implement the real version of `uthread_join()`.

This data structure is often called a thread control block (*TCB*).

3.3.4 Internal `context` API

Some code located in `libuthread/context.c`, and which interface is defined in `libuthread/context.h`, is accessible for you to use. The four functions provided by this library allow you to:

- Allocate a stack when creating a new thread (and conversely, destroy a stack when a thread is finally deleted)
- Initialize the stack and the execution context of the new thread so that it will run the specified function with the specified argument
- Switch between two execution contexts

Useful resources if you would like to further understand how the `context` API works internally:

- https://www.gnu.org/software/libc/manual/html_mono/libc.html#System-V-contexts

3.3.5 Testing

Two applications can help test this phase: - `uthread_hello`: creates a single thread that displays "Hello world!" - `uthread_yield`: creates three threads in cascade and test the yield feature of the scheduler

3.4 Phase 3: `uthread_join()`

When a thread exits, its resources (such as its TCB) are not automatically freed. It must first be "joined" by another thread, which can then (or not) collect the return value of the dead thread. The concept is very similar to `waitpid()` that you have seen for project 1.

In this phase, you need to remove the infinite loop that you used in the previous phase for `uthread_join()` and implement the proper behavior for this function.

When a thread T1 joins another thread T2 (usually, a parent joins its child), there are two possible scenarios:

1. If T2 is still an active thread, T1 must be blocked (i.e. it cannot be scheduled to run anymore) until T2 dies. When T2 dies, T1 is unblocked and collects T2.
2. If T2 is already dead, T1 can collect T2 right away.

Once T2 is collected, its resources can be finally entirely freed.

When T1 is unblocked, it should be scheduled after all the currently runnable threads.

3.4.1 Testing

For testing this phase, you should probably modify the two previous applications and add some proper joining from parents to children and see if the resulting synchronization corresponds to what is expected.

3.5 Phase 4: preemption

Up to this point, uncooperative threads could keep the processing resource for themselves if they never call `uthread_yield()`.

In order to avoid such dangerous behaviour, you will add preemption to your library. The interface of the preemption API is defined in `libuthread/preempt.h` and your code should be added to `libuthread/preempt.c`.

3.5.1 `preempt_start()`

The function that sets up preemption, `preempt_start()`, is a two-step procedure:

1. Install a signal handler that receives alarm signals (of type `SIGVTALRM`)
2. Configure a timer which will fire an alarm (through a `SIGVTALRM` signal) a hundred times per second (i.e. 100 Hz)

Your signal handler, which acts as the timer interrupt handler, will force the currently running thread to yield, so that another thread can be scheduled instead.

Useful resources for this phase:

- https://www.gnu.org/software/libc/manual/html_mono/libc.html#Signal-Actions
- https://www.gnu.org/software/libc/manual/html_mono/libc.html#Setting-an-Alarm

It is strongly encouraged to use `sigaction()` over `signal()`.

3.5.2 `preempt_{enable,disable}()`

The two other functions that you must implement are meant to enable or disable preemption. For that, you will need to be able to block or unblock signals of type `SIGVTALRM`.

Useful resources for this phase:

- https://www.gnu.org/software/libc/manual/html_mono/libc.html#Blocking-Signals

3.5.3 About disabling preemption…

Preemption is a great way to enable reliable and fair scheduling of threads, but it comes with some pitfalls.

For example, if the library is accessing sensitive data structures in order to add a new thread to the system and gets preempted in the middle, scheduling another thread of execution that might also manipulate the same data structures can cause the internal state of the library to become inconsistent.

Therefore, when manipulating shared data structures, preemption should be temporarily disabled so that such manipulations are guaranteed to be performed *atomically*.

However, avoid disabling preemption each time a thread calls the library. Try to disable preemption only when necessary. For example, the creation of a new thread can be separated between sensitive steps that need to be done atomically and non-sensitive steps that can safely be interrupted and resumed later without affecting the consistency of the shared data structures.

A good way to figure out whether preemption should be temporarily disabled while performing a sequence of operations is to imagine what would happen if this sequence was interrupted in the middle and another thread scheduled.

3.5.4 Testing

Add a new test program in the test directory, called `test_preempt.c`, which tests the preemption. Explain in your report why this program demonstrates that your preemptive scheduler works.

Hint: the test program doesn't have to be overly complicated…

4 Deliverable

4.1 Content

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

- `AUTHORS`: student ID of each partner, one entry per line. For example:

  ```
  $ cat AUTHORS
  00010001
  00010002
  $
  ```

- `IGRADING`: **only if your group has been selected for interactive grading for this project**, the interactive grading time slot you registered for.

  - If your group has been selected for interactive grading for this project, this file must contain exactly one line describing your time slot with the format: `%m/%d/%y - %I:%M %p` (see `man date` for details). For example, an appointment on Monday January 15th at 2:10pm would be transcribed as `01/15/18 - 02:10 PM`.

  ```
  $ cat IGRADING
  01/15/18 - 02:10 PM
  $
  ```

- `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.

All the submitted files should be in Unix format.

- If you created the files on Windows, you can use the `dos2unix` tool in order to change the DOS newline sequence into the Unix one:

  ```
  $ dos2unix AUTHORS
  dos2unix: converting file AUTHORS to Unix format...
  $
  ```

- You also need to make sure that your file ends with a newline character. Here is an example when it does not, how to fix it, and what the final result should look like.

  ```
  $ cat AUTHORS
  00010001
  00010002$ echo "" >> AUTHORS
  $ cat AUTHORS
  00010001
  00010002
  $
  ```

4.2 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 p2.bundle master
```

It should create the file `p2.bundle` that you will submit via `handin`.

Before submitting, do 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/p2.bundle -b master uthread
$ cd uthread
$ ls -l
...
$ git log
...
```

4.3 Handin

Your Git bundle, as created above, is to be submitted with `handin` from one of the CSIF computers by **only one person of your group**:

```
$ handin cs150 p2 p2.bundle
Submitting p2.bundle... ok
$
```

You can verify that the bundle has been properly submitted:

```
$ handin cs150 p2
The following input files have been received:
...
$
```

5 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.