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

Prof. Joël Porquet-Lupine

UC Davis, Fall Quarter 2020

Changelog

> **NOTE:**
>
> *The specifications for this project are subject to change at anytime for additional clarification. Make sure to always refer to the **latest** version.*

- v2: Fix typos
- v1: First publication

General information

- Due before **11:59 PM, Thursday, November 12th, 2020**.
- You will be working with a partner for this project.
- The reference work environment is the CSIF.

Objectives of the project

The objectives of this programming project are:

- Implementing one of the most used containers in system programming (a queue/list) as specified by a given API.
- Learning how to test your code, by writing your own testers and maximizing the test coverage.
- Understanding how multiple threads can run within the same process: from the creation of new threads to their termination, and including how to perform context switching during their concurrent execution.
- Implementing one of the most popular synchronization primitives (semaphores) as specified by a given API.
- Writing high-quality C code by following established industry standards.

Program description

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. Be preemptive, that is to provide an interrupt-based scheduler
4. Provide a thread synchronization API, namely semaphores

Constraints

Your code must be written in C, be compiled with GCC and only use the standard functions provided by the GNU C Library (https://www.gnu.org/software/libc/manual/) (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 adopt a sane and consistent coding style and be properly commented when necessary. One good option is to follow the relevant parts of the Linux kernel coding style (https://www.kernel.org/doc/html/latest/process/coding-style.html).

Assessment

Your grade for this assignment will be broken down in two scores:

Auto-grading: ~60% of grade

Running an auto-grading script that tests your code and checks the output against various inputs

Manual review: ~40% of grade

The manual review is itself broken down into different rubrics:

- Submission : ~5%
- Report file: ~40%
- Makefile: ~10%
- Queue implementation: ~10%
- Queue testing: ~10%
- Uthread library: ~15%
- Code style: ~10%

Suggested work phases

Phase 0: Skeleton code

The skeleton code that you are expected to complete is available in `/home/cs150jp/public/p2/`. This code already defines most of the prototypes for the functions you must implement, as explained in the following sections.

```
$ cd /home/cs150jp/public/p2
$ tree
.
├── apps
│   ├── Makefile
│   ├── queue_tester.c
│   ├── sem_buffer.c
│   ├── sem_count.c
│   ├── sem_prime.c
│   ├── sem_simple.c
│   ├── uthread_hello.c
│   └── uthread_yield.c
└── libuthread
    ├── context.c
    ├── Makefile*
    ├── preempt.c*
    ├── private.h
    ├── queue.c*
    ├── queue.h
    ├── sem.c*
    ├── sem.h
    ├── uthread.c*
    └── uthread.h
```

The code is organized in two parts. In subdirectory `apps`, you will find several test applications. The tester `queue_tester.c` focuses solely on the queue implementation, and can be run after Phase 1 is completed. The testers prefixed with `uthread_` make use of the thread library and require Phase 2 to be completed, while the testers prefixed with `sem_` require Phase 3 to be completed.

Subdirectory `libuthread` contains the files composing the thread library that you must complete. The files to complete are marked with a star.

> **IMPORTANT:**
>
> You should have **no** reason to touch any of the headers which are not marked with a star (even if you think you do…).

Copy the skeleton code to your account.

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.

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

Testing

Add a new test program in the apps directory, called `queue_tester.c`, which tests your queue implementation. It is important that your tester should be 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.

A good approach for testing your queue implementation is *unit testing*. The basic idea behind unit testing is to invent all the possible usage scenarios that will trigger the different parts of the implementation, and all the edge cases, in order to guarantee that the implementation always matches the specifications.

Here are two examples to get you started. The first unit test simply checks that creating a queue succeeds, while the second checks a simple enqueue/dequeue scenario:

```
void test_create(void)
{
    queue_t q;

    q = queue_create();
    assert(q != NULL);
}

void test_queue_simple(void)
{
    queue_t q;
    int data = 3, *ptr;

    q = queue_create();
    queue_enqueue(q, &data);
    queue_dequeue(q, (void**)&ptr);
    assert(ptr == &data);
}
```

You can find a comprehensive example of a queue tester in `/home/cs150jp/public/p2/apps/queue_tester_example.c`, which you can complete with more unit tests.

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 certain operations can be applied to every item of a queue:

```
queue_t q;

/* Callback function that increments items */
static void inc_item(void *data)
{
    int *a = (int*)data;
    if (*a == 5)
        queue_delete(q, data);
    else
        *a += 1;
}

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

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

    /* Increment every item of the queue, delete item '5' */
    queue_iterate(q, inc_item);
    assert(data[0] == 2);
    assert(queue_length(q) == 9);
}
```

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

Phase 2: uthread API

In this second phase, you must implement most of the thread management (some is provided to you for free). The *public* 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.

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. a state (running, ready, blocked, etc.)
2. a backup of the CPU registers (for saving the thread upon descheduling and restoring it later)
3. 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 may 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.

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_start()`. This function performs three actions:

1. It registers the so-far single execution flow of the application as the *idle* thread that the library can schedule.
2. It creates a new thread, the *initial thread*, as specified by the arguments of the function.
3. The function finally execute an infinite loop which,
    1. When there are no more threads which are ready to run in the system, stops the idle loop and exits the program.
    2. Or simply yields to next available thread.
    3. (It could also deal with threads that reached completion and destroys their associated TCB.)

Once the *initial thread* is created, it can interact with the other functions of the library, and create new threads, exit, yield, etc.

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.

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 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).

This data structure is often called a thread control block (*TCB*) and will be described by `struct uthread_tcb`.

Internal `context` API

Some code located in `libuthread/context.c`, and which interface is defined in `libuthread/private.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 (https://www.gnu.org/software/libc/manual/html_mono/libc.html#System-V-contexts)

Testing

Two programs 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

Phase 3: 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. To keep track of the number of available resource, a semaphore maintains an internal count, which can already be initialized to a certain positive value when the semaphore is created.

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.

As you can now understand, your semaphore implementation will make use of the blocking functions defined in the private thread API.

Testing

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

- `sem_simple`: simple example of scheduling forced by semaphores
- `sem_count`: alternate counting with two threads and two semaphores
- `sem_buffer`: producer/consumer exchanging data in a buffer
- `sem_prime`: prime sieve implemented with a growing pipeline of threads

It is recommended to look at the code of these testers and understand how they work in order to see how they will stress your semaphore implementation.

Corner case

There is a very specific corner case that you need to consider in order to implement your semaphore correctly. Here is the scenario:

- Thread A calls `down()` on a semaphore with a count of 0, and gets blocked.
- Thread B calls `up()` on the same semaphore, and gets thread A to be awaken
- Before thread A can run again, thread C calls `down()` on the semaphore and snatch the newly available resource.

There are two difficulties with this scenario:

1. The semaphore should make sure that thread A will handle the situation correctly when finally resuming its execution. Theoretically, it should go back to sleep if the resource is not longer available by the time it gets to run. If the thread proceeds anyway, then the semaphore implementation is incorrect.
2. If this keeps happening, thread A will eventually be starving (i.e., it never gets access to the resource that it needs to proceed). Ideally, the semaphore implementation should prevent starvation from happening.

Think about your implementation to *at least* make it correct for that corner case, and *ideally*, find a way to avoid any starvation.

Phase 4: preemption

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

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

> **NOTE:**
>
> This preemption API is not meant to be exposed to user threads, it should stay completely transparent for them. Whenever the user code of a thread is running, preemption should be enabled.

`preempt_{start,stop}()`

The function `preempt_start()` should be called when the uthread library is initializing and sets up preemption. The setup 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.

The function `preempt_stop()` should be called before `uthread_start()` returns, once the multithreading phase of the application ends. It should restore the previous signal action, and restore the previous timer configuration.

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

> **IMPORTANT:**
>
> It is mandatory to use `sigaction()` over `signal()`.

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

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*, as critical sections.

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.

Testing

Add a new test program in the apps 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…

Submission

Content

Your submission should contain, besides your source code (library and tester(s)), the following files:

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

```
$ cat AUTHORS.csv
00010001,jdupont@ucdavis.edu
00010002,mdurand@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 (https://github.com/adam-p/markdown-here/wiki/Markdown-Cheatsheet).
  - It should contain no more than 200 lines and the maximum width for each line should be 80 characters (check your editor's settings to configure it automatically –please spare yourself and do not do the formatting manually).

- 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 -Wextra -Werror`.

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

  The Makefile should use all the advanced mechanisms presented in class (variables, pattern rules, automatic dependency tracking, etc.)

> **IMPORTANT:**
>
> Your submission should be empty of any clutter files (such as executable files, core dumps, backup files, `.DS_Store` files, and so on).

Gradescope

Gradescope will be opened for submission on Tuesday, November 10th at 0:00. At that time, you will be able to submit your project as a Git repository.

> **IMPORTANT:**
>
> There should be only one final submission per group, submitted by one of the two partners.
>
> The other partner should be added to the submission as “group member”.

Academic integrity

Novelty

You are expected to write this project **from scratch**.

Therefore, you cannot use any existing source code available on the Internet, or even reuse your own code if you took this class before.

Authorship

You are also expected to write this project **yourself**.

Asking anyone someone else to write your code (e.g., a friend, or a "tutor" on a website such as Chegg.com) is not acceptable and will result in severe sanctions.

Sources

You must specify in your report any sources that you have viewed to help you complete this assignment. All of the submissions will be compared with MOSS to determine if students have excessively collaborated, or have used the work of past students.

Violation

Any failure to respect the class rules, both as explained above and in the syllabus, or the UC Davis Code of Conduct (http://sja.ucdavis.edu/cac.html) will automatically result in the matter being transferred to Student Judicial Affairs.

Copyright © 2017-2020 Joël Porquet-Lupine

UCDAVIS
COMPUTER SCIENCE