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

UC Davis, Winter Quarter 2017

General information

Due before **11:59 PM, Friday, Feb 10th, 2017**.

You will be working with a partner for this project.

The reference work environment is the CSIF.

Specifications

Note that the specifications for this project are subject to change at anytime for additional clarification.

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 execution threads
2. Schedule the execution of threads in a round-robin fashion
3. Provide a thread synchronization API, namely semaphores
4. Be preemptive, that is to provide an interrupt-based scheduler

A working example of the thread library can be found on the CSIF, at /home/jporquet/ecs150/libuthread.a.

Skeleton code

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

```
$ tree
.
├── libuthread
│   ├── context.c*
│   ├── context.h
│   ├── Makefile*
```

```
│   ├── preempt.c*
│   ├── preempt.h
│   ├── queue.c*
│   ├── queue.h
│   ├── semaphore.c*
│   ├── semaphore.h
│   ├── uthread.c*
│   └── uthread.h
├── Makefile
├── test1.c
├── test2.c
├── test3.c
├── test4.c
└── test5.c
```

The code is organized in two parts. At the root of the directory, there are some 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 normally should not have to touch any of the files which are not marked with a star).

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.

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.

1.1 Makefile

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

This library archive must be the default target of your Makefile, because your Makefile is called from the root Makefile 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.

1.2 Testing

Add a new test program in the root directory (e.g. test-queue.c) which tests your queue implementation. It is important that your test program is comprehensive in order to ensure that your queue implementation is as resistant as possible. It will ensure that you don't encounter bugs when using your queue later on.

A few 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, make sure to try your implementation doesn't crash when receiving NULL pointers as arguments:

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

Phase 2: thread 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.

Note that the thread API is actually composed of two sets: a public API and a private API, as explained below.

Thread definition

Threads are independent execution flows that run concurrently in the address space of a single process (and thus, share the same 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. the set of 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 threads 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.

Public API

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

From the point of view of applications, threads are designated by a number of type uthread_t. Think of it as the equivalent of pid_t for Unix processes.

The first function an application has to call in order to initialize your library is uthread_start(). This function must perform 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, it stops the idle loop and exits the program.
    2. Or it simply yields to next available thread

Once the *initial thread* created, it can interact with the library to create new threads, exit, yield execution, etc.

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

Private API

The private API of the thread library defines the set of functions that can only be accessed from the code of the library itself, and not by applications using the library.

In order to deal with the creation and scheduling of threads, you first 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 state 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 status 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.

At this point, the functions defined in the private API could theoretically be only defined in libuthread/uthread.c and not exported to the rest of the library. But with the implementation of the semaphore API, semaphores will need to have access to these functions in order to manipulate the thread when necessary.

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 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

Testing

Two applications can help test this phase: - test1: creates a single thread that displays "hello world" - test2: creates three threads in cascade and test the yield feature of the scheduler

Phase 3: semaphore API

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 run.

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

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

Testing

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

- test3: simple test with two threads and two semaphores
- test4: producer/consumer exchanging data in a buffer
- test5: prime sieve implemented with a growing pipeline of threads (this test really stresses both the thread management and the semaphore part of the library)

Phase 4: preemption

Up to this point, uncooperative threads could keep the processing resource for themselves if they never called uthread_yield() or never blocked 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/preempt.h and your code should be added to libuthread/preempt.c.

The function that sets up preemption, preempt_start(), is already provided for you to call when you start the thread library. This function configures a timer which will fire an alarm (through a SIGVTALRM signal) a hundred times per second.

Internally, you must provide a timer handler which will force the currently running thread to yield, so that another thread can be scheduled instead.

The other functions that you must implement deal with: - enabling/disabling preemption - saving/restoring preemption (saving means that the current preemption state must be saved and preemption must be disabled; restoring that the previously saved preemption state must be restored) - checking if preemption is currently disabled

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 share state of the library to become inconsistent.

Therefore, when manipulating shared data structures, preemption should probably 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.

As a hint, in the reference implementation, the preempt API is used in the following files:

```
$ grep -l preempt_* libuthread/*.c | uniq
libuthread/preempt.c
libuthread/semaphore.c
libuthread/uthread.c
libuthread/context.c
```

Deliverable

Constraints

Your library must be written in C, be compiled with GCC and only use the standard functions provided by the GNU C Library. It 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.

Content

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

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

  ```
  $ cat AUTHORS
  Jean Dupont,00010001,jdupont@ucdavis.edu
  Marc Durand,00010002,mdurand@ucdavis.edu
  ```

- REPORT.md: a markdown-formatted file containing a description of your submission.

  This file should explain your design choices, how you tested your project, the sources that you may have used to complete this project, etc. and any other relevant information.

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 uthread.bundle master
```

It should create the file uthread.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/uthread.bundle -b master uthread
$ cd uthread
$ git log
...
```

Handin

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

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

You can verify that the bundle has been properly submitted:

```
$ handin cs150 p2
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 README.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. Excessive collaboration, or failure to list external code sources will result in the matter being transferred to Student Judicial Affairs.