

Assignment - In progress

Add attachment(s), then choose the appropriate button at the bottom.

Title Project 1: GTThreads

Due Jan 8, 2015 11:55 pm

Status Not Started

Grade Scale Points (max 10.0)

Instructions

Goal

You are to implement GTThreads -- a preemptive user-level thread package with an API similar to Pthreads (details below). You are to demonstrate your library by further implementing a solution to the classic Dining Philosophers problem.

First you must implement the core of the thread library as well as a thread scheduler. The scheduler must be a preemptive round robin scheduler. Each thread is assigned a time slice (its quantum) for which it is allowed to run; a thread is preempted if it used up its quantum. Preemption can be achieved by using an alarm signal as a timer. You may use the system's virtual time instead of wall-clock time, man `setitimer(2)` for details. The preemption period in microseconds should be specified in a function, (i.e. `gtthread_init(period)`). Any program using your GTThreads package must call the function first.

Collaboration

Students are allowed to discuss concepts and techniques for solving development problems. However, sharing implementations and code is not allowed. Each student is expected to turn in code that he/she wrote him/herself.

GTThreads API

The GTThreads API should contain the following functions. The library interface must match the one specified here exactly. If the signature is different, and as a result, your code is not compiled with test cases, you will get 0 credit.

```
void gtthread_init(long period);
int gtthread_create(gtthread_t *thread,
                   void *(*start_routine)(void *),
                   void *arg);
int gtthread_join(gtthread_t thread, void **status);
void gtthread_exit(void *retval);
void gtthread_yield(void);
int gtthread_equal(gtthread_t t1, gtthread_t t2);
int gtthread_cancel(gtthread_t thread);
gtthread_t gtthread_self(void);
```

Next, you have to implement a mutex synchronization primitive to enable safe concurrent manipulation of shared data. The API for the mutex primitive should be as follows:

```
int  gtthread_mutex_init(gtthread_mutex_t *mutex);
int  gtthread_mutex_lock(gtthread_mutex_t *mutex);
int  gtthread_mutex_unlock(gtthread_mutex_t *mutex);
```

An example `gtthread.h` header file with prototypes for all the API functions listed here is attached. Feel free to use it, modify it (e.g., to add data structures), or create your own. However, be sure you stick to these prototypes exactly or else our testing/grading code will not be able to bind to it properly.

GTThreads Semantics

Each function in GTThreads is analogous to a PThreads function; simply replace the `gtthread_` at the beginning with `pthread_`. There are two exceptions to this naming pattern: `gtthread_init(period)` is a unique function to GTThreads, and `gtthread_yield()` is analogous to the PThread function `sched_yield()`. Be sure to 'man' the functions you are emulating so you understand how they're supposed to behave. Consulting other PThreads references will probably be helpful as well. Your GTThreads should behave identically to PThreads, with the following exceptions:

1. The `gtthread_init(period)` function must be called from the main thread before *any* other GTThreads functions are called. It allows the caller to specify the scheduling period (quantum in microseconds), and may also perform any other initialization you deem necessary. Calling other GTThreads functions before the library is initialized is semantically invalid, and you may assume it will not be done; you do not need to check for this as an error condition.
2. The `gtthread_create(thread, start_routine, arg)` function does not have an `attr` parameter. All your threads should behave as if they have default PThread attributes (i.e. as if you specified `NULL` for `attr`).
3. In addition to not needing to implement attributes, you may have noticed that there is no `gtthread_detach(thread)` function. You are not required to implement detached threads; all threads should be joinable.
4. The `gtthread_cancel()` function does not have to implement deferred cancelation; all cancelled threads should be killed immediately.
5. Instead of a priority scheduler, you should implement a round-robin scheduler. It should still be preemptive, however.
6. The `gtthread_mutex_init(mutex)` function does not have an `attr` parameter. All your mutexes should behave as if they have default mutex attributes (i.e. as if you specified `NULL` for `attr`).
7. You are *not* required to implement `gtthread_mutex_destroy(mutex)`, `gtthread_mutex_trylock(mutex)`, or a static initializer (e.g. `GTTHREAD_MUTEX_INITIALIZER`).
8. Many of these pthread functions either return an error code or return non-zero and set `errno` to an error code when they have an error. These error codes tell the caller what, specifically, went wrong. However, you do not need to return proper error codes in your `gtthread` library; simply returning non-zero when an error occurs is good enough, and you do not ever need to set `errno`. (It's also okay for a function to always return zero if it is written in a way that it cannot ever fail. Error returns are only needed when errors occur.)

Suggestions

- The initial thread of the program (i.e. the one running `main()`) is a thread exactly like any other. It should have a `gtthread_t`, you should be able to get it by calling `gtthread_self()` from the initial thread, and you should be able to specify it as an argument to other GTThreads functions. The only difference in the initial thread is how it behaves when you execute a return instruction. You can find details on this difference in the man page for `pthread_create`.
- One way to achieve preemption is using an alarm signal. In addition, you may use the system's virtual time instead of wall-clock time. Virtual time does not count CPU cycles spent on other

processes, and decrements only when the calling process is running. The `setitimer()` function can be used to generate SIGVTALRM signals at a specified interval. You will need to establish a signal handler for the signal (man `signal(2)` and `signal(7)` for more information on signals). Test your package with different scheduling periods to determine an appropriate value.

- One way to perform context switching is using the `makecontext()` / `getcontext()` / `setcontext()` / `swapcontext()` functions. Note that these functions do not act like normal C functions, which is in fact the point. Consult the man pages on these functions for details.
 - The `*context()` functions are fairly platform-independent, but may be unsafe in signals. If you choose this approach, please report in your README any potential signal-safe issues in your scheduler if there are any.
- Work incrementally! First try to implement thread switching before moving on to the nice library interface.

The Dining Philosophers

Finally, you have to implement a solution to the classic Dining Philosophers problem using your implementation of GTThreads. The problem statement is as follows:

Five philosophers are sitting at a round table doing their two favorite things: eating and thinking. Each philosopher has their own bowl of rice, but unfortunately there are only five chopsticks for the entire table. The chopsticks are spaced equally around the table, one between each pair of neighboring rice bowls. Whenever a philosopher wants to eat, they must acquire a chopstick from the both the left and the right; if one of the two chopsticks is already in use, then the philosopher must wait hungrily until it is released. Whenever a philosopher wants to stop eating and think for a while, they must release any chopsticks they hold and put them back on the table.

In your implementation, each philosopher should have a thread which alternates between eating and thinking, spending a random amount of time in each mode. Chopsticks are a shared resource and each must be protected by a mutex. Each thread should print out status information, such as "Philosopher #1 is hungry", "Philosopher #1 tries to acquire left chopstick", "Philosopher #1 releases right chopstick", etc.

VERY IMPORTANT: Your algorithm should not have deadlocks regardless of the scheduling period used by your thread system and the order that threads run.

Makefile

Makefile should generate `gtthread.a` (assuming `gtthread.h` is provided statically; alternatively you may have your Makefile generate `gtthread.h`.) Both files must be in the directory where Makefile is. If you are not familiar with Makefile, google it. A sample Makefile is attached to this assignment; feel free to use it, modify it, or create your own.

Example

We will go through 25 test cases to evaluate your library. One example (`test1.c`) is attached to this assignment.

This code should be compiled with your library with something like following command:
`gcc -Wall -pedantic -I{...} -o test1 test1.c gtthread.a`

Test Cases Hints

Checks thread creation, arguments, return values for different functions, scheduling and preemption.




Platform

You can develop your code in any architecture. However, make sure that your code runs properly on **Ubuntu 12.04 on 64-bit x86 hardware (no virtualization)**, which is what the TA will run your code on. Note that some linux distributions and different version of kernel may show different behaviors for the functions mentioned above. If you want to use different platform, please discuss with TA.

Deliverables

- Turn in a tarball (.tar.gz file; man tar(1) and see the z, c, and f flags; and example is attached) named `gtthread.tar.gz` that includes the following:
 - Your thread library code.
 - all source file
 - `gtthread.h`, unless it is automatically generated by the Makefile
 - submit this even if you use the attached `gtthraed.h` unmodified
 - Makefile
 - the command `make` should build the `gtthread.a` library file and your Dining Philosophers binary
 - Your implementation code of the Dining Philosophers.
 - all source code
 - your Makefile should build the Dining Philosophers binary
 - A README file including:
 - What Linux platform do you use?
 - How the preemptive scheduler is implemented.
 - How to compile your library and run your program.
 - How you prevent deadlocks in your Dining Philosophers solution.
 - Any thoughts you have on the project, including things that work especially well or which don't work.
 - Your Makefile, `gtthread.h`, `gtthread.a`, Dining Philosophers binary, and README file should all be in (or be built in) the top level directory, not a subdirectory within your tarball.

Additional resources for assignment

-  [gtthread.h](#) (2 KB; Jan 17, 2014 2:45 pm)
-  [Makefile](#) (1 KB; Jan 17, 2014 2:45 pm)
-  [test1.c](#) (1 KB; Jan 17, 2014 2:46 pm)

▶ Model Answer

Submission

This assignment allows submissions by attaching documents only.

Attachments

No attachments yet

Select a file from computer

Don't forget to save or submit