Name: 林星宇
Student ID: 1300012782
January 24, 2016

Report on User-level Thread Library

OS final project

1. Introduction

Threads provide the illusion that different parts of the program are executing concurrently. In this model, threads share the code, heap, and the runtime system. Each thread, however, has a separate stack and a separate set of CPU registers. This programming model also provides synchronization primitives so that different threads can coordinate access to shared resources.

Compared to kernel threads, user threads are more fast and efficient. They does not require modification on the OS. Further more, the representation and the management of the user level threads are more simple. However, the trade-off is that since the kernel thread scheduler has no knowledge of the user level threads, the scheduling policy may not be optimized.

In this project I implemented a simple but sufficient user-level library. The whole package can also be found on my **Github_Repository**.

2. Application programming interfere

- **int uthread_create(uint* tid, Fun func, void* arg);**

  Create a thread and specify the entry function and the arguments in func and arguments. The thread id is returned throught the pointer tid

- **void uthread_yield();**

  Function for a currently running thread to give up the CPU actively

- **void uthread_join(uint tid);**

  Suspend the currently running thread until the thread specified by tid is finished. Can be used for a parent threads to reap a child thread

- **void uthread_self();**

  Return the thread ID of the currently running thread. Crucial for a parallelized program

- **void uthread_runall();**

  When threads are created through the uthread_create, the threads are not immediately executed, but all wait in a ready queue until the runall functions are called

- **void uthread_exit();**

  Exit and reap the currenlty running threads

3. Programming model

The first time a thread is created, a special thread called the "Scheduler thread" is created, which will schedule all the threads using a specified policy. Here what I implemented is a preemptive Round-Robin scheduler, where the length of the time slice can be specified through a macro. All threads wait in the ready queue until the procedure thread_runall is called and start all the threads. The main function will suspend until this function returns and all threads will be reaped.

All threads are maintained in a thread pool and have a status of **{running, runnable, free}**, representing that a thread is running, ready or not allocated respectively.

4. Machanisim used

During the implementation, there are two problems I need to tackle, how to switch context, and how to preempt a currently running thread.

4.1 Ucontext

ucontext is a set of Unix standard system functions provided for user to switch context eaisly. There are four functions in the **<ucontext.h>** that I mainly utilized:

- **int getcontext(ucontext_t *ucp)**

  This function gets the currently context and return in a structure.

- **int setcontext(const ucontext_t *ucp);**

  We can use this to switch to the context specified by ucp

- **void makecontext(ucontext_t *ucp, void (*func)(), int argc, …);**

  After we get the current context, if we want to modify it which means to set the entry point function, we need to call this function and input the entry and arguments.

- **int swapcontext(ucontext_t *oucp, const ucontext_t *ucp);**

  This funcion enables us to easily switch to a context specified by ucp and store the current context in the oucp for later use.

So, when creating a thread, we allocate its runtime stack and specify the context it returns to when the thread is finished. In my implementation, when a thread exits or yield, it returns to the scheduler.

The preemption is thus easily realized by switch the currnt context from thread context to the scheduler context.

4.2 Timer

In order to implement the Round-Robin scheduling, we need a time recorder. The hardware provides us the mechanism to send a signal during each time period. And we use the functions in **<signal.h>** and **<sys/time.h>**

The function "setitimer(ITIMER_REAL, &tick, NULL)" allows us to set up a timer that will send a signal during a fixed time period(for example, every 0.1s). And use the command "signal(SIGALAM, func)" to specify our handler when it receives a time signal. In this handler, we simply swap the current context and the scheduler context.

A little tricky here is that, this signal could be catched even during a ucontext switch. So we need to be super careful with the synchronization problem here.

5. Project structure

The code is structure using CMake 2.8, which can easily generate either a Unix Makefile project, or VS for Windows users.

The library has two files

- **<uthread.h>** includes all the functions statements and the parameters, like the stack size for each thread and length of the time slice. This file needs to be included by users.
- **<uthread.cpp>** Main body of the library. The code is commented with detail, see for yourself.

6. Test and experiment

All codes for testing are in **<test.cpp>**, which has a main function.

Two tests are designed trying to cover the execution path as much as possible

- **Test1** (**Detail result in file 'Debug/test1.out'**)

This one is simple, in order to cover the basic function.First five threads are created and keeps printing hello to the screen and then sleep for 2 seconds. The result is that the five threads take turns to say hello.

Here is a screenshot of the results.

```
test1 started
thread tid 0 created
thread tid 1 created
thread tid 2 created
thread tid 3 created
thread tid 4 created
Hello from thread number 0
Hello from thread number 1
Hello from thread number 2
Hello from thread number 3
Hello from thread number 4
Hello from thread number 0
Hello from thread number 1
Hello from thread number 2
Hello from thread number 3
Hello from thread number 4
Hello from thread number 0
Hello from thread number 1
Hello from thread number 2
Hello from thread number 3
Hello from thread number 4
scheduler finsihed
test1 finished
```

- **Test2 (Detail result in file 'Debug/test2.out')**

This experiment is a little more complexed to cover the "thread_join" function and the preemtive scheduling.

First, five greeting threads are created and print "how are you?" on the screen. Then, five responding threads are created and use "thread_join" to wait for the corresponding greeting threads to finish asking. After that, print "I am fine, thank you and you", which is an "universal-true answer":)

To test the preemptive scheduling, each greeting thread will be given a random amount of useless work so the greeting and the answer may be in random order. But an answer will always be after a greeting.

Here is a screenshot:

```
test2 started
thread tid 0 created
thread tid 1 created
thread tid 2 created
thread tid 3 created
thread tid 4 created
thread tid 5 created
thread tid 6 created
thread tid 7 created
thread tid 8 created
thread tid 9 created
Thread 5: How are you?
Thread 9: How are you?
Answer for thread 5: I am fine, thank you and you?
Answer for thread 9: I am fine, thank you and you?
Thread 8: How are you?
Answer for thread 8: I am fine, thank you and you?
Thread 6: How are you?
Answer for thread 6: I am fine, thank you and you?
Thread 7: How are you?
Answer for thread 7: I am fine, thank you and you?
scheduler finsihed
test2 finished
```