THE UNIVERSITY OF AUCKLAND
NEW ZEALAND

COMPSCI 340 / SOFTENG 370
Operating Systems

Assignment 1 - Making Threads

Worth 7%

final date 11:59pm 21st of August, 2017

Introduction

User level threads are surprisingly easy to make, as long as the operating system provides a few basic requirements. In this assignment you are going to finish writing a thread library using some of the standard Unix system calls.

Threads without preemption

The file `OSA1.c` is a program which creates a thread, executes that thread and then on completion of the thread returns to the main function (which is executing in a sort of thread too).

This program runs fine on the Mac and Linux (including the Windows Subsystem for Linux). However I recommend using Linux in the lab or in a virtual machine because later parts of the assignment (in particular Part3) become OS dependent. The markers will test the programs on the lab Linux image.

Compile and run the program either from the command line:

```
gcc OSA1.c -o OSA1
./OSA1
```

or from within an IDE such as Eclipse. I recommend the command line.

In this case there was no need for a thread as the main function merely waits for the thread to finish before it finishes, just like a very fancy subroutine call. However by the end of the assignment you will have a proper pre-emptive threading system.

The first thing you need to do is to read and understand the code in `OSA1.c`. I will also cover this in lectures.

What is a thread?

The `littleThread.h` header file defines a thread structure as:

```
typedef struct thread {
      int tid;                  // thread identifier
      void (*start)();          // the start function
      jmp_buf environment;      // saved registers
      enum state_t state;       // the state
      void *stackAddr;          // the stack address
      struct thread *prev;      // pointer to the previous thread
      struct thread *next;      // pointer to the next thread
} *Thread;
```

So a thread has a unique identifier `tid` (the first thread is 0, add 1 for each subsequent thread), a `start` function, this is the function which gets called when the thread begins executing. It has an `environment` (which is used by `setjmp` and `longjmp`), a `state`, the address of the bottom of the thread's stack `stackAddr` and pointers to previous and next threads (not used in `OSA1.c` at the moment but you will have to use them). The thread states are simply: `SETUP, RUNNING, READY,` and `FINISHED`. `SETUP` means that the thread is in the process of being created or setup. `READY` means the thread is all set to run but it is not running. `RUNNING` and `FINISHED` are obvious.

The environment of a thread includes the values of the processor's registers so that the thread can resume after not running for a while. See the section on `setjmp` and `longjmp` later in this handout.

The code for the thread is in `threads0.c`. This way the markers can supply different threads to test your code.

What about the stack?

Each thread requires its own stack. This could be allocated by reserving some space in memory and then setting the stack pointer register to the top of this space. However there is a way of doing this without having to use any assembly language.

When signals are sent to Unix processes they normally execute the signal handler on the normal user stack, but it is possible to request that when a signal handler is called it uses an alternate stack. This is partially set up in the `setUpStackTransfer` function.

The `createThread` function allocates space for the stack (and the thread structure). Before it sends a signal to the current process with the `kill` system call it also makes the new stack for the thread act as the signal stack, using the `sigaltstack` function.

Then when the `SIGUSR1` signal is received it will cause the process to jump to the handler function `associateStack` using the new stack for this function.

Each Part

There are 3 parts to this assignment. Do each part in its own directory: `Part1`, `Part2`, `Part3`. Each directory must have your source code for the part: `OSA1.1.c`, `OSA1.2.c`, `OSA1.3.c`, the unmodified header `littleThread.h` and the example thread file for the part: `threads1.c`, `threads2.c`, `threads3.c`.

When the markers are testing your code they will first run each part using the code you provide then they will replace the `threads?.c` files with other thread files to make sure your code works with different threads and different numbers of threads.

Part 1

Take `OSA1.c` change its name to `OSA1.1.c` and put your name and login/UPI in the opening comment. You have to change the code so that if multiple threads are created they will be switched between **as each thread finishes**. There is no pre-emption. After all threads have finished, execution returns to the main function and the process stops. You can change the existing code in any way you like as long as it still works in the same way. You will certainly have to change the `switcher` function.

You should keep all created threads in a circular linked list (that is what the `prev` and `next` pointers are for in the thread struct). You will need to keep track of the current thread (the thread currently running), this will initially be the first thread created. Always insert new threads at the end of the list. Each thread must have a unique thread identifier.

The main thread (the one executing the `main` function) is different, it doesn't have an identifier and should not be inserted in the circular list of threads. It should only be returned to when there are no other threads still able to run.

When the currently running thread finishes you need to remove it from the list and switch to the next thread in the list. You must free the stack space of all finished threads. Keep the message saying you are disposing of the stack space for the finished thread. Do not free the thread structure memory as you need this to print the state of the thread.

Add a scheduler function called `scheduler` to choose the next thread to run. You need to ensure that all thread states are correct. e.g. Threads which have completed have their state changed to `FINISHED`, threads which are waiting to run are `READY` and only the currently executing thread is `RUNNING`.

Add a new function `printThreadStates` which prints out the thread ids and the state of each thread in the order they were created. The const `NUMTHREADS` is the number of threads created. Output should look like this:

```
Thread States
=============
```

```
threadID: 0 state:running
threadID: 1 state:ready
```

In this case thread 0 is the current thread, and hence running, and thread 1 is ready to run. Even when threads have finished their state should still be shown by the `printThreadStates` function.

Call the `printThreadStates` function whenever there is a switch to a new thread. Also call it after creating the threads but before any of them are started, and one last time just as the process finishes.

The output when using the `threads1.c` file must be like this (extra empty lines and spaces can be ignored):

```
robert@ubuntu:Part1$ ./OSA1.1

Thread States
=============
threadID: 0 state:ready
threadID: 1 state:ready

switching to first thread

Thread States
=============
threadID: 0 state:running
threadID: 1 state:ready

hi
hi
hi
hi
hi

disposing 0

Thread States
=============
threadID: 0 state:finished
threadID: 1 state:running

bye
bye
bye
bye
bye

disposing 1

back to the main thread

Thread States
=============
threadID: 0 state:finished
threadID: 1 state:finished
```

Part 2

Copy `OSA1.1.c` from **Part 1** and save it as `OSA1.2.c`. Use the `thread2.c` file for the threads. Add a `threadYield` function to `OSA1.2.c` which will call the scheduler. This should immediately stop the current thread from running and start up the next `READY` thread in the circular linked list. If the current thread is the only existing `READY` thread then it continues.

The output should look like this:

```
robert@ubuntu:Part2$ ./OSA1.2

Thread States
=============
threadID: 0 state:ready
threadID: 1 state:ready
threadID: 2 state:ready

switching to first thread

Thread States
=============
threadID: 0 state:running
threadID: 1 state:ready
threadID: 2 state:ready

hi

Thread States
=============
threadID: 0 state:ready
threadID: 1 state:running
threadID: 2 state:ready

bye

Thread States
=============
threadID: 0 state:ready
threadID: 1 state:ready
threadID: 2 state:running

bye

Thread States
=============
threadID: 0 state:running
threadID: 1 state:ready
threadID: 2 state:ready

hi

Thread States
=============
threadID: 0 state:ready
threadID: 1 state:running
threadID: 2 state:ready

bye

Thread States
=============
threadID: 0 state:ready
threadID: 1 state:ready
threadID: 2 state:running

bye

Thread States
=============
threadID: 0 state:running
threadID: 1 state:ready
threadID: 2 state:ready
```

```
disposing 0

Thread States
=============
threadID: 0 state:finished
threadID: 1 state:running
threadID: 2 state:ready

disposing 1

Thread States
=============
threadID: 0 state:finished
threadID: 1 state:finished
threadID: 2 state:running

disposing 2

back to the main thread

Thread States
=============
threadID: 0 state:finished
threadID: 1 state:finished
threadID: 2 state:finished
```

Part 3

All we need to add now is a way of pre-empting a running thread to enable other threads to get a turn. We can do this with another signal. Look up how to use `setitimer` to send a signal to your process every 20 milliseconds (how many microseconds?). This is where the assignment becomes OS dependent. You must use `ITIMER_VIRTUAL` and its corresponding signal `SIGVTALRM`.

Save the program either from **Part 1** or **Part 2** as `OSA1.3.c` . Add `setUpTimer` and a call to it, just before starting the threads in `threads3.c`. The output should show all three threads taking turns running, similar to this:

```
robert@ubuntu:Part3$ ./OSA1.3

Thread States
=============
threadID: 0 state:ready
threadID: 1 state:ready
threadID: 2 state:ready

switching to first thread

Thread States
=============
threadID: 0 state:running
threadID: 1 state:ready
threadID: 2 state:ready


… (lots of missing output)

Thread States
=============
threadID: 0 state:running
threadID: 1 state:ready
threadID: 2 state:ready

hi
1296337872
hi
```

```
Thread States
=============
threadID: 0 state:ready
threadID: 1 state:running
threadID: 2 state:ready

bye

… (lots of missing output)

Thread States
=============
threadID: 0 state:finished
threadID: 1 state:finished
threadID: 2 state:running

910746567

Thread States
=============
threadID: 0 state:finished
threadID: 1 state:finished
threadID: 2 state:running


Thread States
=============
threadID: 0 state:finished
threadID: 1 state:finished
threadID: 2 state:running

bye
767147961
disposing 2

back to the main thread

Thread States
=============
threadID: 0 state:finished
threadID: 1 state:finished
threadID: 2 state:finished
```

The output will be different every time the program is run.

Questions (put the answers to these in a file called A1answers.txt)

Question 1

There is a comment saying `"Wow!"` in the `switcher` function in `OSA1.c`. This is to indicate something very bad is happening here. Explain what it is.

Question 2

Why are the time consuming calculations in `threads3.c` required in order to demonstrate the effectiveness of the pre-emptive scheduler?

Question 3

In `threads3.c` there is some code around the call to `rand()` to block signals and then allow them again. Explain what can happen if this is not done. Also give an explanation as to why this can happen.

Extra info

- There are definitely differences in the behaviour of programs like this on different Unix/Linux operating systems. For this reason you eventually have to get your solution working on the lab image of Linux.
- As one example, I found that the Windows subsystem for Linux wouldn't allow me to use `ITIMER_VIRTUAL` when calling `setitimer`, I had to use `ITIMER_REAL` instead. Whereas `ITIMER_VIRTUAL` worked fine on an Ubuntu virtual machine on my Mac.
- Remember that the Unix `man` command gives very useful information about system and C function calls, in particular for this assignment: `malloc`, `sigaction`, `sigaltstack`, `setitimer`, `kill`

setjmp and longjmp

The C functions `setjmp` and `longjmp` are used to store the current environment (actually the registers) and restore them at a later time. This means we can effectively store the computation of a thread at this point and then come back to this point when we want to. Just what we need to freeze and unfreeze a thread. But because C uses a stack to keep track of function calls (the corresponding arguments, local variables and return addresses) we must make sure that `longjmp` is called before the stack has lost or overwritten the position where `setjmp` was called.

```
setjmp(thread->environment)
```

will save the current registers in the `environment` field of the `thread` structure. When `setjmp` is called it returns 0. When we come back to exactly the same position in the code because of the matching `longjmp` we can return another value (traditionally 1) so that we can do different things when we return.

```
longjmp(thread->environment, 1);
```

In this way a thread can be setup to run by skipping the code to call its start function when the return value from `setjmp` is zero. When we do a `longjmp` with one as the return value back to the same point the thread's start function is executed. See the `associateStack` and `switcher` functions in `OSA1.c` .

Part 4 - Extra for Experts (just because you can)

Save your `OSA1.3.c` file as `OSA1.4.c` in a `Part4` directory and modify it in the following way. Rather than use the clever but hacky system to set up a new stack for each thread using an alternate signal stack, you have to descend into the murky depths of assembly language to modify the stack pointer directly.

You will have to look up inline assembly on the Web in order to change the stack pointer for each thread.

Remove all the unnecessary pieces of code which are to do with alternate signal stacks and the associated `kill` call.

You could also use `setcontext` and `getcontext` for switching between contexts rather than using `setjmp` and `longjmp`.

Submit

Zip up all of your solutions into one file (`Part1`, `Part2`, `Part3` (and possibly `Part4`) directories and the `A1answers.txt` file).

Submit the zipped file to Canvas. If you resubmit Canvas adds a suffix to the filename. Do not worry about this.

Remember: Your code is going to be tested on the Linux lab image. If your code does not run correctly in this environment you will not get the marks :(.

Include your name and UPI in all files.

Marking guide (28 marks = 7%)

Students name and UPI in every file.

1 mark

Part 1 - OSA1.1.c

There is only the one call to the `switcher` function in the main function.

1 mark

There is a simple `scheduler` function which selects the next thread to run. The threads are maintained in a circular linked list.

1 mark

The stack space for a thread is freed when the thread completes.

1 mark

There is a `printThreadStates` function which prints out the state of all threads in the order of creation of the threads. Even finished threads are shown.

2 marks

Works correctly using given `threads1.c`.

2 marks

Works correctly using different `threads1.c`.

2 marks

Part 2 - OSA1.2.c

There is a `threadYield` function which schedules the next thread to run.

1 mark

Works correctly using given `threads2.c`. The program terminates cleanly by returning to the main function when the threads have all finished.

1 mark

Works correctly using different `threads2.c`.

2 marks

Part 3 - OSA1.3.c

Using given `threads3.c` the program output shows that the threads are being pre-empted.

2 marks

The program terminates cleanly by returning to the main function when the other threads have all finished.

1 mark

Works correctly using different `threads3.c`.

2 marks

Part 4 - OSA1.4.c (no marks but lots of fun)

Question 1

3 marks

Question 2

2 marks

Question 3

4 marks