ECE 434 Introduction to Computer Systems
Project 2: 03-12-2017
Signal Handlers and User Level Threads
Due Date: Monday April 7th 2017, 10.00pm

PART 1

1 Introduction

This part is a slight elaboration of your Project 1. Please modify and expand your Project 1/Question D to do the following tasks:

a) The IPC is now to be conducted with signals and signal handlers but not with Pipes or Shared Memory. You should go over the available signals and figure out how to pass information from the parent to the child and vice versa through signals/interrupts/signal handlers.

b) Your parent process becomes now very impatient…. It does not want to wait for any child process longer than 3 seconds. If a child process takes longer than this, do not incorporate the contribution of this process to the final result. And not only this alone, but the parent should "mark" the slow behaving processes and "notify" all its children processes about "who" this misbehaving process is. And then, have any or all of the rest of children processes attempt to terminate this one (before the parent comes in). What do you observe if more than one process attempts to terminate a given process? Write the proper instructions that ensure all the questions in this part, and figure out the proper signals to be incorporated again either by the parent or the children processes.

c) The user running this program (… this is your group) got very impatient while waiting for all these calculations to be conducted, and decides to be done with the whole experiment by typing: i) CTL-C, ii) select your own interrupt, iii) also the user may want to experiment and attempt to terminate certain processes (children or parent). Your program should contain signal handlers that take care of these actions and give back to the user information about what type of interrupt was issued, on which process, and you may print any other related output that helps the user get more info about the process. Also, you may want to change the handlers to "block" the processes from termination or suspension or in general of accepting the signals. We have defined 3 cases above (i, ii, iii). When can you do the latter and when not? For the cases that you can, please show how you have incorporated this capability to your code and your results.

PART 2

1 Introduction

In this project you are expected to implement your own thread library. Through your involvement you will gain experience with multi-threaded systems.

2 User Level Thread Library

For this part you will implement a cooperative User Level Thread (ULT) library for Linux that can replace the default PThreads libary. Cooperative user level threading is conceptually similar to a concept known as coroutines, in which a programming language provides a facility for switching execution between different contexts, each of which has its own independent stack. A very simple program using coroutines might look like this:

```
void coroutine1()
{
        // Some work
```

```
        yield(coroutine2);
}

void coroutine2()
{
        // Some different work
        if (!done)
            yield(coroutine1);
}

int main()
{
        coroutine1();
}
```

Note that cooperative threading is fundamentally different than the preemptive threading done by many modern operating systems in that every thread must yield in order for another thread to be scheduled.

2.1 Basic User Level Thread Library

Write a non-preemptive cooperative user-level thread (ULT) library that operates similarly to the Linux pthreads library, implementing the following functions:

- mypthread_create
- mypthread_exit
- mypthread_yield
- mypthread_join

Please note that we are prefixing the typical function names with ”my” to avoid conflicts with the standard library. In this ULT model one thread yields control of the processor when one of the following conditions is true:

- thread exits by calling mypthread_exit
- thread explicitly yields by calling mypthread_yield
- thread waits for another thread to terminate by calling mypthread_join

You should use the functionality provided by the Linux functions setcontext(), getcontext(), and swapcontext() in order to implement your thread library. See the Linux manual pages for details about using these functions. These functions allow you to get the current execution context, make new execution contexts, and swap the currently running context with one that was stored.

So, what is a context? It is related to *context switches*—when one process (or thread) is interrupted and control is given to another. It’s called a context switch because you are changing the current execution context (the registers, the stack, and program counter) for another. We could do this from scratch, using inline assembly, but the **getcontext, makecontext, and swapcontext functions** make it much simpler. For example, when getcontext is called, it saves the current execution context in a struct of type ucontext_t. The man page for getcontext describes the elements of this struct. Some of these elements are machine dependent and you don’t have to worry about the majority of them. They may include the current state of CPU registers, a signal mask that defines which signals should be responded to, and of course, the call stack. The call stack is the one element of this struct that you will need to pay some attention to.

The current context must be saved first using getcontext (the new thread's context is based on the saved context). Space for a new stack must be allocated, and the size recorded. Finally, makecontext modifies the saved context, so that when it is activated, it will call a specific function, with the specified arguments.

The newly created context is then activated with either a call to setcontext or swapcontext. The former (setcontext) replaces the current context with a stored context. When successful, a call to setcontext does not return (it begins running in the new context). A call to swapcontext is similar to setcontext, except that it saves the context that was running (a useful thing to do, if you later want to resume the old context later). A successful call to swapcontext also does not return immediately, but it may return later, when the thread that was saved is swapped back in. You probably want to store your thread contexts on the heap.

2.2 Requirements

We will provide a test program (mtsort.c), and simple template (mythread.h). You may not modify the test program, so your library must provide exactly the API that we specify. You should implement all of your code in the provided files mypthread.h, and mypthread.c.

3 Coding and Submission Instructions

3.1 Coding

For this assignment, you should NOT have to create any other files, simply do your implementation in these existing files (use mythread.c, mythread.h, mtsort.c). Either provide a ReadMe file with instructions how to run this program or generate your own makefile.

3.1.1 Implementation

We provide a mypthread.h that defines the required API. You will need to make changes to the types defined here, but do not change the function call API. You will implement your library in mypthread.c. Once you complete your implementation, you may test it using the test program provided. The test program is implemented in mtsort.c.

3.2 Submission

Only one group member should submit the project, but the names of all group members should be entered into the text field on the Sakai submission, and should also appear on the report. A detailed report on justification of your coding and discussion is expected for your project to be complete.

4 Notes

The code has to be written in the C language. Use Sakai to submit it.