

# 91.516 Operating Systems II

Assignment #1 Due June 11

May 28, 2009

UNIX processes are often built to be singly threaded, in which case they have only one user code path using a single stack. Most newer versions of UNIX have support for the **pthread** library, however, and could support processes with multiple threads. Before the POSIX standards group adopted pthreads, programmers were often on their own as far as threading support was concerned, but UNIX has included system call support for crafting a user space thread package for many years.

A process with more than a single asynchronous thread can be structured using the UNIX software context management system calls **makecontext()**, **swapcontext()**, **getcontext()** and **setcontext()**.

In the text you'll find that the basis of a multi-process operating system is really in its **context switch's** ability to save state for a code path running on one stack and begin executing a code path running on a new stack. To simulate this behavior in the UNIX environment will require an **in-process context switch** which can share the processes **quantum** with several code paths, each running on a separate stack.

To allow for the asynchronous execution of multiple threads it is not only necessary to have an in-process context switch, but the context switch must be executed periodically (or under particular circumstances) by threads which are ready to temporarily **yield** a process's remaining quantum time to another thread which can use it. The UNIX signal mechanism can supply the necessary stimulus to effect a context switch.

Signals can be delivered to processes in several ways, but a convenient mechanism for this requirement is to set an **interval timer** to deliver an "alarm clock" type signal when it's time for one thread to yield the CPU to another. The **setitimer()** system call will do this for a process to at least a **10ms granularity** on most systems. When the alarm clock expires, a signal is delivered to the running process which causes the process to save its **current execution state** and move to the code which the programmer installed to handle the alarm signal. This code, of course, is the essence of the context switch. Return from this code will be to another waiting code thread which will now begin to make progress again (or for the first time).

Since threads consist of code which will likely make calls to other functions, each thread must have its own private stack. When your main program starts it will have just a single stack, so you must **allocate heap space for each thread** you expect to start and use this space for that thread's stack. In the **reference code**

I've provided you (this is on the web site), I use a **state buffer** for each thread you will create, and this structure keeps track of a thread. This means that the initial state buffer that you set up for a thread must contain the address of that thread's private stack space in the stack pointer field. The **initial PC** field in the state buffer will have to be set to the thread's **initial function address**.

```
typedef struct state_block{
    int      state;
    void      (*function) ();
    int      function_arg;
    ucontext_t  run_env;
    ucontext_t  rtn_env; ← need to set this up
    stack_t  sigstk;
}STATE_BLOCK;
```

This assignment requires you to write a simple threads package which lets you start up to **10 threads**. Each thread will be packaged in its own top level function and will minimally execute a print statement that indicates when the thread is running. The simultaneous output from the various threads on the terminal widow should show that the process is **clearly switching its time among the threads**.

The reference code I've provided does much of this, but the individual functions which run as threads, currently are required to call directly into the signal handler when they are finished. You must **change this behavior**, so that a thread **simply has to return** when it is done, and the system will somehow know that this thread is done and will continue with any remaining threads or do a full exit if all the threads are done. You have to clearly describe your solution to this problem in your write-up. (**Hint: check out the uc\_link field in the ucontext\_t structure.**)

You must pass in a printed report which includes a properly completed **Course Cover Sheet**, and a **hard copy of your source code** with a minimum **one page write-up** which describes your approach and your **success level**. The code may be developed on any UNIX system, but should be eventually placed on one of the CS systems such that I can read and execute it. Please make sure your that the Course Cover Sheet includes the path name(s) of where I can find your code, and make sure that all directories in the path have at least **x** permission and that the final target files have **r** permission. This is an initial attempt to produce a **context switch**, and in a later assignment we'll explore the possibility of running the **XINU environment** over this context switch. We should be able to produce a good UNIX simulation with your context switch, the XINU source code from the text and an emulated console, stay tuned.