

# 91.516 Operating Systems II

## Assignment #1 Due February 9

### January 26, 2012

Linux/UNIX processes are often built to be singly threaded, in which case they have only one user code path using a single stack. Most versions of Linux/UNIX have support for the **pthread** library, however, and can 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 Linux/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 POSIX 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 Linux/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 Linux/UNIX signal mechanism can supply the necessary stimulus to effect an in-process context switch.

Signals can be delivered to processes in several ways, but a convenient mechanism for this assignment 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 **1ms 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. This code will serve as the agent to transfer control to another ready 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\_BLOCK** structure for each thread created, as a destination for saving thread state and set-up parameters. This is my own local structure to track the thread, but the actual structure that needs to be managed by the system is of type `ucontext_t`, and you can see that I have two instances of this type called `run_env` and `rtn_env` in my local structure. The reference code is only using the `run_env` instance, and the **point of this assignment** is learn how to use the `rtn_env`. So, if I'm about to create a new user-space thread, I need to:

- find a free stateblock (from an array of stateblocks)
- change its state from free to initialized
- fill in the function field with the address of the function this thread will run
- malloc space for a stack for this thread, and set up the `stack_t` entry
- initialize the `run_env` and the `rtn_env`
- use the `makecontext()` system call to integrate previously set fields into the `rtn_env` and `run_env` structures

```
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 as many as **10 threads** (you only need to start 3 or 4 to demo your results). Each thread will be packaged with 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.**)

1. Your submission must be made electronically using the submit command as described below.

2. **All** of your submissions must include a minimum of **four** separate files:
  - **File 1:** A short **write-up** that **first** specifies what you think your **degree of success** with a project is (**from 0% to 100%**), followed by a brief discussion of your approach to the project along with a **detailed description** of any problems that you were **not** able to resolve for this project. **Failure to specifically provide this information will result in a 0 grade** on your assignment. If you do **not disclose** problems in your write-up and problems are detected when your program is tested, you will receive a grade of 0. **Make sure that you include your email address in your write-up so that the corrector can email you your grade.**
  - **File(s) 2(a, b, c, ...):** Your **complete source code**, in one or more **.c** and/or **.h** files
  - **File 3:** A **make file** to build your assignment. This file must be named **Makefile**.
  - **File 4:** A file that includes your **resulting output** run(s) from your project. This is a simple text file that shows your output, but make sure that you annotate it so that it is self descriptive and that all detailed output is well identified.
  
3. The files described above should be the only files placed in one of your subdirectories, and this subdirectory should be the target of your submit command.

The submit syntax issued from a shell prompt is will be provided in class and can be checked for details on the class website at:

[www.cs.uml.edu/~bill/cs516/SubmitDetails.pdf](http://www.cs.uml.edu/~bill/cs516/SubmitDetails.pdf)

Your code may be developed on any Linux/UNIX system, but should be eventually placed on one of the CS Linux systems for submission. 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 XINU simulation with your context switch, the XINU source code from the text and an emulated console, stay tuned.