CS6456 (F2016) Operating Systems

Project 2

Title: Creating a User-Level Thread Package (Part I)
Due: October 4, 2016 (11:00 a.m.)
Points: 10

Project Description

In this project, you will gain experience in creating and context-switching among concurrent threads sharing an address space. UNIX classically provided the notion of a *process* for an activity and switches between processes to support concurrent activities. The address space (code, data, and stack) together with the registers and other state information constitute the state of a process, and these have to be saved and restored when the operating system switches from one process to another. In this project, the unit of activity is a *thread* instead of a process. A thread adds an extra set of CPU state (PC, registers, etc.) and a stack, but shares the rest of the address space with other threads.

You will have a single UNIX process, and you will need to create and manage multiple threads within this UNIX process and address space. In other words, you will implement *user-level* or *many-to-one* threads. The UNIX kernel will still see this as one single process, and it is up to the library you develop to provide the thread creation/termination/management routines to the user program. The interface routines of the library to be implemented are described below. Use the information given in this project as a higher level description of what you are expected to implement. Sample programs to assist in testing your implementation are available as additional attachments for this project on collab.

You should be able to switch from one thread to another on an explicit *uthread_yield()* call which a thread can invoke. Remember that, unlike processes, all the threads of a process share the same code and data. Therefore, you do not have to switch the code and data when a thread yields. Only the stack pointer, registers etc. need to be saved and restored. In addition, you should implement a form of *FCFS* scheduling scheme to switch among runnable threads upon a yield. Note that in this first stage of implementation, we are only supporting non-preemptive scheduling. In the second part of this project, we will add preemptive scheduling, priorities, and support for synchronization among threads.

For this project, you may use the *getcontext(), makecontext(),* and the *swapcontext()* family of system calls to handle context-related operations; using other routines that directly manipulate contexts is not recommended.

Library Interface

Your project will be implemented in Linux using C/C++. You are free to develop on any UNIX/Linux system, but it will be evaluated in the Debian environment. The API calls you must support are as follows.

- **int uthread_init()**
  Initialize the thread library. The function shall return 0 when it succeeds and -1 otherwise. For this project, the call shall fail when the library is already initialized.

- **int uthread_create(void (*func)(int), int val, int pri)**
  Create a new thread with priority *pri* that starts execution with function *func()* whose argument is *val*; i.e. the prototype of *func* is *void func(int val)*. The *uthread_create()* function returns the identifier of the thread thus created (the thread ID starts from 0). For now, the *pri* field is unused (just pass 0 to the function when it is called). When creating a thread, a stack of 1 MB is allocated. The created thread should not run immediately; it should be put at the end of the queue of threads waiting to run.

- **int uthread_yield()**
  For the currently running thread to relinquish the CPU. Remember that we are only supporting non-preemptive scheduling so far, but when the current thread yields, if you have multiple threads suspended due to their having yielded, you should switch to the one that has been suspended the longest (i.e. *FCFS*). On UNIX, this call always succeeds. So your program may just return 0 after the call is successfully executed.

- **void uthread_exit(void *retval)**
  Terminate the currently running thread and return a value via *retval* that is available to another thread in the same process that calls *pthread_join*. The call also makes the scheduler select another thread to run, if there is one. If the exiting thread is the only thread in the process, the entire process should exit. Also note that when the system call *exit()* is invoked in a thread for any reason (e.g. *uthread_create()* failed), the entire process, with all its threads, should exit. Calling *uthread_exit()* is the *only* way a thread should exit (unless it is using *exit()* to terminate the entire process). This means a thread should not return from the *func()* specified in *uthread_create()*. If it does return from *func()* without calling *uthread_exit(),* you may handle this case any way you choose, except that your program should not crash. Note that the way we handle the case for the project is the same as that

handled by UNIX in that behavior when returning from *func()* without calling *uthread_exit()* is undefined. Note also that the function always succeeds and returns nothing.

We will test your library with various test programs that call your thread package via the above API. We will link our test programs against your file implementing this API.

You may use the following command to compile your library program (e.g. threadsLib1.c) and link it to my test program (e.g. myApp.c) to test your program: gcc myApp.c threadsLib2.c –o myApp -wall

Useful UNIX System Calls

- **To create a new thread:** *getcontext()*, then *makecontext()*
- **To switch between threads:** *setcontext()* or *swapcontext()*

In your written evaluation (at least two pages long in **PDF**), be sure to discuss your implementation, specifically: (1) how each of the functionalities above is implemented; (2) the design decisions you made for each functionality; and (3) an explanation of why you made those decisions.

Miscellaneous

(1) The project does NOT allow team work.
(2) You must use a makefile to compile your program.
(3) Your program must be compiled and run successfully in the Debian VM.
(4) Your implementation must be in C/C++ and use the POSIX APIs for threads.
(5) Turn your project in via **collab** in a **tar** file consisting of (i) all headers you have created; (ii) source code file for project 2 (you may use any file name for your library program, although **threadsLib1.c** is a good one to choose); (iii) a make file; and (iv) a write-up showing your design and implementation of the project (at least two pages long in **PDF** format). The **tar** file should be named as follows: **p2**, followed by your computing ID, and followed by the file extension (i.e. **tar**). For example, the **tar** file turned in by **John Smith** should be named **p2js7nk.tar**, where **js7nk** is the computing ID of **John Smith**.
(6) Your program should be successfully compiled. Programs failing to compile will NOT receive any points.