COSC 301: Operating Systems
Lab 7: Threads Alive!
Due: 16 November, 2011
Fall 2011

1 Overview

In this project, we will write a user-level library to support multiple threads within a single Linux process. Our library will be written entirely in user mode, thus it will be non-preemptive. However, much of the code we write will be identical to how a kernel-level threads library would be written, including context switching, scheduling, and implementation of synchronization mechanisms.

The goals for this lab are as follows:

- To learn how threads really work
- To learn how to build a real (and fast?) mutex lock and a real semaphore
- To learn how those pesky condition variables really work from an implementation standpoint

As with the previous lab, work on this lab in groups of 2 or 3. Consider working with someone new for this lab.

Note: there are **two** stages to achieve in this lab. Each stage requires more sophisticated programming and provides more features in your thread library. The stages are structured such that you can still achieve a good grade even if you only get through stage 1. **Work through the stages sequentially**, e.g., don't try to incorporate stage 2 features unless you're confident that everything works well for stage 1. (There is a stage 3: it is entirely optional. Credit in the form of bragging rights only.)

Several files are posted on Moodle to help get started on the various stages of this lab:

SConstruct: scons configuration file for building everything.

threadsalive.h: The main header file for the threads library.

threadsalive.c: The main implementation file for threads library.

ctxtest.c: An example program using the Linux system calls needed to implement the user-level threads library.

test01.c: An example test program that uses stage 1 functionality.

test02.c: An example test program that uses stage 2 functionality.

test03.c: An example test program that uses stage 3 functionality.

cas.c: An example assembly function to use atomic compare and swap instruction available on the Intel x86 architecture.

2 Stage 1 functionality (40 out of 50 points)

In the first stage you will add the following functions to you4 thread library:

Signature	Usage
`void ta_libinit()`	this function is called to initialize the thread library. if if your library does not require any initialization, this function should not do anything but must be present.
`void ta_create(void (*func)(void *), void *arg)`	`ta_create` is used to create a new thread. When the newly created thread starts, it will call the function pointed to by func and pass it the single argument arg.
`void ta_yield(void)`	`ta_yield` causes the current thread to yield the CPU to the next runnable thread.
`int ta_waitall()`	when the main thread calls this, it should wait for all other threads to finish. If all threads run to completion, `ta_waitall()` should return 0. If threads still exist but none are runnable, `ta_waitall()` should return -1.

2.0.1 Creating and Swapping Threads

You will be implementing your thread library on x86 machines running the Linux operating system, as usual. Linux provides some library calls (`getcontext()`, `setcontext()`, `makecontext()`, `swapcontext()`) to help implement user-level thread libraries. `getcontext()` and `makecontext()` are used for creating and modifying a thread context; `setcontext()` and `swapcontext()` are used for transferring execution control to a specific thread context.

The basic data type that is used with these calls is `ucontext_t`. This structure contains two important fields: the stack for the thread context, and a "link" to another context. The stack that you allocate for each thread in your library should be from the heap; a stack size of 128KB or greater is adequate for this lab. The link field points to another thread's context; when the thread finishes execution (i.e., the function in which the thread started is done), the operating system will automatically do a context switch to the thread context referred to by the link pointer. This link pointer can point to any thread context; if it points to NULL, as soon as the thread finishes, your entire program will exit.

Below is an example of how these routines (except setcontext(), which you may not need to use) can be used (the same example code is posted on Moodle as `ctxtest.c`):

```
#include <stdio.h>
#include <stdlib.h>
#include <ucontext.h>
#include <assert.h>

/*
 * three thread contexts.  two for two user-mode threads, and the
 * other to store the state for the main thread in the process.
 * ctx[0] is the saved context for the main thread.
 * ctx[1] is for thread 1.
 * ctx[2] is for thread 2.
 */

static ucontext_t ctx[3];
static int shared_variable = 0;

/*
 * function entry point for the two threads.
 */
static void mythread(int threadid, int x)
{
    shared_variable += 1;
    printf("in thread %d; x is %d and shared_variable is %d\n",
           threadid, x, shared_variable);

    /* do a context switch from thread 1 to thread 2 (or from 2 to
     * 1)  */
    int otherthread = (threadid == 1) ? 2 : 1;
    swapcontext(&ctx[threadid], &ctx[otherthread]);

    shared_variable += 1;
    printf("back in thread %d; shared_variable is %d\n",
           threadid, shared_variable);
}
```

```
/*
 * good ol' main().
 */
int main(int argc, char **argv)
{
#define STACKSIZE 8192
    unsigned char *stack1 = (unsigned char *)malloc(STACKSIZE);
    unsigned char *stack2 = (unsigned char *)malloc(STACKSIZE);

    assert(stack1 && stack2);

    /* initial context for thread 1 */
    getcontext(&ctx[1]);
    /* set up thread 1's stack */
    ctx[1].uc_stack.ss_sp   = stack1;
    ctx[1].uc_stack.ss_size = STACKSIZE;
    /* set up thread 1's link: when thread 1 exits, the main thread
     * (context 0) will take over */
    ctx[1].uc_link          = &ctx[0];
    /* set the thread entry point (function) for thread 1 */
    /* pass 2 argument (int values 1 and 13) to the function */
    makecontext(&ctx[1], mythread, 2, 1, 13);


    /* initial context for thread 2 */
    getcontext(&ctx[2]);
    /* set up thread 2's stack */
    ctx[2].uc_stack.ss_sp   = stack2;
    ctx[2].uc_stack.ss_size = STACKSIZE;
    /* set up thread 2's link.  when thread 2 exits, automatically
     * context switch to thread 1 */
    ctx[2].uc_link          = &ctx[1];
    /* set up thread entry point and function arguments for thread
     * 2.  pass 2 args (int values 2 and 42) to the function */
    makecontext(&ctx[2], mythread, 2, 2, 42);


    /* do a context switch.  switch from thread 0 (main thread)
     * to thread 2 */
    swapcontext(&ctx[0], &ctx[2]);

    /*
     * with the swapcontext() call directly above, we should first
     * start in thread 2, then swap to thread 1 (because of swapcontext()
     * call in mythread()), then swap back to thread 2 (again, because of
     * swapcontext() call in mythread()).  thread 2 should then finish.
     * because of its context link pointer, we'll switch to thread 1.
     * thread 1 should then finish.  because of thread 1's context link,
     * we should switch back to our main thread, and come back here.
     */
    printf ("threads are all done.  i'm back in main!\n");

    /* free stacks and be done. */
    free (stack1);
    free (stack2);
    return 0;
}
```

When you run this, you will get the following:

```
prompt> ./ctxtest
in thread 2; x is 42 and shared_variable is 1
in thread 1; x is 13 and shared_variable is 2
back in thread 2; shared_variable is 3
back in thread 1; shared_variable is 4
threads are all done.  i'm back in main!
prompt>
```

Read the man pages of `getcontext()`, `makecontext()`, and `swapcontext()` to learn more. **Note:** the code above is posted on Moodle, and there are more example programs on the man pages. Before diving into designing

your library, understand what's going on in these programs.

2.0.2 Scheduling Order

This section describes the specific scheduling order that your thread library should follow. Remember that a correct concurrent program must work for all thread interleavings.

All scheduling queues should be FIFO. This includes the ready queue and the queue of threads waiting on a mutex or semaphore or condition variable (these synchronization items only apply to stages 2 and 3).

When a thread calls `ta_create`, the caller does not yield the CPU. The newly created thread is put on the ready queue but is not executed right away.

When the main thread calls `ta_waitall`, it should give up the CPU and the next runnable thread should run. If there are no threads remaining, it should return 0. If there are no runnable threads, it should return -1.

Since you library is implemented in user space, it is NOT preemptive. In order for your library to perform a context switch in stage 1, user threads must either (1) call `ta_yield()` or (2) finish and return from the thread entry function.

2.0.3 Deleting a Thread and Exiting the Program

A thread finishes when it returns from the function that was specified in `ta_create`. Remember to de-allocate the memory used for the thread's stack and context (do this AFTER the thread is really done using it).

2.1 Stage 2 functionality (10 out of 50 points)

For stage 2, you will add semaphores and mutexes to your thread library. You will need to create two types for programs using your library: `talock_t` and `tasem_t`. The definitions for these types will have to be in the file `threadsalive.h`. The functions that need to be added to your library, and their descriptions, are:

Signature	Usage
`void ta_sem_init(tasem_t *sema, int value)`	Create a semaphore and allocate any necessary memory. The semaphore should be initialized to the value given.
`void ta_sem_destroy(tasem_t *sema)`	Destroy a semaphore and release any associated resources.
`void ta_sem_post(tasem_t *sema)`	Atomically increase the semaphore value by one. If any threads are waiting on the semaphore and the semaphore value is greater than zero, the current thread should not yield the processor.
`void ta_sem_wait(tasem_t *sema)`	Wait for a semaphore value to be greater than zero. Atomically decrement the semaphore by one if it is currently greater than zero. If the semaphore is zero at the time of a call to `ta_sem_wait()` the calling thread should block and yield the processor to another ready thread.
`void ta_lock_init(talock_t *lock)`	Called to initialize a lock. The function should allocate any memory necessary for the lock.
`void ta_lock_destroy(talock_t *lock)`	Called to destroy a lock and any resources used.
`void ta_lock(talock_t *lock)`	Lock it. If other threads are waiting on the lock, the current thread should wait until the lock becomes available, allowing another ready thread to run.
`void ta_unlock(talock_t *lock)`	Unlock it. `ta_unlock()` should not cause any context switches.

Note: it is perfectly acceptable implement the lock functions using the semaphore functions. Indeed, I would advise doing that. (If you aren't sure what that means, you should review semaphores in the textbook.)

2.1.1 Ensuring Atomicity

To ensure atomicity of multiple operations, users will use locks that you provide. To implement those locks, you COULD use the "compare and swap" routine provided by the x86 processor family.

Below is a code snippet that allows you to call the compare and swap assembly routine from C (also posted on Moodle).

```
/*
 * check value pointed to by 'ptr'; if it is still equal to the
 * value passed in by 'old', then atomically swap in the 'new' value
 */
int cas(int *ptr, int old, int new)
{
    unsigned char ret;
```

```
    // NB: sete sets a 'byte' not the word

    __asm__ __volatile__ (
        "  lock\n"
        "  cmpxchgl %2,%1\n"
        "  sete %0\n"
        : "=q" (ret), "=m" (*ptr)
        : "r" (new), "m" (*ptr), "a" (old)
        : "memory");

    return ret;
}
```

Note again that your thread library is not preemptive. You have complete control over when a thread runs and hence a thread will not be interrupted and context-switched arbitrarily to another thread. Because of this, you do NOT need to use anything fancy like compare-and-swap to implement locks. Think about why this is and build your locks and condition variables accordingly. If you want to get fancy anyway, have fun.

2.2 Stage 3 functionality (worth bragging rights only!)

Stage 3 adds condition variables to your thread library. A new condition variable type needs to be added: `tacond_t`. Your library will also have to augmented to include the following new functions:

Signature	Usage
`void ta_cond_init(tacond_t *cond)`	Create an initialize a condition variable, including allocating any necessary memory.
`void ta_cond_destroy(tacond_t *cond)`	Destroy a condition variable, releasing any associated resources.
`void ta_wait(talock_t *lock, tacond_t *cond)`	Wait on a condition variable until another thread calls signal(). A call to `ta_wait()` should also release the mutex lock. (Review the discussion on condition variables in the textbook and in the notes.)
`void ta_signal(tacond_t *cond)`	Wake one thread waiting on the condition variable. When a thread calls `ta_signal`, the caller should not immediately yield the CPU. Instead, a woken thread should be put on the ready queue to be run in the future. A woken thread should request the lock when it next runs.

3 Tips and Hints

Follow the stages. Start by implementing `ta_libinit`, `ta_create`, and `ta_yield`. Then get `ta_waitall` working. Test this stuff thoroughly. Write a few test programs that exercise your library. Follow a similar approach with stage two; write some tests and then get the basics in your library working. Save the condition variables for last, or avoid them altogether. Yes, they really are optional.

Use assertion statements copiously in your thread library to check for unexpected conditions. These error checks are essential in debugging concurrent programs, because they help flag error conditions early. Read the man page for assert() for more information.

Declare all internal variables and functions (those that are not called by clients of the library) ”static” to prevent naming conflicts with programs that link with your thread library. For example:

```
static void ta_libinit_helper(void)
{
    // a helper function that user programs shouldn't touch
}
```

In C, this use of the keyword static means that only functions within the current file (e.g., `threadsalive.c`) can access the static functions. Similar things can be done with private library variables.

You'll probably want to reuse your linked list code from previous labs.

4 Submission

Make a tar archive of all your source code and submit that archive to Moodle. To create a tar archive, you can use something similar to the following command: `$ tar czvf lab07.tar.gz *.c *.h SConstruct`.