Apr 26, 2016

Nick Pfister
Mariam Zaim

ECE 695 - Project Report
Kernel Threading in xv6

Problem:

Xv6 (ARM Cortex A9), a simple Unix-like teaching operating system, comes standard with no threading libraries included or supported. Thus the Xv6 kernel is a single threaded implementation. This kernel's single threaded nature does not lend itself well to current or planned versions of Xv6 that support symmetric multiprocessors. Symmetric multiprocessors naturally lend themselves to kernel speedup as they can physically run multiple threads at one time. The motivation for developing and adding a kernel level threading library to Xv6 is to leverage the advantages of a multithreaded kernel, Xv6 on symmetric multiprocessors, and enable multithreading and parallelism in other kernel level applications.

Challenges:

The kernel threading library implemented, kthreads for short, presented two main categories of challenges: a deep understanding of threading and an intimate knowledge of Xv6 kernel's inner workings. One of the largest challenges was understanding the difference between threads and processes in a kernel threading library. The differences are minute and must be very well understood. The main difference can be explained with result between kthread_create() and fork(). Kthread_create() and fork() both create processes inside Xv6, the minute difference is that fork() creates a process with a new pid and kthread_create() creates a process with the same pid and a new tid, thread ID. Our kernel threading library leverages the existing process structure within Xv6. This allows threads within one pid to interact with the rest of the Xv6 kernel without any changes to the rest of the kernel. For example, the Xv6 scheduler was not altered at all to handle the new kernel threading library. With this minute difference defined the following challenges are solvable.

The next pressing challenge was to distinguish processes from threads, as both use the same proc structure and live in the same process table. The process struct in Xv6 had to be altered to also have a tid, thread TID, field. This field had to be initialized to 0 on process creation. Another large challenge that arose while implementing kthread_cond_wait() and kthread_join(), was how to properly put a process or thread to sleep and wake it back up again. This required a deep understanding into the specific states a process can be in within the Xv6 scheduler. Along the same lines, this tinkering led straight to having to debug the kernel threading library from leaving zombies behind. Thread orchestration on the programmer's part, while coding test cases, was another challenge. The program utilizing the kernel threading library must properly orchestrate and use the provided syscalls to ensure that there are no abandoned threads leading to zombies. This challenge arose and was solved in the test cases for both kthread_create,join,exit and the test cases for testing kthread_cond_wait,signal.

Solution:

The main focus of this project was to implement a threading library for kernel level threads in xv6. Our implementation very closely mimicked the creation and implementation of processes in xv6, with the main difference being that threads spawned from a single process all shared the same address space, whereas child processes forked off a single process do not. Our threading library allows applications to create and destroy kernel threads in order to support multithreading and exploit parallelism in tasks. It also provides a basic synchronization primitive in the form of condition variables. The aforementioned functionalities were implemented using the five basic syscalls explained below.

- **Kthread_create (void* (*foo)())** : In this syscall, a new thread structure is allocated, and set to be the child of the calling thread. Stack space is allocated for this new thread, and the page table was made to point to the parent's page table, so that both would share the same address space. This syscall takes in a pointer to a function as the first argument, so the new thread's PC is changed to point to the start of this function, and its state is set to runnable to allow execution to begin.

- **Kthread_exit (void)**: This syscall implementation is used when an application would like to terminate the execution of a thread. The caller wakes up its parent, and then terminates by calling the scheduler and becomes a zombie thread. This function does not return to the caller. At this point, the parent must call kthread_join in order to clean up the remnants of its child threads or wait on them.

- **Kthread_join (int tid)**: This syscall allows the caller to wait until a specific thread terminates. It takes in a thread ID as the first argument. After the thread terminates, all traces of this thread are removed by freeing any structures (e.g stack) allocated to it, and the thread is removed from the process table. This syscall also cleans up any zombie threads that might have terminated themselves by calling kthread_exit.

- **Kthread_cond_wait (int wait_channel)**: With this synchronization primitive, a thread may put itself to sleep until the specified condition variable that it is waiting for has been set by a call to signal(). A mutex lock must be associated with each condition variable, and that burden is not left to the application program, but instead, the internal implementation of kthread_cond_wait handles the acquiring and releasing of the lock. If a thread calls this wait, the programmer must make sure some later thread calls signal to wake waiting thread(s), otherwise they wait forever.

- **Kthread_cond_signal (int wait_channel)**: This syscall is used to wake up any number of threads that are waiting on the corresponding condition variable. The programmer must make sure that the correct conditional variable is passed in order to wake up all waiting threads. One call to signal() will wake all waiters.

Testing:

In order to fully test our kernel threading library, we built a couple different test applications that used all five of our syscalls in various ways. In the most basic test program, we compared the fork and the kthread_create syscalls, and verified the behavior of threads and processes in terms of sharing address space. We observed changes in a global counter variable, and were able to see that changes to this counter by one thread were reflected also in other thread's address spaces. We also used kthread_exit and kthread_join in this test program to ensure that no zombie threads were left behind.

A separate script was used to test the condition variables' wait and signal functions, that was similar to a producer-consumer scenario. However, there were multiple consumers and a single producer in this case: one thread wrote to a global variable, and all other threads waited for this value to be written, i.e. until they were signalled. We ran multiple iterations of this process and observed thread behavior in corner cases, e.g. when there was no value to consume.