CS 537 - Introduction to Operating Systems - Fall 2017

Site /

Project4b

Project 4b: xv6 Kernel Threads

Objectives

There are two objectives to this assignment:

- To build a small thread library in xv6 using two new system calls: `clone()` and `join()`.
- To add some concurrency to your library using spin locks and condition variables.

Updates

The following line has been removed from the specifications: "`fork()` should be updated such that if a multi-threaded process has one thread call `fork()`, the new process should only have one thread - the thread that called `fork()` (it is fine that the stacks allocated on the heap transfer over to the new process)." - we will not be testing your code by having threads fork a new process.

More tests have been released as categorized below.

Overview

In this project, you'll be adding real kernel threads to xv6. Sound like fun? Well, it should. Because you are on your way to becoming a real kernel hacker. And what could be more fun than that?

Specifically, you'll do three things. First, you'll define a new system call to create a kernel thread, called `clone()`, as well as one to wait for a thread called `join()` . Second, you'll use `clone()` and `join()` to build a little thread library, with `thread_create()` / `thread_join()` .Third, you'll create some synchronization mechanisms by adding a simple spinlock with `lock_acquire()` / `lock_release()`, and a condition variable with `cond_wait()` / `cond_signal()` (with 3 new system calls for the condition variables). That's it! And now, for some details.

Note: Start with a clean kernel; no need for your new fancy address space with the stack at the top, for example.

Details

Your new clone system call should look like this: `int clone(void(*fcn)(void*), void *arg, void *stack)`. This call creates a new kernel thread which shares the calling process's address space. File descriptors are copied as in fork. The new process uses `stack` as its user stack, which is passed the given argument arg and uses a fake return PC (`0xffffffff`). You can exit a thread by adding a handler for this exact address to kill the process. The stack should be one page in size. The new thread starts executing at the address specified by `fcn`. Looks familiar? Yes! It's very similar to `exec()`! As with `fork()`, the PID of the new thread is returned to the parent.

There are a few assumptions upon thread creation:

- The parameter `arg` in `clone()` can be a NULL pointer, and `clone()` should not fail.
- The thread can clone another thread but will not fork another process.
- When thread A clones another thread B, the parent of thread B should be A, NOT the parent of A. Note that parent of thread A, thread A and thread B should share the same address space.
- You can assume that a process will have at most 8 threads within it (NTHREADS = 8).

Another new system call is `int join(void **stack)`. This call waits for a child thread that shares the address space with the calling process. It returns the PID of waited-for child or -1 if none. The location of the child's user stack is copied into the argument `stack`.

For example, process A clones thread B and C, now A calls join(); A waits for whichever thread finishes first.

Note that if B clones another thread D, then A will never wait for D as D's parent is B.

You also need to think about the semantics of existing system calls. For example, `wait()` should wait for a child process that does not share the address space with this process. It should also free the address space if this is the last reference to it.

Your thread library will be built on top of this, and just have a simple `thread_create(void (*start_routine)(void*), void *arg)` routine. This routine should call `malloc()` to create a new user stack and use `clone()` to create the child thread and get it running. A

`thread_join()` call should also be created, which calls the underlying `join()` system call, frees the user stack, and then returns.

Next, your thread library should have the following synchronization mechanisms

- A simple spin lock: There should be a type `lock_t` that one uses to declare a lock, and two routines `lock_acquire(lock_t *)` and `lock_release(lock_t *)` , which acquire and release the lock. The spin lock should use x86 atomic exchange to built the spin lock (see the file spinlock.c inside the kernel folder for an example of something close to what you need to do). Another routine, `lock_init(lock_t *)` , is used to initialize the lock as need be. *This lock should be implemented in user mode.*
- Condition variable: there should be a type `cond_t` that one uses to declare a condition variable, and two routines `cond_wait(cond_t *, lock_t *)` and `cond_signal(cond_t *)` , which wait and signal the CV. You may want to look at what `sleep()` and `wakeup()` do in kernel/proc.c for some hints. One last routine `cond_init(cond_t *)` , is used to initialize the CV as need be. **NOTE:** You will need to add 3 new system calls to implement these respective functions, but we will not test them directly so you can name them however you like. *The condition variable should be implemented in kernel mode.*

Your condition variable should operate in the following fashion:

cond_t

- The condition variable should have a queue of size NTHREADS to hold waiting threads (HINT: you can keep a head and tail to form a circular queue).
- The queue should have a lock protecting it (using the same atomic exchange logic).

cond_wait()

- cond_wait should panic (or throw some error) if the calling thread had not acquired the lock or if the queue is full.
- A thread calling cond_wait should acquire the queue's lock, add itself to the queue, put itself to sleep, and release both the calling lock and the queue lock. Once it has been woken up, it should spin until it can reacquire the calling lock and then proceed.

cond_signal()

- A thread calling cond_signal should acquire the queue lock, wake up the thread at the head of the queue, and release the queue lock.

Here is a helpful resource for simple locks and condition variables (ignoring the producer/consumer logic).

To test your code, use the TAs tests, as usual! But of course you should write your own little code snippets to test pieces as you go.

One thing you need to be careful with is when an address space is grown by a thread in a multi-threaded process. Trace this code path carefully and see where a new lock is needed and what else needs to be updated to grow an address space in a multi-threaded process correctly.

One last HINT: To add a user thread library, you'll need to create it in the *user* folder and modify the makefile so that xv6 can provide it to the user space.

Have fun!

The Code

The source code for xv6 can be found in **~cs537-1/ta/xv6/** . Everything you need to build and run and even debug the kernel is in there, as before. We recommend starting with a fresh version for this project!

You may also find the following readings about xv6 useful: xv6 book

You may also find this book useful: Programming from the Ground Up. Particular attention should be paid to the first few chapters, including the calling convention (i.e., what's on the stack when you make a function call, and how it all is manipulated).

Test Along The Way

It is a good habit to get basic functionalities working before moving to advanced features. You can run an individual test with the following command:

shell% ~cs537-1/ta/tests/4b/runtests testname

We will be releasing tests incrementally, so you should not rely on the current span of tests to guarantee a working implementation (you might want to try adding your own tests).

1. Testing thread_create call: create, create2, create3
2. Testing thread_join call: join, join2, join3

3. Basic threading workloads with create + join: recursion, clone_clone, two_threads, recursion2, fork_clone, many_threads
4. Testing locks: locks, recursion3, size
5. Testing CV: cond, cond2, cond3, cond4

Page last modified on November 19, 2017, at 07:44 PM