

Project 3: xv6 Threads

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()`. Then, you'll use `clone()` to build a little thread library, with a `thread_create()` call and `lock_acquire()` and `lock_release()` functions. Finally, you'll show these things work by using the TA's tests. That's it! And now, for some details.

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`). The stack should be one page in size and page-aligned. The new thread starts executing at the address specified by `fcn`. As with `fork()`, the PID of the new thread is returned to the parent.

The other new system call is `int join(int pid)`. This call joins with the child thread specified by the `pid` if it is a child of the calling process. It returns 0 for success or -1 for failure.

You also need to think about the semantics of a couple of existing system calls. For example, `int 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 last reference to it. Finally, `exit()` should work as before but for both processes and threads; little change is required here.

Your thread library will be built on top of this, and just have a simple struct `kthread` `thread_create(void (*start_routine)(void*), void *arg)` routine. This routine should call `malloc()` to create a new user stack, use `clone()` to create the child thread and get it running. A `thread_join(struct kthread)` call should also be used, which calls the underlying `join()` system call, frees the user stack, and then returns.

Your thread library should also have 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 build the spin lock (see the xv6 kernel for an example of something close to what you need to do). One last routine, `lock_init(lock_t *)`, is used to initialize the lock as need be.

To test your code, the TA will provide some test code. Do these tests miss anything? You should test and be sure.

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.

Submission...

Please submit to canvas with all files when you are completed. Please include a readme with the name readme-[username] with documentation about your changes.

As usual, it might be good to read the xv6 book a bit.

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).