Lab 3: XV6 Threads

Overview

In this project, you'll be adding real kernel threads to xv6.

Specifically, you'll do three things. First, you'll define a new system call to create a kernel thread, called `clone()`. 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 writing a test program in which multiple threads are created by the parent, and each insert items into a thread-safe linked list that you will write. That's it! And now, for some details.

Details

Your new syscall should look like this: `int clone(void *stack, int size)`. It does more or less what `fork()` does, except for some major difference:

- Address space: instead of making a new address space, it should use the parent's address space (which is thus shared between parent and child).
- File descriptors should not be duplicates of the the file descriptors of the parent -- they should use the same file descriptor.
- You might also notice a single pointer is passed to the call, and size; this is the location of the child's user stack, which must be allocated before the call to clone is made. Thus, inside `clone()`, you should make sure that when you return, you are running on this stack, instead of the stack of the parent.

Similar to `fork()`, the `clone()` call returns the PID of the child to the parent, and 0 to the newly-created child thread.

There are also some differences in the `exit` call and `wait` calls. For `exit`, we typically close all the file descriptors that a process uses. However, since these are now shared across all threads, this is not a good idea. Similarly, a parent process uses `wait` to reap a child process (free up the rest of its resources such as the stack, and memory). These are also tricky because these are now shared among all the kernel threads of the same process. A good solution to this problem is to keep track of the number of threads that share an address space and only free resources when the last one exits (for file descriptors) or is reaped (for the memory). Note that the last thread to exit may not be the parent.

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 use `clone()` to create the child, and then call `start_routine()` with the argument `arg`. You must prepare the arguments on the thread stack (for this you need to understand the C calling conventions; to help you, there is a book that describes the calling convention (programming from the ground up)). This is likely to be the most challenging component of the assignment.

Your thread library should also implement 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, you should build a simple program that uses `thread_create()` to create some number of threads. The threads will simulate a game of frisbee where each thread passes the frisbee (token) to the next. The location of the frisbee is updated in a critical region protected by a lock. Each thread spins to check the value of the lock. If it is its turn, then it prints a message, and releases the lock.

The frisbee program for testing the threads: a simulation of Frisbee game, the user specifies the number of threads (players), passes in the game. The parent process creates multiples threads (players), each thread accesses the lock, check if it is its turn to throw the frisbee (token), if so, it assigns the id of the next thread to receive the frisbee, increases the number of passes and releases the lock.

Example

```
Input
$ frisbee 4 6

Output:
Pass number no: 1, Thread 0 is passing the token to thread 1
Pass number no: 2, Thread 1 is passing the token to thread 2
Pass number no: 3, Thread 2 is passing the token to thread 3
Pass number no: 4, Thread 3 is passing the token to thread 0
Pass number no: 5, Thread 0 is passing the token to thread 1
Pass number no: 6, Thread 1 is passing the token to thread 2

Simulation of Frisbee game has finished, 6 rounds were played in
total!
```

The Code

Download a fresh copy of xv6 and add the above-mentioned functionality in that. Try to use the thread library as much as possible.

Chapters 2 and 4 of the xv6 books are essential background for this project.

You may also find this book useful: Programming from the Ground Up. 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).

Grades breakdown

- clone() system call: 25%
- thread_create(...) function: 25%
- Lock initialization, acquire & release functions for spin lock: 25%
- Frisbee test program: 25%

Total: 100%

Submission Procedure

Your submission should include, in addition to your code, a writeup that describes the changes that you have done and how you tested your kernel.

The grading of this project will be done by demo in person. You are expected to understand all the implementation and can answer questions about it. You should be expected to be asked questions about your code (and related code) and be able to demo the example and any tests.

The write-up should be short (a few pages at most) and concise. You should also describe how you tested your code, to convince us (and yourselves) that it works!

Note: this is a significantly more challenging assignment than lab 1 and 2. Please start reading given material as early and possible. Have fun!