CS 202: Advanced Operating Systems
University of California, Riverside

Lab #3: xv6 Threads

Due: 12/03/2021, Friday, 11:59 p.m. (Pacific time)

Overview

In this project, you will be adding kernel-level thread support to xv6.

Specifically, you'll do three things. First, you will define a new system call to create a kernel thread, called `clone()`. Then, using the `clone()` system call, you will build a little user-level thread library consisting of `thread_create()`, `lock_acquire()` and `lock_release()` functions. Finally, you will show these things work by writing a user-level test program in which multiple threads are created by the parent, and each thread inserts 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 the following major differences:

- 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 file descriptors of the parent -- they should share the file descriptors.
- 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 by the caller (parent) before the call to clone is made. Thus, inside `clone()`, you should make sure that, when this syscall is returned, a child thread runs on this stack, instead of the stack of the parent. Some basic sanity check is required for input parameters of `clone()`, e.g., `stack` is not null.

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 modifications required for the **`exit()`** and **`wait()`** syscalls. For `exit()`, by default, it closes 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 shared resources when the last one exits or is reaped by wait (like a reference counter). 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 **`int 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 user thread stack. For this, you need to understand the calling conventions of the compiler we use (riscv64-gcc). Do not blindly trust web documents about the RISC-V calling conventions since the calling conventions are determined by the compiler and may change depending on compile options. The best way is to recall the basic C calling conventions and how arguments are passed on the stack (reference: Programming from the ground up). Note that `arg` is a pointer which is 8 bytes in riscv64.

Your thread library should also implement **simple user-level spin lock** routines. 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 **RISC-V 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) and the number of 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 functionalities.

Chapters 2 and 4 of the xv6 book 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 pts
- thread_create(...) function: 25 pts
- Lock initialization, acquire & release functions for spin lock: 25 pts
- Frisbee test program: 25 pts

Total: 100 pts

Submission Procedure:

Your submission should include:

(1) The **entire XV6 source code** with your modifications ('make clean' to reduce the size before submission)
(2) A **writeup (in PDF)** that describes the changes that you have done and how you tested your kernel.
(3) A **demo video** A demo video showing that all the functionalities you implemented can work as expected. Demonstrate the frisbee example.

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