Project 4: Support for concurrency in xv6

Deadline: April 27 11:59 PM

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()` , `lock_acquire()` , `lock_release()` , and `cv_signal()` and `cv_wait()` functions. Finally, you'll show these things work by using the TA's tests. 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 bottom, for example.

Details

Thread Handling System Calls

clone is supposed to create a new thread similar to fork() creating a new process. The clone system call should follow the below signature:

```
int clone(void(*fcn)(void*), void *arg, void *stack)
```

This call should create a new thread sharing the calling process's address space. File descriptors are to be copied as in fork. The newly created thread 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`. Failure of -1 is returned if anything goes wrong during the execution of the system call else pid is returned to the caller of clone (similar to fork).

All threads created via clone should point to the process as it's parent. Threads created by another thread should also follow the same behavior (The primary process should be the parent).

join waits for a thread within the process to complete. It follows the below format:

```
int join(int pid)
```

This call waits for a child thread with pid (as returned by the clone) that shares the address space with the calling process. Note that a wait on the primary thread (the process thread) is invalid and also, join on a thread belonging to another process is also invalid. If the pid value is -1, wait for any of the child thread in the process. The system call returns the PID of waited-for child or -1 if none.

wait/exit 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. Waiting on a thread is not be allowed. It should also free the address space if this is last reference to it.

Finally, `exit()` needs a little modification to work with processes and threads. When the main process exits, it should kill all its children threads and wait for them to exit (similar to join) before it returns. However, child thread exiting does not affect other threads running in the process. It exits itself and returns.

User Mode Library

Your **library** should support three set of interfaces. The user library implementation should be included in one or many c files under user directory, the function declarations included in the file user/user.h and any structure definitions to the header include/types.h.

First, the thread management interfaces should be built on top of the above system calls, and support the following interfaces to be used by the user mode application.

- `int thread_create(void (*start_routine)(void*), void *arg)` routine should call `malloc()` to create a new user stack, use `clone()` to create the child thread and get it running. The stack as mentioned above should be page aligned. The function should return the pid of the newly created thread on success and -1 on failure.
  NOTE: Invoke exit() at the end of start_routine function in your test program before the thread finishes.
- `int thread_join(int pid)` should calls the underlying `join()` system call, and frees the user stack. The function should return a pid value (not necessarily same as input pid since input pid could also be -1 to wait for any child thread) on success or -1 on failure.

Second, **Locks** should also be supported in your library. There should be a type `lock_t` that one uses to declare a lock, and three routines `void lock_acquire(lock_t *)` and `void lock_release(lock_t *)`, to acquire and release the lock. You are supposed to implement a **ticket lock.** The lock primitive should use x86 atomic add (fetch and add) to build the spin lock (see the xv6 kernel for an example of something close to what you need to do). One last routine, `void lock_init(lock_t *)`, is used to initialize the lock as need be.

Finally, **Condition Variables** to be supported with the related routines:

`cond_t` and `void cv_wait(cond_t *, lock_t *)` and `void cv_signal(cond_t *)` . These routines should do what is expected: either put the caller to sleep (and release the lock) or wake a sleeping thread, respectively. You will have to add additional system calls to achieve this.

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.

Have fun!

Submission

Follow the submission guidelines that you did for the projects. Though there are no subprojects for P4, copy your source code into the xv6 folder under p4.

The Code

The code (and associated README) can be found in **~cs537-2/public/html/xv6.tar.gz** . Everything you need to build and run and even debug the kernel is in there, as before.

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).