Kernel Threads

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. That's it! And now, for some details.

Overview

Your new clone system call should look like this: `int clone(void(*fcn)(void *, void *), void *arg1, void *arg2, 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 two arguments (`arg1` and `arg2`) and uses a fake return PC (`0xffffffff`); a proper thread will simply call `exit()` when it is done (and not `return`). 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 (for simplicity, threads each have their own process ID).

The other new system call is `int join(void **stack)`. This call waits for a child thread that shares the address space with the calling process to exit. 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` (which can then be freed).

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. Also, `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 `int thread_create(void (*start_routine)(void *, void *), void *arg1, void *arg2)` routine. This routine should call `malloc()` to create a new user stack, use `clone()` to create the child thread and get it running. It returns the newly created PID to the parent and 0 to the child (if successful), -1 otherwise. An `int thread_join()` call should also be created, which calls the underlying `join()` system call, frees the user stack, and then returns. It returns the waited-for PID (when successful), -1 otherwise.

Your thread library should also have a simple *ticket lock* (read this book chapter for more information on this). There should be a type `lock_t` that one uses to declare a lock, and two routines `void lock_acquire(lock_t *)` and `void lock_release(lock_t *)`, which acquire and release the lock. The spin lock should use x86 atomic add to build the lock -- see this wikipedia page for a way to create an atomic fetch-and-add routine using the x86 `xaddl` instruction. One last routine, `void lock_init(lock_t *)`, is used to initialize the lock as need be (it should only be called by one thread).

The thread library should be available as part of every program that runs in xv6. Thus, you should add prototypes to `user/user.h` and the actual code to implement the library routines in `user/ulib.c`.

One thing you need to be careful with is when an address space is grown by a thread in a multi-threaded process (for example, when `malloc()` is called, it may call `sbrk` to grow the address space of the 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.

Building `clone()` from `fork()`

To implement `clone()`, you should study (and mostly copy) the `fork()` system call. The `fork()` system call will serve as a template for `clone()`, with some modifications. For example, in `kernel/proc.c`, we see the beginning of the `fork()` implementation:

```
int
fork(void)
{
  int i, pid;
  struct proc *np;

  // Allocate process.
  if((np = allocproc()) == 0)
    return -1;

  // Copy process state from p.
  if((np->pgdir = copyuvm(proc->pgdir, proc->sz)) == 0){
    kfree(np->kstack);
    np->kstack = 0;
    np->state = UNUSED;
    return -1;
  }
  np->sz = proc->sz;
  np->parent = proc;
  *np->tf = *proc->tf;
```

This code does some work you need to have done for `clone()`, for example, calling `allocproc()` to allocate a slot in the process table, creating a kernel stack for the new thread, etc.

However, as you can see, the next thing `fork()` does is copy the address space and point the page directory (`np->pgdir`) to a new page table for that address space. When creating a thread (as `clone()` does), you'll want the new child thread to be in the *same* address space as the parent; thus, there is no need to create a copy of the address space, and the new thread's `np->pgdir` should be the same as the parent's -- they now share the address space, and thus have the same page table.

Once that part is complete, there is a little more effort you'll have to apply inside `clone()` to make it work. Specifically, you'll have to set up the kernel stack so that when `clone()` returns in the child (i.e., in the newly created thread), it runs on the user stack passed into clone (`stack`), that the function `fcn` is the starting point of the child thread, and that the arguments `arg1` and `arg2` are available to that function. This will be a little work on your part to figure out; have fun!

x86 Calling Convention

One other thing you'll have to understand to make this all work is the x86 calling convention, and exactly how the stack works when calling a function. This is you can read about in Programming From The Ground Up, a free online book. Specifically, you should understand Chapter 4 (and maybe Chapter 3) and the details of call/return. All of this will be useful in getting `clone()` above to set things up properly on the user stack of the child thread.