Project 4b: 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 build a little thread library, with simple spin locks, condition variables and semaphores. Finally, you'll show these things work by using the TA's tests. That's it! And now, for some details.

Note: Because the VM layout is based on the last project, so to help you start with, we've provided a new xv6 kernel with the new layout, i.e. 2 unmapped pages starting at `0x0` of the address space and one page of stack ending at `USERTOP`. You are also welcome to start from the original kernel or your own implementation from last project.

Details

In your last project, you have moved up the stack to `USERTOP`, which grows downwards along the way. Now for this project, the stack will NOT grow and is of a constant size of one page. The stack size of a new thread is also one page, page-aligned and will NOT grow. It will be placed somewhere below the stack of the parent process.

Here is a diagram to help you better understand the new VM layout designed for threads.

```
+--------------------------------+  <--- USERTOP
|      Stack of the Process      |  1 PAGE
+--------------------------------+
|        Guard Page (FREE)       |  1 PAGE
+--------------------------------+
|        Stack of Thread 1       |  1 PAGE
+--------------------------------+
|        Guard Page (FREE)       |  1 PAGE
+--------------------------------+
|                                |
:                                :
:              ^                 :
|              |                 |
+--------------------------------+  <--- proc->sz
|             Heap               |
+--------------------------------+
|          Code + Data           |  1 PAGE
+--------------------------------+
|        Guard Page (FREE)       |  2 PAGES
+--------------------------------+
```

The reason we need a guard page for the thread stack is because we want to prevent stack overflow. If the threads' stacks are adjacent, then one thread may misbehave, e.g. heavy recursion, and thus access the stack of the neighboring thread. However, this design cannot resolve the scenario where the thread "randomly" access address that is mapped (to physical memory) but only valid to some other thread.

Your new clone system call should look like this: `int clone(void(*fcn)(void*), void *arg)` . This call creates a new kernel thread which shares the calling process's address space. File descriptors are copied as in fork.

The given argument `arg` will get passed to the thread stack. The stack uses a fake return PC (`0xffffffff`). The new thread starts executing at the address specified by `fcn`. Looks familiar? Yes! It's very similar to `exec()`! Similar to fork(), the PID of the new thread is returned to the parent.

There are several assumptions upon thread creations:

- Each process can create and run at most 8 threads concurrently. The routine `clone()` should fail, i.e. return -1 if there are already 8 threads running concurrently.
- ~~The stack of thread A should be inaccessible by other threads that share the same address space - this is essentially the definition of threads, though.~~
- *In your implementation, thread A may directly dereference the address in thread B's stack. However, if this address gets passed to thread A as an argument from user space, you should reject it, i.e. the associated function call should fail.*
- *The `arg` in `clone()` can be a `NULL` pointer, and `clone()` should not fail.*
- The thread can clone another thread but will not fork another process.
- When thread A clones another thread B, the parent of thread B should be A, NOT the parent of A. Note that parent of thread A, thread A and thread B should share the same address space.
- *The heap can grow at most to 18 pages below the `USERTOP`, as the top 18 pages are reserved for the stack of the process and 8 threads.*

Another new system call is `int join(void)` . This call **waits** for a child thread that shares the address space with the calling process. It returns the PID of waited-for child or -1 if none.

For example, process A clones thread B and C, now A calls `join()`, A waits for whichever thread finishes first.

Note that if B clones another thread D, then A will never wait for D as D's parent is B.

When the thread finishes, this routine should also reset the stack that this thread used such that later thread can reuse this stack.

You also need to think about the semantics of a couple of existing system calls. For example, `fork()` will not copy the stacks of the existing threads. `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.

Next, you should create a user thread library that has

- 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 as the hardware support (see the xv6 kernel for an example of something close to what you need to do). Another routine, `lock_init(lock_t *)` , is used to initialize the lock as need be.
- Condition variable: there should be a type `cond_t` that one uses to declare a condition variable, and two routines `cond_wait(cond_t *, lock_t *)` and `cond_signal(cond_t *)`, which wait and signal the CV. You may want to look at what `sleep()` and `wakeup()` do in `kernel/proc.c` for some hints. Another routine `cond_init(cond_t *)`, is used to initialize the CV as need be.
- Semaphore: there should be a type `sem_t` that one uses to declare a semaphore, and two routines

`sem_wait(sem_t *)` and `sem_post(sem_t *)`, which wait and post the semaphore. You should build the semaphore based on the lock and condition variable you implemented. Another routine `sem_init(sem_t *, int)`, is used to initialize the semaphore as need be.

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!

The Code

The code with the new VM layout (and associated README) can be found in **~cs537-1/ta /xv6_rearranged/** . Everything you need to build and run and even debug the kernel is in there, as before.

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

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).

Test Along The Way

It is a good habit to get basic functionalities working before moving to advanced features. You can run an individual test with the following command:

```
shell% ~cs537-1/ta/tests/4b/runtests testname
```

1. Testing clone system call: `clone, clone2, clone3`
2. Testing join system call: `join, join2, join3`
3. Basic threading workloads with clone + join: `recursion, recursion2, fork_clone, many_threads`
4. Threads should not access invalid addresses: `badaddr, badaddr2, guard`
5. Testing locks: `locks, recursion3, size`
6. Testing CV: `cond, cond2, cond3, cond4`
7. Testing semaphores: `sema, producer_consumer_sem`

Note: there will be 20% hidden tests for this part :)