Project #3B - xv6 Kernel Threads

1. 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()

2. 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 with two arguments (arg1 and arg2) and uses a fake return PC (0xffffffff); a proper thread will simply call exit() 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 the 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 routine.

```
int thread_create(void (*start_routine)(void *, void *), void
*arg1, void *arg2)
```

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 *)
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.

3. 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!

4. 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. You can read about this in Programming From The GroundUp, 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.

5. Running Tests

- Use the following script file for running the tests:
```
prompt> ./test-threads.sh -c -v
```

- If you implemented things correctly, you should get some notification that the tests passed. If not …

- The tests assume that xv6 source code is found in the *src/* subdirectory. If it's not there, the script will complain.

- The test script does a one-time clean build of your xv6 source code using a newly generated makefile called *Makefile.test*. You can use this when debugging (assuming you ever make mistakes, that is), e.g.:

```
prompt> cd src/
prompt> make -f Makefile.test qemu-nox
```

- You can suppress the repeated building of xv6 in the tests with the '-s' flag. This should make repeated testing faster:
```
prompt> ./test-threads.sh -s
```

- You can specifically run a single test using the following command
```
./test-mmap.sh -c -t 7 -v
```
The specifically runs the test_7.c alone.

- The other usual testing flags are also available. See the testing appendix for more details.

- This project will have few hidden test cases apart from the provided test cases.

6. Adding test files inside xv6

- Inorder to run the test files inside xv6, manually copy the test files(*test_1.c, test_2.c* etc…) inside xv6/src directory.(preferably in a different name like *xv6test_1.c*, etc…)
- Make the necessary changes to the Makefile

```
UPROGS=\
    _cat\
    _echo\
    _forktest\
    _xv6test_1\
    _grep\
    _init\
    _kill\
    _ln\
    _ls\
    _mkdir\
    _rm\
    _sh\
    _stressfs\
    _usertests\
    _wc\
    _zombie\
```

```
EXTRA=\
    mkfs.c xv6test_1.c ulib.c user.h cat.c echo.c forktest.c
grep.c kill.c\
    ln.c ls.c mkdir.c rm.c stressfs.c usertests.c wc.c zombie.c\
    printf.c umalloc.c\
    README dot-bochsrc *.pl toc.* runoff runoff1 runoff.list\
    .gdbinit.tmpl gdbutil\
```

- Now compile the xv6 using the command "*make clean && make qemu-nox*".

```
prompt> make clean && make qemu-nox
```

- Once it has compiled successfully and you are inside xv6 prompt, you can run the test.

```
$
$ xv6test
```

- You can also add your own test cases to test your solution extensively.
- Once you are inside xv6 qemu prompt in your terminal, if you wish to shutdown xv6 and exit qemu use the following key combinations:
- press Ctrl-A, then release your keys, then press X. (Not Ctrl-X)

Appendix – Test options

The run-tests.sh script is called by various testers to do the work of testing. Each test is actually fairly simple: it is a comparison of standard output and standard error, as per the program specification.

In any given program specification directory, there exists a specific tests/ directory which holds the expected return code, standard output, and standard error in files called n.rc, n.out, and n.err (respectively) for each test n. The testing framework just starts at 1 and keeps incrementing tests until it can't find any more or encounters a failure. Thus, adding new tests is easy; just add the relevant files to the tests directory at the lowest available number.

The files needed to describe a test number n are:

- n.rc: The return code the program should return (usually 0 or 1)
- n.out: The standard output expected from the test
- n.err: The standard error expected from the test
- n.run: How to run the test (which arguments it needs, etc.)
- n.desc: A short text description of the test
- n.pre (optional): Code to run before the test, to set something up
- n.post (optional): Code to run after the test, to clean something up

There is also a single file called pre which gets run once at the beginning of testing; this is often used to do a more complex build of a code base, for example. To prevent repeated time-wasting pre-test activity, suppress this with the -s flag (as described below).

In most cases, a wrapper script is used to call ***run-tests.sh*** to do the necessary work.

The options for run-tests.sh include:

- -h (the help message)
- -v (verbose: print what each test is doing)
- -t n (run only test n)
- -c (continue even after a test fails)
- -d (run tests not from tests/ directory but from this directory instead)
- -s (suppress running the one-time set of commands in pre file)