Operating Systems Project #4 - xv6 Threads

SWE3004-42 Introduction to Operating Systems - Fall 2018

Due date: December 14 (Fri) 11:59pm

1 Goal

In this project, you'll be adding real kernel threads to xv6.

2 Details

Specifically, you'll do three things. First, you'll define a new system call to create a kernel thread, called `clone()`. Then, you'll use `clone()` to build a little thread library, with a `thread_create()` call and `lock_acquire()` and `lock_release()` functions. Finally, you'll show these things work by writing a test program in which multiple threads are created by the parent, and each insert items into a thread-safe linked list that you will write. That's it! And now, for some details.

3 Details

Your thread library will just have `thread_create(void *(*start_routine)(void*), int priority, void *arg, void *stack)` routine. This routine should do more or less what `fork()` does to create the child, except for one major difference: instead of making a new address pace, it should use the parent's address space (which is shared between parent and child), and then call `start_routine()` with the argument `arg`.

You should also implement `void thread_exit(void *retval);`, which terminates the calling thread and returns a value via retval that is available to another thread in the same process that calls `thread_join()`.

`int thread_join(int tid, void **retval)` waits for the thread specified by tid to terminate. If that thread has already terminated, then `thread_join()` returns immediately. `thread_join()` copies the exit status of the target thread into the location pointed to by `*retval`. The call stack of the terminated thread should be freed by the calling thread. On success, it returns 0. If there's no thread with input tid, return -1. There is no parent-child hierarchy among threads. Any thread can invoke `thread_join()` for another thread.

`int gettid(void)` function returns caller's thread ID (TID). If the process is a single threaded process, the thread ID is same as the process ID, which should be returned by `int getpid(void)`. In a multi-thread process, all threads have the same PID, but each one has a unique TID within a process.

The maximum number of threads per process is limited to 8 (including the main thread). Your implementation should support at least 32 processes.

Your thread library should also have 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 `xchgl` to built 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.

You may test your code building a simple program that uses `thread_create()` to create some number of threads; each thread should, in a loop for a fixed number of times, add an element to a shared linked list of integers. For example, if a thread is told to run for 10 iterations through a loop, it should add an element for 0, then 1, then 2, and so on to the list. If there are two threads running, they should each do this in parallel. The linked list should be thread safe, grabbing a lock each time an element is being added to the list, and letting it go when done. At the end, the main thread should call `wait()` (repeatedly, once per child) to wait for all the children to complete; at

this point, the main thread should add up all of the values of the elements on the list, and print out a final value.

The command line arguments for this program are thus:

```
prompt> main numberOfThreads loopCount
```

4 Template Code

You should copy the project template and test case codes from /home/swe3004/project4 directory.

5 Clarifications

- Please ignore the priority of threads. That is, threads will be scheduled by the round robin scheduling policy in xv6. I.e., the scheduling unit is thread, not process.
- Your scheduler can manage processes and threads using the same `struct proc`. The `pid` in `structu proc` will be process id if `isthread` is 0 and thread id if `isthread` is 1.
- If a child thread creates another child thread, the parent of both child threads will be the same main thread, i.e., the thread whose id is is 1.
- `thread_create()` function should look very similar to `fork()`.
- When you create a thread, you need to store its arguments in `trapframe->esp` and deduct -4 to adjust the stack top for the next usage.

6 How to Submit

To create a tarball file, please run `make tarball` command. The tarball file must be submitted to the instructor's account using the following command.

```
$ os_submit project4 your_tar_ball_file
```

For any questions, please post them in Piazza so that we can share your questions and answers with other students and TAs. Please feel free to raise any issues and post any questions. Also, if you can answer other students' questions, you are welcome to do so. You will get some credits for posting questions and answering other students' questions.

7 Late Submission Policy

Late submissions will be accepted but with 20% penalty per day. That is, if you submit 2 days late and test score is 80 points, your penalized score will be 48 points (80 x 60%).