CS 416 - Operating Systems Design

Homework 2
Kernel-Level Threads in xv6

April 5, 2016
Due by Tuesday, April 19th at 11:55 PM

1 Introduction

In this homework, you will kernel-level threads in the xv6 operating system. In addition, you will implement a set of synchronization primitives that allows programmers to write correct multithreaded programs.

2 Implementing Kernel Level Threads in xv6

By default, xv6 does not implement support for threading. For this part, you will add kernel-level thread functionality to xv6.

2.1 Requirements

You must add kernel-level thread functionality to xv6 by implementing some new system calls, as well as making several other supporting modifications.

2.1.1 Kernel

Implement two new system calls to handle threading:

```
int clone(void *(*func) (void *), void *arg, void *stack);
int join(int pid, void **stack, void **retval);
void texit(void *retval);
```

The implementations of these system calls should be as follows:

- **clone(...)** - This system call should behave as a lightweight version of **fork(...)**. In particular, the system call should create a new OS process (**struct proc**) that has its own kernel stack and other structures, but shares the memory address space of the parent. When a user calls **clone(...)**, they should provide a function for the thread to run, a single pointer to an argument, as well as a one-page region of memory to be used as a user stack. The **clone(...)** call should return the PID of the new thread to the parent, and immediately start execution of the function **func** in the new thread's context.
- **join(...)** - This system call should behave similar to **wait()**, and cause the caller to sleep until the thread specified by the *pid* argument terminates. **join(...)** should return 0 on success (negative number if an error), as well as copy the address of the thread's user stack and the return value (passed to **texit(...)**) into the pointers provided as parameters.

- **texit(...)** - This system call should behave similar to **exit()**, but takes a pointer that should be passed to the caller of **join(...)**.

Additionally, you should consider the behavior of some some other system calls:

- **exit()** - If called by a process with active child threads, those threads should be killed and cleaned up before the parent exits.
- **kill()** - If called on a process with active child threads, those threads should be killed and cleaned up before the parent is killed.

2.1.2 Userspace

Once you have implemented the kernel threads facility as well as supporting syscalls, you must implement a pthreads-compatible interface for those syscalls in userspace.

Implement the following in **pthread.c**:

```
int pthread_create(pthread_t *thread, const pthread_attr_t *attr,
        void *(*start_routine) (void *), void *arg);
int pthread_join(pthread_t thread, void **retval);
int pthread_exit(void *retval);
```

These functions should simply be wrappers around your kernel-level **clone()/join()/texit()** facility. You should implement the types required in a new **pthread.h** header file. You may safely use any type for the ***attr_t** types.

2.2 Testing

The starting branch mentioned above will contain several test programs that can be used to test your new threading facility. The programs are as follows:

- **test_clone** - Test the raw system calls.
- **test_pthread** - Test the PThreads API.

These programs are commented out of the Makefile by default. You must uncomment the appropriate lines to compile the programs.

3 Implementing Synchronization Primitives

Now that your xv6 has kernel-level threads, you need to provide synchronization primitives to allow programmers to construct correct multithreaded programs. For this project, you will implement mutexes.

3.1 Requirements

3.1.1 Kernel

You need to extend the xv6 kernel to add support for mutexes. To do this, you will construct a **per-process** tables of 32 mutexes, each addressed by a unique id (offset into the table). Each entry in the table can simply be a struct that contains whatever information required to represent one instance of a mutex.

Additionally, you will need to implement a set of system calls to allow user programs to manage and use these mutexes. These calls should be as follows:

```
int mutex_init(void);
int mutex_destroy(int mutex_id);
int mutex_lock(int mutex_id);
int mutex_unlock(int mutex_id);
```

The implementations of these system calls should be roughly as follows:

- **mutex_init(...)** - Initialize a mutex and mark it as active, then return its id.
- **mutex_destroy(...)** - Mark the mutex as destroyed. A mutex cannot be used again after this call unless it is reinitialized.
- **mutex_lock(...)** - Try to obtain the mutex. If the mutex is already locked by another thread, block the calling thread until the mutex is again available.
- **mutex_unlock(...)** - Release the mutex, and if there are other threads waiting to, obtain it, unblock one.

You should implement process blocking internally using the xv6 **sleep(...)/wakeup(...)** functions.

*Note that threads *MUST* share the mutex-related data structures with their parents and with each other, otherwise, the **mutex_id** will not mean anything across threads.*

3.1.2 Userspace

Once you have implemented the synchronization facility as well as supporting syscalls, you must implement a pthreads-compatible interface for those syscalls in userspace.

Implement the following in **pthread.c**:

```
int pthread_mutex_destroy(pthread_mutex_t *mutex);
int pthread_mutex_init(pthread_mutex_t *mutex,
                       const pthread_mutexattr_t *attr);
int pthread_mutex_lock(pthread_mutex_t *mutex);
int pthread_mutex_unlock(pthread_mutex_t *mutex);
```

These functions should simply be wrappers around your kernel-level synchronization facility. You should implement the types required in a new **pthread.h** header file. You may safely use any type for the ***attr_t** types.

3.2 Testing

The starting branch mentioned above will contain a test program called **test_mutex** that will test your new synchronization facility. Your implementation must be able to pass the tests performed by this code. *This program is commented out of the Makefile by default. You must uncomment the appropriate line to compile the program.*

4 Source Control

To start on this project, use the following commands in your xv6 git repository (assuming you have checked it out as instructed in the document uploaded to Sakai):

```
git fetch origin
git checkout sp16-hw2-start
```

After checking out, use the following commands to create your working branch:

```
git branch sp16-hw2
git checkout sp16-hw2
```

Please ensure that you **git commit** often while you are working. We recommend that you commit each time you have made a complete, significant change, and use a descriptive commit message. Typically in systems development, one feature may be implemented over dozens of individual commits

5 Submission

To submit your work, please follow the following instructions:
Type the following (for each netids field, use both partner's netids, hyphenated like **NETID1-NETID2**):

```
git checkout sp16-hw2
make submit netids=<NETIDS> name=hw2 base=sp16-hw2-start
```

After this is complete, you should have a .tar.gz file containing individual patch file(s) for each commit. Ensure that the tar file actually contains the patches, then submit this tarball on Sakai.

Only ONE student from each group should commit, but you NEED to make sure you give the netids of both partners.

6 Overall Requirements

- The code has to be written in **C** language. You should discuss on Piazza if you think inline **asm** is necessary to accomplish something.
- Do not copy the solution from other students. Use Piazza to discuss ideas of how to implement it. Do not post your solution there. Use Sakai to submit it.
- Commit in your local git for each step of your solution to make sure you can isolate each step later in case you need to refer back.
- Submit a report on **Sakai** named **report.pdf**, detailing what you accomplished for this project, including issues you encountered. This report *must* be named **report.pdf** and *must* be in PDF format.