Brian Crafton
Operating Systems
27 April 2016

Kernel Threads

The assignment that I chose to implement was kernel threads. Despite requiring lots reading code, reading course notes, and reading other online material the implementation ended up being fairly simple. I added two system calls which were clone and join. I also added user functions that built off of these called pthread_create and pthread_join. Lastly I added support for locks by adding a lock structure and pthread_mutex_init, pthread_mutex_lock, and pthread_mutex_unlock.

Clone was by far the most challenging function. I based it off of fork and did a lot of reading the code and online material. I used the same interfaces that were described in the University of Wisconsin's homework assignment (see references). Clone's interface is int clone(void *stack, int size). It is passed the stack that will be used because a thread needs its own stack. Because it is similar to fork, I started with fork and made the necessary changes. Clone and fork differ in that clone uses the same memory space as its parent. So rather than copy the memory space I simply assigned the child process's page directory to point to the parents. This way it would share memory with the parent. The next issue I ran into was the stack. I could not simply copy the stack because there is no reference to the bottom of the parents stack, only the stack pointer and base pointer. Because clone is being called from by the thread create function it is not necessary for the stack to be a complete copy of the parents stack. It need only be a copy of the stack of the function before clone was called. This way the child can return from the clone call and promptly enter the function it is meant to run without needing the stack prior to that point. After completing clone I wrote join. Join was much simpler I knew that it would perform very similarly to wait except that it would not deallocate the process's memory because then it would destroy the parents virtual memory space.

After having the clone and join system calls I was able to begin implementing the pthread_create and pthread_join user level functions that would call the system calls. The pthread_create function would need to call clone which means that it needs to make a clone of the process and then call the function pointer only in the child process. So a conditional statement would be needed to say if the process returning from clone is the thread then call this function, and otherwise return with the child's pid. A stack also needs to be allocated here to be passed to the clone function. This is because a when calling the pthread_create function a user does not want to have to allocate the stack that will be used. After calling the function pointer, the process needs to free its own stack and exit so that the parent process can find it and handle the zombie process. The pthread_join function simply called the join system call.

The last part that I implemented was the locks. This was very easy because xv6 already has the spin lock implementation so I ended up reusing a lot of that code. In order to initialize the lock all that needs to be done is to set the lock state to 0. In order to lock the lock, the thread must call xchg until it gets the lock. In order to free the lock xchg was called with 0 to free the lock.

In order to test this code, I added a new user program called test so that I could test the new user functions, the locks, and the system calls together. All the code does is spawn 5 threads that try to access an integer variable in shared memory and increment it 100,000,000 times. In the end the value of the variable should be 500,000,000 because the 5 threads will all have added 100,000,000 to it. In order to add to it each thread must get the lock and then add all 100,000,000 to it at once. This way each thread will have a chance to add to the variable and will make use of the locks. This code can be run by calling the user program test in the xv6 shell. The results of this program should be as followed:

```
$ test
100000000
200000000
300000000
400000000
500000000
500000000
$
```

References:
pages.cs.wisc.edu/~remzi/Classes/537/Spring2011/Projects/p6.html
http://unixwiz.net/techtips/win32-callconv-asm.html