Dr. Reza Entezari-Maleki

Fall 2021

Final Project
Operating Systems

Keyvan Dadashzadeh, Sina Shabani, Hadi Sheikhi

Due Date: 23:59:59 19th Jan, 2022

1 Introduction

In this project, you'll be adding real kernel threads to xv6. Sound like fun? Well, it should. Because you know fun is our major goal with these projects. 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, a thread_join() 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 adds values to a shared counter.

2 Sytem Call Implementations

2.1 Cloning New Thread

Your new system call should look like this: int clone(void *stack, int size). It does more or less what fork() does, except for one major difference: instead of making a new address space, it should use the parent's address space (which is thus shared between parent and child). You might also notice a single pointer is passed to the call, and size; this is the location of the child's user stack, which must be allocated before the call to clone is made. Thus, inside clone(), you should make sure that when you return, you are running new functions on this stack, instead of the parent's stack. You can access to the parent stack same as the fork() call or by calling myproc() function.

As with fork() , the clone() call returns the pid of child to the parent, and 0 to the newly-created child thread. In the Linux kernel, threads are somehow processes. To be more specific, you should do these steps in the clone():

- define a new proc structure
- set the listed values same as parent (current running process):
  - pgdir (the difference between fork and clone is here, you do not need to copy all pages of parent process, because it is just a thread and has a shared space with the parent process, so a simple assign can solve this part.)
  - sz
  - value inside the tf pointer

  - current working directory (cwd) using idup()
  - child's name. You can copy parent's name using safestrcpy()

- set the trap frame's %eax pointer to 0 (if you know assembly, this returns 0 in the child process, which is the new process you created.
- calculate stack size
- set the stack pointer of new process
- calculate size needed above EBP
- move base pointer of new process below topsize (calculated in the previous item)
- copy parent process's stack to child with memmove() function. (for more information, you should read xv6's document)
- copy all open files of parent process to child process with filedup() function
- set the state of child process to RUNNABLE state.
- return the pid of child process

2.1.1 Calculating Stack Size

As we described above, you have to calculate stack size. But it is not as easy as you think! Let's take a look at the pointers when a function call occurs.

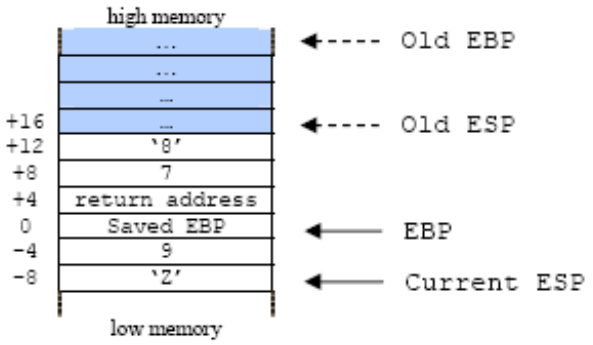

Figure 1: Function calls and stack pointers

As we can see, when a new function call occurs, we should change the stack pointers by considering the old EBP and ESP.

ESP is the stack pointer which grows by allocating new variables in the function.
EBP is so called base pointer which shows the start of stack.
Now imagine we want to calculate stack's size of a function which was called before clone (Old EBP, Old ESP). What should we do? Old EBP is the Saved EBP in the current function. Old ESP is available in the parent process's trapframe.
Stack size can be calculated by subtracting ESP from EBP.

2.1.2 Size Above EBP

Same as stack size, we can simply calculate the size above the current EBP by subtracting current EBP from old EBP. We call this a topsize space.

2.2 Wait For My Child!

In this part you will implement a system call for thread_join(). If you take a look at the wait() system call in xv6, you may realize that it would be the very same thing! And yes, you have to implement a new system call join() same as the wait() with a difference. As you can see, in the wait system call, we will free all the pages of process. Now consider the point that all threads in a process have the same pages shared between them! What a mess! Your task is to handle this difficulty by considering that if the process is not a child process (by comparing its pgdir with the parent), you are allowed to free the pages of ZOMBIE process which was found recently. Other parts of the function are exactly same with wait() system call.

3 Thread Library Implementations

3.1 Create Thread

Your thread library will be built on top of this, and just have a simple thread_create(void *(*start_routine)(void*), void *arg) routine. This routine should use clone() to create the child, and then call start_routine() with the argument arg .

3.2 Join

Your library will also have a thread_join() routine. This call, made by the parent of the thread, simply waits for a thread that the parent created to exit. You can use join() system call in this part.

3.3 Lock

Your thread library should also have a simple spin lock. There should be a type lock_t that one uses to declare a lock containing a simple unsigned integer, 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 to built the spin lock (see the xv6 kernel for an example of something close to what you need to do, the xchg function). One last routine, lock_init(lock_t *) , is used to initialize the lock as need be.

4 Test Your Code!

To test your code, you should build 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 one to a shared counter. The counter should be made thread safe using locks of course. At the end, the main thread should call thread_join() (repeatedly, once per child) to wait for all the children to complete; at this point, the main thread should print out the value of the counter.