CS 537 Spring 2019, Project 4b: Kernel Threads

In this project you will be adding real kernel threads to xv6.

Specifically, you will do three things. First you will 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 will use `clone()` to build a little thread library, with a `thread_create()` call and `lock_acquire()` and `lock_release()` functions. That's it!

Overview

Your new clone system should look like - `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 two arguments (arg1 and arg2) and uses a fake return PC(`0xffffffff`). The stack should be one page in size and page-aligned. The new thread starts executing at the address specified by `fcn()` . As with `fork()` , the PID of new thread is returned to the parent (for simplicity, each thread has its 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 thread to exit. It returns the PID of the waited-for child or -1 if none. The location of child's user stack is copied into the argument `stack` (which can be freed after).

You also need to think about the semantics of a couple of existing system calls. For example, `int wait()` should not free the address space of a thread until all threads sharing the same address space have exited (i.e., `(p->state != ZOMBIE)`). It should only 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 `int thread_create(void (*start_routine)(void *, void *), void *arg1, void *arg2)` routine. 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 (http://pages.cs.wisc.edu/~remzi/OSTEP/threads-locks.pdf) 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 *)` and `void lock_release(lock_t *)` , which acquire and release the lock. The ticket lock should use x86 atomic add to build the lock – see this wikipedia page (https://en.wikipedia.org/wiki/Fetch-and-add) for a way to create an atomic fetch-and-add routine using the x86 `xaddl` instruction. For more details on how to use assembler instructions with C – see this (https://gcc.gnu.org/onlinedocs/gcc/Extended-Asm.html). 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.

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!

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. This is you can read about in Programming From The Ground Up (https://download-mirror.savannah.gnu.org/releases/pgubook/ProgrammingGroundUp-1-0-booksize.pdf), 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.

Extra Pointers

1. Start Early !!
2. Watch Prof. Remzi's old discussion video (https://www.youtube.com/watch?v=G9nW9UbkT7s) specifically the place where function call semantics are explained.
3. Use GDB.
4. Use `fork()` as the blueprint for `clone()` .
5. Read `wait()` and then implement `join()` . Also ensure wait is modified as defined before.
6. Once the syscalls are implemented then hopefully the thread library should fall in place.

Code

Start with a fresh copy of xv6, which you can get

```
cp -r ~cs537-1/xv6-sp19 ./
```

Testing

You can run an individual test with the following command:

```
/u/c/s/cs537-1/tests/p4b/runtests testname
```

1. Functionality of clone: *clone_basic*
2. Correctness of the stack pointer argument to `clone()` and `join()` : *clone_bad*
3. Correctness of the function arguments `arg1, arg2` of `clone()` : *clone_arguments*
4. Verify the modification of `wait()` according to spec: *clone_wait*, *fork_join*
5. Functionaliy of lock: *lock_basic*
6. Correctness of `thread_create()` and `thread_join()` : *threads_basic*
7. Growing the address space in a multi-threaded process: *threads_sbrk*
8. Create multiple threads and child processes to check if the process crash due to memory leak: *threads_many*

Due Date

April 16th, 2019 by 11:59 PM

Submitting your implementation

The handin directory is `~cs537-1/handin/<cs-login>/p4b/` . To submit the solutions copy all xv6 files and directories with your changes to the handin directory. One way to do this would be to navigate to your solution and execute

```
cp -r . ~cs537-1/handin/<cs-login>/p4b/
cd ~cs537-1/handin/<cs-login>/p4b/ && make && make clean
```

Consider the following when submitting the solution -

- If you use any slip days you have to create a `SLIP_DAYS` file in the `/<cs-login>/p4b/` directory otherwise we use the submission on the due date.
- Your files should be directly copied to `~cs537-1/handin/<cs-login>/p4b/` directory. Having subdirectories in `<cs-login>/p4b/` like `<cs-login>/p4b/xv6-sp19` is **not acceptable**.

Collaboration

This project is to be done in groups of size one or two (not three or more). Now, within a group, you can share as much as you like. However, copying code across groups is considered violation of plagiarism policy.

If you are planning to use `git` or other version control systems (*which are highly recommended for this project*), just be careful **not** to put your code in a public repository.

Acknowledgement

The project uses material created by Prof. Remzi for his OS class offered in Spring 2018.