P3B: xv6 Kernel threads

Due Thursday by 9pm **Points** 120
Available Mar 9 at 12am - Apr 13 at 11:59pm about 1 month

xv6 Kernel threads

In this project, you'll be adding real kernel threads to xv6. Sound like fun? Well, it should. Because you are on your way to becoming a real kernel hacker. And what could be more fun than that?

To accomplish this, you will need to understand the structure of a stack, xv6 process state, and per-process memory layout. And, most importantly, you will learn the interaction between parent and child thread.

Specifically, there are 3 steps in this project:

- add 2 new system calls to create a kernel thread: `clone()`, and `join()`.
- Use the new system calls to build a little thread library, with `thread_create()` and `thread_join()` functions, and build a `ticket lock` primitive to protect thread synchronization.
- Use your thread library to complete 2 multi-threaded user programs.

Updates:

- Correction to the discussion slides: in lock_acquire, the checking while loop may also need be atomic. i.e. `while( fetch_and_add(&lock->turn, 0) != turn );`
- The example user programs along with some initial tests can be found in `/p/course/cs537-swift/tests/p3b/user_programs/` (the 2 example programs will only work only if the top-level thread library is complete; make sure to pass the other tests first). **All tests for P3B have been made available under `/p/course/cs537-swift/tests/p3b/`.** On any CSL machine, use `cat /p/course/cs537-swift/tests/p3b/README.md` to read more details about how to list the tests and how to run the tests in batch. **The `/p/course/cs537-swift/tests/p3b/user_programs/pi.c` example is update with correct locks given.**
- You can work in groups of 2 from this project.
- To resolve `undefined reference to malloc/free` from forktest, append umalloc.o to the first line of the _forktest rule in make file: `$(LD) $(LDFLAGS) -N -e main -Ttext 0 -o _forktest forktest.o ulib.o usys.o umalloc.o`

Specifications

You can either code directly on a CSL machine, or on your own machine. Remember to test that your code compiles on a CSL machine before submitting it. You will be using the current version of xv6. Copy the xv6 source code into your private directory using the following commands:

```
prompt> cp /p/course/cs537-swift/public/xv6.tar.gz .
prompt> tar -xvzf xv6.tar.gz
```

1) New system calls

1.1. clone `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`); a proper thread will simply call `exit()` when it is done (and not `return`). 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 the new thread is returned to the parent (for simplicity, threads each have their own process ID).

Hints

Recall that in a process, a child thread shares everything with the parent thread but its private stack. In xv6, each thread has a user stack and a kernel stack. This system call should set up the proper states for the cloned child process, create its own private user stack and kernel stack, and populate the stack and registers with proper values.

1.2. join `int join(void **stack)`

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 process to exit. It returns the PID of waited-for child or -1 if none. The location of the child's user stack is copied into the argument `stack` (which can then be freed).

You also need to think about the semantics of a couple of existing system calls. For example, `int wait()` should wait for a child process that does not share the address space with this process. It should also free the address space if this is last reference to it. Also, `exit()` should work as before but for both processes and threads; little change is required here.

Hints

Similar to how a parent process should wait for its child processes, `join()` will just be the thread-equivalent of `wait()` (which is also in proc.c. surprise, surprise.) What should be different this time? Look into the code and you will find `wait()` just wipes out a zombie child process from the ptable. Think about what is shared between parent-child threads but not parent-child processes. They should not be destroyed by your join.

2) Thread library

2.1. thread_create `int thread_create(void (*start_routine)(void *, void *), void *arg1, void *arg2)`

This routine should call `malloc()` to create a new user stack, use `clone()` to create the child thread and get it running. `start_routine` and the 2 arguments are just the function pointer and arguments to be passed to `clone`.

It returns the newly created PID to the parent and 0 to the child (if successful), -1 otherwise.

2.2. thread_join `int thread_join()`

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.

2.3. ticket lock

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 spin 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. 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).

Hints

xv6 already supports 2 types of locks: `sleeplock` and `spinlock`. You have already encountered some instances of them in the previous 2 projects (i.e. the ptable lock in proc.c). Take a look at how `spinlock.c` is implemented with atomic `xchg(dest, src)`. It is an indivisible operation that is guaranteed to happen at once and protect the variables from race condition: in short, it means no other threads can see or change the content of variables `dest` and `src` in parallel. If all operations on a variable are atomic, no thread can see intermediate values, and the variable is thread-safe.

Your ticket lock implementation should use `FAA()` to atomically fetch and increment the variables in your `lock_t` struct. You can use the example implementation in the last slide from the 3/9 discussion. See updates for lock_acuqire() changes

3) User-Level Application and testing

To demonstrate that your thread library is doing at least some of the right things, you might want to create user-level applications. See updates for the testing programs.

First, you may want to write your own user-level program and try the provided tests. After you have passed all the tests, we will also provide 2 user programs based on your thread library that are initially not protected by locks, which is left to you to complete:

`1. race` where producer and consumer threads collaborate on a shared buffer

`2. pi` where worker threads use random sampling to approximate the value of π.

Your completed programs, along with TA's implementation of the same programs, will be used as part of the tests and contribute to your grade.

4) 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 `proc.c`, we see the beginning of the `fork()` implementation:

```
// Create a new process copying p as the parent.
// Sets up stack to return as if from system call.
// Caller must set state of returned proc to RUNNABLE.


int
fork(void)
{
  int i, pid;
  struct proc *np;
  struct proc *curproc = myproc();

  // Allocate process.
  if((np = allocproc()) == 0){
    return -1;
  }

  // Copy process state from proc.
  if((np->pgdir = copyuvm(curproc->pgdir, curproc->sz)) == 0){
    kfree(np->kstack);
    np->kstack = 0;
    np->state = UNUSED;
    return -1;
  }
  np->sz = curproc->sz;
  np->parent = curproc;
  *np->tf = *curproc->tf;

  // Clear %eax so that fork returns 0 in the child.
  np->tf->eax = 0;

  for(i = 0; i < NOFILE; i++)
    if(curproc->ofile[i])
      np->ofile[i] = filedup(curproc->ofile[i]);
  np->cwd = idup(curproc->cwd);

  safestrcpy(np->name, curproc->name, sizeof(curproc->name));

  pid = np->pid;

  acquire(&ptable.lock);

  np->state = RUNNABLE;
```

```
  release(&ptable.lock);

  return pid;
}
```

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!

6) 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.

Hints

Under `/p/course/cs537-swift/tests/p3b/assembly` is a simple xv6 HelloWorld program. Create a local copy of the directory and inspect its contents. Inside `hello_txt_dump` you can find the following assembly (uncommented) for the `hello` function in `hello.c`. It helps you understand how the stack is set up at the start of a new function.

```
;hello function in C looks like this
;void hello(int arg) {
;        printf(1, "Hello %d\n", arg);
;        exit();
;}
;
;compile to x86 format asm
00000020 <hello>:
20:   f3 0f 1e fb             endbr32  ; ignored this
  ; update stack pointer and base pointer
24:   55                      push   %ebp ; save base pointer register(%ebp) onto the stack
25:   89 e5                   mov    %esp,%ebp ; set base pointer register to the current stack top
```

```
  ; Before calling a routine (i.e. <printf>), all arguments need to be first pushed onto the stack
27:   83 ec 0c                sub    $0xc,%esp ; grow stack, make space for the 3 arguments of print
f
2a:   ff 75 08                pushl  0x8(%ebp) ; arg
2d:   68 68 07 00 00          push   $0x768 ; "Hello %d\n"
32:   6a 01                   push   $0x1 ; 1
34:   e8 c7 03 00 00          call   400 <printf> ; calling printf
39:   e8 65 02 00 00          call   2a3 <exit>
3e:   66 90                   xchg   %ax,%ax
```

6) Handing in your Code

If you are using slip days on this project, you must copy your code into the corresponding slip directory. Similar to p2b, create a file called **slip_days** with the full pathname `/p/course/cs537-swift/turnin/login/p3b/slipdays`, where ***login*** is your CS login name***.*** We will grade the latest submission.

To hand in code, create a `src/` directory under `ontime` or `slipX` and put all the files under your `xv6/` under that `src/` folder.

The final directory tree should look like:

```
/p/course/cs537-swift/turnin/login/p3b/race.c -- the completed producer-consumer race program
/p/course/cs537-swift/turnin/login/p3b/pi.c -- the completed pi estimation program
```

```
/p/course/cs537-swift/turnin/login/p3b/partners.txt
```

```
/p/course/cs537-swift/turnin/login/p3b/slip_days -- should contain exactly a number 1
/p/course/cs537-swift/turnin/login/p3b/slip1/src/Makefile
/p/course/cs537-swift/turnin/login/p3b/slip1/src/proc.c
/p/course/cs537-swift/turnin/login/p3b/slip1/src/...
```