Project #1

WARNING FROM FRIENDLY PROFESSOR

This is certainly the most difficult project thus far. Start early!

*** If possible, do not change the "xv6/Makefile".*
*(It's okay to change the "xv6/user/makefile.mk") ***

1. Threads in xv6

In this project, you'll be adding kernel threads to xv6 and implement a user-level thread library. Specifically, you need to 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()` and `thread_join()` call, `lock_acquire()` and `lock_release()` functions.

1.1. Details

Here is a detailed explanation about what you should implement.

- `int clone(void (*fcn)(void *), void *arg, void *stack)`

  This call creates new kernel thread which shares the calling process's address space. File descriptors are copied as in the `fork`. The new process uses `stack` as its user stack, which is passed the given argument `arg` and uses a fake return address (`0xFFFFFFFF`). The stack should be one page in size (4K) 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.

- `int join(void **stack)`

  This call waits for a child thread that shared the address space with the calling process. It returns the PID of the waited-for child or -1 if none. The location of the child's user stack is copied into the argument `stack`.

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

Your thread library will be built on top of these two system calls.

- `int thread_create(void (*start_routine)(void *), void *arg)`

  This function should call `malloc()` to create a new user stack and use `clone()` to create the child thread and get it running. And return PID of child thread.

- `int thread_join()`

  This function should call `join()`, frees the user stack, and returns the PID of the waited-for child thread. Of course, -1 if no child thread.

Your thread library should also have a simple spin lock. There should be a type `lock_t` that one uses to declare a lock. The spin lock should use x86 atomic exchange to build the spin lock. (See the xv6 kernel for an example of something close to what you need to do).

- `int lock_init(lock_t *lk)`

  This function initializes the `lock_t` structure. Always return 0.

- `int lock_acquire(lock_t *lk)`

  This function acquires lock using `lock_t` structure. Always return 0.

- `int lock_release(lock_t *lk)`

This function release lock using `lock_t` structure. Always return 0.

One thing you need to be careful with is when an address space is grown by a thread in a multi-threaded 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.

1.2. Hints

You need to modify/add codes mostly in the following files:

- `include/syscall.h`
- `kernel/defs.h`
- `kernel/proc.c`
- `kernel/proc.h`
- `kernel/syscall.c`
- `kernel/sysfunc.h`
- `kernel/sysproc.c`
- `user/user.h`
- `user/usys.S`
- `New file that contains your thread library`

As you know, there is no ‘thread’ in xv6. But you don’t need to implement ‘thread’ from scratch. You can just use the ‘process’ of xv6 as kernel thread. This means the `clone` function should work like creating a new kernel process that shares address space with the calling kernel process (c.f. `fork` function copies everything).

2. Deliverable

Create the following two system calls in the xv6 kernel.

- `clone()`
- `join()`

Create the following five functions in thread library (user-level).

- `thread_create()`
- `thread_join()`
- `lock_init()`

- `lock_acquire()`
- `lock_release()`

Define the following new data type

- `lock_t`

3. Grading Criteria

We will use 15 scripts for grading. You can check the descriptions below.

```
clone1 (9): Clone and verify that address space is shared

clone2 (6): Clone and play with the arguments

clone3 (3): Clone and test copied file descriptors (which are not
for sharing)

bad clone (6): Test the return value of clone() with bad stack
arguments

stack (8): Test if it set up stack correctly (and without extra
items)

join1 (9): Basic clone & join test

join2 (6): Test if it manages arguments correctly

join3 (3): Test if join does not handle child processes (forked)

join4 (3): Check the return value of wait()

thread1 (10): Basic thread_create & thread_join test

thread2 (7): Memory leak test

multi (8): Multiple threads with some depth of function calls

locks (8): Basic lock test

noexit (7): Test its behavior when there is no exit call in a
thread

size (7): Test if address space size is updated in threads
```

After finishing your implementation, compress your xv6 source codes using the commands below. Then, submit the compressed file on KLMS.

```
#First, move to a directory containing the xv6 directory
mv <DIRECTORY_CONTAINING_THE_XV6_DIRECTORY>

#Then, compress your xv6 directory
tar  -czvf  <Student ID>_proj1.tar.gz  xv6
```