Project 3

Project 3: Threads

Note: It might be useful to print out this project description page for easy reference.

Important Dates

Out: Monday, October 31st, 2016.

~~**Due:** Wednesday, November 9th, 2016 (11:59 PM CDT).~~

Due: Friday, November 11th, 2016 (11:59 PM CDT).

Submission instructions

Please join a Project 3 Group on Canvas as soon as possible. The available groups can be found **here (https://canvas.northwestern.edu/courses/43589/groups)** . You can join "group 1" through "group 35". Just make sure that you and all your teammates join the same group.

To submit, make sure your xv6 directory contains a file called **team.txt**, which contains all the netids of the members of your group on separate lines. Be sure to type **make clean** in your xv6 directory, then cd up to the directory that contains your xv6 directory. Create a .tar.gz of your xv6 directory by typing:

```
$ tar -zcvf xv6-project3.tar.gz xv6
```

 You must then upload the .tar.gz on the assignment page on Canvas here: **https://canvas.northwestern.edu/courses/43589/assignments/276357 (https://canvas.northwestern.edu/courses/43589/assignments/276357) (https://canvas.northwestern.edu/courses/43589/assignments/276357)**

Rubric

	Points Possible
team.txt file is included and correct	5
project 3 tests	95
TOTAL	**100**
extra credit	20

(https://canvas.northwestern.edu/courses/43589/assignments/276357) (https://canvas.northwestern.edu/courses/43589/assignments/276357)

Resources

- Some useful GDB and QEMU commands: **https://pdos.csail.mit.edu/6.828/2016/labguide.html** ⧉ **(https://pdos.csail.mit.edu/6.828/2016/labguide.html)**
- xv6 textbook: **https://pdos.csail.mit.edu/6.828/2012/xv6/book-rev7.pdf** ⧉ **(https://pdos.csail.mit.edu/6.828/2012/xv6/book-rev7.pdf)** (note that we are using a version of the code from 2012)
- Guide to xv6 codebase: **https://pdos.csail.mit.edu/6.828/2012/xv6/xv6-rev7.pdf** ⧉ **(https://pdos.csail.mit.edu/6.828/2012/xv6/xv6-rev7.pdf)**
- Hexadecimal converter: **https://www.mathsisfun.com/binary-decimal-hexadecimal-converter.html** ⧉ **(https://www.mathsisfun.com/binary-decimal-hexadecimal-converter.html)**

Project Overview

In this project you will implement threads in xv6. This project must be completed in groups of 2-3 people.

Project (100 points)

This project requires you to create two new syscalls to implement threads in xv6. The syscalls are **clone** and **join**. The **clone** syscall creates a new thread. The **join** syscall waits for a child thread to finish. You will also have to make some changes to a few other places in the code to get everything working properly.

Extra credit (20 points)
For extra credit, you can implement a user library of thread-related functions. See below for more details.

Educational Objectives

This project aims to achieve several educational objectives:

- Understand what threads are, how they work, and why they are useful.
- Learn how to implement threads.
- Understand synchronization and its importance when dealing with threads.
- Practice writing your own test cases.
- Level up to Experienced Kernel Hacker.

The Code

You should start from a clean version of the **original xv6 code (http://www.aqualab.cs.northwestern.edu/class/333-eecs343-xv6#download-xv6)** .

Project Description and Requirements

100 points

You will implement threads as lightweight processes which share the same address space as their parent process. This approach is how **LinuxThreads** ⬈ **(https://en.wikipedia.org/wiki/LinuxThreads)** was implemented on earlier versions of the Linux kernel, although it has since been superseded in newer versions.

We will explain the project by going over the syscalls that you need to create in detail.

1) clone

The **clone** syscall creates a new thread of execution by creating a clone of the current process with the same address space. Here is the function signature:

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

The first argument, **fcn**, is a function pointer. It points to a function where the new thread will begin execution. For now, let's refer to this function as **DoThreadWork**.

The second argument, **arg**, is a pointer to some data that will be passed as an argument to the **DoThreadWork** function.

The third argument, **stack**, is a pointer to a page in user space that the new thread will use as its call stack.

Overall, **clone** is very similar to **fork**. It creates a new entry in the process table and copies over the parent process's information into the new entry. However, **clone** does not create a new copy of the memory associated with the parent process. Instead, the new thread will use the same address space as the parent process. Another difference between **clone** and **fork** is that in **fork**, both the parent and the child return from **fork** to the same place, namely the next instruction after the call to **fork**. However, in **clone**, the new thread begins execution in the function that is designated by the function pointer **fcn**.

Here is an example of a simple user program that makes use of the **clone** syscall:

```
#include "types.h"
#include "user.h"


#define PGSIZE (4096)


volatile int global = 1;
```

```
void DoThreadWork(void* arg_ptr);  // function signature for our DoThreadFunction


int
main(int argc, char* argv[])
{
 void* stack = malloc(PGSIZE*2);  // allocate 2 pages of space
 if((uint)stack % PGSIZE) {
   // make sure that stack is page-aligned
   stack = stack + (PGSIZE - (uint)stack % PGSIZE);
 }

 void* arg = NULL;  // our DoThreadWork function is simple and doesn't need an arg
 int clone_pid = clone(DoThreadWork, arg, stack);
 // the main thread of execution (aka parent process) continues executing here
 while(global != 5) {
   ; // wait for child thread to finish
 }
 printf(1, "global: %d\n", global); // prints "global: 5"
 exit();
}


void
DoThreadWork(void* arg_ptr) {
 // clone creates a new thread, which begins execution here
 global = 5;
 exit();
}
```

Requirements (and hints)

Here is a list of specific requirements related to the **clone** syscall:

- The **clone** syscall must use the exact function signature that we have provided above.
- The **clone** syscall must create a new thread of execution with the same address space as the parent.
- The new thread must get a copy of the parent's file descriptors.
    - Hint: see **fork**.
- The new thread must begin execution in the function pointed to by **fcn**.
    - Hint: what register keeps track of the current instruction being executed?
- The new thread must use the page pointed to by **stack** as its user stack.
- The pointer **arg** will be passed to the function (specified by **fcn**) as an argument.
    - Hint: review the calling convention. When a function is called, where on the stack will it expect to find an argument that was passed in?
- The function (specified by **fcn**) will have a fake return address of 0xffffffff.
    - Hint: where on the stack does a function expect to find the return address?
- Each thread will have a unique pid.
- The **parent** field of the thread's **struct proc** will point to the parent process (aka the main thread). If a thread spawns more threads, their **parent** field will point to the main thread (not to the thread that spawned them).
- The **clone** syscall must return the pid of the new thread to the caller. However, if a bad argument is passed to **clone**, or generally if anything goes wrong, **clone** must return -1 to the caller to indicate failure.
    - Hint: **clone** should check that the **stack** argument it receives is page-aligned and that the full page has been allocated to the process.
- In a multi-threaded process, when a thread exits, all other threads must be able to continue running. However, when the main thread (parent process) exits, all of its children threads must be killed and cleaned up.
    - Hint: the **exit()** code will have to change. Also, it may be helpful to add a flag to the **struct proc** indicating whether this entry is a child thread or the main thread.
- In a multi-threaded process, multiple threads must be able to grow the address space without causing race-related errors.

- Hint: walk through the code path for the sbrk syscall. You will need to use a lock in there somewhere.

2) join

As you have seen, **clone** is sort of like **fork**, except for threads. Well, **join** is sort of like **wait()**, except for threads. It waits for a thread to complete. Here is the function signature:

```
int join(int pid)
```

The argument, **pid**, is the pid of the thread to wait for.

Here is an example of a simple user program that makes use of the **join** syscall.

```
#include "types.h"
#include "user.h"

#define PGSIZE (4096)

int global = 1;

void DoThreadWork(void* arg_ptr);

int
main(int argc, char* argv[])
{

 void* stack = malloc(PGSIZE*2);  // as before, allocate 2 pages
 if((uint)stack % PGSIZE){
    stack = stack + (PGSIZE - (uint)stack % PGSIZE); // make sure the stack is page aligned
 }

 int arg = 42;
 int clone_pid = clone(worker, &arg, stack);
 // main thread continues executing...
 int join_pid = join(clone_pid);  // ...but waits for the new thread to complete
                                   // note that join_pid should equal clone_pid
 printf(1, "global: %d\n", global);  // prints "global: 43"
 exit();
}

void
DoThreadWork(void *arg_ptr) {
 int arg = *(int*)arg_ptr;
 global = global + arg;
 exit();
}
```

Requirements (and hints)

Here is a list of specific requirements related to the **join** syscall:

- The **join** syscall must use the exact function signature that we have provided above.
- The **join** syscall must wait for the thread specified by pid to complete.

- ~~The **join** syscall must wait for the thread specified by pid to complete.~~
- The **join** syscall must return the pid of the completed thread. However, if the argument is invalid or generally if anything goes wrong, the syscall must return **-1** to indicate failure. Some notes on what constitutes an invalid argument:
    - Calling **join** on a main thread (a process) should result in a return value of **-1**.
    - Calling **join** on a thread belonging to a different thread group than the caller should also result in a return value of **-1**.
- The **join** syscall must clean up the process table entry that was being used by the thread.
    - Hint: see the how **wait** does this. However, be aware that **wait** cleans up some things that **join** must NOT clean up.
- ~~The **join** syscall must free up the physical memory that was being used for the thread's user stack.~~

3) side effects

Now that we have implemented threads as lightweight processes. The **wait** syscall must change slightly.

Requirements (and hints)

Here is a list of specific requirements related to the **wait** syscall:

- The **wait** syscall must only wait for child processes, not child threads.

Testing

We are providing some basic tests to help you get started. The purpose of these tests is:
1) to give you a testing framework,
2) to help you understand how a user program might invoke your syscalls, and
3) to make it easy to add your own tests.

The provided tests are intentionally not comprehensive and we strongly recommend that you add your own tests. If you are having trouble thinking of your own tests, see the list of Requirements. Typically, each requirement can be translated into *at least* one test. Sometimes you can write many many *many* tests for a single requirement.

One final recommendation. Modern software engineering principles emphasize writing your tests *before* writing your code. This is taught at Northwestern in EECS 111 as part of the **Design Recipe** [↗] **(http://www.htdp.org/2001-01-18/Book/node14.htm)**. In industry, you may hear it described as **test-driven development** [↗] **(https://en.wikipedia.org/wiki/Test-driven_development)**. The point is, we recommend writing tests first, in order to force yourself to define the desired outcome of your code. Then after you've written your code, you can run your tests to get immediate feedback on whether your code is correct or not.

Without further ado, here is how to use the provided tests. Download the .tar.gz below, copy it to a lab machine, and extract it. Inside, you will find a README with instructions on how to run the provided tests as well as how to add your own tests. If anything is unclear or doesn't work, please post to Piazza.
provided-basic-tests.tar.gz (https://canvas.northwestern.edu/courses/43589/files/3013885/download?wrap=1) [↗]
(https://canvas.northwestern.edu/courses/43589/files/3013885/download?wrap=1)

Extra Credit Description and Requirements

20 points

Note: Unlike Project 2, attempting this extra credit should not require modifying the basic functionality above. Therefore, for this project you can just make one submission. We will run the extra credit tests on all submissions.

We believe that the project requirements above are fairly manageable and so we hope that many of you will attempt the extra credit.

For extra credit, you will implement a user library that would allow a developer to make good use of your operating system's threads. This user library will include wrapper functions for creating and joining threads. It also includes synchronization-related functions which will make it easier to write thread-safe programs.

The functions below should be placed in a new file: **user/uthreadlib.c**

1. Thread wrappers
    1. **thread_create**

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

Creates a new thread by first allocating a page-aligned user stack, then calling the **clone** syscall. Returns the pid of the new thread.

2. **thread_join**

```
int thread_join(int pid)
```

Calls join to wait for the thread specified by pid to complete. Cleans up the completed thread's user stack.

2. Locks

Hint: See how spinlock is implemented in the kernel. You will want to use the atomic exchange function, **xchg**.

1. **lock_acquire**

```
void lock_acquire(lock_t* lock)
```

Acquires the lock pointed to by **lock**. If the lock is already held, spin until it becomes available.

2. **lock_release**

```
void lock_release(lock_t* lock)
```

Release the lock pointed to by **lock**.

3. **lock_init**

```
void lock_init(lock_t* lock)
```

Initialize the lock pointed to by **lock**.

3. Condition variables

Hint: In order to implement these user functions, you will have to implement new syscalls.

1. cv_wait

```
void cv_wait(cond_t* conditionVariable, lock_t* lock)
```

Release the lock pointed to by **lock** and put the caller to sleep. Assumes that **lock** is held when this is called. When signaled, the thread awakens and reacquires the lock.

2. cv_signal

```
void cv_signal(cond_t* conditionVariable)
```

Wake the threads that are waiting on **conditionVariable**.