CS 550 Operating Systems, Spring 2018
Programming Project 3

Out: 4/15/2018, Sunday
Due date: 5/13/2018, Sunday 23:59:59

Overview: In this project, you are going to add kernel thread support and implement a small thread library for xv6. This thread library is named `procThread`, as the thread creation API is similar to creating a process (i.e., `fork()`), but the threads created have the general properties of threads (e.g., shared data section/heap, and separated stacks).

1 Baseline xv6 source code

ATTENTION: before you proceed, read the whole spec carefully and make sure you understand all the requirements, especially those highlighted parts in this section, the submission requirements (Section 3), and the grading guidelines (Section 4).

- Log into your GitHub account whose username is the same your BU username.
- Go to the link at the end of this section to accept the assignment. This will create a private repository of your own on the instructor's teaching organization, and start importing the baseline code into your private repository.

  Note: sometimes the import process may take a long time. Do not click the "Cancel" button, which will lead to an empty repository. You can just close the page and come back later.

  If you have clicked the "Cancel" button or it already took really long (e.g., more than two hours), contact the instructor and the TAs so we can delete your repository, and you can click the assignment link to import again.
- Once the import process finishes, you can clone or download the repository into your computer and start working.
- Start early! The instructors and the TAs will be less responsive during the last several days before the due date, and will not respond to any email inquiries regarding the project in the last day.
- Refer back to the spec of PROJ0 and PROJ1 for details of running xv6.

Assignment link: `https://classroom.github.com/a/bbysJelJ`

2 Kernel support for threading, and the **procThread** thread library (100 points)

Your job for this project is to add kernel support for thread management (creation, deletion and joining) in xv6, then implement a small thread library (called `procThread`) based on the new kernel feature.

2.1 **procThread** library APIs

The following `procThread` APIs need to be implemented (in `procThread.c`).

- `int` **`procThread_create`**`(void)`

  This API creates a new thread within the calling process. This API has the following properties:

  - If the new thread is created successfully, both the new thread and the calling thread continue execution from the point where `procThread_create()` returns.
  - The return value of `procThread_create()` is
    - 0 if running in the new thread's context;
    - the thread ID of the newly created thread if running in the context of the calling thread;
    - negative on error.
  - The new thread and calling thread share the calling thread's code, data, and heap, but have separate stacks.

- `void` **`procThread_exit`**`(int ret_val)`

  This API terminates the calling thread and sets the return value of the thread (an integer) to `ret_val` (which can be obtained by the `procThread_join()` API described below).

- `void` **`procThread_join`**`(int tid, int * ret_val_p)`

  This API performs a join operation on the thread with thread id of `tid`. The return value of the thread `tid` is put into an integer allocated by the caller (i.e., `ret_val_p`).

2.2 Kernel support for **procThread** APIs

The key points of understanding and implementing the kernel support for `prochThread` library and interaction between kernel and `procThread` library are described as follows.

- Recall that all threads created within the same process shared the same code, data, heap, but have their own stacks. The private stack of a thread is used to store the function call arguments, local variables, and relevant CPU register values when functions calls are made.

  In this project, the stacks of the child threads (i.e., threads created by the main thread) should be managed by the `procThread` library. Read on for more details.

- You will need to define a new system call named `clone()`. This system call will be called by `procThread_create()` to request kernel to create a new thread in the current process.

  The new `clone()` system call would look like this:

```
int clone(void *stack)
```

This call creates a new thread that shares the calling process's address space.

- Once created, the new thread starts its execution from where the `clone()` system call returns.
- The new thread uses `stack` as its stack, which should be allocated by `procThread_create()` in the user space, and is passed to `clone()` when `clone()` is called. The stack should be one page in size (i.e., 4096 bytes).
- The thread ID of the new thread is returned when running in the parent thread's context, 0 is returned when running in the new thread's context, and a negative value is returned on error.
- For most part of the `clone()` system call, you may refer to the `fork()` system call in xv6 for the implementation.

The following are more hints about implementing the `clone()` system call.

- As suggested previously, the stack of the new thread should be allocated and freed by the `procThread` library in the user space. More specifically, in your implementation of the `procThread_create()` API, you will call `malloc()` to allocate the stack space for the new thread, and pass the address of the stack to kernel via the `clone()` system call. In your implementation of the `procTrhead_join()` API, you will get the thread stack address back via the `join()` system call, and then call `free()` to free up the thread stack.
- Recall that when a system call trap happens, the registers are saved to the trap frame in the calling process's kernel stack. The purpose is to restore the calling process's execution when the system call finishes its execution in the kernel. So in order to ensure the new thread executes from where `clone()` returns, the related registers in the thread's trap frame need to be properly set. More specifically, the ESP and EIP are the CPU registers that need to be properly taken care of.
- Unlike `fork()`, which needs to copy all the user memory for the newly created process, `clone()` does not need to do so as the new thread and the parent thread share almost the entire address space, therefore copying the page table will be sufficient.
- However, the new thread has a separate stack (which is created in `procThread_create()` and pass down to kernel via `clone()`). The stack content of the new thread should be (almost) the same as the stack content of the parent content when `clone()` is called successfully. More details:
  - The main thread is started from the shell by calling `exec()`. You can examine the `exec()` implementation in `exec.c` to understand how the (user) stack of the main thread is created. Also keep in mind that a stack grows from high address to low address.
  - The new thread's stack should have (almost) the same content as the main thread's content after the `clone()` call. You can directly copy the content of the main thread's stack to the new thread's stack. However, a direct copy is not enough,

the "saved EBPs" in the new thread's stack need to be adjusted accordingly, so that both the main thread and the new thread can have there independent call flow (note: this is one of the key points and probably the hardest part of this project). For the correct implementation, you need to be familiar with the stack frame layout in xv6. There are plenty useful resource online. For example: `http://wiki.osdev.org/Stack`.

  - You may notice that in `exec()` when preparing the stack for the main thread, the return PC is set to 0xffffffff (which is why we cannot terminate a process by calling "return" in `main()`, as this would cause a page fault because of the attempt of accessing address 0xffffffff). You can use this fake return PC to get noticed when a thread terminates through returning in `main()`. What you need to do is to handle the page fault with faulting address at 0xffffffff. Also, remember that the return value of a function is recorded in the EAX register.

- You will need to define a new system call named `join()` to perform the join operation. This system call will be called by the `procThread_join()` API to join another thread. The join() system call should look like:

```
void join(int tid, int * ret_p, void ** stack);
```

  - A call of this system call waits for the termination of the thread with thread id of `tid`. If the thread have already terminated, the call returns immediately. Otherwise the calling thread sleeps until the designated thread terminates.
  - Once the `join()` system call returns, the return value of the designated thread is put in to `ret_p`, and the stack of the designated thread is put into `stack` (so that the memory space can be freed in `procThread_join()`).
  - You may refer to `wait()` system call for most details of implementing `join()`.

  The following are more hints of implementing the `join()` system call:

  - Do not assume main thread will always join its child threads. In other words, it is possible that main thread exits before child threads. In the original xv6 implementation, the address space is freed after the process (i.e., the main thread) exists. This will cause problem if the main thread exits before its child threads. Therefore, you will need to implement same some sort of reference counter to track how many threads are sharing the address space. The address space should be freed only after all threads sharing it terminate.
  - The original `wait()` system call implementation should also be changed to deal with the above scenario.

- You will also need to define a system call named `thread_exit()`, which will be called by the `procThread_exit()` API to properly terminate the calling thread. The `thread_exit()` system call should look like:

```
void thread_exit(int ret_val)
```

  The argument `ret_val` is set the return value of the terminating thread, and is passed down from the `procThread_exit()` API.

2.3 Test cases

There are 8 test cases already given in `pt-lib-test.c`.

- TEST1 to TEST4 test threads modifying a global variable. In TEST1, child threads exit before the main thread. In TEST3, the main thread exits before the child threads. TEST2/TEST4 are essentially the same as TEST1/TEST3, except that they call `printf()` to generate some intermediate outputs. Since calling functions involves operations on stacks, correctly setting the stack of the new thread is key to passing the these test cases.
- TEST5 and TEST6 test threads modifying a local variable. In TEST5, the variable is created after the new thread is formed. In TEST6, the local variable is created before the new thread is formed.
- TEST7 test if the handling of thread return/exit value.
- TEST8 test if the stack of the new thread has been properly released.

The expected outputs of all the test cases have been given in the test file. Carefully reading the test code will be helpful for the project.

3 Submit your work

If you use the GitHub website upload function to commit code to the GitHub server, remember to "`make clean`" before uploading. Uploading intermediate files, such as `.o`, `.d` files will lead to points taken (see grading section for details).

A submission link be available on the MyCourses website. **Once your code in your GitHub private repository is ready for grading, submit a file named "DONE", which should include the following info**, to that link:

- Your name and B-number.
- **The SHA1 value (a 40-char string) of the latest commit.** You can obtain it from the GitHub website or from the "`git log`" command. (Note: this is similar to PROJ0.)

Additionally, you can include the following info in this file:

- The status of your implementation (especially, if not fully complete).
- A description of how your code works, if that is not completely clear by reading the code (note that this should not be necessary, ideally your code should be self-documenting).
- Possibly a log of test cases which work and which don't work.
- Any other material you believe is relevant to the grading of your project.

Suggestion: Test your code thoroughly on a CS machine before submitting.

4 Grading

The following are the general grading guidelines for this and all future projects.

(1) **The code in your repository will not be graded until a "DONE" file is submitted to MyCourses**.

(2) The submission time of the "DONE" file shown on the MC system will be used to determine if your submission is on time or to calculate the number of late days. Late penalty is 10% of the points scored for each of the first two days late, and 20% for each of the days thereafter.

(3) Your GitHub username should be the same as your BU username. If your BU username has been used by someone else, contact the instructor for the solution.

If you have multiple GitHub accounts, please make sure you log into the one which has your BU username as the GitHub username before you accept the assignment.

Once you accept the assignment, you can check what account you used to accept the assignment. For example, if your the GitHub account you use to accept the assignment is "abc", the name of your private repository will be "cs550-18s-projX-abc" (with X=0, 1, 2, or 3). If you use a GitHub account whose username is different from you BU username, our script will not be able to locate your repository.

If you made the mistake, you can correct it by logging into the correct GitHub account, and go to the assignment link and accept again.

Any grading issues caused by using incorrect GitHub account will lead to 10 points off.

(4) Uploading intermediate files (e.g., `.o`, `.d` files) will lead to 5 points off ("`make clean`" before uploading can avoid this).

(5) No SHA1 value of the latest commit in the "DONE" file, 5 points off.

(6) If you are to compile and run the xv6 system on the department's remote cluster, remember to use baseline xv6 source code provided by our GitHub classroom. Compiling and running xv6 source code downloaded elsewhere can cause 100% CPU utilization on QEMU.

Removing the patch code from the baseline code will also cause the same problem. So make sure you understand the code before deleting them.

If you are reported by the system administrator to be running QEMU with 100% CPU utilization on QEMU, 10 points off.

(7) If the submitted patch cannot successfully patched to the baseline source code, or the patched code does not compile:

```
1   TA will try to fix the problem (for no more than 3 minutes);
2   if (problem solved)
3     1%-10% points off (based on how complex the fix is, TA's discretion);
4   else
5     TA may contact the student by email or schedule a demo to fix the problem;
6     if (problem solved)
7       11%-20% points off (based on how complex the fix is, TA's discretion);
8     else
9       All points off;
```

So in the case that TA contacts you to fix a problem, please respond to TA's email promptly or show up at the demo appointment on time; otherwise the line 9 above will be effective.

(8) If the code is not working as required in the project spec, the TA should take points based on the assigned full points of the task and the actual problem.

(9) Lastly but not the least, stick to the collaboration policy stated in the syllabus: you may discuss with you fellow students, but code should absolutely be kept private. Any kind of cheating will result in zero point on the project, and further reporting.