Machine Problem 1 - Thread Package

CSIE3310 - Operating Systems
National Taiwan University

Total Points:	100
Release Date:	March 8
Due Date:	March 21, 23:59:00
TA e-mail:	`ntuos@googlegroups.com`
TA hours:	3/10(Thu.) & 3/16(Wed.), 15:30-16:30, CSIE Building R540

Contents

1 MP Description

In this MP, you'll try to implement a user-level thread package with the help of `setjmp` and `longjmp`. The threads explicitly yield when they no longer require CPU time and they won't be interrupted by any other means. When a thread yields or exits, the next thread should run. You'll need to implement the following functions:

- `thread_add_runqueue`
- `thread_yield`
- `dispatch`
- `schedule`
- `thread_exit`
- `thread_start_threading`

The following function has been implemented for you:

- `thread_create`

Each thread should be represented by a `struct thread` which at least contains a function pointer to the thread's function and a pointer of type `void *` as the function parameters. The function of the thread will take the `void *` as its argument when executed. The struct should contain a pointer to its stack and a `jmp_buf` to store it's current state when `thread_yield` is called. It should be enough to use only `setjmp` and `longjmp` to save and restore the context of a thread.

1.1 Function Description

1. `struct thread *thread_create(void (*f)(void *), void *arg)`: This function creates a new thread and allocates the space in stack to the thread. Note, if you would like to allocate a new stack for the thread, it is important that the address of the stack pointer should be divisible by 8. The function returns the initialized structure. If you want to use your own template for creating thread, make sure it works for the provided test cases.

2. `void thread_add_runqueue(struct thread *t)`: This function adds an initialized struct thread into the runqueue. To implement the scheduling functionality, you'll need to maintain a binary tree data structure which is made of `struct thread*`. Each node contains a `*left` pointer, a `*right` pointer, and a `*parent` pointer. You should also maintain the static variable `struct thread *current_thread` that always points to the currently executed thread, and `struct thread *root_thread` that always points to the root thread.

   - If `current_thread`'s left child is empty, make `t` as left child of it.
   - Else if `current_thread`'s right child is empty, make `t` as right child of it.
   - Otherwise, discard `t`, i.e., `t` would not be inserted into the binary tree.

3. `void thread_yield(void)`: This function suspends the current thread by saving its context to the `jmp_buf` in `struct thread` using `setjmp`. The `setjmp` in xv6 is provided to you, therefore you only need to add `#include "user/setjmp.h"` to your code. After saving the context, you should call `schedule()` to determine which thread to run next and then call `dispatch()` to execute the new thread. If the thread is resumed later, `thread_yield()` should return to the calling place in the function.

4. `void dispatch(void)`: This function executes a thread which decided by `schedule()`. In case the thread has never run before, you may need to do some initialization such as moving the stack pointer `sp` to the allocated stack of the thread. The stack pointer `sp` could be accessed and modified using `setjmp` and `longjmp`. Please take a look at `setjmp.h` to understand where the `sp` is stored in `jmp_buf`. If the thread was executed before, restoring the context with `longjmp` is enough. In case the thread's function just returns, the thread needs to be removed from the runqueue and the next one has to be dispatched. The easiest way to do this is to call `thread_exit()`.

5. `void schedule(void)`: This function decides which thread to run next. Please update `current_thread` according to its **preorder traversal**. If it reaches the end, return the root thread as the next one to be executed.

6. `void thread_exit(void)`: This function removes the calling thread from the runqueue, frees its stack and the `struct thread`, updates `current_thread` with the next to-be-executed thread in the runqueue and calls `dispatch()`.

   - If it is a leaf node, simply remove it from the tree.
   - Otherwise, replace the node with the last node in preorder traversal of the subtree rooted at it.

   Furthermore, think about what happens when the last thread exits (should return to the main function by some means).

7. `void thread_start_threading(void)`: This function will be called by the main function after the first thread is added to the runqueue. It should return only if all threads have exited.

1.2 Environment Setup

1. Download the `MP1.zip` from NTUCOOL, unzip it, and enter it.

```
$ unzip MP1.zip
$ cd mp1
```

2. Pull Docker image from Docker Hub.

```
$ docker pull ntuos/mp1
```

3. Use `docker run` to start the process in a container and allocate a TTY for the container process.

```
$ docker run -it -v $(pwd)/xv6:/home/os_mp1/xv6 ntuos/mp1
```

4. You will use the skeleton of `threads.h` and `threads.c` provided in `xv6/user` folder. Make sure you are familiar with the concept of stack frame and stack pointer taught in System Programming. It is also recommended to checkout the appendix given.

1.3 Test Case Specification

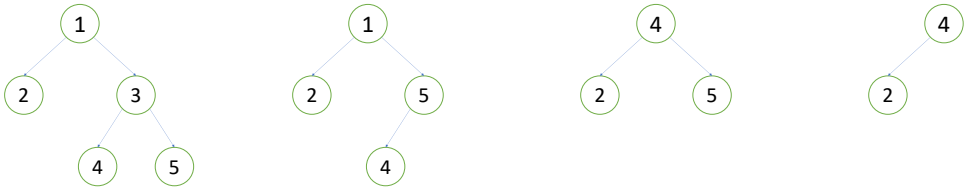
- The maximum depth of the tree is 4 (the root has a depth of 0).
- The main function creates exactly one thread, i.e, the root.

1.4 Sample Output

The output of `mp1-0` should look like the following.

```
$ mp1-0
mp1-0
thread 1: 100
thread 2: 0
thread 3: 10000
thread 1: 101
thread 2: 1
thread 3: 10001
thread 1: 102
thread 2: 2
thread 3: 10002
thread 1: 103
thread 2: 3
thread 3: 10003
thread 1: 104
thread 2: 4
thread 3: 10004
thread 1: 105
thread 2: 5
thread 1: 106
thread 2: 6
thread 1: 107
thread 2: 7
thread 1: 108
thread 2: 8
thread 1: 109
thread 2: 9

exited
```



(a) (b) (c) (d)

Figure 1: Tree structures of public test case `mp1-1`.

1.5 Public Test cases

You can get 40 points (100 points in total) if you pass all public test cases. You can judge the code by running the following command in the docker container (not in xv6; this should run in the same place as `make qemu`). Note that you should only modify `xv6/user/thread.c` and `xv6/user/thread.h`. We do not guarantee that you can get the public points from us if you modify other files to pass all test cases during local testing. We will run `make qemu` to compile your code, but we will not run `make grade` to grade your code.

```
$ make grade
```

If you successfully pass all the public test cases, the output should be similar to the one below.

```
== Test thread package with public testcase 0 (20%) ==
thread package with public testcase 0: OK (12.6s)
== Test thread package with public testcase 1 (20%) ==
thread package with public testcase 1: OK (0.7s)
Score: 40/40
```

If you want to know the details about the test cases, please check `xv6/grade-mp1`, `xv6/user/mp1-0.c` and `xv6/user/mp1-1.c`.

1.6 Note

In `mp1-1` test case:

1. The initial tree structure is Fig 1 (a). Please execute threads one by one according to preorder traversal, which is {`1, 2, 3, 4, 5`}.
2. Thread 3 calls `thread_exit()` first. According to preorder traversal of the subtree rooted at node 3, which is {`3, 4, 5`}, we replace node 3 with node 5 and derive Fig 1 (b). Note that we don't execute thread 5 right after the replacement, i.e., the next thread to be executed should be thread 4.
3. Thread 1 calls `thread_exit()` next. According to preorder traversal of the subtree rooted at node 1, which is {`1, 2, 5, 4`}, we replace node 1 with node 4 and derive Fig 1 (c). Note that we don't execute thread 4 right after the replacement, i.e., the next thread to be executed should be thread 2.
4. Thread 5 calls `thread_exit()` afterwards. Since it is a leaf node, we simply remove it from the tree and derive Fig 1 (d).

2 Submission and Grading

Please compress your xv6 source code as `<whatever>.zip` and upload to NTUCOOL. The filename does not matter since NTUCOOL will rename you submissions. Never compress files we do not request, such as `.o`, `.d`, `.asm` files. You can run `make clean` in the container before you compress. Make sure your xv6 can be compiled by `make qemu`.

2.1 Folder Structure after Unzip

We will unzip your submission by running `unzip *.zip`. Your folder structure **AFTER UNZIP** should be

```
<student_id>
|
+-- threads.c
|
+-- threads.h
```

Note that **the English letters in the `<student_id>` must be lowercase**. E.g., it should be `r10922060` instead of `R10922060`.

2.2 Grading Policy

- There are two public test cases and four private test cases.
  - Public test cases (40%): `mp1-0` and `mp1-1`. 20% each.
  - Private test cases (60%): 15% each.
- You will get a 0 if we cannot compile your submission.
- You will be deducted 10 points if we cannot unzip your file through the command line using the `unzip` command in Linux.
- You will be deducted 10 points if the folder structure is wrong. Using uppercase in the `<student_id>` is also a type of wrong folder structure.
- If your submission is late for n days, your score will be $max(raw\_score - 20 \times \lceil n \rceil, 0)$ points. Note that you will not get any points if $\lceil n \rceil >= 5$.

3 Appendix

[1] Function Pointer. https://en.wikipedia.org/wiki/Function_pointer

[2] Call Stack. https://en.wikipedia.org/wiki/Call_stack