Machine Problem 1 - Thread Package

CSIE3310 - Operating Systems
National Taiwan University

Total Points:	100
Release Date:	March 5
Due Date:	March 19, 23:59:00
TA e-mail:	`ntuos@googlegroups.com`
TA hours:	Wed. & Thu. 10:00-12:00 before the due date, CSIE Building R428

Contents

1 Summary

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. When a thread yields or exits, the next thread should run. The thread can assign additional tasks to other threads, including itself. There are two parts in this MP. In the first part, you'll need to implement the following functions:

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

In the second part, you'll need to implement the following functions:

- `thread_assign_task`

The following function has been implemented for you:

- `thread_create`

Each thread should be represented by a `struct thread` that contains, at a minimum, 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 include a pointer to its stack and some `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.

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 ntuos/mp1
```

4. Execute `xv6`

```
$ make qemu
```

5. 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.

3 Part 1 (60 points)

3.1 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` to the runqueue. To implement the scheduling functionality, you'll need to maintain a circular linked list of `struct thread`. You should implement that by maintaining the `next` and `previous` field in `struct thread` which always points to the next to-be-executed thread and the previously executed thread respectively. You should also maintain the static variable `struct thread *current_thread` that always points to the currently executed thread. Note: Please insert the new thread at the end of the runqueue, i.e. the newly inserted thread should be `current_thread->previous`.

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 will decide which thread to run next. It is actually trivial, since you will just run the next thread in the circular linked list of threads. You can simply change current_thread to the next field of current_thread.

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()`.

   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 some thread is added to the runqueue. It should return only if all threads have exited.

3.2 Sample Output

The output of `mp1-part1-0` should look like the following.

```
$ mp1-part1-0
mp1-part1-0
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
```

4 Part 2 (40 points)

In this part, you are required to implement an additional function `thread_assign_task`. This function enables each thread to manage multiple tasks, with the most recently assigned task being executed first. Note that, child threads should not inherit tasks from their parent when they are created.

4.1 Function Description

1. `void thread_assign_task(struct thread *t, void (*f)(void *), void *arg)`: This function assigns a task to the thread `t`. The second argument, `f`, is a pointer to the task function, while the third argument, `arg`, represents the argument of `f`. If `t` has unfinished tasks, the most recently assigned task will be executed first when `t` is resumed later. The execution of the original thread function must wait until all tasks are finished. Note that, this function only assigns tasks and does not trigger any context switch.

In order to complete this part, you need to modify the following functions:

1. `void thread_yield(void)`: Because this function can also be called in the task function, you should save the context in different `jmp_buf`s according to whether the thread is executing the task function or not. Specifically, if this function is called in the thread function, you can save the context just like in part 1. If this function is called in the task function, you should save the context in another `jmp_buf` to prevent from discarding the context of the thread function.

2. `void dispatch(void)`: If a task is assigned, this function should execute the most recently assigned task function. If this function has never run before, you may need to do some initialization. If this function was executed before, restoring the context with `longjmp` is sufficient. In case this task function just returns, the thread should execute the next task function. The process follows the same approach as with the previous task function. Surely, It is possible for a task to be assigned before the thread executes its thread function.

Feel free to make more modification, such as adding properties in `struct thread`, designing new structure for encapsulating task-related logic, etc. The only requirement is that all defined function should work as described above.

4.2 Reminders

1. When creating a new thread in `thread_create`, ensure that the new thread has no assigned tasks initially.

2. The parameter `struct thread *t` in `thread_assign_task` must exist and not have exited yet.

3. Tasks are executed in Last-Come-First-Serve (LCFS) order. That is, if a thread returns from a task function and there are unfinished tasks, the most recently assigned task will be executed next.

4. While you are encouraged to add properties in `struct thread`, modifying the existing properties is not allowed.

5. The task function may call `thread_create`, `thread_add_runqueue`, `thread_yield`, `thread_exit`, or `thread_assign_task`.

6. The memory space allocated to each thread by `thread_create` is sufficient to execute task functions in all test cases.

7. In all test cases, a thread may have at most 20 unfinished tasks at any moment.

8. If you intend to use global variables in `threads.h`, `threads.c`, or your test files, it is recommended to add the `static` keyword to prevent unexpected situations.

4.3 Sample Output

The output of `mp1-part2-0` should look like the following.

```
$ mp1-part2-0
mp1-part2-0
thread 1: 100
task 2: 101
thread 2: 0
thread 1: 101
thread 2: 1
thread 1: 102
task 2: 103
thread 1: 103
thread 2: 2
thread 1: 104
task 2: 105
thread 2: 3
thread 1: 105
thread 2: 4

exited
```

5 Run Public Test Cases

You can get 35 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.

```
$ 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 part1-0 (10%) ==
thread package with public testcase part1-0: OK (16.8s)
== Test thread package with public testcase part1-1 (10%) ==
thread package with public testcase part1-1: OK (1.3s)
== Test thread package with public testcase part2-0 (5%) ==
thread package with public testcase part2-0: OK (0.8s)
== Test thread package with public testcase part2-1 (5%) ==
thread package with public testcase part2-1: OK (0.9s)
== Test thread package with public testcase part2-2 (5%) ==
thread package with public testcase part2-2: OK (1.0s)
Score: 35/35
```

If you want to know the details about the test cases, please check `xv6/grade-mp1`, `xv6/user/mp1-part1-0.c`, `xv6/user/mp1-part1-1.c`, `xv6/user/mp1-part2-0.c`, `xv6/user/mp1-part2-1.c` and `xv6/user/mp1-part2-2.c`.

6 Submission and Grading

6.1 Source Code

Run the command below to pack your code into a zip named in your **lowercase** student ID, for example, `r11922088.zip`. Upload the zip file to NTUCOOL.

```
$ make STUDENT_ID=<student_id> zip  # set your ID here
```

Please ensure that **your student ID is in lowercase letters**. E.g., it should be `r11922088` instead of `R11922088`. Besides, make sure your xv6 can be compiled by `make qemu`.

6.2 Folder Structure after Unzip

We will unzip your submission using `unzip` command. The unzipped folder structure looks like this.

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

6.3 Grading Policy

- There are 2 public test cases and 4 private test cases in Part 1.
  - Public test cases (20%): `mp1-part1-0` and `mp1-part1-1`. 10% each.
  - Private test cases (40%): 10% each.
- There are 3 public test cases and 4 private test cases in Part 2.
  - Public test cases (15%): `mp1-part2-0`, `mp1-part2-1` and `mp1-part2-2`. 5% each.
  - Private test cases (25%): 5%, 5%, 7.5%, 7.5%.
- You will get 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 $\lceil n \rceil >= 5$.
- Our grading library has a timeout mechanism so that we can handle the submission that will run forever. Currently, the execution time limit is set to 240 seconds. We may extend the execution time limit if we find that such a time limit is not sufficient for programs written correctly. That is, you do not have to worry about the time limit.
- You can submit your work as many times as you want, but only the last submission will be graded. Previous submissions will be ignored.
- The grading will be done on a Linux server.

7 Appendix

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

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

[3] MP1 Slides. link