Machine Problem 1 - Thread Package

CSIE3310 - Operating Systems
National Taiwan University

Total Points:	100
Release Date:	March 4
Due Date:	March 17, 23:59:00
TA e-mail:	`ntuos@googlegroups.com`
TA hours:	Mon. & Wed. 13:00-14:00 before the due date, CSIE Building R439

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 parent thread can send a signal to their children to kill them or trigger their signal handlers. 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_suspend`

- `thread_resume`
- `thread_kill`

The following function has been implemented for you:

- `thread_create`
- `get_thread`
- `thread_register_handler`

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 two `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/os_mp1/xv6 ntuos/mp1
```

4. Enter `os_mp1/xv6` and execute `xv6`

```
$ cd os_mp1/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 the first thread is added to the runqueue. It should return only if all threads have exited.

3.2 Test Case Specification

- The main function creates exactly one thread, i.e, the root.

3.3 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 need to implement two additional functions related to signal generation and handling. Each thread can install different signal handlers to different signals. Child threads should inherit signal handlers from their parent when they are created. Note that, threads have independent signal handlers, i.e. Changing any signal handler of one thread doesn't affect signal handlers of other threads.

4.1 Function Description

1. `void thread_suspend(structure thread *t)`: This function suspends the execution of the current thread. When called, the thread's state is saved, and it is placed into a "suspended" state, meaning it will `no longer be scheduled or dispatched until resumed`. If the thread suspended itself, you may need to call thread_yield(). This is useful for temporarily pausing a thread's work without terminating it, especially when a certain resource is not available or when waiting for an external condition to be met.

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

   (a) `void schedule(void)`: Modify the scheduling logic to ensure that it does not select a suspended thread for execution.

   (b) `void dispatch(void)` Before dispatching a new thread, ensure that the selected thread is not suspended. If the next thread in the runqueue is suspended, continue searching for an active thread.

2. `void thread_resume(structure thread *t)`: This function resumes the execution of a suspended thread (*t). When called, the suspended thread can be scheduled again by the scheduler. This is useful when a thread was suspended due to waiting for a resource or an external condition, and that condition has now been met.

3. `void thread_register_handler(int signo, void (handler*)(int))`: This function receives two arguments: first argument as an integer which represents signal number and second argument as a pointer to the signal-handling function. This function sets `handler` to `current_thread` for signal `signo`. If `current_thread` has registered a signal handler corresponding to this signal, just replace it.

4. `void thread_kill(struct thread *t, int signo)`: This function sends a signal `signo` to `t` but **does not** trigger an immediate context switch. If `t` is the current thread and `t` has a corresponding signal handler for the signal `signo`, the handler function will be executed first when `t` is resumed later. Otherwise, `t` will be killed, that is, `thread_exit()` is called directly when `t` is resumed.

   (a) If a thread returns from its signal handler, it should continue executing the original thread function from the place where it was interrupted by the signal.

   (b) This function only sends a signal and does not trigger any context switch.

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

   (a) `void thread_add_runqueue(struct thread *t)`: In this function, you should let the child thread `t` inherit the signal handlers from `current_thread`.

   (b) `void thread_yield(void)`: Because this function can also be called in the signal handler, you should save the context in different `jmp_buf`s according to whether the thread is executing the signal handler 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 signal handler, you should save the context in another `jmp_buf` to prevent from discarding the context of the thread function.

   (c) `void dispatch(void)`: If a signal came, this function will call the corresponding signal handler. However, if such signal handler has not been registered, then `current_thread` should be killed, i.e. calling `thread_exit()` directly. In case the handler has never run before, you may need to do some initialization. If the signal handler was executed before, restoring the context with `longjmp` is enough. In case the signal handler just returns, the thread should continue executing the original thread function from the place where it was interrupted by the signal. Surely, it is possible that a signal comes before the thread executes its thread function.

4.2 Test Case Specification

(a) There are two types of signals, 0 and 1 respectively.

(b) The parameters of `thread_kill` are legal, such as `struct thread *t` has not exited yet and `signo` must be 0 or 1.

(c) The parameter `struct thread *t` in `thread_kill` must be a child thread of `current_thread`.

(d) Signals will only be sent from a parent thread to its child threads. A parent thread will not sent signals to the same child thread more than once.

(e) Only `thread_yield()` and `thread_exit()` will be called in signal handlers. Both `thread_create()` and `thread_add_runqueue()` won't be called in signal handlers.

(f) `thread_register_handler()` and `thread_kill()` won't be called in the main thread.

(g) For each testing data in part 2, we make `thread_1` suspend for part2-0 after `thread_1` output half of the target value, `thread_2` suspend for part2-1 after output half of the target value and so on.

4.3 Sample Output

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

```
$ mp1-part2-0
mp1-part2-0
thread 1: 100
handler 3: 20
thread 1: 101
handler 3: 22
thread 1: 102
handler 3: 24
thread 3: 10000
thread 1: 103
thread 1: suspending
thread 3: 10001
thread 3: 10002
thread 3: 10003
thread 3: 10004
thread 3: 10005
thread 1: resuming
thread 1: 104
thread 1: 105

exited
```

4.4 Note

In mp1-part2-0 test case:

(a) Because the signal handler for `No.0 signal` has not been registered when `thread 2` is dispatched, `thread 2` should be killed directly.

(b) Because `thread 3` inherited the signal handler for `No.1 signal` from its parent, it should execute such signal handler when `thread 3` is dispatched.

(c) Because `thread 3` returns from its signal handler, it should continue executing its thread function.

(d) `Thread 1` will suspend itself and after `thread 3` finished its output, `thread 3` will resume `thread 1`.

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 (2.2s)
== Test thread package with public testcase part1-1 (10%) ==
thread package with public testcase part1-1: OK (0.6s)
== Test thread package with public testcase part1-2 (10%) ==
thread package with public testcase part1-2: OK (1.6s)
== Test thread package with public testcase part1-3 (10%) ==
thread package with public testcase part1-3: OK (1.0s)
== Test thread package with public testcase part2-0 (10%) ==
thread package with public testcase part2-0: OK (1.0s)
== Test thread package with public testcase part2-1 (10%) ==
thread package with public testcase part2-1: OK (1.0s)
Score: 60/60
```

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-part1-2.c`, `xv6/user/mp1-part2-0.c` and `xv6/user/mp1-part2-1.c`.

6 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`.

6.1 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 `r13944062` instead of `R13944062`.

6.2 Grading Policy

- There are 3 public test cases and 3 private test cases in Part 1.
  - Public test cases (30%): `mp1-part1-0`, `mp1-part1-1`, and `mp1-part1-2`. 10% each.
  - Private test cases (30%): 10% each.
- There are 2 public test cases and 2 private test cases in Part 2.

  - Public test cases (20%): `mp1-part2-0` and `mp1-part2-1`. 10% each.
  - Private test cases (20%): 10% each.
- 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