Operating System 2021 Machine Problem Site

The site provides the information for machine problems for Operating System Course offered in Spring 2021 at National Taiwan University

MP1 Thread Package (Due March 30 at 09:00)

> This website focuses on details of programming homework. For handwriting part, please download the questions from NTU COOL and submit the answers via Gradescope.

In this programming homework, you'll try to create a user-level thread package with the help of `setjmp` and `longjmp`.

1. Description

In this homework, you need to implement a **user-level** thread package. 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. To choose the next thread to run, simply keep the threads in a circular linked list and pick the next one.

You'll need to implement the following functions:

- thread_add_runqueue
- thread_yield
- thread_exit
- dispatch
- thread_start_threading

The following functions has been implemented for you:

- thread_create
- schedule

Each threads 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 it's 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. `struct thread *thread_create(void (*f)(void *), void *arg)`

   Input:

   - a funtion pointer `*thread_create(void (*f)(void *)`
   - arguments of the function `void* arg`

   Return:

   - new thread `struct thread`

   This function should create a new thread and allocate 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 divisable by 8.[1] 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 case.**

2. `void thread_add_runqueue(struct thread *t)`

   Input:

   - a thread pointer `struct thread *t`

   Return:

   - None

   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)`

   Input:

   - None

Return:

- None

This function suspends the current thread by saving its context to the `jmp_buf` in `struct thread` using `setjmp`.[2] The `setjmp` in xv6 is provided to you, you only need to add `#include "user/setjmp.h"` to your code. After saving the context, you should call `schedule` function 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 thread_exit(void)`

   Input:

   - None

   Return:

   - None

   This function removes the calling thread from the runqueue, frees its stack and the `struct thread`, updates the `current_thread` variable 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 main by some means).

5. `void schedule(void)`

   Input:

   - None

   Return:

   - None

   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 dispatch(void)`

   Input:

   - None

   Return:

   - None

   This function will execute a thread, 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 calling `thread_exit`.
7. `void thread_start_threading(void)`

   Input:

   - None

   Return:

   - None

   This function will initialize the threading by calling `schedule` and `dispatch`. 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 exit.

> The skeleton of `threads.c` is provided to you. Please complete the functions in `threads.c`.

2. Environment setup

1. Clone the repo from Github Classroom
   - Link
2. Pull docker image from Docker Hub

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

3. Enter the repo and you will see a folder named `xv6-riscv`, then run the following command to mount the `xv6-riscv` folder to docker container.

```
$ docker run -u $(id -u):$(id -g) -it -v $(pwd)/xv6-riscv:/home/ ntuos/mp1 bash
```

> After entering the container, you may find your user name in container be "I have no name!" or others.

> Please ignore this issue, this doesn't affect your programming task.
> For windows users, please open docker desktop and **run the command in WSL**

You will use the skeleton of `threads.h` and `threads.c` provided in `xv6-riscv/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. Before Compiling

Uncomment line 145-147 in `Makefile` to compile `mp1.c` and `threads.c`

4. Sample Execution

We provide you a public test case `mp1.c` to test out your thread package. It includes a main function and some thread functions.

For grading, we will be using other 3 sets of private test cases. It is recommended to add extra cases to ensure the correctness of your programs.

The output of `mp1.c` should look like the following:

```
$ make qemu
...
init: starting sh
$ mp1
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
$
```

5. Grading

- Programming: 70%
  There are 4 test cases: 1 public and 3 private
  1. Public test case `mp1.c` : 25%
  2. Private test cases: 15% each
- Handwriting: 30%
  There are three questions in handwritting assignment.
  - For some questions, writing a simple code may help

6. Submission

Programming

Push your `xv6-riscv` source code to GitHub. ~~**Never push any other files we do not request**, such as `.o`, `.d`, `.asm` files.~~ You can run `make clean` in container before you push. Make sure your `xv6-riscv` can be compiled and the thread package files, i.e., `threads.c` and `threads.h` are included.

```
Repository
└── xv6-riscv
    ├── user
    │   ├── threads.c
    │   ├── threads.h
    │   └── ...
    ├── Makefile
    └── ...
```

- You will get **0** if ~~`xv6-riscv`~~ **[Update 3/18] thread.c and thread.h** cannot be compiled.
- ~~We might have discount on your grade if your format is wrong, such as pushing `.o` files.~~

[Update 3/18] You can create new files in your repo, and your grade won't be affected. Just make sure to include thread.c and thread.h

Handwriting

- Use the entry code `86KX77` to sign up Gradescope: Link
- Use **traditional Chinese characters** for your name and use **upper-cases** for your student ID.

7. Appendix

Here are some references that might come in handy.

Function Pointer (Must know)

Function Pointer - Wiki.

Call Stack

Call Stack - Wiki.

x86_64 inline assembly (gcc)

Asm Documentation.

- Access stack pointer (placed into `stack_p`):

```
unsigned long stack_p = 0;
asm("mov %%rsp, %0;"
    : "=r" (stack_p));
```

- Write stack pointer (value from `new_stack_p`):

```
unsigned long new_stack_p;
asm("mov %0, %%rsp;"
         :
         : "r" (new_stack_p));
```

8. Debugging

It is recommended to first use gdb (with enhancement such as gdb-gef) to debug the program on x86 machine.

9. Footnote

1. The reason why we need the address of the stack pointer to be divisable by 8 is the architecture of xv6 is rv64 (RISC V 64 bits). The 64-bit architecture has a memory layout aligned by 8 bytes. ↩

2. `setjmp` sets up the local jmp_buf buffer and initializes it for the jump. This routine saves the program's calling environment in the environment buffer specified by the env argument for later use by longjmp. If the return is from a direct invocation, setjmp returns 0. If the return is from a call to longjmp, setjmp returns a nonzero value. Wiki-setjmp.h ↩
