

Programming Assignment 3 - User-Level Thread Library



Deadline: Fri 9 Dec 23:59:59 CST 2022

Github Classroom Link (<https://classroom.github.com/a/Zn5bAd75>).

Video Link (<https://www.youtube.com/watch?v=KmZqkBoFEYU>).

Discussion Space Link (https://github.com/NTU-SP/SP_HW3_release/discussions).

Blue correction: 17:20 at Nov 27

Green correction: 00:00 at Dec 1

Problem Description

In this assignment, we are going to implement a toy example of a user-level thread library. Additionally, we will simulate asynchronous file I/O.

More specifically, there are going to be multiple user-defined threads running simultaneously. Each thread is in charge of solving a computational problem. To the kernel, there is only one process running, so there will never be two threads running at the same time. Instead, you have to use `setjmp` and `longjmp` to context switch between these threads, and make sure they each have a fair share of execution time. When a thread needs to `read` from an external file, it should be put to sleep and let other threads execute. Otherwise, the whole process would have to pause until the call to `read` returns.

We define three possible states for each thread:

- **Running.** The thread is running and occupying the CPU resource.
- **Ready.** The thread has all the input it needs at the moment, and it's waiting for the CPU resource.
- **Waiting.** The thread is waiting for specific file input.

And we maintain two queues: a **ready queue** and a **waiting queue**. The ready queue stores the **running** and **ready** threads, while the waiting queue stores the **waiting** threads. We will then write a scheduler to manage those queues.


Overview

Here is an overview of how each file in the repository work. More detailed explanations are in the next section.

- `main.c` initializes the data structures needed by the thread library, creates the threads, then hands over to `scheduler.c`
- `scheduler.c` consists of a signal handler and a scheduler.
 - In this assignment, we repurpose two signals to help us perform context-switching. `SIGTSTP` can be triggered by pressing `Ctrl+Z` on your keyboard, and `SIGALRM` is triggered by the `alarm` syscall.
 - The scheduler maintains the waiting queue and the ready queue. Each time the scheduler is triggered, it iterates through the waiting queue to check if any of the requested data are ready. Afterward, it brings all available threads into the ready queue.
 - There are two reasons for a thread to leave the ready queue:
 - The thread has finished executing.
 - The thread wants to read from a file, whether or not the data is ready. In this case, this thread is moved to the waiting queue.
- `threads.c` defines the threads that are going to run in parallel. The lifecycle of a thread should be:
 - Created by calling `thread_create` in `main.c`.
 - Calls `thread_setup` to initialize itself.
 - When a small portion of computation is done, call `thread_yield` to check if a context switch is needed.
 - When it needs to read from a file, call `async_read` to gather the data.
 - After all computations are done, call `thread_exit` to clean up.

File structure

main.c

 **Warning:** We will use the default version of `main.c` for grading. Your modifications to this file will be discarded.

All of the functions defined here are completed. **Understanding how they work is not required**, but you're highly encouraged to do so.

`main(argc, argv)`

To run the executable, you have to pass in 4 arguments:

```
./main [timeslice] [fib_arg] [col_arg] [sub_arg]
```

Where `timeslice` is the number of seconds a thread can execute before context switching.

`fib_arg`, `col_arg`, and `sub_arg` are the arguments passed into the 3 thread functions specified later.

`unbuffered_io()`

This function turns `stdin`, `stdout`, and `stderr` into unbuffered I/O.

`init_signal()`

This function initializes two signal masks, `base_mask`, `tstp_mask`, and `alarm_mask`. `base_mask` blocks both `SIGTSTP` and `SIGALRM`, `tstp_mask` only blocks `SIGTSTP`, and `alarm_mask` only blocks `SIGALRM`.

Additionally, it sets `sighandler` as the signal handler for `SIGTSTP` and `SIGALRM`, then blocks both of them.

`init_threads(fib_arg, col_arg, sub_arg)`

This function creates the user-level threads with the given arguments. If an argument is negative, the respective thread will not be created.

`start_threading()`

This function sets up the alarm, then calls the scheduler.

`threadtools.h`

This file contains the definitions of some variables and structures. It may be important to know what they are, but modifications to them are not required.

`struct tcb`

The thread control block (TCB) serves as the storage of per-thread metadata and variables. You are allowed to add more variables if needed.

`struct tcb *ready_queue[], struct tcb *waiting_queue[]`

The ready queue and the waiting queue are defined as arrays for simplicity. Upon the initialization of new threads, a TCB structure should be allocated and appended to the ready queue.

There are 5 defined macros **you have to complete**:

`thread_create(func, id, arg)`

Call the function `func` and pass in the arguments `id` and `arg`. We guarantee that the maximum number of threads created is 16, and no two threads share the same ID.

thread_setup(id, arg)

Initialize the thread control block and append it to the ready queue. This macro creates and opens a named pipe, whose name should follow this format:

```
[thread id]_[function name]
```

This macro should also call `setjmp` so the scheduler knows where to `longjump` when it decides to run the thread. more about the scheduler is [specified later](#). Afterwards, it should return the control to `main.c`.

Note:

- The call to `open` should return immediately. i.e. The named pipes should be opened in nonblocking mode.
- You don't have to handle the case where a file with the same name as the named pipe already exists.

thread_yield()

Every computational problem takes several iterations to finish. After each iteration, a thread should use this macro to check if there's a need to let another thread execute.

This macro should save the execution context, then, **sequentially** unblock SIGTSTP and SIGALRM. If any of the two signals are pending, the signal handler will take over and run the scheduler. If there aren't any signals pending, this macro should block the signals again, then continue executing the current thread.

async_read(count)

Save the execution context, then jump to the scheduler with `longjmp(sched_buf, 2)`. After the thread restarts, call `read` to receive `count` bytes from the named pipe. We make the following guarantees:

- `count` is less than 512.
- Only one other process will open the FIFO, and it only opens the file for writing.
- The other process that opens the FIFO will not close it until all inputs are written.
- `read` will not return a smaller number than `count`, as long as `count` is reasonable.

thread_exit()

Remove the named pipe owned by the thread, then jump to the scheduler with `longjmp(sched_buf, 3)`.

threads.c

This is where the thread functions are defined. The prototype of a function should always be:

```
void thread_function(int id, int arg)
```

Where `id` is the thread ID, and `arg` is the argument passed in.

You should see 3 functions, and one of them is already implemented for you.

Notice:

- A thread function should always call `thread_setup(id, arg)` in the first line.
- All tasks should be implemented **iteratively** instead of recursively.
- Each iteration should take a bit more than 1 second. To guarantee this, call `sleep(1)` after you've done some computations.
- The variables you declare in the stack will be overwritten after context switching. If you wish to keep them, store them in the TCB.
- Both SIGTSTP and SIGALRM should be blocked when the threads are executing.
- **The arguments passed into the threads will follow the restrictions of each algorithm.**

Fibonacci number

A positive integer `n` is passed in as the argument. Your program has to calculate the `n`th Fibonacci number. Specifically,

$$FIB(n) = \begin{cases} 1 & \text{if } n \leq 2 \\ FIB(n-1) + FIB(n-2) & \text{otherwise} \end{cases}$$

For the `i`th iteration, you should print out a line **to the standard output**:

```
[thread id] [FIB(i)]
```

Collatz's conjecture

An integer `n` larger than 1 is passed in as the argument. For each iteration, your program should perform one of the following operations:

- If `n` is even, divide it by 2.
- If `n` is odd, multiply it by 3, then add 1 to it.

Then, it should print out a line **to the standard output**:

```
[thread id] [N]
```

The procedure should stop when N is 1.

Maximum subarray


You should expect N integers written into the FIFO, where N , a positive integer, is passed as the argument. The integers are always 4 characters long, padded with spaces, and followed by a line break. For example,

```
1
0
8848
-5
236
4
```

represents the array `[1, 0, 8848, -5, 236, 4]`.

For each iteration, your program reads an integer, then prints out a line **to the standard output**:


```
[thread id] [max subarray sum so far]
```

 **Note:** A subarray of length 0 is also allowed.

scheduler.c

This file contains the functions that determine the order of execution. There are two functions you have to complete:

sighandler

 **Attention!** This function should only be triggered as a result of `thread_yield`. Under no circumstances should you call this function explicitly in your code.

This function should be set as the signal handler for SIGTSTP and SIGALRM in main.c. Upon executing this function, it should print one of the following lines **to the standard output**:

```
caught SIGTSTP
caught SIGALRM
```

If the signal caught is SIGALRM, you should reset the alarm here.
Then, you should jump to the scheduler with `longjmp(sched_buf, 1)`.

scheduler

There are 4 reasons to jump here:

- called by `main.c`.
- `longjmp(sched_buf, 1)` from `sighandler` triggered by `thread_yield`
- `longjmp(sched_buf, 2)` from `async_read`
- `longjmp(sched_buf, 3)` from `thread_exit`

For the first case, this function should execute the earliest created thread. Otherwise, it should perform the following tasks in order:

- Bring every ready thread from the waiting queue to the ready queue. They should be appended to the ready queue while maintaining their relative order. Then, the holes left in the waiting queue should be filled, while keeping the original relative order.
- Remove the current thread from the ready queue if needed. There are two cases:
 - For `async_read`, move the thread to the end of the waiting queue.
 - For `thread_exit`, clean up the data structures you allocated for this thread, then remove it from the ready queue.
- If you have removed a thread in the previous step, take the thread from the end of the ready queue to fill up the hole.
- Switch to the next thread. There are three cases:
 - For `thread_yield`, you should execute the next thread in the queue.
 - For `async_read` and `thread_exit`, you should execute the thread you used to fill up the hole.
 - If the thread calling `thread_yield`, `thread_exit` or `async_read` is the last thread in queue, execute the first thread.
- When the ready queue is empty, there are two cases:
 - The waiting queue is not empty: you have to wait until one of the threads is ready. If multiple threads are ready, bring all of them in the ready queue.
 - The waiting queue is also empty: return to `main`.

Execution

You can compile the source code or clean up by using the `make` command. If your implementation include additional files, please add them to the `makefile`.



Warning: This assignment only runs on x86_64 Linux machines. Compatibility with other architectures / OSes is not guaranteed.



Note: We use the notations below to represent messages that don't come from the program:

- `// [some text] : comments`
- `^z : SIGTSTP delivered`

Sample execution 1:

```
$ ./main 3 6 40 -1
0 1
0 1
0 2
caught SIGALRM
1 20
1 10
1 5
caught SIGALRM
0 3
0 5
0 8 // thread 0 exits, SIGALRM pending
1 16
caught SIGALRM
1 8
1 4
1 2
caught SIGALRM
1 1
```

Sample execution 2:


```

$ ./main 3 5 3 -1
0 1
0 1
^Zcaught SIGTSTP
1 10
caught SIGALRM
0 2
0 3
0 5 // thread 0 exits, SIGALRM pending
1 5
^Zcaught SIGTSTP // SIGALRM still pending
1 16
caught SIGALRM
1 8
1 4
1 2
caught SIGALRM
1 1

```

Sample execution 3:

```

$ ./main 3 4 -1 2
0 1
^Zcaught SIGTSTP
0 1 // thread 2 leaves the ready queue immediately, so only thread 0 is ready
0 2 // thread 2 receives ' -1\n' right after this line
caught SIGALRM
2 0
0 3 // thread 0 exits, thread 2 does not receive anything, program hangs
2 99 // thread 2 eventually receives ' 99\n'

```

Additional questions

You don't need to answer the following questions. They're perhaps not the main focus of this course, and you won't get any additional points for knowing the answer. However, we figured they may be interesting:

- It may seem weird that we define `thread_create`, `thread_setup`, `thread_yield`, `async_read`, and `thread_exit` as macros. What will happen if you define them as functions instead? Why? What if they're inline functions?
- We implemented the waiting queue and the ready queue with arrays for simplicity. Which data structures will you use alternatively? Preferably, it should be both efficient and scalable.
- You will learn about pthreads later in this course. Why can pthreads allow storing variables in the stack, while our threads have to store them in the TCB? There's a syscall you will learn later that can help fix this problem. Which one is it?

- Some programming languages already support asynchronous functions. They are why we came up with this assignment. Do you know the behaviors of any of them?
- The context switch overhead in our implementation takes $O(n)$ time, where n is the size of the waiting queue. Can you improve the performance, at least in the userspace? On Linux, `inotify` might help you. On BSD, we're not sure if it's doable.

Grading

Warning:

- Please strictly follow the implementation guidelines, or TAs' programs may not work successfully with yours and you will lose points.
- If your submission includes unnecessary files, your grade will be capped at 6.
- Please make sure your submission compiles and runs on CSIE workstations successfully.
- Please strictly follow the output format, or you may not get the full credits.
- Please use `ps -u [student id]` or `htop -u [student id]` to check for dangling processes. To clean up all your processes on a workstation, please execute `pkill -9 -u [student id]`.

We will grade your submission on linux1, but you are encouraged to write and debug your programs on linux2 ~ linux15. Please refer to the Workstation Monitor (<https://monitor.csie.ntu.edu.tw>) to avoid overloaded machines.


- (6.5pt) Your `threadtools.h` and `scheduler.c` work smoothly without `async_read`.
 - The TAs will use their version of `main.c` and `threads.c` for this subtask.
 - (2.5pt) Without `async_read`
 - 1-1 (0.5pt) Your program handles `thread_exit` correctly.
 - 1-2 (0.5pt) Your program performs a context switch after each timeslice.
 - 1-3 (0.5pt) Your program performs a context switch after receiving `SIGTSTP`.
 - 1-4 (1pt) The order of execution is correct.
 - (4pt) With `async_read`
 - 2-1 (0.5pt) The name and the lifecycle of FIFOs are correct.
 - 2-2 (1pt) A thread enters the ready queue right after receiving input.
 - 2-3 (1pt) `async_read` receives the correct data.
 - 2-4 (1pt) The ready queue and the waiting queue are maintained correctly.
 - 2-5 (0.5pt) The scheduler works correctly for an empty ready queue and a nonempty waiting queue.
- (1.5pt) The three functions in `threads.c` are implemented correctly. For each function,
 - 3-1 ~ 3-3 (0.3pt) The last line of output is correct for each input.

- 4-1 ~ 4-3 (0.2pt) Every line of output is correct.

Submission

Your source code should be submitted to GitHub before deadline. The submission should include at least these files:

• Makefile
Programming Assignment 3 - User-...

 [HackMD \(https://hackmd.io?utm_source=view-page&utm_medium=logo-nav\)](https://hackmd.io?utm_source=view-page&utm_medium=logo-nav)

- main.c
- scheduler.c
- threads.c
- threadtools.h

You are allowed to add other files, however, your submission should not include the files below:

- ELF files, such as executables
- FIFOs

The TAs will clone your repository and test your latest commit on the main branch. We will test your code on `linux1.csie.ntu.edu.tw`. Please make sure your codes work as expected on this server.

Github Classrom seems to have a problem where we can't record the submission time of each commit. Therefore, we have to clone your github repo manually after the deadlines. **You should not do any operation on your github repo at 16 Dec 2022 00:00 ~ 01:00.** Otherwise, we cannot guarantee the correctness of the cloned repo.

Reminders

- Plagiarism is STRICTLY prohibited.
- **Early policy**
 - **If you submit your assignment before 12/2 23:59, you will receive a 1-point bonus.**
 - **You will receive at most 8 points after the bonus.**
- Late policy (D refers to the formal deadline, **12/9** 23:59)
 - If you submit your assignment on D+1 or D+2, your score will be multiplied by 0.85.
 - If you submit your assignment between D+3 and D+5, your score will be multiplied by 0.7.
 - If you submit your assignment between D+6 and **12/16**, your score will be multiplied by 0.5.
 - Late submission after **12/16** 23:59 will not be accepted.
- We will judge your submissions after 12/16. Bonuses and penalties are based on your last submission.

Where to Ask Questions

1. Evoke QAs in GitHub discussion NTU-SP/SP_HW3_release (https://github.com/NTU-SP/SP_HW3_release)

- How to open a GitHub Discussion (<https://youtu.be/rgCMmg9IBNw?t=210>).

2. Send emails to `ntusp2022@gmail.com`.

- Before sending mails, please make sure that you have **read the SPEC carefully**, and **no similar questions were asked** in the discussion space, or TAs will be very tired and upset.



- **How to send an Email properly**

- Mail title should be like: [SP Programming HW3] TA hour reservation
- It's better to use your `@ntu.edu.tw`, `@g.ntu.edu.tw`, `@csie.ntu.edu.tw` email accounts to send the mail.
- You should write down your student ID, and name at the end of the mail.
- Please mind your manners.
- Use punctuation marks to make your email easier to read.
- If you want to ask questions about code in the mail, please tell us how to reproduce your problem. Attach a screenshot, a video recording, or your code file will. It will be easier for us to understand your situation.
- Below is an example format:

Title: [SP Programming HW3] TA hour reservation

Sent By: `r11952025@csie.ntu.edu.tw`

Contents:

Dear SP TAs,

Can I have a TA hour appointment at 13:00 - 14:00 Tuesday?

I encounter some problems with `setjmp/longjmp`. Thank you!

Best,

R11952025 資工碩一 王大明

- If you don't follow the upper rules, we may ignore your email. (Most of you guys are good, though. 😊)

3. Ask during TA hours

- Below are the TA hours for this Programming Assignment. Please make appointments by emailing `ntusp2022@gmail.com` before visiting.
 - Tuesday 13:00 - 14:00 at Room 404, CSIE building
 - Wednesday 16:00 - 17:00 at Room 302, CSIE building
 - Friday 15:00 - 16:00 at Room 404, CSIE building