

Programming Assignment 3 - User-Level Thread Library



Deadline: Thu 7 Dec 23:59:59 CST 2023

Github Classroom Link (<https://classroom.github.com/a/2KNweHRO>)

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

Discussion Space Link (<https://discord.com/channels/1148625792083968131/1148626066819268671>)

Frequently Asked Questions ([/be53dHQbRQmgWZAr5RhxpA](#))

Upd 11/23 Description about `bank_operation`

Upd 11/24 We will use our `bank.h` for grading. Please refer to the Grading section.

Upd 11/28 Fixed typo `acquire -> acquire`. Please note that this typo is also present in the sample output.

Upd 11/29 Please refer to `lock()` and `scheduler` section.

Upd 11/29 `thread_exit()` ~~clean up the resources~~

Upd 12/01 Please refer to grading

Upd 12/05 Please refer to grading



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

Problem Description

In this assignment, we are going to implement a toy example of a user-level thread library. Additionally, we will implement a simple mutex lock, and use the lock to protect a bank system.

More specifically, there are going to be multiple user-defined threads running simultaneously. Each thread is in charge of running a specific job. 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 acquire a lock that is held by another thread, it should be put to sleep and let other threads execute.

We define three possible states for each thread:

- **Running.** The thread is running and occupying the CPU resource.
- **Ready.** The thread is waiting for the CPU resource.
- **Waiting.** The thread is awaiting lock acquisition.

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 decides which thread to run next, and brings an available thread from the waiting queue to the ready queue.
- There are two reasons for a thread to leave the ready queue:
 - The thread has finished executing.
 - The thread wants to acquire the lock, but the lock is held by another thread.
- `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.
 - Call `lock()` to acquire a lock
 - Call `unlock()` to release a lock
 - 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] [fact_arg] [bank1_arg] [bank2_arg]
```

Where `timeslice` is the number of seconds a thread can execute before context switching. `fib_arg`, `fact_arg`, `bank1_arg`, and `bank2_arg` are the arguments passed into the 4 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(int fib_arg, int fact_arg, int bank1_arg, int bank2_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.

init_bank()

This function is supposed to initialize the bank structure with a balance of 500 and a lock owner of -1.

bank.h

You are not required to modify this file.

In this task, you are going to maintain a simple bank system.

You can find the definition of the structure in `bank.h`

```
struct Bank {
    int balance;
    int lock_owner;
};
```

- `balance` indicates the balance of the user.
- `lock_owner` records the **thread id** that is currently holding the lock. If the lock is available (not held by any thread) , it is set to `-1` .

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.

Note that these are implemented as **MACRO**.

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 6 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 also print a line **to the standard output**:

```
[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` .

thread_yield()

Most computational problem takes several iterations to finish (except Bank Operation has got only 3 steps). After each iteration (step), 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.

lock()

Check if the lock is available, if so, acquire the lock and continue.

Otherwise, if the lock is held by another thread, save the execution context, then jump to the scheduler with `longjmp(sched_buf, 2)`.

Upd 11/29:

After that, this thread should be put to waiting queue by the scheduler.

When the lock is available, it will automatically acquire the lock and continue its execution. This is done by the scheduler, and we will talk about this more in the `scheduler` section.

unlock()

Release the lock, **only the owner of the lock can release it**.

thread_exit()

Jump to the scheduler with `longjmp(sched_buf, 3)`.

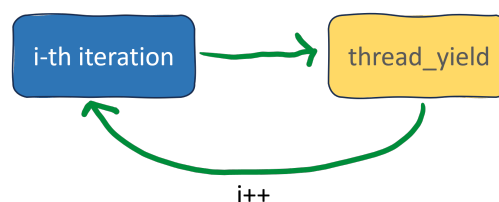
threads.c

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 **Fibonacci number** is already implemented for you. **Fibonacci number** and **Factorial** should call `thread_yield` after each iteration as shown below:



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 every iteration (step).
- 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,

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

Note that you are asked to compute the answer in exact n iterations

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

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

You don't need to handle overflow when using `int32_t`.

Factorial

A positive integer n is passed in as the argument.

You have to compute the factorial of n

Note that you are asked to compute the answer in exact n iterations

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

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

You don't need to consider overflow when using `int32_t`.

Bank Operation

The argument n indicate a operation of a user, $abs(N) > 0$.

You need to perform three steps in order, and you need to call the `thread_yield` at the end of each step.



1. (Step 1) Acquire lock

At this step, you call `lock()` to acquire the lock. **After the lock is acquired**, print the following line **to the standard output**:

```
[thread id] acquired the lock
```

2. (Step 2) Check the balance and update the balance

You can only modify the balance at this step, after acquiring the lock.

- If N is positive, deposit N .
- If N is negative and $\text{balance} \geq \text{abs}(N)$, withdraw $\text{abs}(N)$.

After that, print a line **to the standard output**:

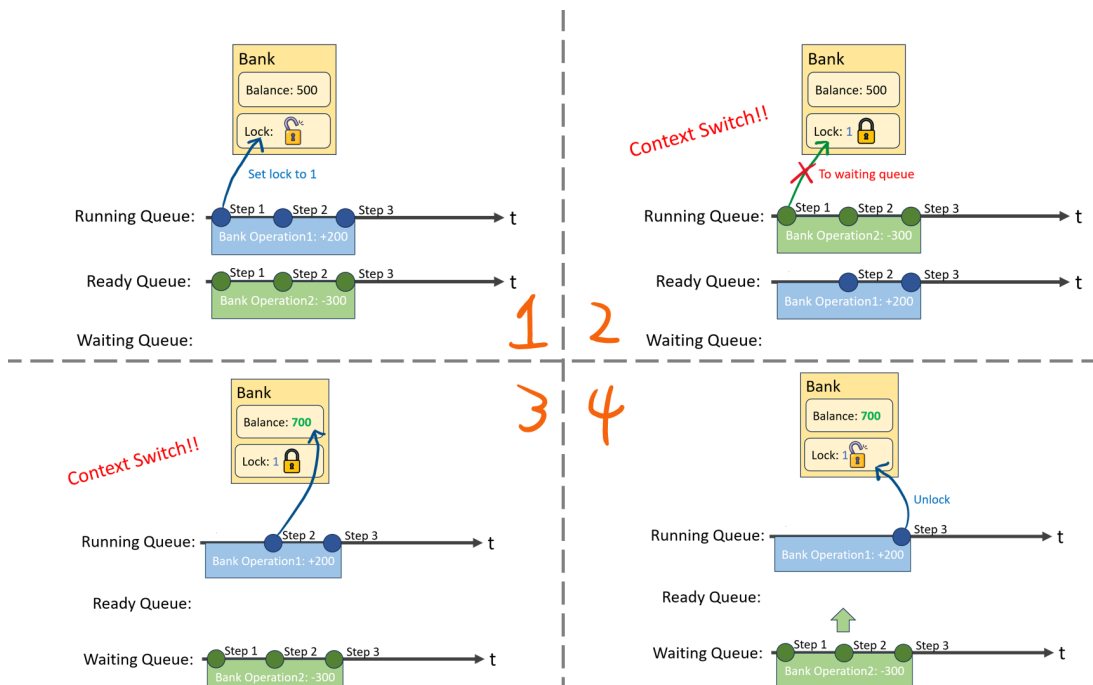
```
[thread id] [balance] [updated balance]
```

3. (Step 3) Release the lock

Call `unlock()` to release the lock. And print the following line **to the standard output**:

```
[thread id] released the lock
```

Example



State 1: The system is running a thread (Bank Operation 1) and has a balance of 500. Bank Operation 1 sets the lock on the Bank to ensure exclusive access (Step 1).

State 2: After context switch. The lock was requested by Bank Operation 2, but the lock was set to 1 in state 1, so Bank Operation 2 is moved to the waiting queue.

State 3 & 4: After context switch. Bank Operation 1 updates the balance from 500 to 700 (step 2), and then unlocks the Bank (step 3).

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 `lock`
- `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:


- If the lock is available, but the waiting queue is not empty, bring the first thread in the waiting queue to the ready queue. And that thread should acquire the lock. (**Upd 11/29**)
- 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 `lock`, 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 `lock` and `thread_exit`, you should execute the thread you used to fill up the hole.
 - If the thread calling `thread_yield`, `thread_exit` or `lock` is the last thread in queue, execute the first thread.
- When both the ready queue and the waiting queue are empty, return to main.

Sample 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.

We've also provide a shell script for you so you can check your output of the sample execution.

 **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`

Case 1

```
$ ./main 3 6 10 0 0
0 fibonacci
1 factorial
0 1
0 1
0 2
caught SIGALRM
1 1
1 2
1 6
caught SIGALRM
0 3
0 5
0 8
1 24 // thread 0 exited, so SIGALRM is pending
caught SIGALRM
1 120
1 720
1 5040
caught SIGALRM
1 40320
1 362880
1 3628800
```

Case 2

```
$ ./main 3 5 3 0 0
0 fibonacci
1 factorial
0 1
0 1
^Zcaught SIGTSTP
1 1
caught SIGALRM
0 2
^Zcaught SIGTSTP
1 2
1 6 // thread 1 exited
0 3
^Zcaught SIGTSTP // SIGTSTP has higher priority than SIGALRM
0 5
```

Case 3

```
$ ./main 3 0 0 200 -900
2 bank_operation
3 bank_operation
2 acquired the lock
^Zcaught SIGTSTP // thread 3 can't acquire lock
2 500 700
2 released the lock
3 acquired the lock
caught SIGALRM
3 700 700 // not enough balance
3 released the lock
```

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`, `lock`, `unlock`, 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?
- Is there any chance that your implementation leads to a deadlock? How do you avoid it?

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 `kill -9 -u [student id]`.

We will grade your submission on ws1, but you are encouraged to write and debug your programs on ws2~ws5.

- (4pt) Your `threadtools.h` and `scheduler.c` work smoothly
 - The TAs will use their version of `main.c`, `threads.c` and `bank.h` for this subtask.
 - **Upd 12/01** Please make sure that your code will work with different thread functions.
 - (2pt) Without `lock`
 - 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 (0.5pt) The order of execution is correct.
 - (2pt) With `lock` and `unlock` (**Upd 12/05**)
 - 2-1 (1pt) lock are maintained correctly and the order of execution is correct when number of threads = 2.
 - 2-2 (1pt) lock are maintained correctly and the order of execution is correct when number of threads > 2.
- (4pt) The two functions in `threads.c` are implemented correctly. **For each function**,
 - The TAs will use their version of `main.c`, `scheduler.c`, `bank.h`, `threadtools.h` for this subtask.
 - 3-1~3-2 (0.4pt) The last line of the output of **factorial** is correct
 - 3-3~3-5 (0.4pt) Every line of the output of **factorial** is correct
 - 4-1~4-2 (0.4pt) Bank balance is correctly updated.
 - 4-3~4-5 (0.4pt) Lock is correctly acquired and released in bank operation.