NTNU
Kunnskap for en bedre verden

Department of Computer Science

TDT4186 - Operating Systems

Lab 4

Userlevel Threads, Concurrency, & Synchronization

Professor:
Di Liu
Teaching Assistant:
Roman K. Brunner
Student Assistants:
Eik Hvattum Røgeberg
Halvor Linder Henriksen
Ole Ludvig Hozman
Daniel Hansen

Handout: 23.03.2023

Introduction

Hello, and welcome to the final lab! In the labs, you have the opportunity to practically try out things you learned in the lectures and the theoretical exercises. The labs are compulsory coursework, you need to **complete three out of four labs** to sit in the exam. As the labs are mandatory coursework, they are also subject to NTNU's plagiarism rules. More on that in Section 3.

If you haven't done yet, please sign up for one of the lab sessions under https://s.ntnu.no/LZZtgbcC. You need to be logged in with your NTNU account in order to access the file. The lab sessions will help you get started

1 Task Description

In this lab we are building a user-space threading library, that allows a single process to have multiple threads. The threads will not execute simultaneously on multiple cores but will share the time the process gets on the core.

The threading library will provide the following functionality:

- The implementation needs to be able to handle the creation of at least **eight additional threads**
- Each thread needs to have its own stack to allow independent execution of functions in the different threads
- The threads can access the memory of other threads
- The threads use a cooperative scheduling model, so the scheduler only gets invoked when a thread yields
- A round-robin scheduler decides which thread to execute next
- A thread join function, which lets a thread wait for another thread to finish and possibly get the return value
- A `whoami` function that allows a thread to find out its own thread id

In order to make the implementation a bit easier, you find the following additions in the handout:

- A small user-space lock library implementation
- A `tswtch` function to switch thread contexts on the CPU. You find the assembly implementation in the `tswtch.S` file, if you are interested in the inner workings
- Stubs for the `thread` and `thread_attr` structs in the `user.h` file
- The thread library declarations are present in the `user.h` file and empty definitions of the functions are found in the `uthread.c` file.

Feel free to add whatever you deem necessary to implement the corresponding functionality. **But don't change the function outlines of the function declarations provided, as we expect that exact function outline for our tests.** You find more details to the functions in the task descriptions further down below.

"Threads" Source: https://comic.browserling.com/53

Background

We will use a slightly different approach than in the earlier labs for the threading library. Until now, we mostly worked in the kernel and exposed this functionality through syscalls. While many modern operating systems keep a lot of code and responsibility in the kernel, other approaches are also possible. Instead of handling the logic in the kernel, parts of the responsibilities could be given to user-space programs, which provide certain operating system services. Another is to give a certain amount of responsibility to each process and implement a part of the logic in a user-space library. In such approaches, the operating system's kernel only provides minimal services and does check critical security actions, but the central part of the logic is running in the user space.

As with everything, these approaches have advantages and disadvantages. Among the advantages, we have smaller binaries with specific responsibilities that are easier to verify for their correctness and thus expose a smaller attack vector. Some might also claim that such systems show higher resiliency, as user-space processes can quickly be restarted upon failure, while kernel failures usually bring the system to a halt.

Some of the frequently mentioned disadvantages of so-called "Micro-Kernels" are that they require many more context switches due to having to switch forth and back between service and program processes. A syscall is relatively costly compared to a function invocation; thus, micro-kernel performance can suffer. We do not want to go too deep into the discussion of monolithic versus micro-kernels here, and you can find many more arguments in favour of one or the other. For this lab, we are going to implement operating system functionality in the user library for XV6.

Threads

The task for this lab is to add a user space threading library to the user lib. The goal is that a single process is split up into multiple threads, executing different functions. Threads are not clearly defined in general, some of them resemble a lightweight process[1], while others are an easy way to handle multiple different tasks but do not necessarily provide parallelism, even when run on a multi-core system. We will implement the latter by enabling a process to create multiple threads and control their execution. Since everything is running in the user space, this is not technically part of the operating system, but it takes over similar responsibility.

[1]The main distinction between process and thread, in this case, is that all threads share the same virtual memory, while processes don't.

In this implementation, threads cannot run simultaneously on multiple cores. But even with simple user-space threads that are timesharing the processes execution quota and a round-robin thread scheduler, we can develop scenarios that require synchronisation for data accesses. This means that we need some locking to guarantee that the state is consistent over multiple non-atomic operations. Suppose we invoke the scheduler based on interrupts (a user process could register with the operating system). In that case, we end up with a situation where we can't possibly guarantee the order of operations being applied to shared data if we are not using any locking structures.

1.1 Task 1: Setting up a process for using threads

In this subtask, you will implement the thread creation, the thread scheduler and the thread yield function. You are not yet required to pass arguments to the thread or get the return value of the thread. You should be able to test it with a small program that creates a thread, yields and the thread then will print "Hello World" to the shell. You can take a look at `ttest.c` in order to take a look at some examples on how the threading library is going to be used.

When a process is started, the first function to be executed (the first address the kernel returns to) is the `_main` function defined in `ulib.c`. If you need to initialize specific structures before the program's main function can start, this is the place to do so. Also, think about how you ensure that the main function (and whatever the main function calls) is also being run by the thread scheduler and does not starve.

Last, you will implement the `twhoami` function in this task. This function should return the thread id of the currently running thread.

See below for a detailed explanation of the interfaces of the functions you are going to implement.

`tcreate` The tcreate function allocates memory and sets up a new thread. Before returning the newly created thread is set to be runnable, so that the scheduler might run the thread the next time it gets to take a scheduling decision. The tcreate function takes the following arguments:

- thread It is the responsibility of the `tcreate` function to allocate memory for the thread struct, and then set the thread argument pointer accordingly so that the newly allocated thread is visible to the caller. This can be done by dereferencing the double-pointer once (e.g. `*thread = (thread *)malloc(sizeof(struct thread));`). The thread double-pointer does not need to point to some meaningful memory region before the call. For an example of how this looks from the user side, see `task1test.c`.
- attr The struct `thread_attr` contains a bunch of settings for the thread. If the pointer is null or values within the struct are null, assume the following default values: `stacksize` = `PGSIZE` and `res_size` = 0. The `stacksize` represents the size of the stack for the new thread in number of bytes and the `res_size` denotes the size of the result value in the number of bytes.
- func This is the pointer to the function to be executed by the thread. Note that the function needs to take a void pointer as the argument and returns a void pointer. That allows us later to pass any arguments and return any results. This value must always be set.
- arg The arg argument contains a pointer to a potential argument value that should be passed to the function within the thread. If a value of 0 is passed, no argument will be passed to the function

`tsched` The tsched function is called when a new thread should be scheduled. It goes through all the currently available threads within the process and selects the next one to run. It can use the `tswtch` function to switch to the next thread.

`tyield` The tyield function can be called by any thread to indicate that another thread should get the run next

Optional tasks

EASIER: Think about how to include the main function in the threading logic. Should the main function be its own ("main") thread, or should it live outside of the threading logic? In either of those cases, how do you ensure that the scheduler schedules the main function from time to time, and where do you save the main function's information (e.g. the context)?

EASIER: How could we keep track of the created threads[2] and the currently running thread? When thinking about the `twhoami` function, remember that at any point, you should be able to return the thread id that is running.

EASIER: Create a simple thread that handles the thread pointer argument and the func pointer. How do you ensure that we are executing the function once we switch to that thread? Think about what the thread context switch does. Make sure to also add this thread to the bookkeeping structures in your solution.

SIMILAR: Implement a round-robin scheduler that cycles through the currently available threads in the system. You can use the procstate enum to track a thread's state.

EASIER: What does yielding in this context mean? What should be done if a thread wants to give up the execution resource (aka the CPU)?

HARDER: While the mandatory tasks only ask to implement a cooperative scheduler, you could think about how to use the timer interrupts (and how to register for them) to pass control to the thread scheduler without the thread yielding. *A word of caution: This is a hard task that requires quite a bit of extra code, so only start with it if you have done all the mandatory tasks.*

1.2 Task 2: Wait for a thread to finish

Sometimes we want to wait for another thread to finish, such that we can either use its result value or just make sure we don't continue before all work has completed. For that we use the `tjoin` function.

tjoin Joins a thread (waits for it) and potentially gets the return value from the thread. It takes the following arguments:

- tid The thread id of the thread to join. Make sure every thread has a unique id
- status A pointer to a memory region that is big enough to hold the result value (can be allocated either on the stack or on the heap using malloc)
- size The size of the result value, so that the corresponding value can be copied into the status section[3]

For now, you can assume that status and size will always be zero (no return value). So you only need to implement the behaviour that waits for a specific thread to finish, before the current thread will continue execution.

Optional Tasks

EASIER: If we are waiting for a thread to finish, how can we wait? What would be a good option?

SIMILAR: There might be multiple threads waiting for another thread to finish. How can we ensure all of them see that the thread has finished?

[2]Note that the thread id is a uint8, so can take at most 256 different values, thus putting a limit on the number of threads per process.

[3]Please note that passing the size on the join call is not a particularly save option to read out the return value, as the joining thread could ask to copy more memory than the actual return value. In our implementation this does not matter for security, as between threads all the memory is accessible anyway.

HARDER: Can you devise a waiting scheme based on a notification, meaning that a waiting thread will not be scheduled unless the thread it is waiting for has already finished?

1.3 Task 3: Passing arguments and receiving return values

Now in the last step, we want to also be able to pass arguments to the thread and receive the computed results. As you might have noticed, the functions that we pass take a void pointer as the argument and return a void pointer. The reason for that is that we can cast any pointer to a void pointer and also cast a void pointer to anything. This also means that the developer needs to ensure it casts only to valid types. For an example on how this is used, take a look at the file `ttest.c` and the functions `test3` and `calculate_rv`.

You might want to add another layer of indirection that handles setting up/tearing down a thread just before the function gets executed (a so-called "wrapper").

Optional Tasks

EASIER: Think about how you could pass the argument to the function in the thread.

SIMILAR: How can you return the result to multiple threads that might be waiting (joining) this thread? The return value for each of the waiting threads should not be shared with the other threads ("private" in the sense that no other accidentally changes the underlying value).

HARDER: Can you implement the same behaviour without a wrapper function? Hint: This could be done by placing the argument pointer in the correct register and handling the return value in the join function.

2 Handing In

Before handing in, we advise you to use the command `make test`, which will test your implementation with a small set of test cases. If you want to see the tests in greater detail, you find the commands that were executed and the expected output in the file `test-lab-l3`. The test script also tests for some of the optional tasks, so you don't need to pass all the tests. We marked the test cases that belong to the mandatory tasks with a tag `[MANDATORY]`. Try to make sure that all the mandatory tests pass before handing in.

After you tested and potentially debugged and fixed your solution, you can run `make prepare-handin` to create an archive of your solution. The archive will be written in the current directory and the file is called `lab-l3-handin.tar.gz`. Take that file and upload it to blackboard on the submission page for lab 3. You can submit multiple solutions, we are going to grade the last submission we received before the deadline.

If you encounter any difficulties with handing in, contact Roman[4] immediately via email, so we can sort things out.

Please note that if you submitted before the deadline and did not contact us in case of any problems, we will proceed to grade your solution. Once your solution is graded and we released the feedback, you cannot resubmit a new solution.

Files to Hand in

- **lab-l4-handin.tar.gz**: The output of the command `make prepare-handin`. Please do not package up your code manually, as this command runs some tests on your side and ensures that we see if it ran on your machine. Additionally, the format it packages it up is the one expected by our automated tests.

No other files need to be submitted for this lab. If you have any comments, please feel free to use the comment field for submissions in Blackboard. If you have any questions please use Piazza or come to the lab sessions. We will not answer questions that are in the comment field to your solution.

Missing Deadlines

Since this is the last lab and we have to hand in who gets to sit in the exam and who not, we cannot grand any extensions. In case of emergencies that prevent you from finishing the lab in time, contact Roman[4] immediately.

3 Plagiarism

You must submit your **own work**. You must write your **own code** and **not copy** it from anywhere else, including your classmates, internet, and automated tools[4]. Failure to do so is considered plagiarism. Detailed guidelines on what constitutes plagiarism can be found at: https://innsida.ntnu.no/wiki/-/wiki/English/Cheating+on+exams.

We check all submitted code for similarities to other submissions and online sources. Plagiarism detection tools have been effective in the past at finding similarities. They have gotten excellent over time, so it is inadvisable to try and outsmart them. So don't do it, not only because we will most likely catch you, but because it is morally wrong and can undermine your academic integrity, even a long time into the future. For more references, see https://www.google.com/search?q=resigns+over+plagiarism+allegations (statistics on the 8th of August: 467'000 results).

[4] roman.k.brunner@ntnu.no

[4] This is not an exhaustive list. Don't copy code from any source