Lab 3

CSCE 313

1 Introduction

The objective of this lab is to implement a **user-space**, **cooperative**, **multi-tasking** library. Let's take a look at what each of these words mean one by one.

User-space means that this library will exist in the user space where all the user programs (like editors, web browsers, etc.) are executed. This means that we *won't* have to inject any code in the kernel (yay!).

Cooperative means that the workers (i.e. the processes/threads/etc. responsible for execution the tasks) in this library cooperatively yield control over to other workers. This is different from how operating systems (OSs) are *usually* implemented. Typically, the OS preempts the worker thereby taking away the control from it, rather than relying on the worker to willingly yield over the control. The reason why OSs typically follow this type of multi-tasking frameworks, known as preemptive multitasking, is because the OS cannot trust the workers. The workers may be faulty or malicious. Therefore preemptive multitasking is necessary to stop untrusted workers. This design, however, comes at a cost of higher complexity because the OS is typically not made aware of when is a good time to stop a worker. To avoid such complexities, we've chosen a cooperative design in which workers can explicitly yield the control whenever they want to. More information about these two design choices can be found here.

Multi-tasking is a catch all term for multi-threading and multi-processing. Strictly speaking in this lab we won't create threads, rather what we'll create are tasks. A task is defined as a unit of work which the parallel runtime (the OS, or a parallel runtime user-space library like this assignment) guarantees will get completed. A parallel runtime is defined as the set of function calls and their behavior which allows the users to parallelize their code.

Typically in user space parallel runtimes the tasks are allowed to share the same heap and the same stack. However, note that in such runtimes the shared stack means that the tasks can potentially, in some implementations, read off the variables allocated on the stack of the task which spawned them. Our library, which you'll implement, does **not** allow the stack to be shared in this manner. In fact in our implementation, you'll *explicitly* allocate the stack for the various tasks.

Moreover, note that in this lab you are **not** required to use more than one processors, but if you want to, we won't stop you. Note that if you don't use multiple threads, that library runtime that you'll implement will technically not be a *parallel* runtime, rather it will be a *concurrent* runtime. **For this lab we'll accept both the implementations. You're free to choose whatever you want to do.**

Since the machine registers are shared between the various tasks (because the library executes on the same processor) we need to save the state of the machine, also called the *context*, whenever we switch to a different task. These contexts, which include the values of various architectural registers, are stored in a specific data structure in the library. Therefore, whenever we want to switch the control over to a new task we need to perform two steps: (1) save the current context, and (2) switch over to the new context stored in the internal data structure. Moreover, since the context holds the complete information of the state of the machine when the context was captured, when we resume from a context it appears as if we went back in time and started from where we had left off, i.e. from the point of context capture.

In this lab, you are **not** expected to write various functions which deal with saving and restoring the snapshots, rather you only need to deal with resuming the correct context at the correct time. We'll use the `ucontext.h` header provided by the standard C library, i.e. `libc`, which is available on `UNIX` systems for this lab. More details on the header can be found in the `man` pages. Please refer to the `man` page for `makecontext` for an example program. You can use whatever `libc` calls and `syscalls` that you want for this lab. However, please note that this lab can be completed using only three function calls declared in `ucontext.h`: `swapcontext`, `getcontext`, and `makecontext`. The TAs will cover the semantic of all these three functions in the lab sessions to get you started.

Since many of you might not have used these functions before, and many of you might not have written a parallel runtime before, we recommend you to **get started as early as possible** since this lab will take some time.

2 The Library Calls

The `ucontext.h` headers and its functions operates on objects of the type `ucontext_t`. More details of this type can be found in the `man` pages for `getcontext`. This structure represents a captured context of the machine.

1. `getcontext`: This function takes a pointer to an object of the type `ucontext_t` and populates the various field of the object such that after the call to `getcontext`, it contains a valid captured context of the machine. Note that this context can then be used to restore the machine state to the point where the context was captured, i.e. to the point just *after* the call to `getcontext`.

2. `makecontext`: This function takes a pointer to an object of the type `ucontext_t` and modifies it. The object which is passed to this function must be a valid context.

This function also takes in a function pointer, the number of arguments to be passed to the function, and all the arguments which are to be passed to the function. This function modifies the context which was passes to it (via the pointer) such that whenever this context is resumed, the control jumps to the first instruction of the function whose function pointer was passed. The function which was passed gets called with the arguments specified in the call to `makecontext`.

However, note that although this call modifies the context such that it executes the right function when it is resumed, it does **not** set up the stack for the call to this function. This task is left to the users of the `makecontext`, which happens to be you in this case.

(a) To setup the stack you need to set the pointer `context.uc_stack.ss_sp` to some space in memory which can be used for the stack (you can simply `malloc` some space for this).

(b) You need to set `context.uc_stack.ss_size` to the size of the stack space which you've allocated.

(c) You need to set `context.uc_stack.ss_flags` to 0 because the child task is not required to handle any signals.

(d) Finally, you need to set `context.uc_link` to `NULL` to indicate that once the function finishes execution, the worker can simply terminate. You can optionally make this field point to another `ucontext_t` object which would be resumed after the specified function exits. Setting this field to `NULL` ensures that the worker would exit after executing the specified function.

Once you've used the `makecontext` call and completed the four tasks listed above, the context which you modified is finally ready to be used.

An example of makecontext

```
getcontext(ucp);
ucp->uc_stack.ss_sp = (char *) malloc(STACK_SZ);
ucp->uc_stack.ss_size = STACK_SZ;
ucp->uc_stack.ss_flags = 0;
ucp->uc_link = NULL;
makecontext(ucp, (void (*)()) fn, 2, arg1, arg2);
```

Figure 1: Setting up `makecontext`

Please refer to the Lab 3 section of the class notes (Figure 1) posted on Canvas for more information about this function.

3. `swapcontext`: This function takes two pointers to two distinct contexts as its arguments. It then saves the current context, i.e. the context of the

caller, in the first argument and restores the execution from the second context. Note that when the execution is restored, the control would jump to wherever the second context was created. This means that if the call to `swapcontext` doesn't fail, then the control is not guaranteed to return to the call site.

Please note that the above description is very brief and you should refer to the `man` pages for more information. The class notes, which are posted on Canvas, also have an example code snippet for the usage of `makecontext` under the Lab 3 section. **However, the Lab 3 described in the notes is not the same as what is described in this document. You need to follow this document, and not the class notes, for this lab. The notes are *only* there for your reference.**

3 API Details

The API of the library that you are required to create is mentioned in the `README.md` file in the starter code. The function signatures of various calls which you need to support are mentioned in the `threading.h` file. The TAs will also go over the function signatures in the lab sessions.

1. `void t_init()`: This is the first function which is called by any program which uses the library. You can use this function to initialize various data structures which your library may or may not be using.
2. `int32_t t_create(fptr foo, int32_t arg1, int32_t arg2)`: This call is used to create a task which will call the function `foo` with two arguments, `arg1` and `arg2`. Note that the `fptr` type is a function pointer of the type `void (*fptr)(int32_t, int32_t)`, i.e. `foo` *must* be a pointer to a function which returns nothing, i.e. `void`, and which *must* take in two 32 bit signed integer arguments `arg1` and `arg2`. For an example of such a function and the usage of `t_create`, please refer to `main.c`.
3. `t_yield()`: This function is called by a worker to indicate that it can relinquish the control over to other workers at the call site of `t_yield`. The existence of this function tells us that this library is a *cooperative* multi-tasking library.
4. `t_finish()`: This function is called by all workers, except the main worker which spawns all the tasks, at the very end to indicate the completion of the assigned task.

Please refer to `threading.h` for more details about the exact API that you need to implement.

4 The Starter Code

The stater code comes with seven files as described below:

1. `Problem.pdf`: This document which describes the task for Lab 3.
2. `main.c`: The code which uses the library that you'll implement. You are **not** allowed to modify this file as this is the test application which we'll use to evaluate your submission. However, if you want to write other programs which use your implementation for testing purposes, you can always copy this file somewhere else and then make changes to it.
3. `Makefile`: The Makefile to build the library and the executable. You can modify this file if you so desire, but you probably won't have to.
4. `README.md`: This file provides more details about the task.
5. `threading.c`: This file holds the implementation of the library. You are expected to modify this file however you want. To get you started, we've provided sections in this file where you can write your code. These sections are marked with `TODO`s.
6. `threading_data.c`: This file defines all the variables which your library implementation may or may not use. You can modify this file if you want, but you probably won't have to. However, note that if you add/remove various variables to/from this file, make sure to add the corresponding `extern` declarations for them in `threading.h`.
7. `threading.h`: This is the header file for the library which is used by `main.c`. This file has all the information about the various calls that you've to implement. You can **only add** declarations in this file but you are not allowed to remove any of the function declarations. If you do remove any of these declarations, then the code would fail to compile.

The starter code is structured as dynamically linked shared library. The code that you'll write in `threading.c` would be used to compile a shared library named `libthreading.so`. This library can then we used with any other program. The `main.c` file implements one such program which uses this library. To figure out the functions supported by this library, `main.c` refers to `threading.h` header file which has all the declarations for `libthreading.so`. When `main.c` is compiled to generate the application, `main`, it is linked with the shared library. As a result, whenever `main` calls one of the library functions, the function instructions are picked from `libthreading.so` dynamically. The `Makefile` handles all the compilation logic. The output of the `Makefile` is one executable, `main`, and one dynamically linked shared library, `libthreading.so`.

5 Hints

Please note that the hints provided here are merely *suggestions*. You are under **no** obligation to follow them.

You can use the `contexts` array defined in `threading_data.c` to store all the contexts. You can also use `current_context_idx` to keep track of which context is active right now.

1. `t_init()`: You can use this function to initialize various data structures that your library uses such as `contexts` and `current_context_idx`.

2. `t_create()`: In this function you can first find an empty entry in the `contexts` array (you can use the `state` field of the `struct worker_context` and `enum context_state` to track which entries are unused, i.e. `INVALID`). Then you can initialize this entry using `getcontext`. After initializing the entry you can then use `makecontext` to modify the context such that it calls the function passed as the argument to `t_create()`. Make sure to also allocate the stack for the context here. You may also *optionally* choose to schedule the new context at this point itself.

3. `t_yield()`: In this function you can first update the current context (whose index is stored in `current_context_idx`) by taking a fresh snapshot using `getcontext`. Then you can search for a `VALID` context entry in the `contexts` array and use `swapcontext` to switch to it. After this you can then compute the number of contexts in the `VALID` state and return the count. Note that since the switch happens before you compute the number of contexts in the `VALID` state, when the original context is resumed, the calculation of this count will be done at that point.

4. `t_finish()`: In this function you can first `free` the stack allocated for the current context. Then you can reset the context entry to all zeros by using `memset`.

6 Rubric

This assignment is worth 100 points among which 20 points are for code which correctly compiles. 70 points are for the correct implementation. The final 10 points are for code which is free of any memory leaks.

Note that since the Address Sanitizer is not fully complaint with the `ucontext.h` library, we'll use another tool called `valgrind` to check for leaks. To use this tool, you need to install it on your machine by running `sudo apt-get install valgrind`. However, note that `valgrind` does **not** play nice with `ucontext.h` by default. Therefore, we'll only use it to detect memory leaks and we'll ignore all the other errors that it reports. To run `valgrind` you can execute `valgrind --leak-check=full --show-leak-kinds=all --track-origins=yes ./main` after building the code. If your code is free of any memory leaks, you should see "`All heap blocks were freed -- no leaks are possible`" printed at the end. If after running `valgrind` you can see this message, then you'll get the 10 points for leak free code. If you face any weird behavior from `valgrind`, please mention it on the Discord chat. It's possible that `valgrind` cannot be used, at least not without some `valgrind` specific code changes, for this lab. *If* this happens to be the case, we'll remove this component and the 10 points for this components would get merged with the 70 points for the code, bringing the total to 80 points.

7 What to Submit

You are required to submit the following four files for Lab 3:

1. `threading.c`
2. `threading.h`
3. `threading_data.c`
4. `Makefile`

Please start this lab assignment early and ask all your questions on Discord. It is quite likely that you'll take some time to complete this task. Good luck and Gig 'em!