# Lab 1: Cooperative User-Level Threads
# Due Tuesday, May 5, 2020, 11:59 p.m. PDT

## Overview

In this lab, you will write a cooperative user-level threads library, chloros (a Greek word meaning “green”) . You will be programming in C++ and x86-64 assembly on `myth` cluster. If you're not familiar with `myth` cluster, see Stanford's Shared Computing Environment.

*User-level* threads are similar to the type of threads you might be familiar with, OS threads, but are implemented entirely in user-level code. They are typically speedier than OS threads since there is no context switching into/out of the kernel. *Cooperative* usel-level threads are user-level threads that release control to the scheduler deliberately, usually through a `Yield()` call. That is, they execute for as long as they want.

A user of your chloros library first calls `Initialize()` to initialize the first user-level thread, one representing the originally executing program. Then, the user calls `Spawn(fn, arg)` to create a new thread that executes `fn` with `arg`. The user calls `Yield()` inside a thread to yield execution to another thread, including the initial one. Finally, the user might call `Wait()` to wait for all of the spawned threads to finish executing before ending the process.

A full program using your library might look like this:

```
#include "chloros.h"

extern bool NotFinished(void*);
extern void DoWork(void*);
extern void* GetWork();

void Worker(void* work) {
  while (NotFinished(work)) {
    DoWork(work);
    chloros::Yield();
  }
}

int main() {
  chloros::Initialize();
  for (int i = 0; i < 10; ++i) {
    chloros::Spawn(Worker, GetWork());
  }

  chloros::Wait();
  return 0;
}
```

This lab is divided into 4 phases and an optional extra credit phase. The first 3 phases will guide you towards implementing cooperative user-level threads. Once you have completed phase 3, the example program above will run as you expect. In phase 4, you will implement simple thread garbage collection so that resources aren't leaked after a thread terminates. In the optional extra credit phase, you will run it on multiple kernel threads and detect data races.

There are unit tests corresponding to each phase, allowing you to mark your progress as you work through the lab.

# Phase 0: Getting Started

First, ensure that you are working on the lab using a machine meeting the following requirements:

- Runs a modern Unix: Linux, BSD, or OS X.
- Runs a 64-bit variant of the OS (check with `arch`).
- Has GCC or Clang and GNU Make installed.

If you don't have a machine that meets these requirements, you can work on the lab through `myth.stanford.edu`.

## Getting the Skeleton Code

To begin, download the skeleton code to your development machine.

## Exploring the Skeleton Code

The majority, if not all of your work will be to modify the files found in the `src/` directory of the skeleton code:

- `src/chloros.cpp` contains thread management functions and user-level thread library functions.
- `src/context_switch.S` contains the assembly routines that implement thread context switching.
- `src/common.cpp` contains debugging and logging utilities.

The `include/` directory contains the header files for these `src/` files where applicable and are named like their `src/` counterpart. You may find the debugging macros helpful.

Finally, the `test/` directory contains the unit testing code. If a test you expect to pass fails, you can look at its test file to see why. You are encouraged to look at the test code to see how the library is supposed to work. The test cases are not complete, and you should add your own test cases.

## Make Targets

You will use `make` to build, test, and create a submission file for this lab. The following targets are defined in the `Makefile`:

- `all`: compiles and packages your library.
- `test`: compiles all test cases.
- `submit`: creates the `submit.tar.gz` submission file.
- `clean`: deletes temporary files.

Calling `make` in your shell will implicitly run the `all` target. The other targets can be invoked using `make {target}`. Ensure that you are in the root of your lab directory before invoking `make`. Try calling `make test` now. You should see test cases failing. If this doesn't work, your machine may not meet the requirements.

## Getting Familiar

Before you continue, read through the `src/` files and some of the unit test files to see how the functions you will write are intended to be used and what we will test for. Places commented with `FIXME` need implementation. Also pay attention to the code style: how variables are named, how code is aligned, and how spacing is used. You should write your code with an equivalent style. Feel free to

use `clang-format` to automatically format your code. Once you have a good grasp of the thread interface and functions, you're ready to start writing code. The comments also contain detailed behavior of functions you will come across, especially where the semantics are not that clear. Be sure to read them.

## Phase 1: Allocating and Deallocating Threads

In phase 1, you will implement the constructor and destructor for `Thread`.

Pay attention to fields in `Thread` structure, which you need to initialize properly. If you are going to run the library on multiple kernel threads (in the extra credit phase), assign thread IDs in a way that avoids data races. Make sure they are unique. And make sure that if stack is required, it is 16-byte aligned. In the destructor, make sure you properly release memory acquired by the thread.

You do not have to worry about `thread_queue` yet at this stage.

## Phase 2: Context Switching

In phase 2, you will write the code to context switch from one thread to another in the assembly function marked `FIXME` in `src/context_switch.S`.

The `context_switch` function will be called on behalf of a usel-level thread during its execution to context switch from one thread to another. For a context switch to be successful, all state of the currently executing thread must be saved in some structure and the state of the thread being switched into must be restored. A thread's execution state is also referred to as its *context*.

A thread's context is already defined in the `Context` structure. All of these registers are known as *callee saved*, which means that the *callee* of a function guarantees that those registers will have the same value when the function returns as they did when the function was called. In other words, a function **A** calling another function **B** can expect those registers to contain the same values after function **B** returns as they had when function **B** was called because the *callee* (**B**) *saves* and restore those values. These registers are defined this way by the System V X64 ABI calling convention, which is followed on most Unix systems running on 64-bit machines. You can read more at the ABI reference link.

Apart from these registers, a thread's full context also includes the values in its stack. Saving and restoring a thread's stack on each context switch would be a very expensive operation, so instead of doing this, we simply give each thread its own unique stack. Then, as long as each thread's stack pointer points to its own unique stack, saving and restoring the stack pointer suffices to save and restore the thread's stack. Later on, you'll write the code that ensures that each thread's stack pointer points to its own unique stack (which you've already allocated in `Thread::Thread(bool)`). The `context_switch` function should only be called on already running threads, so the validity and uniqueness of the stack pointer can be assumed.

You're ready to implement the context switching assembly function in `src/context_switch.S`. You can find the function's specification above its definition. Keep in mind that according to the calling convention, the first two parameters to a function are passed in the `%rdi` and `%rsi` registers. Also note that GCC calls the GNU Assembler implicitly, which uses the GAS syntax for assembly. You may wish to consult the calling convention in the ABI reference or the X86 instruction reference. After you have implemented this routine, you should pass the phase 2 unit tests.

This phase is not meant to grill your knowledge of assembly. You can implement `context_switch` using only two different assembly instructions: `movq` and `ret`.

## Phase 3: Yield and Spawn

In this phase, you will implement `Yield` and `Spawn` functions.

When a thread wants to release control to the scheduler, it yields its execution time by calling `Yield()`. The function's task is to find and execute an available thread. It is a scheduler: it makes decisions about which thread to schedule when and does the necessary bookkeeping to continue scheduling threads reliably in the future.

Each thread is marked with a state of `kReady`, `kRunning`, `kWaiting`, or `kZombie`. A thread is `kReady` when it can be executed, `kRunning` when it *is* executing, `kWaiting` when it is waiting for other threads, and `kZombie` when a thread has finished executing but hasn't had its resources freed and destroyed.

Your `Yield()` function should use and modify these statuses for scheduling. It should implement a round-robin scheduler, performing a linear search for the next `kReady` thread beginning at the currently executing thread. Once it finds a thread, the status of the currently executing thread is set to `kReady` if it was previously `kRunning`, and the newly found thread's status is set to `kRunning`. The pointer to the currently executing thread, `current_thread`, is updated. Finally, the function context switches into the newly found thread.

In this step, you also need to maintain `thread_queue`, which is a global queue of threads that are waiting or runnable. Later, when we run with multiple kernel threads, this data structure can be accessed concurrently by different kernel threads. Consider using locks to protect it. Some locking code is provided as a hint; however, you can remove the locking code if it doesn't make sense to you at this stage, as there is only one kernel thread for the first four phases of the lab. The locking code becomes important only in the extra credit phase.

A thread created by a user executes a function of the user's choice. The `Spawn` function accomplishes exactly this: it allocates a new thread, initializes its context so that the user's function is executed after a context switch, and finally calls `Yield` to yield the caller's execution to the newly initialized thread.

When a thread is allocated by your constructor function, it has an empty context, so context switching into it would likely crash the process. Your task is to write `Spawn` to set up values in the stack so that after the first context switch into the thread, the thread begins executing the `start_thread` routine, a thread start function written in assembly we have provided to you. `start_thread` expects the user's `fn` to be at the top of the stack when it is called, and the argument to that function above that. Thus, you must design the initial stack so that during a given thread's first `context_switch`, `context_switch` returns to `start_thread`, and `start_thread` finds the function at the top of the stack. `start_thread` will set up argument-passing registers and call `ThreadEntry`, where additional initialization and exit logic is handled.

The implementation of this function requires you to write values into the thread's initial stack and then set the stack pointer appropriately. Note that the x64 ABI *mandates* the stack pointer to be 16-byte aligned. The full function specification is above its definition. After you have implemented this routine, you should pass the unit tests.

Hint: the only register needing initialization is `%rsp`.

Note that the provided unit tests are not exhaustive. We may run additional tests during grading.

## Phase 4: Garbage Collection

In this phase, you will implement `GarbageCollect` in `src/chloros.cpp`, a simple thread garbage collection routine.

Upon thread termination, the thread's initial function returns to `ThreadEntry`, and `ThreadEntry` sets the thread's status to `kZombie` so that it doesn't get scheduled again. (Prior to this phase, this thread will just remain in the queue.)

Your task is to ensure that a thread's resources are freed after it has exited. You must implement `GarbageCollect`, which frees resources of all threads with status `kZombie`. Then, insert a call to `GarbageCollect` in an appropriate location in your library, so that terminated threads are recycled as soon as possible. Think carefully about the earliest point at which you can safely reclaim a thread's resources. After this, you should pass the phase 4 unit tests.

## Submission

Once you've completed the tasks above, you're done and ready to submit! Ensure that your lab and tests run as you expect them to on `myth.stanford.edu`. We will grade your lab on the `myth` machines. Any changes made to files outside of the `src/` or `include/` directories will not be submitted with your lab. Ensure that your submission is not dependent on those changes.

Don't forget to check your code for memory leaks using `valgrind`!

Finally, run `make submit` and then proceed to Gradescope to upload your submission.

# Phase 5: Extra Credit

This phase is entirely optional and lack of completion will have no adverse effects on your score. You must successfully complete all phases to be awarded the extra credit.

In this extra credit phase, you will choose **one of** the following tasks. Be creative in implementing and testing your idea, and write up a short report. Note that the questions do not have unit tests.

## Data Races

In this task, you will run your user-level threads on multiple kernel threads and try to detect data races with ThreadSanitizer.

Sample code is provided in `test/phase_extra_credit.cpp`. Think about how the user-level thread library interacts with itself when running on multiple kernel threads, and make sure it doesn't generate data races. Make sure to consider the case where multiple kernel threads access `thread_queue` at the same time.

After you make sure that the library itself is free of data races, reproduce an example of the erroneous double-checked locking pattern (DCLP) described in the Eraser paper. If you use `std::atomic`, note that it uses `std::memory_order_seq_cst` which is sequential consistency by default. If you use it as the pointer to be set, the code will actually be free of data races. So try to relax the constraints. Or try to use a spin-lock that doesn't properly synchronize the code region it's protecting. At any rate, get creative, and construct a non-trivial case where there is a data race.

The reason we ran user-level threads on multiple kernel threads is because user-level threads are not preemptive, so they won't show data races unless you abuse `Yield`. With multiple kernel threads, user-level threads can run concurrently. Compile your example with `-fsanitize=thread`, which uses ThreadSanitizer, a more complicated version of the idea presented in the orginal Eraser paper. It should catch the data race.

Write up an `extra_credit.txt` describing what you have done to create the data race, and how ThreadSanitizer catches it. Your write-up doesn't need to be long, but be sure to explain why your example is a data race, and how ThreadSanitizer catches the data race.

## Non-blocking Operations

You may have noticed that if you make a blocking call in one of your user-level threads, the thread will just block and not yield control to other runnable user-level threads. A real-world user-level thread library would be useless if it could not handle blocking calls.

In this task, your job is to implement a non-blocking mechanism for a system call like `read`. One possible implementation is to have the user-level thread cooperatively tell the scheduler thread that it wants to issue a blocking call, so the scheduler can hand off the work to some background kernel thread, and schedule other work onto the thread that would have been blocked. Another possible implementation is to have the thread receive a handler from the scheduler, and use this handler to query the status of the blocking call. Or another possible implementation is to have the thread pass a callback function to the scheduler when issuing the call. You don't need to implement all of these. Just pick the one you feel most comfortable writing I/O code in.

What happens if the user-level thread is not being cooperative, and makes a system call directly? You could still intercept the call, using techniques like `strace`, and trace it back to the caller. `strace` relies on `ptrace`, an incredibly useful system call for debuggers and tracers, to capture events. You should use this mechanism to augment the scheduler to automatically detect when user-level threads are not being cooperative and making blocking calls directly.

In `extra_credit.txt`, write up what you did, and what design decisions you made. How efficient are the non-blocking I/O calls? How useful is the detection tool? Also remember to write a small example demonstrating the functionality you wrote.