Contents

Project 2 -- thread library

Worth: 15 points
Assigned: September 28, 2022
Due: October 21, 2022

1. Overview

This project will help you understand how threads and monitors are implemented. In this project, you will implement a thread library similar to the one provided in Project 1.

2. Interface to applications

This section describes the interface that your thread library and the infrastructure provide to applications. The interface consists of four classes: `cpu`, `thread`, `mutex`, and `cv` (no `semaphore`), which are declared in `cpu.h`, `thread.h`, `mutex.h`, and `cv.h` (**do not modify these files**). The interface to these classes is the same as the one in Project 1, except for the two differences described below. Because of these differences, Project 1 and 2 use different versions of `cpu.h` and `thread.h`.

- The `cpu::boot` function takes different parameters than in Project 1. These allow the program to specify the number of CPUs, and customize how interrupts are generated.

  ```
  static void boot(unsigned int num_cpus, thread_startfunc_t func, void *arg,
                   bool async, bool sync, int random_seed);
  ```

  The parameter `num_cpus` specifies the number of CPUs that this execution of the library should support. **Core** projects need support only one CPU. **Advanced** projects must support multiple CPUs.

  The parameters `func` and `arg` specify the body of the first thread, and the argument to pass to that body, respectively. When the library is simulating multiple CPUs, one CPU will be chosen by the infrastructure as the starting location for that thread's execution.

  The three parameters `sync`, `async`, and `random_seed` control the behavior of timer interrupts. Timer interrupts are a per-CPU mechanism. When delivered, the OS should pre-empt the thread currently running on that CPU, scheduling the next ready thread if one exists. If no other ready thread exists, the current thread should continue running without being pre-empted.

  - If `sync` is true, timer interrupts will be generated at points in the program that are synchronized to executing instructions.
  - If `async` is true, timer interrupts will be generated periodically, approximately once every millisecond.
  - The parameter `random_seed` controls the pattern of synchronous interrupts. Each value for random_seed potentially generates a different but repeatable pattern of interrupts. This is particularly helpful for re-creating a problem during testing.

  Note that it is possible to enable both synchronous and asynchronous interrupts. The `random_seed` parameter only affects the behavior of synchronous interrupts.

- The `thread` class provides an extra function `thread::yield()`.

  ```
  static void yield();
  ```

  A thread invokes `thread::yield()` to voluntarily relinquish its CPU to the next ready thread if one exists; it does nothign if there are no ready threads. This is particularly helpful in testing to force a particular pattern of execution amongst concurrent threads in the absence of interrupts. Note that `thread::yield` is a `static` member function and is invoked on the `thread` class, not on an instance of that class.

3. Thread library

You will write a thread library that implements the functions in `thread.h`, `mutex.h`, and `cv.h`. You will also implement the `cpu::init` function, which is called by the infrastructure when it starts a CPU.

3.1. Initializing and identifying a CPU

Your thread library will provide the implementation of `cpu::init`. This is the only member of the `cpu` class that your library implements.

When `cpu::boot` is called, the infrastructure instantiates `num_cpus` objects of type `cpu` and invokes the `init` method on each:

```
void init(thread_startfunc_t body, void *arg);
```

One of those invocations will be passed the `body` and `arg` parameters that were passed to `cpu::boot`. The others will be invoked with `nullptr` for both `body` and `arg`. `cpu::init` should cause this CPU to run threads as they become available. Additionally, the invocation with a non-`NULL` body and arg should create a thread that executes `body(arg)`. You may not assume any particular order in which these invocations happen.

`cpu::init` does not return. If it fails, it should throw an appropriate exception, e.g., `std::bad_alloc` if it can't allocate memory.

Your thread library may find it helpful to identify the "current" CPU. The function `cpu::self` returns a pointer to the `cpu` object for the CPU executing the calling thread. Note that `cpu::self` is a `static` member function and is invoked on the `cpu` class, not an instance of that class.

3.2. Creating and swapping threads

You will be implementing your thread library on x86 PCs running the Linux operating system. Linux provides three functions in `ucontext.h` to help implement user-level thread libraries: `makecontext`, `setcontext`, and `swapcontext`. **Note:** our infrastructure provides wrappers to the standard implementations of these to simplify some things for you.

- The `ucontext_t` structure is capable of representing the state of a paused or blocked thread.
- `makecontext` initializes a new thread context.

  ```
  void makecontext(ucontext_t *ucp, void (*func)(), int argc, ...);
  ```

  The parameter `ucp` must point to an allocated object of type `ucontext_t`. The parameter `func` is a pointer to the body that this context should begin executing when it is made active. `argc` specifies how many arguments `body` is expecting, followed by that list of zero or more arguments.

  Here is some example code which shows how to initialize a thread context to represent a new thread. This new thread will use a newly-created stack, and will begin executing a function called `start` that expects three arguments, `arg1`, `arg2`, and `arg3`.

  ```
  /*
   * Direct the new thread to use an allocated stack.  Your thread library
   * should allocate but not initialize STACK_SIZE bytes for each thread's
   * stack.
   */
  char *stack = new char [STACK_SIZE];
  ucontext_ptr->uc_stack.ss_sp = stack;
  ucontext_ptr->uc_stack.ss_size = STACK_SIZE;
  ucontext_ptr->uc_stack.ss_flags = 0;
  ucontext_ptr->uc_link = nullptr;

  /*
   * Set up the new thread to start by calling start(arg1, arg2, arg3).
   */
  makecontext(ucontext_ptr, (void (*)()) start, 3, arg1, arg2, arg3);
  ```

  Note that the manual page for `makecontext` specifies that the arguments to the thread body should be integers for portability. However, it is safe for them to be pointers on our infrastructure. The manual page also specifies that `ucp` must be initialized with `getcontext`; that too is unnecessary in our infrastructure. The wrapper we provide for this routine takes care of both of these for you.

  Hint: resist the urge to assign a non-null value to `uc_link`. It does not solve the problem you think it will solve. Solve that problem some other way.

- `swapcontext` saves the current context and switches to another.

  ```
  int swapcontext(ucontext_t *oucp, const ucontext_t *ucp);
  ```

  `swapcontext` saves the current CPU registers to the context structure pointed to by `oucp`, and loads the saved CPU registers from the context structure pointed to by `ucp`. This effectively pauses the running thread and makes the second context the active thread on this processor. The context in `ucp` must have previously been saved (via `swapcontext`) or constructed (via `makecontext`). This function does not return immediately, but may appear to return if and when the saved context in `oucp` is later activated.

- `setcontext` switches to the context of another thread.

  ```
  int setcontext(const ucontexzt_t *ucp);
  ```

  `setcontext` restores the context structure pointed to by `ucp` to the CPU registers, effectively switching to that context. The context must have previously been saved (via `swapcontext`) or constructed (via `makecontext`). Unlike `swapcontext`, `setcontext` does not save the current running context.

- The `ucontext` facility provides one additional function: `getcontext` saves the context of the current thread.

  ```
  int getcontext(ucontext_t *ucp);
  ```

  It copies the current CPU registers to the context structure pointed to by `ucp`, which must point to an allocated object of type `ucontext_t`. **Hint**: This function is not particularly useful. To see why, think about what happens to the program counter of the current thread after it calls `getcontext`.

3.3. Thread lifetimes

A *thread object* is the data structure (of type `thread`) that represents a thread in an application program. When a thread object is instantiated, it creates a corresponding *stream of execution* that comprises the running thread. The one exception to this is the stream of execution representing the thread created by the call to `cpu::boot`. That thread has no corresponding application-visible thread object, and its stream of execution should begin running once its CPU is initialized. Any other stream of execution is made ready when its corresponding thread object is instantiated; it may begin running upon instantiation or anytime thereafter, depending on the current workload and (for **Advanced** projects) number of available CPUs.

A stream of execution completes when the thread returns from the function that was specified as the thread's body. Soon after the stream of execution completes, the thread library should deallocate the memory used for the thread's stack and TCB.

A thread object is destroyed according to the normal rules of object destruction: through explicit deallocation (if dynamic) or when the thread object goes out of scope (if static). The thread object may be destroyed **before or after** the stream of execution completes, and the destruction of the thread object should not affect the stream of execution (or vice versa).

If the thread object is destroyed before the stream of execution completes, that stream of execution should continue. If the stream of execution completes before the thread object is destroyed, the memory used for the thread's stack and TCB should be deallocated, but the thread object should still be valid. For example, another thread in the application might still `join` with the (now completed) thread.

3.4. Interrupts

The infrastructure we provide supports the use of interrupts. To help ensure atomicity of multiple operations, your thread library will manipulate interrupts using functions provided by the `cpu` class. **Advanced** projects will additionally use these routines to help manage multiple CPUs efficiently.

The infrastructure provides two types of interrupts: *timer interrupts* and *inter-processor interrupts* (IPI). Timer interrupts are generated periodically by the infrastructure, provided they were enabled by `cpu::boot`. Inter-processor interrupts are received by one CPU when another CPU calls `cpu::interrupt_send`. **Core/Advanced**: Core thread libraries must handle timer interrupts. Only Advanced thread libraries need to consider IPIs.

Interrupts can only be received when interrupts are enabled. When a CPU receives an interrupt, the infrastructure consults the `cpu::interrupt_vector_table` for that CPU and calls the interrupt handler that is registered for that type of interrupt. The interrupt handler begins running with interrupts (still) enabled. At the time `cpu::init` is called, interrupts are disabled.

Your thread library is responsible for setting the required entries in the interrupt vector table. `cpu::interrupt_vector_table[TYPE]` specifies the address of the function to call when the specified CPU receives interrupt `TYPE` (`TYPE` can be `cpu::TIMER` or `cpu::IPI`).

- `cpu::interrupt_disable`

  ```
  static void interrupt_disable();
  ```

  `cpu::interrupt_disable` atomically disables interrupts on the CPU executing the invoking thread. This is a static member function.

- `cpu::interrupt_enable`

  ```
  static void interrupt_enable();
  ```

  `cpu::interrupt_enable` atomically enables interrupts on the CPU executing the invoking thread. This is a static member function. Recall that disable/enable pairs cannot be nested; a single call to `enable` enables interrupts no matter how many preceding calls to `disable` have been made.

- `cpu::interrupt_enable_suspend`

  ```
  static void interrupt_enable_suspend();
  ```

  `cpu::interrupt_enable_suspend` atomically (1) enables interrupts on the CPU executing the calling thread and (2) suspends that CPU until it receives an inter-processor interupt (IPI) from another CPU. This is a static member function.

- `cpu::interrupt_send()`

  ```
  void interrupt_send();
  ```

  `cpu::interrupt_send` posts an inter-processor interrupt to the CPU object on which this method is invoked.

Remember that interrupts should be disabled only when executing in your thread library's code. The code outside your thread library should **never** execute with interrupts disabled.

3.5. `cpu::guard`

Advanced. The infrastructure provides a single guard variable (`cpu::guard`) that your thread library must use to provide mutual exclusion on multiprocessors. Remember that the switch invariant for multiprocessors specifies that `cpu::guard` must be `true` when calling `swapcontext` or `setcontext`, and must be set to false after resuming from a stored context but before returning to user code. `cpu::guard` is initialized to `false`.

You may use any of the functions in `std::atomic` to manipulate `cpu::guard`. **Hint**: You are likely to find `store` and `exchange` most useful. Beware of using the `load` function--it usually leads to a race condition.

3.6. Efficiency

Your thread library should manage the CPUs efficiently. Minimize the number of calls to `swapcontext` and `setcontext`, and minimize busy waiting (though some busy waiting in the thread library is inevitable when running on a multiprocessor). Suspend a CPU (using `cpu::interrupt_enable_suspend`) when there are no runnable threads.

When all CPUs are suspended, the infrastructure will exit the process with the message:

```
All CPUs suspended.  Exiting.
```

3.7. Scheduling order

This section describes the specific scheduling order that your thread library should follow. Note that the thread library provided in Project 1 does not guarantee this scheduling order, since that thread library was provided for developing concurrent programs; for a concurrent program to be considered correct, it must work for any scheduling order.

All scheduling queues should be FIFO. This includes the ready queue, the queue of threads waiting for a mutex, the queue of threads waiting on a condition variable, and the queue of threads waiting for a thread to exit. All CPUs should share a single ready queue. Mutexes should be acquired by threads in the order in which the mutex was requested (by `mutex::lock` or in `cv::wait`). **Note** that this requires *handoff locks*, where an unlocking thread grants the lock to the first waiting thread, if there is one. Be sure to use the right algorithm from lecture!

A thread does not yield its CPU when: creating a thread, unlocking a mutex, or calling signal/broadcast on a condition variable. The created or unblocked thread(s) should be put on the ready queue. Each thread on the ready queue should be executed when (a) a CPU becomes available and (b) it is at the head of the queue.

When a thread wakes up in `cv::wait`, it is that thread's responsibility to request the mutex when it next runs.

Your implementation of `cv::wait` should not experience any spurious wakeups.

3.8. Error handling

Operating system code should be robust to bad user programs. These fall into four categories:

- **Questionable style.** A user program may not be written in the style we recommend for concurrent programs, but it may still lie within the bounds of how a program is allowed to use the thread library without causing an exception. Here are a few examples; none of these should be considered an error for Project 2.
  - Calling `thread::yield`.
  - Signaling without holding the mutex (some may not even consider this poor style).
  - Deadlock. Even trivial deadlocks are legal, such as a thread trying to acquire a mutex it has already locked, or a thread trying to `join` with itself.
  - Calling `cv::wait` without checking a condition in a `while` loop.
  - Calling `cv::wait` on one cv with different mutexes.
  - A thread that exits while holding a mutex (the mutex stays locked).
- **Misuse of thread functions** A user program may attempt to use a thread function in a way that is explicitly disallowed. In particular, a thread may try to release a mutex it is not holding. Your thread library should detect such misuses and throw a `std::runtime_error` exception.
- **Resource exhaustion**. The process may exhaust a resource needed by the OS. In this project, the main resource used by the OS is memory. If the thread library is unable to service a request due to lack of memory, it should throw a `std::bad_alloc` exception.
- **Undefined behavior** Some programming errors cause undefined behavior in C++, e.g., using uninitialized variables or pointers to free memory. Examples in this project include creating a thread with a bad function pointer (e.g., `nullptr`) and destroying a thread object while it is still in use by `thread::join`. Your thread library may behave arbitrarily for such errors. In other words, you need not worry about them.

All programs with well-defined behavior (i.e., all but the last category in the list above) can and should be used when testing your thread library, and your thread library should produce well-defined results for them.

The other source of errors are bugs in the OS code itself (in this case, your thread library). While developing the OS, the best behavior in this case is for the OS to detect the bug quickly and assert (this is called a *panic* in kernel parlance). These error checks are essential in debugging concurrent programs, because they help flag error conditions early. **Use assertion statements copiously in your thread library to check for bugs in your code (for reference, 1/8 of the lines of code in the solution thread library are devoted to assertions).**

To make it easier for you to check for errors related to interrupts, the infrastructure provides two functions, `assert_interrupts_disabled` and `assert_interrupts_enabled` that your thread library can call to check that the status of interrupts is as you expect.

Hint: Autograder test cases 19-21 check how well your thread library handles errors.

3.9. Managing `ucontext` structs

Do not initialize a `ucontext` struct by copying another `ucontext` struct. Always initialize a `ucontext` struct through `makecontext`*. Allocate `ucontext` structs dynamically (e.g., via `new`) and manage them by passing or storing pointers to the `ucontext` struct, or by passing/storing pointers to structs that contain a `ucontext` struct. Pointers allow the original `ucontext` struct to never be copied.

Why is it a bad idea to copy a `ucontext` struct? The reason is that you don't know what's in a `ucontext` struct. Byte-for-byte copying (e.g., using `memcpy`) can lead to errors unless you know what's in the struct you're copying. In particular, a `ucontext` struct happens to contain a pointer to one of its data members. If you copy a `ucontext` using `memcpy`, you will copy the value of this pointer, and the new copy will point to the **old** copy's data member. If you later deallocate the old copy (e.g., if it was a local variable), then the new copy will point to garbage.

Unfortunately, it is rather easy to accidentally copy `ucontext` structs. Some of the common ways are:

- passing a `ucontext` by value into a function
- copying the `ucontext` struct into an STL queue
- declaring a local `ucontext` variable is almost always a bad idea, since it practically forces you to copy it.

For the same reason, moving a `ucontext` struct in memory is also dangerous. If you use an STL class to allocate a `ucontext` struct, make sure that STL class doesn't move its objects around in memory. E.g., using vector to allocate `ucontext` structs is a bad idea, because vectors will move memory around when they resize.

*: **Hint (Advanced)**: For multiprocessor thread libraries, there are some meaningful contexts that begin running before your library has a chance to create any. To see why, think about how `cpu::init` is called for each CPU.

3.10. Opaque pointers

3.1. Opaque pointers

You may not modify or rename the header files included in this handout. However, you will probably need to add data and functions for the classes declared in the headers (`cpu` , `thread` , `mutex` , `cv`). We use the *opaque pointer* idiom (sometimes called *pimpl*, for "pointer to implementation") to allow you to add data/functions for a class without changing that class's header file (or recompiling `libcpu.o`).

The `cpu` , `thread` , `mutex` , `cv` classes each provide an `impl_ptr` member, which points to an instance of class `impl` . For example, class `thread` provides `thread::impl_ptr` , which points to an instance of class `thread::impl` .

You may define each `impl` class and use each `impl_ptr` however you like. For example, you can store custom data and functions you need for a `mutex` in an instance of the `mutex::impl` class, then point to this instance via a mutex's `impl_ptr` .

Typically, a class constructor will allocate the `impl` object and initialize the `impl_ptr` to point to the allocated data. For the `cpu` class, do these tasks in `cpu::init` , since you're not writing the `cpu` class constructor.

4. Example application

Here is a short program that uses the above thread library, along with the output generated by the program. Make sure you understand how the CPU is switching between two threads while they're executing the `loop` function. `i` is on the stack and so is private to each thread. `g` is a global variable and so is shared among the two threads.

```
#include <iostream>
#include <cstdlib>
#include "thread.h"

using std::cout;
using std::endl;

int g = 0;

mutex mutex1;
cv cv1;

void loop(void *a)
{
    char *id = (char *) a;
    int i;

    mutex1.lock();
    cout << "loop called with id " << id << endl;

    for (i=0; i<5; i++, g++) {
        cout << id << ":\t" << i << "\t" << g << endl;
        mutex1.unlock();
        thread::yield();
        mutex1.lock();
    }
    cout << id << ":\t" << i << "\t" << g << endl;
    mutex1.unlock();
}

void parent(void *a)
{
    intptr_t arg = (intptr_t) a;

    mutex1.lock();
    cout << "parent called with arg " << arg << endl;
    mutex1.unlock();

    thread t1 ( (thread_startfunc_t) loop, (void *) "child thread");

    loop( (void *) "parent thread");
}

int main()
{
    cpu::boot(1, (thread_startfunc_t) parent, (void *) 100, false, false, 0);
}
```

```
parent called with arg 100
loop called with id parent thread
parent thread:  0       0
loop called with id child thread
child thread:   0       0
parent thread:  1       1
child thread:   1       2
parent thread:  2       3
child thread:   2       4
parent thread:  3       5
child thread:   3       6
parent thread:  4       7
child thread:   4       8
parent thread:  5       9
child thread:   5       10
All CPUs suspended.  Exiting.
```

5. Tips

Start by implementing `cpu::init`, `thread::thread`, and `thread::yield`. Don't worry at first about atomicity in the thread library or supporting multiple processors, but you must ensure that interrupts are enabled whenever user code is running. After you get that system working, implement the other thread library functions.

Hint: Remember that switching between threads must always be mediated by OS code--namely, something you write in your thread library. This is true for any switch, whether it happens within a thread, after a thread ends, or even before it begins. As discussed in lecture, this has particular implications for how threads are created.

Next, add calls to `cpu::interrupt_disable` and `cpu::interrupt_enable` to ensure your library works in the presence of interrupts. You'll need to think about what should happen when an interrupt occurs and how to guarantee atomicity in your thread library when they occur. Finally, **advanced** projects should add support for multiple processors. Use `cpu::guard` to ensure atomicity for multiprocessors, and think about what a CPU should do when there are no runnable threads.

Hint (advanced): Think carefully about how to put a processor to sleep. In particular, think about why it cannot be done directly by the thread context that is ending/blocking. The **hint** at the end of Section 3.9 is relevant to solving this problem.

For full credit, you should use RAII to manage interrupts and, for **advanced** projects, the guard. As discussed in lecture, the RAII model is particularly helpful when functions might terminate either via normal return or thrown exceptions. Note that there may be some places where you have to reach inside the RAII abstraction and manipulate interrupts or the guard directly; you might even want to build such facilities into your wrapper.

We recommend the use of managed pointers for this project. This will require a bit more investment on your part, but we believe this investment will be repaid in fewer/easier to find bugs.

6. Test cases

An integral (and graded) part of writing your thread library will be to write a suite of test cases to validate any thread library. This is common practice in the real world--software companies maintain a suite of test cases for their programs and use this suite to check the program's correctness after a change. Writing a comprehensive suite of test cases will deepen your understanding of how to use and implement threads, and it will help you a lot as you debug your thread library.

Each test case for the thread library will be a short C++ program that uses functions in the thread library (e.g., the example program in Section 4). The name of each test case should start with `test` and end with `.cc` or `.cpp`, e.g., `test1.cpp`.

Each test case should be run without any arguments and should not use any input files. Test cases should exit(0) when run with a correct thread library (normally this will happen when your test case's last runnable thread ends or blocks).

The test cases you submit should call `cpu::boot` with `num_cpus=1` and without enabling asynchronous or synchronous timer interrupts. All the instructor's buggy thread libraries can be exposed with a single CPU and without timer interrupts. **Note:** one consequence of this is that a "complete" test suite cannot be used to verify that your own library is correct, at least because it cannot use interrupts.

Your test suite may contain up to 22 test cases. Each test case must generate less than 1 MB of output, use less than 100 MB of memory, and take less than 60 seconds to run. These limits are much larger than needed for full credit. You will submit your suite of test cases together with your thread library, and we will grade your test suite according to how thoroughly it exercises a thread library. Section 9 describes how your test suite will be graded.

When writing test cases for submission, think carefully about how to exercise the various parts of your thread library that are not related to either interrupts or multiple CPUs. For **expected** test cases, you will want to exercise simple operations with monitors: thread creation and destruction, threads yielding to one another, contending for locks in critical sections, and using condition variables for happens-before constraints. For **edge** test cases, look especialy at the first three categories of Section 3.8, which gives a partial list of interesting edge cases. For each of these cases, write down what you think the correct answer *should* be, and then see if your thread library matches your expectation.

The autograder is able to (and will) use both synchronous and asynchronous interrupts in testing *your* solution. Therefore you should write and run such test cases yourself, even though you can't submit them. **Expected** test cases might include a set of threads with long runtimes that don't voluntarily yield the CPU. Have these be concise enough that you should know what an *incorrect* execution might look like. You should also employ **stress** tests that have many threads running for long periods of time, perhaps exercising many of the thread facilities to give you confidence that your implementation is correct. **Hint**: It is very hard to test all possible combinations of thread library code and interrupts. You should *also* inspect your individual routines to make sure they are obeying the switch invariant. **Hint:** When reading your library routines, imagine what happens if an interrupt is delivered *immediately before* calling `interrupt_disable()` and *immediately after* returning from `interrupt_enable` or (for **advanced** projects) `interrupt_enable_suspend`.

There are some test cases that your submission **cannot** pass unless it has passed some earlier test case(s). In those cases, the line number on which your submission is reported as failing is even less informative than it might otherwise be. Therefore, resist trying to infer what the bug is by the reported line number at which you are failing. For **advanced** libraries, the autograder is able to (and will) use multiple CPUs. Therefore, you should write and run such test cases yourself. **Expected** test cases might include a set of threads that grows and shrinks over time. **Stress** tests are probably even more important for **advanced** libraries. Likewise it is probalby even more importnat to consider interrupts immediately before a call to `interrupt_disable` and immediately after a call to either `interrupt_enable` or `interrupt_enable_suspend`.

7. Project logistics

Write your thread library in C++17 on Linux. Use the default CAEN compiler (with -std=c++17) to compile your programs. As of this writing, the default version is 8.5.0, on RHEL 8. This is also the compiler and OS version used by the autograder, but we cannot guarantee identical behavior for thread libraries containing code with undefined meaning.

You may use any functions included in the standard C++ library, except the C++ thread facilities. You should not use any libraries other

than the standard C++ library. Your thread library code may be in multiple files. Each file name must end with `.cc`, `.cpp`, or `.h` and must not start with `test`.

To avoid conflicting with the C++ thread facilities, do not specify "using namespace std;" in your program.

This Makefile shows how to compile your thread library and an application that uses the thread library (adjust the file names in the Makefile to match your own program).

You are required to document your development process by having your Makefile run `autotag.sh` each time it compiles your thread library (see Makefile above). `autotag.sh` creates a git tag for a compilation; which helps the instructors better understand your development process. `autotag.sh` also configures your local git repo to include these tags when you run "`git push`". To use it, download `autotag.sh` and set its execute permission bit (run "`chmod +x autotag.sh`"). If you have several local git repos, be sure to push to github from the same repo in which you compiled your thread library.

When running `gdb`, you will probably find it useful to direct `gdb` to ignore `SIGUSR1` events (used by the project infrastructure). To do this, use the following command in `gdb`:

```
handle SIGUSR1 nostop noprint
```

The infrastructure simulates multiple CPUs as multiple POSIX threads. You can view and switch among these simulated CPUs via the following commands:

```
info threads
thread <N>          (Note: <N> is the ID of the thread you want to view in gdb.  Thread ID 1 is used by the infrastruc
```

Debugging print statements made by the thread library when interrupts are disabled should use `printf`, not `cout`. This avoids deadlocking with user code (which should use `cout`).

We have created a private github repository for your group (`eecs482/<group>.2`), where `<group>` is the sorted, dot-separated list of your group members' uniqnames. Initialize your local repository by cloning the (empty) repository from github, e.g.,

```
git clone git@github.com:eecs482/uniqnameA.uniqnameB.2
```

8. Grading, auto-grading, and formatting

To help you validate your programs, your submissions will be graded automatically, and the results will be provided to you. You may then continue to work on the project and re-submit. The results from the auto-grader will not be very illuminating; they won't tell you where your problem is or give you the test programs. The main purpose of the auto-grader is to help you know to keep working on your project (rather than thinking it's perfect and ending up with a 0). The best way to debug your program is to generate your own test cases, figure out the correct answers, and compare your program's output to the correct answers. This is also one of the best ways to learn the concepts in the project.

The student suite of test cases will be graded according to how thoroughly they test a thread library using a single CPU and no interrupts. We will judge thoroughness of the test suite by how well it exposes potential bugs. The auto-grader will first compile a test case with a correct thread library and generate the correct output (on `stdout`, i.e., the stream used by `cout`) for this test case. Test cases should not cause any compile or run-time errors when compiled with a correct thread library. The auto-grader will then compile the test case with a set of buggy thread libraries. A test case exposes a buggy thread library by causing it to generate output (on `stdout`) that differs from the correct output. The test suite is graded based on how many of the buggy thread libraries were exposed by at least one test case. This is known as *mutation testing* in the research literature on automated testing.

You may submit your program as many times as you like, and all submissions will be graded and cataloged. We will use your highest-scoring submission, with ties broken in favor of the later submission. If any group member is in EECS 498-002, your highest-scoring submission will be chosen using a (2/3,1/3) weighted average of your core and advanced scores.

You must recompile and `git push` at least once between submissions.

The auto-grader will provide feedback for the first submission of each day, plus 3 bonus submissions over the duration of this project. Bonus submissions will be used automatically--any submission you make after the first one of that day will use one of your bonus submissions. After your 3 bonus submissions are used up, the system will continue to provide feedback for the first submission of each day.

Because you are writing concurrent programs, the auto-grader may return non-deterministic results. In particular, test cases 30-44 are non-deterministic.

Because your programs will be auto-graded, you must be careful to follow the exact rules in the project description. In particular:

- Your thread library should not generate any output. Only the program using your thread library should generate output.
- Do not modify or rename the header files included in this handout.

In addition to the auto-grader's evaluation of your program's correctness, a human grader will evaluate your program on issues such as documentation, coding style, the efficiency, brevity, and understandability of your code, compiler warnings, etc.. Your final score will be the product of the hand-graded score (between 1-1.12) and the auto-grader score.

9. Turning in the project

Submit the following files for your thread library:

- C++ files for your thread library. File names should end in `.cc`, `.cpp`, or `.h` and must not start with `test`. Do not submit the files provided in this handout.
- Suite of test cases. Each test case should be in a single file. File names should start with `test` and end with `.cc` or `.cpp`.

The official time of submission for your project will be the time of your last submission. There is a strict deadline for full credit on the project: Midnight EDT on the day the project is due. Students who have obtained less than 80% at that point (on either the Core or Advanced portion) may continue working on that portion of the project for up to one week to improve their score on that portion to a maximum of 80%.

10. Files included in this handout (zip file)

- `cpu.h`
- `cv.h`
- `libcpu.o`
- `mutex.h`
- `thread.h`
- `autotag.sh`
- `Makefile`

11. Experimental platforms

The files provided in this handout were compiled on RHEL 8. They should work on most other Linux distributions (e.g., Ubuntu) and on Windows Subsystem for Linux (WSL), but these are not officially supported.

We also provide an experimental version of the infrastructure for students who want to develop on MacOS (11.0 or later, including experimental support for ARM-based Macs). If you are developing on MacOS:

- Use `libcpu_macos.o` instead of `libcpu.o`.
- Add `-D_XOPEN_SOURCE` to the compilation flags.