Project 2 -- thread library

Worth: 15 points
Assigned: Tuesday, September 24, 2013
Due: 6:00 pm, Thursday, October 17, 2013

1. Overview

This project will help you understand how threads and monitors are implemented on uniprocessor and multiprocessor systems. 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`, which are declared in `cpu.h`, `thread.h`, `mutex.h`, and `cv.h` (do not modify these files). The interface is the same as the one provided 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, which allow the program to specify the number of CPUs and customize how interrupts are generated. One CPU will create the first thread, which is initialized to call the function pointed to by `func` with the argument `arg`. `async`, `sync`, and `random_seed` control how timer interrupts are generated. As discussed in class, a timer interrupt preempts the thread running on the current CPU and starts the next ready thread. If `sync` is true, timer interrupts will be triggered at points in the program that are synchronized to executing instructions. You can generate different (but repeatable) patterns of synchronous timer interrupt by changing `random_seed`. If `async` is true, a timer interrupt will be triggered on each CPU about once each millisecond. A user program should call `cpu::boot` exactly once (before calling any other thread functions). On success, `cpu::boot` does not return.

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

- The `thread` class provides an extra function `thread::yield()`, which allows a thread to voluntarily give up the CPU to the next runnable thread (if there is one).

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

  Note that `thread::yield` is a `static` member function and is invoked on the `thread` class (not on an instance of the `thread` 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 a function `cpu::init` (this is the only member of the `cpu` class that you will implement). When the user program calls `cpu::boot`, the infrastructure activates `num_cpus` CPUs and causes each CPU to execute `cpu::init`. One of the CPUs will call `cpu::init` with the arguments (`func` and `arg`) passed to `cpu::boot`; the other CPUs will call `cpu::init` with `nullptr` argument values.

`cpu::init` should cause the CPU to run threads as they become available. On success, `cpu::init` should not return to the caller; on failure, `cpu::init` should throw an appropriate exception (e.g., `std::bad_alloc` if it can't allocate memory).

If `cpu::init` is passed a function `func` that is not `nullptr`, it should also create a thread that executes `func(arg)`.

Your thread library may find it useful to identify which CPU a thread is running on. The function `cpu::self` returns a pointer to the `cpu` object for the CPU this thread is running on. Note that `cpu::self` is a `static` member function and is invoked on the `cpu` class (not on an instance of the `cpu` class).

3.2. Creating and swapping threads

You will be implementing your thread library on x86 PCs running the Linux operating system. Linux provides some library functions (`getcontext`, `makecontext`, `swapcontext`) to help implement user-level thread libraries.

Here's an example of how to use these functions to create a new thread (read the Linux manual pages for details):

```
#include <ucontext.h>

/*
 * Initialize a context structure by copying the current thread's context.
 */
getcontext(ucontext_ptr);           // ucontext_ptr has type (ucontext_t *)
```

```
/*
 * Direct the new thread to use a different stack.  Your thread library
 * should allocate 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;

/*
 * Direct the new thread to start by calling start(arg1, arg2).
 */
makecontext(ucontext_ptr, (void (*)()) start, 2, arg1, arg2);
```

Use `swapcontext` to save the context of the current thread and switch to the context of another thread.

3.3. Thread, mutex, and condition variable lifetimes

This section describes two subtle points related to lifetimes of objects.

First, consider that a thread and the object representing that thread are different concepts and have different lifetimes. The thread is a stream of execution; the thread object is a data structure that represents the thread.

A *thread* finishes executing when it returns from the function that was specified in the thread constructor. Soon after the thread finishes executing, the thread library should deallocate the memory used for the thread's stack space and context.

A *thread object* is destroyed according to the normal rules of object destruction. E.g., if the thread object is a local variable of a block, the thread object is destroyed when that block finishes.

The thread object may be destroyed **before or after** the thread finishes executing, and the destruction of the thread object should not affect the thread's execution (or vice versa). If the thread object is destroyed before the thread finishes executing, the thread should continue executing. If the thread finishes executing before the thread object is destroyed, the memory used for the thread's stack and context should be deallocated, but the thread object should not be destroyed (e.g., it could still be used to `join` with the completed thread).

Second, consider when the constructor for a thread, mutex, or condition variable may be called. Threads are *not* allowed to be constructed before the system is initialized (i.e., before `cpu::boot` has been called). However, mutexes and condition variables *are* allowed to be constructed before `cpu::boot` is called (they are often global variables, which are constructed before `main` is called; see Section 4 for an example). This implies that the constructors for mutexes and condition variables must not depend on any cpu or thread library data. In particular, they should be able to work without manipulating interrupts.

3.4. Interrupts

There are two types of interrupts: timer interrupts and inter-processor interrupts (IPI). Timer interrupts are generated periodically by the infrastructure (if they are enabled by `cpu::boot`). Inter-processor interrupts are received by one CPU when another CPU calls `cpu::interrupt_send`. 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 (if interrupts had been disabled, the interrupt would not have been received).

Your thread library is responsible for setting the 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 `TIMER` or `IPI`).

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. E.g., the body of a monitor must run with interrupts enabled and use locks to implement mutual exclusion.

To help ensure atomicity of multiple operations, your thread library will manipulate interrupts using functions provided by the `cpu` class. `interrupt_send` is a normal member function and sends an IPI to the specified CPU. `interrupt_disable`, `interrupt_enable`, and `interrupt_enable_suspend` are `static` member functions; they affect the CPU that called these functions and are invoked on the `cpu` class (not on an instance of the `cpu` class).

- `cpu::interrupt_disable` disables interrupts on the executing CPU.

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

- `cpu::interrupt_enable` enables interrupts on the executing CPU.

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

- `cpu::interrupt_enable_suspend` atomically enables interrupts on the executing CPU and suspends the executing CPU until it receives an inter-processor interrupt (IPI) from another CPU.

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

- `cpu::interrupt_send()` sends an inter-processor interrupt to the specified CPU.

```
    void interrupt_send();
```

3.5. Guard variable

The infrastructure provides an atomic `guard` variable, which thread libraries should use to provide mutual exclusion on multiprocessors. Remember that the switch invariant for multiprocessors specifies that this guard variable must be `true` when calling `swapcontext`. guard is initialized to `false`.

You may use any of the functions in `std::atomic` to manipulate `guard` (e.g., `store`, `exchange`). The `std::atomic` class doesn't provide a test-and-set function, but you can easily achieve the same effect with other functions.

Hint: Beware of using the `load` function--it usually leads to a race condition.

3.6. Efficiency

Your thread library should manage the CPUs efficiently. In particular, your thread library should minimize busy waiting (though some busy waiting in the thread library is inevitable when running on a multiprocessor). In particular, you should 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 (which should 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`).

When a thread creates a thread, unlocks a mutex, or signals/broadcasts a condition variable, the caller does not yield its CPU. The created or unblocked thread should be put on the ready queue and executed when a CPU becomes available.

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

3.8. Error handling

Operating system code should be robust. There are three sources of errors that OS code should handle. The first source of errors is misbehaving user programs. Your thread library should detect when a user program misuses thread functions (e.g., releasing a mutex it hasn't locked) and throw a `std::runtime_error` exception.

There are certain application behaviors that are arguably errors or not. Here is a list of questionable behaviors that should **not** be considered errors: signaling without holding the lock (this is legal with Mesa monitors); a thread that exits while still holding a mutex (the mutex can never be unlocked); 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). Ask on the forum if you're unsure whether you should consider a certain behavior an error.

A second source of error comes from resources that the OS uses, such as hardware devices. 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. Applications can then catch the exception and proceed accordingly.

A third source of error is when the OS code itself (in this case, your thread library) has a bug. 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, 10% of the lines of code in my thread library are 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 20 and 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. Instead, initialize `ucontext` structs through `getcontext`/`makecontext`, and manage them by passing or storing pointers to `ucontext` structs, or by passing/storing pointers to structs that contain a `ucontext` struct (or by passing/storing pointers to structs that contain a pointer to a `ucontext` struct, but this is overkill). The 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 the case of a `ucontext` struct, it happens to contain a pointer to itself (viz. 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. Copying structs is also a bad idea for performance (the `ucontext` struct is 348 bytes on Linux/x86).

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

You should probably use the `new` operator to allocate `ucontext` structs (or the struct containing a `ucontext` struct). 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.

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 namespace std;

int g = 0;

mutex mutex1;
cv cv1;

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

    cout << "loop called with id " << id << endl;

    mutex1.lock();
    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;

    cout << "parent called with arg " << arg << endl;
    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. After you get that system working, implement the other thread functions. 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, add support for multiple processors. Use the guard variable to ensure atomicity for multiprocessors, and think about what a CPU should do when there are no runnable threads.

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.cc`.

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). If you submit your disk scheduler as a test case, remember to adapt its call to `cpu::boot` and to specify all inputs (number of requesters, buffers, and the list of requests) statically in the program. The list of requests should be short to make a good test case (i.e., one that you can trace through what should happen).

The test cases you submit should call `cpu::boot` with `num_cpus=1` and without enabling asynchronous or synchronous timer interrupts. All buggy thread libraries can be exposed with a single CPU and without timer interrupts.

While you cannot submit test cases that use multiple CPUs or enable timer interrupts, you **should** write and run such test cases yourself.

Your test suite may contain up to 22 test cases. Each test case may generate at most 10 KB of output and must 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.

7. 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.

The `cpu`, `thread`, `mutex`, `cv` classes each provide an `impl_ptr` member, which points to a 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` class and initialize the `impl_ptr` to point to the allocated data.

8. Project logistics

Write your thread library in C++ on Linux. The internal functions and global variables in your thread library should be declared `static` to prevent naming conflicts with programs that link with your thread library.

Use `g++ 4.7.0` to compile your programs. To use `g++ 4.7.0` on CAEN computers, put the following command in your startup file (e.g., `~/.profile`):

```
module load gcc
```

You may use any functions included in the standard C++ library, including the STL. You should not use any libraries other than the standard C++ library. To compile an application program (e.g., `app.cc`) with your thread library (e.g., `thread.cc`), run:

```
g++ thread.cc app.cc libcpu.a -ldl -pthread -std=c++11
```

You may add options `-g` and `-Wall` for debugging and `-o` to name the executable.

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`.

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
```

9. 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. We will judge thoroughness of the test suite by how well it exposes potential bugs in a thread library. 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. However, only the feedback from the first submission of each day will be provided to you. Later submissions on that day will be graded and cataloged, but the results will not be provided to you. See the FAQ for why we use this policy.

In addition to this one-per-day policy, you will be given 3 bonus submissions that also provide feedback. These 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 1 feedback per day.

Because you are writing concurrent programs, the auto-grader may return non-deterministic results. In particular, test cases 24-38 for the thread library and test cases 6-9 for the disk scheduler use multiprocessors or asynchronous timer interrupts and are therefore 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 the clarity and completeness of your documentation, coding style, the efficiency, brevity, and understandability of your code, etc.. Your final score will be the product of the hand-graded score (between 1-1.12) and the auto-grader score.

10. 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`.

Each person should also describe the contributions of each team member using the following web form.

The official time of submission for your project will be the time of your last submission (of either project part). Submissions after the due date will automatically use up your late days; if you have no late days left, late submissions will not be counted.

11. Files included in this handout

- `cpu.h`
- `cv.h`
- `libcpu.a`
- `mutex.h`
- `thread.h`