Project 1 -- multi-threaded programming

Worth: 2 points
Assigned: Tuesday, September 10, 2013
Due: 6:00 pm, Tuesday, September 24, 2013

1. Overview

This project will give you experience writing multi-threaded programs using monitors. In this project, you will write a simple concurrent program that schedules disk requests. This concurrent program will use a thread library that we provide.

This project is to be done individually.

2. Thread library interface

This section describes the interface that the thread library and infrastructure provide to applications. You will write a multi-threaded program that uses this interface. 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). For convenience, `thread.h` includes the other header files.

To use the thread library, programs `#include thread.h` and link with `libcpu.a` and `thread.o`.

2.1. `cpu` class

The `cpu` class is declared in `cpu.h` and is used mainly by the thread library. The only part used by applications is the `cpu::boot` function:

```
static void boot(thread_startfunc_t func, void *arg, unsigned int deterministic);
```

`cpu::boot` starts the thread library and creates the initial thread, which is initialized to call the function pointed to by `func` with the single argument `arg`. A user program should call `cpu::boot` exactly once (before calling any other thread functions). On success, `cpu::boot` does not return.

`deterministic` specifies if the thread library should be deterministic or not. Setting `deterministic` to zero makes the scheduling of threads non-deterministic, i.e., different runs may generate different results. Setting `deterministic` to a non-zero value forces the scheduling of threads to be deterministic, i.e., a program will generate the same results if it is run with the same value for deterministic (different non-zero values for deterministic will lead to different results).

Note that `cpu::boot` is a `static` member function and is invoked on the `cpu` class (not on an instance of the `cpu` class).

2.2. `thread` class

The `thread` class is declared in `thread.h`. All functions throw the `std::runtime_error` exception on an error.

The constructor is used to create a new thread. When the newly created thread starts, it will call the function pointed to by `func` and pass it the single argument `arg`.

```
thread(thread_startfunc_t func, void *arg);
```

`join` causes the current thread to block until the specified thread has exited. If the specified thread has already exited, `join` returns immediately.

```
void join();
```

`self()` returns a pointer to the thread object that called `self`. If the object for the current thread has been destroyed, `self` returns `nullptr`.

```
static thread *self();
```

Note that `thread::self` is a `static` member function and is invoked on the `thread` class (not on an instance of the `thread` class).

2.3. `mutex` class

The `mutex` class is declared in `mutex.h`. All functions throw an `std::runtime_error` exception on an error.

The constructor is used to create a new mutex.

```
mutex();
```

`lock` atomically waits for the mutex to be free and acquires it for the current thread.

```
void lock();
```

`unlock` releases the mutex.

```
void unlock();
```

2.4. `cv` class

The `cv` class is declared in `cv.h`. All functions throw an `std::runtime_error` exception on an error.

The constructor is used to create a new condition variable.

```
cv();
```

`wait` atomically releases mutex `m`and waits on the condition queue.

```
void wait(mutex& m);
```

`signal` signals one of the threads on the condition queue.

```
void signal();
```

`broadcast` signals all of the threads on the condition queue.

```
void broadcast();
```

2.5. Example program

Here is a short program that uses threads.

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

using namespace std;

mutex mutex1;
cv cv1;

int child_done = 0;             // global variable; shared between the two threads.

void child(void *a)
{
    char *message = (char *) a;
    mutex1.lock();
    cout << "child run with message " << message << ", setting child_done = 1\n";
    child_done = 1;
    cv1.signal();
    mutex1.unlock();
}

void parent(void *a)
{
    intptr_t arg = (intptr_t) a;
    cout << "parent called with arg " << arg << endl;
    thread t1 ((thread_startfunc_t) child, (void *) "test message");
    mutex1.lock();
    while (!child_done) {
        cout << "parent waiting for child to run\n";
        cv1.wait(mutex1);
    }
    cout << "parent finishing" << endl;
    mutex1.unlock();
}

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

Here are the **two** possible outputs the program can generate.

```
parent called with arg 100
parent waiting for child to run
child run with message test message, setting child_done = 1
parent finishing
All CPUs suspended.  Exiting.
```

```
parent called with arg 100
child run with message test message, setting child_done = 1
parent finishing
All CPUs suspended.  Exiting.
```

3. Disk scheduler

Your task for this project is to write a concurrent program that issues and services disk requests.

The disk scheduler in an operating system receives and schedules requests for disk I/Os. Threads issue disk requests by queueing them at the disk scheduler. The disk scheduler queue can contain at most a specified number of requests (`max_disk_queue`); threads must wait if the queue is full.

Your program should start by creating a specified number of requester threads to issue disk requests and one thread to service disk requests. Each requester thread should issue a series of requests for disk tracks (specified in its input file). Each request is synchronous; a requester thread must wait until the servicing thread finishes handling that requester's last request before that requester issues its next request. A requester thread finishes after all the requests in its input file have been serviced.

Requests in the disk queue are **not** necessarily serviced in FIFO order. Instead, the service thread handles disk requests in SSTF order (shortest seek time first). That is, the next request it services is the request that is closest to its current track. The disk is initialized with its current track as 0.

Keep the disk queue as full as possible; your service thread should only handle a request when the disk queue has the largest possible number of requests. This gives the service thread the largest number of requests to choose from, which in turn helps minimize average seek distance. Note that the value of "the largest possible number of requests" varies depending on how many request threads are still active. When at least `max_disk_queue` requester threads are alive, the largest possible number of requests in the queue is equal to `max_disk_queue`. When fewer than `max_disk_queue` requester threads are alive, the largest number of requests in the queue is equal to the number of living requester threads. You will probably want to maintain the number of living requester threads as shared state.

3.1. Input

Your program will be called with several command-line arguments. The first argument specifies the maximum number of requests that the disk queue can hold. The rest of the arguments specify a list of input files (one input file per requester). I.e., the input file for requester r is argv[r+2], where 0 <= r < (number of requesters). The number of threads making disk requests should be deduced from the number of input files specified.

The input file for each requester contains that requester's series of requests. Each line of the input file specifies the track number of the request (0 to 999). You may assume that input files are formatted correctly. Open each input file read-only (**use `ifstream` rather than `fstream`**).

3.2. Output

After issuing a request, a requester thread should execute the following code (note the space characters in the strings):

```
cout << "requester " << requester << " track " << track << endl;
```

A request is available to be serviced when the requester thread prints this line.

After servicing a request, the service thread should execute the following code (note the space characters in

the strings):

```
cout << "service requester " << requester << " track " << track << endl;
```

A request is considered to be off the queue when the service thread prints this line.

Your program should not generate any other output.

The console is shared between the different threads, so your program's calls to `cout` must be protected by a monitor lock to prevent intermingling output from multiple threads.

3.3. Example input/output

Here is an example set of input files (disk.in0 - disk.in4).

disk.in0	disk.in1	disk.in2	disk.in3	disk.in4
785 53	350 914	827 567	302 230	631 11

Here is one of many possible correct outputs from running the disk scheduler with the following command (the final line of the output is produced by the thread library, not the disk scheduler):

```
disk 3 disk.in0 disk.in1 disk.in2 disk.in3 disk.in4
```

```
requester 2 track 827
requester 1 track 350
requester 3 track 302
service requester 3 track 302
requester 4 track 631
service requester 1 track 350
requester 0 track 785
service requester 4 track 631
requester 3 track 230
service requester 0 track 785
requester 1 track 914
service requester 2 track 827
requester 4 track 11
service requester 1 track 914
requester 0 track 53
service requester 3 track 230
requester 2 track 567
service requester 0 track 53
service requester 4 track 11
service requester 2 track 567
All CPUs suspended.  Exiting.
```

---

4. Project logistics

Write your disk scheduler in C++ on Linux. 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 should also set the `LD_BIND_NOW` environment variable to 1. This works around an apparent incompatibility between the thread library and the Linux dynamic linker. Add the following line to your `~/.profile`:

```
export LD_BIND_NOW=1
```

You may use any functions included in the standard C++ library (including the STL), except for C++11 threads. You should not use any libraries other than the standard C++ library (e.g., pthreads). To compile your disk scheduler (e.g., `disk.cc`), run:

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

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

Your disk scheduler 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 each student (`eecs482/<uniqname>.1`) Initialize your local repository by cloning the (empty) repositories from github, e.g.,

```
git clone git@github.com:eecs482/uniqname.1
```

5. 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 inputs. 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 inputs, understand the constraints on correct answers for these inputs, and determine if your program's output obeys these constraints. This is also one of the best ways to learn the concepts in the project.

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 5-8 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 program should print only the two items specified in Section 3.2.
- Your program should expect several command-line arguments, with the first being `max_disk_queue` and the others specifying the list of input files for the requester threads.
- Do not modify or rename the header files provided in this handout.

6. Turning in the project

Submit the following file for your disk scheduler:

- C++ files for your disk scheduler. File names should end in `.cc`, `.cpp`, or `.h` and must not start with `test`. Do not submit the files provided in this handout.

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.

7. Files included in this handout

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