Project 1 -- multi-threaded programming

Worth: 3 points
Assigned: January 24, 2024
Due: February 7, 2024

1. Overview

This project will give you experience writing multi-threaded programs using mutexes and condition variables. In this project, you will write a small concurrent program that schedules disk requests. Your 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 provides to applications. The interface consists of five classes: `cpu`, `thread`, `mutex`, `cv`, and `semaphore`, which are declared in `cpu.h`, `thread.h`, `mutex.h`, `cv.h`, and `semaphore.h` (do not modify these files).

To use the thread library, `#include` the needed header files and link with `libthread.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. This initial thread calls the function pointed to by `func`, passing the single argument `arg`. A user program should call `cpu::boot` exactly once, before calling any other thread functions. `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; this can be helpful when debugging.

Note that `cpu::boot` is a `static` member function; it is not called on an instance of the `cpu` class).

2.2. `thread` class

The `thread` class is declared in `thread.h`.

The constructor creates a new thread. The thread library causes this new thread to begin execution by calling the function pointed to by `func`, passing it the single argument `arg`.

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

The `join` method causes the calling thread to block until the thread denoted by the method's object has exited. If the specified thread has already exited, `join` returns immediately.

```
void join();
```

2.3. `mutex` class

The `mutex` class is declared in `mutex.h`.

The constructor is used to create a new mutex. Recall that mutexes are initialized to be free.

```
mutex();
```

`lock` atomically waits for the mutex to be free and acquires it for the current thread. If multiple threads are contending for the mutex, the thread library ensures that only one will acquire it at a time. You may not assume anything about the order in which contending threads acquire the mutex.

```
void lock();
```

`unlock` releases the mutex. Throws `std::runtime_error` exception if the calling thread does not hold the mutex.

```
void unlock();
```

2.4. `cv` class

The `cv` class is declared in `cv.h`.

The constructor is used to create a new condition variable.

```
cv();
```

`wait` atomically releases mutex `m` and waits on the condition variable's wait queue. When the thread is awakened, `wait` will attempt to re-acquire the mutex as it would by calling `lock`; after it acquires the mutex, `wait` returns to the calling thread. You may not assume anything about how long a thread takes between awakening and attempting to re-acquire the mutex. Note that, as in most languages, threads may awaken **spuriously**, without any thread calling `signal` or `broadcast`. `wait` throws a `std::runtime_error` exception if the calling thread does not hold the mutex.

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

`signal` awakens one of the threads on the condition queue. If multiple threads are waiting, you may not assume anything about which thread is woken.

```
void signal();
```

`broadcast` awakens all threads on the condition queue.

```
void broadcast();
```

2.5. `semaphore` class

This class is provided so you can try programming with semaphores in lab or on your own. Use mutexes and condition variables (not semaphores) for Project 1.
The `semaphore` class is declared in `semaphore.h`.

The constructor is used to create a new semaphore and initialize it to the specified value.

```
semaphore(unsigned int initial_value);
```

`down` atomically waits for the semaphore to be positive and decrements it.

```
void down();
```

`up` atomically increments the semaphore.

```
void up();
```

2.6. Scheduling and termination

Threads may run at arbitrary speeds; you may not assume anything about their relative speeds.

You may not assume anything about which thread acquires a mutex after that mutex is released. Any of the threads currently waiting for the mutex, or even a thread that calls `lock` later, may be the next to acquire the mutex.

When more than one thread is waiting on a condition variable, you may not assume anything about which will be woken in response to a `signal`, nor may you assume anything about the speed with which the woken thread attempts to re-acquire the mutex.

When more than one thread is waiting in a call to `down`, you may not assume anything about which thread will be woken in response to an `up`.

When no threads can run, the thread library exits and your program terminates.

2.7. Example program

Here is a short program that uses threads.

```
#include <iostream>
#include "cpu.h"
#include "thread.h"
#include "mutex.h"
#include "cv.h"

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

mutex mutex1;
cv cv1;

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

```
void child(void *a)
{
    // with a C-style cast, this would be written as:
    // auto message = (char *) a;
    auto message = static_cast<char *>(a);

    mutex1.lock();
    cout << "child called with message " << message << ", setting child_done = 1" <<
    child_done = 1;
    cv1.signal();
    mutex1.unlock();
}

void parent(void *a)
{
    // with a C-style cast, this would be written as:
    // auto arg = (intptr_t) a;
    auto arg = reinterpret_cast<intptr_t>(a);

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

    // with a C-style cast, this would be written as:
    // thread t1 (child, (void *) "test message");
    thread t1 (child, static_cast<void *>(const_cast<char *>("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()
{
    // with a C-style cast, this would be written as:
    // cpu::boot(parent, (void *) 100, 0);
    cpu::boot(parent, reinterpret_cast<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 called with message test message, setting child_done = 1
parent finishing
No runnable threads.  Exiting.
```

```
parent called with arg 100
child called with message test message, setting child_done = 1
parent finishing
No runnable threads.  Exiting.
```

3. Disk scheduler

Your task for this project is to write a concurrent program that issues and services disk requests. Use mutexes and condition variables (not semaphores) for synchronization.

The disk scheduler in an operating system receives and schedules requests for disk I/Os. Threads issue disk requests by enqueueing them at the disk scheduler and waiting for them to be serviced. The disk scheduler queue can contain at most a specified number of requests (`max_disk_queue`); this limit is passed as a command-line argument to your program. Threads may only make a request if the queue has room to hold the request; otherwise they must wait for the queue to have room.

Your program will create a set of threads: one **servicer** thread and a number of **requester** threads. The number of requester threads will be inferred from the command-line arguments. Each requester thread should issue a series of requests for disk tracks, which are specified in that requester's input file in the order they should be issued. The names of these input files are passed as command-line arguments to your program. Each request is **synchronous**; a requester thread must wait until the servicing thread finishes handling that requester's current request before that requester issues its next request.

Requests in the disk queue are **not** necessarily serviced in first-come, first-served order. Instead, the service thread handles disk requests in **shortest seek time first** (SSTF) order. That is, the next request it services is the request that is closest to its current track. If two requests are equidistant to the current track, then the service thread can pick any of them to service next. 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 requester threads are still active. When at least `max_disk_queue` requester threads are active, the largest possible number of requests in the queue is equal to `max_disk_queue`. When fewer than `max_disk_queue` requester threads are active, the largest number of requests in the queue is equal to the number of living requester threads. A requester is considered to be active from the time the program starts until all its requests have been serviced.

Your program should not force a particular order on which requester should get to issue a request when there is space in the disk queue; this should depend solely on the speed of the threads and the scheduling of mutexes and condition variables.

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 can be inferred 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 the arguments to the program are correct and that the input files are formatted correctly.

3.2. Output

In the critical section where a requester thread issues a request for a particular `track`, that requester thread should call `print_request(requester, track)`. A request is available to be serviced after `print_request` returns.

In the critical section where the service thread services a request for a given `track` by a given requester, the service thread should call `print_service(requester, track)`. A request is considered to have been removed from the queue after `print_service` finishes.

`print_request` and `print_service` are declared in disk.h (do not modify this file) and included in `libthread.o`.

Your submitted program should not generate any other output. You may find it helpful to add debugging output while testing your program, but these should be disabled or removed before submission.

3.3. Example input/output

Here is an example set of input files (disk.in0 - disk.in4). We recommend you download these files, rather than copy-pasting the contents into an editor, since some editors create malformed files when you copy-paste (e.g., missing newline character for the last line).

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):

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

```
requester 0 track 785
requester 1 track 350
requester 2 track 827
service requester 1 track 350
requester 3 track 302
service requester 3 track 302
requester 4 track 631
service requester 4 track 631
requester 1 track 914
service requester 0 track 785
requester 3 track 230
service requester 2 track 827
requester 0 track 53
service requester 1 track 914
requester 4 track 11
service requester 3 track 230
requester 2 track 567
service requester 0 track 53
service requester 4 track 11
service requester 2 track 567
No runnable threads.  Exiting.
```

3.4 Tips

Remember some general tips for concurrent programming.

- Identify shared state, and assign a mutex to (each group of) this state. It is easier to write correct concurrent programs with fewer mutexes (though possibly with slower performance).
- Identify critical sections by noting regions within which invariants are broken or over which state dependencies exist. Acquire the appropriate lock before a critical section starts, release it after it ends. It is easier to write correct concurrent programs with fewer, larger critical sections (though possibly with slower performance).
- Identify ordering constraints: situations in which one thread cannot safely continue. Encode that constraint as a boolean expression over shared state, and assign a condition variable to each condition, associated with the appropriate lock. When a thread cannot safely continue based on the current state, it should call `wait`. When a thread changes state that might allow another thread to continue, it should call `signal` or `broadcast`.
- `wait` should always be called within a loop that checks the condition being waited for.
- The only statement inside the waiting loop (other than debugging print statements) should be `wait`. No state should be changed inside a waiting loop, and thus no thread should call `signal` or `broadcast` within a waiting loop.

4. Project logistics

Write your disk scheduler in C++17 or C++20 on Linux. Use `g++ 12.2.1` (with -std=c++17 or -std=c++20) to compile your programs. To use g++ 12.2.1 on CAEN computers, put the following command in your startup file (e.g., `~/.bash_profile`) and re-login:

```
module load gcc/12.2.1
```

You may use any part of the standard C++ library, except the C++ thread facilities. You should not use any libraries other than the standard C++ library. Your disk scheduler code may be in multiple files. Each file name must end with `.cc`, `.cpp`, `.h`, or `.hpp` and must not start with `test` (filenames that start with `test` will denote test cases in later projects).

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

Here's a simple Makefile that shows how to compile a disk scheduler (adjust the file names in the Makefile to match your own program). If you are using a development environment that uses CMake, the following CMakeLists.txt file may help (on CAEN, you may need to run `cmake -DCMAKE_CXX_COMPILER=g++`).

You are required to document your development process by having your Makefile run `autotag.sh` each time it compiles your disk scheduler (see sample Makefile). `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 disk scheduler.

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

```
handle SIGUSR1 nostop noprint
```

We have created a private github repository for each student (`eecs482/uniqname.1`) Initialize your local repository by cloning the (empty) repository from github, e.g.,

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

5. Grading, autograding, 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 autograder 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 autograder is to help you track progress toward completion. 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, and all submissions will be graded and cataloged. We will use your highest-scoring submission, with ties broken in favor of the later submission. You must recompile and `git push` at least once between submissions.

The autograder 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 a concurrent program, the autograder may return non-deterministic results.

Because your programs will be autograded, 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 files for your disk scheduler:

- C++ files for your disk scheduler. File names should end in `.cc`, `.cpp`, `.h`, or `.hpp` 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. 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 (zip file)

- `cpu.h`
- `cv.h`
- `mutex.h`
- `semaphore.h`
- `thread.h`
- `libthread.o`
- `disk.h`
- `disk.in0`
- `disk.in1`
- `disk.in2`
- `disk.in3`
- `disk.in4`
- `autotag.sh`
- `Makefile`
- `CMakeLists.txt`

8. Experimental platforms

The files provided in this handout were compiled on RHEL 8. They should work on most other Linux distributions (e.g., Ubuntu 22.04) and on Windows Subsystem for Linux (WSL 2, e.g., running

Ubuntu 22.04). We also provide a version of the infrastructure for students who want to develop on MacOS 13. These systems are not officially supported but have been used successfully by prior students.

If you are developing on MacOS:

- Use `libthread_macos.o` instead of `libthread.o`.
- Add `-D_XOPEN_SOURCE` to the compilation flags.
- If you run the project in a debugger, ignore SIGUSR1 signals by issuing the following debugger command:

```
process handle SIGUSR1 -n false -p true -s false
```