Project 1 -- multi-threaded programming

Worth: 5 points
Assigned: January 15, 2025
Due: January 29, 2025

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 manages a pizza delivery system. 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, uintptr_t 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` (`uintptr_t` is an integer type that is large enough to hold a pointer). 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` (`uintptr_t` is an integer type that is large enough to hold a pointer).

```
thread(thread_startfunc_t func, uintptr_t 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. Do not use 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"

mutex mutex1;
cv cv1;

bool child_done = false;        // global variable; shared between the two threads

void child(uintptr_t arg)
{
    auto message = reinterpret_cast<char*>(arg);

    mutex1.lock();
    std::cout << "child called with message " << message << std::endl;
    std::cout << "setting child_done" << std::endl;
    child_done = true;
    cv1.signal();
    mutex1.unlock();
}

void parent(uintptr_t arg)
{
    mutex1.lock();
    std::cout << "parent called with arg " << arg << std::endl;
    mutex1.unlock();

    thread t1 (child, reinterpret_cast<uintptr_t>("test message"));

    mutex1.lock();
    while (!child_done) {
        std::cout << "parent waiting for child to run" << std::endl;
        cv1.wait(mutex1);
    }
    std::cout << "parent finishing" << std::endl;
    mutex1.unlock();
}
```

```
int main()
{
    cpu::boot(parent, 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
parent finishing
No runnable threads.  Exiting.
```

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

3. Pizza delivery system

Your task for this project is to write a concurrent program that manages a pizza delivery system. Use mutexes and condition variables for synchronization.

The pizza delivery system manages D drivers (numbered [0,D)) and handles requests from C customers (numbered [0,C)). Each driver and customer is represented by its own thread. Each customer makes a series of requests, with each request specifying where to bring the pizza.

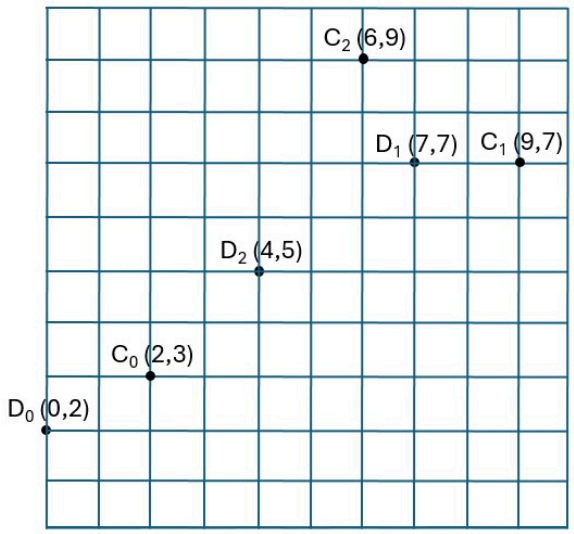
A location in this system is represented as (x,y) coordinates. Distance between locations is defined according to rectilinear distance, i.e., the difference in x coordinates plus the difference in y coordinates.

```
struct location_t {
    unsigned int x;
    unsigned int y;
};
```

A pizza request involves the following sequence of actions, each of which is carried out by the specified library function.

1. Drivers announce that they are available to deliver pizza (`driver_ready`), and customers announce that they are requesting pizza (`customer_ready`). A customer may have at most one pizza request pending at a time.
2. Matches are made (`match`) between a ready customer and a ready driver. A match between a customer and driver excludes that customer and driver from other matches until that customer has paid that driver at that customer's location. A match between a customer C and a driver D is legal if and only if
   - C is one of the closest ready customers to D, i.e., (C ∈ the set of ready customers) ∧ (∀ C_i in the set of ready customers: distance(C,D) ≤ distance(C_i,D))
     and
   - D is one of the closest ready drivers to C, i.e., (D ∈ the set of ready drivers) ∧ (∀ D_i in the set of ready drivers: distance(C,D) ≤ distance(C,D_i)).

   For example, if all customers and drivers in the map below are ready, the only legal matches are C_0-D_0 and C_1-D_1.



3. After a match is made, the matched driver drives to that customer's location (`drive`).
4. After the driver arrives at the customer's location, the customer pays for the pizza (`pay`).
5. After the customer pays, the request is completed, and the driver and customer may handle other requests.

3.1 Library functions

Drivers and customers act by calling the following library functions, which are declared in `pizza.h` and defined in `libthread.o` . These functions are thread safe.

Driver

A driver acts by calling the following library functions. These functions must be called by the specified driver's thread.
A driver announces they are available to deliver pizza by calling `driver_ready` with their driver number and current location. `driver_ready` must be called before a driver can be matched with a customer.

```
void driver_ready(unsigned int driver, location_t location);
```

A driver drives from one location to another by calling `drive` with their driver number, starting location, and ending location. **Driving is a slow operation, so any number of drivers must be able to execute `drive` at the same time.**

```
void drive(unsigned int driver, location_t start, location_t end);
```

Customer

A customer acts by calling the following library functions. These functions must be called by the specified customer's thread.
A customer announces their pizza request by calling `customer_ready` with their customer number and location. `customer_ready` must be called before a driver can be matched to this request.

```
void customer_ready(unsigned int customer, location_t location);
```

After a driver arrives at a customer's location, the customer pays the driver by calling `pay` with their customer number and the driver number they are paying. `pay` must be called before that driver can declare they are ready for another request.

```
void pay(unsigned int customer, unsigned int driver);
```

Any thread

Any thread may announce that a customer and driver have been matched by calling `match` with the matched customer and driver numbers. `match` must be called before a driver drives to the customer.

```
void match(unsigned int customer, unsigned int driver);
```

3.2. Input

Your program will be called with several command-line arguments. The first argument specifies the number of drivers that are delivering pizza (you may assume there is at least one driver). The initial location for each driver is (0,0). The rest of the arguments specify a list of input files (one input file per customer). I.e., the input file for customer c is argv[c+2], where 0 <= c < (number of customers). The number of customers is equal to the number of input files specified. You may assume there is at least one customer.

The input file for each customer contains that customer's series of requests. Each line of the input file specifies the location for this request.

```
location_x location_y
```

You may assume that input files are formatted correctly.

A customer thread ends when all its requests have been completed. A driver thread never ends -- a driver's job is never done, and drivers never run out of pizza!

No special handling should be done for unusual requests (e.g., a request that has the same location as the last location). For each request, a driver should drive from their current location to the requested location, even if these locations happen to be the same. Multiple drivers and customers may occupy the same location.

3.3. Output

Your submitted program should produce output only by calling the provided library functions. These functions ensure mutual exclusion on the output by locking/unlocking `cout_mutex` (declared in `pizza.h`); do not hold this mutex when calling one of the provided library functions.

If you wish to print other output while debugging, you must ensure mutual exclusion with the `cout` statements issued by the library functions by locking/unlocking `cout_mutex`.

3.4. Example input/output

Here is an example set of input files (customer.in0 and customer.in1). 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).

customer.in0	customer.in1
0 4 5 0	7 3 1 0

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

```
pizza 2 customer.in0 customer.in1
```

```
driver 0 ready at (0,0)
driver 1 ready at (0,0)
customer 0 requests pizza at (0,4)
customer 1 requests pizza at (7,3)
customer 0 matched with driver 0
driver 0 driving from (0,0) to (0,4)
customer 1 matched with driver 1
driver 1 driving from (0,0) to (7,3)
customer 1 pays driver 1
customer 0 pays driver 0
customer 1 requests pizza at (1,0)
driver 1 ready at (7,3)
customer 1 matched with driver 1
driver 1 driving from (7,3) to (1,0)
driver 0 ready at (0,4)
customer 0 requests pizza at (5,0)
customer 0 matched with driver 0
```

```
driver 0 driving from (0,4) to (5,0)
customer 0 pays driver 0
driver 0 ready at (5,0)
customer 1 pays driver 1
driver 1 ready at (1,0)
No runnable threads.  Exiting.
```

3.5 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 mutex. 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 pizza delivery program 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 program 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 pizza delivery program (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 pizza delivery program (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 pizza delivery program.

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
```

After you register your github username, and accept the invitation to the EECS 482 github organization, you may use the private github repository we created for you to use for this project. Initialize your local repository by cloning the (empty) repository from github, e.g.,

```
git clone git@github.com:eecs482/anikangi.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 feedbacks over the duration of this project. Bonus feedbacks are used when you view more than one submission from a particular day. After your 3 bonus feedbacks are used up, the system will continue to provide feedback for one submission per 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 generate output only via the library functions specified in Section 3.1.
- Your program should expect several command-line arguments, with the first being being the number of drivers and the others specifying the list of input files for the customers.
- Do not modify or rename the header files provided in this handout.

In addition to the autograder's evaluation of your program's correctness, a human grader will evaluate your program on issues such as compiler warnings, documentation, hard-coded constants, time and space efficiency, code duplication, and understandability. For documentation, you should at least write a block comment at the top of each function that explains the purpose and interface of that function. Your initial project score will be the highest autograder score [0,80] you have achieved by the due date, multiplied by a hand-grading score [1,1.25].

After the due date, you may continue to submit to the autograder and earn more autograder points, but these extra points will not be multiplied by the hand-grading score. For example, if your autograder score at the project due date is 60, your hand-grading score is 1.20, and your autograder score at the end of the semester is 80, then your final project score will be 60*1.20 + (80-60) = 92. The last day to submit to the autograder is April 26, 2025.

6. Turning in the project

Submit the following files:

- C++ files for your pizza delivery program. 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.

7. Files included in this handout (zip file)

- `cpu.h`
- `cv.h`
- `mutex.h`
- `semaphore.h`
- `thread.h`
- `libthread.o`
- `libthread_macos.o`
- `pizza.h`
- `customer.in0`
- `customer.in1`
- `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 clang++/lldb instead of g++/gdb.
- 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
```