Assignment #2.2

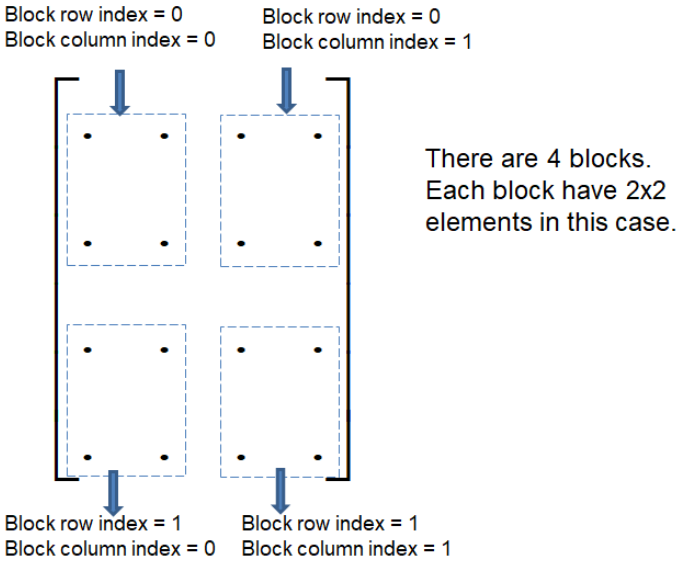
CSD 2180/2182 Fall 2023

Due Date:	As specified on the moodle
Topics covered:	Multi-threading, Synchronization, Semaphore
Deliverables:	To submit all relevant files that will implement the multi-threading matrix multiplication.
Objectives:	To learn and understand multi-threading and synchronization using `std::thread` and `std::mutex`. The student who successfully completes this assignment will understand multi-threading programming.

Programming Statement: Multi-threading Matrix Multiplication

In this assignment, you will implement a multi-threading version of matrix multiplication under Linux. It consists of a set of functions that allow management of worker threads. The main thread starts the initialization and performs the dispatching work by constructing a dispatcher.

We create and split the matrix into blocks to do block matrix multiplication. Each block matrix multiplication is considered as a job request that will be dispatched by the dispatcher. Each worker thread each time completes one block matrix multiplication. Each block in the matrix is assigned with a block row index and column index as illustrated in the following example.



If we have a number of instances for resource, it is better to use Binary semaphore. Otherwise it is better to use mutex when there is a single instance for resource. There is ownership associated with mutex because only owner can release the lock. Semaphore have no ownership.

By way of introduction, we shall describe some of the key concepts here:

- Management of worker threads – this means that we create and manage a pool of threads.
- Queue of job requests - this means that we create a pool of requests and add to the queue.
- Queue of workers - this means that we create a pool of workers and add to the queue. Each worker is associated with a thread. The Worker class defines a thread function.

1 Dispatcher

Dispatcher is specified as follows:

```
#pragma once
#ifndef DISPATCHER_H
#define DISPATCHER_H

#include "worker.h"

#include <queue>
#include <mutex>
#include <thread>
#include <vector>

using namespace std;

class Dispatcher
{
        static queue<Request*> reqs;//job requests
        static queue<Worker*> workers;
        static mutex requestsMutex;
        static mutex workersMutex;
        static vector<Worker*> allWorkers;//workers
        static vector<thread*> threads;//threads
        static mutex jobCountMutex; //mutex for job counter
        static mutex outputMutex; //mutex for final results
        static int nJobs; //counter for total job requests

public:
        static bool init(int workers, int jobs);
        static bool stop();
        static void addRequest(Request* req);
        static bool addWorker(Worker* worker);

        static void lockOutput();
        static void unlockOutput();
        static void decreaseJobs();
```

```
};

#endif
```

1.1 bool init(int workers, int jobs)

This function adds the workers into `allWorkers`. It creates the threads based on `std::thread` (with total number of threads *workers*), each associated with a worker (i.e., running Worker::run() on each object Worker), and adds into `threads`. It also sets the expected number of jobs to `nJobs`.

1.2 bool stop()

This function terminates the worker threads calling `stop()` for the workers and joins all the threads if the job requests are all serviced by checking `nJobs`.

1.3 void addRequest(Request* request)

This function checks whether there is a worker available in the workers queue. If so, it uses `workersMutex` to synchronize the access to workers queue, gets one worker from the queue, assigns the job request and notifies the worker using conditional variable `cv`. Otherwise it adds the request to the requests queue. It uses `requestsMutex` to synchronize the access to requests queue.

1.4 bool addWorker(Worker* worker)

If a request is waiting in the requests queue, assign it to the worker. It uses `requestsMutex` to synchronize the access to requests queue and returns false. Otherwise the function adds the worker to the workers queue. It returns true if the worker was added to the queue and has to wait for its condition variable. Use `workersMutex` to synchronize the access to workers queue.

1.5 void decreaseJobs()

This function decreases the job counter `nJobs` using `jobCountMutex` to synchronize the access to `nJobs`.

1.6 void lockOutput()

This function locks the mutex of `outputMutex`, which is used for the access to the final results of matrix multiplication.

1.7 void unlockOutput()

This function unlocks the mutex of `outputMutex`, which is used for the access to the final results of matrix multiplication..

2 Request

Request is specified as follows:

```
#pragma once
#ifndef REQUEST_H
#define REQUEST_H

#include <string>
#include <mutex>
using namespace std;

typedef void (*logFptr)(string text);

class Request
{
        int numARows; //# of rows for 1st input matrix
        int numAColumns;//# of columns for 1st input matrix
        int numBColumns;//# of rows for 2nd input matrix
        int rowABlkIdx;//Block row index for the block from 1st input matrix
        int colABlkIdx;//Block column index  for the block from 1st input matrix
        int colBBlkIdx;//Block row index for the block from 2nd input matrix
        int blockSz;//block size for block matrix

        float *a; // 1st input for block matrix mulitiplication
        float *b; // 2nd input for block matrix mulitiplication
        float *c; // partial result for block matrix mulitiplication
        float *out; //final result for block matrix mulitiplication

        logFptr outFnc; //print function ptr.
        bool initialized;//whether the request has been initialized

        public:
        Request(int m, int n, int l,
                        float *C, int blkSz,
                        int rowABlockIdx, int colABlockIdx,
                        int colBBlockIdx); //constructor
        void init(float *A, float *B); //allocate memory/copy data
        void setOutput(logFptr fnc) { outFnc = fnc; }
        void process();
        void finish();
};

#endif
```

Assume the final results are stored at `out` and the partial results at `c`. The first input matrix of the block matrix multiplication is stored at `a` and the send input matrix at `b`. Each job request consists of one block matrix multiplication with the block size `blockSz` × `blockSz`. The input matrices for matrix multiplication are split into multiple block

matrices, where `rowABlockIdx` and `colABlockIdx` specify the row and column index for the first input block matrix respectively, and `colABlockIdx` and `colBBlockIdx` specify the row and column index for the second input block matrix respectively.

2.1 Requester()

The constructor `Request(int m, int n, int l, float *C, int rowABlockIdx, int colABlockIdx, int colBBlockIdx, int blkSz)` sets the matrix dimension for the input matrices to compute the matrix multiplication. The first matrix size is $m \times n$ and the second $n \times l$. `C` is the pointer to the final results. `rowABlockIdx` and `colABlockIdx` are respectively the row and column index for the block from the first input matrix. `colBBlockIdx` is the column index for the block from the second input matrix. Please note that `colABlockIdx` is the row index for the block from the second input matrix. `blkSz` is the number of the rows(columns) of the block matrix.

2.2 void init(float *A, float *B)

If it is not initialized, allocate memory for the block matrices `a`, `b` and `c`. It copies the data from the input matrices `A` and `B` to the allocated memory of block matrices.

2.3 void process()

The function computes the block matrix multiplication using the provided `compute()` from mm.h.

2.4 void finish()

The function merges the partial results at `c` from block matrix multiplication into the final result by using `outputMutex` to synchronize the access to `out`. It decrements the job counter `nJobs` calling `Dispatcher::decreaseJobs()`.

3 Worker

Worker is specified as follows:

```
#pragma once
#ifndef WORKER_H
#define WORKER_H

#include "request.h"

#include <condition_variable>
#include <mutex>

using namespace std;

class Worker
{
        condition_variable cv;
```

```
        mutex mtx;
        unique_lock<mutex> ulock;
        Request* req;
        bool running;
        bool ready;

public:
        Worker();
        void run();
        void stop();
        void setRequest(Request* req);
        void getCondition(condition_variable* &cv);
};

#endif
```

3.1 Worker()

It sets its running state to true and ready state to false for this worker. It also prepares a unique_lock `ulock` that will be used in the function of `wait_for` of condition variable `cv`.

3.2 void setRequest(Request* req)

The function sets the associated request to `req` and turns the ready state to true (so that it allows the thread function `run()` to be waken up to serve the request).

3.3 void stop()

It sets its running state to false (e.g. when the main thread terminates the worker thread).

3.4 void getCondition(condition_variable* &cv)

This function returns the condition variable `cv`.

3.5 void run()

This function checks whether the worker is allowed to run. If so, it will repeat the following. It checks whether it is ready to serve the request. If so, it turns its ready state to false, calls `process()` to complete the computation for the request and then `finish()` to merge the partial result into the final results. Otherwise, it should add itself to the queue with `addWorker()` and execute next request or wait. If it has to wait, it uses a loop to deal with spurious wake-ups. It checks whether its ready state is false (not serving any request now) and the thread is allowed to run. If so, it uses condition variable `cv` with `ulock` to wait for timeout (e.g. 1 sec). If it is timed out, the function should keep waiting unless the worker is stopped by the dispatcher.

Rubrics

Your job is to try to implement the multi-threading matrix multiplication described above with all the information given to assist you in this task. All coding should be done in *C++*. If you understood the material and implemented your code carefully, you should not be required to write more than 250 lines of code (not including comments).

The code that you write should use the provided header in `dispatcher.h`, `worker.h` and `request.h` that is listed in the previous sections.

What to submit:

- A file `dispatcher.cpp` that contains the implementation of the dispatcher.
- A file `worker.cpp` that contains the implementation of the worker.
- A file `request.cpp` that contains the implementation of the job request for block matrix multiplication.
- You **should not** submit `dispatcher.h`, `worker.h` and `request.h` i.e., your code should work with the provided headers.

Submission of these files should be according to the stipulations already stated in the syllabus.

In general, the rubrics for this assignment are the following:

- Passing test cases.
- Comments or documentation.
- Coding style and compilation without warnings.

Students are required to provide succinct comments to the given code.

The list above is non-exhaustive. The lecturer reserves the right to impose reasonable penalties for code that violates general practices or does not match the specification in an obvious way that has not been mentioned above. In exceptional cases, the lecturer reserves a discretionary right to allow resubmission or submission after the deadline.

Appendices

A mm.h

Listing 1: mm.h

```
#pragma once
#ifndef MMH
#define MMH
//create input and output matrices
//write into the input and output files
//the size of 1st matrix for multiplication is
//numARows * numACols;
//the size of 2nd matrix for multiplication is
//numACols * numBCols;
//compute the results and store at output fil
//the size of resulting matrix is numARows*numBCols
void createDataset(
        const char *input0_file_name,
        const char *input1_file_name,
        const char *output_file_name,
        int numARows,
        int numACols,
        int numBCols);

//read the input data from the file
//the size of the matrix is height*width
//return the pointer to the input data
float *readData(const char *file_name,
        int *height,
        int *width);

//write the output data to the file
//the size of the matrix is height*width
void writeData(const char *file_name,
        float *data,
        int height,
        int width);

//compute the matrix multiplication using
//the given input matrices from the files
//input0 and input1. the result is saved
//into the file output.
//the size of 1st matrix for multiplication is
//numARows * numACols;
//the size of 2nd matrix for multiplication is
//numACols * numBCols;
//the size of resulting matrix is numARows*numBCols
```

```
void compute(float *output, float *input0, float *input1,
        int numARows, int numAColumns, int numBColumns);

//create the random data for the matrix
//return the pointer to the data of the matrix
float *createData(int height, int width);
#endif
```

B main.cpp

Listing 2: main.cpp

```
#include "dispatcher.h"
#include "request.h"

#include <iostream>
#include <string>
#include <csignal>
#include <thread>
#include <chrono>
#include <cstring>

#include "mm.h"
constexpr float epsilon = 0.0001;
using namespace std;

// Globals
mutex logMutex;

void logFnc(string text)
{
        logMutex.lock();
        cout << text << "\n";
        logMutex.unlock();
}

int main()
{
        int numThreads;
        int numARows;
        int numACols;
        int numBCols;
        int blockSize;

        std::cin >> numThreads;
        std::cin >> numARows;
        std::cin >> numACols;
        std::cin >> numBCols;
        std::cin >> blockSize;
```

```
// Generate job requests.
int nARowBlocks = (numARows - 1) / blockSize + 1;
int nAColBlocks = (numACols - 1) / blockSize + 1;
int nBColBlocks = (numBCols - 1) / blockSize + 1;

// Initialise the dispatcher with some worker threads.
Dispatcher::init(numThreads, nARowBlocks*nAColBlocks*nBColBlocks);
cout << "Initialised.\n";

const char *in_file0 = "input0.raw";
const char *in_file1 = "input1.raw";
const char *out_file = "output.raw";

createDataset(in_file0, in_file1, out_file, numARows, numACols, numBCols);

float *input0_data = readData(in_file0, &numARows,  &numACols);
float *input1_data = readData(in_file1, &numACols, &numBCols);
float *result = new float[numARows * numBCols];
std::memset(result, 0, numARows * numBCols * sizeof(float));

Request* rq = 0;
for (int i = 0; i < nARowBlocks; i++)
{
        for (int j = 0; j < nAColBlocks; j++)
        {
                for (int k = 0; k < nBColBlocks; k++)
                {
                        rq = new Request(numARows, numACols, numBCols,
                                        result, blockSize, i, j, k);

                        //allocate memory and copy from input
                        rq->init(input0_data, input1_data);

                        rq->setOutput(&logFnc);
                        Dispatcher::addRequest(rq);
                }
        }
}

// Cleanup.
Dispatcher::stop();
cout << "Clean-up done.\n";

cout << "Checking...\n";

float *ref = readData(out_file, & numARows, & numBCols);
for (int i = 0; i < numARows; i++)
```

```
        {
                for (int j = 0; j < numBCols; j++)
                {
                        if (abs(result[i*numBCols+j] - ref[i*numBCols+j])
                                > epsilon)
                        {
                                i = numARows;
                                cout << "Error in multi-threading MM...\n";
                                break;
                        }
                }
        }
        return 0;
}
```