CAPS Exam 2022-23 (First Sit)

# Question One

*Describe the following concepts. Use supporting code fragments in the style of C++ as appropriate.*

## 1a. Concurrency

In Computer Science, concurrency is the idea that code can be written in such a way that different parts of the program or its algorithm(s) can be executed out of order, and typically simultaneously, without changing the final outcome versus executing them in order.

A simple but realistic example of applying concurrency in computing is matrix multiplication – something used for the mathematical optimisation of many algorithms:

Assuming 2-dimensional matrices, in non-concurrent code, two nested for-loops can be used to multiply:

for (i = 0; i < row1; i++)

    for (j = 0; j < col2; j++)

        for (k = 0; k < col1; k++)

            result[i][j] += a[i][k] \* b[k][j];

This simple algorithm works but for large matrices, especially when performing many thousands or millions of calculations per second, it will struggle to keep up. The code can be parallelised to work concurrently using threads:

for (int c = 0; c < col1; c++)

    threadHandles.emplace\_back(multiply, a, b, result, c, col1, row1);

In this code, each column of the first matrix is multiplied by each row of the second matrix in separate threads. By dividing the calculation into parts and computing these parts separately, we can execute it in its entirety in much less overall time.

## 1b. Threads

While there are multiple definitions of this term in the wider field of Computer Science, in the context of concurrency and writing multithreaded code, a thread is a logical subdivision of a process which shares its resources with the process’ other threads.

In its most simple form, a thread can be instantiated and passed a function object and arguments to pass to the function; it will then execute this function on a separate compute thread within the same process and return when complete. Continuing the thread example from 1a;

for (int i = 0; i < col1; i++)

    threadHandles[i].join();

The ‘join’ function waits for a thread to finish executing before it allows the thread which called it to continue. This is the most simple way to ensure that a thread has done everything it needs to – an important thing to keep track of as often later parts of the program are dependent on values calculated asynchronously.

In the ‘multiply’ function called in these threads, shared memory is accessed in the form of pointers:

void multiply(int \*\*a, int \*\*b, int \*\*result, int c, int col1, int row1)

{

    unsigned int i, j;

    for (i = 0; i < col1; i++)

        for (j = 0; j < row1; j++)

            result[i][c] += a[i][c] \* b[j][i];

}

The pointers passed into the function as arguments allow each thread to contribute its results to a variable declared in the caller thread.

## 1c. Mutex variable

A mutex variable, which stands for mutual exclusion (object), can be used to forcefully control the order of execution in multithreaded code, by preventing other threads from ‘acquiring’ it when another thread has already done so. This is used to allow threads to complete critical sections of code such as writing to memory or a file, where race conditions would otherwise be in play.

Once again continuing with the matrix multiplication example used above; there is an issue with the ‘multiply’ function passed into each thread: there is no code controlling access of the shared memory, and this is very likely to cause corruption of the data – when two or more threads try to write a value to memory at around the same time, they can end up overriding each other’s operations in strange ways and the value can become erroneous or unreadable entirely. To prevent this, we can use a mutex:

void multiply(int \*\*a, int \*\*b, int \*\*result, int c, int col1, int row1)

{

    unsigned int i, j;

    for (i = 0; i < col1; i++)

        for (j = 0; j < row1; j++)

        {

            mtx.lock();

            result[i][c] += a[i][c] \* b[j][i];

            mtx.unlock();

        }

}

‘mtx’ is declared in the main thread and the same instance of it is accessed by all instantiated threads. When its ‘lock’ function is called by a thread, if another thread does the same, the mutex will block its execution until the first thread calls ‘unlock’. This assures that only one thread is writing to the memory location at a time and prevents any unpredictable behaviour.

## 1d. A future

A future is an object in a multithreaded program which is used to access the result of an asynchronous operation by providing a known place to store the value when it is ready and functionality to check if it is there or wait for it. Any thread with access to the object can, at any point during execution, check to see if a value has been set, wait for the value to be set, or get the value.

# Question Two

## 2a. Define the term deadlock, giving a supporting example.

Deadlock is a term used to describe the state of a multithreaded application when one or more threads become stuck, indefinitely waiting for another thread(s) to take action.

A classic example of a deadlock situation is a traffic jam at a junction. Let’s say there is a crossroads, where both roads have two directions of flow.

## 2b. Using fragments of code show how deadlock can be avoided in C++?

Deadlock is caused by improper design when using mutexes or other locking/thread blocking mechanisms. It can be prevented by using the correct approaches when developing multithreaded code, but if it is present despite good practices then the code obviously must be altered.

# Question Three

*In your answers to this question, consider a compression application, such as 7-Zip or WinRAR, that creates a single compressed file from one or more other files. The uncompressed files are each compressed independently to make up the final compressed file.*

## 3a. What is meant by thread pooling?

Thread pooling is when, in a multithreaded program, instead of instantiating new threads for each task, a so-called ‘pool’ of *n* persistent threads (threads that do not close after they complete execution of a single function) is instantiated at the start of execution or before an expensive concurrent operation. Tasks are provided one by one to each existing thread, with each thread completing its task then checking a queue for its next one. Some or all threads may enter a sleeping state if there are not enough tasks, but the more threads that are working, the more efficient the thread pool is (sleeping threads require some CPU clock cycles and memory, although very little).

This concept is especially applicable to a compression application, as the number of individual files to be compressed is unknown, so a thread pool ensures that if there are many individual files and thus tasks, the program will not waste execution time instantiating a new thread for each one. It is especially important to keep in mind that it is fairly common for there to be many small files present in a file system, and thread instantiation would constitute a noticeable portion of the execution time for compressing such files, further giving merit to the pooling approach in this context.

## 3b. Discuss the benefits and drawbacks of using thread pooling

As already mentioned, the avoidance of repeated instantiation resulting in less CPU time spent instantiating threads as opposed to actually computing results is the primary benefit of thread pooling.

On a more technical level, thread pooling also reduces memory fragmentation due to all of the threads being created at the same time. This can help the operating system execute code within the threads more efficiently due to the reduced fragmentation, however it is a minor benefit especially in modern computing environments.

As for the drawbacks of thread pooling, these mainly come in the development process as a well-designed thread pool should always be an improvement in the finished product.

One of the issues with using a thread pool when writing a multithreaded application is that it adds some development complexity – rather than just spawning threads one after the other whenever a computation is required, a queue of tasks must be managed, and the threads must be designed in such a way that they can fetch from this queue. Part of this complexity is helped by the fact that there are many free to use, existing implementations of thread pools, which removes a large portion of the programming and testing, but the queue of tasks must still be managed.

The second major issue is from an optimisation perspective – as mentioned earlier, a thread pool is at its most efficient when every thread is in use. This may not always be the case however and depending on the workload, it can be difficult for the developer to predict the optimal number of threads. Each thread pool can be created with a distinct number of threads, but sometimes this number cannot be ascertained before execution, so it is not always possible to guarantee maximum efficiency. For this reason, thread pools are not applicable to *all* multithreaded applications, only ones with very large, particularly repetitive or perpetual, workloads.

## 3c. Show how a thread pool may be implemented in C++.

# Question Four

## 4a. Describe your solution to the coursework problem and discuss the performance improvements you undertook including any consequences of the directions you took

Before I started working on my solution, I knew that the fastest code would use an entirely lock-free data structure, as locks are slow(er) by nature. I had read about lock-free data structures and in theory they are straightforward. However, due to my relative inexperience with writing multithreaded code, I knew it would be a challenge to design my solution using no locks whatsoever. This ended up being the case and in the end my solution used more locking than I’d wanted, but my server still nearly equalled the performance of the reference one.

To store topicIDs, I used a map. The values stored at each key were vectors of strings: the list of posts on the topic. The postID was simply the index of the string (post) in the vector; I felt this was a neat approach and despite increasing the complexity of getting the post ID from O(1) to O(n), it eased development significantly.

I used a secondary map mirroring the keys of the first one to store one mutex object per topic, as after testing I found this was enough to prevent deadlocks. Each action on a topic vector was preceded by a lock or shared lock (making sure to use a shared lock when possible, to improve performance) of the corresponding mutex, using the RAII-style ‘lock\_guard’ object in place of lock calls to decrease the complexity of the code.

## 4b. Discuss the process of testing your coursework submission.

Due to the one-off nature of the required testing, I decided to not design any automated tests although I am aware some other students did.

Running each test case one by one, I inputted the resultant throughput values into a Google sheet and used formulas to calculate most of the columns – of the 12 columns of data, only 5 were manual input, 3 of which were repeating values; the runtime and number of POST and READ threads.

I believe my approach was valid as I completed my testing in good time, with plenty of data gathered and easily presentable.

--- END ---