G52OSC Coursework Part 1

Requirements

This coursework involves adapting (or reproducing or copying) the material which you have been given in lectures/labs/samples, or which you can find in the MSDN online, to answer the specific requirements of the coursework.

Copying code and plagiarism

You may freely copy and adapt any code samples that I have given you in this module, including any lab exercises or lecture samples. You may freely copy code samples from the MSDN, which has many samples explaining how to do specific tasks. This coursework assumes that you will do so and doing so is a part of the coursework, therefore you are not passing someone else's code off as your own, thus doing so does not count as plagiarism.

You must not copy code samples from any other source, including another student on this or any other course, or any third party. If you do so then you are attempting to pass someone else's work off as your own and this is plagiarism. The University takes plagiarism extremely seriously and this can result in getting 0 for the coursework, then entire module, or potentially much worse.

Requirements overview

You will be creating a client process with a number of threads which need to process specified resources. You have the responsibility of ensuring that there is no interference between threads which are processing the same resources.

You will also develop a simple server application which will display the progress of the clients. The clients will inform the server of their progress via Windows messages.

Once you have a working version then you will modify the program which you have made to make it use separate processes rather than threads and to share the counters that you have created between the processes. You will also change the communication method to use sockets rather than Windows messages, and ensure that the server resources are still safe from interference between threads.

When you have finished, you will be able to run both sets of clients, the initial multi-threaded process and the second set of processes concurrently, and the final server will show the results of their progress.

Getting help

You MAY ask Jason or one of the lab helpers for help in understanding *what the requirements mean* (i.e. *what* you need to achieve) if they are not clear. Any necessary clarifications will then be added to the Moodle page so that everyone can see them.

You MAY NOT get help from anybody to actually do the coursework (i.e. *how* to do it), including Jason or the lab helpers. This includes in deciding upon an appropriate protocol for different elements. Jason included the time to do the lab exercises in his time plan for the coursework so you should do these first. The coursework will then be a lot easier for you. You MAY get help on any of the lab exercises, since these are designed to help you to do the coursework without giving you the answers directly.

Part A: The multi-threaded client

You have been provided with a file called `DoNotChangeThese.h`, which contains a set of functions which will simulate the processing of a number of resources. IMPORTANT: You must not put your own code into this file since it will be replaced when we test your program.

You will use five specific functions, called `Procedure1()` to `Procedure5()`, which have been provided for you and you can add whatever code you want to, but you must keep the existing functions which are in the files.

Each of your client threads or processes will call one of these procedures, and the specifiers (i.e. `WINAPI` and return type `DWORD`) have been included to allow these to be used as your thread functions if you wish.

You have also been provided with the code for 15 different test cases. Each of the procedures calls three of these. Test cases 1 to 5 are the easiest to use, cases 6 to 10 are harder and cases 11 to 15 are the hardest. While you are testing you may want to just use cases 1 to 5 initially, commenting out the others, then 1 to 10, then finally 1 to 15 when you have the first ones working.

You may add any code you wish to the test cases (the functions called `TestCase1()` to `TestCase15()`) or the main procedures (the functions `Procedure1()` to `Procedure5()`) but MUST keep the current function calls in the same order. Our tests when we mark your work will verify that these functions are all still called and are still in the original order within the correct functions, so you must not change these.

These functions call a number of functions which you must not implement or change in any way in your submitted coursework. The purpose of each of these is as follows:

`InitialCode(int iDelay)` : This simulates the thread doing some initial work before reaching any critical section.

`RemainderCode(int iDelay)` : This simulates the thread doing some work after it has completed a critical section.

`UseResource(int iResourceID)` : This simulates the process using a resource. You need to count the usage and protect the resources for the duration of the call to `UseResource()` as described below.

`UseTwoResources(int iResourceID1, int iResourceID2)` : This is exactly the same as `UseResource()` but it assumes that both resources will be being used at once. You need to increase the count of how many times each of these resources has been used, and protect them from interference from other threads for the duration of this function call, as described below.

`StartToUseResource(int iResourceID)` : This function simulates the fact that you are using a resource, like `the UseResource()` function, but you need to protect the resource from interference from other threads until the end of the function call `to FinishUsingResource()` for that resource number. i.e. your critical section for that resource starts just BEFORE the call to `StartToUseResource()` and ends just AFTER the call to `FinishUsingResource()`.

`FinishUsingResource(int iResourceID)` : This function matches the calls to `StartUsingResource()` and simulates the idea that the thread is no longer using the indicated resource so you no longer need to protect the resource from interference.

`StartingFunction(int iFunctionID)` : This function will help us test your code when we mark it

`EndingFunction(int iFunctionID)` : This function will help us test your code when we mark it

Requirements for the client

In addition to creating the main code for the client, you have two main tasks which must be completed in relation to the above functions: Protecting the resources and counting the number of times that you use them. These involve adding code to the `Procedure1()` to `Procedure5()` and the `TestCase1()` to `TestCase15()` functions to do this. These and the other requirements for the client are listed below:

A1: Produce a multi-threaded client

The client programs are responsible for doing some processing (they will be faking doing anything important though) and sending a constant set of updates to the server to tell it what is happening.

You need to create a program which will run as a single process. You will create 5 worker threads, which will each run one of the functions called `Procedure1()` to `Procedure5()` as its thread function. These have been written for you but you can add whatever code you need to in order to make them work properly. **You must not change or re-order the function calls in the functions** `Procedure1` **to** `Procedure5`**, or change the number of times that the loop repeats.** i.e. keep the existing code and add to it, do not amend or delete any. Note that our tests will only pass if you keep the content of these functions the same as they are – the correct functions need to be called the correct number of times in the correct order.

Each of these functions calls a number of other functions. You will implement the `SendDataToServer()` function yourself (see below) and will add code to the functions called `TestCase1()` to `TestCase15()` to implement the counting and mutual exclusion which is necessary, as discussed below. Again, you may add whatever code you wish to add to the `TestCase1()` to `TestCase15()` functions but **you must not remove or reorder any existing code in the** `TestCase1()` **to** `TestCase15()` **functions**.

You will need to write a program which will initialise anything you need to initialise, start up the five thread functions and wait for them to finish before it should display the total counts of the number of times that each resource was used in its output window and wait for the user to press `ENTER` before it ends the process.

A2: Protecting the resources

You must identify all of the critical sections which will be necessary to protect the resources which are being used (the usage of which is being simulated using the functions on the previous page). Once you have identified the critical sections, you should apply appropriate entry and exit protocols. These could, for instance, involve spin locks, critical section objects, mutex objects, semaphores, or any other protocol that you wish to use. Your mark for this element will depend upon whether your protocols correctly ensure mutual exclusion as well as how efficient your protocol is. For example being in a critical section for an unnecessarily long time will reduce program efficiency, as discussed in the lectures. In addition, some protocols will be more efficient than others in specific circumstances, again as discussed in the lectures.

i.e. you will add code to ensure that any specific resource is not being used by two threads of the multi-threaded process (or two different processes of the multiple process version) at the same time.

A3: Count the number of times that each resource is used

Just before any call to the functions is made you should increment your count of the number of times that this specific resource was used, since you will send these counts to the server.

There are six resources which you need to protect. You will need six different variables somewhere to count how many times each is used, and will need to protect these variables from interference between threads. Every time there is a call to one of the three functions which means that you are

starting to use a resource (`UseResource()`, `UseTwoResources()`, `StartToUseResource()`) you will need to increment the count of how many times that resource has been used. Note: there are comments labelled TODO in the supplied files.

A4: Communicate the information to the server, via Windows messages

A function called `SendDataToServer()` is called at the end of the loop in each of the `Procedure1()` to `Procedure5()` functions. The parameter is the number of the procedure (i.e. the thread) which called it. You should implement this function to send the following information to the server via Windows message (this will be clearer when you read the section about the server):

1) Send a message to the window which displays the resource usage counts six messages, each of which tells it the current number of times that one of the resources has been used.
2) Send to the window which will display the thread progress a single message specifying the thread number (i.e. the procedure number) which called this function.

You should think about whether you need to use a synchronous or an asynchronous message transmission system (i.e. `SendMessage()` or `PostMessage()`) and will need to justify your answer.

i.e. You will send 7 Windows messages in total: 1 for each resource which will tell the server the value of the resource usage count, and one to say which thread has called the `SendDataToServer()` function.

Notes for client implementation

While you are testing, you may of course freely change any implementations of the functions in `DoNotChangeThese.h` but you should put back the original code in here before you do your final tests, since we will use our own versions of these functions for testing, and may test with multiple different versions to be sure that your code works as expected. We may also change the timings and delays in the `SimulateProcessingTime()` function so you must make sure that your solution implements mutual exclusion correctly rather than relying upon any timing issues.

Your implementation of the `SendDataToServer()` function will depend upon how you implement the server, so consider implementing the message server at the same time as doing this part.

Multiple projects: One or multiple solutions?

You will need three projects by the end of this coursework, one for each program (client from Part A, client from Part C, server from Part B (which was modified in Part C)). You may include these in different Visual Studio solutions or in the same solution, we do not mind. You MUST submit all of the source code and other files which are necessary for us to build and run your programs.

C or C++?

You can write your code within either .c or .cpp files. Personally I would recommend using the .cpp file format because it will be more lenient with you (some things which would be errors in C will be accepted in C++, such as overloaded functions with different parameters). Whichever you choose:

- Your program needs to demonstrate your ability to do concurrent programming using the Windows API functions. The lab exercises do this, so just adapt what you did for those.
- You must use the Windows API functions to achieve these tasks, NOT any third party class or function library.
- You MAY use Visual Studio's wizards to create code for you, but will still need to show your understanding by adapting the code. (Note that it will usually create .cpp rather than .c files.)

Part B: The Windows message server

This is the program which will give you feedback about what is happening as your system runs. All of your clients will connect to this and give it updates about the current situation. Each client may be doing a different thing though and may provide the server with the information in a different manner. Within the server program, you must ensure that shared resources are properly protected and that all drawing is performed in an appropriate way.

Create a new project (in the same or a new solution). Create a program which will start a single process which will do the following:

B1: Create two windows

You should write the code to register two window classes, create two windows and have two different window procedures. Register two window classes (with different names) and create a single window of each class. Give the windows an appropriate size to correctly show their contents (which will be discussed below) and an appropriate name.

The first window will show thread (or process, see Part C) progress and will be called the Thread Progress window (give it this window title). The second will show the number of times that each resource has been accessed and will be called the Resource Access window (give it this title).

Your resource access window will show the information for 12 resources, the six from Part A and another 6 from Part C. Until you do Part C, 6 of these counts should just stay as zero.

Your thread progress window will show the information for 10 threads, the five in the process in Part A and one each for the main threads in the 5 processes in Part C.

You have been given the code for handling the `WM_PAINT` message

You will need to create appropriate variables to store the values for the counts of thread progress and resource usage. You will have ten thread progress variables and twelve resource usage count variables. All variables should initially be zero.

B2: Create your own Windows message handlers

You will need to create message handlers (cases within the switch in the relevant window procedures) for the two (or more, depending upon the implementation that you choose) Windows message types which the client could send to the server.

The Thread Progress message (which is sent to say which thread called the `SendDataToServer()` message) should increment the variable for the thread which sent the message. i.e. if thread 1 sent the message then the variable for the progress of thread 1 should be incremented by `1`.

The Resource Usage message should set the resource usage if the new value is greater than the value which is currently recorded for the usage. In theory (although unlikely in practice), you could receive messages in the wrong order (e.g. thread 1 retrieves the count, builds its message, something increments the count, thread 2 gets the count and builds its message, thread 2's message gets sent then thread 1's message gets sent) so you must first check the current value of the resource count and only set it to be the new value that you received if this value is bigger than the old value.

Think carefully about how many threads could be changing each variable and whether you need to protect any of these variables from interference between threads. Note: think again about this when you implement Part C in case it changes. The purpose of this coursework is to get you to think about this kind of thing when writing concurrent programs, so this is important and is not something that you should ask someone the answer about.

Part C: Change to use multiple processes and sockets

In this part you will take the code that you have for parts B and C and extend it, modifying it to use multiple processes rather than threads on the client side, and to use Windows sockets rather than Window messages for the communication method between the client and server.

Create a new project for the new client program, but you will just be modifying the existing server program so will not need a new project. You can create this new project in a new solution or in the same solution.

C1: Convert to using separate processes

Create a new version of your client program. **Keep the old version as well because you will be submitting that as Part A!** The following requirements apply to your new client. You should not alter your old client at all when you implement the Part C requirements.

Alter your (new) client program to do the following:

Rather than starting five different threads, start five different processes. You will need to tell each process which one it is (the command line is an easy option, see lab 4) because each must run one of the `Procedure1()` to `Procedure5()` functions from Part A.

Assume that you have new resources for this part, and that these resources are NOT shared with the resources from Part A. The six resources in Part A are shared between the threads in Part A. Six new resources in Part C are shared between the processes in Part C but are separate from the six in Part A.

Again, you should keep a count of how many times you change each resource and you need to ensure that your counts of resource usage are shared between all processes. You have seen at least one example of this so look back through the lab notes and lecture slides if you are not sure how to do this.

If you wish, you may consider all resource usages in the function calls to be using resources `7` to `12` rather than `1` to `6` (i.e. add `6` to the parameter values for the function calls to `UseResource()`, `UseTwoResources()`, `StartToUseResource()`, `FinishUsingResource()` to consider which resource number they really affect) if it helps you to understand that there are now 12 resources in total, rather than 6, and that these 6 are different from the 6 in Part A.

We will use a different version of the `DoNotChangeThese.h` file to test this new client, so that the resources really will be different and we can verify that you do not allow interference between processes when using these resources.

You will keep the same version of the server as in Part B, but will extend it to cope with the new clients as well as the old one.

C2: Use Windows sockets

Convert the communication method with the server in the client to use Windows sockets instead of Windows messages.

Change the server to open a socket and to listen for connections to it and accept incoming connections from the client processes. Note that all of the clients will have a connection to the server at the same time so you will need to deal with this.

The server should be able to simultaneously deal with Windows messages from the original client process and messages through the socket from multiple client processes from this part.

The new clients should send messages (in the form of structured data/C-style `structs`) rather than the Windows messages. The client should add `5` to all process IDs which are sent (i.e. rather than sending thread values `1` to `5`, instead send process numbers `6` to `10`) and should add `6` to all resource IDs which

are sent. The information in the messages (except for the add 5 or 6 mentioned above) and the circumstances under which of the messages are sent will be the same and the server is responsible for dealing with the messages appropriately.

You will send the message(s) when the `SendDataToServer()` function is called. You may if you wish send the information as one message (telling the server the process which called and the 6 resource counts) or as 7 messages (one telling it the process ID and 6 telling it the resource counts) or any other number of messages you think is sensible. Whatever you decide, you need to send the information of: which process called the function (remember to add 5 to the ID, to get IDs from 6 to 10), and the usage counts of all 6 resources at that time (remember to add 6 to the resource ID if you send it, to get an ID from 7 to 12).

The server should accept the message(s) from the clients and should increment the count of which process sent the message (using IDs 6 to 10, and not 1 to 5) and what the resource counts are. When altering the resource counts, again you should ensure that you only set the value if the new value is larger than the old value, and should protect these counts from interference from other threads if necessary (see the comments in Part B about this).

C3: Run the program and test it fully

Run the server and both sets of client processes/threads and test that your server correctly shows the progress of the different processes.

Ensure that the display in the windows is showing the current thread/process activity counts and the current resource counts, considering the 6 new resources and the 5 new processes as resources 7 to 12 and processes 6 to 10. Note that your answer to Part B may already do this.

Submission

Ensure that you have fully completed the documentation file, since it will be used for the marking.

Create a single zip file containing your documentation file and all three projects for your server, your Part A client and your Part C client program. We will need to rebuild these with our own version of the `DoNotChangeThis.h` file, to test your submission, so your programs need to build and you must not have added any of your code to this file. Test your programs by putting the original file back in and making sure that your programs still run correctly.

Ensure that you delete the `.sdf` file ("SQL Server Compact Edition Database File") from the solution(s) you submit, and anything in the `Debug/` and `Release/` directories before you submit.

You will submit your single zip file through the University of Nottingham, Computer Science, TMA submission system at: https://tma.cs.nott.ac.uk

Marking criteria

Marks will be allocated as follows:

Part A: 40% (the initial client)

Part B: 25% (the initial server)

Part C: 35% (the modified client and server)

In each case the marking will be on the basis of how well you fulfilled the requirements, whether your program works and whether it does what it should do both correctly and efficiently, and for the correct reasons. To assess these, we will consider both the submitted code and the explanations and

justifications which you include in your documentation file. In particular, we will check that your code works correctly, will look for specific inefficiencies and check your explanations to verify that you understood what you did and that you did it for the correct reasons.

When considering how this will be marked, please be aware that it is important that:

A) We can compile and run your programs. Without this there not much else that we can do.
B) Your programs do not crash. Without this it will be hard for us to give you many marks.
C) We can understand the code that you have written. Use the documentation to explain this.
D) We understand why you wrote the code in the way that you did. Again use the documentation to explain this.
E) Your code works correctly. E.g. that you correctly identify the critical sections, enforce mutual exclusion across them, and get the correct results (shown in the server windows).
F) Your code does not lock things unnecessarily.
G) Your code does not hang. I deliberately added some cases where livelock or deadlock is possible so that you learn how to identify the problems and avoid it.

Even more than single-threaded programming, in concurrent programming it is important to understand what you are doing and think through the consequences. You cannot guarantee that a problem will occur in testing when it relies upon specific timing. Be very careful that you understand what you are doing and explain it properly in your documentation file.

Hints and getting started:

Ensure that you do the lab exercises first. They give you sample code which you can copy-paste. Ensure that you understand not only how to finish the exercises but what they actually do and why.

There are no complex programming requirements in this coursework which you have not already seen code to do. You can successfully complete this without using any API functions which you have not already used.

I suggest that you could get started by following these steps:

1. Run the demo program to get an example of the sort of thing you will be creating. Note that I have given you a cut-down demo not the full version so do not expect it to have the correct counts. Also note that you will need the Visual Studio libraries installed for this to work (e.g. have installed V.S. itself).
2. Start your development by creating a new project for the client and a new project for the server.
3. Copy all of the contents of the file called `CopyFromThis.cpp` into your main file for your client.
4. Now create your client process and create five threads, which run the `Procedure1` to `Procedure5` functions.
5. Add code to maintain some counts for the number of times that resources are used.
6. Go through each of the `TestCase` functions and add code to increment the usage count on the relevant resources.
7. Try running it and add some debug information to see what it is doing – e.g. output the resource counts to the console.
8. Implement the server and the communication mechanism between the client and server. When you are ready to draw the contents of the window, copy the contents of the file called `ForYourWindowsMessageServer.cpp` into your `WM_PAINT` handlers for the relevant windows.
9. Implement the rest of parts A and B and get the system working before moving on to Part C.