Knowist Academy

www.knowist.ac

Windows Multithreading Using C/C++

LAB EXERCISES

:

Lab 1: Process Family Tree

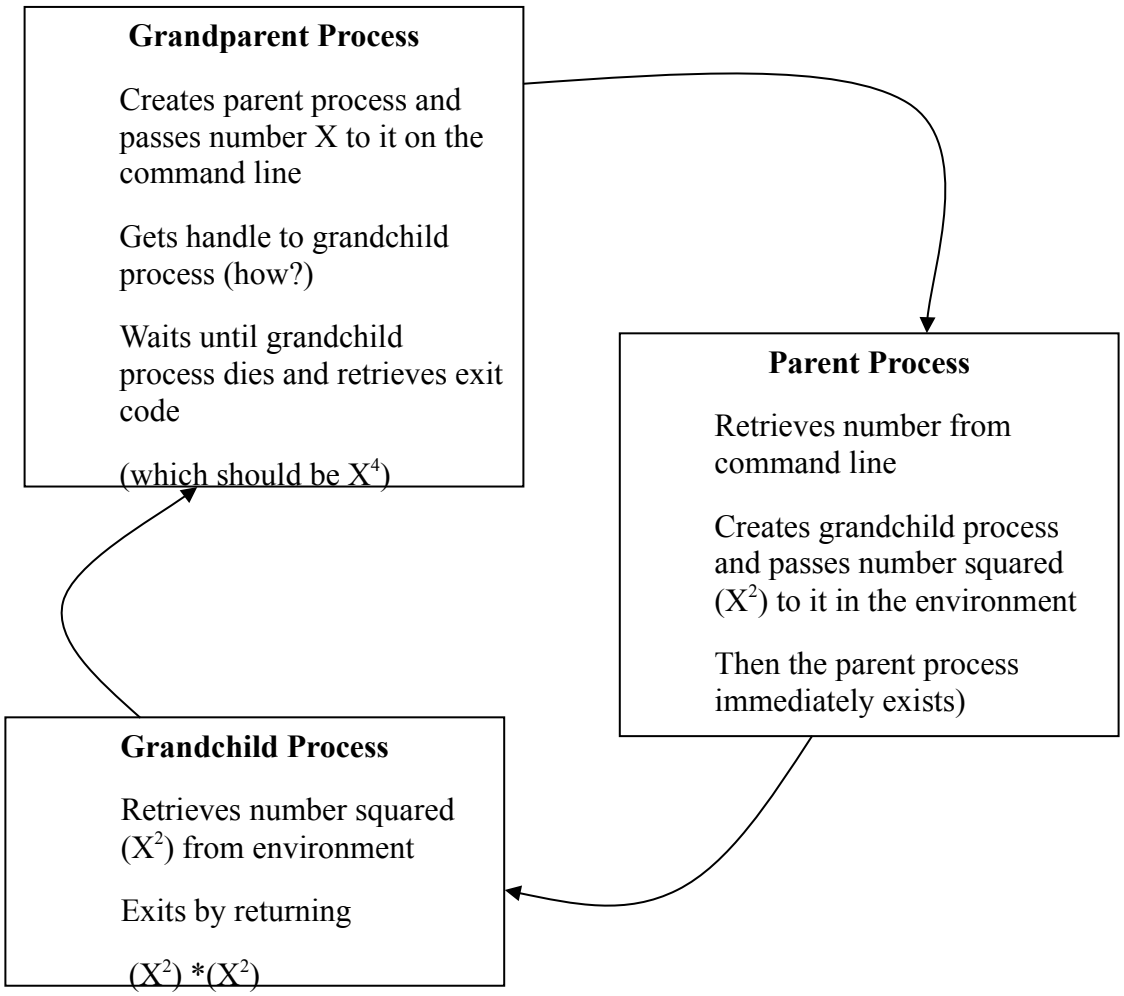
Duration of Lab: 45 Minutes

Aim of Lab: Explore Win32 process creation & lifetime management and see how to extract information from the command line and the environment.

The ProcessFamilyTree workspace contains three projects: Grandparent, Parent and Grandchild. Most of the code in these projects has already been written for you – you need to complete the projects with a few lines of code in appropriate places.

The grandparent process should create the parent process, which in turn creates the grandchild process. A number should be passed along this process hierarchy and squared in each process, and the results should be passed back from grandchild via its exitcode to the grandparent, without the in-between parent intervening.

Apart from the environment, command line and exit code, no form of IPC, re-directed stdio, the registry, files or any other communications mechanism may be used. The `DuplicateHandle` API may not be used.



Lab 1.1: Write Grandparent

Grandparent.exe launches Parent.exe in a new process, and passes a number to it via the command line. Then it calls `OpenProcess` to get a handle to the grandchild process, and waits on it to become signaled, and finally calls `GetExitCodeProcess`.

```
int main(){
 ...
 // Put nNumberToSquare in command line of new process
 sprintf(strCmdLine, "%s %d", PARENT_EXE, NUMBER_TO_SQUARE);

 ZeroMemory(&si, sizeof(si));
 si.cb = sizeof(si);

 // Launch process running parent.exe
 CreateProcess(...);

 // Retrieve process ID of grandchild process (how?)
 ...

CloseHandle(...);

 // Open handle to grandchild process
 hGrandchildProcess = OpenProcess(...);
 // Wait until grandchild dies

 ...
 bRetVal = GetExitCodeProcess(hGrandchildProcess, &dwExitCode);
 if (bRetVal==0)
 {
       printf("GRANDPARENT: Error retrieving exit code \
             of grandchild process - exiting\n");
       print_err();
       getch(); // wait until user can read error from console
       return 0;
 }
 CloseHandle(hGrandchildProcess);
 printf("GRANDPARENT: Success! Number received from \
       grandchild = %lu\n", dwExitCode);
 printf("GRANDPARENT: (which is %d * %d * %d)\n",
       NUMBER_TO_SQUARE, NUMBER_TO_SQUARE, NUMBER_TO_SQUARE);

 getch(); // wait until user can read results from console
 return 0;
}
```

Lab 1.2: Write Parent

Parent.exe launches Grandchild.exe in a process. The parent receives a number from grandparent via the command line, squares it and passes this number to the grandchild in the environment. The parent process should then immediately die.

We should be extremely skeptical about information we read from outside – it could easily contain errors. A well known security defect in earlier versions of web-servers was that they read data from socket, which overwrote the end of the buffer – this data contain memory instructions, which were then executed within the web-server. Treat all values received from the command line, environment, IPC, sockets, etc. with upmost suspicion.

```
int get_num_from_command_line(int *nNumberToSquare)
{
char *strCommandLine;
 // retrieve command line
 strCommandLine = GetCommandLine();
 // extract number from command line
 ...
 // convert to integer
 ...
}

int main()
{
 ...
 nResult = get_num_from_command_line(&nNumberToSquare);
 if (nResult == 0)
 {
       printf("Exiting with error\n");
       return -1; // Not a valid thread ID
 }

 // square number from grandparent
 nNumberToSquare = nNumberToSquare * nNumberToSquare;

 // Put nNumberToSquare in grandchild's environment
 ...

 ZeroMemory(&si, sizeof(si));
 si.cb = sizeof(si);

 // Launch grandchild process
 bRetVal = CreateProcess (NULL, GRANDCHILD_EXE, NULL, NULL,
                          FALSE, 0, NULL, NULL, &si, &piProcess);
```

```
 if (!bRetVal){
       printf("PARENT: Error launching child\n");
       return -1;
 }

 // return process ID of process running grandchild.exe
    to grandparent
 ....
 // The parent can immediately die
}
```

Lab 1.3: Write GrandchildEXE

The GrandchildExe process takes number from the environment, squares it, and returns it as exit code.

```
int get_num_from_environment(int *nNumberToSquare){
 ...
 // retrieve envionment variable
 nResult = GetEnvironmentVariable(
                  NAME_OF_NUMBER_VARIABLE,
                  strEnvBuffer,
                  ENV_BUF_LEN);

 // Check return value
 if (nResult ...)
       ...
 // Convert to integer
       ...
}
int main()
{
int nResult;
int nNumberToSquare;
int nSquaredNumber;

 nResult = get_num_from_environment(&nNumberToSquare);
 if (nResult == 0)
 {
       printf("GRANDCHILD: Exiting with error\n");
       return -1; // cannot be a valid double number (-1*-1 = 1!)
 }
 nSquaredNumber = nNumberToSquare * nNumberToSquare;
 printf("GRANDCHILD: succeeded - returning %d\n", nSquaredNumber);

 // Give grandparent a little time to open process handle
 // otherwise grandchild could exit first, and grandparent's
```

```
 // call to OpenProcess might fail as kernel object has
 // already been destroyed.
 ...
 return nSquaredNumber;
}
```

Auxiliary Questions for Lab 1

Q1.1: Would the new Job Object (available with Windows 2000) help in any way with this project?

Q1.2: List the kernel objects used in the three projects, grandparent, parent and grandchild, and indicate when they are created and destroyed and when their usage counts are incremented and decremented.

Action	Grandparent Kernel Object	Parent Kernel Object	Grandchild Kernel Object
1)	Does not exist	Does not exist	Does not exist
2) Grandparent process launched			
3) Grandparent calls CreateProcess for parent			
4) Parent calls CreateProcess for grandchild			
5) Parent exits			
6) Grandparent detects calls CloseHandle on handle to parent			
7) Grandparent calls OpenProcess to retrieve handle to grandchild's kernel object			
8) Grandchild exits			
9) Grandparent retrieves exit code. Calls CloseHandle on handle to grandchild process kernel object			
10) Grandparent process exits			

Q1.3: After the grandparent process launches the parent process, it tries to get a handle to the grandchild process using the OpenProcess API. How can you ensure that the parent process has created the grandchild process before the grandparent calls OpenProcess?

Q1.4: When parsing the command line, what is the significant difference between argv/argc passed to main and the value returned by lpCommandLine passed to WinMain (apart from the fact that argv is an array of pointers and lpCommandLine is a single string)?

Lab 2: ThreadCounter

Duration of Lab: 45 Minutes

Aim of Lab: Explore Win32 thread creation and optimizing number of threads

The ThreadCounter workspace contains two projects which add numbers in different ways.

The SequentialCount project adds numbers between 1 and 100 (i.e. result=1+2+3..+100). This is a compute-bound problem, and the work can easily be divided into independent sections. For example, if three threads are used – the first thread can add from 1 to 33, the second thread from 34 to 66 and the third thread from 67 to 100. The important point here is that the second and third threads can execute in parallel with the first thread – because they know to start counting from 34 and 67 respectively and do not have to wait for the first thread to complete. Each thread must return its sub-total, which should be summed together to produce the result. The ExcessiveCount project is a one-line modification of SequentialCount which uses an un-necessarily high fixed number of threads. The performance difference should be clear.

The BlockingCount project reads numbers from a file and adds them. This is an I/O bound problem. (Note: For this lab, use blocking I/O. We will explore in a later lab how to use asynchronous I/O).

Lab 2.1: SequentialCount

SequentialCount adds the numbers between 1 and 100, using all the available CPUs in a machine. As it is a compute-bound problem, it makes sense to create one thread per CPU – each thread can work continuously (fully productively) without blocking for I/O. There is no need to create more threads than CPUs as this will only result in extra context switching and deterioration in performance.

The range of values (two shorts) that each thread is meant to summate is passed by value into it as the (long) parameter to the thread, and the total is passed back from the thread as its exit code.

```
DWORD WINAPI AddThreadProc(//Return becomes exit code for thread
  LPVOID lpParameter       // thread parameter
){
 ...;
 return nSubtotal;
}

int main(){
#define UPPER_BOUND 100
#define LOWER_BOUND 1
 ...;
```

```
 GetSystemTime(&SystimeBefore); // measure start time

// First need to calculate the number of CPUs in the machine
 GetSystemInfo(&sysinfo);
 nProcessors = sysinfo.dwNumberOfProcessors;
 ...;
 // Then allocate enough memory to store handles to the threads
 ThreadList = (HANDLE*) malloc(nProcessors*sizeof(HANDLE));
 if (ThreadList==NULL)
 {
       printf("Error malloc'ing memory - exiting\n");
       return 0;
 }

 // Create the threads and pass in the range they must calculate
 ...;

 for (i=0; i< nProcessors-1; i++){
       ...;
       ThreadList[i]=CreateThread(...);
       if (ThreadList[i]==NULL)
             printf("Error creating thread number %d\n", i);
             ...;
 }

 // Last thread is also assigned the remainder all the
 // way to upper bound
 ...;
 ThreadList[i]=CreateThread(...);
 if (ThreadList[i]==NULL)
             printf("Error creating thread number %d\n", i);

 // wait for threads to exit and sum exit codes
 for (i=0; i< nProcessors; i++)
 {
       WaitForSingleObject(ThreadList[i], INFINITE);
       GetExitCodeThread(ThreadList[i], &dwExitCode);
       nTotal += dwExitCode;
       CloseHandle(ThreadList[i]);
 }
 printf("Total of adding %d to %d is %d\n",
                   LOWER_BOUND, UPPER_BOUND, nTotal);
 ...;
 return 0;
}
```

Lab 2.2: ExcessiveCount

Change SequentialCount, so that instead of using `GetSystemInfo` and `dwNumberOfProcessors` to create one thread per CPU, this time hard-code 40 as the number of threads to use.

Measure the performance using `GetSystemTime`.

Lab 2.3: BlockingCount

The 10 text files, num1.txt, num2.txt … num10.txt each contain ten lines of text – each line consists of a number and a carriage return. BlockingCount should summate all the numbers from all the files. Blocking I/O must be used for this project (fopen and fgets).

```
#define NUM_FILES 10
#define FILE_PREFIX"FileWithNumbers"
#define BUF_LEN 200

DWORD WINAPI AddThreadProc(//Return becomes exit code for thread
  LPVOID lpParameter     // thread parameter
){
unsigned long nSubtotal=0;
int numFile = (int) lpParameter;
char strFilename[BUF_LEN];
char strNumbers[BUF_LEN];
long num;
FILE *fp;

 sprintf(strFilename, "%s%d.txt", FILE_PREFIX, numFile);
 if ((fp=fopen(strFilename, "r"))==NULL){
       printf("Error opening %s\n", strFilename);
       return 0;
 }

 ...;

 fclose(fp);
 return nSubtotal;
}
int main(){

 GetSystemTime(&SystimeBefore);
 for (i=0; i< NUM_FILES; i++){
             ThreadList[i]=CreateThread(...);
       if (ThreadList[i]==NULL)
             printf("Error creating thread number %d\n", i);
```

```
  }

  // wait for threads to exit and sum exit codes
  for (i=0; i< NUM_FILES; i++)
  {
        ...;
  }
  // print total, and measure time spent
  ...;

}
```

Auxiliary Questions for Lab 2

Q2.1: Imagine you needed to design a project called DependentCount which adds the numbers between 1 and 100 and after each addition it calculates the modulus of 5 (e.g. ((((1+2%5)+3%5)..+99%5)+100%5). The important point here is that the result at each stage is dependent on the results of a previous stage. Briefly describe a multithreaded approach to this project.

Q2.2: Avoid the obvious problem when you get a brand new PC for Christmas in the year 2022. (Most developers make an assumption about this problem, which is correct now but not in the future, [unlike the y2k and 2038 (when time_t hits zero again) problems, the 2022 problem is not really time related]).

Q2.3: Explain the performance differences between SequentialCount, ExcessiveCount and BlockingCount.

Q2.4: Is it possible that on a dual processor machine SequentialCount will run faster using one thread rather than two? Explain.

Lab 3: MultiSync

Duration of Lab: 45 Minutes

Aim of Lab: Explore Win32 synchronization techniques

The MultiSync workspace consists of a group of projects which explore how to use critical sections, mutexes and events.

Lab 3.1: CritSecSync

The CritSecSync project implements a simple associative database consisting of a single table which stores names of people based on a key. A fully working implementation is already provided – you will need to make one minor adjustment.

Each row consists of three fields - a key which uniquely identifies it, a firstname and a surname. The table can hold a fixed number of rows and storage is provided as a simple fixed-sized array.

```
#define DB_ROW_COUNT 20
typedef struct {
 int key;
 char firstname[MAX_FIRSTNAME_LEN];
 char surname[MAX_SURNAME_LEN];
} row_t;
row_t database[DB_ROW_COUNT];
```

A simple text file containing commands (`ADD, READ, LIST, UPDATE, REMOVE`) and parameters is used to edit the data. The following three lines add a new entry with key=44, firstname=james and surname=smith, then removes the entry with key=65 and then lists all the entries on stdout:

```
 ADD 44 James Smith
 REMOVE 65
 LIST
```

A thread reads this file line by line, and executes the appropriate function. For `ADD`, the `add_row` function is called:

```
void add_row(int key, char *firstname, char *surname){
int index;
 index = retrieve_empty_row(key);
 if (index == -1)
       return;
 strcpy(database[index].firstname, firstname);
 strcpy(database[index].surname, surname);
}
```

Multiple threads can run concurrently parsing separate text files. A full implementation of the CritSecSync project is already provided for you – including file handling, multiple threads, table editing.

You have to change one feature of it. The version provided uses table-level locking (course granularity). There is only one global critical section used for synchronization for table editing. Once a thread needs to execute a command on any row in the table, it locks the entire table. If another thread wishes to edit a different row, it will try to lock the same critical section – but has to wait until the first thread is finished.

Your job is to change table-level locking to row level locking (fine granularity), so that different threads can work in parallel editing different rows.

You need to add a `CRITICAL_SECTION` field to the `row_t` structure, and insert `EnterCriticalSection / LeaveCriticalSection` calls as appropriate to ensure that a thread only edits a row once it has locked that row's critical section. Remove the definition of the global `CRITICAL_SECTION` field, and the calls to `EnterCriticalSection / LeaveCriticalSection` for it.

Lab 3.2: SimpleMutexSync

This lab is a re-implementation of the row-locking solution to CritSecSync, except this time using mutexes instead of critical sections.

You will need to replace:

- the `CRITICAL_SECTION` parameter with a `HANDLE` definition (which represents a mutex),
- the `InitializeCriticalSection` call with a `CreateMutex` call,
- the `DeleteCriticalSection` call with a `CloseHandle` call
- the `EnterCriticalSection` call with a `WaitForSingleObject` call
- the `LeaveCriticalSection` call with a `ReleaseMutex` call

Lab 3.3: DistributedMutexSync

A mutex may be used to synchronies multiple processes.

The DistributedMutexSync project creates a mutex, and then launches a process running a new instance of itself – and the two processes compete for the same mutex – and simulate complex workloads using a shared resource (by calling Sleep and printf).

Complete DistributedMutexSync by inserting the appropriate Mutex calls at the correct locations.

Lab 3.4: SequentialCountEventSync

The SequentialCount (Lab 2.1) project added the numbers between 1 and 100. A primary thread created a number of threads, each of which summated a range of numbers. If four threads were used, the ranges would be (1,25), (26,50), (51,75) and (76,100). The primary thread calculates the ranges. The primary thread passed by value the range (two 16-bit shorts) to each secondary thread as the (32-bit) parameter field in the threadproc. No synchronization was needed.

For this lab, assume the range consisted of two 32-bit longs, so both cannot be passed in one 32-bit value as the parameter field in the threadproc. You will need to implement another approach so that the primary thread can safely pass the range to the secondary threads.

Start by defining a `range_t` structure to hold the upper and lower bounds of the range of values that each thread is meant to calculate.

```
typedef struct {
     long lower_bound;
     long upper_bound;
} range_t;
```

Two techniques which you should not use because they will not work are:

(A) Using TLS – the primary thread cannot write into a secondary thread's TLS

(B) Use mutex/critical section synchronisation to protect access to a single `range_t` structure – you might assume the ordering:

```
Primary thread enters critical section
Primary thread writes values for secondary thread x into range
Primary thread leaves critical section
secondary thread x enters critical section
secondary thread x reads values for secondary thread x from range
secondary thread x leaves critical section
Primary thread enters critical section
Primary thread writes range for secondary thread y into range
Primary thread leaves critical section
secondary thread y enters critical section
secondary thread y reads values for secondary thread y from range
secondary thread y leaves critical section
```

Depending on timeslicing, what could equally well happen is:

```
Primary thread enters critical section
Primary thread writes values for secondary thread x into range
Primary thread leaves critical section
Primary thread enters critical section
Primary thread writes range for secondary thread y into range
```

```
Primary thread leaves critical section
secondary thread x enters critical section
secondary thread x reads values for secondary thread y from range
secondary thread x leaves critical section
secondary thread y enters critical section
secondary thread y reads values for secondary thread y from range
secondary thread y leaves critical section
```

Note that the primary thread wrote the range for secondary thread y before secondary thread x had read its range, which resulted in secondary thread x incorrectly reading the range for secondary thread y. *(Motto: don't assume anything about the ordering of thread execution, useless you specifically enforce it using synchronization!)*

Critical sections and mutexes ensure that only one thread at a time is reading or writing from/to a variable – they do **NOT** ensure ordering between reads and writes from different threads. In this problem we do need such ordering – first the primary thread must write to the variable – then the secondary thread x reads from the variable – then the primary thread writes to the variable – then the secondary thread y reads from the variable. Any other ordering will produce incorrect results.

Two techniques, which would work if we could use the parameter field to the threadproc, are:

(C) Create an array of `range_t` structures large enough to store one per thread. This should be a global array, accessible to all threads. When the primary thread is launching each thread, it writes the range for that thread into the array at an appropriate index, and passes the index as a parameter to the secondary thread, which then can read its range from the array and summate the values. Different secondary threads will be passed different indices into the array, so regardless of the order of processing they will not disturb other threads.

(D) As technique C, but use an automatic variable to store the array within the primary thread, and pass a pointer to a field within the array as the parameter to the threadproc. Note that the thread on whose stack the variable is created must out-live all other threads that use it, or else an access violation error could occur. Also note that both (D) and (E) require that the index is passed to the threadproc as the parameter field.

Assume for this lab that the parameter field to the threadproc is needed for something else and cannot be used as an index.

(E) Create a single global variable of type range_t, and use events to enforce proper ordering

To complete SequentialCountEventSync, copy your solution from SequentialCount (Lab 2.1) and change it to use technique (E).

```
typedef struct {
long lower_bound;
long upper_bound;
} range_t;

range_t therange;

HANDLE hEmptyEvent;

DWORD WINAPI AddThreadProc(// Return becomes thread's exit code
  LPVOID lpParameter     // thread parameter
){
unsigned long nLowerBound;
unsigned long nUpperBound;
unsigned long nSubtotal=0;
unsigned long nCount;

 // retrieve values from range for this thread
 nLowerBound = therange.lower_bound;
 nUpperBound = therange.upper_bound;

 // make event signalled, so primary thread can load range
 // with values for next new thread
 ...;
 nUpperBound++; // one past upper bound
 for (nCount = nLowerBound; nCount < nUpperBound; nCount++){
       nSubtotal += nCount;
 }
 return nSubtotal;
}
int main(){
// variable list as before
HANDLE *ThreadList;
 // GetSystemInfo etc as before
 //Init event which is used to signal that the range is empty
 hEmptyEvent = ...;
 // Create the threads and pass in the range they must calculate
 nIncrement = (UPPER_BOUND - LOWER_BOUND+1) / nProcessors;
 nCurrentLowerBound = LOWER_BOUND;
 for (i=0; i< nProcessors-1; i++){
       // wait until range is empty
       ...;
       therange.lower_bound = nCurrentLowerBound;
       therange.upper_bound = nCurrentLowerBound+nIncrement-1;
       ThreadList[i]=CreateThread(NULL, 0, AddThreadProc, NULL,
```

```
                    0, &dwThreadID);
        if (ThreadList[i]==NULL)
              printf("Error creating thread number %d\n", i);
        nCurrentLowerBound += nIncrement;
  }
  // wait for threads to exit and sum exit codes
  // as before. When we are finished with event, cleanup
  ...;
  // Display Total + cleanup as before
  return 0;
}
```

Auxiliary Questions for Lab 3

Q3.1: Many database engines support page-level locking. Briefly outline the changes needed in the design of CritSecSync (Lab 3.1) to implement this.

Q3.2: Explain the performance differences between CritSecSync (Lab 3.1) and SimpleMutexSync (Lab 3.2).

Q3.3: Imagine a function ConvertToUppercase is called to convert a surname in a row to uppercase. Assume it is used sometimes from within the `add_row` function, code that owns the critical section/mutex, and other times ConvertToUppercase needs to be called from code that does not own the critical section/mutex. Describe how it interacts with critical sections/mutexes to safely edit the row.

Q3.4: An alternative solution to Lab 3.4 is to use the ThreadID as an index into some storage which holds the range information. The primary thread knows the ThreadID of the new secondary thread via the output parameter in CreateThread, and the secondary thread can retrieve its own threadID by calling GetThreadID(). One problem is that the primary thread does not know ThreadID until CreateThread returns. Describe how you could ensure that the primary thread has written the range values into storage based on the secondary thread's thread ID, before the secondary thread tries to read from it.

Lab 4: TLS and DLLs

Duration of Lab: 45 Minutes

Aim of Lab: Explore how thread local storage (TLS) works, and how DLLs and threads interact.

The TlsAndDll workspace contains six projects templates for use in this lab. You will need to change TlsNoSync for lab 4.1, DllMainDemo for lab 4.2, DllSharedSections & DemoEXEforSharedSections for lab 4.3. DemoEXEforDllMain is a test executable for lab 4.2. You do not have to change it. You will need to examine the Newbie project when answering auxiliary question 4.4.

Lab 4.1: TlsNoSync

The TlsNoSync project contains the code from the CritSecSync project (Lab 3.1). You should change it so that no critical sections or mutexes at all are needed, which can be achieved by maintaining an independent database per thread – using thread local storage. Important: The function prototypes of the code may not be changed in any way.

First you will need to remove the Critical Sections function calls. Then you should add once-off initialization of the thread local storage by calling `TlsAlloc` in one thread. Then for each thread call malloc to allocate enough memory for 20 rows in the database and use `TlsSetValue` to store the pointer. Finally add `TlsGetValue` calls as appropriate within the various functions which interact with the database.

Lab 4.2: DllMain Issues

The DllMainDemo project has a bug in it. It creates a temporary thread within the DllMain function and waits on it to terminate. As calls to DllMain are serialized, and as the caller thread in already inside DllMain, the newly created thread will also try to call DllMain and block – but the caller thread is blocked awaiting the newly created thread to terminate – thus deadlock results. Fix the bug – by creating another function in DllMainDemo, and call it explicitly from DemoEXEforDllMain executable, after DllMain returns.

Lab 4.3: Shared Sections in DLLs loaded into different processes

Usually, when the same DLL is loaded into different processes, new instances of the DLL's data variables are created per process. It is possible to use shared sections within the DLL, to signify that a single instance of a DLL's variables should be shared among all processes that load that DLL. To be shared, data variable MUST be provided with a default value (at compile time). However, sometimes the real initial value is not known until run-time and care must be taken that the initialization is done correctly.

DllSharedSections is a DLL with a variable whose value should be loaded from a data file during start-up. Two instances of DemoEXEforSharedSections are launched, and they both load the DLL. Use a mutex to ensure that only one of them loads the value from the file and the other blocks until this has been done.

Auxiliary Questions for Lab 4

Q4.1: Is it possible to complete Lab 4.1 without using TLS? Explain how and discuss the benefits/problems associated with alternative(s).

Q4.2: TLS maintains an integer per thread. Could you allocate one block of memory in a process, and save a pointer to it in each thread's TLS. What effect does this have on the application?

Q4.3: If a DLL is loaded by multiple processes, and uses a shared data section to share memory – what happens if one process crashes (e.g. divides a number by zero somewhere)?

Q4.4: You are now acknowledged as one of the leading multithreading developers in your development team. A new developer has joined and has fully coded up the Newbie project. Unfortunately it has many multithreading bugs in it. Help the new developer by listing the errors (hint: there are seven errors).

Error 1: ---

Error 2: ---

Error 3: ---

Error 4: ----------

Error 5: ----------

Error 6: ----------

Error 7: ----------

Lab 5: Asynchronous I/O

Duration of Lab: 45 Minutes

Aim of Lab: Using asynchronous I/O to do more work per thread's timeslice

When we encounter a project that requires multiple I/O calls, we can use asynchronous I/O techniques to our advantage. The concept involves packing processing of multiple I/O requests inside one thread, rather than having one thread per I/O request. It is quite likely that we have to concurrently manage more I/O requests than there are processors available and asynchronous I/O can be better than the alternative – which is to have one thread per I/O request, which results in a waste of CPU cycles due to excessive context switching.

The AsyncIO workspace for this lab explores different asynchronous I/O techniques using three projects – overlapped, scattergather and completionports.

Lab 5.1: Overlapped

The overlapped project uses three techniques to write 10 Mbytes of data to 10 files. The first technique uses calls to synchronous (blocking) I/O within a for loop in a single thread to write to the ten files. The second technique creates ten threads and uses synchronous I/O with each thread to write to one file. The full code for both these techniques is provided already for you.

The third technique creates one thread per processor, and assigns equal numbers of files to each thread. The threads use overlapping I/O to simultaneously write to a number of files. You need to complete the overlapped section of code – by filling in the WriteFile call and the synchronization needed.

Lab 5.2: Scattergather

The scattergather project writes 10 character buffers to the same file using the new WriteGather API, introduced with Windows 2000 or Windows NT 4 with SP2 or later. Complete the coding needed.

Lab 5.3: Completionports

The completionports project is used to simulate a client-server environment, where ten clients connect simultaneously to one server. The same EXE is used for the parent and ten children. A command line argument specifies what the EXE should do – if no command line argument is provided, then the EXE should run as the server. Otherwise, the command line argument is a number between 0 and 9 that identifies which client the process running the EXE should represent.

Communication is based on a named pipe, which is configured to handle 10 simultaneous connections. The server maintains an I/O completion port and creates one thread per processor to service the clients. Most of the code is written for you, but you must complete the code that interacts with the I/O completion ports. You

also need to decide whether the overlapped structure for the named pipe needs the event field filled in, or can you simply rely on the pipe itself becoming signaled.

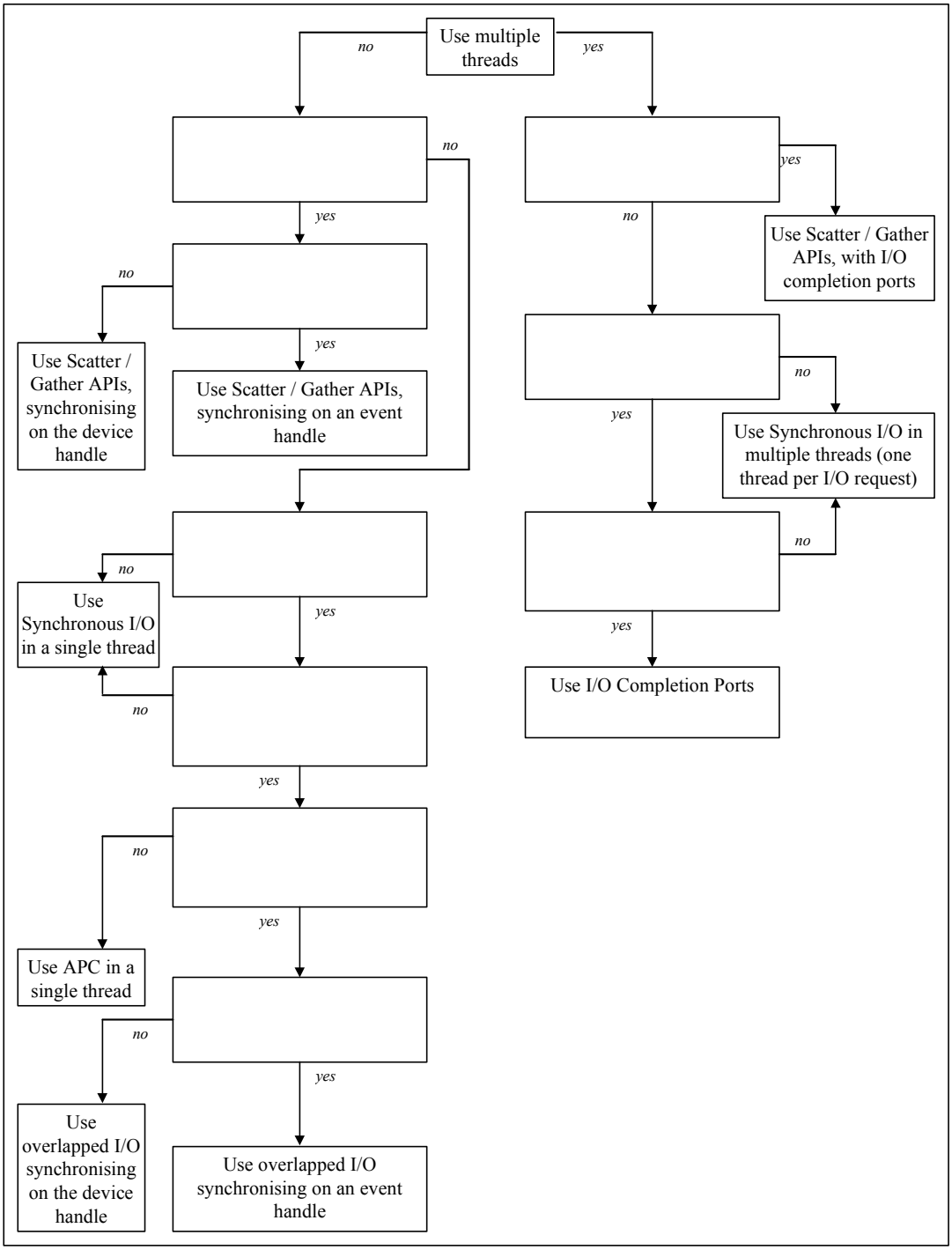
Auxiliary Questions for Lab 5

Q5.1: Explain the difference in performance between using the synchronous and asynchronous methods in lab 5.1.

Q5.2: Your project manager has asked you to provide an estimate of the difference in development resources needed for three different approaches to I/O: (1) the single thread with multiple synchronous I/O calls approach, (2) the one thread per each synchronous I/O request approach, and (3) the overlapped I/O approach. What is your answer?

Q5.3: Outline why scatter/gather is the optimal disk to memory transfer method and why I/O completion ports are recommended for high-throughput server platforms.

Q5.4: Complete this flowchart showing how to make a decision on which I/O technique to use for various scenarios.



Lab 6: Multithreading Architectures

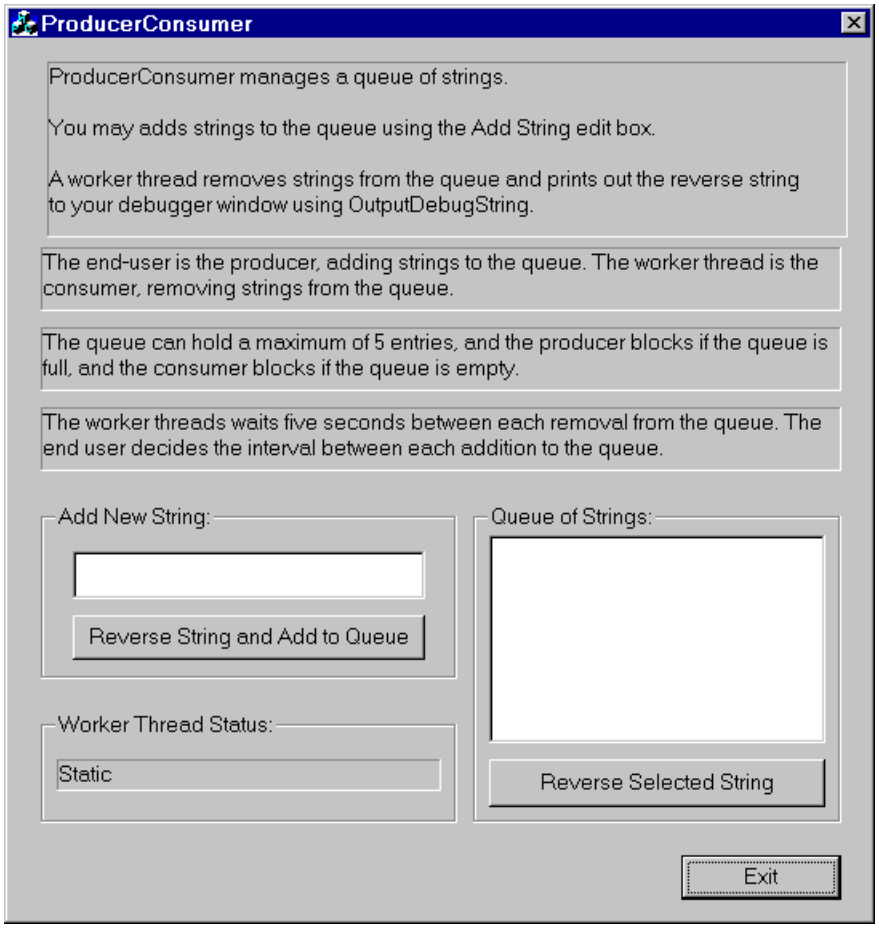
Duration of Lab: 45 Minutes

Aim of Lab: Examine higher-level architectural issues with multithreading.

The architecture workspace contains three projects – comsumerproducer, smartpointers and innervision.

Lab 6.1: ConsumerProducer

The producer-consumer architecture is one method to handle workloads using multiple threads. One or more threads, known as producers, add workitems to a queue, and one or more other threads, known as consumers, remove work-items from the queue. The producers must pause when the queue is full, and the consumers must pause when the queue is empty.



This lab examines how to implement a simple version of a producer consumer design. The queue is modeled in the form of a Windows listbox, so we can see items being added and removed. Work items are modeled in the form of strings, which the consumer thread must reverse (e.g. “hello” becomes “olleh”), and print to the console. The producer is the end-user, who may add items using controls in the

dialog box. The consumer is a worker thread, which removes items from the queue based on a timer.

As an additional complication, the end-user may select a string from the queue and click the "Reverse Selected String" pushbutton to have it reversed. This models a very frequent requirement that functions (in this case, the function you write to reverse the characters in a string), may be called from different pieces of code, some of which may have acquired the necessary synchronization object(s), and others may not.

One final problem is that when the user clicks the exit button, the worker thread should remove the remaining strings in the queue and print out their reverse strings, and only then should the process exit. (Suggestion: When the Exit button is clicked, the (currently empty) function CleanupRoutine will be called. Add an event, and set it to signaled within CleanupRoutine, and in the threadproc for the worker thread add this event handle to the list on which you wait in WaitForMultipleObjects).

To help you with this project, all the user interface is provided for you, and you must complete the threadproc for the worker thread and the code for queue management.

You must add code to the WorkerThreadProc and cleanup_routine functions.

Lab 6.2: Smartpointers

One major problem with multithreading is that the calls you make to use data structures are different form those you use to enter mutual exclusion areas.

It is good design practice to try to combine both, so that to get a pointer to a shared resource a developer must automatically lock the synchronization object protecting that resource.

One technique is to use smart pointers – when instantiated, they quietly lock the synchronization object and then provide access to the data structure. When they go out of scope, then the synchronization object is released.

The smartpointers project using a mutex to protect access to an instance of a class. A `CSyncSmartPtr` template is defined, which has to lock the mutex in its constructor, release it in its destructor, and override the pointer operator (->), to return a pointer to the protected resource. If an error occurs a `CsyncSmartPtrException` exception is to be thrown. The main function initializes the shared resource, and within a local block, instantiates the smart pointer (which locks the resource) and accesses the shared resource, and at the end of the local block the smart pointer goes out of scope and the shared resource becomes available again.

The project is complete except for the CSyncSmartPtr template, which you must write.

```
struct CSyncSmartPtrException {
```

```
 string strReason;
 CSyncSmartPtrException(string reason){strReason=reason;};
};

template <class T, class U>class CSyncSmartPtr {
. . . // YOU SHOULD ADD CODE HERE TO IMPLEMENT THE TEMPLATE!
};

typedef struct {
 int x;
 int y;
} MyPoint_t;

MyPoint_t g_mypoint;
// -------------
// Main Function
// -------------
int main(){
HANDLE hMutex=0;

 // before editing mypoint, must own hMutex
 hMutex = CreateMutex(NULL, FALSE, NULL);
 if (hMutex == NULL){
       printf("Error detected when creating mutex - exiting\n");
       return 1;
 }
 // initialisation
 g_mypoint.x = 3;
 g_mypoint.y = 6;
 try {
       CSyncSmartPtr<MyPoint_t *, HANDLE>
                                 sp(&g_mypoint, hMutex);
       sp->x = 5;
 }
 catch (CSyncSmartPtrException e)
 {
       cout<< e.strReason << "Exception caught\n";
 }     // msp goes out of scope here –
       // terminator called and mutex released

 cout << "result of operation " << g_mypoint.x;
 // cleanup
 CloseHandle(hMutex);
 getch();
 return 0;
}
```

Lab 6.3: Innervision

Innervision is an attempt at what might be called software radiography. The aim is to visualize what is happening inside an application as it executes. The application developer can then easily detect errors, see which areas require tuning and gain a better understanding of how the application performs with real workloads.

In the past many application developers have maintained a textual trace file, into which the application writes important programmer-targeted information. Streams of text from a trace file cannot be rapidly assimilated. This becomes especially problematic when we start developing multithreaded applications, with asynchronous I/O and very high-throughput. The trace files can get very long and different branches of code can generate trace simultaneously and it is difficult to grasp this parallelism from a linear trace file.

Human beings, including application developers, can pick up information much quicker and in greater quantity if it is presented in visual form. Hence the idea of software radiography, which presents an internal dynamic picture of how the application is functioning.

In theory, it is great, but how could it be implemented in practice? Neither the operating system nor any development tools (DevStudio, VC++ etc.) provide any functionality in this area. So it has to be all coded up manually.

A viewer application would need to be written which connects to all the applications that are to be examined. When important events occur in these applications, they send messages to the viewer that graphically displays it.

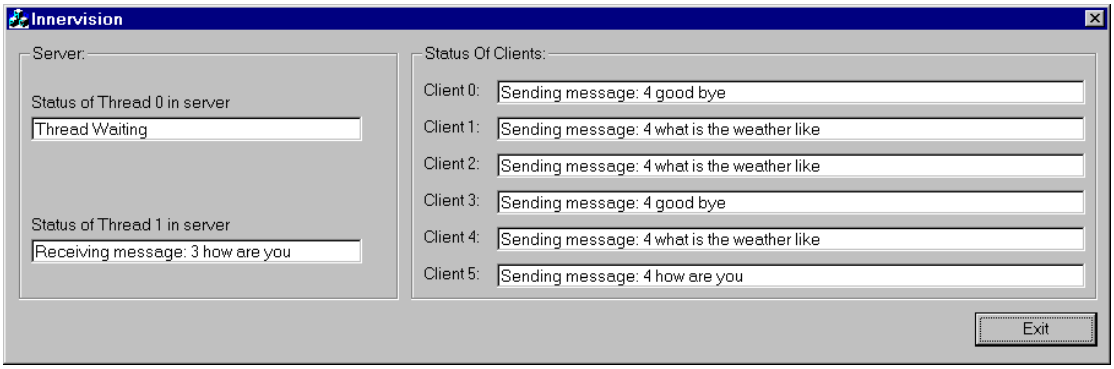
Some initial questions might be:

- Does the viewer have to be custom written per application?
- How does the viewer display the data?
- What types of messages are to be displayed?
- Is the display per work-item (as it flows through all components), per component (as all work-items flow through it) or per message type (generated in different components)?
- What happens if the volume of generated messages is larger than the capacity to display them (e.g. imagine a system handling millions of work items an hour)
- How can it be made non-intrusive (as much as possible)? Software radiography should not alter (too much) how the application runs (as extra commands have to be run, a time-slice might expire early)

Though the Innervision project is currently only a demo, it could evolve in the long term into a library of components, which eventually might provide a HTTP stream to which any web browser could connect (and hence no custom viewer need be

written). A number of interesting ideas (not explored in this lab) would be to use technology based on HTML+TIME, or the successor to the recently abandoned Microsoft Chromeffects (a library of XML-based 3D objects which uses the 3D hardware present in many new PCs), to be hosted by the Microsoft Management Console, and interaction with Web-Based Enterprise Management (WBEM), Windows Management Architecture and Windows Management Instrumentation. They need further study.

For now, Innervision will be custom written, and provides visualization for the completionports example of lab 5.3, which has one server process connecting to 6 client processes, and communicating using named pipes.



The Innervision project is a MFC-based GUI. It has been completed for you and you do not need to add to it in any way.

The InnervisionCompletionPorts project is a modified version of lab 5.3.

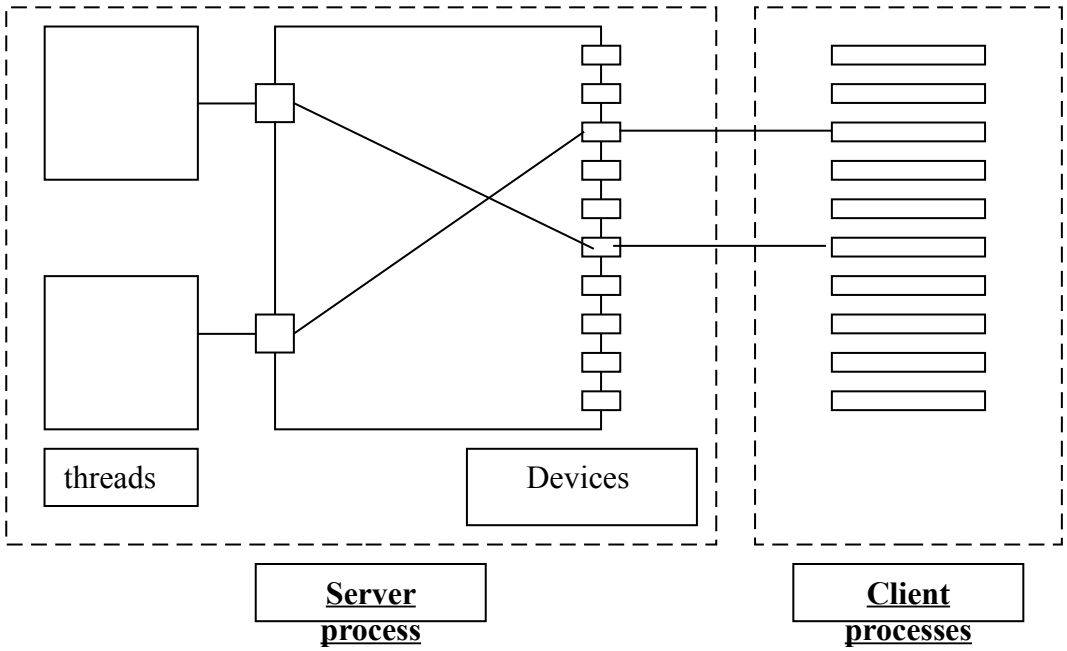
A library of calls (with the xray_ prefix) have been provided which allow InnervisionCompletionPorts to inform Innervision GUI of significant changes in a process's status.

```
HANDLE xray_connect(int nIdentifier);
void xray_disconnect(int nIdentifier, HANDLE hXrayPipe);
void xray_send_msg(HANDLE hXrayPipe, int msg_type, char *msg);
```

You task in this lab is to insert appropriate calls to these xray_ functions in InnervisionCompletionPorts.

How Innervision Works

The completion ports example consists of 7 processes, one server and six clients. The innervision browser is an additional GUI process that visualizes how these seven processes interact. This implementation of Innervision is hard-coded to handle this single scenario.



The first issue to decide is what exactly do we need to visualize, and how many active elements will there be in the picture – one element per process or multiple elements per process. The client process writes a string to a named pipe and this is all that needs to be visualized. The server process manages an I/O completion port and two threads which read the messages from the named pipes. Therefore Innervision needs to visualize eight elements – six named pipes from the clients, and in the server two threads (and possibly in a future version one I/O completion port).

These elements in the server and clients will need somehow to connect to the Innervision process and regularly send information about their status to it. This is a one-way communication. The easiest way to do this is to use named pipes. Each element will need to connect to the Innervision process and identify itself. To identify itself each element could send a message, or alternatively Innervision could know by using different named pipes, (instead of multiple instances of the same named pipes). We could use an identification message for this implementation. Other types of messages include when the element is about to disconnect, and when an important occurrence has happened to that element (e.g. completion port awoke a waiting thread to processing an incoming request). Message formats will be of the type <TAG> <DATA>. The TAGs allows are: CONNECT, DISCONNECT, MSG.

- The DATA for the CONNECT tag is a numeric identifier for the element (taken from the xray.h file).

- The DATA for the DISCONNECT tag is blank.
- The DATA for the MSG tag is a string.

Auxiliary Questions for Lab 6

Q6.1: There are numerous threading architectures available (producer-consumer, boss-worker, client-server). Draw a flowchart showing when to select which one.

Q6.2: SmartPointers solve the problem of acquiring and using a resource. Do they introduce any problem themselves?

Q6.3: When debugging and tracing, which tools does the OS/DevStudio problem to you, and which do you have to manually code up an infrastructure?

Q6.4: You have become the lead designer for a server platform. In the past, this server platform supported end-user configuration using the charge/stop/start technique (e.g. the server read the configuration information [e.g. from registry] at start-up only, and if the end-user wished to change a configuration value, to take effect they had to stop the server and restart it. Your marketing manager says this is no longer acceptable – you will have to support on-the-fly editing of configuration values (e.g. the end-user changes the configuration value, and it immediately takes effect, without have to re-start the server).

How would you propagate the "Configuration changed" message to all threads (considering some of them might be blocked awaiting I/O, some might be using I/O completion ports and some might be blocked in WaitForSingleObject / WaitForMultipleObjects calls waiting on kernel synchronization objects.

A second change is that in the past during shutdown the main thread called TerminateThread to kill all other threads. This is bad practice and you need to change this so that all threads are told to exit themselves. How does this work with implementations based on I/O completion ports, WaitForMultipleObjects and WaitForSingleObject.
