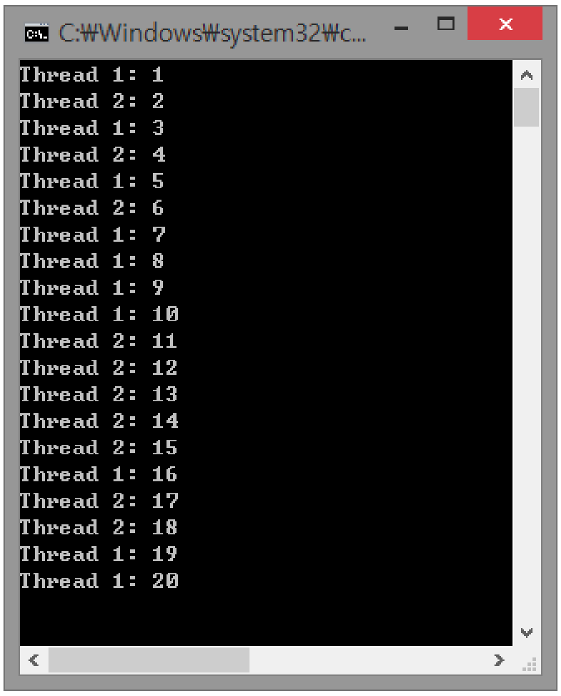
# Lab Exercise 5
- Multithread programming

In this lab, we will practice multi-thread programming and thread synchronization techniques.

## Exercise 1
*- Thread synchronization with `CRITICAL_SECTION`*

Complete the codes with `CRITICAL_SECTION` to synchronize the thread work such as the screen below;-



```
Thread 1: 1
Thread 2: 2
Thread 1: 3
Thread 2: 4
Thread 1: 5
Thread 2: 6
Thread 1: 7
Thread 1: 8
Thread 1: 9
Thread 1: 10
Thread 2: 11
Thread 2: 12
Thread 2: 13
Thread 2: 14
Thread 2: 15
Thread 1: 16
Thread 2: 17
Thread 2: 18
Thread 1: 19
Thread 1: 20
```

The running sequence of each thread will not be same as the picture because OS is managing the thread scheduling.

The original codes what you need to change are below;

```
#include <Windows.h>
#include <iostream>

using namespace std;

static int  g_n = 0;
CRITICAL_SECTION g_CS;

UINT ThreadOne(LPVOID lParam)
{
    for (int i = 0; i < 10; i++) {
        cout << "Thread 1: " << ++g_n << "\n";
    }
    return 0;
}

UINT ThreadTwo(LPVOID lParam)
{
    for (int i = 0; i < 10; i++) {
        cout << "Thread 2: " << ++g_n << "\n";
    }
    return 0;
}

int main(void)
{
    HANDLE hThread[2];
    unsigned long IDThread[2];

    hThread[0] = CreateThread(NULL, 0, (LPTHREAD_START_ROUTINE)ThreadOne,
        (LPVOID)NULL, 0, &IDThread[0]);
    hThread[1] = CreateThread(NULL, 0, (LPTHREAD_START_ROUTINE)ThreadTwo,
        (LPVOID)NULL, 0, &IDThread[1]);
    WaitForMultipleObjects(2, hThread, TRUE, INFINITE);

    return 0;
}
```

## Exercise 2

*- Thread synchronization : Multithread summation calculator*

Let’s make a simple program to find the summation from number 1 to some *input number*. Use one thread to calculate the summation and use the other thread to print out the result and main function is going to wait until all the thread ended.

Here is the code from this basic idea.

```
#include <Windows.h>
#include <iostream>

using namespace std;

unsigned int gNumberTo = 0;
unsigned int gSum = 0;

DWORD WINAPI sum_numbers( void *lpVoid )
{
    unsigned int Number = 1;

    while( Number <= gNumberTo )
    {
        gSum = gSum + Number;
        Number = Number + 1;
    }

    return 0;
}

```

```
DWORD WINAPI print_sum_numbers( void *lpVoid )
{
    cout << "The sum from 1 to " << gNumberTo << " is " << gSum << endl;

    return 0;
}

int main( void )
{
    HANDLE hThread[2];

    unsigned int Answer = 0;

    cout << "Calculate the summation from 1 to ";
    cin >> gNumberTo;

    hThread[0] = CreateThread( NULL, 0, sum_numbers, NULL, 0, NULL );
    hThread[1] = CreateThread( NULL, 0, print_sum_numbers, NULL, 0, NULL );

    WaitForMultipleObjects( 2, hThread, TRUE, INFINITE );

    for( unsigned int i = 0; i <= gNumberTo; i++ )
    {
        Answer += i;
    }
    cout << "[Answer] The sum from 1 to " << gNumberTo << " is " << Answer << endl;

    return 0;
}
```

If you input big number such as 1000000 then you can see the result of calculation from thread ‘*sum_numbers*’ and calculation inside the main function is different.
Fix the code and make two calculation (from thread ‘*sum_numbers*’ and inside the main function) meet the same number.

### Hint & Think

This is because thread ‘*print_sum_numbers*’ start to run before thread ‘*sum_numbers*’ finishes the calculation. You need to fix the codes the thread ‘*print_sum_numbers*’ must start after thread ‘*sum_numbers*’ finishes the calculation.

## Exercise 3

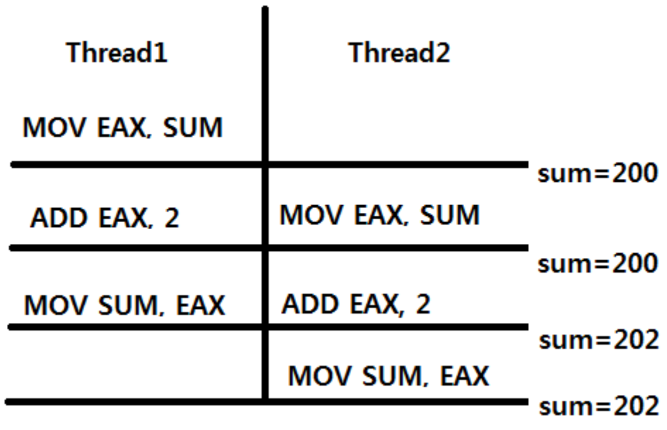
*- Data racing conditions and synchronization*

We know 25,000,000 x 2 x 2 = 100,000,000.
And the codes below are calculating this but if you try to run this codes you can see the result will not be shown exactly 100,000,000.

```
#include <Windows.h>
#include <iostream>

using namespace std;
```

```
#define EXPECTING_NUMBER 100000000 // 100M
#define THREAD_COUNT 2
#define LOOP_COUNT (EXPECTING_NUMBER / (THREAD_COUNT*2))
int sum = 0;

DWORD WINAPI ThreadFunction( void *arg )
{
    int i;
    for( i = 0; i < LOOP_COUNT; i++ ) {
        sum += 2;
    }
    return 0;
}

int main()
{
    HANDLE hThread[THREAD_COUNT];
    DWORD dwThreadID[THREAD_COUNT];
    int i;

    cout << "Calculating expectation result is " << EXPECTING_NUMBER
         << endl;
    for( i = 0; i < THREAD_COUNT; i++ ) {
        hThread[i] = CreateThread( NULL,
                                   0,
                                   ThreadFunction,
                                   NULL,
                                   0,
                                   &dwThreadID[i] );
        if( hThread == 0 ){
            cout << "CreateThread() error" << endl;
            exit( 1 );
        }
    }
    WaitForMultipleObjects( THREAD_COUNT, hThread, TRUE, INFINITE );
    cout << "Calculation result is with thread is " << sum << endl;
}
```

This is the problem of “data racing” and from the assembly codes it could cause the problem like this below;



Describe **WHY** this happen and **HOW** can we make it work properly. Write your descriptions at the front of your codes in comments (using ‘//’ or ‘/* */’).

Try to fix the codes from this problem and make the result of the summation into 100,000,000.

Hint & Think

A Data Race (race condition) occurs when two or more threads can access shared data and they try to change it at the same time. Because the thread scheduling algorithm can swap between threads at any time, you don't know the order in which the threads will attempt to access the shared data. Therefore, the result of the change in data is dependent on the thread scheduling algorithm.

You can see more informations:-
https://goo.gl/hWO7ZZ

## Exercise 4

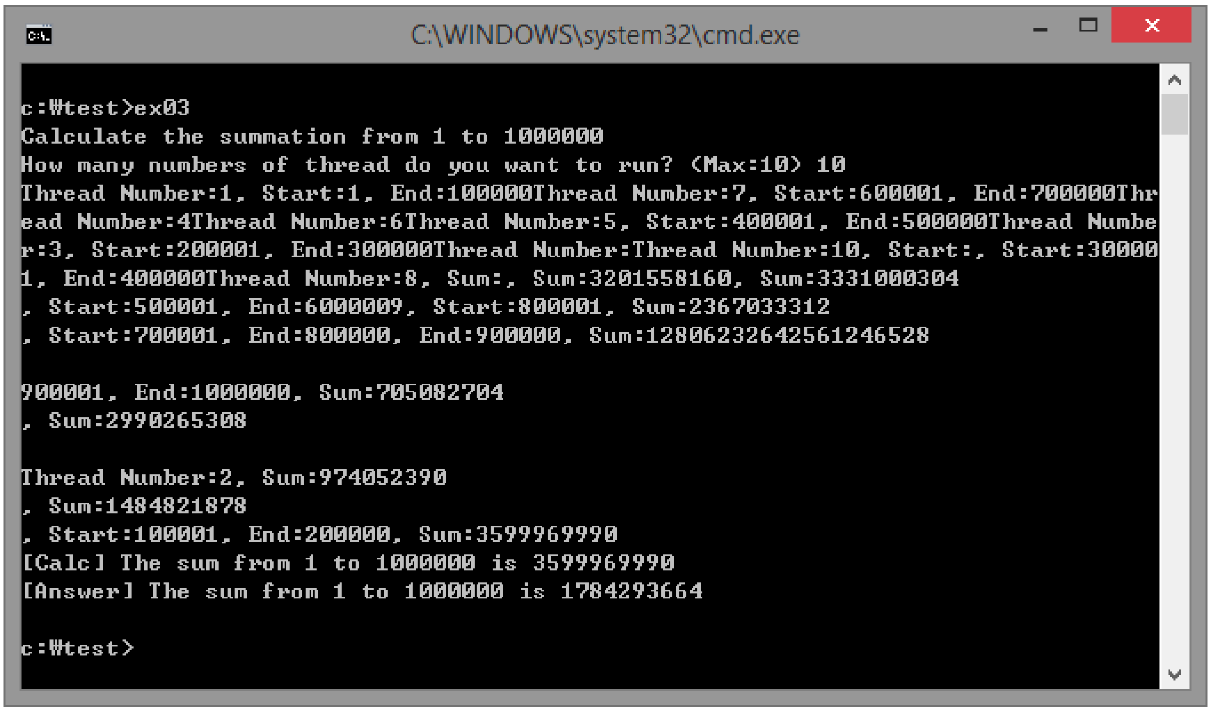
*- Thread synchronization : Multithread summation calculator 2*

Let's update Exercise 2 to speed up the calculation.
Receive the input number of threads before you create the thread and create the threads based on your input number.
Each of the thread will calculate the summation partially and top up the result into the global variable *gSum*.

Here is the code from this basic idea.

```
#include <Windows.h>
#include <iostream>

using namespace std;

struct NumberRange {
    unsigned int Start;
    unsigned int End;
    unsigned int ThreadNumber;
};

unsigned int gNumberTo = 0;
unsigned int gSum = 0;
unsigned int gNumberOfThreads = 1;

DWORD WINAPI sum_numbers( void *arg )
{
  struct NumberRange *pRange;

  pRange = ( struct NumberRange * )arg;
  cout << "Thread Number:" << pRange->ThreadNumber
       << ", Start:" << pRange->Start << ", End:" << pRange->End;
  for( unsigned int i = pRange->Start; i <= pRange->End; i++ )
  {
    gSum += i;
  }
  cout << ", Sum:" << gSum << endl;

  return 0;
}

int main( void )
{
  unsigned int ThreadCount;
  HANDLE hThread[10];

  struct NumberRange Range[10];
  unsigned int Answer = 0;

  cout << "Calculate the summation from 1 to ";
  cin >> gNumberTo;
  cout << "How many numbers of thread do you want to run? (Max:10) ";
  cin >> gNumberOfThreads;

  for( ThreadCount = 0; ThreadCount < gNumberOfThreads; ThreadCount++ )
  {
    Range[ThreadCount].Start = ((gNumberTo / gNumberOfThreads) * ThreadCount) + 1;
    Range[ThreadCount].End = (gNumberTo / gNumberOfThreads) * (ThreadCount + 1);
    if (gNumberTo % gNumberOfThreads && ThreadCount == gNumberOfThreads - 1)
    {
        Range[ThreadCount].End += gNumberTo % gNumberOfThreads;
    }

    Range[ThreadCount].ThreadNumber = ThreadCount + 1;
    hThread[ThreadCount] = CreateThread( NULL, 0, sum_numbers,
                                     (void *)&Range[ThreadCount], 0, NULL );
  }

  WaitForMultipleObjects( ThreadCount, hThread, TRUE, INFINITE );

  cout << "[Calc] The sum from 1 to " << gNumberTo << " is " << gSum << endl;

  for( unsigned int i = 0; i <= gNumberTo; i++ )
  {
    Answer += i;
  }
  cout << "[Answer] The sum from 1 to " << gNumberTo << " is " << Answer << endl;

  return 0;
}
```

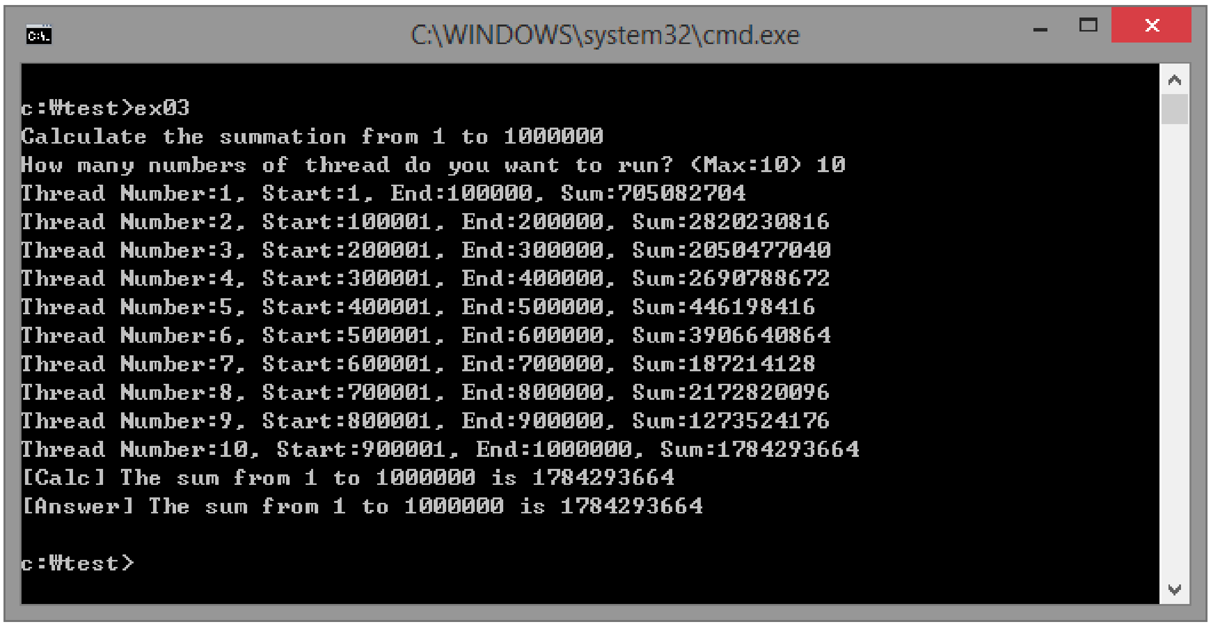
In my computer, it shows the result like below;



```
c:\test>ex03
Calculate the summation from 1 to 1000000
How many numbers of thread do you want to run? (Max:10) 10
Thread Number:1, Start:1, End:100000Thread Number:7, Start:600001, End:700000Thr
ead Number:4Thread Number:6Thread Number:5, Start:400001, End:500000Thread Numbe
r:3, Start:200001, End:300000Thread Number:Thread Number:10, Start:, Start:30000
1, End:400000Thread Number:8, Sum:, Sum:3201558160, Sum:3331000304
, Start:500001, End:6000009, Start:800001, Sum:2367033312
, Start:700001, End:800000, End:900000, Sum:12806232642561246528

900001, End:1000000, Sum:705082704
, Sum:2990265308

Thread Number:2, Sum:974052390
, Sum:1484821878
, Start:100001, End:200000, Sum:3599969990
[Calc] The sum from 1 to 1000000 is 3599969990
[Answer] The sum from 1 to 1000000 is 1784293664

c:\test>
```

The final calculation is 3599969990 and it's different from the actual result (You can see the actual result is 1784293664).

Fix the code and make it work such as below;



```
c:\test>ex03
Calculate the summation from 1 to 1000000
How many numbers of thread do you want to run? (Max:10) 10
Thread Number:1, Start:1, End:100000, Sum:705082704
Thread Number:2, Start:100001, End:200000, Sum:2820230816
Thread Number:3, Start:200001, End:300000, Sum:2050477040
Thread Number:4, Start:300001, End:400000, Sum:2690788672
Thread Number:5, Start:400001, End:500000, Sum:446198416
Thread Number:6, Start:500001, End:600000, Sum:3906640864
Thread Number:7, Start:600001, End:700000, Sum:187214128
Thread Number:8, Start:700001, End:800000, Sum:2172820096
Thread Number:9, Start:800001, End:900000, Sum:1273524176
Thread Number:10, Start:900001, End:1000000, Sum:1784293664
[Calc] The sum from 1 to 1000000 is 1784293664
[Answer] The sum from 1 to 1000000 is 1784293664

c:\test>
```

## Hint & Think

You need to add up the thread synchronization codes.

## Exercise 5
*- Deadlock*

A deadlock occurs when the waiting process is still holding on to another resource that the first needs before it can finish.

So, an example:

Resource A and resource B are used by process X and process Y

1. X starts to use A.
2. X and Y try to start using B
3. Y 'wins' and gets B first
4. now Y needs to use A
5. A is locked by X, which is waiting for Y

Let me explain a real world (not actually real) example for a deadlock situation from the crime movies. Imagine a criminal holds an hostage and against that, a cop also holds an hostage who is a friend of the criminal. In this case, criminal is not going to let the hostage go if cop won't let his friend to let go. Also the cop is not going to let the friend of criminal let go, unless the criminal releases the hostage. This is an endless untrustworthy situation, because both sides are insisting the first step from each other.

Who will act first? No one because each of them waits for the other to act.
-You release the hostage, or I wont release your friend!
-I won't release the hostage unless you release my friend!
- Help!...
-Take it easy man...
COP
CRIMINAL'S FRIEND (HOSTAGE OF COP)
CRIMINAL
HOSTAGE OF CRIMINAL

|  |  |
|---|---|
| COP: | Thread #1 demands Resource #2 but Criminal owns the LOCK |
| CRIMINAL: | Thread #2 demands Resource #1 but Cop owns the LOCK |

|  |  |
|---|---|
| CRIMINALS FRIEND: | Resource #2, the owner of the LOCK is Cop |
| HOSTAGE OF CRIMINAL: | Resource #1, the owner of the LOCK is CRIMINAL |

**Another High Level Explanation of Deadlock : Broken Hearts**

You are dating with a girl and one day after an argument, both sides are heart-broken to each other and waiting for an I-am-sorry-and-I-missed-you call. In this situation, both sides want to communicate each other if and only if one of them receives an I-am-sorry call from the other. Because that neither of each is going to start communication and waiting in a passive state, both will wait for the other to start communication which ends up in a deadlock situation.

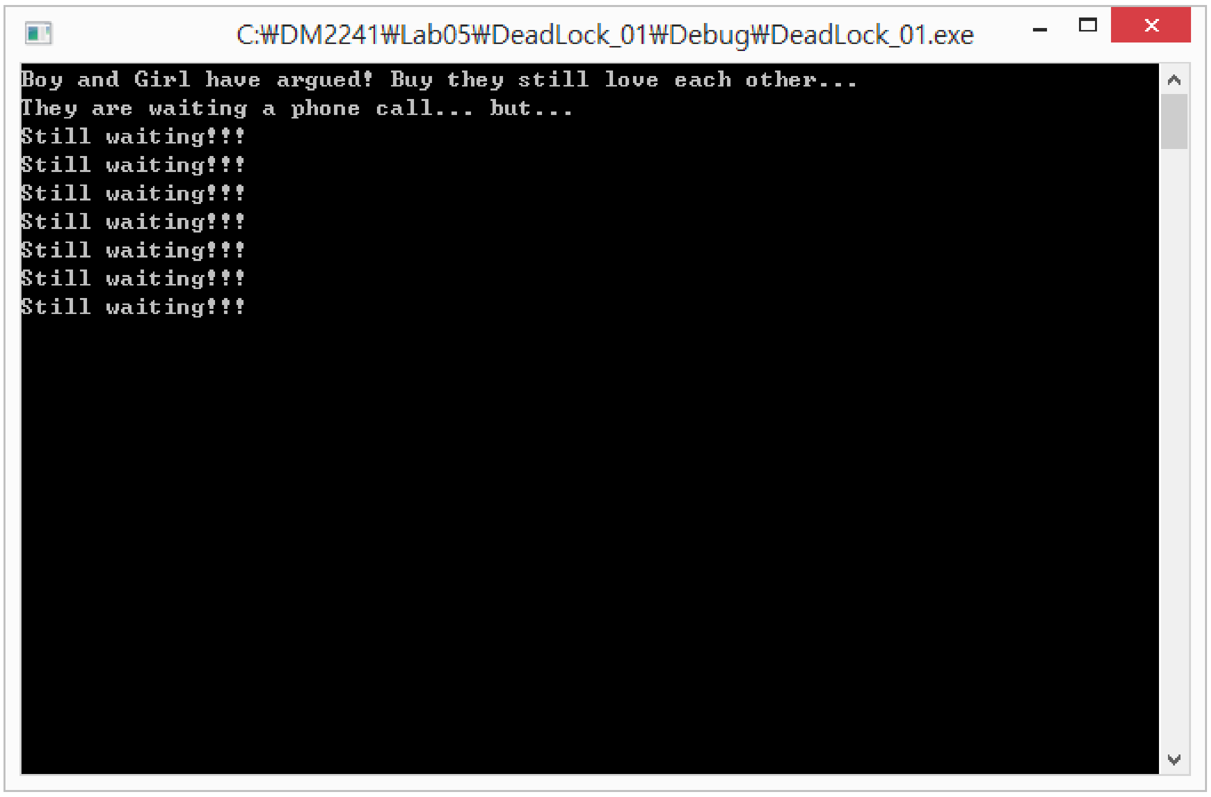
So simply, when two threads needs two different resources and each of them has the lock of the resource that the other need, it is a deadlock.

The best way to avoid deadlocks is to avoid having processes crossover in this way. Reduce the need to lock anything as much as you can.

Try the code below;

```
#include <windows.h>
#include <iostream>

using namespace std;

CRITICAL_SECTION g_CS_Phonecall;

DWORD WINAPI thread_func_boy( void *arg )
{
    EnterCriticalSection( &g_CS_Phonecall );

    cout << "Boy: Let’s make a phone call to her!" << endl;

    return 0;
}

DWORD WINAPI thread_func_girl( void *arg )
{
    EnterCriticalSection( &g_CS_Phonecall );

    cout << "Girl: I better call to him now!" << endl;

    return 0;
}

int main(void)
{
    HANDLE hThread[2];

    InitializeCriticalSection( &g_CS_Phonecall );

    cout << "Boy and Girl have argued! Buy they still love each other..." << endl;
    cout << "They are waiting a phone call... but..." << endl;

    EnterCriticalSection( &g_CS_Phonecall );
    hThread[0] = CreateThread( NULL, 0, thread_func_boy, NULL, 0, NULL );
    hThread[1] = CreateThread( NULL, 0, thread_func_girl, NULL, 0, NULL );

    while( WAIT_TIMEOUT == WaitForMultipleObjects( 2, hThread, TRUE, 1000 ) )
    {
        cout << "Still waiting!!!" << endl;
    }

    DeleteCriticalSection( &g_CS_Phonecall );

    return 0;
}
```

You can see the both of thread are not working anyway forever, just like below!



```
Boy and Girl have argued! Buy they still love each other...
They are waiting a phone call... but...
Still waiting!!!
Still waiting!!!
Still waiting!!!
Still waiting!!!
Still waiting!!!
Still waiting!!!
Still waiting!!!
```

Please make them to call each other!

# Extra Exercises

You don't need to submit these exercises.

## Extra Exercise 1

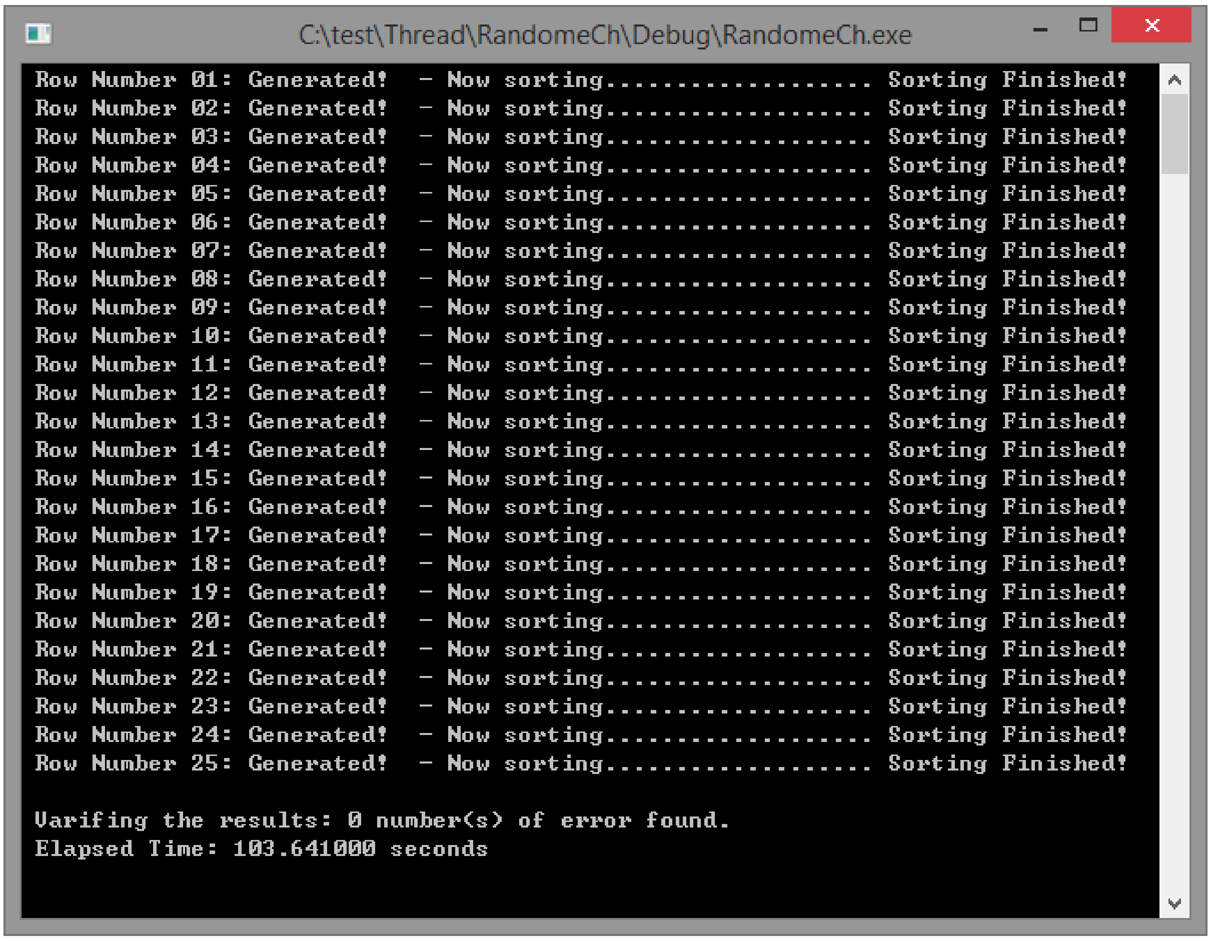
*- Sorting with multithread*

Sorting algorithms are mostly heavy and slow procedures.
Let's try to parallelize the sorting to reduce the processing time.

Here is the example code of generating the random texts in character buffers and sort all the letters in alphabet order.

```
#include <Windows.h>
#include <cstdlib>
#include <iostream>
#include <ctime>
#include <stdio.h>

using namespace std;

#define MAX_ROW_COUNT          25
#define MAX_CHACTER_NUMBER     50000
#define PRINT_SORTING_PROGRESS 20

char gRandomCharacters[MAX_ROW_COUNT][MAX_CHACTER_NUMBER] = { { '\0', }, };

HANDLE hStdout;

int write_to_screen_xy( int Pos_x, int Pos_y, char *OutTextBuffer )
{
    COORD PosXY;

    PosXY.X = Pos_x;
    PosXY.Y = Pos_y;
    SetConsoleCursorPosition( hStdout, PosXY );
    cout << OutTextBuffer;
    fflush( stdout );

    return 0;
}

int generate_random_characters( int Row )
{
    int Ch;
    char TextBuffer[BUFSIZ];

    sprintf_s( TextBuffer, "Row Number %02d: ", (Row+1) );
    write_to_screen_xy( 1, Row, TextBuffer );
    for( Ch = 0; Ch < MAX_CHACTER_NUMBER; Ch++ )
    {
        gRandomCharacters[Row][Ch] = (char)( 'A' + ( rand( ) % 26 ) );
    }
    write_to_screen_xy( (int)(strlen(TextBuffer)+1), Row, "Generated!");

    return 0;
}

int sort_characters( int Row )
{
    int First;
    int Second;
    char TempCh;

    int ProgressCount = 0;
    int ProgressPrint = 0;

    write_to_screen_xy( 27, Row, " - Now sorting" );
    for( First = 0; First < ( MAX_CHACTER_NUMBER - 1 ); First++ )
    {
        for( Second = ( First + 1 ); Second < MAX_CHACTER_NUMBER; Second++ )
        {
            if( gRandomCharacters[Row][First] > gRandomCharacters[Row][Second] )
            {
                TempCh = gRandomCharacters[Row][First];
                gRandomCharacters[Row][First] = gRandomCharacters[Row][Second];
                gRandomCharacters[Row][Second] = TempCh;
            }
        }

        if( ProgressCount > ( MAX_CHACTER_NUMBER / PRINT_SORTING_PROGRESS ) )
        {
            write_to_screen_xy( ( 41 + ProgressPrint ), Row, "." );
            ProgressPrint++;
            ProgressCount = 0;
        }
        else
        {
            ProgressCount++;
        }
    }

    write_to_screen_xy( (41 + PRINT_SORTING_PROGRESS), Row, "Sorting Finished!" );

    return 0;
}

int check_errors( void )
{
    int Error = 0;
    int Row;
    int Ch;

    for( Row = 0; Row < MAX_ROW_COUNT; Row++ )
    {
        for( Ch = 0; Ch < ( MAX_CHACTER_NUMBER - 1 ); Ch++ )
        {
            if( ( '\0' == gRandomCharacters[Row][Ch] ) ||
                ( gRandomCharacters[Row][Ch] > gRandomCharacters[Row][Ch] ) )
            {
                Error++;
            }
        }
    }

    return Error;
}

int main( void )
{
    int RowCount;
    char TextBuffer[BUFSIZ];

    clock_t BeforeClock, AfterClock;
    double  ElapsedSec;

    srand( (unsigned int)time( 0 ) );

    // Get handles to STDOUT.
    hStdout = GetStdHandle( STD_OUTPUT_HANDLE );
    if( hStdout == INVALID_HANDLE_VALUE )
    {
        cout << "Fail to get standard output handle!" << endl;
        return 1;
    }

    for( RowCount = 0; RowCount < MAX_ROW_COUNT; RowCount++ )
    {
        generate_random_characters( RowCount );
    }

    BeforeClock = clock( );
    for( RowCount = 0; RowCount < MAX_ROW_COUNT; RowCount++ )
    {
        sort_characters( RowCount );
    }
    AfterClock = clock( );

    write_to_screen_xy( 1, ( MAX_ROW_COUNT + 1 ), "Varifing the results: " );
    sprintf_s( TextBuffer, "%d number(s) of error found.", check_errors() );
    write_to_screen_xy( 23, ( MAX_ROW_COUNT + 1 ), TextBuffer );

    ElapsedSec = (double)( AfterClock - BeforeClock ) / CLOCKS_PER_SEC;
    sprintf_s( TextBuffer, "Elapsed Time: %f seconds\n", ElapsedSec );
    write_to_screen_xy( 1, ( MAX_ROW_COUNT + 2 ), TextBuffer );

    getchar( );
    return 0;
}
```

Here is the result screen.



```
Row Number 01: Generated!  - Now sorting................... Sorting Finished!
Row Number 02: Generated!  - Now sorting................... Sorting Finished!
Row Number 03: Generated!  - Now sorting................... Sorting Finished!
Row Number 04: Generated!  - Now sorting................... Sorting Finished!
Row Number 05: Generated!  - Now sorting................... Sorting Finished!
Row Number 06: Generated!  - Now sorting................... Sorting Finished!
Row Number 07: Generated!  - Now sorting................... Sorting Finished!
Row Number 08: Generated!  - Now sorting................... Sorting Finished!
Row Number 09: Generated!  - Now sorting................... Sorting Finished!
Row Number 10: Generated!  - Now sorting................... Sorting Finished!
Row Number 11: Generated!  - Now sorting................... Sorting Finished!
Row Number 12: Generated!  - Now sorting................... Sorting Finished!
Row Number 13: Generated!  - Now sorting................... Sorting Finished!
Row Number 14: Generated!  - Now sorting................... Sorting Finished!
Row Number 15: Generated!  - Now sorting................... Sorting Finished!
Row Number 16: Generated!  - Now sorting................... Sorting Finished!
Row Number 17: Generated!  - Now sorting................... Sorting Finished!
Row Number 18: Generated!  - Now sorting................... Sorting Finished!
Row Number 19: Generated!  - Now sorting................... Sorting Finished!
Row Number 20: Generated!  - Now sorting................... Sorting Finished!
Row Number 21: Generated!  - Now sorting................... Sorting Finished!
Row Number 22: Generated!  - Now sorting................... Sorting Finished!
Row Number 23: Generated!  - Now sorting................... Sorting Finished!
Row Number 24: Generated!  - Now sorting................... Sorting Finished!
Row Number 25: Generated!  - Now sorting................... Sorting Finished!

Varifing the results: 0 number(s) of error found.
Elapsed Time: 103.641000 seconds
```

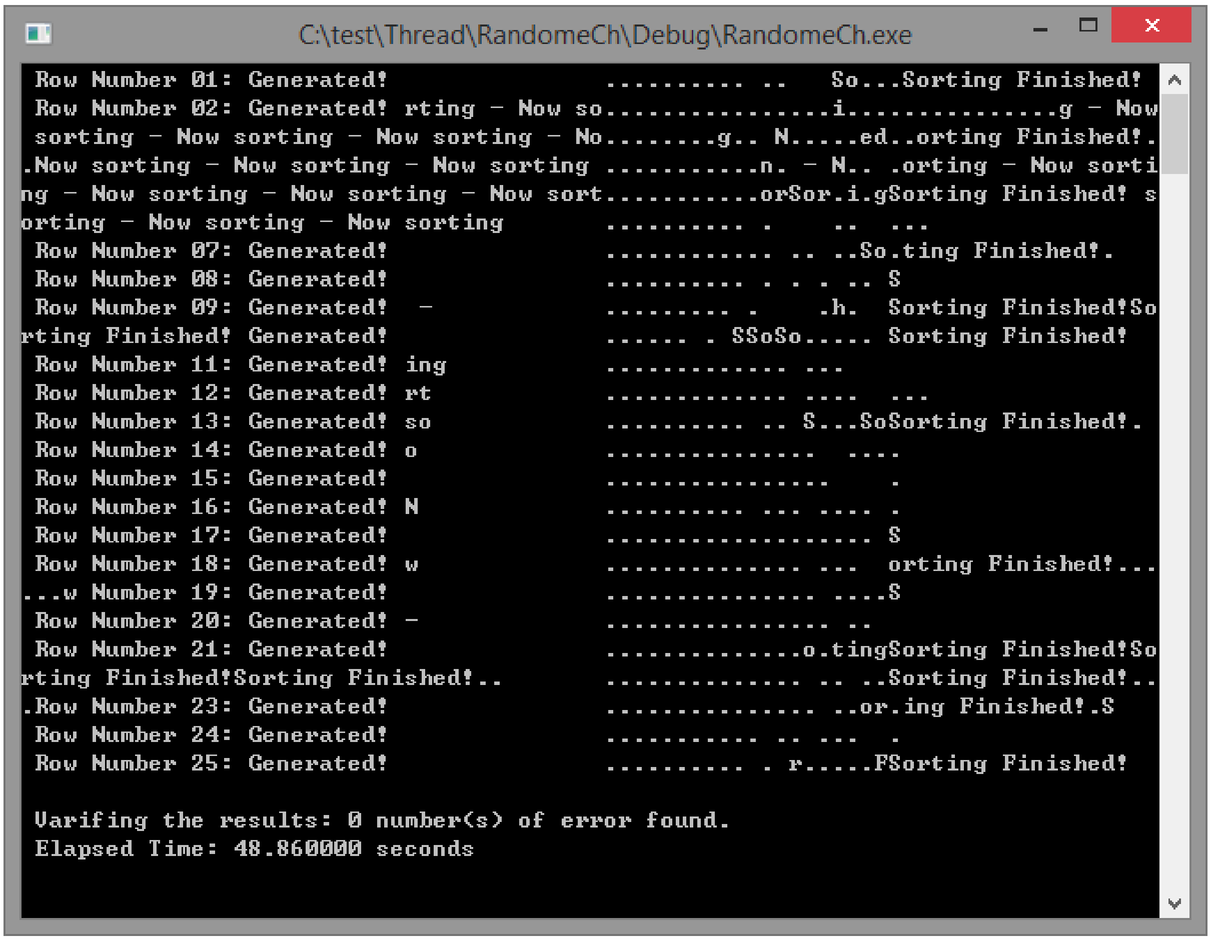
You can see it takes more than 100 seconds! (It will be different from your computer)
If we use multi-threading for each rows, we can reduce the total processing time.

The lines where we need to change are;

```
    for( RowCount = 0; RowCount < MAX_ROW_COUNT; RowCount++ )
    {
        sort_characters( RowCount );
    }
```

You can see *sort_characters()* function is calling and running one after one, sequently. Change the code above and create threads to run *sort_characters()* parallely for each row of *gRandomCharacters[MAX_ROW_COUNT][MAX_CHACTER_NUMBER]*.

You will also need thread synchronization for printing out the progress and results. Without thread synchronization of printing out the message, you will see the screen something like below;



```
 Row Number 01: Generated!                ........... ..   So...Sorting Finished!
 Row Number 02: Generated! rting - Now so................i................g - Now
 sorting - Now sorting - Now sorting - No........g.. N.....ed..orting Finished!.
.Now sorting - Now sorting - Now sorting ...........n. - N.. .orting - Now sorti
ng - Now sorting - Now sorting - Now sort...........orSor.i.gSorting Finished! s
orting - Now sorting - Now sorting        .......... .    ..  ...
 Row Number 07: Generated!                ............ .. ..So.ting Finished!.
 Row Number 08: Generated!                .......... . . . .. S
 Row Number 09: Generated!  -             ......... .    .h.  Sorting Finished!So
rting Finished! Generated!                ...... . SSoSo..... Sorting Finished!
 Row Number 11: Generated! ing            ............. ...
 Row Number 12: Generated! rt             ............. ....  ...
 Row Number 13: Generated! so             .......... .. S...SoSorting Finished!.
 Row Number 14: Generated! o              ...............  ....
 Row Number 15: Generated!                ................    .
 Row Number 16: Generated! N              .......... ... .... .
 Row Number 17: Generated!                ................... S
 Row Number 18: Generated! w              .............. ...  orting Finished!...
...w Number 19: Generated!                ............... ....S
 Row Number 20: Generated! -              ................ ..
 Row Number 21: Generated!                ..............o.tingSorting Finished!So
rting Finished!Sorting Finished!..        .............. .. ..Sorting Finished!..
.Row Number 23: Generated!                ............... ..or.ing Finished!.S
 Row Number 24: Generated!                ........... .. ...  .
 Row Number 25: Generated!                .......... . r.....FSorting Finished!

 Varifing the results: 0 number(s) of error found.
 Elapsed Time: 48.860000 seconds
```

This is because of *int write_to_screen_xy( int Pos_x, int Pos_y, char *OutTextBuffer )* function is not safe with multithread.

For example, this function can be run like this;

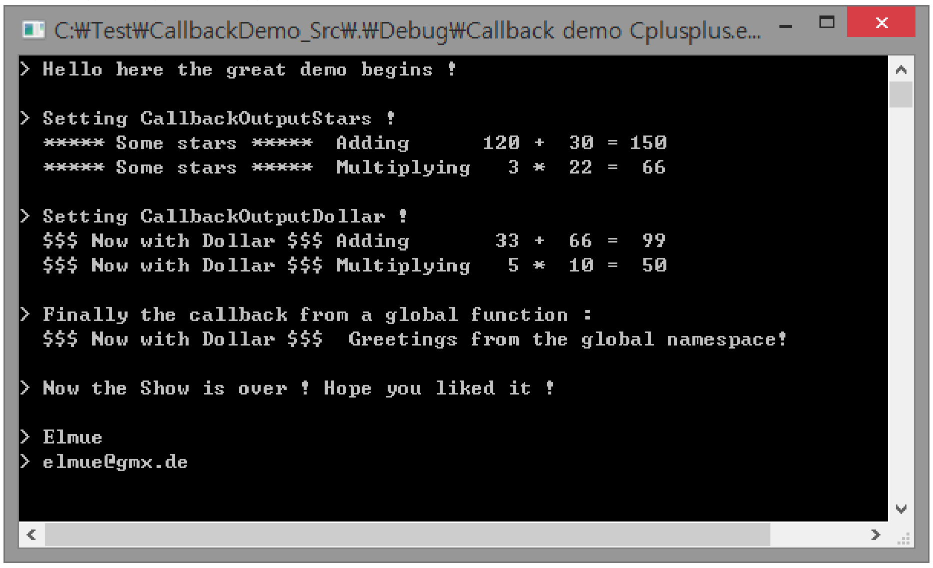
| Thread 1 | Thread 2 | Thread 3 |
|---|---|---|
| `PosXY.X = Pos_x; // = 1` | | |
| | | `PosXY.X = Pos_x; // = 21` |
| | `PosXY.X = Pos_x; // = 44` | |
| | `PosXY.Y = Pos_y; // = 7` | |
| `PosXY.Y = Pos_y; // = 5` | | |
| | | `PosXY.Y = Pos_y; // = 15` |
| `SetConsoleCursorPosition( hStdout, PosXY );` | | |
| | | `SetConsoleCursorPosition( hStdout, PosXY );` |

| | | |
|---|---|---|
| `cout << OutTextBuffer;` | | |
| | `SetConsoleCursorPosition( hStdout, PosXY );` | |
| | | `cout << OutTextBuffer;` |
| | `cout << OutTextBuffer;` | |

For this case, the output from 'Thread 1' will be print on position (21, 15) not at (1, 5) because *`SetConsoleCursorPosition( hStdout, PosXY )`* from 'Thread 3' has executed just before *`cout << OutTextBuffer`* in 'Thread 1'.
So you need thread synchronization in *`int write_to_screen_xy( int Pos_x, int Pos_y, char *OutTextBuffer )`* function.

## Extra Exercise 2
*- Callback function*

Download the codes from the URL (https://goo.gl/rkwHJx) and make it run.
If everything is Ok, you can see the screen such like this.



C:₩Test₩CallbackDemo_Src₩.₩Debug₩Callback demo Cplusplus.e...

```
> Hello here the great demo begins !

> Setting CallbackOutputStars !
  ***** Some stars *****  Adding      120 +  30 = 150
  ***** Some stars *****  Multiplying   3 *  22 =  66

> Setting CallbackOutputDollar !
  $$$ Now with Dollar $$$ Adding       33 +  66 =  99
  $$$ Now with Dollar $$$ Multiplying   5 *  10 =  50

> Finally the callback from a global function :
  $$$ Now with Dollar $$$  Greetings from the global namespace!

> Now the Show is over ! Hope you liked it !

> Elmue
> elmue@gmx.de
```

This exercise you don't have to submit. However, please try to look around how the callback function works.

# More to Read and Check out for Concurrent Programming

More to Read and Check out for Concurrent Programming with multithread programming and new C++ standard, “C++ 11”.

## More to Read 1

*- Thread Handles and Identifiers*

Read the article below :-
https://msdn.microsoft.com/en-us/library/windows/desktop/ms686746(v=vs.85).aspx

This is the microsoft official manual page for the functions of thread handles and identifiers. Find the information of the functions such as DuplicateHandle(), GetCurrentThreadId(), GetCurrentThread(), and OpenThread().

## More to Read 2

*- Suspend and Resume Thread Execution*

We can suspend the processing of created thread and also can resume the processing when we want.
Read the articles below :-
https://msdn.microsoft.com/en-us/library/windows/desktop/ms686342(v=vs.85).aspx
http://www.bogotobogo.com/cplusplus/multithreaded2A.php

This is the part of the controlling the processing of created threads.
Find the information of the functions SuspendThread() and ResumeThread().

## More to Read 3

*- Callback function*

The callback function is a function that is called through a function pointer. If you pass the pointer (address) of a function as an argument to another, when that pointer is used to call the function it points to it is said that a call back is made.

Read the articles:-
[Callback Functions Tutorial] http://goo.gl/HINSMX

[Callback Functions - MSDN] https://goo.gl/RAsLfm
[How to: Implement Callback Functions] https://goo.gl/0iE4bn
[Callback (computer programming)] https://goo.gl/FL57lw
[How to Implement Callbacks in C and C++] http://goo.gl/eOhkCc

## More to Read 4
*- Alertable Wait State and APC (Asynchronous Procedure Call)*

An asynchronous procedure call (APC) is a function that executes asynchronously in the context of a particular thread. When an APC is queued to a thread, the system issues a software interrupt. The next time the thread is scheduled, it will run the APC function.

In an alertable wait operation, the function can return when the specified conditions are met, but it can also return if the system queues an I/O completion routine or an APC for execution by the waiting thread.

Read the articles:-
[Asynchronous Procedure Call] https://goo.gl/YSJW01
[Asynchronous Procedure Call - MSDN] https://goo.gl/sy1fPI

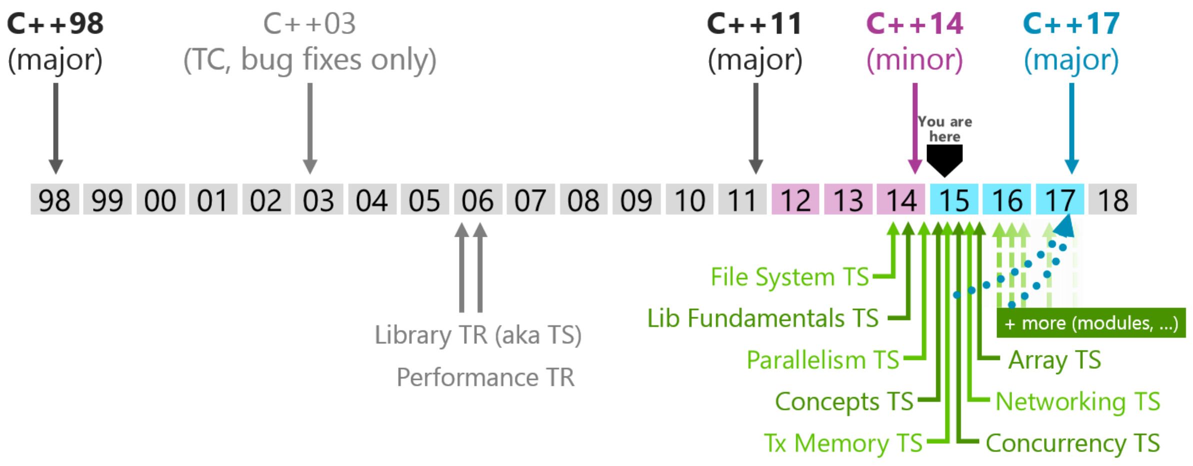
## More to Read 5
*- Videos for concurrent programming with C++ 11 using threads.*

Watch the videos,
https://goo.gl/DXOa3u

There are 10 videos in the link above.
Please try to watch carefully and try the examples on the video by yourself.

## More to Read 6
- C++ status

c++ 11 is the new standard with C++ language.
Now we are using Visual Studio 2013 and this compiler has the most part of C++ 11.
Other compiler such as GCC, cLang, and X-Code also has some implementation of C++ 11 now.
The current status of C++ standard is such as below;



You can read more details of C++ standard status here:-
https://meetingcpp.com/index.php/br/items/cpp-status.html