



Microsoft®
Parallel Computing Platform



Hands-On Lab

Introducing the Visual C++ Concurrency Runtime

Lab version: 1.0.0

Last updated: 3/13/2010



developer & platform **evangelism**

Contents

Introduction.....	4
NUMA Node Groups.....	5
User-Mode Scheduling (UMS)	7
C++ Concurrency Runtime.....	8
Lesson Index	9
Exercise 1: Learning the Basics of the PPL (Background)	10
The Parallel Patterns Library.....	12
A PPL Example.....	13
That's All I Have to Do to Parallelize my Code?	14
Exercise 1: Learning the Basics of the PPL.....	16
Part 1: Using the Parallel Patterns Library	17
Exercise 2: Working with the critical_section class (Background)	21
Data Race - Storage Conflict	22
Motivation	23
Concurrency Runtime Critical Section	24
Exercise 2: Working with the critical_section class.....	25
Part 1: Running the Unsynchronized Code	26
Part 2: Synchronizing with critical_section	28
Exercise 3: Working with the reader_writer_lock (Background)	30
The reader_writer_lock	31
Exercise 3: Working with the reader_writer_lock.....	32
Part 1: Running the Unsynchronized Code	33
Part 2: Synchronization using the reader_writer_lock	35
Exercise 4: Working with the Concurrency Runtime Events (Background).....	37
User Mode Scheduling.....	38
User Mode Scheduling (cont'd)	40
Exercise 4: Working with the Concurrency Runtime Events	42
Part 1: Scheduling the Work	43
Part 2: Running the Windows Event	45
Part 3: Running the Cooperative Event.....	46

Exercise 5: Working with Agents (Background)	47
The Agent Class	48
Messaging Blocks	49
Unbounded Buffer	50
Overwrite Buffer	51
Exercise 5: Working with Agents	52
Part 1: Create the Buffers.....	53
Part 2: Create and Start the Agents	54
Part 3: Implement agent1	55
Part 4: Implement agent2	56
Part 5: Test the Program	57
Lab Summary	58

Introduction

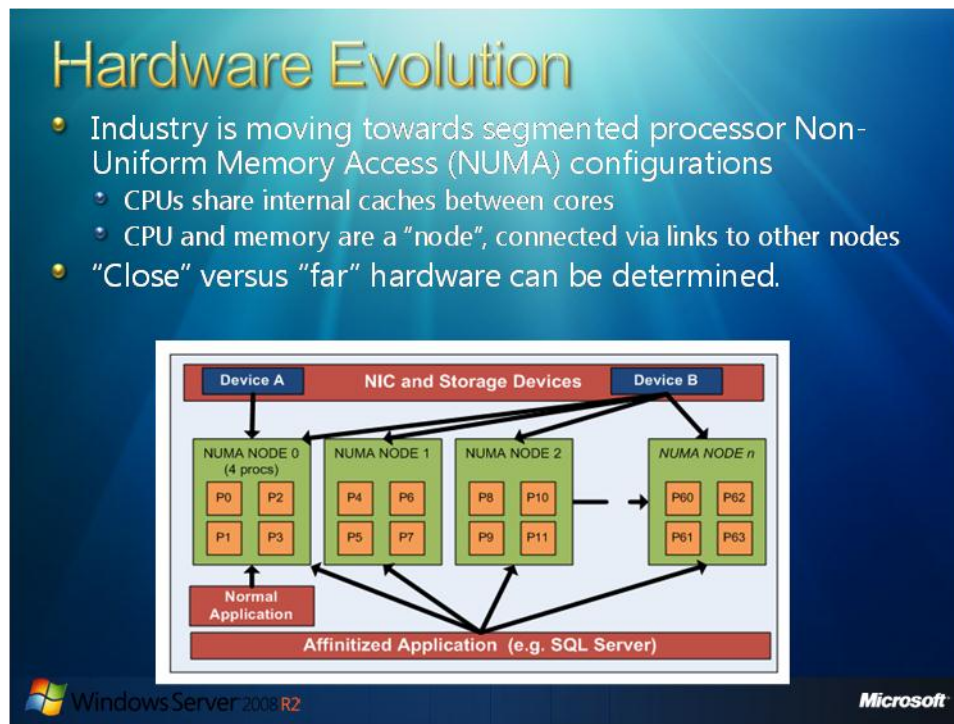


Figure 1

The traditional model for multiprocessor support is Symmetric Multi-Processor (SMP). In this model, each processor has equal access to memory and I/O. As more processors are added, the processor bus becomes a limitation for system performance.

System designers are now using Non-Uniform Memory Access (NUMA) computer architectures to achieve effective increases in processor speed without increasing the load on the processor bus as with SMP architectures. The architecture is non-uniform because each processor is close to some parts of memory and farther from other parts of memory. The processor quickly gains access to the memory it is close to, while it can take longer to gain access to memory that is farther away.

In a NUMA system, CPUs are arranged in smaller systems called nodes. Each node has its own processors and memory, and is connected to the larger system through a cache-coherent inter-connecting bus.

The operating system attempts to improve performance by scheduling threads on processors that are in the same node as the memory being used. It attempts to satisfy memory-allocation requests from within the node, but will allocate memory from other nodes if necessary. Developer may use system API's to discover the topology of the system. You can improve the performance of your applications by using the NUMA functions to optimize scheduling and memory usage.

NUMA Node Groups

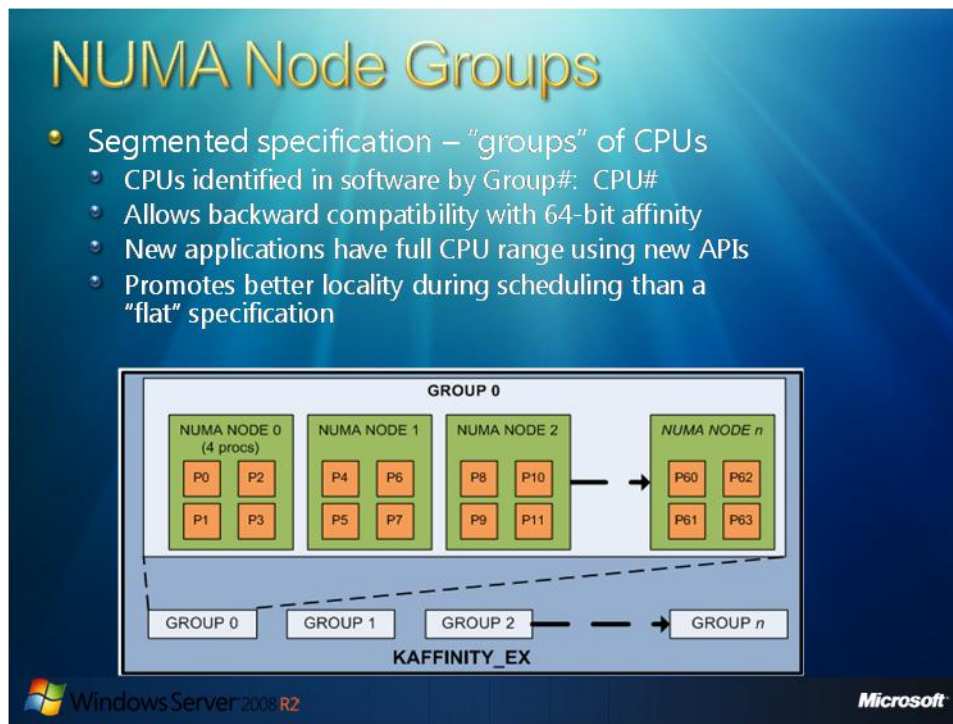


Figure 2

Systems with multiple processors or systems with processors that have multiple cores furnish the operating system with multiple logical processors. A logical processor is one logical instruction execution engine from the perspective of the operating system, application or driver. In effect, a logical processor is a thread.

Windows Server 2008 R2 (and the x64 version of Windows 7) support systems that have more than 64 logical processors. This support is based on the concept of a processor group. A processor group is a static set of up to 64 logical processors that is treated as a single scheduling entity.

When the system starts, the operating system creates processor groups and assigns logical processors to the groups. Systems with fewer than 64 logical processors always have a single group, Group 0. The operating system minimizes the number of groups in a system. For example, a system with 128 logical processors would have two processor groups of 64 processors each.



Figure 3

The operating system takes physical locality into account when assigning logical processors to groups, for better performance. If possible, all of the logical processors in a core, and all of the cores in a physical processor, are assigned to the same group. Physical processors that are physically close to one another are assigned to the same group. Entire NUMA nodes are assigned to the same group, so that a node is a subset of a group. If multiple nodes are assigned to a single group, the operating system chooses nodes that are physically close to one another.

For a discussion of operating system architecture changes to support more than 64 processors and the modifications needed for applications and kernel-mode drivers to take advantage of them, see the whitepaper **Supporting Systems That Have More Than 64 Processors** at

<http://www.microsoft.com/whdc/system/Sysinternals/MoreThan64proc.mspx>.

Scalable application design requires NUMA awareness from several perspectives. Herb Sutter describes this process as "**Maximize Locality, Minimize Contention**" (<http://www.ddj.com/architect/208200273>). Imagine the processor load required to service interrupts from modern 10 Gb/sec network cards, for example. Ideally, interrupt processing and any Deferred Procedure Calls (DPC) occur local to the network device. NUMA locality may be applied to processes, threads, devices, interrupts, and memory. See the Code Gallery entry at <http://code.msdn.microsoft.com/64plusLP> for a detailed walk-thru of NUMA system API's.

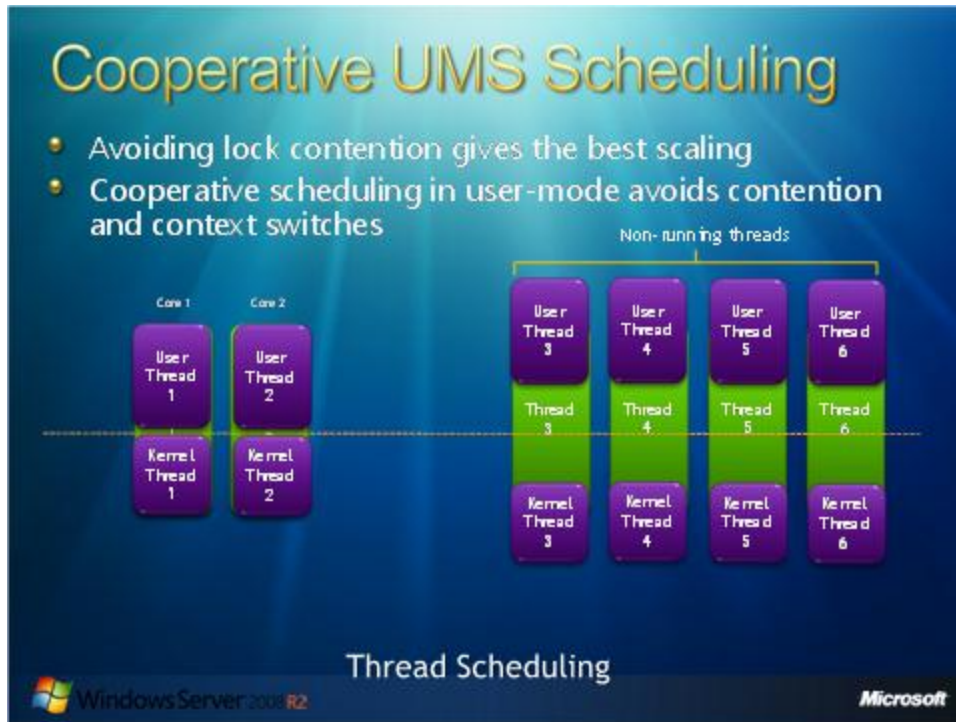


Figure 4

User-Mode Scheduling (UMS)

Windows 7 (x64) and Windows Server 2008 R2 introduce new operating system support for User-Mode Scheduling. User-mode scheduling (UMS) is a light-weight mechanism with system API's that applications can use to schedule their own threads. An application can switch between UMS threads in user mode without involving the system scheduler and regain control of the processor if a UMS thread blocks in the kernel. UMS threads differ from fibers in that each UMS thread has its own thread context instead of sharing the thread context of a single thread. The ability to switch between threads in user mode makes UMS more efficient than thread pools for managing large numbers of short-duration work items that require few system calls.

UMS is recommended for applications with high performance requirements that need to efficiently run many threads concurrently on multi-processor or multi-core systems. To take advantage of UMS, an application must implement a scheduler component that manages the application's UMS threads and determines when they should run. Developers should consider whether their application performance requirements justify the work involved in developing such a component. Applications with moderate performance requirements might be better served by allowing the system scheduler to schedule their threads or to utilize the C++ Concurrency Runtime task scheduler as described within the next section of this document.

UMS is available starting with 64-bit versions of Windows 7 and Windows Server 2008 R2. This feature is not available on 32-bit versions of Windows. Learn more about UMS API's at the MSDN Library online ([http://msdn.microsoft.com/en-us/library/dd627187\(VS.85\).aspx](http://msdn.microsoft.com/en-us/library/dd627187(VS.85).aspx)).

C++ Concurrency Runtime

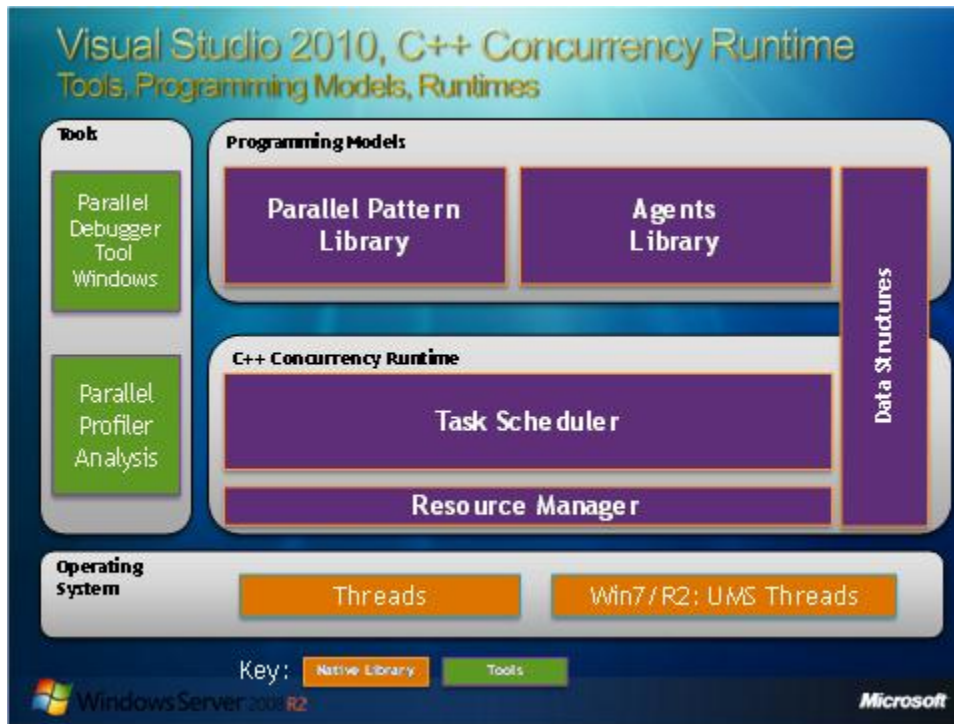


Figure 5

Microsoft recommends that Developers designing applications for multi-core systems with NUMA computer architectures consider using the C++ Concurrency Runtime. This is especially the case for applications within the category of parallel-computing and shared memory computing. *The Concurrency Runtime provides the benefits of concurrent task scheduling without YOU having to build a custom scheduler that is appropriately reentrant, thread-safe, and non-blocking.*

The Concurrency Runtime contains a **Task Scheduler** and a **Resource Manager** component that integrates with the underlying Windows operating system. The Resource Manager manages access to system resources like the collection of CPU's.

On legacy platforms (Vista, XP, Server 2008) the Concurrency Runtime leverages system *thread pools for task scheduling*.

But, on Windows 7 and Windows Sever 2008 R2, the Concurrency Runtime uses the new high-performance **User-Mode-Scheduling (UMS)** operating system capability described within the previous section.

UMS is a light-weight mechanism that an application or library can use to schedule its own threads. An application can switch between threads in user-mode without involving the operating system thread scheduler and thus maintain control of the processor. A UMS based Scheduler never blocks on worker threads making kernel transitions.

The Parallel Platform also supports multiple **Programming Models**. The **Parallel Pattern Library (PPL)** includes mechanisms that provide an easy and convenient way to express *fine-grain parallelism* within your applications. The PPL provides patterns for *Task Execution, Synchronization, and Data Sharing*.

The **Agents Library** (or Asynchronous Agents Library) enables parallel design based upon *workload partitioning, isolation, and data-sharing via message passing*. Agents enable you to express parallelism with *broad granularity*. You may combine both Agents and PPL constructs within the same solution.

The Concurrency Runtime also includes **Data Structures** that are “scheduler aware” enabling you to optimally specify task scheduling requests and custom scheduler policies.

Visual Studio 2010 also includes complementary new tools for parallel application development and testing. These include a new parallel debugger and a new parallel application profiler.

Lesson Index



Figure 6

Lesson 1: Using the Parallel Patterns Library

The Microsoft C++ Concurrency Runtime (ConcRT) includes the Parallel Patterns Library (PPL) to assist you in writing fine-grained parallel programs. This lesson will help you become acquainted several aspects of the PPL. In case you are not familiar with new C++0x language features, this lesson will also guide you through some of the basics.

Lesson 2: Protecting Shared Data using critical_section

When working with code that shares data, the programmer must be very careful to avoid concurrency pitfalls such as data races. Lesson 2 will show you how to synchronize data access using one of the synchronization primitives in the PPL.

Lesson 3: Protecting Shared Data using the reader_writer_lock

Critical Sections may be a great way to synchronize access to shared data that is prone to concurrent access. In case you have a scenario where many threads read the shared data but not too many threads write to shared data, then maybe it's a good idea to use the reader_writer_lock instead.

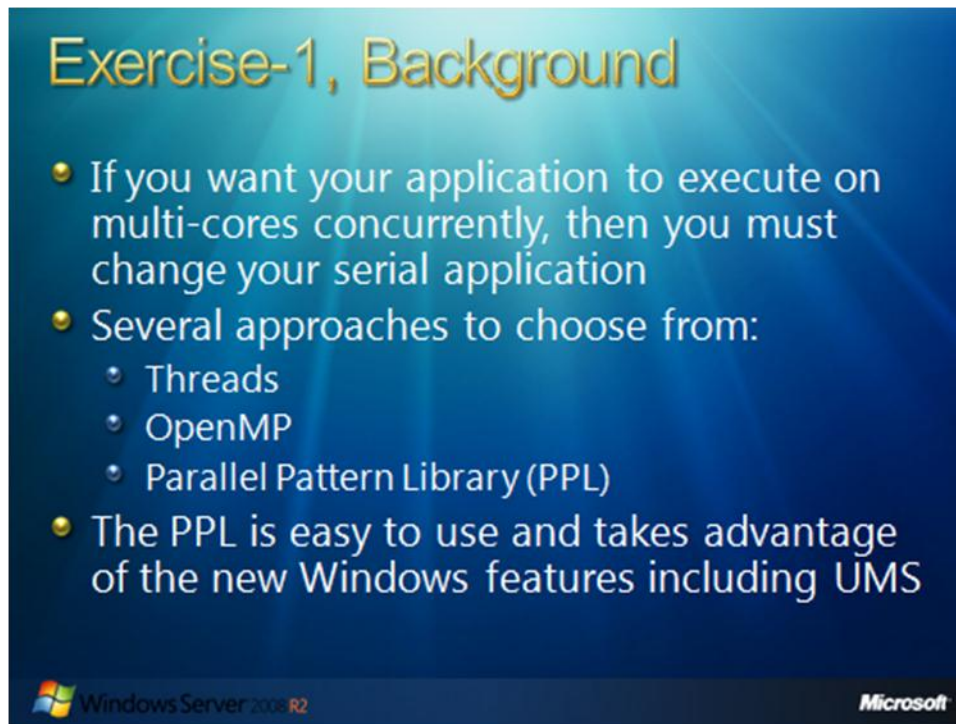
Lesson 4: Working with the Concurrency Runtime Events

Concurrency Runtime events take advantage of UMS (User Mode Scheduling) and are cooperative when it comes yielding for other scheduled tasks. This lesson will show you how to work with Concurrency Runtime events and task groups and how they differ from the regular Win32 events.

Lesson 5: Working with Agents

If your program has an intricate flow and may be parallelized using components that must communicate with one another, then agents might be your best choice. This lesson will teach you how to work with the asynchronous agents library, a data flow message passing library.

Exercise 1: Learning the Basics of the PPL (Background)

**Figure 7**

Developers seeking to execute their native serial applications on multiple cores must find a way to run them in parallel. One choice is to redesign the application using threads, which can be time consuming, prone to errors, and difficult to debug.

Another option is to use the Pattern Parallel Library to enable parallel execution of loops without having to worry about the underlying mechanism. The latter mechanism is more appealing and should reduce develop and testing time. Nonetheless, this does not mean that your code is immune to the concurrency problems. You must still pay careful attention to your code.

The Parallel Patterns Library

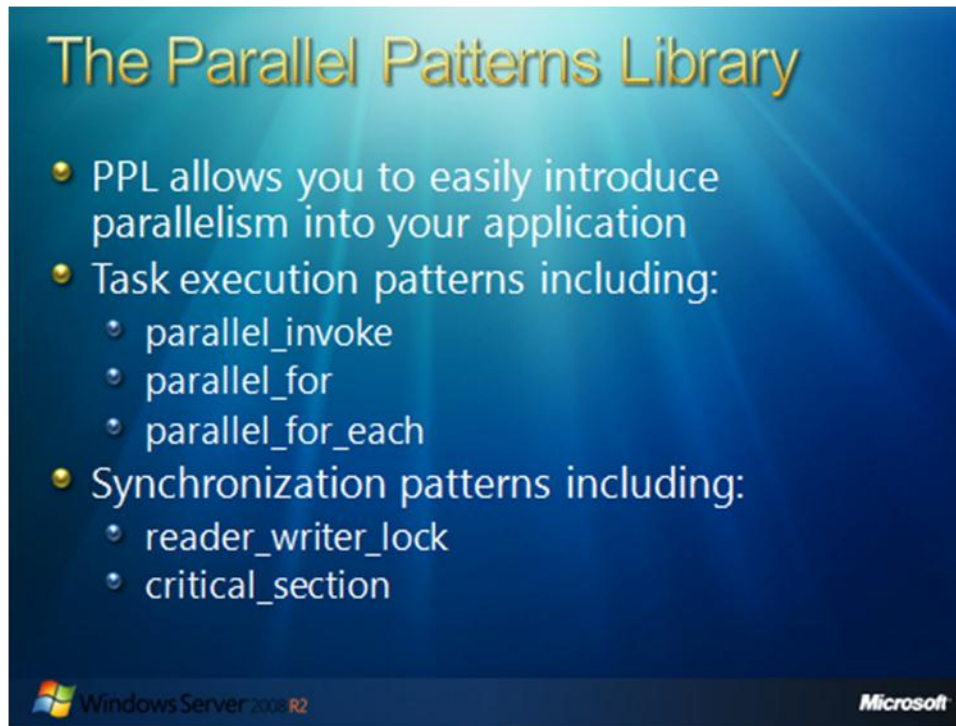


Figure 8

PPL stands for the Parallel Pattern Library and is meant to provide a convenient interface to execute work in parallel. By using PPL, you can introduce parallelism without even having to manage a scheduler. Below are some of the most common patterns.

Task execution patterns:

- `parallel_invoke`: To execute from 2 to 10 tasks in parallel
- `parallel_for`: To execute tasks in parallel over a range of integers
- `parallel_for_each`: To execute tasks in parallel over a collection of items in an STL container

Synchronization patterns:

- `reader_writer_lock`: A cooperative reader-writer lock that yields to other tasks instead of preempting.
- `critical_section`: A cooperative mutual exclusion primitive that yields to other tasks instead of preempting.

Data sharing pattern:

- `combinable`: A scalable object that has a local copy for each thread where processing can be done lock free on the local copy and combined afterwards when parallel processing is done.

A PPL Example

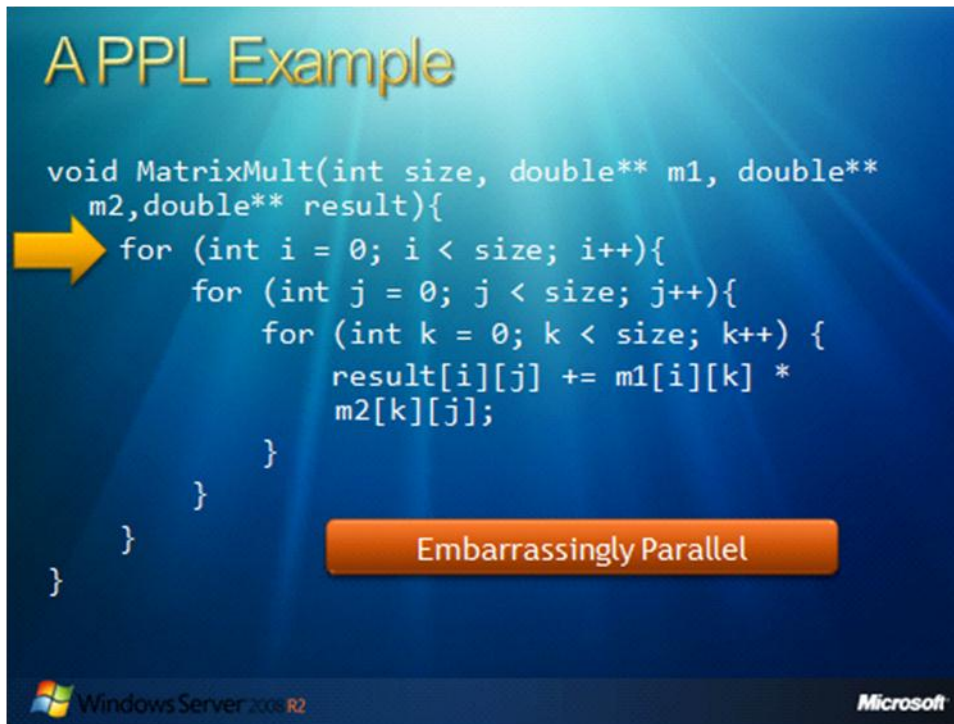


Figure 9

Take a look at the following code that carries matrix multiplication:

C++

```
void MatrixMult(int size, double** m1, double** m2, double** result){
    for (int i = 0; i < size; i++){
        for (int j = 0; j < size; j++){
            for (int k = 0; k < size; k++) {
                result[i][j] += m1[i][k] * m2[k][j];
            }
        }
    }
}
```

The code has nothing out of this world; in fact, probably no one would ever do this for multiplying matrices in real life. But the interesting thing about it is that the outer loop is very embarrassingly parallel.

If we were to use threading APIs to do this, it would require a lot of effort from our part. We would have to deal with partitioning the data, and data synchronization.

Here's that same scenario using the library "parallel_for" from the Parallel Pattern Library.

C++

```
void MatrixMult(int size, double** m1, double** m2, double** result){
    parallel_for (0, size, 1, [&](int i){
```

```
        for (int j = 0; j < size; j++){  
            for (int k = 0; k < size; k++){  
                result[i][j] += m1[i][k] * m2[k][j];  
            }  
        }  
    });  
}
```

One thing you'll notice is that we're using a new feature from C++ Ox called Lambda support. The syntax following the `parallel_for` key-word specifies iteration logic from zero to size in increment steps of 1.

By changing the code to use the `parallel_for`, the Concurrency Runtime Scheduler implicitly executes each iteration as an independent task.

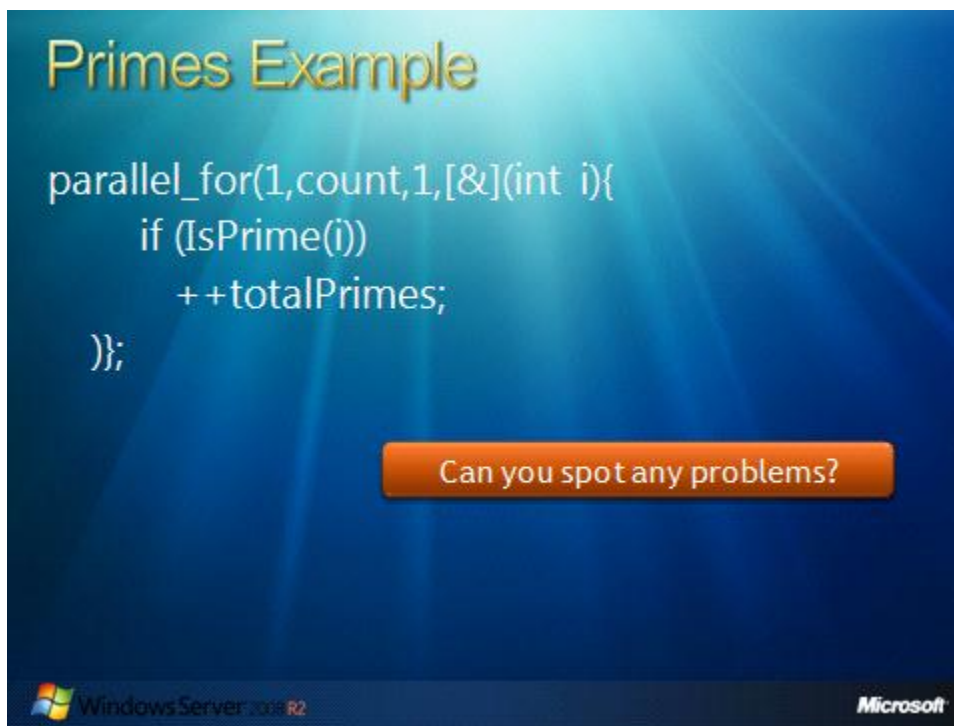


Figure 10

That's All I Have to Do to Parallelize my Code?

We all know that writing code can be a very daunting task. Writing code that executes in parallel adds even more complexity. This means that writing code that will execute in parallel can introduce new problems that did not exist in the serial version of your application.

Take a look at the following code snippet:

```
C++  
for(int i = 1;i < count;++i){
```



```
        if (IsPrime(i))  
++totalPrimes;  
    };
```

The code works perfectly as is. It will find the number of primes by using a function that returns whether the sent parameter is prime or not. The totalPrimes variable is incremented every time such number is found.

In order to parallelize this code using PPL, the code would look like this:

```
C++  
parallel_for(1,count,1,[&](int i){  
    if (IsPrime(i))  
        ++totalPrimes;  
});
```

If we were to run this code on a multi-core machine, we would discover 2 things:

1. The program runs a lot faster than before
2. The program is a lot faster but gives incorrect results with each run

We'll talk about data races later on and you will understand why the parallel code is yielding incorrect results.

Let's get started and parallelize some code with PPL!

Exercise 1: Learning the Basics of the PPL

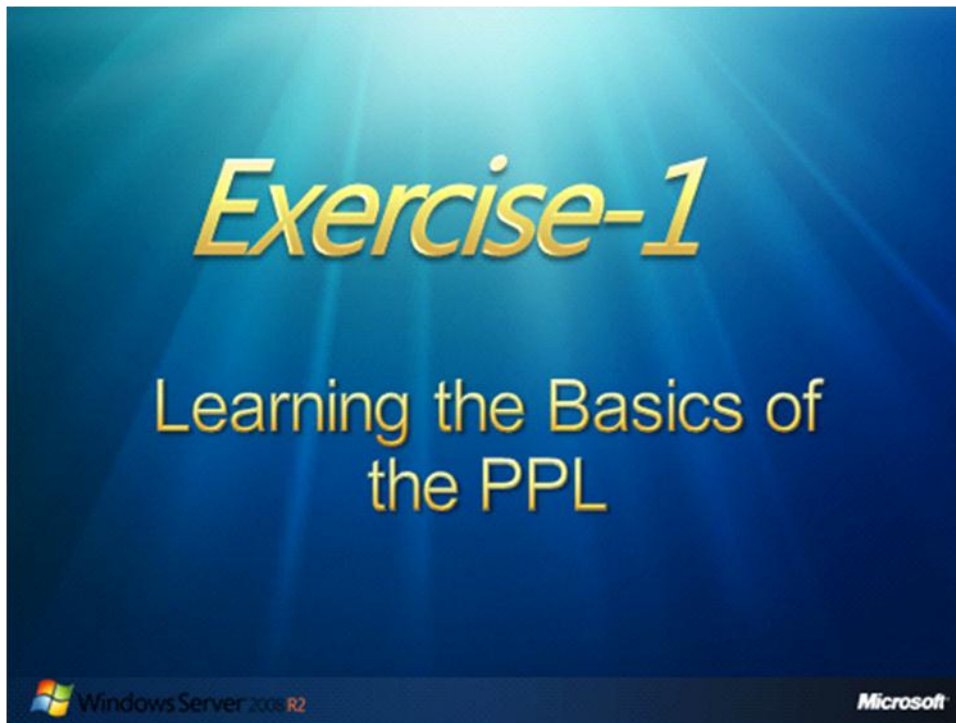


Figure 11

In this exercise you will be working with an application that fills in the values of an array and then calls different methods to carry out operations on the elements of the array. Each array element will be doubled and then will be printed to the screen.

You will learn how to parallelize serial “foreach” and “for” statements.

Part 1: Using the Parallel Patterns Library

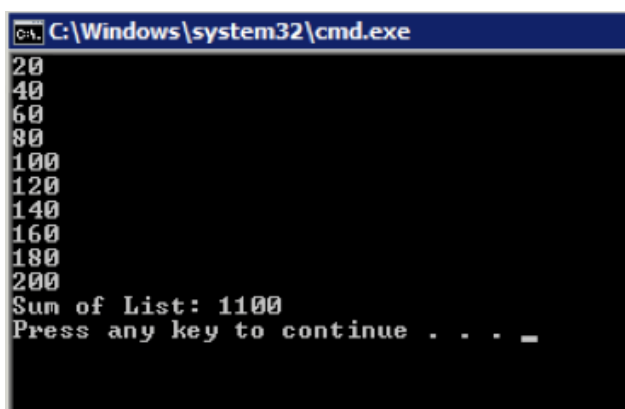
1. Open the **Exercise-1** solution file found at the following folder:

```
C:\Server 2008 R2 Labs\Working with the CRT\Exercise-1\
```

2. From the **Solution Explorer**, double click the **Exercise-1.cpp** file under the **Sources** folder.

Note: Take a minute or two to examine the code. The program calls various functions to fill an array with values and then duplicates all the values in the array. The values of the arrays are printed on screen using a **for_each** statement.

3. From the **Debug** menu, select **Start without Debugging**, you will see the output of the program (click **Yes** if prompted to build the project):



```
C:\Windows\system32\cmd.exe
20
40
60
80
100
120
140
160
180
200
Sum of List: 1100
Press any key to continue . . . _
```

Figure 12

4. Press any key to close the program's window and return to Microsoft Visual Studio 2010.
5. In order to use the **PPL**, include the following header file:

```
C++
```

```
#include <ppl.h>
```

6. In order to include the namespace of the Concurrency namespace, type the following line below the header includes:

```
C++
```

```
using namespace Concurrency;
```

7. Let's parallelize a couple of loops using the PPL. In main, find the **for_each** loop:

```
int _tmain(int argc, _TCHAR* argv[])
{
    array<int, 10> theList = GetTenNumbers();
    theList = DoubleArrayValues(theList);
    for_each(theList.begin(), theList.end(), [&](int n) {
        printf_s("%d\n", n);
    });

    printf("Sum of List: %i\n", SumList(theList));
    return 0;
}
```

Figure 13

8. Change the **for_each** statement to a **parallel_for_each** by replacing the text:

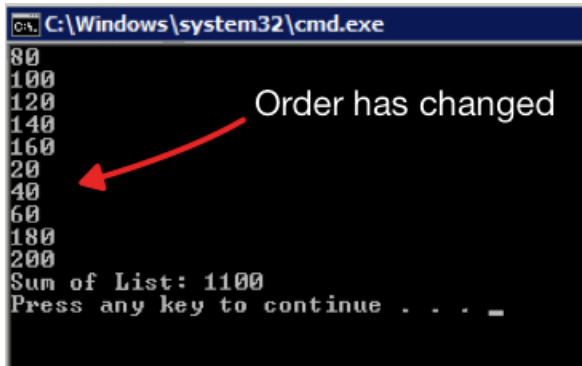
```
parallel_for_each(theList.begin(), theList.end(), [&](int n) {
    printf_s("%d\n", n);
});
```

Figure 14

By specifying the **parallel_for_each**, the loop will now execute in parallel. You do not need to worry on how to scale this up as the Concurrency Runtime scheduler will take care of these details for you.

How do you think the values of the array will now be printed on screen using a **parallel_for_each**? Will anything change?

9. From the **Debug** menu, select **Start without Debugging**, you will see the output of the program (click **Yes** if prompted to build the project):



```
C:\Windows\system32\cmd.exe
80
100
120
140
160
20
40
60
180
200
Sum of List: 1100
Press any key to continue . . . _
```

Figure 15

Note: As you may have guessed it (and if your machine has more than one core/processor), because parallel algorithms such as **parallel_for_each** act concurrently, the parallel version of this example might produce different output than the serial version.

10. Press any key to close the program's window and return to Microsoft Visual Studio 2010.

11. Let's parallelize one of the for-loops in the program. Look for the function called **DoubleArrayValues** (around line 29).

12. **Comment out** the following line of code:

C++

```
for (int i = 0; i < 10; i++) {
```

13. Below the commented line, write down the following:

C++

```
parallel_for(0,10,1,&theList)(int i) {
```

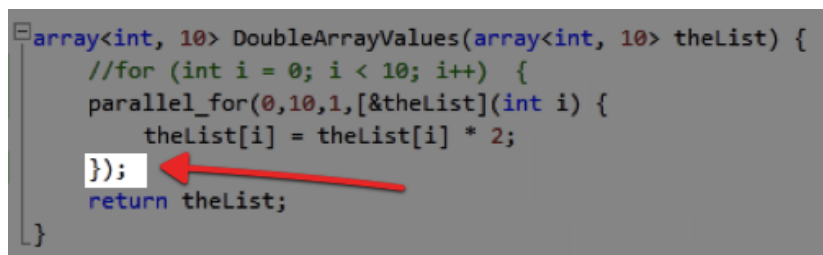
Note: The first parameter corresponds to the starting value of the loop (0), the second one corresponds to the end value of the loop (10), and the third value corresponds to the step value of the loop (1).

The fourth parameter, **[&theList](int i)** is the beginning of a lambda expression. In C++0x, "lambda expressions" implicitly define and construct unnamed function objects, which then behave like handwritten function objects. In this case we are passing the local value **theList** by reference to the function that takes in an **integer** value.

For more information on C++0x lambdas, please visit:

<http://bit.ly/lambda>

14. Since now what used to be the body of our loop is going to behave as the function parameter, we need to **close the parenthesis** right after the closing bracket of the old sequential for loop:



```
array<int, 10> DoubleArrayValues(array<int, 10> theList) {
    //for (int i = 0; i < 10; i++) {
    parallel_for(0,10,1,&theList)(int i) {
        theList[i] = theList[i] * 2;
    });
    return theList;
}
```

Figure 16

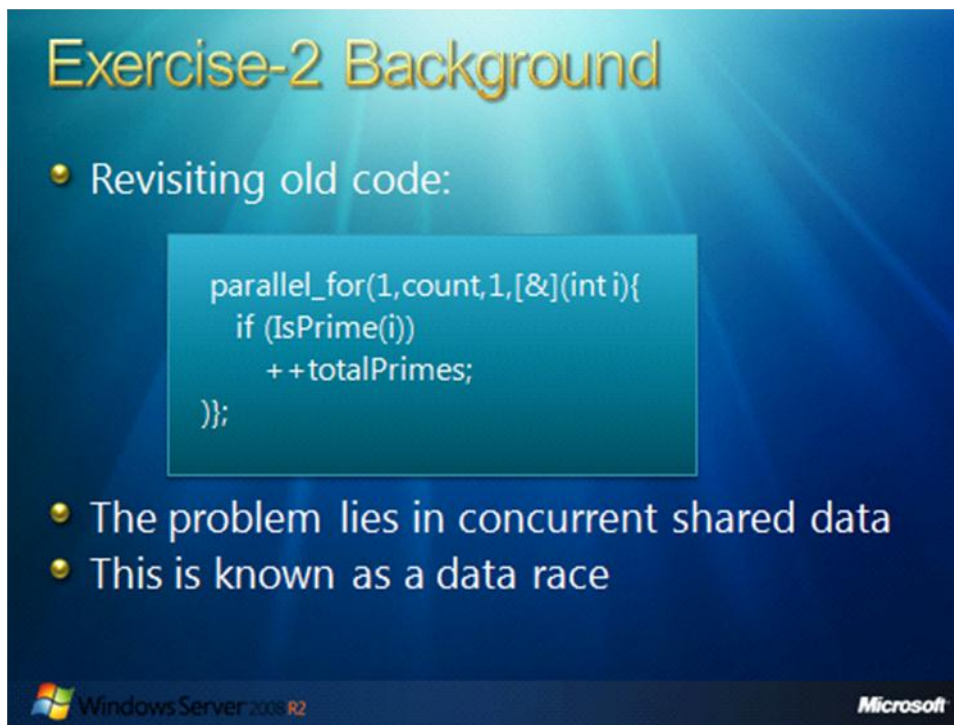
15. Feel free to run the program again, you should get a similar output.

Note: Even though that loop is running in parallel, the program executes too fast to actually notice any performance improvements.

16. Try and parallelize the **for-loop** found in the function **GetTenNumbers**(around line 12).
17. Once you have parallelized the for-loop, from the **Debug** menu, select **Start without Debugging**(click **Yes** if prompted to build the project):.

Note: Did you get the output you were expecting? What do you think is causing the problem?

Exercise 2: Working with the critical_section class (Background)



The slide has a blue gradient background. At the top, the title 'Exercise-2 Background' is written in a yellow, stylized font. Below the title, there is a bullet point with a yellow circle icon, followed by the text 'Revisiting old code:'. Underneath this, a light blue rectangular box contains a C++ code snippet. Below the box, there are two more bullet points with yellow circle icons. At the bottom left of the slide is the Windows Server 2008 R2 logo, and at the bottom right is the Microsoft logo.

Exercise-2 Background

- Revisiting old code:

```
parallel_for(1,count,1,[&](int i){  
    if (IsPrime(i))  
        ++totalPrimes;  
});
```

- The problem lies in concurrent shared data
- This is known as a data race

Windows Server 2008 R2 Microsoft

Figure 17

In the first lesson we saw how not being careful with the data that your program shares can have adverse effects in your application. Revisiting the parallelized version of the primes example:

```
C++  
  
parallel_for(1,count,1,[&](int i){  
    if (IsPrime(i))  
        ++totalPrimes;  
});
```

The problem with this code is that all threads will be accessing and modifying the **totalPrimes** variable. When you have concurrent access to a shared variable, then a thread could read a value; modify that variable's value while another thread reads the old variable's value. At this point, the first thread is going to write the new value to the variable, and later on the second thread will also write the new value to the variable; thus, wiping whatever the first thread wrote. This is a problem commonly known as a **data race** condition.

Data Race - Storage Conflict

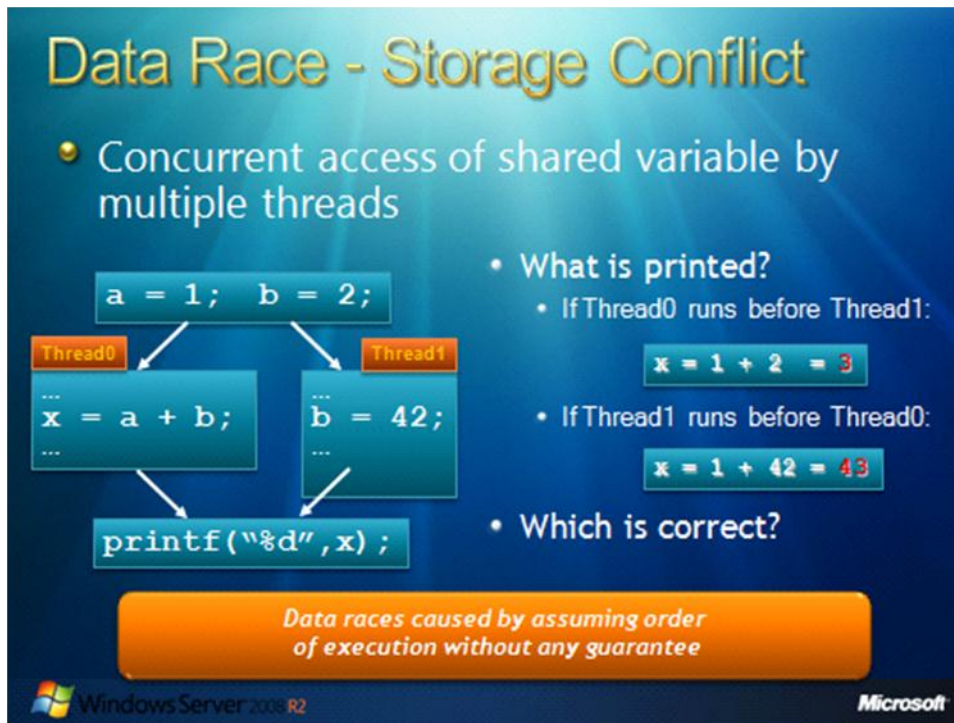


Figure 18

A data race happens the output of a process is not well defined, and it is critically dependent on the timing and sequence of events.

In this particular example, **a** and **b** are assigned a specific value. The application then spawns two threads. One thread modifies the value of the variable, while the other thread reads the value from that same variable. Depending on which thread runs first, the value of the variable “b” will be different, and the final value of “x” will also be different between subsequent runs.

There are many ways to avoid data races, one of which is to use a critical section to make sure that only one thread can modify a particular variable. Although this might solve the problem of a data race, critical sections that make other threads wait for long periods of time can hinder the performance of the application. In a worst case scenario, your multi-threaded application could end up performing worse than its serial version.

Motivation

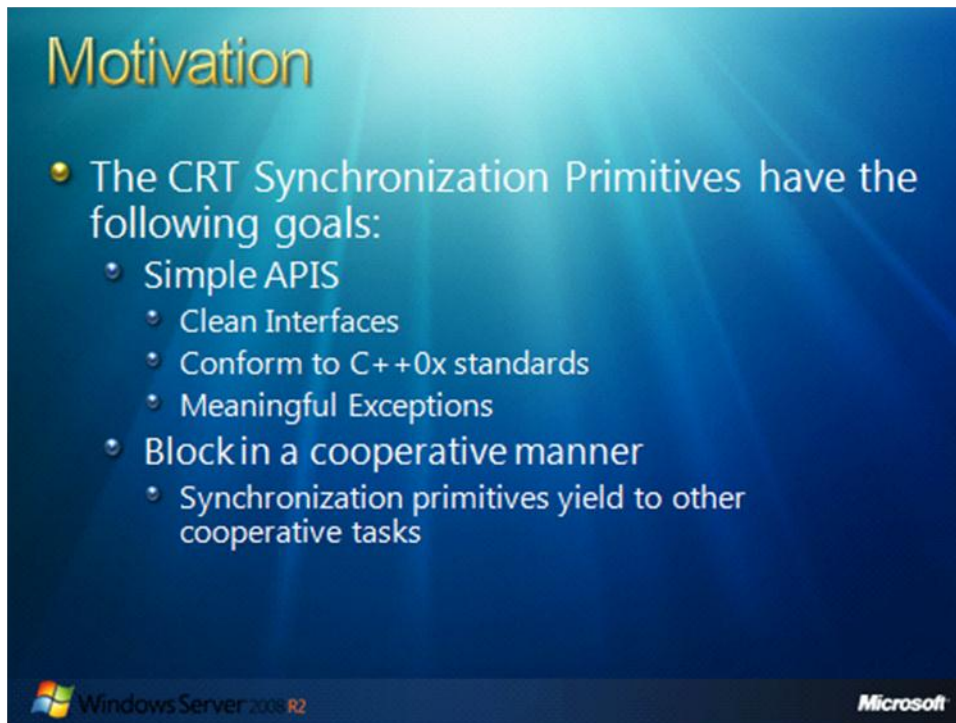


Figure 19

The concurrency runtime's synchronization primitives have the following goals:

- **Simple APIs**

Unlike their Win32 equivalent, concurrency runtime's synchronization primitives don't have C-style initialization and release/destroy types of resource management calls. The exposed interfaces are simple and conform to the C++0x standards and the synchronization objects throw meaningful exceptions on certain illegal operations.

- **Block in a cooperative manner**

The synchronization objects are cooperative in nature, in that they yield to other cooperative tasks in the runtime in addition to preempting.

Concurrency Runtime Critical Section

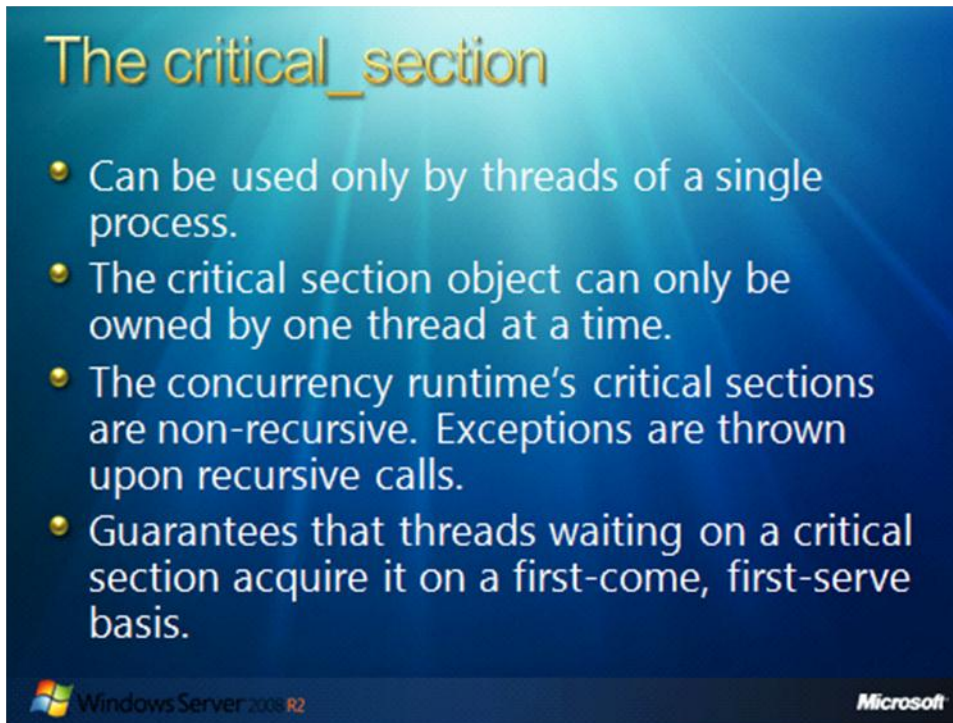


Figure 20

Critical sections represent a non-reentrant, cooperative mutual exclusion object that uses concurrency runtime's facilities to enable cooperative scheduling of work when blocked. This class satisfies all Mutex requirements specified in C++0x standards. The concurrency runtime's critical section provides a C++ façade as compared to its C-styled Win32 equivalent: Windows CRITICAL_SECTION.

Similarity to the Win32 CRITICAL_SECTION:

- Can be used only by threads of a single process.
- The critical section object can only be owned by one thread at a time.

Differences from Win32 CRITICAL_SECTION:

- The concurrency runtime's critical sections are non-recursive. Exceptions are thrown upon recursive calls.
- The concurrency runtime's critical section object guarantees that threads waiting on a critical section acquire it on a first-come, first-serve basis.
- There is no need to explicitly call Initialization/allocation of resources before use of the concurrency runtime's critical section and release resources after the use of the critical section.
- Cannot specify spin count for the concurrency runtime's critical section object.
- The concurrency runtime's critical section enforces cooperative blocking where they yield to other cooperative tasks in the runtime when blocked.
- Exceptions are thrown by the concurrency runtime's critical section object; on unlock calls when the lock is not held, or if a lock is destroyed when being held.

Exercise 2: Working with the critical_section class

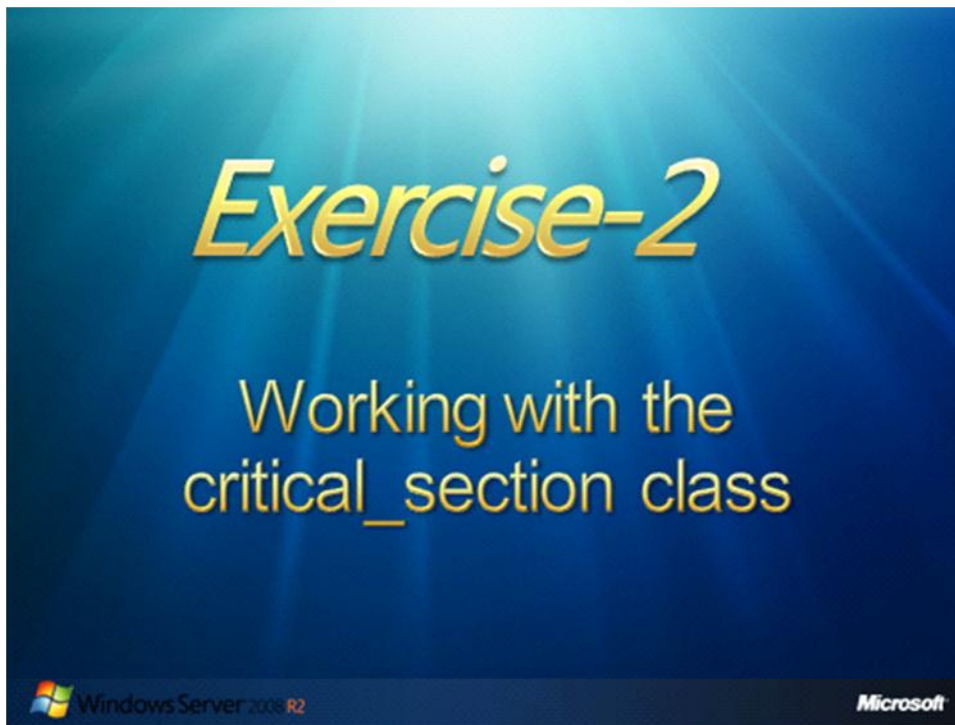


Figure 21

By now, you should be aware that shared variables between can have negative effects on your application. When working with shared data, you can synchronize data access by primarily using:

- Blocking methods such as locks and mutexes
- Non-blocking methods such as lock-free programming techniques.

In this exercise you will learn how to use the critical_section blocking method to make sure that data access is synchronized between threads.

Part 1: Running the Unsynchronized Code

1. Open the **Exercise-2** solution file found at the following folder:

```
C:\Server 2008 R2 Labs\Working with the CRT\Exercise-2\
```

2. From the **Solution Explorer**, double click the **Exercise-2.cpp** file under the **Sources** folder.
3. Scroll down to main and focus on the **parallel_invoke** call (around line 35)

C++

```
parallel_invoke(  
    [&] { FunctionA(); },  
    [&] { FunctionB(); }  
);
```

Note: The **parallel_invoke** algorithm executes a set of tasks in parallel. It does not return until each task has completed. This algorithm is useful when you have several independent, unrelated tasks that you want to execute at the same time.

The **parallel_invoke** algorithm takes as its parameters a series of lambda functions, function objects, or function pointers. The **parallel_invoke** algorithm is overloaded to take from two to ten parameters.

In this case, the **parallel_invoke** call will create a couple of tasks that will execute **FunctionA** and **FunctionB** in parallel.

4. Scroll up and examine the bodies of **FunctionA** and **FunctionB**

Note: You'll notice that both functions are incrementing the global variable **number**.

5. Before we start the program, scroll to the top of the file, you should see the **NUM_ITERATIONS** constant, which is used to specify how many times each function will increment the variable

C++

```
static const int NUM_ITERATIONS = 150000;
```

6. From the **Debug** menu, select **Start without Debugging**(click **Yes** if prompted to build the project):

Note: If you have more than one processor/core in your machine, then most likely the final value of **number** is **not** 300,000. Both threads are sharing the same data (the **number** variable) and they are not synchronizing so that only one of them can read and write to the variable at a time. This is causing a **data race** between the threads, and can have adverse effects in any application with shared data.

7. Run the application a couple more times and note that the final value of **number** is different from the previous run:

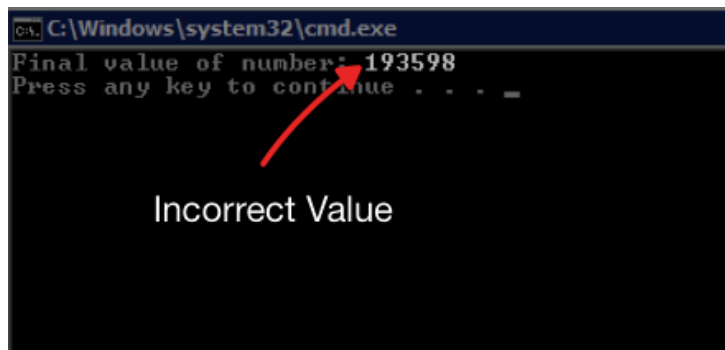


Figure 22

8. Press any key to close the program's window and return to Microsoft Visual Studio 2010.

Part 2: Synchronizing with critical_section

1. Back in Visual Studio, scroll down to the main function.
2. In order to synchronize data access between the threads, declare the following `critical_section` variable as the first line of the main function:

C++

```
critical_section mutex;
```

Note: This `critical_section` type represents a non-reentrant, cooperative mutual exclusion object that uses concurrency runtime's facilities to enable cooperative scheduling of work when blocked.

3. Send the mutex as a parameter to **FunctionA** and **FunctionB** by making the following modifications to the `parallel_invoke` function:

C++

```
parallel_invoke(  
    [&] { FunctionA(&mutex); },  
    [&] { FunctionB(&mutex); }  
);
```

4. Scroll up until you see **FunctionA**.
5. Change **FunctionA's** declaration to specify that it requires a incoming parameter of type `critical_section`:

C++

```
void FunctionA(critical_section* pMutex)
```

6. Inside **FunctionA's** body, lock the mutex so that it will guarantee that this thread will be the only one changing the `numbers` variable by typing the following code **above** the line where `number` gets incremented:

C++

```
pMutex->lock();
```

7. Once the variable is modified, we must release the mutex so that other threads that need to use it will be able to acquire it. Release the mutex by typing the following code **above** the line where `number` gets incremented:

C++

```
pMutex->unlock();
```

8. Repeat steps 5-7 above but this time around with **FunctionB**.
9. From the **Debug** menu, select **Start Without Debugging** (click **Yes** if prompted to build the project)
10. You should now get the correct value from the number variable:

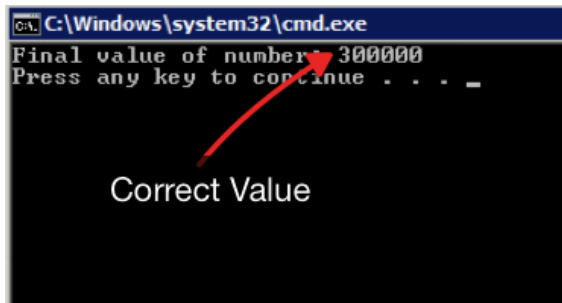


Figure 23

Note: Run the program various times to make sure that the correct result is shown. The critical_section lock is now making sure that only one of the threads can modify the number variable; thus the program will not have any data races.

Exercise 3: Working with the reader_writer_lock (Background)

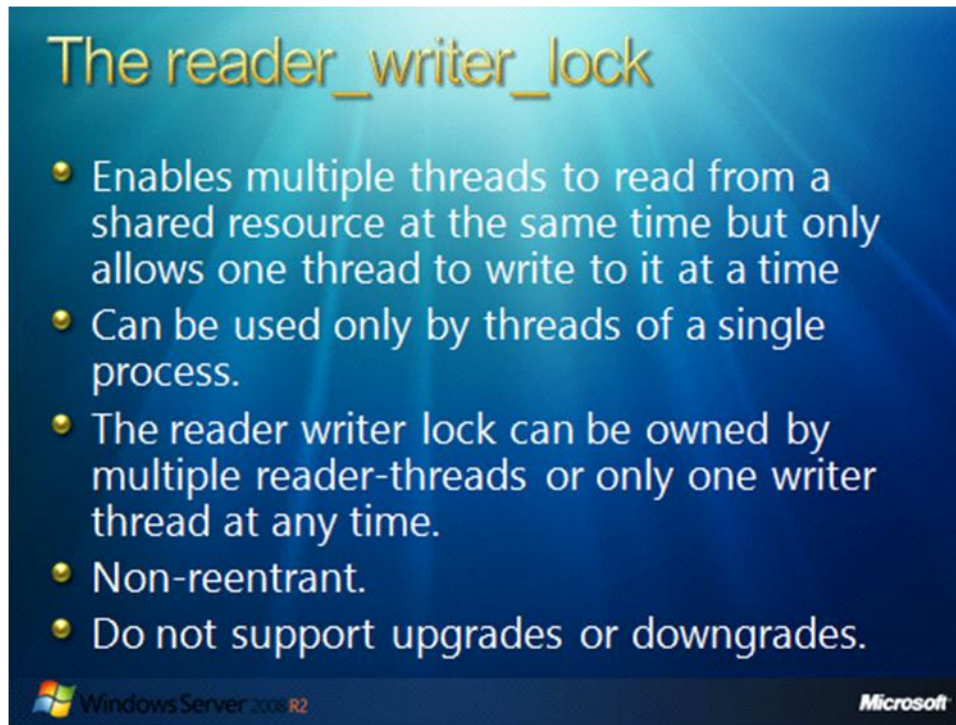


Figure 24

In the previous exercise we showed you the motivation behind using the concurrency runtime's synchronization primitives. You learned how the **critical_section** could be used to synchronize shared data between threads.

If you are ever faced with a scenario where you have multiple readers and very few writers, then using a **critical_section** when readers read the variable might not be the best idea. The Concurrency Runtime introduces a new class called the **reader_writer_lock** that can be of great help when working with such scenarios.

The reader_writer_lock

This class enables multiple threads to read from a shared resource at the same time but only allows one thread to write to it at a time. They share many characteristics with concurrency runtime's critical section, reader writer locks are non-reentrant and block cooperatively. The reader writer lock resembles the Win32 Slim reader/writer locks (SRWLock). The **reader_writer_lock** performs better than critical section in read-mostly environments.

Similarity to Win32 Slim reader/writer locks

- Can be used only by threads of a single process.
- The reader writer lock can be owned by multiple reader-threads or only one writer thread at any time.
- Non-reentrant.
- Do not support upgrades or downgrades.

Differences with Win32 Slim reader/writer locks:

- The concurrency runtime's reader writer lock object guarantees that the order of exclusive (writer) lock-ownership is on a first-come, first-serve basis.
- There is no need to explicitly call Initialization of resources before use of the concurrency runtime's reader writer lock and release after the use of the reader writer lock.
- Cannot specify spin count for the concurrency runtime's reader writer lock object.
- The concurrency runtime's reader writer lock enforces cooperative blocking where they yield to other cooperative tasks in the runtime when blocked.
- The concurrency runtime's reader writer locks give writer preference over readers; i.e. if there are readers and writer(s) simultaneously waiting for the lock, the lock would be handed over to the first writer in queue.
- Exceptions are thrown by the concurrency runtime's reader writer lock object; on recursive calls, or if unlock is called when the lock is not held, or if a lock is destroyed when being held.

Exercise 3: Working with the reader_writer_lock

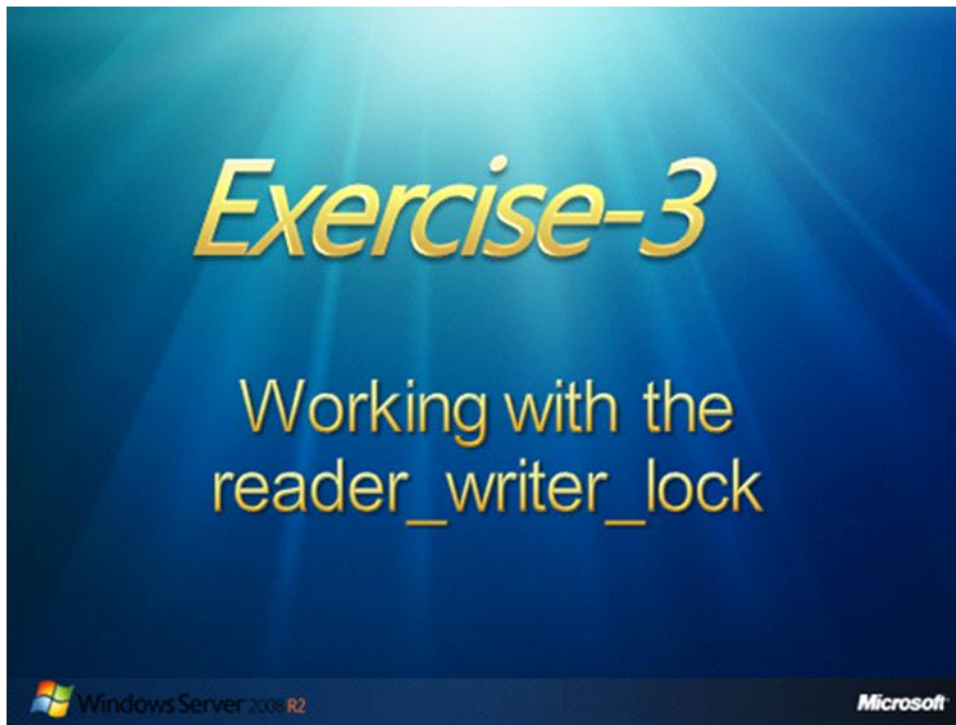


Figure 25

In this exercise you will spawn various threads that will read from a shared variable. The code will also create a thread that will write to the shared data. As you will see, sometimes there will be inconsistencies when a writer thread changes a variable at the same time that thread has just read from it.

You will then use a `reader_writer_lock` to synchronize access so that no data races occur while allowing multiple readers to simultaneously read from the shared data.

Part 1: Running the Unsynchronized Code

1. Open the Exercise-3solution file found at the following folder:

```
C:\Server 2008 R2 Labs\Working with the CRT\Exercise-3\
```

2. From the **Solution Explorer**, double click the **Exercise-3.cpp** file under the **Sources** folder.
3. Scroll down to **main** and focus on the **parallel_invoke** call:

C++

```
parallel_invoke(  
    [&] { Reader(); },  
    [&] { Reader(); },  
    [&] { Reader(); },  
    [&] { Reader(); },  
    [&] { Writer(); }  
);
```

Note: In this example, we will simulate a reader and writer scenario. We'll spawn 4 reader threads and 1 writer thread. The Reader threads will iterate in a loop and read the value from a shared variable. The Writer thread will also iterate and will write to the same shared variable.

4. Scroll up and examine the bodies of the **Reader()** and **Write()** methods.

Note: The **Reader** method simply reads from the **shared Data** variable and writes the fetched value on screen. The **Writer** method increments the variable and writes to screen when it did.

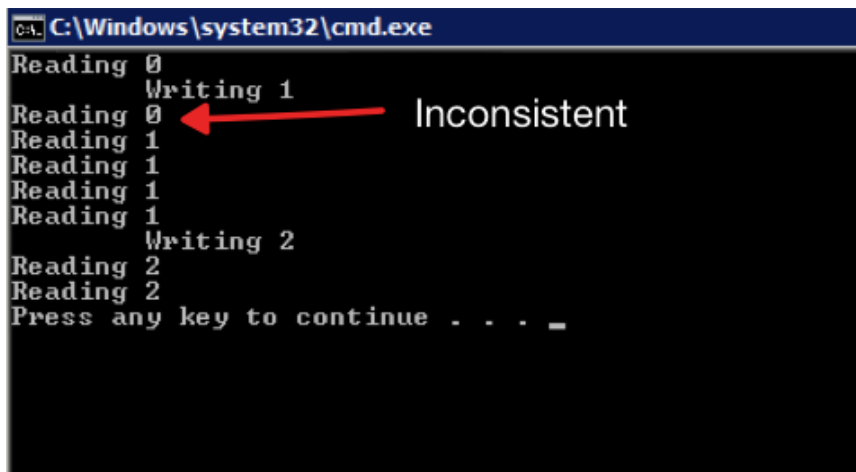
5. Before we start the program, scroll to the top of the file, you should see the **NUM_ITERATIONS** constant, which is used to specify how many the readers and writers will read and write, respectively, the variable:

C++

```
static const int NUM_ITERATIONS = 2;
```

6. From the **Debug** menu, select **Start without Debugging** (click **Yes** if prompted to build the project)

Note: If you have more than one processor/core in your machine, then you might see the scenario below (try running the programs several times until you can see the data race condition):



```
C:\Windows\system32\cmd.exe
Reading 0
Writing 1
Reading 0
Reading 1
Reading 1
Reading 1
Reading 1
Writing 2
Reading 2
Reading 2
Press any key to continue . . . _
```

The screenshot shows a Windows command prompt window with the title bar "C:\Windows\system32\cmd.exe". The output of a program is displayed on a black background with white text. The output shows a sequence of "Reading" and "Writing" operations. A red arrow points from the word "Inconsistent" to the second "Reading 0" line, indicating a race condition where a reader accessed the resource while a writer was also active.

Figure 26

Note: The readers only want to read from the resource, the writer wants to write to it. Obviously, there is no problem if two or more readers access the resource simultaneously. However, if a writer and a reader or two writers access the resource simultaneously, the result becomes indeterminable (which is the case you see above). Therefore the writers must have exclusive access to the resource.

7. Press any key to close the program's window and return to Microsoft Visual Studio 2010.

Part 2: Synchronization using the reader_writer_lock

1. Back in Visual Studio, scroll down to the main function.

Note: In order to synchronize data access between the threads, we could use a `critical_section`, but that would block all readers from reading the shared variable when one Reader is reading from it, which would be a waste. To circumvent this problem, we'll use a different kind of synchronization primitive.

2. Declare the following **reader_writer_lock** variable as the first line of the main function:

C++

```
reader_writer_lock rwlock;
```

Note: This **reader_writer_lock** class enables multiple threads to read from a shared resource at the same time but only allows one thread to write to it at a time.

3. Send the lock as a parameter to the functions **Reader** and **Writer** by making the following modifications to the **parallel_invoke** function:

C++

```
parallel_invoke(  
    [&] { Reader(&rwlock); },  
    [&] { Reader(&rwlock); },  
    [&] { Reader(&rwlock); },  
    [&] { Reader(&rwlock); },  
    [&] { Writer(&rwlock)  
});
```

4. Scroll up until you see the **Reader** method.
5. Change the method's declaration to specify that it requires an incoming parameter of type **reader_writer_lock**:

C++

```
void Reader(reader_writer_lock* pRWLock)
```

6. Inside the **Reader** method's for-loop, obtain the lock that will guarantee that the value of the **shareData** value won't be modified when it's being read and will not block other threads from reading it:

C++

```
pRWLock->lock_read();
```

7. Once the variable is read, we must release the lock to allow other writer threads to change it if needed. Release the lock by typing the following code **below** the line where the **Sleep** method is invoked:

C++

```
pRWLock->unlock();
```

8. Scroll down to the **Writer** function.
9. Change the method's declaration to specify that it requires an incoming parameter of type **reader_writer_lock**:

C++

```
void Writer(reader_writer_lock* pRWLock)
```

10. Inside the **Writer** method's for-loop, capture the lock that will prevent other readers (and writers, if there were any) to read the lock:

C++

```
pRWLock->lock();
```

11. Once the variable is written to, we must release the lock to allow other reader/writer threads to read/write to it (respectively) if needed. Release the lock by typing the following code **below** the line where the **Sleep** method is invoked:

C++

```
pRWLock->unlock();
```

12. From the **Debug** menu, select **Start Without Debugging** (click **Yes** if prompted to build the project)

Note: You should now see that there are no data race conditions between the writer thread and the reader threads:

```
C:\Windows\system32\cmd.exe
Reading 0
Reading 0
Writing 1
Reading 1
Reading 1
Reading 1
Reading 1
Writing 2
Reading 2
Reading 2
Press any key to continue . . . _
```

Figure 27

Note: An interesting point to note here is that even though we use 4 readers, we notice that only 2 readers output their value and the writer takes the lock. This happens because the lock prefers writers. We do not guarantee the order of task execution but if such a guarantee is required, you could consider using events (next exercise).

Exercise 4: Working with the Concurrency Runtime Events (Background)

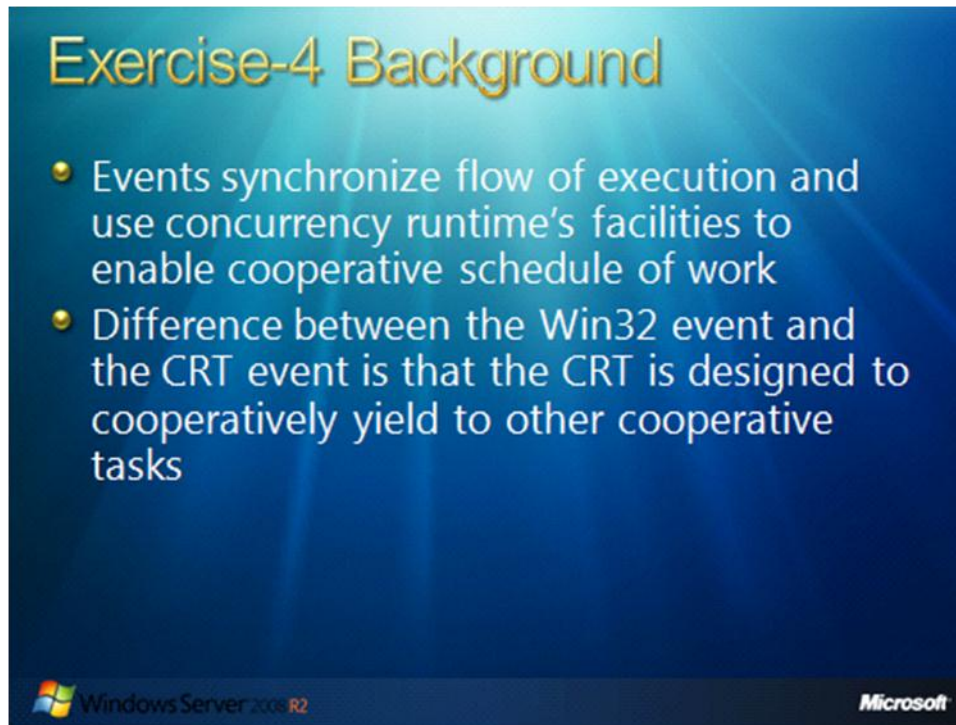


Figure 28

So far we have seen how to synchronize data using the **critical_section** and **reader_writer_lock** synchronization primitives. Next we will see work with the concurrency runtime's event.

This is a bi-state type class that, unlike Critical Section or Reader Writer Lock, does not protect access to shared data. Events synchronize flow of execution and use concurrency runtime's facilities to enable cooperative schedule of work. They behave similar to Win32 manual-reset event. The main difference between the concurrency runtime's event and Win32 event is that the concurrency runtime's event are designed to cooperatively yield to other cooperative tasks in the runtime when blocked in addition to preempting whereas Win32 events are, by design purely pre-emptive in nature.

User Mode Scheduling

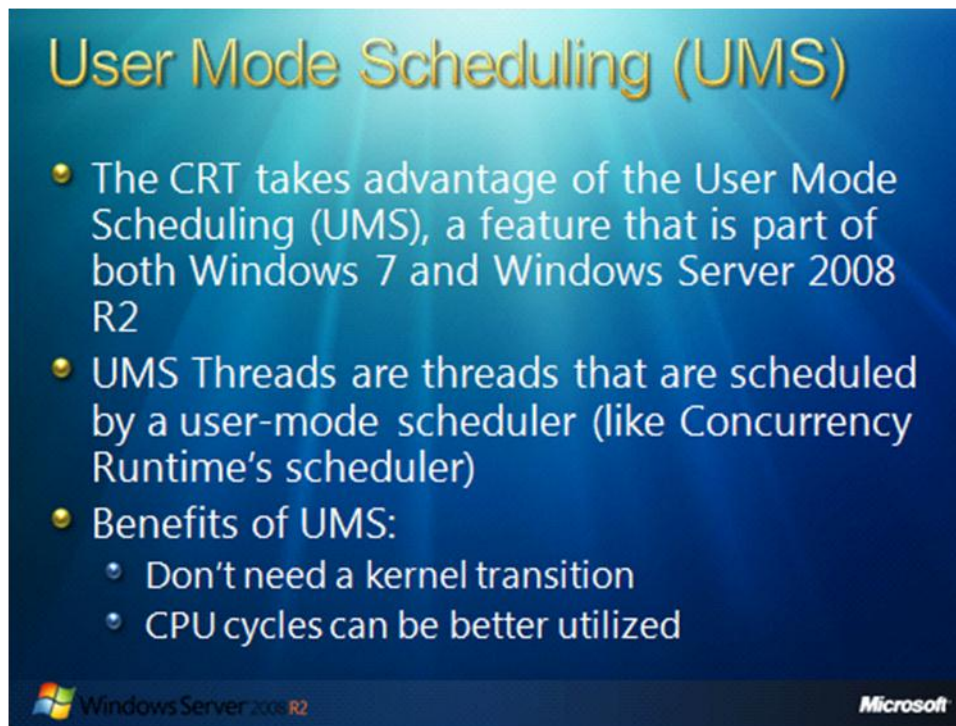


Figure 29

One of the benefits of working with the Concurrency Runtime events is that when they are blocked and cannot go any further will yield to other threads. Concurrency Runtime takes advantage of new User Mode Scheduling (UMS) technology, a feature that is part of both Windows 7 (x64) and Windows Server 2008 R2.

As the name implies, UMS Threads are threads that are scheduled by a user-mode scheduler (like the Concurrency Runtime's scheduler). Scheduling threads in user mode has a couple of advantages:

1. A UMS Thread can be scheduled without a kernel transition, which can provide a performance boost.
2. Full use of the OS's quantum can be achieved if a UMS Thread blocks on a system call or sync event.

To illustrate the benefits of UMS, let's picture a scenario of a very simple scheduler that has a single work queue. In this case, we are trying to schedule 100 tasks in a computer that has 2 CPUs as shown in the diagram below:

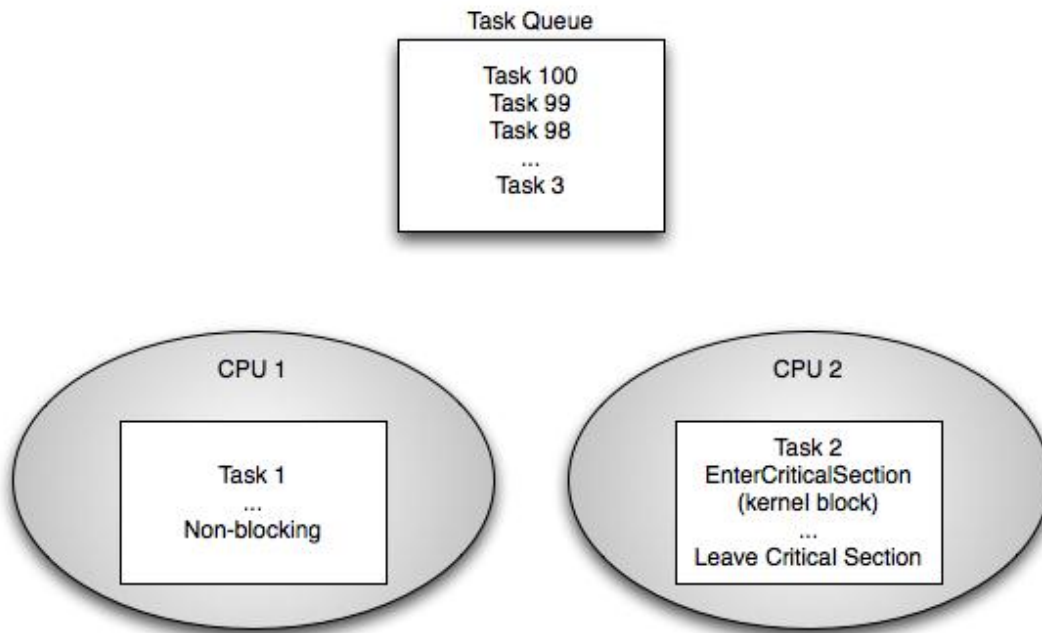


Figure 30

Here we've started with 100 items in our task queue, and two threads have picked up Task 1 and Task 2 and are running them in parallel. Unfortunately, Task 2 is going to block on a critical section. Obviously, we would like the scheduler (i.e. the Concurrency Runtime) to use CPU 2 to run the other queued tasks while Task 2 is blocked. Alas, with ordinary Win32 threads, the scheduler cannot tell the difference between a task that is performing a very long computation and a task that is simply blocked in the kernel. The end result is that until Task 2 unblocks, the Concurrency Runtime will not schedule any more tasks on CPU 2. Our 2-core machine just became a 1-core machine, and in the worst case, all 99 remaining tasks will be executed serially on CPU 1.

User Mode Scheduling (cont'd)

This situation can be improved somewhat by using the Concurrency Runtime's cooperative synchronization primitives (critical_section, reader_writer_lock, event) instead of Win32's kernel primitives. These runtime-aware primitives will cooperatively block a thread, informing the Concurrency Runtime that other work can be run on the CPU. In the above example, Task 2 will cooperatively block, but Task 3 can be run on another thread on CPU 2. All this involves several trips through the kernel to block one thread and unblock another, but it's certainly better than wasting the CPU:

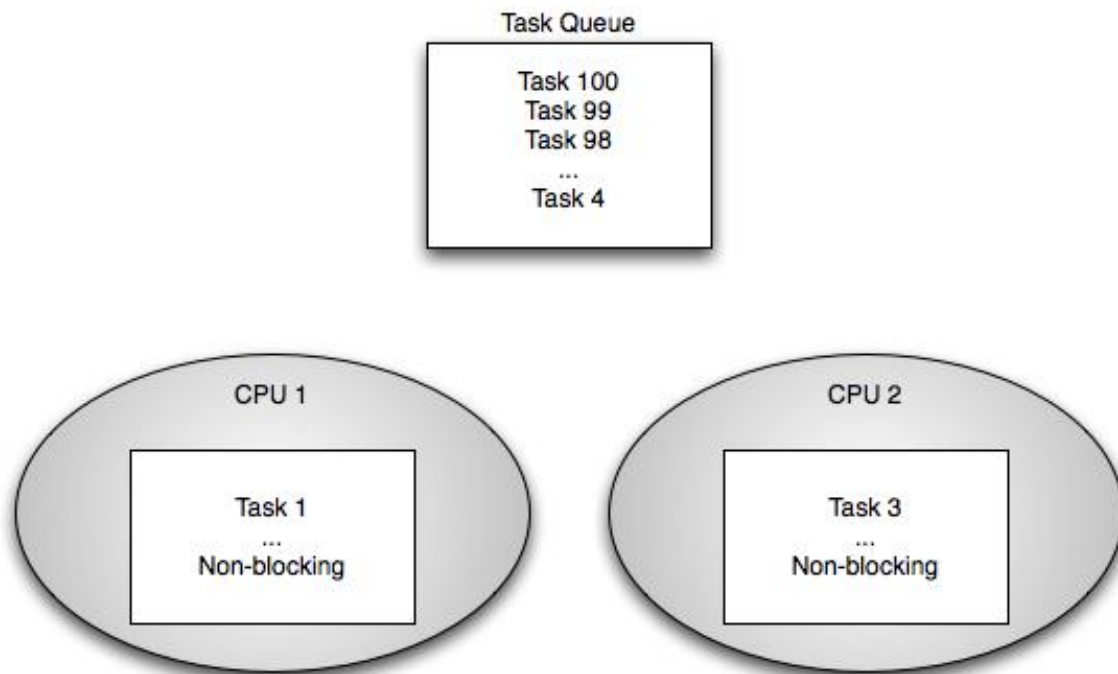
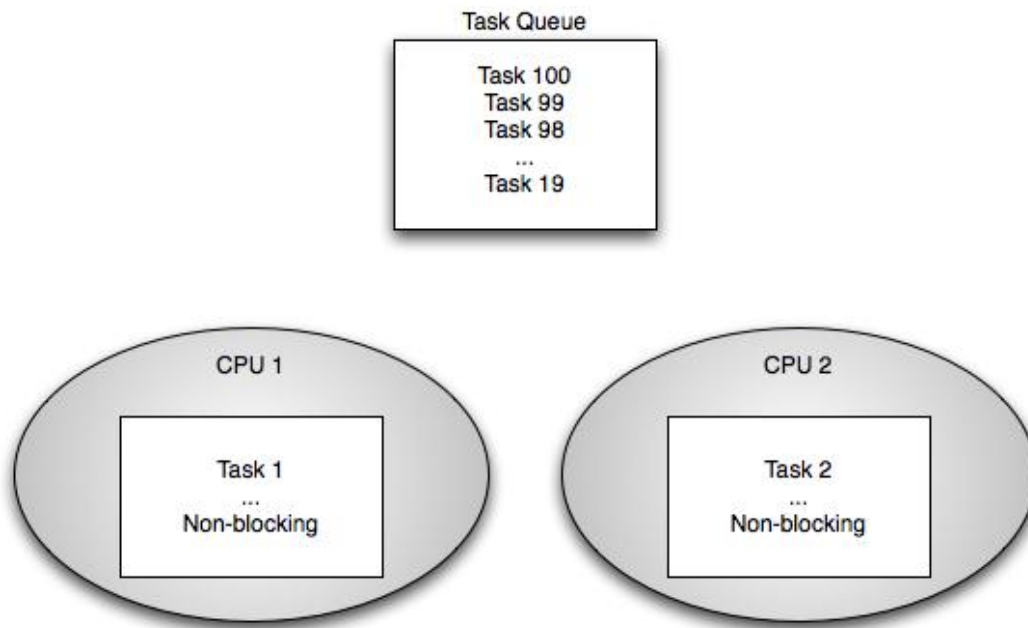


Figure 31

The situation is improved even further on Windows Server 2008 R2 with UMS threads. When Task 2 blocks, the OS gives control back to Concurrency Runtime. It can now make a scheduling decision and create a new thread to run Task 3 from the task queue. The new thread is scheduled in user-mode by the Concurrency Runtime, not by the OS, so the switch is very fast.

Both CPU 1 and CPU 2 can now be kept busy with the remaining 99 non-blocking tasks in the queue. When Task 2 gets unblocked, Windows Server 2008 R2 places its host thread back on a runnable list so that the Concurrency Runtime can schedule it – again, from user-mode – and Task 2 can be continued on any available CPU:

**Figure 32**

The Concurrency Runtime looks for unblocked (runnable) tasks before queued tasks, so it might be somewhat more likely that Task2 will run before the next queued task (19)

Exercise 4: Working with the Concurrency Runtime Events

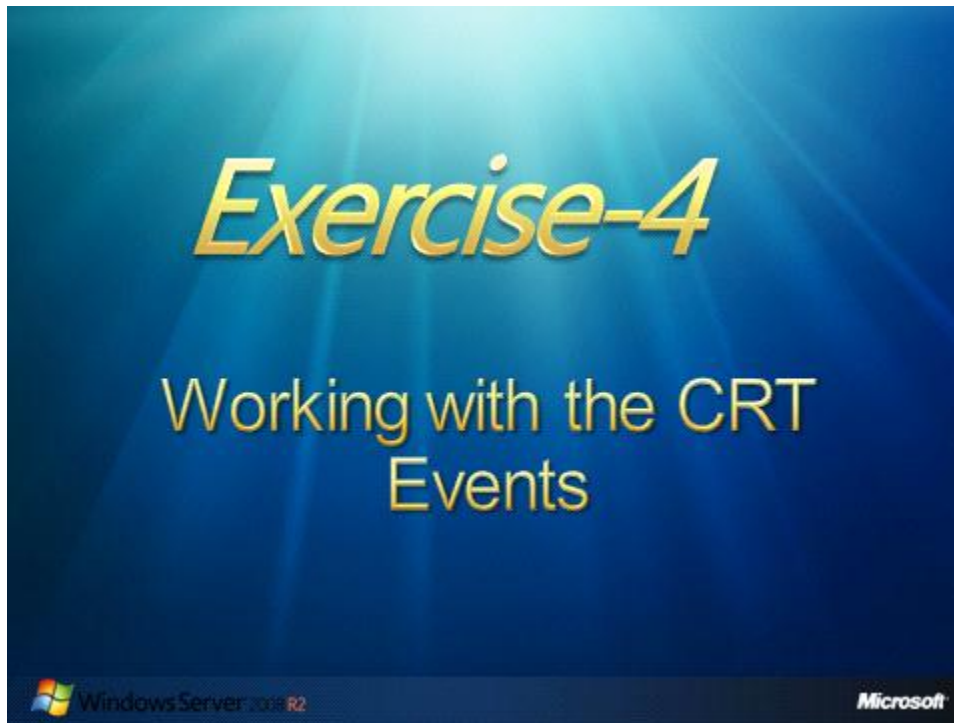


Figure 33

Now that you understand how the Concurrency Runtime takes advantage of UMS to schedule threads to yield execution to other threads when a thread blocks, let's try and see this behavior in code.

In this exercise you will work to schedule various tasks using a Win32 event and observe its behavior when a thread blocks and other threads are waiting. You will then use the Concurrency Runtime event instead of the Win32 event to see if there are any changes when the threads are scheduled.

Part 1: Scheduling the Work

1. Open the **Exercise-4** solution file found at the following folder:

```
C:\Server 2008 R2 Labs\Working with the CRT\Exercise-4\
```

2. From the **Solution Explorer**, double click the **Exercise-4.cpp** file under the **Sources** folder.
3. Examine the code for the **WindowsEvent** class.

Note: This class is a wrapper class to a windows event. We've made the class have the same interface as our runtime cooperative event so that you can easily switch back and forth to appreciate the benefits of cooperative scheduling by later on just changing the event declaration.

4. Scroll down to the main function (around line 37). Write the following line below the comment that reads:

C++

//Create a scheduler that uses two threads:

```
CurrentScheduler::Create(SchedulerPolicy(1, MaxConcurrency, 2));
```

Note: The Concurrency Runtime scheduler is policy driven, so we are talking to the current scheduler and creating a new scheduler policy that will allow a maximum of 2 threads, specified by the parameters sent to SchedulerPolicy.

5. Below that comment that reads **//Declare an Event**, write the following line of code;

C++

```
WindowsEvent e;
```

Note: This is the straightforward declaration of the WindowsEvent wrapper class that you previously examined.

6. Below the comment that reads **//Create a Lambda Waiting for the Event**, write the following code snippet:

```
auto task = [&e]() {  
    printf_s("Waiting in a task for the event\n");  
    e.wait();  
};
```

Note: The **auto** keyword in this case is analogous to the **var** type if you are familiar with VB.Net – it lets the compiler determine the type. In this case we are defining what the task will do when it is later scheduler to run. The lambda [&e] sets the event to be passed by reference to the body of the function that requires no parameters.

The function will print the message to screen and will wait for the event **e** to be set before going any further.

7. In order to schedule the tasks, we will use a **taskgroup**. To do so, type down the following code snippet below the comment that reads *//Create a taskgroup and schedule multiple copies of the task*

C++

```
task_group tg;
for(int i = 0; i < 10; ++i) {
    tg.run(task);
}
```

Note: A task_group is a class defined in <ppl.h> and is used to schedule tasks, wait for them all to complete or cancel any outstanding tasks assigned to it. In this case we are adding 10 instances of the task we defined in step 6 to run.

8. In order to allow the tasks to be scheduled and understand what's going on underneath, sleep the main thread for one second by writing the following line below the comment *//Sleep*

```
Sleep(1000);
```

9. We will now set the event and all tasks that were currently waiting for this to happen will continue execution. Write down the following line below the comment *//Wait for the events*:

C++

```
printf_s(" Setting the event\n");
e.set();
```

10. Next, we must wait for all tasks to finish before going any further. The task_group class allows this by using the wait method. Write down the following line below the *//Wait for the tasks* comment:

C++

```
tg.wait();
```

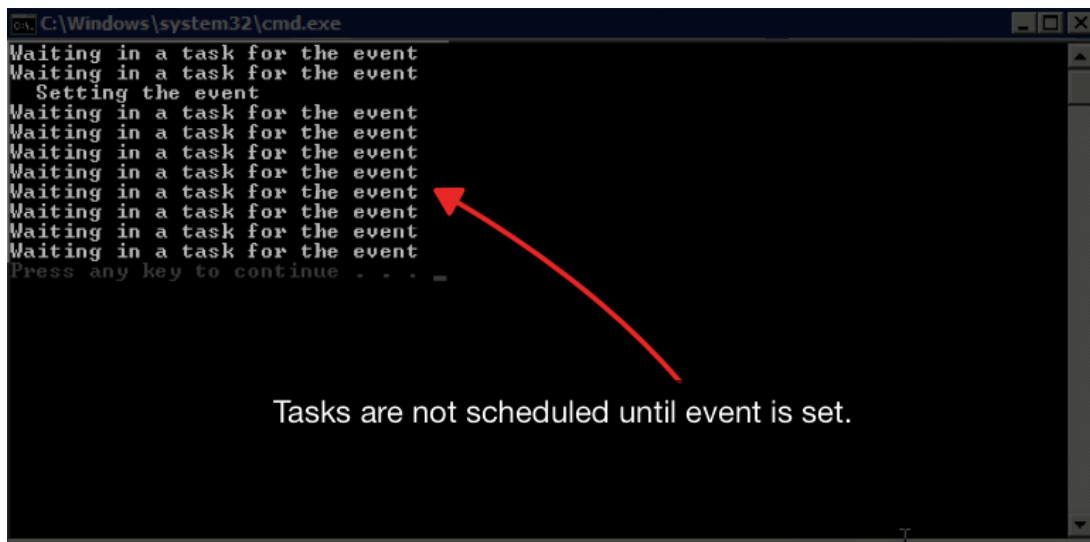
11. From the **Debug** menu, select **Build → Rebuild Solution**(click Yes if prompted to build the project)

Note: Your build should succeed. If you encounter any errors, please backtrack your steps and double-check each instruction.

Part 2: Running the Windows Event

Note: Now that the code is ready, you will run the program and observe its behavior when the tasks are running. Keep in mind at all times that our custom scheduler policy only allows two concurrent threads to run.

1. From the **Debug** menu, select **Start without Debugging**(click **Yes** if prompted to build the project)
2. Take a look at the order in which the tasks are scheduled:



```
C:\Windows\system32\cmd.exe
Waiting in a task for the event
Waiting in a task for the event
Setting the event
Waiting in a task for the event
Waiting in a task for the event
Waiting in a task for the event
Waiting in a task for the event
Waiting in a task for the event
Waiting in a task for the event
Waiting in a task for the event
Press any key to continue . . .
```

Tasks are not scheduled until event is set.

Figure 34

Note: By looking at the output, you can see that two tasks are immediately set. Since we are not using a cooperative scheduling, the two threads will not yield to other threads even though they are sitting idle waiting for the event to take place.

3. Press any key to close the program and return to Visual Studio.

Part 3: Running the Cooperative Event

1. To run the tasks using the cooperative scheduler, change the windows event to a Concurrency Runtime event by replacing the line below the `//Declare an Event` comment:

C++

```
WindowsEvent e;
```

To:

C++

```
event e
```

2. From the **Debug** menu, select **Start without Debugging**(click **Yes** if prompted to build the project)

Note: Take a look at the order in which the tasks are scheduled:

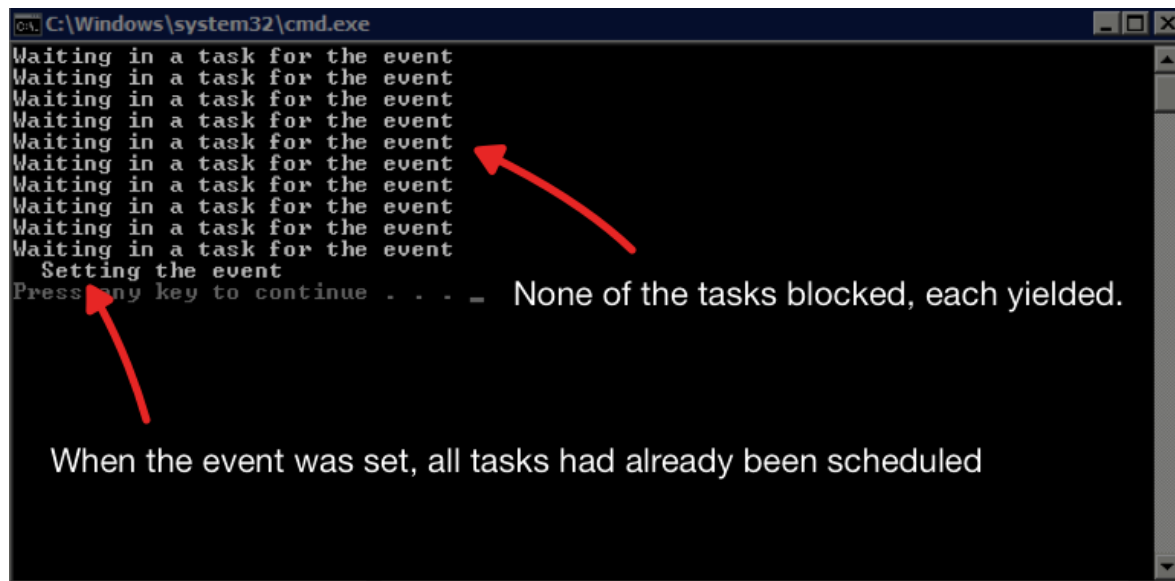


Figure 35

Note: Since now the tasks are scheduled using the Concurrency Runtime event, they yield when they are waiting for a particular event instead of blocking any incoming thread to execute.

Exercise 5: Working with Agents (Background)

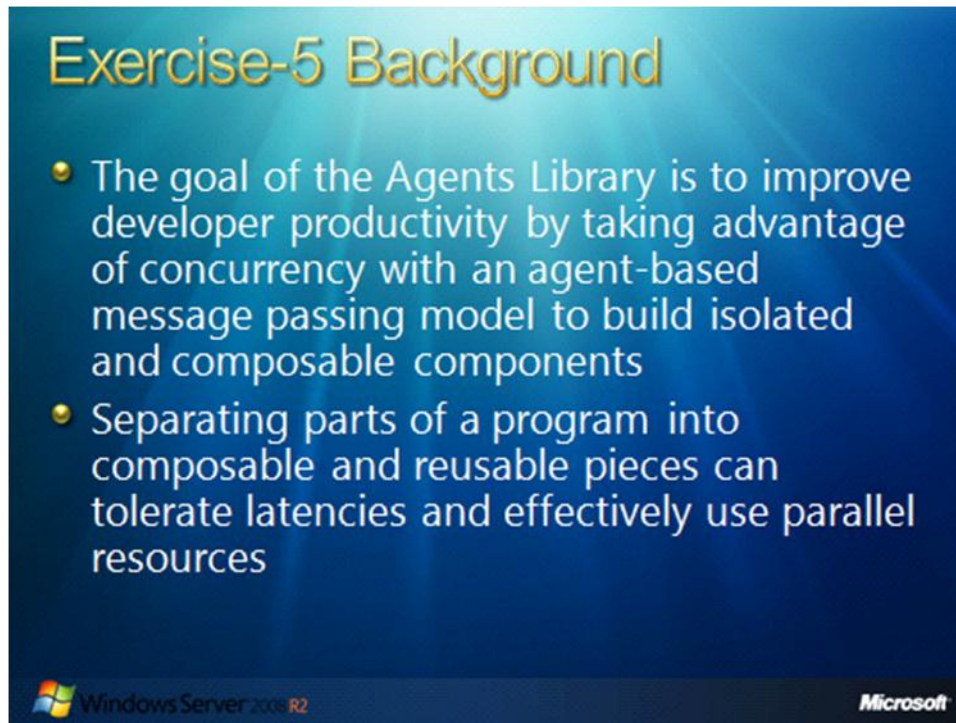


Figure 36

One of the native concurrency components coming in Visual Studio 2010 is the Asynchronous Agents Library. The goal of the Agents Library is to improve developer productivity and enable developers to take advantage of concurrency with an agent-based message passing model to build isolated and composable components.

Programming with an agent-based model allows concurrency to be inherently baked into the program from the start. Separating parts of a program into composable and reusable pieces that follow well-defined boundaries to communicate with message passing can tolerate latencies and effectively use parallel resources. By avoiding sharing memory when possible and focusing on data dependencies, scaling performance can be obtained using higher-level abstractions, like agents, for parallelism.

The Agent Class

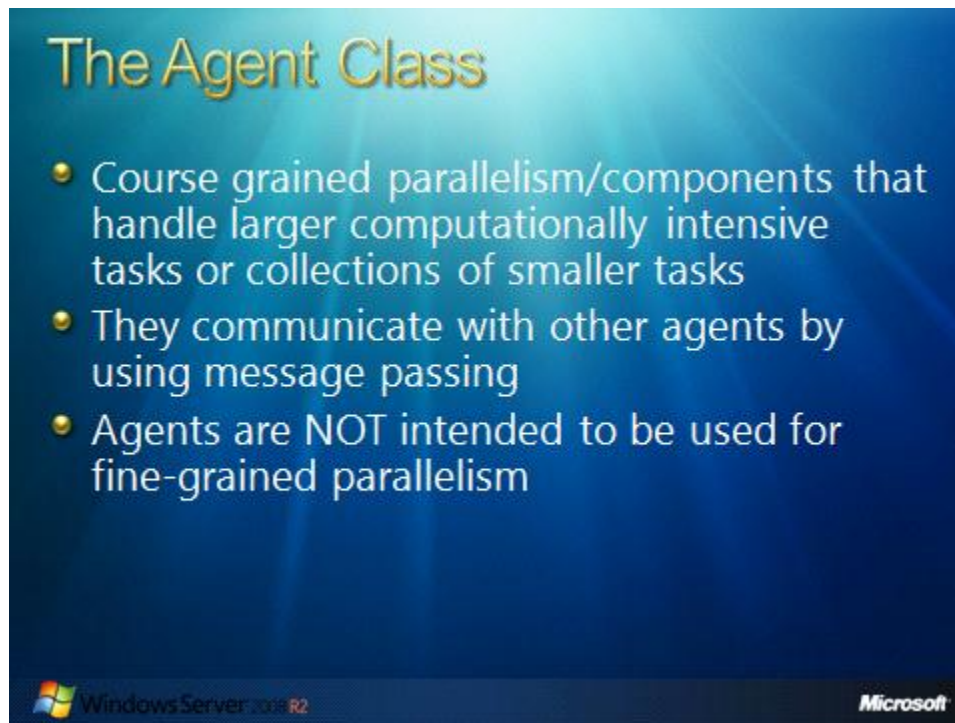


Figure 37

The agent class itself is intended for course grained parallelism/components that handle larger computationally intensive tasks or collections of smaller tasks. Fundamentally, agents are tasks that have an observable lifecycle and communicate with other agents by using message passing. Agents are NOT intended to be used for fine-grained parallelism; for that, the patterns and constructs in the Parallel Patterns Library are better suited.

Essentially an agent is a light weight task (LWT) with observable state and cancellation. The lifecycle of an agent is as follows:

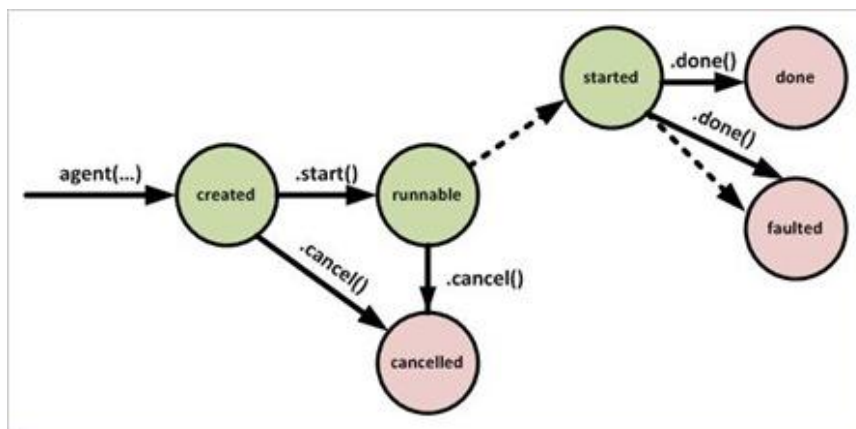


Figure 38

Messaging Blocks



Figure 39

To enable mechanisms for communicating between asynchronous agents and isolated program components, we provide a collection of various message blocks:

- `unbounded_buffer`
- `overwrite_buffer`
- `single_assignment`
- `call`
- `transformer`
- `timer`
- `join`
- `multitype_join`
- `choice`

Message blocks are intended to be used to establish defined communication protocols between isolated components and develop concurrent applications based on data flow. The message blocks provided by the Agents Library can be used in conjunction with the agents class itself or separately. Let's go through some of the ones that will be used in this exercise.

Unbounded Buffer

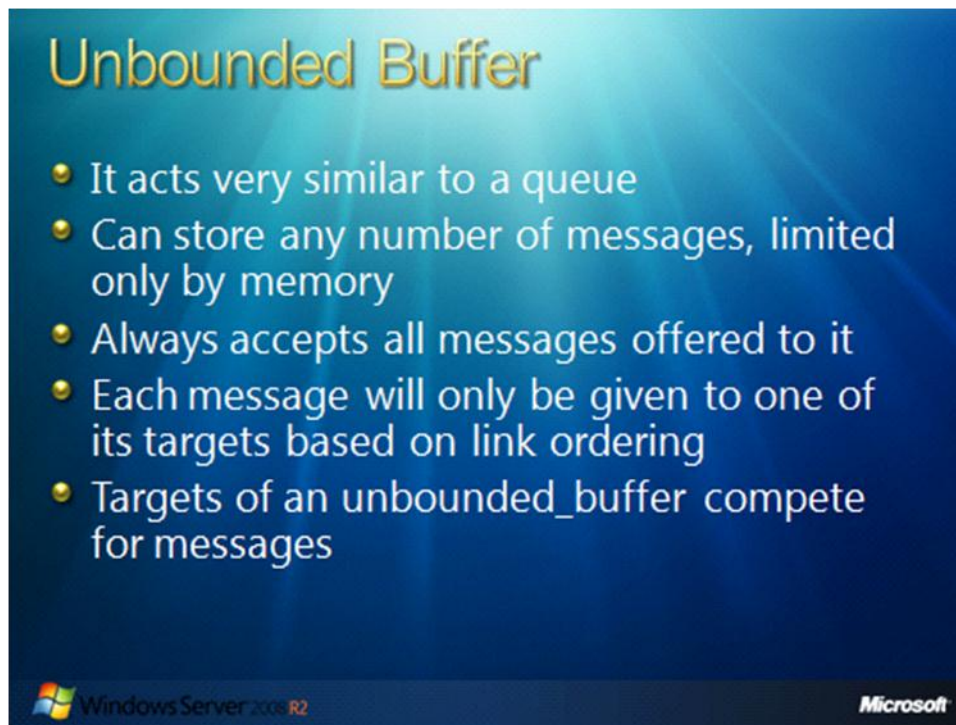


Figure 40

Unbounded_buffer is one of the most basic messaging blocks; it acts very similar to a queue. As its name suggests unbounded_buffer can store any number of messages, limited only by memory, collected from its source links (links to blocks that have the unbounded_buffer as a target).

Unbounded_buffer always accepts all messages offered to it. Messages propagated to an unbounded_buffer are collected into a queue and then offered one at a time to each of its targets. Each message in an unbounded_buffer will only be given to one of its targets based on link ordering. This means targets of an unbounded_buffer compete for messages.

Overwrite Buffer

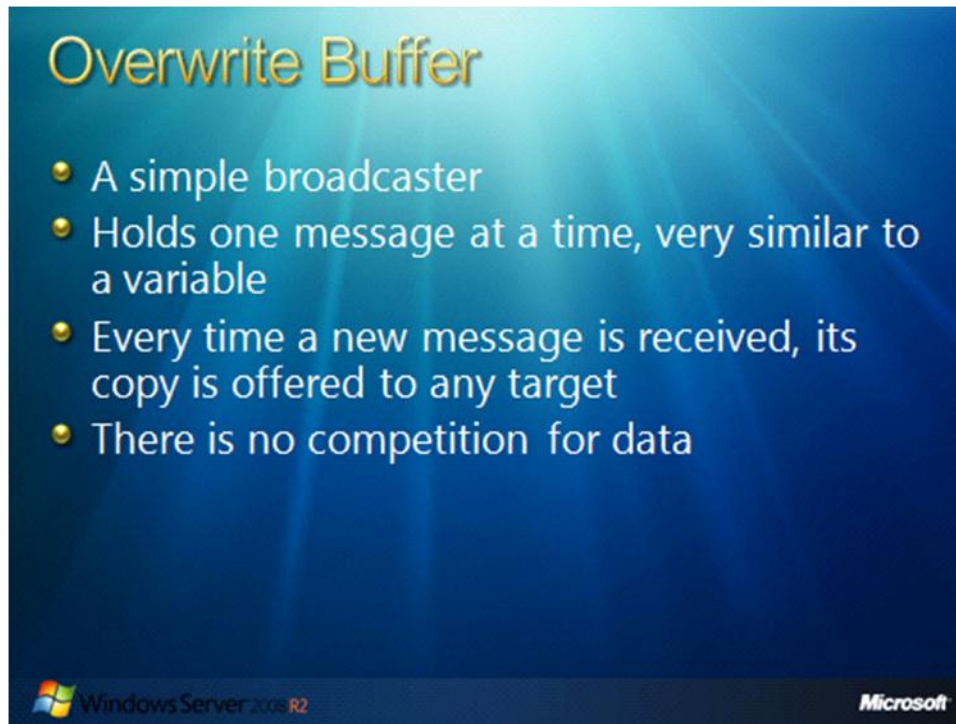


Figure 41

Essentially, an **overwrite_buffer** is a simple broadcaster. Overwrite_buffer is a message block that holds one message at a time, very similar to a variable. Every time an overwrite_buffer receives a message it offers a copy of it to any of its targets and then stores the message internally, replacing any previously stored message. The important thing to note here is there is no competition for data. Every time a message comes into an overwrite_buffer it is offered to all of its targets, then afterwards it can be overwritten at any point.

Exercise 5: Working with Agents

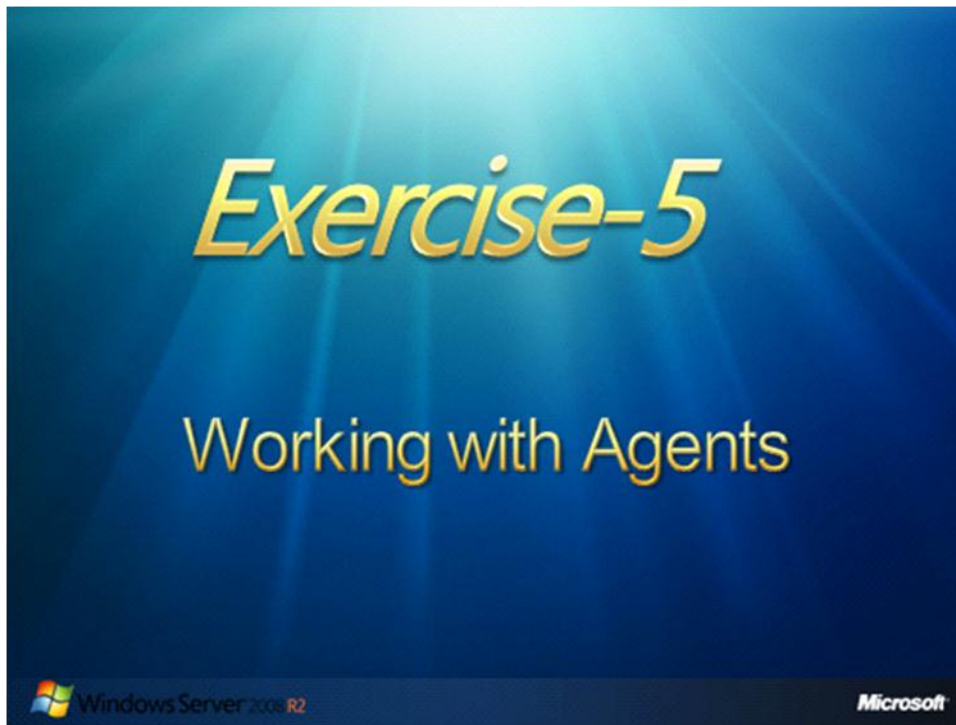


Figure 42

In this exercise you will work with a code that will show you how agents communicate with one another. We'll make use of an unbounded buffer and an overwrite buffer to have a couple of agents communicate with one another. The diagram below shows what will be accomplished:

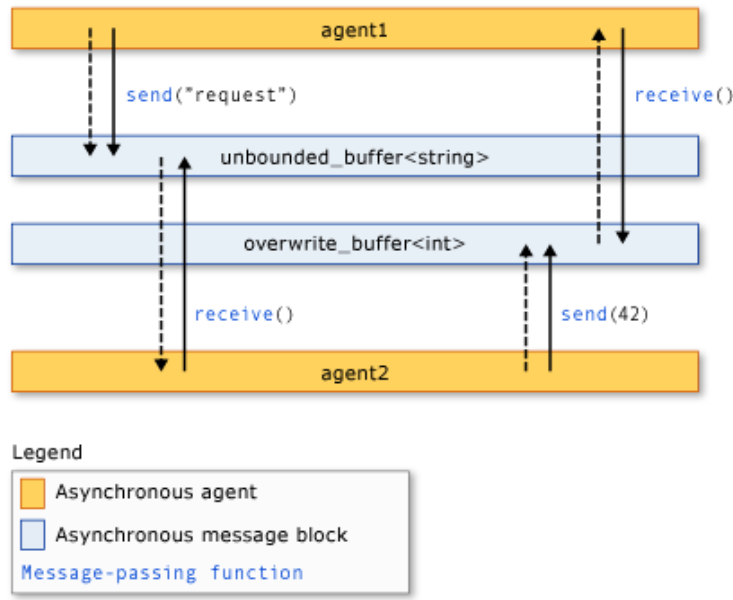


Figure 43

In this illustration, agent1 sends a message to agent2 by using the send function and an unbounded_buffer object. agent2 uses the receive function to read the message. agent2 uses the same method to send a message to agent1. Dashed arrows represent the flow of data from one agent to the other. Solid arrows connect each agent to the message blocks that they write to or read from.

Part 1: Create the Buffers

1. Open the **Exercise-5** solution file found at the following folder:

```
C:\Server 2008 R2 Labs\Working with the CRT\Exercise-5\
```

2. From the **Solution Explorer**, double click the **Exercise-5.cpp** file under the **Sources** folder.
3. Scroll down to the main function to make all the required changes.
4. If you recall from our diagram, agent1 will send a value to agent2 via an unbounded buffer. To declare this buffer, write the following line right below the `// Declare the unbounded buffer below` comment:

C++

```
unbounded_buffer<string> buffer1;
```

Note: An **Unbounded_buffer** is one of the most basic messaging blocks; it acts very similar to a queue. As its name suggests **unbounded_buffer** can store any number of messages, limited only by memory, collected from its source links (links to blocks that have the **unbounded_buffer** as a target).

5. On the other hand, agent2 will send a single value to agent1 (an int). In order to set the overwrite buffer, write the following line below the `// Declare the overwrite buffer below` comment:

C++

```
overwrite_buffer<int> buffer2;
```

Part 2: Create and Start the Agents

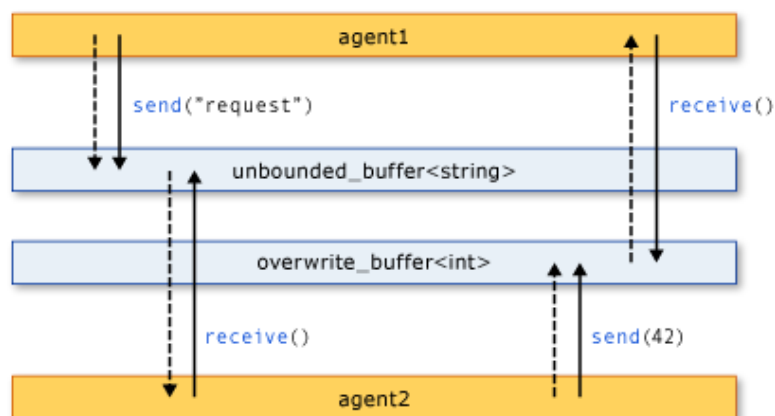
1. While still in main, we need to create the agents that will be used. Write down the following lines below the `// Step 2: Create the agents` comment:

C++

```
agent1 first_agent(buffer2, buffer1);
agent2 second_agent(buffer1, buffer2);
```

Note: Soon enough you will see the constructor signatures of the agents, but for the time being remember that **agent1** will send out **buffer1** to **agent2** and will receive the message sent by **agent2** in **buffer2**.

We've included the diagram again to make it clearer:



Legend

- Asynchronous agent
- Asynchronous message block
- Message-passing function

Figure 44

2. In order to start the agents and have them execute in parallel, type the lines below right after the `// Step 3: Start the agents`. The runtime calls the `run` method on each agent comment:

C++

```
first_agent.start();
second_agent.start();
```


3. Next, we need to wait for both agents to finish, which can be done by invoking the wait method. Type the following lines right below the `// Step 4: Wait for both agents to finish` comment:

C++

```
agent::wait(&first_agent);  
agent::wait(&second_agent);
```

Part 3: Implement agent1

1. Scroll all the way to the top of the file. You will see the declaration of **agent1**, which inherits from the agent class:

C++

```
class agent1 : public agent
```

2. Further down, you will see the constructor for the agent class:

C++

```
public:  
    explicit agent1(ISource<int>& source, ITarget<string>& target)  
        : _source(source)  
        , _target(target)
```

Note: The **ISource** class is the interface for all Source blocks. Source blocks have messages to pass to Target blocks. The **ITarget** class is the interface for all Target blocks. Target blocks consume messages passed down by ISources.

Also notice that the constructor is using an r-value reference in its parameter list (the &). This is a new language construct in Visual Studio 2010 that is part of the draft C++0x standard. An r-value reference here allows the compiler to avoid a copy constructor call and move source and target into `_source` and `_target`, respectively.

3. Scroll down until you see the run method. The first thing we want to do is send the target parameter agent2. To do this, write the following line below the `// Send the request` comment:

C++

```
printf("agent1: Sending Request\n");  
send(_target, string("request"));
```

4. Next, we know that agent2 is going to send us a value back, so we tell the agent to read the response by typing this line below the `// Read the response` comment:

C++

```
int response = receive(_source);  
printf("agent1: received '%d'.\n", response);
```

5. Next, we must signal that the agent has finished by calling the done method, which needs to be typed right after the `// Signal Completion` comment:

C++

```
done();
```

Part 4: Implement agent2

1. Scroll down to the agent2 class. Take a minute to examine the class, you should notice some similarities with the agent1 class.
2. Navigate to the run method of the agent2 class.
3. The first thing we must do is receive the message that was sent by agent1. To do so, type the following lines below the `// Read the request` comment:

C++

```
string request = receive(_source);  
printf("agent2: received '%s'.\n", request.c_str());
```

4. Next, we'll send a message to agent1 via the overwrite buffer. In this case, we'll send the value of 42. Write down the following line below the `//Send the response` comment:

C++

```
printf("agent2: sending response...\n");  
send(_target, 42);
```

5. Next, we must signal that the agent has finished by calling the done method, which needs to be typed right after the `// Signal Completion` comment:

C++

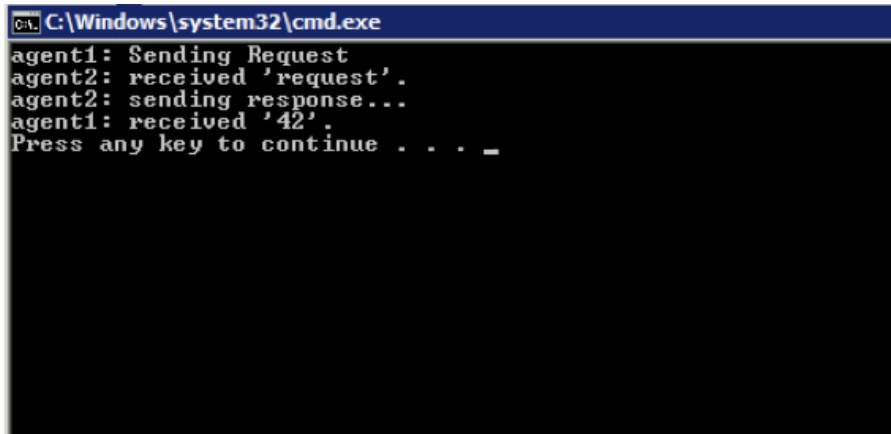
```
done();
```

6. From the Debug menu, select Build → Rebuild Solution

Note: Your build should succeed. If you encounter any errors, please backtrack your steps and double-check each instruction.

Part 5: Test the Program

1. From the **Debug** menu, select **Start without Debugging**
2. Your output should be similar to the one show below:



```
C:\Windows\system32\cmd.exe
agent1: Sending Request
agent2: received 'request'.
agent2: sending response...
agent1: received '42'.
Press any key to continue . . . _
```

Figure 45

Note: As you can see from the output, agent1 starts up and sends the request to agent2, which was already started and waiting for the request from agent1. Once agent2 received the request, it sent the response (42) to agent1. At this point, agent2 had signaled that it was done and agent1 received the number. Shortly after, agent1 also signaled that it was done and the program finished executing.

Lab Summary

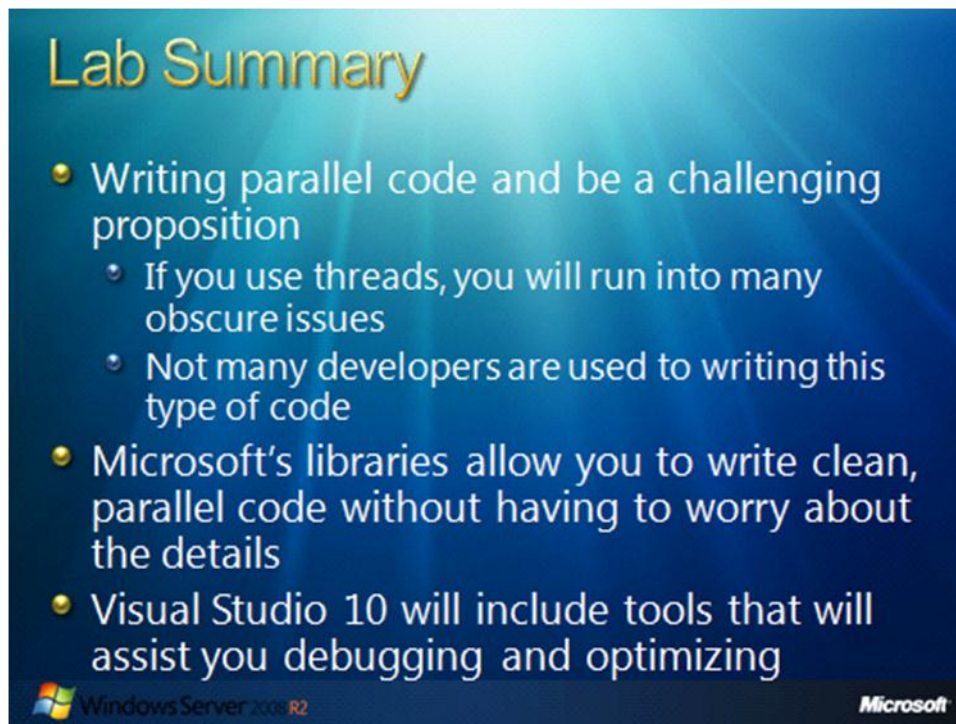


Figure 46

Writing code that executes concurrently can be a challenging proposition that can require a tremendous amount of time and resources. Writing your own work-splitting routines and synchronization implementation may require a great deal of development time and testing.

In order to make parallelization an easier task, Microsoft has encapsulated the hardest parts of parallelization into libraries for developers to exploit. By taking full advantage of the Concurrency Runtime, the Parallel Pattern and Asynchronous Agents Libraries; developers will be writing parallel code in no time without having to worry about the pesky details. Furthermore, Visual Studio 10 will be bundled with Performance tools that will help you to efficiently debug your parallel applications and optimize them accordingly.

Regardless of the parallelization implementation of choice, you should start evaluating your applications in order to find out if there are processor intensive tasks that would benefit from parallelization.

