NETWORK PROGRAMMING

Created by MAI NGỌC CHÂU

MULTI THREAD

CHAPTER 9

Introduction

- A ***thread*** is defined as a single flow of operation within a program. When a program executes on the CPU, it traverses the program statements in a single thread until the thread is complete.
- A ***multithreaded*** application distributes functions among multiple program flows, allowing two or more paths of execution to occur. Each path of execution is a separate thread.

Introduction (cont. 1)

- Multithreaded programs create their own threads that can be executed separately from the main thread of the program.
- All threads created by the primary thread share the same memory address space as the primary thread.

Introduction (cont. 2)

- Often the secondary threads are used to perform computationally intensive functions, allowing the main thread to continue responding to Windows events.
- Without the secondary threads, the user could not select anything from the menu or click any buttons while an application computed a mathematical function.

Introduction (cont. 3)

- The thread is the basic unit of execution on the Windows operating system.
- Each process is scheduled to run on the CPU by its threads.
- On a multiprocessor system, more than one thread can run at a time, allowing the operating system to schedule multiple threads either from the same process, or from separate processes to run simultaneously.

Thread State

Thread State	Description
Initialized	The thread has been initialized but not started.
Ready	The thread is waiting for a processor.
Running	The thread is currently using the processor.
Standby	The thread is about to use the processor.
Terminated	The thread is finished and is ready to exit.
Transition	The thread is waiting for a resource other than the processor.
Unknown	The system is unable to determine the thread state.
Wait	The thread is not ready to use the processor.

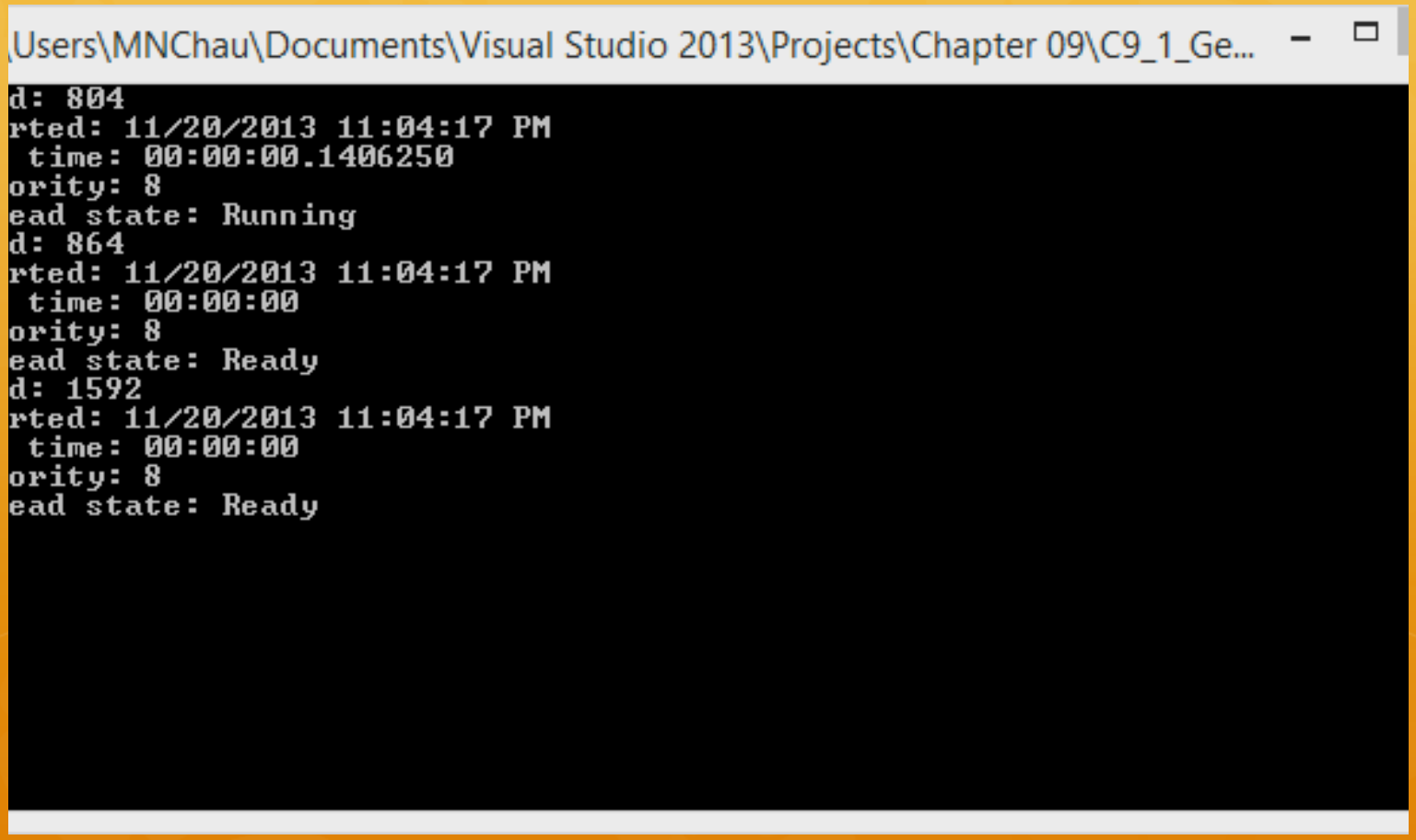
Thread State (cont.)

- Threads have several operational states, which are enumerated using the .NET *ThreadState* enumeration, found in the ***System.Threading*** namespace.
- An individual thread may change states several times during its lifetime, toggling between the Running state and the Standby, Transition, or Wait states. The operating system itself can place a running thread into another state to preempt it with the thread of a high-priority process.

Threads in a Process

```
static void Main(string[] args)
{
    Process thisProc = Process.GetCurrentProcess();
    ProcessThreadCollection myThreads = thisProc.Threads;
    foreach (ProcessThread pt in myThreads)
    {
        DateTime startTime = pt.StartTime;
        TimeSpan cpuTime = pt.TotalProcessorTime;
        int priority = pt.BasePriority;
        ThreadState ts = pt.ThreadState;
        Console.WriteLine("thread: {0}", pt.Id);
        Console.WriteLine("  started: {0}", startTime.ToString());
        Console.WriteLine("  CPU time: {0}", cpuTime);
        Console.WriteLine("  priority: {0}", priority);
        Console.WriteLine("  thread state: {0}", ts.ToString());
    }
    Console.ReadKey();
}
```

Console output



Demo 01

C9_1_GetThreads

Threads in a Process (cont.)

- As you can see in this output, the **C9_1_GetThreads** program shows three separate threads running for this process. There was only one path of execution in the program, so you may be wondering where the other two threads came from?
- The program is not a fully executable program; it is compiled to the Microsoft Intermediate Language (MSIL).
- As each MSIL program is run, the just-in-time (JIT) compiler must also run, processing each statement in the MSIL code → the multiple threads for a single program.

Threads in All Processes

- If we are using a single processor system, only one thread in the list will be in the Running state—the primary thread from the **C9_2_ListThreads** program.
- If we have a multiprocessor system, we should see the same number of Running threads as the number of processors we have.

Demo 02

C9_2_ListThreads

The Thread Class

- The C# language provides the System.Threading namespace, which includes classes for creating and controlling threads within a program.
- Use the Thread class to create a new Thread object, which produces a new thread within the current process.
- The format of the Thread constructor is:

Thread(ThreadStart *start*)

where *start* is a **ThreadStart** delegate.

The Thread Class (cont.)

- The **ThreadStart** delegate points to the method that will be performed within the new thread.

```
.
.
Thread newThread = new Thread(new ThreadStart(newMethod));
.
.
}
void newMethod()
{
.
.
}
```

- The *newMethod* parameter defines a method that is performed in the thread when it receives time on a processor.
- *You can also run methods defined in other classes.*

The Thread Class Methods

Method	Description
Abort()	Terminates the thread
Interrupt()	Interrupts a thread that is in the Wait thread state
Join()	Blocks the calling thread until the thread terminates
Resume()	Resumes a thread that has been suspended
Start()	Causes the operating system to change the thread state to Running
Suspend()	Suspends the execution of the thread

How to Start a Thread

- After the Thread object is created, the **Start()** method must be used to get it going.
- *It is important to remember that a Thread object does not begin processing until the Start() method has been invoked.*
- After the thread starts, it can be manipulated using the Abort(), Interrupt(), Resume() or Suspend() methods.

Demo 03

C9_3_ThreadSample

C9_3_ThreadSample

- The C9_3_ThreadSample program creates two separate threads from the main program thread, using the Thread constructor (along with the ThreadStart constructor).
- The main program performs a for loop, displaying a counter value every second and using the Sleep() static method of the Thread class.
- The second thread does the same thing but uses a two-second delay; and the third thread uses a three-second delay.
- Again, remember that the threads start running only when the Start() method is used, not when the

Using Threads in a Server

- One of the biggest challenges of network server design is accommodating multiple clients.
- We can use Select() method as previous chapter.

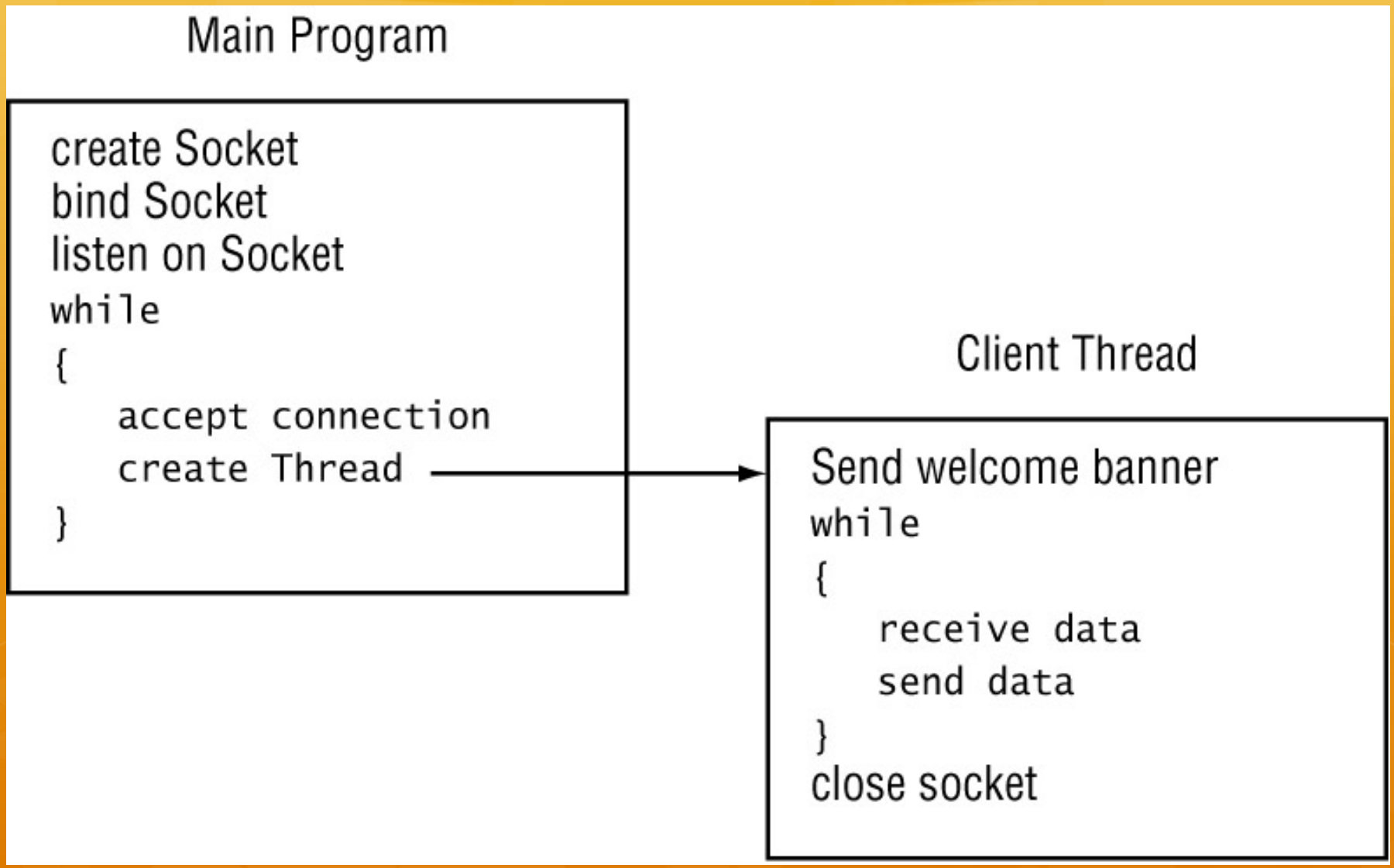
Problem of Select() method

- A problem with using the Select() method to handle multiple clients is that the code can become convoluted and confusing. Trying to accommodate multiple clients as they send and receive data at different intervals can be quite a programming chore.
- When using the Select() method, only one client can be serviced at a time, effectively blocking access to any other clients.
- It would be nice if instead you could split off each client into its own world and deal with its data separately from the rest of the clients.
- Threads allow you to do just that.

Creating a Threaded Server

- The key to creating a threaded server is to map out its handling of each client individually and then extrapolate to hundreds (or thousands) of clients.

Diagram of Threaded Server



Creating a Threaded Server

- The key to the threaded server is to create the main server Socket object in the main program.
- As each client connects to the server, the main program creates a separate Thread object to handle the connection.
- Because the **Socket Accept()** method creates a new Socket object for the connection, the new object is passed to the delegate method and used for all communication with the client from the new thread.
- The original Socket object is then free to accept more client connections.

Demo 04

C9_4_ThreadedTcpSrvr

Demo 04 (cont.)

```
public ThreadedTcpSrvr()
{
    client = new TcpListener(IPAddress.Parse("127.0.0.1"), 9050);
    client.Start();
    Console.WriteLine("Waiting for clients...");
    while (true)
    {
        while (!client.Pending())
        {
            Thread.Sleep(1000);
        }
        ConnectionThread newconnection = new ConnectionThread();
        newconnection.threadListener = this.client;
        Thread newthread = new Thread(new
             ThreadStart(newconnection.HandleConnection));
        newthread.Start();
    }
}
```

Test the Server

- **C9_4_ThreadedTcpSrvr**
- **C8_x_AsyncTcpClient (any TCP Client)**

- Each client program can operate completely independently. You can send messages from clients at will, receiving the correct message back for each client.
- Also, each client can terminate the connection without affecting the other clients.

Take a Rest

A Common Problem

- The last demo program is a consistent model was defined so that both the client and server knew when to expect data to be received and when they should send data.
- However, this is not always the case in network programs.

Example

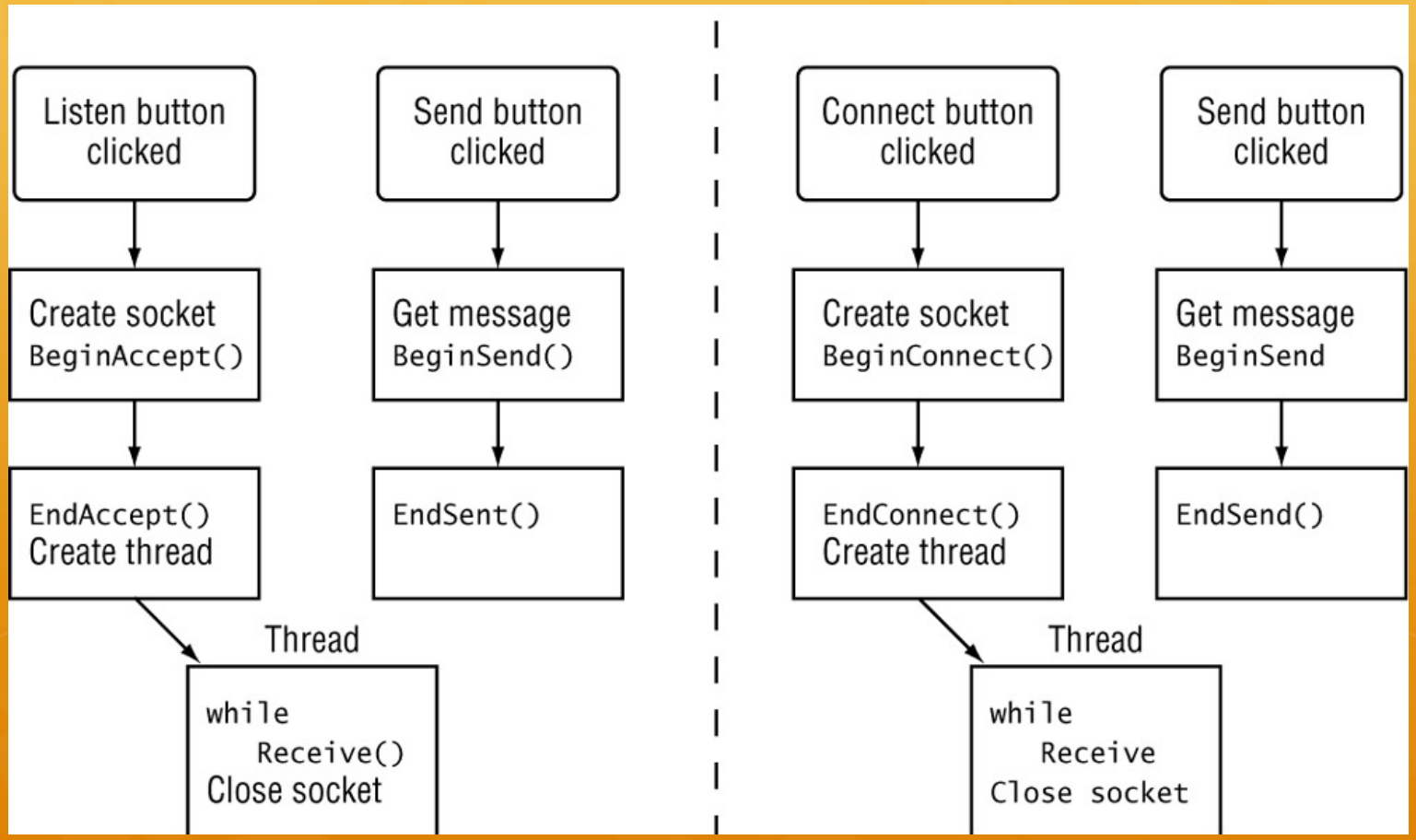
- An example is the common chat program, which allows two people to communicate via a message-board system, where text entered on one system is displayed on the remote system, and vice-versa.
- The problem is that either person can decide to send a message at any time during the connection—this would cause havoc on a network protocol model.
- How could we do this???

Solution

- Using Multi-Thread.

- A secondary thread can be created apart from the main program and used to block on a Receive() method, so any incoming data at any time is displayed on the message board.

Diagram


Listen button clicked
Create socket
BeginAccept()
EndAccept()
Create thread
Thread
while
Receive()
Close socket
Send button clicked
Get message
BeginSend()
EndSent()
Connect button clicked
Create socket
BeginConnect()
EndConnect()
Create thread
Thread
while
Receive
Close socket
Send button clicked
Get message
BeginSend
EndSend()

The Server

- The chat participant that chooses to be the chat server places the program in a listen mode, waiting to accept a new chat client.
- When the client connects, the server creates a separate thread to wait for any incoming data.

The Client

- When the chat client connects to the server, it too creates a separate thread to wait for incoming data from the server.
- The secondary thread is always listening for incoming data.
- The primary thread has to worry only about when the local person wants to send a message to the remote host.

Demo 05

C9_5_TcpChat

Describe

- The program must be used as both the client and server, hence, it must be multifunctional. The main program creates the Windows Form, which includes the following objects:
  - A Listen button for placing the program in listen mode.
  - A Connect button for connecting to a remote chat partner.
  - A Send button for sending messages entered into a TextBox item to the remote partner.
  - A ListBox object for displaying status information and messages received from the remote partner.

Describe (cont.)

```
1 reference
void ButtonListenOnClick(object obj, EventArgs ea)
{
    lbxReceive.Items.Add("Listening for a client...");
    Socket newsock = new Socket(AddressFamily.InterNetwork, SocketType.Stream,
     ProtocolType.Tcp);
    IPEndPoint iep = new IPEndPoint(IPAddress.Any, 9050);
    newsock.Bind(iep);
    newsock.Listen(5);
    newsock.BeginAccept(new AsyncCallback(AcceptConn), newsock);
}
1 reference
```

- When the other chat program connects, the AcceptConn() method creates a separate socket for the client, passing it off to the secondary thread.

Describe (cont.)

- If the customer clicks the Connect button, a socket is created and attempts to connect to the remote host, using the BeginConnect() method.
- After the remote host accepts the connection, the socket is passed off to the secondary thread.
- The sole purpose of the secondary thread is to wait for incoming data, and post it to the list box.

Describe (cont.)

```
{
    int recv;
    string stringData;
    while (true)
    {
        recv = clientSocket.Receive(data);
        stringData = Encoding.ASCII.GetString(data, 0, recv);
        if (stringData == "bye")
            break;
        receiveAddItem(stringData);
    }
    stringData = "bye";
    byte[] message = Encoding.ASCII.GetBytes(stringData);
    clientSocket.Send(message);
    clientSocket.Close();
    receiveAddItem("Connection stopped");
    return;
```

Describe (cont.)

- Because the secondary thread is separate from the main program thread, it can use standard blocking network methods such as **Receive();** the secondary thread won't impede the operation of the Windows Form.

Thread Pools

Thread Pools

- The Windows OS allows you to maintain a pool of **"prebuilt"** threads.
- This *thread pool* supplies worker threads for assignment to specific methods in the application as needed.
- One thread controls the operation of the thread pool, and the application can assign additional threads to processes.

Thread Pools

- By default, there are 25 threads per processor in the thread pool, so this method does not work well for large-scale applications.
- Note:
- You access the thread pool via the .NET **ThreadPool** class.

The ThreadPool Class

- The System.Threading namespace contains the ThreadPool class, which assigns delegates to threads in the thread pool.
- All of the methods in the ThreadPool class are static, so no constructor is ever used to make an instance of a ThreadPool object.
- Instead, once a ThreadPool method is called, the operating system automatically creates a thread pool, and the static ThreadPool methods are used to manipulate the worker threads within the pool.

The ThreadPool Methods

Method	Description
BindHandle()	Binds an operating system handle to the thread pool
GetAvailableThreads()	Gets the number of worker threads available for use in the thread pool
GetMaxThreads()	Gets the maximum number of worker threads available in the thread pool
QueueUserWorkItem()	Queues a user delegate to the thread pool
RegisterWaitForSingleObject()	Registers a delegate waiting for a WaitHandle object
UnsafeQueueUserWorkItem()	Queues an unsafe user delegate to the thread pool but does not propagate the calling stack onto the worker thread
UnsafeRegisterWaitForSingleObject()	Registers an unsafe delegate waiting for a WaitHandle object.

Create new ThreadPool

- To register a delegate for use in a thread pool's thread, use the following format:
  - ThreadPool.QueueUserWorkItem(new WaitCallback(Counter));
- The *Counter* parameter is a delegate for the method run in the thread.

Threadpool Queue

- Unlike Thread objects, once the delegate is placed in the thread pool queue, it will be processed, no other methods are required to start it.
- **Warning:** When the main program thread exits, all thread-pool threads are aborted. The main thread does not wait for them to finish.

Demo 06

C9_6_ThreadPoolSample

Demo 06 (cont.)

- The main program places two method calls in the thread pool queue and then proceeds to perform its own function, counting from zero to nine, with a one-second delay between counts.
- Each of the separate methods performs the same task but with a different delay interval.

Are you tired ???

Thread Pools in a Server

- For servers that service only a few clients (less than the thread pool maximum of 25 threads), the ThreadPool class can quickly service clients in separate threads *without* the overhead of creating separate Thread objects.
- Each client can be assigned to the user work item and processed by an available thread pool's thread.

What should remember?

- Keep in mind that the main program must remain active for as long as each of the ThreadPool objects is active, or the connection will be terminated.
- Once the main program is terminated, each of the thread pool's threads will also terminate, possibly killing an active connection and sending the client into a tailspin—but of course, you will have programmed the client to accommodate such an unruly situation.

Demo 07

C9_7_ThreadPoolTcpSrvr

Describe

```
private TcpListener serverSocket;
1 reference
public ThreadPoolTcpSrvr()
{
    serverSocket = new TcpListener(IPAddress.Parse("127.0.0.1"), 9050);
    serverSocket.Start();
    Console.WriteLine("Waiting for clients...");
    while (true)
    {
        while (!serverSocket.Pending())
        {
            Thread.Sleep(1000);
        }
        ConnectionThread newconnection = new ConnectionThread();
        newconnection.threadListener = this.serverSocket;
        ThreadPool.QueueUserWorkItem(new
             WaitCallback(newconnection.HandleConnection));
    }
}
```

Describe (cont. 1)

- Similar to all of the other TCP server programs, the **C9_7_ThreadPoolTcpSrvr** program creates a socket to listen for new connections (In this program, the *TcpListener* class creates the socket and listen for incoming connections).
- When a connection is detected, a new instance of the ConnectionThread class is created, and the TcpListener object is copied to the class to continue the communication.
- The HandleConnection() method is passed to the QueueUserWorkItem() method to start a new ThreadPool thread with the new connection.

Describe (cont. 2)

- Each client creates a new socket, which is passed off to a new user work item in the thread pool. The user work item is then assigned to a new thread and processed as normal.
- Each client can disconnect without affecting the session of any of the other clients.

Describe (cont. 3)

- When the connection is terminated, the socket is closed and the thread-pool thread terminates.
- The main program is in an endless loop waiting for new clients, there is no risk of terminating active connections unless the main program is manually aborted.

Describe (cont. 4)

- Because the server is running in an endless loop waiting for new clients, you must manually stop it by pressing Ctrl-C.
- If you do this (pressing Ctrl-C) while any connections are active, the connections will abort and the clients will receive an Exception.

Summary

Summary

- Each process comprises one or more threads, which are individual flows of operation within a program.
- Many programs only have one flow of operation, from the start of the program to the end. These programs contain only one thread.
- Others may create secondary threads to process information in a background mode while the main application continues in the foreground.

Summary (cont. 1)

- C# network programs can create secondary threads to process multiple client connections.
- As each new connection attempt is received, it can be passed off to a new thread.
- Each thread is independent of the main program, which continues to wait for new connection attempts.
- The new threads can then process the incoming data from the client.

Summary (cont. 2)

- Although creating new threads is useful, it also demands extensive CPU resources.
- Creating many new threads may cause a significant performance problem as the overhead of creating the new threads is processed.
- An alternative is to use the ThreadPool object, which lets you use threads from the system thread pool to process new client connections.
- These threads have already been created and are waiting to be assigned something to do.
- This technique lessens the overhead of creating new threads for each new client.

THANK YOU!

End Chapter 09