Memory

- Each process has its own virtual address space
  - 32-bit Windows - 4 gigabytes, 64-bit - 8 terabytes
  - the memory of a process is protected against other processes
  - the system uses a page map for mapping virtual address space into addresses of physical memory
- The paging file
  - it is a way for increasing amount of available physical memory
  - pages of memory can be moved from a file to memory and vice versa
  - management of memory pages is invisible for processes

Heap and Virtual Memory

- Using virtual memory
  - `VirtualAlloc(), VirtualFree()`
  - `VirtualLock(), VirtualUnlock()`
  - `VirtualProtect(), VirtualProtectEx()`
- Using a private heap – one or more pages of memory in an address space of a process
  - `HeapCreate(), HeapDestroy()`
  - `HeapAlloc(), HeapReAlloc(), HeapFree(), HeapSize(), HeapValidate()`
- Compatibility with 16-bit versions of Windows:
  - `GlobalAlloc(), GlobalLock(), GlobalReAlloc(), GlobalFree()`
  - `LocalAlloc(), LocalLock(), LocalReAlloc(), LocalFree()`

.NET Garbage Collection

- The system will automatically detect when allocated object is no longer being used and will free it
- Destructor (the `Finalize()` method)
- `IDisposable.Dispose()`
- The `GC` class
  - `Collect()` – forces the garbage collection
  - `GetTotalMemory()` – number of bytes currently thought to be allocated
  - `SuppressFinalize()` – the system will not call the destructor for the specified object
  - `AddMemoryPressure()`, `RemoveMemoryPressure()` `[2.0]`

64-bit Applications

- WOW64 is the x86 emulator that allows 32-bit Windows-based applications to run seamlessly on 64-bit Windows
  - 64-bit Windows does not support running 16-bit Windows-based applications
- Migrating C/C++ code to 64-bit environment
  - use ULONG_PTR type instead of ULONG for addresses
- .NET Framework automatically installs also its 32-bit version on 64-bit systems
  - it allows to run both 32-bit and 64-bit assemblies on 64-bit systems
- The target platform of the application (both native and managed) must be specified before compilation

Processes and Threads

- The application – can use many processes
- The process – an executable program
  - it has all resources necessary to run
  - it has a virtual address space
  - it has the executable code, data, and objects' handles
  - it starts with one thread (the primary thread)
  - it can create many threads
- The thread – a basic unit which can get a slice of the processor's time
  - each thread has its own management of exceptions, priority and a set of structures to remember its context
  - all threads of the process share the address space and system resources of the process

Multitasking

- Multitasking in Windows
  - preemptive multitasking – each thread receives processor's time (about 20 ms)
  - ready for computers with more than one processor
- Advantages
  - applications can work simultaneously
  - parallel tasks of one application can work simultaneously
- Examples of usage:
  - calculations done in the background
  - parallel executing of many tasks (e.g. server's clients)
  - getting input from many devices
  - prioritizing of tasks
- Guideline: use the smallest possible number of thread

Scheduling

- The system controls multitasking by choosing a waiting thread which will get the next slice of processor's time
- Priorities
  - a class of a thread
  - priority of a thread in the class
- Switching context
  - context of stopped thread is stored
  - the first waiting thread starts
- Thread's priority can be changed dynamically
- For computers with many processors, it can be specified which processor will execute the thread

Multithreading

- Creating
  - `CreateThread()`, `CreateRemoteThread()`
- Features
  - the handle - `OpenThread()`, `GetCurrentThread()`
  - the identifier - `GetCurrentThreadId()`
- Sleeping
  - `SuspendThread()`, `ResumeThread()`, `Sleep()`, `SleepEx()`
- Thread Local Storage
  - independent data
- Ending
  - return from thread's function
  - `ExitThread()`, `ExitProcess()`
  - `TerminateThread()`, `TerminateProcess()`

Using Threads in .NET

- Creating
  1. Create the **`Thread`** object (pass **`ThreadStart`** or **`ParametrizedThreadStart`** delegate as a parameter)
  2. Call the **`Start()`** method (will return immediately, check the **`IsAlive`** or **`ThreadState`** properties to determine the state of a thread)
- Pausing and resuming
  - the **`Sleep()`** method (pass a number of milliseconds or the **`Timeout.Infinite`** value)
  - the **`Interrupt()`** method (if a target thread is blocked, **`ThreadInterruptedException`** will be thrown in it)
  - the **`Suspend()`** and **`Resume()`** methods are obsolete

Using Threads in .NET cont.

- Destroying
  - the **`Abort()`** method – the target thread will be stopped permanently
    - **`ThreadAbortException`** is thrown in the target thread
    - if the target thread calls **`ResetAbort()`** method, aborting is cancelled
  - call the **`Join()`** method to wait until the thread has ended
- Priorities
  - the **`Priority`** property (the default value: **`ThreadPriority.Normal`**)

Child Processes

- Creating
  - `CreateProcess()`
- Features
  - the handle - `OpenProcess()`, `GetCurrentProcess()`
  - the identifier - `GetCurrentProcessId()`
- Inheritance
  - handles opened by `CreateFile()` and other functions for creating processes, threads and synchronization objects
  - environment variables
  - the working directory
- Ending
  - `ExitProcess()`, `TerminateProcess()`
  - `GetExitCodeProcess()`

Using Processes in .NET

- The `Process` class
  - methods:
    - `GetCurrentProcess()`, `GetProcessById()`
    - `GetProcesses()`, `GetProcessesByName()`
    - `Start()`
    - `Close()`, `CloseMainWindow()`
    - `Kill()`
  - events: `Disposed()`, `Exited()`
  - properties:
    - `StartInfo (Arguments, FileName, UserName, Password, WorkingDirectory)`
    - `ExitCode`
    - `Id`, `ProcessName`
    - `MainWindowHandle`, `PriorityClass`

Synchronization

- The problem of parallel access to the same data
- Synchronization
  - synchronization objects can be used in one of waiting functions
  - the state of synchronization object can be *signaled* or *nonsignaled*
  - waiting functions stop the thread until synchronization object is *signaled*

Waiting Functions

- For one synchronization object
  - waiting for *signaled* state of the object or timeout
  - `SignalObjectAndWait()`
  - `WaitForSingleObject(), WaitForSingleObjectEx()`
- For many synchronization objects
  - either waiting for the *signaled* state of all objects or only one
  - timeout value can be specified
  - `WaitForMultipleObjects(), WaitForMultipleObjectsEx()`
  - `MsgWaitForMultipleObjects(), MsgWaitForMultipleObjectsEx()`

Synchronization Objects

- Objects dedicated for synchronization:
  - *event* – a notification about an event
  - *mutex* – a mutual exclusion
  - *critical section* – like a mutex, but only for threads of one process
  - *semaphore* – maximum allowed number of threads
  - *waitable timer* – signaled after the specified time
- Other objects which can be used for synchronization
  - *change notification* – change in a directory
  - *console input* – something in an input buffer
  - *job* – the end of all processes from a group
  - *memory resource notification* – change in a memory
  - process – the end of executing a process
  - thread – the end of executing a thread

Synchronization in .NET

- Locking
  - the `lock` keyword in C#
  - the `Monitor` class
  - the `Mutex` class (local or global – visible throughout the operating system)
  - the `ReaderWriterLock` class (an exclusive access for writers, shared for readers)
  - the `Semaphore` class (used to control access to a pool of resources; can be local or global)
- Signaling
  - the `Join()` method of a thread
  - classes derived from the `WaitHandle` class
  - classes: `EventWaitHandle`, `AutoResetEvent`, `ManualResetEvent`
- `Interlocked` methods: `Increment()`, `Decrement()`, `Exchange()`, `CompareExchange()`

Multithreading in User Interface

- All threads can create windows
  - `EnumThreadWindows()`
  - `GetWindowThreadProcessId()`
- There must be a message loop in each thread creating a window
  - `PostThreadMessage()`
  - `SendNotifyMessage()`
  - `SendMessageTimeout()`
  - `SendMessageCallback()`
- The default: there is no synchronization in getting input data
  - `AttachThreadInput()`

GUI Multithreading in Windows Forms

- **Only the main thread can call methods and modify properties of user interface elements**
- Thread-safe calls

```
void Test() {
    Thread thread = new Thread(new
        ThreadStart(MyThreadProc));
    thread.Start();
}

void MyThreadProc() {
  //textBox1.Text = "something";  //WRONG
  SetTextCallback d = new SetTextCallback(SetText);
  Invoke(d, new object[] {"something"});
}
```

  - use the `BackgroundWorker` component

GUI Multithreading in WPF

- WPF applications start with two threads: one for handling rendering and another for managing the UI
  - the rendering thread effectively runs hidden in the background while the UI thread receives input, handles events, and runs application code
  - most applications use a single UI thread
- It is acceptable for one Thread/Dispatcher combination to manage multiple windows, but sometimes several threads do a better job
  - this is especially true if there is any chance that one of the windows will monopolize the thread

GUI Multithreading in WPF cont.

- In general, objects in WPF can only be accessed from the thread that created them
  - it is not generally possible to create an object on one thread, and access it from another (**InvalidOperationException**)
  - frozen objects become read-only can be used on any thread at any time
- The UI thread queues work items inside a **Dispatcher**
  - the Dispatcher selects work items on a priority basis and runs each one to completion
  - every UI thread must have at least one Dispatcher, and each Dispatcher can execute work items in exactly one thread
  - most classes in WPF derive from **DispatcherObject** which stores a reference to the Dispatcher linked to the currently running thread

GUI Multithreading in WPF cont.

- The **Dispatcher** class provides some useful methods:
  - **CheckAccess** – checks if the calling thread has access to the object
  - **VerifyAccess** – as above, but throws **InvalidOperationException** in case of no access
  - **Invoke** – schedules a delegate for execution; it doesn't return until the UI thread actually finishes executing the delegate
  - **BeginInvoke** – as above, but is asynchronous (i.e. it returns immediately)

Interprocess Communication

- Win32 API:
  - clipboard
  - COM - Component Object Model
  - Data Copy - WM_COPYDATA
  - DDE - Dynamic Data Exchange
  - File Mapping, Name Shared Memory
  - Mailslots – unidirectional communication
  - Pipes – bidirectional communication
  - RPC - Remote Procedure Call
  - Windows Sockets
- .NET Framework:
  - .NET Remoting
  - WCF - Windows Communication Foundation [3.0+]

Directories

- Operations
  - `GetCurrentDirectory()`, `SetCurrentDirectory()` – for a process
  - `CreateDirectory()`, `CreateDirectoryEx()`
  - `RemoveDirectory()`
  - `MoveFileEx()`, `MoveFileWithProgress()`
- Files enumeration
  - `FindFirstFile()`, `FindNextFile()`, `FindClose()`
- Change notification
  - `FindFirstChangeNotification()`, `FindNextChangeNotification()`, `FindCloseChangeNotification()`

Operations on Files

- Operations
  - `CreateFile()` – to open a file (including optional creation)
  - `CloseHandle()`
  - `DeleteFile()`
  - `GetShortPathName()`, `GetFullPathName()`
  - `GetTempFileName()`, `GetTempPath()`
  - `CopyFile()`, `CopyFileEx()`, `ReplaceFile()`
  - `MoveFile()`, `MoveFileEx()`, `MoveFileWithProgress()`
  - `LockFile()`, `LockFileEx()`, `UnlockFile()`, `UnlockFileEx()`

Reading and Writing Files

- Reading
  - `ReadFile(), ReadFileEx()`
- Writing
  - `WriteFile(), WriteFileEx()`
- Setting the current position in a file
  - `SetFilePointer()`
  - `SetEndOfFile()`
- Flushing file's buffers
  - `FlushFileBuffers()`

Files Properties

- Security
  - `SetSecurityInfo(), SetNamedSecurityInfo()`
- Attributes
  - `GetFileAttributes(), SetFileAttributes()`
- Size
  - `GetFileSize()`
- Time
  - `GetFileTime(), SetFileTime()`

Files Encryption and Compression

- Encryption (NTFS only)
  - `EncryptFile()`
  - `DecryptFile()`
  - `FileEncryptionStatus()`
- Compression
  - `LZinit()`
  - `LZOpenFile(), LZClose()`
  - `LZCopy()`
  - `LZRead(), LZSeek()`

`System.IO` namespace in .NET

- Operations and information
  - `FileInfo` (static methods), `File` (instance methods)
  - `DirectoryInfo`, `Directory`
  - `DriveInfo` [2.0]
  - `FileSystemWatcher`
- Streams
  - `Stream`, `BufferedStream`, `FileStream`, `MemoryStream`, `UnmanagedMemoryStream`
- Readers and writers
  - `StreamReader`, `StreamWriter`, `BinaryReader`, `BinaryWriter`, `StringReader`, `StringWriter`, `TextReader`, `TextWriter`
- Useful tools
  - `Path`

Directory Listing

```
public static void Main(String[] args) {
  string path = ".";
  if (args.Length > 0) {
    if (File.Exists(args[0])) {
      path = args[0];
    } else {
      Console.WriteLine("{0} not found; using"+
        "current directory:", args[0]);
    }

    DirectoryInfo dir = new DirectoryInfo(path);
    foreach (FileInfo f in dir.GetFiles("*.exe")) {
      String name = f. Name;
      long size = f.Length;
      DateTime creationTime = f.CreationTime;
      Console.WriteLine("{0,-12:N0} {1,-20:g} {2}",
        size, creationTime, name);
    }
  }
}
```

Reading and Writing Binary Data

```
private const string FILE_NAME = "Test.bin";
public static void Main(String[] args) {
  if (File.Exists(FILE_NAME)) {
    Console.WriteLine("{0} exists!", FILE_NAME);
    return;
  }
  FileStream fs = new FileStream(FILE_NAME,
    FileMode.CreateNew);
  BinaryWriter w = new BinaryWriter(fs);
  for (int i = 0; i < 11; i++) {
    w.Write(i);
  }
  w.Close();
  fs.Close();

  fs = new FileStream(FILE_NAME, FileMode.Open,
    FileAccess.Read);
  BinaryReader r = new BinaryReader(fs);
  for (int i = 0; i < 11; i++) {
    Console.WriteLine(r.ReadInt32());
  }
  r.Close();
  fs.Close();
}
```

Reading and Writing Text

```
private const string FILE_NAME = "Test.txt";
public static void Main(String[] args) {
  if (File.Exists(FILE_NAME)) {
    Console.WriteLine("{0} exists!", FILE_NAME);
    return;
  }

  using (StreamWriter sw = File.CreateText(FILE_NAME)) {
    sw.WriteLine("This is my file.");
    sw.WriteLine("Integer {0} double {1}", 1, 4.2);
    sw.Close();
  }

  using (StreamReader sr = File.OpenText(FILE_NAME)) {
    String input;
    while ((input = sr.ReadLine())!=null) {
      Console.WriteLine(input);
    }
    Console.WriteLine ("The end of the stream.");
    sr.Close();
  }
}
```

Append Text

```
private const string FILE_NAME = "Test.txt";
public static void Main(String[] args) {
  using (StreamWriter sw = File.AppendText(FILE_NAME)) {
    sw.Write("\r\nLog Entry : ");
    sw.WriteLine("{0} {1}",
                 DateTime.Now.ToLongTimeString(),
                 DateTime.Now.ToLongDateString());
    sw.WriteLine("  :");
    sw.WriteLine("  :{0}", logMessage);
    sw.WriteLine("-------------------------------");

    sw.Flush();
    sw.Close();
  }
}
```

Using StringReader and StringWriter

```
public static void Main(String[] args) {
  StringBuilder sb = new StringBuilder(
                              "Some number of characters");
  char[] b = {' ','t','o',' ','w','r','i','t','e',
    ' ','t','o','.'};
  StringWriter sw = new StringWriter(sb);
  sw.Write(b, 0, 3);
  Console.WriteLine(sb);
  sw.Close();

  String str = "Some number of characters";
  char[] b = new char[24];
  StringReader sr = new StringReader(str);
  sr.Read(b, 0, 13);
  Console.WriteLine(b);
  sr.Close();
}
```

Isolated Storage

- The data isolated per user and per assembly
  - credentials determine the assembly identity
- An application saves the data to a unique data compartment
  - the data compartment consists of one or more isolated storage files which contain the actual directory locations where the data is stored
  - a location is transparent for the developer – usually on the client, sometimes on the server
    - default for Windows XP: <SYSTEMDRIVE>\Documents and Settings \<user>\Application Data \<user>\Local Settings\Application Data
- Possibilities for administrators:
  - set the trust level
  - limit the size
  - remove all user's persisted data

Screen Saver in Win32 API

- An executable file (.exe or .scr) with strictly specified elements:
  - linked library: scrnsave.lib (ANSI) or scrnsavw.lib (Unicode)
  - **`ScreenSaverProc()`** – a function exported from the module
    - it processes all messages
    - all unhandled messages should be passed to the **`DefScreenSaverProc()`** function
  - **`ScreenSaverConfigureDialog()`** – a function exported from the module
    - it displays a dialog box with the screen saver's configuration
  - **`RegisterDialogClasses()`**
    - it registers custom windows classes or returns **`true`**
  - an icon with number **`ID_APP`** (from Scrnsave.h)
  - a description string with number 1

Screen Saver in .NET

- Just an executable file with .scr extension placed in the \Windows\system32 directory which handles the following command line arguments:
  - /c – show the Settings dialog box, modal to the foreground window
  - /p <HWND> – preview the screen saver as a child of a window with the <HWND> handle
  - /a <HWND> – change password, modal to window <HWND>
  - /s – run the screen saver
  - no parameter – show the Settings dialog box