Pamięć

- Każdy proces ma własną wirtualną przestrzeń adresową
  - 32-bitowe Windows - 4 gigabajty, 64-bitowe - 8 terabajtów
  - ochrona pamięci pomiędzy procesami
  - system używa *page map* do przemapowania wirtualnej przestrzeni procesu na adresy pamięci fizycznej
- *Paging file*
  - sposób na zwiększenie wielkości dostępnej pamięci fizycznej
  - strony pamięci mogą być przenoszone pomiędzy plikiem i pamięcią
  - zarządzanie stronami pamięci jest niewidoczne dla procesów

Sterta i pamięć wirtualna

- Obsługa pamięci wirtualnej
  - `VirtualAlloc(), VirtualFree()`
  - `VirtualLock(), VirtualUnlock()`
  - `VirtualProtect(), VirtualProtectEx()`
- Obsługa prywatnej sterty - bloku jednej lub wielu stron pamięci w przestrzeni adresowej procesu
  - `HeapCreate(), HeapDestroy()`
  - `HeapAlloc(), HeapReAlloc(), HeapFree(), HeapSize(), HeapValidate()`
- Kompatybilność z 16-bitowym Windows:
  - `GlobalAlloc(), GlobalLock(), GlobalReAlloc(), GlobalFree()`
  - `LocalAlloc(), LocalLock(), LocalReAlloc(), LocalFree()`

.NET Garbage Collection

- System automatycznie rozpoznaje obiekty, które nie są już używane i zwalnia je
- Destruktor (metoda `Finalize()`)
- `IDisposable.Dispose()`
- Klasa `GC`
  - `Collect()` – wymuszenie sprawdzenia obiektów i ew. ich zwolnienia
  - `GetTotalMemory()` – zajętość pamięci
  - `SuppressFinalize()` – ustawienie, że dla obiektu podanego jako parametr destruktor nie będzie wołany
  - `AddMemoryPressure()`, `RemoveMemoryPressure()` `[2.0]`

Aplikacje 64-bitowe

- WOW64 jest emulatorem x86 pozwalającym na uruchamianie 32-bitowych aplikacji w systemie 64-bitowym
  - 64-bitowe wersje Windows nie pozwalają na uruchamianie 16-bitowych aplikacji
- Przeniesienie kodu C/C++ do 64-bitowego środowiska
  - dla adresów pamięci używać typu ULONG_PTR zamiast ULONG
- .NET Framework automatycznie instaluje także swoją 32-bitową wersję na systemach 64-bitowych
  - to pozwala na uruchamianie zarówno 32 jak i 64-biwotych *assemblies* na systemie 64-bitowym
- Docelowa platforma (x86 lub x64) musi być określona przed kompilacją
  - dotyczy to zarówno natywnych jak i zarządzanych aplikacji

Procesy i wątki

- Aplikacja - może zawierać wiele procesów
- Proces - wykonywany program
  - dostarcza zasobów niezbędnych do jego wykonania
  - ma wirtualną przestrzeń adresową
  - ma kod wykonywalny, dane, uchwyty do obiektów
  - jest uruchamiany z jednym wątkiem - *primary thread*
  - może tworzyć wiele wątków
- Wątek - podstawowa jednostka, której system przydziela czas procesora
  - każdy wątek ma własną obsługę wyjątków, priorytet i zbiór struktur umożliwiających zapamiętanie kontekstu wątku
  - wątki procesu dzielą jego przestrzeń adresową i zasoby systemowe

Wielozadaniowość *(multitasking)*

- Zasada działania
  - *preemptive multitasking* - każdy wątek otrzymuje czas procesora (około 20 milisekund)
  - obsługa wielu procesorów
- Zalety
  - równoległa praca z wieloma aplikacjami w danej chwili
  - równoległe wykonywanie wielu czynności przez aplikację
- Przykłady zastosowań:
  - obliczenia w tle
  - równoczesna obsługa wielu zadań (np. klientów serwera)
  - obsługa wejścia z wielu urządzeń
  - szeregowanie zadań wg ważności (priotytety)
- Używać tak mało wątków, jak to tylko możliwe

Planowanie *(scheduling)*

- System kontroluje wielozadaniowość poprzez określenie, który z oczekujących wątków otrzyma następny czas procesora
- Priotytety
  - klasa wątku
  - priorytet wątku w klasie
- Zmiana kontekstu
  - zapamiętanie kontekstu zatrzymanego wątku
  - uruchomienie oczekującego wątku (z uwzględnieniem priorytetów)
- Dynamiczna zmiana priotytetu
- Wiele procesorów - możliwość określenia, na których procesorach wątek może być wykonywany

Wielowątkowość *(multithreading)*

- Tworzenie
  - `CreateThread()`, `CreateRemoteThread()`
- Cechy
  - uchwyt - `OpenThread()`, `GetCurrentThread()`
  - identyfikator - `GetCurrentThreadId()`
- Usypianie
  - `SuspendThread()`, `ResumeThread()`, `Sleep()`, `SleepEx()`
- *Thread Local Storage*
  - dane niezależne dla każdego wątku
- Zakończenie
  - zakończenie funkcji wątku
  - `ExitThread()`, `ExitProcess()`
  - `TerminateThread()`, `TerminateProcess()`

Użycie wątków w .NET

- Tworzenie
  1. Utworzyć obiekt **`Thread`** (delegat **`ThreadStart`** lub **`ParametrizedThreadStart`** jako parametr)
  2. Wywołać metodę **`Start()`** (wróci natychmiast, przy pomocy właściwości **`IsAlive`** lub **`ThreadState`** można sprawdzić status nowego wątku)
- Zatrzymanie i ponowne uruchomienie
  - metoda **`Sleep()`** (liczba milisekund lub wartość **`Timeout.Infinite`** jako parametr)
  - metoda **`Interrupt()`** (jeśli wątek jest zablokowany zostanie wyrzucony wyjątek **`ThreadInterruptedException`**)
  - **`Suspend()`**, **`Resume()`** – stare i oznaczone jako do usunięcia

Użycie wątków w .NET c.d.

- Niszczenie
  - `Abort()`
    - `ThreadAbortException` wyrzucany w docelowym wątlku
    - wątek może odmówić zatrzymania - `ResetAbort()`
  - oczekiwanie na zakończenie wątku - `Join()`
- Priorytety
  - właściwość `Priority` (domyślna wartość: `ThreadPriority.Normal`)

Procesy potomne

- Tworzenie
  - `CreateProcess()`
- Cechy
  - uchwyt - `OpenProcess()`, `GetCurrentProcess()`
  - identyfkator - `GetCurrentProcessId()`
- Dziedziczenie
  - uchwyty otwarte przez `CreateFile()` i inne funkcje tworzące procesy, wątki i obiekty synchronizacji
  - zmienne środowiskowe
  - aktualny katalog
- Zakończenie
  - `ExitProcess()`, `TerminateProcess()`
  - `GetExitCodeProcess()`

Użycie procesów w .NET

- Klasa `Process`
  - metody:
    - `GetCurrentProcess()`, `GetProcessById()`
    - `GetProcesses()`, `GetProcessesByName()`
    - `Start()`
    - `Close()`, `CloseMainWindow()`
    - `Kill()`
  - zdarzenia: `Disposed()`, `Exited()`
  - właściwości:
    - `StartInfo (Arguments, FileName, UserName, Password, WorkingDirectory)`
    - `ExitCode`
    - `Id`, `ProcessName`
    - `MainWindowHandle`, `PriorityClass`

Synchronizacja

- Problem równoległego dostępu do danych
- Synchronizacja dostępu do zasobu
  - użycie obiektów synchronizacji w jednej z funkcji oczekujących
  - stan obiektu synchronizacji może być *signaled* lub *nonsignaled*
  - funkcje sygnalizacji blokują wykonanie wątku do chwili zasygnalizowania obiektu

Funkcje oczekiwania

- Jeden obiekt synchronizacji
  - oczekiwanie do chwili zasygnalizowania obiektu lub upłynięcie wskazanego czasu oczekiwania
  - `SignalObjectAndWait()`
  - `WaitForSingleObject()`, `WaitForSingleObjectEx()`
- Wiele obiektów synchronizacji
  - określenie, czy oczekiwanie na jeden obiekt z podanych, czy na wszystkie
  - możliwość podania czasu maksymalnego oczekiwania
  - `WaitForMultipleObjects()`, `WaitForMultipleObjectsEx()`
  - `MsgWaitForMultipleObjects()`, `MsgWaitForMultipleObjectsEx()`

Obiekty synchronizacji

- Obiekty przeznaczone do synchronizacji:
  - *event* - powiadamia o zdarzeniu
  - *mutex* - pozwala na wyłączny dostęp do dzielonego zasobu
  - *critical section* - jak mutex, ale tylko wątki jednego procesu
  - *semaphore* - jednoczesny dostęp określonej liczby wątków
  - *waitable timer* - sygnalizowany po określonym czasie
- Obiekty, które można wykorzystać do synchronizacji
  - *change notification* - zmiana w katalogu
  - *console input* - niepusty bufor wejściowy
  - *job* - zakończenie wszystkich procesów z grupy
  - *memory resource notification* - zmiana w pamięci
  - proces - koniec wykonania
  - wątek - koniec wykonania

Synchronizacja w .NET

- Ograniczenie dostępu
  - `lock` – słowo kluczowe w C#
  - klasa `Monitor`
  - klasa `Mutex` (lokalny lub globalny – widoczny dla wszystkich procesów w systemie)
  - klasa `ReaderWriterLock` (wyłączny dostęp dla piszących, dzielony dla czytających)
  - klasa `Semaphore` (możliwy równoczesny dostęp wielu wątków; może być lokalny lub globalny)
- Sygnalizacja
  - `Join()` – metoda wątlku
  - klasy dziedziczące z `WaitHandle`
  - klasy: `EventWaitHandle`, `AutoResetEvent`, `ManualResetEvent`
- `Interlocked` metody: `Increment()`, `Decrement()`, `Exchange()`, `CompareExchange()`

Wielowątkowość w interfejsie użytkownika

- Każdy wątek może tworzyć okna
  - `EnumThreadWindows()`
  - `GetWindowThreadProcessId()`
- Każdy wątek tworzący okna musi mieć pętle komunikatów
  - `PostThreadMessage()`
  - `SendNotifyMessage()`
  - `SendMessageTimeout()`
  - `SendMessageCallback()`
- Domyślnie nie ma synchronizacji w obsłudze informacji wejściowych
  - `AttachThreadInput()`

Wielowątkowość GUI w Windows Forms

- **Tylko główny wątek może wywołać metody i modyfikować właściwości elementów interfejsu użytkownika**
- Bezpieczne wywołania

```
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"});
}
```

  - alternatywa: `BackgroundWorker`

Wielowątkowość GUI w WPF

- Aplikacje WPF uruchamiane są z dwoma wątkami – jednym do obsługi odrysowywania zawartości i drugim do zarządzania interfejsem użytkownika
  - wątek odrysowujący jest uruchamiany w tle, natomiast wątek interfejsu użytkownika otrzymuje sygnały z myszy i klawiatury, obsługuje zdarzenia i wykonuje kod aplikacji
  - większość aplikacji używa jednego wątku interfejsu użytkownika
- Pojedyncza para Thread/Dispatcher jest w stanie obsłużyć wiele okien, ale czasami zachodzi konieczność wykorzystania większej liczby wątków
  - w szczególności taka konieczność zachodzi w sytuacji, gdy istnieje niebezpieczeństwo, że jedno okno będzie zabierać większość dostępnego czasu

Wielowątkowość GUI w WPF c.d.

- Obiekty WPF mogą być używane tylko przez wątek, który je utworzył
  - próba użycia obiektu przez wątek, który go nie stworzył spowoduje wyjątek **InvalidOperationException**
  - tylko zamrożone (*frozen*) obiekty mogą być używane przez dowolne wątki (ale nie mogą być modyfikowane)
- Wewnątrz obiektu klasy **Dispatcher** kolejkowane są zadania do wykonania
  - Dispatcher wykonuje zadania wg ich priorytetów
  - w każdym wątku UI musi być przynajmniej jeden obiekt klasy Dispatcher (który może być związany z tylko jednym wątkiem)
  - większość klas WPF dziedziczy z klasy **DispatcherObject**, która trzyma referencję do obiektu Dispatcher należącego do aktualnego wątku

Wielowątkowość GUI w WPF c.d.

- Klasa **Dispatcher** udostępnia przydatne metody:
  - **CheckAccess** – sprawdza, czy wątek wywołujący tę metodę ma dostęp do obiektu
  - **VerifyAccess** – j.w., lecz w przypadku braku dostępu rzuca wyjątek **InvalidOperationException**
  - **Invoke** – zapisuje delegata w kolejce do wykonania; czeka z powrotem do momentu zakończenia jego wykonywania
  - **BeginInvoke** – j.w., lecz kończy działanie bez oczekiwania na wykonanie delegata (działa asynchronicznie)

Komunikacja pomiędzy procesami

- Win32 API:
  - *Clipboard* - schowek
  - *COM - Component Object Model*
  - *Data Copy* - WM_COPYDATA
  - *DDE - Dynamic Data Exchange*
  - *File Mapping, Name Shared Memory*
  - *Mailslots* - jednokierunkowa komunikacja
  - *Pipes* - dwukierunkowa komunikacja
  - *RPC - Remote Procedure Call*
  - *Windows Sockets*
- .NET Framework:
  - .NET Remoting
  - WCF - Windows Communication Foundation [3.0+]

Katalogi

- Operacje
  - `GetCurrentDirectory()`, `SetCurrentDirectory()` - dla procesu
  - `CreateDirectory()`, `CreateDirectoryEx()`
  - `RemoveDirectory()`
  - `MoveFileEx()`, `MoveFileWithProgress()`
- Wyliczanie plików
  - `FindFirstFile()`, `FindNextFile()`, `FindClose()`
- Powiadomienia o zmianach
  - `FindFirstChangeNotification()`, `FindNextChangeNotification()`, `FindCloseChangeNotification()`

Operacje na plikach

- Operacje
  - `CreateFile()` - tworzenie, otwieranie
  - `CloseHandle()`
  - `DeleteFile()`
  - `GetShortPathName()`, `GetFullPathName()`
  - `GetTempFileName()`, `GetTempPath()`
  - `CopyFile()`, `CopyFileEx()`, `ReplaceFile()`
  - `MoveFile()`, `MoveFileEx()`, `MoveFileWithProgress()`
  - `LockFile()`, `LockFileEx()`, `UnlockFile()`, `UnlockFileEx()`

Zapis i odczyt z pliku

- Odczyt
  - `ReadFile(), ReadFileEx()`
- Zapis
  - `WriteFile(), WriteFileEx()`
- Ustawienie aktualnej pozycji w pliku
  - `SetFilePointer()`
  - `SetEndOfFile()`
- Wyczyszczenie buforów pliku
  - `FlushFileBuffers()`

Właściwości plików

- Zabezpieczenia
  - `SetSecurityInfo(), SetNamedSecurityInfo()`
- Atrybuty
  - `GetFileAttributes(), SetFileAttributes()`
- Rozmiar
  - `GetFileSize()`
- Czas
  - `GetFileTime(), SetFileTime()`

Szyfrowanie i kompresja plików

- Szyfrowanie (tylko NTFS)
  - `EncryptFile()`
  - `DecryptFile()`
  - `FileEncryptionStatus()`
- Kompresja
  - `LZinit()`
  - `LZOpenFile(), LZClose()`
  - `LZCopy()`
  - `LZRead(), LZSeek()`

`System.IO` [.NET Framework]

- Operacje i informacje
  - **`FileInfo`** (metody statyczne), **`File`** (metody obiektu)
  - **`DirectoryInfo`**, **`Directory`**
  - **`DriveInfo`** [2.0]
  - **`FileSystemWatcher`**
- Strumienie
  - **`Stream`**, **`BufferedStream`**, **`FileStream`**, **`MemoryStream`**, **`UnmanagedMemoryStream`**
- *Readers, writers*
  - **`StreamReader`**, **`StreamWriter`**, **`BinaryReader`**, **`BinaryWriter`**, **`StringReader`**, **`StringWriter`**, **`TextReader`**, **`TextWriter`**
- Przydatne:
  - **`Path`**

Zawartość katalogu

```
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);
    }
  }
}
```

Odczyt i zapis danych binarnych

```
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();
}
```

Odczyt i zapis tekstu

```
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();
  }
}
```

Dopisywanie tekstu

```
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();
  }
}
```

Użycie StringReader i 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

- Dane izolowane na poziomie użytkownika i *assembly*
- Unikalne miejsce zapisu danych aplikacji
  - złożone z jednego lub wielu plików *isolated* storage
  - fizyczna lokalizacja ustalana przez system – zwykle po stronie klienta, czasami po stronie serwera
    - Windows XP: <SYSTEMDRIVE>\Documents and Settings \<user>\Application Data \<user>\Local Settings\Application Data
- Możliwości administracyjne
  - ustawienie poziom zaufania
  - ograniczenie rozmiaru
  - usunięcie danych użytkownika

Wygaszacz ekranu w Win32 API

- Plik wykonywalny (.exe lub .scr) zawierający ściśle określone elementy:
  - dołączona jedna z bibliotek: scrnsave.lib (ANSI), scrnsavw.lib (Unicode)
  - **`ScreenSaverProc()`** – funkcja eksportowana z modułu
    - obsługuje komunikaty przychodzące z systemu
    `DefScreenSaverProc()` dla nieobsłużonych komunikatów
  - **`ScreenSaverConfigureDialog()`** - eksportowana z modułu
    - wyświetla okno dialogowe konfiguracji wygaszacza
  - **`RegisterDialogClasses()`**
    - rejestracja własnych klas okien lub **`return TRUE`**
  - ikona (numer: stała **`ID_APP`** z Scrnsave.h)
  - tekst z opisem wygaszacza (numer: 1)

Wygaszacz ekranu w .NET

- Zwykły plik wykonywalny z rozszerzeniem .scr umieszczony w katalogu \Windows\system32, obsługujący następujące parametry wywołania:
  - /c – pokazać okno ustawień wygaszacza jako modalne dla aktualnego okna
  - /p <HWND> – wyświetlić podgląd wygaszacza w okienku o uchwycie <HWND>
  - /a <HWND> – zmiana hasła, okienko modalne dla okna <HWND>
  - /s – uruchomić wygaszacz
  - bez parametru – pokazać okno ustawień wygaszacza