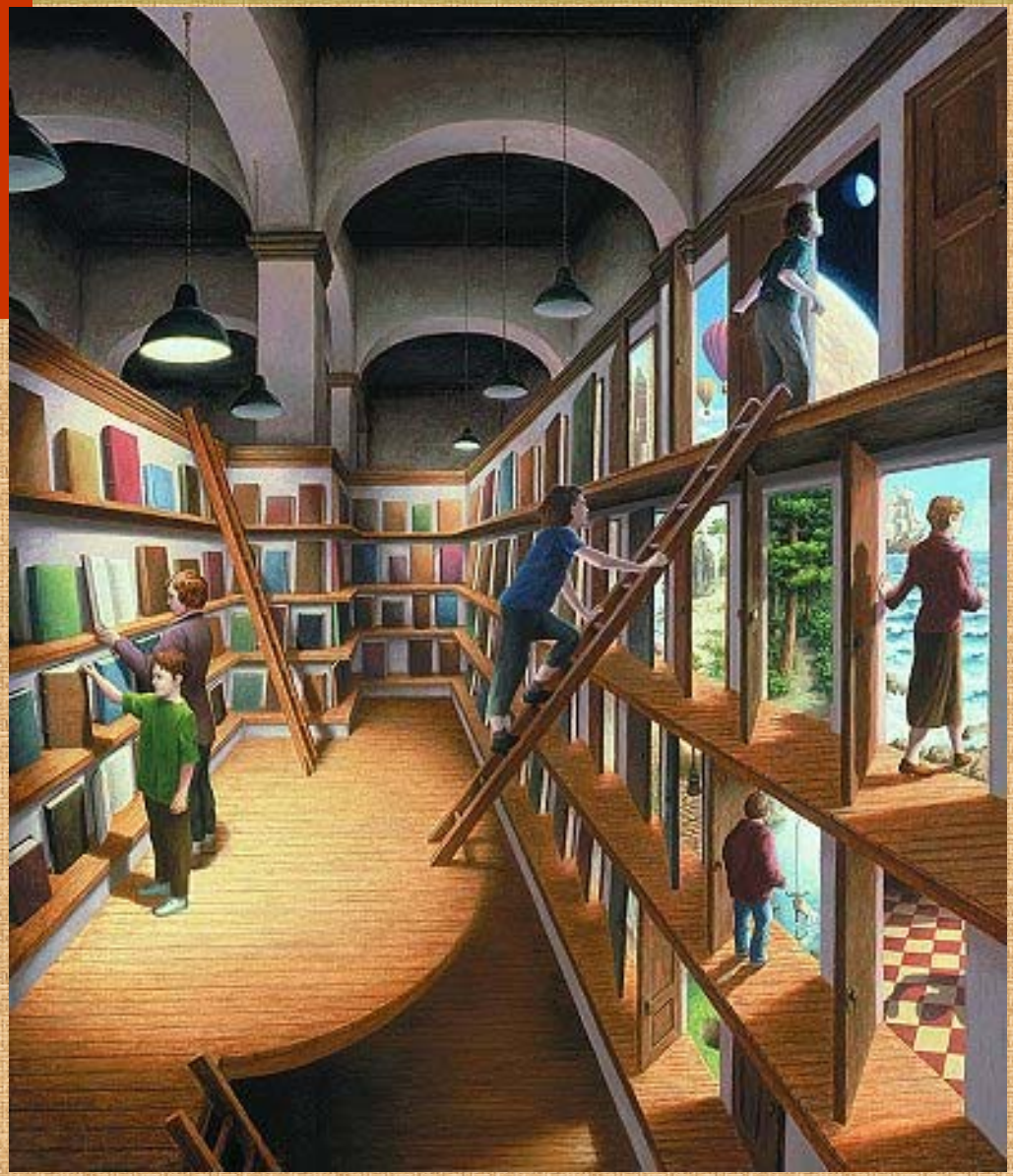
Vývoj aplikací v prostředí .NET

© Katedra řídicí techniky,
ČVUT-FEL Praha

6. přednáška

Omluva

Vzhledem k tomu,
že se předmět přednáší
i v angličtině a není
v mých silách vyrobit
dvojjazyčné prezentace,
nejsou všechny snímky
v češtině.
Děkuji za pochopení.



Rob Gonsalves

Services of OS in DLL Dynamic Link Libraries

Libraries

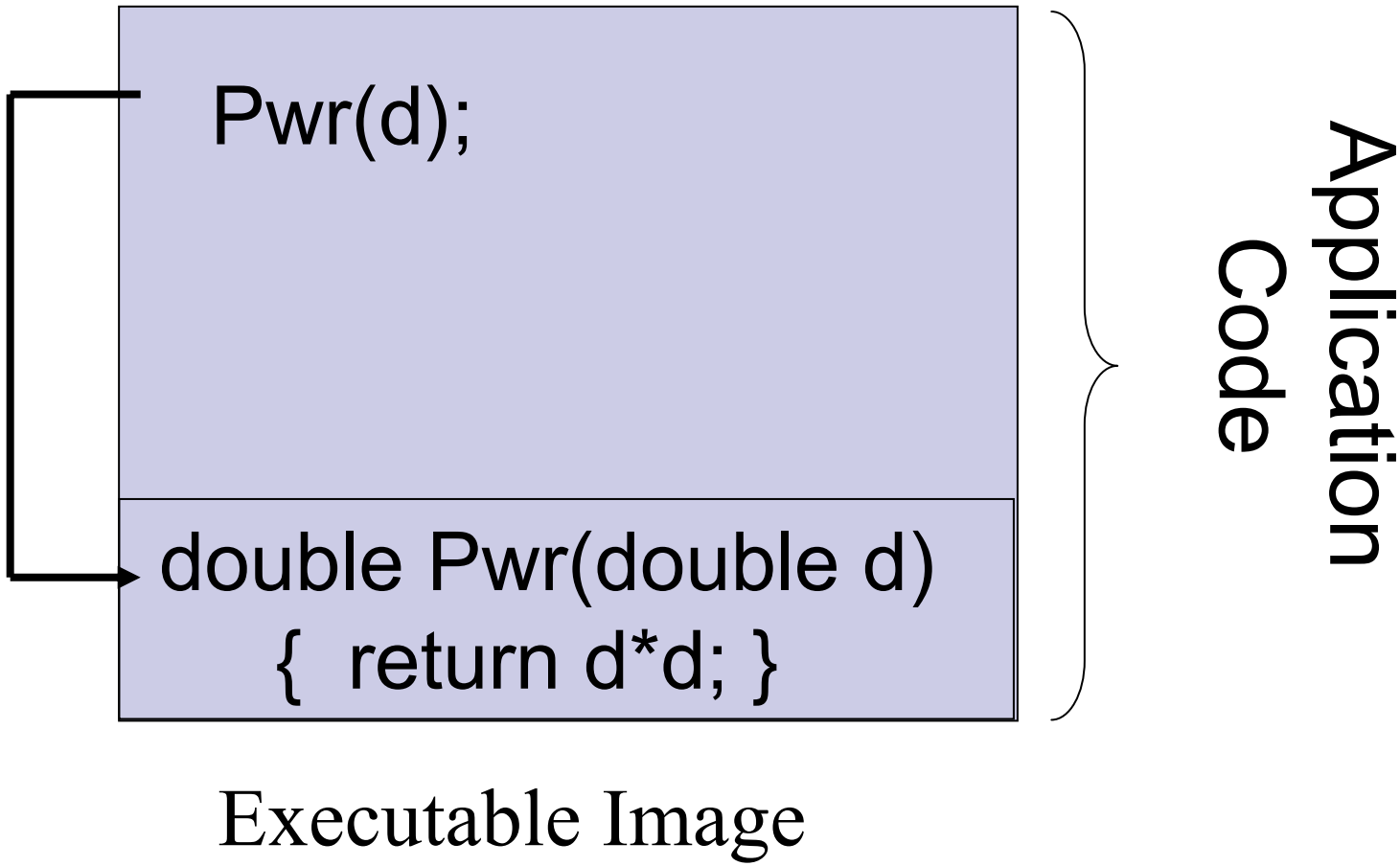
- We've all been to a library, but what is a library in programming?
  - A collection of precompiled routines or functions that a program can use.
- We put commonly used routines in a library so we don't have to re-write them
  - Example: sorting a list of numbers
- Windows and Unix use a special kind of library called Dynamic Link Libraries - implement published interface (*hide true system call from application program*)

Dynamic Link Libraries (DLL)

- A DLL is: A **library** of executable **functions** or **data** that can be used by a Windows application. Example: user32.dll, kernel32.dll
- DLLs provide one or more functions that a Windows program accesses by creating a link to the DLL.
  - The word “Dynamic” means that the link is created whenever the function or data is needed (i.e., while the program is running) instead of being linked at compile time
- DLLs can also contain just data--icons (e.g., shell32.dll), fonts, text, etc.
- A DLL’s extension is usually .dll, but may be .sys, .fon, .drv, .386, etc.

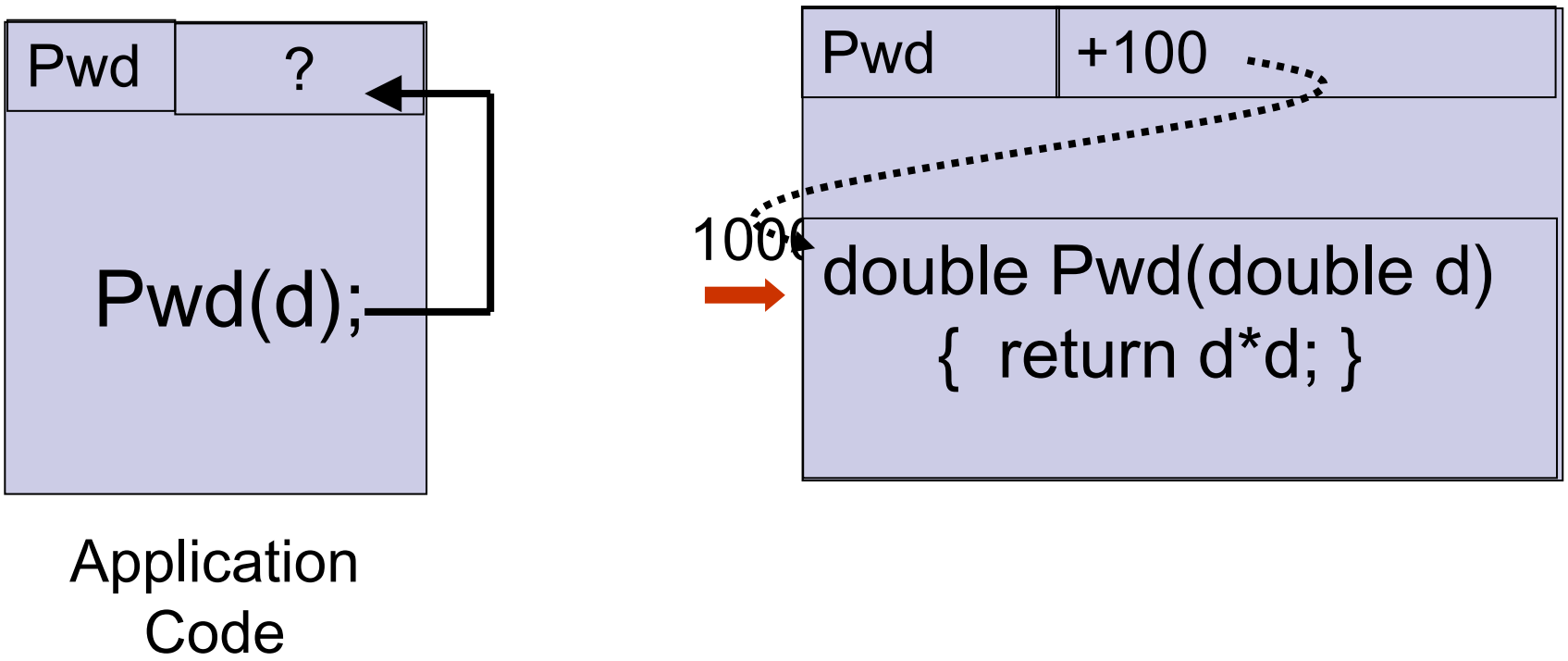
Statically linked library calls

Diagram of program flow



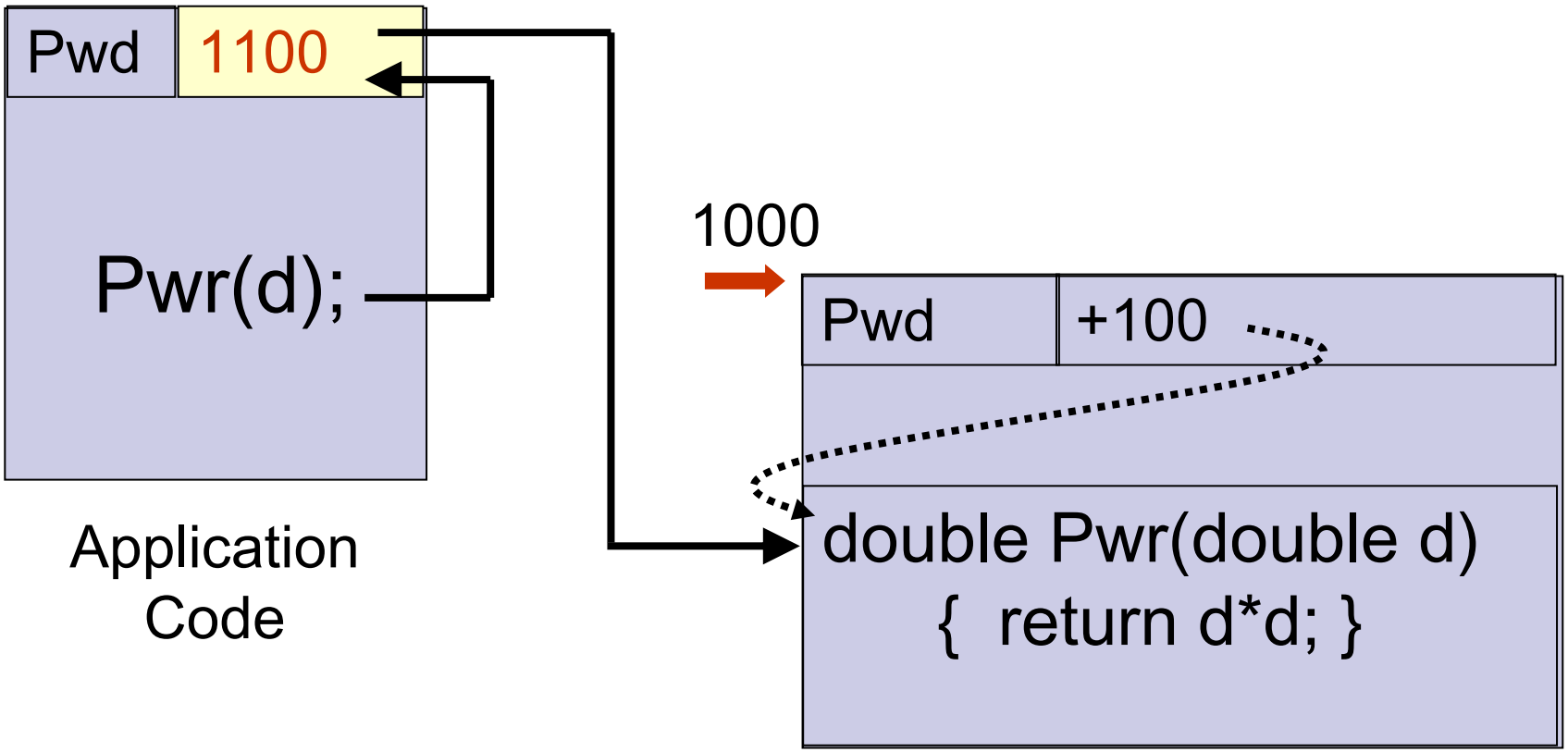
DLL calls animation 1/2:

Diagram of program flow



DLL calls animation 2/2

Diagram of program flow

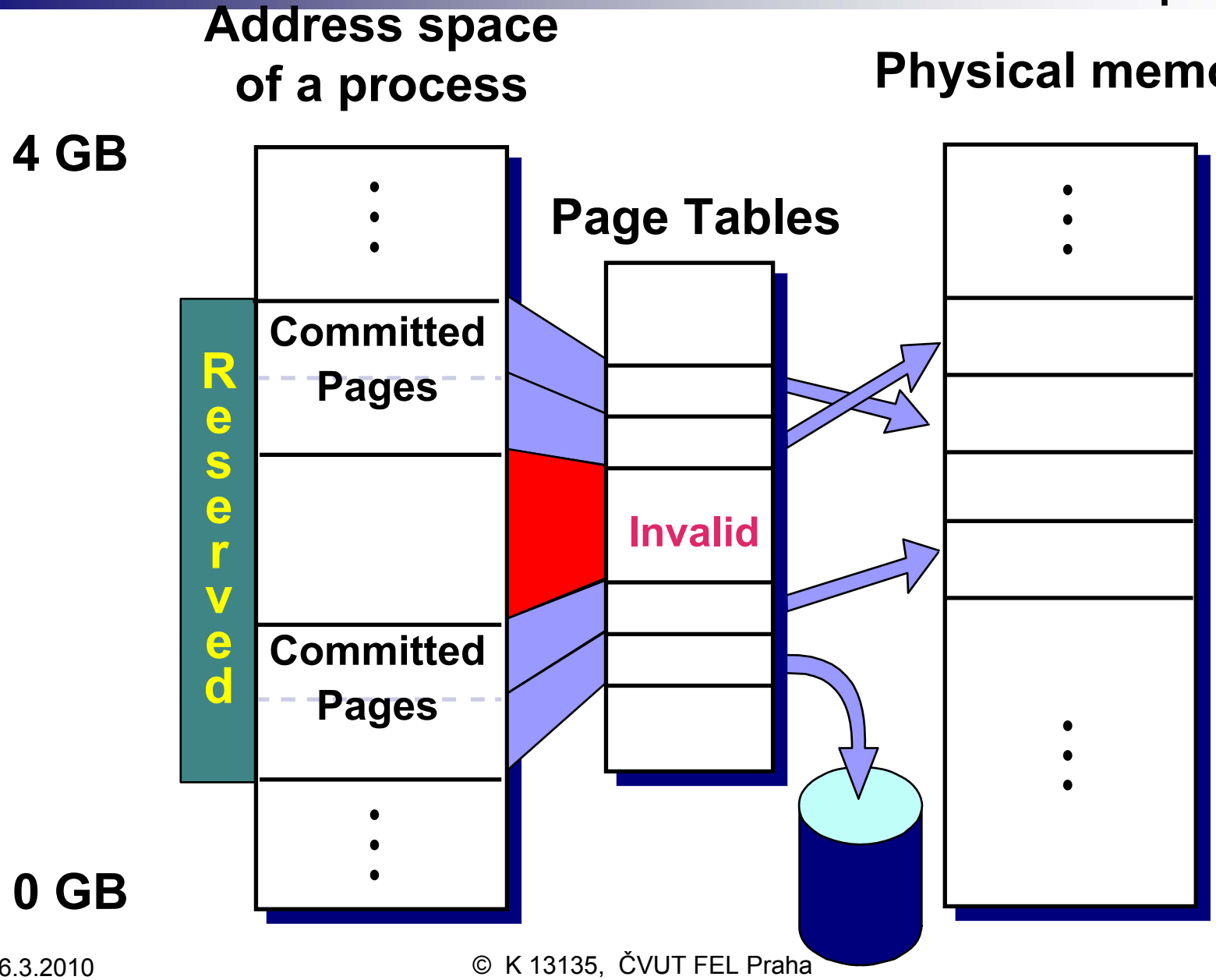


Address Space
of Processes

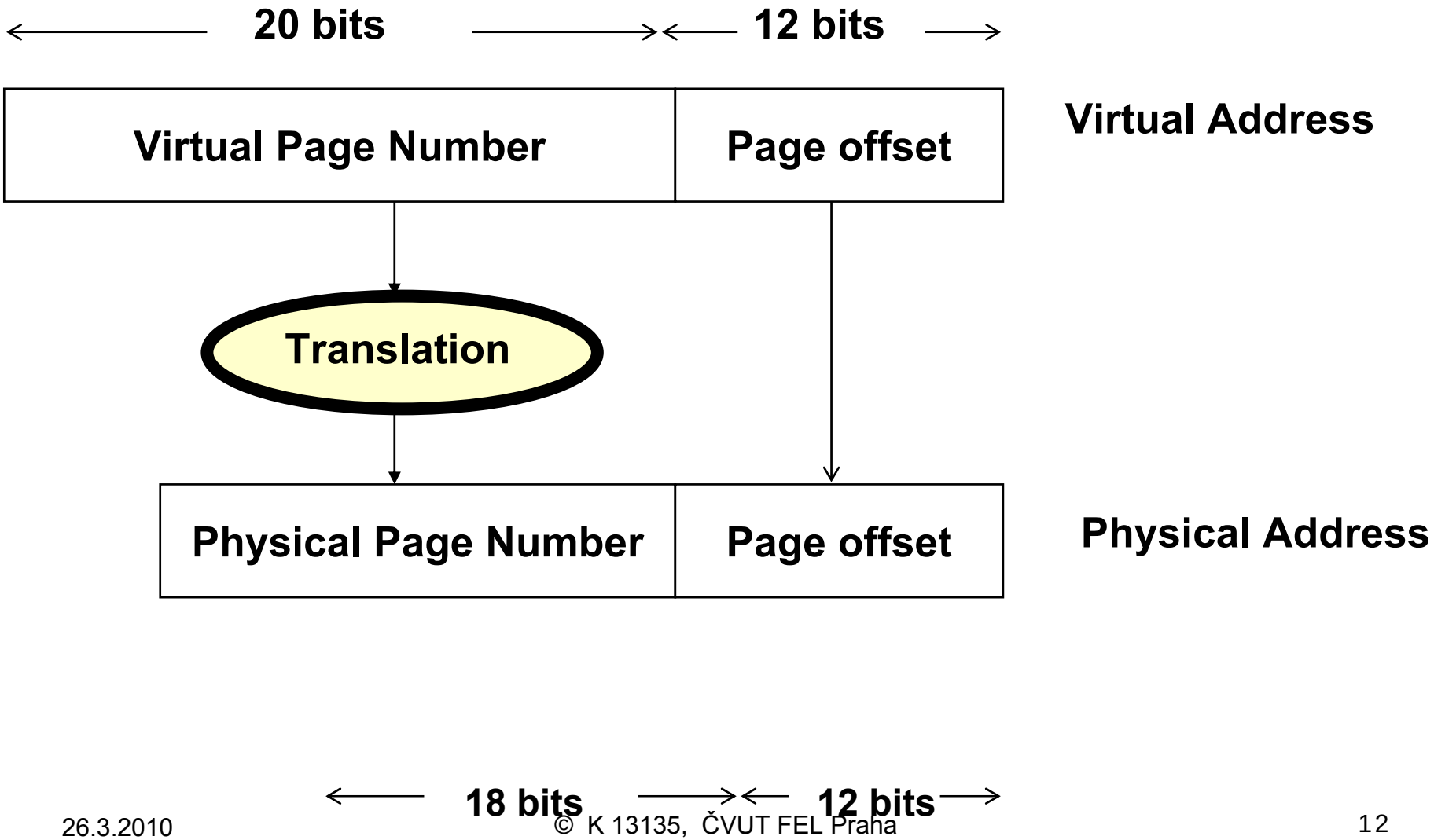
Adresový prostor

- Paměť se mapuje na "adresový prostor" (*address space*)
- 32 bitové procesory:
  adresový prostor 4 GB = 2^{32}
- 64 bitové Itanium:
  fyzicky adresový prostor 2^{64},
  avšak Win64 Vista umožňuje jen 8TB
  - důvod: stránkovací tabulky

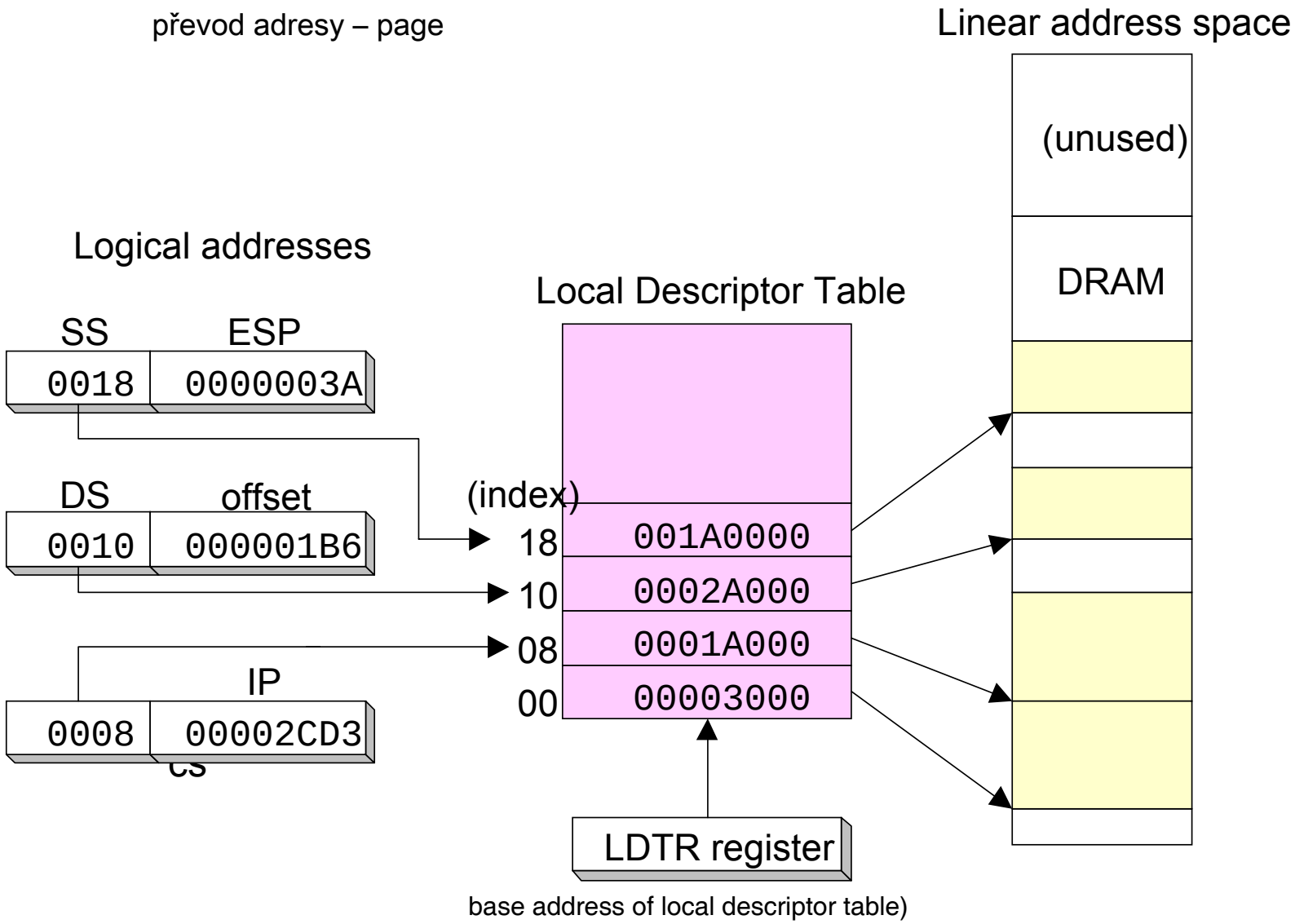
Address Space



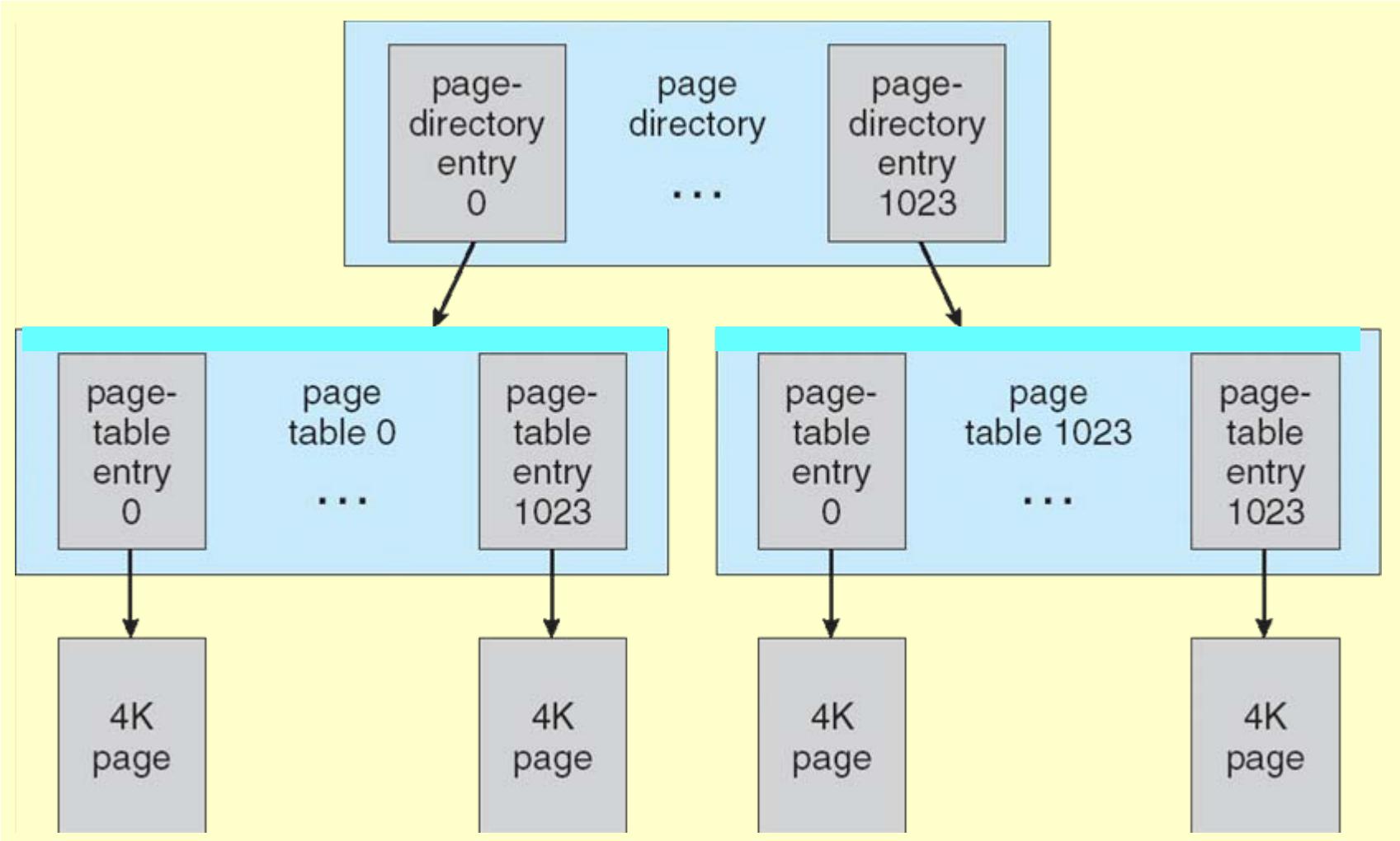
Virtuální a fyzické adresy ve 32 bitových procesorech



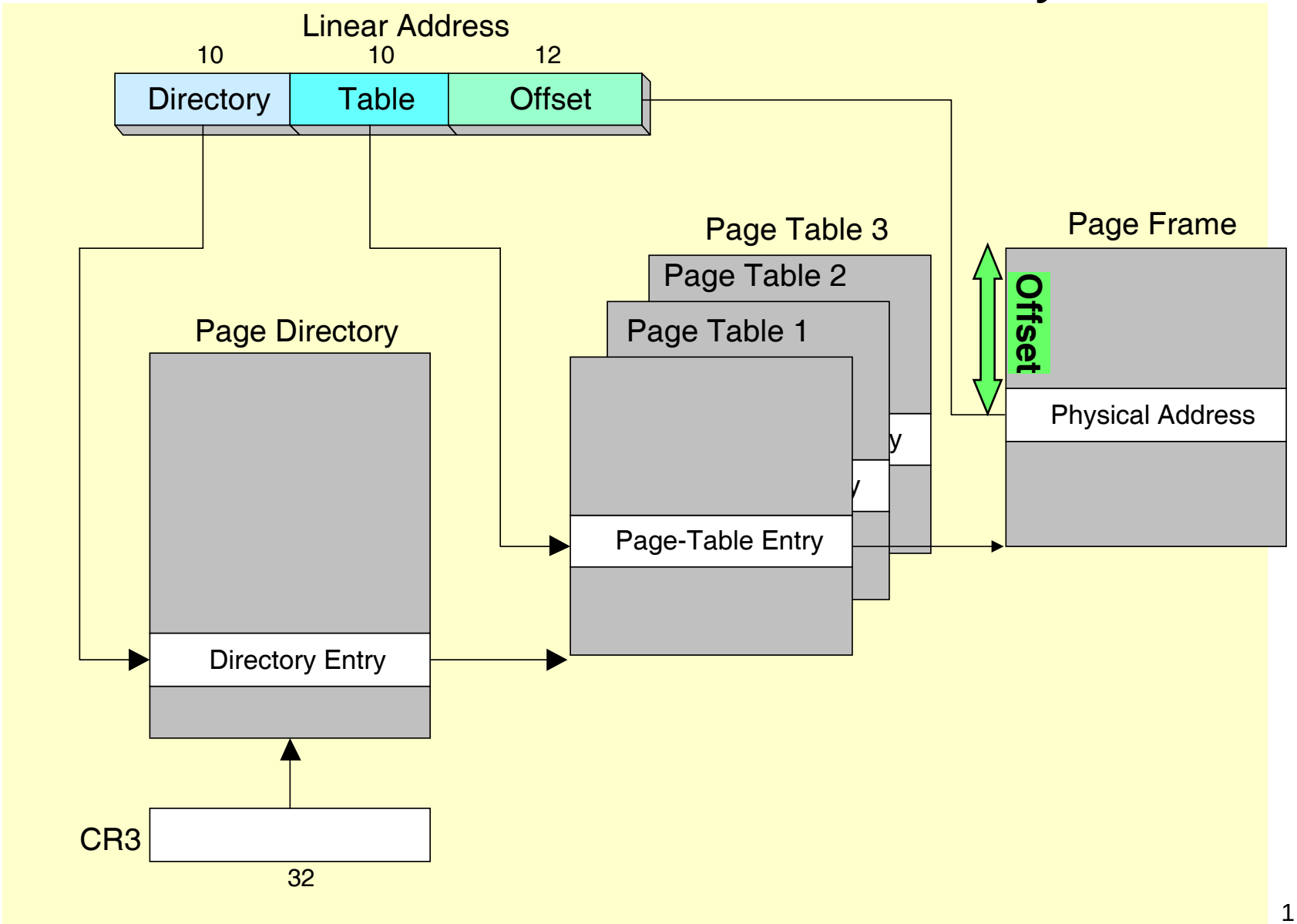
Descriptor Table – tabulka deskriptorů



Princip mapování 1/4 –nakreslený pomocí stromové struktury

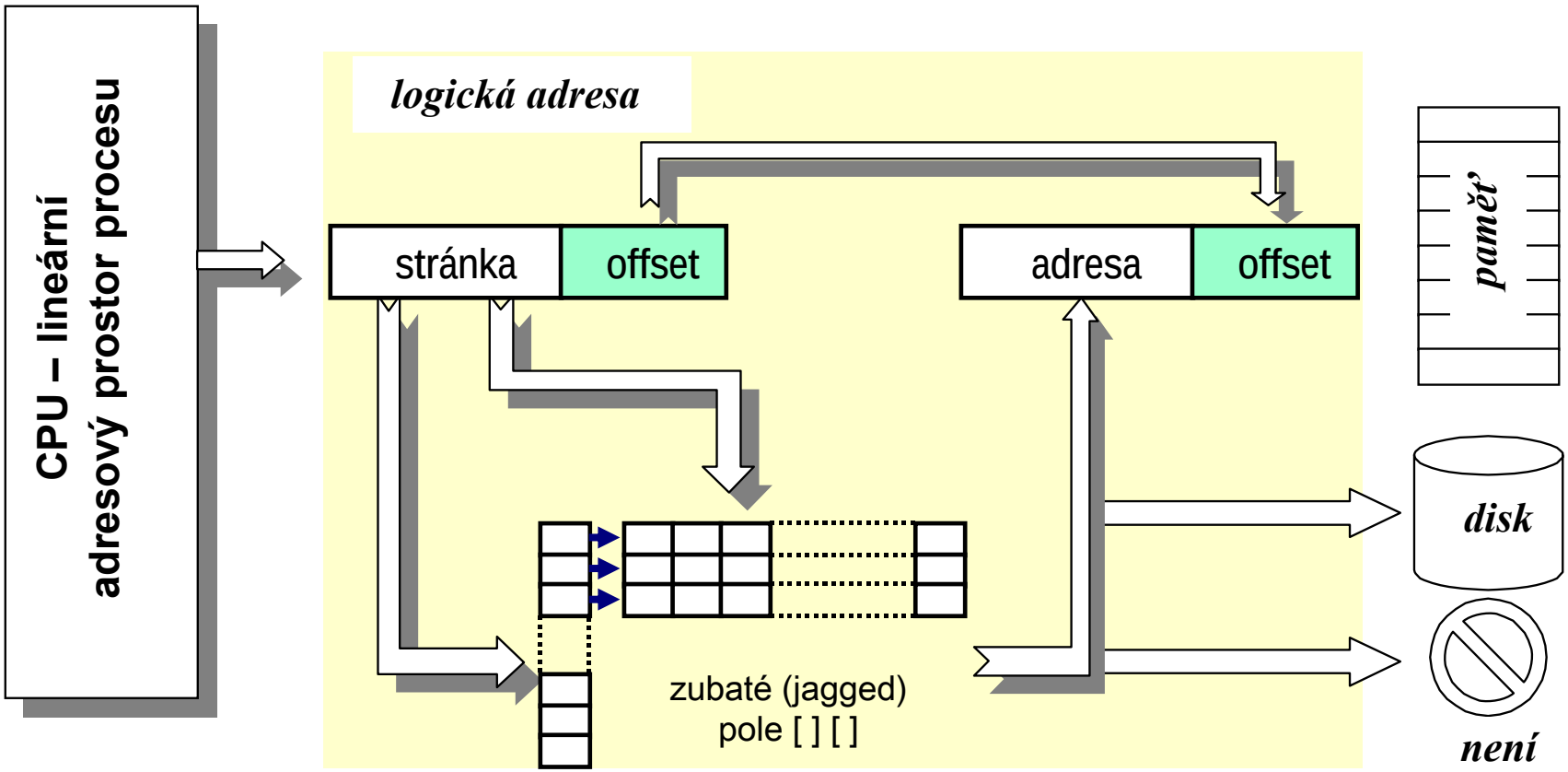


Princip mapování 2/4
– *rozklad lineární adresy na indexy*



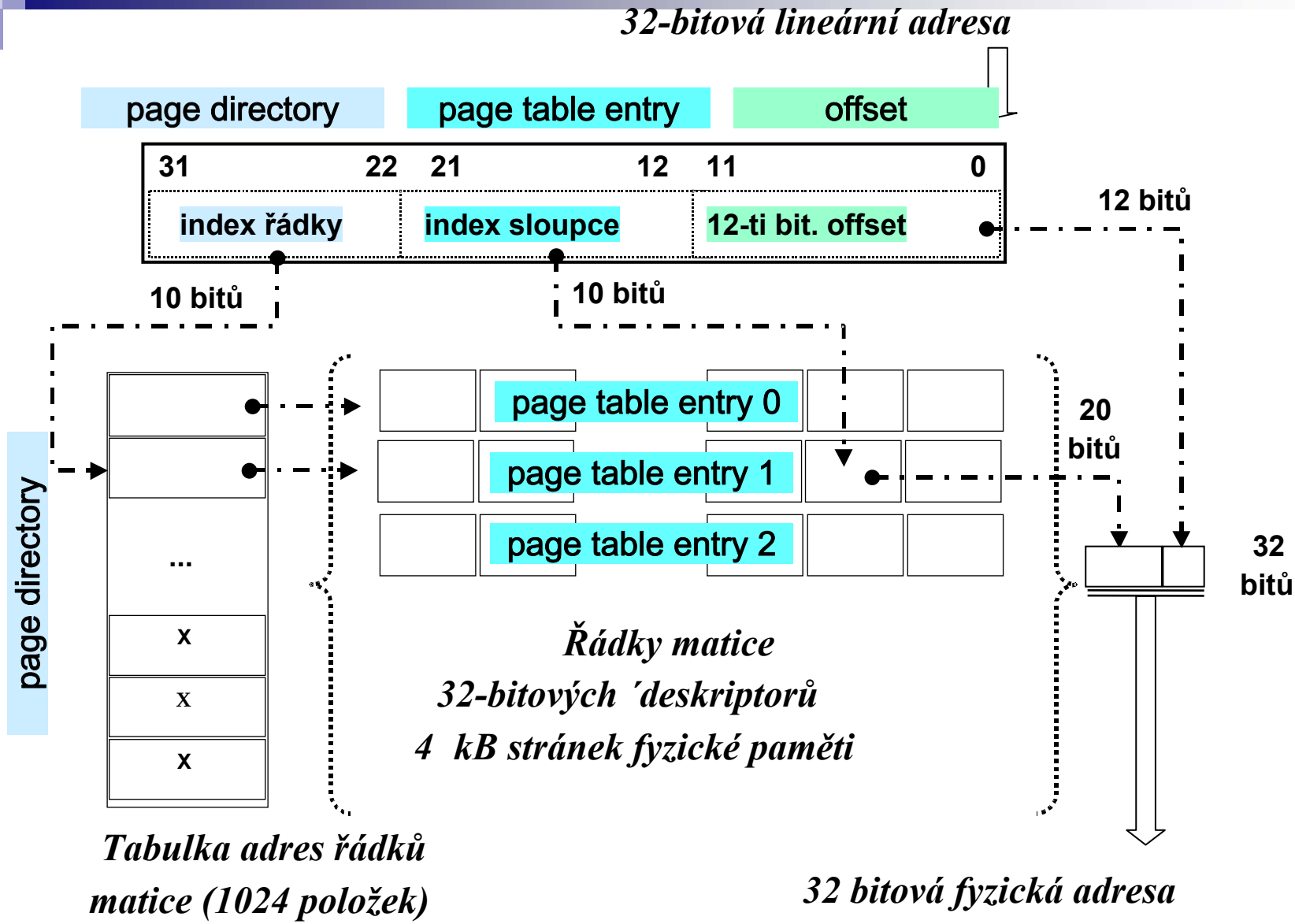
Princip mapování 3/4 – indexy do mapovací matice

- Princip převodu lineární adresy na fyzickou adresu (*page translation*)



Pozn. OS Windows i Linux používají mapování a přidělují paměť po stránkách (zpravidla KB), srovnej s "cluster"=alokační konstanta disku!

Princip mapování 4/4 - operace procesoru

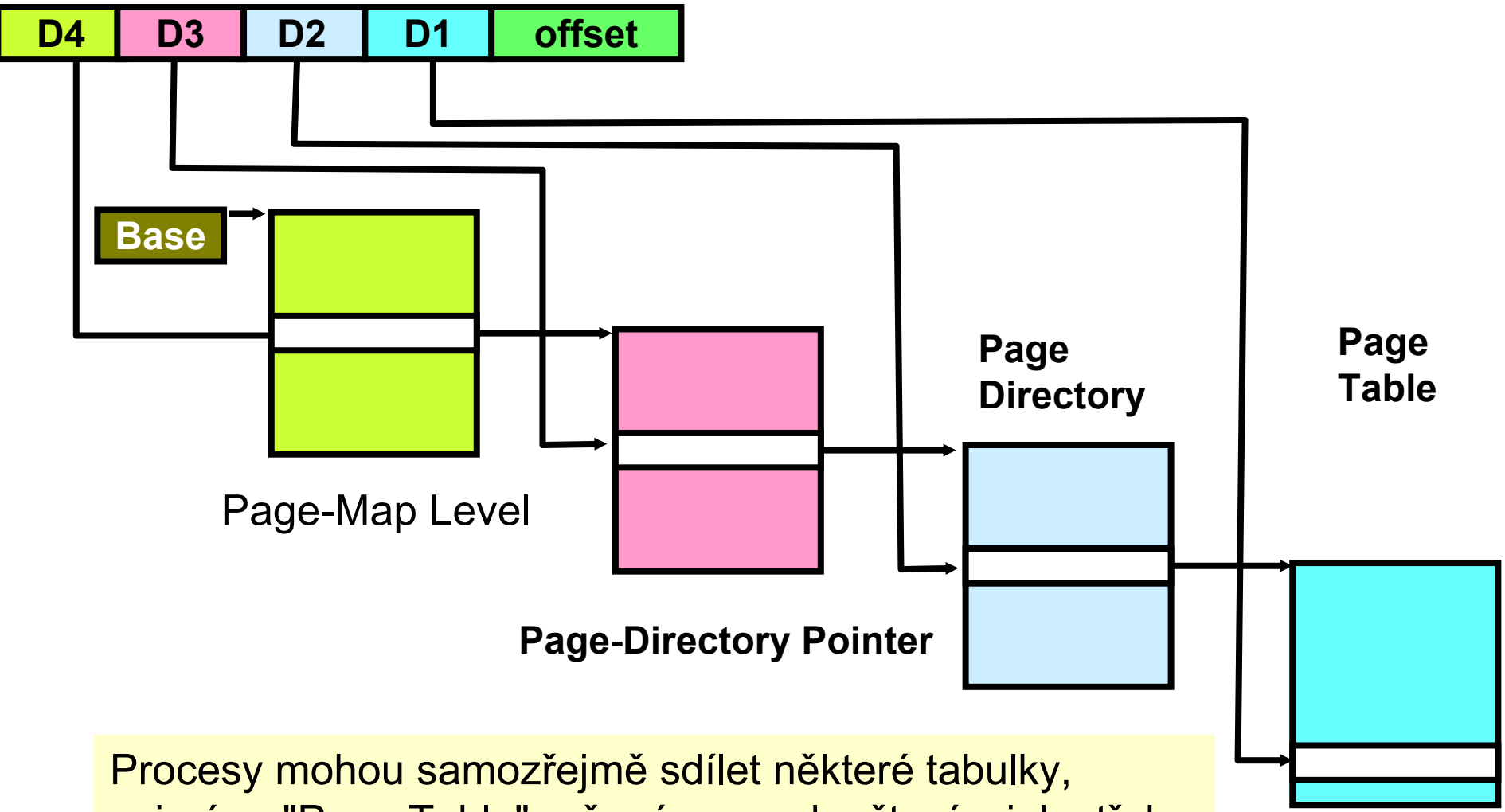


U 64 bitových systému

1. Možno zvětšit délku stránky z 4kB až na 1GB
2. Možno využít víceúrovnové stránkovací tabulky, přidat
   - *Page-Directory Pointer Table* (Win64: 4 až 512 adres)
   - *Page-Map Level* (Win64: až 512 adres)

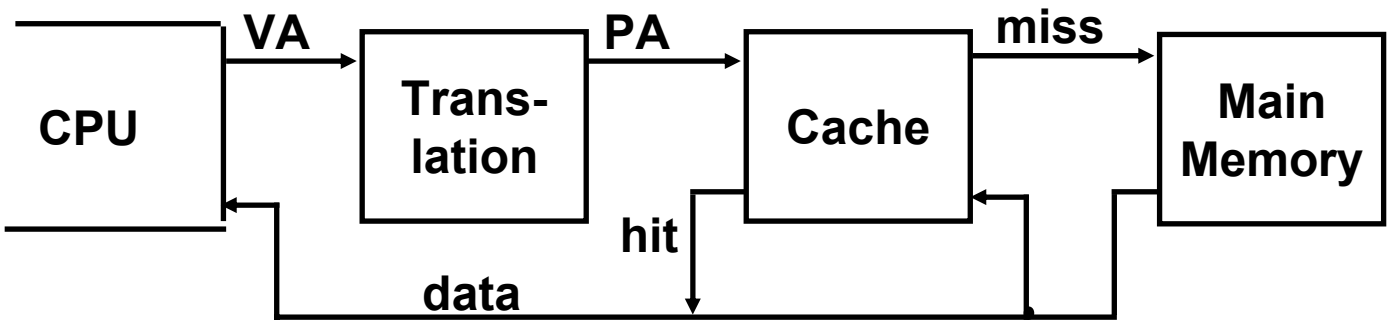
I délka deskriptoru ve všech tabulkách zvýší ze 32 bitů na 64 bitů

Více úrovňové tabulky pro 64 bitové procesory

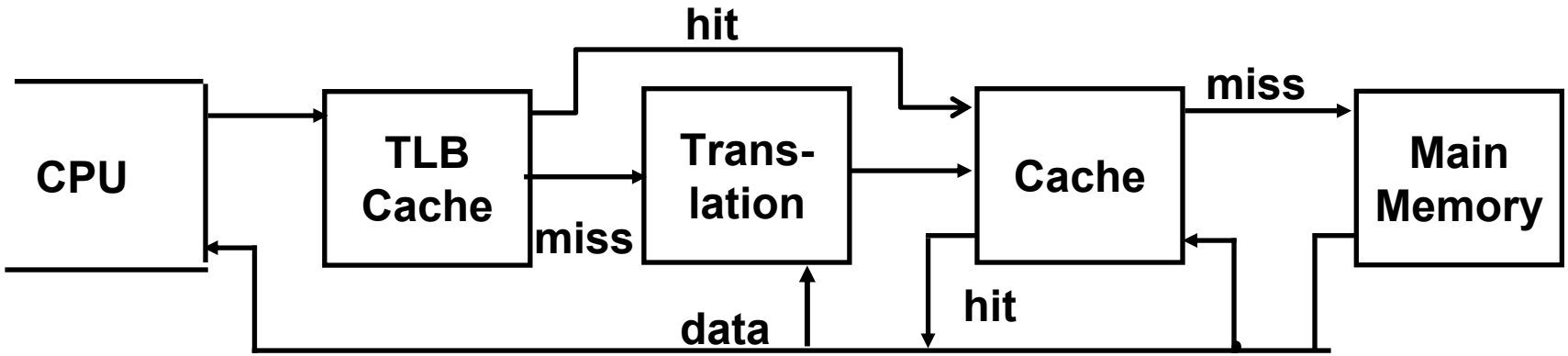


Procesy mohou samozřejmě sdílet některé tabulky, zejména "Page Table" určené pouze ke čtení – jako třeba knihovny

Nástin práce s vyrovnávací pamětí



- Převod by probíhal pomalu, kvůli nutnému čtení stránkovacích tabulek, a to i pro případná data v cashe paměti
- přidává se Translation Lookaside Buffer – paměť převedených adres



Efektivita algoritmu závisí i na tom,
jak procesor může při něm využívat cashe pamět

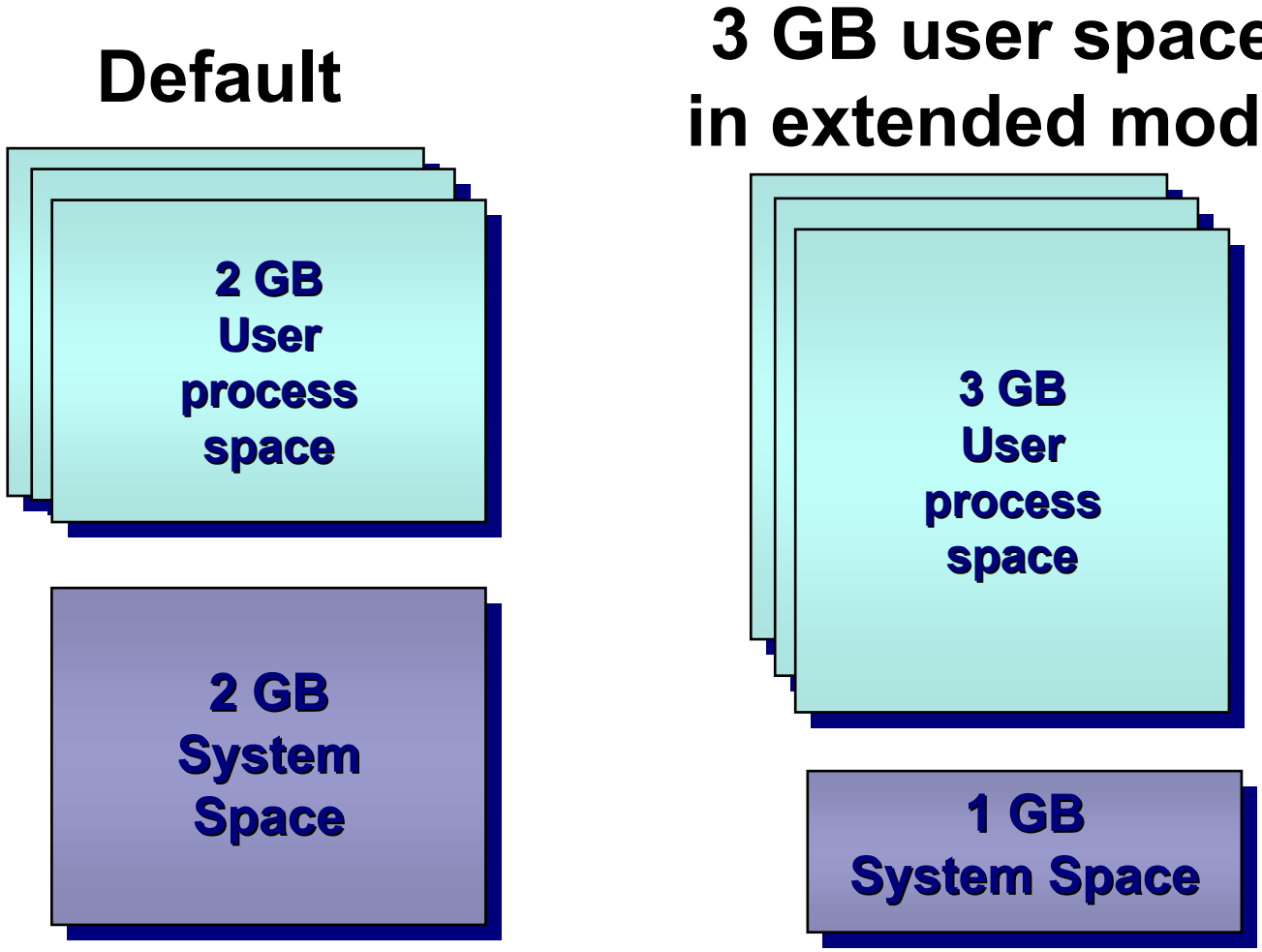
Knihovny v paměti

Technická knihovna v Dejvicích - projekt

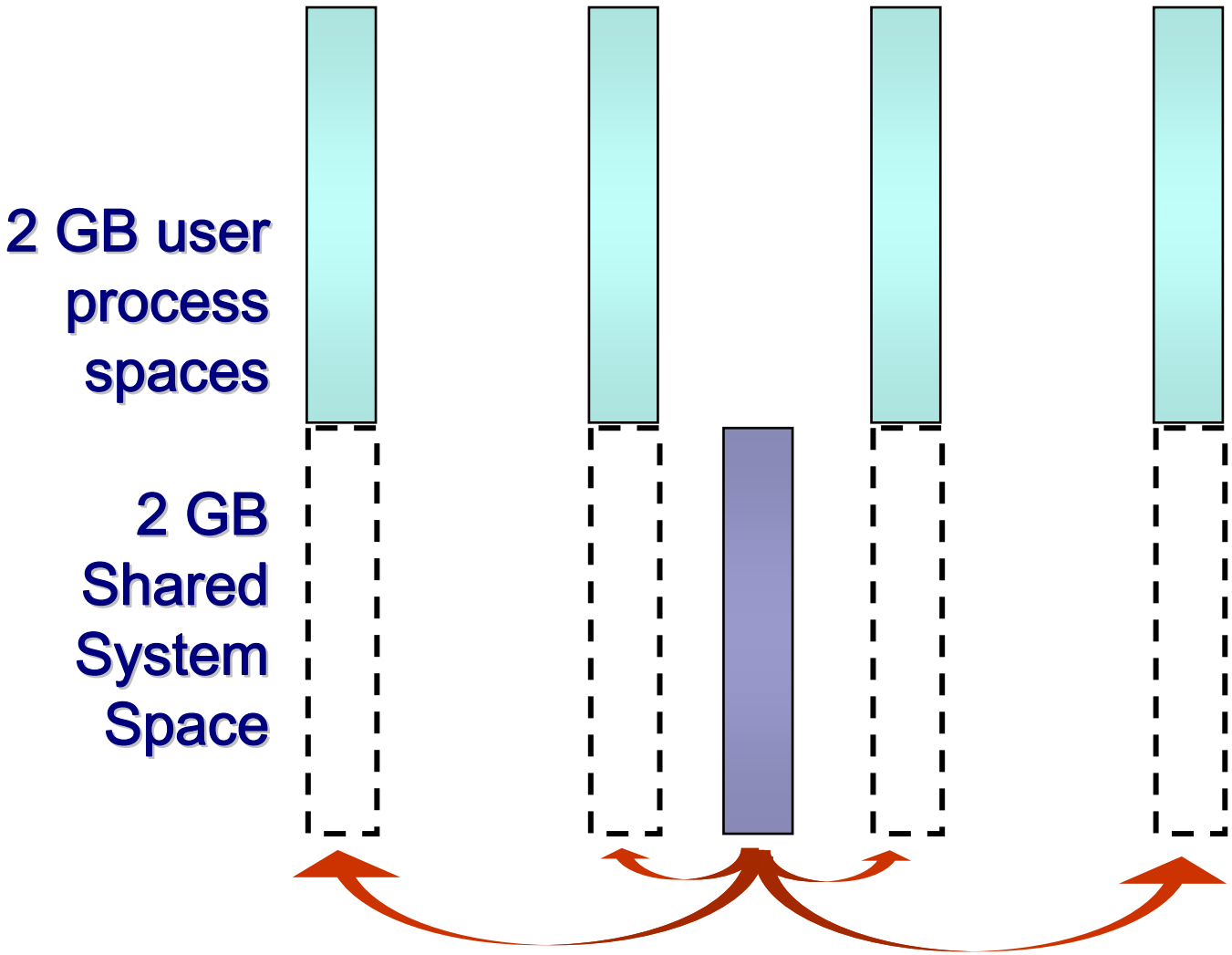
Technická knihovna v Dejvicích - realita

32-bit x86 Address Space

- 32-bits = 4 GB

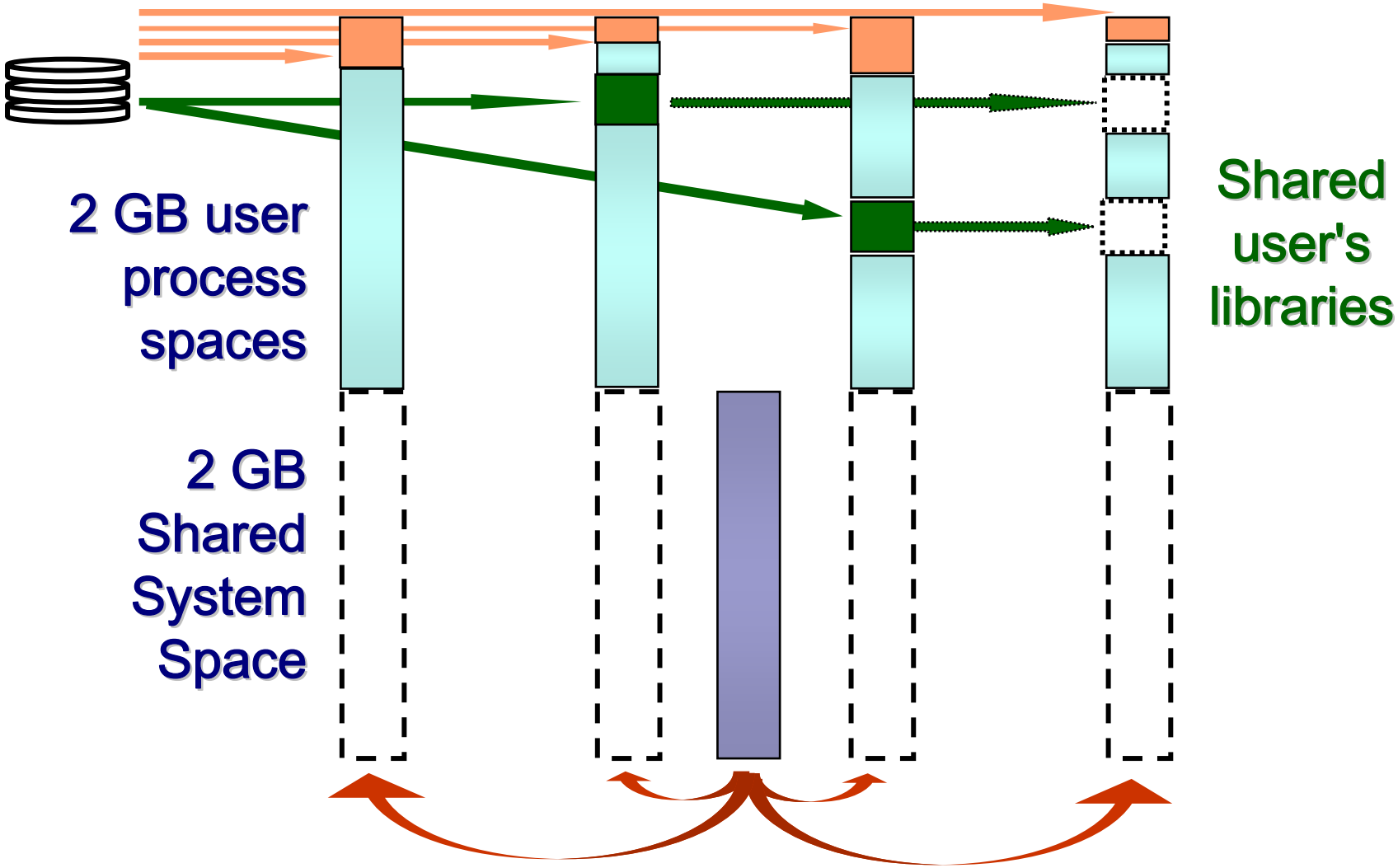


32-bit x86 Address Spaces



32-bit x86 Address Spaces

Exe files and other read only files are only mapped into address spaces

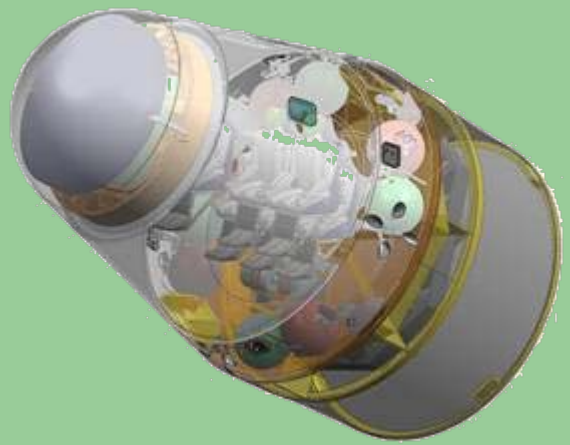


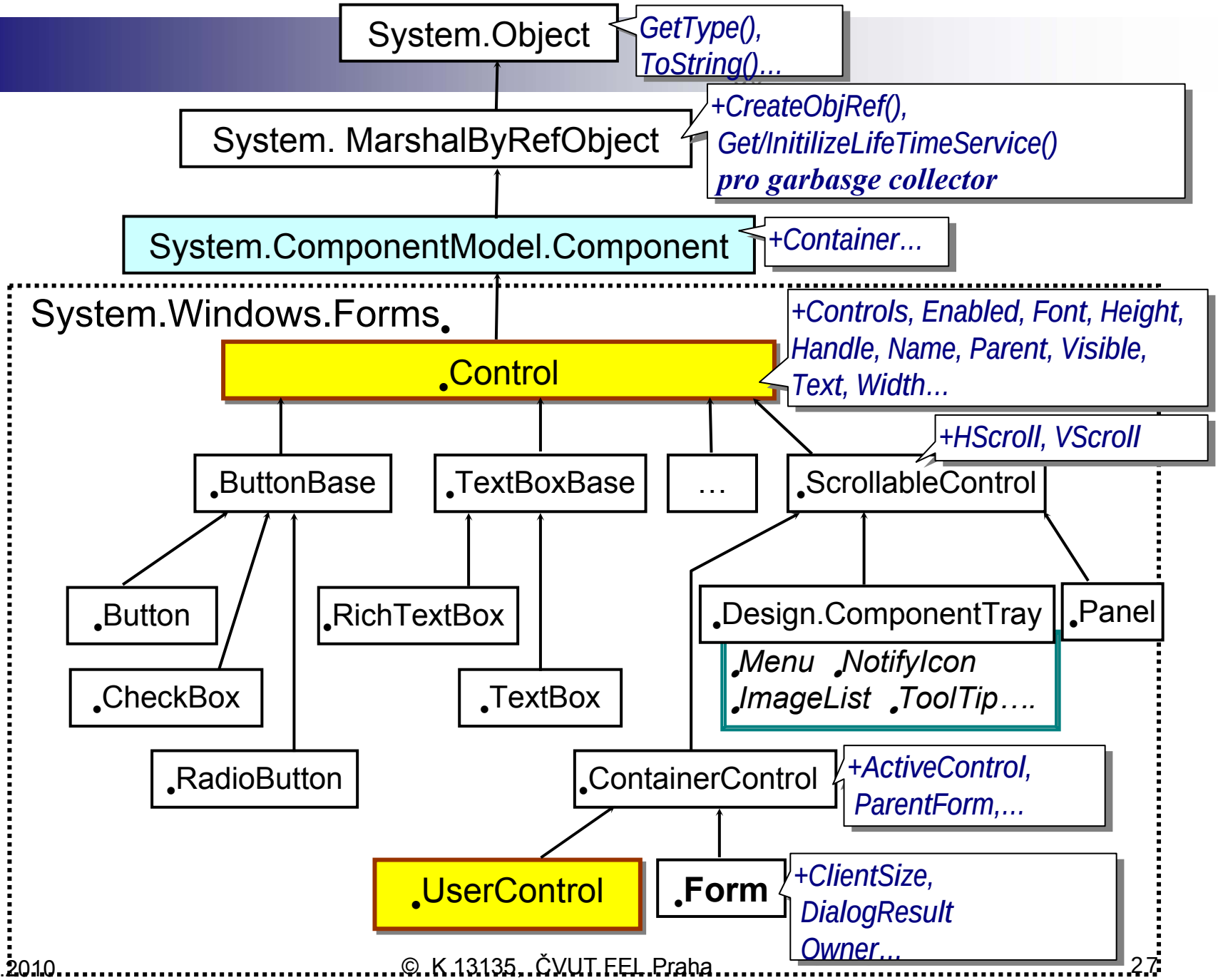
Why DLL?

- Shared code
- Development of applications can be split into more relatively independent parts
- They allow using codes written in other languages

Component

C# DLL
that contains a class implementing the IComponent interface



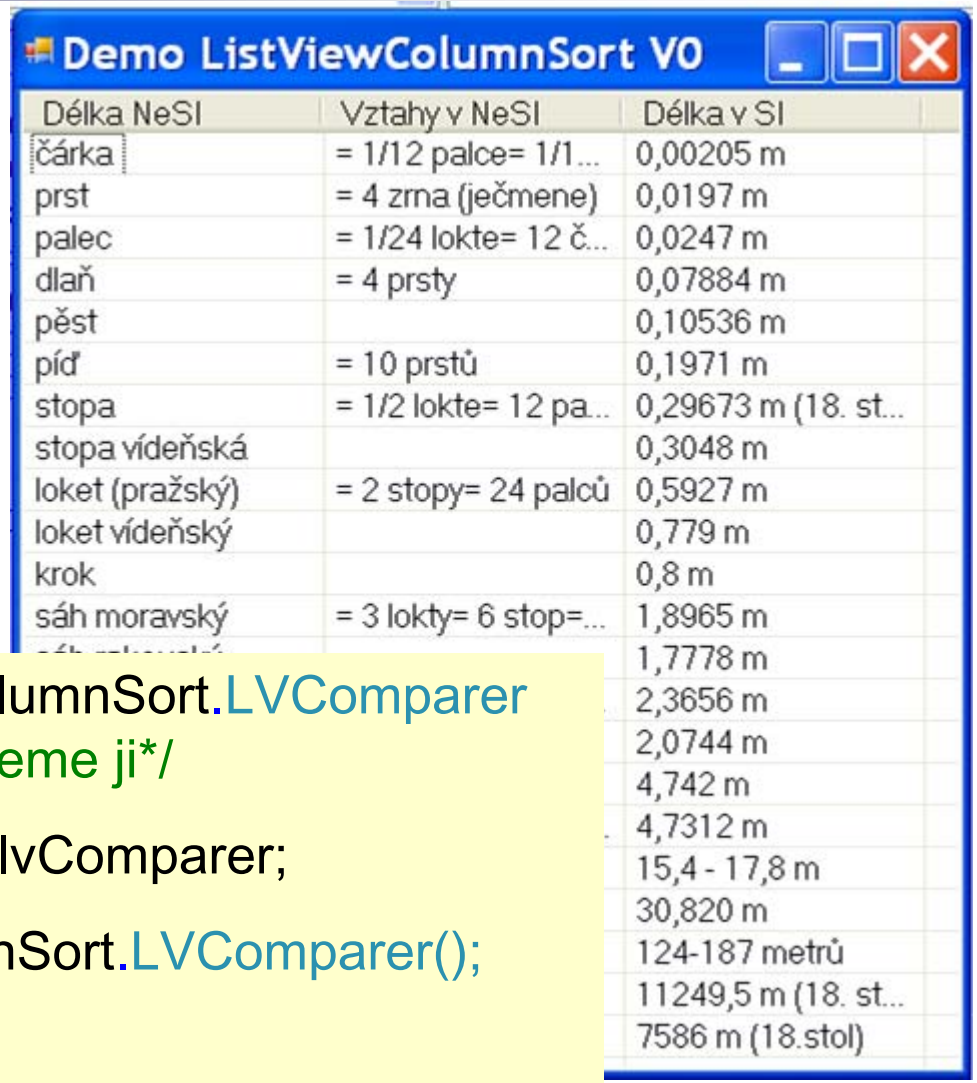

System.Object
GetType(),
ToString()...
System. MarshalByRefObject
+CreateObjRef(),
Get/InitilizeLifeTimeService()
pro garbasge collector
System.ComponentModel.Component
+Container...
System.Windows.Forms.
.Control
+Controls, Enabled, Font, Height,
Handle, Name, Parent, Visible,
Text, Width...
+HScroll, VScroll
.ButtonBase
.TextBoxBase
…
.ScrollableControl
.Button
.CheckBox
.RadioButton
.RichTextBox
.TextBox
.Design.ComponentTray
.Panel
.Menu .NotifyIcon
.ImageList .ToolTip….
.ContainerControl
+ActiveControl,
ParentForm,...
.UserControl
.Form
+ClientSize,
DialogResult
Owner...

Demonstrace vytvoření komponety

- ListView schopný třídit podle sloupců

ListView se tříděním po sloupcích V0

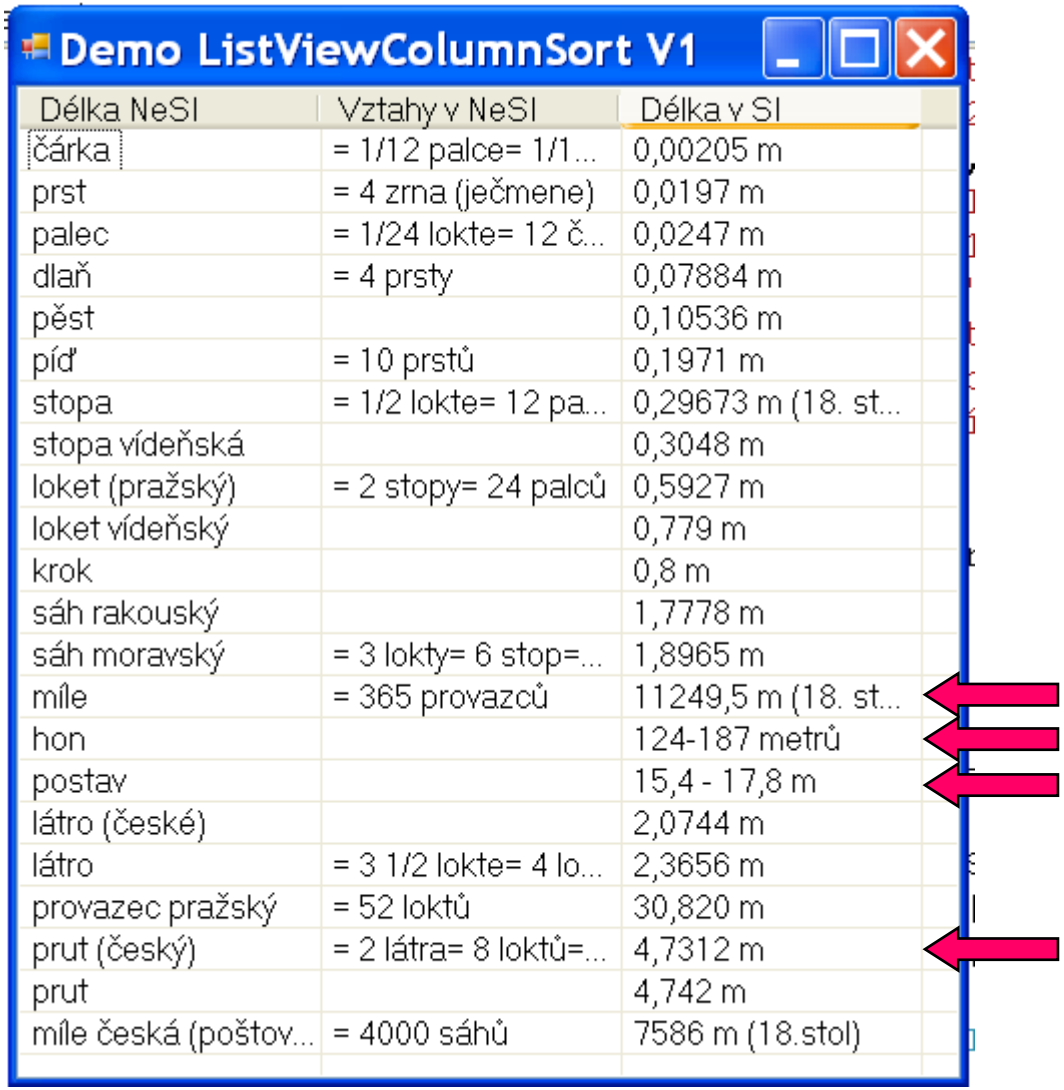
- Neumí třídit




```
/*Vytvoříme tedy třídu ListViewColumnSort.LVComparer implementující IComparer a použijeme ji*/

ListViewColumnSort.LVComparer lvComparer;

lvComparer = new ListViewColumnSort.LVComparer();

/*...*/

lvNeSI.ListViewItemSorter = lvComparer;
```

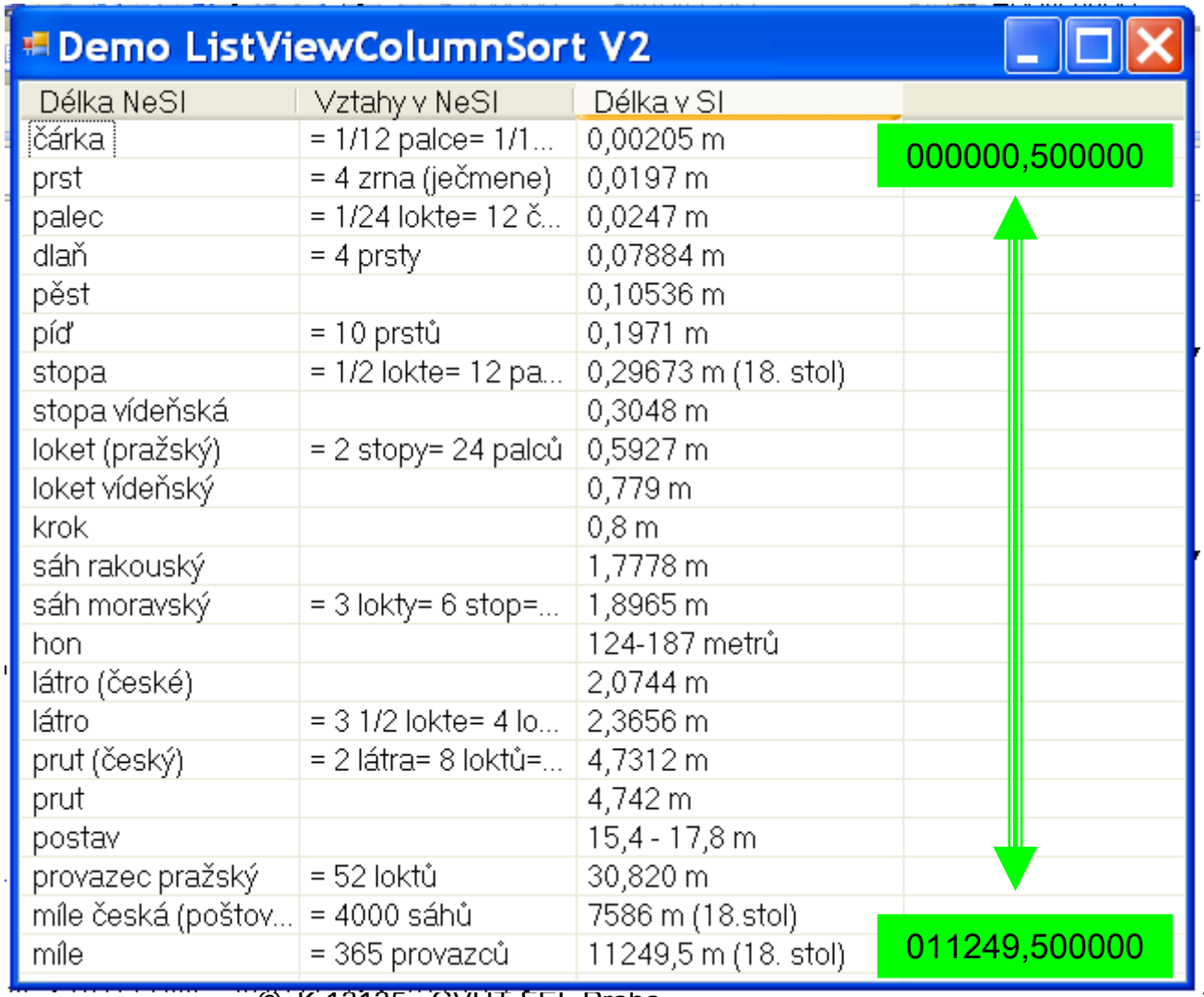
ListView se tříděním po sloupcích V1

- Demonstrace třídí, ale nikoliv dokonale



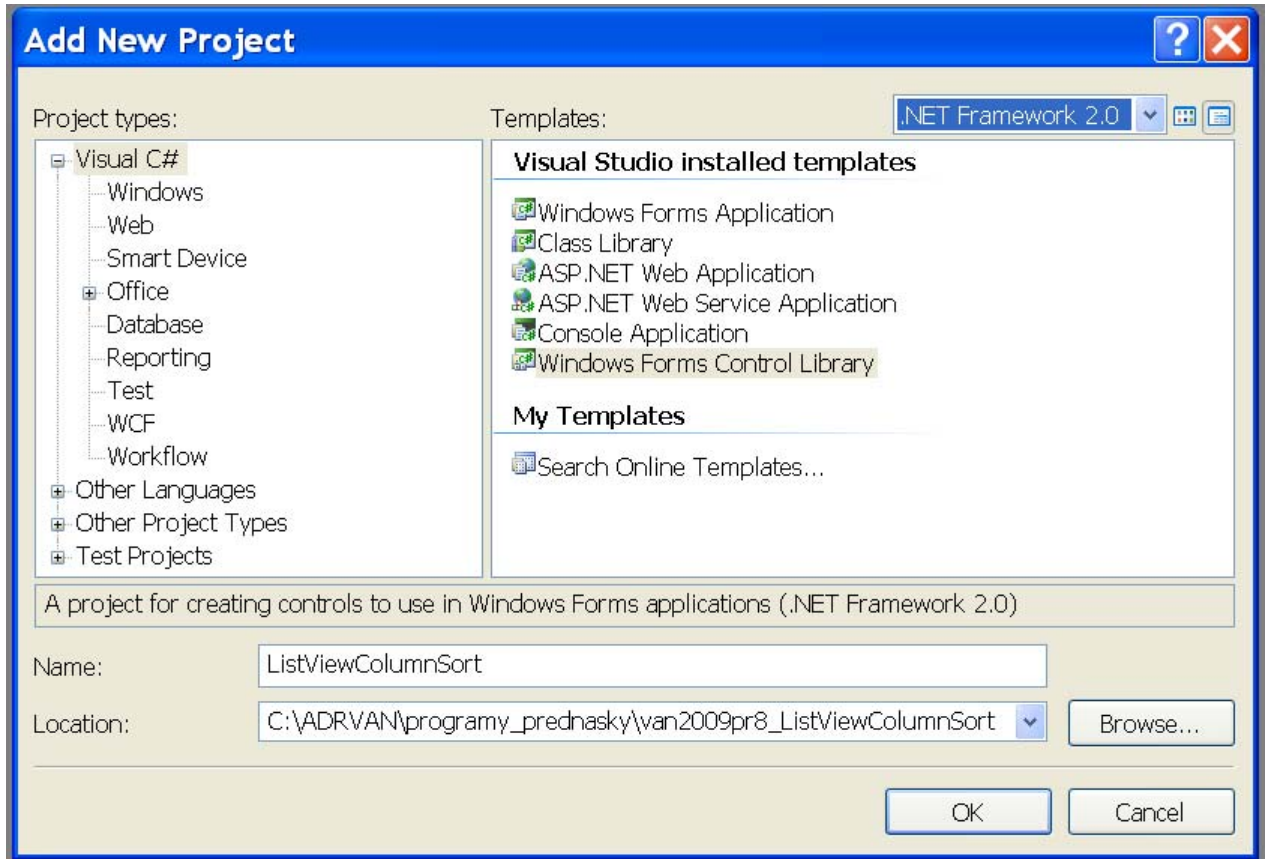
ListView se tříděním po sloupcích V2

- Demonstrace zlepšené třídění pomocí Tagu



Délka NeSI	Vztahy v NeSI	Délka v SI
čárka	= 1/12 palce= 1/1...	0,00205 m
prst	= 4 zrna (ječmene)	0,0197 m
palec	= 1/24 lokte= 12 č...	0,0247 m
dlaň	= 4 prsty	0,07884 m
pěst		0,10536 m
píď	= 10 prstů	0,1971 m
stopa	= 1/2 lokte= 12 pa...	0,29673 m (18. stol)
stopa vídeňská		0,3048 m
loket (pražský)	= 2 stopy= 24 palců	0,5927 m
loket vídeňský		0,779 m
krok		0,8 m
sáh rakouský		1,7778 m
sáh moravský	= 3 lokty= 6 stop=...	1,8965 m
hon		124-187 metrů
látro (české)		2,0744 m
látro	= 3 1/2 lokte= 4 lo...	2,3656 m
prut (český)	= 2 látra= 8 loktů=...	4,7312 m
prut		4,742 m
postav		15,4 - 17,8 m
provazec pražský	= 52 loktů	30,820 m
míle česká (poštov...	= 4000 sáhů	7586 m (18.stol)
míle	= 365 provazců	11249,5 m (18. stol)

Vytvoření komponenty V3

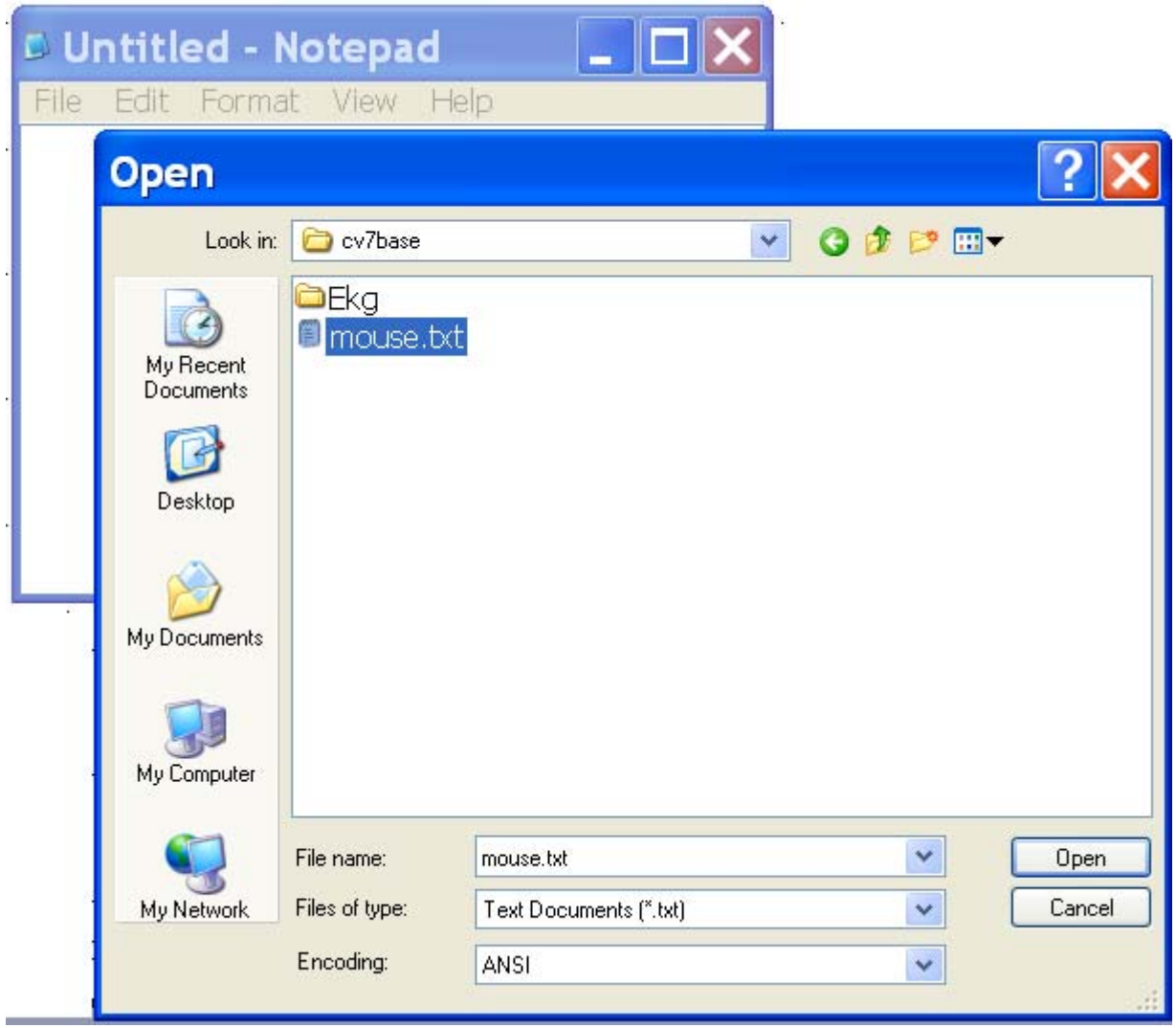


- public partial class ListViewSort : UserControl

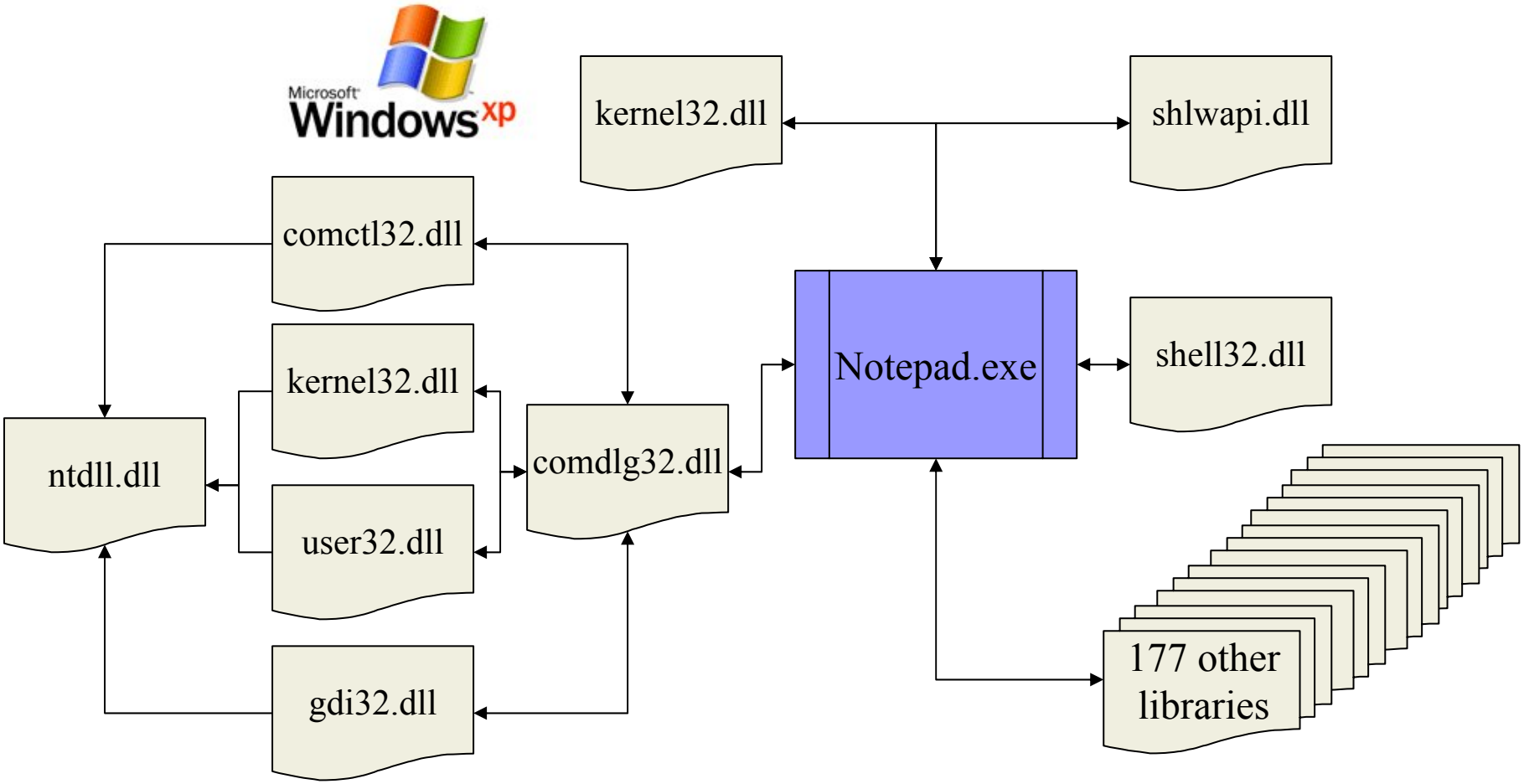
- public partial class ListViewSort : ListView
- // this.AutoScaleMode = System.Windows.Forms.AutoScaleMode.Font;

What about C++ DLLs?

Example - Opening a file in Notepad.exe



Notepad.exe - Opening a file



Is C++ DLL library faster?

Microbenchmark - FFT

[http://www.grimes.demon.co.uk/dotnet/]

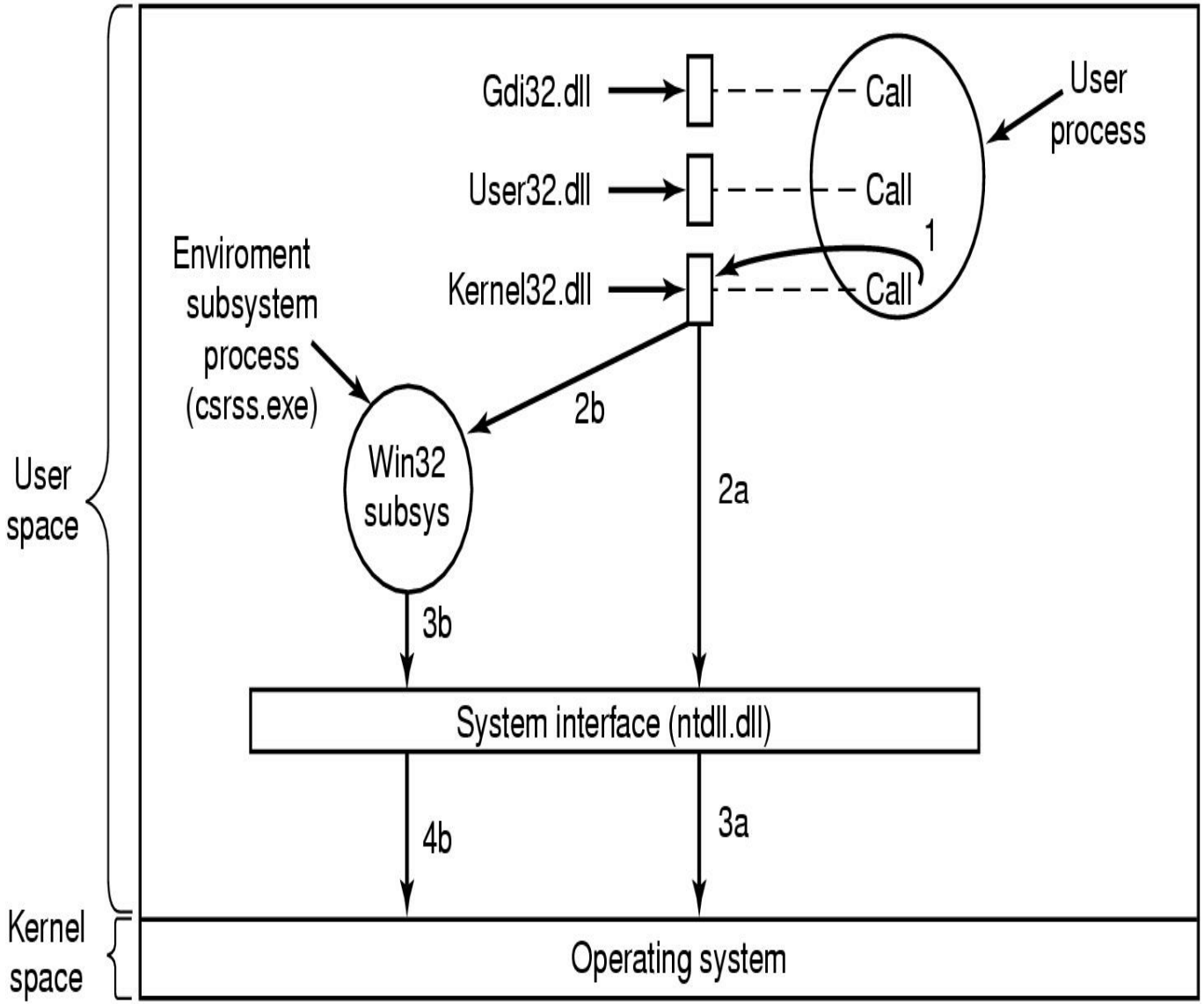
FFT calculations 2^{17} (131072) vzorků

	Not Optimized	Optimized For Space	Optimized For Speed
Unmanaged	45.2 ± 0.1	30.04 ± 0.04	23.06 ± 0.04
Managed C++	23.5 ± 0.1	23.17 ± 0.08	23.36 ± 0.07
C++/CLI	23.5 ± 0.1	23.11 ± 0.07	23.80 ± 0.05
C# Managed	23.7 ± 0.1	**22.78 ± 0.03**	

APIs and DLLs

- There are about 2,000 DLLs under the \windows directory alone, mainly in \windows\system32
- There are 4 main library files:
  - The Native API (kernel level functions) is stored in a file called **ntdll.dll**. The Win32 API libraries make use of this file to do things with hardware
  - The Win32 API is split between 3 files:
    - **kernel32.dll** - File I/O (CreateFile( )), thread management, etc.
    - **user32.dll** - Window (e.g., CreateWindow( )) and Event Messaging (e.g., mouse-clicks) functions
    - **gdi32.dll** - Drawing functions to actually draw the windows we see on the screen (e.g., LineTo( ))
- Other DLLs are written for particular applications and are installed with them (this is why we need to install!)

User Model Components



Example: C++ User's DLL Library

DLLWin32.cpp

```
DLLWIN32_API double Pwd(double d)   {  return d*d;  }
```

DLLWin32.h

```
DLLWIN32_API double Pwd(double d);
```

where* DLLWIN32_API *is define by as:

```
#ifdef DLLWIN32_EXPORTS
#define DLLWIN32_API __declspec(dllexport)
#else
#define DLLWIN32_API __declspec(dllimport)
#endif
```

How to call DLL Library from C#?

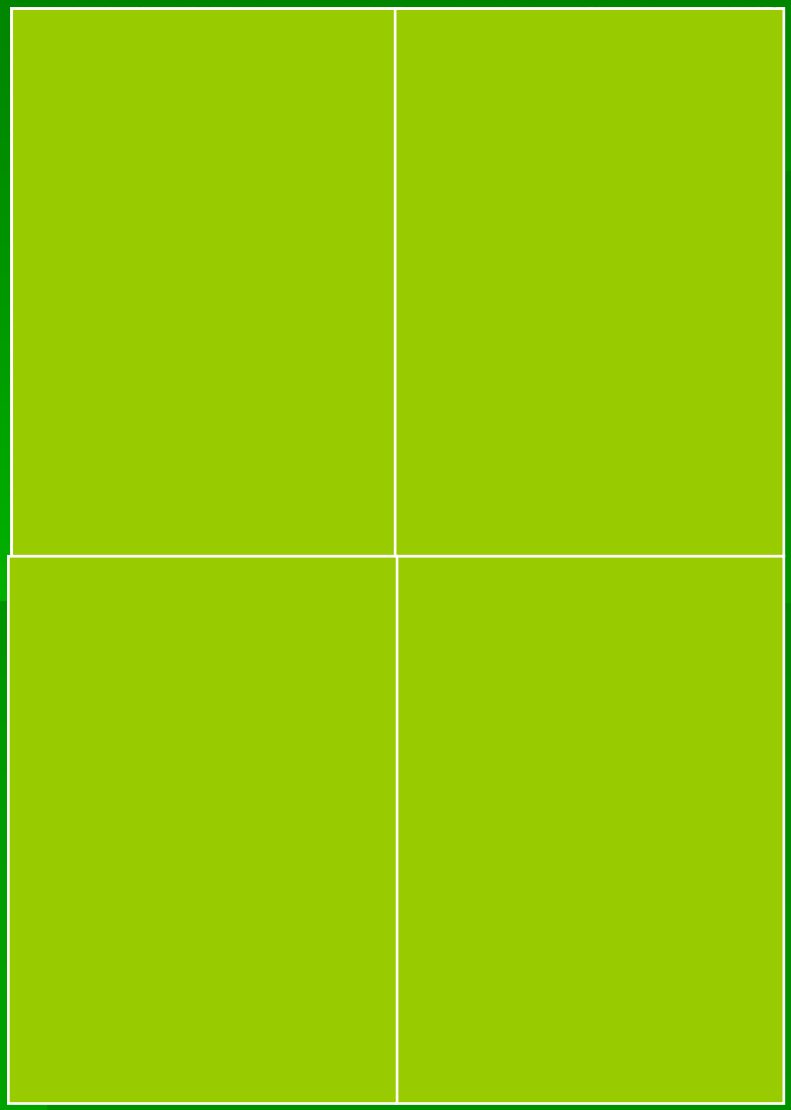
Solution Explorer
→ Add
→ New Project
→ Visual C# Projects
. Console Application

```
using System;
using System.Runtime.InteropServices;
namespace CSharpDLLWin32
{ class Class1
    {   [DllImport("DLLWin32.dll")]
        public static extern double Pwr(double d);
        [STAThread] static void Main(string[] args)
        {  double d=2;
           d=Pwr(d);
    } } // class1
} // namespace
```

It looks very easy...

- ...it was intention of C# developers
- **But to use DLLs correctly, we should know at least about**
  1. Name Mangling
  2. P/Invoke
  3. Marshaling
  4. Attributes

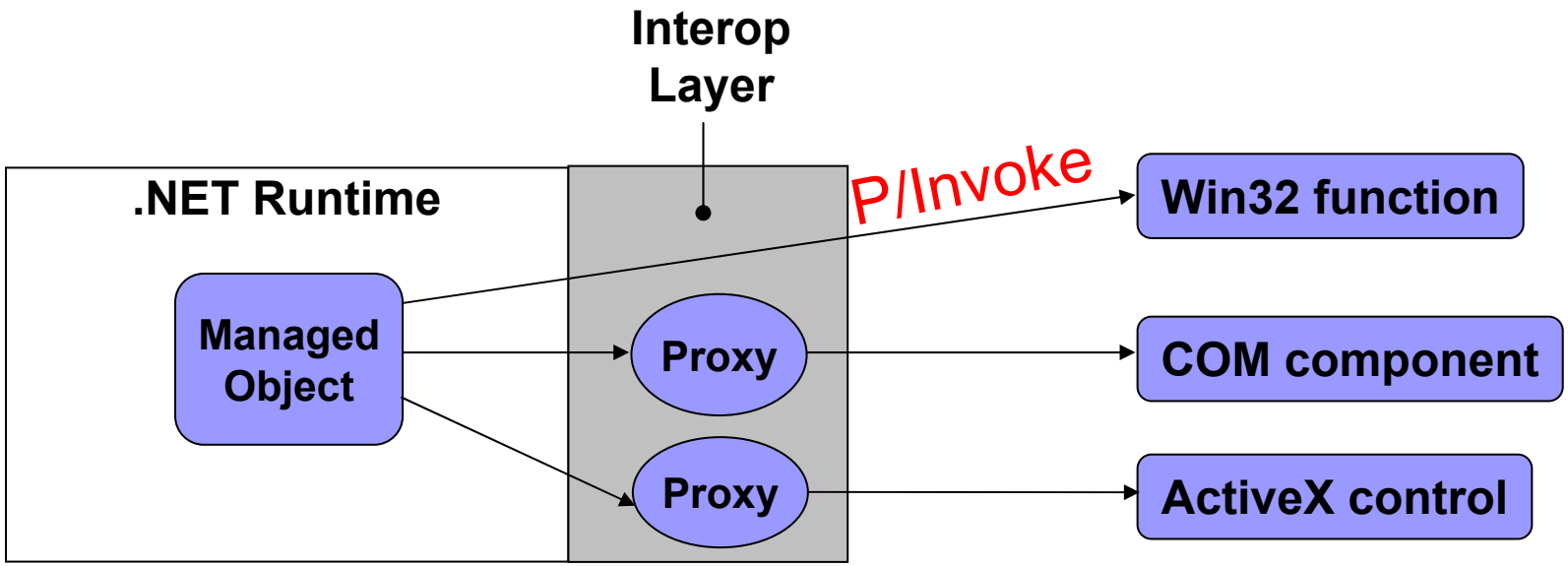
Calling DLL's in .NET



1. P/Invoke
2. Name Mangling
3. Marshaling
4. Attributes

.NET Framework interop

- .NET Framework has full featured, heavyweight Native Interop
  - large and complex
  - seamlessly integrates .NET and Native code



.NET Compact Framework Interop features

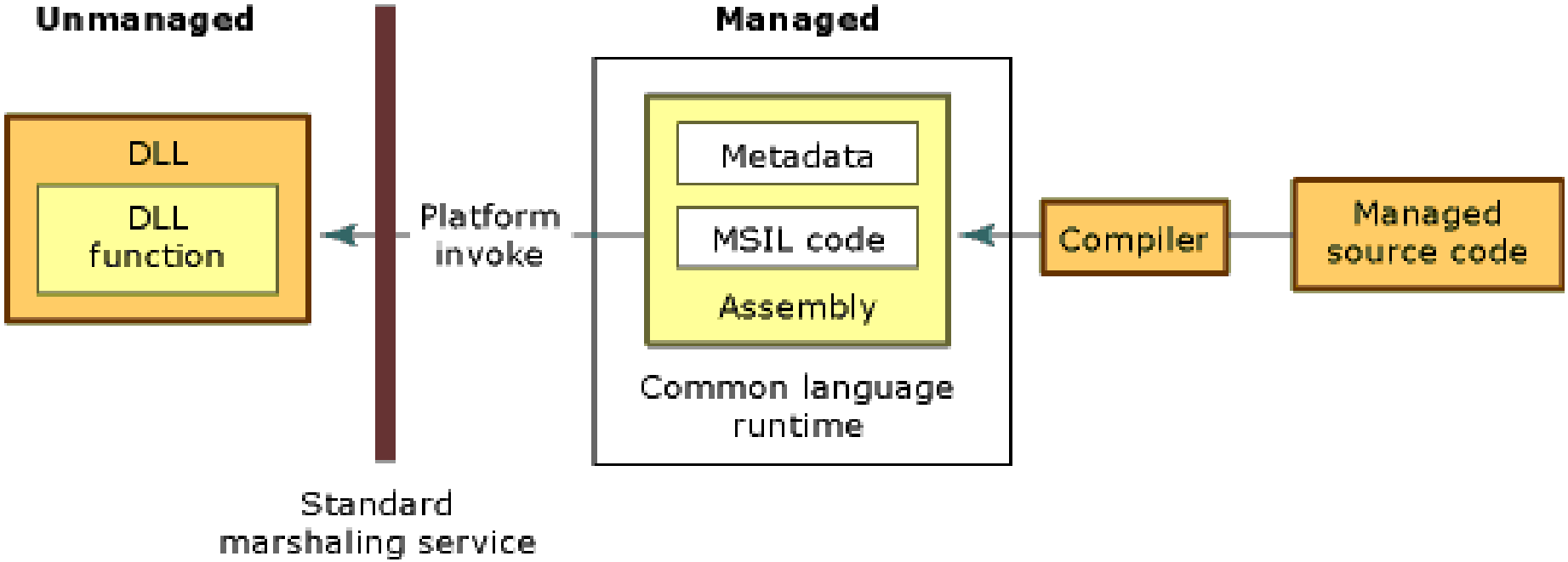
- Many possible interop patterns
  - Compact Framework directly supports some cases
  - custom code typically required
  - a few cases not supported at all

.NET calling Win32 DLL	directly supported
.NET calling COM	no direct support, but some help available*
ActiveX Controls	not supported

*** See appendix A for details on calling COM**

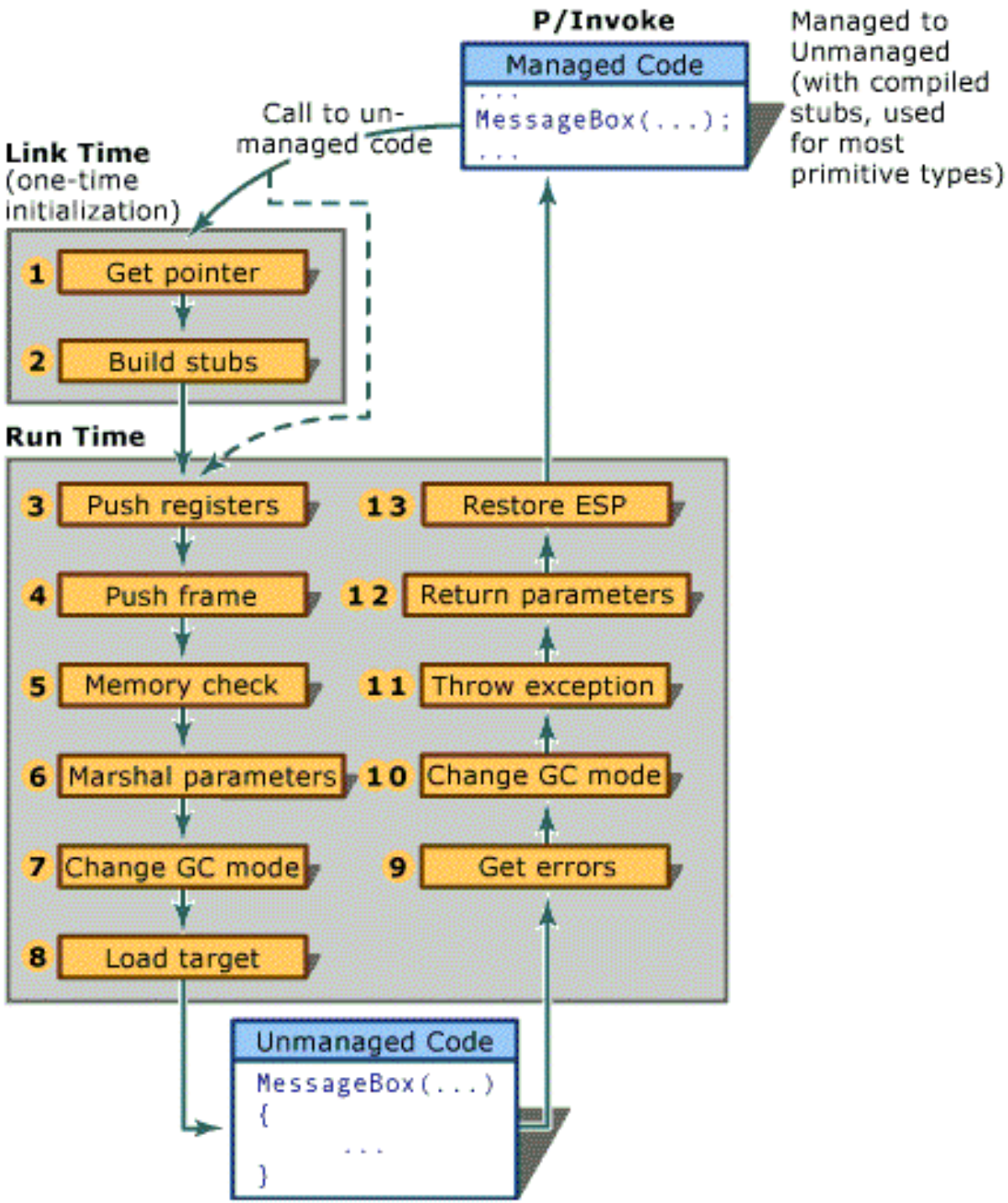
P/Invoke theory

"Platform Invoke call" calls "unmanaged DLL" from "managed" code.

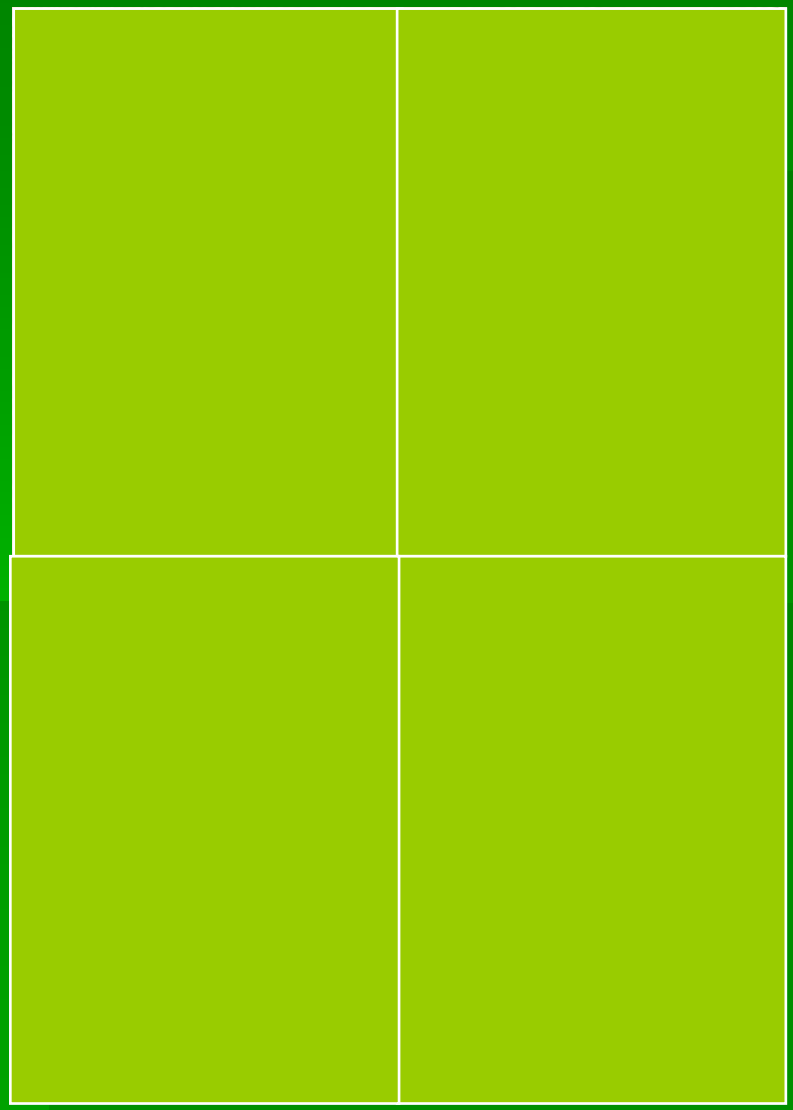


Source: http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dndotnet/html/manunmancode.asp

P/Invoke


P/Invoke
Managed Code
...
MessageBox(...);
...
Managed to Unmanaged (with compiled stubs, used for most primitive types)
Call to un-managed code
Link Time
(one-time initialization)
1 Get pointer
2 Build stubs
Run Time
3 Push registers
4 Push frame
5 Memory check
6 Marshal parameters
7 Change GC mode
8 Load target
13 Restore ESP
12 Return parameters
11 Throw exception
10 Change GC mode
9 Get errors
Unmanaged Code
MessageBox(...)
{
...
}

Calling DLL's in .NET



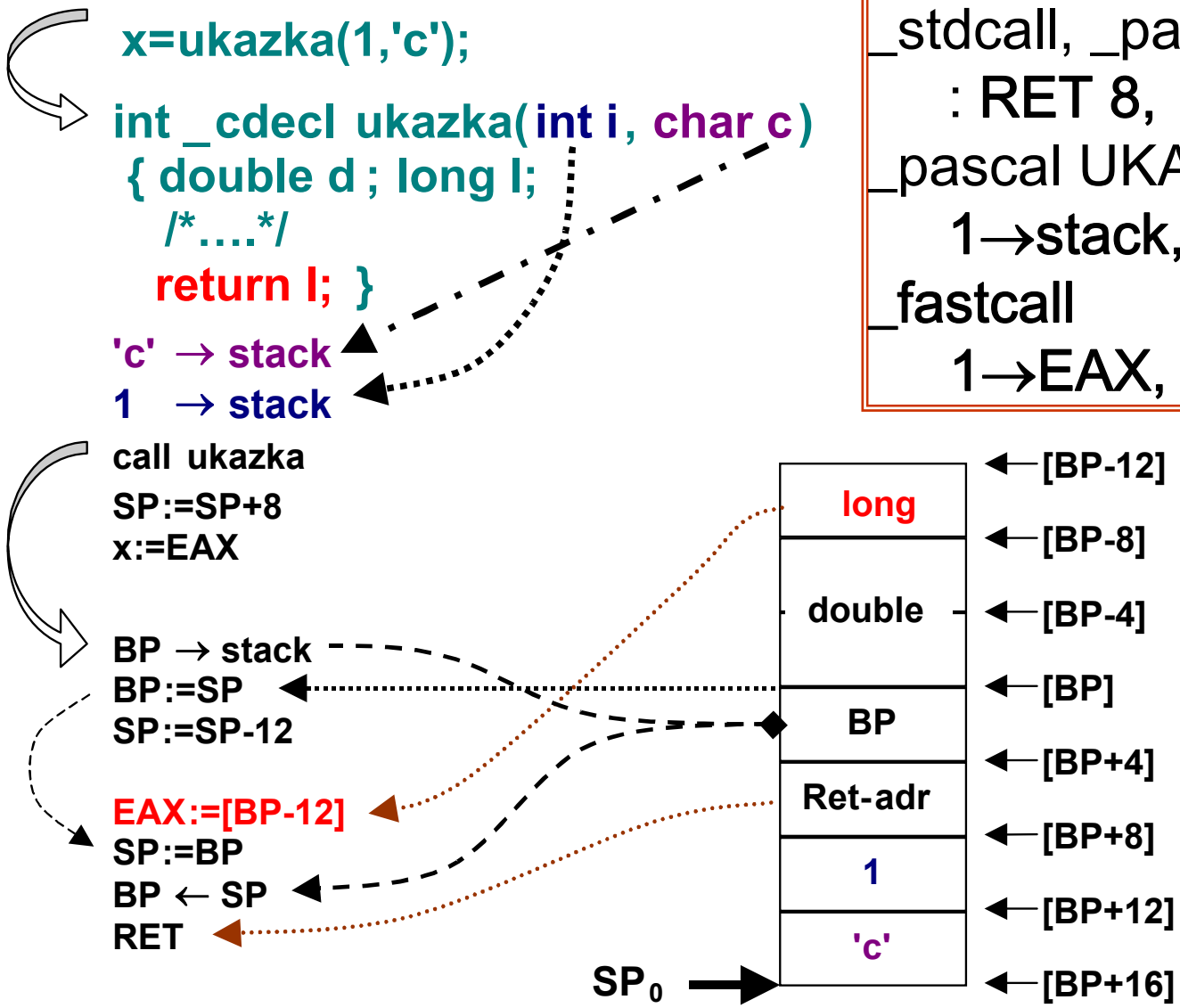
1. P/Invoke
2. Name Mangling
3. Marshaling
4. Attributes

Name Mangling in C++

Name mangling or name decoration
- internal coding identifiers by compiler,

C++ declaration	***Example***	***Note***
void _cdecl fn(int)	_fn	*caller clears stack*
void _stdcall fn(int)	?fn@@int	*fn clears stack*
void _pascal fn(int)	↑fn@@int	*+ upper case*
void _fastcall fn(int)	@fn@@int	*args in EAX,EDX registers*

_cdecl



_stdcall, _pascal, _fastcall
: RET 8,
_pascal UKAZKA
1→stack, 'c'→stack
_fastcall
1→EAX, 'c'→EDX

Name mangling switched off

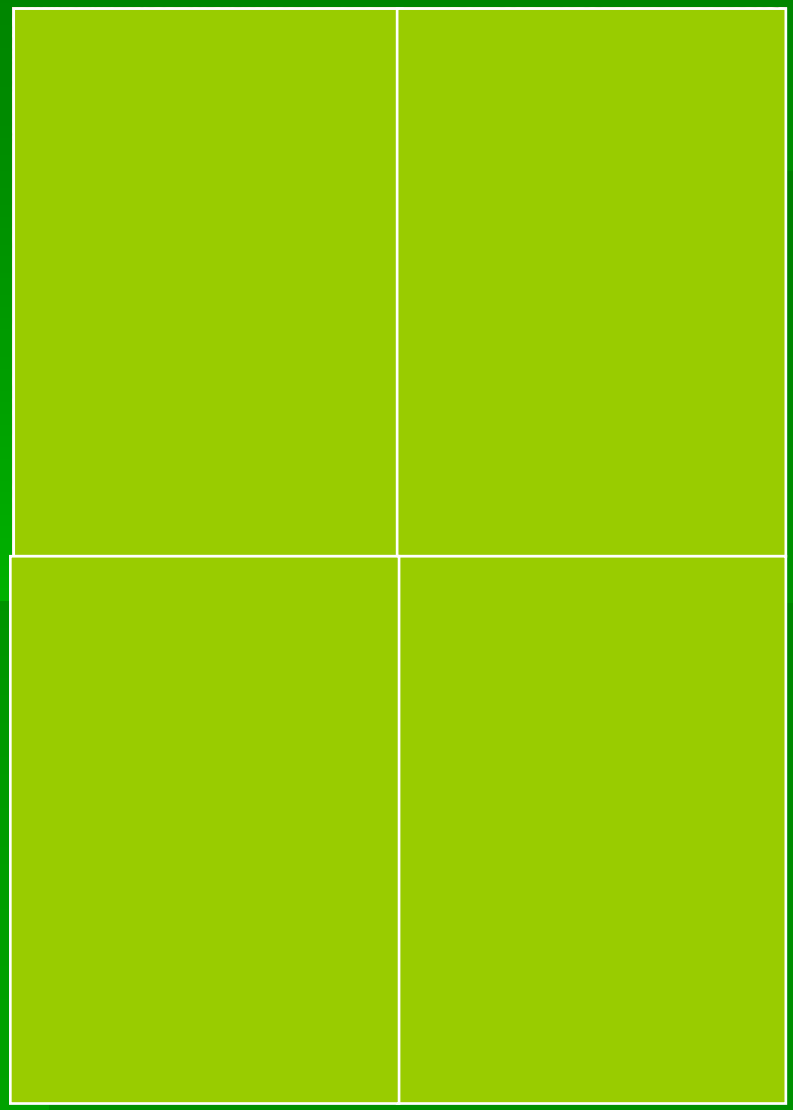
DLLWin32.cpp

```
DLLWIN32_API double Pwd(double d)   { return d*d;  }
```

DLLWin32.h

```
extern "C" DLLWIN32_API double Pwd(double d);
```

Calling DLL's in .NET



1. P/Invoke
2. Name Mangling
3. Marshaling
4. Attributes

Data Marshalling

- .NET and Win32 have different type systems
  - transferring data requires understanding of both systems
  - many simple data types have **consistent** representations
  - more complex types have **inconsistent** representations
  - COM further complicates things

Data Marshaling

- Simple Types
  - Direct Translation
- Simple Objects and ValueTypes
  - Objects that contain simple types
  - Execution engine points at the data of the object
  - Some exceptions
- Complex Objects and Value Types
  - Objects containing other objects need to manually marshal data

Platform Invoke Limitations

- Marshaler can't handle every case
  - Ugly dealing with memory allocation
  - Complex structures can't be marshalled
- Complex P/Invoke is hard to debug
  - Play "convince the marshaller"…
- Can't use C++ objects
- Can waste lots of time and effort
  - Did I say it was tough to debug?
- Solution:
  - Usage of unsafe C# or Managed C++

Pro samostudium

Takto označené snímky

- slouží jako další rozšíření přednášky;
- nebudeme se u nich dlouho zastavovat
- ale nebudou se ani zkoušet.

Marshalling summary

CF Type	C# Type	VB Type	Win32 Type	ByVal	ByRef
Boolean	bool	Boolean	BYTE	Yes	Yes (*)
Int16	short	Short	SHORT	Yes	Yes (*)
Int32	int	Integer	INT32	Yes	Yes (*)
Int64	long	Long	INT64	Yes	Yes (*)
Byte	byte	Byte	BYTE	Yes	Yes (*)
Char	char	Char	WCHAR	Yes	Yes (*)
Single	float	Single	FLOAT	No	Yes (*)
Double	double	Double	DOUBLE	No	Yes (*)
String	string	String	WCHAR *	Yes	No
StringBuilder	StringBuilder	StringBuilder	WCHAR *	Yes	No
User Defined Value Type	struct	Structure	struct or class	No	Yes (*)
class	Class	Class	struct or class	Yes (*)	No
Array of Value Type	T[ ]	T( )	T*/T[ ]	Yes	No

***(read chart left-to-right)**

Marshaling Simple Objects

- Members of classes are always laid out sequentially
- Only classes containing simple types will be automatically marshaled

C#

```
public class Rect {
        int left;
        int top;
        int right;
        int bottom;
}
```

Native Code - C

```
typedef struct _RECT {
        LONG left;
        LONG top;
        LONG right;
        LONG bottom;
} RECT;
```

Data Marshalling and .NET

- Calling Win32 requires .NET runtime assistance
  - runtime responsible for transferring data to/from Win32
  - .NET FW provides a rich, full-featured data transfer layer
    - chooses rich feature set over resource consumption
    - implicitly performs many data transformations
    - marshalling behavior highly customizable
  - .NET CF (NET Compact Framework) provides a very lean data transfer layer
    - chooses minimal resource consumption over feature set
    - performs no data transformation
    - fixed marshalling behavior

Pro samostudium .NET CF marshalling support

- CF provides only fundamental marshalling support
  - by-value marshalling very limited
    - must be 32 bits (4 bytes) or less
    - must be integral
  - by-reference marshalling required for many types
    - requires pointer in Win32 method
    - CF automatically pins reference parameters
      - prevents garbage collector from moving memory
    - classes and structs present special challenge
  - manual marshalling is often required

Pro samostudium Marshalling classes and structs

- Class/struct virtually identical when passed as interop param
  - only difference is how parameter is declared
    - struct passed as **`ref`** param
    - class passed without **`ref`**
    - must be a pointer in Win32 in both cases
- Marshalling class/struct requires marshalling members
  - member value types automatically marshaled
  - member reference types **not** marshaled
    - deep marshalling not implemented in CF
    - must manually marshal or use eVC shim
    - includes string members
  - members always ordered sequentially

Marshaling Complex Objects

- Marshal class
  - Located in System.Runtime.InteropServices
  - Provides advanced functionality to customize marshaling
  - Allows you to get copy managed objects into native code
  - Allows you to read and write raw memory if needed

Marshalling classes and structs: passing as parameters

Win32 structure definition

Win32 function (MyWin32Lib.dll)

```
struct MathData {
  int Val1 ;
  int Val2 ;
  int Result ;
}

extern "C" declspec(dllexport)
void Win32Add(MathData *pData) {
  pData->Result = pData->nVal1 + pData->nVal2;
}
```

Marshalling classes and structs: passing struct as parameter

CF structure

must define param as ref

passing as ref
will be pointer in Win32

```
struct MathStruct{
  int Val1 ;
  int Val2 ;
  int Result ;
}
class Win32Helper{
  [DllImport("MyWin32Lib.dll")]
  void StructAdd(ref MathStruct Data) ;

  int CallAdd(int v1, int v2)  {
    MathStruct s ;
    s.Val1 = v1 ;
    s.Val2 = v2 ;

    Win32Add(ref s) ;

    return s.Result ;
  }
}
```

Marshalling classes and structs: passing class as parameter

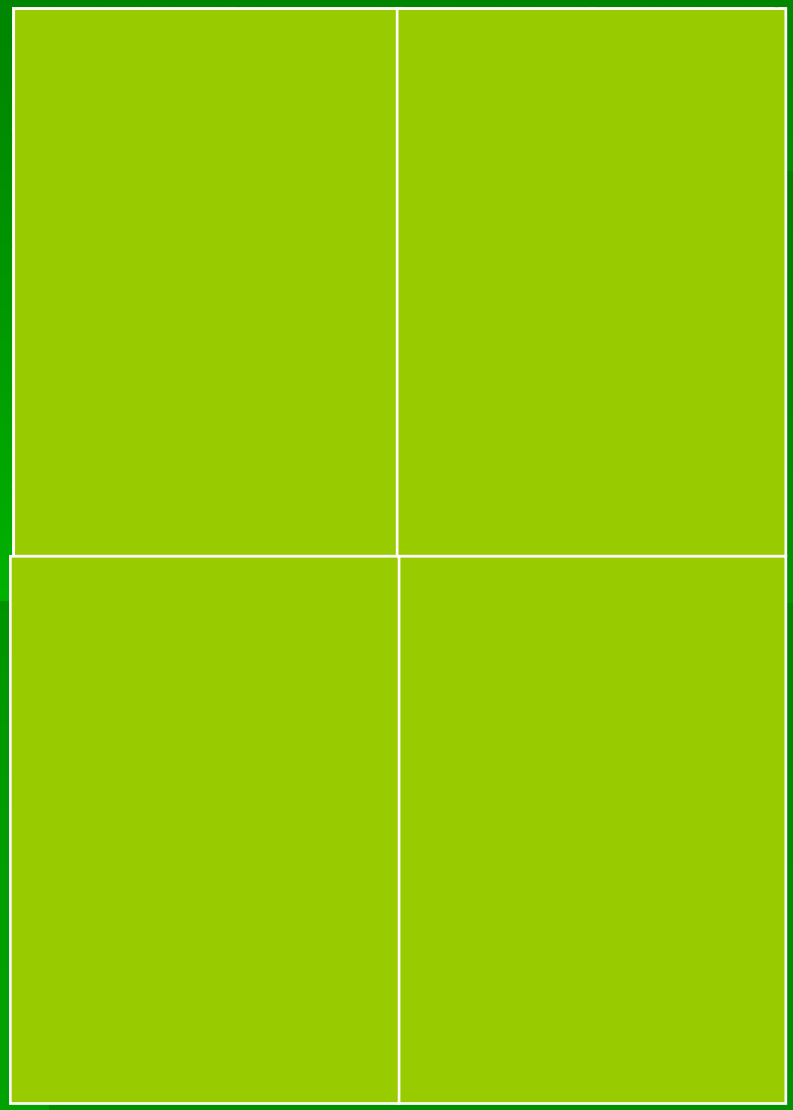
CF class

define param without ref

class always passed as pointer to Win32

```
class MathClass{
  int Val1 ;
  int Val2 ;
  int Result ;
}
class Win32Helper{
  [DllImport("MyWin32Lib.dll")]
  void StructAdd(MathClass Data) ;

  int CallAdd2(int v1, int v2)  {
    MathClass c ;
    c.Val1 = v1 ;
    c.Val2 = v2 ;

    Win32Add(c) ;

    return c.Result ;
  }
}
```

Calling DLL's in .NET



1. P/Invoke
2. Name Mangling
3. Marshaling
4. Attributes

Attributes

- ...appear in square brackets
- ...attached to code elements

```
[HelpUrl("http://SomeUrl/Docs/SomeClass")]
class SomeClass
{
   [WebMethod]
   void GetCustomers() { … }

   string Test([SomeAttr] string param1) {…}
}
```

Attribute Fundamentals

- **Attributes are classes!**

```
class HelpUrl : System.Attribute {
    public HelpUrl(string url) { … }
    …
}
```

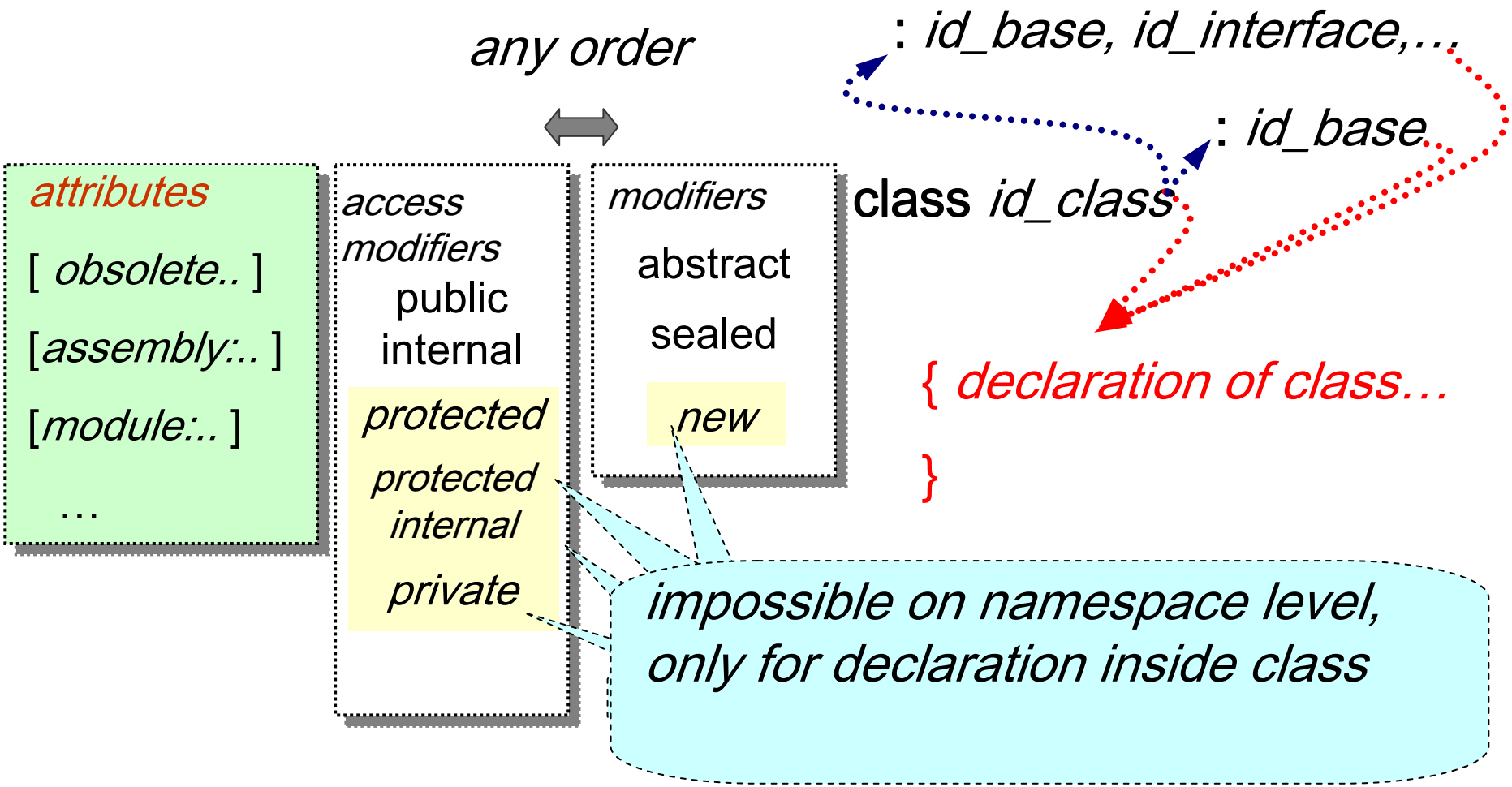
```
[HelpUrl("http://SomeUrl/APIDocs/SomeClass")]
class SomeClass { … }
```

- Easy to attach to types and members

```
Type type = Type.GetType("SomeClass");
object[] attributes =
     type.GetCustomAttributes();
```

- Attributes can be queried at runtime

Syntax of declaration of classes



Calling Into Existing DLLs

- .NET Framework contains attributes to enable calling into existing DLLs
- System.Runtime.InteropServices
  - DLL Name, Entry point, Parameter and Return value marshalling, etc.
- Use these to control calling into your existing DLLs
  - System functionality is built into Framework

Attributes to specify DLL Imports

```
[DllImport("gdi32.dll")]
public static extern
   int CreatePen(int style, int width, int color);
```

```
[DllImport("gdi32.dll",CharSet=CharSet.Auto)]
public static extern
   int GetObject(  int hObject,
                   int nSize,
                   [In, Out] ref LOGFONT lf);
```

Calling unmanaged code with P/Invoke (C#)

- Use *platform invoke* (P/Invoke) to call unmanaged code
  - declare external method using **`DllImport`** attribute
  - mark method “static extern”
  - call method normally
  - can only call non-COM DLL functions

location of **`DllImport`** →

make available →

call →

```
using System.Runtime.InteropServices;

public class DoStuff
{
  [DllImport("coredll.dll")]
  public static extern int SetCursor(int cursorHandle);

  public void Warp(int handle)
  {
    SetCursor(handle);
  }
}
```

Customizing platform call

- `DllImport` uses properties to customize external call
  - can define alternate name using EntryPoint property

real name →
new name →
use new name →

```
using System.Runtime.InteropServices;

public class DoStuff
{
  [DllImport("coredll.dll", EntryPoint = "SetCursor“)]
  public static extern int WarpCursor(int cursorHandle);

  public void Warp(int handle)
  {
    WarpCursor(handle);
  }
}
```

Error Handling

- `DllImport` can make Win32 errors accessible
  - Set **`SetError`** property to True to map in Win32 errors
  - Win32 error available from **`Marshal.GetLastWin32Error`**

map errors →
make call →
win32 err →

```
using System.Runtime.InteropServices;

public class DoStuff
{
  [DllImport("coredll.dll", SetLastError = True)]
  public static extern int SetCursor(int cursorHandle);

  public void Warp(int handle)
  {
    SetCursor(handle);
    if (Marshal.GetLastWin32Error() != 0)
      // handle error
  }
}
```

Calling Into Existing DLLs

- Runtime enables calling “C-Style” functions
- Feature known as “Platform Invoke”
- Attributes define how things work

Which library to use	[DllImport]
How to marshal data	[MarshalAs]
Structure layout in memory	[StructLayout] [FieldOffset]

Using Attributes (contd.)

```
[HelpUrl("http://SomeUrl/MyClass")]
class MyClass  {}

[HelpUrl("http://SomeUrl/MyClass", Tag="ctor")]
class MyClass  {}

[HelpUrl("http://SomeUrl/MyClass"),
 HelpUrl("http://SomeUrl/MyClass", Tag="ctor")]
class MyClass  {}
```

Querying Attributes

- Use reflection to query attributes

```
Type type = typeof(MyClass);

foreach(object attr in type.GetCustomAttributes() )
{
    if ( attr is HelpUrlAttribute )
    {
        HelpUrlAttribute ha = (HelpUrlAttribute) attr;

        myBrowser.Navigate( ha.Url );
    }
}
```

Layout attribute

- C# Union

```
[StructLayout(LayoutKind.Explicit)]
public struct Int16Union
{       [FieldOffset(0)]
        public byte byte0;
        [FieldOffset(1)]
        public byte byte1;

        [FieldOffset(0)]
        public short int16;
        [FieldOffset(0)]
        public ushort uint16;
}
Int16Union var1;
```

```
// v C partially similar
union DATATYPE
{ unsigned char byte0;
  short int int16;
  unsigned short int uint16;
}
var1;
```

StructLayout Attribute

```
[StructLayout ( layout-enum    // Sequential | Explicit | Auto
   [, Pack=packing-size]       // 0, 1, 2, 4, 8, 16, ...
   [, Size=absolute size]
   [, CharSet=charset-enum]    // Ansi | Unicode | Auto
         )]
```

- Pack – controls aligning of fields within structure, if pack is equal to 1 then no gaps.

Atribut Conditional

```
#define _LOG_       // defining symbol _LOG_
/* You can define a symbol, but you cannot assign a value to a symbol. The
   #define directive must appear in the file
   before you use any instructions that are not also directives.
   We can add it into Project Properties:
   → Build → Conditional Compilation Constants  */
using System.Diagnostics; // ConditionalAttribute
class Test
{ // attribute Conditional can be assigned only to methods void
   [Conditional("_LOG_")]
   static public void TestLog(string msg)
   {     Console.WriteLine("Debugging: "+msg);}
}
```

Usage of Conditional

```
/* All TestLog(...) calls are compiled only when
_LOG_  symbol is defined – they need not be deleted
after debugging. */
if(pole[j].CompareTo(pole[j+1])>0)
{ tmp=pole[j]; pole[j]=pole[j+1]; pole[j+1]=tmp;
  Ladeni.TestLog(j.ToString());
/* C# compiler skips all TestLog if _LOG_ is not
defined */
}
```

Example of attributes

```
[ Obsolete("Use Shapes instead of Shape") ]
class Shapes { /*… */ }
class Test3
    { void m() {
      Shape t = new Shape();
/* warning 'Project1.Tvar' is obsolete: Use
   Shape instead of Sh.' */
                    }
    }
```

Attribute Obsolete

```
[Obsolete("Use class CC", true)]    // error
public class C {...}
[Obsolete(" Use class DD")]// warning
public class D {...}
```

Obsolete internally declared as class ObsoleteAttribute

```
public class ObsoleteAttribute : Attribute
 {
    public string Message { get; }
    public bool IsError { get; set; }
    public ObsoleteAttribute() {...}
    public ObsoleteAttribute(string msg) {...}
    public ObsoleteAttribute(string msg, bool error) {...}
 }
```

Obsolete versus ObsoleteAttribute

Pro samostudium

Obsolete and ObsoleteAttribute are the same

```
[Obsolete("message")]
static void Swap(ref IComparable o1,
                 ref IComparable o2) {  /* */ }
[ObsoleteAttribute("message")]
static void Swap(ref IComparable o1,
                 ref IComparable o2) {  /* */ }
[Obsolete]                                  // message = ""
static void Swap(ref IComparable o1,
                 ref IComparable o2) {  /* */ }
```

Attributes

- **Attributes add metadata to your program.** Metadata is information embedded in your program such as compiler instructions or descriptions of data.
- **Attributes exist in two forms:**
  - attributes that are defined in the Common Language Runtime's base class library and
  - custom attributes that you can create, to add extra information to your code. This information can later be retrieved programmatically.
- Attributes can be placed on most any declaration, though a specific attribute might restrict the types of declarations on which it is valid.
- Syntactically, an attribute is specified by placing the name of the attribute, enclosed in square brackets, in front of the declaration of the entity to which it applies. For example, a method with the attribute **DllImport** is declared like this:

```
[System.Runtime.InteropServices.DllImport("user32.dll")]
extern static void SampleMethod();
```

- Many attributes have parameters, which can be either positional, unnamed, or named.

```
[DllImport("user32.dll", SetLastError=false, ExactSpelling=false)]
```

Příště reflexe a jiné…

[Rob Gonsalves]