9. Moduly, BIOS, služby OS, knihovny

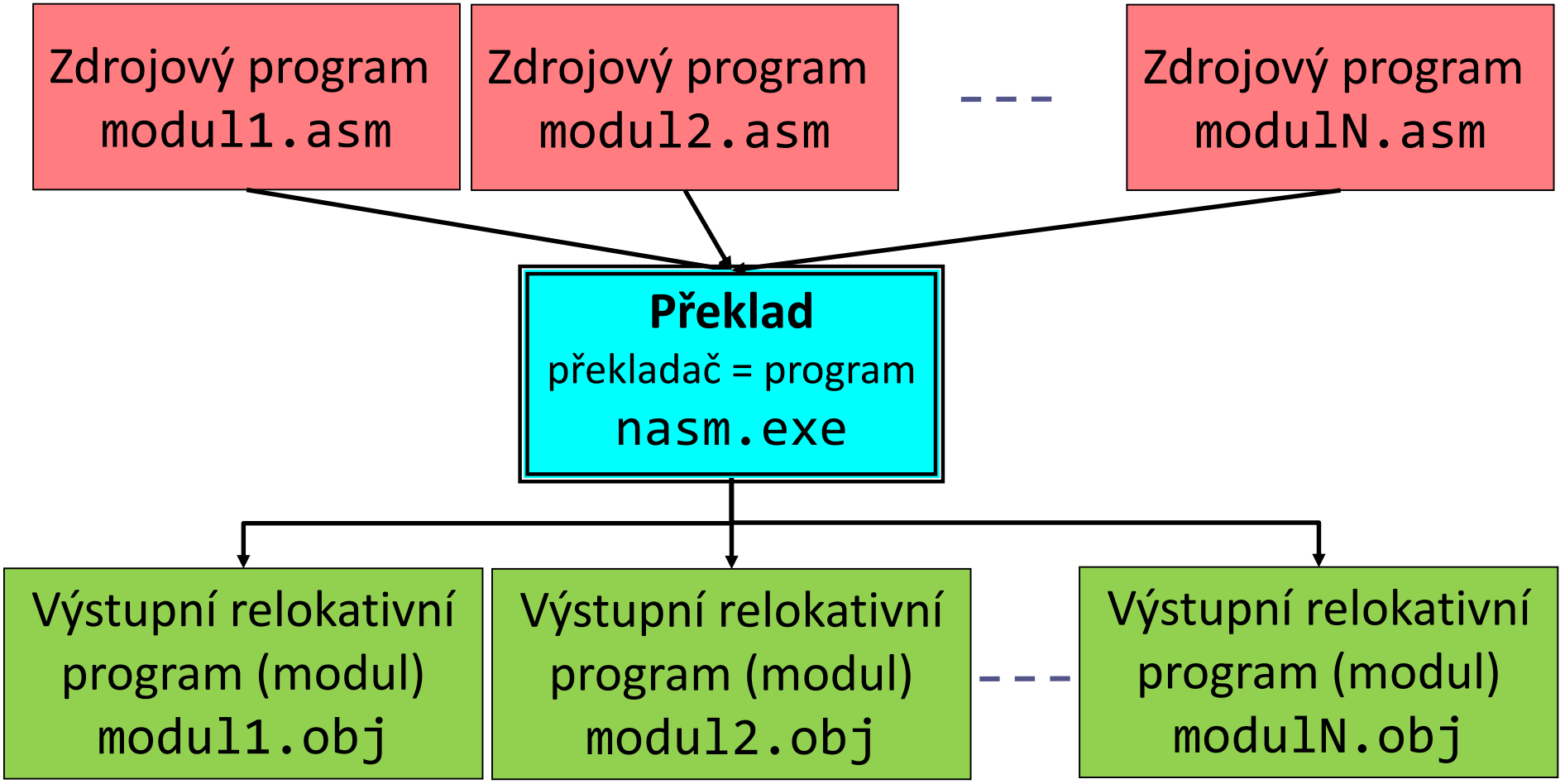
Filip Orság

Modularita – modulární programování

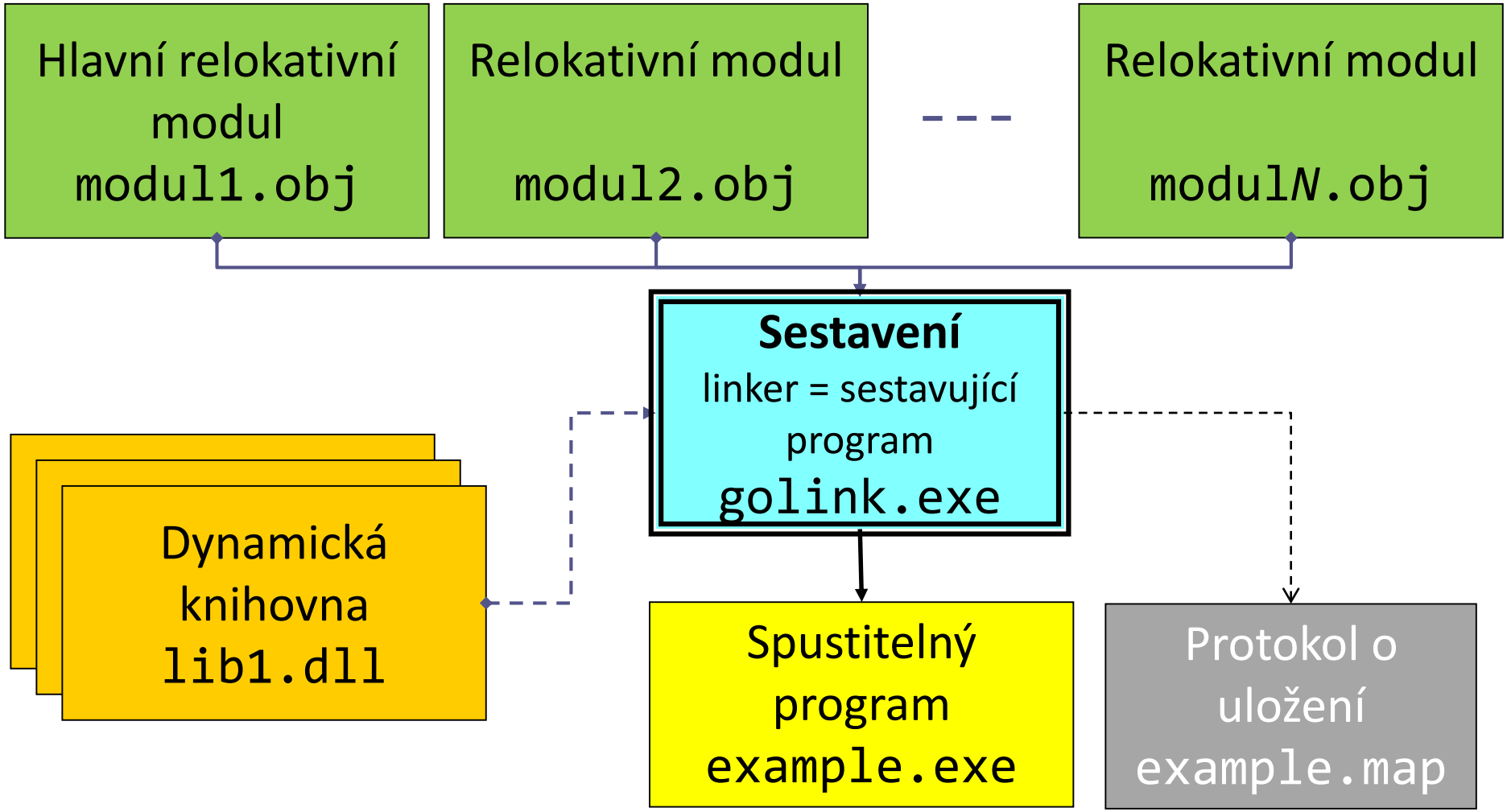
- Běžný požadavek programátorů u projektů většího rozsahu.
- Zpřehledňuje kód rozdělením na menší celky.
- Umožňuje použití tzv. **kódu třetích stran** (tedy ne kód, který si naprogramuji, ale kód, který mi někdo poskytne).
- Kód třetích stran nemusí být poskytnut ve formě zdrojového kódu, ale například jako:
  - **statická knihovna**:
    - relokativní modul = přípona `.obj` (Win), nebo `.o` (Linux)
    - soubor funkcí = přípona `.lib` (Win), nebo `.a` (Linux)
  - **dynamická knihovna**:
    - spustitelný kód = přípona `.dll` (Win), nebo `.so` (Linux)
- Problémy kódu třetích stran:
  - Názvy symbolů = jak se jmenuje funkce, kterou chci volat?
  - Způsob volání = jak předám funkci parametry?
  - Řešení = konvence volání – jednotný standard však neexistuje.

Překlad modulárních programů

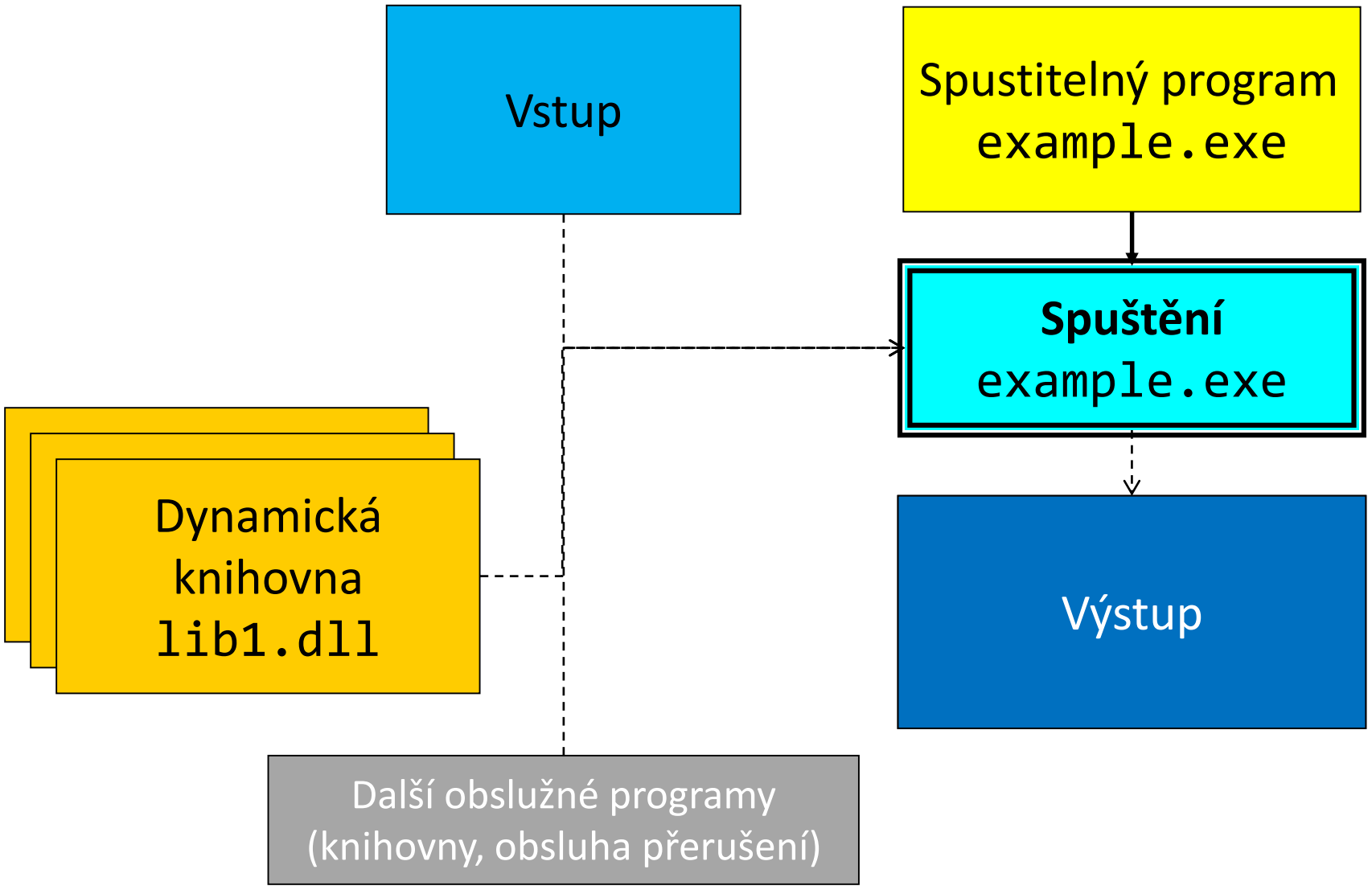
Mám své vlastní moduly = mám zdrojový kód – sám si vše překládám



Sestavení


Hlavní relokativní modul
modul1.obj
Relokativní modul
modul2.obj
Relokativní modul
modulN.obj
Sestavení
linker = sestavující program
golink.exe
Dynamická knihovna
lib1.dll
Spustitelný program
example.exe
Protokol o uložení
example.map

Spuštění


Vstup
Spustitelný program
example.exe
Spuštění
example.exe
Dynamická
knihovna
lib1.dll
Výstup
Další obslužné programy
(knihovny, obsluha přerušení)

Direktivy GLOBAL a EXTERN

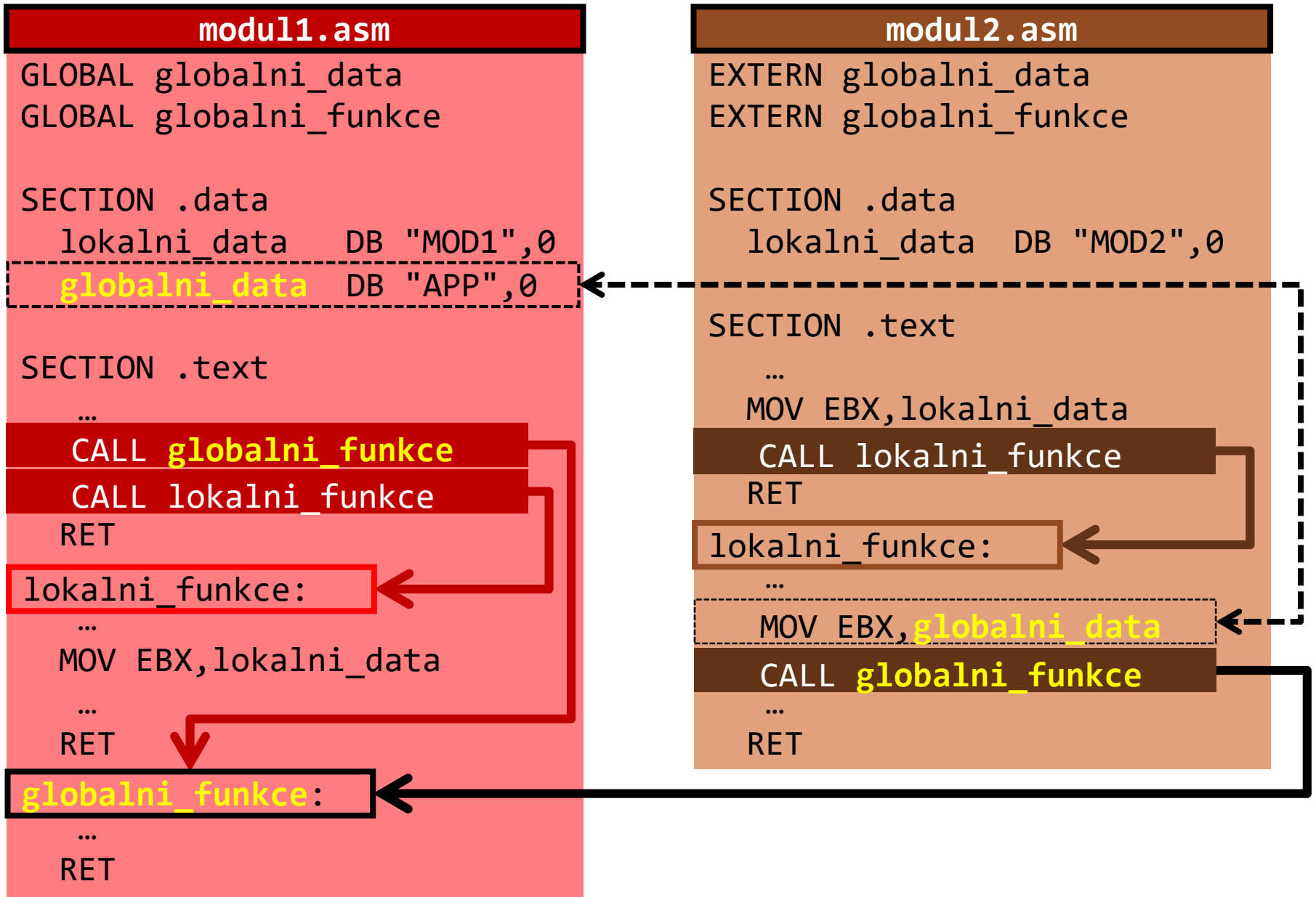
EXTERN *jméno*

- Definuje symbol *jméno* jako externí
  - Vyskytuje se někde jinde – jiný modul, statická nebo dynamická knihovna.
  - Sestavující program poté musí symbol najít.
- Pro jedno jméno lze použít i vícekrát
  - NASM bude další výskyty stejného jméno deklarovaného direktivou EXTERN ignorovat.

GLOBAL *jméno*

- Definuje symbol *jméno* jako globální (tj. viditelný pro **všechny moduly**, které se budou podílet na sestavení).
- Název globálního symbolu musí být v rámci sestavení unikátní = musí být definován pouze jednou, jinak nebude zřejmé, o který ze symbolů se jedná.

Předávání řízení a dat – globální symboly



PŘEDÁVÁNÍ PARAMETRŮ FUNKCÍM

KONVENCE VOLÁNÍ

Předávání parametrů - opakování

- Funkce ke své činnosti potřebují parametry (většinou) => voláme-li funkci, musíme jí předat parametry. Jak?
- Parametry předáváme:
  - **v registrech**
    - problém = málo registrů + musím uchovat jejich obsah (bere čas CPU)
    - určitým způsobem jsou registry globální proměnné
  - **v paměti** (globální proměnné)
    - globální proměnné = jakákoliv změna, kterou na nich funkce provede, se projeví globálně => musíme dávat pozor
  - **na zásobníku** (= speciální paměťový prostor)
    - také předává data v paměti = zásobník
    - předávat lze ukazatel i hodnoty
    - řeší problém globálnosti = jejich změna se neprojeví – po ukončení funkce se parametry z paměti (zásobníku) „uklidí“
- Předávání parametrů přes zásobník je nejčastěji používaná metoda předávání parametrů.

Předávání parametrů – příklad v C

- Mějme funkci, kterou budeme chtít zavolat a předat jí parametry:

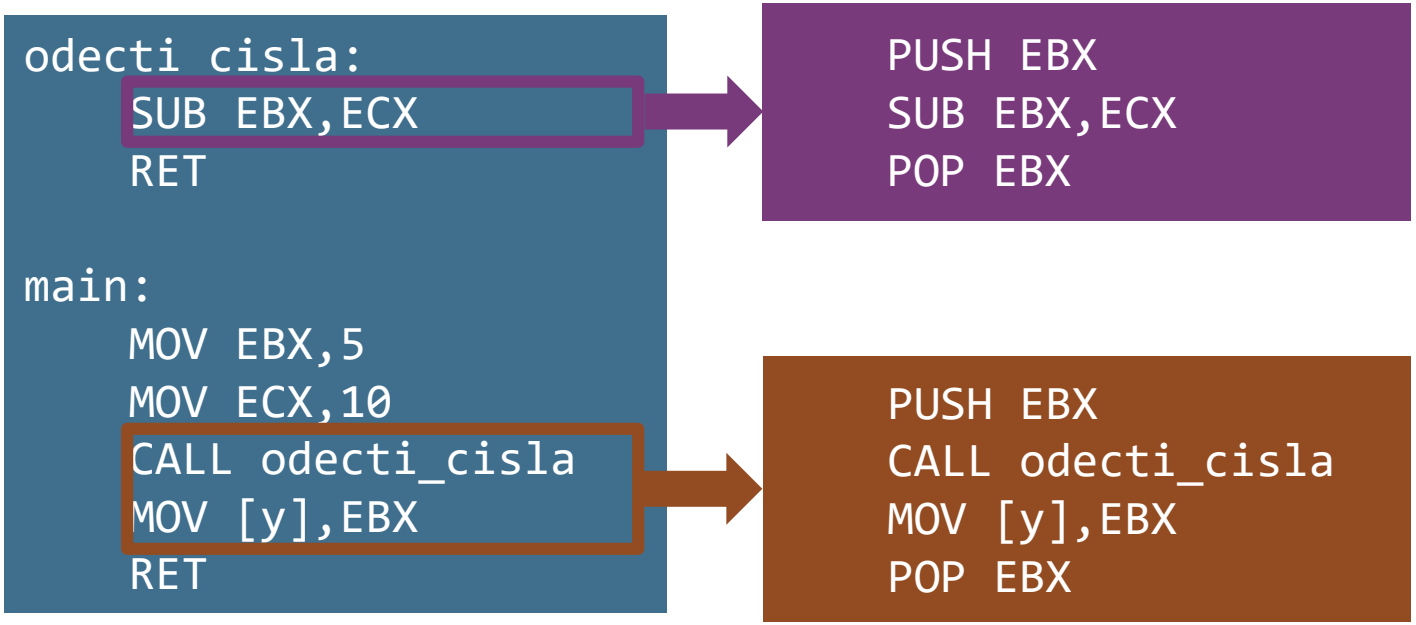
```
int odecti_cisla(int a, int b)
{
    return a - b;
}
```

- Hlavní program, který funkci volá, bude vypadat takto:

```
void main()
{
    int x = 10;
    int y = odecti_cisla(5, x);
}
```

Předávání parametrů v registrech - opakování

- Předchozí program lze zkráceně v asembleru zapsat např. takto:



Problémy:

- Volání funkce změní obsah registru EBX – chtěli jsme to?
- Uložením registru instrukcí PUSH ve funkci ztratím výsledek => musím si schovat obsahy registrů před voláním funkce = musím vědět, které registry funkce mění
- Hodně parametrů = problém s počtem registrů… atd.

Předávání parametrů přes zásobník - opakování

- Jiný příklad zápisu programu – předání parametrů na zásobníku:

```
odecti_cisla:
    MOV EAX,[ESP+4]
    SUB EAX,[ESP+8]
    RET 8
```

```
main:
    PUSH DWORD 10
    PUSH DWORD 5
    CALL odecti_cisla
    MOV [y],EAX
    RET
```

Problémy:

- Pokud ve funkci použijeme instrukci PUSH, budeme těžké sledovat „pozici“ parametru. Řešení = **zásobníkový rámec**.
- Musíme stanovit pořadí parametrů ukládaných na zásobník (zleva doprava nebo zprava doleva).
- Musíme stanovit, jak předat výsledek (buď přes zásobník nebo v registru – běžně se výsledek vrací například v registru EAX).
- Musíme stanovit, kdo zásobník „uklidí“ (funkce po sobě uklidí nebo ten, kdo ji volá).

Zásobníkový rámec - opakování

Stanovení záchytného bodu („záložka“ v registru **EBP**)

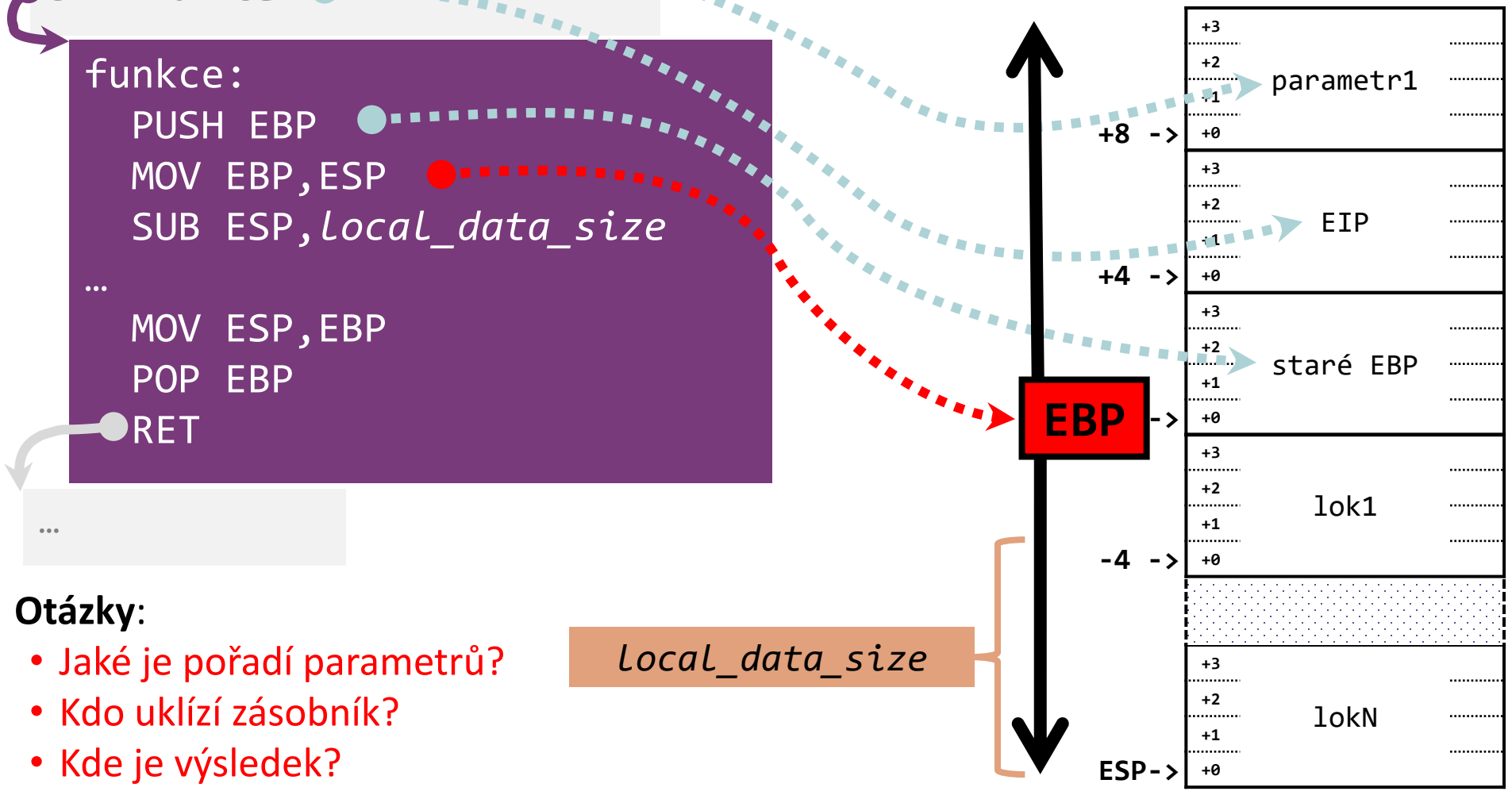
Rezervování místa na lokální data

Obnovení původního stavu zásobníku

```
PUSH parametr1
CALL funkce
```

```
funkce:
   PUSH EBP
   MOV EBP,ESP
   SUB ESP,local_data_size
…
   MOV ESP,EBP
   POP EBP
   RET
```

```
…
```



Otázky:

- Jaké je pořadí parametrů?
- Kdo uklízí zásobník?
- Kde je výsledek?

Volání funkcí – postup – opakování

- volající **uloží na zásobník parametry** v dohodnutém pořadí (dle konvence volání), například instrukcí PUSH
- volající použije instrukci CALL a volaný přebírá řízení
  - funkce s parametry vytvoří zásobníkový rámec
    1. uloží obsah registru EBP na zásobník
    2. do registru EBP zkopíruje obsah registru ESP
    3. přístup k parametrům pomocí `[EBP + posunutí]`
    4. lokální data = snížíme obsah registru ESP o počet bytů, které tato data zabírají (= rezervujeme si prostor na zásobníku = v paměti)
    5. na lokální data se odkazujeme pomocí `[EBP – posunutí]`
    6. **výsledek funkce se předává v registru AL, AX, EAX nebo EDX:EAX (dle velikosti návratového typu), reálné číslo se vrací v FPU v registru ST0**
    7. jakmile volaný ukončí činnost, obnoví hodnotu ESP a EBP a provede návrat instrukcí RET nebo RET *N* (dle konvence volání)
- Po návratu z funkce je nutné uklidit po volání zásobník = odstranit parametry, které jsme tam vložili – buď uklízí volaný při návratu z funkce (instrukcí RET *N*) nebo volající (např. zvýšením hodnoty ukazatele ESP, např. `ADD ESP,12`).

Konvence volání funkcí

Pojmy

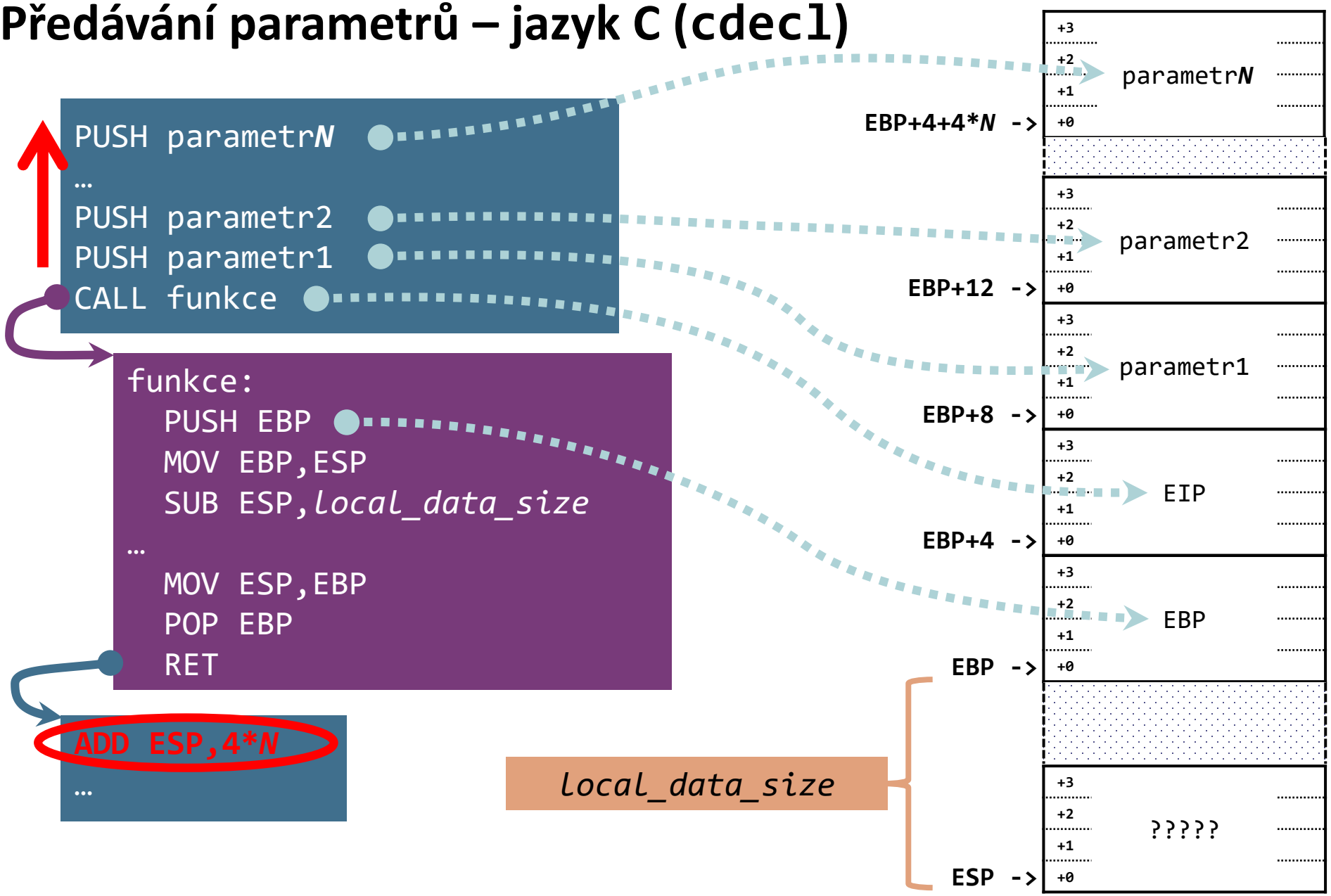
- **volající** (*calling*): ten, kdo volá funkci instrukcí CALL
- **volaný** (*called*, *calee*): ten, kdo je volán = volaná funkce

Konvence volání – `pascal`, `cdecl`, `stdcall`, `fastcall`, a další

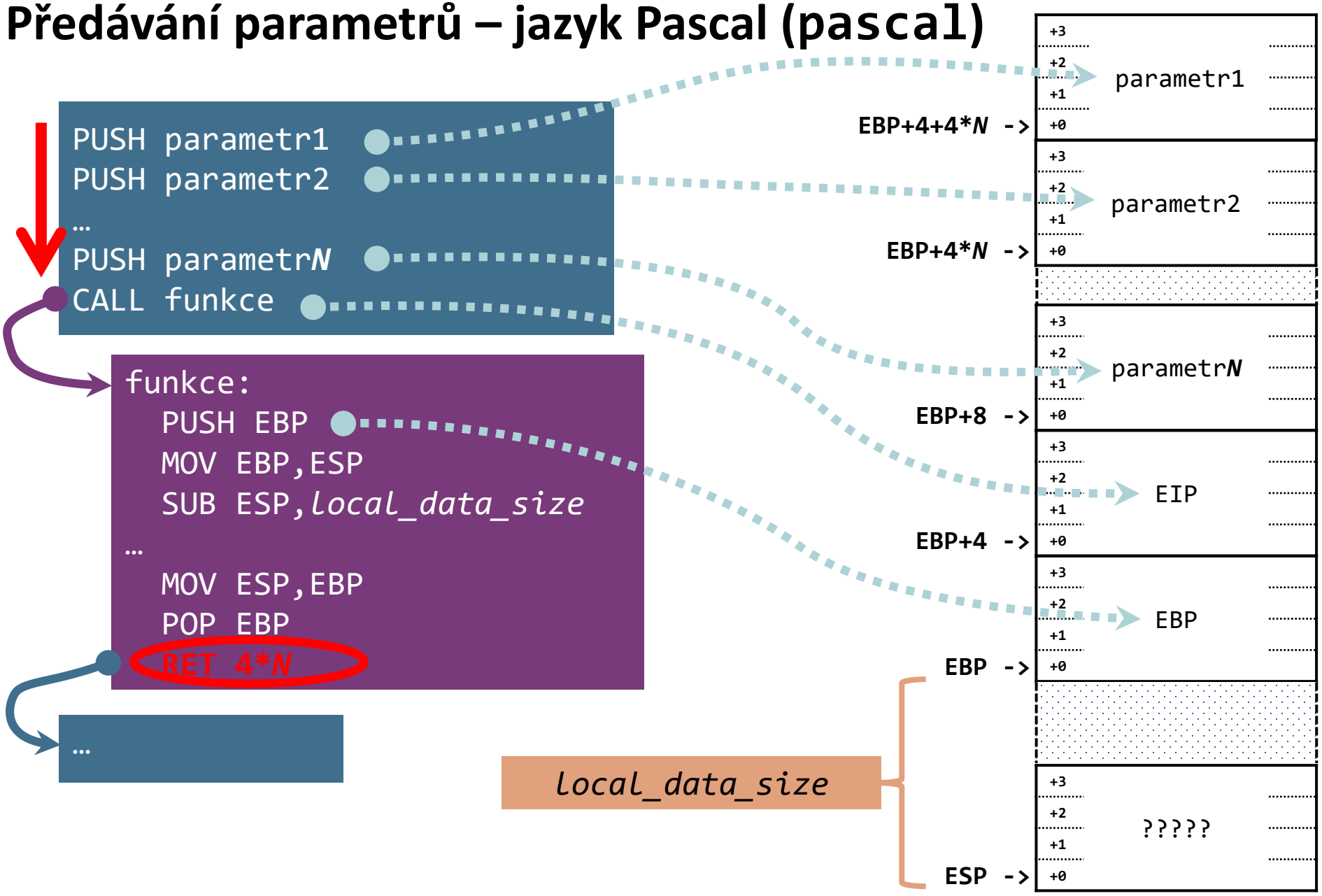
- definuje způsob předání parametrů: **zleva doprava** nebo **zprava doleva**
- určuje, kdo ze zásobníku „uklidí“ parametry
- definuje dekoraci jmen symbolů: přidávají se podtržítka, zavináče a čísla

Konvence volání	Parametry funkce	Zásobní k uklízí	Dekorace jmen (pro jazyk C)	Použito v
`pascal`	zleva doprava	volaný	`symbol`	Pascal
`cdecl`	zprava doleva	volající	`_symbol`	Jazyk C
`stdcall`	zprava doleva	volaný	`_symbol@4`	Win42 API
`fastcall`	první dva parametry v ECX a EDX, zbytek zprava doleva	volaný	`@symbol@8`	různé

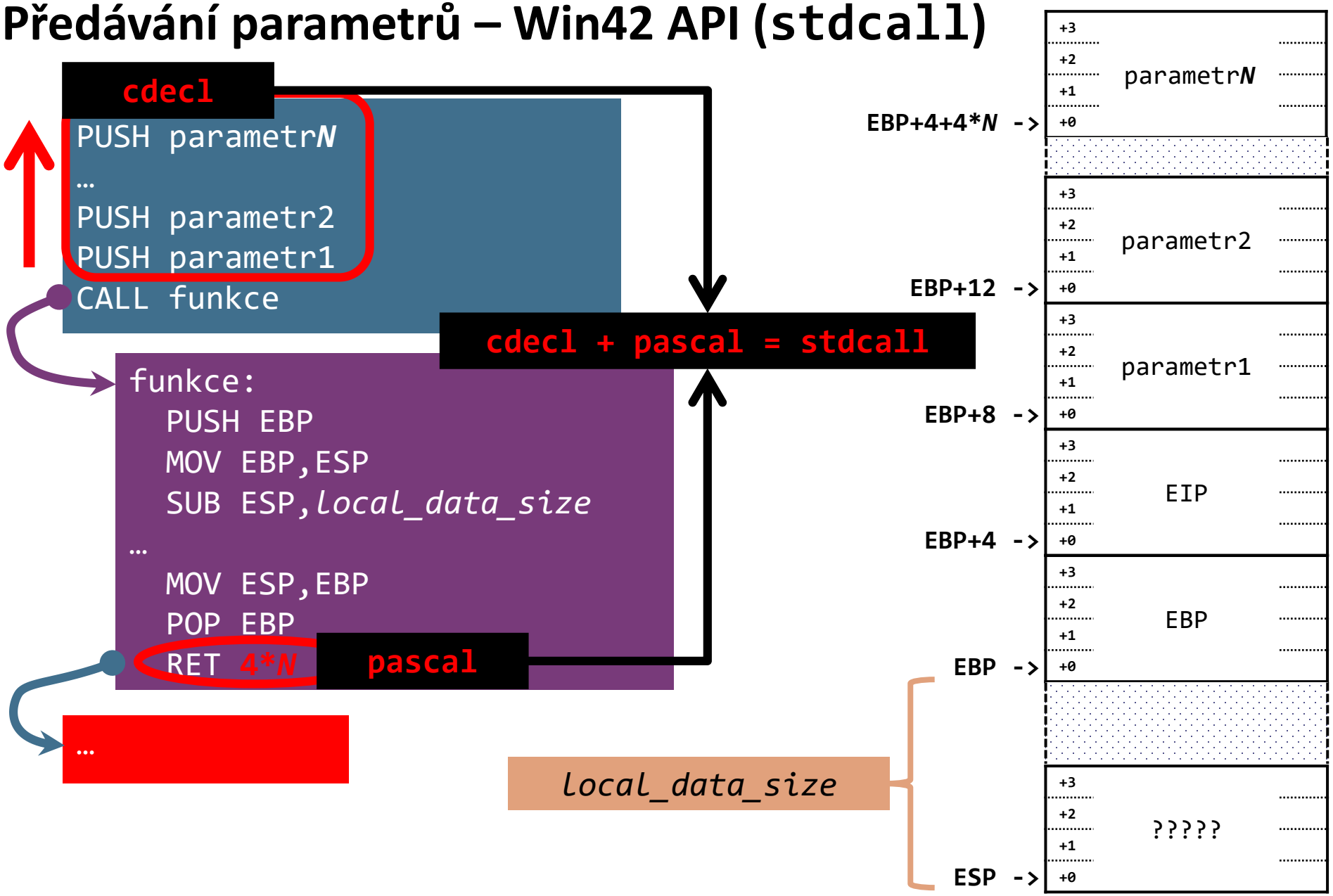
Předávání parametrů – jazyk C (cdecl)



Předávání parametrů – jazyk Pascal (pascal)



Předávání parametrů – Win42 API (stdcall)



Předávání parametrů – příklad 1

Funkce	Volání funkce

`int odecti(int a, int b)`
`{`
`  return a - b;`
`}`
	`void main()`
	`{`
	`  int x = 10;`
	`  int y = odecti(5, x);`
	`}`

```
odecti_cdecl:
    PUSH EBP
    MOV EBP,ESP
    MOV EAX,[EBP+8]
    SUB EAX,[EBP+12]
    POP EBP
    RET
```

```
main:
    PUSH DWORD [x]
    PUSH DWORD 5
    CALL odecti_cdecl
    ADD ESP,8
    MOV [y],EAX
    RET
```

```
odecti_pascal:
    PUSH EBP
    MOV EBP,ESP
    MOV EAX,[EBP+12]
    SUB EAX,[EBP+8]
    POP EBP
    RET 8
```

```
main:
    PUSH DWORD 5
    PUSH DWORD [x]
    CALL odecti_pascal
    MOV [y],EAX
    RET
```

```
odecti_stdcall:
    PUSH EBP
    MOV EBP,ESP
    MOV EAX,[EBP+8]
    SUB EAX,[EBP+12]
    POP EBP
    RET 8
```

```
main:
    PUSH DWORD [x]
    PUSH DWORD 5
    CALL odecti_stdcall
    MOV [y],EAX
    RET
```

Předávání parametrů – příklad 2

Program v jazyce C a analogie v asembleru (konvence volání `stdcall`)

```c
int deleni(int x, int y, int *ptrZb)
{
  int tmp = x/y;
  *ptrZb = x%y;
  return tmp;
}
```

```c
void main()
{
    int a = 10;
    int podil = 0;
    int zbytek = 0;
    podil = deleni(a, 2, &zbytek);
}
```

Zásobníkový rámec

```
[EBP+16] ≈ ptrZb
[EBP+12] ≈ y
[EBP+8]  ≈ x
[EBP+4]  ≈ EIP
[EBP]    ≈ staré EBP
[EBP-4]  ≈ tmp
```

```
deleni:
    PUSH EBP
    MOV EBP,ESP
    SUB ESP,4
    MOV EAX,[EBP+8]          int tmp=x/y;
    CDQ
    IDIV DWORD [EBP+12]
    MOV [EBP-4],EAX
    MOV EAX,[EBP+16]         *ptrZb=x%y;
    MOV [EAX],EDX
    MOV EAX,[EBP-4]          return tmp;
    MOV ESP,EBP
    POP EBP
    RET 12  ; 3 parametry = 3*4 = uklidit 12 B
```

```
main:
    PUSH zbytek     ; ukazatel na zbytek
    PUSH DWORD 2    ; konstantní hodnota
    PUSH DWORD [a]  ; hodnota z proměnné
    CALL deleni
    MOV [podil],EAX
    RET

SECTION .data
    a      DD 10
    podil  DD 0
    zbytek DD 0
```

BIOS

BIOS

- BIOS (*Basic Input/Output System*) = poskytuje standardizovanou množinu služeb na nejnižší úrovni
  - firmware počítače = základní software nejnižší úrovně = vstupně/výstupní funkce, základní funkce pro práci s HW na desce nebo připojeného k desce
  - uložen nejčastěji v ROM, EEPROM nebo flash paměti na základní desce
  - předávání parametrů přes registry, volání = použití instrukce přerušení INT
- Dnes využíván především při startu počítače
  - inicializace a konfigurace HW
  - zavádění operačního systému
  - nahrazován novějším systémem EFI (*Extensible Firmware Interface*)
- V chráněném režimu moderních OS není přístupný
- Ve virtuálním stroji je BIOS emulován
- Ve 32bitových aplikacích nezkoušejte – kód nebude funkční – způsobí ukončení vaší aplikace

Služby BIOSu

INT	Popis
09h	IRQ1: Volá se při stisku, držení a uvolnění klávesy
0Bh	IRQ3: Volají sériové porty 2 a 4 (COM2/4)
0Ch	IRQ4: Volají sériové porty 1 a 3 (COM1/3)
0Dh	IRQ5: Volá řadič pevného disku (XT) nebo druhý paralelní port LPT2 (AT)
0Eh	IRQ6: Volá řadič disketové jednotky
0Fh	IRQ7: Volá paralelní port LPT1 (tiskárna)
10h	Služby zobrazení (základní video režimy a vstup/výstup)
11h	Seznam zařízení
12h	Velikost konvenční paměti
13h	Nízkoúrovňové služby pevného disku
14h	Služby sériového portu
15h	Různé systémové služby
16h	Služby klávesnice
17h	Služby tiskárny
1Ah	Služby hodin reálného času (RTC)

Příklad přerušení BIOSu (INT 0x10)

- přerušení 0x10 obsluhuje zobrazovací zařízení
  - = video – výstup na obrazovku monitoru
- mnoho různých činností: výpis znaku, nastavení atributu, nastavení rozlišení, kreslení bodu …
- **Příklad**: výpis znaku s barevnými atributy – služba 0x09 přerušení 0x10
- Vstup:
  - AH = 0x09, AL = hodnota ASCII znaku, který má být vypsán
  - BH = číslo video stránky, BL = číslo barvy
  - CX = počet opakování výpisu znaku
- Výstup:
  - nemění registry, na obrazovce je CX-krát vypsán barvou BL znak v AL

```
MOV AH,0x09
MOV AL,'A'
MOV BH,0
MOV BL,15
MOV CX,5
INT 0x10
```

Služby operačního systému

Služby (jádra) operačního systému

- **Služby operačního systému** = speciální množina funkcí především pro práci s procesy a hardware
- Nízkoúrovňové, nutné pro běh OS
- Typy služeb jádra OS:
  - spouštění aplikací
  - vstupně výstupní operace
  - práce se soubory
  - komunikace (mezi procesy, s vnějším světem, …)
  - detekce chyb
  - alokování zdrojů
  - ochrana (sebe i programů)
- Služby jádra jsou přístupné například jako
  - speciální softwarové přerušení (MS DOS = `INT 0x21`, Linux = `INT 0x80`)
  - speciální volání instrukcí SYSENTER (SYSCALL)
  - dynamické knihovny (Windows), které zakrývají nízkoúrovňové služby

Příklad služby MS DOS (INT 0x21)

- přerušení 0x21 poskytuje elementární funkce systému MS DOS
- MS DOS = operační systém, který pracuje v reálném režimu, ve Windows nepůjde zavolat a spustit příkaz
- **Příklad**: výpis řetězce na obrazovku – služba číslo 9 přerušení 0x21
- Vstup:
  - AH = 9
  - DS:DX = adresa – ukazatel na řetězec zakončený znakem „$“
- Výstup:
  - AL = 0x24, na obrazovce je vypsán řetězec, který leží na adrese dané v DS:DX

```
SECTION .data
    pText DB “Ahoj!$”
SECTION .text
    MOV DX,offset pText
    MOV AH,9
    INT 0x21
```

Příklad služby OS Linux (INT 0x80)

- přerušení 0x80 poskytuje elementární funkce Linuxu
- na 64bitovém systému lze použít SYSCALL
- **Příklad**: výpis řetězce na obrazovku – služba číslo 4 přerušení 0x80
- Vstup:
  - EBX = file handle (1 = standardní výstup STDOUT)
  - ECX = adresa – ukazatel na řetězec
  - EDX = počet znaků, které mají být vypsány
- Výstup:
  - nemění registry, na obrazovce/v souboru/jinde (určuje EBX) je vypsán řetězec, který leží na adrese dané v ECX a má EDX znaků

```
SECTION .data
    pText DB “Ahoj!”
SECTION .text
    MOV EAX,4
    MOV EBX,1
    MOV ECX,pText
    MOV EDX,5
    INT 0x80
```

Knihovny

Knihovna = speciální softwarový modul obsahující kód s funkcemi, které můžeme využít pro své účely

- vlastní knihovna nebo práce někoho jiného (knihovny třetích stran)
- obsahuje obvykle často používané, speciální funkce
  - např. výpočty ve víceslovní aritmetice, I/O konverze, emulace výpočtů s reálnými čísly, zpřístupnění hardware, služby operačního systému, apod.
- dvojí typ knihoven
  - **statické knihovny** – musí být k dispozici při sestavení programu a jejich kód se běžně vkládá přímo do výsledného spustitelného programu
    - Windows = `*.LIB`, `Linux = *.a`
  - **dynamické knihovny** – musí být k dispozici při spuštění nebo za běhu programu, jejich kód je v externím souboru
    - Windows = `*.DLL`, `Linux = *.so`
- Knihovny DLL (*Dynamic-Link Library*)
  - linkovaná s programem za jeho běhu
  - soubory s příponou `*.DLL` obsahují informace o knihovně, především **seznam funkcí**, které poskytují

NASM a knihovny DLL

Použití DLL = splnění následujících kroků:

- musíme informovat NASM, že funkce (symbol) je externí (direktiva **EXTERN**)
- NASM dá informaci sestavujícímu programu, že daný symbol musí hledat v některém jiném z uvedených modulů při sestavování
- musíme informovat sestavující program, kde najde všechny externí funkce (tj. musíme uvést seznam všech souborů, kde se nachází funkce, které voláme)

Příklad:

- funkce `void ExitProcess(unsigned int uExitCode)` je funkce Win32 API a je v knihovně `kernel32.dll`
- informujeme NASM, že ji chceme použít takto:

```
EXTERN ExitProcess
```

- použití funkce (= její volání) a předání parametrů dle standardu volání, který daná funkce používá (STDCALL pro funkce Win32 API):

```
PUSH dword <exit code>
CALL ExitProcess
```

- informujeme linker, že používáme funkce z knihovny `kernel32.dll`:

```
golink app.obj /console /entry:start kernel32.dll
```

Služby OS jako knihovny – Win32 API

- Soubor DLL knihoven poskytujících stovky funkcí jádra OS
- **Win32 API** *(Windows 32bit Application Programming Interfaces)*
  - Programátorské rozhraní pro přístup ke službám operačního systému = množina funkcí pro práci s aplikacemi a hardware
  - Zahrnuje základní a pokročilé služby, rozhraní grafických zařízení (GDI), uživatelské rozhraní (GUI – funkce dialogových oken, běžných prvků a windows shell), síťové služby, podpora práce s webem, multimédii a grafikou… a mnoho dalších knihoven.
  - Z hlediska programátorského – je k dispozici přibližně:
    - ≈7000 konstant, stovky struktur, ≈1000 funkcí
    - používá konvenci volání **STDCALL**
  - Detailní informace o funkcích naleznete na *http://msdn.microsoft.com*

Ukázka funkce Win32 API - GetCommandLineA

```
LPTSTR GetCommandLineA(void);
```

Funkce vrací ukazatel na řetězec zakončený nulou (ASCIIZ) s textem příkazové řádky. Součástí řetězce je i jméno samotného programu. Spustíte-li například program takto:

```
muj_program.exe parametr1 parametr2
```

Pak bude řetězec obsahovat právě tento text.

```
SECTION .data
    ptrCmdLine DD 0

SECTION .text
    …
    CALL GetCommandLineA
    MOV [ptrCmdLine],EAX  ; EAX obsahuje ukazatel na řetězec
    …                     ; můžeme uložit do paměti pro další použití
```

Standardní knihovna jazyka C

Standardní knihovna jazyka C

- sada standardů pro knihovny a hlavičkové soubory programovacího jazyka C dle normy ISO C
- obsahuje deklarace funkcí jazyka C zajišťující vstup a výstup, práci s textovými řetězci apod.
- standardizované rozhraní jazyka C popsané normou
- konkrétní implementace závisí na operačním systému, jediné, co je shodné, jsou hlavičky a přibližně i chování funkcí
- *run-time* knihovny jazyka C poskytují funkce knihovny a další součásti potřebné pro fungující program (inicializace volání funkce *main*, emulace FPU, apod.)
- při volání těchto funkcí typicky používáme konvenci **CDECL**
- seznam standardních funkcí viz například *https://en.wikipedia.org/wiki/C_standard_library*

Jak použít funkci v asembleru?

- Pokud známe deklaraci, pak je vše jednoduché
  - musíme znát použitou konvenci volání (CDECL v případě standardní knihovny jazyka C)
- Datové typy nás příliš nezajímají – vše je 32bitová hodnota
- Např. funkce

```
int funkce( const char *s, int a, int *ptrB );
```

  - příklad použití v jazyce C:

```
int b = 10, a = funkce(„Ahoj“, 12342, &b);
```

  - příklad použití v asembleru:

```
SEGMENT .data
    a DD -1
    b DD 10
    s DB „Ahoj“,0

SEGMENT .text
    PUSH b              ; předání odkazem
    PUSH dword 12342    ; předání hodnotou
    PUSH s              ; předání odkazem
    CALL funkce
    ADD ESP,12
    MOV [a],EAX         ; výsledek je v EAX
```

Vybrané funkce standardní knihovny jazyka C

- Níže uvedené funkce znáte z IZP…
- Práce s konzolou/terminálem (výstup)

```
int printf( const char *format, ... );
```

- Práce s pamětí (přidělení a uvolnění)

```
void* malloc( size_t size );
void  free  ( void* ptr );
```

- Práce se souborem (otevření, čtení a zavření)

```
FILE *fopen ( const char *filename, const char *mode)
char *fgets ( char *str, int count, FILE *stream );
int   fclose( FILE *stream );
```

- Práce s řetězci (porovnání a zjištění délky)

```
size_t strlen( const char *str );
```

Příklad využití standardní knihovny jazyka C v asembleru

- Následující kód v jazyce C přepíšeme do asembleru:

```c
// Soubor: stdlibtest.c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
int main(void) {
    char* pBuffer = (char *)malloc(100);
    FILE* pFile = fopen("stdlibtest.asm", "r");
    if (pFile) {
        while (fgets(pBuffer, 100, pFile) != NULL) {
            if (strlen(pBuffer) <= 2) printf("; prázdný řádek\n");
            else if (pBuffer[0] != ';') printf("%s", pBuffer);
        }
        fclose(pFile);
    } else {
        printf("Chyba při otvírání souboru!\n");
    }
    free(pBuffer);
}
```

Příklad využití knihovny jazyka C v asembleru (1)

- vložíme „rw42“ a informujeme NASM o externích funkcích
- definujeme řetězce, které se v programu vyskytnou
- definujeme proměnné

```
%include 'rw42-2015.inc'
extern malloc
extern fopen
extern fgets
extern strlen
extern printf
extern strcmp
extern fclose
extern free
```

```
segment .data
    sFileName       DB "stdlibtest.asm",0
    sFileOpenMode   DB "r",0
    sEmptyLine      DB "; prázdný řádek",EOL,0
    sNOP            DB "NOP",EOL,0
    sNIC            DB "; NIC",EOL,0
    sFormatString   DB "%s",0
    sError          DB "Chyba při otvírání"
                    DB " souboru!",EOL,0

    pFile           DD 0
    pBuffer         DD 0
```

Příklad využití knihovny jazyka C v asembleru (2)

- naprogramujeme kód

```
segment .text
; int main(void) {
main:
;    char* pBuffer = (char *)malloc(100);
    PUSH dword 100
    CALL malloc
    ADD ESP,4
    MOV [pBuffer],EAX
;    FILE* pFile = fopen("stdlibtest.asm", "r");
    PUSH sFileOpenMode
    PUSH sFileName
    CALL fopen
    ADD ESP,8
    MOV [pFile],EAX
;    if (pFile) {
    CMP EAX,0
    JE else
```

Příklad využití knihovny jazyka C v asembleru (3)

```
;       while (fgets(pBuffer, 100, pFile) != NULL) {
while:
    PUSH dword [pFile]
    PUSH dword 100
    PUSH dword [pBuffer]
    CALL fgets
    ADD ESP,12
    CMP EAX,0
    JZ endwhile
;           if (strlen(pBuffer) <= 2) printf("; prázdný řádek\n");
    PUSH dword [pBuffer]
    CALL strlen
    ADD ESP,4
    CMP EAX,2
    JNLE elseif_skipcomment

    PUSH sEmptyLine
    CALL printf
    ADD ESP,4
    JMP while
```

Příklad využití knihovny jazyka C v asembleru (4)

```
;            else if (pBuffer[0] != ';') printf("%s", pBuffer);
elseif_skipcomment:
    MOV EAX,[pBuffer]
    CMP byte [EAX],';'
    JE while

    PUSH dword [pBuffer]
    PUSH sFormatString
    CALL printf
    ADD ESP,8
    JMP while
;        }
endwhile:
;        fclose(pFile);
    PUSH dword [pFile]
    CALL fclose
    ADD ESP,4
    JMP endif
```

Příklad využití knihovny jazyka C v asembleru (5)

```
;    } else {
else:
;        printf("Chyba při otvírání souboru!\n");
    PUSH sError
    CALL printf
    ADD ESP,4
;    }
endif:
;    free(pBuffer);
    PUSH dword [pBuffer]
    CALL free
    ADD ESP,4
;}
    RET
```