Obsluha přerušení, výjimek a systémových volání.

Thursday, May 30, 2013 8:26 AM

Zdroj: http://www.kiv.zcu.cz/~safarikj/vyuka/os/prednasky/prednaska05.pdf

http://stackoverflow.com/questions/9968028/returning-from-kernel-mode-to-user-mode

Přerušení

metoda pro asynchronní obsluhu událostí, kdy procesor přeruší vykonávání sledu instrukcí, vykoná obsluhu přerušení a pak pokračuje.

- **vnitřní** - vyvolané procesorem (problém se zpracováním instrukcí, dělení 0, výpadek stránky)
  - Také nazývané **synchronní**, a to proto, že CPU jej vyvolá až po dokončení aktuálně vykonávané instrukce, Intelovský manuál je nazývá **Exception**
- **vnější** - hardwarové, přichází z V/V zařízení, mají přidělena čísla IRQ (Interrupt Request)
  - pro signalizaci přerušení– *kanály přerušení*, reprezentovány čísly IRQ, na jednom IRQ kanálu může být napojeno více zařízení (= sdílený kanál, sdílené IRQ)
  - Také nazývané **asynchronní**, a to proto, že závisí na ticích časovače a může nastat v době kdy CPU právě vykonává nějakou instrukci, Intelovský manuál je nazývá **Interrupt**
- **softwarové** - speciální instrukce INT 0x80 nebo SYSENTER (SYSCALL)

je dulezity registr idtr - obsahuje poiner na tabulku preruseni. prijde preruseni od hardware, dostane se po dratu do i/o apic, ten ho prelozi a ziska cislo preruseni - koukne do tabulky a podle priority v tabulce ho preposle APIC v jadru ktere je uvedeno v te tabulce u toho preruseni. to konkretni jadro dostane preruseni po intrl (interrupt line) a pushne vse co dela na zasobnik, prepne mod do Ring 0 (rezim jadra je jen mod opravneni v CPU, ring 0 znamena ze ti CPU umozni delat cokoli, ring 3 je mod kde bezi aplikace a pamet je omezena jen na tu ktera je pridelena procesu), CPU v tomhle rezimu jadra koukne na IDTR registr kde je adresa vektoru preruseni, pripocte k te adrese 31 (vnitrni preruseni CPU jsou na zacatku vektoru preruseni, pak az jsou ty hardwarovy), koukne kam to ukazuje a dostane pointer na obsluhu preruseni kam taky skoci (porad v rezimu jadra). Ta obsluha muze byt v driveru (interrupt request handler si registruje driver pri zavedeni a je to jen prepsani te adresy ve vektoru preruseni, popr. kdyz ma vic preruseni stejny cislo tak tam je ulozen spojovy seznam a zkousi se ty handlery pekne postupne). obsluha preruseni musi byt co nejkratsi a na konci se musi zavolat instrukce iret, ktera koukne na zasobnik jadra a podle toho co tam uvidi bud vrati rizeni zpet do procesu kterej byl prerusenej, nebo v pripade zanorenyo preruseni vrati obsluhu predchozimu preruseni. Pokud je preruseni !na dlouho!, linux v jeho obsluze jen vyvola nejaky tasklet (lwp proces v jadre) a necha obsluhu na pozdeji (linux checkuje jestli nejake tasklety cekaji a obsluhuje je napr. pri prepinani do/z rezimu jadra, ale takovych mist je v kernelu pry vic.

Další dělení je na maskovatelná a nemaskovatelná

Interrupts:
▪ **Maskable interrupts**: All Interrupt Requests (IRQs) issued by I/O devices give rise to maskable interrupts . A maskable interrupt can be in two states: masked or unmasked; a masked interrupt is ignored by the control unit as long as it remains masked.
▪ **Nonmaskable interrupts**: Only a few critical events (such as hardware failures) give rise to nonmaskable interrupts . Nonmaskable interrupts are always recognized by the CPU.

- Přerušení mají priority - obsluha přerušení může být přerušena přerušením s vyšší prioritou :) - pozn. Priority z hlediska kernelu jsou jen **low** a **high**, tedy přerušitelné a nepřerušitelné. APIC (Advanced Programmable Interrupt Controller ale priority podporuje, takže priority přerušení jako takové Linux nikde neřeší, ale nechává to na hardware (APIC je v každém CPU).

The Linux kernel is reentrant (like all UNIX ones), which simply means that multiple processes can be executed by the CPU. Hedoesn't have to wait till a disk access read is handled by the deadly slow HDD controller, the CPU can process some other stuff until the disk access is finished (which itself will trigger an interrupt if so).
Generally, an interrupt can be interrupted by an other interrupt (preemption), that's called 'Nested Execution'. Depending onthe architecture, there are still some critical functions which have to run without interruption (non-preemptive) by completely disabling interrupts. On x86, these are some time relevant functions (`time.c`, `hpet.c`) and some `xen` stuff.
There are only two priority levels concerning interrupts: 'enable all interrupts' or 'disable all interrupts', so I guess your "high priority interrupt" is the second one. This is the only behavior the Linux kernel knows concerning interrupt priorities and has nothing to do with real-time extensions.
If an interruptible interrupt (your "low priority interrupt") gets interrupted by an other interrupt ("high" or "low"), the kernel saves the old execution code of the interrupted interrupt and starts to process the new interrupt. This "nesting" can happen multiple times and thus can create multiple levels of interrupted interrupts.
Afterwards, the kernel reloads the saved code from the old interrupt and tries to finish the old one.

From <http://unix.stackexchange.com/questions/7124/re-entrency-of-interrupts-in-linux>

- Přerušení je obecně asynchronní vzhledem k přerušenému procesu
- Zpracování přerušení nesmí způsobit čekání, přerušený proces zůstává ve stavu běžící
- čas zpracování přerušení je započítán přerušenému procesu, při zpracování se tedy přistupuje do jeho záznamu proc

Obsluha:

1) uloží IRQ (Interrupt ReQuest) a obsah registrů

2) pošle potvrzení PIC (Programmable Interrupt Controller), který zajišťuje provoz přerušení

3) *modul pro obsluhu přerušení* (Interrupt Handler) vykoná obsluhu přerušení

4) ukončí skokem na ret_from_intr()

Výjimky

- synchronní s procesem (vznikají v důsledku událostí způsobených vykonáváním procesu)

Zpracování:

- Stejné jako u přerušení, 80x86 procesory mají přibližně 20 definovaných výjimek, každá je zpracována přiřazeným exception handlerem - ten většinou nedělá nic jiného než že pošle signál procesu, který výjimku způsobil, kernel musí poskytnout exception handler pro každou definovanou výjimku, příklady výjimek i s jejich číslem: 0 - Divide Error = dělení nulou, 1 - Debug, 4 - Overflow = přetečení…

1) uloží obsah registrů

2) zpracuje výjimku (funkce v jazyku C)

- pošle signál procesu
- zpracuje žádost o stránku

3) ukončí se voláním funkce ret_from_exception()

Table 4-1. Signals sent by the exception handlers

#	Exception	Exception handler	Signal
0	Divide error	divide_error( )	SIGFPE
1	Debug	debug( )	SIGTRAP
2	NMI	nmi( )	None
3	Breakpoint	int3( )	SIGTRAP
4	Overflow	overflow( )	SIGSEGV
5	Bounds check	bounds( )	SIGSEGV
6	Invalid opcode	invalid_op( )	SIGILL
7	Device not available	device_not_available( )	None
8	Double fault	doublefault_fn( )	None
9	Coprocessor segment overrun	coprocessor_segment_overrun( )	SIGFPE
10	Invalid TSS	invalid_TSS( )	SIGSEGV
11	Segment not present	segment_not_present( )	SIGBUS
12	Stack segment fault	stack_segment( )	SIGBUS
13	General protection	general_protection( )	SIGSEGV
14	Page Fault	page_fault( )	SIGSEGV
15	Intel-reserved	None	None
16	Floating-point error	coprocessor_error( )	SIGFPE
17	Alignment check	alignment_check( )	SIGBUS
18	Machine check	machine_check( )	None
19	SIMD floating point	simd_coprocessor_error( )	SIGFPE

Vektor přerušení

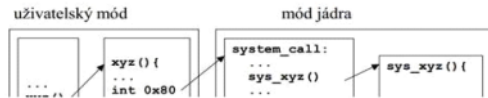
- 0-31 vnitřní přerušení (výjimky), 32 - 255 IRQ přerušení

Vektor přerušení u x86 se nazývá IDT (Interrupt Descriptor Table) - tabulka, která přiřazuje obslužnou rutinu ke každému přerušení

Systémové volání

- mechanismus pro volání funkcí operačního systému aplikacemi
- v standardní knihovně jazyka C je pro každé systémové volání obálková procedura, řízení se předá softwarovým přerušením proceduře jádra syscall()
- system_call - jedna pro všechny služby, požadovaná služba se odlišuje parametrem procedury, který se nazývá číslo

systémového volání



Obsluha:

system_call:

1) uloží obsah registrů (HW kontext)

2) zavolá odpovídající funkci (v jazyku C)

3) ukončí se voláním ret_from_sys_call()

Int 0x80 je legacy instrukce, funguje jen u 32 bit systému a dnes se nepoužívá

- `syscall` is default way of entering kernel mode on `x86-64`. This instruction is not available in 32 bit modes of operation *on Intel processors*.
- `sysenter` is an instruction most frequently used to invoke system calls in 32 bit modes of operation. It is similar to `syscall`, a bit more difficult to use though, but that is kernel's concern.
- `int 0x80` is a legacy way to invoke a system call and should be avoided.
Preferable way to invoke a system call is to use VDSO, a part of memory mapped in each process address space that allow to use system calls more efficiently (for example, by not entering kernel mode in some cases at all). VDSO also takes care of more difficult, in comparison to the legacy `int 0x80` way, handling of `syscall` or `sysenter` instructions.

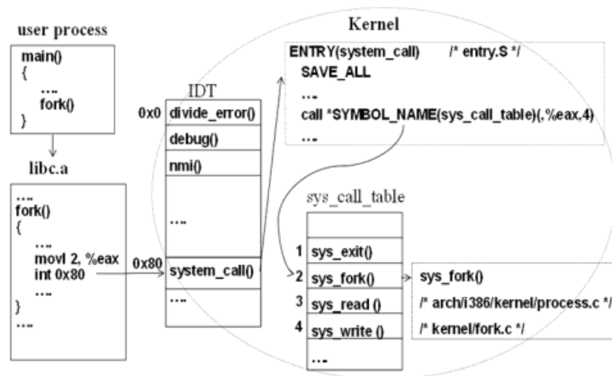
From <http://stackoverflow.com/questions/12806584/what-is-better-int-0x80-or-syscall>

VDSOs (**V**irtual **D**ynamically linked **S**hared **O**bjects) are a way to export kernel space routines to user space applications, using standard mechanisms for linking and loading (i.e. standard ELF format).
It helps to reduce the calling overhead on simple kernel routines, and also can work as a way to select the best system call method on some architectures.
An advantage over other methods is that such exported routines can provide proper DWARF debugging information.
Implementation generally implies hooks in the dynamic linker to find the **VDSO**s.

From <http://en.wikipedia.org/wiki/VDSO>

Cool diagrámek, jak se zpracovává systémové volání:

(z diskuse http://www.unix.com/unix-dummies-questions-answers/178442-x86-interrupts-system-calls.html)



Všeobecné registry procesoru x86 (od 80386 dále)

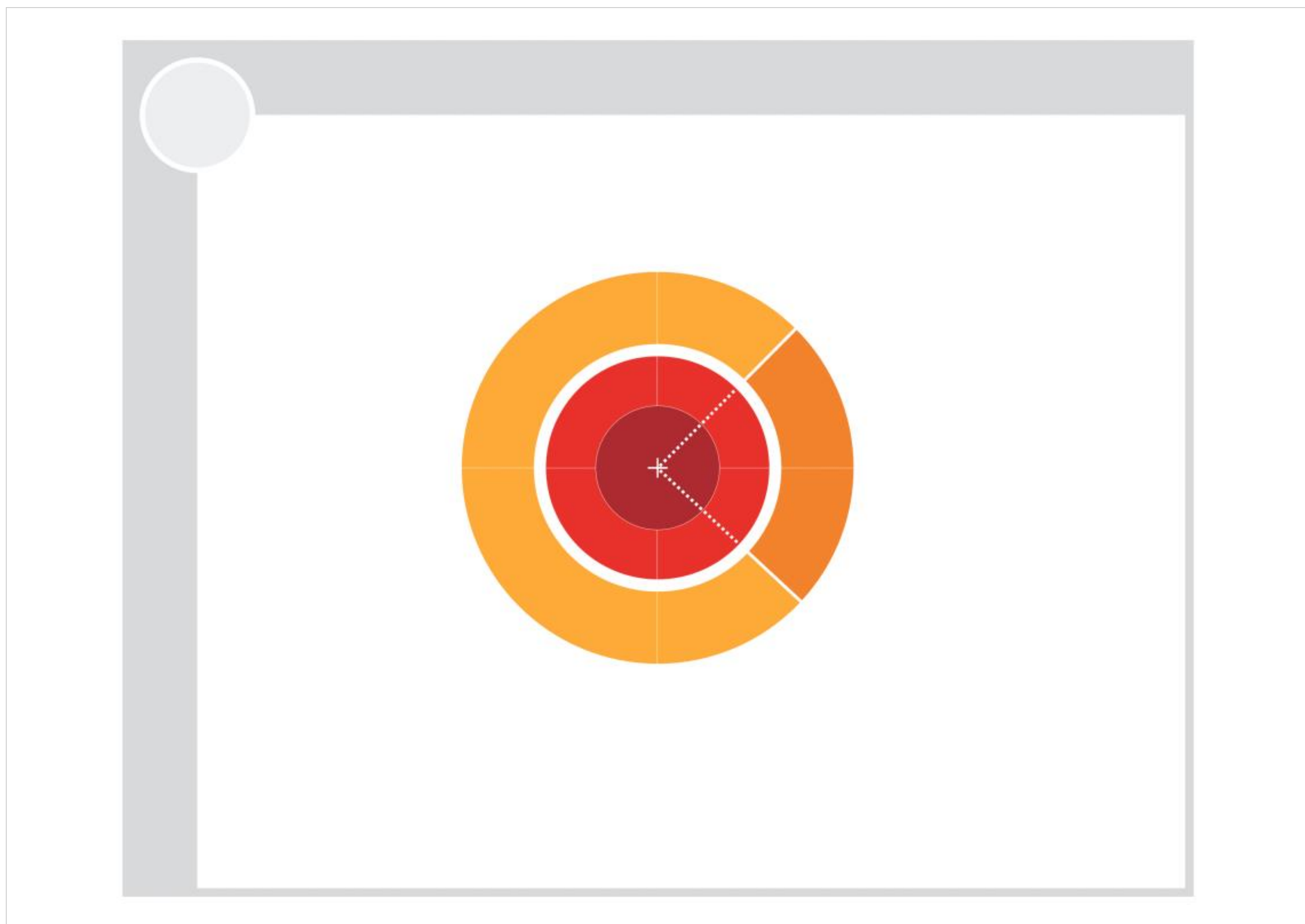
31 – 16	15 – 8	7 – 0	
EAX	AH (AX)	AL	všeob. **střadače** (Accumulators)
EBX	BH (BX)	BL	
ECX	CH (CX)	CL	
EDX	DH (DX)	DL	
ESI	SI		Source Index
EDI	DI		Destination Index
EBP	BP		Base Pointer
ESP	SP		Stack Pointer

- tyto registry jsou obecně použitelné v programu k dočasnému ukládání dat (až na ESP - s tím opatrně)
- naplňují se instrukcí `MOV` *reg*, *hodnota*, např. `MOV BL, 5`

From <https://d.docs.live.net/e3534876709763a3/Dokumenty/ZCU/Statnice/Statnice.docx>

Výtisk

30. května 2013 10:54

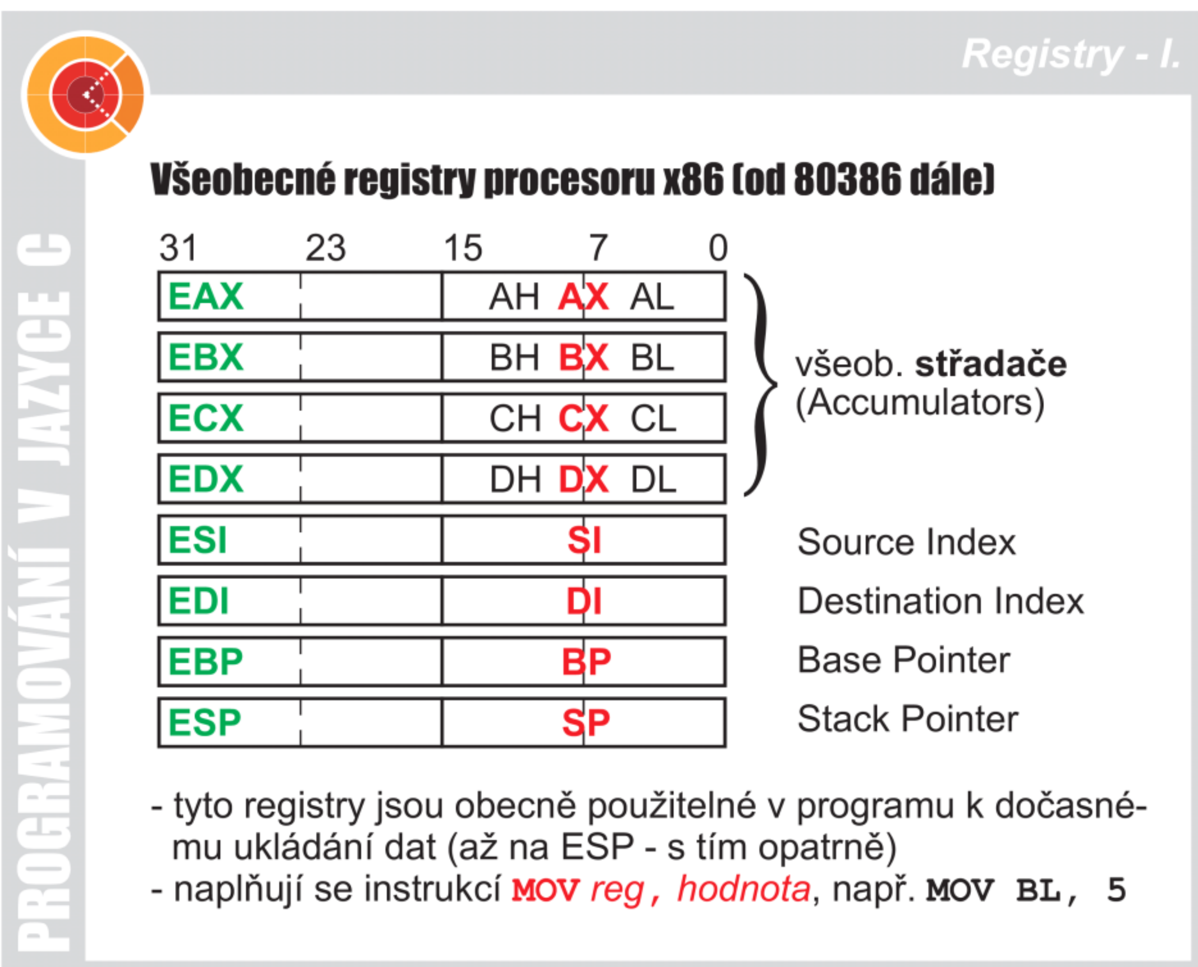


Úvod

PROGRAMOVÁNÍ V JAZYCE C

Programátorský model x86

- programátorským modelem se rozumí soubor vlastností a fyzických součástí procesoru, které ovlivňují jeho programování v nízkoúrovňových jazycích
- zejména popisuje **uspořádání paměti**, využitelné **registry**, nativní **typy dat** daného procesoru, **instrukční soubor**, **přerušovací systém**, atp.


Registry - I.
PROGRAMOVÁNÍ V JAZYCE C
Všeobecné registry procesoru x86 (od 80386 dále)
31 23 15 7 0
EAX AH AX AL
EBX BH BX BL
ECX CH CX CL
EDX DH DX DL
ESI SI
EDI DI
EBP BP
ESP SP
všeob. střadače
(Accumulators)
Source Index
Destination Index
Base Pointer
Stack Pointer
- tyto registry jsou obecně použitelné v programu k dočasné-
mu ukládání dat (až na ESP - s tím opatrně)
- naplňují se instrukcí MOV reg, hodnota, např. MOV BL, 5

Registry - II.

PROGRAMOVÁNÍ V JAZYCE C

Segmentové registry procesoru x86 (od 80386 dále)

15 7 0	
CS	Code Segment
DS	Data Segment
SS	Stack Segment
ES	Extra Segment
FS	Extra Segment
GS	Extra Segment

- viditelná část segmentových registrů je 16-bitová, plnit je lze instrukcí **MOV** nebo spec. instrukcemi **LDS**, **LES**, **LFS**, **LGS** a **LSS**, např. **LDS EBX,** *dword-ptr*, která umístí do registrového páru DS:EBX 32-bitovou adresu *dword-ptr*
- **změna obsahu CS má fatální následky** (CS obsahuje tzv. selektor kódového segmentu)

PROGRAMOVÁNÍ V JAZYCE C

Registry - III.

Registry se zvláštním významem

CS:EIP (Extended Instruction Pointer)

selektor segmentu

není to přímo adresa v paměti, jedná se o **index do tabulky GDT/LDT** (Global/Local Descriptor Table), pozice GDT je v registru GDTR, LDT v registru LDTR

EIP ukazuje na pozici v kódovém segmentu, kde leží právě prováděná instrukce, tj. **pár CS:EIP udává pozici právě vykonávané instrukce**

CS

Adresa	Instrukce
8A00	ADD EAX, 5
8A01	MUL EAX, EBX
8A02	JC 8A0A
8A03	SHR EAX, 1
8A04	CMP EAX, 0
8A05	JE 8A0D
8A06	NEG EAX
EIP ▶ 8A07	JMP 8A0F
8A08	...
8A09	
8A0A	
8A0B	
8A0C	
8A0D	
8A0E	
8A0F	
8A10	

Toto je pouze ilustrativní obrázek - instrukce ve skutečnosti nezabírají stejné množství paměti...

Stránka 6

30. května 2013 10:54

PROGRAMOVÁNÍ V JAZYCE C

Registry - IV.

Registry se zvláštním významem

CS:EIP - pozice vykonávané instrukce (mění ji sám procesor)

SS:ESP - pozice vrcholu zásobníku (mění ji instrukce **PUSH** a **POP**)

} pro programátora (zvláště nezkušeného) **read-only**!

DS:ESI - zdrojová adresa pro instrukce blokového přesunu dat (**MOVS/MOVSB/MOVSW**)

ES:EDI - cílová adresa pro instrukce blokového přesunu dat

```
LDS SI, <adresa zdrojového řetězce>
LES DI, <adresa cílového řetězce>
MOV CX, <počet prvků řetězce>
REP MOVSB
```

kopírování řetězce (pole bytů)

30. května 2013 10:54

Registr EFLAGS

PROGRAMOVÁNÍ V JAZYCE C

Příznakový registr EFLAGS (32-bitový)

Obsahuje dva druhy **vlajek** (**1-bitových příznaků**):

(i) **nastavované procesorem** po provedení instrukce indikují vlastnosti výsledku (**CF**, **PF**, **AF**, **ZF**, **SF**, **OF**),

(ii) **nastavované programátorem** řídí činnost procesoru (**TF**, **IF**, **DF**, **VM**, **RF**, **NT**, **IOPL**).

Virtual-Mode → 17, Resume → 16

0	0	0	0	0	0	0	0	0	0	0	0	0	0	VM	RF
														17	16

0	NT	IOPL		OF	DF	IF	TF	SF	ZF	0	AF	0	PF	1	CF
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

Direction (DF), Interrupt-Enable (IF), Trap (TF)

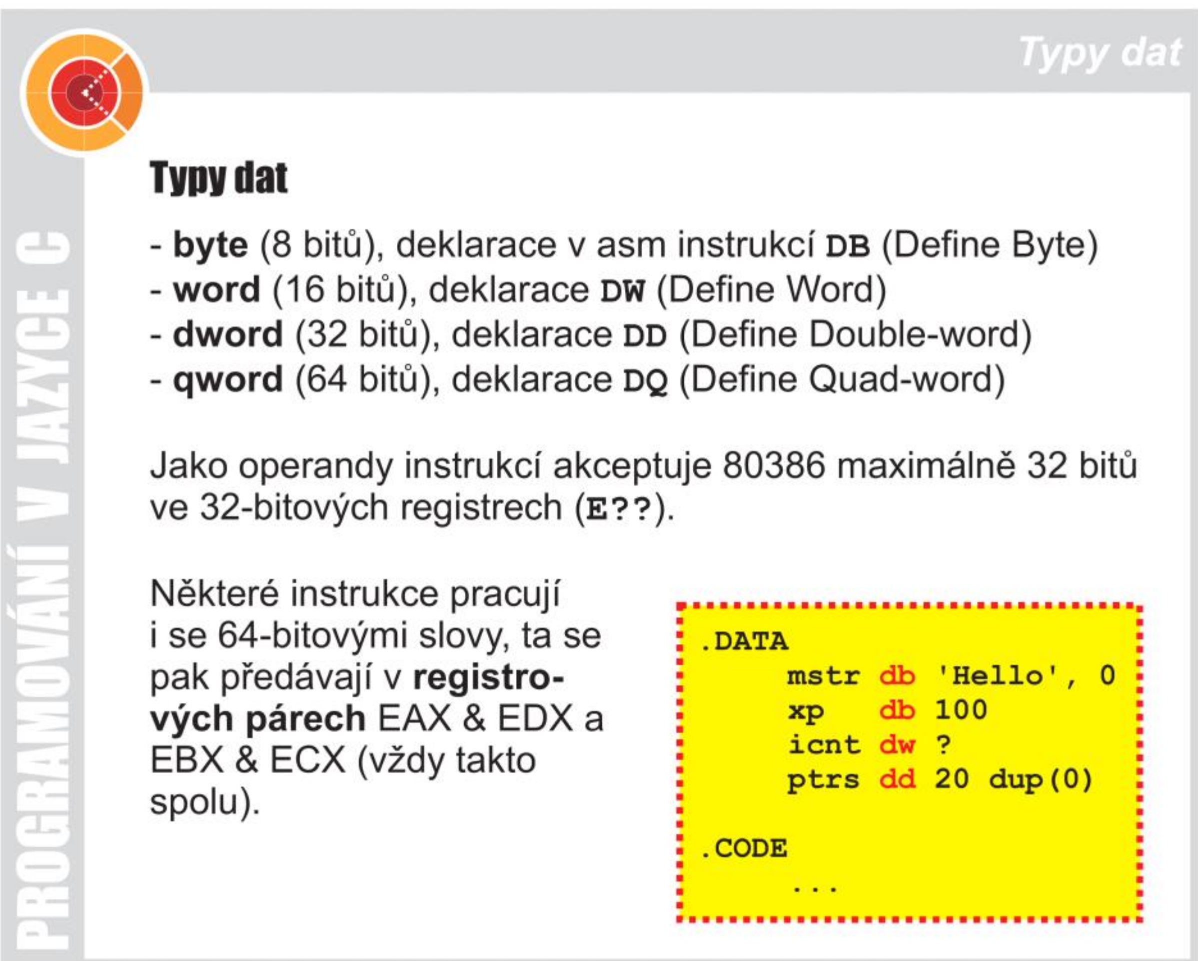
Nested-Task (NT), I/O-Privilege-Level (IOPL)

Overflow (OF), Sign (SF), Zero (ZF)

Aux Carry (AF), Parity (PF), Carry (CF)

Stránka 8

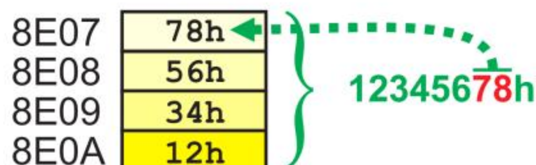
30. května 2013 10:54



PROGRAMOVÁNÍ V JAZYCE C

Endian procesoru (čili uspořádání bytů ve slovech)

Procesory Intel (a jejich klony) používají **Little Endian**, tj. nižší řády jsou na nižších adresách:



Procesory Motorola, AIM PowerPC (po G5), Sun SPARC (starší verze do V9), IBM System/370 používají **Big Endian**, tzn. nižší řády na vyšších adresách (vlastně tak, jak se číslo píše na papír).

Některé procesory umí endian podle potřeby přepínat, např. ARM, SPARC V9, MIPS, PA-RISC, IA64, DEC Alpha, některé PowerPC => tzv. **Bi-Endian**.

Přerušení (8086)

- **tabulka přerušovacích vektorů** je umístěná na fyzickém počátku paměti od adresy 0000:0000
- adresa ukazuje na začátek obslužné rutiny přerušení (musí končit instrukcí `IRET`)
- přerušení může být vyvolané buď HW (z vnějšku přivedením log. úrovně na daný pin procesoru) nebo SW instrukcí `INT n`
- HW přerušení lze **maskovat** vynulováním příznaku `IF` instrukcí `CLI` (kromě vnitřních a NMI).

Adresa			
03FC	segment	offset	INT 0FFh
:			
000C	segment	offset	INT 3
0008	segment	offset	INT 2
0004	segment	offset	INT 1
0000	segment	offset	INT 0

INT 3 → 8E07

```
...
8E07  XOR AX, AX
8E08  IN AX, 00h
8E09  NOT AX
8E0A  IRET
...
```

Činnost CPU při přerušení (8086)

Nastalo přerušení **n** nebo CPU dekódoval instrukci **INT n**:

(i) do zásobníku se uloží registr příznaků (**FLAGS**),
(ii) vynulují se příznaky **IF** a **TF**,
(iii) do zásobníku se uloží registr **CS**,
(iv) **CS** se naplní obsahem adresy **n** * 4 + 2,
(v) do zásobníku se uloží **IP** ukazující na další **neprovedenou instrukci**,
(vi) **IP** se naplní obsahem adresy **n** * 4.

INT n	Význam
0	Dělení nulou (Divide By Zero)
1	Krokovací režim (Single-Step)
2	Nemaskovatelná přerušení (NMI)
3	Ladicí bod (Breakpoint Trap)
4	Přeplnění (Overflow Trap)

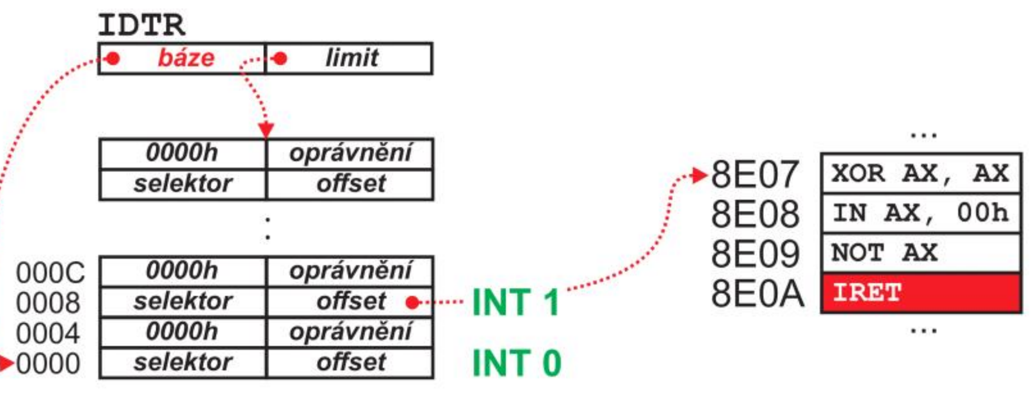
} záleží na operačním systému, jak tato přerušení obslouží

Přerušení - III.

PROGRAMOVÁNÍ V JAZYCE C

Přerušení (80386 a dále) - zjednodušeně

- **tabulka popisovačů segmentů obsluhy přerušení** (IDT = Interrupt Descriptor Table) může být umístěna kdekoliv v paměti, adresa IDT je umístěna v registru `IDTR` (čte/plní se instrukcí `SIDT`/`LIDT`)
- položka IDT se nazývá **popisovač brány přerušení**, brány jsou pro (i) maskovatelná přerušení (Interrupt Gate) a (ii) nemaskovatelná přerušení (Trap Gate)



Podprogramy - I.

PROGRAMOVÁNÍ V JAZYCE C

Volání podprogramů

- záleží na **paměťovém modelu**, jaká návratová adresa se ukládá do zásobníku
- předávání parametrů je na programátorovi či překladači

STACK SEG
SS:ESP
EAX
CODE SEG

Adresa	Zásobník
...0A18	0000
...0A16	0020
...0A14	001C
...0A12	0E7F
...0A10	2007

Adresa	Instrukce
0E7F2001	PUSH EAX
0E7F2002	CALL
0E7F2007	CMP
0E7F2009	JE 0E7F201C
F00A80E0	MOV EAX, SS:[ESP+6]
F00A80E3	ADD EAX, 1
F00A80E4	RETF

zásobník je adresovaný po wordech (16-bit)

Podprogramy - II.

PROGRAMOVÁNÍ V JAZYCE C

Volání podprogramů (assembler)

- pokud se externí modul v assembleru linkuje k programu přeloženému překladačem C, musí se shodovat **paměťový model** a **volací konvence** (způsob předávání p-metrů)
- různé překladače používají různé PM a VK

```
_TEXT SEGMENT WORD PUBLIC 'CODE'
        public _power2
_power2 proc near
        push ebp
        mov ebp, esp
        mov eax, [ebp+4]  ; první argument
        mov ecx, [ebp+6]  ; druhý argument
        shl eax, cl       ; EAX = EAX * ( 2 ^ CL )
        pop ebp
        ret
_power2 endp
_TEXT   ends
        END
```

paměťový model **FLAT**
tj. **CS** = **DS** = **ES** = **SS**

assembler **MASM-like**, překladač **Microsoft Visual C/C++ 2005**

Stránka 15

30. května 2013 10:54

PROGRAMOVÁNÍ V JAZYCE C

Paměťové modely

Paměťové modely

- paměťový model určuje, jaká část programu je umístěna v jakém segmentu (kód, data, zásobník), čím jsou tedy naplněné segmentové registry a jak "velké" jsou pointery
- situace je poněkud nepřehledná, na **16-bitové platformě Intel x86** existuje 6 paměťových modelů:

TINY, **SMALL**, **MEDIUM**, **COMPACT**, **LARGE**, **HUGE**

- moderní 32- a 64-bitové platformy používají zejména model **FLAT**, což je obdoba **TINY**, tj. všechny segmentové registry jsou nastavené na stejnou hodnotu
- výše uvedený paměťový model se takto jeví z hlediska programátora, nikoliv operačního systému - ten techniku segmentace paměti využívá (oprávnění, stránkování, atd.)

- **problematika je značně rozsáhlá a komplikovaná, mimo rámec předmětu PC, zájemci** `http://www.intel.com`

Stránka 16

30. května 2013 10:54

Porty, V/V operace

PROGRAMOVÁNÍ V JAZYCE C

Komunikace procesoru s okolními zařízeními

- děje se pomocí I/O portů, sběrnice může přenášet data buď mezi CPU a pamětí nebo ostatními zařízeními na MB
- signál M/$\overline{IO}$ určuje, za adresa nastavená na adresních vodičích A_0 - A_{15} je adresou paměti nebo I/O portu

```
in  eax, 61h
in  eax, dx

out 20h, eax
out dx, eax
```

```
@L1:
    mov al, 0Ah ; 0Ah - offset 'valid'
    out 70h, al ; 70h - CMOS index port
    in al, 71h  ; 71h - CMOS data port
    test al, 10000000b
    jnz @L1     ; bit7 = 1, znovu

    xor al, al
    out 70h, al

    in al, 71h  ; čteme bajt 0 - sekundy
```