Systémové programovanie

RNDr. Jaroslav Janáček, PhD.

Aplikačné vs. systémové programy

- aplikačné programy
  - riešia aplikačné úlohy
  - nezávislé od HW a detailov OS
- systémové programy
  - podporujú používanie počítača
  - závislé od detailov OS
  - môžu byť závislé od HW
  - napr. OS, kompilátor, assembler, linker, loader, debugger, konfiguračné programy, ...

Zjednodušená štruktúra počítača

- procesor
  - aritmeticko-logická jednotka
  - riadiaca jednotka
  - registre
- hlavná pamäť
  - medzi ňou a procesorom sa často nachádza cache pamäť (alebo hierarchia cache pamätí)
- vstupno-výstupné rozhranie
  - pripojenie periférnych zariadení

Reprezentácia údajov

- dátový typ
  - množina hodnôt
  - množina operácií
- abstraktný dátový typ
  - množina abstraktných entít a operácií bez vzťahu k počítaču
- virtuálny dátový typ – entity prog. jazyka
- fyzický dátový typ
  - fyzicky uložené entity v HW počítača

Reprezentácia údajov

- v HW (register, pamäť) uložené ako reťazec bitov
- kódovanie
  - relácia medzi prvkami dátového typu a jeho reprezentáciou v bitovom reťazci

Celé čísla

- bez znamienka
  - BCD (binary coded decimal)
    - každá číslica desiatkovej sústavy reprezentovaná 4 bitmi
    - príklad: 1000 0110 = 86
    - používajú sa napríklad v hodinových obvodoch
  - v dvojkovej sústave
    - príklad: 1000 0110 = 134 = 0x86

Celé čísla

- znamienko + abs. hodnota
  - príklad: 0100 0110 = 70
1100 0110 = -70
  - dve reprezentácie 0, nepoužiteľnosť sčítačky na odčítanie
- 1's complement
  - kladné čísla majú najvyšší bit 0, -x = NOT(x)
  - príklad: 0100 0110 = 70, 1011 1001 = -70
  - sčítanie sa doplní o pripočítanie prenosu
  - 1111 1111 = 0000 0000

Celé čísla

- 2's complement
  - kladné čísla majú najvyšší bit 0
  - -x = NOT(x) + 1
  - príklad: 0100 0110 = 70
    1011 1010 = -70
  - sčítačka funguje aj so zápornými číslami (a teda aj na odčítanie)
  - najčastejšie používaný formát celých čísel

Viacbytové čísla

- základnou pamäťovou jednotkou je 1 byte = 8 bitov
- čísla zaberajúce viac bytov môžu byť uložené dvoma spôsobmi
  - menej významné byty na nižších adresách
    - little endian, Least Significant Byte First
    - MEM[x]=01_{16}, MEM[x+1]=00_{16} -> 0001_{16}
  - viac významné byty na nižších adresách
    - big endian, Most Significant Byte First
    - MEM[x]=01_{16}, MEM[x+1]=00_{16} -> 0100_{16}

Čísla s pevnou rádovou čiarkou

- používajú sa celé čísla vynásobené príslušnou mocninou základu sústavy
  - napr. čísla v desiatkovej sústave s dvoma desatinnými miestami:
    - hodnota sa vynásobí 100-mi
    - výsledok sa delí 100-mi a zaokrúhli sa podľa zvyšku
  - aditívne operácie sú ľahké a presné
  - násobenie a delenie je komplikovanejšie

Čísla s pohyblivou rádovou čiarkou

- umožňujú zobraziť malé aj veľké čísla
  - s daným počtom platných číslic – t.j. s danou relatívnou presnosťou
    - pri veľkých číslach je absolútna presnosť menšia ako pri malých číslach
- $(-1)^s * m * 2^{e-e_0}$
  - s – znamienko (0=+, 1=-)
  - m – mantisa (1.xxxxxxx v dvojkovej sústave)
  - e – exponent (zväčšený o e_0)

Čísla s pohyblivou rádovou čiarkou

- normalizované čísla
  - $1 <= m < 2$, $0 < e < e_{max}$
  - najvyšší bit mantisy (celočíselný) sa často neuvádza (skrytý bit), považuje sa implicitne za 1
- nenormalizované čísla
  - $0 < m < 1$, $e = 0$, $hodnota=(-1)^s * m * 2^{1-e_0}$
  - používajú sa na vyjadrenie veľmi malých čísel
  - neuvedený celočíselný bit mantisy je 0
  - dochádza k zníženiu relatívnej presnosti

Čísla s pohyblivou rádovou čiarkou

- kladná a záporná nula
  - m=0, e=0
- +∞ a -∞
  - m=1, $e=e_{max}$
- NaN (Not A Number)
  - m>1, $e=e_{max}$

Čísla s pohyblivou rádovou čiarkou

- IEEE 754
  - single precision
    - 32 bitov: se...em...m, 1, 8, 23
    - $e_0 = 127$, $e_{max} = 255$, skrytý bit v mantise
  - double precision
    - 64 bitov: se...em...m, 1, 11, 52
    - $e_0 = 1023$, $e_{max} = 2047$, skrytý bit v mantise
  - double extended precision
    - 80 bitov: se...eim...m, 1, 15, 1+63
    - $e_0 = 16383$, $e_{max} = 32767$, i = celočíselný bit mantisy

Čísla s pohyblivou rádovou čiarkou

- Príklady:
  - $5.75 = 4 + 1 + 0.5 + 0.25 = 101.11_2 = 1.0111_2 * 2^2$
    - s=0, e=2+127=129, $m=1.0111_2 = 1.4375$
    - 01000000101110000000000000000000
  - 0
    - s=0, e=0, m=0
    - 00000000000000000000000000000000
  - -∞
    - s=1, e=255, m=1
    - 11111111000000000000000000000000

Čísla s pohyblivou rádovou čiarkou

- 2^{-126}
  - s=0, e=-126+127=1, m=1
  - 00000000100000000000000000000000
- 2^{-127}
  - s=0, e=0, m=0.1_2
  - 00000000010000000000000000000000
- -2^{-129}
  - s=1, e=0, m=0.001_2
  - 10000000001000000000000000000000

Ďalšie typy údajov

- znaky
  - reprezentované ako čísla v príslušnom kódovaní
    - klasické znakové sady – 1 znak v 1 byte
    - UNICODE – 1 znak v 1 až 4 byte-och
- znakové reťazce
  - dĺžka + text
    - napr. 8 bitové číslo a ďalej príslušný počet znakov
  - znaky s ukončovacím znakom
    - napr. v jazyku C ukončené znakom s hodnotou 0
- bitové reťazce

UNICODE

- Priestor pre 1114112 ($17*2^{16}$) znakov
  - U+0000 – U+10FFFF
  - 17 rovín (planes) s veľkosťou 256x256 znakov
- Kódovania
  - UCS-4, UTF-32
    - 4B na znak (UCS-4 podporuje rozsah až 31b)
  - UCS-2
    - 2B na znak
    - podporuje len 16b (U+0000 – U+FFFF)

UNICODE

- UTF-16
  - 2 alebo 4B na znak
  - znaky z U+0000 – U+FFFF – 2B
  - znaky z U+10000 – U+10FFFF (t.j. 2^{20} znakov) pomocou dvoch 2B symbolov
    - od pôvodnej hodnoty sa odčíta 0x10000 – dostaneme hodnotu z rozsahu 0-0xFFFFF, tú rozdelíme na dve 10b časti
    - prvý symbol bude U+D800 + horných 10b hodnoty
    - druhý symbol bude U+DC00 + dolných 10b hodnoty
    - symboly U+D800 – U+DFFF nereprezentujú žiadne znaky, slúžia práve pre účely UTF-16
  - UTF-16LE vs. UTF-16BE – poradie byte-ov
  - U+FEFF = BOM (Byte Order Mark)

UNICODE

- UTF-8
  - 1 – 4 B na znak, **musí** sa zapísať **minimálnym** počtom
  - kód žiadneho znaku nie je prefixom žiadneho iného
    - umožňuje jednoduché vyhľadávanie podreťazca (rovnako ako pri kódovaniach s pevnou veľkosťou znaku)
  - niekoľko najvyšších bitov prvého byte-u kódu znaku určuje, aký dlhý je kód
    - 0……. - 1B, hodnota znaku je určená v 7 bitoch
    - 110….. - 2B, hodnota znaku je určená v 5+6 = 11 bitoch
    - 1110…. - 3B, hodnota znaku je určená v 4+6+6 = 16 bitoch
    - 11110… - 4B, hodnota znaku je určená v 3+6+6+6 = 21 bitoch
    - 10…… - používa sa ako 2. - 4. byte kódu, obsahuje 6 bitov

UNICODE

- UTF-8
  - pri dekódovaní treba odmietnuť kódy, ktoré nepoužívajú min. počet byte-ov
    - nerešpektovanie vedie často k zneužiteľným zraniteľnostiam
  - napr. kód 0xC0, 0x8A **nie je** povolenou reprezentáciou U+000A, správna je 0x0A.
  - U+0000 – U+007F : 1B: 0x00 – 0x7F
  - U+0080 – U+07FF : 2B: 0xC2,0x80 – 0xDF,0xBF
  - U+0800 – U+FFFF : 3B: 0xE0,0xA0,0x80 – 0xEF,0xBF,0xBF
  - U+10000 – U+10FFFF : 4B: 0xF0,0x90,0x80,0x80 – 0xF4,0x8F,0xBF,0xBF

Jazyk assemblera

- príkazy
  - direktívy
    - riadiace príkazy pre assembler, napr. vyhradenie miesta, deklarácia typu symbolu, ...
  - strojové inštrukcie
    - zodpovedajú inštrukciám procesora
  - makropríkazy
    - používateľom definované makrá

Typy inštrukcií

- prenos dát (medzi reg. a pamäťou)
- aritmetické inštrukcie
  - sčítanie, odčítanie, násobenie, delenie, porovnanie, negácia, ...
- logické inštrukcie
  - AND, OR, XOR, NOT
- riadiace inštrukcie
- vstupno/výstupné inštrukcie
- iné

Registre x86_64

- všeobecné (8, 16, 32, 64 bitov)
- príznakový (EFLAGS)
- segmentové (cs, ds, ss, es, fs, gs)
- riadiace (cr0-cr4), ladiace (dr)
- 80bit floating point (st0-st7)
- MMX, XMM pre maticové operácie
- ďalšie špeciálne registre

Všeobecné registre x86_64

64 bit	32 bit	16 bit	8 bit (low)
rax	eax	ax	al, ah
rbx	ebx	bx	bl, bh
rcx	ecx	cx	cl, ch
rdx	edx	dx	dl, dh
rsi	esi	si	sil
rdi	edi	di	dil
rbp	ebp	bp	bpl
rsp	esp	sp	spl
r8 - r15	r8d - r15d	r8w - r15w	r8l - r15l
rip	eip	ip	

Všeobecné registre x86_64

<table>
<tr><td colspan="8">rax</td></tr>
<tr><td colspan="4"></td><td colspan="4">eax</td></tr>
<tr><td colspan="2"></td><td colspan="2"></td><td colspan="2"></td><td colspan="2">ax</td></tr>
<tr><td colspan="2"></td><td colspan="2"></td><td colspan="2">ah</td><td colspan="2">al</td></tr>
</table>

Syntax inštrukcií

- názov_inštrukcie
- názov_inštrukcie operand
- názov_inštrukcie operand1, operand2

- Typy operandov
  - register
  - hodnota (číslo, znak)
  - odkaz do pamäte
  - meno symbolu, príp. aritm. výraz so symbolmi

Intel vs. AT&T syntax pre x86

- AT&T
  - mená registrov uvádzané s prefixom %
  - hodnoty uvádzané s prefixom $
  - poradie operandov: zdroj, cieľ
  - veľkosť operandu určená podľa posledného písmena mena inštrukcie
    - b – byte, w – word, l – long, q – 64bit
    - v prípade, že jedným z operandov je register, určí sa aj automaticky podľa veľkosti registra
      - napr. movl $5, (%eax)

Intel vs. AT&T syntax pre x86

- Intel
  - mená registrov bez prefixu
  - hodnoty bez prefixu
  - poradie operandov: cieľ, zdroj
  - veľkosť pamäťového operandu určená pri operande
    - byte|word|dword|qword ptr [...]
      - napr. mov dword ptr [eax], 5

Adresné módy

- Registrový
  - mov %eax, %ebx mov ebx, eax
  - ebx := eax
- Nepriamy registrový
  - mov (%rax), %ebx mov ebx, dword ptr [rax]
  - ebx := MEM[rax]
- Nepriamy s doplnkom
  - mov 10(%rax), %ebx mov ebx, dword ptr [rax+10]
  - ebx := MEM[rax+10]

Adresné módy

- Nepriamy s doplnkom a indexom
  - mov 10(%rax, %rsi, 4), %ecx
  - mov ecx, dword ptr [rax+rsi*4+10]
  - ecx := MEM[rax+rsi*4+10]
- Priama adresa
  - mov 100, %eax mov eax, dword ptr [100]
  - eax := MEM[100]
- Priama hodnota
  - mov $5, %eax mov eax, 5
  - eax := 5

Adresné módy

- Relatívna adresa
  - ukladá sa adresa cieľa relatívne k EIP
  - na x86 sa používa pri skokoch a volaniach (jmp, call), v 64-bit móde aj na prístup k dátam
    - mov 100(%rip), %ebx
- Autoinkrementačný/autodekrementačný
  - adresa v registri sa automaticky zväčší/zmenší po (pred) vykonaním inštrukcie
  - na x86 sa používa len pri „reťazcových“ inštrukciách (adresuje sa registrami esi a edi)

Kombinácie adresných módov

- inštrukcia môže zvyčajne obsahovať len jeden odkaz na pamäť
- niektoré procesory umožňujú kombinovať priamu hodnotu len s registrovým, príp. nepriamym registrovým módom
  - t.j. nepovoľujú uviesť súčasne priamu hodnotu a adresu (alebo doplnok)

Štruktúra programu

- program sa bežne skladá z viacerých modulov
  - jednotlivé moduly môžu byť napísané v rôznych prog. jazykoch alebo v assembleri
- modul obsahuje *sekcie*
  - .text – kód
  - .data – inicializované premenné
  - .bss – neinicializované premenné
- sekcie spája *linker*

Štruktúra modulu v assembleri (GNU as)

```
.text
.global f
f: mov a, %eax
   mov %eax, b
   ret

.data
a: .long 10

.bss
.global b
b: .long 0
```

Niektoré direktívy

- .global
  - deklaruje symbol ako globálny, t.j. dostupný aj z iných modulov (inak je symbol dostupný len v rámci modulu)
- .text, .data, .bss
  - prepína sekcie
- .byte, .word, .long, .quad
  - vyhradzuje miesto príslušnej veľkosti (1, 2, 4, 8B) a definuje hodnotu, ktorá sa doň uloží
- .ascii, .asciz
  - vyhradzuje miesto pre textový reťazec (bez/s ukončením 0)

Vybrané inštrukcie x86

- add a, r r:=r+a
- adc a, r r:=r+a+(hodnota bitu prenosu)
- sub a, r r:=r-a
- sbb a, r r:=r-a-(hodnota bitu prenosu)
- inc r r:=r+1 (nemení príznaky)
- dec r r:=r-1 (nemení príznaky)
- neg r r:=-r
- cmp a, b nastaví príznaky podľa b-a

Vybrané inštrukcie x86

- mul a edx:eax := eax * a
- mul a rdx:rax := rax * a
- imul a ako mul, ale znamienkové
- div a eax := edx:eax / a, edx := zvyšok
- div a rax := rdx:rax / a, rdx := zvyšok
- idiv a ako div, ale znamienkové

Vybrané inštrukcie x86

- and a, r r:=r AND a
- or a, r r:=r OR a
- xor a, r r:=r XOR a
  - veľmi užitočný trik: xor %rax, %rax
- not r r:=NOT r
- shl b, a a:=a<<b (b je $… alebo %cl)
- shr b, a a:=a>>b
- sar b, a ako shl, ale znamienkovo

Vybrané inštrukcie x86

- mov s, d d:=s
- xchg a, b a:=:b (výmena obsahu)
- push a ESP-=veľkosť, MEM[ESP]:=a
- pop a a:=MEM[ESP], ESP+=veľkosť
- pushf push flags
- popf pop flags

Vybrané inštrukcie x86

- jmp addr skok na adresu
  - rel. adr. alebo adr. v pamäti/registri
- call addr volanie funkcie
- ret návrat z funkcie
- j*cond* rel8 podmienený skok, ak *cond*
  - z(e) =0, nz(ne) !=0
  - b(c) <0, ae(nc) >=0, a >0, be <=0 – neznam. por.
  - l <0, ge >=0, g >0, le <=0 – znam. porovnanie

Vybrané inštrukcie x86

- cbtw (cbw) ax := znam. rozš. al
- cwtl (cwde) eax := znam rozš. ax
- cltq (cdqe) rax := znam rozš. eax
- cwtd (cwd) dx:ax := znam rozš. ax
- cltd (cdq) edx:eax := znam rozš. eax
- cqto (cqo) rdx:rax := znam rozš. rax
  - **c**onvert **b**yte|**w**ord|**l**ong|**q**uad **t**o **w**ord|**l**ong|**q**uad|**o**cta
  - **c**onvert **w**ord|**l**ong **t**o **d**ouble word|long
  - (originálne Intel mená inštrukcií, e-extended)

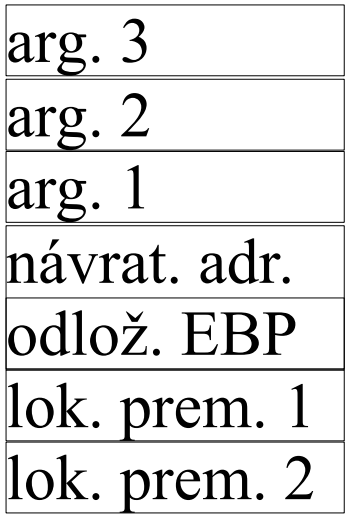
Práca so zásobníkom

- Zásobník umožňuje ukladať položky dát spôsobom LIFO.
- Na x86 v 32-bit. móde je veľkosť základnej položky 4B, v 64-bit móde 8B
- Zásobník rastie smerom „dolu“ (t.j. od vyšších adries k nižším).
- Na najnovšiu položku ukazuje register ESP/RSP.
- Na adresovanie sa používajú registre ESP/RSP a EBP/RBP v nepriamych módoch s doplnkom.

Práca so zásobníkom

- Zásobník (stack) sa používa napr. pre uloženie lokálnych premenných a argumentov funkcií (ak sa nepoužívajú registre).
  - Pred volaním funkcie sa uložia hodnoty argumentov do zásobníka.
  - Inštrukcia call uloží do zásobníka návratovú adr.
  - Vyhradí sa miesto pre lokálne premenné.
  - Pred návratom sa odstránia lok. premenné.
  - Inštrukcia ret vyberie návratovú adresu.
  - Odstránia sa argumenty.

Práca so zásobníkom (C volacia konvencia 32-bit)



16(%ebp)
12(%ebp)
8(%ebp)

(%ebp)
-4(%ebp)
-8(%ebp)

Volanie funkcie

```
push arg3
push arg2
push arg1
call f
add $12, %esp
```

Začiatok funkcie

```
push %ebp
mov %esp, %ebp
sub $8, %esp
```

Koniec funkcie

```
mov %ebp, %esp
pop %ebp
ret
```

Koniec funkcie

```
leave
ret
```

Práca so zásobníkom (Pascal volacia konvencia)

- Argumenty sa ukladajú v poradí (pri C v opačnom poradí).
- Argumenty neodstraňuje volajúci, ale inštrukcia `ret n`, kde n je počet B, ktoré majú byť odstránené zo zásobníka.
- Neumožňuje funkcie s premenlivým počtom argumentov.

Registrová volacia konvencia

- argumenty sa odovzdávajú v registroch
  - v prípade veľa argumentov sa použije hybridná konvencia s ďalšími argumentami v zásobníku
- na x86 v 64-bit móde na Linux-e:
  - prvých 6 argumentov je v rdi, rsi, rdx, rcx, r8, r9
  - ďalšie argumenty sú v zásobníku ako pri C konvencii (teda 7. sa vloží ako posledný pred call)
  - ak funkcia používa variabilný počet argumentov, v al treba ešte poslať počet použitých xmm registrov (0)
    - tie sa používajú na odovzdávanie floating-point argumentov

Hybridná volacia konvencia x86_64

arg. 8	24(%rbp)
arg. 7	16(%rbp)
návrat. adr.	
odlož. RBP	(%rbp)
lok. prem. 1	-8(%rbp)
lok. prem. 2	-16(%rbp)

Volanie funkcie

```
mov arg1, %rdi
mov arg2, %rsi
...
mov arg6, %r9
push arg8
push arg7
call f
add $16, %rsp
```

Začiatok funkcie

```
push %rbp
mov %rsp, %rbp
sub $16, %rsp
```

Koniec funkcie

```
mov %rbp, %rsp
pop %rbp
ret
```

Koniec funkcie

```
leave
ret
```

Hybridná volacia konvencia x86_64

24(%rbp)	argument 8
16(%rbp)	argument 7
8(%rbp)	návratová adresa
(%rbp)	uložená predchádzajúca hodnota rbp
-8(%rbp)	lokálna premenná
...	...
(%esp)	lokálna premenná
	„red zone“
-128(%esp)	

- red zone
  - oblasť, ktorú môže funkcia využiť na dočasné hodnoty, ktoré nepotrebuje zachovať, keď zavolá ďalšiu funkciu

Vracanie hodnôt z funkcie

- Malé hodnoty sa zvyčajne vracajú v registri.
  - na x86 v registri RAX (EAX v 32-bit móde), prípadne dvojici RDX:RAX (EDX:EAX v 32-bit)
- Veľké hodnoty (napr. štruktúra) sa vracajú tak, že volajúci poskytne (ako argument) adresu, kam sa má hodnota výsledku uložiť.
  - v Linuxe na x86 sa posiela ako prvý argument
    - 64-bit: v rdi
    - 32-bit: posledný vložený v zásobníku, odstraňuje sa použitím inštrukcie ret $4
  - pozor – v 64-bit móde sa aj zarovnané štruktúry do veľkosti 16B vracajú v RDX:RAX

Používanie registrov vo funkcii

- funkcia musí zachovať hodnoty registrov
  - 64-bit mód: RSP, RBP, RBX, R12 – R15
  - 32-bit mód: EBP, ESP, EBX, ESI, EDI
  - ak ich chce zmeniť, musí uložiť ich stav na začiatku a obnoviť pred návratom
- voľne môže meniť
  - 64-bit mód: RAX, RCX, RDX, RSI, RDI, R9-R11
  - 32-bit mód: EAX, ECX, EDX

Niektoré špeciality x86_64

- vplyv operácií s 8/16/32-bit registrami na zvyšok 64-bit registra
  - 8 a 16 – zvyšných 56 alebo 48 bitov sa neovplyvní
  - 32 – horných 32 bitov sa vynuluje
- 64-bit priama hodnota
  - dá sa použiť s inštrukciou mov
  - iné inštrukcie zvyčajne akceptujú len 32-bit priamu hodnotu, ktorú znamienkovo rozšíria na 64 bitov
- 64-bit priama adresa
  - len mov z/do RAX, EAX, AX, AL

Príklad assembler + C

```
.text
.global f
f:
push %rbp
mov %rsp, %rbp

mov %rdi, %rax
add %rsi, %rax

mov %rbp, %rsp
pop %rbp
ret
```

```
extern long f(long a, long b);
void g()
{
  long c;
  c = f(10, 20);
}
```

Assembler – prekladač

- prekladá program v jazyku assemblera do strojového kódu
- vytvára pomocné informácie o module potrebné pre linker a loader
- vstupný riadok má tvar
  `[návestie:] inštrukcia|direktíva operandy`
- návestie slúži na odkazovanie sa na adresu
- hodnotu návestiu priradí assembler

Assembler – prekladač

- lokálne návestia
  - môžu sa predefinovávať
  - majú tvar N:, kde N je kladné číslo
  - odkazuje sa na ne:
    - nasledujúci výskyt: Nf
    - predchádzajúci výskyt: Nb
- definícia konštánt
  - MENO = výraz
- výrazy – bežné aritm. operácie

Assembler – prekladač

- na začiatku nastaví hodnotu LC = 0
- keď potrebuje priradiť hodnotu návestiu, použije aktuálnu hodnotu LC
- LC zvyšuje vždy o dĺžku inštrukcie, resp. podľa pokynu direktívy (napr. pri definovaní dát podľa dĺžky dát)
- podľa počtu prechodov cez vstup delíme assemblery na jednoprechodové a dvojprechodové

Dvojprechodový assembler

- 1. prechod
  - číta vstupný text a priraďuje adresy každej inštrukcii alebo dátam
  - vytvára tabuľku symbolov obsahujúcu mená návestí a príslušnú hodnotu LC v čase definície návestia
  - môže doplniť vstupný text o ďalšie informácie
- 2. prechod
  - generuje strojový kód, využíva informácie z 1. p. (hlavne tabuľku symbolov)

Jednoprechodový assembler

- číta vstup len raz
- problém vzniká s návestiami, ktoré sú definované neskôr, ako sú použité
- musí vytvoriť zoznam nedefinovaných návestí s informáciou, kam treba doplniť ich hodnotu, a po skončení prechodu dopniť na príslušné miesta adresy návestí

Makrá

- makro – pomenovaná postupnosť inštrukcií
  - môže mať aj parametre
- pri použití sa nahradí svojím telom
- makro vs. funkcie(podprogramy)
  - volanie funkcie je opráciou procesora
  - „volanie“ makra procesor nevidí
  - makro je vhodné
    - pre krátke postupnosti
    - keď je dôležitá rýchlosť

Makrá

- definícia

```
.macro pridaj co, kam=%eax
   add \co, \kam
.endm
```

- použitie
  - `pridaj $6, %ebx`
  - `pridaj $6`
  - `pridaj kam=%ecx, co=$6`

Makroprocesor

- úloha
  - nájsť a uložiť definície makier
  - nájsť volania makier a nahradiť ich telom makra (so substitúciou parametrov)
- makroprocesor môže byť nezávislý od assemblera alebo môže byť integrovaný
- podľa počtu prechodov delíme makroprocesory na jednoprechodové a dvojprechodové

Dvojprechodový makroprocesor

- 1. prechod
  - hľadá definície makier a ukladá si ich do tabuľky makier spolu so zoznamom parametrov a ich default hodnotami
- 2. prechod
  - číta zdrojový text, ignoruje definície makier
  - ak nájde volanie makra, rozvinie ho podľa jeho definície v tabuľke
  - inak skopíruje riadok na výstup bez zmeny
- nemôže spracovať vnorené definície makier

Jednoprechodový makroprocesor

- v jednom prechode hľadá definície a zároveň robí rozvíjanie makier
- definícia makra musí predchádzať jeho použitiu
- môže byť zakomponovaný do prvého prechodu assemblera

Linker (spájač)

- programy pozostávajú z modulov
- každý modul je samostatne spracovaný kompilátorom alebo assemblerom
  - výsledkom je „object file“ – objektový súbor
- linker spája jednotlivé objektové súbory
  - spája jednotlivé sekcie rovnakého typu z rôznych modulov
  - relokuje adresy

Objektový súbor

- identifikácia
  - meno modulu, čas kompilácie, informácie pre linker
- tabuľka symbolov
  - meno, typ, sekcia a relatívna adresa
  - globálne symboly
  - externé symboly (t.j. nedefinované v module)
  - lokálne symboly (nie je nutné, ale môže sa hodiť)

Objektový súbor

- relokačná tabuľka
  - adresa miesta (kam), špecifikácia hodnoty (napr. meno symbolu), ktorú je treba doplniť a akým spôsobom (napr. pripočítať, pripočítať rozdiel)
- obsah inicializovaných sekcií
  - najmä sekcia .text (kód) a .data (inicializované dáta)
- adresa začiatku programu
- ladiace informácie

Knižnice

- mnohé funkcie sú často využívané mnohými programami
  - často sa implementujú v samostatných moduloch, ktoré sa potom pripájajú k programom
  - takéto moduly sa zvyčajne uložia do tzv. knižníc (archívov objektových súborov)
  - z knižnice linker potom vyberie potrebné moduly
  - príkladom je štandardná knižnica libc, ktorá obsahuje štandardné funkcie jazyka C vrátane rozhraní k systémovým volaniam

Práca linker-a

- načíta informácie o jednotlivých moduloch, buduje globálnu tabuľku symbolov
  - ak nájde nedefinované symboly, prehľadá určené knižnice, aby v nich našiel moduly, ktoré príslušné symboly definujú
  - ak zostanú nedefinované symboly, vyhlási chybu
- pridelí miesto v pamäti jednotlivým sekciám zo všetkých modulov (sekcie rovnakého typu ukladá za seba)

Práca linker-a

- vykoná relokácie
  - na miesta určené údajmi z relokačnej tabuľky pripočíta príslušné hodnoty (adresa symbolu, začiatok sekcie, rozdiel adresy symbolu a miesta relokácie)
- zapíše výsledok do súboru
  - absolute load module – obsahuje definitívny obraz programu, musí sa zaviesť na určenú adresu
  - relative load module – obsahuje relokačnú tabuľku, je potrebné robiť ďalšie relokácie

Loader (zavádzač)

- načíta súbor vytvorený linkerom
- umiestni ho do pamäte
  - v prípade relative load module vykoná relokácie podľa skutočnej adresy
- pridelí pamäť pre neinicializované dáta (sekcia .bss)
- odovzdá riadenie programu

Čas viazania (binding)

- kedy je ukončené mapovanie symbolických mien na fyzické adresy
  - počas písania programu
  - počas prekladu programu
  - počas linkovania (absolute load module)
  - počas loadovania
  - pri uložení bázovej adresy do registra
  - pri vykonávaní inštrukcie
- virtuálne vs. fyzické adresy

Dynamické linkovanie

- moduly môžu byť prilinkované až pri spustení programu alebo až pred prvým použitím funkcie z modulu
- štandardné knižničné funkcie
  - šetrí diskový priestor, nevyžaduje nové linkovanie všetkých programov pri zmene knižničnej funkcie
- „rozšírenia“ programov
  - šetrí pamäť, umožňuje konfigurovať funkcionalitu bez potreby linkovania

Príklad (32-bit)

a.s

```
.data

a:      .long 0

d:      .long 4

.text
.global f

f:      mov d, %eax
        add a, %eax
        mov %eax, b
        ret
```

b.s

```
.bss
.global b

x:      .long 0
        .long 0
b:      .long 0

.text
.global _start
_start:
        call f
        mov b, %eax
        pushl $0
        call _exit
```

Príklad (32-bit)

```
$ as --32 -o a.o a.s
$ objdump -d a.o

a.o:     file format elf32-i386


Disassembly of section .text:

00000000 <f>:
   0:  a1 04 00 00 00          mov    0x4,%eax
   5:  03 05 00 00 00 00       add    0x0,%eax
   b:  a3 00 00 00 00          mov    %eax,0x0
  10:  c3                      ret
```

Príklad (32-bit)

```
$ objdump -rt a.o

a.o:     file format elf32-i386

SYMBOL TABLE:
00000000 l    d  .text 00000000 .text
00000000 l    d  .data 00000000 .data
00000000 l    d  .bss  00000000 .bss
00000000 l       .data 00000000 a
00000004 l       .data 00000000 d
00000000 g       .text 00000000 f
00000000         *UND* 00000000 b


RELOCATION RECORDS FOR [.text]:
OFFSET   TYPE              VALUE
00000001 R_386_32          .data
00000007 R_386_32          .data
0000000c R_386_32          b
```

Príklad (32-bit)

```
$ as --32 -o b.o b.s
$ objdump -d b.o

b.o:     file format elf32-i386


Disassembly of section .text:

00000000 <_start>:
   0:  e8 fc ff ff ff          call   1 <_start+0x1>
   5:  a1 00 00 00 00          mov    0x0,%eax
   a:  6a 00                   push   $0x0
   c:  e8 fc ff ff ff          call   d <_start+0xd>
```

Príklad (32-bit)

```
$ objdump -rt b.o

b.o:     file format elf32-i386

SYMBOL TABLE:
00000000 l    d  .text 00000000 .text
00000000 l    d  .data 00000000 .data
00000000 l    d  .bss  00000000 .bss
00000000 l       .bss  00000000 x
00000008 g       .bss  00000000 b
00000000 g       .text 00000000 _start
00000000         *UND* 00000000 f
00000000         *UND* 00000000 _exit


RELOCATION RECORDS FOR [.text]:
OFFSET   TYPE              VALUE
00000001 R_386_PC32        f
00000006 R_386_32          b
0000000d R_386_PC32        _exit
```

Príklad (32-bit)

```
$ ld -m elf_i386 -o rel.o -r a.o b.o
$ objdump -d rel.o

rel.o:     file format elf32-i386


Disassembly of section .text:

00000000 <f>:
   0:  a1 04 00 00 00          mov    0x4,%eax
   5:  03 05 00 00 00 00       add    0x0,%eax
   b:  a3 00 00 00 00          mov    %eax,0x0
  10:  c3                      ret
  11:  90                      nop
  12:  90                      nop
  13:  90                      nop

00000014 <_start>:
  14:  e8 fc ff ff ff          call   15 <_start+0x1>
  19:  a1 00 00 00 00          mov    0x0,%eax
  1e:  6a 00                   push   $0x0
  20:  e8 fc ff ff ff          call   21 <_start+0xd>
```

Príklad (32-bit)

```
$ objdump -rt rel.o

rel.o:     file format elf32-i386

SYMBOL TABLE:
00000000 l    d  .text  00000000 .text
00000000 l    d  .data  00000000 .data
00000000 l    d  .bss   00000000 .bss
00000000 l       .data  00000000 a
00000004 l       .data  00000000 d
00000000 l       .bss   00000000 x
00000008 g       .bss   00000000 b
00000000 g       .text  00000000 f
00000014 g       .text  00000000 _start
00000000         *UND*  00000000 _exit


RELOCATION RECORDS FOR [.text]:
OFFSET   TYPE              VALUE
00000001 R_386_32          .data
00000007 R_386_32          .data
0000000c R_386_32          b
00000015 R_386_PC32        f
0000001a R_386_32          b
00000021 R_386_PC32        _exit
```

Príklad (32-bit)

```
$ ld -m elf_i386 --dynamic-linker=/lib/ld-linux.so.2 -o dyn a.o b.o -lc
$ objdump -d dyn
dyn:     file format elf32-i386
Disassembly of section .plt:
08048170 <_exit@plt-0x10>:
 8048170:    ff 35 98 92 04 08    pushl  0x8049298
 8048176:    ff 25 9c 92 04 08    jmp    *0x804929c
 804817c:    00 00                add    %al,(%eax)
    ...
08048180 <_exit@plt>:
 8048180:    ff 25 a0 92 04 08    jmp    *0x80492a0
 8048186:    68 00 00 00 00       push   $0x0
 804818b:    e9 e0 ff ff ff       jmp    8048170 <_exit@plt-0x10>
Disassembly of section .text:
08048190 <f>:
 8048190:    a1 a8 92 04 08       mov    0x80492a8,%eax
 8048195:    03 05 a4 92 04 08    add    0x80492a4,%eax
 804819b:    a3 b4 92 04 08       mov    %eax,0x80492b4
 80481a0:    c3                   ret
 80481a1:    90                   nop
 80481a2:    90                   nop
 80481a3:    90                   nop
080481a4 <_start>:
 80481a4:    e8 e7 ff ff ff       call   8048190 <f>
 80481a9:    a1 b4 92 04 08       mov    0x80492b4,%eax
 80481ae:    6a 00                push   $0x0
 80481b0:    e8 cb ff ff ff       call   8048180 <_exit@plt>
```

Príklad (32-bit)

```
$ objdump -rt dyn
dyn:     file format elf32-i386
SYMBOL TABLE:
080480d4 l    d  .interp   00000000              .interp
080480e8 l    d  .hash 00000000              .hash
080480fc l    d  .dynsym   00000000              .dynsym
0804811c l    d  .dynstr   00000000              .dynstr
08048136 l    d  .gnu.version   00000000              .gnu.version
0804813c l    d  .gnu.version_r 00000000              .gnu.version_r
0804815c l    d  .rel.plt  00000000              .rel.plt
08048170 l    d  .plt  00000000              .plt
08048190 l    d  .text 00000000              .text
080481b8 l    d  .eh_frame  00000000              .eh_frame
080491f4 l    d  .dynamic  00000000              .dynamic
08049294 l    d  .got.plt  00000000              .got.plt
080492a4 l    d  .data 00000000              .data
080492ac l    d  .bss  00000000              .bss
080492a4 l       .data 00000000              a
080492a8 l       .data 00000000              d
080492ac l       .bss  00000000              x
080491f4 l     O .dynamic  00000000              _DYNAMIC
08049294 l     O .got.plt  00000000              _GLOBAL_OFFSET_TABLE_
080492b4 g       .bss  00000000              b
080492ac g       *ABS* 00000000              _edata
08048190 g       .text 00000000              f
00000000       F *UND* 00000000              _exit@@GLIBC_2.0
080492b8 g       *ABS* 00000000              _end
080481a4 g       .text 00000000              _start
080492ac g       *ABS* 00000000              __bss_start
```

Príklad (32-bit)

```
$ objdump -RT dyn

dyn:     file format elf32-i386

DYNAMIC SYMBOL TABLE:
00000000      DF *UND* 00000000  GLIBC_2.0   _exit


DYNAMIC RELOCATION RECORDS
OFFSET   TYPE              VALUE
080492a0 R_386_JUMP_SLOT   _exit
```

Typy relokácií (32-bit)

- Nech inštrukcia obsahuje zapísanú adresu A, záznam v relokačnej tabuľke pre miesto P odkazuje na symbol s hodnotou S
- Pri relokácii sa zmení nasledovne:
  - R_386_32
    - A + S
    - používa sa v súvislosti s adr. módom priama adresa
    - S je buď začiatok sekcie alebo priamo adresa zodpovedajúca konkrétnemu symbolu
  - R_386_PC32
    - A + S – P
    - používa sa v súvislosti s adr. módom relatívna adresa pre globálne a externé symboly
  - R_386_JUMP_SLOT
    - S
    - používa sa v súvislosti s funkciami z dynamicky linkovaných knižníc

Relokácie pri dyn. linkovaní

- vytvorí sa sekcia .plt (Procedure Linkage Table)
  - v nej sa vytvoria „náhrady“ funkcií
    - skáču na adresu z tabuľky (tzv. GOT – Global Offset Table)
    - pre položku GOT existuje relokácia typu R_386_JUMP_SLOT
    - adresu do GOT doplní dynamický linker
      - často až pri prvom volaní funkcie

Príklad (64-bit)

a.s

```
.data

a:  .long 0

d:  .long 4

.text
.global f

f:  mov d, %eax
    add a, %eax
    mov %eax, b
    ret
```

b.s

```
.bss
.global b

x:  .long 0
    .long 0
b:  .long 0

.text
.global _start
_start:
    call f
    mov b, %eax
    xor %rdi, %rdi
    call _exit
```

Príklad (64-bit)

```
$ as -o a.o a.s
$ objdump -d a.o

a.o:     file format elf64-x86-64


Disassembly of section .text:

0000000000000000 <f>:
   0:  8b 04 25 00 00 00 00   mov    0x0,%eax
   7:  03 04 25 00 00 00 00   add    0x0,%eax
   e:  89 04 25 00 00 00 00   mov    %eax,0x0
  15:  c3                     retq
```

Príklad (64-bit)

```
$ objdump -rt a.o
a.o:     file format elf64-x86-64

SYMBOL TABLE:
0000000000000000 l    d  .text   0000000000000000 .text
0000000000000000 l    d  .data   0000000000000000 .data
0000000000000000 l    d  .bss 0000000000000000 .bss
0000000000000000 l       .data   0000000000000000 a
0000000000000004 l       .data   0000000000000000 d
0000000000000000 g       .text   0000000000000000 f
0000000000000000         *UND*   0000000000000000 b


RELOCATION RECORDS FOR [.text]:
OFFSET           TYPE              VALUE
0000000000000003 R_X86_64_32S      .data+0x0000000000000004
000000000000000a R_X86_64_32S      .data
0000000000000011 R_X86_64_32S      b
```

Príklad (64-bit)

```
$ as -o b.o b.s
$ objdump -d b.o
b.o:     file format elf64-x86-64


Disassembly of section .text:

0000000000000000 <_start>:
   0:  e8 00 00 00 00          callq  5 <_start+0x5>
   5:  8b 04 25 00 00 00 00    mov    0x0,%eax
   c:  48 31 ff                xor    %rdi,%rdi
   f:  e8 00 00 00 00          callq  14 <b+0xc>
```

Príklad (64-bit)

```
$ objdump -rt b.o

b.o:     file format elf64-x86-64

SYMBOL TABLE:
0000000000000000 l    d  .text   0000000000000000 .text
0000000000000000 l    d  .data   0000000000000000 .data
0000000000000000 l    d  .bss 0000000000000000 .bss
0000000000000000 l       .bss 0000000000000000 x
0000000000000008 g       .bss 0000000000000000 b
0000000000000000 g       .text   0000000000000000 _start
0000000000000000         *UND*   0000000000000000 f
0000000000000000         *UND*   0000000000000000 _exit


RELOCATION RECORDS FOR [.text]:
OFFSET           TYPE              VALUE
0000000000000001 R_X86_64_PC32     f+0xfffffffffffffffc
0000000000000008 R_X86_64_32S      b
0000000000000010 R_X86_64_PC32     _exit+0xfffffffffffffffc
```

Príklad (64-bit)

```
$ ld --dynamic-linker=/lib64/ld-linux-x86-64.so.2 -o dyn a.o b.o -lc
$ objdump -d dyn
dyn:     file format elf64-x86-64
Disassembly of section .plt:
0000000000400220 <_exit@plt-0x10>:
  400220:    ff 35 d2 01 20 00    pushq  0x2001d2(%rip)# 6003f8 <_GLOBAL_OFFSET_TABLE_+0x8>
  400226:    ff 25 d4 01 20 00    jmpq   *0x2001d4(%rip)# 600400 <_GLOBAL_OFFSET_TABLE_+0x10>
  40022c:    0f 1f 40 00          nopl   0x0(%rax)
0000000000400230 <_exit@plt>:
  400230:    ff 25 d2 01 20 00    jmpq   *0x2001d2(%rip)# 600408 <_GLOBAL_OFFSET_TABLE_+0x18>
  400236:    68 00 00 00 00       pushq  $0x0
  40023b:    e9 e0 ff ff ff       jmpq   400220 <_exit@plt-0x10>
Disassembly of section .text:
0000000000400240 <f>:
  400240:    8b 04 25 14 04 60 00 mov    0x600414,%eax
  400247:    03 04 25 10 04 60 00 add    0x600410,%eax
  40024e:    89 04 25 20 04 60 00 mov    %eax,0x600420
  400255:    c3                   retq
  400256:    90                   nop
  400257:    90                   nop
0000000000400258 <_start>:
  400258:    e8 e3 ff ff ff       callq  400240 <f>
  40025d:    8b 04 25 20 04 60 00 mov    0x600420,%eax
  400264:    48 31 ff             xor    %rdi,%rdi
  400267:    e8 c4 ff ff ff       callq  400230 <_exit@plt>
```

Príklad (64-bit)

```
$ objdump -rt dyn
dyn:     file format elf64-x86-64

SYMBOL TABLE:
0000000000400158 l    d  .interp    0000000000000000              .interp
0000000000400178 l    d  .hash  0000000000000000              .hash
0000000000400190 l    d  .dynsym    0000000000000000              .dynsym
00000000004001c0 l    d  .dynstr    0000000000000000              .dynstr
00000000004001de l    d  .gnu.version    0000000000000000              .gnu.version
00000000004001e8 l    d  .gnu.version_r  0000000000000000              .gnu.version_r
0000000000400208 l    d  .rela.plt  0000000000000000              .rela.plt
0000000000400220 l    d  .plt   0000000000000000              .plt
0000000000400240 l    d  .text  0000000000000000              .text
0000000000400270 l    d  .eh_frame  0000000000000000              .eh_frame
00000000006002b0 l    d  .dynamic   0000000000000000              .dynamic
00000000006003f0 l    d  .got.plt   0000000000000000              .got.plt
0000000000600410 l    d  .data  0000000000000000              .data
0000000000600418 l    d  .bss   0000000000000000              .bss
0000000000600410 l       .data  0000000000000000              a
0000000000600414 l       .data  0000000000000000              d
0000000000600418 l       .bss   0000000000000000              x
00000000006002b0 l     O .dynamic   0000000000000000              _DYNAMIC
00000000006003f0 l     O .got.plt   0000000000000000              _GLOBAL_OFFSET_TABLE_
0000000000000000       F *UND*  0000000000000000              _exit@@GLIBC_2.2.5
0000000000600420 g       .bss   0000000000000000              b
0000000000600418 g       *ABS*  0000000000000000              _edata
0000000000400240 g       .text  0000000000000000              f
0000000000600428 g       *ABS*  0000000000000000              _end
0000000000400258 g       .text  0000000000000000              _start
0000000000600418 g       *ABS*  0000000000000000              __bss_start
```

Príklad (64-bit)

```
$ objdump -RT dyn
dyn:     file format elf64-x86-64

DYNAMIC SYMBOL TABLE:
0000000000000000      DF *UND*   0000000000000000
GLIBC_2.2.5 _exit


DYNAMIC RELOCATION RECORDS
OFFSET           TYPE              VALUE
0000000000600408 R_X86_64_JUMP_SLOT  _exit
```

Typy relokácií (64-bit)

- Nech inštrukcia obsahuje zapísanú adresu A, záznam v relokačnej tabuľke pre miesto P odkazuje na symbol s hodnotou S
- Pri relokácii sa zmení nasledovne:
  - R_X86_64_32S
    - A + S
    - používa sa v súvislosti s adr. módom priama adresa
    - S je buď začiatok sekcie alebo priamo adresa zodpovedajúca konkrétnemu symbolu
  - R_X86_64_PC32
    - A + S – P
    - používa sa v súvislosti s adr. módom relatívna adresa pre globálne a externé symboly
  - R_X86_64_JUMP_SLOT
    - S
    - používa sa v súvislosti s funkciami z dynamicky linkovaných knižníc

Position Independent Code

- kód, ktorý je možné zaviesť na ľubovoľnú adresu bez potreby úprav
  - napr. v dynamicky linkovaných knižniciach
    - aby bolo možné jednu fyzickú kópiu namapovať do adresného priestoru rôznych procesov (a nie nutne na rovnakú virtuálnu adresu)
  - tiež na umožnenie randomizácie adresného priestoru procesu
    - sťaženie niektorých útokov na chybné programy
  - nesmie používať adresný mód priama adresa
    - lebo ten si vyžaduje relokáciu
    - treba všetko riešiť relatívnymi adresami alebo nepriamo

Príklad (64-bit PIC)

a.s

```
.data

a:  .long 0

d:  .long 4

.text
.global f

f:  mov d(%rip), %eax
    add a(%rip), %eax
    mov %eax, b(%rip)
    ret
```

b.s

```
.bss
.global b

x:  .long 0
    .long 0
b:  .long 0

.text
.global _start
_start:
    call f
    mov b(%rip), %eax
    xor %rdi, %rdi
    call _exit@plt
```

Príklad (64-bit PIC)

```
$ as -o a.o a.s
$ objdump -d a.o
a.o:     file format elf64-x86-64


Disassembly of section .text:

0000000000000000 <f>:
   0:  8b 05 00 00 00 00       mov    0x0(%rip),%eax
   6:  03 05 00 00 00 00       add    0x0(%rip),%eax
   c:  89 05 00 00 00 00       mov    %eax,0x0(%rip)
  12:  c3                      retq
```

Príklad (64-bit PIC)

```
$ objdump -rt a.o
a.o:     file format elf64-x86-64

SYMBOL TABLE:
0000000000000000 l    d  .text   0000000000000000 .text
0000000000000000 l    d  .data   0000000000000000 .data
0000000000000000 l    d  .bss 0000000000000000 .bss
0000000000000000 l       .data   0000000000000000 a
0000000000000004 l       .data   0000000000000000 d
0000000000000000 g       .text   0000000000000000 f
0000000000000000         *UND*   0000000000000000 b


RELOCATION RECORDS FOR [.text]:
OFFSET           TYPE              VALUE
0000000000000002 R_X86_64_PC32     .data
0000000000000008 R_X86_64_PC32     .data+0xfffffffffffffffc
000000000000000e R_X86_64_PC32     b+0xfffffffffffffffc
```

Príklad (64-bit PIC)

```
$ as -o b.o b.s
$ objdump -d b.o
b.o:     file format elf64-x86-64


Disassembly of section .text:

0000000000000000 <_start>:
   0:  e8 00 00 00 00          callq  5 <_start+0x5>
   5:  8b 05 00 00 00 00       mov    0x0(%rip),%eax
   b:  48 31 ff                xor    %rdi,%rdi
   e:  e8 00 00 00 00          callq  13 <b+0xb>
```

Príklad (64-bit PIC)

```
$ objdump -rt b.o
b.o:     file format elf64-x86-64

SYMBOL TABLE:
0000000000000000 l    d  .text   0000000000000000 .text
0000000000000000 l    d  .data   0000000000000000 .data
0000000000000000 l    d  .bss 0000000000000000 .bss
0000000000000000 l       .bss 0000000000000000 x
0000000000000008 g       .bss 0000000000000000 b
0000000000000000 g       .text   0000000000000000 _start
0000000000000000         *UND*   0000000000000000 f
0000000000000000         *UND*   0000000000000000
_GLOBAL_OFFSET_TABLE_
0000000000000000         *UND*   0000000000000000 _exit


RELOCATION RECORDS FOR [.text]:
OFFSET           TYPE              VALUE
0000000000000001 R_X86_64_PC32     f+0xfffffffffffffffc
0000000000000007 R_X86_64_PC32     b+0xfffffffffffffffc
000000000000000f R_X86_64_PLT32    _exit+0xfffffffffffffffc
```

Príklad (64-bit PIC)

```
$ ld -pie --dynamic-linker=/lib64/ld-linux-x86-64.so.2 -o dyn a.o b.o -lc
$ objdump -d dyn
dyn:     file format elf64-x86-64
Disassembly of section .plt:
00000000000002c0 <_exit@plt-0x10>:
 2c0:    ff 35 ca 01 20 00    pushq  0x2001ca(%rip)
 2c6:    ff 25 cc 01 20 00    jmpq   *0x2001cc(%rip)
 2cc:    0f 1f 40 00          nopl   0x0(%rax)
00000000000002d0 <_exit@plt>:
 2d0:    ff 25 ca 01 20 00    jmpq   *0x2001ca(%rip)
 2d6:    68 00 00 00 00       pushq  $0x0
 2db:    e9 e0 ff ff ff       jmpq   2c0 <_exit@plt-0x10>
Disassembly of section .text:
00000000000002e0 <f>:
 2e0:    8b 05 c6 01 20 00    mov    0x2001c6(%rip),%eax        # 2004ac <d>
 2e6:    03 05 bc 01 20 00    add    0x2001bc(%rip),%eax        # 2004a8 <a>
 2ec:    89 05 c6 01 20 00    mov    %eax,0x2001c6(%rip)        # 2004b8 <b>
 2f2:    c3                   retq
 2f3:    90                   nop
00000000000002f4 <_start>:
 2f4:    e8 e7 ff ff ff       callq  2e0 <f>
 2f9:    8b 05 b9 01 20 00    mov    0x2001b9(%rip),%eax        # 2004b8 <b>
 2ff:    48 31 ff             xor    %rdi,%rdi
 302:    e8 c9 ff ff ff       callq  2d0 <_exit@plt>
```

Príklad (64-bit PIC) v C

```
#include <stdio.h>
int main()
{
  printf("%p\n", main);
  return 0;
}
```

```
$ gcc -o pietest pietest.c
$ gcc -pie -fpic -o pietest2 pietest.c
$ ./pietest
0x40050c
$ ./pietest
0x40050c
$ ./pietest2
0x7fba3b357840
$ ./pietest2
0x7f3781ec2840
```

Systémové volania

- umožňujú programom využívať služby OS
- na spodnej úrovni sú realizované využitím špeciálnych inštrukcií, ktoré zabezpečia predanie riadenie do jadra OS
  - softvérové prerušenie (Linux na i386: int $0x80)
  - volacie brány (call gates)
  - špeciálne inštrukcie
- programy využívajú štand. knižnice, ktoré zakrývajú detaily volania

Vytváranie nového procesu

- `pid_t fork(void)`
  - sys/types.h, unistd.h
  - vytvorí kópiu procesu
  - rodičovi vráti PID nového procesu
  - dieťaťu vráti 0
  - pri chybe vráti -1
  - oba procesy sú inak identické a môžu pokračovať v činnosti nezávisle na sebe

Ukončenie procesu

- `void exit(int status)`
  - stdlib.h
  - ukončí proces s návratovou hodnotou status
- `pid_t wait(int *status)`
  - sys/types.h, sys/wait.h
  - počká na ukončenie dieťaťa, vráti jeho PID
  - ak status != NULL, uloží informácie o návratovej hodnote (`WEXITSTATUS(hodnota)`)

„Démonizácia“ procesu

- `int daemon(int nochdir, int noclose)`
  - nochdir – prikazuje nezmeniť aktuálny adresár na /
  - noclose – prikazuje nepresmerovať štand. vstup a výstup na /dev/null
  - funkcia zabezpečí odpojenie procesu od riadiaceho terminálu a jeho beh na pozadí
  - vráti 0 (ok) alebo -1 (chyba)

Vstup/výstup

- File Descriptor – číslo identifikujúce otvorený súbor/zariadenie/socket/...
  - 0 – štandardný vstup
  - 1 – štandardný výstup
  - 2 – štandardný chybový výstup
- Operácie
  - open, close – otvorenie, zatvorenie
  - read, write, lseek – čítanie, zápis, posun pozície

open

- `int open(const char *pathname, int flags[, mode_t mode])`
  - `sys/types.h, sys/stat.h, fcntl.h`
  - otvorí súbor pathname spôsobom určeným flags, pri vytvorení súboru požaduje práva mode
  - vráti -1 pri chybe, inak deskriptor
  - flags
    - O_RDONLY – otvorí len na čítanie
    - O_WRONLY – otvorí len na zápis
    - O_RDWR – otvorí aj na čítanie aj na zápis

open, close

- flags môžu byť doplnené (|) o:
  - O_CREAT (ak neexistuje, vytvorí)
  - O_EXCL (chyba, ak existuje)
  - O_TRUNC (skráti na 0)
  - O_APPEND (pred každým zápisom sa posunie na koniec)
  - O_SYNC (synchrónne zápisy)
- `int close(int fd)`
  - unistd.h
  - zatvorí deskriptor fd, vrári 0 alebo -1

read, write

- `ssize_t read(int fd, void *buf, size_t count)`
  - unistd.h
  - prečíta z fd do buf najviac count bytov
  - vráti počet prečítaných alebo -1
- `ssize_t write(int fd, const void *buf, size_t count)`
  - unistd.h
  - zapíše do fd z buf count bytov
  - vráti počet zapísaných alebo -1

lseek, fsync

- `off_t lseek(int fildes, off_t offset, int whence)`
  - sys/types.h, unistd.h
  - nastaví aktuálnu pozíciu súboru filedes na offset bytov od miesta určeného whence:
    - SEEK_SET – od začiatku
    - SEEK_CUR – od aktuálnej pozície
    - SEEK_END – od konca
  - vráti novú aktuálnu pozíciu alebo -1
- `int fsync(int fd)`
  - vyprázdni (zapíše) cache

Spracovanie chýb

- Keď funkcia vráti -1, nastaví globálnu premennú errno (errno.h).
- `char *strerror(int errnum)`
  - string.h
  - vráti reťazec popisujúci zadanú chybu
- `void perror(const char *s)`
  - stdio.h
  - vypíše chybu určenú errno na stderr

Blokovanie procesu

- Štandardné volania I/O alebo wait zostanú čakať, kým úspešne neskončia.
- Spôsobuje to problém, ak proces potrebuje reagovať na viac ako jednu udalosť, napr.:
  - čítať z viac ako 1 deskriptoru,
  - občas volať wait, no robiť aj niečo iné.

waitpid

- `pid_t waitpid(pid_t pid, int *status, int options)`
  - sys/types.h, sys/wait.h
  - pid == -1 – čaká na ľubovoľné dieťa (ako wait)
  - options == WNOHANG – ak žiadne dieťa neskočilo, vráti 0 namiesto čakania

select

- ```
  int  select(int n,
  fd_set *readfds,
  fd_set *writefds,
  fd_set *exceptfds,
  struct timeval *timeout)
  ```
- `FD_CLR(int fd, fd_set *set)`
- `FD_ISSET(int fd, fd_set *set)`
- `FD_SET(int fd, fd_set *set)`
- `FD_ZERO(fd_set *set)`
  ```

select

- sys/select.h
- monitoruje 3 množiny deskriptorov:
  - readfds – či je možné čítať
  - writefds – či je možné zapisovať
  - exceptfds – či došlo k výnimkám
- timeout – určuje, ako dlho má select čakať, ak je NULL, tak nekonečne
  - struct timeval {
    long tv_sec; /* seconds */
    long tv_usec; /* microseconds */
    };
- n = max deskriptor + 1

select

- vráti počet deskriptorov, kde nastala očakávaná udalosť
  - 0 = došlo k timeoutu
  - -1 = došlo k chybe
- množiny obsahujú tie deskriptory, kde došlo k udalosti
- FD_CLR, FD_SET vymaže/pridá deskriptor do množiny
- FD_ZERO vyprázdni množinu
- FD_ISSET testuje prítomnosť v množine

Signály

- Signály sú udalosti (chyby, externé udalosti) na ktoré môže proces reagovať.
- `sighandler_t signal(int signum, sighandler_t handler)`
  - signal.h, typedef void (*sighandler_t)(int);
  - inštaluje handler pre signál signum
    - SIG_IGN – ignorovať
    - SIG_DFL – štandardné správanie
  - vráti predchádzajúci handler

Signály

- SIGHUP – zatvorenie riadiaceho terminálu, pri démonoch zvyčajne požiadavka na rekonfiguráciu
- SIGINT – Ctrl+C
- SIGPIPE – zápis do zatvorenej rúry/socketu
- SIGTERM – žiadosť o ukončenie procesu
- SIGQUIT – Ctrl+\ - žiadosť o okamžité skončenie
- SIGCHLD – ukončenie dieťaťa
- SIGKILL – násilné ukončenie procesu

Vyvolanie signálu

- `int raise(int sig)`
- `int kill(pid_t pid, int sig)`
  - signal.h, sys/types.h
  - raise vygeneruje určený signál
  - kill pošle určený signál určenému procesu
    - pid = -1 – každému procesu

„Automatický“ wait

```
#include <sys/types.h>
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <signal.h>
#include <sys/wait.h>

void childdied(int sig)
{
  while (waitpid(-1, NULL, WNOHANG) > 0);
  signal(SIGCHLD, childdied);
}

int main()
{
  signal(SIGCHLD, childdied);
  char buf[128];
  int pid;
  pid = fork();
  if (pid == -1) { perror("fork"); exit(1); }
  if (pid == 0)
  {
    ...
  }
  else
  {
    ...
  }
}
```

Vstup z terminálu

- štandardné čítanie z terminálu
  - bufferované po riadkoch
  - interpretujú sa niektoré špeciálne znaky
  - automaticky sa robí „echo“
- niekedy to nemusí byť vhodné
  - potrebujeme reagovať na jednotlivé znaky
  - potrebujeme detekovať začiatok písania (select)
  - nechceme zobrazovať vstup (heslá)

Nastavenie terminálu

- `int tcgetattr(int fd, struct termios *termios_p)`
- `int tcsetattr(int fd, int optional_actions, struct termios *termios_p)`
  - termios.h, unistd.h
  - man termios
  - tcgetattr získa, tcsetattr nastaví atribúty terminálu
  - optional_actions
    - TCSANOW – nastaví hneď
    - TCSADRAIN – nastaví po odoslaní výstupu
    - TCSAFLUSH – ako TCSADRAIN a navyše ignoruje prijatý a neprečítaný vstup

Nastavenie terminálu

- struct termios obsahuje
  - tcflag_t c_iflag; vstupné atribúty
  - tcflag_t c_oflag; výstupné atribúty
  - tcflag_t c_cflag; riadiace atribúty
  - tcflag_t c_lflag; lokálne atribúty
  - cc_t c_cc[NCCS]; riadiace znaky
    - VINTR Ctrl+C výskyt vyvolá signál SIGINT
    - VQUIT Ctrl+\ výskyt vyvolá signál SIGQUIT
    - VERASE DEL,BS výskyt vymaže predch. znak
    - VKILL Ctrl+U vymaže celý riadok
    - VEOF Ctr+D signalizuje koniec vstupu
    - VSUSP Ctrl+Z výskyt vyvolá signál SIGSTP

Nastavenie terminálu

- vybrané lokálne atribúty
  - ISIG
    - keď sa vo vstupe vyskytne znak definovaný pre generovanie signálu, znak sa odstráni zo vstupu a pošle sa príslušný signál
  - ICANON
    - povoľuje kanonický mód (štandardne povolený), v ktorom sa vstup bufferuje po riadkoch a interpretujú sa znaky na mazanie znaku/riadku a znak pre koniec vstupu (a niekoľko ďalších)
  - ECHO
    - zapína „automatické echo“, t.j. prijatý znak sa automaticky posiela na výstup

Nastavenie terminálu

- príklad – vypnutie kanonického módu

```
struct termios termattr;

if (tcgetattr(0, &termattr) < 0)
{
  perror("tcgetattr");
  exit(1);
}

termattr.c_lflag &= ~ICANON;

if (tcsetattr(0, TCSANOW, &termattr) < 0)
{
  perror("tcsetattr");
  exit(1);
}
```

Nastavenie terminálu

- príklad – zapnutie kanonického módu

```
struct termios termattr;

if (tcgetattr(0, &termattr) < 0)
{
  perror("tcgetattr");
  exit(1);
}

termattr.c_lflag |= ICANON;

if (tcsetattr(0, TCSANOW, &termattr) < 0)
{
  perror("tcsetattr");
  exit(1);
}
```

Sieťová komunikácia

- komunikuje sa pomocou socket-ov
  - socket predstavuje koncový bod komunikácie
  - so socket-om sa manipuluje prostredníctvom file descriptor-a, podobne ako so súborom
- typy socketov
  - SOCK_DGRAM
    - nespoľahlivá komunikácia bez spojenia, protokol UDP
  - SOCK_STREAM
    - spoľahlivá komunikácia so spojením (funguje ako rúra pre byty), protokol TCP

TCP

- server (čaká na pripojenie klienta)
  - vytvorí socket (socket)
  - nastaví adresu (bind)
  - začne počúvať (listen)
  - príjme spojenie (accept), dostane nový deskriptor
  - komunikuje na deskriptore prijatého spojenia (send, recv, write, read, select)
  - požiada o ukončenie spojenia (shutdown)
  - zatvorí (zruší) socket (close)

TCP

- klient (nadväzuje spojenie)
  - vytvorí socket (socket)
  - môže nastaviť adresu (bind)
  - požiada o spojenie (connect)
  - komunikuje na deskriptore (send, recv, write, read, select)
  - požiada o ukončenie spojenia (shutdown)
  - zatvorí (zruší) socket (close)

UDP

- server/klient
  - vytvorí socket (socket)
  - môže nastaviť lokálnu adresu (bind) a adresu partneta (connect)
  - komunikuje na deskriptore (sendto, recvfrom, send, recv, write, read, select)
    - send a write môže použiť, len ak použil connect
  - zatvorí (zruší) socket (close)
- dáta sa prenášajú po jednotlivých správach

Adresy

```
struct sockaddr_in {
    sa_family_t    sin_family; /* address family: AF_INET */
    u_int16_t      sin_port;   /* port in network byte order */
    struct in_addr  sin_addr;  /* internet address */
};

/* Internet address. */
struct in_addr {
    u_int32_t      s_addr;     /* address in network byte order */
};
```

čísla portov sa udávajú v „sieťovom“ formáte

Adresy

- `uint16_t htons(uint16_t hostshort)`
- `uint16_t ntohs(uint16_t netshort)`
  - netinet/in.h
  - htons konvertuje short do sieťového formátu
  - ntohs konvertuje short zo sieťového formátu
- `char *inet_ntoa(struct in_addr in)`
  - netinet/in.h, arpa/inet.h
  - vráti textovú reprezentáciu adresy

Adresy

- `int inet_aton(const char *cp, struct in_addr *inp)`
  - konvertuje textovú reprezentáciu IP adresy na struct in_addr a vráti nenulu, ak je adresa ok, 0 ak nie je
- `in_addr_t inet_addr(const char *cp)`
  - podobne ako inet_aton, ale vracia adresu, INADDR_NONE pri chybe (zodpovedá 255.255.255.255 !!!)
- INADDR_ANY – znamená ľubovoľnú adresu

socket

- int socket(int domain, int type, int protocol)
  - sys/types.h, sys/socket.h
  - vytvorí socket a vrári deskriptor
  - domain == PF_INET (pre IPv4)
  - type
    - SOCK_STREAM pre TCP
    - SOCK_DGRAM pre UDP
  - protocol == 0
  - vytvorí socket a vráti deskriptor

bind

- `int bind(int sockfd, struct sockaddr *my_addr, socklen_t addrlen)`
  - sys/types.h, sys/socket.h
  - nastaví lokálnu adresu (IP adresa + port) socketu
  - vráti 0 (ok) alebo -1 (chyba)

```
int fd;
struct sockaddr_in sa;

sa.sin_family = AF_INET;
sa.sin_addr.s_addr = INADDR_ANY;
sa.sin_port = htons(cislo_portu);

fd = socket(PF_INET, SOCK_STREAM, 0);
bind(fd, &sa, sizeof(sa));
```

listen

- `int listen(int s, int backlog)`
  - sys/socket.h
  - nastaví socket do počúvacieho módu
  - backlog = max. počet spojení čakajúcich na prijatie
  - vráti 0 (ok) alebo -1 (chyba)

accept

- `int accept(int s, struct sockaddr *addr, socklen_t *addrlen)`
  - sys/types.h, sys/socket.h
  - príjme spojenie na sockete v počúvacom stave
  - v addr vráti adresu druhej strany
  - addrlen ukazuje na premennú obsahujúcu dĺžku adresy (po návrate udáva skutočnú dĺžku)
  - vráti nový deskriptor alebo -1 (chyba)
  - pôvodný deskriptor zostáva ďalej počúvať

accept - príklad

```
int fd, newfd;
struct sockaddr_in me, peer;
socklen_t peerlen;

fd = socket(PF_INET, SOCK_STREAM, 0);

me.sin_family = AF_INET; me.sin_addr.s_addr =
INADDR_ANY;
me.sin_port = htons(cislo_portu);
bind(fd, &me, sizeof(me));
listen(fd, 5);

while (mam_bezat)
{
  peerlen = sizeof(peer);
  newfd = accept(fd, &peer, &peerlen);
  komunikuj(newfd, &peer);
}
close(fd);
```

connect

- `int  connect(int sockfd, const struct sockaddr *serv_addr, socklen_t addrlen)`
  - sys/types.h, sys/socket.h
  - pri SOCK_STREAM vytvorí spojenie na zadanú adresu
  - pri SOCK_DGRAM nastaví adresu druhej strany
  - vráti 0 (ok) alebo -1 (chyba)

```
struct sockaddr_in dst; int fd;
dst.sin_family = AF_INET;
dst.sin_port = htons(cislo_portu);
dst.sin_addr.s_addr = inet_addr("127.0.0.1");

fd=socket(PF_INET, SOCK_STREAM, 0);
connect(fd, &dst, sizeof(dst));
```

shutdown

- `int shutdown(int s, int how)`
  - sys/socket.h
  - požiada o ukončenie spojenia (pre SOCK_STREAM)
  - how
    - SHUT_RD – už nebudeme čítať
    - SHUT_WR – už nebudeme písať
    - SHUT_RDWR – oboje
  - zvyčajne sa použije SHUT_WR, a čaká sa na ukončenie z druhej strany (prečítanie 0 B) a následne sa zavolá close.

recv, recvfrom, read

- `ssize_t recv(int s, void *buf, size_t len, int flags)`
- `ssize_t  recvfrom(int s, void *buf, size_t len, int flags, struct sockaddr *from, socklen_t *fromlen)`
  - sys/types.h, sys/socket.h
  - čítajú z s max. len B do buf, vrátia počet prečítaných alebo -1 (0 znamená koniec spojenia)
  - recvfrom vracia aj informácie o adrese druhej strany (viď accept)

recv, recvfrom, read

- flags
  - MSG_PEEK
    - prečíta, ale aj ponechá na vstupe
  - MSG_DONTWAIT
    - nečaká na dáta
    - ak by inak čakal, vráti -1 a nastaví errno na EAGAIN
- read (viď vstup zo súborov)
  - dá sa použiť aj na socket, funguje analogicky ako recv s flags == 0

send, sendto, write

- `ssize_t send(int s, const void *msg, size_t len, int flags)`
- `ssize_t sendto(int s, const void *msg, size_t len, int flags, const struct sockaddr *to, socklen_t tolen)`
  - zapíše max. len B z msg do s
  - sendto pre SOCK_DGRAM obsahuje aj adresu druhej strany, send použije adresu nastavenú pomoco connect
  - vrátia počet zapísaných B alebo -1 (chyba)

send, sendto, write

- flags
  - MSG_DONTWAIT
    - nečaká na možnosť odoslať dáta
    - ak by musel čakať, vráti -1 a errno nastaví na EAGAIN
  - MSG_NOSIGNAL
    - zablokuje vznik signálu SIGPIPE
- write (viď výstup do súborov)
  - môže sa použiť aj na socket, funguje analogicky ako send s flags == 0
- Pri zápise môže vzniknúť signál SIGPIPE

Preklad z mien na IP adresy

- `int getaddrinfo(const char *node, const char *service, const struct addrinfo *hints, struct addrinfo **res)`
  - sys/types.h, sys/socket.h, netdb.h
  - node – meno počítača (alebo IP adresa ako text)
  - service – meno služby alebo číslo portu (ako text)
    - alebo NULL
  - hints – doplňujúce požiadavky
  - res – smerník na premennú pre uloženie spájaného zoznamu výsledkov
  - vracia 0, ak OK, chybový kód pri chybe

Preklad z mien na IP adresy

- ```
  struct addrinfo {
                  int              ai_flags;
                  int              ai_family;
                  int              ai_socktype;
                  int              ai_protocol;
                  socklen_t        ai_addrlen;
                  struct sockaddr *ai_addr;
                  char            *ai_canonname;
                  struct addrinfo *ai_next;
              };
  ```
  - používa sa aj na uloženie hintov – doplňujúcich požiadaviek na prevod, aj na jednotlivé položky v spájanom zozname výsledkov
  ```

Preklad z mien na IP adresy

- hints
  - ai_family – rodina protokolov (AF_INET, AF_INET6, AF_UNSPEC)
  - ai_socktype – typ socketu (SOCK_STREAM, SOCK_DGRAM)
  - ai_protocol – číslo protokolu (0)
  - ai_flags – špeciálne flagy
    - AI_CANONNAME – vyžiada kanonické meno
    - AI_PASSIVE – ak node ==NULL, použije INADDR_ANY
    - AI_NUMERICHOST – node musí obsahovať IP adresu
    - AI_V4MAPPED – používa sa pre IPv6 socket na IPv4 komunikáciu
  - všetko ostatné 0, resp. NULL

Preklad z mien na IP adresy

- ak vráti 0, tak *res bude ukazovať na spájaný zoznam (spájaný cez ai_next, ukončený NULL) vyplnených položiek typu struct addrinfo
  - ai_canonname – kanonické meno (v 1. položke)
  - ai_family, ai_socktype, ai_protocol – hodnoty vhodné pre socket(...)
  - ai_addrlen – dĺžka adresovacej štruktúry
  - ai_addr – smerník na adresovaciu štruktúru typu ako struct sockaddr_in (a pod.)

Preklad z mien na IP adresy

- `void freeaddrinfo(struct addrinfo *res)`
  - uvoľní pamäť alokovanú getaddrinfo
- `const char *gai_strerror(int errcode)`
  - vráti textový popis chybového kódu

Preklad z mien na IP adresy

```
struct addrinfo hints;
struct addrinfo * res = NULL;
struct addrinfo * p;

memset(&hints, 0, sizeof(hints));
hints.ai_family = AF_INET;
hints.ai_socktype = SOCK_STREAM;
hints.ai_protocol = 0;
hints.ai_flags = 0;

if((x=getaddrinfo("xy.sk", "80", &hints, &res)) == 0)
{
  for (p = res; p != NULL; p=p->ai_next)
  { ...
  }
  freeaddrinfo(res);
} else printf("chyba: %s\n", gai_strerror(x));
```

Práca s časom

- `time_t time(time_t *t)`
  - time.h
  - vráti aktuálny čas ako počet sekúnd od 0:00:00 UTC 1.1.1970
- `struct tm *gmtime(const time_t *timep)`
- `struct tm *localtime(const time_t *timep)`
  - vrátia smerník na struct tm, kde je rozpísaný zadaný čas ako UTC, resp. ako lokálny čas

Práca s časom

- `char *asctime(const struct tm *tm)`
- `char *ctime(const time_t *timep)`
  - vrátia textový reťazec popisujúci zadaný čas (v prípade ctime chápaný ako lokálny čas)
- `time_t mktime(struct tm *tm)`
  - vráti zadaný rozpísaný čas (chápaný ako lokálny čas) v tvare „počet sekúnd od 1.1.1970“

Práca s časom

```
struct tm {
    int     tm_sec;         /* seconds */
    int     tm_min;         /* minutes */
    int     tm_hour;        /* hours */
    int     tm_mday;        /* day of the month */
    int     tm_mon;         /* month */
    int     tm_year;        /* year */
    int     tm_wday;        /* day of the week */
    int     tm_yday;        /* day in the year */
    int     tm_isdst;       /* daylight saving time */
};
```

Spustenie programu

- `int execl(const char *path, const char *arg, ...)`
- `int execlp(const char *file, const char *arg, ...)`
  - unistd.h
  - nahradí proces novým procesom zo súboru path resp. file a odovzdá mu argumenty
  - prvý (povinný) argument je „meno programu“
  - posledný argument je NULL

História IA32

procesor	registre	dátová zbernica	adresová zbernica	max. veľkosť pamäte	max. veľkosť segmentu	poznámka
8086	16 bit	16 bit	20 bit	1 MB	64 KB	
80286	16 bit	16 bit	24 bit	16 MB	64 KB	protected mode
80386	32 bit	32 bit	32 bit	4 GB	4 GB	V86, stránkovanie
80486	32 bit	32 bit	32 bit	4 GB	4 GB	pipelining, on chip 8KB L1 cache
Pentium	32 bit	64 bit	32 bit	4 GB	4 GB	2 pipelines, 8+8 KB cache, MMX
Pentium Pro/II/III (P6)	32 bit	64 bit	36 bit	64 GB	4 GB	3 ins/clock, L2 cache, SSE
Pentium 4	32 bit	64 bit	36 bit	64 GB	4 GB	SSE2, ...

Register EFLAGS

- CF(carry, bit 0) – prenos z najvyššieho bitu
- PF(parity, bit 2) – indikuje párny počet 1 v najnižšom byte
- AF (aux. carry, bit 4) – prenos z bitu 3
- ZF (zero, bit 6) – indikuje nulový výsledok
- SF (sign, bit 7) – indikuje znamienko – najvyšší bit výsledku
- TF (trace, bit 8) – po každej inštrukcii generuje prerušenie 1
- IF (interrupt, bit 9) – ak je 0, sú zakázané ext. prerušenia
- DF (direction, bit 10) – smer reťazcových inštr. 0+, 1-
- OF (overflow, bit 11) – indikuje pretečenie pri znam. oper.
- bit 1 = 1, bity 3,5,15,22-31 = 0, rezervované

Register EFLAGS

- IOPL (bit 12,13) – potrebná úroveň ochrany na I/O inštrukcie
- NT (nested task, bit 14) – používané pri prepínaní procesov
- RF (resume, bit 16) – blokuje ladiace prerušenia
- VM (V86 mode, bit 17) – zapína V86 mód
- AC (alignment check, bit 18) – zapína kontrolu zarovnania
- VIF (virtual IF, bit 19), VIP (virtual int. pending, bit 20) – slúžia pre virtuálne prerušenia
- ID (bit 21) – ak sa dá meniť, procesor podporuje inštrukciu CPUID

Všeobecné registre

- eax – akumulátor, operand, výsledok
- ebx – adresa dát v segmente DS
- ecx – počítadlo pre cyklické operácie
- edx – adresovanie I/O operácií
- esi – adresa zdrojového reťazca
- edi – adresa cieľového reťazca
- esp – stack pointer (v segmente SS)
- ebp – adresa dát na zásobníku (v seg. SS)

Segmentové registre

- 16 bitové
- CS – segment pre kód
- DS – segment pre dáta
- SS – segment pre zásobník
- ES – pomocný segment pre dáta, cieľový segment pre reťazcové inštrukcie
- FS, GS – ďalšie pomocné segmenty pre dáta (od 80386)

Segmentovaný pamäťový model

- lineárny adresný priestor rozdelený na segmenty
  - bázová adresa, veľkosť, typ
- adresa
  - segment (určený segmentovým reg.)
    - default možno zmeniť prefixom cs:,ds:,es:,ss:,fs:,gs:
  - offset 32 bit (16 bit v 16 bit móde)
  - lienárna adresa = bázová adresa seg. + offset

Segmentovaný pamäťový model

- inštrukcia jmp má 2 podoby
  - near jump
    - cieľová adresa 32 bit (16 bit v 16 bit móde)
    - nemení hodnotu CS, len EIP
  - far jump
    - cieľová adresa 48 bit (32 bit v 16 bit móde)
    - mení CS aj EIP
    - ljmp $segment, $offset
    - ljmp **nepriamy_operand* – najprv offset, vyššie seg.

Segmentovaný pamäťový model

- inštrukcia call má 2 podoby
  - near call
    - uloží EIP, mení EIP
  - far call
    - uloží CS, potom EIP, mení CS aj EIP
    - lcall ...
- inštrukcia ret
  - near return – obnoví len EIP
  - far return – obnoví EIP aj CS
    - lret

Flat pamäťový model

- vytvoria sa 2 segmenty, ktoré sa prekrývajú
  - majú rovnakú bázovú adresu a limit
  - jeden má typ „kód“, druhý „dáta“
  - CS ukazuje na kódový segment
  - DS=ES=SS=FS=GS ukazujú na dátový segment
- používajú sa len near jump, call a ret
- chránený flat model
  - kódový a dátový segment sú disjunktné
  - nemožno zamieňať „kódové“ a „dátové“ adresy

Módy procesora

- Real mode
  - kompatibilný s 8086
- Protected mode (od 80286)
  - zavádza ochranu prístupu k pamäti, I/O portom, inštrukciám s globálnym dopadom
  - umožňuje adresovať všetku pamäť
- Virtual 86 mode (od 80386)
  - umožňuje vykonávať program pre reálny mód ako úlohu v systéme používajúcom chránený mód

Real Mode

- default po resete procesora
- segmenty majú veľkosť 64KB
- bázová adresa segmentu je obsah príslušného segmentového registra vynásobený 16
- offset je 16 bitový
- základná veľkosť operandov je 16 bitov
- segmenty sa prekrývajú
  - 0x0000:0x1111 = 0x0100:0x0111 = 0x0111:0x0001

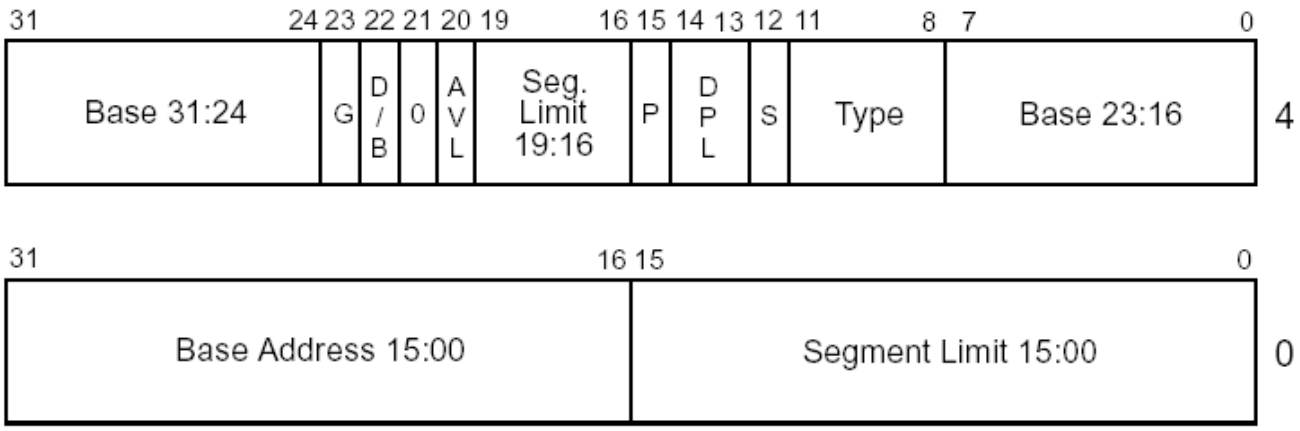
Protected Mode (chránený mód)

- bázová adresa, veľkosť a typ segmentu je určený v popisovači segmentu (segment descriptor)
- offset môže byť 16 alebo 32 bitový
- základná veľkosť operandu je 16 alebo 32 bitov
- popisovače sú v tabuľkách
  - GDT (Global Descriptor Table)
  - LDT (Local Descriptor Table)
- segmentový register
  - viditeľná časť (16 bitov) – selector
  - neviditeľná časť – kópia údajov z popisovača

Protected Mode

- Selektor
  - bity 0,1 – RPL(requested privilege level)
  - bit 2
    - 0 – popisovač je v GDT
    - 1 – popisovač je v LDT
  - bity 3 – 15
    - index do tabuľky popisovačov
  - neviditeľná časť sa naplní vtedy, keď sa naplní selektor

Segment Descriptor



AVL — Available for use by system software
BASE — Segment base address
D/B — Default operation size (0 = 16-bit segment; 1 = 32-bit segment)
DPL — Descriptor privilege level
G — Granularity
LIMIT — Segment Limit
P — Segment present
S — Descriptor type (0 = system; 1 = code or data)
TYPE — Segment type

Segment Descriptor

- bázová adresa (bity 16-39, 56-63)
  - lineárna 32 bitová bázová adresa segmentu
- limit (bity 0-15, 48-51)
  - maximálna hodnota offsetu
    - v bytoch ak G (bit 55) = 0
    - v 4KB jednotkách ak G = 1
- S (bit 44)
  - 0 = systémový segment, 1 = kód/dáta
- Typ (bity 40-43)

Segment Descriptor

- DPL (bity 45-46)
  - descriptor privilege level
- P (bit 47)
  - segment present
    - 1 – segment existuje, údaje sú platné
    - 0 – segment neexistuje, bity 0-39, 48-63 sú voľne k dispozícii
- AVL (bit 52)
  - voľne k dispozícii

Segment Descriptor

- D/B (bit 54)
  - pre kódový segment (D – default data/addr size)
    - 0 – offset a základná veľkosť operandu je 16 bitov
    - 1 – offset a základná veľkosť operandu je 32 bitov
  - pre zásobník (B – big)
    - 0 – zásobník je adresovaný reg. SP (16 bit)
    - 1 – zásobník je adresovaný reg. ESP (32 bit)
  - pre expand-down dátový segment (B – big)
    - 0 – max. offset = 0xFFFF
    - 1 – max. offset = 0xFFFFFFFF

Segment Descriptor

- Typ segmentu
  - 0EWA – dátový segment
    - E – expand up(0) / down (1)
    - W – writeable (zapisovateľný) (1) / read-only (0)
  - 1CRA – kódový segment
    - C – conforming
    - R – readable (čitateľný) (1) / execute-only (0)
  - A – accessed
    - procesor nastaví pri uložení hodnoty do seg. registra

Segment Descriptor

- Systémový segment
  - 1 – 16 bit TSS
  - 2 – LDT
  - 3 – 16 bit TSS (busy)
  - 4 – 16 bit call gate
  - 5 – task gate
  - 6 – 16 bit interrupt gate
  - 7 – 16 bit trap gate
  - 9 – 32 bit TSS
  - 11 – 32 bit TSS (busy)
  - 12 – 32 bit call gate
  - 14 – 32 bit interrupt gate
  - 15 – 32 bit trap gate
  - 0, 8, 10, 13 - rezervované

Register GDTR

- 48 bitov
- obsahuje bázovú adresu a limit GDT
  - bity 0-15 – limit
  - bity 16-47 – lineárna 32 bitová adresa
  - uloženie hodnoty do GDTR
    - lgdt *nepriamy_operand*
  - uloženie hodnoty z GDTR do pamäte
    - sgdt *nepriamy_operand*

Register LDTR

- obsahuje bázovú adresu a limit LDT
- viditeľná časť
  - 16 bitov
  - selektor do GDT
    - popisovač systémového segmentu typu LDT (2)
  - uloženie hodnoty do/z LDTR
    - lldt *16bit-register_alebo_nepriamy_operand*
    - sldt *16bit-register_alebo_nepriamy_operand*
- neviditeľná časť obsahuje údaje z popisovača

Register CR0

- manipulácia špeciálnou verziou mov
- 32 bitov
  - PE(0) – zapína protected mode
  - MP(1), EM(2), TS(3), ET(4), NE(5) – ovplyvňujú spracovanie a umožňujú emuláciu FPU inštrukcií
  - WP(16) – ovplyvňuje page write-protection
  - AM(18) – alignment mask – povoľuje kontrolu zarovnania
  - PG(31) – paging enable – zapína stránkovanie

Ochrany v Protected Mode

- kontrola limitov
  - nemožno pristupovať mimo limit segmentu
- kontrola typu
  - zapisovať možno len do zapisovateľných dátových segmentov
  - čítať možno len dátové segmenty a čitateľné kódové segmenty
  - vykonávať možno len inštrukcie v kódových segmentoch

Ochrany v Protected Mode

- 4 úrovne privilégií
  - 0 najvyššia – jadro OS
  - 3 najnižšia – bežné aplikácie
  - aktuálna = CPL = CS.RPL
- privilegované inštrukcie
  - t.j. inštrukcie s globálnym dopadom
  - len na úrovni 0
- I/O inštrukcie, manipulácia s IF
  - len na úrovni IOPL a vyššej

Prístup k dátam

- DPL >= CPL AND DPL >= RPL
  - kontroluje sa pri plnení segmentového reg.
  - môžeme pristupovať len k dátam nižšej úrovne
  - RPL môžeme použiť na zníženie svojej úrovne oprávnení
- pri plnení SS musí platiť
  - DPL = RPL = CPL
- nesplnenie má za následok výnimku #GP

Predávanie riadenia

- jmp a call môžu použiť selektor
  - kódového segmentu
  - call gate, ktorá obsahuje selektor kódového segmentu
  - TSS, ktorý obsahuje selektor kódového segmentu
  - task gate, ktorá ukazuje na TSS, ktorý obsahuje selektor kódového segmentu

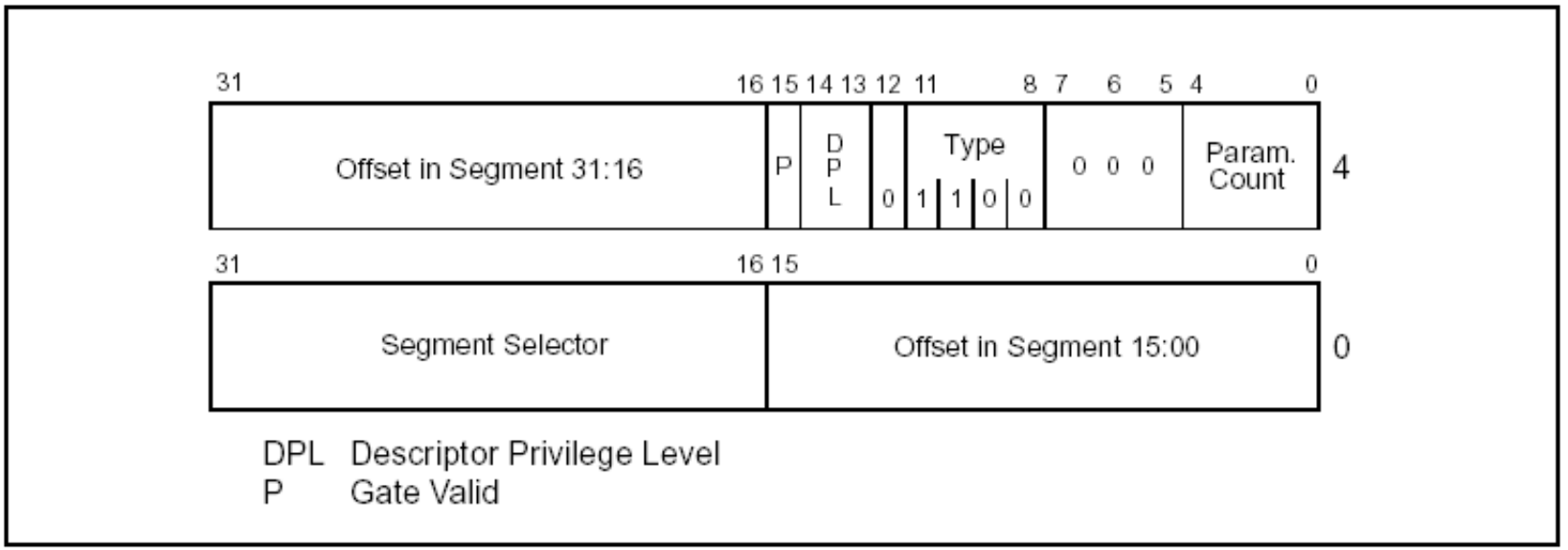
Predávanie riadenia – priamo

- ak je cieľový segment „nonconforming“
  - t.j. C = 0
  - DPL = CPL AND RPL <= CPL
  - CPL sa nezmení
  - možno predávať riadenie len do segmentu rovnakej úrove

Predávanie riadenia – priamo

- ak je cieľový segment „conforming“
  - t.j. C = 1
  - DPL <= CPL
  - RPL sa ignoruje
  - možno predať riadenie len do segmentu rovnakej alebo vyššej úrovne
  - CPL sa nezmení

Predávanie riadenia cez bránu



- typ 4 (16 bit), 12 (32 bit)
- selektor cieľového kódového segmentu
- offset v cieľovom segmente
- počet 16/32 bitových parametrov

Predávanie riadenia cez bránu

- CPL <= DPL brány, RPL <= DPL brány
- call
  - DPL cieľového kódového segmentu <= CPL
  - ak conforming, CPL sa nezmení
  - ak nonconforming, CPL := DPL cieľového seg.
- jmp
  - ak conforming, DPL cieľového seg. <= CPL
  - ak nonconforming, DPL cieľového seg. = CPL
  - CPL sa nezmení

Zásobníky pre rôzne úrovne

- pri zmene CPL sa mení aj použitý zásobník
  - pre bežiacu úlohu sú (v TSS) definované hodnoty SS a ESP pre úrovne 0, 1, 2
  - ak sa mení CPL, tak dôjde aj k zmene SS a ESP
  - na novom zásobníku budú:
    - SS, ESP
    - skopírovaných n parametrov z volajúceho zásobníka
    - CS, EIP

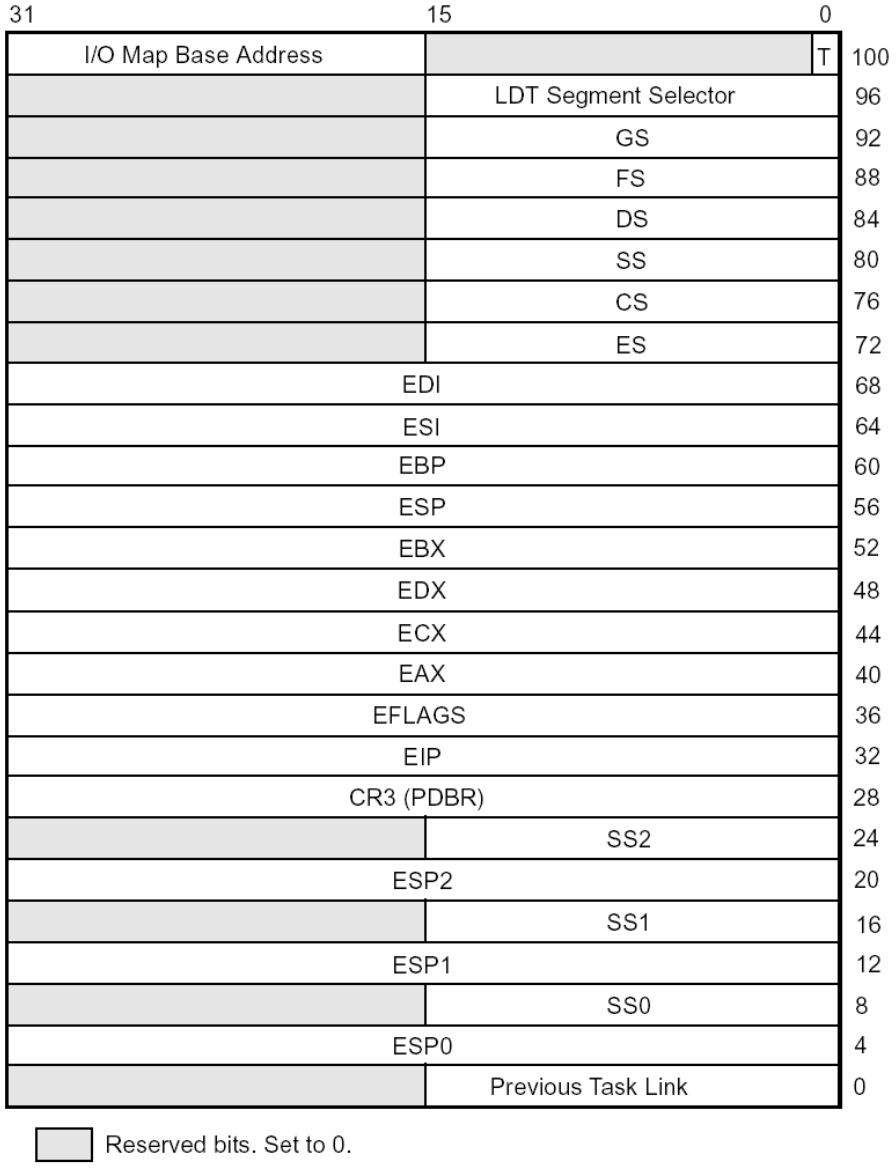
Predanie riadenia - ret

- RPL = CPL
  - nedochádza k zmene úrovne
- RPL > CPL
  - zo zásobníka sa získa pôvodné SS, ESP
  - ak ret n, n bytov sa vyberá z oboch zásobníkov
  - kontroluje sa, či „sedia“ hodnoty:
    - DPL, RPL (t.j. nové CPL) a C pre kódový segment
    - DPL a RPL pre nový zásobník s novým CPL
  - nulujú sa ostatné seg. registre, ak odkazujú na neprípustný segment

Správa úloh (tasks)

- stav úlohy
  - obsah segmentových registrov
  - obsah všeobecných registrov
  - obsah EFLAGS
  - obsah EIP
  - obsah LDTR, TR, CR3
  - I/O mapa
  - stack pointers na zásobníky pre úrovne 0, 1, 2
  - odkaz na predchádzajúcu úlohu

Task State Segment (TSS)



- systémový segment
- popisovač v GDT
- obsahuje stav úlohy
- dynamické položky
  - všeobecné registre
  - segmentové registre
  - EFLAGS
  - EIP
  - Previous Task Link

Task State Segment (TSS)

- statické položky (procesor len číta)
  - LDT selektor – obsahuje hodnotu pre LDTR
  - CR3
  - SSx, ESPx – zásobníky pre vyššie úrovne
  - T – ak 1, pri prepnutí na túto úlohu sa generuje ladiace prerušenie
  - I/O mapa – bitová mapa pre I/O porty
    - 1=zakázaný, 0=povolený
    - nasledovaná (na adrese limit): bytom 0xff

Task Register (TR)

- identifikuje aktuálnu úlohu
- viditeľná časť
  - 16 bit selektor do GDT pre TSS
  - načítanie hodnoty do TR
    - ltr *reg16/mem16*
  - uloženie hodnoty z TR
    - str *reg16/mem16*
- neviditeľná časť
  - kópia popisovača

Prerušenia a výnimky

- 0 – 255
- prerušenia
  - externé – od hardvéru
    - maskovateľné – sú obslúžené, ak IF=1
    - nemaskovateľné (NMI)
  - softvérové – inštrukciou int n
- výnimky
  - chybové stavy
  - softvérové (int 3, into, bound)
  - machine check exceptions – indikácia HW chýb

Výnimky

- fault
  - po návrate sa znovu vykoná inštrukcia, ktorá výnimku spôsobila
- trap
  - po návrate sa vykoná inštrukcia, ktorá nasleduje za inštrukciou, ktorá spôsobila výnimku
- abort
  - návrat nie je možný
  - závažné, neodstrániteľné chyby

Výnimky

číslo	označnie	názov	typ	chybový kód	zdroj
0	#DE	Divide Error	Fault	nie	div, idiv
1	#DB	Debug	Fault/Trap	nie	int 1, prístup k dátam/kódu
2		NMI	Interrupt	nie	nemaskovateľné prerušenie
3	#BP	Breakpoint	Trap	nie	int 3
4	#OF	Overflow	Trap	nie	into
5	#BR	Bound Range Exceeded	Fault	nie	bound
6	#UD	Undefined Opcode	Fault	nie	nedefinovaná inštrukcia
7	#NM	Device Not Available	Fault	nie	floating point inštrukcia
8	#DF	Double Fault	Abort	0	dvojitá výnimka
9		Coprocessor Segment Overrun	Fault	nie	floating point
10	#TS	Invalid TSS	Fault	áno	prepnutie procesu
11	#NP	Segment Not Present	Fault	áno	plnenie seg. reg.
12	#SS	Stack Segment Fault	Fault	áno	plnenie SS, zásobníkové op.
13	#GP	General Protection	Fault	áno	chyba pri kontrole ochrany
14	#PF	Page Fault	Fault	áno	výpadok stránky
16	#MF	Math Fault	Fault	nie	chyba pri floating point
17	#AC	Alignment Check	Fault	0	kontrola zarovnania
18	#MC	Machine Check	Abort	nie	
19	#XF	SIMD Floating Point	Fault	nie	SSE, SSE2
15,20-31		rezervované			

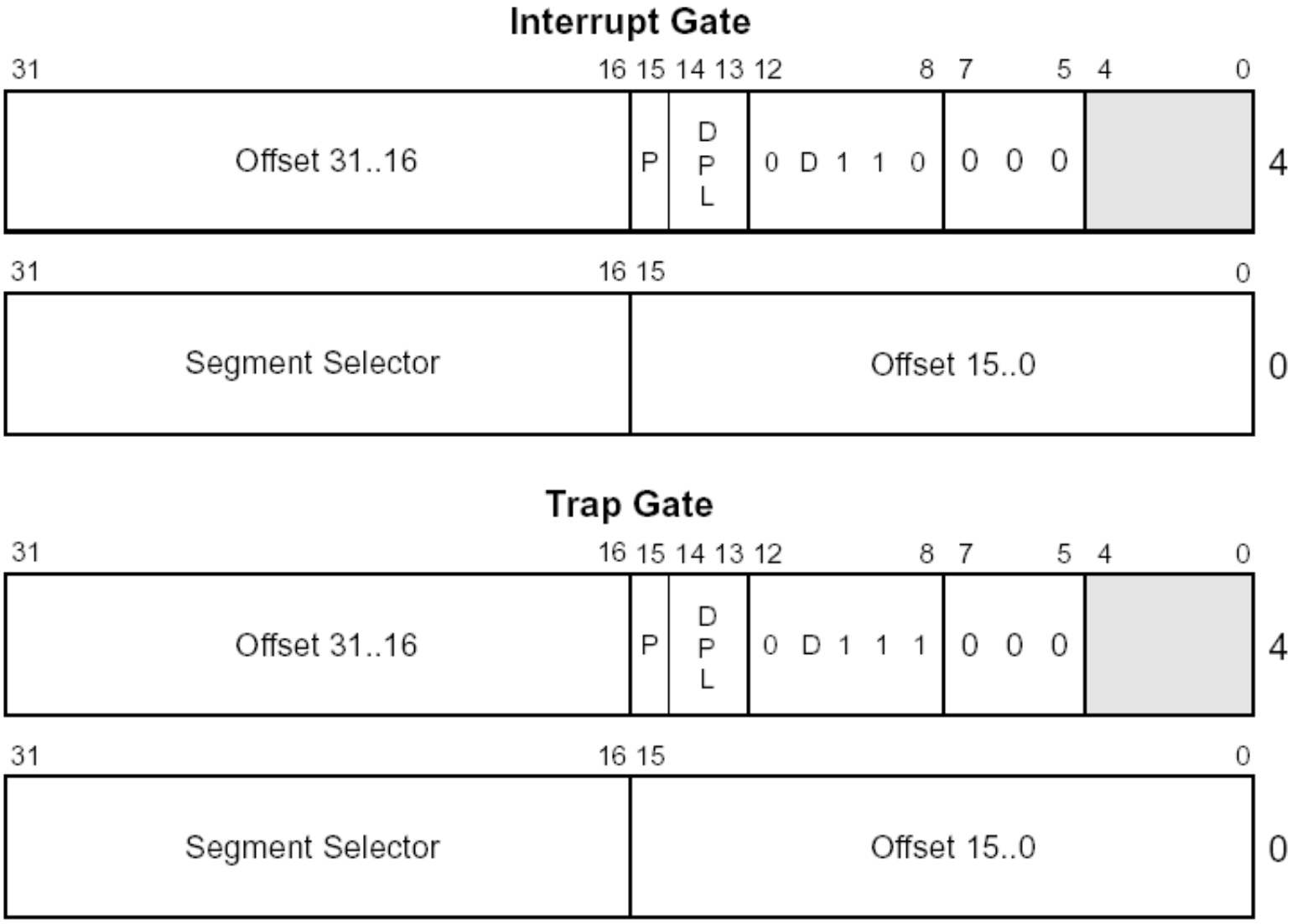
Interrupt Descriptor Table (IDT)

- priraďuje jednotlivým prerušeniam a výnimkám obslužné funkcie
- štruktúra ako GDT
- bázová adresa a limit uložené v 48 bitovom registri IDTR
  - bity 0-15 limit, bity 16-47 bázová adresa
  - lidt *48-bit_nepriamy_operand*
  - sidt *48-bit_nepriamy_operand*

Interrupt Descriptor Table (IDT)

- IDT môže obsahovať
  - task gate
    - prerušenie spôsobí prepnutie procesu ako pri call
  - interrupt gate a trap gate
    - ako call do iného segmentu
    - navyše ukladá EFLAGS
      - ak nedochádza k zmene CPL: EFLAGS, CS, EIP
      - ak dochádza k zmene CPL: SS, ESP, EFLAGS, CS, EIP
    - vynuluje TF, VM, NT
    - pri interrupt gate vynuluje aj IF
    - návrat inštrukciou iret

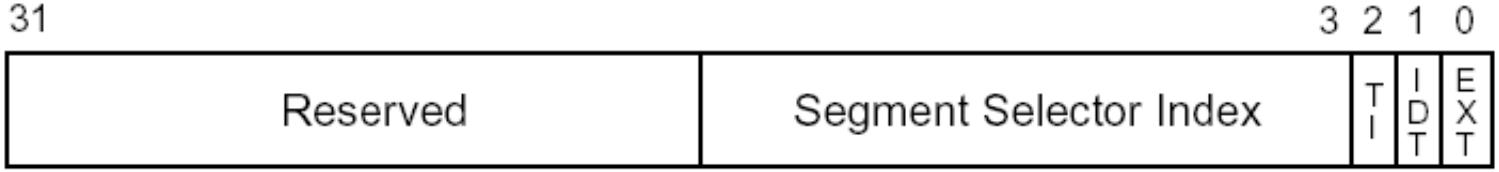
Interrupt Descriptor Table (IDT)



Kontroly prístupu pri prerušeniach

- prístup k bráne
  - kontrolovaný pri softvérových prerušeniach
  - CPL <= DPL brány
- prístup ku kódovému segmentu
  - kontrolovaný vždy pri interrupt a trap gate
  - CPL >= DPL
  - pre výnimky a HW prerušenia
    - zvyčajne DPL=0, aby ich bolo možné obslúžiť vždy

Chybový kód



- niektoré výnimky generujú chybový kód
  - EXT – zdroj výnimky je externý
  - IDT – selektor ukazuje do IDT(1), GDT/LDT(0)
  - TI – selektor ukazuje do LDT(1), GDT(0)
- uloží sa na zásobník po EIP
  - treba odstrániť pred návratom

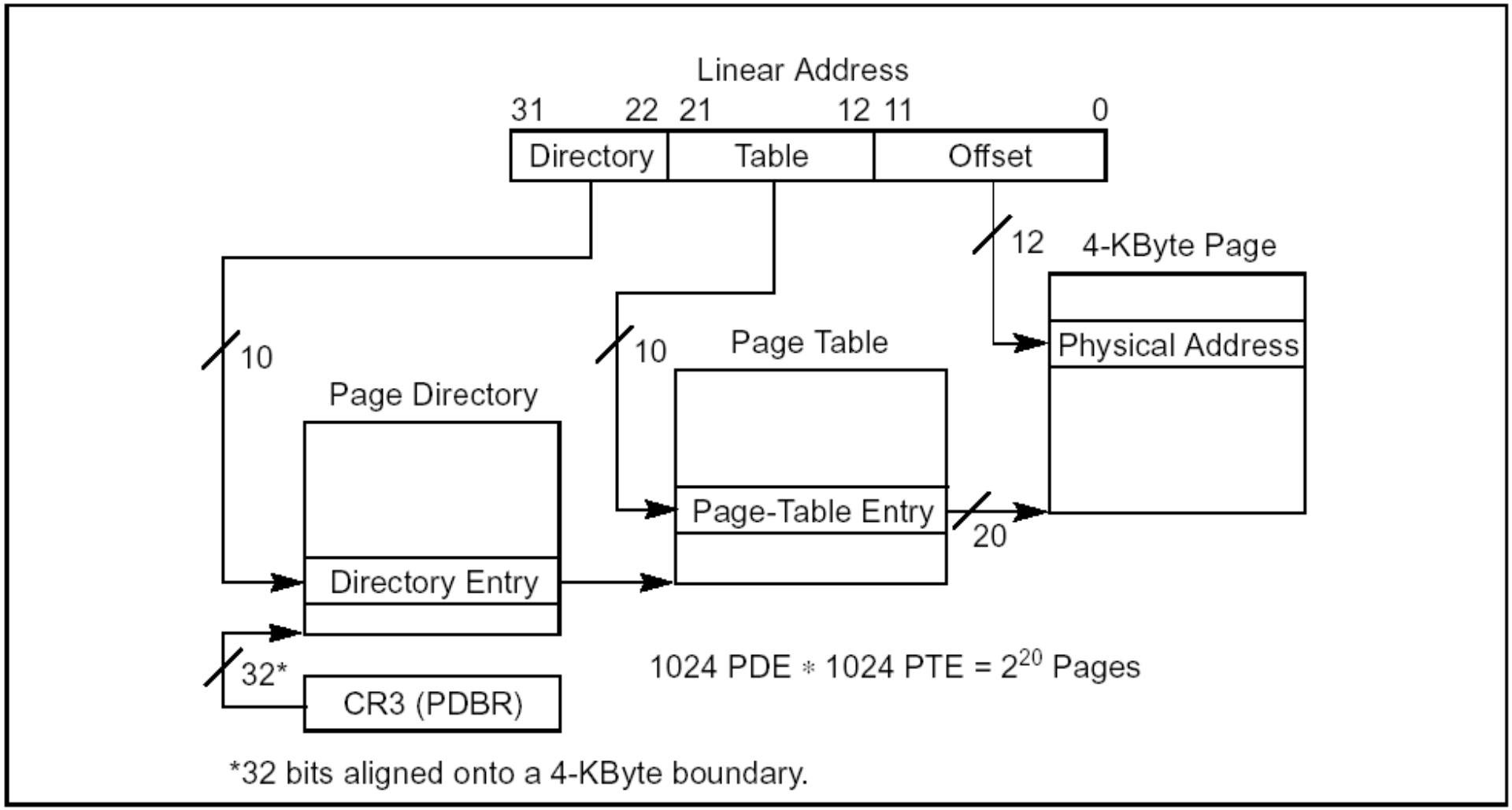
Prerušenia v reálnom móde

- používa sa Interrupt Vector Table
  - každá položka obsahuje 4 bytovú adresu
    - nižších 16 bitov offset
    - vyšších 16 bitov segment
  - do zásobníka sa ukladá FLAGS (16 bit), CS, IP
  - vynuluje sa IF, TF
  - návrat inštrukciou iret
  - na 8086 je IVT na adrese 0 a limit je 0x3FF
  - bázová adresa a limit sú určené v IDTR

Stránkovanie

- umožňuje transformovať lineárnu adresu získanú po segmentovaní na fyzickú adresu
- adresný priestor je rozdelený na stránky
  - veľkosť stránky je 4 KB
  - na novších procesoroch môže byť aj 4 alebo 2 MB
- stránkovanie nie je možné v reálnom móde
- zapína sa nastavením bitu PG(31) v CR0
- register CR3 obsahuje fyzickú adresu adresára stránok (page directory)

Stránkovanie



Stránkovanie

- adresár stránok a tabuľka stránok
  - 4 KB = 1024 položiek, 32 bitov na položku
  - bity 31 – 12: fyzická adresa tabuľky stránok alebo rámca
  - bity 11 – 9: voľné
  - bit 6: „dirty“
    - procesor nastaví na 1 pri zmene obsahu stránky
    - len v tabuľke stránok
  - bit 5: „accessed“
    - procesor nastaví na 1 pri prístupe k stránke/tabuľke

Stránkovanie

- bit 2: „user“
  - 1 = user page
    - proces na CPL=3 môže stránku použiť
  - 0 = supervisor page
    - proces na CPL=3 nemôže stránku použiť
- bit 1: „read/write“
  - 0 = read only
  - 1 = read/write
- bit 0: „present“
  - 1 = obsah položky je platný
  - 0 = prístup vyvolá výnimku #PF, bity 1-31 sú voľné

Ochrana stránok

- kombinácia user (U) a read/write(W)
  - U = PD.U & PT.U
  - W= PD.W & PT.W
  - CR0.WP = 1
    - ani proces s CPL < 3 nemôže zapisovať do read-only
  - CR0.WP = 0
    - obmedzenie zápisu sa nezťahuje na proces s CPL < 3

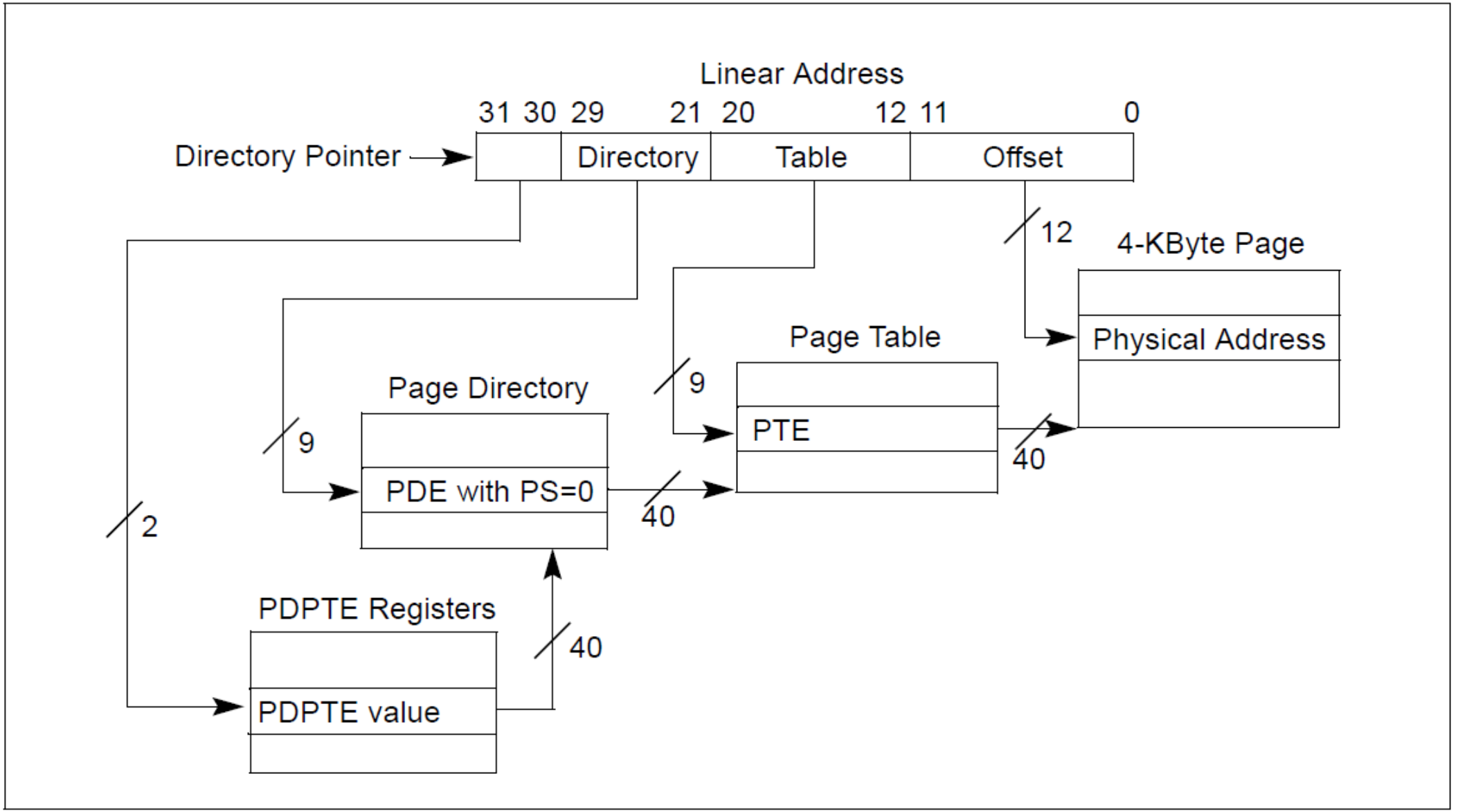
Výpadok stránky (#PF, 14)

- chybové slovo
  - bit 0: P
    - 0 = výpadok stránky z dôvodu neplatnej stránky
    - 1 = výnimka z dôvodu porušenia ochrany
  - bit 1: W
    - 0 – operácia čítania, 1 – operácia zápisu
  - bit 2: U
    - 1 – proces mal CPL=3
- register CR2
  - lineárna adresa, ktorá spôsobila výpadok

PAE stránkovanie

- zapína sa v CR4
- CR3 ukazuje na Page Directory Pointer Table (PDPT)
  - 4 položky po 64 bitoch
  - nahrávajú sa do špeciálnych PDPTE registrov
  - každá riadi stránkovanie pre 1GB lineárneho adresného priestoru
  - fyzické adresy môžu byť až 52-bitové

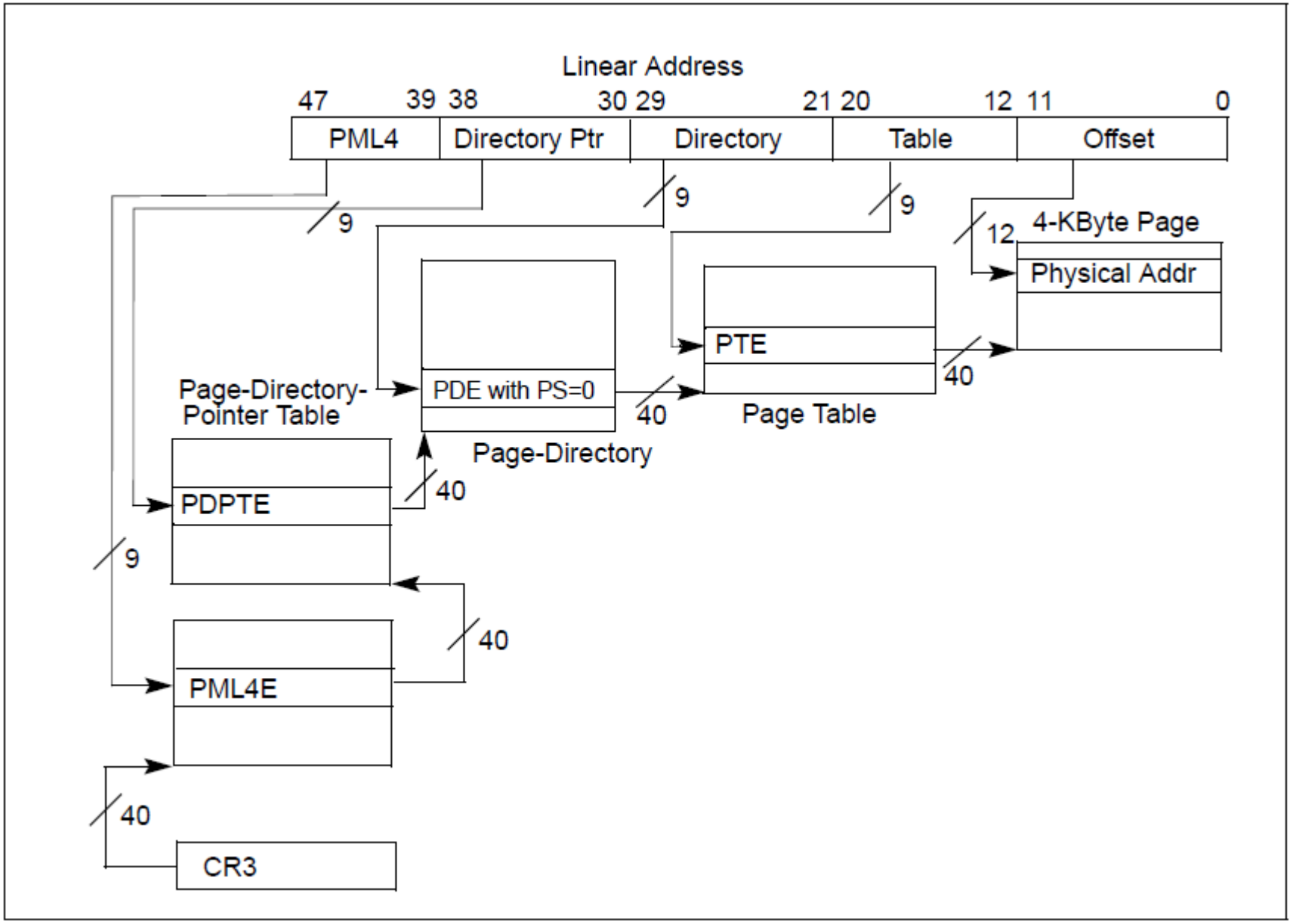
PAE stránkovanie



4-úrovňové stránkovanie

- používa sa v 64-bit móde
- transformuje 48-bitové lineárne adresy na 52-bitové fyzické adresy
- CR3 obsahuje adresu PML4 tabuľky
- PML4 obsahuje 512 64-bit položiek s adresou PDPT
- PDPT obsahuje 512 64-bit položiek s adresou adresára stránok (PD)
- PD obsahuje 512 64-bit položiek s adresou PT
- PT obsahuje 512 64-bit položiek so 40 bitmi adresy stránky

4-úrovňové stránkovanie



Virtual 86 Mode

- umožňuje vykonávanie kódu určeného pre reálny mód ako úlohu v chránenom móde
- ak má byť takých úloh viac, je nutné stránkovanie
- výpočet adries je ako v reálnom móde
- zapína sa nastavením VM v EFLAGS
  - prepnutím na úlohu, ktorá má VM=1
  - iret z 32-bitového segmentu s CPL=0
- vypne sa prerušením

Virtual 86 Mode

- V86 proces je na úrovni CPL=3
- inštrukcie povolené, ak IOPL=3
  - int, sti, cli, popf, pushf, iret
- I/O inštrukcie
  - sú povolené na základe I/O mapy v TSS
  - nezávisia od IOPL
- nepovolené inštrukcie generujú #GP (13)
  - je možné ich emulovať

Prerušenia vo V86

- výnimky a externé prerušenia
  - trap gate alebo interrupt gate
    - musí ukazovať na DPL=0, C=0 segment
    - uloží na zásobník:
      - GS, FS, DS, ES, SS, ESP, EFLAGS, CS, EIP, [chybový kód]
    - vynuluje DS, ES, FS, GS, vynuluje VM
    - pri iret dôjde k obnove registrov zo zásobníka
  - task gate
    - dôjde k prepnutiu úlohy
    - pri iret dôjde k prepnutiu naspäť

Prerušenia vo V86

- softvérové prerušenia (int n)
  - ak je IOPL=3
    - spracujú sa rovnako ako výnimky a externé prerušenia
    - DPL brány musí byť 3
  - ak je IOPL<3
    - generujú #GP

64-bitové rozšírenia (IA-32e)

- tzv. IA-32e mód
- po resete sú 64-bit rozšírenia vypnuté
  - procesor funguje ako 32-bit
- po zapnutí IA-32e môže procesor pracovať v 2 módoch:
  - compatibility mode
  - 64-bit mode
- v IA-32e móde nie je podporovaný V86

Compatibility mode

- slúži na beh 16 a 32-bitových aplikácií v chránenom móde
- z pohľadu aplikácie sa neodlišuje od chráneného módu
  - segmentácia funguje rovnako ako v 32-bit móde
- OS musí byť 64-bitový

64-bit mode

- zväčšuje registre na 64 bitov
- sprístupňuje ďalších 8 všeob. registrov (spolu 16)
  - RAX, RBX, RCX, RDX, RSI, RDI, RBP, RSP, R8 – R15
- adresy majú dĺžku 64 bitov
  - priama adresa a doplnok len 32 bitov
  - 64-bit priama adresa sa dá použiť len s mov z/do AL/AX/EAX/RAX
- operandy môžu mať 8, 16, 32, 64 bitov
  - priama 64-bit hodnota len pre mov do registra
- zavádza relatívny adresný mód (doplnok (%rip))

64-bit mode

- v podstate ruší segmentáciu
  - nekontroluje limity
  - bázová adresa = 0
    - => jedine flat pamäťový model
    - výnimka FS, GS
  - zostáva kontrola typov segmentov a úrovní oprávnení
    - avšak jej dopad je limitovaný ako dôsledok vyššie uvedených

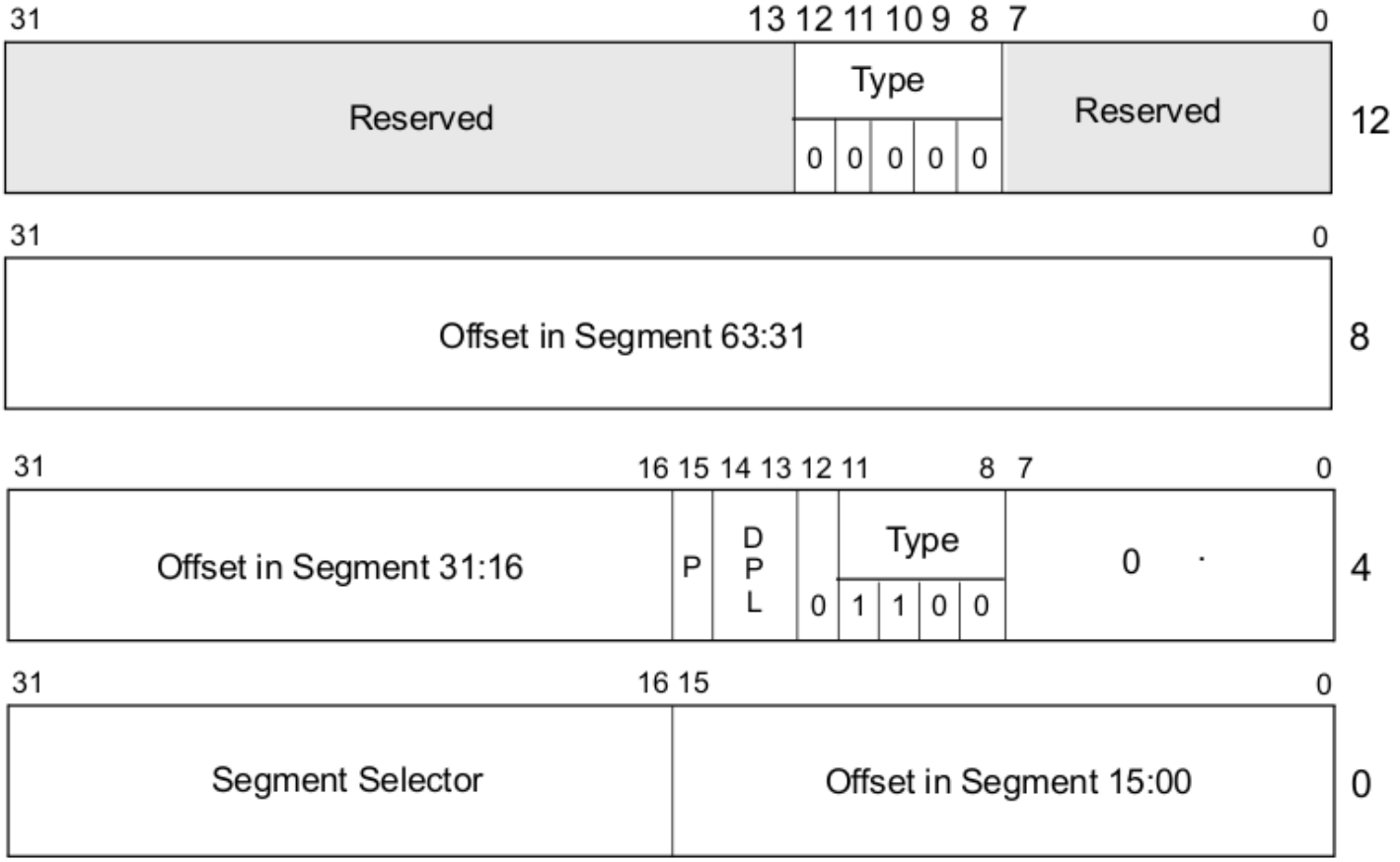
64-bit mode

- lineárne adresy musia byť v **kanonickom** tvare
  - bity 63 až (M-1) musia byť rovnaké
  - M je max. podporovaná dĺžka lineárnej adresy
    - typicky 48
- často sa adresný priestor delí na 2 časti
  - 0 – 00007FFF FFFFFFFF
    - t.j. bit 47 = 0, kladné adresy, pre aplikácie, 128 TB
  - FFFF8000 00000000 – FFFFFFFF FFFFFFFF
    - t.j. bit 47 = 1, záporné adresy, pre OS, 128 TB

IA-32e

- menia sa niektoré typy popisovačov
  - call gate
  - interrupt gate, trap gate
  - LDT
  - TSS
- rozširujú sa, aby v nich mohla byť 64 bitová adresa
- nový bit v popisovači kódového segmentu (L)
  - L=1: default veľkosť operandu 32-bit, adresy 64-bit

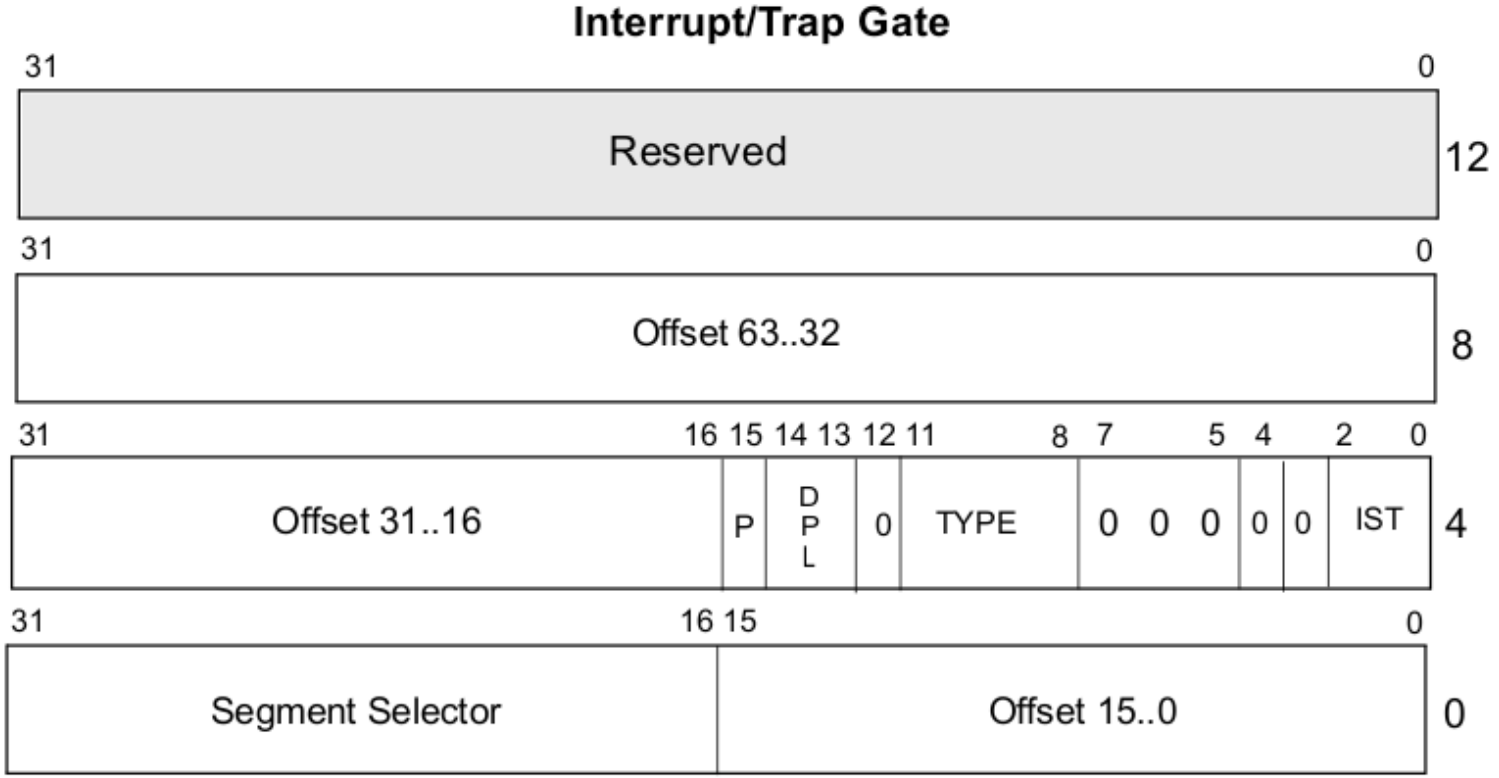
IA-32e call gate



IA-32e predávanie riadenia

- call gate musí ukazovať na 64-bit (L=1) kódový segment
- položky na zásobníku sú 64-bitové
- neexistuje podpora pre kopírovanie argumentov pri prepnutí zásobníka
- pri prepnutí zásobníka sa SS nastaví na 0 (+CPL), nahrá sa len nový RSP z TSS
  - bázová adresa a limit sa v 64-bit móde aj tak ignorujú

Prerušenia v IA-32e

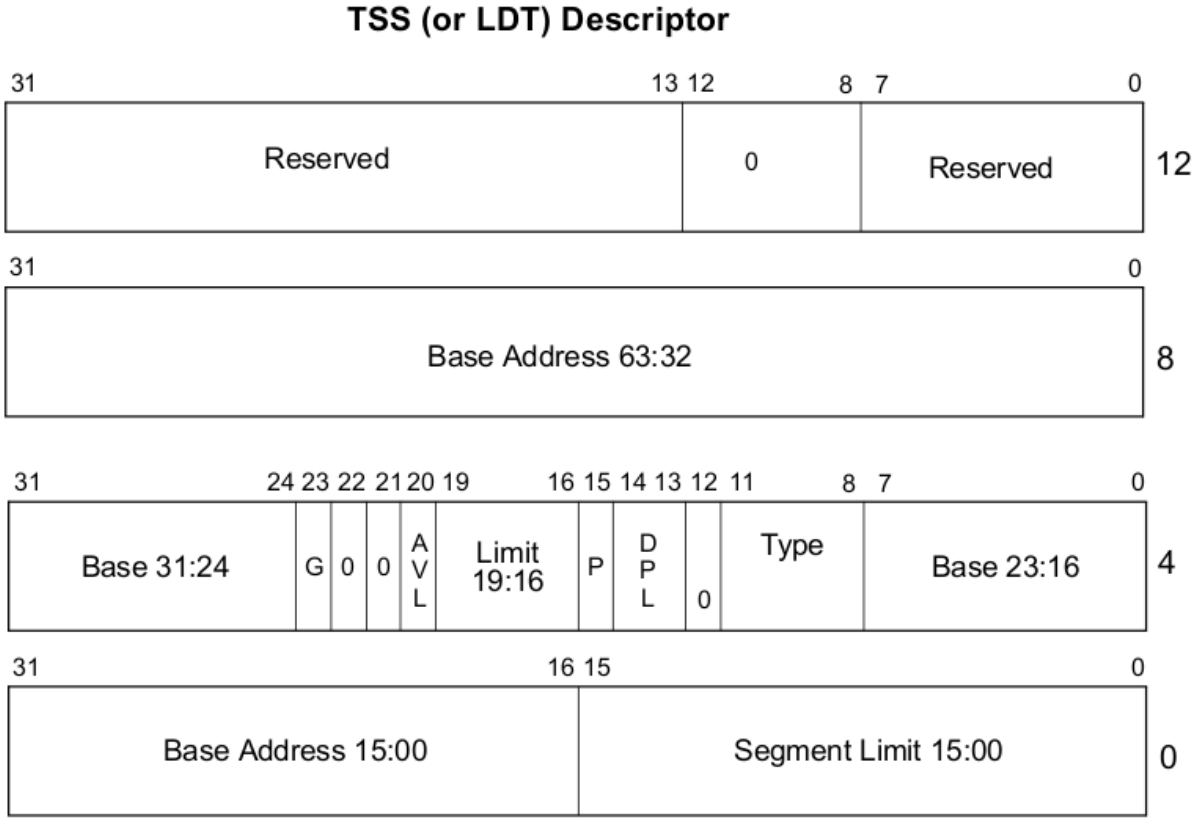


DPL	Descriptor Privilege Level
Offset	Offset to procedure entry point
P	Segment Present flag
Selector	Segment Selector for destination code segment
IST	Interrupt Stack Table

Prerušenia v IA-32e

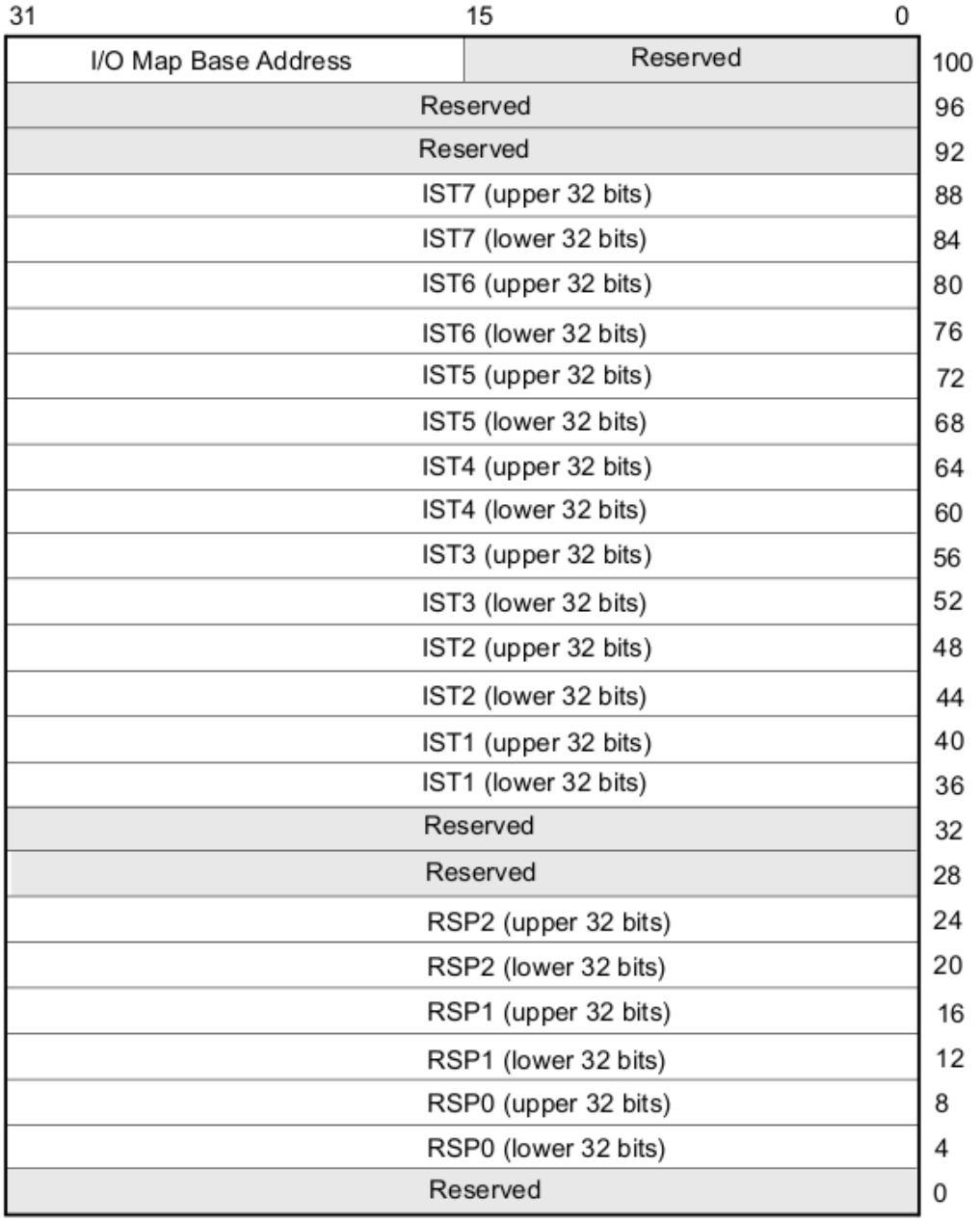
- spracovanie podobné ako v 32-bit móde
- veľkosť zásobníkovej položky 64 bitov
- SS:RSP sa ukladá vždy (aj keď sa CPL nemení)
- cieľový segment musí byť 64-bitový (L=1)
- nuluje SS, ak dochádza k zmene CPL
- zavádza sa možnosť aby niektoré prerušenia mohli použiť vlastný zásobník

TSS a LDT popisovač v IA-32e



AVL	Available for use by system software
B	Busy flag
BASE	Segment Base Address
DPL	Descriptor Privilege Level
G	Granularity
LIMIT	Segment Limit
P	Segment Present
TYPE	Segment Type

TSS v IA-32e


31
15
0
I/O Map Base Address
Reserved
100
Reserved
96
Reserved
92
IST7 (upper 32 bits)
88
IST7 (lower 32 bits)
84
IST6 (upper 32 bits)
80
IST6 (lower 32 bits)
76
IST5 (upper 32 bits)
72
IST5 (lower 32 bits)
68
IST4 (upper 32 bits)
64
IST4 (lower 32 bits)
60
IST3 (upper 32 bits)
56
IST3 (lower 32 bits)
52
IST2 (upper 32 bits)
48
IST2 (lower 32 bits)
44
IST1 (upper 32 bits)
40
IST1 (lower 32 bits)
36
Reserved
32
Reserved
28
RSP2 (upper 32 bits)
24
RSP2 (lower 32 bits)
20
RSP1 (upper 32 bits)
16
RSP1 (lower 32 bits)
12
RSP0 (upper 32 bits)
8
RSP0 (lower 32 bits)
4
Reserved
0

TSS v IA-32e

- RSP0, RSP1, RSP2
  - hodnoty pre RSP pri prepnutí zásobníka pri použití call gate alebo pri prerušení
- IST1 – IST7
  - hodnoty pre RSP pre špecifické zásobníky, ktoré môžu byť použité pre konkrétne prerušenia (podľa nastavenia v interrupt/trap gate)