

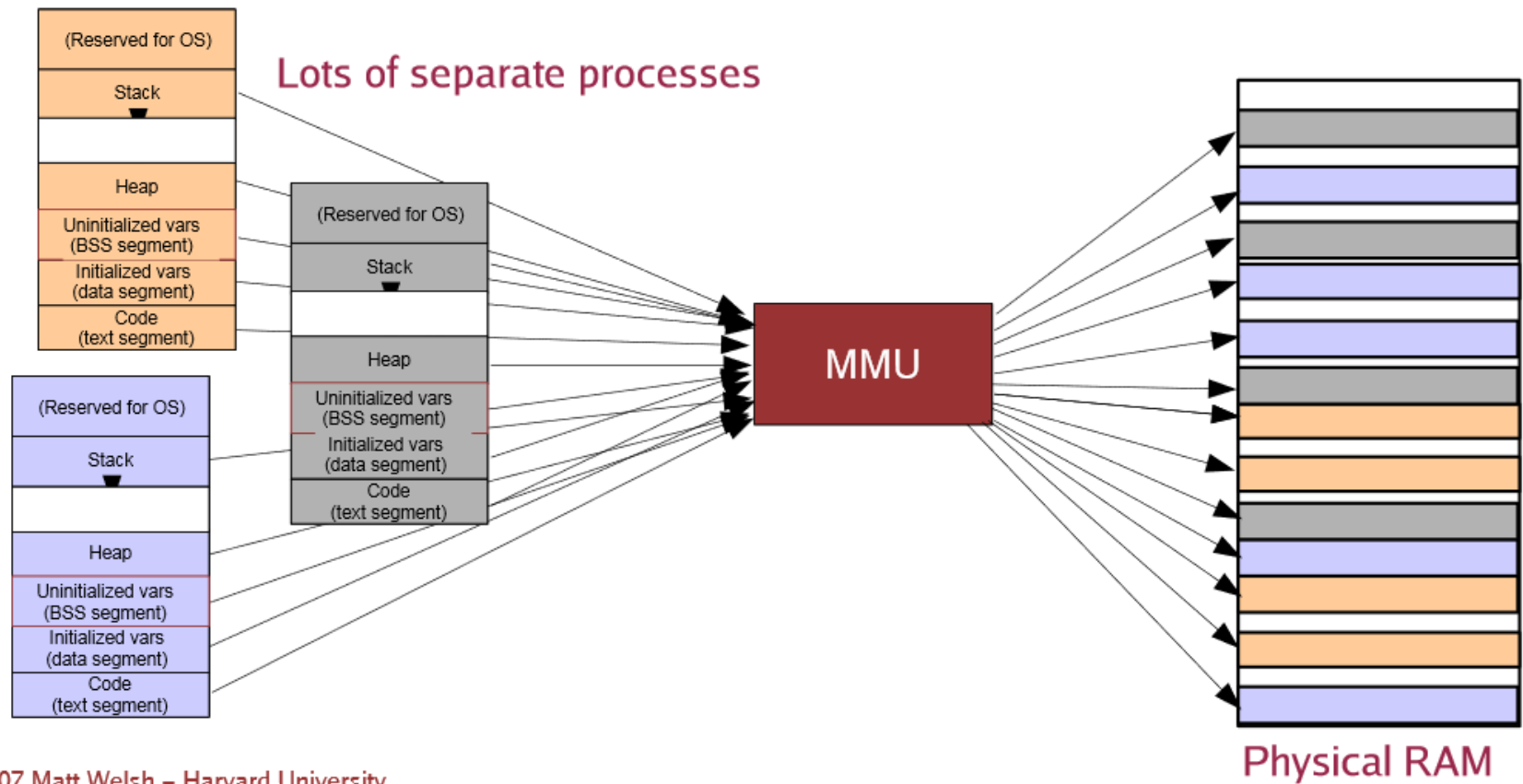
12. Memory management II.

ZOS 2024, L. PEŠIČKA

Stránkování

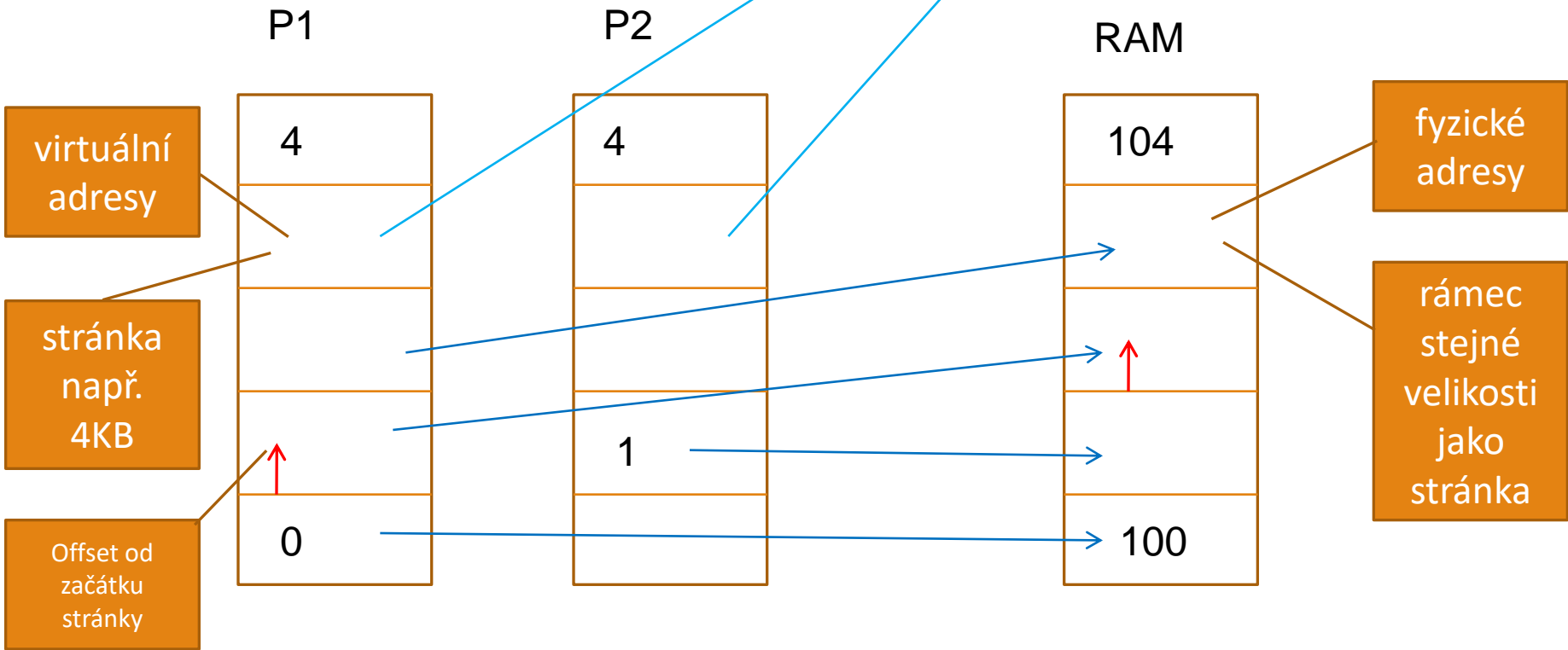
- Každý proces má k dispozici **virtuální adresní prostor** v rozsahu $0 \dots 2^p - 1$ bytů, používá **virtuální adresy** (p .. počet bitů ukazatele, např. 32 bitů)
- Mapování virtuální stránky -> fyzické rámce je pro proces transparentní
- Mapování provádí MMU (memory management unit) s využitím tabulky stránek a TLB cache
- Operační paměť RAM – používá fyzické adresy

Rozptýlení v RAM



Stránková paměť

Swap na disku



Tabulka stránek
Procesu 1

0	->	100
1	->	102
2	->	103

Tabulka stránek
Procesu 2

0	->	115
1	->	101
...		...

Tabulka rámců

100	– obsazený (P1)
101	– obsazený (P2)
...	

Tabulka stránek procesu: 1
Velikost stránky: 4096 B

stránka	rámec	další atributy
0	100	
1	102	
2	103	
3	x	swap: 0
4	110	

Pokud bychom počítali fyzické adresy pro proces 2, používali bychom tabulku stránek procesu 2

Je dána VA 500, vypočítejte fyzickou adresu.
Je dána VA 12300, vypočítejte fyzickou adresu 😊

Je dána VA 4099:
 $4099 / 4096 = 1$, offset 3
Tabulka_stranek_naseho_procesu [1] = 102 .. druhý rámec
 $FA = 102 * 4096 + 3 = 417795$

Výpadek stránky:

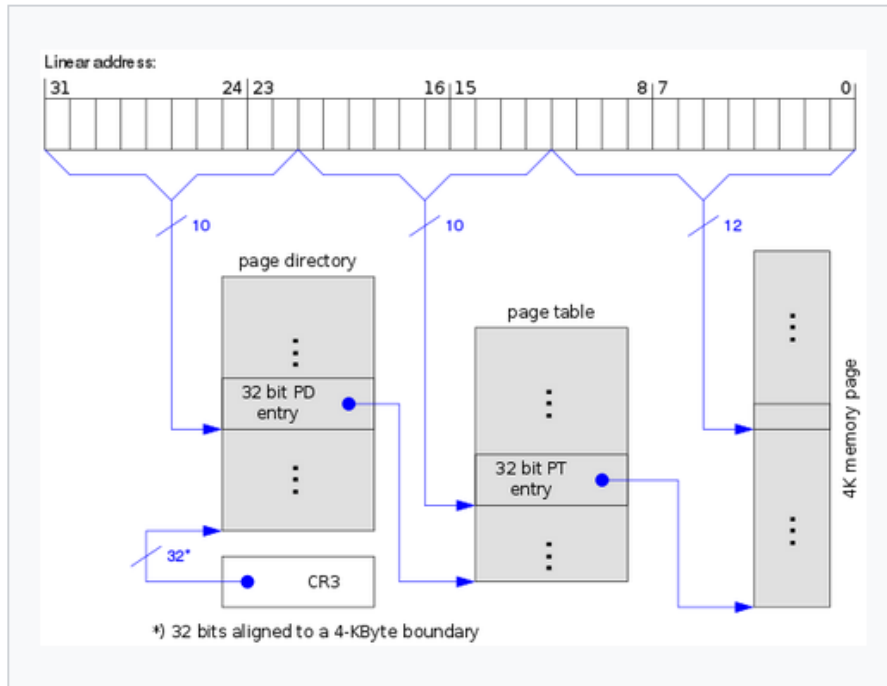
Stránka není v operační paměti, ale ve swapu na disku

Velikost stránky

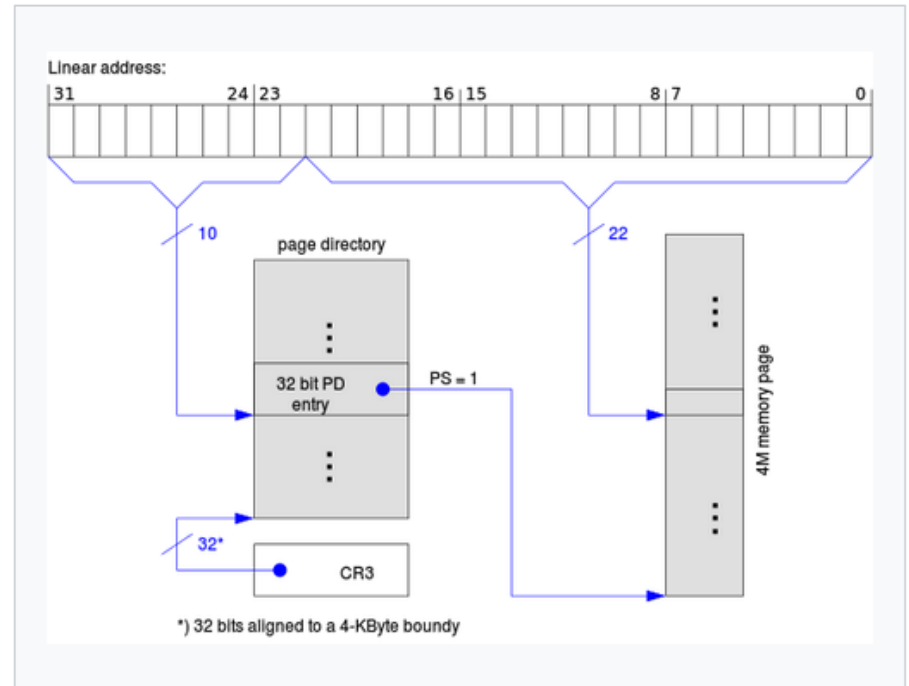
- dvouúrovňová tabulka stránek
 - 4KB, 4MB
- čtyřúrovňová tabulka stránek x86-64
 - stránky 4KB, 2MB, až 1GB

Velikost stránky 4KB vs. 4MB

Page table structures



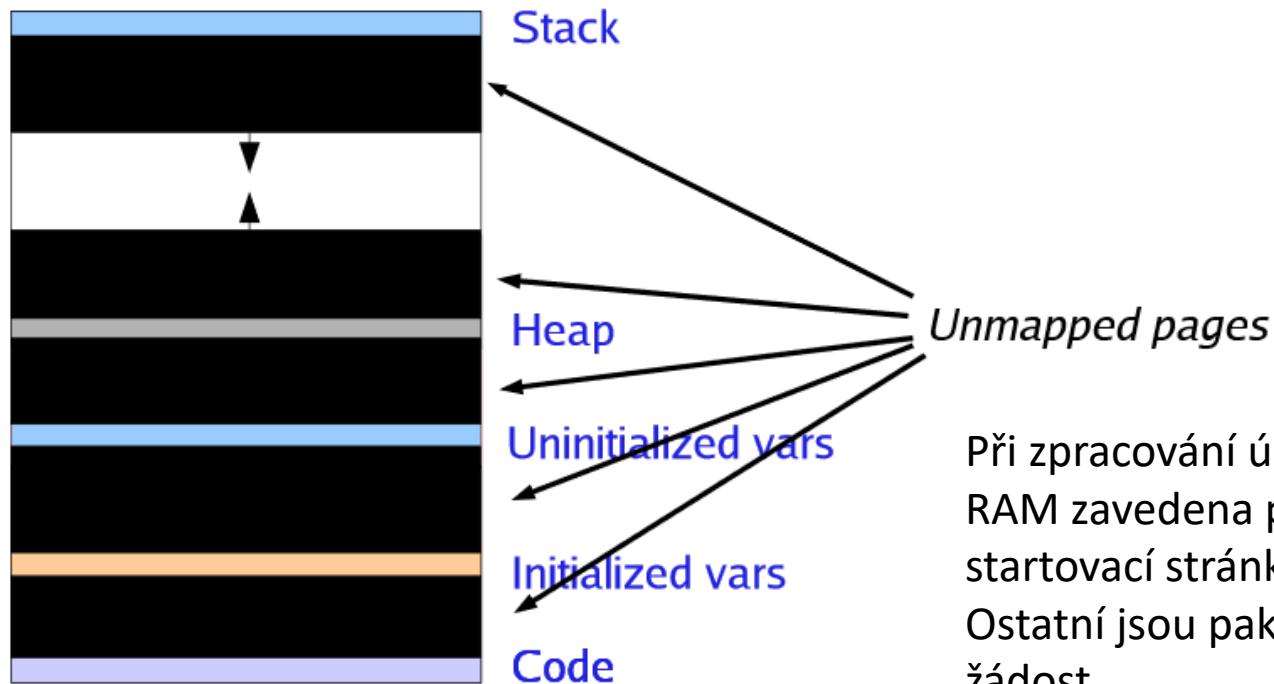
No PAE, 4 KB pages



No PAE, 4 MB pages

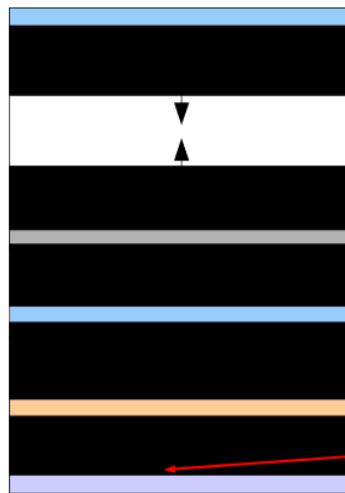
https://en.wikipedia.org/wiki/Physical_Address_Extension

Spuštění procesu

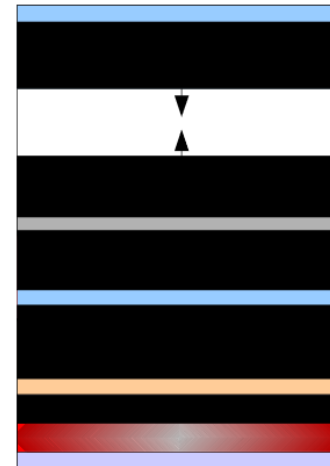


Při zpracování úlohy je do RAM zavedena pouze první startovací stránka. Ostatní jsou pak zaváděny na žádost. Nepotřebné se do operační paměti nedostanou.

Spuštění procesu

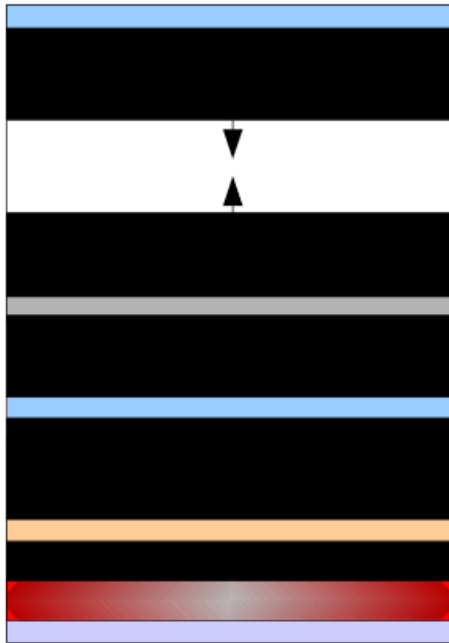


Reference next instruction

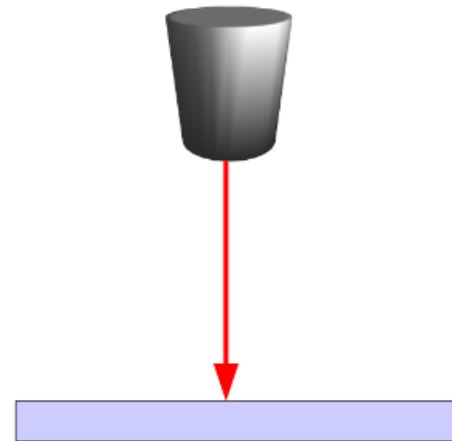


Page fault!!!

Spuštění procesu



*OS reads missing page
from executable file on
disk*



Poznámky

- Sdílené regiony paměti
 - Volání `shmget`, `shmat`, `shmdt` (viz cvičení)
 - „chci sdílenou paměť 2000B s klíčem 5678“
- Paměťově mapované soubory
 - Volání `mmap`
 - Soubor na disku se tváří jako blok paměti
 - Zápis do stránky -> nastaví bit modified
 - Při odstranění z paměti zapíše na disk

Mechanismus VP - výhody

- **Rozsah** virtuální paměti
 - (32bit: 2GB pro proces + 2GB pro systém, nebo 3 proces +1 systém)
 - Adresový prostor úlohy není omezen velikostí fyzické paměti
 - Multiprogramování – není omezeno rozsahem fyzické paměti
- **Efektivnější** využití fyzické paměti
 - Není vnější fragmentace paměti
 - Nepoužívané části adresního prostoru úlohy nemusejí být ve fyzické paměti

Mechanismus VP - nevýhody

- **Režie při převodu** virtuálních adres na fyzické adresy
- **Režie procesoru**
 - údržba tabulek stránek a tabulky rámců
 - výběr stránky pro vyhození, plánování I/O
- **Režie I/O** při čtení/zápisu stránky
- **Paměťový prostor** pro tabulky stránek
 - Tabulky stránek v RAM, často používaná část v TLB
- **Vnitřní fragmentace**
 - Přidělená stránka nemusí být plně využita

Rozdělení paměti pro proces (!!!)

pokus.c:

```
int x = 5; int y = 7;      // inicializovaná data
```

```
void fce1() {
```

```
    int pom1, pom2;      // na zásobníku
```

```
    ... }
```

```
int main (void) {
```

```
    ...
```

```
    ptr = malloc(1000);   // halda
```

```
    fce1();
```

```
    return 0;
```

```
}
```



2+2: 0..2GB proces, 2GB..4GB OS

3+1: 3GB proces, 1GB OS

Rozdělení paměti pro proces

Roste halda



Roste zásobník



Máme-li více vláken => více zásobníků, limit velikosti zásobníku

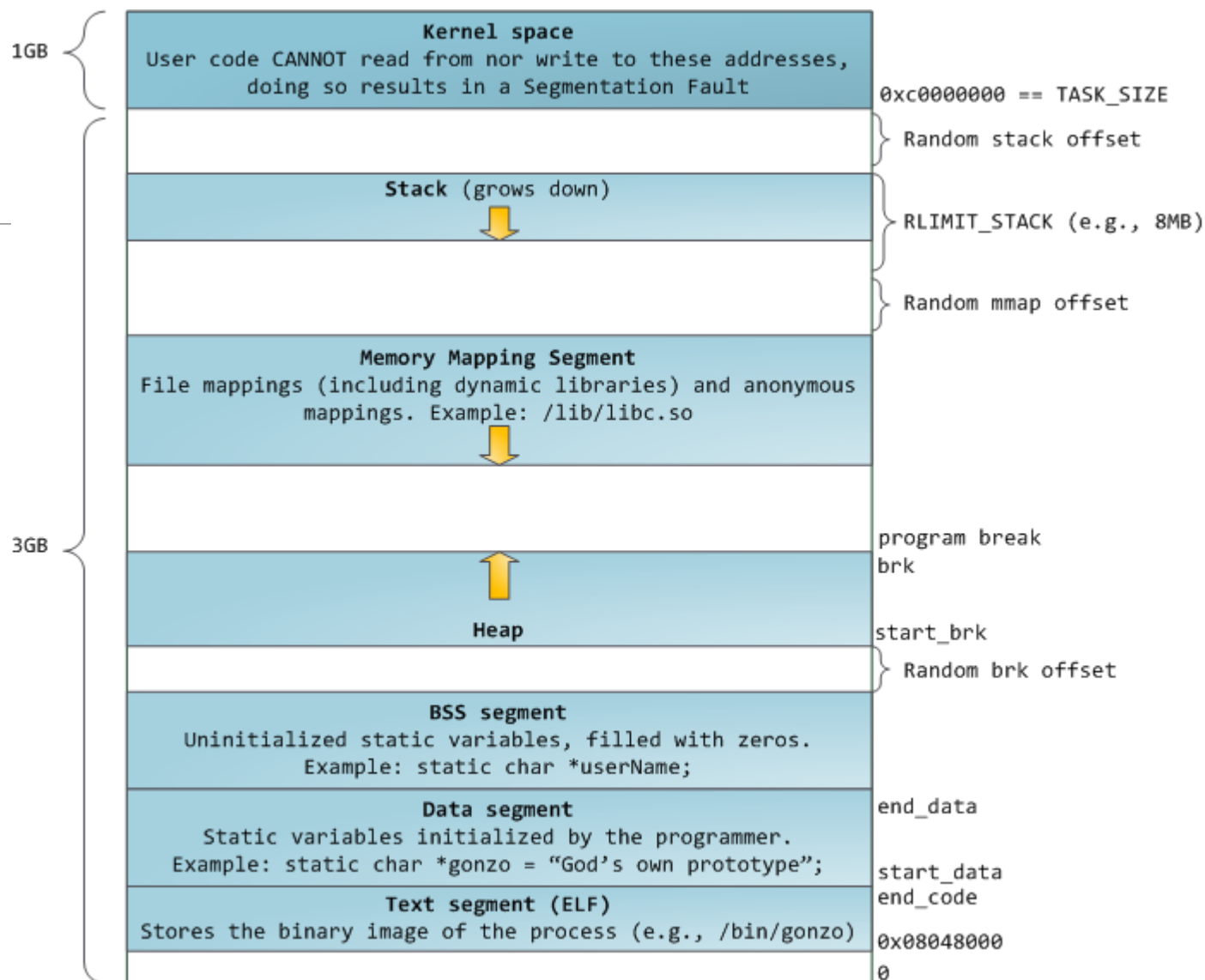


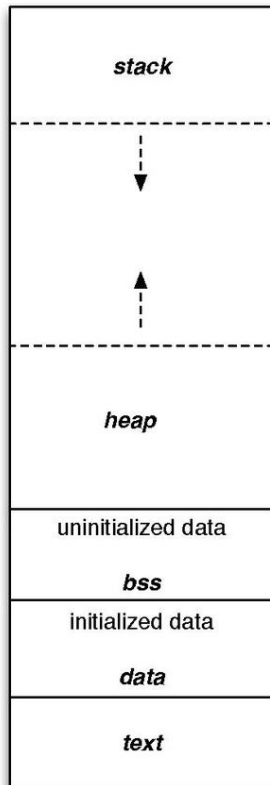
poznámka

- sdílená knihovna (jednou v RAM), může vyžadovat vlastní datovou oblast, privátní pro každý proces, kam si ukládá svoje data
- Často v OS snaha mapovat sdílené knihovny od různých virtuálních adres při každém běhu daného programu (aby vždy nebyla od stejné adresy) – z bezpečnostních důvodů

Další ukázka
rozdělení
paměti

zapamatovat
pojem
BSS
=
neinicializova
ná data





Ukázat:

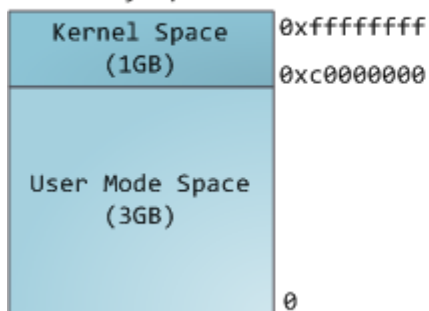
https://en.wikipedia.org/wiki/Data_segment

Zdroj:

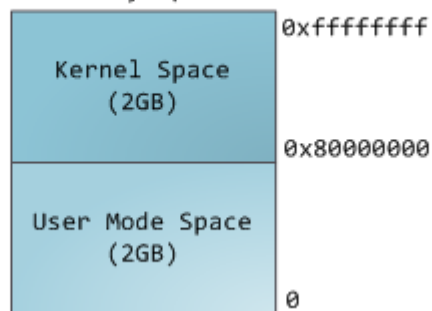
<https://en.wikipedia.org/wiki/.bss>

Rozdělení paměti

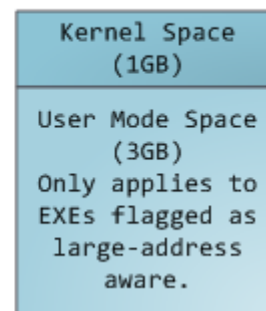
Linux User/Kernel
Memory Split



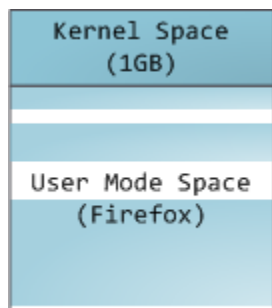
Windows, default
memory split



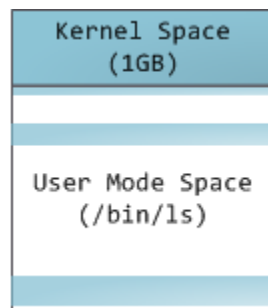
Windows booted
with /3GB switch



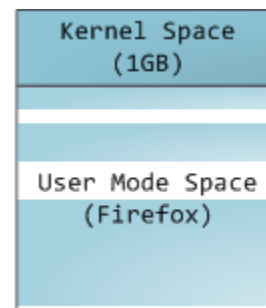
rozdělení
paměti
Linux,
Windows



Process
Switch
→



Process
Switch
→



přepínání
kontextu
mezi
různými
procesy

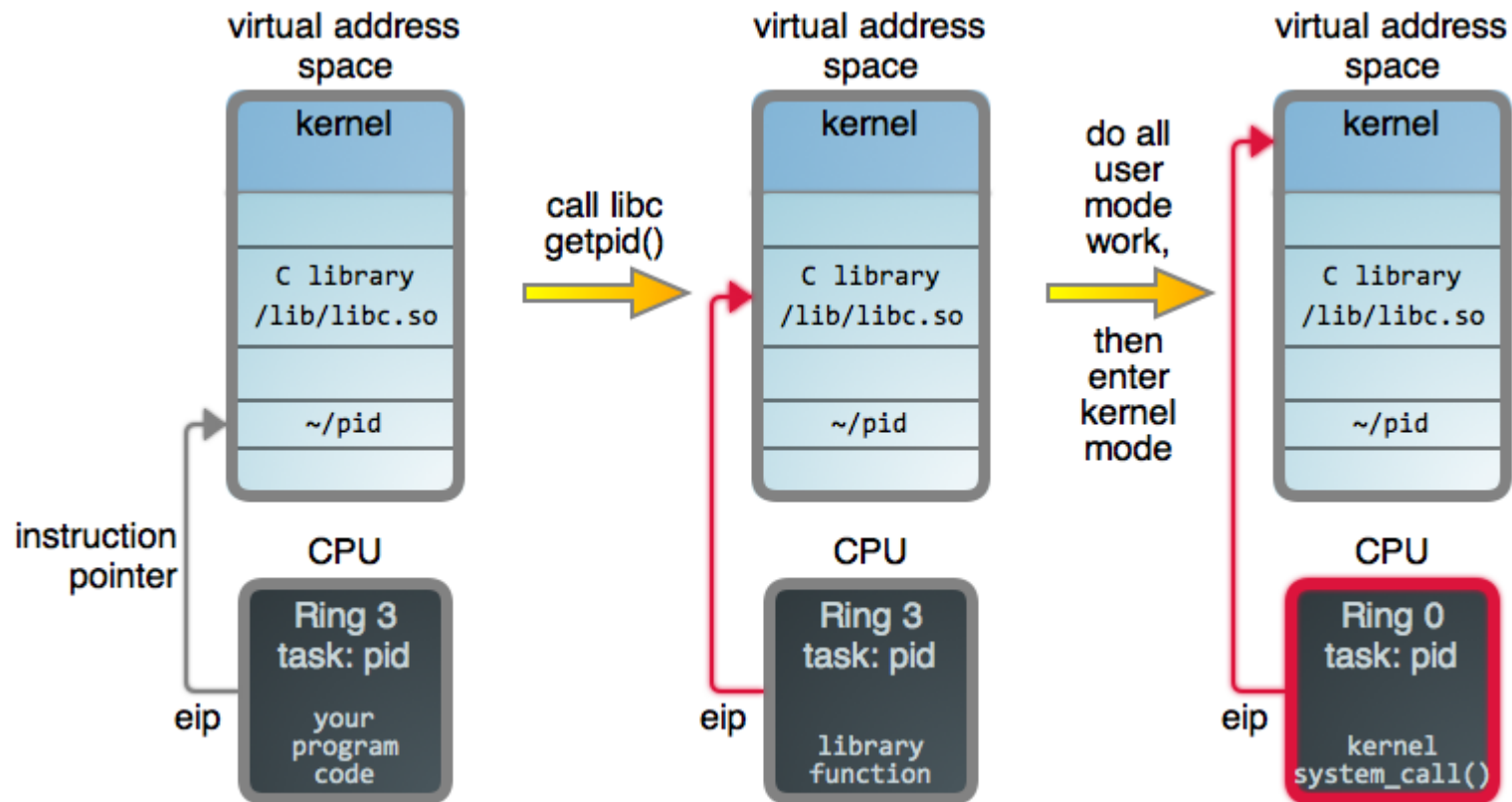
Příklad getpid

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

```
int main() {
    pid_t p = getpid();
    printf("%d\n", p);
}
```

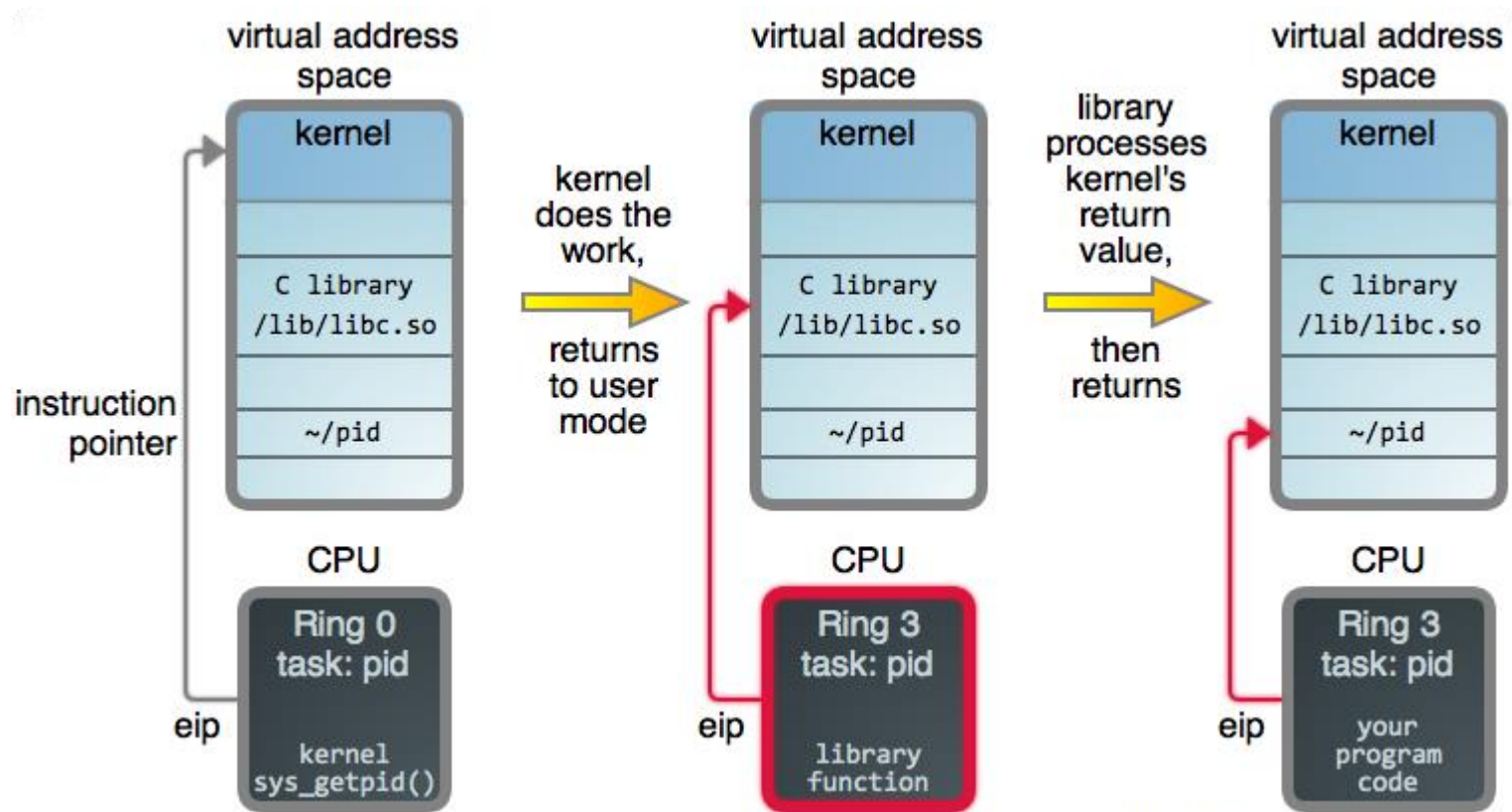
<https://manybutfinite.com/post/system-calls/>

Příklad getpid – 1. část



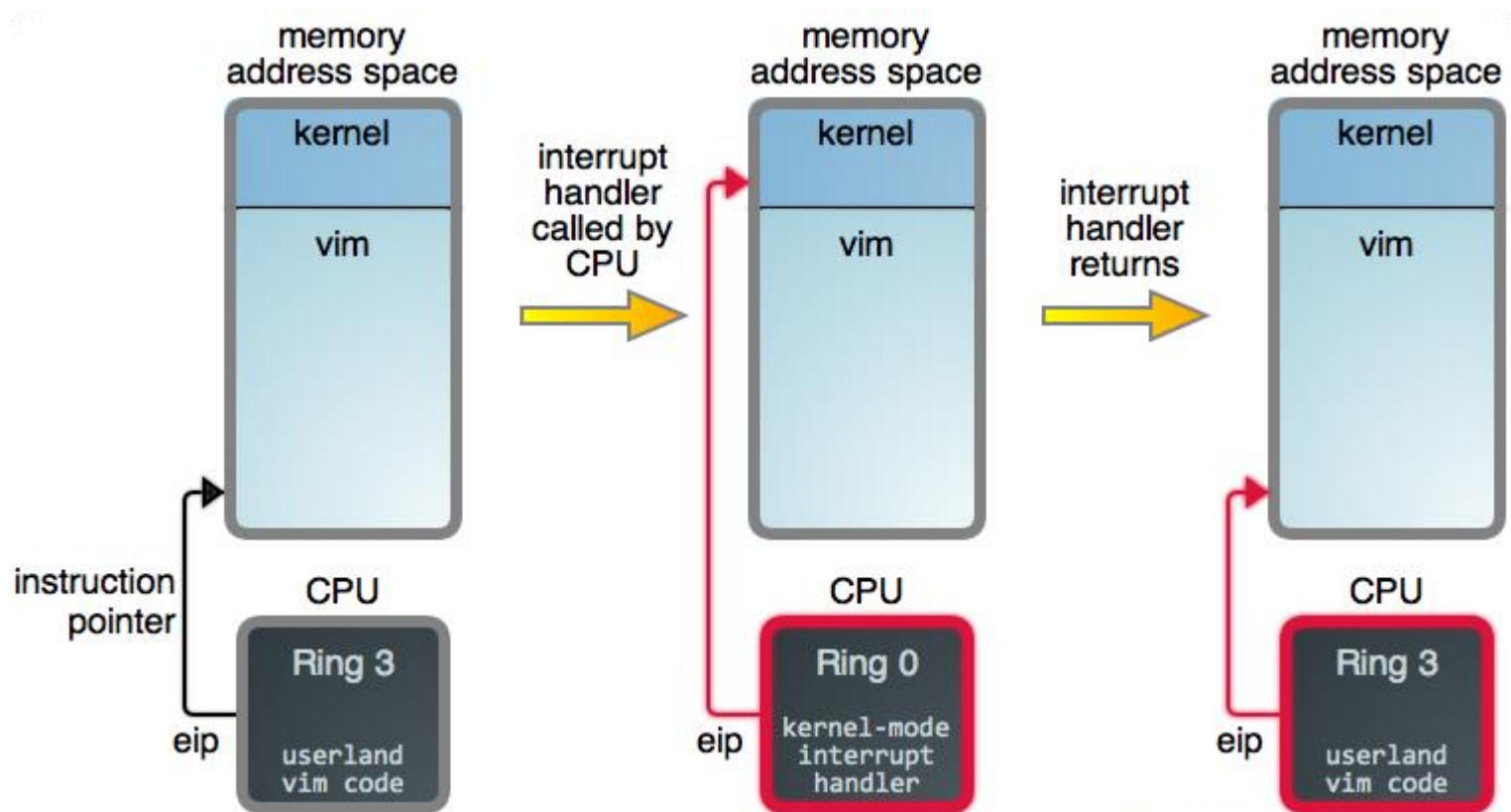
<http://duartes.org/gustavo/blog>

Příklad getpid – 2. část



<http://duartes.org/gustavo/blog>

Příklad – obsluha přerušení



<http://duartes.org/gustavo/blog>

Příklad – obsluha přerušení

- příchod přerušení -> přepne do režimu jádra (ring 0)
- Nezmění aktivní úlohu
- Úloha čeká na návrat z obsluhy přerušení

Poznámky

Bylo popsáno stránkování.

Další technikou virtuální paměti je segmentace.

Nakonec budou ukázány systémy, které používají obou technik – segmentace se stránkováním.

Moderní CPU podporují segmentaci i stránkování, typicky segmentaci se stránkováním.

Segmentace

- Dosud diskutovaná VP – **jednorozměrná**
 - Proces: adresy < 0 , **maximální virtuální adresa** $>$
- Často výhodnější – **více samostatných virtuálních** adresových prostorů
- Př. – samostatný segment pro haldu a zásobník, každý z nich se může zvětšovat („nesrazí se“)
- Paměť nejlépe více nezávislých adresových prostorů - **segmenty**

Segmentace

- **Segment** – logické seskupení informací
- Každý segment – **lineární** posloupnost adres od **0**
- Programátor o segmentech **ví**, používá je **explicitně** (adresuje konkrétní segment)
- Např.
 - Kód přeloženého programu (CS)
 - Globální proměnné
 - Hromada (DS)
 - Zásobník návratových adres (SS)

Segmentace

- Každý **segment** – **logická entita**
→ má smysl, aby měl **samostatnou ochranu**
- Různá granularita
 - Můžeme si představit segment pro kód, data, zásobník
 - Sdílená knihovna – samostatný segment – sdílený mezi více programy
 - Vždy záleží na konkrétním systému
 - Dnes se spíše preferuje stránkování

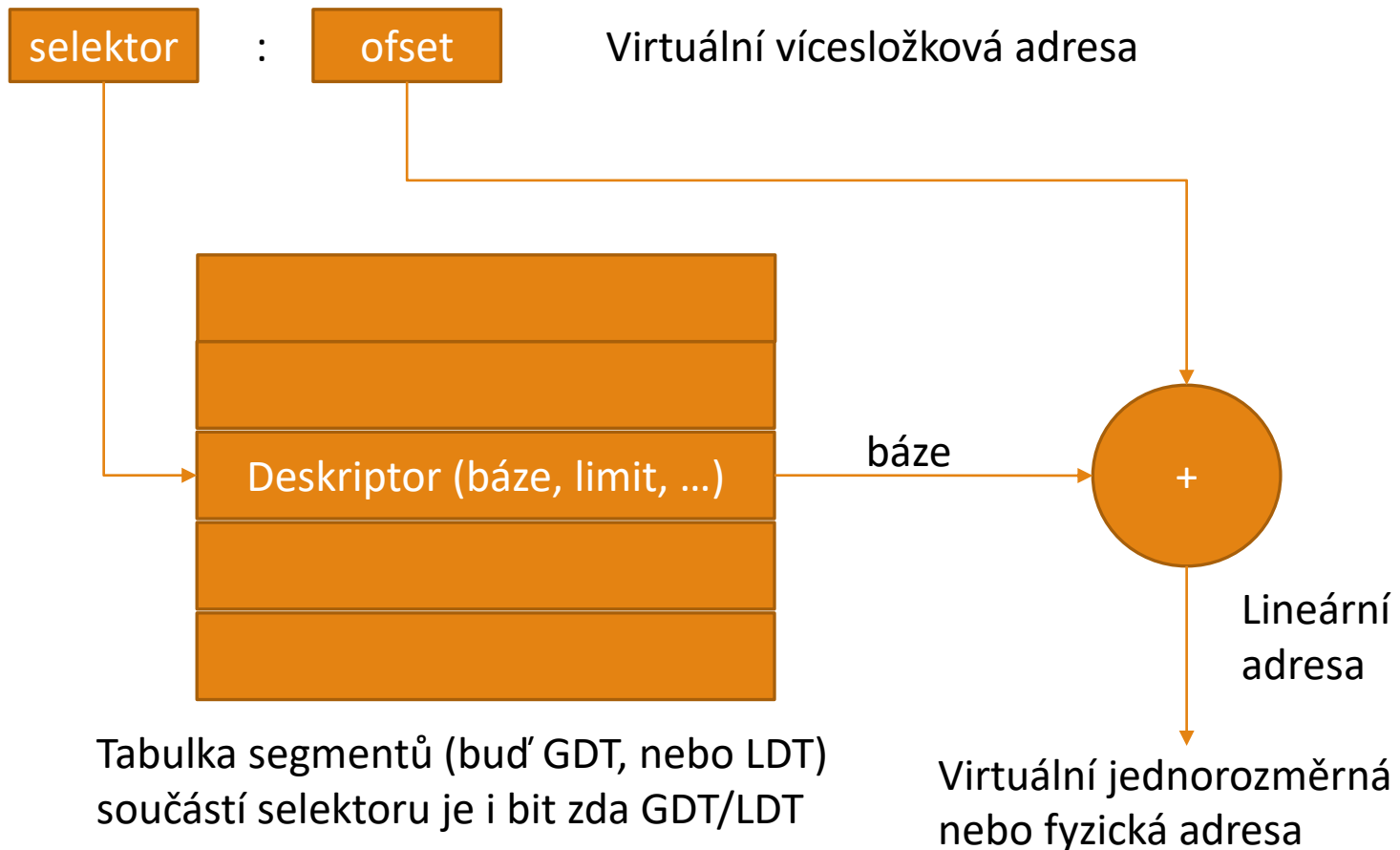
Čistá segmentace

- Každý odkaz do paměti – dvojice (**selektor, offset**)
 - **Selektor** – obsahuje mj. odkaz do tabulky segmentů, určuje segment
 - **Offset** – relativní adresa v rámci segmentu
- Technické prostředky musí umět přemapovat dvojici (selektor, offset) na **lineární adresu** (je již fyzická když není dále stránkování)
- **Tabulka segmentů** – každá položka má
 - Počáteční adresa segmentu (**báze**)
 - Rozsah segmentu (**limit**)
 - Příznaky ochrany segmentu (čtení, zápis, provádění – rwx)

GDT/LDT

- Tabulka segmentů – obsahuje deskriptory segmentu
- GDT (Globální tabulka deskriptorů segmentu)
 - Jedna globální pro celý systém
- LDT (Lokální tabulka deskriptorů segmentu)
 - Každý proces může mít vlastní lokální tabulku segmentů
- Jedním bitem GDT/LDT volíme, jaká z těchto tabulek se použije
- Systém musí být schopen najít, kde leží LDT daného procesu
- Někdy se využívá např. jen globální tabulka deskriptorů (GDT)

Selektor, offset, deskriptor (!!)

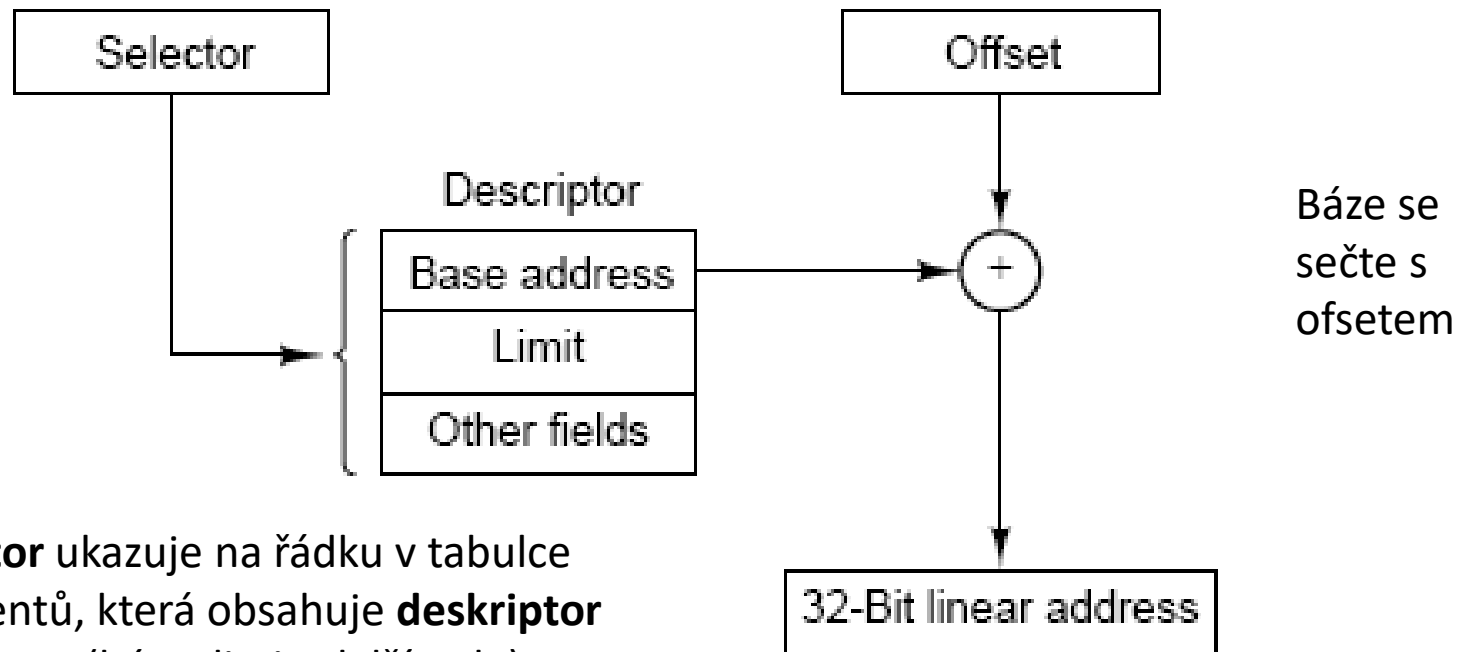


Kontroly (!)

Několik kontrol porušení ochrany paměti

- Ukazuje selektor na vyplněnou řádku v tabulce segmentů?
 - Pokud ne (např. samé 0) – ukončení procesu
- Platí, že offset \leq limit?
 - Pokud ne – ukončení procesu
- Podle příznaků, nechci sahat na segment, kam může např. jen jádro?
 - Opět může vést k ukončení procesu
- Lineární adresa
 - Jedno číslo od 0 výše
 - Může být **fyzickou** adresou do RAM, pokud máme **jen segmentaci**
 - Může být **virtuální** adresou, pokud ještě aplikujeme **dále stránkování**

(selektor, offset) => lineární adresa



Selektor ukazuje na řádku v tabulce segmentů, která obsahuje **deskriptor** segmentu (báze, limit, další pole)

Lineární adresa = jednorozměrná 0..xyz

Převod na fyzickou adresu (!!)

- PCB obsahuje odkaz na **tabulku segmentů** procesu (když má LDT)
- Odkaz do paměti má tvar (selector: offset)
- Možnost **sdílet** segment mezi **více** procesy

příklad instrukce: *LD R, sel:offset*

1. Selektor – **index** do tabulky segmentů – obsahuje descriptor segmentu
2. Kontrola **offset < limit**, ne – porušení ochrany paměti
3. Kontrola zda dovolený způsob použití; ne – chyba
4. Adresa = **báze + offset**

Segmentace

Mnoho věcí podobných jako přidělování paměti po sekcích, ale rozdíl:

- Po sekcích – pro různé procesy
- Segmenty – pro části procesu

Stejné problémy jako přidělování paměti po sekcích

- Externí fragmentace paměti
- Mohou zůstat malé díry

Segmentace na žádost

- Segment – **zavedený** v paměti nebo **odložený** na disku
- Adresování segmentu co není v paměti – **výpadek** segmentu – zavede do paměti – není-li místo – jiný segment odložen na disk
- HW podpora – bity v tabulce segmentů
 - Bit segment je zaveden v paměti (Present / absent)
 - Bit referenced
- Používal např. systém OS/2 pro i80286 – pro výběr segmentu k odložení algoritmus Second Chance

Segmentace se stránkováním

- velké segmenty – nepraktické celé udržovat v paměti
- Stránkování segmentů
 - V paměti pouze potřebné stránky

Adresy (!!!)

virtuální adresa -> lineární adresa -> fyzická adresa

virtuální – používá proces (sektor:offset)

lineární – po segmentaci (už jednorozměrné číslo od 0 výše)
pokud není dále stránkování, tak už
představuje i fyzickou adresu

fyzická – adresa do fyzické paměti RAM
(CPU jí vystaví na sběrnici)

Dnešní procesory

- **segmentace**
- **stránkování**
(některé CPU ale neumí segmentaci vypnout,
pak je segmentace se stránkováním)
- **segmentace se stránkováním**
- **tabulka LDT (Local Descriptor Table)**
 - Např. každý proces může mít vlastní
 - Např. segmenty lokální pro proces (kód,data,zásobník)
- **tabulka GDT (Global Descriptor Table)**
 - pouze jedna, sdílená všemi procesy
 - systémové segmenty, včetně OS

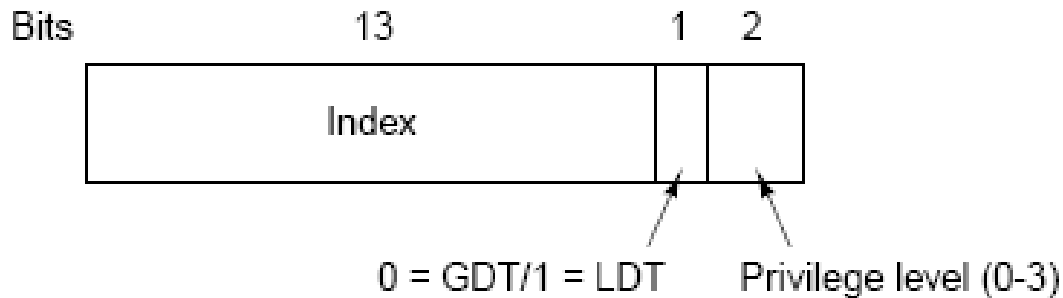
využití LDT tabulek často
operační systémy nahrazují
stránkováním

Segmentové registry

- Pentium a výše má 6 segmentových registrů:
 - CS (Code Segment)
 - DS (Data Segment)
 - SS (Stack Segment)
 - další: ES, FS, GS
- přístup do segmentu
→ do segmentového registru se zavede **selektor segmentu**

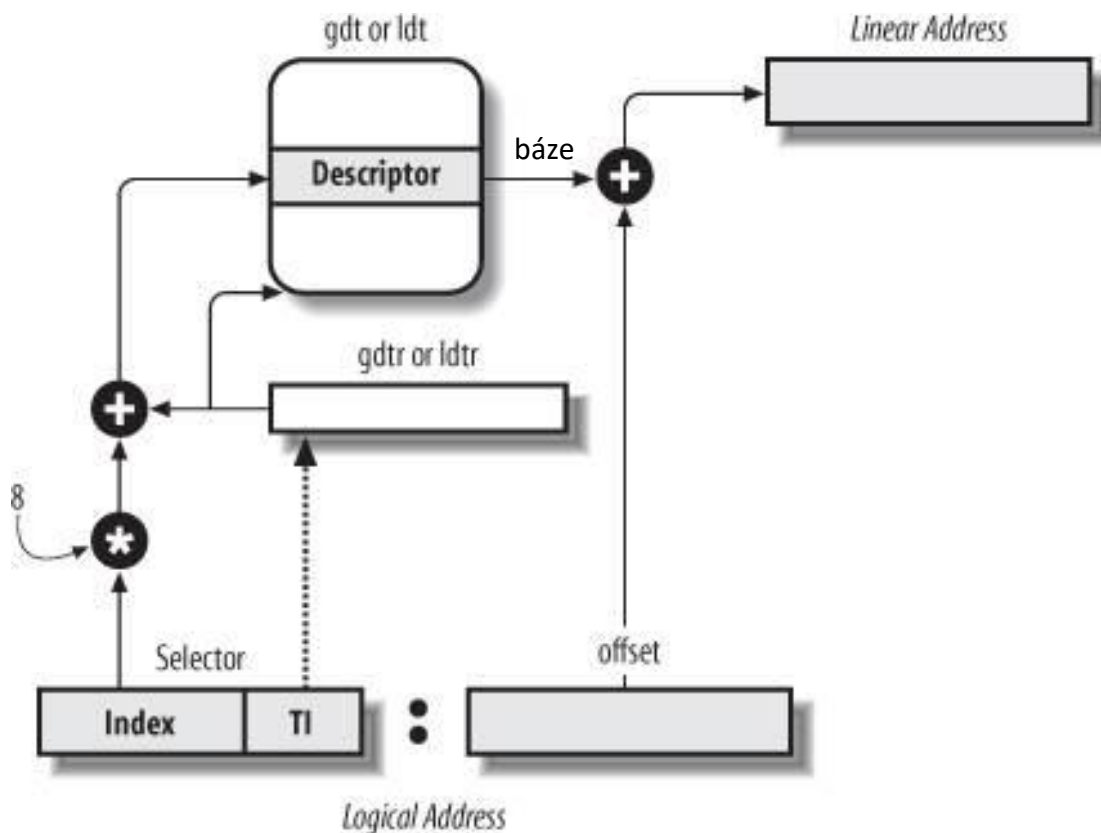
Selektor segmentu (!!)

- Selektor – 16bitový
- 13bitů – index to GDT nebo LDT
- 1 bit – 0=GDT, 1=LDT
- 2 bity – úroveň privilegovanosti (0-3, 0 – jádro, 3 – uživ. proces)



Selektor segmentu

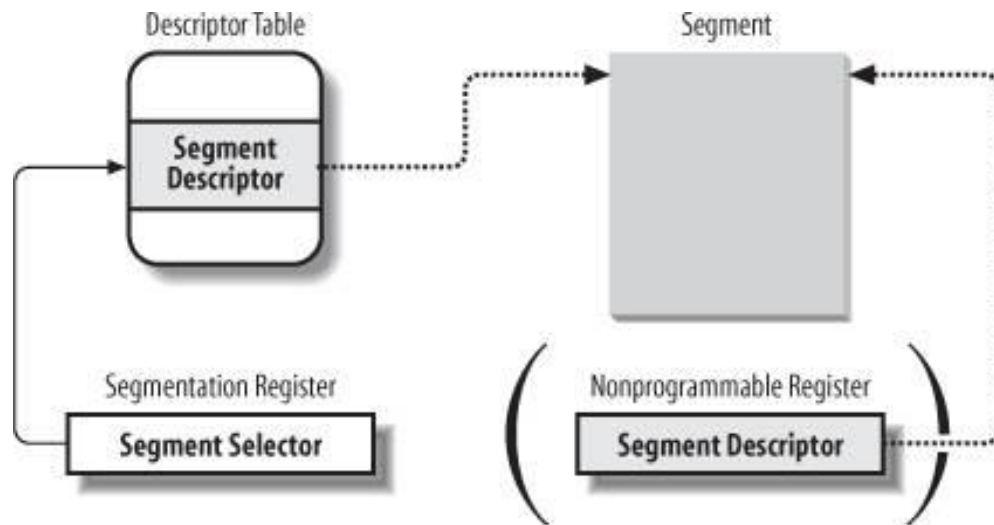
- 13 bitů – index, tj. $\max 2^{13} = 8192$ položek
 - selektor 0 – indikace nedostupnosti segmentu
- při zavedení selektoru do segmentového registru CPU také zavede odpovídající **descriptor** z LDT nebo GDT do vnitřních registrů CPU (pro urychlení)
 - bit 2 selektoru – pozná, zda LDT nebo GDT
 - popisovač segmentu na adrese (selektor and 0fff8h) + začátek tabulky



1. z TI pozná, zda použije GDT nebo LDT
2. z indexu selektoru spočte adresu deskriptoru
3. přidá offset k bázi (viz deskriptor), získá lineární adresu

neprogramovatelné registry spojené se segmentovými registry

Rychlý přístup k deskriptoru segmentu (!)



Zrychlení -> nemusí se pořád dívat do tabulky deskriptorů

logická adresa:

segment selektor + offset
(16bitů) (32bitů)

zrychlení převodu:

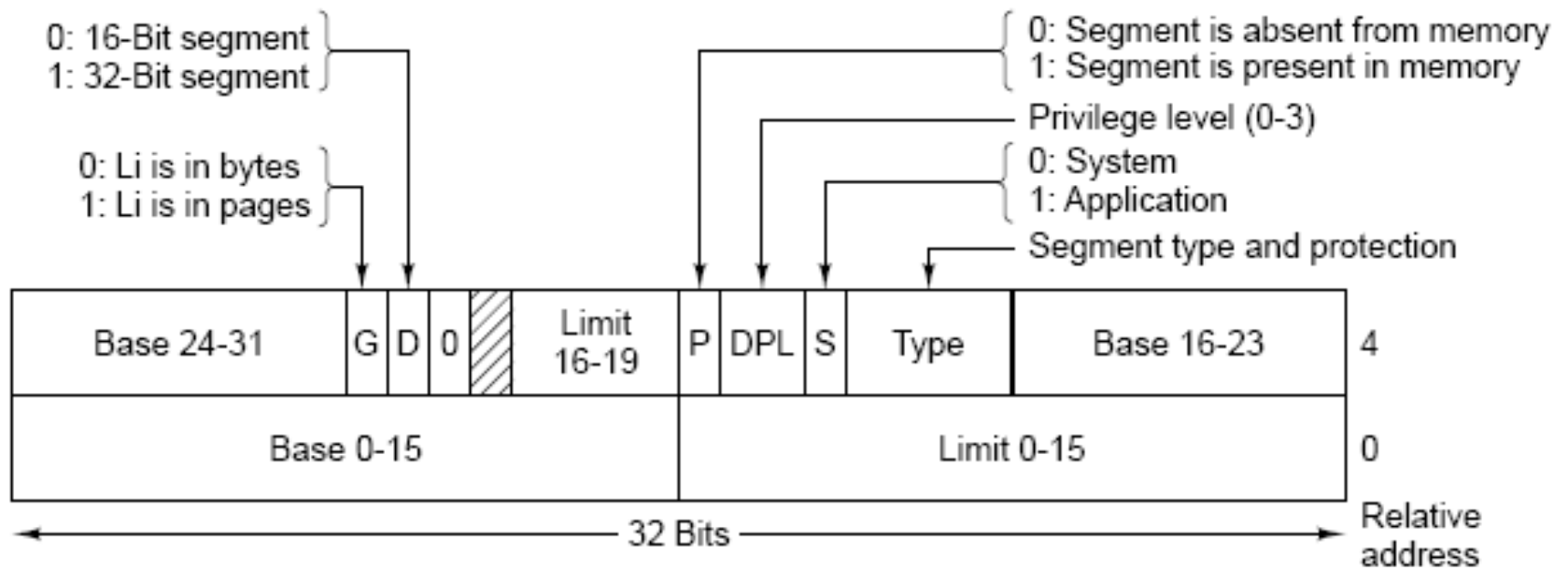
přídavné neprogramovatelné registry (pro každý segmentový registr)

když se nahraje segment **selektor** do segmentového registru, odpovídající **deskriptor** se nahraje do odpovídajícího neprogramovatelného registru

Deskriptor segmentu (!!)

- 64bitů
 - 32 bitů **báze**
 - 20 bitů **limit**
 - v bytech, do 1MB (2^{20})
 - v 4K stránkách (do 2^{32}) ($2^{12} = 4096$)
 - **příznaky**
 - typ (kód nebo data/zásobník) a ochrana segmentu
 - segment přítomen v paměti..

Deskriptor segmentu (8 bytů)



Podrobný popis

Úplný popis jednotlivých hodnot např. zde:

https://en.wikipedia.org/wiki/Segment_descriptor

DPL – privilege level (ring) požadovaný pro přístup k tomuto deskriptoru (porovná privilegia v selektoru s DPL, např. uživatelský 3 by chtěl přistoupit k segmentu s privilegiem jádra 0 -> ochrana)

Type

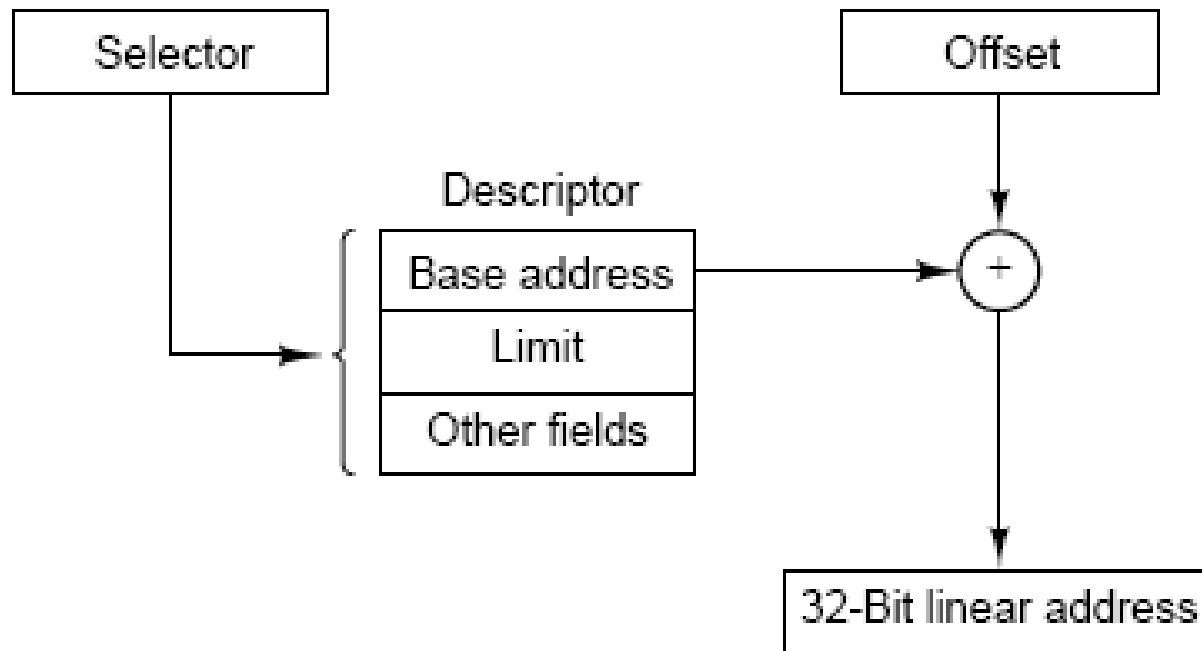
– kódový segment nebo data/zásobník

- C – kód může být volaný z méně privilegovaných úrovní
- E – zda roste od báze adresy k báze+limit, nebo opačně (zásobník)
- R – readable, když ne lze z něj spustit kód, ale ne číst
- W - zda lze do něj zapisovat
- A – zda je přístupovaný, nastavovaný HW, shazovaný softwarově

Konverze na fyzickou adresu

- Proces adresuje paměť pomocí **segmentového registru**
- CPU použije odpovídající **deskriptor segmentu** v interních registrech
- pokud segment není – výjimka
- kontrola offset > limit – výjimka
- 32bit. **lineární adresa** = **báze** + **offset**
- není-li stránkování – jde již i o **fyzickou** adresu
- je-li stránkování, další převod přes tabulku stránek

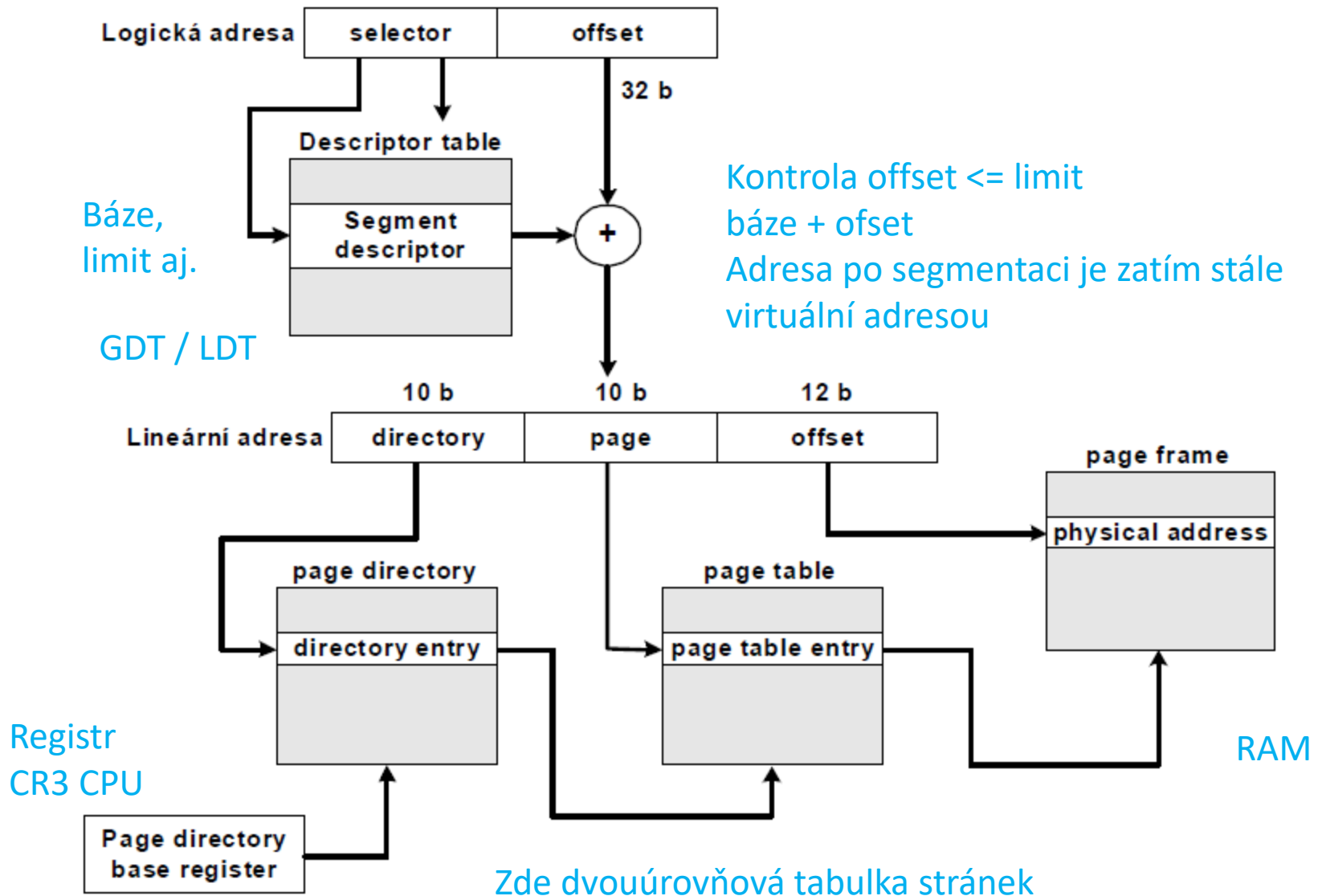
Konverze na fyzickou adresu I.



Konverze na fyzickou adresu II.

- Pokud je dále stránkování –
lineární adresa je VA pro stránkování,
mapuje se na fyzickou pomocí tabulek stránek
- dvouúrovňové mapování

Komplexní schéma převodu VA na FA (pamatovat !!!!)



Poznámka

- Často některé OS využívají primárně stránkování
 - „ošizená“ segmentace
 - segmentaci nejde vypnout!
 - použijeme segmenty, ale dáme je přes celou paměť
 - base = 0
 - limit = MAX
- Segmenty pro kód a data (systémové ring 0, uživatelské ring 3)
 - Kernel Code
 - Kernel Data
 - User Code
 - User Data

Poznámky - důležité

Jakou strategii moderní systémy využívají?

<http://stackoverflow.com/questions/24358105/do-modern-oss-use-paging-and-segmentation>

Modern OSes "do not use" segmentation. Its in quotes because they use 4 segments: Kernel Code Segment, Kernel Data Segment, User Code Segment and User Data Segment. What does it means is that all user's processes have the same code and data segments (so the same segment selector). The segments only change when going from user to kernel. So, all the path explained on the section 3.3. occurs, but they use the same segments and, since the page tables are individual per process, a page fault is difficult to happen.

4 segmenty:
Kernel Code
Kernel Data
User Code
User Data

<http://stackoverflow.com/questions/3029064/segmentation-in-linux-segmentation-paging-are-redundant> - ještě více vysvětleno

Implicitní nebo explicitní určení segmentu

Příklad v assembleru:

■ Implicitní

- `jmp $8052905`

-- implicitně použije **CS**
(instrukce skoku)

- `mov $8052905, %eax`

-- implicitně použije **DS**
(instrukce manipulace s daty)

■ Explicitní

- `mov %ss:$8052905, %eax`

-- explicitně použije **SS**

Linux segmentation

- ❑ Since x86 segmentation hardware cannot be disabled, Linux just uses NULL mappings
- ❑ Linux defines four segments
 - Set segment base to 0x00000000, limit to 0xffffffff
 - segment offset == linear addresses
 - User code (segment selector: __USER_CS)
 - User data (segment selector: __USER_DS)
 - Kernel code (segment selector: __KERNEL_CS)
 - Kernel data (segment selector: __KERNEL_DATA)
 - arch/i386/kernel/head.S

Segment protection

- ❑ **Current Privilege level (CPL)** specifies privileged mode or user mode
 - Stored in current code segment descriptor
 - User code segment: **CPL = 3**
 - Kernel code segment: **CPL = 0**
- ❑ **Descriptor Privilege Level (DPL)** specifies protection
 - Only accessible if **CPL ≤ DPL**
- ❑ Switch between user mode and kernel mode (e.g. system call and return)
 - Hardware load the corresponding segment selector (**__USER_CS** or **__KERNEL_CS**) into register **cs**

Paging

- ❑ Linux uses up to 4-level hierarchical paging
- ❑ A linear address is split into five parts, to seamlessly handle a range of different addressing modes
 - Page Global Dir
 - Page Upper Dir
 - Page Middle Dir
 - Page Table
 - Page Offset
- ❑ Example: 32-bit address space, 4KB page without **physical address extension** (hardware mechanism to extend address range of physical memory)
 - Page Global dir: 10 bits
 - Page Upper dir and Page Middle dir are not used
 - Page Table: 10 bits
 - Page Offset: 12 bits

Paging in 64 bit Linux

Platform	Page Size	Address Bits Used	Paging Levels	Address Splitting
Alpha	8 KB	43	3	10+10+10+13
IA64	4 KB	39	3	9+9+9+12
PPC64	4 KB	41	3	10+10+9+12
sh64	4 KB	41	3	10+10+9+12
X86_64	4 KB	48	4	9+9+9+9+12

64bitové systémy

- x86-64 architektura nepoužívá segmentaci v long modu CPU (64bitový mód)
- CS, SS, DS, ES – báze 0, limit 2^{64}
- FS, GS – stále mohou mít nenulové adresy báze, lze je využít pro speciální účely
- <https://www.codeproject.com/Articles/1273844/The-Intel-Assembly-Manual-3>
- https://en.wikipedia.org/wiki/Memory_segmentation

Poznámky – 64bitové systémy

<http://stackoverflow.com/questions/26898104/is-memory-segmentation-implemented-in-the-latest-version-of-64-bit-linux-kernel>

<http://stackoverflow.com/questions/21165678/why-64-bit-mode-long-mode-doesnt-use-segment-registers>

http://en.wikipedia.org/wiki/Memory_segmentation

Řídící registry CPU

(Registry CR0 až CR4 – nastavení CPU), v tabulce obsah **CRO**:

Bit	Name	Full Name	Description
31	PG	Paging	If 1, enable paging and use the CR3 register, else disable paging
30	CD	Cache disable	Globally enables/disable the memory cache
29	NW	Not-write through	Globally enables/disable write-back caching
18	AM	Alignment mask	Alignment check enabled if AM set, AC flag (in EFLAGS register) set, and privilege level is 3
16	WP	Write protect	Determines whether the CPU can write to pages marked read-only
5	NE	Numeric error	Enable internal x87 floating point error reporting when set, else enables PC style x87 error detection
4	ET	Extension type	On the 386, it allowed to specify whether the external math coprocessor was an 80287 or 80387
3	TS	Task switched	Allows saving x87 task context upon a task switch only after x87 instruction used
2	EM	Emulation	If set, no x87 floating point unit present, if clear, x87 FPU present
1	MP	Monitor co-processor	Controls interaction of WAIT/FWAIT instructions with TS flag in CR0
0	PE	Protected Mode Enable	If 1, system is in protected mode, else system is in real mode

Řídící registry CPU

- **CR2** – obsahuje adresu, která způsobila výjimku výpadek stránky
- **CR3** – obsahuje adresu page directory (aby věděl, kde začíná datová struktura pro tabulku stránek)

Další viz:

http://en.wikipedia.org/wiki/Control_register

Procesory x86

- real mode (MS-DOS)

- po zapnutí napájení, žádná ochrana paměti
- $FA = \text{Segment} * 16 + \text{Offset}$
- FA 20bitová, segment, offset .. 16bitové

V reálném režimu CPU (např. MS DOS) používal segmenty jinak, než jsme zvyklí

- protected mode (dnešní OS) – to co zde popisujeme

- nastavíme tabulku deskriptorů (min. 0, kód, data)
- a nastavíme PE bit v CR0 registru

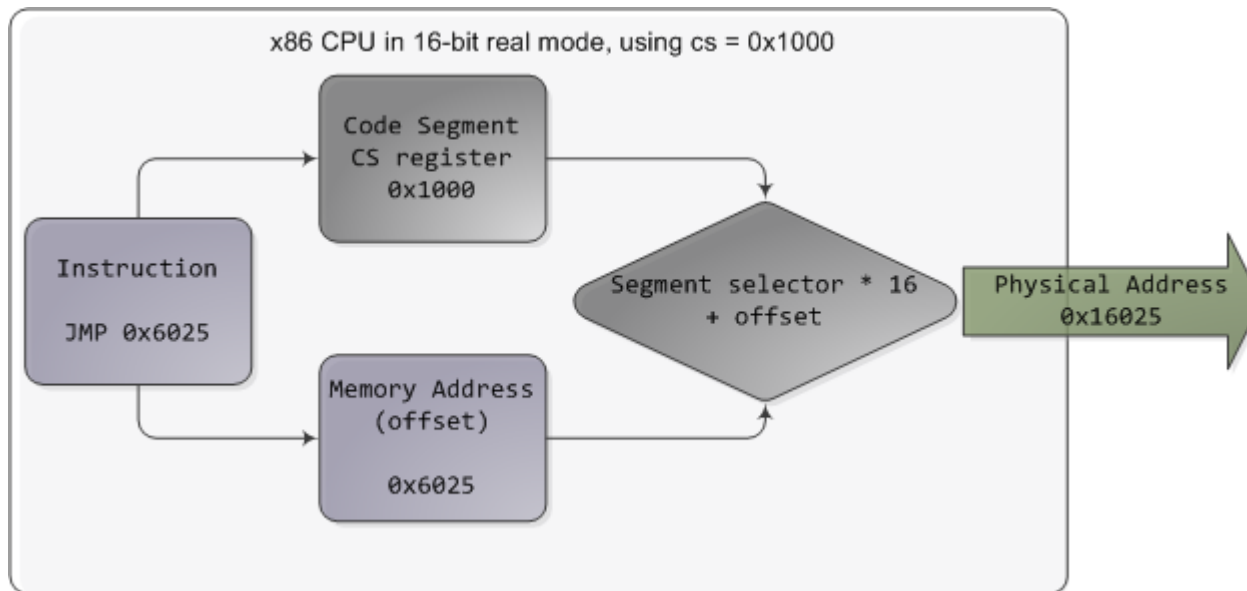
- virtual 8086 mode

- pro kompatibilitu

x86-64 procesory používají
long mode

Více viz studijní materiál v
coursewaru

MS-DOS a práce se segmenty v reálném módu



Obsah segmentového registru * 16 + ofset, nic víc

Obrázek: <https://manybutfinite.com/post/memory-translation-and-segmentation/>

Procesory & přerušení

- reálný mód

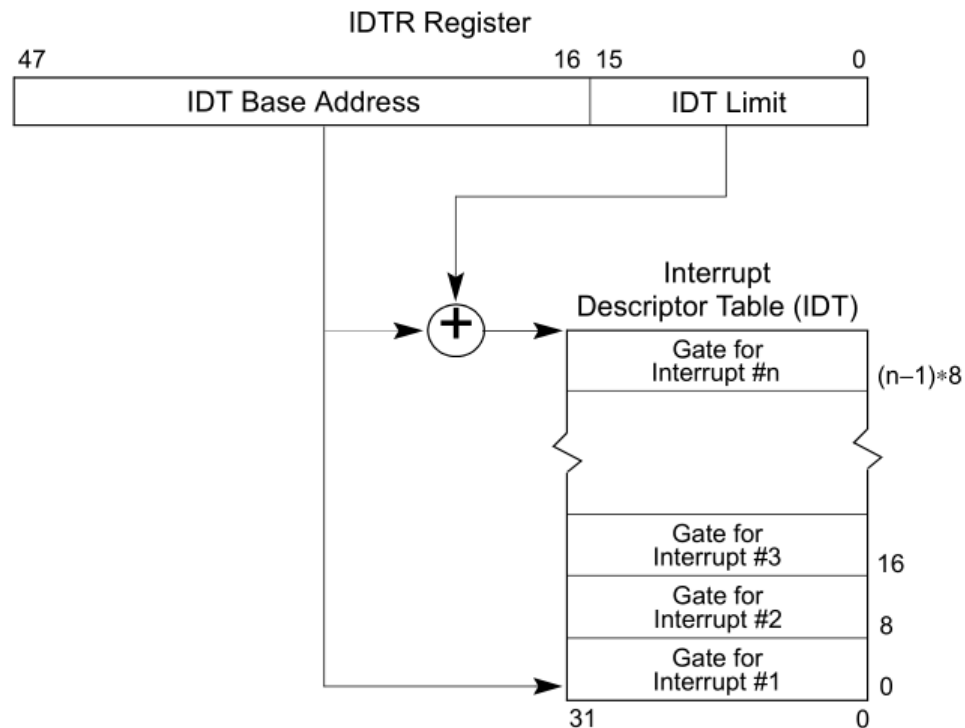
- První kilobyte (0..1023) RAM
 - interrupt vector table (256x4B)

- chráněný mód

- IDT (Interrupt Descriptor Table)
- pole 8bytových deskriptorů (indexovaných přerušovacím vektorem)
- naplněná IDT tabulka 2KB (256x8B)
- Umístění tabulky je v registru **IDTR**

Podrobně: http://wiki.osdev.org/Interrupt_Descriptor_Table

IDTR registr



Určuje, **kde**
tabulka vektorů
přerušení **začíná**
(báze)

i její **velikost**
Báze+limit =konec
tabulky

Protected mode – další pojmy

■ call gates

- volání předdefinované funkce přes [CALL FAR selector_callgate](#)
- volání funkce s vyšší privilegií oprávnění (uživatelský kód může zavolat kód jádra)
- spíše se používá SYSENTER/SYSEXIT - rychlejší

■ task state segment (TSS)

- pro přepínání tasků
- může obsahovat hodnoty x86 registrů
- Win/Linux naproti tomu používají softwarový task switching, řeší si po svém, většinou chtějí udělat i něco navíc

Poznámky - Brandejs

5.5.4 Předání řízení do instrukčního segmentu pomocí brány

Brány (Gate, Call Gate) lze použít při volání podprogramu umístěného v segmentu s vyšší úrovní oprávnění, než jakou má volající segment. Volání podprogramu pomocí brány lze uskutečnit jenom tehdy, platí-li numerický vztah $CPL \geq DPL$ volaného podprogramu. Z toho vyplývá, že v žádném případě nelze předávat řízení do segmentu majícího nižší úroveň oprávnění než volající (tedy ani pomocí brány).

Tato podmínka musí být dodržena proto, aby např. jádro operačního systému případným voláním podprogramu nižší úrovně oprávnění, do něhož mohlo být neautorizovaně zasáhnuto, nezpůsobilo zhroucení celého systému.

Vyšší číslo = nižší oprávnění

Jádro 0, program v uživatelském režimu 3

Uživatelský kód $CPL = 3$ branou mohl volat kód jádra $DPL = 0$

Bránilo se tomu, aby jádro $CPL = 0$ nevolalo kód uživatelský ($DPL = 3$), který mohl být pochybný 😊

CPL

DPL

code/descriptor

priviledge level

Poznámky - Brandejs

5.5.5 Brány

Brána¹ (Gate) je popisovač uložený v tabulce popisovačů segmentů čtyř možných významů:

1. brána pro předání řízení do segmentu vyšší úrovně oprávnění (**Call Gate**),
2. brána pro nemaskující přerušení (**Trap Gate**),
3. brána pro maskující přerušení (**Interrupt Gate**) – obě popsány na str. 92,
4. brána zpřístupňující segment stavu procesu (**Task Gate**) – popsána na str. 85.

Interrupt Gate zakazuje přerušení (IF flag v EFLAGS), TrapGate ne
Interrupty na HW události

TrapGate – reakce na CPU instrukce,
např. INT 3 – breakpoint pro debugger

Poznámky - Brandejs

5.6.1 Segment stavu procesu a registr TR

O **každém procesu** (aktivním i neaktivním) jsou v paměti ve speciálním segmentu uloženy **všechny potřebné informace**. Tento segment se nazývá **segment stavu procesu** (TSS – Task State Segment).

Stav procesu je dán obsahem všech registrů procesoru. Je-li proces momentálně neaktivní, je jeho stav zapsán v TSS tak, aby po aktivaci byly procesu zajištěny stejné podmínky, jaké měl před přerušением činnosti. Každý popisovač TSS je vlastně popisovačem systémového segmentu a smí být uložen pouze v GDT. Adresa popisovače TSS právě aktivního procesu je uložena ve speciálním registru **TR** (Task Register) tak, jak je to uvedeno na obr. 5.25. Na každý TSS ukazuje právě jeden popisovač TSS (TSS není reentrantní).

Přepínání tasků si většina OS řeší softwarově a nespolehá se na task gates – většinou chce OS udělat i něco navíc

poznámky

Podrobný a názorný popis

<https://stackoverflow.com/questions/3425085/the-difference-between-call-gate-interrupt-gate-trap-gate>

PAE

PAE is the ability of x86 to use **36 address bits instead of 32**. This increases the available memory from **4GB to 64GB**. The 32-bit applications still see only a 4GB address space, but the OS can map (via paging) memory from the high area to the lower 4GB address space. This extension was added to x86 to cope with the (nowadays not enough) limit of 4GB, before 64-bit CPUs came to the foreground.

Proces stále vidí max. 4GB, ale operační paměť už může být 64GB. Překlenout období, kdy může mít systém více RAM, ale 64bitové CPU ještě nebylo tolik rozšířené.

Zdroje:

<https://www.codeproject.com/Articles/1273844/The-Intel-Assembly-Manual-3>

https://cs.wikipedia.org/wiki/Physical_Address_Extension

Pamatuj !

- Běžný procesor v PC může běžet v reálném nebo chráněném módu
- Po zapnutí napájení byl puštěn **reálný mód**, ten využíval např. MS-DOS – není zde však žádný mechanismus ochrany
- Dnešní systémy přepínají procesor ihned do **chráněného režimu** (ochrana segmentů uplatněním limitu, ochrana privilegovanosti kontrolou z jakého ringu je možné přistupovat)
- 64bitové CPU používají long mode

Chráněný režim - segmenty

- 1 GDT – může mít až 8192 segmentů
- můžeme mít i více LDT (každá z nich může mít opět 8192 segmentů) a použít je pro ochranu procesů
- některé systémy využívají jen GDT, a místo LDT zajišťují ochranu pomocí stránkování

Chráněný režim – adresy !!!!

VA(selektor,offset) =segmentace==> LA =stránkování==> FA

VA je virtuální adresa, **LA** lineární adresa, **FA** fyzická adresa
selektor určí odkaz do tabulky segmentů => **deskriptor** (v GDT nebo LDT)

selektor obsahuje mj. bázi a limit ; **LA = báze + offset**

segmentaci nejde vypnout, stránkování ano

zda je zapnuté stránkování - bit v řídicím registru procesoru (**CRO**)

je-li vypnuté stránkování, lineární adresa představuje fyzickou adresu

chce-li systém používat jen stránkování,

roztáhne segmenty přes celý adresní prostor (překrývají se)

Linux: využívá primárně stránkování, Windows: obojí (ale viz předchozí komentáře)

Poznámky - prepaging

Prepaging:

do paměti se zavádí chybějící stránka a stránky okolní

Pamatuj

- v C o paměť žádáme `malloc()` a máme ji uvolnit `free()`
- paměť nám přidělí knihovna – **alokátor paměti**, která spravuje volnou paměť
- pokud alokátor nemá volnou paměť k dispozici, **požádá operační** systém systémovým voláním o přidělení další části paměti (další stránky)

Alokace paměti pro procesy

- explicitní správa paměti
 - Programátor se musí postarat o uvolnění paměti
 - Nevýhoda – občas zapomene
- čítání referencí
 - Každý objekt má u sebe čítač referencí
- garbage collection
 - Pokročilé algoritmy

Čítání referencí

- Ke každému objektu přiřazen **čítač referencí** (vytvořen: 1)
- Někdo si uloží referenci na objekt – **zvýšení** čítače
- Reference mimo rozsah platnosti nebo přiřazena nová hodnota jinam – **snížení** čítače
- Čítač na nule – **uvolnění objektu** z paměti
- Nevýhoda – cyklus – dva ukazují na sebe ale nic dalšího na ně
- Nevýhoda - režie

Garbage collection (GC)

- automatická správa paměti
- Speciální algoritmus (Garbage Collector) vyhledává a uvolňuje úseky paměti, které již proces nevyužívá
- Zjišťování, které objekty jsou z kořene programu nedostupné, nevede na ně žádný živý ukazatel (reference)
- Nejprve se používalo čítání referencí, potom pokročilejší algoritmy

Garbage collection

- sledovací algoritmy

- Přeruší běh programu a vyhledávají dosažitelné objekty

- Algoritmus Mark and Sweep

- Všechny objekty – hodnota navštíven na FALSE
- Projde všechny objekty, kam se lze dostat
- Navštíveným nastaví navštíven na TRUE
- Na závěr – objekty s příznakem FALSE lze uvolnit z paměti
- Nevýhoda – přerušení běhu programu (nejde pro realtime systémy)
- Nevýhoda – když nechává živé objekty na místě – fragmentace paměti

- Kopírovací algoritmus

- Rozdělí haldy na dvě části (aktivní a neaktivní)
- Pokud se při alokaci nevejde do dané části haldy – provede se úklid
- Úklid – prohození aktivní a neaktivní části, do aktivní se kopírují živé objekty ze staré
- Dvojnásobná velikost haldy, potřeba kopírovat objekty, které přežijí úklid

Garbage Collection

- Generační algoritmy

- Paměť do několika částí (generací)
- Objekty vytvářeny v nejmladší, po dosažení stáří přesunuty do starší generace
- Pro každou generaci – úklid v různých časových intervalech, i různé algoritmy
- Nejmladší generace se zaplní – všechny dosažitelné v nejmladší zkopírovány do starší generace

GC - použití

- část běhového prostředí
- přídatná knihovna
- jazyk Java (i např. 4 druhy garbage collectorů, vše generační)
- Platforma .NET
 - [https://msdn.microsoft.com/en-us/library/ee787088\(v=vs.110\).aspx](https://msdn.microsoft.com/en-us/library/ee787088(v=vs.110).aspx)

.NET z předchozího odkazu:

There are three generations of objects on the heap:

- **Generation 0.** This is the youngest generation and contains short-lived objects. An example of a short-lived object is a temporary variable. Garbage collection occurs most frequently in this generation.

Newly allocated objects form a new generation of objects and are implicitly generation 0 collections, unless they are large objects, in which case they go on the large object heap in a generation 2 collection.

Most objects are reclaimed for garbage collection in generation 0 and do not survive to the next generation.

- **Generation 1.** This generation contains short-lived objects and serves as a buffer between short-lived objects and long-lived objects.
- **Generation 2.** This generation contains long-lived objects. An example of a long-lived object is an object in a server application that contains static data that is live for the duration of the process.

.NET z předchozího odkazu:

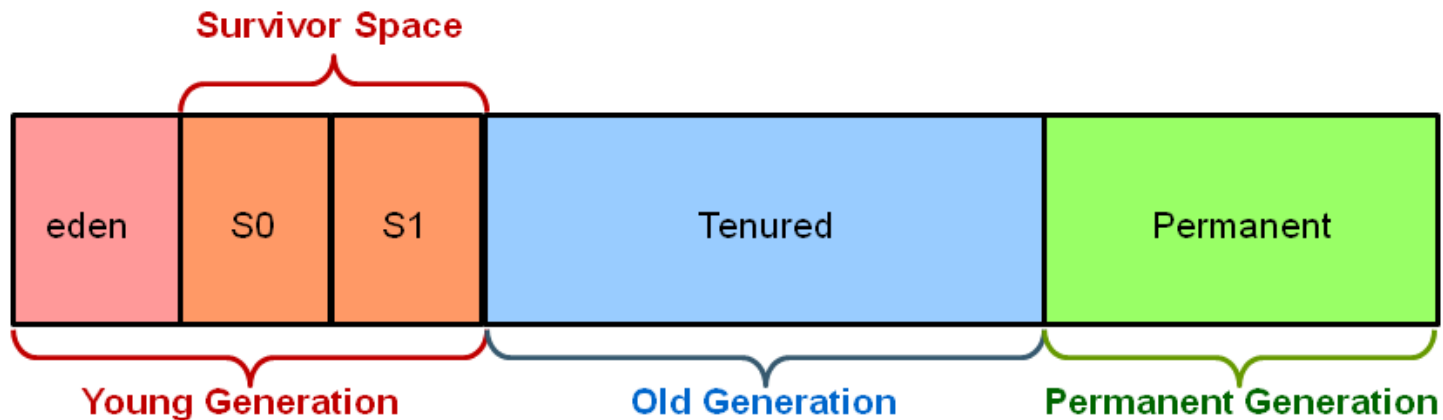
Survival and promotions

Objects that are not reclaimed in a garbage collection are known as **survivors**, and are promoted to **the next generation**. Objects that survive a generation **0** garbage collection are promoted to generation **1**; objects that survive a generation **1** garbage collection are promoted to generation **2**; and objects that survive a generation **2** garbage collection **remain in generation 2**.

When the garbage collector detects that the survival rate is high in a generation, it increases the threshold of allocations for that generation, so the next collection gets a substantial size of reclaimed memory. The CLR continually balances two priorities: not letting an application's working set get too big and not letting the garbage collection take too much time.

Java - GC

Hotspot Heap Structure



malloc, brk, sbrk

- malloc není systémové volání, ale knihovní funkce
viz `man 3 malloc` x `man 2 fork`
- malloc alokuje paměť z haldy
- velikost haldy se nastaví dle potřeby systémovým voláním `sbrk`
- `brk` – syscall – nastaví adresu konce datového segmentu procesu
- `sbrk` – syscall – zvětší velikost datového segmentu o zadaný počet bytů (0 – jen zjistím současnou adresu)

používané vs. nepoužívané objekty

```
Object x = new Foo();
```

```
Object y = new Bar();
```

```
x = new Quux();
```

```
/* víme, že Foo object původně přiřazený x nebude nikdy dostupný, jde o syntactic garbage */
```

```
if ( x.check_something() )
```

```
    { x.do_something(y); }
```

```
System.exit(0);
```

```
/* y může být semantic garbage, ale nevíme, dokud x.check_something() nevrátí návratovou hodnotu */
```


Velikost stránky v OS

- Standardní velikost je 4096 bytů (4KB)
- huge page size: 4MB
- large page size: 1GB

Zjištění velikosti stránky - Linux

```
#include <stdio.h>
#include <unistd.h>

int main(void) {

    printf("Velikost stranky je %ld bytu.\n",
        sysconf(_SC_PAGESIZE) );

    return 0;

}
```

příkazem na konzoli:
`getconf PAGESIZE`

`man sysconf`

eryx.zcu.cz: 4096
ares.fav.zcu.cz: 4096

Zjištění velikosti stránky - WIN

```
#include "stdafx.h"
```

```
#include <stdio.h>
```

```
#include <Windows.h>
```

```
int _tmain(int argc, _TCHAR* argv[]) {
```

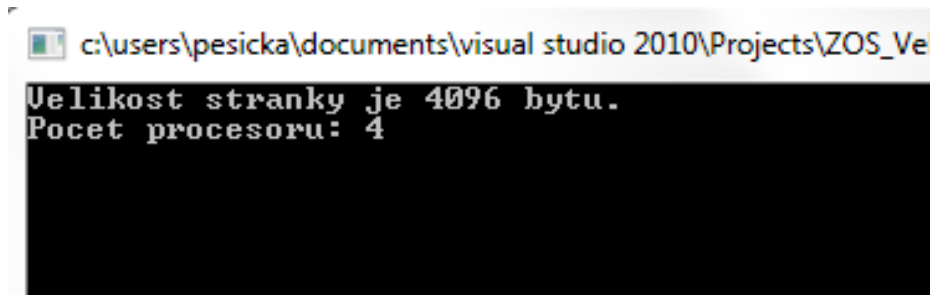
```
    SYSTEM_INFO si;
```

```
    GetSystemInfo(&si);
```

```
    printf("Velikost stranky je %u bytu.\n", si.dwPageSize);
```

```
    printf("Pocet procesoru: %u\n", si.dwNumberOfProcessors);
```

```
    getchar(); return 0; }
```



```
c:\users\pesicka\documents\visual studio 2010\Projects\ZOS_Ve
Velikost stranky je 4096 bytu.
Pocet procesoru: 4
```

A large orange scroll graphic with a vertical strip on the left and a horizontal strip at the top, framing the text.

Následuje směs dalších důležitých věcí z hlediska OS

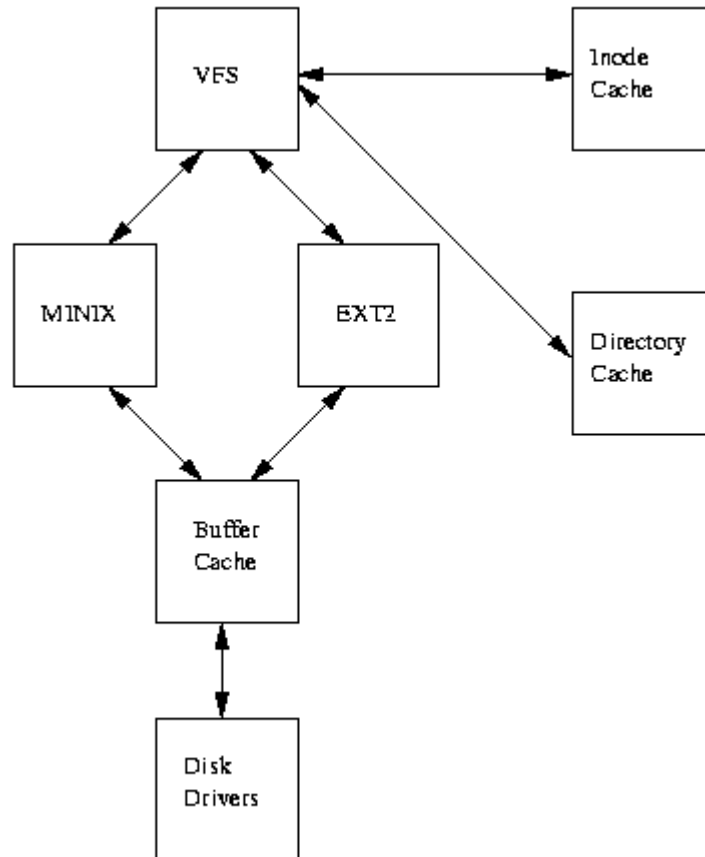
Amdahlův zákon

- Určuje urychlení výpočtu při užití více procesorů
- Urychlení je limitováno sekvenčními částmi výpočtu
tj. u 2 CPU není urychlení 2x, ale pouze část kódu lze provést paralelně

Citace z materiálů prof Ježka:

Když se něco dělá paralelně, může se získat výsledek rychleji, ale ne vždy.
(1 žena porodí dítě za 9 měsíců, ale 9 žen neporodí dítě za 1 měsíc)

VFS



Všimněte si různých cachí, které slouží k urychlení práce operačního systému

Škodlivé programy

- malicious software, malware
- mohou vyžadovat hostitelský program nebo jsou nezávislé
- **Bacteria**
 - Konzumuje systémové zdroje tím, že se replikuje
- **Logic bomb**
 - Pokud v systému nastanou určité podmínky, provede se nějaká škodlivá akce
- **Trapdoor / Backdoor**
 - Tajný nedokumentovaný vstupní bod do programu, obejdou se běžné přístupové kontroly

Škodlivé programy

■ Trojan Horse

- Tajná nedokumentovaná rutina obsažená v jinak užitečném programu. Při spuštění programu dojde i k vykonání tohoto kódu.

■ Virus

- Kód zapouzdřený v programu, kopíruje se a vkládá do dalších programů. Kromě vlastního šíření vykonává různé škodlivé činnosti.

■ Worm (červ)

- Program může replikovat sebe a šířit se z počítače na počítač přes síť. Na cílovém stroji ve své činnosti pokračuje. Může vykonávat další škodlivé činnosti.

Škodlivé programy

■ Ransomware

- Zablokuje počítačový systém nebo šifruje zapsaná data a pak požaduje od oběti výkupné za obnovení přístupu k nim.

■ Adware

- Změna chování systému
- Webové prohlížeče odkazují na jiné stránky, mění domovské stránky, obtěžuje reklamou

■ Spyware

- Špionážní software
- Čísla účtů, přístup k bankovníctví, seznam nainstalovaných programů

Projevy škodlivého kódu

- Mazání, zašifrování dat
- DOS (denial of service) – odmítnutí služby
 - Např. přetížení serveru požadavky
- Rozesílání Spamu
- Vzdálené ovládání, krádež informací
- Posílání drahých textovek, hovory (mobilní zařízení)

Antiviry

- Detekce
 - Zjistit, že je soubor napadený
- Identifikace
 - Zjistit, o jaký virus se jedná
- Odstranění
- Databáze signatur virů (aktualizace)
- Heuristika

Bezpečnostní chyby - příklad

- ukázka staré chyby v Unixu
- Příkaz `lpr` vytiskne zadaný soubor na tiskárnu
- `lpr -r` navíc soubor po vytištění zruší
- Ve starších verzích Unixu bylo možné, aby kdokoliv vytisknul a zrušil `/etc/passwd`

Přetečení bufferu uloženého na zásobníku

- Programy vytvořené v C
- Nekontroluje např. meze polí, umožňuje přepsat část paměti
- Volání procedury – na zásobník uloží návratovou adresu
- Spustí proceduru – vytvoří místo pro lokální proměnné
- Uživatel zadá delší řetězec, přepíše část další paměti, včetně návratové adresy
- Návrat – instrukce RET – začne vykonávat instrukce od jiné adresy (pokud náhodné, program většinou zhavaruje)

Virtualizace

- Možnost provozovat více virtuálních strojů na jednom fyzickém stroji

Výhody:

- Snadnější údržba
- Zálohování, snapshoty strojů
- Neovlivňují se navzájem

Často možnost migrace (i za běhu) na jiný fyzický stroj, aby bylo možné provést jeho údržbu.

Příklady virtualizace

- VirtualBox
 - Na uživatelském PC možnost spustit další virtuální stroj
- Hyper-V
 - Virtualizační technologie od Microsoftu
- XEN, KVM
 - Virtualizace používaná na serverech pro běh virtuálních serverů

Ověřování uživatelů

- Autentizace (authentication)
 - Jednoznačné prokázání totožnosti (identity) uživatele
 - **Identifikace** – vůbec vědět, kdo je („řekni své jméno“)
 - **Autentizace** – ověření, zda je opravdu tím, za koho se vydává
-
- Uživatel zadá jméno (identifikuje se), po zadání hesla dojde k jeho autentifikaci – systém ověří, zda je opravdu tím, za koho se vydává

Metody autentikace

- Uživatel něco **zná**
 - Heslo
 - Transformační funkce (výzva – odpověď, např. $y = 2 * x + 7$)
- Uživatel něco **má**
 - Kalkulátor pro jednorázová hesla
- Uživatel má určité **biometrické vlastnosti**
 - Otisky prstů
 - Sítnice oka
 - FaceID
 - Spolehlivost ?

Autorizace

- Řízení přístupu ke zdrojům systému
- Např. zda máme právo nějaký soubor číst, zapisovat do něj...
- Systému musíme nejprve prokázat kdo jsme (autentizace) a systém dále ověří, zda pro danou činnost máme právo (autorizace)