

# 10.

## Memory management II.

---

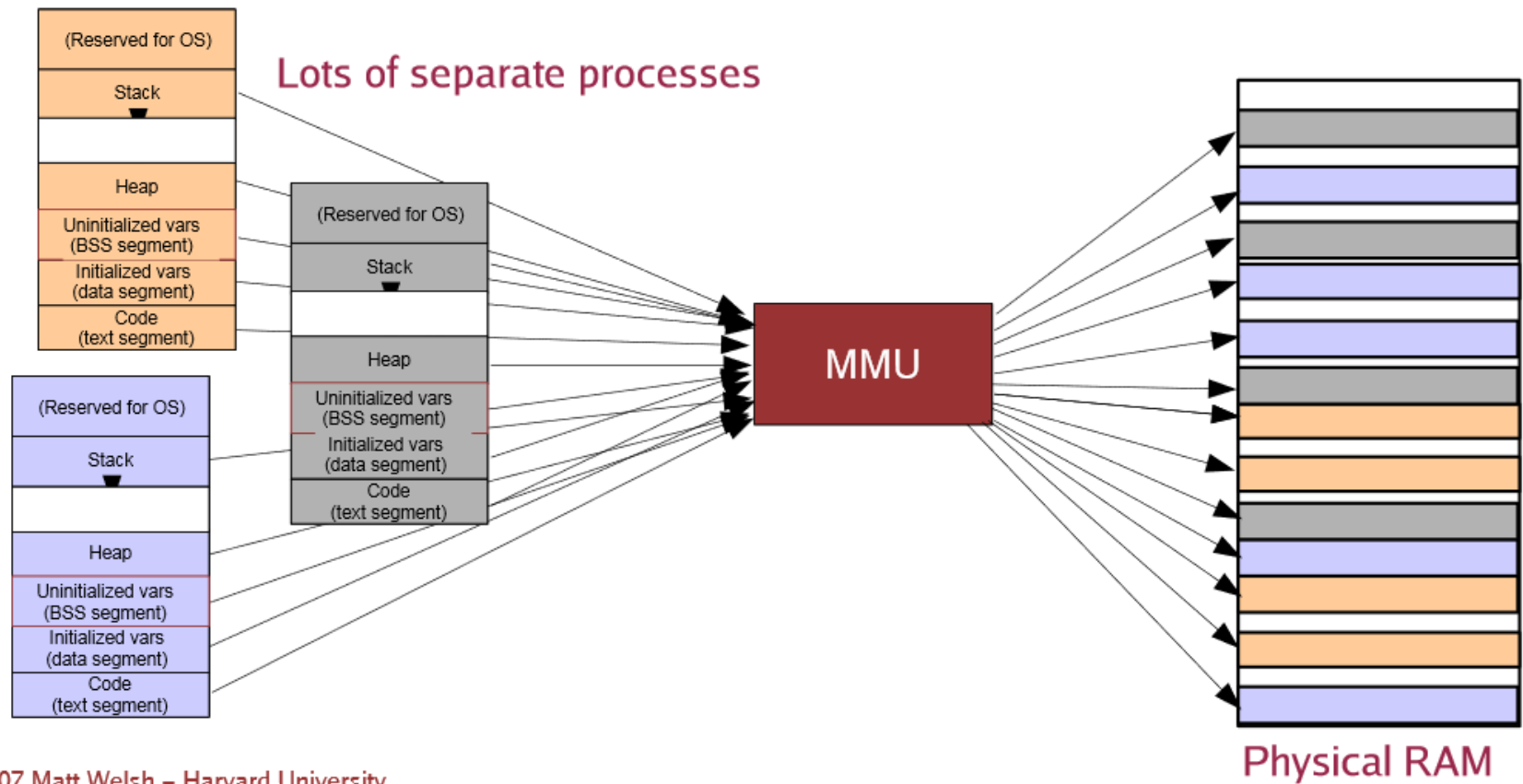
ZOS 2016, L. PEŠIČKA

# Stránkování

---

- Každý proces má k dispozici virtuální adresní prostor v rozsahu  $0 \dots 2^p - 1$  bytů  
( $p$  .. počet bitů ukazatele, např. 32 bitů)
- Mapování virtuální stránky -> fyzické rámce  
je pro proces transparentní

# Rozptýlení v RAM



# Stránkovaná paměť

Swap na disku



P1

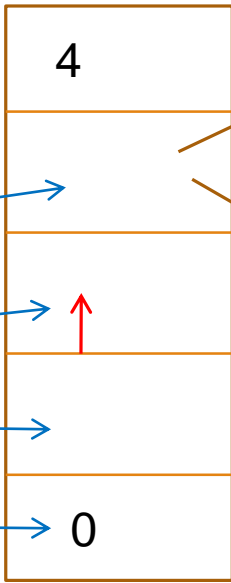
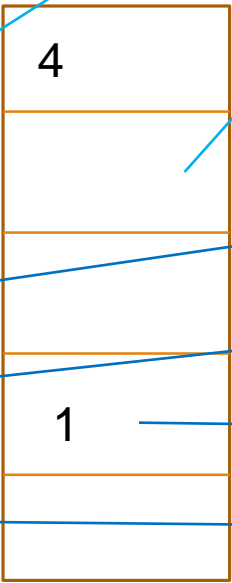
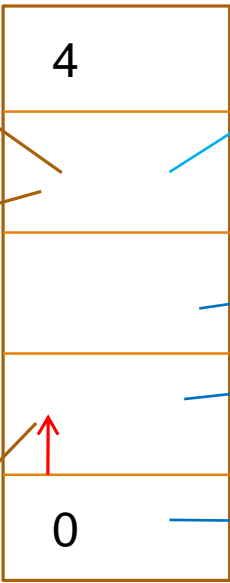
P2

RAM

virtuální adresy

stránka např. 4KB

Offset od začátku stránky



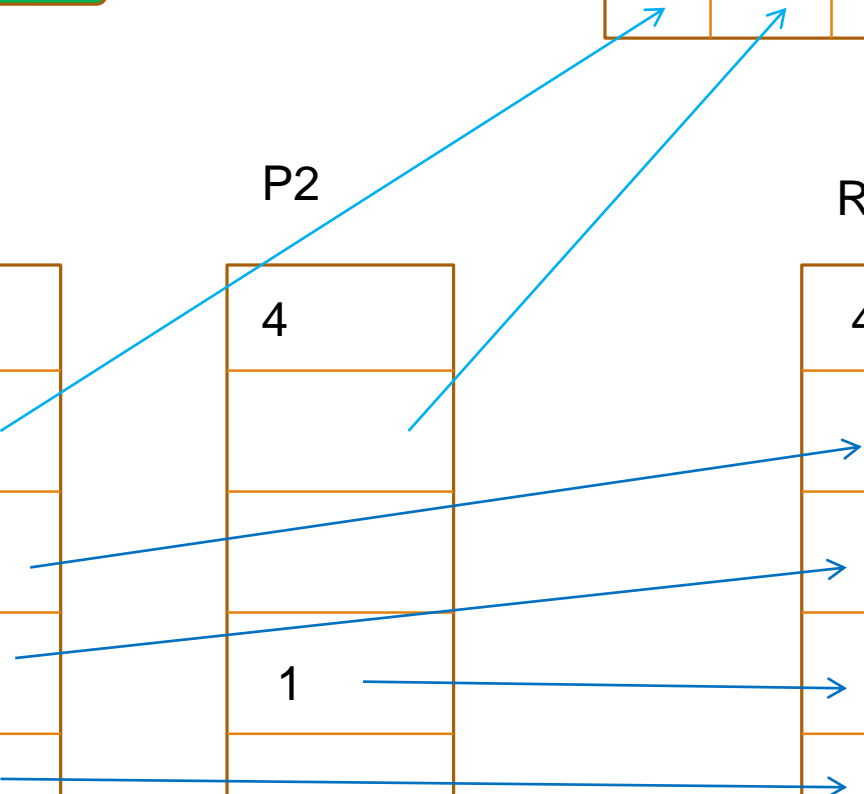
fyzické adresy

rámec stejné velikosti jako stránka

Tabulka stránek Procesu 1

Tabulka stránek Procesu 2

Tabulka rámců



Tabulka stránek procesu: 1  
Velikost stránky: 4096 B

stránka	rámec	další atributy
0	0	
1	2	
2	3	
3	x	swap: 0
4		

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 ] = 2 .. druhý rámec  
 $FA = 2 * 4096 + 3 = 8195$

Výpadek stránky:

Stránka není v operační paměti, ale ve swapu na disku

# Tabulka stránek – podrobněji (!!!!)

Index stránky	Číslo rámce (20 bitů)	příznak platnosti	Příznaky ochrany	Bit modifikace (dirty)	Bit referenced	Adresa ve swapu
0	3	valid	rx	1	1	---
1	4	valid	rw	1	1	---
2	---	invalid	ro	0	0	4096

valid  
invalid

rw, rx, ro,...

zda je třeba rámec uložit  
do swapu při odstranění z  
RAM

zda byla stránka  
přístupována (čtení či  
zápis) v poslední době

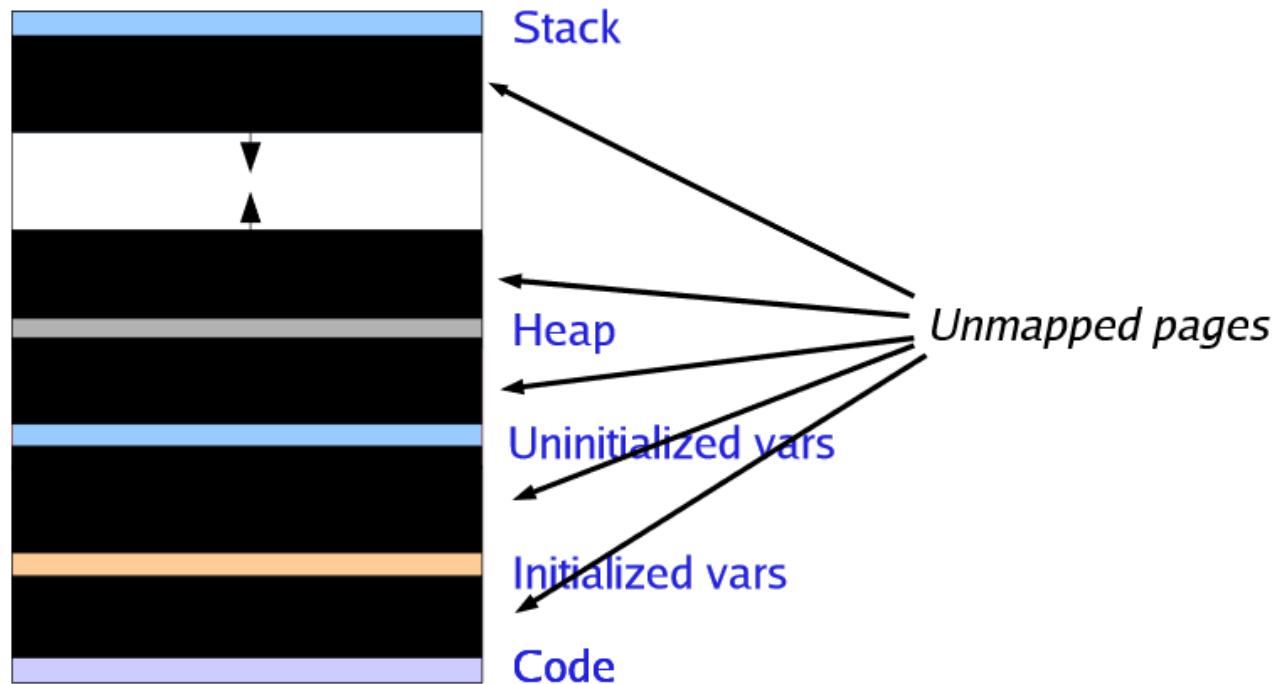
# Tabulka stránek (TS) - podrobněji

---

- součástí PCB (tabulka procesů) – kde leží jeho TS
- velikost záznamu v TS .. 32 bitů
- číslo rámce .. 20 bitů
  
- dvouúrovňová tabulka stránek
  - 4KB, 4MB
- čtyřúrovňová tabulka stránek x86-64
  - stránky 4KB, 2MB, až 1GB

# Spuštění procesu

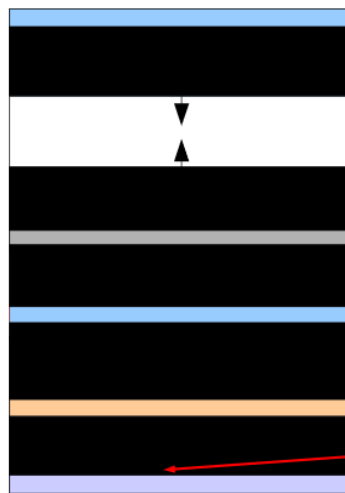
---



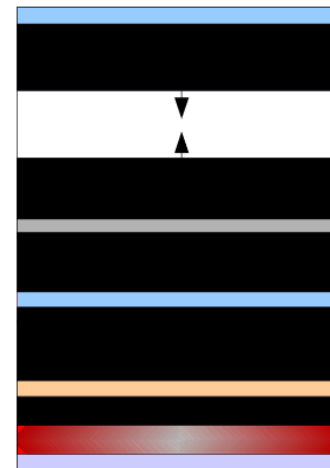


# Spuštění procesu

---



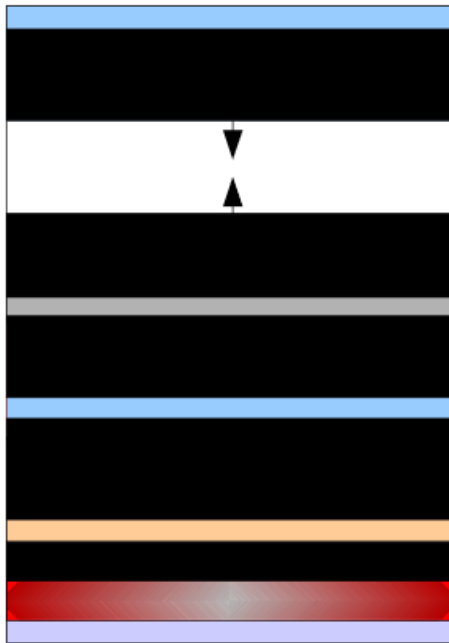
*Reference next instruction*



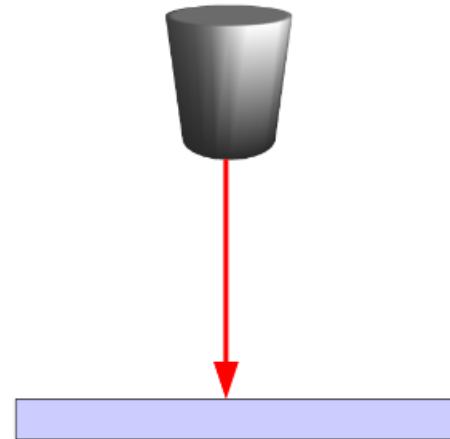
*Page fault!!!*

# Spuštění procesu

---



*OS reads missing page  
from executable file on  
disk*



# Sdílení paměti mezi procesy

---

- 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

# Obsah

---

FIFO + Beladyho anom.

MIN / OPT

LRU

NRU

Second Chance, Clock

Aging

-----

Segmentování

I/O

## Algoritmy nahrazování stránek paměti

Použijí se, pokud potřebujeme uvolnit místo v operační paměti pro další stránku:

nastal výpadek stránky, je třeba někam do RAM zavést stránku a RAM je plná..

nějakou stránku musíme z RAM odstranit, ale jakou?

# Algoritmus MIN / OPT

---

- optimální – **nejmenší možný** výpadek stránek
- Vyhodíme zboží, které nejdelší dobu nikdo nebude požadovat.
- stránka označena počtem instrukcí, po který se k ní nebude přistupovat
- $p[0] = 5$ ,  $p[1] = 20$ ,  $p[3] = 100$
- výpadek stránky – vybere s nejvyšším označením
- vybere se stránka, která bude zapotřebí nejpozději v **budoucnosti**

# MIN / OPT

---

- **není realizovatelný** (křišťálová koule)
  - jak bychom zjistili dopředu která stránka bude potřeba?
- algoritmus pouze pro **srovnání** s realizovatelnými
- Použití pro běh programu v **simulátoru**
  - uchovávají se odkazy na stránky
  - spočte se počet výpadků pro MIN/OPT
  - Srovnání s jiným algoritmem (o kolik je jiný horší)

# Least Recently Used (LRU)

---

- **nejdéle nepoužitá** (pohled do minulosti)
- princip lokality
  - stránky používané v posledních instrukcích se budou pravděpodobně používat i v následujících
  - pokud se stránka dlouho nepoužívala, pravděpodobně nebude brzy zapotřebí
- **Vyhazovat zboží, na kterém je v prodejně nejvíce prachu = nejdéle nebylo požadováno**

# LRU

---

- obtížná implementace
- **sw řešení (není použitelné)**
  - seznam stránek v **pořadí referencí**
  - výpadek – vyhození stránky ze začátku seznamu
  - zpomalení cca 10x, nutná podpora hw



# LRU – HW řešení - čítač

---

## ■ HW řešení – čítač

- MMU obsahuje čítač (64bit), při každém přístupu do paměti zvětšen
- každá položka v tabulce stránek – pole pro uložení čítače
- odkaz do paměti:
  - obsah čítače se zapíše do položky pro odkazovanou stránku
- výpadek stránky
  - vyhodí se stránka s nejnižším číslem

## LRU – HW řešení - matice

---

- MMU udržuje **matici  $n * n$  bitů**
  - $n$  – počet rámců
- všechny prvky **0**
- **odkaz na stránku** odpovídající  $k$ -tému rámcu
  - všechny bity  $k$ -tého **řádku** matice na **1**
  - všechny bity  $k$ -tého **sloupce** matice na **0**
- **řádek** s nejnižší binární hodnotou
  - nejdéle nepoužitá stránka

## LRU – matice - příklad

---

reference v pořadí: 3 2 1 0

0.1.2.3						0.1.2.3						0.1.2.3						0.1.2.3					
0.	0	0	0	0	0	0.	0	0	0	0	0	0.	0	0	0	0	0	0.	0	0	0	0	0
1.	0	0	0	0	0	1.	0	0	0	0	0	1.	1	0	1	1	1	1.	0	0	1	1	1
2.	0	0	0	0	0	2.	1	1	0	1	1	2.	1	0	0	1	1	2.	0	0	0	1	1
3.	1	1	1	1	0	3.	1	1	0	0	0	3.	1	0	0	0	0	3.	0	0	0	0	0

# LRU - vlastnosti

---

## ■ výhody

- z časově založených (realizovatelných) nejlepší
- Beladyho anomálie nemůže nastat

## ■ nevýhody

- každý odkaz na stránku – aktualizace záznamu (zpomalení)
  - položka v tab. stránek
  - řádek a sloupec v matici
- LRU se pro stránkovanou virtuální paměť příliš nepoužívá
- LRU ale např. pro blokovou cache souborů

# Not-Recently-Used (NRU)

---

- snaha vyhazovat **nepoužívané stránky**
- HW podpora:
  - **stavové bity Referenced (R)** a **Dirty (M = modified)**
  - v tabulce stránek
- bity **nastavované HW** dle způsobu přístupu ke stránce
- **bit R** – nastaven na 1 při **čtení** nebo **zápisu** do stránky
  - pravidelně nulován (aby označoval referenci v poslední době)
- **bit M** – na 1 při **zápisu** do stránky
  - stránku je třeba při vyhození zapsat na disk
  - bit **zůstane** na 1, dokud ho SW nenastaví zpět na 0

# algorithmus NRU

---

- začátek – všechny stránky  $R=0$ ,  $M=0$
- bit **R** nastavován OS **periodicky** na 0 (přerušení čas.)
  - odliší stránky referencované **v poslední době !!**
- 4 kategorie stránek (R,M)
  - třída 0:  $R = 0$ ,  $M = 0$*
  - třída 1:  $R = 0$ ,  $M = 1$  -- z třídy 3 po nulování R*
  - třída 2:  $R = 1$ ,  $M = 0$*
  - třída 3:  $R = 1$ ,  $M = 1$*
- NRU vyhodí **stránku z nejnižší neprázdné třídy**
- výběr mezi stránkami ve stejné třídě je **náhodný**

# NRU

---

- pro NRU platí – lepší je **vyhodit modifikovanou** stránku, která **nebyla použita** 1 tik, než nemodifikovanou stránku, která se právě používá
- **výhody**
  - jednoduchost, srozumitelnost
  - efektivně implementovaný
- **nevýhody**
  - výkonost (jsou i lepší algoritmy)

# Náhrada bitů R a M - úvaha

---

jak by šlo simulovat R,M bez HW podpory?

- start procesu – všechny stránky jako nepřítomné v paměti
- odkaz na stránku – výpadek
  - OS interně nastaví R=1
  - nastaví mapování stránky v READ ONLY režimu
- pokus o zápis do stránky – výjimka
  - OS zachytí a nastaví M=1,
  - změní přístup na READ WRITE



# Algoritmy Second Chance a Clock

---

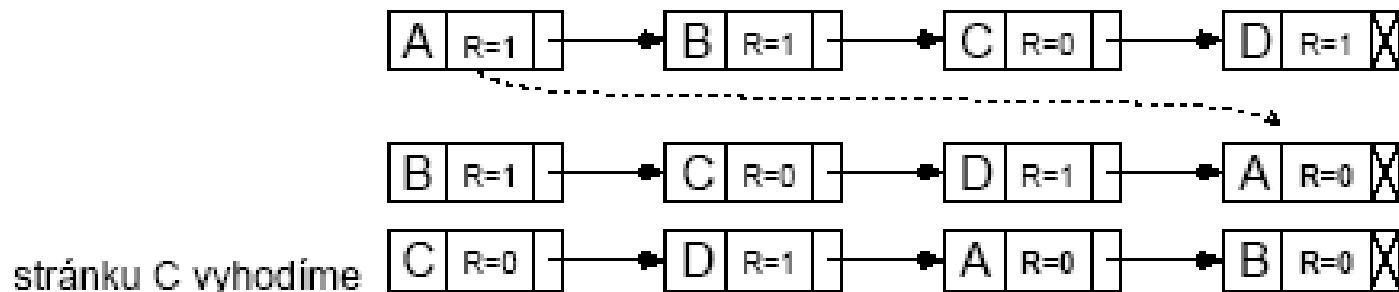
- vycházejí z FIFO
  - **FIFO** – obchod vyhazuje zboží zavedené před nejdelší dobou, ať už ho někdo chce nebo ne
  - **Second Chance** – evidovat, jestli zboží v poslední době někdo koupil (ano – prohlásíme za čerstvé zboží)
- modifikace FIFO – zabránit vyhození často používané

# Second Chance

---

- algoritmus Second Chance
- dle bitu R (referenced) nejstarší stránky
  - $R = 0$  ... stránka je nejstarší, nepoužívaná – vyhodíme
  - $R = 1$  ... nastavíme  $R=0$ , přesuneme na konec seznamu stránek (jako by byla nově zavedena)

## Příklad Second Chance



1. Krok – nejstarší je A, má  $R = 1$  – nastavíme  $R$  na 0 a přesuneme na konec seznamu
2. Druhá nejstarší je B, má  $R = 1$  – nastavíme  $R$  na 0 a opět přesuneme na konec seznamu
3. Další nejstarší je C,  $R = 0$  – vyhodíme ji

# Second Chance

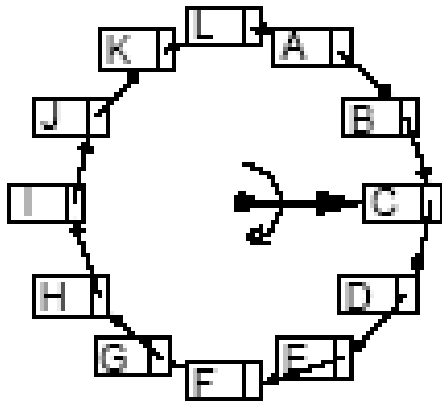
---

- SC vyhledá nejstarší stránku, která nebyla referencována v poslední době
- Pokud všechny referencovány – čisté FIFO
  - Všem se postupně nastaví R na 0 a na konec seznamu
  - Dostaneme se opět na A, nyní s  $R = 0$ , vyhodíme ji
- Algoritmus končí nejvýše po (počet rámců + 1) krocích

# Algoritmus Clock

Optimalizace datových struktur algoritmu Second Chance

- Stránky udržovány v **kruhovém** seznamu
- Ukazatel na nejstarší stránku – „ručička hodin“



Výpadek stránky – najít stránku k vyhození

Stránka kam ukazuje ručička

- má-li **R=0**, **stránku vyhodíme** a ručičku posuneme o jednu pozici
- má-li **R=1**, **nastavíme R na 0**, ručičku posuneme o 1 pozici, opakování,..

Od SC se liší pouze implementací

Varianty Clock používají např. BSD UNIX

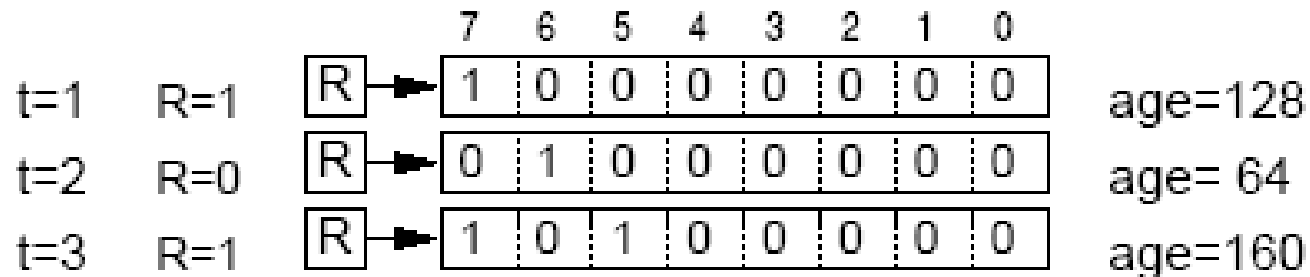
# SW aproximace LRU - Aging

---

- LRU vyhazuje vždy nejdéle nepoužitou stránku
- Algoritmus **Aging**
  - Každá položka tabulky stránek – pole stáří (age), N bitů (8)
  - Na počátku age = 0
  - Při každém **přerušení časovače** pro **každou** stránku:
    - Posun pole stáří o 1 bit vpravo
    - Zleva se přidá hodnota bitu R
    - Nastavení R na 0
- Při výpadku se vyhodí stránka, jejíž pole age má **nejnižší hodnotu**

# Aging

---



Age := age shr 1;                      posun o 1 bit vpravo

Age := age or (R shl N-1);      zleva se přidá hodnota bitu R

R := 0;                                  nastavení R na 0

# Aging x LRU

---

- Několik stránek může mít **stejnou hodnotu age** a nevíme, která byla odkazovaná **dříve** (u LRU jasné vždy) – hrubé rozlišení (po ticích časovače)
- Age se může snížit na 0
  - nevíme, zda odkazovaná před 9ti nebo 1000ci tiky časovače
    - Uchovává pouze **omezenou historii**
    - V praxi není problém – tik 20ms, N=8, nebyla odkazována 160ms – nejspíše není tak důležitá, můžeme jí vyhodit
- stránky se stejnou hodnotou age – vybereme **náhodně**



# Shrnutí algoritmů

---

- **Optimální algoritmus (MIN čili OPT)**

- Nelze implementovat, vhodný pro srovnání

- **FIFO**

- Vyhazuje nejstarší stránku
- Jednoduchý, ale je schopen vyhodit důležité stránky
- Trpí Beladyho anomálií

- **LRU (Least Recently Used)**

- Výborný
- Implementace vyžaduje spec. hardware, proto používán zřídka

důležité je  
uvědomit si,  
kdy tyto  
algoritmy  
zafungují –  
potřebujeme v  
RAM uvolnit  
rámec

# Shrnutí algoritmů II.

---

## ■ NRU (Not Recently Used)

- Rozděluje stránky do 4 kategorií dle bitů R a M
- Efektivita není příliš velká, přesto používán

## ■ Second Chance a Clock

- Vycházejí z FIFO, před vyhozením zkontrolují, zda se stránka používala
- Mnohem lepší než FIFO
- Používané algoritmy (některé varianty UNIXu)

## ■ Aging

- Dobře aproximuje LRU – efektivní
- Často prakticky používaný algoritmus

# Alokace fyzických rámců (!!)

---

- Alokace fyzických rámců

- Globální a lokální alokace

- Globální – pro vyhození se uvažují **všechny** rámce

- Lepší průchodnost systému – častější

- Na běh procesu má vliv chování ostatních procesů

- Lokální – uvažují se pouze **rámce alokované procesem** (tj. obsahující stránky procesu, jehož výpadek stránky se obsluhuje)

- Počet stránek alokovaných pro proces se nemění

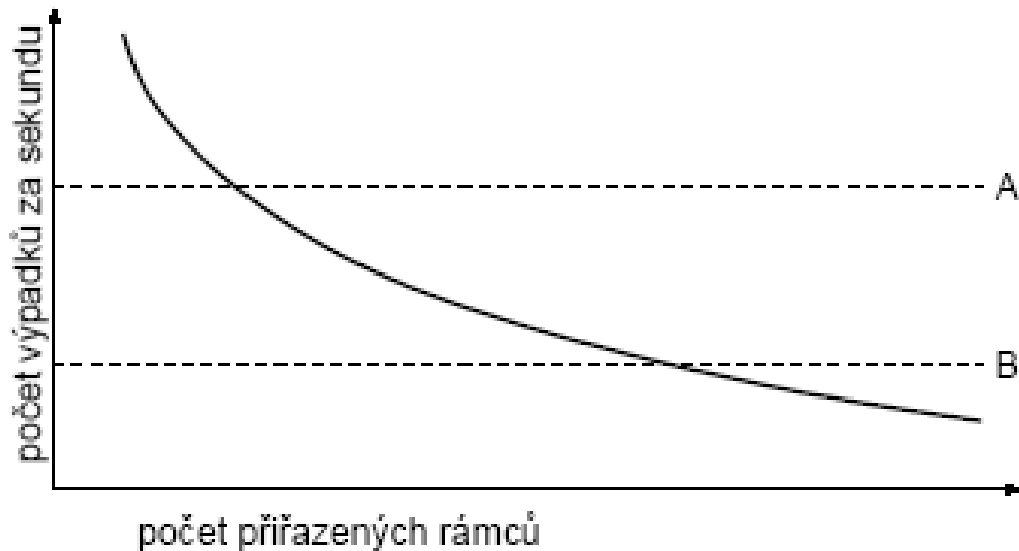
- Program se vzhledem k stránkování chová přibližně stejně při každém běhu

# Lokální alokace

---

- Kolik rámců dát každému procesu?
- **Nejjednodušší** – všem procesům dát stejně
  - Ale potřeby procesů jsou různé
- **Proporcionální** – každému proporcionální díl podle velikosti procesu
- **Nejllepší** – podle frekvence výpadků stránek za jednotku času (Page Fault Frequency, PFF) !!!!
  - Pro většinu rozumných algoritmů se PFF snižuje s množstvím přidělených rámců

# Page Fault Frequency (PFF)



PFF udržet v roz. mezích:

if  $PFF > A$

přidáme procesu rámce

if  $PFF < B$

proces má asi příliš paměti  
rámce mu mohou být

odebrány

PFF snadno změříme – nastává při výpadku stránky  
(výjimka – ošetření obsluhou přerušení)

# Zloděj stránek (page daemon)

---

- v systému se běžně udržuje určitý počet volných rámců
- když klesne pod určitou mez, pustí **page daemon - kswapd** (zloděj stránek), ten uvolní určité množství stránek (rámců)
- když se čerstvě uvolněné stránky hned nepřidělí, lze je v případě potřeby snadno vrátit příslušnému procesu

# Zamykání stránek

---

zabrání odložení stránky

- části jádra
- stránka, kde probíhá I/O
- tabulky stránek
- nastavení uživatelem – `mlock()` , viz man 2 `mlock`

# mlock

```
eryx.zcu.cz - PuTTY
MLOCK(2) Linux Programmer's Manual MLOCK(2)

NAME
    mlock, munlock, mlockall, munlockall - lock and unlock memory

SYNOPSIS
    #include <sys/mman.h>

    int mlock(const void *addr, size_t len);
    int munlock(const void *addr, size_t len);

    int mlockall(int flags);
    int munlockall(void);

DESCRIPTION
    mlock() and mlockall() respectively lock part or all of the calling
    process's virtual address space into RAM, preventing that memory from
    being paged to the swap area. munlock() and munlockall() perform the
    converse operation, respectively unlocking part or all of the calling
    process's virtual address space, so that pages in the specified virtual
    address range may once more to be swapped out if required by the kernel
    memory manager. Memory locking and unlocking are performed in units of
    whole pages.
```



# Zahlcení a pracovní množina stránek (!)

---

- Proces pro svůj rozumný běh potřebuje **pracovní množinu stránek**
- Pokus se pracovní množiny stránek aktivních procesů **nevejdou** do paměti, nastane **zahlcení** (trashing)
- **Zahlcení**
  - V procesu nastane **výpadek stránky**
  - Paměť je **plná** (není volný rámec) – je třeba nějakou stránku **vyhodit**, stránka pro vyhození bude ale brzo **zapotřebí**, bude se muset vyhodit jiná používaná stránka ...
- Uživatel pozoruje – systém intenzivně pracuje s diskem a běh procesů se řádově zpomalí (více času stránkování než běh)
- Řešení – při zahlcení snížit úroveň multiprogramování (zahlcení lze detekovat pomocí PFF)

# 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

# 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.

# Rozdělení paměti pro proces (!!!)

pokus.c:

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

```
void fce1() {
```

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

```
    ... }
```

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

```
    ...
```

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

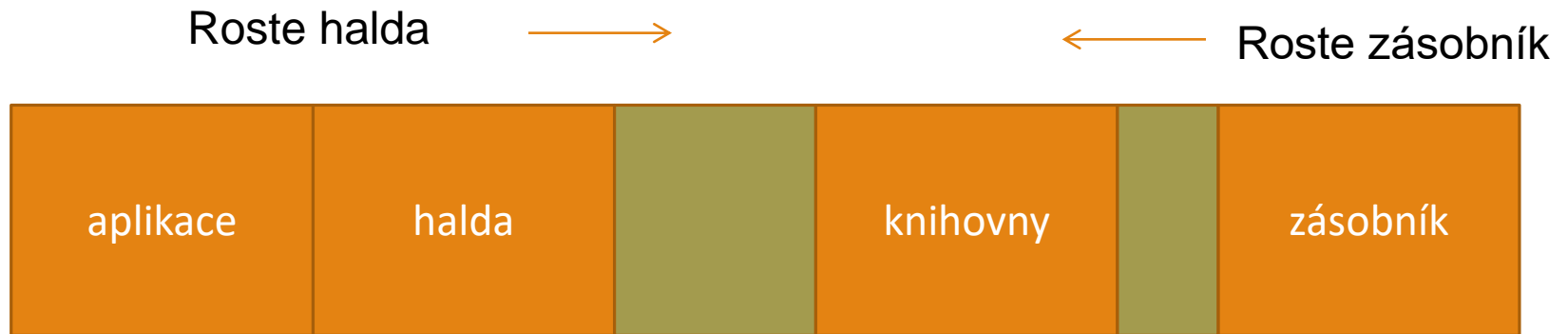
```
    fce1();
```

```
    return 0;
```

```
}
```



# Rozdělení paměti pro proces



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



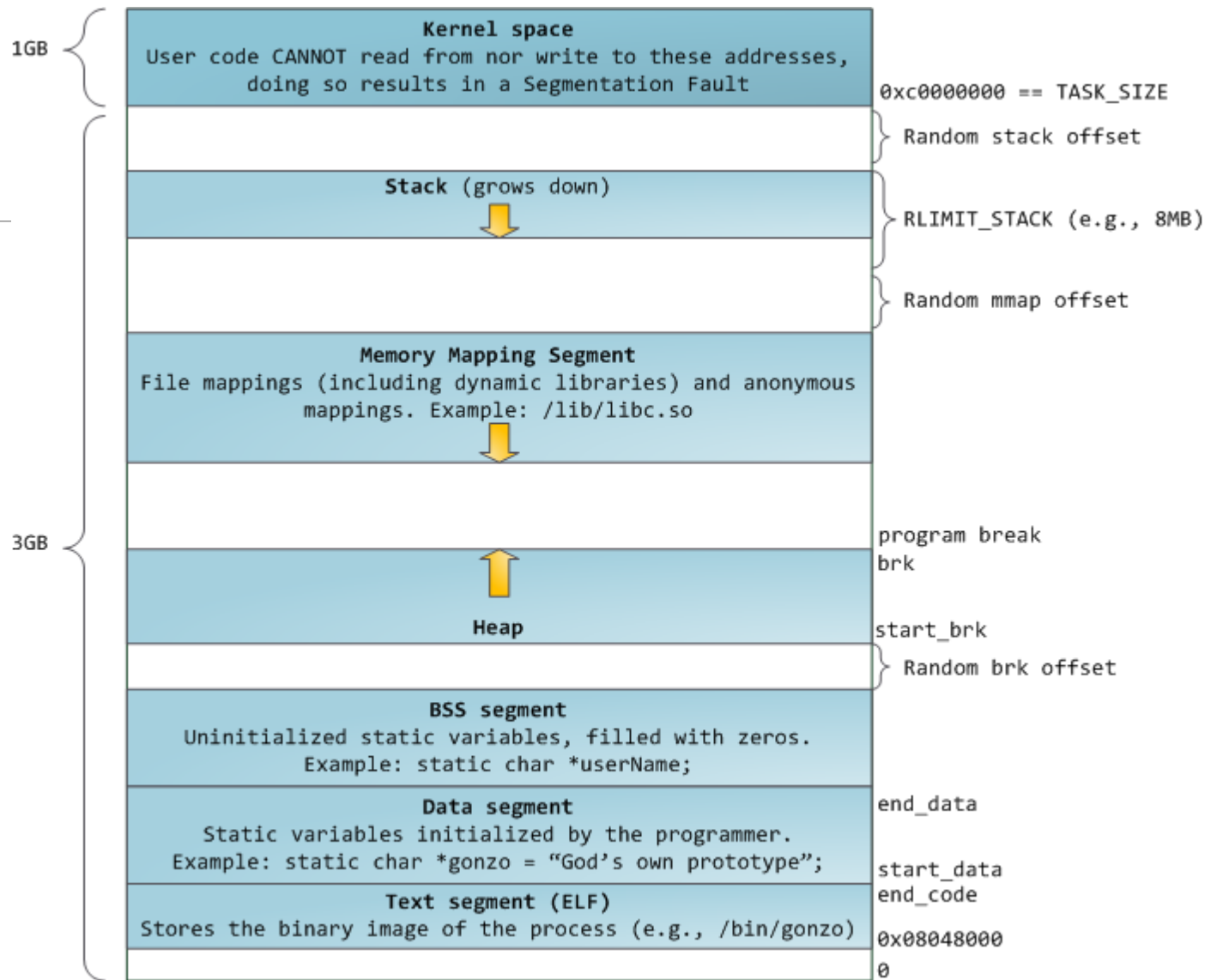
# poznámka

---

- 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

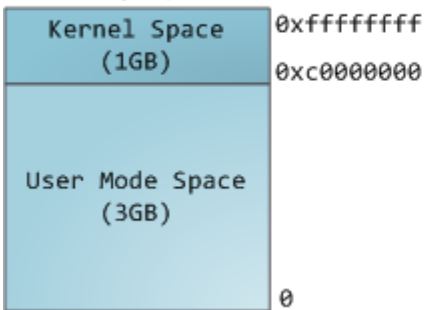
zapamatovat  
pojem  
BSS  
=  
neinicializova  
ná data



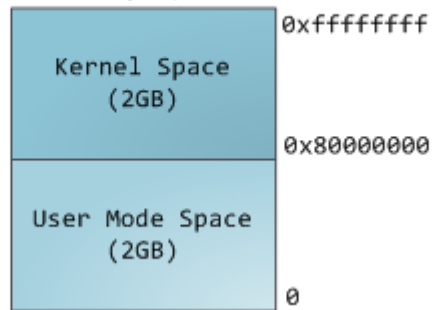


# Rozdělení paměti

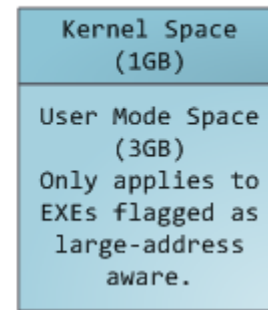
Linux User/Kernel  
Memory Split



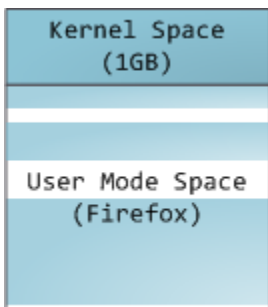
Windows, default  
memory split



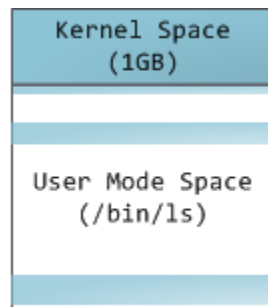
Windows booted  
with /3GB switch



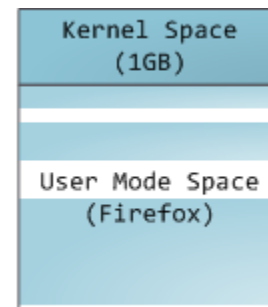
rozdělení  
paměti  
Linux,  
Windows



Process  
Switch  
→



Process  
Switch  
→



přepínání  
kontextu  
mezi  
různými  
procesy

# Segmentace

---

- Dosud diskutovaná VP – **jednorozměrná**
  - Proces: adresy  $< 0, \text{maximální virtuální adresa}>$
- Často výhodnější – **více samostatných virtuálních** adresových prostorů
- Př. – máme několik tabulek a chceme, aby jejich velikost mohla růst
- 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. **překladač jazyka** – **samostatné** segmenty může mít pro
  - Kód přeloženého programu
  - Globální proměnné
  - Hromada
  - Zásobník návratových adres
- Možné i jemnější dělení – segment pro každou funkci

# Segmentace

---

- Lze použít pro implementaci
  - **Přístup k souborům**
    - 1 soubor = 1 segment
    - Není třeba open, read ..
  - **Sdílené knihovny**
    - Programy využívají rozsáhlé knihovny
    - Vložit knihovnu do segmentu a **sdílet** mezi více programy
- Každý **segment** – **logická entita**
  - má smysl, aby měl **samostatnou ochranu**

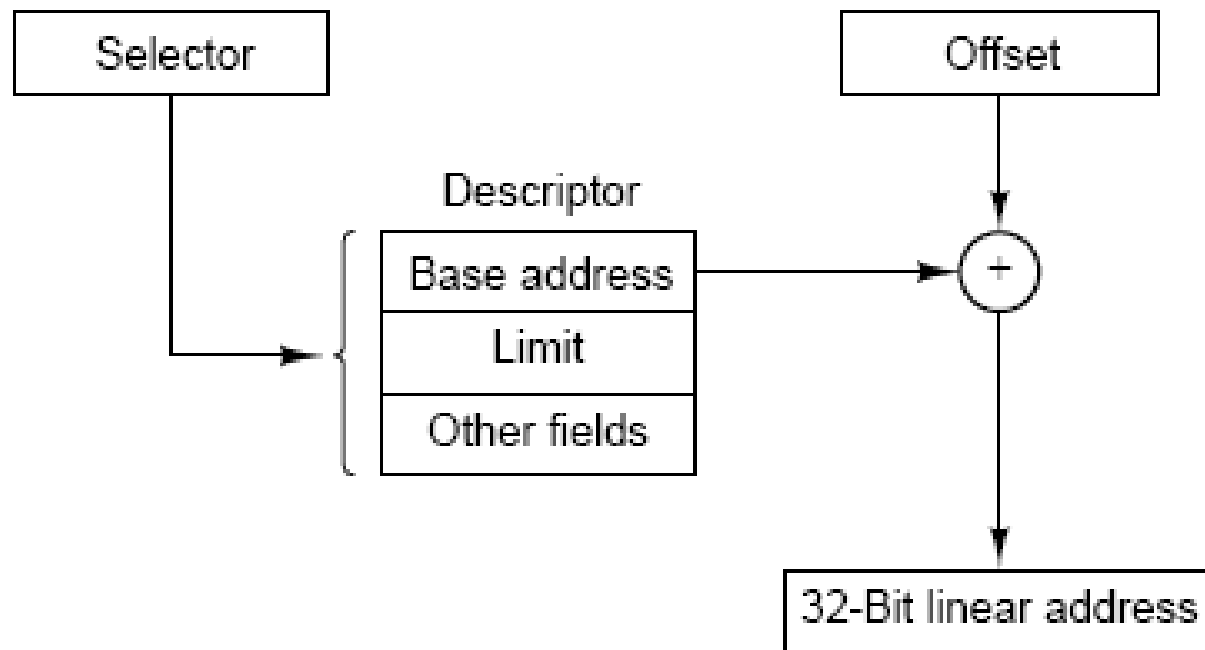
# Čistá segmentace

---

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

(selektor, offset) =>  
lineární adresa

---



# Převod na fyzickou adresu (!!)

---

- PCB obsahuje odkaz na **tabulku segmentů** procesu
- Odkaz do paměti má tvar (selektor, offset)
- Často 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 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
- Myšlenka stránkování segmentů
  - V paměti pouze potřebné stránky
- Implementace – např. každý segment vlastní tabulka stránek

# Adresy (!!!)

---

virtuální adresa -> lineární adresa -> fyzická adresa

virtuální – používá proces (sektor:offset)

lineární – po segmentaci (už jedno 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 Intel procesory

---

- segmentace
- stránkování
- segmentace se stránkováním
- **tabulka LDT (Local Descriptor Table)**
  - Je jich více
  - 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

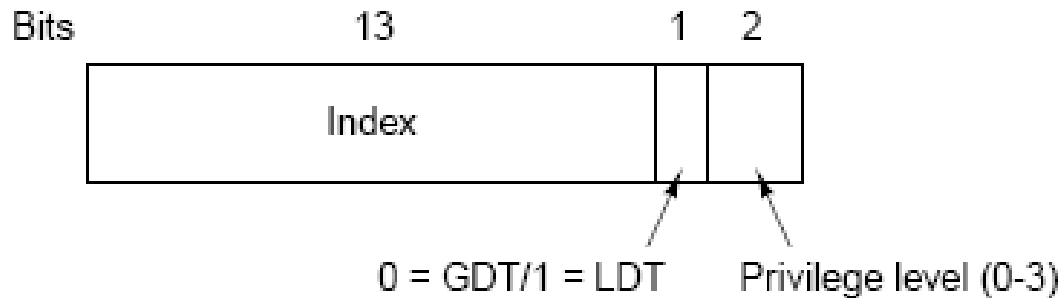
---

- 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 (!!)

---

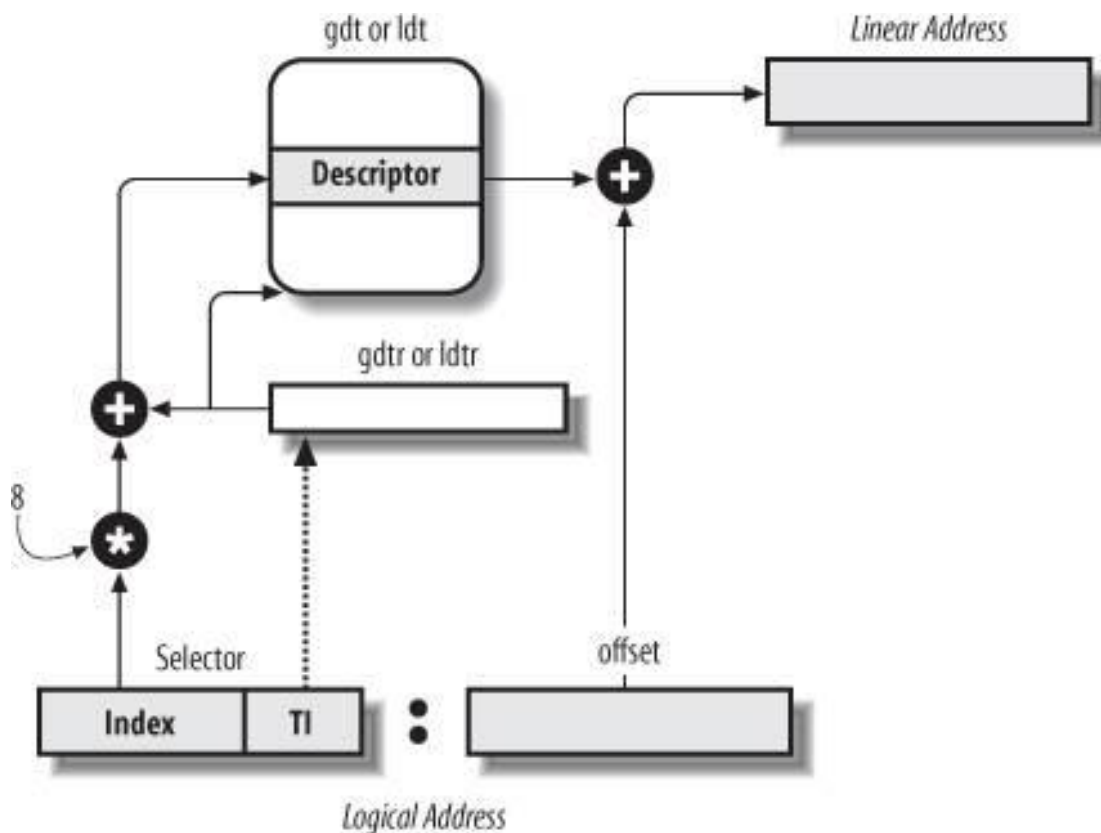
- Selektor – 16bitový
- 13bitů – index to GDT nebo LDT
- 1 bit – 0=GDT, 1=LDT
- 2 bity – úroveň privilegovanosti (0-3, 0 – jádro, 3 – uživ. proces)



# 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 popisovač z LDT nebo GDT do vnitřních registrů CPU
  - bit 2 selektoru – pozná, zda LDT nebo GDT
  - popisovač segmentu na adrese (selektor and 0fff8h) + zač. tabulky

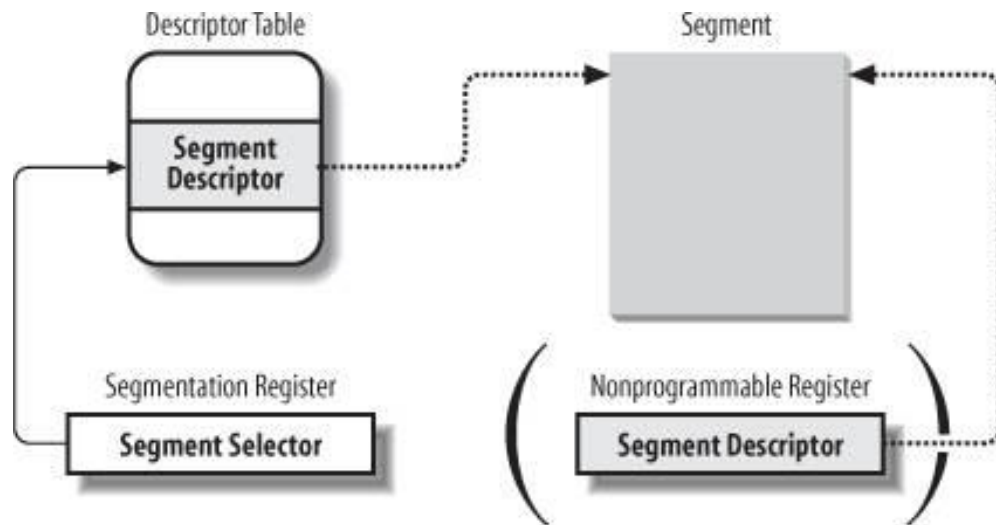


1. z TI pozná, zda použije GDT nebo LDT
2. z indexu selektoru spočte adresu deskriptoru
3. přidá offset k bázi (viz deskriptor), získá lineární adresu

neprogramovatelné registry spojené se segmentovými registry



# Rychlý přístup k deskriptoru segmentu



logická adresa:

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

zrychlení převodu:

přídavné neprogramovatelné registry (pro každý segm. reg.)

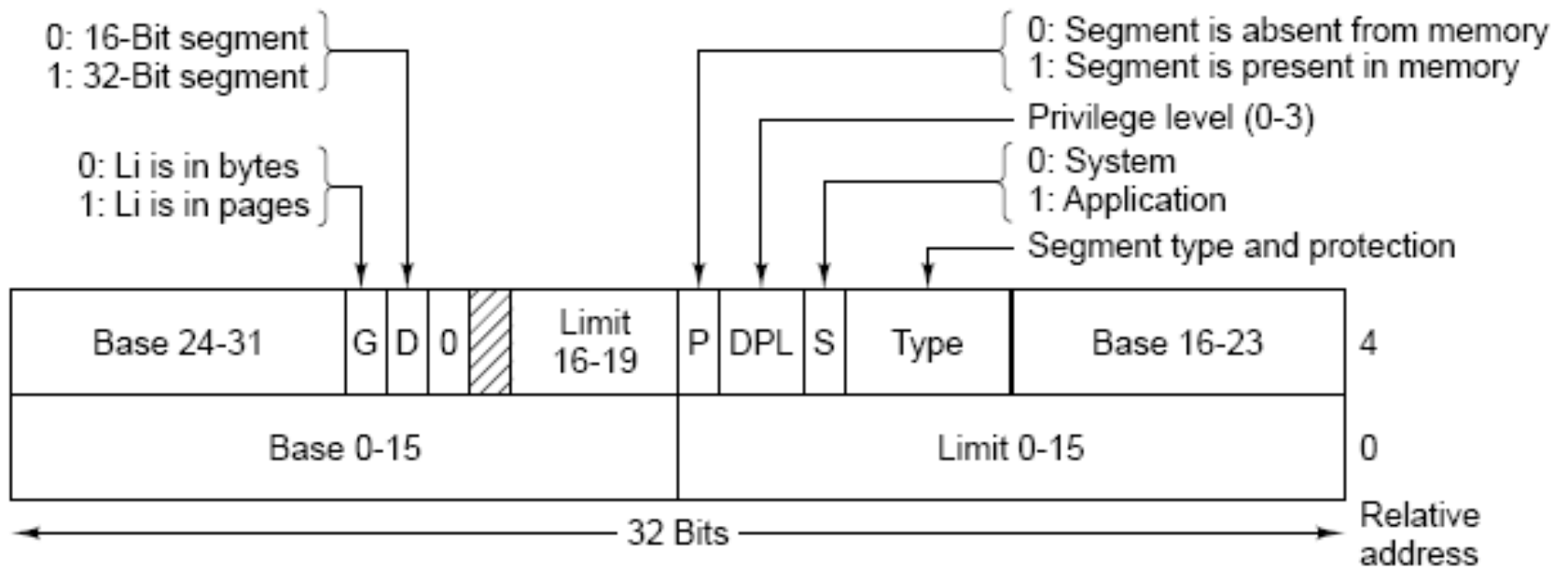
když se nahraje segment **selektor** do segmentového registru, odpovídající **deskriptor** se nahraje do odpovídajícího neprogramovatelného registru

# 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 a ochrana segmentu
    - segment přítomen v paměti..

# Deskriptor segmentu (8 bytů)



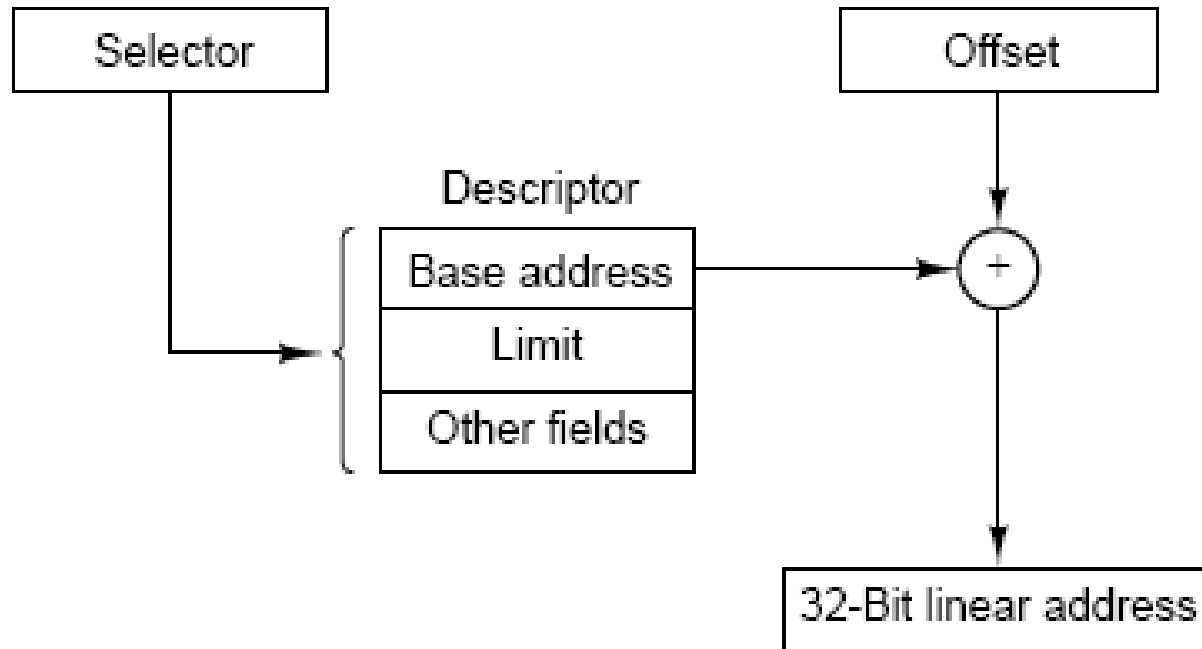
# Konverze na fyzickou adresu

---

- Proces adresuje paměť pomocí **segmentového registru**
- CPU použije odpovídající **popisovač 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.

---

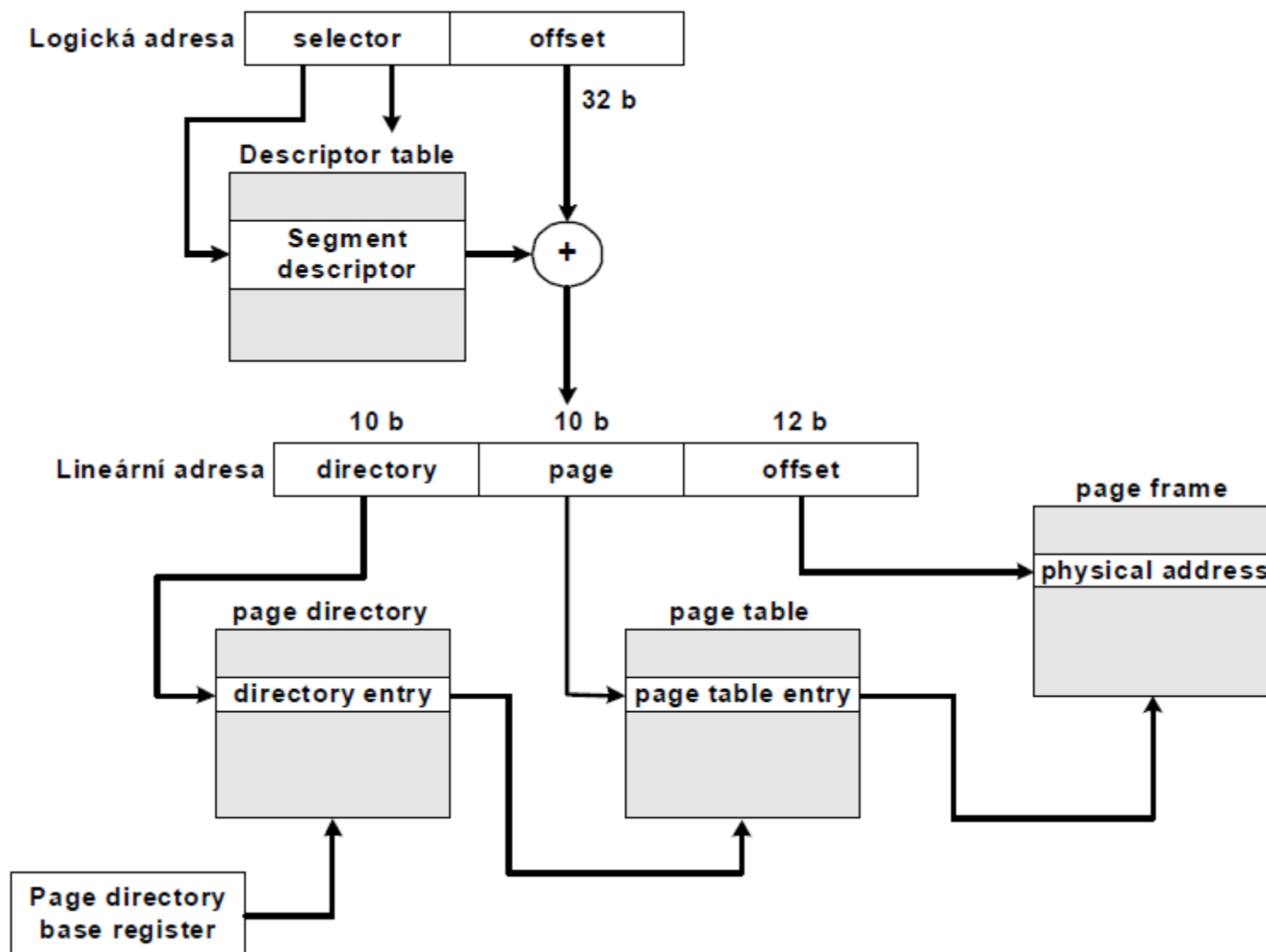


## Konverze na fyzickou adresu II.

---

- 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 !!!!)



# Shrnutí

---

CPU Intel Pentium a výše umožňuje:

- čistá segmentace
- čisté stránkování (Linux)
  - base = 0
  - limit = MAX
  - pozn. segmentaci nejde vypnout
- stránkované segmenty (Windows)



# 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**

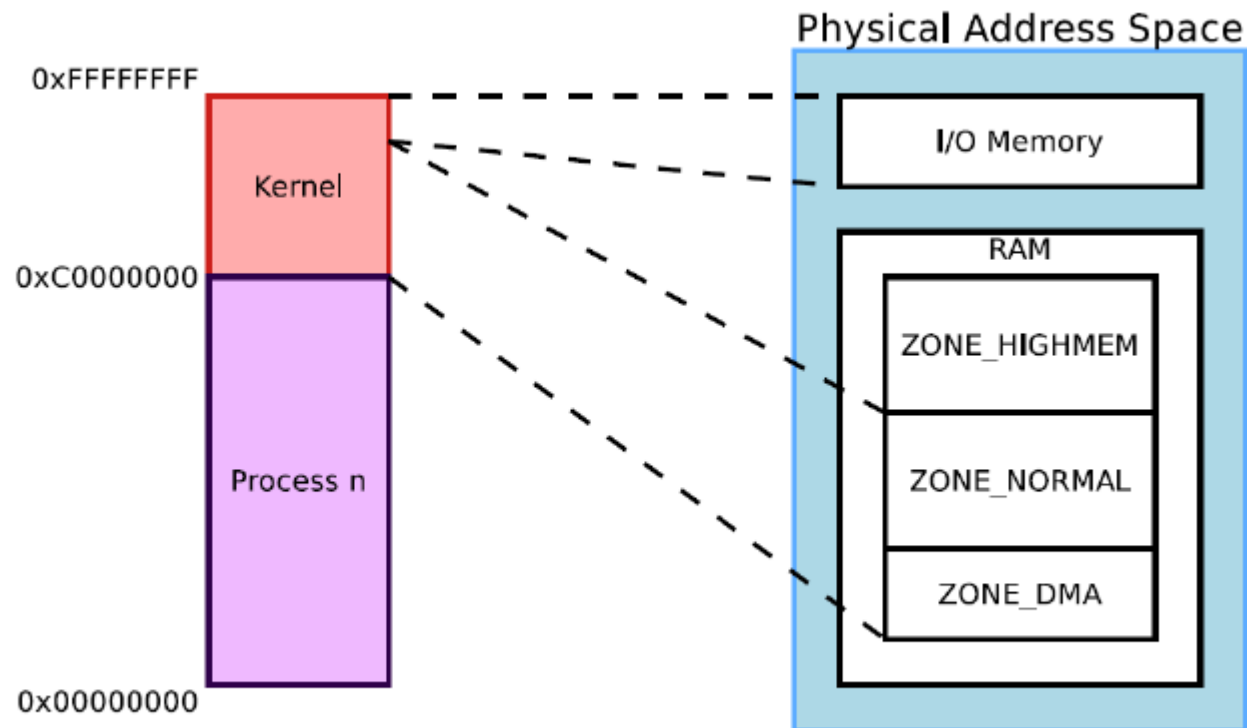
# Paging

- ❑ 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

# Linux – paměťové zóny (32bit)



<http://free-electrons.com/doc/training/linux-kernel/linux-kernel-slides.pdf>

# Linux – paměťové zóny

---

- pro 32bitové systémy
- struct zone (include/linux/mmzone.h)
- ZONE\_DMA: < 16MB
  - Trvale mapovaná do jádra
- ZONE\_NORMAL: 16MB – 896MB
  - Trvale mapovaná do jádra, většina alokací zde
- ZONE\_HIGH\_MEM: >896MB
  - Není trvale mapovaná do jádra
  - 1GB přes ZONE\_DMA a ZONE\_NORMAL
  - 3GB uživatelské přes ZONE\_HIGH\_MEM



# Linux – paměťové zóny

- x86-64 architektura – DMA32 zóna (4GB)

```
ares.fav.zcu.cz - PuTTY
pesicka@ares:~$ cat /proc/buddyinfo
Node 0, zone DMA 0 0 0 2 1 1 1 0
1 1 3
Node 0, zone DMA32 1618 1078 968 562 322 166 23 55
4 0 1
Node 0, zone Normal 10153 6581 1531 532 205 32 4 0
0 0 1
pesicka@ares:~$
```

# 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
- Viz [https://en.wikipedia.org/wiki/Memory\\_segmentation](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](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](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
- 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

# 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](http://wiki.osdev.org/Interrupt_Descriptor_Table)

# Protected mode – další pojmy

---

- call gates
  - volání předdefinované funkce přes CALL FAR
  - volání funkce s vyšší privilegií oprávnění
  - spíše se používá SYSENTER/SYSEXIT
- task state segment (TSS)
  - pro přepínání tasků
  - může obsahovat hodnoty x86 registrů
  - Win/Linux naproti tomu používají softwarový task switching



# 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.

# Poznámky - Brandejs

---

## 5.5.5 Brány

**Brána<sup>1</sup> (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.

# 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).

# 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)

# 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 stránkování, Windows: obojí (ale viz předchozí komentáře)

# Poznámky - prepaging

---

Prepaging:

do paměti se zavádí chybějící stránka a stránky okolní

# Literatura

---

Zdroje několika použitých obrázků a výborný text k prohlédnutí:

<http://www.eecs.harvard.edu/~mdw/course/cs161/notes/>