

# 11.

# Memory management

---

ZOS 2024, L. PEŠIČKA

A solid orange horizontal bar at the bottom of the slide.

# Osnova

---

Základní moduly OS

Modul pro správu procesů

**Modul pro správu paměti**

Modul pro správu periférií

Modul pro správu souborů

# Správa hlavní paměti

---

## Ideál programátora

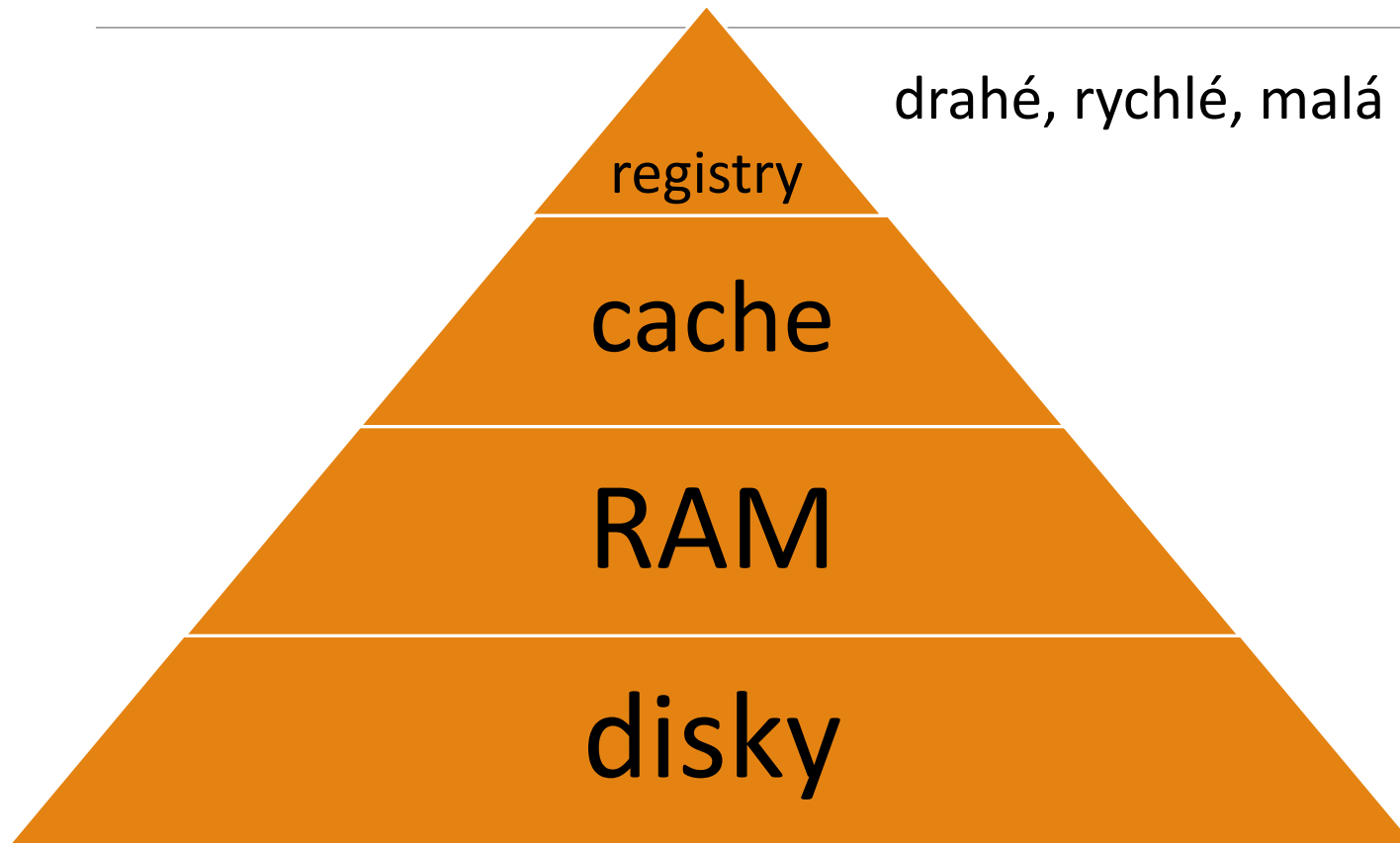
- Paměť nekonečně velká, rychlá, levná
- Zároveň persistentní (uchovává obsah po vypnutí napájení)
- Bohužel neexistuje

## Reálný počítač – hierarchie pamětí („pyramida“)

- Registry CPU
- Cache paměť - malé množství, rychlá
- RAM paměť – 4GB, 8GB, 16GB u dnešních PC
- Pevné disky – 1-10TB, pomalé, persistentní, SSD vs. rotační

# Paměťová pyramida

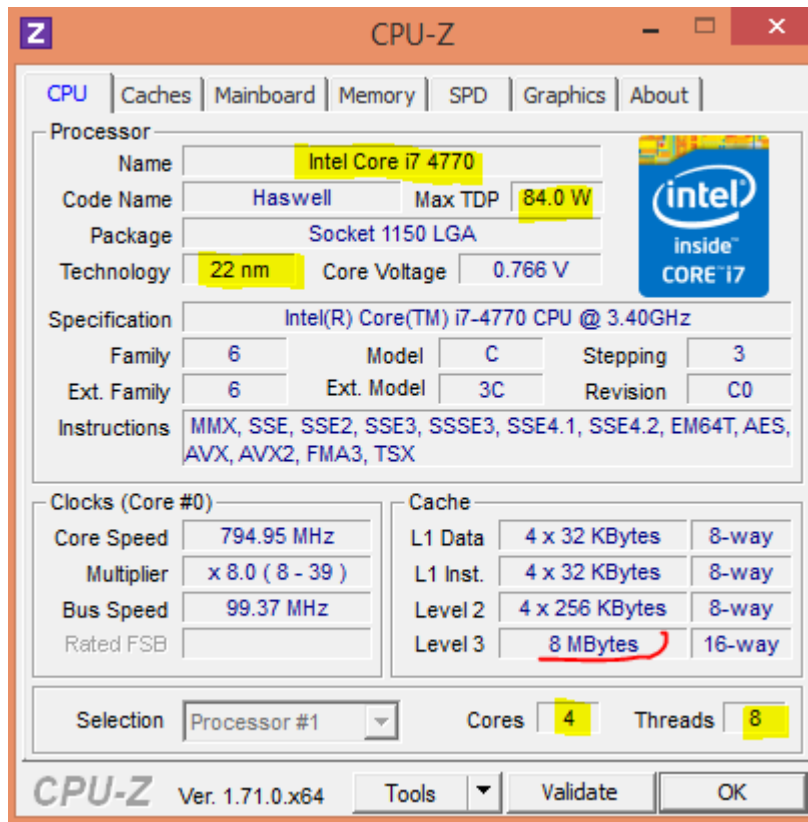
---



drahé, rychlé, malá kapacita

levné, pomalé, velká kapacita

# Informace o CPU



Typ - Intel Core i7 4770  
Kolik hřeje – TDP – 84W  
Výrobní technologie – 22 nm

Počet jader: 4  
Počet vláken (threads): 8

Každé jádro umí 2 vlákna  
(hyperthreading)

Cache paměti:

L3 8MB

L2

L1 instrukční, datová

Výrazný vliv na výkon

# Správce paměti

---

- Část OS, která spravuje paměť
- Udržuje informaci, které části paměti se používají a které jsou volné
- Alokuje paměť procesům podle potřeby
  - funkce **malloc** v jazyce C, (**new** v Pascalu)
- Zařazuje paměť do volné paměti po uvolnění procesem
  - funkce **free** v jazyce C, (**release** v Pascalu)

# Příklad alokace

---

- proces požádá o alokaci  $n$  bajtů paměti funkcí  
ukazatel = `malloc(n)`
- malloc je knihovní fce alokátoru paměti (součást glibc)
- paměť je alokována z **haldy** (heapu) !
- alokátor se podívá, zda má volnou paměť k dispozici, když ne, požádá OS o přidělení dalších stránek paměti (systémové volání `sbrk`)
- proces uvolní paměť, když už ji nepotřebuje voláním `free(ukazatel)`

# Volání brk(), sbrk()

---

## Description

**brk()** and **sbrk()** change the location of the *program break*, which defines the **end of the process's data segment** (i.e., the program break is the first location after the end of the uninitialized data segment). Increasing the program break has the effect of **allocating memory** to the process; decreasing the break deallocates memory.

**brk()** sets the end of the data segment to the value specified by *addr*, when that value is reasonable, the system has enough memory, and the process does not exceed its maximum data size (see **setrlimit(2)**).

**sbrk()** **increments** the program's data space by *increment* bytes. Calling **sbrk()** with an *increment* of 0 can be used to find the current location of the program break.

Zkuste: `man brk`

`brk (adresa)`  
`sbrk(increment)`



# Příklad alokace

zkuste: `man malloc`  
Zjednodušený příklad, ve skutečnosti jsou alokátory poměrně složité algoritmy

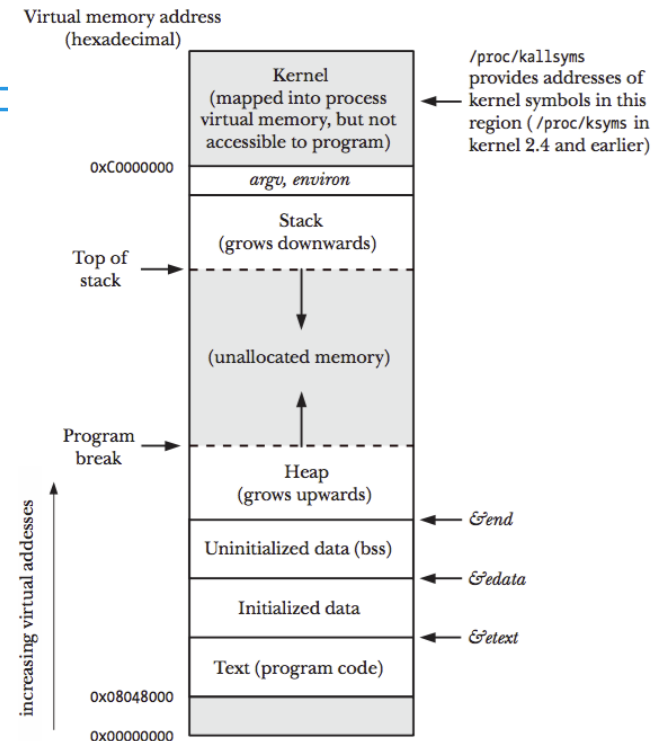
Příklad:

1. proces bude chtít alokovat 500B, zavolá `malloc`
2. alokátor zkontroluje, že nemá volnou paměť, požádá OS o přidělení stránky paměti (4KB) – `sbrk`
3. proces je obsloužen, dostane paměť 500B
4. proces bude chtít dalších 200B, zavolá `malloc`
5. alokátor už má paměť v zásobě, rovnou ji přidělí procesu
6. když už proces paměť nepotřebuje, zavolá `free`

# Další materiály

<https://medium.com/@andrestc/implementing-malloc-and-free-ba7e7704a473>

<http://gee.cs.oswego.edu/dl/html/malloc.html>



# man malloc

Windows:  
také malloc() nebo HeapAlloc()

```
MALLOC(3)                                Linux Programmer's Manual                                MALLOC(3)

NAME
    calloc, malloc, free, realloc - Allocate and free dynamic memory

SYNOPSIS
    #include <stdlib.h>

    void *calloc(size_t nmemb, size_t size);
    void *malloc(size_t size);
    void free(void *ptr);
    void *realloc(void *ptr, size_t size);

DESCRIPTION
    calloc() allocates memory for an array of nmemb elements of size bytes
    each and returns a pointer to the allocated memory. The memory is set
    to zero. If nmemb or size is 0, then calloc() returns either NULL, or
    a unique pointer value that can later be successfully passed to free().

    ↪ malloc() allocates size bytes and returns a pointer to the allocated
    memory. The memory is not cleared. If size is 0, then malloc()
    returns either NULL, or a unique pointer value that can later be sucâ
    cessfully passed to free().

    ↪ free() frees the memory space pointed to by ptr, which must have been
    returned by a previous call to malloc(), calloc() or realloc(). Otherâ
    wise, or if free(ptr) has already been called before, undefined behavâ
    ior occurs. If ptr is NULL, no operation is performed.
```

# poznámka k pointerům (!!!)

---

ukazatel = malloc (size)

takto získaný ukazatel obsahuje **virtuální adresu**, tj. není to přímo adresa do fyzické paměti (RAM) !!!

virtuální adresa se uvnitř procesoru převede **na fyzickou adresu** (s využitím tabulky stránek atd.) !!

Vrací-li malloc hodnotu 500, tak 500 není adresa v RAM.  
Je to virtuální adresa, která bude na fyzickou teprve převedena uvnitř CPU.

# Poznámka k virtuálním adresám

---

- proces P1 může manipulovat s proměnnou na virtuální adrese 5123
- proces P2 může také manipulovat se svou proměnnou na virtuální adrese 5123
- Každý z procesů má ale vlastní mapování virtuální adresy na fyzickou adresu do RAM
- Tedy i když budou pracovat se stejnou virtuální adresou, bude ukazovat na jiné místo v RAM !!!

# Jádro a paměť

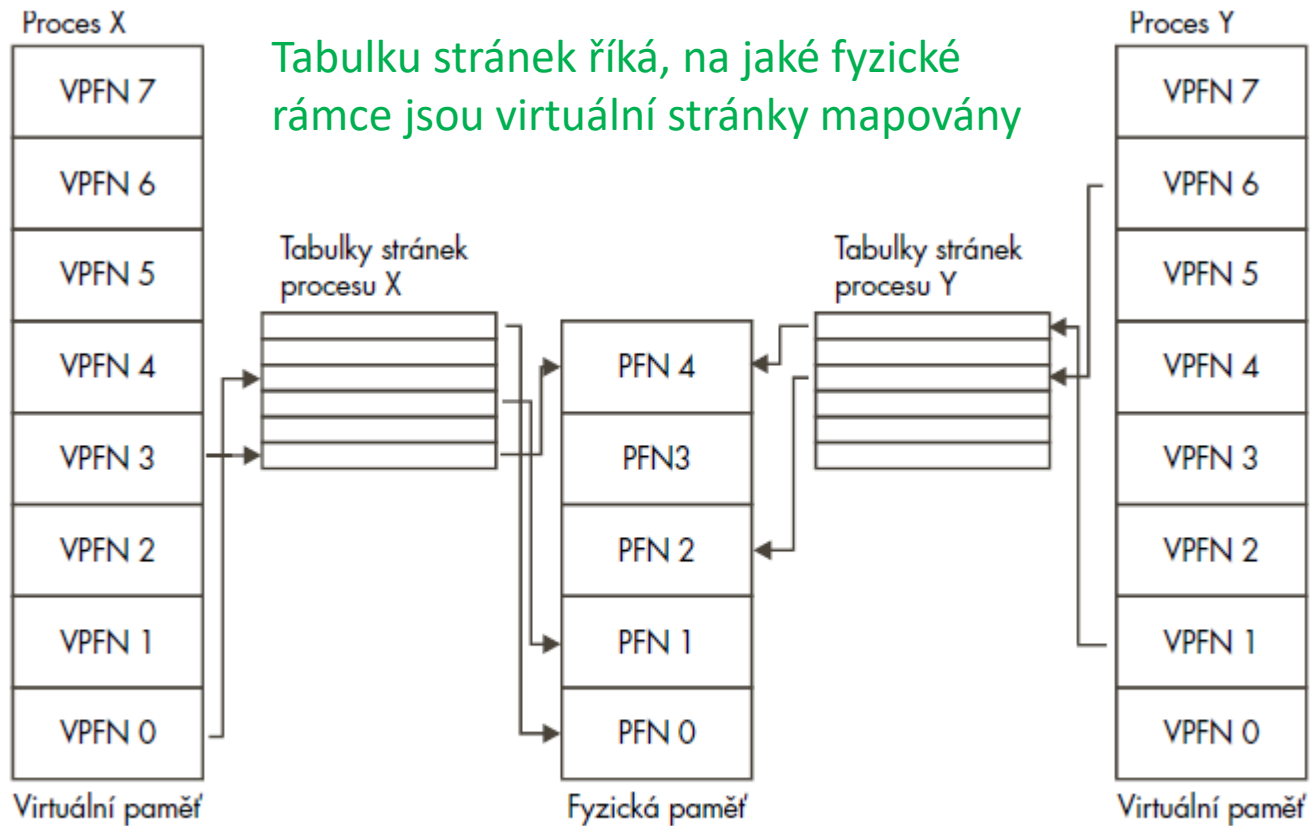
---

Virtuální adresování také umožňuje vytváření virtuálních částí paměti ve dvou rozdělených oblastech, jedna bývá rezervována pro jádro (prostor jádra) a ostatní pro aplikace (uživatelský prostor). Procesor nepovoluje aplikacím, aby adresovaly paměť jádra, a tedy aby aplikace poškodila běžící jádro. Toto důležité rozdělení paměťového prostoru hodně přispívá nynější koncepci současných obecných jader a je téměř univerzální ve většině systémů, ačkoliv některá jádra (například Singularity) volí jiné metody.

Zdroj:

[https://cs.wikipedia.org/wiki/J%C3%A1dro\\_%28informatika%29](https://cs.wikipedia.org/wiki/J%C3%A1dro_%28informatika%29)

## Uživatelské procesy (ukázka stránkování)



**Obrázek 3.1**

Abstraktní model mapování virtuálních adres na fyzické adresy

Každý proces má svojí vlastní tabulku stránek

# Mechanismy správy paměti

---

Od nejjednodušších (program má veškerou paměť) po  
propracovaná schémata (stránkování se segmentací)

Dvě kategorie:

## Základní mechanismy

- Program je **v paměti po celou dobu** svého běhu

## Mechanismy s odkládáním

- Programy **přesouvány mezi hlavní pamětí** a diskem



# Základní mechanismy pro správu paměti

---

Nejprve probereme základní mechanismy  
(Program je v paměti po celou dobu běhu)

1. Jednoprogramové systémy
2. Multiprogramování s pevným přidělením paměti
3. Multiprogramování s proměnnou velikostí oblasti

# Jednoprogramové systémy

---

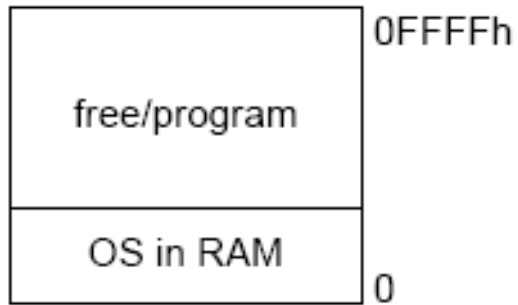
- Spouštíme **pouze jeden program v jednom čase**
- Uživatel – zadá příkaz , OS zavede program do paměti
- Dovoluje použít **veškerou paměť, kterou nepotřebuje OS**
- Po skončení procesu lze spustit další proces

## Tři příklady rozdělení paměti:

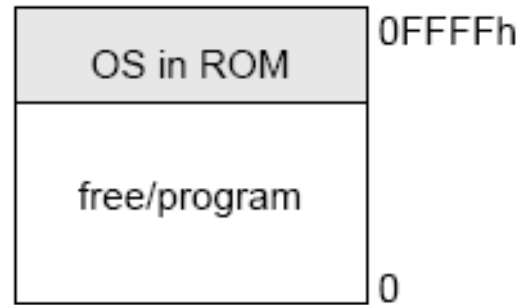
- a) **OS ve spodní části** adresního prostoru v **RAM** (minipočítače)
- b) **OS v horní části** adresního prostoru v **ROM** (zapouzdřené systémy)
- c) **OS v RAM, výchozí obslužné rutiny v ROM**  
(na PC – MS DOS v RAM, BIOS v ROM)

# Jednoprogramové systémy

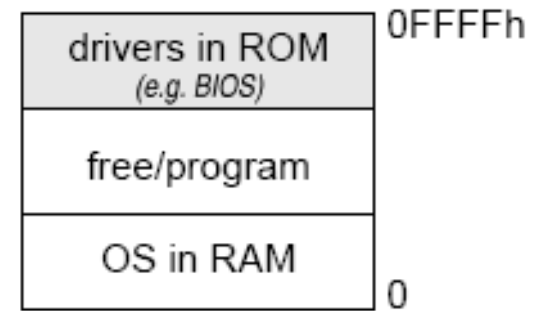
---



a)



b)



c)

# Multiprogramování s pevným přidělením paměti

---

- Paralelní nebo pseudoparalelní běh více programů  
= multiprogramování (nebo také multiprocesní systém)
- Práce více uživatelů, maximalizace využití CPU apod.
- Nejjednodušší schéma – rozdělit paměť na  $n$  oblastí (i různé velikosti)
  - V historických systémech – rozdělení ručně při startu stroje
  - Po načtení úlohy do oblasti je obvykle část oblasti nevyužitá
  - Snaha umístit úlohu do nejmenší oblasti, do které se vejde

# Pevné rozdělení sekcí

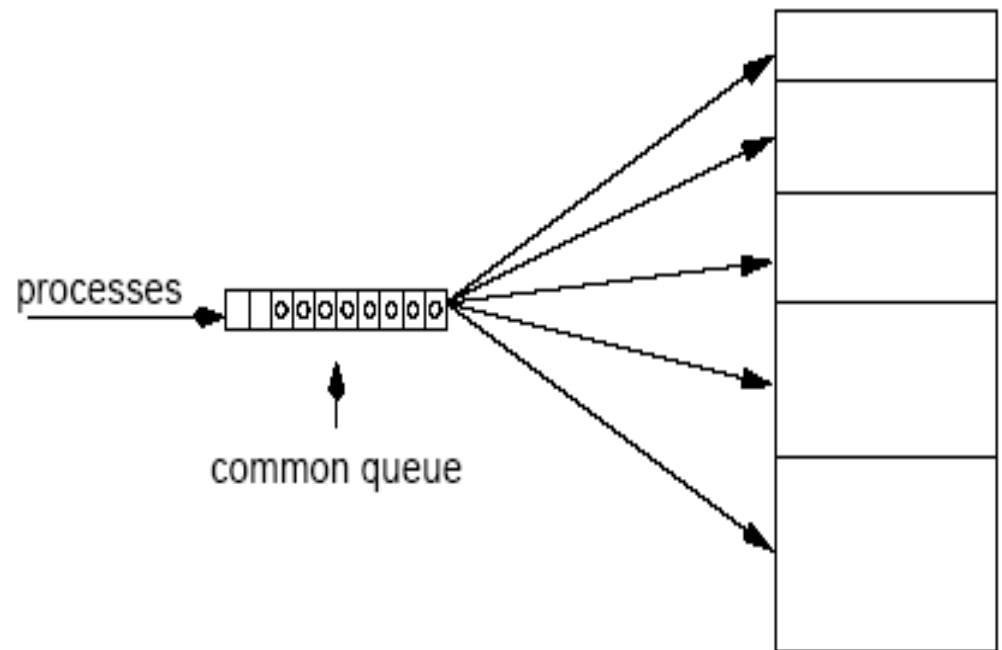
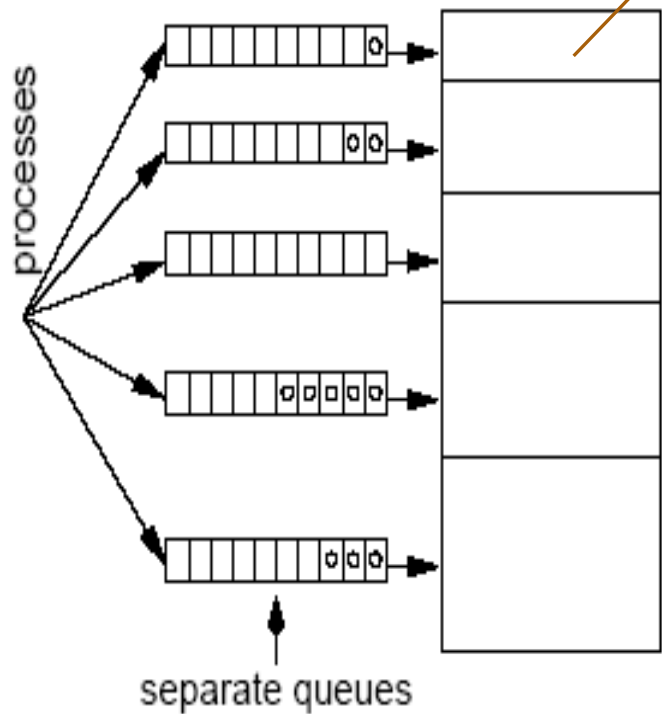
---

Několik strategií:

1. **Více front**, každá úloha do nejmenší oblasti, kam se vejde
2. **Jedna fronta** – po uvolnění oblasti z fronty vybrat **největší úlohu**, která se vejde (tj. není FIFO!)

# Pevné rozdělení sekcí

Sekce mají různé velikosti



# Pevné rozdělení sekcí - vlastnosti

---

## Strategie 1. – více fronta

- Procesy čekají ve frontě na nejmenší oblast kam se vejdou
- Může se stát, že existuje neprázdná oblast, která se nevyužije, protože úlohy čekají na jiné oblasti

## Strategie 2. – společná fronta

- Diskriminuje malé úlohy (**vybíráme největší co se vejde**)
  - ale malým bychom měli obvykle poskytnout nejlepší službu
- Řešení – mít vždy malou oblast, kde poběží malé úlohy
- Řešení – s každou úlohou ve frontě sdružit „čítač přeskočení“, bude zvětšen při každém přeskočení úlohy; po dosažení mezní hodnoty už nesmí být úloha přeskočena

# Pevné rozdělení sekcí - poznámky

---

- Používal např. systém OS/360  
(Multiprogramming with Fixed Number of Tasks)
- Multiprogramování zvyšuje využití CPU
  - Proces – část času  $p$  tráví čekáním na dokončení I/O
  - $N$  procesů – pst, že **všechny** čekají na I/O je:  $p^n$
  - Využití CPU je  $u = 1 - p^n$



# Příklad

---

využití CPU je  $u = 1 - p^n$

pokud proces tráví 80% času čekáním,  $p = 0.8$

$n = 1$     ...     $u = 0.2$  (20% času CPU využito)

$n = 2$     ...     $u = 0.36$  (36%)

$n = 3$     ...     $u = 0.488$  (49%)

$n = 4$     ...     $u = 0.5904$  (59%)

$n$  je tzv. **stupeň multiprogramování**

Je zde zjednodušení - předpokládá nezávislost procesů, což při jednom CPU není pravda.

# Multiprogramování s proměnnou velikostí oblastí

---

- Úloze je přidělena paměť dle požadavku
- V čase se mění
  - Počet oblastí
  - Velikost oblastí
  - Umístění oblastí
- Zlepšuje využití paměti
- Komplikovanější alokace / dealokace

# Př.: IBM OS/360



batch processing  
operating system

1964

zdroj obrázku:

[http://www.maximumpc.com/article/features/ibm\\_os360\\_windows\\_31\\_software\\_changed\\_computing\\_forever](http://www.maximumpc.com/article/features/ibm_os360_windows_31_software_changed_computing_forever)

# IBM OS/360

- **Single Sequential Scheduler (SSS)**
  - Option 1
  - Primary Control Program (PCP)
- **Multiple Sequential Schedulers (MSS)**
  - Option 2
  - **Multiprogramming with a Fixed number of Tasks (MFT)**
  - MFT 2
- **Multiple Priority Schedulers (MPS)**
  - Option 4
  - VMS<sup>[NB 1]</sup>
  - **Multiprogramming with a Variable number of Tasks (MVT)**
  - Model 65 Multiprocessing (M65MP)



zdroj:  
<http://www.escapistmagazine.com/forums/read/18.85690-Esoteric-Operating-Systems-The-History-of-OS-360-and-its-successors>

# Problém mnoha volných oblastí

---

- Může vzniknout mnoho volných oblastí (děr)
  - Paměť se „rozdrobí“
- **Kompaktace paměti** (compaction)
  - Přesunout procesy směrem dolů (procesy, volné místo)
  - **Drahá** operace (pokud 1B .. 10ns, pak 256MB .. 2.7s)
  - Neprovádí se bez speciálního HW

# Volná x alokovaná paměť

---

Pro zajištění správy paměti se používají:

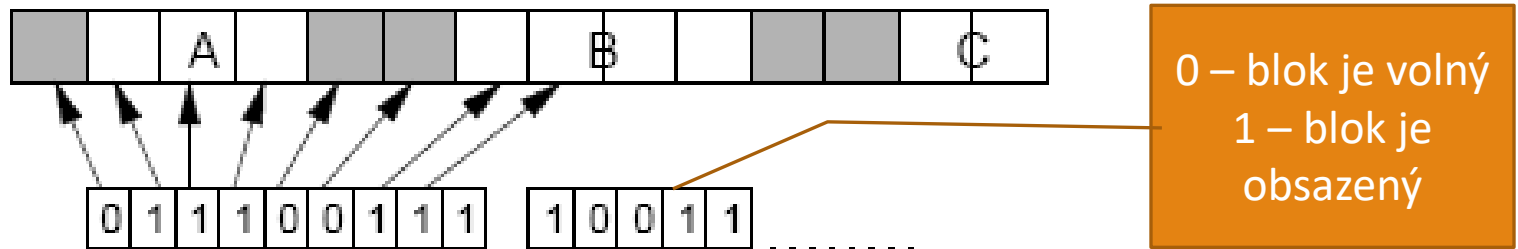
1. bitové mapy
2. seznamy  
(prohledání first fit, best fit, next fit, ...)
3. buddy systems

U každého bloku paměti potřebujeme rozhodnout, zda je volný nebo někomu přidělený

# Správa pomocí bitových map

Paměť rozdělíme na alokační jednotky stejné délky (B až KB)

S každou jednotkou 1bit ( volno x obsazeno)



Menší alokační jednotky → větší bitmapa

Větší jednotky → více nevyužitá paměť

Alokační jednotka 4 byty (32bitů):

na každých 32bitů paměti potřebujeme 1bit signalační  
tedy .. 1/33 paměti spotřebuje bitmapa

# Bitové mapy

---

- výhoda
  - konstantní velikost bitové mapy
- Nevýhoda
  - najít požadovaný úsek  $N$  volných jednotek
  - Náročné, příliš často se nepoužívá pro RAM
  - Ale používá se např.  
pro určení volných bloků a volných i-uzlů na disku

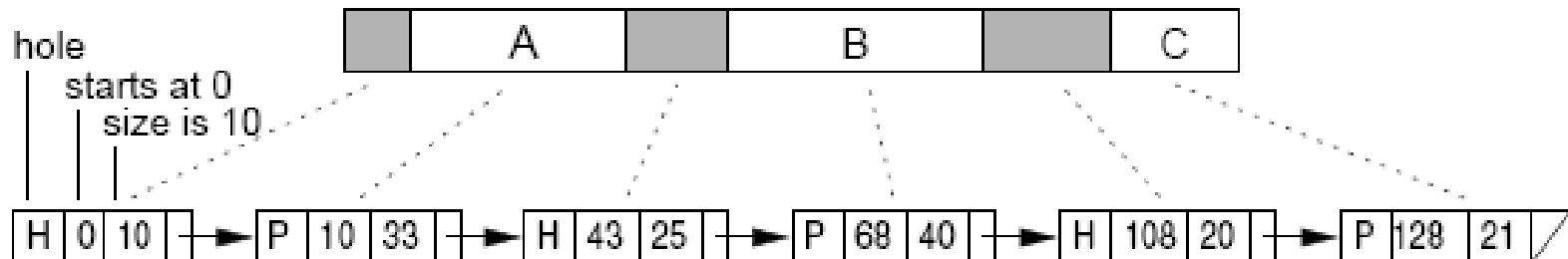


# Správa pomocí seznamů

Seznam alokovaných a volných oblastí (procesů, děr)

Položka seznamu:

- Info o typu – proces nebo díra (P vs. H)
- Počáteční adresa oblasti
- Délka oblasti



# Práce se seznamem

---

- Proces skončí → P se nahradí H (dírou)
- Dvě díry vedle sebe → **sloučí** se do jedné

Seznam seřazený podle počáteční adresy oblasti

Může být obousměrně vázaný seznam  
– snadno k předchozí položce

Jak prohledávat seznam, když proces potřebuje alokovat paměť?

# Alokace – first fit, next fit

---

- **First Fit** (první vhodná)

- Prohledávání od začátku, dokud se nenajde **dostatečně velká** díra
- Díra se rozdělí na část pro proces a nepoužitou oblast (většinou „nesedne“ přesně)
- Rychlý, prohledává co nejméně

- **Next Fit** (další vhodná)

- Prohledávání začne tam, kde skončilo předchozí
- O málo horší než *first fit*

# Alokace best fit

---

- **Best fit** (nejmenší/nejlepší vhodná)
  - Prohlédne celý seznam, vezme nejmenší díru, do které se proces vejde
  - Pomalejší – prochází celý seznam
  - Více ztracené paměti než FF,NF – zaplňuje paměť malými nepoužitelnými dírami
- **Worst fit (největší díra) – není vhodné**
  - nepoužívá se

# Urychlení

---

## Oddělené seznamy pro proces a díry

- Složitější a pomalejší dealokace
- Vyplatí se při rychlé alokaci paměti pro data z I/O zařízení
- *Alokace* – jen seznam děr
- *Dealokace* – složitější – přesun mezi seznamy, z děr do procesů

## Oddělené seznamy, seznam děr dle velikosti

- Optimalizace best fitu
- První vhodná – je i nejmenší vhodná, rychlost First fitu
- Režie na dealokaci – sousední fyzické díry nemusí být sousední v seznamu

# Další varianty – Quick Fit

---

## ■ Quick Fit

- Samostatné seznamy děr nejčastěji požadovaných délek
- Díry velikosti 4KB, 8KB,...
- Ostatní velikosti v samostatném seznamu
- Alokace – rychlá
- Dealokace – obtížné sdružování sousedů

# Šetření paměti

---

- Díry můžeme využít k uložení seznamu děr
  - Hodně volného místa – máme prostor na delší seznam
- Obsah díry
  - 1. slovo – velikost díry
  - 2. slovo – ukazatel na další díru

# KVIZ

---

Jaký je vzájemný poměr počtu  
děr a procesů?

Předpokládejme, že pro daný proces alokujeme paměť  
jednorázově (v celku)



# Asymetrie mezi procesy a dírami

---

- Dvě sousední díry (H) se sloučí
- Dva procesy (P) se nesloučí

Při normálním běhu je počet děr **poloviční** oproti počtu procesů

# Přehled

---

Správa paměti:

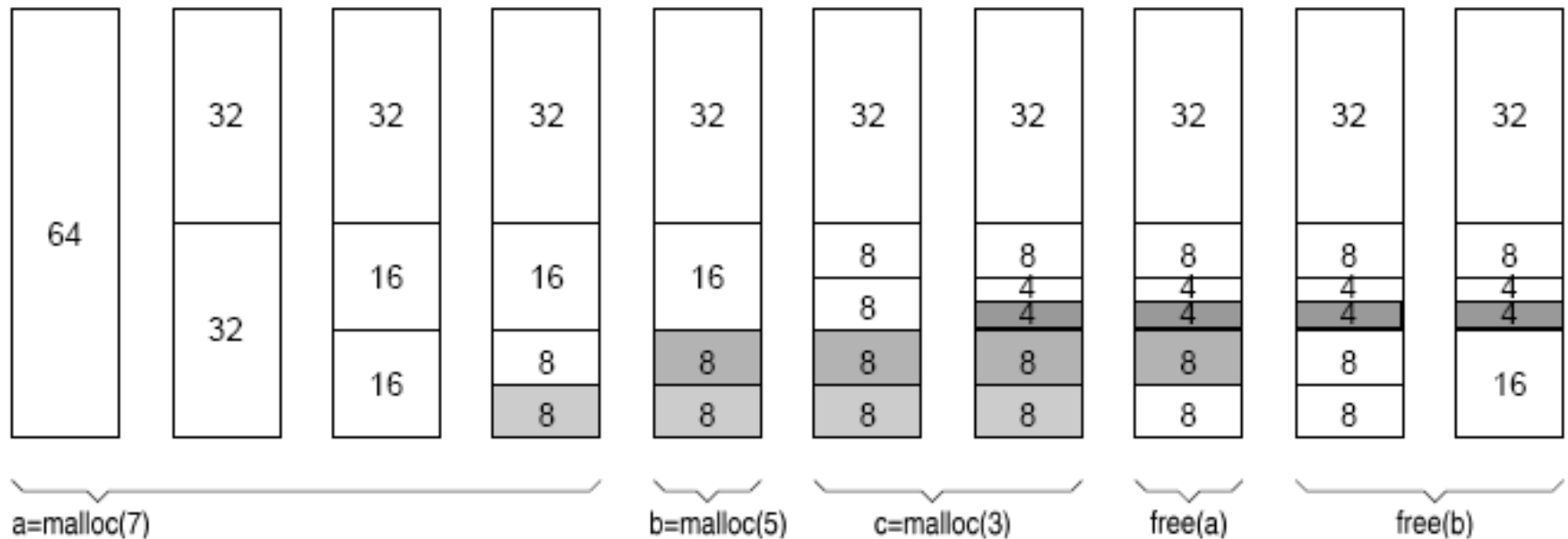
1. Bitové mapy
2. Seznamy ( first fit, next fit, ...)
3. Buddy systems

# Buddy systems

---

- Seznamy volných bloků 1, 2, 4, 8, 16 ... alokačních jednotek až po velikost celé paměti
- Nejprve seznamy prázdné vyjma 1 položky v seznamu o velikosti paměti
- Př.: Alokační jednotka 1KB, paměť velikosti 64KB
- Seznamy 1, 2, 4, 8, 16, 32, 64 (7 seznamů)
- Požadavek se zaokrouhlí na mocninu dvou nahoru
  - např. požadavek 7KB na 8KB
- Blok 64KB se rozdělí na 2 bloky 32KB (buddies) a dělíme dále...

# Buddy system



Nejmenší dostatečně velký blok se rozdělí

Dva volné sousední bloky stejné velikosti (buddies) – spojí se do většího bloku

# Buddy system

---

Neefektivní (plýtvání místem) x rychlý

- Chci 9KB, dostanu 16KB
- Alokace paměti – vyhledání v seznamu dostatečně velkých děr
- Slučování – vyhledání buddy

Linux - pro správu paměti jádra používá buddy system

viz: *cat /proc/buddyinfo*

# Použití algoritmů

---

Použití first fit, next fit:

Fce malloc v jazyce C

žádá OS o větší blok paměti a získanou paměť pak aplikaci přiděluje v některých systémech algoritmem **first fit** či **next fit**

# Správa paměti

---

- „paměťová pyramida“
- absolutní adresa
- relativní adresa
  - *počet bytů od absolutní adresy (nějakého počátku)*
- fyzický prostor adres
  - fyzicky k dispozici výpočetnímu systému (RAM)
- logický adresní prostor
  - využívají procesy

# Modul pro správu paměti

---


- informace o přidělení paměti
  - volná - která část paměti je volná
  - přidělená (a kterému procesu)
- přidělování paměti na žádost
- uvolnění paměti
  - zařazení k volné paměti
- odebírá paměť procesům
- ochrana paměti
  - chrání přístup k paměti jiného procesu
  - chrání přístup k paměti OS



# Memory management

## ■ Základní mechanismy

- Bez odkládání a stránkování
- Jednoprogramové systémy
- Multiprogramování s pevným přidělením paměti
- Multiprogramování s proměnnou velikostí oblasti
- Správa paměti
  - Bitové mapy
  - Seznamy
    - First fit, best fit, next fit
  - Buddy system



Celý proces se  
musí vejít do  
paměti



Shrnutí již probrané části

---

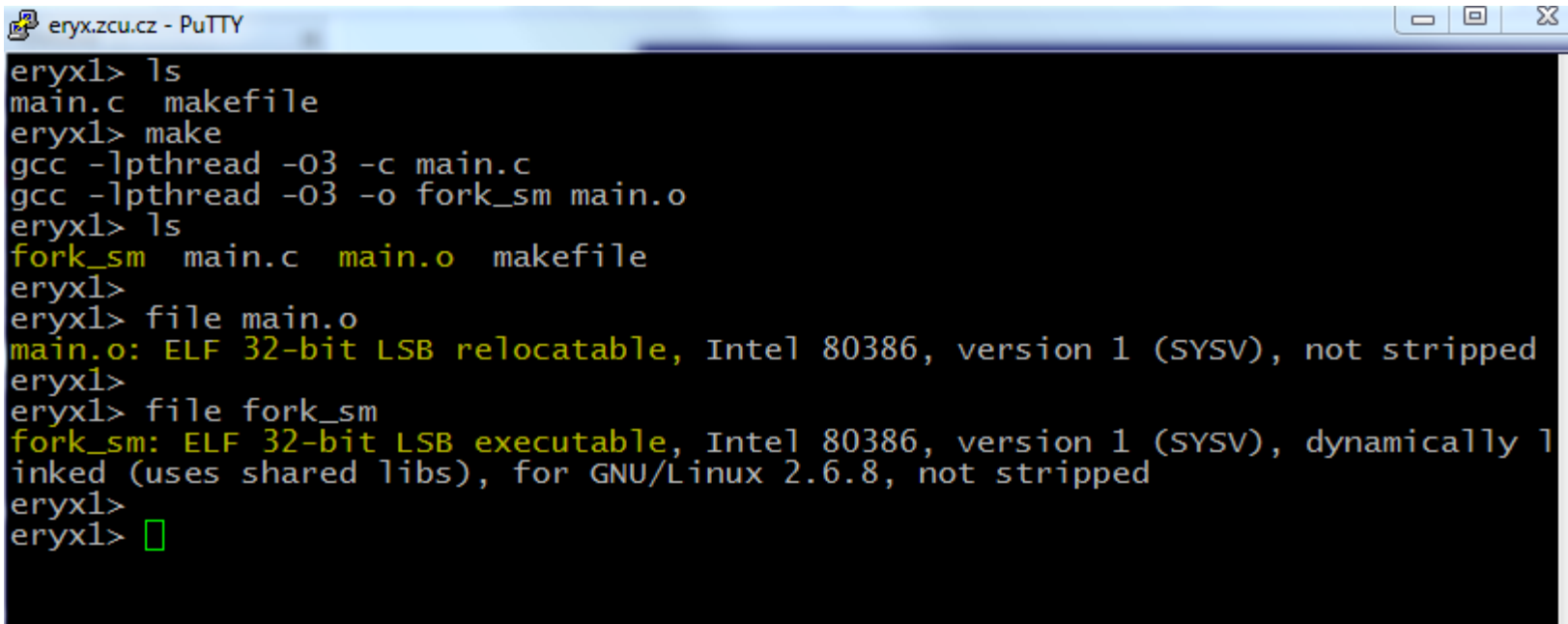
# Statická a dynamická relokační

# Relokace a ochrana

---

- Problémy při multiprogramování (více programů současně v paměti):
  - Relokace
    - Programy běží na různých (fyzických) adresách
    - Jednou je ve fyzické paměti od adresy 1000, jindy od 2000
  - Ochrana
    - Paměť musí být chráněna před zasahováním jiných programů

# ukázka překladu .c programu



```
eryx1> ls
main.c  makefile
eryx1> make
gcc -lpthread -O3 -c main.c
gcc -lpthread -O3 -o fork_sm main.o
eryx1> ls
fork_sm main.c main.o makefile
eryx1>
eryx1> file main.o
main.o: ELF 32-bit LSB relocatable, Intel 80386, version 1 (SYSV), not stripped
eryx1>
eryx1> file fork_sm
fork_sm: ELF 32-bit LSB executable, Intel 80386, version 1 (SYSV), dynamically linked (uses shared libs), for GNU/Linux 2.6.8, not stripped
eryx1>
eryx1> █
```

zdrojový soubor  
objektový modul  
spustitelný soubor

main.c  
main.o  
fork\_sm

Jen vsuvka na rozlišení pojmů:  
zdrojový kód, objektový modul,  
spustitelný soubor

# Relokace při zavedení do paměti

---

jak je program **vytvořen a spuštěn**:



překladač + linker

## ■ Překlad a sestavení programu

- Aplikace ve **vysokoúrovňovém** jazyce, např. C
- Větší SW – rozděleny do modulů – musejí být přeloženy a sestaveny do spustitelného programu
- **Objektové moduly**
  - Výsledkem překladu
  - Příkazy ve zdrojovém textu – přeloženy do stroj. instrukcí
  - Zůstávají symbolické odkazy – adresy proměnných, procedur, funkcí

# Relokace při zavedení do paměti 2

---

- Výsledný spustitelný program
  - Sestavení ([linkování](#)) modulů a knihoven
- Při sestavení se řeší hlavně [externí reference](#)
  - Všechna místa výskytů referencí – seznam
  - Když už je adresa známa – vloží se všude, kde se používá
  - Symbolické odkazy se převedou na číselné hodnoty
  - Výsledek – spustitelný program

# Relokace při zavedení do paměti 3

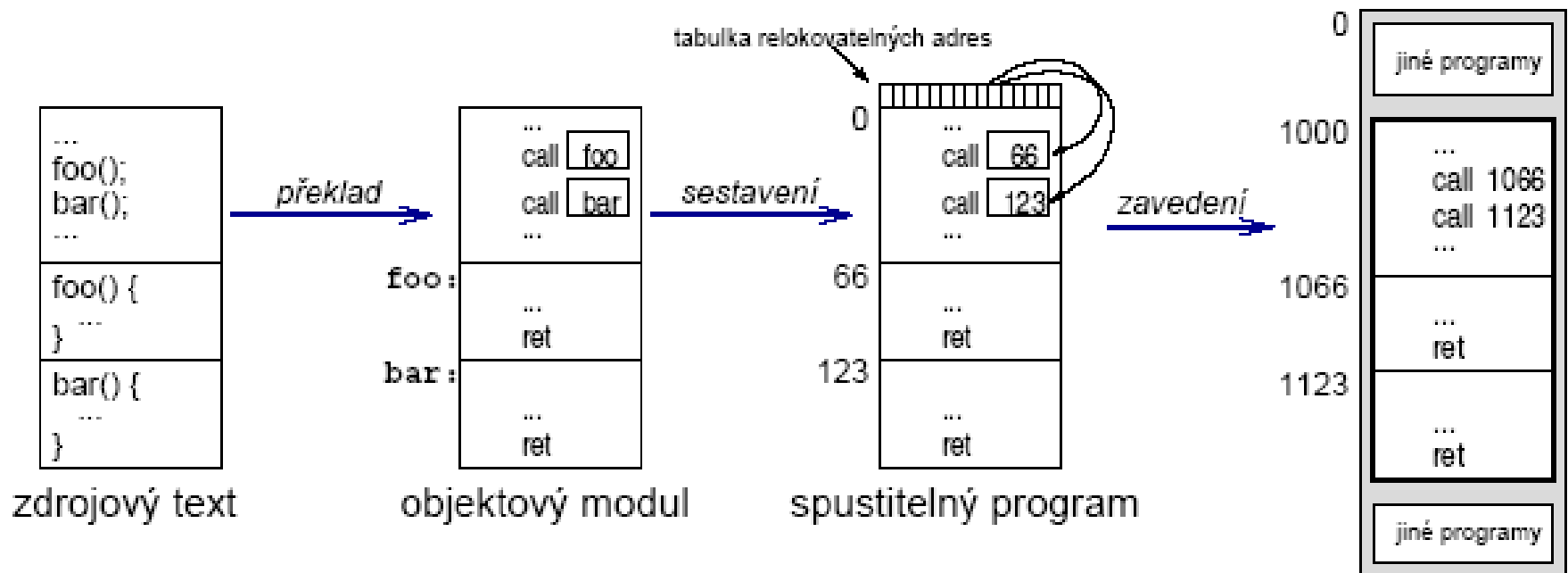
---

- Komplikace při více programech v paměti
  - Příklad
  - První instrukcí programu volání podprogramu *call 66*
  - Program v paměti od adresy *1000*, ve skutečnosti provede call *1066*
- Jedno z řešení – **modifikovat instrukce programu při zavedení do paměti**
  - **Linker** – do spustitelného programu přidá seznam nebo bitmapu označující místa v kódu obsahující adresu
  - Při zavádění programu do paměti se každé adrese **přičte adresa začátku oblasti**



Jak se to  
dříve  
řešilo

# Statická relokační při zavedení do paměti





# Statická relokalace

---

- Popsanému způsobu se říká statická relokalace
- Adresy se natvrdo přepíše správnými
- Např. IBM OS/360 MFT

# Mechanismy ochrany paměti

---

dále budou popsány dva odlišné mechanismy ochrany paměti:

- mechanismus přístupového klíče
- mechanismus báze a limitu

Uvidíme, že mechanismus báze a limitu se používá i při dynamické relokaaci

# Ochrana – přístupový klíč

---

- Problém - proces mohl zasahovat do paměti jiných procesů
- IBM 360 – **přístupový klíč**
  - Paměť rozdělena do bloků 2KB
  - Každý blok – sdružený hw 4 bitový kód ochrany
  - PSW procesoru obsahuje 4 bitový klíč
  - Při pokusu o přístup k paměti jejíž kód ochrany se liší od klíče PSW – výjimka
  - Kód ochrany a klíč může měnit jen OS (privilegované instrukce)
  - Výsledek – ochrana paměti

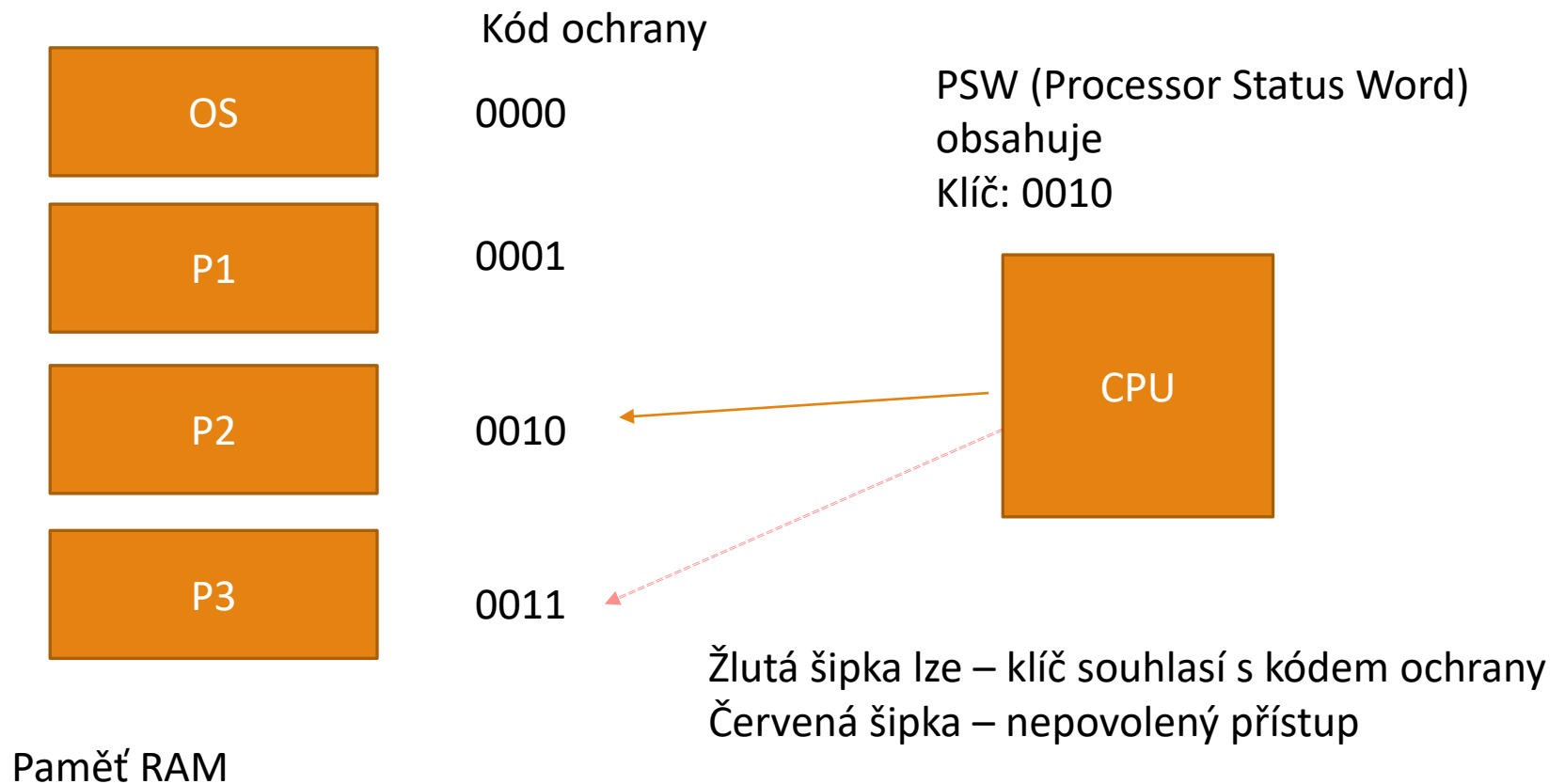


Klíč je  
spjatý s  
procesem

PSW označuje stavový registr u IBM 360

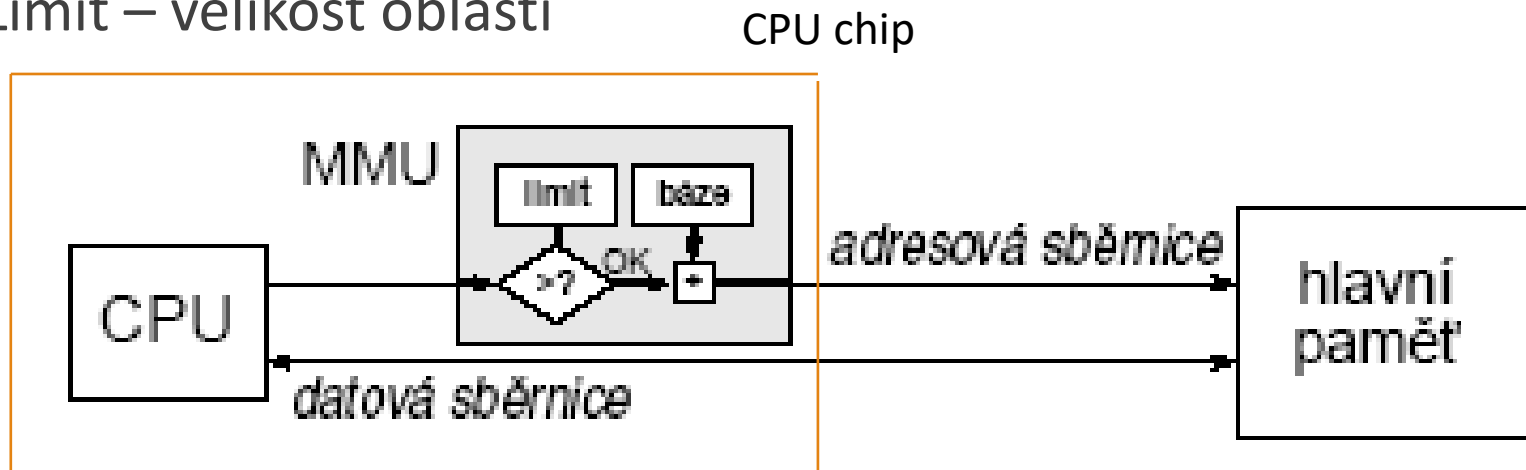


Možnou metodou ochrany paměti  
je ochrana přístupovým klíčem



# Ochrana - mechanismus báze a limitu

- Jednotka správy paměti MMU (je uvnitř CPU!!!)
- Dva registry – **báze** a **limit**
- Báze – počáteční adresa oblasti
- Limit – velikost oblasti



Zde je MMU jen pro názornost mimo CPU, jinak je jeho součástí

# Mechanismus báze a limitu

---

- Funkce MMU (uvnitř CPU)
  - převádí adresu používanou procesem na adresu do fyzické paměti
  - Nejprve zkontroluje, **zda adresa není větší než limit**
    - **Ano** – výjimka, **Ne** – k adrese přičte bázi
- Pokud báze 1000, limit 60:
  - Přístup na adresu 55 – **ok, výsledek 1055**
  - Přístup na adresu 66 – není ok, **výjimka**
- Zajistí nejen ochranu, ale i dynamickou relokaaci

# Dynamická relokalace

---

- Provádí se **dynamicky za běhu**
- Lze použít mechanismus báze a limitu
- Nastavení **báze a limitu** může měnit pouze OS (privilegované instrukce)
  
- Např. dříve 8086
  - slabší varianta (nemá limit, jen bázi)
  - báze registry = segmentové registry **DS,SS,CS,ES**

# poznámka

---

Mechanismus báze a limitu nám vyřešil  
jak dynamickou relokaaci, tak i ochranu paměti

- můžeme zavést program od různé fyzické adresy (relokace)
- můžeme zajistit, aby proces nepřistupoval mimo svůj povolený rozsah paměti (ochrana)



# Co dělat, pokud se **všechny** spuštěné procesy nevejdou do paměti? (!!)

---

## 2 strategie

- **Odkládání celých procesů** (swapping)
  - Nadbytečný proces se odloží na disk
  - Daný proces se ale stále celý vejde do fyzické paměti
  - Např. UNIX Version 7
- **Virtuální paměť** - překrývání, stránkování, segmentace
  - V paměti nemusí být procesy celé
  - **Překrývání (overlays)**
  - Virtuální paměť se stránkováním, segmentací

---

Správa paměti s odkládáním celých procesů

(Proces se vejde do fyzické paměti)

# Odkládání celých procesů

---

co víme o velikosti procesu?

- data procesu mohou **růst**
- pro proces alokováno **o něco více** paměti, než je třeba
- potřeba více paměti, než je alokováno:
  1. **přesunout** proces do větší oblasti (díry)
  2. překážející proces **odložit** – prostor pro růst procesu
  3. **odložit** žadatele o paměť, dokud nebude prostor
  4. proces **zrušit** (odkládací paměť je plná)

# Odkládání celých procesů

---

- proces – dvě rostoucí části
  - data, zásobník (co se kde alokuje?)
  - možnost rozrůstání **proti sobě**
  - překročení velikosti – přesun, odložit, zrušit

# Alokace odkládací oblasti

---

tj. jak vyhradit prostor pro proces na disku:

- na celou dobu běhu programu („pořád do stejného místa“)
- alokace při každém odložení
- stejné algoritmy jako pro přidělení paměti
- velikost oblasti na disku
  - násobek alokační jednotky disku

# Virtuální paměť

---

- mechanismus překrývání (overlays)
  - Starší, umožnil běh „velkého“ programu
- Stránkování, segmentace
  - Dnes používané
  - Nejčastěji stránkování

Virtuální paměť je to, co se dnes nejčastěji používá.

# Překrývání (overlays)

---

- **program** – rozdělen na **moduly**
- **start** – spuštěna část 0, při skončení zavede část 1 ...
- **zavádění modulů**
  - více překryvných modulů + data v paměti současně
  - moduly zaváděny dle potřeby (nejen 0,1,2,..)
  - mechanismus odkládání (jako odkládání procesů)
- kdo zařizuje zavádění modulů?
- kdo navrhuje rozdělení dat na moduly?

# Překrývání

---

- zavádění modulů **zařizuje OS**
- **rozdělení** programů i dat na části – navrhuje **programátor**
  - vliv rozdělení na výkonnost, komplikované
  - pro každou úlohu nové rozdělení
  - proto je raději snaha, aby se o vše postaral OS
- příklad – overlay.pas
  - Hlavní modul je v paměti zavedený a dále se v ní střídají moduly 1 a 2



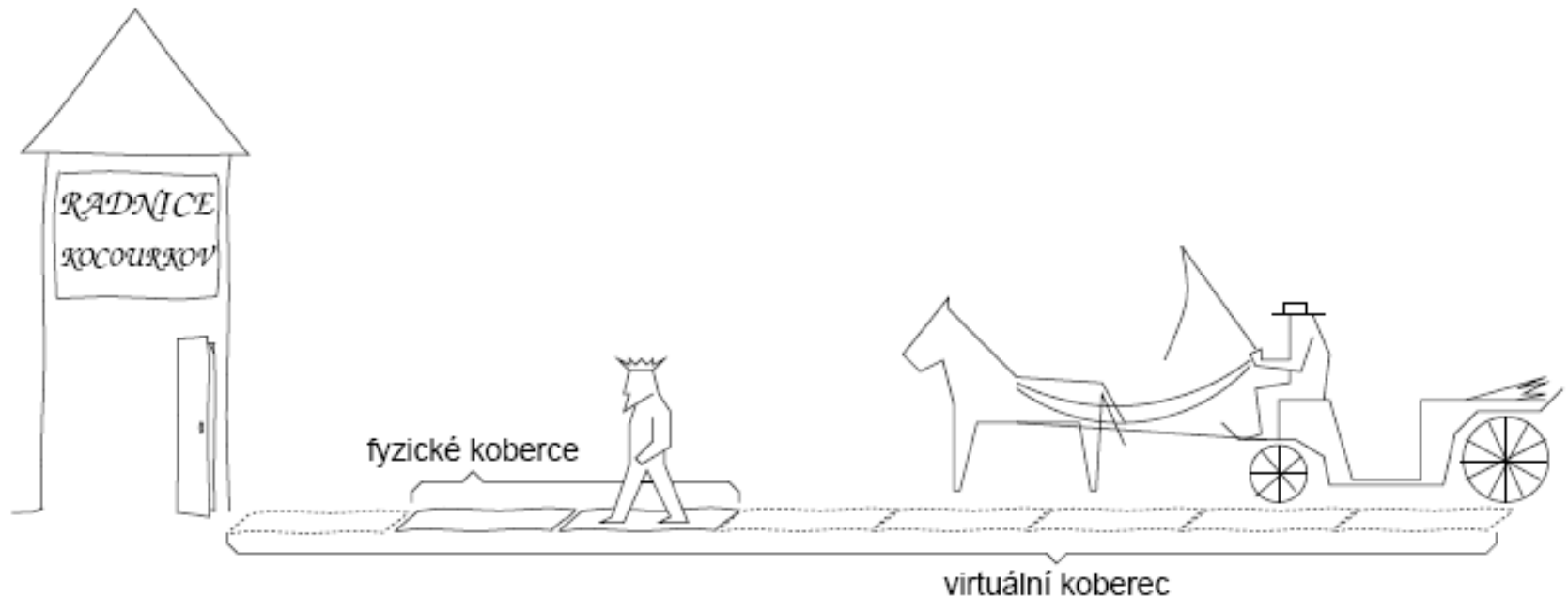
# Virtuální paměť

---

- potřebujeme rozsáhlý adresový prostor
- ve skutečné paměti je **pouze část** adresového prostoru
  - jinak by to bylo příliš drahé
- zbytek může být odložen na disku
- **kterou část mít ve fyzické paměti?**
  - tu co právě potřebujeme 😊

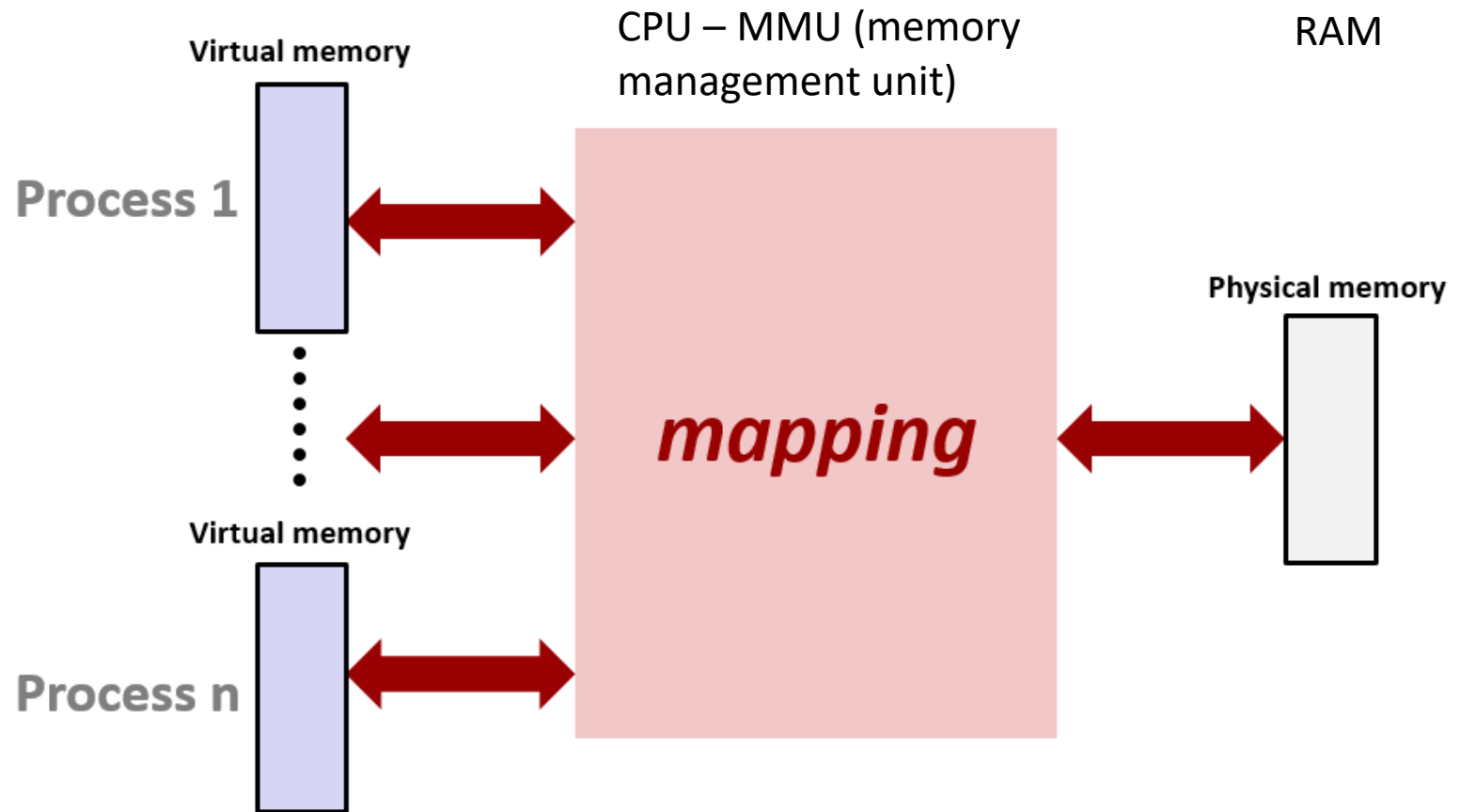
# Historie – královský koberec

---

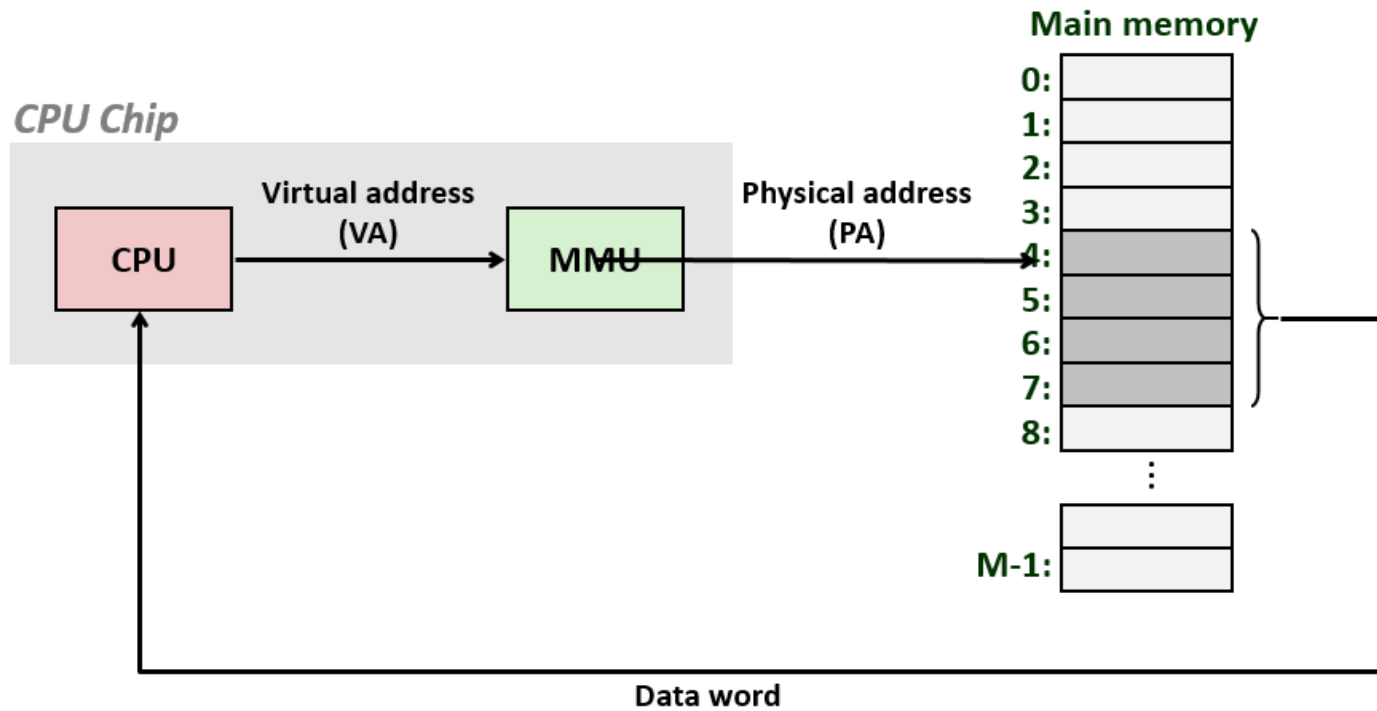


Na pokrytí celé cesty stačí pouze dva fyzické koberce

# Virtuální adresy



# Vztah virtuální a fyzické adresy



# Virtuální adresy

---

- fyzická paměť RAM slouží jako cache virtuálního adresního prostoru procesů (!)
- procesor – používá virtuální adresy u procesů
- Pokud požadovaná část VA Prostoru **JE** ve fyzické paměti
  - MMU převede  $VA \Rightarrow FA$ , přístup k paměti
- Pokud požadovaná část **NENÍ** ve fyzické paměti
  - OS ji musí načíst z disku do RAMky
  - I/O operace – přidělení CPU jinému procesu
- většina systémů virtuální paměti používá stránkování

# Mechanismus stránkování (paging)

---

- program používá virtuální adresy  
(virtuální adresa obsahuje **číslo stránky** a **offset**)
- Musíme rychle zjistit, zda je požadovaná adresa v paměti
  - ANO – převod VA => FA
  - NE – je třeba zavést z disku do paměti
- co nejrychlejší – děje se při každém přístupu do paměti

# Pojmy – důležité !!!

---

- VAP – stránky (pages) pevné velikosti
  - nejčastěji 4KB, další běžné: 2MB, 4MB a 1GB
- fyzická paměť – rámce (page frames) stejné velikosti jako stránky
- rámec může obsahovat PRÁVĚ JEDNU stránku
- na známém místě v paměti – tabulka stránek
  - Každý proces má svojí tabulku stránek
  - Hodnota registru CR3 CPU ukazuje na tabulku stránek
- tabulka stránek poskytuje mapování virtuálních stránek na rámce

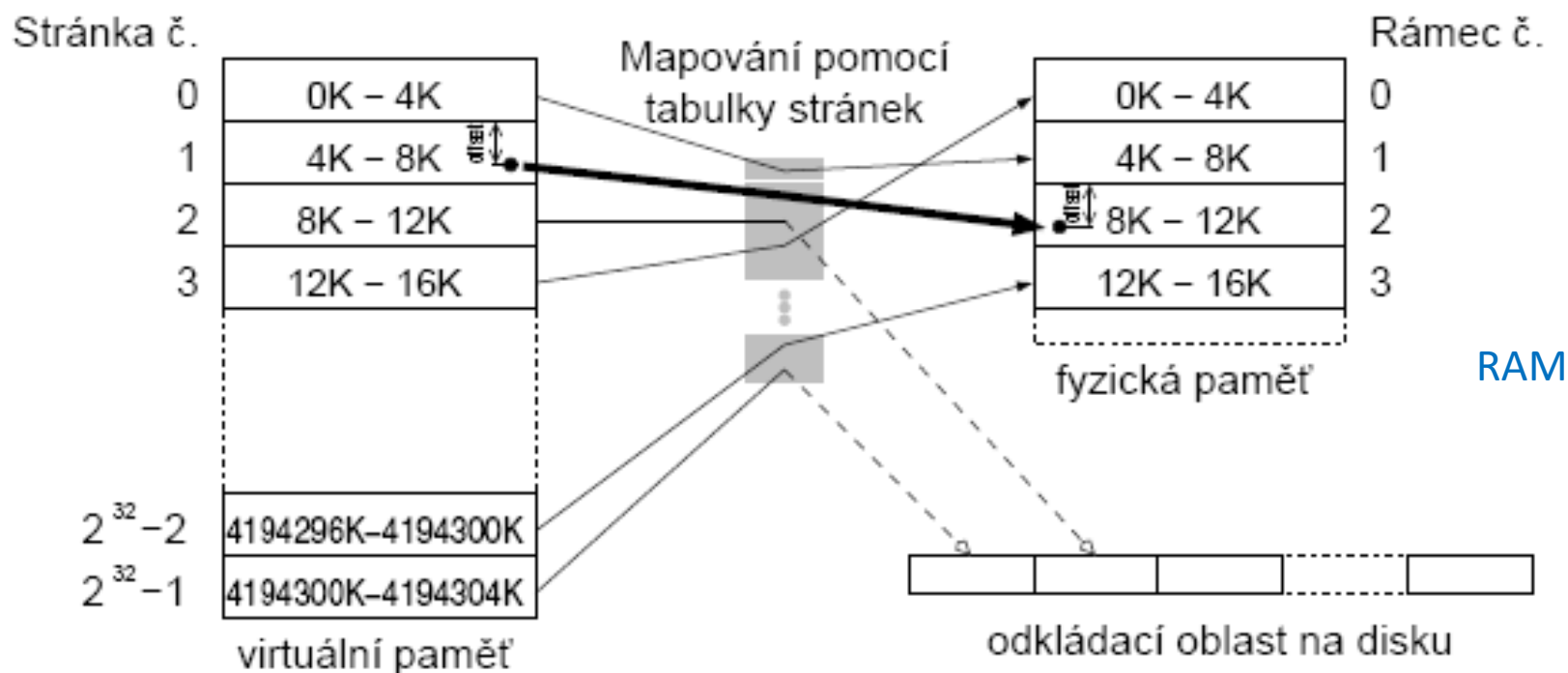
# Shrnutí pojmů

---

- virtuální adresní prostor
- fyzický adresní prostor
- procesy používají VA nebo FA?
- co dělá MMU?
- k čemu slouží tabulka stránek?
- stránka
- rámec



Stránky jsou mapovány na rámce v RAM, nebo jsou uloženy v odkládací paměti na disku



Proces P1

Disk – swapovací partition nebo swap soubor

# Stránkovaná paměť

Swap na disku



P1

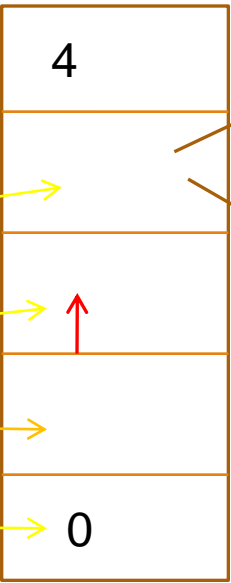
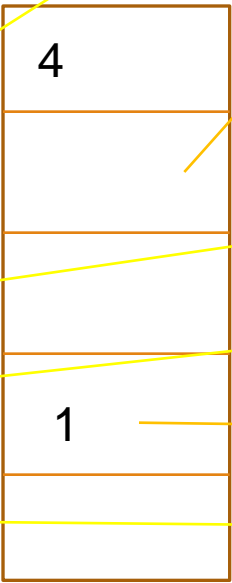
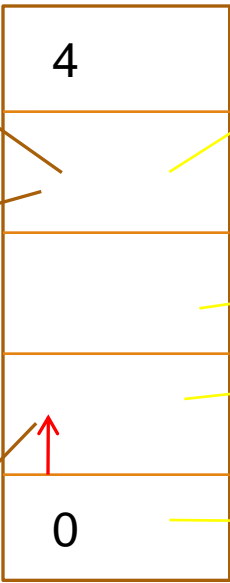
P2

RAM

virtuální adresy

stránka např. 4KB

Offset od začátku stránky



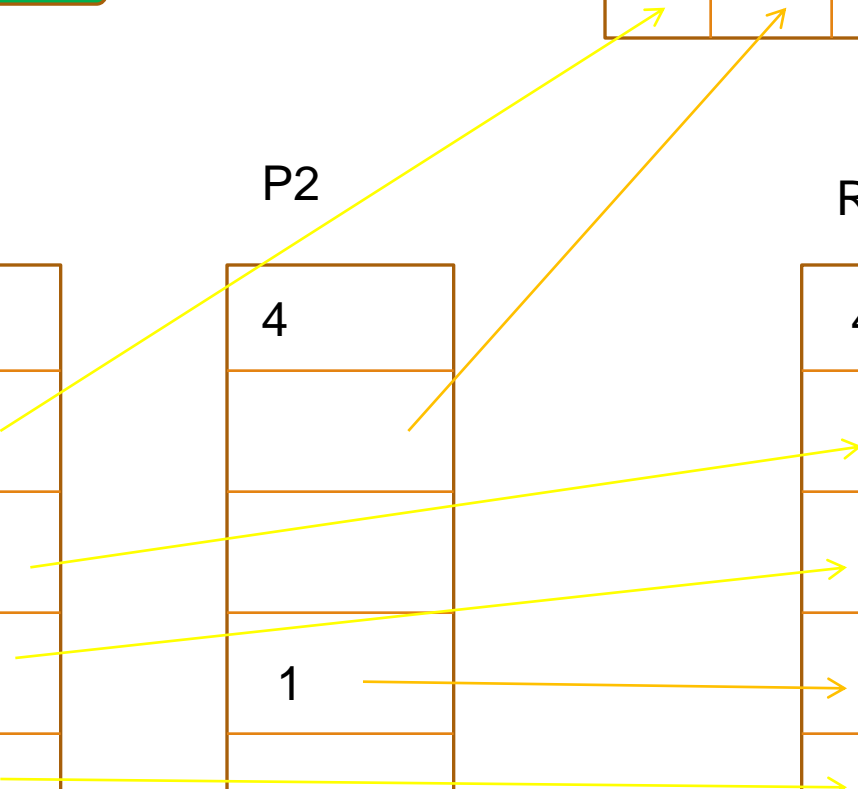
fyzické adresy

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(!!)

Číslo stránky (index)	Číslo rámce	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 (!!!)

---

- **číslo rámce**

- Udává, v kterém rámci v RAM je stránka uložena

- **příznak platnosti**

- Říká, zda je daná stránka v RAM nebo není (Present / absent bit)
- V některých systémech – 2 bity (present/absent zda se vůbec používá a druhý bit zda je v RAM nebo není, Intel používá jeden bit a při page fault se rozhodne)
- Pokud není bit nastaven a přistupujeme na ní – page fault a řeší se co dále (je ve swapu, není vůbec namapovaná, ..)

- **příznak ochrany**

- Zda je stránka jen pro čtení, nebo i pro zápis (případně lze z ní vykonávat kód)
- Příklad – data rw, kód rx

- **bit modifikace**

- Pokud stránka byla modifikována (zápis), znamená to, že pokud je i ve swapu, tak tam je nyní neaktální – při odložení z paměti do swapu je třeba znovu zapsat

# Tabulka stránek

---

- **bit referenced**

- Zda byla stránka v nedávné době přístupována či ne
- Slouží pro page replacement algoritmy (nevyhodit z RAM stránku, co byla nedávno použita)

Můžou být další údaje, např.:

- **bit caching**

- Povolení / zákaz kešování dané stránky

- **adresa ve swapu**

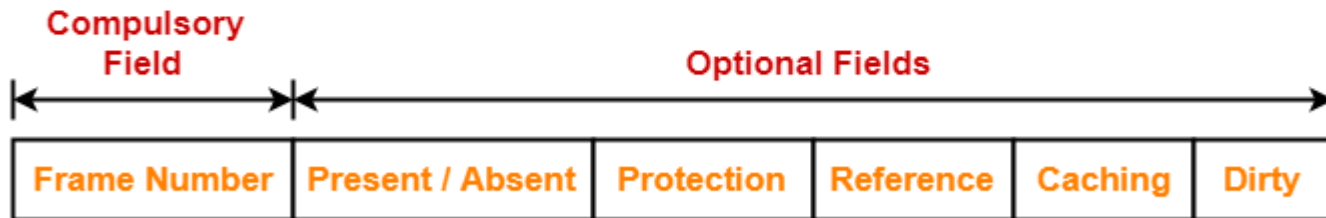
- Někde musí být uložena informace, jak danou stránku ve swapu najít

# Poznámka

## Page Table Entry-

- A page table entry contains several information about the page.
- The information contained in the page table entry varies from operating system to operating system.
- The most important information in a page table entry is frame number.

In general, each entry of a page table contains the following information-



**Page Table Entry Format**

zdroj, další podrobné informace:

<https://www.gatevidyalay.com/tag/page-table/page/4/>

# Poznámka 2

---

You may notice that in the Intel example, there are no separate valid and present bits, but rather just a present bit (P). If that bit is set ( $P=1$ ), it means the page is both present and valid. If not ( $P=0$ ), it means that the page may not be present in memory (but is valid), or may not be valid. An access to a page with  $P=0$  will trigger a trap to the OS; the OS must then use additional structures it keeps to determine whether the page is valid (and thus perhaps should be swapped back in) or not (and thus the program is attempting to access memory illegally). This sort of judiciousness is common in hardware, which often just provide the minimal set of features upon which the OS can build a full service.

Zdroj: <http://pages.cs.wisc.edu/~remzi/OSTEP/vm-paging.pdf>



# Tabulka stránek (TS)

---

- velikost záznamu v tabulce stránek .. 32 bitů
  - číslo rámce .. 20 bitů
- Tabulka stránek je součástí PCB (záznamu v tabulce procesů)
  - PCB je řádka v tabulce procesů, obsahující info o procesu
  - PCB obsahuje informaci, kde leží tabulka stránek procesu

# Výpočet adresy - stránkování

---

Pojmy:

VA	virtuální adresa
FA	fyzická adresa
str	číslo stránky
offset	offset
ramec	číslo rámce

Dále předpokládáme velikost stránky 4096B

# Příklad s uvedením výpočtu

---

Je dána  $VA(p1) = 100$ . Určete FA.

Velikost stránky je 4096 bytů (4KB).

Tabulka stránek procesu p1 je následující:

Číslo stránky	rámec
0	1
1	2
2	---
3	0

Nezapomeň: máme-li více procesů, každý má svojí tabulku stránek.

# Výpočet adresy – stránkování

---

1. Virtuální adresu rozdělíme na číslo stránky a offset
  - $Str = VA \div 4096$  (celočíslné dělení, 4096 je velikost stránky)
  - $Offset = VA \bmod 4096$  (zbytek po dělení)
2. Převod pomocí tabulky stránek  
převedeme číslo stránky na **číslo rámce**
  - **$tab\_str[0] = 1$**  (pro stránku 0 je číslo rámce 1)
  - $tab\_str[1] = 2$
  - $tab\_str[2] = --$  stránka není namapována
  - $tab\_str[3] = 0$
  - Pro  $VA = 100$  je stránka 0, offset 100 => tedy rámec 1

# Výpočet adresy - stránkování

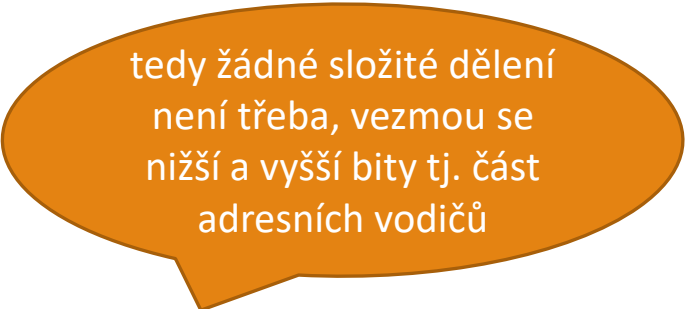
---

3. Z čísla rámce a offsetu sestavíme fyzickou adresu:

$$FA = \text{ramec} * 4096 + \text{offset}$$

$$FA = 1 * 4096 + 100$$

$$FA = 4196 \text{ v daném případě}$$



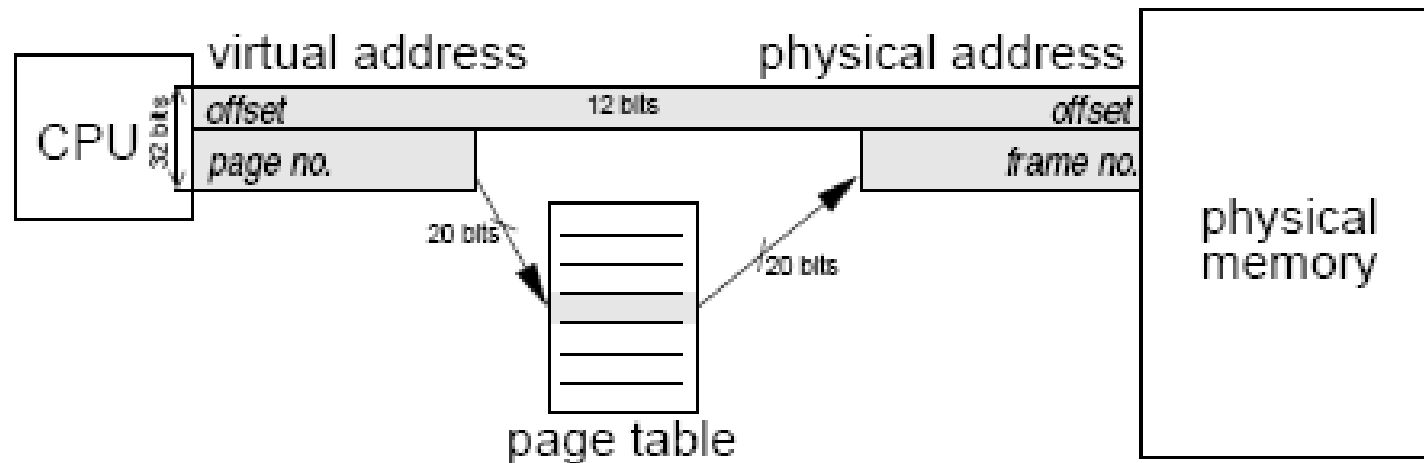
tedy žádné složité dělení  
není třeba, vezmou se  
nižší a vyšší bity tj. část  
adresních vodičů

V reálném systému dělení znamená rozdělení na vyšší a nižší bity adresy (proto mocnina dvou velikost str.)

Nižší bity – offset

Vyšší bity – číslo stránky

# Stránkování



32 bit adresa – 20 bitů číslo stránky, 12 bitů offset

Offset zůstává beze změny

MMU je součást CPU (tj. i manipulace s page table)

# Výpadek stránky (!!!)

---

- viz příklad, pro adresu 8192 -> str 2, offset 0
- Výpadek stránky
  - Stránka není mapována
  - Výpadek stránky způsobí **výjimku**, zachycena OS (pomocí **přerušení**)
  - OS **iniciuje zavádění** stránky a **přepne na jiný** proces
  - Po zavedení stránky OS upraví mapování (tabulku stránek)
  - Proces může pokračovat
  - Vyřešit: KAM stránku zavést a ODKUD ?

# Výpadek stránky

---

1. Pokud daná stránka procesu není namapována na určitý rámec ve fyzické paměti a chceme k ní přistoupit

dojde k **výpadku stránky** – vyvolání **přerušení** operačního systému.

2. Operační systém se postará o to, aby danou **stránku** zavedl do nějakého **rámce** ve fyzické paměti, nastavil mapování (upravil tabulku stránek) a poté může přístup proběhnout.



# Náročnost

---

- Velký rozsah tabulky stránek
  - Např. 1 milion stránek, ne všechny obsazeny
- Rychlý přístup
  - Nemůžeme pokaždé přistupovat k tabulce stránek
  - Různá HW řešení, kopie části tabulky v MMU -> TLB cache

Tabulka stránek může být velmi rozsáhlá – pro urychlení např. kopie části tabulky stránky v MMU (memory management unit)

# Vnější fragmentace

---

## Vnější / externí

- Zůstávají nepřidělené (nepřidělitelné) úseky paměti
- Např. dynamické přidělování viz dříve – malé díry

Při **stránkování vnější fragmentace nenastává**, všechny stránky jsou přidělitelné (jsou stejně velké), ale nastává fragmentace vnitřní.

# Vnitřní fragmentace

---

## Vnitřní fragmentace

- Část **přidělené** oblasti je **nevyužita**  
(dostaneme přidělenou stránku, ale využijeme z ní jen část !)

Stránkování:

Při **stránkování** nastává **vnitřní** fragmentace.  
V průměru polovina poslední stránky procesu je prázdná.

# Stránkování - poznámky

---

- **Čisté** stránkování - bez odkládací oblasti (swapu) !!
- Častější je ale stránkování i s využitím swapu
- Souvislý logický adresní prostor procesu  
**mapován do** nesouvislých částí paměti RAM
- OS udržuje:
  - 1 tabulka rámců
  - Tabulku stránek pro každý proces



# Tabulka rámců

---

Pro správu FYZICKÉ paměti RAM

Je třeba informace, které rámce jsou  
volné vs. obsazené

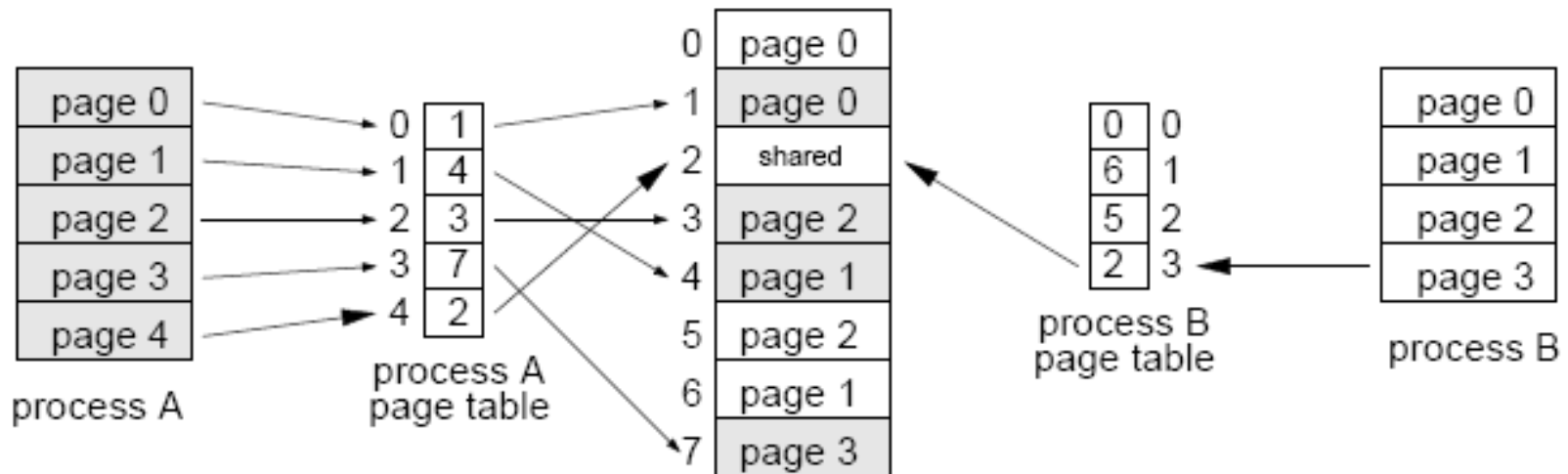


# Tabulka stránek procesu

---

- Mapuje číslo stránky na číslo fyzického rámce
- Další informace – např. příznaky ochrany
- Řeší problém relokační a ochrany
  - Relokace – mapování VA na FA
  - Ochrana – v tabulce stránek uvedeny **pouze** stránky, ke kterým má proces **přístup** (-> jinde se nedostane)
- Přepnutí na jiný proces
  - MMU přepne na jinou tabulku stránek

# Stránkování a sdílená paměť



Stránkování umožňuje i přístup do **sdílené paměti**, v každém procesu může být dokonce sdílená paměť mapována **od jiné adresy**

# Problémy

---

Velikost tabulky stránek

- Pomůže víceúrovňová struktura

Rychlost převodu VA -> FA

- TLB (Translation Look-aside Buffer)

na dalších slidech budou tyto problémy dále rozebrány

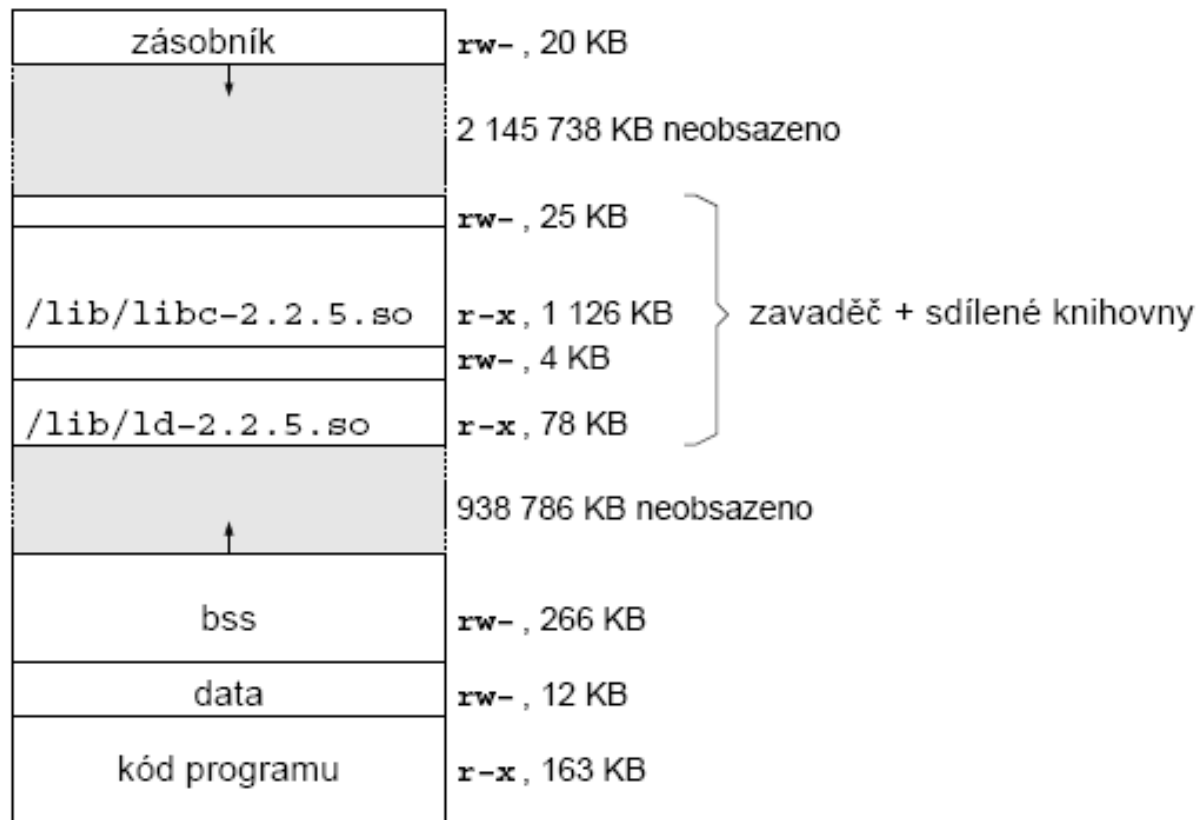


# Velikost tabulky stránek

---

- VA 32 bitů
  - 12bitů ofset (pozice ve stránce – stránka 4KB)
  - 20 bitů číslo stránky (převáděné na číslo rámkce)
    - Stránek je  $2^{20}$  (cca přes milion)
    - Každá položka zabírá 4B ..  $(2^{20}) * 4B = 4MB$  zabírá pro **každý** proces
- Proces využívá jen část prostoru VA
  - Kód
  - Data (inicializovaná, a neinicializovaná)
  - Sdílené knihovny a jejich data
  - Od nejvyšší adresy zásobník - roste dolů

# Rozdělení paměti pro proces



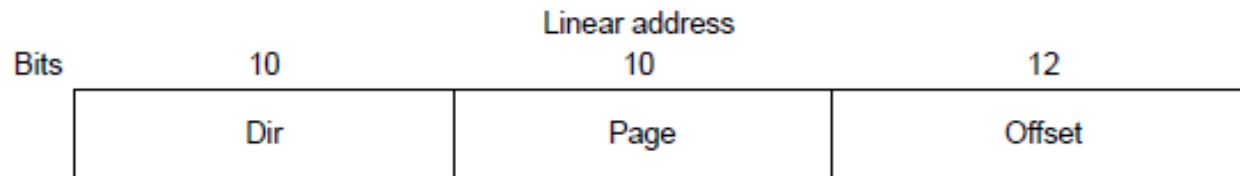
Obrázek převzatý z literatury

# Víceúrovňová tabulka stránek

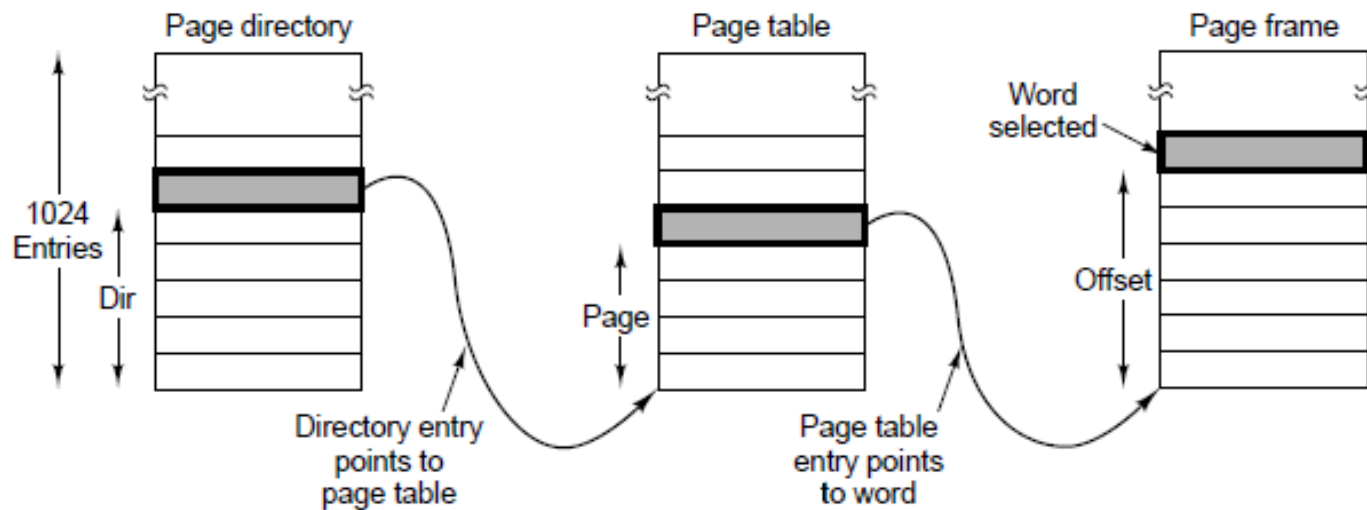
---

- Mít v tabulce stránek jen ty, představující využívanou paměť  
=> víceúrovňová tabulka stránek
- VA 32 bitů
  - PT1 – 10 bitů , index do tab. stránek 1. úrovně
  - PT2 – 10 bitů, index do tab. stránek 2. úrovně
  - Offset – 12bitů
- PT1=0 (kód a data), PT1=1 (sdílené knihovny)  
PT=1023 (zásobník); ostatní nepřirazeno!

# Víceúrovňová tabulka stránek



(a)



(b)

# Rychlost převodu (!)

---

- Každý přístup do paměti – sáhne do tabulky stránek
  - 2x více paměťových přístupů
    - musíme sáhnout do tabulky stránek a pak do paměti kam chceme
- **TLB (Translation Look-aside Buffer) (!!!!)**
  - HW cache
  - Vstup: číslo\_stránky
  - Výstup: číslo\_rámce nebo odpoví, že neví
  - Dosáhneme zpomalení jen 5 až 10 %
  - Přepnutí kontextu na jiný proces
    - problém (vymazání cache,..)
    - než se TLB opět zaplní a tedy naučí mapování – pomalý přístup

# Obsah položky v tabulce stránek (!!!)

---

- Číslo rámce
- Příznak platnosti (valid / invalid)
- Příznaky ochrany (rw, ro, ..)
- Bit modified (dirty)
  - zápis do stránky nastaví na 1
- Bit referenced
  - Přístup pro čtení / zápis nastaví na 1
- Další ...

# Invertovaná tabulka stránek

---

VA 64bitů , stránka 4KB,  $2^{52}$  stránek – moc

Invertovaná tabulka stránek

Položky pro každý fyzický rámec

- Omezený počet – dán velikostí RAM
- VA 64bitů, 4KB stránky, 256MB RAM – 65536 položek

Forma položky: (id procesu, číslo stránky)

# Invertovaná tabulka stránek - převod

---

Pokud je položka v TLB  
– zařídí HW, jinak OS (SW)

SW:

- Prohledávání invertované tabulky stránek
- Položka nalezena – (číslo stránky, číslo rámce) do TLB
- Tabulka hashovaná podle virtuální adresy (pro optimalizaci)



# Stránkování na žádost (využívá odkládací prostor)

---

## Vytvoření procesu

- Vytvoří prázdnou tabulku stránek
- Alokace místa na disku pro odkládání stránek
- Některé implementace – odkládací oblast inicializuje kódem programu a daty ze spustitelného souboru

## Při běhu

- Žádná stránka v paměti,
- 1. přístup – **výpadek stránky (page fault)**
- OS zavede požadovanou stránku do paměti
- Postupně v paměti tzv. **pracovní množina** stránek

# Pracovní množina stránek

---

Má-li proces svou **pracovní množinu stránek v paměti**, může pracovat **bez mnoha výpadků**

dokud se pracovní množina stránek **nezmění**, např.  
do **další fáze** výpočtu

Pracovní množina stránek daného procesu – kolik stránek musí mít ve fyzické paměti, aby mohl nějaký čas pracovat bez výpadků stránky

# Ošetření výpadku - scénář

---

Počítáme v uživatelském režimu, následuje instrukce přečti číslo na adrese 7000

MMU zjistí, že příslušná stránka procesu není v RAM. Vyvolá se přerušení **výpadek stránky**.

V tabulce stránek má informaci, že stránka leží ve swapu (a kde).

# Ošetření výpadku stránky (důležité !)

---

1. Výpadek – mechanismem **přerušení** (!! ) vyvolán OS
2. OS zjistí, pro kterou stránku nastal výpadek
3. OS určí umístění stránky na disku
  - Často je tato informace přímo v tabulce stránek
4. Najde rámec, do kterého bude stránka zavedena
  - Co když jsou všechny rámce obsazené?
5. Načte požadovanou stránku do rámce ( DMA přenos...)
6. Změní odpovídající mapovací položku v tabulce stránek
7. Návrat..
8. CPU provede instrukci , která způsobila výpadek

# Problém

---

Všechny rámce obsazené, kterou stránku vyhodit ??

->

Algoritmy nahrazování stránek

Všechny rámce v paměti RAM jsou plné. Přesto musíme nějaký z nich uvolnit (odložit na disk), abychom mohli do RAM dát ten, který potřebujeme. Jak rozhodnout, který rámec vyhodit?

# Algoritmy nahrazování stránek

---

- Uvolnit rámec pro stránku, co s **původní stránkou**?
- Pokud byla stránka modifikována ( $\text{dirty}=1$ ), uložit na disk
- Pokud modifikovaná nebyla a má již stejnou kopii na disku (swap), pouze uvolněna

# Algoritmy nahrazování stránek

---

**Kterou** stránku vyhodit?

Takovou, která se dlouho nebude potřebovat..

Chtělo by křišťálovou kouli...

(pohled do budoucnosti)



# Algoritmus FIFO

---

- Udržovat seznam stránek v pořadí, ve kterém byly zavedeny
- Vyhazujeme nejstarší stránku  
(nejdéle zavedenou do paměti – první na seznamu)
- Není nejvhodnější
- Často používané stránky mohou být v paměti dlouho  
(*analogie s obchodem, nejdéle zavedený výrobek – chleba*)
- Trpí Beladyho anomálií



# Beladyho anomálie

---

Předpokládáme:

Čím více bude rámců paměti, tím nastane méně výpadků.

Belady našel příklad pro algoritmus FIFO, kdy to neplatí.

\* algoritmus FIFO, řetězec odkazů (referencí): 0 1 2 3 0 1 4 0 1 2 3 4

3 rámce: ref.: 0 1 2 3 0 1 4 0 1 2 3 4

```
-----  
1| . 0 1 2 3 0 1 4 4 4 2 3 3  
2| . . 0 1 2 3 0 1 1 1 4 2 2  
3| . . . 0 1 2 3 0 0 0 1 4 4  
-----
```

P P P P P P P P P P = 9 výpadků

4 rámce: ref.: 0 1 2 3 0 1 4 0 1 2 3 4

```
-----  
1| . 0 1 2 3 3 3 4 0 1 2 3 4  
2| . . 0 1 2 2 2 3 4 0 1 2 3  
3| . . . 0 1 1 1 2 3 4 0 1 2  
4| . . . . 0 0 0 1 2 3 4 0 1  
-----
```

P P P P P P P P P P = 10 výpadků

\* tj. pro 3 rámce nastane 9 výpadků, pro 4 rámce 10 výpadků

\* objev pana Beladyho způsobil vývoj teorie stránkovacích algoritmů a jejich vlastností

# 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** (je to ta 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	1	1	1
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	0	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

Řádka 3: na 1

Sloupec 3: na 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

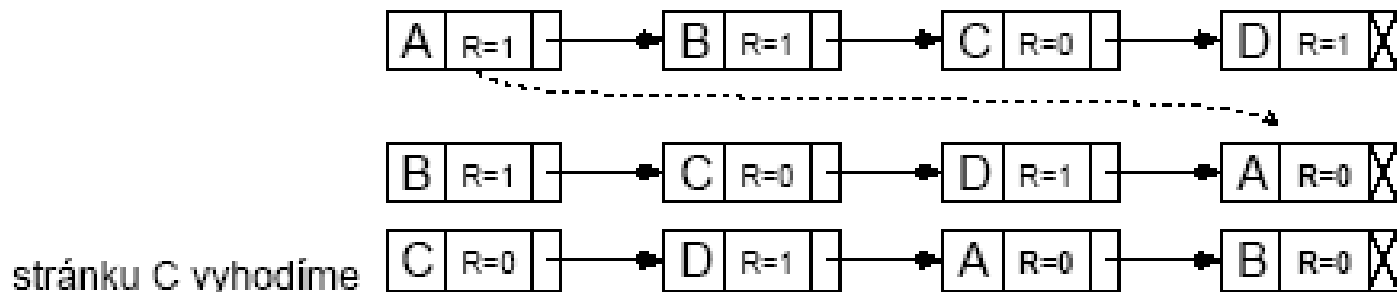
# Second Chance

---

- 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

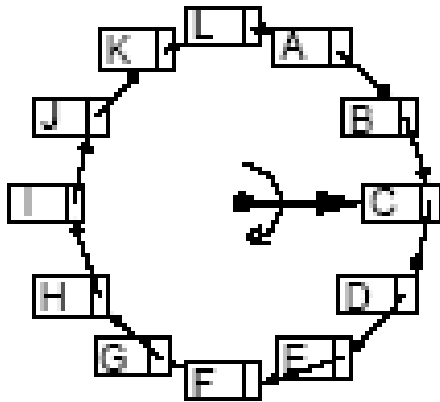
---

- 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

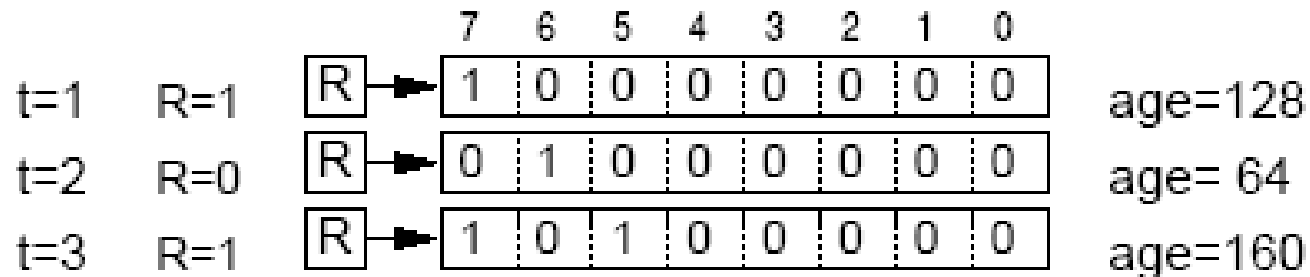
# SW aproximace LRU - Aging

---

- 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
- 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**

# Nahrazování stránek paměti

- FIFO + Beladyho anom.
  - MIN / OPT
  - LRU
  - NRU
  - Second Chance, Clock
  - Aging
- 

## 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?

# 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 vyhození 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

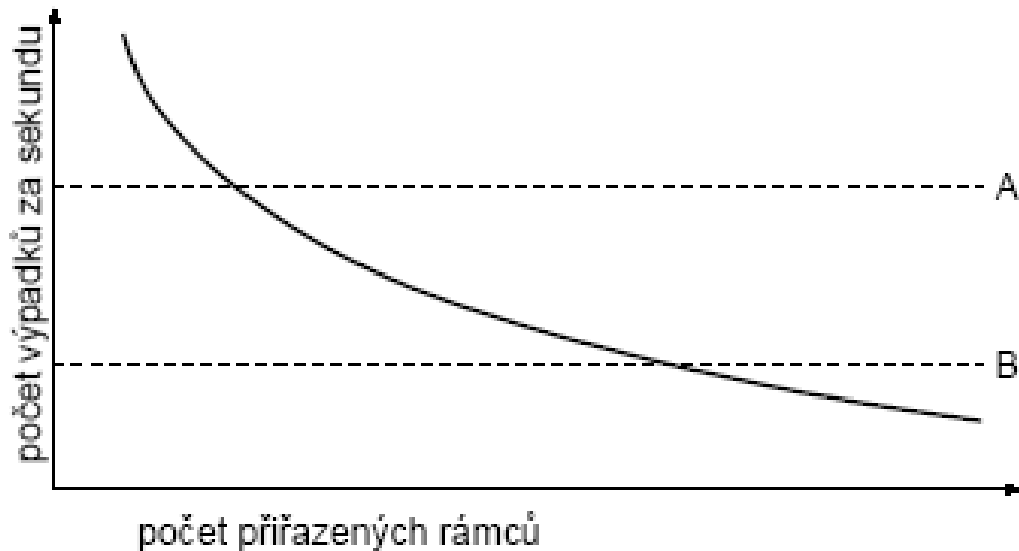
- Program se vzhledem k stránkování chová přibližně stejně při každém běhu

# Lokální alokace

---

- Kolik rámců dát každému procesu?
- **Nejjednodušší** – všem procesům dát stejně
  - Ale potřeby procesů jsou různé
- **Proporcionální** – každému proporcionální díl podle velikosti procesu
- **Nejlepší** – 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)



# Literatura

---

Základní principy OS včetně správy paměti:

<http://pages.cs.wisc.edu/~remzi/OSTEP/>