**BI-SPOL-17 Virtualizace paměti pomocí stránkování, principy překladu logických adres na fyzické, algoritmy pro nahrazování stránek**

BI-OSY

#### Logická organizace paměti

**Virtuální adresový prostor (VAS)**

* Každý proces má svůj VAS, který se mapuje do fyzické paměti
* Jakým způsobem bude VAS procesu mapován do fyzické paměti závisí na konkrétní architektuře CPU a použitém OS
* VAS typicky obsahuje oblasti pro následující struktury  
  Obsah obrázku text

  Popis byl vytvořen automaticky
* Při rozmístění struktur ve VAS a při alokaci fyzické paměti je nutné pamatovat na to, že některé struktury (halda, zásobník) mění svoji velikost během existence procesu
* Některé struktury ve VAS mohou být sdílené mezi procesy (TEXT, knhovny, …)
* *Address space layout randomization (ASLR)*: z důvodu bezpečnosti (např. útok buffer-overflow) mohou být struktury ve VAS umisťovány na náhodné adresy
* Stránkování je jednou z možností implementace

**Chceme vyřešit následující problém**

* Fyzická paměť je po určitém čase fragmentovaná (obsahuje mnoho malých volných oblastí) – problém najít dostatečně velkou souvislou oblast pro nový proces
* Např. když v buddy systému mám malé zabrané oblasti a uvolním každý lichý, aby to nešlo mergnout.

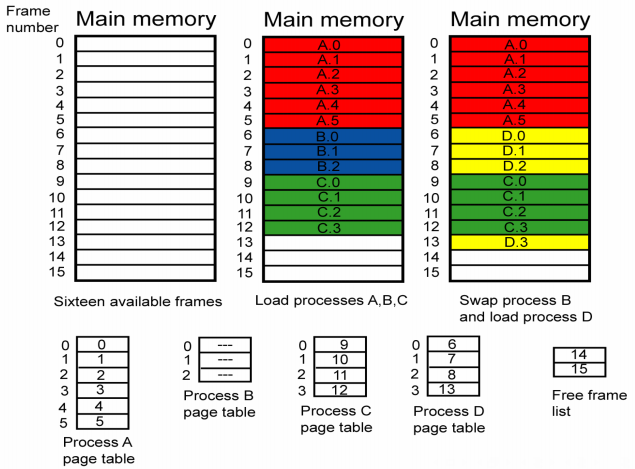
*Řešení*: Virtuální adresový prostor (VAS) budeme alokovat jako množinu malých oblastí, tím odpade problém s nalezením velké souvislé oblasti ⇒ **stránkování**

### 

### Virtualizace paměti pomocí stránkování

**Stránkování**

* fyzická (hlavní) paměť je rozdělená na úseky stejné velikosti – **rámce** (frames, např. 4kB)
* VAS je rozdělený na úseky stejné velikosti – **stránky** (pages)
* **velikost rámce = velikost stránky**
* jednotlivé bloky stránky VAS se nahrávají do volných rámců fyzické paměti
* OS si musí pamatovat
  + rámce přidělené jednotlivým procesům (např. pomocí tabulky stránek)
  + volné rámce



*Jak funguje stránkování*

#### Virtuální paměť

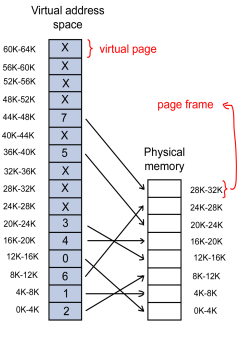
Příklad

* ve 32-bitovém OS, jeden 32-bitový proces může adresovat až B = 4 GB
* V 64-bitovém OS, jeden 64-bitový proces může teoreticky adresovat až B = 16 EB
* Většina procesů ale reálně používá pouze zlomek prostoru ze svého VAS

Problém

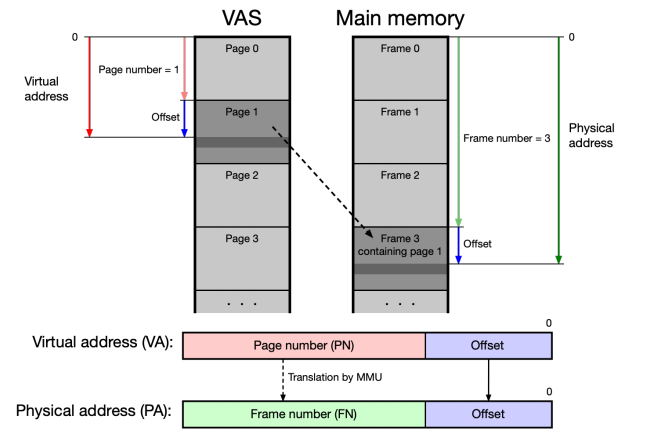
* VAS jednoho procesu může být větší než instalovaná fyzická paměť systému
* OS umožňuje současně spustit až tisíce procesů → součet velikostí jednotlivých VAS je opět větší než instalovaní fyzická paměť systému

**Řešení**

* Virtuální paměť
* VAS každého procesu je automaticky rozdělen na menší kousky a ve fyzické paměti musí být pouze kousky aktuálně používané, zbytek používaného VAS je na disku
* proces je rozdělen na menší kousky, ve fyzické paměti jsou jen ty aktuálně používané a zbytek VAS je na disku
* tzn. OS poskytuje virtuální paměť, která může přesahovat fyzické rozměry RAM, ale ta si bude držet jen malé kousky, zbytek není používán
* většinou kombinována se stránkováním

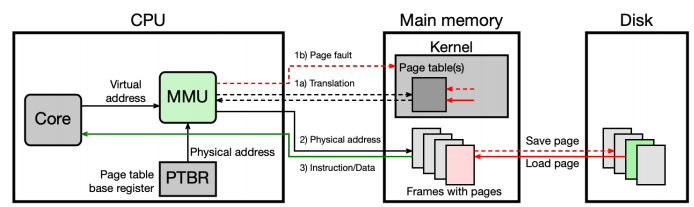
#### Virtuální paměť se stránkováním

* většinou je virtuální paměť kombinována se stránkováním
* **Princip**
  + proces používá adresy (**virtuální adresy**), které adresují virtuální adresový prostor
  + VAS je rozdělen na stejně velké souvislé oblasti (**virtuální stránky**). V závislosti na architektuře CPU je minimální velikost 4/8 KB
  + korespondující úseky ve fyzické paměti stejné velikosti jsou nazývány **rámce stránek** (page frames)
  + v hlavní paměti musí být aspoň stránky aktuálně používané



**Memory management unit (MMU)**

* MMU společně s OS zajištuje, že **proces má iluzi, že celý jeho VAS je v hlavní paměti** a pro adresaci instrukcí/dat používá virtuální adresy (vztažené k začátku VAS)  
  (např. když máme VAS 4GB a RAM 1GB, program může využít celý 4GB VAS, ale některá data/instrukce budou uložena na disku)
* **překládá virtuální adresy na fyzické**
* Výpadek stránky (Page fault)
  + Pokud není virtuální stránka v hlavní paměti, MMU způsobí přerušení a „požádá“ OS o nahrání příslušné stránky do fyzické paměti (RAM)
  + OS nejdříve najde vhodný rámec fyzické paměti (pokud je nutné, uloží jeho obsah na disk), a pak do něj nahraje požadovanou virtuální stránku z disku nebo v něm „vytvoří“ novou stránku (zásobník, heap)
  + Pokud je RAM plná, musí se najít stránka, která se „vyhodí“ (uloží na disk) – pro vybrání vhodné stránky slouží algoritmy pro nahrazování stránek

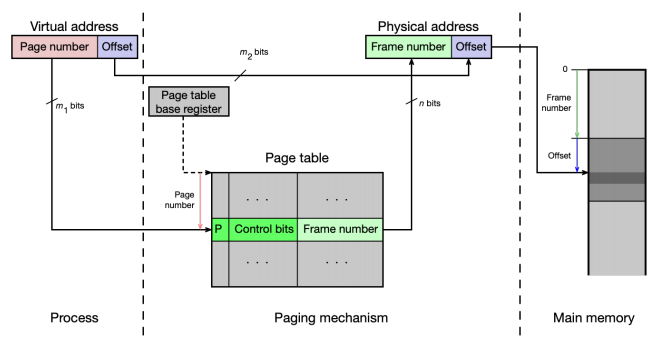


### Překlad logických adres na fyzické

* OS si musí udržovat informace o tom, do kterých rámců fyzické paměti se namapovaly jednotlivé stránky VAS jednotlivých procesů, a které rámce fyzické paměti jsou zatím volné
* Tuto informaci si OS udržuje v následujících strukturách, které jsou závislé na ISA použitého procesoru
  + Tabulka stránek
  + Víceúrovňová tabulka stránek
  + Invertovaná tabulka stránek
* pro urychlení překladu pak MMU využívá **Translation lookaside buffer** (TLB), ve kterém jsou informace o naposledy překládaných virtuálních adresách

#### Tabulka stránek

* tabulka obsahuje pro každou stránku VAS daného procesu jednu řádku, ve které jsou uložené následující informace
  + **číslo rámce**, do kterého se tato stránka namapovala
  + **kontrolní bity**
    - present bit (P): informace, zda stránka je v hlavní paměti
    - reference bit (R): informace, zda se ke stránce přistupovalo
    - modify bit (M): informace, zda byl obsah stránky modifikován
    - přístupová práva
    - cache disable/enabled: zda jsou registry periferií mapovány přímo do paměti
    - Read/write (R/W): informace, zda lze stránku modifikovat
    - User/supervisor (U/S): informace, zda lze na stránku přistupovat v uživatelském módu, …
* číslo stránky (page number) funguje jako **index** do této tabulky
* OS si musí udržovat **pro každý proces jednu tabulku**



**Příklad**

Obsah obrázku text

Popis byl vytvořen automaticky

32-bitový CPU – znamená, že používáme 32 bitů pro adresaci VAS – tedy VAS = B

4KB stránka = B – stránka nám zabírá 12 bitů

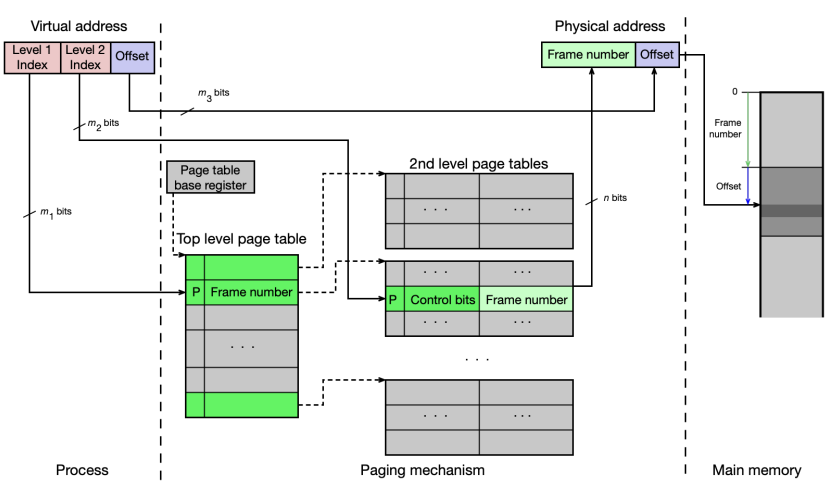
Offset = definuje pozici bajtu uvnitř stránky (případně rámce)

**Problémy**

* Přestože proces používá jen zlomek místa ze svého VAS, tak tabulka stránek obsahuje informaci i o nepoužitých stránkách.
* Pro každý proces musí OS držet jednu tabulku stránek
* **→ Plýtvání pamětí**
  + Řešením je víceúrovňová tabulka stránek

#### Víceúrovňová tabulka stránek

* odstraňuje nevýhodu tabulky stránek, umožňuje držet v paměti pouze informace o používaných stránkách → u procesu, které alokují málo paměti, můžeme ušetřit paměť
* Pro *n* úrovňovou tabulku stránek platí
  + Virtuální adresa se skládá z n indexů, které ukazují do tabulek jednotlivých úrovní, a offsetu
  + Fyzická adresa se skládá z čísla rámce a offsetu
  + Tabulky stránek úrovní 1,…,n-1 obsahují „present bit“ a číslo rámce, ve kterém je uložena/začíná tabulka následující úrovně
  + Tabulka stránek úrovně n obsahuje „present bit“ a číslo rámce, ve kterém je uložena samotná virtuální stránka
* V hlavní paměti je vždy „Top level tabulka“ (tabulka úrovně 1)
* Tabulky ostatních úrovní v paměti být nemusí, pokud proces nepoužívá stránky z oblastí VAS, která tyto tabulky pomáhají adresovat → šetří se fyzická paměť
* dělají se vlastně tabulky tabulek (vyplatí se až do 4. úrovně)
* ušetřené místo jde na úkor pomalejšího překladu (pro urychlení překladu se používá spolu s TLB)
* virtuální adresa je nasekaná na několik částí a každá úroveň určuje pozici v jedné úrovni tabulek, v poslední úrovni je opět číslo rámce a dopočítá se pozice ve fyzické paměti
* první úroveň (Top level page table) je přítomná vždy, ostatní podle toho, jak jsou využívány (to je to ušetření paměti)



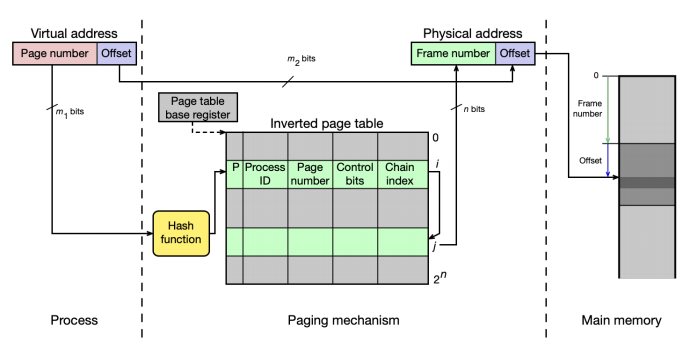
**Příklad**

**Obsah obrázku text

Popis byl vytvořen automaticky**

#### Invertovaná tabulka stránek

* tabulka obsahuje pro každý rámec fyzické paměti jednu řádku, ve které jsou uložené následující informace
  + **číslo stránky**, která je nahrána do tohoto rámce
  + **číslo procesu**, do jehož VAS tato stránka patří
  + **kontrolní bity**
    - present bit
    - reference bit
    - modify bit
    - přistupová práva
    - cache disable/enabled
  + **index zřetězení** (chain)
* OS si musí udržovat pouze jednu tabulku pro celý systém
* Číslo stránky se pomocí hašovací funkce přepočte na index do tabulky. Protože počet stránek je větší, než počet rámců fyzické paměti, několik různých stránek se může namapovat na stejnou řádku v tabulce → proto se používá technika zřetězení
* udržuje jednu tabulku pro celý systém
* tabulka obsahuje pro každý rámec fyzické paměti jednu řádku
* zabírá méně místa než tabulka stránek, ale hledání v této tabulce je pomalé → pro urychlení překladu se používá společně s TLB



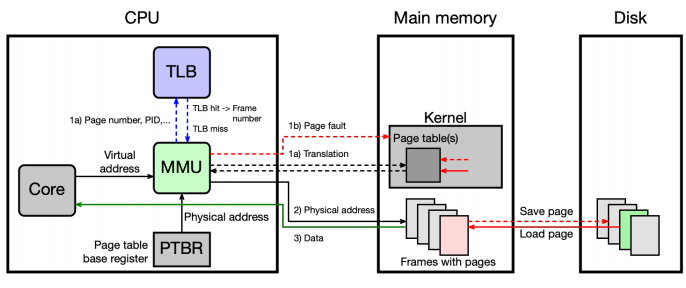
**Příklad**

Obsah obrázku text

Popis byl vytvořen automaticky

#### Translation lookaside buffer (TLB)

* za účelem urychlení překladu , MMU zde hledá překlady
* v praxi implementován jako n-way cache
* **obsahuje informace o posledně překládaných virtuálních adresách** (číslo stránky-číslo rámce)
* počet položek TLB je výrazně menší než počet virtuálních stránek/počet fyzických rámců
* položka TLB obsahuje
  + valid bit: zde je platná položka
  + číslo stránky
  + číslo rámce
  + ASID: identifikátor adresního prostoru
  + Control bits, …
* Pokud aplikace používá/alokuje velkou část ze svého VAS, pak může docházet k velké frekvenci TLB miss (časté přepisování položek v TLB)



**Segmentace**

* jednorozměrný VAS má problémy při implementaci pomocí fyzických oblastí (problém s fragmentací, sdílení strukur mezi procesy...)
* segmentace rozděluje VAS do několika segmentů s různými vlastnostmi
* v rámci segmentů se používá stránkování
* pro překlad adres se používá tabulka segmentů (obsahuje kontrolní bity, velikost segmentu a počáteční adresu segmentu)
* číslo segmentu funguje jako index do tabulky segmentů
* pro každý proces si OS udržuje 1 tabulku segmentů
* segmentace se stránkováním: virtuální prostor má segmenty, které jsou složeny ze stránek. Nejdřív se hledá segment, v segmentu stránka, v té je číslo rámce

### Algoritmy pro nahrazování stránek

* když jsou všechny rámce fyzické (hlavní) paměti obsazené, musí OS najít vhodný rámec, který se uvolní → k tomu slouží algoritmy pro náhradu stránek
* *požadavky* *na algoritmy*:
  + **minimalizace počtu výpadků stránek**
  + rychlost
  + jednoduchá implementace
* *Princip algoritmů*
  + K instrukcím/datům ve VAS procesu se nepřistupuje náhodně
    - Instrukce se většinou vykonávají sekvenčně, pouze občas pokračujeme instrukcí, na vzdálené adrese, která je uložena na jiné stránce (větvení výpočtu, cyklus, skok, …)
    - Data jsou uložena blízko sebe (halda, zásobník, pole, zřetězený seznam, …) na jedné nebo několika stránkách přistupujeme k nim často sekvenčně (procházení pole/seznamu, push/pop na zásobník, …)
    - **Platí princip prostorové a časové lokality**

#### Optimální algoritmus

* nahradí se stránka, která má čas příštího přístupu nejdelší (bude se k ní přistupovat za nejdelší dobu)
* generuje minimální počet výpadků stránek
* nereálná (neznáme budoucnost), ale dá se použít pro porovnání kvality

Obsah obrázku stůl

Popis byl vytvořen automaticky

#### NRU algoritmus (Not Recently Used)

Obsah obrázku text

Popis byl vytvořen automaticky

* jednoduchý na pochopení
* rozumně složitá implementace
* relativně malý počet výpadků stránek

Obsah obrázku stůl

Popis byl vytvořen automaticky

#### FIFO algoritmus (First in, First Out)

* OS si udržuje seznam všech stránek, které jsou aktuálně v RAM
* V okamžiku, kdy se stránka nahraje do hlavní paměti, přidá se její záznam na konec seznamu
* první ze seznamu je vhodný kandidát na náhradu
* jednoduchý na pochopení a implementaci
* nahrazuje se stránka, která je v paměti nejdéle
* Algoritmus nezohledňuje, kdy se ke stránce přistupovalo, ale pouze kdy se stránka nahrála do hlavní paměti
  + **Způsobuje relativně velký počet výpadků stránek**

**Obsah obrázku stůl

Popis byl vytvořen automaticky**

#### Clock algoritmus

* modifikace FIFO
* seznam stránek jako kruhová fronta
* na počátku ukazuje ručička na první položku fronty
* pro každou stránku si pamatujeme R bit
  + když se stránka nahraje do paměti, OS nastaví R bit na hodnotu **1**
  + při každém přístupu (čtení/zápis) ke stránce se nastaví R bit na hodnotu **1**
* Postup při hledání vhodné stránky pro náhradu
  + Pokud ručička ukazuje na stránku, jejíž R bit má hodnotu 1, potom se resetuje R bit na 0 a ručička se posune na následující stránku (položku) ve frontě
  + Předchozí krok se bude opakovat, dokud ručička nebude ukazovat na stránku s R bitem rovným hodnotě 0
    - Tato stránka se nahradí a ručička se nastaví na následující stránku ve frontě
* Vlastnosti:
  + Rozumně složitá implementace
  + Algoritmus generuje nízký počet výpadků stránek

*Obsah obrázku stůl

Popis byl vytvořen automaticky*

* *two-handed clock:* dvě ručičky, ručičky se pohybují stejně, první ručička nastavuje R bit na 0, druhá vyřazuje, pokud je R stále 0, rozevření ručiček definuje časové okno

#### LRU algoritmus (Least Recent Used)

* vhodný kandidát je stránka, ke které se nepřistupovalo nejdelší dobu
* Vlastnosti:
  + Dobrá aproximace optimálního algoritmu
  + **problematická implementace** pomocí hw čítače, kdy každá položka v tabulce stránek bude navíc obsahovat položku “time-of-used”, ta se inkrementuje při každém přístupu do paměti a vyřazují se ty s nejmenší hodnotou

Obsah obrázku stůl

Popis byl vytvořen automaticky

#### Aging algoritmus

* softwarová simulace LRU

Princip

* pro každou stránku si sytém pamatuje
  + R bit, který se nastaví při přístupu (čtení/zápis) ke stránce
  + N-bitový čítač C, který má všechny bity nastavené na 1 při načtení stránky do paměti
* Systém periodicky pro každou stránku
  + Posune obsah čítače C doprava o jeden bit
  + Nastaví hodnotu nejvýznamnějšího bitu čítače C na hodnotu R bitu
  + Resetuje hodnotu R bitu na hodnotu 0
* Vhodným kandidátem pro náhradu bude stránka jejíž čítač C má nejmenší hodnotu

Vlastnosti

* Implementace tohoto algoritmu má menší režii než LRU
* Algoritmus není tak přesný jako LRU
  + Pro každou stránku si nepamatuje přesný čas přístupu, ale pouze „interval“, ve kterém se ke stránce přistupovalo
  + O každé stránce si pamatujeme pouze omezenou historii díky omezenému počtu bitů čítače C

