

# OSY

## 12. Operační systémy a jejich architektury. Systémová volání, vlákna, procesy. Správa virtuální a fyzické paměti, souborové systémy. Bezpečnost, virtualizace.

<https://cw.fel.cvut.cz/old/courses/b4b35osy/start>

### Operační systémy a přerušení

#### Úkoly OS:

- Spouštět a dohlížet uživatelské programy
- Efektivní využití HW
- Usnadnit řešení uživatelských problémů
- Učinit počítač (snáze) použitelný

#### Jádro OS běží když:

- nastane přerušení nebo výjimka
- uživatelský program zavolá službu OS (systémové volání)
- na začátku spuštění počítače, připraví vše pro běh procesů a spustí první proces. Pak už jen čeká na přerušení, výjimky a systémová volání.

V operačním systému jsou **přerušení** a **výjimky** mechanismy, které umožňují přerušení běžícího programu a přechod do režimu jádra operačního systému (tzv. privilegovaný režim). Tyto mechanismy jsou využívány pro zpracování událostí, chyb a asynchronních požadavků.

**Výjimka** (exception) je chybová situace, která se vyskytne během provádění programu. Může být způsobena například nedostupnou pamětí, dělením nulou nebo pokusem o neplatnou operaci. Když operační systém zachytí výjimku, provede obslužnou rutinu výjimky, která se stará o zpracování a řešení chyby. Obslužná rutina výjimky je spuštěna v privilegovaném režimu a může provést různé akce, jako například ukončení programu, obnovení správného stavu nebo zobrazení chybového hlášení.

Oba mechanismy, přerušení a výjimky, slouží k asynchronnímu vyvolání pozornosti operačního systému a umožňují mu správně reagovat na události a chyby v systému.

**Multitasking** - Zdánlivé spuštění více procesů současně je nejčastěji implementováno metodou sdílení času tzv. Time-Sharing Systems (TSS) - rozšiřuje plánovací pravidla o rychlé (spravedlivé, cyklické) přepínání mezi procesy řešícími zakázky interaktivních uživatelů

Architektura - viz APO (Assembly, cyklus CPU, výjimky a přerušení)

**Ochrana jádra OS** - Uživatel má do jádra OS přístup pouze přes obsluhu přerušení, nebo podobný mechanismus

#### Zdroje přerušení:

- **Vnitřní přerušení**
  - problém při zpracování strojové instrukce
  - instrukce nebo data nejsou v paměti - chyba stránky, chyba segmentu instrukci nelze provést - dělení nulou, ochrana paměti, nelegální instrukce
  - nutno reagovat okamžitě, nelze dokončit instrukci, někdy nelze ani načíst instrukci
- **Vnější přerušení**
  - vstupní/výstupní zařízení asynchronní s během procesoru
  - signalizace potřeby reagovat na vstup/výstup
  - reakce po dokončení vykonávané instrukce
- **Programové přerušení** – strojová instrukce provede přerušení
  - využívá se k ochraně jádra OS
  - obsluha přerušení může používat privilegované instrukce
  - lze spustit pouze kód připravený OS

### Systémová volání, vlákna, procesy

## System Calls

- The main interface between the operating system kernel and user space is the set of system calls
- On Linux, about 400 system calls that provide the main kernel services
- File and device operations, networking operations, inter-process communication, process management, memory mapping, timers, threads, synchronization primitives, etc.
- This interface is stable over time: only new system calls can be added by the kernel developers
- This system call interface is wrapped by the C library, and user space applications usually never make a system call directly but rather use the corresponding C library function

### Služby jádra jsou číselovány

- Registr eax obsahuje číslo požadované služby
- Ostatní registry obsahují parametry, nebo odkazy na parametry
- Problém je přenos dat mezi pamětí jádra a uživatelským prostorem
  - malá data lze přenést v registrech – návratová hodnota funkce
  - velká data – uživatel musí připravit prostor, jádro z/do něj nakopíruje data, předává se pouze adresa (ukazatel)
  - dle ChatGPT: registrový přenos, zásobníkový přenos, předávání ukazatelů, pomocí sdílené(shared) memory

### Služby jádra jsou očíslovány - Application Binary Interface – ABI

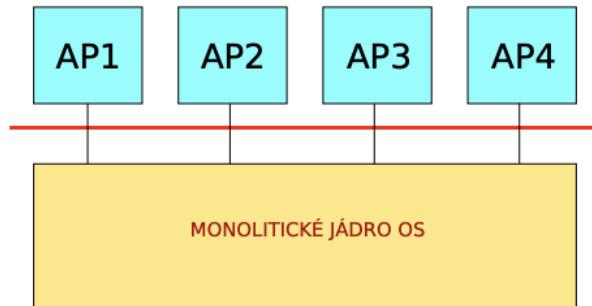
- definuje rozhraní na úrovni strojového kódu

### Přehled služeb jádra(pomocí POSIX API)

- Práce se soubory - open, close, read, write, lseek
- Správa souborů a adresářů - mkdir, rmdir, link, unlink, mount, umount, chdir, chmod, stat
- Správa procesů - fork, waitpid, execve, exit, kill, signal

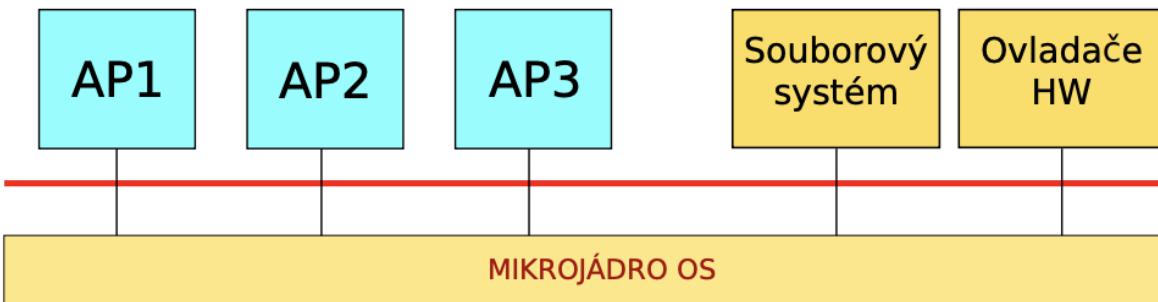
### Monolitické jádro vs mikrojádro

- Klasický monolitický OS
  - Non-process Kernel OS
  - Procesy – jen uživatelské a systémové programy
  - Jádro OS je prováděno jako monolitický (byť velmi složitý) program v privilegovaném režimu
    - „USB MIDI má přístup ke klíči k šifrování disku :-“ CVE-2016-2384
- Služba jádra OS je typicky implementována jako kód v jádře, běžící jako přerušení využívající paměťový prostor volajícího programu



#### Procesově orientované jádro

- OS je soustavou systémových procesů
- Funkcí jádra je tyto procesy separovat, ale umožnit přitom jejich kooperaci
  - Minimum funkcí je potřeba dělat v privilegovaném režimu
  - Jádro pouze ústředna pro přepojování zpráv
  - Řešení snadno implementovatelné i na multiprocesorech
- Malé jádro ⇒ mikrojádro ( $\mu$ -jádro) – (microkernel)



#### Procesy

##### Program:

- je soubor (např. na disku) přesně definovaného formátu obsahující instrukce, data, údaje potřebné k zavedení do paměti a inicializaci procesu

##### Proces:

- je spuštěný program – objekt jádra operačního systému provádějící výpočet podle programu
- je charakterizován svým paměťovým prostorem a kontextem (prostor v RAM se přiděluje procesům – nikoli programům!)
- může vlastnit (kontext obsahuje položky pro) otevřené soubory, I/O zařízení a komunikační kanály, které vedou k jiným procesům, ...

- obsahuje jedno či více vláken

Proces je identifikovatelný jednoznačným číslem v každém okamžiku své existence - PID (Process IDentifier)

#### Co tvoří proces:

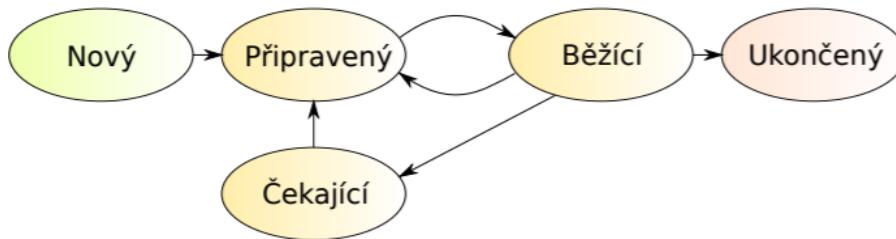
- Obsahy registrů procesoru (čítač instrukcí, ukazatel zásobníku, příznaky FLAGS, uživatelské registry, FPU registry)
- Otevřené soubory
- Použitá paměť: Zásobník – .stack, Data – .data, Program – .text

Rodič vytváří nový proces (potomka) voláním služby **fork** - vznikne identická kopie rodičovského procesu až na:

- návratovou hodnotu systémového volání
- hodnotu PID, PPID – číslo rodičovského procesu

návratová hodnota určuje, kdo je potomek a kdo rodič (0 – jsem potomek, PID – jsem rodič a získávám PID potomka)  
potomek může použít volání služby **exec** pro nahrazení programu ve svém adresním prostoru jiným programem

#### Stavy



Přepnutí od jednoho procesu k jinému nastává výhradně v důsledku nějakého přerušení (či výjimky)

Procesy mohou čekat v různých frontách na vykonání (na přidělení procesoru, na dokončení I/O, na přidělení místa v hlavní paměti, na synchronizační událost, na zvětšení adresního prostoru)

#### Meziprocesní komunikace

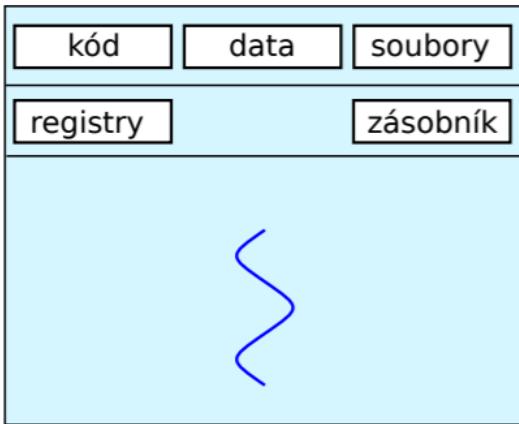
Přehled meziprocesní komunikace

Název	Anglicky	Standard
Signál	Signal	POSIX
Roura	Pipe	POSIX
Pojmenovaná roura	Named pipe	POSIX
Soubor mapovaný do paměti	Memory-mapped file	POSIX
Sdílená paměť	Shared memory	System V IPC
Semafor	Semaphore	System V IPC
Zasílání zpráv	Message passing	System V IPC
Soket	Socket	Networking

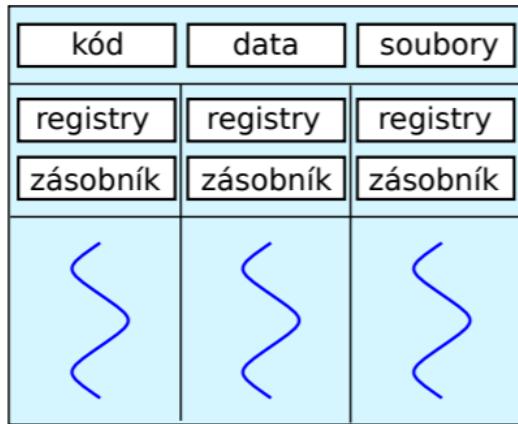
#### Vlákna

##### Vlákno:

- je sekvence instrukcí vykonávaných procesorem
- sdílí s ostatními vlákny procesu paměťový prostor a další atributy procesu (soubory, ... )
- má vlastní hodnoty registrů CPU



jednovlákノvý proces



vícevlákノvý proces

Objekt vytvářený v rámci procesu a viditelný uvnitř procesu

Tradiční proces je proces tvořený jediným vlákem

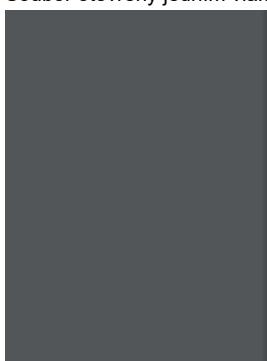
Vlákna podléhají plánování a přiděluje se jim strojový čas i procesory

Vlákno se nachází ve stavech: běží, připravené, čekající, ukončené

Když vlákno neběží, je kontext vlákna uložený v TCB (Thread Control Block) – analogie PCB

Vlákno může přistupovat k globálním proměnným a k ostatním zdrojům svého procesu, data jsou sdílena všemi vlákny stejného procesu:

- Změnu obsahu globálních proměnných procesu vidí všechna ostatní vlákna téhož procesu
- Soubor otevřený jedním vlákem je viditelný pro všechna ostatní vlákna téhož procesu



Co patří komu?

kód programu	proces
lokální proměnné	vlákno
globální proměnné	proces
otevřené soubory	proces
zásobník	vlákno
správa paměti	proces
čítač instrukcí	vlákno
registry CPU	vlákno
plánovací stav	vlákno
uživatelská práva	proces



#### Přednosti:

- Vlákno se vytvoří i ukončí rychleji než proces
- Přepínání mezi vlákny je rychlejší než mezi procesy
- Dosáhne se lepší strukturalizace programu

Všechna vlákna jednoho procesu sdílejí společný adresní prostor

#### Vlákna na uživatelské úrovni:

- OS zná jenom procesy
- Vlákna vytváří uživatelská knihovna, která střídavě mění spuštěná vlákna procesu
- Pokud jedno vlákno na něco čeká, ostatní vlákna nemohou běžet, protože jádro OS označí jako čekající celý proces
- Pouze staré systémy, nebo jednoduché OS, kde nejsou vlákna potřeba

#### Vlákna na úrovni jádra:

- OS Procesy a vlákna jsou plně podporované v jádře
- Moderní operační systémy (Windows, Linux, OSX, Android)
- Vlákno je jednotka plánování činnosti systému

Realizace: - knihovna PThread, Java - třída Thread

#### Plánování procesů

preemptivní - CPU může být procesu násilně odebrán

nepreemptivní - nemůže

## Typy:

- FCFS (First-Come First-Served)
- SPN (SJF) (Shortest Process Next)
- SRT (Shortest Remaining Time) - preemptivní SPN
- cyklické (Round-Robin) - každý proces dostane CPU na malý úsek a pak je vložen na konec fronty
- zpětnovazební (Feedback) - penalizace za dlouhé využití CPU

## Synchronizace

- cílem je zabránit současný přístup více vláken do kritické sekce programu

## Nástroje:

- **semafor** - Semafor je typ zámku založený na čítači. Semafor umožňuje dvě operace:
  - *wait* (acquire, take): Tato operace se volá před vstupem do kritické sekce. Při zavolání operace se vyhodnocuje hodnota čítače: pokud je nenulová, vlácko dekrementuje čítač (typicky o 1) a pokračuje dál ve vykonávání kritické sekce; pokud je nulová, vlácko se zablokuje až do doby, než je čítač opět nenulový.
  - *post* (release, give): Tato operace se volá ihned po vystoupení z kritické sekce. Při zavolání operace se inkrementuje čítač (typicky o 1). Inkrementování čítače o  $n$  způsobí probuzení  $n$  čekajících vláken (viz operace *wait*).

Semafor lze inicializovat na libovolné číslo - toto číslo vyjadřuje, kolik vláken současně může být v kritické sekci. Mutex lze simulovat semaforem s počáteční hodnotou čítače 1.

- **mutex** - typ zámku, který do kritické sekce vpustí jen jedno vlácko současně. Ostatní vlákná se před vstupem do kritické sekce zablokují, *lock()* a *unlock()* metody, lze uzamykat rekursivně
- **monitor** - Monitor je typ zámku, který spojuje koncepty vlastníka a množiny čekajících vláken, kde vlákná čekají až do splnění určité podmínky. Vlákna mezi sebou mohou komunikovat a sdělovat si vzájemně, že byla podmínka splněna a mohou čekání přerušit. Vlastník se také může svého vlastnictví vzdát a připojit se tak k ostatním čekajícím vláknům. Platí, že kritickou sekci může spustit pouze aktuální vlastník monitoru.
- **Spin-lock** - semafor využívající aktivní čekání vlákna(*busy wait*) (v některých případech je to rychlejší než režie s procesy)

## Deadlock

- nastane v případě, kdy dva procesy čekají na uvolnění zámku tím druhým procesem naráz
- **Coffmanovy podmínky** - 4 nutné podmínky pro to, aby mohl deadlock nastat
  - Vzájemné vyloučení (Mutual exclusion) - Prostředek může v jednom okamžiku používat jenom jeden proces (aby nedošlo k porušení konzistence dat).
  - Drž a čekej (Hold & wait) - Proces může žádat o další prostředky, i když už má nějaké přiděleny.
  - Neodnímatelnost (No preemption) - Jakmile proces zmíněný prostředek vlastní, nelze mu ho bezpečně odejmout, musí ho sám vrátit.
  - Cyklické čekání (Circular wait) - Je možné uzavřít cyklus z procesů čekající každý na svého předchůdce – respektive k deadlocku dojde, jakmile je tento cyklus uzavřen.

## Správa virtuální a fyzické paměti, souborové systémy

Podobné jako v APO

### FAP

- fyzický adresní prostor
- skutečná paměť počítače – RAM
- velikost závisí na možnostech základní desky a na osazených paměťových modulech

### LAP

- logický adresní prostor
- někdy také virtuální paměť
- velikost záleží na architektuře CPU

### Výhody systému bez správy paměti:

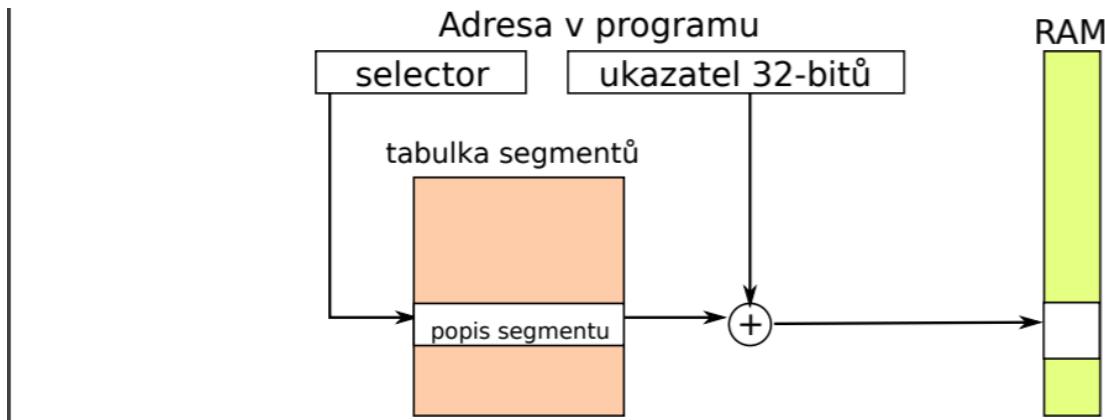
- rychlosť přístupu do paměti
- jednoduchost implementace
- lze používat i bez operačního systému – robustnost

### Nevýhody systému bez správy paměti:

- Nelze kontrolovat přístup do paměti (kdokoli může cokoli v paměti přepsat)
- Omezení paměti vlastnostmi HW

## Segmentace

Program je kolekce segmentů - každý má svůj logický význam (hlavní program, procedura, funkce, objekt a jeho metoda, proměnné, pole, ..)  
Základní úkol – převést adresu typu (segment selector, offset) na adresu FAP



## Výhody segmentace

- Segment má délku uzpůsobenou skutečné potřebě
  - minimum vnitřní fragmentace
  - Lze detekovat přístup mimo segment, který způsobí chybu segmentace – výjimku typu „segmentation fault“
- Lze nastavovat práva k přístupu do segmentu
- Lze pohybovat s daty i programem v fyzické paměti - posun počátku segmentu je pro aplikační proces neviditelný a nedetekovatelný

## Nevýhody segmentace

- Alokace segmentů v paměti je netriviální úloha
  - Segmenty mají různé délky.
  - Při běhu více procesů se segmenty ruší a vznikají nové. Problém s externí fragmentací
- Režie při přístupu do paměti

## Fragmentace

*Externí (vnější) fragmentace* - Celkové množství volné paměti je sice dostatečné, aby uspokojilo požadavek procesu, avšak prostor není souvislý, takže ho nelze přidělit

*Interní (vnitřní) fragmentace* - Přidělená díra v paměti je o málo větší než potřebná, avšak zbytek je tak malý, že ho nelze využít

## Stránkování

FAP se dělí na úseky zvané rámce - Pevná délka, zpravidla v celistvých mocninách 2

LAP se dělí na úseky zvané stránky - Pevná délka, shodná s délkou rámčů

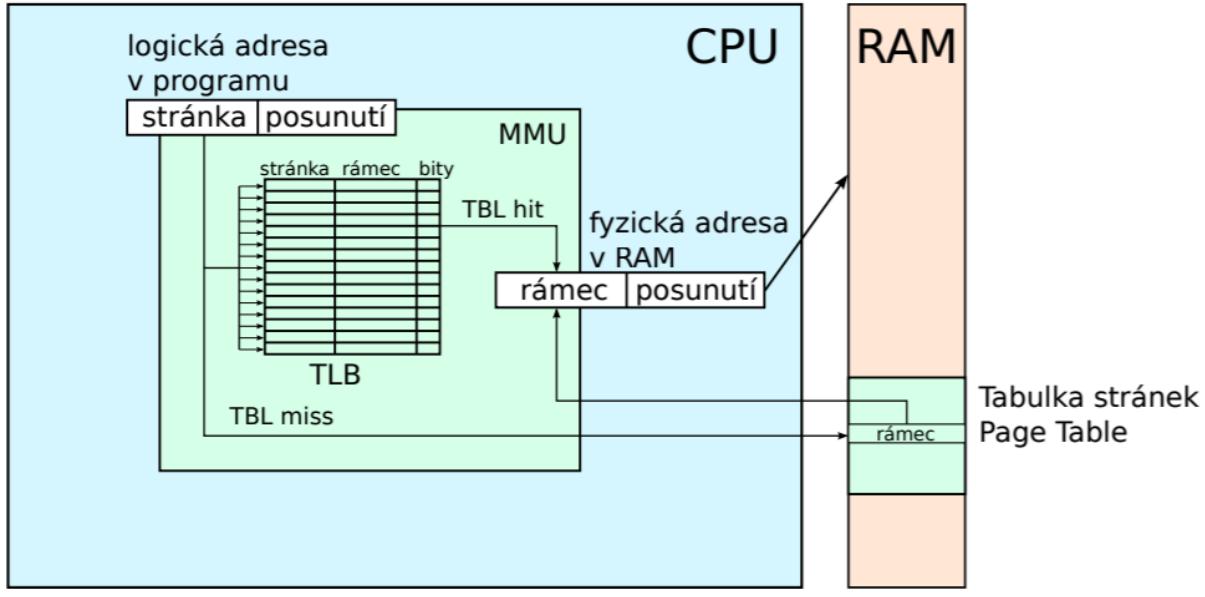
Proces o délce n stránek se umístí do n rámčů, rámce ale nemusí v paměti bezprostředně sousedit

Mechanismus překladu logická adresa → fyzická adresa - pomocí tabulky stránek (PT = Page Table)

Může vznikat vnitřní fragmentace - stránky nemusí být zcela zaplněny

## Nevýhody stránkování

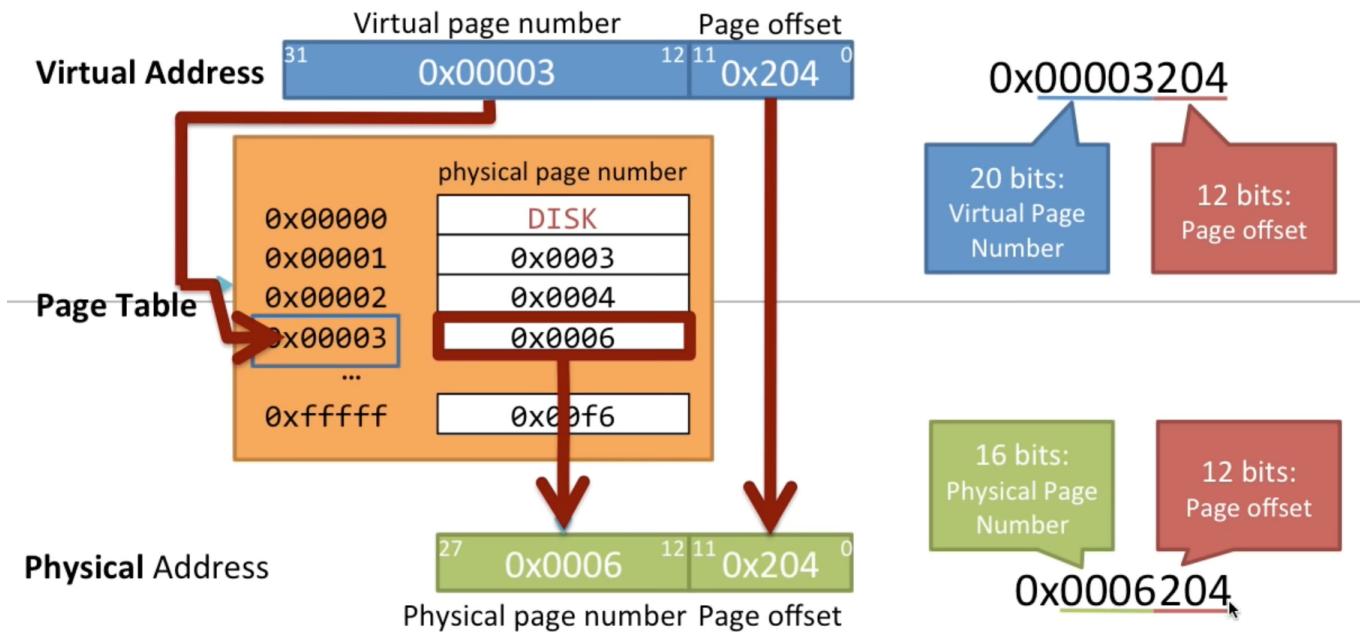
1. Fragmentace paměti: Paměť je rozdělena na menší stránky, což může vést k fragmentaci virtuální i fyzické paměti. Fragmentace paměti může ovlivnit výkon systému a může se stát obtížným najít dostatečně velký blok paměti pro zavedení nové stránky.
2. Zvýšená režie paměti: Při přístupu k paměti jsou potřebné další překlady adres mezi virtuální a fyzickou pamětí. Tyto překlady a správa tabulek stránek zvyšují režii paměti a mohou snížit efektivitu systému.
3. Zpoždění přístupu k paměti: Při přístupu k stránkám, které nejsou aktuálně v paměti, je nutné provést operaci nazývanou "page fault" (chyba stránky), která způsobuje zpoždění. Pokud je stránka načítána z disku, může dojít k výraznému zpomalení přístupu k paměti a ovlivnění celkového výkonu systému.
4. Potřeba správy tabulek stránek: Stránkování vyžaduje správu tabulek stránek, které mapují virtuální adresy na fyzické adresy. Správa těchto tabulek může zahrnovat dodatečnou režii a vyžadovat dodatečný prostor v paměti.
5. Vypršení stránek na disku: Pokud je fyzická paměť vyčerpána a je třeba uvolnit místo pro nové stránky, je nutné vyprázdnit stránky z paměti na disk. Toto vyprazdňování a přesouvání stránek může způsobit zpoždění a snížit výkon systému.



Může být i více úrovní tabulek

#### Překlad virtuální adresy na fyzickou (zdroj: YT David Lack-Schaffer)

- page offset podle velikosti stránky, zůstává stejný i v FAP
- číslo stránky se převede pomocí stránkovací tabulky na fyzické číslo stránky



# What happens if a page is not in RAM?

- Page Table Entry says the page is on disk
- Hardware (CPU) generates a **page fault exception**
- The hardware jumps to the OS page fault handler to clean up
  - The OS chooses a page to evict from **RAM** and write to **disk**
  - If the page is **dirty**, it needs to be written back to disk first
  - The OS then reads the page from disk and puts it in **RAM**
  - The OS then changes the **Page Table** to map the new page
- The OS jumps back to the instruction that caused the page fault.
  - (This time it won't cause a page fault since the page has been loaded.)

"Dirty" means the data has been changed (written). If the page has not been written since it was loaded from disk, then it doesn't have to be written back.

**Q: How long does this take?**

No time  
 A short time  
 A long time  
 An amazingly, incredibly, painfully long time

**A: An amazingly, incredibly,**

**painfully long time**

Disks are *much* slower than RAM, so every time you have a page fault it takes an amazingly, incredibly, painfully long time.

## Virtualizace paměti

Kdy stránku zavádět do FAP? (Fetch policy)

- Stránkování při spuštění
  - Program je celý vložen do paměti při spuštění
  - velmi nákladné a zbytečné, předem nejsou známy nároky na paměť, dříve se nevyužívalo, dnes je využívána
- Stránkování či segmentace na žádost (Demand Paging/Segmentation)
  - Tzv. „líná metoda“, nedělá nic dopředu
  - Řeší problémy s dynamickou alokací proměnných
- Předstránkování (Prepaging)
  - Nahrává stránku, která bude pravděpodobně brzy použita
- Čištění (Pre-cleaning)
  - změněné rámce jsou ukládány na disk v době, kdy systém není vytížen
- Kopírovat při zápisu (copy-on-write)
  - Při tvorbě nového procesu není nutné kopírovat žádné stránky, ani kódové ani datové. U datových stránek se zruší povolení pro zápis.
  - Při modifikaci datové stránky nastane chyba, která vytvoří kopii stránky a umožní modifikace

Stránkování – **politika nahrazování** - Musí se vyhledat vhodná stránka pro nahradu, když je FAP plná:

- LRU (Least recently used) - zahodí se nejdéle nepoužitá stránka
- Druhá šance - zahodí se až stránka, ke které se přistoupilo a byla již použita
- NRU (not recently used) - druhá šance s přidáním dirty bitu (stránka byla modifikována), zahodí se primárně stránka, která byla využita i modifikována

Thrashing – "Výprask" – Jestliže proces nemá v paměti dost stránek, dochází k výpadkům stránek velmi často a počítač nedělá nic jiného než výměny stránek

## Copy on write metoda

The Copy on Write mechanism is a memory management technique used by modern operating systems to optimize memory usage and reduce overhead when creating new processes. Copy on Write works by allowing multiple processes to share the same memory pages until one of the processes modifies the page.

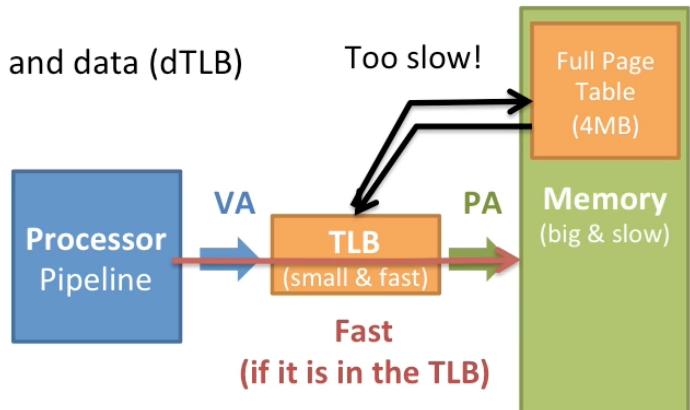
## TLB a víceúrovňové stránkování

- Cache obsahuje skutečná data z paměti
- TLB obsahuje pouze mapování do paměti(urychluje proces překladu adres)

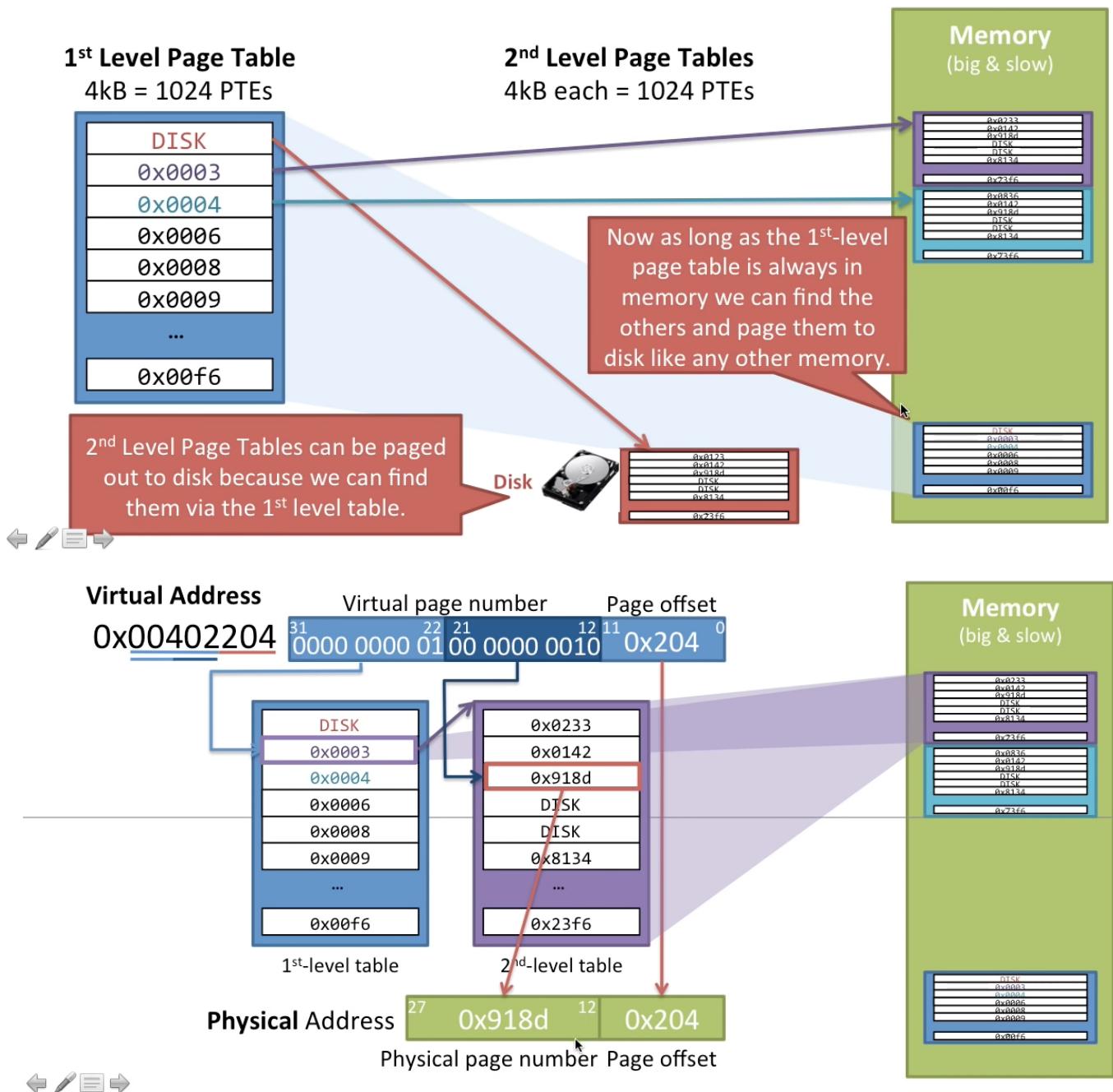
- V TLB je tag - číslo stránky a příslušné číslo rámce v FAP

- To make VM fast we add a special **Page Table cache**:  
the **Translation Lookaside Buffer (TLB)**
  - Fast: less than 1 cycle (have to do it for every memory access)
  - Very similar to a cache
- To be fast, TLBs must be small:
  - Separate TLBs for instructions (iTLB) and data (dTBL)
  - 64 entries, 4-way (4kB pages)
  - 32 entries, 4-way (2MB pages)
 (Page Table is 1M entries)

Lots of locality!  
Miss rates are typically  
only a few percent.

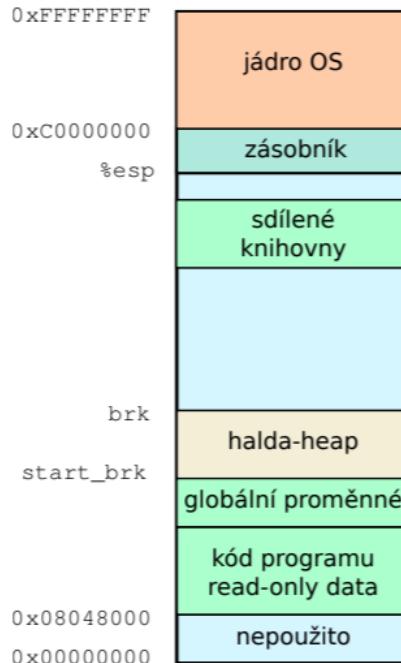


# Multi-level page tables



Virtuální paměťový prostor

- Virtuální paměťový prostor procesu je rozdělen na:
  - systémovou část – dostupnou pro proces systémovými voláními (1GiB pro OS, Windows má dokonce 2GiB)
  - uživatelskou část – prostor pro program, zásobník (pro všechna vlákna) a jeho data
    - část program, statická data
    - část halda – heap, dynamická data až do 3GiB
    - část mma – mapovaná paměť, dynamické knihovny
    - část zásobník, limit 8MiB
  - paměťová mapa procesu je dostupná v /proc/\_pid\_/maps



### Typy alokace paměti:

- Explicitní – program alokuje a uvolňuje paměť pro dynamické proměnné (např. funkce malloc a free v jazyce C, new/delete v C++)
- Implicitní – program alokuje paměť pro nové proměnné, ale již je neuvolňuje (např. garbage collector v Java nebo Pythonu)

Hlavní cíle:

- co nejrychlejší provedení funkcí malloc a free, měla by být rychlejší než lineárně k počtu alokovaných bloků
- minimalizovat fragmentaci paměti, co nejlepší využití paměti souvisí s minimální fragmentací (vnitřní i vnější)

Vedlejší cíle:

- Prostorová lokalita:
  - bloky alokované v podobném čase by měly být blízko u sebe
  - bloky podobné velikosti by měly být blízko u sebe
- Implementace by měla být robustní:
  - operace free by měla proběhnout pouze na správně alokovaném objektu
  - alokace by měla umožnit kontrolovat, zda se jedná o odkaz na alokované místo

Různé způsoby:

- alokace a uvolňování
- udržování informace o volných blocích
- výběru volného bloku

viz [https://cw.fel.cvut.cz.old/\\_media/courses/b4b35osy/lekce07.pdf](https://cw.fel.cvut.cz.old/_media/courses/b4b35osy/lekce07.pdf)

## Souborové systémy

- Trvalé úložiště dat. Rotační nebo flash.
- Posloupnost bloků nebo sektorů
- Oddíly - jeden fyzický disk, vícero logicky → mám jednu plotnu ale disky C,D,E
- Na začátku disku je tabulka definující typ, (jméno), počáteční a koncový sektor oddílu
  - Master Boot Record (MBR) – pozůstatek MS-DOSu, 1. sektor na disku (512 B), obsahuje místo pro 4 oddíly.
  - GUID Partition Table (GPT) – modernější, více informací, „neomezený“ počet oddílů.

### Souborový systém

- Způsob organizace dat na pevném disku
- Data uložená v pojmenovaných souborech
- Soubory v adresářích (složkách)

- Hierarchická struktura adresářů
- Adresář je seznam dvojic («jméno souboru», «umístění»)
- Metadata = data o datech
- Organizace pomocí B stromu

#### Rozložení dat na disku

- Souborový systém definuje velikost bloku (např. 4 KiB)
- Prostor na disku je vždy alokován v násobcích velikosti bloku
- Superblok určuje umístění kořenového adresáře a další informace o souborovém systému Vždy na předem známém místě (např. 1. blok na disku)  
Často uložen ve více kopíech
- Informace o volných blocích OS musí mít přehled, který blok je volný a který obsazený Podobně jako v alokátorech paměti – např. freelist Typicky bitová mapa (1 bit na blok) Kopie v paměti pro urychlení přístupu (cache)
- Bloky ukládající obsah souborů Existuje mnoho způsobů, jak je organizovat

### Základní možnosti uložení obsahu – vlastnosti

- Soubor je vždy uložen v souvislém úseku bloků
  - Podobné alokaci paměti
  - Rychlý přístup k datům (lokalita)
  - Neflexibilní, způsobuje fragmentaci a nutnost přemisťovat soubory
- Spojové seznamy
  - Každý blok obsahuje kromě dat i odkaz na další blok, adresář odkazuje na 1. blok souboru
  - Výhodné pro sekvenční přístup k souborům, nevýhodné pro vše ostatní
  - Nemožnost „mapovat“ data z disku přímo do paměti
  - Jeden špatný sektor na disku (porucha) může způsobit „ztrátu“ zbytku souboru
- Indexové struktury
  - „Indexový blok“ obsahuje ukazatele (čísla bloků) na mnoho jiných bloků
  - Vhodnější pro náhodný přístup, stále poměrně dobré pro sekvenční přístup
  - Může být potřeba použít více indexových bloků

### Souborový systém FAT

File Allocation Table

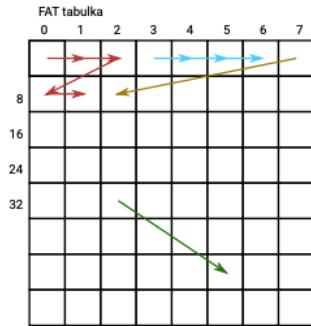
13 / 41

- Starý souborový systém s mnoha nedostatků a omezeními.
- Něco mezi spojovými seznamy a indexovou strukturou.
- Základní jednotka „cluster“ (4–32 KiB)
- FAT12:  $2^{12}$  clusterů, FAT16:  $2^{16}$ , FAT32:  $2^{28}$
- Rozložení disku:



- MBR – master boot record (info o soub. systému, tj. velikost, jméno, počet kopíí FAT tabulek atd.)
- FAT1, 2 – dvě kopie FAT tabulek (redundance)

Adresář	
jméno	0
soubor2	3
soubor3	7
soubor4	34



- Jedna položka FAT tabulky má 12/16/32 bitů a odpovídá jednomu clusteru na disku
- Hodnota položky udává číslo následujícího clusteru (konec šipky) nebo -1 značící konec souboru.
- Číslo 1. clusteru se najde v položce adresáře
- Pro urychlení přístupu je tabulka uchovávána v paměti

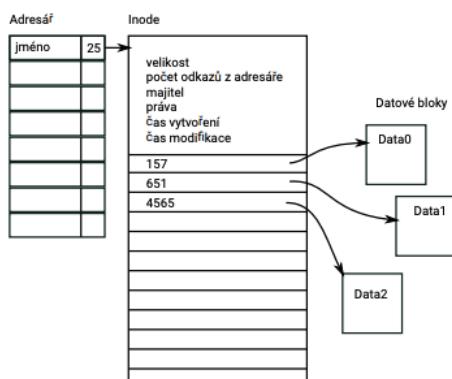
### Nevýhody

- Fragmentace
- Omezená velikost (např.  $2^{32} \times 4\text{KiB}$ )
- Nutnost procházet FAT položky sekvenčně (zpomaluje náhodný přístup u velkých souborů)

## Indexový souborový systém

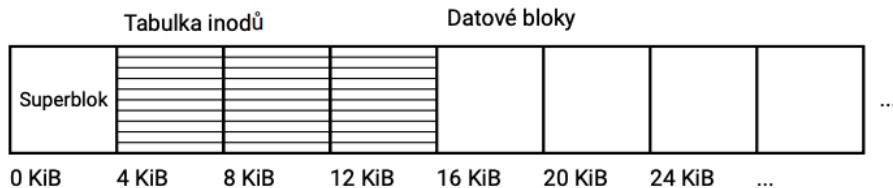
Základ mnoha UNIXových souborových systémů (např. Linuxový ext2 – ext4).

- Metadata o jednotlivých souborech jsou uložena v datové struktuře zvané **inode**.
- Položka adresáře obsahuje kromě jména souboru i číslo (pořadí) inode
- inode obsahuje pevný počet odkazů na datové bloky
  - Z offsetu v souboru lze jednoduše spočítat, který odkaz použít pro přístup k datům (dobré pro náhodný přístup)
- Několik inode se vejde do 1 bloku (velikost inode bývá např. 128 B)



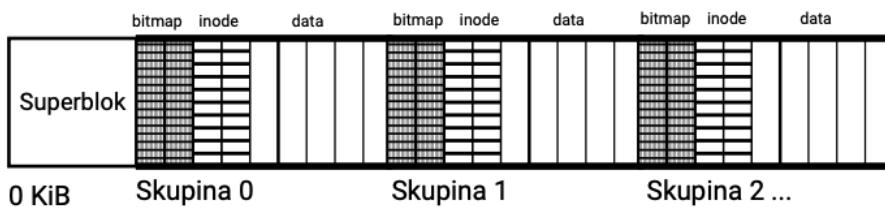
## Rozložení na disku

- Pevný počet inodů
- inode lze nalézt na základě jeho indexu v tabulce
- inode je zkratka *index node*
- Superblok – informace o souborovém systému
  - celková délka, počet inode, ...
  - počet volných bloků a inode
  - odkaz na záložní kopii superbloku
- Kořenový adresář: např. v inode č. 0



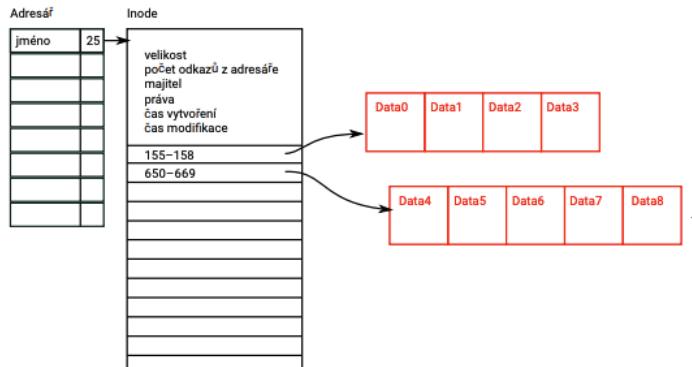
## Skupiny (ext2–4)

- Při práci se souborem je potřeba pracovat s bitmapou, inodem a datovými bloky
- Disky (zejména rotační, ale částečně i SSD) přistupují rychleji k blokům uložených blízko sebe
- Co když datové bloky budou až na konci disku?
  - Hlavičky disků musí pořád jezdit mezi začátkem (bitmapy, tab. inode) a koncem disku (data)
- Řešení: skupiny
  - Souborový systém se snaží alokovat datové bloky ve stejné skupině jako inode souboru



## Extents

- Tabulky bloků nejsou efektivní pro obrovské soubory, velká režie
- Moderní souborové systémy mohou odkazovat místo na jednotlivé bloky na celé souvislé skupiny bloků
- Odkazovaná skupina s více než jedním blokem se nazývá **extent**
- Implementováno v: ext4, NTFS, btrfs, ...



## Žurnálovací systém souborů

- Před tím, než se začne souborový systém modifikovat, se uloží seznam potřebných modifikací na vyhrazené místo – žurnál
- Pokud dojde k pádu systému, zkонтroluje se žurnál, změny disku v něm nalezené se provedou dodatečně
- Žurnálování se někdy nazývá „dopředné logování“
- Implementováno: NTFS, ext3, ...

## Možné scénáře pádu systému

- 1 Do žurnálu se zapíše pouze část transakce
  - Souborový systém (SS) je konzistentní a obsahuje původní data
  - Při startu OS se zjistí, že transakce v žurnálu není kompletní (viz další slide) a ignoruje se.
- 2 Do žurnálu zapíšeme celou transakci, ale neaktualizují se bloky SS
  - Při startu OS aktualizujeme bloky SS podle informací v žurnálu
- 3 Zapíšeme celou transakci, aktualizujeme bloky systému, ale neodstraníme transakci ze žurnálu
  - Při startu OS se bloky přepíší ze žurnálu – žádná změna, už zapsané byly a transakce se odstraní ze žurnálu
- 4 Zapíše se pouze část transakce (např.  $TxB$ ,  $I_{v2}$  a  $TxE$ , bez  $B_{v2}$  a  $D_{v2}$ )
  - **Problém!**
  - HW disku se snaží provádět optimalizace a může změnit pořadí vykonávání příkazů zaslaných OS
  - OS musí disku posílat speciální příkazy (tzv. bariéry), aby se data skutečně zapsala v potřebném pořadí
  - Bariéra garantuje, že příkazy zaslané před bariérou budou vykonány před příkazy zaslanými po bariéře
  - Při zápisu transakce do žurnálu se tedy disk posílá sekvence příkazů:  $TxB$ ,  $I_{v2}$ ,  $B_{v2}$ ,  $D_{v2}$ , **bariéra**,  $TxE$

Sekvenční čtení - vhodnější spojové struktury - FAT, méně vhodné indexové systémy(ext)

Náhodné čtení - nevhodné FAT, vhodnější indexové - ext

## Bezpečnost

### TCB

Všechny systémy obsahují entity, kterým se věří

- pokud selžou, systém nemusí být bezpečný

- hardware, OS, administrátor serveru, ...
- Trusted Computing Base (TCB):
- množina všech takových entit
- Bezpečné systémy musí mít důvěryhodné TCB
- minimalizace TCB je klíčem k důvěryhodnosti

## ACL vs. schopnosti

### Seznamy řízení přístupu (ACL)

- Proces musí být schopen **zjistit jaké objekty existují** (pojmenování) a pak teprve je může používat (a nebo mu je k nim přístup odemřen)
- Typicky to feší tzv. **ambientní autorita** – tj. každý proces má všechna práva uživatele, který ho spustil (např. „vidí“ celý souborový systém a může zjistit, kteří další uživateli jsou v systému).
  - Pokud program spouští jiný program, potomkovi nelze jednoduše práva omezit.
  - V Linuxu se dnes tento problém řeší pomocí „jmenných prostorů“ (**namespaces**), ale není to elegantní a trpí to některými nedostatkami

### Schopnosti

- Neexistuje ambientní autorita, každému procesu jsou **delegovány** jen ty **schopnosti**, které potřebuje (zásada nejmenší pravomoci).
- Např. proces nevidí všechny soubory, ale jen soubory (či celé adresárové stromy), které mu rodič nebo nějaká služba „deleoval(a)“.
- Nikdo nemůže delegovat schopnosti, které sám nemá.

## Přetečení zásobníku

- Programátor zapomene zkontrolovat velikost proměnných na zásobníku
- Uživatel může předat programu více dat, než je velikost proměnné na zásobníku
- To můžezpůsobit přepsání dalších lokálních proměnných, návratové adresy, parametrů, ...
- Program pak většinou „spadne“ (segmentation fault)
- Nebo toho můžeme zneužít a donutit program, aby dělal to, co chceme my.

Předejít útoku lze pomocí náhodného rozložení adresního prostoru

### Běžné mechanismy zabezpečení v OS:

- Systémy pro kontrolu přístupu - kontrola, k čemu může daný proces přistupovat
- Autentizační systémy - potvrzení identity toho, jehož „jménem“ proces běží
- Logování - Kvůli auditům, detekci útoků, vyšetřování a obnovu
- Šifrování souborových systémů - HW lze šifrovat celý disk, SW jen souborový systém (nelze šifrovat partition table)
- Správa pověření (credentials)
- Automatické aktualizace

Safety – ochrana okolí před systémem

Security – ochrana systému před okolím

## Virtualizace

# Co je virtualizace?

- Základní myšlenka: abstrakce hardwaru počítače a jeho částečná emulace v SW
  - Operační systém, který běžel na fyzickém hardwaru běží na virtuálním HW (**virtuální stroj, VM**)
- Hlavní komponenty:
  - **Hostitel** (host) – fyzický hardware na kterém vše běží
  - **Hypervizor a virtual machine monitor (VMM)** – SW implementující virtuální hardware
  - **Host** (guest) – SW (typicky OS) běžící na virtuálním HW
- Jeden hostitel typicky může hostit více hostů najednou
- Používaná od 70. let, zejména na mainframech firmy IBM
  - Popek a Goldberg definovali požadavky pro virtualizaci počítačové architektury v r. 1974
  - Architektura x86 je plně virtualizovatelná od r. 2005

## Výhody virtualizace

- **Izolace** virtuálních strojů (VM) mezi sebou
  - V ideálním případě se útok či virus nerozšíří z jednoho VM do ostatních
- **Nezávislost** softwaru na hardwaru konkrétního počítače
- Běh **více různých OS** na jednom počítači
- Možnost **pozastavení** běhu VM
  - Pozastavenou VM lze zkopírovat či přesunout na jiný počítač a **pokračovat** v běhu jinde.
- Možnost **živé migrace** – běžící VM je přesunut na jiného hostitele bez přerušení přístupu uživatelem
- Vytváření **šablon**
  - šablona = obraz OS s aplikací, distribuce všeho dohromady zákazníkům, možnost spustit víckrát
- Při **vývoji OS** „nepadá“ celý počítač
  - Lze použít pro vývoj částí OS nezávislých na HW
  - Pro vývoj ovladačů většinou nevhodné
  - Reverzní inženýrství...

## Typy virtualizace

- **Virtualizace celého systému** – hostovaný systém neví, že běží na virtualizovaném systému (viz dále)
- **Paravirtualizace** – virtualizovaný systém ví, že neběží na skutečném HW a „dobrovolně“ spolupracuje s hypervizorem (tj. explicitně volá jeho služby)
- **Emulace celého systému** – vše je emulováno v SW, včetně vykonávání instrukcí. Např. Qemu umí vykonávat programy pro ARM na x86.
- **Virtualizace běhového prostředí programu** – Java VM, C# VM (mimo rámec tohoto předmětu)
- **Kontejnerizace aplikací** – viz dále

## Virtualizace CPU

- Klasické CPU vykonává kód ve dvou režimech (módech):
  - **uživatelském** (uživatelské aplikace, x86: Ring 3)
  - **privilegovaném** (jádro OS, x86: Ring 0)
- Při virtualizaci:
  - nemůžeme nechat vykonávat hostované jádro v privilegovaném režimu – nebyla by zajištěna izolace VM mezi sebou
  - potřebujeme implementovat **virtuální uživatelský** a **virtuální privilegovaný** režim
  - **skutečný privilegovaný režim** použijeme pro hypervizor
  - uživatelský i privilegovaný virtuální režim běží ve **skutečném uživatelském režimu** procesoru (sdílí ho)
- **Důsledek:** Virtuální stroj je z pohledu OS/hypervizoru velmi podobný běžnému procesu.
- **Jak zajistit, že při systémových voláních z virtuálního uživatelského režimu přejdeme do virtuálního privilegovaného režimu a ne do skutečného privilegovaného režimu?**

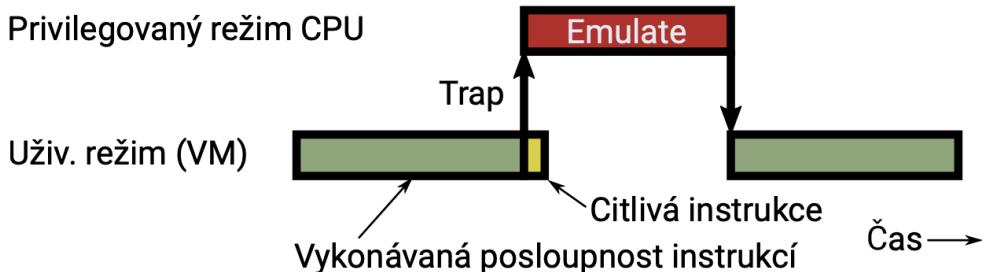
# Trap-and-emulate

Základní princip virtualizace

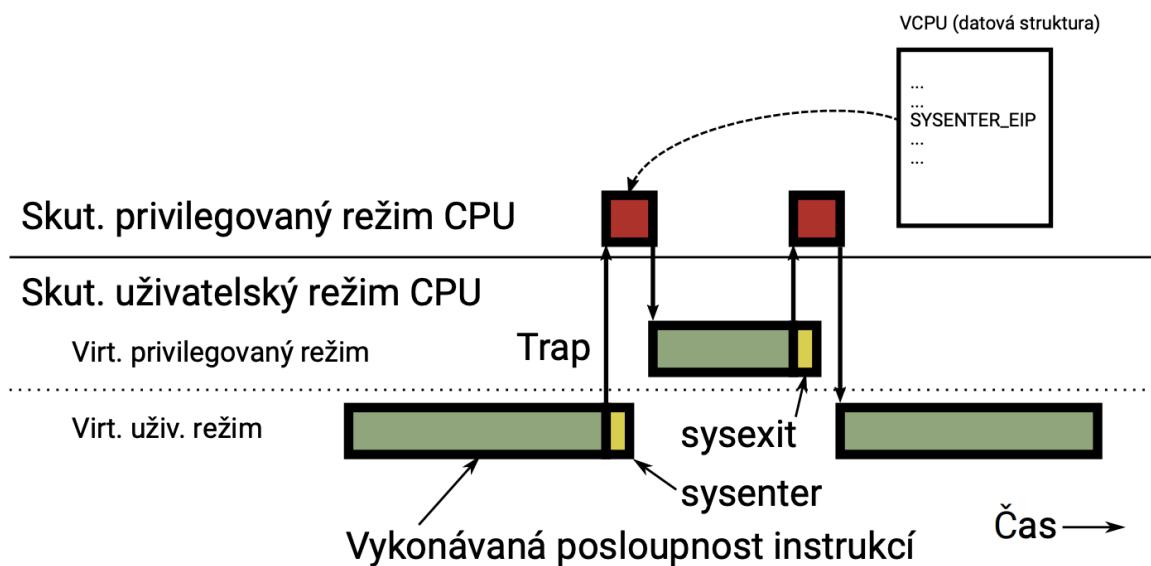
Popek a Goldberg: Požadavky na virtualizovatelnost architektury CPU

„Všechny citlivé instrukce musí být zároveň privilegované instrukce.“

- **Citlivé instrukce:** Mění *globální stav* hostitele nebo se chovají rozdílně v závislosti na *globálním stavu*.
  - Příklad: Instrukce CLI (zákaz přerušení) mění globální stav CPU.
  - Chování instrukce SYSENTER závisí na globálním stavu – přepne procesor do privilegového módu a skočí na **vstupní bod jádra OS** (ale každý hostovaný OS má jiný vstupní bod).
- **Privilegované instrukce:** Pokus o jejich vykonání v uživatelském módu způsobí výjimku (**trap**), která je předána do privilegového módu k obsloužení



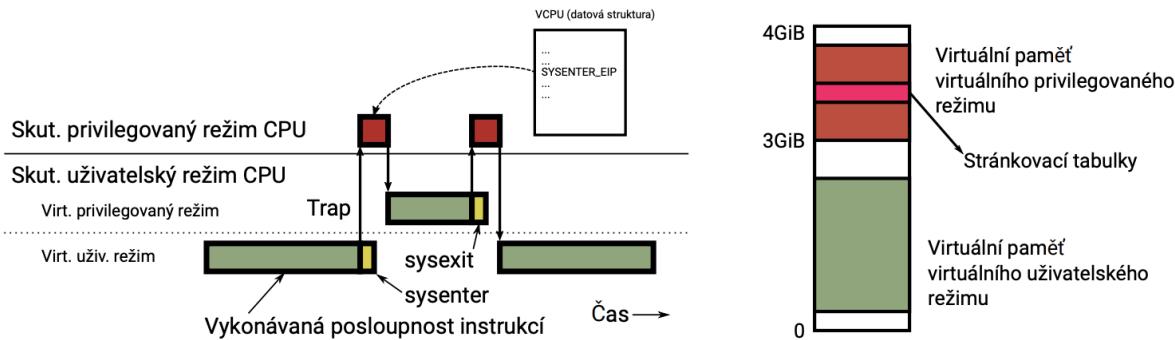
## Virtualizace systémového volání



- Uživatelský kód běží stejně rychle jako bez virtualizace
- Přechody do jádra a zpět jsou pomalejší kvůli emulaci
  - Hypervizor/VMM si přečte virtualizovaný registr SYSENTER\_EIP (položka v datové



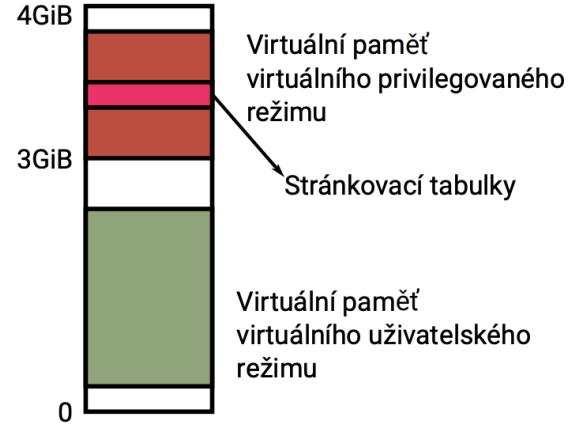
# Virtualizace jednotky správy paměti (MMU)



- Základ bezpečnosti OS je, že kód běžící v uživatelském režimu nemůže modifikovat paměť jádra
- Virtuální uživatelský a privilegový režim ale běží ve skutečném uživatelském režimu a tudíž mají oba stejná oprávnění.
- VMM musí emulovat jednotku správy paměti
  - Při běhu virtuálního uživatelského režimu VMM nastaví CPU, aby používalo stránkovací tabulku, kde je povolen přístup pouze k virtuální uživatelské paměti.
  - Při běhu virtuálního privilegového režimu musí být přístup i do paměti hostovaného jádra – jiná stránkovací tabulka.
  - Jádro hostovaného OS nemůže mít přístup ke skutečné stránkovací tabulce.
  - Jak se dá řešit virtualizace přístupu ke stránkovacím tabulkám (např. `Ptab::insert_mapping()` v OS NOVA)?

## Virtualizace stránkovacích tabulek

- Trap-and-emulate & stínové stránkovací tabulky
- Hypervizor zpřístupní hostovanému jádru paměť, kde jsou uloženy virtuální (tzv. stínové) stránkovací tabulky, pouze pro čtení
- Při pokusu o zápis do stránkovacích tabulek (`Ptab::insert_mapping()`) dojde k výjimce (trap)
- VMM se podívá, jak chtěl hostovaný OS stránku nastavit a zkонтroluje, jestli host nepodvádí, nedělá chybu (izolace) atd. Pokud je vše v pořádku, upraví skutečnou stránkovací tabulku (emulate)



# Hardwarově asistovaná virtualizace

- Implementace
  - Trap-and-emulate je v mnoha případech pomalá
- Moderní CPU implementují to, co tradičně dělal hypervizor, přímo v HW
- Intel: VT-x
  - Zavádí nový mód procesoru: **non-root execution**
  - V tomto módu jsou všechny citlivé instrukce zároveň privilegované (v původním (tzv. root) módu to neplatí; viz např. instrukce `pushf`, ...)
  - Existují tedy módy: Root ring 0–3, non-root ring 0–3
  - Hypervizor/VMM může nakonfigurovat, které instrukce způsobí výjimky a přechod z non-root do root módu (tzv. VM Exit).
  - Instrukce `sysenter` (přechod z ring 3 do ring 0) je možné vykonat v non-root módu bez VM Exitu
  - Trap-and-emulate se používá pro emulaci I/O

