

# ZOS

## přehled znalostí

L. Pešička

Verze: 29. května 2018

Průběžně upravováno

# Obsah

- Úvod
- Moduly OS
- Základní pojmy
  - Systémové volání
  - Přerušení
- Jednotlivé moduly podrobně

# Základní pravidlo

Důležité je znát nejen **fakta**, ale **porozumět** jim  
a zasadit je do kontextu celého OS.

Příklad:

algoritmus Second Chance využívá bitu Referenced tak, že ... (**fakta**)

a kdy Second Chance použijeme? (**porozumění**)

když je RAM plná a potřebujeme uvolnit rámec pro stránku, kterou  
potřebujeme mít v paměti

# Základní pojmy

- OS je „aktivní“ ve dvou případech:  
Něco po něm chceme (systémové volání), nebo při přerušení (časovač, klávesnice, výjimka).  
Dále dbá na ochranu (paměti, neoprávněný přístup k souborům) a spravedlivé rozdělování času CPU (plánovač vláken).
- Přerušení
- Systémové volání
- Ochrana paměti
- Souběh
- Kritická sekce

# Souběh (race condition)

- Chybné současné zpracování sdílených dat
  - Pokud by byla data zpracována postupně, k chybě by nedošlo
  - Kritická oblast – označení dat, která jsou souběhem ohrožena
  - Kritická sekce – část programu, která pracuje s daty v kritické oblasti
- 
- Analogie:
  - Křižovatka, kam z různých směrů současně vjedou auta
  - K problému může dojít, ale nemusí – ale situace je riziková

# Přesah (!!)

Vřele doporučuji najít si na courseware  
a prolistovat přednášky:

KIV/OS (Ing. Tomáš Koutný, Ph.D.)

# Operační systém – z čeho se skládá?

- Modul pro správu procesů (a vláken)
- Modul pro správu paměti
- Modul pro správu I/O
- Správa souborů
- Bezpečnost

Každou z těchto částí je potřeba znát.  
Stejně tak je potřeba vědět, jak navzájem spolupracují.

Chci vytvořit proces -> potřebuji nějaký paměťový prostor.  
Chci zapsat do souboru -> potřebuji nějaký ovladač disku nebo síťové karty apod.

# Moduly OS - podrobněji

- **správa procesu** (proces, vlákno, přerušení, přeplánování procesu, plánování procesů)
- **správa paměti** (stránkovaná, segmentovaná, segmentace se stránkováním, swapovací soubor, PFF, trashing, sdílená paměť - glibc 1x v paměti)
- **správa I/O** (RAM prostor, i/o prostor IN a OUT, pozornost zařízení přerušením, ovladač - v režimu jádra)
- **soubory** (VFS, volné bloky x obsazené, jak realizovat adresář, FAT, i-uzly, NTFS, žurnálování)
- **bezpečnost** (napříč: uživ/privileg režim, unixová práva, ACL, nesáhnu do cizí paměti, ...)



# Základní pojmy

- Systémové volání
- Přerušení
- Uživatelský a privilegovaný režim

# Systémové volání

Definice:

- **Mechanismus používaný aplikacemi k volání služeb operačního systému.**

Důvod:

- V **uživatelském režimu** není možné celou řadu věcí vykonat – **není přímý přístup k HW**, nelze tedy přímo přečíst blok z disku, tedy otevřít soubor, číst z něj a zapisovat do něj.
- Pokud aplikace takovou činnost požaduje, nezbývá jí, než **požádat o** danou **službu operační systém**.
- Operační systém **zkontroluje**, zda má aplikace (puštěná nějakým **uživatel**em) pro danou činnost oprávnění a pokud ano, požadovanou činnost vykoná. (Kontrola může být např. podle ACL, zda má proces daného uživatele právo zapisovat do souboru).

# Realizace systémového volání

Dvě možnosti:

- Softwarové přerušení – instrukce **INT číslo**, např. INT 0x80
- Speciální instrukce CPU – instrukce **sysenter**, případně podobná

# Poznámka

- Systémové volání a přerušení jsou dva odlišné pojmy.
- Systémové volání lze realizovat pomocí sw přerušení (int 0x80), nebo s využitím speciální instrukce (sysenter aj.).
- Přerušení v OS plní i jiné důležité funkce.

# Realizace systémové volání SW přerušením

1. MOV EAX, číslo služby
2. MOV EBX, další parametry...
3. INT 0x80

Příklad v assembleru

1. Do registru CPU s názvem EAX dáme číslo požadované služby
2. Do dalších registrů další potřebné parametry (liší se službu od služby)
3. INT 0x80 – instrukce pro SW přerušení  
(V rámci SW přerušení se CPU přepne do privilegovaného režimu a během obsluhy přerušení zakáže další přerušení)

# Jak zjistím, jaké služby jsou k dispozici?

Např. na <http://syscalls.kernelgrok.com/> pro Linux

- Je zde vidět číslo volání (dáme do registru EAX)
- Požadované parametry co uvést v dalších registrech

## Linux Syscall Reference

Show <input type="text" value="All"/> entries		Registers						Search: <input type="text"/>
#	Name	eax	ebx	ecx	edx	esi	edi	Definition
0	<b>sys_restart_syscall</b>	0x00	-	-	-	-	-	<b>kernel/signal.c:2058</b>
1	<b>sys_exit</b>	0x01	int error_code	-	-	-	-	<b>kernel/exit.c:1046</b>
2	<b>sys_fork</b>	0x02	<b>struct pt_regs *</b>	-	-	-	-	<b>arch/alpha/kernel/entry.S:716</b>
3	<b>sys_read</b>	0x03	unsigned int fd	char __user *buf	size_t count	-	-	<b>fs/read_write.c:391</b>
4	<b>sys_write</b>	0x04	unsigned int fd	const char __user *buf	size_t count	-	-	<b>fs/read_write.c:408</b>
5	<b>sys_open</b>	0x05	const char __user *filename	int flags	int mode	-	-	<b>fs/open.c:900</b>
6	<b>sys_close</b>	0x06	unsigned int fd	-	-	-	-	<b>fs/open.c:969</b>
7	<b>sys_waitpid</b>	0x07	pid_t pid	int __user *stat_addr	int options	-	-	<b>kernel/exit.c:1771</b>
8	<b>sys_creat</b>	0x08	const char __user *pathname	int mode	-	-	-	<b>fs/open.c:933</b>

# Není to nepohodlné, pamatovat si čísla služeb?

- Ano, je.
- Proto jsou k dispozici funkce, které „obalí“ jednotlivá systémová volání, takže si nemusíme pamatovat jejich čísla.

Např:

`getpid(), open(), fork(), creat(), execve()`

V Linuxu – v nápovědě sekce 2 - systémová volání:

man 2 open, man 2 fork, man 2 execve

# Knihovnní funkce

- Někdy se nevolá systémové volání přímo.
- Aplikace zavolání **knihovnní funkci** a teprve ona zavolá systémové volání
- Výhoda – skryje podrobnosti o systémovém volání na různých platformách, může provést další užitečné činnosti.
- Např. volání `fopen()` z `stdio.h`
- Srovnejte v Linuxu: `man 2 open` x `man 3 fopen`



# Systémové volání – práce s procesy

Následující systémová volání je potřeba znát:

volání	popis
fork()	Vytvoří nový proces
wait()	Čeká na dokončení procesu
waitpid(id)	Čeká na dokončení konkrétního procesu s PID rovno číslu id
_exit()	Ukončení procesu
execve()	Spustí jiný program v rámci aktuálního procesu

# Systemové volání – práce s pamětí

volání	popis
mmap()	Paměťově mapovaný soubor
munmap()	Odstraní mapování paměťově mapovaného souboru
mlock()	Zamkne paměťovou stránku v paměti, aby nemohla být stránkována do swapu (potřebujeme jí ponechat v RAM)
munlock()	Odemkne stránky v daném adresním rozsahu
brk(), sbrk()	Nastavení konce datového segmentu

# Systémové volání – práce se soubory

volání	popis
creat()	Vytvoří soubor
open()	Otevře soubor v požadovaném režimu (čtení, zápis)
close()	Uzavře soubor
read()	Čtení ze souboru
write()	Zápis do souboru
lseek()	Posune ukazovátka v souboru – pro přímý přístup (z libovolné pozice)

Soubory **sekvenční** – můžeme číst a zapisovat **jen postupně** (nemají lseek)

Soubory **s přímým přístupem** – čtení a zápis **na libovolnou pozici** v souboru (**lseek**)

Dnes většina souborů s přímým přístupem, tj. můžeme se v něm posouvat libovolně.

Např. při přehrávání filmu se můžete podívat na začátek, přeskočit na konec, vrátit se na prostředek.

Systémové volání open() vs. knihovní funkce fopen().

# Systemové volání – práce se zprávami

název	popis
msgsnd()	Poslání zprávy
msgrcv()	Příjem zprávy

Důležitý pojem IPC – InterProcess Communication

Možnosti meziprocesové komunikace:

- Zasílání zpráv
- Sdílená paměť
- Roury

# Přerušení

- Přerušení = událost
- Obsluha přerušení = obsluha události
- Příklad události:  
HW zařízení si žádá pozornost (tik časovače, stisk klávesy, přišel rámec síťové kartě,...)  
  
Událostí může být i zpracování výjimky (dělení nulou, neplatná instrukce atp.)

# Přerušení - motivace

- **Událost !!!**
- Typicky asynchronní (přijde neočekávaně) nebo synchronní (instrukce SW přerušení v programu), pak přijde očekávaně
- Analogie z reálného života
  - S někým si povídáte
  - Zazvoní telefon, vyřídíte telefon
  - Vráťte se k předchozímu povídání

# Přerušení ( = událost)

Definice:

Metoda pro (asynchronní) obsluhu událostí, kdy procesor přeruší vykonávání sledu instrukcí, vykoná obsluhu přerušení a pak pokračuje v předchozí činnosti.

Rozdělení:

- HW přerušení (vnější) – obsluha HW zařízení
- SW přerušení – synchronní, instrukcí **INT x** v kódu procesu
- Vnitřní přerušení (výjimky) – procesor oznamuje chyby při vykonávání instrukcí

# Poznámka k přerušení

Liší se místem vzniku:

- V hardwaru (časovač, klávesnice, nějaká periferie)
- Instrukce programu (INT 0x80)
- Uvnitř procesoru (narazí na nějakou výjimku – dělení nulou, stránka není v paměti ale ve swapu, neplatný kód instrukce atp.)

Procesor když narazí na dělení nulou nemůže zůstat bezradný, vždy musí vědět jak dál – vyhodí výjimku, a na příslušné obsluhu výjimky je podprogram, jak pokračovat dále, např. při dělení nulou ukončit daný proces, kdy k dělení došlo



# HW přerušení

Příklady:

- Časovač (timer)
  - Po tiku časovače se vykoná přerušení
  - V rámci obsluhy přerušení např. kontrola, zda už neuběhlo časové kvantum pro plánování procesů
- Stisknutí klávesy na klávesnici
- Pohyb myši
- Disk signalizuje, že má k dispozici požadovaná data

HW zařízení žádá operační systém o pozornost („věnuj se mi“)

# Poznámky

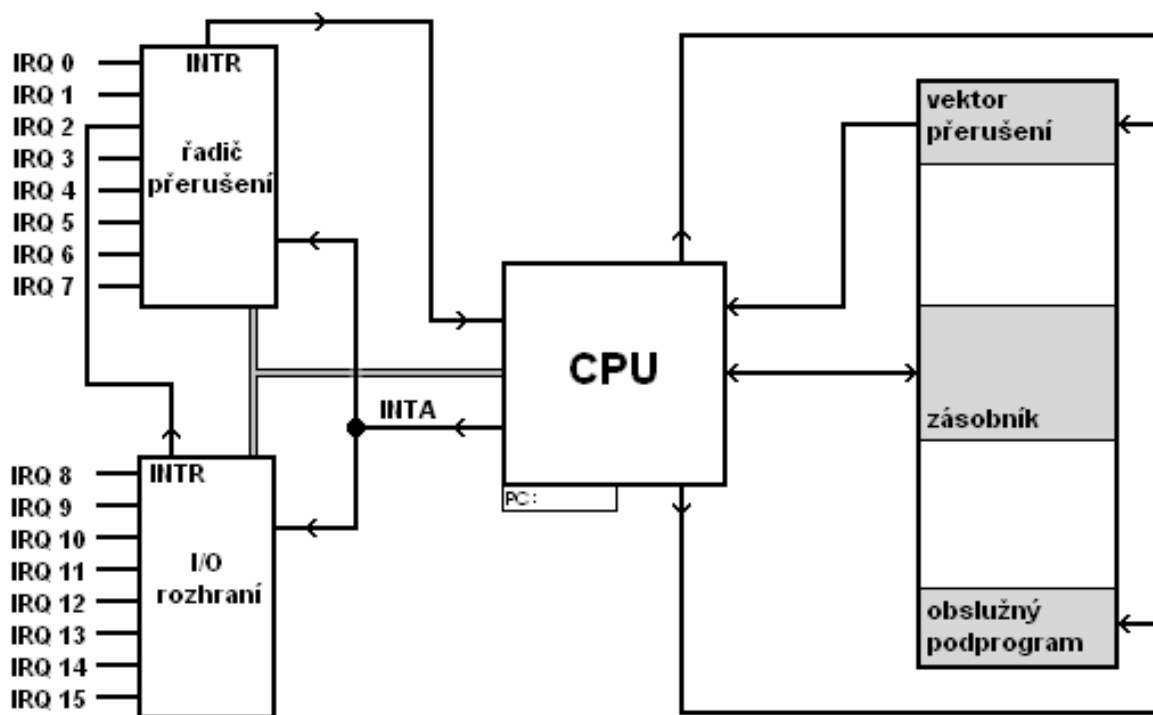
**IRQ** – signál, kterým zařízení (**časovač, klávesnice**) žádá procesor o přerušení zpracovávaného procesu za účelem provedení obsluhy požadavku zařízení

**NMI** – nemaskovatelné přerušení, např. nezotavitelná hw chyba (non-maskable interrupt)

# HW přerušení - zpracování

1. Vnější zařízení vyvolá požadavek o přerušení.
2. I/O rozhraní vyšle signál IRQ na řadič přerušení.
3. Řadič přerušení vygeneruje signál INTR – „někdo“ žádá o přerušení a vyšle ho k procesoru.
4. Procesor se na základě maskování rozhodne obsloužit přerušení a signálem INTA se zeptá, jaké zařízení žádá o přerušení.
5. Řadič přerušení identifikuje zařízení, které žádá o přerušení a odešle číslo typu přerušení k procesoru.
6. Procesor uloží stavové informace o právě zpracovávaném programu do zásobníku.  
(registr FLAGS a CS:EIP, zablokuje další přerušení nastavením IF v registru FLAGS)
7. Podle čísla typu příchozího přerušení nalezne ve vektoru přerušení adresu příslušného obslužného podprogramu.
8. Vyhledá obslužný podprogram obsluhy přerušení v paměti a vykoná ho.
9. Po provedení obslužného programu opět obnoví uložené stavové informace ze zásobníku (registr FLAGS a CS:EIP) a přerušený program pokračuje dál.

# HW přerušení - zpracování



Zdroj: wikipedia

# SW přerušení

- Instrukcí **INT x**, kde x je číslo **0 – 255**
- Vyvolání služby OS v Linuxu: **INT 0x80**
- SW přerušení se používá jako mechanismus pro systémové volání
- Při vyvolání přerušení se procesor přepne do privilegovaného režimu, kdy je možné vykonávat všechny instrukce (např. IN, OUT)
- Je synchronní – tj. nastane, když začneme zpracovávat instrukci INT daného procesu

# SW přerušení - zpracování

1. Do zásobníku se uloží stavové informace o právě zpracovávaném procesu. (registr FLAGS a CS:EIP).
2. Zakáže se další přerušení. (nastavením IF v registru FLAGS)
3. Procesor zjistí vektor přerušení (dle operandu za instrukcí INT).
4. Nalezne obslužný podprogram (dle tabulky vektorů přerušení) a vykoná ho.
5. Po návratu z podprogramu obnoví uložené stavové informace o přerušeném programu.

Pozn.: na zásobník se uloží jen návratová adresa a stavový registr (flags), uložení dalších registrů na zásobník je už potom úkolem obslužné rutiny.

# Obsluha přerušení (!!!)

- I. Mechanismus vyvolání přerušení (vyvolání instrukcí: **INT 0x80**)
  - Na zásobník se uloží registr příznaků **FLAGS**
  - Zakáže se přerušení (vynuluje příznak IF – Interrupt Flag v registru FLAGS)
  - Na zásobník se uloží návratová adresa (**CS:IP**) ukazující na instrukci, kde budeme po návratu z přerušení pokračovat
- II. **Kód obsluhy přerušení – „píše programátor OS“**
  - Na zásobník uložíme hodnoty registrů (abychom je procesu nezměnili)
  - Vlastní kód obsluhy (musí být rychlý, případně naplánujeme další věci)
  - Ze zásobníku vybereme hodnoty registrů (aby přerušený proces nic nepoznal)
- III. Návrat z přerušení (instrukce: **IRET**)
  - Ze zásobníku je vybrána návratová adresa (**CS:IP**) – kde budeme pokračovat
  - Ze zásobníku se obnoví registr **FLAGS** – obnoví původní stav povolení přerušení

# Vnitřní přerušení (výjimky)

- Dělení nulou (5/0 kolik to asi bude?)
- Neplatná instrukce (CPU: hmm.. Tuhle instrukci neznám..)
- Nedostupnost koprocessoru (není matematický koprocessor)
- Výpadek stránky (stránka je ve swapu místo v RAM)
- Nepřístupný segment (adresuji segment, který není v tabulce segmentů)



# Výjimky – poznámka a rozšíření

Výjimka = přerušení generované CPU

- Trap – breakpoint
- Fault – opravitelná chyba (např. výpadek stránky, dělení nulou)
- Abort – neopravitelná chyba  
(CPU nedokáže zavolat obsluhu výjimky, Double Fault)

# Tabulka vektorů přerušení

Motivace:

Procesor ví **číslo přerušení**, ale to mu nestačí. Potřebuje znát, kde leží **adresa obslužného podprogramu**, který má při daném přerušení vykonat.

Tedy mapování:

$f(i) = \text{adresa\_obsluzneho\_podprogramu}$

kde  $i$  je číslo přerušení (0 až 255)

Toto mapování zprostředkuje tabulka vektorů přerušení.

# Tabulka vektorů přerušení

Příklad:

Procesor narazí v procesu na instrukci **INT 0x80**.

Na indexu 0x80 (128 dekadicky) najde adresu obslužného podprogramu, na kterou skočí pro obsluhu daného přerušení.

V tomto konkrétním případě je to u Linuxu vstupní bod do jádra monolitického systému, kde se následně dle hodnoty v registru EAX zavolá příslušné systémové volání.

# Tabulka vektorů přerušení

- Datová struktura
- 256 položek (0-255), po 4B, tedy velikost 1KB
- Od adresy 0 do adresy 1023
- Toto planí v reálném módu CPU

V protected módu CPU:

- IDT (Interrupt Descriptor Table)
- Pole 8bytových deskriptorů (místo 4B v předchozím případě)
- Naplněná IDT tabulka 2KB (256 x 8B)
- **Registr IDTR** udává, kde v paměti leží tabulka vektorů přerušení (IDT)

# Tabulka vektorů přerušení

Definice:

Tabulka vektorů přerušení je datová struktura, ve které se uschovávají vektory přerušení.

Vektor přerušení – obsahuje adresu (první instrukce) podprogramu pro obsluhu daného přerušení.

# Dvě části obsluhy přerušení

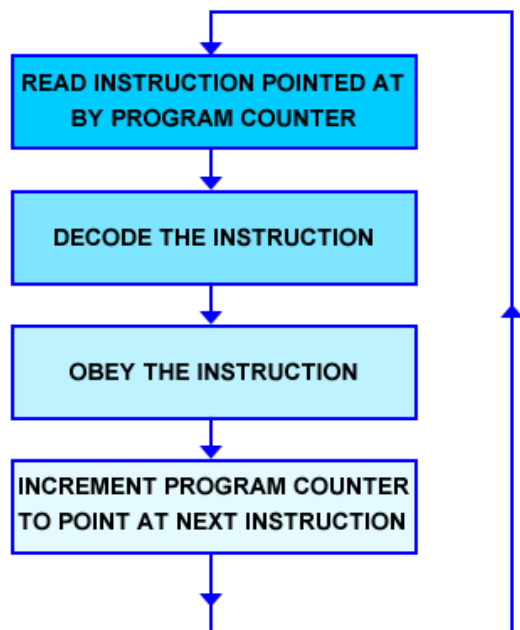
- první část  
ve vlastním režimu obsluhy přerušení  
velmi rychlé (stabilita)
- odložená část  
může napláňovat další část, která se vykoná „až bude čas“

# Sdílení IRQ více zařízeními

- na jedno IRQ lze registrovat několik obslužných rutin (registrovány při inicializaci ovladače)
- do tabulky vektorů přerušení je zavěšena „superobsluha“
- superobsluha pouští postupně jednotlivé zaregistrované obsluhy, až jedna z nich zafunguje
- pokud dané přerušení naráz více zařízeními – zavolá opakovaně

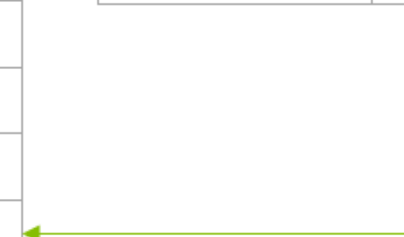
# Čítač instrukcí

- Říká se mu obecně program counter (PC)
- Reálně jde o dvojici registrů CS:EIP
- Ukazuje do paměti na instrukci, která je na řadě pro vykonávání na CPU



Location	Instruction
0	LOAD R0 1
1	ADD R0 1
2	WRITE 0 R0
3	READ R1 0
4	LOAD R2 4
5	COMPARE R0 R2
6	JUMPLT 2
7	EXIT
8	LOAD R1 4

Program Counter	3
-----------------	---





# Scénář příkladu (!!)

1. Uživatel Pepa spustí program (textový editor), který poběží jako proces **p1**.
2. Proces bude chtít otevřít soubor **ahoj.txt**.
3. O otevření souboru musí proces požádat operační systém **systémovým voláním open()**.
4. Soubor **ahoj.txt** bude ve filesystemu chráněný pomocí **ACL (Access Control List)**, kdo k němu smí přistoupit.
5. Jádro operačního systému zkontroluje, zda jej smí Pepa otevřít, a pokud ano, soubor otevře (naplní příslušné datové struktury).

uživatel Pepa (UID= 1015)

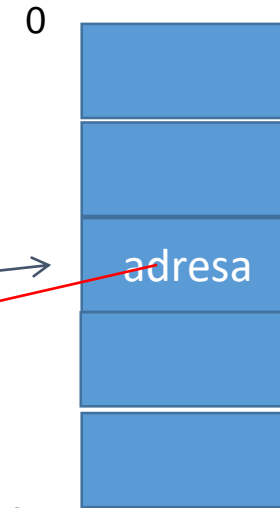
CPU ví, zda je v uživatelském nebo  
privilegovaném režimu

proces p1 (PID = 202)  
volá systémové volání  
open("ahoj.txt")

uživatelský  
režim

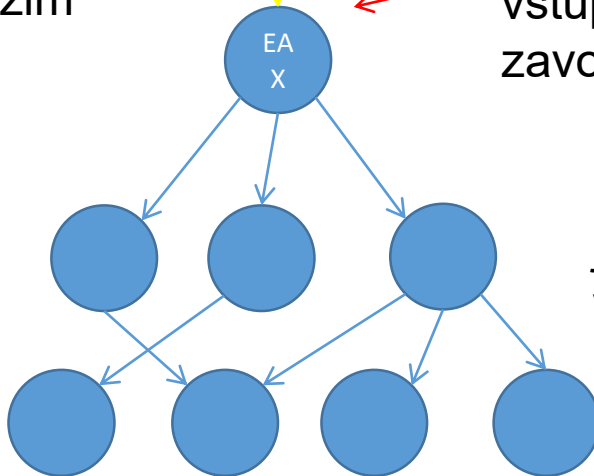
odpovídající instrukce procesu:  
EAX <- číslo služby open  
EBX, ECX, .. <- další param.  
INT 0x80

Tabulka vektorů  
přerušení



privilegovaný  
režim

vstupní bod jádra, dle EAX  
zavolá příslušné volání



jednotlivá systémová  
volání

RAM

256 indexů  
0 .. 255  
každá položka  
obsahuje  
4B adresu  
obslužné rutiny

vektor přerušení zabírá 1KB paměti od adresy 0 v RAM  
(tzv. reálný režim CPU) nebo dle registru IDTR

# Jak jádro rozhodne, že má uživatel k souboru přístup?

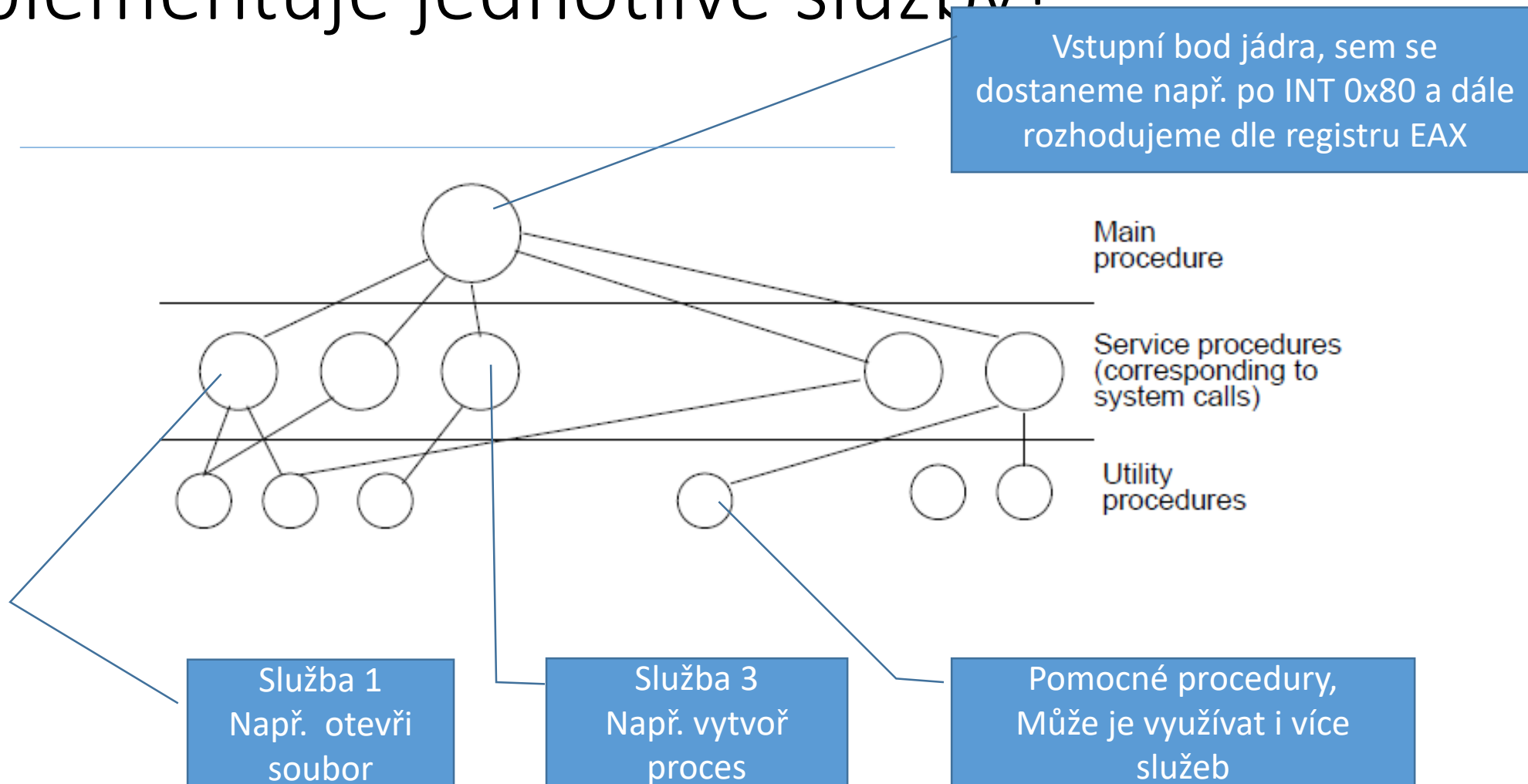
Implementace volání `open()` zjistí:

- na kterém filesystemu (fs) ahoj.txt leží  
`ntfs, fat32, ext3, ext4, xfs, ...`
- zda daný fs podporuje ACL (komplexní práva) nebo základní unixová práva (vlastník, skupina, ostatní) nebo žádná kontrola práv (FAT)  
`ACL slouží ke kontrole přístupových práv`
- zkontroluje, zda ACL vyhovují pro daného uživatele a daný mód otevření souboru (uid, čtení/zápis)

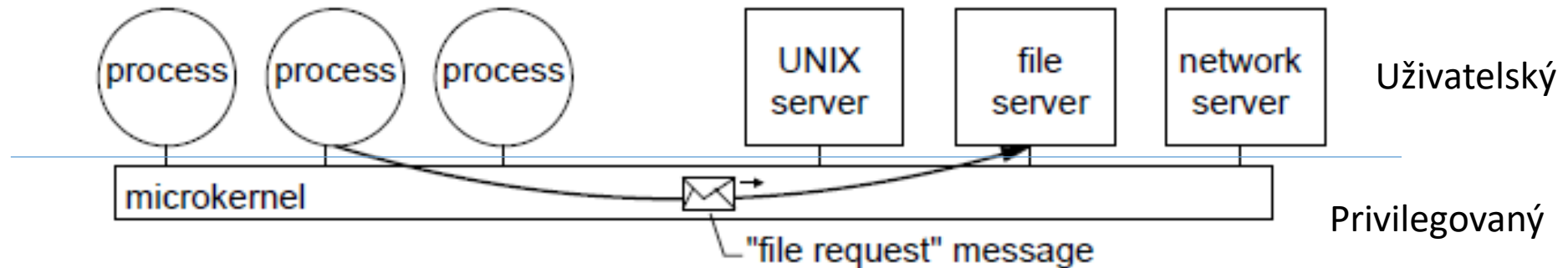
# Typy jádra OS

- monolitické: příklad Linux
  - souborový systém – typicky v jádře
  - vliv na stabilitu celého systému
  - ale může být fs i v userspace – FUSE
- mikrojádro: příklad GNU/Hurd
  - kolekce serverů Hurd běží na mikrojádro GNU/Hurd nebo L4
  - servery: souborový systém, síť, ...
- hybridní: Microsoft Windows

# Jak monolitické jádro implementuje jednotlivé služby?



# Mikrojádro



Mikrojádro – základní služby, běží v privilegovaném režimu

1. proces vyžaduje službu
2. mikrojádro předá požadavek příslušnému serveru nebo službu samo vykoná (když jsou vyžadovány privilegované instrukce)
3. server vykoná požadavek

Snadná vyměnitelnost serveru za jiný

Chyba serveru nemusí být fatální pro celý operační systém (není v jádře) – nespadne celý systém

Distribuované systémy - server může běžet i na jiném uzlu sítě

# Mikrojádro – poznámky

- Mikrojádro může běžet na 1 PC
  - Často mluvíme o modelu klient-server, lze snadno použít pro vytvoření distribuovaného systému - to ale neznamená, že by nefungovalo na 1 uzlu (tedy 1 PC)
  - Server může být (a nejčastěji je) proces běžící na stejném uzlu
- Skládá se mikrojádro + servery
  - V této kombinaci zastanou stejnou službu jako monolitické jádro
  - Například Debian GNU/Hurd lze nainstalovat na PC
    - Mikrojádro Mach
    - Sada serverů Hurd

# Mikrojádro - poznámky

- Samotné mikrojádro by bez serverů bylo k ničemu
- Některé činnosti musí stále vykonávat mikrojádro (je v privilegovaném režimu, tedy pouze zde lze vykonávat privilegované instrukce)



# Info o systémech (wikipedia)

## Windows 7

Web:	<a href="#">Windows 7</a> 
Vyvíjí:	<a href="#">Microsoft</a>
Rodina OS:	<a href="#">Windows NT</a>
Druh:	<b>Uzavřený vývoj</b>
Aktuální verze:	Service pack 1 SP1 / 15.3.2011
Způsob aktualizace:	<a href="#">Windows Update</a>
Správce balíčků:	<a href="#">Windows Installer</a>
Podporované platformy:	x86, x86_64
Typ kernelu:	<b>Hybridní jádro</b>
Implicitní uživatelské rozhraní:	Grafické uživatelské rozhraní
Licence:	Microsoft <a href="#">EULA</a>
Stav:	finální verze

## Linux



Rodina OS:	<a href="#">Unix-like</a>
Aktuální verze:	3.2 / 4. ledna 2012
Podporované platformy:	<a href="#">IA-32</a> , <a href="#">x86-64</a> , <a href="#">PowerPC</a> , <a href="#">ARM</a> , <a href="#">m68k</a> , <a href="#">DEC Alpha</a> , <a href="#">SPARC</a> , <a href="#">hppa</a> , <a href="#">IA-64</a> , <a href="#">MIPS</a> , <a href="#">s390</a> a další
Typ kernelu:	<b>Monolitické jádro</b>
Implicitní uživatelské rozhraní:	<a href="#">GNOME</a> , <a href="#">KDE</a> , <a href="#">Xfce</a> a jiné
Licence:	<a href="#">GNU GPL</a> a jiné
Stav:	Aktuální

A blue horizontal scroll graphic with rounded ends and a vertical strip on the left side, resembling a rolled-up document.

Procesy, stavy procesů, plánování

# Významné identifikátory

- PID            id procesu, získáme fcí getpid()
- PPID          id rodiče, získáme fcí getppid()
- TID           id vlákna
- UID           id uživatele

příkazy: id, ps

# Nový proces vykonávající nějaký program

- Windows:

**CreateProcess()**

– proces bude vykonávat kód dle uvedeného programu

- Linux:

kombinace **fork()** a **execve()**

**fork** – nový proces (nový PID), ale stejný kód, liší se jen návratovým kódem forku

**execve** – nahradí kód, který daný proces vykonává

(volání exec většinou tvar execve, execl aj.)

# Jak funguje shell?

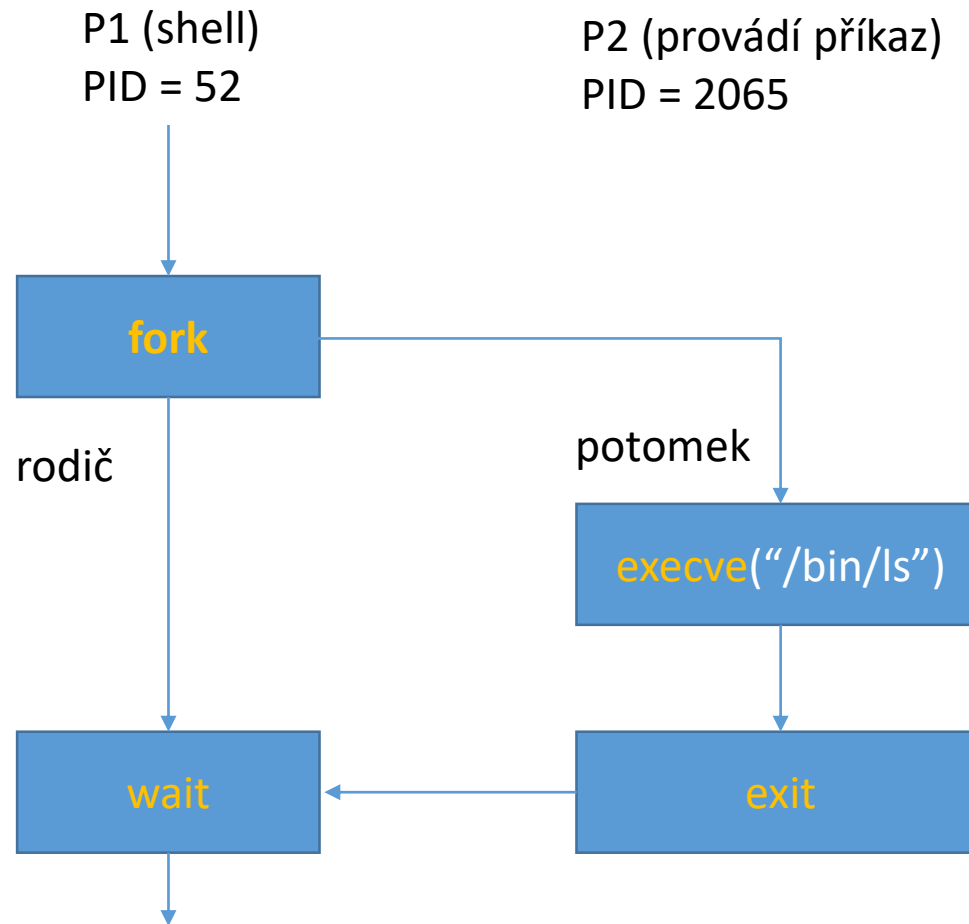
- shell – příkazový interpret, známe `/bin/bash`
- shell čeká na zadání příkazu
- shell se „naforkuje“
- co mu zadáme na příkazovou řádku pustí execem
- čeká na dokončení tohoto potomka (není-li `&` na konci příkazů)
- a pak se cyklus znovu opakuje

# Příklad fork a execve

Potomek může místo sebe spustit jiný program – volání **execve()** – nahradí obsah paměti procesem spouštěným ze zadaného souboru

1. *if (fork() == 0)*
2.       *execve("/bin/ls", argv, envp);*
3. *else*
4.       *wait(NULL);*

# Graf procesů



# Windows (Win32)

- Vytvoření procesu službou **CreateProcess**
- Vytvoří nový proces, který vykonává program zadaný jako parametr
- Mnoho parametrů – vlastnosti procesu



# Ukázka pod Windows

```
STARTUPINFO StartInfo; // name structure
PROCESS_INFORMATION ProcInfo; // name structure
memset(&ProcInfo, 0, sizeof(ProcInfo)); // Set up memory block
memset(&StartInfo, 0, sizeof(StartInfo)); // Set up memory block
StartInfo.cb = sizeof(StartInfo); // Set structure size
int res = CreateProcess(NULL, "MyApp.exe", NULL, NULL, NULL, NULL, NULL, NULL, &StartInfo, &ProcInfo); // starts
MyApp
if (res)
{
    WaitForSingleObject(ProcInfo.hThread, INFINITE); // wait forever for process to finish
    SetFocus(); // Bring back focus
}
```

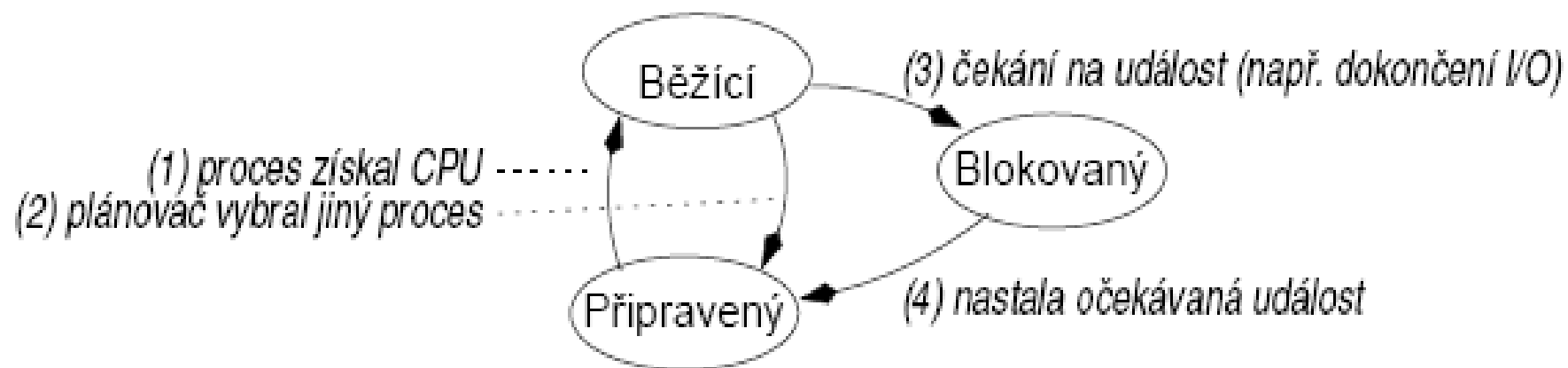
příklad viz

<http://msdn.microsoft.com/en-us/library/windows/desktop/ms682512%28v=vs.85%29.aspx>

# Základní stavy procesu

- **Běžící (running)**
  - skutečně využívá CPU, vykonává instrukce
- **Připravený (ready, runnable)**
  - dočasně pozastaven, aby mohl jiný proces pokračovat
- **Blokovaný (blocked, waiting)**
  - neschopný běhu, dokud nenastane externí událost

# Základní stavy procesu (!!)



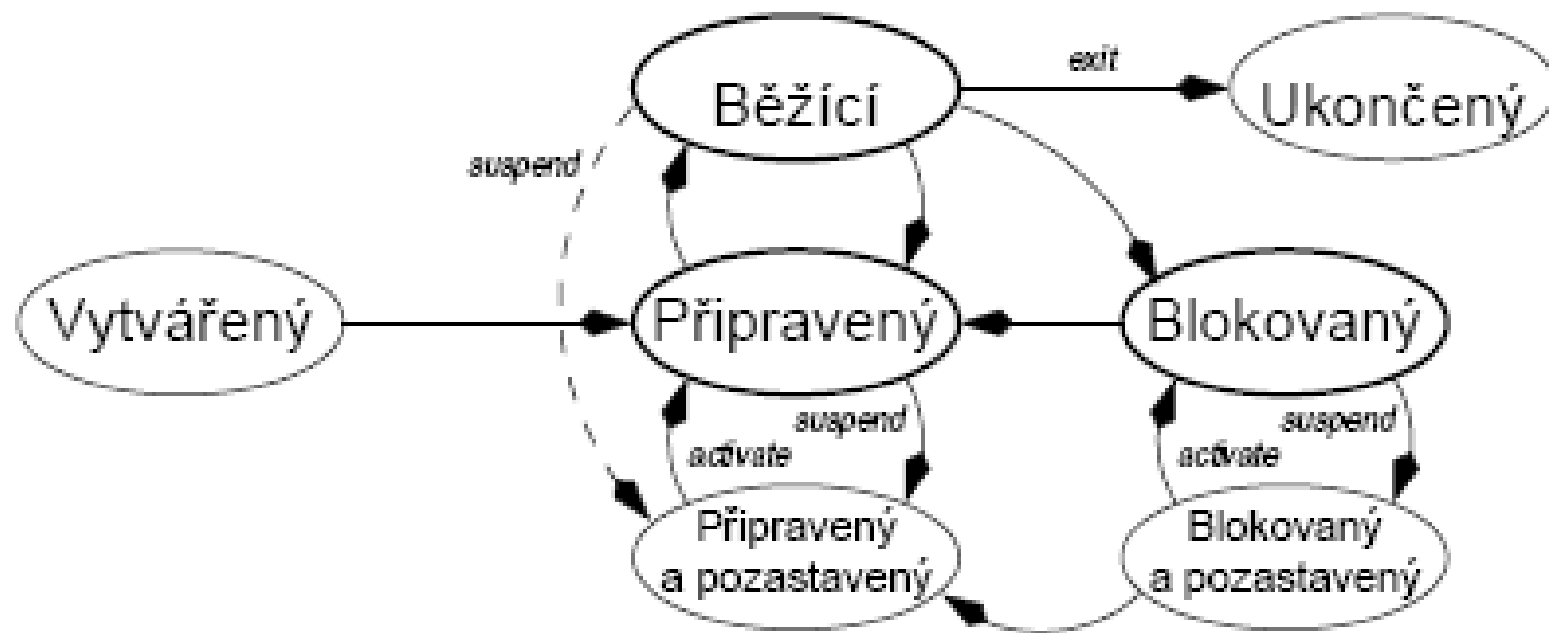
# Přechody stavů procesu

1. Plánovač **vybere** nějaký proces
2. Proces je **pozastaven**, plánovač vybere jiný proces (typicky - vypršelo časové kvantum)
3. Proces se **zablokuje**, protože čeká na událost (zdroj – disk, čtení z klávesnice)
4. **Nastala** očekávaná **událost**, např. zdroj se stal dostupný

# Stavy procesů

- Jádru OS obsahuje plánovač
- Plánovač určuje, který proces bude běžet
- Nad OS řada procesů, střídají se o CPU
- Stav procesu **pozastavený**
- V některých systémech může být proces **pozastaven** nebo **aktivován**
- V diagramu přibudou **dva** nové stavy

# Stavy procesů



# Tabulka procesů

OS si musí vést evidenci, jaké procesy v systému v danou chvíli existují.

Tato informace je vedena v **tabulce procesů**.

Každý proces v ní má záznam, a tento záznam se nazývá **process control block (PCB)**.

Na základě informací zde obsažených se plánovač umí rozhodnout, který proces dále poběží a bude schopen tento proces spustit ze stavu, v kterém byl naposledy přerušen.

# PCB (Process Control Block) !

- OS udržuje tabulku nazývanou **tabulka procesů**
- Každý proces v ní má položku zvanou **PCB** (Process Control Block)
- PCB obsahuje všechny informace potřebné pro **opětovné spuštění** přerušného procesu
  - Procesy se o CPU střídají, tj. jeho běh je přerušovaný
- Konkrétní obsah PCB – různý dle OS
- Pole správy **procesů**, správy **paměti**, správy **souborů**  
(!!)



# Položky - správa procesů

- Identifikátory (číselné)
  - Identifikátor procesu - PID
  - Identifikátor uživatele - UID
- Stavová informace procesoru
  - Univerzální registry,
  - Ukazatel na další instrukci - PC
  - ukazatel zásobníku SP
  - Stav CPU – PSW (Program Status Word)
- Stav procesu (běžící, připraven, blokován)
- Plánovací parametry procesu (algoritmus, priorita)

# Položky – správa procesů II

- Odkazy na rodiče a potomky
- Účtovací informace
  - Čas spuštění procesu
  - Čas CPU spotřebovaný procesem
- Nastavení meziprocesové komunikace
  - Nastavení signálů, zpráv

# Položky – správa paměti

- Popis paměti

- Ukazatel, velikost, přístupová práva

1. Úsek paměti s kódem programu

2. Data – hromada

- Pascal – new release
  - C – malloc, free

3. Zásobník

- Návrátové adresy, parametry funkcí a procedur, lokální proměnné

# Položky – správa souborů

- **Nastavení prostředí**
  - Aktuální pracovní adresář
- **Otevřené soubory**
  - Způsob otevření – čtení / zápis
  - Pozice v otevřeném souboru

# PCB

Pointer	Process state
Process number	
Program counter	
<b>Registers</b>	
Memory limits	
List of open files	
...	

# Systemy s časovým kvantem

Jakou vlastnost potřebuje mít operační systém, aby mohl plánovat procesy po časových kvantech?

⇒ musí mít časovač

každý tick časovače – hw přerušení (drát, přerušovací vektor, obsluha přerušení)

v rámci obsluhy přerušení zvýší počítadlo tiků

po určitém počtu tiků – uplynulo kvantum – plánovač zjistí kdo dál poběží – dispatcher přepne kontexty procesů

# Plánování procesů – 3 druhy plánovače

- **Krátkodobé – CPU scheduling**  
kterému z připravených procesů bude přidělen procesor;  
vždy je ve víceúlohovém systému
- **Střednědobé – swap out**  
odsun procesu z vnitřní paměti na disk
- **Dlouhodobé – job scheduling**  
výběr, která úloha bude spuštěna  
dávkové zpracování  
(dostatek zdrojů – spust' proces)
- **Liší se – frekvencí spouštění plánovače**

# Stupeň multiprogramování

- Počet procesů v paměti
- Zvyšuje: long term scheduler (dluhodobý)
- Snižuje: middle term scheduler (střednědobý)

Kdy je vhodné snížit stupeň multiprogramování?

- Málo dostupné RAM – soupeří o ní – neustálý přesun stránek mezi RAM a swapem -> zahlcení (trashing)



# Zombie a sirotek

- **Zombie**

- Proces dokončil svůj kód
- Stále má záznam v tabulce procesů
- Čekání, dokud rodič nepřečte exit status (voláním `wait()` ); příkaz `ps` zobrazuje stav “Z”

- **Sirotek**

- Jeho kód stále běží, ale skončil rodičovský proces
- Adoptován procesem `init`

# Zombie - ukázka

```
#include <stdio.h>
int main (void) {
    int i,j;
    i = fork();
    if (i == 0)
        printf ("Jsem potomek s pidem %d, rodic ma %d\n", getpid(), getppid());
    else {
        printf ("Jsem rodic s pidem %d, potomek ma %d\n", getpid(), i);
        for (j=10; j<100; j++) j=11; // rodič neskončí, nekonečná smyčka
    }
}
```



Vlákna

# Vlákna

Vlákna mohou být implementována:

- V jádře
- V uživatelském prostoru
- Kombinace

Zná jádro pojem vlákna?

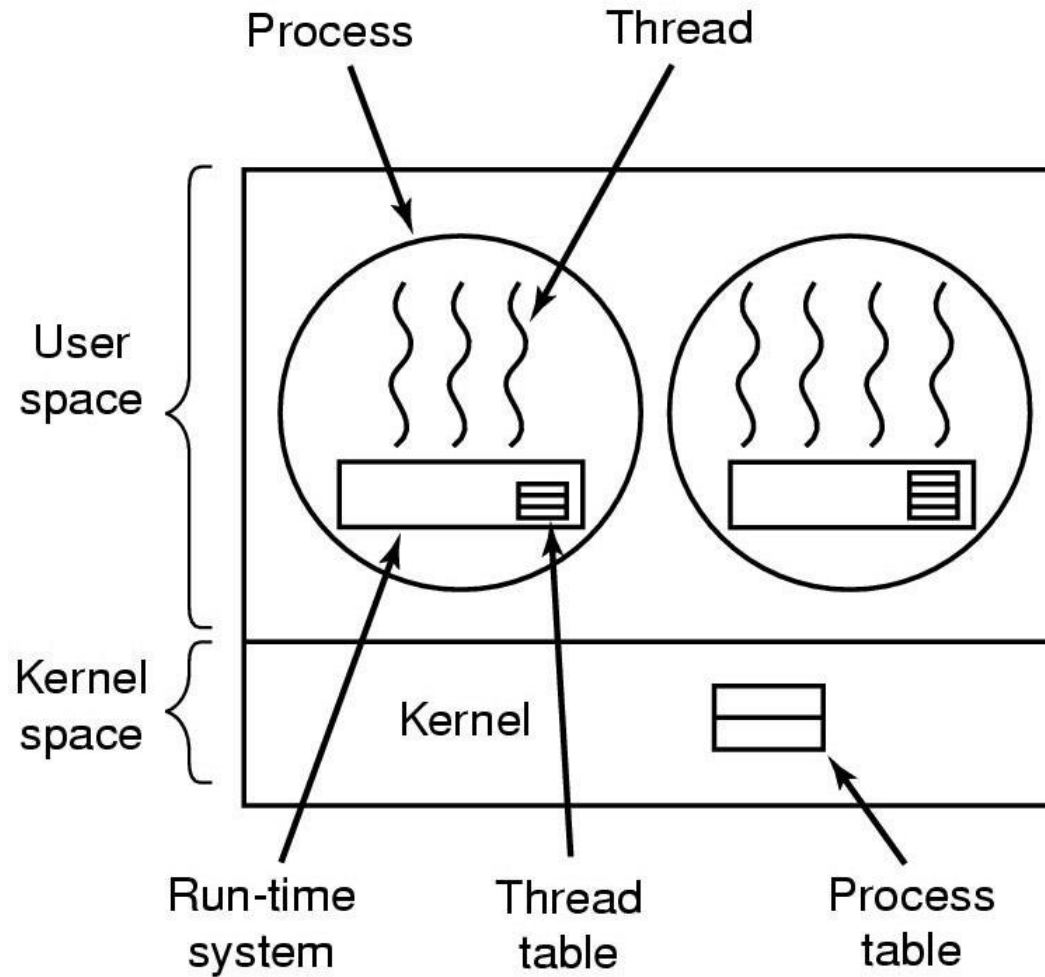
Jsou v jádře plánována vlákna nebo procesy?

# Vlákna v User Space

Jádro plánuje procesy,

O vláknech nemusí vůbec vědět.

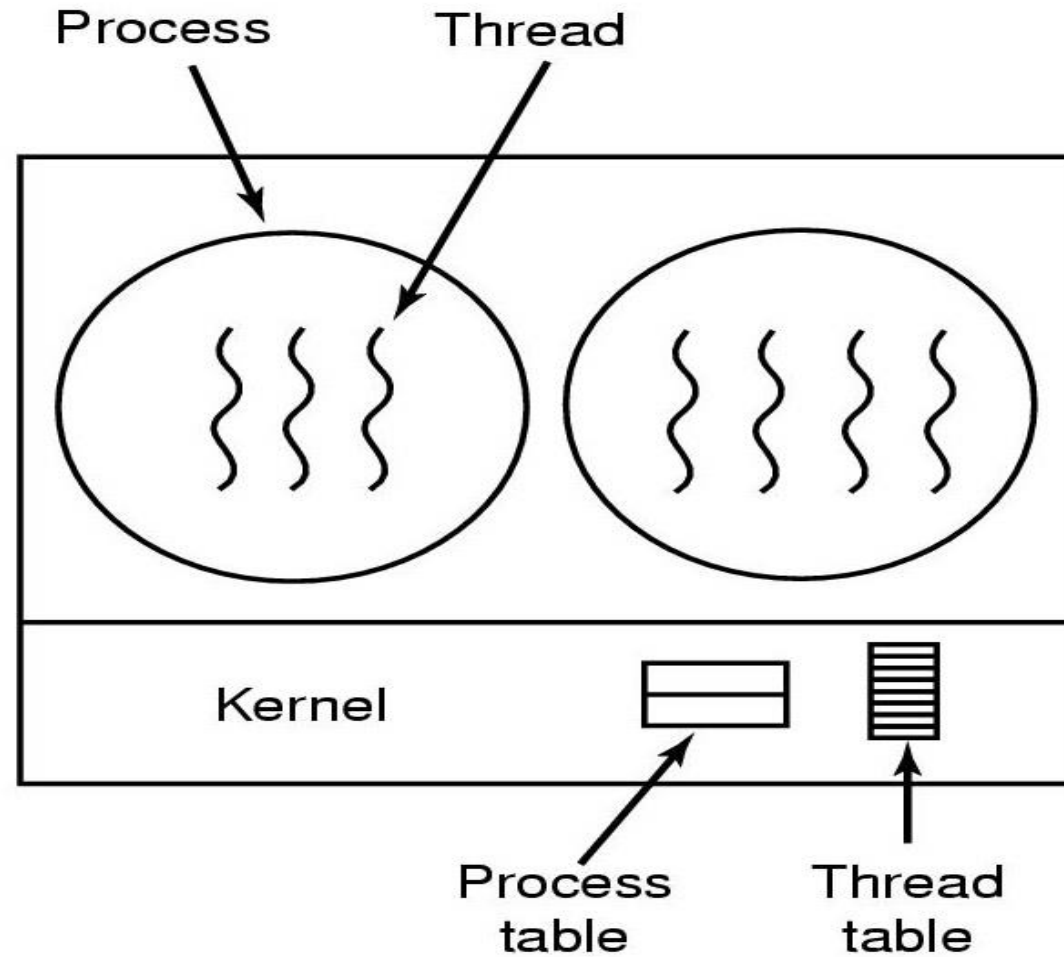
Pokud vlákno zavolá systémové volání, celý proces se zablokuje



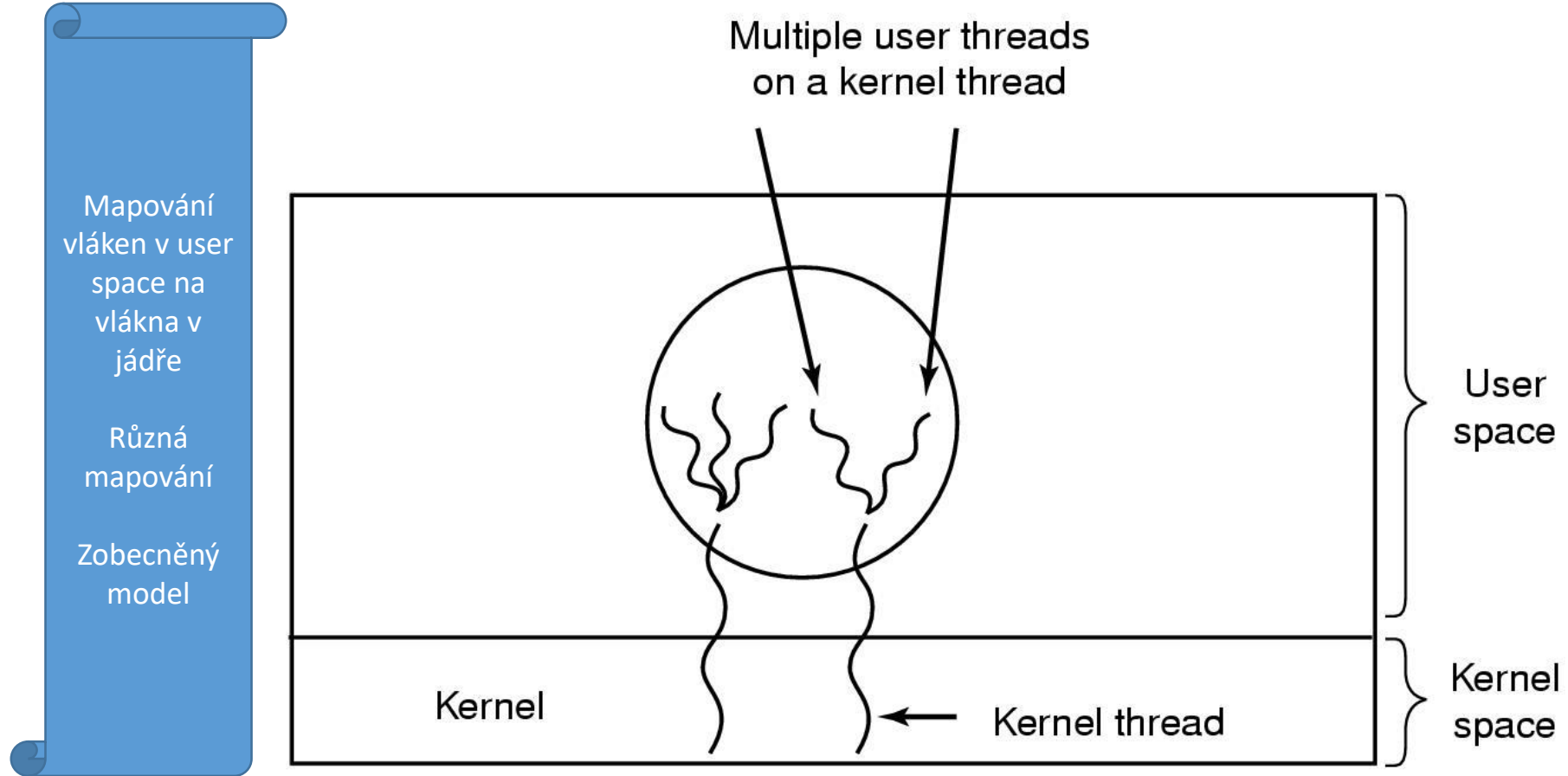
# Vlákna v jádře

Jádro  
plánuje  
jednotlivá  
vlákna.

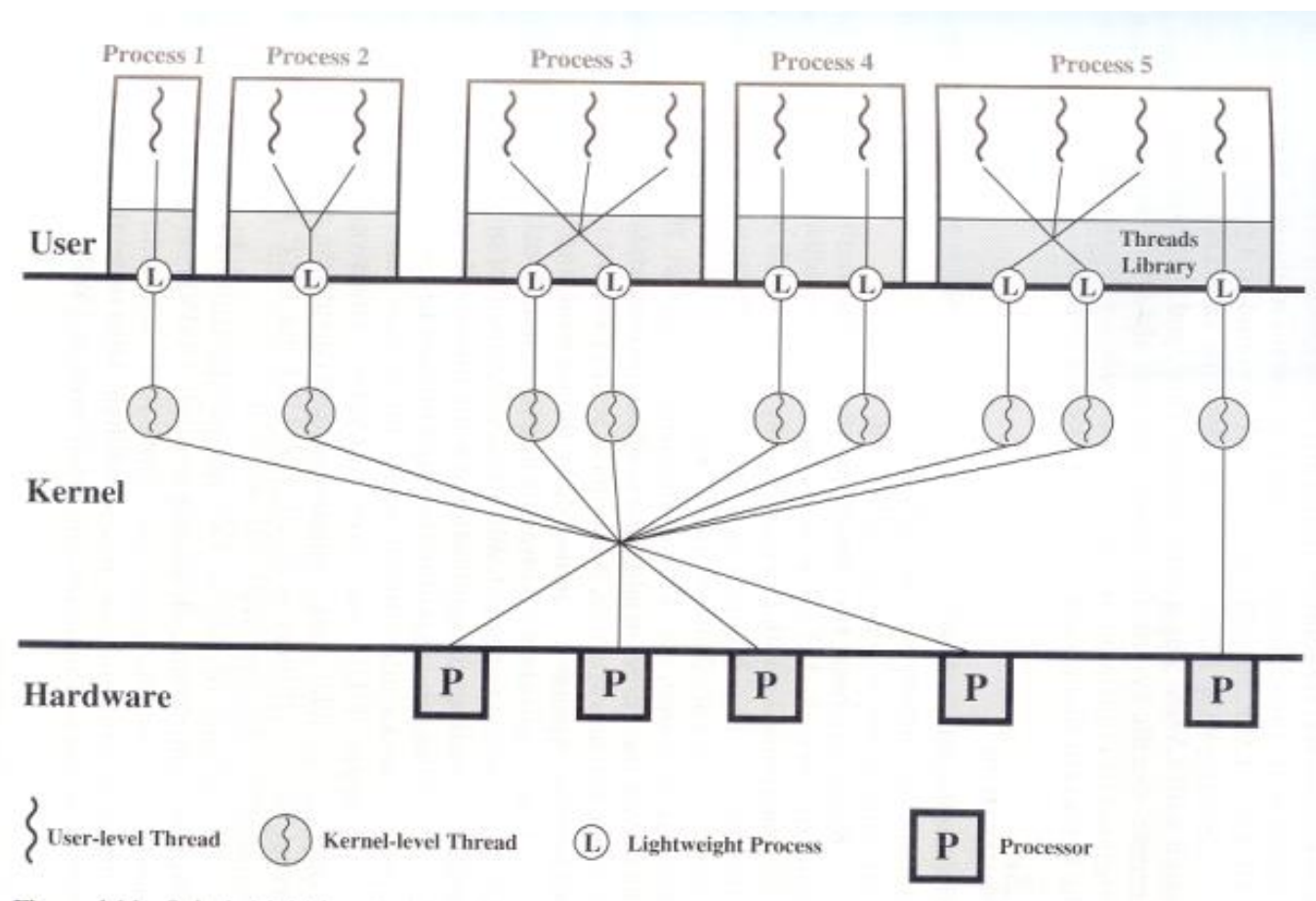
Kromě  
tabulky  
procesů má  
i tabulku  
vláken.



# Hybridní implementace



# Vlákná Solaris





# Vlákna – C – pthread.h

1. **Management vláken** (**create, detach, join**)
2. **Mutexy** (**create, destroy, lock, unlock**)
3. **Podmínkové proměnné** (**create, destroy, wait, signal**)
4. **Další synchronizace** (**read-write locks, bariéry**)

Rozhraní specifikované IEEE POSIX 1003.1c (1995)

# Vlákná – C – pthread.h

funkce	popis
<b>pthread_create</b>	<b>Vytvoří nové vlákno.</b> Jako kód vlákna se bude vykonávat funkce, která je zadaná jako parametr této funkce Defaultně je vytvořené vlákno v joinable stavu.
<b>pthread_join</b>	Čeká na dokončení jiného vlákna, vlákno na které se čeká musí být v joinable stavu
<b>pthread_detach</b>	Vlákno bude v detached stavu – nepůjde na něj čekat pomocí pthread_join Paměťové zdroje budou uvolněny hned, jak vlákno skončí (x zabrání synchronizaci)
<b>pthread_exit</b>	Naše vlákno končí, když doběhne funkce, kterou vykonává, nebo když zavolá pthread_exit

# Vlákna - Java

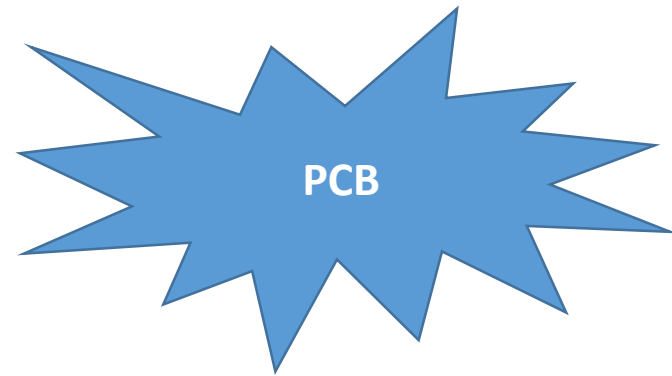
- Vlákno – instance třídy `java.lang.Thread`
- Odvodit potomka, překrýt metodu `run()`
  - Vlastní kód vlákna
- Spuštění vlákna – volání metody `start()` třídy Thread
- Další možnost - třída implementující rozhraní `Runnable`

```
class Něco implements Runnable {  
    public void run() { ... } }
```

# Proces UNIXU

## – PCB obsahuje informace:

- Proces ID, proces group ID, user ID, group ID
- Prostředí
- Pracovní adresář
- Instrukce programu
- Registry
- Zásobník (stack)
- Halda (heap)
- Popisovače souborů (file descriptors)
- Signal actions
- Shared libraries
- IPC (fronty zpráv, roury, semaforey, sdílená paměť)



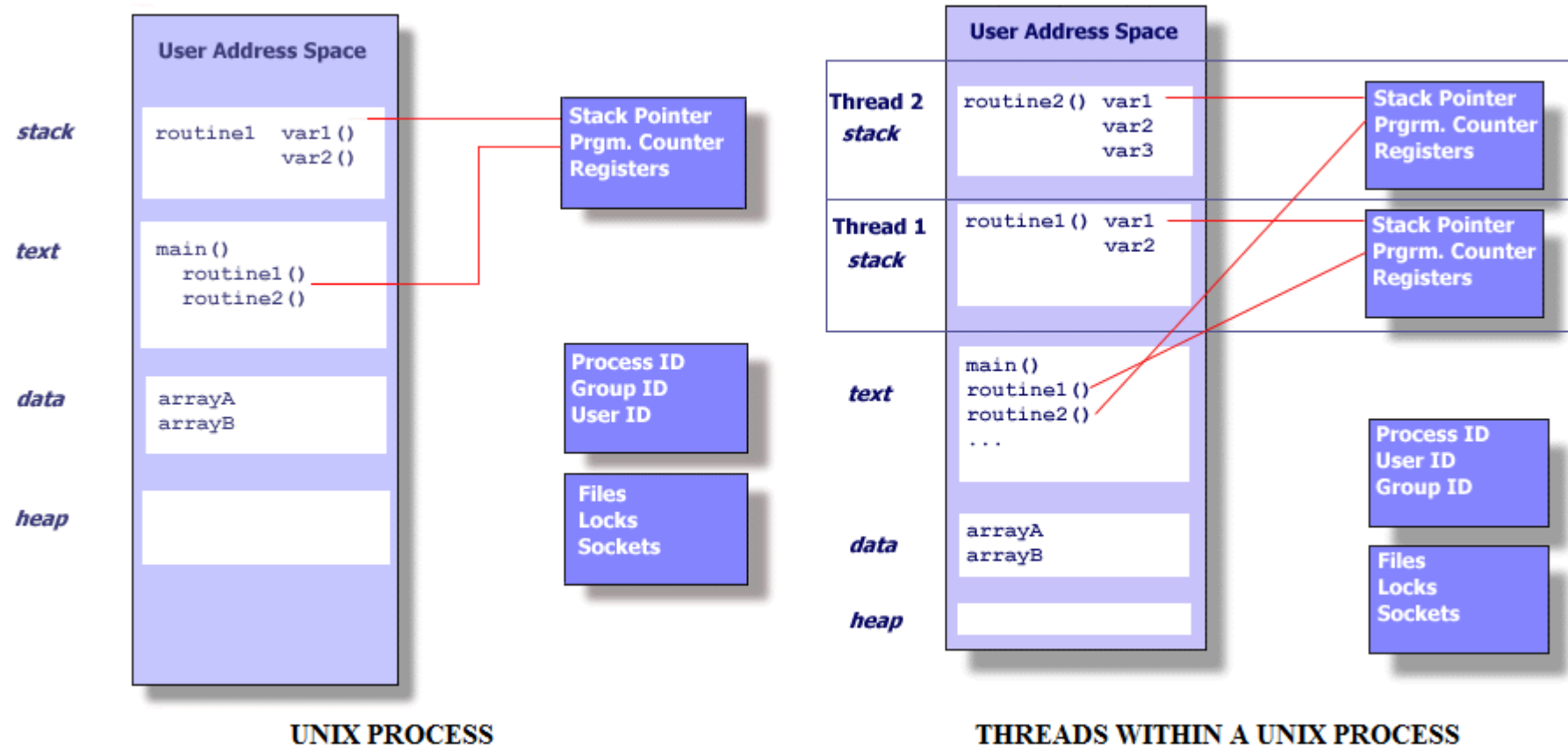
# Vlákno má vlastní (!!):

- Zásobník (stack pointer)
- Registry
- Plánovací vlastnosti (policy, priority)
- Množina pending a blokováných signálů
- Data specifická pro vlákno

Všechna vlákna uvnitř stejného procesu sdílejí stejný adresní prostor

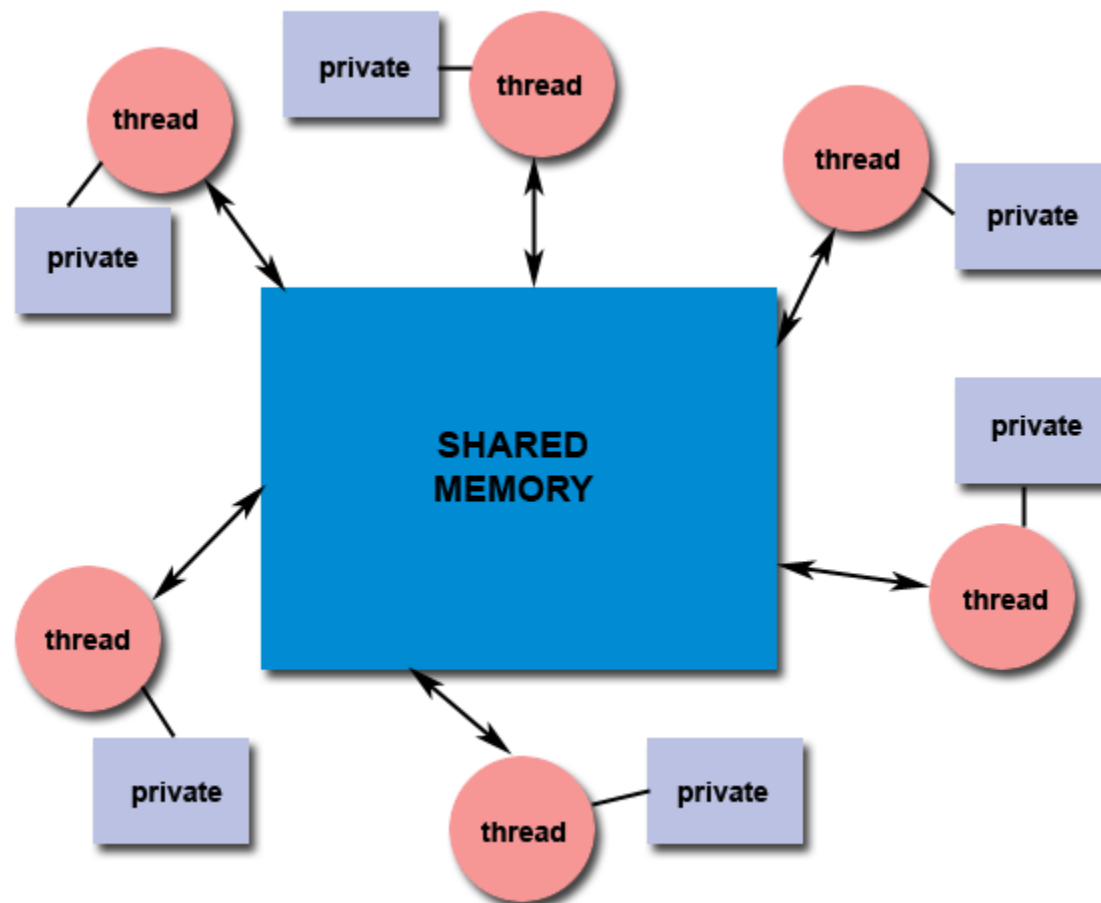
Mezivláknová komunikace je efektivnější a snadnější než meziprocesová

proces vs. proces s více vlákn  
(rozdělení paměti je jen ilustrativní)

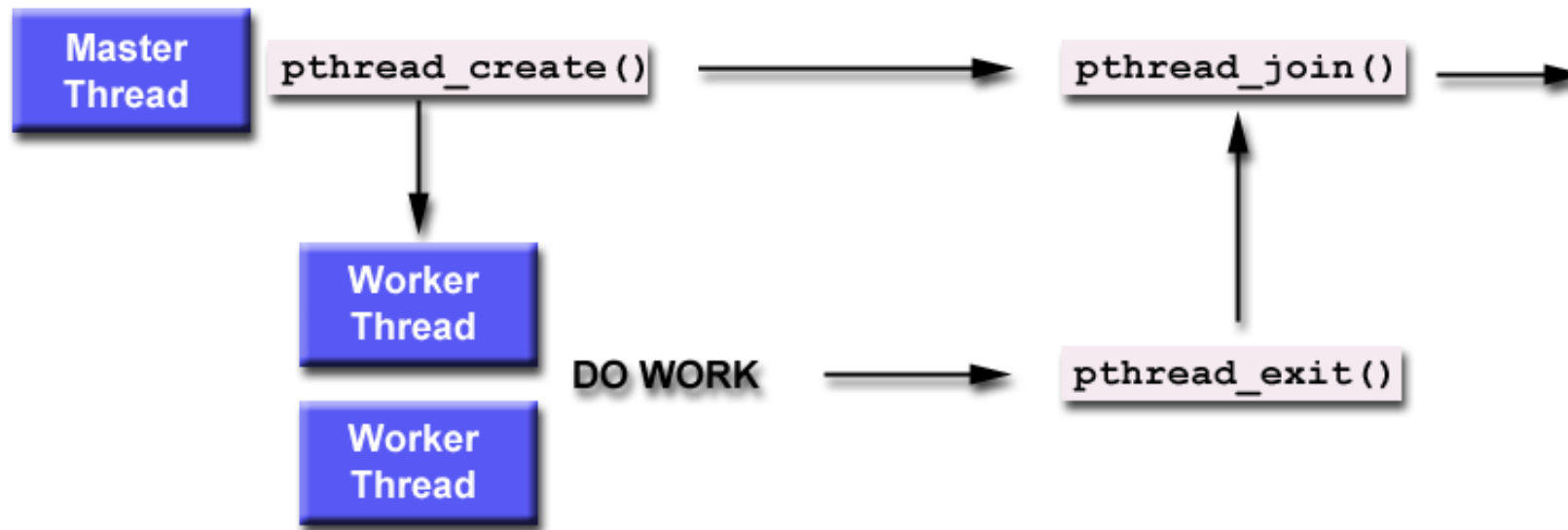


# Globální a privátní paměť vlákna

více vláken  
stejného procesu



# Čekání na dokončení vláken

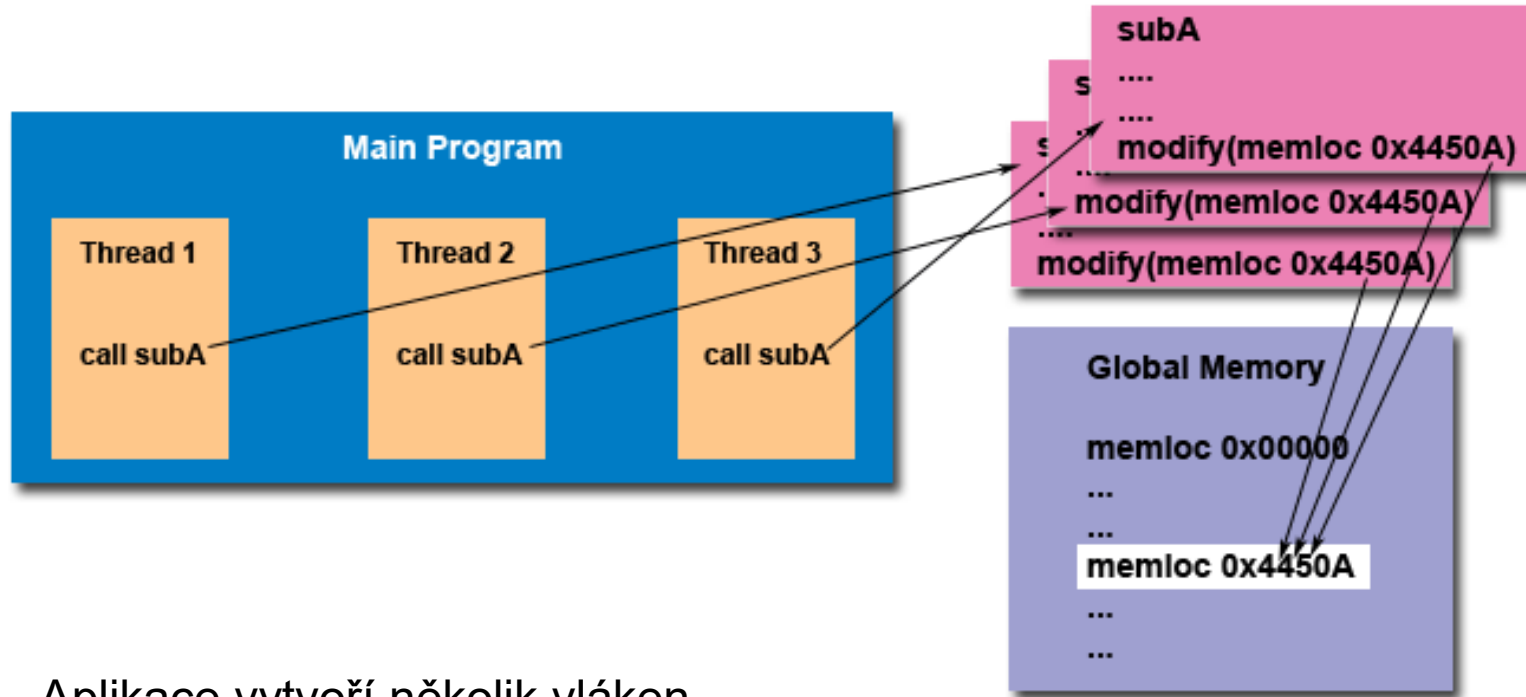




# Možnosti ukočení vlákna

- Vlákno dokončí „proceduru vlákna“
- Vlákno kdykoliv zavolá `pthread_exit()`
- Vlákno je zrušené jiným vláknem `pthread_cancel()`
- PROCES při zavoláním `execve()` nebo `exit()`
- Pokud `main()` skončí první bez explicitního volání `pthread_exit()`

# Vláknová bezpečnost (thread-safe)



Aplikace vytvoří několik vláken

Každé vlákno vyvolá stejnou rutinu

Tato rutina modifikuje společná globální data

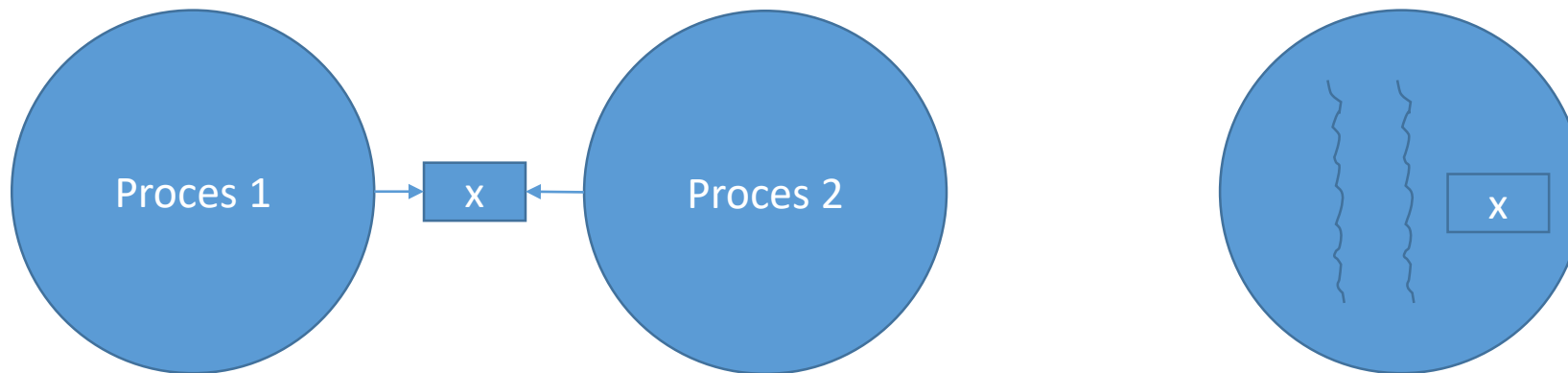
– pokud nemá synchronizační mechanismy, není thread-safe

A blue horizontal scroll graphic with rounded ends and a vertical strip on the left side, resembling a rolled-up document.

Souběh, kritická sekce

# Sdílená paměť mezi procesy nebo vlákny

- Procesy požádají operační systém o přidělení úseku sdílené paměti - voláním `shmget()`
- Vlákna stejného procesu paměť sdílejí



# Příklad dvou procesů

cobegin

...

$x := x + 1;$

...

||

...

$x := x + 1;$

...

coend

1.proces

2.proces

společná  
paměť:

$x$

Příkaz – rozpadne se na nízkoúrovňové instrukce

$x := x + 1;$


1. Načtení hodnoty  $x$  do registru (LD R, x)
2. Zvýšení hodnoty  $x$  (INC R)
3. Zápis nové hodnoty do paměti (LD x, R)

Pokud oba procesy provedou příkazy **sekvenčně**, bud  
mít  $x$  správně  **$x+2$**

# Chybné pořadí vykonání

Přepnutí v nevhodném okamžiku.. Pseudoparalelní běh

1.	P1: LD R, x	// x je 0, R je 0
2.		
3.		
4.		
5.	P1:	// x je 1, R je 0 – rozpor
6.	INC R	// x je 1, R je 1
7.	LD x, R	// x je 1, R je 1



Výsledek – **chyba**, neprovedlo se každé zvětšení, místo 2 je 1

# Výskyt souběhu

- časový souběh se projevuje **nedeterministicky**
  - **může nastat kdykoliv**
- většinu času běží programy bez problémů
- hledání chyby je obtížné



# Kritická sekce

- sekvenční procesy
  - komunikace přes společnou datovou oblast
- **kritická sekce** (critical section, region)
  - místo v programu, kde je prováděn přístup ke společným datum – read x, write x, x++
- úloha – jak implementovat, aby byl v kritické sekci v daný okamžik pouze 1 proces

# Počet kritických sekcí

- Kritická sekce nemusí být jedna
- Pokud procesy sdílejí tři proměnné  $x$ ,  $y$ ,  $z$ 
  - Každá z nich představuje  $KS_x$ ,  $KS_y$ ,  $KS_z$

Mohli bychom sice říci, že jde o jednu KS, ale potom bychom zbytečně blokovali přístup k  $y$ , řešíme-li souběh nad  $x$  atd.

Analogie: když potřebujeme zamknout řádku tabulky v databázi, není potřeba zamykat celou tabulku, která může mít třeba milion záznamů – vliv na výkon systému

# Pravidla pro řešení časového souběhu

1. **Vzájemné vyloučení** - žádné dva procesy nesmějí být **současně** uvnitř své kritické sekce
2. Proces běžící **mimo kritickou sekci nesmí blokovat** jiné procesy (např. jim **bránit ve vstupu** do kritické sekce)
3. Žádný proces nesmí na vstup do své kritické sekce **čekat nekonečně dlouho** (jiný vstupuje **opakovaně**, neumí se **dohodnout** v konečném čase, kdo vstoupí první)

# Možnosti řešení časového souběhu

1. Zákaz přerušení
2. Aktivní čekání - TSL
3. Zablokování procesu – semafor, monitor, mutex

# Synchronizační mechanismy

Semafor, mutex, monitor, TSL, CAS

# Obeční popis – synchronizační primitiva

- Definice (sem: datové struktury, operace)
- Použití (sem: ošetření KS, synchronizace, ...)
- Implementace

U každého synchronizačního primitiva vždy uvažujte, jak daný mechanismus definovat, uveďte příklad jeho použití a návrh, jak by šel tento mechanismus implementovat s využitím jiných primitiv.

# Semafor

- Datové struktury:
  - `int S;` // celé číslo s – hodnota semaforu
  - `queue f;` // fronta procesů (vláken) blokováných nad semaforem
- Operace:
  - `P()` // snižuje hodnotu semaforu, může být blokující
  - `V()` // zvyšuje hodnotu semaforu

# Semafor - implementace

Operace P:

```
if s > 0
    s--;
else
    blokuj vlákno ve frontě (f);
```

Operace V:

```
if neprazdna_fronta(f)
    vzbud_jeden_z(f);
else
    s++;
```

Kód uvnitř operací P a V musí být atomický, než se vykoná, nikdo další nevstoupí do obsluhy P ani V



# Semafor – ošetření kritické sekce

- Používáme semafor s počáteční hodnotou 1 (dovnitř smí nanejvýš jeden)

semaphore s = 1;

Vlákno 1:

```
P(s);  
kritická sekce  
V(s);
```

Vlákno 2:

```
P(s);  
kritická sekce  
V(s);
```

Vlákno 3:

```
P(s);  
kritická sekce  
V(s);
```

# Semafor – předávání řízení – štafetový kolík

- Používáme semafor s počáteční hodnotou 0 (blokace, čekáme, až nás někdo uvolní)

semaphore s1 = 0;

semaphore s2 = 0;

Vlákno 1:

“Přeji Vám”

V(s1);

Vlákno 2:

P(s1);

“hezký ”

V(s2);

Vlákno 3:

P(s2);

“den!”

# Semafor – synchronizace

- Používáme semafor s počáteční hodnotou  $\Rightarrow 1$ , např. 5 (kolik volných položek v buffferu můžeme zaplnit)
- Klasická úloha producent – konzument
- Buffer na N položek
  - Hlídáme, kolik zbývá ještě volných (když chce producent zapisovat)
  - Hlídáme, kolik je zaplněných (chce-li konzument číst)

# Samostatná práce

Proces P1:

Print("Ahoj ")

Print("je")

Proces P2:

Print("dnes ")

Print("krásně.")

Proces P3:

Print("venku ")

P1,P2, P3 běží paralelně.  
Ošetřete SEMAFORY,  
aby vždy byla vypsána správná  
věta:

Ahoj dnes je venku krásně.

# Producent & Konzument - implementace

semaphore

e = N;                   // prázdných položek

f = 0;                   // plných položek

m = 1;                  // vzájemné vyloučení (mutex)

# P&K – implementace II. (!)

```
cobegin
```

```
  while true do { producent
```

```
  begin
```

```
    produkuj záznam;
```

```
    P(e);
```

```
    // je volná položka?
```

```
    P(m); vlož do bufferu; V(m);
```

```
    V(f);
```

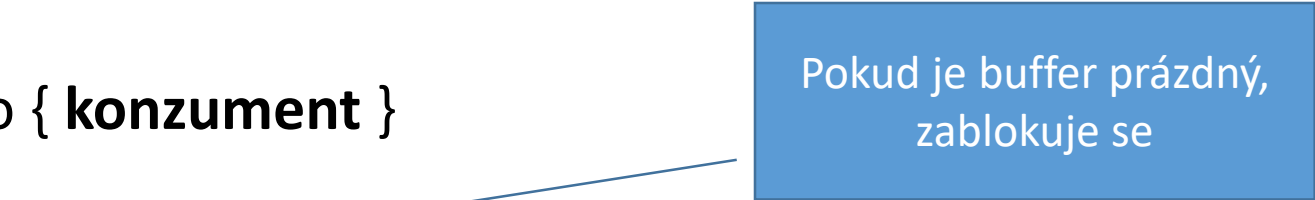
```
    // zvětší obsazených
```

```
  end {while}
```

Není-li volná položka v  
bufferu, zablokuje se

# P&K – implementace III.

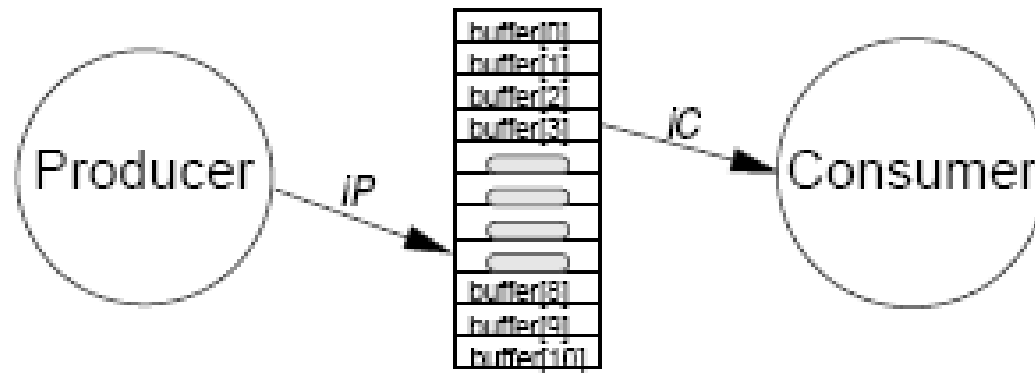
```
||  
while true do { konzument }  
begin  
    P(f); // je plná nějaká položka?  
    P(m); vyber z bufferu; V(m);  
    V(e); // zvětši počet prázdných  
    zpracuj záznam;  
end {while}  
coend.
```



Pokud je buffer prázdný,  
zablokuje se

# P&K poznámky

Vyrovnávací paměť se často implementuje jako **pole** – buffer [0..N-1]





# Semafor - nevýhody

- Snadno vytvoříme deadlock
  - Dávat pozor na počáteční hodnotu semaforu
- Volání  $P()$  a  $V()$  mohou být roztroušeny v kódu aplikace daleko od sebe, ztrácí se přehlednost

# Binární semafor

- Může nabývat pouze hodnot  $s = 0$  nebo  $s = 1$
- Pokud  $s = 1$  a zavoláme  $V(s)$  -> chyba
- Kontrola, zda nedochází k předchozí situaci – snáze odhalíme chybu v kódu, protože chyba v souběhu nad kritickou sekcí se nemusí vždy projevit (revizor vás také vždy nechytne, když jedete načerno).

# Semafor - C

## Ošetření KS semaforem:

- `#include <semaphore.h>`
- `sem_t s;`
- `sem_init(&s, 0, 1);`

.. využijeme semafor  
.. typ semafor  
.. inicializace semaforu na hodnotu **1** !!

- `sem_wait(&s);`
- KS
- `sem_post(&s);`

.. operace P(s);  
.. kritická sekce  
.. operace V(s);

# Semafor – C - inicializace

```
sem_init(&s, 0, 1);
```

Semafor s

Počáteční hodnota 1

0 ...	semafor sdílený vlákny jednoho procesu	
1 ...	semafor sdílený mezi procesy, měl by být v	regionu sdílené
paměti		

# Semafor – C - pojmenovaný

- místo inicializace **sem\_init** se otevírá **sem\_open**  
`#include <semaphore.h>`

```
int main() {  
    sem_t *sem1;  
    sem1 = sem_open("/mujsem1", O_CREAT, 0777, 10);  
    ...  
    sem_wait(), sem_trywait(), sem_post(), sem_getvalue()  
    sem_close(sem1);  
    sem_unlink("/mujsem1");  
}
```

# Semafor - Java

- `import java.util.concurrent.Semaphore;`
- `Semaphore sem = new Semaphore(1);`
- **`sem.acquire();`**
- *.. kritická sekce ..*
- **`sem.release();`**

# Mutex

- Pokud nám stačí pouze použití k ošetření kritické sekce
- Mutex má často následující omezení:
  - Kdo zamknul mutex, ten jej musí také odemknout
- Použití pro kritickou sekci – OK
- Použití pro štafetový kolík – NE (jeden zamyká, jiný odmyká)

# Mutex reentrantní

- Stejně vlákno může mutex zamknou **násobně**
  - Běžný mutex – druhé zamknutí - zablokování
- Ale potom musí dané vlákno provést stejný počet odemknutí



# Mutex reentrantní - příklad

```
var m : Mutex // A non-recursive mutex, initially unlocked.  
  
function lock_and_call(i : Integer)  
    m.lock()  
    callback(i)  
    m.unlock()  
  
function callback(i : Integer)  
    if i > 0  
        lock_and_call(i - 1)
```

Zdroj: wikipedia

Nerentrantní mutex způsobí deadlock, reentrantní zde bude fungovat v pořádku

# Mutex – C – pthread.h

1. `pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;`

2. `pthread_mutex_lock(&lock);`      `-- zamkni`

## 3. Kritická Sekce

4. `pthread_mutex_unlock(&lock);`      `-- odemkni`

# Mutex – C – pthread.h - pokračování

- **pthread\_mutex\_trylock(&lock);**
  - zkusí zamknout, pokud má zámek někdo jiný vrátí se hned
- **pthread\_mutex\_destroy(&lock);**
  - zruší zámek

# Mutex - Java

- `import java.util.concurrent.locks.Lock;`
- `import java.util.concurrent.locks.ReentrantLock;`
- `Lock lock = new ReentrantLock();`
- `lock.lock();`                      `-- zamkni`
- `KS`                                  `-- kritická sekce`
- `lock.unlock();`                      `-- odemkni`

# Časté chyby !!

- Nelze říct, že do kritické sekce smí více procesů
  - To že je sekce kritická = uvnitř smí být pouze jeden proces (vlákno)
- Semafor s hodnotou 5 nechrání přístup ke kritické sekci, ale může udávat, kolik položek lze ještě např. vložit do bufferu s omezenou velikostí (viz producent – konzument)

# Monitor

- Výhody oproti semaforu
  - Ošetření kritických sekcí, synchronizace v jednom modulu (není roztroušené po celém kódu programu) – přehlednost
- Obecný monitor
  - V monitoru může být  $N$  procesů (vláken)
  - Z nich 1 je aktivní a  $N-1$  je blokových
- Podmínková proměnná
  - Představuje frontu procesů blokových nad podmínkou (!!)

# Monitor – příklad - hospoda

- Funkce zákazníka:  
`getpivo()`, `getvino()`
- Funkce závozníka:  
`zavoz(int piv, int vin)`
- Zákazník volá `getpivo()`
  - Není-li pivo dostupné, blokuje nad podmínkovou proměnnou
- Závozník kromě zvýšení proměnné počtu piv a vin signalizuje závoz piva – přivezeli 100 piv, 100x signalizuje `signal(c1)`

# Monitor – příklad - hospoda

```
monitor hospoda {  
    int piv;           // počet piv na skladě  
    int vin;           // počet vín na skladě  
    condition c1;      // čekáme na pivo  
    condition c2;      // čekáme na víno  
  
    void get_pivo();    // zákazník chce pivo  
    void get_vivo();    // zákazník chce víno  
  
    void zavoz(int zpiv, int zvin);  
                        // závozník dovezl daný počet piv a vín  
}
```



# Monitor hospoda - zákazník

```
void get_pivo() {  
    if (piv < 1)  
        wait(c1);  
    piv --;  
}
```

Lepší je použití while místo if  
„dvakrát měř, jednou řež“  
po vzbuzení otestovat, jestli je podmínka opravdu splněna  
Pokud závozník používá broadcast, tak je while nutností.

# Monitor hospoda – vylepšení – zákazník

```
void get_pivo() {  
    while (piv < 1)  
        wait(c1);  
    piv --;  
}
```

# Monitor hospoda - závoz

```
void zavoz(int zpiv, int zvin) {  
    int i,j;  
  
    piv = piv + zpiv;  
    for (i=0;i<zpiv;i++) signal(c1);  
  
    vin = vin + zvin;  
    for (j=0;j<zvin;j++) signal(c2);  
  
}
```

# Monitor hospoda – závoz - broadcast

```
void zavoz(int zpiv, int zvin) {  
    int i,j;  
  
    piv = piv + zpiv;  
    broadcast(c1);  
  
    vin = vin + zvin;  
    broadcast(c2);  
  
}
```

# Monitor – C – pthread.h

- Mutex + podmínková proměnná = monitor

```
pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;  
pthread_cond_t cond = PTHREAD_COND_INITIALIZER;
```

```
pthread_mutex_lock(&lock);  
    pthread_cond_wait(&cond, &lock);  
    pthread_cond_signal(&cond);  
    pthread_cond_broadcast(&cond);  
pthread_mutex_unlock(&lock);
```

# Monitor – Java – synchronizovaná metoda (blok)

```
class Counter{  
    // sdílená proměnná  
    private int currentValue = 0;  
  
    public synchronized int next() {  
        // kritická sekce  
        // musí proběhnout atomicky  
        return (++currentValue);  
    }  
}
```

# Monitor – Java – java.util.concurrent

- balík java.util.concurrent.locks

```
Lock zamek = new ReentrantLock();  
Condition podminka1 = zamek.newCondition();  
Condition podminka2 = zamek.newCondition();
```

- Nad instancí třídy Condition lze pro uspání volat metody `await()`, `awaitUninterruptibly()`, `awaitNanos(long nanosTimeout)`
- Pro probuzení lze volat metody `signal()`, `signalAll()`

# Bariéra

- bariéra nastavená na N vláken
  - vlákna volají `barrier()`
  - N-1 vláken se zde zablokuje
  - když přijde N-té vlákno, všichni najednou projdou bariérou
- 
- použití: např. iterační výpočty



# Bariéra- C

```
pthread_mutex_t bariera_zamek = PTHREAD_MUTEX_INITIALIZER;  
pthread_cond_t cond_bariera = PTHREAD_COND_INITIALIZER;  
int bariera_cnt = 0;
```

```
void barrier() {  
    pthread_mutex_lock(&bariera_zamek);  
    bariera_cnt++;  
    if (bariera_cnt < N)  
        pthread_cond_wait(&cond_bariera, &bariera_zamek)  
    else {  
        bariera_cnt = 0;  
        pthread_cond_broadcast(&cond_bariera);  
    }  
    pthread_mutex_unlock(&bariera_zamek);  
}
```

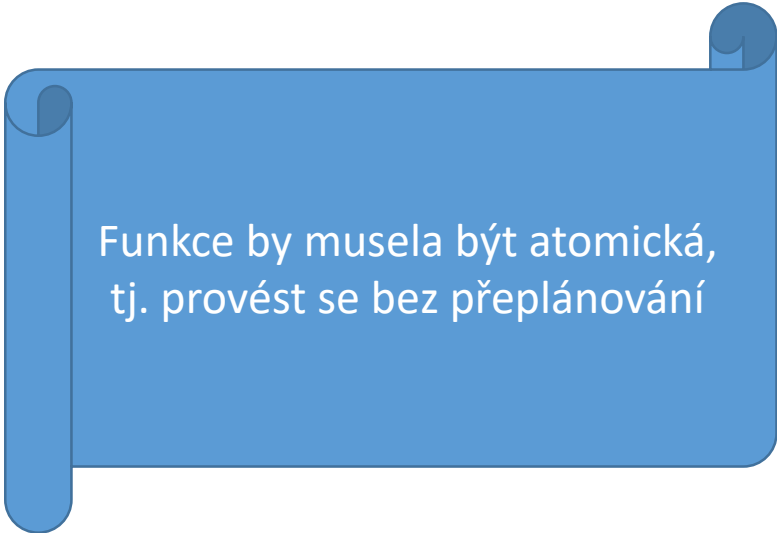
# Bariéra - Java

```
private synchronized void barrier() {  
    cnt++;  
    if ( cnt < N)  
        try { wait(); } catch (InterruptedException ex) {}  
    else {  
        cnt = 0;  
        notifyAll();  
    }  
}
```

# Instrukce TSL – obecně

- Tato nebo obdobná poskytována HW počítače: TSL R, zamek
- Proveďte atomicky dvě operace
  - Nastaví zámek na hodnotu ZAMCENO
  - Vrátí původní hodnotu zámku

```
#define ZAMCENO 1
atomic int TSL(int *zamek) {
    int oldValue;
    oldValue = zamek;
    zamek = ZAMCENO;
    return oldValue }
```

A blue callout box with rounded corners and a shadow, containing text. It has a small circular tab on the top right and a larger rectangular tab on the bottom left.

Funkce by musela být atomická,  
tj. provést se bez přepínání

# Instrukce TSL - logika

- Chceme získat zámek
- Zavoláme  
i = TSL(&zamek);
- Otestujeme:
  - i je rovno ODEMCENO (tj. 0)
    - Původní hodnota zámku byla ODEMCENO
    - Instrukce TSL zámek nastavila na ZAMCENO
    - Jelikož TSL je atomické, zámek se podařilo zamknout nám
    - Zámek je náš 😊
  - i je rovno ZAMCENO (tj. 1)
    - Zámek už byl zamknutý
    - Instrukce TSL jej sice opět nastavila na ZAMCENO
    - Ale zámek není stejně náš, musíme zkusit znovu 😞



# Instrukce TSL – implementace zámku

Je dáno:

```
#define ZAMCENO 1
```

```
#define ODEMCENO 0
```

```
int zamek;
```

```
int TSL(&zamek)                //atomickou TSL
```

Napište kód metod:

```
void TSL_spinlock()            // zamkni
```

```
void TSL_spinunlock()          // odemkni
```

(použití: *TSL\_spinlock(); KS ; TSL\_spinunlock()* )

Napište kód metod TSL\_spinlock() a TSL\_spinunlock()

# Instrukce TSL – implementace zámku

## Spin\_lock:

TSL R, lock	:: atomicky R=lock, lock=1
CMP R, 0	:: byla v lock 0?
JNE spin_lock	:: pokud ne cykluj dál
RET	:: návrat, vstup do KS

## Spin\_unlock:

LD lock, 0	:: ulož hodnotu 0 do lock
RET	

# Instrukce TSL – implementace zámku (Linux)

spin\_lock:

TSL R, lock

CMP R, 0                   ;; byla v lock 0 ?

JE cont                    ;; pokud byla, skočíme

Loop: CMP lock, 0   ;; je lock 0 ?

JNE loop                ;; pokud ne, skočíme

JMP spin\_lock           ;; pokud ano, skočíme

Cont: RET                ;; návrat, vstup do KS

# Instrukce CAS - Filozofie

- Compare and Swap
- Jiná filozofie – místo zamykání spoléháme, že nám pod rukama hodnotu proměnné nikdo nezmění
- S předpokládanou hodnotou provedeme výpočet
- Pokud nám risk vyšel a předpokládaná hodnota se nezměnila, můžeme dosadit vypočtenou hodnotu 😊
- Pokud nevyšel, musíme výpočet opakovat pro novou předpokládanou hodnotu 😞
- Není deterministické



# Instrukce CAS - Implementace

- Poskytována procesorem
- Prove atomicky test hodnoty, pokud je shoda, nastaví na novou hodnotu
- Vždy vrátí původní hodnotu

```
atomic int CAS(int *pointer,  
    int expectedValue, int newValue) {  
    int oldValue = *pointer;  
    if (oldValue == expectedValue)  
        *pointer = newValue;  
    return oldValue;  
}
```

# Instrukce CAS - použití

`int CAS (x, ocekavana, nova)`

- provede atomicky:
- vrátí původní hodnotu x
- pokud `x == ocekavana`, nastaví x na nova

návratovou hodnotou CAS je původní hodnota x

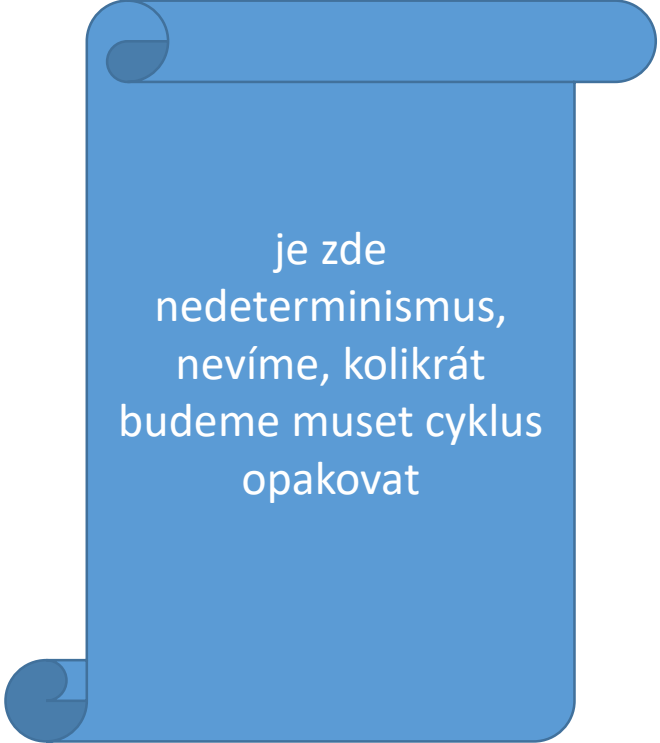
`puvodni = CAS(x,ocekavana,nova)`

- pokud `puvodni == ocekavana`
  - povedlo se, nepočítali jsme zbytečně
  - pokud ne, risk nevyšel, musíme počítat znovu

# Instrukce CAS - použití

znovu:

```
snimek = x;  
nova = nejaky_vypocet(snimek);  
puvodni = CAS(x, snimek, nova);  
if puvodni == snimek  
    printf(“”Uspěli jsme ”);  
else  
    goto znovu;
```



je zde  
nedeterminismus,  
nevíme, kolikrát  
budeme muset cyklus  
opakovat

# Rovnocennost synchronizačních primitiv

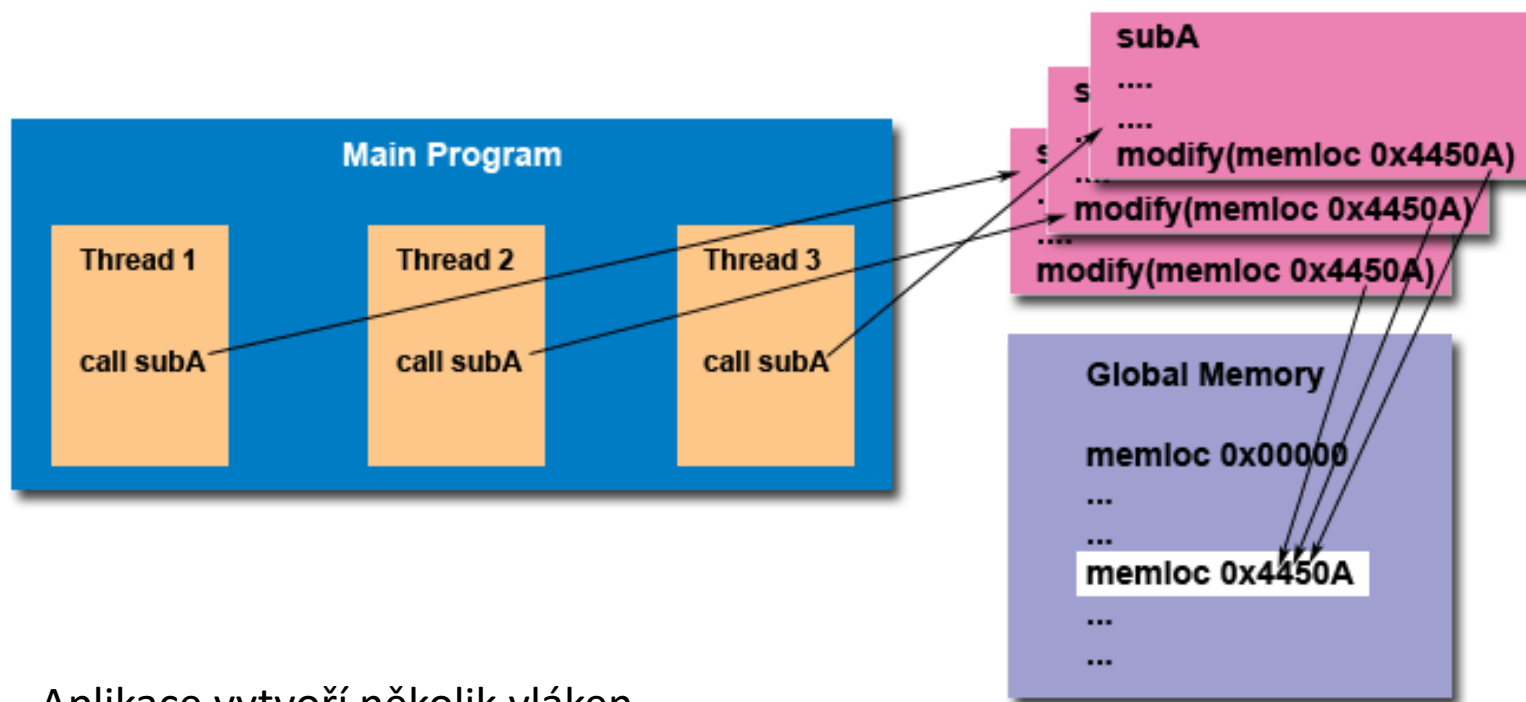
- Semafor a monitor – stejné vyjadřovací schopnosti
- Co lze udělat semaforem, lze i monitorem a naopak
- Ne všechna jsou rovnocenná – mutex s koncepcí vlastnictví neumí tolik co obecný semafor (štafetový kolík)

# Ukázka: semafor pomocí monitoru

```
monitor monsem {  
    int sem;  
    condition c1;  
  
    void P() {  
        if sem > 0  
            sem=sem-1;  
        else  
            wait(c1);  
    }  
}
```

```
void V() {  
    if not empty(c1)  
        signal(c1);  
    else  
        sem=sem+1;  
}  
  
void monIni(int param)  
    { sem = param; }  
  
init {sem = 1;}  
}
```

# Vláknová bezpečnost (thread-safe)



Aplikace vytvoří několik vláken  
Každé vlákno vyvolá stejnou rutinu  
Tato rutina modifikuje společná globální data –  
pokud nemá synchronizační mechanismy, není  
thread-safe

# Vyhladovění (starvation)

Definice:

- Procesu je opakovaně odmítnut přístup ke zdrojům, které pro svoji práci potřebuje.

Příklady:

Např. přístup k CPU díky nevhodnému plánovacímu algoritmu.

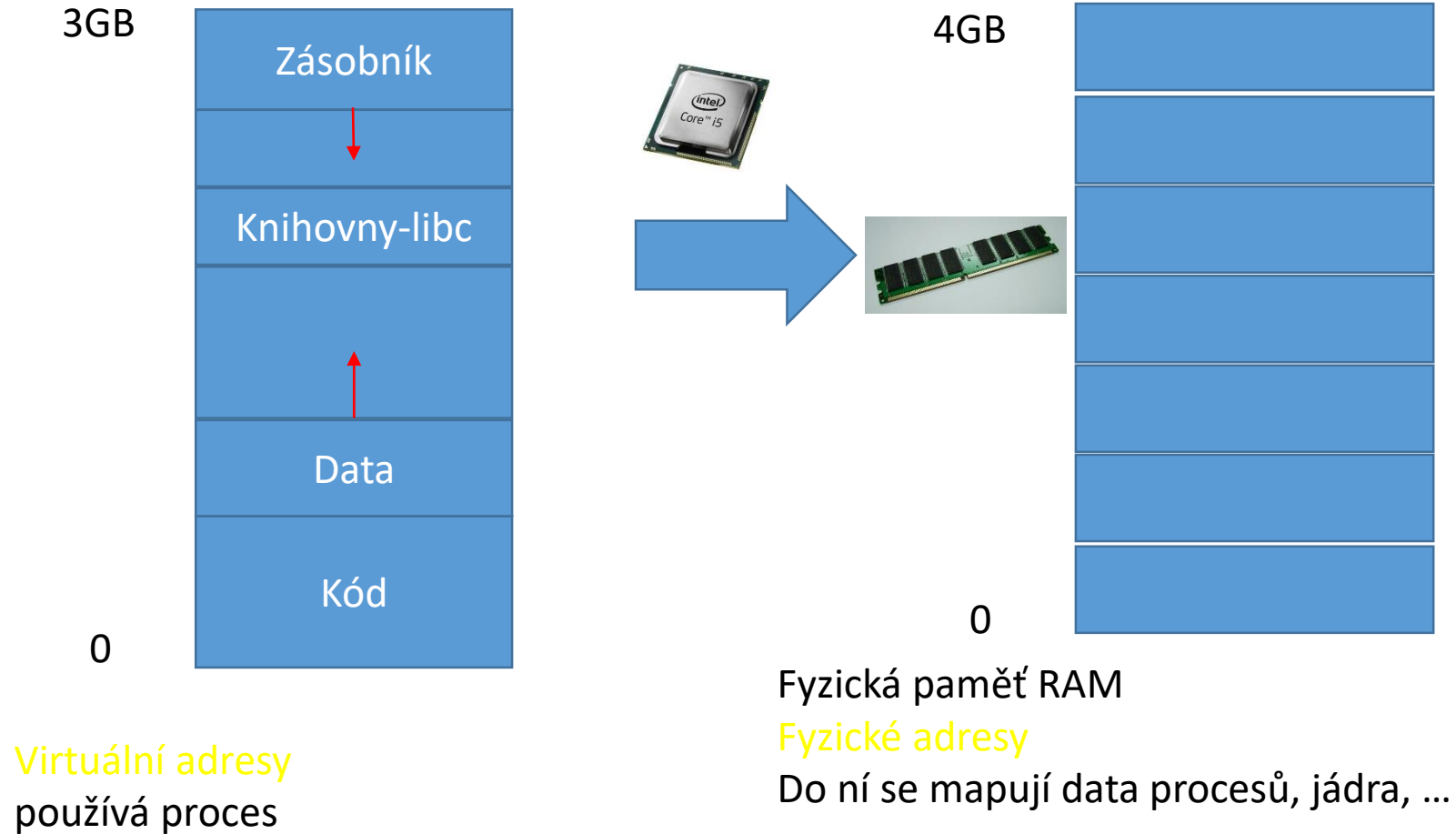
Nebo při vzájemném vyloučení – např. přístup k vidličkám u večeřících filozofů.

A blue horizontal scroll graphic with rounded ends and a vertical strip on the left side, resembling a rolled-up document.

# Správa paměti



# Adresní prostor procesu



# Paměť RAM

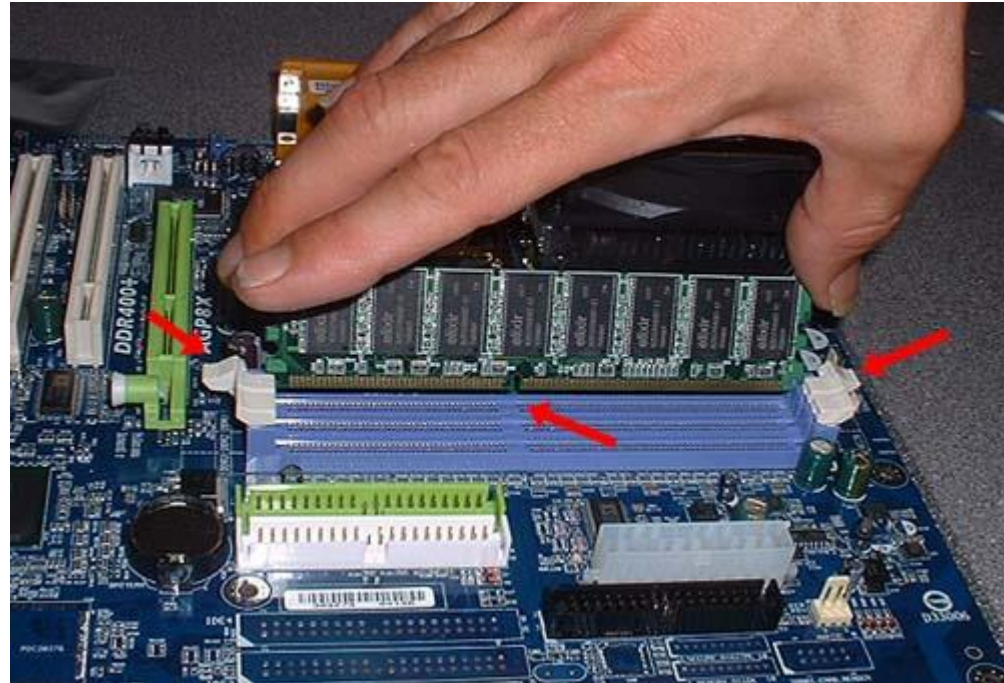
Fyzická operační paměť RAM

Při vypnutí napájení ztratí svůj obsah

Tvořena paměťovými chipy

Typická velikost RAM v dnešních PC a NB je:

- 4 GB
- 8 GB
- 16 GB



Zdroj obrázku: <http://www.custom-build-computers.com/Fitting-PC-Ram.html>

# Paměť – příklad alokace

```
ptr = malloc (100);
```

Virtuální adresa  
začátku alokované  
oblasti paměti

Chceme přidělit 100B paměti

Na haldě se alokuje 100B

Na začátek alokované oblasti odkazuje virtuální adresa ptr (nějaké číslo)

Oblast je mapována do fyzické paměti (RAM) od nějaké jiné fyzické adresy

# Registry (příklad architektura x86)

- malé úložiště dat uvnitř procesoru
- **obecné registry**
  - **EAX, EBX, ECX, EDX** .. jako 32ti bitové
  - **AX, BX, CX, DX** .. využití jako 16ti bitové (dolních 16)
  - **AL, AH** .. využití jako 8bitové
- **obecné registry - uložení offsetu**
  - **SP** .. offset adresy vrcholu **zásobníku (!)**
  - **BP** .. pro práci se zásobníkem
  - **SI** .. offset zdroje (source index)
  - **DI** .. offset cíle (destination index)

# Registry

- segmentové registry
  - CS code segment (kód)
  - DS data segment (data)
  - ES extra segment
  - FS volně k dispozici
  - GS volně k dispozici
  - SSstag segment (zásobník)

# Registry

- speciální
  - **IP** .. offset vykonávané instrukce (CS:IP)
  - **FLAGS** .. zajímavé jsou jednotlivé bity
    - **IF** .. interrupt flag (přerušeno-zakázáno-povoleno)
    - **ZF** .. zero flag (je-li výsledek operace 0)
    - **OF, DF, TF, SF, AF, PF, CF**

bližší info např. [http://cs.wikipedia.org/wiki/Registr\\_procesoru](http://cs.wikipedia.org/wiki/Registr_procesoru)  
jde nám o představu jaké registry a k jakému účelu jsou

# Registry (x86-64)



64bit

- For 16-bit operations, the two bytes of Register A are addressed as AX
- For 32-bit operations, the four bytes of Register A are addressed as EAX
- For 64-bit operations, the eight bytes of Register A are addressed as RAX

zdroj:

[http://pctuning.tyden.cz/index2.php?option=com\\_content&task=view&id=7475&Itemid=28&pop=1&page=0](http://pctuning.tyden.cz/index2.php?option=com_content&task=view&id=7475&Itemid=28&pop=1&page=0)

# Rozdělení paměti pro proces (!!!)

pokus.c:

```
int x =5; int y = 7; // inic. data
```

```
void fce1() {  
    int pom1, pom2; // na zásobníku  
    ... }
```

```
int main (void) {  
    ...  
    malloc(1000); // halda  
    fce1();  
    return 0;  
}
```

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



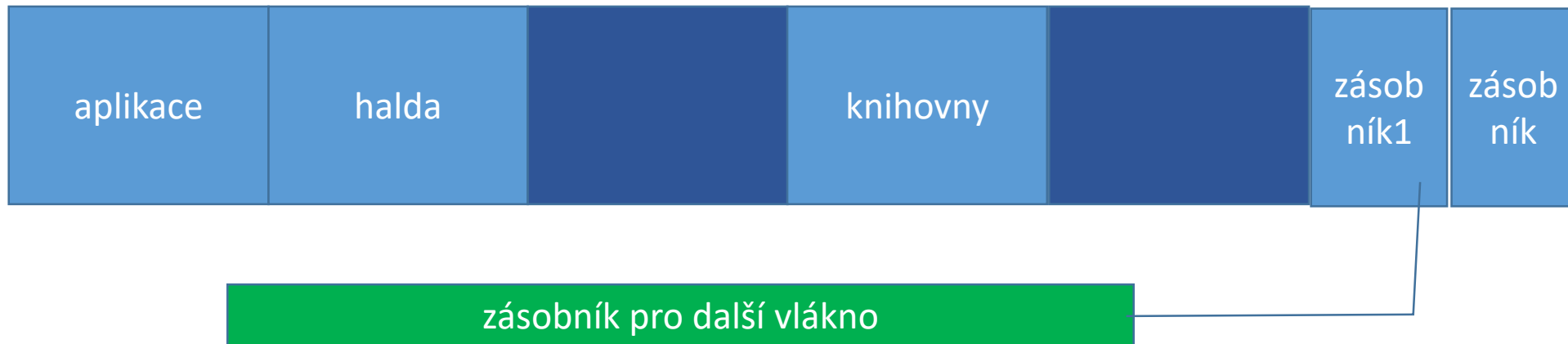
3+1: 3GB proces, 1GB OS



# Rozdělení paměti pro proces

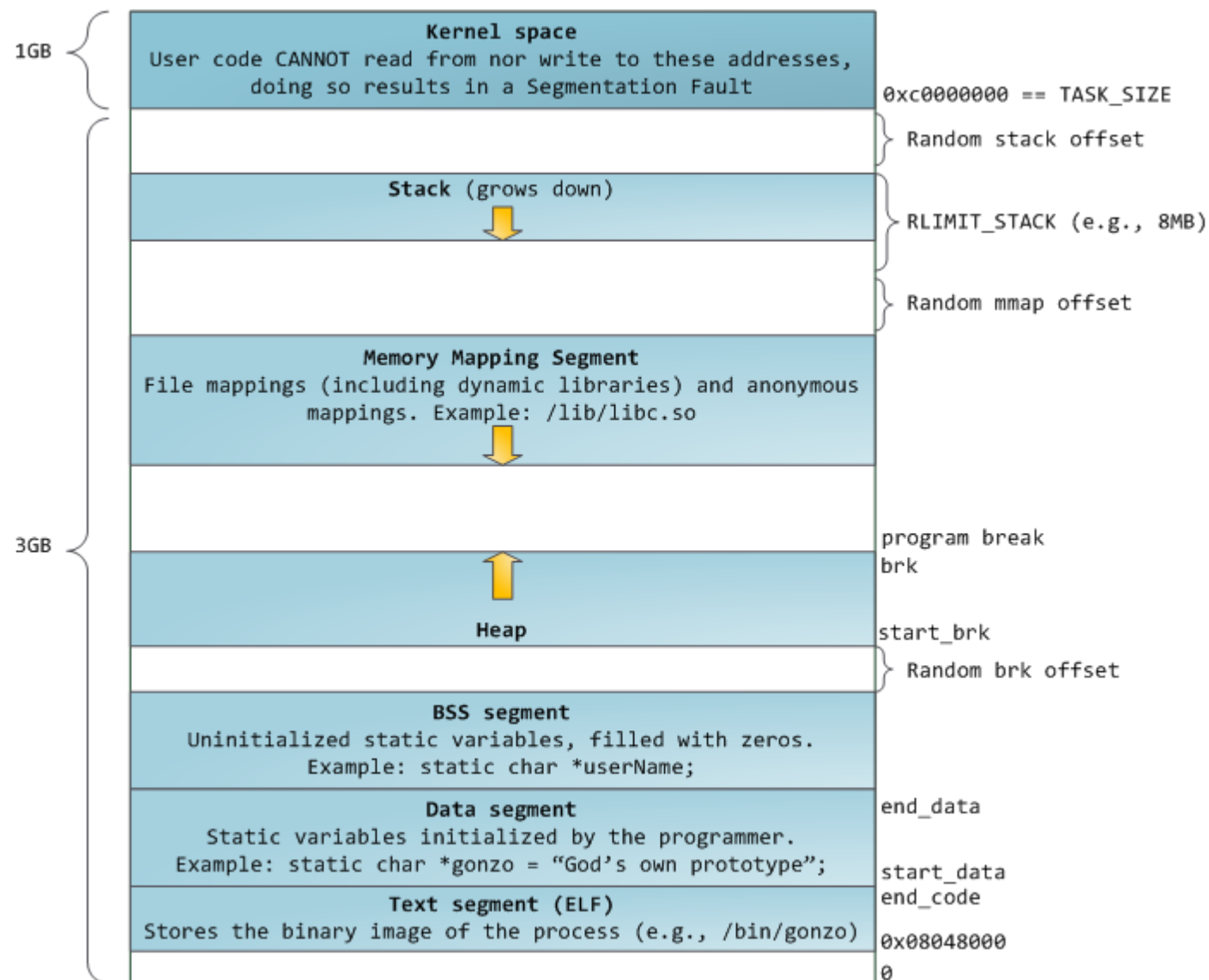


Máme-li více vláken => více zásobníků, limit velikosti zásobníku



Další ukázka  
rozdělení  
paměti

zapamatovat  
pojem BSS

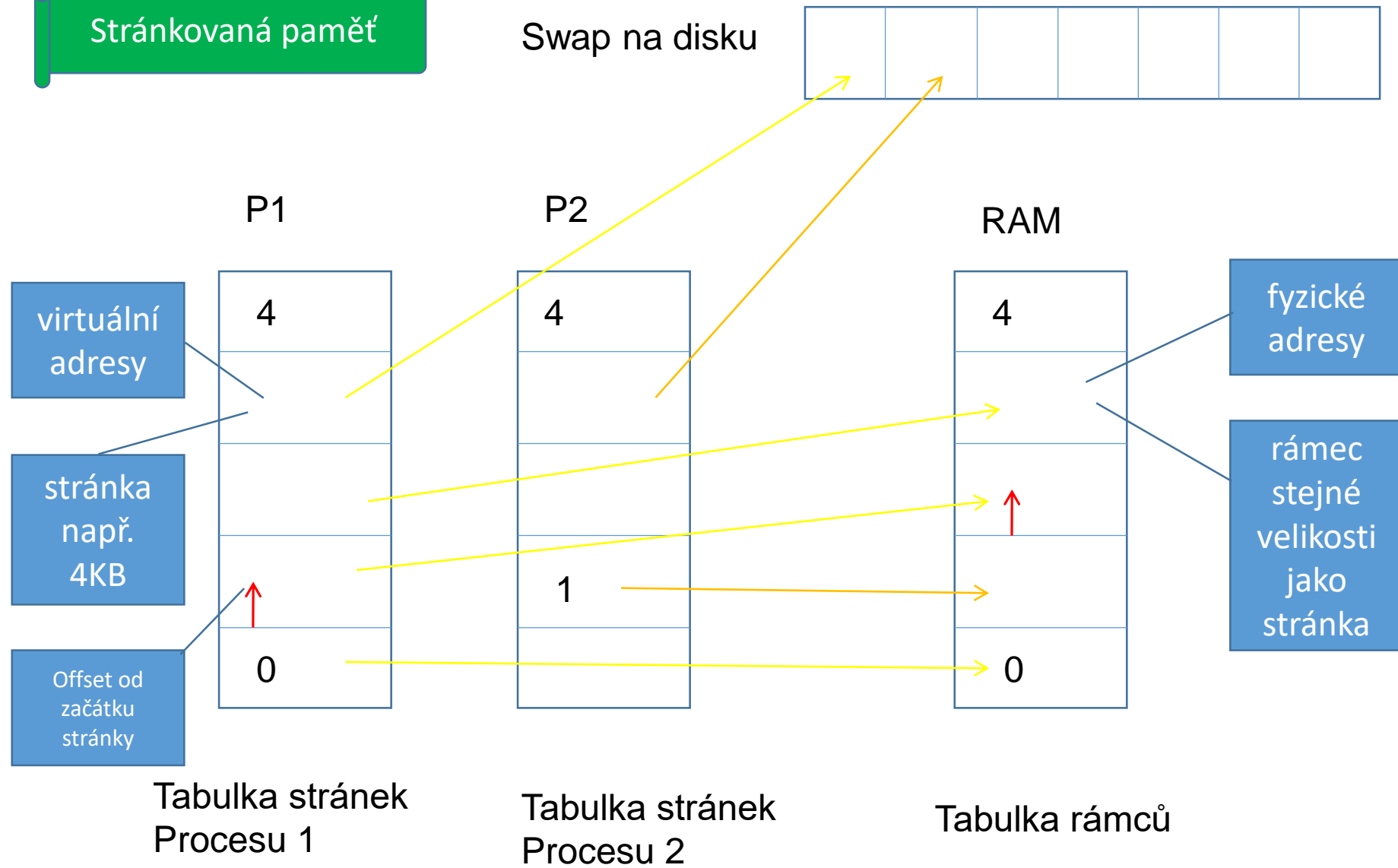


zdroj: <http://duartes.org/gustavo/blog/category/linux> DOPORUČUJI !!

# Stránkovaná paměť

- Tabulka stránek pro každý proces (leží v RAM)
- Pro zrychlení převodu TLB cache
- Fyzická paměť RAM – rozdělena na rámce
- Adresní prostor procesu – rozděleno na stránky
- Typická velikost stránky: 4 KB, 2MB, 4MB
- Stránka leží buď v RAM, nebo na disku typicky ve swapu

## Stránkovaná paměť



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	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

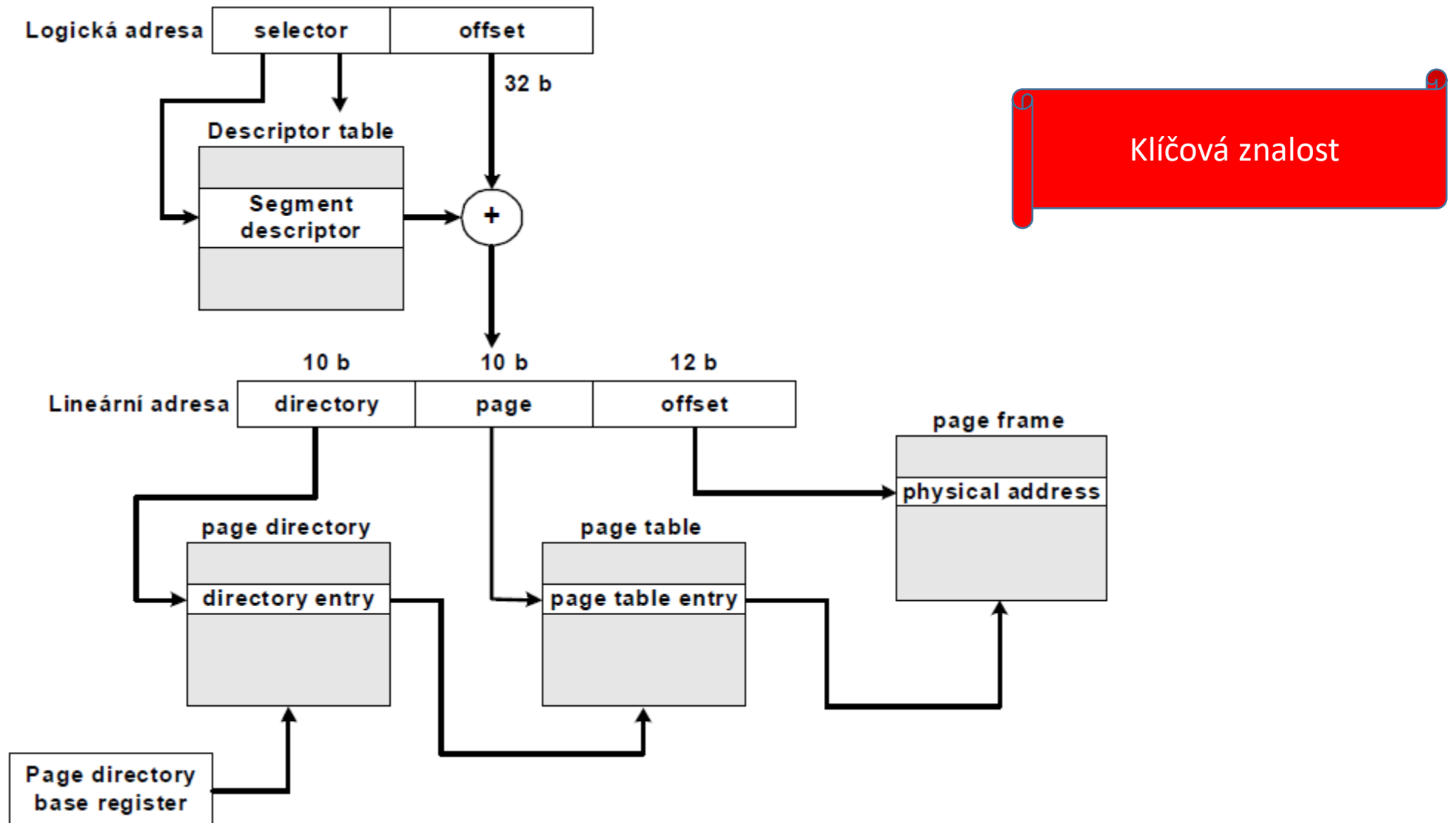
valid  
invalid

rw, rx, ro,...

zda je třeba rámec uložit  
do swapu při odstranění z  
RAM

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

# Komplexní schéma převodu VA na FA (Segmentace se stránkováním - pamatovat !!!!)



# Adresy (!!!)

virtuální adresa -> lineární adresa -> fyzická adresa

virtuální – používá proces (sektor:offset)

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

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



# Dnešní Intel procesory

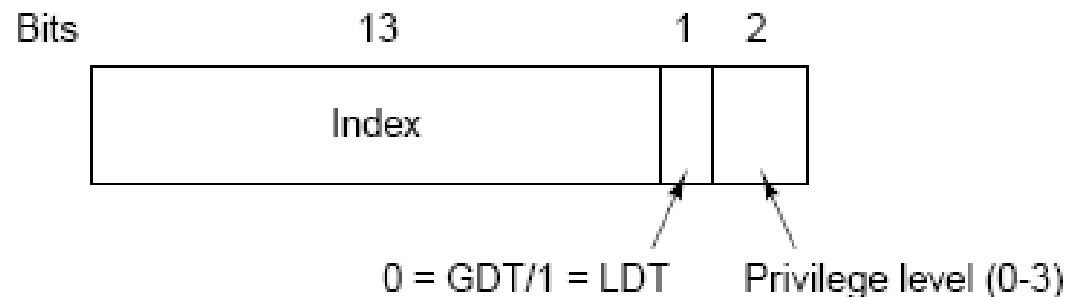
- segmentace
- stránkování
- segmentace se stránkováním
- **tabulka LDT (Local Descriptor Table)**
  - Je jich více
  - segmenty lokální pro proces (kód,data,zásobník)
- **tabulka GDT (Global Descriptor Table)**
  - pouze jedna, sdílená všemi procesy
  - systémové segmenty, včetně OS

# Segmentové registry

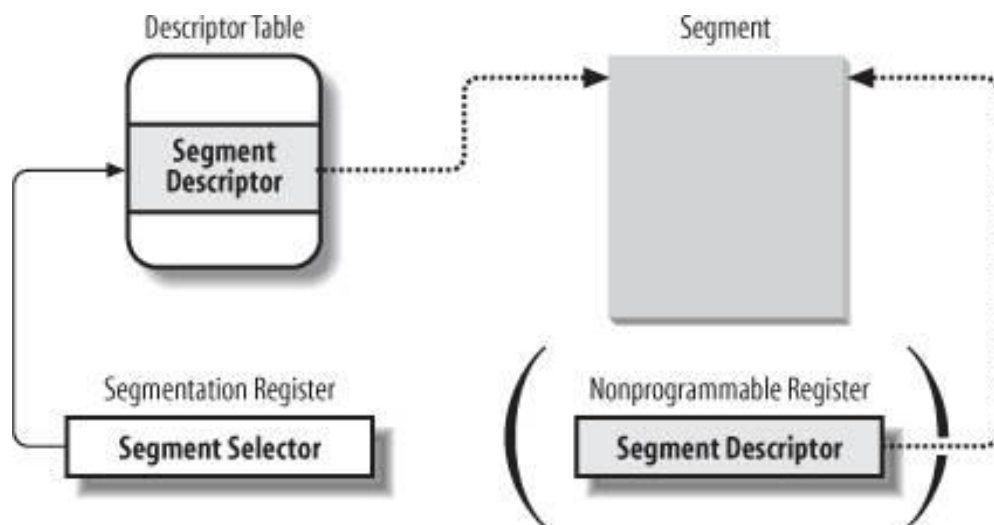
- Pentium a výše má 6 segmentových registrů:
  - CS (Code Segment)
  - DS (Data Segment)
  - SS (Stack Segment)
  - další: ES, FS, GS
- přístup do segmentu → do segmentového registru se zavede selektor segmentu

# Selektor segmentu (!!)

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



# Rychlý přístup k deskriptoru segmentu



logická adresa:

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

zrychlení převodu:

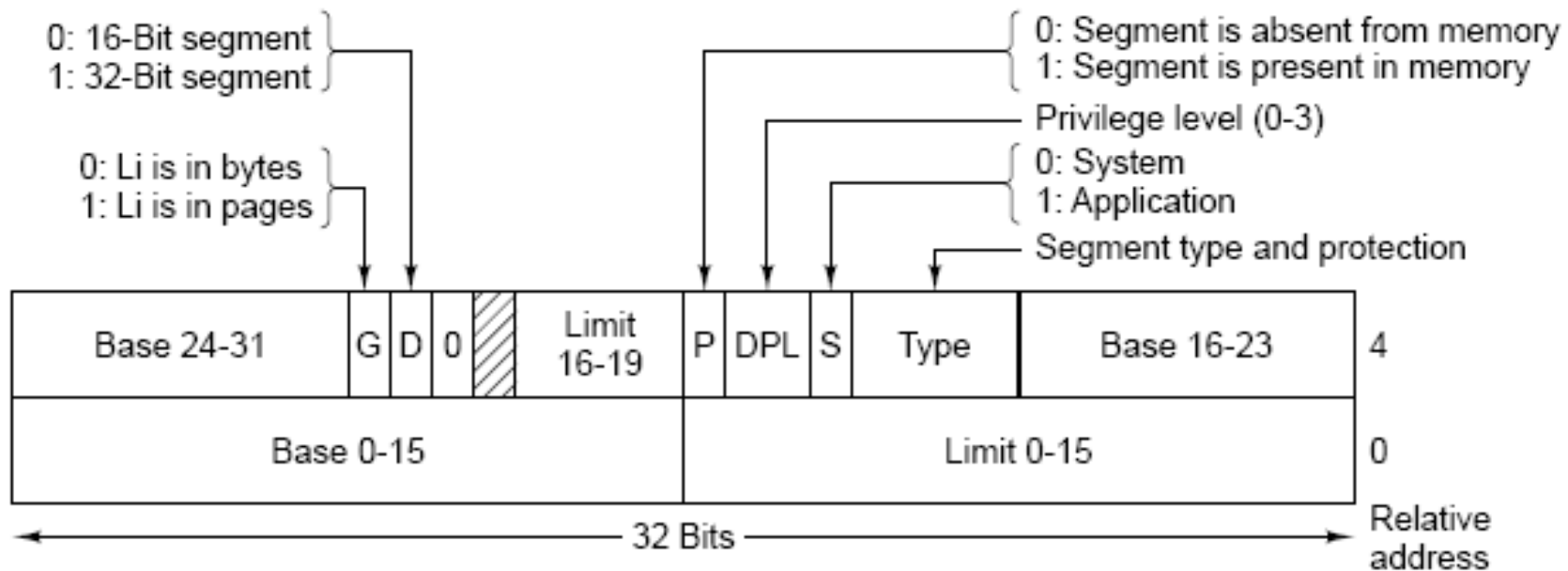
přídavné neprogramovatelné registry (pro každý segm. reg.)

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

# Deskriptor segmentu (!!)

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

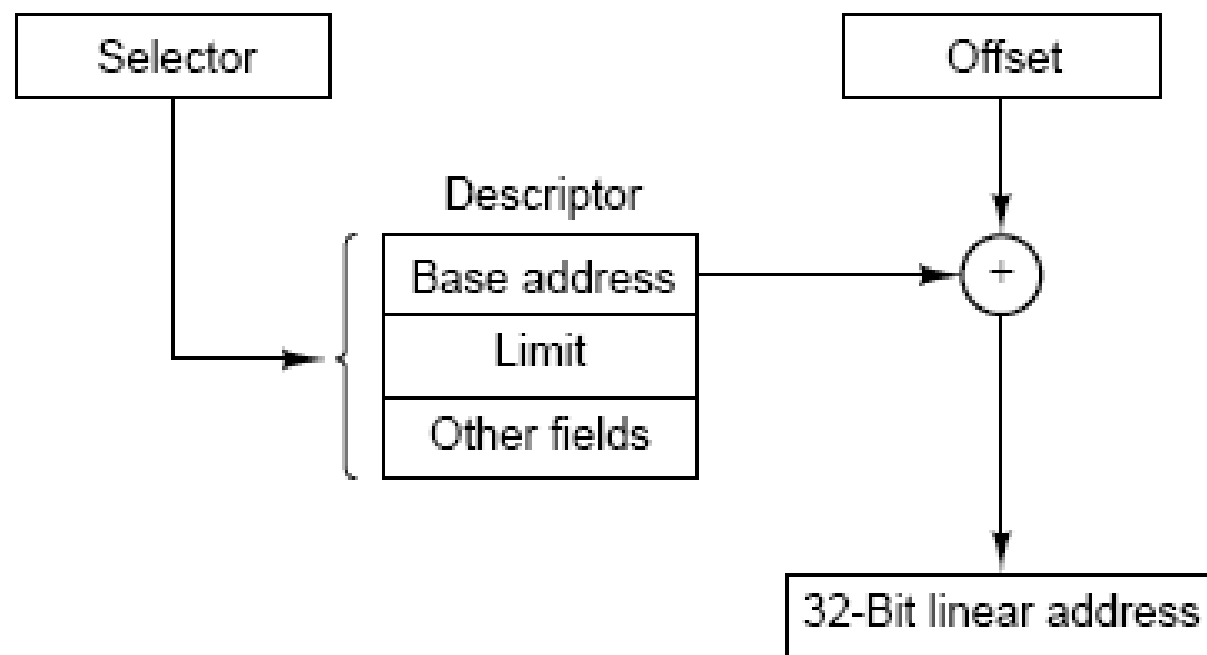
# Deskriptor segmentu (8 bytů)



# Konverze na fyzickou adresu

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

## Konverze na fyzickou adresu I.





# Implicitní nebo explicitní určení segmentu

## Příklad v assembleru:

- Implicitní

- `jmp $8052905` -- implicitně použije **CS**  
(instrukce skoku)
- `mov $8052905, %eax` -- implicitně použije **DS**  
(instrukce manipulace s daty)

- Explicitní

- `mov %ss:$8052905, %eax`-- explicitně použije **SS**

# Poznámky

Jakou strategii moderní systémy využívají?

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

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

4 segmenty:  
Kernel Code  
Kernel Data  
User Code  
User Data

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

# Linux segmentation

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

# Pracovní množina stránek

- Stránky, ke kterým proces přistupuje v nějakém časovém intervalu
- Pojem PFF
- Detekce zahlacení pomocí PFF, uživatel si všimne 😊

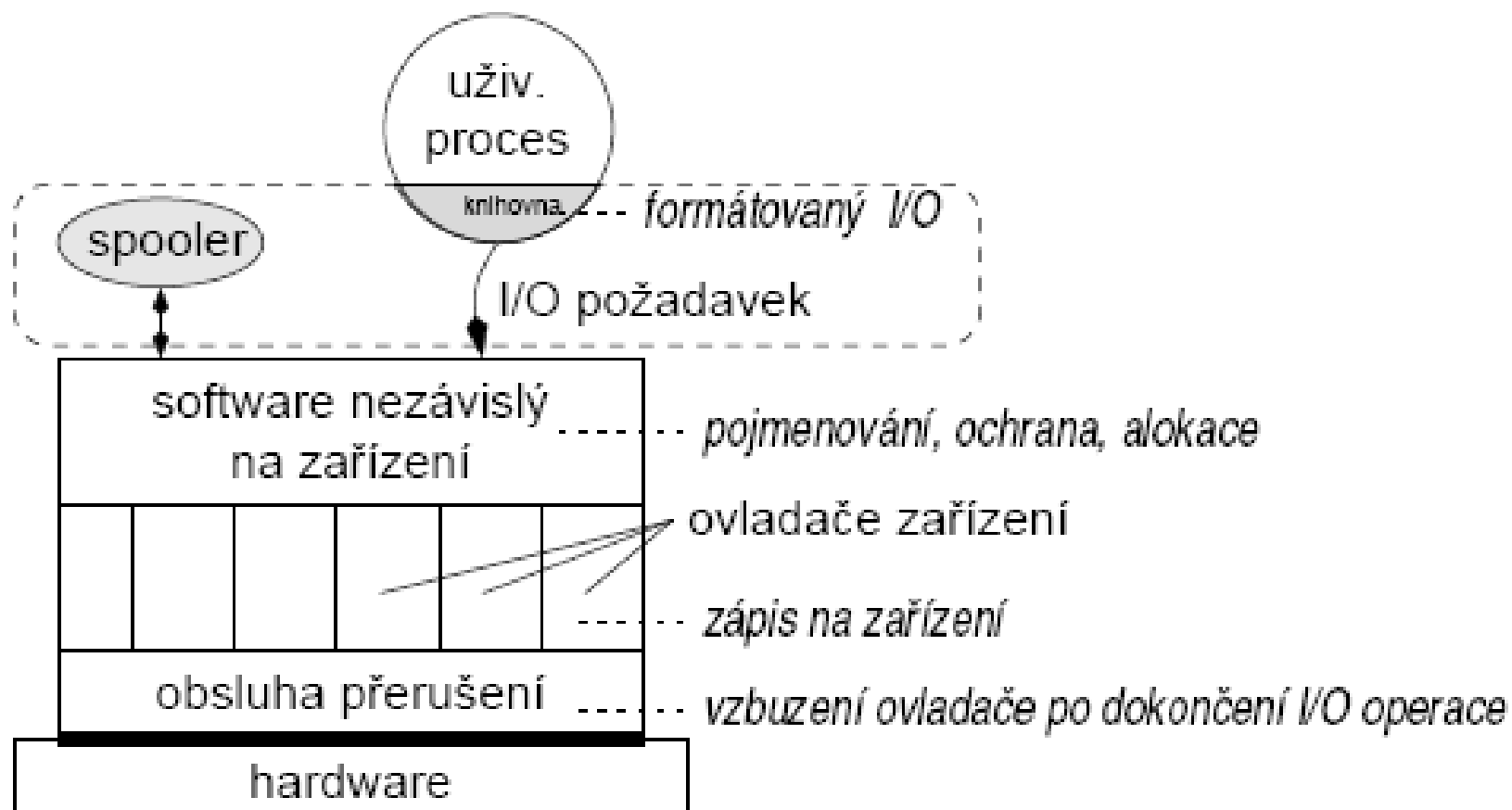
A blue horizontal scroll graphic with a vertical bar on the left side and decorative scroll ends at the top and bottom.

Správa I/O

# Principy I/O software

typicky strukturován do 4 úrovní:

1. obsluha přerušení (nejnižší úroveň v OS)
2. ovladač zařízení
3. SW vrstva OS nezávislá na zařízení
4. uživatelský I/O SW



# 1. Obsluha přerušení

- Ovladač zadá I/O požadavek a „usne“
  - Lze využít semafor s počáteční hodnotou nula
  - Zavolá P(sem)
- Řadič vyvolá přerušení ve chvíli dokončení I/O požadavku
  - Obsluha přerušení vzbudí ovladač V(sem)
  - Obsluha přerušení musí být co nejkratší
  - Ovladač může dále zpracovávat data ze zařízení



## 2. Ovladač zařízení

- Veškerý kód závislý na konkrétním I/O zařízení
- Např. ovladač zvukové karty od daného výrobce
- Ovladač jako jediný zná HW podrobnosti daného zařízení
- Ovladače mohou být pro celou třídu zařízení, např. ovladač SCSI disků nebo jen pro jedno konkrétní zařízení
- Chyby v ovladači mohou ohrozit stabilitu systému
- Může být problém sehnat ovladač pro určitý OS

### 3. SW vrstva OS nezávislá na zařízení

- I/O fce společné pro všechna zařízení daného druhu
- Pojmenování zařízení (LPT1, /dev/lp0)
- Ochrana zařízení (přístupová práva)
- Alokace a uvolnění vyhrazených zařízení  
( některá mohou být v jednu chvíli jen pro jeden proces)
- Vyrovnávací paměti  
(bloky pevné délky, využití bufferu)

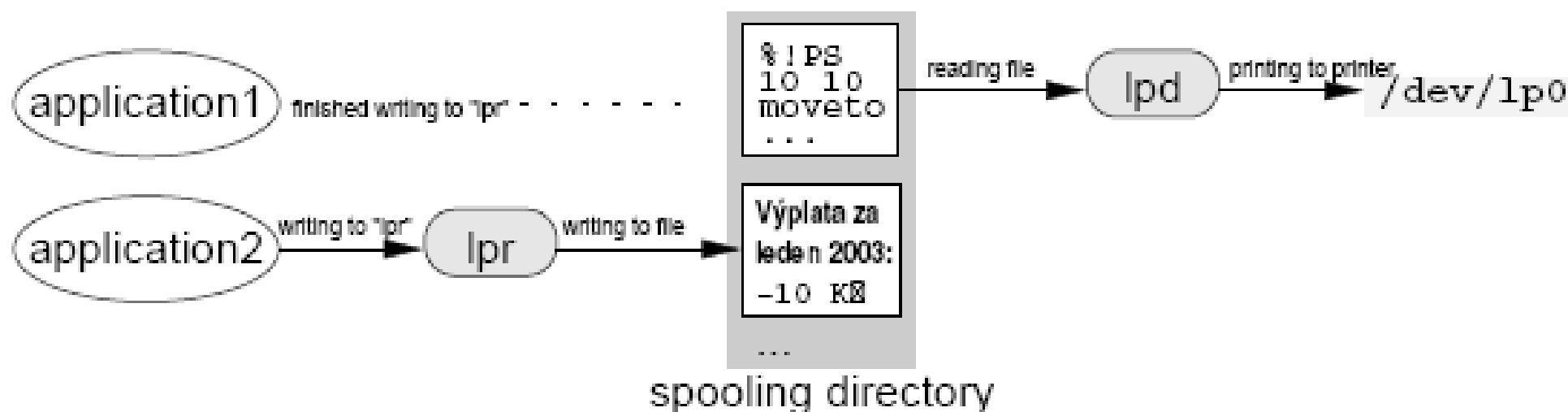
## 4. I/O SW v uživatelském režimu

- Knihovny sestavené s programem
  - Např. formátování: `printf("%.2d:%.2d\n", hodin, minut)`
- Spooling
  - Implementován pomocí procesů v uživatelském režimu
  - Způsob obsluhy vyhrazených I/O zařízení
  - Brání např. tomu, aby proces alokoval zařízení a pak hodinu nic nedělal

# Příklad spoolingu – tisk v Unixu

- k fyzické tiskárně má přístup – pouze 1 speciální proces
  - daemon **lpd**
  - Tiskne na tiskárně
- proces „tiskne“ do souboru na disku
  - proces chce tisknout, spustí **lpr** a naváže s ním komunikaci
  - proces předává tisknutá data programu lpr
  - lpr zapíše data do souboru v určeném adresáři
    - spooling directory – přístup jen lpr a lpd
  - dokončení zápisu – lpr oznámí lpd, že soubor je připraven k vytisknutí, lpd soubor vytiskne a zruší

# Příklad spoolingu - pokračování



**lpd** – démon (služba) čte ze spoolovacího adresáře a přistupuje k tiskárně

**lpr** – data, která chce aplikace vytisknout se zapisují do spoolovacího adresáře

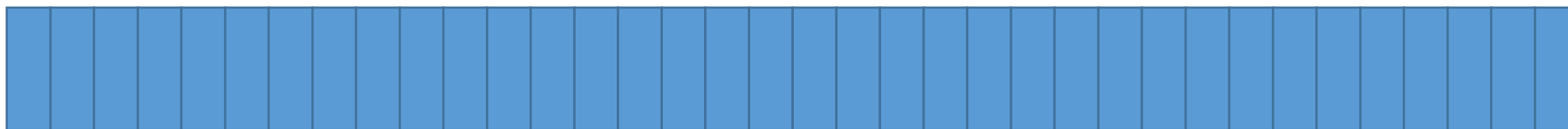
A blue horizontal scroll graphic with a vertical strip on the left side, resembling a rolled-up document. The text is centered on the scroll.

# Správa souborů

# Motivace

- Představte si velké množství volných přihrádek (datových bloků)
- Potřebujeme vymyslet vhodný systém, jak do těchto přihrádek uložit data jednotlivých souborů.
- Systém si musí pamatovat, které přihrádky patří v určitém pořadí kterému souboru a které jsou volné
- Systém musí být tak důmyslný, aby bylo možné soubory zvětšovat, aniž by bylo potřeba hýbat s již obsazenýma přihrádkama.

# Motivace



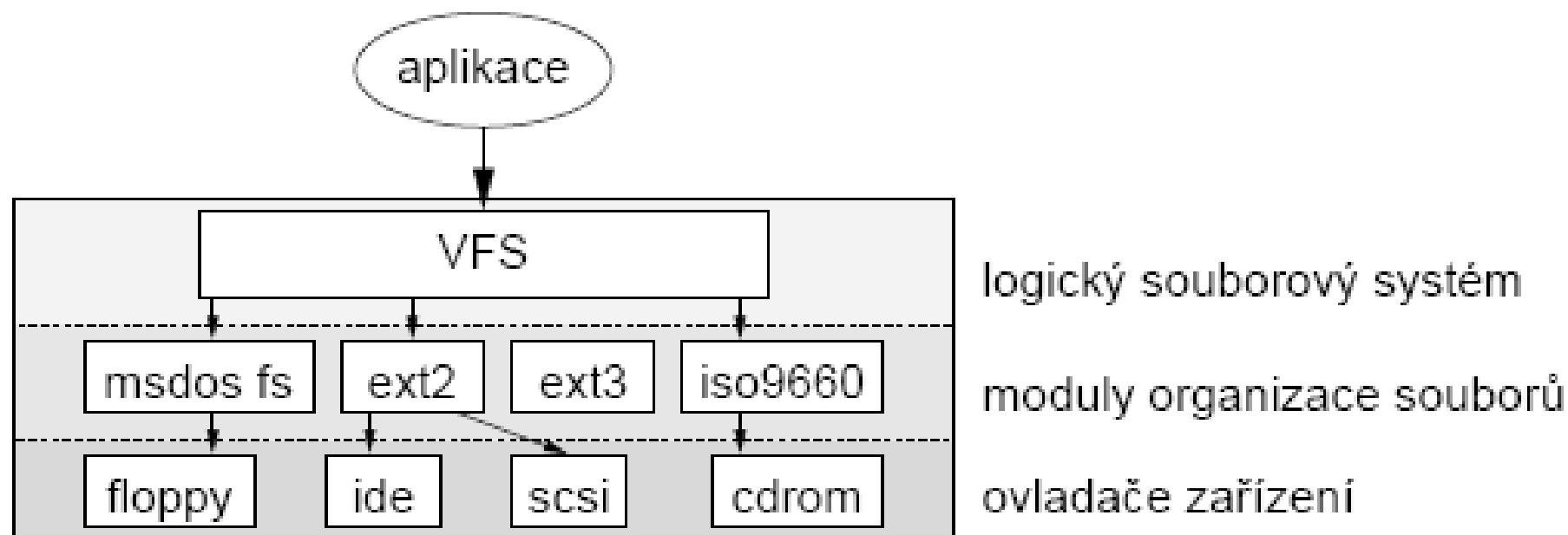
- Které bloky jsou volné?
- Které bloky patří našemu souboru, a v jakém pořadí?
- Jak zajistím prodloužení našeho souboru, aniž bych musel manipulovat s ostatními obsazenými bloky?
- Všechny tyto otázky řeší souborový systém, každý jiným způsobem – FAT, NTFS, ext4, xfs apod.



# Motivace - VFS

- Uživatelé nezajímá, jaký konkrétní systém souborů se používá.
- Uživatel chce např. otevřít pro čtení a zápis soubor ahoj.txt
- Uživateli je jedno:
  - V jakém konkrétním souborovém systému je ahoj.txt uloženo.
  - Zda jsou data souboru na rotačním disku, SSD disku, nebo leží někde na serveru.
- Proto je nad konkrétním FS ještě virtuální VFS, který požadavek uživatele předá konkrétnímu filesystému.

# Implementace souborových systémů (!)



# Implementace fs - vrstvy

## 1. Logický (virtuální) souborový systém

- Volán aplikacemi
- Kód společný pro všechny fs
- Předá požadavek konkrétnímu fs

## 2. Modul organizace souborů

- Konkrétní souborový systém (např. ext4, ntfs, fat, xfs)
- Ví, který blok je volný a který patří konkrétnímu souboru
- Správa volného prostoru a alokace bloků

## 3. Ovladače zařízení

- Pracuje s daným zařízením (disketa, cdrom, sata disk, síťový redirektor)
- Přečte/zapíše logický blok

# Jak VFS pracuje s konkrétním fs?

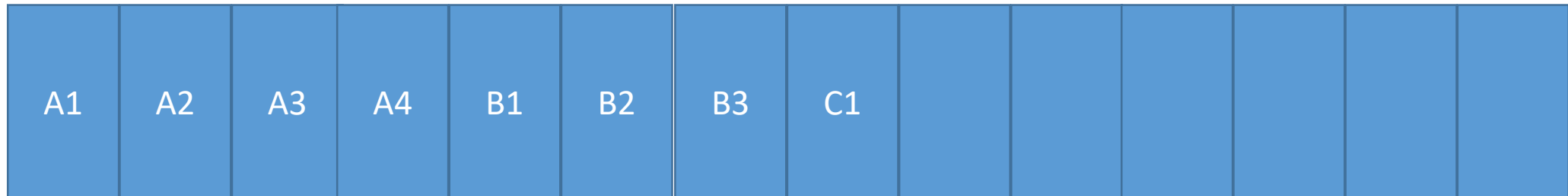
- Nový filesystem, který chceme používat se nejprve v systému **zaregistruje**
- Díky registraci VFS ví, jak zavolat jeho metody open, read, write pro konkrétní soubor
- Při požadavku na soubor VFS napřed zjistí, na kterém filesystemu leží:
  - Viz např. příkaz mount
  - /bin/ls může ležet na ext4, /home/pesi/f1.txt na xfs
  - Pro čtení /bin/ls zavolá **VFS->ext4->read**
  - Pro čtení /home/pesi/f1.txt zavolá **VFS->xfs->read**

Co je na tom skvělého?

Když vymyslíte vlastní filesystem a zaregistrujete jej u VFS, operační systém s ním bude moci pracovat 😊

# Kontinuální alokace

- Výhoda
  - Velmi rychlé čtení, najdeme snadno požadovaný blok
- Nevýhoda
  - Problém zvětšení souborů
- Kdy je vhodné
  - Write once média – na CD/DVD zapíšeme a pak už jen čteme



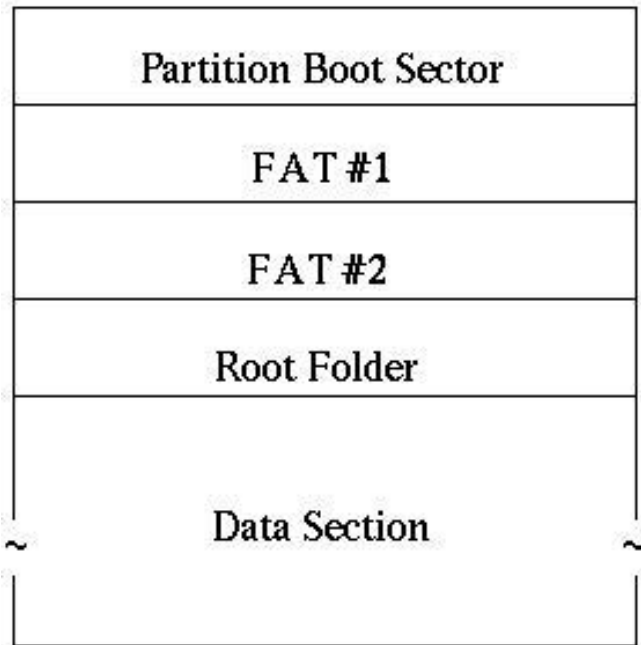
# Co musí obsahovat adresář při kontinuální alokaci?

- Název souboru
- Počáteční číslo bloku
- Velikost
  - Zjistíme, kolik bloků soubor zabírá
  - Víme, jak velká část posledního bloku je využita
- Příklad:  
a1.txt, 200, 9242  
(soubor a1.txt začíná na bloku 200 a má velikost 9242 bytů)

# FAT

- Jeden ze základních souborových systémů
- Většina OS s ním umí pracovat
- Vhodný např. pro menší paměťové karty
- Nemá přístupová práva
- Jen atributy (systém, archive, hidden apod.)

# Diskový oddíl s FAT



Diskový oddíl, který je naformátován na FAT (např. FAT32)

1. boot sektor (pokud se z daného oddílu bootuje)
2. kopie FAT tabulky, typicky jsou dvě
3. hlavní adresář daného oddílu
4. data

Zdroj obrázku:

<http://yliqepyh.keep.pl/fat-file-system-specifications.php>



## Tabulka FAT

Položka č.: 0	5	
1	2	
2	3	
3	4	
4	9	
5	6	
6	7	
7	8	
8	13	
9	10	
10	-1	
11	volný	
12	volný	
13	-1	

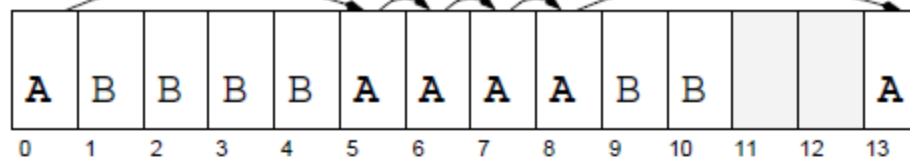
soubor A začíná zde  
soubor B začíná zde

Fakt tu je !  
Jdem dál na  
6

značka konce souboru  
(EOF marker)

5 znamená, že na indexu 5 je  
další odkaz na blok souboru A

Na indexu 5 je i datový blok  
souboru A (kus filmu)

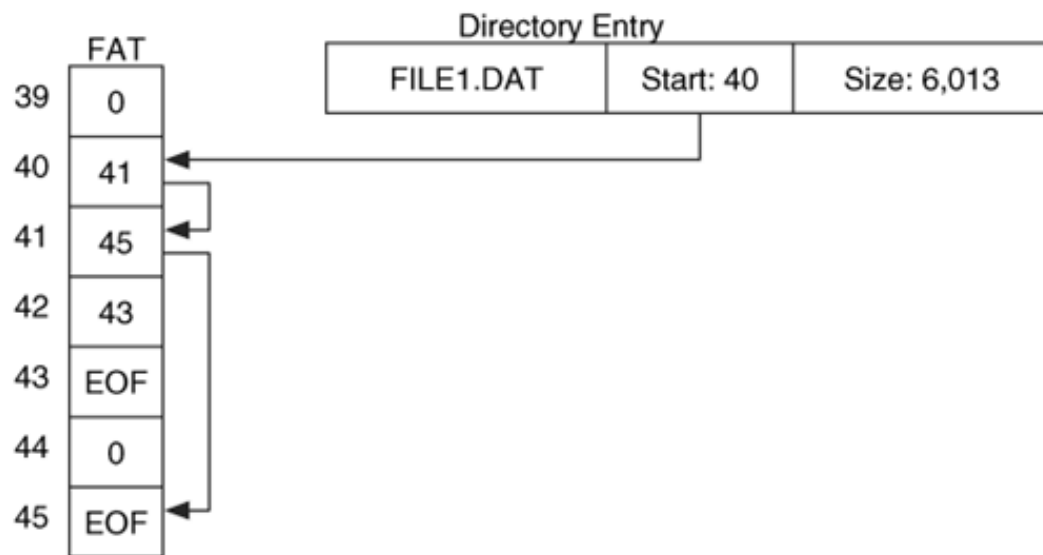


datové bloky na disku

Co je na FAT důležité?

Pokud bych chtěl např. k souboru B přidat další datový blok,  
nemusím s ničím hýbat, pouze do FAT(10) vložím číslo 11, a do  
FAT(11) dám -1 a soubor B je prodloužený

# FAT – jak poznáme, kde soubor začíná?



V tomto obrázku by velikost bloku byla cca 2KB – 2048B  
Značka EOF je číslo symbolizující konec souboru.

Z adresáře poznáme:

- Jméno souboru
- Kde začíná (40)
- Velikost (6013 bytů)

Víme tedy, kde začíná soubor (40) a umíme určit, jaká část posledního přiděleného bloku je využita daty souboru (zbytek po dělení 6013 velikostí bloku)

# Co musí obsahovat adresář při použití FAT?

- Název souboru
- Počáteční číslo bloku
- Velikost
  - Zjistíme, kolik bloků soubor zabírá
  - Víme, jak velká část posledního bloku je využita
- Příklad:  
f1.txt, 800, 12342  
soubor f1.txt začíná na bloku 800 a má velikost 12342 bytů  
při velikosti bloku 1024 bytů zabírá soubor celkem: 13 bloků  
poslední blok není zcela zaplněný

# Různé druhy FAT

- FAT12, 12 bitů,  $2^{12} = 4096$  bloků, diskety
- FAT16, 16 bitů,  $2^{16} = 65536$  bloků
- FAT32,  $2^{28}$  bloků, blok 4-32KB, cca 8TB
- ExFAT – používá B+ strom místo tabulky

# NTFS

- nativní fs Windows od NT výše
- **žurnálování**  
všechny zápisy na disk se zapisují do žurnálu, pokud uprostřed zápisu systém havaruje, je možné dle stavu žurnálu zápis dokončit nebo anulovat => konzistentní stav
- **access control list**  
přidělování práv k souborům ( x FAT)
- **komprese**  
na úrovni fs lze soubor nastavit jako komprimovaný

# NTFS pokračování

- **šifrování**  
EFS (encrypting file system),  
transparentní – otevřu ahoj.txt, nestarám se, zda je  
šifrovaný
- **diskové kvóty**  
max. velikost pro uživatele na daném oddíle  
dle reálné velikosti (ne komprimované)
- **pevné a symbolické linky**

# NTFS struktura (!!)

- 64bitové adresy clusterů .. cca 16EB
- clustery **číslovány od začátku partition** – logická čísla clusterů
- systém jako obří databáze  
záznam v ní odpovídá souboru
- základ 11 systémových souborů - metadata  
hned po formátování svazku
- **\$Logfile** – žurnálování
- **\$MFT** (Master File Table) – nejdůležitější (!!)  
záznamy o všech souborech, adresářích, metadatach  
hned za boot sektorem, za ním se udržuje zóna volného místa

# NTFS struktura

- **\$MFTMirr** – uprostřed disku, obsahuje část záznamů \$MFT, při poškození se použije tato kopie
- **\$Badclus** – seznam vadných clusterů
- **\$Bitmap** – sledování volného místa  
0 – volný
- **\$Boot, \$Volume, \$AttrDef, \$Quota, \$Upcase, .**

podrobnosti:

<http://technet.microsoft.com/en-us/library/cc781134%28WS.10%29.aspx>



# NTFS – způsob uložení dat (!!!)

- **kódování délkou běhu**
- od pozice 0 máme např. uloženo:  
A1, A2, A3, B1, B2, A4, A5, C1, ...
- soubor A bude popsán fragmenty
- **fragment**
  - index
  - počet bloků daného fragmentu
- v našem příkladě pro soubor A dva fragmenty:
  - 0, 3 (od indexu 0 patří tři bloky souboru A)
  - 5, 2 (od indexu 5 patří dva bloky souboru A)

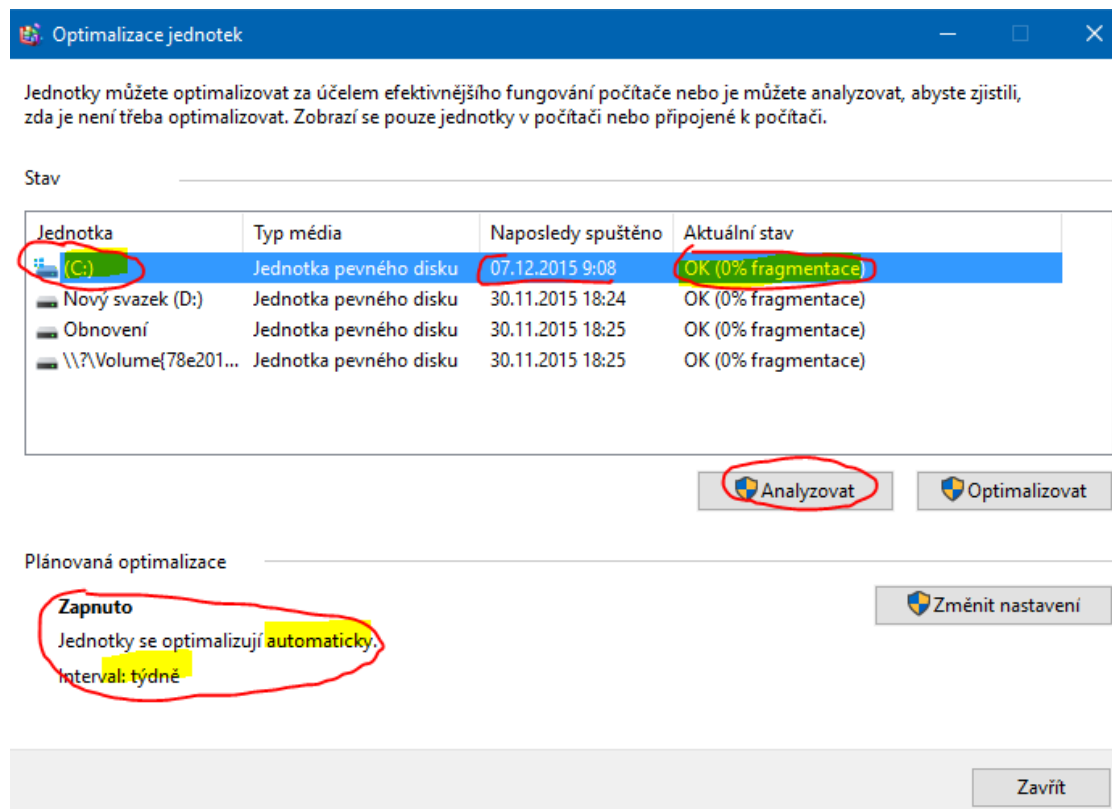
# NTFS – způsob uložení dat

- V ideálním případě 1 soubor = 1 fragment (výhody kontinuální alokace)
- Defragmentovat můžeme jak celou partition, tak jen vybrané soubory (přes utility v sysinternals)

- Kontrola:

Explorer -> disk C: -> pravá myš -> Vlastnosti -> Nástroje -> Optimalizovat a defragmentovat

# NTFS - defragmentace

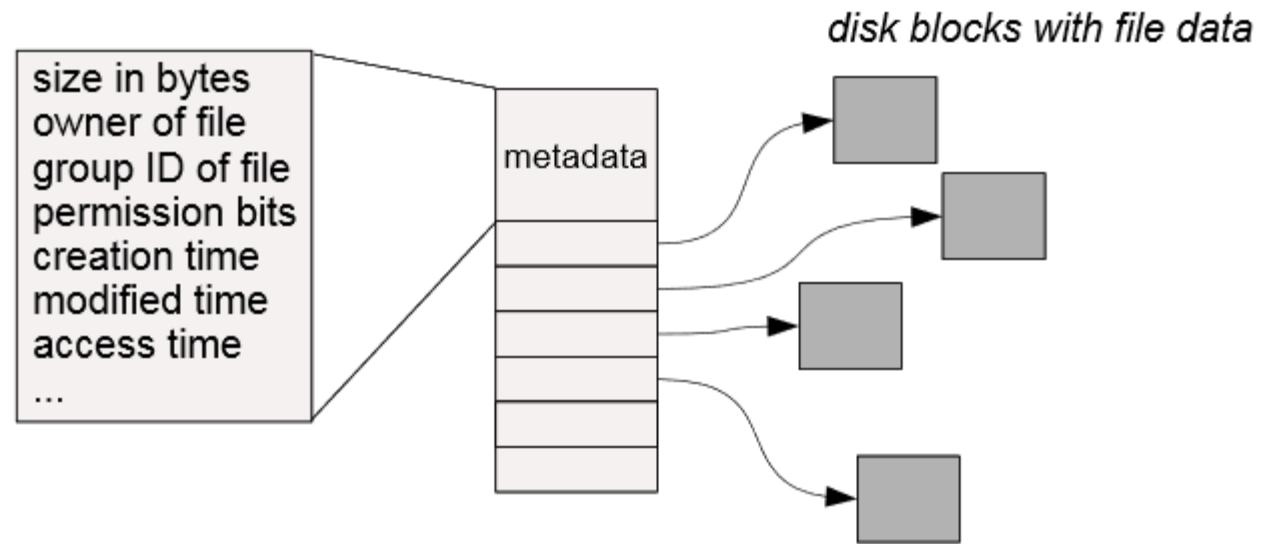


# Systémy využívající i-uzlů (!)

- Každý soubor a adresář je reprezentovaný i-uzlem (!!!!)
- i-uzel - datová struktura
  - Metadata popisující vlastníka souboru, přístupová práva, velikost
  - Umístění bloků souboru na disku
    - Přímé, nepřímé 1. 2. 3. úroveň
    - Abychom věděli, jaké bloky přistupovat

# i-uzel

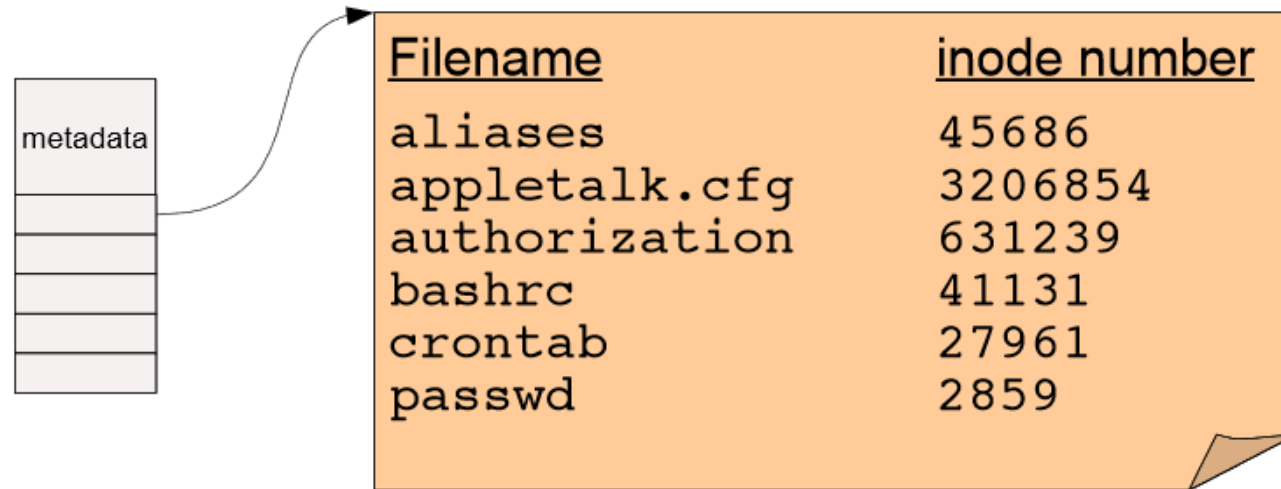
i-uzel neobsahuje jméno souboru  
!!!



Obrázek znázorňuje jeden i-uzel (metadata , přímé adresy diskových bloků)  
Pamatuj: 1 i-uzel = 1 soubor (obyčejný, adresář)

# Adresář systému s i-uzly

Soubor obsahující dvojici  
(filename, inode number)



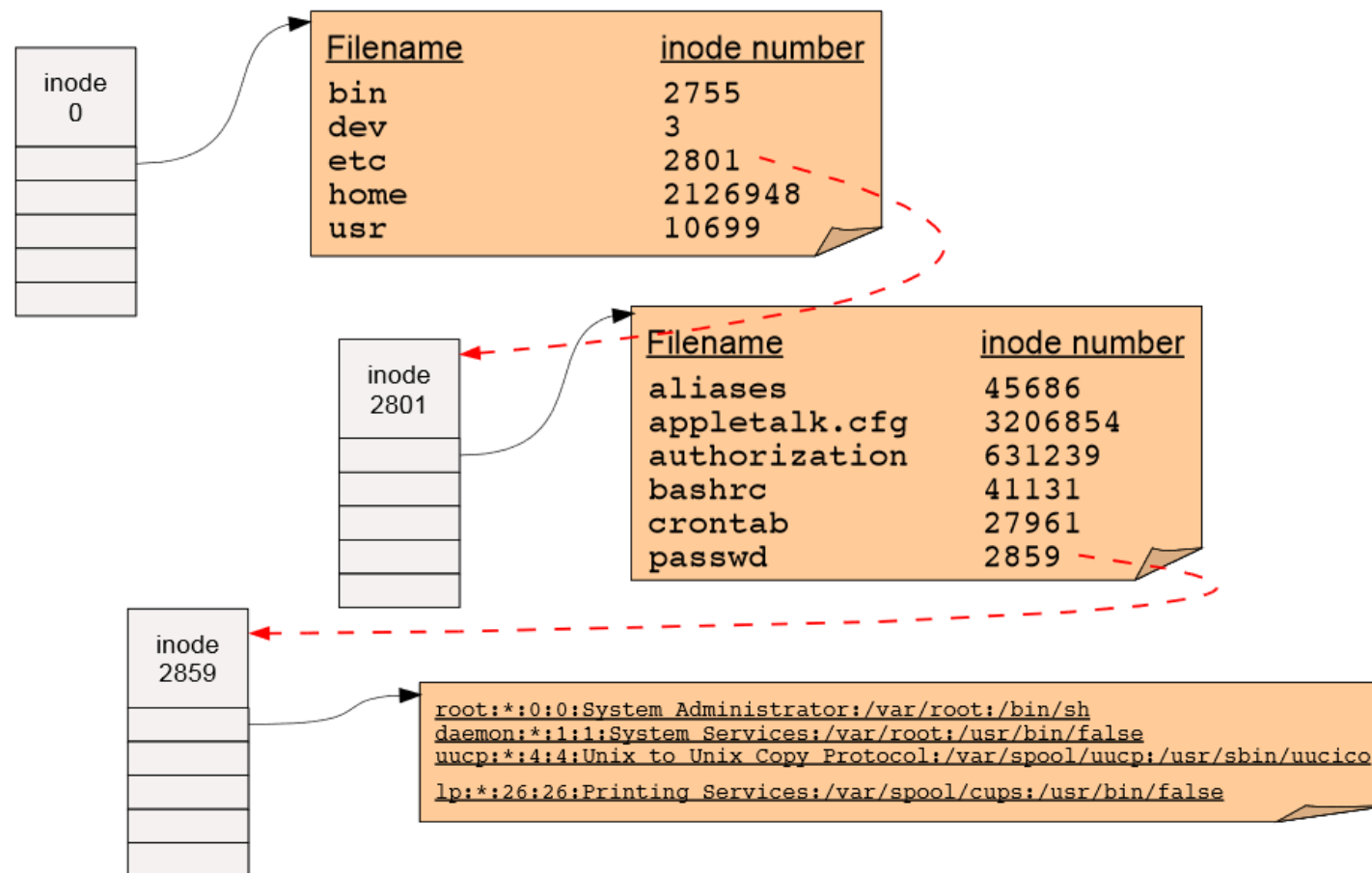
The diagram illustrates a system directory structure. On the left, a vertical stack of six light purple rectangular boxes represents a metadata table. The top box is labeled 'metadata'. A curved arrow originates from the right side of the second box from the top and points to the top-left corner of a larger orange box on the right. This orange box contains a table with two columns: 'Filename' and 'inode number'. The table lists several system files and their corresponding inode numbers.

<u>Filename</u>	<u>inode number</u>
aliases	45686
appletalk.cfg	3206854
authorization	631239
bashrc	41131
crontab	27961
passwd	2859

# PAMATUJ pro systém s i-uzly

- Položkou adresáře je dvojice (jméno souboru, číslo i-uzlu)
- Velikost zde udáváná není – je součástí i-uzlu
- Naopak i-uzel neobsahuje název souboru, ten je v adresáři !

# Prohledání cesty k souboru



Jak se dostanu k souboru `/etc/passwd`?



# Umístění i-uzlů na disku

- 1 i-uzel = 1 soubor
- Pevný počet i-uzlů = max. počet souborů na daném oddílu disku (určeno při vytvoření fs)
- Pokud nám dojdou i-uzly, nevytvoříme již další nový soubor
- Pokud dojdou i-uzly, ale datové bloky zbývají, můžeme prodloužit velikost stávajících souborů

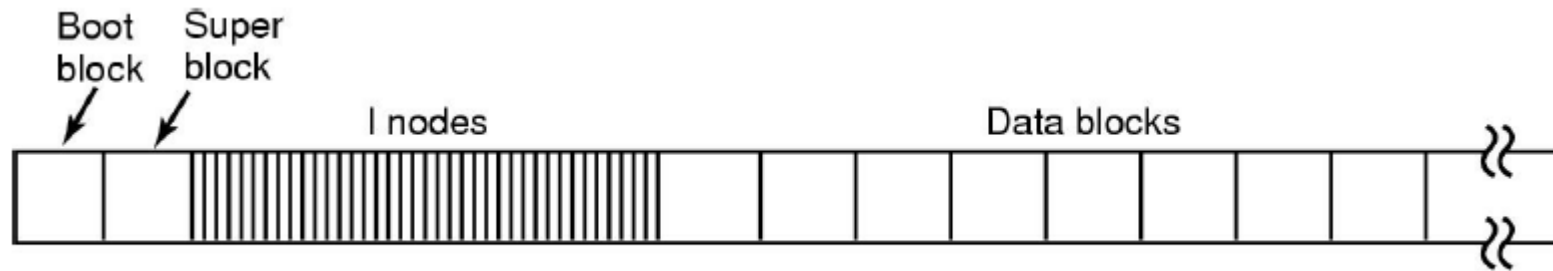
# Unixové systémy s využitím i-uzlů (původní koncepce)

takto vypadá partiton disku (např. `/dev/sda1`)

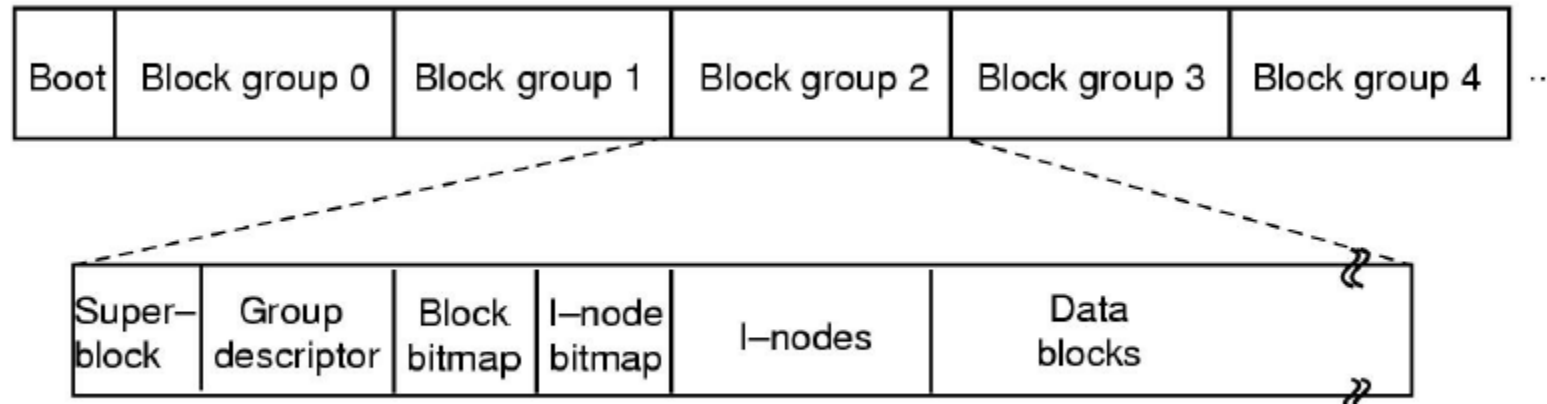
původní rozdělení v Unixových systémech

novější rozdělení, např. v ext2 viz další slide

superblock – příznak čistoty, verze, počet i-nodů, velikost alokační jednotky,  
seznam volných bloků

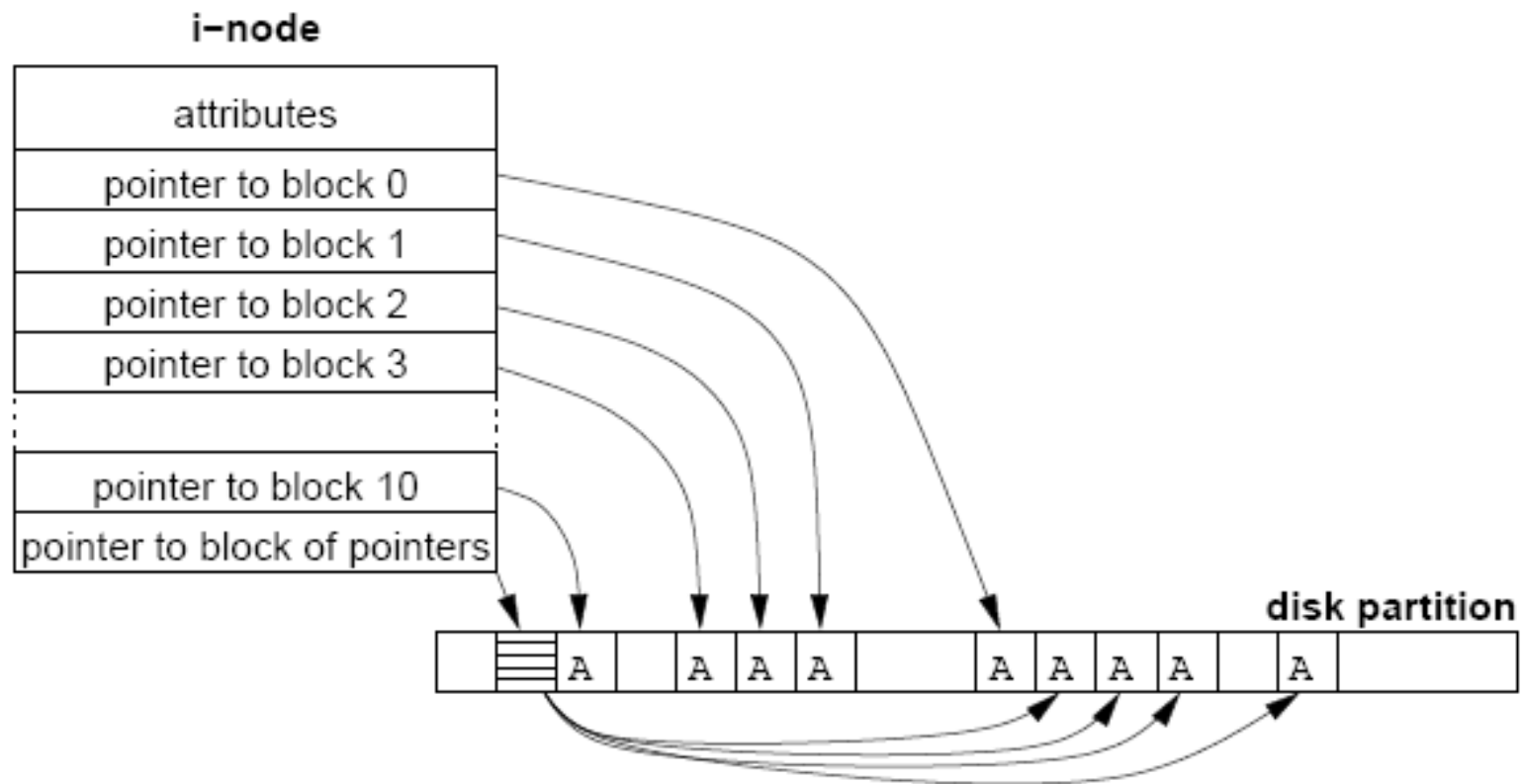


# Unixové systémy s využitím i-uzlů (novější, např. ext2)



skupiny i-nodů a datových bloků v jednotlivých skupinách (block group)  
**duplikace nejdůležitějších údajů** v každé skupině (superblock, group descriptor)

Jsou zde 2 bitmapy – který i-node je volný, který blok je volný (!!!)



Pamatuj:

Cca 10-12 přímých odkazů na bloky obsahující data souboru

1. nepřímý – odkaz na datový blok obsahující seznam odkazů na data
2. nepřímý
3. nepřímý

# Příklady filesystemů

- **FAT**

- MS DOS, paměťové karty
- Nepoužívá ACL – u souborů není žádná info o přístupových právech
- snadná přenositelnost dat mezi různými OS

- **NTFS**

- Používá se ve Windows XP/7.../10
- Používají ACL: k souboru je přiřazen seznam uživatelů, skupin a jaká mají oprávnění k souboru (!!!!)

- **Ext2**

- Použití v Linuxu, nemá žurnálování
- Práva – standardní unixová (vlastní, skupina, others), lze doplnit i ACL (komplexnější, ale samozřejmě zpomalení)

- **Ext3**

- Použití v Linuxu, má žurnál (rychlejší obnova konzistence po výpadku)

# Příklady filesystemů

- ext4
  - stejně jako ext2, ext3 používá inody
  - extenty – souvislé logické bloky
    - může být až 128MB oproti velkému počtu 4KB bloků
  - nanosekundová časová razítka
- xfs
  - Také používá žurnál
- jfs

# Přístupová práva

- Nastavení práv v Linuxu – příkaz chmod
- Změna vlastníka v Linuxu – příkaz chown
- `chown 777 s1.txt`
  - vlastník (u), skupina (g), ostatní (o)
  - Trojice práv
    - R – read
    - W – write
    - X – execute (spustit)

# Přístupová práva - speciální

- Set EUID bit (s)
  - Pustit program s právy vlastníka programu, nikoliv s právy uživatele, který program pustil
  - Např. pro změnu hesla – aby bylo možné nový hash zapsat do /etc/shadow, kam může zapisovat pouze root
- Sticky bit (t)
  - Na adresář /tmp, aby vytvořené soubory mohl mazat pouze jejich vlastník a ne ostatní uživatelé (a dále vlastník adresáře – root)



# Přístupová práva - ACL

- Klasická unixová práva – malé možnosti nastavení
  - Jen vlastník, skupina, ostatní
- Často potřebujeme více
  - Potřebujeme nastavit práva pro více uživatelů, pro více skupin
- Řešení – použití ACL
  - ACL používá např. NTFS
  - ACL lze využít i v Linuxu, např. ext4

# Přístupová práva – ACL - příklad

ACL k souboru dokument1.txt

Uživatel nebo skupina	ID uživatele nebo skupiny	Povolená práva
0	101 (pepa)	read
0	102 (lenka)	read, write
1	501 (zamestnanci)	read, write
1	502 (studenti)	read

ACL pod Linuxem:

<http://www.abclinuxu.cz/clanky/bezpecnost/acl-prakticky>

# Přístupová práva – ACL - Linux

- `setfacl -m user:milos:rw soubor`
- `getfacl soubor`
- `ls -l`  
`-rw-rw----+ 1 ondra uzivatele 0 Nov 19 16:08 soubor`  
+ říká, že jsou použita rozšířená práva

# Kontrola přístupových práv

Kdo se souborem smí pracovat (číst, zapisovat)?

- Proces běží pod nějakým uživatelem
- Kontrola, zda daný uživatel či skupina je oprávněná přistupovat

1. Neřeší se – např. FAT

2. Klasická unixová práva – vlastník, skupina, ostatní – r w x

3. ACL – komplexní práva

- Můžu nastavit pro více uživatelů různá práva (pepa, tomas, lenka)
- Můžu nastavit pro více skupin různá práva (studenti, zamestnanci)

## Kontrola konzistence souborového systému

- Kontrola datových bloků
- Kontrola počtu jmen a odkazů na ně
- Žurnálování

# Jak OS pozná, že filesystem nebyl korektně odpojen a že má provést kontrolu?

- Časté problémy mohou vzniknout při neočekávaném výpadku napájení
- Při mountování fs se do fs zapíše, že je systém připojen
- Při odpojení fs OS zapíše, že došlo ke korektnímu odpojení
- Chceme-li připojit fs a OS zjistí značku, že byl fs připojen, tedy že nebyl korektně odpojen -> jeden z důvodů provést test konzistence
- Pokud už byl fs připojen N-krát -> preventivní kontrola
- Pokud byla poslední kontrola před dlouhou dobou -> preventivní kontrola

# Kontrola datových bloků

- Datový blok může být v jednom ze dvou stavů
  - Volný
  - Patří právě jednomu souboru
- Každý jiný stav je špatný
  - (0,0) – veden že není volný ani obsazený
  - (1,1) – veden současně jako volný i jako obsazený – nebezpečný stav
  - (2,0) – stejný blok přiřazen dvěma souborům  
(stejně špatná situace, jako stejný lístek do kina prodán 2ma lidem)

# Počet výskytu souboru v adresářích

- 1 výskyt – jednomu souboru odpovídá jedno jméno v adresáři
- 0 výskytu – soubor je sice na disku, ale není v žádném adresáři
  - Není jak se k němu dostat
  - Systém může uložit soubor do lost+found
- Více výskytů – referenční číslo v i-uzlu musí odpovídat počtu výskytů
  - Jinak hrozí nesmazání souboru při smazání posledního názvu
  - Nebo hrozí smazání souboru, i když na něj ještě nějaké jméno odkazuje



# Žurnálování

Data se zapisují v transakcích  
přesun FS z jednoho konzistentního stavu do jiného také konzistentního  
Často se žurnálují jen metadata

1. Transakce se zapíše do žurnálu (logu)
2. Speciální značka – konec žurnálu
3. Zápis na disk
4. Odstranění značky konec žurnálu

# Žurnálování

- Zápis na disk = transakce
- Před zápisem na disk je transakce zapsaná do žurnálu
- Do žurnálu je přidána značka zurnal-kompletní
- Po úspěšném provedení transakce se žurnál vymaže

Dle stavu žurnálu:

- Prázdný – není třeba dělat nic
- Transakce bez značky zurnal-kompletní - smaže žurnál
- Transakce aw značkou zurnal-kompletní – provedeme transakci a smažeme žurnál

# Žurnálování - kontrola

- Při připojení diskové oblasti se zkontroluje stav žurnálu:
  - a) Žurnál je prázdný
    - data jsou konzistentní, není třeba nic dělat
  - b) Žurnál obsahuje data, ale ne značku konec žurnálu
    - data jsou jen v žurnálu, nezačal zápis do datových bloků
    - stačí smazat žurnál – o něco jsme přišli, ale data jsou konzistentní
  - c) Žurnál obsahuje data i značku konec žurnálu
    - žurnál je kompletní, přepíšeme do datových bloků
    - data budou stále konzistentní

# RAIDy

- RAID 0, 1, 5, 6, 10, 50
- Nenahrazuje zálohování, proč?
  - Ochrání před výpadkem HW – vadným diskem
  - Ale přijde uživatel – „před týdnem jsem si smazal soubor s1, můžete mi ho obnovit?“

# Unixová práva x ACL

- Unixová práva
  - Kdo je vlastník
  - Kdo je skupina
  - Práva pro vlastníka (r,w,x)
  - Práva pro skupinu (r,w,x)
  - Příkazy chmod, chown
- ACL
  - Flexibilnější
  - Práva pro uživatele – Pepa, Tomáš, Lenka
  - Práva pro více skupin – projekt1, projekt2, studenti, zam, externisté

# Další

- Vyhladovění
- Uváznutí
- Coffmannovo podmínky
- Jak se předcházet a reagovat na uváznutí
- Plánování procesů
  - Časové kvantum, preemptivní plánovač, kdy dochází k preempci, ...