VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

# the Little Book of IOS
# Poznámky z přednášek

2019/2020 (PDF z 19. června 2020)

Corse & LeO

# Obsah

**Preambule**

Dokument je dělen dle přednášek a toho, co se probíralo na přednáškách IOS (ak. r. 2019/2020). Tedy každá hlavní kategorie značí číslo přednášky, podkategorie značí probírané téma a případně jsou použity i podpodkategorie pro rozdělení velkých témat do více úseků.

Je docela možné, že spousta věcí bude pouze přepisem obsahu prezentace, nicméně takto celý dokument (ani jediný řádek, až na výjimky, jako je třeba deadlock) zpracováván nebyl. Dokument byl zpracován za pomoci záznamů přednášek, a to tak, abych dané téma či látku dostatečně pochopil a zároveň bylo zajištěno nějakým způsobem, aby dané vysvětlení nebylo až moc polopatické a stačilo např. ke zkoušce.

Někdy pro pochopení dané látky byly vhodné obrázky, někde byly i nutností (např. princip souborových systémů). Také jsou obrázky použity pro pseudokódy, protože nějaké normální zpracování kódu v LaTeXu by trvalo zbytečně dlouho. Všechny obrázky (i pseudokódy) byly použity z různých prezentací IOS. Moje rozhodně nejsou – pod popiskem každého obrázku je tak zmínka o tom, odkud daný obrázek pochází.

Co se týče pochopení obsahu, bylo by dobré spolu před čtením jakékoli kapitoly (=přednášky) navštívit danou přednášku nebo si ji alespoň pustit ze záznamu. Bez toho je možné, že některé věci nebude možné pochopit.

# 1

**První přednáška:** Úvod do předmětu, přehled operačních systémů, základní pojmy, jádro operačního systému a jejich typy, historie vývoje operačních systémů, přehled technického vybavení, klasifikace počítačů, operačních systémů, hlavní směry ve vývoji operačního systému.

## 1.1 Úvod, přehled operačních systémů

Operační systém je významnou částí výpočetních systémů; ty zahrnují:

- hardware,
- operační systém,
- uživatelské aplikační programy,
- uživatele.

Přehled některých OS:

- GNU/Linux
  - Debian – Ubuntu
  - Red Hat: RHEL, Fedora, CentOS
  - SuSE
  - Gentoo
  - Arch Linux
  - Slackware (nejstarší live distribuce Linuxu)
- BSD
  - FreeBSD, OpenBSD
- GNU
  - zn. GNU Is Not Unix
- MS Windows
- Mac OS X
  - jádro XNU = X is Not Unix
- Android, iOS
- Minix
  - používá Intel ve svých čipech

## 1.2 Základní pojmy

Operační systém je program (resp. kolekce programů), která vytváří spojující mezivrstvu mezi hardware operačního systému, uživateli a jejich uživatelskými aplikačními programy. OS dále spotřebovává zdroje, jako jsou paměť nebo čas CPU. (TLDR: SW spojující hardware, uživatele a programy.)

**Cíle OS:**

- maximální využití zdrojů počítače – drahé počítače, levnější pracovní síla (dříve)
- jednoduchost použití počítačů – levné počítače, drahá pracovní síla (dnes převažuje)

**Základní role OS:**

- správce prostředků
  - paměť, procesor, periferie
  - dovoluje sdílet prostředky efektivně a bezpečně
- tvůrce prostředí pro uživatele a jejich aplikační programy
  - vytváření abstrakcí, virtuálních objektů (resp. poskytuje standardní rozhraní, které zjednodušuje přenositelnost aplikací a zaučení uživatelů)
  - abstrakce jsou například: proces, program, soubor
  - problémy abstrakcí: menší efektivita a nepřístupné některé nízkoúrovňové operace

**OS zahrnuje:**

- jádro (kernel),
- systémové knihovny a utility (= systémové aplikační programy),
- textové (Shell) či grafické uživatelské rozhraní (X Window System).

Přesná definice, co vše OS zahrnuje, neexistuje. Různé firmy a komunity to chápou různě. (GNU to chápe např. jako projekt svobodného OS zahrnující jádro, utility, GUI, TUI, vývojové prostředky a knihovny, . . . )

### Definice:

***proces*** je aktivita řízená programem (podrobněji se jim věnujeme od 8.4)
***program*** je předpis, návod na nějakou činnost zakódovaný vhodným způsobem
***soubor*** je kolekce záznamů sloužící primárně jako základní jednotka pro ukládání dat na vnějších paměťových médiích
***adresář*** je kolekce souborů

## 1.3 Jádro operačního systému

Jedná se o nejnižší a nejzákladnější část OS. Zavádí se jako první a běží po celou dobu běhu počítačového systému (tzv. reaktivní systém, spíš než transformační). Navazuje přímo na hardware (případně virtualizovaný HW) a pro uživatele a uživatelské aplikace jej zcela zapouzdřuje.

**Běží v privilegovaném režimu:**

- je možné měnit obsah registru HW, je možné zadávat příkazy HW (není možné v uživatelském režimu)
- musí být podporováno v HW

**Jádro (obecně) zajišťuje:**

- základní správu prostředků a tvorbu základního prostředí jak pro uživatele, tak pro zbytek OS
- zahrnuje všechny operace, kdy je potřeba přímo komunikovat s hardware (přepínání kontextu – jádro, plánování procesů – někdy v jádru, někdy mimo, zavedení stránky z disku, ...)
- služby pro zbytek OS a uživatele, některé zajišťuje automaticky
- některé služby nejsou poskytovány automaticky, uživatel/aplikace si o ně musí žádat, nazýváme to volání služeb, tzv. *system-call* (= systémová volání), která musí být implementována užitím specializovaných instrukcí (Intel: SW přerušení, syscall, sysenter)

**Rozlišujeme dva typy rozhraní OS:**

***kernel interface*** také ABI, Kernel ABI – přímá volání jádra pomocí specializovaných instrukcí

***library interface*** rozhraní vyšší úrovně (např. C knihovny), typické služby jsou např. `printf` z C – volá funkce ze systémových knihoven, mohou, ale nemusí vést na volání služeb jádra (běžné aplikace pracují s tímto rozhraním)

### Definice:

***transformační systém*** je systém, který dostane nějaký vstup, zpracuje ho a udělá nějaký výstup (překladač); pokud se zacyklí, nastává chyba

***reaktivní systém*** se spustí a do (teoreticky) nekonečna reaguje na podněty uživatele (spusť proces – spustí proces); pokud přestane pracovat, chyba

***přepínání kontextu*** je situace, kdy na CPU běží proces, ten chci pozastavit a nechat běžet jiný proces

***instrukce syscall a sysenter*** jakmile aplikace (běží v uživ. režimu) zavolá takovou instrukci, dojde ke kontrolovanému přepnutí do režimu jádra, provede se služba a poté se přepne zpět

***ABI*** = Application Binary Interface

## 1.4 Typy jader OS

### Monolitická jádra

- vysokoúrovňové komplexní rozhraní s řadou služeb, abstrakcí, které mohou používat vyšší vrstvy OS
- všechny subsystémy jsou implementovány v privilegovaném režimu, režimu jádra, a zahrnují např. správu paměti, plánování, meziprocesovou komunikaci, souborové systémy, ...
- výhody: vysoká efektivita díky provázanosti
- nevýhody: malá flexibilita při práci s jádrem (v souborovém systému je chyba, chci změnit jen implementaci souborového systému za novou verzi a vše ostatní nechat – nelze, je nutné celý systém zastavit a znovu nastartovat, nelze měnit nic za běhu)

### Monolitická jádra s modulární strukturou

- vylepšení koncepce monolitických jader
- umožňuje zavádět a odstraňovat subsystémy jádra v podobě tzv. modulů za běhu
- výhody: není nutné celý systém zastavovat a znovu bootovat pro výměnu jednoho modulu, vyšší bezpečnost – zavedou se jen moduly, které se budou používat
- používané např. ve FreeBSD, Linux

### Mikrojádra

- snaha minimalizovat rozsah jádra a rozsah jeho služeb
- nabízí jednoduché rozhraní, malý počet abstrakcí, služeb, typicky nabízí nejzákladnější správu CPU, I/O zařízení, paměti, ...
- většina služeb nabízených monolitickými jádry (ovladače, významné části správy paměti, plánování) je implementována mimo jádro v tzv. serverech (neběží v privilegovaném režimu).
- výhody: flexibilita (více současně běžících implementací různých služeb, dynamické spouštění, zastavování...), zabezpečení (chyba v serveru / útok na ně neznamená ovládnutí celého OS, ale jen daného serveru)
- nevýhody: výrazně vyšší režie

### Generace mikrojader

- 1. generace – např. Mach
- 2. generace – např. L4, menší režie než 1. generace
- 3. generace – např. seL4 nebo ProvenCore, důraz na zabezpečení, návrh s ohledem na možnost formální verifikace

### Hybridní jádra

- *něco mezi mikrojádry a monolitickými jádry*
- jádra založená na mikrojádrech, rozšířená o kód, který by mohl být implementován ve formě serveru, je ale za účelem menší režie těsněji provázán s mikrojádrem a běží v jeho režimu
- používané např. v Mac OS X (Mach + BSD), Windows NT (a vyšší)

### Definice:

***servery (v oblasti mikrojader)*** jsou procesy
***formální verifikací*** rozumíme ověření určitých vlastností systému s platnosti matematického důkazu

### Linux příkazy:

***lsmod*** – vypíše aktuálně zavedené moduly jádra
***rmmod*** – maže moduly jádra
***modprobe*** – zavede modul do jádra

## 1.5 Historie vývoje OS

### Definice:

***přerušení*** je elektrický signál, který jde od periferie po sběrnici k procesoru, na CPU vyvolá *obsluhu přerušení* – mechanismus umožňující rozběhnout operaci na periferii a o tu periferii se nestarat (periferie poté oznámí konec operace) (podrobně se tomu věnuje oddíl 2.5)

***multitasking*** je současný běh více aplikací na jednom procesoru (může být s preemptivním nebo nepreemptivním plánováním)

***nepreemptivní plánování*** znamená, že úloha, která aktuálně běží na CPU, může být od CPU „odstavena“ pouze tehdy, když nějak zakomunikuje s jádrem (= požádá o službu jádra, např. periferní operace), dokonce lze použít specializované služby pro přepnutí kontextu (proces se dobrovolně vzdá CPU, tzv. yield služby); výhoda: snadná implementace, nevýhoda: pokud se proces zacyklí (chyba), celý systém se zablokuje (pořad běží jedna úloha – více viz 8.16)

***preemptivní plánování*** proces může být odstaven od CPU bez nutnosti komunikace s jádrem, např. pomocí přerušení (jakéhokoli typu – více viz 8.16)

## 1.6 Přehled technického vybavení

### Procesor (CPU):

- řadič, ALU, registry (IP, SP), instrukce, . . .

### Pamět:

- adresa
- hiearchie paměti (cache, RAM, disky, . . . – bank pamětí může být vice)
  - paměti se liší spotřebou, kapacitou, rychlostí, cenou za jednotku
  - na vrcholu hiearchie jsou registry (nejrychlejší, nejvyšší cena za jednotku, malá kapacita)
  - cache (vyrovnávací paměti různých úrovní, L1 = level 1, L2, L3)
  - primární paměť RAM
  - sekundární paměti – disky (SSD, HDD)
  - vyrovnávací paměti disku
  - terciární paměti (zálohy – nejnižší cena za jednotku, nejpomalejší, největší kapacita – pásky, CD/DVD, externí disky, cloudy, síťové disky, . . . )

### Periferie:

- disk (HDD, SSD, . . . ), klávesnice, monitor (I/O porty, přerušení, DMA)

**Sběrnice:**

- propojují jednotlivé komponenty
- na vrcholu hiearchie jsou sběrnice propojující CPU a paměť (FSB – Front Side Bus, HyperTransport QPI – Quick Path Interconnect)
- diskové sběrnice (SATA/ATA, SCSI/SAS, USB)
- další sběrnice (NVLink – připojování nVidia GPU, PCI – rozšiřující karty nebo disky, CAPI – IBM Tauer CPU, propojování CPU a akcelerátoru)

### Definice:

***I/O porty*** = vstupně-výstupní porty, představují paměťově oddělený prostor od adresového prostoru běžné paměti, s těmito adresami se komunikuje speciálními instrukcemi (Intel: `inout`)

***paměťově mapované I/O*** jsou částí adresového prostoru běžné paměti, která není použita pro práci s pamětí, ale adresy jsou přesměrované do HW (to, co zapíšu na danou adresu, nebude v paměti, ale v nějakém registru HW)

***DMA = Direct Memory Access*** souvisí s nezávislou činnosti periferií – periferie mohou přímo komunikovat s hardware (řadič disku si sám z adresy paměti načte data a přes sběrnice je přenáší na disk nebo naopak)

## 1.7 Klasifikace počítačů

**Dle účelu:**

- univerzální,
- specializované
  - vestavěné (palubní pc, spotřební elektronika, . . . )
  - aplikačně orientované (řízení databází, síťové servery, . . . )
  - vývojové (zkoušení nových technologií)

**Podle výkonnosti:**

- vestavěné PC, tablety, mobily, . . .
- osobní počítače (PC) a pracovní stanice (workstation) – dnes se nerozlišuje
- servery
- střediskové počítače (mainframe) – vyrábí IBM, laděné na obrovský I/O výkon a vysokou spolehlivost
- superpočítače – laděné na surový výpočetní výkon (vědecké výpočty, simulace)

## 1.8 Klasifikace OS

**Podle účelu:**

- univerzální (UNIX, Linux, Windows, . . . )
- specializované (real-time: RT-Linux, databáze, web, mobilní – iOS, Android)

**Podle počtu uživatelů:**

- jednouživatelské (CP/M, MS-DOS, . . . )
- víceuživatelské (UNIX, Windows, . . . )

**Podle počtu současně běžících úloh:**

- jednoúlohové
- víceúlohové (multitasking, ne/preemptivní)

### Definice:

***soft real-time*** – doporučení, aby se akce vykonávaly v reálném čase
***hard real-time*** – akce se *musí* vykonávat v reálném čase

## 1.9 Implementace OS

OS se obtížně programují a ladí, protože to jsou velké programové systémy, paralelní a asynchronní systémy, systémy závislé na technickém vybavení.

**Důsledky:**

- setrvačnost při implementaci (snaha neměnit kód, který pracuje spolehlivě)
- používání technik pro minimalizaci výskytu chyb (inspekce zdrojového kódu, rozsáhlé testování, podpora vývoje technik formální verifikace)

### Definice:

***paralelní systém*** znamená, že zde běží více aktivit současně
***paralelní asynchronní systémy*** – procesy se přepínají v okamžicích, které nelze dopředu přesně předpovědět

## 1.10 Hlavní směry ve vývoji OS

- neustálé vylepšování architektur (snižování režie jader)
- bezpečnost, spolehlivost
- podpora stále většího počtu procesorů, více jader
- virtualizace
- distribuované zpracování (cloudy, kontejnery, Internet of Things)
- OS tabletů, mobilů, vestavěných systémů, . . .
- vývoj nových technik návrhu a implementace OS (podpora formální verifikace)

### Definice:

***bezpečnost*** znamená, že je systém odolný vůči vnějším útokům
***spolehlivost*** znamená, že systém „nespadne sám od sebe“

# 2

**Druhá přednáška:** Unix – úvod: historie UNIXu (nezkouší se), příčiny úspěchu UNIXu, varianty UNIXu, základní koncepty, struktura jádra, komunikace s jádrem – hardwarová přerušení. Přehled programování v UNIXu: nástroje programátora, ...

## 2.1 Příčiny úspěchu UNIXu

- víceprocesový, víceuživatelský,
- napsán v C – přenositelný,
- zpočátku (a později) šířen ve zdrojovém tvaru,
- *„mechanism, not policy“*,
- *„fun to hack“*,
- jednoduché uživatelské rozhraní (terminál),
- skládání složitějších programů z jednodušších (tvoření aplikací typu filtr),
- hierarchický systém souborů,
- konzistentní rozhraní periferních zařízení

### Definice:

***mechanism, not policy*** – snaha oddělit části aplikací (například GUI: oddělit základní rutiny pro vykreslování grafiky od politik, čili koncové nástavby – barvy oken, umístění tlačítek, ... – systematické rozdělení vede k lepšímu ladění a lepší optimalizaci algoritmů a zároveň k rychlým změnám politik

***fun to hack*** – lidé se na vývoji podílí, protože je to baví (nejen proto, že jsou za to placeni)

***aplikace typu filtr*** – jednoduché otevřené aplikace, na vstupu mají textový dokument v otevřené podobě, vstup zpracují a na výstupu opět otevřený dokument (žádné binární, zakódované)

## 2.2 Varianty UNIXu

**Hlavní větve OS UNIXového typu:**

- UNIX System V (původní systém z AT&T),
- BSD UNIX (FreeBSD, NetBSD, ...),
- firemní varianty (AIX, Solaris, ...)
- Linux

**Související normy:**

- XPG – X/OPEN, SVR4 – AT&T,SUN, OSF/1, Single UNIX Specification,
- POSIX – IEEE standard,
- Single UNIX Specification v3/v4 – shell, utility (CLI), API

### Definice:

***POSIX*** je striktní podmnožina Single UNIX Specification, je to standard definující základní textové příkazové rozhraní OS + API

## 2.3 Základní koncepty

Jsou dvě základní koncepce (abstrakce) UNIXu: **procesy** a **soubory**.

Procesy mezi sebou komunikují pomocí různých mechanismů meziprocesové komunikace – *IPC* (Inter-Process Communication) – roury, signály, semafory, sdílená paměť, sockets, zprávy, streams, … a pro komunikaci používají nějaké I/O rozhraní (read, write, close, …)

### Definice:

***procesy*** jsou abstrakcí nějaké probíhající aktivity (viz 1.2)
***soubory*** jsou abstrakcí dat (viz 1.2)

## 2.4 Struktura jádra UNIXu

Základní podsystémy jsou správa souborů a správa procesů.

**Popis:**

- Na horním okraji jádra (směrem k uživatelům, aplikacím) je vrstva implementující rozhraní volání služeb, prostřednictvím které jádro přebírá žádosti o služby od aplikací. Rozhraní kontroluje, zda ten, kdo o službu žádá, ji může volat, zda jsou parametry validní, a poté rozhraní předává požadavek dál do jádra.
- Aplikace mohou s jádrem komunikovat přímo, nicméně nejčastěji komunikují s jádrem přes knihovny (viz 1.3).
- Na druhém okraji (těsně nad HW) je vrstva abstrakce hardware.
- Mezi správou souborů a hardware se nachází ovladače, poté vrstva vyrovnávacích pamětí, které souborové systémy používají ke zrychlení práce s relativně pomalými disky (HDD, SSD – oproti RAM pomalé) – OS se snaží vyhnout opakovanému čtení stejných dat, proto si v jednom okamžiku načte víc dat, než uživatel zadá, uloží si data do vyrovnávací paměti (při dostatku paměti) a data načítá odtud. (Např. C knihovny jsou používané každým druhým programem – jsou v paměti téměř pořád).

### Definice:

***ovladače*** jsou programy sloužící k řízení (zadávání příkazů, přebírání stavových informací, řešení mimořádných stavů konkrétních periferií) – lze je (jako i příslušná zařízení) rozdělit na znaková a bloková (kratší definice viz 5.9)
***znaková zařízení*** jsou zařízení komunikující po jednotlivých znacích (klávesnice)
***bloková zařízení*** komunikují po blocích (disk – sektory, resp. bloky)
***komunikací s jádrem*** rozumíme nastavování parametrů hardware, vydávání příkazů HW, obsluhu různých stavů, do kterých se HW dostává a o kterých je CPU a jádro informováno prostřednictvím přerušení
***nastavování parametrů HW*** se děje pomocí I/O portu nebo paměťově mapovaných operací (viz 1.6)

## 2.5 Komunikace s jádrem a hardwarová přerušení

Služby jádra jsou operace, jejichž realizace je pro procesy zajišťována jádrem. Explicitně je možné o provedení určité služby žádat prostřednictvím systémových volání (viz 1.3).

**Příklady některých služeb jádra (systémová volání v UNIXu):**

- open, close, read – otevře/zavře/čte soubor,
- write – zapisuje,
- kill – pošle signál,
- fork – duplikuje proces,
- exec – přepíše kód,
- exit – ukončí proces.

### 2.5.1 Hardwarová přerušení

- *hardware interrupt* je mechanismus, kterým HW zařízení oznamují jádru asynchronně vznik události, které je zapotřebí obsloužit (další možná definice viz 1.5),
- žádosti o HW přerušení (IRQ, interrupt request) přichází jako elektrické signály do řadiče přerušení (APIC),
- procesor s řadičem přerušení komunikuje pomocí I/O portu.

**Příjem nebo obsluhu HW přerušení lze zakázat:**

- maskováním přerušení,
- na CPU (instrukce CLI/STI na Intel/AMD – zakážou se všechna kromě NMI),
- čistě programově v jádře (přerušení se přijme, ale jádro si jen poznamená jeho příchod a neobsluhuje se)

**NMI:**

- *non-maskable interrupt* je HW přerušení, které nelze zamaskovat na řadiči ani zakázat na CPU,
- používá se při kritických chybách paměti, sběrnice, ... (alternativně se používá pro ladění nebo řešení uváznutí v jádře – „NMI watchdog“)

**Přerušení mohou vznikat i v CPU – jsou to synchronní přerušení, tzv. výjimky (= exceptions):**

- trap – po obsluze se pokračuje další instrukcí (breakpoint, overflow, ...)
- fault – po obsluze se znovu opakuje instrukce, která výjimku vyvolala (výpadek stránky, dělení 0, ...)
- abort – dochází k závažným problémům detekovaným CPU, není jasné, jak pokračovat – provedení se ukončí (zanořené výjimky typu fault, chyby HW detekované CPU)

**Mohou existovat i další typy přerušení:** (tato přerušení obsluhuje CPU zcela specifickým způsobem, často mimo vliv jádra, např. na Intel/AMD)

- Interprocessor interrupt (IPI)
  - meziprocesorové přerušení
  - používá se pro přeposílání přerušení z jednoho CPU na druhý nebo pro správu cache (každý CPU má svoji cache, do nich mohou mít CPU načtený stejné adresy z paměti – pokud dojde ke změnám v paměti, musí CPU informovat ostatní CPU o změně)
- System management Interrput (SMI)
  - přerušení typu správa systému
  - může být vyvoláno HW i SW ve zvláštních situacích

  - pokud se takové přerušení vyvolá, dostane se ke slovu firmware, který provádí obsluhu různých chybových stavů (přehřátí, vybitá baterie, ...)
  - v rámci SMI neběží běžné aplikace ani jádro, obsluha SMI nesmí běžet příliš dlouho (systém se může dostat do nekonzistentního stavu)

### 2.5.2 Zakazování přerušení

**Proc přerušení zakazovat?**

- v rámci obsluhy jednoho přerušení může nastat další přerušení,
- např. na CPU běží výpočet, něco nastane na disku, disk pošle přerušení, to dojde k CPU a jádro začne přerušení obsluhovat, v ten moment se něco stane na klávesnici a přijde další přerušení,
- poté dále v rámci obsluhy může jádro upravovat různé své interní struktury, které mohou být v nekonzistentním stavu (např. zřetězené seznamy procesů [ukazatele], různě si je propojuje a než je stihne propojit, přijde další proces a může sáhnout do paměti, kam nemá),
- proto obsluha přerušení musí být synchronizována a v případě, že se v rámci přerušení provádí nějaká kritická operace, je nutné vyloučit ostatní (všechna) přerušení

Pokud však zakážeme (nějaká/všechna) přerušení, abychom se mohli věnovat obsluze jednoho, které pak budu obsluhovat příliš dlouho, systém se může dostat do nekonzistentního stavu (jako u SMI). Používají se proto dva přístupy:

- je snaha zakazovat jen přerušení s nižšími prioritami,
- rozdělit obsluhu přerušení do více částí (úrovní).

**Obsluha přerušení je často dělená na dvě úrovně:**

- 1. úroveň:
  - má být co nejkratší,
  - v rámci obsluhy přerušení se zakomunikuje nezbytným způsobem s HW (převzetí dat z/do HW, vydání příkazu HW, ...) a naplánuje se běh 2. úrovně,
  - nelze použít běžné synchronizační prostředky (protože např. na CPU běží nějaký výpočet, přijde přerušení z disku, jádro začne řešit 1. úroveň obsluhy, ale obsluha není proces)
- 2. úroveň:
  - dokončuje obsluhu přerušení,
  - provádí se operace, kdy není potřeba komunikovat s hardware,
  - nemusí se zakazovat přerušení,
  - může běžet v speciálních procesech (interrupt threads ve FreeBSD nebo tasklety/softIRQ v Linuxu),
  - mohou se použít běžné synchronizační prostředky

### 2.5.3 Přerušení a ovladače zařízení

- při inicializaci ovladače (v Linuxu je to typicky modul) nebo při jeho prvním použití se musí registrovat k obsluze určitého IRQ,
- buď u některých zařízení se používají (historicky) zafixovaná čísla přerušení,
- nebo ovladač může zjistit číslo přerušení tak, že zakomunikuje s řadičem sběrnic; pokud to nefunguje,
- ovladač vydá příkaz zařízení, které má ovládat, aby začalo vysílat nějaká přerušení (a „poslouchala na sběrnici, kdo se ozve“),
- poté se zaregistruje k obsluze příslušného přerušení a hardware se přes tabulku přerušení ovladač „dostane ke slovu“,

- více zařízení však může používat stejné číslo žádosti o přerušení
  - v takovém případě jádro vytvoří zřetězený seznam ovladačů, které mají zájem o dané přerušení
  - ovladače musí být napsané tak, že pokud jim dojde přerušení, o které mají zájem, musí zakomunikovat s tím zařízením a zeptat se ho, zda opravdu to zařízení poslalo dané přerušení
  - pokud ano, obslouží se, pokud ne, předá se řízení přerušení dalšímu ovladači v seznamu

### 2.5.4 Příklad komunikace s jádrem

Synchronní komunikace je proces–jádro, asynchronní je hardware–jádro. Detailnější příklad (na téma přístupy na disk) viz 5.9).

- proces A zavolá službu read() a jádro ihned začne volání obsluhovat (synchronní)
- nejprve se podívá do cache, zda tam už nejsou data, o která má zájem proces A
- pokud ano, tak mu je rychle nakopíruje z cache na adresu, kterou požaduje proces (bez komunikace s diskem)
- pokud data nejsou v cache, proces A bude pozastaven a jádro vydá prostřednictvím ovladačů disku příkaz k načtení určitého objemu dat, typicky více než zadá uživatel, a načítá do vyrovnávací paměti (ne na požadovanou adresu)
- na procesoru dále běží proces B, taky požádá o read(), zopakuje se to samé, co u A
- až disk dokončí operace jednoho z procesů (nemusí být v pořadí volání), disk pošle přerušení na CPU
- jádro bude informováno, že má potřebná data pro proces A/B
- z cache nakopíruje požadovaná data na požadovanou adresu
- poté se proces A/B probudí a běží dál, to samé se stane u dalšího procesu

## Definice (pro 2.5.x):

***asynchronní*** – bez přímé (okamžité) vazby na to, co dělá jádro nebo aplikace (tiskárna tiskne – operace někdy skončí – ale nikdy nevím dopředu, kdy přesně)

***synchronní*** – CPU něco provede a ihned se zavolá přerušení (např. dělení 0)

***řadič přerušení*** – interrupt controller, hardwarová jednotka, která předává přerušení do CPU – registruje příchozí IRQ, ty se dle priorit předávají do CPU (přerušení je možné také zamaskovat – nepředávat dál do CPU) v podobě čísla přerušení, CPU se automaticky přepne do chráněného režimu a spustí obslužnou rutinu definovanou jádrem (přerušení 1 – provede xxx, 2 – xxx, . . . )

***APIC*** – Advanced Programmable Interrupt Controller, distribuovaný systém, každý CPU má lokální APIC, externí zařízení mohou být připojena přímo, nebo přes I/O APIC

***NMI watchdog*** – jádro si nadefinuje, že časovač mu každých *n* časových jednotek pošle toto přerušení – pokud dojde v jádře k uváznutí při obsluze jiného přerušení a všechna přerušení budou zakázána, toto se vždy dostane do CPU (jádro se může zotavit)

***výpadek stránky*** – když proces bude sahat do paměti a sáhne na stránku, která v ní není (detekuje se, že stránka tam není) – porušení ochrany paměti – jádro zkontroluje, zda proces nesahá, kam by neměl, a pokud ne, tak mu stránku nahraje zpět do paměti a znovu se provede ta stejná instrukce (viz 13.2)

***běžné synchronizační prostředky*** jsou např. semafory nebo zámky, synchronizují procesy (viz 10.1)

**Linux:** Základní statistiky o obsluze přerušení jsou v `proc/interrupts`.

## 2.6 Nástroje programátora UNIXu

X Window System, vzdálený přístup přes X Window, užitečné příkazy na Linuxu, ovládání vimu, apod. – více viz 2. přednáška IOS, u zkoušky to nebývá.

# 3

## 3.1 Bash, shell, experimenty

# 4

## 4.1 Bash, shell, experimenty

Třetí a čtvrtá přednáška je věnována hlavně shellu, prochází se prakticky různé příkazy a provádí se experimenty, apod. – lepší je zhlédnout + na zkoušce nic takového nebývá.

# 5

**Pátá přednáška:** Správa souborů: pevný disk, diskové sběrnice, sektory, parametry pevných disků, SSD, problematika zápisu na SSD, zabezpečení disků, disková pole (RAID), uložení dat na disku, fragmentace, přístup na disk a jeho plánování, logický disk.

## 5.1 Pevný disk

**Popis:**

- uvnitř mají řadu kulatých ploten, záznam se provádí na každém z těch dvou povrchů, je v soustředných kružnicích (= *tracks*, stopy)
- všechny plotny jsou na stejné ose, přidělané k sobě a rotují současně
- k načítání slouží sada hlaviček, čtecí a zápisové, jsou tam v tolika kusech, kolik je tam povrchů (např. 3 plotny = 6 povrchů = 6 hlaviček), všechny umístěné na jednom rameni, všechny hlavičky se pohybují současně
- hlavičky jsou nastavené na sadě několika stop (kružnic) o stejném průměru = cylindr
- stopy se dělí na sektory
- velikosti sektoru byly dříve 512 B, u CD/DVD 2048 B, dnes 4096 B

**Adresace sektorů:**

- ze začátku se používal CHS – určí se, se kterým cylindrem chci pracovat, dále s kterou hlavou a jakým sektorem v rámci stopy,
- v současné době se používá LBA, kde jsou sektory (bloky) číslované (jako adresy v paměti) od 0 po n, disková jednotka si musí tato čísla převádět na CHS

**Periferní/disková rozhraní:**

- používají se pro připojení disků,
- nejběžněji se používá ATA, dříve se používala v paralelní verzi (PATA – jednotlivé byty se posílaly paralelně, při rostoucích rychlostech byl problém zajistit synchronizaci těchto dat), nyní v sériové verzi (SATA)
- také se používá SCSI či SAS (Serial Attached SCSI), USB, FireWire, FibreChannel, Thunderbolt, PCI Express nebo NVMe (připojování nejrychlejších SSD),
- nad těmito rozhraními může být další HW rozhraní propojující tyto sběrnice, jako třeba AHCI, OHCI, UHCI, …

**Diskové sběrnice se liší:**

- rychlostí (SATA do 6 Gbit/s, SAS 22,5 Gbit/s),
- počtem připojených zařízení (SATA desítky, SAS 65 535),
- maximální délkou kabelů (SATA 1–2 m, SAS 10 m),
- architekturou připojení (u SAS možnost připojení jednoho zařízení více cestami),
- seznamem příkazů, které zařízení umí (flexibilita při chybách, selháních, zotavení, …)

Přes diskové sběrnice je možné mít připojené i jiné typy pamětí, jako jsou flash disky, SSD, pásky, CD/DVD/BD či terciární paměti. V systému vzniká hierarchie pamětí, viz 1.6.

### Definice:

***cylindr*** (v HDD) je množina stop o stejném průměru
***sektor*** je nejmenší jednotka diskového prostoru, který mi umožní disková elektronika načíst nebo zapsat

***(diskový) blok*** je sektor v HDD
***alokační blok/blok souborového systému*** je nejmenší jednotka, kterou OS umožní alokovat
***CHS*** – Cylinder Head Sector
***LBA*** – Linear Block Adress

## 5.2 Parametry pevných disků

Přístupová doba sestává z **doby vystavení hlav** a **rotačního zpoždění**.

Typické parametry současných disků jsou kapacita, průměrná doba přístupu (jednotky ms u HDD), otáčky a přenosová rychlost. U přenosových rychlostí se rozlišuje *sustained tranfer rate* a *maximum transfer rate*.

Mazání dat probíhá tak, že se přepíšou metadata, pouze se poznamená (OS), že daný soubor byl smazán.

### Definice:

***doba vystavení hlav*** – pokud nejsou nastavené hlavičky na stopě, se kterou chci pracovat (málokdy), tak je nutné jimi pohnout (víc zasunout dovnitř nebo vysunout)
***rotační zpoždění*** je doba, než mi pod správně nastavenou hlavičku najede sektor (narotuje se disk)
***maximum transfer rate*** je špičková přenosová rychlost, jak maximálně rychle je schopen disk komunikovat po krátkou dobu (typicky rychlost předání dat z vyrovnávacích pamětí disku)
***sustained transfer rate*** je opravdová rychlost čtení z ploten

**Linux:** *hdparm [-t]* umožňuje změřit přenosovou rychlost a měnit parametry disku, `-T` měří rychlost přenosů z vyrovnávací paměti OS (RAM).

## 5.3 Solid State Drive – SSD

Mohou být založena na různých technologiích, nejčastěji na nevolatilních pamětech NAND flash nebo DRAM (se zálohovaným napájením) či na kombinacích.

### 5.3.1 Klady a zápory SSD

**Výhody:**

- rychlý (okamžitý) náběh,
- náhodný přístup (mikrosekundy),
- větší přenosové rychlosti (stovky MB/s, ATA do 600 MB/s, 3,5 GB/s s M.2, 7 GB/s s PCI Express 4),
- zápis může být mírně pomalejší,
- tichý provoz, lepší mechanická a magnetická odolnost,
- obvykle nižší spotřeba (neplatí pro DRAM).

**Nevýhody:**

- vyšší cena za jednotku prostoru,
- omezený počet přepisů (nevýznamné pro běžný provoz),
- větší riziko katastrofického selhání,
- menší výdrž mimo provoz (při vypnutém napájení a skladování),
- komplikace se zabezpečením (bezpečné mazání nebo šifrování přepisem dat – vyžaduje speciální podporu).

### 5.3.2 Problematika zápisu u SSD

NAND flash SSD jsou organizovány do stránek (typicky 4 KiB) a ty jsou sdruženy do bloku (typicky 128 stránek = 512 KiB).

**Zápis nebo přepis dat:**

- prázdné stránky lze zapisovat jednotlivě (přepisovat ne!),
- pokud chci přepisovat jednu stránku, je nutné celý blok načíst do paměti, vymazat (zresetovat) a v paměti upravený blok načíst zpět (= write amplification, zesílení zápisu – mnohonásobné zpomalení),
- problém je menší při sekvenčním (počkám až budu mít dost dat tak, aby pokryly blok) než při náhodném zápisu do souboru.

**Problém se šifrováním a bezpečným mazáním:**

- kvůli způsobu, jakým SSD přepisují data, se data několikrát přesouvají po disku,
- proto disk musí poskytovat HW podporu pro bezpečné mazání nebo šifrování.

**Řešení problému přepisů u SSD:**

- typicky má SSD více stránek (bloků), než je deklarovaná kapacita,
- při přepsání se zapíše do volné stránky,
- po smazání dostatku stránek (tak, že tvoří blok) se blok zresetuje – příkazem TRIM souborový systém sdělí SSD, které stránky již nejsou používané (a které bloky může SSD smazat),
- řadič SSD může stránky přesouvat tak, aby si některé bloky uvolnil (pokud je v bloku málo stránek, přesunou se a blok se zresetuje),
- TRIM nelze použít vždy (typicky pokud v souborovém systému máme obraz jiného souborového systému, nemusí být možné sdělit základnímu FS informace o prázdných blocích, apod. nebo databáze, které si ukládají data do velkého předalokovaného prostoru, či obrazy virtuálních strojů a virtuální disky)

Řadič SSD přesouvá i dlouho nezměněné stránky, aby minimalizoval počet přepisů stránek.

### Definice:

***nevolatilní*** – po ztrátě napájení zůstane obsah zachován (alespoň po nějakou *rozumnou* dobu
***stránka*** – nejmenší jednotka dat, kterou lze do SSD zapsat

## 5.4 Zabezpečení disku

Disková elektronika typicky ukládaná data (sama o sobě) zabezpečuje kódy, které umí při následném čtení detekovat a případně opravit chyby – používá *ECC*. Detekce a oprava chyb je pouze v režii disku, pokud disk detekuje chybu a není příliš velká, chybu opraví a data uloží na jiný sektor, poznačí si, že ten sektor nemá používat.)

Existují technologie, které umožňují zjistit, v jakém stavu disk je (statistiky přemapování, počet chybných sektorů, ...) – *S.M.A.R.T* (podporovaná všemi „rozumnými“ disky).

Pak je možné ještě provádět testování na úrovni OS, např. e2fsck nebo badblocks nebo si některé FS (RFS, ZFS) provádějí kontinuální kontroly toho, co se ve FS děje. Tyto utility nebo FS mohou chyby detekovat a varovat o nich, rovnou je opravit (pokud není chyba příliš velká) či vyřadit použití některých sektorů.

**Definice:**

***ECC*** – Error Correction Code
***S.M.A.R.T.*** – Self Monitoring Analysis and Reporting Technology
***kontinuální kontroly ve FS*** – FS si ukládají kontrolní součty, při práci se souborem kontrolují, jestli součty souhlasí

**Linux:**

***smartctl*** je příkaz umožňující využití technologie S.M.A.R.T (testy disku, statistiky, . . . )
***smartd*** je nadstavbou smartctl (pravidelné spouštění testů, . . . )

## 5.5 Disková pole (RAID)

RAID je technologie umožňující z většího počtu (levnějších a ne příliš spolehlivých, výkonných) disků vytvořit jeden disk, který je rychlejší a spolehlivější.

**Může být implementován:**

- hardwarově (do rozšiřující karty připojíme několik disků, karta implementuje RAID),
- subsystémem v jádře,
- některé souborové systémy mají implementaci RAID v sobě.

Různých typů RAID je několik (tzv. raid levels):

### 5.5.1 RAID 0

- data jsou rozložena po dvou či více discích, ale každý datový blok je uložen jen na jednom disku (např. dva disky, 0 a 1, první datový blok [sektor, skupina sektorů] je na 0, druhý na 1, třetí na 0, . . . )
- vyšší efektivita čtení či zápisu,
- je možné paralelně číst či zapisovat (do více disků),
- prudce snižuje spolehlivost – pokud selže jeden disk, přijdu o data na něm

### 5.5.2 RAID 1

- disk mirroring, pro 2 a více disků,
- všechny bloky dat se zapisují na všechny disky,
- možnost číst a zapisovat paralelně,
- vyšší spolehlivost (data jsou na všech discích)

### 5.5.3 RAID 2

- nejsložitější, proto se příliš nepoužívá,
- používá zabezpečovací Hemingovy kódy,
- k určitému počtu datových disků je určitý počet zabezpečovacích disků,
- data se rozdělují a ukládají na datových discích na úrovni bitů, k nim se dopočítávají zabezpečovací kódy (např. 4 datové + 3 zabezpečovací),
- bity dat se rozloží do všech disků (zapisuje se po sektorech)
- jediný RAID, který umí detekovat chyby, některé i sám opravit, dokonce umí i zjistit, který disk selhal

### 5.5.4 RAID 3

- v praxi se nepoužívá
- jednodušší zabezpečení než RAID 2 v podobě *paritních bitů*,
- rozkládá data po bytech či skupinách bytů, které zabezpečuje paritním zabezpečením (např. 4 disky – 3 datové a 1 paritní)

### 5.5.5 RAID 4

- podobně jako RAID 3, ale rozkládání se provádí na úrovni bloků (sektorů),
- nevýhoda u RAID 3 i 4 je přetížení paritního disku – při zápisu/čtení se vždy pracuje s paritním diskem (a datovým) – na paritní disk se zapisuje tolikrát častěji, kolik mám datových disků, tzn. větší pravděpodobnost selhání

### 5.5.6 RAID 5

- prakticky se už používá,
- funkce paritního disku není vyhrazena pro jeden disk, ale mezi disky tzv. rotuje,
- např. v konfiguraci se 4 disky, první 3 datové bloky se uloží na 3 disky, na posledním bude parita, pro další trojici se uloží na 3. disk, pro další na 2., další na 1., a potom zase na poslední, apod. $\Rightarrow$ rovnoměrné zatížení disku,
- díky paritě jsme schopni opět detekovat a korigovat chybu v jednom disku (počet bitů není sudý – chybí tam parity bit),
- parita se počítá dle sektorů (první bit 1. sektoru, první 2. sektoru, ...),
- pokud selže více disků, nelze dopočítat bity (data)

### 5.5.7 RAID 6

- parita se ukládá 2×,
- dokáže se vyrovnat se selháním až 2 disků,
- větší redundance dat (obětují se 2 disky jako parita)

RAID je možné vytvořit i na jednom fyzickém disku (na kterém jsou logické disky).

## 5.6 Opravy chyb u paritních disků

- paritní disky používané u RAID 3, 4, 5, 6,
- jakmile člověk určí disk, který selhal, je možné zreprodukovat jeho obsah,
- příklad: 4 disky, 1 paritní, třetí datový selže
  - první byty v datových jsou 010 (potom v paritním, aby byl sudý počet, je 1), další byty jsou 111 (lichá parita, do paritního disku se doplní 1 na sudou), další jsou 011 (sudá – v paritním je 0),
  - selže třetí disk, vymění se za nový, prázdný,
  - dopočítají se data opět na sudou paritu: mám první byty 01? a v paritním 1 – aby byla sudá, v novém disku musí být 0, další byty 11? a v paritním 1 – v novém musí být 1, apod. ...

### Definice (pro 5.5.x):

***RAID*** – Redundant Array of Independent Disks
***parita bitů*** je sudost/lichost bitů, počet sudých/lichých 1 bitu
***parity bit*** – na MSB se přidá 1 pokud počet 1 (bitů) je lichý

## 5.7 Uložení dat na disku

Disková jednotka neumožní pracovat s ničím menším než sektor, ale typicky OS si sektory nějak seskupí (do větší jednotky) a neumožní pracovat s ničím menším, než je *alokační blok*.

**Logická a fyzická následnost:**

- 1 alokační blok se namapuje fyzicky za sebou na diskovém prostoru,
- více alokačních bloků již nemusí být fyzicky na disku za sebou (FS se však snaží o to, aby tomu tak bylo)

### Definice:

***alokační blok neboli cluster*** je skupina pevného počtu sektorů, typicky mocnina 2 (nejméně $2^0$ = 1 alokační blok), pro sektory v rámci alokačního bloku je zaručeno, že jdou za sebou logicky i fyzicky (na disku) v souboru, dále je to nejmenší jednotkou diskového prostoru, se kterým běžně pracuje jádro (a tedy i souborový systém, uživatel).

## 5.8 Fragmentace

### 5.8.1 Externí fragmentace

Externí fragmentací rozumíme jev, který vzniká v pamětech postupným obsazováním a uvolňováním paměti, kdy v paměti vzniká sekvence oblastí, které jsou volné a použité (různými soubory).

**Příklad externí fragmentace:**

- na disku vytvořím soubor 1, zabírá určité místo,
- poté další soubor 2,
- poté soubor 1 chci zvětšit, tak se soubor 1 rozdělí na 2 části – soubor 1.1 (původní místo kde byl – před s2) a soubor 1.2, který bude za souborem 2,
- stejným způsobem zvětším soubor 2 a vznikne sekvence s1.1, s2.1, s1.2, s2.2,
- nyní se rozhodnu smazat první soubor a budu mít sekvenci volné místo, s2.1, volné místo, s2.2 == externí fragmentace.

Externí fragmentace je i na plně obsazeném disku, kde stačí, aby byl disk obsazen soubory nespojitě (tzn. jeden soubor je rozdělen do více částí, není uložený na jednom místě, např. s1.1, s2, s1.2 nebo viz příklad nahoře).

**Negativní dopady externí fragmentace:**

- na disku za určitých okolností (v běžných FS nevznikají) mohou vzniknout části prostoru, které jsou již dále nevyužitelné, protože jsou příliš malé (TLDR: vznik volných úseků, které nejdou využít)
- okolnosti, při kterých vzniknou nevyužitelné části prostoru:
  - při alokování diskového prostoru spojitě (na míru souboru či jeho částem, nepřidělování po jednotkách pevné velikosti) a navíc budu mít dolní mez určující velikost diskového prostoru tak, aby byl použitelný (může vzniknout v souvislosti s tím, že do použitých diskových oblastí si mohu ukládat pomocné informace, k čemu se používají pokud bude informace větší než „volná díra“ – nepoužitelná)
  - mám na disku bloky volného místa o požadované velikosti (1 GB, soubor 0,5 GB), ale protože chci ukládat spojitě (soubor o velikosti 1,5 GB), nelze takové místo využít
  - vznikne nespojitě rozložený soubor (viz příklad) a je nutné si pamatovat v pomocných datech = metadatech informace o tom, kde jednotlivé části souboru jsou (ukládají se na místa, kde jednotlivé části jsou, „odkazují“ se na další metadata – další části smazaného souboru),

- čím více částí souboru, tím více metadat – čím více fragmentované, tím více je přístup na data pomalejší (u HDD se čeká navíc na natočení hlaviček a rotace disku)

**Souborové systémy se snaží negativní dopady fragmentace minimalizovat:**

- rozložením souborů po disku (snaha ukládat soubory na disk tak, aby nebyly nutně za sebou, ale byl mezi nimi volný prostor),
- používání předalokace (uživatel si požádá FS o vymezení určitého prostoru na disku, např. databáze),
- odložená alokace (= *allocate-on-flush*, FS nezapíše ihned po změně souboru, ale chvíli počká – počítá s tím, že uživatel bude chtít měnit soubor „za chvíli“ znovu – až nebude delší dobu docházet ke změnám, poté hledá vhodný volný prostor)

Při (intenzivním) běžném používání disku se však fragmentaci nelze vyhnout. Pokud by byla fragmentace příliš výrazná, je možné použít defragmentační nástroje, které provádějí kopírování, přesouvání částí souboru a reorganizaci diskového prostoru tak, aby se fragmentace odstranila – časově náročná operace.

Prvního negativního dopadu externí fragmentace (nevyužitelné a příliš malé oblasti) je možné se zbavit při používání alokace po jednotkách pevné velikosti – alokační bloky – vždy je ale snaha alokovat spojité (v horším případě se alokuje nespojitě, pokud to nejde).

### 5.8.2 Interní fragmentace

Nespojitá alokace po jednotkách pevné velikosti (alokační bloky) má výhodu, že redukuje dopady externí fragmentace, ale potom vytváří interní fragmentaci. Interní fragmentace se obvykle toleruje.

**Příklad interní fragmentace:**

- chci alokovat soubor o velikosti 9 000 B,
- mám 4 KiB velké alokační bloky,
- potom je nutné alokovat 12 KiB pro tento soubor,
- ty zbývající 3 KiB v posledním alokačním bloku zůstanou nevyužité.

Existuje několik málo souborových systémů, které se snaží řešit interní fragmentaci (ReiserFS, ZFS) pomocí techniky zvané *tail packing* („zbavování ocásku souborů“) – více souborů může používat 1 fyzický alokační blok (zaplní se volné místo). Většina souborových systémů toto však nepodporuje.

## 5.9 Přístup na disk

Když chce proces načítat, zpracovávat data (viz 2.5.4):

- zavolá službu k tomu určenou (read, write, . . . – může být zabaleno i v nějakém knihovním volání, např. scanf zavolá read),
- dojde k předání řízení jádru, dostane se ke slovu jádro,
- podívá se do cache, pokud tam ta data má, předá je, pokud ne, musí je načíst z disku
- s diskem komunikuje přes I/O porty nebo paměťově mapované I/O porty,
- disku se předávají příkazy přes jeho rozhraní (ATA disk – ATA příkazy), jdou z FS přes ovladač příslušného disku (poté to prochází sběrnicemi), ten komunikuje s řadičem disku,
- disk dostane příkaz, jakmile doběhne operace, disk pošle přerušení na procesor,
- tam se dostává ke slovu jádro, to zpracuje přerušení a zachová se podle něj (úspěch – předá data, chyba – zpracuje ji)

**Definice:**

***ovladač*** je software, který umí komunikovat s určitým typem zařízení, jiná definice viz 2.4

## 5.10 Plánování přístupů na disk

Součástí jádra je subsystém nazývaný *plánovač diskových operací*, který shromažďuje požadavky od FS (načtení, zapsání dat z/do disku). Plánovač si ukládá požadavky do svých plánovacích front, požadavky případně přeuspořádá a předává dál ovladači či řadiči disku k realizaci.

Plánovač se snaží minimalizovat režii disku.

Jednou ze strategií přeuspořádávání požadavků (u HDD) je použití **výtahového algoritmu (elevator, SCAN alghorithm):**

- snaha, aby se hlavička disku plynule pohybovala od středu k okraji a zpět a vyřizovat požadavky dle pohybu hlavičky,
- modifikace SCAN algoritmu je například Circual SCAN, kdy se požadavky vyřizují pouze při jednom směru,
- další modifikace jsou LOOK a C-LOOK, kde se hlavička nepohybuje od středu k okraji, ale pouze v tom rozsahu, kde je potřeba provádět operace.

**Plánovač se může snažit více operací sloučit do jedné operace** (např. operace v rámci jednoho bloku se sdruží):

- takové kroky mají význam i u SSD,
- snaha vyvažovat požadavky jdoucí od jednotlivých uživatelů (procesů),
- implementace priorit (prioritnější proces – požadavky se vykonají dříve),
- snaha odkládat operace tak (v naději), že je bude poté možné sloučit,
- snaha implementovat časová omezení na dobu čekání požadavků,
- může implementovat paralelizaci požadavků předávaných do diskového subsystému (modernější a velmi výkonné SSD umí řešit operace paralelně).

**Linux:**

Pro zjištění, jaký plánovač právě používáme, se stačí podívat do */sys/block/<devname>/queue/scheduler*

## 5.11 Logický disk

V počítači je možné mít vícero fyzických disků, které je dále možné rozdělit na logické disky, a konkrétní souborové systémy je možné instalovat na logické disky. Pro správu a vytváření logických disků lze použít programy cfdisk, disk, gparted, …

### 5.11.1 Způsob uložení informací o diskových oblastech na disku

- MBR
  - v prvním (nultém) sektoru byla tabulka obsahující rozdělení na 1–4 primární oddíly
  - pokud bylo nutné použít více oddílů, potom se primární nahradila rozšířenou diskovou oblastí, která se dále mohla rozdělit na podoblasti zvané logické diskové oblasti, každá z nich popsána formou zřetězeného seznamu, EBR
  - používané u starších PC
- GPT
  - je tabulka (pole) o až 128 odkazech na jednotlivé diskové oblasti,

  - stejný vyhrazený prostor jako u MBR

### 5.11.2 LVM

- správce logických oblastí,
- umožňuje pokročilejší tvorbu logických disků a
- do logického disku přidávat fyzické disky (za běhu),
- LVM může být buď přímo ve FS nebo v části jádra (mezi FS a plánovačem).

### 5.11.3 Různé typy souborových systémů

- fs (první fs na UNIXu), ufs, ufs2,
- ext2, ext3, ext4,
- btrfs (inspirován ZFS),
- ReiserFS, HSF+/APFS (Mac OS X), XFS, JFS, HPFS,
- FAT, VFAT, FAT32, exFAT (rodina FAT vznikla v MS-DOS, pote používaný ve Windows – velmi jednoduché a široce podporované),
- F2FS (fs pro efektivní práci se SSD), ISO9660, UDF, Lustre, GPFS (clustery, superpočítače),
- ZoneFS (ZFS).

Po koupi nového disku a rozdělení na logické disky je nutné se rozhodnout, jaký souborový systém na příslušném logickém disku bude používán – je nutné disk **zformátovat** pro použití. Dříve se používalo i nízkoúrovňové formátování (staré disky s nestabilním magnetickým záznamem).

### 5.11.4 Chyby disku (souvislost s FS)

Na disku mohou vznikat chyby běžným opotřebením nebo třeba nevhodným vypnutím napájení, je zapotřebí opravit řídicí struktury souborového systému (program fsck – kontroluje konzistenci FS nebo žurnálování, copy-on-write, soft updates, . . . ).

### 5.11.5 Další typy souborových systémů

**Virtuální souborový systém (VFS)** je vrstva, která v jádře zastřešuje všechny ostatní souborové systémy z toho důvodu, aby jiné subsystémy jádra nemusely pracovat speciálním způsobem s různými souborovými systémy (více v 8.2).

Existují také různé síťové souborové systémy, třeba NFS (viz také 8.3).

**Speciální souborové systémy**

- neukládají žádná data, obsah není nikde na disku ani neexistuje žádná speciální část paměti
- zpřístupňují např. aktuální stav jádra – adresář `/sys` (*sysfs filesystem*),
- *procfs filesystem* v adresáři `/proc` zpřístupňuje informace o běžících procesech (a částečně o stavu jádra),
- *tmpfs* vytváří souborový systém v RAM.

## Definice (pro 5.11.x):

***logický disk*** je taky disková oblast, partition
***MBR*** = master boot record
***EBR*** = extended boot record
***GPT*** = GUID Partition Table, GUID = Globally Unique Identifier

***LVM*** = Logical Volume Manager

***formátování*** znamená, že se nainstalují metadata (řídicí data) souborového systému do příslušné diskové oblasti, v rámci toho se mohou vymazat všechna data na dané oblasti

# 6

**Šestá přednáška:** pokračování správy souborů. Žurnálování, jeho implementace a alternativy, copy-on-write, klasický UNIXový systém souborů, i-uzly, kde a jak jsou data uložena, počty odkazů, limit maximální velikosti souboru, výhody a nevýhody FS, jiné způsoby organizace souborů, ext4, NTFS, organizace volného prostoru na disku, deduplikace, typy souborů v UNIXu, adresář, montování disku.

## 6.1 Žurnálování

Žurnálování je technika založená na vytváření žurnálu.

- souborové systémy se žurnálem jsou třeba ext3, ext4, ufs, XFS, JFS, NTFS, ...
- žurnálování umožňuje *spolehlivější* (nikdy nemáme obecně zajištěno, že se *nic* špatného stát nemůže) a rychlejší návrat (než nějaké utility) do konzistentního stavu po chybách
- data obvykle žurnálována nejsou (velká režie), ale mohou být
- závisí na tom, že operace, které žurnálování implementují, se provedou ve správném pořadí – nutnost spolupráce s plánovačem, také disky si samy data přeuspořádávají (nelze nijak ovlivnit)

**Žurnál:**

- zápis žurnálu je předřazený,
- vytváří se v něm cyklicky přepisovaný buffer,
- předřazenost zápisu do žurnálu mi zaručí, že operace pokryté žurnálováním jsou atomické – vytváří transakce

Kompromis mezi žurnálováním a nežurnálováním dat je **předřazení zápisu dat na disk před zápisem metadat do žurnálu** (a následně zápis ostrých metadat na disk). Příklad:

- zapisují do souboru – buď vytvářím zcela nový, nebo ho zvětšují (typické způsoby zápisu),
- při zvětšování se nejprve zapíšou data na disk za existující data (bez poznamenávání informace o tom, že se soubor zvětšuje),
- pokud operace selže, soubor zůstane v původním stavu (díky neupraveným metadatům původního souboru),
- teprve až data budou na disku, tak se do žurnálů zapíše informace o zvětšování souboru,
- poté se změní metadata souboru (a uživatel se k datům dostane),
- při selhání napájení v moment, kdy jsou metadata v žurnálu, ale ne na disku, je možné tato metadata obnovit

**Proces mazání souboru na disku:**

- odstranění záznamu z adresáře,
- uvolnění uzlu (metadat souboru),
- uvolnění oblastí použitých souborem

### Definice:

***žurnál*** je speciální soubor či speciální oblast na disku sloužící pro záznamy modifikovaných metadat (dat o datech), případně i dat před jejich zápisem na disk (v podobě běžných dat)

***předřazení*** znamená, že zápis do žurnálu se provede před ostrým zápisem „užitečných“ dat (či metadat) na disk

***atomická operace*** – buď operace uspěje celá (všechny dílčí kroky), nebo neuspěje vůbec (žádný dílčí krok)

### 6.1.1 Implementace žurnálování

Existují 2 základní přístupy k implementaci žurnálování. **REDO:**

- implementace na základě dokončení transakcí,
- používá např. ext3, ext4,
- sekvence dílčích operací (vytvářející tu operaci, kterou chci provést) se zapíše do žurnálu (začátek, konec transakce, kontrolní součet),
- poté se operace provádí na disku,
- po úspěšném dokončení se transakce ze žurnálu uvolní,
- při selhání a poté zotavení se systém podívá do žurnálu, podívá se po neuvolněných transakcích, jestli jsou celé – počáteční a koncová značka, jestli sedí kontrolní součet – pokud vše sedí, systém provede všechny operace znovu

**UNDO:**

- implementace na základě anulace transakcí,
- v kombinaci s REDO se používá v NTFS,
- prokládá záznam dílčích operací (které se mají provést) do žurnálu a následně jejich provedení na ostrých datech (zaznamená dílčí operaci – provede ji),
- proběhne celá transakce – záznam ze žurnálu se uvolní,
- při chybě se eliminují všechny nedokončené transakce (všechny provedené dílčí kroky se musí vrátit – vrátí se disk do původního stavu)

Při implementaci žurnálování je klíčové **dodržení pořadí kroků, ve kterém se provádějí**. (U REDO je nutné, aby se nejprve zapsaly sekvence operací do žurnálu a teprve poté se prováděly operace na disku.) Pokud tato sekvence nebude dodržena, žurnál nebude správně fungovat.

### 6.1.2 Copy-on-write

COW je alternativou k žurnálování používanou například v ZFS (OpenZFS), BTRFS, ReFS (Resilient File System).

- kopie při zápisu,
- založeno na tom, že všechna nová data/metadata se zapíší na disk a poté se zpřístupní,
- využívá se přitom toho, že obsah disku je popsán hierarchickou stromovou strukturou,
- změny se provádějí v souladu s touto strukturou (od listů ke kořeni),
- pokud vypadne napájení v moment, kdy data (bloky) nejsou ještě zpřístupněna, data nejsou dostupná z kořene stromu a jakoby se nic nestalo,
- pokud se mi tyto data podaří zapsat úspěšně, postupně začnu upravovat všechny uzly vedoucí až ke kořeni a zpřístupním nová data,
- teprve po modifikaci kořenů se stanou změněná data (uzly) dostupné
  - kořen je nutné zabezpečit, aby nedošlo k chybě při zápisu do něj
  - starý kořenový uzel se nepřepisuje, ale pouze se tam zapíše nová verze kořenového záznamu (s časovým razítkem)
  - současně tam bude zabezpečovací kód (kontrolní součet),
  - pokud dojde ke krachu systému, stačí si načíst všechny kořeny, zkontrolovat kontrolní součty, vybrat si všechny, kde sedí kontrolní součty, s nejnovějším časovým razítkem a tyto použít (pokud dojde k chybě, než se stihne zapsat nový kořen, použije se ten původní)

**Výhodami copy-on-write jsou:**

- snímky souborového systému (zapamatuje se pouze kořenový uzel – minimální režie),
- klony souborového systému (vytvoří se požadovaný počet kopií kořenového uzlu),
- výhodou je, že nezměněné uzly (data) a listy budou na disku pouze jednou, pouze je nutné si pamatovat změněné uzly/listy a cestu ke kořeni + kořen (kopie stále ukazují na stejný strom).

## Definice:

***stromová struktura (copy-on-write)*** je vyhledávací strom, který popisuje veškerý obsah disku, typicky se v něm vyhledává na základě unikátní identifikace souborů

***adresáře (copy-on-write)*** jsou speciální soubory uloženy na disku, které jsou dostupné ve stromě (stromové struktuře)

***snímek souborového systému*** – uloží se obsah disku tak, že je možné se k němu později vrátit

***klon souborového systému*** je vytvoření 2 kopii souborového systému a od daného okamžiku je možné s každou kopií pracovat samostatně (např. při větším počtu VM, kdy všechny VM sdílí stejný počáteční obsah disku, ale od určitého momentu každá VM chce obsah měnit samostatně)

### 6.1.3 Další alternativy žurnálování

**Soft updates:**

- používá se v UFS (FreeBSD systémy),
- FS se snaží sledovat závislosti mezi tím, jaká data a metadata se mění,
- uzpůsobuje pořadí zápisu metadat a dat na disk tak, aby v jakémkoli okamžiku byl obsah na disku konzistentní (až na možnost vzniku „garbage“)

**Log-structured file systems:**

- logovací souborové systémy (= strukturované jako log),
- používá se v LFS, UDF, F2FS,
- celý souborový systém má charakter jednoho velkého logu,
- který se zapisuje v cyklicky přepisované paměti napříč celým diskem,
- poslední obsah disku je vždy dostupný přes poslední záznam (a odkazy, které z něj vedou),
- při provádění změn se přidají data např. za aktuální konec využitého diskového prostoru, přidá se k tomu záznam (o tom co se změnilo), zpřístupní se data z posledního záznamu.

## Definice:

***garbage*** je část prostoru na disku, která se tváří jako obsazená, ale není

***logem*** rozumíme soubor, který obsahuje záznamy o změnách (log = zápis o změnách)

## 6.2 Klasický UNIXový systém souborů (FS)

Je původní souborový systém UNIXu (70. léta). Vyvinul se z něj UFS, z něj zase ext2, ext3 (poté vznikl i ext4, který už je odlišnější).

**Souborový systém byl rozčleněn (na úrovni logických disků) na:**

- boot blok – obsahoval informace (kód, část kódu) potřebné pro zavedení při startu,
- super blok – informace o souborovém systému (typ, verze, velikost, počet i-uzlů, volné místo, kořenový adresář, volné i-uzly, . . . ),
- tabulka i-uzlu – tabulka (pole n i-uzlů) s popisy souborů,
- datové bloky – data souboru, metadata (pomocné adresovací bloky).

**Základní rozložení FS bylo zmodifikováno v navazujících FS:**

- datové bloky byly rozděleny do skupin,
- každá skupina měla svoje i-uzly,
- důvodem byla lepší lokalita, prostorová blízkost dat a metadat (typicky při práci se soubory jsou nutná i jeho metadata),
- poté tedy ta struktura vypadala takto: boot blok, super blok, úsek i-uzlu, úsek dat, úsek i-uzlu, úsek dat, . . .

### 6.2.1 i-uzel

I-uzel je základní datová struktura reprezentující každý jeden soubor v typických UNIXových systémech. (Při formátování se určí dopředu maximální počet souborů, které na diskovém oddílu budou existovat.) Ke každému souboru musí existovat i-uzel. Ten obsahuje metadata o souboru:

- stav i-uzlu (alokovaný, volný)
- typ souboru (obyčejný, adresář, zařízení, pojmenovaná roura, . . . ),
- délka souboru v bajtech,
- časy **mtime** (poslední modifikace dat – zápis), **atime** (poslední přístup – čtení), **ctime** (poslední modifikace i-uzlu),
- UID, GID,
- přístupová práva (číslo, např. 0644 = rw-r–r–),
- počet pevných odkazů (neboli jmen souborů),
- informace o tom, kde se nachází data o souboru (tabulka odkazů na datové bloky a další informace nebo odkazy na pomocné bloky s dalšími metadaty, např. ACL, extended attributes, dtime – údaj o smazání souboru, . . . )

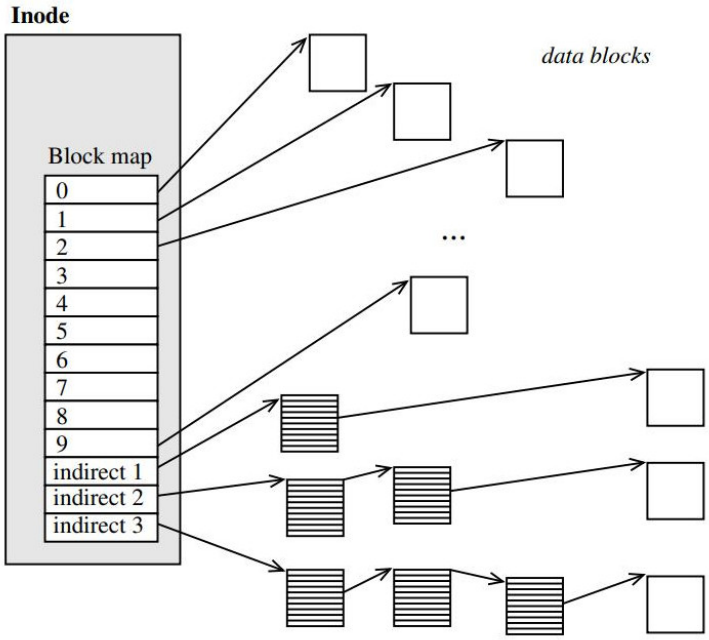
Jméno souboru **není** v i-uzlu, ale je uloženo v adresáři.

### Definice:

***UID*** je číslo identifikace vlastníka
***GIT*** je číslo identifikace skupiny
***ACL*** – Access Control List, přístupové seznamy rozšiřující základní UNIXová práva tak, že je možné přiřadit konkrétní práva ke konkrétním uživatelům
***extended attributes*** – rozšířené atributy, s jakými specifickými právy se může soubor např. spouštět
***ctime*** je poslední změna i-uzlu, využitelné např. pokud se zfalšuje mtime, poslední změna souboru se pozná právě podle ctime

### 6.2.2 Kde a jak jsou uložena data

- v i-uzlu je řada přímých (až 10, novější FS mají 12) i nepřímých odkazů na data,
- přímé odkazy odkazují na alokační bloky na disku,
- pokud je potřeba více odkazů, použije se nepřímý odkaz první úrovně, který odkazuje na speciální alokační blok neobsahující data, ale další přímé odkazy na data,
- pokud nestačí ani to, použije se nepřímý odkaz druhé úrovně, který odkazuje na pomocný adresovací blok, který obsahuje další nepřímé odkazy 1. úrovně, které odkazují na další přímé odkazy (a ty odkazují na data) – vzniká strom;
- pokud ani to stačit nebude, použije se adresovací blok 3. úrovně, vedoucí na adresovací bloky s nepřímými odkazy 2. úrovně, každý z nich na bloky s odkazy 1. úrovně, ty vedou na přímé odkazy a ty na data.




Obrázek 1: Z prezentace IOS: Souborové systémy, slide 23 – odkazy v i-uzlech

### 6.2.3 Počty odkazů

- nepřímý odkaz 1. úrovně – při 4 KiB clusteru je to $1024$ odkazů (1 odkaz = 4 B) = 1024 datových bloků,
- nepřímý odkaz 2. úrovně – při 4 KiB clusteru je to $1024^2$ odkazů = stejný počet datových bloků,
- nepřímý odkaz 3. úrovně – při 4 KiB clusteru je tam $1024^3$ odkazů = stejný počet datových bloků

### 6.2.4 Limit maximální velikosti souboru

Počtem přímých odkazů nepřímého odkazu 3. úrovně je dán maximální počet bloků, které je možné v tomto souborovém systému uložit. Teoretický limit velikosti souboru je tak:

$$10 \cdot D + N \cdot D + N^2 \cdot D + N^3 \cdot D$$

kde $D$ je velikost bloku v bajtech (běžně 4096 B), $M$ je velikost odkazů na blok v bajtech (běžně 4 B), $N = D/M$, je počet odkazů v bloku.

Toto omezení velikosti je pouze jedním z omezení, která velikosti souboru omezují. **Další omezení jsou daná:**

- dalšími datovými strukturami a typy, které používá FS (např. datový typ délky souborů v bajtech v i-uzlu),
- strukturami VFS (veškerá práce s jakýmkoli FS musí projít přes VFS),
- rozhraním jádra,
- architekturou systému (32b – velikost souboru bude 32b číslo + MSB je použit pro indikaci chyby [-1 bit pro data] – soubory maximálně do 2 GiB, dnes běžná architektura 64b – 64b velikosti)

Existuje **Large File System Support**, kde ve 32b systému se nahradí všechny údaje kde se pracuje s velikostí větším datovým typem – podpora souborů *větších než 2 GiB*.

**Linux:**

*du [soubor]* vypíše zabrané místo v blocích vč. režie (metadat)

*ls -l [soubor]* vypíše velikost souboru v bajtech (pouze užitečná data)

*df* vypíše volné místo na discích

*ls -i [soubor]* zpřístupní číslo i-uzlu souboru

*is -e /dev/... n* – výpis i-uzlu n na /dev/...

*dumpe2fs* – základní informace o souborovém systému ext2,3,4

*/dev/zero* je soubor typu zařízení generující proud nul

*dd if=[source] of=[dest]* je nízkoúrovňové kopírování

### 6.2.5 Výhody a nevýhody architektury FS

Aneb proč byl navržen FS právě tak, jak byl. Architektura FS je totiž ovlivněna snahou o minimalizaci jejich režie s relativně pomalými disky (HDD, SDD), jedná se zejména o běžné operace se soubory, jako je průchod souborem (otevřu – procházím od začátku do konce) či přesun (seek), zvětšování či zmenšování (vč. mazání) souborů.

**Je nutné vzít do úvahy, z jakých (mikro)operací se tyto operace skládají**. Jsou to operace:

- vyhledávání adresy prvního nebo určitého bloku souboru,
- vyhledávání následujících bloků,
- přidání či odebrání bloků,
- alokace či dealokace volného souboru (informace o volných oblastech, minimalizace externí fragmentace)

FS a jeho následníci UFS, ext2, ext3 (ext4 už není jeho následník!) představují kompromis s ohledem převážně na malé soubory. (Tyto FS fungují skvěle pro malé soubory – u větších souborů je nutné procházet či měnit větší objem metadat.)

Jistou optimalizací používanou i u klasických FS pro malé soubory je uložení dat přímo do i-uzlu (pokud se tam data vlezou).

### Definice:

***symbolický odkaz*** je soubor odkazující na jiný soubor (používá uložení dat přímo do i-uzlu)
***rychlé symlinky*** mají data v i-uzlu
***pomalé symlinky*** mají data mimo i-uzel

## 6.3 Jiné způsoby organizace souborů

### 6.3.1 Kontinuální uložení

- neboli spojité uložení souboru na disku,
- na disku je jeden spojitý úsek dat reprezentující soubor,
- výhodami jsou rychlé vyhledání adresy určitého bloku nebo vyhledávání následujících bloků,
- nevýhody: soubory nebude možné jednoduše zvětšovat pokud budou příliš blízko u sebe (bude nutné jej přesunout na jiné volné a větší místo, pokud to půjde, či provést defragmentaci a poté zvětšit soubor)
- nepoužívá se příliš (kvůli své nevýhodě)

### 6.3.2 Zřetězené seznamy alokačních bloků

- každý alokační blok obsahuje svá (užitečná) data a na konci obsahuje odkaz na následující alokační blok,
- výhodami jsou rychlý přístup na začátek či průchod daty,
- nevýhodou je přesun na náhodné místo v souboru – nutnost přečíst celý soubor až po daný blok (1 GiB soubor, chci poslední blok – musím přečíst celý),
- další nevýhodou je rozprostření metadat po celém disku – při drobné chybě na disku přijdu o data (tedy i metadata, kde jsou odkazy na následující bloky) a dojde k velké ztrátě dat (všechna data za ztracenými daty jsou nepřístupná),
- není příliš vhodný, ale je jednoduchý, používá se v souborových systémech FAT

### 6.3.3 FAT

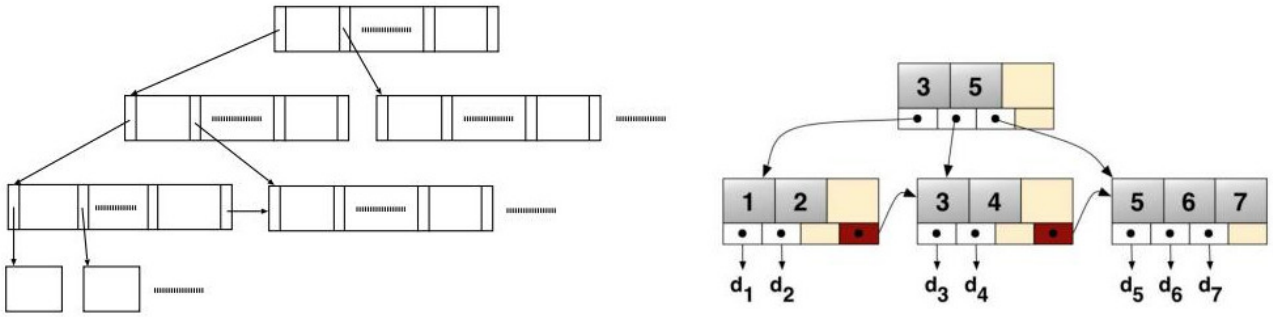
- File Allocation Table,
- od zřetězených seznamů se liší tím, že seznamy popisující rozložení souborů na disku jsou uloženy v separátní oblasti na disku (tzv. FAT),
- data jsou ve FAT koncentrovaná – rychleji se prohledávají, lze vytvořit více kopií FAT (prevence okamžité ztráty dat při chybě),
- stále vznikají problémy s rychlostí při náhodném přístupu (stále jde o zřetězený seznam),
- tabulka je pole, které obsahuje pro každý blok na disku 1 položku, každá položka obsahuje odkaz na další blok/položku,
- používá se i dodnes (a je to velmi rozšířené), protože je to jednoduché (např. vestavěné systémy)

### 6.3.4 B+ stromy

- jsou datovou strukturou převzatou z databázových systémů,
- mají dva typy uzlů – vnitřní a listové,
- vnitřní uzly jsou kořen, jeho následníci (kromě listových) obsahují odkaz na následníka a vyhledávací klíč,
- listové uzly také obsahují odkazy a vyhledávací klíče, odkazy vedou na data na disku, poslední odkaz na posledním listu odkazuje na list na stejné úrovni (jsou tak propojeny lineárním seznamem),
- používají se za účelem popisu rozložení dat na disku (obsah souboru, poté vyhledávací klíč bude offset – číslo logického bloku v rámci souboru) nebo se používají pro adresáře (klíče budou jména souborů) nebo pro popis celého obsahu disku (klíč je dvojice i-uzel a posuv souboru)

**Vyhledávání v B+ stromu:**

- při hledání klíče $k$ se podívám, zda je klíč menší jak klíč $k_0$, pokud ano, půjdu níže, kde je $k_0$, pokud ne, zjistím, jestli je klíč mezi $k_0$ a $k_1$, pokud ano, jdu druhým směrem, opakuji po $k_n$,
- pokud jsem níž, opakuji to samé, co výš, dokud nedojdu k listovým uzlům,
- zde hledaný klíč najdu nebo zjistím že v této struktuře klíč není,
- poté mám odkaz na datový blok,
- v případě, že chci číst dál, jdu lineárně po sobě po následujících listových uzlech




Obrázek 2: Z prezentace IOS: Souborové systémy, slide 27 – B+ strom

**Práce s B+ stromy:**

- jsou zde limity, jak moc/málo mají být uzly zaplněné (strom se udržuje vyvážený) – pro uzly s $m$ odkazy máme klíče $0, 1, \ldots$ až $m - 2$ klíčů (odkazů je o 1 méně než klíčů + číslování od 0),
- pokud je strom tvořen jediným kořenem – nejméně může mít 1 odkaz, maximálně $m - 1$ odkazů (poslední odkaz je použit jako ukončovač seznamu listů),
- pokud to není jediný kořen, tak má nějaké následníky, minimálně jich tak má 2, maximálně $m$,
- vnitřní uzel tak má $\lceil \frac{m}{2} \rceil$ (zaokr. nahoru) až $m$ odkazů, list $\lceil \frac{m}{2} \rceil - 1$ (opět $\frac{m}{2}$ zaokr. nahoru) až $m - 1$ odkazů,
- vložení:
  - nejprve projdeme stromem od kořene k listům,
  - najdeme, kam chceme vložit,
  - podíváme se, zda má list volný odkaz,
  - pokud ano, použijeme ho, pokud ne, list se rozštěpí na 2 poloviny a podívám se o úroveň výš, zda je možné namísto 1 listu linkovat 2 listy,
  - pokud ano, přidá se odkaz, pokud ne, nadřazený uzel se musí rozštěpit a postupovat o úroveň výš,
  - … štěpí se strom až případně se rozštěpí kořen a strom bude mít 2 kořeny
- rušení:
  - jde se opět od listové úrovně, tak, že se zruší odkaz v listu,
  - zkontroluje se, zda je uzel zaplněný v rámci daných limitů,
  - pokud ano, good, pokud ne, podívám se na sousední uzly a pokusím se provést přerozdělení tak, aby byly všechny uzly naplněný v rámci limitů,
  - pokud se to nepodaří, tak dojde ke sloučení listů,
  - posunu se o úroveň výš, zruším jeden odkaz, zkontrolují opět limity, zopakuji to samé,
  - … až se může stát, že se zruší i kořen

B+ stromy a jeho varianty jsou používány pro popis diskového prostoru v souborových systémech jako XFS, JFS, ZFS, Btrfs, ReFS, …, v omezené podobě tzv. stromu extentů v ext4, podobna struktura je i v NTFS.

### 6.3.5 Extent

- používá se ke zrychlení práce s velkými soubory,
- umožňují zmenšit objem metadat (je možné říct, že některé alokační bloky jsou uloženy pospolu = vytváří extent), potom budu popisovat rozložení souborů po extentech (ne po alokačních blocích),
- přinese lépe vyvážené indexové struktury,
- rychlejší mazání,
- jsou použity snad ve všech systémech s B+ stromy,
- B+ strom se snadno kombinuje s extenty
  - neplatí pro klasický UNIXový strom, protože ve stromu jsou explicitně uložené vyhledávací klíče, ale UNIXový nemá v žádných strukturách (i-uzlech) uloženou velikost datových bloků (protože jsou všechny stejné a konstantní)

Pokud používáme B+ stromy, tak rychlému spojitému průchodu pomáhá listová úroveň, pokud je prolinkování listů použito. Pro malé soubory může představovat B+ strom zbytečnou režii (data se buď uloží přímo v i-uzlu nebo nebo z něj máme přímé odkazy na extenty z i-uzlu, do max. 4 extentů).

## Definice:

***extent*** je jednotka vystavená na bloky, posloupnost proměnného počtu alokačních bloků (jdoucích za sebou logicky v souboru i fyzicky na disku).

## 6.4 ext4

Používá pro popis rozložení dat na disku strom extentů. Pro „malé soubory“ (myšleno soubory, na které je možné se odkazovat až 4 extenty, pak tyto extenty budou odkazované přímo z i-uzlu; TLDR soubory s malým počtem extentů).

### Definice:

***strom extentů*** je v principu B+ strom degradovaný na maximálně 5 úrovní, bez používání vyvažování (např. přerozdělování uzlů při mazání) a zřetězení listu

## 6.5 NTFS

Základní datovou strukturou popisující disk je MFT – Master File Table (má pro každý soubor alespoň 1 řádek, na 0. řádku popisuje samo sebe, 1. řádek případně kopie MFT, případně metadata, poté obsahy souborů).

**Obsah souboru může být reprezentován buď:**

- pokud jde o krátký soubor, bude uložen v MFT v jeho řádku (včetně metadat),
- soubor je rozdělen na extenty (ty jsou odkazované přímo z řádků souboru v MFT, tak, že v řádce souboru jsou informace o počátečním VCN a LCN a počet clusterů, který daný extent obsahuje) – vyhledává se v tom stejně jako v B+ stromu
- pokud je extentu potřeba více, než se vleze na jeden řádek, alokují se pomocné řádky (z hlavního řádku vedou odkazy na pomocné, z pomocných vedou odkazy na disk) – procházení je opět ve stylu B+ stromu

### Definice:

***VCN*** – virtual cluster number, logický blok souboru
***LCN*** – logical cluster number, číslo fyzického bloku (souvisí s tím že je to na logickém disku)

## 6.6 Organizace volného prostoru na disku

V klasickém UNIXovém FS a řadě jeho následovníků (UFS, ext2, ext3), také v NTFS se používají bitové mapy, kde pro každý blok mám 1 bit. V bitové mapě je možné poté vyhledávat pomocí bitových mask – zrychlí vyhledávání.

**Další možné způsoby organizace volného prostoru:**

- použijí se alokační seznamy (zřetězení volných bloků na disku),
- zřetězení volných položek v tabulce (FAT),
- B+ strom (udržování informací o tom, kde je volné místo, adresace velikosti a/nebo offsetem)
- volný prostor může být také organizován po extentech

## 6.7 Deduplikace

- podporovaná ZFS, NTFS, Btrfs, XFS, ... (ext4 ne)
- snaží se odhalit opakované ukládání stejných dat na disk, uložit je pouze jednou a poté se na ně vícenásobně odkazovat,
- systémy s deduplikací se snaží taková data detekovat (sekvence bitů, bloků, extentů, ... )
- založeno na kryptografickém hashování (hledají se data se stejným popisem, určí se shoda),
- může být realizováno při zápisu nebo dodatečně (žádost uživatele),
- může uspořit diskový prostor (při virtualizaci, na mail serverech, repozitáře, ... ), paměťový prostor i čas (zamezí se opakovanému čtení i zápisu),
- při menším objemu duplikace může naopak zvýšit spotřebu CPU času, spotřebu paměťového i diskového prostoru

**Rozdíl oproti copy-on-write:**

- copy-on-write se může uchytit (klony, snímky) pouze tehdy, pokud duplikáty vzniknou činností samotného FS (např. vytvoření virtuálek),
- zatímco deduplikace aktivně vyhledává duplikáty (např. uživatel, co stahuje stejné reklamní letáky)

## 6.8 Typy souborů v UNIXu

- - je obyčejný soubor,
- d adresář,
- b blokový speciální soubor,
- c znakový speciální soubor,
- l symbolický odkaz (symlink),
- p pojmenovaná roura,
- s socket

### Definice:

***speciální soubor*** je typ souboru reprezentující hardwarové zařízení (disk, paměť, ... ), se kterým se komunikuje po blocích nebo znacích

***symbolický odkaz*** je soubor obsahující jméno jiného souboru (odkazuje na něj)

## 6.9 Adresář

Adresář je kolekce jiných souboru na nejvyšší úrovni abstrakce. Soubor obsahuje množinu dvojic jméno a unikátní číselné označení.

**Jméno souboru:**

- dříve limit 14 znaků, dnes až 255 (na konci musí být `'\0'`)
- ve jméně nesmí být `/` nebo `'\0'`
- platí, že každý adresář v POSIX systému vždy obsahuje minimálně 2 jména: `.` (odkaz na sebe) a `..` (odkaz na rodičovský adresář)

**Číslo souboru:**

- u klasického souboru UNIXu je to číslo i-uzlu,
- v jiných případech to slouží jako klíč do dané vyhledávací struktury (B+ strom)

**Implementace adresářů:**

- používají se různé přístupy, liší se jednoduchostí implementace či rychlostí vyhledávání/vkládání,
- seznam (obsah souboru bude tvořen seznamem),
- B+ stromy (v NTFS, XFS, JFS, APFS nebo ext3/4 – ty používají H-stromy: 1–2 úrovně, bez vyvažování a vyhledává se na základě zahashovaného jména)
- hashovací tabulky v např. ZFS

Soubor v UNIXu může mít více jmen. Další jména se vytváří pomocí příkazu `ln`.

**Linux:** `ln [existující jméno] [nové jméno]` vytvoří další jméno souboru

## 6.10 Montování disku

**Princip montování (připojování) disku:**

- v UNIXu není žádné označení disku (A:, C:, ...), ale máme jeden adresářový „strom“,
- v systému je jeden kořenový logický disk,
- další logické disky se připojují programem `mount` do existujícího adresářového stromu (kořenový adresář zařízení se „slepí“ s adresářem v mém stromu)

Připojovací volby se mohou zadávat ručně (v terminálu) nebo se mohou předpřipravit do `/etc/fstab`. Soubor `/etc/mtab` obsahuje tabulku aktuálně připojených disků.

**Novější technologie umožňující automatické montování nově připojených zařízení:**

Na Linuxu běžně pracuje systém *udev*, který

- rozpozná, že se připojilo nové zařízení,
- vytvoří odpovídající soubor typu blokové zařízení (`/dev/...`),
- informuje o tom zbytek systému pomocí sběrnice D-Bus,
- aplikace typu správce souborů pak může provést automatické montování (a další akce),
- přednost má vždy `/etc/fstab`,
- identifikace se nemusí provádět jen zařízením (`/dev/...`), je možné si vygenerovat unikátní identifikátor a používat ten (*UUID*)

**Technologie Automounter:**

- subsystém jádra,
- připojuje automaticky potřebné disky v situaci, kdy se pokusíme přistoupit na pozici adresářového stromu, kam by takovýto disk měl být připojeny (např. na `/mnt` má být připojena flashka, nemusí být mounted, uživatel dá `cd /mnt`, automounter to zjistí a připojí flashku sem),
- má také nějaký čas, po kterém disk automaticky odpojí, pokud se ním nepracuje

**Union mount:**

- technologie umožňující sjednocující montování (v UNIXu dostupná pomocí souborového systému UnionFS),
- umožňuje do jednoho přípojného bodu namontovat více disků,
- obsah přípojného bodu je sjednocením obsahů disků,
- v případě, že na více discích jsou soubory se stejnými jmény, vznikají kolize, ty se řeší např. předdefinováním priorit připojovaných FS a zpřístupní se soubor daného jména z logického disku, který má největší prioritu

- UnionFS má copy-on-write sémantiku, což umožňuje emulaci přepisování nepřepisovatelných médií (v jedné větvi nepřepisovatelné CD, například s Linuxovou distribucí, současně se do stejného bodu připojí běžný disk s vyšší prioritou – na začátku bude disk prázdný, budou vidět všechny soubory z CD, jakmile se pokusím přepsat něco, UnionFS vytvoří kopii na přepisovatelný disk)

**Linux:** `mount [co-pripojit] [kam-pripojit]` připojí logický disk

# 7

**Sedmá přednáška:** Pokračování správy souborů, symbolické odkazy.

## 7.1 Symbolické odkazy

- symbolické odkazy jsou samostatné soubory odkazující na existující soubor,
- systém při otevření automaticky otevře cílový soubor – vícenásobné zpracování cesty (cesta k symlinku a cesta uvnitř něj),
- soubor se smaže, pokud jeho počet jmen klesne na 0,
- symlink může odkazovat na neexistující soubor (při otevření dojde k chybě),
- může odkazovat i na jiný logický disk,
- ze symlinků lze vytvořit cyklus (jeden odkazuje na druhý a druhý na první) – v systému je předdefinovaný maximální počet na sebe odkazujících symlinků (při překročení dojde k chybě),
- symlinky lze využít např. při upgradu systému.

**Rozdíl rychlých a pomalých symlinků:**

- obsahem symlinku je jméno cílového souboru,
- pokud jméno souboru není příliš dlouhé (vleze se do i-uzlu), potom se uloží do i-uzlu = rychlý symlink (stačí otevřít jen i-uzel),
- pokud se jméno nevleze do i-uzlu, alokují se normálně alokační bloky na disku = pomalý symlink

**Linux:** `ln -s [existující soubor] [symbolický odkaz]` vytvoří symbolický odkaz

## 7.2 Blokové a znakové speciální soubory

Soubory reprezentující rozhraní souborového systému k fyzickým (opravdový HW) či virtuálním zařízením (terminály, . . . ). Souborový systém vytváří souborové rozhraní, tím umožňuje tyto soubory při určitých operacích identifikovat (jméno souborů, např. `/dev/sdX`), s celým zařízením lze také pracovat jako se souborem.

Typicky zařízení sídlí v adresáři `/dev`.

**Běžné typy zařízení:**

- /dev/hda – (dříve) označení pro první *fyzický* disk na prvním ATA/PATA rozhraní
- /dev/hda1 – (dříve) první *logický* disk na hda
- /dev/sda – první fyzický disk SCSI, navíc i disky SATA/PATA (jádro nad těmito disky emuluje SCSI)
- /dev/mem – obsah paměti (RAM)
- /dev/zero – nekonečný zdroj nul (bajtů 0)
- /dev/null – soubor typu černá díra – cokoli se do něj zapíše, zahodí se (přesměrování výstupů programů tak, aby nás neotravoval), při čtení se tváří jako prázdný soubor
- /dev/random, /dev/urandom – generátor (pseudo)náhodných čísel
- /dev/tty – terminál
- /dev/lp0 – tiskárny
- /dev/mouse – myš
- /dev/dsp – zvuková karta
- /dev/loop – zařízení typu smyčka, umožňuje připojit **soubor jako disk** (obraz souborového systému) k adresáři, jako by se jednalo o nový fyzický disk

Tato označení závisí na použitém systému (Linux, distribuce, . . . ). Výhoda zavedení speciálních souborů je, že umožňují identifikovat zařízení, se kterými chci pracovat.

## 7.3 Přístupová práva

V UNIXu jsou typicky rozlišena na práva pro *vlastníka*, *skupinu vlastníků* a *ostatní*. Existuje rozšíření *ACL* (Access Control List).

**Uživatelé:**

- jsou definováni administrátorem systému (root) v `/etc/passwd`,
- mají definovaná svá UID – uživatelská čísla (root UID = 0),
- každý soubor má svého vlastníka,
- `chown` – změna vlastníka souboru (pouze root)

**Skupiny:**

- definuje administrátor systému v `/etc/group`,
- mají svá GID – číslo identifikující skupinu uživatelů,
- v každé skupině je uvedeno, kdo do té skupiny patří,
- každý uživatel může být členem více skupin,
- jedna z nich je aktuální (používá se při vytváření souboru)

**Linux:**

`groups` – výpis skupin uživatele

`chgrp` – změna skupiny souboru

`newgrp` – nový shell s jiným aktuálním GID

## 7.4 Typy přístupových práv

**Obyčejné soubory:** r, w, x – právo číst, zapisovat a spustit soubor jako program.

**Adresáře:**

- r – právo číst obsah adresáře,
- w – právo zapisovat (vytvářet a rušit soubory),
- x – právo přistupovat k souborům v adresáři (možnost provést např. `cd [adresář]`, ...)

**Typický výstup přístupových práv je:**

- ve formátu: [1:typ souboru] [3:práva vlastníka] [3:práva skupiny] [3:práva ostatních]
- např.: -rwx—r– (číselné vyjádření v osmičkové soustavě: 0704) je obyčejný soubor, vlastník má všechna práva, skupina žádná a ostatní mají práva na čtení

Změna přístupových práv se děje pomocí `chmod`. (Pro nespustitelné soubory je běžný chmod 0644).

**Linux:** `chmod [1:pro koho][nová práva] [soubor/y]` změna přístupových práv

## 7.5 Sticky bit

Příznak, který pokud bude přiřazen nějakému adresáři, tak se vytvoří adresář, ve kterém i přes právo čtení a zápisu souborů mohou uživatele rušit pouze ty soubory, které sami vytvořili. Typickým příkladem je adresář `/tmp`. (TLDR: uživatel může mazat, ale pouze to, co vlastní).

## 7.6 (S|E)UID, (S|E)GID

**S procesy jsou spojeny různé identifikátory, např.:**

- UID – reálná identifikace uživatele (číslo uživatele, který daný proces spustil)
- EUID – efektivní identifikátor používaný pro kontrolu přístupových práv (většinou stejné jako UID)
- GID – reálná identifikace skupiny (kdo spustil proces)
- EGID – efektivní GID (stejné chování jako u EUID)

Vlastník programu může propůjčit svá práva komukoliv, kdo spustí program s nastaveným SUID bitem. (Běžně se to používá v případech, kdy administrátor propůjčuje svá práva uživatelům, např. `passwd` nebo `ping`).

Při použití SUID bitu bude UID = uživatele, který proces spustil a EUID = identifikace vlastníka (= který práva půjčil). Pokud budou práva propůjčená (použito SUID), místo x se vypíše s, pokud tam x není, vypíše se S.

## 7.7 Typická struktura adresářů v UNIXu

**FHS – Filesystem Hierarchy Standard (Linux), (část hierarchie):**

- /bin – programy pro všechny uživatele (spustitelné, mohou být zapotřebí při bootování, musí být dostupné lokálně)
- /boot – soubory pro zavaděč systému (obrazy jádra, počáteční FS)
- /dev – rozhraní zařízení (speciální soubory)
- /etc – konfigurační soubory pro systém i aplikace
- /home – domovské adresáře uživatelů
- /lib – sdílené knihovny a moduly jádra
- /media – přípojný bod pro přenosná zařízení
- /mnt – přípojný bod pro (dočasné) FS
- /proc – informace o procesech a jádru
- /root – domovský adresář superuživatele
- /run – dočasné informace o běžícím systému (démoni)
- /sbin – programy pro superuživatele (nutné pro bootování, ne vše je spustitelné superuživatelem)
- /sys – informace o jádru, zařízeních, modulech, ...
- /tmp – dočasné pracovní soubory (obsah se maže při restartu)
- /usr – obsahuje dále adresáře a soubory, které nejsou nutné při zavádění systému, struktura je zde podobná jako u /
  - bin, sbin, lib,
  - include (hlavičkové soubory),
  - share (soubory je možné sdílet nezávislé na architektuře),
  - local (kořen další hierarchie určená pro lokální nestandardní instalace programů),
  - src (zdrojové texty jádra)
- /var – soubory měnící se za běhu systému
  - log (záznamy o činnosti systému),
  - spool (pomocné soubory pro tisk),
  - mail (poštovní přihrádky uživatelů)

**Nové téma: Datové struktury a algoritmy pro vstup a výstup**

## 7.8 Použití vyrovnávacích pamětí

Cílem použití cache (vyrovnávacích pamětí) je minimalizace počtu pomalých operací s periferiemi (disky). Hierarchie: kolekce, sbírka dílčích vyrovnávacích pamětí (s velikostí jednoho alokačního bloku, či násobku), nazývá se buffer-pool, může mít pevnou velikost, ale obvykle se mění.

## 7.9 Operace se soubory

### 7.9.1 Čtení

**První čtení alokačního bloku:**

- zjistí se, zda je blok v paměti,
- pokud ne, naalokuje se nový blok, může se využít již nějaký systémem předalokovaný a nevyužitý,
- načtou se data z disku, přesunou se do vyrovnávací paměti,
- vyrovnávací paměť je v prostoru jádra (běžné procesy zde nemají přístup),
- vykousne se z načtených dat ta část, o kterou má uživatel/proces zájem,
- nakopíruje se to do adresového prostoru uživatelského prostoru

**Při dalším čtení:**

- nejprve se opět vyhledá, zda je blok v paměti,
- pokud ano, nebude se číst z disku, pouze se z alokačního bloku vykousne část, o kterou má uživatel/proces zájem,
- tato část se uživateli předá

### 7.9.2 Zápis

**Postup při zápisu:**

- nejprve se zjistí, zda je blok v paměti,
- pokud ne, přidělí se vyrovnávací paměť,
- načtou se data z disku do vyrovnávací paměti,
- jádro převezme od procesu, který chce zapisovat, data, která chce zapsat,
- přepíše jimi danou část alokačního bloku, dirty bit se změní (0 na 1),
- operace končí (neprovede se zápis na disk),
- časem se provede zpožděný zápis na disk a vynuluje se dirty bit

Systém sám od sebe s periodou přepisuje obsah z caches na disky, lze si to vynutit pomocí sync či fsync.

Pokud je známo, že se přepíše celý alokační blok (nebo se jedná o nový blok), buffer se vynuluje a nenačítají se data z disku do cache.

## Definice:

***dirty bit*** je indikátor toho, jestli jsou data cache sladěna s obsahem na disku (0 – data v cache jsou shodná s těmi na disku, 1 – data cache != data disk – nuluje se zpožděným zápisem)

### 7.9.3 Otevření souboru pro čtení

Pokud soubor ještě **nebyl otevřen**:

- systém musí vyhodnotit cestu a najít číslo i-uzlu (resp. číslo datové struktury poskytující informace o daném souboru – přístupová práva, kde jsou uložena data),
  - při tom se postupně načítají i-uzly všech adresářů vedoucí na soubor,
  - poté se načte i-uzel souborů,
  - systém používá *d-entry cache* (speciální vyrovnávací paměť použitá pro překlad odpovídajících jmen souborů na i-uzel)
  - dále alokuje položku v tabulce **V-uzlu**,
  - z disku se načte i-uzel,
  - vloží se do nově alokované položky = vzniká rozšířená paměťová kopie i-uzlu,
  - budou tam i informace navíc (jako je počet odkazů na danou položku – s daným i-uzlem může pracovat více procesů).
- v tabulce popisovačů vytvoříme novou položku,
  - tato tabulka je uložena v záznamu o procesu (tabulka procesu v jádře) nebo v uživatelské oblasti,
  - použije se nejnižší volná položka zde,
  - naplní se odkazem na položku v tabulce otevřených souborů,
- pokud se otevření vydaří, vrátí se číslo popisovače, pokud ne, tak vrací -1.

Tolik tabulek se používá pro zamezení duplikaci údajů. Během otevírání se provádí kontrola přístupových práv. Soubor je možné otevřít v režimu pro čtení, zápis, nebo čtení i zápis.

Další otevření souboru (**již jednou otevřeného**):

- opět se vyhodnotí cesta k souboru a získá se číslo i-uzlu,
- systém se podívá do tabulky V-uzlů,
- zjistí, že i-uzel už tam je,
- nebude se znovu i-uzel načítat z disku, pouze se zvýší čítač použití i-uzlu,
- tabulka V-uzlu musí být vyhledávací (typicky vyhledávací struktury jako hash tabulka, strom, . . . ),
- naalokuje se nová položka v tabulce otevření (náplní se režimem otevření, pozicí, odkazem na sdílený V-uzel),
- naalokuje se nově položka ve file descriptoru ukazující na nové otevření (a ta se vrátí)

**Je možné přidávat i další identifikátory, např.:**

- příznak, že má být soubor vytvořen, pokud neexistuje,
- pokud existuje, má být zkrácen na 0,
- otevřít v režimu přidávání (kdekoli je aktuálně ukazovátko v souboru, tak v případě zápisu se automaticky posune na konec a tam se přidá),
- synchronní zápis (operace zápisu skončí až tehdy, když se data zapíšou opravdu na disk)

**Při chybě:**

- `open()` vrací -1,
- nastaví se chybový kód, který blíže popisuje, co se stalo (do knihovní proměnné `errno`),
- existují standardní chybové kódy,
- lze použít standardní knihovní funkci `perror()`

### Definice:

***V-uzly*** je v podstatě tabulkou i-uzlů souborového systému VFS
***tabulka popisovačů*** je pole s řádky číslovanými od 0 (0: stdin, 1: stdout, 2: stderr)
***tabulka procesů v jádře*** je část adresového prostoru, ve kterém má jádro uložené pomocné informace k procesům a má sem přístup pouze jádro

**Linux:**
`fd = open([jméno souboru], [režim])` otevře soubor

#### 7.9.4 Čtení a zápis z/do souboru

**Čtení:**

- zkontroluje se platnost popisovače (otevření popisovače, soubor pro čtení),
- pokud se jedná o první přístup, naalokuje se cache, načtou se data do cache a z cache se příslušná data použijí,
- pokud už jsou data v cache, načtou se odtud,
- předání se děje z cache (RAM, jádro) do pole (RAM, část adresového prostoru procesu),
- funkce vrací počet opravdu přečtených bajtů nebo -1 při chybě (+ nastaví errno).

**Zápis:**

- funguje podobně jako `read()`,
- před vlastním zápisem kontroluje dostupnost diskového prostoru a tento prostor alokuje (rezervuje),
- vrací počet opravdu zapsaných bajtů nebo -1

**Linux:**

`read`([popisovač], [adresa paměti, kam se má zapsat], [kolik bajtů se má načíst]): přečte soubor
`write`([popisovač], [adresa paměti, ze které se načtou data], [kolik bajtů se zapíše]): zápis do souboru

#### 7.9.5 Přímý přístup k souboru

Náhodné přesouvání v souboru. **Postup:**

- zkontroluje, zda je popisovač platný (je soubor otevřen?)
- nastaví pozici
  - SEEK_SET – např. 200 – posunu se od 200 bajtů od začátku,
  - SEEK_CUR – od aktuální pozice,
  - SEEK_END – od konce souboru,
- nelze se posunou před začátek souboru,
- je ale možné se posunout za konec souboru (a zapsat),
- vrací se výsledná pozice od začátku souboru nebo -1

Posunem za konec souboru a následným zápisem vznikají tzv. *řídké soubory* (sparse files):

- umožňuje na disku o nějaké kapacitě vytvořit soubor, který má zdánlivě větší velikost než samotný disk,
- bloky do kterých se nezapisovalo nejsou alokovány a nezabírají diskový prostor (při čtení se považují za 0),
- také může vzniknout mazáním uprostřed souboru (hole punching)

**Linux:** `lseek`([popisovač souboru], [offset], [oproti čemu se chci posouvat]): přímý přístup k souboru


Obrázek 3: Z prezentace IOS: Správa souborů – řídké soubory

### 7.9.6 Zavření souboru

- zkontroluje se platnost file descriptoru (je vůbec otevřený?),
- uvolní se daná položka v tabulce popisovačů,
- systém se podívá na odkazovanou položku v tabulce otevřených souborů,
- sníží se počítadlo o 1,
- pokud bude počítadlo != 0, uzavírání skončí,
- pokud bude počítadlo == 0, pokračuje se do
- příslušné položky tabulky V-uzlu, sníží se zde počítadlo o 1,
- pokud bude zde počítadlo != 0, uzavření skončí,
- pokud bude zde == 0, soubor se definitivně uzavře,
- z tabulky V-uzlu se uvolní z paměti i-uzel (se změněnými údaji – čas zápisu, přístupů, modifikace i-uzlu, ...),
- naplánuje se blok, ve kterém je i-uzel uložen,
- časem se i-uzel zapíše na disk,
- funkce vrací 0 nebo -1 při chybě

Pokud se proces ukončí, automaticky se zavřou všechny jeho deskriptory. Uzavření souboru nezpůsobí uložení obsahu jeho vyrovnávací paměti na disk.

**Linux:** `close`([popisovač souboru]) zavře soubor

### 7.9.7 Duplikace deskriptoru souboru

- zkontroluje se platnost deskriptoru (je soubor otevřen?),
- zkopíruje obsah původního popisovače do nového (odkaz ve fd tabulce se zkopíruje do další položky v této tabulce – oboje ukazují na stejnou položku tabulky otevřených souborů + inkrementuje se počitadlo),
- automaticky se nový deskriptor uzavře (pokud je otevřen),
- vrací index nově vytvořené položky nebo -1,
- typické použití je u přesměrování (stdin/stdout)

**Linux:** `dup`([popisovač]) duplikace deskriptoru (duplikuje existující popisovač do nejvyššího volného nového) `dup2`([popisovač], [nový popisovač]) duplikace deskriptoru (do kterého popisovače se duplikuje)

### 7.9.8 Rušení souboru

- vyhodnotí se cesta, zkontroluje se platnost jména souboru, přístupová práva (zápis),
- odstraní se pevný odkaz (=hard link) mezi jménem souboru a i-uzlem,
- zmenší se počet jmen v i-uzlu,
- pokud je počet jmen == 0 a i-uzel nikdo nepoužívá, i-uzel může být uvolněn a mohou být uvolněny všechny bloky souborů,
- dokud má soubor alespoň 1 jméno nebo nemá žádné jméno ale je alespoň jednou otevřen, nelze soubor z disku opravdu smazat,
- funkce vrací 0 nebo -1 při chybě

Je možné provést unlink na otevřený soubor (smaže se až po jeho uzavření) a pracovat s ním dále – využití při instalacích nových verzích programů, které aktuálně běží.

**Linux:** `unlink`([jméno souboru, příp. cesta]) ruší soubor

`shred` – bezpečné mazání

### 7.9.9 Další operace se soubory

**Linux:**

`creat, open` – vytvoření souboru

`rename` – přejmenování souboru

`truncate, ftruncate` – zkrácení souboru

`fcntl, lock` – zamykání záznamů

`chmod, chown` – změna atributů

`utime-` – umožňuje změnit časy práce se soubory (neumožňuje změnit čas modifikace i-uzlu)

`stat` – získání atributů (velikost, práva, ...)

`sync, fsync` – vynucení si zápisu vyrovnávacích pamětí na disk

### 7.9.10 Adresářové soubory

Obsahuje dvojice číslo i-uzlu a jména souborů. Adresáře nelze zapisovat či číst po bajtech.

**Linux:**

`mkdir` – tvoří se adresáře (vytvoří položky `.` a `...`)

`opendir` – otevře adresář

`readdir` – čte adresář

`closedir` – zavře adresář

`creat, link, unlink` – modifikace se provádí nepřímo vytvářením a modifikacemi souborů

### 7.9.11 Blokové a znakové speciální soubory

Představují rozhraní k blokovým, či znakovým zařízením (`/dev/...`, viz 7.2).

- lze je vytvořit pomocí `mknod`,
- typicky tyto soubory vytváří jádro či démoni (udev, devd – při připojení zařízení se vytvoří automaticky příslušný soubor)

Při použití běžných souborových operací jádro mapuje operace na odpovídající podprogramy, které ty operace implementují pro daný typ zařízení s využitím **tabulek**:

- znakových zařízení,
- blokových zařízení

Tyto tabulky obsahují ukazatele na funkce implementující příslušné operace v ovladačích daných zařízení.

Speciální soubory na disku zabírají **pouze i-uzel**, kromě běžných údajů mají v i-uzlu typ souboru a 2 dvadaje:

- hlavní číslo,
  - major number,
  - udává typ zařízení
  - odkazuje do tabulky zařízení (hlavní číslo = $n$-tý řádek tabulky),
- vedlejší číslo
  - minor number,

  - udává instanci zařízení
  - používá se jako parametr při volání určité operace – parametr funkce ovladače (číslo = které zařízení se má přesně použít)
- typ souboru určuje tabulku (blok., znak.)

**Linux:**

`mknod` vytvoří speciální soubory

***ovladač*** je sada podprogramů pro řízení určitého typu zařízení (viz 2.4 nebo 5.9)

## 7.10 Terminály

Terminály jsou fyzická či logická zařízení umožňující (primárně) textový vstup a výstup systémů (po řádcích), editaci vstupního řádku či použití speciálních znaků (Ctrl+C vyvolává SIGINT, Ctrl-D značí konec vstupu, ...).

**Rozhraní:**

- /dev/tty – pro každý proces, který má řídicí terminál, odkazuje na jeho řídicí terminál
- /dev/ttyS1 – fyzické terminály na sériové lince,
- /dev/tty1 – virtuální terminály (konzole),
- pseudoterminály (/dev/ptmx – master, /dev/pts/1, ...) tvořený dvojicí master/slave, po každém otevřením se vytvoří nový slave – emuluje komunikaci přes sériovou linku (umožňuje pro propojení určitých částí, např. SSH – propojení klienta se vzdáleným klientem)

**Různe režimy zpracování znaků** (řádkové disciplíny, line discipline):

- raw – neprovádí se zpracování znaků,
- cooked – zpracování všech řídicích znaků,
- cbreak – provádí zpracování malého počtu znaků (Ctrl+c, mazání, ...)

Nastavení režimu zpracování znaků je možné pomocí `stty`. Dále je možné nastavit **režim terminálu**:

- příkazy `tset`, `tput`, `reset`, ...
- proměnná TERM, ve které je uložen aktuální typ terminálu,
- typy terminálů (příkazy `terminfo`, `termcap`)

Tyto příkazy komunikují s terminálem pomocí *escape sekvencí*. Knihovna `curses` je standardní knihovna pro řízení terminálů či tvorbu aplikací s terminálovým uživatelským rozhraním.

### Definice:

***escape sekvence*** jsou sekvence znaků escape, příkaz [parametry], escape, příkaz, ...

## 7.11 Roury

Roury jsou prostředkem meziprocesové komunikace. Rozlišujeme:

**Nepojmenované roury**

- nemají adresářovou položku, tedy neexistují v souborovém systému,
- lze s nimi pracovat pouze tak, že se vytvoří pomocí volání `pipe` (vrátí čtecí a zápisový deskriptor), jakmile dojde k uzavření, práce s rourou končí,
- mohou s ní pracovat běžně pouze příbuzné procesy,
- je dostupná pomocí popisovačů z tabulky popisovači (při klonu procesu se naklonuje popisovací tabulka – proces bude ukazovat na stejné místo v tabulce otevřených souborů),
- jediná výjimka, jak je možné odkaz na nepojmenovanou rouru předat, je přes UNIXové sockety (kromě klonování procesu),
- vytváří se v kolonách (např. paralelně běžící procesy p1 | p2 | p3 – na přesměrování se používají nepojmenované roury)

**Pojmenované roury**

- vytvářejí se pomocí `mkfifo`,
- existují v souborovém systému,
- mohou se zavřít, otevřít, apod.

Roury slouží jako mechanismus meziprocesové komunikace. Implementované jsou jako kruhový buffer s omezenou kapacitou. Procesy komunikující přes rouru jsou synchronizovány.

### Definice:

***příbuzné procesy*** – pokud jeden proces otevře rouru a začne se *klonovat* (viz 8.10), všechny tyto procesy mohou s rourou pracovat
***konzumenti*** – procesy, které čtou
***producenti*** – procesy, které zapisují

# 8

**Osmá přednáška:** Dokončení souborových systémů: roury, sockety, VFS. Procesy.

## 8.1 Sockety

Umožňují jak síťovou (klient-server, TCP, UDP), tak lokální (souborový systém) komunikaci.

Pro vytvoření socketu se používá volání `socket`:

- následně se čeká na připojení (`bind` – propojit socket s TCP/UDP portem či souborem, `listen` – začínám čekat, `accept` – příjem příchozího spojení),
- klient se připojí pomocí `connect`,
- příjem a vysílání zpráv (`recv`/`send` či `read`/`write` – volání vrací popisovače otevřených souborů),
- uzavření (`close`)

Sokety podporují blokující i neblokující I/O. Při práci s více sockety je možné je obsluhovat více procesy či vlákny (příkaz `select`) – typy souborů, u kterých může nastat potřeba čekat na možnost provedení určité operace. Sokety taky mají výhodu, že je možné vytvořit aplikace, které mohou běžet distribuovaně v síti.

### Definice:

***blokující režim*** – pokud chci načítat data ze socketu, budu pozastaven, dokud se nějaká data neobjeví
***select*** umožňuje testovat, zda na popisovači (či na množině popisovačů) je dostupná nějaká operace
***pasivní čekání*** – nespotřebovává se CPU čas, energie, ...

## 8.2 VFS

Virtual File System. Definice viz 5.11.5. Komunikace s různými FS se přenáší z uživatele na autora FS, který pokud chce, aby daný FS byl využitelný, musí ho provázat s VFS (propojení VFS a uživatele řeší vývojáři).

Typická datová struktura VFS jsou **V-uzly** = rozšířené paměťové kopie i-uzlu, které kromě dat i-uzlu obsahují další data:

- jako počet odkazů na V-uzel z tabulky otevřených souborů,
- ukazatele na funkce implementující operace nad i-uzlem (v patřičném FS).

## 8.3 NFS

Network File System. Zpřístupňuje soubory uložené na vzdálených systémech.

Jedná se o **systém klient-server**:

- klient požádá např. o čtení ze souboru,
- všechny operace prochází přes VFS,
- požadavek se z VFS předá na NFS klienta,
- NFS klient předá požadavek na NFS server,
- NFS server pracuje s lokálním FS již na vzdáleném PC,
- ten pracuje také s VFS (ale na serveru), prostřednictvím něj získá data z lokálního FS,
- data poté putují přes síť zpět k uživateli.

Umožňuje **kaskádování** – je možné si lokálně do jednoho adresářového stromu připojit vzdálený strom (a do něj připojit další vzdálený souborový systém). **Autentizujeme se nejčastěji přes UID, GID** (musí existovat důvěra mezi správcem lokálního a vzdáleného systému), může ale používat i jiné mechanismy (kryptografie, ...).

NFS verze 3:

- starší, bezstavová verze – nepoužívá operace otevírání, uzavírání souborů, každá operace si musí nést veškeré informace o souboru,
- na straně klienta nemá cache (složitá implementace), na straně serveru cache,
- nemá podporu zamykání (operace pro zamknutí záznamu souboru jsou prázdné).

NFS verze 4:

- stavová,
- cache na straně klienta,
- podpora zamykání.

## 8.4 Spooling

Simulatenous perpiheral operation on-line (simultánní online provádění periferních operací). Jedná se o **provádění online operací bez čekání periferní operace (výstup) na periferiích, které nemusí online prokládání dat od různých procesů či uživatelů podporovat**. (Příklad: síťová tiskárna – spousta uživatelů, každý chce, aby tisk se provedl okamžitě.)

Výstup se provede do vyrovnávací paměti spool (soubor), systém si vede frontu čekajících úloh, operaci, do fronty se zařadí odkaz na vytvořený soubor, úloha se dokončí po uvolnění periferie.

**Linux:** `/var/spool` obsahuje soubory spool

**Nové téma:** Správa procesů.

Správa procesů (process management) zahrnuje:

- přepínání kontextu (*dispatcher*, vždy v režimu jádra),
- plánovač (nemusí být v jádře),
- správu paměti,
- podporu meziprocesové komunikace (signály, roury, sockety, synchronizace -. semafory, mutexy, . . . )

### Definice:

***přepínáním kontextu*** rozumíme fyzické odebírání procesoru jednomu procesu a přidělování jinému procesu (také viz 1.3)

***plánovač*** rozhoduje, který proces či procesy poběží a případně jak dlouho

## 8.5 Proces

Definice viz 1.2. Proces je běžící program, tedy aktivní entita, abstrakce aktivity probíhající v systému. Program je naopak pasivní entita (definice viz 1.2).

**Proces je v OS definován:**

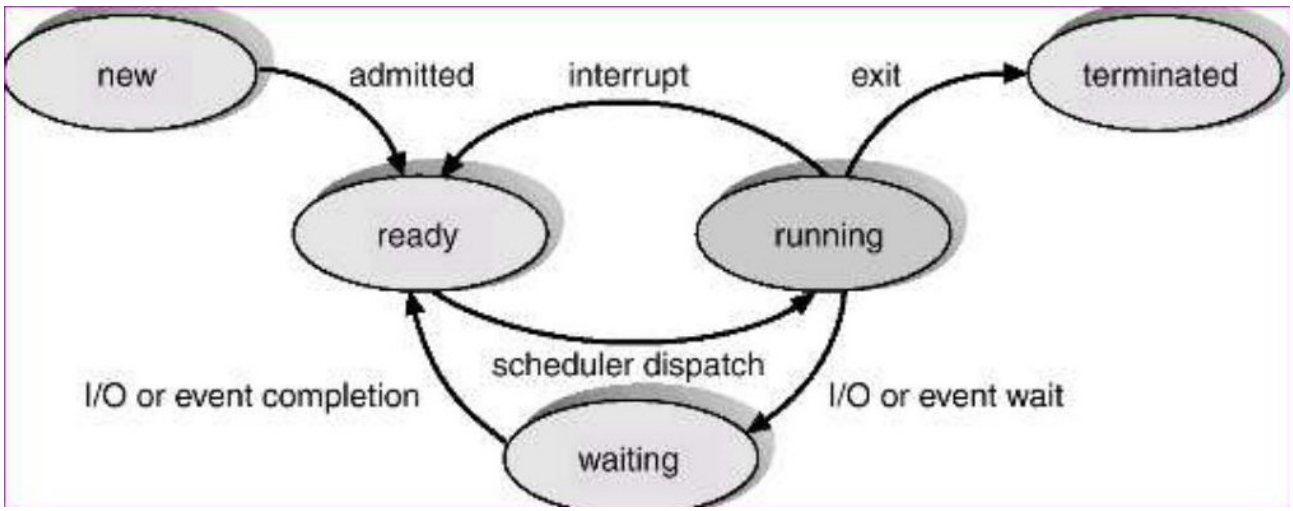
- unikátním identifikátorem (PID – process identifier),
- stavem plánování,
- řídicím programem,
- obsahem registru (běžných – EAX, EBX, EIP, . . . ),
- zásobníkem (aktivační záznamy – informace o rozpracovaných funkcích),
- daty (statická ne/inicializovaná data, hromady, individuálně alokovaná paměť),
- tím, jaké další vazby a zdroje OS využívá (jaké soubory má aktuálně otevřené, signály, obslužné funkce signálů, PPID, UID, GID, semafory, sdílená paměť, sdílené knihovny, . . . )

## 8.6 Stavy plánování a jejich změny

Nejzákladnější plánovací diagram (většiny různých OS) – **stavy procesu**.

**Stavy plánování procesů (obecně):**

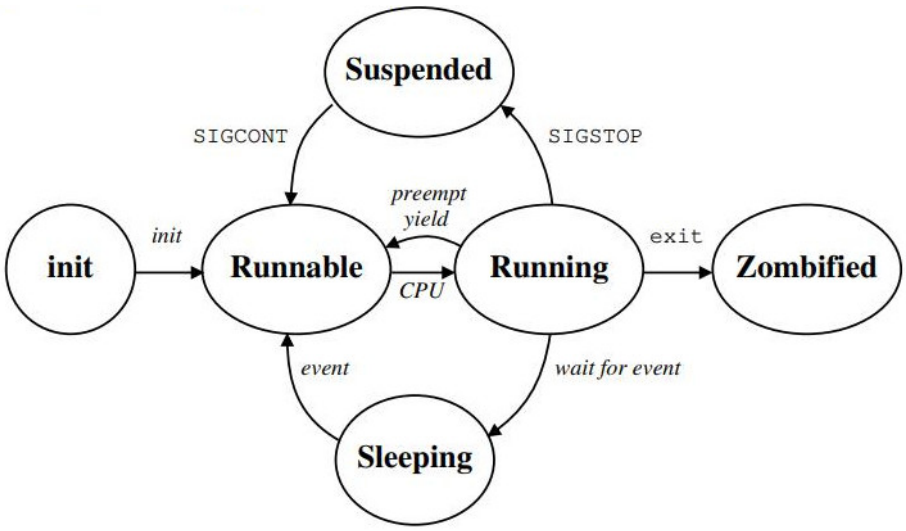
- new – proces je inicializován (vytváří se struktury, co jej popisují, data, proces případně čeká ve vstupní frontě dlouhodobého plánovače, . . . )
- ready – proces čeká na krátkodobý plánovač na přidělení procesoru, dispatcher provede přepnutí, přepne se do running
- running – proces může být přerušen (preemptivni plánování – opět stav ready), může požádat o službu jádro (I/O operace, sync) – stav waiting,
- waiting – proces čeká na dokončení operace (služba jádra), poté půjde do stavu ready,
- terminated – proces skončí (proces se v systému nějakou dobu vyskytuje ve stavu ukončený)




Obrázek 4: Z prezentace IOS: Správa procesu – stavový diagram obecného plánování procesu

**Stavy plánování procesů v UNIXu:**

- init – vytvořený, neinicializovaný,
- runnable – připraven běžet,
- running – proces běží (přidělen procesor), v případě preempce (proces se vzdá CPU) jde do runnable,
- sleeping – proces požádá o I/O či sync akci (čeká na dokonce operace), po realizaci bude runnable
- suspended – pomocí signálu SIGSTOP může být proces zmrazen a čeká čeká na rozmrazení (signál SIGCONT),
- zombified – ukončení procesů a přechod do stavu zombie (mátoha), proces skončil, odebrány všechny zdroje, pouze o něm zůstává záznam v tabulce procesů (zde je jeho návratový kód, dokud si ho někdo nepřevezme)




Obrázek 5: U prezentace IOS: Správa procesů – stavový diagram plánování procesů v UNIXu

OS s procesem pracuje tak, že je reprezentován pomocí struktury PCB (process control block), někdy také task control block či task struct.

**PCB zahrnuje (přímo nebo formou odkazů):**

- identifikátory spojené s procesem,
- stav plánování,

- obsah registru (v okamžiku, kdy je pozastaven),
- plánovací informace (priorita, ukazatele na plánovací fronty, ...),
- informace spojené se správou paměti (tabulky stránek při použití stránkované paměti),
- informace spojené s účtováním (sumarizuje informace o běhu procesu – spotřeba CPU, ...),
- informace o využití I/O zdrojů (otevřené soubory, tabulka popisovačů, ...)

PCB může být buď jedna struktura nebo může být rozděleno na několik částí.

## 8.7 Části procesu v paměti v UNIXu

První součást paměti využité procesem je **uživatelský adresový prostor** (user address space) – je přístupný procesu (může z této části paměti číst a psát do ní), obsahuje:

- kód (který je řízen, code area/text segment),
- data (ne/inicializovaná, hromada, alokovaná paměť),
- zásobník,
- soukromá data sdílených knihoven, sdílené knihovny či sdílená paměť

Další část informací o procesech bývá v některých případech (ne vždy) umístěna v **uživatelské oblasti** (např. Linux tento koncept nepoužívá, vše má v tabulce procesů):

- informace jsou uloženy pro každý proces v části uživatelského adresového prostoru, která *není procesům přístupná*,
- je to přístupné jádru,
- u každého procesu si v této části ukládá informace o procesu,
- je tam:
  - část PCB, která je používána zejména za běhu procesu,
  - PID, UID, EID, GID, EGID, PPID (identifikátor rodiče),
  - obsah registrů,
  - deskriptory souborů (informace o tom, který deskriptor je použit jako stdin, stdout),
  - obslužné funkce signálů (funkce, které se budou volat pro obsluhu signálů),
  - účtování,
  - pracovní, kořenový adresář

**Další záznamy jsou v tabulce procesů:**

- uloženo trvale v jádru,
- informace o procesu, které jsou důležité, i když proces neběží:
- PID, PPID, UID, EID, ...,
- stav plánování,
- událost, na kterou proces čeká,
- plánovací informace (pro plánovač – při rozhodování, který proces dál poběží či ne – podle priority, spotřeba času, viz 8.16),
- čekající signály (signály, které mohou přijít, i když proces neběží),
- odkaz na tabulky dat reprezentující rozložení procesů, dat, kódů, zásobníků, ...

Poté jsou ještě záznamy v **tabulce paměťových regionů:**

- jak je rozdělen uživatelský adresový prostor na regiony (= souvislý kus paměti použitý pro kód, zásobník, ...),
- velikost těchto regionů,
- globální tabulka regionů (odkaz z této na lokální tabulky regionů),
- regiony bývají členěny na stránky (tabulky stránek)

### Definice:

***logický adresový prostor*** je rozsah všech logických adres, které se mapují do fyzické paměti

***zásobník jádra*** je separátní zásobník, který je někdy používán pro ukládání rozpracovaných funkcí jádra v okamžiku, kdy jádro provádí službu pro daný proces

## 8.8 Kontext procesu

Kontext je jiné označení pro **stav procesu**. Rozlišujeme:

- uživatelský kontext – část stavu procesu popisující část paměti dostupnou procesu samotnému (kód, zásobník, data),
- registrový kontext,
- systémový kontext – část stavu procesu nedostupná samotnému procesu (uživatelská oblast, položky tabulky procesů, paměťové regiony, ...)

## 8.9 Systémová volání nad procesy UNIXu

= standardní POSIXová volání:

- fork, exec, exit, wait, waitpid,
- kill, signal – synchronizace,
- getpid, getppid – získávání identifikátorů, ...

**Identifikátory spojené s procesy v UNIXu:**

- identifikace procesu PID (vlastní identifikátor procesu),
- identifikace předka PPID (proces, který daný proces vytvořil, rodič),
- uživatel (a skupina), který proces spustil – UID, GID,
- efektivní uživatel či skupina – EUID, EGID (viz 7.6),
- uložené EUID a EGID – procesům umožňují dočasně se zbavit vysokých práv, která získal (proces se dobrovolně vzdá vyšších práv — uloží se a poběží s běžnými právy – v okamžiku provádění kritických operací si práva zase navýší – ochrana před chybami v programech),
- v Linuxu FSUID, FSGID (filesystem UID/GID – oddělená zvýšená privilegia pro práci s FS),
- PGID, SID (process group identifier, session identifier – skupina procesů či sezení, do kterých proces patří)

### Definice:

***sezení*** je skupina skupin procesů vytvářející se typicky při práci s terminály

## 8.10 Vytváření procesu

Procesy v UNIXu vznikají **voláním služby `fork`**. Fork je volání, které se **zavolá jednou**, pokud nedojde k chybě, **skončí dvakrát**. Na základě volání fork vzniká **vztah rodič-potomek** (parent-child) a hierarchie procesů. Výsledkem forku je totiž **duplikace procesu**. Vzniká takřka identická kopie potomka, který dědí:

- řídicí kód, data, zásobník, sdílenou paměť, otevřené soubory, obsluhu signálů, většinu synchronizačních prostředků, ...
- pro efektivitu používá pro práci s pamětí copy-on-write
- kopie se liší v návratovém kódu `fork`, identifikátorech, údajích spojených s plánováním a účtováním, nedědí se čekající signály, souborové zámky a některé další zdroje či nastavení

**Návratové kódy fórků:**

- 0: fork se zdařil, `if (pid == 0) { kód potomka }`
- $-1$: fork se nezdařil, `if (pid == -1) { kód rodiče }`
- cokoli jiného (PID potomka): `else { kód rodiče }`

## 8.11 Hierarchie procesu v UNIXu

Prvním procesem, který vzniká a vytváří jej jádro, je **proces `init` s PID=1** (novější a aktuálně často používaná implementace: `systemd`). Tento proces je **předkem všech ostatních uživatelských procesů**. **PPID initu je 0**.

Existují také procesy jádra (kernel threads/processes), jejich předkem `init` **není**:

- jejich kód je součástí jádra,
- vyskytuje se i proces s PID=0, vzniká úplně jako první, podílí se na inicializaci jádra, následně se mění na swapper (pokud je na systému použit) nebo na čekací smyčku nebo je to používáno jako procesová obálka pro vlákna jádra (na Linuxu se tento proces nevypisuje)

`init` se podílí na **inicializaci systému**, poté **přebírá návratové kódy procesů, které osiřely dřív, než skončily** (tedy rodič skončí dřív než potomek) – teoreticky by byl zombie procesem do nekonečna, proto `init` přebírá jeho návratový kód a umožní mu odchod ze systému.

### Definice:

***swapper*** je proces, který v případě akutního nedostatku paměti některé procesy pozastaví a zcela uloží na disk (veškeré části paměti zabírané procesem)

***kthread*** je (na Linuxu) „init procesem“ pro procesy jádra, má PID=2

**Linux:** `pstree` výpis stromu procesů

## 8.12 Změna programu – `exec`

Umožňuje v rámci existujícího procesu „vyměnit jeho vnitřnosti“ – **zahodit existující kód a nahradit ho kódem jiným**. `exec` je funkce, která se **zavolá jednou a neskončí vůbec** (je v nějakém kódu, ten kód přestane běžet) pokud nedojde k chybě.

Pokud v procesu zavolám `exec`:

- dědí řadu rysů svého předka,
- zůstává mu řada zdrojů a vazeb s OS (identifikátory, otevřené soubory, ...),

- zanikají vazby a zdroje vázané na původní kód (obslužné funkce signálů, sdílená paměť, paměťově mapované soubory, semafory).

Skupina funkcí exec:

- `execve` (základní volání), `execl`, `execlp`, `execle`, `execv`, ...

Ve Windows se procesy vytváří voláním `CreateProcess(...)`, které zahrnuje funkčnost `fork` i `exec`.

## 8.13 Čekání na potomka – `wait, waitpid`

Slouží k tomu, aby mohl **rodič čekat na dokončení činnosti svých potomků**.

Volání `wait`:

- čeká na ukončení *jednoho z potomků*,
- vrací číslo potomka, který skončil (případně $-1$ pokud přijde signál, který čekání přeruší, nebo pokud čekáme na potomka a žádného nemáme),
- může to být operace blokující, pokud žádný z potomků ještě neskončil,
- pokud některý potomek skončil dřív před voláním `wait`, volání okamžitě skončí a vrátí se návratový kód potomka.

Volání `waitpid`:

- umožňuje čekání na ukončení určitého potomka určité skupiny dle PID,
- umožňuje čekání i na pozastavení či probuzení (SIGSTOP, SIGCON).

## 8.14 Start systému

- (nejprve) dostane se ke slovu firmware PC (UEFI/BIOS),
- načtení a spuštění zavaděče OS (někdy se zavaděčů používá několik, např. BIOS využíval základní kód MBR – série zavaděčů),
- načtou se inicializační funkce jádra a samotné jádro, spuštění inicializačních funkcí,
- inicializační funkce jádra vytvoří proces 0, další procesy jádra a proces `init`,
- proces `init` pokračuje v inicializaci systémy, spouští démony a procesy,
- v určitém okamžiku se z něj spustí procesy umožňující práci s X Window System a přihlášení v GUI (GDM, SDDM, LightDDM),
- na konzolích se spustí `getty` (*Ctrl+Alt+F1, F2, ...*) – umožní uživateli zadat přihlašovací jméno, změní se na `login`, načte od uživatele heslo, poté se změní na shell, ze kterého se spouští další procesy, po ukončení se opět spouští `getty`,
- proces `init` i po inicializaci nadále běží, přebírá návratové kódy procesů, jejich rodič skončil dřív než příslušný proces, také řeší reinicializaci systému (na přání uživatele či výpadek napájení)

### Definice:

***firmware*** je program uložený v nevolatilních pamětech, provádějící kontrolu HW a případnou inicializaci HW

## 8.15 Úrovně běhu

Systém úrovní běhu byl zaveden již v UNIX System V. Rozlišují se úrovně běhu 0–6: (některé měly předpřipravený standardní význam, některé si definoval administrátor)

- 0 = halt – zastavení systému,
- 1/s/S = single user mode – jednouživatelský režim, používá správce systému pro administrativní úlohy,
- 6 = reboot – automatické restartování systému,
- ostatní runlevely (2–5) definovány na různých systémech různě,
- je možné změnit úrovně běhu (režimy) pomocí `tellinit [N]`.

Konfigurace úrovní běhu:

- v adresáři `/etc/rcX.d` (X=úroveň běhu) jsou skripty spouštěné při vstupu do dané úrovně,
- nejprve se volají skripty začínající písmenem $K$ v pořadí daném číslem za tím $K$ (volají se s argumentem `stop`),
- poté se volají skripty začínající $S$ (volají se s argumentem `start`),
- start, stop – definuje se co se má spustit či jak se co má zastavit,
- v adresáři `/etc/init.d` vytvoříme skript, který požadovanou službu bude umět spouštět, na patřičné místo K a S odkazů se vytvoří symlink na požadovaný skript,
- skripty v `init.d` typicky přijímají parametry `start`, `stop`, `reload`, `restart`
- tyto skripty se nemusí volat pouze při změně úrovně běhu, ale je možné je volat i ručně – z `/etc/init.d`
- v souboru `/etc/inittab` je horní, hlavní úroveň systému, kde se popisuje např. implicitní úroveň běhu či jaké úrovně běhu jsou podporované

Existují různé nové implementace procesu `init` – dnes nejběžnější je **`systemd`**:

- základní úrovně běhu jsou nahrazeny jednotkami (*units*), které mají různé typy (targets, services, ...),
- spouští inicializační jednotky paralelně na základě jejich závislostí (výhodou je, že inicializaci systému je možné provádět paralelně, zatímco `init` postupuje sekvenčně),
- emulují se úrovně běhu (zpětná kompatibilita),
- užitečné jsou adresáře `/lib/systemd` či `/usr/lib/systemd`

## 8.16 Plánování procesů

Procesy plánuje *plánovač*.

Rozlišujeme 2 základní **plánovací algoritmy** (definice viz 1.5):

- nepreemptivní plánování (typicky I/O operace, konec – volání `exit`, proces se sám musí vzdát CPU – `yield`),
- preemptivní plánování (typicky přerušení od časovače, může jít i o jiné, třeba od disku)

Rozlišujeme **3 „typy“ plánování**:

- dlouhodobé plánování – které úlohy budou připuštěny do systému,
- střednědobé plánování – procesy paměť mají, nebo nemají – jedná se o systém swapování,
- krátkodobé plánování – procesy mají paměť – přepínání mezi úlohami

**Systém swapování** (= střednědobé plánování):

- v případě nedostatku paměti některé procesy pozastaví, odebere jim veškerou paměť a uloží je na disk,
- tyto procesy jsou vyřazeny z plánování (nemají paměť – nemohou běžet),

- při žádosti o spuštění úlohy úloha pak nebude spuštěna ihned, ale systém čeká na uvolnění systémových zdrojů (služba čeká ve frontě) – poté už se jedná o dlouhodobé plánování (rozhoduje se o tom, které úlohy budou vůbec připuštěny do systému)

### Definice:

***plánovač*** rozhoduje, který proces či procesy poběží (a případně jak dlouho)
***systémy s nepreemptivním plánováním*** = systémy s kooperovaným plánováním (procesy musí aktivně spolupracovat, kooperovat)

## 8.17 Přepnutí kontextu (procesu)

Přepnutí kontextu na příkladu – dispečer na základě rozhodnutí plánovače přepíná mezi procesem A a B:

- bude muset uchovat stav registrů (některých, ale včetně řídicích registrů) procesu A do PCB (nebo task struct v Linuxu; viz 8.6),
- dojde k úpravě některých řídicích struktur v jádře (úprava plánovacích struktur, účtovacích struktur, ...),
- obnova uložených hodnot registrů procesu B,
- dojde k předání řízení procesu na adresu, kde bylo dříve přerušeno provádění procesu B,
- tato akce se musí provádět v režimu jádra

Neukládá se a neobnovuje celý stav procesu (při pozastavení procesu není nutné na disk ukládat obsah paměti, ukládají se pouze registry).

Přepnutí může trvat **řádově stovky až tisíce instrukcí/cyklů** – jádra umožňují měnit interval, v jakém přicházejí přerušení z časovače – je třeba si dát pozor, aby ten interval nebyl příliš krátký, protože začne poté převažovat režie systému nad užitečným během (neustálé ukládání a obnovování obsahů registrů).

# 9

**Devátá přednáška:** Dokončení správy procesů.

## 9.1 Klasické plánovací algoritmy

### 9.1.1 FCFS

- *first come, first served*,
- plánovací algoritmus založený na jednoduché FIFO frontě,
- proces, který nově vznikne nebo je uvolněn z čekání na nějaké operaci (I/O, sync, ...), případně proces, který se vzdá CPU, se zařadí na konec fronty,
- procesy, které poběží, se vybírají ze začátku fronty,
- jedná se o nepreemptivní algoritmus (k přepnutí kontextu dojde, pouze pokud se běžící proces vzdá CPU či zavolá službu jádra)

### 9.1.2 Round-robin

- preemptivní obdoba FCFS,
- pracuje podobně jako FCFS,
- každý proces má přiděleno nějaké časové kvantum,
- jakmile je mu přidělen CPU, proces běží a poběží nanejvýš po dobu časového kvanta,
- po vypršení času je procesu odebrán CPU a je zařazen na konec fronty,
- CPU se přidělí procesu ze začátku fronty

### 9.1.3 SJF

- *shortest job first*,
- nejprve se provede nejkratší úloha,
- algoritmus přiděluje CPU tomu procesu, který aktuálně deklaruje nejkratší dobu pro svůj další běh na CPU, po který nebude žádat o žádné I/O operace (tzv. CPU burst),
- běh úloh se dělí na výpočetní práce (CPU burst) a poté na periody, kdy se komunikuje s periferiemi (disky, sítě, ...),
- nepreemptivní algoritmus (nepřerušuje proces před dokončením jeho aktuální výpočetní fáze),
- statisticky minimalizuje průměrnou dobu čekání a zvyšuje propustnost systému,
- je nutné dopředu znát dobu běhu procesu na CPU v jejích jednotlivých výpočetních fázích, respektive musí existovat možnost tyto doby rozumně odhadnout (na základě předchozího chování těchto úloh),
- dává smysl pro opakovaně prováděné úlohy,
- používá se zejména v dávkových (specializovaných) systémech,
- nevýhodou algoritmu je náchylnost ke **stárnutí** při čekání na nějaký zdroj (CPU, zámek, ...)
  - pokud nějaký proces deklaruje délku CPU burst a v systému budou neustále kratší procesy s touto délkou, tyto procesy ho budou neustále předbíhat (nikdy se tak k CPU nedostane)

### Definice:

***stárnutí*** (hladovění, starvation) je situace, kdy některý proces, který žádá o zdroj, na daný zdroj čeká bez záruky, že jej někdy získá.

### 9.1.4 SRT

- *shortest remaining time*,
- preemptivní obdoba SJF

### 9.1.5 Víceúrovňové plánování

- procesy rozdělené do různých skupin (typicky dle priority, ale ne nutně – např. dle typu procesu),
- každá skupina procesu může používat jiný dílčí plánovací algoritmus (FSFS, round-robin, SJF, ...) s různými parametry,
- kromě toho máme další („hlavní“) algoritmus, který rozhoduje, která skupina procesů dostane CPU čas – často jednoduše na základě priorit skupin,
- poté je další plánovací algoritmus, který plánuje mezi skupinami

### 9.1.6 Víceúrovňové plánování se zpětnou vazbou

- skupiny procesů jsou rozděleny dle priorit,
- proces, který se stane nově připraveným běžet (nově vznikne, je uvolněn z čekání, ...) je zařazen do skupiny procesů s nejvyšší prioritou,
- v této skupině běží a postupně klesá do nižších priorit,
- až spadne do nejnižší úrovně (plánován round-robin),
- používají se varianty, kdy proces má přednastavenou statickou prioritu a zařadí se do plánovací úrovně této priority, a poté má i dynamickou prioritu, která se může zvyšovat i snižovat, typicky se priority mění tak, že pokud nějaký proces spotřebovává mnoho CPU času, priorita se sníží; pokud proces čeká na mnoho I/O operací, priorita se zvýší,
- cílem je zajistit rychlou reakci interaktivních procesů (=komunikujících s uživatelem),

## 9.2 Plánovač v Linuxu (od verze 2.6.23)

Používá se **víceúrovňové prioritní plánování se 100 základními statickými prioritními úrovněmi**:

- priority 1-99 jsou vyhrazeny pro procesy reálného času (algoritmy FCFS s preempcí na základě priorit nebo round-robin),
- priorita 0 jsou běžné procesy plánované *CFS* plánovačem (viz níže),
- v rámci úrovně 0 se používají *podúrovně* v rozmezí -20 až 19, nejvyšší podúroveň je -20 (je možné běžně uživatelsky nastavovat příkazy `nice` a `renice`),
- v rámci úrovně 0 se rozlišují 3 typů procesů: běžné, dávkové a idle,
- základní prioritní úroveň (RT proces v 1-99, běžný proces) a typ plánování (round-robin, FCFS, ...) je možné nastavit pomocí služby `sched_setscheduler`,
- později přidáno plánování pro sporadické periodické úlohy – založeno na strategii *earliest deadline first* (převzato z realtimových OS)

### Definice:

***FCFS s preempcí na základě priorit*** – pokud během běhu procesu doběhne I/O operace či `sync` operace, která způsobí proveditelnost procesů s vyšší prioritou, dojde k přepnutí kontextu (nedochází k přepnutí kontextu na základě časových kvant s procesy s stejnou prioritou)

***dávkové procesy*** mají mírnou penalizaci z hlediska priorit, mají ale delší kvantum

***idle procesy*** mají nízkou prioritu, předpokládá se, že se dostanou ke slovu až v okamžik, když v systému nic užitečnějšího není

***sporadické periodické úlohy*** jsou úlohy, které běží, mají periodické výpočetní fáze (očekávaná fáze – známe dobu jak dlouho trvají + máme časový limit, dokdy se mají výpočty provést), které se provádějí čas od času

## 9.3 Completely Fair Scheduler

CFS plánovač:

- snaží se explicitně každému procesu poskytnout odpovídající procento strojového času s ohledem na jeho priority (4 procesy, stejná priorita $\Rightarrow$ všechny 25 % CPU času),
- u každého procesu si vede údaje o tom, kolik virtuálního CPU času už ten proces na CPU strávil,
- vede si údaj o minimálním stráveném CPU čase (dává nově připraveným procesům),
- procesy udržuje ve vyhledávací struktuře (red-black tree) podle využitého CPU času,
- při rozhodování, který proces poběží, ze struktury vezme ten, který aktuálně strávil nejméně času na CPU,
- proces nechá běžet po časové kvantum, které spočítá na základě priorit,
- **virtuální procesorový čas**: situace: 2 procesy běžící celý den (každý půl dne), přijde nový proces, fyzicky na CPU běžel 0 s (nejvyšší priorita, běžel by půl dne a ty 2 by byly off), proto virtuální CPU čas – nový proces dostane čas menší než minimální strávený čas všemi procesy (ne 0),
- algoritmus má podporu pro skupinová plánování, umí rozdělovat čas spravedlivě pro skupiny procesů (spouštěné z různých terminálů, od různých uživatelů, . . . )

## 9.4 Plánování ve Windows NT a novějších

- Používá se víceúrovňové prioritní plánování se zpětnou vazbou na základě interaktivity:
- 32 prioritních úrovní:
  - 0: nulování volných stránek paměti (aby se nedostaly informace od jednoho uživatele/procesu k jinému)
  - 1–15: běžné procesy
  - 16–31: procesy reálného času
- základní priorita procesu je dána kombinací plánovací třídy a plánovací úrovně (v rámci třídy),
- priorita se dynamicky snižuje či zvyšuje:
- zvýší se priorita procesů spojených s oknem, které je v popředí,
- zvýší se priorita procesů spojených s oknem, do kterého přichází vstupní zprávy (myš, časovač, klávesnice, . . . ),
- zvýší se priorita procesů uvolněných z čekání (I/O operace),
- zvýšená priorita se po každém vyčerpání kvanta sníží o jednu úroveň (až do dosažení základní priority)

## 9.5 Inverze priorit

Jedná se o **problém, nežádoucí stav**, který je nutné řešit!

- jedná se o situaci, kdy v OS máme různé prioritní procesy, málo prioritní proces si naalokuje si nějaký zdroj, zamkne si přístup k nějakému sdílenému zdroji (soubor, adresa v paměti, síťový port, . . . ),
- procesy s vyšší prioritou tyto procesy předbíhají,
- proces s nižší prioritou se čas od času dostane ke slovu, provede pár instrukcí, musí čekat,
- (má naalokovaný zdroj, chce s ním něco provést, to ale trvá dlouho, protože ho pořád někdo předbíhá),
- může nastat, že některý z prioritních procesů potřebuje právě ten zdroj, který si zamkl tento nízkoprioritní proces,

- proces s vyšší prioritou bude muset čekat – *virtuálně* se zvýší (rapidně) priorita nízkoprioritíiho procesu (= je to vedlejší efekt, reálně má prioritu pořád stejnou!),
- v systému je možné mít procesy se střední prioritou, které tento zdroj nepotřebují, ty budou předbíhat dále nízkoprioritní proces,
- proces s vysokou prioritou musí čekat, zatímco procesy se střední a nižší prioritou mají najednou „pomyslně vyšší prioritu“,
- tento jev může a nemusí vadit – může způsobit sníženou odezvu systému, nicméně můžou se zablokovat i některé kritické procesy reálného času (ovládání hardware, . . . )

**Možnosti řešení inverze priorit:**

- prioritní strop (priority ceiling – procesy v kritické sekci získávají nejvyšší prioritu,
- priority inheritance – procesy v kritické sekci, které blokují procesy s vyšší prioritou, po dobu běhu v kritické sekci dědí prioritu čekajícího procesu (s nejvyšší prioritou),
- na jednoprocesorových systémech se používá technika, kdy se po dobu běhu v kritické sekci zakáže přerušení

Další komplikace během plánování:

- víceprocesorové systémy – nutné vyvažovat výkon (aby na jednom jádru CPU neběžely 4 procesy a na zbytku 0), respektovat obsah cache CPU, lokalitu paměti (neuniformní přístup do paměti)
- hard real-time systémy – nutnost zajistit garantovanou odezvu některých akcí

### Definice:

***kritická sekce*** je sekce v kódu, kde se pracuje výlučným způsobem se sdílenými prostředky

***neuniformní přístup do paměti*** – paměť je dělena na paměťové jednotky, každá připojená k jinému CPU, ale všechny procesy mohou přistupovat do všech paměťových jednotek (za různou dobu)

## 9.6 Vlákna, úlohy, skupiny procesů

Vlákna, neboli threads:

- označovaná jako odlehčené procesy (LWP – lightweight process),
- výpočty (odpovídají vláknům) běží paralelně v jednom procesu,
- vlákna mají vlastní obsah registrů, vlastní zásobník,
- všechna vlákna sdílí stejný řídicí kód, data, další zdroje (otevřené soubory, signály),
- výhody: rychleji se spouští, přepíná, efektivnější práce (dle systému – v UNIXu je díky principu forkování rozdíl mezi vlákny a procesy menší než jiných OS)

## 9.7 Úlohy, skupiny procesů, sezení

**Úloha (job)** se pojí se shellem, je to skupina skupina paralelně běžících procesů spuštěných jedním příkazem, příkazy propojené do kolony (p1 | p2 | p3 – pipeline).

**Skupina procesů (process group) v UNIXu:**

- množina paralelně běžících procesů, se kterými je možné provádět operace jako s celkem,
- skupině je možné poslat signál jako jedné jednotce,
- předek také může čekat na libovolného potomka z určité skupiny,
- každý proces je právě v jedné skupině procesů, po vytvoření vždy je to skupina jeho předka,

- skupina může a nemusí mít vedoucího – běžně její první proces, dokud neskončí; pokud skončí, skupina je bez vedoucího,
- skupina je identifikována vedoucím skupiny; pokud vedoucí skupiny skončí, není možné jeho číslo recyklovat a použít pro ID skupiny

**Sezení v UNIXu:**

- množina skupin procesů,
- každá skupina procesů je v jednom sezení,
- sezení může a nemusí mít vedoucího,
- může mít řídicí terminál (`/dev/tty`),
- v rámci sezení platí, že jedna skupina je na popředí (čte z terminálu), ostatní jsou na pozadí,
- pokud terminál končí, signálem je SIGHUP, informován je vedoucí sezení (typicky shell), standardně se všem procesům, na které nebyl užit příkaz `nohup`/`disown`, pošle navíc SIGHUP, pokud jsou procesy pozastaveny, pošle se signál SIGCONT

## 9.8 Komunikace procesu

Používají se prostředky **IPC – inter-process communication**:

- signály (umožňují zasílat mezi procesy číselnou informaci),
- roury,
- zasílání zpráv (umožňují posílat řetězcová data),
- sdílená paměť,
- sockety,
- RPC (remote procedure call)

## 9.9 Signály

V základní verzi je číslo (int), které je procesu zasláno prostřednictvím pro to zvlášť definovaného rozhraní (= signály zajišťuje OS). Jsou generovány:

- při chybách (aritmetická chyba, chyby sběrnic, ...),
- externích událostech (dostupnost I/O, vypršení časovače, ...),
- na žádost procesu – IPC (procesy si mohou navzájem posílat signály, ale jádru signál nelze zaslat, jádro není proces),
- vznikají obvykle asynchronně k činnosti programu (program něco provádí, v okamžiku, který nelze předpovědět, přijde signál)

Je nutné pečlivě zvažovat obsluhu signálů, aby aplikaci **signál neshodil** – vznikají chyby, které se objevují jen zřídka a špatně se ladí (tzv. race conditions). Je tak třeba využívat technik **pokročilého testování**: vkládání šumu (uměle v náhodných okamžicích se snažíme programy zpozdit, enumerace proložení akcí programu), **nástroje pro verifikaci s formálními základy** (statická analýza, model checking).

Mezi běžně používané signály patří:

- SIGHUP – odpojení, ukončení terminálu,
- SIGINT – přerušení z klávesnice (Ctrl+C),
- SIGKILL – signál č. 9, tvrdé ukončení
- SIGSEGV (chyba segmentace, mimo přidělenou paměť – špatný ukazatel, střelba do nohy)
- SIGBUS – chybná práce s pamětí,

- SIGPIPE – zápis do roury bez čtenáře,
- SIGALRM – signál od časovače,
- SIGTERM – měkké ukončení (lze vyvrátit),
- SIGUSR1, SIGUSR2 – uživatelské signály (uživatel si je může nadefinovat)
- SIGCHLD – pozastavení či ukončení potomka,
- SIGCONT – dochází při uvolnění z čekání,
- SIGSTOP, SIGSTP (Ctrl+Z) – tvrdé / měkké pozastavení,
- další viz `man 7 signal`

### Definice:

***race conditions*** jsou časově závislé chyby (závisí na tom, jak se v čase zpracovávají paralelní operace)

#### 9.9.1 Předefinování obsluhy signálu

Mezi implicitní reakce na signál patří **ukončení procesu** (případně s generováním *core dump*), **ignorování signálu, zmražení či rozmrazení procesu**.

Předefinovat obsluhu lze u všech signálů **mimo SIGKILL, SIGSTOP**. U SIGCONT vždy dojde k odblokování procesu (a následně se provede nadefinovaná akce).

Vlastní předefinování obsluhy:

- použijí se funkce `signal` (základní použití: jaký signál chci obsluhovat a jakou funkcí jej chci obsloužit; funkce má jediný parametr – číslo signálu)
- nebo `sigaction` (určí se, jaká funkce bude obslužná + možnost nastavení blokování signálu během obsluhy, další speciální režimy, . . . )
- vice viz `man signal` nebo `man sigaction`

Přednastavené konstanty:

- SIG_DFL – přednastavený signál má být obsluhován implicitním způsobem
- SIG_IGN – signál má být ignorován

Z obslužné funkce je možné volat pouze bezpečné knihovní funkce (viz `man 7 signal`, na konci seznam funkcí, které se mohou použít při obsluze signálů).

#### 9.9.2 Blokování signálů

Je možné nastavit masku blokování signálu, volání:

- `sigprocmask` (řekneme, jaké nastavení signálů měníme),
- pomocí SIG_BLOCK (co chceme blokovat), SIG_UNBLOCK (odblokovat), SIG_SETMASK (natvrdo nastavit masku blokovaných signálů).

Blokování se řeší pomocí **bitových masek**, které jsou typu `sigset_t`. K vytváření masek je možné použít předdefinovaná makra `sigemptyset`, `sigfillset`, `segaddset`, `sigdelset`. Nelze blokovat signály **SIGKILL, SIGSTOP, SIGCONT**.

**Nastavení blokování se dědí.** (Proces si nastaví blokování – `fork` – zdědí to i potomci; při `exec` obslužné funkce zanikají!). Pokud chceme zjistit, zda nějaké signály čekají, zavoláme `sigpending` (předá se ukazatel na masku signálů). Pokud nějaký signál je zablokován, ale přijde vícekrát, zapamatuje se jeho **výskyt pouze jednou** (neplatí pro *realtime signály*).

### 9.9.3 Zasílání signálů

Slouží k tomu volání `kill` (s parametry `pid`, komu chceme signál poslat, a číslo signálu, který zasíláme). Umožňuje zasílat signály:

- jednomu konkrétnímu procesu (`pid` kladné),
- skupině procesů (0 – ve skupině, ve které proces je),
- všem procesům, kterým proces může signál poslat (pid = $-1$ nebo zápornější číslo – pošle se dané skupině: pid = $-10 \Rightarrow$ všem v skupině 10).

Aby mohl proces zaslat signál jinému procesu, musí odpovídat **jeho UID, EUID či saved set-user-ID cílového procesu** (nelze posílat signály někomu jinému), případně se musí jednat o privilegovaného odesílatele (např. EUID=0 nebo CAP_KILL).

Může se použít i `sigqueue` pro volání s realtime signály.

### 9.9.4 Čekání na signál

Měli bychom na signály čekat **pasivně** (nikoli se aktivně neustále dokola ptát, zda signál už přišel). Buď to:

- jednoduché čekání `pause`,
- obvykle lepší zabezpečené čekání `sigsuspend` – je možné specifikovat masku signálů, které mají být blokovány po dobu čekání, a atomicky přepnout mezi signály, které jsou blokovány do začátku čekání a od začátku čekání (může se stát, že proběhne test, zda přišel nějaký signál a zjistí se, že nepřišel – začne se čekat, signál přijde mezi testem a začátkem čekání – čekání do nekonečna)

# 10

**Desátá přednáška:** Nové téma – synchronizace procesů.

## 10.1 Synchronizace procesů

Synchronizace slouží k tomu, aby procesy navzájem **nekolidovaly a v systému nedocházelo k nekonzistentním stavům či poruchám dat.** Aby při běhu procesu nedošlo k nekonzistenci dat, používá se synchronizace procesů, která zajišťuje správné pořadí provádění spolupracujících procesů.

Když jádro dává **časové kvantum nějakému procesu, znamená to, že dává procesu nějaký časovač**, a když doběhne, **spustí se přerušení**.

**Race condition:**
*Časově závislá chyba* (*race condition* nebo *souběh*) je chyba, která může vzniknout při přístupu ke sdíleným zdrojům, datům – vzniká při přístupu ke sdíleným zdrojům kvůli různému pořadí provádění jednotlivých paralelních výpočtů v systému.

## 10.2 Kritické sekce

Kritické sekce jsou **úseky kódu napříč různými procesy**, ve kterých **nesmí dojít k přepnutí kontextu** (nebo v nich nesmí být několik procesů zároveň), **aby nedošlo k chybě souběhu dat** (např. více procesů pracuje se stejnými [globálními] proměnnými). V programu může být více různých kritických sekcí.

Problémem kritické sekce rozumíme **problém zajištění konkrétní synchronizace procesů na množině sdílených kritických sekcí**, což zahrnuje:

- **vzájemné vyloučení** – nanejvýš 1 (či $k$) proces/ů může do kritické sekce vstoupit (nebo: nanejvýš 1 proces je v daném okamžiku v dané množině sdílených kritických sekcí),
- dostupnost kritických sekcí:
- pokud je kritická sekce volná, chceme do ní vstoupit (nebo: opakovaně volná v alespoň určitých okamžicích, proces nemůže neomezeně čekat na přístup k ní),
- je potřeba se totiž vyhnout:
  - **uváznutí** – deadlock, viz níže a později, *brace yourselves*,
  - **blokování** – situace, kdy sdílený prostředek je volný, proces na něj ale musí dlouho čekat,
  - **stárnutí** (hladovění) – proces se snaží vstoupit do kritické sekce, ale nikdy k tomu nedojde, protože ho plánovač nikdy nevybere ve správný okamžik.

## 10.3 Problémy vznikající na kritické sekci

*Data race* – **dochází k závodu mezi daty** – situace, kdy nastanou dva přístupy ke zdroji s výlučným přístupem ze dvou procesů bez synchronizace, alespoň jeden přístup je zápis.

- **Deadlock** viz 11.3, *wait for it…*
- **Stárnutí** je situace, kdy proces čeká na podmínku, která nemusí nastat – v případě kritické sekce je touto podmínkou umožnění vstupu do kritické sekce
- **Livelock** – speciální případ stárnutí – každý proces z určité neprázdné množiny procesů běží a vykonává nějaký kus kódu, ale opakovaně v něm žádá o nějaký zdroj s výlučným přístupem, který vlastní nějaký z procesů dané množiny, nikdy se nedostanou k tomu, aby udělaly, co chtěly (nikdy nedojde ke vstupu do kritické sekce) – podobné deadlocku, ale s aktivním čekáním (ten kód něco dělá),

- **Blokování při přístupu do kritické sekce** je situace, kdy proces, který žádá o vstup do kritické sekce, musí čekat, přestože je kritická sekce volná a ani o žádnou z dané množiny sdílených kritických sekcí žádný další proces nežádá.

## 10.4 Způsoby řešení problému kritické sekce

Musí dojít k **vzájemnému vyloučení** (nesmí dojít k data race) a musí zajišťovat **dostupnost kritické sekce** (minimálně nesmí dojít k deadlocku) a zároveň synchronizační prostředky musí být efektivní.

**Petersonův algoritmus (krátký popis):**

- omezen pro 2 procesy (existují i rozšíření pro více procesů),
- řeší synchronizace kritické sekce bez hardwaru,
- nebere ohled na to, jaká ta kritická sekce je,
- v kritické sekci může být maximálně 1 proces,
- procesy o sobě vzájemně neví (neví, co druhý proces dělá, pouze vědí, že druhý proces existuje),
- pracuje s sdílenými poli booleovských proměnných (na začátku `false`, `false`) a sdílenou proměnnou `turn = 0`,
- pole nám říká, který z procesů chce přistoupit do kritické sekce,
- `turn` nám říká, kdo je zrovna na tahu (před vstupem do kritické sekce),
- první proces chce přistoupit do kritické sekce – nastaví `turn` na `1-i` ($i$ je index v bool poli), na tahu je druhý proces,
- začne se provádět prázdný cyklus (cyklí – aktivně čeká – dokud je nastaven flag `1-i` a zároveň je na tahu ten druhý proces),
- proces 1 provede kritickou sekci a nastaví svůj bool na `false`

**Bakery algoritmus L. Lamporta:**

- vyloučení pro $n$ procesů ($n$ znám dopředu),
- přijdeš do pekařství (nebo spíš na poštu), vezmeš si lístek s číslem a čekáš, až na tebe dojde řada,
- před vstupem do kritické sekce získá proces přístupový lístek, jehož hodnota je větší než čísla přidělená již čekajícím procesům,
- pracuje s sdíleným booleovským polem příznaků (jestli hledám hodnotu maximálního ticketu), sdílenou int hodnotou ticketu a lokálními int hodnotami,
- v první fázi se snaží algoritmus zjistit, jaké je největší číslo ticketu, a poté si vzít ticket s hodnotou $max + 1$,
- dva (nebo) více procesů může mít stejnou hodnotu ticketu,
- poté se nastaví bool na falše,
- ve druhé fázi proces čeká, až na něj přijde řada,
- nejprve počká, až jiný proces (od 0) dohledá svoje maximum,
- poté se čeká, až bude na mě řada (ticket kladný, ostatní procesy nebudou chtít přistoupit do kritické sekce),
- až proces projde do kritické sekce, hodnota jeho ticketu se nastaví na 0,
- v okamžiku, kdy mají procesy stejné číslo ticketu, přednost má proces s nižším číslem procesu (PID),
- jedním problémem algoritmu je neustálé zvyšování čísel (nutnost si dávat pozor na přetečení)

Ani jeden z předešlých algoritmů nemusí na dnešních CPU fungovat, protože jsou **silně závislé na pořadí přístupu do paměti**.

## 10.5 Využití atomických instrukcí pro synchronizaci

Některé přístupy k synchronizaci jsou založené na využití atomických instrukcí. To jsou instrukce, které jsou vždy provedeny najednou a nemohou být uprostřed své činnost přerušeny. Jejich atomicita je zajištěna HW.

**TestAndSet:**

- v Intelu `lock bts`, atomická instrukce,
- lze si ji představit jako funkci, co vrací bool a potřebuje odkaz na `target` (bool),
- nejdřív si uloží to, co je v paměti, do pomocné proměnné,
- poté na místo v paměti uloží `true`,
- vrátí, co tam bylo předtím (původní hodnotu),
- při synchronizaci se to dá využít například tak,
- že proces čeká, až na daném paměťovém místě bude hodnota `false`,
- projde do kritické sekce,
- nastaví proměnnou na `false` a může do ní přistoupit někdo jiný,
- kritická sekce je chráněna tzv. zámkem

**Swap:**

- v Intelu `lock xchg`, atomická instrukce,
- nic nevrací, vymění atomicky hodnoty ve dvou místech v paměti,
- při syncu se dá využít tak, že se zamkne kritická sekce prohozením hodnot dvou proměnných,
- vstoupíme do kritické sekce, a poté zámek odemkneme

Uvedená řešení zahrnují možnost **aktivního čekání**, a proto se také označují často jako **spinlock** (neustále se točí dokola a ptají se na platnost podmínky). Obecně jsou příliš drahé – procesy provádějí neužitečný kód, pouze berou CPU čas.

Lze je však využít **na krátkých, neblokujících kritických sekcích bez preempce** (tam, kde není přepnutí CPU).

Přístup do paměti (RAM) od CPU trvá kolem 100–150 cyklů. Opakovaný zápis sdíleného paměťového místa je problematický z hlediska **zajištění konzistence cache v multiprocesorových systémech** (zatěžuje se sdílená paměťová sběrnice) – řešením je při aktivním čekání **pouze číst**.

## 10.6 Semafory

Semafor je synchronizační nástroj nevyžadující aktivní čekání (nebo alespoň minimalizující- – může se vyskytnout uvnitř implementace operací nad semaforem). Jedná se o **celočíselnou proměnnou dostupnou pouze dvěma základními atomickými operacemi**:

- `lock` (také `decrement`, `wait` nebo `P`) – zamknutí semaforu, proces čeká než může vstoupit do kritické sekce, vstoupí do ní, ostatní procesy budou čekat,
- `unlock` (také `increment`, `signal` nebo `V`, v POSIXu `sem_post`) – odemknutí semaforu, jeden (či více) z čekajících procesů se probudí a dostane se do kritické sekce.

Aktuální hodnota $S$ semaforu značí, kolik procesu do něj ještě může vstoupit.

- $S$ je kladné – odemknuto (hodnota = kolik procesu ještě může vstoupit do kritické sekce),
- $S$ je záporné – zamknuto (absolutní hodnota $S$ udává počet čekajících procesů)

Dále je možné mít rozšíření semaforu o:

- neblokující zamknutí – pokud je možné zamknout semafor, tak se to provede, jinak se nebude čekat a provede se jiná akce,
- zamknutí s horní mezí na dobu čekání – maximální čekací doba (poté se proces probudí a bude dělat něco jiného),
- současné zamknutí více semaforů

Semafory spolu s OS zabezpečují sdílený přístup do kritické sekce tak, že **se čekající procesy uspí** (místo aktivního čekání) a po uvolnění se opět probudí. (Pozn.: **pořadí přístupů procesů není garantováno**, tzn. při uspání procesů se netvoří fronta, ale spíše nějaká množina procesů.)

**Práce se semafory:**

- vytvoření proměnné typu `semaphore` (sdílený semafor),
- inicializace semaforů,
- před vstupem do kritické sekce zámek semaforů,
- po výstupu z kritické sekce odemčení semaforu

Provádění locku a unlocku **musí být atomické** (jejich tělo představuje taky kritickou sekci). Atomicita lock a unlock se řeší:

- zákazem přerušení,
- vzájemným vyloučením s využitím atomických instrukcí a aktivního čekáním – s využitím spinlocku (používá se u multiprocesorových systémů, čeká se pouze na vstup do lock/unlock – krátkou dobu)

Používají se také:

- read-write zámky – pro čtení lze zamknout vícenásobně,
- reentrantní zámky – proces může stejný zámek zamknout vícekrát,
- mutexy – binární semafory, mohou být odemknuty pouze tím, kdo ho zamkl,
- futexy – rychlé mutexy používané v Linuxu v user-space (při detekci konfliktu se volá služba jádra)

**Implementace semaforu:** Implementace semaforu (konkrétně `lock` a `unlock`) tvoří další kritické sekce, které se také musí zabezpečit (zamknout) **jiným synchronizačním prostředkem**. Používá se spinlock na začátku obou funkcí, následně odemknutí spinlocku před výstupem z nich.

U zkoušky se implementuje semafor doplněný o kód spinlocku (TestAndSet nebo swap). Zde spinlock (na začátku) pouze chrání kritickou sekci lock a unlock – aktivní čekání se zde dá tolerovat.

```
typedef struct {
  int value;
  process_queue *queue;
} semaphore;

lock(S) {
  S.value--;
  if (S.value < 0) {
    // remove the process calling lock(S) from the ready queue
    C = get(ready_queue);
    // add the process calling lock(S) to S.queue
    append(S.queue, C);
    // switch context, the current process has to wait to get
    // back to the ready queue
    switch();
  }
}

unlock(S) {
  S.value++;
  if (S.value <= 0) {
    // get and remove the first waiting process from S.queue
    P = get(S.queue);
    // enable further execution of P by adding it into
    // the ready queue
    append(ready_queue, P);
  }
}
```

Obrázek 6: Z prezentace IOS: Synchronizace procesů – kod implementace semaforu

# 11

**Jedenáctá přednáška:** Pokračování a dokončení synchronizace procesů, monitory, deadlock.

## 11.1 Monitory

Synchronizační prostředky (ještě) vyšší úrovně (než semafory). Problém semaforů v reálném kódu je, že **je zde spousta sdílených dat, která se budou vzájemně vylučovat** – nechceme mít celý program zamknutý – bude zde snaha zamknutí minimalizovat – může se stát, že se někde v nějaké větvi **zapomene lock/unlock a nastane problém** (v reálném kódu tak semafory jednoduché nejsou).

Proto vznikl komfortnější synchronizační mechanismus – monitory. V syntaktické podobě vypadají takto (jazyk Ada, běžně se takto nepoužívají):

```
monitor monitor-name {
  shared variable declarations

  procedure body P1 (...) {
    ...
  }
  procedure body P2 (...) {
    ...
  }
  {
    initialization code
  }
}
```

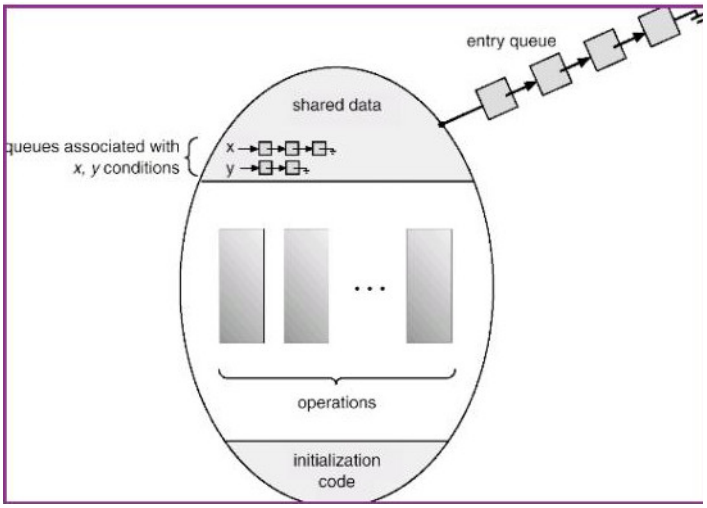
Obrázek 7: Z prezentace IOS: Synchronizace procesů – kód monitoru

**Monitor:**

- je možné si představit jako datovou strukturu podobné třídě se sdílenými proměnnými,
- ty jsou sdíleny různými procesy pracující s monitorem,
- procesy se mohou k proměnným dostat přes procedury (metody monitoru),
- automatický monitor zajišťuje, že na něm v daném okamžiku poběží nejvýše 1 metoda,
- kdokoli chce pracovat se sdílenými zdroji, musí používat tyto metody, a tak se automaticky zajistí zamykání a odemykání zdrojů,
- použití – do monitoru se nadefinují sdílené zdroje, dále mechanismy, jak se s nimi má pracovat (procedury), dále konstruktor (inicializační kód)

**Popis monitoru:**

- slupka (okraje) představují ochrannou bariéru monitoru,
- do monitoru je možné vstoupit definovanými operacemi (dveře),
- pokaždé, když do monitoru někdo vstoupí, tak se dveře uzamknou,
- další „zájemci o vstup“ jsou zařazení do čekací fronty vně monitoru (zařazování tím, že volají dané operace v okamžiku zamknutého monitoru),
- pokud se chtějí synchronizovat, použijí buď předdefinované podmínky programátorem, s každou z těch podmínek se pojí další čekací fronta (chci se sync pomocí podmínky x – wait_x() – zařazení do fronty x – současně bude vybrán někdo, kdo čeká na vstup monitoru a bude možnost v něm běžet – já čekám ve frontě x na někoho, kdo se do monitoru dostane a provede `signal` či `notify` – může mě uvolnit z fronty x)




Obrázek 8: Z prezentace IOS: Synchronizace procesů – grafické ztvárnění monitoru

**Co když se procesy chtějí synchronizovat uvnitř monitoru?**

- používá se mechanismus podmínek – conditions,
- speciální datová struktura s definovanými operacemi nad ní, pomocí nich se procesy synchronizují,
- je možné volat `wait()` – synchronizovat se s někým na nějaké podmínce,
- je možné někomu, kdo na podmínce čeká, poslat uvolňující signál (operace `signal()` nebo `notify()`)

**`signal` a `notify`:**

- v monitoru může běžet jen jeden proces,
- pokud v něm jsou dva (z toho jeden ve frontě dané podmínky – např. proces ve frontě x, druhý v monitoru a volá signal/notify):
  - `signal()` – po zavolání pokračuje ten, kdo signál dostane (ten kdo je ve frontě, např. x) a běží v monitoru, druhý proces se buď zařadí do čekací fronty, nebo se zařadí do další fronty uvnitř monitoru, kde takovéto procesy čekají (a mají přednost před procesy ve frontě venku)
  - `notify()` – po zavolání pokračuje volající, resp. ten, kdo signál dostal (ten, co byl ve frontě, se přesune buď do fronty mimo monitor, či do jiné fronty uvolněných, ale čekajících procesů)
- čeká se, až bude proces odejde z monitoru nebo začne čekat (zařadí se do fronty podmínky)
- pokud nikdo na podmínce nečeká a někdo na ni použije `signal` či `notify`, jedná se o prázdnou operaci (nic se neprovede)

Monitory je možné implementovat **s využitím semaforu** (vstupní semafory, semafory ke každé podmínce, pro prioritní procesy – pozastavené procesy ve frontě uvnitř monitoru). Monitory jsou dostupné v Javě či C#. V POSIXu (C, C++, . . . ) jsou k dispozici alespoň podmínky (kombinace semaforů a podmínek) – `pthread_cont_t` a funkce `pthread_cond_wait/signal/broadcast`.

## 11.2 Některé klasické synchronizační problémy

### 11.2.1 Problém producenta a konzumenta

- máme smečku procesů, předem rozdělených, či se dynamicky dělí na producenty a konzumenty, komunikují spolu přes vyrovnávací paměť (např. pole s kruhovým bufferem),
- nutné procesy synchronizovat pro správnou práci s pamětí,
- řešením jsou tři semafory (full, empty, mutex),
- full – říká, kolik položek je aktuálně v cache k dispozici, zamykáním si rezervují položku v paměti ke spotřebě,
- empty – říká, kolik je volných slotů v cache, zamykáním se rezervuje kapacita pro zápis do cache,
- mutex – pomocný semafor, který se zamkne při práci s ukazovátky do cache,
- **semafory je nutné vhodně inicializovat** (full=0, empty=max. hodnota cache, mutex=1)

```
semaphore full, empty, mutex;

// Initialization:
init(full,0);
init(empty,N);
init(mutex,1);
```

Obrázek 9: Z prezentace IOS: Synchronizace procesů – synchronizační prostředky producentů a konzumentů

**Pseudokód producenta:**

- pracují tak, že vyprodukují položku, kterou chtějí zapsat do cache,
- nejprve zamknou empty (pokud se to podaří – je tam volné místo a rezervují si ho pro sebe),
- zamknou mutex,
- začnou zapisovat do cache (posunou ukazovátko),
- odemkne se přístup do cache – odemknutí mutex a full (konzumentům se dá najevo, že je zde položka ke konzumaci),
- pracuje v nekonečném cyklu `do() {...} while(1)`

**Pseudokod konzumenta:**

- nejprve zamknou full (při úspěchu – ve vyrovnávací paměti je něco ke konzumaci, položku si alokovali),
- zamknou si přístup k cache (zamkne se mutex),
- z vyrovnávací paměti se odstraní 1 položka,
- odemkne se přístup k cache (mutex) a následně se odemkne empty (dá se najevo producentům, že se v cache uvolnila položka),
- zkonzumuje se položka,
- pracuje v nekonečném cyklu `do() {...} while(1)`

– Producent:

```
do {
   ...
   // produce an item I
   ...
   lock(empty);
   lock(mutex);
   ...
   // add I to buffer
   ...
   unlock(mutex);
   unlock(full);
} while (1);
```

– Konzument:

```
do {
   lock(full)
   lock(mutex);
   ...
   // remove I from buffer
   ...
   unlock(mutex);
   unlock(empty);
   ...
   // consume I
   ...
} while (1);
```

Obrázek 10: Z prezentace IOS: Synchronizace procesů – pseudokód producenta a konzumenta

### 11.2.2 Problém čtenářů a písařů

- dva typy procesů – čtenáři a písaři,
- pracují se sdílenou pamětí – čtenáři ji mohou pouze číst, písaři jen měnit,
- může současně číst libovolný počet čtenářů (nemění se obsah paměti),
- pokud zapisuje nějaký písař, nesmí nikdo ani číst, ani psát,
- používá se sdílená proměnná readcount – počet čtenářů,
- nutné použít semafor mutex (chránící přístup k readcount) a semafor wrt (semafor písařů),
- opět je nutná inicializace (počet čtenářů=0, oba semafory=1)

```
int readcount;
semaphore mutex, wrt;

// Initialization:
readcount=0;
init(mutex,1);
init(wrt,1);
```

Obrázek 11: Z prezentace IOS: Synchronizace procesů – synchronizační prostředky čtenářů a písařů

**Pseudokód písaře:**

- pokud chce zapisovat, zamkne semafor wrt,
- až se mu to podaří, zapisuje,
- po dopsání odemkne wrt,
- pracuje v `do() {...} while(1)`

**Pseudokód čtenáře:**

- v okamžiku, kdy chce číst, je nutné zamknout přístup k proměnné readcount (mutex), poté ji inkrementovat (zjistí tak, jestli je první),
- pokud readcount==1, tak se jedná o prvního čtenáře,
- potom se zamkne wrt a odemkne se mutex (readcount),
- začne číst (pokud přijde další čtenář, pouze readcount inkrementuje a čte),

- jakmile čtenář dočte, zamkne mutex,
- dekrementuje se readcount, pokd je ==0, znamená to, že je posledním čtenářem a musí odemknout přístup písařům (wrt),
- odemkne se přístup k readcount (mutex),
- pracuje v `do() {...} while(1)`

– Čtenář:

```
do {
  lock(mutex);
  readcount++;
  if (readcount == 1)
    lock(wrt);
  unlock(mutex);
  ...
  // reading is performed
  ...
  lock(mutex);
  readcount--;
  if (readcount == 0)
    unlock(wrt);
  unlock(mutex);
  ...
} while (1);
```

– Písař:

```
do {
  ...
  lock(wrt);
  ...
  // writing is performed
  ...
  unlock(wrt);
  ...
} while (1);
```

Obrázek 12: Z prezentace IOS: Synchronizace procesů – pseudokód čtenářů a písařů

Toto řešení má nevýhodu – **hrozí vyhladovění písařů** – pokud do kritické sekce vejde 1 čtenář, přijde písař a začne čekat, přijde další čtenář, vejde do kritické sekce, první odejde, druhý taky, ale opět vejde první čtenář – do nekonečna se tu střídají a písař nikdy nezapíše. Vyhladovění je někdy tolerováno, ale je dobré se mu vyhnout (malá pravděpodobnost, že se čtenáři budou do nekonečna střídat).

**Řešení vyhladovění:**

- použije se další semafor – wrt_waiting – čekající písař,
- písař nejprve zamkne wrt_waiting, poté wrt, po zamknutí wrt odemkne wrt_waiting,
- čtenáři na začátku provedou lock wrt_waiting, unlock wrt_waiting a poté pokračují dál (zajistí, že pokud nějaký písař začne čekat, tak všichni čtenáři dočtou, a po návratu na začátek nebudou schopni provést lock a unlock)

### 11.2.3 Problém večeřících filozofů

- 5 filozofů, kteří reprezentují procesy,
- sednou se kolem kulatého stolu, chtějí jíst a přemýšlet; jedí pomocí asijských hůlek, které budou mít rozděleny tak, že mezi každými filosofy je jedna hůlka, přičemž k jezení potřebují dvě – 5 hůlek pro 5 filozofů,
- cyklus: získá si svojí levou a pravou hůlku, může se najíst, položí hůlky zpět, přemýšlí, poté opět jí,
- reprezentuje situaci, kdy mezi synchronizovanými procesy je cyklická závislost

**Možnost řešení:**

- ne dokonalé řešení, možnost uváznutí (=špatné) a
- je možné, že dva filozofové budou jíst současně 1 hůlkou,

- zavede se 5 binárních semaforů (pole semaforů),
- všechny inicializované tak, že jsou odemknuté

**Filozof I:**

- $i$ = ID, hodnota 0–4,
- zamkne hůlku $i$, následně zamkne hůlku $(i + 1)\ mod\ 5$,
- pokud se podaří zámek obou hůlek, nají se,
- následně odemkne obě hůlky,
- může přemýšlet,
- `do() {...} while(1)`

```
semaphore chopstick[5];

// Initialization:
for (int i=0; i<5; i++) init(chopstick[i],1);

// Philospher i:
do {
  lock(chopstick[i])
  lock(chopstick[(i+1) % 5])
  ...
  // eat
  ...
  unlock(chopstick[i]);
  unlock(chopstick[(i+1) % 5]);
  ...
  // think
  ...
} while (1);
```

Obrázek 13: Z prezentace IOS: Synchronizace procesu – pseudokód (špatného) řešení problému večeřících filozofů

**Gde deadlock?**

- procesy se při plánování se mohou naskládat tak,
- že si každý vezme svou levou hůlku (každý hůlku se stejným číslem jako jeho ID),
- následně si každý z nich bude chtít získat tu druhou,
- každý bude trvat na tom, že si svoji nechá a chce tu druhou,
- celý systém se zastaví – deadlock

**Řešení:**

- jednou z možností je získávat obě hůlky současně,
- např. pomocí SysV semaforů, umožňují zamknout pole semaforů,
- další možností je získávat hůlky asymetricky – alespoň 1 z filozofů bude brát hůlky v obráceném pořadí (např. každý lichý filozof bude brát hůlky nejprve $i$ a poté $(i + 1)\ mod\ 5$ a sudí nejprve $(i + 1)\ mod\ 5$ a potom $i$)

## 11.3 Deadlock (uváznutí)

### 11.3.1 Definice

**Uváznutím (deadlockem) při přístupu ke zdrojům s výlučným (omezeným) přístupem rozumíme situaci, kdy každý proces z určité neprázdné množiny procesů je pozastaven a čeká na uvolnění nějakého zdroje s výlučným (omezeným) přístupem vlastněného nějakým procesem z dané množiny, který jediný může tento zdroj uvolnit, a to až po dokončení práce s ním.**

$$\exists P \subseteq \mathbb{P} : (P \neq \emptyset \wedge \forall p \in P\ (\text{pozastaven}(p) \wedge \exists r \in \mathbb{R} : (\text{čeká}(p, r) \wedge \exists p' \in P : (\text{vlastní}(p', r) \wedge \text{uvolnitelný\_pouze\_vlastníkem\_po\_dokončení\_práce}(r)))))$$

**Vysvětlení definice:**

- neprázdná množina procesů = $\{1, 2, 1000, \dots\}$, aspoň jeden, ale ne nula! – používáme tuto definici, jiné používají množinu procesů, kde jsou alespoň dva[1],
- výlučný přístup = zdroj, který používá v daném okamžiku nanejvýš 1 proces,
- omezený přístup = zobecněné kritické sekce, kde zdroj může používat určitý počet procesů, ale ne neomezený počet,
- pozastaven = není tam aktivní čekání, neběží (pokud to tam nebude, nebylo by možné rozlišit deadlock a livelock),
- zdroj vlastněný nějakým procesem z dané množiny = obvykle jiný, může to být i stejný proces,
- jediný ... po dokončení jeho použití = pracujeme s tím, že se používá zámek, který se používá neomezeně (žádný časový zámek), nikdo nemůže zdroj vzít, odemknout, nikde není žádný časovač

**Obecnější definice** (s možností uváznutí i bez prostředků s výlučným přístupem – např. zasílání zpráv):
Uváznutím rozumíme situaci, kdy každý proces z nějaké neprázdné množiny procesů je pozastaven a čeká na nějakou událost, která by mohla nastat pouze tehdy, pokud by mohl pokračovat některý z procesů z dané množiny. (Je to např. situace, kdy proces 1 čeká na zprávu od procesu 2, ten čeká na zprávu od 3, 3 čeká na 4, ... proces 5 čeká na zprávu od 1.)

### 11.3.2 Typický příklad deadlocku

- při přístupu ke zdrojům s výlučným omezeným přístupem,
- v praxi mohou být jednotlivá volání v kódu velmi daleko od sebe a zamykány mohou být jen za určitých podmínek,
- uváznutí se tak projeví jen zřídka a špatně se odhaluje a ladí

[1] Za deadlock se může považovat i situaci vzniklou při *doublelockingu* – self deadlock, proces zamkne zdroj a pak se jej pokusí zamknout ještě jednou.

```
semaphore mutex1, mutex2;

init(mutex1,1);  // Initialization:
init(mutex2,1);
...
// Process 1              // Process 2
lock(mutex1);             lock(mutex2);
...                       ...
lock(mutex2);             lock(mutex1);
```

Obrázek 14: Z prezentace IOS: Synchronizace procesů – příklad uváznutí

**Popis:**

- dva semafory, oba na začátku odemknuté,
- za předpokladu použití standardních semaforů se standardní sémantikou, není zde jádro, které např. hlídá situaci a jeden z procesů by zabilo, ...
- jeden proces zamkne první mutex (semafor), něco provede, pokusí se zamknout druhý,
- druhý proces zamkne druhý mutex, něco provede a zamkne první,
- nastane deadlock

### 11.3.3 Coffmanovy podmínky

**K tomu, aby uváznutí nastalo (mohlo nastat), je nutné a postačující splnění čtyř podmínek:**

- vzájemné vyloučení při používání prostředků,
- vlastnictví alespoň jednoho zdroje, pozastavení a čekání na další,
- prostředky může vrátit (pouze) ten proces, který je vlastní, a to po dokončení jejich využití (zdroj je vlastněn procesem, pouze ten proces ho může vrátit a to po dokončení využití toho zdroje),
- cyklická závislost na sebe (navzájem) čekajících procesů (rozumí se tím to, že jeden proces čeká na druhý – žádné aktivní čekání v cyklu)

### 11.3.4 Řešení uváznutí

- prevence uváznutí,
- vyhýbání se uváznutí,
- detekce a zotavení

Základní společná myšlenka všech řešení je **princip, kterým se snaží zrušit platnost alespoň jedné z nutných podmínek k uváznutí** – když se jedna z Coffmanovych podmínek zneplatní, nemůže dojít k uváznutí.

### 11.3.5 Prevence uváznutí

**Řeší, jak tyto podmínky zneplatnit:**

- první (vyloučení)
  - nepoužívat žádné sdílené prostředky, nebo
  - užívat pouze takové sdílené prostředky, které umožňují skutečně současný sdílený přístup
  - a u kterých není nutné vzájemné vyloučení procesu;
  - ne vždy problém půjde vyřešit tímto způsobem.

- druhá (vlastnictví)
  - proces může žádat o prostředky pouze tehdy, pokud žádné nevlastní,
  - (když chce proces používat současně 5 zdrojů v jednom okamžiku, musí zamknout všechny současně – získá, nebo čeká – systematicky v kódu zamykám všechny zdroje, nebo žádný zdroj – kontrola zámku: pokud proces již něco zamkl, nemůže znovu zamykat – musí vše uvolnit a teprve poté může zamknout),
  - nevýhodou je nutnost zamknutí všech zdrojů, které se používají současně (např. 30 min se používá 1 zdroj, poté 1 min se používá další zdroj současně s prvním, ale musí se zamknout na 30 min oba zdroje),
- třetí (návrat)
  - nebudu vyžadovat, aby proces vždy získal všechny zdroje, které chce používat současně v 1 okamžik,
  - ale umožním mu získávat postupně další a další zdroje, postupně přizamykat tyto zdroje,
  - ale jen v situaci, kdy je opravdu možné tyto zdroje získat,
  - pokud se proces pokusí něco přizamknout a ono se to nepovede, bude to řešeno speciálním způsobem
  - (např. tak, že proces bude zabit a všechny zdroje mu budou odebrány),
  - nevýhodou je situace, kdy v okamžiku, kdy se proces zabije, už mohl s některými zdroji něco dělat, zdroje mohou být v nekonzistentním stavu (ideálně se pokusit vrátit zdroje do konzistentního stavu),
- čtvrtá (cyklická závislost)
  - zavedením uspořádáním nad zdroji (očíslování),
  - je možné tyto zdroje získávat pouze od určitého pořadí (např. od nejmenšího k největšímu – začnu 1, pak 2, pak 3, . . . – pokud zamknu 5, můžu zamknout 5+, ale 3 už ne),
  - buď se konstrukcí programu zajistí, aby se vždy zamykalo v tomto pořadí, a poté analýzou, auditem se ověří, zda opravdu má program takovou zamykací disciplínu,
  - anebo budu mít systém zamykání, který toto bude kontrolovat

Ke všem bodům se dá říct, že buď bude program navrhován tak, aby byl konformní s použitou strategií, nebo se vynucení strategie kontrolovat až při běhu (což ale není jednoduché a spolehlivé).

### 11.3.6 Vyhýbání se uváznutí

Dalo by se to se chápat jako prevence 4. Coffmanovy podmínky.

**Obecný princip:**

- procesy musí předem deklarovat (jádru, systému správy zdrojů) informace o tom, jaké zdroje a jak budou používat, resp. které zdroje se budou používat a kolik jednotek zdroje se bude používat – každý proces musí říct, kolik jednotek kterého zdroje bude používat (a další informace, např. „budu používat zdroje 1, 2, 3, 4, nikdy nebudu používat současně 1 a 2, apod.“),
- systém přidělování zdrojů si vede informace, co ty procesy deklarovaly,
- o jejich možných požadavcích,
- současně si vede informace o aktuálním stavu přidělování (ví, kdo co vlastní a kdo o co požádal),
- v okamžiku, kdy systém má neuspokojené žádosti, je uspokojí tehdy, pokud nemůže vzniknout žádná cyklická závislost na sebe čekajících procesů, ani v tom nejhorším možném případu, který by mohl nastat s ohledem na to, co procesy deklarovaly

**Příklad:**

- situace v bance,
- od klientů mám svěřené peníze na termínované vklady, vím, kdy si klienti mohou své vklady vybrat,
- přijde někdo a bude chtít půjčku,

- je nutné se zamyslet, kolik zdrojů mám a kdy si to klienti mohou vybrat,
- půjčku mohu dát jen tehdy, když v nejhorším možném případě (všichni klienti si půjdou vybrat, co mohou) se nedostanu do dluhů

**Algoritmus založený na grafu alokace zdrojů:**

- řeší problém vyhýbání se uváznutí,
- pouze systémy, kde má každý zdroj pouze jednu instanci,
- je veden systémem, který zdroje přiděluje, ten si průběžně udržuje graf vztahů mezi procesy a zdroji: dva typy uzlů (procesy a zdroje) a tři typy hran – hrany od zdroje k procesu (který zdroj je kým vlastněn), hrany od procesu ke zdroji (kdo o který proces *žádá* a kdo o který zdroj *může požádat*),
- zdroj je přidělen pouze tehdy, pokud systém posoudí, že v budoucnu nemůže nastat cyklická závislost procesů, jednoduše tím, že „cvičně“ provede otočení žádostí na hranu vlastnictví (z P2 do R2 udělá R2 do P2), pokud v té situaci vznikne v grafu cyklus, znamená to, že v budoucnu by mohl vzniknout deadlock,
- v takovém okamžiku systém nepovolí přidělení zdroje (proces bude muset dál čekat na uvolnění zdroje, i když je volný)

Pokud by se používaly obecně zdroje se zobecněnou kapacitou, použil by se tzv. bankéřův algoritmus.

### 11.3.7 Detekce uváznutí a zotavení z něj

Systém přidělování zdrojů **umožní případný vznik uváznutí** (pokud pomineme to, že externě k množině uváznutých procesů máme *strážného anděla*, který uváznutí vyřeší), **ale periodicky se detekuje, jestli k uváznutí nedošlo** (běží speciální proces, zajišťující, že nebude přebit prioritami jiných procesů), a pokud ano, provede se zotavení.

**Detekce uváznutí:**

- vedení grafu vlastnictví zdrojů a čekání na zdroje (obdobný jako u vyhýbání se uváznutí) – stejný počet uzlů, 2 typy hran (někdo žádá o zdroj, někdo vlastní zdroj),
- pokud vznikne v grafu cyklus, vím, že uváznutí nastalo

**Zotavení z uváznutí:**

- alespoň některým procesům, které uvázly, odeberu zdroje,
- proces se buď zruší, nebo se pozastaví s tím, že může pokračovat, až bude moci získat všechny zdroje, které potřebuje,
- může nastat problém – procesy zabiju, mohou mít ve zdrojích rozpracované nějaké operace – zdroje mohou být v nekonzistentním stavu – nutnost buď nechat systém uváznutý, nebo se spokojit s nekonzistencemi, nebo systém navrhnout tak, že při zabití procesu se nezabije ihned, ale prvně provede zotavení (rollback – anuluje své operace a dostane zdroje do konzistentního stavu)

## 11.4 Formální verifikace, verifikace s formálními kořeny

Možnosti odhalování nežádoucího chování systému (uváznutí, stárnutí):

- inspekce systému – než se kód nasadí, kromě vývojáře kód musí projít i někdo další (či skupina), kteří schválí, že kód pochopili a je podle nich bezchybný,
- simulace, testování
  - vestavěné systémy – vytvoří si model systému a z něj se generuje kód, na modelu ověřují chování systému – nevýhodou testování na jedné jednotce (modelu) je to, že se chyba nemusí projevit (nedeterminismus se nemusí projevit)

  - paralelní programy – vkládání šumu do plánování (na kritická místa před ně se vloží náhodná zpoždění, přepnutí kontextu), zvýší se tím množství proložených aktivit a šance, že se projeví nějaká chyba,
  - dynamická analýza – sleduje se, co se děje v systému, poté je snaha extrapolovat (=odhadovat), co by se mohlo stát,
  - formální verifikace či verifikace s formálními kořeny – pokud se řekne, že program nemá chybu či určitou vlastnost, tak ji má s platností matematického důkazu
- obvykle se používají kombinace výše uvedených přístupů

Experimentuje se i s automatickými opravami – je zde **běhový systém, který monitoruje, co se děje**; pokud uvidí, že nastala chyba, **pokusí se ji opravit automaticky** (např. sledování data race conditions: při porušení vzájemného vyloučení při přístupu ke sdílené proměnné se automaticky přidá zámek – mohu ale tzv. pacienta zabít – způsobit uváznutí, proto se například místo toho před prací s danou proměnnou místo vkládání zámku vynutí přepnutí kontextu – zvýší se šance, že se projde kritickou sekci bez přepnutí kontextu)

**Proces formální verifikace:**

- vytvoření modelu – vytvoří se zdrojový kód či model, který se bude verifikovat (či kombinace, např. část jádra OS a model OS, odlehčená implementace),
- specifikace vlastností, které mají být ověřeny – mohou to být generické vlastnosti, např. v systému nesmí být deadlock, nebo složitější vlastnosti, např. v cache není nikdy více než 10 položek,
- kontrola (automatická), zda model splňuje specifikaci

**Ověřování se provádí metodami:**

- model checking (kontrola modelů),
- theorem proving (dokazování teorémů),
- static analysis (statická analýza)

Nad rámec verifikace se také provádí automatická analýza dle dané specifikace – dodá se specifikace, jaké vlastnosti systém má mít, pro určité třídy systému je možné automaticky vysyntetizovat korektní implementaci.

### 11.4.1 Theorem proving

- teorémem je zde věta, která říká: „Můj program XXX splňuje specifikaci YYY.“,
- používají se poloautomatické dokazovací prostředky (evidují, co už jste dokázali, mají databázi standardně platných skutečnosti z logiky, znají pravidla správného odvozování),
- vyžaduje se expert, který určuje, jak se důkaz má vést (přesto se to používá, např. mikrojádra ProvenCore či seL4),
- existují i plně automatické dokazovače (rozhodovací procedury, umí automaticky říct, zda platí, či ne) – obvykle pro omezené fragmenty logik, spíše se používají jako pomocné

### 11.4.2 Model checking

- obvykle plně automatizovaný přístup, prostředek,
- založen na systematickém generování stavu systému a stavového prostoru (systematicky se hledá, zda někde není chyba),
- nevýhodou je obrovský počet stavů – např. při $N$ dvoustavových procesech (procesy začnou a skončí) může vygenerovat $2^N$ stavů – problém stavové exploze

### 11.4.3 Static analysis

- snaha analyzovat a verifikovat systém na základě jeho zdrojového kódu, aniž by se tento kód prováděl (nebo alespoň ne v původní sémantice – např. celočíselné proměnné 0–1235, pamatuji si, jen jestli je hodnota záporná, 0, či kladná),
- nejjednodušší statický analyzátor je `grep` (znají se syntaktické vzory chyb a grepem se vyhledávají),
- má různé podoby: data flow analysis, constraint analysis, type analysis, abstract interpretation, symbolic execution, ...
- nástroje: Facebook Infer, Frama-C, Microsoft SDV, SpotBugs, cppcheck, ...

# 12

**Dvanáctá přednáška:** Začátek správy paměti.

## 12.1 Správa paměti

Aby program mohl byt proveden, musí být spuštěn – musí být nad ním vytvořen proces, musí mu být přidělen procesor a také paměť (a případně další zdroje – soubory, . . . ).

**Rozlišujeme:**

- logický adresový prostor – LAP – je virtuální adresový prostor, se kterým pracuje CPU při provádění kódu (uživatelského či jádra – každý proces i jádro mají své logické adresové prostory),
- fyzický adresový prostor – FAP – adresový prostor fyzických adres paměti (obsahuje adresy, které se umisťují na adresové sběrnici, chceme-li z paměti načíst nebo do ní zapsat – je společný pro všechny procesy i jádro)

Často logický adresový prostor **jádra bývá podprostorem logického adresového prostoru jednotlivých procesů**. Všechny logické adresové prostory procesu (v Linuxu) se překrývají ve stejné části – **v části, kde je LAP jádra**. LAP jádra **není** procesům přístupný. Výhodou je, že při přechodu z režimu procesu na režim jádra se **pouze zpřístupní tento LAP** nebo při přepínání procesů (kromě změn mapování části LAP procesu) se **nemusí měnit mapování pro část LAP jádra**.

**Proces pracuje s logickými adresami, ale na adresovou sběrnici se umisťují fyzické adresy**. Toto mapování provádí **MMU (Memory Management Unit):**

- HW jednotka specializovaná na překlad logických adres na fyzické,
- dnes běžnou součástí čipu CPU,
- provádí překlad na základě datových struktur,
- obsah struktur je částečně uložen ve speciálních registrech, částečně v hlavní paměti systému,
- součástí MMU je cache, obvykle tzv. **TLB**, pro urychlení překladu

**Komunikace CPU a paměti:**

- na CPU běží programy (CPU pracuje s logickými adresami),
- při čtení nebo zápisu z/na logickou adresu se adresa předá do MMU,
- MMU provede překlad logické adresy na fyzickou,
- MMU umístí fyzickou adresu na sběrnici a po ní se přenesou data mezi pamětí a CPU

## 12.2 Přidělování paměti

Existuje více úrovní přidělování paměti. V nejnižších (z hlediska blízkosti k HW) se přiděluje FAP pro zamapování do LAP; vyšší vrstvou je např. přidělování paměti přes knihovní funkce (`malloc`, mimo režim jádra, či `kmalloc`, `vmalloc` v jádře); ještě na vyšší úrovni pak např. přidělování v rámci aplikací.

Nejnižší úroveň přidělování je **implementována v jádře a jedná se o přidělování FAP pro zamapování LAP**. Běžné způsoby přidělování paměti (a mapování LAP na FAP):

- přidělování po spojitých blocích (contiguous memory allocation),
- segmentech,
- stránkách,
- kombinace výše uvedeného (Intel – segmenty a stránky)

**Funkce malloc:**

- při žádosti o alokaci nějakého kusu paměti (počtu bajtů)
- musí malloc požádat jádro o přidělení FAP,
- požaduje od jádra větší blok paměti (segment, stránka),
- z bloku paměti se vykousne požadovaný počet bajtů,
- ty dostane k dispozici uživatel,
- při volání dalších mallocu, dokud požadovaný počet bajtů nebude větší než přidělený blok, nepůjdou požadavky do jádra, ale bude se čerpat již přidělený prostor

## 12.3 Contiguous Memory Allocation

Mechanismus mapování logických adres na fyzické a přidělování paměti po spojitých blocích. Jedná se o **nejjednodušší mechanismus jak z hlediska HW, tak obsluhy OS**. V běžných výpočetních systémech se příliš **nepoužívá**, nicméně je vhodný pro jednoduché a vestavěné aplikace, které mají běžet na jednoduchém HW.

**Popis:**

- je to po spojitých blocích (něco jako spojité ukládání dat),
- k popisu takového mapování je třeba znát, ve kterém FAP je zamapován počátek LAP,
- je nutné vědět, jak je úsek paměti velký (pro odchycení přístupu mimo meze tohoto prostoru)

**Překlad adresy:**

- MMU si pro aktuálně běžící proces pamatuje 2 údaje,
- v limitním registru si pamatuje, kolik proces paměti dostal,
- v relokačním registru si pamatuje, na jakou fyzickou adresu byl zamapován LAP procesů,
- když dostane logickou adresu,
- zjistí, zda je adresa v rámci naalokovaného prostoru,
- pokud ne – chyba při přístupu do paměti, pošle se přerušení typu trap, obvykle je proces předčasně ukončen,
- pokud ano – mapování se provede tak, že se použije bázová adresa, na kterou je zamapován FAP procesů, sečte se s logickou adresou prostoru = mám adresu ve FAP (fyzickou adresu)

**Příklad překladu LA na FA:**

- proces 1 má začátek LAP na začátku FAP (konec na FAP 100 000), proces 2 někde uprostřed (LA 0 = FA 1 mil.),
- překlad LA 10 procesu 1,
- zkontroluje se, zda 10 je menší než 100 000 (= konec LAP procesu 1),
- bázová adresa 0 se přičte s LA, tedy 0 + 10 = 10,
- překlad LA 10 procesu 2,
- zkontroluje se, zda 10 je v rámci rozmezí (ano),
- bázová adresa 1 000 000 se přičte k 10, tedy 1 000 010

**Tento mechanismus má řadu nevýhod:**

- výrazně se zde projevuje externí fragmentace paměti (FAP),
- přidělováním a uvolňováním úseků paměti vzniká posloupnost obsazených a neobsazených úseku paměti, úseky mohou být obsazeny různými procesy,
- nejhorším dopadem je, že při sčítání volných úseků paměti může být prostor pro přidělení paměti procesu dostatečný, ale tyto volné úseky nejsou (nemusí být) spojité, takže zde zdánlivě není dostatek paměti pro daný proces (není možné provést alokaci),

- problémy se zvětšováním prostoru daného procesu,
- snaha o minimalizace dopadů externí fragmentace pomocí různých strategií (first fit se nepoužívá, namísto toho například best fit, worst fit či binary buddy, ...),
- provádí se dynamická reorganizace paměti (nákladné)
- není možné rozumně řídit přístupová práva v rámci přidělené paměti (nelze: část paměti pro čtení, část pro zápis, ...),
- není možné také sdílet část adresového prostoru (vše nebo nic),
- při virtualizaci paměti (swapování) je nutné odložit veškerou paměť na disk a poté ji vrátit zpět – pomalé, může být zbytečné.

### Definice:

***bázová adresa*** – počátek LAP (tj. adresa 0 v LAP) ve FAP (nebo: počáteční adresa LAP procesů ve FAP)

***first fit*** – prochází se volnými úseky a použije se první volný úsek

***best fit*** – podívám se na seznam volných úseků a vybere se ten, který je dostatečně velký, ale vyberu ten nejmenší z dostatečně velkých

***worst fit*** – paradoxně lepší jak best fit, opak best fit, hledá se úsek dostatečně velký a použije se ten největší (zbyde nepoužitý velký kus, který se využije později, např. při dalším zvětšování)

***binary buddy*** – udržuje se seznam volných úseků paměti, najdu si úsek paměti, který odpovídá nejlépe, a pokud přesahuje, resp. je 2x větší než požaduji, rozdělím ho na polovinu a opět zjistím, zda je úsek 2x větší, pokud ano, dělím úseky tak dlouho, až dojdu k úseku paměti, který nelze rozdělit na poloviny tak, aby byl uspokojen daný požadavek, a paměť se přidělí

## 12.4 Segmentace paměti

- LAP je rozdělen na kolekci segmentů,
- segmenty mohou být přiděleny překladačem/programátorem jednotlivým částem procesu (částem dat, procedurám, zásobníku, ...),
- každý segment má číslo a velikost,
- LA je číslo segmentu a posun v něm,
- jednotlivé segmenty patřící jednomu procesu nemusí být zamapovány spojitě (jeden segment bude zamapován spojitě, ale různé segmenty nemusí)

**Překlad adresy LA na FA:**

- MMU potřebuje pracovat s tabulkou segmentů, která je uložená v RAM – v MMU je odkaz na začátek tabulky (= pole),
- logická adresa je dělena na číslo segmentu $s$ a posuv v rámci segmentu $d$ (displacement),
- při překladu se vezme číslo segmentu $s$, např. při práci s $s = 10$ se podívá do řádku tabulky 10,
- na příslušném řádku se najde, jakou má segment velikost, jakou má bázovou adresu,
- pokud bude např. $d = 1000$, podívá se, jestli je d v rámci paměťového prostoru, který je přidělen,
- pokud ne – výjimka, chyba,
- pokud ano – vezme se bázová adresa segmentu, sečte se to s posuvem a mám FA

**Příklad překladu:**

- mám LA se segmentem $s = 10$ a posuvem $d = 1000$,
- přistoupím na 10. řádek tabulky segmentu,
- zde budu mít limit (velikost segmentu, zde např. 100 000) a jeho bázi (odpovídající počáteční adresu ve FAP),

- pokud $d$ je v limitu (1000 < 100 000?),
- vezmu bázi (např. 1 000 000), sečtu ji s $d$,
- tzn. 1 000 000 + 1000 a dostanu fyzickou adresu (1 001 000)

**Výhody:**

- mohou být použity jako jemnější jednotka ochrany při přístupu do paměti (některé mohou označeny jako pro čtení, některé pro zápis, některé v režimu jádra, ...),
- jemnější jednotka pro odkládání paměti na disk (odložit se mohou segmenty, ne celá paměť procesu),
- jemnější jednotka pro sdílení,
- implementace je jednoduchá,
- paměť je přidělována nespojitě, zmírňují se dopad externí fragmentace

**Nevýhody:**

- při zvětšování opět dopad externí fragmentace,
- možný zdroj chyb, segmentace je viditelná procesu

## 12.5 Stránkování

Je aktuálně nejpoužívanějším mechanismem mapování LAP na FAP. LAP je rozdělen na **jednotky pevné velikosti – stránky**, FAP je rozdělen na odpovídající **jednotky stejné velikosti – rámce**. (Nejčastěji bývá velikost stránky 4 KiB.)

### 12.5.1 Vlastnosti

**Výhody:**

- paměť je přidělována po rámcích (ty se zamapují do stránek),
- neviditelné pro uživatelské procesy,
- minimalizují se problémy s externí fragmentaci (podobně jako clustery u disků):
  - pořád vznikají úseky volných a využitých, nicméně nejmenší nevyužitá „díra“ v paměti je 1 rámec, ten se vždy dá využít (nespojité přidělování paměti),
  - je snaha přidělovat paměti po spojitých posloupnostech rámců (pokud je to možné), např. pomocí algoritmu binary buddy
- jemná jednotka ochrany přístupu do paměti (každá jednotlivá stránka může být r, rw, v uživatelském režimu či v režimu jádra, NX bit – jestli je možné obsah stránky spouštět jako kód),
- jemná jednotka sdílení (paměť sdílená mezi procesy lze sdílet pro stránkách),
- při nedostatku paměti se odkládá po jednotlivých stránkách

**Nevýhody:**

- složitější implementace,
- větší režie,
- interní fragmentace (podobně jako u disků),
- možné snížení rychlosti přístupu do paměti – nespojitá alokace, projeví se větším počtem kolizí v caches,
- zpomalování alokace a dealokace paměti (delší práce se strukturami, které popisují aktuální obsah paměti),
- jsou vnímány jako výrazně menší než výhody systému – proto jsou používány nejvíce

### 12.5.2 Mapování logických adres na fyzické

V nejjednodušším případě se používají **jednoduché – jednoúrovňové tabulky stránek**. OS udržuje **informaci o volných rámcích** (záleží na OS, HW nezajímá), pro každý **proces si udržuje tabulku stránek** (musí být strukturovaná tak, aby tomu daná architektura rozuměla).

### 12.5.3 Tabulky stránek

- logická adresa je rozdělena na číslo stránky ($p$ – place) a na posuv stránky ($d$ – displacement),
- číslo stránky se použije jako index do tabulky stránek (= pole v paměti),
- v MMU je registr, který bude ukazovat na to, kde má daný proces uloženou tabulku stránek,
- číslo stránky se vezme jako index do tabulky stránek ($p = 10 \Rightarrow$ přistoupím na 10. položku),
- pokud byla stránka alokována (= má přidělen rámec), na daném řádku najdu číslo rámce,
- číslo rámce se spojí s posuvem a dostanu FA

Na řádku tabulky stránek, který odpovídá dané stránce, kde je uloženo odpovídající číslo rámce, **jsou uložené řídicí příznaky mapování – platnosti mapování, přístupů, modifikace, přístupová práva** (r, rw, user režim, jádro režim, možnost provádění), **globality**.

Tabulky stránek jsou udržovány v **hlavní paměti (RAM)**, **zvlášť** pro každý proces, MMU mají ve speciálním registru **pouze ukazatel na začátek tabulky stránek**, při přepínání kontextu se **mění pouze ukazatel** na začátek tabulky stránek. (Konkrétně u Intelu se ten registr jmenuje *CR3*.)

Neprovedeme-li žádnou další optimalizaci, tak každý jednotlivý přístup do paměti (pro data či instrukce) se změní **z jednoho přístupu na 2** (při načtení dat z RAM – nejprve musím jít do tabulky stránek, poté načíst vlastní data – obojí jsou v RAM) – zpomalení o 100 procent. Používá se tak **vyrovnávací paměť' TLB – Translation Look-aside Buffer** pro urychlení práce s pamětí.

**Příklad překladu LA na FA:**

- LA, tvořená číslem stránky $p = 10$, posuvem $d = 1000$,
- MMU bude mít v paměti umístěnou tabulku stránek, v registru bude odkaz na adresu, kde se nachází tabulka stránek,
- MMU použije $p = 10$ jako index do tabulky stránek, přistoupí na řádek 10 tabulky stránek,
- součástí obsahu řádku je odpovídající číslo rámce (např. 500),
- dostaneme fyzickou adresu, tvořena rámcem $f = 500$ a posuvem $d = 1000$

## Definice:

***příznak platnosti mapování*** – zda daný adresový blok je, nebo není použit (ne všechny tabulky stránek v daném okamžiku musí být využity)

***příznak přístupu*** – byla stránka od okamžiku zavedení paměti zpřístupněna? (dále slouží jako informace, zda je stránka vhodná na odložení do paměti – jádro čas od času prochází bity a přístupové bity nuluje, při hledání kandidáta na odložení se zjišťuje, zda v několika periodách bylo ke stránce přistoupeno nebo ne – pokud delší dobu ne, bude odložena)

***příznak modifikace*** – byla, nebo nebyla stránka modifikována? (Při odložení stránky a poté znovuzavedení do paměti – aby se nemodifikovaná stránka neodložila znovu.)

***příznak globality*** – když je stránka globální, je sdílená za běhu různých procesů, typicky se používá pro stránky jádra

### 12.5.4 TLB

- obsahuje dvojice číslo stránky a číslo rámce + jsou tam některé řídicí příznaky spojené s mapováním (oprávnění, modifikace),
- v TLB nejsou celé stránky či rámce – je to cache pro mapování čísla stránky na číslo rámce),
- typicky implementovaná jako (částečně) asociativní paměť,

**Někdy je operace částečně asociativní:**

- několik bytů z LA je použito klasickým způsobem adresování,
- zbytek je použit pro asociativní vyhledávání,
- např. pokud se vymezí pro klasické adresování 2 byty, TLB se rozdělí na 4 části/bloky (00,01,10,11), v rámci bloku se bude hledat asociativně

**Překlad:**

- obdobně jako bez TLB,
- místo přístupu do paměti na určitý řádek tabulky bude HW hledat, zda někde v TLB je položka s číslem 10 a pokud ano, vezme k tomu odpovídající číslo rámce,
- pokud takové vyhledání nastane, říká se tomu TLB hit, použije se okamžitý překlad – číslo rámce a posuv a jsem v FAP,
- pokud se nepodaří úspěšně vyhledat položku, použije se stejný postup jako bez TLB (přes paměť) – TLB miss

**Příklad překladu s TLB:**

- architektura s jednoúrovňovou tabulkou stránek, k dispozici TLB, který je částečně asociativní, 2 bity se použijí pro rozlišení částí TLB a zbytek se prohledává asociativně,
- LA, rozdělená na číslo stránky $p$ – to na rozděleno na horní 2 bity (adresování v rámci TLB), zbytek bude číslo stránky (p=01 a zbytek 10), posuv $d = 1\ 000$,
- TLB rozděleno na 4 bloky, jeden z nich adresován bity 00, další 01, další 10, a poslední 11,
- p=01 10 mi říká, že mám jíst do 2. části,
- v bloku budou čísla stránek, odpovídající rámce,
- všechny řádky se prohledají paralelně – je v některém z těch řádků číslo 10 ?,
- ano – vezmu odpovídající rámec (např. 1 000 000) – TLB hit,
- FA bude $f = 1\ 000\ 000$ a posuv $d = 1\ 000$,
- ne – nastane TLB miss, je nutné jít do klasické tabulky stránek a tam hledat

K neúspěšnému vyhledání může dojít **při přístupu k instrukčnímu kódu** (čtení instrukce, operandů), **u každého čtení může dojít k neúspěšnému vyhledání opakovaně** (např. 32b architektura, 4B instrukce, nesprávně zarovnaná instrukce, 2 B na začátku 1. stránky, 2 B na začátku 2. stránky – je nutné provést překlad čísel na odpovídající rámec u obou stránek – může dojít 2x k TLB miss) – pro instrukci velkou 4 B, která načítá 4 B z paměti, jsou nutné 4 překlady a $4\times$ může nastat TLB miss.

**Pokud dojde k TLB miss:**

- HW bude hledat automaticky v tabulce stránek – platí u architektur, kde je TLB řízený hardwarem (většina architektur),
- jsou i softwarově řízené TLB; pokud nastane TLB miss, nastane přerušení od MMU k CPU, jádro si tento překlad provede, specializovanými instrukcemi se naplní překlad do TLB a překlad adresy se opakuje (CPU jako MIPS, SPARC)

Někdy může být použito více TLB, např. 64b architektury obvykle mají překlad náročnější – je potřeba více TLB (např. dvouúrovňová TLB – jedna pro překlad kódových adres, druhá pro adresy dat).

**Při přepnutí kontextu** (mění se hodnota ukazatele tabulky stránek v MMU registru) **je nutné invalidovat obsah TLB a znovu naplnit jej naplnit; používají se optimalizace:**

- používají se globální stránky (stránky používané jádrem – jsou na stejném místě),
- na některých CPU se ještě do řádku TLB doplňuje identifikátor procesu, pro který je mapování platné (vyhledává se na základě čísla procesů a čísla stránky),
- při změnu obsahu tabulek stránek je také nutná invalidace obsahu TLB (neprojevily by se změny v tabulce stránek)

Údaje se do (HW řízených) TLB dostávají tak, že **se překlad nahrává do TLB při prvním přístupu na stránku**, při nedostatku překladů se nějaký překlad odstraní. HW také **počítá s tím, že budeme s pamětí pracovat spojitě**, tak si může někdy nahrát do TLB překlady následujících adres **dopředu**. **V případě SW řízených TLB může být obsah nahráván speciálními instrukcemi jádrem**.

## Definice:

***asociativní paměť*** – také obsahem adresovatelná paměť; nefunguje jako pole, vyhledává podle (části) obsahu paralelním porovnáváním se všemi položkami

### 12.5.5 Efektivita stránkování s TLB

Efektivní přístupová doba je:

$$(\tau + \epsilon)\alpha + (2\tau + \epsilon)(1 - \alpha)$$

- kde $\tau$ je vybavovací doba RAM,
- $\epsilon$ je vybavovacídoba TLB,
- $\alpha$ je pravděpodobnost úspěšného vyhledání v TLB (TLB hit ratio, $1 - \alpha$ je tedy pravděpodobnost neúspěchu),
- jedna se vážený průměr, kde se spočítá doba přístupu do paměti, pokud se zadaří vyhledání $(\tau + \epsilon)\alpha$,
- pokud se nezadaří vyhledání, počítá se doba *dvou* přístupů do paměti $2\tau + \epsilon$

Tento vztah je sestaven za předpokladu, že poté, co se neúspěšně vyhledá v TLB, půjdu do RAM, najdu překlad v RAM, a poté půjdu do RAM pro data – v praxi to funguje tak, že pokud **MMU nalezne překlad v RAM, automaticky ho ihned doplní do TLB, provede opakovaný překlad pro TLB, a až poté jde pro data do paměti**. (V praxi by přístupová doba při TLB miss byla $2\tau + 2\epsilon + \delta$, kde $\delta$ je čas pro provedení úpravy TLB.)

Je **důležité, aby bylo vyhledávání TLB úspěšné, jinak se přístupová doba do paměti bude výrazně zpomalovat**. Úspěšnost TLB ovlivňuje:

- velikost TLB (to ovlivní výrobce čipu),
- dobrá lokalita odkazů programu (může ovlivnit programátor)

**Příklad:**

- inicializace čtvercové matice, NxN prvků,
- v paměti linearizována, ukládají se typicky *po řádcích*,
- inicializace 2 způsoby,
- první způsob inicializuje matici po sloupcích, druhý po řádcích,

- v případě přístupu po řádcích je v souladu s uložením paměti po řádcích – bude výrazně efektivnější z hlediska možného počtu neúspěšného vyhledání TLB; přístup po sloupcích bude méně efektivní,
- v případě TLB s jednou položkou a 2x2 matice, kde prvek zabírá půlku stránky, nastane při přístupu po řádcích 2x TLB miss, po sloupcích nastane 4x TLB miss

### Definice:

***lokalita odkazů*** udává, s kolika různými shluky adres (adresy blízko sebe, shluk = 1 stránka) pracuje v daném procesu za krátký časový okamžik program (pokud shluků adres není mnoho, program má dobrou lokalitu adres)

#### 12.5.6 Implementace tabulek stránek

Kdyby se tabulky stránek implementovaly jako jednoúrovňové, zabraly by příliš moc paměti.

Pro 32b systémy (Intel) se stránkami o velikosti 4 KiB (12 bitů se odkousne na posuv stránky), zbývá 20 bitů LA, udávají číslo stránky a počet řádků v tabulce stránek (odpovídá počtu stránek), odpovídá to >1 milionu položek. Má-li mít 1 položka tabulky stránek 4 B (je tam číslo stránky, rámce – 20 bitové, k tomu řídicí příznaky – celkem potřeba 27 bitů + zarovnání na bajty – 32 bitů), dostáváme tak 4 MiB pro jednu tabulku stránek pro každý proces (běžně může běžet 100 procesů – jen na tabulky stránek by bylo potřeba 400 MiB)

Pro 64b systémy je problém ještě horší – také se používají 4KiB stránky, teoreticky by šlo použít až 52 bitů na adresu stránky – pro položku bude potřeba 8 B.

#### 12.5.7 Hierarchické tabulky stránek

Nebude zde 1 tabulka stránek pro 1 proces, ale pro **1 proces bude více tabulek v hierarchické struktuře**, ne všechny dílčí tabulky musí být v daném okamžiku alokované (vznikají *tabulky tabulek tabulek… stránek*).

**Princip fungování** hierarchické tabulky na příkladu **dvouúrovňové tabulky stránek (používané u i386)**:

- 32b logická adresa, 12 dolních bitů je vymezeno na posuv v rámci stránky ($2^{12}$ = 4KiB velikost stránky),
- zbývajících 20 bitů je pro číslo stránky (rozděleno na horní a dolní část po 10b), používají se jako indexy do dílčí tabulky stránek 2. úrovně a následné dílčí tabulky stránek 1. úrovně,
- dílčí tabulka 2. úrovně je pro daný proces právě jedna – označuje se jako *adresář stránek*,
- v registru CR3 (v MMU na Intelu) je uložen odkaz na začátek adresáře stránek (= tabulka 2. úrovně, forma pole) pro aktuálně běžící proces,
- první úroveň tabulek stránek nemusí být použita příslušným příznakem v adresáři stránek (je možné říct, že příslušná položka adresáře stránek nepoužívá 1. úroveň – pracujeme s velkými stránkami – zde s posuvem 22 bitů = 4MiB stránky)

**Překlad LA na FA (na příkladu výše):**

- vezme se obsah tech horních 10 bitů (31–22), je to 10b číslo, které se použije jako *index* do adresáře stránek (tabulky 2. úrovně),
- index = číslo řádku,
- na této položce najdu *odkaz, čili adresu* začátku dílčí tabulky stránek 1. úrovně (těch může být víc),
- vezmu číslo uložené v dalších 10 bitech (21–12), použiju ho opět jako index do této tabulky,
- zde bude odpovídající číslo rámce,
- za předpokladu, že v části, kde jsou řídicí příznaky, je uvedeno, že mapování je platné,
- číslo rámce se vezme, přidá se k tomu posuv a mám FA

**Efektivita přístupu do paměti dvouúrovňové tabulky stránek:**

- z jednoho přístupu se stanou 3 přístupy,
- nejprve musím do adresáře stránek (první přístup),
- poté do dílčí tabulky stránek (druhý přístup),
- až poté mohu do paměti (třetí přístup),
- pokud nebude pracovat TLB, zpomalení bude o 200 procent

**Tabulky stránek na x86-64 systémech (čtyřúrovňové tabulky):**

- 64 bitové adresy, dvouúrovňové tabulky stránek nestačí,
- dolních 12 bitů je použito jako offset stránky ($\Rightarrow$ 4KiB stránky),
- číslo stránky má (47–12) 36 bitů,
- tedy máme 48 bitů logické adresy (+ je vyžadováno znaménkové rozšíření do 64 bitů, bit 47 se tedy musí opakovat až po bit 63),
- číslo stránky je rozděleno na 4 indexy (odspodu po 9 bitech index do 1. úrovně, 2. úrovně, 3., 4.),
- 1. úrovni se říká *dílčí tabulka stránek* (page table, PT),
- 2. úrovni se říká *adresář stránek* (page-directory table, PD),
- 3. úroveň je *tabulka ukazatelů* (page-directory-pointer table, PDP),
- 4. úroveň žádné extra jméno nemá (page-map-level-4 table, PML4)

**Překlad LA na FA ve čtyřúrovňové tabulce stránek na x86-64 systémech:**

- opět se používá registr CR3, ve kterém je uložen odkaz na začátek tabulky 4. úrovně. Adresa PML4 je v CR3 uložená mezi bity 51–12, protože fyzické adresy mohou mít maximálně 52 bitů (limit architektury) a PML4 má velikost jedné stránky ($2^{12}$ B, 4 KiB), spodních 12 bitů se tak používá na indexaci v ní,
- mezi bity 47–39 se vezme index do tabulky 4. úrovně, tam se najde odkaz na začátek tabulky 3. úrovně,
- použije se další index (bity 38–30), dostaneme odkaz na začátek tabulky 2. úrovně,
- použije se poslední index (20–12), dostaneme odkaz na dílčí tabulku stránek, tam dostaneme číslo rámce, k němu se přičte posuv,
- dostáváme fyzickou adresu,
- jeden řádek v tabulce zabírá zde 8 B, při 4KiB tabulce na adresaci jednoho řádku stačí 9 bitů (proto tady 9, ne 10 jako u 32b)

V x86-64 dojde **bez TLB ke zpomalení o 400 procent** – 5 přístupů do paměti. Konečná adresa nemusí být nutně až v poslední PT, s použitím nějakých příznaků je opět možné adresovat „velké stránky“, pak může být adresování:

- čtyřúrovňové – stránky o velikosti 4 KiB,
- tříúrovňové – stránky velké 2 MiB,
- dvouúrovňové – stránky velké 1 GiB (u některých procesorů)

### 12.5.8 Hierarchické tabulky stránek a TLB

Při použití hierarchických tabulek stránek zásadně **roste význam TLB**:

- na 64b CPU jsou větší, mají složitější organizaci,
- mají víceúrovňové TLB, bývá oddělena zvlášť pro datové stránky a instrukční stránky (i7 – dvě úrovně TLB, zvlášť je úroveň datová a instrukční, poté je společná TLB úrovně 2)

Další možnosti **optimalizace práce s TLB:**

- globální stránky – stránky jádra, které jsou sdílené mezi jednotlivými procesy, jsou označené jako globální a nemusí se vyhazovat při přepnutí kontextu,
- vstupy TLB spojené s identifikátory procesů – kromě čísla stránky, rámce bude na jednotlivých řádcích TLB také ID procesu,
- používá se spekulativní dopředné nahrávání překladu TLB,
- používají se specializované cache (kromě TLB) pro ukládání položek (úrovní) tabulek stránek

**Zanořené hierarchické tabulky stránek (Intel/AMD):**

- při virtualizaci vznikají zanořené tabulky stránek (tabulky stránek host OS, tabulky stránek virtuálního OS),
- v případě 64b procesorů se bude používat 8 úrovní tabulek stránek (4 pro VM, další 4 pro hosta),
- používá se tak odlišení položek v TLB používané na fyzickém stroji pro různé virtuální stromy – je zde identifikátor pro virtuální PC (VPID na Intelu nebo ASID na AMD)

### 12.5.9 Příklad na čtyřúrovňové tabulce stránek

Chci provést 8B instrukci, která bude načítat 8 B z paměti. **Kolik přístupů do paměti se v nejhorším případě provede?**

- instrukce a data mohou být na jiných stránkách, oboje samostatně mohou být na rozmezí 2 stránek (4 B 1. stránka, 4 B 2.), navíc každou stránku nemusím mít fyzicky v prostoru za sebou (nelze to načíst jedním čtením – čtu že 2 rámců),
- jen pro načtení instrukce a dat budou potřeba 4 přístupy do paměti (pouze jde o samotné čtení instrukce a dat! – nutný ještě překlad),
- překlad se provádí pro každou stránku zvlášť přes 4úrovňovou tabulku stránek,
- provádět se budou $4 \cdot 4$ přístupy (4 stránky, 4 úrovně) pro překlad adres, tzn. dalších 16 přístupů,
- celkem tak $16 + 4 = 20$ přístupů do paměti

### 12.5.10 Hashované tabulky stránek

- logická adresa členěna na číslo stránky $p$, posuv stránky $d$,
- MMU si vede překladovou tabulku, která má charakter *hash tabulky*, ve speciálním registru bude odkaz na začátek hash tabulky,
- vezme se číslo stránky, prožene se hashovací funkci (ta je implementována v HW, MMU) – vypadne odkaz do hashovací tabulky (číslo řádku hash tabulky),
- problém – více různých stránek se může namapovat na stejnou položku v hashovací tabulce,
- není zde tak přímo umístěn překlad, ale odkaz na zřetězený seznam překladových položek (číslo stránky, rámce),
- tento seznam MMU musí projít a dohledat případný překlad,
- až najdeme odpovídající položku, najdeme rámec, ten umístíme do adresy místo čísla stránky, přidá se posuv – dostaneme FA,
- efektivita závisí na délce zřetězených seznamů – pokud bude hash funkce špatná, efektivita bude špatná.

### 12.5.11 Modifikace hashovaných tabulek stránek

- nemusí se používat celé číslo stránky pro hashování,
- je možné použít jen několik bitů stránek pro rozlišení různých hashovacích tabulek

**Fixní počet překladových položek:**

- zřetězený seznam překladových položek bývá nahrazen fixním počtem překladových položek, které se ukládají do hashovací tabulky na daný řádek (namísto aby v tabulce byl odkaz na seznam, tabulka bude „širší“ a na 1 řádku bude 4–8 překladových položek a hledá se v rámci řádku),
- pokud překlad nebude nalezen, neznamená to, že stránka není mapována, pouze se pošle přerušení, jádro zjistí, že se nepodařilo dohledat na daném řádku, proto musí jít do tabulek stránek, které si vede ve vlastní režii (SW tabulka stránek),
- zde si dohledá, jestli překlad existuje (pokud ne, nastane *výpadek stránky*),
- pokud existuje, upraví se tabulka stránek tak, aby tam dané číslo bylo,
- používá se např. u CPU PowerPC nebo na CPU Itanium (položky mohou být zřetězeny, ale HW zřetězený seznam nepoužívá – pokud je první položka překlad, použije se, pokud ne – přerušení, jádro se podívá, jestli má nějaký další seznam překladových položek a čistě v SW se dohledá překlad, pokud ho najde, provede se úprava tabulky stránek a nový překlad)

**Může být sdílena všemi procesy:**

- je nutné do překladových položek umisťovat číslo stránky, ale i odpovídající číslo procesu,
- hashovací funkcí se prožene číslo stránky i číslo procesu,
- dohledává se tak dle čísla stránky i procesu,
- namísto čísla procesu se může ještě pracovat s čísly paměťového regionu, každý proces má svá čísla regionu, lokální čísla regionu se mapují na čísla globální – umožňuje sdílení regionů (lokální regiony procesu se můžou namapovat na 1 fyzický region – sdílený), adresa je dána číslem regionu a číslem stránky,
- kromě čísla stránky mám ještě lokální region v LA, region se převede na globální číslo regionu, číslo stránky se prožene hash funkcí a bude se vyhledávat – používá se u PowerPC a Itanium

## Definice:

***paměťové regiony*** – LAP je dělen na stránky a na vyšší úrovni je dělen na regiony – skupiny stránek, které následují za sebou, jsou proměnné velikosti (něco jako extenty), jsou použity za určitým účelem (datový, kódový, . . . )

### 12.5.12 Příklad na sdílené hashovací tabulce stránek s regiony

LAP je rozdělen na stránky, ty jsou **děleny na regiony** (např. 6 stránek – region 1 má první 3 stránky, region 2 ostatní),

- LA = lok. číslo regionu, číslo stránky, posuv v rámci stránky,
- lokální číslo regionu se přeloží přes tabulku na globální číslo regionů, to se spojí s číslem stránky,
- posuv se nemění,
- glob. číslo regionu a číslo stránky se prožene hashovací funkci,
- dostanu odkaz do hash tabulky, na určitý řádek,
- rozdělena na určitý (pevný) počet záznamů,
- vždy je tam číslo regionu, číslo stránky, odpovídající rámec, potom další číslo regionu, stránky, rámec atd.,
- prohledávám, jestli se někde v příslušné položce nachází dvojice globální číslo regionu a číslo stránky,
- k tomu najdu odpovídající rámec

### 12.5.13 Invertovaná tabulka stránek

- nepřekládá číslo stránek na číslo rámce, ale obráceně – číslo rámce na číslo stránky,

- mapuje rámce na stránky,
- její řádky odpovídají rámcům, je zde tolik položek, kolik mám rámců,
- tabulka je nutné sdílená – pro všechny procesy,
- je to pole,
- hashování je řešené hardwarově

**Překlad:**

- v položkách tabulky se kromě čísla stránky ukládá i PID procesu,
- vezmu PID, číslo stránky,
- prohledávám *od začátku do konce*,
- zjišťuji, jestli některý řádek odpovídá PID a stránce, které mě zajímá,
- nalezl jsem překlad,
- odpovídající rámec má hodnotu $i$, kde $i$ je řádek tabulky, na kterém jsem odpovídající překlad našel.

**Výhody:**

- úspora paměti (1 tabulka pro všechny)

**Nevýhody:**

- prohledávání od začátku do konce je neakceptovatelné – příliš pomalé – řeší se kombinaci s hashováním,
- není moc používaná,
- komplikace s tím, jak implementovat sdílení stránek – pokud více procesů bude sdílet nějaký rámec, stejné položce bude odpovídat více dvojic PID a číslo stránky, jenže může zde být jen jedna dvojice – neustále bude docházet k výpadkům,
- jádro si musí (paralelně) tak vést klasické tabulky stránek, těmi projde, zjistí že překlad je možný, opraví ho, bude chtít první překlad a jedna položka se bude vyhazovat,
- ještě kromě stránek jsou zde regiony, indexovat se pomocí čísla regionu a čísla stránky

**Kombinace s hashováním:**

- místo hledání od začátku do konce,
- se vezme PID a číslo stránky, proženu hashovací funkci,
- dostanu odkaz do tabulky,
- prohledávám od tohoto místa,
- typicky je uvnitř tabulky zřetězení

### 12.5.14 Příklad na invertované tabulce stránek s hashováním

- mám PID procesu, jeho LA = číslo stránky a posuv v rámci stránky,
- vezmu PID a číslo stránky, proženu hashovací funkci,
- dostanu odkaz na nějaký řádek invertované tabulky stránek (v MMU je odkaz na začátek),
- na daném řádku bude uloženo, pro jaký proces je mapování, pro jakou stránku je mapování, a odkaz na další stránku procesu,
- pokud PID a číslo stránky nesouhlasí s daným řádkem, je zde odkaz na jiný řádek, kde má daný proces jinou stránku,
- pokud najdu odpovídající PID a číslo stránky,
- nalezli jsme odpovídající rámec,
- číslo rámce je číslo položky – řádek v tabulce

# 13

**Třináctá přednáška:** Dokončení správy paměti.

## 13.1 Virtualizace paměti

**PAE – page address extension** – Intel, rozšíření na $2^{36}$ bitů fyzické adresy, kde 4 horní bity nastavoval OS. To umožnilo **využít až 64 GiB paměti na 32b systémech** (z hlediska procesu ale bylo možné použít maximálně 4 GiB).

Virtualizace paměti umožňuje procesu a jádru pracovat **s oddělenými lineárními logickými adresovými prostory**. Každý proces vidí svůj prostor, mezi sebou nekolidují. Pro jednotlivé procesy jsou přístupy **transparentní – neví o tom, že v jeden okamžik část používaného FAP využívá např. jiný proces**.

Výhodou je **mensi spotřeba paměti, rychlejší odkládání na disk, zavádění do paměti** (není nutné odložit či zavést celý adresový prostor procesu).

Část paměti **lze odložit na disk a v případě potřeby opět nahrát do PC**. Z disku se části LAP zavádí do FAP pouze tehdy, pokud je to nutné.

Hovoříme pak o:

- stránkování na žádost,
- segmentování na žádost.

## 13.2 Stránkování na žádost

Stránky **jsou zaváděny do paměti, jen pokud k nim přistupujeme**. OS uchovává informace o tom, které stránky jsou využité, a které ne (resp. v tabulce stránek udržuje bit, který určuje, jestli je stránka v paměti, nebo na disku).

Pokud po vyhledání čísla stránky (a příslušného rámce) **je stránka uložena v paměti**, postupuje se normálně (přeloží se na FA). Pokud ne, **pošle se přerušení OS (trap) – jedná se o výpadek stránky** (page fault).

U jiných tabulek stránek (hash, invertovaných) se HW podívá do seznamu stránek, pokud tam stránka nebude, OS se podívá ještě do svých SW tabulek stránek. Pokud však zjistí, že se přistupuje na LA, která není namapována v paměti, dojde k **výpadku stránky**.

Výpadek stránky **je přerušení od MMU, které udává, že nelze převést adresu**, čili že není definováno mapování v tabulce stránek.

## 13.3 Obsluha výpadku stránky

- kontrola, zda proces neodkazuje mimo přidělený adresový prostor (pokud ano, **segfault**, jádro proces ukončí),
- alokace rámce,
  - proces přistupuje do paměti, která není v FA namapovaná,
  - použije se volný rámec, pokud nějaký volný je,
  - pokud není, vybereme si stránku v paměti, která má již přidělený rámec (*victim page* – oběť), odložíme oběť na disk (pokud byla stránka změněna), uvolníme rámec a použijeme uvolněný rámec
- inicializace stránky (po alokaci závislá na předchozím stavu stránky),
  - pokud jde o první odkaz na stránku: pokud je to kód či inicializovaná data, načte se z programu, u všech ostatních se data **vynulují** (kvůli bezpečnosti),

  - pokud to nebyl první přístup (stránka už byla v minulosti uvolněna z FAP): pokud to je kód či konstantní data, načtou se z programu, ostatní: pokud byla modifikovaná, ze swapu se stránka vrátí zpět do FAP, jinak se obsah opět vynuluje
- úprava tabulky stránek – upravit odkaz, kam vede LA (rámec se změní),
- proces je připraven na opakováni instrukce, která výpadek způsobila (je ve stavu připravený)

## 13.4 Výkonnost stránkování na žádost

Efektivní doba přístupu do paměti:

$$(1 - p)T + pD$$

- kde $p$ je *page fault rate* = pravděpodobnost výpadku stránky; $(1 - p)$ je pravděpodobnost, že k výpadku nedojde,
- $T$ doba přístup bez výpadku,
- $D$ doba přístup s výpadkem.

Doba přístupu bez výpadků je mnohem menší než doba přístupu s výpadkem. Závisí na **dostatku paměti a přiměřeném počtu procesů, lokalitě odkazů v procesech, vhodném výběru zaváděných či odkládaných stránek** (algoritmus výběru „oběti“) – snaha mít $p$ co nejmenší.

## 13.5 Počet výpadků stránek

Máme 1 instrukci, kolik výpadků při jejím zpracování může nastat? Může k němu dojít při **čtení instrukce, při práci s každým z jejích operandů**, u obou může dojít k výpadku **vícenásobně**.

Vícenásobné výpadky mohou být způsobeny:

- nezarovnáním instrukce (půlka instrukce v 1. stránce, druhá půlka ve 2. stránce),
- nezarovnáním dat (jako instrukce výše),
- data jsou delší než 1 stránka (instrukce pracující s mnoho daty, např. na Intelu MOVSB),
- výpadky tabulek stránek na různých úrovních – např. hierarchická tabulka stránek, kdy tabulky mohou být velké, proto se odkládají do paměti a může opět docházet k výpadkům – obvykle alespoň část tabulek stránek je chráněna před výpadkem stránek (zejména tabulka stránek nejvyšší úrovně)

**Příklad (podobný bude u zkoušky):**

Jaký je maximální počet výpadků stránek v systému se stránkami o velikosti 4 KiB, čtyřúrovňovou tabulkou stránek, u které pouze dílčí tabulka nejvyšší úrovně je chráněna proti výpadku, při provádění nenačtené instrukce o délce 4 B, která přesouvá 8 KiB z jedné adresy paměti na jinou?

- instrukce bude na rozmezí dvou stránek
  - při přístupu na 1 stránku je nutné projít celou hierarchickou tabulku stránek (máme 4 úrovně, 1 chráněná),
  - v tab 1. úrovně je odkaz na tab. 2. úrovně, zde k výpadku může dojít (+1), tu je odkaz na tab 3. úrovně (+1), stejně u 4. úrovně (+1), to, kam ukazuje tab. 4. úrovně (na FAP) opět nemusí být v paměti (+1),
  - pro zpracování 1. části instrukce může dojít ke 4 výpadkům stránek,
  - zpracování 2. části: 1. stránka bude úplně na konci všech tabulek stránek – při inkrementaci se musí
  - načíst úplně nové 4 úrovně tabulek stránek – opět může dojít k 4 výpadkům jako předtím,
  - celkem 4 + 4 = 8 výpadků po načtení instrukce
- instrukce se musí spustit, přesouvá 8 KiB dat (při 4 KiB stránkách),

  - nejhorší případ: zdrojová data o velikosti 8 KiB jsou rozložena na 3 stránkách (2 KiB první stránka, 4 KiB druhá, 2 KiB třetí),
  - cílová data budou na tom obdobně,
  - abych dostal první stránku zdroje, může opět dojít ke 4 výpadkům,
  - při práci s druhou stránkou – opět první je na konci seznamu, tedy +1 bude úplně v jiných tabulkách – další +4 výpadky,
  - poslední stránka – protože LA následují za sebou, první dvě stránky byly na rozmezí (konec – začátek tabulek), nyní bude ve stejných tabulkách stránek jako předchozí – tedy +1 výpadku stránek (je možný pouze výpadek na úrovni rámce)
  - celkem +9 výpadků
  - u cíle úplně stejný případ jako u zdroje, tedy +9 výpadků,
  - provádění instrukce zabere 9 + 9 = 18 výpadků,
- tedy dohromady při vykonání jedné této instrukce může dojít celkem k 8 + 18 = 26 výpadkům stránek (maximální možný počet, nejhorší případ)

## 13.6 Odkládání stránek

K tomuto může dojít při výpadku stránky. Může být odložení:

- lokální – v rámci procesu (u kterého došlo k výpadku),
- globální – bez ohledu na to, kterému procesu patří která stránka.

Typicky je neustále udržován **určitý počet volných rámců**:

- pokud počet volných rámců klesne pod určitou mez, aktivuje se page daemon (zloděj stránek), který běží tak dlouho, dokud neuvolní dostatečný počet stránek,
- při výpadku stránky se použije rámec z množiny volných rámců,
- lze doplnit heuristikou, která uvolené stránky okamžitě nepřiděluje, ale zjišťuje, jestli nebyla vybraná oběť, která není správná

## 13.7 Algoritmy výběru odkládaných stránek (obětí)

### 13.7.1 FIFO

- *first in, first out*,
- odstraňuje stránku, která byla zavedena do paměti před nejdelší dobou (a nebyla dosud odstraněna),
- např. proces pracuje se 4 stránkami (1, 2, 3, 4), chce pátou, není dost paměti – uvolní se stránka 1, rámec, kde byla 1, se použije pro 5,
- výhody – jednoduchá implementace,
- problémy:
  - může odstranit starou stránku, která se ale často používá,
  - trpí *Beladyho anomálii* – očekáváme, že v systému, ve kterém často dochází k odkládání a zvětší se paměť systému, tak k odkládání bude docházet méně často – to zde pravda být nemusí – počet výpadků vzroste při zvětšení paměti,
- dá se trochu změnit – omezit problém s odstraňováním starých, ale používaných rámců: umístíme takový rámec do množiny volných rámců, přidělíme jiný volný rámec, pokud bychom chtěli rámec použit, ihned se získá tento uvolněný rámec (signál – špatná oběť, vybere se jiná), pokud ne, odloží se

### 13.7.2 LRU

- *least recently used*,
- snaží se odkládat nejdéle nepoužitou stránku,
- dobrá aproximace hypotetického ideálního algoritmu (znal by budoucnost a podle požadavků z budoucnosti by rozhodoval, co aktuálně odložit tak, aby byl počet výpadků minimální – zatím to nejde)
- problémy:
  - při cyklických průchodech polí se algoritmus může chovat velmi neefektivně – např. systém se 4 rámci, dělám bubble sort na 5 paměťových blocích (první 4 bloky budou na všech rámcích, zpracují poslední, první vyhodím a dám na první rámec, poté potřebují něco pro první blok, vyhodím druhý rámec, ... pořád dokola)
  - řeší se to například strategii odstranění naposledy použité stránky (MRU – *most recently used*)
  - další problém je problematická implementace – je nutné umět detekovat použitou stránku (nutná HW podpora, např. časové razítko)
- používají se aproximace LRU

Aproximace LRU **pomoci omezené historie referenčního bitu stránek** (page aging):

- použije se jeden bit – referenční bít,
- tento bit HW nastaví na 1 při každém přístupu,
- jádro si vede omezenou historii tohoto bitu (pro jednotlivé stránky),
- periodicky se posouvá obsah historie doprava (shiftuje se referenční bit doprava),
- poté se referenční bit v tabulce stránek vynuluje,
- poté při výběru obětí jádro projde všechny historie a vybere z nich nejmenší hodnotu (zn. používala se stránka před nejdelší dobou)

Aproximace LRU **algoritmem druhé šance**:

- stránky jsou uloženy v kruhovém seznamu,
- máme jeden ukazatel, tím seznam procházíme,
- vynulujeme referenční bit dané stránky,
- pokud už bude 0, použijeme danou stránku jako oběť,
- také se tomu říká clock algorithm

**Modifikace algoritmu druhé šance:**

- upřednostňují se jako oběti nemodifikované stránky (ušetří se 1 zápis na disk) – modifikované se zapíší na disk a dostanou další šanci,
- dva ukazatele, které prostupují frontou – jeden nuluje referenční bit, druhý odstraňuje oběti (double-handed clock algorithm),
- Linux:
  - fronty aktivních a neaktivních stránek – pokud nějaká stránka je během 1 periody zpřístupněna $2\times$, tak se přesune do fronty aktivních stránek, pokud ne, přehodí se do fronty neaktivních stránek (z té se vybírají oběti),
  - systém se snaží odkládat stránky, které jsou použité jako různé vyrovnávací paměti
  - pokud má proces namapován určitý počet stránek, OS se snaží odstraňovat jeho neaktivní stránky,
  - odkládání se provádí po určitých počtech (najednou), čím více je paměť zaplněna, tím více se rámců uvolní,
  - je možné nastavit procesu příznak, aby se jeho stránky neodkládaly (nebyly oběti),

  - při kritickém nedostatku paměti jádro ukončuje některé procesy (většinou: pokud dojde RAM i swap zároveň) – obvykle to ale bývá příliš pozdě,
  - v Linuxu existuje služba EarlyOOM (není služba jádra!) – začne tyto procesy ukončovat dřív při nedostatku paměti

## 13.8 Alokace rámců procesům (resp. jádru)

- důležité hlavně u lokálního výběru,
- u globálního výběru lze použít pro řízení výběru obětí,
- je třeba mít vždy přidělen minimální počet rámců pro provedení jedné instrukce – jinak dojde k nekonečnému vyměňování stránek potřebných k provedení instrukce,
- dále se používají různé heuristiky pro určení počtu rámců pro procesy:
  - podle velikost programu, priority, objemu fyzické paměti, ...,
  - na základě pracovní množiny stránek – množina stránek, které používá proces za nějakou určitou dobu (aproximace pomocí referenčního bitu),
  - sledování frekvence výpadku procesu (více výpadků, častěji ⇒ proces dostane více fyzické paměti)

Přidělování rámců **s využitím pracovní množiny, kombinace lokální a globální výměny, využívá např. Windows**:

- procesy a jádro mají jistý minimální a maximální počet rámců,
- pokud je dost paměti, rámce se jim dodávají až do dosažení maxima (může se i zvětšit),
- pokud se dosáhne maxima a není dost paměti, provede se lokální výměna,
- pokud je volných rámců výrazný nedostatek, OS začne procházet běžící procesy a začne jim odebírat určitý počet stránek na základě omezené historie přístupů,
- při výběru obětí se dává přednost procesům běžícím méně často,
- snaží se vyhýbat těm procesům, které běží na popředí, a těm procesům, které měly v poslední době mnoho výpadků,
- počáteční pracovní meze se zjistí při startu systému dle velikosti fyzické paměti,
- při odkládání na disk vybrané oběti jí dá ještě 2. šanci – nechá si ji ještě chvíli v paměti, přidělí jí jiný rámec, aby bylo možné korigovat chyby při volbách oběti,
- umí swapovat všechna data procesu na disk (typicky dlouho neaktivní proces)

## 13.9 Trashing

Trashing je problém, který může nastat při odkládání stránek na disk. Vznikne, pokud máme **systém s vysokou mírou paralelismu** (resp. mnoho spuštěných procesů) – procesy potřebují nějaký **minimální počet rámců, se kterými musí pracovat**. Pokud je počet procesů velký, paměti **je málo a často může docházet k odkládání na disk**. Potom může dojít k tomu, že **proces stráví více času náhradou stránek než užitečným výpočtem**.

Občas v nějakých OS existuje **swapper** (služba OS), ten pozastaví některé procesy a odloží veškerou paměť na disk. Možností je také ukončit některé procesy.

## 13.10 Poznámky

**Prepaging** – do fyzické paměti zavádí více stránek zároveň (start procesu, po odswapováni, ... – zrychlení).

**Zamykání stránek:**

- zabraňuje se odložení,

- užívá se např. u stránek, do nichž probíhá I/O (stránky se mohou mapovat do zařízení – nechceme tyto stránky ukládat na disk), u (části) tabulek stránek (např. nejvyšší úroveň), u (některých) stránek jádra, na přání uživatele (např. v paměti pracujeme s citlivými daty)

**Sdílení stránek** Dá se sdílet:

- kód programu (proces vícekrát spuštěný, není třeba mít stejný kód v paměti několikrát; nebo sdílené knihovny),
- konstantní data či doposud nemodifikovaná data u kopií procesů (copy-on-write technika),
- mechanismus IPC (2 procesy mohou sdílet stejnou paměť, používají ji třeba pro synchronizaci),
- sdílení paměťově mapovaných souborů

**Sdílené (dynamicky linkované) knihovny**

- uložené ve FAP pouze jednou,
- .dll či .so,
- výhody: menší programy (není nutné do programu dát všechny funkce, část může být v knihovně), lepší využití prostoru na disku, možnost aktualizovat knihovny,
- nevýhody: závislost programů na dalších souborech a verzích knihoven, možný pomalejší start programu (ale knihovna může být již v paměti), možné pomalejší volání funkcí (kompilátor zde nemůže provádět optimalizace, navíc volání musí jít přes sestavovací tabulky)

**Copy-on-write**

- při spuštění `fork` se nevytvoří kopie veškeré paměti procesu,
- tedy mám dva LAP, ale oba procesy sdílejí stejné rámce v FAP, ty jsou označeny jako copy-on-write,
- znamená, že stránky se sdílí mezi procesy, dokud do nich jeden z procesů nezapíše

**Sdílená paměť**

- shared memory,
- forma IPC, více procesů má mapovány stejné fyzické stránky do LAP

**Paměťově mapované soubory**

- celý soubor se namapuje do LAP,
- procesy si soubor namapují do svého LAP, pracují s ním, jako by přistupovaly do paměti,
- využití stránkování na žádost pro práci se soubory,
- použití např. při práci s databázemi,
- dostupné např. prostřednictvím `mmap()`

**Paměťové regiony**

- jednotka vyššího strukturování paměti v UNIXu,
- v rámci paměťového regionu je každá adresa (region pro kód, haldu, zásobník, data, ...),
- používají segmentové registry, neukazují na počátky segmentu, ale o jaký region jde (pro zásobník, kód, ...),
- umožňuje jistý druh práce s danou pamětí,
- např. region s kódem – je možné zakázat zápis

**Standardní rozložení adresového prostoru procesu v Linuxu (i386):**

- region na adrese 0 – nepoužitý (`NULL`),
- nad ním je text segment – kód programu (provádění kódu, žádný zápis),

- segment se statickými inicializovanými daty (modifikace, ale nespouštět),
- prostor pro neinicializované statické proměnné (zaplněno nulami),
- halda (heap) – práce např. přes volání `malloc`, roste od nějaké adresy nahoru,
- paměť použitá na mapování LAP na sdílené rámce paměti či sdílená paměťová místa,
- zásobník (stack; omezená velikost, „roste dolů“),
- nejvýše je prostor namapovaný pro funkce kernelu – kernel space

THE END