

VYSOKÉ UCENÍ TECHNICKÉ V BRNĚ  
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

IOS - Operační systémy  
poznámky z přednášek

# Obsah

<b>1</b>	<b>5</b>
1.1	Uvod, prehľad operacných systému . . . . . 5
1.2	Základní pojmy . . . . . 6
1.3	Jadro operacního systému . . . . . 7
1.4	Typy jader OS . . . . . 8
1.5	Historie vyvoje OS . . . . . 10
1.6	Prehľad technického vybavení . . . . . 10
1.7	Klasifikace počítačů . . . . . 11
1.8	Klasifikace OS . . . . . 12
1.9	Implementace OS . . . . . 12
1.10	Hlavní směry ve vývoji OS . . . . . 13
<b>2</b>	<b>14</b>
2.1	Příčiny úspěchu UNIXu . . . . . 14
2.2	Variety UNIXu . . . . . 15
2.3	Základní koncepty . . . . . 15
2.4	Struktura jádra UNIXu . . . . . 16
2.5	Komunikace s jádrem a hardwarová přerušeni . . . . . 17
2.5.1	Hardwarové přerušeni . . . . . 17
2.5.2	Zakazování přerušeni . . . . . 18
2.5.3	Přístupy k zakazování přerušeni . . . . . 19
2.5.4	Ovládací zařízení a přerušeni . . . . . 19
2.5.5	Příklad komunikace s jádrem . . . . . 20
2.6	Nástroje programátora UNIXu . . . . . 21
<b>3</b>	<b>21</b>
3.1	Bash, shell, experimenty . . . . . 21
<b>4</b>	<b>21</b>
4.1	Bash, shell, experimenty . . . . . 21
<b>5</b>	<b>22</b>
5.1	Pevný disk . . . . . 22
5.2	Parametry pevných disků . . . . . 23
5.3	Solid State Drive - SSD . . . . . 24
5.3.1	Klady a zápory SSD . . . . . 24
5.3.2	Problematika zápisu u SSD . . . . . 24
5.4	Zabezpečení disku . . . . . 25
5.5	Disková pole (RAID) . . . . . 26
5.5.1	RAID 0 . . . . . 26
5.5.2	RAID 1 . . . . . 26
5.5.3	RAID 2 . . . . . 26
5.5.4	RAID 3 . . . . . 27
5.5.5	RAID 4 . . . . . 27
5.5.6	RAID 5 . . . . . 27
5.5.7	RAID 6 . . . . . 27
5.6	Opravy chyb u paritních disků . . . . . 28
5.7	Uložení dat na disk . . . . . 28
5.8	Fragmentace . . . . . 29

5.8.1	Externi fragmentace . . . . .	29
5.8.2	Interni fragmentace . . . . .	30
5.9	Přístup na disk . . . . .	31
5.10	Planování přístupu na disk . . . . .	31
5.11	Logický disk . . . . .	32
5.11.1	Způsob uložení informací o diskových oblastech na disku . . . . .	32
5.11.2	LVM . . . . .	32
5.11.3	Různé typy souborových systémů . . . . .	32
5.11.4	Chyby disku (souvislost s FS) . . . . .	33
5.11.5	Další typy souborových systémů . . . . .	33
<b>6</b>		<b>34</b>
6.1	Zápisování . . . . .	34
6.1.1	Implementace zápisování . . . . .	35
6.1.2	Copy-on-write . . . . .	36
6.1.3	Další alternativy zápisování . . . . .	37
6.2	Klasický UNIXový systém souborů (FS) . . . . .	38
6.2.1	i-uzel . . . . .	38
6.2.2	Kde a jak jsou uložena data . . . . .	39
6.2.3	Typy odkazů . . . . .	40
6.2.4	Limit maximální velikosti souboru . . . . .	40
6.2.5	Výhody a nevýhody architektury FS . . . . .	41
6.3	Jiné způsoby organizace souborů . . . . .	42
6.3.1	Kontinuální uložení . . . . .	42
6.3.2	Zrežované seznamy alokací bloků . . . . .	42
6.3.3	FAT . . . . .	42
6.3.4	B+ stromy . . . . .	43
6.3.5	Extent . . . . .	45
6.4	EXT4 . . . . .	46
6.5	NTFS . . . . .	46
6.6	Organizace volného prostoru na disku . . . . .	46
6.7	Deduplikace . . . . .	47
6.8	Typy souborů v UNIXu . . . . .	47
6.9	Adresář . . . . .	48
6.10	Montování disku . . . . .	49
<b>7</b>		<b>51</b>
7.1	Symbolické odkazy . . . . .	51
7.2	Blokové a znakové speciální soubory . . . . .	52
7.3	Přístupová práva . . . . .	53
7.4	Typy přístupových práv . . . . .	54
7.5	Sticky bit . . . . .	54
7.6	SUID, SGID . . . . .	54
7.7	Typická struktura adresáře v UNIXu . . . . .	55
7.8	Použití vyrovnávacích pamětí . . . . .	56
7.9	Operace se soubory . . . . .	57
7.9.1	Čtení . . . . .	57
7.9.2	Zápis . . . . .	57
7.9.3	Otevření souboru pro čtení . . . . .	58
7.9.4	Čtení a zápis z/do souboru . . . . .	59

7.9.5	Prímy prístup k súboru . . . . .	60
7.9.6	Zavrení súboru . . . . .	61
7.9.7	Duplikácia deskriptoru súboru . . . . .	61
7.9.8	Rúsení súboru . . . . .	62
7.9.9	Dalsí operácie se súbory . . . . .	62
7.9.10	Adresárové súbory . . . . .	62
7.9.11	Blokové a znakové špeciálne súbory . . . . .	63
7.10	Terminály . . . . .	64
7.11	Roury . . . . .	65
<b>8</b>		<b>66</b>
8.1	Sockety . . . . .	66
8.2	VFS . . . . .	66
8.3	NFS . . . . .	67
8.4	Spooling . . . . .	67
8.5	Proces . . . . .	68
8.6	Stavy plánovania a jejích zmeny . . . . .	69
8.7	Časti procesu v pameti v UNIXu . . . . .	71
8.8	Kontext procesu . . . . .	72
8.9	Systemová volání nad procesy UNIXu . . . . .	73
8.10	Vytvárení procesu . . . . .	73
8.11	Hierarchie procesu v UNIXu . . . . .	74
8.12	Zmena programu – exec . . . . .	74
8.13	Čekání na potomka - wait, waitpid . . . . .	75
8.14	Start systému . . . . .	75
8.15	Úrovně behu . . . . .	76
8.16	Plánování procesu . . . . .	77
8.17	Prepnutí kontextu (procesu) . . . . .	77
<b>9</b>		<b>78</b>
9.1	Kratké plánovací algoritmy . . . . .	78
9.1.1	FCFS . . . . .	78
9.1.2	Round-robin . . . . .	78
9.1.3	SJF . . . . .	78
9.1.4	SRT . . . . .	79
9.1.5	Viceúrovňové plánování . . . . .	79
9.1.6	Viacerúrovňové plánování se zpětnou vazbou . . . . .	79
9.2	Planovac v Linuxu (od verze 2.6.23) . . . . .	80
9.3	Completely Fair Scheduler . . . . .	80
9.4	Planování ve Windows NT a novejších . . . . .	81
9.5	Inverze priorit . . . . .	81
9.6	Vlákna, úlohy, skupiny procesu . . . . .	82
9.7	Úlohy, skupiny procesu, sezení . . . . .	83
9.8	Komunikace procesu . . . . .	84
9.9	Signály . . . . .	84
9.9.1	Predefinování obsluhy signálu . . . . .	85
9.9.2	Blokování signálu . . . . .	85
9.9.3	Zasílání signálu . . . . .	86
9.9.4	Čekání na signál . . . . .	86

<b>10</b>		<b>87</b>
10.1	Synchronizace procesu . . . . .	87
10.2	Kritické sekce . . . . .	87
10.3	Problémy vznikající na kritické sekci . . . . .	88
10.4	Způsoby řešení problému kritické sekce . . . . .	88
10.5	Využití atomických instrukcí pro synchronizaci . . . . .	90
10.6	Semaforey . . . . .	91
<b>11</b>		<b>93</b>
11.1	Monitory . . . . .	93
11.2	(Některé) Klasické synchronizační problémy . . . . .	96
11.2.1	Problém producenta a konzumenta . . . . .	96
11.2.2	Problém čtenáře a pisáře . . . . .	98
11.2.3	Problém večerících filozofů . . . . .	100
11.3	Deadlock (uvaznutí) . . . . .	102
11.3.1	Definice . . . . .	102
11.3.2	Typický příklad deadlocku . . . . .	102
11.3.3	Coffmanovy podmínky . . . . .	103
11.3.4	Řešení uvaznutí . . . . .	103
11.3.5	Prevence uvaznutí . . . . .	104
11.3.6	Vyhýbání se uvaznutí . . . . .	105
11.3.7	Detekce uvaznutí a zotavení . . . . .	106
11.4	Formální verifikace, verifikace s formálními kořeny . . . . .	107
11.4.1	Theorem proving . . . . .	108
11.4.2	Model checking . . . . .	108
11.4.3	Static analysis . . . . .	108
<b>12</b>		<b>109</b>
12.1	. . . . .	109

# 1

**První přednáška:** Úvod do predmetu, prehľad operacnich systemu, zakladni pojmy, jadro operacniho systemu a jejich typy, historie vyvoje operacnich systemu, prehľad technickeho vybaveni, klasifikace pocitacu, operacnich systemu, hlavni smery ve vyvoji operacniho systemu.

## 1.1 Úvod, prehľad operacnich systemu

Operacni system je vyznamnou casti vypocetnich systemu, ty zahrnuji:

- hardware,
- operacni system,
- uzivatelske aplikacni programy,
- uzivatele.

Prehled nekterych OS:

- GNU/Linux
  - GNU/Debian - Ubuntu
  - Red Hat - RHEL, Fedora, Cent OS
  - SuSE
  - Gentoo, Arch Linux, Slackware (= nejstarsi live distribuce linuxu)
- BSD
  - FreeBSD, OpenBSD
- GNU
  - zn. GNU Is Not Unix
- MS Windows
- Mac OS X
  - jadro XNU = X is Not Unix
- Android, iOS
- Minix
  - pouziva intel ve svych cipech

## 1.2 Zakladni pojmy

Operacni system je program (resp. kolekce programu), která vytváří spojující mezivrstvu mezi hardware operacního systému a uživateli a jejich uživ. aplik. programy. OS dále spotřebovává zdroje, jako jsou paměť nebo čas CPU. (tldr: sw, spojující hardware, uživatele a programy)

### Cile OS:

- maximalní využití zdrojů počítače - drahé počítače, levnější pracovní síla (drive)
- jednoduchost použití počítačů - levné pc, drahá pracovní síla (dnes převládá)

### Zakladni role OS:

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

### OS zahrnuje:

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

Presná 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 jím venujeme od 8.4)

*program* je předpis, návod na nějakou činnost zakódovaný vhodným způsobem

*soubor* je kolekce zaznamů (obvykle Byte) sloužící primárně jako základní jednotka pro ukládání dat na vnějších paměťových médiích

*adresář* je kolekce souborů

### 1.3 Jadro operacniho systemu

Jedna se o nejnizsi a nejzakladnejsi cast OS. Zavadi se jako prvni a bezi po celou dobu behu pocitacoveho systemu (tzv. reaktivni system, spis nez transformacni). Navazuje primo na hardware (pripadne virtualizovany HW) a pro uzivatele a uziv. aplik. zcela zapouzduje.

#### Bezi v privilegovanem rezimu:

- je mozne menit obsah registru hw, je mozne zadavat prikazy hw (neni mozne v uzivatelskem rezimu)
- musi byt podporovano v hardware

#### Jadro (obecne) zajistuje:

- zakladni spravu prostredku a tvorbu zakladniho prostredi jak pro uzivatele tak pro zbytek OS
- zahrnuje vsechny operace, kdy je potreba primo komunikovat s hardware (prepinani kontextu - jadro, plaovani procesu - nekdy v jadre, nekdy mimo, zavedeni stranky z disku, ..)
- sluzby pro zbytek OS a uzivatele, nektere zajistuje automaticky
- nektere sluby nejsou poskytovany automaticky, musi si o ne zadat, nazyvame to volani sluzeb, tzv. *system-call* (= systemova volani), ktere musi byt implemenovana uzitim specializovanych instrukci (intel: *sw preruseni*, *syscall*, *sysenter*)

#### Rozlisujeme dva typy rozhrani OS:

- *kernel interface* (nebo taky: ABI, Kernel ABI) - prime volani jadra pomoci specializovanych instrukci
- *library interface* - rozhrani vyssi urovne (napr. C knihovny), typicke sluzby jsou napr. *printf* z C - volaji se funkce ze systemovych knihoven, mohou ale nemusi vest na volani sluzeb jadra (bezne aplikace pracuji s timto rozhranim)

#### definice:

*transformacni system* je system, ktery dostane nejaky vstup, zpracuje ho a udela nejaky vystup (prekladac) - pokud se zacykli = chyba

*reaktivni system* se spusti a do (teoreticky) nekonecna reaguje na podnety uzivatele (spust procesu - spusti procesu) - pokud prestane pracovat = chyba

*prepinani kontextu* je situace, kdy na CPU bezi proces, ten chci pozastavit a nechat bezet jiny proces

*instrukce syscall a sysenter* - jakmile aplikace (bezi v uziv. rezimu) zavola takovou instrukci, dojde ke kontrolovani prepnuti do rezimu jadra, provede se sluzba, a pote se prepne zpet

ABI = Application Binary Interface



## 1.4 Typy jader OS

### Monolitická jadra

- vysokourovnove komplexni rozhrani s radou sluzeb, abstrakci, které mohou pouzivat vyssi vrstvy OS
- vsechny subsystemy jsou implementovany v privilegovanem rezimu, rezimu jadra, a zahrnuji napr. spravu pameti, planovani, meziprocessovou komunikaci, souborove systemy, ..
- vyhody: vysoka efektivita díky provazanosti
- nevychody: mala flexibilita pri praci s jadrem (ve filesystemu je chyba, chci zmenit jen implementaci filesystemu za novou verzi a vse ostatni nechat - nelze, je nutne cely system zastavit a znovu nastarovat, nelze menit nic za behu)

### Monolitická jadra s modulární strukturou

- vylepseni koncepce monolitických jader
- umoznuje zavadet/odstranovat subsystemu jadra v podobe tzv. modulu za behu
- vyhody: neni nutne cely system zastavovat a znovu bootovat pro vymenu jednoho modulu, vyssi bezpecnost - zavedou se jen moduly, které se budou pouzivat
- pouzivane v napr. FreeBSD, Linux

### Mikrojadra

- snaha minimalizovat rozsah jadra a rozsah jeho sluzeb
- nabizi jednoduche rozhrani, maly pocet abstrakci, sluzeb, typicky nabizi nejzakladnejsi spravu CPU, I/O zarizeni, pameti, ..
- vetsina sluzeb nabizenych monolitickymi jadry (ovladace, vyznamne casti spravy pameti, planovani) je implemenovana mimo jadro v tzv. serverech (nebezi v privilegovanem rezimu).
- vyhody: flexibilita (vice soucasne bezicich implementaci ruznych sluzeb, dynamicke spousteni, zastavovani..), zabezpeceni (chyba v serveru / utok na ne neznamená ovladnuti celeho OS, ale jen daneho serveru)
- nevychody: vyrazne vyssi rezie

### Generace mikrojader

- 1. generace - napr. Mach
- 2. generace - napr. L4, mensi rezie nez 1. gen
- 3. generace - napr. seL4 nebo ProvenCore, duraz na zabezpeceni, navrh s ohledem na moznost formalni verifikace

## Hybridni jadra

- "neco mezi mikrojadry a monolitickymi jadry"
- jadra zalozena na mikrojadrech, rozsirena o kod, který by mohl byt implementovan ve forme serveru, je ale za ucelem mensi rezie tesneji provazan s mikrojadrem a bezi v jeho rezimu
- pouzivane v napr. Mac OS X (Mach + BSD), Windows NT (a vyssi), ...

## definice:

*servery (v oblasti mikrojader) jsou procesy*

*formalni verifikaci rozumime overeni urcitych vlastnosti systemu s platnosti matematickeho dukazu*

## linux prikazy:

*lsmod* - vypise aktualne zavedene moduly jadra

*rmmod* - maze moduly jadra

*modprobe* - zavadeni modulu do jadra

## 1.5 Historie vyvoje OS

### definice:

*preruseni* je elektricky signal, který jde od periferie po sbernici k procesoru, na CPU vyvola obsluhu preruseni - mechanismus umoznujici rozbehnout operaci na periferii a o tu periferii se nestarat (periferie pote oznami konec operace) (podrobne se tomu venuje oddil 2.5)

*multitasking* je soucasny beh vice aplikaci na jednom procesoru (muze byt s preemtivnim nebo nepreemtivnim planovanim)

*nepreemtivni planovani* zn. ze uloha, kt. aktualne bezi na CPU muze byt od CPU "odstavena" pouze tehdy, kdyz nejak zakomunikuje s jadem (= pozada o sluzbu jadra, napr. periferni operace), dokonce lze pouzit specializovane sluzby pro prepnuti kontextu (proces se dobrovolne vzda CPU, tzv. yield sluzby) - vyhoda: snadna implementace, nevychoda: pokud se proces zacykli (chyba), cely system se zablokuje (porad bezi 1 uloha - vice viz 8.16)

*preemtivni planovani* - proces muze byt odstaven od CPU bez nutnosti komunikace s jadem, napr. pomoci preruseni (jakehokoli typu - vice viz 8.16)

## 1.6 Prehled technickeho vybaveni

### Procesor (CPU):

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

### Pamet:

- adresa
- hierarchie pameti (cache, RAM, disk, ... - bank pameti muze byt vice)
  - pameti se lisi spotrebou, kapacitou, rychlosti, cenou za jednotku
  - na vrcholu hierarchie jsou registry (nejrychlejsi, nejvyssi cena za jednotku, mala kapacita)
  - cache (vyrovnavaci pameti, ruznych urovni, L1 = level 1, L2, L3, ..)
  - primarni pamet RAM
  - sekundarni pameti - disk (SSD, HDD)
  - vyrovnavaci pameti disku
  - tercialni pameti (zalozy - nejnizsi cena za jednotku, nejpomalejsi, nejvetsi kapacita - pasky, CD/DVD, externi disky, cloudy, sitove disky, ..)

### Periferie:

- disk (HDD, SSD,...), klavesnice, monitor (I/O porty, preruseni, DMA)

### Sbernice:

- propojuji jednotlivé komponenty
- na vrcholu hierarchie jsou sbernice propojujici CPU a pamet (FSB - Front Side Bus, HyperTransport QPI - Quick Path Interconnect)

- diskove sbernice (SATA/ATA, SCSI/SAS, USB)
- dalsi sbernice (NVLink - pripojovani nVidia GPU, PCI - rozsirujici karty ci disky, CAPI - IBM Tauer CPU, propojovani CPU a akceleratoru)

#### **definice:**

*I/O porty* = vstup-vystupni porty, predstavuji pametove oddeleny prostor od adresoveho prostoru bezne pameti, s temito adresami se komunikuje specialnimi instrukcemi (intel: inout)

*pametove mapovane I/O* je cast adresoveho prostoru bezne pameti neni pouzita pro praci s pameti, ale adresy jsou presmerovane do HW (neco co zapisu na danou adresu nebude v pameti ale v nejakem registru HW)

*DMA* zn. Direct Memory Access, souvisi s nezavislou cinnosti periferii - periferie mohou primo komunikovat s hardware (radic disku si sam z adresy pameti nacte data a pres sbernice je prenasi na disk, nebo naopak)

## **1.7 Klasifikace pocitacu**

#### **Dle ucelu:**

- univerzalni,
- specializovane
  - vestavene (palubni pc, spotrebni elektronika, ..)
  - aplikacne orientovane (rizeni db, sitove servery, ..)
  - vyvojove (zkouseni novych technologii)

#### **Podle vykonnosti:**

- vestavene pc, tablety, mobily, ..
- osobni pocitace (PC) a pracovni stanice (workstation) - dnes se nerozlisuje
- servery
- strediskove pocitace (mainframe) - vyrabi IBM, ladene na obrovsky I/O vykon a vysokou spolehlivost
- superpocitace - ladene na surový vypocetni vykon (vedecke vypocty, simulace)

## 1.8 Klasifikace OS

### Podle ucelu:

- univerzalni (UNIX, Linux, Windows, ..)
- specializovane (real-time - RT-Linux, databaze, web - z/VSE, mobilni - iOS, Android)

### Podle poctu uzivatelu:

- jednouzivatelske (CP/M, MS-DOS,..)
- viceuzivatelske (UNIX, Windows, ..)

### Podle poctu soucasne bezicich uloh:

- jednoulohove
- viceulohove (multitasking, ne/preemptivni)

### definice:

*soft real-time* - doporučení aby se akce vykonávaly v reálném case

*hard real-time* - akce se musí vykonávat v určitém case

## 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* zn. že zde běží více aktivit současně

*paralelní asynchronní systémy* - procesy se prepínají v okamžicích, které nelze dopředu přesně předpovědět

### 1.10 Hlavní směry ve vývoji OS

- neustálé vylepšování architektur (snížení rezí 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* zn., že systém je odolný vůči vnějším útokům

*spolehlivost* zn., že systém "nespadne sám od sebe"

## 2

**Druha prednaska:** Unix - uvod: historie UNIXu (nezkousi se), priciny uspechu UNIXu, varianty UNIXu, zakladni koncepty, struktura jadra, komunikace s jadrem - hardwarova preruseni. Prehled programovani v UNIXu: nastroje programatora, ..

### 2.1 Priciny uspechu UNIXu

- viceprocesovy, viceuzivatelsky,
- napsan v C - prenositelny,
- zpocatku (a pozdeji) siren ve zdrojovem tvaru,
- "mechanism, not policy",
- "fun to hack",
- jednoduche uzivatelske rozhrani (terminal),
- skladani slozitejsich programu z jednodussich (tvoreni aplikaci typu filtr),
- hierarchicky system souboru,
- konzistentni rozhrani perifernich zarizeni

#### definice:

*"mechanism, not policy"* zn. snaha oddelit casti aplikaci (napr. GUI - oddelit zakladni rutiny pro vykreslovani grafiky od politik, tzn. koncove nastavby - barvy oken, umisteni tlacitek, .. - systematicke rozdeleni vede k lepsim optimalizacim a ladenim algoritmu a zaroven rychlym zmenam politik)

*"fun to hack"* zn., lide se na vyvoji podili, protoze je to bavi (nejen protoze jsou za to placeni)

*aplikace typu filtr* - jednoduche otevrene aplikace, na vstupu maji textovy dokument v otevrene podobě, vstup zpracuji a na vystupu opet otevreny dokument (zadne binarni, zakodovane)

## 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 příkazy rozhraní OS + API

## 2.3 Základní koncepty

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

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

### definice:

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

*soubory* jsou abstrakcí dat (viz 1.2)



## 2.4 Struktura jadra UNIXu

Zakladni podsystemy jsou sprava souboru a sprava procesu.

### Popis:

- Na hornim okraji jadra (smerem k uzivatelum, aplikacim) je vrstva implementujici rozhrani volani sluzeb, prostrednictvim ktere jadro prebira zadosti o sluzby od aplikaci. Rozhrani kontroluje zda ten, kdo o sluzbu zada ji muze volat, zda jsou parametry validni a rozhrani predava pozadavek dal do jadra.
- Aplikace mohou s jadem komunikovat primo, nicmene nejcasteji komunikuji s jadem pres knihovny. (viz. 1.3)
- Na druhem okraji (tesne nad HW) je vrstva abstrakce hardware.
- Mezi spravou souboru a hardware se nachazi ovladace, pote vrstva vyrovnavacich pameti, ktere souborove systemy pouzivaji ke zrychleni prace s relativne pomalymi disky (HDD, SSD - oproti RAM pomale)
  - OS se snazi vyhnout opakovanemu cteni stejných dat, proto si v jednom okamziku nacte vic dat nez uzivatel zada, ulozi si data do vyrovnavaci pameti (pri dostatku pameti) a data nacita odtud. (napr. C knihovny jsou pouzivane kazdym druhym programem - jsou v pameti temer porad).

### definice:

*ovladace* jsou programy slouzici k rizeni (zadavani prikazu, prebirani stavovych informaci, reseni mimoradnych stavu konkretnich periferii) - lze je (jako i prislusna zarizeni) rozdelit na znaková a bloková (kratsi definice viz 5.9)

*znakova zarizeni* jsou zarizeni komunikujici po jednotlivych znacich (klavesnice)

*bloková zarizeni* komunikuji po blocich (disk - sektory, resp. bloky)

*komunikaci s jadem* rozumime nastavovani parametru hardware, vydavani prikazu hw, obsluhu ruznych stavu do kterych se hw dostava (a o kterych je CPU a jadro informovano prostrednictvim preruseni)

*nastavovani parametru hw* se deje pomoci I/O portu nebo pametove mapovanych operaci (viz 1.6)

## 2.5 Komunikace s jadem a hardwarova preruseni

Sluzby jadra jsou operace, jejich realizace je pro procesy zajistovana jadem. Explicitne je mozne o provedeni urcite sluzby zadat prostrednictvim system call (viz 1.3).

### Priklady nekterych sluzeb jadra (systemova volani v UNIXu):

- open, close, read - otevře/zavře/čte soubor,
- write - zapisuje,
- kill - posle signal,
- fork - duplikuje proces,
- exec - prepíše kód,
- exit - ukončí proces.

### 2.5.1 Hardwarove preruseni

- hardware interrupt je mechanismus, kterým HW zařízení oznamují jádru asynchronne vznik udalosti, které je zapotřebí obsloužit (další možná definice viz 1.5),
- žádosti o HW preruseni přichází jako elektrické signály (IRQ) do radice preruseni (APIC),
- procesor s radicí preruseni komunikuje pomocí I/O portu.

### Přijem nebo obsluhu HW preruseni lze zakázat:

- maskováním preruseni,
- na CPU (instrukce CLI/STI na Intel/AMD - zakazou se všechna kromě NMI),
- čisté programy v jádře (preruseni se přijme, ale jádro si jen poznamená jeho příchod a neobsluhuje se)

### NMI:

- non-maskable interrupt je HW preruseni, které nelze zamaskovat na radici ani zakázat na CPU,
- používá se při kritických chybách paměti, sbernice, .. (alternativně se používá pro ladění / řešení uvážnutí v jádře "NMI watchdog")

### Preruseni mohou vznikat i v CPU - jsou to synchronní preruseni, tzv. vyjimky (= exceptions):

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

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

- Interprocessor interrupt (IPI)

- meziprocesorove preruseni
- pouziva se pro preposilani preruseni z jednoho CPU na druhy nebo pro spravu cache (kazdy CPU ma svoji cache, do nich mohou mit CPU nacteny stejne adresy z pameti - pokud dojde ke zmenam v pameti, musi CPU informovat ostatni CPU o zmene)
- System management Interrupt (SMI)
  - preruseni typu sprava systemu
  - muze byt vyvolano HW i SW ve zvlastnich situacich
  - pokud se takove preruseni vyvola, tak se dostane ke slovu firmware, který provadi obsluhu ruznych chybovych stavu (prehrati, vybita baterie, ..)
  - v ramci SMI nebezi bezne aplikace ani jadro, nesmi obsluha SMI bezet prilis dlouho (system se muze dostat do nekonzistentniho stavu)

### **2.5.2 Zakazovani preruseni**

#### **Proc preruseni zakazovat?**

- v ramci obsluhy jednoho preruseni muze nastat dalsi preruseni,
- napr. na CPU bezi vypocet, neco nastane na disku, disk posle preruseni, to dojde k CPU a jadro zacne preruseni obsluhovat, v ten moment se neco stane na klavesnici a prijde dalsi preruseni,
- pote dale v ramci obsluhy muze jadro upravovat ruzne sve interni struktury, které mohou byt v nekonzistentnim stavu (napr. zretezene seznamy procesu [ukazatele], ruzne si je projuje, nez je stihne propojit, prijde dalsi proces a muze sahnout do pameti kam nema),
- proto obsluha preruseni musi byt synchronizovana a v pripade, ze se v ramci preruseni provadi nejaka kriticka operace je nutne vyloucit ostatni (vsechna) preruseni

### 2.5.3 Pristupy k zakazovani preruseni

Pokud vsak zakazu (nejaka/vsechna) preruseni, abych se mohl venovat obsluze jednoho a budu ho obsluhovat prilis dlouho, system se muze dostat do nekonzistentniho stavu (jako u SMI). Pouzivaji se proto dva pristupy:

- je snaha zakazovat jen preruseni s nizsimi prioritami,
- rozdelit obsluhu preruseni do vice casti (urovni).

**Obsluha preruseni je casto delena na dve urovne:**

- 1. uroven:
  - ma byt co nejkratsi,
  - v ramci obsluhy preruseni se zakomunikuje nezbytnym zpusobem s HW (prevzani dat z/do HW, vydani prikazu HW, ..) a naplanuje se beh 2. urovne,
  - nelze pouzit bezne synchronizacni prostredky (protoze napr. CPU bezi nejaky vypocet, prijde preruseni z disku, jadro zacne resit 1. uroven obsluhy, nicmene obsluha != proces)
- 2. uroven:
  - dokoncuje obsluhu preruseni,
  - provadi se operace, kdy neni potreba komunikovat s hardware,
  - nemusí se zakazovat preruseni,
  - muze bezet v specialnich procesech (interrupt threads ve FreeBSD nebo tasklety/softIRQ v Linuxu),
  - mohou se pouzit bezne synchronizacni prostredky

### 2.5.4 Ovladace zarizeni a preruseni

- pri inicializaci ovladace (v Linuxu je to typicky modul) nebo pri jeho prvni pouziti se musi registrovat k obsluze urcitého IRQ,
- bud u nekterych zarizeni se pouzivaji (historicky) zafixovana cisla preruseni,
- nebo ovladac muze zjistit cislo preruseni tak, ze zakomunikuje s radicem sbernic, pokud to nefunguje,
- ovladac vyda prikaz zarizeni, ktere ma ovladat, aby zacalo vysilat nejaka preruseni (a "poslouchala" sbernici, "kdo se ozve"),
- pote se zaregistruje k obsluze prislusného preruseni a hardware se pres tabulku preruseni ovladac "dostane ke slovu",
- vice zarizeni vsak muze pouzivat stejne cislo zadosti o preruseni
  - v takovem pripade jadro vytvori zretezeny seznam ovladacu, ktere maji zajem o dane preruseni
  - ovladace musi byt napsane tak, ze pokud jim dojde preruseni (o ktere maji zajem), tak musi zakomunikovat s tim zarizenim a zeptat se ho, zda opravdu to zarizeni poslalo dane preruseni
  - pokud ano - obslouzi se, pokud ne - preda se rizeni preruseni dalsimu ovladaci v seznamu

### 2.5.5 Příklad komunikace s jádrem

Synchronní komunikace je proces-jadro, asynchronní je hardware-jadro. Příklad (detailnější, ale na téma přístupu na disk viz 5.9):

- proces A zavola službu read() a jádro ihned začne volání obsluhovat (synchronní)
- nejprve se podívá do cache zda data, o která má zájem proces A už tam nejsou
- 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 ovladače disku příkaz k načtení určitého objemu dat, typicky více než žádá uživatel a načítá do vyrovnávací paměti (ne na požadovanou adresu)
- na procesoru daleko bezí proces B, taky požádá o read(), zopakuje se to samé co u A
- až disk dokončí operaci jednoho z procesů (nemusí být v pořadí volání), disk pošle přerušeni 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 bezí dál, to samé se stane u dalšího procesu

#### definice (pro 2.5.x):

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

*synchronní* zn., že CPU něco provede a ihned se zavola přerušeni (např. dělení 0)

*IRQ* = interrupt request

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

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

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

*vypadek stranky* zn., (paměť je rozdělena na části, které mohou být rozděleny na disk) když proces bude sahát do paměti a sahne na stránku, která v ní není - detekuje se že stránka tam není - porušení ochrany paměti - jádro zkontroluje, zda proces nesáhá kam by neměl, a pokud ne, tak mu stránku nahraje zpět do paměti a znovu se provede ta stejná instrukce

*bezne synchronizační prostředky* jsou např. semaforey nebo zamky a synchronizují procesy

#### linux:

základní statistiky o obsluze přerušeni jsou v */proc/interrupts*

## **2.6 Nastroje programatora UNIXu**

X-Window system, vzdaleny pristup pres X-Window uzitecne prikazy na linuxu, ovladani vimu, apod. - vice viz. 2. prednaska IOS, u zkousky to nebyva.

## **3**

### **3.1 Bash, shell, experimenty**

## **4**

### **4.1 Bash, shell, experimenty**

Treti a ctvrtá prednaska je venovana hlavne shellu, prochazi se prakticky ruzne prikazy a provadi se experimenty, apod. - lepsi je shlednout + na zkousce nic takoveho nebyva.

## 5

**Pata prednaska:** Sprava souboru: pevný disk, diskové sbernice, sektory, parametry pevných disků, SSD, problematika zápisu SSD, zabezpečení disku, disková pole (RAID), uložení dat na disk, fragmentace, přístup na disk a jeho plánování, logický disk.

### 5.1 Pevný disk

#### Popis:

- uvnitř mají radu kulatých ploten, zaznam se provádí na každém z těchto dvou povrchů, je v soustředných kruznicích (= tracks, stopy)
- všechny plotny jsou na stejné ose, přidelané 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 nastaveny na sadě několika stop (kruznic) o stejném průměru = cylindr,
- stopy se dělí na sektory
- velikosti sektorů byly dříve 512B, u CD/DVD 2048B, dnes 4096B

#### Adresace sektorů:

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

#### Periferní či disková rozhraní:

- používají se pro připojení disku,
- nejbezpečnější 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 sbernice, jako třeba AHCI, OHCI, UHCI, ..

### Diskove sbernice se lisi:

- rychlosti (SATA do 6 Gbit/s, SAS 22.5 Gbit/s),
- poctem pripojenych zarizeni (SATA desitky, 65535 SAS),
- maximalni delkou kabelu (1-2m SATA, 10m SAS),
- architekturou pripojeni (moznost pripojeni jednoho zarizeni vice cestami u SAS),
- seznamem prikazu, ktere to zarizeni umi (flexibilita pri chybach, selhani, zotaveni, ..)

Pres diskove sbernice je mozne mit pripojene i jine typy pameti, jako jsou flash disky, SSD, pasky, CD/DVD/BD ci terciální pameti. V systému vzniká hierarchie pameti, viz. 1.6.

### definice:

*cylindr* (v HDD) je množina stop o stejném průměru

*sektor* je nejmenší jednotka diskového prostoru, který mi umožní disková elektronika nacist nebo zapsat

*blok* nebo *diskový blok* je sektor v HDD

*alokací blok* nebo *blok souborového systému* je nejmenší jednotka, kterou umožní alokovat OS

*CHS* zn. Cylinder Head Sector

*LBA* zn. Linear Block Address

## 5.2 Parametry pevných disku

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

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

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

### definice:

*doba vystavení hlaviček* zn., že pokud nejsou nastaveny hlavičky na stope, se kterou chci pracovat (malokdy), tak je nutné pohnout hlavičkami (víc zasunout dovnitř nebo vysunout)

*rotací zpoždění* je doba než mi pod správně nastavenou hlavičku najede sektor (narotuje se disk)

*maximum transfer rate* je spíková přenosová rychlost, jak maximálně rychle je schopen disk komunikovat po krátkou dobu (typicky rychlost předání dat z vyrovnávacích pamětí disku)

*sustained transfer rate* opravdová rychlost čtení z ploten

### linux:

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



## 5.3 Solid State Drive - SSD

Mohou byt zalozena na ruznych technologiich, nejcasteji na nevolatilnich pametech NAND flash nebo DRAM (se zalohovanyim napajenim) ci na kombinacich.

### 5.3.1 Klady a zapory SSD

#### Vyhody:

- rychly (okamzity) nabeh,
- nahodny pristup (mikrosekundy),
- vetsi prenosove rychlosti (stovky MB/s, ATA do 600MB/s, 3.5GB/s s M.2, 7GB/s s PCI Express 4),
- zapis muze byt mirne pomalejsi,
- tichy provoz, lepsi mechanicka a magneticka odolnost,
- obykle nizsi spotreba (neplati pro DRAM).

#### Nevyhody:

- vyssi cena za jednotku prostoru,
- omezeny pocet prepisu (nevyznamne pro bezny provoz),
- vetsi riziko katastrofickeho selhani,
- mensi vydrz mimo provoz (pri vyplem napajeni a skladovani),
- komplikace se zabezpecenim (bezpecne mazani nebo sifrovani prepisem dat - vyžaduje specialni podporu).

### 5.3.2 Problematika zapisu u SSD

NAND flash SSD jsou organizovany do stranek (typicky 4KiB) a ty jsou sdruzeny do bloku (typicky 128 stranek = 512 KiB).

#### Zapis nebo prepis dat:

- prazdne stranky lze zapisovat jednotlivé (prepisovat ne!),
- pokud chci prepisovat (jednu stranku), je nutne cely blok nacist do pameti, vymazat (zresetovat) a v pameti upraveny blok nacist zpet (= write amplification, zesileni zapisu - mnohonasobne zpomaleni),
- problem je mensi pri sekvencnim (pockam az budu mit dost dat tak aby pokryly blok) nez pri nahodnem zapisu do souboru.

#### Problem se sifrovanim a bezpecnym mazanim:

- diky tomu jak SSD prepisuji data se data nekolikrat presouvaji po disku,
- proto disk musi poskytovat hw podporu pro bezpecne mazani nebo sifrovani.

#### Reseni problemu prepisu u SSD:

- typicky ma SSD vice stranek-bloku nez je deklarovana kapacita (pri prepsani se zapise do volne stranky),
- po smazani dostatku stranek (tak ze tvori blok) se blok zresetuje - prikazem TRIM souborovy system sdeli SSD, ktere stranky jiz nejsou pouzivane (a ktere bloky muze SSD smazat),
- radic SSD muze stranky presouvat tak, aby si nektare bloky uvolnil (pokud je v bloku malo stranek, presunou se a blok se zresetuje),
- TRIM nelze pouzit vzdy (typicky pokud v souborovem systemu mame obraz jineho souboroveho systemu, nemusi byt mozne sdelit zakladnimu filesystemu informace o praznych blocich, apod. nebo data-baze, ktere si ukladaji data do velkeho predalokovaného prostoru, ci obrazy virtulanich stroji a virtualni disky)

Radic SSD presouva i dlouho nezmenene stranky, aby minimalizoval pocet prepisu stranek.

#### **definice:**

*nevolatilni* zn., ze pokud se vypne napajeni, tak obsah zustane zachovan (alespon po nejakou rozumnou dobu)  
*stranka* je nejmensi jednotka dat, kterou lze do SSD zapsat

## **5.4 Zabezpeceni disku**

Diskova elektronika typicky na ukladana data (sama o sobe) zabezpecuje kody, ktere umi pri naslednem cteni detekovat a pripadne opravit chyby - pouziva ECC. (detekce a oprava chyb je pouze v rezii disku, pokud disk detekuje chybu a neni prilis velka, chybu opravi a data ulozi na jiny sektor, poznaci si, ze ten sektor nema pouzivat)

Existuje technologie, ktere umoznuji zjistit, v jakem stavu disk je (statistiky, premapovani, pocet chybných sektorů, ..) - S.M.A.R.T (podporovana vsemi "rozumnymi" disky)

Pak je mozne jeste provadet testovani na urovni OS, napr. e2fsck nebo badblocks nebo si nektare filesystemy (RFS, ZFS) provadeji kontinualni kontroly toho, co se ve filesystemu deje. Tyto utility nebo filesystemy mohou chyby detekovat (a varovat) nebo opravit (pokud neni chyba prilis velka) ci vyradit pouziti nekterych sektorů.

#### **definice:**

*ECC* = Error Correction Code

*S.M.A.R.T* = Self Monitoring Analysis and Reporting Technology

*kontinualni kontroly (fs)* zn., ze si ukladaji sve dalsi kontrolni soucty, a pote si kontroluji pri praci se souborem, zda kontroly souhlasí

#### **linux:**

*smartctl* je prikaz umoznujici vyuziti technologie S.M.A.R.T (testy disku, statistiky, ..)

*smartd* je nadstavbou smartctl (pravidelne spousteni testu, ..)

## 5.5 Diskova pole (RAID)

RAID je technologie umoznujici z vetsiho poctu (levnejsich a ne prilis spolehlivych, vykonnych) disku vytvorit jeden disk, který je rychlejsi a spolehlivejsi.

### Muze byt implementovan:

- hardwarove (do rozsirujici karty pripojime nekolik disku a ta implementuje RAID),
- subsystemem v jadře,
- nektère souborove systemy maji implementaci RAID v sobe.

Ruznych typu RAID je nekolik (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 (napr. 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, přijde 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 určen počet zabezpečovacích disků,
- data se ukládají na datových discích na úrovni bytu, k nim se dopocítávají zabezpečovací kódy (napr. 4 datové - 3 zabezpečovací),
- byty dat se rozloží do všech disků (ofč ty se musí převést do bajtu a sektorů a zapisuje se to po sektorech)
- jediný RAID, který umí detekovat chyby, nektère i sám opravit, dokonce umí i zjistit, který disk selhal

#### 5.5.4 RAID 3

- jednoduší zabezpečení než RAID 2, v podobě paritních bytů,
- rozkládá data po bajtech či skupinách bajtů, které zabezpečuje paritním zabezpečením (např. 4 disky - 3 datové a 1 paritní).

#### 5.5.5 RAID 4

- je analogie (tak jako RAID 3, akorát ..),
- provádí se rozkládání na úrovni bloku-sektoru,
- 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 máme 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. .. = 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 sektoru (první bit 1. sektoru, první 2. sektoru, ..),
- pokud selže více disků, nelze dopocítat bity (data)

#### 5.5.7 RAID 6

- parita se ukládá 2x,
- dokáže se vyrovnat se selháním až 2 disků,
- větší redundance dat (obětuje 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 používají RAID 3 - 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 (suda - v paritním je 0)
  - selže třetí disk, vymění se za nový, prázdný
  - dopocítají se data opět na sudou paritu: mám první byty 01? a v paritním 1 - aby byla suda, 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 (bitu)* je sudost/lichost bitu, počet sudých/lichých 1 bitu

*parity bit* zn., že na MSB se přidá 1 pokud počet 1 (bitu) je lichý

## 5.7 Uložení dat na disku

Disková jednotka neumí pracovat s ničím menším než sektor, ale typický OS si sektory nějak seskupí (do větší jednotky) a neumí 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 (filesystem 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 diskovou diskového prostoru, se kterým bezne pracuje jádro (filesystem, uživatel).

## 5.8 Fragmentace

### 5.8.1 Externi fragmentace

Rozumíme jev, který vzniká v pamětech postupným obsazováním a uvolňováním paměti, kdy v paměti vzniká sekvence oblastí, které jsou volné a použité (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 uložen na jednom místě, např. s1.1, s2, s1.2 nebo viz příklad výše).

#### 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 alokaci 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 mohou ukládat pomocné informace, k čemu se používají - pokud bude informace větší než "volná díra" - nepoužitelná) - mám na disku bloky volného místa o požadované velikosti (1GB, soubor, 0.5GB), ale protože chci ukládat spojitě (soubor o velikosti 1.5GB), 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 metadata - čím více fragmentované - tím více je přístup na data pomalejší (u HDD se čeká navíc na natocení hlaviček a rotaci disku)

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

- rozložení souboru po disku (snáze ukládat soubory na disk tak, aby nebyly nutné za sebou, ale bylo mezi nimi volný prostor),
- používání předalokace (uživatel si požádá filesystem o vymezení určitého prostoru na disku, např. databáze),
- odložená alokace (=allocate-on-flush, filesystem nezapíše ihned po změně souboru, ale chvíli počká - počítá s tím, že uživatel bude chtít menit soubor "za chvíli" znovu - až nebude delší dobu docházet ke

zmenam, pote hleda vhodny volny prostor)

Pri (intenzivnim) beznem pouzivani disku se vsak fragmentaci nelze vyhnout. Pokud by byla fragmentace prilis vyrazna, je mozne pouzit defragmentacni nastroje, které provadeji kopirovani, presouvani casti souboru a reorganizaci diskoveho prostoru tak, aby se fragmentace odstranila - casove narocna operace.

Prvniho negativniho dopadu externi fragmentace (nevyuzitelne a prilis male oblasti) je mozne se zbavit pri pouzivani alokaci po jednotkach pevne velikosti - alokacni bloky - vzdy je ale snaha alokovat spojite (v horsim pripade alokuji nespojite, pokud to nejde)

### **5.8.2 Interni fragmentace**

Nespojita alokace po jednotkach pevne velikosti (alokacni bloky) ma vyhodu, ze redukuje dopady externi fragmentace, ale potom vytvari interni fragmentace. Interni fragmentace se obvykle toleruje.

#### **Priklad interni fragmentace:**

- chci alokovat soubor o velikosti 9 000 B,
- mam 4 KiB velke alokacni bloky,
- potom je nutne alokovat 12 KiB pro tento soubor,
- ty zbyvajici 3 KiB v poslednim alokacnim bloku zustanou nevyuzite.

Existuje nekolik malo filesystemu, které se snazi resit interni fragmentaci (ReiserFS, ZFS) pomoci techniky zvane 'tail packing' ('zbavovani ocasku"souboru) - vice souboru muze pouzivat 1 fyzicky alokacni blok (zaplni se volne misto). Vetsine filesystemu toto vsak nepodporuje.

## 5.9 Prístup na disk

(viz 2.5.5) Proces když chce načítat/zpracovávat data, 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 paměťové mapované I/O porty, disku se předávají příkazy přes jeho rozhraní (ATA disk - ATA příkazy), jdou z filesystemu přes ovladač příslušného disku (pote to prochází sbernicemi), ten komunikuje s radicem disku - disk dostane příkaz, jakmile dobehne operace, disk posle prerušení na procesor, tam se dostává ke slovu jádro, to zpracuje prerušení a zachová se podle něj (úspěch - předá data, chyba - zpracuje ji).

### definice:

*ovladac* je software, který umí komunikovat s určeným typem zařízení, jina definice viz. 2.4

## 5.10 Planování přístupu na disk

Součástí jádra je subsystem nazývaný plánovač diskových operací, který shromažďuje požadavky od filesystemu (načtení, zápsání dat z/do disku). Plánovač si ukládá požadavky do svých plánovacích front, požadavky případně přeuspořádává a předává dal ovladači či radici disku k realizaci.

Plánovač se snaží minimalizovat rezii disku.

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

- snaha, aby se hlavička disku plynule pohybovala od středu k okraji a zpět a vyřizovat požadavky dle pohybu hlavičky,
- modifikace SCAN algoritmu je například Circular 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 vyřizovat požadavky jdoucí od jednotlivých uživatelů (procesů),
- implementace priorit (prioritnější proces - požadavky se vykonají dříve),
- snaha odkladat operace tak (v naději), že je bude pote možné sloučit,
- snaha implementovat časová omezení na dobu čekání požadavku,
- může implementovat paralelizaci požadavků předávaných do diskového subsystemu (modernější a velmi výkonné SSD umí resit operace paralelně).

### linux:

pro zjištění, jaký plánovač používáme se stací podívat do `/sys/block/<devname>/queue/scheduler`



## 5.11 Logický disk

V pocitaci 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 (nulém) sektoru byla tabulka obsahující rozdělení na 1-4 primární partitions
  - pokud bylo nutné použít více partitions, potom místo primární se nahradila rozšířenou diskovou oblastí, která se dále mohla rozdělit na podoblasti zvané logické diskové oblasti, každá z nich popsána formou zřetězeného seznamu, EBR
  - používane 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 filesystému nebo v části jádra (mezi filesystémem a plánovačem).

### 5.11.3 Různé typy souborových systémů

- fs (první fs na Unixu), ufs, ufs2,
- ext2, ext3, ext4,
- btrfs (inspirovan ZFS),
- ReiserFS, HSF+/APFS (Mac OS X), XFS, JFS, HPFS,
- FAT, VFAT, FAT32, exFAT (rodina FAT vznikla v MSDOS, poté používány ve Windows - velmi jednoduché a široce podporované),
- F2FS (fs pro efektivní práci se SSD), ISO9660, UDF, Lustre, GPFS (clustery, superpocítace),
- 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 nízkourovnňové formátování (staré disky s nestabilním magnetickým záznamem).

#### 5.11.4 Chyby disku (souvislost s FS)

Na disku mohou vznikat chyby beznym opotrebenim, nevhodnym vypnutim napajeni, je zapotrebi opravit ridici struktury souboroveho systemu (program fsck - kontroluje konzistenci filesystemu nebo zurnalovani, copy on write, soft updates, ..).

#### 5.11.5 Dalsi typy souborovych systemu

**Virtualni souborovy system (VFS)** je vrstva, která v jadře zastresuje vsechny ostatni souborove systemy z toho duvodu, aby jine subsystemy jadra nemusely pracovat specialnim zpusobem s ruznymi souborovymi systemy. (viz take 8.2)

Existuji take ruzne sitove souborove systemy, treba NFS. (viz take 8.3)

#### Specialni souborove systemy

- neukladaji zadna data, obsah neni nikde na disku ani neexistuje zadna specialni cast pameti
- zpristupnuji napr. aktualni stav jadra - adresar /sys, sysfs filesystem,
- procfs filesystem v adresari /proc zpristupnuje informace o bezicich procesech (ale i o nejakych castech stavu jadra),
- tmpfs zase vytvari souborovy system v RAM.

#### definice: (pro 5.11.x)

*logicky disk* je taky diskova oblast, partition

*MBR* = master boot record

*EBR* = extended boot record

*GPT* = GUID Partition Table, GUID = Globally Unique Identifier

*LVM* = Logical Volume Manager

*formatovani* zn., ze se nainstaluji metadata (ridici data) souboroveho systemu do prislusne diskove oblasti, v ramci toho se mohou vymazat vsechna data na dane oblasti

## 6

**Šestá přednáška:** pokračování Spravy souborů. Zurnalování, jeho implementace a alternativy, Copy-on-write, Klasický UNIXový systém souborů FS, i-uzly, kde a jak jsou data uložena, počty odkazu, limit maximální velikosti souborů, 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 Zurnalování

Je technika založena na vytváření zurnálu.

- souborové systémy se zurnalem jsou třeba ext3, ext4, ufs, XFS, JFS, NTFS, ..
- zurnalování 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 zurnalována nejsou (velká režie), ale mohou být
- závisí na tom, že operace, které zurnalování implementují, se provedou ve správném pořadí – nutnost spolupráce s plánovačem, takže disky si samy data přeuspořádávají (nelze nijak ovlivnit)

#### Zurnál:

- zápis zurnálu je předřazený,
- vytváří se v něm cyklický prepisovaný buffer,
- předřazenost zápisu do zurnálu mi zaručí, že operace pokryté zurnalováním jsou atomické - vytváří transakce

Kompromis mezi zurnalováním a nezurnalováním dat je **předřazení zápisu dat na disk před zápisem metadat do zurnálu** (a následně zápis ostrých metadat na disk). Příklad:

- zapisují do souboru - buď vytvořím zcela nový nebo ho zvětšuji (typické způsoby zápisu),
- při zvětšování se nejprve zapisují 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 zurnálu 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 zurnálu, ale ne na disku, je možné tyto metadata obnovit

#### Proces mazání souboru na disku:

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

- uvolnění oblasti použitých tím souborem

#### **definice:**

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

*předřazený* znamená, že zápis do zurnálu se provede před ostrým zápisem "užitečných" dat (či metadat) na disk  
*atomicke operace* zn., že buď operace uspěje celá (všechny dílčí kroky) nebo neuspěje vůbec (žádný dílčí krok)

### **6.1.1 Implementace zurnalování**

Existují 2 základní přístupy k implementaci zurnalování.

#### **REDO:**

- implementace na základě dokončení transakcí,
- používá např. ext3, ext4,
- sekvence dílčích operací (vytvářející tu operaci, kterou chci provést) se zapisuje do zurnálu (začátek, konec transakce, kontrolní součet),
- poté se operace provádí na disku,
- po úspěšném dokončení se transakce ze zurnálu uvolní,
- při selhání a poté zotavení se systém podívá do zurnálu, podívá se po neuvolených transakcích, jestli jsou celé - počáteční a koncová značka, jestli sedí kontrolní součet - pokud vše sedí, tak 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 zurnálu a následně jejich provedení na ostrých datech (zaznamená dílčí operaci - provede ji),
- proběhne celá transakce - záznam ze zurnálu se uvolní,
- při chybě se eliminují všechny nedokončené transakce (všechny provedené dílčí kroky se musí vrátit - vrátí se disk do původního stavu)

Při implementaci zurnalování je klíčové **dodržení poradi kroku, ve kterém se provádějí**. (U REDO např. je nutné, aby se nejprve zapsaly sekvence operací do zurnálu a teprve poté se prováděly operace na disku) Pokud tato sekvence nebude dodržena, zurnal nebude správně fungovat.

### 6.1.2 Copy-on-write

Je alternativa k zurnalovani pouzivana napriklad v ZFS (OpenZFS), BTRFS, ReFS (Resilient File System).

- kopie pri zapisu,
- zalozeno na tom, ze vsechna nova data / metadata se zapisi na disk, a pote se zpristupni,
- vyuziva se pritom toho, ze obsah disku je popsam hierarchickou stromovou strukturou,
- zmeny se provadeji v souladu s touto strukturou (od listu ke koreni),
- pokud vypadne napajeni v moment, kdy data (bloky) nejsou jeste zpristupnena, data nejsou dostupne z korene stromu a jakoby se nic nestalo,
- pokud se mi tyto data podari zapsat uspesne, postupne zacnu upravovat vsechny uzly vedouci az ke koreni a zpristupnim nova data,
- teprve po modifikaci korenu se stanou zmenena data (uzly) dostupne
  - koren je nutne zabezpecit, aby nedoslo k chybe pri zapisu do nej
  - stary korenovy uzel se neprepisuje, ale pouze se tam zapise nova verze korenoveho zaznamu (s casovym razitkem)
  - soucasne tam bude zabezpecovaci kod (kontrolni soucet),
  - pokud dojde ke krachu systemu, staci si nacist vsechny koreny, zkontrolovat kontrolni soucty, vybrat si vsechny, kde sedi kontrolni soucty, s nejnovejsim casovym razitkem a tyto pouziju (pokud dojde k chybe nez se stihne zapsat novy koren, pouzije se ten puvodni)

#### Vyhodami copy-on-write jsou:

- snimky souboroveho systemu (zapamatuje se pouze korenovy uzel - minimalni rezie),
- klony souboroveho systemu (vytvori se pozadovany pocet kopii korenoveho uzlu),
- vyhodou je, ze nezmenene uzly (data) a listy budou na disku pouze jednou, pouze je nutne si pamatovat zmenene uzly / listy a cestu ke koreni + koren (kopie stale ukazuji na stejny strom).

#### definice:

*stromova struktura* (copy-on-write) je vyhledavaci strom, který popisuje veskerý obsah disku, typicky se v něm vyhledava na zaklade unikatni identifikace souboru

*adresare* (copy-on-write) jsou specialni soubory ulozeny na disku, které jsou dostupne ve strome (stromove strukture)

*snimek souboroveho systemu* ulozi se obsah disku tak, ze je mozne se k nemu pozdeji vratit

*klon souboroveho systemu* je vytvoreni 2 kopii souboroveho systemu a od daneho okamziku je mozne s kazdou kopii pracovat samostatne (napr. pri vetsim poctu VM, kdy vsechny VM sdili stejny pocatecni obsah disku, ale od urciteho momentu kazda VM chce obsah menit samostatne)

### 6.1.3 Dalsi alternativy zurnalovani

#### Soft updates:

- pouziva se v UFS (FreeBSD systemy),
- filesystem se snazi sledovat zavislosti mezi tim jaka data a metadata se meni,
- uzpusobuje poradi zapisu metadat a dat na disk tak, aby v jakemkoli okamziku byl obsah na disku konzistentni (az na moznost vzniku "garbage")

#### Log-structured file systems:

- logovaci souborove systemy (= strukturovane jako log),
- pouziva se v LFS, UDF, F2FS,
- cely souborovy system ma charakter jednoho velkeho logu,
- ktery se zapisuje v cyklicky prepisovane pameti napric celym diskem,
- posledni obsah disku je vzdy dostupny pres posledni zaznam (a odkazy, ktere z nej vedou),
- pri provadeni zmen se pridaji data napr. za aktualni konec vyuziteho diskoveho prostoru, prida se k tomu zaznam (o tom co se zmenilo), zpristupni se data z posledniho zaznamu.

#### definice:

"garbage" je cast prostoru na disku, ktera se tvari jako obsazena, ale neni

*logem* rozumime soubor, ktery obsahuje zaznamy o zmenach (nebo: log = zapis o zmenach)

## 6.2 Klasický UNIXový systém souboru (FS)

Je původní filesystem unixu (70. leta). Vyvinul se z něj UFS, z něj zase EXT2, 3 (pote vznikl i EXT4).

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

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

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

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

**definice:**

*i-uzel* je základní datová struktura reprezentující každý jeden soubor v typických UNIXových systémech (pozn. při formátování se určí dopředu maximální počet souboru, které na diskovém oddílu budou existovat)

### 6.2.1 i-uzel

Základní datová struktura popisující soubor v UNIXu (nebo viz definice hore). Ke každému souboru musí být i-uzel. Ten obsahuje metadata o souboru:

- stav i-uzlu (alokovaný, volný)
- typ souboru (obyčejný, adresar, zařízení, pojmenovaná roupa, ..),
- délka souboru v bajtech,
- čas 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 souboru),
- 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, ..)

Jmeno souboru neni v i-uzlu, ale je ulozeno v adresari.

#### definice:

*UID* je cislo identifikace vlastnika

*GID* je cislo identifikace skupiny

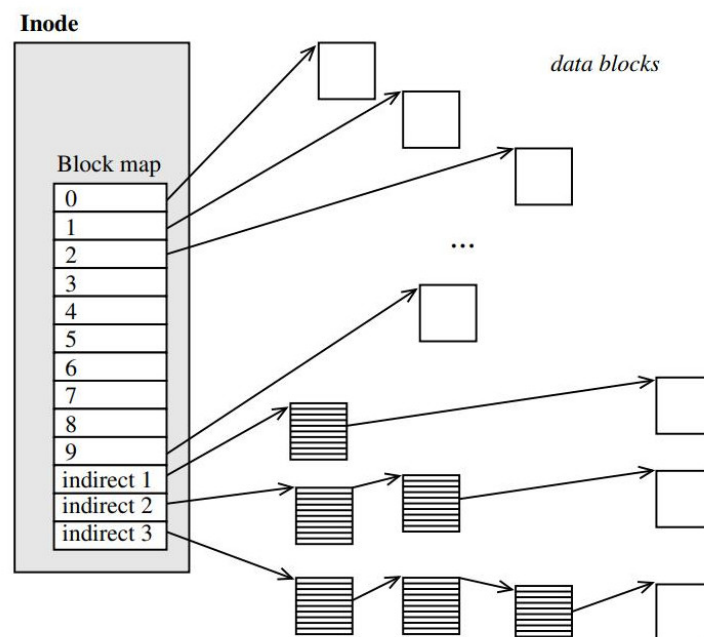
*ACL* zn. Access Control List, pristupove seznamy rozsirujici zakladni UNIXova prava tak, ze je mozne priradit konkretni prava ke konkrétním uživatelům

*extended attributes* zn. rozsirene atributy, s jakými specifickými práve se může soubor napr. spouštět

*ctime* je poslední změna i-uzlu, využitelné napr. 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 rada primych (az 10, novejsi unixove fs maji 12) i neprimych odkazu na data,
- prime odkazy odkazují na alokacni bloky na disku,
- pokud je potreba vice odkazu, pouzije se neprimy odkaz první urovne, který odkazuje na specialni alokacni blok neobsahující data, ale dalsi prime odkazy na data,
- pokud nestaci ani to, pouzije se neprimy odkaz druhé urovne, který odkazuje na pomocny adresovaci blok, který obsahuje dalsi neprime odkazy 1. urovne, které odkazují na dalsi prime odkazy (a ty odkazují na data) - vzniká strom
- pokud ani to stacit nebude, pouzije se adresovaci blok 3. urovne, vedouci na adresovaci bloky s neprimými odkazy 2. urovne, každý z nich na bloky s odkazy 1. urovne, ty vedou na prime odkazy a ty na data.



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



### 6.2.3 Počty odkazu

- nepřímý odkaz 1. úrovně - při 4 KiB clusteru je to 1024 odkazů (1 odkaz = 4B) = 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

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

$$10 * D + N * D + N^2 * D + N^3 * D$$

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

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

- dalšími datovými strukturami a typy, které používá FS (např. datový typ délky souboru v bajtech v i-uzlu),
- strukturami VFS (veskera práce s jakýmkoli filesystemem 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 nebo dnes bezpečná 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 jak 2 GiB*.

#### linux:

`du [soubor]` vypíše zabrané místo v blocích včetně (metadat)

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

`df` vypíše volné místo na discích

`ls -li [soubor]` zprístupní číslo i-uzlu souboru

`ls -le /dev/... n` - vypis i-uzlu n na /dev/...

`dumpe2fs` - základní informace o souborovém systému ext2,3,4

`/dev/zero` je soubor typu zařízení generující proud 0

`dd if=[source] of=[dest]` je nízkourovňové kopírování

### 6.2.5 Vyhody a nevahody architektury FS

Neboli proc bylo navrzen FS prave tak, jak je. Architektura FS je totiz ovlivnena snahou o minimalizaci jejich rezie s relativne pomalymi disky (HDD, SDD), jedna se zejmena o bezne operace se soubory, jako je pruchod soubourem (otevru - prochazim od zacatku do konce) ci presun (seek), zvetsovani ci zmensovani (vc. mazani) souboru.

**Je nutne vzit do uvahy, z jakych (mikro)operaci se tyto operace sestavaji.** Jsou to operace:

- vyhledavani adresy prvnio nebo urcitedho bloku souboru,
- vyhledavani nasledujicich bloku,
- pridani ci odebrani bloku,
- alokace ci dealokace volneho souboru (informace o volnych oblastech, minimalizace externi fragmentace)

FS a jeho naslednici UFS, EXT2, EXT3 (EXT4 uz neni jeho naslednik!) predstavuji kompromis s ohledem prevazne na male soubory. (tyto fs funguji skvele pro male soubory - u vetsich souboru je nutne prochazet ci menit vetsi objem metadat)

Jistou optimalizaci pouzivanou i u klasickych filesystemu pro male soubory je ulozeni dat primo do i-uzlu. (pokud se tam data vlezou).

#### **definice:**

*symbolicky odkaz* je soubor odkazujici na jiny soubor (pouziva ulozeni dat primo do i-uzlu)

*rychle symlinky* maji data v i-uzlu

*pomale symlinky* maji data mimo i-uzel

## 6.3 Jine způsoby organizace souboru

### 6.3.1 Kontinuální uložení

- neboli spojitě uložení souboru na disku,
- na disku je jeden spojitý úsek dat reprezentující soubor,
- výhodami jsou rychle vyhledání adresy 1./určitého bloku nebo vyhledávání následujících bloků,
- nevýhody: soubory nebude možné jednoduše zvětšovat pokud budou příliš blízko u sebe (bude nutné je 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 Zrežované seznamy alokací bloků

- každý alokační blok obsahuje své (užitečné) 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 přejít celý soubor až po daný blok (1 GiB soubor, chci poslední blok - musím přejít celý),
- další nevýhodou je rozptýlení 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 vhodná, nicméně se používá v souborových systémech FAT

### 6.3.3 FAT

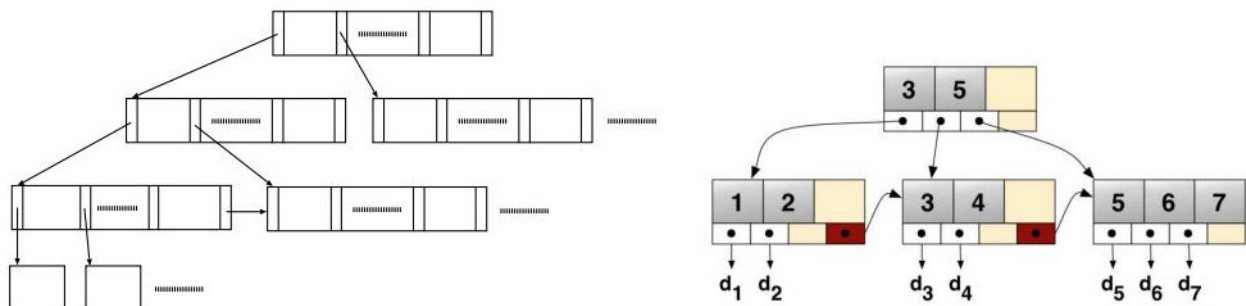
- File Allocation Table,
- od zrežovaných seznamů se liší tím, že seznamy popisující rozložení souboru na disku jsou uloženy v separátní oblasti na disku (tzv. FAT),
- kde jsou tato data koncentrována - rychleji se procházejí, lze vytvořit tak více kopií FAT (prevence okamžité ztráty dat při chybě),
- stále vznikají problémy s rychlostí při náhodném přístupu (stále jde o zrežovaný 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 rozsáhlé), protože je to jednoduché (např. vestavěné systémy)

### 6.3.4 B+ stromy

- jsou datovou strukturou převzatou z databazových systému,
- mají dva typy uzlu - vnitřní a listové,
- vnitřní uzly jsou koreň, jeho následníci kromě listových, obsahují odkaz na následníka a vyhledávací klic,
- 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 klic bude offset - číslo logického bloku v rámci souboru) nebo se používají pro adresare (klíče budou jména souborů) nebo pro popis celého obsahu disku (klic je dvojice i-uzel a posuv souboru)

#### Vyhledávání v B+ stromu:

- při hledání klíče (k) se podívám, zda je klic menší jak klic k<sub>0</sub>, pokud ano, půjdu níže, kde je k<sub>0</sub>, pokud ne, zjistím, jestli je klic mezi k<sub>0</sub> a k<sub>1</sub>, pokud ano, jdu druhým směrem, opakuji po k<sub>n</sub>,
- pokud jsem níže, opakuji to samé co vys až nedojdu k listovému uzlu,
- zde hledaný klic najdu nebo zjistím že v této struktuře klic není
- poté mám odkaz na datový blok,
- v případě že chci číst dál, tak jdu lineárně po sobě po následujících listových uzlech



Obrázek 2: z prezentace IOS: Souborové systémy, slide 27 - B+ strom

### Prace s B+ stromy:

- jsou zde limity jak moc/malo mají být uzly zaplneny (strom se udržuje vyvážený) - pro uzly s  $m$  odkazy máme klice 0, 1, .. až  $m - 2$  klicu (odkazu je o 1 méně než klicu + číslování od 0),
- pokud je strom tvořen solo korenem - nejméně může mít 1 odkaz, maximálně  $m - 1$  odkazu (poslední odkaz je použit jako ukončovač seznamu listů),
- pokud to není solo koreň, tak má nějaké následníky, minimálně jich má tak 2, maximálně  $m$ ,
- vnitřní uzel má tak  $\frac{m}{2}$  (zaokr. nahoru) až  $m$  odkazů, list  $\frac{m}{2} - 1$  (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,
  - ... štepí se strom až případně se rozštěpí kořen a strom bude mít 2 kořeny
- rusení:
  - se opět od listové úrovně, tak, že se zruší odkaz v listu,
  - zkontroluje se, zda je uzel zaplněn v rámci daných limitů,
  - pokud ano - gut, pokud ne - podívám se na sousední uzly a pokud se provede přerozdělení tak, aby byly všechny uzly naplněny v rámci limitů,
  - pokud se to nepodaří, tak dojde ke sloučení listů,
  - posunu se o úroveň výš, zruším jeden odkaz, zkontroluji 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 filesovech jako XFS, JFS, ZFS, Btrfs, ReFS, .. v omezené podobě tzv. stromu extentů v EXT4, podobná struktura je i v NTFS

### definice:

*solo koreň* = jediný kořen

### 6.3.5 Extent

- pouziva se ke zrychleni prace s velkými soubory,
- umožnuji zmenšit objem metadat (je možné říct, že některé alokační bloky jsou uloženy pospolu = vytváří extent), potom budu popisovat rozložení souboru 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 klacický Unixový strom - protože ve stromu jsou explicitně uloženy vyhledávací klíče, ale Unixový nemá v žádných strukturách [i-uzel] uloženou velikost datového bloku [protože jsou všechny stejné a konstantní])

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

#### **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

Pouziva pro popis rozlozeni dat na disku strom extentu. Pro "male soubory"(mysleno soubory, na ktere je mozne se odkazovat az 4 extenty, pak tyto extenty budou odkazovane primo z i-uzlu; tldr soubory s malym poctem extentu)

### definice:

*strom extentu* je v principu B+ strom degradovany na maximalne 5 urovni, bez pouzivani vyvazovani (napr. prerozdelovani uzlu pri mazani) a zretezeni listu

## 6.5 NTFS

Zakladni datovou strukturou popisujici disk je MFT - Master File Table (ma pro kazdy soubor alespon 1 radek, na 0. radku popisuje samo sebe, 1. radek pripadne kopie MFT, pripadne metadata, pote obsahy souboru).

### Obsah souboru muze byt reprezentovan bud:

- pokud jde o kratky soubor, bude ulozen v MFT v jeho radku (vcetne metadat),
- soubor je rozdelen na extenty (ty jsou odkazovane primo z radku souboru v MFT, tak ze v radce souboru jsou informace o pocatecnim VCN a LCN a pocet clusteru, ktery dany extent obsahuje) - vyhledava se v tom stejne jako v B+ stromu
- pokud je extentu potreba vice nez se vleze na jeden radek, alokuji se pomocne radky (z hlavniho radku vedou odkazy na pomocne, z pomocnych vedou odkazy na disk) - prochazeni je opet ve stylu B+ stromu

### definice:

*VCN* - virtual cluster number, logicky blok souboru

*LCN* - logical cluster number, cislo fyzickeho bloku (souvisi s tim ze je to na logickem disku)

## 6.6 Organizace volneho prostoru na disku

V klasickem Unixovem FS a rade jeho nasledovniku (UFS, EXT2, 3), take v NTFS se pouzivaji bitove mapy, kde pro kazdy blok mam 1 bit. V bitove je mozne pote vyhledavat pomoci bitovych mask - zrychli vyhledavani.

### Dalsi mozne zpusoby organizace volneho prostoru:

- pouziji se alokacni seznamy (zretezeni volnych bloku na disku),
- zretezeni volnych polozek v tabulce (FAT),
- B+ strom (udrzovani informaci o tom, kde je volne misto, adresace velikosti a/nebo offsetem)
- volny prostor muze byt take organizovan po extentech.

## 6.7 Deduplikace

- podporována ZFS, NTFS, Btrfs, XFS, (ext4 ne) ..
- snaží se odhalit opakované ukládání stejných dat na disk a uloží je pouze jednou a odkazuje se na ně vícenásobně.
- 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 ušetřit diskový prostor (při virtualizaci, na mail serverech, repozitářích, ..), paměťový prostor i čas (omezí 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 filesystemu (např. vytvoření virtuálu),
- zatímco deduplikace aktivně vyhledává duplikáty (např. uživatel, co stahuje stejné reklamní letáky)

## 6.8 Typy souborů v UNIXu

- - je obvyklý 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, hw, paměť) se kterým se komunikuje po blocích nebo znacích

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



## 6.9 Adresar

Je to kolekce jinych souboru na nejvyssi urovni abstrakce. Soubor obsahuje mnozinu dvojic jmeno a unikatni ciselne oznaceni.

### Jmeno souboru:

- drive limit 14 znaku, dnes az 255 (na konci musi byt '`\0`')
- ve jmene nesmi byt / nebo '`\0`'
- lati ze kazdy adresar v POSIX systemu vzdy obsahuje minimalne 2 jmena: . (odkaz na sebe) a .. (odkaz na rodicovsky adresar)

### Cislo souboru:

- u klasickeho souboru unixu je to cislo i-uzlu,
- v jinych pripadech to slouzi jako klic do dane vyhledavaci struktury (B+ strom)

### Implementace adresaru:

- pouzivaji se ruzne pristupy, lisi se jednoduchosti implementace ci rychlosti vyhledavani (vkladani),
- seznam (obsah souboru bude tvorena seznamem),
- B+ stromy (v NTFS, XFS, JFS, APFS nebo EXT3/4 - ty pouzivaji H-stromy: 1-2 urovne, bez vyvazovani a vyhledava se na zaklade zahashovaneho jmena)
- hashovaci tabulky v napr. ZFS

Soubor v UNIXu muze mit vice jmen. Dalsi jmena se vytvari pomoci prikazu `ln`

### linux:

`ln [existujici jmeno] [nove jmeno]` vytvori dalsi jmeno souboru

## 6.10 Montovani disku

### Princip motnovani disku:

- v UNIXu neni zadne oznaceni disku (A:, C:, ..), ale mame jeden adresarovy "strom",
- v systemu je jeden korenovy logicky disk,
- dalsi logicke disky se pripojuji programem *mount* do existujiciho adresaroveho stromu (korenovy adresar zarizeni se "slepi"s adresarem v mem stromu)

Pripojovací volby se mohou zadavat rucne (v terminalu) nebo se mohou predpripravit do */etc/fstab*. Soubor */etc/mtab* obsahuje tabulku aktualne pripojenych disku.

### Novejsi technologie umoznuji automaticke montovani nove pripojenych zarizeni:

- na linuxu bezne pracuje system *udev*, který
- rozpozna, ze se pripojilo nove zarizeni,
- vytvori odpovidajici soubor typu blokove zarizeni (*/dev/..*),
- informuje o tom zbytek systemu pomoci sbernice D-Bus,
- aplikace typu spravce souboru pak muze provest automaticke montovani (a dalsi akce),
- prednost ma vzdy */etc/fstab*,
- identifikace se nemusi provadet jen zarizenim (*/dev/..*), je mozne si vygenerovat unikatni identifikator a pouzivat ten (UUID)

### Technologie Automounter:

- subsystem jadra,
- pripojuje automaticky potrebne disky v situaci, kdy se pokusime pristoupit na pozici adresaroveho stromu, kam by takovyto disk mel byt pripojeny (napr. na */mnt* ma byt pripojena flashka, nemusi byt *mountla*, uzivatel da *cd /mnt*, *automounter* to zjisti a pripoji flashku sem),
- ma take nejaky cas, po kterem disk automaticky odpoji, pokud se nim nepracuje

### Union mount:

- technologie umoznujici sjednocujici montovani (v unixu dostupna pomoci filesystemu *UnionFS*),
- umoznuje do jednoho pripojneho bodu namontovat vice disku,
- obsah pripojneho bodu je sjednocenim obsahu disku,
- v pripade, ze na vice discich jsou soubory se stejnými jmeny vznikaji kolize, ty se resi napr. preddefinovanim priorit pripojovanych filesystemu a zprístupni se soubor daneho jmena z logickeho disku, který ma nejvetsi prioritu
- *UnionFS* ma copy-on-write semantiku, což umoznuje emulaci prepisovani neprepisovatelných medií (v 1 vetvi CD, neprepisovatelné, na tom je linuxova distribuce, soucasne se do stejneho bodu pripoji bezny disk s vyssi prioritou - na zacatku bude disk prazdny, budou videt vsechny soubory z CD, jakmile se

pokusim prepsat neco, UnionFS vytvori kopii na prepisovatelny disk)

**linux:**

*mount [co-pripojit] [kam-pripojit]* pripoji logicky disk

**Sedma prednaska:** Pokracovani Spravy souboru. Symbolicke odkazy,

### 7.1 Symbolicke odkazy

- je samostatny soubor odkazujici na existujici soubor,
- system pri otevreni automaticky otevre cilovy souboru - vicenasobne zpracovani cesty (ceska k symlinku a cesta uvnitr nej),
- soubor se smaze, pokud jeho pocet jmen klesne na 0,
- symlink muze odkazovat na neexistujici soubor (pri otevreni dojde k chybe),
- muze odkazovat i na jiny logicky disk,
- lze ze symlinku vytvorit cyklus (jeden odkazuje na druhy a druhy na prvni) - v systemu je preddefinovany maximalni pocet na sebe odkazujicich symlinku (pri prekrojeni dojde k chybe),
- symlinky lze vyuzit napr. pri upgradu systemu.

#### Rozdil rychlych/pomalych symlinku:

- obsahem symlinku je jmeno ciloveho souboru,
- pokud jmeno souboru neni prilis dlouhe (vleze se do i-uzlu), potom se ulozi do i-uzlu = rychly symlink (staci otevrit jen i-uzel),
- pokud se jmeno nezleze do (i-uzlu), alokuji se normalne alokacni bloky na disku = pomaly symlink

#### linux:

*ln -s [existujici soubor] [symbolicky odkaz]* vytvori symbolicky odkaz

## 7.2 Blokové a znakové speciální soubory

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

Typický zařízení sídlí v adresáři /dev

### Beze typy zařízení:

- /dev/hda - (drive)označení pro první fyzický disk na prvním ATA/PATA rozhraní
- /dev/hda1 - (drive) 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 0 bajtů
- /dev/null - soubor typu černá díra - cokoli se do něj zapisuje, do se zahodí (přesměrování výstupu programu tak, aby nás neotravovalo), při čtení se chová jako prázdný soubor
- /dev/random - generátor náhodných čísel
- /dev/tty - terminál
- /dev/lp0 - tiskárna
- /dev/mouse - myš
- /dev/dsp - zvuková karta
- /dev/loop - zařízení typu (smýčka) loop mi umožňuje připojit soubor jako disk (obraz souborového systému) k adresáři, jakoby 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 chceme pracovat.

### 7.3 Pristupova prava

V UNIXu jsou typicky rozlisena na prava pro vlastnika, skupinu vlastniku a ostatni. Existuje rozsireni ACL (access control list).

#### **Uzivatele:**

- jsou definovani administratorem systemu (root) v /etc/passwd,
- maji definovana sva UID - uzivatelska cisla (root UID = 0),
- kazdy soubor ma sveho vlastnika,
- chown - zmena vlastnika souboru (pouze root),

#### **Skupiny:**

- definuje administrator systemu v /etc/group,
- maji sva GID - cislo identifikujici skupinu uzivatele,
- v kazde skupine je uvedeno, kdo do te skupiny patri,
- kazdy uzivatel muze byt clenem vice skupin,
- jedna z nich je aktualni (pouziva se pri vytvoreni souboru)

#### **linux:**

*groups* - vypis skupin uzivatele

*chgrp* - zmena skupiny souboru

*newgrp* - novy shell s jinym aktualnim GID

## 7.4 Typy pristupovych prav

**Obycejne soubory:** r, w, x - pravo cist, zapisovat a spustit soubor jako program.

**Adresare:**

- r - pravo cist obsah adresare,
- w - pravo zapisovat (vytvaret a rusit soubory),
- x - pravo pristupovat k souborum v adresari (moznost cd adresar, ls -l adresar, ..)

**Typicky vystup pristupovych prav je:**

- ve formatu: [1:typ souboru] [3:prava vlastnika] [3:prava skupiny] [3:prava ostatnich]
- napr.: -rwx—r- (ciselne vyjadreni v 8 soustave 0704)
- je obycejny soubor, vlastnik ma vsechna prava, skupina zadna a ostatni maji prava na cteni

Zmena pristupovych prava se deje pomoci chmod. (Pro nespustitelne soubory je bezny chmod 0644).

**linux:** `chmod [1:pro koho][nova prava] [soubor/y]` zmena pristupovych prav

## 7.5 Sticky bit

Priznak, který pokud bude prirazen nejakemu adresari, tak se vytvori adresar, ve kterem i pres pravo cteni a zapisu souboru mohou uzivatele rusit pouze ty soubory, které sami vytvorili. (typicky adresar /tmp) (tldr: uzivatel muze mazat, ale pouze to co vlastní)

## 7.6 SUID, SGID

**S procesy jsou spojeny identifikatory jako jsou:**

- UID - realna identifikace uzivatele (cislo uzivatele, který dany proces spustil)
- EUID - efektivni identifikator pouzivany pro kontrolu pristupovych prav (vetsinou stejne jako UID)
- GID - realna identifikace skupiny (kdo spustil proces)
- EGID - efektivni GID, (stejne chovani jako u EUID)

Vlastnik programu muze propujcit sva prava komukoliv, kdo spusti program s nastavenym SUID. (tldr: propujci se prava mezi uzivateli, bezne se to pouziva v pripadech, kdy administrator propujcuje sva prava uzivatelum, napr. passwd)

Pri pouziti SUID bude UID = uzivatele, který proces spustil a EUID = identifikace vlastnika (= který prava pujcil). Pokud budou prava propujcena (pouzito SUID), misto x se vypise s, pokud tam x neni, vypise se S.

## 7.7 Typická struktura adresaru v UNIXu

### FHS - Filesystem Hierarchy Standard (Linux), (cast hierarchie):

- /bin - programy pro vsechny uzivatele (spustitelne, mohou byt zapotrebi pri bootovani - musi byt dostupne lokalne)
- /boot - soubory pro zavadeč systému (obrazy jadra, počáteční fs)
- /dev - speciální soubory - rozhraní zařízení
- /etc - konfigurační soubory pro systém i aplikace
- /home - domovské adresáře uživatelů
- /lib - sdílné knihovny a moduly jadra
- /media - připojny bod pro přenosná zařízení
- /mnt - připojny bod pro dočasné filesystemy
- /proc - informace o procesech a jádru
- /root - domovský adresář superuživatele
- /run - dočasné informace o běžícím systému (demony)
- /sbin - programy pro superuživatele (nutné pro bootování, ne vše je spustitelné superuživatelem)
- /sys - informace o jádru, zařízeních, modulech, ..
- /tmp - dočasné pracovní soubory (obsah se mže 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 nezávisle na architektuře),
  - local (koreň další hierarchie určena pro lokální nestandardní instalace programu),
  - src (zdrojové texty jadra)
- /var - soubory menící se za běhu systému
  - obsahuje log (záznamy o činnosti systému),
  - spool (pomocné soubory pro tisk),
  - mail (poštovní příhradky uživatelů)



**Nove tema: Datove struktury a algoritmy pro vstup/vystup**

## **7.8 Pouziti vyrovnacich pameti**

Cilem pouziti cache (vyrovnacich pameti) je minimalizace poctu pomalych operaci s periferiemi (disky). Hierarchie: kolekce, sbirka dilcich vyrovnacich pameti (s velikosti 1 alokacniho bloku, ci nasobku), nazyva se buffer-pool, muze mit pevnou velikost, spise je promenna.

## 7.9 Operace se soubory

### 7.9.1 Cteni

#### Prvni cteni alokacniho bloku:

- zjistí se, zda je blok v pameti,
- pokud ne, naalokuje se nový blok, může se využít již nějaký systémem predalokovaný a nevyužitý,
- nactou se data z disku, přesunou se do vyrovnávací pameti,
- vyrovnávací pamet je v prostoru jádra (bezne procesy zde nemají přístup),
- vykousne se z nactených dat tu část, o kterou má uživatel/proces zájem,
- nakopíruje se to do adresového prostoru uživatelského prostoru

#### Pri dalsim cteni:

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

### 7.9.2 Zapis

#### Postup pri zapisu:

- nejprve se zjistí, zda je blok v pameti,
- pokud ne, přideli se vyrovnávací pamet,
- nactou se data z disku do vyrovnávací pameti,
- jádro převezme od procesu, který chce zapisovat data, která chce zapsat,
- prepíše jimi danou část alokacního 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

System sám od sebe s periodou prepisuje cache na disky, lze si to vynutit pomocí sync či fsync.

Pokud je známo, že se prepíše celý alokacní blok (nebo se jedná o nový blok), buffer se vynuluje a nenacitají 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 = disk, 1 - data cache != data disk - nuluje se zpožděným zápisem)

### 7.9.3 Otevreni souboru pro cteni

Pokud soubor jeste **nebyl otevren**:

- system musi vyhodnotit cestu a naleznou cislo i-uzlu (resp. cislo datove struktury poskytujici informace o danem souboru - pristupova prava, kde jsou ulozena data),
  - pri tom se postupne nacistaji i-uzly vseh adresaru vedouci na soubor,
  - pote se nacte i-uzel souboru,
  - system pouziva d-entry cache (specialni vyrovnavaci pamet pouzita pro preklad odpovidajicich jmen souboru na i-uzel)
  - dale alokuje polozku v tabulce V-uzlu,
  - z disku se nacte i-uzel,
  - vlozi se do nove alokovane polozky = vznikla rozsirena pametova kopie i-uzlu,
  - budou tam i informace navic (jako je pocet odkazu na danou polozku - danym i-uzlem muze pracovat vice procesu).
- v tabulce popisovacu vytvorime novou polozku,
  - tato tabulka je ulozena v zaznamu o procesu (tabulka procesu v jadre) nebo v uzivatelske oblasti,
  - pouzije se nejnizsi volna polozka zde,
  - naplni se odkazem na polozku v tabulce otevrenych souboru,
- pokud se otevreni vydari, vrati se cislo popisovace, pokud ne, tak vraci -1.

Tolik tabulek se pouziva pro zamezeni duplikaci udaju. Behem otevirani se provadi kontrola pristupovych prav. Soubor je mozne otevrit v rezimu pro cteni, zapis, cteni i zapis.

Dalsi otevreni souboru (**jiz jednou otevreny**):

- opet se vyhodnoti cesta k souboru a ziska se cislo i-uzlu,
- system se podiva do tabulku V-uzlu,
- zjistí, ze i-uzel uz tam je,
- nebude se znovu i-uzel nacistat z disku, pouze se zvysi citac pouziti i-uzlu,
- tabulka V-uzlu musi byt vyhledavaci (typicky vyhledaci struktury jako hash tabulka, strom, ..),
- naalokuje se nova polozka v tabulce otevreni (naplni se rezimem otevreni, pozici, odkazem na sdileny V-uzel),
- naalokuje se nove poloza ve file descriptoru ukazujici na nove otevreni (a ta se vrati)

### Je možné pridávat i další identifikatory, napr.:

- příznak, že má být soubor vytvořen pokud neexistuje,
- pokud existuje, má být zkrácen na 0,
- otevřít v režimu přidávání (kdekoli je aktuálně ukazovátka 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 zapisou opravdu na disk)

### Při chybě:

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

### linux:

`fd = open([jméno souboru], [režim]);` otevře soubor

`V-uzly` je tabulka i-uzlů filesystemu VFS

`tabulka popisovacu` je pole s řádky číslovanými od 0 (0 - `stdin`, 1 - `stdout`, 2 - `stderr`)

`tabulka procesu` v jádře je část adresového prostoru, ve kterém má jádro uloženy pomocné informace k procesu a má sem přístup pouze jádro

## 7.9.4 Čtení a zápis z/do souboru

### Čtení:

- zkontroluje se platnost popisovace (otevření popisovace, 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 požadovaných z cache (RAM, jádro) do pole (RAM, cache adresového prostoru procesu),
- funkce vrátí 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),
- vrátí počet opravdu zapsaných bajtů nebo -1

**linux:**

*read([popisovac], [adresa pameti, kam se ma zapsat], [kolik bajtu se ma nacist])* precte soubor  
*write([popisovac], [adresa pameti, ze ktere se nactou data], [kolik bajtu se zapise])* zapis do souboru

### 7.9.5 Primy pristup k souboru

Nahodne presouvani v souboru. **Postup:**

- zkontroluje zda je popisovac platny (je soubor otevren?)
- nastavi pozici offset bajtu od whence
  - SEEK\_SET - napr. 200 - posunu se od 200 bajtu od zacatku,
  - SEEK\_CUR - od aktualni pozice,
  - SEEK\_END - od konce souboru),
- nelze se posunout pred zacatek souboru,
- je ale mozne se posunout za konec souboru (a zapsat),
- vraci se vysledna pozice od zacatku souboru nebo -1

Posunem za konec souboru a naslednym zapisem vznikaji tzv. ridke soubory (sparse files):

- umoznuje na disku o nejake kapacite vytvorit soubor, který ma zdanlive vetsi velikost nez samotny disk,
- bloky do kterych se nezapisovalo nejsou alokovany a nezabiraji diskovy prostor (pri cteni se povazuji za 0),
- take muze vzniknout mazanim uprostred souboru (hole punching)



Obrázek 3: z prezentace IOS: Sprava souboru - ridke soubory

**linux:**

*lseek([popisovac souboru], [offset], [oproti cemu se chci posouvat])* primy pristup k souboru

### 7.9.6 Zavreni souboru

- zkontroluje se platnost file descriptoru (je vubec otevreny?),
- uvolni se dana polozka v tabulce popisovacu,
- system se podiva na odkazovanou polozku v tabulce otevrenych souboru,
- snizi se pocitadlo o 1,
- pokud bude pocitalo != 0, uzavirani skonci,
- pokud bude pocitadlo == 0, pokracuje se do
- prislusne polozky tabulky V-uzlu, snizi se zde pocitadlo o 1,
- pokud bude zde pocitadlo != 0, uzavreni skonci,
- pokud bude zde == 0, soubor se definitivne uzarve,
- uvolni se z pameti i-uzel z tabulky V-uzlu (se zmenenymi udaji - cas zapisu, pristupu, modifikace i-uzlu, ..),
- naplanuje se blok, ve kterem je i-uzel ulozen,
- casem se i-uzel zapise na disk,
- funkce vraci 0 nebo -1 pri chybe

Pokud se proces skonci, automaticky se zavrou vsechny jeho deskriptory. Uzavreni souboru nezpůsobí uložení obsahu jeho vyrovnávací paměti na disk.

#### linux:

*close([popisovac souboru])* zavře soubor

### 7.9.7 Duplikace deskriptoru souboru

- zkontroluje se platnost deskriptoru (je soubor otevren?),
- zkopiruje obsah puvodniho popisovace do noveho (odkaz ve fd tabulce se zkopiruje do dalsi polozky v teto tabulce - oboje ukazuji na stejnou polozku tabulky otevrenych souboru + inkrementuje se pocitadlo),
- automaticky se novy deskriptor uzavre (pokud je otevren),
- vraci index nove vytvorene polozky nebo -1,
- typicke pouziti je u presmerovani (stdin/stdout)

#### linux:

*dup([popisovac])* duplikace deskriptoru (duplikuje existujici popisovac do nejvyssiho volneho noveho)

*dup2([popisovac], [novy popisovac])* duplikace deskriptoru (do ktereho popisovace se duplikuje)

### 7.9.8 Ruseni souboru

- vyhodnoti se cesta, zkontroluje se platnost jmena souboru, pristupova prava (zapis),
- odstrani se pevny odkaz (=hard link) mezi jmenem souboru a i-uzlem,
- zmensi se pocet jmen v i-uzlu,
- pokud je pocet jmen == 0 a i-uzel nikdo nepouziva - i-uzel muze byt uvolnen a mohou byt uvolneny vsechny bloky souboru,
- dokud ma soubor alespon 1 jmeno nebo nema zadne jmeno ale je otevren alespon 1x, nelze soubor z disku opravdu smazat,
- funkce vraci 0 nebo -1 pri chybe

Je mozne provest unlink na otevreny soubor (smaze se az po jeho uzavreni) a pracovat s nim dale, vyuziti pri instalacich novych verzich programu, ktere aktualne bezi. (upgradovat upgradovaci program)

#### **linux:**

*unlink([jmeno soubor, prip. cesta])* rusi soubor  
*shred* - bezpecne mazani

### 7.9.9 Dalsi operace se soubory

#### **linux:**

*creat, open* - vytvoreni souboru  
*rename* - prejmenovani souboru  
*truncate, ftruncate* - zkraceni souboru  
*fcntl, lock* - zamykani zaznamu  
*chmod, chown* - zmena atributu  
*utime* - umoznuje zmenit casy prace se soubory (neumoznuje zmenit cas modifikace i-uzlu)  
*stat* - ziskani atributu (velikost, prava, ..)  
*sync, fsync* - vynuceni si zapisu vyrovnacich pameti

### 7.9.10 Adresarove soubory

Obsahuje dvojice cislo i-uzlu a jmena souboru. Adresare nelze zapisovat ci cist po bajtech.

#### **linux:**

*mkdir* - tvori se adresare (vytvori polozky . a ..)  
*opendir* - otevre adresar  
*readdir* - cte adresar  
*closedir* - zavre adresar  
*creat, link, unlink* - modifikace se provadi nepriamo vytvarenim/modifikacemi souboru

### 7.9.11 Blokové a znakové speciální soubory

Představují rozhraní k blokovým / 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 tyto 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 údaje**:

- 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

*ovladac* je sada podprogramů pro řízení určitého typu zařízení (nebo viz `xx` nebo viz `xx`)



## 7.10 Terminaly

Jsou fyzická či logická zařízení umožňující (primárně) textový vstup a výstup systému (po radcích), editace vstupního řádku či speciální znaky (Ctrl+C SIGINT, Ctrl-D konec vstupu, ..)

### Rozhraní:

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

### Různé režimy zpracování znaku (radkové disciplíny - line discipline):

- raw - neprovádí se zpracování znaku,
- cooked - zpracování všech řídicích znaků,
- cbreak - provádí zpracování maleho počtu znaku (ctrl+c, mazání, ..)

Nastavení režimu zpracování znaku je možné pomocí stty. Dale je možné nastavit **režim terminálu**:

- příkazy tset, tput, reset,..
- proměnnou TERM, ve které uložen aktuální typ terminálu,
- typy terminálu (příkazy terminfo, termcap)

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

### definice:

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

## 7.11 Roury

Jsou prostředkem meziprocsove komunikace. Rozlisujeme:

### Nepojmenovane roury

- nemaji adresarovou polozku, tedy neexistuji v souborovem systemu,
- lze s nimi pracovat pouze tak, ze se vytvori pomoci volani pipe (vrati cteci a zapisovy deskriptor), jakmile dojde k uzavreni - prace s rourou konci,
- mohou s ni pracovat bezne pouze pribuzne procesy,
- je dostupna pomoci popisovacu z tabulky popisovaci (pri klonu procesu se naklonuje tabulka popisovaci - proces bude ukazovat na stejne misto v tabulce otevrenych souboru),
- jedina vyjimka, jak je mozne odkaz na nepojmenovanou rouru predat je pres UNIXove sockety (krome klonovani procesu),
- vytvari se v kolonach (napr. paralelne bezici procesy  $p1 \rightarrow p2 \rightarrow p3$  - na presmerovani se pouzivaji nepojmenovane roury)

### Pojmenovane roury

- vytvari se pomoci `mknod` i s `mknfio`,
- existuji v souborovem systemu,
- mohou se zavrit, otevrit, apod.

Roury slouzi jako mechanismus meziprocsove komunikace. Implementovane jako kruhovy buffer s omezenou kapacitou. Procesy komunikujici pres rouru jsou synchronizovany.

#### definice:

*pribuzne procesy* - pokud jeden proces otevri rouru a zacne se klonovat, vsechny tyto procesy mohou s rourou pracovat

*konzumenti* - procesy, ktere ctou

*producenti* - procesy, ktere zapisuji

## 8

**Osma prednaska:** Dokonceni souborovych systemu: Roury, sockety. Procesy:

### 8.1 Sockety

Umoznuji jak sitovou (klient-server, TCP, UDP) tak lokalni (filesystem) komunikaci.

Pro vytvoreni socketu se pouziva volani socket:

- nasledne se ceka na pripojeni (bind - propojit socket s TCP/UDP portem ci souborem, listen - zacnam cekat, accept - prijem prichoziho spojeni),
- klient se pripoji pomoci (connect),
- prijem a vysilani zprav (recv / send ci read / write - volani vraci popisovace otevrenych souboru),
- uzarevni (close)

Sockety podporuji blokujici i neblokujici I/O. Pri praci s vice sockety je mozne je obsluhovat vice procesy - vlakny (prikaz select) - typy souboru, u kterych muze nastat potreba cekat na moznost provedeni urcite operace. Sockety taky maji vyhodu, ze je mozne vytvorit aplikace, které mohou bezet distribuovane v siti.

#### definice:

*blokujici rezim (I/O)* - pokud chci nacist data ze socketu, budu pozastaven, dokud se nejaka data neobjevi  
*select* umoznuje testovat, zda na popisovaci je dostupna nejaka operace (ci mnozine popisovacu)  
*pasivni cekani* - nespotrebovava se CPU cas, energie, ..

### 8.2 VFS

Virtual File System. Definice viz. 5.11.5. Komunikace s ruznymi filesystemy se prenasi z uzivatele na autora fs, který pokud chce, aby dany fs byl vyuzitelny, musi ho provazat s VFS (propojeni VFS a uzivatele resi vyvojari).

Typicka datova struktura VFS jsou **V-Uzly** = rozsirene pametove kopie i-uzlu, které krome dat i-uzlu obsahují dalsi data:

- jako pocet odkazu na v-uzel z tabulky otevrenych souboru,
- ukazatele na funkce implementujici operace nad i-uzlem (v patricnem filesystemu).

### 8.3 NFS

Network File System. Zprístupňuje súbory uložené na vzdialených systémoch.

Jedna se o **system klient-server**:

- klient požada o čtení ze souboru (napr.),
- 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 filesystemem již na vzdáleném PC,
- ten pracuje také s VFS (ale na serveru), prostřednictvím něj získá data s lokálního filesystemu,
- data poté putují zpět přes síť k uživateli zpět

Umožňuje **kaskadování** - je možné si lokálně do jednoho adresářového stromu připojit vzdálený strom (a do něj připojit další vzdálený filesystem). **Autentizujeme se nejčastěji přes UID, GID** (musí existovat důvera mezi správcem lokálního a vzdáleného systému). Nebo se používá jiné mechanismy (kryptografie, ..)

NFS verze 3:

- starsi, 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 zamky (operace pro zamknutí záznamu souboru jsou prázdné)

NFS verze 4:

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

### 8.4 Spooling

Simultaneous peripheral operation on-line (simultánní online provádění periferních operací). Jedná se o **provádění online bez čekání periferní operace (výstup) na perifériích, které nemusí online prokládání dat od různých procesů či uživatelů podporovat**. (např. 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, operací, do fronty se zaradi odkaz na vytvořený soubor, úloha se má dokončit po uvolnění periférie.

**linux:**

*/var/spool* obsahuje soubory spool

**Nove tema:** Sprava procesu.

Sprava procesu (process management) zahrnuje:

- prepínání kontextu (dispatcher, vždy v režimu jádra),
- plánovač (nemusí být v jádře),
- správu paměti,
- podporu meziprocесовé komunikace (signály, roury, sockety, synchronizace - semaforey, mutexy, ..)

**definice:**

*prepínání kontextu* rozumíme fyzicky odebrání procesoru jednomu procesu a přidělování jmenu procesu (take viz 1.3)

*plánovač* rozhoduje, který proces či procesy pobeží 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:**

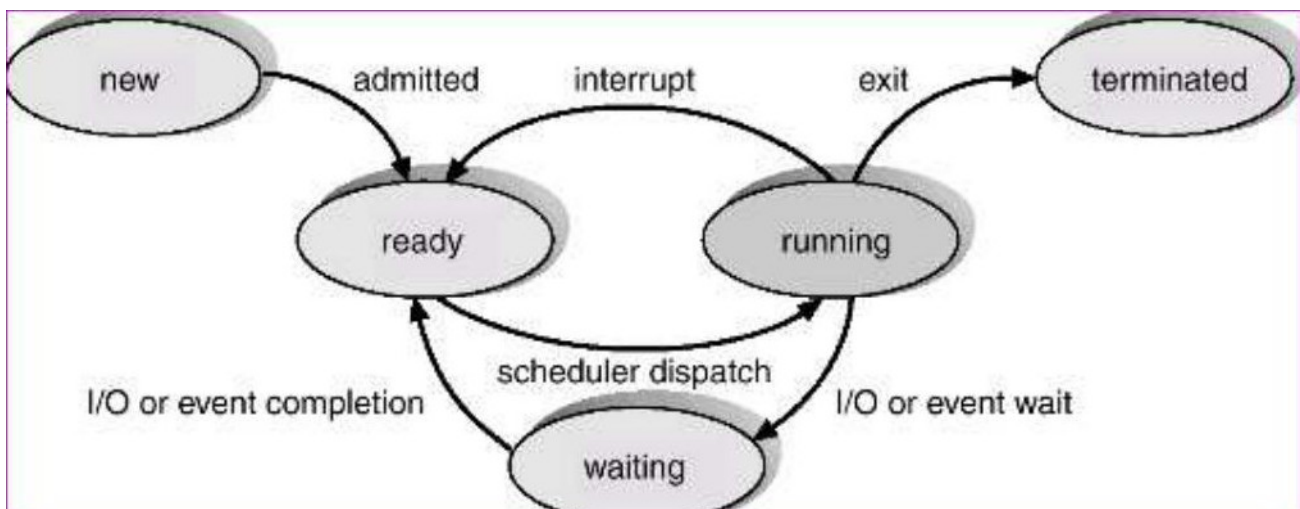
- unikátním identifikátorem (PID - process identifier),
- stavem plánování,
- řídicím programem,
- obsahem registru (bežných - EAX, BX, EIP, ..),
- zásobníkem (aktivací záznamy - informace o rozpracovaných funkcích),
- daty (statická ne/inicializovaná data, hromady, individuálně alokované paměti),
- tím, jaké další vazby a zdroje OS využívá (jaké soubory má aktuálně otevřené, signály, služební funkce signálu, PPID, UID, GID, semaforey, sdílená paměť, sdílené knihovny, ..)

## 8.6 Stavy planování a jejich změny

Nejzákladnější plánovací diagram (většiny/všech různých OS) - **stavy procesu**.

**Stavy planování procesu (obecně):**

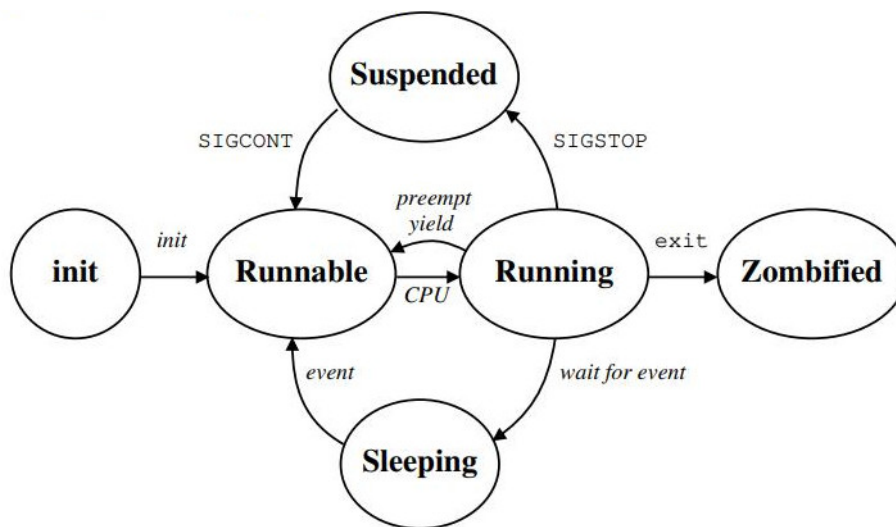
- new - proces je inicializován (vytváří se struktury co jej popisují, data, proces případně čeká ve vstupní frontě dlouhodobého plánovače, ..)
- ready - proces čeká na krátkodobý plánovač na přidělení procesoru, dispatcher provede přepnutí, přepne se do running
- running - proces může být přerušen (preemptivní plánování - opět stav ready), může požadovat službu jádra (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 planovani procesu v UNIXu:

- init - vytvoreny, neinicializovany,
- runnable - pripraven bezet,
- running - proces bezi (pridelen procesor), v pripade preempce (proces se vzda CPU) - jde do runnable,
- sleeping - proces pozada o I/O ci sync akci (ceka na dokonce operace), po realizaci bude runnable
- suspended - pomoci signalu SIGSTOP muze byt proces zmrazen a ceka ceka na rozmrazeni (signal SIGCONT),
- zombified - ukonceni procesu a prechod do stavu zombie (matoha), proces skoncil, odebrany vsechny zdroje, pouze o nem zustava zaznam v tabulce procesu (zde je jeho navratovy kod dokud si ho nekdo neprevezme)



Obrázek 5: z prezentace IOS: Sprava procesu - stavovy diagram planovani procesu v UNIXu

OS s procesem pracuje tak, ze je reprezentovan pomoci struktury PCB (process control block), nekdy take task control block ci task struct.

### PCB zahrnuje (primo / formou odkazu):

- identifikatory spojene s procesem,
- stav planovani,
- obsah registru (v okamziku kdy je pozastaven),
- planovaci informace (priorita, ukazatele na planovaci fronty, ..),
- informace spojene se spravou pameti (tabulky stranek pri pouziti strankovaci pameti),
- informace spojene s uctovanim (sumarizuje informace o behu procesu - spotreba CPU, ..),
- informace o vyuziti I/O zdroju (otevrene soubory, tabulka popisovacu, ..)

PCB muze byt bud jedna struktura nebo muze byt rozdeleno na nekolik casti.

## 8.7 Casti procesu v pameti v UNIXu

První součást paměti využívané procesem je **uživatelský adresový prostor** (user address space) - je přístupný procesu (může z této části paměti číst/pisat), obsahuje:

- kód (který je řízen, code area/text segment),
- data (ne/inicializována, hromada, alokována 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 procesu):

- uložena pro každý proces v části uživatelského adresového prostoru, která není přístupná procesu,
- 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 registru,
- deskriptory souborů (informace o tom, které otevření souborů reprezentuje stdin, stdout),
- obslužné funkce signálů (funkce, které se budou volat pro obsluhu signálů),
- uctování,
- pracovní, kořenový adresář

**Další záznamy jsou v tabulce procesu:**

- uloženo trvale v jádru,
- informace o procesu, které jsou důležité i když proces nebeží:
- PID, PPID, UID, EID, ...,
- stav plánování,
- událost, na kterou proces čeká,
- plánovací informace (pro plánovač při rozhodování který proces dál pobeží či ne - priorita, spotřeba času, ..),
- čekající signály (signály, které mohou přijít, i když proces nebeží),
- odkaz na tabulky dat reprezentující rozložení procesu, dat, kódu, zásobníku, ..



Pote jsou jeste zaznamy v **tabulce pametovych regionu**:

- jak je rozdelen uzivatelsky adresovy prostor na regiony (= souvisly kus pameti pouzity za urcity region, kod, zasobnik, ..),
- velikost techto regionu,
- globalni tabulka regionu (odkaz z teto na lokalni tabulky regionu),
- regiony byvaji cleneny na stranky (tabulky stranek)

**definice:**

*logicky adresovy prostor* - rozsah vseh logickych adres, ktere se mapuji do fyzicke pameti

*zasobnik jadra* je separatni zasobnik nekdy pouzivan pro ukladani rozpracovanych funkci jadro v okamziku kdy jadro provadi sluzbu pro dany proces

## **8.8 Kontext procesu**

Je jine oznaceni pro **stav procesu**. Rozlisujeme:

- uzivatelsky kontext - cast stavu procesu popisujici cast pameti dostupnou procesu samotnemu (kod, zasobnik, data),
- registrovy kontext,
- systemovy kontext - cast stavu procesu nedostupna samotnemu procesu (uzivatelska oblast, polozky tabulky procesu, pametove regiony,..)

## 8.9 Systemova volani nad procesy UNIXu

= standardni POSIXova volani:

- fork, exec, exit, wait, waitpid,
- kill, signal - synchronizace,
- getpid, getppid - ziskavani identifikatoru, ...

### Identifikatory spojene s procesy v UNIXu:

- identifikace procesu PID (vlastni identifikator procesu),
- identifikace predka PPID (proces, který dany proces vytvoril, rodic),
- realny-skutecny uzivatel, skupina uzivatelu, který proces spustil - GID, GID,
- efektivni uzivatel ci skupina - EUID, EGID (viz. SUID, SGID v x.x),
- ulozene EUID a EGID - procesu umoznuji docasne se zbavit vysokych prav, které získal (proces se dobrovolne vzda vyssich prav - ulozi se a pobezi s beznymi pravi - v okamziku provadeni kritickych operaci si prava zase navysi – ochrana pred chybami v programech),
- v linuxu FSUID, FSGID (file system UID/GID - oddelena zvysena privilegia pro praci s filesystemu),
- PGID, SID (process group identifier, session identifier - skupina procesu ci sezeni, do kterych proces patri)

### definice:

*sezeni* je skupina skupin procesu vytvarejici se typicky pri praci s terminaly

## 8.10 Vytvareni procesu

Procesy v UNIXu vznikaji **volanim sluzby fork**. Fork je volani, které se **zavola jednou**, pokud nedojde k chybe, tak **skonci 2x**. Na zaklade volani fork vzniká **vztah rodic-potomek (parent-child) a hierarchie procesu**. Vysledkem forku je totiz **duplikace procesu**. Vznika takrka identicka kopie potomka, který dedi:

- ridici kod, data, zasobnik, sdilenou pamet, otevrene soubory, obsluhu signalu, vetsinu synchronizacnich prostredku, ..
- pro efektivitu pouziva pro praci s pameti copy-on-write
- kopie se lisi v navratovem kodu fork, identifikatorech, udajich spojenych s planovanim a uctovanim, nededi se cekajici signaly, souborove zamky a nektare dalsi zdroje ci nastaveni ..

### Navratove kody forku:

- 0 - fork se zdaril, if(pid==0) kod pro potomka
- -1 - fork se nezdaril, if(pid==-1) kod rodice
- cokoli jineho (PID potomka) - else kod pro rodice

## 8.11 Hierarchie procesu v UNIXu

Prvním procesem, který vzniká a vytváří ho jádro je **proces init s PID=1** (aktuální novější implementace je pojmenována jako systemd - příkaz pstree). Tento proces je **predkem všech ostatních uživatelských procesu. PPID initu je 0.**

Existují také procesy jádra (kernel threads/processes), jejich init predkem 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 či je to používáno jako procesová obálka pro vlákna jádra (na Linuxu se tento proces nevypisuje)

Init se podílí na **inicializaci systému**, poté **prebírá návratové kódy procesu, které skončí, ale které osířely driv nez skončily** (tedy rodič skončí driv nez potomek) - teoreticky by byl zombie procesem do nekonečna, proto init prebírá jeho návratový kód a umožní mu odchod ze systému.

### definice:

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

*kthread* je proces, na Linuxu init pro procesy jádra, má PID=2

### linux:

*ps tree* vypisí stromu procesu

## 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:

- poradí dědičnou řadu rysů svého předka,
- zůstává mu rada zdrojů a vazeb OS (identifikatory, otevřené soubory, ..),
- zanikají vazby a zdroje vázané na původní řidici (obslužné funkce signálu, sdílená paměť, paměťové mapované soubory, semaforey).

Skupina funkcí exec:

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

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

### 8.13 Cekani na potomka - wait, waitpid

Slouzi k tomu, aby mohl **rodic cekat (pasivne) na dokonceni cinnosti svymi potomky**.

Volani wait:

- ceka na ukonceni 1 z potomku,
- vraci cislo potomku, který skoncil (pripadne -1 pokud prijde signal, který cekani prerusi nebo pokud cekame na potomka a zadneho nemame),
- muze to byt operace blokujici, pokud zadny z potomku jeste neskoncil,
- pokud nektery potomek skoncil driv pred volanim wait, okamzite volani wait skonci a vrati se navratovy kod.

Volani waitpid:

- umoznuje cekani na ukonceni urciteho potomka urcite skupiny dle PID
- umoznuje cekani i na pozastaveni ci probuzeni (SIGSTOP, SIGCON).

### 8.14 Start systemu

- (nejprve) dostane se ke slovu firmware PC (UEFI/BIOS),
- nacteni a spusteni zavadece OS (nekdy se zavadecu pouziva nekolik, napr. BIOS vyuzival zakladni kod MBR - serie zavadecu),
- nactou se inicializacni funkce jadra a samotneho jadro, spusteni inicializacnich funkci,
- inicializacni funkce jadra vytvori proces 0, dalsi procesy jadra a proces init,
- proces init pokracuje v inicializaci systemy, spustu demony a procesy,
- v urcitem okamziku se z nej spusti procesy umoznujici prihlaseni v GUI (GDM, SDDM, LightDDM) bud z nich nebo s nim spolupracujici procesu se spousti procesy pro praci s X Windows,
- na konzolich se spusti getty (ctrl+alt+f1,f2,..) - umozni uzivateli zadat prihlasovaci jmeno, zmeni se na login, nacte od uzivatele heslo, pote se zmeni na shell, ze ktereho se spousti dalsi procesy, po ukonceni se opet spousti getty,
- proces init i po inicializaci nadale bezi, prebira navratove kody procesu, jejich rodic skoncil driv nez prislusny proces, take resi reinicializaci systemu (na prani uzivatele ci vypadek napajeni)

#### **definice:**

*firmware* je program ulozeny v nevolatilnich pametech, provadejici kontrolu hardware, pripadnou inicializaci hw

## 8.15 Urovne behu

System urovni behu byl zaveden jiz v UNIX System V. Rozlisuji se urovne behu 0-6: (nektare mely predpripraveny standardni vyznam, nektare si definoval administrator)

- 0=halt - zastaveni systemu,
- 1=single user - jednouzivatelsky rezim, pouziva spravce systemu,
- 3-6=definovane rezimy adminem,
- 6=reboot - automaticke restartovani,
- s/S=jednouzivatelsky rezim - daji se zde ale definovat ruzne akce, ktere se maji provadet pri prepnuti do techto rezimu,
- je mozne zmenit urovne behu (rezimy) pomoci *tellinit N*.

Konfigurace urovni behu:

- v adresari /etc/rcX.d (X=uroven behu), jsou skripty spoustene pri vstupu do dane urovne,
- nejprve se volaji skripty zacinajici K v poradí danem císlem za tím K (volaji se s argumentem stop),
- pote se volaji skripty zacinajici S (volaji se s argumentem start),
- start, stop - definuje se co se ma spustit ci jak se co ma zastavit,
- v adresari /etc/init.d vytvorime skript, ktery pozadovanou sluzbu bude umet spoustet, na patricne misto K a S odkazu se vytvori symlink na pozadovany skript,
- skripty v inid.d typicky prijimaji parametry start, stop, reload, restart
- tyto skripty se nemusi volat pouze pri zmene urovne behu, ale je mozne je volat i rucne - z /etc/inid.d
- v souboru /etc/inittab je horni, hlavni uroven systemu, kde se popisuje napr. implicitni uroven behu ci jake urovne behu jsou podporovane

Existuji ruzne nove implementace procesu init - dnes nejbeznejsi je **systemd**:

- zakladni urovne behu jsou nahrazeny jednotky (units), ktere maji ruzne typy (targets, services, ..),
- spousti inicializacni jednotky paralelne na zaklade jejich zavislosti (= vyhodou je, ze inicializace systemu je mozne provadet paralelne, zatimco init se dela sekvencne),
- emuluji se urovne behu (zpetna kompatibilita),
- uzitecne jsou adresare /lib/systemd ci /usr/lib/systemd, .. (podrobne informace o systemd)

## 8.16 Planování procesu

Procesy planuje planovač.

Rozlišujeme 2 **planovací algoritmy** (definice viz. 1.5):

- nepreemptivní planování (typicky I/O operace, konec - volání exit, vzdá se CPU - yield),
- preemptivní planování (typicky prerušení od časovače, může jít i o jiné, třeba od disku)

Rozlišujeme 3 **”typy” planování**:

- dlouhodobé planování - které úlohy budou připuštěny do systému,
- střednědobé planování - procesy mají paměť / nemají paměť - jedná se o systém swapování,
- krátkodobé planování - procesy mají paměť - prepínání mezi úlohami

**System swapování** (= střednědobé planování):

- v případě nedostatku paměti některé procesy pozastaví, odebere jim veškerou paměť a uloží je na disk,
- tyto procesy jsou vyřazeny z planování (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 planování (rozhoduje se o tom, které úlohy budou vůbec připuštěny do systému)

**definice:**

*planovač* rozhoduje který proces či procesy pobeží a případně jak dlouho

*systemy s nepreemptivním planováním* = systémy s kooperativním planováním (procesy musí spolupracovat, kooperovat)

## 8.17 Prepnutí kontextu (procesu)

Prepnutí kontextu na příkladu - dispečer na základě rozhodnutí planovače prepíná mezi procesem A a B:

- bude muset uchovat stav registru (některých, ale včetně řídicích registru) procesu A do PCB (nebo task struct v linuxu),
- dojde k úpravě některých řídicích struktur v jádře (úprava planovacích struktur, uctovacích struktur, ...),
- obnova uložených hodnot registru procesu B,
- dojde k předání řízení procesu na adresu, kde bylo dříve prerušeno 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).

Přesto prepnutí může trvat **řádky/stovky/tisíce instrukcí** - jádra umožňují interval v jakém přicházejí prerušení z časovače - je třeba si dát pozor, aby ten interval nebyl příliš krátký, protože začne poté převazovat režie systému nad užitečným během (neustálé ukládání a obnovování obsahu registru).

## 9

**Devata prednaska:** Dokonceni spravy procesu.

### 9.1 Kratke planovaci algoritmy

#### 9.1.1 FCFS

- first come, first served,
- planovaci algoritmus zalozeny na jednoduche FIFO fronte,
- proces, který nove vznikne nebo je uvolnen z cekani na nejake operaci (I/O, sync, ..), pripadne proces, který se vzda CPU,
- se zaradi na konec fronty,
- procesy, které pobezi se vybiraji ze zacatku fronty,
- jedna se o nepreemptivni algoritmus (prepnuti kontextu dojde pokud se bezici proces vzda CPU ci zavola sluzbu jadra)

#### 9.1.2 Round-robin

- preemptivni obdoba FCFS,
- pracuje podobne jako FCFS,
- kazdy proces ma prideleno nejake casove kvantum,
- jakmile je mu pridelen CPU, proces bezi a pobezi nanejvys po dobu casoveho kvanta,
- po vyprseni casu je procesu odebran CPU a je zarazen na konec fronty,
- CPU se prideli procesu ze zacatku fronty

#### 9.1.3 SJF

- shortest job first,
- nejprve se provede nejkratsi uloha,
- algoritmus prideluje CPU tomu procesu, který aktualne deklaruje nejkratsi dobu pro svuj dalsi beh na CPU, po který nebude zadat o zadne I/O operace (tzv. CPU burst),
- beh uloh se deli na vypocetni prace (CPU burst) a pote na periody, kdy se komunikuje s periferiemi (disky, site,...),
- nepreemptivni algoritmus (neprerusuje proces pred dokoncenim jeho aktualni vypocetni faze),
- statisticky minimalizuje prumernou dobu cekani a zvysuje propustnost systemu,
- je nutne dopredu znat dobu behu procesu na CPU v jejich jednotlivych vypocetnich fazich, ci musi tu byt moznost tyto doby rozumne odhadnout (na zaklade predchoziho chovani techto uloh),
- dava smysl pro opakovane provadene ulohy,
- pouziva se zejmena v davkovych (specializovanych) systemech,

- nevýhodou algoritmus je stárnutí (hladovení, starvation) - ke stárnutí dochází při čekání na nějaké zdroje (CPU, zámek, ..) je situace, kdy některý proces, který o ten zdroj žádá, na něj čeká bez záruky, že jej někdy získá,
- 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)

#### 9.1.4 SRT

- shortest remaining time,
- preemtivní obdoba SJF,
- je zde prevence při vzniku či uvolnění procesu z čekání (když se uvolní nový proces a deklaruje kratší výpočetní fázi než ten, který dosud běžel - může být prováděn on),

#### 9.1.5 Víceúrovňové plánování

- procesy rozděleny 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 procesu dostane CPU čas - často jednoduše na základě priorit skupin,
- poté je další plánovací algoritmus, který planuje mezi skupinami

#### 9.1.6 Víceúrovňové plánování se zpětnou vazbou

- skupiny procesy jsou rozděleny dle priorit,
- proces, který se stane nově připraveným bezet (nově vznikne, je uvolněn z čekání, ..) je zařazen do skupiny procesu s nejvyšší prioritou.
- v této skupině bezí 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, proces čeká na mnoho I/O operací - priorita se zvýší,
- cílem je zajistit rychlou reakci interaktivních procesů,

#### definice:

*interaktivní procesy* jsou procesy komunikující s uživatelem



## 9.2 Planovac v Linuxu (od verze 2.6.23)

Pouziva se **viceurovneve prioritni planovani se 100 zakladnimi statickymi prioritnimi urovnemi**:

- priority 1-99 jsou vyhrazeny pro procesy realneho casu (algoritmy FCFS s preemci na zaklade priorit nebo round-robin),
- priorita 0 jsou bezne procesy planovane CFS planovacem,
- v ramci urovne 0 se pouzivaji podurovne v rozmezi -20 az 19, nejvyssi poduroven je -20 (je mozne bezne uzivatelsky nastavovat prikazy nice/renice),
- v ramci urovne 0 se rozlisuji 3 typu procesu (bezne, davkove a idle procesy),
- zakladni prioritni uroven (RT proces v 1-99, bezny proces) a typ planovani (round-robin, FCFS, ..) je mozne nastavit pomoci sluzby sched\_setscheduler,
- pozdeji pridano planovani pro sporadicke periodicke ulohy - zalozeno na strategii earliest deadline first (prezvalo z RT OS)

### definice:

*FCFS s preemci na zaklade priorit* - pokud behem behu procesu dobehne I/O operace ci sync operace, která zpusobi proveditelnost procesu s vyssi prioritou, dojde k prepnuti kontextu (nedochazi k prepnuti kontextu na zaklade casovych kvant s procesy s stejnou prioritou)

*davkove procesy* maji mirnou penalizaci s hlediska priorit, maji ale delsi kvantum

*idle* maji nizkou prioritu, procesy, u ktery se predpoklada, ze se dostanou ke slovu az v okamzik, kdyz v systemu nic uzitecnejsiho neni

*sporadicke periodicke ulohy* jsou ulohy, které bezi, maji periodicke vypocetni faze (ocekavana faze - znamo dobu jak dlouho trvaji + mame casovy limit, dokdy se maji vypocty provest), které se provadeji cas od casu

## 9.3 Completely Fair Scheduler

Neboli CFS planovac:

- snazi se explicitne kazdemu procesu poskytnout odpovidajici procento strojoveho casu s ohledem na jeho priority (4 procesy, stejná priorita - vsichni 25 procent CPU),
- u kazdeho procesu si vede udaje o tom, kolik virtualniho CPU casu uz ten proces na CPU straval,
- vede si udaj o minimalnim stravenem CPU case (dava nove pripravenym procesum),
- procesy udrzuje ve vyhledavaci strukture red-black tree podle vyuziteho CPU casu,
- pri rozhodovani, ktery proces pobezi, z strukturu vezme ten, ktery aktualne straval nejmene casu na CPU,
- proces necha bezet po casove kvantum, které spocita na zaklade priorit,
- virtualni procesorovy cas - situace: 2 procesy bezici cely den (kazdy pul dne), prijde novy proces, fyzicky na CPU bezel 0s (nejvyssi priorita, bezel by pul dne a ty 2 by byly off), proto virtualni CPU cas - novy proces dostane cas mensi nez minimalni straveny cas vsemi procesy (ne 0),
- algoritmus ma podporu pro skupinova planovani, umi rozdelovat cas spravdlive pro skupiny procesu (spoustene z ruznych terminalu, od ruznych uzivatelu, ..)

## 9.4 Planovani ve Windows NT a novejsich

- Pouziva se viceurovnove prioritni planovani se zpetnou vazbou na zaklade interaktivity:
- 32 prioritnich urovni, 0 - nulovani volnych stranek pameti (aby pres ne se nedostaly informace od jednoho uzivatele k jinemu), 1 - 15 bezne procesy, 16 - 31 procesy realneho casu,
- zakladni priorita procesu je dana kombinaci planovaci tridy a planovaci urovne (v ramci tridy),
- priorita se dynamicky snizuje ci snizuje:
- zvysi se priorita procesu spojene s oknem, ktere je v popredi,
- zvysi se priorita procesu spojene s oknem, do ktere prichazi vstupni zpravy (mys, casovac, klavesnice, ..),
- zvysuje se priorita procesu uvolnenych z cekani (I/O operace),
- zvysena priorita se po kazdem vycerpani kvanta snizu o jedno uroven (az do dosazeni zakladni priority)

## 9.5 Inverze priorit

Jedna se o **nezadouci problem**, ktery je nutne resit:

- jedna se o situaci, kdy v OS mame ruzne prioritni procesy, malo prioritni proces si naalokuje si nejaky zdroj, zamkne si pristup k nejakemu sdilenemu zdroji (soubor, adresa v pameti, sitovy port, ..),
- viceprioritni procesy tyto procesy predbihaji,
- nizkoprioritni proces se cas od casu dostane ke slovu, provede par instrukci, musi cekat,
- (ma naalokovany zdroj, chce s nim neco provest, to ale trva dlouho, protoze ho porad nekdo predbiha),
- muze nastat, ze nektery z viceprioritnich procesu potrebuje prave ten zdroj, ktery si zamknul tento nizkoprioritni proces,
- vysoceprioritni proces bude muset tak cekat - virtualne se zvysi (rapidne) priorita nizkoprioritniho procesu (= je to vedlejsi efekt, prioritu ma porad stejnou!),
- v systemu je mozne mit stredneprioritni procesy, ktere tento zdroj nepotrebuji, ty budou predbihat dale nizkoprioritni proces,
- vysokoprioritni proces musi cekat, zatimco stredneprioritni a nizkoprioritni maji najednou "vyssi prioritu",
- tento jev muze a nemusí vadit - muze zpusobit snizenou odezvu systemu, nicmene muzou se zablockovat i nektere kriticke procesy realneho casu (ovladani hardware, ..)

### Moznosti reseni inverze priorit:

- prioritni strop - priority ceiling - procesy v kriticke sekci ziskavaji nejvyssi prioritu,
- priority inheritance - procesy v kriticke sekci, ktery blokuje vyse prioritni procesy po dobu behu v kriticke sekci dedí prioritu cekajiciho procesu (s nejvyssi prioritou),
- na jednoprocessorovych systemech se pouziva technika, kdy po dobu behu v kriticke sekci se zakaze pre-ruseni

Dalsi komplikace behem planovani:

- viceprocesorove systemy - nutne vyvazovat vykon (aby na jednom jadru CPU nebezely 4 procesy a na zbytku 0), respektovat obsah cache CPU, lokalitu pameti (neuniformni pristup do pameti)
- hard real-time systemy - nutnost zajistit garantovanou odezvu nekterych akci

**definice:**

*kriticka sekce* je sekce v kodu, kde se pracuje vylucnym zpusobem se sdilenymi prostredky

*neuniformni pristup do pameti* - pamet je delena na pametove jednotky, kazda pripojena k jinemu CPU, ale vsechny procesy mohou pristupovat do vseh pametovych jednotek (za ruznou dobu)

## 9.6 Vlákna, ulohy, skupiny procesu

Vlákna, neboli threads:

- oznacovana jako odlehcene procesy (LWP - lightweight process),
- vypocty (odpovidaji vlaknum) bezici paralelne v jednom procesum,
- vlakna maji vlastni obsah registru, vlastni zasobnik,
- vsechna vlakna sdili stejny ridici kod, data, dalsi zdroje (otevrene soubory, signaly),
- vyhody: rychleji se spousti, prepina, efektivnejsi prace (dle systemu - v UNIXu diky fork rozdil mezi vlakny a procesy je mensi nez jinych OS), ..

## 9.7 Ulohy, skupiny procesu, sezení

**Uloha (job)** se poji se shellem, je skupina skupina paralelně bezicích procesu spustených jedním příkazem, příkazy propojené do kolony ( $p1 \text{ — } p2 \text{ — } p3$  - pipeline).

### Skupina procesu (process group) v UNIXu:

- množina procesu paralelně bezicích, se kterými je možné provádět operace jako s celkem,
- skupině je možné poslat signál jako 1 jednotce,
- predek také může čekat na libovolného potomka z určité skupiny,
- každý proces právě v 1 skupině procesu, po vytvoření vždy je to skupina jeho předka,
- skupina může a nemusí mít vedoucího - její první proces, dokud neskončí (pokud skončí - skupina 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 procesu,
- každá skupina procesu je v jednom sezení,
- sezení může a nemusí mít vedoucího,
- může mít řídící 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 použit příkaz nohup/disown, pošle navíc SIGHUP, pokud jsou procesy pozastaveny, tak pošle signál SIGCONT

## 9.8 Komunikace procesu

Pouziva se **IPC - inter-process communication**:

- signaly (umoznuji zasilat mezi CPU informace pomoci cisla),
- roury,
- zasilani zprav (umoznuji posilat retezcova data),
- sdilena pamet,
- sockety,
- RPC (remote procedure call), ...

## 9.9 Signaly

V zakladni verzi je cislo (int), ktere je procesu zaslano prostrednictvim pro to zvlast definovaneho rozhrani (= signaly v OS, nikoli na vodicich). Jsou generovany:

- pri chybach (aritmeticka chyba, chyby sbernic, ..),
- externich udalostech (dostupnost I/O, vyprseni casovace, ..),
- na zadost procesu - IPC (meziprocesova komunikace, procesy si mohou navzajem posilat singnaly, ale jadro signal nelze zaslát - jadro není proces),
- vznikaji obvykle asynchronne k cinnosti programu (program neco provadi, nezavisle na tom co provadi v okamziku ktery nelze predpovedet prijde signal)

Je nutne peclive zvazovat obsluhu signalu, aby aplikaci **signal neshodil, vznikaji chyby, ktere se objevuji jen zridka** (spatne ladeni, tzv. race conditions) - vede to na vyuzivani technik pro **pokrocile testovani** (vkladani sumu - umele v nahodnych okamzicich se snazime programy zpozdit, enumerace prolozeni akci programu), **nastroje pro verifikaci s formalnimi zaklady** (staticka analyza, model checking).

Mezi bezne pouzivane signaly patri:

- SIGHUP - odpojeni, ukonceni terminalu,
- SIGINT - preruseni z klavesnice (Ctrl+C),
- SIGKILL - signal c.9, tvrde ukonceni
- SIGSEV (mimo pridelenou pamet - spatny ukazatel), SIGBUS - chybna prace s pameti,
- SIGPIPE - zapis do roury bez ctenare,
- SIALRM - signal od casovace,
- SIGTERM - mekke ukonceni (lze vyvratit),
- SIGUSR1, SIGUSR2 - uzivatelske signaly (uzivatel si je muze nadefinovat)
- SIGCHLD - pozastaveni ci ukonceni potomka,
- SIGCONT - dochazi pri uvolneni z cekani,
- SIGSTOP, SIGSTP (Ctrl+Z) - tvrde / mekke pozastaveni,

- dalsi viz man 7 signal

**definice:** *race conditions* jsou casove zavisle chyby (zavisi na tom, jak se v case na sobe nakladaji paralelni akce)

### 9.9.1 Predefinovani obsluhy signalu

Mezi implicitni reakce na signal patri **ukoncení procesu** (pripadne s generovani core dump), **ignorovani signalu**, **zmrazení ci rozmrazení procesu**.

Predefinovat obsluhu lze u vsech signalu **mimo SIGKILL, SIGSTOP**. U SIGCONT vzdy dojde k odblokovani procesu (a nasledne se provede predefinovana akce).

Vlastni predefinovani obsluhy:

- pouziji se funkce signal (zakladni - jaky signal chci obsluhovat a jakou funkci, funkce ma jediny parametr - cislo signalu)
- nebo sigaction (urci se jaka funkce bude obsluzna, moznst nastaveni blokovani signalu behem obsluhy, dalsi specialni rezimy, .. )
- vice viz. man signal nebo man sigaction

Prednastavene konstanty:

- SIG\_DFL - prednastaveny signal ma byt obsluhovan implicitnim zpusobem
- SIG\_IGN - signal ma byt ignorovan

Z obsluzne funkce je mozne volat pouze bezpecne knihovni funkce. (viz. man 7 signal - na konci seznam funkci, ktere se mohou pouzit pri obsluze funkci)

### 9.9.2 Blokovani signalu

Je vhodne nastavit masku blokovani signalu, volani:

- sigprocmask (rekneme jake nastaveni signalu menime),
- pomoci SIG\_BLOCK (co chceme blokovat), SIG\_UNBLOCK (odblokovat), SIG\_SETMASK (natvrdo nastavit masku blokovanych signalu).

Blokovani se resi pomoci **bitovych masek**, ktere jsou typu sigset\_t. K vytvoreni masek je mozne pouzit predefinovana makra sigemptyset, sigfillset, sigaddset, sigdelset. Nelze blokovat signaly **SIGKILL, SIGSTOP, SIGCONT**.

**Nastaveni blokovani se dedi blokovani potomku** (proces si nastavi blokovani - fork - dedi to i potomci, pri execu obsluzne funkce zanikaji). Pokud chceme zjistit, zda nejake signaly cekaji, zavolame sigpending (preda se ukazatel na masku signalu). Pokud nejaky signal je zablokovan, ale prijde vicekrat, zapamtuje se jeho **vyskyt pouze 1x**. (neplati pro realtime signaly)

### 9.9.3 Zasilani signalu

Slouzi k tomu volani kill (s parametry pid - komu chceme signal poslat a cislo signalu, který zasilame). Umoznuje zasilat signaly:

- jednomu konkretnimu procesu (pid kladne),
- skupine procesu (0 - ve skupine, ve které proces je),
- vsem procesum, kterym proces muze signal poslat (pid = -1, nebo zapornejsi cislo - posle se dane skupine [-10] = vsem v skupine 10).

Aby mohl proces zaslat signal jinemu procesu, musi odpovidat **jeho UID, EUID ci saved set-user-ID ciloveho procesu** (nelze posilat signaly nekomu jinemu), pripadne se musi jednat o privilegovaneho odesilatele (napr. EUID=0, nebo CAP\_KILL).

Muze se pouzit i sigqueue pro volani s realtime signaly.

### 9.9.4 Cekani na signal

Meli bychom na signaly cekat **pasivne** (nikoli se aktivne neustale dokola ptat, zda signal uz prisel). Budto:

- jednoduche cekani pause,
- obvykle lepsi zabezpecene cekani sigsuspend - je mozne specifikovat masku signalu, které maji byt blokovany po dobu cekani a atomicky prepnout mezi signaly jsou blokovany do zacatku cekani a od zacatku cekani (muze se stat, ze probehne test, zda prisel nejaky signal a zjistí se, ze neprišel - začne se cekat a přijde mezi testem a začátkem čekání - čekání do nekonečna)

## 10

**Desata prednaska:** Nove tema - synchronizace procesu.

### 10.1 Synchronizace procesu

Synchronizace slouzi k tomu, aby procesy si vzajemne **nekolidovali a nedochazelo v systemu k nekonzistentnim stavum (ci datum)**. – Aby pri behu procesu nedoslo ke nekonzistenci dat, pouziva se synchronizace procesu (zajistuje spravne poradi provadeni spolupracujicich procesu).

Kdyz jadro dava **casove kvantum** **nejakemu procesu**, **tak to znamena, ze dava procesu nejaky casovac**, a kdyz dobehne, **spusti se preruseni**.

#### **Race condition:**

Casove zavisla chyba, ci race condition nebo soubeh, chyba, která muze vzniknout pri pristupu ke sdilenym zdrojum, datum - vznika pri pristupu ke sdilenym zdrojum kvuli ruznemu poradi provadeni jednotlivych paralelnich vypoctu v systemu.

### 10.2 Kriticke sekce

Jsou **useky kodu napric ruznymi procesy**, ve kterych **nesmi dojit k prepnuti kontextu** (nebo v nich nesmi byt nekolik procesu zaroven), **aby nedoslo k chybe soubehu dat**. (napr. vice procesu pracuje se stejnymi [globalnimi] promennymi) V programu muze byt **vice** kritickych sekci.

Problemem kriticke sekce rozumime **problem zajisteni konkretni synchronizae procesu na mnozine sdilenych kritickych sekci**, coz zahrnuje:

- vzajemne vyloucení - nanejvys 1 (ci k) procesu muze do kriticke sekce vstoupit (nebo: nanejvys 1 proces je v danem okamziku v dane mnozine sdilenych kritickych sekci).
- dostupnost kritickych sekci:
- pokud je kriticka sekce volna, chceme do ni vstoupit (nebo: opakovane volna v alespon urcitych okamzicich, proces nemuze neomezene cekat na pristup k ni),
- je potreba se totiz vyhnout:
  - uvaznuti (deadlock, viz nize a pozdeji),
  - blokovani (situace, kdy sdileny prostredek je volny, proces na nej ale musi dlouho cekat),
  - starnuti (hladoveni, proces se snazi vstoupit do kriticke sekce, ale nikdy k tomu nedojde, protoze ho planovac nikdy nevybere ve spravny okamzik).



### 10.3 Problemy vznikající na kritické sekci

Data race - **dochází k závodu mezi daty** - situace kdy jsou třeba dva přístupy ke zdroji s vylučným přístupem ze dvou procesů bez synchronizace, alespoň jeden přístup je zápis. Jedná se o:

- Uvaznutí (deadlock) při přístupu ke zdrojům s vylučným (omezeným) přístupem rozumíme situaci, kdy každý proces z určité množiny procesů je pozastaven a čeká na uvolnění zdroje s vylučným (omezeným) přístupem vlastněného nějakým procesem z dané množiny, který jediný může tento zdroj uvolnit,
- Livelock - speciální případ stárnutí - dochází k tomu s tím, že procesy vykonávají nějakou činnost, ale nikdy se nedostanou k tomu, aby udělali co chtěli (nikdy nedojde ke vstupu do kritické sekce),
- Blokování při přístupu do kritické sekce je situace, kdy proces, jenž žádá o vstup do kritické sekce musí počkat, protože je protože je kritická sekce volná a ani o zadnou z dané množiny sdílených kritických sekcí žádný další proces nežádá,
- Stárnutí je situace, kdy proces čeká na podmínku, která nemusí nastat - v případě kritické sekce je touto podmínkou umožnění vstupu do kritické sekce

### 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 sync 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 (init false, false) a sdílenou proměnnou turn = 0,
- pole nam říká, který z procesů chce přistoupit do kritické sekce,
- turn nam 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 (dokud je nastaven flag 1 - i a zároveň je na tahu ten druhý proces - cykli),
- proces 1 provede kritickou sekci a nastaví svůj bool na false

**Bakery algoritmus L. Lamport:**

- vyloučení pro n procesů (n znám dopředu),
- přijde do pekarství (nebo spíš na poštu), vezme si lístek s číslem a čeká, až na tebe dojde řada,
- před vstupem do kritické sekce získá proces přístupový lístek, jehož hodnota je větší než čísla přidělena již čekajícím procesům,

- pracuje s sdíleným booleovským polem příznaku (jestli hledám hodnotu maximálního ticketu), sdíleným int hodnoty ticketu a lokálními int hodnotami,
- v první fázi se snaží algoritmus zjistit, jaké je největší číslo ticketu a poté si vezme ticket s hodnotou maximum+1 více,
- dva (nebo) více procesů může mít stejnou hodnotu ticketu,
- poté se nastaví bool na false,
- ve druhé fázi proces čeká, až bude na něj 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 nejnižším číslem procesu (PID),
- jedním problémem algoritmu je neustálé zvyšování čísel (nutnost si dávat pozor na přetečení)

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

## 10.5 Vyuziti atomickych instrukci pro synchroniaci

Pouzivaji se spise tyto instrukce. Jsou zalozeny na vyuziti instrukci, jejich atomicita je zajistena hw.

### TestAndSet:

- v intelu lock bts, atomicka instrukce,
- lze si ji predstavit jako funkci co vraci bool a potrebuje odkaz na typ target (bool),
- nejdriv si ulozi to co je v pameti do pomocne promenne,
- pote na misto pameti ulozi true,
- vrati co tam bylo predtim (puvodni hodnotu),
- pri synchronizaci se to da vyuzit napriklad tak,
- ze proces ceká, az na danem pametovem miste bude hodnota false,
- projde do kriticke sekce,
- nastavi promennou na false a muze do ni pristoupit nekdo jiny,
- kriticka sekce je chranena tzv. zamkem

### Swap:

- v intelu lock a xchg, atomicka instrukce,
- nic nevraci, vymeni atomicky hodnoty ve dvou mistech v pameti,
- pri syncu se da vyuzit tak, ze se zamkne kriticka sekce prohozenim hodnot dvou promennych,
- vstoupime do kriticke sekce, a pote zamek odemkneme

Uvedena reseni zahrnuji moznost **aktivniho cekani**, a proto se take oznacuji casto jako **spinlock** (neustale se toci dokola a ptaji se na platnost podminky). Obecne jsou prilis drahe - procesy provadeji neuzitecny kod, pouze berou CPU cas.

Lze je vsak vyuzit **na kratkych, neblokujicich kritickych sekcich bez preemce** (tam kde neni prepnuti CPU).

Pristup do pameti (RAM) od CPU trva kolem 100-150 instrukci. Opakovany zapis sdileneho pametoveho miste je problematicky z hlediska **zajisteni konzistence cache v multicpu systemech** (zatezuje se sdilena pametova sbernice) - resenim je pri aktivnim cekani **pouze cist**.

### definice:

*atomicka instrukce* je instrukce, u ktere je garantovano, ze je atomicka

*atomicita instrukce* zn., ze instrukce nemuze byt prerusena

*prazdny while (sync)* - aktivni cekani

## 10.6 Semaforey

Synchronizacni nastroj nevyžadující aktivní čekání (nebo alespoň minimalizující - může se vyskytnout uvnitř implementace operací nad semaforem). Jedná se o **celociselnou proměnnou přístupnou dvěmi základními atomickými operacemi** (hodnota = kolik procesů do něj ještě může vstoupit):

- lock - zamknutí semaforu, proces vstoupí do kritické sekce a ostatní budou uspany,
- unlock - odemknutí semaforu - jiné procesy se probudí a vstoupí do kritické sekce.

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 (počte se proces probudí a bude dělat něco jiného),
- současně zamknutí více semaforů

Zabezpečuje sdílený přístup do kritické sekce tak, že **čekající procesy uspi** (místo aktivního čekání) a po uvolnění je opět probudí. (Pozn.: **poradí přístupu procesu není garantováno** (tzn. po uspaní procesu se netvoří fronta, ale spíše nějaká množina procesů))

Semantika celocíselné proměnné S - semaforu:

- S je kladné - odemknuto (kolik procesů ještě může přistoupit do kritické sekce),
- S je záporné - zamknuto (absolutní hodnota S udává počet čekajících procesů)

**Práce se semaforem:**

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

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

- zákazem přerušování,
- vzájemným vyloučením s využitím atomických instrukcí a aktivním 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 - stejný zámek může proces zamknout vícekrát,
- mutexy - binární semaforey, mohou být odemknuty pouze tím, kdo ho zamkl

- futexy - rychle mutexy pouzivane v linuxu v user-space (pri detecti konfliktu se vola sluzba jadra)

### Implementace semaforu:

Implementace semaforu (konkretne lock a unlock) tvori dalsi kriticke sekce, ktere se take musi zabezpecit (zamknout) **jiny synchronizacnim prostredkem**. Pouziva se spinlock na zacatku obou funkci, nasledne odemknuti spinlocku pred switchem, po ifu else a provedeni odemknuti.

U zkousky se implementuje semafor doplneny o kod spinlocku (TestAndSet nebo swap). Zde spinlock (na zacatku) pouze chrani kritickou sekci lock a unlock - aktivni cekani se zde da 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: prevzat z prezentace IOS: Synchronizace procesu - kod implementace semaforu

# 11

**Jedenactá přednáška:** Pokračování a dokončení synchronizace procesu, monitorů, deadlock

## 11.1 Monitory

Synchronizační prostředky (jste) vyšší úrovně (než semaforey). Problém semaforu v reálném kódu je, že **je zde spousta sdílených dat, které 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 semaforey jednoduše nejsou).

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

```
monitor monitor-name {
    shared variable declarations

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

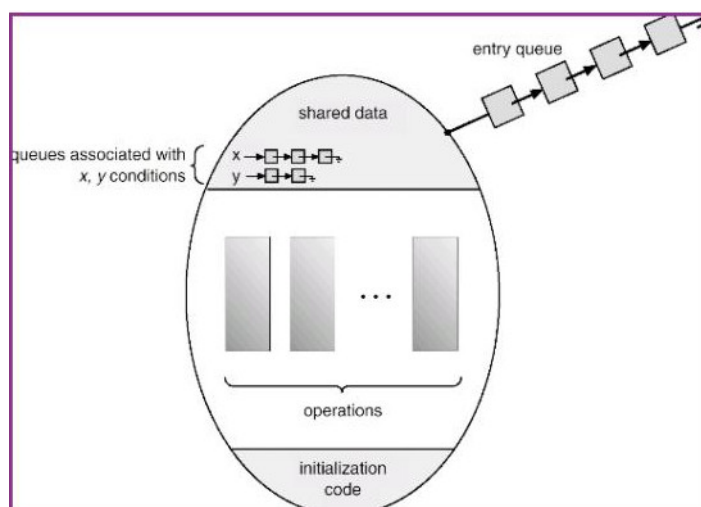
Obrázek 7: z prezentace IOS: Synchronizace procesu - kód monitoru

### Monitor:

- je možné si představit jako datovou strukturu podobně trídě se sdílenými proměnnými,
- ty jsou sdíleny různými procesy pracujícími s monitorem,
- procesy se mohou k proměnným dostat přes procedury (metody monitoru),
- automaticky monitor zajišťuje, že v daném okamžiku pobeží na něm nejvýše 1 metoda,
- kdokoli chce pracovat se sdílenými zdroji, musí používat tyto metody a zde se automaticky zajistí zamknutí/odemknutí zdroje,
- použití - do monitoru se nadefinují sdílené zdroje, dále mechanismy, jak se k nim má pracovat (procedury), dále konstruktor (inicializační kód)

### Popis monitoru:

- slupka (okraje) predstavuji ochrannou barieru monitoru,
- do monitoru je mozne vstoupit definovanymi operacemi (dvere),
- pokazde kdyz do monitoru nekdo vstoupi, tak se dvere uzamcou.
- dalsi "zajemci" o vstup jsou zarazeni do cekaci fronty vne monitoru (zarazovani tim, ze volaji dane operace v okamziku zamknutého monitoru),
- pokud se chteji synchronizovat, pouziji bud preddefinovane podminky programatorem, s kazdou z tech podminek se poji dalsi cekaci fronta (chci se sync pomoci podminky  $x$  -  $\text{wait}_x()$  - zarazeni do fronty  $x$  - soucasne bude vybrán nekdo, kdo ceka na vstup monitoru a bude moznost v nem bezet - ja cekam ve fronte  $x$  na nekoho, kdo se do monitoru dostane a provede signal ci notify - muze me uvolnit z fronty  $x$ )



Obrázek 8: z prezentace IOS: Synchronizace procesu - graficke ztvarneni monitoru

### Co kdyz se procesy chteji synchronizovat uvnitr monitoru?

- pouzivaji se mechanismus podminek - conditions,
- specialni datova struktura, s definovanymi operacemi nad ni, pomoci nich se procesy synchronizuji,
- je mozne volat  $\text{wait}()$  - synchronizovat se s nekym na nejake podmince,
- je mozne nekomu kdo na podmince ceka poslat uvolnujici signal (operace  $\text{signal}()$  nebo  $\text{notify}()$ )

### Rozliseni signal a notify:

- v monitoru muze bezet jen jeden proces,
- pokud v nem jsou dva (z toho jeden ve fronte dane podminky [napr. proces ve fronte  $x$ , druhy v monitoru a vola  $\text{signal/notify}$ ])
- $\text{signal}()$  - po zavolani pokracuje ten, kdo signal dostane (ten kdo je ve fronte [ $x$ ]) a bezi v monitoru, druhy proces se zaradi bud do cekaci fronty ci se zaradi do dalsi fronty uvnitr monitoru, kde takoveto procesy cekaji (a maji prednost pred procesy ve fronte venku)
- $\text{notify}()$  - po zavolani pokracuje volajici, resp ten kdo signal dostal (ten co byl ve fronte se presune bud do fronty mimo monitor ci do jine fronty uvolnenych procesu ale cekajicich)

- ceka se, az bude proces odejde z monitoru nebo zacne cekat (zaradi se do fronty podminky)
- pokud nikdo na podmince neceka a nekdo na ni pouzije signal ci notify, jedna se o prazdnou operaci (nic se neprovede)

Monitory je mozne implementovat **s vyuzitim semaforu** (vstupni semafore, semafore ke kazde podmince, pro prioritni procesy - pozastavene procesy ve fronte uvnitr monitoru). Monitory jsou dostupne v Jave ci C#. V POSIXu (C, C++, ..), jsou k dispozici alespon podminky (kombinace semaforu a podminek) - `pthread_cond_t` a funkce `pthread_cond_wait` / `signal` / `broadcast`.



## 11.2 (Některé) Klasické synchronizační problémy

### 11.2.1 Problem producenta a konzumenta

- máme soubor procesů, předem rozdělený, 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 semaforey (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 ukazovatkem do cache,
- **semaforey 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 procesu - sync prostředky producentu a konzumentu

### Pseudokod producenta:

- pracuji tak, ze vyprodukuji polozku, kterou chteji zapsat do cache,
- nejprve zamknou empty (pokud se to podari - je tam volne misto a rezervuji si ho pro sebe),
- zamknou mutex,
- zacnou zapisovat do cache (posunou ukazovatko),
- odemkne se pristup do cache - odemknuti mutex a full (konzumentum se da najevo, ze je zde polozka ke konzumaci),
- pracuje v nekonecnem cyklu do while 1

### Pseudokod konzumenta:

- nejprve zamknou full (pri uspechu - ve vyrovnací pameti je neco ke konzumaci, polozku si alokovali),
- zamknou si pristup k cache (zamkne se mutex),
- z vyrovnací pameti se odstrani 1 polozka,
- odemkne se pristup k cache (mutex) a nasledne se odemkne empty (da se najevo producentum, ze se v cache uvolnila polozka),
- zkonzumuje se polozka,
- pracuje v nekonecnem cykly 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 procesu - pseudokod producenta a konzumenta

### 11.2.2 Problem ctenaru a pisaru

- dva typy procesu - ctenari a pisari,
- pracuji se sdilenou pameti - ctenari ji mohou pouze cist, pisari jen menit,
- muze soucasne cist libovolny pocet ctenaru (nemeni se obsah pameti),
- pokud nejaky pisar, nesmi nikdo ani cist, ani psat,
- pouzva se sdilena promenna readcount - pocet ctenaru,
- nutne pouzit semafor mutex (chranici pristup k readcount) a semafor wrt (semafor pisaru),
- opet je nutna inicializace (pocet ctenaru=0, oba semafore=1)

```
int readcount;
semaphore mutex, wrt;

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

Obrázek 11: z prezentace IOS: Synchronizace procesu - sync prostredky ctenaru a pisaru

#### Pseudokod pisare:

- pokud chce zapisovat zamkne semafor wrt,
- az se mu to poradi, zapisuje,
- po dopsani odemkne wrt,
- pracuje v do while 1 cyklu

#### Pseudokod ctenare:

- v okamziku kdyz chce cist,
- je nutne zamknout pristup k promenne readcount (mutex), pote ji inkrementovat (zjisti tak, jestli je prvni),
- pokud readcount==1, tak se jedna o prvniho ctenare,
- potom se zamkne wrt a odemkne se mutex (readcount),
- zacne cist (pokud prijde dalsi ctenar, pouze readcount inkrementuje a cte),
- jakmile ctenar docte, zamkne mutex,
- dekrementuje se readcount, pokud je ==0, znamena to, ze je poslednim ctenarem a musi odemknout pristup pisarum (wrt),
- odemkne se pristup k readcount (mutex),
- pracuje v do while 1 cyklu

Toto reseni ma nevychodu - **hrozi vyhladoveni pisaru** - pokud do kriticke sekce vejde 1 ctenar, prijde pisar a zacne cekat, prijde dalsi ctenare, vejde do kriticke sekce, prvni odejde, druhy taky, ale opet vejde prvni ctenar -

<p>– <u>Písař</u>:</p> <pre> do {     ...     lock(wrt);     // writing is performed     ...     unlock(wrt);     ... } while (1); </pre>	<p>– <u>Čtenář</u>:</p> <pre> 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); </pre>
---	--

Obrázek 12: z prezentace IOS: Synchronizace procesu - pseudokod ctenaru a pisaru

do nekonecna se tu stridaji a pisar nikdy nezapise. Vyhladoveni je nekdy tolerovano, ale je dobre se mu vyhnout (mala pravdepodobnost, ze ctenari se budou do nekonecna stridat).

#### **Reseni vyhladoveni:**

- pouzije se dalsi semafor - wrt\_waiting - cekajici pisar,
- pisar nejprve zamkne wrt\_waiting, pote wrt, po zamknuti wrt odemkne wrt\_waiting,
- ctenari na zacatku provedou lock wrt\_waiting, unlock wrt\_waiting a pote pokracuji dal (zajisti, ze pokud nejaky pisar zacne cekat, tak vsichni ctenari doctou, a po navratu na zacatek nebudou schopni provest lock unlock)

### 11.2.3 Problem vecericich filozofu

- 5 filozofu, kteri reprezentuji procesy,
- sesednou se kolem kulateho stolu, kteri budou jist a debatovat, jedi pomoci asijskych hulek, ktere budou mit rozdeleny tak, ze mezi kazdymi filozofy je jedna hulka (ji se dvema..) - 5 hulek pro 5 filozofu,
- cyklus: ziska si svoji levou a pravou hulku, muze se najist, polozi hulky zpet, premysli, pote opet ji,
- reprezentuje situaci, kdy mezi synchronizovanymi procesy je cyklicka zavislost

#### Moznost reseni:

- ne dokonale reseni, moznost uvaznuti (=spatne) a
- je mozne, ze dva filozofove budou jist soucasne 1 hulkou,
- zavede se 5 binarnich semaforu (pole semaforu),
- vsechny inicializovane tak, ze jsou odemknute

#### Filozof I:

- $i = \text{ID}$ , hodnota 0-4,
- zamkne hulku  $i$ , nasledne zamkne hulku  $i+1$  modulo 5,
- pokud se podari zamek obou hulek, naji se,
- nasledne odemkne obe hulky,
- muze premyslet,
- do while 1 cyklus

```
semaphore chopstick[5];

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

// Philosopher 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 - pseudokod reseni tohoto problemu

**Gde deadlock?**

- procesy za planovani se mohou naskladat tak,
- ze kazdy si vezme svoji levou hulku (kazdy hulku se stejnym cislem jako jeho cislo 4-4, 0-0),
- nasledne si kazdy z nich bude chtit ziskat tu druhou,
- bude kazdy trvat na tom, ze si svoji necha a chce tu druhou,
- cely system se zastavi - deadlock

**Reseni deadlocku zde:**

- jednou z moznosti je ziskavat obe hulky soucasne,
- napr. pomoci semaforu system 5, umoznuji zamknout pole semaforu,
- dalsi moznosti je ziskavat hulky asymetricky - alespon 1 z filozofu bude brat hulky v obracenem poradi (napr. kazdy lichy filozof bude brat hulky nejprve  $i$  a pote  $i+1$  modulo 5 a sudi nejprve  $i+1$  modulo 5 a potom  $i$ )

## 11.3 Deadlock (uvaznuti)

### 11.3.1 Definice

*Uvaznutim (deadlockem) pri pristupu ke zdrojům s vylucnym (omezenym) pristupem rozumime situaci, kdy kazdy proces z nejake neprazdne mnoziny procesu je pozastaven a ceka na uvolneni nejakeho zdroje s vylucnym (omezenym) pristupem vlastneného nějakým procesem z dane množiny, který jediný může tento zdroj uvolnit, a to až po dokončení jeho použití.*

**Vysvetlení definice:**

- je specialni pripad deadlocku,
- neprazdna mnozina procesu = 1, 2, 1000, ... ne 0! - pouzivame tuto definici, jine pouzivaji mnozinu procesu kde je jich alespon 2,
- vylucny pristup = zdroj, který pouziva v danem okamziku nanejvys 1 proces,
- omezeny pristup = zobecnene kriticke sekce, kde zdroj muze pouzivat urcity pocet procesu, ale ne neomezeny pocet,
- pozastaven = neni tam aktivni cekani, nebezi (pokud to tam nebude, nebylo by mozne rozlisit deadlock a livelock),
- zdroj vlastneny nějakým procesem z dane množiny = obvykle jiny, muze to byt i stejny proces,
- jediný ... po dokončení jeho použití = pracujeme s tim, ze se pouziva zamek, který se pouziva neomezene (zadny casovy zamek), nikdo nemuze zdroj vzit, odemknout, nikde neni zadny casovac,
- kardinalita-pocet procesu množiny procesu=1 - tzv. self deadlock - proces zamkne zdroj a pak ho pokusi zamknout jeste jednou (doublelocking)

**Obecnejsi definice:** (s moznosti uvaznuti i bez prostredku s vylucnym pristupem - napr. zasilani zprav):

Uvaznutim rozumime situaci, kdy kazdy proces z nejake neprazdne množiny procesu je pozastaven a ceka na nějakou udalost, která by mohla nastat pouze tehdy, pokud by mohl pokracovat nektery z procesu z dane množiny. (je to napr. situace kdy proces 1 ceka na zpravu od procesu 2, ten ceka na zpravu od 3, 3 ceka na 4, ... proces 5 ceka na zpravu od 1 - obecnejsi uvaznuti)

### 11.3.2 Typicky priklad deadlocku

- pri pristupu ke zdrojům s vylucnym omezenym pristupem,
- v praxi mohou byt jednotlivá volani ve zdrojaku velmi daleko od sebe a zamykany mohou byt jen za urcitych podmínek,
- uvaznuti se tak projevi jen zridka a spatne se odaluje

```

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 procesu - příklad uvaznutí

#### Popis (kodu):

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

#### 11.3.3 Coffmanovy podmínky

Rozvádí to, co je obsahem uvaznutí do 4 nutných a postačujících podmínek, které jsou zapotřebí pro uvaznutí.

**K tomu aby uvaznutí mohlo nastat je nutné ci postačuji následující podmínky:**

- vzájemné vyloučení při používání prostředku,
- vlastnictví alespoň 1 zdroje, pozastavení a čekání na další,
- prostředky vrácí (pouze) ten proces, který jej 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 druhého - žádné aktivní čekání v cyklu)

#### 11.3.4 Řešení uvaznutí

- prevence uvaznutí,
- vyhybání se uvaznutí,
- detekce a zotavení

Základní společná myšlenka všech řešení je **princip, kterým se snaží zrušit platnost alespoň jedné z nutných podmínek k uvaznutí** (Coffmanových podmínek - když se jedna zneplatní - nemůže dojít k uvaznutí).



### 11.3.5 Prevence uvaznuti

#### Resi, jak tyto podminky zneplatnit:

- první (vyloučení)
  - nepoužívat žádné sdílené prostředky nebo užívat sdílené prostředky,
  - ale pouze takové, 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 zadat 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 zamkykám všechny nebo žádný zdroj - kontrola zamku: pokud proces již něco zamkl, nemůže znovu zamýkat - musí vše uvolnit a 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 - musí se zamknout na 30 min oba zdroje),
- třetí (navrácení)
  - nebude vyžadovat, aby proces vždy získal všechny zdroje, které chce používat současně v 1 okamžiku,
  - 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 zabít 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 pokusí vrátit zdroje do konzistentního stavu),
- čtvrtá (cyklická závislost)
  - zavedením uspořádání nad zdroji (ocislování),
  - je možné tyto zdroje získávat pouze od určitého pořadí (např. od nejmenšího k největšímu - začnu 1, pak 2, pak 3, .. - pokud zamknu 5, musu zamknout 5+, ale 3 už ne),
  - buď se konstrukcí programu zajistí, aby se vždy zamykalo v tomto pořadí, a poté analýzou, auditem se overí, 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.

### 11.3.6 Vyhybani se uvaznuti

Dalo by se to se chapat jako prevence 4. Coffmanovy podminky. **Obecny princip:**

- procesy musi predem deklarovat (jadru, systemu zpravy zdroju) informace o tom, jake zdroje a jak budou pouzivat, resp. ktere zdroje se budou pouzivat a kolik jednotek zdroje se bude pouzivat - nejjedoduseji musi kazdy proces rict, kolik jednotek ktereho zdroje bude pouzivat (dalsi informace napr. budu pouzivat zdroje 1 2 3 4, nikdy nebudu pouzivat soucasne 1 a 2, apod.),
- nasledne system pridelovani zdroju si vede informace co ty procesy deklarovaly,
- o jejich moznych pozadavcich,
- soucasne si vede o aktualnim stavu pridelovani (vi kdo co vlastni a kdo o co pozadal),
- v okamziku kdy system ma neuspokojene zadosti, ty uspokoji tehdy, pokud nemuze vzniknout zadna cyklicka zavislost na sebe cekajicich procesu, ani v tom nejhorsim moznem pripadu, který by mohl nastat s ohledem na to, co procesy deklarovaly

#### **Příklad:**

- situace v bance,
- od klientu mam sverene penize na terminovane vklady, vim kdy si klienti mohou sve vklady vybrat,
- prijde nekdo a bude chtit pujcku,
- je nutne se zamyslet kolik zdroju mam a kdy si to klienti mohou vybrat,
- pujcu mohu dat jen tehdy, kdyz v nejhorsim moznem pripade (vsichni klienti si pujdou vybrat co mohou) se nedostanu do dluhu,
- mohu penize pujcit

#### **Algoritmus zalozeny na grafu alokace zdroju:**

- resi problem vyhybani se uvaznuti,
- pro pripad binarnich zdroju,
- je veden systemem, který zdroje prideluje, ten si prubezne udržuje graf vztahu mezi procesy a zdroji - dva typy uzlu (procesy a zdroje) a tri typy hran - hrany od zdroje k procesu (který zdroj je kým vlastnen, zvýraznena, "tucna" hrana), hrany  $P_i$  a  $R_i$  od procesu ke zdroji (kdo o který proces zada),  $P_i$  a  $R_i$  od procesu ke zdroji (kdo o který zdroj muze pozadat - zvýraznena hrana, "tucna")
- zdroj system pridely pouze tehdy, pokud posoudi, ze v budoucnu nemuze nastat cyklicka zavislost procesu, jednoduse tim, ze "cvicne"provede otoceni zadosti na hranu vlastnictvi (z  $P_2$  do  $R_2$  udela  $R_2$  do  $P_2$ ), pokud v te situaci vznikne v grafu cyklus, znamena to, ze v budoucnu by mohl vzniknout deadlock,
- v takovem okamziku system nepovoli prideleni zdroje (proces bude muset dal cekat na uvolneni zdroje, i když je volny)

Pokud by se pouzivaly obecne zdroje se zobecnenou kapacitou, pouzil by se tzv. bankeruv algoritmus.

### 11.3.7 Deteckce uvaznuti a zotaveni

System pridelovani zdroji **umozni pripadny vznik uvaznuti** (pokud pomineme to, ze externe k mnozine uvaznutych procesu mame "strazneho andela", který uvaznuti vyresi), **ale periodicky** (bezi specialni proces, zajisťující ze nebude prebit prioritami jinych procesu) **se detekuje, jestli k uvaznuti nedoslo**, a pokud ano, provede se zotaveni.

#### **Detekce uvaznuti:**

- vedeni grafu vlastnictvi zdroju a cekani na zdroje (obdobny jak u vyhybani se uvaznuti) - stejny pocet uzlu, 2 typy hran (nekdo zada o zdroj, nekdo vlastni zdroj),
- pokud vznikne v grafu cyklus, vim, ze uvaznuti nastalo

#### **Zotaveni z uvaznuti:**

- alespon nekterym procesum, ktere uvazly, odeberu zdroje,
- proces se bud zrusi pozastavi se s tim, ze muze pokracovat, az bude moci ziskat vsechny zdroje, ktere potrebuje,
- muze nastat problem - procesy zabiju, mohou mit ve zdrojich rozpracovane nejake operace - zdroje mohou byt v nekonzistentnim stavu - nutnost but nechat system uvaznuty, nebo se spokojit s nekonzistencemi, nebo system navrhnout tak, ze pri zabiti procesu se nezabije ihned, ale prvne provede zotaveni (rollback - anuluje sve operace a dostane zdroje do konzistentniho stavu)

## 11.4 Formalni verifikace, verifikace s formalnimi koreny

Moznosti odhalovani nezadouchi chovani ystemu (uvaznuti, starnuti):

- inspekce systemu - nez se kod nasadi, krome vyvojare kod musi projit i nekdo dalsi (ci skupina), kteri schvali, ze kod pochopili a je podle nich bezchybny,
- simulace, testovani - vestavene systemy - vytvori si model systemu a z nej se generuje kod, na modelu overuji chovani systemu - nevyhodou testovani na jedne jednotce (modelu) je to, ze se chyba nemusí projevit (nedeterminismus se nemusí projevit) - paralelni programy - vkladani sumu do planovani (na kriticka mista pred ne se vlozi nahodne zpozdeni, prepnuti kontextu), zvysi se tim mnozstvi prolozenych aktivit a sance ze se najde chyba,
- dynamicka analyza - sleduje se co se deje v systemu, pote je snaha extrapolovat (=”vestit”), co by se mohlo stat,
- formalni verifikace ci verifikace s formalnimi koreny - pokud se rekne, ze program nema chybu ci urcitou vlastnost, tak ji ma s platnosti matematickeho dukazu,
- nebo kombinace vyse uvedenych pristupu

Experimentuje se i s automatickymi opravami - je zde **behovy system, který monitoruje, co se deje**, pokud uvidi, ze nastala chyba, **pokusi se ji opravit automaticky** (napr. sledovani date race conditions - pri poruseni vzajemneho vylouceni pri pristupu ke sdilene promenne se automaticky prida zamek - mohu ale tzv. pacienta zabít - zpusobit uvaznuti, proto se napríklad místo toho před práci s danou promennou místo vkladani zamku vynuti prepnuti kontextu - zvysi se sance, ze se projde kritickou sekci bez prepnuti kontextu)

**Proces formalni verifikace:**

- vytvoreni modelu - vytvori se zdrojovy kod, ci model, který se bude verifikovat (ci kombinaci, napr. cast jadra OS a model OS, odlehcenou implementaci),
- specifikace vlastnosti, které mají být overeny - mohou to být genericke vlastnosti napr. v systemu nesmi být deadlock nebo slozitejsi vlastnosti napr. v cache není nikdy více než 10 položek,
- kontrola (automaticka), zda model splnuje specifikaci

**Overovani se provadi metodami:**

- model checking (kontrola modelu),
- theorem proving (dokazovani teoremu),
- static analysis (staticka analyza)

Nad ramec verifikace se take provadi automaticka analyza dle dane specifikace - doda se specifikace, jake vlastnosti system ma být, pro urcite tridy systemu je mozne automaticky vysyntetizovat korektni implemetaci.

#### 11.4.1 Theorem proving

- teoremem je zde veta, která říká 'Můj program XXX splňuje XXX specifikaci',
- používají se poloautomatické dokazovací prostředky (evidují, co už jste dokázali, mají databáze standardně platných skutečností z logiky, znají pravidla správného odvozování),
- vyžaduje se expert, který určuje, jak se důkaz má vést (přesto se to používá, např. Mikrojadra ProvenCore či seL4),
- existují i plně automatické dokazovace (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 stavů systému a stavového prostoru (systematicky se hledá, zda někde není chyba),
- nevýhodou je obrovský počet stavů - např. při  $N$  dvoustavových procesech (procesy začínou a končí), může se vygenerovat  $2^N$  stavů - problém stavové exploze

#### 11.4.3 Static analysis

- snaha analyzy a verifikace systému na základě jeho zdrojového kódu, aniž by se tento kód prováděl (nebo alespoň ne v původní semantice - např. celocíselné proměnné 0 - 1235, pamatují si jen jestli je hodnota záporná, 0 či kladná),
- nejjednodušší statický analyzátor je grep (najde 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**

**Dvanacta prednaska:** Zacatek spravy pameti.

### **12.1**

TUDU