VYSOKE UCENI TECHNICKE V BRNE
FAKULTA INFORMACNICH TECHNOLOGII

IOS - Operacni systemy

poznamky z prednasek (the Little Book of IOS)

2019/2020

Corse & US Layout

Obsah

Preambule

Dokument je delen dle prednasek a toho, co se probiralo na prednaskach IOS (ak. r. 2019/2020). Tedy kazda hlavni kategorie znaci cislo prednasky, podkategorie znaci probirane tema, a popr. se pouzily i podpodkategorie pro rozdeleni velkych temat do vice useku.

Je docela mozne, ze spousta veci bude pouze prepisem obsahu prezentace, nicmene takto cely dokument (ani jediny radek, az na vyjimky jako je treba deadlock) zpracovavan nebyl. Dokument byl zpracovan za pomoci zaznamu prednasek, a to tak, abych dane tema ci latku dostatecne pochopil a zaroven bylo zajisteno nejakym zpusobem, aby dane vysvetleni nebylo az moc polopate a stacilo napr. ke zkousce.

Nekdy pro pochopeni dane latky byly vhodne obrazky, nekde byly i nutnosti (napr. princip fs). Take jsou obrazky pouzity pro pseudokody, protoze nejake normalni zpracovani kodu v latexu by trvalo zbytecne dlouho. Vsechny obrazky (i pseudokody) byly pouzity z ruznych prezentaci IOS. Moje rozhodne nejsou - pod popiskem kazdeho obrazku je tak zminka o tom, odkud dany obrazek pochazi.

Diakritika nebyla pouzita z duvodu US-EN layout. (jedna se o zapisky, ne bestseller ...)

Co se tyce pochopeni obsahu, bylo by dobre spolu pred ctenim jakekoli kapitoly (=prednasky) navstivit danou prednasku nebo si ji alespon pustit ze zaznamu. Bez toho je mozne, ze nektere veci nebude mozne pochopit.

1

Prvni prednaska: Uvod do predmetu, prehled operacnich systemu, zakladni pojmy, jadro operacniho systemu a jejich typy, historie vyvoje operacnich systemu, prehled technickeho vybaveni, klasifikace pocitacu, operacnich systemu, hlavni smery ve vyvoji operacniho systemu.

1.1 Uvod, prehled 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), ktera vytvari spojujici mezivrstvu mezi hardware operacniho systemu a uzivateli a jejich uziv. aplik. programy. OS dale spotrebovava zdroje, jako jsou pamet nebo cas CPU. (tldr: sw, spojujici hardware, uzivatele a programy)

Cile OS:

- maximalni vyuziti zdroju pocitace - drahe pocitace, levnejsi pracovni sila (drive)
- jednoduchost pouziti pocitacu - levne pc, draha pracovni sila (dnes prevazuje)

Zakladni role OS:

- spravce prostredku
  - pamet, procesor, periferie
  - dovoluje sdilet prostredky efektivne a bezpecne
- tvurce prostredi pro uzivatele a jejich aplikacni programy
  - vytvareni abstrakci, virtualnich objektu (resp. poskytuje standardni rozhrani, ktere zjednodusuje prenositelnost aplikaci a zauceni uzivatelu)
  - abstrakce jsou napr.: proces, program, soubor
  - problemy abstrakci jsou mensi efektivita a nepristupne nektere nizkourovnove operace

OS zahrnuje:

- jadro (kernel),
- systemove knihovny a utility (= systemove aplikacni programy),
- textove (shell) ci graficke uzivatelske rozhrani (X Window).

Presna definice, co vse OS zahrnuje neexistuje. Ruzne firmy a komunity to chapou ruzne. (GNU to chape napr. jako projekt svobodneho OS, zahrnujici jadro, utility, GUI, TUI, vyvojove prostredky a knihovny, ...)

definice:

proces je aktivita rizena programem (podrobneji se jim venujeme od 8.4)
program je predpis, navod na nejakou cinnost zakodovany vhodnym zpusobem
soubor je kolekce zaznamu (obvykle Byte) slouzici primarne jako zakladni jednotka pro ukladani dat na vnejsich pametovych mediich
adresar je kolekce souboru

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 zapouzdruje.

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 jadru, 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 proces - spusti proces) - 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 kontrolovanemu prepnuti do rezimu jadra, provede se sluzba, a pote se prepne zpet

ABI = Application Binary Interface

1.4 Typy jader OS

Monoliticka jadra

- vysokourovnove komplexni rozhrani s radou sluzeb, abstrakci, ktere mohou pouzivat vyssi vrstvy OS
- vsechny subsystemy jsou implementovany v privilegovanem rezimu, rezimu jadra, a zahrnuji napr. spravu pameti, planovani, meziprocesovou komunikaci, souborove systemy, ..
- vyhody: vysoka efektivita diky provazanosti
- nevyhody: 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)

Monoliticka jadra s modularni strukturou

- vylepseni koncepce monolitickych 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, ktere 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 soucaasne bezicich implementaci ruznych sluzeb, dynamicke spousteni, zastavovani..), zabezpeceni (chyba v serveru / utok na ne neznamena ovladnuti celeho OS, ale jen daneho serveru)
- nevyhody: 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, ktery 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, ktery 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 jadrem (= 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, nevyhoda: 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 jadrem, napr. pomoci preruseni (jakehokoli typu - vice viz 8.16)

1.6 Prehled technickeho vybaveni

Procesor (CPU):

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

Pamet:

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

Periferie:

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

Sbernice:

- propojuji jednotlive komponenty
- na vrcholu hiearchie 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 surovy 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 - doporuceni aby se akce vykonavaly v realnem case
hard real-time - akce se musi vykonavat v urcitem case

1.9 Implementace OS

OS se obtizne programuji a ladi, protoze to jsou velke programove systemy, paralelni a asynchronni systemy, systemy zavisle na technickem vybaveni.

Dusledky:

- setrvacnost pri implementaci (snaha nemenit kod, ktery pracuje spolehlive)
- pouzivani technik pro minimalizaci vyskytu chyb (inspekce zdrojoveho kodu, rozsahle testovani, podpora vyvoje technik formalni verifikace)

definice:

paralelni system zn. ze zde bezi vice aktivit soucasne
paralelni asynchronni systemy - procesy se prepinaji v okamzicich, ktere nelze dopredu presne predpovedet

1.10 Hlavni smery ve vyvoji OS

- neustale vylepsovani architektur (snizovani rezii jader,)
- bezpecnost, spolehlivost
- podpora stale vetsiho poctu procesoru, vice jader
- virtualizace
- distribuovane zpracovani (cloudy, kontejnery, Internet of Things)
- OS tabletu, mobilu, vestavenych systemu, ...
- vyvoj novych technik navrhu a implementace OS (podpora formalni verifikace)

definice:

bezpecnost zn., ze system je odolny vuci vnejsim utokum
spolehlivost zn., ze system ”nespadne sam 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 podobe, vstup zpracuji a na vystupu opet otevreny dokument (zadne binarni, zakodovane)

2.2 Varianty UNIXu

Hlavni vetve OS UNIXoveho typu:

- UNIX System V (puvodni system z AT&T),
- BSD UNIX (FreeBSD, NetBSD, ..),
- firemni varianty (AIX, Solaris, ..)
- Linux

Sousisejici 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 striktni podmnozina Single UNIX Specification, je to standard definujici zakladni textove prikazove rozhrani OS + API

2.3 Zakladni koncepty

Jsou dve zakladni koncepce (abstrakce) UNIXu: **procesy** a **soubory**.

Procesy mezi sebou komunikuji pomoci ruznych mechanismu meziprocesove komunikace - IPC (Inter-Process Communication) - roury, signaly, semafory, sdilena pamet, sockets, zpravy, streams, .. a pro komunikaci pouzivaji nejake I/O rozhrani (read, write, close, ..)

definice:

procesy jsou abstrakci probihajici nejake aktivity (viz 1.2)
soubory jsou abstrakci 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 jadrem komunikovat primo, nicmene nejcasteji komunikuji s jadrem pres knihovny. (viz. 1.3)
- Na druhem okraji (tesne nad HW) je vrstva abstrakce hadrware.
- 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 stejnych 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 znakova a blokova (kratsi definice viz 5.9)

znakova zarizeni jsou zarizeni komunikujici po jednotlivych znacich (klavesnice)

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

komunikaci s jadrem 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 jadrem a hardwarova preruseni

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

Priklady nekterych sluzeb jadra (systemova volani v UNIXu):

- open, close, read - otevre/zavre/cte soubor,
- write - zapisuje,
- kill - posle signal,
- fork - duplikuje proces,
- exec - prepise kod,
- exit - ukonci proces.

2.5.1 Hardwarove preruseni

- hardware interrupt je mechanismus, kterym HW zarizeni oznamuji jadru asynchronne vznik udalosti, ktere je zapotrebi obslouzit (dalsi mozna definice viz 1.5),
- zadosti o HW preruseni prichazi jako elektricke signaly (IRQ) do radice preruseni (APIC),
- procesor s radicem preruseni komunikuje pomoci I/O portu.

Prijem nebo obsluju HW preruseni lze zakazat:

- maskovanim preruseni,
- na CPU (instrukce CLI/STI na Intel/AMD - zakazou se vsechna krome NMI),
- ciste programve v jadre (preruseni se prijme, ale jadro si jen poznamena jeho prichod a neobsluhuje se)

NMI:

- non-maskable interrupt je HW preruseni, ktere nelze zamaskovat na radici ani zakazat na CPU,
- pouziva se pri kritickych chybach pameti, sbernice, .. (alternativne se pouziva pro ladeni / reseni uvaznuti v jadre ”NMI watchdog”)

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

- trap - po obsluze se pokracuje dalsi intrukci (breakpoint, overflow, ..)
- fault - po obsluze se znovu opakuje intrukce, ktera vyjimku vyvolala (vypadek stranky, deleni 0, ..)
- abort - dochazi k zavaznym problemum detekovanym CPU, neni jasne jak pokracovat - provedeni se ukonci (zanorene vyjimky typu fault, chyby HW detekovane CPU)

Mohou existovat i dalsi typy preruseni: (tato preruseni obsluhuje CPU zcela specifickym zpusobem (casto 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 Interrput (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, ktery 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, ktere mohou byt v nekonzistentnim stavu (napr. zretezene seznamy procesu [ukazatele], ruzne si je projuje, nez je stihne propojit, prije 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,
  - nemusi se zakazovat preruseni,
  - muze bezet v specialnich procesech (interrup 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 prvnim pouziti se musi registrovat k obsluze urciteho 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 prislusneho 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 Priklad komunikace s jadrem

Synchronni komunikace je proces-jadro, asynchronni je hardware-jadro. Priklad (detailnejsi, ale na tema pristupy na disk viz 5.9):

- proces A zavola sluzbu read() a jadro ihned zacne volani obsluhovat (synchronni)
- nejprve se podiva do cache zda data, o ktera ma zajem proces A uz tam nejsou
- pokud ano, tak mu je rychle nakopiruje z cache na adresu, kterou pozaduje proces (bez komunikace s diskem)
- pokud data nejsou v cache, proces A bude pozastaven a jadro vyda prostrednictvim ovladacu disku prikaz k nacteni urciteho objemu dat, typicky vice nez zada uzivatel a nacita do vyrovnavaci pameti (ne na pozadovanou adresu)
- na procesoru dale bezi proces B, taky pozada o read(), zopakuje se to same co u A
- az disk dokonci operace jednoho z procesu (nemusi byt v poradi volani), disk posle preruseni na CPU
- jadro bude informovano, ze ma potrebna data pro proces A/B
- z cache nakopiruje pozadovana data na pozadovanou adresu
- pote se proces A/B probudi a bezi dal, to same se stane u dalsiho procesu

definice (pro 2.5.x):

asynchronni zn., bez prime-okamzite vazby na to co dela jadro nebo aplikace (tiskarna tiskne - operace nekdy skonci - ale nikdy nevim dopredu kdy presne)

synchronni zn., ze CPU neco provede a ihned se zavola preruseni (napr. deleni 0)

IRQ = interrupt request

radic preruseni = interrput controller, hardwarova jednotka, ktera predava preruseni do CPU - registruje prichozi IRQ, ty se dle priorit predavaji do CPU (preruseni je mozne take zamaskovat - nepredavat dal do CPU) v podobe cisla preruseni, CPU se automaticky prepne do chraneneho rezimu a spusti obsluznou rutinu definovanou jadrem (preruseni 1 - provede xxx, 2 - xxx, ..)

APIC = Advanced Programmable Interrupt Controller - distribuovany system, kazdy CPU ma lokalni APIC, externi zarizeni mohou byt pripojena primo / pres I/O APIC

NMI watchdog - jadro si nadefinuje, ze casovac mu kazdych n casovych jednotek posle toto preruseni - pokud dojde v jadre k uvaznuti pri obsluze jineho preruseni a vsechna preruseni budou zakazana, toto se vzdy dostane do CPU (jadro se muze zotavit)

vypadek stranky zn., (pamet je rozdelena na casti, ktere mohou byt rozdeleny na disk) kdyz proces bude sahat do pameti a sahne na stranku, ktera v ni neni - detekuje se ze stranka tam neni - poruseni ochrany pameti - jadro zkontruluje, zda proces nesaha kam by nemel, a pokud ne, tak mu stranku nahraje zpet do pameti a znovu se provede ta stejna instrukce

bezne synchronizacni prostredky jsou napr semafory nebo zamky a synchronizuji procesy

linux:

zakladni statistiky o obsluze preruseni 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 ctvrta 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: pevny disk, diskove sbernice, sektory, parametry pevnych disku, SSD, problematika zapisu SSD, zabezpeceni disku, diskova pole (RAID), ulozeni dat na disku, fragmentace, pristup na disk a jeho planovani, logicky disk.

5.1 Pevny disk

Popis:

- uvnitr maji radu kulatych ploten, zaznam se provadi na kazdem z tech dvou povrchu, je v soustrednich kruznicich (= tracks, stopy)
- vsechny plotny jsou na stejne ose, pridelane k sobe a rotuji soucasne
- k nacitani slouzi sada hlavicek, cteci a zapisove, jsou tam v tolika kusech, kolik je tam povrchu (napr. 3 plotny = 6 povrchu = 6 hlavicek), vsechny umistene na jednom rameni, vsechny hlavicky se pohybuji soucasne
- hlavicky jsou nastavene na sade nekolika stop (kruznic) o stejnem prumeru = cylindr,
- stopy se deli na sektory
- velikosti sektoru byly drive 512B, u CD/DVD 2048B, dnes 4096B

Adresace sektoru:

- ze zacatku se pouzival CHS - urci se se kterym cylindrem chci pracovat, dale s kterou hlavou a jakym sektorem v ramci stopy,
- v soucasne dobe se pouziva LBA, kde jsou sektory (bloky) cislovane (jako adresy v pameti) od 0 po n, diskova jednotka si musi tato cisla prevadet na CHS

Periferni ci diskova rozhrani:

- pouzivaji se pro pripojeni disku,
- nejbezneji se pouziva ATA, drive se pouzivala v paralelni verzi (PATA - jednotlive byty se posilaly paralelne, pri rostoucich rychlostech byl problem zajistit synchronizaci techto dat), nyni v seriove verzi (SATA)
- take se pouziva SCSI ci SAS (Serial Attached Scasi), USB, FireWire, FibreChannel, Thunderbold, PCI Express nebo NVMe (pripojovani nejrychlejsich SSD),
- nad temito rozhranimi muze byt dalsi HW rozhrani propojujici tyto sbernice, jako treba 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 tercialni pameti. V systemu vznika hierarchie pameti, viz. 1.6.

definice:

cylindr (v HDD) je mnozina stop o stejnem prumeru
sektor je nejmensi jednotka diskoveho prostoru, ktery mi umozni diskova elektronika nacist nebo zapsat
blok nebo *diskovy blok* je sektor v HDD
alokacni blok nebo *blok souboroveho systemu* je nejmensi jednotka, kterou umozni alokovat OS
CHS zn. Cylinder Head Sector
LBA zn. Linear Block Adress

5.2 Parametry pevnych disku

Pristupova doba sestava z **doby vystaveni hlav** a **rotacniho zpozdeni**.

Typicke parametry soucasnych disku jsou kapacita, prumerna doba pristupu (jednotky ms u HDD) , otacky a prenosova rychlost. U prenosovych rychlosti se rozlisuje *sustained tranfer rate* a *maximum transfer rate*.

Mazani dat probiha tak, ze se prepisou metadata, pouze se poznamena (OS), ze dany soubor byl smazan.

definice:

doba vystaveni hlavicek zn., ze pokud nejsou nastavene hlavicky na stope, se kterou chci pracovat (malokdy), tak je nutne pohnout hlavickami (vic zasunout dovnitr nebo vysunout)
rotacni zpozdeni je doba nez mi pod spravne nastavenou hlavicku najede sektor (narotuje se disk)
maximum transfer rate je spickova prenosova rychlost, jak maximalne rychle je schopen disk komunikovat po kratkou dobu (typicky rychlost predani dat z vyrovnavacich pameti disku)
sustained transfer rate opravdova rychlost cteni z ploten

linux:

hdparm [-t] umoznuje zmerit prenosovou rychlost a menit parametry disku, -T meri rychost prenosu z vyrovnavaci pameti OS (RAM)

5.3 Solid State Drive - SSD

Mohou byt zalozena na ruznych technologiich, nejcasteji na nevolatilnich pametech NAND flash nebo DRAM (se zalohovanym 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 - vyzaduje specialni pdporu).

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 jednotlive (prepisovat ne!),
- pokud chci prepisovat (jednu stranku), je nutne cely blok nacist do pameti, vymazat (zresetovat) a v pameti upraveny blok nacist zpet (= write emplification, 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 nektere 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 databaze, ktere si ukladaji data do velkeho predalokovaneho 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 chybnych sektoru, ..) - S.M.A.R.T (podporovana vsemi ”rozumnymi”disky)

Pak je mozne jeste provadet testovani na urovni OS, napr. e2fsck nebo badblocks nebo si nektere 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 sektoru.

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 souhlasi

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, ktery je rychlejsi a spolehlivejsi.

Muze byt implementovan:

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

Ruznych typu RAID je nekolik (tzv. raid levels).

5.5.1 RAID 0

- data jsou rozlozena po dvou ci vice discich, ale kazdy datovy blok je ulozen jen na jednom disku (napr. dva disky, 0 a 1, prvni datovy blok [sektor, skupina sektoru] je na 0, druhy na 1, treti na 0, ...)
- vyssi efektivita cteni ci zapisu,
- je mozne paralelne cist ci zapisovat (do vice disku)
- prudce snizuje spolehlivost - pokud selze jeden disk, prijdu o data na nem

5.5.2 RAID 1

- disk mirroring, pro 2 a vice disku,
- vsechny bloky dat se zapisuji na vsechny disky,
- moznost cist a zapisovat paralelne,
- vyssi spolehlivost (data jsou na vsech discich)

5.5.3 RAID 2

- nejslozitejsi, proto se prilis nepouziva,
- pouziva zabezpecovaci Hemingovy kody,
- k urcitemu poctu datovych disku je urcity pocet zabezpecovacich disku,
- data se ukladaji na datovych discich na urovni bytu, k nim se dopocitavaji zabezpecovaci kody (napr. 4 datove - 3 zabezpecovaci),
- byty dat se rozlozi do vsech disku (ofc ty se musi prevest do bajtu a sektoru a zapisuje se to po sektorech)
- jediny RAID, ktery umi detekovat chyby, nektere i sam opravit, dokonce umi i zjistit, ktery disk selhal

5.5.4 RAID 3

- jednodussi zabezpeceni nez RAID 2, v podobe paritnich bytu,
- rozklada data po bajtech ci skupinach bajtu, ktere zabezpecuje partinim zabezpecenim (napr. 4 disky - 3 datove a 1 paritni).

5.5.5 RAID 4

- je analogie (tak jako RAID 3, akorat ..),
- provadi se rozkladani na urovni bloku-sektoru,
- nevyhoda u RAID 3 i 4 je pretizeni paritniho disku - pri zapisu/cteni se vzdy pracuje s paritnim diskem (a datovym) - na paritni disk se zapisuje tolikrat casteji, kolik mam datovych disku, tzn. vetsi pravdepodobnost selhani

5.5.6 RAID 5

- prakticky se uz pouziva,
- funkce paritniho disku neni vyhrazena pro jeden disk, ale mezi disky tzv. rotuje,
- napr. v konfiguraci se 4 disky, prvni 3 datove bloky se ulozi na 3 disky, na poslednim bude parita, pro dalsi trojici se ulozi na 3. disk, pro dalsi na 2., dalsi na 1., a potom zase na posledni, apod. .. = rovnomerne zatizeni disku,
- diky parite jsme schopni opet detekovat a korigovat chybu v jednom disku (pocet bitu neni sudy - chybi tam parity bit),
- parita se pocita dle sektoru (prvni bit 1. sektoru, prvni 2. sektoru, ..),
- pokud selze vice disku, nelze dopocitat bity (data)

5.5.7 RAID 6

- parita se uklada 2x,
- dokaze se vyrovnat se selhanim az 2 disku,
- vetsi redudance dat (obetuji se 2 disky jako parita)

RAID je mozne vytvorit i na jednom fyzickem disku (na kterem jsou logicke disky).

5.6 Opravy chyb u paritnich disku

- paritni disky pouzivane u RAID 3 - 6,
- jakmile clovek urci disk, ktery selhal, je mozne zreprodukovat jeho obsah,
- priklad: 4 disky, 1 paritni, treti datovy selze
  - prvni byty v datovych jsou 010 (potom v paritnim aby byl sudy pocet je 1), dalsi byty jsou 111 (licha parita, do paritniho disku se doplni 1 na sudou), dalsi jsou 011 (suda - v paritnim je 0)
  - selze treti disk, vymeni se za novy, prazdny
  - dopocitaji se data opet na sudou paritu: mam prvni byty 01? a v paritnim 1 - aby byla suda, v novem disku musi byt 0, dalsi byty 11? a v paritnim 1 - v novem musi byt 1, apod. ...

definice: (pro 5.5.x)

RAID = Redundant Array of Independent Disks
parita (bitu) je sudost/lichost bitu, pocet sudych/lichych 1 bitu
parity bit zn., ze na MSB se prida 1 pokud pocet 1 (bitu) je lichy

5.7 Ulozeni dat na disku

Diskova jednotka neumozni pracovat s nicim mensim nez sektor, ale typicky OS si sektory nejak seskupi (do vetsi jednotky) a neumozni pracovat s nicim mensim, nez je alokacni blok.

Logicka a fyzicka naslednost:

- 1 alokacni blok se namapuje fyzicky za sebou na diskovem prostoru,
- vice alokacnich bloku jiz nemusi byt fyzicky na disku za sebou (filesystem se vsak snazi o to, aby tomu tak bylo)

definice:

alokacni blok neboli cluster je skupina pevneho poctu sektoru, typicky mocnina 2 (nejmene 2^0 = 1 alokacni blok), pro sektory v ramci alokacniho bloku je zaruceno, ze jdou za sebou logicky i fyzicky (na disku) v souboru, dale je to nejmensi jednotkou diskovou diskoveho prostoru, se kterym bezne pracuje jadro (filesystem, uzivatel).

5.8 Fragmentace

5.8.1 Externi fragmentace

Rozumime jev, ktery vznika v pametech postupnym obsazovanim a uvolnovanim pameti, kdy v pameti vznika sekvence oblasti, ktere jsou volne a pouzite (a pouzite ruznymi soubory).

Priklad externi fragmentace:

- na disku vytvorim soubor 1, zabira urcite misto,
- pote dalsi soubor 2,
- pote soubor 1 chci zvetsit, tak se soubor 1 rozdeli na 2 casti - soubor 1.1 (puvodni misto kde byl - pred s2) a soubor 1.2, ktery bude za souborem 2,
- stejnym zpusobem zvetsim soubor 2 a vznikne sekvence s1.1, s2.1, s1.2, s2.2,
- nyni se rozhodnu smazat prvni soubor a budu mit sekvenci volne misto, s2.1, volne misto, s2.2 == externi fragmentace.

Externi fragmentace je i na plne obsazenem disku, kde staci, aby byl disk obsazen soubory nespojite (tzn. jeden soubor je rozdelen do vice casti, neni ulozeny na jednom miste, napr. s1.1, s2, s1.2 nebo viz priklad hore).

Negativni dopady externi fragmentace:

- na disku za urcitych okolnosti (v beznych FS nevznikaji) mohou vzniknout casti prostoru, ktere jsou jiz dale nevyuzitelne, protoze jsou prilis male (tldr vznik volnych useku, ktere nejdou vyuzit)
  - okolnosti (pri kterych vzniknou nevyuzitelne casti prostoru): pri alokovani diskoveho prostoru spojite (na miru souboru ci jeho castem, nepridelovani po jednotkach pevne velikosti) a navic budu mit dolni mez urcujici velikost diskoveho prostoru tak, aby byl pouzitelny (muze vzniknout v souvislosti s tim, ze do pouzitych diskovych oblasti si mohu ukladat pomocne informace, k cemu se pouzivaji - pokud bude informace vetsi nez ”volna dira”- nepouzitelna) - mam na disku bloky volneho mista o pozadovane velikosti (1GB, soubor, 0.5GB), ale protoze chci ukladat spojite (soubor o velikosti 1.5GB), nelze takove misto vyuzit
  - vznikne nespojite rozlozeny soubor (viz priklad) a je nutne si pamatovat v pomocnych datech = metadatech informace o tom, kde jednotlive casti souboru jsou (ukladaji se na mista, kde jednotlive casti jsou, ”odkazuji”se na dalsi metadata - dalsi casti smazaneho souboru),
- cim vice casti souboru - tim vice metadat - cim vice fragmentovane - tim vice je pristup na data pomalejsi (u HDD se ceka navic na natoceni hlavicek a rotace disku)

Souborove systemy se snazi negativni dopady fragmentace minimalizovat:

- rozlozeni souboru po disku (snaza ukladat soubory na disk tak, aby nebyly nutne za sebou, ale bylo mezi nimi volny prostor),
- pouzivani predalokace (uzivatel si pozada filesystem o vymezeni urciteho prostoru na disku, napr. databaze),
- odlozena alokace (=allocate-on-flush, filesystem nezapise ihned po zmene souboru, ale chvili pocka - pocita s tim, ze uzivatel bude chtit menit soubor ”za chvili”znovu - az nebude delsi dobu dochazet 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, ktere 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 fragmetace:

- 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, ktere 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 Pristup na disk

(viz 2.5.5) Proces kdyz chce nacitat/zpracovavat data, zavola sluzbu k tomu urcenou (read, write, .. - muze byt zabaleno i v nejakem knihovnim volani, napr. scanf zavola read), dojde k predani rizeni jadru, dostane se ke slovu jadro, podiva se do cache, pokud tam ta data ma, preda je, pokud ne, musi je nacist z disku - s diskem komunikuje pres I/O porty nebo pametove mapovane I/O porty, disku se predavaji prikazy pres jeho rozhrani (ATA disk - ATA prikazy), jdou z filesystemu pres ovladac prislusneho disku (pote to prochazi sbernicemi), ten komunikuje s radicem disku - disk dostane prikaz, jakmile dobehne operace, disk posle preruseni na procesor, tam se dostava ke slovu jadro, to zpracuje preruseni a zachova se podle nej (uspech - preda data, chyba - zpracuje ji).

definice:

ovladac je software, ktery umi komunikovat s urcitym typem zarizeni, jina definice viz. 2.4

5.10 Planovani pristupu na disk

Soucasti jadra je subsystem nazyvany planovac diskovych operaci, ktery shromazduje pozadavky od filesystemu (nacteni, zapsani dat z/do disku). Planovac si uklada pozadavky do svych planovacich front, pozadavky pripadne preusporadava a predava dal ovladaci ci radici disku k realizaci.

Planovac se snazi minimalizovat rezii disku.

Jednou ze strategii preusporadavani pozadavku (u HDD) je pouziti **vytahoveho alogritmu (elevator, SCAN alghorithm):**

- snaha, aby se hlavicka disku plynule pohybovala od stredu k okraji a zpet a vyrizovat pozadavky dle pohybu hlavicky,
- modifikace SCAN algoritmu je napriklad Circual SCAN, kdy se pozadavky vyrizuji pouze pri jednom smeru,
- dalsi modifikace jsou LOOK a C-LOOK, kde se hlavicka nepohybuje od stredu k okraji, ale pouze v tom rozsahu, kde je potreba provadet operace.

Planovac se muze snazit vice operaci sloucit do jedne operace (napr. operace v ramci jednoho bloku se sdruzi):

- takove kroky maji vyznam i u SSD,
- snaha vyvazovat pozadavky jdouci od jednotlivych uzivatelu (procesu),
- implementace priorit (prioritnejsi proces - pozadavky se vykonaji drive),
- snaha odkladat operace tak (v nadeji), ze je bude pote mozne sloucit,
- snaha implementovat casova omezeni na dobu cekani pozadavku,
- muze implementovat paralelizaci pozadavku predavanych do diskoveho subsystemu (modernejsi a velmi vykonne SSD umi resit operace paralelne).

linux:

pro zjisteni, jaky planovac pouzivame se staci podivat do */sys/block/<devname>/queue/scheduler*

5.11 Logicky disk

V pocitaci je mozne mit vicero fyzickych disku, ktere je dale mozne rozdelit na logicke disky a konkretni souborove systemy je mozne instalovat na logicke disky. Pro spravu a vytvareni logickych disku lze pouzit programy cfdisk, disk, gparted, ..

5.11.1 Zpusob ulozeni informaci o diskovych oblastech na disku

- MBR
  - v prvnim (nultem) sektoru byla tabulka obsahujici rozdeleni na 1-4 primarni partitions
  - pokud bylo nutne pouzit vice partitions, potom misto primarni se nahradila rozsirenou diskovou oblasti, ktera se dale mohla rozdelit na podoblasti zvane logicke diskove oblasti, kazda z nich popsana formou zretezeneho seznamu, EBR
  - pouzivane u starsich PC
- GPT
  - je tabulka (pole) o az 128 odkazech na jednotlive diskove oblasti,
  - stejny vyhrazeny prostor jako u MBR

5.11.2 LVM

- spravce logickych oblasti,
- umoznuje pokrocilejsi tvorbu logickych disku a
- do logickeho disku pridavat fyzicke disky (za behu),
- LVM muze byt bud primo ve filesystemu nebo v casti jadra (mezi filesystemem a planovacem).

5.11.3 Ruzne typy souborovych systemu

- fs (prvni 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, pote pouzivany ve Windows - velmi jednoduche a siroce podporovane),
- F2FS (fs pro efektivni prace se SSD), ISO9660, UDF, Lustre, GPFS (clustery, superpocitace),
- ZoneFS (ZFS).

Po koupe noveho disku a rozdeleni na logicke disky je nutne se rozhodnout, jaky souborovy system na prislusnem logickem disku bude pouzivan - je nutne disk **zformatovat** pro pouziti. Drive se pouzivalo i nizkourovnove formatovani (stare disky s nestabilnim magnetickym zaznamem).

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, ktera v jadre 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

Sesta prednaska: pokracovani Spravy souboru. Zurnalovani, jeho implementace a alternativy, Copy-on-write, Klasicky UNIXovy system souboru FS, i-uzly, kde a jak jsou data ulozena, pocty odkazu, limit maximalni velikosti souboru, vyhody a nevyhody FS, jine zpusoby organizace souboru, EXT4, NTFS, Organizace volneho prostoru na disku, deduplikace, typy souboru v UNIXu, adresar, montovani disku

6.1 Zurnalovani

Je technika zalozena na vytvareni zurnalu.

- souborove systemy se zurnalem jsou treba ext3, ext4, ufs, XFS, JFS, NTFS, ..
- zurnalovani umoznuje spolehlivejsi (nikdy nemame obecne zajisteno, ze se nic spatneho stat nemuze) a
- rychlejsi navrat (nez nejake utility) do konzistentniho stavu po chybach
- data obvykle zurnalovana nejsou (velka rezie), ale mohou byt
- zavisi na tom, ze operace, ktere zurnalovani implementuji, se provedou ve spravnem poradi – nutnost spoluprace s planovacem, take disky si samy data preusporadavaji (nelze nijak ovlivnit)

Zurnal:

- zapis zurnalu je předřazený,
- vytvari se v nem cyklicky prepisovany buffer,
- předřazenost zapisu do zurnalu mi zaruci, ze operace pokryte zurnalovanim jsou atomicke - vytvari transakce

Kompromis mezi zurnalovanim a nezurnalovanim dat je **předřazení zapisu dat na disk pred zapisem metadat do zurnalu** (a nasledne zapis ostrych metadat na disk). Priklad:

- zapisuji do souboru - bud vytvarim zcela novy nebo ho zvetsuji (typicke zpusoby zapisu),
- pri zvetsovani se nejprve zapisou data na disk za existujici data (bez poznamenavani informace o tom, ze se soubor zvetsuje),
- pokud operace selze, soubor zustane v puvodnim stavu (diky neupravenym metadatum puvodniho souboru),
- teprve az data budou na disku, tak se do zurnalu zapise informace o zvetsovani souboru,
- pote se zmeni metadata souboru (a uzivatel se k datum dostane),
- pri selhani napajeni v moment, kdy jsou metadata v zurnalu, ale ne na disku, je mozne tyto metadata obnovit

Proces mazani souboru na disku:

- odstraneni zaznamu z adresare,
- uvolneni uzlu (metadat souboru),

- uvolneni oblasti pouzitych tim souborem

definice:

zurnal je specialni soubor ci specialni oblast na disku slouzici pro zaznamy modifikovanych metadat (dat o datech), pripadne i dat pred jejich zapisem na disk (v podobe beznych dat)
předřazený znamena, ze zapis do zurnalu se provede pred ostrym zapisem "uzitecnych"dat (ci metadat) na disk
atomicke operace zn., ze but operace uspeje cela (vsechny dilci kroky) nebo neuspeje vubec (zadny dilci krok)

6.1.1 Implementace zurnalovani

Existuji 2 zakladni pristupy k implementaci zurnalovani.
REDO:

- implementace na zaklade dokonceni transakci,
- pouziva napr. ext3, ext4,
- sekvence dilcich operaci (vytvarejici tu operaci, kterou chci provest) se zapise do zurnalu (zacatek, konec transakce, kontrolni soucet),
- pote se operace provadi na disku,
- po uspesnem dokonceni se transakce ze zurnalu uvolni,
- pri selhani a pote zotaveni se system podiva do zurnalu, podiva se po neuvolnenych transakcich, jestli jsou cele - pocatecni a koncova znacka, jestli sedi kontrolni soucet - pokud vse sedi, tak system provede vsechny operace znovu

UNDO:

- implementace na zaklade anulace transakci,
- v kombinaci s REDO se pouziva v NTFS,
- proklada zaznam dilcich operaci (ktere se maji provest) do zurnalu a nasledne jejich provedeni na ostrych datech (zaznamena dilci operaci - provede ji),
- probehne cela transakce - zaznam ze zurnalu se uvolni,
- pri chybe se eliminuji vsechny nedokoncene transakce (vsechny provedene dilci kroky se musi vratit - vrati se disk do puvodniho stavu)

Pri implementaci zurnalovani je klicove **dodrzeni poradi kroku, ve kterem se provadeji**. (U REDO napr. je nutne, aby se nejprve zapsaly sekvence operaci do zurnalu a teprve pote se provadely operace na disku) Pokud tato sekvence nebude dodrzena, zurnal nebude spravne 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, ktery popisuje veskery obsah disku, typicky se v nem vyhledava na zaklade unikatni identifikace souboru

adresare (copy-on-write) jsou specialni soubory ulozeny na disku, ktere 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 Klasicky UNIXovy system souboru (FS)

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

Souborovy system byl rozclenen (na urovni logickych disku) na:

- boot blok - obsahoval informace (kod, cast kodu) potrebne pro zavedeni pri startu,
- super blok - informace o souborovem systemu (typ, verze, velikost, pocet i-uzlu, volne misto, korenovy adresar, volne i-uzly, ..),
- tabulka i-uzlu - tabulka (pole n i-uzlu) pouzita s popisy souboru,
- datove bloky - data souboru, metadata (pomocne adresovaci bloky).

Zakladni rozlozeni FS bylo zmodifikovano v navazujicich filesystemech:

- datove bloky byly rozdeleny do skupin,
- kazda skupina mela svoje i-uzly,
- duvodem byla lepsi lokalita, prostorova blizkost dat a metadat (typicky pri praci se soubory jsou nutne i jeho metadata),
- pote tedy ta struktura vypadala takto: boot blok, super blok, usek i-uzlu, usek dat, usek i-uzlu, usek dat, ..

definice:

i-uzel je zakladni datova struktura reprezentujici kazdy jeden soubor v typickych UNIXovych systemech (pozn. pri formatovani se urci dopredu maximalni pocet souboru, ktere na diskovem oddilu budou existovat)

6.2.1 i-uzel

Zakladni datova struktura popisujici soubor v UNIXu (nebo viz definice hore). Ke kazdemu souboru musi byt i-uzel. Ten obsahuje metadata o souboru:

- stav i-uzlu (alokovany, volny)
- typ souboru (obycejny, adresar, zarizeni, pojmenovana roura, ..),
- delka souboru v bajtech,
- casy mtime (posledni modifikace dat - zapis), atime (posledni pristup - cteni), ctime (posledni modifikace i-uzlu),
- UID, GID,
- pristupova prava (cislo, napr. 0644 = rw-r–r–),
- pocet pevnych odkazu (neboli jmen souboru),
- informace o tom, kde se nachazi data o souboru (tabulka odkazu na datove bloky a dalsi informace nebo odkazy na pomocne bloky s dalsimi metadaty, napr. ACL, extended attributes, dtime - udaj o smazani souboru, ..)

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

definice:

UID je cislo identifikace vlastnika

GIT je cislo identifikace skupiny

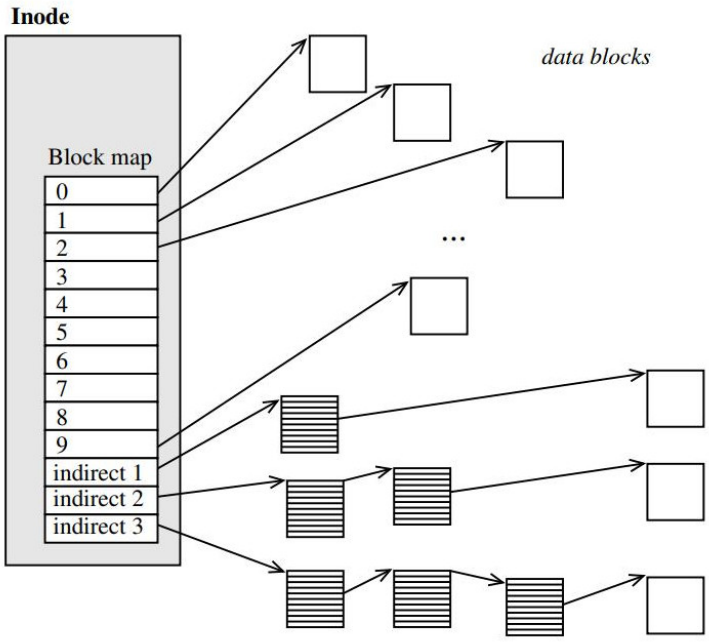
ACL zn. Access Control List, pristupove seznamy rozsirujici zakladni UNIXova prava tak, ze je mozne priradit konkretni prava ke konkretnim uzivatelum

extended attributes zn. rozsirene atributy, s jakymi specifickymi prave se muze soubor napr. spoustet

ctime je posledni zmena i-uzlu, vyuzitelne napr. pokud se zfalsuje mtime, posledni zmena souboru se pozna prave podle ctime

6.2.2 Kde a jak jsou ulozena data

- v i-uzlu je rada primych (az 10, novejsi unixove fs maji 12) i neprimych odkazu na data,
- prime odkazy odkazuji na alokacni bloky na disku,
- pokud je potreba vice odkazu, pouzije se neprimy odkaz prvni urovne, ktery odkazuje na specialni alokacni blok neobsahujici data, ale dalsi prime odkazy na data,
- pokud nestaci ani to, pouzije se neprimy odkaz druhe urovne, ktery odkazuje na pomocny adresovaci blok, ktery obsahuje dalsi neprime odkazy 1. urovne, ktere odkazuji na dalsi prime odkazy (a ty odkazuji na data) - vznika strom
- pokud ani to stacit nebude, pouzije se adresovaci blok 3. urovne, vedouci na adresovaci bloky s neprimymi odkazy 2. urovne, kazdy z nich na bloky s odkazy 1. urovne, ty vedou na prime odkazy a ty na data.



Obrázek 1: z prezentace IOS: Souborove systemy, slide 23 - odkazy v i-uzlech

6.2.3 Pocty odkazu

- neprimy odkaz 1. urovne - pri 4 KiB clusteru je to 1024 odkazu (1 odkaz = 4B) = 1024 datovych bloku,
- neprimy odkaz 2. urovne - pri 4 KiB clusteru je to 1024^2 odkazu = stejny pocet datovych bloku,
- neprimy odkaz 3. urovne - pri 4 KiB clusteru je tam 1024^3 odkazu = stejny pocet datovych bloku

6.2.4 Limit maximalni velikosti souboru

Poctem primych odkazu neprimeho odkazu 3. urovne je dan maximalni pocet bloku, ktere je mozne v tomto souborovem systemu ulozit. Teoreticky limit velikosti souboru je tak:

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

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

Toto omezeni velikosti je pouze jednim z omezenim, ktere velikosti souboru omezuji. **Dalsi omezeni jsou dana:**

- dalsimi datovymi strukturami a typy, ktere pouziva FS (napr. datovy typ delky souboru v bajtech v i-uzlu),
- strukturami VFS (veskera prace s jakymkoli filesystemem musi projit pres VFS),
- rozhranim jadra,
- architekturou systemu (32b - velikost souboru bude 32b cislo + MSB je pouzit pro indikaci chyby [-1 bit pro data] - soubory maximalne do 2 GiB nebo dnes bezna architektura 64b - 64b velikosti)

Existuje Large File System Support, kde ve 32b systemu se nahradi vsechny udaje kde se pracuje s velikosti vetsim datovym typem - podpora souboru *vetsich jak 2 GiB*.

linux:

du [soubor] vypise zabrane misto v blocich vc. rezie (metadat)
ls -l [soubor] vypise velikost souboru v bajtech (pouze uzitecna data)
df vypise volne misto na discich
ls -i [soubor] zpristupni cislo i-uzlu souboru
is -e /dev/... n - vypis i-uzlu n na /dev/...
dumpe2fs - zakladni informace o souborovem systemu ext2,3,4
/dev/zero je soubor typu zarizeni generujici proud 0
dd if=[source] of=[dest] je nizkourovnove kopirovani

6.2.5 Vyhody a nevyhody 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 prvniho nebo urciteho 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 zpusoby organizace souboru

6.3.1 Kontinualni ulozeni

- neboli spojite ulozeni souboru na disku,
- na disku je jeden spojity usek dat reprezentujici soubor,
- vyhodami jsou rychle vyhledani adresy 1./urciteho bloku nebo vyhledavani nasledujicich bloku,
- nevyhody: soubory nebude mozne jednoduse zvetsovat pokud budou prilis blizko u sebe (bude nutne jej presunout na jine volne a vetsi misto, pokud to pujde ci provest defragmentaci a pote zvetsit soubor)
- nepouziva se prilis (kvuli sve nevyhode)

6.3.2 Zretezene seznamy alokacnich bloku

- kazdy alokacni blok obsahuje sva (uzitecna) data a na konci obsahuje odkaz na nasledujici alokacni blok,
- vyhodami jsou rychly pristup na zacatek ci pruchod daty,
- nevyhodou je presun na nahodne misto v souboru - nutnost precist cely soubor az po dany blok (1 GiB soubor, chci posledni blok - musim precist cely),
- dalsi nevyhodou je rozprostreni metadat po celem disku - pri drobne chybe na disku prijdu o data (tedy i metadata, kde jsou odkazy na nasledujici bloky) a dojde k velke ztrate dat (vsechna data ”za”ztracenymi daty jsou nepristupna),
- neni prilis vhodna, nicmene se pouziva v souborovych systemech FAT

6.3.3 FAT

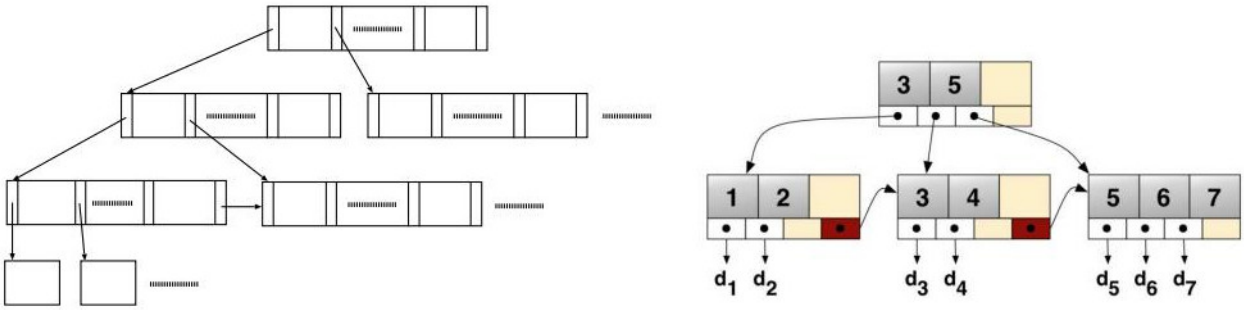
- File Allocation Table,
- od zretezenych seznamu se lisi tim, ze seznamy popisujici rozlozeni souboru na disku jsou ulozeny v separatni oblasti na disku (tzv. FAT),
- kde jsou tato data koncentrovana - rychleji se prohledavaji, lze vytvorit tak vice kopii FAT (prevence okamzite ztraty dat pri chybe),
- stale vznikaji problemy s rychlosti pri nahodnem pristupu (stale jde o zretezeny seznam)
- tabulka je pole, ktere obsahuje pro kazdy blok na disku 1 polozku, kazda polozka obsahuje odkaz na dalsi blok/polozku,
- pouziva se i dodnes (a je to velmi rozsirene), protoze je to jednoduche (napr. vestavene systemy)

6.3.4 B+ stromy

- jsou datovou strukturou prevzatou z databazovych systemu,
- maji dva typy uzlu - vnitrni a listove,
- vnitrni uzly jsou koren, jeho naslednici krome listovych, obsahuji odkaz na naslednika a vyhledavaci klic,
- listove uzly take obsahuji odkazy a vyhledavaci klice, odkazy vedou na data na disku, posledni odkaz na poslednim listu odkazuje na list na stejne urovni (jsou tak propojeny linearnim seznamem),
- pouzivaji se za ucelem popisu rozlozeni dat na disku (obsah souboru, pote vyhledavaci klic bude offset - cislo logickeho bloku v ramci souboru) nebo se pouzivaji pro adresare (klice budou jmena souboru) nebo pro popis celeho obsahu disku (klic je dvojice i-uzel a posuv souboru)

Vyhledavani v B+ stromu:

- pri hledani klice (k) se podivam, zda je klic mensi jak klic k0, pokud ano, pujdu nize, kde je k00, pokud ne, zjistim, jestli je klic mezi k0 a k1, pokud ano, jdu druhym smerem, opakuji po kn,
- pokud jsem niz, opakuji to same co vys az nedojdu k listovym uzlum,
- zde hledany klic najdu nebo zjistim ze v teto strukture klic neni
- pote mam odkaz na datovy blok,
- v pripade ze chci cist dal, tak jdu linearne po sobe po nasledujicich listovych uzlech



Obrázek 2: z prezentace IOS: Souborove systemy, slide 27 - B+ strom

Prace s B+ stromy:

- jsou zde limity jak moc/malo maji byt uzly zaplneny (strom se udrzuje vyvazeny) - pro uzly s m odkazy mame klice $0, 1,$ .. az $m - 2$ klicu (odkazu je o 1 mene nez klicu + cislovani od 0),
- pokud je strom tvoren solo korenem - nejmene muze mit 1 odkaz, maximalne $m - 1$ odkazu (posledni odkaz je pouzit jako ukoncovac seznamu listu),
- pokud to neni solo koren, tak ma nejake nasledniky, minimalne jich ma tak 2, maximalne m,
- vnitrni uzel ma tak $\frac{m}{2}$ (zaokr. nahoru) az m odkazu, list $\frac{m}{2} - 1$ (opet $\frac{m}{2}$ zaokr. nahoru) az $m - 1$ odkazu,
- vlozeni:
  - nejprve projdeme stromem od korene k listum,
  - najdeme kam chceme vlozit,
  - podivame se, zda ma list volny odkaz,
  - pokud ano - pouzijeme ho, pokud ne - list se rozstepi na 2 poloviny a podivam se o uroven vys, zda je mozne namisto 1 listu linkovat 2 listy,
  - pokud ano - prida se odkaz, pokud ne - nadrazeny uzel se musi rozstepit a postupovat o uroven vys,
  - ... stepi se strom az pripadne se rozstepi koren a strom bude mit 2 koreny
- ruseni:
  - se opet od listove urovne, tak, ze se zrusi odkaz v listu,
  - zkontroluje se, zda je uzel zaplneny v ramci danych limitu,
  - pokud ano - gut, pokud ne - podivam se na sousedni uzly a pokud se provest prerozdeleni tak, aby byly vsechny uzly naplneny v ramci limitu,
  - pokud se to nepodari, tak dojde ke slouceni listu,
  - posunu se o uroven vys, zrusim jeden odkaz, zkontroluji opet limity, zopakuji to same,
  - ... az se muze stat, ze se zrusi i koren

B+ stromy a jeho varianty jsou pouzivany pro popis diskoveho prostoru v filesystemech jako XFS, JFS, ZFS, Btrfs, ReFS, .. v omezene podobe tzv. stromu extentu v EXT4, podobna struktura je i v NTFS

definice:

solo koren = jediny koren

6.3.5 Extent

- pouziva se ke zrychleni prace s velkymi soubory,
- umoznuji zmensit objem metadat (je mozne rict, ze nektere alokacni bloky jsou ulozeny pospolu = vytvari extent), potom budu popisovat rozlozeni souboru po extentech (ne po alokacnich blocich),
- prinese lepe vyvazene indexove struktury,
- rychlejsi mazani,
- jsou pouzity snad ve vsech systemech s B+ stromy,
- B+ strom se snadno kombinuje s extenty (neplati pro klacicky Unixovy strom - protoze ve stromu jsou explicitne ulozene vyhledavaci klice, ale Unixovy nema v zadnych strukturach [i-uzel] ulozenou velikost datovych bloku [protoze jsou vsechny stejne a konstantni])

Pokud pouzivame B+ stromy, tak rychlemu spojitemu pruchodu pomaha listova uroven, pokud je prolinkovani listu pouzito. Pro male soubory muze predstavovat B+ strom zbytecnou rezii (data se bud ulozi primo v i-uzlu nebo nebo z nej mame prime odkazy na extenty z i-uzlu [do max. 4 extentu])

definice:

extent je jednotka vystavena na bloky, posloupnost promenneho poctu alokacnich bloku (jdoucich 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 clustero, 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

- podporovana ZFS, NTFS, Btrfs, XFS, (ext4 ne) ..
- snazi se odhalit opakovane ukladani stejnych dat na disk a ulozi je pouze jednou a odkazuje se na ne vicenasobne.
- systemy s deduplikaci se snazi takova data detekovan (sekvence bitu, bloku, extentu, ..)
- zalozeno na kryptografickem hashovani (hledaji se data se stejnym popisem, urci se shoda),
- muze byt realizovano pri zapisu nebo dodatecne (zadost uzivatele),
- muze usporit diskovy prostor (pri virtualizaci, na mail serverech, repozitare, ..), pametovy prostor i cas (zamezi se opakovanemu cteni i zapisu),
- pri mensim objemu duplikace muze naopak zvysit spotrebu CPU casu, spotrebu pametoveho i diskoveho prostoru

Rozdil oproti copy-on-write:

- copy-on-write se muze uchytit (klony, snimky) pouze tehdy, pokud duplikaty vzniknou cinnosti samotneho filesystemu (napr. vytvoreni virtualek),
- zatimco deduplikace aktivne vyhledava duplikaty (napr. uzivatel, co stahuje stejne reklamni letaky)

6.8 Typy souboru v UNIXu

- - je obycejny soubor,
- d adresar,
- b blokovy specialni soubor,
- c znakovy specialni soubor,
- l symbolicky odkaz (symlink),
- p pojmenovana roura,
- s socket

definice:

specialni soubor je typ souboru reprezentujici hardwarove zarizeni (disk, hw, pamet) se kterym se komunikuje po blocich nebo znacich
symbolicky odkaz je soubor obsahujici jmeno jineho souboru (odkazuje na nej)

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)

Pripojovaci 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, ktery
- 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 stejnymi jmeny vznikaji kolize, ty se resi napr. preddefinovanim priorit pripojovanych filesystemu a zpristupni se soubor daneho jmena z logickeho disku, ktery ma nejvetsi prioritu
- UnionFS ma copy-on-write semantiku, coz umoznuje emulaci prepisovani neprepisovatelnych medii (v 1 vetvi CD, neprepisovatelne, 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

7

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 cyklkus (jeden odkazuje na druhy a druhy na prvni) - v systemy je preddefinovany maximalni pocet na sebe odkazujicich symlinku (pri prekroceni 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 nezleve do (i-uzlu), alokuji se normalne alokacni bloky na disku = pomaly symlink

linux:

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

7.2 Blokove a znakove specialni soubory

Soubory reprezentujici rozhrani souboroveho systemu k fyzickym (opravdovy hw) ci virtualnim zarizenim (xterminaly, ..). Souborovy system vyvtari souborove rozhrani, tim umoznuje tyto soubory pri urcitych operacich identifikovat (jmeno soubory, napr. /dev/sdX), s celym zarizenim lze take pracovat jako se souborem.

Typicky zarizeni sidli v adresari /dev

Bezne typy zarizeni:

- /dev/hda - (drive)oznaceni pro prvni fyzicky disk na prvnim ATA/PATA rozhrani
- /dev/hda1 - (drive) prvni logicky disk na hda
- /dev/sda - prvni fyzicky disk SCSI, navic i disky SATA/PATA (jadro nad temito disky emuluje SCSI)
- /dev/mem - obsah pameti (RAM)
- /dev/zero - nekonecny zdroj 0 bajtu
- /dev/null - soubor typu cerna dira - cokoli se do nej zapise, do se zahodi (presmerovani vystupu programu tak, aby nas neotravoval), pri cteni se tvari jako prazdny soubor
- /dev/random - generator nahodnych cisel
- /dev/tty - terminal
- /dev/lp0 - tiskarny
- /dev/mouse - mys
- /dev/dsp - zvukova karta
- /dev/loop - zarizeni typu (smycka) loop mi umoznuje pripojit soubor jako disk (obraz souboroveho systemu) k adresari, jakoby se jednalo o novy fyzicky disk

Tato oznaceni zavisi na pouzitem systemu (Linux, distribuce, ..). Vyhoda zavedeni specialnich souboru je, ze umoznuji identifikovat zarizeni, se kterymi chci 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 systeu v /etc/group,
- maji sva GID - cislo identifikujici skupinu uzivatelu,
- 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 vytvareni 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, ktery pokud bude prirazen nejakemu adresari, tak se vytvori adresar, ve kterem i pres pravo cteni a zapisu souboru mohou uzivatele rusit pouze ty soubory, ktere sami vytvorili. (typicky adresar /tmp) (tldr: uzivatel muze mazat, ale pouze to co vlastni)

7.6 SUID, SGID

S procesy jsou spojeny identifikatory jako jsou:

- UID - realna identifikace uzivatele (cislo uzivatele, ktery 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, ktery proces spustil a EUID = identifikace vlastnika (= ktery prava pujcil). Pokud budou prava propujcena (pouzito SUID), misto x se vypise s, pokud tam x neni, vypise se S.

7.7 Typicka 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 zavadec systemu (obrazy jadra, pocatecni fs)
- /dev - specialni soubory - rozhrani zarizeni
- /etc - konfiguracni soubory pro system i aplikace
- /home - domovske adresare uzivatelu
- /lib - sdielne knihovny a moduly jadra
- /media - pripojny bod pro prenosna zarizeni
- /mnt - pripojny bod pro docasne filesystemy
- /proc - informace o procesech a jadru
- /root - domovsky adresar superuzivatele
- /run - docasne informace o bezicim systemu (demony)
- /sbin - programy pro superuzivatele (nutne pro bootovani, ne vse je spustitelne superuzivatelem)
- /sys - informace o jadru, zarizenich, modulech, ..
- /tmp - docasne pracovni soubory (obsah se maze pri restartu)
- /usr - obsahuje dale adresare a
  - soubory, ktere nejsou nutne pri zavadeni systemu, struktura je zde podobna jako u / :
  - bin, sbin, lib,
  - include (hlavickove soubory),
  - share (soubory je mozne sdilet nezavisle na architekture),
  - local (koren dalsi hierarchie urcena pro lokalni nestandardni instalace programu),
  - src (zdrojove texty jadra)
- /var - soubory menici se za behu systemu
  - obsahuje log (zaznamy o cinnosti systemu),
  - spool (pomocne soubory pro tisk),
  - mail (postovni prihradky uzivatelu)

Nove tema: Datove struktury a algoritmy pro vstup/vystup

7.8 Pouziti vyrovnavacich pameti

Cilem pouziti cache (vyrovnavacich pameti) je minimimalizace poctu pomalych operaci s periferiemi (disky). Hierarchie: kolekce, sbirka dilcich vyrovnavacich 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:

- zjisti se, zda je blok v pameti,
- pokud ne, naalokuje se novy blok, muze se vyuzit jiz nejaky systemem predalokovany a nevyuzity,
- nactou se data z disku, presunou se do vyrovnavaci pameti,
- vyrovnavaci pamet je v prostoru jadra (bezne procesy zde nemaji pristup),
- vykousne se z nactenych dat tu cast, o kterou ma uzivatel/proces zajem,
- nakopiruje se to do adresoveho prostoru uzivatelskeho prostoru

Pri dalsim cteni:

- nejprve se opet vyhleda, zda je blok v pameti,
- pokud ano,
- nebude se cist z disku, pouze se z alokacniho bloku vykousne cast, o kterou ma uzivatel/proces zajem,
- tato cast se uzivateli preda

7.9.2 Zapis

Postup pri zapisu:

- nejprve se zjisti, zda je blok v pameti,
- pokud ne, prideli se vyrovnavaci pamet,
- nactou se data z disku do vyrovavaci pameti,
- jadro prevezme od procesu, ktery chce zapisovat data, ktera chce zapsat,
- prepise jimi danou cast alokacniho bloku, dirty bit se zmeni (0 na 1),
- operace konci (neprovede se zapis na disk),
- casem se provede zpozdeny zapis na disk a vynuluje se dirty bit

System sam od sebe s periodou prepisuje cache na disky, lze si to vynutit pomoci sync ci fsync.

Pokud je znamo, ze se prepise cely alokacni blok (nebo se jedna o novy blok), buffer se vynuluje a nenacitaji se data z disku do cache.

definice:

dirty bit je indikator toho, jestli jsou data cache sladena s obsahem na disku (0 - data v cache = disk, 1 - data cache != data disk - nuluje se zpozdenym zapisem)

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 nacitaji i-uzly vsech 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 = vznika 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,
- zjisti, ze i-uzel uz tam je,
- nebude se znovu i-uzel nacitat 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 mozne pridavat i dalsi identifikatory, napr.:

- priznak, ze ma byt soubor vytvoren pokud neexistuje,
- pokud existuje, ma byt zkracen na 0,
- otevrit v rezimu pridavani (kdekoli je aktualne ukazovatko v souboru, tak v pripade zapisu se automaticky posune na konec a tam se prida),
- synchronni zapis (operace zapisu skonci az tehdy, kdyz se data zapisou opravdu na disk)

Pri chybe:

- open vraci -1,
- nastavi se chybovy kod, ktery blize popisuje co se stalo (do knihovni promenne errno),
- existuji standardni chybove kody,
- lze pouzit standardni knihovni funkci perror

linux:

fd = open([jmeno souboru], [rezim]); otevre soubor
V-uzly je tabulka i-uzlu filesystemu VFS
tabulka popisovacu je pole s radky cislovanymi od 0 (0 - stdin, 1 - stdout, 2 - stderr)
tabulka procesu v jadre je cast adresoveho prostoru, ve kterem ma jadro ulozene pomocne informace k procesum a ma sem pristup pouze jadro

7.9.4 Cteni a zapis z/do souboru

Cteni:

- zkontroluje se platnost popisovace (otevreni popisovace, soubor pro cteni),
- pokud se jedna o prvni pristup, naalokuje se cache, nactou se data do cache a z cache se prislusna data pouziji,
- pokud uz jsou data v cache, nactou se odtud,
- predani se deje pozadovanych z cache (RAM, jadro) do pole (RAM, cache adresoroveho prostoru procesu),
- funkce vraci pocet opravdu prectenych bajtu nebo -1 pri chybe (+ nastavi errno).

Zapis:

- funguje podobne jako read,
- pred vlastnim zapisem kontroluje dostupnost diskoveho prostoru a tento prostor alokuje (rezervuje),
- vraci pocet opravdu zapsanych bajtu 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 posunou 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, ktery 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 nezpusobi ulozeni obsahu jeho vyrovnavaci pameti na disk.

linux:

close([popisovac souboru]) zavre 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 vyrovnavacich 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 neprimo vytvarenim/modifikacemi souboru

7.9.11 Blokove a znakove specialni soubory

Predstavuji rozhrani k blokovym / znakovym zarizenim (/dev/..., viz. 7.2)

- lze je vytvorit pomoci mknod,
- typicky tyto soubory vytvari jadro ci demoni (udev, devd - pri pripojeni zarizeni se vytvori automaticky prislusny soubor)

Pri pouziti beznych souborovych operaci jadro mapuje operace na odpovidajici podprogramy, ktere ty operace implementuji pro dany typ zarizeni s vyuzitim **tabulek**:

- znakovych zarizeni,
- blokovych zarizeni

Tyto tabulky obsahuji ukazatele na funkce implementujici prislusne operace v ovladacich danych zarizeni.

Specialni soubory na disku zabiraji **pouze i-uzel, krome beznych udaji maji v i-uzlu typ souboru a 2 udaje**:

- hlavni cislo,
  - major number,
  - udava typ zarizeni
  - odkazuje do tabulky zarizeni (hlavni cislo = n-ty radek tabulky),
- vedlejsi cislo
  - minor number,
  - udava instanci zarizeni
  - pouziva se jako parametr pri volani urcite operace - parametr funkce ovladace (cislo = ktere zarizeni se ma presne pouzit)
- typ souboru urcuje tabulku (blok, znak.)

linux:

mknod vytvori specialni soubory
ovladac je sada podprogramu pro rizeni urciteho typu zarizeni (nebo viz xx nebo viz xx)

7.10 Terminaly

Jsou fyzicka ci logicka zarizeni umoznujici (primarne) textovy vstup a vystup systemu (po radcich), editace vstupniho radku ci specialni znaky (Ctrl+C SIGINT, Ctrl-D konec vstupu, ..)

Rozhrani:

- /dev/tty - pro kazdy proces, ktery ma ridici terminal, odkazuje na jeho ridici terminal
- /dev/ttyS1 - fyzicke terminaly na seriove lince,
- /dev/tty1 - virtualni terminaly (konzole),
- pseudoterminaly (/dev/ptmx - master, /dev/pts/1,..) tvoreny dvojici master / slave, po kazdem otevrenim se vytvori novy slave - emuluje komunikaci pres seriovou linku (umoznuje pro propojeni urcitych casti, napr. SSH - propojeni klienta se vzdalenym klientem)

Ruzne rezimy zpracovani znaku (radkove discipliny - line discipline):

- raw - neprovadi se zpracovani znaku,
- cooked - zpracovani vsech ridicich znaku,
- cbreak - provadi zpracovani maleho poctu znaku (ctrl+c, mazani, ..)

Nastaveni rezimu zpracovani znaku je mozne pomoci stty. Dale je mozne nastavit **rezim terminalu**:

- prikazy tset, tput, reset,..
- promennou TERM, ve ktere ulozen aktualni typ terminalu,
- typy terminalu (prikazy terminfo, termcap)

Tyto prikazy komunikuji s terminalem pomoci *escape sekvenci*. Knihovna curses je standardni knihovna pro rizeni terminalu ci tvorbu aplikaci s terminalovym uzivatelskym rozhranim.

definice:

escape sekvence jsou sekvence znaku escape, prikaz [parametry], escape, prikaz, ..

7.11 Roury

Jsou prostredkem meziprocesove 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 — p2 — p3 - na presmerovani se pouzivaji nepojmenovane roury)

Pojmenovane roury

- vyvtari se pomoci mknod is mkfifo,
- existuji v souborovem systemu,
- mohou se zavrit, otevrit, apod.

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

definice:

pribuzne procesy - pokud jeden proces otevre 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 - zacinam 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)

Sokety 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. Sokety taky maji vyhodu, ze je mozne vytvorit aplikace, ktere mohou bezet distribuovane v siti.

definice:

blokujici rezim (I/O) - pokud chci nacitat 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, ktery 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, ktere krome dat i-uzlu obsahuji 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. Zpristupnuje soubory ulozene na vzdalenych systemech.

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

- klient pozada o cteni ze souboru (napr.),
- vsechny operace prochazi pres VFS,
- pozadavek se z VFS preda na NFS klienta,
- NFS klient preda pozadavek na NFS server,
- NFS server pracuje s lokalnim filesystemem jiz na vzdalenem PC,
- ten pracuje take s VFS (ale na serveru), prostrednictvim nej ziska data s lokalniho filesystemu,
- data pote putuji zpet pres sit k uzivateli zpet

Umoznuje **kaskadovani** - je mozne si lokalne do jednoho adresaroveho stromu pripojit vzdaleny strom (a do nej pripojit dalsi vzdaleny filesystem). **Autentizujeme se nejcasteji pres UID, GID** (musi existovat duvera mezi spravcem lokalniho a vzdaleneho systemu). Nebo se pouziva jine mechanismy (kryptografie, ..)

NFS verze 3:

- starsi, bezestavova verze - nepouziva operace otevirani, uzavirani soubori, kazda operace si musi nest veskere informace o souboru,
- na strane klienta nema cache (slozita implementace), na strane serveru cache,
- nema podporu zamkyani (operace pro zamknuti zaznamu souboru jsou prazdne)

NFS verze 4:

- stavova,
- cache na strane klienta,
- podpora zamykani

8.4 Spooling

Simulatenous perpiheral operation on-line (simulatenalni online provadeni perifernich operaci). Jedna se o **provadeni online bez cekani periferni operace (vystup) na periferiich, ktere nemusi online prokladani dat od ruznych procesu ci uzivatelu podporovat**. (napr. sitova tiskarna - spousta uivatelu, kazdy chce, aby tisk se provedl okamzite)

Vystup se provede do vyrovnavaci pameti spool (soubor), system si vede frontu cekajicich uloh, operaci, do fronty se zaradi odkaz na vytvoreny soubor, uloha se ma dokoncim po uvolneni periferie.

linux:

/var/spool obsahuje soubory spool

Nove tema: Sprava procesu.

Sprava procesu (process management) zahrnuje:

- prepinani kontextu (dispatcher, vzdy v rezimu jadra),
- planovac (nemusi byt v jadre),
- spravu pameti,
- podporu meziprocesove komunikace (signaly, roury, sockety, synchronizace - semafory, mutexy, ..)

definice:

prepinani kontextu rozumime fyzicky odebirani procesoru jednomu procesu a pridelovani jimenu procesu (take viz 1.3)
planovac rozhoduje, ktery proces ci procesy pobezi a pripadne jak dlouho

8.5 Proces

Definice viz 1.2. Proces je bezici program, tedy aktivni entita, abstrakce aktivity probihajici v systemu. Program je naopak pasivni entita (definice viz 1.2).

Proces je v OS definovan:

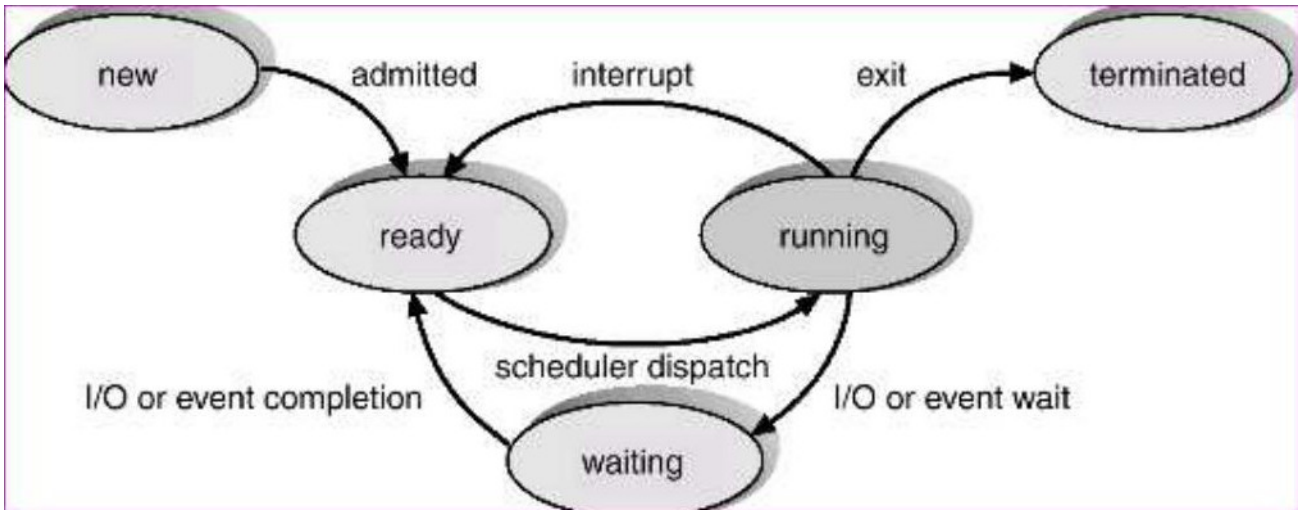
- unikatnim identifikatorem (PID - process identifier),
- stavem planovani,
- ridicim programem,
- obsahem registru (beznych - EAX, BX, EIP, ..),
- zasobnikem (aktivacni zaznamy - informace o rozpracovanych funkcich),
- daty (staticka ne/inicializovana data, hromady, individualne alokovane pameti),
- tim, jake dalsi vazby a zdroje OS vyuziva (jake soubory ma aktualne otevreny, signaly, obsluzne funkce signalu, PPID, UID, GID, semafory, sdilena pamet, sdilene knihovny, ..)

8.6 Stavy planovani a jejich zmeny

Nejzakladnejsi planovaci diagram (vetsiny/vsech ruznych OS) - **stavy procesu**.

Stavy planovani procesu (obecne):

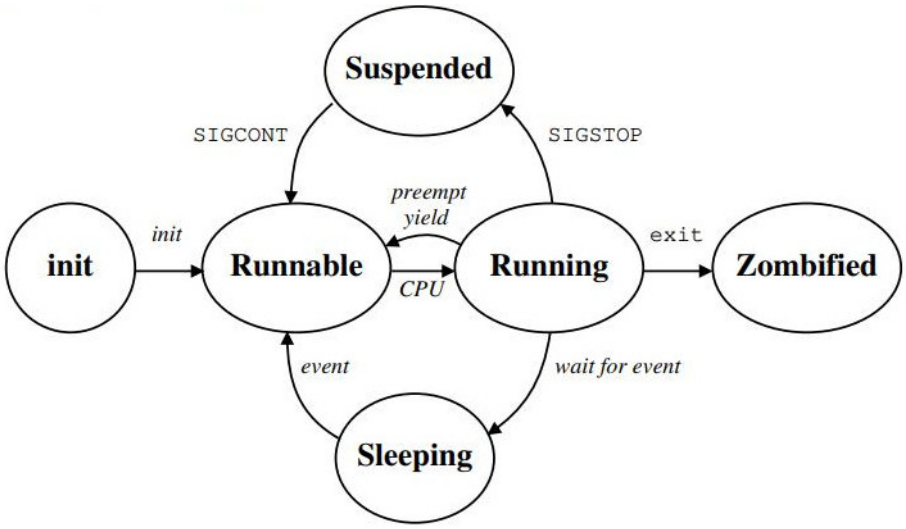
- new - proces je inicializovan (vytvari se struktury co jej popisuji, data, proces pripadne ceka ve vstupni fronte dlouhodobeho planovace, ..)
- ready - proces ceka na kratkodoby planovac na prideleni procesoru, dispatcher provede prepnuti, prepne se do running
- running - proces muze byt preruen (preemtivni planovani - opet stav ready), muze pozadat o sluzbu jadro (I/O operace, sync) - stav waiting,
- waiting - proces ceka na dokonceni operace (sluzba jadra), pote pujde do stavu ready,
- terminated - proces skonci (proces se v systemu nejakou dobu vyskytuje ve stavu ukonceny)



Obrázek 4: z prezentace IOS: Sprava procesu - stavovy diagram obecneho planovani 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

Prvni soucast pameti vyuzite procesem je **uzivatelsky adresovy prostor** (user address space) - je pristupny procesu (muze z teto casti pameti cist/psat), obsahuje:

- kod (ktery je rizen, code area/text segment),
- data (ne/inicializovana, hromada, alokovana pamet),
- zasobnik,
- soukroma data sdilenych knihoven, sdilene knihovny ci sdilena pamet

Dalsi cast informaci o procesech byva v nekterych pripadech - ne vzdy umistena v **uzivatelske oblasti** (napr. linux tento koncept nepouziva - vse ma v tabulce procesu):

- ulozena pro kazdy proces v casti uzivatelskeho adresoveho prostoru, ktera neni pristupna procesu ,
- je to pristupne jadru,
- u kazdeho procesu si v teto casti uklada informace o procesu je tam:
- cast PCB, ktera je pouzivana zejmena za behu procesu,
- PID, UID, EID, GID, EGID, PPID (identifikator rodice),
- obsah registru,
- deskriptory souboru (informace o tom, ktere otevreni souboru reprezentuje stdin, stdout),
- obsluzne funkce signalu (funkce, ktere se budou volat pro obsluhu signalu),
- uctovani,
- pracovni, korenovy adresar

Dalsi zaznamy jsou v tabulce procesu:

- ulozeno trvale v jadru,
- informace o procesu, ktere jsou dulezite i kdyz proces nebezi:
- PID, PPID, UID, EID, ...,
- stav planovani,
- udalost, na kterou proces ceka,
- planovaci informace (pro planovac pri rozhodovani ktery proces dal pobezi ci ne - priorita, spotreba casu, ..),
- cekajici signaly (signaly, kt. mohou prijit, i kdyz proces nebezi),
- odkaz na tabulky dat reprezentujici rozlozeni procesu, dat, kodu, zasobniku, ..

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 vsech 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, ktery dany proces vytvoril, rodic),
- realny-skutecny uzivatel, skupina uzivatelu, ktery proces spusitl - 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, ktere ziskal (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, ktere se **zavola jednou**, pokud nedojde k chybe, tak **skonci 2x**. Na zaklade volani fork vznika **vztah rodic-potomek (parent-child) a hierarchie procesu**. Vysledkem forku je totiz **duplikace procesu**. Vznika takrka identicka kopie potomka, ktery 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 nektere 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

Prvnim procesem, ktery vznika a vyvtvari ho jadro je **proces init s PID=1** (aktualni novejsi implementace je pojmenovana jako systemd - prikaz pstree). Tento proces je **predkem vsech ostatnich uzivatelskych procesu**. **PPID initu je 0**.

Existji take procesy jadra (kernel threads/processes), jejich init predkem neni:

- jejich kod je soucasti jadra,
- vyskytuje se i proces s PID=0, vznika uplne jako prvni, podili se na inicializaci jadra, nasledne se meni na swapper (pokud je na systemu pouzit) nebo na cekaci smycku ci je to pouzivano jako procesova obalka pro vlakna jadra (na linuxu se tento proces nevypisuje)

Init se podili na **inicializaci systemu**, pote **prebira navratove kody procesu, ktere skonci, ale ktere osirely driv nez skoncily** (tedy rodic skonci driv nez potomek) - teoreticky by byl zombie procesem do nekonecna, proto init prebira jeho navratovy kod a umozni mu odchod ze systemu.

definice:

swapper je proces, ktery slouzi k tomu, ze v pripade akutniho nedostatku pameti nektere procesy pozastavi a zcela ulozi na disk (veskere casti pameti zabirane procesem)

kthread je proces, na linuxu init pro procesy jadra, ma PID=2

linux:

pstree vypis stromu procesu

8.12 Zmena programu – exec

Umoznuje v ramci existujiciho procesu vymenit ”jeho vnitrnosti”- **zahodit existujici kod a nahradit ho kodem jinym**. Exec je funkce, ktera se **zavola jednou a neskonci vubec** (je v nejakem kodu, ten kod prestane bezet) pokud nedojde k chybe.

Pokud v procesu zavolam exec:

- porad dedi radu rysu sveho predka,
- zustava mu rada zdroju a vazeb OS (identifikatory, otevrene soubory, ..),
- zanikaji vazby a zdroje vazane na puvodni ridici (obsluzne funkce signalu, sdilena pamet, pametove mapovane soubory, semafory).

Skupina funkci exec:

- execve (zakladni volani), execl, execlp, execle, execv, ..

Pozn.: Ve Windows se procesy vytvari volanim CreateProcess(...), ktere zahrnuje funkcnost fort 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, ktery skoncil (pripadne -1 pokud prijde signal, ktery 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, spostu 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 hadrware, pripadnou inicializaci hw

8.15 Urovne behu

System urovni behu byl zaveden jiz v UNIX System V. Rozlisuji se urovne behu 0-6: (nektere mely predpripraveny standardni vyznam, nektere 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 poradi danem cislem za tim 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 Planovani procesu

Procesy planuje planovac.

Rozlisujeme 2 **planovaci algoritmy** (definice viz. 1.5):

- nepreemtivni planovani (typicky I/O operace, konec - volani exit, vzda se CPU - yield),
- preemtivni planovani (typicky preruseni od casovace, muze jit i o jine, treba od disku)

Rozlisujeme **3 "typy"planovani**:

- dlouhodobe planovani - ktere ulohy budou pripusteny do systemu,
- strednedobe planovani - procesy maji pamet / nemaji pamet - jedna se o system swapovani,
- kratkodobe planovani - procesy maji pamet - prepinani mezi ulohami

System swapovani (= strednedobe planovani):

- v pripade nedostatku pameti nektere procesy pozastavi, odebere jim veskerou pamet a ulozi je na disk,
- tyto procesy jsou vyrazeny z planovani (nemaji pamet - nemohou bezet),
- pri zadosti o spusteni ulohy uloha pak nebude spustena ihned, ale system ceka na uvolneni systemovych zdroju (sluzba ceka ve fronte) - pote uz se jedna dlouhodobe planovani (rozhoduje se o tom, ktere ulohy budou vubec pripusteny do systemu)

definice:

planovac rozhoduje ktery proces ci procesy pobezi a pripadne jak dlouho
systemy s neepremtivnim planovanim = systemy s kooperovanym planovanim (procesy musi spolupracovat, kooperovat)

8.17 Prepnuti kontextu (procesu)

Prepnuti kontextu na prikladu - dispecer na zaklade rozhodnuti planovace prepina mezi procesem A a B:

- bude muset uchovat stav registru (nekterych, ale vcetne ridicich registru) procesu A do PCB (nebo task struct v linuxu),
- dojde k uprave nekterych ridicich struktur v jadre (uprava planovacich struktur, uctovacich struktur, ..),
- obnova ulozenych hodnot registru procesu B,
- dojde k predani rizeni proceu na adresu, kde bylo drive preruseno provadeni procesu B,
- tato akce se musi provadet v rezimu jadra

Neuklada se a neobnovuje cely stav procesu (pri pozastaveni procesu neni nutne na disk ukladat obsah pameti, ukladaji se pouze registry).

Presto prepnuti muze trvat **radove stovky/tisice instrukci** - jadra umoznuji interval v jakem prichazeji preruseni z casovace - je treba si dat pozor, aby ten interval nebyl prilis kratky, protoze zacne pote prevazovat rezie systemu nad uzitecnym behem (neustale ukladani a obnovovani 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, ktery nove vznikne nebo je uvolnen z cekani na nejake operaci (I/O, sync, ..), pripadne proces, ktery se vzda CPU,
- se zaradi na konec fronty,
- procesy, ktere pobezi se vybiraji ze zacatku fronty,
- jedna se o nepreemtivni algoritmus (prepnuti kontextu dojde pokud se bezici proces vzda CPU ci zavola sluzbu jadra)

9.1.2 Round-robin

- preemtivni 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, ktery aktualne deklaruje nejkratsi dobu pro svuj dalsi beh na CPU, po ktery 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,..),
- nepreemtivni 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 (specializovancyh) systemech,

- nevyhodou algoritmus je starnuti (hladoveni, starvation) - ke starnuti dochazi pri cekani na nejake zdroje (CPU, zamek, ..) je situace, kdy nektery proces, ktery o ten zdroj zada, na nej ceka bez zaruky, ze nej nekdy ziska,
- pokud nejaky proces deklaruje delku CPU burst a v systemu budou neustale kratsi procesy s touto delkou, tyto procesy ho budou neustale predbihat (nikdy se tak k CPU nedostane)

9.1.4 SRT

- shortest remaining time,
- preemtivni obdoba SJF,
- je zde prevence pri vzniku ci uvolneni procesu z cekani (kdyz se uvolni novy proces a deklaruje kratsi vypocetni fazi nez ten, ktery dosud bezel - muze byt provaden on),

9.1.5 Viceurovnove planovani

- procesy rozdeleny do ruznych skupin (typicky dle priority, ale ne nutne - napr. dle typu procesu),
- kazda skupina procesu muze pouzivat jiny dilci planovaci algoritmus (FSFS, round-robin, SJF, ..) s ruznymi parametry,
- krome toho mame dalsi (”hlavni”) algoritmus, ktery rozhoduje, ktera skupina procesu dostane CPU cas - casto jednoduse na zaklade priorit skupin,
- pote je dalsi planovaci algoritmus, ktery planuje mezi skupinami

9.1.6 Viucerovnove planovani se zpetnou vazbou

- skupiny procesy jsou rozdeleny dle priorit,
- proces, ktery se stane nove pripravenym bezet (nove vznikne, je uvolnen z cekani, ..) je zarazen do skupiny procesu s nejvyssi prioritou.
- v teto skupine bezi a postupne klesa do nizsich priorit,
- az spadne do nejnizsi urovne (planovan round-robin),
- pouzivaji se varianty, kdy proces ma prednastavenou statickou prioritou a zaradi se do planovaci urovne teto priority, a pote ma i dynamickou prioritu, ktera se muze zvysovat i snizovat, typicky se priority meni tak, ze pokud nejaky proces spotrebovava mnoho CPU casu - priorita se snizi, proces ceka na mnoho I/O operaci - priorita se zvysi,
- cilem je zajistit rychlou reakci interaktivnich procesu,

definice:

interaktivni procesy jsou procesy komunikujici s uzivatelem

9.2 Planovac v Linuxu (od verze 2.6.23)

Pouziva se **viceurovnove 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 (prezvato z RT OS)

definice:

FCFS s preemci na zaklade priorit - pokud behem behu procesu dobehne I/O operace ci sync operace, ktera 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, ktere bezi, maji periodicke vypocetni faze (ocekavana faze - zname dobu jak dlouho trvaji + mame casovy limit, dokdy se maji vypocty provest), ktere 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, stejna priorita - vsichni 25 procent CPU),
- u kazdeho procesu si vede udaje o tom, kolik virtualniho CPU casu uz ten proces na CPU stravil,
- 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 stravil nejmene casu na CPU,
- proces necha bezet po casove kvantum, ktere 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 spravedlive 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 uzivatel 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 sdroji (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 strednepriorit ni procesy, ktere tento zdroj nepotrebuji, ty budou predbihat dale nizkoprioritni proces,
- vysokoprioritni proces musi cekat, zatimco strednepriorit ni a nizkoprioritni maji najednou ”vyssi prioritu”,
- tento jev muze a nemusi vadit - muze zpusobit snizenou odezvu systemu, nicmene muzou se zablokovat 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 dedi prioritu cekajiciho procesu (s nejvyssi prioritou),
- na jednoprocesorovych systemech se pouziva technika, kdy po dobu behu v kriticke sekci se zakaze preruseni

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 vsech pametovych jednotek (za ruznou dobu)

9.6 Vlakna, ulohy, skupiny procesu

Vlakna, 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, sezeni

Uloha (job) se poji se shellem, je skupina skupina paralelne bezicich procesu spustenych jednim prikazem, prikazy propojene do kolony (p1 — p2 — p3 - pipeline).

Skupina procesu (process group) v UNIXu:

- mnozina procesu paralelne bezicich, se kterymi je mozne provadet operace jako s celkem,
- skupine je mozne poslat signal jako 1 jednotce,
- predek take muze cekat na libovolneho potomka z urcite skupiny,
- kazdy proces prave v 1 skupine procesu, po vytvoreni vzdy je to skupina jeho predka,
- skupina muze a nemusi mit vedouciho - jeji prvni proces, dokud neskonci (pokud skonci - skupina bez vedouciho),
- skupina je identifikovana vedoucim skupiny, pokud vedouci skupiny skonci, neni mozne jeho cislo recyklovat a pouzit pro ID skupiny

Sezeni v UNIXu:

- mnozina skupin procesu,
- kazda skupina procesu je v jednom sezeni,
- sezeni muze a nemusi mit vedouciho,
- muze mit ridici terminal (/dev/tty),
- v ramci sezeni plati, ze jedna skupina je na popredi (cte z terminalu), ostatni jsou na pozadi,
- pokud terminal konci, signalem je SIGHUP, informovan je vedouci sezeni (typicky shell), standardtne se vsem procesum, na ktere nebyl uzit prikaz nohup/disown, posle navic SIGHUP, pokud jsou procesy pozastaveny, tak posle signal 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 (aritemticka chyba, chyby sbernic, ..),
- externich udalostech (dostupnost I/O, vyprseni casovace, ..),
- na zadost procesu - IPC (meziprocesova komunikace, procesy si mohou navzajem posilat singnaly, ale jadru signal nelze zaslat - jadro neni 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 **ukonceni procesu** (pripadne s generovani core dump), **ignorovani signalu, zmrazeni ci rozmrazeni procesu**.

Predefinovat obsluhu lze u vsech signalu **mimo SIGKILL, SIGSTOP**. U SIGCONT vzdy dojde k odblokovani procesu (a nasledne se provede preddefinovana 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 vytvareni masek je mozne pouzit preddefinovana makra sigemptyset, sigfillset, segaddset, 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, ktery zasilame). Umoznuje zasilat signaly:

- jednomu konkrektnimu procesu (pid kladne),
- skupine procesu (0 - ve skupine, ve ktere 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, ktere 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 zjisti se, ze neprisel - zacne se cekat a prijde mezi testem a zacatkem cekani - cekani do nekonecna)

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, ktera muze vzniknout pri pristupu ke sdilenym zdrojum, datum - vznika pri pristupu ke zdilenym 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 vylouceni - nanejvys 1 (ci k) procesu muze do kriticke sekce vstouput (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 vstouput (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 vznikajici na kriticke sekci

Data race - **dochazi k zavodu mezi daty** - situace kdy jsou treba dva pristupy ke zdroji s vylucnym pristupem ze dvou procesu bez synchronizace, alespon jeden pristup je zapis. Jedna se o:

- Uvaznuti (deadlock) pri pristupu ke zdrojum s vylucnym (omezenym) pristupem rozumime situaci, kdy kazdy proces z urcite mnoziny procesu je pozastaven a ceka na uvolneni zdroje s vylucnym (omezenym) pristupem vlastneneho nejakym procesem z dane mnoziny, ktery jediny muze teto zdroj uvolnit,
- Livelock - specialni pripad starnuti - dochazi k tomu s tim, ze procesy vykonavaji nejakou cinnost, ale nikdy se nedostanou k tomu, aby udelali co chteli (nikdy nedojde ke vstupu do kriticke sekce),
- Blokovani pri pristupu do kriticke sekce je situace, kdy proces, jenz zada o vstup do kriticke sekce musi cekat, prestoze je prestoze je kriticka sekce volna a ani o zadnou z dane mnoziny sdilenych kritickych sekci zadny dalsi proces nezada,
- Starnuti je situace, kdy proces ceka na podminku, ktera nemusi nastat - v pripade kriticke sekce je touto podminkou umozneni vstupu do kriticke sekce

10.4 Zpusoby reseni problemu kriticke sekce

Musi dojit k **vzajemnemu vylouceni** (nesmi dojit k data race) a musi zajistovat **dostupnost kriticke sekce** (minimalne nesmi dojit k deadlocku) a zaroven synchronizacni prostredky musi byt efektivni.

Petersonuv algoritmus (kratky popis):

- omezen pro 2 procesy (existuji i rozsireni pro vice procesu),
- resi sync kritiske sekce bez hardwaru,
- nebere ohled na to, jaka ta kriticka sekce je,
- v kriticke sekci muze byt maximalne 1 proces,
- procesy o sobe vzajemne nevi (nevi co druhy proces dela, pouze vedi, ze druhy proce existuje),
- pracuje s sdilenymi polemi booleovskych promennych (init false, false) a sdilenou promennou turn = 0,
- pole nam rika, ktery z procesu chce pristoupit do kriticke sekce,
- turn nam rika, kdo je zrovna na tahu (pred vstupem do kriticke sekce),
- prvni proces chce pristoupit do kriticke sekce - nastavi turn na 1 - i (i je index v bool poli), na tahu je druhy proces,
- zacne se provadet prazdny cyklus (dokud je nastaven flag 1 - i a zaroven je na tahu ten druhy proces - cykli),
- proces 1 provede kritickou sekcu a nastavi svuj bool na false

Bakery algoritmus L. Lamporta:

- vylouceni pro n procesu (n znam dopredu),
- prijdes do pekarstvi (nebo spis na postu), vezmete si listek s cislem a cekas, az na tebe dojde rada,
- pred vstupem do kriticke sekce ziiska proces pristupovy listek, jehoz hodnota je vetsi nez cisla pridelena jiz cekajicim procesum,

- pracuje s sdilenym booleovskym pole priznaku (jestli hledam hodnotu maximalniho ticketu), sdilenym int hodnoty ticketu a lokalnimi int hodnotami,
- v prvni fazi se snazi algoritmus zjistit, jake je nejvetsi cislo tiketu a pote si vzit ticket s hodnotou maximum+1 vice,
- dva (nebo) vice procesu muze mit stejnou hodnotu ticketu,
- pote se nastavi bool na false,
- ve druhe fazi proces ceka, az bude na nej rada,
- nejprve pockam, az jiny proces (od 0) dohleda svoje maximum,
- pote se ceka, az bude na me rada (ticket kladny, ostatni procesy nebudou chtit pristoupit do kriticke sekce),
- az proces projde do kriticke sekce, hodnota jeho ticketu se nastavi na 0,
- v okamziku, kdy maji procesy stejne cislo ticketu, prednost ma proces s nizsim cislem procesu (PID),
- jednim problemem algoritmu je neustale zvysovani cisel (nutnost si davat pozor na preteceni)

Ani jeden z predeslych algoritmu nemusi na dnesnich CPU fungovat, protoze jsou **silne zavisle na poradi pristupu do pameti**.

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 ceka, 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 Semafory

Synchronizacni nastroj nevyzadujici aktivni cekani (nebo alespon minimalizujici - muze se vyskytnout uvnitr implementace operaci nad semaforem). Jedna se o **celociselnou promennou prostupnout dvemi zakladnimi atomickymi operacemi** (hodnota = kolik procesu do nej jeste muze vstoupit):

- lock - zamknuti semaforu, proces vstoupi do kriticke sekce a ostatni budou uspany,
- unlock - odemknuti semaforu - jine procesy se probudi a vstoupi do kriticke sekce.

Dale je mozne mit rozsireni semaforu o:

- neblokujici zamknuti - pokud je mozne zamknout semafor, tak se to provede, jinak se nebude cekat a provede se jina akce,
- zamknuti s horni mezi na dobu cekani - maximalni cekaci doba (pote se proces probudi a bude delat neco jine),
- soucasne zamknuti vice semaforu

Zabezpecuje sdileny pristup do kriticke sekce tak, ze **cekajici procesy uspi** (misto aktivniho cekani) a po uvolneni jej opet probudi. (Pozn.: **poradi pristupu procesu neni garantovano** (tzn. po uspani procesu se netvori fronta, ale spise nejaka mnozina procesu))

Semantika celociselne promenne S - semaforu:

- S je kladne - odemknuto (kolik procesu jeste muze pristoupit do kriticke sekce),
- S je zaporne - zamknuto (absoultni hodnota S udava pocet cekajicich procesu)

Prace se semafory:

- vytvoreni promenne typu semaphore (sdileny semafor),
- inicializace semaforu,
- pred vstupem do kriticke sekce zamek semaforu,
- po vstupu z kriticke sekce odemceni semaforu

Provadeni locku a unlocku **musi byt atomicke** (jejich telo predstavuje taky kritickou sekci). Atomicita lock a unlock se resi:

- zakazem preruseni,
- vzajemnem vyloucenim s vyuzitim atomickych instrukci a aktivnim cekanim - s vyuzitim spinlocku (pouziva se u multiprocesorovych systemu, ceka se pouze na vstup do lock/unlock - kratkou dobu)

Pouzivaji se take:

- read-write zamky - pro cteni lze zamknout vicenasobne,
- reentrantni zamky - stejny zamek muze proces zamknout vicekrat,
- mutexy - binarni semafory, mohou byt odemknuty pouze tim, 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) **jinym 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

Jedenacta prednaska: Pokracovani a dokonceni synchronizace procesu, monitory, deadlock

11.1 Monitory

Synchronizacni prostredky (jeste) vyssi urovne (nez semafory). Problem semaforu v realnem kodu je, ze **je zde spousta sdilenych dat, ktere se budou vzajemne vylucovat** - nechceme mit cely program zamkly - bude zde snaha zamknuti minimalizovat - muze se stat, ze se nekde v nejake vetvi **zapomene lock/unlock a nastane problem** (v realnem kodu tak semafory jednoduche nejsou).

Proto vznikl komfortnejsi synchronizacni mechanismus - monitory. V systakticke podobe vypadaji takto (jazyk Ada, bezne se takto nepouzivaji):

```
monitor monitor-name {
  shared variable declarations

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

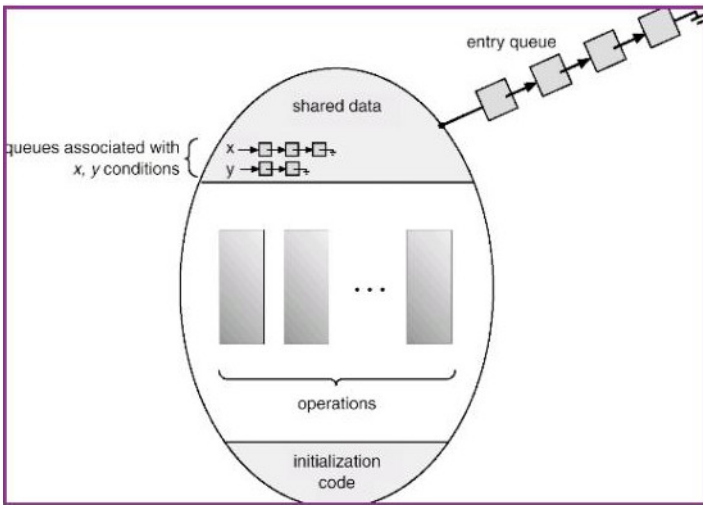
Obrázek 7: z prezentace IOS: Synchronizace procesu - kod monitoru

Monitor:

- je mozne si predstavit jako datovou strukturu podobne tride se sdilenymi promennymi,
- ty jsou sdileny ruznymi procesy pracujici s monitorem,
- procesy se mohou k promennym dostat pres procedury (metody monitoru),
- automaticky monitor zajistuje, ze v danem okamziku pobezi na nem nejvyse 1 metoda,
- kdokoli chce pracovat se sdilenymi zdroji, musi pouzivat tyto metody a zde se automaticky zajisti zamek/odemknuti zdroju,
- pouziti - do monitoru se nadefinuji sdilene zdroje, dale mechanismy, jak se k nim ma pracovat (procedury), dale konstruktor (inicializacni kod)

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 zamknuteho 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 - wait_x() - zarazeni do fronty x - soucasne bude vybran 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 wait() - synchronizovat se s nekym na nejake podmince,
- je mozne nekomu kdo na podmince ceka poslat uvolnujici signal (operace signal() nebo 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 signal/notify]
- 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)
- 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 semafory, semafory 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_cont_t a funkce pthread_cond_wait / signal / broadcast.

11.2 (Nektere) Klasicke synchronizacni problemy

11.2.1 Problem producenta a konzumenta

- mame smecku procesu, predem rozdeleny, ci se dynamicky deli na producenty a konzumenty, komunikuji spolu pres vyrovnavaci pamet (napr. pole s kruhovym bufferem),
- nutne procesy synchronizovat pro spravnou praci s pameti,
- resenim jsou tri semafory (full, empty, mutex),
- full - rika kolik polozek je aktualne v cache k dispozici, zamykanim si rezervuji polozku v pameti ke spotrebe,
- empty - rika kolik je volnych slotu v cache, zamykanim se rezervuje kapacita pro zapis do cache,
- mutex - pomocny semafor, ktery se zamkne pri praci s ukazovatky do cache,
- **semafory je nutne vhodne 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 prostredky 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 vyrovnavaci pameti je neco ke konzumaci, polozku si alokovali),
- zamknou si pristup k cache (zamkne se mutex),
- z vyrovnavaci 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 readcout) a semafor wrt (semafor pisaru),
- opet je nutna inicializace (pocet ctenaru=0, oba semafory=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 readcout==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, pokd 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 nevyhodu - **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 -

– Písař:

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

– Čtenář:

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

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 filosofy je jedna hulka (ji se dvema..) - 5 hulek pro 5 filozoru,
- 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 filosofove budou jist soucasne 1 hulkou,
- zavede se 5 binarnich semaforu (pole semaforu),
- vsechny inicializovane tak, ze jsou odemknute

Filozof I:

- i = 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);

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

Obrázek 13: z prezentace IOS: Synchronizace procesu - pseudokod reseni tohoto problemu

Gde deadlock?

- procesy za planovani se mohou naskladat tak,
- ze kazdy si vezmne svoji 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 zdrojum 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 vlastneneho nejakym procesem z dane mnoziny, ktery jediny muze tento zdroj uvolnit, a to az po dokonceni jeho pouziti.

Vysvetleni 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, ktery 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 nejakym procesem z dane mnoziny = obvykle jiny, muze to byt i stejny proces,
- jediny ... po dokonceni jeho pouziti = pracujeme s tim, ze se pouziva zamek, ktery se pouziva neomezene (zadny casovy zamek), nikdo nemuze zdroj vzit, odemknout, nikde neni zadny casovac,
- kardinalita-pocet procesu mnoziny 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 mnoziny procesu je pozastaven a ceka na nejakou udalost, ktera by mohla nastat pouze tehdy, pokud by mohl pokracovat nektery z procesu z dane mnoziny. (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 zdrojum s vylucnym omezenym pristupem,
- v praxi mohou byt jednotliva volani ve zdrojaku velmi daleko od sebe a zamykany mohou byt jen za urcitych podminek,
- 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 - priklad uvaznuti

Popis (kodu):

- dva semafory, oba na zacatku odemknute,
- za predpokladu pouziti standardnich semaforu se standardni semantikou, neni zde jadro, ktere napr. hlida situaci a jeden z procesu by zabilo, ..
- jeden proces zamkne prvni mutex, neco provede, pokusi se zamknout druhy,
- druhy proces zamkne druhy mutex, neco provede a zamkne prvni,
- nastane deadlock

11.3.3 Coffmanovy podminky

Rozvadi to, co je obsahem uvaznuti do 4 nutnych a postacujicich podminek, ktere jsou zapotrebi pro uvaznuti.

K tomu aby uvaznuti mohlo nastat je nutne ci postacuji nasledujici podminky:

- vzajemne vylouceni pri pouzivani prostredku,
- vlastnictvi alespon 1 zdroje, pozastaveni a cekani na dalsi,
- prostredky vraci (pouze) ten proces, ktery jej vlastni a to po dokonceni jejich vyuziti (zdroj je vlastnen procesem, pouze ten proces ho muze vratit a to po dokonceni vyuziti toho zdroje),
- cyklicka zavislost na sebe (navzajem) cekajicich procesu (rozumi se tim to, ze jeden proces ceka na druheho - zadne aktivni cekani v cyklu)

11.3.4 Reseni uvaznuti

- prevence uvaznuti,
- vyhybani se uvaznuti,
- detekce a zotaveni

Zakladni spolecna myslenka vsech reseni je **princip, kterym se snazi zrusit platnost alespon jedne z nutnych podminek k uvaznuti** (Coffmanovych podminek - kdyz se jedna zneplatni - nemuze dojit k uvaznuti).

11.3.5 Prevence uvaznuti

Resi, jak tyto podminky zneplatnit:

- prvni (vylouceni)
  - nepouzivat zadne sdilene prostredky nebo uzivat sdilene prostredky,
  - ale pouze takove, ktere umoznuji skutecne soucasny sdileny pristup
  - a u kterych neni nutne vzajemne vylouceni procesu (ne vzdy problem pujde vyresit timto zpusobem),
- druha (vlastnictvi)
  - proces muze zadat o prostredky pouze tehdy, pokud zadne nevlastni,
  - (kdyz chce proces pouzivat soucasne 5 zdroju v jednom okamziku, musi zamknout vsechny soucasne - ziska nebo ceka - systematicky v kodu zamkykam vsechny nebo zadny zdroj - kontrola zamku: pokud proces jiz neco zamkl, nemuze znovu zamykat - musi vse uvolnit a pote muze zamknout),
  - nevyhodou je nutnost zamknuti vsech zdroju, ktere se pouzivaji soucasne (napr. 30 min se pouziva 1 zdroj, pote 1 min se pouziva dalsi zdroj soucasne s prvnim - musi se zamknout na 30 min oba zdroje),
- treti (navrat)
  - nebudu vyzadovat, aby proces vzdy ziskal vsechny zdroje, ktere chce pouzivat soucasne v 1 okamzik,
  - ale umoznim mu ziskavat postupne dalsi a dalsi zdroje, postupne prizamykat tyto zdoje,
  - ale jen v situaci, kdy je opravdu mozne tyto zdroje ziskat,
  - pokud se proces pokusi neco prizamknout a ono se to nepovede, bude to reseno specialnim zpusobem
  - (napr. tak, ze proces bude zabit a vsechny zdroje mu budou odebrany),
  - nevyhodou je situace, kdy v okamziku kdy se proces zabije, uz mohl s nekterymi zdroji neco delat, zdroje mohou byt v nekonzistentnim stavu (idealne se pokust vratit zdroje do konzistentniho stavu),
- ctvrta (cyklicka zavislost)
  - zavedenim usporadanim nad zdroji (ocislovani),
  - je mozne tyto zdroje ziskavat pouze od urciteho poradi (napr. od nejmensiho k nejvetsimu - zacnu 1, pak 2, pak 3, .. - pokud zamknu 5, muzu zamknout 5+, ale 3 uz ne),
  - bud se konstrukci programu zajisti, aby se vzdy zamykalo v tomto poradi, a pote analyzou, auditem se overi, zda opravdu ma program takovou zamykaci disciplinu,
  - anebo budu mit system zamykani, ktery toto bude kontrolovat

Ke vsem bodum se da rict, ze bud bude program navrhovan tak, aby byl konformni s pouzitou strategii nebo se vynuceni stragie kontrolovat az pri behu.

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, ktery by mohl nastat s ohledem na to, co procesy deklarovaly

Priklad:

- 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, ktery zdroje prideluje, ten si prubezne udrzuje graf vztahu mezi procesy a zdroji - dva typy uzlu (procesy a zdroje) a tri typy hran - hrany od zdroje k procesu (ktery zdroj je kym vlastnen, zvyraznena, ”tucna”hrana), hrany Pi a Ri od procesu ke zdroji (kdo o ktery proces zada), Pi a Ri od procesu ke zdroji (kdo o ktery zdroj muze pozadat - zvyraznena 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 P2 do R2 udela R2 do P2), 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 cekal na uvolneni zdroje, i kdyz je volny)

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

11.3.7 Detecke uvaznuti a zotaveni

System pridelovani zdroji **umozni pripadny vznik uvaznuti** (pokud pomineme to, ze externe k mnozine uvaznutych procesu mame "strazneho andela", ktery uvaznuti vyresi), **ale periodicky** (bezi specialni proces, zajistujici 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 nezadouciho 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 nemusi projevit (nedeterminismus se nemusi 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, ktery 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 zabit - zpusobit uvaznuti, proto se napriklad misto toho pred praci s danou promennou misto 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, ktery se bude verifikovat (ci kombinaci, napr. cast jadra OS a model OS, odlehcenou implementaci),
- specifikace vlastnosti, ktere maji byt overeny - mohou to byt genericke vlastnosti napr. v systemu nesmi byt deadlock nebo slozitejsi vlastnosti napr. v cache neni nikdy vice nez 10 polozek,
- 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 byt, pro urcite tridy systemu je mozne automaticky vysyntetizovat korektni implemetaci.

11.4.1 Theorem proving

- teoremem je zde veta, ktera rika 'Muj program XXX spluje XXX specifikaci',
- pouzivaji se poloautomaticke dokazovaci prostrek (eviduji, co uz jste dokazali, maji db standardne platnych skutecnosti z logiky, znaji pravidla spravneho odvozovani),
- vyzaduje se expert, ktery urcuje, jak se dukaz ma vest (presto se to pouziva, napr. Mikrojadra ProvenCore ci seL4),
- existuji i plne automaticke dokazovace (rozhodovaci procedury, umi automaticky rict zda plati ci ne) - obvykle pro omezene fragmenty logik, spise se pouzivaji jako pomocne

11.4.2 Model checking

- obvykle plne automatizovany pristup, prostredek,
- zalozen na systematickem generovani stavu systemu a stavoveho prostoru (systematicky se hleda, zda nekde neni chyba),
- nevyhodou je obrovsky pocet stavu - napr. pri N dvoustavovych procesech (procesy zacnou a skonci), muze se vygenerovat 2^N stavu - problem stavove exploze

11.4.3 Static analysis

- snaha analyzy a verifikace systemu na zaklade jeho zdrojoveho kodu, aniz by se tento kod provadel (nebo alespon ne v puvodni semantice - napr. celociselne promenne 0 - 1235, pamatuji si jen jestli je hodnota zaporna, 0 ci kladna),
- nejjednodusi staticky analyzator je grep (naji se syntakticke vzory chyb a grepem se vyhledavaji),
- ma ruzne podoby: data flow analysis, constraint analysis, type analysis, abstract interpretation, symbolic execution, ..
- nastroje: Facebook Infer, Frama-C, Microsoft SDV, SpotBugs, cppcheck, ...

12

Dvanacta prednaska: Zacatek spravy pameti.

12.1 Sprava pameti

Aby program mohl byt proveden, musi byt spusten - musi byt nad nim vytvoren proces, msui mu byt pridelen procesor a take pamet (a dalsi zdroje - soubory, ..)

Rozlisujeme:

- logicky adresovy prostor - LAP - je virtualne adresovy prostor, se kterym pracuje CPU pri provadeni kodu (uzivatelskeho ci jadra - kazdy proces i jadro maji sve logicke adresove prostory),
- fyzicky adresovy prostor - FAP - adresovy prostor fyzickych adres pameti (obsahuje adresy, ktere se umistuji na adresove sbernici, chcemeli z pameti nacist nebo do ni zapsat - je spolecny pro vsechny procesy i jadro)

Casto logicky adresovy prostor **jadra byva podprostorem logickeho adresoveho prostoru jednotlivych procesu**. Vsechny logicke adresove prostory procesu (v Linuxu) se prekryvaji ve stejne casti - **v casti, kde je LAP jadra**. LAP jadra **neni** ale procesum pristupny. Vyhodou je, ze pri prechodu rezimu procesu na rezim jadra se **pouze zpristupni tento LAP** nebo pri prepinani procesu (krome zmen mapovani casti LAP procesu) se **nemusi menit mapovani pro cast LAP jadra**.

Proces pracuje s logickymi adresami, ale na adresovou sbernici se umistuji fyzicke adresy. Toto mapovani provadi **MMU (Memory Management Unit):**

- HW jednotka specializovana na predklad logickych adres na fyzicke,
- dnes bezne soucasti cipu CPU,
- provadi preklad na zaklade datovych struktur,
- obsah struktur je castecne ulozen ve specialnich registrech, castecne v hlavni pameti systemu,
- soucasti MMU je cache, obvykle TLB, pro urychleni prekladu

Komunikace CPU a pameti:

- na CPU bezi programy (CPU pracuje s logickymi adresami),
- pri cteni/zapisu logicke adresy,
- adresa se preda do MMU,
- MMU provede preklad logicke adresy na fyzickou,
- MMU umisti fyzickou adresu sbernici a po ni se prenesou data/kod mezi pameti a CPU

12.2 Pridelovani pameti

Existuje vice urovni pridelovani pameti. V **nejnizsich (z hlediska blizkosti hw) se prideluje FAP pro zamapovani do LAP**, pote jsou napr. pridelovani pameti pres knihovni funkce (malloc - mimo rezim jadra ci kmalloc, vmalloc - jadro), az po vyseurovnove pridelovani v ramci aplikaci.

Nejnizsi uroven pridelovani je **implementovana v jadre a jedna se o pridelovani FAP pro zamapovani LAP**. Bezne zpusoby pridelovani pameti (a mapovani LAP na FAP):

- pridelovani po spojitych blocich (contiguous memory allocation),
- segmentech,
- strankach,
- kombinace vyse uvedeneho (intel - segmenty a stranky)

Funkce malloc:

- pri zadosti o alokaci nejakeho kusu pameti (poctu bajtu),
- musi malloc pozadat jadro o prideleni FAP,
- pozaduje od jadra vetsi blok pameti (segment, stranka),
- z bloku pameti se vykousne pozadovany pocet bajtu,
- ty dostane k dispozici uzivatel,
- pri volani dalsich mallocu, dokud pozadovany pocet bajtu nebude vetsi nez prideleny blok, nepujdou pozadavky do jadra, ale bude se cerpat jiz prideleny prostor

12.3 Contiguous Memory Allocation

Mechanismus mapovani logickych adres na fyzicke a pridelovani pameti po spojitych blocich. Jedna se o **nejjednodusi mechanimus z hlediska hw, tak obsluhy OS**. V beznych vypocetnich systemech se prilis **nepouziva**, nicmene je vhodny pro jednoduche a vestavene aplikace, ktere maji bezet na jednoduchem hw.

Popis:

- je to po spojitych blocich (neco jako ukladani dat spojite),
- k popisu takoveho mapovani je treba znat, ve kterem FAP je zamapovan pocatek LAP,
- je nutne vedet, jak je usek pameti velky (pro odchyceni pristupu mimo meze tohoto prostoru)

Preklad adresy:

- MMU si pro aktualne bezici proces pamatuje 2 udaje,
- v limitnim registru si pamatuje, kolik proces pameti dostal,
- v relokacnim registru si pamatuje, na jakou fyzickou adresu byl zamapovan LAP procesu,
- pokud dostanu logickou adresu napr. 10,
- zjisti, zda je adresa v ramci naalokovaneho prostoru,

- pokud ne - chyba pri pristupu do pameti, posle se preruseni typu trap, obvykle je proces predcasne ukoncen,
- pokud ano - mapovani se provede tak, ze se pouzije bazova adresa, na kterou je zamapovan FAP procesu, secte se s logickou adresou prostoru = mam adresu ve FAP (fyzickou adresu)

Priklad prekladu LA na FA:

- proces 1 ma zacatek LAP na zacatku FAP (konec na FAP 100 000), proces 2 nekde uprostred (LA 0 = PA 1 mil.),
- preklad LA 10 procesu 1,
- zkontroluje se, zda 10 je mensi nez 100 000 (= konec LAP p1),
- bazova adresa 0 se pricte s LA, tedy 0 + 10 = 10,
- preklad LA 10 procesu 2,
- zkontroluje se, zda 10 je v ramci rozmezi (ano),
- bazova adresa 1 000 000 se pricte k 10, tedy 1 000 010

Tento mechanismus ma radu nevyhod:

- vyrazne se zde projevuje externi fragmentace pameti (FAP),
- pridelovanim a uvolnovanim useku pameti vznika poslouppnost obsazenych a neobsazenych useku pametu, useky mohou byt obsazeny ruznymi procesy,
- nejhorsim dopadem je, ze pri scitani volnych useku pameti muze byt prostor pro prideleni pameti procesu dostatecny, ale tyto volne useky nejsou-nemusi byt spojite, takze zde zdanlive neni dostatek pameti pro dany proces (neni mozne provest alokaci),
- problemy se zvetsovani prostoru daneho procesu,
- snaha o minimalizace dopadu externi fragmentace pomoci ruznych strategii (first fit se nepouziva, namisto toho napriklad best fit, worst fit ci binary buddy, ..),
- provadi se dynamicka reorganizace pameti (nakladne)
- neni mozne rozumne ridit pristupova prava v ramci pridelene pameti (nelze: cast pameti pro cteni, cast pro zapis, ..),
- neni mozne take sdilest cast adresoveho prostoru (vse nebo nic),
- pri virtualizaci pameti (swapovani) je nutne odlozit veskerou pamet na disk a pote ho vratit zpet - pomale, muze byt zbytecne.

definice:

bazova adresa - pocatek LAP (tj. adresa 0 v LAP) ve FAP (nebo: pocatecni adresa LAP procesu ve FAP)

first fit - prochazi se volnymi useky a pouzije se prvni volny usek

best fit - podivam se na seznam volnych useku a vybere se ten, ktery je dostatecne velky, ale vyberu ten nejmensi z dostatecne velkych,

worst fit - paradoxne lepsi jak best fit, opak best fit, hleda se usek dostatecne velky a pouzije se ten nejvetsi (zbyde nepouzity velky kus, ktery se vyuzije pozdeji, napr. pri dalsim zvetsovani)

binary buddy - udrzuje se seznam volnych useku pameti, najdu si usek pameti, ktery odpovida nejlepe, a pokud presahuje, resp. je 2x vetsi nez pozaduji, rozdelim ho na polovinu a opet zjistim, zda je usek 2x vetsi, pokud ano, delim useky tak dlouho, az dojdu k useku pameti, ktery nelze rozdelit na poloviny, tak aby byl uspokojen dany pozadavek a pamet se prideli

12.4 Segmentace pameti

- LAP je rozdelen na kolekci segmentu,
- segmenty mohou byt prideleny prekladacem, programatorem, jednotlivym castem procesu (castem dat, proceduram, zasobniku, ..),
- kazdy segment ma cislo a velikost,
- LA je cislo segmentu a posun v nem,
- jednotlive segmenty patrici jednomu procesu nemusi byt zamapovany spojite (jeden segment ano, spojite, ale ruzne segmenty nemusi)

Preklad adresy LA na FA:

- MMU potrebuje pracovat s tabulkou udaji - tabulkou segmentu, ulozena v RAM, v MMU je odkaz na zacatek tabulky (=pole),
- logicka je adresa je delena na cislo segmentu (s) a posuv v ramci segmentu (d),
- pri prekladu se vezme cislo segmentu (s), napr. pri praci s s=10 se podiva do radku tabulky 10,
- na prislusnem radku se najde jakou ma segment velikost, jakou ma bazovou adresu,
- pokud bude napr. d=1000, podiva se jestli je d v ramci pametoveho prostoru, ktery je pridelen,
- pokud ne - vyjimka, chyba,
- pokud ano - vezme se bazova adresa segmentu, secte se to s posuvem a mam FA

Priklad prekladu:

- mam LA se segmentem s=10 a posuvem d=1000,
- s 10 pristoupim na 10. radkek tabulky segmentu,
- zde budu mit limit (velikost segmentu, zde napr. 100 000) a jeho bazi,
- pokud d je v limitu (1000 ¡ 100 000?),
- vezmu bazi (napr. 1 000 000), sectu ji s d,
- tzn. 1 000 000 + 1000 a dostanu fyzickou adresu (1 001 000)

Vyhody:

- mohou byt pouziti jako jemnejsi jednotka ochrany pri pristupu do pameti (nektere mohou oznaceny jako pro cteni, nektere zapis, nektere v rezimu jadra, ..),
- jemnejsi jednotka pro odkladani pameti na disk (odlozit se mohou segmenty, ne cela pameti procesu),
- jemnejsi jednotka pro sdileni,

- implementace je jednoducha,
- pamet je pridelovana nespojita, zmirnuji se dopad externi fragmentace

Nevyhody:

- pri zvetsovani opet dopad externi fragmentace,
- mozny zdroj chyb, segmentace je viditelna procesu

12.5 Strankovani

Je aktualne nejpouzivanejsim mechanismem mapovani LAP na FAP. LAP je rozdelen **jednotky pevne velikosti - stranky**, FAP je rozdelen na odpovidajici **jednotky stejne velikosti - ramce**. (nejcasteji velikost stranky je 4 KiB)

12.5.1 Vlastnosti

Vyhody:

- pamet je pridelovana po ramcich (ty se zapamuji do stranek),
- neviditelne pro uzivatelske procesu,
- minimaluzuji se problemy s externi fragmentaci (podobne jako clustery u disku):
  - porad vznikaji useky volnych a vyuzitych, nicmene nejmensi nevyuzita ”dira”v pameti je 1 ramec, ten se vzdy da vyuzit (nespojite),
  - mozne snizeni rychlosti pristupu do pameti (nespojita alokace), proveji se vetsi pocet kolizi v caches,
  - zpomalovani alokace a dealokace pameti (delsi prace se strukturami co popisuji aktualni obsah pameti),
  - je snaha pridelovat pameti po spojitych posloupnostech ramcu (pokud je to mozne), napr. pomoci algoritmu binary buddy
- jemna jednotka ochrany pristupu do pameti (kazda jednotliva stranka muze byt r, rw, uziv. rezim ci rezim jadra, je mozne provadet NX bit),
- jemna jednotka sdileni (pamet sdilena mezi procesy lze sdilet pro strankach),
- pri nedostatku pameti se odklada po jednotlivych strankach,

Nevyhody:

- slozitejsi implementace,
- vetsi rezie,
- interni fragmentace (podobne jako u disku),
- jsou vnimany jako vyrazne mensi nez vyhody systemu - proto jsou pouzivany nejvice

definice:

NX bit - na nekterych architekturach specifikace, jestli obsah dane stranky lze interpretovat jako kod a provadet

12.5.2 Mapovani logickych adres na fyzicke

V nejjednodusim pripade se pouzivaji **jednoduche - jednourovnove tabulky stranek**. OS udrzuje **informaci o volnych ramcich** (zalezi na OS, hw nezajima), pro kazdy **proces si udrzuje tabulku stranek** (musi byt struktrovana tak, aby tomu dana architektora rozumela)

12.5.3 Tabulky stranek

- logicka adresa je rozdelena na cislo stranky (p-place) a na posuv stranky (d-displacement),
- cislo stranky se pouzije jako index do tabulky stranek (=pole v pameti),
- v MMU je registr, ktery bude ukazovat na to, kde ma dany proces ulozenou tabulku stranek,
- cislo stranky se vezme jako index do tabulky stranek (p=10, pristoupim na 10. polozku),
- pokud byla stranka alokovana (=ma pridelen ramec), na danem radku najdu cislo ramce,
- cislo ramce se spoji s posuvem a dostanu FA

Na radku tabulky stranek, ktery odpovida dane strance, kde je ulozeno odpovidajici cislo ramce, **jsou ulozene ridici priznaky mapovani - platnosti mapovani, pristupu, modifikace, pristupova prava** (r, rw, user rezim, jadro rezim, moznost provadeni), **globality**.

Tabulky stranek jsou udrzovany v **hlavni pameti (RAM)**, **zvlast** pro kazdy proces, MMU maji ve specialnim registru **pouze ukazatel na zacatek tabulky stranek**, pri prepinani kontextu se **meni pouze ukazatel** na zacatek tabulky stranek. (konkretne u intelu se ten registr jmenuje CR3)

Neprovedeme-li zadnou dalsi optimalizaci, tak kazdy jednotlivy pristup do pameti (pro data ci instrukce) se zmeni **z jednoho pristupu na 2** (pri nacteni dat z RAM - nejprve musim jit do tabulky stranek, pote nacist vlastni data - oboji jsou v RAM) - zpomaleni o 100 procent. Pouziva se tak **vyrovnavaci pamet TLB - Translation Look-aside Buffer** pro urychleni prace s pameti.

Priklad prekladu LA na FA:

- LA, tvorena cislem stranky p=10, posuvem d=1000,
- MMU bude mit v pameti umistenou tabulku stranek, v registru bude odkaz na adresu, kde se nachazi tabulka stranek,
- MMU pouzije p=10 jako index do tabulky stranek, pristoupi na radek 10 tabulky stranek,
- soucasti obsahu radku je odpovidajici cislo ramce (napr. 500),
- dostaneme fyzickou adresu, tvorena ramcem f=500 a posuvem d=1000

definice:

priznak platnosti mapovani - zda je dany adresovy blok nebo neni pouzit (ne vsechny tabulky strankem v danem okamziku musi byt vyuzity)

priznak pristupu - byla stranka od okamziku zavedeni pameti zpristupnena? (dale slouzi jako informace, zda je stranka vhoda na odlozeni do pameti - jadro cas od casu prochazi bity a pristupove bity nuluje, pri hledani

kandidata na odlozeni se zjistuje, zda v nekolika periodach bylo ke strance pristoupeno nebo ne - pokud delsi dobu ne - bude odlozena)

priznak modifikace - byla, nebyla modifikovana stranka? (pri odlozeni stranky a pote znovu zavedeni do pameti - aby se nemodifikovana stranka neodlozila znovu)

priznak globality - kdyz je stranka globalni, je sdilena za behu ruznych procesu, typicky se pouziva pro stranky jadra

12.5.4 TLB

- obsahuje dvojice cislo stranky a cislo ramce + jsou tam nektere ridici priznaky spojene s mapovanim (opravneni, modifikace),
- v TLB nejsou cele stranky ci ramce (je to cache pro mapovani cisla stranky na cislo ramce),
- typicky implementovana jako (castecne) asociativni pamet,
- do pameti se pristupuje tak, ze TLB vsechny cisla dvojici cislo stranek a cislo ramce a HW se podiva, zda jestli tam je alespon 1, ktery ma mapovani pro dane cislo stranky a pokud ano, vybavi se odpovidajici ramec

Nekdy je operace castecne asociativni:

- nekolik bytu z LA je pouzito klasickym zpusobem adresovani,
- zbytek je pouzit pro asociativni vyhledavani,
- napr. pokud se vymezi pro klasicke adresovani 2 byty, TLB se rozdeli na 4 casti (00,01,10,11) - bloky a v ramci bloku se bude hledat asociativne

Preklad:

- obdobne jako bez TLB,
- misto pristupu do pameti na urcity radek tabulky hw bude hledat, zda nekde v tabulce je polozka s cislem 10 a pokud ano, vezme k tomu odpovidajici cislo ramce,
- pokud takove vyhledani nastane, rika se tomu TLB hit, pouzije se okamzity preklad - cislo ramce a posuv a jsem v FAP,
- pokud se nepodari uspesne vyhledat polozku, pouzije se stejny postup jako bez TLB (pres pamet) - TLB miss

Priklad prekladu s TLB:

- architektura s 1-urovnovou tabulkou stranek, k dispozici TBL, ktera je castecne asociativni, 2 byty se pouziji pro rozliseni casti TBL a zbytek se prohledava asociativne,
- LA, rozdelena na cislo stranky p - to na rozdeleno na horni 2 byty (adresovani v ramci TLB), zbytek bude cislo stranky (p=01 a zbytek 10), posuv d=1000,
- TLB rozdeleno na 4 bloky, jeden z nich adresovan bity 00, dalsi 01, dalsi 10, a posledni 11,
- p=01 10 mi rika, ze mam jist do 2. casti,
- v bloku budou cisla stranek, odpovidajici ramce,

- vsechny radky se prohledaji paralelne - je v nekterem z tech radku cislo 10 ?,
- ano - vezmu odpovidajici ramec (napr. 1 000 000) - TLB hit,
- FA bude f=1 000 000 a posuv d=1 000,
- ne - nastane TLB miss, je nutne jit do klasicke tabulky stranek a tam hledat

K neuspesnemu vyhledani muze dojit **pri pristupu k instrukcnimu kodu** (cteni instrukce, operandu) **u kazdeho cteni muze dojit k neuspesnemu vyhledani opakovane** (napr. 32b architektura - 4b instrukce, nespravne zarovnana instrukce, 2b na zacatku 1. stranky, 2b na zacatku 2. stranky - je nutne provest preklad cisel odpovidajici ramec u obou stranek - muze dojit 2x k TLB miss) - pro instrukci (velka 4b) ktera nacita 4b z pameti jsou nutne 4 preklady a 4x muze nastat TLB miss.

Pokud dojde k TLB miss:

- hw bude hledat automaticky v tabulce stranek - plati u architektur, kde je TLB hw rizene (vetsina architektur),
- jsou i SW rizene TLB, pokud nastane TLB miss, nastane preruseni od MMU k CPU, jadro si tento preklad provede, specializovanymi instrukcemi se naplni preklad do TLB a preklad adresy se opakuje (CPU jako MIPS, SPARC))

Nekdy muze byt pouzito vice TLB, napr. 64b architektury obvykle maji **preklad narocnejsi - je potreba vice TLB** (typicky 2-urovnova TLB - jedna pro preklad kodovych adres, druha pro kodovych adres).

Pri prepnuti kontextu (se meni hodnota ukazatele tabulky stranek v MMU registru) **je nutne invalidovat obsah TLB a znovu naplnit jeho obsah - pouzivaji se optimalizace:**

- pouzivaji se globalni stranky (stranky pouzivane jadrem - jsou na stejnem miste),
- na nekterych CPU se jeste do radku TLB doplnuje identifikator procesu, pro ktery je mapovani platne (vyhledava se na zaklade cisla procesu a cisla stranky),
- pri zmenu obsahu tabulek stranek je nutna take invalidizace obsahu TLB (neprojevily by se zmeny v tabulce stranek)

Udaje se do TLB dostavaji (v pripade hw rizenych) tak, ze **se preklad nahrava do TLB pri prvnim pristupu na stranku**, (pri nedostatku prekladu se nejaky preklad odstrani) **v pripade sw rizenych TLB muze byt obsah nahravan specialnimi instrukcemi jadrem**. HW take **pocita s tim, ze budeme s pameti pracovat spojite**, tak si muze nekdy nahrat do TLB preklady nasledujicich adres **dopredu**.

definice:

asociativni pamet - pamet neni adresovana adresou, ale vybavuje si obsah na zaklade casti jeho obsahu

12.5.5 Efektivnost strankovani s TLB

Efektivni pristupova doba je:

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

- kde τ je vybavovaci doba RAM,
- ϵ je vybavovaci doba TLB,
- α je pravdepodobnost uspesnych vyhledani v TLB (TLB hit ratio, $1 - \alpha$ je tedy pravdepodobnost neuspesnych vyhledani),
- jedna se o vazeny prumer, kde se spocita doba pristupu do pameti pokud se zadari vyhledani $(\tau + \epsilon)\alpha$,
- pokud se nezadari vyhledani $2\tau + \epsilon$ (2x pristup do pameti)
- napr. pro $\tau = 100ns, \epsilon = 20ns, \alpha = 0.98$, pote dojde ke zpomaleni o 22 procent

Tento vztah je sestaven za predpokladu, ze pote co se neuspesne vyhleda v TLB, pujdu do RAM, najdu preklad v RAM, a pote pujdu do RAM pro data - v praxi to funguje tak, ze pokud **MMU nalezne preklad v RAM, automaticky ho ihned doplni do TLB, provede opakovany preklad pro TLB, a az pote jde pro data/kod do pameti**. (vztah v praxi by pristupova doba mela $2\tau + 2\epsilon + \delta$, delta - provede se uprava TLB)

Je **dulezite aby bylo TLB uspesne, jinak se pristupova doba do pameti bude rychle zpomalovat**, uspesnost TLB ovlivnuje:

- velikost TLB (to ovlivni vyroci cipu),
- dobra lokalita odkazu programu (muze ovlivnit programator)

Priklad:

- inicializace matice, MAX x MAX prvku,
- v pameti linearizovana, ukladaji se typicky po radcich,
- inicializace 2 zpusoby,
- prvni zpusob incializuje matici po sloupcich, druhy po radcich,
- v pripade pristupu po radcich je v souladu s ulozenim pameti po radcich - bude vyrazne efektivnejsi z hlediska mozneho poctu neuspesneho vyhledani TLB, zatimco pristup po sloupcich bude mene efektivni,
- v pripade 2x2 matice po radcich nastane 2x TLB miss (pri TLB kapacite o jedne polozce), po sloupcich nastane 4x TLB miss

definice:

lokalita odkazu programu udava s kolika ruzymi shluky adres (adresy blizko sebe, shluk = 1 stranka) pracuje v danem procesu za kratky casovy okamzik program (pokud shluku adres neni mnoho - program ma dobrou lokalitu adres)

12.5.6 Implementace tabulek stranek

Kdyby se tabulky stranek implementovali jako jednourovnove, zabraly by prilis moc pameti.

Pro 32b systemy (intel) se strankami o velikosti 4 KiB (12 bitu se odkousne na posuv stranky), zbyva 20 bitu LA, udavaji cislo stranky a pocet radku v tabulce stranek (odpovida poctu stranek), odpovida to pres 1 milionu polozek. Ma li mit 1 polozka tabulky stranek 4B (je tam cislo stranky, ramce - 20 bitove, k tomu ridici priznaky - celkem potreba 27 bitu + zarovnani na bajty - 32 bitu), dostavame tak 4 MiB pro jednu tabulku stranek pro kazdy procesu (bezne muze bezet 100 procesu - jen na tabulky stranek je potreba 400 MiB)

Pro 64b systemy je problem jeste horsi - take se pouzivaji 4 KiB stranky, teoreticky by slo pouzit az 52 bitu na adresu stranky - pro polozku bude potreba 8B.

12.5.7 Hierarchicke tabulky stranek

Nebude zde 1 tabulka stranek pro 1 proces, ale pro **1 proces bude vice tabulek v hierarchicke strukture**, ne vsechny dilci tabulky musi byt v danem okamziku alokovane (vznikaji *tabulky tabulek stranek*).

Princip fungovani hierarchicke tabulky na prikladu **dvouurovnove tabulky stranek (pouzivane u i386)**:

- 32b logicka adresa, 12 dolnich bytu jsou vymezeny na posuv v ramci stranky ($2^12 = 4KiB$ velikost stranky),
- zbyvajicich 20 bytu je pro cislo stranky (rozdeleno na horni a dolni cast po 10b), pouzivaji se jako indexy do dilci tabulky stranek 2. urovne a nasledne dilci tabulky stranek 1. urovne,
- dilci tabulka 2. urovne je pro dany proces prave 1 - jedna se o adresar stranek,
- v registru CR3 (v MMU na Intelu) je ulozen odkaz na zacatek adresare stranek (=pole) (pro aktualne bezici proces),
- prvni uroven tabulek stranek nemusi byt pouzita prislusnym priznakem v adresari stranek (je mozne rict, ze prislusna polozka adresare stranek nepouziva 1. uroven - pracujeme s velkymi strankami - tedy pracujeme s posuvem 22 bytu - 4 MiB stranky)

Preklad LA na FA (na prikladu vyse):

- vezme se obsah tech hornich 10 bytu (31-22), je to 10b cislo, ktere se pouzije jako index do tabulky stranek 2. urovne,
- index = cislo radku,
- na teto polozce najdu odkaz na zacatek dilci tabulky stranek 1. urovne (tech muze byt vic),
- vezmu cislo ulozene v dalsich 10 bytech (21-12), pouziju ho opet jako index do teto tabulky,
- zde bude odpovidajici cislo ramce,
- za predpokladu ze v casti, kde jsou ridici priznaky je uvedeno, ze mapovani je platne,
- cislo ramce se vezme, prida se k tomu posuv a mam pristup do pameti

Efektivita pristupu do pameti (2-urovnove tabulky stranek, viz vyse):

- z jednoho pristupu se stanou 3 pristupy,
- nejprve musim do adresare stranek (1x),
- pote do dilci tabulky stranek (2x),

- az pote mohu do pameti (3x),
- pokud nebude pracovat TLB, zpomaleni bude o 200 procent

Tabulky stranek na x86-64 systemech (4-urovnove tabulky):

- 64 bitove adresy, 2 urovnove tablky stranek nestaci,
- dolnich 12 bitu je pouzit jako offset stranky (v pripade 4 KiB stranek),
- cislo stranky ma (47-12) 36 bitu,
- tedy mame 48 bitu logicke adresy, s tim ze mame jeste znamenkove rozsireni (47. bit se ma zopakovat az po bit 63),
- cislo stranky je rozdeleno na 4 indexu (od spodu po 9 bitech index do 1. urovne, 2. urovne, 3., 4.),
- prvni urovni se rika dilci tabulka stranke,
- 2. urovni se rika adresar stranek,
- 3. uroven je tabulka ukazatelu,

Preklad LA na FA (na prikladu vyse, tj. 4-urovnove tabulky na 64bit systemech):

- opet se pouziva registr CR3, ve kterem je ulozen odkaz na zacatek tabulky 4. urovne (adresa je ulozena mezi bity 51-12 - fyzicke adresy se nepouzivaji 64bitove, ale pouziva se maximalne 52 bytu - architekturalni limit - dolnich 12 bytu je posuv v ramci stranky),
- mezi bity 47-39 se vezme index do tabulky 4. urovne, tam se najde odkaz na zacatek tabulky 3.urovne,
- pouzije se dalsi index (byty 38-30), dostaneme odkaz na zacatek tabulky 2. urovne, ...,
- pouzije se posledni index (20-12), dostaneme odkaz na dilci tabulku stranek, tam dostaneme cislo ramce, k nemu se pricte posuv,
- dostavame fyzickou adresu,
- jeden radek v tabulce zabira zde 8 bajtu, pri velikosti 4 KiB tabulky na adresaci jednoho radku staci 9 bytu (proto tady 9, ne 10 jak u 32b)

V x86-64 dojde **bez TLB zpomaleni o 400 procent - 5 pristupu** do pameti. Takze zde zavisi (velikost stranek), jake se pouzije adresovani:

- ctyrurovnove - stranky o velikosti 4 KiB,
- triurovnove - stranky velke 2 MiB,
- dvouurovnove - stranky velke 1 GiB

12.5.8 Hierarchicke stranky - TLB

Roste zde vyrazne **vyznam TLB**:

- na 64 bitovych CPU jsou vetsi, maji slozitejsi organizaci,
- maji viceurovnove TLB, byva oddelena zvlast pro datove stranky a instrukcni stranky (i7 - dve urovne TLB, zvlast je uroven datova a instrukcni, pote je spolecna TLB urovne 2)

Dalsi moznosti **optimalizace prace s TLB:**

- globalni stranky - stranky jadra, ktere jsou sdilene mezi jednotlivymi procesy jsou oznacene jako globalni a nemusi se vyhazovat pri prepnuti kontextu,
- vstupy TLB spojene s identifikatory procesu - krome cisla stranky, ramce bude na jednotlivych radcich TLB bude zde proces,
- pouziva se spekulativni dopredne nahravani prekladu TLB,
- pouzivaji se specializovane cache (krome TLB) pro ukladani polozek (urovni) tabulek stranek

Zanorene hierarichicke tabulky stranek (Intel/AMD):

- pri virtualizaci vznikaji zanorene tabulky stranek (tabulky stranek host OS, tabulky stranek virtualniho OS),
- v pripade 64 bit procesoru se bude pouzivat 8 urovni tabulek stranek (4 pro virtualni pc, dalsi 4 pro hosta),
- pouziva se tak odliseni polozek v TLB pouzivane na fyzickem stroji pro ruzne virtualni strome - je zde identifikator pro virtualni pc (VPID na Intelu nebo ASID na AMD)

12.5.9 Priklad na 4-urovnove tabulce stranek

Chci provest 8 bajtovou instrukci, ktera bude nacitat 8 bajtu z pameti. **Kolik pristupu do pameti se v nejhorsim pripade provede?**:

- instrukce a data mohou byt na jinych strankach, oboje samostatne mohou byt na rozmezi 2 stranek (4b 1. stranka, 4b 2.), navic kazdou stranku nemusim mit fyzicky v prostoru za sebou (nelze to nacist jednim ctenim - ctu ze 2 ramcu),
- jen pro nacteni instrukce a dat budou potreba 4 pristupy do pameti (pouze jde o samotne cteni instrukce a dat! - nutny jeste preklad),
- preklad se provadi pro kazdou stranku zvlast pres 4-urovnovou tabulku stranek,
- provadet se budou 4x4 pristupy (4 stranky, 4 urovne) pro preklad adres, tzn. dalsich 16 pristupu,
- celkem tak 16 + 4 = 20 pristupu do pameti

12.5.10 Hashovane tabulky stranek

- logicka adresa clenena na cislo stranky p, posuv stranky d,
- MMU si vede prekladovou tabulku, ktera ma charakter hash tabulky, ve specialnim registru bude odkaz na zacatek hash tabulky,
- vezme se cislo stranky, prozene se hashovaci funkci (ta je implementovana v HW, MMU) - vypadne odkaz do hashovaci tabulky (cislo radku hash tabulky),
- problem - vice ruznych stranek se muze namapovat na stejnou polozku v hashovaci tabulce,
- neni zde tak primo umisteny preklad, ale odkaz na zretezeny senzam prekladovych polozek (cislo stranky, ramce),
- tento seznam MMU musi projit a dohledat pripadny preklad,
- az najdeme odpovidajici polozku, najdeme ramec, ten umistime do adresy misto cisla stranky, prida se posuv - yaay fyzicka adresa,
- efektivita zavisi na delce zretezenych seznamu - pokud bude hash funkce spatna - efektivita bude spatna,

Da se modifikovat napr.:

- nemusi se pouzivat cele cislo stranky pro hashovani,
- je mozne pouzit jen nekolik bytu stranek pro rozliseni ruznyh hash tabulek

12.5.11 Dalsi modifikace hashovanych tabulek stranek

Fixni pocet prekladovych polozek:

- zretezeny seznam prekladovych polozek byva nahrazen fixnim poctem prekladovych polozek, ktere se ukladaji do hashovaci tabulky na dany radek (namisto aby v tabule byl odkaz na seznam, tabulka bude ”sirsi”a na 1 radku bude 4-8 predkladovych polozek a hleda se v ramci radku),
- pokud preklad nebude nelazen, neznamena to, ze stranka neni mapovana, pouze se posle preruseni, jadro zjisti ze se nepodarilo dohledat na danem radku, proto musi jit do tabulek stranek, ktere si vede ve vlastni rezii (SW tabulek stranek),
- zde si dohleda, jestli preklad existuje (pokud ne = nastane vypadek stranky),
- pokud existuje - upravi se tabulka stranek tak, aby tam dane cislo bylo,
- pouziva se napr u CPU PowerPC (superpc) nebo na CPU Itanium (polozky mohou byt zretezeny, ale hw zretezeny seznam nepouziva - pokud je prvni polozka preklad, pouzije se, pokud ne - preruseni, jadro se podiva jestli ma nejaky dalsi seznam prekladovy polozek a ciste sw se dohleda preklad, pokud ho najde, provede se uprava tabulky stranek a novy preklad)

Hashovana tabula stranek muze byt sdilena vsemi procesy:

- je nutne do prekladovych polozek umistovat cislo stranky ale i odpovidaji cislo procesu,
- hashovaci funcki se prozene cislo stranky i cislo procesu,
- dohledava se dle cisla stranky i procesy,

- namisto cislo procesu se muze jeste pracovat s cisly pametoveho regionu, kazdy proces ma sva cisla regionu, lokalni cisla regionu se mapuji na cisla globalni - umozuje sdileni regionu (lokalni regiony procesu se muzou namapovat na 1 fyzicky region - sdileny), adresa je dana cislem regionu a cislem stranky,
- krome cisla stranky mam jeste lokalni region v LA, region se prevede na globalni cislo regionu, cislo stranky se prozene hash funkci a bude se vyhledavat - pouziva se u PowerPC a Itanium

definice:

pametove regiony - LAP je delen na stranky a na vyssi urovni je delen na regiony - skupiny stranek, ktere nasleduji za sebou, jsou promenne velikosti (neco jako extenty), jsou pouzity za urcitym ucelem (datovy, kodovy,..)

12.5.12 Priklad na sdilene hasovaci tabulce stranek s regiony

LAP je rozdelen na stranky, ty jsou **deleny na regiony** (napr. 6 stranek - region 1 ma prvni 3 stranky, region 2 ostatni),

- LA = lok. cislo regionu, cislo stranky, posuv v ramci stranky,
- lokalni cislo regionu se prelozi pres tabulku na globalni cislo regionu, to se spoji s cislem stranky,
- posuv se nemeni,
- glob. cislo regionu a cislo stranky se prozene hashovaci funkci,
- dostanu odkaz do hash tabulky, na urcity radek,
- rozdelena na urcity (pevny) pocet zaznamu,
- vzdy je tam cislo regionu, cislo stranky, odpovidajici ramec, potom dalsi cislo regionu, stranky, ramec, atd.,
- prohledavam, jestli se nekde v prislusne polozce nachazi dvojice globalni cislo regionu a cislo stranky,
- k tomu najdu odpovidajici ramec

12.5.13 Invertovana tabulka stranek

- nepreklada cislo stranek na cislo ramce, ale obracene - cislo ramce na cislo stranky,
- mapuje ramce na stranky,
- jeji radky odpovidaji ramcum, je zde tolik polozek, kolik mam ramcu,
- tabulka je nutne sdilena - pro vsechny procesy,
- je to pole,
- hashovani je resene hardwarove

Preklad:

- v polozkach tabulky se krome cisla stranky uklada i PID procesu,
- vezmu PID, cislo stranky,

- prohledavam od zacatku do konce,
- zjistuji, jestli nektery radek odpovida PID ktere me zaijma,
- a odpovida strance kterou hledam,
- nalezl jsem preklad,
- odpovidajici ramec ma hodnotu i, kde i je radek tabulky, na kterem jsem odpovidajici preklad nasel,

Vyhody:

- uspora pameti (1 tabulka pro vsechny)

Nevyhody:

- prohledavani od zacatku do konce je neakceptovatelne - prilis pomale - resi se kombinaci s hashovanim,
- neni nejpouzivanejsi,
- komplikace s tim, jak implementovat sdileni stranek - pokud vice procesu bude sdilet nejaky ramec, stejne polozce bude odpovidat vice dvojic PID a cislo stranky, jenze muze zde byt jen jedna dvojice - neustale bude dochazet k vypadkum,
- jadro si musi (paralelne) tak vest klasicke tabulky stranek, temi projde, zjisti ze preklad je mozny, opravi ho, bude chtit prvni preklad a jedna polozka se bude vyhazovat,
- jeste krome stranek jsou zde regiony, indexovat se pomoci cisla regionu a cisla stranky

Kombinace s hashovanim:

- misto hledani od zacatku do konce,
- se vezme PID a cislo stranky - prozenu hashovaci funkci,
- dostanu odkaz do tabulky,
- prohledavam od tohoto mista,
- typicky zretezeni je uvnitr tabulky

12.5.14 Priklad na invertovane tabulce stranek s hashovanim

- mam PID procesu, jeho LA = cislo stranky a posuv v ramci stranky,
- vezmu PID a cislo stranky, prozenu hash funkci,
- dostanu odkaz na nejaky radek invertovane tabulky stranek (v MMU je odkaz na zacatek),
- na danem radku bude ulozeno, pro jaky proces je mapovani, pro jakou stranku je mapovani, a odkaz na dalsi stranku procesu
- pokud PID a cislo stranky nesouhlasi s danym radkem, je zde odkaz na jiny radek, kde ma dany proces jinou stranku,
- pokud najdu odpovidajici PID a cislo stranky,
- nalezli jsme odpovidajici ramec,

- je umisteny - nikde, cislo ramce je cislo polozky - radek v tabulke

13

Posledni prednaska: Dokonceni spravy pameti.

13.1 Strankovani a segmentace na zadost

PAE - page adress extension - intel, rozsireni na 2^36 bitu fyzicke adresy, kde 4 horni bity nastavoval OS. To umoznilo **vyuzit az 64 GiB pameti na 32b** systemech (z hlediska procesu ale bylo mozne pouzit maximalne 4 GiB / proces).

Virtualizace pameti umoznuje procesu a jadru pracovat **s oddelenymi linearnimi logickymi adresovymi prostory**. (kazdy proces vidi svuj prostor, mezi sebo nekoliduji) Pro jednotlive procesy jsou pristupy **transparentni - nevi o tom, ze v jeden okamzik, cast pouzivane FAP pouziva napr. jiny proces**.

Vyhodou je **mensi spotreba pameti, rychlejsi odkladani na disk, zavadeni do pameti** (neni nutne odlozit ci zavest cely adresovy prostor procesu).

Cast pameti **lze odlozit na disk a v pripade potreby opet nahrat do PC**. (z disku se casti LAP zavadi do FAP pouze tehdy, pokud je to nutne).

Hovorime pak o:

- strankovani na zadost,
- segmentovani na zadost.

13.2 Strankovani na zadost

Stranky **jsou zavadeny do pameti, jen pokud k nim pristupujeme**. OS se stara o to, ze uchovava **informace** o tom, ktere stranky jsou vyuzite a ktere ne (resp. v tabulce stranek - flag - bit, urcuje jestli je stranka v pameti nebo na disku).

Pokud po vyhledani cisla stranky (a prislusneho ramce) **je stranka ulozena v pameti**, postupuje se normalne (prelozi se na FA). Pokud ne, **posle se preruseni OS (trap) - jedna se o vypadek stranky** (page fault).

U jinych tabulek stranek (hash, invertovanych) **hw se podiva do seznamu stranek**, pokud tam stranka nebude, OS se podiva jeste **do svych sw tabulek stranek**. Pokud vsak zjisti, ze se pristupuje na LA, ktera neni namapovana v pameti - dojde k **vypadku stranky**.

Vypadek stranky **je preruseni od MMU, udava ze nelze prevest adresu** == neni definovano mapovani v tabulce stranek.

13.3 Obsluha vypadku stranky

- kontrola, zda proces neodkazuje mimo prideleny adresovy prostor (pokud ano - segfault, jadro proces ukonci), alokace ramce,
  - proces pristupuje tedy do pameti, ktera neni v FA namapovana,
  - pouzije se volny ramec, pokud nejaky volny je,
  - pokud neni - vybereme si stranku v pameti, ktera ma jiz prideleny ramec (victim page - obet), odlozime obet na disk, pokud byla stranka zmenena, pokud zmenena nebyla (uz je na disku), uvolnime ramec a pouzijeme uvolneny ramec
- inicializace stranky (po alokaci zavisla na predchozim stavu stranky),

  - pokud jde o prvni odkaz na stranku - pokud je to kod ci inicializovana data - nacte se z programu, vse ostatni - data se nevynuluji (kvuli bezpecnosti),
  - pokud to nebyl prvni pristup - stranka uz byla v minulosti uvolena z FAP - pokud to je kod ci konstantni data - nactou se z programu, ostatni - pokud byla modifikovana, ze swapu se stranka vrati zpet do FAP, jinak se obsah opet vynuluje
- uprava tabulky stranek - upravit odkaz, kam vede LA (ramec se zmeni),
- proces je pripraven na opakoveni instrukce, ktera vypadek zpusobila (je ve stavu pripraveny)

13.4 Vykonnost strankovani na zadost

Efektivni doba pristupu do pameti:

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

- kde p je page fault rate = pravdepodobnost vypadku stranky ($(1-p)$ je pravdepodobnost ze k vypadku nedojde),
- T doba pristupu bez vypadku,
- D doba pristupu s vypadkem

Doba pristupu bez vypadku je mnohem mensi nez doba pristupu s vypadkem. - zavisi na tom **lokality odkazu v procesech, vhodny vyber zavadenych ci odkladanych stranek** (algoritmus vyberu "obeti"), **dostatek pameti, jemu prideleny pocet procesu, ..** - snaha mit p co nejmensi.

13.5 Pocet vypadku stranek

Mame 1 instrukci, kolik vypadku pri jejim zpracovani muze nastat ? Muze dojit pri **cteni instrukce, pri praci s kazdym z jejich operandu**, u obou muze dojit k vypadku **vicenasone**.

Vicenasobne vypadky mohou byt zpusobeny:

- nezarovnanim instrukce (instrukce se nachazi pulka v 1 strance a druha pulka ve 2. strance),
- nezarovnanim dat (jako instrukce vyse),
- data jsou delsi nez 1 stranka (instrukce pracujici s mnoho daty, napr. na Intelu MOVSB),
- vypadky tabulek stranek na ruznych urovich - napr. hiearachicka tabulka stranek, kdy tabulky mohou byt velke, proto se odkladaji do pameti a muze opet dochazet k vypadkum - obvykle alespon cast tabulek stranek je chranena pred vypadkem stranek (zejmena tabulka stranek nejvyssi urovne)

Priklad (podobny u zkousky):

- Jaky je maximalni procet vypadku stranek v systemu se strankami o velikosti 4 KiB, 4-urovnovou tabulkou stranek, u ktere pouze dilci tabulka nejvyssi urovne je chranena proti vypadku, pri provadeni nenactene instrukce o delce 4 B, ktera presouva 8 KiB z jedne adresy pameti na jinou?
- instrukce bude na rozmezi dvou stranek
  - pri pristupu na 1 stranku je nutne projit celou hierarchickou tabulku stranek (mame 4 urovne, 1 chranena),
  - v tab 1. urovne je odkaz na tab. 2. urovne, zde k vypadku muze dojit (+1), tu je odkaz na tab 3. urovne, stejny case (+1), stejne u 4. urovne (+1), to, kam ukazuje tab. 4. urovne (na FAP) opet nemusi byt v pameti (+1)
  - pro zpracovani 1 casti instrukce muze dojit ke 4 vypadkum stranek,
  - zpracovani 2. casti - 1. stranka bude uplne na konci vsech tabulek stranek - pri inkrementaci se musi tak
  - nacistuplne nove 4 urovne tabulek stranek - opet muze dojit k 4 vypadkum jako predtim
  - celkem 4 + 4 = 8 vypadku po nacteni instrukce
- instrukce se musi spustit, presouva 8 KiB dat (pri 4 KiB strankach),
  - nejhorsi pripad - src data o velikosti 8 KiB jsou rozlozena na 3 strankach (2 KiB 1 stranka, 4 KiB druha, 2 KiB treti),
  - dst data budou obdobne,
  - abych dostal prvni stranku src, muze opet dojit ke 4 vypadkum,
  - pri praci s druhou strankou - opet prvni je na konci seznamu, tedy +1 bude uplne v jinych tabulkach - dalsi +4 vypadky,
  - posledni stranka - protoze LA nasleduji za sebou, prvni dve stranky byly na rozmezi (konec - zacatek tabulek), potom +1 nyni bude ve stejnych tabulkach stranek jako predchozi - tedy +1 vypadku stranek (je mozny pouze vypadek na urovni ramce)
  - celkem +9 vypadku
  - u dst uplne stejny pripad jako u src - tedy +9 vypadku

  - provadeni instrukce zabere tak 9+9 vypadku = 18 vypadku,

- tedy dohromady pri vykonani jedne teto instrukce muze dojit celkem 8 + 18 = 26 vypadku stranek (maximalni mozny pocet, nejhorsi pripad)

13.6 Odkladani stranek

K tomuto muze dojit pri vypadku stranky. Muze byt odlozeni:

- lokalni - v ramci procesu (u ktereho doslo k vypadku),
- globalni - bez ohledu na to, kteremu procesu patri ktera stranka

Typicky je neustale udrzovan **urcity pocet volnych ramcu**:

- pokud pocet volnych ramcu klesne pod urcitou mez, aktivuje se page daemon (zlodej stranek), ktery bezi tak dlouho, dokud neuvolni dostatecny pocet stranek,
- pri vypadku stranky se pozuje ramec z mnoziny volnych ramcu,
- lze doplnit heuristikou, ktera uvolene stranky okamzite neprideluje, ale zjistuje, jestli nebyla vybrana obet, ktera neni spravna

13.7 Algoritmy vyberu odkladanych stranek (obeti)

13.7.1 FIFO

- first in first out,
- odstranuje stranku, ktera byla zavedena do pameti pred nejdelsi dobou (a nebyla dosud odstranena),
- napr. proces pracujes se 4 strankami (1,2,3,4), chce patou, neni dost pameti - uvolni se stranka 1, ramec kde byla 1 se pouzije pro 5,
- vyhody - jednoducha implementace,
- problemy:
  - muze odstranit starou stranku, ktera se ale casto pouziva,
  - trpi Beladyho anomalii - ocekavame, ze v systemu, ve kterem casto dochazi k odkladani a zvetsi se pamet systemu, tak k odkladani bude dochazet mene casto - to zde pravda byt nemusi - pocet vypadku vzroste pri zvetseni pameti,
- da se trochu zmenit - omezit problem s odstranovanim starych ale pouzivanych ramcu - umistime takovy ramec do mnoziny volnych ramcu, pridelime jiny volny ramec, pokud bychom chteli ramec pouzit, ihned se ziska tento uvolneny ramec (signal - spatna obet, vybere se jina), pokud ne - odlozi se

13.7.2 LRU

- least recently used,
- snazi se okladat nejdele nepouzitou stranku,

- dobra aproximace hypotetickeho idealniho algoritmu (znal by budoucnost a podle pozadavku z budoucnsti by rozhodoval, co aktualne odlozit tak, aby byl pocet vypadku minimalni - zatim to nejde)
- problemy:
  - pri cyklickych pruchodech poli se algoritmus muze chovat velmi pomalu - napr. system se 4 ramci, delam bubble sort na 5 pametovych blocich (prvni 4 bloky budou na vsech ramcich, zpracuji posledni - prvni vyhodim a dam na prvni ramec, pote potrebuji neco pro prvni blok - vyhodim druhy ramec, ... porad dokola ..)
  - resi se to napriklad strategii odstraneni naposledy pouzite stranky (MRU - most recently used)
  - dalsi problem je problematicka implementace - je nutne umet detekovat pouzitou stranku (nutna HW podpora, napr. casove razitko)
- pouzivaji se aproximace LRU

Aproximace LRU **pomoci omezene historie referencniho bitu stranek** (page againg):

- pouzije se jeden bit - referencni bit,
- tento bit hw nastavi pri kazdem pristupu na 1,
- jadro si vede omezenou historii tohoto bitu (pro jednotlive stranky),
- periodicky se posouva obsah historie doprava (aka: shiftuje se referencni bit doprava),
- pote se referencni bit v tabulce stranek vynuluje,
- pote pri vyberu obeti jadro projde vsechny historie a vybere z nich nejmensi hodnotu (zn. pouzivala se stranka pred nejdelsi dobou)

Aproximace LRU **algoritmem druhe sance**:

- stranky jsou ulozeny v kruhovem seznamu,
- mame jeden ukazatel, ten seznam prochzaime,
- vynulujeme referencni bit dane stranky,
- pokud uz bude 0, pouzijeme danou stranku jako obet,
- take se tomu rika clock algorithm

Modifikace algoritmu druhe sance:

- uprednostnuji se jako obeti nemodifikovane strane (usetri se 1 zapis na disk) - modifikovane se zapisi na disk a dostanou dalsi sanci,
- dva ukazatele, ktere prostupuji frontou - jeden nuluje referencni bit, druhy odstranuje obeti (double-handed clock algorithm),
- linux:
  - fronty aktivnich a neaktivnich stranek - pokud nejaka stranka je behem 1 periody zpristupnena 2x, tak se presune do fronty aktivnich stranek, pokud ne, prehodi se do fronty neaktivnich stranek (z te se vybiraji obeti),

  - system se snazi odkladat stranky, ktere jsou pouzite pro ruzne cache, pokud maji procesy namapovany urcity pocet
  - stranek, OS se snazi odstranovat jeho neaktivni stranky,
  - odkladani se provadi po urcitych poctech (najednou), cim vice je pamet zaplnena, tim vice se ramcu uvolni,
  - je mozne nastavit procesu priznak, aby se jeho stranky neodkladaly (nebyly obeti),
  - pri kritickem nedostatku pameti jadro ukoncuje nektere procesy (vetsinou: pokud dojde RAM i swap zaroven) - pro
  - spoustu pripadu to byva ale az pozde,
  - v linuxu je zde sluzba EarlyOOM (neni sluzba jadra!) - zacne tyto procesy ukoncovat driv pri nedostatku pameti

13.8 Alokace ramcu procesum (resp. jadru)

- dulezite hlavne u lokalniho vyberu,
- u globalniho vyberu lze pouzit pro rizeni vyberu obeti,
- je treba mit vzdy pridelen minimalni pocet ramcu pro provedeni 1 instrukce - jinak dojde k nekonecnemu vymenovani stranek potrebnych k provedeni instrukce,
- dale se pouzivaji ruzne heuristiky pro urceni poctu ramcu pro procesy:
  - podle velikost programu, priority, objemu fyzicke pameti, ..,
  - na zaklade pracovni mnoziny stranek - mnozina stranek, ktere pouziva proces za nejakou urcitou dobu (aproximace s pomoci referencnciho bit),
  - sledovani frekvence vypadku procesu (vice vypadku, casteji - proces dostane vice fyzicke pameti)

Pridelovani ramcu **s vyuzitim pracovni mnoziny, kombinace lokalni a globalni vymeny vyuzivaji nektere systemy Windows**:

- procesy a jadro maji jisty minimalni a maximalni porcet ramcu
- pokud je dost pameti, ramce se jim dodavaji az do dosaseni maxima (muze se i zvetsit),
- pokud se dosahne maxima a neni dost pameti, provede se lokalni vymena,
- pokud je volnych ramcu vyrazny nedostatek, OS zacne prochazet bezici procesy a zacne jim odebirat urcity pocet stranek na zaklade omezene historie pristupu
- pri vyberu obeti se dava prednost procesum bezicim mene casto,
- snazi se vyhybat tem procesum, ktere bezi na popredi ci tem procesum, ktere mely v posledni dobe mnoho vypadku,
- pocatecni meze pracovnich meze se zjisti pri startu systemu dle velikosti fyzicke pameti,
- pri odkladani na disk vybrane obeti ji da jeste 2. sanci - necha si ji jeste chvili v pameti, pridaji ji jiny ramec, aby bylo mozne korigovat chyby pri volbach obeti,

- umi swapovat vsechna data procesu na disk (typicky dlouho neaktivni proces)

13.9 Trashing

Problem, ktery muze nastat pri odkladani stranek na disk. Vznikne pokud mame **system s vysokou mirou paralelismu** (resp. mnoho spustenych procesu) - procesy potrebuji nejaky **minimalni pocet ramcu, se kterymi musi pracovat**. Pokud je pocet procesu velky, pameti **je malo a casto muze dochazet k odkladani na disk**. Potom muze dojit k tomu, ze **proces stravi vice casu nahradou stranek nez uzitecnym vypoctem**.

Obcas v nejakych OS je **swapper** (sluzba OS), ten pozastavi nektere procesy a odlozi veskerou pamet na disk. Nebo je moznosti ukoncit nektere procesy.

13.10 Poznamky

Prepaging - do fyzicke pameti zavadi vice stranek zaroven (start procesu, po odswapovani, .. - zrychleni).

Zamkykani stranek:

- zabranuje se odlozeni,
- uziva se napr. u stranek, do nichz probiha I/O (stranky se mohou mapovat do zarizeni - nechceme tyto stranky ukladat na disk), u (casti) tabulek stranek (napr. nejvyssi uroven), u (nekterych) stranek jadra, na prani uzivatele (napr. v pameti pracujeme s citlivymi daty)

Sdileni stranek Da se sdilet:

- kod programu (proces vicekrat spusteny, neni treba mit stejny kod v pameti nekolikrat nebo sdilene knihovny),
- konstantni data ci doposud nemodifikovana data u kopii procesy (copy-on-write technologie),
- mechanismus IPC (2 procesy mohou sdilet stejnou pamet, pouzivaji ji treba pro synchronizaci),
- sdileni pametove mapovanych souboru

Sdilene knihovny

- ulozene ve FAP pouze jednou,
- .dll ci .so,
- vyhody - mensi programy (neni nutne do programu dat vsechny funkce, cast muze byt v knihovne), lepsi vyuziti prostoru na disku, moznost aktualizovat knihovny,
- nevyhody - zavislost programu na dalsich souborech a verzich knihoven, mozny pomalejsi start programu (knihovna zas muze byt jiz v pameti), mozne pomalejsi volani funkci (kompilator zde nemuze provadet optimalizace - v knihovne, navic volani musi jit pres sestavovaci tabulky)

Copy-on-write

- pri spusteni fork se nevytvori kopie veskere pameti procesu,
- tedy mam dve LA, ale
- oba procesy sdileji stejne ramce v FAP, ty jsou oznaceny jako copy-on-write,

- znamena, ze stranky se sdili mezi procesy, dokud jeden z procesu do nich nezapise

Sdilena pamet

- shared memory,
- forma IPC, vice procesu ma mapovany stejne fyzicke stranky do LAP

Pametove mapovane soubory

- cely soubor se namapuje do LAP,
- procesy si soubor namapuji do sveho LAP, pracuji s nim jakoby pristupovali do pameti,
- vyuziti strankovani na zadost pro praci se soubory,
- pouziti napr. pri praci s databazemi,
- dostupne napr. prostrednictvim mmap();

Pametove regiony

- jednotka vyssiho strukturovani pameti v UNIXu,
- v ramci pametoveho reginu je kazda adresa (region pro kod, haldu, zasobnik, data, ..),
- pouzivaji segmentove registry, neukazuji na pocatky segmentu, ale o jaky region jde (pro zasobnik, kod, ..),
- umoznuje jisty druh prace s danou pameti,
- napr. region s kodem - je mozne zakazat zapis

Standardni rozlozeni adresoveho prostoru procesu v Linuxu (i386):

- region na adrese 0 - nepouzity,
- nad nim je text segment - kod programu (provadeni kodu, zadny zapis),
- segment se statickymi inicializovanymi daty (modifikace, ale nespoustet),
- prostor pro neinicializovane staticke promenne (zaplneno nulami),
- halda (heap) - prace pres volani malloc pres C napr., roste od nejake adresy nahoru,
- pamet pouzita na mapovani LAP na sdilene ramce pameti ci sdilena pametova mista,
- zasobnik (stack),
- nejvyse je prostor namapovany pro funkce kernelu - kernel space

THE END