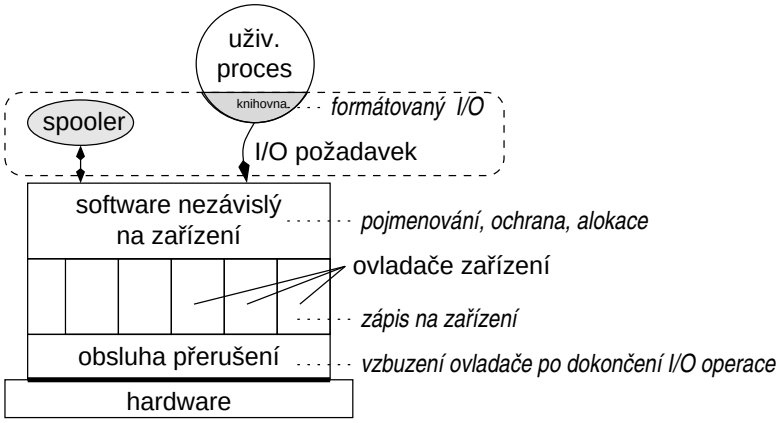
KIV/ZOS 2003/2004
Přednáška 11 a 12

Principy vstupně/výstupního software
====================================

* I/O SW v OS je typicky strukturován do 4 úrovní
 - každá úroveň má určenou úlohu
 - úroveň (vrstva) má rozhraní s vyššími a nižšími vrstvami

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



1. Obsluha přerušení
----

* řadič vyvolá přerušení ve chvíli dokončení I/O požadavku
* obvykle je snaha, aby se přerušením nemusely zabývat vrstvy na vyšší úrovni
* typicky ovladač zadá I/O požadavek, usne (např. pomocí P(sem) apod.),
  po příchodu přerušení ho obsluha přerušení vzbudí (např. V(sem))
* obsluha přerušení je časově kritická, musí být co nejkratší

2. Ovladače zařízení
----

* ovladače obsahují veškerý kód závislý na I/O zařízení
 - ovladač jediný zná HW podrobnosti: způsob komunikace s řadičem
   zařízení, ví o sektorech a stopách disku, pohybu diskového raménka,
   start & stop motoru
 - ovladač ovládá všechna zařízení daného druhu nebo případně třídu
   příbuzných zařízení (např. ovladač SCSI disků ovládá všechny SCSI
   disky)

* funkce ovladače:
 - ovladači předán příkaz vyšší vrstvou, např. ”zapiš data do bloku n”
   nebo ”přečti data z bloku n”
 - pokud ovladač ještě obsluhuje předchozí požadavek, zařadí nový požadavek
   do fronty
 - přijde-li požadavek na řadu:
  . ovladač zadá příkazy řadiči (nastavení hlavy, přečtení sektoru apod.)
  . zablokuje se do vykonání požadavku (např. pomocí P(sem))
    (pokud je ale operace rychlá, např. pouze zápis do registru,
    neblokuje se)
  . po dokončení operace (vzbuzení obsluhou přerušení) zkontroluje, zda
    nenastala chyba
  . pokud OK, předá výsledek (status + případná data) vyšší vrstvě
    (status - obvykle datová struktura pro hlášení chyb)
  . pokud jsou další požadavky ve frontě požadavků, jeden z nich se
    vybere a spustí

* ovladače často vytvářejí výrobci HW, proto musí být dobře definované
  rozhraní mezi OS a ovladači
* ovladače podobných zařízení mají stejné rozhraní
  (např. v Linuxu: bloková, síťové karty, zvukové karty, ..., znaková apod.)

3. SW vrstva OS nezávislá na zařízení
---

* poskytuje I/O funkce společné pro všechna zařízení daného druhu
  (např. společné fce pro všechna bloková zařízení)
* definuje rozhraní s ovladači
* poskytuje jednotné rozhraní uživatelskému SW

* typicky poskytuje následující funkce:
 - pojmenování zařízení (v DOSu: LPT1, UNIX: /dev/lp0) atd.
 - ochrana zařízení (přístupová práva)
 - alokace a uvolnění vyhrazených zařízení
   (některá zařízení, jako tiskárna, plotter, magnetická páska atd.
   mohou být v jednu chvíli použity pouze jedním procesem)
 - vyrovnávací paměti
  . bloková zařízení - čtou/zapisují data v blocích pevné délky
  . pomalá zařízení - čtení/zápis může probíhat z/do bufferu
 - hlášení chyb
 - jednotnou velikost bloku pro bloková zařízení

* v moderních OS (UNIX a Linux, Windows NT) se zařízení jeví jako objekty v
  souborovém systému (přesněji řečeno: v mnoha OS je tato vrstva součástí
  logického souborového systému, viz dále)

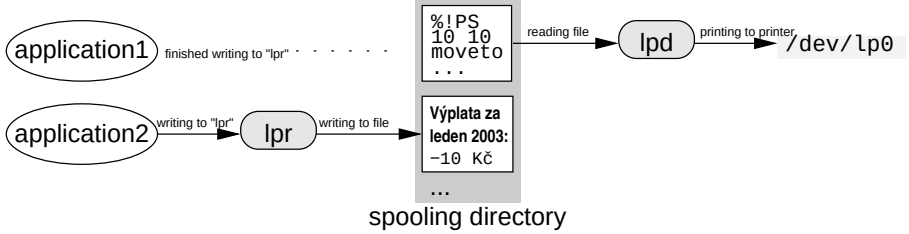
4. I/O SW v uživatelském režimu
---

* programátor používá v programech I/O funkce nebo příkazy jazyka
 - např. printf v C, writeln v Pascalu = knihovny sestavené s programem
 - provádějí formátování - printf(”%.2d:%.2d\n”, hodin, minut)
 - často mají vlastní vyrovnávací paměť velikosti 1 blok

* spooling - implementován pomocí procesů běžících v uživatelském
  režimu
 - spooling = způsob obsluhy vyhrazených I/O v multiprogramovém systému
 - ”co kdyby proces alokoval zařízení a pak s ním hodinu nic nedělal?”

Např. tisk v systémech typu UNIX:

* k tiskárně má přístup pouze jeden speciální proces - daemon ”lpd”
* proces vygeneruje celý soubor, daemon ”lpd” ho vytiskne
* přesněji
 - proces, který chce tisknout, spustí program ”lpr” a naváže s ním
   komunikaci
 - proces předává tisknutá data programu ”lpr”, ten zapíše data do souboru v
   určeném adresáři (spooling directory; k datům v adresáři mohou přistupovat
   pouze lpr a lpd)
 - po dokončení zápisu program ”lpr” oznámí daemonu lpd, že soubor je
   připraven pro vytisknutí, deamon soubor vytiskne a zruší



spooling directory

* spooling používán i v dalších situacích, např. pro přenos elektronické
  pošty

Souborové systémy

* téměř všechny aplikace potřebují trvale uchovávat data
* hlavní požadavky:
 - možnost uložit velké množství dat
 - informace musí zůstat zachována i po ukončení procesu
 - data musejí být přístupná více procesům

* proto vznik magnetických pásek, disků atd.
* při přístupu k zařízení mají všechny procesy společné problémy, např.:
 - alokace prostoru na disku
 - pojmenování dat
 - ochrana dat před neoprávněným přístupem
 - zotavení po havárii (např. po nečekaném výpadku napájení)

* OS pro přístup k médiím poskytuje abstrakci od fyzických vlastností
  média - soubor
 - soubor = pojmenovaná množina souvisejících informací

* pojem ”souborový systém” (file system, dále jen fs) se používá ve
  dvou významech:
 . konvence pro ukládání a přístup k souborům (datové struktury
   a algoritmy)
 . část OS, která poskytuje mechanismus pro ukládání a přístup k datům,
   implementuje danou konvenci.

Současné OS obvykle implementují více fs z důvodů kompatibility
se staršími verzemi a s ostatními OS:

Např.:

* Windows 2000: základní fs je NTFS
 - ostatní podporované: FAT12, FAT16, FAT32, ISO 9660 (pro CD-ROM média)...
* Linux: ext2 (základní fs), případně ext3, ReiserFS, JFS, XFS
 - ostatní podporované: FAT12 až FAT32, ISO 9660, Minix, VxFS, OS/2 HPFS,
   SysV fs, UFS, NTFS (pouze read only)...

Poznámka pro zajímavost (historický vývoj)

 * v prvních systémech vstup z děrných štítků, výstup na tiskárnu;
   pojmem ”soubor” se míní množina děrných štítků
 * později magnetické pásky: vstup z pásky, výstup na pásku
   pojmem ”soubor” se míní množina štítků nebo množina záznamů na
   magnetické pásce
 * nyní data uchovávána většinou na magnetických a optických discích
   definice (ISO 2382-4:1987):
  - soubor: pojmenovaná posloupnost záznamů, které lze zpracovávat jako
            celek
  - záznam: strukturovaný datový objekt tvořený konečným počtem
            pojmenovaných položek

 * takto strukturované soubory najdeme např. v jazyce Pascal

[]

Uživatelské rozhraní fs

* vlastnosti fs z pohledu uživatele:
 - konvence pro pojmenování souborů
 - vnitřní struktura souboru
 - typy souborů
 - způsob přístupu
 - atributy a přístupová práva
 - služby OS pro práci se soubory.

Konvence pro pojmenování souborů

* při vytvoření souboru proces určuje jméno souboru
* pravidla pro vytváření jmen se poněkud liší mezi OS

* např. Windows NT vs. UNIX a Linux
 - rozlišuje systém mezi malými a velkými písmeny?
  . Windows NT (Win32 API) nerozlišuje: ”ahoj” == ”Ahoj” == ”AHOJ”
  . UNIX/Linux rozlišuje: ”ahoj”, ”Ahoj” a ”AHOJ” jsou rozdílná jména
 - jaká může být délka názvu souboru?
  . Windows NT 256 znaků (NTFS)
  . UNIX/Linux obvykle alespoň 256 znaků (podle typu fs)
 - množina znaků?
  . ve všech běžných systémech mohou názvy souborů obsahovat písmena a číslice
  . Windows NT: znaková sada UNICODE

   tj. βετα je legální jméno souboru ve Windows NT

 - Linux: jméno může obsahovat všechny 8 bitové znaky (kromě ’/’ a char(0))
* přípony?
 - v MS DOSu jméno souboru 8 znaků + 3 znaky přípona
 - Windows NT, UNIX: možno více přípon
* další omezení?
 - Windows NT: mezera nesmí být první a poslední znak apod.

Typy souborů

Mnoho OS podporuje více typů souborů, např.:

* obyčejné soubory - obsahují data zapsaná aplikacemi
 - obvykle rozlišení alespoň na textové nebo binární
 - textové: řádky textu, ukončeny znaky CR (Mac), LF (UNIX), nebo CR+LF (MS DOS, Windows)
 - binární - všechny ostatní; OS rozumí struktuře spustitelných souborů

* adresáře - systémové soubory, které udržují strukturu fs

* v Linuxu/UNIXu ještě:
 - znakové speciální soubory, blokové speciální soubory - rozhraní pro se vstupně/výstupními zařízeními (např. /dev/lp0 = tiskárna)
 - pojmenované roury - pro komunikaci mezi procesy
 - symbolické odkazy atd.

Dále budeme mluvit převážně o obyčejných souborech.

Vnitřní struktura (obyčejného) souboru

* 3 časté způsoby:
 - soubor je nestrukturovaná posloupnost bytů
 - soubor je posloupnost záznamů
 - soubor je strom záznamů

* nestrukturovaná posloupnost bytů
 - OS obsah souboru nezajímá, interpretace souboru je na aplikacích
 - maximální flexibilita - programy mohou soubory strukturovat jak chtějí

* posloupnost záznamů pevné délky, každý záznam má vnitřní strukturu
 - operace čtení vrátí záznam, operace zápisu změní/přidá záznam
 - používáno v historických systémech: např. soubory se záznamy 80 znaků obsahovaly obraz děrných štítků...
 - v současných systémech se téměř nepoužívá

* strom záznamů, záznamy nemusejí mít stejnou délku
 - každý záznam obsahuje pole klíč (na pevné pozici v záznamu)
 - záznamy seřazeny podle klíče, aby bylo možné rychle vyhledat záznam s požadovaným klíčem

 - používá se na velkých mainframech pro komerční zpracování dat
   (případně pro některé systémové soubory)

Způsob přístupu k souboru
....

* sekvenční přístup
 - procesy mohou číst data pouze v pořadí, v jakém jsou uloženy
   v souboru (čtou postupně od prvního záznamu, nemohou přeskakovat)
 - možnost ”přetočit” a číst opět od začátku (viz fce rewind() v C)
 - v prvních OS, kde data uchovávána na magnetických páskách

* přímý přístup (random access file)
 - čtení v libovolném pořadí nebo podle klíče
 - přímý přístup nutný např. pro databáze
 - jak určit kde začít číst?
  . každá operace určuje pozici
  . OS udržuje pozici čtení/zápisu, novou pozici lze nastavit speciální
    operací ”seek”

* v některých OS pro mainframy se při vytvoření souboru určilo, zda je
  sekvenční nebo s přímým přístupem (OS mohl používat rozdílné
  strategie uložení souboru)
* všechny současné OS pouze soubory s přímým přístupem

Atributy
....

* informace sdružená se souborem
* některé atributy interpretuje OS, jiné systémové programy a aplikace
* významně se liší mezi jednotlivými OS

* ochrana souboru, např. kdo je vlastník souboru, množina přístupových
  práv, heslo apod.
* příznaky - určují vlastnosti souboru, např. hidden (soubor se
  neobjeví při výpisu), archive (soubor nebyl zálohován), temporary
  (soubor bude automaticky zrušen), read-only, text/binary, random
  access ... jako návrháři OS si můžeme vymyslet spoustu dalších
* pokud se k záznamu přistupuje pomocí klíče, pak délka záznamu, pozice
  a délka klíče
* velikost; datum vytvoření, poslední modifikace, posledního přístupu
  apod.

Služby OS pro práci se soubory
----

* většina současných OS základní model podle systému UNIX
* základní filosofie UNIXu - ”méně je někdy více”
=> několik jednoduchých pravidel:

* veškerý vstup/výstup je prováděn pouze pomocí souborů
  - obyčejné soubory - data, spustitelné programy...
  - zařízení - disky, tiskárny, ...
  - zacházení se všemi typy pomocí stejných služeb systému
* obyčejný soubor je uspořádaná posloupnost bytů
  - význam dat znají pouze programy, které soubor čtou/zapisují
  - interní struktura souboru OS nezajímá
* jeden typ souboru je seznam souborů - adresář
  (jinými slovy: adresář je také soubor)
  - soubory a adresáře jsou koncepčně umístěny v adresáři
* ”speciální soubory” pro přístup k zařízením (v DOSu PRN:, COM1: apod.)

Poznámka pro zajímavost:

 Výše uvedená pravidla se mohou zdát samozřejmá, ale před příchodem
 systému UNIX samozřejmá nebyla:

 * většina systémů před UNIXem měla samostatné služby pro čtení/zápis
   terminálu, zápis na tiskárnu, čtení/zápis do souboru

```
 * mnoho systémů před i po UNIXu mnoho různých druhů souborů s různou
   strukturou a metodami přístupu
 * tyto systémy poskytovaly ”více služeb”, model podle UNIXu však má
   nezanedbatelnou výhodu - podstatně menší složitost
 * proto téměř všechny moderní systémy základní rysy modelu převzaly

[]

Základní služby pro práci se soubory
....................................

* než začneme se souborem pracovat, musíme ho otevřít
 - je-li otevření úspěšné, vrátí služba pro otevření souboru malé celé
   číslo - popisovač souboru (file descriptor)
 - popisovač souboru používáme v dalších službách - při čtení ze
   souboru apod.

* otevření souboru: fd = open(jméno-souboru, způsob)
 - jméno-souboru - řetězec pojmenovávající soubor
 - způsob - pouze pro čtení, pouze pro zápis, čtení i zápis
 - fd - vrácený popisovač souboru (file descriptor)
 - otevření souboru nalezne informace o souboru na disku a vytvoří pro
   soubor potřebné datové struktury OS
 - popisovač souboru = index do tabulky souborů uvnitř OS

* vytvoření souboru: fd = creat(jméno-souboru, přístupová-práva)
 - vytvoří nový soubor s daným jménem a otevře ho pro zápis
 - pokud soubor existoval, zkrátí ho na nulovou délku
 - fd - vrácený popisovač souboru (file descriptor)

* operace čtení ze souboru: read(fd, buffer, počet-bytů)
 - přečte zadaný počet-bytů ze souboru fd do bufferu
 - může přečíst méně, pokud v souboru zbývá méně (konec souboru -
   přečte 0 bytů)

* operace zápisu do souboru: write(fd, buffer, počet-bytů)
 - význam parametrů stejný jako u read
 - uprostřed - přepíše, konec - prodlouží
 - read i write vrací počet skutečně přečtených/zapsaných bytů
 - operace read() a write() jediné operace pro čtení a zápis
 - samy o sobě poskytují sekvenční přístup k souboru

* nastavení pozice v souboru: lseek(fd, offset, odkud)
 - nastaví offset příští čtené/zapisované slabiky souboru
 - odkud = od začátku souboru, od konce souboru, od aktuální pozice
 - poskytuje přímý přístup k souboru

* zavření souboru: close(fd)
 - uvolní datové struktury alokované OS pro soubor


Příklad použití rozhraní - kopírování souboru:

Poznámka (pro neznalé jazyka C)

 * rozhraní UNIXu definováno pro jazyk C, proto je příklad v jazyce C
   v jazyce C   -> lze převést do jazyka Pascal
 - { }          -> begin/end
 - /* */        -> (* *)
 - x = y        -> x := y
 - x == y       -> x = y

* použity fce open, creat, read, write, close.

----------------------[ fragment kódu ]------------------------------
        int src, dst, in;

        src = open(”původní”, O_RDONLY); /* otevření zdrojového */
        dst = creat(”nový”, MODE); /* vytvoření cílového */
```

```
        while (1) /* while true = nekonečná smyčka */
        {
                in = read(src, buffer, sizeof(buffer)); /* čteme */
                if (in == 0) /* konec souboru? */
                {
                        close(src); /* zavřeme soubory */
                        close(dst);
                        return; /* skončíme */
                }
                write(dst, buffer, in); /* zapíšeme přečtená data */
        }
----------------------[ konec fragmentu ]-----------------------------
```

Další služby pro práci se soubory
.....

* další služby pro změnu přístupových práv, zamykání apod.
* v tom se systémy liší, protože nejvíce závislé na konkrétních
  mechanismech ochrany apod.

* např. systémy UNIX:
 - zamykání fcntl(fd, cmd)
 - zjištění informací o souboru (typ, přístupová práva, velikost, ...)
   stat(file_name, buf), případně fstat(fd, buf)

Paměťově mapované soubory
.....

* někteří návrháři OS se domnívají, že jakékoli rozhraní typu
  open/read/write/close je nepohodlné oproti přístupu k paměti

* některé OS - např. UNIX a Linux - proto poskytují možnost mapování souboru
  do adresního prostoru procesu (služby systému mmap(), munmap())
 - mapovat je možné i jen část souboru

Příklad:

 * představme si OS umožňující mapování souborů do paměti
 * délka stránky je 4 KB
 * máme soubor délky 64 KB, chceme ho mapovat do adresního prostoru
   od 512 KB (512*1024=524 288, tj. mapujeme od adresy 524 288)
 * po mapování: 0-4 KB souboru bude mapováno na adresu 512 KB až 516 KB atd.
 * čtení z adresy 524 288 čte byte 0 souboru, z následující adresy byte 1 atd.

[]

* implementace paměťově mapovaných souborů:
 - OS použije soubor jako odkládací prostor (swapping area) pro určenou
   část virtuálního adresního prostoru
 - čtení/zápis na výše uvedenou adresu 524 288 způsobí výpadek stránky,
   do rámce se načte obsah první stránky souboru
 - pokud je modifikovaná stránka vyhozena (při nedostatku volných rámců),
   zapíše se do souboru
 - po skončení práce se souborem se zapíší všechny modifikované stránky

* problémy paměťově mapovaných souborů
 - není známa přesná velikost souboru, nejmenší jednotka je stránka
 - problém nekonzistence pohledů na soubor, pokud je zároveň mapován a
   zároveň se k němu přistupuje klasickým způsobem

Adresářová struktura
---

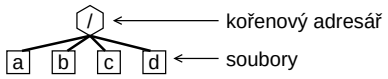
* jedna oblast (partition) obsahuje jeden fs
* fs se skládá ze 2 součástí:
 - množina souborů, obsahujících data
 - adresářová struktura, udržuje informace o všech souborech v daném fs

Adresář překládá jméno souboru na informace o souboru (umístění,
velikost, typ...)

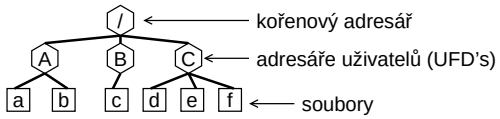
Základní požadavky na adresář:
* procházení souborovým systémem (přepnu se do adresáře)
* výpis adresáře (jaká jména jsou v adresáři?)
* vytvoření a zrušení souboru
* přejmenování souboru

Nejčastější schemata logické struktury adresářů:
(odpovídá vývoji v OS)

* jednoúrovňový adresář
 - např. původní verze MS DOSu
 - nejjednodušší struktura - všechny soubory jsou v jediném adresáři
 - všechny soubory musejí mít jedinečná jména
 - problém zejména pokud více uživatelů

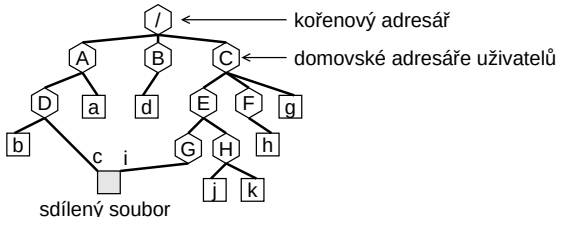


* dvouúrovňový adresář
 - adresář pro každého uživatele (User File Directory, UFD)
 - OS prohledává pouze UFD, nebo - pokud je specifikováno - adresář jiného
   uživatele (např. [user]file)
 - problém se systémovými příkazy - jsou to spustitelné soubory; pro ně
   zavedeme speciální adresář (pokud se příkaz nenalezne v adresáři
   uživatele, vyhledá se v systémovém adresáři)



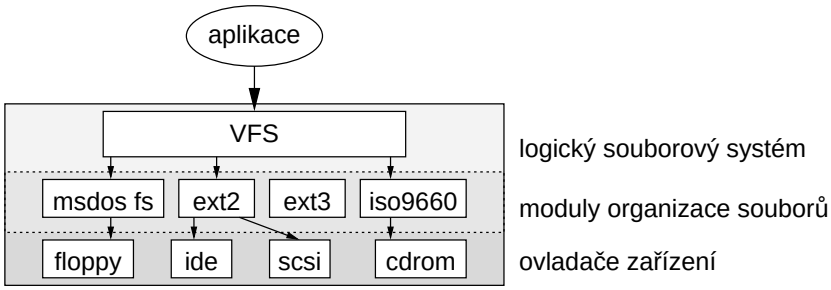
* adresářový strom
 - zobecnění dvouúrovňového adresáře, inspirováno zejména UNIXem
 - dnes nejčastější, např. DOS, Windows NT
 - adresář (podadresář) obsahuje množinu souborů a adresářů
 - souborový systém začíná kořenovým adresářem ”/”
   (MS: ”\”, protože v původním MS DOSu se znak ”/” používal pro volby)
 - cesta k souboru (jméno-souboru v open, creat) absolutní nebo relativní
 - absolutní:
  . začíná od kořenovým adresářem a vyjmenovává adresáře, kterými
    musíme projít, abychom se dostali k souboru (poslední část je název
    souboru)
  . jednotlivé adresáře odděleny znakem ”/”
  . příklad: /home/luki/vyuka/zos/pr/p8mm.txt
 - relativní:
  . aplikace většinou přistupují k souborům v jednom adresáři
  . defaultní prefix = pracovní adresář
  . použije se, kdykoli cesta nezačíná znakem ”/”
  . příklad: soub.txt nebo prednasky/soubor.txt - přidá se pracovní
    adresář /home/luki -> /home/luki/soub.txt.

* Acyklický graf adresářů
 - např. dva programátoři, spolupracující na společném projektu,
   oba by chtěli mít společný podadresář ve svém adresáři
 - sdílení společného souboru nebo podadresáře - stejný soubor nebo
   podadresář může být vidět ve dvou různých adresářích
 - flexibilnější než strom, komplikovanější
  . rušení souborů a adresářů - kdy můžeme zrušit? (až když již už není v
    žádném adresáři; se souborem sdružen počet odkazů z adresářů, při každém
    remove(soubor) snížen, 0 = není odkazován)
  . jak zajistit, aby graf byl acyklický?
    (existují algoritmy pro zjištění, zda je cyklus, ale drahé pro fs)

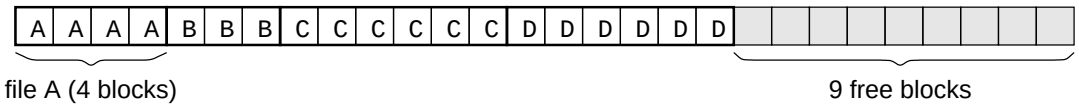



```
* Obecný graf adresářů
 - je obtížné zajistit, aby graf byl acyklický
 - problém
  . prohledávání grafu - nejčastěji omezení počtu prošlých adresářů
    (Linux)
  . rušení souboru - pokud je cyklus, může být počet odkazů > 0 i když
    soubor již není přístupný
    => garbage collection: projít celý fs, označit všechny přístupné
       soubory, zrušit nepřístupné; drahé, pro fs zřídka používáno

* nejčastější adresářový strom (MS DOS...)
* UNIX už od původních verzí acyklický graf - hard links - sdílení pouze
  souborů - nemohou vzniknout cykly

Základní služby pro práci s adresáři
....................................

Dnes téměř všechny systémy podle systému UNIX:

* pracovní adresář - související služby:
 - nastavení pracovního adresáře: chdir(adresář)
 - zjištění pracovního adresáře: getcwd(buffer, počet-znaků)

* práce s adresářovou strukturou:
 - vytváření a rušení adresářů: mkdir a rmdir
   . mkdir(adresář, přístupová-práva)
   . rmdir(adresář) - musí být prázdný
 - zrušení souboru: remove
   . remove(jméno-souboru)
 - přejmenování souboru: rename
   . rename(jméno-souboru, nové-jméno-souboru)
   . pokud je požadováno, provádí také přesun mezi adresáři
 - čtení adresářů - UNIX/POSIX:
   . DIRp = opendir(adresář) - otevře adresář
   . položka = readdir(DIRp) - čte jednotlivé položky adresáře
   . closedir(DIRp) - zavře adresář
   . stat(jméno-souboru, statbuf) -

Služby pro čtení adresářů opendir()... vznikly později, než ostatní
=> různých systémech odlišně s podobnou sémantikou (DOS:
   findfirst/findnext ...)



Implementace souborových systémů
================================

* problémy:
 - jak bude fs vypadat pro uživatele?
 - jaké algoritmy a datové struktury použít pro implementaci?
 => různá úroveň abstrakce

V dnešních OS je implementace souborů rozdělena nejméně do 3 vrstev, a to na
logický souborový systém, modul organizace souborů, a ovladače zařízení.

* logický souborový systém (v některých OS ”virtuální souborový
  systém”, VFS)
 - je volán aplikacemi
 - tvoří rozhraní s moduly organizace souborů
 - obsahuje kód společný pro všechny typy fs
```

```
 - převádí jméno souboru na informaci o souboru
 - udržuje informace o otevřeném souboru (otevřen pro čtení/zápis,
   pozice čtení/zápisu)
 - zodpovědný za ochranu a bezpečnost (ověřování přístupových práv)

* modul organizace souborů
 - implementuje konkrétní souborový systém
 - čte/zapisuje datové bloky souboru
   . datové bloky souboru jsou číslovány sekvenčně od 0 do N-1
   . převod převod čísla bloku souboru na diskovou adresu
   . volání ovladače pro čtení/zápis bloku
 - správa volného prostoru + alokace volných bloků
 - údržba datových struktur fs

* nejnižší úroveň - ovladače zařízení
 - interpretují požadavky typu: přečti logický blok 6456 ze zařízení 3

Příklad (OS Linux):
```



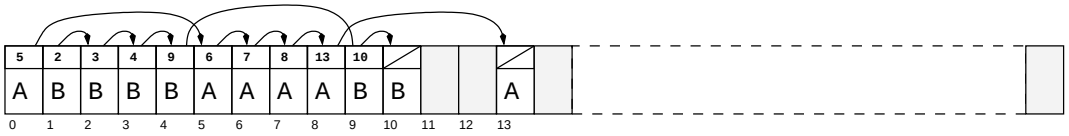

```
Implementace souborů
--------------------

* nejdůležitější otázka - které diskové bloky patří kterému souboru
* probereme pouze nejzákladnější metody

* v dalším textu budeme předpokládat, že fs coby datová struktura
 - je umístěna v nějaké oblasti (disk partition)
 - bloky v oblasti jsou očíslovány od 0

Kontinuální alokace
...................

* nejjednodušší - soubor jako kontinuální posloupnost diskových bloků
* např. při blocích velikosti 1 KB by soubor A o velikosti 4 KB
  zabíral 4 po sobě následující bloky atd.
```




```
* jednoduchá implementace: potřebujeme znát pouze číslo prvního bloku
  a celkový počet bloků souboru (může být obsaženo např. v adresáři)
* čtení je velmi rychlé - přesuneme hlavičku na začátek souboru, čtené
  bloky jsou za sebou

* problém - při běhu OS soubory vznikají a zanikají, případně mění
  velikost
 - na začátku můžeme zapisovat soubory sériově do volného místa na konci
 - po jeho zaplnění budeme potřebovat využít volné místo po zrušených
   souborech
 - abychom vybrali vhodnou díru, potřebovali bychom znát konečnou délku
   souboru - což je informace, kterou většinou nemáme

* kontinuální alokace se dnes používá pouze na read-only a write-once
  médiích
* například v základní verzi souborového systému ISO 9660 pro CD ROMy
```

Seznam diskových bloků
....

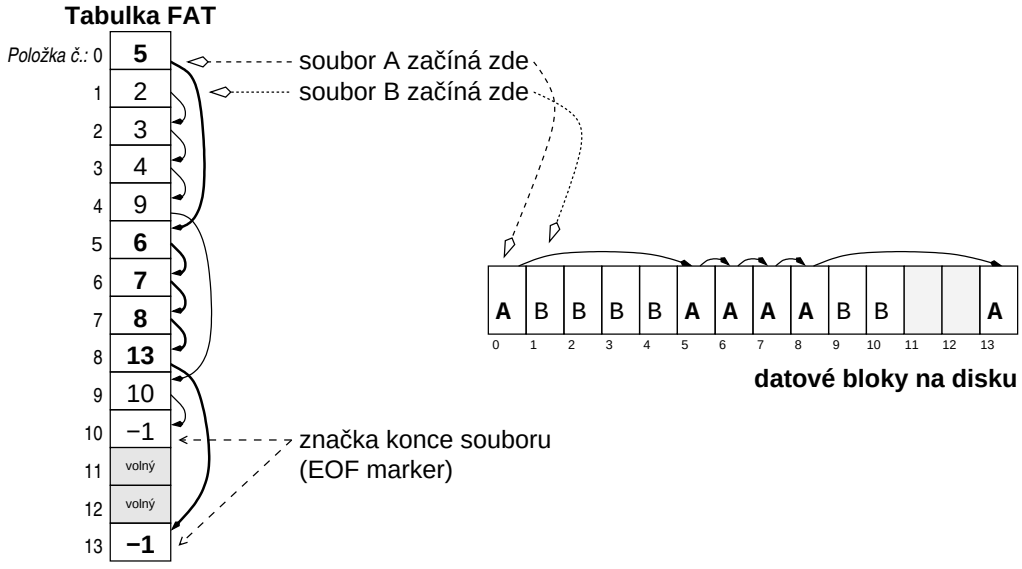
* myšlenka svázat diskové bloky do seznamu - nebude vnější fragmentace
* na začátku diskového bloku je uložen odkaz na další blok souboru,
  zbytek bloku obsahuje data souboru



* pro přístup k souboru stačí znát pouze číslo prvního bloku souboru
  (může být v adresáři)
* sekvenční čtení bez potíží, ale přímý přístup je simulován pomocí
  sekvenčního - pomalé
* někdy může být nevýhodou, že velikost dat v bloku není mocnina 2
  (část bloku je zabrána odkazem na další blok souboru)

FAT
...

* výše uvedené nevýhody seznamu diskových bloků lze odstranit
  přesunutím odkazů do samostatné tabulky
* tato tabulka se nazývá FAT (File Allocation Table)
 - každému diskovému bloku odpovídá jedna položka v tabulce FAT
 - položka FAT obsahuje číslo dalšího bloku souboru (je to zároveň
   odkaz na další položku FAT)
 - řetězec odkazů je ukončen speciální značkou, která není platným
   číslem bloku

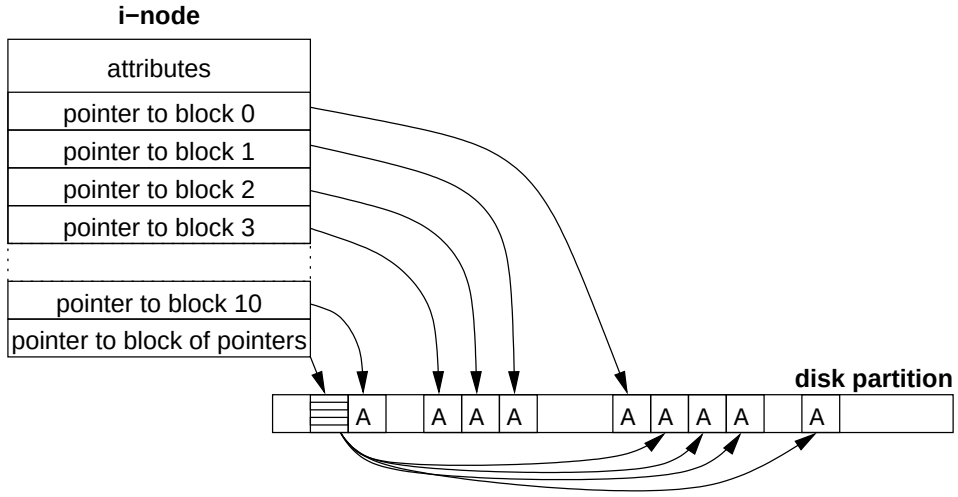


* nevýhodou je velikost tabulky FAT: předpokládejme 80 GB disk, bloky
  velikosti 4 KB => 20 mil. položek, každá alespoň 3 byty => 60 MB FAT
 - to má důsledky pro výkonnost (FAT nebudeme udržovat celou v paměti)

* FAT se používá v MS DOSu, ..., Windows 98, Windows me, z důvodů
  kompatibility podporují i Windows NT/2000/XP
 - podle velikosti položky v tabulce FAT:
  . 12 bitů = FAT12 - omezení na 2^12 (= 4096) bloků, původně pro diskety
  . 16 bitů = FAT16 - omezení na 2^16 (= 65536) bloků
  . FAT32 - omezení na 2^28 bloků (blok 4-32 KB, tj. teoreticky do 8 TB)

I-uzly
......

* s každým souborem sdružená datová struktura i-uzel (angl. i-node,
  zkratka z index-node)

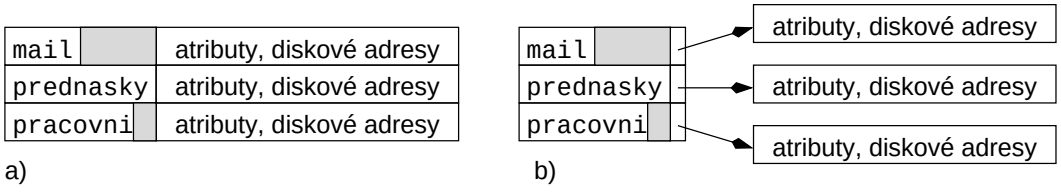


* i-uzel obsahuje atributy souboru, diskové adresy prvních N bloků a
  jeden nebo více odkazů na diskové bloky obsahující další diskové adresy
  (případně obsahující odkazy na bloky obsahující adresy)
* výhoda - po otevření souboru můžeme zavést i-uzel a případný blok
  obsahující další adresy do paměti => urychlí přístup k souboru
* používají tradiční fs v UNIXu a Linuxu (dále)

Implementace adresářů
----

* před čtením musí být soubor otevřen
* otevření voláním open(jméno, jak) - zadáme jméno, potřebujeme převést
  na informaci potřebnou pro nalezení dat
* mapování jméno -> informace o datech poskytují adresáře

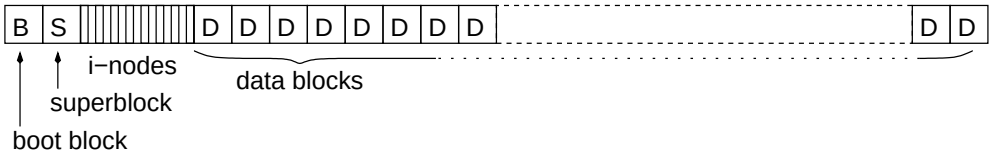
* v mnoha systémech jsou adresáře speciálním typem souboru
* typicky pole datových struktur, 1 položka na soubor
* 2 základní uspořádání:
 - adresář obsahuje jméno souboru, atributy, diskovou adresu souboru
   (v závislosti na implementaci souborů např. adresu prvního bloku apod.)
 - adresář obsahuje pouze jméno + odkaz na jinou datovou strukturu
   obsahující další informace (např. i-uzel)
 - běžné jsou oba způsoby i kombinace



* způsob (a) implementuje např. MS DOS a Windows, způsob (b) je
  používán v UNIXu a Linuxu

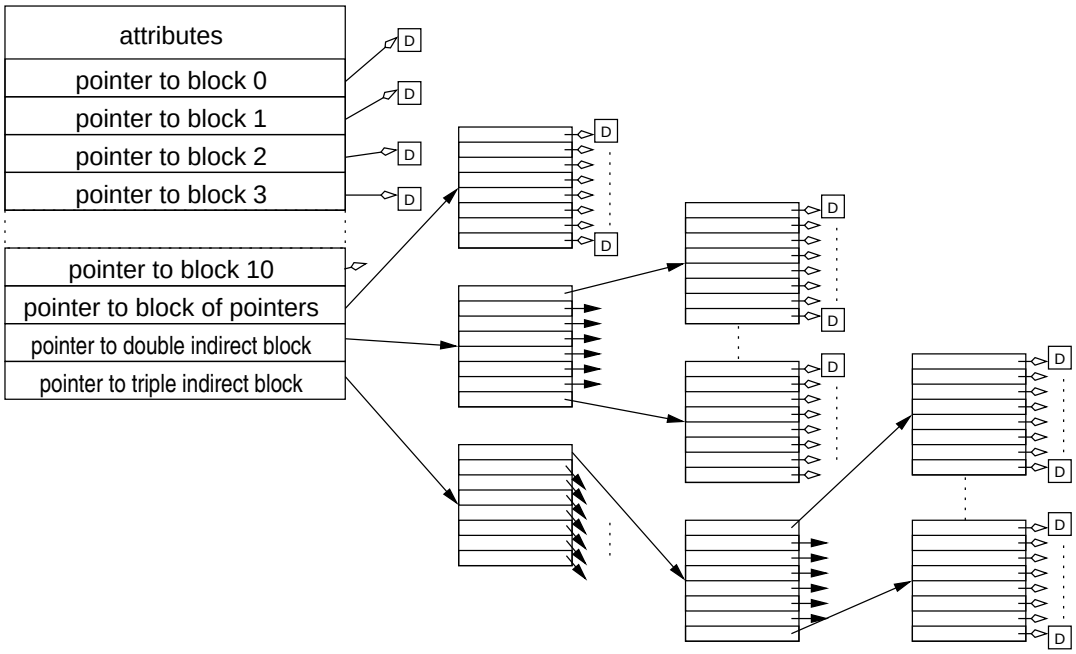
Příklad souborového systému (fs z UNIXu v7)

 * struktura fs na disku
  - boot blok - fs nevyužívá, často kód pro zavedení OS
  - superblok - informace o fs: počet i-uzlů a datových bloků apod.
  - i-uzly - tabulka pevné velikosti, i-uzly číslovány od 1
  - datové bloky - všechny soubory a adresáře

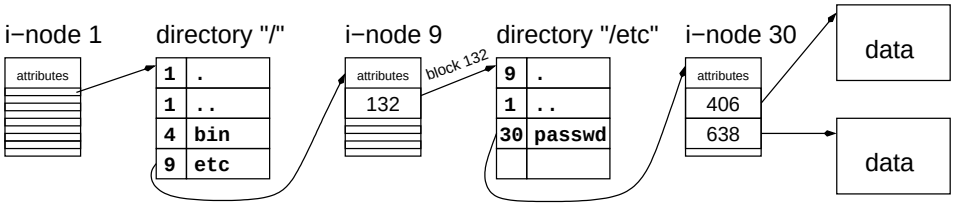



```
 * implementace souborů - i-uzly
  - i-uzel obsahuje:
    . atributy
    . odkazy na prvních 10 datových bloků souboru
    . odkaz na blok obsahující odkazy na datové bloky (nepřímý odkaz)
    . odkaz na blok obsahující odkazy na bloky obsahující odkazy na
      datové bloky (dvojitě nepřímý odkaz)
    . trojitě nepřímý odkaz
```

UNIX v7 i−node




```
 * implementace adresářů - tabulka obsahující jméno souboru a číslo
   jeho i-uzlu
 * informace o volných blocích - seznam, jeho začátek je v superbloku

 * popsaný fs se používal v nejstarších verzích OS UNIX, je velmi
   jednoduchý ale má problémy s efektivitou
 * moderní fs UNIXu a Linuxu (ufs, ext2 apod.) jsou založeny na
   podobných základních myšlenkách (i-uzly apod.), ale eliminují
   nevýhody (stručný popis ufs je v JUU na str. 164)

Adresáře v systému UNIX v7:

 * adresář obsahuje pouze jméno souboru a číslo i-uzlu
 * číslo i-uzlu je indexem do tabulky i-uzlů na disku
 * každý soubor a adresář má právě jeden i-uzel
 * v i-uzlu jsou všechny atributy a čísla diskových bloků
 * kořenový adresář má číslo i-uzlu 1
```



```
 * nalezení cesty k souboru /etc/passwd podle obrázku:
  - v kořenovém adresáři najdeme položku "etc"
  - i-uzel č. 9 obsahuje adresy diskových bloků pro adresář "/etc"
  - v adresáři "/etc" (diskový blok 132) najdeme položku "passwd"
  - i-uzel 30 popisuje soubor "/etc/passwd"

[]

Příklad pouze pro zajímavost (adresáře v systému Windows 98)

  * položka adresáře obsahuje:
   - jméno souboru (8 bytů) + příponu (3 byty)
   - atributy (1)
   - NT (1) - Windows 98 nepoužívají (rezervováno pro WinNT)
   - datum a čas vytvoření (5)
   - čas posledního přístupu (2)
   - horních 16 bitů počátečního bloku souboru (2)
   - čas posledního čtení/zápisu
   - spodních 16 bitů počátečního bloku souboru (2)
   - velikost souboru (4)
  * dlouhá jména mají pokračovací položky

  * veškeré "podivnosti" této struktury jsou z důvodu kompatibility
    s MS DOSem
[]

Sdílení souborů
...............

* pokud chceme, aby byl soubor nebo adresář viditelný ve více
  podadresářích nebo pod více jmény - 2 způsoby implementace

* pokud má každý soubor datovou strukturu, která ho popisuje (např.
  i-uzel), můžeme v adresářích vytvořit více odkazů na stejný soubor
  (tzv. pevné odkazy, hard links)
 - všechny odkazy (=všechna jména souboru) jsou rovnocenné
 - v popisu souboru (např. v i-uzlu) musí být počet odkazů (při vytvoření
   souboru bude počet odkazů 1, počet odkazů nutno zvětšit při vytvoření a
   zmenšit při zrušení odkazu)
 - soubor zanikne po zrušení posledního odkazu (tj. pokud počet odkazů = 0)

* vytvoříme nový typ souboru, který bude obsahovat jméno odkazovaného
  souboru (symbolický odkaz); OS místo symbolického odkazu otevře
  odkazovaný soubor
 - obecnější (symbolický odkaz může obsahovat cokoli), ale má větší režii

Správa volného prostoru
.......................

* jak udržovat informaci, které bloky jsou volné?
* nejčastěji pomocí seznamů nebo pomocí bitových map

* seznam: seznam diskových bloků, blok obsahuje odkazy na volné bloky a
  adresu dalšího bloku (který opět obsahuje obsahuje odkazy na volné
  bloky a adresu dalšího bloku seznamu)
 - uvolnění bloků - přidáme adresy do seznamu, pokud není místo blok
   zapíšeme
 - potřebujeme bloky pro soubor - používáme adresy ze seznamu, pokud
   nejsou přečteme další blok adres volných bloků
 - pokud není na disku volné místo, seznam volných bloků je prázdný a
   nezabírá místo
 - problém pokud pro soubor potřebujeme volný blok s určitými vlastnostmi
   (např. ve stejném cylindru jako předchozí blok souboru), museli bychom
   seznam prohledávat což je drahé (seznam je uložen na disku!)
```

```
* bitová mapa - má konstantní velikost, snazší vyhledání volného bloku
  s určitými vlastnostmi
* většina současných fs používá bitovou mapu

Kvóty
.....

* aby uživatelé nezabrali příliš prostoru, může správce každému
  uživateli nastavit kvótu = maximální počet bloků obsazených soubory
  uživatele
* ve víceuživatelských OS, OS pro servery (UNIX, Windows 2000...)

* implementace: [ v roce 2003 vynecháno ]

Spolehlivost fs
...............

* ztráta dat je většinou větší katastrofa než zničení počítače
 - pokud vám do počítače praští blesk nebo přijdou záplavy, dá se v
   nejhorším případě koupit nový
 - vaše ztracená data nekoupíte (diplomová práce, databáze zákazníků...)
* fs musí být nejspolehlivější část OS, snaží se chránit data
 - správa vadných bloků (viz předchozí přednáška)
 - některé fs se snaží rozprostřít důležité datové struktury tak, aby
   fs byl (alespoň částečně) čitelný i po poškození konkrétního povrchu
   (např. ufs v UNIXu)

Konzistence fs
..............

* OS přečte blok souboru, změní ho, zapíše
* co když nastane havárie před tím, než jsou všechny modifikované bloky
  zapsány? => fs může být v nekonzistentním stavu
* s většinou OS se dodává systémový program, který kontroluje
  konzistenci fs (UNIX: fsck, Windows: scandisk resp. chkdsk)
* program je automaticky spuštěn při startu po havárii systému

* program může provádět následující testy konzistence fs:
 - konzistence informace o diskových blocích souborů (ve většině fs
   může blok patřit pouze jednomu souboru nebo být volný)
 - konzistence adresářové struktury (jsou všechny adresáře a soubory
   dostupné? apod.)

* kontrola konzistence informace o diskových blocích souborů
 - program vytvoří dvě tabulky, obsahující čítač pro každý blok
  . tabulka počtu výskytů bloku v souboru
  . tabulka počtu výskytů bloku v seznamu volných bloků
  . všechny položky obou tabulek na počátku inicializovány na 0
 - program prochází informace o souborech (např. i-uzly) a přitom
   inkrementuje položky odpovídající blokům souboru v první tabulce
 - pak prochází seznam nebo bitmapu volných bloků a inkrementuje
   příslušné položky ve druhé tabulce
 - je-li fs konzistentní, bude mít každý blok 1 buď v první nebo ve
   druhé tabulce

* pro konzistentní fs budou tabulky vypadat např. takto:

        číslo bloku: 0  1  2  3  4  5  6  7  8  9 11 12 13 14 15
 výskyt v souborech: 1  1  0  0  1  0  0  1  1  1  1  0  1  0  0
        volné bloky: 0  0  1  1  0  1  1  0  0  0  0  1  0  1  1

* možné chyby a jejich závažnost:
 - 0-0: blok se nevyskytuje v žádné tabulce (”missing block”)
  . není závažné, pouze redukuje kapacitu fs
  . opravíme vložením do seznamu volných bloků
 - 0-2: blok je dvakrát nebo vícekrát v seznamu volných
  . problém, blok by mohl být alokován dvakrát
  . opravíme seznam volných bloků, aby se vyskytoval pouze jednou
 - 1-1: blok patří souboru a zároveň je na seznamu volných
```

```
  . problém, blok by mohl být alokován podruhé
  . blok vyjmeme ze seznamu volných bloků
 - 2-0: blok patří do dvou nebo více souborů
  . nejzávažnější problém, nejspíše už došlo ke ztrátě dat
  . alokujeme nový blok, problematický blok do něj zkopírujeme a
    upravíme i-uzel druhého souboru; uživatel by měl být informován o
    problému

Domácí úkol:

  Jaké chyby odhalíte v následujícím fs a jak jsou závažné?

        číslo bloku: 0  1  2  3  4  5  6  7  8  9 11 12 13 14 15
 výskyt v souborech: 1  2  0  0  1  0  0  1  1  1  1  1  1  0  0
        volné bloky: 0  0  1  1  0  0  1  0  0  0  0  1  0  1  1

[]

* kontrola konzistence adresářové struktury
 - tabulka čítačů, jedna položka pro každý soubor
 - program prochází rekurzivně celý adresářový strom
 - položku pro soubor program zvýší pro každý výskyt souboru v adresáři
 - pak zkontroluje zda odpovídá počet odkazů v i-uzlu (”i”) s počtem
   výskytů v adresářích (”a”)

* možné chyby:
 - i>a: soubor by nebyl zrušen ani po zrušení všech odkazů v adresářích
  . není závažné, ale soubor by zbytečně zabíral místo
  . řešíme nastavením počtu odkazů (v i-uzlu) na správnou hodnotu
 - i<a: soubor by byl zrušen po zrušení $i$ odkazů, ale v adresářích
    budou ještě jména
  . potenciálně katastrofální, položky v adresáři by odkazovaly na
    neexistující soubory
  . opět řešíme nastavením počtu odkazů na správnou hodnotu
 - a=0, i>0: ztracený soubor, na který není v adresáři odkaz
  . ve většině systémů program soubor zviditelní na předem určeném
    místě (např. adresář ”/lost+found” v Linuxu)

* různé další heuristické kontroly
 - odpovídají jména souborů konvencím OS?
  . pokud ne, soubor může být nepřístupný -> změníme jméno
 - nejsou přístupová práva nesmyslná?
  . pokud nemá vlastník přístup k souboru, může to být problém
 - atd.

Poznámka (journaling file systems)

 Kontrola konzistence fs může pro rozsáhlé disky trvat minuty, případně i
 hodiny. Jako částečné řešení tohoto problému vznikly moderní tzv.
 žurnálující fs (v Linuxu jsou to např. souborové systémy ext3 (což je
 žurnálující rozšíření ext2), ReiserFS a další).

 Žurnálující fs před každým zápisem na disk vytvoří na disku záznam
 popisující plánované operace; pak operace provede a záznam zruší. V případě
 výpadku napájení lze na disku najít informaci (žurnál) o všech operacích
 které mohly být v době havárie rozpracované, což značně zjednodušuje
 kontrolu konzistence fs.

[]

Výkonnost fs
............

* přístup k disku je řádově pomalejší než přístup do paměti
 - např. 5-10 ms bude trvat seek
 - pak rotační zpoždění - čekáme až bude požadovaný blok pod hlavou
 - pak se bude číst např. 10 MB/s (cca 40x pomalejší než přístup do
   hlavní paměti)
```

* proto při návrhu fs různé optimalizace, např.
 - cachování [kešování] diskových bloků v paměti
 - přednačítání (read-ahead) - do cache se předem načítají bloky, které se budou potřebovat při sekvenčním čtení souboru
 - redukce pohybu diskového raménka pro po sobě následující bloky souboru
 - atd.

Mechanismy ochrany

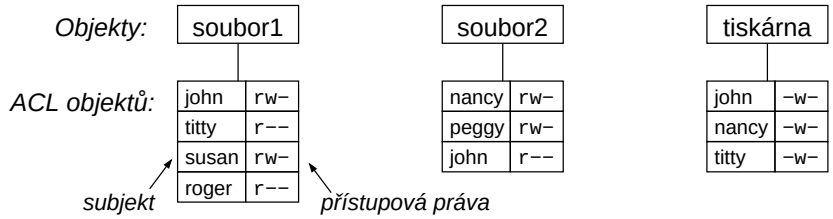
* soubory je třeba chránit před neoprávněným přístupem
* kromě souborů existují v OS další objekty, které je třeba chránit
  - může být HW (segmenty paměti, I/O zařízení),
  - SW objekty (procesy, semafory...)

* systém musí uchovávat informace o přístupových právech subjektů k objektům
 - subjekt = entita schopná přistupovat k objektům (většinou proces)
 - přístupové právo je právo vykonat jednu z možných operací, například Read, Write, Execute, Delete...
 - objekt = cokoli k čemu je třeba omezovat přístup pomocí přístupových práv

* informace o přístupových právech může být ve dvou různých podobách: ACL nebo capability list
 - ACL = s objektem sdružen seznam subjektů a jejich přístupových práv
 - capability list (čti [kejpa-]) = se subjektem je sdružen seznam objektů a přístupových práv k nim

Mechanismus ACL (Access Control Lists)

* s každým objektem sdružen seznam subjektů které mohou k objektu přistupovat (zde subjekt = uživatel, resp. jeho procesy),
 - pro každý uvedený subjekt je v ACL množina přístupových práv k objektu

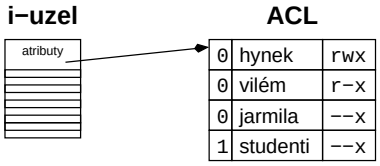


* mnoho operačních systémů umožňuje sdružovat subjekty do tříd nebo do skupin
 - například skupina na ZČU by skupina ”studenti” obsahovala všechny studenty, ”zaměstnanci” všechny zaměstnance apod.
 - skupiny mohou být uvedeny na místě subjektu v ACL
 - zjednodušuje administraci systému atd. (v ACL nemusím uvádět všechny studenty jmenovitě)

Mechanismus ACL používá mnoho moderních fs (NTFS ve Windows NT, XFS v Linuxu apod.)

Poznámka (kterak by bylo možné implementovat ACL pro UNIX):

 * základní fs pro UNIX a Linux mechanismus ACL nemají
 * jak by ho bylo možné doplnit?
  - v i-uzlu by byla část tabulky ACL, pokud by se nevešla celá do i-uzlu tak odkaz na diskový blok, obsahující zbytek ACL
  - každá položka ACL by obsahovala:
   . subjekt: číslo uživatele nebo číslo skupiny (uživatel a skupina jsou v UNIXu určeni čísly UID a GID) + 1 bit (rozlišení uživatel/skupina)
   . přístupová práva implementována N bitovým slovem, 1=právo přiděleno a 0=právo odejmuto



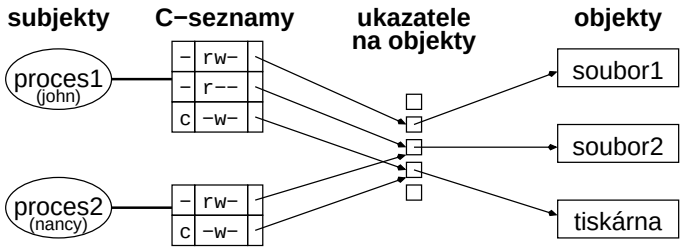
[]

Mechanismus capability lists (C-seznamy)
.....

* s každým subjektem (procesem) sdružen seznam objektů ke kterým může
  přistupovat a jakým způsobem (tj. přístupová práva)
* seznam nazýván ”capability list” (C-list = C-seznam) a jednotlivé
  položky ”capabilities”

* struktura ”capability” má prvky:
 - typ objektu
 - práva - obvykle bitová mapa popisující dovolené operace nad objektem
 - odkaz na objekt, např. číslo i-uzlu, číslo segmentu apod.

* problém - zjištění, kdo všechno má k objektu přístup + zrušení
  přístupu velmi obtížné - znamenalo by to najít pro objekt všechny
  capability + odejmout práva
* řešení - odkaz neukazuje na objekt, ale na nepřímý objekt; systém může
  zrušit nepřímý objekt, tím zneplatní odkazy na objekt ze všech C-seznamů



* C-seznamy začnou být zajímavé pokud jsou jediný způsob odkazu na objekt
  (capability chápána jako bezpečný ukazatel; název ”capability-based
  addressing)
 - ruší rozdíl mezi objekty na disku, v paměti (segmenty) nebo na jiném
   stroji (objekty by se dokonce mohly přesouvat za běhu)
 - proto mechanismus C-seznamů najdeme v některých distribuovaných systémech
   (např. Hydra, Mach apod.)

Mechanismus ACL se ovšem používá mnohem častěji.

Případová studie: operační systém Linux [v roce 2003 uvedeno pouze pro zajímavost]
==========

* volně šířený systém - kdokoli si ho může nainstalovat a dále šířit
* zdrojové texty - kdokoli může modifikovat jádro nebo kterýkoli
  systémový program
* vypadá a chová se jako UNIX - výhodou kompatibilita s UNIXem pro
  programátora (volně šířené knihovny apod.)

Systém Linux se skládá z následujících komponent:

* monolitické jádro OS
 - jediné běží v režimu jádra
 - poskytuje základní abstrakce (procesy, virtuální paměť, soubory
   apod.)
 - aplikace žádají o služby jádra prostřednictvím volání služeb systému
 - umožňuje za běhu přidávat a vyjímat moduly (poskytují možnost
   přidat/zrušit části kódu jádra na žádost - nejčastěji ovladače)

* systémové knihovny

- definují standardní množinu fcí pro použití v aplikacích
- některé knihovní fce musejí komunikovat s jádrem OS (např. všechny fce provádějící zápis do souboru, jako write(2), fwrite(3) apod.)
- mnoho knihovních fcí jádro nevolá (např. matematické fce, jako je sin(3), cos(3), tan(3), ...)

* systémové programy - programy provádějící specializované úlohy v systému, např.
  - nastavení některého konfigurace systému při startu (nastavení klávesnice loadkeys(1), rychlost sériového rozhraní setserial(8))
  - démonové procesy - běží po dobu běhu systému (inetd(8) - obsluha příchozích síťových spojení, syslogd(8) - zápis žurnálových souborů (log files)).

* uživatelské programy - např. ls(1), sh(1), oowriter(1) apod.

Knihovny i jádro mají ještě vlastní strukturování.

Poznámka (jádro versus distribuce)

Technicky se názvem ”Linux” myslí pouze jádro systému, protože všechny uživatelské programy, knihovny a většina systémových programů vznikla v nezávislých projektech (většina základních programů jako ls(1), cp(1) apod. pochází z projektu GNU).

Aby uživatelé nemuseli hledat a konfigurovat jednotlivé programy běžící pod jádrem systému Linux, vznikly tzv. distribuce, např. RedHat, SUSE, Debian apod., které obsahují:

* jádro
* programy z různých zdrojů (základní programy, grafické prostředí...)
* správu balíků (RedHat - RPM, Debian - dpkg...)

[]

Vnitřní implementace Linuxu je podobná tradičním UNIXovým systémům, proto zde odkazuji na kap. 13 v ”Jemném úvodu do systému UNIX”.

❇