# Operační systémy

IOS 2015/2016

**Tomáš Vojnar**

vojnar@fit.vutbr.cz

**Vysoké učení technické v Brně**
**Fakulta informačních technologií**
**Božetěchova 2, 612 66 Brno**

❖ Tyto slajdy jsou určeny pro studenty předmětu IOS na FIT VUT. Obsahují základní popis vývoje, struktury a vlastností operačních systémů se zaměřením na systém UNIX.

❖ Předpokládají se základní znalosti programování a přehled základních pojmů z oblasti počítačů.

❖ Obsah slajdů může být v některých případech stručný, podrobnější informace jsou součástí výkladu: Velmi silně se předpokládá účast na přednáškách a doporučuje se vytvářet si poznámky.

❖ Předpokládá se také aktivní přístup ke studiu, vyhledávání informací na Internetu či v literatuře (jimiž doporučuji si případně doplnit obsah slajdů), konfrontace obsahu slajdů s jinými zdroji a také praktické experimentování s GNU/Linuxem (či FreeBSD nebo podobnými UNIXovými systémy).

# Motivace

❖ Operační systém je významnou částí prakticky všech výpočetních systémů, jež typicky zahrnují:

- hardware,
- operační systém (OS),
- uživatelské aplikační programy a
- uživatele.

❖ I pro ty, kteří se nebudou nikdy podílet na vývoji žádného OS, má studium OS velký význam:

- OS mají významný vliv na fungování celého výpočetního systému a znalost principů jejich fungování je tedy zapotřebí při vývoji:
  - hardware, na kterém má běžet nějaký OS a
  - software, který má prostřednictvím nějakého OS efektivně využívat zdroje hardware, na kterém běží – což by mělo platit vždy.

- OS jsou obvykle značně komplikované systémy, jež prošly a procházejí dlouhým vývojem, při kterém:
  - byla učiněna řada rozhodnutí ovlivňujících filozofii vývoje počítačových systémů (otevřený x uzavřený software, ovlivňování počítačového trhu na základě vlastností OS, vývoj SW v komunitách vývojářů, virtualizace, ...),
  - byla použita řada zajímavých algoritmů, způsobů návrhu architektury SW, metodologií a technik softwarového inženýrství apod., jež jsou inspirující i mimo oblast OS.

❖ Zapojení se do vývoje některého OS či jeho části není navíc zcela nepravděpodobné:

- Není např. možné dodávat nově vyvinutý hardware (rozšiřující karty apod.) bez podpory v podobě ovladačů (driverů).
- Pro potřeby různých aplikací (např. u vestavěných počítačových systémů) může být zapotřebí vyvinout/přizpůsobit vhodný OS.
- Zapojení se do vývoje některého otevřeného OS (Linux, FreeBSD, mikrojádra typu Hurd či L4, ...) je značnou intelektuální výzvou a může přinést příslušné intelektuální uspokojení.

# Organizace studia

❖ Přednáška:

- 3h/týden
- principy a vlastnosti operačních systémů (UNIX obvykle jako případová studie)
- UNIX z uživatelského a částečně programátorského pohledu

❖ Samostatná práce:

- min. 2h/týden
- samostudium, experimenty, ..., úkoly

❖ Hodnocení:

| | | |
|---|---|---|
| projekty | 30b | |
| půlsem. zkouška | 10b | (zápočet: min 10b z půlsem. zk. a projektů) |
| semestrální zkouška | 60b | (minimum: 27b) |
| celkem | 100b | |

*Pro ilustraci: v loňském roce úspěšnost cca 66 %.*

# Zdroje informací

- WWW stránky kursu: `http://www.fit.vutbr.cz/study/courses/IOS/public/`
- Diskusní fóra kursu: jsou dostupná v IS FIT – diskutujte!
- Literatura: koupit či vypůjčit (např. v knihovně FIT, ale i v jiných knihovnách).
  - Je-li některá kniha dlouhodobě vypůjčena některému zaměstnanci, neváhejte a kontaktujte ho.
- Internet:
  - vyhledávače (google, ...); často jsou užitečné dokumenty typu HOWTO, FAQ, ...
  - encyklopedie: `http://www.wikipedia.org/` – velmi vhodné při prvotním ověřování některých bodů z přednášek při studiu, lze pokračovat uvedenými odkazy, nutné křížové ověřování.
  - dokumentační projekty: `http://www.tldp.org/`, ...
- UNIX, Linux: program `man` („RTFM!“), GNU `info` a další dokumentace (`/usr/share/doc`, `/usr/local/share/doc`), ...

# Literatura


[1] Silberschatz, A., Galvin, P.B., Gagne, G.: Operating Systems Concepts, 9th ed., John Wiley & Sons, 2012. (Významná část přednášek je založena na tomto textu.)

[2] Tanenbaum, A.S.: Modern Operating Systems, 4th ed., Pearson, 2014.

[3] Tanenbaum, A.S., Woodhull, A.S.: Operating Systems Design and Implementation, 3rd ed., Prentice Hall, 2006.

[4] Raymond, E.S.: The Art Of Unix Programming, Addison-Wesley, 2003. `http://www.catb.org/~esr/writings/taoup/html/`

[5] Russinovich, M., Solomon, D., Ionescu, A.: Windows Internals, 6th ed., Microsoft Press, 2012.

[6] Skočovský, L.: Principy a problémy operačního systému UNIX, 2. vydání, 2008. `http://skocovsky.cz/paposu2008/`


Uvedeny jsou pouze některé základní, zejména přehledové knihy. Existuje samozřejmě řada dalších knih, a to včetně knih specializovaných na detaily různých subsystémů operačních systémů (plánovač, správa paměti, souborové systémy, ovladače, ...). Tyto knihy často vycházejí přímo ze zdrojových kódů příslušných subsystémů, které uvádějí s příslušným komentářem.

# Základní pojmy

# Význam pojmu operační systém

❖ Operační systém je program, resp. kolekce programů, která vytváří spojující mezivrstvu mezi hardware výpočetního systému (jenž může být virtualizován) a uživateli a jejich uživatelskými aplikačními programy.

❖ Cíle OS – kombinace dvou základních, do jisté míry protichůdných cílů, jejichž poměr se volí dle situace:

- Maximální využití zdrojů počítače:
  - drahé počítače, levnější pracovní síla,
  - zejména dříve (nebo na speciálních architekturách).
- Jednoduchost použití počítačů:
  - levné počítače a drahá pracovní síla,
  - dnes převažuje.

❖ Dvě základní role OS:

- Správce prostředků (paměť, procesor, periferie).
  - Dovoluje sdílet prostředky efektivně a bezpečně.
  - Více procesů sdílí procesor, více programů sdílí paměť, více uživatelů a souborů obsazuje diskový prostor, ...

- Tvůrce prostředí pro uživatele a jejich aplikační programy (tzv. *virtuálního počítače*).
  - Poskytuje standardní rozhraní, které zjednodušuje přenositelnost aplikací a zaučení uživatelů.
  - Poskytuje abstrakce:
    - Technické vybavení je složité a značně různorodé – práci s ním je nutné zjednodušit.
    - Problémy abstrakcí: menší efektivita, nepřístupné některé nízkoúrovňové operace.
    - Příklady abstrakcí: proces[a], soubor[b], virtuální paměť, ...

---

[a] *Program: předpis, návod na nějakou činnost zakódovaný vhodným způsobem (zdrojový text, binární program). Proces: činnost řízená programem.*

[b] *Soubor: kolekce záznamů sloužící (primárně) jako základní jednotka pro ukládání dat na vnějších paměťových médiích. Adresář: kolekce souborů.*

❖ Výše uvedené není zadarmo! OS spotřebovává zdroje (paměť, čas procesoru).

❖ OS se obvykle chápe tak, že zahrnuje:

- jádro (kernel),
- systémové knihovny a utility (systémové aplikační programy),
- textové a/nebo grafické uživatelské rozhraní.

❖ Přesná definice, co vše OS zahrnuje, však neexistuje:

- Někdy bývá OS ztotožňován takřka pouze s jádrem.
- GNU – *GNU is Not UNIX*: Projekt vývoje „svobodného“ (*free*) OS, který zahrnuje jádro (Linux, Hurd), utility, grafické i textové rozhraní (bash, Gnome, KDE), vývojové prostředky a knihovny (gcc, gdb, ...) i řadu dalších programů (včetně kancelářských programů a her).
- Microsoft Windows: Je či není prohlížeč Internetu či přehrávač videa nedílnou součástí OS?

# *Jádro OS*

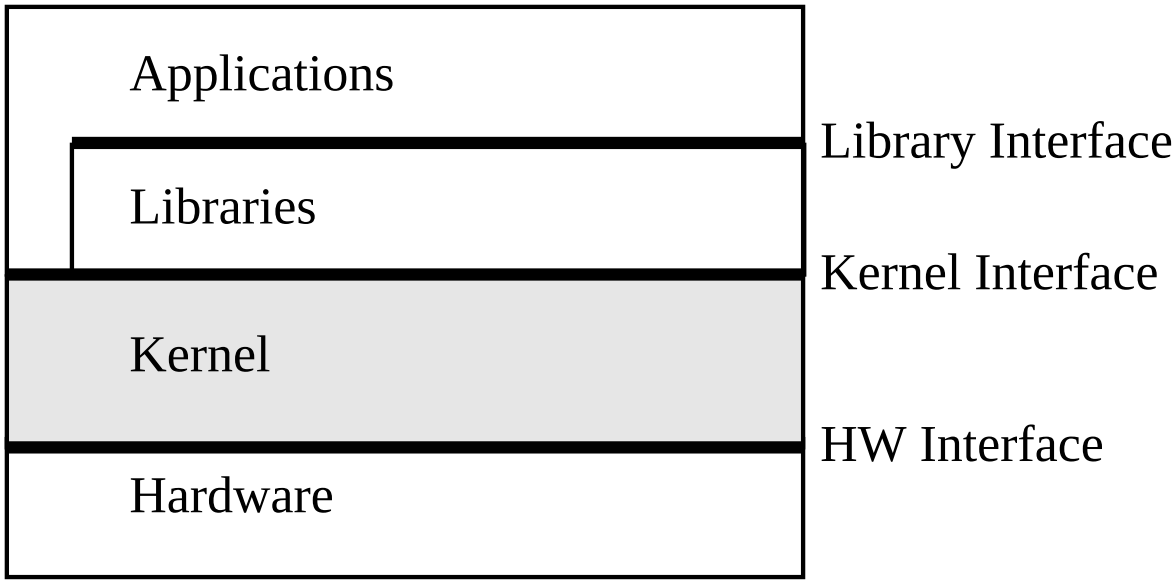
❖ Jádro OS je nejnižší a nejzákladnější část OS.

- Zavádí se první a běží po celou dobu běhu počítačového systému (tzv. *reaktivní* spíše než *transformační* program).
- Navazuje přímo na hardware, příp. virtualizovaný HW, a typicky ho pro uživatele a uživatelské aplikace zcela zapouzdřuje.
- Běží obvykle v privilegovaném režimu.
  - V tomto režimu může provádět libovolné operace nad (virtualizovaným) HW počítače.
  - Za účelem oddělení a ochrany uživatelů a jejich aplikačních programů nesmí tyto mít možnost do tohoto režimu libovolně vstupovat.
  - Nutná HW podpora v CPU.
- Zajišťuje základní správu prostředků a tvorbu prostředí pro vyšší vrstvy OS a uživatelské aplikace.
  - To, jaké konkrétní služby jádro nabízí, je výsledkem zvoleného kompromisu mezi efektivitou, bezpečností, flexibilitou a příp. dalšími aspekty.
  - Jedná se ovšem minimálně o tu část služeb, která bezprostředně vyžaduje interakci s HW (přepínání kontextu, zavádění stránek, nastavování parametrů HW apod.).

❖ Systémové i uživatelské aplikační programy mohou explicitně žádat jádro o služby prostřednictvím systémových volání (*system call*). Děje se tak přímo nebo nepřímo s využitím specializovaných instrukcí (např. u x86 softwarové přerušení nebo `SYSCALL`/`SYSENTER`), které způsobí kontrolovaný přechod do režimu jádra.

❖ Rozlišujeme dva typy rozhraní:

- Kernel Interface: přímé volání jádra specializovanou instrukcí
- Library Interface: volání funkcí ze systémových knihoven, které mohou (ale nemusí) vést na volání služeb jádra



# Typy jader OS

❖ Monolitická jádra:

- Vytváří vysokoúrovňové komplexní rozhraní s řadou služeb a abstrakcí nabízených vyšším vrstvám.
- Všechny subsystémy implementující tyto služby (správa paměti, plánování, meziprocesová komunikace, souborové systémy, podpora síťové komunikace apod.) běží v privilegovaném režimu a jsou těsně provázané za účelem vysoké efektivity.

❖ Vylepšením koncepce monolitických jader jsou monolitická jádra s modulární strukturou:

- Umožňuje zavádět/odstraňovat subsystémy jádra v podobě tzv. modulů za běhu.
- Např. FreeBSD či Linux (`lsmod`, `modprobe`, `rmmod`, ...)

❖ Mikrojádra:

- Minimalizují rozsah jádra a nabízí jednoduché rozhraní, s jednoduchými abstrakcemi a malým počtem služeb pro nejzákladnější správu procesoru, vstup/výstupních zařízení, paměti a meziprocesové komunikace.
- Většina služeb nabízených monolitickými jádry (včetně např. ovladačů, významných částí správy paměti či plánování) je implementována mimo mikrojádro v tzv. serverech, jež neběží v privilegovaném režimu.
- Příklady mikrojáder: Mach, QNX, L4 (pouhých 7 služeb oproti více než 200 u jádra 2.6 Linuxu; verze L4: Fiasco, Pistachio, seL4, ...), ...

- Výhody mikrojáder:
  - Flexibilita – možnost více současně běžících implementací různých služeb, jejich dynamické spouštění, zastavování apod.
  - Zabezpečení – servery neběží v privilegovaném režimu, chyba v nich/útok na ně neznamená ihned selhání/ovládnutí celého OS.

- Nevýhoda mikrojáder: vyšší režie – výrazně vyšší u mikrojader 1. generace (Mach), lepší je situace u (stále experimentálních) mikrojader 2. generace (např. L4: minimalismus, optimalizace pro konkrétní architekturu), nicméně tato nevýhoda přetrvává.

❖ Hybridní jádra:

- Mikrojádra rozšířená o kód, který by mohl být implementován ve formě serveru, ale je za účelem menší režie těsněji provázán s mikrojádrem a běží v jeho režimu.
- Příklady: Mac OS X (Mach kombinovaný s BSD), Windows NT (a vyšší), ...

❖ Exojádra:

- Experimentální jádra poskytující velmi nízké rozhraní zaměřené hlavně na bezpečné sdílení prostředků (a ne na tvorbu abstrakcí či standardního rozhraní). Doplněno o knihovny implementující služby jinak nabízené běžně jádrem.
- Příklady: Aegis, Nemesis, ...

# Historie vývoje OS

❖ Je nutné znát historii, protože se opakuje... :-)

❖ První počítače:

- Knihovna podprogramů (například pro vstup a výstup) – zárodek OS.
- Dávkové zpracování: důležité je vytížení stroje, jednoduchá podpora OS pro postupné provádění jednotlivých úloh seřazených operátory do dávek.
- Multiprogramování – více úloh zpracovávaných současně:
  - Překrývání činnosti procesoru a vstup/výstupního podsystému (vyrovnávací paměti, přerušení).
  - Zatímco jedna úloha běží, jiná může čekat na dokončení I/O (problém: ochrana paměti, řešeno technickými prostředky).
  - OS začíná být významnou částí programového vybavení (nevýhoda: OS zatěžuje počítač).

❖ Příchod levnějších počítačů:

- Interaktivnost, produktivita práce – lidé nečekají na dokončení zpracování dávky.
- Stále se ještě nevyplatí každému uživateli dát počítač – terminály a sdílení času: timesharing/multitasking, tj. „současný“ běh více aplikací na jednom procesoru.
- Problém odezvy na vstup: preemptivní plánování úloh[a].
- Oddělené ukládání dat uživatelů: systémy souborů.
- Problémy s přetížením počítače mnoha uživateli.
- OS řídí sdílení zdrojů: omezené použití prostředků uživatelem (priority, quota).

---

[a] *Nepreemptivní plánování: Procesor může být procesu odebrán, pokud požádá jádro o nějakou službu (I/O operaci, ukončení, vzdání se procesoru). Preemptivní plánování: OS může procesu odebrat procesor i „ proti jeho vůli“ (tedy i když sám nežádá o nějakou službu jádra), a to na základě příchodu přerušení při určité události (typicky při vypršení přiděleného časového kvanta, ale také dokončení I/O operace jiného procesu apod.).*

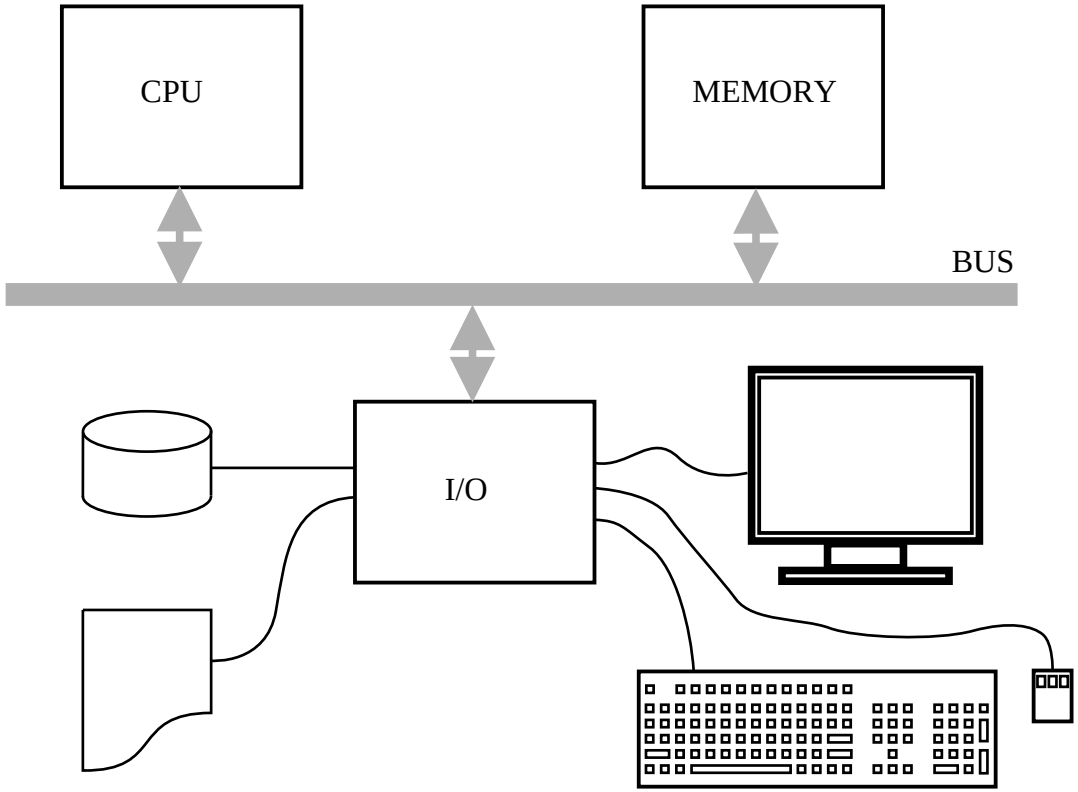
❖ Ještě levnější počítače:

- Každý uživatel má svůj počítač, který musí být levný a jednoduchý (omezená paměť, chybějící ochrana paměti, jednoduchý OS – na poli OS jde o návrat zpět: CP/M, MS-DOS).
- Další pokrok v technologiích – sítě, GUI.
- Nové OS opět získávají vlastnosti starších systémů (propojení přes síť si opět vynucuje patřičné ochrany, ...).

❖ Následoval (následuje) podobný vývoj u nejprve málo výkonných kapesních počítačů a mobilů s omezenými OS (jednoúlohovost apod.), nyní již ale běžně s více-jádrovými procesory a převzetím mnoha rysů běžných OS.

❖ Další podobný vývoj: např. senzorové sítě, Internet of Things apod. – prozatím málo výkonné uzly (spotřeba energie), ale co bude následovat?

# Přehled technického vybavení




Procesor: řadič, ALU, registry (IP, SP), instrukce

Paměť: adresa, hierarchie pamětí (cache, ...)

Periferie: disk, klávesnice, monitor, (I/O porty, paměťově mapované I/O, přerušení, DMA)

Sběrnice: FSB, HyperTransport, QPI, NVLink, CAPI, PCI, USB, IEEE1394, ATA/SATA, SCSI/SAS, ...

# Klasifikace počítačů

- Klasifikace počítačů podle účelu:
  - univerzální,
  - specializované:
    - vestavěné (řízení technologických zařízení, palubní počítače, spotřební elektronika, ...),
    - aplikačně orientované (databázové, výpočetní, síťové servery, ...),
    - vývojové (zkoušení nových technologií), ...

- Klasifikace počítačů podle výkonnosti:
  - vestavěné počítače, handheldy, mobily, ...,
  - osobní počítače (personal computer),
  - pracovní stanice (workstation),
  - servery,
  - střediskové počítače (mainframe),
  - superpočítače.

# Klasifikace OS

- Klasifikace OS podle účelu:
  - univerzální (UNIX, Windows, Linux, ...),
  - specializované:
    - real-time (QNX, RXS, RT-Linux, ...),
    - databáze, web, ... (např. z/VSE),
    - mobilní zařízení (Android, iOS, Windows Phone, ...), ...

- Klasifikace OS podle počtu uživatelů:
  - jednouživatelské (CP/M, MS-DOS, ...) a
  - víceuživatelské (UNIX, Windows, ...).

- Klasifikace OS podle počtu současně běžících úloh:
  - jednoúlohové a
  - víceúlohove (multitasking: ne/preemptivní).

# *Příklady dnes používaných OS*

| typ počítače | příklady OS |
|---|---|
| mainframe | MVS, z/OS, VM/CMS, VM/Linux, z/VSE, Windows |
| superpočítače | často varianty Linuxu/UNIXu, Windows HPC |
| server | UNIX, FreeBSD, Linux, Windows |
| PC | Windows, MacOS X, Linux |
| real-time | QNX, RT-Linux |
| handheldy, mobily | Android, iOS, Windows Mobile, Firefox OS, Linux |

# *Implementace OS*

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

- Z výše uvedeného plyne:
  - Jistá setrvačnost při implementaci: snaha neměnit kód, který již spolehlivě pracuje.
  - Používání řady technik pro minimalizaci výskytu chyb, např.:
    - inspekce zdrojového kódu (důraz na srozumitelnost!),
    - rozsáhlé testování,
    - podpora vývoje technik automatizované (formální) verifikace.

# Hlavní směry ve vývoji OS

- Pokročilé architektury (mikrojádra, hybridní jádra, ...): zdůraznění výhod, minimalizace nevýhod, možnost kombinace různých, současně běžících OS, ...
- Bezpečnost a spolehlivost.
- Podpora hard real-time (průmyslové procesy), soft real-time (multimédia).
- Multiprocessing (SMP, ...).
- Virtualizace.
- Distribuované zpracování, clustery, gridy, Internet of Things.
- OS kapesních počítačů (handheld), vestavěných systémů, mobilů, ...
- Vývoj nových technik návrhu a implementace OS (podpora formální verifikace, ...).
- ...

# Základy práce s UNIXem

# Studentské počítače s UNIXem na FIT

- studentské UNIXové servery: eva (FreeBSD), merlin (Linux/CentOS)
- PC s Linuxem (CentOS): běžně na učebnách (dual boot)

❖ Přihlášení:

```
Login: xnovak00
Password:       # neopisuje se - změna: příkaz passwd
...
$               # prompt - vyzývací znak shellu
```

❖ Vzdálené přihlášení:

- programy/protokoly pro vzdálení přihlášení: ssh, telnet (telnet neužívat!)
- putty – ssh klient pro Windows

# Základní příkazy

- Práce s adresáři a jejich obsahem:
  - `cd` – change directory
  - `pwd` – print working directory
  - `ls [-al]` – výpis obsahu adresáře/informace o souborech
  - `mkdir` – make a directory
  - `rmdir` – remove a directory
  - `mv` – přesun/přejmenování souboru/adresáře (move)
  - `cp [-r]` – kopie souboru/adresáře (copy)
  - `rm [-ir]` – smazání souboru/adresáře (remove)

❖ Výpis obsahu souboru:

- `cat` – spojí několik souborů na std. výstup
- `more`/`less` – výpis po stránkách
- `head -13 FILE` – prvních 13 řádků
- `tail -13 FILE` – posledních 13 řádků

- `file FILE` – informace o typu obsahu souboru

# *Speciální znaky*

- `^C` – ukončení procesu na popředí
- `^Z` – pozastavení procesu na popředí (dále `bg`/`fg`)
- `^S`/`^Q` – pozastavení/obnovení výpisu na obrazovku
- `^\` – `^C` a výpis „core dump“

- `^D` – konec vstupu (`$^D` ukončí shell)

- `^H` – smazání posledního znaku (backspace)
- `^W` – smazání posledního slova
- `^U` – smazání současného řádku

# Operační systémy

IOS 2015/2016

**Tomáš Vojnar**

vojnar@fit.vutbr.cz

**Vysoké učení technické v Brně**
**Fakulta informačních technologií**
**Božetěchova 2, 612 66 Brno**

# Unix – úvod

# Historie UNIXu

- 1961: CTSS (Compatible Time-Sharing System):
  - vyvinut na MIT, jeden z prvních OS podporujících sdílení času (další takový OS byl Plato II z University of Illinois),
  - jeden z prvních OS s emailem, zárodkem shellu, jedním z prvních programů pro formátování textu (RUNOFF, předchůdce nroff).

- 1965: MULTICS – Multiplexed Information and Computation Service (Bell Labs, MIT, General Electric):
  - v zásadě neúspěšný OS, „zhroutil se vlastní vahou“ (přílišná obecnost, přílišný rozsah služeb, přespecifikovanost), měl ale významný vliv na další vývoj OS,
  - hierarchický souborový systém, paměťově mapované soubory, dynamické linkování, bezpečnostní okruhy, ACLs, dynamická rekonfigurace, předchůdce shellu, ...

- 1969: Začátek vývoje nového OS – PDP 7 (K. Thompson, D. Ritchie a několik jejich kolegů z Bell Labs, AT&T).

- 1970: Zavedeno jméno UNIX (původně UNICS).

- 1971: PDP-11 (24KB RAM, 512KB disk) – text processing.

- 1972: asi 10 instalací.
- 1973: UNIX přepsán do C.
- 1974: Článek: „Unix Timesharing System“, asi 600 instalací.
- 1977: Berkeley Software Distribution – BSD.
- 1978: SCO (Santa-Cruz Operation) – první Unixová společnost.
- 1979: UNIX Version 7 – verze UNIXu blízká současným verzím.
- 1980:
  - DARPA si vybrala UNIX jako platformu pro implementaci TCP/IP.
  - Microsoft a jeho XENIX.
- 1981: Microsoft, smlouva s IBM, QDOS, MS-DOS.
- 1982: Sun Microsystems.
- 1983:
  - AT&T UNIX System V.
  - BSD 4.2 – síť TCP/IP.
  - GNU, R. Stallman.

- 1984: X/OPEN XPG.
- 1985: POSIX (IEEE).
- 1987: AT&T, SUN: System V Release 4 a OSF/1.
- 1990: Windows 3.0.
- 1991: Solaris, Linux.
- 1992: 386BSD.
- 1994: Single Unix Specification (The Open Group).
- 1998:
  - začátek prací na sloučení základu SUS a POSIX,
  - open source software.
- 2002: SUS v3 – zahrnuje POSIX, poslední revize 2008 (SUS v4).

- 2015: Spolupráce Red Hat a Microsoft v oblasti cloudových řešení a podpory .NET.

# Příčiny úspěchu

- Mezi příčiny úspěchu UNIXu lze zařadit:
  - víceprocesový, víceuživatelský,
  - napsán v C – přenositelný,
  - zpočátku (a později) šířen ve zdrojovém tvaru,
  - „mechanism, not policy“,
  - „fun to hack“,
  - jednoduché uživatelské rozhraní,
  - skládání složitějších programů z jednodušších,
  - hierarchický systém souborů,
  - konzistentní rozhraní periferních zařízení, ...

- Řada z těchto myšlenek je inspirující i mimo oblast OS.

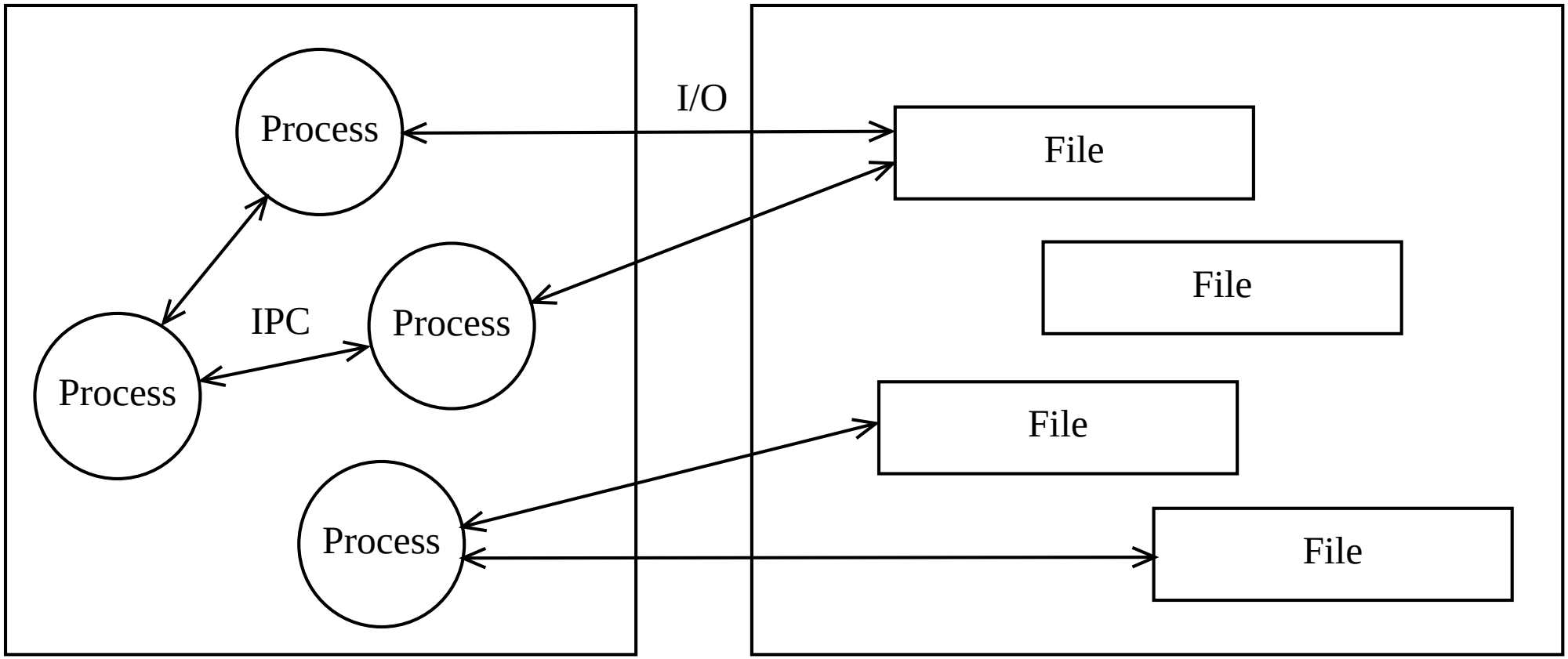
# *Varianty UNIXu*

- Hlavní větve OS UNIXového typu:
  - UNIX System V,
  - BSD UNIX,
  - různé firemní varianty (AIX, Solaris, ...),
  - Linux.

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

# Základní koncepty

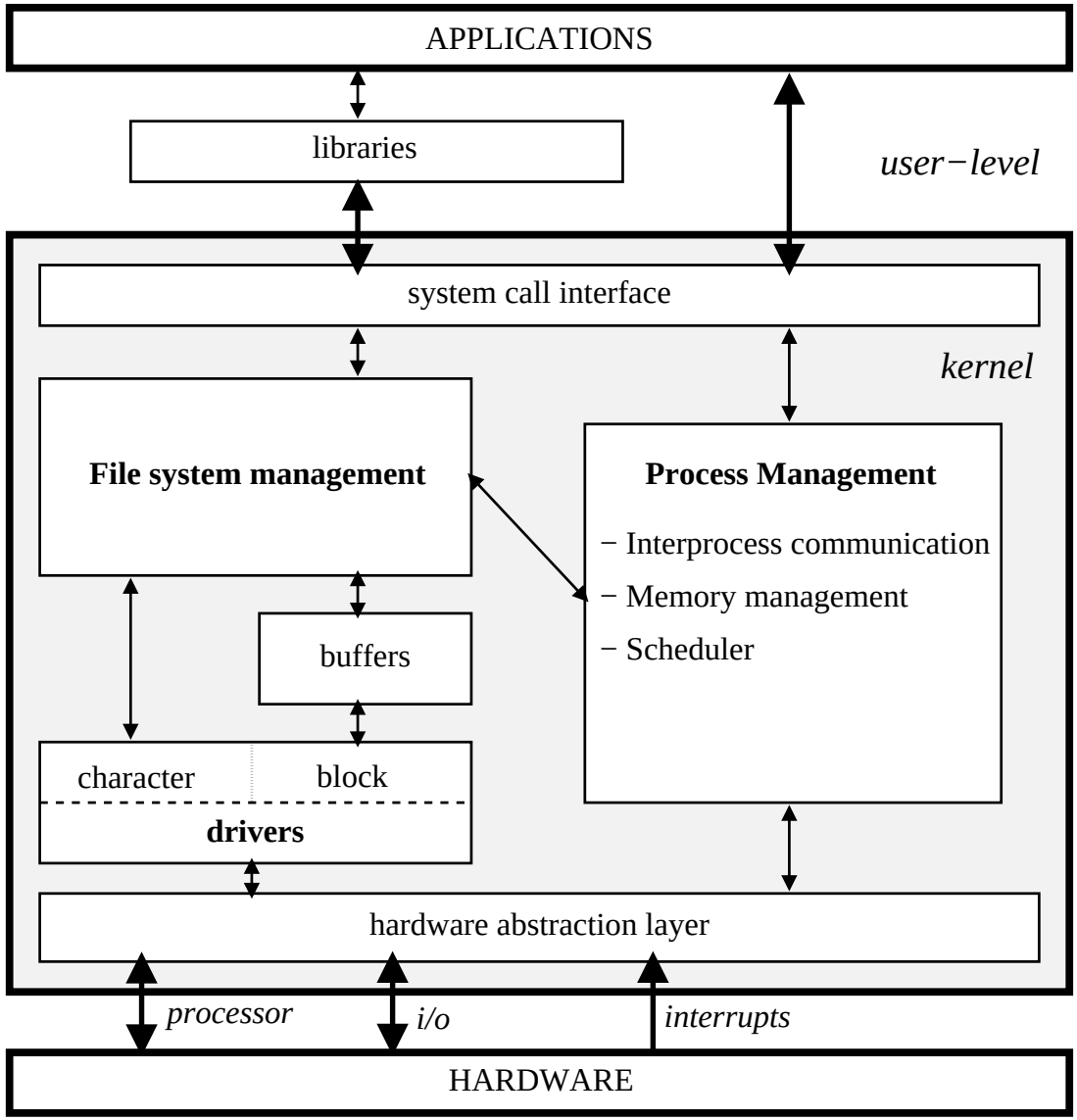
- Dva základní koncepty/abstrakce v UNIXu: procesy a soubory.




IPC = Inter-Process Communication – roury (pipes), signály, semafory, sdílená paměť, sockets, RPC, zprávy, streams...
I/O = Input/Output.

# Struktura jádra UNIXu

- Základní podsystémy UNIXu:
  - Správa souborů
    (File Management)
  - Správa procesů
    (Process Management)



# Komunikace s jádrem

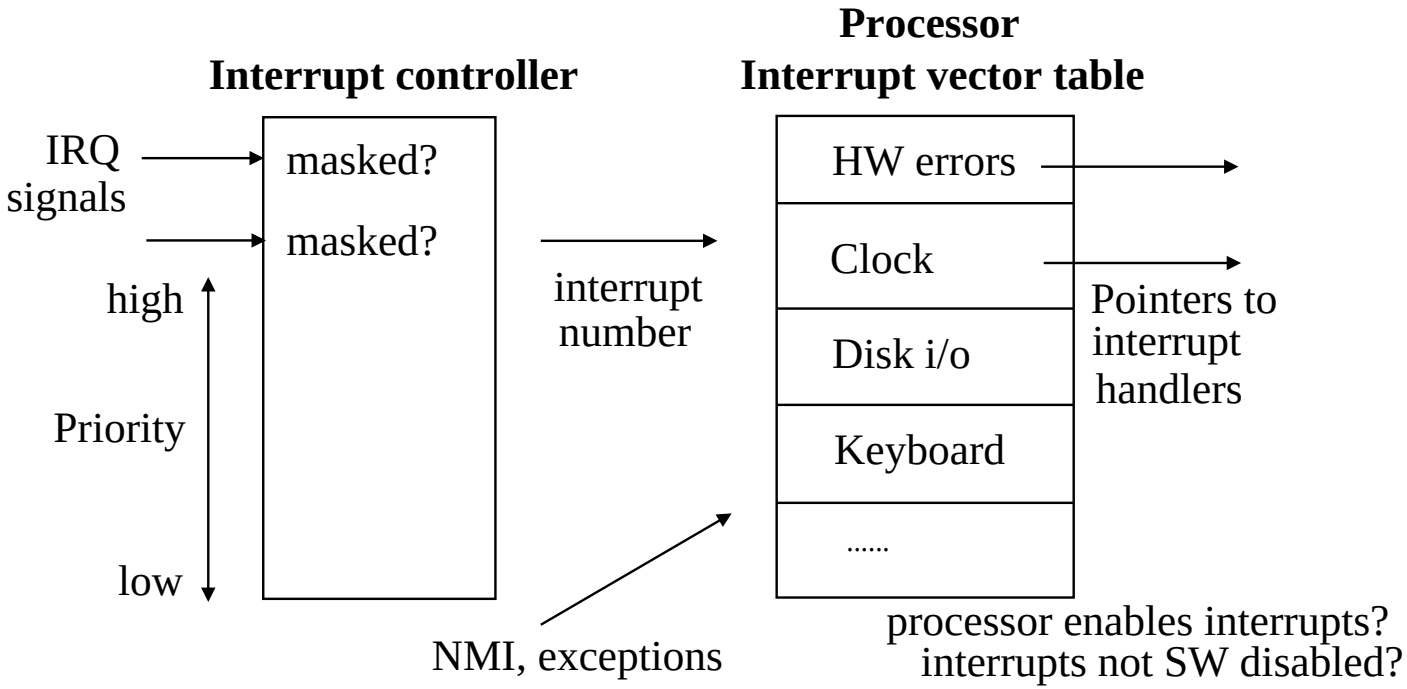
❖ Služby jádra – operace, jejichž realizace je pro procesy zajišťována jádrem. Explicitně je možno o provedení určité služby žádat prostřednictvím systémového volání („system call“).

❖ Příklady některých služeb jádra dostupných systémovým voláním v UNIXu:

| služba | jaká operace se provede |
|---|---|
| open | otevře soubor |
| close | zavře soubor |
| read | čte ze souboru |
| write | zapisuje |
| kill | pošle signál |
| fork | duplikuje proces |
| exec | přepíše kód |
| exit | ukončí proces |

❖ HW přerušení (hardware interrupts) – mechanismus, kterým HW zařízení oznamují jádru asynchronně vznik událostí, které je zapotřebí obsloužit.

- Žádosti o HW přerušení přichází jako elektrické signály do řadiče přerušení.
  Na PC dříve PIC, nyní APIC (distribuovaný systém: každý procesor má lokální APIC, externí zařízení mohou být připojena přímo nebo přes I/O APIC – součást chip setu).
- Přerušení mají přiřazeny priority, dle kterých se oznamují procesoru.
- Přijme-li procesor přerušení s určitým číslem (tzv. interrupt vector) vyvolá odpovídající obslužnou rutinu (interrupt handler) na základě tabulky přerušení, přičemž automaticky přejde do privilegovaného režimu.
- Řadič může být naprogramován tak, že některá přerušení jsou maskována, případně jsou maskována všechna přerušení až po určitou prioritní úroveň.



❖ Příjem nebo obsluhu HW přerušení lze také zakázat:

- na procesoru (instrukce CLI/STI na Intel/AMD),
- čistě programově v jádře (přerušení se přijme, ale jen se poznamená jeho příchod a dále se neobsluhuje).

❖ NMI (non-maskable interrupt): HW přerušení, které nelze zamaskovat na řadiči ani zakázat jeho příjem na procesoru (typicky při chybách HW bez možnosti zotavení: chyby paměti, sběrnice apod., někdy také pro ladění či řešení uváznutí v jádře: “NMI watchdog”).

❖ Přerušení také vznikají přímo v procesoru – synchronní přerušení, výjimky (exceptions):

- trap: po obsluze se pokračuje další instrukcí (breakpoint, přetečení apod.),
- fault: po obsluze se (případně) opakuje instrukce, která výjimku vyvolala (např. výpadek stránky, dělení nulou, chybný operační kód apod.),
- abort: nelze určit, jak pokračovat, provádění se ukončí (např. některé zanořené výjimky typu fault, chyby HW detekované procesorem – tzv. „machine check mechanism“).

❖ Při obsluze přerušení je zapotřebí dávat pozor na současný příchod více přerušení a možnost přerušení obsluhy přerušení:

- Nutnost synchronizace obsluhy přerušení tak, aby nedošlo k nekonzistencím ve stavu jádra díky interferenci částečně provedených obslužných rutin.
- Využívají se různé mechanismy vyloučení obsluhy přerušení (viz předchozí slajd).
- Při vyloučení obsluhy je zapotřebí věnovat pozornost době, po kterou je obsluha vyloučena:
  - zvyšuje se latence (odezva systému),
  - může dojít ke ztrátě přerušení (nezaznamenají se opakované výskyty),
  - výpočetní systém se může dostat do nekonzistentního stavu (zpoždění času, ztráta přehledu o situaci vně jádra, neprovedení některých akcí v hardware, přetečení vyrovnávacích pamětí apod.).
- Vhodné využití priorit, rozdělení obsluhy do více částí (viz další slajd).

❖ Obsluha přerušení je často rozdělena na dvě úrovně:

- 1. úroveň:
  - Zajišťuje minimální obsluhu HW (přesun dat z/do bufferu, vydání příkazů k další činnosti HW apod.) a plánuje běh obsluhy 2. úrovně.
  - Během obsluhy na této úrovni může být zamaskován příjem dalších přerušení stejného typu, stejné či nižší priority, případně i všech přerušení.
- 2. úroveň:
  - Postupně řeší zaznamenaná přerušení, nemusí zakazovat přerušení.
  - Vzájemné vyloučení se dá řešit běžnými synchronizačními prostředky (semafory apod. – nelze u 1. úrovně, kde by se blokoval proces, v rámci jehož běhu přerušení vzniklo).
  - Mohou zde být samostatné úrovně priorit.
  - Obsluha může běžet ve speciálních procesech (interrupt threads ve FreeBSD, tasklety/softIRQ v Linuxu).

❖ Mohou existovat i další speciální typy přerušení, která obsluhuje procesor zcela specifickým způsobem, často mimo vliv jádra. Např. na architekturách Intel/AMD:

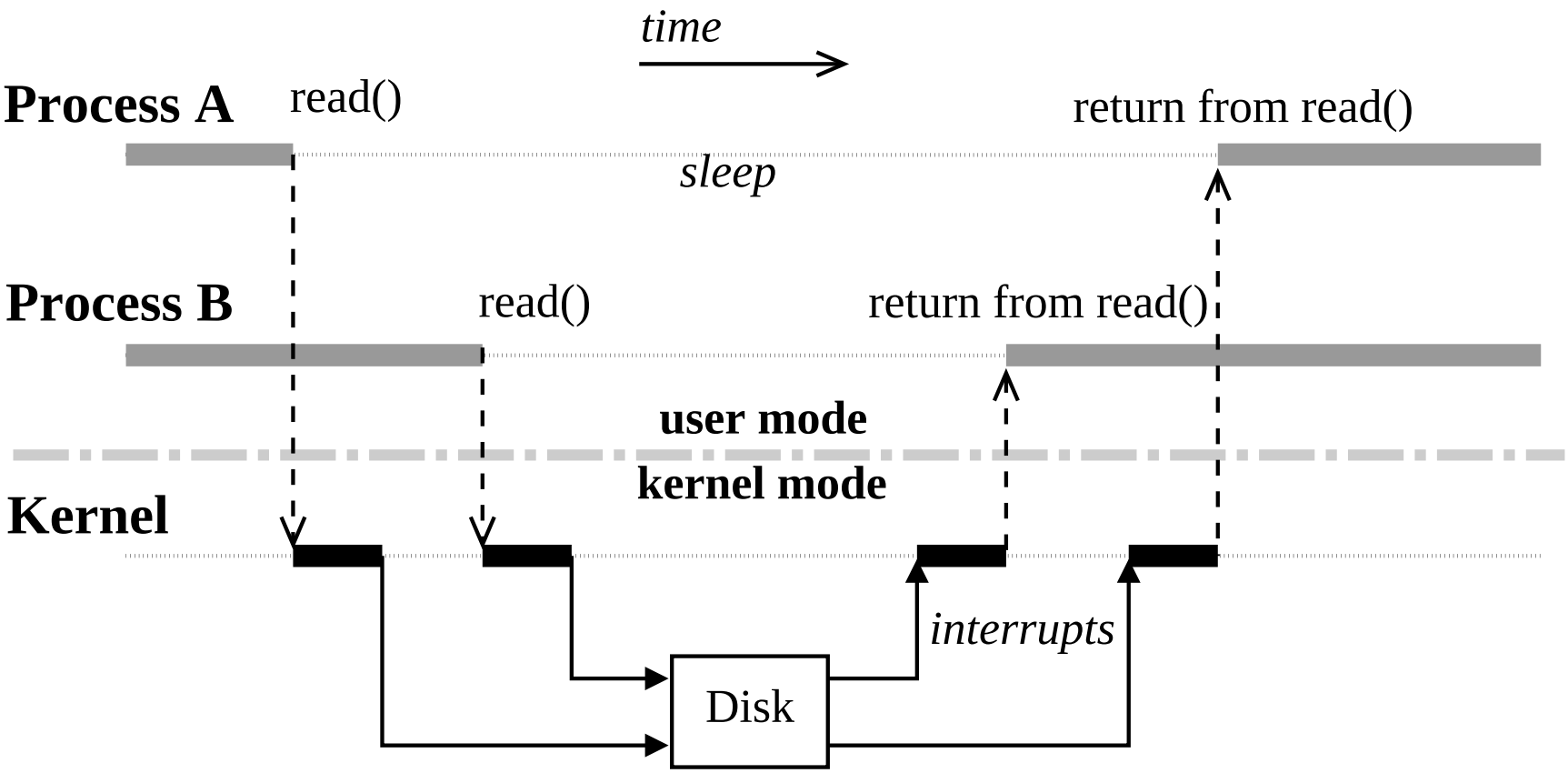
- Interprocessor interrupt (IPI): umožňuje SW běžícímu na jednom procesoru poslat určité přerušení jinému (nebo i stejnému) procesoru – využití např. pro zajištění koherence některých vyrovnávacích pamětí ve víceprocesorovém systému (TLB), předávání obsluhy přerušení či v rámci plánování.
- System management interrupt (SMI): generováno HW i SW, způsobí přechod do tzv. system management mode (SMM) procesoru.
  - V rámci SMM běží firmware, který může být využit k optimalizaci spotřeby energie, obsluhu různých chybových stavů (přehřátí) apod.
  - Po dobu běhu SMM jsou prakticky vyloučena ostatní přerušení, což může způsobit zpožďování času v jádře a případné další nekonzistence stavu jádra oproti okolí, které mohou vést k pádu jádra.
- HW signály jako RESET, INIT, STOPCLK, FLUSH apod.

❖ Ovladače zařízení a přerušení:

- Při inicializaci ovladače (v Linuxu typicky modul), nebo jeho prvním použití, se musí registrovat k obsluze určitého IRQ.
- Příslužné IRQ je buď standardní, zjistí se přes konfigurační rozhraní sběrnic komunikací s jejich řadičem, nebo sondováním (zařízení je vyzváno ke generování přerušení a sleduje se, ke kterému dojde).
- IRQ může být sdíleno: jsou volány všechny registrované obslužné rutiny a musí být schopny komunikací s příslušným zařízením rozpoznat, zda přerušení bylo vygenerováno příslušným zařízením.

❖ Základní statistiky o obsluze přerušení v Linuxu: `/proc/interrupts`.

- Příklad komunikace s jádrem:
  - synchronní: proces-jádro
  - asynchronní: hardware-jádro



# Operační systémy

IOS 2015/2016

**Tomáš Vojnar**

vojnar@fit.vutbr.cz

**Vysoké učení technické v Brně**
**Fakulta informačních technologií**
**Božetěchova 2, 612 66 Brno**

# Programování v UNIXu: přehled

# Nástroje programátora

- Prostředí pro programování zahrnuje:
  - API OS a různých aplikačních knihoven,
  - CLI a GUI,
  - editory,
  - překladače a sestavovače/interprety,
  - ladící nástroje,
  - nástroje pro automatizaci překladu,
  - ...
  - dokumentace.

- CLI a GUI v UNIXu:
  - CLI: shell (sh, ksh, csh, bash, dash, ...)
  - GUI: X-Window

# X-Window Systém

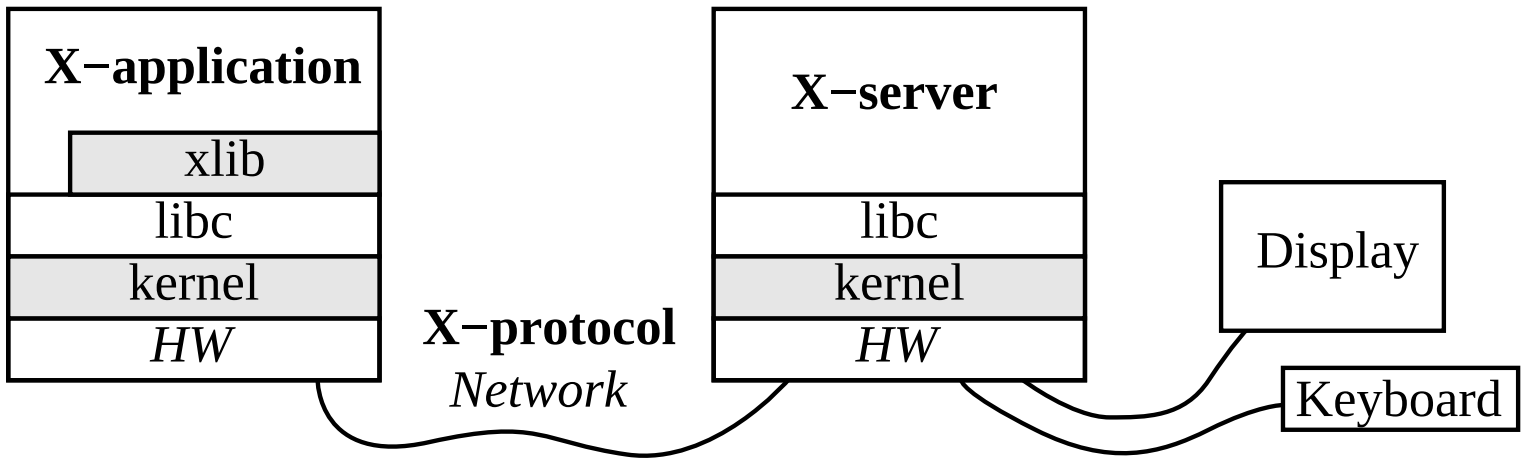
❖ Základní charakteristiky:

- grafické rozhraní typu client-server, nezávislé na OS, umožňující vzdálený přístup,
- otevřená implementace: XFree86/X.Org,
- mechanismy, ne politika — výhoda či nevýhoda?

❖ X-server: zobrazuje grafiku, ovládá grafický HW, myš, klávesnici...; s aplikacemi a správcem oken komunikuje přes X-protokol.

❖ Window Manager: správce oken (dekorace, změna pozice/rozměru, ...); s aplikacemi komunikuje přes ICCM protokol (Inter-Client Communication Protocol).

❖ Knihovna `xlib`: standardní rozhraní pro aplikace, implementuje X-protokol, ICCM, ...



# *Vzdálený přístup přes X-Window*

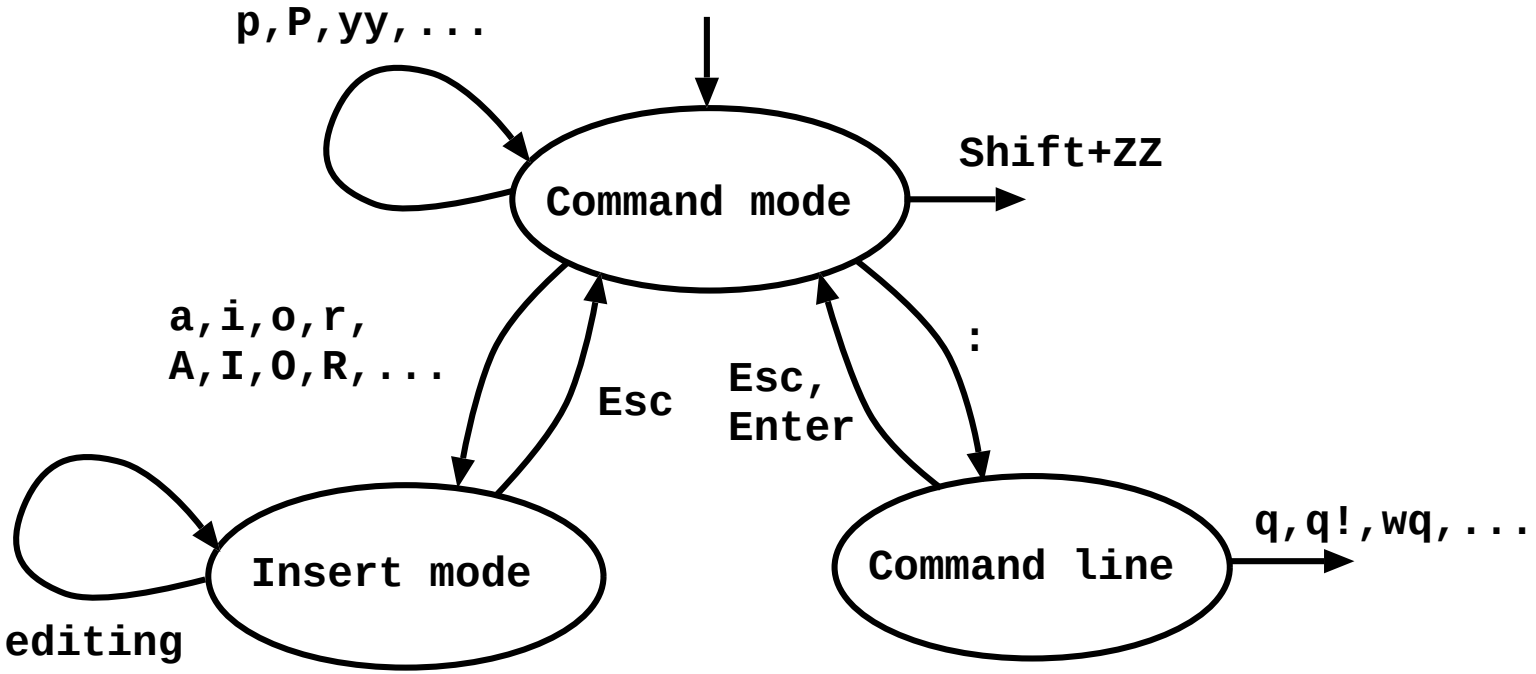
- Spuštění aplikace s GUI ze vzdáleného počítače:
  - lokální systém: `xhost + ...`
  - vzdálený systém: `export DISPLAY=...` a spuštění aplikace
  - tunelování přes ssh: `ssh -X`

- Vnořené GUI ze vzdáleného počítače: `Xnest`.

# Editory, `vim`

- Textové editory běžné v UNIXu:
  - v terminálu: `vi`, `vim`, `emacs`, ...
  - grafické: `gvim`, `xemacs`, `gedit`, `nedit`, ...

- Tři režimy vi, vim:



# Užitečné příkazy ve `vim`

- Mazání – smaže a vloží do registru:
  - znak: `x`/`X`
  - řádek: `dd`
  - konec/začátek řádku: `dEnd` / `dHome`
  - konec/začátek slova: `dw` / `db`
  - konec/začátek odstavce: `d}` / `d{`
  - do znaku: `dt` znak

- Změna: `r`, `R` a `cc`, `cw`, `cb`, `c+End/Home`, `c}`, `ct`+ukončující znak, ...

- Vložení textu do registru: `yy`, `yw`, `y}`, `yt`+ukončující znak, ...

- Vložení registru do textu: `p`/`P`

- Bloky: (`v`+šipky)/(`Shift-v`+šipky)/(`Ctrl-v`+šipky)+`y` /`d`, ...

- Vícenásobná aplikace: číslo+příkaz (např. `5dd`)
- undo/redo: `u` /`Ctrl-R`
- Opakování posledního příkazu: `.` (tečka)
- Vyhledání: `/` regulární výraz
- Aplikace akce po vzorek: `d/` regulární výraz

# Regulární výrazy

❖ Regulární výrazy jsou nástrojem pro konečný popis případně nekonečné množiny řetězců. Jejich uplatnění je velmi široké – nejde jen o vyhledávání ve `vim`!

❖ Základní regulární výrazy (existují také rozšířené RV – viz dále):

| znak | význam |
|---|---|
| obyčejný znak | daný znak |
| `.` | libovolný znak |
| `*` | $0 - n$ výskytů předchozího znaku |
| `[`*množina*`]` | znak z množiny, např: `[0-9A-Fa-f]` |
| `[^`*množina*`]` | znak z doplňku množiny |
| `\` | ruší řídicí význam následujícího znaku |
| `^` | začátek řádku |
| `$` | konec řádku |
| `[[:`*k*`:]]` | znak z dané kategorie *k* podle `locale` |

❖ Příklad: `"^ *\* [0-9][0-9]* *$"`

# Příkazová řádka ve `vim`

- Uložení do souboru: `w`, případně `w!`
- Vyhledání a změna: řádky `s/` regulární výraz `/` regulární výraz (`/g`)
- Adresace řádků: číslo řádku, interval (`x,y`), aktuální řádek (`.`), poslední řádek (`$`), všechny řádky (`%`), nejbližší další řádek obsahující řetězec (`/řetězec/`), nejbližší předchozí řádek obsahující řetězec (`?řetězec?`)
- Příklad – vydělení čísel 10:

```
:%s/\([0-9]*\)\([0-9]\)/\1.\2/
```

# Základní dokumentace v UNIXu

- `man`, `info`, `/usr/share/doc`, `/usr/local/share/doc`, HOWTO, FAQ, ..., README, INSTALL, ...

- `man` je rozdělen do sekcí (`man n name`):
  1. Executable programs or shell commands
  2. System calls (functions provided by the kernel)
  3. Library calls (functions within program libraries)
  4. Special files (usually found in `/dev`)
  5. File formats and conventions, e.g., `/etc/passwd`
  6. Games
  7. Miscellaneous (including macro packages and conventions)
  8. System administration commands (usually only for root)
  9. Kernel routines (Non standard)

- `apropos name` – kde všude se v `man` mluví o `name`.

# Bourne shell

- Skriptování:
  - Interpret: program, který provádí činnost programu, který je jeho vstupem.
  - Skript: textový soubor s programem pro interpret.
  - Nevýhody: pomalejší, je třeba interpret.
  - Výhody: nemusí se překládat (okamžitě spustitelné), čitelný obsah programu.

- Spuštění skriptu v UNIXu:

```
sh skript.sh          # explicitní volání interpretu

chmod +x skript.sh    # nastaví příznak spustitelnosti
./skript.sh           # spuštění

. ./skript.sh         # spuštění v aktuálním shellu
```

❖ “Magic number” = číslo uvedené na začátku souboru a charakterizující jeho obsah:

- U spustitelných souborů jádro zjistí na základě magic number, jak soubor spustit: tj. u binárních programů jejich formát určující způsob zevedení do paměti, u intepretovaných programů pak, který interpret použít (první řádek: `#!/cesta/interpret`).
- Výhoda: možnost psát programy v libovolném skriptovacím jazyku.

❖ Příklady:

```
#!/bin/sh              - skript pro Bourne shell

#!/bin/ksh             - skript pro Korn shell
#!/bin/csh             - skript pro C shell

#!/usr/bin/perl        - skript v Perlu
#!/usr/bin/python      - skript v Pythonu

\177ELF                - binární program - formát ELF
```

# Speciální znaky (metaznaky)

- Jsou interpretovány shellem, znamenají provedení nějaké speciální operace.
- Jejich speciální význam lze zrušit například znakem \ těsně před speciálním znakem.
- Poznámky:

| znak | význam a příklad použití |
|---|---|
| # | poznámka do konce řádku<br>`echo "text" # poznámka` |

- Práce s proměnnými:

| | |
|---|---|
| $ | zpřístupnění hodnoty proměnné<br>`echo $TERM` |

## ❖ Přesměrování vstupu a výstupu:

| znak | význam a příklad použití |
|---|---|
| > | přesměrování výstupu, přepíše soubor<br><br>`echo "text" >soubor # přesměrování stdout`<br>`příkaz 2>soubor # přesměrování stderr`<br><br>`příkaz [n]>soubor # n je číslo, implicitně 1`<br><br>Čísla zde slouží jako popisovače otevřených souborů (file handle).<br>Standardně používané a otevírané popisovače:<br>• `stdin=0` standardní vstup (klávesnice)<br>• `stdout=1` std. výstup (obrazovka)<br>• `stderr=2` std. chybový výstup (obrazovka) |

| znak | význam a příklad použití |
|---|---|
| `>&` | duplikace popisovače pro výstup<br><br>`echo "text" >&2 # stdout do stderr`<br>`příkaz >soubor 2>&1 # přesm. stderr i stdout`<br>`příkaz 2>&1 >soubor # stdout do souboru,`<br>`# stderr na obrazovku`<br><br>`m>&n # m a n jsou čísla` |
| `>>` | přesměrování výstupu, přidává do souboru<br><br>`echo "text" >> soubor`<br>`příkaz 2>>log-soubor` |

| znak | význam a příklad použití |
|---|---|
| `<` | přesměrování vstupu<br><br>`příkaz < soubor` |
| `<<token` | přesměrování vstupu, čte ze skriptu až po *token*, který musí být samostatně na řádku – tzv. “here document”<br><br>`cat >soubor <<__END__`<br>`jakýkoli text, i $PROMENNA`<br>`kromě ukončovacího řádku`<br>`__END__`<br><br>Varianty:<br>`<<\token` quoting jako ’, dále žádná expanze<br>`<<-token` možno odsadit tabelátory |

## ❖ Zástupné znaky ve jménech souborů:

| znak | význam a příklad použití |
|---|---|
| `*` | zastupuje libovolnou sekvenci libovolných znaků mimo / a . na začátku jména (což lze ale ovlivnit proměnnými shellu), shell vyhledá všechna odpovídající jména souborů a nahradí jimi příslušný vzor – pokud žádné nenajde, expanzi neprovede (lze ovlivnit opět proměnnými shellu):<br>`ls *.c`<br>`ls *archiv*gz`<br>`ls .*/*.conf  # soubory s příponou conf ve skrytých adresářích`<br><br>**Poznámka**: Programy nemusí zpracovávat tyto expanzní znaky samy.<br>**Poznámka**: Pozor na limit délky příkazového řádku! |
| `?` | zastupuje 1 libovolný znak jména souboru (výjimky viz výše)<br>`ls x???.txt`<br>`ls soubor-?.txt` |
| `[množina]` | zastupuje jeden znak ze zadané množiny (výjimky viz výše)<br>`ls [A-Z]*`<br>`ls soubor-[1-9].txt` |

❖ Skládání příkazů:

| znak | význam a příklad použití |
|---|---|
| `\|` | přesměrování `stdout` procesu na `stdin` dalšího procesu, slouží pro vytváření *kolon* procesů-filtrů:<br>`ls \| more`<br>`cat /etc/passwd \|`<br>`  awk -F: '{print $1}' \| sort`<br>`příkaz \| tee soubor \| příkaz` |
| `‘příkaz‘` | je zaměněno za standardní výstup příkazu (command substitution)<br>`ls -l ‘which sh‘`<br>`DATUM=‘date +%Y-%m-%d‘  # ISO formát`<br>`echo Přihlášeno ‘who\|wc -l‘ uživatelů` |

| znak | význam a příklad použití |
|---|---|
| `;` | sekvence příkazů na jednom řádku<br><br>`ls ; echo ; ls /` |
| `\|\|` | provede následující příkaz, pokud předchozí neuspěl (`exitcode<>0`)<br><br>`cc program.c \|\| echo Chyba překladu` |
| `&&` | provede následující příkaz, pokud předchozí uspěl (`exitcode=0`)<br><br>`cc program.c && ./a.out` |

❖ Spouštění příkazů:

| znak | význam a příklad použití |
|---|---|
| `(příkazy)` | spustí *subshell*, který provede příkazy<br><br>`( echo "Text: "`<br>`  cat soubor`<br>`  echo "konec"`<br>`) > soubor2` |
| `&` | spustí příkaz *na pozadí* (pozor na výstupy programu)<br><br>`program &` |

## ❖ Rušení významu speciálních znaků (quoting):

- Znak \ ruší význam jednoho následujícího speciálního znaku (i znaku "nový řádek").

```
echo \* text \*   \\
echo "fhgksagdsahfgsjdagfjkdsaagjdsagjhfdsa\
jhdsajfhdsafljkshdafkjhadsk"
echo 5 \> 2 text \$TERM
```

- Uvozovky "" ruší význam speciálních znaků kromě: `$proměnná`, `‘příkaz‘` a \.

```
echo  3 * 4 = 12 # chyba, pokud
                 # jsou v adresáři soubory
echo "3 * 4 = 12"
echo "Dnešní datum: ‘date‘"
echo "PATH=$PATH"
echo "\ <\"test\"> *** ‘date‘ *** $PATH *** "
```

- Apostrofy ’’ ruší speciální význam všech znaků v řetězci.

```
echo ’$<>*‘ jakýkoli text kromě apostrofu ’
echo ’toto ->’\’’<- je apostrof’
echo ’*\** \" $PATH ‘ls‘ <> \’
```

# Postup při hledání příkazů

- Po zadání příkazu postupuje shell následovně:
  1. Test, zda se jedná o funkci nebo zabudovaný příkaz shellu (např. `cd`), a případné provedení této funkce/příkazu.
  2. Pokud se jedná o příkaz zadaný i s cestou (např. `/bin/sh`), pokus provést program s příslušným jménem v příslušném adresáři.
  3. Postupné prohlížení adresářů v `PATH`.
  4. Pokud program nenalezne nebo není spustitelný, hlásí chybu.

- Poznámka: Vlastní příkazy do `$HOME/bin` a přidat do `PATH`.

# *Vestavěné příkazy*

- Které příkazy jsou vestavěné závisí na použitém interpretu.
- Příklad: `cd`, `wait`, ...
- Výhoda: rychlost provedení.
- Ostatní příkazy jsou běžné spustitelné soubory.

# Příkaz `eval`

- `eval příkaz`:
  - Jednotlivé argumenty jsou načteny (a je proveden jejich rozvoj), výsledek je konkatenován, znovu načten (a rozvinut) a proveden jako nový příkaz.
  - Možnost za běhu sestavovat příkazy (tj. program) na základě aktuálně čteného obsahu souboru, vstupu od uživatele apod.

- Příklady:

```
echo "text > soubor"
eval echo "text > soubor"
eval ‘echo x=1‘ ; echo "x=$x"
```

# Ukončení skriptu

- Ukončení skriptu: `exit [číslo]`
  - vrací exit-code `číslo` nebo exit-code předchozího příkazu,
  - vrácenou hodnotu lze zpřístupnit pomocí `$?`,
  - možné hodnoty:
    - 0 – O.K.
    - <>0 – chyba

- Spuštění nového kódu: `exec příkaz`:
  - nahradí kód shellu provádějícího `exec` kódem daného příkazu,
  - spuštění zadaného programu je rychlé – nevytváří se nový proces,
  - bez parametru umožňuje přesměrování vstupu/výstupu uvnitř skriptu.

# Správa procesů

| | |
|---|---|
| `ps` | výpis stavu procesů |
| `nohup` | proces nekončí při odhlášení |
| `kill` | posílání signálů procesům |
| `wait` | čeká na dokončení potomka/potomků |

- Příklady:

```
ps ax           # všechny procesy
nohup program   # pozor na vstup/výstup
kill -9 1234    # nelze odmítnout
```

# Subshell

❖ Subshell se implicitně spouští v případě použití:

| | |
|---|---|
| `./skript.sh` | spuštění skriptu (i na pozadí) |
| `(příkazy)` | skupina příkazů |

❖ Subshell dědí proměnné prostředí, nedědí lokální proměnné (tj. ty, u kterých nebyl proveden `export`).

❖ Změny proměnných a dalších nastavení v subshellu se neprojeví v původním shellu!

❖ Provedení skriptu aktuálním interpretem:

- příkaz `.`
- např. `. skript`

❖ Posloupnost příkazů `{ příkazy }` – stejné jako `( )`, ale nespouští nový subshell.

❖ Příklad – možné použití `{ }` (a současně demonstrace jedné z programovacích technik používaných v shellu):

```
# Changing to a log directory.

cd $LOG_DIR

if [ "`pwd`" != "$LOG_DIR" ]  # or   if [ "$PWD" != "$LOG_DIR" ]
                              # Not in /var/log?
then
  echo "Cannot change to $LOG_DIR."
  exit $ERROR_CD
fi  # Doublecheck if in right directory, before messing with log file.

# However, a far more efficient solution is:

cd $LOG_DIR || {
  echo "Cannot change to $LOG_DIR." >&2
  exit $ERROR_CD;
}
```

# Proměnné

- Rozlišujeme proměnné:
  - lokální (nedědí se do subshellu)

```
PROM=hodnota
PROM2="hodnota s mezerami"
```

  - proměnné prostředí (dědí se do subshellu)

```
PROM3=hodnota
export PROM3   # musíme exportovat do prostředí
```

- Příkaz `export`:
  - `export seznam_proměnných`
  - exportuje proměnné do prostředí, které dědí subshell,
  - bez parametru vypisuje obsah prostředí.

❖ Přehled standardních proměnných:

| | |
|---|---|
| `$HOME` | jméno domovského adresáře uživatele |
| `$PATH` | seznam adresářů pro hledání příkazů |
| `$MAIL` | úplné jméno poštovní schránky pro e-mail |
| `$USER` | login jméno uživatele |
| `$SHELL` | úplné jméno interpretu příkazů |
| `$TERM` | typ terminálu (viz termcap/terminfo) |
| `$IFS` | obsahuje oddělovače položek na příkazové řádce – implicitně mezera, tabelátor a nový řádek |
| `$PS1` | výzva interpretu na příkazové řádce – implicitně znak `$` |
| `$PS2` | výzva na pokračovacích řádcích – implicitně znak > |

❖ Další standardní proměnné:

| | |
|---|---|
| `$$` | číslo = PID interpretu |
| `$0` | jméno skriptu (pokud lze zjistit) |
| `$1..$9` | argumenty příkazového řádku (dále pak `${n}` pro $n \geq 10$) |
| `$*`/`$@` | všechny argumenty příkazového řádku |
| `"$*"` | všechny argumenty příkazového řádku jako 1 argument v "" |
| `"$@"` | všechny argumenty příkazového řádku, individuálně v "" |
| `$#` | počet argumentů |
| `$?` | exit-code posledního příkazu |
| `$!` | PID posledního příkazu na pozadí |
| `$-` | aktuální nastavení shellu |

❖ Příklady:

```
echo "skript: $0"
echo první argument: $1
echo všechny argumenty: $*
echo PID=$$
```

❖ Použití proměnných:

| | |
|---|---|
| `$PROM text` | mezi jménem a dalším textem musí být oddělovací znak |
| `${PROM}text` | není nutný další oddělovač |
| `${PROM-word}` | `word` pokud nenastaveno |
| `${PROM+word}` | `word` pokud nastaveno, jinak nic |
| `${PROM=word}` | pokud nenastaveno, přiřadí a použije `word` |
| `${PROM?word}` | pokud nenastaveno, tisk chybového hlášení `word` a konec (exit) |

- Příkaz `env`:
  - `env nastavení_proměnných program [argumenty]`
  - spustí program s nastaveným prostředím,
  - bez parametrů vypíše prostředí.

- Proměnné pouze pro čtení:
  - `readonly seznam_proměnných`
  - označí proměnné pouze pro čtení,
  - subshell toto nastavení nedědí.

- Posun argumentů skriptu:
  - příkaz `shift`,
  - posune `$1 <- $2 <- $3` ...

# Čtení ze standardního vstupu

❖ Příkaz `read seznam_proměnných` čte řádek ze `stdin` a přiřazuje slova do proměnných, do poslední dá celý zbytek vstupního řádku.

❖ Příklady:

```
echo "x y z" | (read A B; echo "A='$A'  B='$B'")

IFS=","; echo "x,y z" | (read A B; echo "A='$A'  B='$B'")

IFS=":"; head -1 /etc/passwd | (read A B; echo "$A")
```

# Příkazy větvení

- Příkaz `if`:

```
if seznam příkazů
then
  seznam příkazů
elif seznam příkazů
then
  seznam příkazů
else
  seznam příkazů
fi
```

Příklad použití:

```
if [ -r soubor ]; then
    cat soubor
else
    echo soubor nelze číst
fi
```

# Testování podmínek

- Testování podmínek:
  - konstrukce `test výraz` nebo `[ výraz ]`,
  - výsledek je v `$?`.

| výraz | význam |
|---|---|
| `-d file` | je adresář |
| `-f file` | je obyčejný soubor |
| `-r file` | je čitelný soubor |
| `-w file` | je zapisovatelný soubor |
| `-x file` | je proveditelný soubor |
| `-t fd` | deskriptor `fd` je spojen s terminálem |
| `-n string` | neprázdný řetězec |
| `string` | neprázdný řetězec |
| `-z string` | prázdný řetězec |
| `str1 = str2` | rovnost řetězců |
| `str1 != str2` | nerovnost řetězců |

| výraz | význam |
|---|---|
| `int1 -eq int2` | rovnost čísel |
| `int1 -ne int2` | nerovnost čísel |
| `int1 -gt int2` | $>$ |
| `int1 -ge int2` | $>=$ |
| `int1 -lt int2` | $<$ |
| `int1 -le int2` | $<=$ |
| `! expr` | negace výrazu |
| `expr1 -a expr2` | and |
| `expr1 -o expr2` | or |
| `\( \)` | závorky |

## ❖ Příkaz case:

```
case výraz in
  vzor { | vzor }* )
    seznam příkazů
    ;;
esac
```

### Příklad použití:

```
echo -n "zadejte číslo: "
read reply
case $reply in
   "1")
        echo "1"
        ;;
   "2"|"4")
        echo "2 nebo 4"
        ;;
    *)
        echo "něco jiného"
        ;;
esac
```

# Cykly

❖ Cyklus `for`:

```
for identifikátor [ in seznam slov ]  # bez []: $1 ...
do
    seznam příkazů
done
```

Příklad použití:

```
    for i in *.txt ; do
       echo Soubor: $i
    done
```

## ❖ Cyklus `while`:

```
while seznam příkazů # poslední exit-code se použije
do
    seznam příkazů
done
```

### Příklad použití:

```
    while true ; do
        date; sleep 1
    done
```

## ❖ Cyklus `until`:

```
until seznam příkazů # poslední exit-code se použije
do
    seznam příkazů
done
```

❖ Ukončení/pokračování cyklu:

```
break, continue
```

❖ Příklady:

```
stop=ne
while [ "$stop" != ano ]; do
    echo -n "má skript skončit: "
    read stop
    echo $stop
    if [ "$stop" = ihned ] ; then
        echo "okamžité ukončení"
        break
    fi
done
```

# Zpracování signálů

- Příkaz `trap`:
  - `trap [příkaz] {signál}+`
  - při výskytu signálu provede příkaz,
  - pro ladění lze užít `trap příkaz DEBUG`.

- Příklad zpracování signálu:

```
#!/bin/sh

trap 'echo Ctrl-C; exit 1' 2  # ctrl-C = signál č.2

while true; do
   echo "cyklíme..."
   sleep 1
done
```

# Vyhodnocování výrazů

- Příkaz `expr výraz`:
  - Vyhodnotí výraz, komponenty musí být odděleny mezerami (pozor na quoting!).
  - Operace podle priority:

```
\*     /      %
+      -
=      \>     \>=    \<    \<=    !=
\&
\|
```

  - Lze použít závorky: `\( \)`

- Příklady:

```
V=‘expr 2 +  3  \* 4‘ ; echo $V

expr 1 = 1  \&  0 != 1 ; echo $?

expr "$P1" = "$P2" # test obsahu proměnných

V=‘expr $V + 1‘    # V++
```

❖ Řetězcové operace v `expr`:

`String : Regexp`
`match String Regexp`
– vrací délku podřetězce, který vyhovuje `Regexp`, nebo 0

`substr String Start Length`
– získá podřetězec od zadané pozice

`index String Charlist`
– vrací pozici prvního znaku ze seznamu, který se najde

`length String`
– vrací délku řetězce

# *Korn shell – `ksh`*

❖ Rozšíření Bourne shellu, starší verze `ksh88` základem pro definici POSIX, jeho důležité vlastnosti jsou zabudovány rovněž v bash-i.

❖ Příkaz `alias`: `alias rm='rm -i'`.

❖ Historie příkazů: možnost vracet se k již napsaným příkazům a editovat je (bash: viz šipka nahoru a dolů a `^R`).

❖ Vylepšená aritmetika:

- příkaz `let`, např. `let "x=2*2"`,
- operace: `+ - * / % ! < > <= >= == != = ++`,
- vyhodnocení bez spouštění dalšího procesu,
- zkrácený zápis:

```
(( x=2 ))
(( x=2*x ))
(( x++ ))
echo $x
```

❖ Vylepšené testování:

```
[[ ]]
(výraz)
výraz && výraz
výraz || výraz
```

Zbytek stejně jako `test`.

❖ Substituce příkazů:

`‘command‘` `$(command)`

❖ Speciální znak “vlnovka”:

| | | |
|---|---|---|
| `~` | `$HOME` | domovský adresář |
| `~user` | | domovský adresář daného uživatele |
| `~+` | `$PWD` | pracovní adresář |
| `~-` | `$OLDPWD` | předchozí prac. adresář |

❖ Primitivní menu:

```
select identifikátor [in seznam slov]
do
    seznam příkazů
done
```

– funguje jako cyklus; nutno ukončit!

❖ Pole:

```
declare -a p          # pole (deklarace je nepovinná)
p[1]=a
echo ${p[1]}
p+=(b c)              # přidání prvků
echo ${p[*]}
p=([1]=er [2]=rror) # celé pole
p+=([5]=c [6]=d)    # přidání na pozici

declare -A q          # asociativní pole
q[abc]=xyz
q[def]=mno
echo ${q[*]}
echo ${!q[*]}         # použité klíče
```

❖ Příkaz `printf`: formátovaný výpis na standardní výstup.

❖ Zásobník pro práci s adresáři:

- `pushd` – uložení adresáře do zásobníku,
- `popd` – přechod do adresáře z vrcholu zásobníku,
- `dirs` – výpis obsahu zásobníku.

## ❖ Příkaz `set`

- bez parametrů vypíše proměnné,
- jinak nastavuje vlastnosti shellu:

| parametr | akce |
|---|---|
| `-n` | neprovádí příkazy |
| `-u` | chyba pokud proměnná není definována |
| `-v` | opisuje čtené příkazy |
| `-x` | opisuje prováděné příkazy |
| `--` | další znaky jsou argumenty skriptu |

- vhodné pro ladění skriptů.

## ❖ Příklady:

```
set -x -- a b c *
for i ; do echo $i; done
```

## ❖ Zpracování přepínačů – `getopts`:

```
# Handling options a, b with a parameter, c.

while getopts :ab:c o
do      case "$o" in
        a)      echo "Option 'a' found.";;
        b)      echo "Option 'b' found with parameter '$OPTARG'.";;
        c)      echo "Option 'c' found.";;
        *)      echo "Use options a, b with a parameter, or c." >&2
                exit 1;;
        esac
done

((OPTIND--))

shift $OPTIND

echo "Remaining arguments: '$*'"
```

# Omezení zdrojů

- Restricted shell: zabránění shellu (a jeho uživateli) provádět jisté příkazy (použití `cd`, přesměrování, změna `PATH`, spouštění programů zadaných s cestou, použití `exec`...).

- `ulimit`: omezení prostředků dostupných shellu a procesům z něho spuštěným (počet procesů, paměť procesu, počet otevřených souborů, ...).

- `quota`: omezení diskového prostoru pro uživatele.

# Funkce

- Definice funkce:

```
function ident ()
{
    seznam příkazů
}
```

- Parametry jako u skriptu: `$1` ...
- Ukončení funkce s `exit-code`: `return [exit-code]`.
- Definice lokální proměnné: `typeset prom`.
- Možnost rekurze.

# Správa prací – job control

- Job (úloha) v shellu odpovídá prováděné koloně procesů (pipeline).
- Při spuštění kolony se vypíše `[jid] pid`, kde `jid` je identifikace úlohy a `pid` identifikace posledního procesu v koloně.
- Příkaz `jobs` vypíše aktuálně prováděné úlohy.
- Úloha může být spuštěna na popředí, nebo pomocí `&` na pozadí.
- Úloha běžící na popředí může být pozastavena pomocí `^Z` a přesunuta na pozadí pomocí `bg` (a zpět pomocí `fg`).
- Explicitní identifikace úlohy v rámci `fg`, `bg`, `kill`,...: %jid

# *Interaktivní a log-in shell*

❖ Shell může být spuštěn v různých režimech – pro bash máme dva významné módy, které se mohou kombinovat:

- interaktivní bash (parametr `-i`, typicky vstup/výstup z terminálu) a
- log-in shell (parametr `-l` či `-login`).

❖ Start, běh a ukončení interpretu příkazů závisí na režimu v němž shell běží. Např. pro interaktivní log-in bash platí:

1. úvodní sekvence: `/etc/profile` (existuje-li) a dále `~/.bash_profile`, `~/.bash_login`, nebo `~/.profile`,
2. tisk `$PS1`, zadávání příkazů,
3. `exit`,`^D`,`logout` – ukončení interpretu s provedením `~/.bash_logout`.

❖ Výběr implicitního interpretu příkazů:

- `/etc/passwd`
- `chsh` – change shell

# Shrnutí expanzí v shellu

- Při provádění příkazu shell provádí následující expanze:
  1. Zleva doprava rozvoj
     - složených závorek (např. `a{b,c,d}e` na `abe ace ade`) – není ve standardu,
     - vlnovek,
     - proměnných,
     - vložených příkazů a
     - aritmetických výrazů `$((...))`.
  2. Rozčlenění na argumenty dle `IFS`.
  3. Rozvoj jmen souborů.
  4. Odstranění kvotování.

# Utility UNIXu

❖ Utiliy UNIXu:

- užitečné programy (asi 150),
- součást normy SUSv3/v4,
- různé nástroje na zpracování textu atd.

❖ Přehled základních programů:

| | |
|---|---|
| `awk` | jazyk pro zpracování textu, výpočty atd. |
| `cmp` | porovnání obsahu souborů po bajtech |
| `cut` | výběr sloupců textu |
| `dd` | kopie (a konverze) části souboru |
| `bc` | kalkulátor s neomezenou přesností |

*Pokračování na dalším slajdu...*

## ❖ Přehled základních programů – pokračování...

| | |
|---|---|
| `df` | volné místo na disku |
| `diff` | rozdíl textových souborů (viz i `tkdiff`) |
| `du` | zabrané místo na disku |
| `file` | informace o typu souboru |
| `find` | hledání souborů |
| `grep` | výběr řádků textového souboru |
| `iconv` | překódování znakových sad |
| `nl` | očíslování řádků |
| `od` | výpis obsahu binárního souboru |
| `patch` | oprava textu podle výstupu `diff` |
| `sed` | neinteraktivní editor textu |
| `sort` | řazení řádků |
| `split` | rozdělení souboru na menší |
| `tr` | záměna znaků v souboru |
| `uniq` | vynechání opakujících se řádků |
| `xargs` | zpracování argumentů (např. po `find`) |

# Program `grep`

- Umožňuje výběr řádků podle regulárního výrazu.

- Existují tři varianty:
  - `fgrep` – rychlejší, ale neumí regulární výrazy
  - `grep` – základní regulární výrazy
  - `egrep` – rozšířené regulární výrazy

- Příklady použití:

```
fgrep -f seznam  soubor
grep '^ *[A-Z]'  soubor
egrep '(Jan|Honza) +Novák'  soubor
```

- Rozšířené (extended) regulární výrazy:

| znak | význam |
|---|---|
| + | $1-n$ výskytů předchozího podvýrazu |
| ? | $0-1$ výskyt předchozího podvýrazu |
| {$m$} | $m$ výskytů předchozího podvýrazu |
| {$m$,$n$} | $m-n$ výskytů předchozího podvýrazu |
| (*r*) | specifikuje podvýraz, např: `(ab*c)*` |
| \| | odděluje dvě varianty, např: `(ano\|ne)?` |

# Manipulace textu

- Program `cut` – umožňuje výběr sloupců textu.

```
cut -d: -f1,5 /etc/passwd
cut -c1,5-10 soubor  # znaky na pozici 1 a 5-10
```

- Program `sed`:
  - Neinteraktivní editor textu (streaming editor).
  - Kromě základních editačních operací umožňuje i podmíněné a nepodmíněné skoky (a tedy i cykly) a práci s registrem.

```
sed 's/novák/Novák/g'  soubor
sed 's/^[^:]*/-/'  /etc/passwd
sed -e "/$xname/p" -e "/|/d"  soubor_seznam
sed '/tel:/y/0123456789/xxxxxxxxxx/'  soubor
sed -n '3,7p'  soubor
sed '1a\
tento text bude přidán na 2. řádek'  soubor
sed -n '/start/,/stop/p' soubor
```

## ❖ Program `awk`:

- AWK je programovací jazyk vhodný pro zpracování textu (často strukturovaného do tabulek), výpočty atd.

```
awk '{s+=$1}END{print s}' soubor_cisel
awk '{if(NF>0){s+=$1;n++}}
     END{print n " " s/n}' soubor_cisel
awk -f awk-program  soubor
```

## ❖ Program `paste`:

- Spojení odpovídajících řádků vstupních souborů.

```
paste -d\| sloupec1.txt sloupec2.txt
```

# Porovnání souborů a patchování

- Program `cmp`:
  - Porovná dva soubory nebo jejich části byte po bytu.

```
cmp soubor1 soubor2
```

- Program `diff`:
  - Výpis rozdílů textových souborů (porovnává řádek po řádku).

```
diff old.txt new.txt
diff -C 2 old.c new.c
diff -urN dir1 dir2
```

- Program `patch`:
  - Změna textu na základě výstupu z programu `diff`.
  - Používá se pro správu verzí programů (cvs, svn, ...).

# Hledání souborů

- Program `find`:
  - Vyhledání souborů podle zadané podmínky a provedení určitých akcí nad nalezenými soubory.

```
find . -name '*.c'
find / -type d
find / -size +1000000c -exec ls -l {} \;
find / -size +1000000c -execdir command {} \;
find / -type f -size -2 -print0 | xargs -0 ls -l
find / -type f -size -2 -exec ls -l {} +
find . -mtime -1
find . -mtime +365 -exec ls -l {} \;
```

# Řazení

- Program `sort`:
  - seřazení řádků.

```
sort  soubor
sort  -u  -n  soubor_čísel
sort  -t:  -k3,3n  /etc/passwd
```

- Program `uniq`:
  - odstranění duplicitních řádků ze seřazeného souboru.

- Program `comm`:
  - výpis unikátních/duplicitních řádků seřazených souborů.

```
comm soubor1 soubor2 # 3 sloupce
comm  -1 -2  s1 s2   # jen duplicity
comm  -2 -3  s1 s2   # pouze v s1
```

# Další nástroje programátora

- skriptovací jazyky a interprety (perl, python, tcl, ...)
- překladače (cc/gcc, c++/g++, ...)
- assemblery (nasm, ...)
- linker `ld` (statické knihovny `.a`, dynamické knihovny `.so`)
  - výpis dynamických knihoven používaných programem: `ldd`,
  - knihovny standardně v `/lib` a `/usr/lib`,
  - cesta k případným dalším knihovnám: `LD_LIBRARY_PATH`,
  - run-time sledování funkcí volaných z dynamických knihoven: `ltrace`.

- Program `make`:
  - automatizace (nejen) překladu a linkování,
  - příklad souboru `makefile` (pozor na odsazení tabelátory):

```
test: test.o tisk.o
        gcc $(CFLAGS) -o test test.o tisk.o

test.o: test.c
        gcc $(CFLAGS) -c test.c

tisk.o: tisk.c
        gcc $(CFLAGS) -c tisk.c

clean:
        rm -f *.o test
```

  - použití: `make`, `make CFLAGS=-g`, `make clean`.

- automatizovaná konfigurace – GNU `autoconf`:
  - Generuje na základě šablony založené na volání předpřipravených maker skripty pro konfiguraci překladu (určení platformy, ověření dostupnosti knihoven a nástrojů, nastavení cest, ...), překlad a instalaci programů šířených ve zdrojové podobě.
  - Používá se mj. spolu s `automake` (usnadnění tvorby makefile) a `autoscan` (usnadnění tvorby šablon pro `autoconf`).
  - Použití vygenerovaných skriptů: `./configure`, `make`, `make install`
- ladění: debugger např. `ddd` postavený na `gdb` (překlad s ladícími informacemi `gcc -g ...`)
- sledování volání jádra: `strace`
- profiling: profiler např. `gprof` (překlad pomocí `gcc -pg ...`)

# Operační systémy

IOS 2015/2016

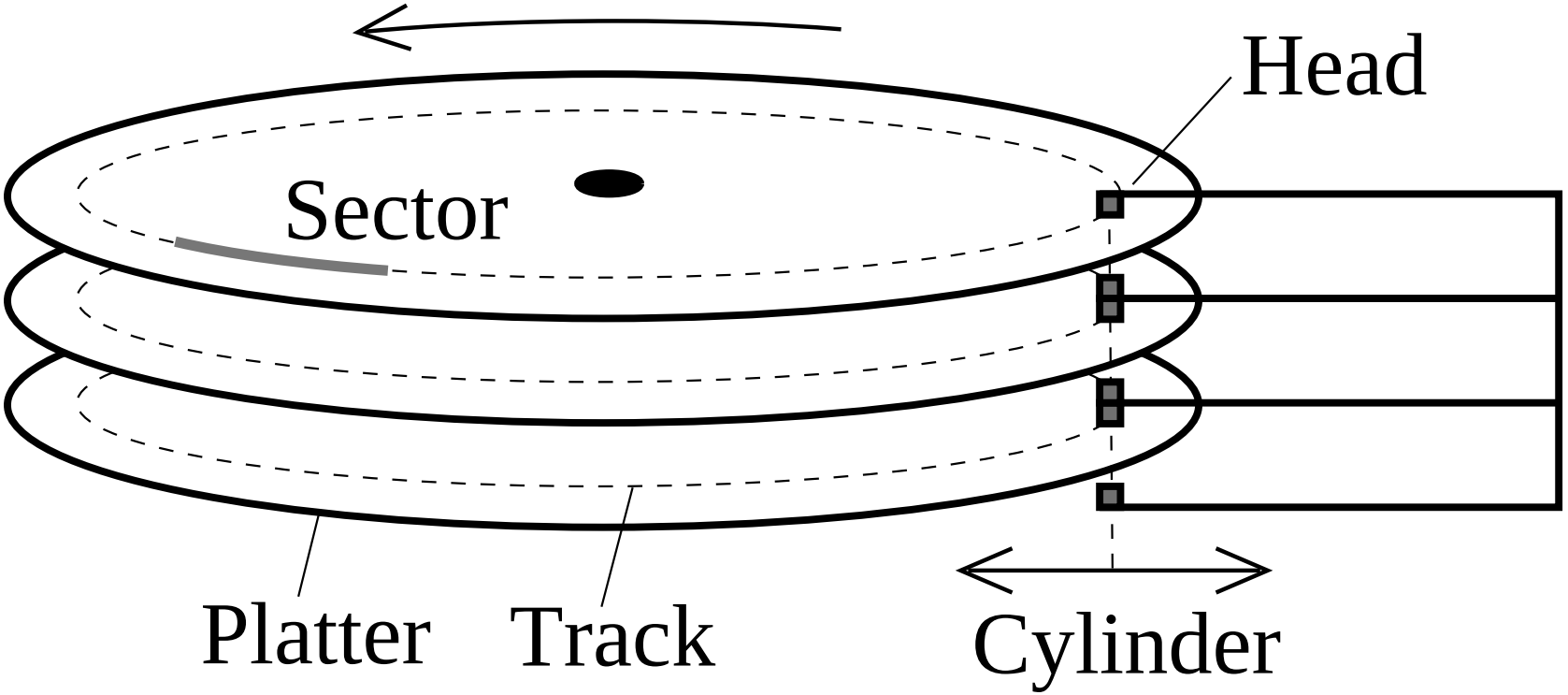
**Tomáš Vojnar**

vojnar@fit.vutbr.cz

**Vysoké učení technické v Brně**
**Fakulta informačních technologií**
**Božetěchova 2, 612 66 Brno**

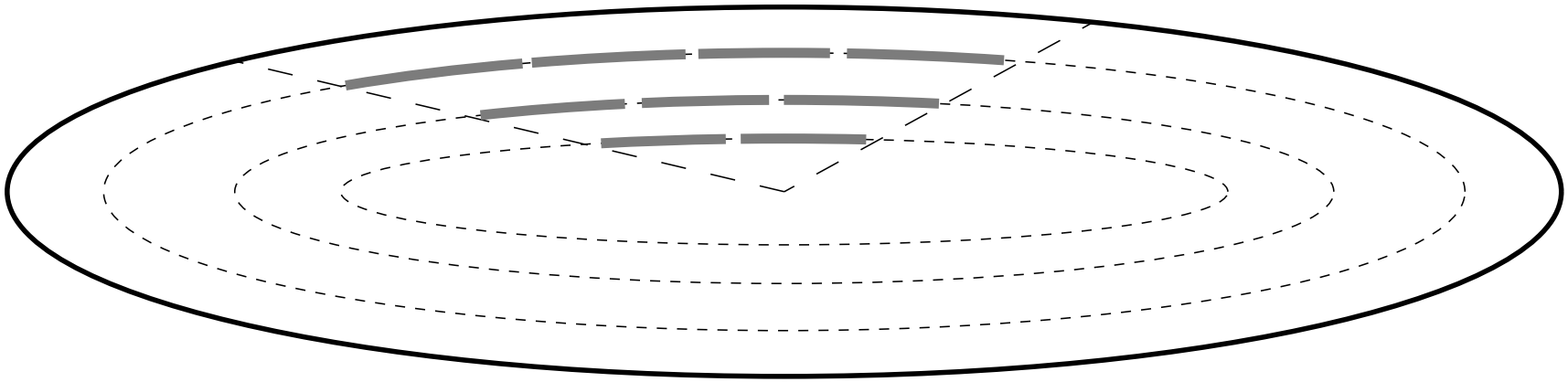
# Správa souborů

# Pevný disk




- Diskový sektor: nejmenší jednotka, kterou disk umožňuje načíst/zapsat.
- Velikost sektoru: typicky 512B, u nových disků 4096B (s emulací 512B).
- Adresace sektorů:
  - CHS = Cylinder, Head (typicky 1-6 hlav), Sector
  - LBA = Linear Block Address (číslo 0..N)

❖ Rozložení sektorů na pevném disku:



❖ Zejména dříve se používalo tzv. prokládání (interleaving) sektorů – z hlediska čísla za sebou následující sektory nebyly fyzicky uloženy za sebou. To proto, aby přes pomalý řadič disku a pomalou komunikaci se systémem nehrozilo, že se za sebou následující sektory nestačí načíst v rámci jedné otáčky disku a bude se muset čekat na další otočku disku, kdy se příslušný sektor vrátí opět zpět.

❖ Pro připojení disků se používá řada různých diskových (či obecněji periferních) rozhraní: primárně ATA (IDE)/SATA či SCSI/SAS, ale také USB, FireWire, FibreChannel, Thunderbolt aj.

❖ Diskové sběrnice se liší mj. rychlostí (např. do 16Gbit/s u SATA, 12 Gbit/s SAS), počtem připojitelných zařízení (desítky SATA/65535 SAS), max. délkou kabelů (1-2m SATA, 10m SAS), architekturou připojení (např. více cest k zařízení u SAS), podporovanými příkazy (flexibilita při chybách).

❖ Stejným způsobem jako disky mohou být zpřístupněny i jiné typy pamětí: flash disky, SSD, pásky, CD/DVD, ...

❖ Vzniká hierarchie pamětí, ve které stoupá kapacita a klesá rychlost a cena/B:

- primární paměť: RAM (nad ní ještě registry, cache L1-L3)
- sekundární paměť: pevné disky, SSD (mají také své cache)
- terciární paměť: pásky, CD, DVD, ...

# Parametry pevných disků

❖ Přístupová doba = doba vystavení hlav + rotační zpoždění.

❖ Typické parametry současných disků (orientačně – neustále se mění):

| | |
|---|---|
| kapacita | do 10 TB |
| průměrná doba přístupu | od nízkých jednotek ms |
| otáčky | 4200-15000/min |
| přenosová rychlost | desítky až nízké stovky MB/s |

❖ U kapacity disku udávané výrobcem/prodejcem je třeba dávat pozor, jakým způsobem ji počítá: GB = $10^9$B nebo $1000 * 2^{20}$B nebo ... Správně: GiB = $1024^3$ = $2^{30}$B.

❖ U přenosových rychlostí pozor na sustained transfer rate (opravdové čtení z ploten) a maximum transfer rate (z bufferu disku).

❖ Možnost měření přenosových rychlostí: `hdparm -t`.

- `hdparm` umožňuje číst/měnit celou řadu dalších parametrů disků.
- Pozor! `hdparm -T` měří rychlost přenosu z vyrovnávací paměti OS (tedy z RAM).

# Solid State Drive – SSD

❖ SSD je nejčastěji založeno na nevolatilních pamětech NAND flash, ale vyskytují se i řešení založená na DRAM (se zálohovaným napájením) či na kombinacích.

❖ Výhody SSD:

- rychlý (v zásadě okamžitý) náběh,
- náhodný přístup – přístupová doba od jednotek $\mu$s (DRAM) do desítek či nízkých stovek $\mu$s,
- větší přenosové rychlosti – stovky MB/s (do cca 600 MB/s), zápis může být mírně pomalejší (viz dále).
- tichý provoz, mechanická a magnetická odolnost, ...,
- obvykle nižší spotřeba (neplatí pro DRAM).

❖ Nevýhody SSD:

- vyšší cena za jednotku prostoru (dříve i nižší kapacita, dnes již až do 15 TB),
- omezený počet přepisů (ne příliš významné pro běžný provoz),
- možné komplikace se zabezpečením (např. bezpečné mazání/šifrování přepisem dat vyžaduje speciální podporu – data mohla být diskem při přepisech zapsána na několik míst).

# Problematika zápisu u SSD

❖ NAND flash SSD jsou organizovány do stránek (typicky 4KiB) a ty do bloků (typicky 128 stránek, tj. 512KiB).

❖ Prázdné stránky lze zapisovat jednotlivě. Pro přepis nutno načíst celý blok do vyrovnávací paměti, v ní změnit, na disku vymazat a pak zpětně zapsat.

- Problém je menší při sekvenčním než při náhodném zápisu do souboru.

❖ Řešení problémů s přepisem v SSD:

- Aby se problém minimalizoval, SSD může mít více stránek, než je oficiální kapacita.
- Příkaz TRIM umožňuje souborovému systému sdělit SSD, které stránky nejsou používány a lze je opět považovat za prázdné.
- Novější řadiče SSD provádí samy uvolňování stránek na základě dostupných dat (např. na základě informací o stránkách přesouvaných interně v SSD).
- Přesto jistý rozdíl v rychlosti čtení/zápisu může zůstat.
- TRIM navíc nelze užít vždy (souborové systémy uložené jako obrazy, kde nelze uvolňovat bloky, které nejsou na samém konci obrazu; podobně u RAID či databází ukládajících si data do velkého předalokovaného souboru).

❖ Aby řadič SSD minimalizoval počet přepisů stránek, může přepisovanou stránku zapsat na jinou pozici; případně i přesouvá dlouho neměněné stránky.

# Zabezpečení disků

❖ Disková elektronika používá ECC = Error Correction Code: k užitečným datům sektoru si ukládá redundantní data, která umožňují opravu (a z hlediska OS transparentní realokaci), nebo alespoň detekci chyb.

❖ S.M.A.R.T. – Self Monitoring Analysis and Reporting Technology: moderní disky si automaticky shromažďují řadu statistik, které lze použít k předpovídání/diagnostice chyb. Viz `smartctl`, `smartd`, ...

❖ Rozpoznávání a označování vadných bloků (bad blocks) může probíhat také na úrovni OS (např. `e2fsck` a `badblocks`), pokud si již s chybami disk sám neporadí (což je ale možná také vhodná doba disk raději vyměnit).

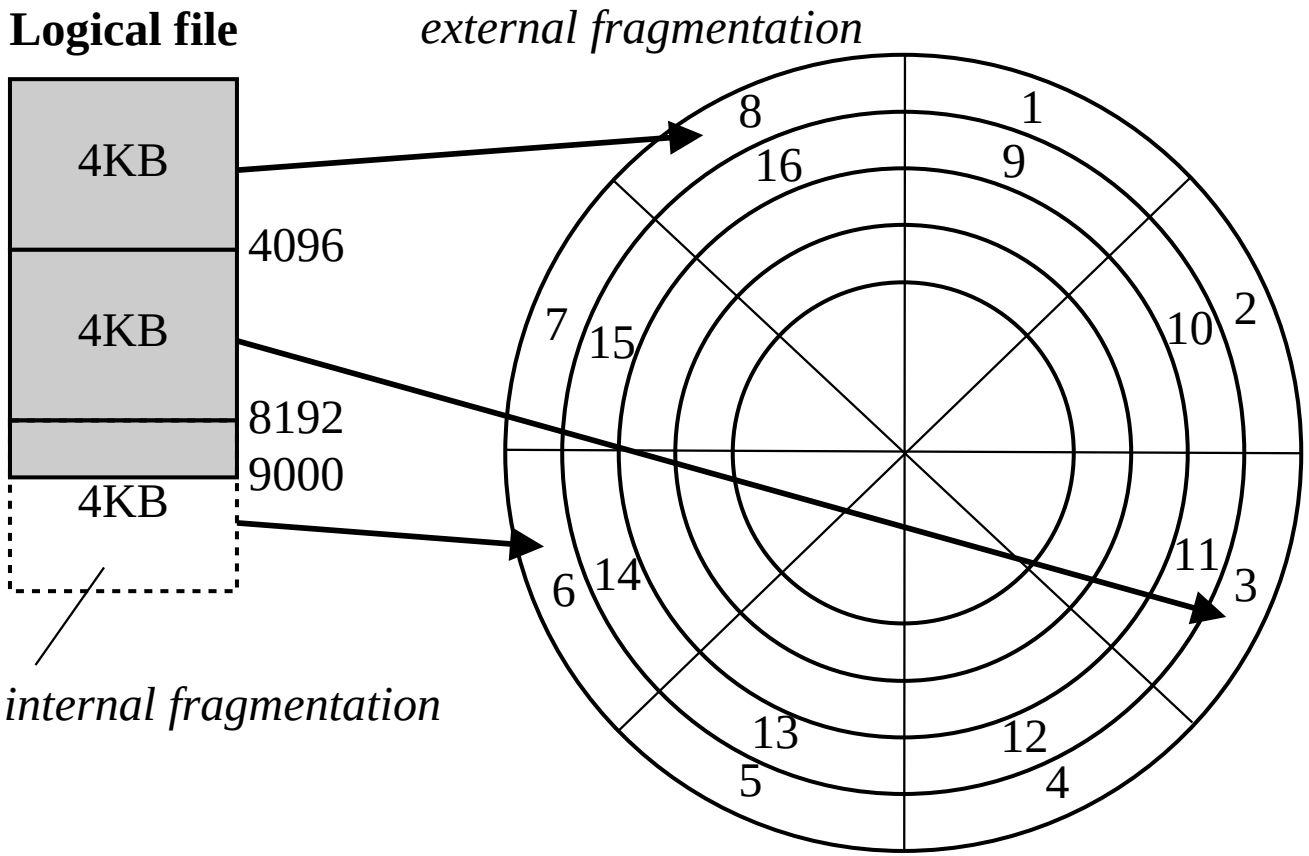
# Disková pole

❖ RAID (Redundant Array of Independent Disks):

- RAID 0 – *disk striping*, následné bloky dat rozmístěny na různých discích, vyšší výkonnost, žádná redundance.
- RAID 1 – *disk mirroring*, všechna data ukládána na dva disky, velká redundance (existuje také RAID 0+1 a RAID 1+0/10).
- RAID 2 – data rozdělena mezi disky po bitech, použito zabezpečení Hammingovým kódem uloženým na zvláštních discích (např. 3 bity zabezpečení pro 4 datové: chybu na 1 disku lze automaticky opravit, na 2 discích detekovat).
- RAID 3 – sekvence bajtů dat jsou rozděleny na části uložené na různých discích, navíc je užit disk s paritami.
- RAID 4 – bloky dat na různých discích a paritní bloky na zvláštním disku.
- RAID 5 – jako RAID 4, ale paritní a datové bloky jsou rozloženy na všech discích, redukce kolizí u paritního disku při zápisu.
- RAID 6 – jako RAID 5, ale parita uložena 2x, vyrovná se i se ztrátou 2 disků.

# Uložení souboru na disku

❖ Alokační blok: skupina pevného počtu sektorů, typicky $2^n$ pro nějaké $n$, následujících logicky (tj. v souboru) i fyzicky (tj. na disku) za sebou, která je nejmenší jednotkou diskového prostoru, kterou OS čte či zapisuje při běžných operacích.




❖ Poznámka: Někdy se též užívá označení cluster.

# Fragmentace

❖ Při přidělování a uvolňování prostoru pro soubory dochází k tzv. externí fragmentaci – na disku vzniká posloupnost volných oblastí a oblastí použitých různými soubory (může tedy existovat i na plně obsazeném disku), což má dva možné důsledky:

- Vzniknou některé nevyužité oblasti příliš malé na to, aby se daly využít (např. pro spojité přidělování prostoru souborům, které je aktuálně zapotřebí uložit).
- Při nespojitém přidělování prostoru po alokačních blocích (nebo při nespojitém přidělování bez dolního omezení využitelného prostoru) výše uvedený problém nevzniká, ale data souboru jsou na disku uložena nespojitě – složitější a pomalejší přístup (menší vliv u SSD, ale i tam se může projevit).
- Moderní souborové systémy užívají různé techniky k minimalizaci externí fragmentace: rozložení souborů po disku, předalokace (alokuje se více místa, než je momentálně zapotřebí), odložená alokace (odkládá zápis, než se nasbírá více požadavků a je lepší povědomí, kolik je třeba alokovat).
- Přesto bývají k dispozici nástroje pro defragmentaci.

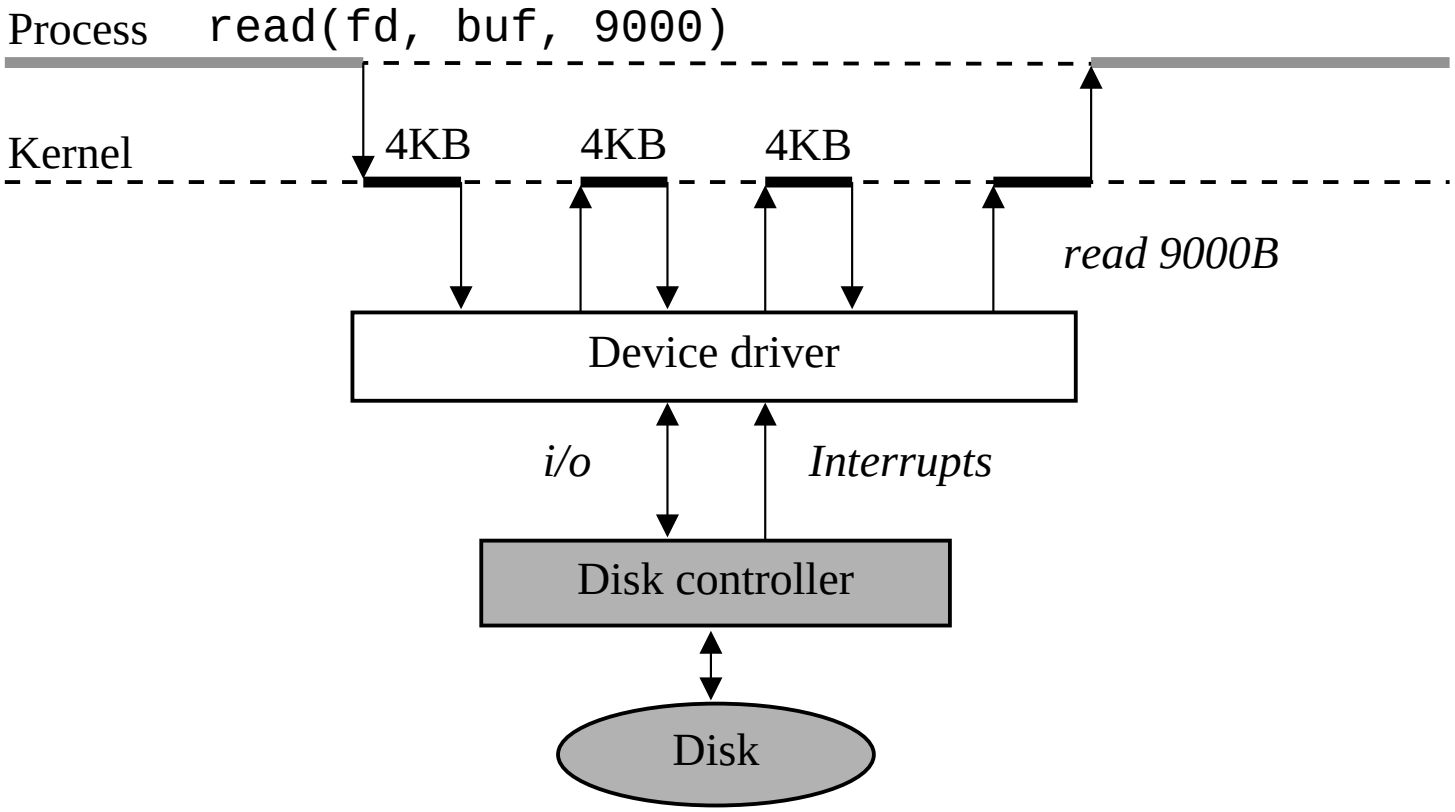
❖ Interní fragmentace – nevyužité místo v posledním přiděleném alokačním bloku – plýtvání místem.

- Některé souborové systémy umožňují sdílení posledních alokačních bloků více soubory.

# Přístup na disk

❖ Prostřednictvím I/O portů a/nebo paměťově mapovaných I/O operací (HW zajišťuje, že některé adresy RAM ve skutečnosti odkazují do interní paměti I/O zařízení) se řadiči disku předávají příkazy definované diskovým rozhraním (ATA, SCSI, ...).

❖ Přenos z/na disk je typicky řízen řadičem disku s využitím technologie přímého přístupu do paměti (DMA). O ukončení operací či chybách informuje řadič procesor (a na něm běžící jádro OS) pomocí přerušení.



# Plánování přístupu na disk

- Pořadí bloků čtených/zapisovaných na disk ovlivňuje plánovač diskových operací.
  - Přicházející požadavky na čtení/zápis jsou ukládány do vyrovnávací paměti a jejich pořadí je případně měněno tak, aby se minimalizovala režie diskových operací.
  - Např. tzv. výtahový algoritmus (elevator algorithm, SCAN algorithm) pohybuje hlavičkami od středu k okraji ploten a zpět a vyřizuje požadavky v pořadí odpovídajících pozici a směru pohybu hlaviček.
  - Další plánovací algoritmy: Circular SCAN (vyřizuje požadavky vždy při pohybu jedním směrem – rovnoměrnější doba obsluhy), LOOK (pohybuje se jen v mezích daných aktuálními požadavky – nižší průměrná doba přístupu), C-LOOK, ...
  - Plánovač může sdružovat operace, vyvažovat požadavky různých uživatelů, implementovat priority operací, odkládat operace v naději, že je bude možno později propojit, implementovat časová omezení možného čekání operací na provedení apod.

- V Linuxu možno zjistit/změnit nastavení prostřednictvím `/sys/block/<devicename>/queue/scheduler`.

# Logický disk

- Dělení fyzického disku na logické disky – diskové oblasti (partitions):
  - Na systémech PC tzv. MBR (Master Boot Record) obsahuje tabulku diskových oblastí s 1-4 primárními diskovými oblastmi, jedna z nich může být nahrazena rozšířenou diskovou oblastí,
  - Rozšířená disková oblast se dělí na logické diskové oblasti popsané EBR (Extended Boot Record) nacházejícími se v jejich prvním sektoru a vytvářejícími zřetězený seznam.
  - Pro správu diskových oblastí lze užít programy `cfdisk`, `fdisk`, `gparted`, ...
  - LVM = Logical Volume Manager: umožňuje tvorbu logických disků přesahujících hranice fyzického disku, snadnou změnu velikosti, přidávání a ubírání disků, tvorbu snímků, ...

- Formátování – program `mkfs`; existuje (existovalo) také nízkoúrovňové formátování.

- Kontrola konzistence souborového systému: program `fsck`.

❖ Různé typy souborových systémů: fs, ufs, ufs2, ext2, ext3, ext4, btrfs, ReiserFS, HFS+ (Mac OS X), XFS (od Silicon Graphics, původně pro IRIX), JFS (od IBM, původně pro AIX), ZFS (od Sunu, původně pro OpenSolaris), HPFS, FAT, VFAT, FAT32, NTFS, ReFS, F2FS, ISO9660 (Rock Ridge, Joliet), UDF, Lustre (Linuxové clustry a superpočítače), GPFS (clustry a superpočítače), ...

❖ Virtuální souborový systém (VFS) – vrstva, která zastřešuje všechny použité souborové systémy a umožňuje pracovat s nimi jednotným, abstraktním způsobem.

❖ Síťové souborové systémy: NFS, ...

❖ Speciální souborové systémy: procfs, sysfs (souborové systémy informující o dění v systému a umožňující nastavení jeho parametrů), tmpfs (souborový systém alokující prostor v RAM a sloužící pro ukládání dočasných dat), ...

# Žurnálování

- Žurnál slouží pro záznam modifikovaných metadat (příp. i dat) před jejich zápisem na disk.
  - Obvykle implementován jako cyklicky přepisovaný buffer ve speciální oblasti disku.
  - Operace pokryté žurnálováním jsou atomické – vytváří transakce: buď uspějí všechny jejich dílčí kroky nebo žádný (např. mazání souboru v UNIXU znamená odstranění záznamu z adresáře, pak uvolnění místa na disku).
- Systémy souborů se žurnálem: ext3, ext4, ufs, ReiserFS, XFS, JFS, NTFS, ....
- Umožňuje spolehlivější a rychlejší návrat do konzistentního stavu po chybách.
- Data obvykle nejsou žurnálována (byť mohou být): velká režie.
- Kompromis mezi žurnálováním a nežurnálováním dat představuje předřazení zápisu dat na disk před zápis metadat do žurnálu: zajistí konzistenci při chybě během zápisu dat na konec souboru (částečně zapsaná nová data nebudou uvažována).

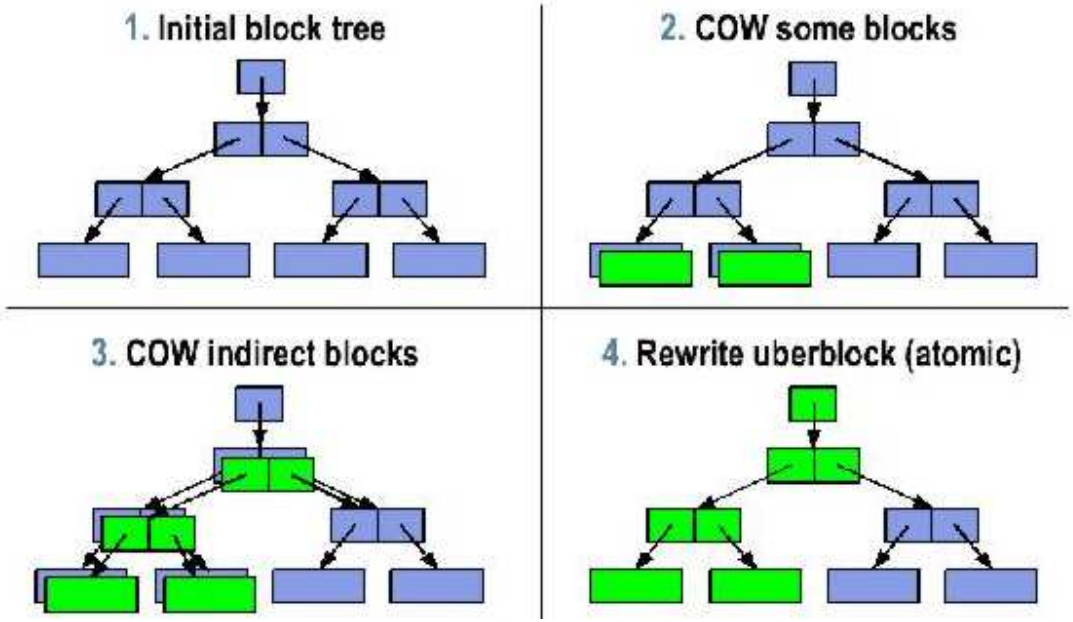
# *Implementace žurnálování*

- Implementace na základě dokončení transakcí (REDO), např. ext3/4:
  - sekvence dílčích operací se uloží nejprve do žurnálu mezi značky označující začátek a konec transakce (příp. spolu s kontrolním součtem),
  - poté se dílčí operace provádí na disku,
  - uspějí-li všechny dílčí operace, transakce se ze žurnálu uvolní,
  - při selhání se dokončí všechny transakce, které jsou v žurnálu zapsány celé (a s korektním kontrolním součtem).
- Implementace na základě anulace transakcí (UNDO):
  - záznam dílčích operací do žurnálu a na disk se prokládá,
  - proběhne-li celá transakce, ze žurnálu se uvolní,
  - při chybě se eliminují nedokončené transakce.
- UNDO a REDO je možno kombinovat (NTFS).
- Implementace žurnálování musí zajišťovat správné pořadí zápisu operací, které ovlivňuje plánování diskových operací v OS a také případně jejich přeuspořádání v samotném disku.

# *Alternativy k žurnálování*

❖ Copy-on-write (např. ZFS, btrfs) – nejprve zapisuje nová data či metadata na disk, pak je zpřístupní:

- Změny provádí hierarchicky v souladu s hierarchickým popisem obsahu disku (jde o vyhledávací strom popisující rozložení dat a metadat na disku, ne adresářový strom; data vyhledává na základě unikátní identifikace souborů a posuvu v nich).
- Začne měněným uzlem, vytvoří jeho kopii a upraví ji. Potom vytvoří kopii uzlu nadřazeného změněnému uzlu, upraví ji tak, aby odkazovala přílušným odkazem na uzel vytvořený v předchozím kroku atd.
- Na nejvyšší úrovni se udržuje několik verzí kořenového záznamu se zabezpečovacím kódem a časovými razítky. Po chybě bere kořen s nejnovějším časovým razítkem a správným kontrolním součtem.



# *Alternativy k žurnálování*

❖ Poznámka: CoW nabízí rovněž bázi pro implementaci:

- snímků souborového systému (uložení stavu v určitém okamžiku s možností pozdějšího návratu – stačí zálohovat si starší verze kořene a z něj dostupné záznamy)
- klonů souborového systému (vytvoření kopií, které jsou v budoucnu samostatně manipulovány – vzniká několik kopií kořene, které se dále mění samostatně).

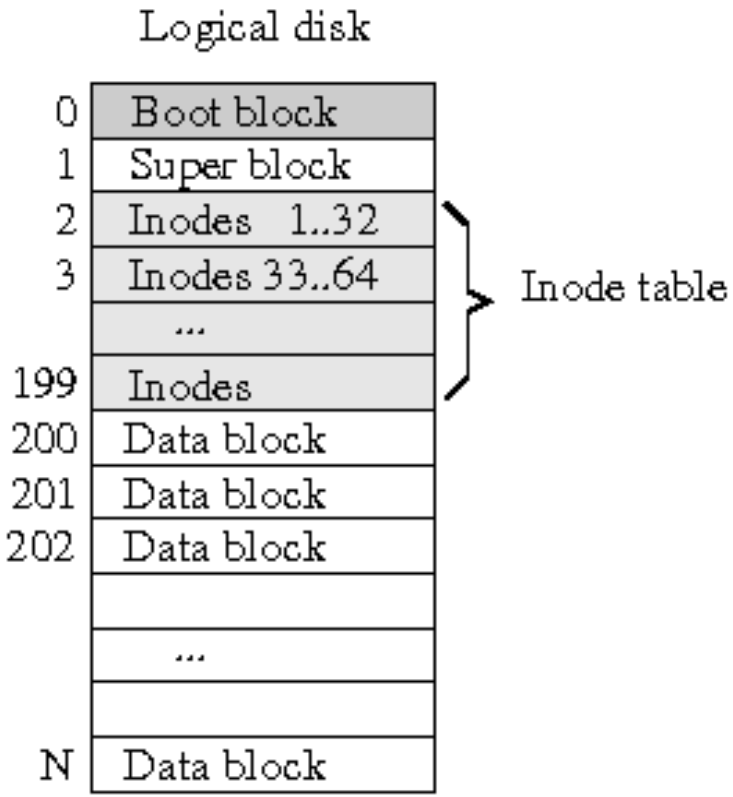
V obou případech vznikají částečně sdílené stromové struktury popisujících různé verze obsahu souborového systému.

❖ Další možnosti:

- Soft updates (např. UFS): sleduje závislosti mezi změněnými metadaty a daty a zaručuje zápis na disk v takovém pořadí, aby v kterékoli době byl obsah disku konzistentní (až na možnost vzniku volného místa považovaného za obsazené).
- Log-structured file systems (LFS, UDF, F2FS): celý souborový systém má charakter logu (zapsaného v cyklicky přepisované frontě) s obsahem disku vždy přístupným přes poslední záznam (a odkazy z něj).

# Klasický UNIXový systém souborů (FS)

| | |
|---|---|
| boot blok | pro zavedení systému při startu |
| super blok | informace o souborovém systému (typ, velikost, počet i-uzlů, volné místo, volné i-uzly, kořenový adresář,...) |
| tabulka i-uzlů | tabulka s popisy souborů |
| datové bloky | data souborů a bloky pro nepřímé odkazy |




❖ Modifikace základního rozložení FS v navazujících souborových systémech:

- Disk rozdělen do skupin bloků.
- Každá skupina má své i-uzly a datové bloky a také svůj popis volných bloků: lepší lokalita.
- Superblok se základními informacemi o souborovém systému je rovněž uložen vícenásobně.

# *i-uzel*

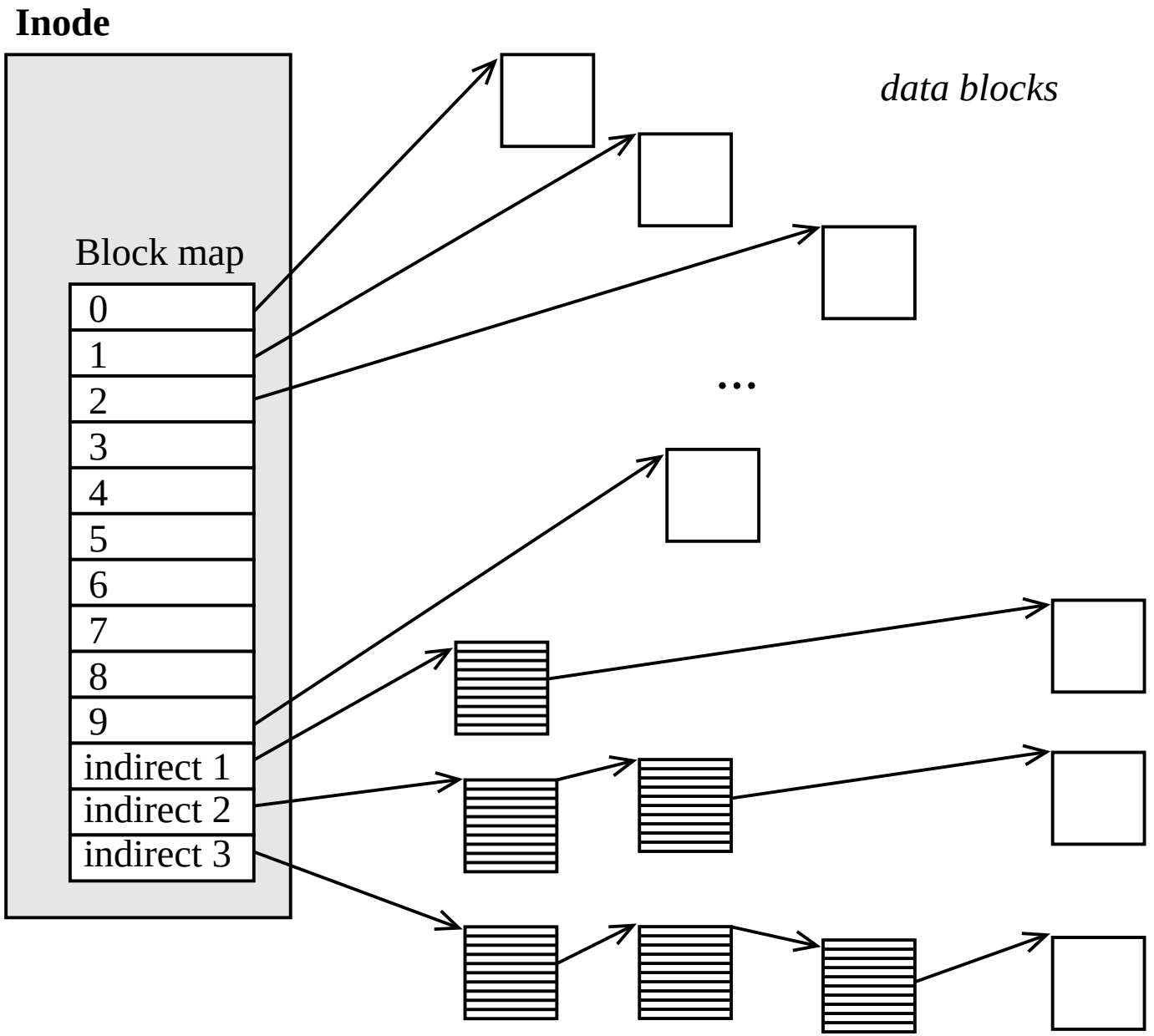
❖ Základní datová struktura popisující soubor v UNIXu.

❖ i-uzel obsahuje metadata:

- stav i-uzlu (alokovaný, volný)
- typ souboru (obyčejný, adresář, zařízení, ...)
- délka souboru v *bajtech*
- mtime = čas poslední modifikace dat
- atime = čas posledního přístupu
- ctime = čas poslední modifikace i-uzlu
- UID = identifikace vlastníka (číslo)
- GID = identifikace skupiny (číslo)
- přístupová práva (číslo, například 0644 znamená rw-r–r–)
- počet pevných odkazů (jmen)
- tabulka odkazů na datové bloky:

| | |
|---|---|
| 10 | přímých odkazů (12 u ext2, ...) |
| 1 | nepřímý odkaz první úrovně |
| 1 | nepřímý odkaz druhé úrovně |
| 1 | nepřímý odkaz třetí úrovně |

- (odkaz na) další informace (ACL, extended attributes, dtime, ...)


Inode
Block map
0
1
2
3
4
5
6
7
8
9
indirect 1
indirect 2
indirect 3
data blocks
...

❖ Teoretický limit velikosti souboru:

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

kde:

- $N = D/M$ je počet odkazů v bloku, je-li $M$ velikost odkazu v bajtech (běžně 4B),
- $D$ je velikost bloku v bajtech (běžně 4096B).

❖ Velikost souborů je omezena také dalšími strukturami FS, VFS, rozhraním jádra a architekturou systému (32b/64b) – viz Large File System support: podpora souborů $> 2GiB$.

❖ Co vypisují programy o velikosti souborů?

- `du soubor` – zabrané místo v blocích, včetně režie,
- `ls -l soubor` – velikost souboru v bajtech,
- `df` – volné místo na namontovaných discích.

❖ Zpřístupnění i-uzlu:

- `ls -i soubor` – číslo i-uzlu souboru soubor,
- `ils -e /dev/... n` – výpis i-uzlu $n$ na `/dev/...`

❖ Základní informace o souborovém systému ext2/3/4: `dumpe2fs`.

❖ Architektura souborových systémů je ovlivňována snahou o minimalizaci jejich režie při průchodu, přesunu v souboru (seek), zvětšování/zmenšování souboru:

- snadnost vyhledání adresy prvního/určitého bloku souboru,
- snadnost vyhledání lineárně následujících bloků,
- snadnost přidání/ubrání dalších bloků,
- snadnost alokace/dealokace volného prostoru (informace o volných oblastech, minimalizace externí fragmentace).

❖ FS (a řada jeho následníků UFS, ext2, ext3) představuje kompromis s ohledem na převážně malé soubory.

- U větších souborů nutno procházet/modifikovat větší objem metadat.

❖ Další optimalizace pro malé soubory: data přímo v i-uzlu (např. u symbolických odkazů definovaných dostatečně krátkou cestou, tzv. fast symlinks).

# Jiné způsoby organizace souborů

❖ Kontinuální uložení: jedna spojitá posloupnost na disku.

- Problémy se zvětšováním souborů díky externí fragmentaci nebo obsazení prostoru hned za koncem souboru.

❖ Zřetězené seznamy bloků: každý datový blok obsahuje kromě dat odkaz na další blok (nebo příznak konce souboru).

- Při přístupu k náhodným blokům či ke konci souboru (změna velikosti) nutno projít celý soubor.
- Chyba kdekoliv na disku může způsobit ztrátu velkého objemu dat (rozpojení seznamu).

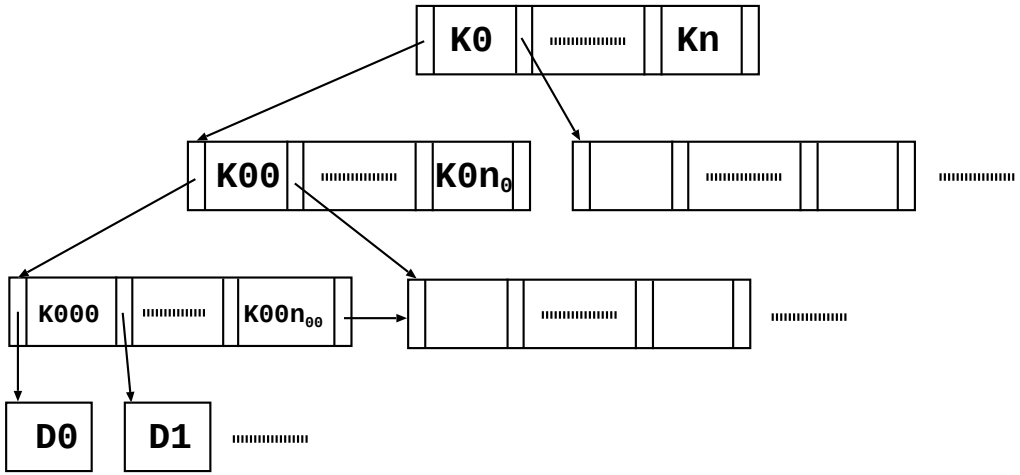
❖ FAT (File Allocation Table): seznamy uložené ve speciální oblasti disku. Na začátku disku je (pro vyšší spolehlivost zdvojená) tabulka FAT, která má položku pro každý blok. Do této tabulky vedou odkazy z adresářů. Položky tabulky mohou být zřetězeny do seznamů, příp. označeny jako volné či chybné.

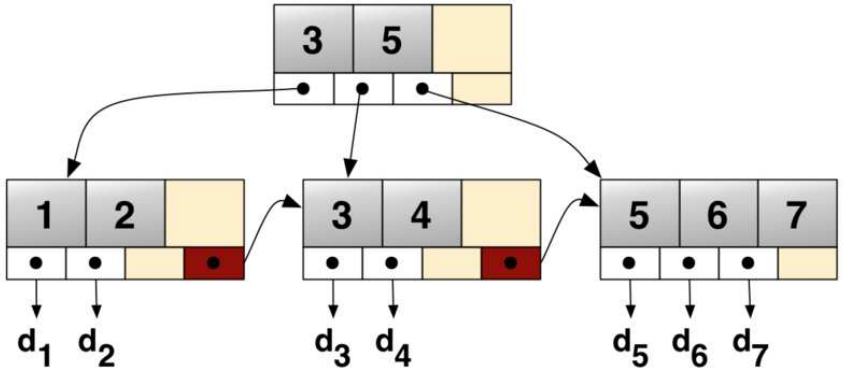
- Opět vznikají problémy s náhodným přístupem.

# Jiné způsoby organizace souborů

❖ B+ stromy:

- Vnitřní uzly obsahují sekvenci $link_0, key_0, link_1, key_1, ..., link_n, key_n, link_{n+1}$, kde $key_i < key_{i+1}$ pro $0 \leq i < n$. Hledáme-li záznam s klíčem $k$, pokračujeme $link_0$, je-li $k < key_0$; jinak $link_i$, $1 \leq i \leq n$, je-li $key_{i-1} \leq k < key_i$; jinak užijeme $key_{n+1}$.
- Listy mají podobnou strukturu. Je-li $key_i = k$ pro nějaké $0 \leq i \leq n$, $link_i$ odkazuje na hledaný záznam. Jinak hledaný záznam neexistuje.
- Poslední odkaz $link_{n+1}$ v listech je užit k odkazu na následující listový uzel pro urychlení lineárního průchodu indexovanými daty.



# *Jiné způsoby organizace souborů*

❖ B+ stromy:

- Vkládá se na listové úrovni. Dojde-li k přeplnění, list se rozštěpí a přidá se nový odkaz do nadřazeného vnitřního uzlu. Při přeplnění se pokračuje směrem ke kořeni. Nakonec může být přidán nový kořen.
- Ruší se od listové úrovně. Klesne-li zaplněnost sousedních uzlů na polovinu, uzly se spojí a ruší se jeden odkaz na nadřazené úrovni. Rušení může pokračovat směrem ke kořeni. Nakonec může jedna úroveň ubýt.

❖ B+ stromy a jejich různé varianty jsou použity pro popis diskového prostoru přiděleného souborům v různých moderních souborových systémech:

- XFS, JFS, ReiserFS, btrfs, ...,
- v omezené podobě tzv. stromů extentů v ext4 (pouze pět úrovní, bez vyvažování, bez zřetězení listů), podobná struktura je i v NTFS.

# Jiné způsoby organizace souborů

❖ V moderních systémech se často indexuje alokovaný prostor po tzv. extentech, tj. posloupnostech proměnného počtu bloků jdoucích za sebou logicky v souboru a uložených i fyzicky na disku za sebou:

- zrychluje se práce s velkými soubory: menší, lépe vyvážené indexové struktury; menší objem metadat, které je třeba procházet a udržovat; lepší lokalita dat i metadat.

❖ Extenty jsou použity ve všech výše zmíněných systémech s B+ stromy a jejich variantami.

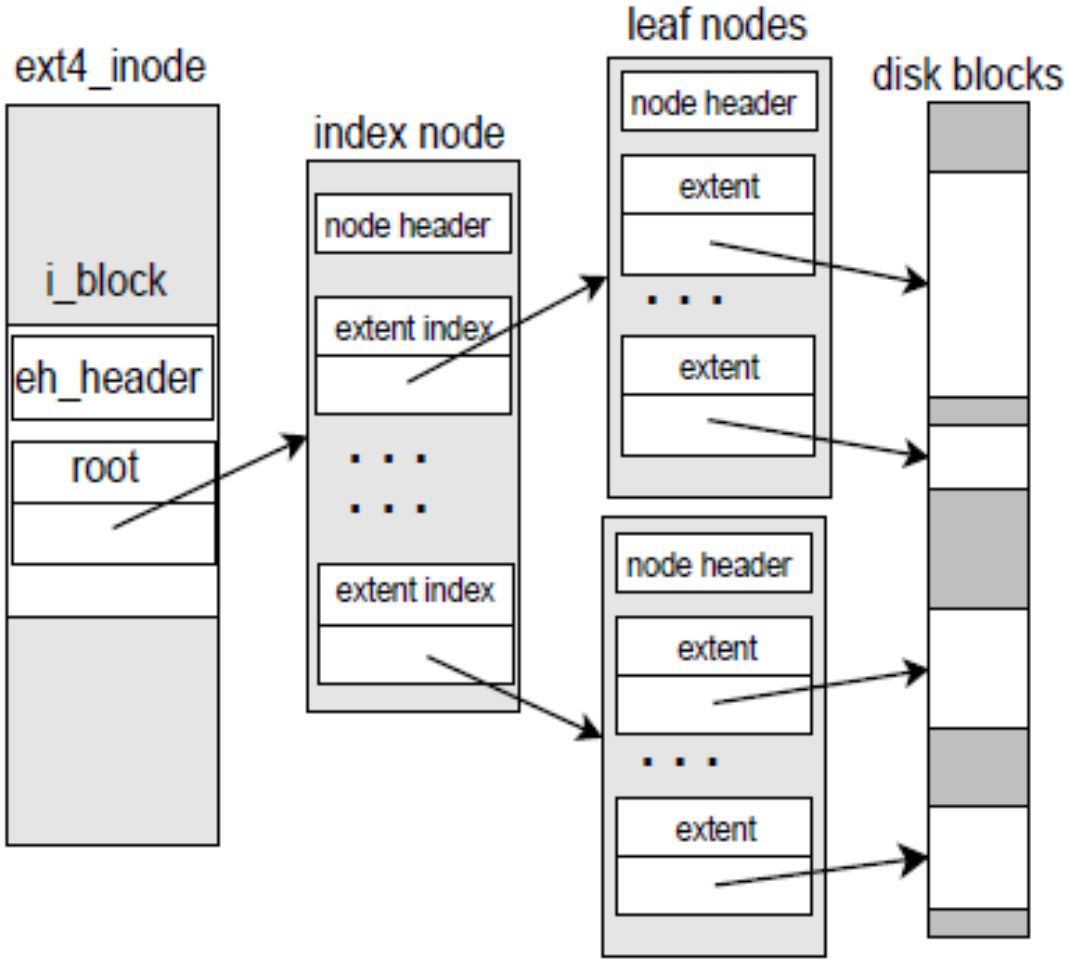
- B+ stromy se snadno kombinují s extenty. To neplatí pro klasický Unixový strom, který není kompatibilní s adresováním jednotek proměnné velikosti.

❖ Lineárnímu průchodu může pomoci prolinkování listů vyhledávacích stromů, je-li použito.

❖ Pro malé soubory může B+ strom představovat zbytečnou režii: používá se přímé uložení v i-uzlu nebo přímé odkazy na extenty z i-uzlu (do určitého počtu).

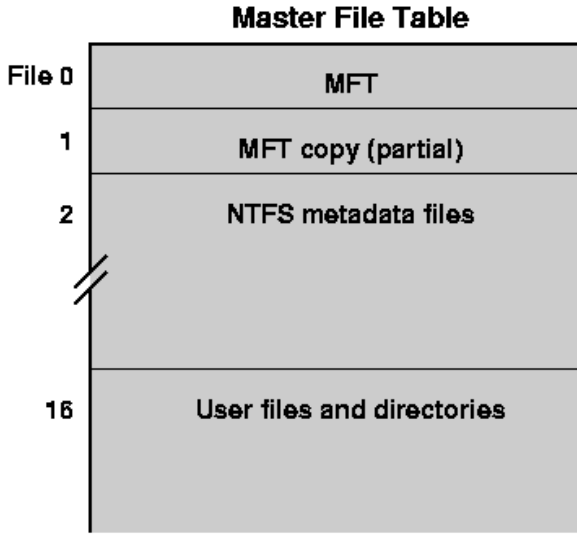
# Ext4

❖ Strom extentů – v principu B+ strom degradovaný na max. 5 úrovní bez vyvažování a bez zřetězení listů:



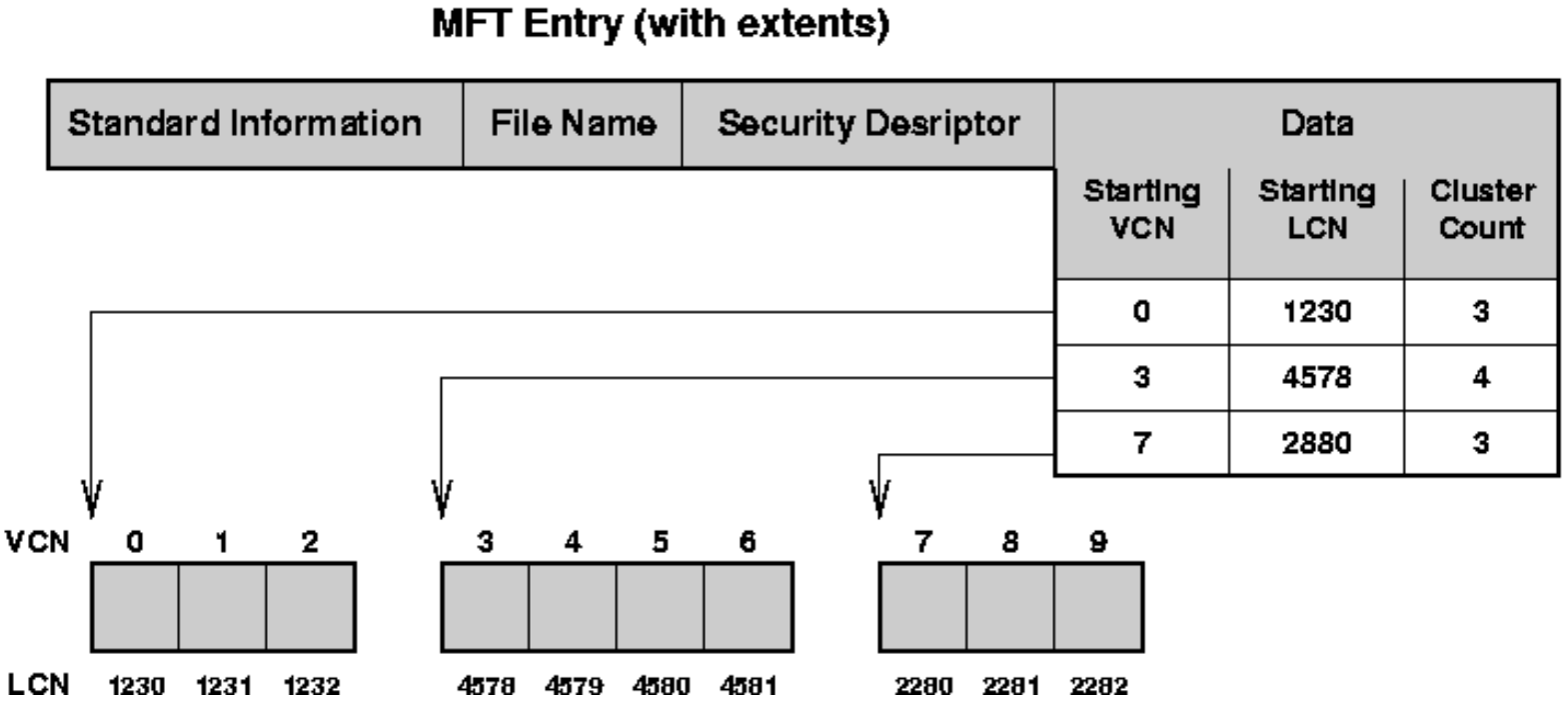

❖ Malé soubory: až 4 extenty odkazované příimo z kořenového uzlu extentového stromu umístěného v i-uzlu, příp. přímo v i-uzlu (symbolické odkazy).

# NTFS




❖ MFT – Master File Table: alespoň jeden řádek pro každý soubor.

❖ Obsah souboru přímo v záznamu MFT odpovídajícím příslušnému souboru, nebo rozdělen na extenty odkazované z tohoto záznamu, nebo z pomocných MFT záznamů odkazovaných z primárního MFT záznamu ve stylu B+ stromu.



# Organizace volného prostoru

❖ Organizace volného prostoru v klasickém Unixovém FS a řadě jeho následovníků (UFS, ext2, ext3) a také v NTFS: bitová mapa s jedním bitem pro každý blok.

- Umožňuje zrychlit vyhledávání volné souvislé oblasti pomocí bitového maskování (test volnosti několika bloků současně).

❖ Další způsoby organizace volného prostoru:

- seznam,
- označení (zřetězení) volných položek v tabulce bloků (FAT),
- B+ strom (adresace velikostí a/nebo offsetem),
- někdy se také – jako u btrfs – eviduje jen obsazený prostor podle pozice na disku (a případně pouze v paměti se vytváří pomocné struktury pro efektivnější vyhledávání – např. red-black stromy bitových map).

❖ Volný prostor může být také organizován po extentech.

# Typy souborů v UNIXu

- Příkaz `ls -l` vypisuje typ jako první znak na řádku:

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

# Adresář

- Soubor obsahující množinu dvojic – "hard-links" – `(jméno souboru, číslo souboru)`:
  - `jméno souboru`
    - mělo v tradičním UNIXu délku max 14 znaků,
    - dnes je typicky až 255 znaků,
    - může obsahovat jakékoli znaky kromě `'/'` a `'\0'`,
  - `číslem souboru` je u klasického Unixového FS (a souborových systémů z něj odvozených) číslo i-uzlu, které je indexem do tabulky i-uzlů logického disku (v jiných případech může sloužit jako klíč pro vyhledávání v B+ stromu apod.).

- Adresář vždy obsahuje jména:
  - `.` odkaz na sebe
  - `..` odkaz na rodičovský adresář

- Poznámka: rychlost vyhledávání/vkládání: seznam, B+ stromy a jejich varianty: NTFS, XFS,JFS, btrfs, HFS+, ext3/4 (H-stromy: 1 nebo 2 úrovně, neužívá vyvažování, vyhledává na základě zahashovaného jména), ...

❖ Soubor v Unixu může mít více jmen:

- `ln jmeno-existujiciho-souboru nove-jmeno`

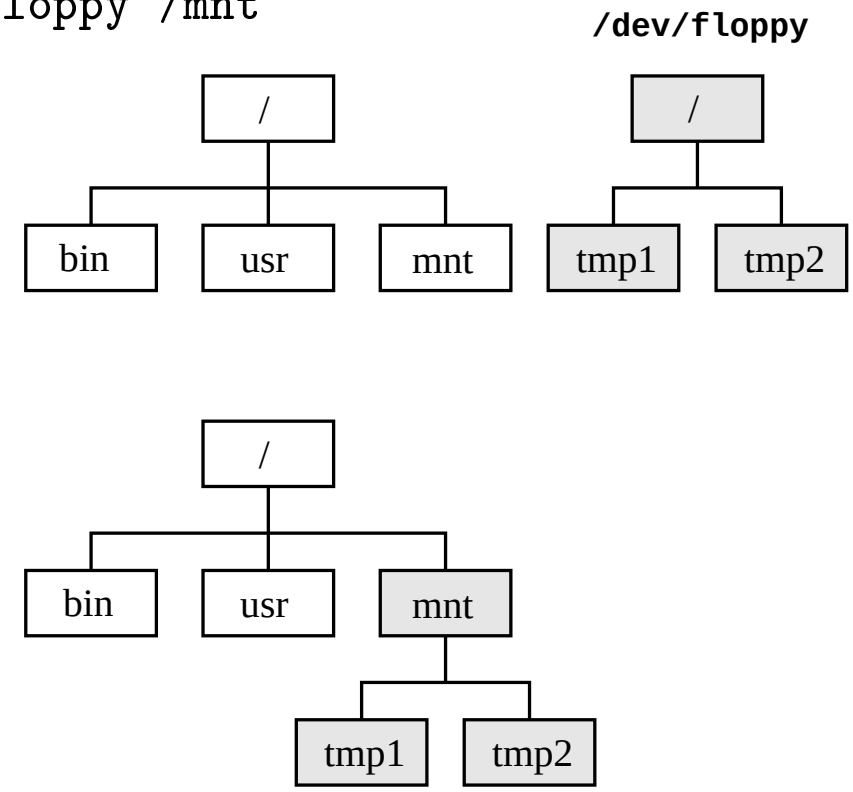
❖ Omezení: Obě jména musí být v rámci jednoho logického disku!

❖ Rušení souboru (`rm soubor`) ruší pevný odkaz `(jméno, číslo i-uzlu)` a snižuje počitadlo odkazů v i-uzlu. Dokud je počitadlo nenulové, soubor se nemaže.

# Montování disků

- Princip montování disků:
  - Všechny soubory jsou v jednom “stromu” adresářů.
  - V systému je jeden kořenový logický disk, další logické disky lze připojit programem `mount` do již existujícího adresářového stromu.

- Příklad: `mount /dev/floppy /mnt`



❖ Poznámky:

- Parametry příkazu `mount` – viz `man mount`.
- Soubor `/etc/fstab` – popis disků typicky připojovaných na určité pozice adresářového stromu.
- Soubor `/etc/mtab` – tabulka aktuálně připojených disků.
- Některé novější technologie umožnují automatické montování nově připojených zařízení. Např. udev dynamicky vytváří rozhraní souborového systému na zařízení v adresáři `/dev` a informuje zbytek systému prostřednictvím sběrnice D-Bus, aplikace nautilus pak může provést automatické montování a další akce (parametry může zjišťovat automaticky, čerpat z různých nastavení zúčastněných technologií, ale přednost má stále `/etc/fstab`).
- Automounter – automaticky připojuje potřebné disky při pokusu o přístup na pozici adresářového stromu, kam by měly být připojeny, a také po určité době neaktivity disky odpojuje (výhodné zejména u sítových souborových systémů).
- Union mount:
  - Montuje více disků (adresářů) do jednoho místa – obsah je pak sjednocením obsahu namontovaných adresářů s tím, že se vhodným způsobem řeší kolize (např. prioritou zdrojových disků/adresářů).
  - Plan9, Linux – UnionFS, ...
  - UnionFS: má copy-on write sémantiku: soubor původně v read-only větvi, při změně se uloží do read-write větve s vyšší prioritou.

# Symbolické odkazy

```
ln -s existující-soubor symbolický-odkaz
```

- V datech souboru typu “symlink” je jméno jiného cílového souboru.
- Jádro při otevření souboru automaticky provede otevření cílového souboru.
  - Nutné vícenásobné zpracování cesty (cesta k symlinku, cesta uvnitř symlinku).
- Po zrušení cílového souboru zůstává symlink nezměněn – přístup k souboru přes něj vede k chybě.
- Symlink může odkazovat na i jiný logický disk.
- Řešení cyklů: omezený počet úrovní odkazů.
- Rychlé symlinky: uloženy v i-uzlu, pomalé symlinky: uloženy ve zvláštním souboru (užívá se tehdy, je-li cesta, která definuje symlink, příliš dlouhá pro uložení do i-uzlu).

# Blokové a znakové speciální soubory

❖ Blokové a znakové speciální soubory implementují souborové rozhraní k fyzickým či virtuálním zařízením.

| soubor | tvoří souborové rozhraní na zařízení |
|---|---|
| /dev/hda | první fyzický disk (master) na prvním ATA/PATA rozhraní (dříve) |
| /dev/hda1 | první logický disk (partition) na hda |
| /dev/sda | první fyzický disk SCSI či emulované SCSI (SATA/PATA/usb flash) |
| /dev/mem | fyzická paměť |
| /dev/zero | nekonečný zdroj nulových bajtů |
| /dev/null | soubor typu "černá díra"– co se zapíše, to se zahodí; při čtení se chová jako prázdný soubor |
| /dev/random | generátor náhodných čísel |
| /dev/tty | terminál |
| /dev/lp0 | první tiskárna |
| /dev/mouse | myš |
| /dev/dsp | zvuková karta |
| /dev/loop | souborové systémy nad soubory (`losetup`/`mount -o loop=...`) |
| ..... | ..... |

❖ Poznámka: Názvy závisí na použitém systému (Linux).

❖ Výhoda zavedení speciálních souborů: Programy mohou použít běžné souborové rozhraní pro práci se soubory i na čtení/zápis z různých zařízení.

❖ Příklady práce se speciálními soubory:

```
dd if=/dev/hda of=mbrbackup bs=512 count=1
cat /dev/hda1 | gzip >zaloha-disku.gz
cp /dev/zero /dev/hda1 # vynulování disku
```

# Přístupová práva

- V UNIXu jsou typicky rozlišena práva pro vlastníka, skupinu a ostatní.
  (Rozšíření: ACL (access control lists), viz `man acl`, `man setfacl`...)

- Uživatelé:
  - Uživatele definuje administrátor systému (root): `/etc/passwd`,
  - UID: číslo identifikující uživatele (root UID = 0).
  - Příkaz `chown` – změna vlastníka souboru (pouze root).

- Skupiny:
  - Skupiny definuje administrátor systému (root): `/etc/group`,
  - GID: číslo identifikující skupinu uživatelů,
  - Uživatel může být členem více skupin, jedna z nich je aktuální (používá se při vytváření souborů).
  - Příkaz
    - `groups` – výpis skupin uživatele,
    - `chgrp` – změna skupiny souboru,
    - `newgrp` – nový shell s jiným aktuálním GID.

# Typy přístupových práv

| obyčejné soubory | |
|---|---|
| `r` | právo číst obsah souboru |
| `w` | právo zapisovat do souboru |
| `x` | právo spustit soubor jako program |

| adresáře | |
|---|---|
| `r` | právo číst obsah (`ls adresář`) |
| `w` | právo zapisovat = vytváření a rušení souborů |
| `x` | právo přistupovat k souborům v adresáři (`cd adresář`, `ls -l adresář/soubor`) |

❖ Příklad: `-rwx---r--` (číselné vyjádření: `0704`):

- obyčejný soubor,

vlastník: čtení, zápis, provedení

- skupina: nemá žádná práva

ostatní: pouze čtení

❖ Změna přístupových práv – příkaz `chmod`:

```
chmod a+rw soubory   # všichni mohou číst i zapisovat
chmod 0644 soubor    # rw-r--r--
chmod -R u-w .       # zakáže zápis vlastníkovi
chmod g+s soubor     # nastaví SGID -- viz dále
```

❖ Výpis informací o souboru:

```
ls -l soubor

-rw-r--r-- 1 joe joe  331 Sep 24 13:10 .profile
typ
 práva
          počet pevných odkazů
            vlastník
                skupina
                     velikost
                          čas poslední modifikace
                                       jméno souboru
```

# Sticky bit

❖ Sticky bit je příznak, který nedovoluje rušit cizí soubory v adresáři, i když mají všichni právo zápisu.

```
chmod +t adresar      # nastaví Sticky bit
chmod 1777 /tmp
```

❖ Příklad: `/tmp` má práva `rwxrwxrwt`

# SUID, SGID

❖ Určení práv pro procesy:

| | |
|---|---|
| UID | reálná identifikace uživatele = kdo spustil proces |
| EUID | efektivní UID se používá pro kontrolu přístupových práv (pro běžné programy je rovno UID) |
| GID | reálná identifikace skupiny = skupina toho, kdo spustil proces |
| EGID | efektivní GID se používá pro kontrolu přístupových práv (pro běžné programy je rovno GID) |

❖ Vlastník programu může propůjčit svoje práva komukoli, kdo spustí program s nastaveným SUID.

❖ Příklad: Program `passwd` musí editovat soubor `/etc/shadow`, do kterého má právo zápisu pouze superuživatel `root`.

❖ Příklad propůjčených přístupových práv: `-rwsr-Sr-x fileUID fileGID`

- `s` = je nastaveno `x`, `S` = není nastaveno `x`,
- v našem příkladu `s`: SUID=set user identification, EUID:=fileUID
- v našem příkladu `S`: SGID=set group identification: EGID:=fileGID

# *Typická struktura adresářů v UNIXu*

- FHS = Filesystem Hierarchy Standard

| | |
|---|---|
| /bin | programy pro všechny (nutné při bootování) |
| /dev | obsahuje speciální soubory – rozhraní na zařízení |
| /etc | konfigurační soubory pro systém i aplikace |
| /home | domovské adresáře uživatelů |
| /lib | sdílené knihovny, moduly jádra (nutné při bootování) |
| /proc | obsahuje informace o procesech |
| /root | domovský adresář superuživatele |
| /sbin | programy pro superuživatele (nutné při bootování) |
| /tmp | dočasné pracovní soubory |

*Pokračování na další straně...*

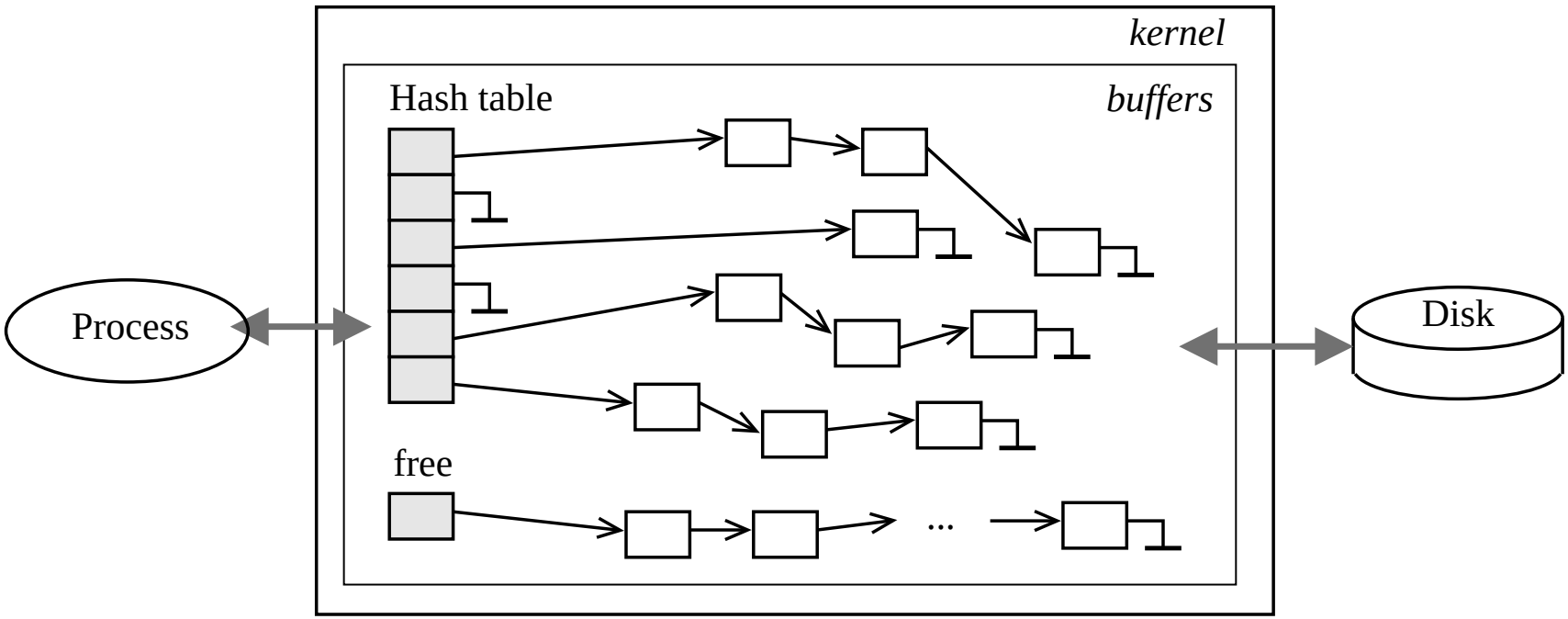
*Typická struktura adresářů v UNIXu – pokračování:*

| | |
|---|---|
| /usr | obsahuje soubory, které nejsou nutné při zavádění systému – může se přimontovat až po bootu (například ze sítě) a může být pouze pro čtení (například na CD) |
| /usr/bin,sbin | programy, které nejsou třeba pro bootování |
| /usr/lib | knihovny (statické i dynamické) |
| /usr/include | hlavičkové soubory pro jazyk C atd. |
| /usr/share | soubory, které lze sdílet (například přes síť) nezávisle na architektuře počítače |
| /usr/local | další hierarchie bin, sbin, lib,... určená pro lokální (nestandardní) instalace programů |
| /usr/src | zdrojové texty jádra systému a programů |
| /var | obsahuje soubory, které se mění při běhu systému |
| /var/log | záznamy o činnosti systému |
| /var/spool | pomocné soubory pro tisk atd. |
| /var/mail | poštovní přihrádky uživatelů |

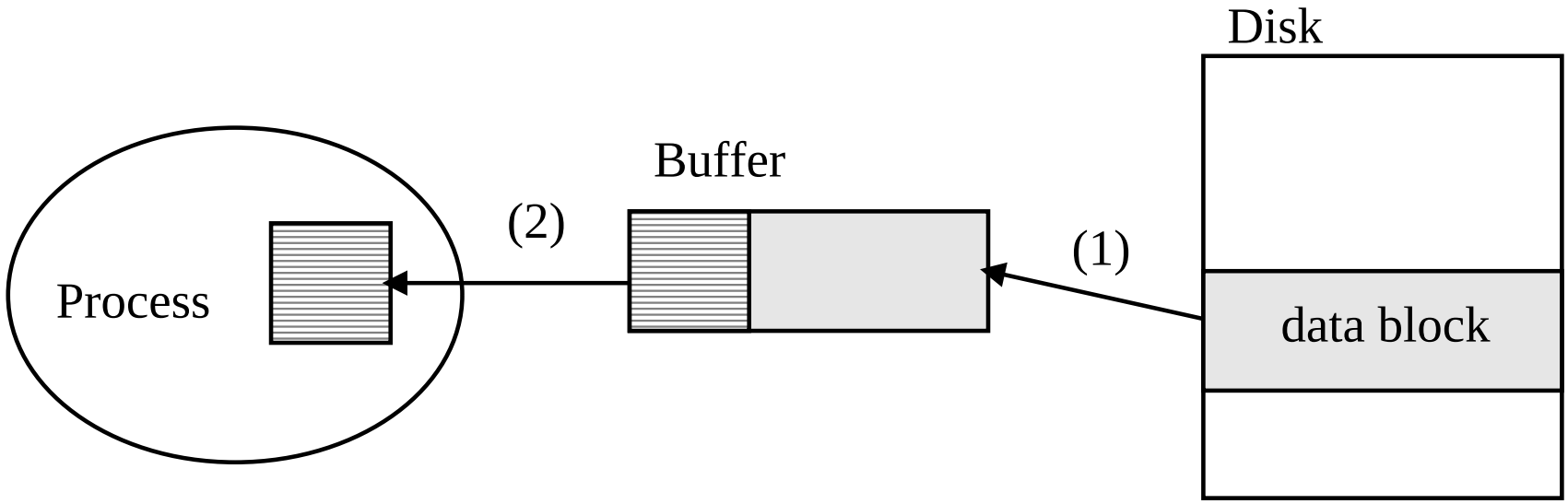
# Datové struktury a algoritmy pro vstup/výstup

# Použití vyrovnávacích pamětí

- I/O buffering:
  - buffer = vyrovnávací paměť (VP)
  - Cílem je minimalizace počtu pomalých operací s periferiemi (typicky s disky).
  - Dílčí vyrovnávací paměti mívají velikost alokačního bloku (příp. jejich skupiny) a jsou sdruženy do kolekce (tzv. buffer pool) pevné či proměnné velikosti umožňující snadné vyhledávání.
- Možná implementace:



# Čtení

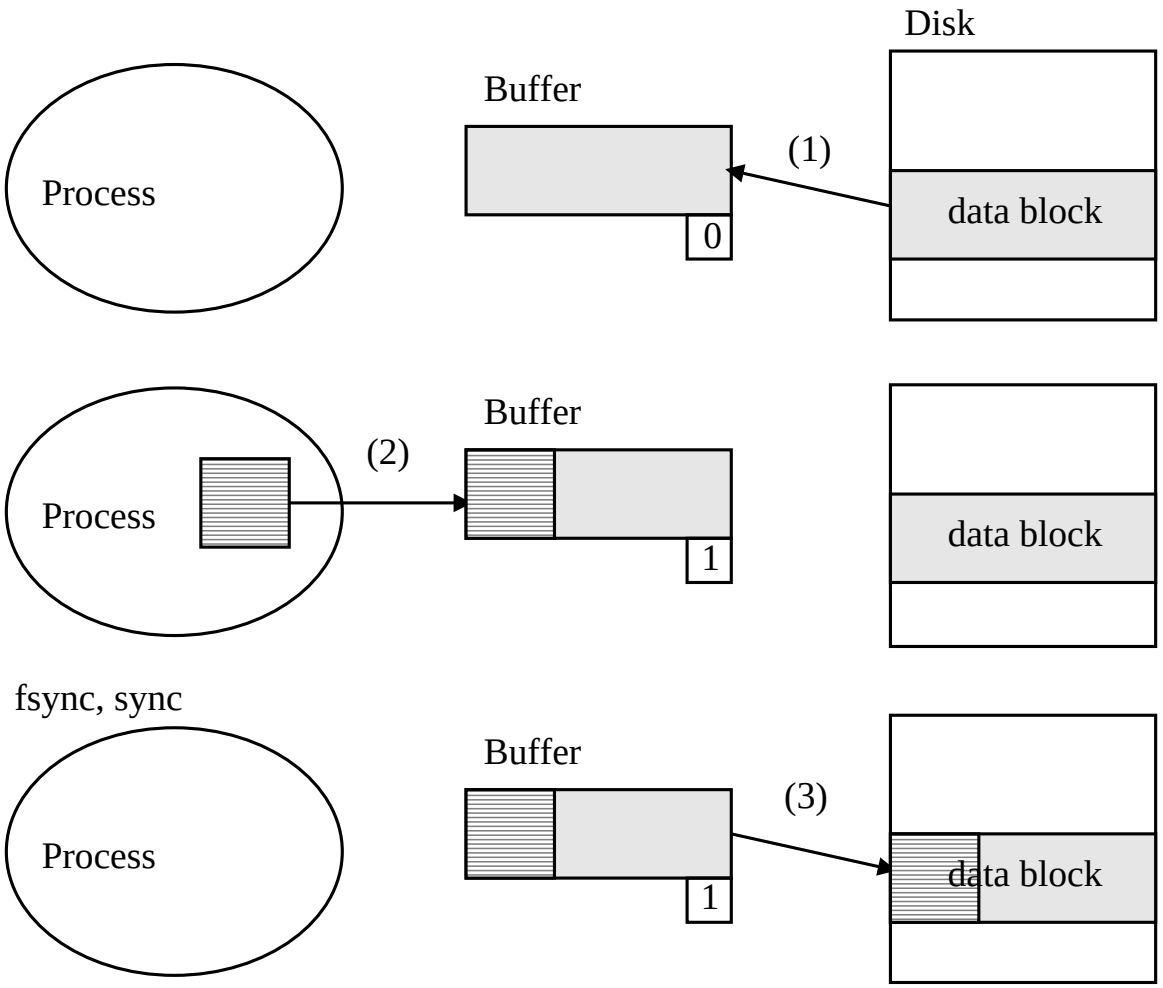



- Postup při prvním čtení (`read`):
  1. přidělení VP a načtení bloku,
  2. kopie požadovaných dat do adresového prostoru procesu (RAM→RAM).

- Při dalším čtení už pouze (2).

- Čtení, které překročí hranice bloku provede opět (1) a (2).

# Zápis




- Postup při zápisu (`write`):
  1. přidělení VP a čtení bloku do VP (pokud se nezapisuje na konec souboru),
  2. zápis dat do VP (RAM→RAM), nastaví se příznak modifikace (dirty bit),
  3. zpožděný zápis na disk, nuluje příznak.

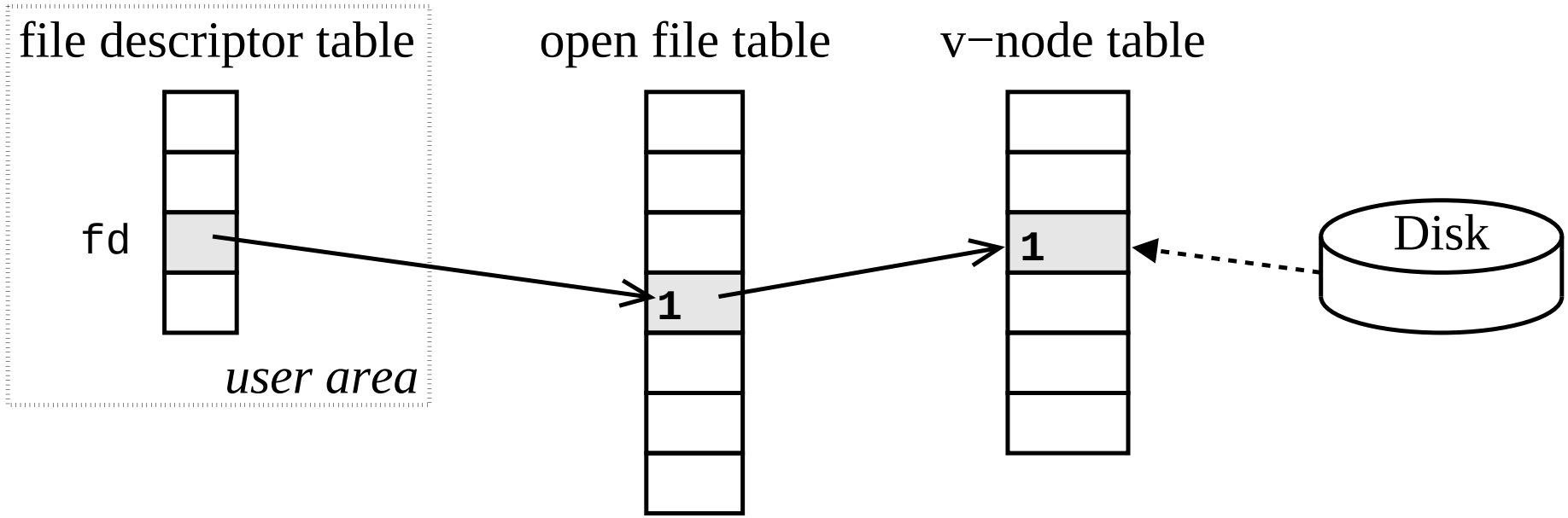
# Otevření souboru pro čtení

```
fd = open("/dir/file", O_RDONLY);
```

❖ V případě, že soubor ještě nebyl otevřen:

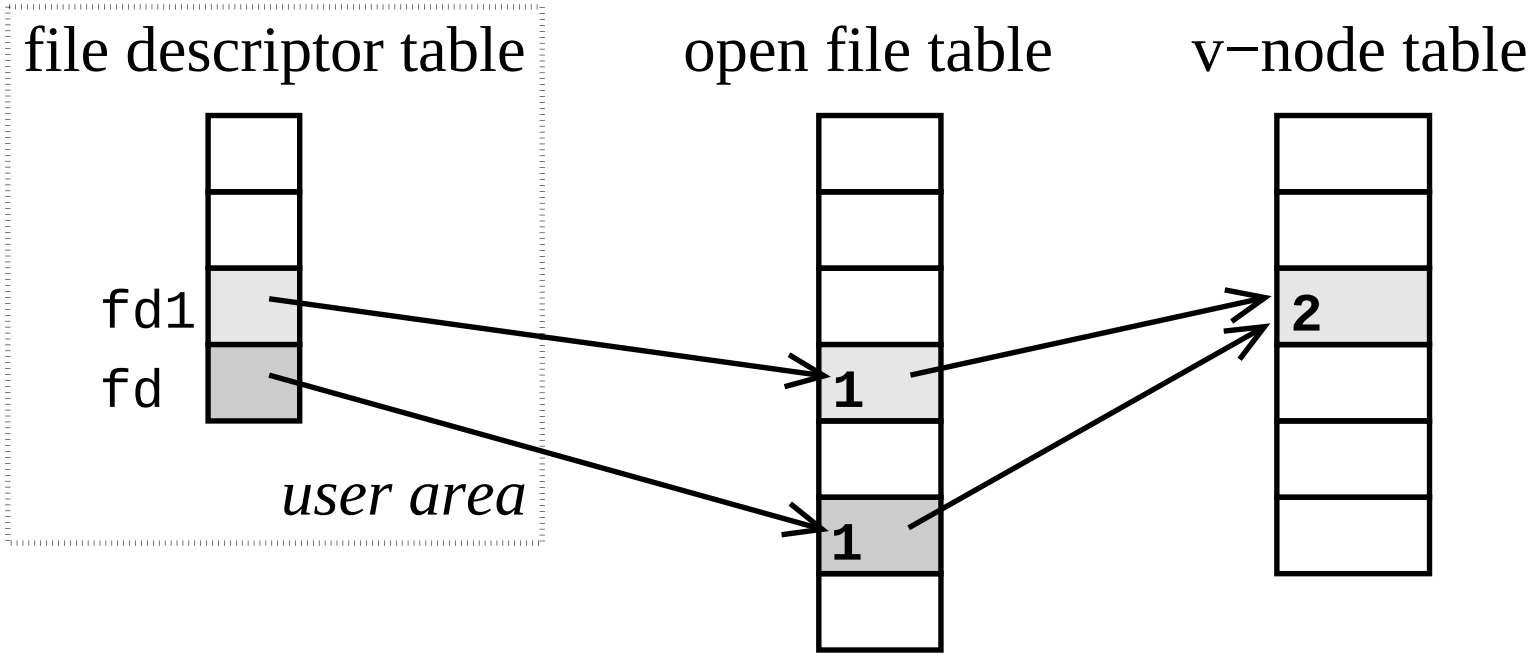
1. Vyhodnotí cestu a nalezne číslo i-uzlu (postupně načítá i-uzly adresářů a obsah těchto adresářů, aby se dostal k číslům i-uzlů pod-adresářů či hledaného souboru – čísla i-uzlů pro některá jména mohou samozřejmě být ve speciálních vyrovnávacích pamětech, tzv. d-entry cache).
2. V systémové tabulce aktivních i-uzlů vyhradí novou položku a načte do ní i-uzel. Vzniká rozšířená paměťová kopie i-uzlu: v-uzel.
3. V systémové tabulce otevřených souborů vyhradí novou položku a naplní ji:
   - odkazem na položku tabulky v-uzlů,
   - režimem otevření,
   - pozicí v souboru (0),
   - čítačem počtu referencí na tuto položku (1).
4. V poli deskriptorů souborů v uživatelské oblasti procesu (nebo v záznamu o procesu v jádře) vyhradí novou položku (první volná) a naplní ji odkazem na položku v tabulce otevřených souborů.
5. Vrátí index položky v poli deskriptorů (nebo -1 při chybě).

❖ Ilustrace prvního otevření souboru (čísla v obrázku udávají čítače počtu referencí na danou položku):



❖ Otevření již jednou otevřeného souboru:

1. Vyhodnotí cestu a získá číslo i-uzlu.
2. V systémové tabulce v-uzlů nalezne již načtený i-uzel: tabulka v-uzlů musí být implementována za tím účelem jako vyhledávací tabulka.
3. Zvýší počitadlo odkazů na v-uzel o 1.
4. A další beze změny.



❖ Při otevírání se provádí kontrola přístupových práv.

❖ Soubor je možno otevřít v režimu:

- čtení,
- zápis,
- čtení i zápis

modifikovaných volbou dalších parametrů otevření:

- přidávání,
- zkrácení na nulu,
- synchronní zápis,
- ...

❖ Při chybě vrací `-1` a nastaví chybový kód do knihovní proměnné `errno` (pro standardní chybové kódy viz `man errno`; lze užít např. spolu s knihovní funkcí `perror`). Podobně je tomu i u ostatních systémových volání v UNIXu.

# Čtení a zápis z/do souboru

- Čtení ze souboru – `n = read(fd, buf, 3000);`
  1. Kontrola platnosti `fd`.
  2. V případě, že jde o první přístup k příslušné části souboru, dojde k alokaci VP a načtení bloků souboru z disku do VP. Jinak dochází k alokaci VP a diskové operaci jen tehdy, je-li je to nutné (viz slajd k vyrovnávacím pamětem).
  3. Kopie požadovaných dat z VP (RAM, jádro) do pole `buf` (RAM, adresový prostor procesu).
  4. Funkce vrací počet opravdu přečtených bajtů nebo -1 při chybě (při současném nastevní `errno`).

- Zápis do souboru – `n = write(fd, buf, 3000);`
  - Funguje podobně jako `read` (viz slajd k vyrovnávacím pamětem).
  - Před vlastním zápisem kontroluje dostupnost diskového prostoru a prostor rezervuje.
  - Funkce vrací počet opravdu zapsaných bajtů nebo -1 při chybě.

# Přímý přístup k souboru

```
n = lseek(fd, offset, whence);
```

❖ Postup při žádosti o přímý přístup k souboru:

1. Kontrola platnosti `fd`.
2. Nastaví pozici `offset` bajtů od
   - začátku souboru pro `whence=SEEK_SET`,
   - aktuální pozice pro `whence=SEEK_CUR`,
   - konce souboru pro `whence=SEEK_END`.
3. Funkce vrací výslednou pozici od začátku souboru nebo -1 při chybě.

❖ Poznámka: Hodnota parametru `offset` může být záporná, nelze však nastavit pozici před začátek souboru.

- Sparse files – “řídké soubory”:
  - Vznikají nastavením pozice za konec souboru a zápisem.
  - Bloky, do kterých se nezapisovalo nejsou alokovány a nezabírají diskový prostor. Při čtení se považují za vynulované.

# Zavření souboru

```
x = close(fd);
```

- Postup při uzavírání souboru:
  1. Kontrola platnosti `fd`.
  2. Uvolní se odpovídající položka v tabulce deskriptorů, sníží se počitadlo odkazů v odpovídající položce tabulky otevřených souborů.
  3. Pokud je počitadlo odkazů nulové, uvolní se odpovídající položka v tabulce otevřených souborů a sníží se počitadlo odkazů ve v-uzlu.
  4. Pokud je počitadlo odkazů nulové, i-uzel se z v-uzlu okopíruje do VP a uvolní.
  5. Funkce vrací nulu nebo -1 při chybě.

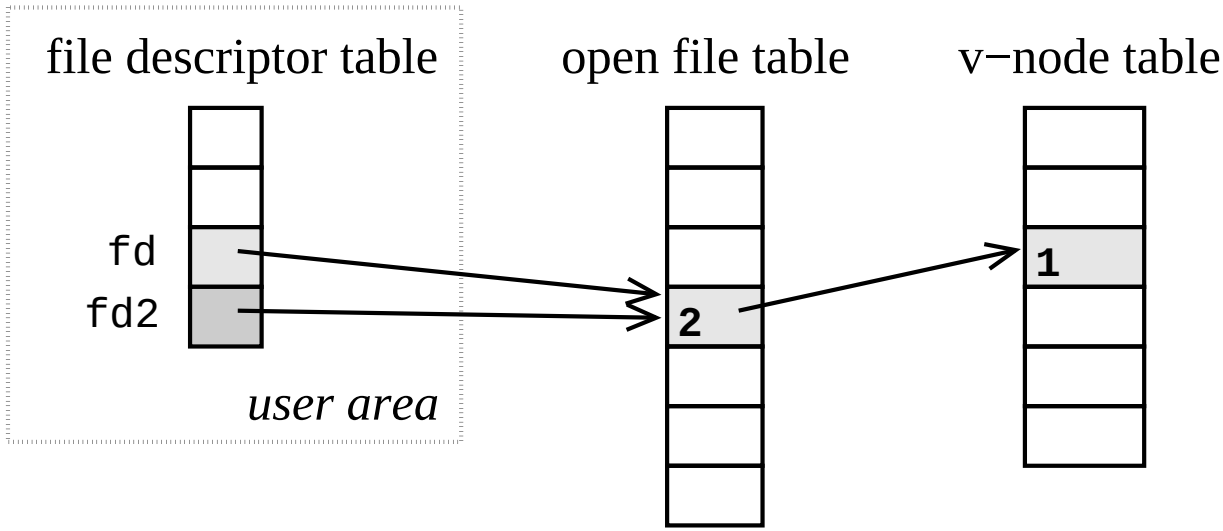
- Pokud se ukončuje proces, automaticky se uzavírají všechny jeho deskriptory.

- Uzavření souboru nezpůsobí uložení obsahu jeho VP na disk!

# Duplikace deskriptoru souboru

```
fd2 = dup(fd);
fd2 = dup2(fd,newfd);
```

- Postup při duplikaci deskriptoru:
  1. Kontrola platnosti `fd`.
  2. Kopíruje danou položku v tabulce deskriptorů do první volné položky (`dup`) nebo do zadané položky (`dup2`). Je-li deskriptor `newfd` otevřen, `dup2` ho automaticky uzavře.
  3. Zvýší počitadlo odkazů v odpovídající položce tabulky otevřených souborů.
  4. Funkce vrací index nové položky nebo -1 při chybě.




- Poznámka: Použití pro přesměrování stdin/stdout.

# Rušení souboru

```
x = unlink("/dir/file");
```

- Postup při rušení souboru:
  1. Vyhodnocení cesty, kontrola platnosti jména souboru a přístupových práv.
  2. Odstraní pevný odkaz (hard link) mezi jménem souboru a i-uzlem. Vyžaduje právo zápisu do adresáře.
  3. Zmenší počitadlo jmen v i-uzlu.
  4. Pokud počet jmen klesne na nulu a i-uzel nikdo nepoužívá, je i-uzel uvolněn včetně všech používaných bloků souboru. Je-li i-uzel používán, bude uvolnění odloženo až do okamžiku zavření souboru (počitadlo otevření souboru klesne na 0).
  5. Funkce vrací nulu nebo -1 při chybě.

- Poznámky:
  - Proces může provést `unlink` na otevřený soubor a dále s ním pracovat až do jeho uzavření.
  - Je možné zrušit spustitelný soubor, i když běží jím řízené procesy (výhoda při instalaci nových verzí programů).
  - Bezpečnější mazání: `shred`.

# Další operace se soubory

- Vytvoření souboru: `creat`, `open`
- Přejmenování: `rename`
- Zkrácení: `truncate`, `ftruncate`
- Zamykání záznamů: `fcntl` nebo `lockf`
- Změna atributů: `chmod`, `chown`, `utime`
- Získání atributů: `stat`
- Zápis VP na disk: `sync`, `fsync`

# Adresářové soubory

- Adresáře se liší od běžných souborů:
  - vytváří se voláním `mkdir` (vytvoří položky `.` a `..`),
  - mohou být otevřeny voláním `opendir`,
  - mohou být čteny voláním `readdir`,
  - mohou být uzavřeny voláním `closedir`,
  - modifikaci je možné provést pouze vytvářením a rušením souborů v adresáři (`creat`, `link`, `unlink`, ...).

- Poznámka: Adresáře nelze číst/zapisovat po bajtech!

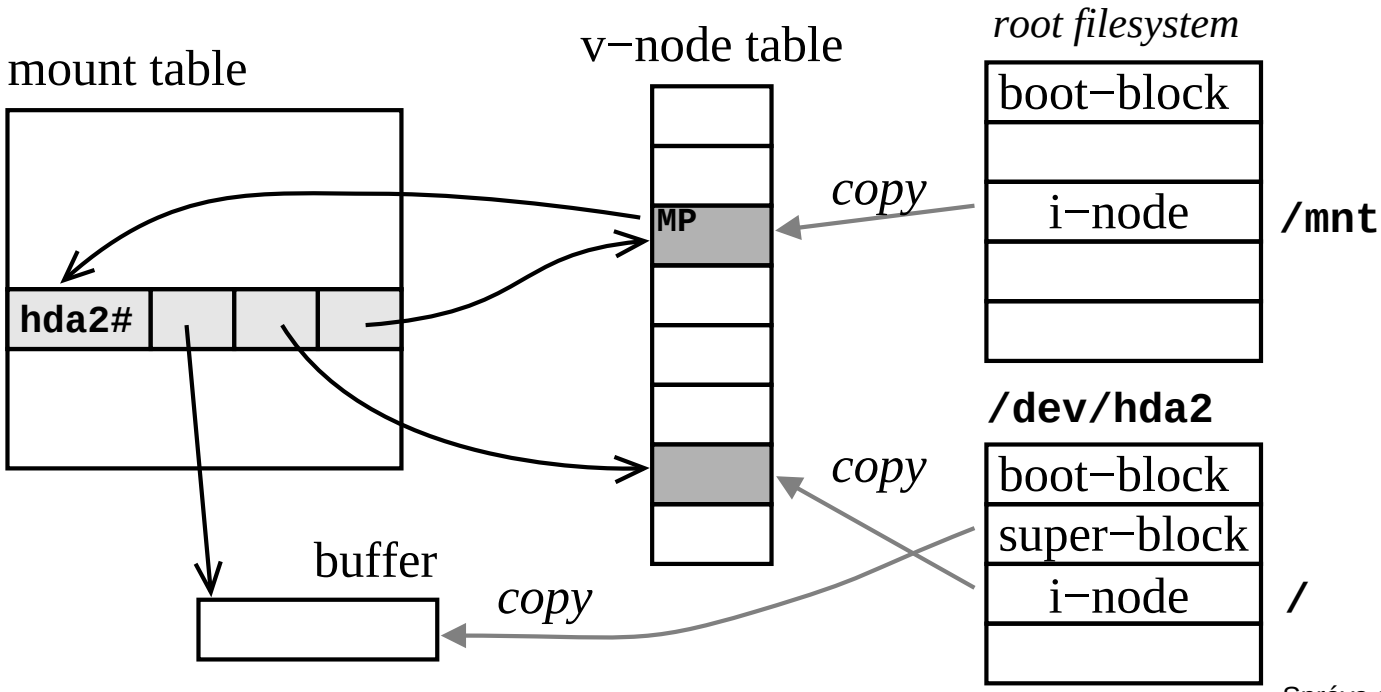
- Příklad obsahu adresáře:

| | |
|---|---|
| 32577 | . |
| 2 | .. |
| 2361782 | Archiv |
| 1058839 | Mail |
| 1661377 | tmp |

# Připojení systému souborů

- V počítači může být více fyzických disků.
- Disk můžeme rozdělit na části (*partition*, *volume*).
- Každá část je identifikována speciálním souborem (`/dev/hda1`, `/dev/hda2`, ...).
- Logický disk může obsahovat souborový systém.
  - `mount` připojí systém souborů do hlavního adresářového stromu,
  - `umount` odpojí systém souborů.

❖ Příklad: `x = mount("/dev/hda2","/mnt",parameters);`



# *Blokové a znakové speciální soubory*

❖ Jádro mapuje běžné souborové operace (`open`, `read`, ...) nad blokovými a znakovými speciálními soubory na odpovídající podprogramy tyto operace implementující prostřednictvím dvou tabulek:

- tabulky znakových zařízení a
- tabulky blokových zařízení.

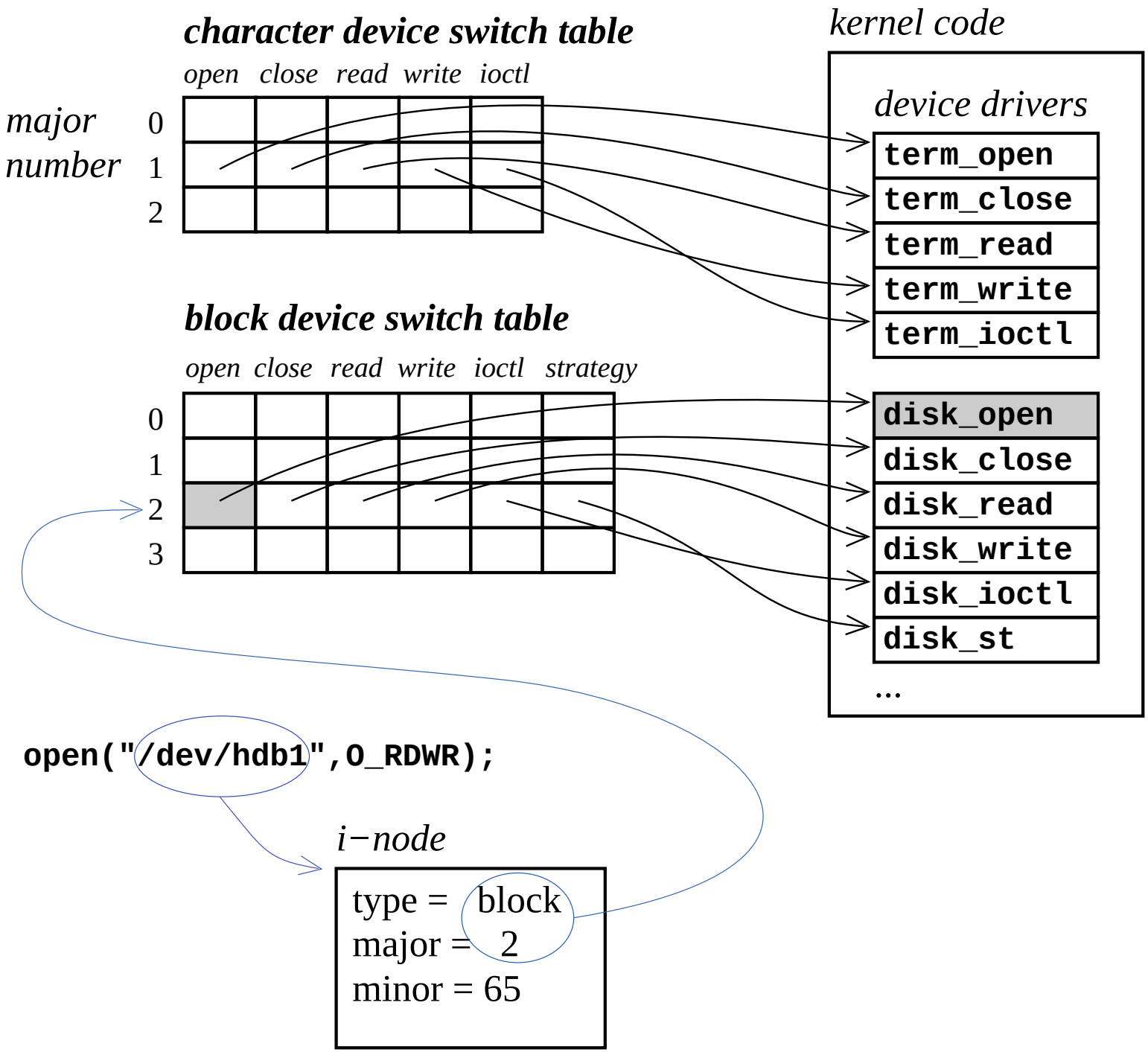
❖ Zmíněné tabulky obsahují ukazatele na funkce implementující příslušné operace v ovladačích příslušných zařízení.

❖ Ovladač (*device driver*) je sada podprogramů pro řízení určitého *typu* zařízení.

❖ Rozhraní zařízení – speciální soubory (např. `/dev/tty`): mají v i-uzlu mj. typ souboru a dvě čísla:

| *číslo* | *význam* |
|---|---|
| hlavní číslo (major number) | typ zařízení |
| vedlejší číslo (minor number) | instance zařízení |

- Typ souboru (blokový nebo znakový) určuje tabulku.
- Hlavní číslo se použije jako index do tabulky zařízení.
- Vedlejší číslo se předá jako parametr funkce ovladače (identifikace instance zařízení).


character device switch table
open close read write ioctl
major number
0
1
2
kernel code
device drivers
term_open
term_close
term_read
term_write
term_ioctl
block device switch table
open close read write ioctl strategy
0
1
2
3
disk_open
disk_close
disk_read
disk_write
disk_ioctl
disk_st
...
open("/dev/hdb1",O_RDWR);
i−node
type = block
major = 2
minor = 65

# Terminály

❖ Terminály – fyzická nebo logická zařízení umožňující (primárně) textový vstup/výstup systému: vstup/výstup po řádcích, editace na vstupním řádku, speciální znaky (Ctrl-C, Ctrl-D, ...), ...

❖ Rozhraní:

- `/dev/tty` – řídící terminál aktuálního procesu, odkaz na příslušné terminálové zařízení,
- `/dev/ttyS1`, ... – terminál na sériové lince,
- `/dev/tty1`, ... – virtuální terminály (konzole),
- pseudoterminály (např. `/dev/ptmx` a `/dev/pts/1`, ...) – tvořeny párem master/slave emulujícím komunikaci přes sériovou linku (např. použito u X-terminálu či ssh).

❖ Různé režimy zpracování znaků (line discipline):

| režim | význam |
|---|---|
| raw | bez zpracování |
| cbreak | zpracovává jen některé znaky (Ctrl-C, ...) |
| cooked | zpracovává vše |

❖ Nastavení režimu zpracování znaků (nastavení ovladače terminálu): program `stty`.

❖ Nastavení režimu terminálu (tedy fyzického zařízení nebo emulujícího programu):

- proměnná `TERM` – typ aktuálního terminálu,
- databáze popisu terminálů (možnosti nastavení terminálu): `terminfo` či `termcap`.
- nastavení příkazy: `tset`, `tput`, `reset`, ...

❖ Knihovna `curses` – standardní knihovna pro řízení terminálu a tvorbu aplikací s terminálovým uživatelským rozhraním (včetně menu, textových oken apod.).

# Roury

- Roury (pipes) – jeden z typů speciálních souborů. Rozlišujeme:
  - roury nepojmenované
    - nemají adresářovou položku,
    - `pipe` vrací dva deskriptory (čtecí a zápisový), jsou přístupné pouze příbuzným procesům (tvůrce fronty je přímý či nepřímý předek komunikujících procesů nebo jeden z nich),
    - vytváří se v kolonách (např. `p1 | p2 | p3`),
  - Roury pojmenované – vytvoření `mknod` či `mkfifo`.

- Roury representují jeden z mechanismů meziprocesové komunikace.

- Implementace: kruhový buffer s omezenou kapacitou.

- Procesy komunikující přes rouru (producent a konzument) jsou synchronizovány.

# Sockets

❖ Umožňují síťovou i lokální komunikaci.

❖ Lokální komunikace může probíhat přes sockety pojmenované a zpřístupněné v souborovém systému).

❖ API pro práci se sockets:

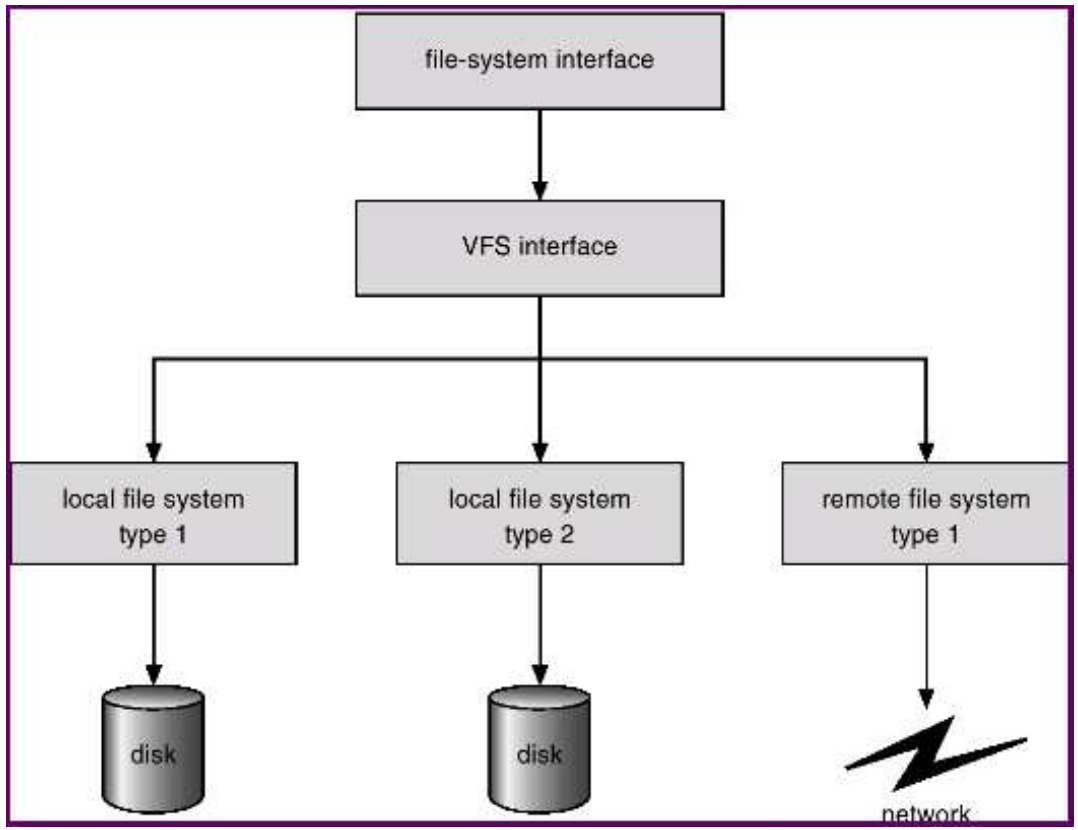
- vytvoření (`socket`),
- čekání na připojení (`bind`, `listen`, `accept`),
- připojení ke vzdálenému počítači (`connect`),
- příjem a vysílání (`recv`, `send`, `read`, `write`),
- uzavření (`close`).

❖ Sockets podporují blokující/neblokující I/O. U neblokujícího režimu je možno použít `select` pro současnou obsluhu více sockets jedním procesem (vláknem).

# VFS

❖ VFS (Virtual File System) vytváří jednotné rozhraní pro práci s různými souborovými systémy, odděluje vyšší vrstvy OS od konkrétní implementace jednotlivých souborových operací na jednotlivých souborových systémech.

❖ Pro popis souborů používá rozšířené i-uzly (tzv. v-uzly), které mohou obsahovat např. počet odkazů na v-uzel z tabulky otevřených souborů, ukazatele na funkce implementující operace nad i-uzlem v patřičném souborovém systému apod.

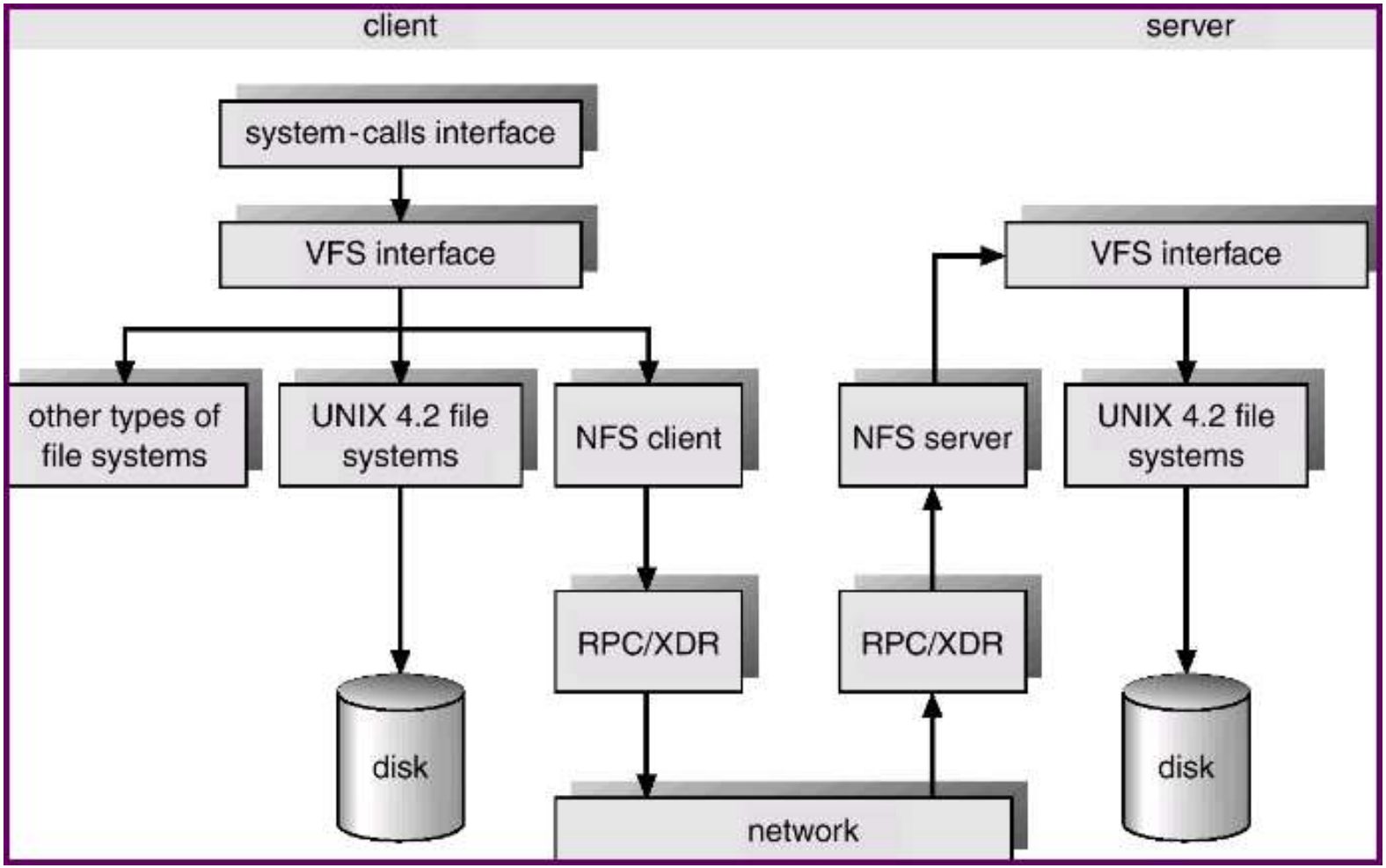


❖ VFS v Linuxu:

- Používá objektově-orientovaný přístup se 3 základními typy objektů:
  - souborové systémy,
  - v-uzly pro soubory s nimiž se pracuje,
  - otevřené soubory,
- Každý z těchto objektů obsahuje odkazy na funkce implementující příslušné operace.
- Navíc má VFS tabulku s odkazy na funkce, které umožňují provést pro jednotlivé známé souborové systémy operaci mount.
- Pracuje s tzv. položkami adresářů d-entries – mapují jméno uložené v adresáři na v-uzel, ukládá se v d-cache pro urychlení překladu jmen na v-uzly.

# NFS

❖ NFS (Network File System) – transparentně zpřístupňuje soubory uložené na vzdálených systémech.



❖ Umožňuje kaskádování: lokální připojení vzdáleného adresářového systému do jiného vzdáleného adresářového systému.

❖ Autentizace často prostřednictvím uid a gid – pozor na bezpečnost!

❖ NFS verze 3:

- bezestavové – nepoužívá operace otevírání a uzavírání souborů, každá operace musí nést veškeré potřebné argumenty,
- na straně klienta se neužívá cache,
- nemá podporu zamykání.

❖ NFS verze 4:

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

# Spooling

- Spooling = simultaneous peripheral operations on-line.

- spool = vyrovnávací paměť (typicky soubor) pro zařízení (nejčastěji tiskárny), které neumožňují prokládané zpracování dat různých procesů.

- Výstup je proveden do souboru, požadavek na jeho fyzické zpracování se zařadí do fronty, uživatelský proces může pokračovat a zpracování dat se provede, až ně přijde řada.

- V Unixu/Linuxu: `/var/spool`

# Operační systémy

IOS 2015/2016

**Tomáš Vojnar**

vojnar@fit.vutbr.cz

**Vysoké učení technické v Brně**
**Fakulta informačních technologií**
**Božetěchova 2, 612 66 Brno**

# Správa procesů

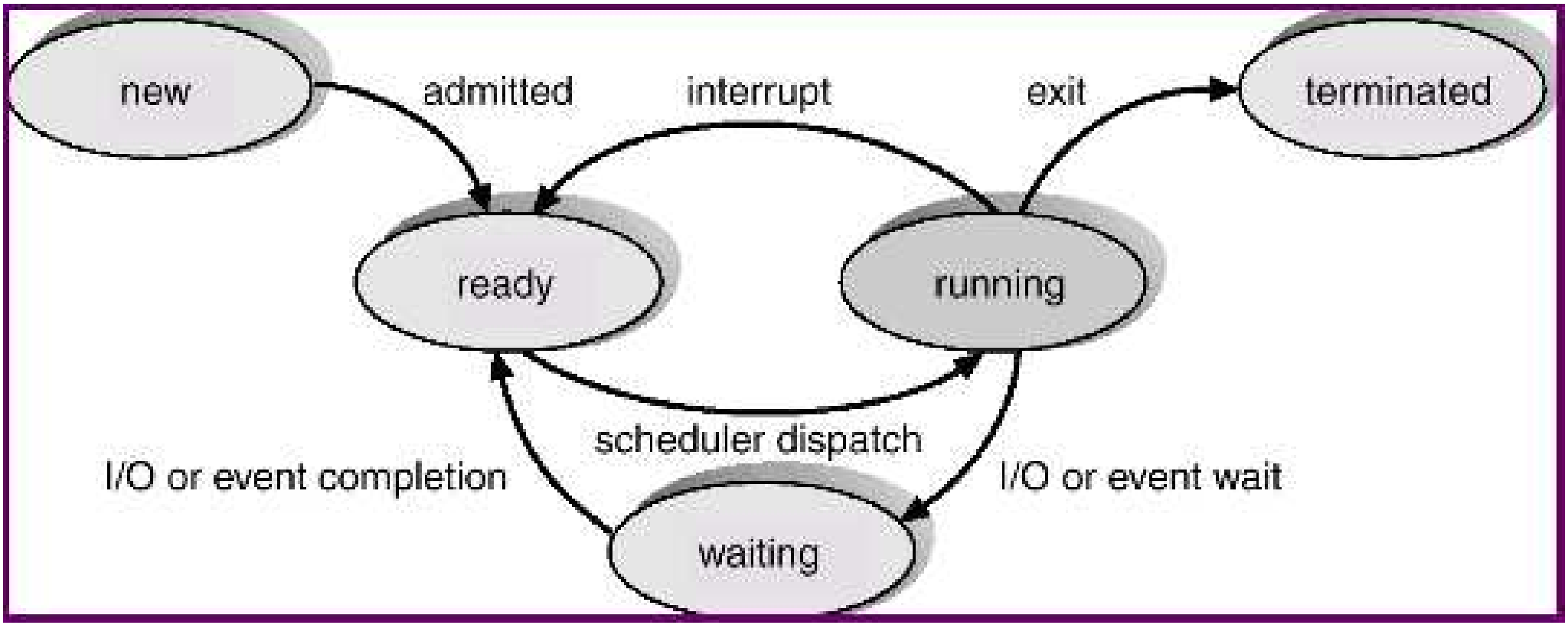
- Správa procesů (process management) zahrnuje:
  - přepínámí kontextu – dispatcher,
  - plánovač (scheduler) – přiděluje CPU procesům,
  - správu paměti (memory management) – přiděluje paměť,
  - podporu meziprocesové komunikace (IPC) – signály, RPC, ...

# Proces

- Proces = běžící program.
- Proces je v OS definován:
  - identifikátorem (PID),
  - stavem jeho plánování,
  - programem, kterým je řízen,
  - obsahem registrů (včetně EIP a ESP apod.),
  - daty a zásobníkem,
  - využitím dalších zdrojů OS a vazbou na další objekty OS (otevřené soubory, signály, PPID, UID, GID, ...).

# Stavy plánování a jejich změny

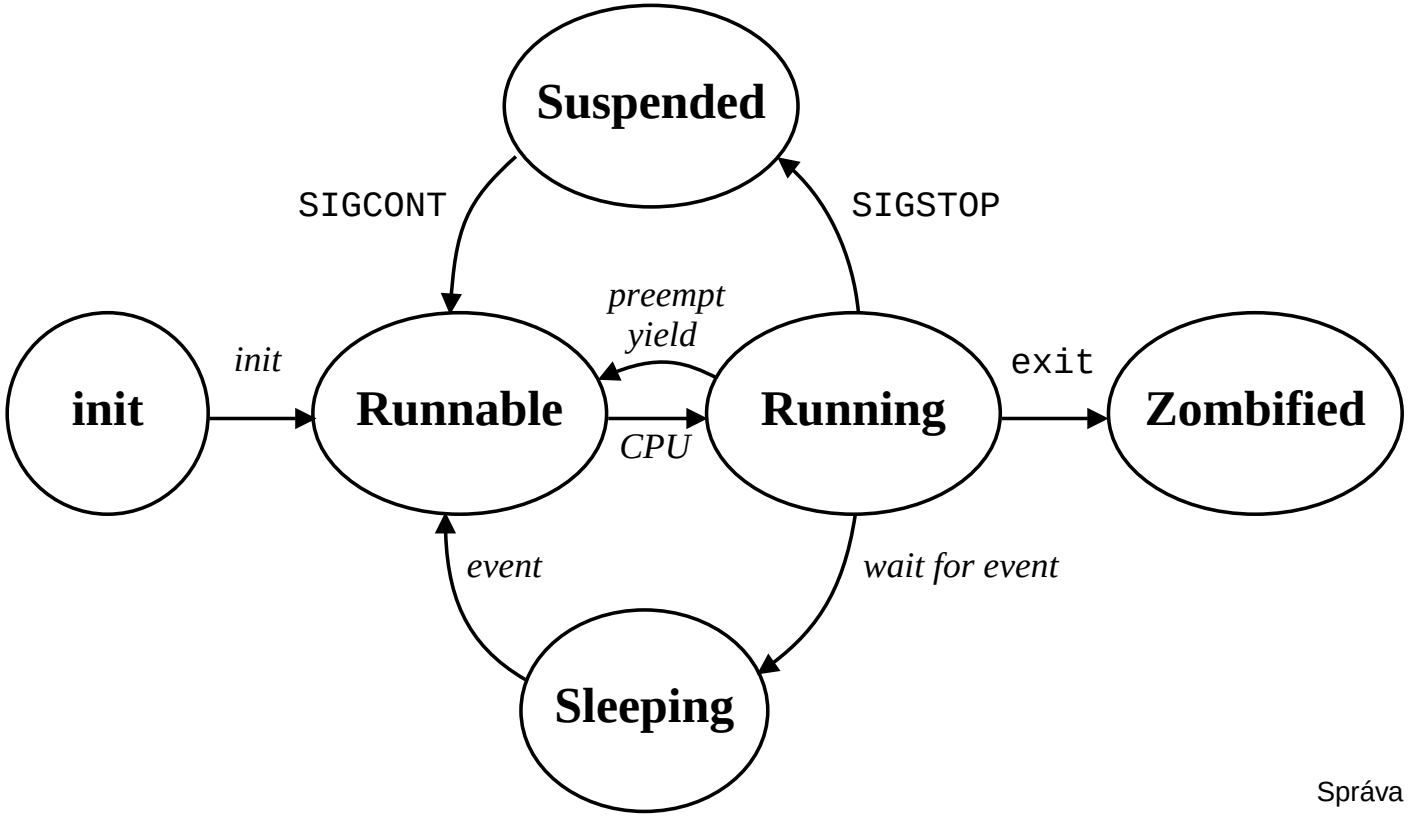
- Běžně se rozlišují následující stavy procesů:



❖ Stavy plánování procesu v Unixu:

| stav | význam |
|---|---|
| Vytvořený | ještě neinicializovaný |
| Připravený | mohl by běžet, ale nemá CPU |
| Běžící | používá CPU |
| Mátoha | po `exit`, rodič ještě nepřevzal exit-code |
| Čekající | čeká na událost (např. dokončení `read`) |
| Odložený | "zmrazený"signálem `SIGSTOP` |

❖ Přechodový diagram stavů plánování procesu v Unixu:



❖ V OS bývá proces reprezentován strukturou označovanou jako PCB (Process Control Block) nebo též task control block či task struct apod.

❖ PCB zahrnuje:

- identifikátory spojené s procesem,
- stav plánování procesu,
- obsah registrů (včetně EIP a ESP apod.),
- plánovací informace (priorita, ukazatele na plánovací fronty, ...),
- informace spojené se správou paměti (tabulky stránek, ...),
- informace spojené s účtováním (spotřeba procesoru, ...),
- využití I/O zdrojů (otevřené soubory, používaná zařízení., ...).

❖ PCB může být někdy rozdělen do několika dílčích struktur.

# Části procesu v paměti v Unixu

- Uživatelský adresový prostor (user address space) přístupný procesu:
  - kód (code area/text segment),
  - data (inicializovaná/neinicializovaná data, heap),
  - zásobník,
  - soukromá data sdílených knihoven, sdílené knihovny, sdílená paměť, individuálně alokovaná paměť.
- Uživatelská oblast (user area) – ne vždy použita:
  - Uložena zvlášť pro každý proces spolu s daty, kódem a zásobníkem v user address space příslušného procesu (s nímž může být odložena na disk).
  - Je ale přístupná pouze jádru.
  - Obsahuje část PCB, která je používána zejména za běhu procesu:
    - PID, PPID, UID, EID, GID, EGID,
    - obsah registrů,
    - deskriptory souborů,
    - obslužné funkce signálů,
    - účtování (spotřebovaný čas CPU, ...),
    - pracovní a kořenový adresář, ...

❖ Záznam v tabulce procesů (process table):

- Uložen trvale v jádru.
- Obsahuje zejména informace o procesu, které jsou důležité, i když proces neběží:
  - PID, PPID, UID, EID, GID, EGID,
  - stav plánování,
  - událost, na kterou se čeká,
  - plánovací informace (priorita, spotřeba času, ...),
  - čekající signály,
  - odkaz na tabulku paměťových regionů procesu,
  - ...

❖ Tabulka paměťových regionů procesu (per-process region table) – popis paměťových regionů procesu (spojitá oblast virtuální paměti použitá pro data, kód, zásobník, sdílenou paměť) + příslušné položky tabulky regionů, tabulka stránek.

❖ Zásobník jádra využívaný za běhu služeb jádra pro daný proces.

# *Kontext procesu*

- Někdy se též používá pojem kontext procesu = stav procesu.

- Rozlišujeme:
  - uživatelský kontext (user-level context): kód, data, zásobník, sdílená data,
  - registrový kontext,
  - systémový kontext (system-level context): uživatelská oblast, položka tabulky procesů, tabulka paměťových regionů procesu, ...

# Systémová volání nad procesy v Unixu

- Systémová volání spojená s procesy v Unixu:
  - `fork`, `exec`, `exit`, `wait`, `waitpid`,
  - `kill`, `signal`,
  - `getpid`, `getppid`,
  - ...
- Identifikátory spojené s procesy v UNIXu:
  - identifikace procesu PID,
  - identifikace předka PPID,
  - reálný (skutečný) uživatel, skupina uživatelů UID, GID,
  - efektivní uživatel, skupina uživatelů EUID, EGID,
  - uložená EUID, uložená EGID – umožňuje dočasně snížit efektivní práva a pak se k nim vrátit (při zpětném nastavení se kontroluje, zda se proces vrací k reálnému ID nebo k uloženému EUID),
  - v Linuxu navíc FSUID a FSGID (pro přístup k souborům se zvýšenými privilegii),
  - skupina procesů a sezení, do kterých proces patří (PGID, SID).

# Vytváření procesů

❖ Vytváření procesů v UNIXu – služba `fork`: duplikuje proces, vytvoří takřka identického potomka:

- dědí řídící kód, data, zásobník, sdílenou paměť, otevřené soubory, obsluhu signálů, většinu synchronizačních prostředků, ...;
- liší se v návratovém kódu `fork`, identifikátorech, údajích spojených s plánováním a účtováním (spotřeba času, ...), nedědí čející signály, souborové zámky a některé další specilizované zdroje a nastavení.

```
pid=fork();
if (pid==0) {
    // kód pro proces potomka
    // exec(....), exit(exitcode)
} else if (pid==-1) {
    // kód pro rodiče, nastala chyba při fork()
    // errno obsahuje bližší informace
} else {
    // kód pro rodiče, pid = PID potomka
    // pid2 = wait(&stav);
}
```

❖ Vzniká vztah rodič–potomek (parent–child) a hierarchie procesů.

# Hierarchie procesů v Unixu

- Předkem všech uživatelských procesů je init s PID=1.

- Existují procesy jádra (kernel processes/threads), jejichž předkem init není:
  - Jejich kód je součástí jádra.
  - Někdy se vyskytuje proces s PID=0, kterým bývá swapper nebo plánovač (na Linuxu pouze idle, nevypisuje se).
  - Na Linuxu existuje process jádra kthreadd, který spouští ostatní procesy jádra a je jejich předkem.
  - Vztahy mezi procesy jádra nejsou příliš významné a mohou se lišit.

- Pokud procesu skončí předek, jeho předkem se automaticky stane init, který později převezme jeho návratový kód (proces nemůže definitivně skončit a jako zombie čeká, dokud neodevzdá návratový kód).

- Výpis stromu procesů: např. `pstree`.

# Změna programu – `exec`

- Skupina funkcí:
  - `execve` – základní volání,
  - `execl`, `execlp`, `execle`, `execv`, `execvp`.
  - `execl("/bin/ls", "ls", "-l", NULL);`
    – argumenty odpovídají `$0`, `$1`, ...

- Nevrací se, pokud nedojde k chybě!

- Procesu zůstává řada jeho zdrojů a vazeb v OS (identifikátory, otevřené soubory, ...), zanikají vazby a zdroje vázané na původní řídící kód (obslužné funkce signálů, sdílená paměť, paměťově mapované soubory, semafory).

- Windows: `CreateProcess(...)` – zahrnuje funkčnost `fork` i `exec`.

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

- Systémové volání `wait` umožňuje pasivní čekání na potomka.
  - Vrací číslo ukončeného procesu (nebo -1: příchod signálu, neexistence potomků).
  - Přes argument zpřístupňuje návratový kód potomka.
  - Pokud nějaký potomek je již ukončen a čeká na předání návratového kódu, končí okamžitě.

- Obecnější je systémové volání `waitpid`:
  - Umožňuje čekat na určitého potomka či potomka z určité skupiny.
  - Umožňuje čekat i na pozastavení či probuzení z pozastavení příjmem signálu.

# Start systému

- Typická posloupnost akcí při startu systému:
  1. BIOS.
  2. Načtení a spuštění zavaděče, obvykle v několika fázích (využívá mbr a případně další části disku).
  3. Načtení inicializačních funkcí jádra a samotného jádra, spuštění inicializační funkcí.
  4. Inicializační funkce jádra mj. vytvoří proces jádra 0, ten vytvoří případné další procesy jádra a proces *init*.
  5. *init* načítá inicializační konfigurace a spouští další démony a procesy.
     - V určitém okamžiku spustí *gdm*/*kdm* pro přihlášení v grafickém režimu (z něj se pak spouští další procesy pro práci pod X Window).
     - Na konzolích spustí *getty*, který umožní zadat přihlašovací jméno a změní se na *login*. Ten načte heslo a změní se na shell, ze kterého se spouští další procesy. Po ukončení se na terminálu spustí opět *getty*.
     - Proces *init* i nadále běží, přebírá návratové kódy procesů, jejichž rodič již skončil a řeší případnou reinicializaci systému či jeho částí při výskytu různých nakonfigurovaných událostí nebo na přání uživatele.

# Úrovně běhu

- V UNIXu System V byl zaveden systém úrovní běhu – SYSV init run-levels: 0-6, s/S (0=halt, 1=single user, s/S=alternativní přechod do single user, 6=reboot).
- Změna úrovně běhu: `telinit N`.
- Konfigurace:
  - Adresáře */etc/rcX.d* obsahují odkazy na skripty spouštěné při vstupu do určité úrovně. Spouští se v pořadí daném jejich jmény, skripty se jménem začínajícím *K* s argumentem *stop*, skripty se jménem začínajícím *S* s parametrem *start*.
  - Vlastní implementace v adresáři */etc/init.d*, lze spouštět i ručně (např. také s argumentem *reload* či *restart* – reinicializace různých služeb, např. sítě).
  - Soubor */etc/inittab* obsahuje hlavní konfigurační úroveň: např. implicitní úroveň běhu, odkaz na skript implementující výše uvedené chování, popis akcí při mimořádných situacích, inicializace konzolí apod.
- Existují různé nové implementace procesu *init*:
  - Upstart: důraz na řízení událostmi generovanými jím samým (např. při ukončení nějaké jím spuštěné úlohy) příp. generovanými jádrem nebo dalšími procesy.
  - Initng, systemd: spouští inicializační akce paralelně na základě jejich závislostí.

# Plánování procesů

❖ Plánovač rozhoduje, který proces poběží a případně, jak dlouho.

❖ Preemptivní plánování: ke změně běžícího procesu může dojít, aniž by tento jakkoliv přepnutí kontextu napomohl, a to na základě přerušení (typicky od časovače, ale může se jednat i o jiné přerušení).

❖ Nepreemptivní plánování: ke změně běžícího procesu může dojít pouze tehdy, pokud to běžící proces umožní předáním řízení jádru (žádá o službu – typicky I/O operaci, končí, nebo se sám vzdává procesoru – volání `yield`).

❖ Vlastní přepnutí kontextu řeší na základě rozhodnutí plánovače tzv. dispečer.

❖ Plánování může být též ovlivněno systémem swapování rozhodujícím o tom, kterým procesům je přidělena paměť, aby mohly běžet, případně systémem spouštění nových procesů, která může spuštění procesů odkládat na vhodný okamžik.

- V těchto případech někdy hovoříme o střednědobém a dlouhodobém plánování.

# Přepnutí procesu (kontextu)

❖ Dispečer odebere procesor procesu A a přidělí ho procesu B, což typicky zahrnuje:

- úschovu stavu (některých) registrů (včetně různých řídících registrů) v rámci procesu A do PCB,
- úpravu některých řídicích struktur v jádře,
- obnovu stavu (některých) registrů v rámci procesu B z PCB,
- předání řízení na adresu, kde bylo dříve přerušeno provádění procesu B.

❖ Neukládá se/neobnovuje se celý stav procesů: např. se uloží jen ukazatel na tabulku stránek, tabulka stránek a vlastní obsah paměti procesu může zůstat.

❖ Přepnutí trvá přesto typicky stovky až tisíce instrukcí: interval mezi přepínámím musí být tedy volen tak, aby režie přepnutí nepřevážila běžný běh procesů.

# Klasické plánovací algoritmy

❖ Klasické plánovací algoritmy uvedené níže se používají přímo, případně v různých modifikacích a/nebo kombinacích

❖ FCFS (First Come, First Served):

- Procesy čekají na přidělení procesoru ve FIFO frontě.
- Při vzniku procesu, jeho uvolnění z čekání (na I/O, synchronizaci apod.), nebo vzdá-li se proces procesoru, je tento proces zařazen na konec fronty.
- Procesor se přiděluje procesu na začátku fronty.
- Algoritmus je nepreemptivní a k přepnutí kontextu dojde pouze tehdy, pokud se běžící proces vzdá procesoru (voláním služeb např. pro I/O, konec, dobrovolné vzdání se procesoru – volání `yield`).

❖ Round-robin – preemptivní obdoba FCFS:

- Pracuje podobně jako FCFS, navíc má ale každý proces přiděleno časové kvantum, po jehož vypršení je mu odebrán procesor a proces je zařazen na konec fronty připravených procesů.

# Klasické plánovací algoritmy

- SJF (Shortest Job First):
  - Přiděluje procesor procesu, který požaduje nejkratší dobu pro své další provádění na procesoru bez vstup/výstupních operací (tzv. CPU burst).
  - Nepreemptivní algoritmus, který nepřerušuje proces před dokončením jeho aktuální výpočetní fáze.
  - Minimalizuje průměrnou dobu čekání, zvyšuje propustnost systému.
  - Nutno znát dopředu dobu běhu procesů na procesoru nebo mít možnost tuto rozumně odhadnout na základě předchozího chování.
  - Používá se ve specializovaných (např. dávkových) systémech.
  - Hrozí stárnutí (někdy též hladovění – starvation):
    - Stárnutím při přidělování zdrojů (procesor, zámek, ...) rozumíme obecně situaci, kdy některý proces, který o zdroj žádá, na něj čeká bez záruky, že jej někdy získá.
    - V případě SJF hrozí hladovění procesů čekajících na procesor a majících dlouhé výpočetní fáze, které mohou být neustále předbíhány kratšími výpočty.

- SRT (Shortest Remaining Time): obdoba SJF s preempcí při vzniku či uvolnění procesu.

# Klasické plánovací algoritmy

- Víceúrovňové plánování:
  - Procesy jsou rozděleny do různých skupin (typicky podle priority, ale lze i jinak, např. dle typu procesu).
  - V rámci každé skupiny může být použit jiný dílčí plánovací algoritmus (např. FCFS či round-robin).
  - Je použit také algoritmus, který určuje, ze které skupiny bude vybrán proces, který má aktuálně běžet (často jednoduše na základě priority skupin).
  - Může hrozit hladovění některých (obvykle nízko prioritních) procesů.

- Víceúrovňové plánování se zpětnou vazbou:
  - Víceúrovňové plánování se skupinami procesů rozdělenými dle priorit.
  - Nově vznikající proces je zařazen do fronty s nejvyšší prioritou, postupně klesá do nižších prioritních front, nakonec plánován round-robin na nejnižší úrovni.
  - Používají se varianty, kdy je proces zařazen do počáteční fronty na základě své statické priority. Následně se může jeho dynamická priorita snižovat, spotřebovává-li mnoho procesorového času, nebo naopak zvyšovat, pokud hodně čeká na vstup/výstupních operacích.
    - Cílem je zajistit rychlou reakci interaktivních procesů.

# *Plánovač v Linuxu verze 2.6*

- Víceúrovňové prioritní plánovaní se 100 základními statickými prioritními úrovněmi:
  - Priority 1–99 pro procesy reálného času plánované FCFS (s preempcí na základě priorit) nebo round-robin.
  - Priorita 0 pro běžné procesy plánované tzv. CFS plánovačem.
  - V rámci úrovně 0 se dále používají podúrovně v rozmezí -20 (nejvyšší) až 19 (nejnižší) nastavené uživatelem příkazy `nice` či `renice`.
  - Základní prioritní úroveň a typ plánování mohou ovlivnit procesy s patřičnými právy: viz funkce `sched_setscheduler`.

# Completely Fair Scheduler

- CFS (Completely Fair Scheduler) – novější jádra verze 2.6:
  - Snaží se explicitně každému procesu poskytnout odpovídající procento strojového času (dle jejich priorit).
  - Vede si u každého procesu údaj o tom, kolik (virtuálního) procesorového času strávil.
  - Navíc si vede údaj o minimálním stráveném procesorovém čase, který dává nově připraveným procesům.
  - Procesy udržuje ve vyhledávací stromové struktuře (red-black strom) podle využitého procesorového času.
  - Vybírá jednoduše proces s nejmenším stráveným časem.
  - Procesy nechává běžet po dobu časového kvanta spočteného na základě priorit a pak je zařadí zpět do plánovacího stromu.
  - Obsahuje podporu pro skupinové plánování: Může rozdělovat čas spravedlivě pro procesy spuštěné z různých terminálů (a tedy např. patřící různým uživatelům nebo u jednoho uživatele sloužící různým účelům).

# Plánování ve Windows NT a novějších

- Víceúrovňové prioritní plánovaní se zpětnou vazbou na základě interaktivity:
  - 32 prioritních úrovní: 0 – nulování volných stránek, 1 – 15 běžné procesy, 16 – 31 procesy reálného času.
  - Základní priorita je dána staticky nastavenou kombinací plánovací třídy a plánovací úrovně v rámci třídy.
  - Systém může prioritu běžných procesů dynamicky zvyšovat či snižovat:
    - Zvyšuje prioritu procesů spojených s oknem, které se dostane na popředí.
    - Zvyšuje prioritu procesů spojených s oknem, do kterého přichází vstupní zprávy (myš, časovač, klávesnice, ...).
    - Zvyšuje prioritu procesů, které jsou uvolněny z čekání (např. na I/O operaci).
    - Zvýšená priorita se snižuje po každém vyčerpání kvanta o jednu úroveň až do dosažení základní priority.

# *Inverze priorit*

❖ Problém inverze priorit:

- Nízko prioritní proces si naalokuje nějaký zdroj, více prioritní procesy ho předbíhají a nemůže dokončit práci s tímto zdrojem.
- Časem tento zdroj mohou potřebovat více prioritní procesy, jsou nutně zablokovány a musí čekat na nízko prioritní proces.
- Pokud v systému jsou v tomto okamžiku středně prioritní procesy, které nepotřebují daný zdroj, pak poběží a budou dále předbíhat nízko prioritní proces.
- Tímto způsobem uvedené středně a nízko prioritní procesy získávají efektivně vyšší prioritu.

❖ Inverze priorit nemusí, ale může, vadit: může zvyšovat odezvu systému a způsobit i vážnější problémy, zejména pokud jsou blokovány nějaké kritické procesy reálného času (ovládání hardware apod.).

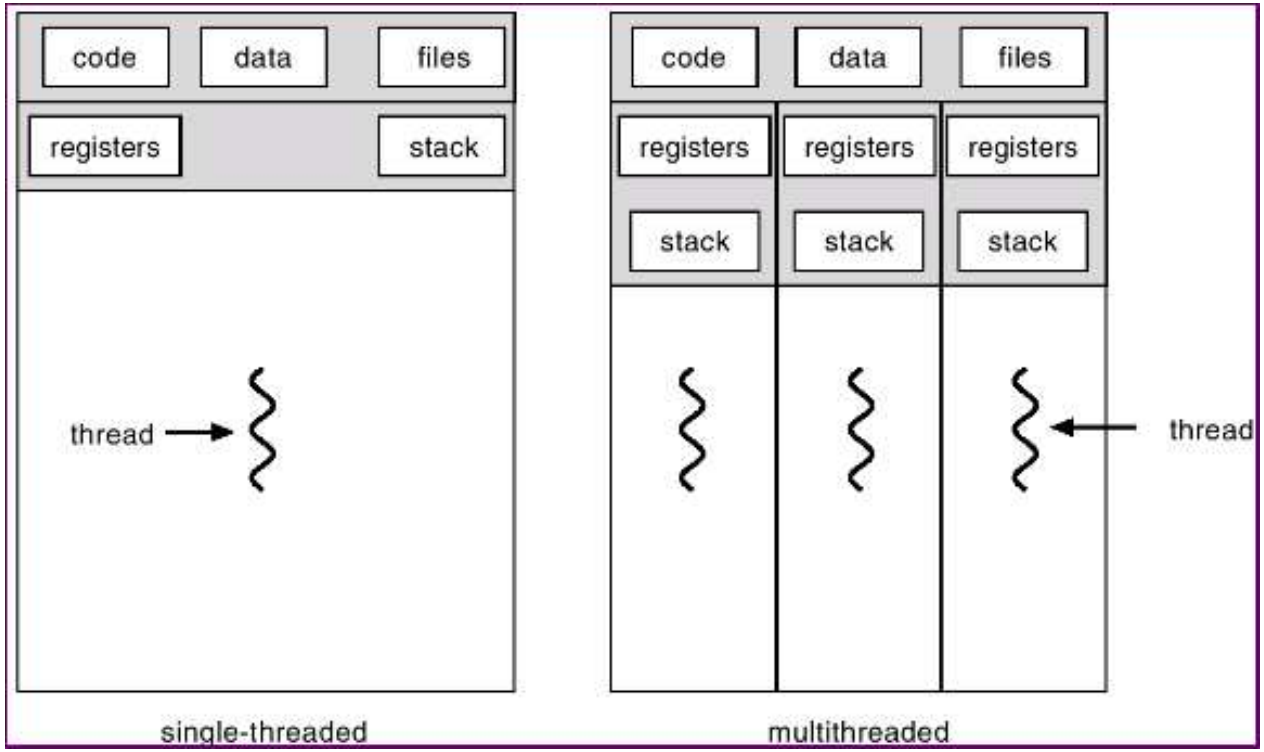
# *Inverze priorit a další komplikace plánování*

- Možnosti řešení inverze priorit:
  - Priority ceiling: procesy v kritické sekci získávají nejvyšší prioritu.
  - Priority inheritance: proces v kritické sekci, který blokuje výše prioritní procesy, dědí (po dobu běhu v kritické sekci) prioritu čekajícího procesu s největší prioritou.
  - Zákaz přerušení po dobu běhu v kritické sekci (na jednoprocesorovém systému): proces v podstatě získává vyšší prioritu než všichni ostatní.

- Další výraznou komplikaci plánování představují:
  - víceprocesorové systémy – nutnost vyvažovat výkon, respektovat obsah cache procesorů, příp. lokalitu pamětí (při neuniformním přístupu do paměti),
  - hard real-time systémy – nutnost zajistit garantovanou odezvu některých akcí.

# Vlákna, úlohy, skupiny procesů

- Vlákno (thread):
  - “odlehčený proces” (LWP – lightweight process),
  - v rámci jednoho klasického procesu může běžet více vláken,
  - vlastní obsah registrů (včetně EIP a ESP) a zásobník,
  - sdílí kód, data a další zdroje (otevřené soubory, signály),
  - výhody: rychlejší spouštění, přepínání.



# Úlohy, skupiny procesů, sezení

❖ Úloha (job) v bashi (a podobných shellech): skupina paralelně běžících procesů spuštěných jedním příkazem a propojených do kolony (pipeline).

❖ Skupina procesů (process group) v UNIXu:

- Množina procesů, které je možno poslat signál jako jedné jednotce. Předek také může čekat na dokončení potomka z určité skupiny (`waitpid`).
- Každý proces je v jedné skupině procesů, po vytvoření je to skupina jeho předka.
- Skupina může mít tzv. vedoucího – její první proces, pokud tento neskončí (skupina je identifikována číslem procesu svého vedoucího).
- Zjišťování a změna skupiny: `getpgid`, `setpgid`.

❖ Sezení (session) v UNIXu:

- Každá skupina procesů je v jednom sezení. Sezení může mít vedoucího.
- Vytvoření nového sezení: `setsid`.
- Sezení může mít řídící terminál (`/dev/tty`).
- Jedna skupina sezení je tzv. na popředí (čte z terminálu), ostatní jsou na pozadí.
- O ukončení terminálu je signálem SIGHUP informován vedoucí sezení (typicky shell), který ho dále řeší (všem, na které nebyl užit nohup/disown, SIGHUP, pozastaveným navíc SIGCONT).

# Komunikace procesů

- IPC = Inter-Process Communication:
  - signály (`kill`, `signal`, ...)
  - roury (`pipe`, `mknod p`, ...)
  - zprávy (`msgget`, `msgsnd`, `msgrcv`, `msgctl`, ...)
  - sdílená paměť (`shmget`, `shmat`, ...)
  - sockety (`socket`, ...)
  - RPC = Remote Procedure Call
  - ...

# Signály

❖ Signál (v základní verzi) je číslo (`int`) zaslané procesu prostřednictvím pro to zvláště definovaného rozhraní. Signály jsou generovány

- při chybách (např. aritmetická chyba, chyba práce s pamětí, ...),
- externích událostech (vypršení časovače, dostupnost I/O, ...),
- na žádost procesu – IPC (`kill`, ...).

❖ Signály často vznikají asynchronně k činnosti programu – není tedy možné jednoznačně předpovědět, kdy daný signál bude doručen.

❖ Nutno pečlivě zvažovat obsluhu, jinak mohou vzniknout “záhadné”, zřídka se objevující a velice špatně laditelné chyby (mj. také motivace pro *formální verifikaci*).

- Mezi běžně používané signály patří:
  - `SIGHUP` – odpojení, ukončení terminálu
  - `SIGINT` – přerušení z klávesnice (`Ctrl-C`)
  - `SIGKILL` – tvrdé ukončení
  - `SIGSEGV`, `SIGBUS` – chybný odkaz do paměti
  - `SIGPIPE` – zápis do roury bez čtenáře
  - `SIGALRM` – signál od časovače (`alarm`)
  - `SIGTERM` – měkké ukončení
  - `SIGUSR1`, `SIGUSR2` – uživatelské signály
  - `SIGCHLD` – pozastaven nebo ukončen potomek
  - `SIGCONT` – pokračuj, jsi-li pozastaven
  - `SIGSTOP`, `SIGTSTP` – tvrdé/měkké pozastavení

- Další signály – viz `man 7 signal`.

# Předefinování obsluhy signálů

❖ Mezi implicitní (přednastavené) reakce na signál patří: ukončení procesu (příp. s generováním core dump), ignorování signálu, zmrazení/rozmrazení procesu.

❖ Lze předefinovat obsluhu všech signálů mimo `SIGKILL` a `SIGSTOP`. `SIGCONT` lze předefinovat, ale vždy dojde k odblokování procesu.

❖ K předefinování obsluhy signálů slouží funkce:

- `sighandler_t signal(int signum, sighandler_t handler);` kde `typedef void (*sighandler_t)(int);`
  - `handler` je ukazatel na obslužnou funkci, příp. `SIG_DFL` či `SIG_IGN`.
- `int sigaction(int signum, const struct sigaction *act, struct sigaction *oldact);`
  - komplexnější a přenosné nastavení obsluhy,
  - nastavení blokování signálů během obsluhy (jinak stávající nastavení + obsluhovaný signál),
  - nastavení režimu obsluhy (automatické obnovení implicitní obsluhy, ...),
  - nastavení obsluhy s příjmem dodatečných informací spolu se signálem.

❖ Z obsluhy signálu lze volat pouze vybrané bezpečné knihovní funkce: u ostatních hrozí nekonzistence při interferenci s rozpracovanými knihovními voláními.

# Blokování signálů

- K nastavení masky blokovaných signálů lze užít:
  - `int sigprocmask(int how, const sigset_t *set, sigset_t *oldset);`
  - `how` je `SIG_BLOCK`, `SIG_UNBLOCK` či `SIG_SETMASK`.

- K sestavení masky signálů slouží `sigemptyset`, `sigfillset`, `sigaddset`, `sigdelset`, `sigismember`.

- Nelze blokovat: `SIGKILL`, `SIGSTOP`, `SIGCONT`.

- Nastavení blokování se dědí do potomků. Dědí se také obslužné funkce signálů, při použití `exec` se ovšem nastaví implicitní obslužné funkce.

- Zjištění čekajících signálů: `int sigpending(sigset_t *set);`

- Upozornění: Pokud je nějaký zablokovaný signál přijat vícekrát, zapamatuje se jen jedna instance! (Neplatí pro tzv. real-time signály.)

# Zasílání signálů

❖ `int kill(pid_t pid, int sig);` umožňuje zasílat signály

- určitému procesu,
- skupině procesů,
- všem procesům, kterým daný proces může signál poslat (mimo `init`, pokud nemá pro příslušný signál definovánu obsluhu).

❖ Aby proces mohl zaslat signál jinému procesu musí odpovídat jeho UID nebo EUID UID nebo saved set-user-ID cílového procesu (`SIGCONT` lze zasílat všem procesům v sezení – session), případně se musí jednat o privilegovaného odesílatele (např. EUID=0, nebo kapabilita `CAP_KILL`).

❖ Poznámka: `sigqueue` – zasílání signálu spolu s dalšími daty.

# Čekání na signál

- Jednoduché čekání: `int pause(void);`
  - Nelze spolehlivě přepínat mezi signály, které mají být blokovány po dobu, kdy se čeká, a mimo dobu, kdy se čeká.

- Zabezpečené čekání: `int sigsuspend(const sigset_t *mask);`
  - Lze spolehlivě přepínat mezi signály blokovanými mimo a po dobu čekání.
  - `mask` jsou blokovány po dobu čekání, po ukončení se nastaví původní blokování.

❖ Příklad – proces spustí potomka a počká na signál od něj:

```
#include <signal.h>
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>

/* When a SIGUSR1 signal arrives, set this variable. */
volatile sig_atomic_t usr_interrupt = 0;

void synch_signal (int sig) {
  usr_interrupt = 1;
}

/* The child process executes this function. */
void child_function (void) {

  /* Let the parent know you're here. */
  kill (getppid (), SIGUSR1);

  exit(0);

}
```

❖ Pokračování příkladu – pozor na zamezení ztráty signálu mezi testem a čekáním:

```
int main (void) {
  struct sigaction usr_action;
  sigset_t block_mask, old_mask;
  pid_t pid;

  /* Establish the signal handler. */
  sigfillset (&block_mask);
  usr_action.sa_handler = synch_signal;
  usr_action.sa_mask = block_mask;
  usr_action.sa_flags = 0;
  sigaction (SIGUSR1, &usr_action, NULL);

  /* Create the child process. */
  if ((pid = fork()) == 0) child_function ();  /* Does not return. */
  else if (pid==-1) { /* A problem with fork - exit. */ }

  /* Wait for the child to send a signal */
  sigemptyset (&block_mask);
  sigaddset (&block_mask, SIGUSR1);
  sigprocmask (SIG_BLOCK, &block_mask, &old_mask);
  while (!usr_interrupt)
    sigsuspend (&old_mask);
  sigprocmask (SIG_UNBLOCK, &block_mask, NULL);
  ...
```

# Synchronizace procesů

Tomáš Vojnar
vojnar@fit.vutbr.cz

Vysoké učení technické v Brně
Fakulta informačních technologií
Božetěchova 2, 612 66 BRNO

1. dubna 2016

# Synchronizace procesů

– **Současný přístup** několika paralelních procesů ke **sdíleným zdrojům** (sdílená data, sdílená I/O zařízení) může vést k nekonzistencím zpracovávaných dat.

– **Časově závislá chyba** (neboli race condition): chyba vznikající při přístupu ke sdíleným zdrojům kvůli různému pořadí provádění jednotlivých paralelních výpočtů v systému, tj. kvůli jejich různé relativní rychlosti.

– Zajištění konzistence dat vyžaduje mechanismy **synchronizace procesů** zajišťující **správné pořadí** provádění spolupracujících procesů.

– **Příklad**: Mějme proměnnou `N`, která obsahuje počet položek ve sdíleném bufferu (např. `N==5`) a uvažujme provedení následujících operací:

```
konzument: N--  ||  producent: N++
```

– Na strojové úrovni a při přepínání kontextu může dojít k následujícímu:

```
producent: register1 = N (register1 == 5)
producent: register1 = register1 + 1 (register1 == 6)
konzument: register2 = N (register2 == 5)
konzument: register2 = register2 - 1 (register2 == 4)
producent: N = register1 (N == 6)
konzument: N = register2 (N == 4  !!!!!)
```

– Výsledkem může být 4, 5, nebo 6 namísto jediné správné hodnoty 5!

# Kritické sekce

– Máme $n$ procesů soutěžících o přístup ke sdíleným zdrojům. Každý proces je řízen určitým programem.

– **Sdílenými kritickými sekcemi** daných procesů rozumíme ty úseky jejich řídících programů přistupující ke sdíleným zdrojům, jejichž provádění jedním procesem vylučuje současné provádění libovolného z těchto úseků ostatními procesy.

– Je možný výskyt více sad sdílených kritických sekcí, které navzájem sdílené nejsou (např. při práci s různými sdílenými proměnnými).

– Obecnějším případem pak je situace, kdy sdílené kritické sekce nejsou vzájemně zcela vyloučeny, ale může se jich současně provádět nejvýše určitý počet.

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

- Vzájemné vyloučení (*mutual exclusion*): nanejvýš jeden (obecně $k$) proces(ů) je v daném okamžiku v dané množině sdílených KS.
- Dostupnost KS:
  - Je-li KS volná (resp. opakovaně volná alespoň v určitých okamžicích), proces nemůže neomezeně čekat na přístup do ní.
  - Je zapotřebí se vyhnout:
    * uváznutí,
    * blokování a
    * stárnutí.

# Problémy vznikající na kritické sekci

– **Data race** (časově závislá chyba nad daty): dva přístupy ke zdroji s výlučným přístupem ze dvou procesů bez synchronizace, alespoň jeden přístup je pro zápis (zvláštní případ chybějícího vzájemného vyloučení).

– **Uváznutí** (**deadlock**) **při přístupu ke zdrojům s výlučným (omezeným) přístupem**: situace, kdy každý proces z určité množiny procesů je pozastaven a čeká na uvolnění zdroje s výlučným (omezeným) přístupem vlastněného nějakým procesem z dané množiny, který jediný může tento zdroj uvolnit. [[ *Obecnější pojetí uváznutí viz dále.* ]]

– **Blokování** (**blocking**) **při přístupu do KS**: situace, kdy proces, jenž žádá o vstup do kritické sekce, musí čekat, přestože je kritická sekce volná (tj. žádný proces se nenachází v ní ani v žádné sdílené kritické sekci) a ani o žádnou z dané množiny sdílených kritických sekcí žádný další proces nežádá.

– **Stárnutí** (též hladovění, **starvation**): situace, kdy proces čeká na podmínku, která nemusí nastat. V případě kritické sekce je touto podmínkou umožnění vstupu do kritické sekce.

– Při striktní interpretaci jsou uváznutí i blokování zvláštními případy stárnutí.

– Zvláštním případem stárnutí je také tzv. **livelock**, kdy všechny procesy z určité množiny běží, ale provádějí jen omezený úsek kódu, ve kterém opakovaně žádají o určitý zdroj s výlučným přístupem, který vlastní některý z procesů dané skupiny (situace podobná uváznutí, ale s aktivním čekáním).

# Petersonův algoritmus

– Možné řešení problému KS pro dva procesy:

```
bool flag[2] = { false, false };  // shared array
int turn = 0;                     // shared variable

// process i (i==0 or i==1):
do {
  //...

  flag[i] = true;
  turn = 1-i;

  while (flag[1-i] && turn != i) ;  // busy waiting
  // critical section
  flag[i] = false;
  //  remainder section
} while (1);
```

– Existuje **zobecnění Petersonova algoritmu** pro $n$ procesů.

# Bakery algoritmus L. Lamporta

– Vzájemné vyloučení pro *n* procesů:

- Před vstupem do KS proces získá “přístupový lístek”, jehož číselná hodnota je větší než čísla přidělená již čekajícím procesům (resp. procesu, který je již v KS).
- Držitel nejmenšího čísla a s nejmenším PID může vstoupit do KS (více procesů může lístek získat současně!).
- Čísla přidělovaná procesům mohou teoreticky neomezeně růst.

```
bool flag[N] = {false}; // shared array
int ticket[N] = { 0 };  // shared array
int j, max=0;           // local (non-shared) variables

// process i
while (1) {
  // ... before the critical section
  flag[i] = true; // finding the max ticket
  for (j = 0; j < N; j++) {
    if (ticket[j] > max) max = ticket[j];
  }
  ticket[i] = max + 1; // take a new ticket
```

## Bakery algoritmus – pokračování

```
  flag[i] = false;
  // give priority to processes with smaller tickets
  //                                (or equal tickets and smaller PID)
  for (j = 0; j < N; j++) {
    while (flag[j]);
    if (ticket[j] > 0 &&
         (ticket[j] < ticket[i] ||
         (ticket[j] == ticket[i] && j<i))) {
           while (ticket[j] > 0);
     }
  }
  // the critical section
  ticket[i] = 0; max = 0;
  // the remainder section
}
```

– Pozor na možnost přetečení u čísel lístků!

# Využití atomických instrukcí pro synchronizaci

– Založeno na využití instrukcí, jejichž atomicita je zajištěna hardware. Používá se častěji než specializované algoritmy bez využití atomických instrukcí.

– Atomická instrukce typu **TestAndSet** (např. `LOCK BTS`):

```
bool TestAndSet(bool &target) {
  bool rv = target;
  target = true;
  return rv;
}
```

– Využití TestAndSet pro synchronizaci na KS:

```
bool lock = false; // a shared variable

// ...
while (TestAndSet(lock)) ;
// critical section
lock = false;
// ...
```

– Atomická instrukce typu **Swap** (např. `LOCK XCHG`):

```
void Swap(bool &a, bool &b) {
  bool temp = a;
  a = b;
  b = temp;
}
```

– Využití Swap pro synchronizaci na KS:

```
bool lock = false; // a shared variable

// ...
bool key = true;   // a local variable
while (key == true)
  Swap(lock,key);
// critical section
lock = false;
// ...
```

– Uvedená řešení vzájemného vyloučení založená na specializovaných instrukcích zahrnují možnost **aktivního čekání**, proto se také tato řešení často označují jako tzv. **spinlock**.

– Lze užít na **krátkých, neblokujících kritických sekcích bez preempce** (alespoň bez preempce na použitém procesoru: proto bývá vlastní použití atomické instrukce uzavřeno mezi zákaz/povolení přerušení).

– Opakovaný zápis sdíleného paměťového místa je problematický z hlediska zajištění **konzistence cache** v multiprocesorových systémech (zatěžuje sdílenou paměťovou sběrnici) – řešením je **při aktivním čekání pouze číst**:

```
// ...
while (TestAndSet(lock))
  while (lock) ;
// ...
```

– Uvedená řešení **nevylučují možnost stárnutí**: bývá tolerováno, ale existují řešení, které tento problém odstraňují.

# Semafory

– Synchronizační nástroj nevyžadující (nebo minimalizující) aktivní čekání – aktivní čekání se v omezené míře může vyskytnout uvnitř implementace operací nad semaforem, ale ne v kódu, který tyto operace používá.

– Jedná se v principu o celočíselnou proměnnou přístupnou dvěmi základními atomickými operacemi:

- **lock** (také P či down) – zamknutí/obsazení semaforu, volající proces čeká dokud není možné operaci úspěšně dokončit a
- **unlock** (také V či up) – odemknutí/uvolnění semaforu.

Dále může být k dispozici inicializace, případně různé varianty zmíněných operací, např. neblokující zamknutí (vždy ihned skončí s příznakem úspěšnosti), pokus o zamknutí s horní mezí na dobu čekání, současné zamknutí více semaforů atp.

– Sémantika celočíselné proměnné $S$ odpovídající semaforu:

- $S > 0$ – odemknuto (hodnota $S > 1$ se užívá u zobecněných semaforů, jež mohou propustit do kritické sekce více než jeden proces),
- $S \leq 0$ – uzamknuto (je-li $S < 0$, hodnota $|S|$ udává počet procesů čekajících na semaforu).
  - Někdy se záporné hodnoty neužívají a semafor se zastaví na nule.

– Využití semaforů pro synchronizaci na KS:

```
semaphore mutex; // shared semaphore

init(mutex,1); // initially mutex = 1
// ...
lock(mutex);
// critical section
unlock(mutex);
// ...
```

– POZOR! Semafory obecně negarantují obsluhu procesů v určitém pořadí (přestože při jejich implementaci bývá využita čekací fronta) ani vyhnutí se stárnutí.

– Konceptuální **implementace semaforu**:

```
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);
  }
}
```

– Provádění lock a unlock musí být atomické. Jejich tělo představuje rovněž kritickou sekci!!!

– Řešení atomicity lock a unlock:

- zákaz přerušení,
- vzájemné vyloučení s využitím atomických instrukcí a aktivním čekáním, tj. s využitím spinlocku:
  - položka reprezentující spinlock je doplněna do struktury reprezentující semafor,
  - spinlock se musí zamknout na vstupu do `lock` a `unlock` a odemknout před výstupem z nich nebo před začátkem čekání v `lock`;
  - používá se u multiprocesorových systémů (spolu se zákazem přerušení pro minimalizaci doby běhu na daném procesoru);
  - čekání pouze na vstup do lock/unlock, ne na dokončení dlouhé uživatelské KS.

– Používají se také:

- read-write zámky – pro čtení lze zamknout vícenásobně,
- reentrantní zámky – proces může stejný zámek zamknout opakovaně,
- mutexy – binární semafory, které mohou být odemknuty pouze těmi procesy, které je zamkly (umožňuje optimalizovanou implementaci).

– POSIX: Semafory dostupné prostřednictvím volání:

- starší rozhraní (System V): `semget`, `semop`, `semctl`,
- novější (viz `man sem_overview`): `sem_open`, `sem_init`, `sem_post`, `sem_wait`, `sem_getvalue`, `sem_close`, `sem_unlink`, `sem_destroy`,
- POSIXová vlákna: `pthread_mutex_lock`, `pthread_mutex_unlock`, ...

– Linux: futexes – fast user-space locks:

- používá se běžná celočíselná proměnná ve sdílené paměti s atomickou inkrementací/dekrementací v uživatelském režimu na úrovni assembleru,
- při detekci konfliktu se volá pro řešení konfliktu jádro – služba `futex` (hlavní operace `FUTEX_WAIT` a `FUTEX_WAKE`),
- rychlost vyplývá z toho, že při malém počtu konfliktů se zcela obejde režie spojená s voláním služeb jádra.

# Monitory

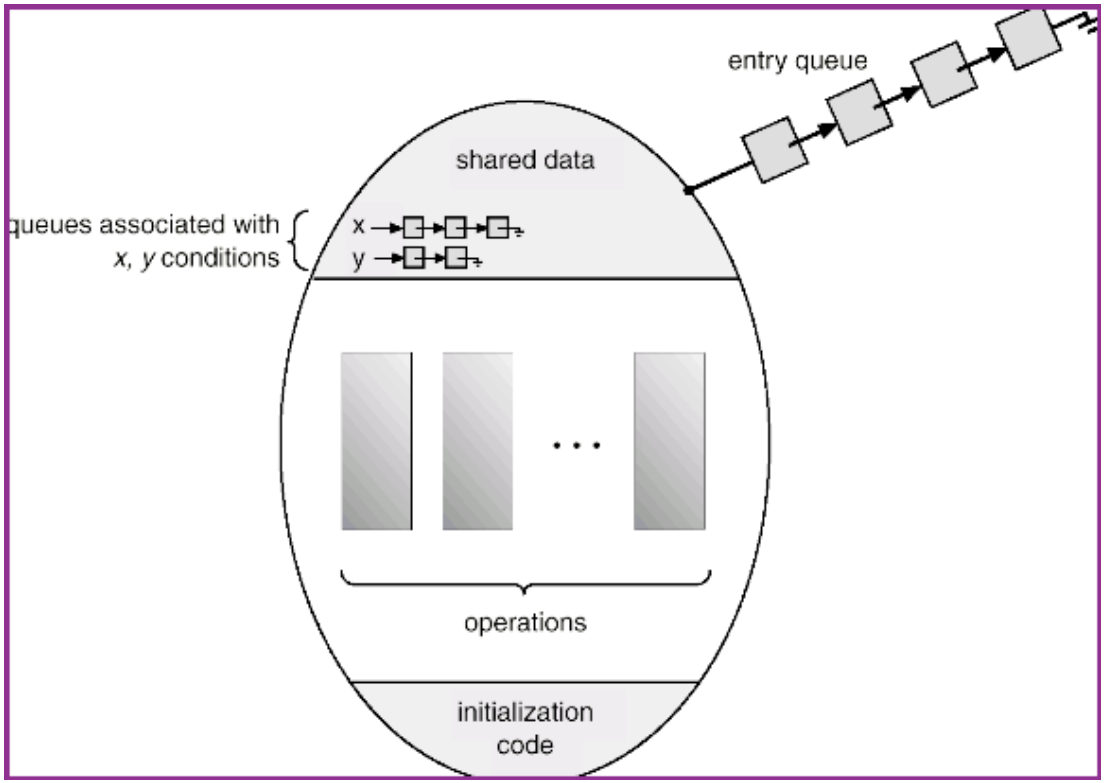
– Jeden z vysokoúrovňových synchronizačních prostředků. Zapouzdřuje data, má definované operace, jen jeden proces může provádět nějakou operaci nad chráněnými daty:

```
monitor monitor-name {
  shared variable declarations

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

– Pro možnost čekání uvnitř monitoru jsou k dispozici tzv. podmínky (*conditions*), nad kterými je možné provádět operace:

- wait() a
- signal(), resp. notify() – pokračuje příjemce/odesílatel signálu; nečeká-li nikdo, jedná se o prázdnou operaci.




– Implementace možná pomocí semaforů.

– Monitory jsou v určité podobě použity v Javě (viz klíčové slovo `synchronized`). Pro POSIXová vlákna jsou k dispozici podmínky `pthread_cond_t` a související funkce `pthread_cond_wait`/ `signal`/ `broadcast`.

# Některé klasické synchronizační problémy

– Komunikace **producenta a konzumenta** přes vyrovnávací paměť s kapacitou omezenou na $N$ položek:

– Synchronizační prostředky:

```
semaphore full, empty, mutex;

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

– Problém **čtenářů a písařů**: libovolný počet čtenářů může číst; pokud ale někdo píše, nikdo další nesmí psát ani číst.

– Synchronizační prostředky:

```
int readcount;
semaphore mutex, wrt;

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

– Čtenář:

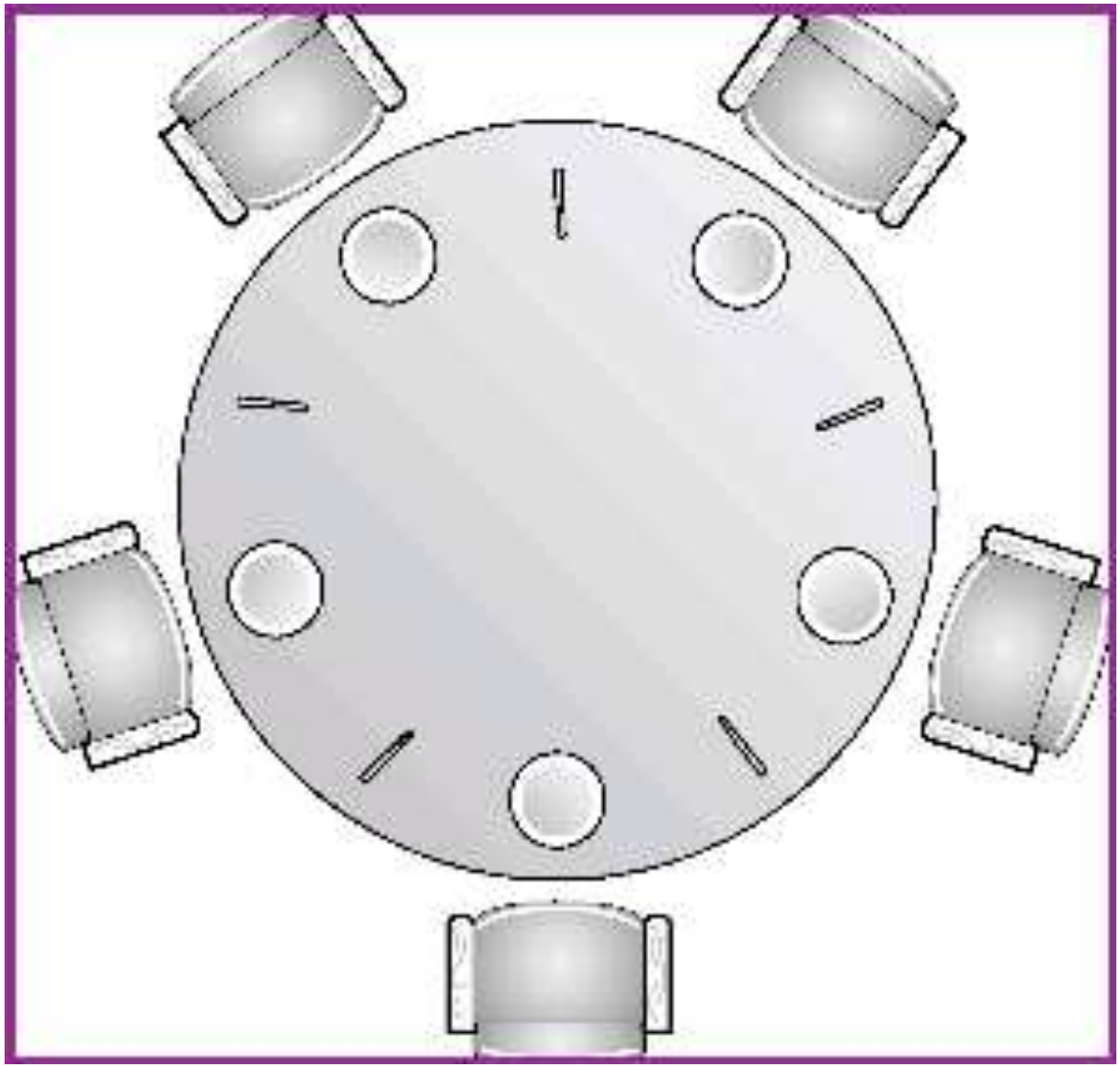
```
do {
  lock(mutex);
  readcount++;
  if (readcount == 1)
    lock(wrt);
  unlock(mutex);
  ...
  // reading is performed
  ...
  lock(mutex);
  readcount--;
  if (readcount == 0)
    unlock(wrt);
  unlock(mutex);
  ...
} while (1);
```

– Písař:

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

– Hrozí “vyhladovění” písařů: přidat další semafor.

– Problém **večeřících filozofů**:



– Řešení (s možností uváznutí):

```
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);
```

– Lepší řešení: získávat obě hůlky současně, získávat hůlky asymetricky, ...

# Uváznutí (deadlock)

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

– Typický příklad (v praxi samozřejmě mohou být příslušná volání ve zdrojovém kódu velmi daleko od sebe a v daném pořadí mohou být zamykány jen za určitých podmínek, takže se uváznutí projeví jen zřídka a špatně se odhaluje; uváznutí také může vyžadovat větší počet procesů):

```
semaphore mutex1, mutex2;

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

– **Obecnější definice**, která počítá i s možností uváznutí bez prostředků s výlučným přístupem (např. při zasílání zpráv): **Uváznutím** rozumíme situaci, kdy každý z množiny procesů je pozastaven a čeká na událost, která by mohla nastat pouze, pokud by mohl pokračovat některý z procesů z dané množiny.

– **Nutné a postačující podmínky uváznutí při přístupu ke zdrojům s výlučným přístupem (Coffmanovy podmínky):**

1. vzájemné vyloučení při používání prostředků,
2. vlastnictví alespoň jednoho zdroje, pozastavení a čekání na další,
3. prostředky vrací proces, který je vlastní, a to po dokončení jejich využití,
4. cyklická závislost na sebe čekajících procesů.
   (Pozor: Nesouvisí nijak s pojmem aktivního čekání cyklením v čekací smyčce.)

– **Řešení:**

- prevence uváznutí,
- vyhýbání se uváznutí,
- detekce a zotavení,
- (ověření, že uváznutí nemůže nastat).

# Prevence uváznutí

– Zrušíme platnost některé z nutných podmínek uváznutí – například:

1. Nepoužívat sdílené prostředky nebo užívat sdílené prostředky, které umožňují (skutečně současný) sdílený přístup a u kterých tedy není nutné vzájemné vyloučení procesů.
2. Proces může žádat o prostředky pouze, pokud žádné nevlastní.
3. Pokud proces požádá o prostředky, které nemůže momentálně získat, je pozastaven, všechny prostředky jsou mu odebrány a proces je zrušen, nebo se čeká, až mu mohou být všechny potřebné prostředky přiděleny.
4. Prostředky jsou očíslovány a je možné je získávat pouze od nejnižších čísel k vyšším (nebo v jiném zvoleném pořadí vylučujícím vznik cyklické závislosti procesů).

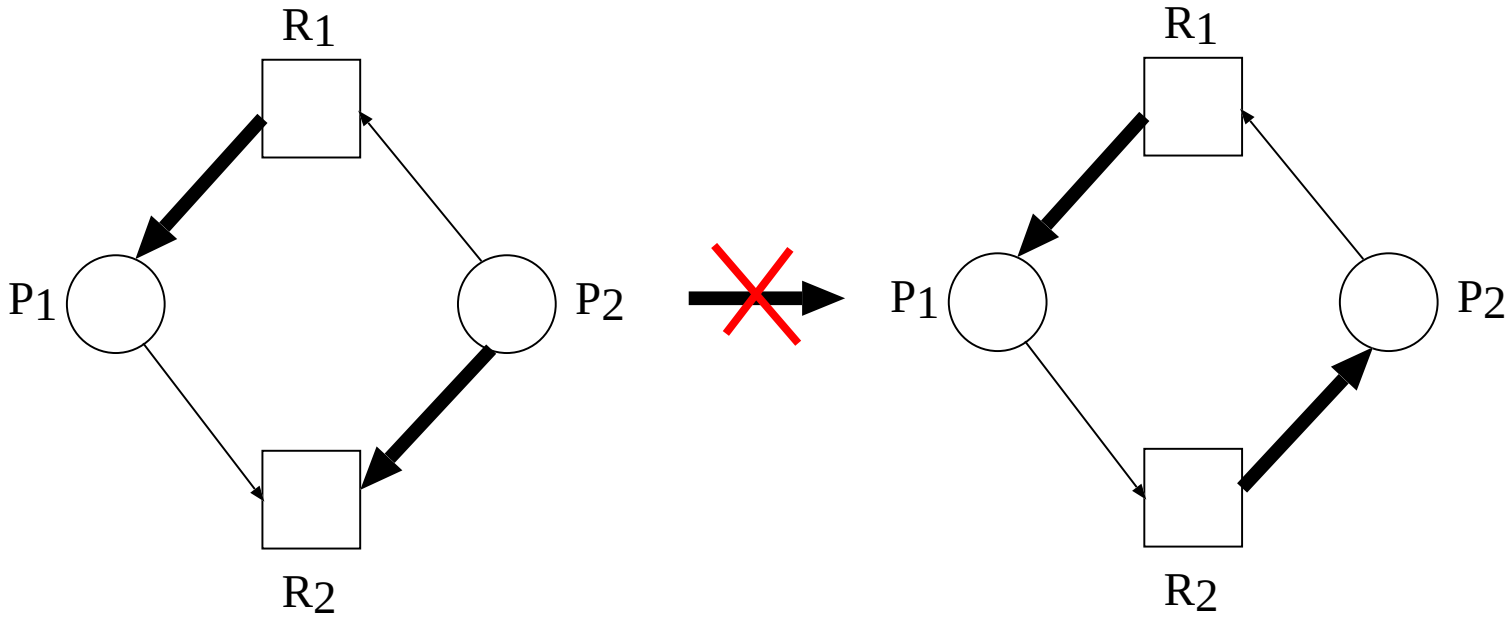
# Vyhýbání se uváznutí

– Procesy předem deklarují určité informace o způsobu, jakým budou využívat zdroje:
v nejjednodušším případě se jedná o **maximální počet** současně požadovaných zdrojů jednotlivých typů.

– Předem známé informace o možných požadavcích jednotlivých procesů a o aktuálním stavu přidělování se využijí k rozhodování o tom, které požadavky mohou být uspokojeny (a které musí počkat) tak, **aby nemohla vzniknout cyklická závislost** na sebe čekajících procesů i v nejhorší možné situaci, která by mohla v budoucnu vzniknout při deklarovaném chování procesů.

– Existuje řada algoritmů pro vyhýbání se uváznutí – např. algoritmus založený na **grafu alokace zdrojů** pro systémy s jednou instancí každého zdroje:

- Vytváří se graf vztahů mezi procesy a zdroji se dvěma typy uzlů (procesy $P_i$ a zdroje $R_j$) a třemi typy hran: který zdroj je kým vlastněn ($R_i \Rightarrow P_j$), kdo o který zdroj žádá ($P_i \Rightarrow R_j$), kdo o který zdroj může požádat ($P_i \rightarrow R_j$).
- Zdroj je přidělen pouze tehdy, pokud nehrozí vznik cyklické závislosti čekajících procesů, což by se projevilo cyklem v grafu při záměně hrany žádosti za hranu vlastnictví.




– Zobecněním je tzv. bankéřův algoritmus pro práci s více instancemi zdrojů.

# Detekce uváznutí a zotavení

– Uváznutí může vzniknout; periodicky se přitom detekuje, zda k tomu nedošlo, a pokud ano, provede se zotavení.

– **Detekce uváznutí**: graf vlastnictví zdrojů a čekání na zdroje procesy (podobný jako graf alokace zdrojů, ale bez hran vyjadřujících možnost žádat o zdroj); cyklus indikuje uváznutí.

– **Zotavení z uváznutí**:

- Odebrání zdrojů alespoň některým pozastaveným procesům, jejich přidělení ostatním a později umožnění získat všechny potřebné zdroje a pokračovat (případně jejich restart či ukončení).
- V každém případě anulace nedokončených operací (*rollback*), nebo nutnost akceptace možných nekonzistencí.

# Formální verifikace

– Pokud použitý systém synchronizace procesů sám spolehlivě nezabraňuje vzniku uváznutí (či jiných nežádoucích chování), je vhodné ověřit, že nad ním navržené rešení je navrženo tak, že žádné nežádoucí chování nehrozí.

– Možnosti odhalování nežádoucího chování systémů (mj. uváznutí či stárnutí) zahrnují:

- inspekce systému (nejlépe nezávislou osobou),
- simulace, testování, vkládání „šumu“ do plánování, dynamická analýza (extrapolace sledovaného chování),
- formální verifikace,
- nebo kombinace všech uvedených přístupů.

– **Formální verifikace** (na rozdíl od simulace a testování) umožňuje nejen vyhledávat chyby, ale také dokázání správnosti systému s ohledem na zadaná kritéria (což znamená, že žádné chyby nezůstaly bez povšimnutí).

– Experimentuje se i s kombinacemi dynamické analýzy za běhu systému s automatickou opravou, nebo alespoň omezením projevů chyby (např. vložení synchronizace – možnost uváznutí (!), vynucení přepnutí kontextu a získání celého časového kvanta před kritickou sekcí, ...).

– **Proces formální verifikace**:

- vytvoření modelu (lze přeskočit při práci přímo se systémem – případně se vytváří model okolí ověřované části systému),
- specifikace vlastnosti, kterou chceme ověřit (některé vlastnosti mohou být generické – např. absence deadlocku, null pointer exceptions apod.),
- (automatická) kontrola, zda model splňuje specifikaci.

– Základní přístupy k formální verifikaci zahrnují:

- *model checking*
- *theorem proving*
- *static analysis*

– **Theorem proving**:

- Využívá (typicky) poloautomatický dokazovací prostředek (PVS, Isabel,Coq,ACL/2,...).
- Vyžaduje obvykle experta, který určuje, jak se má důkaz vést (prestože se v poslední době objevila řada automatických rozhodovacích procedur pro různé logické fragmenty – lze užít pro automatické ověřování fragmentů kódu bez cyklů (automaticky/ručně dodané anotace cyklů a funkcí) či v kombinaci s jinými přístupy).

– **Model checking**:

- Využívá obvykle automatický prostředek (Spin, SMV, Slam/SDV, Blast, JPF, ...).
- Využívá typicky generování a prohledávání stavového prostoru.
- Hlavní nevýhodou je problém stavové exploze, kdy velikost stavového prostoru roste exponenciálně s velikostí modelu, případně práce s neomezeným počtem stavů.

– **Static analysis**:

- Snaha o ověření příslušných vlastností na základě popisu modelu či systému, aniž by se tento prováděl a procházel se stavový prostor (případně se provádí jen na určité abstraktní úrovni).
- Různé podoby: data flow analysis, constraint analysis, type analysis, abstract interpretation.

# Verifikace na FIT – VeriFIT

– Řada témat sahajících od **teoretického výzkumu** (teorie jazyků a automatů a její využití, teorie různých logik), přes **pokročilé algoritmy a datové struktury** potřebné pro efektivní implementaci verifikačních metod po **reálné případové studie**:

- formální verifikace parametrických a nekonečně stavových systémů (např. programy s dynamickými datovými strukturami: seznamy, stromy, ...) pomocí automatů a logik (nástroje Predator a Forester) – partneři LIAFA (Paříž), Verimag (Grenoble), Uppsala University, ...
- inteligentní testování s vkládáním „šumu“, dynamická analýza a sebe-opravování paralelních programů (Java, C) – EU projekt Shadows (partneři: IBM Haifa Research Laboratories, ...).

– Jedna z hlavních aplikací v oboru **Matematické metody v IT** na FIT.

– **Zájemci o možnou bakalářskou práci, diplomovou práci, ... jsou vítáni!**

# Správa paměti

Tomáš Vojnar
vojnar@fit.vutbr.cz

Vysoké učení technické v Brně
Fakulta informačních technologií
Božetěchova 2, 612 66 BRNO

18. dubna 2016

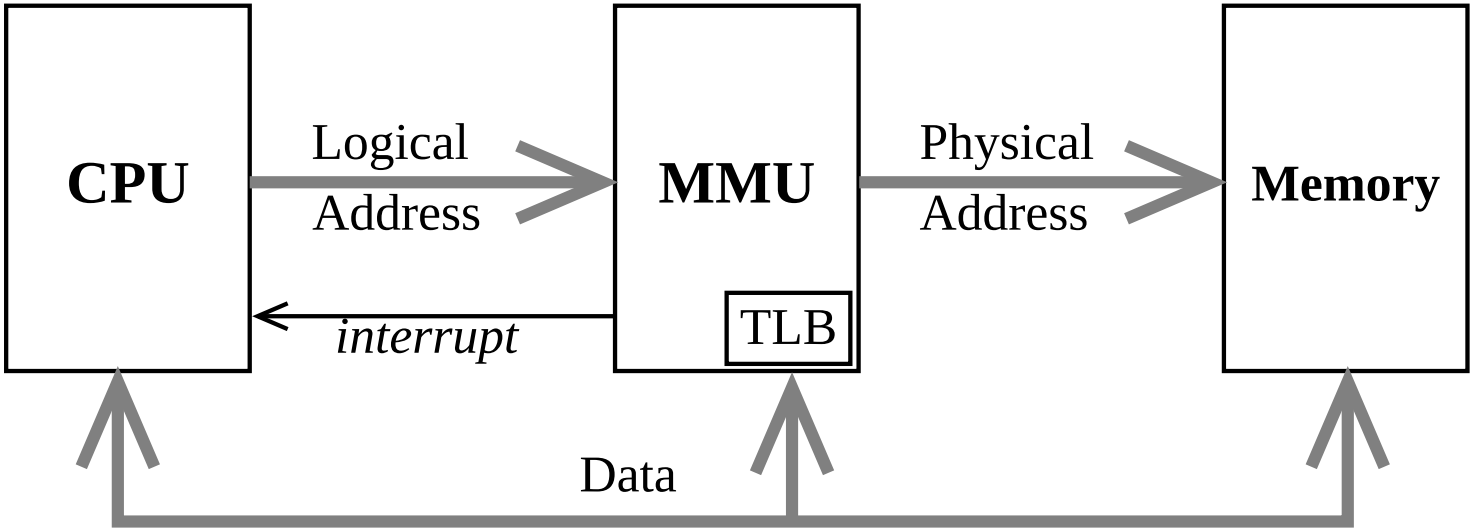
# Správa paměti

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

– Rozlišujeme:

- **logický adresový prostor** (LAP): virtuální adresový prostor, se kterým pracuje procesor při provádění kódu (každý proces i jádro mají svůj),
- **fyzický adresový prostor** (FAP): adresový prostor fyzických adres paměti (společný pro všechny procesy i jádro).

– **MMU (*Memory Management Unit*)** = HW jednotka pro překlad logických adres na fyzické (dnes součást čipu procesoru):




– MMU využívá speciálních registrů a případě i hlavní paměti systému; pro urychlení překladu může obsahovat různé vyrovnávací paměti (např. TLB).

# Přidělování paměti

– Na nejnižší úrovni z hlediska blízkosti HW se můžeme v jádře setkat s přidělováním FAP pro zamapování do LAP, které je konzistentní se způsobem překladu LAP na FAP, jenž je podporován HW daného výpočetního systému:

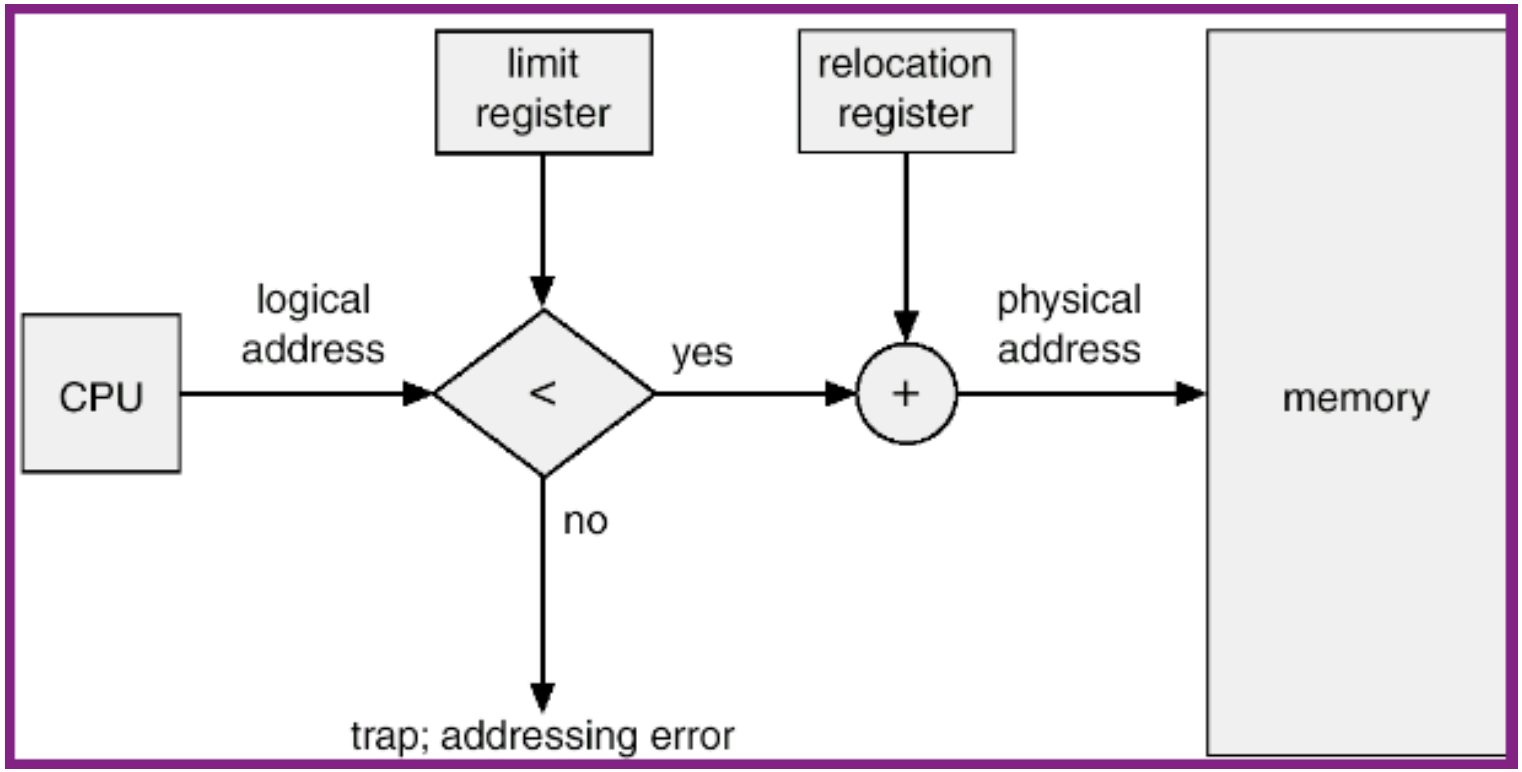
- spojité bloky (*contiguous memory allocation*),
- segmenty,
- stránky,
- kombinace výše uvedeného.

– Na vyšší úrovni se pak používá přidělování LAP pro konkrétní potřeby uživatelských procesů (`malloc`, ... – implementováno mimo režim jádra) i pro běžnou potřebu v jádře (`kmalloc`, `vmalloc`, ...), a to v rámci bloků LAP již zamapovaných do přidělených úseků FAP.

# Contiguous Memory Allocation

– Procesům jsou přidělovány spojité bloky paměti určité velikosti.

– Snadná implementace (jak v HW, tak obsluha v OS):



– Významně se projevuje **externí fragmentace** paměti (FAP):

- přidělováním a uvolňováním paměti vzniká posloupnost obsazených a neobsazených úseků paměti různé velikosti způsobující, že volné místo může být nevyužitelné, protože je nespojité,
- minimalizace pomocí různých strategií alokace paměti (přičemž je třeba brát do úvahy také režii spojenou s daným přidělováním) – mimo *first fit* lze užít např. *best fit*, *worst fit*, *binary buddy*, ...,
- problém se zvětšováním přiděleného prostoru,
- dynamická reorganizace paměti (nákladné!).

– Při nedostatku paměti nutno odkládat na disk veškerou paměť procesu: může být zbytečné, pomalé.

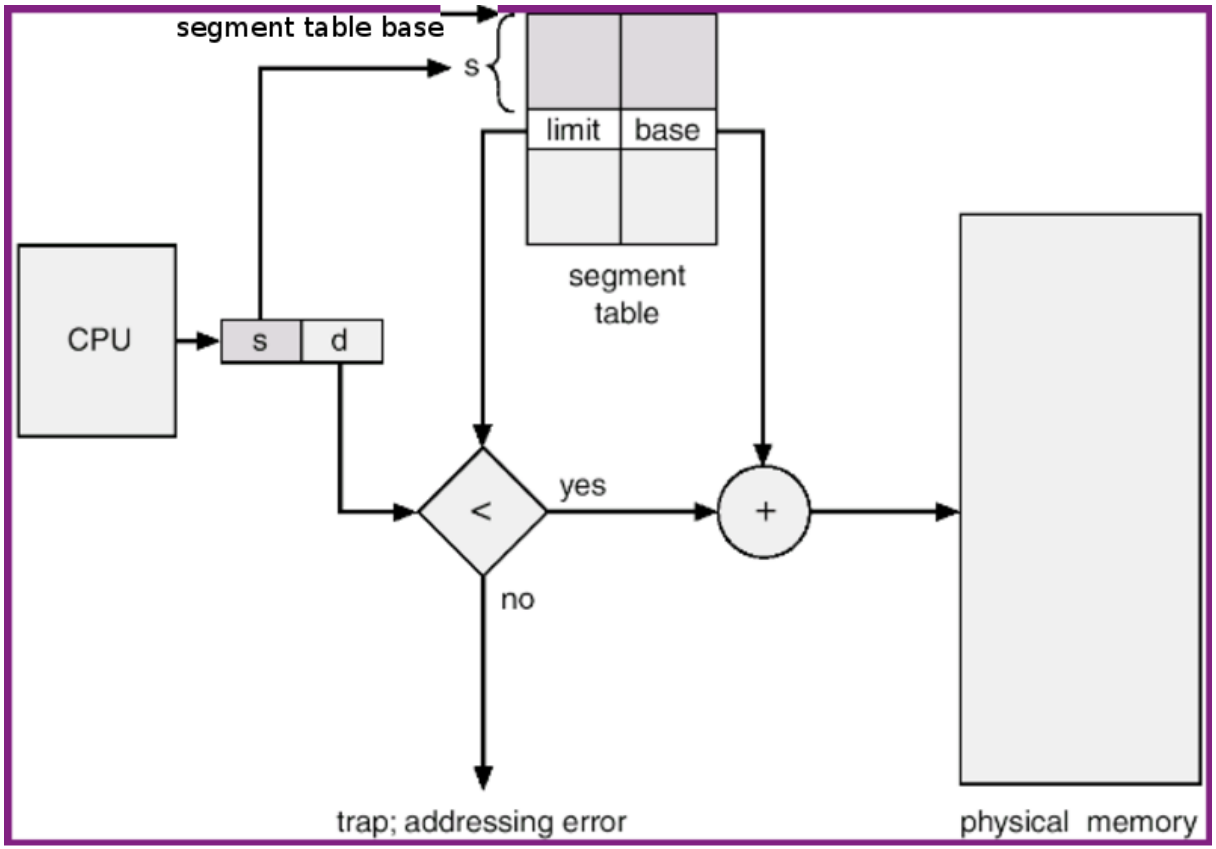
– Není možné jemně řídit přístupová práva, není možné sdílet části paměti.

– Nemusí způsobit **interní fragmentaci**, ve strukturách popisujících obsazení paměti je pak ale nutné pracovat s úplnými adresami. Pro usnadnění evidence přidělené/volné paměti a odstranění možnosti vzniku velmi malých neobsazených úseků se může přidělovat v násobcích určitých bloků, což způsobí interní fragmentaci (toleruje se, vzniká i u ostatních mechanismů přidělování paměti).

# Segmentace paměti

– LAP rozdělen na kolekci segmentů. Různé segmenty mohou být přiděleny překladačem (programátorem) jednotlivým částem procesu (např. procedurám, částem dat, zásobníku, ...).

– Každý segment má číslo a velikost; logická adresa sestává z čísla segmentu a posunu v něm:



– Segmenty mohou být využity jako jednotka ochrany, odkládání a/nebo sdílení paměti (segmenty jen pro čtení, sdílené segmenty, ...).

– Implementace je stále poměrně jednoduchá.

– Paměť přidělována po segmentech; zmírnění dopadů externí fragmentace a jemnější odkládání než u přidělování jediné oblasti – ale problém přetrvává.

– Segmentace je viditelná procesu: komplikace při překladu (tvorbě programů), možnost chyb.

# Stránkování

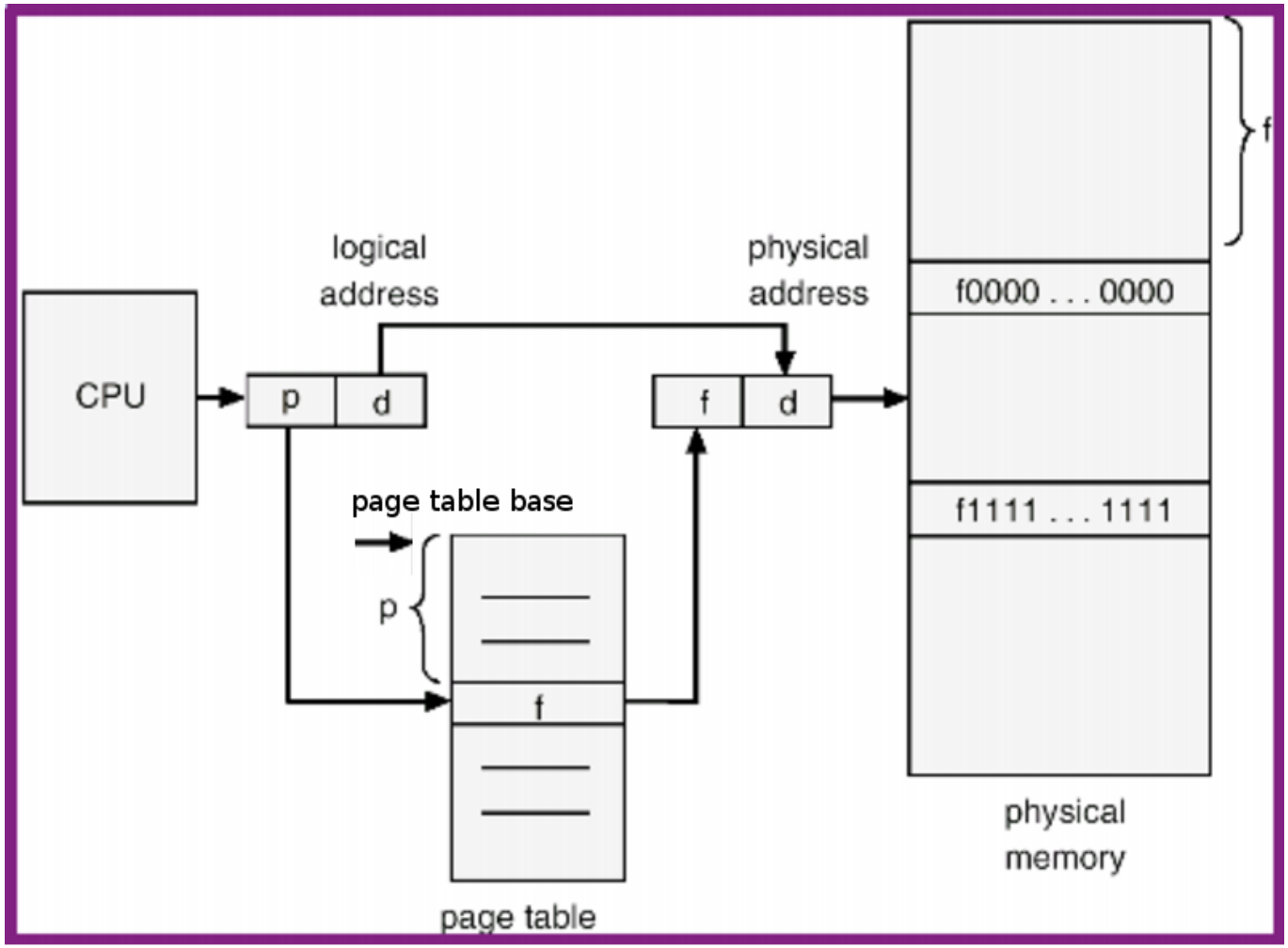
– LAP rozdělen na jednotky pevné velikosti: **stránky** (*pages*).

– FAP rozdělen na jednotky stejné velikosti: **rámce** (*frames*).

– Vlastnosti:

- paměť přidělována po rámcích,
- neviditelné pro uživatelské procesy,
- minimalizovány problémy s externí fragmentací,
  - nevzniká nevyužitelný volný prostor,
  - možné snížení rychlosti přístupu do paměti (větší počet kolizí v různých vyrovnávacích pamětech) a alokace/dealokace (delší práce se strukturami popisujícími aktuální obsah paměti),
  - proto v praxi je snaha přidělovat paměť pokud možno po spojitých posloupnostech rámců: např. pomocí algoritmu „binary buddy“,
- jemná jednotka ochrany (r/rw, user/system, možnost provádění – tzv. NX bit) a/nebo sdílení,
- jemná kontrola odkládání po stránkách,
- složitější implementace, větší režie,
- interní fragmentace.

– V nejjednodušším případě tzv. jednoduchých (či jednoúrovňových) tabulek stránek, OS udržuje **informaci o volných rámcích** a pro každý proces (a jádro) **tabulku stránek** (*page table*):



– Tabulka stránek obsahuje popis mapování logických stránek do FAP a příznaky modifikace, přístupu, přístupová práva (r/rw, user/system, možnost provádění), příznak globality (neodstraňováno automaticky z TLB při přepnutí kontextu), ...

– Tabulky stránek jsou udržovány v hlavní paměti; speciální registr MMU (CR3 u x86) obsahuje adresu začátku tabulky stránek pro aktuální proces.

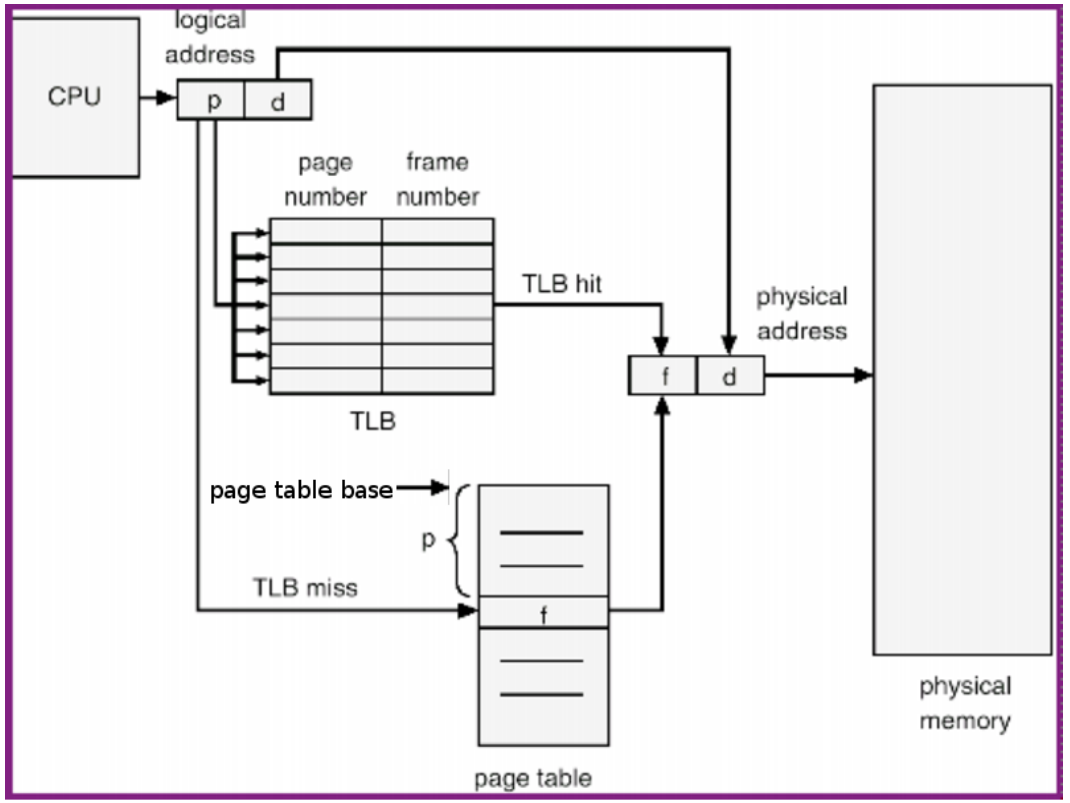
– Každý odkaz na data/instrukce v paměti vyžaduje (u jednoduché tabulky stránek) dva přístupy do paměti: do tabulky stránek a na vlastní data/instrukci.

– Urychlení pomocí rychlé hardwarové asociativní vyrovnávací paměti **TLB (*Translation Look-aside Buffer*)**.

– TLB obsahuje dvojice (číslo stránky,číslo rámce) + některé z příznaků spojených s daným mapováním v tabulkách stránek (přístupová oprávnění, příznak modifikace, příp. další).

– POZOR! V TLB nejsou celé stránky či rámce!

– TLB se prohledává paralelně na základě čísla stránky, a to buď plně (viz obrázek níže), nebo částečně (dojde k indexaci skupiny buněk TLB dle části čísla stránky a pak k paralelnímu dohledání – např. by mohly být užity 2 bity z čísla stránky k rozlišení 4 množin dvojic stránek prohledávaných již tak, jak je znázorněno níže).



– POZOR! K TLB miss může dojít jak při čtení instrukce, tak při čtení jejich operandů, a to u instrukce i operandů i vícenásobně (nezarovnaná instrukce, nezarovnaná data, data delší než jedna stránka).

– Po TLB miss:

- U HW řízených TLB HW automaticky hledá v tabulce stránek.
- U SW řízených TLB (např. MIPS, SPARC) musí v tabulce stránek hledat jádro a patřičně upravit obsah TLB.

– Někdy může být použito více TLB: zvlášť pro stránky obsahující kód a data a/nebo hierarchie více úrovní TLB (různá rychlost, kapacita, cena, spotřeba).

– Při přepnutí kontextu nutno obsah TLB invalidovat. Optimalizace:

- použití globálních stránek označených zvláštním příznakem v tabulce stránek a TLB či
- spojení záznamu v TLB s identifikací procesu (např. PCID u Intelu – viz dále).

– Invalidace TLB nutná samořejmě i po změně obsahu tabulek stránek. Může-li ovlivněné záznamy používat více procesorů, nutno invalidovat TLB na všech procesorech.

– Některé procesory mohou dopředu nahrávat do TLB překlad pro odhadované dále prováděné instrukce.

– **Efektivnost stránkování** velmi silně závisí na úspěšnosti TLB:

- Efektivní přístupová doba: $(\tau+\varepsilon)\alpha+(2\tau+\varepsilon)(1-\alpha)$, kde
  - $\tau$: vybavovací doba RAM
  - $\varepsilon$: vybavovací doba TLB
  - $\alpha$: pravděpodobnost úspěšných vyhledání v TLB (*TLB hit ratio*)
- Např. pro $\tau = 100ns$, $\varepsilon = 20ns$ a $\alpha = 0.98$ dostaneme průměrné zpomalení o 22%.
- TLB hit ratio významně závisí na lokalitě odkazů programu.

– Výše uvedený vztah je sestaven za předpokladu, že po TLB miss a překladu přes tabulky stránek se získaná adresa ihned použije. V praxi se často získaný překlad nejprve vloží do TLB a pak se opakuje překlad přes TLB – pak je ve výše uvedeném vztahu nutno při TLB miss připočíst další přístup do TLB a také čas pro úpravu TLB.

– **Lokalita odkazů** = vlastnost programu – míra toho, kolik různých shluků adres (odpovídajících typicky adresám v různých stránkách) bude proces potřebovat v krátkém časovém úseku.

**Příklad:**

– Při ukládání matic po řádcích je

```
for (j = 0; j < MAX; j++)
  for (i = 0; i < MAX; i++)
    A[i,j] = 0;
```

mnohem méně výhodné než

```
for (i = 0; i < MAX; i++)
  for (j = 0; j < MAX; j++)
    A[i,j] = 0;
```

– Rozdíl se projeví ještě výrazněji při virtualizaci paměti a odkládání stránek na disk.

# Implementace tabulek stránek

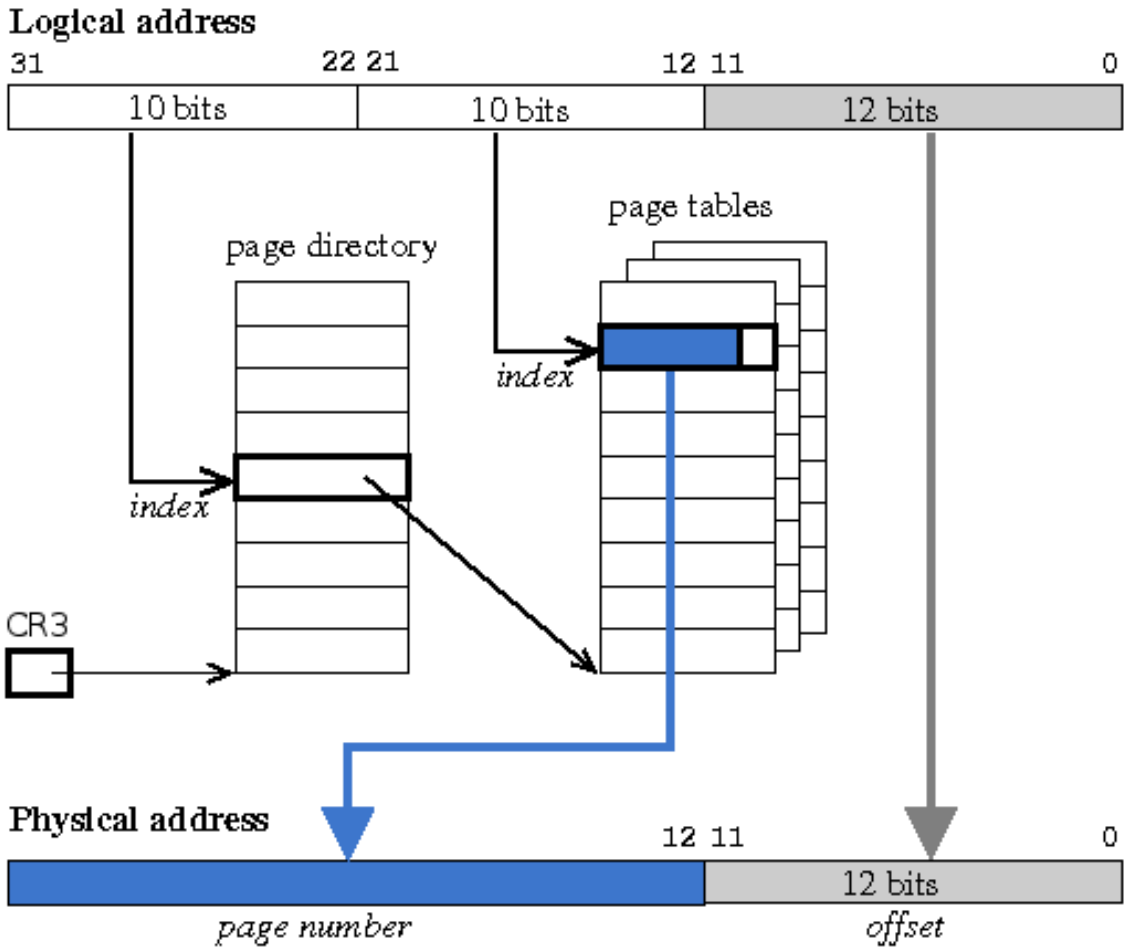
– Tabulky stránek mohou být značně rozsáhlé.

– Pro 32b systém se stránkami o velikosti 4KiB ($2^{12}$) může mít tabulka více než milión položek (přesně $2^{32}/2^{12} = 2^{20} = 1,048,576$ položek). Má-li položka tabulky stránek 4B, dostaneme 4MiB na tabulku stránek pro každý proces, což je příliš na spojitou alokaci (pro 100 procesů 400MiB jen pro tabulky stránek!).

– Pro 64b systémy problém exponenciálně roste: jednoúrovňová tabulka pro 4KiB stránky by měla $2^{64}/2^{12} = 2^{52} = 4,503,599,627,370,496$ položek).

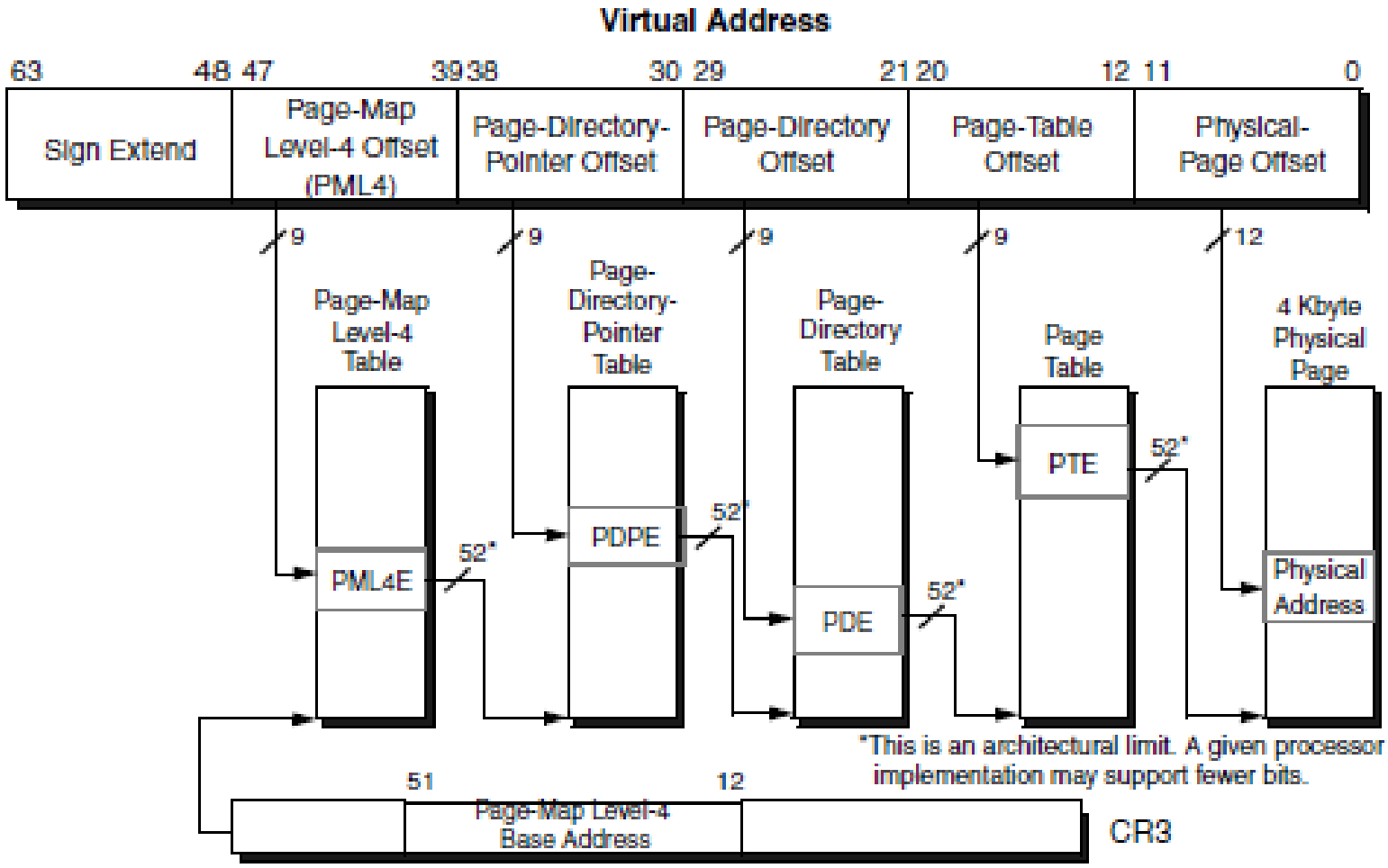
– **Hierarchické tabulky stránek**: tabulka stránek je sama stránkována, vznikají tabulky tabulek stránek, ... (což ovšem dále zpomaluje přístup).

– **Příklad:** dvouúrovňová tabulka stránek procesorů i386+




– Příznak v položkách adresáře stránek určuje, zda je použita i nižší úroveň stránkování. Tímto způsobem je možno pracovat se stránkami o velikosti 4 KiB ($2^{12}$ B) a 4 MiB ($2^{22}$ B).

– 4-úrovňová **tabulka stránek na x86-64**:




– Příznak v položkách tabulek PDPE a PDE určuje, zda je použita i nižší úroveň stránkování. Tímto způsobem je možno pracovat se stránkami o velikosti 4 KiB ($2^{12}$ B), 2 MiB ($2^{21}$ B) a u některých procesorů i 1 GiB ($2^{30}$ B).

– U hierarchických tabulek stránek dále roste zpoždění přístupu do paměti a **roste význam TLB**:

- větší velikost, složitější organizace – více úrovní, oddělení TLB pro překlad adres dat a kódu,
- např. Intel Core i7: zvlášť datová a instrukční TLB 1. úrovně, společná TLB 2. úrovně.

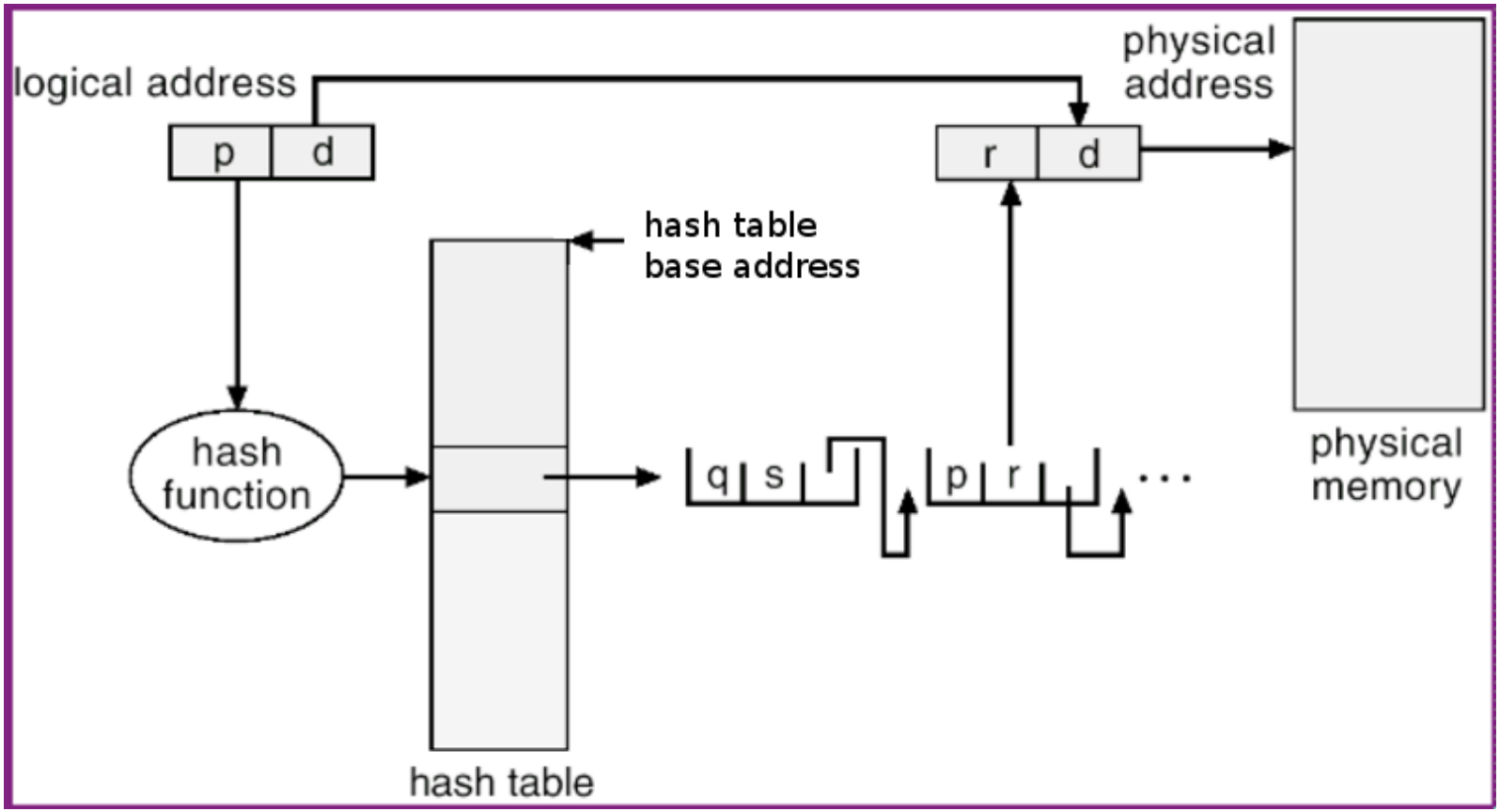
– Další možnosti optimalizace práce s TLB:

- **globální stránky** – záznamy o nich se neodstraňují z TLB při přepnutí kontextu (lze užít např. u stránek jádra, pokud LA jádra je podprostorem LA každého procesu, a to ve fixní části těchto LA),
- **vstupy TLB spojené s identifikátory procesů** – opět není nutno vždy odstraňovat z TLB (např. PCID – process-context ID u procesorů Intel),
- **spekulativní dopředné nahrávání překladu do TLB**,
- využití **specializovaných cache** (mimo TLB) pro ukládání položek (některých úrovní) tabulek stránek.

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

- podpora virtualizace HW – jedna úroveň 4-úrovňových hierarchických tabulek stránek pro virtuální stroje, další pro překlad „fyzických“ adresových prostorů těchto strojů na opravdu fyzické adresy;
- používá příznak globality stránek a identifikaci virtuálního stroje (Intel: VPID – virtual processor ID, AMD: ASID – adress space ID) v položce TLB pro minimalizaci počtu záznamů TLB invalidovaných při přepnutí kontextu.

– **Hashované tabulky stránek**:




– V překladových položkách ve zřetězeném seznamu může a nemusí být celé číslo stránky (nemusí tam být celé, pokud hash funkce některé bity čísla stránky spolehlivě odliší – např. pokud nikdy nebudou kolidovat čísla stránek s odlišnými *n* dolními bity apod.).

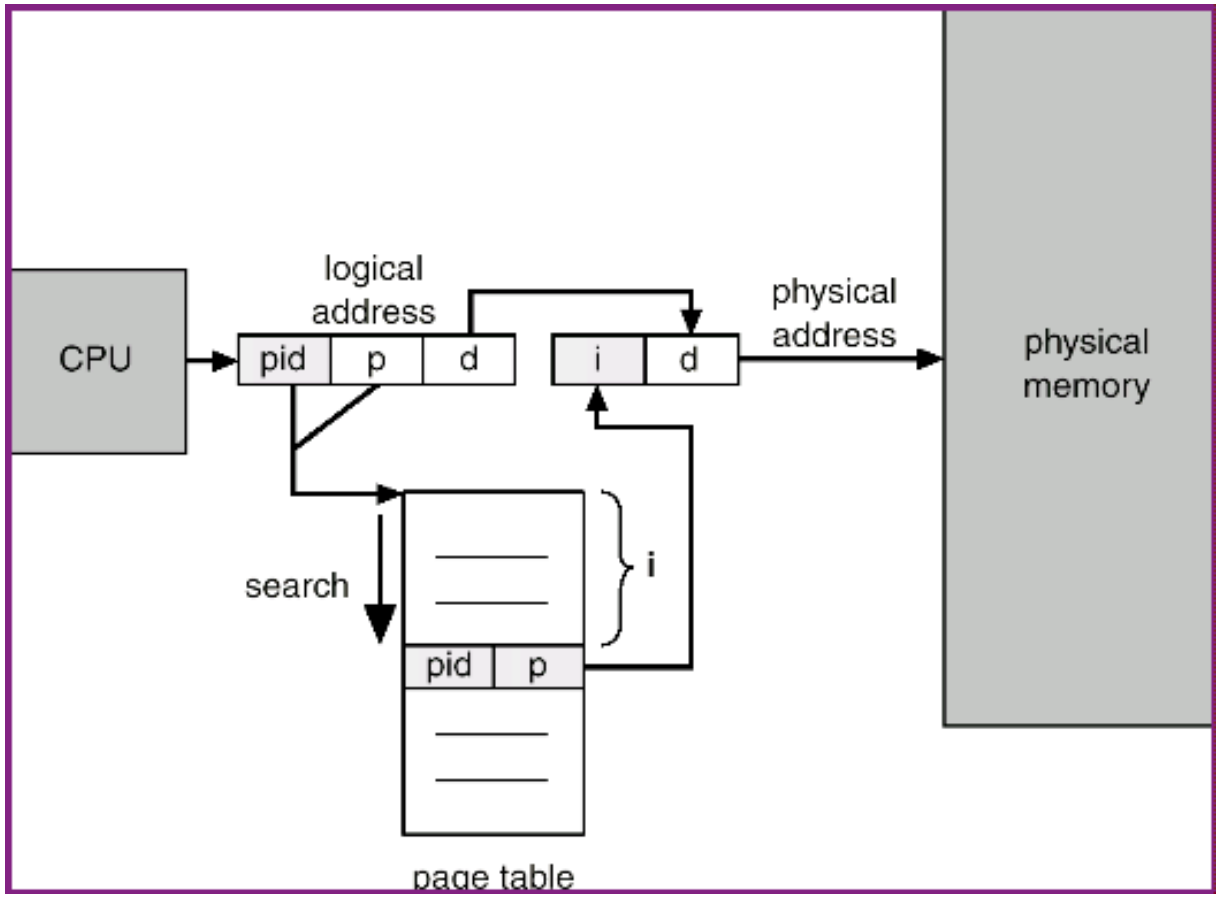
– Obecný zřetězený seznam překladových položek se stejnou hodnotou hash funkce může být nahrazen **fixním počtem překladových položek** ukládaných pro danou hodnotu hash funkce.

- Pokud v takovém případě překlad není nalezen, neznamená to, že stránka není mapována – jádro musí na základě svých pomocných struktur v paměti (čistě programových tabulek stránek, které si jádro musí vést) dohledat překlad a doplnit ho do hashované tabulky stránek namísto některé jiné překladové dvojice.
- Užívá se např. u procesorů PowerPC (několik překladových položek pro jednu hodnotu hash funkce) či Itanium (překladové položky mohou být zřetězeny do seznamu, ale HW tento seznam neprochází, podívá se jen na první položku, zbytek je v režii jádra).

– Hashovaná tabulka stránek **může být sdílená všemi procesy** (čímž se blíží invertované tabulce stránek zmíněné dále).

- V takovém případě se v překladových položkách mimo čísla stránky a čísla rámce udržuje i číslo procesu (nebo jeho část).
- Namísto čísla procesu zde také může být číslo paměťového regionu (nebo jeho část): LA procesů je rozdělen na regiony, čísla regionů lokální v rámci procesů se převádí pomocí další specializované tabulky na globální čísla regionů (některé regiony mohou takto být sdíleny). Pro hashování se pak používá globální číslo regionu a číslo stránky.
- Tento mechanismus je užit u procesorů PowerPC i Itanium.

– **Invertovaná tabulka stránek** – jediná tabulka udávající pro každý rámec, který proces má do něj namapovánu kterou stránku:




– Možno kombinovat s hashováním (např. IBM RS/6000), kdy hashovací funkce spočte počáteční odkaz do invertované tabulky a její položky pak mohou být zřetězeny do seznamu (díky tomu, že index položek invertované tabulky odpovídá číslu rámce, zde ale není ukládáno explicitně číslo rámce).

– OS si pro potřeby správy paměti (odkládání na disk apod.) vede dále i klasické tabulky stránek. Problematická je implementace sdílení stránek (v daném okamžiku je k dispozici mapování jen pro jeden proces – možno řešit mapováním nikoliv pro procesy, ale pro čísla regionů).

# Stránkování a segmentace na žádost

– **Virtualizace paměti** umožňuje procesům (a jádru) pracovat s oddělenými lineárními logickými adresovými prostory.

– V případě mapování mezi LAP a FAP pomocí stránek či segmentů je navíc možné zajistit, aby ne všechen využitý LAP byl celý umístěn ve fyzické paměti.

– **Výhodou** je menší spotřeba paměti, rychlejší odkládání na disk a zavádění do paměti (není zapotřebí odložit nebo zavést celý využitý adresový prostor procesu).

– Pro uložení části LAP, které aktuálně nejsou ve FAP, se využívá prostor na disku.

– Z disku se příslušné části LAP zavádí do FAP pouze tehdy, je-li to zapotřebí (nebo v rámci různých optimalizací i spekulativně těsně před jejich odhadovaným použitím).

– Hovoříme pak o:

- **stránkování na žádost** (*demand paging*) a
- **segmentování na žádost** (*demand segmenting*).

# Stránkování na žádost

– Stránky jsou zaváděny do paměti jen tehdy, jsou-li zapotřebí (příp. jsou v rámci optimalizací spekulativně zaváděny v předstihu).

– Informace o odložených stránkách, příp. stránkách naalokovaných, ale ještě nenačtených si jádro vede ve svých pomocných strukturách – nejsou uloženy v tabulkách stránek, se kterými pracuje MMU (!).

– Stránka je zapotřebí tehdy, dojde-li k odkazu na ni. V jedno- či víceúrovňové tabulce stránek je příznak, zda příslušné stránce je přidělen rámec. Pokud ne, dojde k **výpadku stránky** (*page fault*).

– U hashované či invertované tabulky se to, že stránce není přidělen rámec, pozná z neúspěšného prohledání příslušného seznamu stránek se stejnou hodnotou hashovací funkce či celé tabulky.

– Výpadek stránky je přerušení od MMU udávající, že nelze převést adresu (není definováno mapování v tabulce stránek).

# Obsluha výpadku stránky

– Typická obsluha výpadku stránky v jádře vypadá takto:

1. Kontrola, zda se proces neodkazuje mimo přidělený adresový prostor.
2. Alokace rámce:
   - použijeme volný rámec, pokud nějaký volný je,
   - pokud není,
     - vybereme vhodnou stránku s přiděleným rámcem (*victim page*),
     - stránku odložíme na swap (byla-li modifikována, což udává příznak modifikace v tabulce stránek),
     - použijeme uvolněný rámec.
3. Inicializace stránky po alokaci závislá na předchozím stavu stránky:
   - první odkaz na stránku:
     kód: načtení z programu,
     inicializovaná data: načtení z programu,
     vše ostatní: vynulování (nelze ponechat původní obsah – bezpečnost),
   - stránka byla v minulosti uvolněna z FAP:
     kód: znovu načte z programu (jen pokud nelze přepisovat kódové stránky),
     konstantní data: totéž co kód,
     ostatní: pokud byla modifikována, je stránka ve swapu a musí se načíst zpět do FAP.
4. Úprava tabulky stránek: namapování zpřístupňované stránky na přidělený rámec.
5. Proces je připraven k opakování instrukce, která výpadek způsobila (je ve stavu „připravený“).

# Výkonnost stránkování na žádost

– Efektivní doba přístupu do paměti: $(1-p)T + pD$, kde

- $p$: *page fault rate* = pravděpodobnost výpadku stránky,
- $T$: doba přístupu bez výpadku,
- $D$: doba přístupu s výpadkem.

– Vzhledem k tomu, že $T << D$, musí být $p$ co nejmenší:

- dostatek paměti a jemu přiměřený počet procesů (s ohledem na jejich paměťové nároky),
- vhodný výběr zaváděných a odkládaných stránek,
- lokalita odkazů v procesech.

# Počet výpadků

– K výpadkům může dojít jak při čtení instrukce, tak při práci s každým z jejich operandů, a to u instrukce i u každého z operandů i vícenásobně.

– Vícenásobné výpadky mohou být způsobeny:

- nezarovnáním instrukce,
- nezarovnáním dat,
- daty delšími než jedna stránka,
- výpadky tabulek stránek různých úrovní – a to i vícekrát na stejné úrovni hierarchických tabulek stránek při dotazech na různé dílčí tabulky stránek nacházející se na stejné úrovni.
  - Obvykle alespoň část tabulek stránek je chráněna před výpadkem stránek (zejména se to týká u hierarchických tabulek stránek dílčí tabulky nejvyšší úrovně) mj. proto, aby bylo možno obsloužit výpadky stránek.

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

# Odkládání stránek

– K odložení stránky může dojít při výpadku stránky. Může být provedeno odložení:

- **lokální** – tj. v rámci procesu, u kterého došlo k výpadku
  - je zapotřebí vhodný algoritmus alokace rámců pro použití procesy,
- **globální** – tj. bez ohledu na to, kterému procesu patří která stránka.

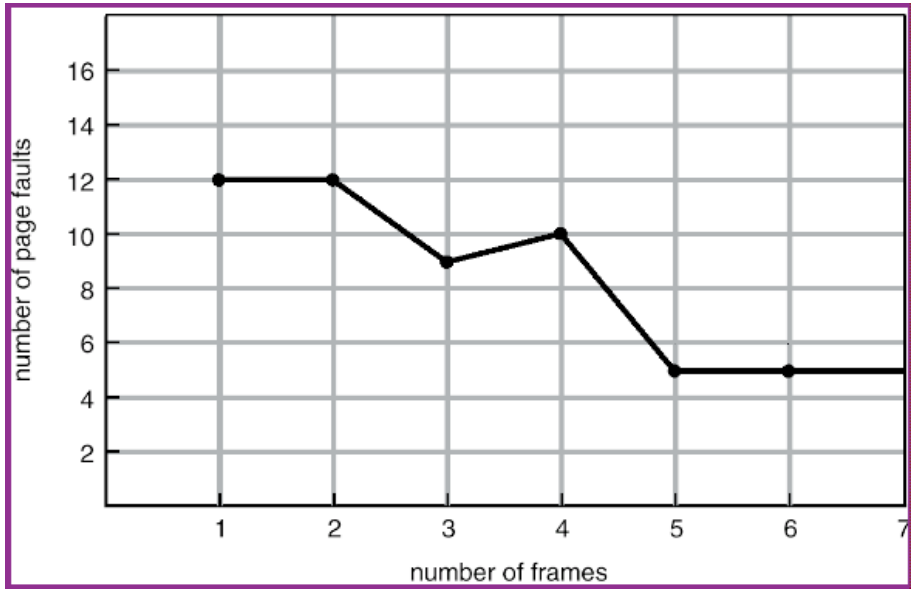
– Typicky je ale neustále udržován určitý počet volných rámců:

- Pokud počet volných rámců klesne pod určitou mez, aktivuje se **page daemon** („zloděj stránek“), který běží tak dlouho, dokud neuvolní dostatečný počet stránek (příp. paměť uvolňuje po částech a dává prostor k běhu ostatním procesům).
- Při výpadku stránky se pak použije rámec z množiny volných rámců.
- Lze doplnit heuristikou, kdy čerstvě uvolněné stránky se okamžitě nepřidělí a zjistí-li se, že byla zvolena nesprávná oběť, lze je snadno vrátit příslušnému procesu k použití.

# Algoritmy výběru odkládaných stránek

– **FIFO**:

- Odstraňuje stránku, která byla zavedena do paměti před nejdelší dobou a dosud nebyla odstraněna.
- Jednoduchá implementace.
- Může odstranit „starou“, ale stále často používanou stránku.
- Trpí tzv. Beladyho anomálií.




- Lze užít v kombinací s přemístěním uvolněného rámce do množiny volných rámců, přidělením jiného volného rámce a možností ihned získat zpět právě uvolněný rámec při následném výpadku signalizujícím, že byla zvolena nesprávná „oběť“.

– **LRU (*Least Recently Used*)**:

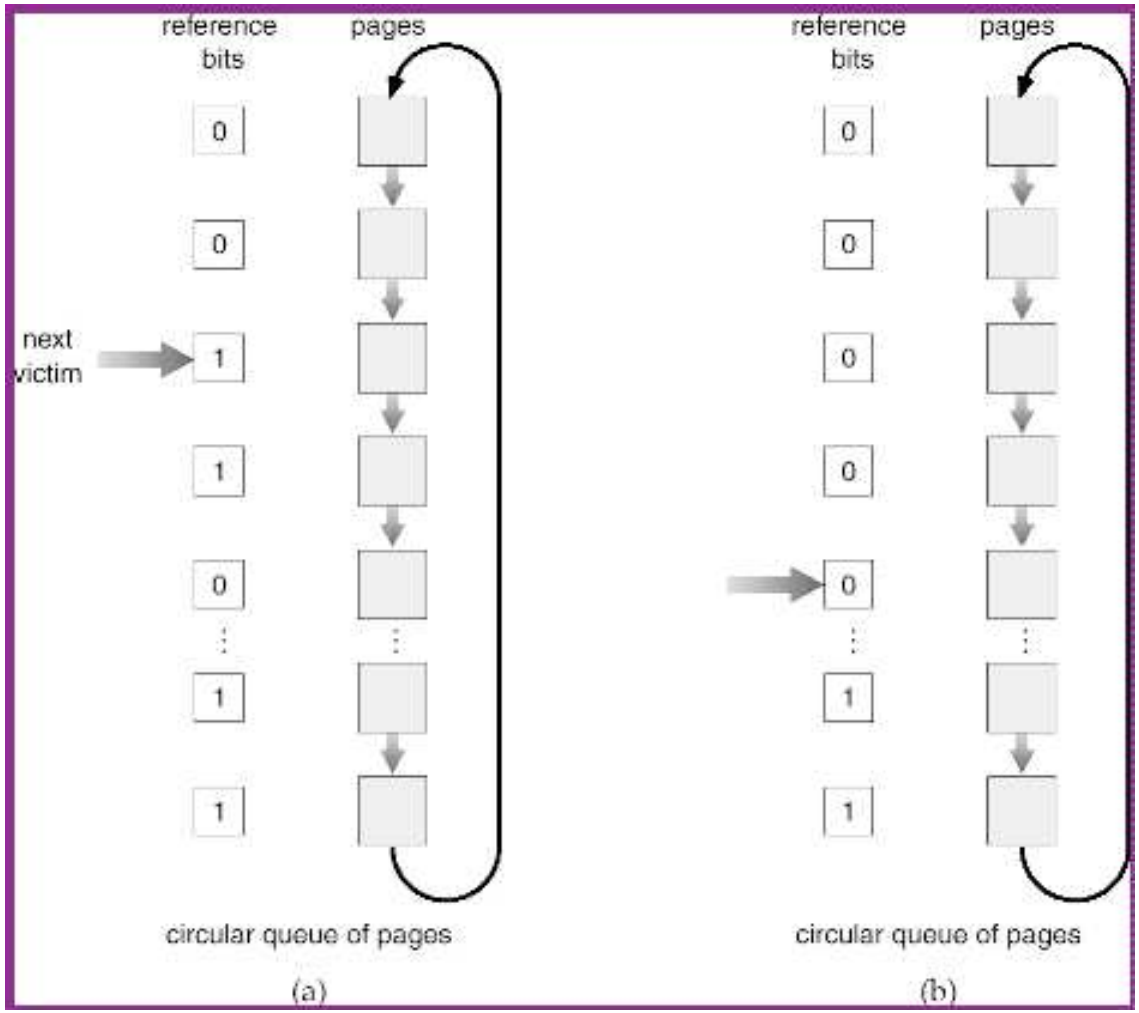
- Odkládá nejdéle nepoužitou stránku.
- Velmi dobrá aproximace hypotetického ideálního algoritmu (tj. algoritmu, který by znal budoucnost a podle budoucích požadavků rozhodoval, co aktuálně odložit tak, aby počet výpadků byl v budoucnu minimální).
  - Někdy se uvádí problémy s cyklickými průchody rozsáhlými poli, spolu se snahou takové přístupy detekovat a řešit zvlášť např. strategií odstranění naposledy použité stránky (most recently used – MRU).
- Problematická implementace vyžadující výraznou HW podporu (označování stránek časovým razítkem posledního přístupu či udržování zásobníku stránek, jehož vrcholem je naposledy použitá stránka).
- Používají se aproximace LRU.

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

- Referenčí bit stránky je HW nastaven při každém přístupu,
- jádro si vede omezenou historii tohoto bitu pro jednotlivé stránky,
- periodicky posouvá obsah historie doprava,
- na nejlevější pozici uloží aktuální hodnotu referenčního bitu a vynuluje ho,
- oběť je vybrána jako stránka s nejnižší číselnou hodnotou historie.
  - Ukládáme-li 4 bity historie a máme stránky s historií 0110 a 1100, odstraníme první z nich (nejlevější bit je poslední reference).

– **Aproximace LRU algoritmem druhé šance**:

- Stránky v kruhovém seznamu, postupujeme a nulujeme referenční bit, odstraníme první stránku, která již nulový referenční bit má.
- Často používaný algoritmus (též označovaný jako tzv. clock algorithm).



– **Modifikace algoritmu druhé šance**:

- upřednostnění nemodifikovaných stránek jako obětí (modifikované se zapíší na disk a dostanou další šanci),
- dva ukazatele procházející frontou s určitým rozestupem – jeden nuluje referenční bit, druhý odstraňuje oběti (tzv. double-handed clock algorithm),
- Linux:
  - fronty „aktivních“ a „neaktivních“ stránek: stránka zpřístupněná dvakrát během jedné periody nulování referenčních bitů a odkládání neaktivních stránek se přesouvá do fronty aktivních stránek, z aktivní fronty se odstraňuje do neaktivní, z neaktivní se pak vybírají oběti,
  - systém se snaží nejprve odkládat stránky použité pro různé vyrovnávací paměti; při určitém počtu stránek namapovaných procesy přejde na odstraňování jejich stránek: prochází procesy a jejich stránky (propojené ve zvláštních seznamu), snaží se odstranit vždy alespoň určitý počet stránek z procesu, neodstraňuje stránky z aktivního seznamu (nebo alespoň referencované),
  - provádí odkládání po určitých počtech projdených a odložených stránek, agresivita se zvyšuje s rostoucím nedostatkem paměti,
  - tzv. „swap token“: stránky procesu, kterému je tato značka udělena, jsou přeskočeny při výběru obětí,
  - při kritickém nedostatku paměti ukončuje některé procesy.

# Alokace rámců procesům (resp. jádru)

– Přidělování rámců procesům (resp. jádru) je významné zejména při lokálním výběru obětí, kde řídí přidělení množiny rámců, v rámci kterých se pak provádí lokální výměny.

– U globálního výběru lze použít pro řízení výběru obětí.

– Je třeba mít vždy přidělen minimální počet rámců pro provedení jedné instrukce: jinak dojde k nekonečnému vyměňování stránek potřebných k provedení instrukce.

– Dále se užívají různé heuristiky pro určení počtu rámců pro procesy (resp. jádro):

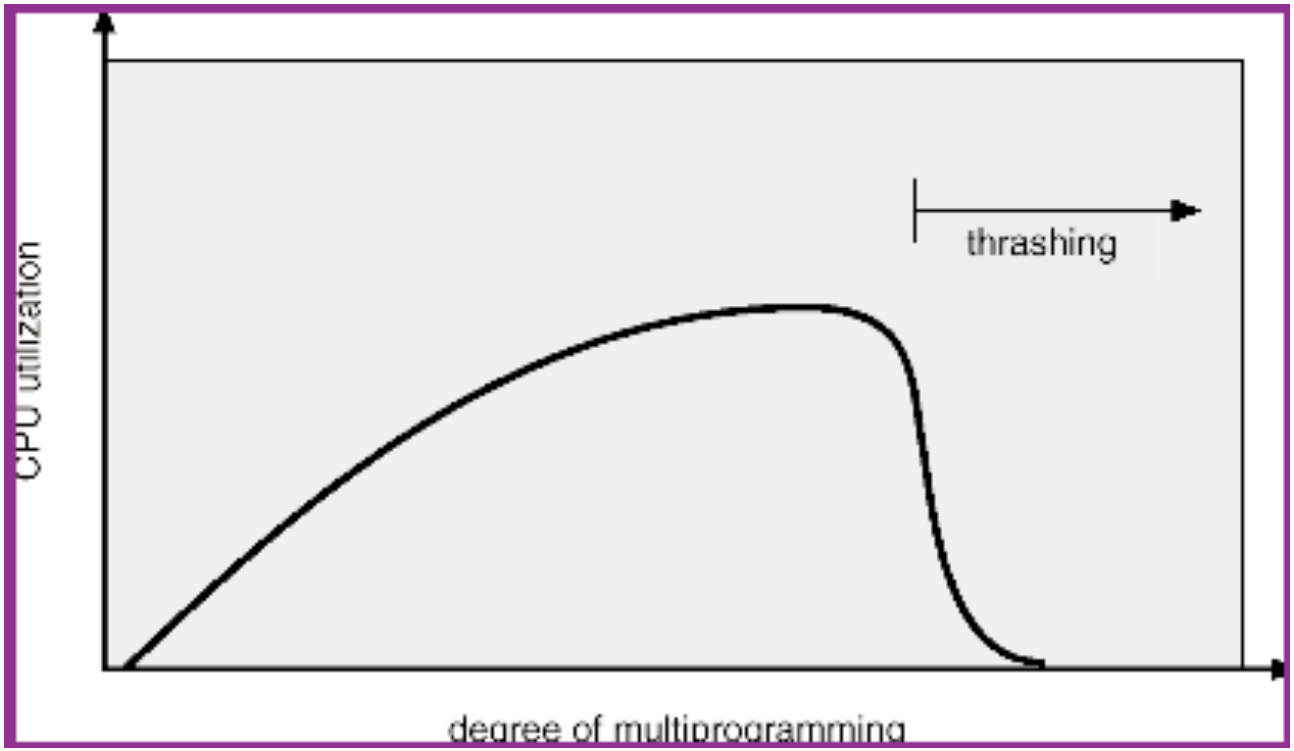
- Úměrně k velikosti programu, prioritě, objemu fyzické paměti, ...
- Na základě **pracovní množiny stránek**, tj. množiny stránek použitých procesem (resp. jádrem) za určitou dobu (aproximace s pomocí referenčního bitu).
- Přímo na základě sledování **frekvence výpadků** v jednotlivých procesech.

– Přidělování rámců s využitím pracovní množiny a kombinace lokální a globální výměny používají novější systémy **Windows**:

- Procesy a jádro mají jistý minimální a maximální počet rámců (meze velikosti pracovní množiny),
  - při dostatku paměti se jim rámce při potřebě přidávají až po dasažení maxima (příp. se povolí i zvětšení maxima),
  - při dosažení maxima, není-li nadbytek paměti, se uplatní lokální výměna,
  - při výrazném nedostatku volných rámců se do vytvoření patřičné zásoby volných rámců systém snaží (s agresivitou rostoucí s rostoucím nedostatkem paměti) odebírat procesům určitý počet v poslední době nepoužitých stránek (u použitých stránek nuluje referenční bit a dává jim další šanci).
- Při výběru obětí dává přednost větším procesům běžícím méně často; vyhýbá se procesům, které způsobily v poslední době mnoho výpadků a procesu, který běží na popředí.
- Oběti v rámci v poslední době nepoužitých stránek vybírá na základě omezené historie přístupů.
- Počáteční meze pracovních množin odvozuje při startu systému z velikosti fyzické paměti.
- Vybrané oběti se snaží nepřidělit k jinému použití okamžitě, aby bylo možno korigovat chyby při volbě obětí.
- Uvedený mechanismus je doplněn odkládáním (swapováním) celých procesů (vybírá dlouho neaktivní procesy).

# Thrashing

– I při rozumné volbě výběru odstraňovaných stránek může nastat tzv. **thrashing**: Proces (v horším případě systém) stráví více času náhradou stránek než užitečným výpočtem.




– Při vážném nedostatku paměti **swapper** (je-li v systému implementován) pozastaví některé procesy a odloží veškerou jejich paměť.

– Jinou možností je ukončení některých procesů.

# Poznámky

– **Prepaging**: snaha zavádět do systému více stránek současně (zejména při startu procesu či po odswapování, ale i za běžné činnosti s ohledem na předpokládanou lokalitu odkazů).

– **Zamykání stránek**:

- zabraňuje odložení,
- užívá se např.
  - u stránek, do nichž probíhá I/O,
  - u (částí) tabulek stránek,
  - u (některých) stránek jádra,
  - na přání uživatele (k jeho vyjádření slouží v POSIXu – v rámci přednastavených limitů a oprávnění – volání `mlock()`): citlivá data, (soft) real-time procesy.

# Sdílení stránek

– Stránkování umožňuje jemnou kontrolu sdílení paměti.

– **Sdílení stránek**:

- kód programů (procesy řízené stejným programem, sdílené knihovny),
- konstantní data nebo doposud nemodifikovaná data u kopií procesů (technologie copy-on-write),
- mechanismus IPC,
- sdílení paměťově mapovaných souborů.

# Sdílené knihovny

– Sdílené knihovny ( `.dll`, `.so`): kód, který je v dané verzi v FAP (a na disku) maximálně jednou a může být sdílen více procesy (procesy nemusí být řízeny stejným programem).

- Výhody: menší programy – lepší využití FAP i diskového prostoru, možnost aktualizovat knihovny.
- Nevýhody:
  - závislost programů na dalších souborech a verzích knihoven,
  - možný pomalejší start programu (je nutné dynamicky sestavit; na druhou stanu se ale zase může ušetřit díky nutnosti nezavádět již zavedené stránky),
  - možné pomalejší volání (nepřímé volání přes sestavovací tabulky; je ale možno ušetřit díky lepší lokalitě paměti – méně výpadků, lepší využití cache).

# Copy-on-Write

– Při spuštění procesu pomocí `fork` se nevytvoří kopie veškeré paměti procesu.

– Vytvoří se pouze tabulky stránek a stránky se poznačí jako copy-on-write.

– K vytvoření fyzické kopie stránky dojde až při pokusu o zápis jedním z procesů.

– **Poznámka:** `vfork` jako alternativa k `fork` s copy-on-write: paměť je skutečně sdílena.

## Sdílená paměť (*shared memory*)

– Forma IPC: Více procesů má mapovány stejné fyzické stránky do LAP.

– `shmget`, `shmat`, `shmctl`, `shmdt`

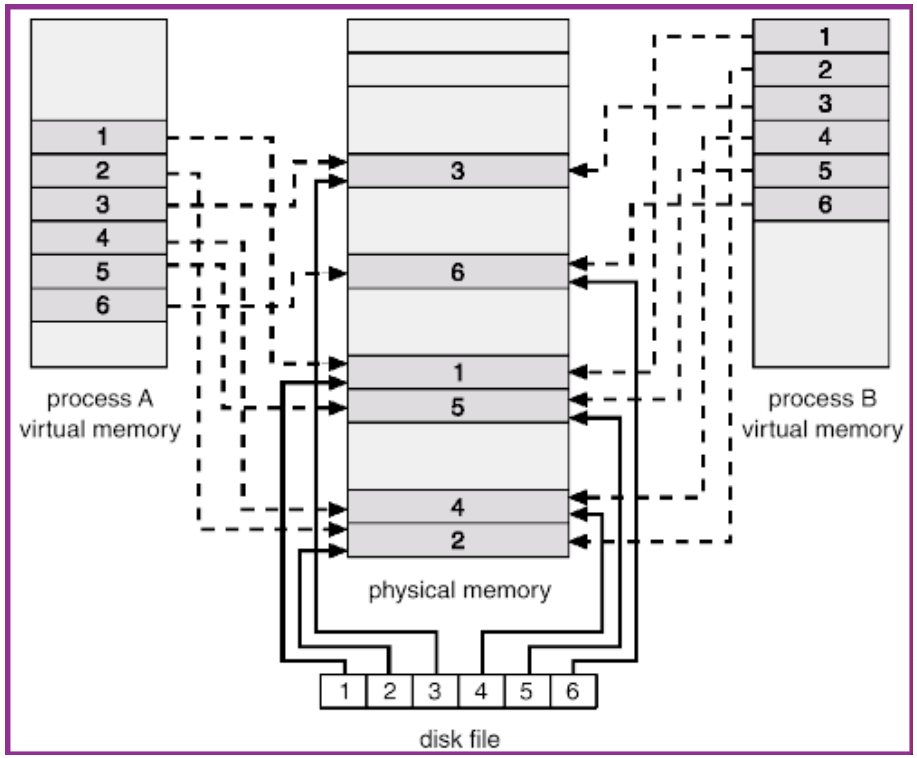
**Příklad:** GIMP plugin

# Paměťově mapované soubory

– Bloky souborů jsou mapovány do stránek v paměti.

– Soubory jsou stránkováním na žádost načteny po stránkách do paměti a dále může být práce se soubory realizována standardním přístupem do paměti namísto použití `read()`/`write()`.

– Šetří se režie se systémovým voláním, kopírování do bufferu a pak do paměti. Umožňuje sdílený přístup k souborům.




– Dostupné prostřednictvím volání `mmap()`.

# Paměťové regiony

– Paměťové regiony jsou jednotkou vyššího strukturování paměti v Unixu (používají se i další: např. v Linuxu tzv. uzly a zóny).

– Jedná se o spojité oblasti virtuální paměti použité za určitým účelem (data – statická inicializovaná data, statická neinicializovaná data, hromada; kód; zásobník; úseky individuálně mapované paměťi – sdílená paměť, paměťově mapované soubory, anonymní mapování).

– Každý proces může mít tabulku regionů procesu udávající pozici regionu v LAP procesu, přístupová práva k regionu a odkazující na systémovou tabulku regionů s dalšími informacemi o regionu, sdílenými mezi procesy – případně má každý proces všechny potřebné údaje přímo ve své tabulce regionů.

– V systémové tabulce regionů je uvedena velikost regionu, typ, i-uzel souboru případně mapovaného do regionu apod.

– Systém může provádět nad regionem jako celkem některé operace: zvětšování/zmenšování (u datového regionu) `brk`, swapování, mapování do paměti aj.

# Rozložení adresového prostoru procesu v Linuxu (i386)

(Anatomy of a Program in Memory, G. Duarte)