

Operační systémy

Tomáš Hudec

Tomas.Hudec@upce.cz

<http://fei-as.upce.cz/~hudec/vyuka/os/>

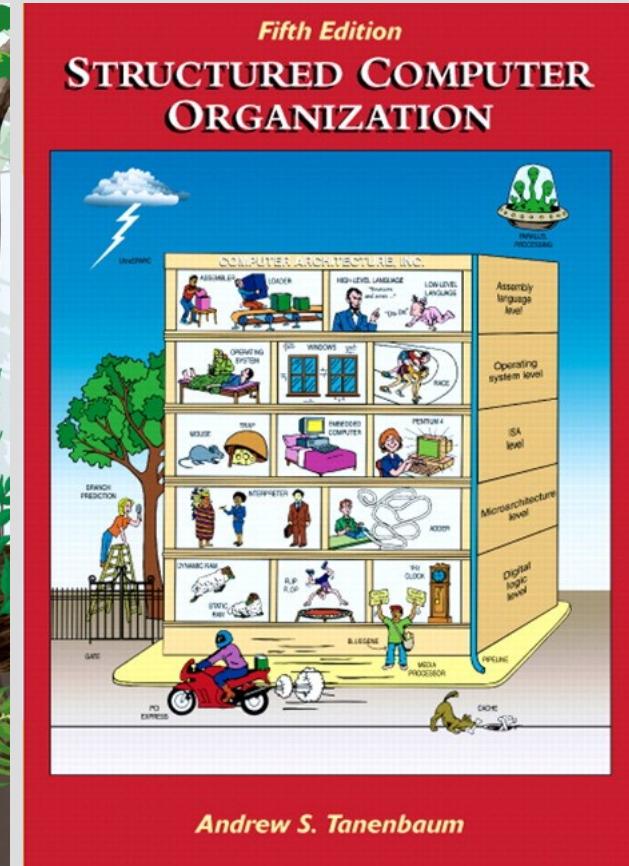
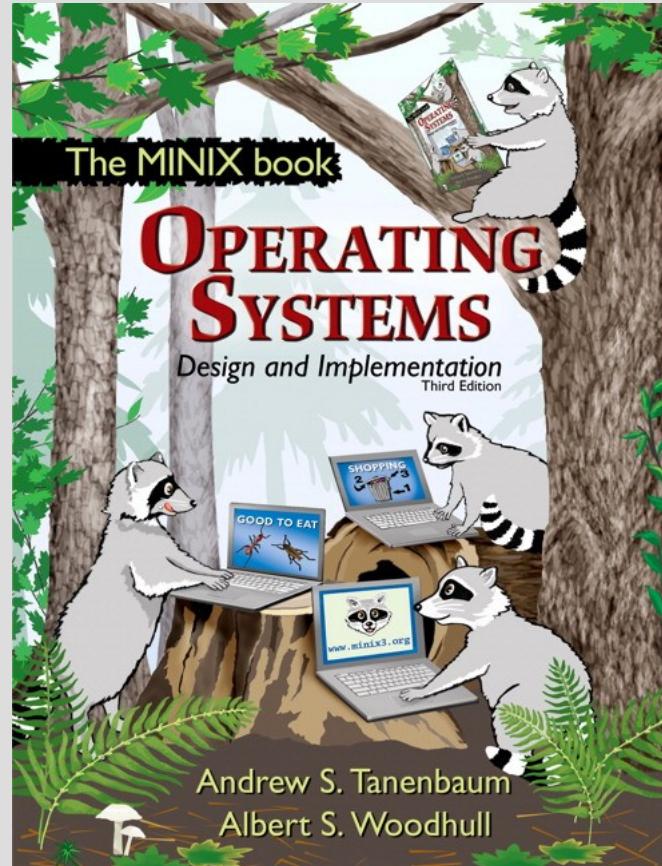
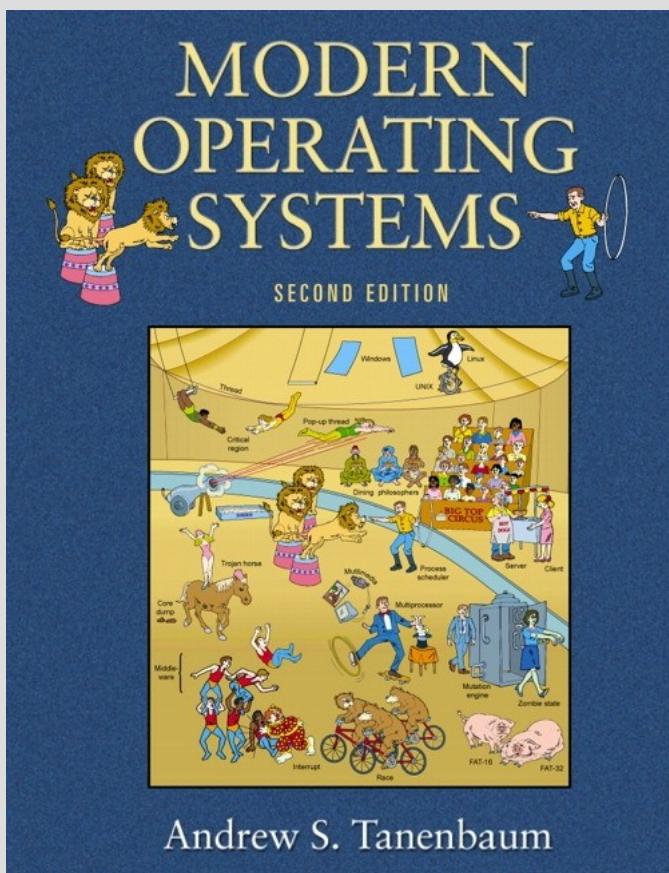
Osnova

- definice OS
- historie
- rozdělení dle určení
- koncepce
- systémová volání
- rozdělení dle struktury
- vlastnosti moderního OS

Literatura

- Tanenbaum A.: *Modern Operating Systems*. 3. vydání. Pearson, 2014. ISBN 987-1-292-02577-3.
- Tanenbaum, A.: *Structured Computer Organization*. 6. vydání. Pearson, 2013. ISBN 978-0-273-76924-3.
- Tanenbaum, A. – Woodhull, A.: *Operating Systems: Design and Implementation*. 3. vydání. Pearson, 2009. ISBN 978-0-13-505376-3.
- *Intel 64 and IA-32 Architectures: Software Developer's Manual: Volume 3A: System Programming Guide, Part 1* [online]. Intel, c 2007–2015 [cit. 2015-01-28]. URL: <http://www.intel.com/design/processor/manuals/253668.pdf>

Literatura (obrázek)



Citáty

“Software is like sex: it's better when it's free.”

– Linus Torvalds, 1996

“I started Linux as a desktop operating system.
And it's the only area where Linux hasn't
completely taken over.

That just annoys the hell out of me.”

– Linus Torvalds, 2012

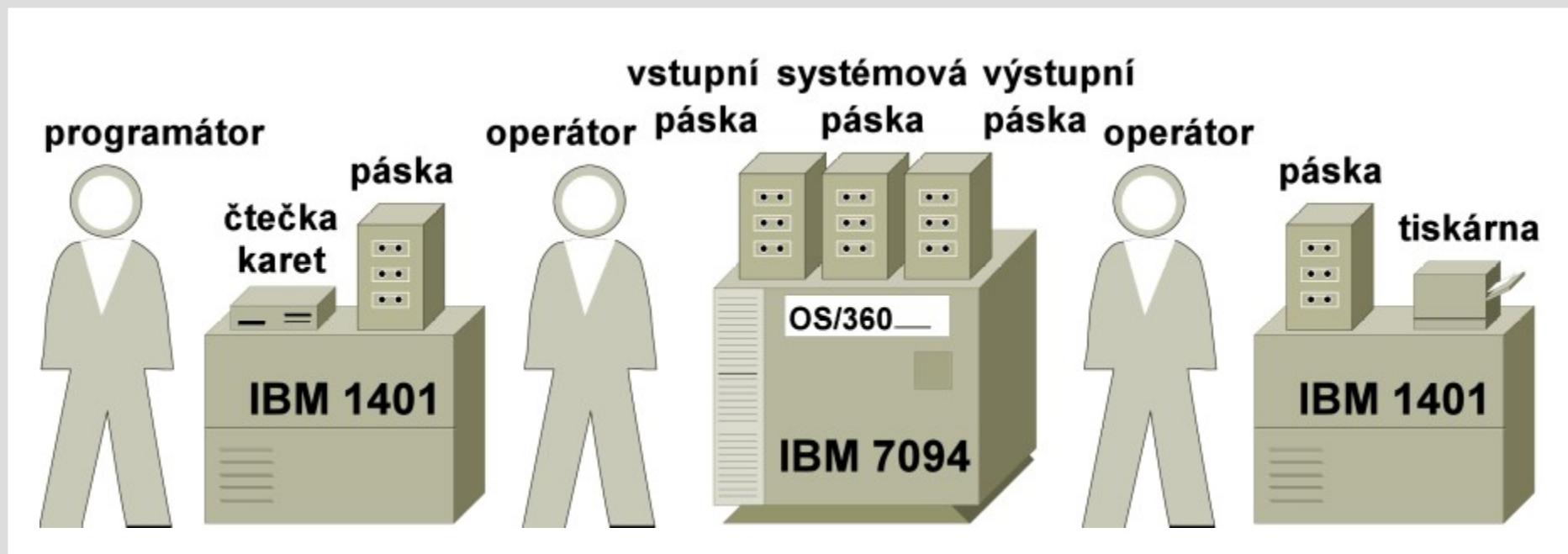
Co je to OS

- **rozšíření stroje (virtualizace)** – pohled „shora“
 - zjednodušující interface, abstrakce
 - příklad: čtení/zápis na disk
- **správce prostředků** – pohled „zdola“
 - procesory, paměti, V/V zařízení
 - příklad: tisk na tiskárnu
 - multiplexing (sharing) – sdílení prostředků
 - v čase (CPU)
 - v prostoru (RAM)

Historie OS, generace (1)

- 1. G 1945–55: elektronky, zásuvné karty
 - počítače zabíraly celé místnosti, OS neexistoval
- 2. G 1955–65: tranzistory a dávkové systémy
 - mainframes
 - obsluha se již dělí
 - designéři, builders, operátoři, programátoři a údržbáři
 - jazyk FORTRAN nebo assembler
 - dávkové systémy

Dávkový systém (obrázek)



Programátoři nosí karty k IBM 1401, kde se programy přepíší do dávky na pásku. Po zaplnění pásky ji operátor přesune k IBM 7094, kde se dávka úloh zpracuje a vygeneruje se výstupní páiska. Operátor ji pak přenese k dalšímu IBM 1401, kde se z ní výsledky přečtou a vytisknou.

Historie OS, generace (2)

- 3. G 1965–80: IO, multiprogramming
 - SSI (small-scale integrated circuits)
 - IBM System/360, OS/360
 - multiprogramming
 - spooling (Simultaneous Peripheral Operation On Line)
 - timesharing, CTSS (Compatible Time Sharing System)
 - MULTICS
 - UNIX
 - POSIX standard

Historie OS, generace (3)

- 4. G 1980–současnost: osobní počítače
 - LSI (large scale integration)
 - (předchůdci) OS:
 - CP/M
 - DOS
 - GUI
 - Mac OS
 - X Window
 - NeXTSTEP
 - Windows

ZOO OS (1)

- **mainframe** OS – sálové počítače
 - obrovská kapacita V/V operací – IBM OS/360, z/OS
- **serverové** OS
 - síťové služby – UNIX, BSD, Linux, Windows Server
- distribuované (rozptylené, vícepočítačové) OS
 - **clustery**, paralelní počítače
 - obvykle modifikace existujících OS (Linux), QNX
- **osobní** OS – PC: mac OS, Linux, Windows

ZOO OS (2)

- **real-time** OS – důležité je dodržení termínů
 - QNX, VxWorks, RT-Linux
- **vestavěné** (embedded) OS
 - PDA, TV-sety, mikrovlnky, mobily
 - QNX, VxWorks, PalmOS, iOS, Linux
- smart card OS, SIM card OS
 - specializované miniaturní vestavěné systémy
 - platební karty – Chip OS (COS, MACOS), MULTOS

Ontogeneze rekapituluje fylogenezi

- z biologie:
 - ontogeneze = vývoj jedince
 - fylogeneze = vývoj druhů
- vývoj OS pro (nová) jednodušší a menší zařízení postupuje podobným procesem jako vývoj OS celkově

Koncepce OS

- procesy
- správa paměti
- správa vstupů a výstupů
- správa úložišť
- systémová volání

Procesy

- programy, které běží v systému
 - adresový prostor (core image), přidělené prostředky
 - spuštění, ukončení procesu, pozastavení, ...
- tabulka procesů, PCB (process control block)
- komunikace mezi procesy
- signály (alarm, V/V operace, ...)
- identifikace uživatele

Problematika – nežádoucí stavy

- problémy souběhu
 - vyhladovění, deadlock, vzájemné vylučování
 - OS nabízí prostředky na jejich řešení
- deadlock – stav uváznutí, příklad:
 - dva procesy potřebují dvě zařízení (A, B)
 - proces 1 má přiděleno zařízení A
 - proces 2 má přiděleno zařízení B
 - oba čekají na uvolnění druhého zařízení

Správa paměti, V/V

- správa paměti
 - logický adresní prostor
 - fyzický adresní prostor
 - virtuální paměť
- vstupy a výstupy (V/V)
 - OS má subsystém správy V/V zařízení
 - ovladače (drivers)

Soubory, souborové systémy

- souborové systémy
 - kořenový adresář (root)
 - cesta (path)
 - absolutní
 - relativní
 - pracovní adresář
- soubory a operace – čtení, zápis, posun
 - file descriptor, handle
- speciální soubory – blokové, znakové, roura

Systémová volání

- volání služeb jádra OS
- volání probíhá většinou přes knihovnu:
 - parametry na stack (v opačném pořadí)
 - volání systémové funkce v knihovně
 - knihovna: nastavení registru na typ volání
 - knihovna: instrukce TRAP (skok do jádra OS)
 - jádro: dispatch, volání příslušného ovladače
 - (návrat do knihovny a programu)

Příklady systémových volání

- procesy
 - vznik, nahrazení, čekání na ukončení, ukončení
- soubory (V/V)
 - otevření, zavření, čtení, zápis, stat, ioctl
- adresáře a souborové systémy
 - vytvoření, zrušení, odkazy, připojování FS
- ostatní (práva, signály, ...)
 - změna práv, signál, zjištění času

Systémová volání – Win32

- Win32 API (Application Interface)
 - vrstva mezi skutečnými funkcemi OS a aplikacemi
 - oproti systému UNIX – založeno na událostech (zpracování fronty zpráv)
 - obsahuje API pro GUI
 - extrémně velké (tisíce procedur)
 - POSIX: asi sto systémových volání

Dělení OS dle struktury – monolitické jádro

- monolitické OS (The Big Mess)
 - vše v jednom – vnitřně nečleněné jádro
 - z hlediska hierarchie volání procedur
 - každá procedura může volat libovolnou jinou
 - mohou mít i strukturu:
 - hlavní program (obsahuje dispatcher)
 - obslužné procedury
 - užitkové procedury
 - procedury mají pevně definované rozhraní

Dělení OS dle struktury – vícevrstvé jádro

- vícevrstvé OS
 - vrstva smí volat jen procedury stejné nebo nejbližší nižší vrstvy – zajištěno HW (úrovně oprávnění)
 - MULTICS
 - Dijkstra: THE (Technische Hogeschool Eindhoven):
 - 5 – operátor
 - 4 – uživatelské programy
 - 3 – správa V/V zařízení, buffering
 - 2 – komunikace mezi procesy a konzolí operátora
 - 1 – správa paměti
 - 0 – alokace CPU a multiprogramming

Dělení OS dle struktury – virtuální jádro

- **virtuální stroje**, virtualizace na úrovni jádra
 - obecně: VM monitor / hypervizor – multiprogramming
 - již v r. 1972: IBM VM/370 + OS/360 nebo CMS
 - jediné jádro OS + kontejnery: izolace skupin procesů, uživatelů (vč. správce), sítě i souborového systému
 - Solaris 11 – kernel zones
 - FreeBSD Jails
 - Linux-Vserver – modifikované jádro Linuxu
 - OpenVZ – kontejnery, modifikované jádro Linuxu
 - Linux Containers (LXC, od verze jádra 2.6.24, 2008) – cgroups: izolace prostředků: CPU, paměť, disk I/O, síť, ...

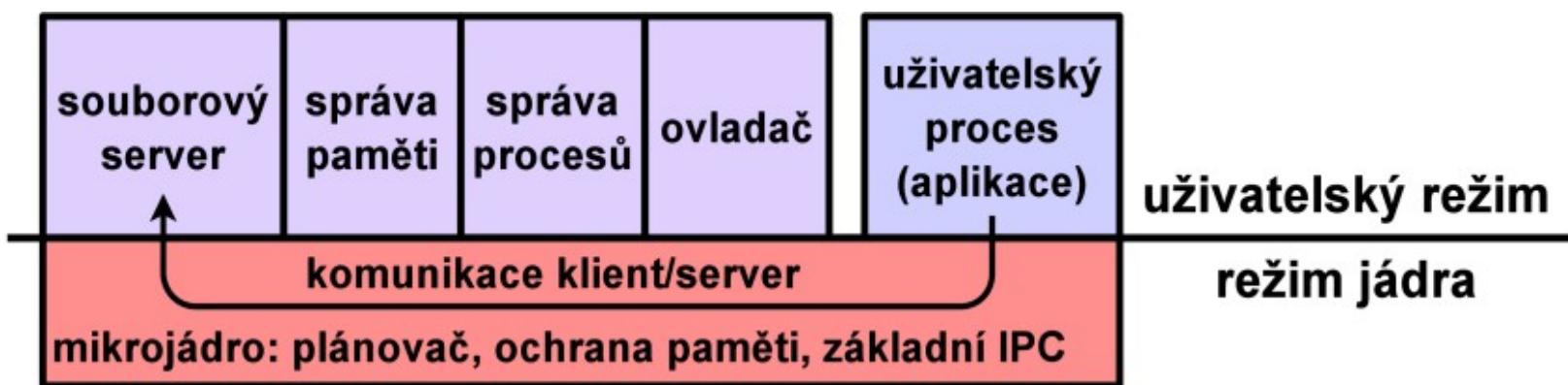
Dělení OS dle struktury – exokernel

- **exokernely**
 - klon počítače založen na rozdělení HW
 - multiplex systémových prostředků
 - minimalistické jádro se základními abstrakcemi
 - důležitá designová rozhodnutí o abstrakcích lze učinit až na vyšší (uživatelské, programátorské) úrovni
 - koncept vznikl na MIT
 - projekty Aegis (proof of concept) a XOK

Dělení OS dle struktury – mikrojádro

- model klient-server, **mikrojádro** (microkernel)
 - trend moderních OS
 - mikrojádro – správa komunikace mezi procesy
 - klientské procesy – správa paměti, FS, ovladače, ...
 - náročné na implementaci i režii
 - GNU/Hurd (Hird of Unix-Replacing Daemons, Hird = Hurd of Interfaces Representing Depth)
 - QNX [kjunix]
 - unixový real-time OS (pro vestavěná zařízení)
 - MINIX 3

Mikrojádro (obrázek)



Minimalistické jádro zajišťuje jen nejnutnější funkce:

- mikrogramování – alokace CPU (plánovač)
- zajištění ochrany paměti
- základní meziprocesová komunikace

Vlastnosti moderního OS

- preemptivní (a efektivní) plánování procesů
- izolace procesů
- efektivní správa paměti
 - operační paměť – logické adresování, virtualizace
 - úložiště – souborové systémy
- víceuživatelský OS, izolace uživatelů
 - implementace oprávnění (procesů, souborů)
- podpora IPC – komunikace a synchronizace

OS – Přehled hardware

Tomáš Hudec

Tomas.Hudec@upce.cz

<http://fei-as.upceuecebny.cz/usr/hudec/vyuka/os/>

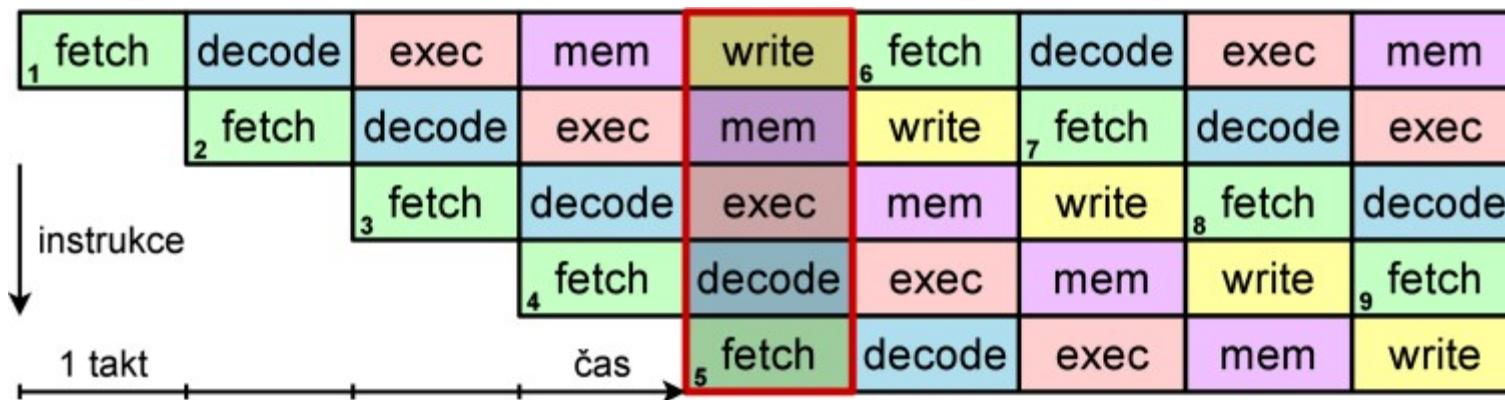
Procesory – CPU (1)

- CPU – Central Processing Unit
 - zpracovává instrukce
 - RISC – Reduced Instruction Set Computer
 - CISC – Complete Instruction Set Computer
 - zpracování instrukcí má obvykle tyto fáze
 - **fetch** – načtení instrukce
 - **decode** – dekódování
 - **execute** – provedení
 - **write-back** – zápis výsledků
 - ALU – aritmeticko-logické jednotky
 - FPU – numerický koprocessor (Floating Point Unit)

Procesory – CPU (2)

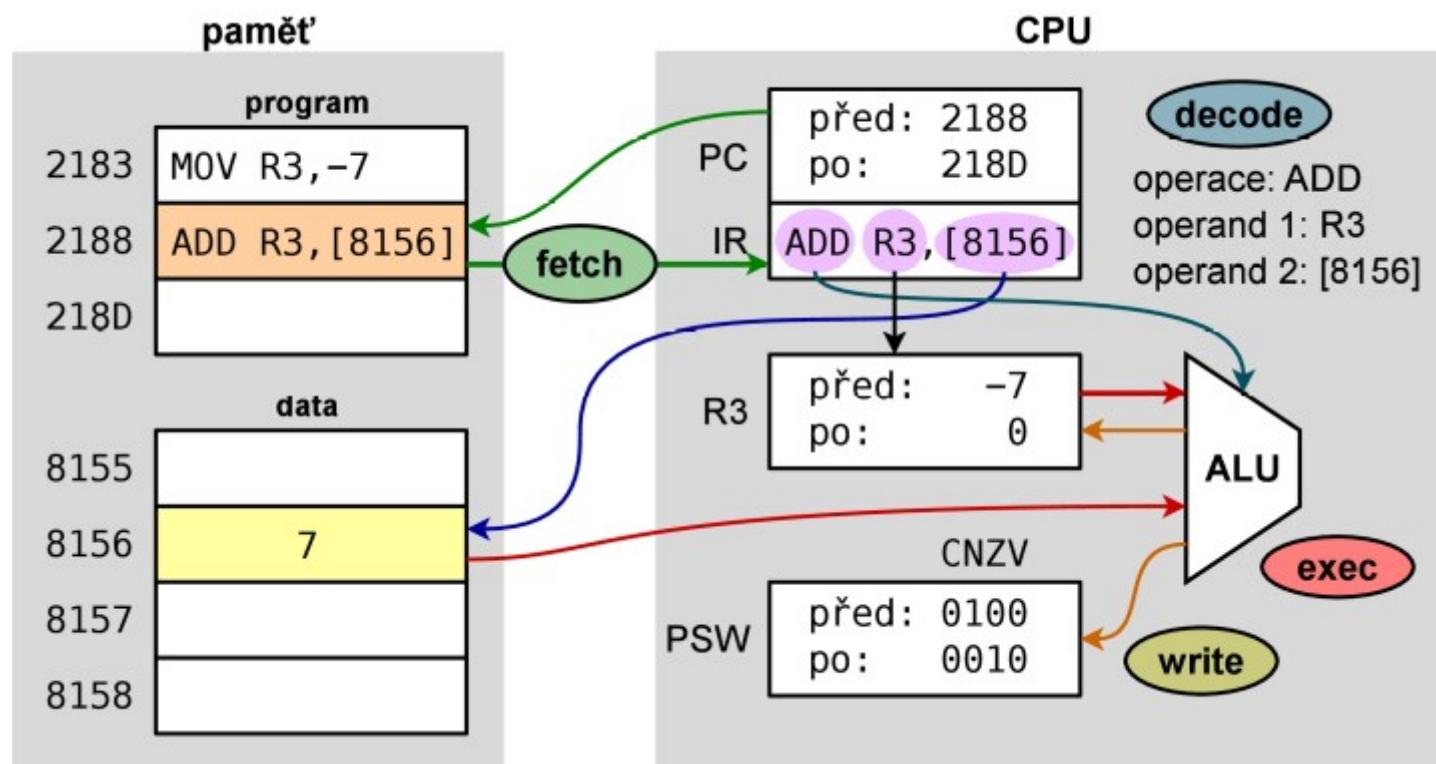
- CPU – Central Processing Unit
 - **registry**:
 - program counter, instruction register, stack pointer
 - PSW (program status word) – příznaky C, N, Z, V
 - C = carry, N = negative, Z = zero, V = overflow
 - ostatní registry (obecné, datové, adresní, privátní, ...)
 - zvyšování výkonu
 - pipeline, superskalární CPU (instrukční paralelismus)
 - spekulativní provádění instrukcí
 - hyperthreading (sdílení částí CPU mezi vlákny)
 - více jader

Pipeline (obrázek)



- riscová pipeline:
 1. fetch – načtení
 2. decode – dekódování
 3. execute – provedení
 4. memory access – čtení paměti
 5. write back – zápis výsledků do registrů

Provedení instrukce (obrázek)



Režimy CPU

- obvykle aspoň dva režimy CPU
 - **user** – není povoleno vše
 - **kernel** – privilegovaný režim (např. přístup k HW)
 - architektura x86 má čtyři režimy: ring 0 – ring 3
 - Vanderpool (VT) / Pacifica (AMD-V) – nový ring -1
- instrukce **TRAP**
 - skok z režimu user do režimu kernel
 - systémové volání
 - ošetření výjimek

Paměť

- registry procesoru, CPU cache, operační paměť
- disk, pánska, CD, DVD, EEPROM, flash RAM
- ochrana operační paměti – změny: režim **kernel**
 - paměti procesů navzájem
 - jádro × procesy
- **relokace**
 - zavedení procesu na libovolnou adresu
 - virtuální adresa (CPU) × fyzická adresa (RAM)
 - **MMU** (Memory Management Unit) provádí převod

Cache

- rychlá paměť – obvykle drahá
 - cache – rychlá mezipaměť (např. mezi CPU a RAM)
 - obecně paměť mezi rychlým a pomalejším zařízením
- využívá **principu lokality odkazů** v paměti
 - tendence odkazovat se do omezené oblasti paměti
 - činitel úspěšnosti (Hit Ratio) se pak bude blížit jedné i při malé kapacitě paměti cache
- střední přístupová doba: $T_s = T_c + (1 - HR) \cdot T_{OP}$ ($T_c \ll T_{OP}$)
 - HR blízko 1 → přístup je blízký přístupu do cache

Vstupně-výstupní zařízení

- zařízení, řadič (řídicí jednotka, controller)
- OS zjednodušuje práci s V/V zařízeními
 - ovladače zařízení pro OS
 - zařazení přímo do jádra OS
 - načtení ovladačů při spuštění systému
 - načtení ovladačů za běhu systému – USB, IEEE 1394
- registry na zařízení, **V/V porty**, přerušení
- přístup pomocí instrukcí CPU
- **DMA** (Direct Memory Access)

Přístup k V/V zařízení

- přístup pomocí instrukcí CPU
 - zápis na V/V, kontrola stavu, čekání – neproduktivní
- přístup s využitím přerušení
 - zápis na V/V, provádění jiných operací – produktivní
 - po dokončení: přerušení – pokračování zápisu
- **DMA** (Direct Memory Access) a **IRQ**
 - zápis adresy a rozsahu dat v RAM do V/V, příkaz
 - provádění jiných operací – produktivní
 - po dokončení: přerušení a obsluha

Přerušení (interrupt, IRQ)

- V/V přerušení
 - dokončení operace, chybový stav
- časovač – důležitý pro preemci
 - generuje přerušení v daných intervalech
- chyby procesu, výjimky
 - pokus o přístup do zakázané oblasti paměti,
neplatná instrukce (privilegovaná v režimu user)
- chyby HW
 - chyba parity paměti, výpadek napájení

Průběh zpracování přerušení

- CPU provádí proces a HW vygeneruje IRQ (interrupt request = požadavek na přerušení)
 - **CPU**: dokončení rozpracované instrukce
 - **CPU**: skok na obslužnou rutinu přerušení
 - před tím se uloží adresa návratu (na systémový stack)
 - **rutina**: uloží se kontext (registry z CPU)
 - do tabulky procesů (nebo na systémový stack)
 - **rutina**: obsluha přerušení (IRQ časovače: plánovač)
 - **rutina**: vrátí se kontext (do CPU), návrat do procesu
 - plánovač může naplánovat také jiný proces

Sběrnice

- komunikace částí počítače mezi sebou
 - interní, lokální: CPU, FSB – paměť
 - interní: ISA, PCI, AGP, PCI-E, SCSI, IDE, EIDE, ATAPI, ATA, Ultra ATA, SATA
 - externí: PCMCIA, PC Card, ExpressCard, SCSI, eSATA, IEEE 1394, USB, ...
- OS spravuje zařízení připojená na sběrnice
 - plug-and-play
 - BIOS (Basic Input Output System)

Shrnutí – požadavky OS na HW

- **přerušovací systém**
 - umožní efektivní využití CPU, nutné pro DMA
- **časovač** – pravidelně generované přerušení
 - umožní preempcí
- CPU s podporou režimů **kernel** a **user**
 - **ochrana** paměti – měnit přístup jen v režimu kernel
 - neprivilegovaná operace v režimu user – TRAP
- CPU s podporou **virtualizace paměti, MMU**
 - **logické adresování** – umožnění relokace

OS – Procesy a vlákna

Tomáš Hudec

Tomas.Hudec@upce.cz

<http://fei-as.upceucebny.cz/usr/hudec/vyuka/os/>

Osnova

- procesy
 - příkazy pro procesy
 - procesy – systémová volání
- signály
 - signály – systémová volání
- vlákna
 - vlákna – posixová volání

Proces

- instance programu v paměti systému
 - **program** = recept (na dort)
 - **proces** = pečení (příprava dortu dle tohoto receptu)
- provádění: sekvenční × multiprogramming
- **tabulka procesů**, PCB (Process Control Block)
 - identifikace procesu, adresový prostor, stav, přidělené prostředky, práva, čas běhu, ...
- hierarchie procesů
 - vztah rodič–potomek, strom

Vznik procesu

- původcem je jádro
 - při inicializaci systému
 - první proces (v posixových systémech obvykle **init**)
 - služby jádra (mikrojádrový OS)
- původcem je proces
 - systémové volání
 - uživatel zadá příkaz, který interpret (shell) zpracuje tak, že systémovým voláním vytvoří další proces
 - proces může být aktivován také stiskem tlačítka (dávkové a vestavěné systémy)

Zánik procesu

- dobrovolné ukončení (systémovým voláním)
 - normální – úloha byla dokončena
 - při detekování chyby – např. chybné vstupy
- nedobrovolné ukončení
 - fatální (neošetřená) chyba
 - př.: neplatná adresa, nepovolená instrukce, dělení nulou
 - jádro proces ukončí
 - zabití jiným procesem (uživatelem)

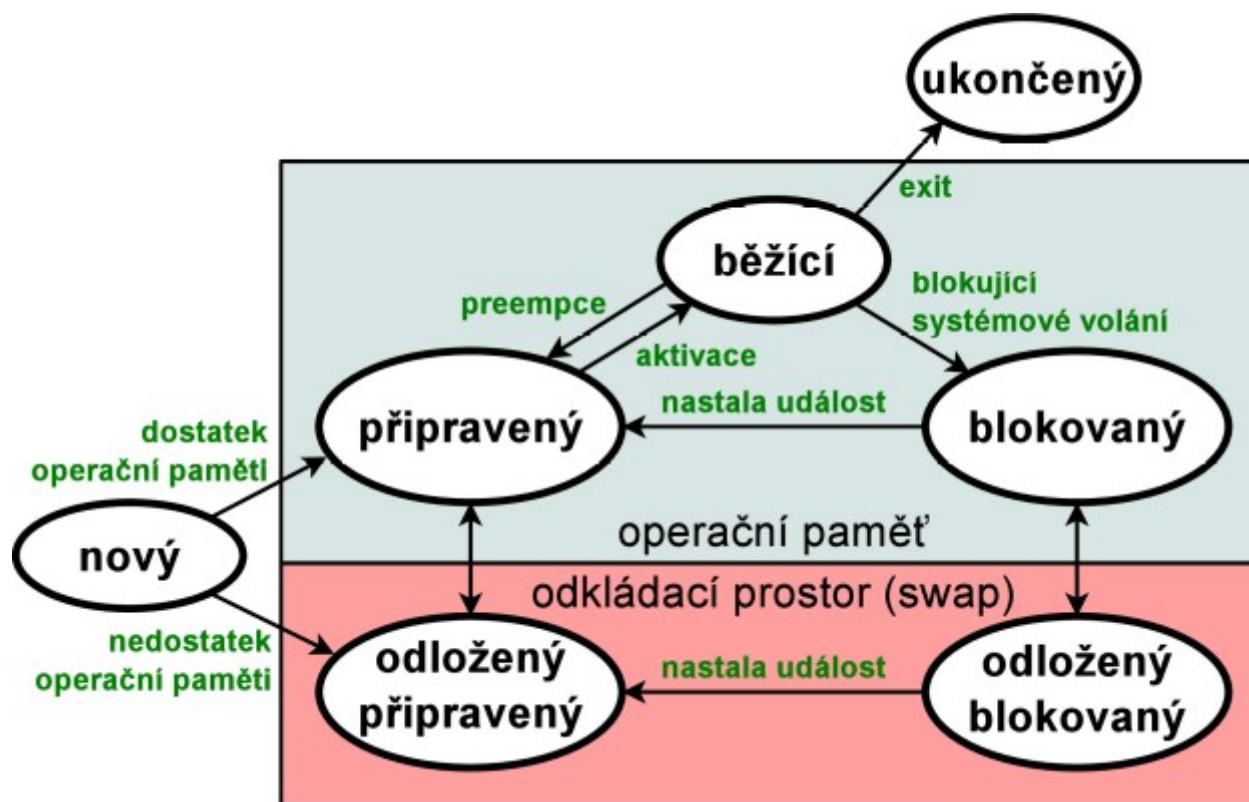
Stavy procesů

- třístavový model:
 - běžící (running) – používá CPU
 - připravený (ready) – pozastaven jádrem OS
 - blokovaný (blocked) – čekající na vnější událost
- scheduler – plánovač
 - vybírá připravený proces pro běh – přiděluje CPU
 - odebírá CPU běžícím procesům – preempce
 - může se aktivovat také při systémových voláních
 - řídí se plánovacím algoritmem

Rozšířené stavy procesů

- základní stavy lze rozšířit – 7stavový model
 - nový (new)
 - nelze zatím spustit (nemá ještě všechny prostředky)
 - ukončený (exit)
 - již se nemůže spustit, ale je třeba ještě držet v paměti jeho informace, např. kvůli účtování (accounting)
 - odložený blokovaný (blocked, suspended)
 - blokovaný proces zabírá paměť, více takových procesů pak ubírá paměť běžícím, proto se proces z paměti odloží na disk (swap)
 - odložený připravený (ready, suspended)
 - nastala již událost, na niž blokovaný proces čekal, ale proces je stále ještě na disku

Stavy procesů (obrázek)



sedmistavový model

Implementace procesů

- tabulka procesů nebo též PCB
 - adresový prostor: kód (text), stack, data, heap
 - přidělené prostředky: otevřené soubory, semafory, ...
 - kontext (stav): registry CPU, mapování paměti
 - atributy: id, údaje plánovače, práva, časy, účtování, ...
- při přerušení (např. při V/V) – přepnutí kontextu
 - uložení kontextu (stavu) procesu
 - obsluha ovladačem v jádře
 - plánovač rozhodne, který proces poběží poté

Manuálové stránky

- rozděleny do sekcí, uloženy v /usr/share/man
sekce popis
 - 1 základní uživatelské příkazy
 - 2 služby jádra – systémová volání
 - 3 knihovní funkce
 - 5 formáty souborů, protokoly, struktury v C
 - 7 různé (protokoly, normy apod.)
 - 8 příkazy pro správu systému
- příkaz man(1) – číslo v závorce udává sekci
 - např. pro příkaz kill(1) a systémové volání kill(2):
 - **man kill**
 - **man 2 kill**

Příkazy pro procesy (UNIX) (1)

- seznam procesů: ps(1), pstree(1)
 - všechny procesy: **ps -e [-f | -l]**
- sledování zátěže: top(1), prstat(1)
- sledování délky fronty procesů: xload(1)
 - nebo uptime(1), také **cat /proc/loadavg**
- sledování využití procesoru: mpstat(1)
 - např. 10 měření po 1 sekundě: **mpstat 1 10**

Příkazy pro procesy (UNIX) (2)

- získání PID podle jména: pidof(1)
- nalezení podle kritérií: pgrep(1)
- poslání signálu: kill(1)
- poslání signálu podle jména: killall(1)
- poslání signálu dle kritérií: pkill(1)
- nastavení priority: nice(1), renice(1)
 - nice: od -20 (maximální priorita) do 19 (min.)

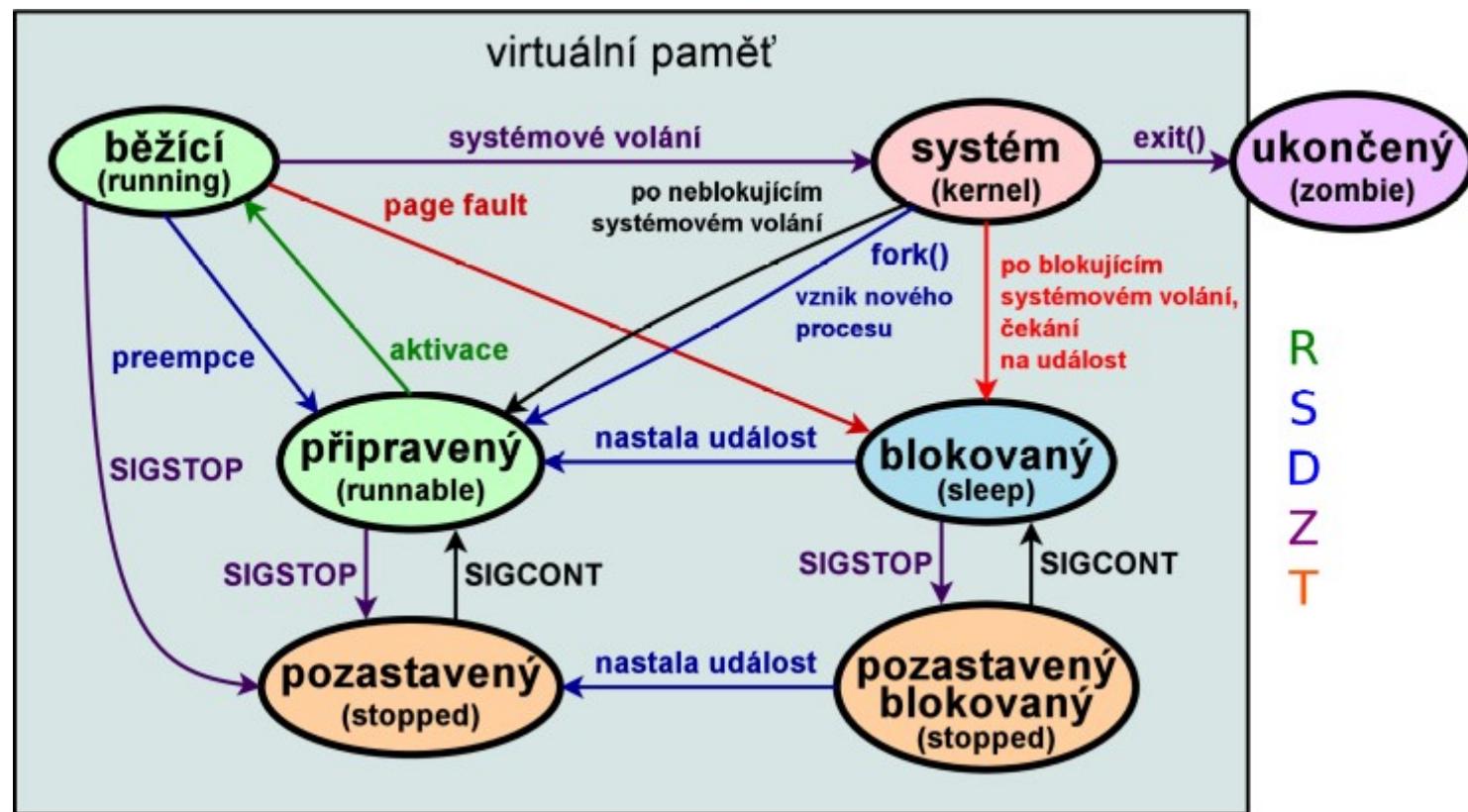
Atributy procesů podle ps (UNIX)

označení	id	popis
PID	pid	identifikace procesu
%CPU	%cpu	využití CPU
%MEM	%mem	využití paměti
CMD	args	příkaz s argumenty
START	bsdstart	čas vzniku procesu
TIME	bsdtime	celkový čas využití CPU – MMM:SS
TIME	cputime	celkový čas využití CPU – [DD-]HH:MM:SS
ELAPSED	etime	celkový čas běhu procesu – [DD-]HH:MM:SS
USER	user	efektivní uživatel (UID), aliasy euser, uname
GROUP	group	efektivní skupina (GID), alias egroup
RUSER	ruser	reálný uživatel (UID)
RGROUP	rgroup	reálná skupina (GID)
NI	nice	hodnota priority nice
STAT	stat	stavy: D, R, S, T, Z, X; atributy: <, N, L, s, I, +
SZ	sz	velikost ve fyzických stránkách (text, data, stack)
VSZ	vsized	velikost virtuální paměti v KiB

Stavy procesů (Linux)

- stavy procesů – jak je vypisuje ps(1):
 - R – připravený nebo běžící (runnable / running)
 - S – blokovaný (sleep)
 - Z – ukončený (defunct, zombie)
 - T – pozastavený nebo krokovaný (stopped, traced)
 - „preempce uživatelem“
 - D – nepřerušitelný spánek (uninterruptible sleep)
 - proces zavolal systémové volání (blokující, proto sleep), které nelze přerušit (např. signálem)

Stavy procesů (Linux) (obrázek)



stavový model procesu v Linuxu (a UNIXu)

Procesy – systémová volání

- fork(2) – vytvoření procesu
- exec(3), execve(2) – nahrazení kódu procesu
- exit(3), _exit(2) – ukončení procesu
- wait(2), waitpid(2) – čekání na změnu potomka
 - na ukončení, případně pozastavení, obnovení běhu
- getpid(2), getppid(2) – zjištění (P)PID
- kill(2), raise(3) – zaslání signálu / zabítí procesu
- signal(2), sigaction(2) – obsluha signálu

Procesy – Win32 API

- CreateProcess() – vytvoření procesu
- ExitProcess() – ukončení procesu
- WaitForSingleObject(),
WaitForMultipleObjects(),
GetExitCodeProcess() – čekání na událost
(ukončení procesu), zjištění návratového kódu
- GetCurrentProcessID() – zjištění PID
- TerminateProcess() – zabítí procesu

Procesy – vytvoření, nahrazení

- vytvoření procesu – **fork(2)**

```
#include <unistd.h>  
pid_t fork(void);
```

– vrací: 0 = potomek, 0 < PID = rodič, -1 = chyba

- nahrazení procesu – **execve(2)**

```
#include <unistd.h>  
  
int execve(const char *filename,  
           char *const argv[], char *const envp[]);
```

– vrací: úspěch = bez návratu, -1 = chyba

- viz též knihovní funkce system(3), exec(3)

Procesy – ukončení

- ukončení – `_exit(2)`

```
#include <unistd.h>
```

```
void _exit(int status);
```

– ukončí proces okamžitě, raději tedy: `exit(3)`

```
#include <stdlib.h>
```

```
void exit(int status);
```

– návratovou hodnotu může rodič získat voláním `wait(2)` nebo `waitpid(2)`

Procesy – čekání na ukončení

- čekání na změnu stavu – **wait(2)**
 - změny: ukončení, signál pozastavení, signál pokračování

```
#include <sys/types.h>
```

```
#include <sys/wait.h>
```

```
pid_t wait(int *status);
```

- ekvivalentní: **waitpid(-1, &status, 0)**;

```
pid_t waitpid(pid_t pid, int *status,  
int options);
```

- Příklad – kompletní je např. v manuálové stránce **wait(2)**:

```
w = waitpid(cpid, &status, WUNTRACED | WCONTINUED);  
if (w == -1) { perror("waitpid"); exit(EXIT_FAILURE); }  
if (WIFEXITED(status))  
    printf("exited, status=%d\n", WEXITSTATUS(status));
```

Procesy – spuštění příkazu, získání PID a PPID

- spuštění příkazu v shellu – system(3)

```
#include <stdlib.h>  
  
int system(const char *command);
```

- spustí příkaz v shellu, čeká na dokončení
- vrací: 127 = příkaz nenalezen, jinak EC příkazu

- zjištění PID, PPID (parent PID) – getpid(2)

```
#include <sys/types.h>  
  
#include <unistd.h>  
  
pid_t getpid(void);  
pid_t getppid(void);
```

Signály

- jednoduché zprávy – signal(7)
 - SW obdoba HW přerušení
- pouze posixové systémy
- implicitně je posílá OS při určitých událostech
- explicitně se posílají příkazem kill(1)
- zpracování:
 - ukončení (s eventuálním coredump), pozastavení, pokračování procesu, ignorování

Seznam (některých) signálů

č.	signál	akce	popis
1	SIGHUP	exit	zavěšení, ukončení session
2	SIGINT	exit	ukončení CTRL+C
3	SIGQUIT	core	ukončení CTRL+\
6	SIGABRT	core	ukončení
15	SIGTERM	exit	ukončení
9	SIGKILL	exit	ukončení, nelze zachytit ani ignorovat
	SIGTSTP	stop	pozastavení CTRL+Z
	SIGSTOP	stop	pozastavení, nelze zachytit ani ignorovat
	SIGCONT	resume	pokračování pozastaveného procesu
11	SIGSEGV	core	porušení ochrany paměti
	SIGCHLD	ignore	potomek skončil
14	SIGALRM	exit	alarm
4	SIGILL	core	neplatná instrukce
5	SIGTRAP	core	krokování, ladění
8	SIGFPE	core	výjimka plovoucí řádové čárky

Signály – poslání signálu, obsluha

- poslání signálu procesu – kill(2)

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

int kill(pid_t pid, int sig);
```

– vrací: 0 = úspěch, -1 = chyba

- nastavení obsluhy signálu (**zastaralé, zavrženo**)

```
#include <signal.h>

typedef void (*sighandler_t)(int);
sighandler_t signal(int signum,
                     sighandler_t handler);
```

Signály – nedostatky nastavení obsluhy voláním signal(2)

- signal(2) je **zastaralé** nastavení obslužné rutiny pro zpracování signálu a trpí nedostatky
 - nastavení obsluhy je obvykle pouze jednorázové
 - nelze 100% zaručit opětovné zavolání obslužné rutiny při přijetí dvou signálů rychle po sobě
 - neimplementuje automatické maskování signálů
 - nedostatečně standardizované
 - nedostatečně použitelné chování ve vláknech
 - existují rozdílné implementace

Signály – nastavení obsluhy

- změna obsluhy signálu – `sigaction(2)`

```
#include <signal.h>

int sigaction(int signum, const struct
              sigaction *act, struct sigaction *oldact);

– vrací: 0 = úspěch, -1 = chyba

struct sigaction {
    void      (*sa_handler)(int);    // obsluha, SIG_DFL, SIG_IGN
    void      (*sa_sigaction)(int, siginfo_t *, void *);
    sigset_t sa_mask;    // maska blokovaných signálů při obsluze
    int       sa_flags;   // nastavení chování
    void      (*sa_restorer)(void); // zastaralé, nepoužívat
};
```

Signály – maskování, množiny

- blokování signálů, nastavení množin signálů

```
#include <signal.h>

int sigprocmask(int how, const sigset_t *set,
                 sigset_t *oldset);

- how: SIG_BLOCK, SIG_UNBLOCK, SIG_SETMASK

int sigemptyset(sigset_t *set);
int sigfillset(sigset_t *set);
int sigaddset(sigset_t *set, int signum);
int sigdelset(sigset_t *set, int signum);
- vrací: 0 = úspěch, -1 = chyba

int sigismember(const sigset_t *set, int signum);
- vrací: 1 nebo 0 = úspěch, -1 = chyba
```

Signály – nahrazení zastaralého volání signal(2) voláním sigaction(2)

obsluha signálu pomocí zastaralého volání signal(2)

```
void handler(int signo) {                                // obslužná rutina
    sigset_t mask;                                     // maska pro signály
    signal(signo, handler);                            // znovunastavení obslužné rutiny
    sigfillset(&mask);                                // naplnění množiny signálů
    sigprocmask(SIG_SETMASK, &mask, NULL); // nastavení masky
    // obsluha signálu
}
signal(SIGINT, handler);                                // (jednorázové) nastavení obslužné rutiny
```

obsluha signálu pomocí sigaction(2)

```
void handler(int signo) {                                // obslužná rutina
    // obsluha signálu
}
struct sigaction sa, old_sa;                            // struktura pro obsluhu signálu:
sa.sa_handler = handler;                             // název obslužné rutiny
sa.sa_flags = SA_RESTART;                            // příznaky: restart přerušeného syst. volání
sigfillset(&sa.sa_mask);                           // maska
sigaction(SIGINT, &sa, &old_sa); // (trvalé) nastavení obslužné rutiny
```

Signály – alarm, čekání na signál

- nastavení poslání upozornění – alarm(2)

```
#include <unistd.h>
```

```
unsigned int alarm(unsigned int seconds);
```

– vrací: počet sekund zbývajících do alarmu, který byl nastaven předchozím voláním (0 = nebyl)

- čekání na konkrétní signály – sigsuspend(2)

```
#include <signal.h>
```

```
int sigsuspend(const sigset_t *mask);
```

– vrací: vždy -1 (s chybou **EINTR**)

Signály – nastavení časovače

- nastavení upozornění po čase – setitimer(2)

```
#include <sys/time.h>
```

```
int getitimer(int which, struct itimerval *val);  
int setitimer(int which, const struct itimerval  
*value, struct itimerval *ovalue);
```

- tři časovače (hodnota which):

- ITIMER_REAL – **SIGALRM**, měří reálný čas
- ITIMER_VIRTUAL – **SIGVTALRM**, měří čas CPU procesu
- ITIMER_PROF – **SIGPROF**, čas CPU procesu + jádra

- vrací: 0 = úspěch, -1 = chyba

Vlákna

- **proces** – související prostředky jako celek
 - adresní prostor, environment, pracovní adresář, otevřené soubory, obsluha signálů, nástroje IPC (semafory, sockety), účtování (accounting), ...
- **vlákno** – „odlehčený proces“ – samostatně pouze:
 - **stack** (lokální data podprogramů) a
 - **plánovací položky**, tj. **stav** (připraveno / blokováno / běží), **kontext** (uložené registry CPU), priorita apod.
 - ostatní je sdíleno s ostatními vlákny procesu

Vlákna – motivace

- kvaziparalelismus – stejný motiv jako proces
- jednodušší a rychlejší správa (vznik, ...)
 - `pthread_create(3)` až 50× rychlejší než `fork(2)`
- výkon – záleží na aplikaci ($CPU \times V/V$)
- určitě užitečné na SMP
- příklady využití:
 - textový procesor (vstup, V/V, formátování)
 - web-server (sítové spojení, předání stránky)

Vlákna × procesy

- blokující systémová volání (SV) → snadnější programování
- paralelismus, nižší režie → zvýšení výkonu
- příklad web-serveru

	typ SV	snadné programování	paralelismus	režie
jeden proces	blokující	ano	ne	nízká
jeden proces	neblokující	spíše ne	ano, 1 CPU	nízká
skupina procesů	blokující	ano	ano	vysoká
vlákna	blokující	ano	ano	nízká

Implementace vláken

- implementace vláken bez podpory OS
 - pomocí knihovných funkcí – problémy:
 - blokující volání převést na neblokující
 - page-fault – stránka není v operační paměti
 - je třeba plánovač vláken – obvykle pracné
- implementace v jádře OS
 - vzdálená volání – více režie
 - není potřeba neblokujících volání
- hybridní implementace (např. Solaris)

Výhody a nevýhody implementací

- implementace bez podpory jádra OS
 - + lepší režie – rychlejší vznik, přepnutí kontextu
 - + nevyžaduje se přechod do režimu jádra
 - + strategie plánovače se dá přizpůsobit aplikaci
 - složitá implementace (neblokující volání, plánovač)
 - page-fault zastaví všechna vlákna

Výhody a nevýhody implementací

- implementace v jádře OS
 - + lze provádět i vlákno procesu, jehož jiné vlákno způsobilo page-fault
 - + není třeba neblokujících volání
 - horší režie – přechod do režimu jádra
 - pevná strategie plánovače vláken

Problémy při používání vláken

- globální proměnné
 - nutné samostatné alokacé pro každé vlákno (`errno`)
 - přístup ke sdíleným proměnným (kritické sekce)
- nereentrantní volání některých knih. funkcí
 - např. alokace paměti (`malloc`)
- znalost implementace signálů a jejich obsluhy
 - které vlákno má dostat signál, které se má přerušit
- stack (automatické zvětšení při přetečení?)

Posixová vlákna

- posixová knihovna pthread.h(7)

```
#include <pthread.h>
```

- POSIX.1c – redefinice globální proměnné `errno`

```
#include <errno.h>
```

- špatně: ~~extern int errno;~~

- komplikace se symbolem `_REENTRANT`

```
gcc -D _REENTRANT prog.c -lpthread -o prog
```

- zajistí reentrantnost funkcí
- zajistí přenositelnost

Vlákna POSIX – vytvoření

- vytvoření vlákna – `pthread_create(3)`

```
int pthread_create(pthread_t *restrict thread,  
                  const pthread_attr_t *restrict attr,  
                  void *(*start_routine)(void*),  
                  void *restrict arg);
```

- spustí nové vlákno – funkci `start_routine(arg)`
- `thread` – identifikace vlákna
- `attr` – nastavení atributů, může být `NULL`
- vrací: 0 = úspěch, jinak číslo chyby

Vlákna POSIX – ukončení

- ukončení vlákna – `pthread_exit(3)`

```
void pthread_exit(void *value_ptr);
```

– volá se implicitně při ukončení funkce (`return`)

- čekání na ukončení vlákna – `pthread_join(3)`

```
void pthread_join(pthread_t thread,
                  void **value_ptr);
```

– `value_ptr` – návratová hodnota z `pthread_exit`

- může být `NULL`, pokud ji nepotřebujeme

Vlákna POSIX – odpojení

- odpojení vlákna – `pthread_detach(3)`

```
int pthread_detach(pthread_t thread);
```

- nastaví automatické uvolnění zdrojů vlákna
- na vlákno pak nelze čekat pomocí `pthread_join`
 - nelze tedy získat návratovou hodnotu
 - informace lze ale předat pomocí globální proměnné
- lze též nastavit atributem
- vrací: 0 = úspěch, jinak číslo chyby

Vlákna POSIX – zrušení

- zrušení vlákna – `pthread_cancel(3)`

```
int pthread_cancel(pthread_t thread);
```

- ukončí vlákno `thread`
- vlákno může registrovat ukončovací funkce
 - `pthread_cleanup_push(3)`
 - zavolají se před ukončením vlákna
- typ ukončení: asynchronní (ihned) nebo odložené
 - `pthread_setcanceltype(3)`
- vrací: 0 = úspěch, jinak číslo chyby

Vlákna POSIX – id, atributy

- získání identifikace vlákna – `pthread_self(3)`

```
pthread_t pthread_self(void);
```

- porovnání vláken – `pthread_equal(3)`

```
int pthread_equal(pthread_t t1, pthread_t t2);
```

– vrací: $0 = t_1 \text{ a } t_2$ jsou různá vlákna, $0 \neq$ stejná

- manipulace s atributy vláken

`pthread_attr_destroy(3), pthread_attr_getdetachstate(3),
pthread_attr_getstackaddr(3), pthread_attr_getstack(3),
pthread_attr_getstacksize(3), pthread_attr_getschedpolicy(3),
pthread_attr_getschedparam(3), pthread_attr_getscope(3),
pthread_attr_getinheritsched(3)`

Vlákna – Win32 API

- CreateThread() – vytvoření vlákna
- ThreadExit() – ukončení vlákna
- WaitForSingleObject(),
WaitForMultipleObjects() – čekání na událost
(ukončení vlákna)
- SetPriorityClass(), SetThreadPriority() –
manipulace s plánovacími atributy

OS – Plánování procesů

Tomáš Hudec

Tomas.Hudec@upce.cz

<http://fei-as.upce.cz/usr/hudec/vyuka/os/>

Plánování – scheduling

- **scheduler** – plánovač
 - rozhoduje, který proces (vlákno) má CPU
 - řídí se plánovacím algoritmem
- historie:
 - dávkové systémy – jediná fronta, jediná úloha
 - potřeba multiprogramování, neboť 80 % času bylo CPU nevyužito – zdržování na V/V
 - více paměti – multiprogramming, timesharing
 - potřeba naplánovat běh procesů optimálně

Typy plánování (dle času)

- dlouhodobé (long term)
 - které nové úlohy se mají zpracovávat, které ukončit
 - rozhoduje o množství procesů v systému
- střednědobé (medium term)
 - které procesy se mají odložit nebo vrátit do paměti
 - efektivní práce s omezenou operační pamětí
- krátkodobé (short term, dispatching)
 - který připravený proces dostane CPU, na jak dlouho

Cíle plánování obecně a podle typu OS

- spravedlnost – každý proces stejně času
- dodržování strategie (priorit)
- efektivní využití zdrojů, rovnováha zatížení
 - snaha využívat všechny části systému současně
- interaktivní systémy
 - minimalizace odezvy (response time)
 - čas mezi zadáním příkazu a odezvou
 - proporcionalita
 - vyhovět očekáváním uživatelů

Cíle plánování podle typu OS

- dávkové systémy
 - maximalizovat propustnost (throughput)
 - počet vykonaných úloh za jednotku času (h)
 - minimalizovat obrat (turnaround time)
 - průměrný čas na vykonání úlohy
 - využití CPU – využívat maximálně CPU
- systémy real-time
 - **respektování lhůt** – zabránění ztráty dat
 - **předvídatelnost** – zabránění degradace kvality
 - např. multimediální systémy

Režimy plánování

- nepreemptivní
 - proces se musí sám vzdát CPU (nebo blokovat)
- preemptivní
 - plánovač rozhoduje, kdy který proces má CPU
 - (efektivně) plánovat lze pouze v případě, že je k dispozici přerušovací systém a časovač
 - časovač „tiká“ typicky na frekvenci 100 Hz
 - přerušení nastává tedy každých 10 ms
 - plánování spotřebovává také čas CPU – režie

Typy procesů

- vstupně-výstupně orientovaný proces
 - většinu času čeká na dokončení operací V/V
 - typická je krátká výpočetní doba
 - časté používání blokujících systémových volání
 - typické pro interaktivní procesy
- výpočetně orientovaný proces
 - používá intenzivně procesor
 - blokující volání téměř nepoužívá

Plánovací algoritmy

- historie, dávkové systémy
 - fronta jednotlivých úloh (FIFO), víceúlohová FIFO
 - podle odhadu doby běhu úlohy
- moderní plánovací algoritmy
 - **round-robin** – spravedlivé střídání úloh
 - **prioritní** – dle důležitosti úlohy
 - **uživatelsky férové** – spravedlivé mezi uživateli
 - **termínové** – dodržení lhůt na systémech real-time

První přijde, první mele

- **First-Come First-Served** (fronta FIFO)
 - nepreemptivní
 - nové úlohy se zařadí do fronty
 - po ukončení aktuálního procesu se přidělí CPU procesu, který čekal ve frontě nejdéle
 - krátké procesy musejí zbytečně dlouho čekat
 - zvýhodňuje výpočtově orientované procesy
 - procesy bez V/V čekají pouze jednou
 - procesy s V/V čekají při každém dokončení operace

Nejkratší úloha první

- Shortest Job First
 - nepreemptivní
 - spustí se proces s nejkratší očekávanou dobou provádění
 - krátké procesy mají přednost
 - závislé na dobrém odhadu délky běhu procesu
 - hrozí vyhladovění dlouhodobých procesů

Nejkratší zbývající následuje

- Shortest Remaining Time Next
 - preemptivní varianta SJF
 - spustí se proces s nejkratší očekávanou dobou do dokončení
 - dále minimalizuje obrat (turnaround time)

Cyklická obsluha

- Round-Robin
 - preempce založená na časovači
 - každý proces dostane časové kvantum na CPU
 - přepnutí je prováděno při vypršení kvanta nebo při volání blokujícího systémového volání
 - je třeba optimalizovat délku kvanta
 - Příklad:
 - kvantum 4 ms, context switch 1 ms
 - CPU pracuje produktivně jen 80 % času
 - typické nastavení frekvence časovače je 100 Hz

Prioritní plánování

- priority based scheduling
- dává se přednost procesu s vyšší prioritou
- obvykle více front pro připravené procesy
- nízká priorita může mít za následek „vyhladovění“ procesu (starvation)
 - proces se již nedostane k CPU, „smrt hladem“
 - pro zabránění je třeba např. prioritu přizpůsobovat v závislosti na době čekání a historii běhu procesu

Plánování se zárukou

- **Guaranteed Scheduling – Fair-Share**
 - zaručuje každému uživateli stejné podmínky
 - n uživatelů na systému
 - každý dostane časové kvantum $1/n$
 - příklad:
 - uživatel A spustí 9 procesů
 - uživatel B spustí 1 proces
 - při RR má uživatel A 90 % času CPU, B pouze 10 %
 - při FS se využití CPU rozdělí mezi A a B na 50 %
 - B: jeden proces 50 % času CPU
 - A: devět procesů si rozdělí 50 % času, jeden má cca 5,56 %

Loteriové plánování

- **Lottery Scheduling**
 - každý proces dostane tiket(y) a periodicky se losuje
 - „výherní“ proces získá čas CPU
 - důležité procesy mohou mít více tiketů
 - procesy jsou si rovny, ale některé jsou si „rovnější“ (parafráze Orwell)
 - kooperativní procesy si mohou předávat tikety
 - lze použít jako approximace jiných algoritmů
 - snadná implementace

Plánování na systémech reálného času (real-time)

- pro úlohy reálného času je důležité **dokončení ve stanoveném termínu** (nikoliv rychlosť)
- jen periodické události:
 - systém je **plánovatelný**, je-li suma časů potřebných na obsloužení událostí dělená jejich periodami menší nebo rovna jedné
 - lze plánovat **staticky** (table-driven)
 - tabulky stanoví, kdy která úloha má být spuštěna
- aperiodické události: **dynamické** plánování

$$\sum_{i=1}^n \frac{C_i}{T_i} \leq 1$$

Plánování na systémech reálného času a preempce

- **preemptivní**
 - běžící proces může být přerušen
 - procesem s vyšší prioritou
 - procesem s bližším termínem dokončení
- **nepreemptivní**
 - běžící proces nesmí být přerušen
 - proces se musí vzdát procesoru sám
 - př.: proces musí v reálném čase bez přerušení komunikovat s externím zařízením

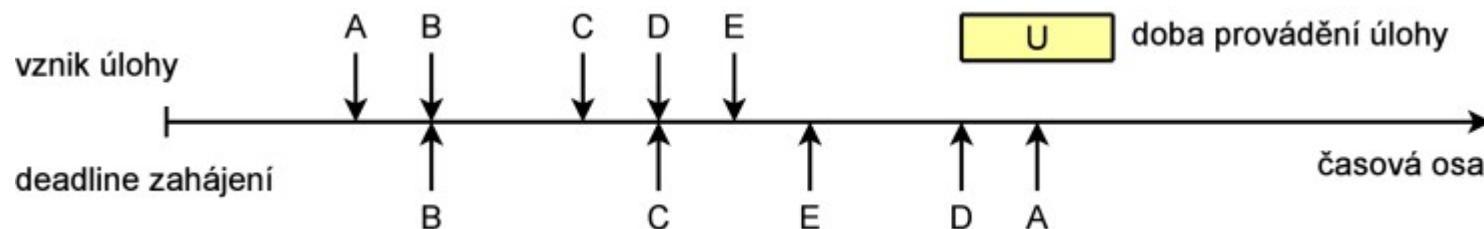
Termínové plánování – EDF (Earliest Deadline First)

- dokončení všech úloh ve stanoveném termínu
- ke spuštění vybírá úlohu s nejbližším termínem (deadline) zahájení / ukončení
- minimalizuje se podíl úloh, které nejsou dokončeny v požadovaném termínu
- na plánovatelných jednoprocesorových systémech s preempcí je optimální

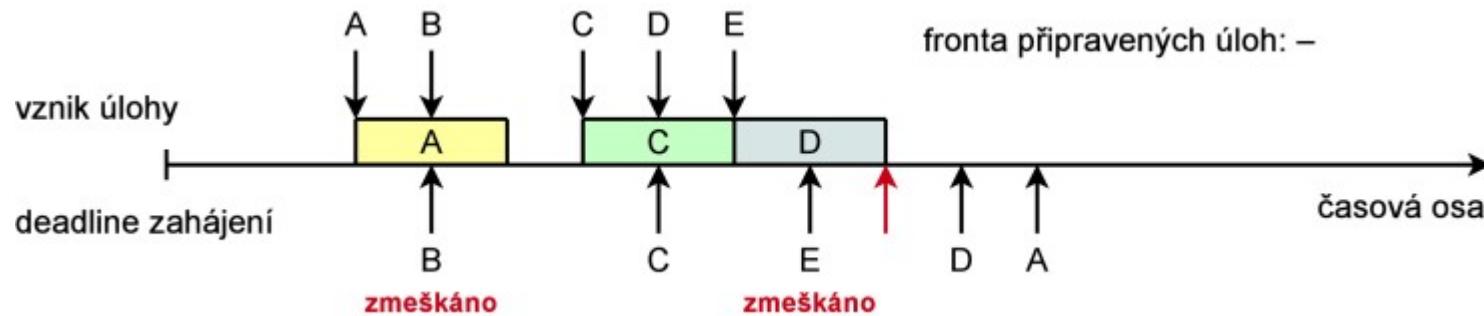
Nevýhody termínového plánování

- při přetížení systému není předvídatelné
 - skupina ovlivněných procesů je závislá na čase, kdy nastalo přetížení
- implementace v HW je náročná (přesnost)
 - termíny je třeba reprezentovat konečnými čísly
- je třeba znát termíny a dobu zpracování úloh
- kritické sekce: např. úlohy A, B, C (dle termínů)
 - C je v KS, A požaduje přístup do KS (blokuje)
 - plánuje se B, C neuvolní KS a zmešká se termín A

Plánování RT úloh – FIFO bez preempce (obrázek)

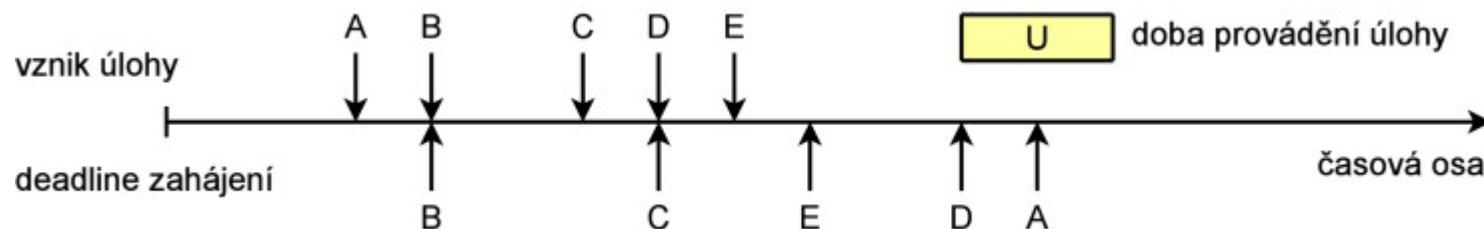


výchozí předpoklady

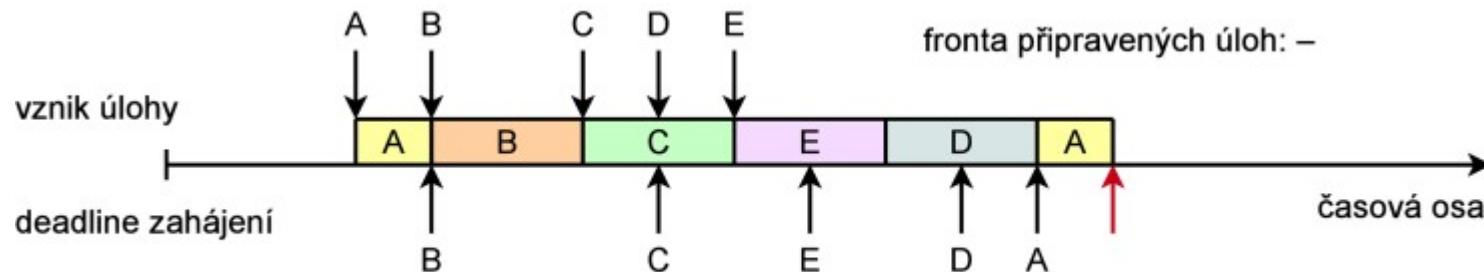


First-Come First-Served (nepreemptivní)

Plánování RT úloh – EDF s preempcí (obrázek)

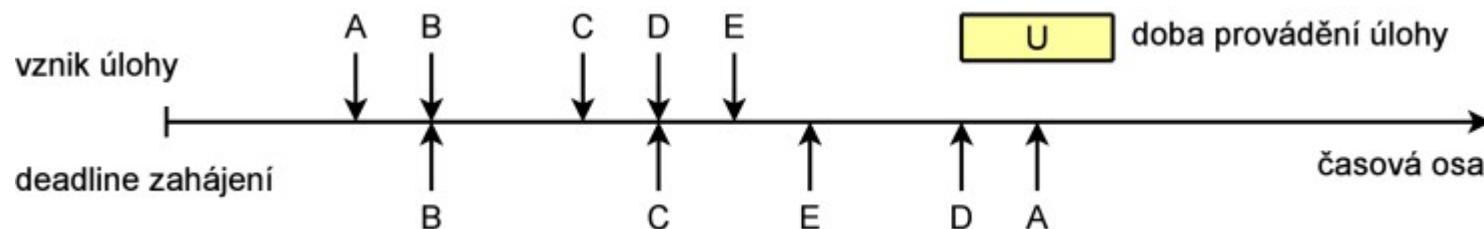


výchozí předpoklady

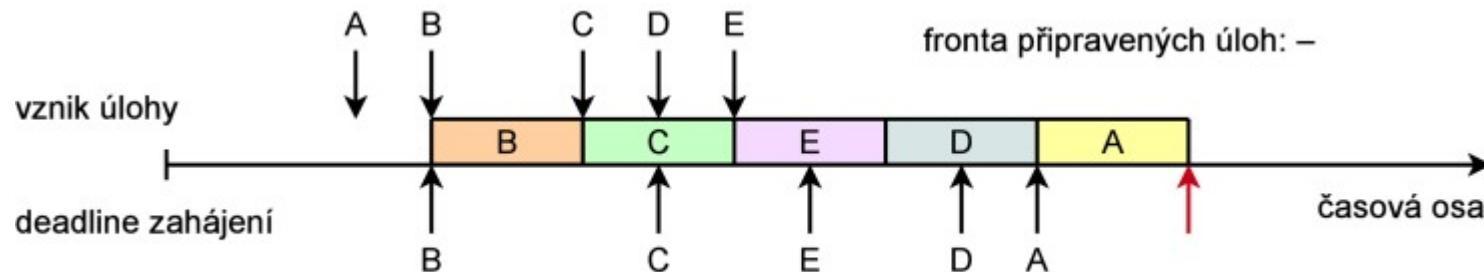


Earliest Deadline First (preemptivní)

Plánování RT úloh – EDF bez preempce (obrázek)



výchozí předpoklady

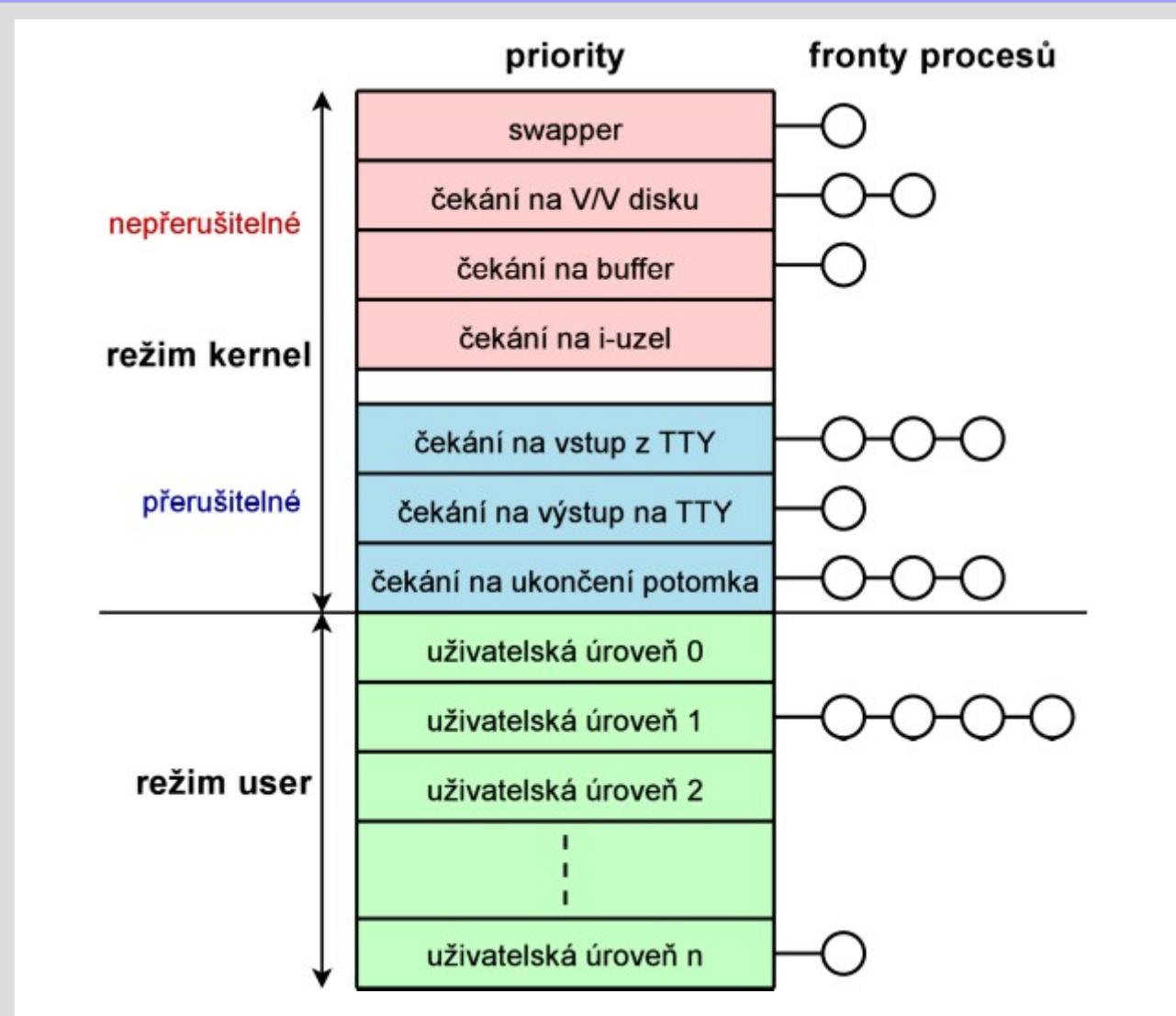


Earliest Deadline First (nepreemptivní)

Plánování v Unixu

- založeno na prioritním plánování
- priorita pro proces je dvojí
 - pro běh v režimu jádra
 - přiřazuje se procesu, když přechází do spícího stavu
 - pevná priorita podle typu systémového volání
 - přerušitelná a nepřerušitelná
 - pro běh v uživatelském režimu
 - nastaví se po návratu z režimu jádra
 - dynamická priorita

Plánování v Unixu (obrázek)



Plánování v Linuxu – přehled

- verze 1.2 (1995) – round-robin
- 2.2 (1999) – class based scheduling
 - priority, podpora SMP, jediný seznam úloh
- 2.4 (2001) – plánovač $O(n)$
 - čas CPU rozdělen do epoch, jediný seznam úloh
- 2.6 (2003) – plánovač $O(1)$
 - samostaná fronta pro každé CPU, dvě pole úloh
- 2.6.23 (2007) – **CFS** (Completely Fair Scheduler)

Dynamická priorita

- jádrem určený bonus dle historie běhu procesu
 - čekajícím se priorita zvyšuje, běžícím snižuje
- uživatelsky ovlivnitelná část – hodnota nice(2)
 - od -20 (maximální priorita) do 19 (minimální priorita)
 - výchozí hodnota je nula
 - hodnota říká, jak „milý“ (nice) je uživatel na ostatní
 - změna o ± 1 znamená zhruba $\pm 5\%$ času CPU
 - správce (root) smí hodnotu nastavovat libovolně
 - uživatel smí prioritu pouze snižovat (příp. obnovit)

Linux 2.4 – plánovač $O(n)$

- procesorový čas je rozdělen do epoch
 - každý proces má vypočítané časové kvantum
 - v rámci epochy je může využívat po částech
 - když všechny běhuschopné procesy vyčerpaly svá kvanta, epocha končí a začíná epocha nová
 - přepočítají se časová kvanta **VŠECH** procesů – $O(n)$
 - časové kvantum je dynamické
 - základní frekvence je $100 \cdot \text{HZ} / 1000$,
HZ = 100 Hz, tj. 10 tiků \approx 100ms kvantum
 - jediný seznam procesů pro všechny procesory

Linux 2.6 – plánovač $O(1)$

- založeno na **prioritách** – 140 úrovní
 - priorita real-time (0–99) má vždy přednost
- rozlišuje interaktivní procesy
 - podle průměrné doby čekání na CPU
- fronta procesů (***runqueue***) pro každé CPU
 - má dvě struktury: ***active*** a ***expired***, každá obsahuje
 - **seznam** procesů **pro každou prioritu** (140 front)
 - **bitová mapa** – neprázdnost seznamu pro každou prioritu
 - počet procesů

Linux 2.6 – plánovač $O(1)$ – preempce

- preempce (činnost plánovače):
 - právě přerušený proces:
 - (dynamická) priorita a časové kvantum jsou přepočítány
 - přesunut na konec příslušného seznamu (dle priority)
 - **interaktivní** a **real-time** zůstává v *active*, jinak do *expired*
 - aktivace procesu s nejvyšší prioritou v *active*
 - nejvyšší nastavený bit v bitmapě určí seznam – **$O(1)$**
 - využití instrukce typu *find-first-bit-set*
 - prázdné *active* → prohození s *expired*
 - neexistuje-li běhuschopný proces, HLT (čekání na IRQ)

Priorita a časové kvantum $O(1)$

- priorita p v jádře: 0–139 (0 je maximální priorita)
 - procesy real-time: $rt_priority > 0$ (99 je max. priorita)
 $p = 99 - rt_priority$ statická hodnota (0–98)
 - ostatní procesy: $rt_priority = 0$, dynamická 100–139
 $p = \max(100, \min(139, 120 + nice - b + 5))$ $b \in \{0, 10\}$
 - bonus b = (průměrná doba čekání na CPU v ms) / 100
 - proces je interaktivní, pokud $b - 5 \geq (120 + nice) / 4 - 28$
- základní časové kvantum pro $p \geq 100$
$$t = (140 - p) \cdot 20 \quad \text{pro } p < 120 \quad 420 - 800 \text{ ms}$$
$$t = (140 - p) \cdot 5 \quad \text{pro } p \geq 120 \quad 5 - 100 \text{ ms}$$

Priorita a plánovací třídy

- třídy plánovače – `sched_setscheduler(2)`, `chrt(1)`
 - procesy **real-time** (statická priorita 0–99)
 - absolutní přednost před procesy s dynamickou prioritou
 - třída `SCHED_FIFO` (nepreemptivní)
 - `SCHED_RR` (round-robin)
 - **ostatní** (dynamická priorita, základní je $120 + nice$)
 - čekání na CPU (blokování) zvyšuje prioritu
 - třída `SCHED_OTHER` (`SCHED_NORMAL`), později navíc
 - `SCHED_BATCH` (od v. 2.6.16, 2006)
 - `SCHED_IDLE` (od v. 2.6.23, 2007)

Plánovací třídy real-time

- **SCHED_FIFO** (nepreemptivní), **SCHED_RR**
- statická priorita – hodnota `rt_priority` > 0
- preempce pouze v těchto případech:
 - preempce procesem s vyšší prioritou
 - blokující systémové volání
 - zavolání `sched_yield(2)`
 - proces je vložen na konec fronty pro svou prioritu
 - **SCHED_RR** navíc po vypršení časového kvanta
 - `sched_rr_get_interval(2)`, typicky 100 ms

Ostatní plánovací třídy

- výchozí **SCHED_NORMAL**, **rt_priority = 0**
 - může běžet pouze tehdy, když neexistuje běhuschopný proces s prioritou real-time
- **SCHED_BATCH** (od jádra 2.6.16)
 - jádro vždy předpokládá, že proces je výpočetní
 - proces dostane malou penalizaci
 - vhodné pro neinteraktivní výpočetní procesy
- **SCHED_IDLE** (od 2.6.23, součást CFS)
 - nice nemá význam, větší penalizace než nice +19

Completely Fair Scheduler

- jádro 2.6.23 (2007) – autor Ingo Molnár
 - autorem konceptu Con Kolivas – RSDS (SD)
 - Rotating **Staircase Deadline** Scheduler
- zohledňuje odlišné požadavky systémů
 - desktopové – minimální odezva
 - serverové – maximální výkon
 - lze za běhu přepínat
- nepoužívá klasickou frontu procesů
 - fronty nahradil strom (**Red-Black Tree**) + váhy

CFS – Red-Black Tree

- lepší výkon než samovyvažovací stromy (AVL)
 - nejdelší cesta z kořene do listu není více než dvakrát delší než kterákoli jiná
 - nedokonale vyvážený strom, ale s nízkou režií
- klíčem je vážená **virtuální doba běhu** (VRT)
 - nízké hodnoty vlevo od kořene, vysoké vpravo
 - nejlevější úloha dostane CPU
 - uzlem může být skupina procesů (společná VRT)
 - př. procesy stejného uživatele

Plánovací třída deadline

- **SCHED_DEADLINE** od v. 3.14 (2014)
 - nejvyšší priorita (vyšší než real-time)
 - založeno na **EDF** (Eealiest Deadline First) a **CBS** (Constant Bandwidth Server)
 - podle lhůty dokončení (EDF) s podporou rezervací (CBS)
 - nelze nastavit, pokud by systém nebyl plánovatelný
 - parametry: **runtime** \leq **deadline** \leq **period** (v ns)
 - runtime – obvykle $>$ průměrná (nebo nejdelší) doba běhu
 - deadline – nejzazší doba dokončení (od začátku periody)

CFS + Real-Time + Deadline

- limit pro RT a DL: max. 95 % času CPU
 - brání vyhladovění ne-RT procesů
 - lze měnit v `/proc/sys/kernel/sched_*`
 - `sched_rt_runtime_us` – max. pro RT úlohy: 950 000 µs
 - `sched_rt_period_us` – 100 % času CPU: 1 000 000 µs
- runqueue pro každé CPU
 - DL: Red-Black Tree, klíč: termín dokončení
 - RT: pole 100 seznamů (front) pro jednotlivé priority
 - ostatní: Red-Black Tree, klíč: virtual-runtime

OS – Konkurence procesů a IPC

Tomáš Hudec

Tomas.Hudec@upce.cz

<http://fei-as.upceucebny.cz/usr/hudec/vyuka/os/>

Konkurence procesů (vláken) a problémy současného běhu

- prostředky poskytované systémem jsou **sdílené**
 - paměť, CPU, soubory, ...
 - procesy o prostředky soupeří – konkurence
 - vznikají **problémy souběhu (race conditions)**
 - provádění procesu může ovlivnit ostatní procesy
 - některé prostředky nelze (v jednom okamžiku) sdílet
 - přidělení prostředku procesu může omezit ostatní procesy
- při sdílení je třeba komunikace procesů (vláken)
 - synchronizace

Problémy souběhu

- **vzájemné vylučování** – mutual exclusion
 - v každém okamžiku smí mít přístup ke sdílenému prostředku pouze jeden proces
- **synchronizace**
 - proces čeká na dokončení operace jiného procesu
- **stav uváznutí** – deadlock, livelock
 - procesy vzájemně čekají na uvolnění prostředků
- **vyhladovění** – starvation
 - nekončící čekání procesu na získání prostředku

Přístup ke sdíleným prostředkům

- bez řízení přístupu ke sdíleným prostředkům:
 - může dojít k porušení konzistence dat
 - výsledek akcí procesů může záviset na pořadí, v jakém dostanou procesy přidělen procesor
 - tomu je třeba zabránit
- přístup ke sdíleným prostředkům je třeba řídit
 - zajištění integrity – zavedení **kritické sekce**
 - v jednom okamžiku smí sdílená data měnit jeniný proces
 - OS: nástroje **IPC** (Inter-Process Communication)

Příklad se sdílenou proměnnou

- procesy P_1 a P_2 sdílejí znakovou proměnnou z
- provádějí stejnou funkci **echo**

```
echo () {  
    načti z;  
    vypiš z;  
}
```

P_1
načti z ;

vypiš z ;

přepnutí
kontextu

P_2
načti z ;
vypiš z ;

- procesy mohou být přerušeny kdykoliv
 - přeruší-li OS proces P_1 po provedení **načti z** ,
 - pak proces P_1 vypíše znak načtený procesem P_2 !

Příklad s tiskárnou

- procesy P_1 a P_2 potřebují tisknout
- provádějí operaci **tiskni**

```
tiskni(tiskárna t) {  
    tisk(t, řádek1);  
    tisk(t, řádek2);  
}
```

P_1
tisk ř1;

tisk ř2;

přepnutí
kontextu

P_2
tisk ř1;
tisk ř2;

- procesy mohou být přerušeny kdykoliv
 - přepne-li OS z procesu P_1 mezi tiskem řádků na P_2 ,
 - pak tiskárna vytiskne mix řádků obou procesů

Kritická sekce (KS)

- část kódu, kde proces manipuluje se sdíleným prostředkem a současná manipulace jiným procesem by vedla k problému (nekonzistenci)
- provádění tohoto kódu musí být **vzájemně výlučné**
 - v každém okamžiku smí být v kritické sekci (pro daný prostředek) pouze jedený proces
 - proces musí žádat o povolení vstupu do kritické sekce

Struktura programu s KS

- **vstupní sekce** (entry section)
 - implementace povolení vstupu do KS
- **kritická sekce** (critical section)
 - manipulace se sdílenými prostředky
- **výstupní sekce** (exit section)
 - implementace uvolnění KS pro jiné procesy
- **zbytková sekce** (remainder section)
 - zbytek kódu procesu

Předpoklady řešení vstupu do KS

- procesy se provádějí **nenulovou rychlostí**
- **žádné** předpoklady o relativní rychlosti procesů
- uvažujeme i víceprocesorové systémy
 - předpoklad: paměťové místo smí v jednom okamžiku zpřístupnit vždy pouze jedený procesor
- **žádné** předpoklady o prokládaném provádění
 - procesy se nemusí pravidelně střídat v běhu
- stačí specifikovat vstupní a výstupní sekci

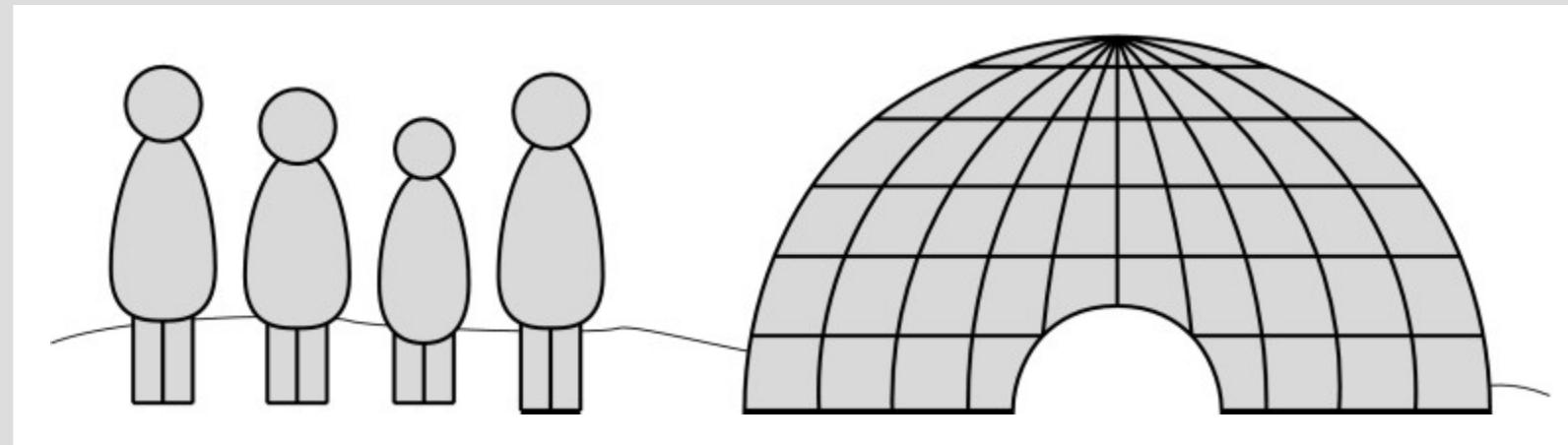
Požadované vlastnosti řešení KS

- vzájemné vylučování (mutual exclusion)
 - vždy jediný proces v KS
- pokrok v přidělování (progress)
 - na rozhodování o vstupu do volné KS se mohou podílet výhradně procesy, které nejsou ve ZS
 - toto rozhodnutí musí padnout v konečném čase
 - volná KS \Rightarrow požadavku musí být vyhověno
- omezené čekání (bounded waiting)
 - mezi požadavkem na vstup do KS a vyhověním smí do KS vstoupit pouze omezený počet jiných procesů

Typy řešení KS

- SW řešení
 - použití algoritmu pro vstupní a výstupní sekci
- HW řešení
 - využití speciálních instrukcí procesoru
- řešení OS
 - nabízí programátorovi prostředky pro řešení KS (datové typy a funkce)
- řešení programovacího jazyka
 - konkurenční / souběžné programování

SW řešení KS: aktivní čekání



Příklad:

- svého šamana (kritickou sekci) může v daném čase navštívit pouze jedený eskymák (proces)
- iglú má malý vchod, takže dovnitř může vstoupit vždy jen jeden jeden eskymák, aby si přečetl jméno napsané na tabuli
- je-li na tabuli napsané jeho jméno, může k šamanovi
- je-li na tabuli napsané jiné jméno, iglú opustí a čeká
- čas od času eskymák opět vstoupí do iglú podívat se na tabuli

SW řešení KS

- budou ukázány potenciální algoritmy snažící se o řízení vstupu do KS
- nejprve budou předpokládány pouze dva procesy
 - označení procesů: P_0 a P_1
- poté se správné řešení zobecní pro n procesů
 - označení: P_i aktuální proces a P_j ostatní procesy

SW řešení – algoritmus 1

- sdílená proměnná **locked**
 - udává obsazenost KS
 - inicializovaná na 0
- proces P_i , čeká, dokud je KS obsazena (**locked** \neq 0)
 - jakmile je KS volná, nastaví se **locked** = 1
- pokud oba procesy současně zjistí volnou KS, oba nastaví obsazeno a vstoupí do ní
 - **požadavek vzájemného vylučování není splněn!**

```
proces P,  
repeat  
    while  
        (locked != 0);  
    locked = 1;  
    KS;  
    locked = 0;  
    ZS;  
forever
```

SW řešení – algoritmus 2

- sdílená proměnná **turn**
 - inicializovaná na 0 nebo 1
- kritická sekce (KS) procesu P_i se provádí, jestliže **turn = i**
- proces P_i aktivně čeká (busy waiting), když je proces P_j v kritické sekci
 - požadavek vzájemného vylučování je splněn
- **požadavek pokroku není splněn!**
 - vyžadována alternace kritických sekcí

```
proces  $P_i$ 
repeat
    while (turn != i);
    KS;
    turn = j;
    ZS;
forever
```

SW řešení – algoritmus 2 – test

- předpokládejme, že P_0 má dlouhou zbytkovou sekci (ZS) a P_1 ji má krátkou
- pokud $\text{turn} = 0$, vstoupí P_0 do KS a pak provádí dlouhou ZS ($\text{turn} = 1$)
- zatím P_1 vstoupí do KS a provede pak krátkou ZS ($\text{turn} = 0$) a hned se znova pokusí vstoupit do KS – **požadavek je ale odmítnut!**
- proces P_1 musí čekat, dokud P_0 nedokončí ZS

```
proces Pi
repeat
    while (turn != i);
    KS;
    turn = j;
    ZS;
forever
```

SW řešení – algoritmus 3

- sdílená proměnná **flag**
 - pro každý proces: **flag[i]**
- požadavek vstupu do KS:
flag[i] = true
- požadavek vzájemného vylučování je splněn
- **požadavek pokroku není splněn!**
 - po sekvenci: P_0 : **flag[0] = true**;
 P_1 : **flag[1] = true**; – **DEADLOCK**

```
proces P,  
repeat  
    flag[i] = true;  
    while (flag[j]);  
    KS;  
    flag[i] = false;  
    ZS;  
forever
```

SW řešení – Petersonův algoritmus

- inicializace
 - **flag[i] = false**, i = 0..1
 - **turn = 0** nebo **1**
- signalizace připravenosti ke vstupu do KS nastavením **flag[i] = true**
- pokud se oba procesy pokusí vstoupit do KS současně, pouze jeden bude mít potřebnou hodnotu proměnné **turn**

```
proces P,  
repeat  
    flag[i] = true;  
    turn = j;  
    while (flag[j]  
          && turn == j);  
    KS;  
    flag[i] = false;  
    zs;  
forever
```

SW řešení – Peterson – analýza

- vzájemné vylučování splněno
 - není možné, aby P_0 a P_1 byly oba současně v KS
 - nemůže nastat `turn == i` a `flag[i] == true` pro každý P_i
- pokrok a omezenost čekání
 - pokud P_j nepožaduje vstup do KS, je `flag[j] == false` a P_i tedy může vstoupit do KS
 - P_i nemůže vstoupit do KS dokud P_j požaduje KS (`flag[j]`) a současně je P_j na řadě (`turn == j`)

```
proces Pi
repeat
    flag[i] = true;
    turn = j;
    while (flag[j]
           && turn == j);
    KS;
    flag[i] = false;
    ZS;
forever
```

SW řešení KS pro n procesů

- Leslie Lamport's bakery algorithm
 - každý proces dostane před vstupem do KS číslo
 - držitel nejmenšího čísla smí vstoupit do KS
 - dostanou-li P_i a P_j stejná čísla, přednost má $P_{\min(i, j)}$
 - ve výstupní sekci nastaví proces přidělené číslo na 0
- poznámky k zápisu:
 - $(a, b) < (c, d)$, když $a < c$ nebo když $a = c$ a $b < d$
 - $\max(a_0, \dots, a_k)$ je takové $b \geq a_i$ pro $i = 0, \dots, k$

SW řešení KS pro n procesů

- sdílená data
 - **bool** `choosing[n]`; – proces vybírá číslo
 - inicializace všech na **false**
 - **int** `number[n]`; – přidělené číslo číslo
 - inicializace všech na nulu
- korektnost algoritmu závisí na faktu
 - je-li P_i v KS a P_k si právě vybral své číslo, pak
$$(\text{number}_i, i) < (\text{number}_k, k)$$

SW řešení KS – algoritmus bakery

```
proces Pi
repeat
    choosing[i] = true;
    number[i] = max(number[0]..number[n-1]) + 1;
    choosing[i] = false;
    for (j = 0; j < n; ++j) {
        while (choosing[j]); // kvůli souběhu výpočtu max
        while (number[j] != 0 && // Pj požaduje KS
               (number[j],j) < (number[i],i)); // Pj má přednost
    }
    KS;
    number[i] = 0;
    ZS;
forever
```

Vliv chyb procesu

- splněním všech tří kritérií (vzájemné vylučování, pokrok, omezené čekání) je řešení odolné vůči chybám ve zbytkové sekci (ZS)
 - chyba ve ZS je totéž, co neomezeně dlouhá ZS
- řešení však **nemůže zajistit odolnost vůči chybám v KS**
 - pokud proces P_i havaruje v KS,
 - pro ostatní procesy je stále v KS
 - a ty se do ní nedostanou

Aktivní čekání

- označuje se též jako **spin lock**
- procesy čekající na vstup do KS spotřebovávají zbytečně a neproduktivně čas procesoru
 - zvláště pokud je KS dlouhá
 - efektivnější by bylo čekající procesy blokovat
- vhodné pouze tehdy, když je KS velmi krátká
 - ve srovnání s délkou časového kvanta běhu na CPU

HW řešení – výchozí předpoklady

- procesy se provádějí v procesoru kontinuálně, dokud nevyvolají službu OS nebo nejsou přerušeny
- k **přerušení** procesu může dojít **pouze na hranicích instrukcí**
 - mezi dokončením jedné instrukce a zahájením další
- přístup k paměti je obvykle výlučný
 - důležité zejména pro systémy SMP

HW řešení – zákaz přerušení (1)

- proces běží, dokud nezavolá službu OS nebo není přerušen
 - plánovač využívá přerušení
 - zákaz přerušení způsobí, že proces nemůže být přerušen, tudíž žádný jiný proces nemůže vstoupit do KS
- vzájemné vylučování je zajištěno

pouze na jednoprocesorových systémech

 - na systémech SMP není zaručeno!

```
proces P,  
repeat  
    disable irqs;  
    KS;  
    enable irqs;  
    ZS;  
forever
```

HW řešení – zákaz přerušení (2)

- zvyšuje latenci systému
 - během KS nemůže být obsloužena žádná událost – vadí zejména v multimediálních a RT systémech
- v KS se nesmí volat služba OS
 - jádro OS by mohlo aktivovat plánovač, jenž by mohl přepnout na jiný proces vstupující do KS
- zákaz přerušení je privilegovaná instrukce
- závěr: **obecně nevhodné řešení**

```
proces P,  
repeat  
    disable irqs;  
    KS;  
    enable irqs;  
    ZS;  
forever
```

HW řešení – speciální instrukce

- přístup k paměťovému místu je obvykle výlučný
- lze tedy navrhnut instrukci, která **atomicky** provede dvě akce s jedním paměťovým místem
 - čtení a zápis jako nedělitelná operace
- provedení instrukce nelze přerušit
 - context switch probíhá pouze na hranicích instrukcí
 - zaručí nám tak **vzájemné vylučování**, a to i na **víceprocesorových systémech**
 - ostatní požadavky je třeba řešit algoritmicky

HW řešení – instrukce test-and-set

- jediná instrukce procesoru
přečte příznak a současně
ho nastaví
 - byl-li příznak již nastaven,
nové nastavení nic nemění
 - KS je obsazena
 - nebyl-li příznak nastaven, proces smí vstoupit do KS
 - instrukce je nepřerušitelná, proto jiný proces vyhodnotí
touto instrukcí příznak správně
- lze využít pro více různých KS

instrukce test-and-set

```
int testAndSet(int *lck)
{
    if (*lck == 0) {
        *lck = 1;
        return 0; // KS
    } else {
        return 1; // čekání
    }
}
```

HW řešení – nevýhody test-and-set

- při čtení příznaku se používá aktivní čekání (AČ)
- může dojít k **vyhladovění**
 - soupeří-li několik procesů o vstup do KS po jejím uvolnění, dostane se do ní první, který provede instrukci, ostatní mohou vyhladovět
- může vzniknout **deadlock**
 - proces s nízkou prioritou je přerušen v KS
 - proces s vyšší prioritou požaduje vstup do KS (AČ)
 - nikdy se jí nedočká, protože proces s nižší prioritou nedostane šanci ji opustit

HW řešení KS – test-and-set

- sdílená prom. **locked**
 - inicializovaná na 0
 - vstupní sekce využívá instrukci test-and-set
 - ve výstupní sekci stačí přiřazovací příkaz
- první proces nastaviv **locked**, vstoupí do KS
 - ostatní aktivně čekají prováděním test-and-set
 - provádění je vzájemně výlučné i na SMP

inicializace

```
int locked = 0;  
proces P,  
repeat  
    while  
        (testAndSet(&locked));  
    KS;  
    locked = 0;  
    ZS;  
forever
```

HW řešení KS – instrukce xchg

- Intel x86 – instrukce **xchg**
 - vymění obsah dvou proměnných
- sdílená proměnná **locked**
 - inicializace na 0
- lokální prom. **s** pro každý P_i
 - ve vstupní sekci nastavena na 1
 - v aktivním čekání **s** vyměňuje hodnotu s **locked**
 - je-li **locked** = 0, nastaví se na 1 a končí čekání
 - proces vstupuje do KS

```
proces  $P_i$ 
int s;
repeat
    s = 1;
    while (s)
        xchg(s,locked);
    KS;
    locked = 0;
    ZS;
forever
```

Systémy SMP a SW řešení KS

- je třeba uvažovat
 - cache na každém procesoru
 - typ cache z hlediska zápisu změn
 - write through
 - write back (behind)
 - umístění a vyrovnávání (caching) sdílené paměti
 - způsob zápisu do paměti jednotlivými procesory
 - řazení zápisu pro procesor
 - řazení zápisu různými procesory a viditelnost změn

Systémy SMP a cache – příklad

- uvažujte následující sekvenci na systému SMP
 - inicializace je: $A = 0$, $B = 0$
 - procesor 1 zapíše hodnotu 1 na adresu A
 - procesor 1 zapíše hodnotu 1 na adresu B
 - procesor 2 čeká na změnu hodnoty na adrese B
 - aktivní čekání dokud je B nulové
 - procesor 2 přečte hodnotu z adresy A
- Jakou hodnotu procesor 2 přečte?

Systémy SMP a cache – příklad

procesor 1

```
mov [A], 1    # zápis do prom. A  
mov [B], 1    # zápis do prom. B
```

procesor 2

```
loop:  
cmp [B], 0    # porovnání hodnot  
jz loop      # rovnost ⇒ skok  
mov R, [A]    # čtení prom. A
```

datová cache procesoru 1

A: 1
B: 1

datová cache procesoru 2

B: 1 # aktualizovaná hodnota
A: 1 # aktualizovaná hodnota

systémová paměť

A: 1 # změněná proměnná A
B: 1 # změněná proměnná B

Co se uloží do registru R?

Systémy SMP a cache – Peterson

procesor 1 – proces 0

```
mov [flag],1    # true do flag[0]
mov [turn],1    # zápis 1 do turn
cmp [flag+1],0 # je-li flag[1] false,
jz  KS          # skok do KS
```

procesor 2 – proces 1

```
mov [flag+1],1 # true do flag[1]
mov [turn],0    # zápis 0 do turn
cmp [flag],0   # je-li flag[0] false,
jz  KS          # skok do KS
```

datová cache procesoru 1

flag[0]: 1
flag[1]: 0
turn: 1

datová cache procesoru 2

flag[0]: 0
flag[1]: 1
turn: 0

systémová paměť je pomalejší, tudíž zápis se neprojeví ihned

flag[0]: 0
flag[1]: 0
turn: 0

Důsledek: **oba procesy jsou v KS!**

Systémy SMP a zápis do paměti

- řazení zápisu do paměti – **write ordering**
 - dřívější zápis do paměti procesorem bude vidět **před** pozdějším zápisem
 - **vyžaduje write through cache**
- sekvenční konzistence – **sequential consistency**
 - pokud procesor 1 zapíše na adresu A
 - **před** zápisem procesoru 2 na adresu B,
 - pak nová hodnota na A musí být viditelná **všemi** procesory **před** změnou na B
 - **vyžaduje nepoužívat cache pro sdílená data!**

Systémy SMP – důsledky

- SW algoritmy řešící vstup do KS vyžadují
 - write ordering (změny CPU-cache → RAM)
 - sequential consistency (změny RAM → CPU-cache)
 - víceprocesorové systémy toto **nemusejí zaručovat**
 - superskalární CPU s pipeline – dopad na rychlosť
- **čistě SW řešení bychom se měli vyvarovat**
- HW instrukce test-and-set na SMP způsobí
 - zápis změn z cache do hlavní paměti
 - aktualizaci cache na všech procesorech

Řešení KS pomocí OS – **semafor**

- synchronizační nástroj
 - prostředek OS
- nevyžaduje aktivní čekání
 - dokud je KS obsazená
 - čekající proces je blokován a
 - zařazen do fronty procesů čekajících na uvolnění KS
 - po uvolnění KS je z fronty vybrán další proces

Řešení OS – definice semaforu

- semafor je obecně datová struktura
 - celočíselný čítač
 - fronta procesů
 - atomické operace **init**, **wait** a **signal**
- k čítači se přistupuje výhradně pomocí operací
- operace musí být provedeny **atomicky**
 - operaci smí provádět vždy jen jediný proces

semafor

```
typedef struct {  
    int count;  
    fifo_t *q;  
} sem_t;
```

```
void sem_init(sem_t *s, int v);  
void sem_wait(sem_t *s);  
void sem_post(sem_t *s);
```

Řešení OS – operace semaforu

- **init** inicializuje čítač
- **wait** snižuje čítač
 - je-li záporný
 - zařadí volající vlákno do fronty a blokuje je
- **signal** zvyšuje čítač
 - je-li fronta neprázdná
 - vybere vlákno z fronty
 - zařadí je do seznamu připravených vláken

semafor – implementace

```
void sem_init(sem_t *s, int v) {  
    s->count = v;  
}  
  
void sem_wait(sem_t *s) {  
    s->count--;  
    if (s->count < 0) {  
        fifo_put(s->q, self);  
        block calling thread self;  
    }  
}  
  
void sem_post(sem_t *s) {  
    s->count++;  
    if ((t = fifo_get(s->q)))  
        activate thread t;  
}
```

Řešení OS – semafor – pozorování

- čítač je třeba inicializovat nezápornou hodnotou
- **nezáporný čítač** udává počet procesů, jež smějí bez blokování zavolat **wait**
- absolutní hodnota **záporného čítače** udává, **kolik procesů čeká ve frontě** na semafor
- **atomicita** oper. zaručuje **vzájemné vylučování**
 - programové bloky operací se semaforem jsou vlastně také **kritické sekce**
 - OS musí tyto KS v operacích ošetřit

Řešení OS – semafor – KS operací

- kritické sekce v implementaci operací semaforu jsou velmi krátké
 - typicky asi deset instrukcí
- vhodné ošetření KS
 - zákaz přerušení
 - pouze jednoprocesorové systémy
 - spin lock užitím HW nebo SW metody
 - HW spin lock je na víceprocesorových systémech nutností

Použití semaforu – řešení KS

- sdílený semafor **s** se inicializuje na počet procesů smějících vstoupit do KS
 - typicky na jeden
- vstupní sekce volá **wait**
- výstupní sekce volá **signal**
- obecně lze povolit neblokující volání **wait** pro n procesů
 - inicializujeme-li semafor na n

```
jeden z procesů Pi,  
sem_init(&s, 1);  
  
proces Pi,  
repeat  
    sem_wait(&s);  
    KS;  
    sem_post(&s);  
    zs;  
forever
```

Použití semaforu – synchronizace

- příkaz **S2** v procesu **P₁** musí být proveden až **po provedení S1** v **P₀**
- čítač semaforu **sync** inicializujeme na 0
- proces **P₁** před provedením **S2** počká na signál od **P₀**
- provede-li se nejprve **signal**, pak **wait** neblokuje

```
 inicializace
sem_init(&sync, 0);
proces P0
vypočti x; // S1
sem_post(&sync);

proces P1
sem_wait(&sync);
použij x; // S2
```

Binární semafor

- místo čítače je použita Booleovská proměnná
 - inicializace na false
- obvykle se označuje pojmem **mutex**
 - odvozeno od využití pro vzájemné vylučování – **mutual exclusion**
 - nelze jej (samostatně) využít pro synchronizaci
- typicky se operace **wait** a **signal** označují termíny **lock** a **unlock**

binární semafor

```
typedef struct {  
    bool locked;  
    fifo_t *q;  
} mutex_t;
```

Binární semafor – implementace

- operace **lock** (wait) a **unlock** (signal) nepočítají počty procesů
 - **lock** je neblokující pouze pro jeden proces
 - **unlock** odemkne mutex, je-li fronta procesů prázdná

mutex – implementace

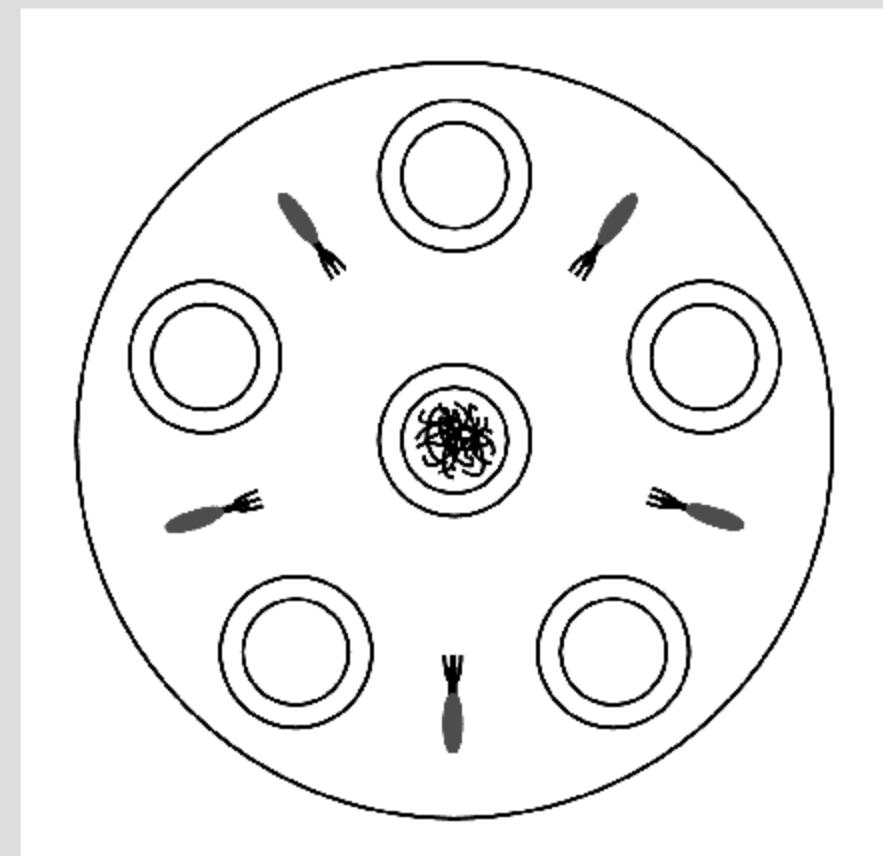
```
void mutex_lock(mutex_t *m) {  
    if (m->locked) {  
        fifo_put(m->q, self);  
        block calling thread self;  
    } else {  
        m->locked = true;  
    }  
}  
  
void mutex_unlock(mutex_t *m) {  
    if ((t = fifo_get(m->q))) {  
        activate thread t;  
    } else {  
        m->locked = false;  
    }  
}
```

Semafor – hodnocení

- semafor je výkonný nástroj pro řešení
 - vzájemného vylučování
 - synchronizace procesů
- protože jsou operace **wait** a **signal** volány z různých procesů, může být obtížné plně porozumět jejich působení
 - použití musí být korektní u **všech** procesů
 - jediné chybné použití může způsobit problémy s celou skupinou procesů

Problém obědvajících filosofů

- klasický synchronizační problém
- u stolu sedí pět filosofů
 - každý bud' přemýšlí,
 - nebo jí
- při jídle každý potřebuje dvě vidličky
- k dispozici je pouze pět vidliček



Problém filosofů – návrh řešení

- každý filosof odpovídá jednomu procesu
- jedení – kritická sekce
- přemýšlení – zbytková sekce
- vidličky – sdílené prostředky
 - pro výlučný přístup k vidličkám je třeba pro každou použít semafor

inicializace

```
sem_t fork[n]; // n = 5
for (i = 0; i < n; ++i)
    sem_init(&fork[i], 1);

proces Pi
repeat
    think; // ZS
    sem_wait(&fork[i]);
    sem_wait(&fork[(i+1)%n]);
    eat; // KS
    sem_post(&fork[(i+1)%n]);
    sem_post(&fork[i]);
forever
```

Problém filosofů – analýza návrhu

- každý filosof čeká na vidličku po své levé i pravé ruce, dokud se neuvolní
- avšak může nastat následující situace:
 - všichni se ve stejný okamžik rozhodnou najít
 - každý zvedne vidličku po své levé ruce
 - všichni čekají na uvolnění vidličky po své pravé ruce
⇒ DEADLOCK

```
proces Pi
repeat
    think; // ZS
    sem_wait(&fork[i]);
    sem_wait(&fork[(i+1)%n]);
    eat; // KS
    sem_post(&fork[(i+1)%n]);
    sem_post(&fork[i]);
forever
```

Problém filosofů – řešení

- je třeba dovolit zvedat vidličky nejvýše čtyřem ($n - 1$) filosofům
- pak vždy aspoň jeden filosof může jíst
 - ostatní musejí čekat (tři s vidličkou v ruce, jeden bez vidličky)
- pro omezení zvedání vidliček lze použít semafor

inicializace

```
sem_t waiter;  
sem_init(&waiter, n - 1);  
  
proces Pi  
repeat  
    think; // ZS  
    sem_wait(&waiter);  
    sem_wait(&fork[i]);  
    sem_wait(&fork[(i+1)%n]);  
    eat; // KS  
    sem_post(&fork[(i+1)%n]);  
    sem_post(&fork[i]);  
    sem_post(&waiter);  
forever
```

Problém filosofů – analýza řešení

inicializace

```
for (i = 0; i < n; ++i) sem_init(&fork[i], 1); // vidličky  
sem_init(&waiter, n - 1); // n = 5, pouze čtyři smějí zvedat vidličku
```

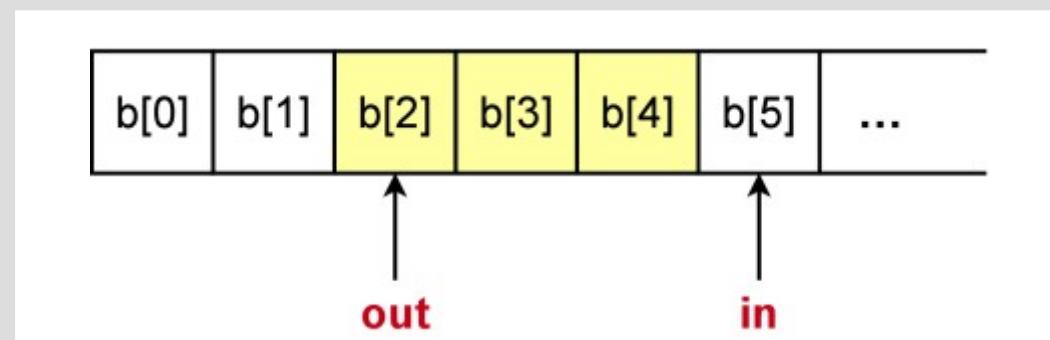
proces P_i	P_0	P_1	P_2	P_3	P_4	f_0	f_1	f_2	f_3	f_4	w
think; // ZS	ZS	ZS	ZS	ZS	ZS	1	1	1	1	1	4
sem_wait(&waiter);	R	R	R	R	B _w	0	0	0	0		-1
sem_wait(&fork[i]);	R	R	R	R	R						
sem_wait(&fork[(i+1)%n]);	B _{f₁}	B _{f₂}	B _{f₃}	R	B _{f₀}						
eat; // KS	KS	KS	KS	KS	KS	-1	0	1	0		
sem_post(&fork[(i+1)%n]);	R	R	R	R							
sem_post(&fork[i]);	R	R	R	R		0	1				
sem_post(&waiter);				R		0	1				

Problém producenta a konzumenta

- dva typy procesů
 - producenti – produkují výrobky (data)
 - konzumenti – spotřebovávají data
- pro lepší efektivitu je třeba zavést mezisklad
 - vyrovnávací paměť na vyprodukovaná data
 - nevzniknou tak zbytečné čekací doby
- pro řízení přístupu do skladu lze použít semafory
 - kapacita skladu – pro počet výrobků na skladě
 - možnost vstupu do skladu

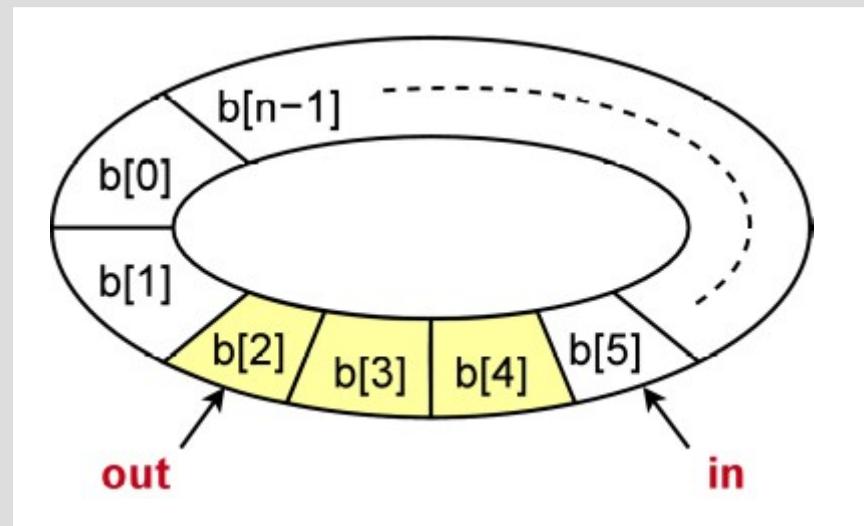
Nekonečná kapacita skladu

- vyrovnávací paměť (buffer) – sklad
 - pole s nekonečnou kapacitou
- pole musí mít nezávislý přístup pro čtení a zápis
 - dva indexy pole
 - index prvního volného místa pro zápis – **in**
 - index prvního obsazeného místa pro čtení – **out**



Kruhový buffer

- v praxi je kapacita vyrovnávací paměti omezená
 - pole je propojeno do kruhu – kruhový buffer
 - po poslední položce následuje zase první
 - je třeba hlídat kapacitu skladu (bufferu)
 - po zaplnění musejí producenti čekat



Producent / konzument – funkce

Producent

```
repeat
    item = produce_item();
    while (count == n);
    buffer[in] = item;
    in = (in + 1) % n;
    count++;
forever
```

Konzument

```
repeat
    while (count == 0);
    item = buffer[out];
    out = (out + 1) % n;
    count--;
    consume_item(item);
forever
```

- producentů i konzumentů může být několik
- je třeba zajistit
 - vzájemné vylučování při práci s bufferem a indexy
 - hlídání obsazenosti bufferu (stavy prázdný a plný)

Producent / konzument – řešení

Inicializace

```
sem_init(&item_count, 0);           // počet položek v bufferu  
sem_init(&mutex, 1);              // vzájemné vylučování  
// neúplné řešení
```

Producent

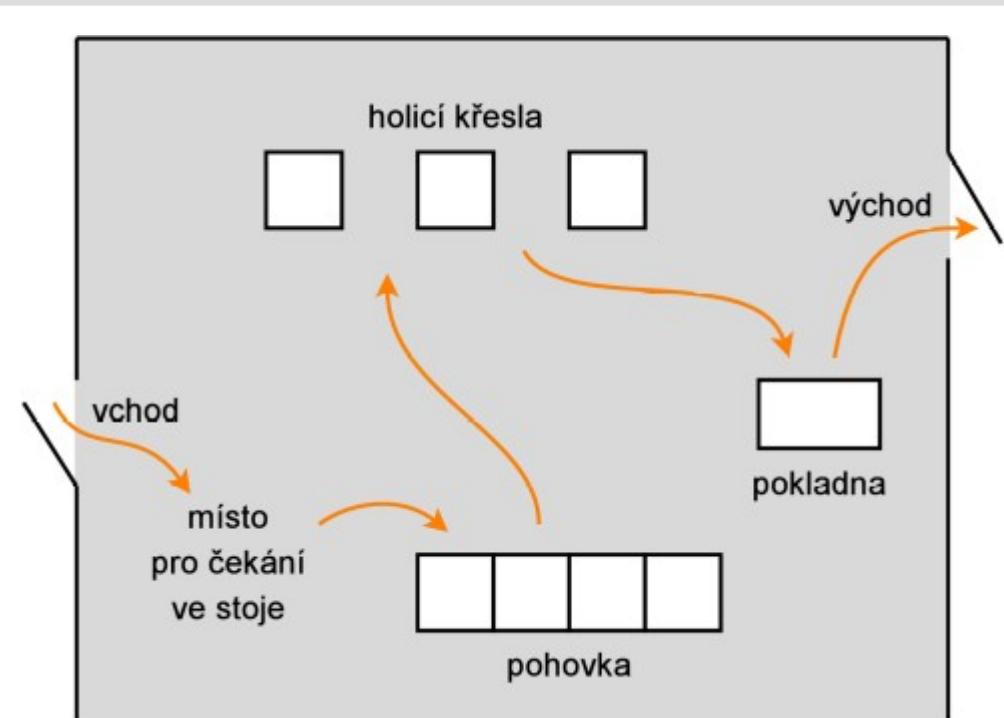
```
repeat  
    item = produce_item();  
    sem_wait(&mutex);  
    buffer[in] = item;  
    in = (in + 1) % n;  
    sem_post(&mutex);  
    sem_post(&item_count);  
forever
```

Konzument

```
repeat  
    sem_wait(&item_count);  
    sem_wait(&mutex);  
    item = buffer[out];  
    out = (out + 1) % n;  
    sem_post(&mutex);  
    consume_item(item);  
forever
```

Barbershop Problem

- u holiče jsou jen 3 křesla
 - a tedy jen 3 holiči
- dovnitř holičství se vejde nejvýše 20 zákazníků
 - další musí čekat venku nebo odejít
- na pohovce je místo pro 4 čekající zákazníky
 - ostatní musí čekat ve stoj
- pokladna je pouze jedna
 - inkasovat může v jednom okamžiku pouze jedený holič



Potřebujete ostříhat / oholit?

- když je holič volný, přesune se nejdéle čekající zákazník z pohovky na křeslo
- nejdéle stojící zákazník pak usedne na pohovku
- platit může v daném okamžiku vždy pouze jeden zákazník
- holiči obsluhují zákazníky
 - pokud nečeká žádný zákazník, holiči spí

Co je nutné sledovat?

- vzájemné vylučování
 - kapacitu holičství, pohovky, křesel a pokladen
- synchronizaci
 - umístění právě jednoho zákazníka do křesla
 - přítomnost zákazníka v křesle (chci oholit)
 - dokončení holení
 - opuštění křesla a přechod k pokladně
 - zaplacení a vydání účtenky

Barbershop Problem – inicializace

Barbershop Problem – init

limity

```
sem_init(max_capacity, 20);      // místnost  
sem_init(sofa, 4);              // pohovka  
sem_init(chair, 3);             // křesla
```

synchronizace

```
sem_init(cust_ready, 0);          // zákazník je na křesle  
for (i = 0; i < chairs; ++i)  
    sem_init(finished[i], 0);       // holič dostříhal  
sem_init(payment, 0);            // zákazník zaplatil  
sem_init(receipt, 0);           // holič vydal účtenku
```

Barbershop Problem – algoritmus

Zákazník

```
sem_wait(max_capacity);
enter_barbershop();
sem_wait(sofa);
sit_on_sofa();
sem_wait(chair);
get_up();
sem_post(sofa);
ch = sit_in_chair();
sem_post(cust_ready);
sem_wait(finished[ch]);
leave_chair();
sem_post(chair);
pay();
sem_post(payment);
sem_wait(receipt);
exit_shop();
sem_post(max_capacity);
```

Holič

```
repeat
    sem_wait(cust_ready);
    ch = go_to_chair();
    cut_hair();
    sem_post(finished[ch]);
    go_to_cash_register();
    sem_wait(payment);
    accept_pay();
    sem_post(receipt);
forever
```

IPC dle OS UNIX System V

- **UNIX System V** – AT&T, Bell Labs, 1983
 - přímý následník původního UNIXu z roku 1969
 - je základem mnoha implementací
 - AIX (IBM), Solaris (Sun Microsystems) nebo HP-UX (HP)
- mezičlenová komunikace – svipc(7), ftok(3)
 - prostředky OS zpřístupněné systémovými voláními
 - fronty zpráv – msgget(2), msgctl(2), msgop(2)
 - sady semaforů – semget(2), semctl(2), semop(2)
 - segmenty sdílené paměti – shmget(2), shmctl(2), shmop(2)
 - XSI (X/Open System Interface) rozšíření POSIX.1

IPC dle norem **POSIX.1-2001**

- standardizace mezičílené komunikace
- implementuje nověji a s lepším designem IPC
 - POSIXová volání jsou knihovní funkce, opírají se však o systémová volání
 - sdílené prostředky
 - fronty zpráv – mq_overview(7)
 - sady semaforů – sem_overview(7)
 - sdílená paměť – shm_open(3), mmap(2)
 - obvykle se s nimi pracuje obdobně jako se soubory
 - open, close, unlink, práva

Systémy vyhovující POSIXu

- certifikované
 - AIX (IBM), HP-UX (HP), IRIX (SGI), OS X od verze 10.5 (Apple), Solaris (Sun, Oracle), Tru64 UNIX (DEC, Compaq, HP), UnixWare (AT&T, Novell, SCO, Xinuos), QNX Neutrino (BlackBerry)
- vyhovující, kompatibilní a většinově kompatibilní
 - Linux, Android
 - BSD: FreeBSD, NetBSD, OpenBSD
 - aj.: VxWorks, MINIX, Darwin (jádro OS X a iOS), ...

POSIX a MS Windows

- rozšíření, POSIX-compliant IDE
 - Cygwin, MinGW
- Microsoft POSIX subsystem
 - volitelné rozšíření pro Windows do verze Win 2000
 - pouze POSIX.1-1990, bez vláken a socketů
- Interix, později Windows Services for UNIX, později Subsystem for UNIX-based Applications
 - pro Windows Server 2003 R2 a pozdější
 - zavrženo od Win 8, vypuštěno od 2012 R2 a 8.1

Posixové semafory

- hlavičkový soubor semaphore.h(7)

```
#include <semaphore.h>
```

```
gcc ... -lrt ... # připojit knihovnu real-time (librt.so)
```

- čítací semafor s nezáporným čítačem
 - POSIX dovoluje u volání `sem_getvalue(3)` vracet i zápornou hodnotu – délku fronty
- existují dva typy semaforů – `sem_overview(7)`
 - pojmenované – jméno je tvaru `/name`
 - nepojmenované – jsou jen v paměti

Posixové semafory – typy

- s pojmenovanými semafory se pracuje podobně jako se soubory
 - otevření (open), zavření (close), odstranění (unlink)
 - procesy mohou sdílet semafor stejného jména
 - operace wait a post (signal)
- nepojmenované semafory lze alokovat
 - v paměti procesu – sdílení mezi vlákny
 - ve sdílené paměti – sdílení mezi procesy
 - musí nasdílet pomocí shmget(2) nebo shm_open(3)

Semafora POSIX – inicializace

- inicializace – **sem_init(3)**

```
int sem_init(sem_t *sem, int pshared,  
             unsigned int value);
```

- inicializuje čítač semaforu **sem** na hodnotu **value**
- **pshared** udává typ sdílení
 - nulová hodnota – v rámci procesu mezi vlákny
 - nenulová hodnota – mezi procesy
 - semafor pak musí být ve sdílené paměti
- vrací: 0 = úspěch, -1 = chyba, **errno** udává chybu

Semafora POSIX – wait a post

- čekání – **sem_wait(3)**
`int sem_wait(sem_t *sem);`
 - snižuje čítač semaforu **sem**
 - blokuje proces, pokud by se snižovalo do záporu
 - vrací: 0 = úspěch, -1 = chyba, **errno** udává chybu
- signalizace – **sem_post(3)**
`int sem_post(sem_t *sem);`
 - zvyšuje čítač, probudí jeden čekající proces
 - vrací: 0 = úspěch, -1 = chyba, **errno** udává chybu

Semafora POSIX – linuxová implementace wait a post

posixová implementace v Linuxu – nezáporným čítač + futex

```
int sem_wait(sem_t *s) {
    if (atomic_dec_if_positive(s->count))          // je-li čítač kladný,
        return 0;                                // atomicky jej sníží o jedna a ukončí se
    atomic s->nwaiters++; // jinak zvýší počet čekajících vláken
    while (1) { // futex = fast userspace mutex, řadí vlákna do prioritní fronty
        futex_wait(s->count, 0, ...); // dokud je čítač nulový, blokuje
        if (atomic_dec_if_positive(s->count)) // kladný čítač sníží
            break; // a cyklus končí
    } // snížilo-li čítač na nulu mezitím jiné vlákno, cyklus blokování pokračuje
    atomic s->nwaiters--; // počet čekajících vláken je snížen
}

int sem_post(sem_t *s) {
    atomic s->count++; // atomicky zvýší hodnotu čítače o jedna
    if (s->nwaiters > 0) // existuje-li u semaforu blokované vlákno,
        futex_wake(s->count, 1, ...); // první se odblokuje (dle priority)
} // probuzení prvního z fronty nezaručuje jeho naplánování před jiným
```

Semafora POSIX – linuxový **futex**

- **futex** – fast userspace mutex
 - operace implementuje knihovna v userspace
 - volání jádra jen při potřebě blokovat a probouzet
 - operace **wait** – parametry: proměnná, hodnota, ...
 - zablokuje volající vlákno, má-li proměnná danou hodnotu
 - blokované vlákno je dle své priority vloženo do fronty
 - operace **wake** – parametry: proměnná, počet, ...
 - probudí daný počet vláken čekajících na dané proměnné
 - přestože jsou vlákna vybírána z fronty, **není zaručeno jejich naplánování před ostatními nad stejnou proměnnou**

Semafora POSIX – linuxová implementace wait a post (2)

vlákno 1

```
sem_wait(&s) :  
    s->count--;  
    return;  
// provádí KS  
  
// konec KS  
sem_post(&s) :  
    s->count++;  
    futex_wake(...);  
    // probudí vlákno 2
```

vlákno 2

```
sem_wait(&s) :  
    // čítač je nulový  
    futex_wait(...);  
    // blokuje  
    // je první ve frontě
```

// vlákno je připraveno

// čítač je nulový
// vlákno opět blokuje

vlákno 3

```
// vlákno běží  
sem_wait(&s) :  
    s->count--;  
    return;  
// provádí KS
```

Semafora POSIX – get, destroy

- zjištění hodnoty čítače – `sem_getvalue(3)`
`int sem_getvalue(sem_t *sem, int *sval);`
 - vloží hodnotu čítače na adresu `sval`
 - vrací: 0 = úspěch, -1 = chyba, `errno` udává chybu
- uvolnění prostředků – `sem_destroy(3)`
`int sem_destroy(sem_t *sem);`
 - uvolní prostředky inicializovaného semaforu
 - vrací: 0 = úspěch, -1 = chyba, `errno` udává chybu

Semafora POSIX – omezené wait

- omezené čekání – `sem_timedwait(3)`

```
int sem_timedwait(sem_t *sem,  
                  const struct timespec *abs_timeout);
```

- blokuje než vyprší timeout
- vrací: 0 = OK, **ETIMEDOUT** při nutnosti čekat nadále

- pokus o čekání – `sem_trywait(3)`

```
int sem_trywait(sem_t *sem);
```

- neblokující – vrací **EAGAIN** při nutnosti čekat

Semafora POSIX – otevření

- otevření pojmenovaného sem. – `sem_open(3)`
`sem_t *sem_open(const char *name, int *oflag);`
 - otevře sdílený semafor jména `name` (tvar: `/jméno`)
 - obsahuje-li `oflag O_CREAT`, semafor se vytvoří (neexistuje-li) a je nutné zadat další dva argumenty
`sem_t *sem_open(const char *name, int *oflag,
mode_t mode, unsigned int value);`
 - `mode`: práva (`S_IRWXU, S_IRUSR`, ...), respektuje umask
 - `value`: inicializace čítače semaforu
 - vrací: semafor nebo `SEM_FAILED` při chybě, `errno`
 - `#include <fcntl.h>` (oflag) a `<sys/stat.h>` (mode)

Semafora POSIX – zavření, zrušení

- zavření semaforu – `sem_close(3)`

```
int sem_close(sem_t *sem);
```

- zavře pojmenovaný semafor
 - vrací: 0 = úspěch, -1 = chyba, `errno` udává chybu

- uvolnění prostředků – `sem_unlink(3)`

```
int sem_unlink(const char *name);
```

- odstraní jméno semaforu
 - prostředky semaforu jsou uvolněny, až všechny procesy semafor zavřou
 - vrací: 0 = úspěch, -1 = chyba, `errno` udává chybu

Semafora System V – IPC

- alokace semaforů – `semget(2)`

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>

int semget(key_t key, int nsems, int semflg);
```

- vrátí identifikátor sady semaforů podle klíče `key`,
při chybě vrací `-1` (a nastaví `errno`)
- nová sada `nsems` semaforů je vytvořena, pokud
 - `key = IPC_PRIVATE` (lepší název by byl `IPC_NEW`) nebo
 - `key ≠ IPC_PRIVATE`, sada sdružená s klíčem `key`
neexistuje a `semflg` obsahuje `IPC_CREAT`

Semafony System V – alokace

- alokace semaforů (pokračování) – `semget(2)`
`int semget(key_t key, int nsems, int semflg);`
 - semafory **nejsou** inicializovány (POSIX.1-2001)
 - struktura **semid_ds** je naplněna – viz `semctl(2)`
 - vlastník a skupina podle volajícího procesu, práva a počet semaforů v sadě podle parametrů, čas modifikace
 - práva lze specifikovat dolními 9 bity v **semflg**
 - bit **x** (execute) systém nevyužívá; práva – viz též `stat(2)`
 - je-li dán **IPC_CREAT** a **IPC_EXCL** a semafor již existuje, vrací chybu (**EEXIST**)

Semafora System V – operace

- operace se sadou semaforů – `semop(2)`

```
int semop(int semid, struct sembuf *sops,  
          unsigned nsops);
```

- provede `nsops` operací se semafory s id `semid`
 - operace jsou v poli `sops`, jehož položky jsou:

```
struct sembuf {  
    unsigned short semnum; // číslo semaforu (čísluje se od 0)  
    short sem_op; // číslo, o které se změní čítač, nebo 0 (čekej)  
    short sem_flg; // příznaky: SEM_UNDO, IPC_NOWAIT  
}
```

- `SEM_UNDO` – při ukončení procesu se zruší změna
- `sem_op` je obvykle `-1` (`wait`) nebo `(+1)` (`signal`)
 - hodnota `sem_op` `0` znamená čekání na nulu

Semafora System V – ovládání

- ovládání sady semaforů – `semctl(2)`

```
int semctl(int semid, int semnum, int cmd, ...);
```

- vykoná příkaz `cmd` na semaforu `semnum` v `semid`
 - `IPC_STAT` (zjištění info), `IPC_SET` (nastavení práv),
`IPC_RMID` (okamžité odstranění sady semaforů),
`SETVAL`, `SETALL` (nastavení čítače/ů semaforu/ů), ...
- podle `cmd` je třeba i čtvrtý argument
 - typ je nutné deklarovat v programu!

```
union semunion {  
    int val;                      // hodnota semaforu (pro SETVAL)  
    struct semid_ds *buf;          // buffer pro IPC_STAT a IPC_SET  
    unsigned short *array;         // pole pro SETALL a GETALL  
};
```

Semafora System V – příklad

použití semaforu System V

definice union

```
typedef union {
    int val;
    struct semid_ds *buf;
    unsigned short *array;
} semunion_t;
```

deklarace proměnných

```
int sID;
semunion_t sdata;
key_t sKey = 1357;
// lokálně pro každé vlákno:
struct sembuf sops;
```

inicializace

```
sID = semget(sKey, 1,
IPC_CREAT|0666); // umask platí
sdata.val = 1;
semctl(sID, 0, SETVAL, sdata);
```

wait a signal

```
sops.sem_num = 0;
sops.sem_flg = SEM_UNDO;
sops.sem_op = -1; // wait
semop(sID, &sops, 1);
sops.sem_op = +1; // signal
semop(sID, &sops, 1);
```

odstranění

```
semctl(sID, 0, IPC_RMID);
```

Semafora – Win32 API

- CreateSemaphore() – alokace / otevření semaforu
- OpenSemaphore() – otevření pojmenovaného semaforu
- WaitForSingleObject() – operace wait
- ReleaseSemaphore() – operace signal
- CloseHandle() – uvolnění prostředků

Posixová vlákna a mutexy

- mutex je zámek, který zaručuje
 - atomicitu a vzájemné vylučování
 - lze zamknout pouze jediným vláknem
 - neaktivní čekání
 - vlákno je při pokusu zamknout zamčený mutex blokováno
- tři typy mutexů (dva jsou rozšiřující)
 - fast mutex – lze zamknout pouze jednou
 - recursive mutex – lze zamknout „na více západů“
 - error checking mutex – zamykání zamčeného selže

Mutex – inicializace

- deklarace a inicializace – `pthread_mutex_init(3)`

```
pthread_mutex_t a_mutex =  
PTHREAD_MUTEX_INITIALIZER;
```

- fast mutex – lze zamknout jen jednou
- zamčení jedním vláknem podruhé = DEADLOCK

```
pthread_mutex_t a_mutex =  
PTHREAD_RECURSIVE_MUTEX_INITIALIZER_NP;
```

- recursive mutex – lze zamknout **jedním** vláknem vícekrát
- jiná vlákna jsou při zamykání (již zamčeného mutexu) vždy blokována
- musí se zamykajícím vláknem vícekrát odemknout

Mutex – zamčení

- zamčení – `pthread_mutex_lock(3)`

```
int pthread_mutex_lock(pthread_mutex_t *mutex) ;
```

- zamkne mutex; je-li už zamčený, blokuje vlákno
 - vrací: 0 = úspěch, nenulová hodnota = číslo chyby

- pokus o zamčení – `pthread_mutex_trylock(3)`

```
int pthread_mutex_trylock(pthread_mutex_t *m) ;
```

- zamkne mutex; je-li už zamčený, neblokuje
 - rekurzivní mutex se podaří vlastnícímu vláknu zamknout
 - vrací: 0 = úspěch, nenulová hodnota = číslo chyby

Mutex – odemčení

- odemčení – `pthread_mutex_unlock(3)`

```
int pthread_mutex_unlock(pthread_mutex_t *m) ;
```

- odemkne mutex zamčený tímto vláknem
 - nelze odemknout vláknem nevlastnícím zámek
 - čeká-li jiné vlákno na odemčení, je odblokováno
 - výběr vlákna (čeká-li jich více) závisí na plánovací strategii
- vrací: 0 = úspěch, nenulová hodnota = číslo chyby

Mutex – zrušení, inicializace

- zrušení – `pthread_mutex_destroy(3)`

```
int pthread_mutex_destroy(pthread_mutex_t *m);
```

- zruší mutex – stane se neplatným, neinicializovaným
 - zamčený mutex nelze zrušit

- mutex lze znovu inicializovat

```
int pthread_mutex_init(pthread_mutex_t *mutex,  
                      const pthread_mutexattr_t *restrict attr);
```

- necháme-li `attr NULL`, budou atributy implicitní
- vrací: 0 = úspěch, nenulová hodnota = číslo chyby

Mutex – řešení kritické sekce

inicializace

```
pthread_mutex_t a_mutex = PTHREAD_MUTEX_INITIALIZER;  
int rc;
```

použití

```
rc = pthread_mutex_lock(&a_mutex);           // wait  
if (rc) {  
    perror("pthread_mutex_lock");  
    pthread_exit(NULL);  
}  
KS; // kritická sekce  
rc = pthread_mutex_unlock(&a_mutex);          // signal  
if (rc) {  
    perror("pthread_mutex_unlock");  
    pthread_exit(NULL);  
}  
ZS; // zbytková sekce
```

Mutexy – Win32 API

- CreateMutex(), CreateMutexEx() – alokace / otevření mutexu
- OpenMutex() – otevření pojmenovaného mutexu
- WaitForSingleObject() – operace lock
- ReleaseMutex() – operace unlock
- CloseHandle() – uvolnění prostředků

Posixová vlákna a podmínky

- podmínková proměnná
 - slouží k synchronizaci vláken
 - umožňuje vláknu neaktivně čekat na událost
 - nezaručuje exkluzivitu přístupu
 - je třeba k ní přistupovat pomocí mutexu
 - na událost může čekat i několik vláken
 - událost oznamuje některé vlákno signálem
 - signál může probudit jediné vlákno
 - lze poslat i všesměrový signál a probudit všechna vlákna
 - nečeká-li žádné vlákno, je signál ztracen

Podmínky – inicializace

- deklarace a inicializace – `pthread_cond_init(3)`

```
pthread_cond_t cond = PTHREAD_COND_INITIALIZER;
```

- inicializuje podmínkovou proměnnou (při překladu)
 - pro inicializaci v době běhu (run-time), je třeba volat

```
int pthread_cond_init(pthread_cond_t *restrict
cond, const pthread_condattr_t *restrict attr);
```

- necháme-li `attr NULL`, budou atributy implicitní
 - vrací: 0 = úspěch, nenulová hodnota = číslo chyby

Podmínky – signalizace události

- signalizace události – `pthread_cond_signal(3)`

```
int pthread_cond_signal(pthread_cond_t *cond);
```

- signalizuje událost, probudí jedno čekající vlákno

- POSIX.1 explicitně stanovuje „**alespoň jedno**“

- SMP: není efektivní ošetřovat krajní možnost probuzení více vláken

- **nečeká-li žádné vlákno, je signál ztracen**

```
int pthread_cond_broadcast(pthread_cond_t *cond);
```

- signalizuje událost, probudí všechna čekající vlákna

- vrací: 0 = úspěch, nenulová hodnota = číslo chyby

Podmínky – čekání na událost

- čekání na událost – `pthread_cond_wait(3)`

```
int pthread_cond_wait(pthread_cond_t *restrict  
cond, pthread_mutex_t *restrict mutex);
```

- blokuje vlákno, dokud nedostane signál
- před čekáním je třeba zamknout **mutex**

```
int pthread_cond_timedwait(pthread_cond_t  
*restrict cond, pthread_mutex_t *restrict  
mutex, const struct timespec *restrict abst);
```

- časově omezené čekání – **abst** je absolutní čas
- vrací: 0 = úspěch, nenulová hodnota = číslo chyby
 - v případě chyby zůstane **mutex** zamčený

Podmínkové proměnné – příklad

inicializace

```
pthread_cond_t cond = PTHREAD_COND_INITIALIZER;
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
bool done = false; // příznak dokončení
```

vlákno 1 – čeká na událost (např. vypočtení hodnoty x)

```
pthread_mutex_lock(&mutex); // zamčení mutexu
if (!done) // kontrola příznaku: není-li hotovo,
    pthread_cond_wait(&cond, &mutex); // čekání na signál
pthread_mutex_unlock(&mutex); // odemčení mutexu
use(x); // použití sdílené proměnné x
```

vlákno 2 – signalizuje událost (hodnota x je už platná)

```
compute(x); // nastavení hodnoty proměnné x
pthread_mutex_lock(&mutex); // zamčení mutexu
done = true; // nastavení příznaku dokončení
pthread_cond_signal(&cond); // signalizace dokončení
pthread_mutex_unlock(&mutex); // odemčení mutexu
```

Podmínkové proměnné – implementace čekání

vlákno 1 – čeká na událost

```
pthread_mutex_lock(&mutex);           // 1. zamčení mutexu
if (!done)                            // 2. kontrola příznaku: není-li hotovo,
    pthread_cond_wait(&cond, &mutex);   // 3. čekání na signál
pthread_mutex_unlock(&mutex);        // 13. odemčení mutexu
```

vlákno 2 – signalizuje událost

```
pthread_mutex_lock(&mutex);           // 4. blokace, 7. zamčení mutexu
done = true;                          // 8. nastavení příznaku dokončení
pthread_cond_signal(&cond);          // 9. signalizace dokončení
pthread_mutex_unlock(&mutex);        // 11. odemčení mutexu
```

knihovna – implementace čekání na signál

```
pthread_cond_wait(..._cond_t *cond, ..._mutex_t *mutex) {
    pthread_mutex_unlock(&mutex); // 5. odemčení mutexu
    // 6. blokování volajícího vlákna – bude probuzeno signálem
    pthread_mutex_lock(&mutex);  // 10. blokace, 12. zamčení mutexu
```

Synchronizace – Win32 API

- `CreateEvent()` – alokace / otevření objektu
- `OpenEvent()` – otevření pojmenovaného objektu
- `SetEvent()`, `ResetEvent()` – signalizace události, zrušení signalizace události
 - na rozdíl od `pthread_cond_signal(3)` se signál, na který žádné vlákno nečeká, neztratí
- `WaitForSingleObject()` – čekání na událost
- `CloseHandle()` – uvolnění prostředků

Posixová vlákna a bariéry

- objekt **bariéra** pro synchronizaci vláken
 - umožňuje vláknu neaktivně čekat na ostatní vlákna
 - jakmile k bariéře dospěje daný počet vláken, bariéra propustí všechna vlákna – paralelní běh
 - definováno normou POSIX.1-2001 a Single UNIX Specification, Version 3
 - nutno definovat jeden z následujících symbolů **před** všemi direktivami **#include**

```
#define __XOPEN_SOURCE 600 // obvykle tento  
#define __POSIX_C_SOURCE 200112L
```

Bariéry – inicializace

- deklarace a inicializace – `pthread_barrier_init(3)`

```
pthread_barrier_t a_barrier;  
  
int pthread_barrier_init(  
    pthread_barrier_t *restrict barrier,  
    const pthread_barrierattr_t *restrict attr,  
    unsigned count);
```

- `attr` nastavuje sdílení mezi procesy, smí být `NULL`
- hodnota `count` udává, kolik vláken musí zavolat funkci `pthread_barrier_wait(3)` pro návrat z ní
- vrací: 0 = úspěch, nenulová hodnota = číslo chyby

Bariéry – inicializace a zrušení atributů

- deklarace, inicializace a zrušení atributů bariéry

```
pthread_barrierattr_t attr;  
  
int pthread_barrierattr_init(  
    pthread_barrierattr_t *attr);
```

- inicializuje **attr** výchozími hodnotami

```
int pthread_barrierattr_destroy(  
    pthread_barrierattr_t *attr);
```

- zruší atributy (dealokuje)
- vrací: 0 = úspěch, nenulová hodnota = číslo chyby

Bariéry – sdílení mezi procesy

- nastavení a zjištění sdílení mezi procesy

```
int pthread_barrierattr_setpshared(
```

```
    pthread_barrierattr_t *attr,
```

```
    int pshared);
```

```
int pthread_barrierattr_getpshared(
```

```
    const pthread_barrierattr_t *restrict attr,
```

```
    int *restrict pshared);
```

- hodnota **pshared**: **PTHREAD_PROCESS_SHARED**

- bariéra musí být ve sdílené paměti mezi procesy

- nesdíleno: **PTHREAD_PROCESS_PRIVATE**

Bariéry – čekání na bariéře

- čekání na bariéře – `pthread_barrier_wait(3)`

```
int pthread_barrier_wait(  
    pthread_barrier_t *barrier);
```

- blokuje vlákno, dokud tuto funkci nezavolá počet vláken stanovený inicializací `pthread_barrier_init(3)`
- pak jsou všechna vlákna odblokována současně
- vrací při úspěchu pro jedno (nespecifikované) vlákno hodnotu **PTHREAD_BARRIER_SERIAL_THREAD** a pro ostatní nulu; nenulová hodnota = číslo chyby
 - bariéra je znova připravena pro použití na daný počet

Bariéry – zrušení

- uvolnění prostředků – `pthread_barrier_destroy(3)`

```
int pthread_barrier_destroy(  
    pthread_barrier_t *barrier);
```

- odstraní bariéru a uvolní s ní sdružené prostředky
- vrací: 0 = úspěch, nenulová hodnota = číslo chyby

Bariéry – příklad

inicializace

```
#define _XOPEN_SOURCE 600                                // pro přenositelnost
#include <pthread.h>
pthread_barrier_t a_barrier;                            // deklarace
if (pthread_barrier_init(&a_barrier, NULL, 5)) {
    perror("barrier init");
    exit(EXIT_FAILURE);
}
```

použití (v pěti vláknech)

```
switch (pthread_barrier_wait(&a_barrier)) {
    case PTHREAD_BARRIER_SERIAL_THREAD: // jedno z vláken
        (void) pthread_barrier_destroy(&a_barrier);
    case 0: break;                      // ostatní vlákna
    default: perror("barrier wait"); exit(EXIT_FAILURE);
}
```

Bariéry – Win32 API

- InitializeSynchronizationBarrier() – inicializace
- EnterSynchronizationBarrier() – čekání
 - návratová hodnota: 1 vlákno TRUE, ostatní FALSE
- DeleteSynchronizationBarrier() – zrušení

Předávání zpráv

- komunikační prostředek OS pro procesy
- je nutné vzájemné vylučování
 - dochází k výměně informací
 - zajišťuje OS
- systémová volání
 - **send**(cíl, zpráva)
 - **receive**(zdroj, zpráva)

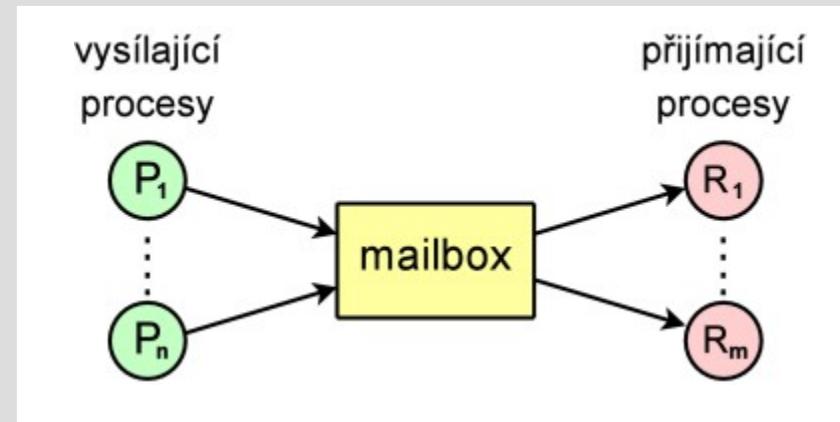
Předávání zpráv – adresace

- adresace cíle může být pro **send** příp. i **receive**
 - **přímá** – adresujeme cílový proces
 - s případnou možností určovat obecné adresy
 - **nepřímá** – adresujeme frontu zpráv (mailbox)
 - různé procesy pak mohou zprávy vyzvedávat
 - implementace může umožňovat také vybírání jen určitých typů zpráv
- **receive** může dostat parametrem hodnotu, pomocí které potvrdí přijetí
 - umožňuje zjistit doručení při neblokujícím **send**

Předávání zpráv – mailbox a port

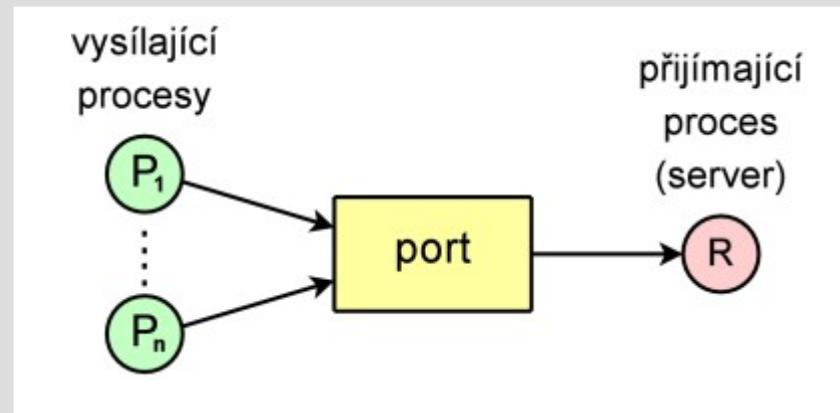
- **mailbox**

- je vlastněn párem
vysílač / přijímač
- může být sdílen více
vysílači / přijímači



- **port**

- svázán s jediným
přijímačem – serverem
- model klient / server



Předávání zpráv – (ne)blokování

- možné implementace blokování **send** a **receive**
 - **neblokující send**
 - nečeká se na doručení (vyzvednutí zprávy adresátem)
 - může blokovat při zaplnění fronty
 - **blokující send**
 - čeká se na vyzvednutí zprávy adresátem
 - **neblokující receive**
 - není-li dostupná žádná zpráva, vrací receive chybu
 - **blokující receive**
 - adresát je blokován, dokud není dostupná zpráva
- **blokující send a blokující receive – rendezvous**

Předávání zpráv – (ne)blokování, typická implementace

- typická implementace blokování **send** a **receive** pro fronty zpráv s omezenou velikostí
 - **neblokující send**
 - blokuje pouze při zaplnění fronty zpráv
 - **blokující receive**
 - příjemce je blokován, není-li dostupná žádná zpráva
 - neblokující varianty lze nastavit parametrem, volání pak místo blokování vrací chybu

Předávání zpráv – řešení KS

- sdílená fronta **mutex** se inicializuje zasláním jedné zprávy
- vstupní sekce volá **blokující receive**
- výstupní sekce volá **neblokující send**
- první proces, který provede **receive**, se dostane do KS, ostatní jsou blokovány

```
jeden z procesů  $P_i$ ,  
send(mutex, "go");  
proces  $P_i$ ,  
repeat  
    receive(mutex, & $msg$ );  
    KS;  
    send(mutex,  $msg$ );  
    ZS;  
forever
```

Předávání zpráv – synchronizace

- příkaz **S2** v procesu **P₁** musí být proveden až **po provedení S1** v **P₀**
- vyprázdníme mailbox **sync**
- **P₁** před provedením **S2** volá **blokující receive**
- **P₀** po provedení **S1** volá **neblokující send**
- provede-li se nejprve **send**, **receive** neblokuje

inicializace

```
// vyprázdnění fronty zpráv
```

proces P₀

```
vypočti x; // S1
```

```
send(sync, msg);
```

proces P₁

```
receive(sync, &msg);
```

```
použij x; // S2
```

Producent / konzument – fronta zpráv s omezenou kapacitou

- do fronty zpráv **storage** zasílají producenti položky
 - jedná se o sklad (buffer)
- kapacita skladu je daná maximální velikostí fronty
 - zaplní-li producenti sklad, bude send blokovat
- konzumenti vybírají položky voláním **receive**
- funguje pro více producentů i konzumentů

Producent

repeat

```
msg = produce_item();  
send(storage, msg);
```

forever

Konzument

repeat

```
receive(storage, &msg);
```

```
consume_item(msg);
```

forever

Zprávy System V – IPC

- alokace fronty – msgget(2)

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>

int msgget(key_t key, int msgflg);
```

- vrátí identifikátor fronty zpráv podle klíče **key**,
při chybě vrací -1 (a nastaví **errno**)
- nová fronta zpráv je vytvořena, pokud
 - **key** = **IPC_PRIVATE** (lepší název by byl **IPC_NEW**) nebo
 - **key** ≠ **IPC_PRIVATE**, fronta sdružená s klíčem **key**
neexistuje a **msgflg** obsahuje **IPC_CREAT**

Zprávy System V – alokace

- alokace (pokračování) – msgget(2)

```
int msgget(key_t key, int msgflg);
```

- struktura `msqid_ds` je naplněna – viz msgctl(2)
 - vlastník a skupina podle volajícího procesu, práva dle `msgflg`, čas modifikace na aktuální, maximální velikost fronty na **MSGMNB** a zbytek parametrů je vynulován
 - práva lze specifikovat dolními 9 bity v `msgflg`
 - je-li dáno **IPC_CREAT** a **IPC_EXCL** a fronta existuje, vrací chybu (**EEXIST**)

Zprávy System V – zaslání zprávy

- poslání zprávy do fronty zpráv – msgop(2)

```
void *msgsnd(int msqid, const void *msgp,  
size_t msgsz, int msgflg);
```

- zkopíruje zprávu do fronty s id **msqid**
 - **msgp** je ukazatel na strukturu, v níž **mtext** je pole o velikosti **msgsz** – dovolena je i nulová velikost

```
struct msgbuf {  
    long mtype;           // typ zprávy; musí být > 0  
    char mtext[1];         // obsah zprávy  
}
```

- je-li ve frontě dost místa, je volání neblokující
 - není-li místo, blokuje dle **IPC_NOWAIT** v **msgflg**

Zprávy System V – přijetí zprávy

- přijetí zprávy z fronty zpráv – msgop(2)

```
ssize_t msgrcv(int msqid, void *msgp,  
    size_t msgsz, long msgtyp, int msgflg);
```

- zkopíruje zprávu z fronty **msqid** na adresu **msgp**
- **msgsz** udává maximální velikost položky **mtext**
 - je-li zpráva větší, rozhoduje **MSG_NOERROR** v **msgflg** – bud' se zpráva zkrátí (část je ztracena) nebo vrací chybu
- **msgtyp** určuje, které zprávy se mají vybírat
 - 0 – první ve frontě, > 0 – první daného nebo jiného typu (**MSG_EXCEPT** v **msgflg**), < 0 – první nejmenší typ hodnoty $\leq |msgtyp|$

Zprávy System V – ovládání fronty

- ovládání front zpráv – msgctl(2)

```
int msgctl(int msqid, int *cmd,  
           struct msqid_ds *buf);
```

- vykoná příkaz **cmd** na frontě zpráv
 - **IPC_SET** – nastavení velikosti fronty a oprávnění
 - **IPC_RMID** – okamžité odstranění fronty
 - **IPC_STAT** – zjištění atributů (naplní strukturu **msqid_ds**)
- datová struktura **msqid_ds** obsahuje
 - oprávnění (vlastník, skupina, práva), časy (poslední zaslání, přijetí, změna), počet zpráv ve frontě, maximální velikost fronty v bajtech, PID (poslední zaslání, přijetí)

Zprávy System V – data fronty

- datová struktura `msqid_ds` – `msgctl(2)`

```
struct msqid_ds {  
    struct ipc_perm msg_perm;      // oprávnění (vlastnictví a práva)  
    time_t msg_stime;            // čas posledního msgsnd(2)  
    time_t msg_rtime;            // čas posledního msgrcv(2)  
    time_t msg_ctime;            // čas posledního změny  
    msgqnum_t msg_qnum;          // počet zpráv ve frontě  
    msglen_t msg_qbytes;         // maximální velikost fronty  
    pid_t msg_lspid;             // PID posledního msgsnd(2)  
    pid_t msg_lrpid;             // PID posledního msgrcv(2)  
};  
  
struct ipc_perm {  
    key_t __key;                // klíč předaný msgget(2)  
    uid_t uid, gid, __cuid, __cgid; // UID/GID vlastníka a tvůrce  
    unsigned short mode;         // práva  
    unsigned short __seq;        // pořadové číslo  
};
```

Posixové fronty zpráv

- fronty zpráv – mq_overview(7)

```
#include <mqueue.h>
```

- knihovna librt – nutnost linkování
`gcc ... -lrt ...`
 - lepší design než původní IPC UNIX System V
 - nemusejí být implementovány všude
 - Linux od verze jádra 2.6.6 (2004), glibc od verze 2.3.4

- Linux

- rozhraní proc: /proc/sys/fs/mqueue/
 - lze připojit i virtuální FS typu mqueue

Zprávy POSIX – vytvoření fronty

- vytvoření nebo otevření fronty – mq_open(3)
`mqd_t mq_open(const char *name, int oflag);`
 - vytvoří / otevře posixovou frontu zpráv – jako open(2)
 - jméno fronty je tvaru /jméno
 - **oflag** – způsob otevření (**O_CREAT**, **O_RDWR**, ...)
 - obsahuje-li **oflag O_CREAT**, přidat 2 parametry
`mqd_t mq_open(const char *name, int oflag,
mode_t mode, struct mq_attr *attr);`
 - **mode** – přístupová práva (**S_IRWXU**, **S_IRUSR**, ...) – respektuje se nastavení umask(2)
 - **attr** může být **NULL** (výchozí hodnoty), viz mq_getattr(3)
 - vrací **queue descriptor** nebo (**mqd_t**) -1 při chybě

Zprávy POSIX – atributy

- zjištění / nastavení atributů – mq_getattr(3)

```
mqd_t mq_getattr(mqd_t mqdes,  
                  struct mq_attr *attr);  
  
mqd_t mq_setattr(mqd_t mqdes, struct mq_attr  
                  *newattr, struct mq_attr *oldattr);  
  
struct mq_attr {  
    long mq_flags;           // 0 nebo O_NONBLOCK  
    long mq_maxmsg;         // max. počet zpráv ve frontě (10)  
    long mq_msgsize;        // max. velikost zprávy v bajtech (8 KiB)  
    long mq_curmsgs;        // aktuální počet zpráv ve frontě  
};
```

- mq_setattr smí modifikovat pouze mq_flags
- vrací: 0 = úspěch, -1 = chyba, errno udává chybu

Zprávy POSIX – uzavření, odpojení

- uzavření fronty – `mq_close(3)`
`mqd_t mq_close(msq_t mqdes);`
 - uzavře posixovou frontu zpráv deskriptoru `mqdes`
 - ruší případnou registraci procesu na upozornění
- uvolnění prostředků – `mq_unlink(3)`
`mqd_t mq_unlink(const char *name);`
 - podobné `unlink(2)` – odstraní jméno fronty
 - prostředky fronty jsou uvolněny, až všechny procesy frontu uzavřou
 - vrací: 0 = úspěch, -1 = chyba, `errno` udává chybu

Zprávy POSIX – zaslání

- zaslání zprávy – `mq_send(3)`

```
mqd_t mq_send(mqd_t mqdes, const char *msg_ptr,  
size_t msg_len, unsigned msg_prio);
```

- vloží do fronty `mqdes` zprávu na adrese `msg_ptr`
délky $0 \leq \text{msg_len} \leq \text{mq_msgsize}$ (atribut fronty)
- řazení do fronty je podle priority `msg_prio`
 - vyšší hodnota = vyšší priorita
- volání je blokující, pokud je fronta plná a `mq_flags`
(atribut fronty) neobsahuje `O_NONBLOCK`
- vrací: 0 = úspěch, -1 = chyba, `errno` udává chybu

Zprávy POSIX – přijetí

- přijetí zprávy – `mq_receive(3)`

```
ssize_t mq_receive(mqd_t mqdes, char *msg_ptr,  
size_t msg_len, unsigned *msg_prio);
```

- zkopíruje z fronty `mqdes` zprávu na adresu `msg_ptr`
max. délky `msg_len` \geq `mq_msgsize` (atribut fronty)
- není-li `msg_prio` `NULL`, je vrácena i priorita
- volání je blokující, pokud je fronta prázdná a
`mq_flags` (atribut fronty) neobsahuje `O_NONBLOCK`
- vrací: délku zprávy, $-1 =$ chyba, `errno` udává chybu

Zprávy POSIX – operace s limitem

- zaslání / přijetí s limitem – mq_timedsend(3)

```
mqd_t mq_timedsend(mqd_t mqdes, const char
                     *msg_ptr, size_t msg_len, unsigned msg_prio,
                     const struct timespec *abs_timeout);

ssize_t mq_timedreceive(mqd_t mqdes, char
                        *msg_ptr, size_t msg_len, unsigned *msg_prio,
                        const struct timespec *abs_timeout);
```

- volání blokuje nejvýše po dobu podle **abs_timeout**
 - timeout je absolutní čas od Epochy (doba od 1. 1. 1970)

```
struct timespec {
    time_t      tv_sec;           // sekundy
    long        tv_nsec;          // nanosekundy
};
```

Zprávy POSIX – upozornění (1)

- upozornění na neprázdnou frontu – mq_notify(3)

```
mqd_t mq_notify(mqd_t mqdes,
                 const struct sigevent *notification);

struct sigevent {
    int sigev_notify;                  // metoda upozornění
    int sigev_signo;                  // upozorňující signál
    union sigval sigev_value;         // data předaná při upozornění
    void (*sigev_notify_function) (union sigval);           // funkce pro upozornění vláknem
    void *sigev_notify_attributes; // atributy vláknové funkce
};

union sigval {                      // hodnota předaná při upozornění
    int sigval_int;
    void *sigval_ptr;
};
```

Zprávy POSIX – upozornění (2)

- upozornění na neprázdnou frontu – mq_notify(3)

```
mqd_t mq_notify(mqd_t mqdes,  
const struct sigevent *notification);
```

 - upozorní proces přijde-li zpráva do prázdné fronty
 - registrace je po upozornění zrušena
 - **sigev_notify** nastavuje typ upozornění
 - **SIGEV_NONE** – pouze registrace procesu, bez upozornění
 - **SIGEV_SIGNAL** – upozornění signálem **sigev_signo**
 - **SIGEV_THREAD** – upozornění vytvořením vlákna
 - je-li **notification NULL**, ruší se registrace procesu
 - pouze jediný proces se smí registrovat

Monitory

(řešení problémů souběhu)

- koncept: B. Hansen 1972, C. A. R. Hoare 1974
- konstrukce ve vyšším programovacím jazyce pro stejné služby jako semafory (synchronizace a vzájemné vylučování), snadněji ovladatelné
- vyskytují se v řadě jazyků pro souběžné programování (concurrent programming)
 - Concurrent Pascal, Modula-3, uC++, Java, C#
 - implementace (Java, C#) se liší od původního konceptu
- mohou být implementovány pomocí semaforů

Monitor (nástroj pro řešení problémů souběhu)

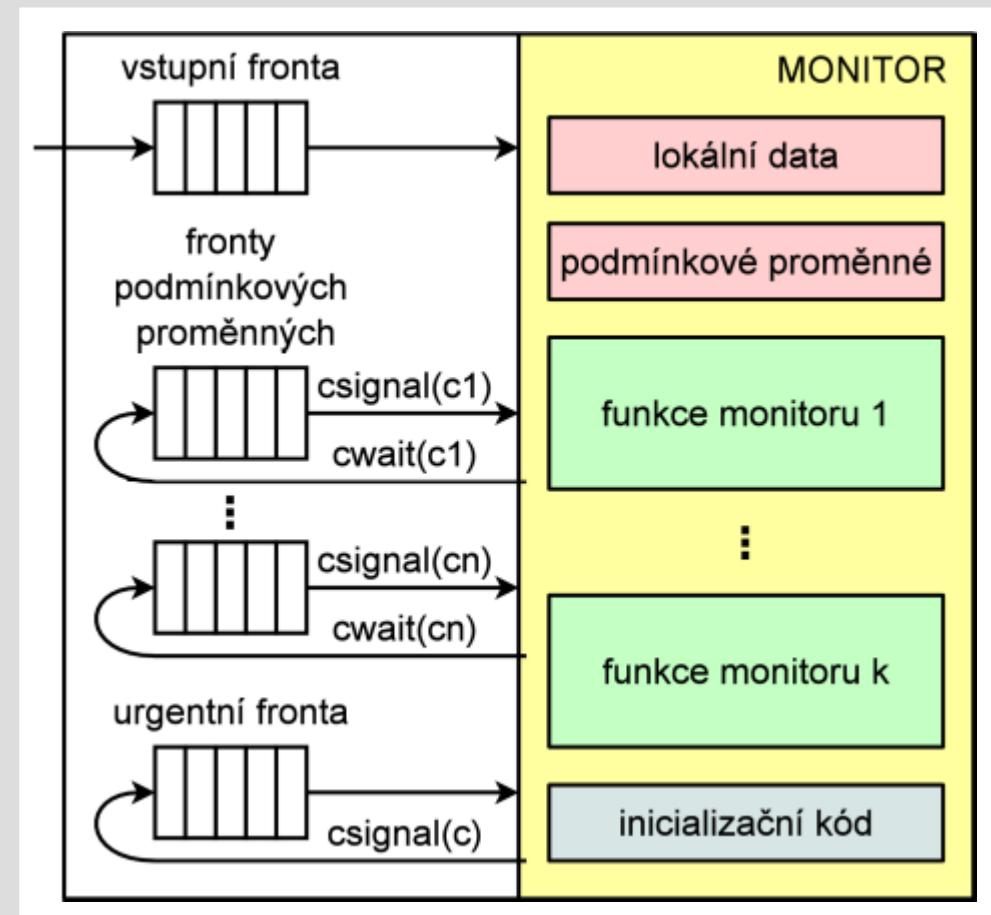
- **SW modul** (podobný objektu / třídě)
 - lokální proměnné – sdílená data
 - tato data nejsou viditelná vně monitoru
 - funkce zpřístupňující lokální data
 - inicializační část
- **v monitoru** (jeho funkci) **smí být** v daném okamžiku **pouze jediné vlákno**
 - monitor tak zajišťuje **vzájemné vylučování**
 - **synchronizaci** lze zajistit podmínkovými proměnnými

Monitor – podmínkové proměnné

- synchronizační nástroj monitoru
- podmínkové proměnné jsou lokální v monitoru a dostupné pouze pomocí funkcí monitoru
- lze je měnit pouze dvěma funkciemi monitoru
 - **cwait(cv)** – blokuje vlákno, dokud není zavoláno:
 - **csignal(cv)** – obnoví provádění vlákna blokovaného podmínkovou proměnnou cv
 - je-li takových vláken více, vybere se jedno z nich
 - **není-li žádné takové vlákno, neprovede se nic**

Monitor (obrázek)

- čekající vlákna jsou ve frontě vstupní nebo podmínkové
- provedením cwait(c) se vlákno zařadí do podmínkové fronty
- **csignal(c)** aktivuje jedno čekající vlákno z podmínkové fronty
 - **csignal** blokuje (není-li to poslední příkaz ve funkci)



Monitor a Java

- implementace se odlišuje, je třeba úprav:
 - lokální proměnné je třeba deklarovat jako privátní
 - všechny metody je třeba deklarovat **synchronized**
 - Java umožňuje způsobit výjimku (přerušení) v KS!
 - ošetření výjimky KS – uvést data do konzistentního stavu
 - existuje jediná anonymní podmínková proměnná ovládaná pomocí wait() a notify() či notifyAll()
 - vstupní ani podmínková „fronta“ není FIFO
- HANSEN, Per Brinch: Java's Insecure Parallelism. In: ACM SIGPLAN Notices [online]. 1999, č. 34, 38–45. [cit. 2012-11-12]. ISSN 0362-1340. Odkaz: <[PBHansenParalelism.pdf](#)>.

Monitor a Java 5 (2004)

- přidány nástroje pro souběžné programování
 - semafor, bariéra, synchronizační nástroje
 - monitor s podmínkovými proměnnými lze napodobit pomocí zámku

```
final Lock monitor = new ReentrantLock();
final Condition c1 = monitor.newCondition();
final Condition c2 = monitor.newCondition();
monitor.lock();
try { // kritická sekce, může generovat výjimku přerušení
    c1.await();      // čekání na podmínu
    c2.signal();     // signalizace, probudí čekající vlákno
}
finally { monitor.unlock(); }
```

Monitor a C# (.NET Framework)

- implementace se odlišuje
 - zámek (KS) je použit na (privátní) objekt
 - použito příkazem **lock (obj) { ... }**
 - vstup do KS: Monitor.Enter(obj) před blokem / v bloku **try**
 - výstup z KS: Monitor.Exit(obj) v bloku **finally**
 - čekání a signalizace, jediná podmínková proměnná: Monitor.Wait() a Monitor.Pulse() či Monitor.PulseAll()
 - nečeká-li žádné vlákno, signál je ztracen
 - podmínková fronta nemá přednost před tou vstupní
 - do KS může vstoupit vlákno a změnit signalizovaný stav

Monitor a C# – rozdíly ve verzích a možné problémy – C# 3.0

- implementace **lock (obj) { body; }**
 - C# 3.0

```
var temp = obj;
Monitor.Enter(temp);
// instrukce no-op + výjimka zde = neuvolněný zámek → DEADLOCK
try { body; } // critical section
finally { Monitor.Exit(temp); }
```
 - korektnost závisí na (ne)optimalizaci kompilátoru
 - vloží-li kompilátor před blok try instrukci no-op, může během ní dojít k vyvolání výjimky (thread abort exception) a zámek zůstane zamčený
 - důsledek: možný **DEADLOCK**

Monitor a C# – rozdíly ve verzích a možné problémy – C# 4.0

- implementace **lock** (*obj*) { *body*; }
- C# 4.0

```
bool lockWasTaken = false;
var temp = obj;
try { Monitor.Enter(temp, ref lockWasTaken); body; }
finally { if (lockWasTaken) Monitor.Exit(temp); }
```
- lepší řešení zámku?
 - nekonzistentní stav v *body* + výjimka (přerušení) → uvolnění zámku = **zpřístupnění nekonzistentních dat!**
 - **ošetření výjimky KS** – uvést data do konzistentního stavu
 - Lippert, Eric: *Locks and exceptions do not mix* [online]. 2009 [cit. 2015-11-25]. Odkaz: <[Eric Lippert's Blog](#)>.

Producent / konzument – monitor

Producent

repeat

```
    item = produce_item();  
    PCmon.append(item);
```

forever

Konzument

repeat

```
    PCmon.take(&item);  
    consume_item(item);
```

forever

Monitor: proměnné, init

```
monitor PCmon {  
    item_t buffer[BUF_SIZE];  
    cond not_full, not_empty;  
    int in=0, out=0, count=0;
```

```
void append(item_t item) {  
    while (count == BUF_SIZE)  
        cwait(not_full);  
    buffer[in] = item;  
    in = (in+1)%BUF_SIZE;  
    count++;  
    csignal(not_empty);  
}  
  
void take(item_t *item) {  
    while (count == 0)  
        cwait(not_empty);  
    *item = buffer[out];  
    out = (out+1)%BUF_SIZE;  
    count--;  
    csignal(not_full);  
}
```

Producent / konzument – monitor – while vs. if

- stačilo by použití **if** místo **while**?
 - po probuzení signálem je monitor volný jak pro probuzené vlákno, tak i pro jiná vlákna ve vstupní frontě
 - záleží na implementaci přednosti podmínkové a vstupní fronty

```
void append(item_t item) {  
    while (count == BUF_SIZE)  
        cwait(not_full);  
    buffer[in] = item;  
    in = (in+1)%BUF_SIZE;  
    count++;  
    csignal(not_empty);  
}  
  
void take(item_t *item) {  
    while (count == 0)  
        cwait(not_empty);  
    *item = buffer[out];  
    out = (out+1)%BUF_SIZE;  
    count--;  
    csignal(not_full);  
}
```

Komunikace procesů – roura

- roura (pipe) – POSIX.1
 - jednosměrný komunikační nástroj pro dva procesy
 - FIFO
 - nezávislé ukazatele pozice pro čtení a zápis
 - pozice lze měnit pouze čtením/zápisem
 - co bylo přečteno se odstraní
 - jeden proces zapisuje, druhý čte
 - dokud není roura otevřena druhou stranou, operace na ní blokují

Roura – vytvoření

- vytvoření roury – pipe(2)

```
#include <unistd.h>  
int pipe(int filedes[2]);
```

- vytvoří pár deskriptorů propojených rourou
 - jeden pro čtení: `filedes[0]`
 - jeden pro zápis: `filedes[1]`
- vrací: 0 = úspěch, -1 = chyba, `errno` udává chybu

Komunikace procesů – socket

- **socket** (někdy též **soket**) – POSIX.1
 - obousměrný komunikační nástroj pro dva procesy
 - domény (rodiny) socketů
 - UNIX domain – lokální
 - IPv4 / IPv6 – síťový nad IP
 - další síťové: IPX, X.25, AX.25, ATM, Appletalk, ...
 - typy socketů:
 - STREAM – proudový spojovaný (nad IP odpovídá TCP)
 - DGRAM – datagramový nespojovaný (UDP)
 - a jiné: sekvenční (SEQPACKET), surový (RAW), ...

Socket – vytvoření páru propojených deskriptorů

- vytvoření páru deskriptorů – socketpair(3)

```
#include <sys/types.h>
#include <sys/socket.h>

int socketpair(int d, int type, int protocol,
               int sv[2]);
```

- vytvoří pár deskriptorů **sv[]** propojených socketem
 - oba pro obousměrnou komunikaci: čtení i zápis
 - doména **d**: **AF_UNIX**, **AF_INET**, **AF_INET6** aj.
 - **type**: **SOCK_STREAM**, **SOCK_DGRAM** aj.
 - **protocol**: **0** znamená výchozí pro daný typ socketu
- vrací: **0** = úspěch, **-1** = chyba, **errno** udává chybu

Socket – klient

- klientský proces a socket
 - překlad adresy – **getaddrinfo(3)** / **gethostbyname(3)**
 - alokace socketu – **socket(2)**
 - svázání socketu s lokálním portem – **bind(2)**
 - pouze volitelně, jinak OS přiřadí port automaticky
 - navázání spojení (proudový socket) – **connect(2)**
 - pro datagramový socket nastaví jen výchozí adresu cíle
 - komunikace – **write(2)** / **send(2)**, **read(2)** / **recv(2)**
 - pro datagramový **sendto(2)** a **recvfrom(2)**
 - zavření – **close(2)**

Socket – server

- serverový proces a socket
 - nastavení lokální adresy a portu – **getaddrinfo(3)**
 - alokace socketu – **socket(2)**
 - svázání socketu s lokálním portem – **bind(2)**
 - zahájení naslouchání – **listen(2)**
 - přijetí spojení (pouze proudový socket) – **accept(2)**
 - vytvoří nový socket pro komunikaci s klientem
 - komunikace – **read(2)** / **recv(2)**, **write(2)** / **send(2)**
 - pro datagramový **recvfrom(2)** a **sendto(2)**
 - zavření – **close(2)**

Prostředky komunikace procesů

- prostředky komunikace
 - soubor, databáze
 - pomalé, náhodný přístup, současný přístup je třeba řídit
 - roura
 - proudový přístup (FIFO), jednosměrná komunikace
 - socket
 - proudový přístup (FIFO), obousměrná síťová komunikace
 - fronty zpráv
 - exkluzivní přístup (FIFO), (jednosměrná) komunikace
 - sdílená paměť
 - nejrychlejší, náhodný přístup, současný přístup nutno řídit

OS – Deadlock a prevence

Tomáš Hudec

Tomas.Hudec@upce.cz

<http://asuei01.upceucebny.cz/usr/hudec/vyuka/os/>

Prostředky poskytované OS

- OS poskytuje procesům systémové **prostředky** (zdroje, resources)
 - HW zařízení, soubory, semafory, (sdílená) paměť
- zajímají nás prostředky s exkluzivním přístupem
- prostředky můžeme rozdělit na
 - **odejmutelné** (preemptable)
 - např. CPU, operační paměť (lze odložit na swap)
 - **neodejmutelné** (non-preemptable)
 - např. DVD-RW (při vypalování), tiskárna

Řízení přístupu k prostředkům

- k některým prostředkům si mohou přístup řídit samy procesy
 - např. pomocí semaforů
- příklad – prostředky A a B, procesy P_1 a P_2
 - oba prostředky jsou vyžadované současně
 - prostředky alokujeme postupně
 - alokujeme semafor pro každý prostředek
- **záleží na pořadí alokace!**

Řízení přístupu – příklad

- uvažujme postupnou alokaci pro každý proces
 - fungující příklad
- pokud ovšem alokace proběhne u některého procesu v opačném pořadí
 - může nastat **DEADLOCK**

proces P₁

```
mutex_lock (&mA) ; // získá A  
mutex_lock (&mB) ; // čeká na B  
použij_prostředky (A, B) ;  
mutex_unlock (&mB) ;  
mutex_unlock (&mA) ;
```

proces P₂

```
mutex_lock (&mB) ; // získá B  
mutex_lock (&mA) ; // čeká na A  
použij_prostředky (A, B) ;  
mutex_unlock (&mA) ;  
mutex_unlock (&mB) ;
```

Deadlock (stav uváznutí)

- vzájemné zablokování, smrtící objetí, uváznutí
- definice **stavu uváznutí** (zablokování)
 - skupina procesů je ve **stavu uváznutí**, když **každý proces** ve skupině **čeká na událost**, kterou může vyvolat **pouze jiný proces ze skupiny**
 - protože událost nelze vyvolat jinak než procesem ve skupině, budou tyto procesy nekonečně čekat
 - předpokládáme, že procesy nelze probudit asynchronně
 - např. signálem, při jehož obsluze by se mohla událost vyvolat

Livelock

- procesy běží, ale nečiní žádný pokrok
- procesy mohou reagovat na stav jiného procesu, ale tyto změny nevedou k pokroku
- na rozdíl od stavu deadlock **procesy nejsou ve stavu čekání**
- př.: dva lidé potkavše se v úzké uličce
 - oba neustále současně uhýbajíce na stejnou stranu
- definice stavu livelock se v pramenech liší

Starvation (vyhladovění)

- nekončící čekání procesu na přidělení prostředku, proces nečiní žádný pokrok
 - proces v soupeření o prostředek neustále prohrává
 - např. nízká priorita při soutěži o procesorový čas
- nejedná se o deadlock
 - proces by učinil pokrok, kdyby jej ostatní procesy neustále „nepředbíhaly“
 - v jistém smyslu lze však uvažovat, že důsledkem stavu uváznutí (deadlock / livelock) je vyhladovění

Grafické modelování požadavků

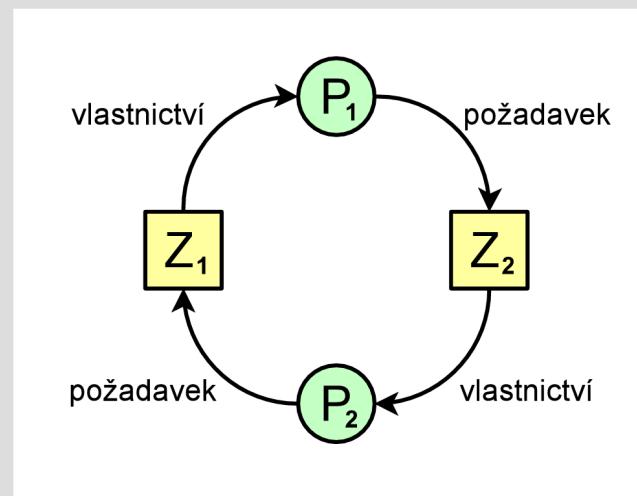
- prostředky reprezentujeme pravoúhelníky
- procesy reprezentujeme kruhy
- alokaci (držení) prostředku Z procesem P znázorňujeme šipkou od Z k P
- blokující požadavek procesu R na přidělení prostředku Y znázorňujeme šipkou od R k Y



Coffmanovy podmínky vzniku stavu uváznutí (publ. 1971)

- vzájemné vylučování (mutual exclusion)
 - prostředky lze vlastnit pouze jediným procesem
- alokace a čekání (hold and wait)
 - proces vlastnící prostředek může požadovat další
- neodnímatelné prostředky (non-preemptable)
 - OS je nemůže odejmout, musí být explicitně uvolněny vlastnícím procesem
- cyklické čekání (circular wait)
 - řetěz vzájemně čekajících procesů uzavírá cyklus

Cyklické čekání (obrázek)



- proces P_1 vlastní prostředek Z_1
a je blokován požadavkem na prostředek Z_2 ,
- proces P_2 vlastní prostředek Z_2
a je blokován požadavkem na prostředek Z_1 ,

Řešení a prevence stavu uváznutí

- ignorování problému (pštrosí přístup)
- detekce a obnovení
 - obnova (rollback) stavu bez uváznutí (checkpoint)
 - násilné odebrání prostředku
 - zabití některého procesu (př.: Linux OOM-killer)
- vyloučení možnosti vzniku (**prevence**)
 - negování alespoň jedné z předešlých podmínek
 - splnění **všech** předešlých podmínek je **nutné**, aby vznikl deadlock – **stačí vyloučit jedinou**

Prevence vzniku uváznutí (1)

- negace vzájemného vylučování
 - spooling (např. tiskárna)
- negace alokace a čekání (hold and wait)
 - zajistit alokaci všech potřebných prostředků naráz
 - proces je blokován, dokud vše není dostupné
 - proces pak může čekat velmi dlouho
 - některé prostředky zůstávají dlouho nevyužity
- negace neodnímatelných prostředků
 - zavést možnost násilného odebrání prostředku
 - nelze vždy – např. vypalování médií (DVD)

Prevence vzniku uváznutí (2)

- negace cyklického čekání (circular wait)
 - nedovolit alokaci prostředku, pokud by vznikl cyklus
 - definování lineárního uspořádání na prostředcích
 - přidělují se vždy pouze prostředky s vyšším pořadovým číslem než má jakýkoli procesu již přidělený prostředek
 - může nastat zbytečné odmítnutí požadavku na přidělení

Prevence vzniku uváznutí (3)

- dovolení pouze bezpečných stavů
 - bezpečný stav
 - není stav uváznutí a **existuje plánovací pořadí**, při kterém všechny procesy mohou být dokončeny, i když všechny procesy naráz budou požadovat **maximum svých** potřebných zdrojů (prostředků)
 - nebezpečný stav
 - stav, kdy může (ale nemusí vždy) nastat uváznutí
 - pokud by každý proces požadoval maximum svých deklarovaných zdrojů a ani jednomu nebude moci systém vyhovět, jedná se o nebezpečný stav

Bankéřův algoritmus

- algoritmus publikoval Dijkstra 1965
- řešení situace bankéře při jednání s klienty, kterým poskytuje půjčku
 - pokud požadavek klienta vede k nebezpečnému stavu, je tento požadavek odmítnut
- výchozí předpoklady
 - pevný počet prostředků
 - každý proces deklaruje své maximální požadavky
 - postupná alokace prostředků

Bankéřův algoritmus

- prostředek je na požadavek přidělen jen tehdy, vede-li situace do bezpečného stavu
- slabiny algoritmu
 - nelze vždy garantovat pevný počet prostředků
 - proces nemusí znát maximální požadavky předem
- důsledek: algoritmus je mnohdy prakticky nepoužitelný

Prevence uváznutí – shrnutí

podmínka

vzájemné vylučování
alokace a čekání
neodnímatelné prostředky
cyklické čekání

možné řešení

spooling prostředků
alokovat vše naráz
dovolit násilné odebrání
uspořádání prostředků,
bankéřův algoritmus

OS – Správa paměti

Tomáš Hudec

Tomas.Hudec@upce.cz

<http://fei-as.upceucebny.cz/usr/hudec/vyuka/os/>

Operační paměť

- jeden z nejdůležitějších prostředků spravovaných operačním systémem
- procesy pro svůj běh potřebují paměť
- efektivní správa nutnosti
- ochrana paměti
 - jádra OS před procesy
 - paměť procesu před ostatními procesy

Požadavky na operační paměť

- **relokace** paměti procesu
- **ochrana** paměti
- možnost **sdílení** paměti mezi procesy
- logická a fyzická organizace
 - logické členění paměti (procesu)
 - adresace (logické adresy → fyzické adresy)

Relokace paměti procesu

- přemístění paměti procesu na jiné adresy
- program nesmí pracovat s absolutními adresami fyzické operační paměti
 - programátor nemůže vědět, na které adresy je program do paměti zaveden
 - proces může být pozastaven a jeho paměť odložena na disk (swap)
- proces musí pracovat s logickými adresami
 - za běhu se převedou na skutečné fyzické adresy

Ochrana paměti

- procesy nesmějí přímo přistupovat k paměti
 - jádra OS
 - ostatních procesů
- kontrola přístupu musí být prováděna v době běhu procesu (run-time access control)
 - nelze provést podle kódu programu
- ochrana může mít různé stupně
 - čtení, zápis, provádění

Sdílení paměti mezi procesy

- procesy mohou potřebovat sdílet paměť
 - nutné pro efektivnější provádění procesů
 - soubory nebo jiné mechanismy výměny dat mohou být pomalé
- výhodné je sdílet paměť mezi procesy, které provádějí stejný program (knihovny)
 - kód procesu (program) je v paměti sdílen
 - vede k výrazné úspoře paměti
 - méně časté odkládání paměti na disk (swap)

Logická organizace paměti

- programy jsou obvykle psány modulárně
 - např. program, knihovny
- moduly vyžadují různý stupeň ochrany paměti
 - pouze pro čtení, zápis, pouze provádění
- moduly lze pro zvýšení efektivity sdílet
 - dynamicky připojované knihovny

Fyzická organizace omezeného množství paměti

- fyzická (skutečná) paměť nemusí vždy stačit
- techniky pro umožnění běhu procesů, které vyžadují více paměti, než je dostupné
 - **překrývání (overlaying)**
 - různé moduly programu nejsou vyžadovány současně
 - moduly používají stejnou fyzickou oblast paměti
 - **swapping**
 - odkládání procesu z operační paměti na sekundární
 - využití levné sekundární paměti (disku)

Techniky přidělování paměti

- historické
 - pevné dělení (fixed partitioning)
 - paměť je rozdělena na pevně definované oblasti
 - dynamické dělení
 - paměť je přidělována dle požadavků procesů
- moderní
 - lze alokovat paměť pouze pro část procesu
 - segmentace (segmentation) – víceméně na ústupu
 - stránkování (paging)

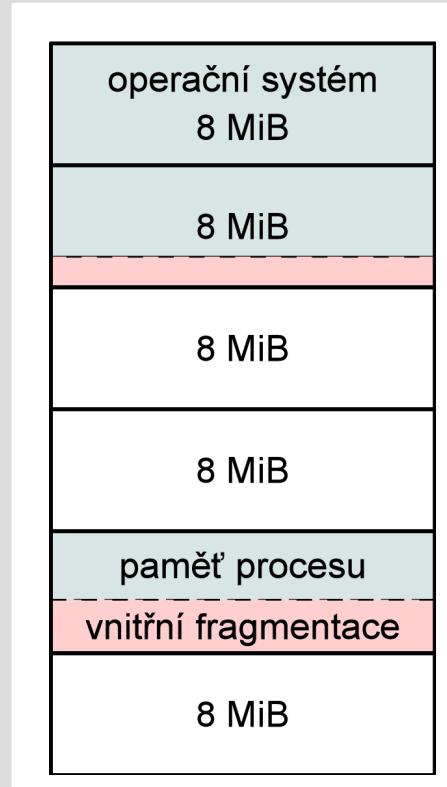
Pevné dělení paměti

- fixed partitioning
- dostupná paměť je rozdělena do oblastí (partitions) s pevnými hranicemi
- stejná velikost všech oblastí
 - proces je zaveden do libovolné volné oblasti
 - jsou-li všechny oblasti obsazeny, lze některý proces odložit na disk (swap out)
 - nevejde-li se proces do oblasti, musí programátor použít techniku překrývání (overlays)

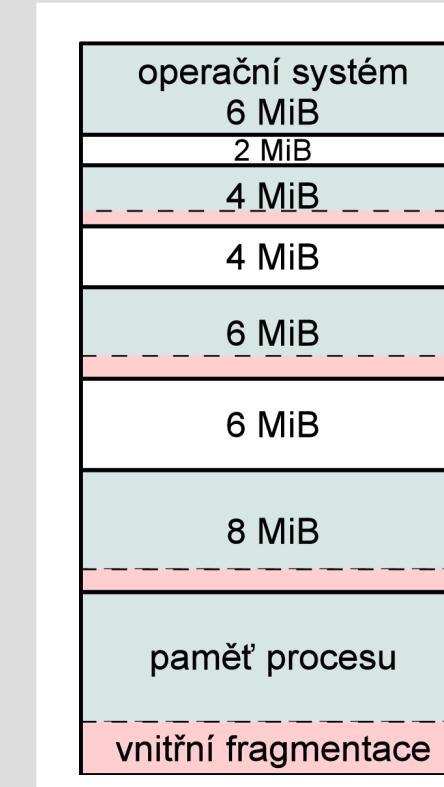
Pevné dělení paměti

- stejná velikost všech oblastí
 - jakýkoli malý proces zabere celou oblast
 - neefektivní využití paměti
 - **vnitřní fragmentace** (internal fragmentation) – proces nevyužije veškerou přidělenou paměť
- nestejná velikost oblastí
 - redukuje se vnitřní fragmentace
 - ale neodstraňuje se úplně

Pevné dělení (obrázek)



stejná velikost oblastí



nestejná velikost oblastí

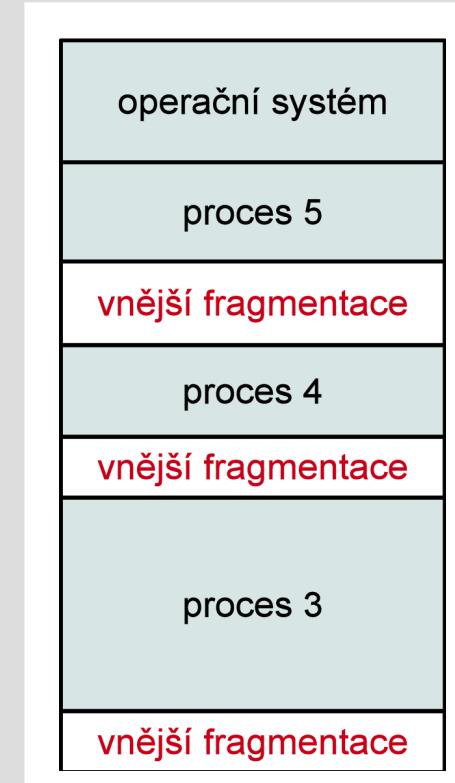
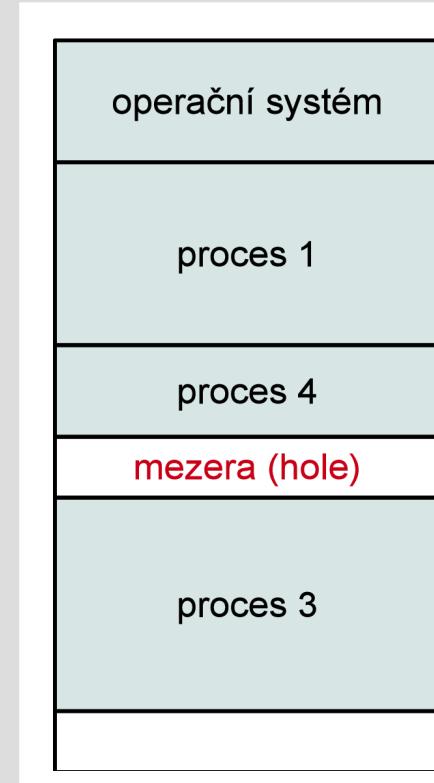
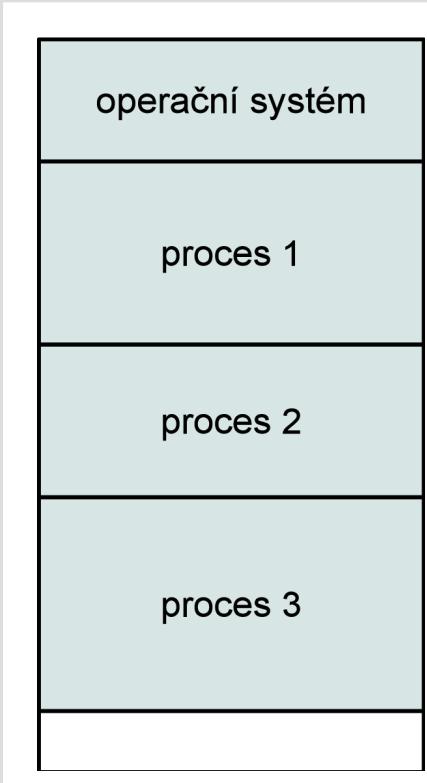
Umist'ovací algoritmus

- stejná velikost oblastí – libovolná volná oblast
- nestejná velikost oblastí
 - procesy čekající na přidělení paměti tvoří frontu
 - fronta samostatná pro každou velikost oblasti
 - fronta se pro proces vybírá tak, aby se minimalizovala vnitřní fragmentace – nejmenší oblast, kam se vejde
 - blokuje procesy, které by mohly být ve větší volné oblasti
 - fronta jediná
 - přidělí se nejmenší volná oblast, do které se proces vejde
 - rychlejší, ale za cenu větší vnitřní fragmentace

Dynamické dělení paměti

- proměnný počet oblastí i jejich velikost
- proces dostane tolik paměti, kolik potřebuje
 - **odstraněna vnitřní fragmentace**
- po ukončení procesu vznikají v paměti **mezery**
 - sem lze umístit jen proces, který se tam ještě vejde
 - při umístění procesu do mezery obvykle zůstane mnohem menší mezera – ještě obtížněji využitelná
 - **vnější fragmentace** (external fragmentation)
 - odstranit lze defragmentací – relokačí paměti procesů tak, aby vznikla souvislá volná oblast – **setřesení**

Dynamické dělení (obrázek)



alokace paměti pro
procesy 1, 2 a 3

ukončení procesu 2,
alokace pro proces 4

ukončení procesu 1,
alokace pro proces 5

Umisťovací algoritmus best-fit

- nejlépe padnoucí oblast – **best-fit**
 - vybere stejnou nebo nejmenší volnou oblast, do které se proces vejde
 - nejméně výkonná metoda
 - nejmenší možná fragmentace
 - vždy je použita nejmenší vyhovující oblast
 - fragmenty jsou malé, ale rychle přibývají
 - nutnost častého provádění setřesení (compaction) obsazené paměti

Umist'ovací algoritmus first-fit

- první padnoucí oblast – **first-fit**
 - paměť se prohledává vždy od začátku
 - vybere první volnou oblast, do které se proces vejde
 - rychlejší než best-fit
 - prohledávání zpomaluje výskyt velkého počtu obsazených oblastí na začátku paměti
 - tato část paměti se vždy zbytečně prohledává

Umist'ovací algoritmus next-fit

- následující padnoucí oblast – **next-fit**
 - paměť se prohledává vždy od oblasti, do které se naposledy umisťovalo
 - vybere první volnou oblast, do které se proces vejde
 - je-li po umístění procesu do volné oblasti zbytek oblasti ještě dostatečně velký, umístí se další proces sem
 - nejčastěji se umisťuje na konci paměti
 - obvykle tam je nejvíce volného místa
 - tendence dělit velké oblasti paměti na menší
 - nejrychlejší metoda

Umisťovací algoritmus worst-fit

- největší padnoucí oblast – exact-or-worst-fit
 - vybere stejně velkou volnou oblast jako proces, pokud existuje, jinak největší volnou oblast
 - paměť se prohledává do nalezení stejně velké oblasti
 - při nenalezení stejně velké oblasti prohledáváme celou paměť
 - tendence dělit velké oblasti paměti na menší
 - může mít za následek nemožnost přidělení paměti velkému procesu
 - nejhorší využití paměti – lowest memory utilization

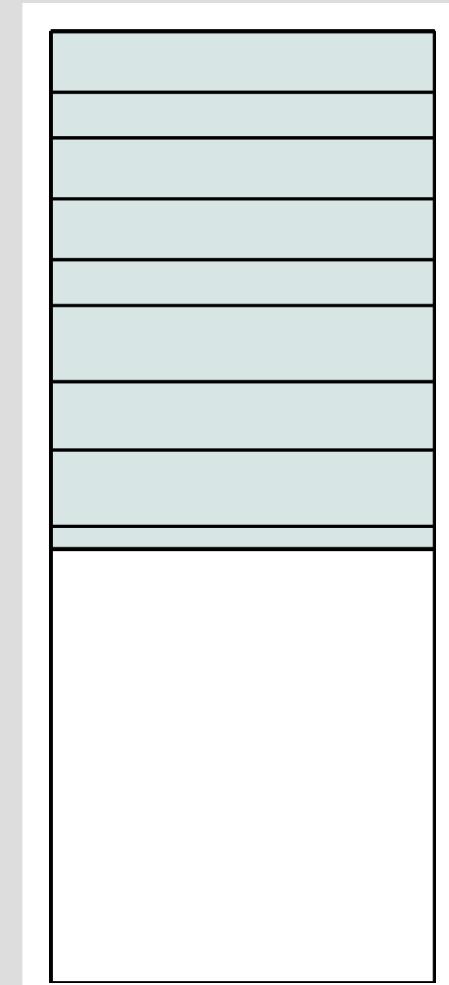
Umíst'ovací algoritmy (obrázek)



před umístěním



po umístění



po setřesení

Možné optimalizace algoritmů

- algoritmy můžeme zrychlit
 - evidence volných oblastí v setříděném seznamu
 - prohledává se pak seznam místo celé paměti
 - rychlosť vyhledání volné oblasti je pak stejná
 - pro metody first-fit a best-fit
 - metoda next-fit je bezúčelná
 - při uvolnění paměti přiléhající k volné oblasti je třeba sousední volné oblasti sloučit v jednu
- rychlé nalezení – **quick-fit**
 - vedeme další seznam běžně alokovaných velikostí

Typy adres

- **fyzická adresa**
 - používá ji Memory Management Unit (**MMU**)
 - absolutní adresa
- **logická adresa**
 - používá ji **CPU**
 - virtuální adresa – je nutno ji převést na fyzickou
 - absolutní adresa
 - vzhledem k logickému adresovému prostoru (procesu)
 - relativní adresa
 - vzhledem k počátku/konci oblasti (např. vrchol zásobníku)

Virtuální paměť

- skutečná paměť
 - fyzická operační paměť (FOP) v počítači (RAM)
- virtuální paměť
 - CPU je obvykle schopné adresovat větší množství paměti, než je skutečně instalováno
 - logický adresový prostor zahrnuje
 - skutečnou fyzickou operační paměť – RAM
 - část sekundární paměti (disku) – swap

Dělení adresového prostoru

- rozdělíme-li adresový prostor procesu na části, stačí, aby pouze některé z nich byly v RAM
 - zbytek adresového prostoru může být na disku
 - proces může běžet, i když nemá v RAM celý svůj adresový prostor
 - adresový prostor procesu může být větší než RAM
 - lze provádět více procesů současně
 - je-li více rozpracovaných procesů, je větší pravděpodobnost, že bude některý ve stavu připraven, a tudíž je procesor lépe využit

Dělení adresového prostoru

- **resident set**
 - část adresového prostoru procesu, která je v RAM
- **swap**
 - sekundární paměť (disk), kam lze odložit (dočasně) nepožívané části adresového prostoru procesu
 - přistupuje-li proces k místu (data, kód), které není v RAM, generuje se přerušení (a skok do OS)
 - OS pak proces blokuje, dokud nenahraje příslušnou část jeho adresového prostoru do RAM
 - po dokončení čtení je generováno přerušení a proces je převeden do stavu připravený

Princip lokality odkazů

- proces má tendenci přistupovat k okolí svého adresového prostoru, kam přistupoval nedávno
 - týká se dat (proměnné ve stejné datové části)
 - i kódu (zejména při provádění cyklů)
- lze obvykle odvodit, kterou část paměti bude proces v nejbližší budoucnosti používat
 - **virtuální paměť může** tedy **fungovat efektivně**
 - z disku se může dopředu načíst do RAM více částí paměti procesu z okolí právě vyžadované oblasti

Thrashing

- vzniká při špatné organizaci odkládání částí paměti procesu na disk
 - část paměti procesu je odložena na disk těsně před tím, než ji proces potřebuje
 - tuto část paměti je pak třeba následně nahrát zpět do RAM
 - proces pak zbytečně čeká v blokovaném stavu
- může být následkem nedostatečného dodržení principu lokality odkazů
 - neoptimalizovaného překladu (kompilátorem)
 - neefektivního programování (programátorem)

HW podpora pro virtuální paměť

- procesor musí umět pracovat s logickou (virtuální) adresou
 - CPU předává virtuální adresu paměťové jednotce
- MMU (Memory Management Unit) převádí virtuální adresu na adresu skutečnou (v RAM)
- nenachází-li se adresovaná část v RAM, je třeba generovat přerušení a předat řízení OS
 - OS vydá příkaz pro nahrání příslušné části paměti z disku do RAM

Stránkování paměti

- fyzická operační paměť **RAM** je rozdělena na oblasti malé velikosti – **rámce** (frames)
- logický adresový prostor **procesu** se rozdělí na stejně velké části jako RAM – **stránky** (pages)
- OS udržuje pro každý proces **tabulku** přiřazení stránek k rámcům
- logická adresa se skládá z **čísla stránky** a **offsetu**
 - offset je relativní adresa vzhledem k začátku stránky
 - číslo stránky = logická adresa / velikost stránky

Přidělení stránek (obrázek)

rámeč
0
1
2
3
4
5
6
7
8
9
10
11
12
13
14

alokace rámců

rámeč
0
1
2
3
4
5
6
7
8
9
10
11
12
13
14

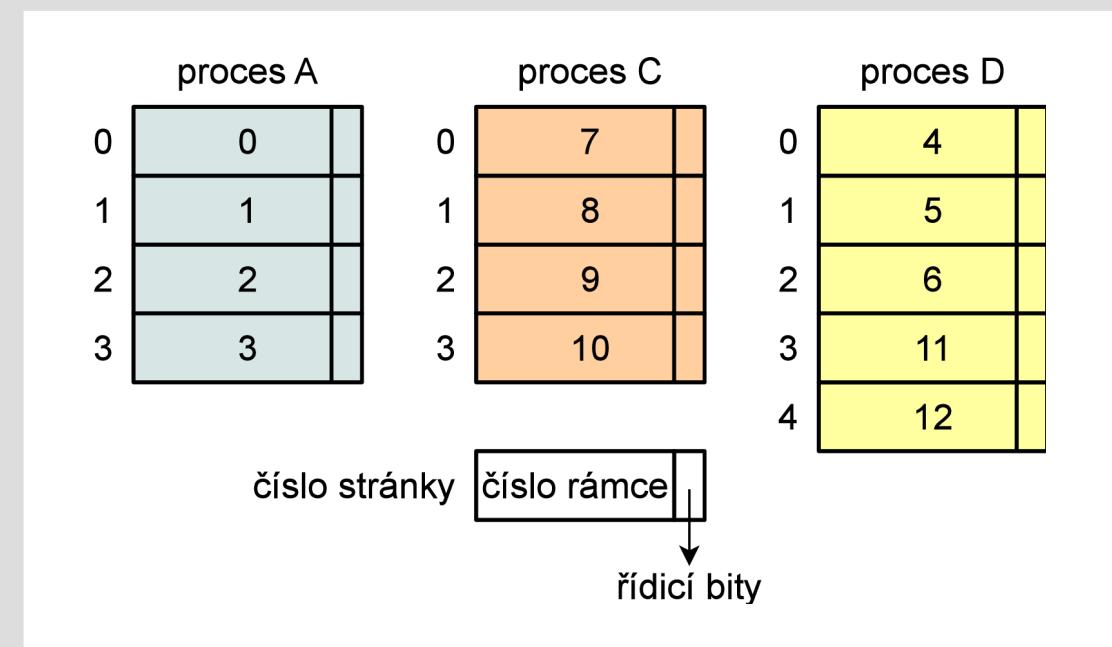
ukončení procesu B

rámeč
0
1
2
3
4
5
6
7
8
9
10
11
12
13
14

alokace pro proces D

Stránkové tabulky (obrázek)

rámeč	
0	proces A – stránka 0
1	proces A – stránka 1
2	proces A – stránka 2
3	proces A – stránka 3
4	proces D – stránka 0
5	proces D – stránka 1
6	proces D – stránka 2
7	proces C – stránka 0
8	proces C – stránka 1
9	proces C – stránka 2
10	proces C – stránka 3
11	proces D – stránka 3
12	proces D – stránka 4
13	
14	

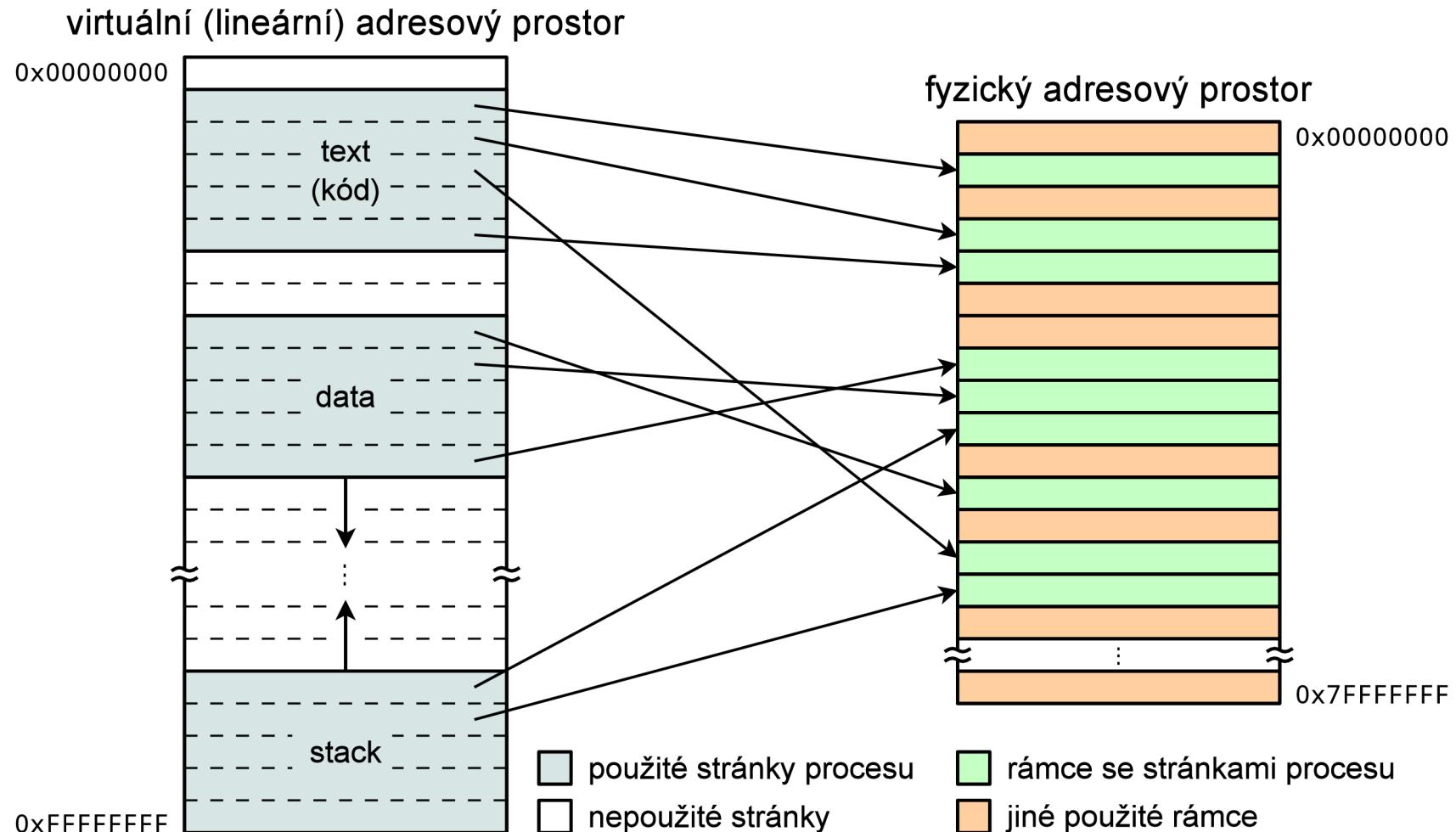


stránkové tabulky procesů
indexem je číslo stránky

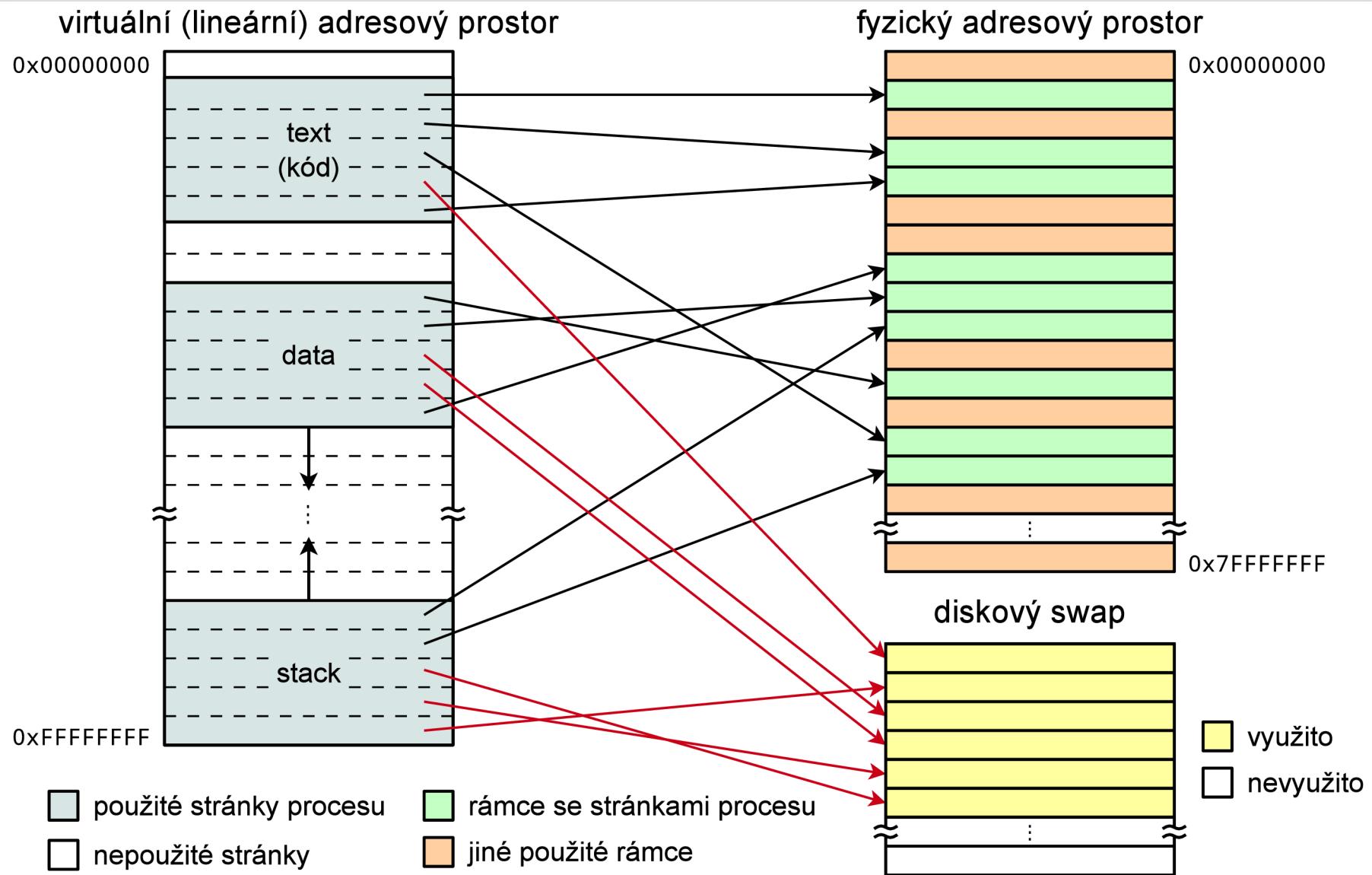
Stránkování paměti

- logický adresový prostor procesu je **lineární**
- skutečné umístění stránek v RAM je **nespojité**
- velikost stránek je daná HW, typicky KiB až MiB
 - IA-32: **4 KiB** a 4 MiB (bez PAE) nebo 2 MiB (s PAE)
- stránková tabulka obsahuje příznaky (**řídicí byty**)
 - **přítomnost** stránky v RAM (present)
 - stránka je **používaná** (referenced, accessed)
 - **modifikovaný** obsah (modified, dirty)
 - nebyla-li stránka v RAM modifikovaná a je již na disku, není třeba ji znova zapisovat při uvolňování rámce

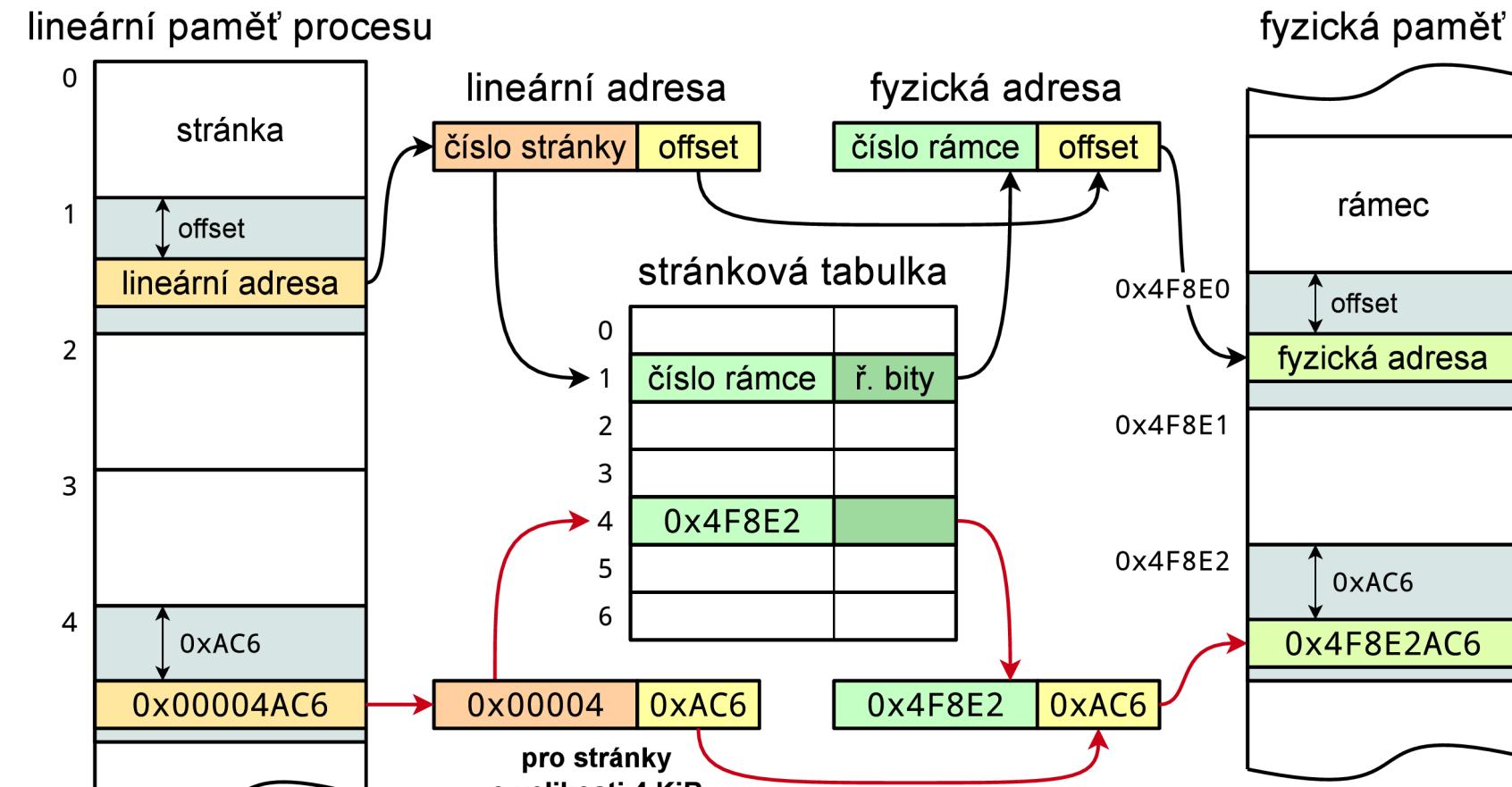
Zobrazení stránek procesu do rámci FOP (obrázek)



Zobrazení stránek procesu do virtuální paměti (obrázek)



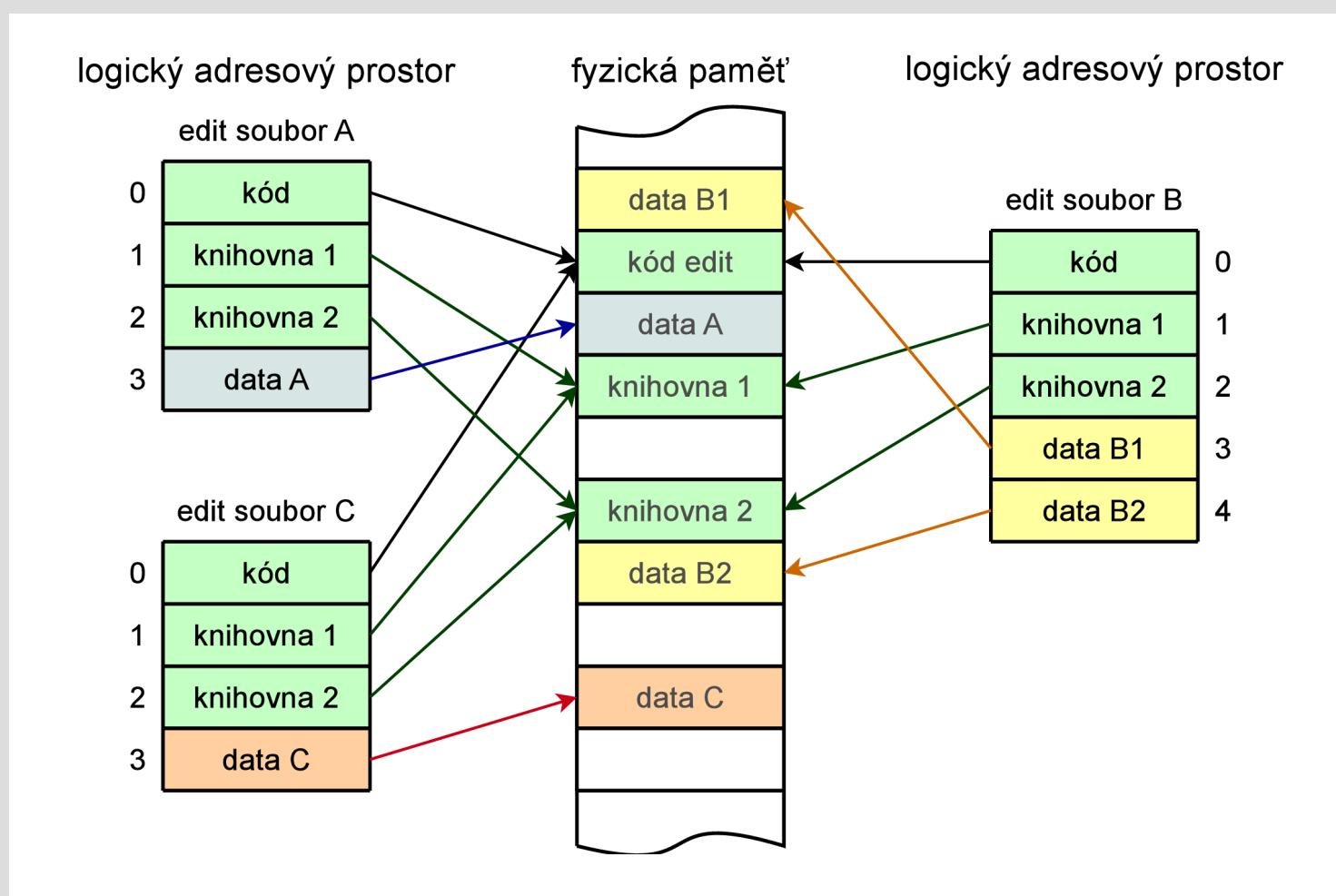
Stránkování – převod virtuální adresy na fyzickou (obrázek)



Sdílení stránek

- stejné stránky různých procesů mohou sdílet rámcem
 - je zbytečné mít v paměti stejně kopie dat
 - je snadné sdílet stránky v režimu read-only (kód)
 - je třeba brát ohled na odkládání obsahu sdílených rámců na swap
- lze sdílet i stránky obsahující stránkové tabulky
 - např. dva procesy vykonávající stejný program mohou sdílet část stránkové tabulky příslušející přiřazení stránek pro kód programu (a knihovny)

Stránkování – sdílení (obrázek)



Sdílení stránek v OS UNIX

- po provedení systémového volání fork(2)
 - rodič i potomek má svou vlastní stránkovou tabulkou
 - stránkové tabulky jsou identické kopie
 - procesy sdílejí celou virtuální paměť
 - všechny stránky jsou označeny read-only
 - při zápisu do stránky je vyvoláno přerušení (porušení ochrany) a předá se řízení OS (TRAP)
 - OS vytvoří kopii stránky (každý proces má nyní vlastní) a nastaví obě kopie do režimu read-write – **copy-on-write**
 - důsledek: většina stránek je sdílena

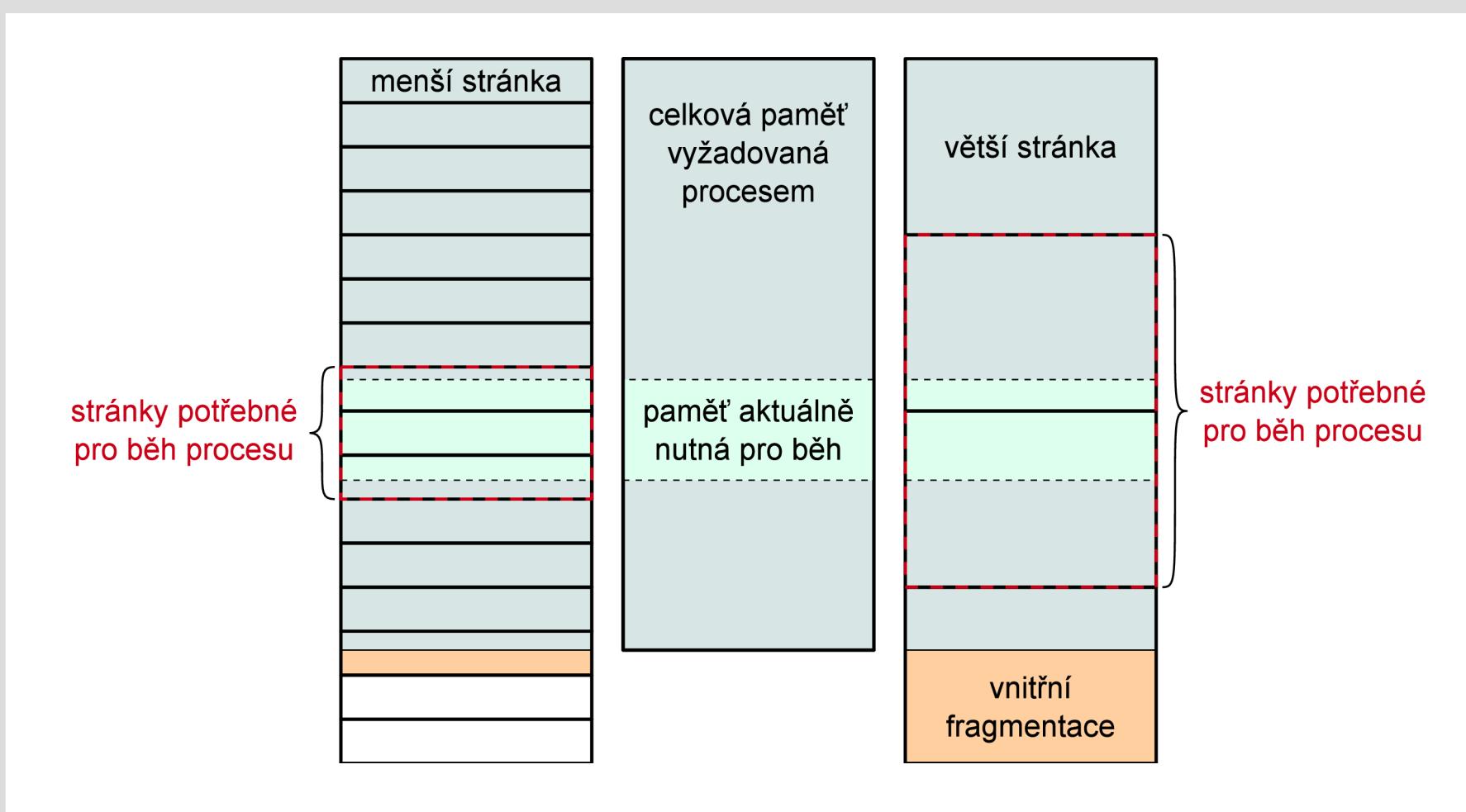
Velikost stránek – menší

- paměť se alokuje po stránkách pevné velikosti
 - vzniká **vnitřní fragmentace**
- menší velikost stránky
 - menší vnitřní fragmentace
 - více stránek na proces
 - větší stránková tabulka – zabírá další paměť
 - více rámců v RAM = lepší hospodaření s RAM
 - v RAM budou stránky z okolí posledních odkazů na paměť a počet neúspěšných přístupů se bude snižovat

Velikost stránek – větší

- větší velikost stránky
 - větší vnitřní fragmentace
 - menší stránková tabulka
 - méně rámců v RAM = horší hospodaření s RAM
 - v RAM budou díky velikosti stránek i bloky paměti, které nejsou aktuálně využívány – princip lokality
 - při nedostatku RAM se bude počet neúspěšných přístupů zvyšovat – hrozí thrashing
 - sekundární paměť je efektivnější při přenosu dat po větších blocích

Velikost stránek (obrázek)



Rozsah stránkové tabulky

- rozsah stránkové tabulky může být velký
 - virtuální adresa 32 bitů, velikost stránky 4 KiB:
 - offset 12 bitů ($2^{12} = 4 \text{ Ki}$), adresový rozsah 4 GiB (2^{32}), pro počet stránek 20 bitů $\rightarrow 2^{32-12} = 2^{20} = 1 \text{ Mi}$ stránek
 - má-li položka stránkové tabulky 32 bitů, pak její celková velikost je **4 MiB** ($1 \text{ Mi} \cdot 4 \text{ B}$) **pro každý proces**
 - proces většinu stránek nepoužije
 - typicky používá stránky z počátku adresového rozsahu (kód a data) a stránky z konce rozsahu (stack)
 - udržovat celou tabulku v paměti je zbytečné
 - 64bitová adresa: velikost tabulky 32 PiB ($2^{64-12} \cdot 8 \text{ B}$)
 - nerealizovatelné!

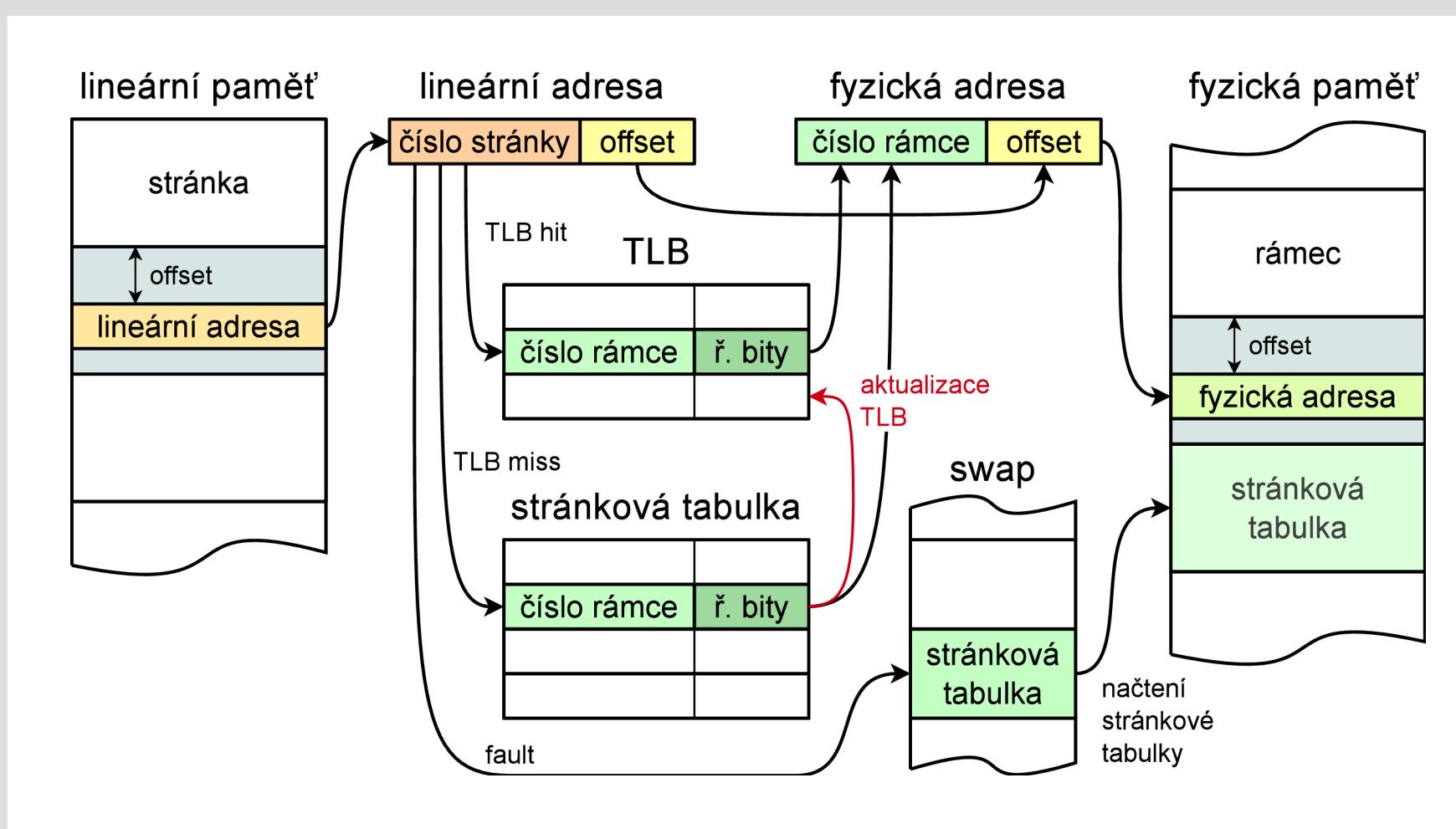
Umístění stránkových tabulek

- stránkové tabulky musí být také v paměti
 - velikost tabulky může být velká, proto se tabulka umisťuje také **do virtuální paměti**
 - v RAM nemusí být tabulka celá
 - část může být na disku (swap)
 - při přístupu procesu na paměťové místo ve stránce, jejíž položka ve stránkové tabulce není v RAM, znamená čtení z disku – zpomalení překladu
 - po načtení položky se může zjistit, že ani stránka není v RAM (**page fault**) – další čtení z disku a zpomalení běhu procesu

Translation Lookaside Buffer (TLB)

- speciální cache pro položky stránkové tabulky
- obsahuje několik posledních použitých položek stránkové (příp. invertované) tabulky
- při hledání položky stránkové tabulky se prohledá nejprve TLB
 - kontrola přítomnosti v TLB může být paralelizována
 - nalezení (hit) – převod virtuální adresy na fyzickou
 - nenalezení (miss) – doplnění položky do TLB
 - doplnění může být z části tabulky v RAM nebo z disku

Použití TLB (obrázek)



Invertovaná stránková tabulka

- místo stránek se v tabulce evidují rámce
 - velikost je závislá na instalovaném množství RAM
 - výrazná úspora: **jediná tabulka pro celý systém**
 - položky obsahují kromě čísla stránky také ID procesu
 - 64bitové položky, 4KiB stránky, 1 GiB paměti RAM:
 - velikost: $1 \text{ GiB} / 4 \text{ KiB} \cdot 8 \text{ B} = 2 \text{ MiB}$ (2^{18} 64b. záznamů)
- převod virtuální adresy má ale vyšší režii
 - tabulka je indexovaná čísly rámců, nikoliv stránek, tj. je třeba tabulku prohledat celou – složitost $> O(1)$
 - částečné zrychlení – použití TLB
 - jiná metoda zrychlení – použití hashovací tabulky

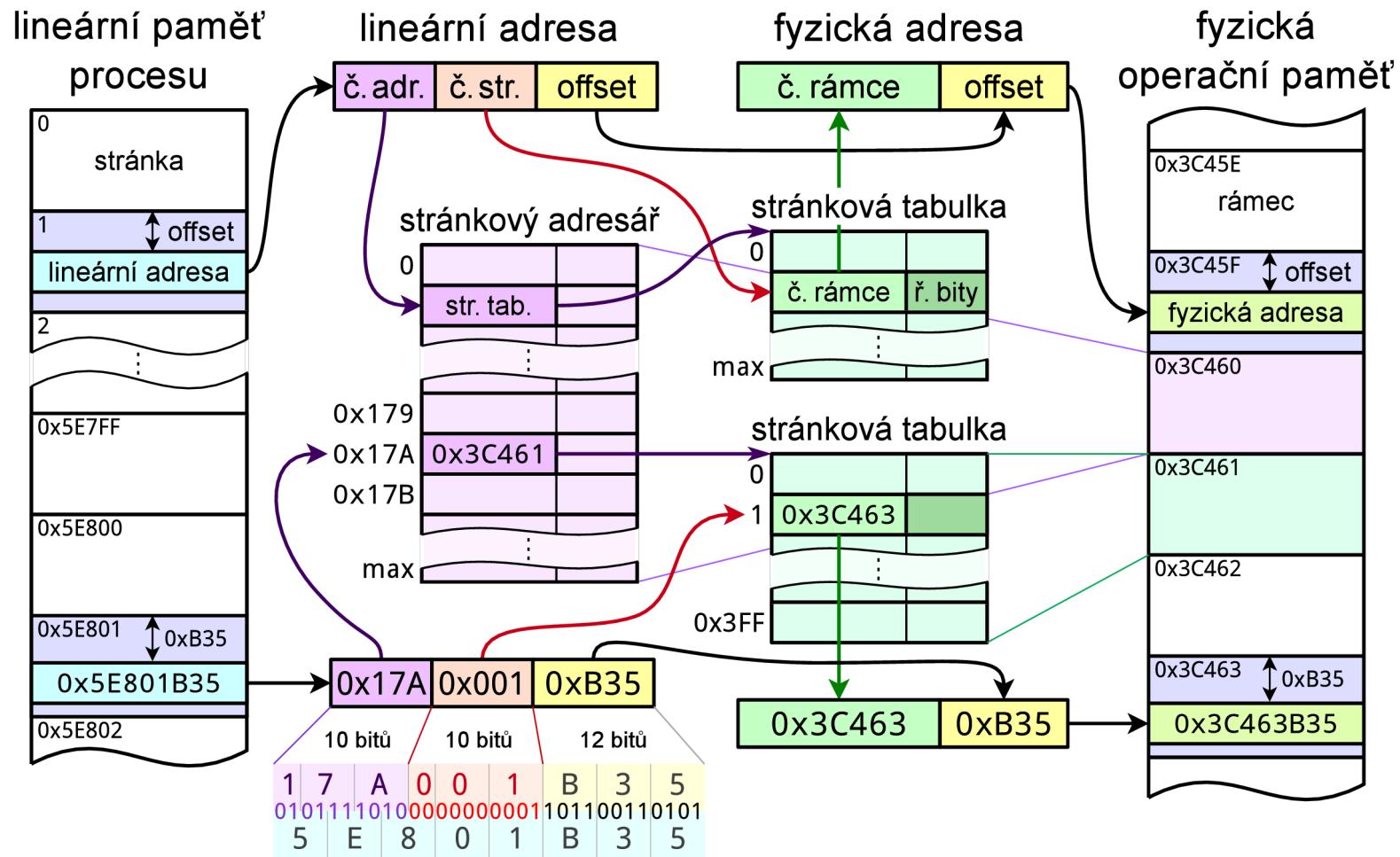
Víceúrovňové stránkové tabulky (1)

- velký rozsah stránkové tabulky lze řešit použitím hierarchických stránkových tabulek
 - virtuální adresa se rozdělí na offset a dva indexy do stránkových tabulek dvou úrovní
 - první index určí položku **stránkového adresáře**, ze které se odvodí **adresa stránkové tabulky** druhé úrovně
 - druhý index určí položku **stránkové tabulky**, která obsahuje **číslo rámce**
 - př.: 4KiB stránky: 12 bitů offset a 2 indexy 2×10 bitů
 - vytvoří se tak 1024 položkové (2^{10}) tabulky stránek
 - stránková tabulka zabírá jeden rámec ($2^{10} \cdot 4 \text{ B} = 4 \text{ KiB}$)

Víceúrovňové stránkové tabulky (2)

- vytvoří se stromová struktura tabulek
- hlavní tabulka (adresář) zůstává v RAM
- tabulky druhé úrovně jsou ve virtuální paměti
 - mohou být odloženy na disk
 - nemusejí existovat – mohou být vytvářeny dynamicky za běhu procesu
 - když proces alokuje další paměť a je potřeba ji umístit do nových stránek
- používáno architekturami IA-32 (2–3 úrovně) a IA-32e (x86_64, 4 a více úrovní)

Stránkování – dvouúrovňové stránkové tabulky (obrázek)



Stránkování na platformě IA-32

- virtuální adresa: 32 bitů
 - proces může mít max. 4 GiB paměti (2^{32})
- velikost stránky: 4 KiB
 - případně 4 MiB (bez PAE) nebo 2 MiB (s PAE)
- fyzická adresová sběrnice: 36 bitů
 - fyzické operační paměti může být max. 64 GiB
 - pro použití více než 4 GiB je nutná podpora PAE
 - bez PAE: dvouúrovňová struktura stránkových tabulek

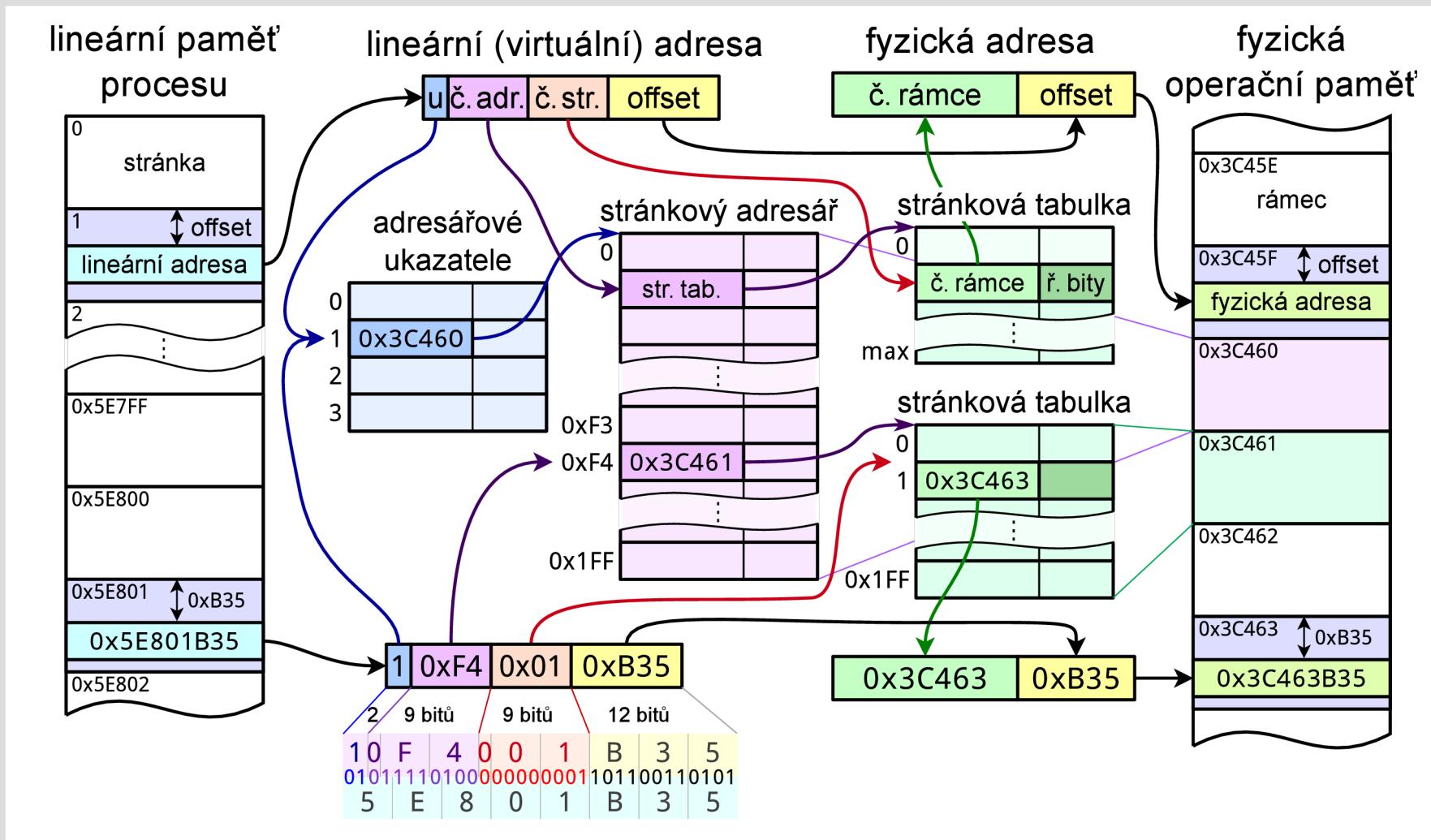
IA-32: řídicí bity v položce stránkové tabulky / adresáře

- 32bitový režim – dolních 12 bitů je řídicích
 - stránková tabulka (PT), stránkový adresář (PD)
 - 0 – **P** – **present** – použitá položka
 - 1 – R/W – zápis povolen
 - 2 – U/S – user/supervisor – přístup v uživatelském režimu
 - 3 – PWT – page-level write-through
 - 4 – PCD – page-level cache disable
 - 5 – **A** – **accessed (referenced)** – stránka byla použita
 - 6 – PT: **D** – **dirty (modified)** – změněna, PD: ignored
 - 7 – PT: PAT/reserved, PD: PS – page size (4KiB/4MiB)
 - 8 – PT: G/ignored – global (podle CR4.PGE), PD: ignored
 - 9–11 – ignored

Režim PAE na platformě IA-32

- PAE (Physical Address Extension)
 - tří- / dvouúrovňové stránkování (4KiB / 2MiB stránky)
 - lineární adresa má 32 bitů, fyzická 36 (až 52) bitů
 - 2 bity: ukazatele na stránkové adresáře (registry PDPTEi)
 - 9 bitů: index do stránkového adresáře
 - 9 bitů: index do stránkové tabulky (pouze pro 4KiB stránky)
 - 12 bitů / 21 bitů: offset ve 4KiB / 2MiB stránce
 - položky stránkové tabulky / adresáře mají 64 bitů
 - číslo rámce: 24–40 bitů (4KiB stránky), 15–31 bitů (2MiB)
 - řídicí bity: dolních 12 bitů, bit 63: XD (execute-disable), pro 2MiB stránky: PAT je bit 12 (bit 7 je PS)

Stránkování – tříúrovňové stránkové tabulky (obrázek)



Adresace IA-32e (64-bit)

- IA-32e – dva podrežimy
 - kompatibilní: virtuální adresace je 32bitová
 - 64bitový: 48 bitů virtuálních → 40–52 bitů fyzických
 - 2^{48} B = 256 TiB pro proces, až 2^{52} B = 4 PiB fyzické RAM
 - 64bitový režim: stránky 4 KiB, 2 MiB nebo 1 GiB
 - čtyř-, tří- a dvouúrovňové stránkové tabulky
 - 9 bitů: PML4 (page map level four)
 - 9 bitů: index do tabulky ukazatelů na stránkové adresáře
 - 9 bitů: index do stránkového adresáře, pouze 4KiB a 2MiB
 - 9 bitů: index do stránkové tabulky, pouze pro 4KiB stránky
 - 12 / 21 / 30 bitů: offset ve 4KiB / 2MiB / 1GiB stránce

Stránkové algoritmy

- algoritmy pro manipulaci se stránkami procesů
- strategie zavádění – **fetch policy**
 - které stránky a **kdy** se zavedou **do paměti**
- strategie umisťování – **placement policy**
 - **kam** se umístí do fyzické paměti (do kterého rámce)
- strategie nahrazování – **replacement policy**
 - které stránky **se odloží** (na swap)
- strategie uklízení (čištění) – **cleaning policy**
 - **kdy se uloží** modifikované stránky na swap

Zavádění stránek do RAM

- strategie zavádění – **fetch policy**
 - určuje, **která** stránka se má **kdy** zavést do RAM
- **demand paging**
 - zavádění stránek jen tehdy, jsou-li potřeba
 - vyskytne-li se odkaz na paměťové místo v dané stránce
 - způsobuje neúspěšné přístupy při zavádění procesu
- **lookahead paging**
 - zavádění stránek předem
 - dle principu lokality odkazů se zavedou stránky z okolí té právě vyžadované (využívá se sekvenční čtení z disku)

Umisťování stránek do RAM

- strategie umisťování – **placement policy**
 - určuje, **do které části RAM** (rámce) se stránka zavede
 - typicky se tím OS nezabývá
 - vybírá obvykle libovolný volný rámec
 - přístupová doba (access time) je stejná pro všechny adresy v RAM
 - skutečnou adresu používá pouze HW

Nahrazování stránek v RAM

- strategie nahrazování – **replacement policy** (RP)
 - určuje, které stránky se z RAM odloží na disk, je-li potřeba načíst jiné stránky
- ideální strategie – odstranění stránky, která bude nejdelší dobu nevyužitá
 - nelze předem přesně určit – pouze se odhaduje
- pro stránky, které nesmí být odstraněny z RAM, se používá **zamykání rámců** (frame locking)
 - jádro OS, V/V buffery, klíčové řídicí struktury OS

RP – Least Recently Used (LRU)

- nahradí stránku, na kterou nebyl nejdelší dobu žádný odkaz
 - dle principu lokality odkazů je to stránka s nejmenší pravděpodobností výskytu odkazu v nejbližší budoucnosti
- režijně náročná strategie
 - u každé stránky je třeba udržovat údaj o čase posledního odkazu na ni (nebo stačí čítač)
 - při nahrazování je třeba najít patřičnou stránku
 - lze řešit pomocí seřazeného seznamu stránek

RP – Least Recently Used (LRU) – možné implementace

- seřazený seznam má vysokou režii
 - při každém přístupu do paměti je třeba aktualizovat
- rozšíření řídicích bitů položky stránkové tabulky
 - při přístupu ke stránce uloží MMU do stránkové položky navýšenou hodnotu globálního čítače
 - při výskytu page-fault se vyhledá nejnižší hodnota
- evidence matice $n \times n$ bitů, kde $n =$ počet rámců
 - při přístupu ke stránce v rámci číslo k nastaví MMU v k -tém řádku jedničky a v k -tém sloupci nuly
 - page-fault → řádek matice s nejnižším číslem (↗)

RP – Not Recently Used (NRU)

- každá stránka má v řídicích bitech položky
 - odkazovaná stránka (referenced bit – R)
 - nastaví se při každém čtení stránky nebo jejím zápisu
 - periodicky se tento bit nuluje
 - změněná stránka (modified bit – M)
 - nastaví se při změně obsahu stránky
- nahradí libovolnou stránku z nejnižší třídy
 1. $R = 0, M = 0$ (neodkazovaná, nezměněná)
 2. $R = 0, M = 1$ (neodkazovaná, změněná)
 3. $R = 1, M = 0$ (odkazovaná, nezměněná)
 4. $R = 1, M = 1$ (odkazovaná, změněná)

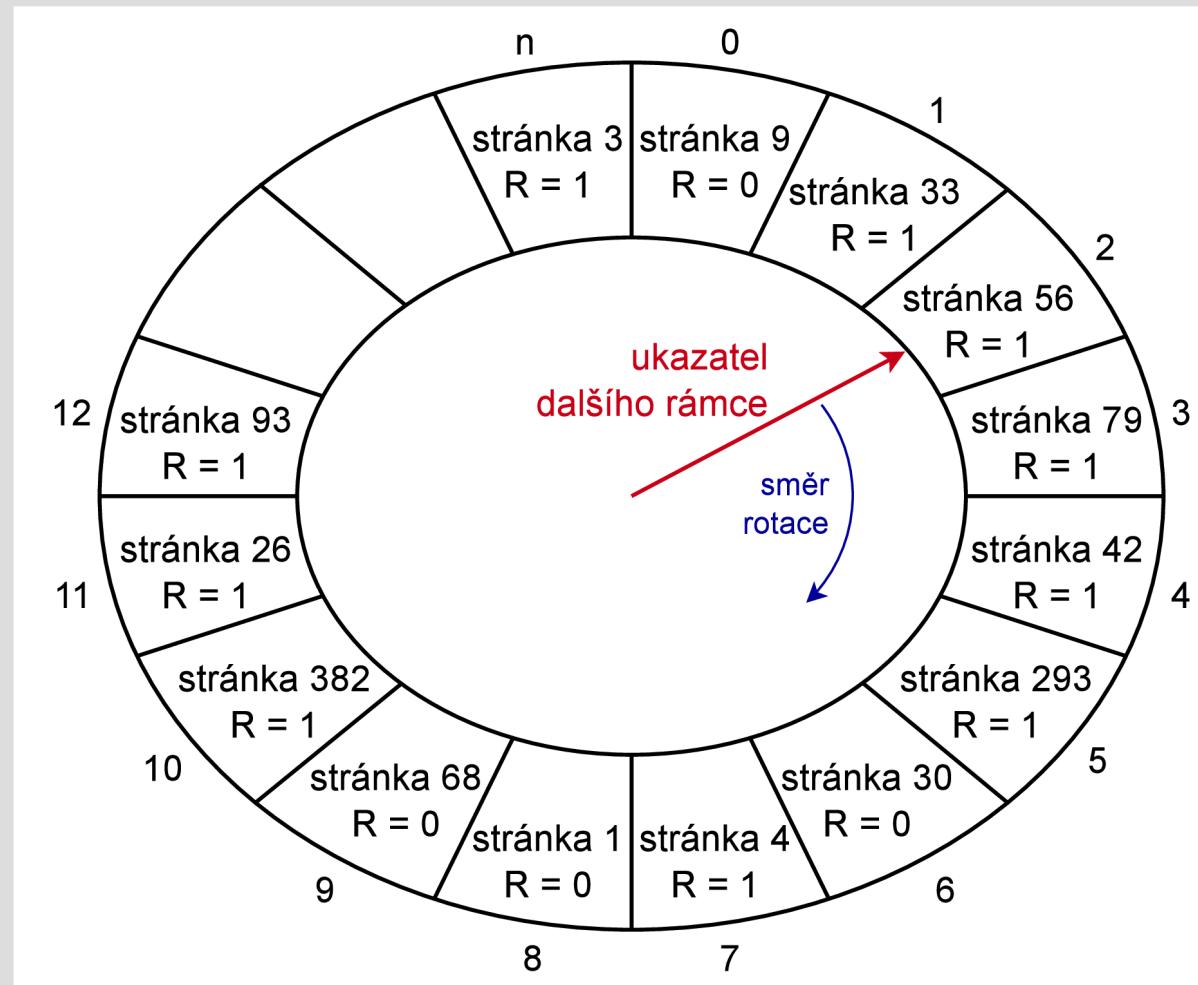
RP – First-In, First-Out (FIFO)

- nahradí stránku, která byla nejdéle ve FOP
 - OS udržuje seznam stránek ve frontě
 - nové položky jsou připojovány na konec fronty
 - první ve frontě je tedy stránka nejdéle používaná
- nenáročná strategie, ale nevýhodná
 - může odstranit stránku, která je právě používaná
- modifikace strategie – **second chance**
 - je-li bit referenced – $R = 1$, je vynulován a stránka je přesunuta na konec fronty

RP – Clock Policy

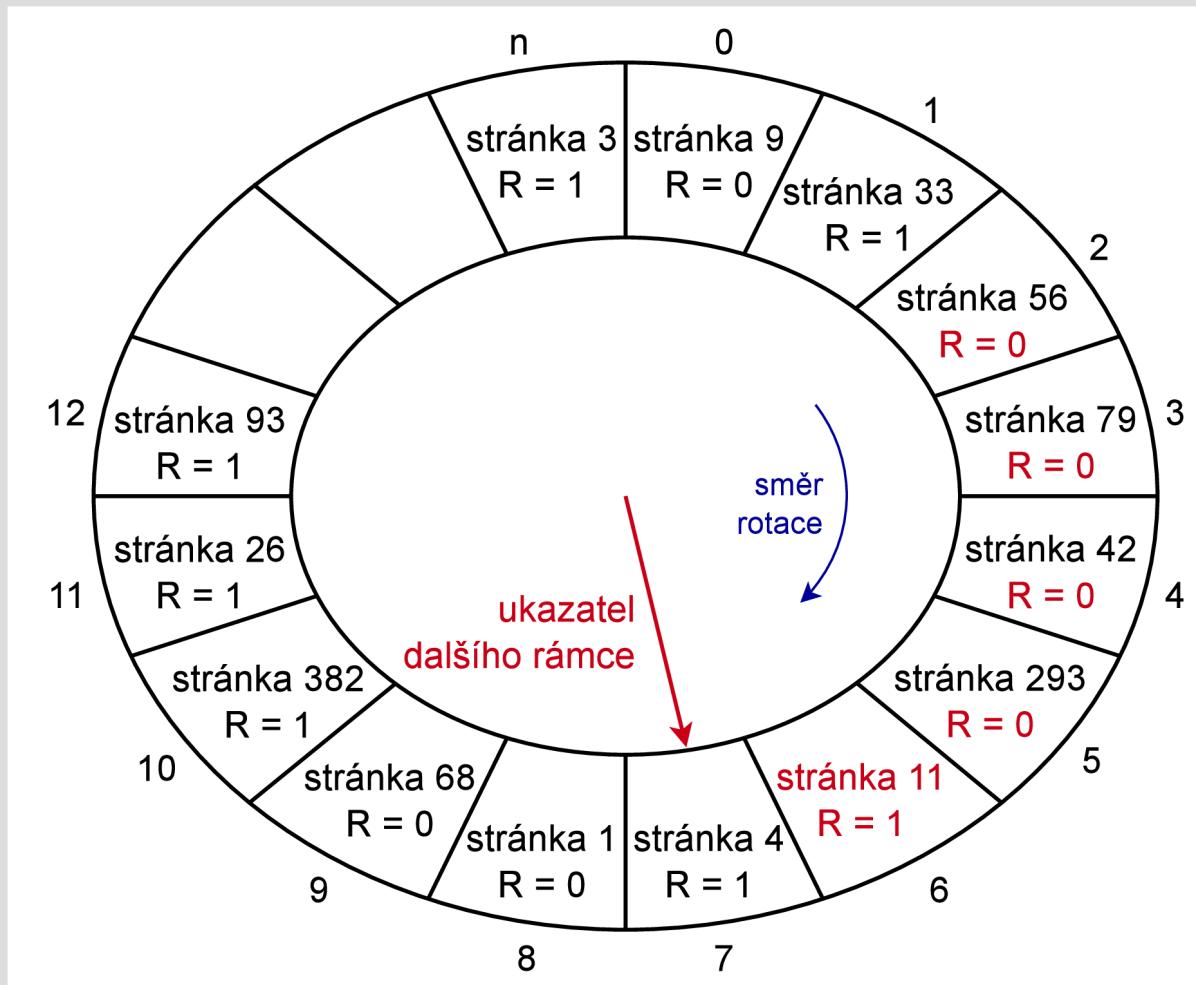
- rámce jsou v kruhovém seznamu
 - index určuje poslední položku
 - není-li volný rámcem a nastane page fault, zvyšuje se index tak dlouho, dokud se nenajde stránka s bitem použití (referenced) $R = 0$
 - tato stránka se nahradí stránkou novou
 - při procházení seznamu se bit R nuluje

Clock Policy (obrázek 1)



stav před umístěním stránky číslo 11

Clock Policy (obrázek 2)



umístění stránky číslo 11

RP – Working Set Clock Policy

- vylepšení nedostatku clock policy
 - při nalezení položky určené pro nahrazení se kontroluje také řídicí bit modifikace (M)
 - má-li nalezená stránka (určená pro nahrazení, $R = 0$) nastaven bit M, musí se uložit na disk, a čekalo by se na dvě diskové operace (zápis, čtení)
 - místo nahrazení změněné stránky proto algoritmus WSClock jen naplánuje diskovou operaci (uložení stránky na disk) a pokračuje ve vyhledávání
 - důsledek: nahrazuje se stránka s nulovými bity R i M
 - zdržení pouze čtením, nikoliv zápisem

Porovnání algoritmů RP

algoritmus hodnocení

ideální

neimplementovatelný

LRU

výborný, ale náročný na režii

NRU

velmi hrubá approximace LRU

FIFO

odstraňuje používané stránky

2nd chance

výrazné vylepšení FIFO

clock

vylepšená implementace 2nd chance

WSClock

efektivní vylepšení clock

Uklízení (čištění) stránek

- strategie uklízení (čištění) – **cleaning policy**
 - určuje, kdy se (modifikované) stránky uloží na disk
- **demand cleaning**
 - ukládání stránky z rámce vybraného pro nahrazení
 - proces čekající po page fault čeká na přenos 2 stránek
- **precleaning**
 - periodické ukládání stránek v dávce
 - může vést ke zbytečným zápisům
 - obsah stránky se po uložení na disk může opět změnit

Buffering stránek (1)

- rámce vybrané pro nahrazení (provádí průběžně algoritmus RP) se vkládají do dvou seznamů
 - seznam volných stránek – **free page list**
 - obsah těchto rámců nebyl od jejich načtení změněn
 - při nahrazení není třeba jejich obsah zapisovat na disk
 - seznam změněných stránek – **modified page list**
 - při nahrazení se obsah těchto rámců musí uložit
 - rámec vybraný pro nahrazení je uložen na konec příslušného seznamu a bit přítomnosti stránky v RAM je ve stránkové tabulce vynulován

Buffering stránek (2)

- nastane-li page fault, hledá se v seznamech, jestli daná stránka ještě není v RAM
 - je-li v některém seznamu, nastaví se zpět bit přítomnosti a stránka se ze seznamu odstraní
 - není-li v žádném seznamu, je stránka načtena do rámce ze začátku seznamu volných stránek a stránka je z tohoto seznamu odstraněna
 - vyprázdní-li se seznam volných stránek, obsah rámců ze seznamu změněných stránek je zapsán na disk a stránky se přeřadí do seznamu volných

Volba velikosti resident set

- pevná alokace – **fixed-allocation policy**
 - procesu je při nahrávání vyhrazen pevný počet rámců RAM (podle kritérií)
 - rovné či proporcionální rozdělení rámců mezi procesy
 - při page fault, se musí uvolnit **rámec stejného procesu**
- proměnná alokace – **variable-allocation policy**
 - počet rámců procesu se průběžně může měnit
 - zvětšuje se při vysoké frekvenci výskytů page fault
 - snižuje se při nízké frekvenci výskytů page fault
 - vyžaduje režii OS při odhadu chování procesů

Segmentace paměti

- paměť procesu je rozdělena na nespojité oblasti
 - obvykle odpovídají modulům
 - např. kód programu, kód knihoven, stack, data, heap
 - umožňuje nezávislou kompliaci modulů
- oblasti mohou být různě velké
- alokace paměti je podobná dynamickému dělení paměti
 - místo alokace pro proces alokujeme pro segment (modul procesu, knihovnu)

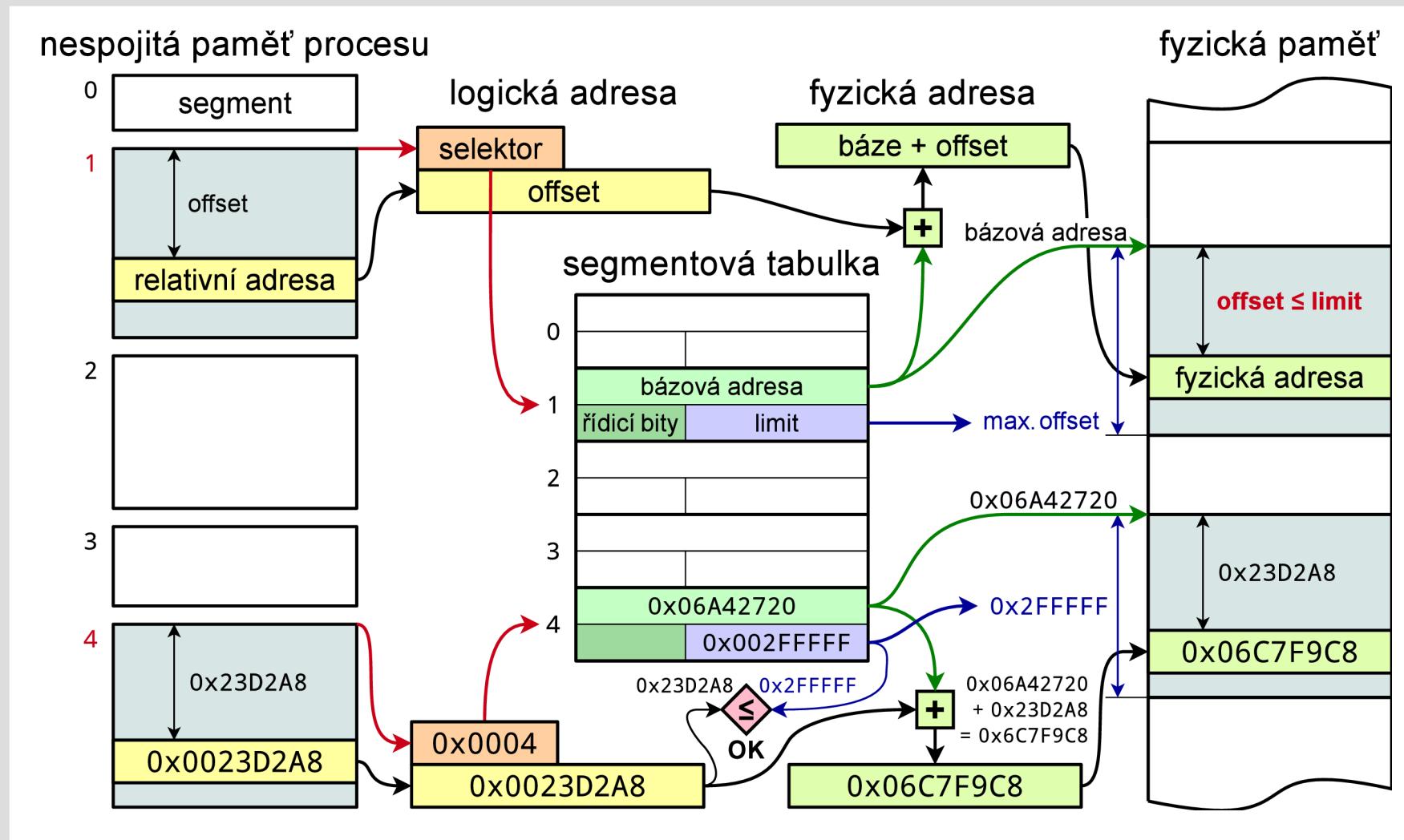
Segmentace paměti – vlastnosti

- zjednodušuje adresaci datových struktur
- zjednodušuje sdílení mezi procesy
 - procesy sdílejí celé segmenty (např. kód)
- logická adresa se skládá z **čísla segmentu** a **offsetu** (relativní adresa vzhledem k začátku)
- OS udržuje pro každý proces tabulku umístění segmentů v paměti společně s jejich délkami
 - implicitně řeší problém ochrany
 - položky tabulky obsahují také řídicí byty

Segmentace – registry CPU (x86)

- bázové (segmentové) registry
 - CS – kódový segment
 - SS – zásobník (stack segment)
 - DS, ES, FS, GS – (extra) datový segment
- offsetové registry – relativní adresa k bázi
 - SP – stack pointer – vrchol zásobníku
 - BP – base pointer – typicky adresuje lokální proměnné na zásobníku
 - SI, DI – source, destination index
 - BX – obecný registr používaný i jako offset

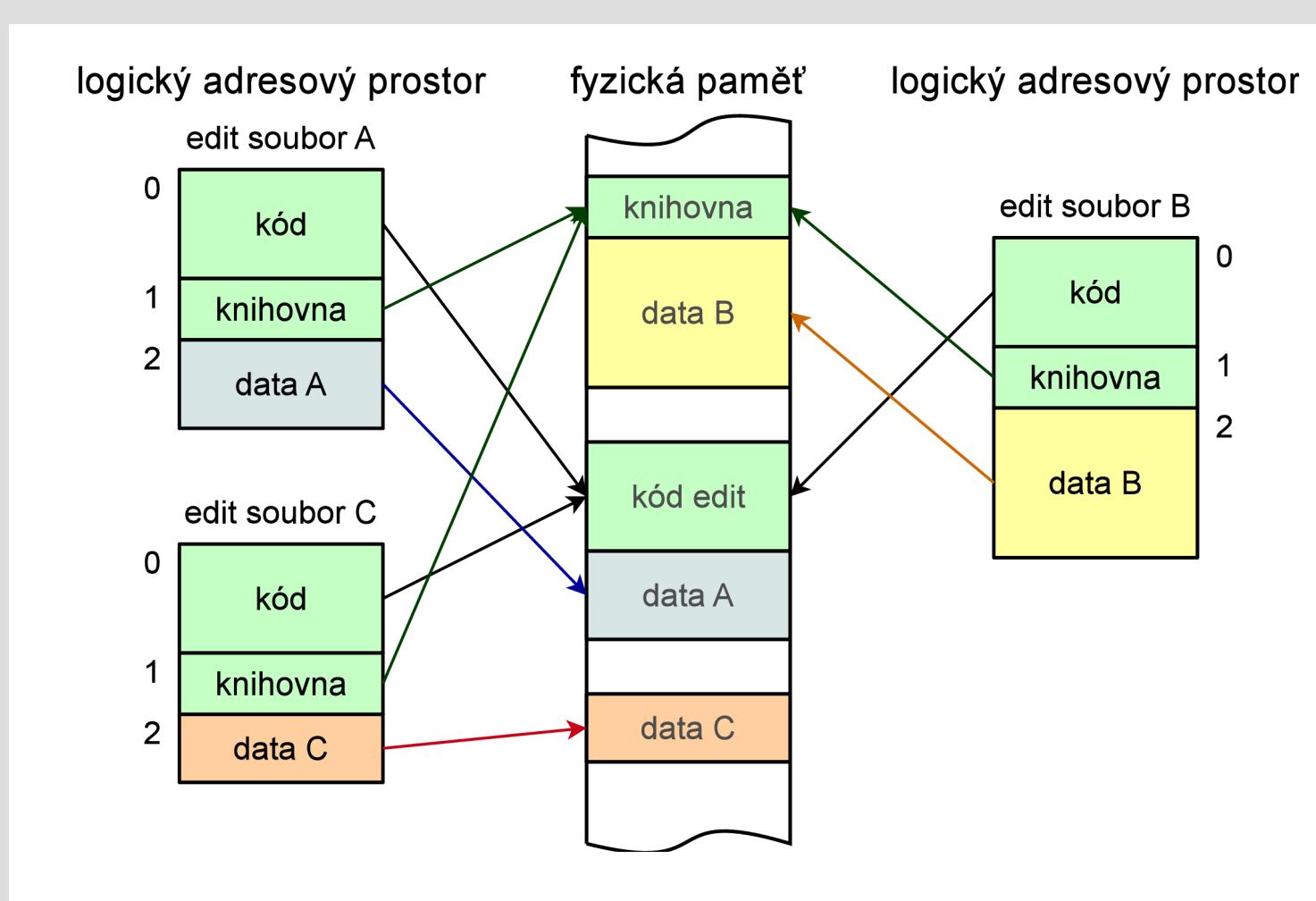
Segmentace – převod virtuální adresy na fyzickou (obrázek)



Segmentace – sdílení segmentů

- moduly procesů mohou být snadno sdíleny
- segmentové tabulky procesů mohou odkazovat na stejné segmenty
- procesy pak sdílejí celý segment
 - výhodné pro sdílení kódu procesů
 - vhodné i pro sdílení dat mezi procesy
 - logičtější než sdílení jednotlivých stránek

Segmentace – sdílení (obrázek)



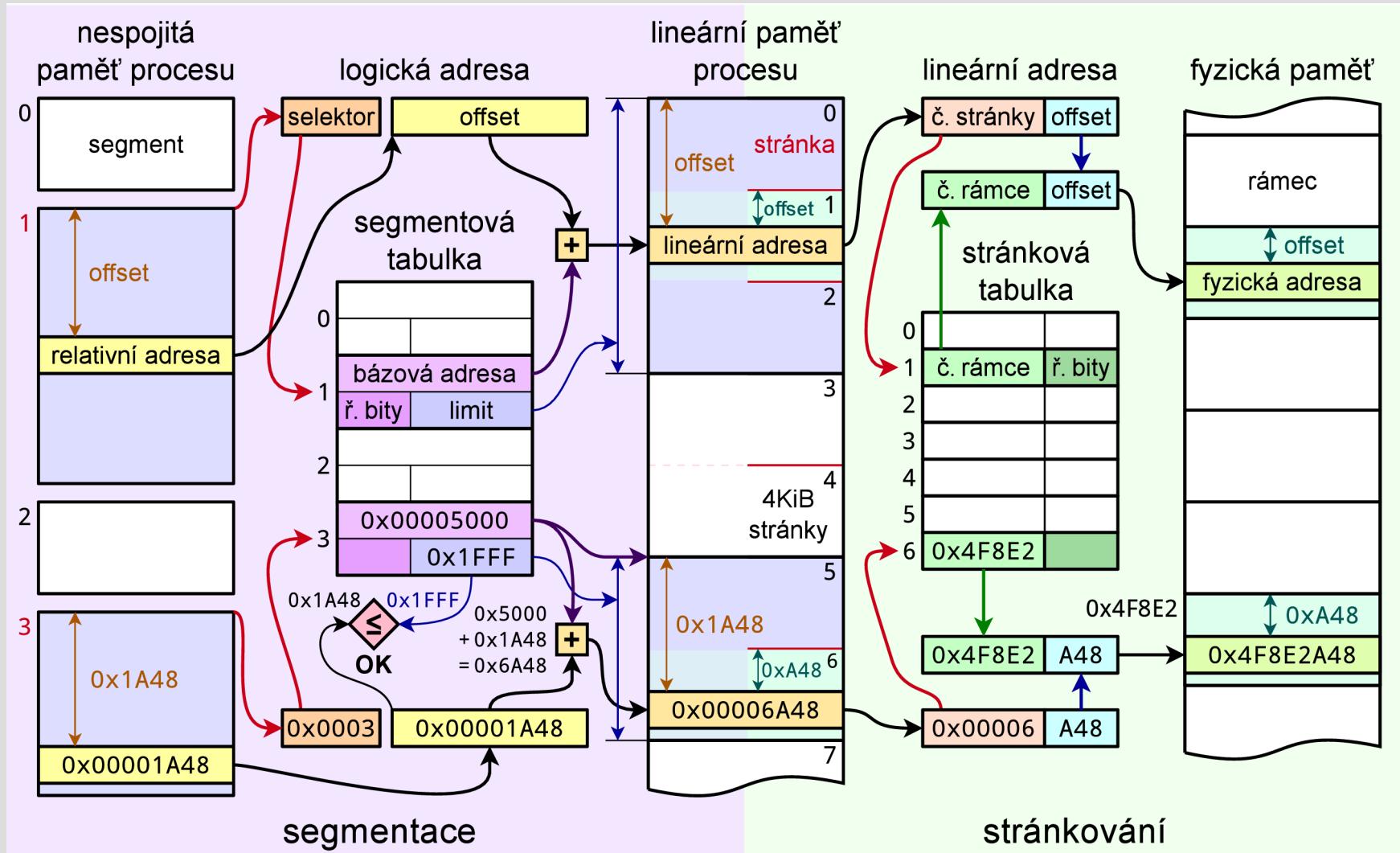
Segmentace – hodnocení

- výhody
 - přirozená alokace paměti – bez vnitřní fragmentace
 - segmenty mohou být zaváděny na vyžádání
 - sdílení segmentů jako logických celků procesu
 - přirozené řešení ochrany paměti
- nevýhody (podobné jako u dynamického dělení)
 - velké segmenty, horší hospodaření s pamětí
 - vnější fragmentace
 - nespojitá paměť – netransparentní pro programátora
 - důsledek: **segmentace je spíše na ústupu**

Kombinace segmentace se stránkováním

- rozdelením segmentů na stránky
 - se odstraní vnější fragmentace
 - segmenty mohou růst bez nutnosti přesunu v RAM
- proces pak má
 - jednu segmentovou tabulkou
 - stránkovou tabulkou
 - buď pro každý segment,
 - nebo jednu společnou
- ochrana a sdílení: na úrovni segmentů / stránek

Segmentace a stránkování – převod virtuální adresy (obrázek)



Architektura IA-32

- *Intel 64 and IA-32 Architectures: Software Developer's Manual: Volume 3A: System Programming Guide, Part 1* [online]. Intel, September 2015, c 1997–2015 [cit. 2015-09-28]. Odkaz: <intel.com>.
- podporuje segmentaci
 - poskytuje izolaci (ochranu) modulů
 - v chráněném režimu **nelze vypnout** (Ize resetem)
- podporuje stránkování
 - virtuální paměť typu demand-paged
 - implementuje také izolaci procesů

IA-32 – segmentace

- segmentace poskytuje rozdělení procesorem adresovatelné paměti (**lineární adresový prostor**) do chráněných segmentů
 - segmenty jsou učeny pro kód, data, stack procesů nebo pro systémové struktury (TSS, LDT apod.)
 - každý proces má svou vlastní sadu segmentů
 - segmenty zajišťují izolaci procesů a umožňují nastavit omezení možných operací
 - čtení, zápis, provádění

IA-32 – segmentace – adresa

- **logická adresa** (vzdálený ukazatel, **far pointer**)
 - **selektor** – unikátní identifikátor segmentu (16 bitů)
 - 13 bitů index, 1 bit Table Indicator (GDT/LDT), 2 bity RPL
 - index do globální / lokální tabulky deskriptorů (GDT, LDT)
 - deskriptor – položka GDT / LDT – určuje bázovou adresu segmentu, jeho velikost a práva na něm
 - RPL (Requested Privilege Level) – stupeň ochrany (0–3)
 - **offset** – relativní adresa vzhledem k bázové adrese
 - 32 bitů v režimu IA-32, 64 bitů v režimu IA-32e
 - bázová adresa + offset = lineární adresa
 - logická adresa (nespojitého prostoru procesu) se převádí na lineární adresu (do spojitého prostoru)

IA-32 – basic flat model

- **basic flat model** (plochý model paměti)
 - OS i procesy mají přístup k celé nesegmentované paměti
 - musejí se vytvořit dva popisovače segmentů
 - kódový segment (registrový CS) – nelze zapisovat
 - datový segment (registry DS, SS, ES, FS, GS)
 - oba segmenty mapují celý adresový prostor (4 GiB)
- používáno v 64bitovém režimu IA-32e (x86_64), který de facto nepoužívá segmentaci
 - ignorují se bázová adresa (je vždy 0), limit i atributy

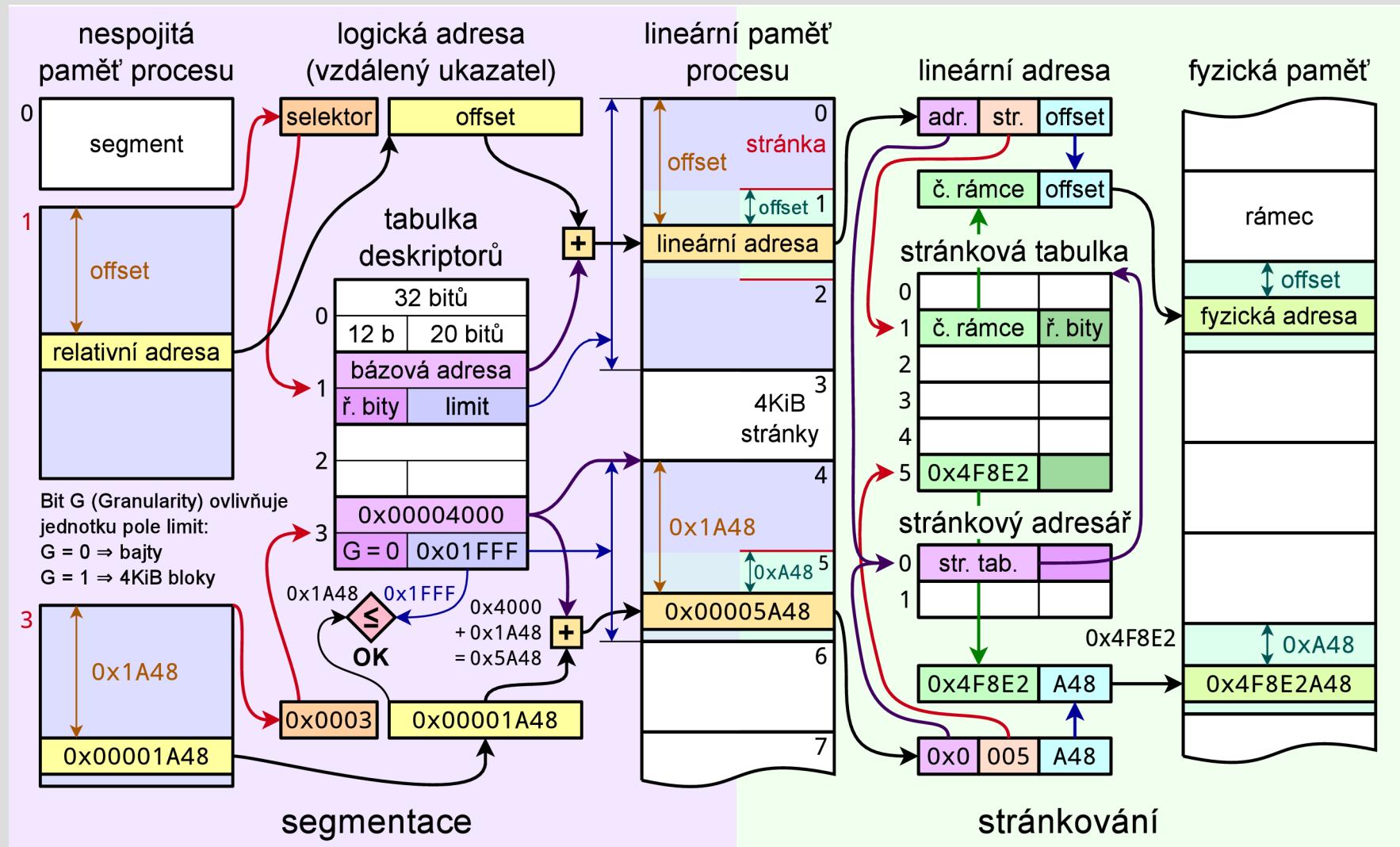
IA-32 – protected flat model

- **protected flat model** (chráněný plochý model)
 - podobné basic flat model, pouze limitní registry jsou nastaveny na velikost skutečně instalované fyzické paměti
 - výjimka ochrany je generovaná při přístupu do neexistující paměti
 - minimální ochrana paměti
 - další mechanismy ochrany paměti a izolaci lze nastavit použitím stránkování
- používáno 32bitovými systémy Linux i Windows

IA-32 – multi-segment model

- **multi-segment model** (vícesegmentový model)
 - plné využívání možností segmentace
 - paměť procesu je rozdělena na několik segmentů
 - každý proces má segmentovou tabulku s informacemi o segmentech (adresa, limit, práva)
 - segmenty mohou mít různá oprávnění včetně provádění určitých operací – stupně ochrany (ring levels)
 - přístup a práva kontroluje hardware
- používáno systémem OS/2

IA-32 – segmentace a stránkování, převod virtuální adresy (obrázek)



Posixové sdílení paměti

- hlavičkové soubory

```
#include <sys/types.h>
#include <sys/mman.h>
#include <sys/stat.h>    // pro konstanty S_I*
#include <fcntl.h>        // pro konstanty O_*
```

- knihovna librt – nutnost linkování
`gcc ... -lrt ...`
- se sdíleným paměťovým objektem se pracuje podobně jako se souborem – `shm_open(3)`
- objekt lze připojit do paměti procesu – `mmap(2)`

Sdílení paměti – vytvoření oblasti

- vytvoření nebo otevření – `shm_open(3)`

```
int shm_open(const char *name, int oflag,  
mode_t mode);
```

- vytvoří sdílený paměťový objekt – podobné `open(2)`
 - `oflag` – způsob otevření (`O_CREAT`, `O_RDWR`, ...)
 - `mode` – přístupová práva (`S_IRWXU`, `S_IRUSR`, ...) – respektuje se nastavení `umask(2)`
- přenositelnost: jméno začíná / (další / neobsahuje)
- vrací **memory descriptor** – ten se používá dále
 - nastavení velikosti nového objektu – `ftruncate(2)`
 - promítnutí objektu do paměti procesu – `mmap(2)`

Sdílení paměti – velikost, zrušení

- nastavení velikosti sdílené paměti – ftruncate(2)

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

int ftruncate(int fd, off_t length);
```

- nastaví velikost objektu daného **fd** na **length**
- nově alokovaná paměť je inicializovaná nulami

- zrušení – shm_unlink(3)

```
int shm_unlink(const char *name);
```

- podobné unlink(2) – odstraní jméno objektu
- po odpojení objektu se uvolní i paměť

Sdílení paměti – promítnutí objektu do paměti procesu (1)

- promítnutí objektu do paměti procesu – mmap(2)

```
#include <sys/mman.h>

void *mmap(void *start, size_t length,
           int prot, int flags, int fd, off_t offset);
```

- vrací adresu, na kterou jádro OS zobrazilo obsah souboru či sdílenou paměť s deskriptorem **fd** o velikosti **length** od pozice **offset**
- pokud **start** není **NULL**, požaduje se tato adresa
 - jádro bere parametr pouze jako vodítko (tip)
- **start** i **offset** musí být na hranici stránky

Sdílení paměti – promítnutí objektu do paměti procesu (2)

- promítnutí objektu do paměti procesu – mmap(2)

```
void *mmap(void *start, size_t length,  
           int prot, int flags, int fd, off_t offset);
```

- **prot** určuje režim přístupu
 - PROT_EXEC, PROT_READ, PROT_WRITE, PROT_NONE
 - více hodnot se nastavuje operací OR
- **flags** nastavuje mj. viditelnost změn jinými procesy
 - MAP_SHARED, MAP_PRIVATE, MAP_FIXED, ...
- při chybě vrací MAP_FAILED a nastaví errno

Sdílení paměti – odpojení objektu z paměti procesu

- odpojení sdílené paměti – munmap(2)

```
#include <sys/mman.h>  
  
void *munmap(void *start, size_t length);
```

- zruší zobrazení objektu do virtuální paměti procesu
 - další odkazy do oblasti jsou neplatné a způsobují chybu
- ukončení procesu ruší zobrazení automaticky
- **uzavření deskriptoru nezruší zobrazení**
- shm_unlink také nezruší zobrazení
 - pouze odstraní jméno objektu

Paměťově orientované vstupně-výstupní operace

- memory-mapped IO – mmap(2)
- umožňuje k blokům souboru přistupovat jako k blokům paměti
 - zjednodušuje přístup k souboru
 - soubor je načítán do paměti jádrem OS metodou demand-paging
 - více procesů může promítnout do své paměti stejný soubor
 - stránky pak mohou být mezi procesy sdíleny

Sdílení paměti System V IPC

- alokace – `shmget(2)`

```
#include <sys/ipc.h>
#include <sys/shm.h>

int shmget(key_t key, size_t size, int shmflg);
```

- vrátí identifikátor sdíleného paměťového segmentu podle klíče `key`, při chybě vrací `-1` (nastaví `errno`)
- nový sdílený segment o velikosti `size` (zvětšený na násobek `PAGE_SIZE`) je vytvořen, pokud
 - `key = IPC_PRIVATE` (lepší název by byl `IPC_NEW`) nebo
 - `key ≠ IPC_PRIVATE`, segment neexistuje a `shmflg` obsahuje `IPC_CREAT`

Sdílení paměti System V – alokace

- alokace (pokračování) – `shmget(2)`

```
int shmget(key_t key, size_t size, int shmflg);
```

- nově alokovaný segment je vyplněn nulami
- struktura `shmid_ds` je naplněna – viz `shmctl(2)`
 - vlastník a skupina podle volajícího procesu, práva a velikost podle parametrů, čas modifikace na aktuální, zbytek parametrů je vynulován (včetně `shm_nattach`)
- práva lze specifikovat dolními 9 bity v `shmflg`
- je-li dáno `IPC_CREAT` a `IPC_EXCL` a segment existuje, vrací chybu (`EEXIST`)

Sdílení paměti System V – operace

- operace se sdílenou pamětí – `shmop(2)`

```
#include <sys/ipc.h>
#include <sys/shm.h>

void *shmat(int shmid, const void *shmaddr,
            int shmflg);
```

- připojí segment sdílené paměti s id **shmid**
 - adresu necháme zvolit systémem zadáním **NULL**
- odpojí segment paměti předtím připojený **shmat**
- sníží počet odvolávek – **shm_nattach**

Sdílení paměti System V – ovládání

- ovládání segmentu sdílené paměti – `shmctl(2)`

```
#include <sys/ipc.h>
#include <sys/shm.h>

int shmctl(int shmid, int *cmd,
           struct shmid_ds *buf);
```

- vykoná příkaz `cmd` na sdíleném segmentu
 - `IPC_STAT` (zjištění info), `IPC_SET` (nastavení práv), `IPC_RMID` (nastavení značky pro odstranění segmentu)
- datová struktura `shmid_ds` obsahuje
 - práva, vlastníka, velikost, počet připojení, časy (připojení, odpojení, změny), PID (alokátor, poslední přístup), ...

OS – Správa souborů

Tomáš Hudec

Tomas.Hudec@upce.cz

<http://fei-as.upceucebny.cz/usr/hudec/vyuka/os/>

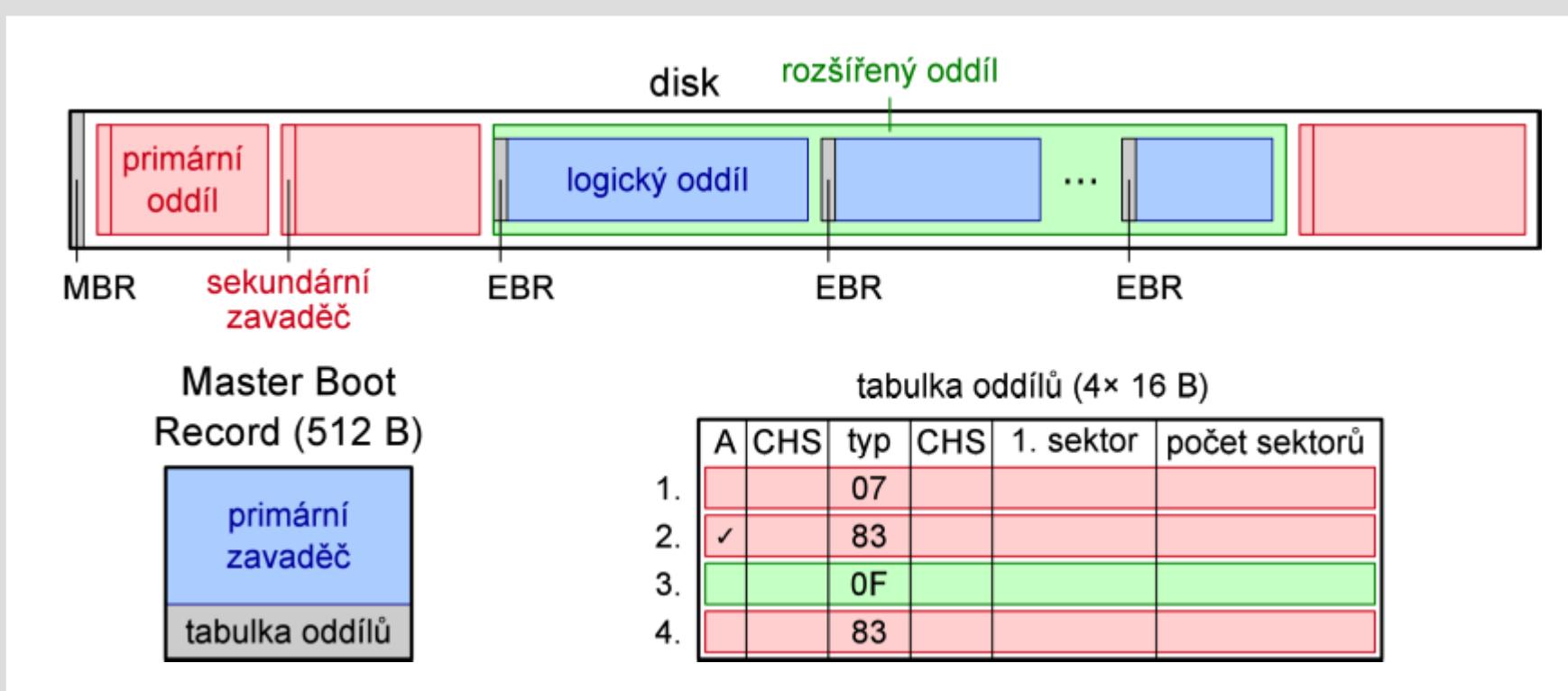
Úložiště, důvody dělení na oddíly

- úložiště (**storage**)
 - magnetický (mechanický) disk, SSD (Solid State)
 - lze dělit na **oddíly (partitions)**
 - jiný souborový systém (např. pro jiný OS)
 - snadnější správa – oddělení systému od dat
 - např. adresáře uživatelů nemohou zaplnit systémový disk
 - možnost snadnější obnovy části dat při přepsání či při HW chybě – chyba se může týkat jen některého oddílu
 - oddíl pro swap – na rozdíl od souboru oddíl netrpí datovou fragmentací – rychlejší přístup

Diskový oddíl

- **MBR** (Master Boot Record) – první sektor disku
 - tabulka rozdělení disku na oddíly (čtyři záznamy)
 - primární – lze z něj zavést OS sekundárním zavaděčem
 - rozšířený – dělí se na logické disky
 - zavaděč systému, může být univerzální – skok na sekundární zavaděč na aktivním primárním oddíle
- **GPT** (GUID Partition Table) – standard UEFI
 - za „protective MBR“ + záloha na konci disku
 - až 128 (primárních) oddílů (1 záznam má 128 B)
 - odstraňuje limity MBR, nutné pro disky nad 2 TiB

Diskové oddíly MBR (obrázek)



EBR (Extended BR) má v tabulce oddílů dva záznamy:
následující logický oddíl + odkaz na další EBR

Soubor

- **soubor (file)**
 - univerzální forma dlouhodobého uložení dat v sekundární paměti (na disku)
 - vstupní data pro programy
 - uložení výstupních dat
 - uložení programů
 - distribuce, archivace programů a dat
 - sdílení dat

Adresář

- adresář, složka (directory, folder)
 - umožňuje hierarchické uspořádání souborů
 - obvykle je reprezentován speciálním souborem, jehož obsah je interpretován systémem
 - obsahuje informace o souborech
- adresáře tvoří hierarchickou strukturu – strom
 - hlavní adresář se označuje jako kořenový (root)
 - adresář aktuální – pracovní (current, working)
 - cesta (path) určuje logické umístění souboru
 - absolutní (od kořene), relativní (z aktuálního adresáře)

Systém správy souborů

- **souborový systém (file system, FS)**
 - umožňuje uživatelům a procesům přístup k souborům (podle jména) na paměťovém médiu
 - vytváří logickou (adresářovou) strukturu souborů
 - eviduje metadata souborů (atributy, oprávnění)
 - organizuje uspořádání souborů na médiu
 - programátoři nemusejí vyvíjet vlastní SW prostředky pro manipulaci s daty na médiu, alokaci místa apod.
 - sjednocuje přístup k datům různého typu a původu
 - vstup a výstup na různé periferie se neliší

Příklady souborových systémů

- unixové systémy
 - BSD: UFS, Solaris: UFS, ZFS, IRIX: XFS, AIX: JFS
 - Linux: ext2/3/4, ReiserFS, Btrfs, NILFS, F2FS
 - podporuje mnoho dalších unixových i neunixových
- OS X [ou es ten], iOS, Mac OS
 - HFS, HFS Plus, UFS (od OS X; z NeXTSTEP, BSD)
- Windows
 - FAT, NTFS, exFAT, ReFS (Resilient FS, od Win8)
- síťové – NFS, SMB (CIFS)

Úložiště

- **DAS** (Directly Attached Storage)
 - lokální paměťové úložiště (blokové zařízení)
 - SCSI, SATA, IDE
- **NAS** (Network Attached Storage)
 - souborový systém zpřístupněný síťovým protokolem
 - NFS, CIFS (SMB, Samba), AFS
- **SAN** (Storage Area Network)
 - blokové zařízení zpřístupněné síťovým protokolem
 - iSCSI, Fibre Channel

RAID

- Redundant Array of Independent Disks
 - zapojení více disků do jednoho virtuálního zařízení
 - HW: závislé na řadiči, SW: výpočet parity v CPU
 - výhodou je rychlosť a/nebo redundance
 - **RAID-0** – stripping: prokládané ukládání
 - rychlejší, ale výpadek disku znamená ztrátu všeho
 - **RAID-1** – mirroring: zrcadlené ukládání
 - rychlejší, redundantní, kapacita jen jednoho disku
 - **RAID-5** – stripping + parity: prokládané s paritou
 - rychlejší, redundantní, kapacita snížená o jeden disk

Metadata

- metadata (data o datech)
 - unixové systémy – superblok, i-uzel (i-node)
 - NTFS – boot sektor, Master File Table (MFT)
 - FAT – boot sektor, adresář a alokační tabulka
- metadata souborového systému (FS)
 - velikost FS, vlastnosti FS, seznam volných bloků, odkaz na evidenci metadat souborů
- metadata souborů – atributy
 - typ, velikost, čas změny, oprávnění, umístění, ...

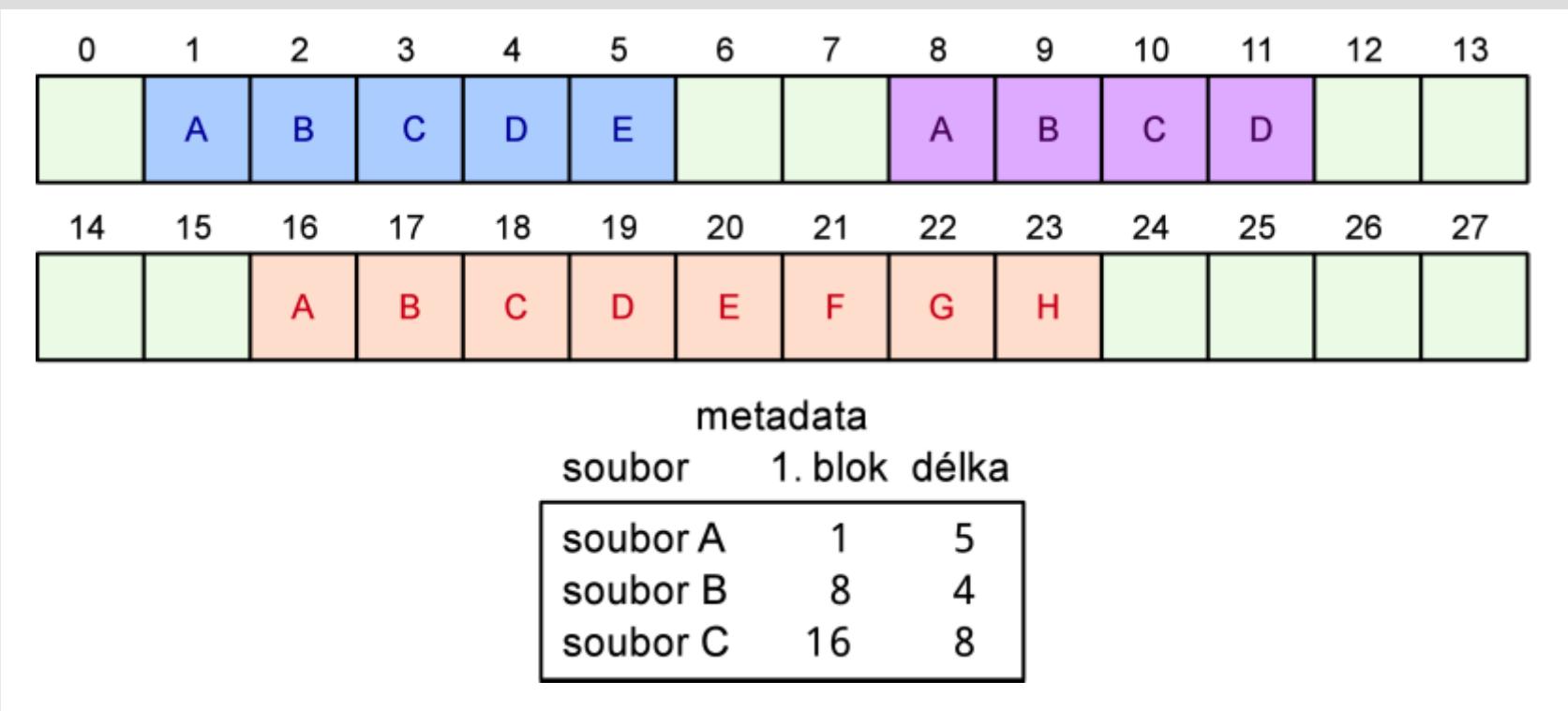
Alokace prostoru na médiu

- soubory alokují prostor po **alokačních blocích**
 - alokační blok (**cluster**) má velikost několik sektorů
 - obvykle mocnina dvou, tj. 1, 2, 4, 8, 16, ...
 - sektor je nejmenší alokovatelná jednotka na médiu
 - typicky 512 bajtů, 2 KiB (CD) nebo 4 KiB (disky od r. 2010)
 - blok nemusí být využit zcela – **vnitřní fragmentace**
- metody alokace
 - souvislá (contiguous allocation)
 - řetězená (chained allocation)
 - indexová (indexed allocation)

Souvislá alokace

- souboru jsou přiděleny po sobě jdoucí bloky
- v metadatech souboru se eviduje:
 - adresa prvního (počátečního) bloku
 - velikost souboru (v alokačních blocích)
- při alokaci prostoru na médiu dochází k **vnější fragmentaci**
 - vznikají díry, které je obtížné využít
 - soubor nemůže (bez přemístění) růst nad limit volného prostoru za posledním blokem

Souvislá alokace (obrázek)

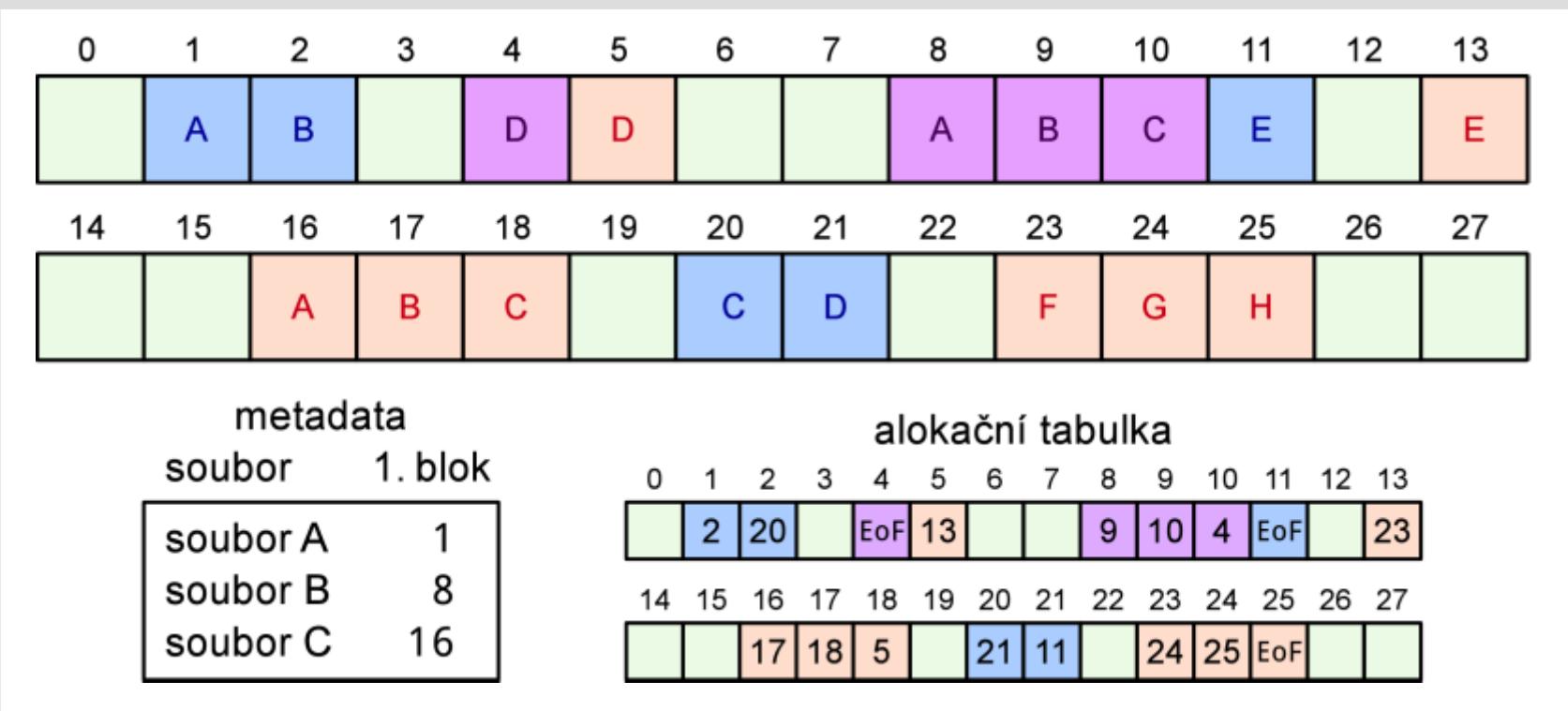


metadata obsahují odkaz na 1. blok a délku

Řetězená alokace

- alokují se jednotlivé bloky
- alokační tabulka obsahuje zřetězený seznam
 - každý blok obsahuje odkaz na následující blok
- **odstraněna vnější fragmentace**
 - souboru lze přidělit libovolný další blok
- logicky sousedící bloky mohou být na různých místech na médiu – **datová fragmentace**
- př.: souborový systém FAT (DOS / Windows)

Řetězená alokace (obrázek)



metadata obsahují odkaz na 1. blok,
alokační tabulka obsahuje pro každý blok odkaz na další

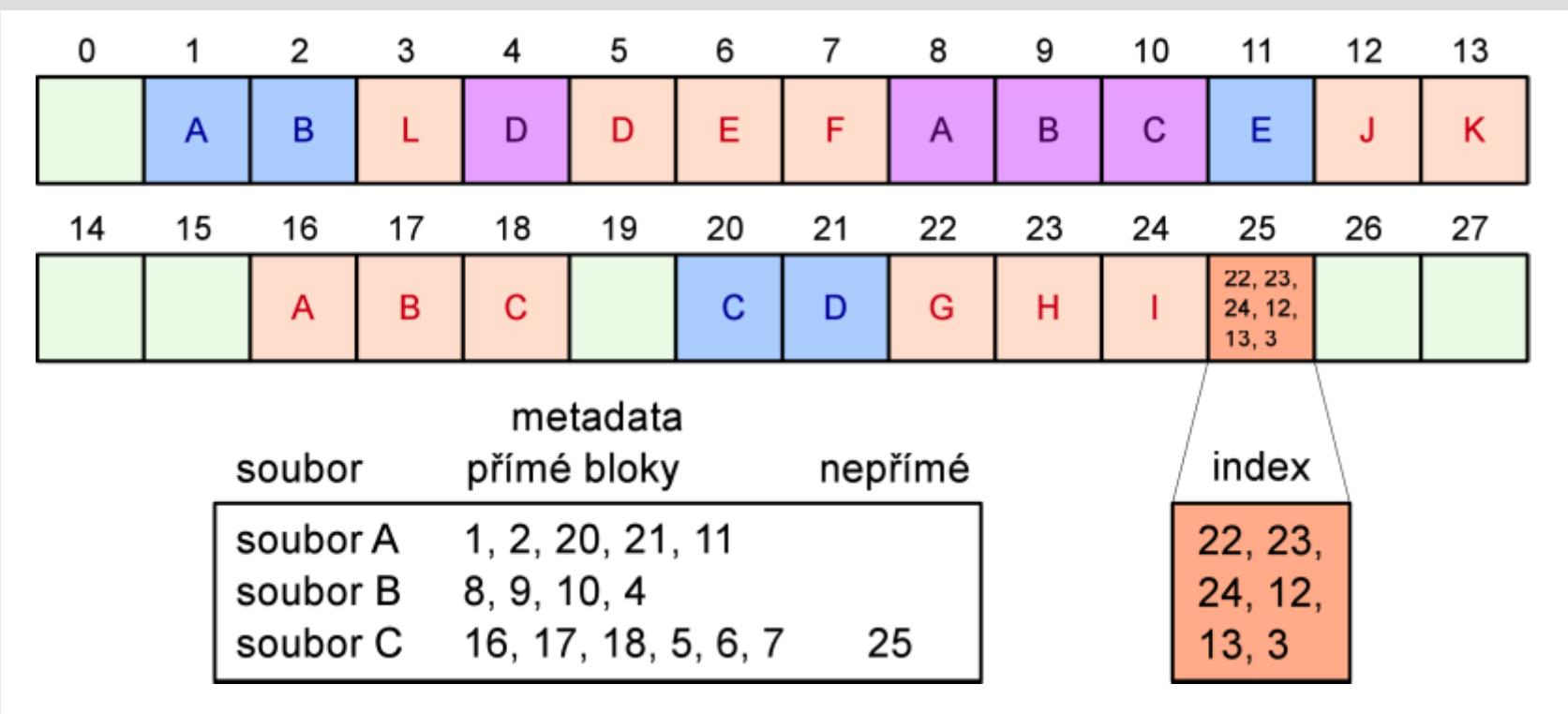
File Allocation Table (FAT)

- FAT12, FAT16, FAT32
 - číslo udává rozsah – počet bitů na položku
 - maximální počet alkokačních bloků (clusterů)
 - maximální velikost FS pro danou velikost clusteru
 - FAT16 a cluster osm 512B sektorů: $2^{16} \cdot 4 \text{ KiB} = 256 \text{ MiB}$
 - FAT12 a cluster čtyři 512B sektory: $2^{12} \cdot 2 \text{ KiB} = 8 \text{ MiB}$
 - FAT32 pro číslo clusteru užívá jen 28 bitů ze 32
 - max. počet clusterů je tím prakticky omezen na 2^{28}
 - velikost FAT je daná počtem clusterů na FS
 - velikost FS / velikost clusteru · velikost položky FAT

Indexová alokace

- index obsahuje seznam přidělených bloků
 - odstraněna vnější fragmentace
- index se může odkazovat na blok s indexem
 - nepřímý index (víceúrovňový)
- bloky mohou mít i různou velikost
 - redukce vnitřní fragmentace
- př.: unixové souborové systémy
 - i-uzel obsahuje několik přímých odkazů na bloky a (nepřímé) odkazy na další indexy bloků

Indexová alokace (obrázek)

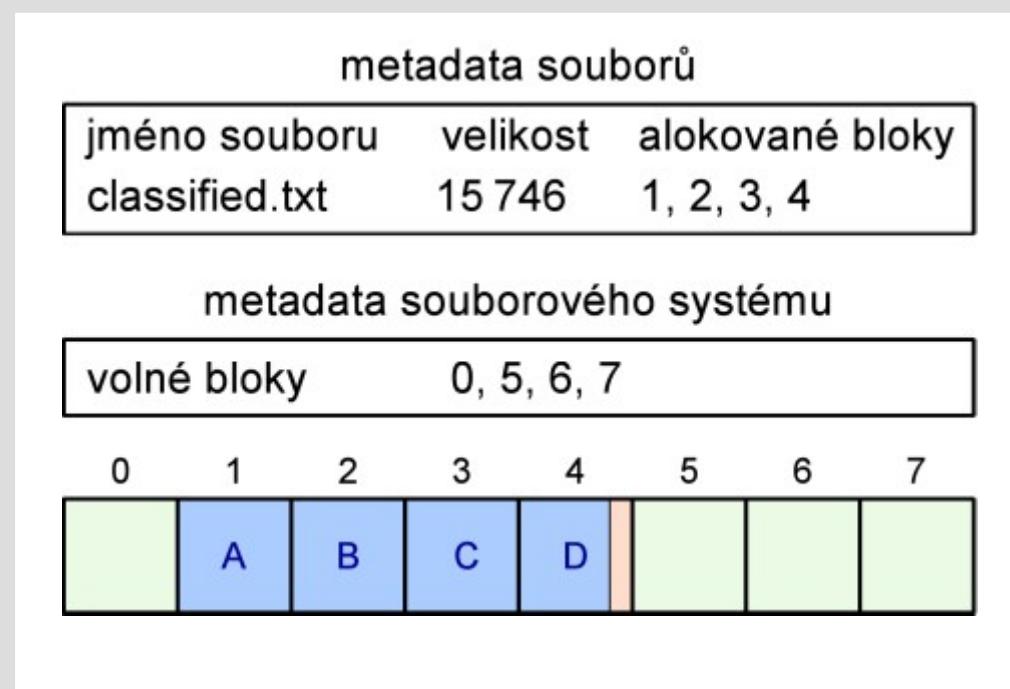


metadata obsahují index s odkazy na bloky, některé odkazy mohou být nepřímé – na blok s indexem

Konzistence souborového systému

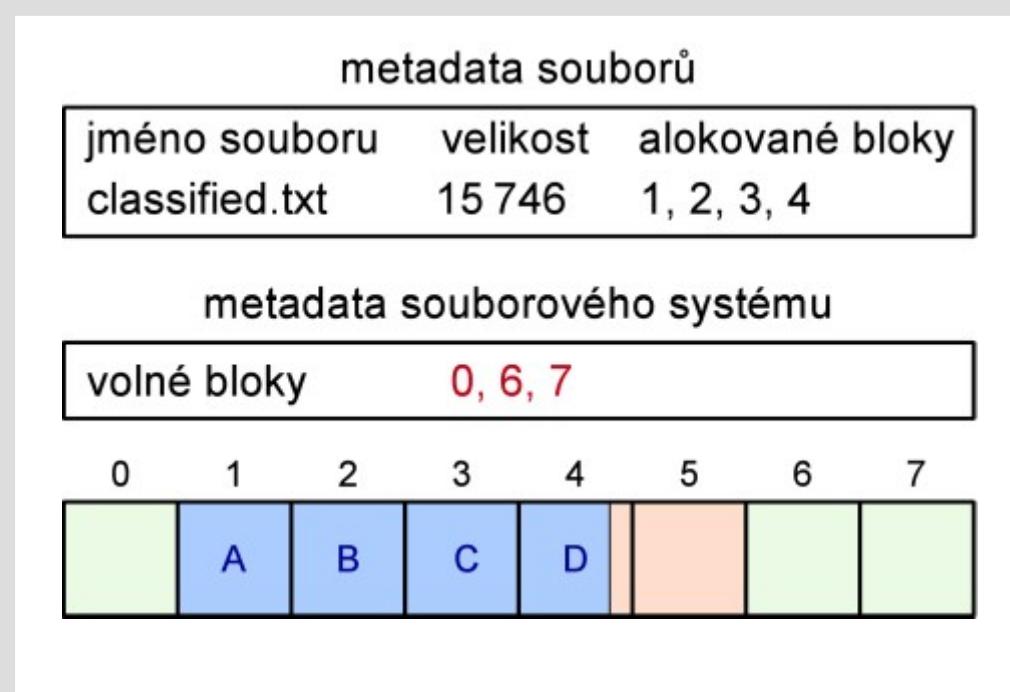
- zápis dat do souboru znamená provedení operací na různých místech na médiu
 - metadata souborového systému
 - alokace nových bloků, aktualizace seznamu volných bloků
 - datová část souborového systému
 - zápis nových dat souboru
 - metadata souboru
 - aktualizace přidělených bloků, velikosti, času změny
- změny probíhají postupně – **vznik nekonzistence**
 - po pádu systému je třeba provést kontrolu

Zápis do souboru (obrázek 1)



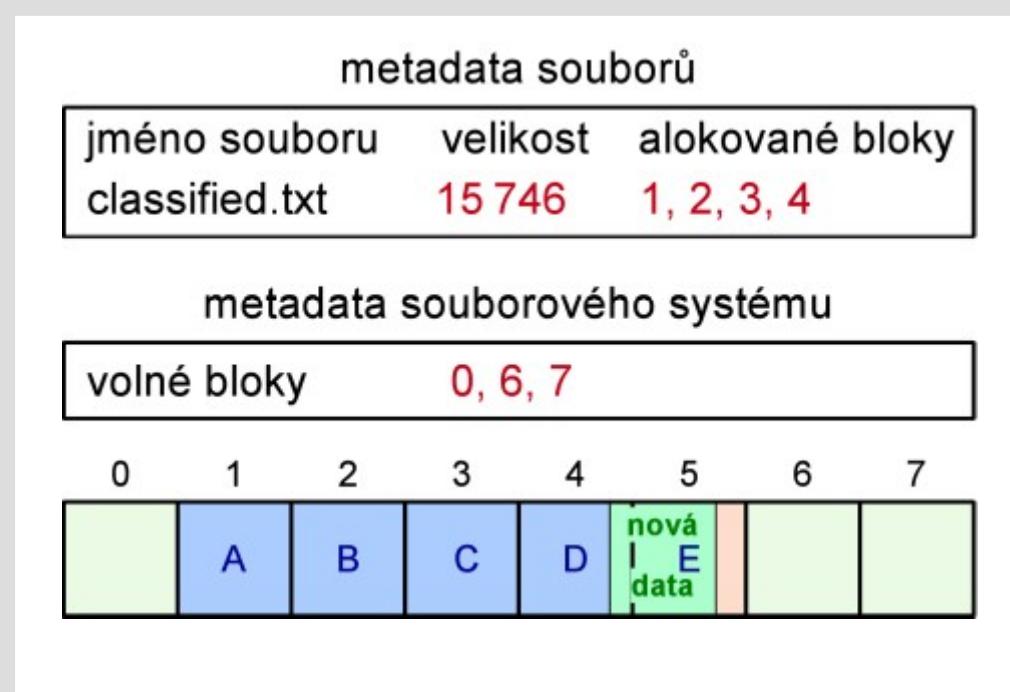
výchozí stav

Zápis do souboru (obrázek 2)



alokace bloků na souborovém systému
červeně jsou uvedena nekonzistentní metadata

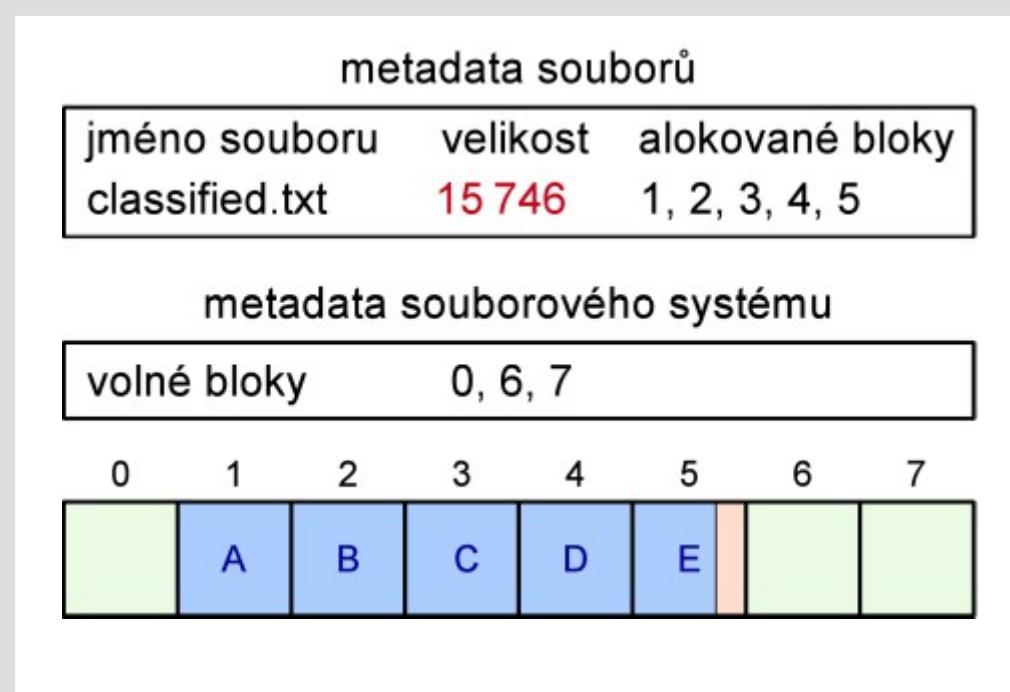
Zápis do souboru (obrázek 3)



zápis nových dat do souboru

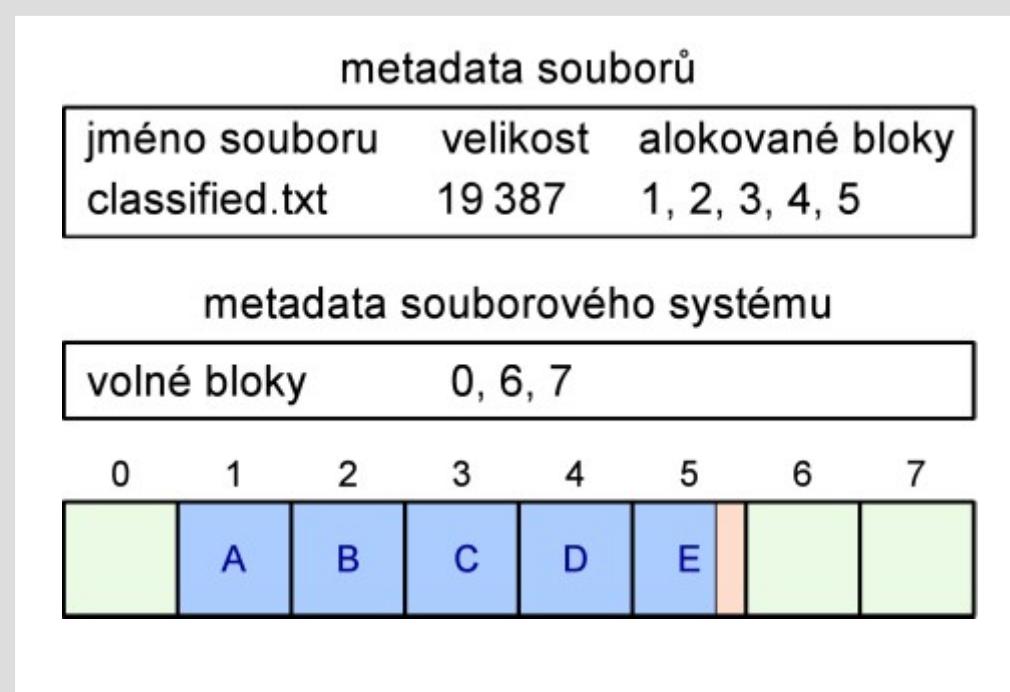
červeně jsou uvedena nekonzistentní metadata

Zápis do souboru (obrázek 4)



aktualizace alokovaných bloků souboru
červeně jsou uvedena nekonzistentní metadata

Zápis do souboru (obrázek 5)

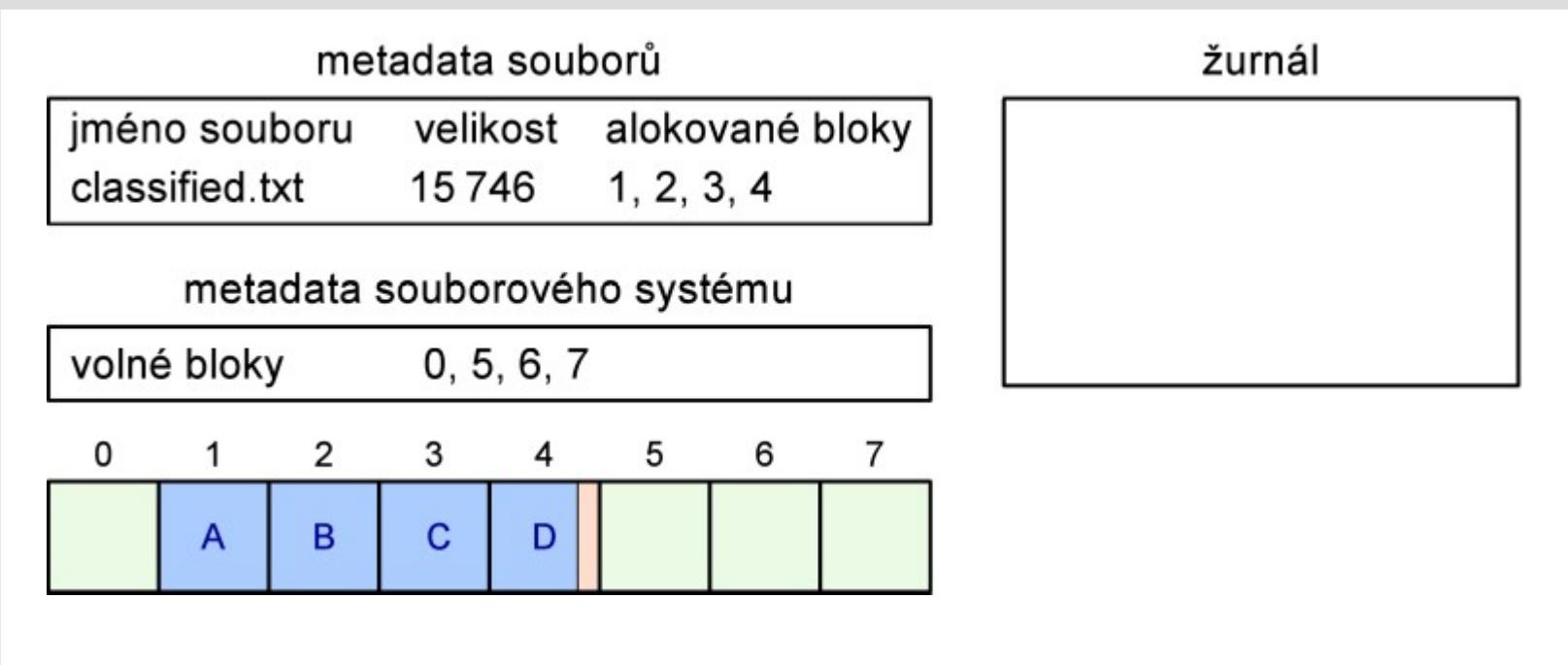


aktualizace velikosti souboru
konečný stav

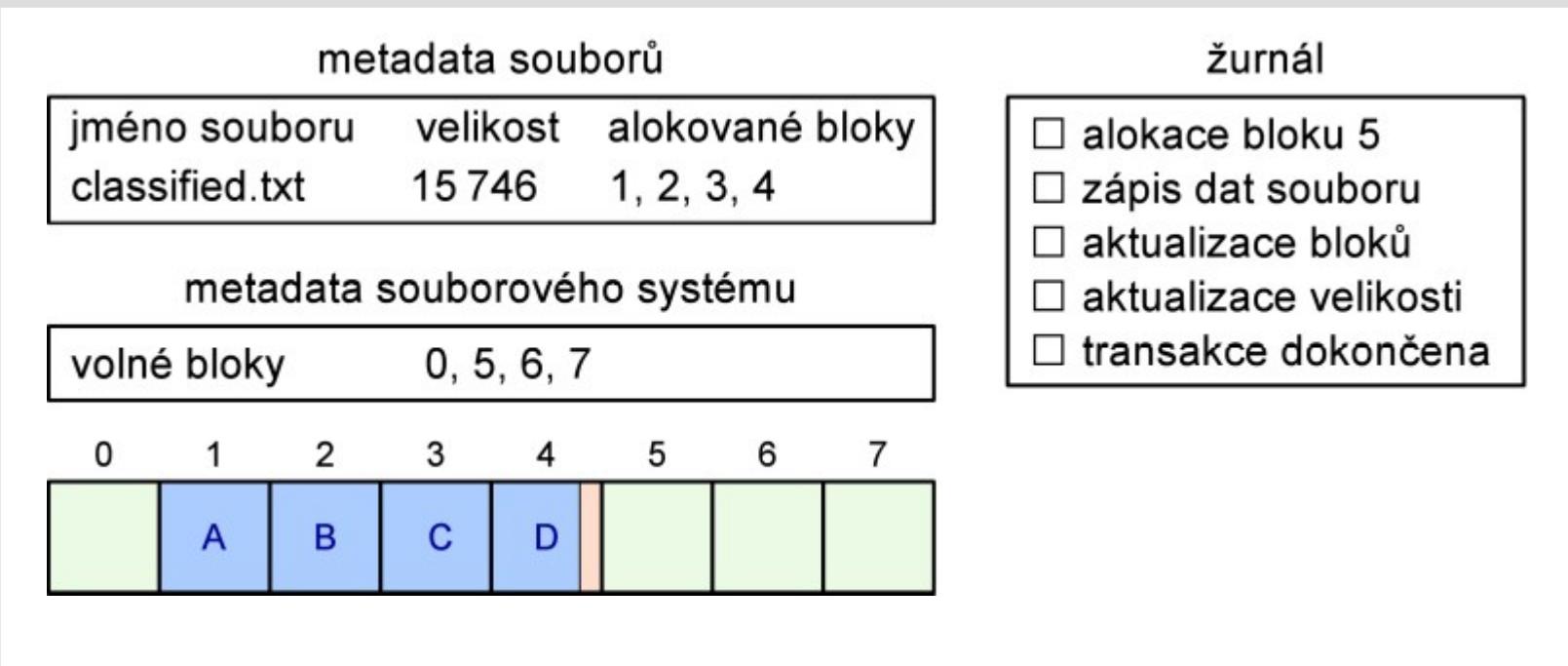
Zachování konzistence – žurnál

- žurnál (journal) – transakční log
 - zabraňuje ztrátě konzistence dat při pádu systému
 - žurnál nezabraňuje ztrátě dat při pádu systému
 - změny metadat (případně i dat) se zapisují do transakčního logu, což je kruhový buffer
 - teprve po potvrzení zápisu do žurnálu se provede patřičná změna souborového systému – **dvojí zápis = zpomalení**
 - při startu systému po pádu není třeba kontrolovat konzistenci celého souborového systému – **zrychlení**
 - prochází se žurnál a zkонтroluje se (a případně se opraví) konzistence pouze na místech posledních změn

Zápis do souboru s použitím žurnálu (obrázek 1)

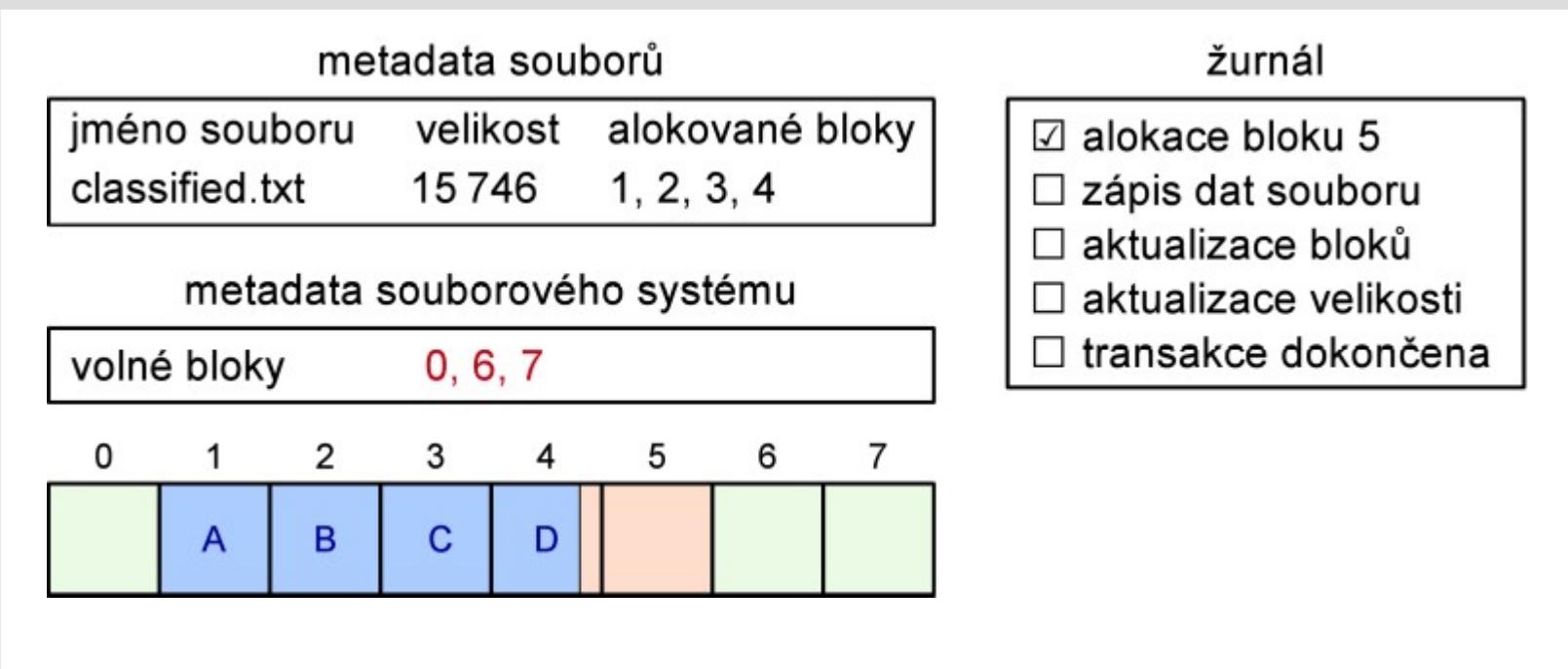


Zápis do souboru s použitím žurnálu (obrázek 2)



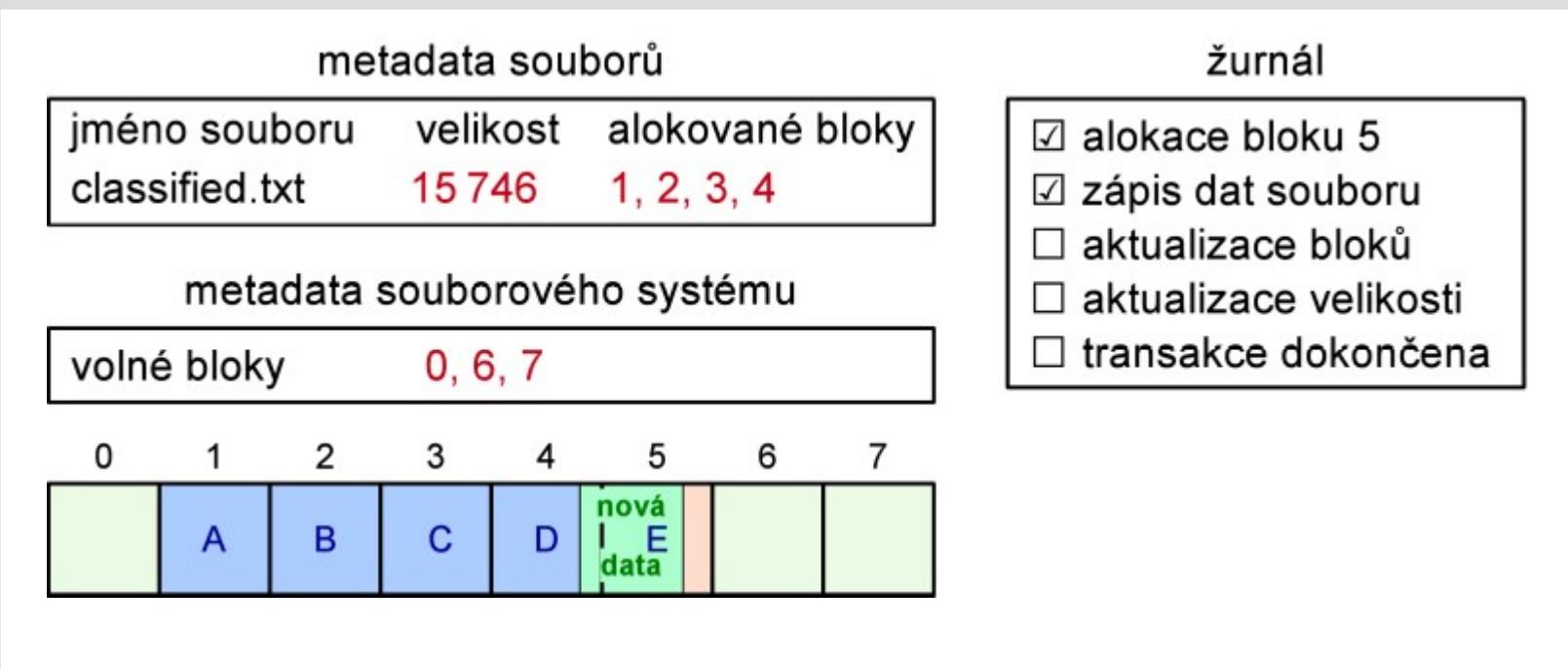
zápis transakce do žurnálu

Zápis do souboru s použitím žurnálu (obrázek 3)



alokace bloků na souborovém systému
červeně jsou uvedena nekonzistentní metadata

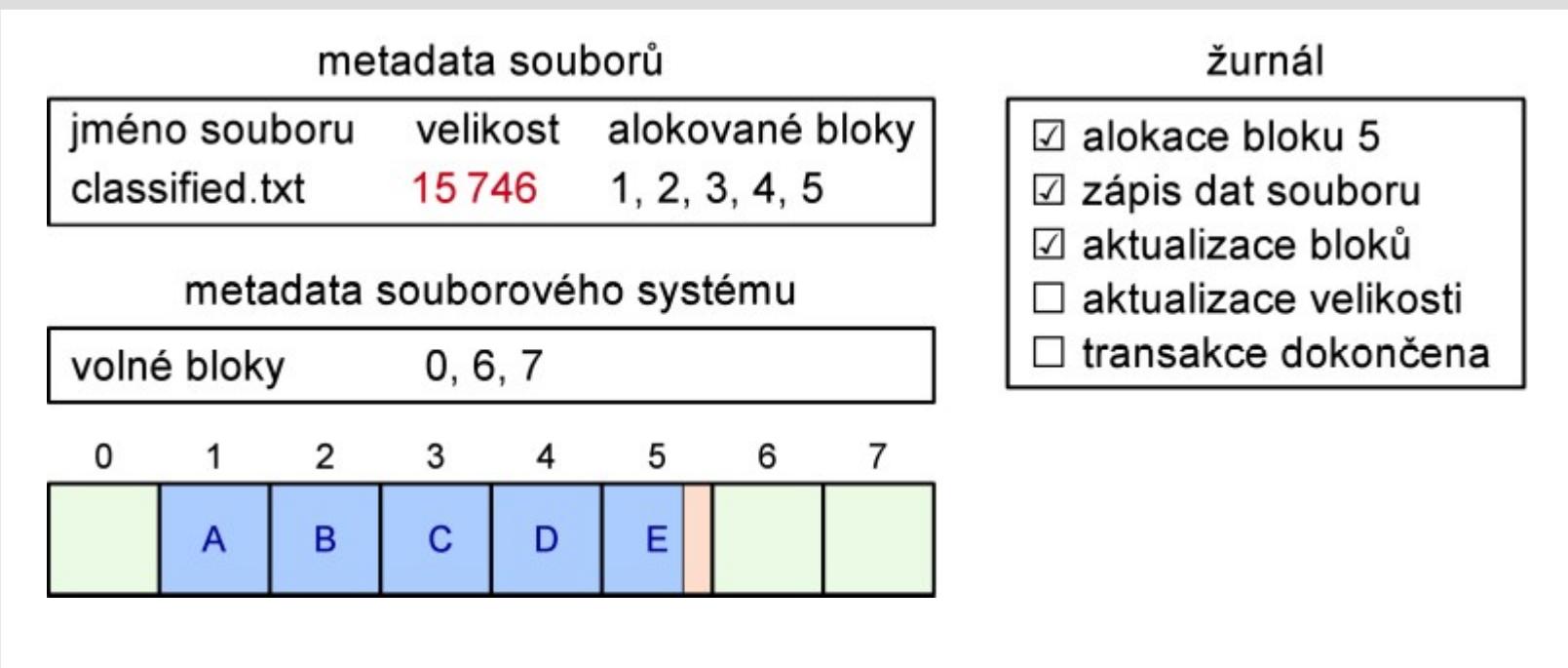
Zápis do souboru s použitím žurnálu (obrázek 4)



zápis nových dat do souboru

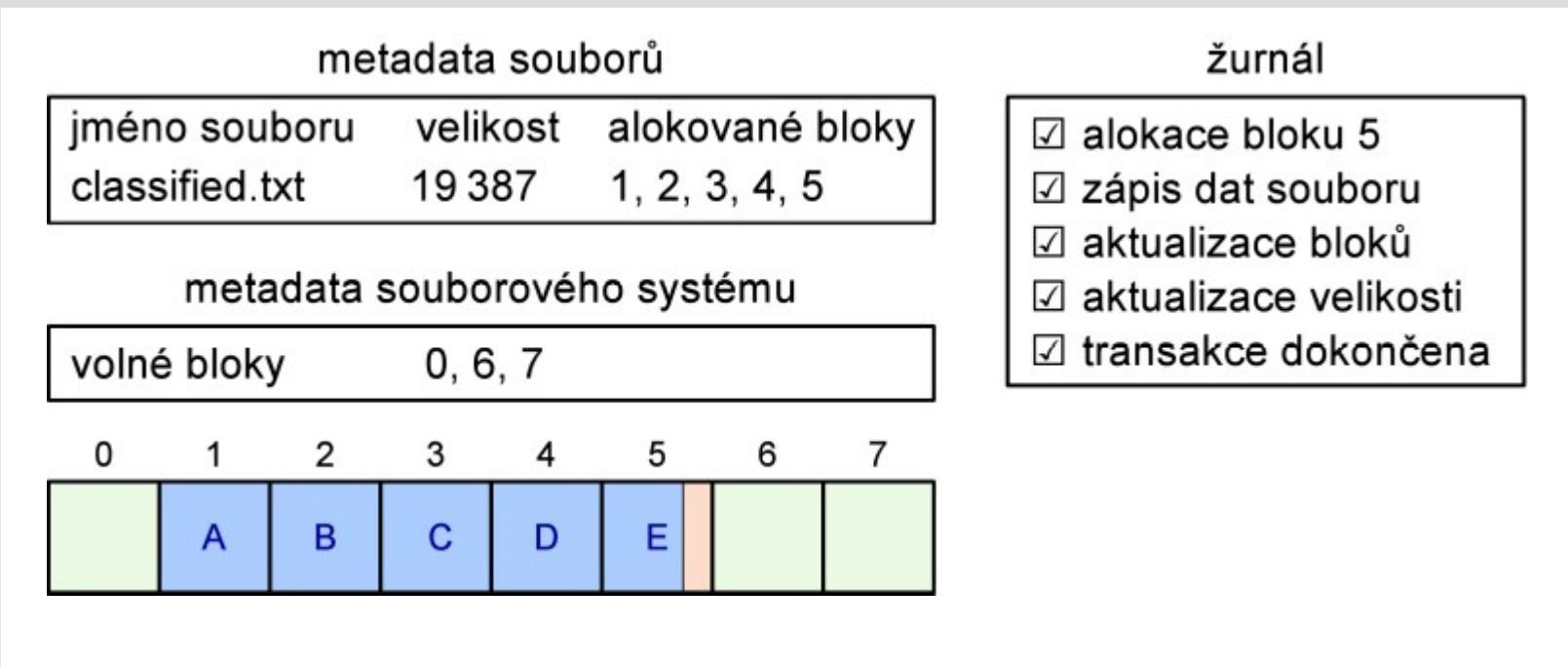
červeně jsou uvedena nekonzistentní metadata

Zápis do souboru s použitím žurnálu (obrázek 5)



aktualizace alokovaných bloků souboru
červeně jsou uvedena nekonzistentní metadata

Zápis do souboru s použitím žurnálu (obrázek 6)



žurnál

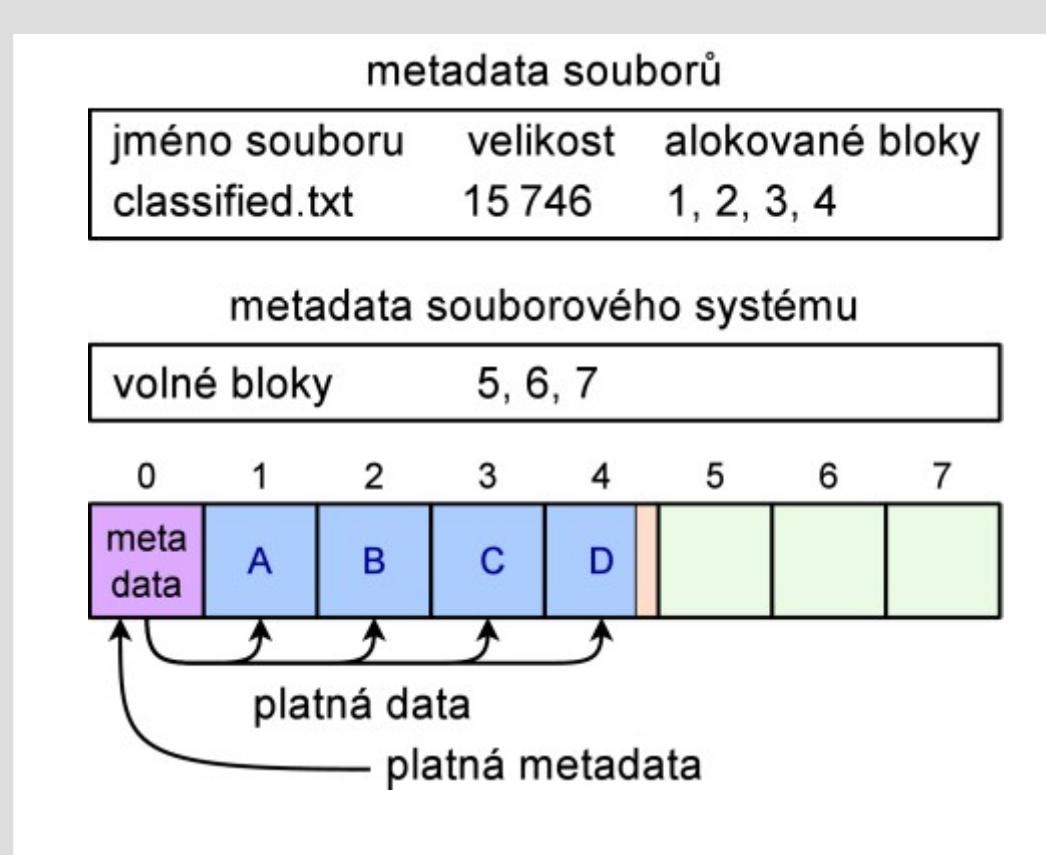
- alokace bloku 5
- zápis dat souboru
- aktualizace bloků
- aktualizace velikosti
- transakce dokončena

aktualizace velikosti souboru
dokončení transakce, konečný stav

Zachování konzistence – copy-on-write

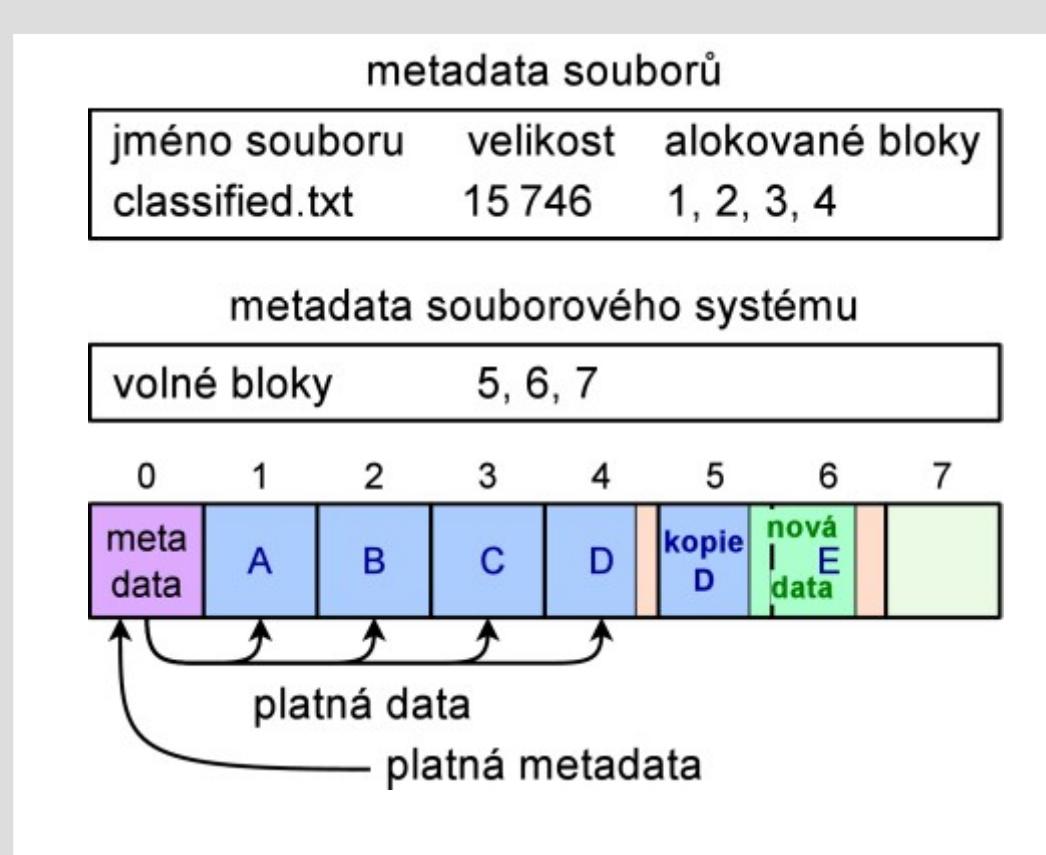
- **copy-on-write** – kopírování bloku při změně
 - nekonzistence vzniká při změně informací – zápisu
 - souborový systém **nikdy nemodifikuje** bloky na místě
 - při přepisu se **modifikuje kopie bloku**
 - stejným způsobem se aktualizují metadata
 - po dokončení zápisu se **atomicky** provede zneplatnění původních dat a metadat a potvrdí se platnost nových
 - **souborový systém je tedy vždy konzistentní**
 - data se nezapisují dvakrát (nevýhoda žurnálu odstraněna)
 - nevýhoda: větší datová fragmentace
 - souborové systémy: ZFS, Btrfs, NILFS, F2FS, ReFS

Zápis do souboru metodou copy-on-write (obrázek 1)



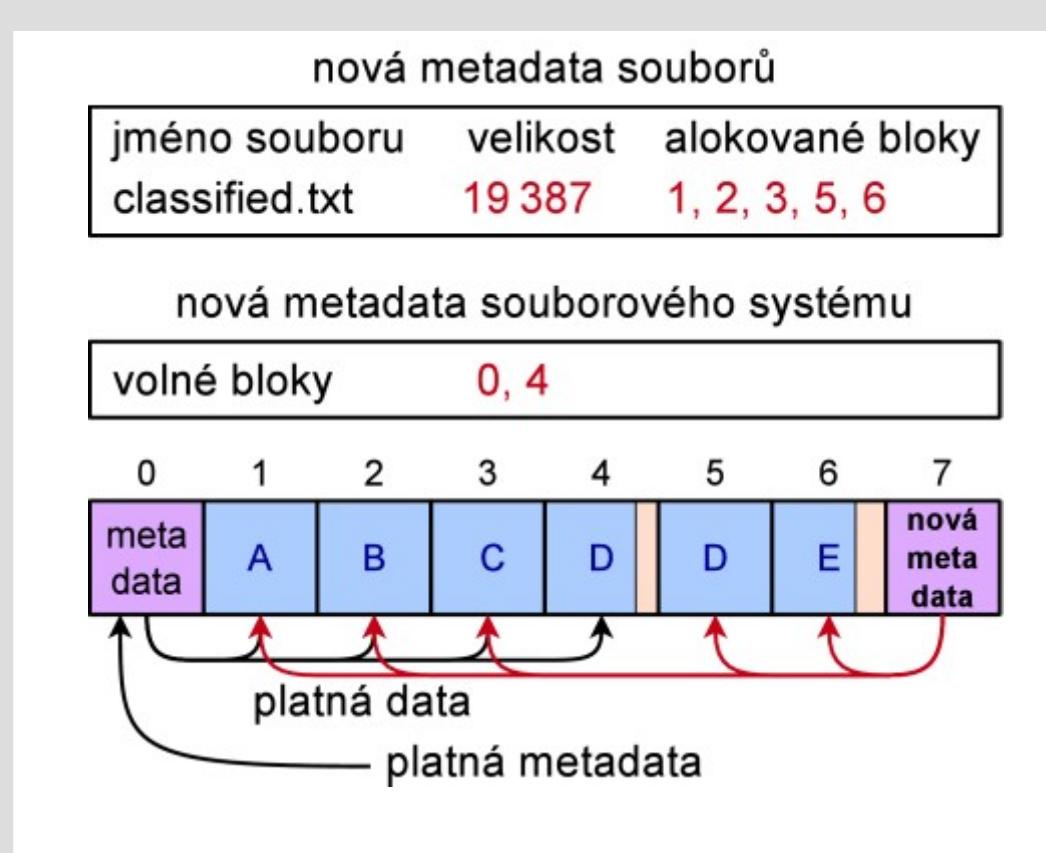
výchozí stav

Zápis do souboru metodou copy-on-write (obrázek 2)



zápis nových dat souboru

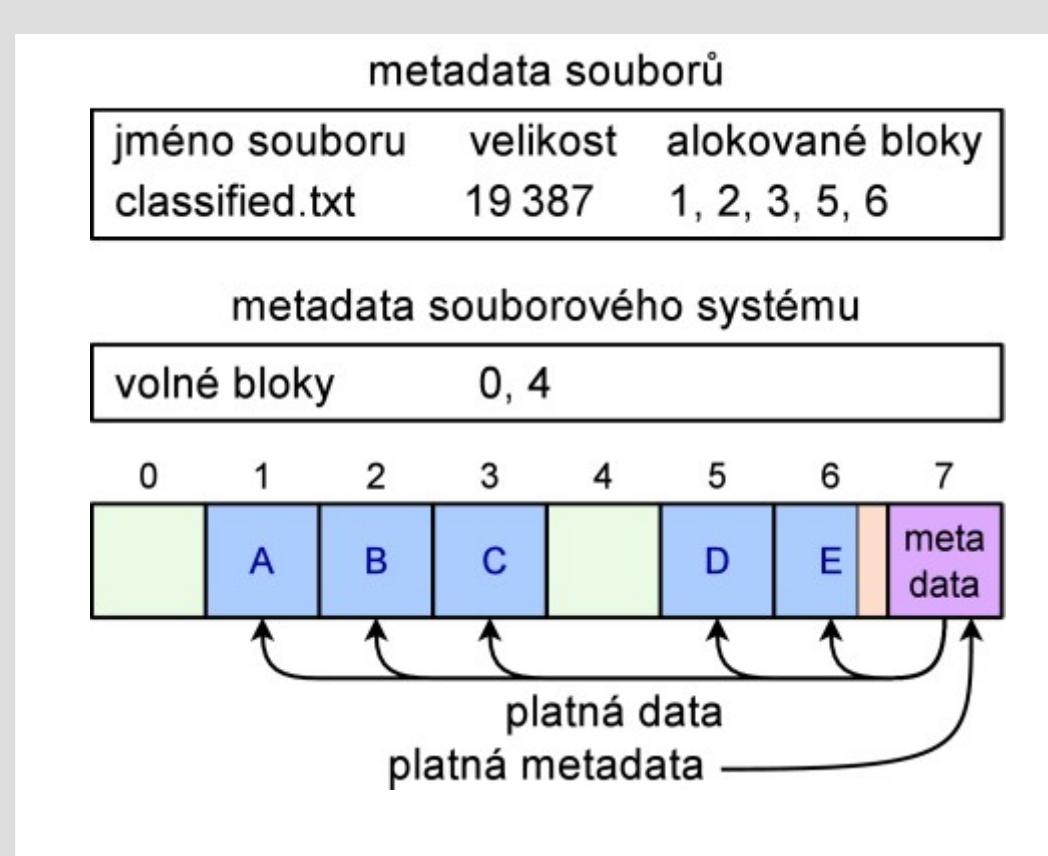
Zápis do souboru metodou copy-on-write (obrázek 3)



zápis nových metadat

zatím je stále platný blok s původními metadaty

Zápis do souboru metodou copy-on-write (obrázek 4)



nastavení platnosti nových dat a metadat
konečný stav

Konzistence souborového systému – příklady

- souborové systémy bez konzistence po pádu
 - FAT, exFAT, ext2, HFS (Apple), HPFS (IBM OS/2)
- žurnálovací souborové systémy
 - úplné: ext3, ext4, ReiserFS, UDF (transakční)
 - pouze metadata: NTFS (MS, transakční), JFS (IBM), XFS (SGI), HFS+ (od Mac OS X 10.2.2), UFS (od Solaris 7, FreeBSD 7.0), VMFS (VMware)
- souborové systémy s podporou copy-on-write
 - ZFS, Btrfs, NILFS, F2FS, ReFS

Speciální soubory – odkazy

- tvrdý (pevný) odkaz (hard link)
 - více jmen pro týž soubor (ne adresář) na stejném FS
 - data i metadata jsou na médiu uložena pouze jednou
 - ke smazání dojde při odstranění posledního odkazu
- symbolický odkaz (symbolic link), speciální typ
 - textová záměna za jiné jméno (absolutní či relativní)
 - změna jména je pro procesy transparentní
 - liší se od zástupce (shortcut), což je normální soubor
- NTFS junction point – odkaz na adresář

Speciální soubory – roura, socket, zařízení

- pojmenovaná roura (**named pipe**)
 - jednosměrný komunikační nástroj pro dva procesy
 - jeden proces zapisuje, jeden čte (má nezávislé ukazatele)
 - co bylo přečteno se odstraní
- pojmenovaný socket – adresa = jméno souboru
- zařízení (**device file**) – jednotný přístup k HW
 - blokový (s náhodným přístupem) – disky
 - znakový (proudový přístup po bajtech)
 - terminál, tiskárna, zvuková karta, skener

Speciální soubory – identifikace

- POSIX: první znak na řádku ve výpise `ls -l`
 - **directory** – adresář: jména souborů s čísly i-uzlů
 - **link** – symbolický odkaz: obsahuje jméno cíle
 - pozn.: hard link není speciální soubor
 - **pipe** – pojmenovaná roura
 - **socket** – pojmenovaný soket
 - **block** – blokový speciální soubor: bloková zařízení
 - **char** – znakový speciální soubor: znaková zařízení
 - identifikace zařízení pomocí čísel (hlavní a vedlejší)

Kritéria pro souborové systémy

- rychlosť prístupu
- snadnosť modifikácie
- ekonomie využití pamäti
 - minimalizácia nevyužitelného prostoru (vnútř bloků)
 - minimalizácia redundancie (indexácia, žurnály)
- snadnosť údržby
- spôsobnosť – zachovanie konzistencie
- podpora funkcií – práva, kvóty, speciálne súbory

Oprávnění obecně (1)

- **žádná** – se souborem nelze nijak manipulovat
 - nelze zjistit ani existenci souboru
- **znalost existence** – bez dalších oprávnění
- **provádění (execute)** – soubor lze spustit
 - soubor nemusí být možné číst (tedy ani kopírovat)
- **čtení (read)** – lze zjistit obsah souboru
 - lze tedy soubor zkopirovat

Oprávnění obecně (2)

- přidávání (**append**) – lze přidávat data na konec
- přepisování, zápis (**update, write**)
 - do souboru lze zapisovat a přepisovat jeho data
- mazání (**delete**) – soubor lze odstranit
- vytvoření (**create**) – lze vytvořit nový soubor
- změna oprávnění (**changing permissions**)
 - změna přístupových práv jiným uživatelům

Příklady oprávnění – UNIX

- unixové systémy
 - zvlášť práva pro vlastníka, skupinu a ostatní
 - soubor je vlastněn jedním uživatelem a jednou skupinou
 - **read** – čtení obsahu souboru
 - adresář: zobrazení položek adresáře
 - **write** – změna obsahu souboru
 - adresář: vytváření a mazání položek (libovolného vlastníka) v adresáři
 - **execute** – spuštění programu
 - adresář: vstup do adresáře, čtení metadat položek

Příklady oprávnění – Novell Netware

- Novell Netware
 - **s**upervisory – všechna práva
 - **a**ccess control – přidělování práv ostatním
 - **r**ead – čtení, adresář: vypsat obsah
 - **w**rite – zápis, adresář: zápis do existujících souborů
 - **c**reate – adresář: vytváření souborů,
soubor: obnovení smazaného souboru
 - **e**rase – smazání
 - **m**odify – změna atributů (nikoliv práv)
 - **f**ile scan – znalost existence souboru

Příklady oprávnění – NTFS

- NTFS (New Technology File System, Windows)
 - Full Control, Traverse Folder, Execute File, List Folder, Read Data, Read Attributes, Read Extended Attributes, Create Files, Write Data, Create Folders, Append Data, Write Attributes, Write Extended Attributes, Delete Subfolders and Files, Delete, Read Permissions, Change Permissions, Change Ownership

Příklady atributů – FAT

- souborový systém FAT
 - **read-only** – soubor je pouze pro čtení
 - soubor nelze smazat ani přepsat
 - **hidden** – skrytý, nezobrazuje se ve výpisu adresáře
 - **system** – systémový, skrytý + omezení odstranění
 - **archive** – soubor je připraven k archivaci (záloze)
 - atribut se nastavuje vždy při zápisu do souboru

Příklady atributů – NTFS

- souborový systém NTFS
 - read-only, hidden, system, archive – jako FAT
 - not content indexed – neindexovaný obsah souboru
 - off-line – data jsou na jiném (vzdáleném) zařízení
 - soubor není přístupný při provozu off-line
 - může též znamenat, že soubor je jen lokální kopí (cache)
 - temporary – dočasný soubor
 - compressed – automaticky komprimovaný soubor
 - encrypted – šifrovaný pomocí EFS
 - sparse – řídký soubor

Příklady atributů – extended FS (1)

- linuxový souborový systém ext2, ext3, ext4
 - **append only** – lze pouze přidávat na konec
 - **compressed** – komprimovaný (pouze ext4)
 - no **dump** – nearchivovat programem dump,
 - **extent format** – pro alokaci se používají velké souvislé oblasti bloků (extents, pouze ext4)
 - redukce datové fragmentace
 - **immutable** – soubor nelze nijak měnit ani odstranit
 - data **journaling** – žurnálovat i data souboru (od ext3)
 - no **Atime updates** – neaktualizuje se čas přístupu

Příklady atributů – extended FS (2)

- linuxový souborový systém ext2, ext3, ext4
 - **secure deletion** – bezpečné mazání (pouze ext4)
 - při mazání se obsah bloků přepíše nulami
 - **undeletable** – při mazání se uloží data (pouze ext4)
 - umožní obnovu smazaného souboru
 - no **tail-merging** – neslučovat poslední blok souboru
 - **Synchronous updates** – synchronní zápis do souboru
 - **synchronous Directory updates** – dtto do adresáře
 - **Top of directory hierarchy** – podadresáře nesouvisejí
 - alokace místa podadresářům na různých místech na médiu

Linuxové souborové systémy 1

- ext2/3/4 – nativní linuxový
 - ext3 (2001, jádro 2.4.15) – přidává žurnál
 - ext4 (2008, 2.6.19) – vyšší limity, extenty, ...
- ReiserFS (Hans Reiser)
 - první žurnálovací FS v Linuxu (2001, 2.4.1)
 - efektivní pro mnoho malých souborů v adresáři
- JFS (IBM AIX a OS/2, Linux 2.4.20 2002)
 - JFS1: AIX 3.1 1990, JFS2: OS/2 1999, AIX 5L 2001
 - strom B+ pro rychlé hledání, extenty, komprese

Linuxové souborové systémy 2

- XFS (SGI IRIX 5.3 1993, Linux 2004)
 - patch dostupný od jádra 2.4.2 (2001)
 - efektivní pro paralelní ukládání, škálovatelný
 - strom B+ (var. B*) pro rychlé hledání, extenty, freeze
 - Red Hat: výchozí FS od RHEL 7 (2014)
 - Red Hat nadále investuje do jeho vylepšování
 - žurnálování založené na checkpoints
 - podpora sdílených extentů, svazků (subvolume)
 - copy-on-write – pouze pro data (2018)

Linuxové souborové systémy 3

- Btrfs (vývoj zahájen 2007: IBM, Oracle)
 - verze 1.0 v Linuxu 2.6.29 (2009)
 - „on-disk format“ prohlášen za stabilní až 2014
 - CoW, defragmentace a přidávání zařízení online, vícesvazkový, RAID, snímky, komprese, zálohování
 - plán: šifrování, kontrola online, RAID5, inkrementální zál.
 - má nahradit ext4 (ext4 lze povýšit na Btrfs)
 - RHEL 6.8 (2016) a 7.4 (2017) končí podporu

Linuxové souborové systémy 4

- NILFS (Linux 2.6.13 2005)
 - New Implementation of a Log-structured FS
 - specifický typ CoW vhodný pro SSD a flash
 - continuous checkpoints, lze je připojit (read-only)
 - nízká latence
- F2FS (Samsung 2012, Linux 3.8 2012)
 - log-formát pro NAND flash (SSD, eMMC, SD)
 - vysoká propustnost, defragmentace online, atomické operace, checkpoints

Souborový systém ZFS (1)

- Zettabyte File System, pro Solaris 10, 2004
 - Sun: „*The last word in filesystems.*“
 - obsahuje vrstvu správy oddílů včetně podpory RAID
 - (téměř) neomezená kapacita
 - 256 ZiB – max. velikost zpool, max. 2^{64} zpools
 - 16 EiB – max. velikost souboru
 - až 2^{48} atributů souboru, max. velikost atributu 16 EiB
 - variabilní velikost alokačního bloku
 - téměř žádná nutná správa, automount, online scrub
 - dynamické rozložení zátěže (dynamic stripping)

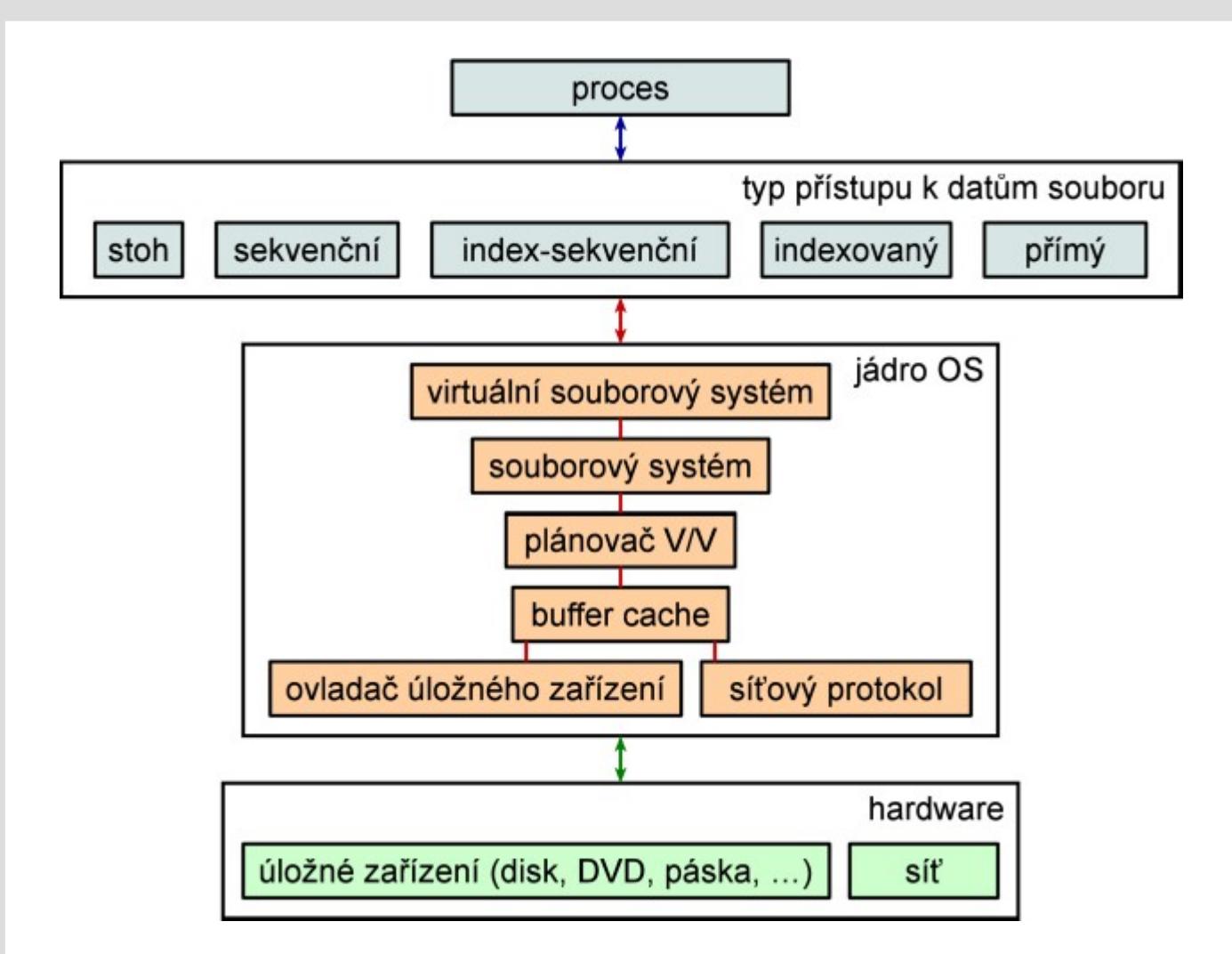
Souborový systém ZFS (2)

- Zettabyte File System
 - prověřená integrita dat – kontrolní součty bloků
 - možnost zapisovat data redundantně + self healing
 - transakční model copy-on-write + cache
 - ZIL (ZFS Intent Log, zápisová cache) lze umístit na SSD
 - ARC (Adaptive Replacement Cache, čtecí cache) v RAM + L2ARC na disku (SSD)
 - snapshots, clones (zapisovatelný snapshot), dump
 - deduplikace dat
 - komprese, šifrování, kvóty, rezervace místa

OpenZFS

- OpenSolaris – otevřená licence CDDL (2005)
 - 2008: ZFS zařazeno do FreeBSD 7.0
 - 2008: začala nativní implementace pro Linux (ZoL)
 - 2010: konec podpory (Oracle), fork OS: illumos
 - 2013: ZFS pro Linux stabilní
 - podporováno např. od Ubuntu 16.04 LTS (2016)
 - verze 0.8 (2019): šifrování, autotrim, projektové kvóty, pool checkpoints, direct IO
 - 2013–2020: sjednocení zdrojů FreeBSD + Linux
 - OpenZFS 2.0 (plán vydání 2020), v 3.0 + macOS (2021)

Architektura správy souborů (obrázek)



Architektura správy souborů (1)

- proces přistupuje k datům dle jejich struktury
- virtuální souborový systém implementuje základní operace se soubory + metadata
 - vytvoření, otevření, zavření, čtení/zápis, atributy
 - může uchovávat data v operační paměti (cache)
- skutečný souborový systém určuje jakým způsobem jsou data s metadaty uložena
 - udržuje řídicí struktury (volné místo, adresáře)
 - eviduje přiřazení logických bloků souborům

Architektura správy souborů (2)

- plánovač základních V/V operací odpovídá za
 - inicializaci a ukončení vstupně-výstupních operací
 - plánování a řazení nevyřízených operací
 - výkonová optimalizace
 - používá část operační paměti jako buffer cache pro urychlení operací – zapisuje data se zpožděním
- ovladač úložného zařízení
 - odpovídá za správné příkazy pro hardware
 - zpracovává hardwarové přerušovací signály vyvolané V/V operacemi

Organizace souborů – terminologie

- pole (field)
 - základní element dat (proměnná)
 - charakterizováno délkou a datovým typem
- záznam (record)
 - seskupení vzájemně souvisejících polí
 - jednotka dat (záznam zaměstnance, výrobku, ...)
- soubor (file)
 - entita – posloupnost podobných záznamů
 - identifikován jménem (a cestou)

Organizace souborů – stoh

- **stoh (pile)**
 - soubor bez nutné vnitřní struktury
 - ukládání záznamů s různými poli
 - přístup k záznamu – úplné prohledání (exhaustive search)
 - příklady:
 - textové soubory – záznamy jsou řádky
 - ploché (flat) soubory – např. /etc/passwd

Organizace souborů – sekvenční soubor

- sekvenční soubor (sequential file)
 - všechny záznamy mají stejnou délku i strukturu všech polí (fixed format)
 - jedno pole je jednoznačným klíčem
 - záznamy mohou být uspořádány dle klíče
 - nové záznamy se ukládají do transakčního logu
 - hlavní soubor se aktualizuje dávkově po seřazení všech záznamů – šetří čas a snižuje riziko ztráty dat

Organizace souborů – index-sekvenční soubor

- **index-sekvenční soubor (indexed sequential file)**
 - sekvenční soubor + index klíčů s ukazatelem na záznam v hlavním souboru
 - úplný (exhaustive) index – nad neseřazeným souborem
 - neúplný index – záznam se dohledává sekvenčně
 - výrazná úspora času oproti sekvenčnímu hledání
 - např. milion záznamů – čistě sekvenční procházení potřebuje průměrně 500 000 přístupů
 - index pro každý 1 000. záznam – průměrně 1 000 přístupů (500 přístupů pro sekvenční hledání v indexu a 500 přístupů v hlavním souboru) ⇒ **zlepšení 500×**

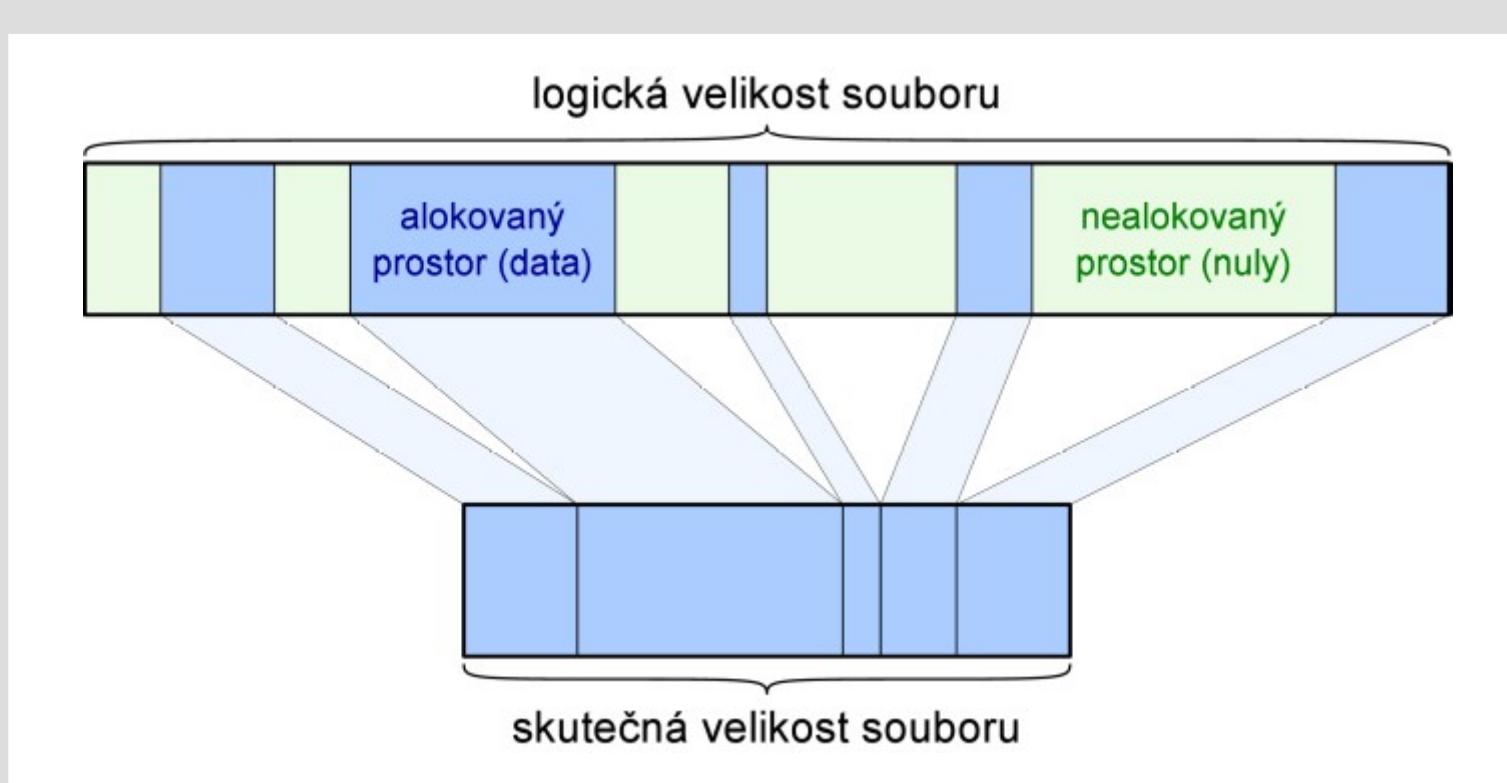
Organizace souborů – indexovaný soubor

- **indexovaný soubor (indexed file)**
 - soubor + indexy pro různá klíčová pole s ukazateli na záznamy v hlavním souboru
 - úplný index klíčových polí, podle kterých není soubor seřazen
 - nad seřazenými záznamy může být i neúplný index
 - konkrétní záznam je nutné (sekvenčně) dohledat

Organizace souborů – přímý soubor

- soubor s přímým přístupem (hashed / direct file)
 - adresa bloku, ve kterém je uložen záznam, je odvozena z klíče pomocí hešovací funkce
 - každý záznam musí mít klíč
 - více různých klíčů může dávat stejnou adresu (**kolize** hešovací funkce) – po zaplnění bloku jsou záznamy ukládány
 - na volná místa v následujících blocích nebo
 - do přeplňovacího souboru (**overflow file**)
 - lze využít tzv. **řídké soubory** (**sparse files**)

Řídký soubor (obrázek)



úseky nul jsou reprezentovány metadaty
nejsou pro ně alokovány žádné bloky

OS – MP, RT a vestavěné systémy

Tomáš Hudec

Tomas.Hudec@upce.cz

<http://fei-as.upceucebny.cz/usr/hudec/vyuka/os/>

Víceprocesorové systémy

- MP (multiprocessor) systems
- systémy s více procesory
 - vícejádrové procesory
 - více procesorů funkčně (stejných či různých) propojených společnou (systémovou) sběrnicí
 - více samostatných systémů s vlastním procesorem propojených společnou sběrnicí nebo pouze sítí

Kategorie počítačových systémů

- **SISD** (single instruction, single data)
 - jeden procesor zpracovává jednu množinu dat jedním proudem instrukcí
- **SIMD** (single instruction, multiple data)
 - jedním proudem instrukcí zpracovává více procesorů více různých množin dat
 - každá instrukce se provede současně v n procesorech
 - každý procesor zpracovává jiná data

Kategorie počítačových systémů

- **MISD** (multiple instruction, single data)
 - více procesorů provádí různé operace nad jednou množinou dat
 - v podstatě nebylo nikdy realizováno
- **MIMD** (multiple instruction, multiple data)
 - více procesorů zpracovává různými proudy instrukcí více různých množin dat

MP systémy podle vazby

- MP systém s volnou vazbou – **loosely coupled**
 - každý procesor má vlastní operační paměť a V/V subsystém
 - různé typy vazby
 - společná sběrnice – Common Bus (např. VME)
 - společné disky – Common Disk
 - nic společného – Common Nothing (vazba LAN)
- MP systém s těsnou vazbou – **tightly coupled**
 - procesory sdílejí operační paměť
 - řízen jedním operačním systémem

MP systémy podle symetrie

- **symetrický víceprocesorový systém – SMP**
 - procesory jsou shodné
 - jádro OS může provádět libovolný procesor
 - procesy i vlákna lze provádět libovolným procesorem
- **asymetrický víceprocesorový systém**
 - procesory jsou funkčně specializované
 - V/V procesory, grafické procesory, FPU apod.
 - systém je řízen centrálním procesorem

Granularita úlohy

- každou úlohu můžeme rozčlenit na úseky, které lze provést samostatně
 - takové úseky lze ve víceprocesorovém systému provádět paralelně na různých procesorech
- pokud jeden úsek potřebuje pro svoji činnost výsledky jiného, musí na tyto výsledky čekat
 - je nutná komunikace a synchronizace
- různé typy úloh se liší velikostí a počtem takových úseků

Granularita a stupeň vazby (1)

- hrubě granulovatelná úloha – méně úseků
 - vhodné jsou kooperující procesy
- jemněji granulovatelná úloha – úseky kratší
 - požadavky na komunikaci a synchronizaci častější
 - vhodnější jsou vlákna – pokud běží na oddělených procesorech, přinášejí výrazné zvýšení výkonnosti
- čím je vazba MP systému volnější, tím větší časové ztráty přináší komunikace a synchronizace

Granularita a stupeň vazby (2)

- pro hrubě granulované úlohy postačí MP systém s volnou vazbou
 - je obvykle levnější
- pro jemně granulované úlohy je nutné použít MP systém s těsnou vazbou
 - jinak by paralelní řešení nebylo efektivní

Paralelismus (1)

- **nezávislý paralelismus**
 - v jednotlivých procesorech běží nezávislé procesy
 - nevyžaduje synchronizaci
 - zkrácení střední doby odezvy pro uživatele
- **velmi hrubý paralelismus**
 - distribuované zpracování rozptýlené do více uzlů sítě představujících jedno výpočetní prostředí
 - počítačové shluky (clusters)
 - vhodný, když interakce mezi procesy nejsou časté
 - přenos zpráv síti zpomalí komunikaci

Paralelismus (2)

- **hrubý paralelismus**
 - jako provádění více procesů na jednom procesoru (multiprocesing), ale rozložené na víc procesorů
- **střední paralelismus**
 - paralelní zpracování nebo multitasking v rámci jedné aplikace
 - jedna aplikace je tvořena více vlákny
 - interakce mezi vlákny jsou obvykle časté

Plánování procesů na MP

- fronta připravených procesů
 - pro všechny procesy jedna (globální)
 - pro každou prioritu samostatná
- všechny fronty plní společnou „zásobárnu“ (pool) procesorů
 - procesu se přiřadí první volný procesor
- složitější plánovací algoritmy se při použití více procesorů obvykle nepoužívají

Plánování vláken na MP

- sdílení zátěže (load sharing)
 - žádný proces není přiřazen k určitému procesoru
- skupinové plánování (gang scheduling)
 - související vlákna jsou plánována tak, aby běžela na různých procesorech současně
- pevné přiřazení procesoru
(dedicated processor assignment)
 - vlákna jsou přiřazena specifickému procesoru
- dynamické plánování (dynamic scheduling)

Sdílení zátěže

- zátěž se rozděluje mezi procesory náhodně
 - procesu je přiřazen libovolný volný procesor
- zajišťuje se, aby žádný procesor nezůstal nevyužitý
 - zátěž se rozděluje rovnoměrně
- není potřebný centralizovaný plánovač
 - důležité u systémů s volnou vazbou
- používá globální fronty

Nevýhody sdílení zátěže

- globální fronta vyžaduje výlučný přístup
 - představuje úzký profil, jestliže o přidělení práce žádá více procesorů najednou
- je nepravděpodobné, že přerušené vlákno bude znova spuštěno ve stejném procesoru
 - snižuje se efektivita použití cache
- jestliže v globální frontě čekají všechna vlákna, nebudou všem vláknům jednoho procesu přiděleny procesory ve stejnou dobu

Skupinové plánování

- všem vláknům tvořícím jeden proces (thread gang) se přidělují procesory současně
- užitečné pro aplikace, jejichž výkonnost by výrazně poklesla, kdyby některá část aplikace neběžela
 - typicky v případech, kdy vlákna vyžadují vzájemnou synchronizaci

Pevné přiřazení procesoru

- když se plánuje spuštění aplikace, jsou všem jejím vláknům napevno přiřazeny procesory
- některé procesory mohou zůstat nevyužity
 - př.: vlákna V1 a V3: CPU 1, vlákna V2 a V4: CPU 2
 - V1 běží, V2 a V4 jsou blokována, V3 je připraveno
 - CPU 2 – nevyužito
- brání přepínání procesů (vláken)
 - př.: V1 a V2 běží, V3 blokuje, V4 je připraveno
 - vláknu V1 vypršelo kvantum, ale nebude přerušeno

Dynamické plánování

- počet vláken procesu se může dynamicky měnit
- OS upravuje zátěž s cílem zlepšit využití systému – obsazuje volné procesory
 - nově příchozím úlohám může být přiřazen procesor obsazený úlohou, která právě používá více než jeden procesor
 - požadavek úlohy trvá, dokud není k dispozici volný procesor
 - nové úlohy dostanou procesor ještě před již existujícími běžícími aplikacemi – snížení odezvy

Reálný čas a počítač

- procesy v počítači jsou reakcí na události v okolí systému nebo mají takové události vyvolat
- události v okolí systému probíhají v reálném čase
 - počítač nemá na tok času vliv
 - procesy s událostmi musí držet krok
 - musí např. dokázat řídit chemickou reakci tak, aby nedošlo k jejímu nechtěnému zastavení nebo naopak k explozi

Správná funkce systémů

- správná funkce systému nezávisí pouze na formální správnosti výpočtů, ale také na tom, kdy jsou výsledky k dispozici!
 - opožděně získané výsledky pro nás ztrácejí význam
 - výsledky výpočtů ztrácejí časem aktuálnost
 - opoždění výsledku nás může i ohrozit

Systémy pracující v reálném čase

- RT (Real-Time) Systems
- pojem obvykle používáme pro oblast technických kybernetických systémů
 - výstupy mohou být závislé na
 - aktuálních hodnotách vstupů (kombinační automat)
 - aktuálních hodnotách a historii vstupů (sekvenční automat, systém s pamětí)
 - oba typy lze realizovat bez počítače, ale druhý typ se dnes obvykle realizuje pomocí počítače
 - systémy s pamětí mohou být samoučící (mohou měnit pravidla chování na základě získaných zkušeností)

Systémy pracující v reálném čase – příklady

- řízení výrobních procesů
- robotika
- řízení letového provozu
- telekomunikační systémy
- řízení laboratorních experimentů
- řízení chemických reakcí

Specifické požadavky RTS

- zpracování dat ve stanoveném časovém limitu
 - zpoždění přenosu můžezpůsobit nestabilitu nebo nefunkčnost systému
- minimalizace rizika selhání systému
 - v oblastech, kde selhání ohrožuje lidské životy, musí systém vyhovovat zvláštním předpisům
 - homologace, dependabilita
- konstrukční a signálová unifikace
 - v systému lze kombinovat produkty různých stran
 - snižuje vývojové, výrobní i provozní náklady

Minimalizace rizika selhání systému

- vysoce spolehlivý a odolný hardware
- redundance prvků a subsystémů (HW i SW)
 - redundantní subsystémy a komunikační cesty
 - záložní řídicí prvky nebo distribuované řízení
 - týká se OS!
- řízená výkonová degradace systému v případě poruchy
 - při snížení výkonnosti v důsledku poruchy plní systém pouze kritické úlohy (mission-critical)

Real Time Operating System

- operační systém pro počítače pracující v reálném čase
- RTOS je charakterizován
 - deterministickým chováním
 - operace se provádějí v pevných předem určených časech nebo v předem určených časových intervalech
 - u každé operace je známo, kdy nejpozději skončí
 - krátkou dobou odezvy
 - vysokou spolehlivostí

Doba odezvy

- čas, ve kterém musí systém přiměřeně reagovat na událost
- časové měřítko je relativní
 - někdy může být sekunda příliš dlouhý čas, jindy na nějaké té sekundě nezáleží
- závisí na aplikační oblasti
 - stovky mikrosekund – např. řízení reaktoru
 - až desítky sekund – např. systém pro rezervaci letenek, jízdenek apod.

Doba odezvy exaktně

- **doba odezvy** – za jak dlouho operační systém reaguje na požadavek přerušení
 - tento čas nesmí překročit předem stanovenou hodnotu
 - skládá se z doby **latence** (interrupt latency)
 - doba mezi okamžikem příchodu požadavku na přerušení a okamžikem, kdy se začne provádět odpovídající obslužný program
 - a doby **obsluhy** přerušení (interrupt processing)
 - doba potřebná k vlastnímu zpracování přerušení

Rozdělení RTS

- obecně neplatí, že v reálném čase znamená velmi rychle
 - **hard real-time**
 - existují absolutní časové limity, při jejichž překročení je odezva zcela bezcenná, systém selže
 - **soft real-time**
 - časové limity jsou pouze přibližné, jejich překročení pouze sníží užitečnost systému
 - **firm real-time**
 - odezva po časovém limitu je bezcenná, nicméně systém může snést několik málo zmeškání

Spolehlivost

- mission critical system
 - porucha může mít katastrofální důsledky
- dependable system
 - systém natolik spolehlivý a bezpečný, že na něm můžeme být zcela závislí
- fault tolerant systém – odolný proti poruchám
 - porucha může snížit výkonnost systému, ale nesmí ho vyřadit z funkce
 - přednost mají úlohy kritické pro funkci systému, úlohy s nižší prioritou se provádějí, jen když na ně zbývá čas

Typické vlastnosti RTOS (1)

- rychlé přepínání kontextu
 - např. s HW podporou – více sad registrů, aby se při přerušení nemusely ukládat registry do RAM
- prioritní plánování, preempce procesů i jádra
 - umožňuje systému rychlou reakci na události
 - typická je architektura mikrojádro
- malé rozměry
 - OS obsahuje jen nejnutnější prvky
 - často používané také jako vestavěný systém

Typické vlastnosti RTOS (2)

- rychlý souborový systém
 - rychlé čtení a ukládání dat snižuje dobu odezvy
- podpora speciálních systémových služeb
 - alarm, timeout apod.
- spolehlivost
- multitasking s komunikací procesů (IPC)
 - spolupracující procesy musí být schopny rychle komunikovat a vzájemně se synchronizovat
 - semafory, signály, fronty zpráv, ...

Specifické požadavky na plánování procesů v RTOS

- některé procesy musí být trvale v oper. paměti
 - odložení na disk – nepřípustné prodloužení odezvy
- práva a priority procesů závisí na jejich účelu
 - procesy důležité pro správné chování a bezpečnost systému musejí mít přednost
- minimalizace intervalů se zákazem přerušení
 - pro řešení kritických oblastí se nepoužívá
- systém musí být plánovatelný

RTOS – příklady

- QNX [kjunix] nebo [kjú en ex]
 - unixový systém založený na mikrojádře
 - určen zejména pro vestavěné systémy
- VxWorks
 - určen zejména pro vestavěné systémy
- RTLinux – modifikace Linuxu s RT-jádrem
- Windows Embedded Compact (Windows CE)
- a další:
 - LynxOS, eCos, ThreadX, RTEMS, OS-9

Vestavěné systémy

- embedded systems [imbedid]
- počítačové systémy, které jsou součástí jiných (obvykle technických) systémů
 - obvykle představují jejich řídicí složku
 - nebo tvoří jejich podsystémy
- obvyklá je schopnost práce v reálném čase

Vestavěné systémy (obrázek)



OS ve vestavěných systémech

- OS je pro uživatele transparentní
 - jeho činnost se uživateli jeví jako funkce podsystému nebo aplikace
 - nevyžaduje zvláštní údržbu
 - nevyžaduje zvláštní znalosti

Podíl vestavěných systémů na trhu s mikroprocesory

- ročně se prodá asi 8 miliard mikroprocesorů a mikropočítačů*
 - z toho jdou jen necelá 2 % do sektoru klasické výpočetní techniky
 - osobní počítače, servery
 - přes 98 % jde do sektoru vestavěných systémů

* Nick Tredennick, Gilder Technology Report, 2004

<http://www.gildertech.com/public/Telecosm2004/Presentations/Wednesday%20PPT/1020-Tredennick.ppt>

Přínos pro uživatele

- obvykle modernizace mechanického nebo elektromechanického systému
 - snížená cena
 - zlepšená funkce
 - zvýšený výkon
 - zvýšená spolehlivost
 - pokud je systém správně navržen a dobře otestován

Nosné aplikáční oblasti (1)

- specializované počítače
 - funkce podobná jako běžné počítače, ale ve specifickém provedení
 - video-hry, přenosné počítače, PDA, ...
- domácnosti
 - domácí spotřebiče, elektronika
 - mikrovlnné trouby, set-top boxy, videorekordéry, kamery, fotoaparáty, audio přehrávače, ...

Nosné aplikáční oblasti (2)

- řídicí systémy
 - zpětnovazební regulace v reálném čase
 - dopravní prostředky, technologické procesy, jaderné reaktory, ...
- zpracování signálu
 - zpracování souvislých proudů dat v reálném čase
 - radar, sonar, video, ...
- telekomunikace a sítě
 - přepínání a směrování přenosu dat
 - pevné a mobilní telefonní sítě, Internet, ...

Typické požadavky (1)

- malá spotřeba
 - bateriové napájení
 - omezená možnost chlazení
- odolnost
 - horko, mráz, vibrace, nárazy, ...
 - kolísání napájení, rušení, blesky
 - vlhkost a zkrápění vodou, koroze
 - nesprávné zacházení

Typické požadavky (2)

- malé rozměry a váha
 - přenosná elektronika
 - dopravní prostředky
 - přebytek váhy znamená vyšší provozní náklady
- reaktivita
 - výpočty probíhají jako odezva na externí události
 - periodické – rotační stroje, zpětnovazební řídicí smyčky, ...
 - aperiodické – tlačítka, ...

Typické požadavky (3)

- funkce v reálném čase
 - správnost je částečně funkcí času
- spolehlivost a bezpečnost
 - musí fungovat správně, ale hlavně nesmí fungovat nepřijatelně!

správný, bezpečný	nesprávný, bezpečný
správný, nebezpečný	nesprávný, nebezpečný
- extrémní cenová citlivost
 - snížení ceny o jednotky až desítky Kč může znamenat zvýšení prodeje o miliony kusů

Typické požadavky (souhrn)

- malá spotřeba
- odolnost
- malé rozměry a váha
- reaktivita
- funkce v reálném čase
- spolehlivost a bezpečnost
- cenová citlivost

OS – Design a bezpečnost

Tomáš Hudec

Tomas.Hudec@upce.cz

<http://fei-as.upceucebny.cz/usr/hudec/vyuka/os/>

Podstata problému designování

- je důležité mít jasný cíl, co přesně požadujeme
 - úspěch jazyka C (Ken Thompson, Dennis Ritchie)
 - jasný cíl: systémové programování
 - vyšší programovací jazyk pro napsání OS (UNIX)
 - neúspěch jazyka PL/I (IBM 1960)
 - potřeba podpory jazyků Fortran a Cobol
 - někteří tvrdili, že Algol je lepší než oba dva
 - IBM sestavila komisi pro navrhnutí jazyka pro všechny:
 - vzít to nejlepší z jazyků Fortran, Cobol a Algol
 - žádná sjednocující myšlenka (vize)

Cíle při návrhu OS

- cíle se liší v závislosti na určení OS
 - vestavěné, serverové, osobní systémy
- hlavní společné cíle při navrhování OS
 - definovat abstrakce
 - poskytnout základní (primitivní) operace
 - zajistit izolaci
 - spravovat hardware

Abstrakce s operacemi

- definování abstrakcí
 - zřejmě nejnáročnější část
 - definování procesů, souborů – poměrně zřejmé
 - další abstrakce: vlákna, synchronizace, signály, paměťový model, V/V, IPC, ...
 - abstrakce jsou obvykle **datové struktury**
- poskytnutí základních (primitivních) operací
 - jednoduché operace nad strukturami (abstrakcemi)
 - **operace** jsou uživateli (programátorovi) přístupné skrz **systémová volání**

Izolace

- zajištění **izolace**
 - uživatelé systému si nesmí zasahovat do svých procesů, systémových prostředků (zdrojů) a dat
 - chyba procesu uživatele neovlivní zbytek systému
 - data se obvykle ukládají do souborů
 - alokované zdroje eviduje struktura procesu
 - **izolují se** tedy **procesy** (jejich alokované zdroje)
 - poskytuje se **řízení ochrany** (přístupová práva)
 - dat, souborů, ale i dalších prostředků a operací nad nimi
 - nabídnout možnost **sdílení**
 - izolace tedy musí být selektivní

Správa HW

- spravování HW
 - čipy pro řadiče přerušení, systémové sběrnice, vstupně-výstupní zařízení
 - ovladače zařízení (device drivers)
 - abstrakce HW

Proč je těžké navrhnut OS?

- pro vývoj HW – Moorův zákon
- nikdo nedefinoval zákon o zlepšování OS
 - i v současnosti existují OS, které jsou v klíčových ohledech (jako spolehlivost) horší než UNIX Version 7 ze 70. let 20. století
 - Proč?
- důvodů je celá řada
 - včetně nedodržení dobrých designových principů
 - OS je od podstaty jiný než aplikace
 - rozsáhlost, složitost, ...

Důvody náročnosti návrhu OS

1. rozsáhlost, komplexnost, složitost
2. konkurence – uživatelé, procesy, prostředky
3. ochrana – nepřátelské prostředí
4. sdílení dat a prostředků
5. životnost
6. obecnost použití
7. přenositelnost
8. zpětná kompatibilita

Principy a vodítka (1)

- jednoduchost
 - Antoine de St. Exupéry: „*Dokonalosti nedosáhneme tehdy, když není nic, co bychom přidali, ale tehdy, když už není, co bychom odebrali.*“
 - méně je více – KISS (Keep It Simple, Stupid)
- úplnost
 - Albert Einstein: „*Vše by mělo být tak jednoduché, jak to jde, ale ne jednodušší.*“
 - OS by měl dělat to, co má, ale nic navíc
 - minimum mechanismů s jasným chováním

Principy a vodítka (2)

- jednoduchost a úplnost
 - každá část, vlastnost, funkce, systémové volání by měla dělat jednu věc a tu dělat dobře a správně
 - přidáváme-li novou vlastnost, je třeba položit si otázku: „Co hrozného by se stalo, když to nepřidáme?“
 - je-li odpověď:
„Nic, ale někdo by to někdy mohl potřebovat.“, pak je lepší tohle zahrnout do knihovny, nikoliv do jádra OS (i když to bude pomalejší)

Principy a vodítka (3)

- jednoduchost × úplnost
 - Tony Hoare: „*There are two ways of constructing a software design:*
 - *One way is to make it so simple that there are obviously no deficiencies, and*
 - *the other way is to make it so complicated that there are no obvious deficiencies.*
 - *The first method is far more difficult.*“

Principy a vodítka – příklady

- MINIX
 - systém je sada procesů se správou paměti, souborovým systémem a ovladači, které běží jako samostatné procesy
 - tři systémová volání (send, receive, sendrec)
 - systém řeší předávání zpráv mezi subsystémy
 - stačí dvě – sendrec je pouhá optimalizace, aby stačil jeden TRAP (skok do jádra)
- Amoeba
 - jediné systémové volání – RPC (Remote Procedure Call) – velmi podobné sendrec ze systému MINIX

Principy a vodítka (4)

- efektivita implementace
 - pokud by měla být vlastnost (systémové volání) neefektivní, pravděpodobně nemá cenu ji mít
 - programátor (uživatel OS) by měl také vědět cenu těchto volání
 - např. posunutí pozice v souboru:
 - lseek – prosté posunutí pozice (změna proměnné)
 - read – provedení V/V operace
 - je-li intuitivní předpoklad ceny volání špatný, budou psát programátoři neefektivní programy

Paradigmata – sladění systému

- architectural coherence
 - architektonická souvislost (promyšlenost)
 - jak jsou vlastnosti systému sladěny v celek
 - pohled zákazníka na systém, zákazníci jsou
 - uživatelé (spouštějí aplikace) – sledují GUI
 - programátoři (píší aplikace) – používají systémová volání
- **paradigma uživatelského rozhraní**
 - top-down design (od GUI k systémovým voláním)
 - bottom-up design (od systémových volání ke GUI)

Paradigmata (různá pojetí)

- **spouštěcí** paradigma – execution paradigm
 - algoritmický přístup
 - provádí se algoritmus, volají se podprogramy, ...
 - OS je pro programátora serverem, poskytuje služby ve formě systémových volání
 - řízení událostmi
 - v cyklu se zpracovávají události a reaguje se na ně
 - vhodné pro interaktivní systémy
- **datové** paradigma
 - jak přistupovat k datům a periferiím
 - např. UNIX: „vše je soubor“

Bezpečnost

- cíle:
 - důvěrnost dat
 - integrita dat
 - dostupnost systému
- příčiny ztráty dat:
 - vyšší moc – živelní pohroma, válka, ...
 - HW či SW chyby – CPU, disky, chyby SW, ...
 - lidský faktor – chybné vstupy, ztráta disku, počítače, ...

Útoky na systém (1)

- útoky zvenčí
- útoky zevnitř
 - od uživatelů, pro které je systém určen!
- způsoby napadení
 - **trojské koně**
 - nevinně vypadající programy, které mají „funkci“ navíc
 - **login spoofing**
 - imitace přihlašovací obrazovky OS
 - **viry, červi**

Útoky na systém (2)

- způsoby napadení
 - zadní vrátka (trap door, back door)
 - vložená metoda přístupu navíc, např. při speciální kombinaci vstupních dat se zaručí další práva
 - logické (časované) bomby
 - po určité době nebo za určitých okolností se spouští škodlivý kód
 - buffer / stack overflow
 - chyba neošetření maximální délky vstupu
 - delší vstup pak přepíše další proměnné nebo dokonce kód a způsobí neočekávané chování programu

Útoky na systém (3)

- způsoby napadení
 - skenování portů (port scanning)
 - zjišťování spuštěných služeb a jejich obslužných programů pro následný pokus o využití známé bezpečnostní slabiny
 - DoS / DDoS – (Distributed) Denial of Service
 - zahlcení serveru množstvím požadavků, takže se služba zastaví nebo alespoň velmi zpomalí
 - distribuovaná verze – požadavky přicházejí z mnoha různých (již napadených) uzlů sítě
 - při podvržení adresy lze dosáhnout blokace části sítě

Metody útoku

- pasivní odposlech dat (neautorizovaný)
- aktivní odposlech
 - změna zprávy nebo její části
 - podvržení zprávy (s platnými adresami)
 - znovuzaslání dříve zachycené zprávy
 - vydávání se za jiný uzel dané sítě

Proslulé útoky na UNIX

- lpr a /etc/passwd
 - tiskový démon umožňoval tisk všem uživatelům a také odstranění souboru
- core a /etc/passwd
 - po pádu procesu systém uloží soubor core (obraz paměti procesu)
- mkdir a pomalý systém
 - vytváření nového i-uzlu jako root

Techniky ochrany

- domény ochrany
 - stanovení domén a práv
- ACL – Access Control List
 - stanovení seznamu práv na objekty pro uživatele
- capability – C-list
 - stanovení práv procesům
 - MAC (Mandatory Access Control)
 - Linux od r. 1998: SELinux (NSA), AppArmor (SuSE)
- firewally a komunikační filtry

Ochrana uživatele

- **autentizace (autentifikace)** – ověření identity
 - jméno a heslo – důležité je užívat správná hesla
 - hesla na jedno použití (tokens)
 - soukromý klíč (např. SSH)
 - použití smart card nebo podobného HW
 - biometrie – otisky prstů, oční sítnice apod.
- **autorizace** – ověření oprávnění

Ochrana dat

- zamezení přístupu (přístupová práva)
 - lze jen v systému
 - je-li systém zcizen nebo se data přenášejí, je třeba použít jinou ochranu
- šifrování
 - symetrická kryptografie
 - asymetrická kryptografie
 - jednosměrné funkce
 - digitální podpisy

Šifrování

- kódování × šifrování
- označení:
 - M = zpráva
 - Ke = šifrovací klíč, Kd = dešifrovací klíč
 - E = šifrovací funkce, D = dešifrovací funkce
 - S = zašifrovaná zpráva, $S = E(M, Ke)$
 - dešifrování: $D(S, Kd) = D(E(M, Ke), Kd) = M$
 - symetrická kryptografie: tajný klíč $Ke = Kd$
 - asymetrická krypt.: veřejný klíč $Ke \neq$ tajný klíč Kd

Šifrování – příklad

- princip zaslání zašifrované zprávy od A k B
 - příjemce B si vygeneruje klíče Ke_B a Kd_B
 - klíč Ke_B příjemce zveřejní, Kd_B si ponechá v tajnosti
 - odesílatel A získá veřejný klíč Ke_B příjemce B
 - A zašle tajnou zprávu $S = E(M, Ke_B)$
 - zpráva lze dešifrovat pouze klíčem Kd_B , takže ji získá pouze příjemce B: $D(S, Kd_B) = M$

Podepisování – příklad

- princip zaslání podepsané zprávy od A k B
 - odesílatel A si vygeneruje klíče Ke_A a Kd_A
 - klíč Ke_A odesílatel zveřejní, Kd_A si ponechá v tajnosti
 - A zašle (veřejnou) zprávu M a připojí k ní podpis $S = D(h(M), Kd_A)$, kde h je vhodná hashovací funkce
 - příjemce B získá veřejný klíč Ke_A odesílatele A
 - B ověří původ zprávy tak, že získá $h' = E(S, Ke_A)$ a následně porovná $h(M)$ s h'
 - útočník nezná Kd_A , tudíž nemůže zfalšovat podpis

Šifrování – podmínky

- pokud máme M i S , pak získat klíče Ke a Kd je výpočetně náročné
- generování párů klíčů Ke a Kd musí být snadné
 - $D(E(M, Ke), Kd) = M$
 - klíče Ke a Kd nesmí být od sebe odvoditelné
- algoritmy E a D musí být efektivní, snadno použitelné a inverzní – $E(D(M, Kd), Ke) = D(E(M, Ke), Kd)$
- máme-li klíč Ke , musí být problém určení odpovídajícího klíče Kd výpočetně náročný

Šifrování – výpočetní náročnost

- Co znamená **výpočetně náročný** problém?
 - řešení nelze snadno spočítat v konečném čase
- příklady
 - problém faktorizace součinu obrovských prvočísel
 - knapsack problem (problém batohu se závažími)
 - další NP-úplné problémy (NP = nedeterministicky polynomiální)
- NP-úplnost – NP je třída obtížnosti
 - úplné nedeterministicky polynomiální problémy jsou **zjednodušeně řečeno** ty nejtěžší problémy

Třídy složitosti

- třída P
 - všechny úlohy lze řešit v polynomiálně omezeném čase na **deterministickém** Turingově stroji
- třída NP
 - úlohy lze řešit v polynomiálně omezeném čase na **nedeterministickém** Turingově stroji
 - jsou to ty problémy, jejichž **řešení lze ověřit v polynomiálním čase**, ovšem nevíme, zda je lze také v polynomiálním čase vyřešit

Turingův stroj

- teoretický model počítače popsaný matematikem Alanem Turingem
 - skládá se z konečného automatu, programu ve tvaru pravidel přechodové funkce a pravostranně nekonečné pásky pro zápis mezivýsledků
- nedeterministický Turingův stroj
 - umožňuje v každém kroku rozvětvit výpočet na n větví, v nichž se posléze řešení hledá současně
 - ekvivalentně se hovoří o stroji, který na místě rozhodování uhodne správnou cestu výpočtu

NP-úplné problémy

- NP-úplné (NP-complete, NPC) problémy jsou
 - takové nedeterministicky polynomiální problémy, na které jsou polynomiálně redukovatelné všechny ostatní problémy z třídy NP
 - třídu NP-úplných úloh tvoří v jistém smyslu ty nejtěžší úlohy z NP
 - pro svou složitost (obtížnost nalezení řešení) se využívají v kryptografii

Problém P versus NP

- v teoretické informatice se takto označuje otázka, zda platí rovnost $P = NP$
- považuje se za nejdůležitější otevřený problém tohoto oboru
 - zařazený mezi sedm tzv. problémů tisíciletí
 - Clay Mathematics Institute vypsal 24. května 2000 **za rozhodnutí vztahu odměnu 1 000 000 dolarů**
 - je zřejmé, že $P \subseteq NP$
 - předpokládá se, že $NP \neq P$

Problém P versus NP (obrázek)



Problém P versus NP

- nalezení **deterministického** polynomiálního algoritmu pro libovolnou NP-úplnou úlohu znamená
 - všechny nedeterministicky polynomiální problémy jsou řešitelné v polynomiálním čase
 - tedy třída NP se „zhroutí“ do třídy P: $P = NP$
 - důsledek: zhroucení dnešní kryptografie

Šifrování – příklad s faktORIZACÍ

- zvolíme prvočísla p, q $p = 3, q = 11$
- vypočítáme $n = p \cdot q$ $n = 3 \cdot 11 = 33$
- vybereme libovolné Kd nesoudělné s $L(n)$:
 - $L(n) = (p - 1)(q - 1)$ $Kd = \text{např. } 13$
 - $\max(p, q) < Kd < L(n)$ $\text{NSD}(13, 20) = 1$
- vypočítáme Ke :
 - $0 < Ke < L(n)$ $L(n) = 2 \cdot 10 = 20$
 - $Ke \cdot Kd = 1 \pmod{L(n)}$ $11 < Ke < 20$

$$Ke = 17$$

$$0 < Ke < 20$$

$$17 \cdot 13 = 1 \pmod{20}$$

Šifrování – příklad s faktORIZACÍ

- nyní můžeme šifrovat hodnoty 0 až $(n - 1)$
 - šifrování znaku zprávy M_i : $S_i = M_i^{Ke} \pmod{n}$
 - dešifrování znaku zprávy S_i : $M_i = S_i^{Kd} \pmod{n}$
- Ke a n jsou známé
- Kd lze těžko odvodit, protože neznáme p a q
- např. šifrujeme „ZDAR“ ($A = 1, B = 2, \dots$)
 - $E("26\ 04\ 01\ 18", 17, 33) = "05\ 16\ 01\ 06" = \text{„EPAD“}$
 - $D("05\ 16\ 01\ 06", 13, 33) = "26\ 04\ 01\ 18" = \text{„ZDAR“}$