Základy operačních systémů

I.

KIV/ZOS 2013

Kontaktní informace

- Ing. Ladislav Pešička
- UL401
- *pesicka@kiv.zcu.cz*
  - *Předmět zprávy začít: ZOS*
- Úřední hodiny
  - St 10:00 až 11:00
  - Pa 10:00 až 11:00
- Veškeré informace v coursewaru

Požadavky na zápočet

- 2 zápočtové testy
  - pass / failed
  - 1 náhradní termín
- Semestrální práce
  - program + dokumentace

1. zápočtový test

- časově koncem října / začátek listopadu
- v době cvičení po ½
- napsat složitější script v /bin/bash
- a teoretická otázka
- Např:

```
skript –p1 s1.txt
skript –p2
skript –p3 > vys.txt
```

2. zápočtový test

- teoretický
- časově začátek prosince
- otázky z přednášek
- řešení příkladů podobných těm na cvičení

ZOS cvičení

- Základy Linuxu
  - distribuce, jádro, struktura
  - uživatelské ovládání
  - příkazy, spojování příkazů
  - příkazové skripty
- Paralelní procesy, souběhy a ošetření
  - ošetření kritické sekce, uvíznutí, ...
  - reálná implementace Java, C, ..
- Témata z přednášek

Zkouška

- Písemný test
  - Test na 60 min. bez pomůcek
  - zvolit správnou odpověď, odpovědět na otázku, doplnit či nakreslit diagram atd..
  - Ústní pohovor nad písemkou

ZOS

- Obecné principy OS
  - Není zaměřen na 1 systém, vychází z Unixu
  - Není hodnocením, který systém je lepší
- KIV/OS, KIV/PPR
  - Pokračováním, Unix / Linux, paralelizace
- Praxe
  - Základy práce s Linuxem
  - Práce se sdílenými zdroji, ošetření kritické sekce

ZOS přednášky = struktura OS (!)

- modul pro správu procesů
  - program, proces, vlákno, plánování procesů a vláken
  - kritická sekce, synchronizace (semafory, …)
  - deadlock, vyhladovění
- modul pro správu paměti
  - virtuální paměť: stránkování, segmentace
- modul pro správu I/O
- modul pro správu souborů
- síťování
- bezpečnost

- Kde všude můžete nalézt OS?
- Ukázky zařízení

(s využitím materiálu
Introduction to embedded systems)

zubní kartáček

CPU: 8-bit

• řízení rychlosti

• časovač

• nabíjení

OS? NE

Prodejní terminál

Point-of-Sale (POS) Terminal

Microprocessor:
Intel X86 Celeron

OS: Windows XP Embedded

Kuka robot arms welding a Mercedes

Svařovací robot

Microprocessor: X86

OS: Windows CE OS & Others

OS v běžném životě - mobily

- iOS 7.0
  - Apple iPhone, iPad
  - Výrazná změna vzhledu
  - Většina zařízení poslední verze
- Android
  - Poslední verze Android 4.3
  - Na Linuxovém jádru
  - Roztříštěnost verzí mezi uživateli
- Windows Phone 8

Zdroj obrázku: developer.android.com

OS ve vesmíru

NASA's Twin Mars Rovers.

Microprocessor:
Radiation Hardened
20Mhz PowerPC

Commercial Real-time OS

Software and OS was developed during multi-year flight to Mars
and downloaded using a radio link

OS – příklady použití

- Servery, pracovní stanice, notebooky
  - MS Windows, GNU/Linux, Solaris
- Mobilní zařízení, tablety
  - Windows CE, Symbian, Linux, Android, …
- Routery, AP, SOHO síťová zařízení
  - Cisco IOS, Linux, VxWorks
- Embedded zařízení
  - Bankomaty, stravovací systémy, lékařské přístroje
  - Windows CE, Windows XP embedded, Linux

Co všechno tvoří OS?

- Není všeobecná definice
- Vše co dodavatel poskytuje jako OS ?
  - Windows – kalkulačka, hra miny, malování, …
- Program, běžící po celou dobu běhu systému ?
  - Ale Linux, moduly, zavádění na žádost v případě potřeby
- SLOC (Source lines of code)
  - Windows XP: 40 milionů řádků
  - Linux kernel 3.10 16,9 mil. ř.
  - Distribuce Debian 4.0 283 mil. ř.

Operační Systém - definice

OS je softwarová vrstva (základní programové vybavení), jejíž úlohou je spravovat hardware a poskytovat k němu programům jednotné rozhraní

- OS zprostředkovává aplikacím přístup k hardwaru
- OS koordinuje a poskytuje služby aplikacím
  - Analogie – dopravní systém, vláda, ..
- OS je program, který slouží jako prostředník mezi aplikacemi a hardwarem počítače.

Privilegovaný a uživatelský režim

- Jádro OS běží v tzv. privilegovaném režimu
  - Všechny instrukce CPU povoleny
  - *Privileg. režim není v MS DOS, různé embedded systémy*
  - *Někdy část OS v uživatelském režimu*
  - *Interpretované systémy (JVM)*
- Aplikace – v uživatelském režimu
  - Některé instrukce zakázány
    např. přímý přístup k disku, jeho zformátování zákeřnou aplikací
  - Aplikace musí požádat OS o přístup k souboru, ten rozhodne zda jej povolí
- OS může zasahovat do běhu aplikací
- Aplikace může požádat OS o službu

aplikace nemá přímý přístup k HW

OS

Dva základní pohledy na OS:

- **Rozšířený stroj** (shora dolů)
- **Správce zdrojů** (zdola nahoru)

OS jako rozšířený stroj

- Holý počítač
  - Primitivní a obtížně programovatelný (I/O)
  - Např. disky …
    - Práce s hlavičkou disku
    - Alokace dealokace bloků dat
    - Víc programů chce sdílet stejné médium
- Jako programátor chceme
  - Jednoduchý pohled – pojmenované soubory
  - OS skrývá před aplikacemi podrobnosti o HW (přerušení, správu paměti..)

OS jako rozšířený stroj

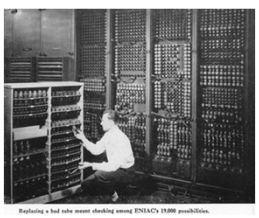
- Strojové instrukce (holý stroj)
- Vysokoúrovňové služby (rozšířené instrukce)
  - Systémová volání
- Z pohledu programátora
  - Pojmenované soubory
  - Neomezená paměť
  - Transparentní I/O operace
- ZOS zkoumá, jaké služby a jak jsou v OS implementovány

OS jako správce zdrojů

- OS jako poskytovatel / správce zdrojů (resource manager)
- Různé zdroje (čas CPU, paměť, I/O zařízení)
- OS – správná a řízená alokace zdrojů procesům, které je požadují (přístupová práva)
- Konfliktní požadavky na zdroje
  - V jakém pořadí vyřízeny
  - Efektivnost, spravedlivost

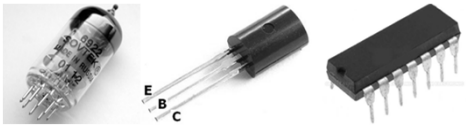
Historický vývoj

- Vývoj hw -> vývoj OS
- 1. počítač – ENIAC, 15.2.1946
  - Tělocvična
  - 18 000 elektronek
  - Regály, chlazení
  - 5000 operací/s



Generace počítačů

1. Elektronky
2. Tranzistory
3. Integrované obvody
4. LSI, VLSI (mikroprocesory,..)



1.Generace (1945-55)

- Elektronky, propojovací desky
- Programování
  - V absolutním jazyce
  - Propojování zdířek na desce
  - Později děrné štítky, assemblery, knihovny, FORTRAN
  - Numerické kalkulace
- Způsob práce
  - **Stejní lidé** – stroj navrhli, postavili, programovali !
  - Zatrhnout blok času na rozvrhu, doufat, že to vyjde
- OS ještě neexistují

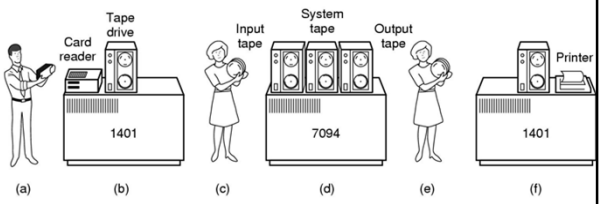
2. Generace (1955-65)

- Tranzistory, dávkové OS
- Vyšší spolehlivost; klimatizované sály
- Oddělení návrhářů, výroby, operátorů, programátorů, údržby
- Mil $ - velké firmy, vlády, univerzity
- Způsob práce
  - Vyděrovat štítky s programem
  - Krabici dát operátorovi
  - Výsledek vytisknut na tiskárně
- Optimalizace
  - Na levném stroji štítky přenést na magnetickou pásku

2. generace

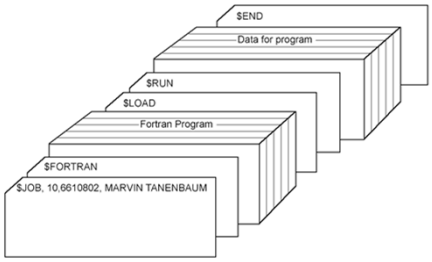
- Sekvenční vykonávání dávek
- Ochrana systému – kdokoliv dokázal shodit
- OS IBSYS = IBM SYSTÉM FOR 7094
- Pokud úloha prováděla I/O, CPU čekal..
  - Čas CPU je drahý
- Viz slidy Tanenbaum

Dávkové systémy



– vezmi štítky k 1401
– štítky se zkopírují na pásek (tape)
– pásek na 7094, provádí výpočet
– output tape na 1401 vytiskne výstup

History of Operating Systems



- Struktura typické dávky – 2nd generation

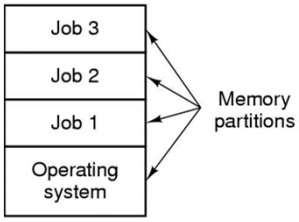
3. Generace (1965-80)

- Integrované obvody, multiprogramování
- Do té doby 2 řady počítačů
  - Vědecké výpočty
  - Komerční stroje – banky, pojišťovny
- IBM 360 – sjednocení
  - Malé i velké stroje
  - Komplexnost – spousta chyb

3. generace

- Multiprogramování
  - Doba čekání na I/O neefektivní (věda OK, banky 80-90% čekání)
  - Více úloh v paměti
    - Napřed konstantní počet
    - HW pro ochranu paměti
- Každá úloha ve vlastní oblasti paměti; zatímco jedna provádí I/O, druhá počítá …

History of Operating Systems



- Multiprogramming system
  - three jobs in memory – 3rd generation

3. generace

- Spooling
  - Na vstupu – ze štítků na disk, úloha se zavede z disku
  - Na výstupu – výsledky na disk před výtiskem na tiskárně

  spooling se dnes používá typicky pro sdílení tiskárny mezi uživateli
  uživatel svůj požadavek vloží do tiskové fronty
  až je tiskárna volná, speciální proces vezme požadavek z fronty a vytiskne jej
- Stále dávkové systémy
  - Dodání úlohy, výsledek – několik hodin

3. generace

- Systémy se sdílením času (time shared system)
  - Varianta multiprogramování
  - CPU střídavě vykonává úlohy
  - Každý uživatel má on-line terminál
- CTSS (MIT 1962) *Compatible Time Sharing Sys.*
- MULTICS

Minipočítače

- DEC PDP (1961)
  - Cca 3.5 mil Kč , „jako housky“
  - Až PDP11 – nekompatibilní navzájem
- Výzkumník Bell Labs pracující na MULTICSu Ken Thompson – našel nepoužívanou PDP-7, napsal omezenou jednouživat. verzi MULTICSu vznik UNIXu a jazyka C (1969)

4. Generace (1980)

- Mikroprocesory, PC
- GUI x CLI
- Síťové a distribuované systémy
- MS DOS, Unix, Windows NT
- UNIX – dominantní na nonIntel;
- Linux, BSD – rozšíření i na PC
  - Výzkum Xerox PARC – vznik GUI
  - Apple Macintosh
- film „Piráti ze Silicon Valley“

Dělení OS

- Dle úrovně sdílení CPU
- Jednoprocesový
  - MS DOS, v daném čase v paměti aktivní 1 program
- Multiprocesový
  - Efektivnost využití zdrojů
  - Práce více uživatelů

Dělení OS

- Dle typu interakce
- Dávkový systém
  - Sekvenční dávky, není interakce
  - i dnes má smysl, viz. meta.cesnet.cz
- Interaktivní
  - Interakce uživatel – úloha
  - Víceprocesové – interakce max. do několika sekund (Win, Linux, ..)

OS reálného času (!)

- Výsledek má smysl, pouze pokud je získán v nějakém omezeném čase
- Přísné požadavky aplikací na čas odpovědi
  - Řídící počítače, multimedia
- Časově ohraničené požadavky na odpověď
  - Řízení válcovny plechu, výtahu mrakodrapu ☺
- Nejlepší snaha systému
  - Multimedia, virtuální realita
- Př: RTLinux, RTX Windows, VxWorks

Hard realtime OS

- Zaručena odezva v **ohraničeném** čase
- Všechna zpoždění a režie systému ohraničeny

Omezení na OS:

- Často není systém souborů
- Není virtuální paměť
- Nelze zároveň sdílení času
- Řízení výroby, robotika, telekomunikace

Soft realtime OS

- Priorita RT úloh před ostatními
- Nezaručuje odezvu v daném čase
- Lze v systémech sdílení času
- RT Linux
- Multimédia, virtuální realita

Další dělení OS

- Dle velikosti HW
  - Superpočítač, telefon, čipová karta
- Míra distribuovanosti
  - Klasické - centralizované 1 a více CPU
  - Paralelní
  - Síťové
  - Distribuované
    - virtuální uniprocesor
    - Uživatel neví kde běží programy, kde jsou soubory

Další dělení OS

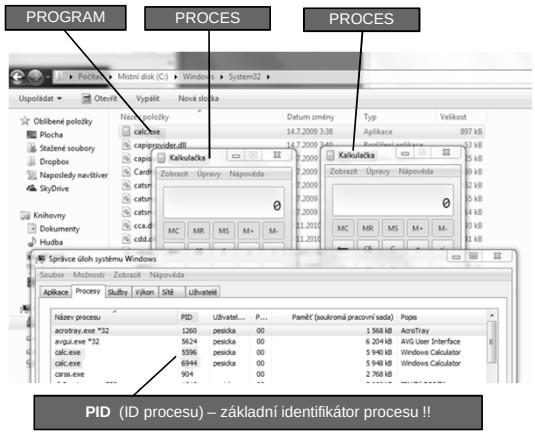
- Podle počtu uživatelů
  - Jedno a víceuživatelské
- Podle funkcí
  - Univerzální
  - Specializované (např. Cisco IOS)

Základní funkce operačního systému (!)

- správa procesů
- správa paměti
- správa souborů
- správa zařízení - I/O subsystém
- síťování (networking)
- ochrana a bezpečnost
- uživatelské rozhraní

Správa procesů

- Program
  - Spustitelný kód, v binární podobě
  - Nejčastěji uložený na disku
  - Např. C:\windows\system32\calc.exe
- proces – instance běžícího programu
  - Přidělen čas CPU
  - Potřebuje paměťový prostor
  - vstupy a výstupy
  - *Dle jednoho programu můžeme spustit více procesů*



Správa paměti

- správa hlavní paměti
  - Přidělování paměti procesům
    - alokace / dealokace paměti dle potřeby
    - Virtuální adresování (stránkování, segmentace)
  - Správa informace o volné a obsazené paměti
    - Která část paměti je volná, která obsazená a kým
  - Odebírání paměti skončenému procesu
  - Ochrana paměti
    - Přístup pouze pro oprávněné procesy

Soubory

- soubory
  - vytváření a rušení souborů
  - vytváření a rušení adresářů
  - primitiva pro manipulaci
    - se soubory
    - s adresáři
  - správa volného prostoru vnější paměti
  - mapování souborů na vnější paměť
  - rozvrhování diskových operací

I/O subsystém

- I/O subsystém
  - správa paměti pro buffering, caching, spooling
  - společné rozhraní ovladačů zařízení
  - ovladače pro specifická zařízení

  ovladače – kámen úrazu každého OS

Ochrana a bezpečnost

- ochrana a bezpečnost
  - ke zdrojům smí přistupovat pouze autorizované procesy

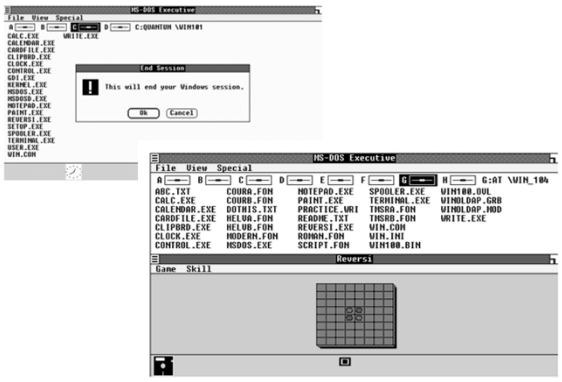
  - specifikace přístupu
  - mechanismus ochrany (souborů, paměti)

  ACL, capabilities, ...

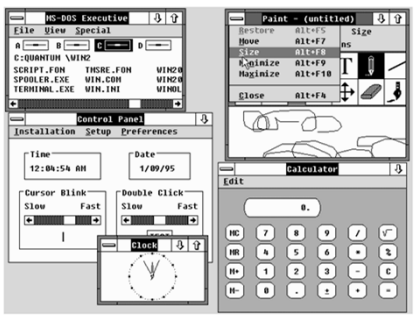
Uživatelské rozhraní

- uživatelské rozhraní

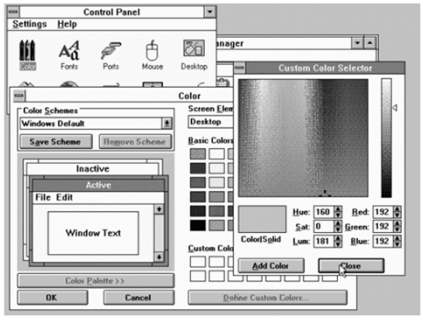
- CLI (command line interface)
- GUI (graphical user interface)
  - *ukázky z www.zive.cz*



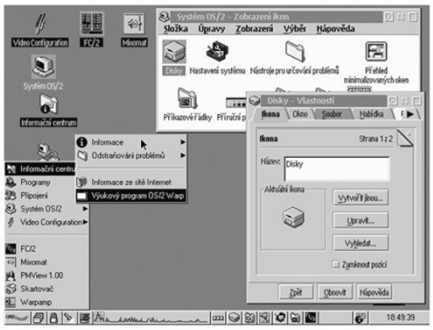
Win 2.0



Win 3.1



OS/2 Warp4



2 základní režimy OS

- **Uživatelský režim**
  - V tomto režimu běží aplikace (word, kalkulačka,..)
  - Nemůžou vykonávat všechny instrukce, např. přímý přístup k zařízení (tj. zapiš datový blok *x* na disk *y*)
    - Proč? Jinak by škodlivá aplikace mohla např. smazat disk
    - Jak se tomu zabrání? Aplikace musí požádat jádro o službu, jádro ověří, zda aplikace má na podobnou činnost oprávnění a jádro činnost provede
- **Privilegovaný režim (režim jádra)**
  - Zde jsou povoleny všechny instrukce procesoru
  - Běží v něm jádro OS, které mj. vykonává služby (systémová volání), o které je aplikace požádá

Jak se dostat z uživatelského režimu do režimu jádra?

Jde o přepnutí „mezi dvěma světy“, v každém z nich platí jiná pravidla

- Softwarové přerušení – instrukce INT 0x80
  - Stejně jako při hardwarovém přerušení (např. stisk klávesy): začne se vykonávat kód přerušení a vykoná se příslušné systémové volání
- Speciální instrukce (sysenter)
  - Speciální instrukce mikroprocesoru

Systémové volání

- Pojem **systémové volání** znamená vyvolání služby operačního systému, kterou by naše uživatelská aplikace nemohla sama vykonat, např. již zmíněný přístup k souboru na disku.
- Aplikace může volat systémové volání přímo (*open, creat*), nebo prostřednictvím knihovní funkce (v C např. fopen), která následně požádá o systémové volání sama.
- Výhodou knihovní funkce je, že je na různých platformách stejná, ať už se vyvolání systémové služby děje různým způsobem na různých platformách.

Systémové volání – příklad (!)

1. Do vybraného registru (EAX) uložím číslo služby, kterou chci vyvolat
   - Je to podobné klasickému číselníku
   - Např. služba 1- vytvoření procesu, 2- otevření souboru, 3- zápis do souboru, 4- čtení ze souboru, 5- výpis řetězce na obrazovku atd.
2. Do dalších registrů uložím další potřebné parametry
   - Např. kde je jméno souboru který chci otevřít
   - Nebo kde začíná řetězec, který chci vypsat
3. Provedu instrukci, která mě přepne do světa jádra
   - tedy INT 0x80 nebo sysenter
4. V režimu jádra se zpracovává požadovaná služba
   - Může se stát, že se aplikace zablokuje, např. čekání na klávesu
5. Návrat, uživatelský proces pokračuje dále

Příklad

Jen ideový, v reálném systému se příslušné registry mohou jmenovat jinak

služba

```
LD AX, 5      .. Budeme volat službu 5 (tisk řetězce)
LD BX, 100    .. Od adresy 100 bude uložený řetězec
LD CX, 10     .. Délka řetězce, co se bude tisknout
INT 0x80      .. Vyvolání sw přerušení, přechod do světa jádra
…             .. Vykonání systémového volání
…             .. Kód naší aplikace pokračuje dále
```

parametry

Příklad reálný – Linux - getpid.c

- do registru EAX dáme číslo služby 20
- systémové volání přes int 0x80
- v registru EAX máme návratovou hodnotu pro naši službu (getpid)

```
int pid;

int main() {

    __asm__(
        "movl $20, %eax  \n"    /* getpid system call */
        "int  $0x80      \n"    /* syscall */
        "movl %eax, pid  \n"    /* get result */
        );
  printf("Test volani systemove sluzby...\nPID: %d\n", pid);


 return 0;
}
```

Jak zjistím číslo služby?

např.
http://www.cli.di.unipi.it/~gadducci/SOL-11/Local/referenceCards/LINUX_System_Call_Quick_Reference.pdf
(kolik různých volání jste napočítali?)

Možnosti programátora

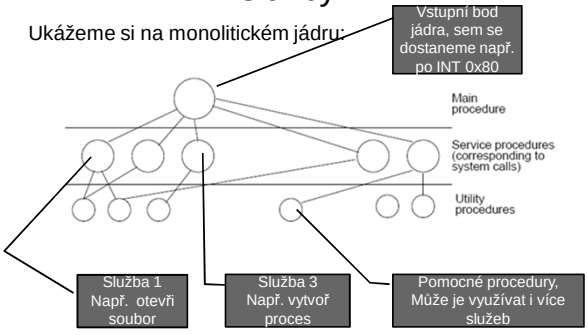
- inline assembler a INT 0x80
  viz předchozí slide
- použití instrukce syscall()
  id1 = syscall (SYS_getpid);
- přímo volání getpid() .. wrapper v libc knihovně
  id2 = getpid();

Další možnosti uložení parametrů

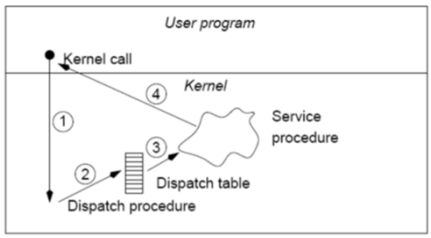
- Musím nějak jádru říci, kterou službu chci a další parametry
- Informaci můžeme uložit
  - Do registrů
  - Na zásobník
  - Na předem danou adresu v paměti
  - Kombinací uvedených principů
- Příklad informace:
  chci po OS službu 2 (natočení piva) číslo služby uložím do AX, do registru BX uložím velikost piva (malé), do registru CX stupeň (desítka)…
- *Jádro ví, že když je zavolána služba 2, tak jak má interpretovat obsah registrů BX a CX*

Jak jádro implementuje jednotlivé služby?

Ukážeme si na monolitickém jádru:



Vyvolání služby systému (opakování)



Vyvolání služby systému (opakování)

- Parametry uložíme na určené místo
  - registry, zásobník
- Provedeme speciální instrukci (1)
  - vyvolá obsluhu v jádře
  - přepne do privilegovaného režimu
- OS převezme parametry, zjistí, která služba je vyvolána a provede službu
- návrat zpět
  - Přepnutí do uživatelského režimu

Poznámka

- Systémové volání nevyžaduje přepnutí kontextu na jiný proces
- Je zpracováno v kontextu procesu, který jej vyvolal

Co znamená INT x?

- instrukce v assembleru pro x86 procesory, která generuje sw přerušení
- x je v rozsahu 0 až 255
- paměť od 0 do je 256 4bytových ukazatelů (celkem 1KB), obsahují adresu pro obsluhu přerušení – vektor přerušení
- HW interrupty jsou mapovány na dané vektory prostřednictvím programovatelného řadiče přerušení

INT 0x80

- v 16kové soustavě 80, dekadicky 128
- pro vykonání systémového volání
- do registru EAX se dá číslo systémového volání, které chceme vyvolat

Jak se aplikace dostane do režimu jádra? (opakování)

- Softwarové přerušení
  - Volající proces způsobí softwarové přerušení
  - Na platformě x86: instrukce int 0x80
  - Přerušení se začne obsluhovat, procesor se přepne do režimu jádra a začne se provádět kód jádra
- Speciální instrukce
  - Novější, rychlejší
  - Platforma x86: instrukce sysenter, sysexit

Může se lišit na různých platformách

Přerušení (interrupt)

- Přerušení patří k základním mechanismům používaným v OS
- **Asynchronní** obsluha události, procesor **přeruší** vykonávání sledu instrukcí (části kódu, které se právě věnuje), **vykoná** obsluhu přerušení (tj. instrukce v obslužné rutině přerušení) a **pokračuje** předchozí činností
- Analogie:
  - vařím oběd (vykonávám instrukce běžného procesu),
  - zazvoní telefon (přijde přerušení, je to asynchronní událost – kdykoliv)
  - Vyřídím telefon (obsluha přerušení)
  - Pokračuji ve vaření oběda (návrat k předchozí činnosti)
- Některé systémy mají víceúrovňová přerušení (vnoření)
  - (telefon přebije volání, že na někoho v sousedním pokoji spadla skříň)

Druhy přerušení (!!)

- **Hardwarové přerušení (vnější)**
  - Přichází z I/O zařízení, např. stisknutí klávesy na klávesnici
  - **Asynchronní** událost – uživatel stiskne klávesu, kdy se mu zachce
  - Vyžádá si pozornost procesoru bez ohledu na právě zpracovávanou úlohu
  - Doručovány prostřednictvím **řadiče přerušení** (umí stanovit **prioritu** přerušením,aj.)
- **Vnitřní přerušení**
  - Vyvolá je sám procesor
  - Např. pokus o dělení nulou, výpadek stránky paměti (!!)
- **Softwarové přerušení**
  - Speciální strojová instrukce (např. zmiňovaný příklad INT 0x80)
  - Je **synchronní**, vyvolané záměrně programem (chce službu OS) volání služeb operačního systému z běžícího procesu (!!) uživatelská úloha nemůže sama skočit do prostoru jádra OS, ale má právě k tomu softwarové přerušení
- Doporučuji přečíst: *http://cs.wikipedia.org/wiki/P%C5%99eru%C5%A1en%C3%AD*

Kdy v OS použiji přerušení? (to samé z jiného úhlu pohledu)

- **Systémové volání** (volání služby OS)
  - Využiji softwarového přerušení a instrukce INT
- **Výpadek stránky paměti**
  - V logickém adresním prostoru procesu se odkazuji na stránku, která není namapovaná do paměti RAM (rámec), ale je odložená na disku
  - Dojde k přerušení – výpadek stránky
    - Běžící proces se pozastaví
    - Ošetří se přerušení – z disku se stránka natáhne do paměti (když je operační paměť plná, tak nějaký rámec vyhodíme dle nám známých algoritmů ☺)
    - Pokračuje původní proces přístupem nyní už do paměti RAM
- **Obsluha HW zařízení**
  - Zařízení si **žádá pozornost**
  - Klávesnice: stisknuta klávesa
  - Zvukovka : potřebuji poslat další data k přehrání
  - Síťová karta: došel paket

Vektor přerušení

- I/O zařízení signalizuje přerušení *(něco potřebuji)*
- Přerušení přijde na nějaké lince přerušení (IRQ, můžeme si představit jeden drát ke klávesnici, jiný drát k sériovému portu, další k časovači atd.)
- Víme číslo drátu (např. IRQ 1), ale potřebujeme vědět, na jaké adrese začíná obslužný program přerušení
- Kdo to ví? … vektor přerušení
- **Vektor přerušení** je vlastně index do pole, obsahující adresu obslužné rutiny, vykonané při daném typu přerušení

Poznámka k přerušením

„signál“ operačnímu systému, že nastala nějaká událost, která vyžaduje ošetření (vykonání určitého kódu) - asynchronní

- Hardwarové přerušení
  - původ v HW
  - Stisk klávesy, pohnutí myši
  - Časovač (timer)
  - Disk, síťová karta, ztráta napájení,..
- Softwarová přerušení
  - Pochází ze SW
  - Obvykle z procesu v uživatelském režimu

Poznámka k přerušením

Příchod přerušení, z tabulky přerušení pozná, kde leží obslužný kód pro dané přerušení

Pozn. pro sw přerušení 0x80 ukazuje v tabulce přerušení (vektor přerušení) na vstupní bod OS

Maskování přerušení – v době obsluhy přerušení (musí být rychlá) lze zamaskovat méně důležitá přerušení

Sw přerušení jsou nemaskovatelná

Přijde-li přerušení… (!!)

- Přijde signalizace přerušení
- Dokončena rozpracovaná strojová instrukce
- Na zásobník uložena adresa následující instrukce, tj. kde jsme skončili a kde budeme chtít pokračovat !!!!!!!!
- Z vektoru přerušení zjistí adresu podprogramu pro obsluhu přerušení
- Obsluha - rychlá
  - Na konci stejný stav procesoru jako na začátku
- Instrukce návratu RET, IRET
  - Vyzvedne ze zásobníku návratovou adresu a na ní pokračuje !!!
- Přerušená úloha (mimo zpoždění) nepozná, že proběhla obsluha přerušení

Knihovní funkce

- mechanismy volání jádra se v různých OS liší
- knihovní funkce
  - programovací jazyky zakrývají služby systému, aby se jevily jako běžné knihovní funkce
  - Jazyk C: printf("Retezec");
    - Knihovní funkce se sama postará, aby vyvolala vhodnou službu OS na dané platformě pro výpis řetězce
- Ne vždy musí volání knihovní funkce znamenat systémové volání, např. matematické funkce

Architektury OS

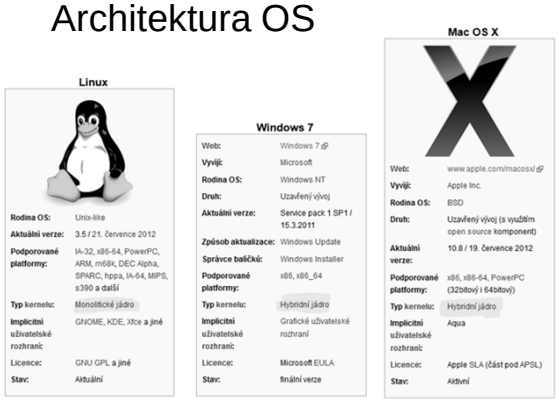
OS = jádro + systémové nástroje

Jádro se zavádí do operační paměti při startu a zůstává v činnosti po celou dobu běhu systému

Základní rozdělení:

- Monolitické jádro – jádro je jeden funkční celek
- Mikrojádro – malé jádro, oddělitelné části pracují jako samostatné procesy v user space
- Hybridní jádro - kombinace

Architektura OS



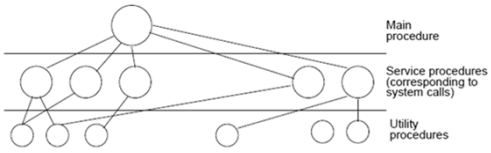
Monolitické jádro

- Jeden spustitelný soubor
- Uvnitř moduly pro jednotlivé funkce
- Jeden program, řízení se předává voláním podprogramů
- Příklady: UNIX, Linux, MS DOS

Typickou součástí jádra je např. souborový systém

Linux je monolitické jádro OS, s podporou zavádění modulů za běhu systému

Monolitické jádro (!)

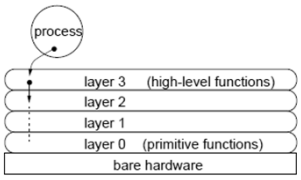


Main procedure – vstupní bod jádra, na základě čísla služby zavolá servisní proceduru
Service procedure – odpovídá jednotlivým systémovým voláním (zobrazení řetězce, čtení ze souboru, aj.)
Service procedure volá pro splnění svých cílů různé pomocné utility procedures (lze je opakovaně využít v různých voláních)

Vrstvené jádro

- Výstavba systému od nejnižších vrstev
- Vyšší vrstvy využívají primitiv poskytovaných nižšími vrstvami
- Hierarchie procesů
  - Nejníže vrstvy komunikující s HW
  - Každá vyšší úroveň poskytuje abstraktnější virtuální stroj
  - Může být s HW podporou – pak nelze vrstvy obcházet (obdoba systémového volání)
- Příklady: THE, MULTICS

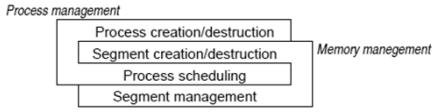
Vrstvené jádro



OS THE:
Úroveň 0 .. Virtualizace CPU (přepínání mezi procesy)
vyšší vrstva už předpokládá existenci procesů
Úroveň 1 .. Virtualizace paměti
vyšší vrstvy už nemusí řešit umístění části procesů v paměti

Funkční hierarchie

- Problém jak rozčlenit do vrstev
  - „dřív slepice, nebo vejce?“
  - správa procesů vs. správa paměti
- Některé moduly vykonávají více funkcí, mohou být na více úrovních v hierarchii
- Př: Pilot



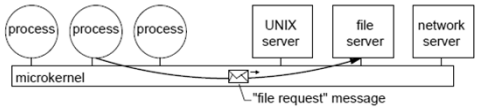
Mikrojádro (!)

- Model klient – server
- Většinu činností OS vykonávají samostatné procesy mimo jádro (servery, např. systém souborů)
- Mikrojádro
  - Poskytuje pouze nejdůležitější nízkoúrovňové funkce
    - Nízkoúrovňová správa procesů
    - Adresový prostor, komunikace mezi adresovými prostory
    - Někdy obsluha přerušení, vstupy/výstupy
  - Pouze mikrojádro běží v privilegovaném režimu
    - Méně pádů systému

Mikrojádro

- Výhody
  - vynucuje modulární strukturu
  - Snadnější tvorba distribuovaných OS (komunikace přes síť)
- Nevýhody
  - Složitější návrh systému
  - Režie
    (4 x přepnutí uživatelský režim <–> jádro)
- Příklady: QNX, Hurd, OSF/1, MINIX, Amoeba

Mikrojádro

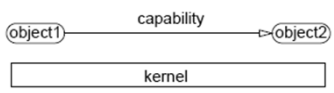


Mikrojádro – základní služby, běží v privilegovaném režimu
1. proces vyžaduje službu
2. mikrojádro předá požadavek příslušnému serveru
3. server vykoná požadavek

Snadná vyměnitelnost serveru za jiný
Chyba serveru nemusí být fatální pro celý operační systém (není v jádře)
Server může event. běžet na jiném uzlu sítě (distribuov. syst.)

Objektově orientovaná struktura

- Systém je množina objektů (soubory, HW zařízení)
- Capability = odkaz na objekt + množina práv definujících operace, spravuje jádro
- Jádro si vynucuje tento abstraktní pohled
- Jádro kontroluje přístupová práva
- Př: částečně Windows NT (Win 2000, XP,..) - hybridní



Hybridní jádro

- Kombinuje vlastnosti monolitického a mikrojádra
- Část kódu součástí jádra (monolitické)
- Jiná část jako samostatné procesy (mikrojádro)

- Příklady
  - Windows NT (Win 2000, Win XP, Windows Server 2003, Windows Vista,..)
  - Windows CE (Windows Mobile)
  - BeOS

GNU/Linux, GNU/Hurd

GNU/Linux
- GNU programy, např. gcc (R. Stallman)
- Monolitické jádro OS – Linux (Linus Torvald)

GNU/Hurd
- GNU programy, např. gcc
- Mikrojádro Hurd

Linux

- Pojem Linux jako takový označuje jádro operačního systému
- Pokud hovoříme o operačním systému, správně bychom měli říkat GNU/Linux, ale toto přesné označení používá jen málo distribucí, např. *Debian GNU/Linux*

- Flame war – debata mezi Linusem Torvaldsem a Andrewem Tanenbaumem, že Linux měl být raději mikrokernel

Linux - odkazy

Interaktivní mapa Linuxového jádra:
http://www.makelinux.net/kernel_map

Jaké jádro je nyní aktuální? (např. 3.11.1)
http://kernel.org/

Spuštění operačního systému bez instalace:
Live CD, Live DVD, Live USB
Např. Knoppix (http://knoppix.org/)

Kernel Version	Files	Lines
3.0	36,788	14,651,135
3.1	37,095	14,776,002
3.2	37,626	15,004,006
3.3	38,091	15,171,607
3.4	38,573	15,389,393
3.5	39,101	15,601,911
3.6	39,738	15,873,569
3.7	40,912	16,197,233
3.8	41,532	16,422,416
3.9	42,435	16,692,421
3.10	43,029	16,961,031

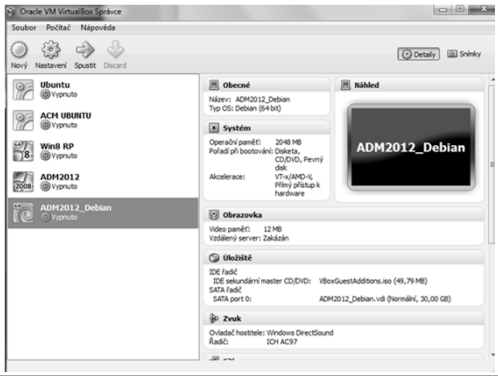
viz http://www.zive.cz/clanky/podivejte-se-kdo-opravdu-vyviji-linux/sc-3-a-170587/default.aspx

Virtualizace

- Možnost nainstalovat si virtuální počítač a provozovat v něm jiný operační systém, včetně přístupu k síti aj.

- VirtualBox
- VmWare
- Xen (např. virtualizace serverů), KVM aj.

VirtualBox



Literatura, použité zdroje

Obrázky z některých slidů (20, 21, 24) pocházejí z knížky

Andrew S. Tanenbaum: Modern Operating Systems

vřele doporučuji tuto knihu, nebo se alespoň podívat na slidy ke knize dostupné mj. na webu předmětu v *Přednášky -> Odkazy*

02. Koncepce OS
Procesy, vlákna

ZOS 2013

Vektor přerušení

Kód	Číslo přerušení	Popis
B	INT 00H	Dělení nulou
B	INT 01H	Krokování
B	INT 02H	Nemaskovatelné přerušení
B	INT 03H	Bod přerušení (breakpoint)
B	INT 04H	Přetečení
B	INT 05H	Tisk obrazovky
B	INT 06H	Nesprávný operační kód
B	INT 07H	Není koprocesor
B	INT 08H IRQ0	Přerušení od časovače
B	INT 09H IRQ1	Přerušení od klávesnice
	INT 0aH IRQ2	EGA vertikální zpětný běh
	INT 0bH IRQ3	COM2
	INT 0cH IRQ4	COM1
	INT 0dH IRQ5	Přerušení harddisku
B	INT 0eH IRQ6	Přerušení řadiče disket
	INT 0fH IRQ7	Přerušení tiskárny
B	INT 10H	Služby obrazovky
	INT 11H	Seznam vybavení
	INT 12H	Velikost volné paměti
B	INT 13H	Diskové vstupně-výstupní operace

Příklad možného mapování (původní IBM PC) , může být různé

dva pojmy:
INT …
IRQ …

všimněte si IRQ0 je zde na INT 08H, na vektoru 08H (tj.od adresy 8*4) bude adresa podprogramu k vykonání

IRQ – Interrupt Request

IRQ – signál, kterým zařízení (časovač, klávesnice) žádá procesor o přerušení zpracovávaného procesu za účelem provedení důležitější akce

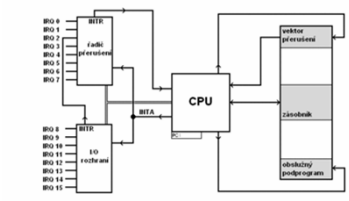
IRQL – priorita přerušovacího požadavku
(Interrupt Request Level)

NMI – nemaskovatelné přerušení, např. nezotavitelná hw chyba (non-maskable interrupt)

Obsluha HW přerušení

1. zařízení sdělí řadiči přerušení, že potřebuje přerušení
2. řadič upozorní CPU, že jsou čekající (pending) přerušení
3. až je CPU ochotné přijmout přerušení tak přeruší výpočet a zeptá se řadiče přerušení, které nejdůležitější čeká a spustí jeho obsluhu
4. uloží stav procesu, provede základní obsluhu zařízení, informuje řadič o dokončení obsluhy, obnoví stav procesu a pokračuje se dále

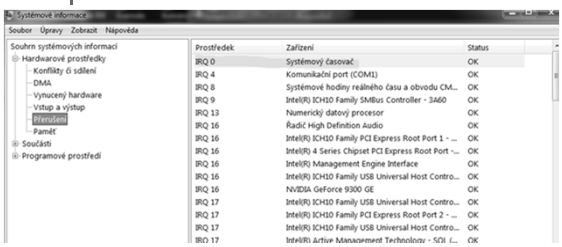
Řadič přerušení



2 integrované obvody Intel 8259
1. spravuje IRQ 0 až 7
(master, na IRQ2 je připojen druhý)
2. spravuje IRQ 8 až 16
novější systémy Intel APIC Architecture (typicky 24 IRQ)

obrázek – zdroj wikipedia

IRQ pod Win: msinfo32.exe



Linux: cat /proc/interrupts

Koncepce OS

- Základní abstrakce:
  - procesy
  - soubory
  - uživatelská rozhraní

Procesy

- **Proces** – instance běžícího programu
- **Adresní prostor** procesu
  - MMU (Memory Management Unit) zajišťuje soukromí
  - kód spustitelného programu, data, zásobník
- S procesem sdruženy registry a další info potřebné k běhu procesu = stavové informace
  - registry – čítač instrukcí PC, ukazatel zásobníku SP, univerzální registry

Registry (příklad architektura x86)

- malé úložiště dat uvnitř procesoru
- obecné registry
  - EAX, EBX, ECX, EDX .. jako 32ti bitové
  - AX, BX, CX, DX .. využití jako 16ti bitové (dolních 16)
  - AL, AH .. využití jako 8bitové
- obecné registry - uložení offsetu
  - SP .. ofset adresy vrcholu **zásobníku (!)**
  - BP .. pro práci se zásobníkem
  - SI .. offset zdroje (source index)
  - DI .. ofset cíle (destination index)

Registry

- segmentové registry
  - CS code segment (kód)
  - DS data segment (data)
  - ES extra segment
  - FS volně k dispozici
  - GS volně k dispozici
  - SS stag segment (zásobník)

Registry

- speciální
  - IP .. offset vykonávané instrukce (CS:IP)
  - FLAGS .. zajímavé jsou jednotlivé bity
    - IF .. interrupt flag (přerušení zakázáno-povoleno)
    - ZF .. zero flag (je-li výsledek operace 0)
    - OF, DF, TF, SF, AF, PF, CF

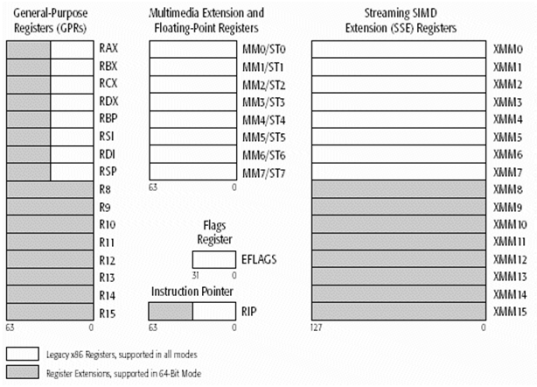
bližší info např. http://cs.wikipedia.org/wiki/Registr_procesoru
jde nám o představu jaké registry a k jakému účelu jsou

Registry (x86-64)

64bit

- For 16-bit operations, the two bytes of Register A are addresses as AX
- For 32-bitoperations, the four bytes of Register A are addressed as EAX
- For 64-bitoperations, the eight bytes of Register A are addressed as RAX

zdroj:
http://pctuning.tyden.cz/index2.php?option=com_content&task=view&id=7475&Itemid=28&pop=1&page=0



zdroj: http://www.usenix.org/events/bsdcon/full_papers/linden/linden_html/

Základní služby OS pro práci s procesy

- **Vytvoření nového procesu**
  - fork v UNIXu, CreateProcess ve Win32
- **Ukončení procesu**
  - exit v UNIXu, ExitProcess ve Win32
- **Čekání na dokončení potomka**
  - wait (waitpid) v UNIXu,
  - WaitForSingleObject ve Win32

Další služby - procesy

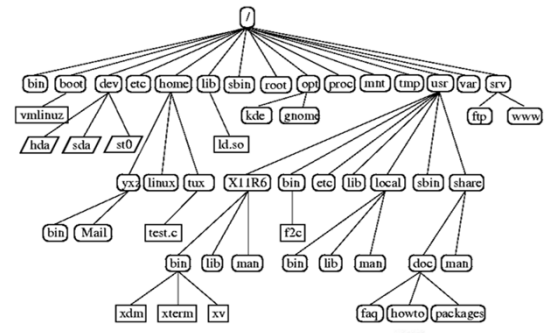
- Alokace a uvolnění paměti procesu
- Komunikace mezi procesy (IPC)
- Identifikace ve víceuživatel. systémech
  - identifikátor uživatele (UID)
  - skupina uživatele (GID)
  - proces běží s UID toho, kdo ho spustil
  - v UNIXu – UID, GID – celá čísla
- Problém uvíznutí procesu

Soubory

- Zakrytí podrobností o discích a I/O zařízení
- Poskytnutí abstrakce – soubor
- Systémová volání
  - vytvoření, zrušení, čtení, zápis
- Otevření a uzavření souboru – open, close
- Sekvenční nebo náhodný přístup k datům
- Logické sdružování souborů do adresářů
- Hierarchie adresářů – stromová struktura

Soubory II.

- Ochrana souborů, adresářů přístupovými právy
  - kontrola při otevření souboru
  - pokud není přístup – chyba
- Připojitelnost souborových systémů
  - Windows – disk určený prefixem *C:, D:*
  - Unix – *kamkoliv* v adresářovém stromu



Linux – filesystem
zdroj: http://www-uxsup.csx.cam.ac.uk/pub/doc/suse/suse9.0/userguide-9.0/ch24s02.html
http://www.cs.wits.ac.za/~adi/courses/linuxadmin/content/module2doc.html

Uživatelské rozhraní

- řádková – CLI (Command Line Interface)
- grafická uživ. rozhraní (GUI)

- původně UI součást jádra
- v moderních OS – jedním z programů, možnost náhrady za jiné

UI – obrázky

UI jako součást jádra **UI v uživ. režimu**

Kolik přepnutí kontextu je potřeba?
vs. vliv na stabilitu jádra OS

Uživatelské rozhraní - příklady

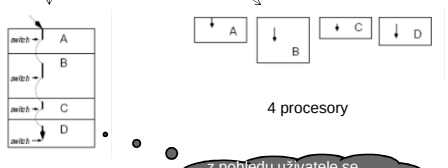
- GUI Linux
  - systém XWindow (zobrazování grafiky) a grafické prostředí (správci oken,...) – programy v uživatelském režimu
- Windows NT,2000,XP
  - grafická část v jádře
  - logická část (v uživatelském režimu)
  - výkon vs. stabilita

Proces jako abstrakce

- Běžící SW – organizován jako množina sekvenčních procesů
- Proces – běžící program včetně obsahu čítače instrukcí, registrů, proměnných; běží ve vlastní paměti
- Koncepčně každý proces – vlastní virtuální CPU
- Reálný procesor – přepíná mezi procesy (multiprogramování)
- Představa množiny procesů běžících (pseudo)paralelně

Ukázka

4 procesy, každý má vlastní bod běhu (čítač instrukcí)
pseudoparalelní běh x paralelní (více CPU)



z pohledu uživatele se nám jeví jako paralelní vykonávání procesů

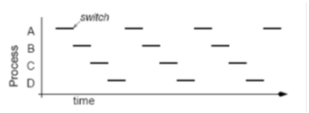
Pseudoparalelní běh

- Pseudoparalelní běh – v jednu chvíli aktivní pouze jeden proces

Vyprší časové kvantum, nebo chce I/O operaci

- Po určité době pozastaven a spuštěn další
- Po určité době všechny procesy vykonají část své činnosti

Pseudoparalelní běh



Procesy A, B, C,D se střídají na procesoru
Z obrázku se zdá, že na procesoru stráví vždy stejnou dobu, ale nemusí tomu tak být – např. mohou požadovat I/O operaci a „odevzdají“ procesor dříve

Rychlost procesů

- Rychlost běhu procesu není konstantní.
- Obvykle není ani reprodukovatelná.
- Procesy nesmějí mít vestavěné předpoklady o časování (!)
- Např. doba trvání I/O různá.
- Procesy neběží stejně rychle.

Proces běží v reálném systému, který se věnuje i dalším procesům, obsluze přerušení atd., tedy nesmíme spoléhat, že poběží vždy stejně rychle..

Stavy procesu

- Procesy často potřebují komunikovat s ostatními procesy:
- *ls –l | more*
- proces ls vypíše obsah adresáře na std. výstup
- more zobrazí obrazovku a čeká na klávesu
- More je připraven běžet, ale nemá žádný vstup – zablokuje se dokud vstup nedostane

Oba jsou spuštěny současně

Kdy proces neběží

Nemůže, na něco čeká

- Blokování procesu – proces nemůže pokračovat, protože čeká na zdroj (vstup, zařízení, paměť), který není dostupný – proces nemůže logicky pokračovat

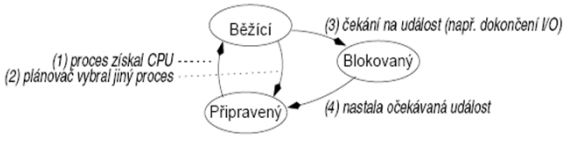
Chtěl by, ale není volný CPU

- Proces může být připraven pokračovat, ale CPU vykonává jiný proces – musí počkat, až bude CPU „volné“

Základní stavy procesu

- **Běžící** (running)
  – skutečně využívá CPU, vykonává instrukce
- **Připravený** (ready, runnable)
  – dočasně pozastaven, aby mohl jiný proces pokračovat
- **Blokovaný** (blocked, waiting)
  – neschopný běhu, dokud nenastane externí událost

Základní stavy procesu (!!)



Přechody stavů procesu

1. Plánovač **vybere** nějaký proces
2. Proces je **pozastaven**, plánovač vybere jiný proces (typicky - vypršelo časové kvantum)
3. Proces se **zablokuje**, protože čeká na událost (zdroj – disk, čtení z klávesnice)
4. **Nastala** očekávaná **událost**, např. zdroj se stal dostupný

Stavy procesů

- Jádro OS obsahuje plánovač
- Plánovač určuje, který proces bude běžet
- Nad OS řada procesů, střídají se o CPU

- Stav procesu **pozastavený**
- V některých systémech může být proces **pozastaven** nebo **aktivován**
- V diagramu přibudou **dva** nové stavy

Stavy procesů

Tabulka procesů

OS si musí vést evidenci, jaké procesy v systému v danou chvíli existují.
Tato informace je vedena v **tabulce procesů**.

Každý proces v ní má záznam, a tento záznam se nazývá process control block (PCB).

Na základě informací zde obsažených se plánovač umí rozhodnout, který proces dále poběží a bude schopen tento proces spustit ze stavu, v kterém byl naposledy přerušen.

PCB (Process Control Block) !

- OS udržuje tabulku nazývanou tabulka procesů
- Každý proces v ní má položku zvanou PCB (Process Control Block)
- PCB obsahuje všechny informace potřebné pro opětovné spuštění přerušeného procesu
  - Procesy se o CPU střídají, tj. jeho běh je přerušovaný
- Konkrétní obsah PCB – různý dle OS
- Pole správy procesů, správy paměti, správy souborů (!!)

Položky - správa procesů

- Identifikátory (číselné)
  - Identifikátor procesu - PID
  - Identifikátor uživatele - UID
- Stavová informace procesoru
  - Univerzální registry,
  - Ukazatel na další instrukci - PC
  - ukazatel zásobníku SP
  - Stav CPU – PSW (Program Status Word)
- Stav procesu (běžící, připraven, blokován)
- Plánovací parametry procesu (algoritmus, priorita)

Položky – správa procesů II

- Odkazy na rodiče a potomky
- Účtovací informace
  - Čas spuštění procesu
  - Čas CPU spotřebovaný procesem
- Nastavení meziprocesové komunikace
  - Nastavení signálů, zpráv

Položky – správa paměti

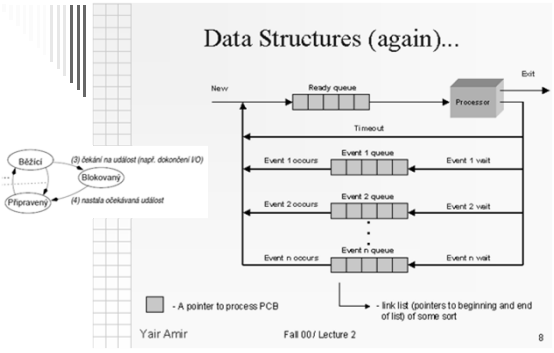
- Popis paměti
  - Ukazatel, velikost, přístupová práva
1. Úsek paměti s kódem programu
2. Data – hromada
   - Pascal – new release
   - C – malloc, free
3. Zásobník
   - Návratové adresy, parametry funkcí a procedur, lokální proměnné

Položky – správa souborů

- Nastavení prostředí
  - Aktuální pracovní adresář
- Otevřené soubory
  - Způsob otevření – čtení / zápis
  - Pozice v otevřeném souboru

PCB

Pointer	Process state
Process number	
Program counter	
Registers	
Memory limits	
List of open files	
…	



Viz http://www.cs.jhu.edu/~yairamir/cs418/os2/sld007.htm

Poznámky

- Stav Nový
  - Proces přejde z nový do stavu Připravený
- Stav Ukončený
  - Přechod ze stavu běžící do ukončený, např. voláním exit

Častou chybou je, že lidé kreslí přechod ze stavu Nový do stavu Běžící, napřed se musí jít přes Připravený ! Stejně tak, do stavu Ukončený jdeme ze stavu Běžící.

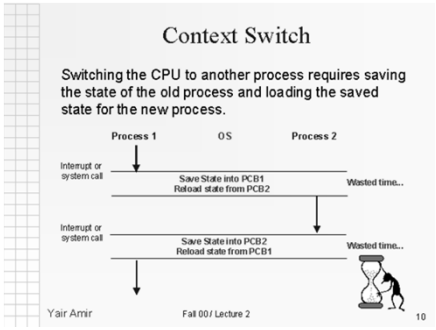
Ukončení procesu - možnosti

I. Proces úspěšně vykoná kód programu ☺

II. Skončí rodičovský proces a OS nedovolí pokračovat child procesu (záleží na OS, někdy ano někdy ne)

III. Proces překročí limit nějakého zdroje

Přepnutí procesu



Přepnutí procesu - průběh

- Systém nastaví časovač – pravidelně přerušení
- Na předem definovaném místě – adresa obslužného programu přerušení
- CPU po příchodu přerušení provede:
  - Uloží čítač instrukcí PC do zásobníku
  - Načte do PC adresu obsluž. programu přerušení
  - Přepne do režimu jádra

Přepnutí procesu - II

K přeplánování procesu nedojde při každém tiku časovače, ale až, když jich je tolik, že vyprší časové kvantum

- Vyvolána obsluha přerušení:
  - Uloží obsah registrů do zásobníku
  - Nastaví nový zásobník
- Plánovač nastaví proces jako ready, vybere nový proces pro spuštění
- Přepnutí kontextu
  - Nastaví mapu paměti nového procesu
  - Nastaví zásobník, načte obsah registrů
  - Provede návrat z přerušení – RET (do PC adresa ze zásobníku, přepne do uživatelského režimu)

Rychlost CPU vs. paměti

Cílem následující vsuvky je říci, že výkon systému může degradovat nejenom časté střídání procesů, protože se pořád musí přepínat kontext,
ale i fakt, že informace v cachi se po přepnutí na jiný proces stane neaktuální, a cache paměti chvíli trvá, než se naplní aktuálními daty, což má také vliv na výkon systému.

Rychlost CPU vs. paměť

- CPU
  - Rychlost – počet instrukcí za sekundu
  - Obvykle nejrychlejší komponenta v systému
  - Skutečný počet instrukcí závisí na rychlosti, jak lze instrukce a data přenášet z a do hlavní paměti
- Hlavní paměť
  - Rychlost v pamětových cyklech (čtení, zápis)
  - O řád pomalejší než CPU
  - Proto důvod používat cache paměť

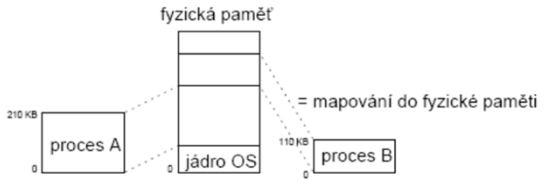
Rozdíly rychlostí – „pyramida“

- CPU registry – rychlé – zápisníková pamět, 32x32 nebo 64x64 bitů, žádné zpoždění při přístupu
- Cache – malá paměť s vysokou rychlostí,
  - princip lokality,
  - pokud jsou data v cache – dostaneme velmi rychle, 2 tiky hodin
- RAM
- Vnější pamět
  - Mechanická, pomalejší, větší kapacita, levnější cena za bit

MMU – Memory Management Unit

- Více procesů v paměti
  - Každý proces pamět pro sebe, např. od adresy 0 (relokace)
  - Ochrana – nemůže zasahovat do paměti ostatních procesů ani jádra
- Mezi CPU a pamětí je MMU
  - Program pracuje s virtuálními adresami
  - MMU je převede na fyzické adresy

MMU



Výkonnostní důsledky

- Pokud program nějakou dobu běží – v cache jeho data a instrukce – dobrá výkonnost
- Při přepnutí na jiný proces – převažuje přístup do hlavní paměti (keš není naučená)
- Nastavení MMU se musí změnit
- Přepnutí mezi úlohami i přepnutí do jádra (volání služby OS) – relativně drahé (čas)

Služby pro práci s procesy

- Jednoduché systémy
  - Všechny potřebné procesy spuštěny při startu systému
  - Běží po celou dobu běhu systému – žádné služby nepotřebujeme
  - Některé zapouzdřené (embedded) systémy

UNIX a Linux

- Služba **fork()** – vytvoří přesnou kopii rodičovského procesu
- Návratová hodnota – rozliší mezi rodičem a potomkem (potomek dostane 0)
- *pid = fork();*
- *if (pid == 0) potomek else rodic*
- Potomek může činnost ukončit pomocí exit()
- Rodič může na potomka čekat – wait()

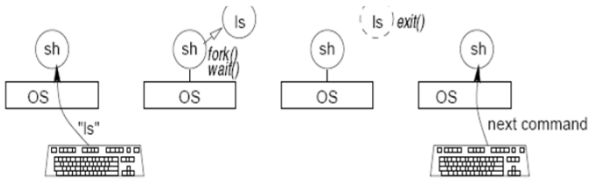
UNIX

- Potomek může místo sebe spustit jiný program – volání execve() – nahradí obsah paměti procesem spouštěným ze zadaného souboru

1. *if (fork() == 0)*
2. *execve("/bin/ls", argv, envp);*
3. *else*
4. *wait(NULL);*

Příkazový interpret

- Spouští příkaz – vytvoří nový proces, čeká na jeho dokončení; ukončení – volání sl. systému



Win32

- Vytvoření procesu službou CreateProcess
- Mnoho parametrů – vlastnosti procesu

Win32 ukázka

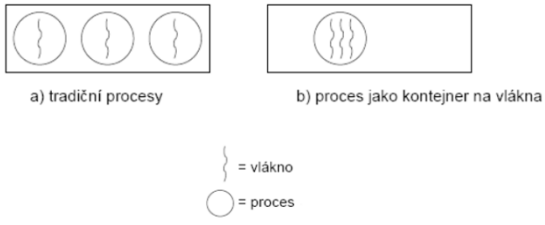
```
STARTUPINFO StartInfo; // name structure
PROCESS_INFORMATION ProcInfo; // name structure
memset(&ProcInfo, 0, sizeof(ProcInfo)); // Set up memory block
memset(&StartInfo, 0 , sizeof(StartInfo)); // Set up memory block
StartInfo.cb = sizeof(StartInfo); // Set structure size
int res = CreateProcess(NULL, "MyApp.exe", NULL, NULL, NULL, NULL, NULL, NULL, &StartInfo, &ProcInfo); // starts
MyApp
if (res)
{
    WaitForSingleObject(ProcInfo.hThread, INFINITE); // wait forever for process to finish
    SetFocus(); // Bring back focus
}
```

příklad viz
http://msdn.microsoft.com/en-us/library/windows/desktop/ms682512%28v=vs.85%29.aspx

Procesy a vlákna

- Tradiční OS – každý proces svůj vlastní adresový prostor a místo kde běží (bod běhu)
- Často výhodné – více bodů běhu, ale ve stejném adresovém prostoru
- Bod běhu – vlákno (thread, lightwight process)
- Více vláken ve stejném procesu - multithreading

Procesy a vlákna



Vlákna (!!)

- Vlákna v procesu sdílejí adresní prostor, otevřené soubory (atributy procesu)
- Vlákna mají soukromý čítač instrukcí, obsah registrů, soukromý zásobník
  - Mohou mít soukromé lokální proměnné
- Původně využívána zejména pro VT výpočty na multiprocesorech (každé vlákno vlastní CPU, společná data)

Vlákna – použití dnes

- Rozsáhlejší výpočet a rozsáhlejší i/o
- Interaktivní procesy – jedno vlákno pro komunikaci s uživatelem, další činnost na pozadí
- www prohlížeč – jedno vlákno příjem dat, další zobrazování a interakce s uživatelem
- Textový procesor – vstup dat, přeformátování textu
- Servery www – jedno vlákno pro každého klienta

Multithreading

- Podporován většinou OS
  - Linux, Windows
- Podporován programovacími jazyky
  - Java, knihovny v C, …
- Proces začíná svůj běh s jedním vláknem, ostatní vytváří za běhu programově (konstrukce vytvoř vlákno)
- Režie na vytvoření vlákna a přepnutí kontextu menší než v případě procesů (!)

Poznámka (terminologie)

- Jeden proces – více vláken
  - Ošetření souběžného přístupu ke sdílené paměti
- Více procesů sdílejících pamět
  - Ošetření souběžného přístupu ke sdílené paměti
- V literatuře např. při řešení synchronizace, se většinou nerozlišuje, zda uvažujeme souběžný přístup vláken nebo procesů ke společné paměti

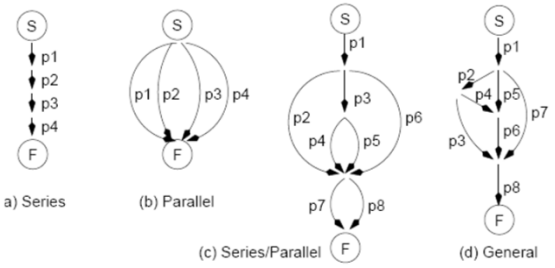
Programové konstrukce pro vytváření vláken

- Statické
  - Proces obsahuje deklaraci pevné množiny podprocesů (např. tabulka)
  - Všechny spuštěny při spuštění procesu
- Dynamické
  - Procesy mohou vytvářet potomky dynamicky
  - častější
- Pro popis – precendenční grafy

Precedenční grafy

- Popis pro vyjádření různých relací mezi procesy
- Process flow graph
- Acyklický orientovaný graf
- Běh procesu p_i – orientovaná hrana grafu
- Vztahy mezi procesy – seriové nebo paralelní spojení – spojením hran

Precedenční grafy



Fork, join, quit

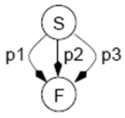
Mechanismus pro obecný popis paralelních aktivit

primitivum	funkce
fork X;	Spuštění nového vlákna od příkazu označeného návěštím X; nové vlákno poběží paralelně s původním
quit ;	Ukončí vlákno
joint t, Y;	Atomicky (nedělitelně) provede: t= t – 1; if (t==0) then goto Y;

Běh procesů odpovídající precendenčnímu grafu

Nevíme, který z procesů doběhne první a který poslední, jen jeden z možných běhů

Zápis pomocí fork-join-quit




```
      n=3;              // tři procesy
      fork L2;          // spustíme vlákno od L2
      fork L3;          // spustíme vlákno od L3
      p1; join n, L4; quit;    // jen 1. vlákno
L2:   p2; join n, L4; quit;    // jen 2. vlákno
L3:   p3; join n, L4; quit;    // jen 3. vlákno
L4:   ….                       // zde jen poslední
F:    ….
```

Poznámky k fork-join-quit

+ obecný zápis
- špatná čitelnost (přehlednost)

V některé literatuře se neuvádí quit, a předpokládá se join = join + quit

Správně vnořené precedenční grafy

S(a,b) – sériové spojení procesů
(za procesem **a** následuje **b**)

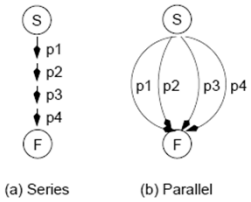
P(a,b) – paralelní spojení procesů **a** a **b**

Precedenční graf je správně vnořený, pokud může být popsán kompozicí funkcí S a P

Příklady správně vnořených grafů

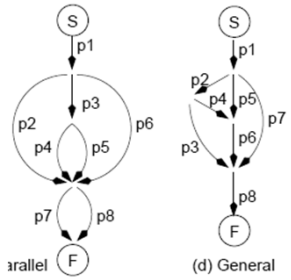
S (p1, S(p2, S(p3, p4)))

P (p1, P(p2, P(p3, p4)))



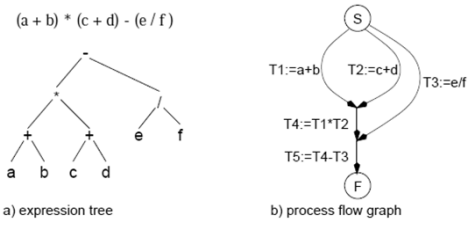
Graf (d) není správně vnořený
Nelze jej popsat kompozicí S a P

Graf vlevo lze:



S (p1, S(P(p2, P(S(p3, P(p4,p5)), p6)), P(p7,p8))

Příklad vyhodnocení aritmetického výrazu



Vznikají správně vnořené procesy; dodržet maximální paralelismus !

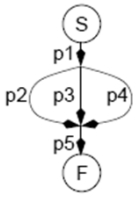
Abstraktní primitiva cobegin, coend

- Dijkstra (1968), původně parbegin,..
- Specifikuje sekvence programu, která má být spuštěna paralelně

cobegin
$C_1 \parallel C_2 \parallel \ldots \parallel C_n$
coend

Každé Ci … autonomní segment kódu (blok)
Samostatné vlákno pro všechna Ci
Ci běží nezávisle na ostatních
Program pokračuje za coend až po skončení posledního Ci

Příklad – cobegin, coend




```
begin
  C1;
  cobegin
    C2 || C3 || C4
  coend
  C5
end
```

Vztah cobegin/coend a funkcí P, S

- Každý segment kódu Ci lze dekomponovat na sekvenci příkazů pi:
  S (pi1, S (pi2, …))
- Konstrukce cobegin $C_1 \parallel C_2 \parallel$.. coend odpovídají vnoření funkcí:
  $P(C_1, P(C_2, \ldots))$

Příklad – aritmetický výraz (a+b) * (c+d) – (e/f)

```
begin
   cobegin
      begin
         cobegin
               T1 = a+b || T2 = c+d
         coend
         T4 = T1 * T2
      end
      || T3 = e/f
   coend
   T5 = T4 – T3
end
```

Maximální paralelismus

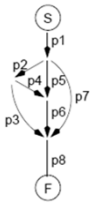
Část výpočtu spustím ihned jak je to možné

Např. T1,T2,T3

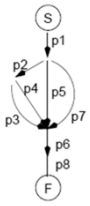
Příklad – fork, join, quit (a+b) * (c+d) – (e/f)

```
      n := 2;
      fork L3;
      m := 2;
      fork L2;
      t1 := a + b;   join m, L4; quit;
L2:   t2 := c + d;   join m, L4; quit;
L4:   t4 := t1 * t2; join n, L5; quit;
L3:   t3 := e/f;     join n, L5; quit;
L5:   t5 := t4-t3;
```

Lze nesprávně vnořený graf upravit?



(a) General



(b) "properly nested"

Můžeme „beztrestně“ posunout proces p6?

Ne vždy !!

Pokud jsou závislé, a p6 musí běžet paralelně s p3 a p7, např. si vyměňují zprávy, pak toto nelze.

Fork – join – quit popíše i nesprávně vnořené grafy

Př. iterace

```
      for i:=1 to m do
         for j:=1 to n do
            fork E;
      quit;
E:    A[i][j]:= ...
      join t, R;
      quit;
R:    ...
```

Soukromé kopie proměnných rodičovského vlákna
Každé vlákno vytvořené fork E má soukromou kopii i, j
Deklarace typu „private“

Ada – statická deklarace podprocesu

```
process p
  deklarace ..   // mohou být další definice
begin              podprocesů, spuštěny při
  …                spuštění p
end
```

Ada – dynamická deklarace podprocesu

```
process type p2          // šablona
  deklarace ..
begin
  …
end
begin
  q = new p2;
end
```

Vlákna v systému UNIX a jazyce C

- Knihovna libpthread
- Jako vlákno se spustí určitá funkce
- Návratem z této funkce vlákno zanikne

Základní funkce

funkce	popis
t = pthread_create(..f..)	Podprogram f se spustí jako vlákno vrací id vlákna
pthread_exit ()	Odpovídá quit, může předat návratovou hodnotu
x = pthread_join (t)	Čeká na dokončení vlákna t vrací hodnotu předanou voláním exit
pthread_detach (t)	Na dokončení vlákna se nebude čekat joinem
pthread_cancel (t)	Zruší jiné vlákno uvnitř stejného procesu

zkuste: man pthread_create

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

void *vlakno(void *m) /* podprogram pro vlákno */
{
        int i;

        for (i=0; i<10000; i++)
                write(1, m, 1);
        return NULL;
}

int main(int argc, char *argv[])
{
        pthread_t th1, th2;

        pthread_create(&th1, NULL, vlakno, "*"); /* vytvoří vlákno */
        pthread_create(&th2, NULL, vlakno, ".");
        pthread_join(th1, NULL);        /* čeká na dokončení vlákna */
        pthread_join(th2, NULL);

        return 0;
}
```

Překlad programu s vlákny

na stroji eryx.zcu.cz:

gcc –lpthread –o jedna jedna.c

./jedna

- gcc .. překladač
- -lpthread .. použijeme knihovnu vláken
- -o jedna .. výsledný spustitelný soubor
- jedna.c .. zdrojový kód v C
- ./jedna .. spustíme program z akt. adresáře

Java – základ práce s vlákny

- Třída java.lang.Thread
- Programátor vytvoří podtřídu s vlastní metodou run() .. činnost vlákna
- Spustí se vytvořením instance podtřídy a spuštěním metody start()

```
MyThread t = new MyThread();
t.start();
```

Java – rozhraní Runnable

- Rozhraní Runnable
- Třída může definovat metodu run(), ale sama nemusí být potomkem třídy Thread
- Viz pozdější cvičení

Další materiály

- Viz texty k přednáškám: *p2proc.pdf*

03. Synchronizace procesů

ZOS 2013, L. Pešička

Opakování

- Kde je uložený PID procesu?
  v PCB v tabulce procesů
- Jaké systémové volání vytvoří nový proces?
  Linux: fork() Windows: fce CreateProcess()
- Jakým způsobem spustím jiný program?
  Linux: execve() , často v kombinaci s fork()

Stavy procesů – poznámky k implementaci v Linuxu

- Zombie
  - Proces dokončil svůj kód
  - Stále má záznam v tabulce procesů
  - Čekání, dokud rodič nepřečte exit status (voláním wait()); příkaz ps zobrazuje stav "Z"
- Sirotek
  - Jeho kód stále běží, ale skončil rodičovský proces
  - Adoptován procesem init

Jak na zombii?

```
#include <stdio.h>
int main (void) {
    int i,j;
    i = fork();
if (i == 0)
  printf ("Jsem potomek s pidem %d, rodic ma %d\n", getpid(),
    getppid());
else {
  printf ("Jsem rodic s pidem %d, potomek ma %d\n", getpid(), i);
  for (j=10; j<100; j++) j=11;  // rodic neskonci, nekonečná smyčka
  }
}
```

Potomek skončí hned, ale rodič se točí ve smyčce

Plánování procesů

- **Krátkodobé** – CPU scheduling
  kterému z připravených procesů bude přidělen procesor; vždy ve víceúlohovém

typický plánovač jak jej známe

- **Střednědobé** – swap out
  odsun procesu z vnitřní paměti na disk
- **Dlouhodobé** – job scheduling
  výběr, která úloha bude spuštěna
  dávkové zpracování
  (dostatek zdrojů – spusť proces)

Liší se – frekvencí spouštění plánovače

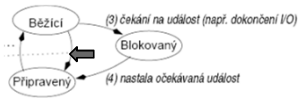
Plánování procesů

Stupeň multiprogramování

- Počet procesů v paměti
- Zvyšuje: long term scheduler
- Snižuje: middle term scheduler

Ne v každém OS musí být všechny tři typy plánovače, typicky jen krátkodobý plánovač

Plánování



- Nepreemptivní
  - Proces skončí
  - Běžící -> Blokovaný
    - Čekání na I/O operaci
    - Semafor
    - Čekání na ukončení potomka
- Preemptivní
  - Navíc přechod: Běžící -> Připravený
    - Uplynulo časové kvantum

Proces opustí CPU: jen když skončí, nebo se zablokuje

Preemptivní navíc opustí CPU při uplynutí časového kvanta
Problém – proces může být přerušen kdykoliv, bohužel i v nevhodný čas

Vlákna

Vlákna mohou být implementována:
- V jádře
- V uživatelském prostoru
- Kombinace

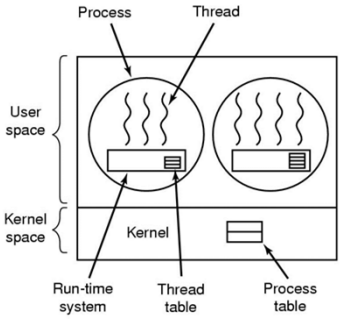
Zná jádro pojem vlákna?
Jsou v jádře plánována vlákna nebo procesy?

Vlákna v User Space

Jádro plánuje procesy,

O vláknech nemusí vůbec vědět.

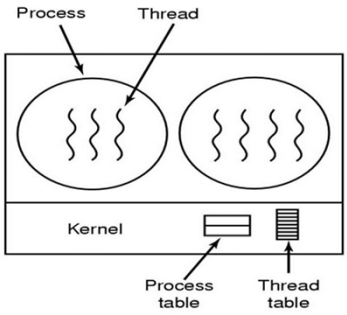
Pokud vlákno zavolá systémové volání, celý proces se zablokuje



Vlákna v jádře

Jádro plánuje jednotlivá vlákna.

Kromě tabulky procesů má i tabulku vláken.

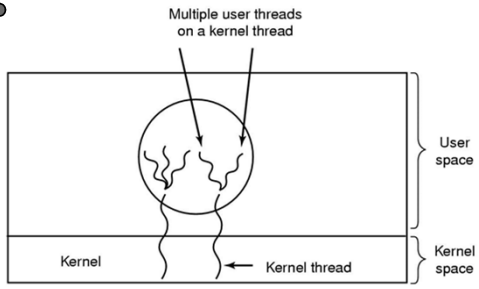


Hybridní implementace

Mapování vláken v user space na vlákna v jádře

Různá mapování

Zobecněný model



Modely - vlákna

- one-to-one (1:1) .. vlákna v jádře
  - Každé vlákno – separátní "proces" v jádře
  - Plánovač jádra je plánuje jako běžné procesy
  - Základní jednotkou plánování jsou vlákna
- many-to-one (M:1) .. vlákna jen v user space
  - User level plánovač vláken
  - Z pohledu jádra – vlákna 1 procesu jako pouze 1 proces
- many-to-many (M:N)
  - Komerční unixy (Solaris, Digital Unix, IRIX)

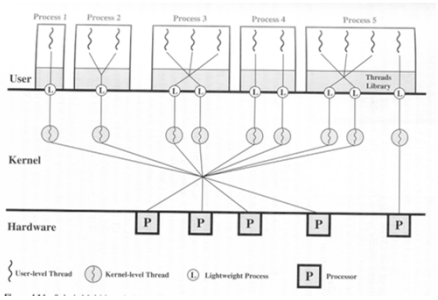
Vlákna - Solaris

Často uváděný příklad obecné koncepce vláken

- Uživatelská vlákna, vlákna jádra
- Lehké procesy (LWP)
- Každý proces – min. 1 LWP
- Uživatelská vlákna multiplexována mezi LWP procesy
- Pouze vlákno napojené na některý LWP může pracovat
- Ostatní blokována nebo čekají na uvolnění LWP

- Každý LWP proces – jedno vlákno jádra
- Další vlákna jádra bez LWP – např. obsluha disku
- Vlákna jádra – multiplexována mezi procesory

Vlákna Solaris



Linux

- Systémové volání **clone()**
  - Zobecněná verze fork()
  - One-to-one model
  - Dovoluje novému procesu sdílet s rodičem
    - Paměťový prostor
    - File descriptors
    - Signal handlers
  - Specifické pro Linux, není přenositelné (portable), není obecně v unixových systémech

Můžeme říci, co z uvedeného bude sdíleno

Pthreads

Knihovna vláken

- Historicky každý výrobce měl svoje řešení
- UNIX – IEEE POSIX 1003.1c standard (1995)
  - POSIX .. jednotné rozhraní, přenositelnost programů
  - Implementace POSIX threads – Pthreads

- gcc –lpthread -o vlakna vlakna.c (překlad na eryxu)

- http://yolinux.com/TUTORIALS/LinuxTutorialPosixThreads.html
- http://www.root.cz/clanky/programovani-pod-linuxem-tema-vlakna/
  - Série článků, procesy, vlákna, synchronizace, …

PTHREADS

- Rozhraní specifikované IEEE POSIX 1003.1c (1995)
- Implementace splňující tento standard: POSIX threads , pthreads
- Popis v pthread.h

1. **Management** vláken (create, detach, join)
2. **Mutexy** (create, destroy, lock, unlock)
3. **Podmínkové proměnné** (create, destroy, wait, signal)
4. **Další synchronizace** (read-write locks, bariéry)

Implementace v Linuxu - dříve

- Název: Linux threads
  - Starší
  - Používala clone()
  - Využívala signály SIGUSR1 a SIGUSR2 pro koordinaci vláken, nemohl je použít uživatel
  - zde je jen pro zajímavost

Implementace v Linuxu - dnes, kernel 2.6.* a další

Native Posix Thread Library (NPTL)

- Také využívá systém.volání clone()
- Synchronizační primitivum futex
- Implementace 1:1
  - Vlákno vytvořené uživatelem pthread_create() odpovídá 1:1 plánovatelné entitě v jádře (task)
  - Výhodou – rychlost 100 000 vláken na IA-32 2s bez NPTL cca 15 min

phtreads – základní funkce

funkce	popis
pthread_create()	Vytvoří nové vlákno
pthread_join()	Čeká na dokončení vlákna

Vlákna: základní funkce

- **#include <pthread.h>** .. vlákna pthread
- **pthread_t a, b;** .. id vláken a,b
- **pthread_create(&a, NULL, pocitej, NULL)**
  - a – id vytvořeného vlákna
  - NULL – atributy vlákna (man pthread_attr_init)
  - pocitej – funkce vlákna
  - NULL – argument předaný funkci pocitej
  - Návratová hodnota 0 – vlákno se podařilo vytvořit
- **pthread_join(a, NULL);**
  - Čeká na dokončení vlákna s id a
  - Vlákno musí být v joinable state (ne detach, viz atributy)
  - NULL – místo null lze číst návrat. hodnotu

Příklad – vlákna – fce vlákna

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

void *print_message_function( void *ptr )
{
    char *message;
    message = (char *) ptr;
    printf("%s \n", message);
}
```

funkce vlákna

Příklad – vlákna - main

```
main()
{
    pthread_t thread1, thread2;
    char *message1 = "Thread 1";
    char *message2 = "Thread 2";
    int  iret1, iret2;

   /* vytvoříme 2 vlákna, každé pustí podprogram s různým parametrem */

    iret1 = pthread_create( &thread1, NULL, print_message_function, (void*)
message1);
    iret2 = pthread_create( &thread2, NULL, print_message_function, (void*)
message2);
```

Příklad – vlákna - main

```
/* hlavni vlákno bude čekat na dokončeni spuštěných vláken   */
 /* jinak by mohlo hrozit, že skončí dřív než spuštěná vlákna */
                    ===== zde 1+2=3 vlákna ====

    pthread_join( thread1, NULL);
    pthread_join( thread2, NULL);
                              ===== zde 1 hlavní vlákno ====
    printf("Thread 1 returns: %d\n",iret1);
    printf("Thread 2 returns: %d\n",iret2);
    exit(0);
}
```

Jiný příklad: předání parametru vláknu

```
//vytvareni vlaken
   for (i = 0; i < THREAD_COUNT; i++) {
      thID = malloc(sizeof(int));
      *thID = i + 1;
      pthread_create(&threads[i], NULL, thread, thID);
   }
// funkce vlakna
void *thread(void * args) {
printf("Jsem vlakno %d\n", *((int *) args)  );
}
```

Proces UNIXU – obsahuje informace:

- Proces ID, proces group ID, user ID, group ID
- Prostředí
- Pracovní adresář
- Instrukce programu
- Registry
- Zásobník (stack)
- Halda (heap)
- Popisovače souborů (file descriptors)
- Signal actions
- Shared libraries
- IPC (fronty zpráv, roury, semafory, sdílená paměť)

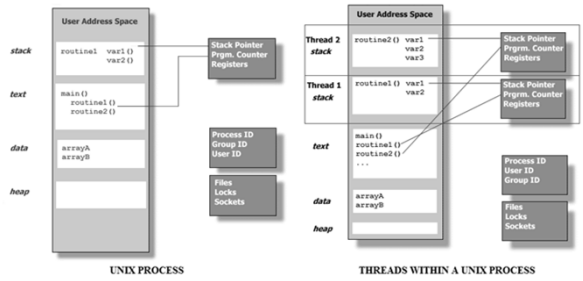
Vlákno má vlastní (!!):

- Zásobník (stack pointer)
- Registry
- Plánovací vlastnosti (policy, priority)
- Množina pending a blokovaných signálů
- Data specifická pro vlákno

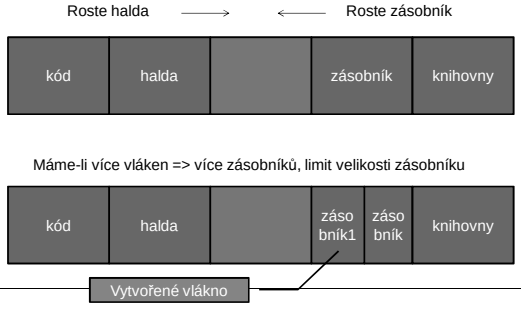
Všechna vlákna uvnitř stejného procesu sdílejí stejný adresní prostor

Mezivláknová komunikace je efektivnější a snadnější než meziprocesová

proces vs. proces s více vlákny
(rozdělení paměti je jen ilustrativní)



Rozdělení paměti pro proces

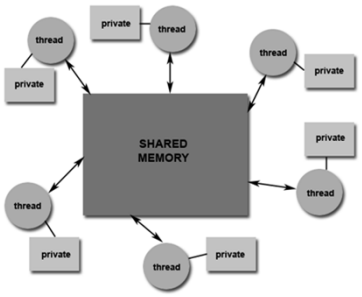


Zásobník pro vlákno

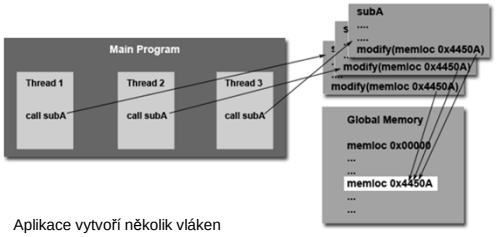
- Při vytvoření vlákna můžeme specifikovat velikost zásobníku
- Je potřeba celkem šetřit..
  Při max. velikost 8MB * 512 vláken = 4 GB

Globální a privátní paměť vlákna

více vláken
stejného procesu

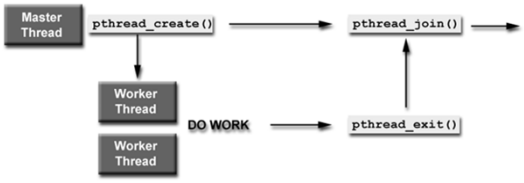


Vláknová bezpečnost (thread-safe)



Aplikace vytvoří několik vláken
Každé vlákno vyvolá stejnou rutinu
Tato rutina modifikuje společná globální data
– pokud nemá synchronizační mechanismy, není thread-safe

Čekání na dokončení vláken



Možnosti ukočení vlákna

- Vlákno dokončí „proceduru vlákna“
- Vlákno kdykoliv zavolá pthread_exit()
- Vlákno je zrušené jiným přes pthread_cancel()
- PROCES je ukončen zavoláním exec() nebo exit()
- Pokud main() skončí první bez explicitního volání pthread_exit()

Vlákna - Java

- Vlákno – instance třídy java.lang.Thread
- Odvodit potomka, překrýt metodu run()
  - Vlastní kód vlákna
- Spuštění vlákna – volání metody start() třídy Thread
- Další možnost - třída implementující rozhraní Runnable

class Něco implements Runnable {
public void run() { … } }

Problémy preemptivních systémů

Pokud je systém preemptivní (což často chceme, aby se procesy rychle střídaly na CPU), může dojít k odstavení procesu od procesoru v nevhodný čas

Např. manipuluje se sdílenou datovou strukturou, a než dokončí všechny potřebné akce, dojde k přeplánování na jiný proces (vlákno), což může vést ke špatnému výsledku

Taková chyba se může projevit velmi nepravidelně, třeba 1x za 100 000 běhů programu.

Synchronizace procesů

- Časový souběh
- Kritická sekce
- Algoritmy pro přístup do kritické sekce
- Semafory

Časový souběh

- Procesy sdílejí společnou paměť – čtení a zápis
- Může nastat časový souběh (race condition)
- Př.: dva procesy zvětšují asynchronně společnou proměnnou X

Příklad dvou procesů

```
cobegin
   ...
   x := x + 1;
   ...
   ||
   ...
   x := x + 1;
   ...
coend
```

1.proces

2.proces

společná paměť:

x

Příkaz na nízkoúrovňové instrukce

x := x + 1;

1. Načtení hodnoty x do registru (LD R, x)
2. Zvýšení hodnoty x (INC R)
3. Zápis nové hodnoty do paměti (LD x, R)

Pokud oba procesy provedou příkazy sekvenčně, bud mít x správně x+2

Chybné pořadí vykonání

Přepnutí v nevhodném okamžiku.. Pseudoparalelní běh

1. P1: LD R, x // x je 0, R je 0
2. P2: LD R, x // x je 0, R je 0
3. INC R // x je 0, R je 1
4. LD x, R // x je 1, R je 1
5. P1: // x je 1, R je 0 – rozpor
6. INC R // x je 1, R je 1
7. LD x, R // x je 1, R je 1

Výsledek – chyba, neprovedlo se každé zvětšení, místo 2 je 1

Chybné vykonání – 2 CPU

Chyba i při paralelním běhu

```
Proces 1:          Proces 2:

  LD R, x            ...
  INC R              LD R, x
  LD x, R            INC R
  ...                LD x, R
```

K chybě může dojít jak při pseudoparalelním běhu, tak i při paralelním běhu

Př. bankovní transakce

Dva procesy přístup do databáze

- Účet := účet + 20 000 1. proces
- Účet := účet – 15 000 2. proces

Správný výsledek?
Možné výsledky?

Časový souběh – další příklady

- Přidávání prvku do seznamu
  - Častá činnost v systémovém programování
- Přístup do souboru
  - 2 procesy chtějí vytvořit soubor a zapsat do něj
  - 1. proces – zjistí, že soubor není
  - ... přeplánování ...
  - 2. proces – zjistí, že soubor není, vytvoří a zapíše
  - 1. proces – pokračuje, vytvoří a zapíše
    - znehodnotí činnost druhého procesu

Výskyt souběhu

- časový souběh se projevuje nedeterministicky
- většinu času běží programy bez problémů
- hledání chyby je obtížné

Řešení časového souběhu

- pokud čtení a modifikace atomicky
  - atomicky = jedna nedělitelná operace
  - souběh nenastane
- hw většinou není praktické zařídit
- sw řešení
  - v 1 okamžiku dovolíme číst a zapisovat společná data pouze 1mu procesu
  - => ostatním procesům zabránit

Kritická sekce

- sekvenční procesy
  - komunikace přes společnou datovou oblast
- kritická sekce (critical section, region)
  - místo v programu, kde je prováděn přístup ke společným datům
- úloha – jak implementovat, aby byl v kritické sekci v daný okamžik pouze 1 proces

Společná datová oblast

- hlavní paměť (sdílené proměnné x,y,z,..)
- soubor
  - pokud 1 proces pracuje s jinou hodnotou, než jakou očekává jiný proces
  - zamykání částí souboru – řeší časový souběh
- každá kritická sekce se vztahuje ke konkrétním datům, ke kterým se v ní přistupuje

Počet kritických sekcí

- Kritická sekce nemusí být jedna
- Pokud procesy sdílejí tři proměnné x, y, z
  - Každá z nich představuje KS1, KS2, KS3

Mohli bychom sice říci, že jde o jednu KS, ale potom bychom zbytečně blokovali přístup k y, řešíme-li souběh nad x atd.

Analogie: když potřebujeme zamknout řádku tabulky v databázi, není potřeba zamykat celou tabulku, která může mít třeba milion záznamů – vliv na výkon systému

Struktura procesů

```
cobegin
P1: while true do          // nekonečná smyčka
 begin
   nevinná_činnost;        // pouze s vlastními daty
   kritická_sekce          // přístup do sdílených dat
end
||
P2: ...                    // totéž co P1
coend
```

Cílem slidu je říci, že činnost procesu se skládá z částí, kdy pracuje s vlastními daty a z částí, kdy přistupuje ke sdíleným datům

Kritická sekce

Proces, který chce do kritické sekce musí počkat, až z ní jiný proces vystoupí

Pravidla pro řešení časového souběhu (!)

1. Vzájemné vyloučení - žádné dva procesy nesmějí být současně uvnitř své kritické sekce
2. Proces běžící mimo kritickou sekci nesmí blokovat jiné procesy (např. jim bránit ve vstupu do kritické sekce)
3. Žádný proces nesmí na vstup do své kritické sekce čekat nekonečně dlouho (jiný vstupuje opakovaně, neumí se dohodnout v konečném čase, kdo vstoupí první)

Možnosti řešení

- Zákaz přerušení
- Aktivní čekání
- Zablokování procesu

Zákaz přerušení

- vadí přeplánování procesů
  - výsledek přerušení v systémech se sdílením času
- zákaz přerušení –> k přepínání nedochází
  - zakaž přerušení;
  - kritická sekce;
  - povol přerušení;

Zákaz přerušení II.

- nejjednodušší řešení – na uniprocesoru (1 CPU)
- není dovoleno v uživatelském režimu (jinak by uživatel zakázal přerušení a už třeba nepovolil...)
- používáno často uvnitř jádra OS
- ale není vhodné pro uživatelské procesy

Aktivní čekání - předpoklady

- zápis a čtení ze společné datové oblasti jsou nedělitelné operace
  - současný přístup více procesů ke stejné oblasti povede k sekvenčním odkazům v neznámém pořadí
  - platí pro data <= délce slova
- kritické sekce nemohou mít přiřazeny prioritu
- relativní rychlost procesů je neznámá
- proces se může pozastavit mimo kritickou sekci

Algoritmus 1 – procesy přistupují střídavě

```
program striktni_stridani;                    := přiřazení (Java =)
var turn: integer;                            =  porovnání (Java ==)
begin
   turn := 1;
   cobegin
        P1: while true do
          begin
               while turn = 2 do;     // čekací smyčka
                     KS;              // kritická sekce
               turn := 2              // a může druhý
          end
```

Algoritmus 1 pokračování

```
||

  P2: while true do
    begin
         while turn = 1 do;     // čekací smyčka
               KS;              // kritická sekce
         turn:= 1               // a může první
    end
coend
end
```

Algoritmus 1

- Problém – porušuje pravidlo 2
- Pokud je jeden proces podstatně rychlejší, nemůže vstoupit do kritické sekce 2x za sebou

Aktivní čekání

- Aktivní čekání
  - Průběžné testování proměnné ve smyčce, dokud nenabude očekávanou hodnotu
- Většinou se snažíme vyhnout
  - plýtvá časem CPU
- Používá se, pokud předpokládáme krátké čekání
  - spin lock

Algoritmus - Peterson

- První úplné řešení navrhl Dekker, ale je poměrně složité
- Jednodušší a elegantnější algoritmus navrhl Peterson (1981)
  - viz dále řešení pro 2 procesy
  - lze i zobecnit

Peterson – enter_CS()

```
program petersonovo_reseni;
var turn: integer;
interested: array [0..1] of boolean;        // na začátku {false, false}

procedure enter_CS(process: integer);
var other: integer;
begin
    other:=1-process;                       // ten druhý proces
    interested[process]:=true;              // oznámí zájem o vstup
    turn:=process;                          // nastaví příznak
    while turn=process and interested[other]=true do ;
end;
```

Peterson – leave_CS()

```
procedure leave_CS(process: integer);
    begin
        interested[process]:=false;         // oznámí odchod z KS
end;
```

Peterson – použití enter_CS() a leave_CS()

```
begin
    interested[0]:=false; // inicializace
    interested[1]:=false;
    cobegin
        while true do {cyklus - vlákno 1}
            begin
                enter_CS(0);
                KS1;
                leave_CS(0);
            end {while}
    || {vlákno 2 analogické}
    coend
end.
```

Z funkce enter_CS se vrátí až tehdy, když je kritická sekce volná !

Zavoláním leave_CS dáme najevo, že kritická sekce končí a dovnitř může někdo vejít

Peterson - vysvětlení

while turn=process and interested[other]=true do ;

Pokud chce do KS pouze jeden z procesů:

interested[other] bude false, a smyčka končí

Pokud chtějí do KS oba dva:

rozhoduje první část turn == process
turn bude vždy mít hodnotu 0, nebo 1, nic jiného
jeden z procesů skončí čekací smyčku

Peterson – vysvětlení podrobněji

- na začátku není v KS žádný proces
- první proces volá enter_CS(0)
  - interested[0] := true; turn := 0;
  - nebude čekat ve smyčce, interested[1] je false
- nyní proces 2 volá enter_CS(1)
  - interested[1]:= true; turn := 1;
  - čeká ve smyčce, dokud interested[0] nebude false (leave_CS)
- pokud oba volají enter_CS téměř současně...
  - oba nastaví interested na true
  - oba nastaví turn na své číslo ALE provede se sekvenčně, 0 OR 1
  - např. druhý proces bude jako druhý ☺, tedy turn bude 1
  - oba se dostanou do while, první proces projde, druhý čeká

Spin lock s instrukcí TSL (!!)

- hw podpora:
- většina počítačů – instrukci, která otestuje hodnotu a nastaví paměťové místo v jedné nedělitelné operaci
- operace Test and Set Lock – TSL, TS:
- TSL R, lock
  - LD R, lock
  - LD lock, 1
  - R je registr CPU
  - lock – buňka paměti, 0 false nebo 1 true; boolean;

TSL

- Provádí se nedělitelně (atomicky) – žádný proces nemůže k proměnné lock přistoupit do skončení TSL
- Multiprocesor – zamkne paměťovou sběrnici po dobu provádění instrukce

TSL - použití

- Proměnná typu zámek – na počátku 0
- Proces, který chce vstoupit do KS – test
  - Pokud 0, nastaví na 1 a vstoupí do KS
  - Pokud 1, čeká
- Pokud by TSL nebyla atomická
  - Jeden proces přečte, vidí 0 .. Přeplánování..
  - Druhý proces přečte, vidí 0, nastaví 1, vstoupí KS
  - První proces naplánován, zapíše 1 a je také v KS

Implementace zámku

```
Spin_lock:
        TSL R, lock     ;; atomicky R=lock, lock=1
        CMP R, 0        ;; byla v lock 0?
        JNE spin_lock   ;; pokud ne cykluj dál
        RET             ;; návrat, vstup do KS
Spin_unlock:
        LD lock, 0      ;; ulož hodnotu 0 do lock
        RET
```

Implementace zámku – pozn.

- Cyklus přes návěští spin_lock dokud lock je 1
- Když někdo jiný vyvolá spin_unlock, přečte 0 a může vstoupit do KS
- Pokud na vstup do KS čeká více procesů
  - Hodnotu 0 přečte jenom jeden z nich (první kdo vykoná TSL)

Implementace – jádro Linuxu

```
spin_lock:
        TSL R, lock
        CMP R, 0        ;; byla v lock 0 ?
        JE cont         ;; pokud byla, skočíme
Loop:   CMP lock, 0     ;; je lock 0 ?
        JNE loop        ;; pokud ne, skočíme
        JMP spin_lock   ;; pokud ano, skočíme
Cont:   RET             ;; návrat, vstup do KS
```

Náhrada TSL

- **Uniprocesor**
  - Nedělitelnost zakázáním přerušení (DI/EI, CLI/STI)
- **Multiprocesor**
  - Primitivní operace s uzamčením sběrnice
- Př. I8086:
  - MOV AL, 1 ; do AL 1
  - LOCK XCHG AL, X ; zamkne sběrnici pro XCHG
    ; zamění AL a X

TSL – v pseudokódu

```
atomic function TSL (var x: boolean) : boolean;
begin
   TSL := x;
   x := true;
end;
```

Instrukci TSL si můžeme namodelovat s využitím atomické funkce (provede se nedělitelně)

Implementace spin-locku

```
type lock = boolean;

procedure spin_lock (var m: lock);
begin
  while TSL(m) do ;     {čeká dokud je m true}
end;
```

Implementace spin-locku

```
procedure spin_unlock (var m: lock);
begin
   m := false;
end;
```

Pozn. V literatuře TSL někdy se nastavuje true, někdy false; chce to předem znát sémantiku

Problém řešení s aktivním čekáním

- Peterson, spin-lock
- Ztracený čas CPU
  - Jeden proces v KS, další může ve smyčce přistupovat ke společným proměnným – krade paměťové cykly aktivnímu procesu
- Problém inverze priorit
  - Pouze zde připustíme, že procesy mají prioritu
  - Dva procesy, jeden s vysokou H a druhý s nízkou L prioritou, H se spustí jakmile připraven

Problémy akt. čekání, problém: inverze priorit

- L je v kritické sekci
- H se stane připravený (např. má vstup)
- H začne aktivní čekání
- L ale nebude už nikdy naplánován, nemá šanci dokončit KS
- H bude aktivně čekat do nekonečna

- **Problém inverze priorit**

Řešení problémů s akt. čekáním

- hledala se primitiva, která proces zablokují, místo aby čekal aktivně

Semafory (!!)

- Dijkstra (1965) navrh primitivum, které zjednodušuje komunikaci a synchronizaci procesů – semafory
- Semafor – **proměnná**, obsahuje nezáporné celé číslo
- Semaforu lze přiřadit hodnotu pouze při deklaraci
- Nad semafory pouze **operace** P(s) a V(s)

Struktura semaforu (!!)

```
typedef struct {
   int hodnota;
   process_queue *fronta;
} semaphore;
```

hodnota semaforu: 0, 1, 2, ..

fronta procesů čekajících na daný semafor

Operace P(!!)

Operace P(S):

```
if S > 0
   S--;
else
  zablokuj_proces;
```

zablokuje proces, který chtěl provést operaci P:

- přidá jej do fronty procesů čekajících na daný semafor
- stav procesu označí jako blokovaný

Operace V(!!)

Operace V(S):

```
if (proces_blokovany_nad_semaforem)
    jeden_proces_vzbud;
else
   s++;
```

podívá se, zda je fronta prázdná či ne

označí stav procesu jako připravený vyjme proces z fronty na semafor

(Pokud je nad semaforem S zablokovaný jeden nebo více procesů, vzbudí jeden z procesů; proces pro vzbuzení je vybrán náhodně)

Pamatuj

Semafor je tvořen celočíselnou proměnnou s a frontou procesů, které čekají na semafor, a jsou nad ním implementovány operace P() a V()

s může nabývat hodnot 0, 1, 2, ..

Hodnota 0 znamená, že je semafor zablokovaný, a prvolání, operace P() se daný proces zablokuje

Nenulová hodnota s znamená, kolik procesů může zavolat operaci P(), aniž by došlo k jejich zablokování

Pro vzájemné vyloučení je tedy počáteční hodnotu s potřeba nastavit na 1, aby operaci P() bez zablokování mohl vykonat jeden proces

Poznámky

- Operace P a V jsou nedělitelné (atomické) akce
  - Jakmile začne operace nad semaforem – nikdo k němu nemůže přistoupit dokud operace neskončí nebo se nezablokuje
- Několik procesů současně ke stejnému semaforu
  - Operace se provede sekvenčně v libovolném pořadí

Poznámky - terminologie

- V literatuře P(s) někdy wait(s) nebo down(s)
- V(s) nazýváno signal(s) nebo up(s)
- Původní označení z holandštiny
- P proberen – otestovat
- V verhogen – zvětšit
- *Pomůcka – např. abecední pořadí operací*

Vzájemné vyloučení – pomocí semaforů

- Vytvořit semafor s hodnotou 1
- Před vstupem do KS – P(s)
- Po dokončení KS – V(s)

P(s); … KS … ; V(s);

- Je-li libovolný proces v KS
  - Potom S je 0, jinak S je 1

Vzájemné vyloučení

Do kritické sekce smí vstoupit pouze 1 proces současně

Na počátku je vstup do kritické sekce volný

Vzájemné vyloučení (!!!)

```
var s: semaphore = 1;
cobegin
   while true do
   begin
         ...
         P(s);
         KS1;
         V(s);
         ...
   end
|| {totéž dělá další proces}
coend
```

Na začátku je vstup do kritické sekce volný, tedy hodnota semaforu 1

Z funkce P(s) se vrátíme, až když je vstup do kritické sekce volný

Zavoláním V(s) signalizujeme, že je kritická sekce nyní volná a dovnitř může někdo další

Otázky k uvedenému příkladu

Co kdybychom na začátku semafor špatně inicializovali na hodnotu 2?

Co kdybychom na začátku semafor špatně inicializovali na hodnotu 0?

Co kdybychom zapomněli po dokončení kritické sekce zavolat V(s) ?

Co kdybychom před vykonáním kritické sekce nezavolali operaci P(s)?

Použití semaforů

Binární semafor může nabývat jen hodnot 0 a 1

- Vzájemné vyloučení
  - **Mutexy**, binární semafory .. 0 a 1
- Kooperace procesů
  - Problém omezených zdrojů (např. velikost bufferu)
  - Obecné semafory .. 0, 1, 2 ..

Pro vzájemné vyloučení můžeme samozřejmě využít obecný semafor, binární nám navíc může ohlídat, že hodnoty budou jen 0 a 1

Vzájemné vyloučení - KS

- Procesy soupeří o zdroj
- Ke zdroji může chtít přistupovat víc než 1 proces v daném čase
- Každý proces může existovat bez ostatních
- Interakce POUZE pro zajištění serializace přístupu ke zdroji

Kooperace procesů

- Procesy se navzájem **potřebují**, potřeba vzájemné výměny informací
- Nejjednodušší případ – pouze synchronizační signály
- Obvykle – i další informace – např. zasíláním zpráv

Producent – konzument (!!!)

Producent – konzument je jedna ze základních synchronizačních úloh z teorie OS.

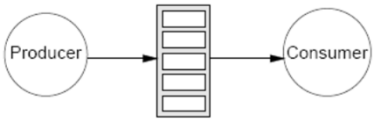
Cílem je správně synchronizovat přístup k sdílenému bufferu omezené velikosti – ošetřit mezní stavy, kdy je prázdný a naopak plný.

Měli byste umět v obecné podobě tuto úlohu vyřešit s využitím tří semaforů.

Problém producent-konzument

- Problém ohraničené vyrovnávací paměti (bounded buffer problem, Dijkstra 1968)
- Dva procesy společnou paměť (buffer) pevné velikosti N položek
- Jeden proces – producent generuje nové položky a ukládá do vyrov. paměti
- Paralelně konzument – data vyjímá a spotřebovává

Producent - konzument



Př. Hlavní program tiskne x tiskový server, blok – 1 stránka

Př. Obslužný prog. čte data ze zařízení x hlavní program je zpracovává

Různé rychlosti procesů

- Procesy – různé rychlosti – zabezpečit, aby nedošlo k přetečení / podtečení
- Konzument musí být schopen **čekat** na producenta, nejsou-li data
- Producent – **čekat** na konzumenta, je-li buffer plný

Prod-konz. pomocí semaforů

- Pro synchronizaci obou procesů
- Pro vzájemné vyloučení nad KS
- Proces se zablokuje P, jiný ho vzbudí V
- Semafory:
- **e** – počet prázdných položek v bufferu dostupných producentovi (empty)
- **f** – počet plných položek ještě nespotřebovaných konz. (full)

Třetí semafor

- Semafor **m** pro vzájemné vyloučení
- Přidávání a vybírání ze společné paměti může být obecně kritickou sekcí

P&K - implementace

```
Var
   e: semaphore = N;        // prázdných
   f: semaphore =  0;       // plných
   m: semaphore = 1;        // mutex
```

P&K – implementace II. (!)

```
cobegin
   while true do { producent}
   begin
      produkuj záznam;
      P(e);                 // je volná položka?
      P(m); vlož do bufferu; V(m);
      V(f);                 // zvětší obsazených
   end {while}
```

Není-li volná položka v bufferu, zablokuje se

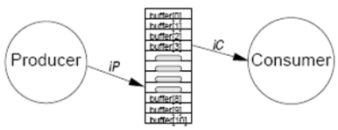
P&K – implementace III.

```
||
  while true do { konzument }
  begin
        P(f);                 // je plná nějaká položka?
        P(m); vyber z bufferu; V(m);
        V(e);                 // zvětši počet prázdných
        zpracuj záznam;
  end {while}
coend.
```

Pokud je buffer prázdný, zablokuje se

P&K poznámky

- Vyrovnávací paměť se často implementuje jako pole – buffer [0..N-1]



P&K poznámky

- Oba procesy – vlastní index do pole buffer
- Např. operace přidej do bufferu:
- *buffer[iP]:=polozka; iP:=(iP+1) mod N;*

- Pokud je buffer jako pole, vzájemné vyloučení pro přístup dvou procesů nebude potřebné, každý přistupuje pouze k těm, ke kterým má přístup dovolen operací V(s)

Literatura

obrázek Solaris LWP procesy

z knížky

W.Stalling: Operating systems – Internals and design Principles

04. Mutexy, monitory

ZOS 2013, L. Pešička

Semafory

- Ošetření kritické sekce
  - ukázka – více nezávislých kritických sekcí
- Synchronizace: Producent – konzument
  - možnost procesu zastavit se a čekat na událost
  - 2 události
    - buffer prázdný – čeká konzument
    - buffer plný – čeká producent
  - „uspání procesu“ – operace semaforu P

Problém spícího holiče – jen zadání (The barbershop problem)

Holičství

- čekárna s N křesly a holičské křeslo
- žádný zákazník – holič spí
- zákazník vstoupí
  - všechna křesla obsazena – odejde
  - holič spí – vzbudí ho
  - křesla volná – sedne si na jedno z volných křesel

Napsat program, koordinující činnost holiče a zákazníků

Je celá řada podobných synchronizačních úloh, cílem je pomocí synchronizačních mechanismů ošetřit úlohu, aby fungovala korektně a nedocházelo např. k vyhladovění …

Literatura pro samostatnou práci

The Little Book of Semaphores
(Allen B. Downey)

kniha volně dostupná na netu:
http://greenteapress.com/semaphores/

Serializace: událost A se musí stát před událostí B
Vzájemné vyloučení: události A a B se nesmí stát ve stejný čas

Mutexy, monitory

- Mutexy, implementace
- Implementace semaforů
- Problémy se semafory
- Monitory
  - Různé reakce na signal
- Implementace monitorů

Obsah další části přednášky

Mutexy

- Potřeba zajistit vzájemné vyloučení
  - chceme „spin-lock“ bez aktivního čekání
  - nepotřebujeme obecně schopnost semaforů čítat
- mutex – mutual exclusion
  - paměťový zámek

Mutex řeší vzájemné vyloučení a je k systému šetrnější než čistě aktivní čekání spin-lock, můžeme jej naimplementovat např. pomocí TSL instrukce a volání yield

Implementace mutexu – podpora jádra OS

Oblíbená instrukce TSL

```
mutex_lock:TSL R, mutex     ;;  R:=mutex a mutex:=1
CMP R, 0                    ;; byla v mutex hodnota 0?
JE ok                       ;; pokud byla na OK
CALL yield                  ;; vzdáme se procesoru -
                               naplánuje se jiné vlákno
JMP mutex_lock              ;; zkusíme znovu, pozdějí
ok: RET

mutex_unlock:
LD mutex, 0                 ;; ulož  0 do  mutex
RET
```

Vzdát se CPU

Implementace mutexu – volání yield

Šetří CPU

- volající se dobrovolně vzdává procesoru ve prospěch jiných procesů (vláken,…)
- jádro OS přesune proces mezi připravené a časem ho opět naplánuje
- použití – např. vzájemné vyloučení mezi vlákny stejného procesu
- lze implementovat jako knihovnu prog. jazyka

Moderní OS – semafory, mutexy

Moderní OS nám dávají k dispozici určitou množinu synchronizačních nástrojů, z nichž si programátor vybírá

- obecné (čítající) semafory
  - obecnost
  - i pro řešení problémů meziprocesové komunikace
- mutexy
  - paměťové zámky, binární semafory
  - pouze pro vzájemné vyloučení
  - při vhodné implementaci efektivnější

Spin-lock (aktivní čekání)

- spin-lock – vhodný, pokud je čekání krátké a procesy běží paralelně
- není vhodné pro použití v aplikacích
  - aplikace – doba čekání se může velmi lišit
- obvykle se používá uvnitř jádra OS, v knihovnách, ...

když víme, že budeme čekat jen krátce

Mutex x binární semafor

- Společné – použití pro vzájemné vyloučení
- Často se v literatuře mezi nimi příliš nerozlišuje
- Někdy jsou zdůrazněny rozdíly

uvědomte si, kdy nám toto může vadit

Mutex

s koncepcí vlastnictví:

Odemknout mutex může jen stejné vlákno/proces, který jej zamkl (!)

pamatovat si co znamená pojem mutex s koncepcí vlastnictví

Renetrantní mutex

- Stejné vlákno může získat několikrát zámek
- Stejně tolikrát jej musí zas odemknout, aby mohlo mutex získat jiné vlákno
- Viz: http://en.wikipedia.org/wiki/Reentrant_mutex

Na ukázku různých variant: reentrantní mutex, futex, …

Futex

- Userspace mutex, v Linuxu
- V kernel space: wait queue (fronta)
- V user space: integer (celé číslo, zámek)

Vždy se řeší otázka rychlosti, ceny

Systémové volání je obvykle nákladná záležitost, proto snaha minimalizovat jejich počet

- Vlákna/procesy mohou operovat nad číslem v userspacu s využitím atomických operací a systémové volání (které je drahé) jen pokud je třeba manipulovat s frontou čekajících procesů (vzbudit čekající proces, dát proces do fronty čekajících)
- Viz http://en.wikipedia.org/wiki/Futex

Dále bude ukázána obecná implementace semaforu a implementace semaforu s využitím mutexu

Implementace semaforu obecná – datové struktury

```
typedef struct  {
        int value;                  // hodnota semaforu
        struct process *list;       // fronta zablokovaných
}                                   // procesů
```

Zatímco předpokládáme, že hodnota semoforu je >= 0 pro vnitřní implementaci můžeme připustit i záporné hodnoty (udávají počet blokovaných procesů)

Implementace semaforu obecná - P

```
P (semaphore s) {
   s.value--;
   if (s.value < 0)
        blokuj(s.list);
}
```

blokuj – zablokuje volající proces, zařadí jej do fronty čekajících na daný semafor s.list

Implementace semaforu obecná - V

```
V (semaphore s) {
   s.value++;
   if (s.value <= 0)
      if (s.list != NULL) {   // někdo spí nad S
        vyjmi_z_fronty(p);
        vzbud (p);            // blokovaný -> připrav.
   }
}
```

Semafory implementace s využitím mutexu

- S každým semaforem je sdruženo:
- celočíselná proměnná **s.c**
  - pokud může nabývat i záporné hodnoty
  - |s.c| vyjadřuje počet blokovaných procesů
- binární semafor **s.mutex**
  - vzájemné vyloučení při operacích nad semaforem
- seznam blokovaných procesů **s.L**

Seznam blok. procesů

- Proces, který nemůže dokončit operaci P bude zablokován a uložen do seznamu procesů s.L blokovaných na semaforu s
- Pokud při operaci V není seznam prázdný
  - vybere ze seznamu jeden proces a odblokuje se

Uložení datové struktury semafor

- semafory v jádře OS
  - přístup pomocí služeb systému
- semafory ve sdílené paměti

Popis implementace

```
type semaphore = record
   m: mutex;        // mutex pro přístup k semaforu
   c: integer;      // hodnota semaforu
   L: seznam procesu
end
```

Popis implement. – operace P

```
P(s): mutex_lock(s.m);
   s.c := s.c - 1;
   if s.c < 0 then
        begin
              zařad' volající proces do seznamu s.L;
              označ volající proces jako "BLOKOVANÝ";
              naplánuj některý připravený proces;
              mutex_unlock(s.m);
              přepni kontext na naplánovaný proces
        end
   else
        mutex_unlock(s.m);
```

Popis implement. – operace V

```
V(s): mutex_lock(s.m);
        s.c := s.c + 1;
        if s.c <= 0 then
          begin
              vyber a vyjmi proces ze sez. s.L;
              odblokuj vybraný proces
          end;
      mutex_unlock(s.m);
```

Popis implementace

- Pseudokód
- Skutečná implementace řeší i další detaily
  - Organizace datových struktur
    - Pole, seznamy, …
  - Kontrola chyb
    - Např. jeli při operaci V záporné s.c a přitom s.L je prázdné

Popis implementace

- Implementace v jádře OS
  - Obvykle používá aktivní čekání (spin-lock nad s.mutex)
  - Pouze po dobu operace nad obecným semaforem – max. desitky instrukcí - efektivní

Mutexy vs. semafory

- Mutexy – vzájemné vyloučení vláken v jednom procesu
  - Např. knihovní funkce
  - Často běží v uživatelském režimu
- Obecné semafory – synchronizace mezi procesy
  - Implementuje jádro OS
  - Běží v režimu jádra
  - Přístup k vnitřním datovým strukturám OS

Problémy se semafory

- primitiva P a V – použita kdekoliv v programu
- Snadno se udělá chyba
  - Není možné automaticky kontrolovat při překladu

Chyby – přehození P a V

Přehození P a V operací nad mutexem:

1. V()
2. kritická sekce
3. P()

Důsledek – více procesů může vykonávat kritickou sekci současně

Chyby – dvě operace P

1. P()
2. Kritická sekce
3. P()

Důsledek - deadlock

Chyby – vynechání P, V

- Proces vynechá P()
- Proces vynechá V()
- Vynechá obě

Důsledek – porušení vzájemného vyloučení nebo deadlock

Monitory

- Snaha najít primitiva vyšší úrovně, která zabrání části potenciálních chyb
- Hoare (1974) a Hansen (1973) nezávisle na sobě navrhli vysokoúrovňové synchronizační primitivum nazývané monitor
- Odlišnosti v obou návrzích

Monitor

- Monitor – na rozdíl od semaforů – jazyková konstrukce
- Speciální typ modulu, sdružuje data a procedury, které s nimi mohou manipulovat
- Procesy mohou volat proceduru monitoru, ale nemohou přistupovat přímo k datům monitoru

Monitor

- V monitoru může být v jednu chvíli AKTIVNÍ pouze jeden proces
- Ostatní procesy jsou při pokusu o vstup do monitoru pozastaveny

Terminologie OOP

- Snaha chápat kritickou sekci jako přístup ke sdílenému objektu
- Přístup k objektu pouze pomocí určených operací – metod
- Při přístupu k objektu vzájemné vyloučení, přístup po jednom

Monitory

příklad se vztahuje k syntaxi Pascalu, tak jak monitor implementoval např. sw Baci

- Monitor – Pascalský blok podobný proceduře nebo funkci
- Uvnitř monitoru definovány proměnné, procedury a funkce
- Proměnné monitoru – nejsou viditelné zvenčí
  - Dostupné pouze procedurám a funkcím monitoru
- Procedury a funkce – viditelné a volatelné vně monitoru

Příklad monitoru

```
monitor m;
   var proměnné ...
       podmínky ...

   procedure p; { procedura uvnitř monitoru }
   begin
       ...
   end;
begin
   inicializace;
end;
```

Příklad

- Použití pro vzájemné vyloučení

```
monitor m;              // příklad – vzájemné vyloučení
   var x: integer;
   procedure inc_x;                 { zvětší x }
   begin
        x:=x+1;
   end;
   function get_x: integer;         { vrací x }
   begin
        get_x:=x
   end
begin
   x:=0
end; { inicializace x };
```

Problém dosavadní definice

- Výše uvedená definice (částečná) – dostačuje pro vzájemné vyloučení
- ALE nikoliv pro synchronizaci – např. řešení producent/konzument
- Potřebujeme mechanismus, umožňující procesu se pozastavit a tím uvolnit vstup do monitoru
- S tímto mechanismem jsou monitory úplné

Synchronizace procesů v monitoru

- Monitory – speciální typ proměnné nazývané podmínka (condition variable)
- Podmínky
  - definovány a použity pouze uvnitř monitoru
  - Nejsou proměnné v klasickém smyslu, neobsahují hodnotu
  - Spíše odkaz na určitou událost nebo stav výpočtu (mělo by se odrážet v názvu podmínky)
  - Představují frontu procesů, které na danou podmínku čekají

Operace nad podmínkami

- Definovány 2 operace – wait a signal

C.wait

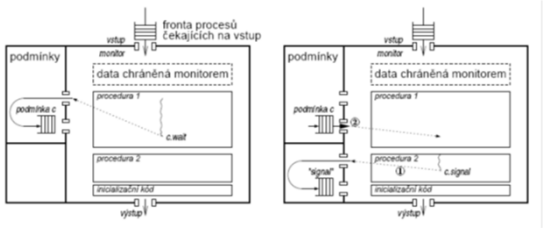
- Volající bude pozastaven nad podmínkou C
- Pokud je některý proces připraven vstoupit do monitoru, bude mu to dovoleno

Operace nad podmínkami

C.signal

- Pokud existuje 1 a více procesů pozastavených nad podmínkou C, reaktivuje jeden z pozastavených procesů, tj. bude mu dovoleno pokračovat v běhu uvnitř monitoru
- Pokud nad podmínkou nespí žádný proces, nedělá nic ☺
  - Rozdíl oproti semaforové operaci V(sem), která si “zapamatuje”, že byla zavolána

Schéma monitoru



Problém s operací signal

- Pokud by signál pouze vzbudil proces, běžely by v monitoru dva
  - Vzbuzený proces
  - A proces co zavolal signal
- ROZPOR s definicí monitoru
  - V monitoru může být v jednu chvíli aktivní pouze jeden proces
- Několik řešení

Řešení reakce na signal

- Hoare
  - proces volající c.signal se pozastaví
  - Vzbudí se až poté co předchozí rektivovaný proces opustí monitor nebo se pozastaví
- Hansen
  - Signal smí být uveden pouze jako **poslední** příkaz v monitoru
  - Po volání signal musí proces opustit monitor

Jak je to v BACI?

- Monitory podle Hoara
- Waitc (cond: condition)
- Signalc (cond: condition)
  - semantika dle Hoara
- Waitc (cond: condition, prio: integer)
  - Čekajícímu je možné přiřadit prioritu
  - Vzbuzen bude ten s nejvyšší prioritou
    Nejvyšší priorita – nejnižší číslo prio

Monitory v jazyce Java

- Existují i jiné varianty monitorů, např. zjednodušené monitory s primitivy wait a notify v jazyce Java a dalších
- S každým objektem je sdružen monitor, může být i prázdný
- Metoda nebo blok patřící do monitoru označena klíčovým slovem synchronized

Monitory - Java

```
class jméno {
   synchronized void metoda() {
        ....
   }
}
```

Monitory - Java

- S monitorem je sdružena jedna podmínka, metody:
- wait() – pozastaví volající vlákno
- notify() – označí jedno spící vlákno pro vzbuzení, vzbudí se, až volající opustí monitor (x c.signal, které pozastaví volajícího)
- notifyAll() – jako notify(), ale označí pro vzbuzení všechna spící vlákna

Monitory - Java

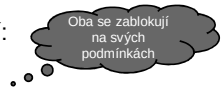
- Pozn. Jde vlastně o třetí řešení problému, jak ošetřit volání signal
- Čekající může běžet až poté, co proces volající signál opustí monitor

Monitory Java – více podmínek

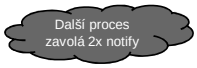
- Více podmínek, může nastat následujcií (x od Hoarovských monitorů)
- Pokud se proces pozastaví, protože proměnná B byla false, nemůže počítat s tím, že po vzbuzení bude B=true

Více podmínek - příklad

- 2 procesy, nastalo zablokování:
  - P1: if not B1 then c1.wait;
  - P2: if not B2 then c2.wait;



- Proces např. P3 běžící v monitoru způsobí splnění obou podmínek a oznámí to pomocí
  - If B1 then c1.notify;
  - If B2 then c2.notify;



Více podmínek - příklad

- Po opuštění monitoru se vzbudí P1
- Proces1 způsobí, že B2=false
- Po vzbuzení P2 bude B2 false, i když by logicky předpokládal, že tomu tak není
- Volání metody wait by mělo být v cyklu (x od Hoarovskych)
- While not B do c.wait;

Java – volatile proměnné

- poznámka
- Vlákno v Javě si může vytvořit soukromou pracovní kopii sdílené proměnné
- Zapíše zpět do sdílené paměti pouze při vstupu/výstupu z monitoru
- Pokud chceme zapisovat proměnnou při každém přístupu – deklarovat jako volatile

Shrnutí - monitory

- Základní varianta – Hoarovské monitory
- V reálných prog. jazycích varianty
  - Prioritní wait (např. BACI)
  - Primitiva wait a notify (Java, Borland Delpi)
- Výhoda monitorů
  - Automaticky řeší vzájemné vyloučení
  - Větší odolnost proti chybám programátora
- Nevýhoda
  - Monitory – koncepce programovacího jazyka, překladač je musí umět rozpoznat a implementovat

Práce s vlákny v C

- Některé myšlenky i v rozhraní LINUXu pro práci s vlákny
- Úsek kódu ohraničený
  - pthread_mutex_lock(m) .. pthread_mutex_unlock(m)
- Uvnitř lze používat obdobu podmínek z monitorů

Práce s vlákny v C

- pthread_cond_wait(c, m) - atomicky odemkne m a čeká na podmínku
- pthread_cond_signal(c) - označí 1 vlákno spící nad c pro vzbuzení
- pthread_cond_broadcast(c) - označí všechna vlákna spící nad c pro vzbuzení

Řešení producent/konzument pomocí monitoru

```
Monitor ProducerConsumer
var
   f, e: condition;
   i: integer;
```

```
procedure enter;
begin
if i=N then wait(f);        {pamět je plná, čekám }
enter_item;                 { vlož položku do bufferu }
i:=i+1;
if i=1 then signal(e);      { prvnií položka => vzbudím konz. }
end;

procedure remove;
begin
if i=0 then wait(e);        { pamět  je prázdná => čekám }
remove_item;                { vyjmi položku z bufferu }
i:=i-1;
if i=N-1 then signal(f);    { je zase místo }
end;
```

Inicializační sekce

```
begin
i:=0; { inicializace }
end
end monitor;
```

{ A vlastní použití monitoru dále: }

```
begin                          // začátek programu
   cobegin
        while true do     { producent}
        begin
          produkuj zaznam;
          ProducerConsumer.enter;
        end {while}
        ||
        while true do     { konzument }
        begin
          ProducerConsumer.remove;
          zpracuj zaznam;
        end {while}
   coend
end.
```

Implementace monitorů pomocí semaforů

- Monitory musí umět rozpoznat překladač programovacího jazyka
- Přeloží je do odpovídajícího kódu
- Pokud např. OS poskytuje semafory může je využít pro implementaci monitoru

Co musí implementace zaručit

1. Běh procesů v monitoru musí být vzájemně vyloučen (pouze 1 aktivní v monitoru)
2. Wait musí blokovat aktivní proces v příslušné podmínce
3. Když proces opustí monitor, nebo je blokován podmínkou AND existuje >1 procesů čekajících na vstup do monitoru
   => musí být jeden z nich vybrán

Implementace monitoru

- Existuje-li proces pozastavený jako výsledek operace signal, pak je vybrán
- Jinak je vybrán jeden z procesů čekajících na vstup do monitoru

4. Signal musí zjistit, zda existuje proces čekající nad podmínkou
   - Ano –aktuální proces pozastaven a jeden z čekajících reaktivován
   - Ne – pokračuje původní proces

Implementace monitoru

```
Semafory
   m = 1;        // chrání přístup do monitoru
   u = 0;        // pozastavení procesu při signal()
  w[i] = 0;      // pozastavení při wait()
                 // pole t semaforů, kolik je podmínek
Čítače
   ucnt = 0;     // počet pozastavení pomocí signal
   wcnt[i]       // počet pozastavených na dané
                 // podmínce voláním wait
```

Vstup do monitoru, výstup z monitoru

Každý proces vykoná následující kód

```
P(m);                  // vstup – zamkne semafor
…                      // tělo procedury v monitoru
                       // výstupní kód
if ucnt > 0 then       // byl někdo zablokovaný
   V(u);               //že volal signal? Ano – pustíme ho
else                   // jinak pustíme další
   V(m);               // proces do monitoru
```

Implementace volání c.wait()

```
wcnt [i] = wcnt [i] + 1;
if ucnt > 0 then          // někdo bude pokračovat
   V(u);                  // blokovaný na signál
else                      // nebo ze vstupu
   V(m);
P(w[i]);                  // čekáme na podmínce
wcnt [i] = wcnt [i] - 1;  // čekání skončilo
```

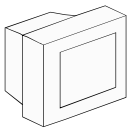
Implementace volání c.signal()

```
ucnt = ucnt + 1;
If wcnt [i] > 0 then      // někdo čekal nad ci
begin
   V(w[i]);               // pustíme čekajícího
   P(u);                  // sami čekáme
end;
ucnt = ucnt-1;            // čekání skončilo
```

05. Meziprocesová komunikace – doplnění - zprávy, RPC

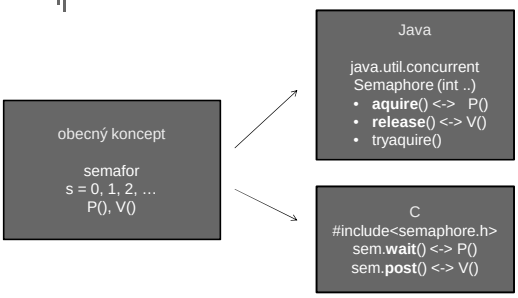
ZOS 2013

Monitory – opakování



- Kolik procesů může být najednou v monitoru?
- Kolik **aktivních** procesů může být najednou v monitoru?
- Co je to podmínková proměnná?
- Čím se liší následující sémantiky volání signal?
  - Hoare
  - Hansen
  - Java
- Musím uvnitř monitoru dávat pozor na současný přístup k proměnným monitoru?

Semafor
obecný koncept, programovací jazyk



Java semafory
(vybrané operace)

dokumentace:
http://docs.oracle.com/javase/7/docs/api/java/util/concurrent/Semaphore.html

java.util.concurrent.Semaphore

funkce	popis
public Semaphore(int permits)	Vytvoří semafor inicializovaný na hodnotu permits
acquire()	Operace P() nad semaforem
release()	Operace V() nad semaforem
tryAcquire()	Neblokující pokus o P()

C semafory
(Posixové semafory)

```
#include <semaphore.h>
sem_t s;
```

Funkce	popis
sem_init(&s, 0, **1**);	Inicializuje semafor na hodnotu **1** prostřední hodnota říká: 0 – semafor mezi vlákny 1 – semafor mezi procesy
sem_wait(&s);	Operace P() nad semaforem
sem_post(&s);	Operace V() nad semaforem
sem_destroy(&s);	Zrušení semaforu

System V semafory

- pro doplnění
- alokují se, používají, ruší podobně jako sdílená paměť
- semafor je zde pole čítačů (dle alokace)
- semget()
- semctl()
- semop()

C Mutex

funkce	popis
pthread_mutex_t m = PTHREAD_MUTEX_INITIALIZER;	Inicializace mutextu Implicitně je odemčený
pthread_mutex_**destroy** (&m)	Zrušení mutexu
pthread_mutex_**lock** (&m)	Pokusí se zamknout mutex. Pokud je mutex již zamčený, je volající vlákno zablokováno.
pthread_mutex_**unlock** (&m)	Odemkne mutex
pthread_mutex_**trylock** (&m)	Pokusí se zamknout mutex. Pokud je mutex již zamčený, vrátí se okamžitě s kódem EBUSY

C Mutex - příklad

```
#include <pthread.h>
pthread_mutex_t  mutex = PTHREAD_MUTEX_INITIALIZER;
int x;

void inkrementuj() {
  pthread_mutex_lock (&mutex);
       x++;    /* kritická sekce */
  pthread_mutex_unlock (&mutex);
}
```

Ukázky programů

Courseware KIV/ZOS
=> Cvičení
=> Materiály ke cvičení
=> C, Java příklady

Opakování

Kde je uložený PID?
V PCB.

- proces má PID, vlákno má TID
  - proces může mít více vláken
  - každé vlákno PC (CS:IP) – ukazuje, jaká instrukce se má vykonat („program counter“)
- hierarchie procesů
  - proces si udržuje info o rodiči PPID - getppid()
- **proces – jednotkou přidělování prostředků**
- **vlákno – jednotkou plánování**

Opakování

fork() – vytvoří duplicitní kopii aktuálního procesu

exec() – nahradí program v aktuálním procesu jiným programem

wait() – rodič může čekat na dokončení potomka

strace – jaká systémová volání proces volá (!!)

strace ls je.txt neni.txt

- bude nás zajímat program ls
- vidíme volání execve(“/bin/ls”, …)
- vidíme chybový výstup write(2, …)
- vidím stand. výstup write(1, …)

Meziprocesová komunikace

- Předávání zpráv
- Primitiva send, receive
- Mailbox, port
- RPC
- Ekvivalence semaforů, zpráv, …
- Bariéra, problém večeřících filozofů

Meziprocesová komunikace

Procesy mohou komunikovat:

- Přes sdílenou paměť
  (předpoklad: procesy na stejném uzlu)

- Zasíláním zpráv
  (na stejném uzlu i na různých uzlech)

Linux - signály

Signály představují jednu z forem meziprocesové komunikace

- signál – speciální zpráva zaslaná jádrem OS procesu
- iniciátorem signálu může být i proces
- zpráva neobsahuje jinou informaci než číslo signálu
- jsou asynchronní - mohou přijít kdykoliv, ihned jej proces obslouží (přeruší provádění kódu a začne obsluhovat signál)
- Signál je specifikován svým číslem, často se používají symbolická jména (SIGTERM, SIGKILL, …)
- Používají s v Linuxu, nejsou ve Windows v této podobě

Linux - signály

příkaz	popis
ps aux	Informace o procesech
kill -9 1234	Pošle signál č. 9 (KILL) procesu s PID číslem 1234
man kill	Nápověda k signálům
man 7 signal	Nápověda k signálům
kill -l	Vypíše seznam signálů

Linux - signály

man 7 signal

Události generující signály

- Stisk kláves (*CTLR+C* generuje *SIGINT*)
- HW přerušení (dělení nulou)
- Příkaz kill (1), systémové volání kill (2)
- Mohou je generovat uživatelské programy – kill (2)

Reakce na signály

- Standardní zpracování
- Vlastní zpracování naší funkcí
- Ignorování signálu (ale např. SIGKILL, SIGSTOP nelze)

```
#!/bin/bash
obsluha() {
 echo "Koncim..."
 exit 1
}
# pri zachyceni signalu SIGINT se vykona funkce: obsluha
trap obsluha INT

SEC=0
while true ; do
 sleep 1
 SEC=$((SEC+1))
 echo "Jsem PID $$, ziju $SEC"
done
# sem nikdy nedojdeme
exit 0
```

Skript zareaguje na Ctrl+C zavoláním funkce obluha()
Příkaz trap definuje jaká funkce se pro obsluhu daného signálu zavolá

Linux – využití signálu při ukončení práce OS

Vypnutí počítače:
INIT: Sending all processes the TERM signal
INIT: Sending all processes the KILL signal

Proces init pošle všem podřízeným signál TERM
=> tím žádá procesy o ukončení a dává jim čas učinit tak korektně
Po nějaké době pošle signál KILL, který nelze ignorovat a způsobí ukončení procesu.

Přehled vybraných signálů

SIGTERM	Žádost o ukončení procesu
SIGSEGV	Porušení segmentace paměti
SIGABORT	Přerušení procesu
SIGHUP	Odříznutí od terminálu (nohup ignoru.)
SIGKILL	Bezpodmínečné zrušení procesu
SIQQUIT	Ukončení terminálové relace procesu
SIGINT	Přerušení terminálu (CTRL+C)
SIGILL	Neplatná instrukce
SIGCONT	Navrácení z pozastavení procesu
SIGSTOP	Pozastavení procesu (CTRL+Z)
SIGINT	Ukončení procesu na popředí

Manuálové stránky: man 7 signal

Datové roury

- jednosměrná komunikace mezi 2 procesy
- data zapisována do roury jedním procesem lze dalším hned číst
- data čtena přesně v tom pořadí, v jakém byla zapsána

cat /etc/passwd | grep josef | wc -l

Datové roury

systémové volání pipe:

int pipe (int filedes[2])
fides[0] .. odsud čteme
fides[1] .. sem zapisujeme

Problém sdílené paměti

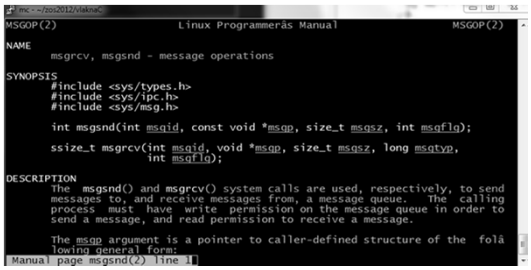
- Vyžaduje umístění objektu ve sdílené paměti
- Někdy není **vhodné**
  - Bezpečnost – globální data přístupná kterémukoliv procesu bez ohledu na semafor
- Někdy není **možné**
  - Procesy běží na různých strojích, komunikují spolu po síti
- Řešení – předávaní zpráv

Předávání zpráv – send, receive

Zavedeme 2 primitiva:
- send (adresát, zpráva) - odeslání zprávy
- receive(odesilatel, zpráva) - příjem zprávy

- Send
  - Zpráva (libovolný datový objekt) bude zaslána adresátovi
- Receive
  - Příjem zprávy od určeného odesilatele
  - Přijatá zpráva se uloží do proměnné (dat.struktury) „zpráva“

Jak Linux?



Vlastnosti

- synchronizace (blokující – neblokující)
- unicast, multicast, broadcast
- přímá komunikace x nepřímá komunikace
- délka fronty zpráv
- pevná vs. proměnná délka zprávy

Primitiva send a receive mohou mít celou řadu rozličných vlastností. V dalších slidech budou postupně rozebrány

Synchronizace

- blokující (synchronní)
- neblokující (asynchronní)
  - čeká send na převzetí zprávy příjemcem?
  - co když při receive není žádná zpráva?
- většinou send neblokující, receive blokující

Send - receive

- blokující send
  - čeká na převzetí správy příjemcem
- **neblokující send**
  - vrací se ihned po odeslání zprávy
  - většina systémů
- **blokující receive**
  - není-li ve frontě žádná zpráva, zablokuje se
  - většina systémů
- neblokující receive
  - není-li zpráva, vrací chybu

Receive s omezeným čekáním

- receive (odesilatel, zprava, t)
  - čeká na příchod zprávy dobu t
  - pokud zpráva nepřijde, vrací se volání s chybou

Další možná varianta

Adresování

- send 1 příjemce nebo skupina?
  - Pošleme jednomu nebo více příjemcům
- receive 1 odesílatel nebo různí?
  - Přijmeme pouze od jednoho odesilatele nebo od kohokoliv

Skupinové a všesměrové adresování

- skupinové adresování (multicast)
  - zprávu pošleme skupině procesů
  - zprávu obdrží každý proces ve skupině
- všesměrové vysílání (broadcast)
  - zprávu posíláme "všem" procesům
  - tj. více nespecifikovaným příjemcům
- pozn.: další varianta - anycast (IPv6)

Poznámky

Většina systémů umožňuje:

- odeslání zprávy skupině procesů
- příjem zprávy od kteréhokoliv procesu

Další otázky

- vlastnosti fronty zpráv
  - kolik jich může obsahovat, je omezená?
- pokus odeslat zprávu a fronta zpráv plná?
  - většinou odesilatel pozastaven
- v jakém pořadí jsou zprávy doručeny?
  - většinou v pořadí FIFO
- jaké je zpoždění mezi odesláním zprávy a možností zprávu přijmout?
- jaké mohou v systému nastat chyby, např. mohou se zprávy ztrácet?

Délka fronty zpráv (buffering)

- nulová délka
  žádná zpráva nemůže čekat
  odesílatel se zablokuje – "randezvous"
- omezená kapacita
  blokování při dosažení kapacity
- neomezená kapacita
  odesilatel se nikdy nezablokuje

Poznámka

- Volbu konkrétního chování primitiv send a receive provádějí návrháři operačního systému
- Některé systémy nabízejí několik alternativních primitiv send a recive s různým chováním

Terminologická poznámka

neblokující send

- v některých systémech send, který se vrací ihned – ještě před odesláním zprávy
- odeslání se provádí paralelně s další činností procesu
- používá se zřídka

Předpoklady pro další text

- send je neblokující, receive blokující
- receive – umožnuje příjem od libovolného adresáta – receive(ANY,zpráva)
- fronta zpráv – dostatečně velká na všechny potřebné zprávy
- zprávy doručeny v pořadí FIFO a neztrácejí se

Producent – konzument pomocí zpráv

- symetrický problém
- producent generuje plné položky
  - pro využití konzumentem
- konzument generuje prázdné položky
  - pro využití producentem

Úlohu producent/konzument jsme řešili s využitím semaforů, monitorů, nyní tedy i zasíláním zpráv

```
cobegin
   while true do            { producent }
   begin
        produkuj záznam;
        receive(konzument, m);
                // čeká na prázdnou položku
        m := záznam;
                // vytvoří zprávu
        send(konzument, m);
                // pošle položku konzumentovi
   end {while}
||
```

Blokující operace

```
for i:=1 to N do            { inicializace }
   send(producent, e);
        // pošleme N prázdných položek
while true do               { konzument }
begin
   receive(producent, m);
        // přijme zprávu obsahující data
   záznam := m;
   send(producent, e);
        // prázdnou položku pošleme zpět
   zpracuj záznam;
end {while}
coend.
```

Blokující operace

Komunikující procesy

procesy nemusejí být na stejném stroji, ale mohou komunikovat po síti

Problém určení adresáta

- dosud – zprávy posíláme procesům
- jak určit adresáta, pojmenovat procesy
- procesy nejsou trvalé entity
  - v systému vznikají a zanikají
- nebo více instancí stejného programu
- řešení – adresujeme frontu zpráv
  - nepřímá komunikace

Chtěli bychom např. poslat zprávu webovému serveru *Apache*, ale při každém spuštění stroje bude mít jiný PID

Neadresujeme proces, ale frontu zpráv

Adresování fronty zpráv

- proces pošle zprávu
  - zpráva se připojí k určené frontě zpráv (vložení zprávy do fronty)
- jiný proces přijme zprávu
  - vyjme zprávu z dané fronty

Mailbox, port

Termíny používané v teorii OS, neplést s pojmem mailbox jak jej běžně znáte

- mailbox
  - fronta zpráv využívaná více odesílateli a příjemci
  - obecné schéma
  - operace receive – drahá, zvláště pokud procesy běží na různých strojích
- port
  - omezená forma mailboxu
  - zprávy může vybírat pouze jeden příjemce

Mailbox, port

Implementace mechanismu zpráv – další problémy

- problémy, které nejsou u semaforů ani monitorů, zvláště při komunikaci po síti
- ztráta zprávy
  - potvrzení o přijetí (acknowledgement)
  - pokud vysílač nedostane potvrzení do nějakého časového okamžiku (timeout), zprávu pošle znovu
- ztráta potvrzení
  - zpráva dojde ok, ztratí se potvrzení
  - číslování zpráv, duplicitní zprávy se ignorují

Problém autentizace

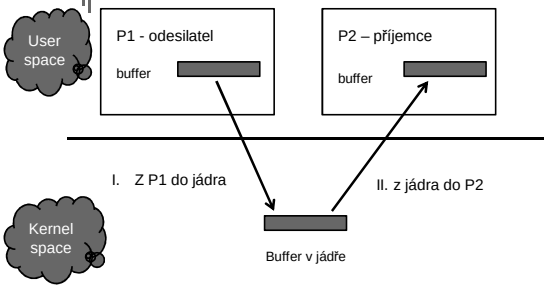
- problém autentizace
  - ověřit, že nekomunikují s podvodníkem
  - zprávy je možné šifrovat
  - klíč známý pouze autorizovaným uživatelům (procesům)
  - zašifrovaná zpráva obsahuje redundanci – umožní detekovat změnu zašifrované zprávy
  - Pozn. Symetrické a asymetrické šifrování, podpisy zpráv

Lokální komunikace (!)

Na stejném stroji – snížení režie na zprávy

- Dvojí kopírování (!)
  - z procesu odesílatele do fronty v jádře
  - z jádra do procesu příjemce
- rendezvous
  - eliminuje frontu zpráv
  - send zavolán dříve než receive – odesílatel zablokován
  - vyvolán send i receive – zprávu zkopírovat z odesílatele přímo do příjemce
  - efektivnější, ale méně obecné (např. jazyk ADA)

Dvojí kopírování (!!!)



Lokální komunikace II.

- využití mechanismu virtuální paměti (!)
  - další možnost, lze využít např. při randevous
  - paměť obsahující zprávu je přemapována
    - z procesu odesílatele
    - do procesu příjemce
  - zpráva se nekopíruje
    - Virtuální paměť umí "čarovat" komu daný kus fyzické paměti namapuje a na jaké adresy

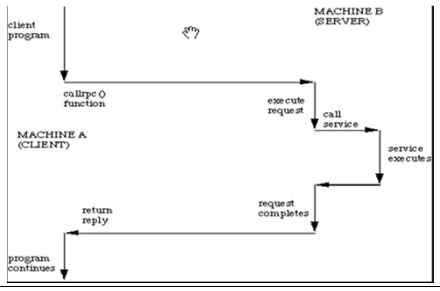
Typické aplikace

- struktura klient – server
  - klient požaduje vykonat práci
  - server práci vykonává
- klienti vyžadují od serveru data nebo provedení operace nad daty
- WWW klient x WWW server (Apache, IIS)
- účetní aplikace x databázový systém (Oracle, MySQL)
- klientský OS x souborový server (AFS)

Volání vzdálených procedur (RPC)

- používání send receive – opět nestrukturované
- Birell a Nelson (1984)
- dovolit procesům (klientům) volat procedury umístěné v jiném procesu (serveru)
- mechanismus RPC (Remote Procedure Call)
- variantou RPC je i volání vzdálených metod (Remote Method Invocation, RMI) v OOP, např. Javě
- snaha aby co nejvíce připomínalo lokální volání

RPC



spojka klienta, serveru

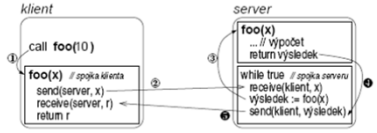
- klientský program sestaven s knihovní funkcí, tzv. spojka klienta (client stub)
  - reprezentuje vzdálenou proceduru v adresním prostoru klienta
  - stejné jméno, počet a typ argumentů jako vzdálená procedura
- program serveru sestaven se spojkou serveru (server stub)
- spojky zakrývají, že volání není lokální

Kroky komunikace

1. Klient zavolá spojku klienta, reprezentující vzdálenou proceduru
2. Spojková procedura argumenty zabalí do zprávy, pošle ji serveru
3. Spojka serveru zprávu přijme, vezme argumenty a zavolá proceduru
4. Procedura se vrátí, návrat. hodnotu pošle spojka serveru zpět klientovi
5. Spojka klienta přijme zprávu obsahující návrat. hodnotu a předá ji volajícímu

Příklad

- volání foo(x: integer): integer



1. Klient volá spojku klienta foo(x) s argumentem x=10.
2. Spojka klienta vytvoří zprávu a pošle jí serveru:
   procedure foo(x: integer):integer;
   begin
   send(server, m); // zpráva obsahuje argument, tj. hodnotu "10"
3. Server přijme zprávu a volá vzdálenou proceduru:
   receive(klient, x); // spojka přijme zprávu, tj. hodnotu "10"
   vysledek = foo(x); // spojka volá fci foo(10)
4. Procedura foo(x) provede výpočet a vrátí výsledek.
5. Spojka serveru výsledek zabalí do zprávy a pošle zpět spojce klienta:
   send(klient, výsledek);
6. Spojka klienta výsledek přijme, vrátí ho volajícímu (jako kdyby ho spočetla sama):
   receive(server, výsledek);
   foo = výsledek;
   return;

RPC – více procedur

- rozlišeny číslem
- spojka klienta ve zprávě předá kromě parametrů i číslo požadované procedury

Na serveru:
while true do begin
receive(klient, m); // zpráva obsahující č. procedury a parametry
if (m.číslo_procedury == 1) then výsledek = foo(m.x);
if (m. číslo_procedury == 2) then výsledek = bar(m.x);
...
send(klient, výsledek); // odešli zpět návratovou hodnotu
end

RPC

dnes nejpoužívanější jazyková konstrukce pro implementaci distribuovaných systémů a programů bez explicitního předávání zpráv

- DCE RPC, Java RMI, CORBA

Programování RPC

- Jazyk IDL (Interface Definition Language)
  - Definujeme rozhraní mezi klientem a serverem (datové typy, procedury)
- Kompilátor jazyka IDL
  - Vygeneruje spojky pro klienta i server
- Server sestavíme se spojkou serveru
- Spojka klienta
  - Podoba knihovny
  - Sestavujeme s ní klientské programy

Problémy RPC

Volání vzdálené procedury přináší problémy, které při lokálním přístupu nejsou

- Parametry předávané odkazem
  - Klient a server – různé adresní prostory
  - Odeslání ukazatele nemá smysl
  - Pro jednoduchý datový typ, záznam, pole – trik
    - Spojka klienta pošle odkazovaná data spojce serveru
    - Spojka serveru vytvoří nový odkaz na data atd.
    - Modifikovaná data pošle zpátky na klienta
    - Spojka klienta přepíše původní data
- Globální proměnné
  - Použití není možné x lokálních procedur

Reprezentace informace

- Společný problém pro předávání zpráv i RPC
- Stroje různé architektury
  - Může se lišit vnitřní reprezentace datových typů
  - Kódování řetězců
    - Udaná délka nebo ukončovací znak
    - Kódování jednotlivých znaků
  - Numerické typy
    - Způsob uložení (little endian, big endian)
    - Velikost (integer 32 nebo 64 bitů)

Problém může představovat komunikace mezi heterogenní mi systémy

Little & big endian

- Chceme uložit: 4a3b2c1d (32bit integer)
- Big endian
  - Nejvýznamnější byte (MSB) na nejnižší adrese
  - Motorola 68000, SPARC, System/370
  - V paměti od nejnižší adresy: 4a, 3b, 2c, 1d
- Little endian
  - Nejméně významný byte (LSB) na nejnižší adrese
  - **Intel x86**, DEC VAX
  - V paměti od nejnižší adresy: 1d, 2c, 3b, 4a

Další varianty endians (jen pro doplnění)

- Bi-endian
  - Lze nastavit (např. mode bit), jaký formát uložení se bude používat
  - Např. IA-64, defaultně little-endian
- Middle-endian
  - Starší formát, jen poznámka
  - Např. 3b, 4a, 1d, 2c

Otestování sw (jen ukázka)

```
#define LITTLE_ENDIAN   0
#define BIG_ENDIAN      1

int machineEndianness() {
    short s = 0x0102;
    char *p = (char *) &s;
    if (p[0] == 0x02)
        return LITTLE_ENDIAN;
    else
        return BIG_ENDIAN; }
```

Řešení portability

- Definovat, jak budou data reprezentována při přenosu mezi počítači – síťový formát
  - Před odesláním do síťového formátu
  - Po přijetí do lokálního formátu
- Problém rozdílné velikosti
  - Nová množina numerických typů, stejná velikost na všech podporovaných architekturách
  - Int32_t – integer 32bitů, <stdint.h>, ISO C99

Síťový formát (jen pro doplnění)

- TCP/IP
  - Network byte order (big endian)
  - Celé číslo – nejvýznamější byte jako první (MSB)
- Konverzní funkce např. <netinet/in.h>
- htonl, htons
- ntohl, ntohs
- ("host to net, net to host, short/long)

Sémantika volání RPC

- lokální volání fce – právě jednou
- vzdálené volání
  - chyba při síťovém přenosu (tam, zpět)
  - chyba při zpracování požadavku na serveru
  - klient neví, která z těchto chyb nastala
  - volající havaruje po odeslání zprávy před získáním výsledku

Sémantika volání RPC

- právě jednou
- alespoň 1x
  - opakované volání po timeoutu
  - dle charakteru operace
- nejvýše 1x
  - klient volání neopakuje
  - při timeoutu – chyba, ošetření výjimek

Idempotentní operace

- operace, kterou lze opakovat se stejným efektem, jaký mělo její první provedení
- pro sémantiku alespoň 1x
- x = x + 10 vs. x = 20
- vypinac (zapni), vypinac (vypni) x vypinac (prepni)

Ekvivalenty uvedených primitiv

- Lze implementovat semafory pomocí zpráv a zprávy pomocí semaforů, …, ?
- Tj. má obojí stejnou vyjadřovací sílu?

Zprávy pomocí semaforů

- Využijeme řešení problému producent-konzument
- send: Vložení zprávy do bufferu
- receive: Vyjmutí zprávy z bufferu

Semafor pomocí zpráv

Semafory pomocí zpráv

- Pomocný synchronizační proces (SynchP)
  - Pro každý semafor udržuje čítač (hodnotu semaforu)
  - A seznam blokovaných procesů
- Operace P a V
  - Jako funkce, které provedou odeslání požadavku
  - Poté čekají na odpověď pomocí receive
- SynchP – v jednom čase jeden požadavek
  - Tím zajištěno vzájemné vyloučení

Semafor pomocí zpráv

- Pokud SynP obdrží požadavek na operaci P
  - a čítač semafor > 0, odpoví ihned
  - Jinak neodpoví – čímž volajícího zablokuje
- Pokud SyncP obdrží požadavek na operaci V
  - A je blokovaný proces
    - Jednomu blokovanému odpoví, čímž ho vzbudí

Stejná vyjadřovací síla

- Lze ukázat, že je možné implementovat
  - Semafory pomocí monitoru
  - Monitory pomocí semaforů
- Všechna dříve uvedená primitiva mají stejnou vyjadřovací sílu
- Platí to i o mutexech? Ano, někdy..

Semafor pomocí mutexů

- Např. Barz (1983)

```
type semaphore = record
                    val: integer;
                    m: mutex;        // pro vzájemné vyloučení
                    d: mutex;        // pro blokování (delay)
                 end;
procedure Initsem(var s: semaphore, count: integer);
begin
   s.val:=count;
   s.m :=ODEM;                 // odemčeno
if count=0 then s.d := ZAM     // zamčeno, někdo musí provést V(s)
   else s.d := ODEM            // odemčeno
end;
```

```
procedure P(var s: semaphore);
begin
   mutex_lock(s.d);  // když s.val=0, čekáme
   mutex_lock(s.m); // kritická sekce –
                    // přístup k s.val
   s.val := s.val - 1;     // vždy bude platit s.val>=0
   if s.val > 0 then
      mutex_unlock(s.d); // další proces do operace P
   mutex_unlock(s.m)     // konec přístupu k val
end;
```

```
procedure V(var s: semaphore);
begin
   mutex_lock(s.m);
   s.val := s.val + 1;
   if s.val = 1 then
        mutex_unlock(s.d)
   mutex_unlock(s.m)
end;
```

Omezení mutexů v některých implementacích

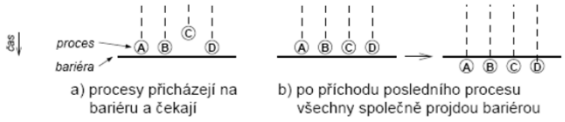
Z důvodů efektivity někdy omezení mutexů:

- Mutex smí odemknout pouze vlákno, které předtím provedlo jeho uzamknutí (POSIX.1)
  - Nelze použít pro implementaci obecných semaforů
  - Potom slabší než výše uvedená primitiva

Bariéry

- Synchronizační mechanismus pro skupiny procesů
- Použití ve vědecko-technických výpočtech
- Aplikace – skládá se z fází
  - Žádný proces nesmí do následující fáze dokud všechny procesy nedokončily fázi předchozí
- Na konci každé fáze – synchronizace na bariéře
  - Volajícího pozastaví
  - Dokud všechny procesy také nezavolají barrier
- Všechny procesy opustí bariéru současně

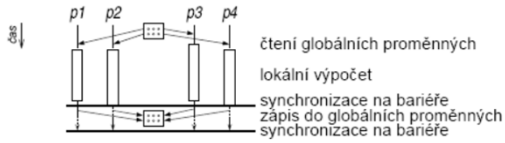
Bariéra



Bariéra – iterační výpočty

- Jednotlivé kroky výpočtu
- Matice X(i+1) z matice X(i)
- Každý proces počítá 1 prvek nové matice
- Synchronizace pomocí bariery

Bariera – iterační výpočty



Klasické problémy IPC

IPC – Interprocess Communication

- Producent- konzument
- Večeřící filozofové
- Čtenáři - písaři
- Spící holič
- Řada dalších úloh

Vždy když někdo vymyslí nový synchronizační mechanismus, otestuje se jeho použitelnost na těchto klasických úlohách

Problém večeřících filozofů

- Dijkstra 1965, dining philosophers
- Model procesů soupeřících o výhradní přístup k omezenému počtu zdrojů
  - Může dojít k zablokování, vyhladovění
- „Test elegance“ nových synchronizačních primitiv

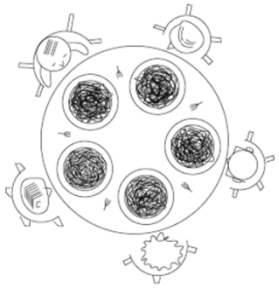
Zablokování a vyhladovění jsou dva rozdílné pojmy, uvidíme dále..

Problém večeřících filozofů

- 5 filozofů sedí kolem kulatého stolu
- Každý filozof má před sebou talíř se špagetami
- Mezi každými dvěma talíři je vidlička
- Filozof potřebuje dvě vidličky, aby mohl jíst

Ukázka:
http://webplaza.pt.lu/~onarat/

Problém večeřících filozofů



Problém večeřících filozofů

- Život filozofa – jí a přemýšlí
  - Tyto fáze se u každého z nich střídají
- Když dostane hlad
  - Pokusí se vzít si dvě vidličky
    - Uspěje – nějakou dobu jí, pak položí vidličky a pokračuje v přemýšlení
- Úkolem
  - Napsat program pro každého filozofa, aby pracoval dle předpokladů a nedošlo k potížím
  - aby se každý najedl

Problém večeřících filozofů

Očíslujeme filozofy: 0 .. 4

Očíslujeme vidličky

filozof 0: levá vidlička 0, pravá 1
filozof 1: levá vidlička 1, pravá 2
…

Procedura zvedni(v)

- Počká, až bude vidlička k dispozici a pak jí zvedne

Kód filozofa - chybný

```
const N = 5;
procedure filosof(i: integer);
begin
    premyslej();
    zvedni (i);
    zvedni ((i+1) mod N);
    jez();
    poloz(i);
    poloz((i+1) mod N)
end;
```

Všichni filozofové běží dle stejného kódu, např. 5 vláken, každé vykonává kód:

filosof(0)
filosof(1)
…
filosof(4)

Problém uvíznutí (deadlock)

Popis chyby:

Všichni filozofové zvednou najednou levou vidličku, žádný z nich už nemůže pokračovat, dojde k deadlocku

Deadlock:

cyklické čekání dvou či více procesů na událost, kterou může vyvolat pouze některý z nich, nikdy k tomu však nedojde

f[0] čeká, až f[1] položí vidličku, f[1] čeká na f[2], 2-3, 3-4, 4-0 = cyklus

Modifikace algoritmu

Filozof zvedne levou vidličku a zjistí, zda je pravá vidlička dostupná.

Pokud není, položí levou, počká nějakou dobu a zkusí znovu.

Stále chybná …

Pokud by filozofové vzali najednou levou vidličku, budou běžet cyklicky (vidí, že pravá není volná, položí..)

Proces není blokovaný (x od deadlock), ale běží bez toho, že by vykonával užitečnou činnost

Analogie – situace „až po vás“ přednost ve dveřích

Vyhladovění (starvation)

proces se nedostane k požadovaným zdrojům

Řešení pomocí monitoru

Když chce i-tý filozof jíst, zavolá funkci chci_jist(i)

Když je najezen, zavolá mam_dost(i)

Ochrana před uvíznutím:

obě vidličky musí zvednout najednou, v kritické sekci, uvnitř monitoru

Řešení je vícero, následující např. nezabrání dvěma konspirujícím filozofům, aby bránili jíst třetímu

```
Monitor vecerici_filozofove;
const N=5;
var
   f : array [0 .. N-1] of integer;  {počet vidliček dostupných filozofovi}
   a : array [0 .. N-1] of integer;  {podmínka vidličky k dispozici}
   i : integer;

procedure chci_jist ( i: integer)
begin
   if f [ i ]  < 2 then a [ i ].wait;
   decrement ( f [ (i-1) mod N] );   { sniž o jedna vidličky levému  }
   decrement ( f [ (i+1) mod N] );   { sniž o jedna vidličky pravému}
end;
```

```
procedure  mam_dost ( i: integer)
begin
   increment( f [ (i-1) mod N] );    { zvětši o jedna vidličky levému  }
   increment ( f [ (i+1) mod N] );   { zvětši o jedna vidličky pravému}

   if   f [ (i-1) mod N] = 2 then      { má levej soused 2 vidličky? }
                  a [ (i-1) mod N] . signal;
   if   f [ (i+1) mod N] = 2 then      { má pravej soused 2 vidličky? }
                  a [ (i+1) mod N] . signal;
end;

begin                          { inicializace monitoru, filozof má 2 vidličky }
   for i:=0 to 4 do
         f[i] := 2
end;
```

06.
Čtenáři – písaři
Plánování procesů

ZOS 2013

1. zápočtový test

- 12. a 13. listopadu 2014
  - v čase cvičení (na poloviny dle domluvy na cvičení)
  - u školního PC pod speciálním účtem
  - 30 minut čistého času na test
  - otázka z 1.-4. prezentace přednášek
  - různé varianty testů
- Hodnocení
  - Každý úkol ANO/NE zisk bodů
  - nadpoloviční většina bodů

1. zápočtový test

- Pomůcky k dispozici
  - manuálové stránky (man)
  - dostanete vytištěný seznam základních příkazů (viz portál předmětu)
  - nic víc

tipy:
- syntaxe: help if, help case, help test, man test
- umět poznat, že je skript spuštěný s 0, 1, 2, .. parametry
- umět iterovat přes soubory (včetně adresářů) v daném adresáři

Semestrální práce

viz podmínky na
coursewaru -> Samostatná práce

Problém čtenářů a písařů

- modeluje přístup do databáze
- rezervační systém (místenky, letenky)

- množina procesů, souběžné čtení a zápis
  - souběžné čtení lze
  - výhradní zápis (žádný další čtenář ani písař)

Častá praktická úloha, lze realizovat s předností čtenářů, nebo s předností písařů.
Pro komerční aplikace je samozřejmě vhodnější přednost písařů.

```
var
   m=1 : semaphore;    {mutex, chrání čítač}
   w=1: semaphore;     {přístup pro zápis }
   rc = 0: integer;    { počet čtenářů }

procedure writer;
begin
   P(w);
        // zapisuj
   V(w)
end;
```

```
procedure reader;
begin
  P(m);
  rc := rc + 1;
  if rc = 1 then P(w);    //1. čtenář blok. písaře
  V(m);
      // čti
  P(m);
  rc := rc - 1;
  if rc=0 then V(w);      // poslední čtenář odblok. pí.
  V(m)
end;
```

Čtenáři – písaři popis

- čtenáři
  - první čtenář provede P(w)
  - další zvětšují čítač rc
  - po "přečtení" čtenáři zmenšují rc
  - poslední čtenář provede V(w)
- semafor w
  - zabrání vstupu písaře, jsou-li čtenáři
  - zabrání vstupu čtenářům při běhu písaře:
    - prvnímu zabrání P(w)
    - ostatním brání P(m)
- toto řešení je s předností čtenářů
  - písaři musí čekat, až všichni čtenáři skončí

Implementace zámků v operačních a databázových systémech

- přístup procesu k souboru nebo záznamu databázi
- výhradní zámek (pro zápis)
  - nikdo další nesmí přistupovat
- sdílený zámek (pro čtení)
  - mohou o něj žádat další procesy
- granularita zamykání
  - celý soubor x část souboru
  - tabulka x řádka v tabulce

Implementace zámků v OS

Linux, UNIX lze zamknout část souboru funkcí

fcntl (fd, F_SETLK, struct flock)

```
int fd;
struct flock fl;
fd = open("testfile", O_RDWR);
fl.l_type = F_WRLCK;          - zámek pro zápis
fl.l_whence = SEEK_SET;       - pozice od začátku souboru
fl.l_start = 100; fl,l_len = 10;  - pozice, kolik
fcntl (fd,F_SETLK, &fl);      - zamkneme pro zápis
 // vrací -1 pokud se nepovede
```

Implementace zámků v OS

odemknutí

```
fl.l_type = F_UNLCK;          - odemknutí
fl.l_whence = SEEK_SET;       - pozice od začátku souboru
fl.l_start = 100; fl,l_len = 10;  - pozice, kolik
fcntl (fd,F_SETLK, &fl);      - nastavíme
```

operace

```
F_SETLK     - set / clear lock, nečeká
F_GETLK     - info o zámku
F_SETLKW    - nastavení zámku, čeká
              když je zamčený
```

Implementace zámků v OS

- zámky jsou odstraněny, když proces skončí (teoreticky)
- zámky poradní (advisory)
  - nejsou vynucené
  - pro kooperující procesy
  - defaultní chování
- zámky mandatory
- různé způsoby zamykání:
  fcntl, flock

Zámky v DB systémech

např. s každým záznamem databáze sdružen zámek
funkce:

db_lock_r(x)	zámek pro čtení
db_lock_w(x)	uzamkne záznam x pro zápis
db_unlock_r(x)	odemčení záznamu x
db_unlock_w(x)	dtto

čtenáři – písaři s předností písařů

```
type zamek = record
 wc, rc: integer := 0;          // počet písařů a čtenářů
 mutw: semaphore := 1;          // chrání přístup k čítači wc
 mutr: semaphore := 1;          // chrání přístup k čítači rc
 wsem: semaphore := 1;          // blokování písařů
 rsem: semaphore := 1;          // blokuje 1. čtenáře (písař)
 rdel: semaphore := 1;          // blokování ostatních čtenářů
end;
```

Algoritmus je složitější, ale praktičtější
uveden jen na ukázku
:= symbol přiřazení, = symbol porovnání

```
procedure db_lock_w(var x: zamek);

   // uzamčení záznamu pro zápis
begin
   P(x.mutw);
   x.wc:=x.wc+1;
   if x.wc=1 then P(x.rsem);  // 1.písař zablokuje 1. čtenáře
   V(x.mutw);
   P(x.wsem);                 // blokování písařů
end;
```

```
procedure db_unlock_w(var x: zamek);

// odemčení zápisů pro zápis
// sníží počet písařů, poslední písař odblokuje čtenáře

begin
   V(x.wsem);                 // odblokování písařů
   P(x.mutw);
   x.wc:=x.wc-1;
   if x.wc=0 then V(x.rsem); // poslední písař pustí 1.čten.
   V(x.mutw)
end;
```

```
procedure db_lock_r(var x: zamek);
begin
    P(x.rdel);                // nejsou blokováni ostatní čtenáři
    P(x.rsem);                // není blokován 1. čtenář
    P(x.mutr);
    x.rc:=x.rc+1;
    if x.rc=1 then P(x.wsem);      // 1. čtenář zablokuje písaře
    V(x.mutr);
    V(x.rsem);
    V(x.rdel)
end;

procedure db_unlock_r(var x: zamek);
begin
    P(x.mutr);
    x.rc:=x.rc-1;
    if x.rc=0 then V(x.wsem);      // poslední čtenář odblokuje písaře
    V(x.mutr)
end;
```

Další problémy meziprocesové komunikace

- problém spícího holiče
- problém populárního pekaře (Lamport 1974)
  - Google: Lamport baker
  - Každý zákazník dostane unikátní číslo
- plánovač hlavičky disku

- další probrané
  - problém večeřících filozofů
  - producent – konzument
  - čtenáři – písaři
- Knížka *The Little Book of Semaphores* (zdarma pdf)

Plánování procesů

Základní stavy procesu
- běžící
- připraven – čeká na CPU
- blokován – čeká na zdroj nebo zprávu

- nový (new) – proces byl právě vytvořen
- ukončený (terminated) – proces byl ukončen

Správce procesů – udržuje tabulku procesů
Záznam o konkrétním procesu – PCB (Process Control Block) – souhrn dat potřebných k řízení procesů

opakování (!!)

v Linuxu je datová struktura task_struct, která obsahuje informace o procesu (tj. představuje PCB)

- každý proces má záznam (řádku) v **tabulce procesů**
- tomuto záznamu se říká **PCB** (process control block)

- PCB obsahuje všechny potřebné informace (tzv. kontext procesu) k tomu, abychom mohli proces kdykoliv pozastavit (odejmout mu procesor) a znovu jej od tohoto místa přerušení spustit (Program Counter: CS:EIP)

- proces po opětovném přidělení CPU pokračuje ve své činnosti, jako by k žádnému přerušení vykonávání jeho kódu nedošlo, je to z jeho pohledu transparentní

opakování (!!)

- kde leží tabulka procesů?
  v paměti RAM, je to datová struktura jádra OS
- kde leží informace o PIDu procesu?
  v tabulce procesů -> v PCB (řádce tabulky) tohoto procesu
- jak procesor ví, kterou instrukci procesu (vlákna) má vykonávat?
  podle program counteru (PC, typicky CS:EIP), ukazuje na oblast v paměti, kde leží vykonávaná instrukce; obsah CS:EIP, stejně jako dalších registrů je součástí PCB

opakování (!!)

- jak vytvořím nový proces?
  systémovým voláním fork()

- jak vytvořím nové vlákno?
  voláním pthread_create()

- jak spustím jiný program?
  systémovým voláním execve()
  začne vykonávat kód jiného programu v rámci existujícího procesu

Plánovač x dispatcher

- plánovač vs. dispatcher
- dispatcher předává řízení procesu vybranému short time plánovačem:
  - přepnutí kontextu
  - přepnutí do user modu
  - skok na vhodnou instrukci daného programu

- více připravených procesů k běhu – plánovač vybere, který spustí jako první
- plánovač procesů (scheduler), používá plánovací algoritmus (scheduling algorithm)

Pamatuj

Plánovač určí, který proces (vlákno) by měl běžet nyní.

Dispatcher provede vlastní přepnutí z aktuálního běžícího procesu na nově vybraný proces.

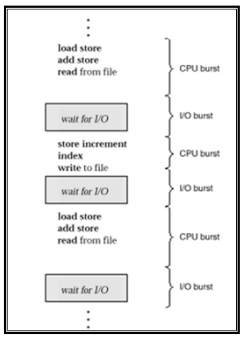
Plánování procesů - vývoj

- dávkové systémy
  - Úkol: spustit další úlohu, nechat ji běžet do konce
  - Uživatel s úlohou nekomunikuje, zadá program plus vstupní data např. v souboru
  - O výsledku je uživatel informován, např. e-mailem aj.
- systémy se sdílením času
  - Můžeme mít procesy běžící na pozadí
  - interaktivní procesy – komunikují s uživatelem
- kombinace obou systémů (dávky, interaktivní procesy)
- chceme: přednost interaktivních procesů
  - Srovnejte: odesílání pošty x zavírání okna

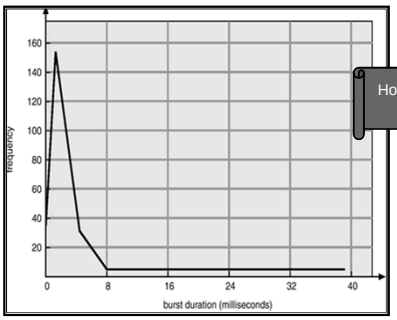
Střídání CPU a I/O aktivit procesu

Během vykonávání procesu
- CPU burst (vykonávání kódu)
- I/O burst (čekání)
- střídání těchto fází
- končí CPU burstem

Typicky máme:
hodně krátkých CPU burstů
málo dlouhých



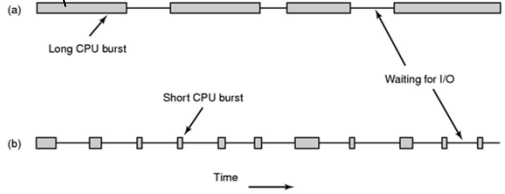
Histogram CPU burstů



Hodně krátkých burstů

Plánování

počítám



a) CPU-vázaný proces („hodně času tráví výpočtem“)
b) I/O vázaný proces („hodně času tráví čekáním na I/O“)

Uveďte příklady CPU vázaného a I/O vázaného procesu

Preemptivní vs. non-preemptivní plánování

- Non-preemptivní
  - každý proces dokončí svůj CPU burst (!!!!)
  - proces si podrží kontrolu nad CPU, dokud se jí nevzdá (I/O čekání, ukončení)
  - lze v dávkových systémech, není příliš vhodné pro time sharingové (se sdílením času)
  - Win 3.x non-preemptivní (kooperativní) plánování
  - od Win95 preemptivní
  - od Mac OS 8 preemptivní
  - na některých platformách je stále

Jaký má vliv non-preemptivnost systému na obsluhu kritické sekce u jednojádrového CPU?

Preemptivní vs. non-preemptivní plánování

- Preemptivní plánování
  - proces lze přerušit **KDYKOLIV** během CPU burstu a naplánovat jiný (-> problém kritických sekcí !!!)
  - dražší implementace kvůli přepínání procesů (režie)
  - Vyžaduje speciální hardware – timer (časovač)
    časovač je na základní desce počítače, pravidelně generuje hardwarová přerušení systému od časovače

Část výkonu systému spotřebuje režie nutná na přepínání procesů.
K přepnutí na jiný proces také může dojít v nevhodný čas (ošetření KS).
Preemptivnost je ale u současných systémů důležitá, pokud potřebujeme interaktivní odezvu systému.
Časovač tiká (generuje přerušení), a po určitém množství tiků se určí, zda procesu nevypršelo jeho časové kvantum.

Otázky preemptivní plánování

- Nutnost koordinovat přístup ke sdíleným datům
- preempce jádra OS
  - přeplánování ve chvíli, kdy se manipuluje s daty (I/O fronty) používanými jinými funkcemi jádra..
  - UNIX (když nepreemptivní)
    - čekání na dokončení systémového volání
    - nebo na dokončení I/O
    - výhodou jednoduchost jádra
    - nevýhodou výkon v RT a na multiprocesorech

Preempce se může týkat nejen uživatelských procesů, ale i jádra OS. Linux umožňuje zkompilovat preemptivní jádro.

Cíle plánování

All systems
Fairness - giving each process a fair share of the CPU
Policy enforcement - seeing that stated policy is carried out
Balance - keeping all parts of the system busy

Batch systems
Throughput - maximize jobs per hour
Turnaround time - minimize time between submission and termination
CPU utilization - keep the CPU busy all the time

Interactive systems
Response time - respond to requests quickly
Proportionality - meet users' expectations

Real-time systems
Meeting deadlines - avoid losing data
Predictability - avoid quality degradation in multimedia systems

Některé cíle jsou společné, jiné se liší dle typu systému

Zajímavosti

V roce 1973 provedli na MITu shut-down systému IBM 7094 a našli low priority proces, který nebyl dosud spuštěný a přitom byl založený …..

Zajímavosti

..v roce 1967 ..

Plánovač (!)

- rozhodovací mód
  - okamžik, kdy jsou vyhodnoceny priority procesu a vybrán proces pro běh
- prioritní funkce
  - určí prioritu procesu v systému
- rozhodovací pravidla
  - jak rozhodnout při stejné prioritě

Tři zásadní údaje, které charakterizují plánovač

Plánovač – **Rozhodovací mód**

- nepremptivní
  - Proces využívá CPU, dokud se jej sám nevzdá (např. I/O)
  - jednoduchá implementace
  - vhodné pro dávkové systémy
  - nevhodné pro interaktivní a RT systémy
- preemptivní
  - kdy ?
    - přijde nový proces (dávkové systémy)
    - periodicky – kvantum (interaktivní systémy)
    - jindy – priorita připraveného > běžícího (RT)
  - náklady
    - přepínání procesů, logika plánovače

Plánovač – **Prioritní funkce**

- Funkce, bere v úvahu parametry procesu a systémové parametry
- určuje prioritu procesu v systému
- externí priority
  - třídy uživatelů, systémové procesy
- priority odvozené z chování procesu (dlouho neběžel, čekal …)
- Většinou dvě složky – statická a dynamická priorita
  - Statická – přiřazena při startu procesu
  - Dynamická – dle chování procesu (dlouho čekal, aj.)

Prioritní funkce (!)

priorita = statická + dynamická

proč 2 složky?
pokud by chyběla:

- statická – nemohl by uživatel např. při startu označit proces jako důležitější než jiný
- dynamická – proces by mohl vyhladovět, mohl by být neustále předbíhán v plánování jinými procesy s větší prioritou

Plánovač – Prioritní funkce

Co všechno může vzít v úvahu prioritní funkce:

- čas, jak dlouho využíval CPU
- aktuální zatížení systému
- paměťové požadavky procesu
- čas, který strávil v systému
- celková doba provádění úlohy (limit)
- urgence (RT systémy)

Plánovač – **Rozhodovací pravidlo**

- malá pravděpodobnost stejné priority
  - náhodný výběr
- velká pravděpodobnost stejné priority
  - cyklické přidělování kvanta
  - chronologický výběr (FIFO)

Prioritní funkce může být navržena tak, že málokdy vygeneruje stejné priority, nebo naopak může být taková, že často (nebo když se nepoužívá vždy) určí stejnou hodnotu. Pak nastupuje rozhodovací pravidlo.

Cíle plánovacích algoritmů

Každý algoritmus nutně upřednostňuje nějakou třídu úloh na úkor ostatních.

- dávkové systémy
  - dlouhý čas, omezí ze přepínání úloh
- interaktivní systémy
  - Interakci s uživatelem, tj. I/O úlohy
- systémy reálného času
  - Dodržení deadlines

Společné cíle

- spravedlivost
  - srovnatelné procesy srovnatelně obsloužené
- vynucovat stanovená pravidla
- efektivně využít všechny části systému
- nízká režie plánování

Dávkové systémy (!)

- průchodnost (throughput)
  - počet úloh dokončených za časovou jednotku
- průměrná doba obrátky (turnaround time)
  - průměrná doba od zadání úlohy do systému do dokončení úlohy
- využití CPU

Průchodnost a průměrná doba obrátky jsou různé údaje ! Někdy snaha vylepšit jednu hodnotu může zhoršit druhou z nich.

Dávkové systémy

- maximalizace průchodnosti nemusí nutně minimalizovat dobu obrátky
- modelový příklad:
  - dlouhé úlohy následované krátkými
  - upřednostňování krátkých
  - bude tedy dobrá průchodnost
  - dlouhé úlohy se nevykonají
    - doba obrátky bude nekonečná

Interaktivní systémy

Minimalizace doby odpovědi

Vs.

Efektivita – drahé přepínání mezi procesy

Realtimové systémy

- Dodržení deadlines
- Předvídatelnost
  - Některé akce pravidelné (generování zvuku)

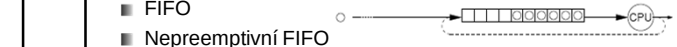
Př. Obsluha GSM části telefonu

Plánování úloh v dávkových systémech

- FCFS (First Come First Served)
- SJF (Shortest Job First)
- SRT (Shortest Remaining Time)
  - Preemptivní varianta SJF
- Multilevel Feedback

FCFS (First Come First Served)

- FIFO
- Nepreemptivní FIFO
- Základní varianta
  - Nově příchozí na konec fronty
  - Úloha běží dokud neskončí, poté vybrána další ve frontě



- Co když provádí I/O operaci?
  1. Zablokována, CPU se nevyužívá (základní varianta)
  2. Nebo se zablokuje, po dokončení I/O zařazena na **konec** fronty (častá varianta)
  - Vstupně výstupně vázané úlohy znevýhodněny před výpočetně vázanými
  - Další možná modifikace –> po dokončení I/O na začátek fronty

FCFS příklad

V čase O budou v systému procesy P1, P2, P3 přišlé v tomto pořadí.

proces	Doba trvání (s)
P1	15
P2	5
P3	4

doba obrátky: odešel - přišel

0 15 20 24

průměrná doba obrátky: (15+20+24) / 3 = 19,666

SJF (Shortest Job First)

- Nejkratší úloha jako první
- Předpoklad – známe přibližně dobu trvání úloh
- Nepreemptivní
  - Jedna fronta příchozích úloh
  - Plánovač vybere vždy úlohu s nejkratší dobou běhu
- Optimalizuje dobu obrátky

0 4 9 24

průměrná doba obrátky: (4+9+24) / 3 = 12,3

Výpočet průměrné doba obrátky

Do systému přijdou úlohy A,B,C,D s dobou běhu: 8, 4, 4, 4 minut.

FCFS

- Spustí v pořadí A, B, C, D dle strategie FCFS
- Doba obrátky:
  - A 8 minut
  - B 8+4 = 12 minut
  - C 8+4+4 = 16 minut
  - D 8+4+4 +4 = 20 minut
- Průměrná doba obrátky: (8+12+16+20) / 4 = 14 minut

Výpočet průměrné doby obrátky

- SJF
- V pořadí B, C, D, A
  - B 4 minuty
  - C 4+4 = 8 minut
  - D 4+4+4 = 12 minut
  - A 4+4+4+8 = 20 minut
- Průměrná doba obrátky (4+8+12+20) / 4 = 11 minut
- Průměrná doba obrátky je v tomto případě lepší

SRT (Shortest Remaining Time)

- Úlohy můžou přicházet **kdykoliv**
- Preemptivní (!)
  - Plánovač vždy vybere úlohu, jejíž **zbývající** doba běhu je nejkratší
- Př. preempce: Právě prováděné úloze zbývá 10 minut, do systému přijde úloha s dobou běhu 1 minutu – systém prováděnou úlohu pozastaví a nechá běžet novou úlohu
- Možnost vyhladovění dlouhých úloh (!) => předbíhány

SRT příklad

Čas příchodu	Název úlohy	Doba úlohy (s)
0	P1	7
0	P2	5
3	P3	1

V čase 0 máme na výběr P1, P2. Naplánujeme P2 s kratší dobou běhu

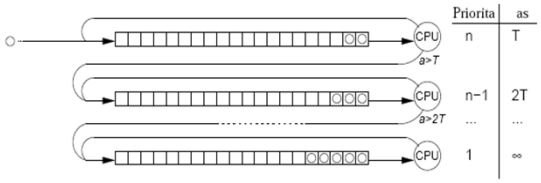
V čase 3 přijde do systému nová úloha. Zkontrolujeme zbývající doby běhu úloh: P1(7), P2 (2), P3(1). Naplánujeme P3.

Jakmile skončí P3, naplánujeme P2, až doběhne, P1

Multilevel feedback

- N prioritních úrovní
- Každá úroveň má svojí frontu úloh
- Úloha vstoupí do systému s nejvyšší prioritou (!)
- Na každé prioritní úrovni
  - Stanoveno maximum času CPU, který může úloha obdržet
  - Např.: T na úrovni n, 2T na úrovni n-1 atd.
  - Pokud úloha překročí tento limit, její priorita se sníží
  - Na nejnižší prioritní úrovni může úloha běžet neustále nebo lze překročení určitého času považovat za chybu
- Proces obsluhuje nejvyšší neprázdnou frontu (!!)

Multilevel feedback



Výhoda – rozlišuje mezi I/O-vázanými a CPU-vázanými úlohami
Upřednostňuje I/O vázané

Shrnutí – dávkové systémy

algoritmus	Rozh. mód	Prioritní funkce	Rozh. pravidlo
FCFS	Nepreemptivní	P(r) = r	Náhodně
SJF	Nepreemptivní	P(t) = -t	Náhodně
SRT	Preemptivní (při příchodu úlohy)	P(a,t) = a-t	FIFO nebo náhodně
MLF	nepreemptivní	Viz popis ☺	FIFO v rámci fronty

r celkový čas strávený úlohou v systému
t předpokládaná délka běhu úlohy
a čas strávený během úlohy v systému

07.
Plánování procesů
Deadlock

ZOS 2013, L. Pešička

Plánování procesů

- v dávkových systémech
- v interaktivních systémech
- Příklad – Windows 2000 (NT/XP/Vista/7)
- Ve víceprocesorových systémech
- V systémech reálného času
- Plánování procesů x plánování vláken

Plánování procesů v interaktivních systémech

- potřeba docílit, aby proces neběžel „příliš dlouho“
  - možnost obsloužit další procesy
- každý proces – jedinečný a nepredikovatelný
  - nelze říct, jak dlouho poběží, než se zablokuje (nad I/O, semaforem, ...)
- vestavěný systémový časovač v počítači
  - provádí pravidelně přerušení (tiky časovače, clock ticks)
  - vyvolá se obslužný podprogram v jádře
  - rozhodnutí, zda proces bude pokračovat, nebo se spustí jiný (preemptivní plánování)

Algoritmus cyklické obsluhy – Round Robin (RR)

- jeden z nejstarších a nejpoužívanějších
- každému procesu přiřazen časový interval
  - časové kvantum, po které může běžet
- proces běží na konci kvanta
  - preemce, naplánován a spuštěn další připravený proces
- proces skončí nebo se zablokuje před uplynutím kvanta
  - stejná akce jako v předchozím bodě ☺

Round Robin

Pokud proces nevyužije celé časové kvantum, okamžitě se naplánuje další proces, na nic se nečeká (je třeba max. využít procesor)

Round Robin

- jednoduchá implementace plánovače
  - plánovač udržuje seznam připravených procesů
  - Při vypršení kvanta nebo zablokování –> vybere další proces

Procesu je nedobrovolně odebrán procesor, přejde do stavu připravený

Proces žádá I/O **dobrovolně** se vzdá CPU, přejde do stavu blokovaný

Obslužný program přerušení časovače

- v jádře
- nastavuje interní časovače systému
- shromažďuje statistiky systému
  - kolik času využíval CPU který proces, ...
- po uplynutí kvanta (resp. v případě potřeby) zavolá plánovač

1 kvantum – odpovídá více přerušením časovače

- Časovač může proces v průběhu časového kvanta přerušit vícekrát.
- přerušení 100x za sekundu (příklad)
  - 10 ms mezi přerušeními
- pokud kvantum 50 ms
  - přeplánování každý pátý tik

vhodná délka časového kvanta

- krátké
  - přepnutí procesů chvíli trvá (uložení a načtení registrů, přemapování paměti, ...)
  - přepnutí kontextu 1ms, kvantum 4ms – 20% velká režie
- dlouhé
  - vyšší efektivita; kvantum 1s – menší režie
  - pokud kvantum delší než průměrná doba držení CPU procesem – preempce je třeba zřídka
  - problém interaktivních procesů – 10 uživatelů stiskne klávesu, odezva posledního procesu až 10s

vhodná délka kvanta - shrnutí

- krátké kvantum – snižuje efektivitu (režie)
- dlouhé – zhoršuje dobu odpovědi na interaktivní požadavky
- kompromis ☺
- pro algoritmus cyklické obsluhy obvykle 20 až 50 ms
- kvantum nemusí být konstantní
  - změna podle zatížení systému
- pro algoritmy, které se lépe vypořádají s interaktivními požadavky lze kvantum delší – 100 ms

Problém s algoritmem cyklické obsluhy

- v systému výpočetně vázané i I/O vázané úlohy
- výpočetně vázané – většinou kvantum spotřebují
- I/O vázané – pouze malá část kvanta se využije a zablokují se
- výpočetně vázané – získají nespravedlivě vysokou část času CPU
- modifikace VRR (Virtual RR, 1991)
  - procesy po dokončení I/O mají přednost před ostatními

Prioritní plánování

- předpoklad RR: všechny procesy stejně důležité
- ale:
  - vyšší priorita zákazníkům, kteří si „připlatí“
  - interaktivní procesy vs. procesy běžící na pozadí (odesílání pošty)
- prioritu lze přiřadit staticky nebo dynamicky:
- staticky
  - při startu procesu, např. Linux – nice
- dynamicky
  - přiradit I/O větší prioritu, použití CPU a zablokování

Priorita

priorita = statická + dynamická
- obsahuje obě složky – výsledná jejich součtem
- statická (při startu procesu)
- dynamická (chování procesu v poslední době)

- kdyby pouze statická složka a plánování jen podle priorit – běží pouze připravené s nejvyšší prioritou
- plánovač snižuje dynamickou prioritu běžícího procesu při každém tiku časovače; klesne pod prioritu jiného - přeplánování

Dynamická priorita

- V kvantově orientovaných plánovacích algoritmech:

- dynamická priorita např. dle vzorce 1 / f
- f – velikost části kvanta, kterou proces naposledy použil
- zvýhodní I/O vázané x CPU vázaným

Pokud nevyužil celé kvantum, jeho dynamická priorita se zvyšuje, např. pokud využil posledně jen 0.5 kvanta, tak 1/0,5 = 2, pokud celé kvantum využil 1/1=1

Spojení cyklického a prioritního plánování

- prioritní třídy
  - v každé třídě procesy se stejnou prioritou
- prioritní plánování mezi třídami
  - Bude obsluhována třída s nejvyšší prioritou
- cyklická obsluha uvnitř třídy
  - V rámci dané třídy se procesy cyklicky střídají

- obsluhovány jsou pouze připravené procesy v nejvyšší neprázdné prioritní třídě

A kdy se dostane na další fronty?

Prioritní třídy

Máme zde priority, třídy i časová kvanta

4 prioritní třídy

dokud procesy v třídě 3 – spustit cyklicky každý na 1 kvantum

pokud třída 3 prázdná – obsluhujeme třídu 2

(prázdná => žádný proces danou prioritu nemá, nebo je ve stavu blokovaný, čeká např. na I/O)

jednou za čas – přepočítání priorit

procesům, které využívaly CPU se sníží priorita

Prioritní třídy

- dynamické přiřazování priority
  - dle využití CPU **v poslední době**
  - priorita procesu
    - snižuje se při běhu
    - zvyšuje při nečinnosti
- cyklické střídání procesů

- OS typu Unix
  - Mají 30 až 50 prioritních tříd

Plánovač spravedlivého sdílení

- problém:
  - čas přidělován každému procesu nezávisle
  - Pokud uživatel má více procesů než jiný uživatel -> dostane více času celkově
- spravedlivé sdílení
  - přidělovat čas každému uživateli (či jinak definované skupině procesů) proporcionálně, bez ohledu na to, kolik má procesů
  - máme-li N uživatelů, každý dostane 1/N času

= spravedlnost vůči uživateli

Spravedlivé sdílení

- nová položka priorita skupiny spravedlivého plánování
  - Zavedena pro každého uživatele
- obsah položky
  - započítává se do priority každého procesu uživatele
  - odráží poslední využití procesoru všemi procesy daného uživatele

Má-li uživatel Pepa procesy p1, p2, p3
a pokud proces p3 bude využívat CPU hodně často, budou touto položkou penalizovány i další procesy uživatele Pepa

Spravedlivé sdílení - implementace

- každý uživatel – položka g
- obsluha přerušení – inkrementuje g uživatele, kterému patří právě běžící proces
- jednou za sekundu rozklad: g=g/2
  - Aby odrážel chování **v poslední době**, vzdálená minulost nás nezajímá
- priorita P (p,g) = p – g
- pokud procesy uživatele využívaly CPU v poslední době – položka g je vysoká, vysoká penalizace

Plánování pomocí loterie

- Lottery Scheduling (Waldspurger & Weihl, 1994)
- cílem – poskytnout procesům příslušnou proporci času CPU
- základní princip:
  - procesy obdrží tikety (losy)
  - plánovač vybere náhodně jeden tiket
  - vítězný proces obdrží cenu – 1 kvantum času CPU
  - důležitější procesy – více tiketů, aby se zvýšila šance na výhru (celkem 100 losů, proces má 20 – v dlouhodobém průměru dostane 20% času)

Loterie - výhody

řešení problémů, v jiných plán. algoritmech obtížné

- spolupracující procesy – mohou si předávat losy
  - klient posílá zprávu serveru a blokuje se
  - může serveru propůjčit všechny své tikety
  - po vykonání požadavku server tikety vrátí
  - nejsou-li požadavky, server žádné tikety nepotřebuje

Loterie - výhody

- rozdělení času mezi procesy v určitém poměru
  - neplatí u prioritního plánování, co je to že má proces prioritu např. 30?
  - proces – tickety – šance vyhrát
- zatím spíše experimentální algoritmus

Shrnutí

Algoritmus	Rozhodovací mód	Prioritní funkce	Rozhodovací pravidlo
RR	Preemptivní vyprší kvantum	P() = 1	cyklicky
prioritní	Preemptivní P jiný > P	Viz text	Náhodně, cyklicky
spravedlivé	Preemptivní P jiný > P	P(p,g)=p-g	cyklicky
loterie	Preemptivní vyprš. kv.	P() = 1	Dle výsledku loterie

Příklad – Windows 2000/XP/…

- 32 prioritních úrovní, 0 až 31 (nejvyšší)
- pole 32 položek
  - každá položka – ukazatel na seznam připravených procesů
- plánovací algoritmus – prohledává pole od 31 po 0
  - nalezne neprázdnou frontu
  - naplánuje první proces, nechá ho běžet 1 kvantum
  - po uplynutí kvanta – proces na konec fronty na příslušné prioritní úrovni

Windows – skupiny priorit

priorita	**popis**
0	Nulování stránek pro správce paměti
1 .. 15	Obyčejné procesy
16 .. 31	Systémové procesy

Windows - priority

- 0 .. pokud není nic jiného na práci
- 1 .. 15 – obyčejné procesy
- aktuální priorita – <bázová, 15>
- bázová priorita – základní, může ji určit uživatel voláním *SetPriorityClass*
- aktuální priorita se mění – viz dále
- procesy se plánují přísně podle priorit, tj. obyčejné pouze pokud není žádný systémový proces připraven

Windows – změna akt. priority

- dokončení I/O zvyšuje prioritu o
  - 1 – disk, 2 – sériový port, 6 – klávesnice, 8 – zvuková karta
- vzbuzení po čekání na semafor, mutex zvýší o
  - 2 - pokud je proces na popředí (řídí okno, do kterého je posílán vstup z klávesnice)
  - 1 – jinak
- proces využil celé kvantum
  - sníží se priorita o 1
- proces neběžel dlouhou dobu
  - na 2 kvanta priorita zvýšena na 15 (zabránit inverzi priorit)

Windows – plánování na vláknech

proces A = 10 spustitelných vláken
proces B = 2 spustitelná vlákna
předpokládáme - stejná priorita

každé vlákno cca 1/12 CPU času
NENÍ 50% A, 50% B

nedělí ferově mezi procesy, ale mezi vlákna

Idle threads

- Jeden pro každý CPU
  - „pod prioritou 0“
  - účtování nepoužívaných clock threadů
  - umožní nastavit vhodný power management – volá HAL (hardware abstraction layer)

Zero page thread

- Jeden pro celý systém
- Běží na úrovni priority 0
- Nuluje nepoužívané stránky paměti

Bezpečnostní opatření, když nějakému procesu přidělíme stránku paměti, aby v ní nezůstali data jiného procesu „z dřívějška“, aby se nedostal k informacím, ke kterým se dostat nemá

Kvantum, stretching

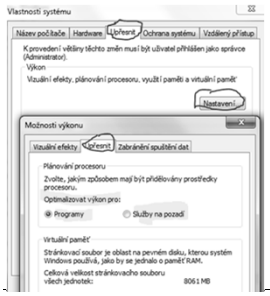
- kvantum stretching
  - maximum 6 tiků (3x)
  - middle 4 tiky (2x)
  - none 2 tiky
- Na desktopu je defaultní kvantum 2 ticky vlákna na popředí – může být stretching
- na serveru je kvantum vždy 12 ticků, není kvantum stretching
- standardní clock tick je 10 nebo 15 ms

Zjištění hodnoty časovače

Program clockres ze sady Sysinternal

```
C:\Izos2012\prednasky\07\dalsi\SysinternalsSuite>clockres

ClockRes v2.0 - View the system clock resolution
Copyright (C) 2009 Mark Russinovich
Sysinternals - www.sysinternals.com

Maximum timer interval: 15.600 ms
Minimum timer interval: 0.500 ms
Current timer interval: 10.000 ms
```

Win 7 – vlastnosti systému – upřesnit – optimalizovat výkon pro



registrový klíč:
HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Control\PriorityControl

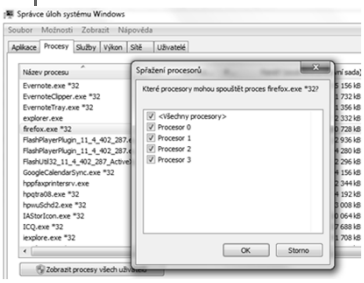
Win32PrioritySeparation 2
6bitů: XX XX XX
kvantum
- krátké, dlouhé
- proměnné, pevné
- navýšení pro procesy na popředí: 2x, 3x)
viz
http://technet.microsoft.com/library/Cc976120

Windows: vlákénka (fibers)

- kromě vláken i fibers
- fibers plánuje vlastní aplikace, nikoliv centrální plánovač jádra
- vytvoření fiberu: CreateFiber
- nepreemptivní plánování – odevzdá řízení jinému vlákenku přes SwitchToFiber

příklad:
http://msdn.microsoft.com/en-us/library/windows/desktop/ms686919%28v=vs.85%29.aspx

Windows - Afinita



afinita

určení CPU (jádra CPU), na kterých může proces běžet

hard afinity
seznam jader

soft afinitiy
vlákno přednostně plánováno na procesor, kde běželo naposledy

Přečtěte si…

http://cs.wikipedia.org/wiki/Plánování_procesů
http://en.wikipedia.org/wiki/Scheduling_%28computing%29
shrnutí – vhodné pro zopakování

http://cs.wikipedia.org/wiki/Preempce_%28informatika%29
http://cs.wikipedia.org/wiki/Změna_kontextu
http://cs.wikipedia.org/wiki/Mikrojádro

http://cs.wikipedia.org/wiki/Round-robin_scheduling
http://cs.wikipedia.org/wiki/Priority_scheduling
http://cs.wikipedia.org/wiki/Earliest_deadline_first (RTOS)
http://cs.wikipedia.org/wiki/Completely_Fair_Scheduler (CFS)

Linux

- vlastní jádro
  (nepreemptivní, dobrovolně preemptivní, preemptivní)
- epocha
  - čas přidělený procesu
  - když jej všechny procesy po částech spotřebují, začíná nová epocha, tedy dostanou nový přidělený čas
- plánovače (nastavitelné per proces)
  - SCHED_FIFO – pro RT úlohy bez přerušení
  - SCHED_RR (RoundRobin) – RT úlohy, preemptivně
  - SCHED_BATCH – pro dávkové úlohy
  - SCHED_OTHER – běžné úlohy (nice, dynamické priority)

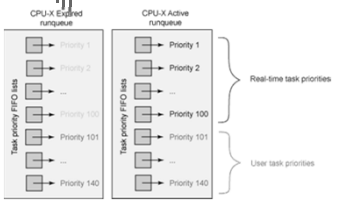
Linux scheduler

verze do 2.6
multilevel feedback queue (pozor, trochu jiný)
procesy mají time slice
priority 0-140

- 0 – 99 real-time úlohy, kvantum 200ms
- 100-140 nice tasks level, kvantum 10ms

dvě fronty

- active queue
  - když je prázdná, vymění se jejich role
- expired queue
  - sem přijde proces, když vyčerpá celý svůj time slice



Active RunQueue,
Expired RunQueue

zdroj obrázku:
http://www.ibm.com/developerworks/linux/library/l-scheduler/index.html

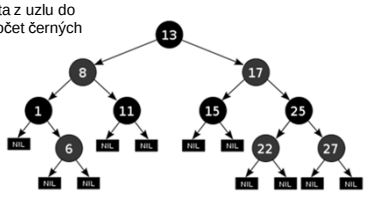
statická priorita 0..99, k běhu vybrán s nejvyšší statickou prioritou
statická priorita 0 (SCHED_BATCH, SCHED_OTHER)
statická priorita >0 (SCHED_FIFO, SCHED_RR)
dynamická priorita (-20 až 19, viz nice)
přečíst:
http://www.root.cz/clanky/pridelovani-procesoru-procesum-a-vlaknum-v-linuxu/

Linux scheduler

- O(1) scheduler
  - verze 2.6-2.6.23
  - fronta připravených pro každý procesor
  - pole active, expired ; v active nic – nová epocha
- **Completely Fair Scheduler**
  - verze jádra 2.6.23
  - red-black strom místo front
  - klíč: spent processor time ; nanosekundy
  - rovnoměrné rozdělení času procesům

Red-black tree

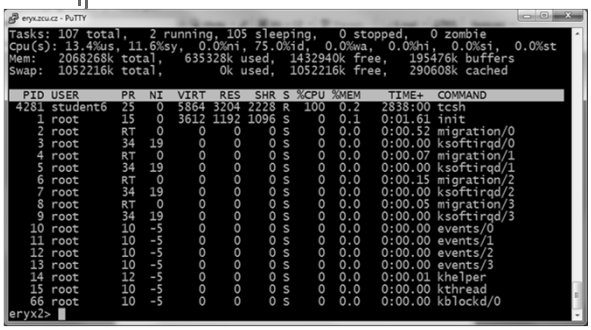
viz wikipedia
self-balancing binary search tree
uzel je červený nebo černý, kořen je černý
všechny listy jsou černé
každá jednoduchá cesta z uzlu do listu obsahuje stejný počet černých uzlů



Linux – stavy procesů

obrázek z: http://www.linuxzone.cz/index.phtml?ids=9&idc=252

Linux – příklad top



1. PID procesu
2. USER – identita uživatele
3. PRI – aktuální priorita daného procesu
4. NICE – výše priority příkazem nice
   - Záporné číslo – vyšší priorita
   - Kladné číslo – sníží prioritu (běžný uživatel)
5. VIRT – celková velikost procesu
   - Kód + zásobník + data
6. RES – velikost použité fyzické paměti
7. SHR – sdílená paměť
8. STAT – stav procesu
9. %CPU – kolik procent CPU nyní využívá
10. %MEM – procento využití fyzické paměti daným proc.
11. TIME – celkový procesorový čas
12. COMMAND - příkaz

Příkaz nice

Změna priority procesu

- Běžný uživatel 0 až +19, tedy pouze snižovat
- root: -20 (nejvyšší) až +19 (nejnižší)

eryx2> /bin/bash
eryx2> nice -n -5 sleep 10
nice: cannot set niceness: Permission denied
eryx2> nice -n +5 sleep 10
Pozn: záleží i na shellu, který máme

Příkaz renice

Změna priority běžícího procesu
Běžný uživatel

- může měnit jen u svých procesů
- opět pouze snižovat

eryx2> renice +10 32022
32022: old priority 5, new priority 10

Proces – stav blokovaný (Unix)

- čeká na událost – ve frontě
- přerušitelné signálem (terminál, sockety, pipes)
  - procesy označené s
    - signál – syscall se zruší – návrat do userspace
    - obsluha signálu
    - znovu zavolá přerušené syst. volání (pokud požadováno)
- nepřerušitelné
  - procesy označené D
  - operace s diskem – skončí v krátkém čase
- plánovač mezi nimi nerozlišuje

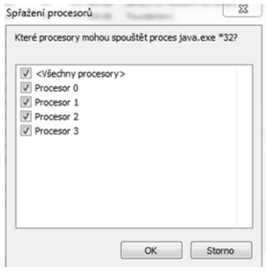
Plánování – víceprocesorové stroje

- nejčastější architektura
  - těsně vázaný symetrický multiprocesor
  - procesory jsou si rovné, společná hlavní paměť
- Přiřazení procesů procesorům – ukázka
  - Permanentní přiřazení
    - Menší režie, některá CPU mohou zahálet
    - Afinita procesu k procesoru, kde běžel naposledy
    - Někdy procesoru přiřazen jediný proces – RT procesy
  - Společná fronta připravených procesů
    - Plánovány na libovolný procesor

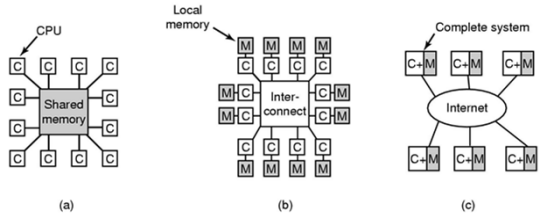
afinita

na jakých CPU může daný proces běžet

správce úloh systému Windows – procesy – vybrat proces – pravá myš – nastavit spřažení



Multiprocesorové systémy



Architektury:

- shared memory model (sdílená paměť)
- message passing multiprocessor (předávání zpráv)
- wide area distributed system (distribuovaný systém)

Víceprocesorové stroje

Plánování vláken

Některé paralelní aplikace – podstatně větší výkonnost, pokud jejich vlákna běží současně

- Zkrátí se vzájemné čekání vláken

Plánování v systémech reálného času

Charakteristika RT systémů

- RT procesy řídí nebo reagují na události ve vnějším světě
- Správnost závisí nejen na výsledku, ale i na čase, ve kterém je výsledek vyprodukován
- S každou podúlohou – sdružit deadline – čas kdy musí být spuštěna nebo dokončena
- Hard RT – času musí být dosaženo
- Soft RT – dosažení deadline je žádoucí

Systémy RT

Podúlohy procesu (události, na které se reaguje)

- Aperiodické – nastávají nepredikovatelně
- Periodické – v pravidelných intervalech

Zpracování události vyžaduje čas

Pokud je možné všechny včas zpracovat

=> systém je plánovatelný (schedulable)

Plánovatelné RT systémy

- Je dáno
  - *m* – počet periodických událostí
  - výskyt události *i* s periodou P_i vyžadující C_i sekund
- Zátěž lze zvládnout, pokud platí:

$$\sum_{i=1}^{m} \frac{C_i}{P_i} \leq 1$$

Plánovací algoritmy v RT

- Statické nebo dynamické
- Statické
  - Plánovací rozhodnutí před spuštěním systému
  - Předpokládá dostatek informací o vlastnostech procesů
- Dynamické
  - Za běhu
  - Některé algoritmy provedou analýzu plánovatelnosti, nový proces přijat pouze pokud je výsledek plánovatelný

Vlastnosti současných RT

- Malá velikost OS –> omezená funkčnost
- Snaha spustit RT proces co nejrychleji
  - Rychlé přepínání mezi procesy nebo vlákny
  - Rychlá obsluha přerušení
  - Minimalizace intervalů, kdy je přerušení zakázáno
- Multitasking + meziprocesová komunikace (semafory, signály, události)
- Primitiva pro zdržení procesu o zadaný čas, čítače časových intervalů
- Někdy rychlé sekvenční soubory (viz později)

Zpátky obecně k plánování procesů

Plánování procesů a vláken

- Plánování procesů – vždy součást OS
- Plánování vláken
  - Běh vláken plánuje OS
    - Kernel-level threads
  - Běh vláken plánován uživatelským procesem
    - User-level threads
    - OS o existenci vláken nic neví

Plánování vláken

- Vlákna plánována OS
  - Stejné mechanismy a algoritmy jako pro plánování procesů
  - Často plánována bez ohledu, kterému procesu patří (proces 10 vláken, každé obdrží časové kvantum)

Plánování vláken

- Vlákna plánována uvnitř procesu
  - Běží v rámci času, který je přidělen procesu
  - Přepínání mezi vlákny – systémová knihovna
  - Pokud OS neposkytuje procesu pravidelné "přerušení", tak pouze nepreemtivní plánování
  - Obvykle algoritmus RR nebo prioritní plánování
  - Menší režie oproti kernel-level threads, menší možnosti
- Windows 2000> a Linux – vlákna plánována jádrem
- Některé varianty UNIXu – user-level threads

Dispatcher

- Dispatcher
  - Modul, který předá řízení CPU procesu vybraným short-term plánovačem
- Provede:
  - Přepnutí kontextu
  - Přepnutí do uživatelského modu
  - Skok na danou instrukci v uživatelském procesu
- Co nejrychlejší, vyvolán během každého přepnutí procesů

Scheduler – protichůdné požadavky

- příliš časté přepínání procesu – velká režie
- málo časté – pomalá reakce systému
- čekání na diskové I/O, data ze sítě – probuzen a brzy (okamžitě) naplánován – pokles přenosové rychlosti
- multiprocesor – pokud lze, nestřídat procesory
- nastavení priority uživatelem

Poznámka – vyhladovění procesu

V roce 1973 na MITU shut down stroje IBM 7094
Nalezen proces, který nebyl spuštěn od roku 1967

Poznámka - simulace

- Trace tape – monitorujeme běh reálného systému, zaznamenáváme posloupnost událostí
- Tento záznam použijeme pro řízení simulace
- Lze využít pro porovnávání algoritmů
- Trace tape – nutno uložit velké množství dat

Uvíznutí (deadlock)

- Příklad:
- Naivní večeřící filozofové – vezmou levou vidličku, ale nemohou vzít pravou (už je obsazena)
- Uvíznutí (deadlock); zablokování

Uvíznutí – alokace I/O zařízení

Výhradní alokace I/O zařízení

zdroje:

Vypalovačka CD (V), scanner (S)

procesy:

A, B – oba úkol naskenovat dokument a zapsat na vypalovačku

1. A žádá V a dostane, B žádá S a dostane
2. A žádá S a čeká, B žádá V a čeká -- **uvíznutí !!**

Uvíznutí – zamykání záznamů v databázi, semafory

- Dva procesy A, B požadují přístup k záznamům R,S v databázi
- A zamkne R, B zamkne S, …
- A požaduje S, B požaduje R

Vymyslete příklad deadlocku s využitím semaforů

Zdroje

- přeplánovatelné (preemtable)
  - lze je odebrat procesu bez škodlivých efektů
- nepřeplánovatelné (nonpremeptable)
  - proces zhavaruje, pokud jsou mu odebrány

Zdroje

- Sériově využitelné zdroje
  - Proces zdroj alokuje, používá, uvolní
- Konzumovatelné zdroje
  - Např. zprávy, které produkuje jiný proces
  - Viz producent – konzument

Také zde uvíznutí:

1. Proces A: … receive (B,R); send (B, S); ..
2. Proces B: … receive (A,S); send (A, R); ..

Dále budeme povídat o sériově využitelných zdrojích,

problémy jsou stejné

Více zdrojů stejného typu

Některé zdroje – více exemplářů

Proces žádá zdroj daného typu – jedno který dostane

Např. bloky disku pro soubor, paměť, …

- Př. 5 zdrojů a dva procesy A,B
  1. A požádá o dva zdroje, dostane (zbydou 3)
  2. B požádá o dva zdroje, dostane (zbude 1)
  3. A žádá o další dva, nejsou (je jen 1), čeká
  4. B žádá o další dva, nejsou, čeká – nastalo uvíznutí

Zaměříme se na situace, kdy 1 zdroj každého typu

Práce se zdrojem

- Žádost (request)
  - Uspokojena bezprostředně nebo proces čeká
  - Systémové volání
- Použití (use)
  - Např. tisk na tiskárně
- Uvolnění (release)
  - Proces uvolní zdroj
  - Systémové volání

Uvíznutí - definice

- Obecný termín zdroj – zařízení, záznam, …

V množině procesů nastalo uvíznutí, jestliže každý proces množiny čeká na událost, kterou může způsobit jiný proces množiny

- Všichni čekají – nikdo událost nevygeneruje, nevzbudí jiný proces

Podmínky vzniku uvíznutí (!!!)

Coffman, 1971

1. vzájemné vyloučení
   - Každý zdroj je buď dostupný nebo je výhradně přiřazen právě jednomu procesu

2. hold and wait
   - Proces držící výhradně přiřazené zdroje může požadovat další zdroje

Podmínky vzniku uvíznutí

3. nemožnost odejmutí
   - Jednou přiřazené zdroje nemohou být procesu násilně odejmuty (proces je musí sám uvolnit)

4. cyklické čekání
   - Musí být cyklický řetězec 2 nebo více procesů, kde každý z nich čeká na zdroj držený dalším členem

Vznik uvíznutí - poznámky

- Pro vznik uvíznutí – musejí být splněny všechny 4 podmínky
  - 1. až 3. předpoklady, za nich je definována 4. podmínka
- Pokud jedna z podmínek není splněna, uvíznutí nenastane

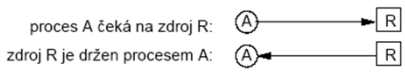
- Viz příklad s CD vypalovačkou
  - Na CD může v jednu chvíli zapisovat pouze 1 proces
  - CD vypalovačku není možné zapisovacímu procesu odejmout

Modelování uvíznutí

Graf alokace zdrojů

- 2 typy uzlů
  - Proces – zobrazujeme jako kruh
  - Zdroj – jako čtverec
- hrany
  - Hrana od zdroje k procesu:
    - zdroj držen procesem
  - Hrana od procesu ke zdroji:
    - proces blokován čekáním na zdroj

Modelování uvíznutí



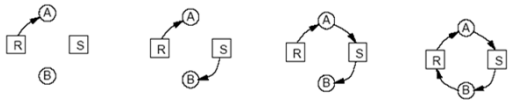
Cyklus v grafu – nastalo uvíznutí

Uvíznutí se týká procesů a zdrojů v cyklu

Uvíznutí

zdroje: Rekorder R a scanner S; procesy: A,B

1. A žádá R dostane, B žádá S dostane
2. A žádá S a čeká, B žádá R a čeká - uvíznutí



Uvíznutí - poznámky

- Cyklus v grafu – nutnou a postačující podmínkou pro vznik uvíznutí
- Závisí na pořadí vykonávání instrukcí procesů
- Pokud nejprve alokace a uvolnění zdrojů procesu A, potom B => uvíznutí nenastane

Uvíznutí - poznámky

1. A žádá R a S, oba dostane, A oba zdroje uvolní
2. B žádá S a R, oba dostane, B oba zdroje uvolní

- Nenastane uvíznutí
- Při některých běžích nemusí uvíznutí nastat – hůře se hledá chyba

Uvíznutí – pořadí alokace

- Pokud bychom napsali procesy A,B tak, aby oba žádaly o zdroje R a S ve stejném pořadí – uvíznutí nenastane

1. A žádá R a dostane, B žádá R a čeká
2. A žádá S a dostane, A uvolní R a S
3. B čekal na R a dostane, B žádá S a dostane

08.
Deadlock
Přidělování paměti

ZOS 2013, L. Pešička

Obsah

- Deadlock
  - Jak předcházet, detekovat, reagovat
- Metody přidělování paměti

Jak se vypořádat s uvíznutím

1. Problém uvíznutí je zcela ignorován
2. Detekce a zotavení
3. Dynamické zabránění pomocí pečlivé alokace zdrojů
4. Prevence, pomocí strukturální negace jedné z dříve uvedených nutných podmínek pro vznik uvíznutí

1. Ignorování problému

- Předstíráme, že problém neexistuje ☺
  - „pštrosí algoritmus“
- Vysoká cena za eliminaci uvíznutí
  - Např. činnost uživatelských procesů je omezena
  - Neexistuje žádné univerzální řešení
- Žádný ze známých OS se nezabývá uvíznutím uživatelských procesů
  - Snaha o eliminaci uvíznutí pro činnosti jádra

U uživatelských procesů uvíznutí neřešíme, snažíme se, aby k uvíznutí nedošlo v jádře OS

2. Detekce a zotavení

- Systém se nesnaží zabránit vzniku uvíznutí
- Detekuje uvíznutí
- Pokud nastane, provede akci pro zotavení

Samotná detekce uvíznutí nemusí být snadná

- Detekce pro 1 zdroj každého typu
  - Při žádostech o zdroj OS konstruuje graf alokace zdrojů
  - Detekce cyklu – pozná, zda nastalo uvíznutí
  - Různé algoritmy detekce cyklu (teorie grafů)
    - Např. prohledávání do hloubky z každého uzlu, dojdeme-li do uzlu, který jsme již prošli - cyklus

Zotavení z uvíznutí (pokračování 2.)

Zotavení pomocí preempce

- Vlastníkovi zdroj dočasně odejmout
- Závisí na typu zdroje – často obtížné či nemožné
  - Tiskárna – po dotištění stránky proces zastavit, ručně vyjmout již vytištěné stránky, odejmout procesu a přiřadit jinému

Zotavení z uvíznutí – zrušení změn

- Zotavení pomocí zrušení změn (rollback)
  - **Častá uvíznutí** – checkpointing procesů
    = zápis stavu procesů do souboru, aby proces mohl být v případě potřeby vrácen do uloženého stavu
  - **Detekce uvíznutí** – nastavení na dřívější checkpoint, kdy proces ještě zdroje nevlastnil (následná práce ztracena)
  - Zdroj přiřadíme uvízlému procesu – zrušíme deadlock
  - Proces, kterému jsme zdroj odebrali – pokusí se ho alokovat - usne

Zotavení z uvíznutí – zrušení procesu

- Zotavení pomocí zrušení procesu
  - Nejhorší způsob – zrušíme jeden nebo více procesů
  - Zrušit proces v cyklu
    - Pokud nepomůže zrušit jeden, zrušíme i další
- Často alespoň snaha zrušit procesy, které je možné spustit od začátku

3. Dynamické zabránění

- Ve většině systémů procesy žádají o zdroje po jednom
- Systém rozhodne, zda je přiřazení zdroje bezpečné, nebo hrozí uvíznutí
- Pokud bezpečné – zdroj přiřadí, jinak pozastaví žádající proces
- Stav je bezpečný, pokud existuje alespoň jedna posloupnost, ve které mohou procesy doběhnout bez uvíznutí
- I když stav není bezpečný, uvíznutí nemusí nutně nastat

Bankéřův algoritmus pro jeden typ zdroje

- Předpokládáme více zdrojů stejného typu
  - Např. N magnetopáskových jednotek
- Algoritmus plánování, který se dokáže vyhnout uvíznutí (Dijkstra 1965)
- Bankéř na malém městě, 4 zákazníci – A, B, C, D
- Každému garantuje půjčku (6, 5, 4, 7) = 22 dohromady
- Bankéř ví, že všichni zákazníci nebudou chtít půjčku současně, pro obsluhu zákazníků si ponechává pouze 10

Bankéřův algoritmus

Zákazník	Má půjčeno	Max. půjčka
A	1	6
B	1	5
C	2	4
D	4	7

Bankéř má volných prostředků: 10 – (1+1+2+4) = 2

Stav je bezpečný, bankéř může pozastavit všechny požadavky kromě C

Dá C 2 jednotky, C skončí a uvolní 4, může použít pro D nebo B atd.

Bankéřův algoritmus (B o 1 více)

Zákazník	Má půjčeno	Max. půjčka
A	1	6
B	2	5
C	2	4
D	4	7

Dáme B o jednotku více; zůstane nám volných prostředků: 1

Stav není bezpečný – pokud všichni budou chtít maximální půjčku, bankéř nemůže uspokojit žádného – nastalo by uvíznutí
Uvíznutí nemusí nutně nastat, ale s tím bankéř nemůže počítat …

Rozhodování bankéře

Zkusí „jako by“ přidělit zdroj a zkoumá, zda je nový stav bezpečný

- U každého požadavku – zkoumá, zda vede k bezpečnému stavu:
  - Bankéř předpokládá, že požadovaný zdroj byl procesu přiřazen a že všechny procesy požádaly o všechny bankéřem garantované zdroje
  - Bankéř zjistí, zda je dostatek zdrojů pro uspokojení některého zákazníka; pokud ano – předpokládá, že zákazníkovi byla suma vyplacena, skončil a uvolnil (vrátil) všechny zdroje
  - Bankéř opakuje předchozí krok, pokud mohou všichni zákazníci skončit, je stav bezpečný

Vykonání požadavku

- Proces požaduje nějaký zdroj
- Zdroje jsou poskytnuty pouze tehdy, pokud požadavek vede k bezpečnému stavu
- Jinak je požadavek odložen na později – proces je pozastaven

Bankéřův algoritmus pro více typů zdrojů

- zobecněn pro více typů zdrojů
- používá dvě matice (sloupce – třídy zdrojů, řádky – zákazníci)
  - matice přiřazených zdrojů (current allocation matrix)
    - který zákazník má které zdroje
  - matice ještě požadovaných zdrojů (request matrix)
    - kolik zdrojů kterého typu budou procesy ještě chtít

	Zdroj R	Zdroj S	Zdroj T
Zák. A	3	0	1
Zák. B	0	1	0
Zák. C	1	1	1
Zák. D	1	1	0

Matice přiřazených zdrojů

	Zdroj R	Zdroj S	Zdroj T
Zák. A	1	1	0
Zák. B	0	1	1
Zák. C	3	1	0
Zák. D	0	0	1

Matice ještě požadovaných zdrojů

zavedeme vektor A volných zdrojů (available resources)

např. A = (1, 0, 1) znamená jeden volný zdroj typu R, 0 typu S, 1 typu T

Určení, zda je daný stav bezpečný

1. V matici ještě požadovaných zdrojů hledáme řádek, který je menší nebo roven A.
   Pokud neexistuje, nastalo by uvíznutí.
2. Předpokládáme, že proces obdržel všechny požadované zdroje a skončil. Označíme proces jako ukončený a přičteme všechny jeho zdroje k vektoru A.
3. Opakujeme kroky 1. a 2., dokud všechny procesy neskončí (tj. původní stav byl bezpečný), nebo dokud nenastalo uvíznutí (původní stav nebyl bezpečný)

Bankéřův algoritmus & použití v praxi

- publikován 1965, uváděn ve všech učebnicích OS
- v praxi v podstatě nepoužitelný
  - procesy obvykle nevědí dopředu, jaké budou jejich maximální požadavky na zdroje
  - počet procesů není konstantní (uživatelé se přihlašují, odhlašují, spouštějí procesy, ...)
  - zdroje mohou zmizet (tiskárně dojde papír ...)
- nepoužívá se v praxi pro zabránění uvíznutí
- odvozené algoritmy lze použít pro detekci uvíznutí při více zdrojích stejného typu

4. Prevence uvíznutí

jak skutečné systémy zabraňují uvíznutí?
viz 4 Coffmanovy podmínky vzniku uvíznutí

1. vzájemné vyloučení – výhradní přiřazování zdrojů
2. hold and wait – proces držící zdroje může požadovat další
3. nemožnost zdroje odejmout
4. cyklické čekání

pokud některá podmínka nebude splněna – uvíznutí strukturálně nemožné

P1 – Vzájemné vyloučení

- prevence – zdroj nikdy nepřiřadit výhradně
- problém pro některé zdroje (tiskárna)
- spooling
  - pouze daemon přistupuje k tiskárně
  - nikdy nepožaduje další zdroje – není uvíznutí
- spooling není možný pro všechny zdroje (záznamy v databázi)
- převádí soutěžení o tiskárnu na soutěžení o diskový prostor – 2 procesy zaplní disk, žádný nemůže skončit

P2- Hold and wait

- proces držící výhradně přiřazené zdroje může požadovat další zdroje
- požadovat, aby procesy alokovaly všechny zdroje před svým spouštěním
  - většinou nevědí, které zdroje budou chtít
  - příliš restriktivní
  - některé dávkové systémy i přes nevýhody používají, zabraňuje deadlocku
- pokud proces požaduje nové zdroje, musí uvolnit zdroje které drží a o všechny požádat v jediném požadavku

P3 – Nemožnost zdroje odejmout

- odejímat zdroje je velmi obtížné

P4 – Cyklické čekání

- Proces může mít jediný zdroj, pokud chce jiný, musí předchozí uvolnit – restriktivní, není řešení ☹
- Všechny zdroje očíslovány, požadavky musejí být prováděny v číselném pořadí
  - Alokační zdroj nemůže mít cykly
  - Problém – je těžké nalézt vhodné očíslování pro všechny zdroje
  - Není použitelné obecně, ale ve speciálních případech výhodné (jádro OS, databázový systém, …)

Př. Dvoufázové zamykání

- V DB systémech
- První fáze
  - Zamknutí všech potřebných záznamů v číselném pořadí
  - Pokud je některý zamknut jiným procesem
    - Uvolní všechny zámky a zkusí znovu
- Druhá fáze
  - Čtení & zápis, uvolňování zámků
- Zamyká se vždy v číselném pořadí, uvíznutí nemůže nastat

Shrnutí přístupu k uvíznutí (!)

- Ignorování problému – většina OS ignoruje uvíznutí uživatelských procesů
- Detekce a zotavení – pokud uvíznutí nastane, detekujeme a něco s tím uděláme (vrátíme čas – rollback, zrušíme proces …)
- Dynamické zabránění – zdroj přiřadíme, pouze pokud bude stav bezpečný (bankéřův algoritmus)
- Prevence – strukturálně negujeme jednu z Coffman. podmínek
  - Vzájemné vyloučení – spooling všeho
  - Hold and wait – procesy požadují zdroje na začátku
  - Nemožnost odejmutí – odejmi (nefunguje)
  - Cyklické čekání – zdroje očíslujeme a žádáme v číselném pořadí

Vyhladovění

- Procesy požadují zdroje – pravidlo pro jejich přiřazení
- Může se stát, že některý proces zdroj nikdy neobdrží
  - I když nenastalo uvíznutí !
- Př. Večeřící filozofové
  - Každý zvedne levou vidličku, pokud je pravá obsazena, levou položí
  - Vyhladovění, pokud všichni zvedají a pokládají současně

Vyhladovění 2

- Př. Přiřazování zdroje strategií SJF
  - Tiskárnu dostane proces, který chce vytisknout nejkratší soubor
  - 1 proces chce velký soubor, hodně malých požadavků – může dojít k vyhladovění, neustále předbíhán
- Řešení – FIFO
- Řešení – označíme požadavek časem příchodu a při překročení povolené doby setrvání v systému bude obsloužen

Terminologie

- Blokovaný (blocked, waiting), někdy: čekající
  - Základní stav procesu
- Uvíznutí, uváznutí, deadlock, někdy: zablokování
  - Neomezené čekání na událost
- Vyhladovění, starvation někdy: umoření
  - Procesy běží, ale nemohou vykonávat žádnou činnost
- Aktivní čekání (busy wait), s předbíháním (preemptive)

Bernsteinovy podmínky

- [illegible]

Windows – ukázky funkcí

Správa vláken

CreateThread()
SuspendThread(), ResumeThread()
ExitThread() // ukončení vlákna
TerminateThread() // ukončí jiné vlákno

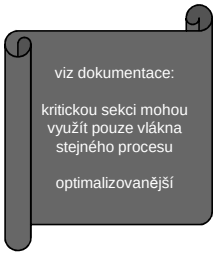
WaitForSingleObject() // čeká na jeden
WaitForMultipleObjects() // čeká na 1 nebo všechny
CloseHandle()

Windows - synchronizace

Kritické sekce

InitializeCriticalSection()
DeleteCriticalSection()

EnterCriticalSection()
LeaveCriticalSection()



Windows - synchronizace

Mutexy

CreateMutex()
OpenMutex()
WaitForSingleObject() // čekáme na mutex
WaitForMultipleObjects()
ReleaseMutex() // uvolníme mutex
CloseHandle()

Windows - semafory

Semafory

CreateSemaphore(), // inic. hodnota, max. hodnota
WaitForSingleObject(), // operace P()
WaitForMultipleObjects()
ReleaseSemaphore(), // operace V()
CloseHandle()

http://msdn.microsoft.com/en-us/library/windows/desktop/ms686946%28v=vs.85%29.aspx

Windows - synchronizace

Eventy

CreateEvent()
SetEvent()
ResetEvent()
WaitForSingleObject()
WaitForMultipleObjects()
CloseHandle()

Windows

Atomické operace

InterlockedIncrement() // inkrementuje o 1
InterlockedDecrement()
InterlockedExchange() // nastaví novou hodnotu
// a vrátí původní

Windows

priorita vláken

SetThreadPriority()
GetThreadPriority()

Osnova

Základní moduly OS

- Modul pro správu procesů - probráno
- Modul pro správu paměti - nyní začínáme
- Modul pro správu periférií
- Modul pro správu souborů

Správa hlavní paměti

- Ideál programátora
  - Paměť nekonečně velká, rychlá, levná
  - Zároveň persistentní (uchovává obsah po vypnutí napájení)
  - Bohužel neexistuje
- Reálný počítač – hierarchie pamětí („pyramida“)
  - Registry CPU
  - Malé množství rychlé cache paměti
  - Stovky MB až gigabajty RAM paměti
  - GB na pomalých, levných, persistentních discích

Správce paměti

- Část OS, která spravuje paměť
- Udržuje informaci, které části paměti se používají a které jsou volné
- Alokuje paměť procesům podle potřeby
  - funkce **malloc** v jazyce C, (new v Pascalu)
- Zařazuje paměť do volné paměti po uvolnění procesem
  - funkce **free** v jazyce C, (release v Pascalu)

Jak to reálně funguje? (!!)

- proces požádá o alokaci n bajtů paměti funkcí
  ukazatel = malloc (n)
- malloc je knihovní fce alokátoru paměti (součást glibc)
- paměť je alokována z haldy (heapu) !
- alokátor se podívá, zda má volnou paměť k dispozici, když ne, požádá OS o přidělení dalších stránek paměti (systémové volání sbrk)
- proces uvolní paměť, když už ji nepotřebuje voláním free(ukazatel)

Příklad alokace

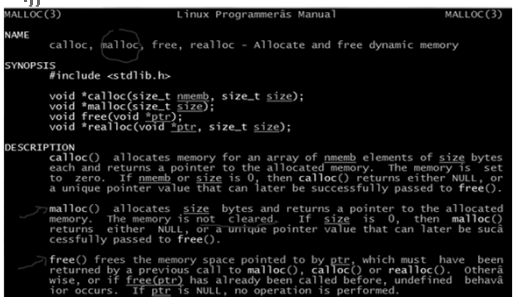
zkuste: man malloc

Příklad:

1. proces bude chtít alokovat 500B, zavolá malloc
2. alokátor koukne, nemá volnou paměť, požádá OS o přidělení stránky paměti (4KB) – sbrk
3. proces je obsloužen, dostane paměť
4. proces bude chtít dalších 200B, zavolá malloc
5. alokátor už má paměť v zásobě, rovnou ji přidělí procesu
6. když už proces paměť nepotřebuje, zavolá free

man malloc

Windows:
také malloc() nebo HeapAlloc()



```
MALLOC(3)                  Linux Programmerás Manual                  MALLOC(3)

NAME
       calloc, malloc, free, realloc - Allocate and free dynamic memory

SYNOPSIS
       #include <stdlib.h>

       void *calloc(size_t nmemb, size_t size);
       void *malloc(size_t size);
       void free(void *ptr);
       void *realloc(void *ptr, size_t size);

DESCRIPTION
       calloc()  allocates memory for an array of nmemb elements of size bytes
       each and returns a pointer to the allocated memory.  The memory is  set
       to  zero.  If nmemb or size is 0, then calloc() returns either NULL, or
       a unique pointer value that can later be successfully passed to free().

       malloc()  allocates  size  bytes and returns a pointer to the allocated
       memory.  The memory is not  cleared.   If  size  is  0,  then  malloc()
       returns  either  NULL, or a unique pointer value that can later be sucá
       cessfully passed to free().

       free() frees the memory space pointed to by ptr, which must  have  been
       returned by a previous call to malloc(), calloc() or realloc().  Otherá
       wise, or if free(ptr) has already been called before, undefined  behavá
       ior occurs.  If ptr is NULL, no operation is performed.
```

poznámka k pointerům

ukazatel = malloc (size)

takto získaný ukazatel obsahuje virtuální adresu, tj. není to přímo adresa do fyzické paměti (RAM) !!

virtuální adresa se uvnitř procesoru převede na fyzickou adresu (s využitím tabulky stránek atd.)

Mechanismy správy pamětí

Od nejjednodušších (program má veškerou paměť) po propracovaná schémata (stránkování se segmentací)

Dvě kategorie:

- Základní mechanismy
  - Program je v paměti po celou dobu svého běhu
- Mechanismy s odkládáním
  - Programy přesouvány mezi hlavní pamětí a diskem

Základní mechanismy pro správu paměti

Nejprve probereme základní mechanismy
Bez odkládání a stránkování

1. Jednoprogramové systémy
2. Multiprogramování s pevným přidělením paměti
3. Multiprogramování s proměnnou velikostí oblasti

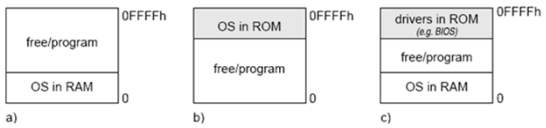
Jednoprogramové systémy

- Spouštíme pouze jeden program v jednom čase
- Uživatel – zadá příkaz , OS zavede program do paměti
- Dovoluje použít veškerou paměť, kterou nepotřebuje OS
- Po skončení procesu lze spustit další proces

Tři varianty rozdělení paměti:

a) OS ve spodní části adresního prostoru v RAM (minipočítače)
b) OS v horní části adresního prostoru v ROM (zapouzdřené systémy)
c) OS v RAM, ovladače v ROM
(na PC – MS DOS v RAM, BIOS v ROM)

Jednoprogramové systémy



Multiprogramování s pevným přidělením paměti

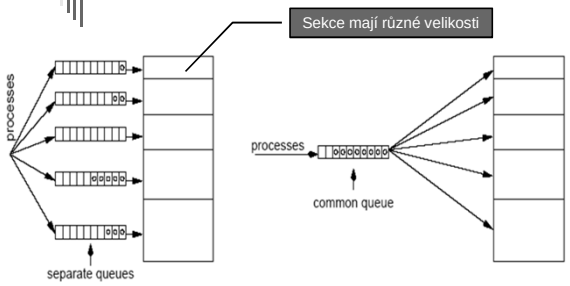
- Většina současných systémů – paralelní nebo pseudoparelelní běh více programů = multiprogramování
- Práce více uživatelů, maximalizace využití CPU apod.
- Nejjednodušší schéma – rozdělit paměť na n oblastí (i různé velikosti)
  - V historických systémech – rozdělení ručně při startu stroje
  - Po načtení úlohy do oblasti je obvykle část oblasti nevyužitá
  - Snaha umístit úlohu do nejmenší oblasti, do které se vejde

Pevné rozdělení sekcí

Několik strategií:

1. Více front, každá úloha do nejmenší oblasti, kam se vejde
2. Jedna fronta – po uvolnění oblasti z fronty vybrat největší úlohu, která se vejde

Pevné rozdělení sekcí



Pevné rozdělení sekcí - vlastnosti

- Strategie 1.
  - Může se stát, že existuje neprázdná oblast, která se nevyužije, protože úlohy čekají na jiné oblasti
- Strategie 2.
  - Diskriminuje malé úlohy (vybíráme největší co se vejde) x malým bychom měli obvykle poskytnout nejlepší službu
  - Řešení – mít vždy malou oblast, kde poběží malé úlohy
  - Řešení – s každou úlohou ve frontě sdružit „čítač přeskočení“, bude zvětšen při každém přeskočení úlohy; po dosažení mezní hodnoty už nesmí být úloha přeskočena

Pevné rozdělení sekcí - poznámky

- Používal např. systém OS/360 (Multiprogramming with Fixed Number of Tasks)

- Multiprogramování zvyšuje využití CPU
- Proces – část času p tráví čekáním na dokončení I/O
- N procesů – pst, že **všechny** čekají na I/O je: p^n
- Využití CPU je u = 1 – p^n

Poznámky

- Využití CPU je u = 1 – p^n
- Pokud proces tráví 80% času čekáním, p = 0.8
- n = 1 … u = 0.2 (20% času CPU využito)
- n = 2 … u = 0.36 (36%)
- n = 3 … u = 0.488 (49%)
- n = 4 … u = 0.5904 (59%)

- n je tzv. stupeň multiprogramování
- Zjednodušení, předpokládá nezávislost procesů, což při jednom CPU není pravda

Poznámky

- Při multiprogramování – všechny procesy je nutné mít alespoň částečně zavedeny v paměti, jinak neefektivní
- Odhad velikosti paměti
- Fiktivní PC 32MB RAM, OS 16MB, uživ. programy po 4MB
  - Max. 4 programy v paměti
- Čekání na I/O 80% času, využití CPU u=1 -0.8^4 = 0.5904
- Přidáme 16MB RAM, stupeň multiprogramování n bude 8
  - Využití CPU u=1 -0.8^8 = 0.83222784
- Přidání dalších 16MB – 12 procesů, u = 0.9313
- První přidání zvýší průchodnost 1.4x (o 40%)
  další přidání 1.12x (o 12%) – druhé přidání se tolik nevyplatí

Multiprogramování s proměnnou velikostí oblasti

- Úloze je přidělena pamět dle požadavku
- V čase se mění
  - Počet oblastí
  - Velikost oblastí
  - Umístění oblastí
- Zlepšuje využiti paměti
- Komplikovanější alokace / dealokace

Př.: IBM OS/360

batch processing operating system

1964

zdroj obrázku:
http://www.maximumpc.com/article/features/ibm_os360_windows_31_software_changed_computing_forever

IBM OS/360

- Single Sequential Scheduler (SSS)
  - Option 1
  - Primary Control Program (PCP)
- Multiple Sequential Schedulers (MSS)
  - Option 2
  - Multiprogramming with a Fixed number of Tasks (MFT)
  - MFT 2
- Multiple Priority Schedulers (MPS)
  - Option 4
  - VMS[NB 1]
  - Multiprogramming with a Variable number of Tasks (MVT)
  - Model 65 Multiprocessing (M65MP)

zdroj:
http://www.escapistmagazine.com/forums/read/18.85690-Esoteric-Operating-Systems-The-History-of-OS-360-and-its-successors

zdroj: http://en.wikipedia.org/wiki/OS/360_and_successors

Problém mnoha volných oblastí

- Může vzniknout mnoho volných oblastí (děr)
  - Paměť se „rozdrobí“
- Kompaktace paměti (compaction)
  - Přesunout procesy směrem dolů
  - Drahá operace (1B .. 10ns, 256MB .. 2.7s)
  - Neprovádí se bez speciálního HW

Volná x alokovaná paměť

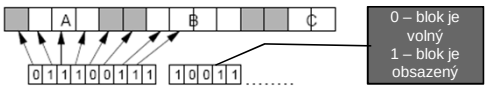
Pro zajištění správy paměti se používají:

1. bitové mapy
2. seznamy
   - first fit, best fit, next fit, …
3. buddy systems

U každého bloku paměti potřebujeme rozhodnout, zda je volný nebo někomu přidělený

Správa pomocí bitových map

- Paměť rozdělíme na alokační jednotky stejné délky (B až KB)
- S každou jednotkou 1bit (volno x obsazeno)



Menší alokační jednotky – větší bitmapa
Větší jednotky – více nevyužité paměti
Alokační jednotka 4 byty (32bitů):
na každých 32bitů paměti potřebujeme 1bit signalizační
tedy .. 1/33 paměti

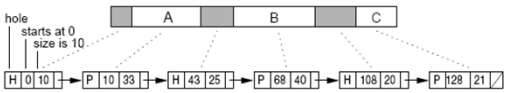
Bitové mapy

+ konstantní velikost bitové mapy

- najít požadovaný úsek *N* volných jednotek
  - Náročné, příliš často se nepoužívá pro tento účel

Správa pomocí seznamů

- Seznam alokovaných a volných oblastí (procesů, děr)
- Položka seznamu:
  - Info o typu – proces nebo díra (P vs. H)
  - Počáteční adresa oblasti
  - Délka oblasti



Práce se seznamem

- Proces skončí – P se nahradí H (dírou)
- Dvě H vedle sebe – sloučí se

Seznam seřazený podle počáteční adresy oblasti
Může být obousměrně vázaný seznam
– snadno k předchozí položce

Jak prohledávat seznam, když proces potřebuje alokovat paměť?

Alokace – first fit, next fit

- First Fit (první vhodná)
  - Prohledávání, dokud se nenajde dostatečně velká díra
  - Díra se rozdělí na část pro proces a nepoužitou oblast (většinou „nesedne“ přesně)
  - Rychlý, prohledává co nejméně
- Next Fit (další vhodná)
  - Prohledávání začne tam, kde skončilo předchozí
  - O málo horší než *first fit*

Alokace best fit

- Best fit (nejmenší/nejlepší vhodná)
  - Prohlédne celý seznam, vezme nejmenší díru, do které se proces vejde
  - Pomalejší – prochází celý seznam
  - Více ztracené paměti než FF,NF – zaplňuje paměť malými nepoužitelnými dírami
- Worst fit (největší díra) – není vhodné
  - nepoužívá se

Urychlení

- Oddělené seznamy pro proces a díry
  - Složitější a pomalejší dealokace
  - Vyplatí se při rychlé alokaci paměti pro data z I/O zařízení
  - *Alokace* – jen seznam děr
  - *Dealkoace* – složitější – přesun mezi seznamy, z děr do procesů
- Oddělené seznamy, seznam děr dle velikosti
  - Optimalizace best fitu
  - První vhodná – je i nejmenší vhodná, rychlost First fitu
  - Režie na dealokaci – sousední fyzické díry nemusí být sousední v seznamu

Další varianty – Quick Fit

Quick Fit

- Samostatné seznamy děr nejčastěji požadovaných délek
- Díry velikosti 4KB, 8KB,…
- Ostatní velikosti v samostatném seznamu
- Alokace – rychlá
- Dealokace – obtížné sdružování sousedů

Šetření paměti

- Místo samostatného seznamu děr lze využít díry
- Obsah díry
  - 1. slovo – velikost díry
  - 2. slovo – ukazatel na další díru

Např. alokátor paměti pro proces v jazyce C pod Unixem používá strategii next fit (viz ukázka malloc dříve)

KVIZ

Jaký je vzájemný poměr počtu děr a procesů?

Předpokládejme, že pro daný proces alokujeme paměť jednorázově (v celku)

Asymetrie mezi procesy a dírami

- Dvě sousední díry (H) se sloučí
- Dva procesy (P) se nesloučí

Při normálním běhu je počet děr poloviční oproti počtu procesů

Opakování

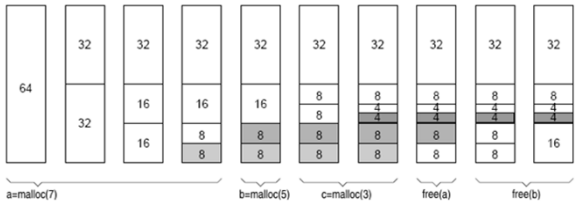
Správa paměti:

1. Bitové mapy
2. Seznamy (first fit, …)
3. Buddy systems

Buddy systems

- Seznamy volných bloků 1, 2, 4, 8, 16 … alokačních jednotek až po velikost celé paměti
- Nejprve seznamy prázdné vyjma 1 položky v seznamu o velikosti paměti
- Př.: Alokační jednotka 1KB, paměť velikosti 64KB
- Seznamy 1, 2, 4, 8, 16, 32, 64 (7 seznamů)
- Požadavek se zaokrouhlí na mocninu dvou nahoru
  - např. požadavek 7KB na 8KB
- Blok 64KB se rozdělí na 2 bloky 32KB (buddies) a dělíme dále…

Buddy system



Nejmenší dostatečně velký blok se rozdělí
Dva volné sousední bloky stejné velikosti (buddies) – spojí se do většího bloku

Buddy system

Neefektivní (plýtvání místem) x rychlý

- Chci 9KB, dostanu 16KB
- Alokace paměti – vyhledání v seznamu dostatečně velkých děr
- Slučování – vyhledání buddy

Použití algoritmů

U řady algoritmů můžeme pozorovat, že se nepoužívají ke svému „původnímu účelu“, tj. ke správě hlavní paměti, ale používají se pro řešení dílčích úkolů.

Např. runtimová knihovna požádá OS o přidělení stránky paměti, a získanou oblast dále přiděluje procesu, když si o ní zažádá funkcí malloc() – a zde se uplatní další strategie správy paměti

Použití algoritmů

- Přidělení paměti procesům
  – dnes mechanismy virtuální paměti
- Další oblasti použití
  – přidělování paměti uvnitř jádra nebo uvnitř procesu

Buddy system

Jádro Linuxu běží ve fyzické paměti, pro správu paměti jádra používá buddy system

viz: *cat /proc/buddyinfo*

Použití algoritmů

Použití first fit, next fit:

Fce malloc v jazyce C
žádá OS o větší blok paměti a získanou paměť pak aplikaci přiděluje algoritmem first fit či next fit

Správa odkládacího prostoru -
Linux spravuje odkládací prostor pomocí bitové mapy algoritmem next fit

09. Memory management

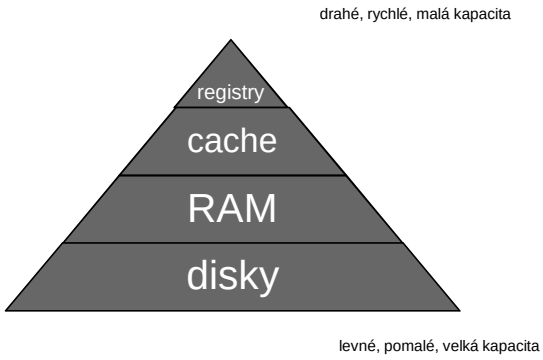
ZOS 2013, L. Pešička

Administrativa

- 2. zápočtový test
  11. 12. 2013 od 18:30 v EP130

Správa paměti

- „paměťová pyramida“
- absolutní adresa
- relativní adresa
  - *počet bytů od absolutní adresy (nějakého počátku)*
- fyzický prostor adres
  - fyzicky k dispozici výpočetnímu systému
- logický adresní prostor
  - využívají procesy



Modul pro správu paměti

- informace o přidělení paměti
  - která část je volná
  - přidělená (a kterému procesu)
- přidělování paměti na žádost
- uvolnění paměti, zařazení k volné paměti
- odebírá paměť procesům
- ochrana paměti
  - přístup k paměti jiného procesu
  - přístup k paměti OS

Memory management

- Základní mechanismy
  - Bez odkládání a stránkování
  - Jednoprogramové systémy
  - Multiprogramování s pevným přidělením paměti
  - Multiprogramování s proměnnou velikostí oblasti
  - Správa paměti
    - Bitové mapy
    - Seznamy
      - First fit, best fit, next fit
    - Buddy system

Celý proces se musí vejít do paměti

Opakování z minulé přednášky

Statická a dynamická relokace

Relokace a ochrana

- Problémy při multiprogramování (více programů současně v paměti):
- Relokace
  - Programy běží na různých (fyzických) adresách
  - jednou je ve fyzické paměti od adresy X, jindy od Y
- Ochrana
  - Paměť musí být chráněna před zasahováním jiných programů

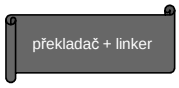
ukázka překladu .c programu

```
eryx.zcu.cz - PuTTY
eryx1> ls
main.c  makefile
eryx1> make
gcc -lpthread -O3 -c main.c
gcc -lpthread -O3 -o fork_sm main.o
eryx1> ls
fork_sm  main.c  main.o  makefile
eryx1>
eryx1> file main.o
main.o: ELF 32-bit LSB relocatable, Intel 80386, version 1 (SYSV), not stripped
eryx1>
eryx1> file fork_sm
fork_sm: ELF 32-bit LSB executable, Intel 80386, version 1 (SYSV), dynamically l
inked (uses shared libs), for GNU/Linux 2.6.8, not stripped
eryx1>
eryx1>
```

zdrojový soubor	main.c
objektový modul	main.o
spustitelný soubor	fork_sm

Relokace při zavedení do paměti

jak je program vytvořen a spuštěn:



- Překlad a sestavení programu
  - Aplikace ve vysokoúrovňovém jazyce
  - Větší SW – rozděleny do modulů – musejí být přeloženy a sestaveny do spustitelného programu
  - Objektové moduly
    - Výsledkem překladu
    - Příkazy ve zdrojovém textu – přeloženy do stroj. instrukcí
    - Zůstávají symbolické odkazy – adresy prom., procedur,fcí

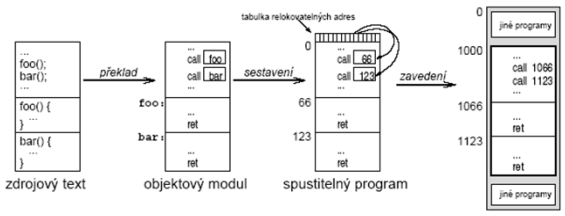
Relokace při zavedení do paměti 2

- Výsledný spustitelný program
  - Sestavení (linkování) modulů a knihoven
- Při sestavení se řeší hlavně externí reference
  - Všechna místa výskytů referencí – seznam
  - Když už je adresa známa – vloží se všude, kde se používá
  - Symbolické odkazy se převedou na číselné hodnoty
  - Výsledek – spustitelný program

Relokace při zavedení do paměti 3

- Komplikace při více programech v paměti
  - Příklad
  - 1. instrukcí programu volání podprogramu *call 66*
  - Program v paměti od adresy *1000*, ve skutečnosti provede call *1066*
- Jedno z řešení – modifikovat instrukce programu při zavedení do paměti
  - Linker – do spustitelného programu přidá seznam nebo bitmapu označující místa v kódu obsahující adresu
  - Při zavádění programu do paměti se každé adrese přičte adresa začátku oblasti

Relokace při zavedení do paměti



Statická relokace

- Popsanému způsobu se říká statická relokace
- Adresy se natvrdo přepíší správnými
- Např. OS/MFT od IBM

dále budou popsány mechanismy ochrany paměti:
- mechanismus přístupového klíče
- mechanismus báze a limitu

Ochrana – přístupový klíč

- Proces mohl zasahovat do paměti jiných procesů
- IBM 360 – **přístupový klíč**
  - Pamět rozdělena do bloků 2KB
  - Každý blok – sdružený hw 4 bitový kód ochrany
  - PSW procesoru obsahuje 4 bitový klíč
  - Při pokusu o přístup k paměti jejíž kód ochrany se liší od klíče PSW – výjimka
  - Kód ochrany a klíč může měnit jen OS (privilegované instrukce)
  - Výsledek – ochrana paměti

Klíč je spjatý s procesem

Možnou metodou ochrany paměti je ochrana přístupovým klíčem

Ochrana - mechanismus báze a limitu

- Jednotka správy paměti MMU (je uvnitř CPU)
- Dva registry – báze a limit
- Báze – počáteční adresa oblasti
- Limit – velikost oblasti

Mechanismus báze a limitu

- Funkce MMU
  - převádí adresu od procesu na adresu do fyzické paměti
  - Nejprve zkontroluje, zda adresa není větší než limit
    - Ano – výjimka, Ne – k adrese přičte bázi
- Pokud báze 1000, limit 60
  - Přístup na adresu 55 – ok, výsledek 1055
  - Přístup na adresu 66 – není ok, výjimka

Dynamická relokace

- Provádí se dynamicky za běhu
- patří sem uvedený mechanismus báze a limitu
- Nastavení báze a limitu může měnit pouze OS (privilegované instrukce)
- Např. 8086 – slabší varianta (nemá limit, jen báze)
- Bázové registry = segmentové registry DS,SS,CS,ES

Správa paměti s odkládáním celých procesů

(Proces se vejde do fyzické paměti)

Správa paměti s odkládáním celých procesů

- Pro dávkové systémy – dosud uvedené mechanismy - přiměřené (jednoduchost, efektivita)
- Systémy se sdílením času – víc procesů, než se jich vejde do paměti současně
- 2 strategie
  - Odkládání celých procesů (swapping)
    - Nadbytečný proces se odloží na disk
    - Např. UNIX Version 7; co platí pro velikost procesu?
  - Virtuální paměť – v paměti nemusí být procesy celé
    - Překrývání (overlays), virtuální paměť

Odkládání celých procesů

co víme o velikosti procesu?

- data procesu mohou růst
- pro proces alokováno o něco více paměti, než je třeba
- potřeba více paměti, než je alokováno:
  - přesunout proces do větší oblasti (díry)
  - překážející proces odložit – prostor pro růst procesu
  - odložit žadatele o paměť, dokud nebude prostor
  - proces zrušit (odkládací paměť je plná)

Odkládání celých procesů

- proces – dva rostoucí segmenty
  - data, zásobník (co se kde alokuje?)
  - možnost rozrůstání proti sobě
  - překročení velikosti – přesun, odložit, zrušit

Alokace odkládací oblasti

tj. jak vyhradit prostor pro proces na disku:

- na celou dobu běhu programu („pořád do stejného místa“)
- alokace při každém odložení

stejné algoritmy jako pro přidělení paměti

velikost oblasti na disku
– násobek alokační jednotky disku

Virtuální paměť

Proces > dostupná fyzická paměť

{ proces může být i větší
než dostupná fyzická paměť}

Virtuální paměť

- program větší než dostupná fyzická paměť
- mechanismus překrývání (overlays)
- virtuální paměť

Virtuální paměť je to, co se dnes nejčastěji používá

Překrývání (overlays)

- program – rozdělen na moduly
- start – spuštěna část 0, při skončení zavede část 1 …
- časté zavádění některých modulů
  - více překryvných modulů + data v paměti současně
  - moduly zaváděny dle potřeby (nejen 0,1,2,..)
  - mechanismus odkládání (jako odkládání procesů)
- kdo zařizuje zavádění modulů?
- kdo navrhuje rozdělení dat na moduly?

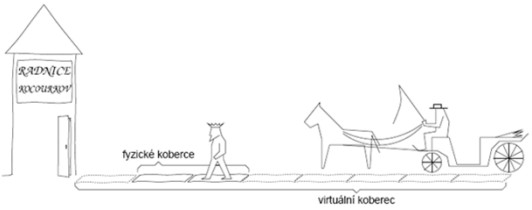
Překrývání

- zavádění modulů zařizuje OS
- rozdělení programů i dat na části – navrhuje programátor
  - vliv rozdělení na výkonnost, komplikované
  - pro každou úlohu nové rozdělení
- příklad – overlay.pas
- snaha, aby se o vše postaral OS

Virtuální paměť

- potřebujeme rozsáhlý adresový prostor
- ve skutečné paměti je pouze část adresového prostoru
  - jinak by to bylo příliš drahé
- zbytek může být odložen na disku
- kterou část mít ve fyzické paměti?
  - tu co právě potřebujeme ☺

Historie – královský koberec



Na pokrytí celé cesty stačí pouze dva fyzické koberce

Virtuální adresy

- fyzická paměť slouží jako cache virtuálního adresního prostoru procesů (!)
- procesor – používá virtuální adresy
- Pokud požadovaná část VAProstoru JE ve fyzické paměti
  - MMU převede VA=>FA, přístup k paměti
- požadovaná část NENÍ ve fyzické paměti
  - OS ji musí přečíst z disku
  - I/O operace – přidělení CPU jinému procesu
- většina systémů virtuální paměti používá stránkování

Mechanismus stránkování (paging)

- program používá virtuální adresy
- Musíme rychle zjistit, zda je požadovaná adresa v paměti
  - ANO – převod VA => FA
- co nejrychlejší – děje se při každém přístupu do paměti

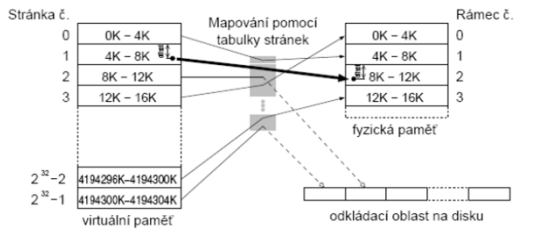
Pojmy – důležité !!!

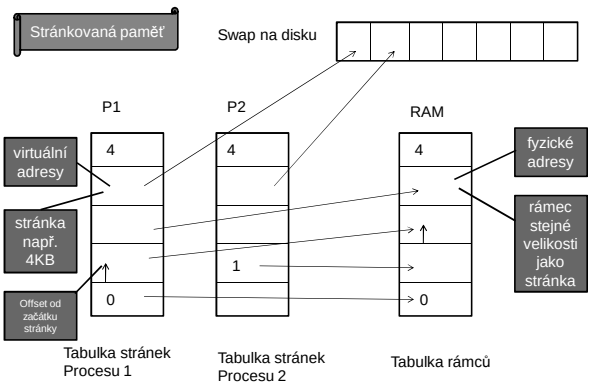
- VAP – stránky (pages) pevné délky
  - délka mocnina 2, nejčastěji 4KB, běžně 512B - 8KB
- fyzická paměť – rámce (page frames) stejné délky
- rámec může obsahovat PRÁVĚ JEDNU stránku
- na známém místě v paměti – tabulka stránek
- tabulka stránek poskytuje mapování virtuálních stránek na rámce

Opakování

- virtuální adresní prostor
- fyzický adresní prostor
- procesy používají VA nebo FA?
- co dělá MMU?
- k čemu slouží tabulka stránek?
- stránka
- rámec

Stránky jsou mapovány na rámce v RAM, nebo jsou uložené v odkládací paměti na disku





Tabulka stránek procesu: 1
Velikost stránky: 4096 B

stránka	rámec	další atributy
0	0	
1	2	
2	3	
3	x	swap: 0
4		

Pokud bychom počítali fyzické adresy pro proces 2, používali bychom tabulku stránek procesu 2

Je dána VA 500, vypočítejte fyzickou adresu.
Je dána VA 12300, vypočítejte fyzickou adresu ☺

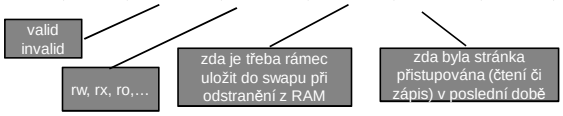
Je dána VA 4099:
4099 / 4096 = 1, offset 3
Tabulka_stranek_naseho_procesu [1] = 2 .. druhý rámec
FA = 2 * 4096 + 3 = 8195

Výpadek stránky:

Stránka není v operační paměti, ale ve swapu na disku

Tabulka stránek - podrobněji

Číslo stránky	Číslo rámce	příznak platnosti	Příznaky ochrany	Bit modifikace (dirty)	Bit referenced	Adresa ve swapu
0	**3**	valid	rx	1	1	---
1	**4**	valid	rw	1	1	---
2	**---**	invalid	ro	0	0	4096



Tabulka stránek (TS) - podrobněji

- součástí PCB (tabulka procesů) – kde leží jeho TS
- velikost záznamu v TS .. 32 bitů
- číslo rámce .. 20 bitů

Výpočet adresy - stránkování

Pojmy:

VA	virtuální adresa
FA	fyzická adresa
str	číslo stránky
offset	offset
ramec	číslo rámce

Dále předpokládáme velikost stránky 4096B

Příklad s uvedením výpočtu

Je dána VA(p1) = 100. Určte FA.
Velikost stránky je 4096 bytů (4KB).
Tabulka stránek procesu p1 je následující:

Číslo stránky	rámec
0	1
1	2
2	---
3	0

Nezapomeň: máme-li více procesů, každý má svojí tabulku stránek.

Výpočet adresy – stránkování

1. Virtuální adresu rozdělíme na číslo stránky a offset
   - Str = VA div 4096 (dělení, 4096 je velikost stránky)
   - Offset = VA mod 4096 (zbytek po dělení)
2. Převod pomocí tabulky stránek
   převedeme číslo stránky na číslo rámce
   - **tab_str[0] = 1** (pro stránku 0 je číslo rámce 1)
   - tab_str[1] = 2
   - tab_str[2] = -- stránka není namapována
   - tab_str[3] = 0
   - Pro VA = 100 je stránka 0, offset 100 => tedy rámec 1

Výpočet adresy - stránkování

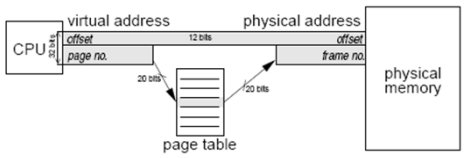
3. Z čísla rámce a offsetu sestavíme fyzickou adresu:

FA = ramec*4096 + offset
FA= 1*4096+ 100
FA= 4196 v daném případě

tedy žádné složité dělení není třeba, vezmou se nižší a vyšší bity tj. adresní vodiče

V reálném systému dělení znamená rozdělení na vyšší a nižší bity adresy (proto mocnina dvou velikost str.)
Nižší bity – offset
Vyšší bity – číslo stránky

Stránkování



32 bit adresa – 20 bitů číslo stránky, 12 bitů offset
Offset zůstává beze změny

Výpadek stránky (!!!)

- viz příklad, pro adresu 8192 str 2, offset 0
- Výpadek stránky
  - Stránka není mapována
  - Výpadek stránky způsobí **výjimku**, zachycena OS (pomocí **přerušení**)
  - OS iniciuje zavádění stránky a přepne na jiný proces
  - Po zavedení stránky OS upraví mapování (tabulku stránek)
  - Proces může pokračovat
  - Vyřešit: KAM stránku zavést a ODKUD ?

Výpadek stránky

Pokud daná stránka procesu není namapována na určitý rámec ve fyzické paměti a chceme k ní přistoupit

dojde k výpadku stránky – vyvolání **přerušení** operačního systému.

Operační systém se postará o to, aby danou stránku zavedl do nějakého rámce ve fyzické paměti, nastavil mapování a poté může přístup proběhnout.

Náročnost

- Velký rozsah tabulky stránek
  - Např. 1 milion stránek, ne všechny obsazeny
- Rychlý přístup
  - Nemůžeme pokaždé přistupovat k tabulce stránek
  - Různá HW řešení, kopie části tabulky v MMU …

Tabulka stránek může být velmi rozsáhlá – pro urychlení např. kopie části tabulky stránky v MMU (memory management unit)

Vnější fragmentace

- Vnější / externí
  - Zůstávají nepřidělené (nepřidělitelné) úseky paměti
  - Např. dynamické přidělování – malé díry

Při stránkování vnější fragmentace nenastává, všechny stránky jsou přidělitelné (jsou stejně velké)

Vnitřní fragmentace

- Vnitřní fragmentace
  - Část přidělené oblasti je nevyužita (dostaneme přidělenou stránku, ale využijeme z ní jen část !)

Stránkování:

V průměru polovina poslední stránky procesu je prázdná

Čisté stránkování

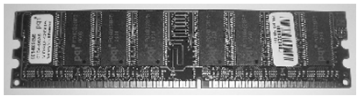
Bez odkládací oblasti

Souvislý logický adresní prostor procesu mapován do
nesouvislých částí paměti

OS udržuje:
- 1 tabulka rámců
- Tabulku stránek pro každý proces

Tabulka rámců

- Pro správu FYZICKÉ paměti
- Je třeba informace, které rámce jsou volné vs. obsazené

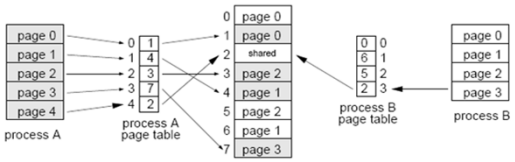


zdroj obrázku: http://www.lisak.cz/pocitac-jako-skladacka.html

Tabulka stránek procesu

- Mapuje číslo stránky na číslo fyzického rámce
- Další informace – např. příznaky ochrany
- Řeší problém relokace a ochrany
  - Relokace – mapování VA na FA
  - Ochrana – v tabulce stránek pouze stránky, ke kterým má proces přístup
- Přepnutí na jiný proces
  - MMU přepne na jinou tabulku stránek

Stránkování



Stránkování umožňuje i přístup do sdílené paměti, v každém procesu může být dokonce sdílená paměť mapována od jiné adresy

Problémy

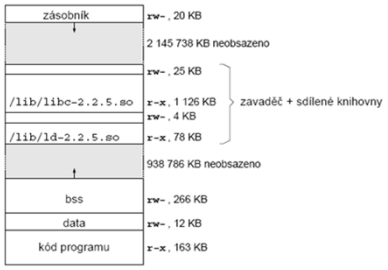
- Velikost tabulky stránek
  - Pomůže víceúrovňová struktura
- Rychlost převodu VA -> FA
  - TLB (Transaction Look-aside Buffer)

na dalších slidech budou tyto problémy dále rozebrány

Velikost tabulky stránek

- VA 32 bitů
  - stránka 4KB (12 bitů)
  - Stránek 2^20 (20 bitů)
    - Každá položka 4B .. 2^20*4 = 4MB celkem pro **každý** proces
- Proces využívá jen část VA
  - Kód
  - Data (inicializovaná, a neinicializovaná)
  - Sdílené knihovny a jejich data
  - Od nejvyšší adresy zásobník - roste dolů

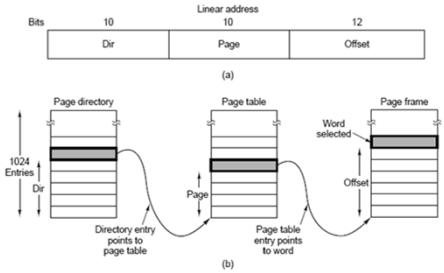
Rozdělení paměti pro proces



Velikost tabulky stránek

- Mít v tabulce stránek jen ty, představující existující paměť => víceúrovňová tabulka stránek
- VA 32 bitů
  - PT1 – 10 bitů , index do tab. stránek 1. úrovně
  - PT2 – 10 bitů, index do tab. stránek 2. úrovně
  - Offset – 12bitů
- PT1=0 (kód a data), PT1=1 (sdílené knihovny) PT=1023 (zásobník); ostatní nepřiřazeno!

Velikost tabulky stránek



Rychlost převodu (!)

- Každý přístup – sáhne do tabulky stránek
  - 2x více paměťových přístupů
    - musíme sáhnout do tabulky stránek a pak do paměti kam chceme
- TLB (Transaction Look-aside Buffer) (!!!!)
  - HW cache
  - Dosáhneme zpomalení jen 5 až 10 %
  - Přepnutí kontextu na jiný proces
    - problém (vymazání cache,..)
    - než se TLB opět zaplní – pomalý přístup

Obsah položky v tabulce stránek (!!!)

- Číslo rámce
- Příznak platnosti (valid / invalid)
- Příznaky ochrany (rw, ro, ..)
- Bit modified (dirty)
  - zápis do stránky nastaví na 1
- Bit referenced
  - Přístup pro čtení / zápis nastaví na 1
- Další …

Invertovaná tabulka stránek

- VA 64bitů , stránka 4KB, 2^52 stránek – moc

Invertovaná tabulka stránek

- Položky pro každý fyzický rámec
  - Omezený počet – dán velikostí RAM
  - VA 64bitů, 4KB stránky, 256MB RAM – 65536 položek
- Forma položky: (id procesu, číslo stránky)

Invertovaná tabulka stránek - převod

- Pokud je položka v TLB
  – zařídí HW, jinak OS (SW)

SW:

- Prohledávání invertované tabulky stránek
- Položka nalezena – (číslo stránky, číslo rámce) do TLB
- Tabulka hashovaná podle virtuální adresy (pro optim.)

Stránkování na žádost (už odkládací prostor)

- Vytvoření procesu
  - Vytvoří prázdnou tabulku stránek
  - Alokace místa na disku pro odkládání stránek
  - Některé implementace – odkládací oblast inicializuje kódem programu a daty ze spustitelného souboru
- Při běhu
  - Žádná stránka v paměti,
  - 1. přístup – výpadek stránky (page fault)
  - OS zavede požadovanou stránku do paměti
  - Postupně v paměti tzv. pracovní množina stránek

Pracovní množina stránek

Má-li proces svou pracovní množinu stránek v paměti, může pracovat bez mnoha výpadků

dokud se pracovní množina stránek nezmění, např. další fáze výpočtu

Pracovní množina stránek daného procesu – kolik stránek musí mít ve fyzické paměti, aby mohl nějaký čas pracovat bez výpadků stránky

Ošetření výpadku stránky (důležité !)

1. Výpadek – mechanismem přerušení (!!) vyvolán OS
2. OS zjistí, pro kterou stránku nastal výpadek
3. OS určí umístění stránky na disku
   - Často tato informace přímo v tabulce stránek
4. Najde rámec, do kterého bude stránka zavedena
   - Co když jsou všechny rámce obsazené?
5. Načte požadovanou stránku do rámce
6. Změní odpovídající položku v tabulce stránek
7. Návrat..
8. HW dokončí instrukce, která způsobila výpadek

Problém

- Všechny rámce obsazené, kterou stránku vyhodit ??

Algoritmy nahrazování stránek

Všechny rámce v paměti RAM jsou plné. Přesto musíme nějaký z nich uvolnit (odložit na disk), abychom mohli do RAM dát ten, který potřebujeme. Jak rozhodnout, který rámec vyhodit?

Algoritmy nahrazování stránek

- Uvolnit rámec pro stránku, co s původní stránkou?
- Pokud byla stránka modifikována (dirty=1), uložit na disk
- Pokud oproti kopii na disku nebyla modifikována, pouze uvolněna

Algoritmy nahrazování stránek

- Kterou stránku vyhodit?

Takovou, která se dlouho nebude potřebovat..

Chtělo by křišťálovou kouli…

Algoritmus FIFO

- Udržovat seznam stránek v pořadí, ve kterém byly zavedeny
- Vyhazujeme nejstarší stránku (nejdéle zavedenou – první na seznamu)

Není nejvhodnější

Často používané stránky mohou být v paměti dlouho
(analogie s obchodem, nejdéle zavedený výrobek – chleba)

Trpí Beladyho anomálií

Beladyho anomálie

Předpokládáme:

Čím více bude rámců paměti, tím nastane méně výpadků.

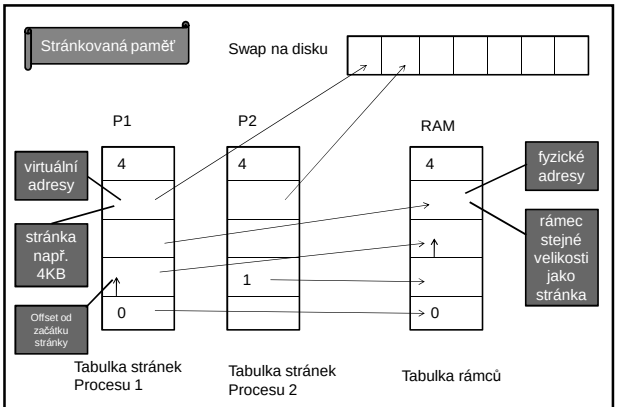
Belady našel příklad pro algoritmus FIFO, kdy to neplatí.

```
* algoritmus FIFO, řetězec odkazů (referencí): 0 1 2 3 0 1 4 0 1 2 3 4

3 rámce:   ref.:0 1 2 3 0 1 4 0 1 2 3 4
           ---------------------------
           1| . 0 1 2 3 0 1 4 4 4 2 3 3
           2| . . 0 1 2 3 0 1 1 1 4 2 2
           3| . . . 0 1 2 3 0 0 0 1 4 4
           ---------------------------
                P P P P P P P     P P = 9 výpadků

4 rámce:   ref.:0 1 2 3 0 1 4 0 1 2 3 4
           ---------------------------
           1| . 0 1 2 3 3 3 4 0 1 2 3 4
           2| . . 0 1 2 2 2 3 4 0 1 2 3
           3| . . . 0 1 1 1 2 3 4 0 1 2
           4| . . . . 0 0 0 1 2 3 4 0 1
           ---------------------------
                P P P P     P P P P P P P = 10 výpadků

* tj. pro 3 rámce nastane 9 výpadků, pro 4 rámce 10 výpadků
* objev pana Beladyho způsobil vývoj teorie stránkovacích algoritmů a
  jejich vlastností
```

10.

Memory management II.

ZOS 2013, L. Pešička



Tabulka stránek procesu: 1
Velikost stránky: 4096 B

stránka	rámec	další atributy
0	0	
1	2	
2	3	
3	x	swap: 0
4		

Pokud bychom počítali fyzické adresy pro proces 2, používali bychom tabulku stránek procesu 2

Je dána VA 500, vypočítejte fyzickou adresu.
Je dána VA 12300, vypočítejte fyzickou adresu ☺

Je dána VA 4099:
4099 / 4096 = 1, offset 3
Tabulka_stranek_naseho_procesu [1] = 2 .. druhý rámec
FA = 2 * 4096 + 3 = 8195

Výpadek stránky:

Stránka není v operační paměti, ale ve swapu na disku

Tabulka stránek - podrobněji

Číslo stránky	Číslo rámce	příznak platnosti	Příznaky ochrany	Bit modifikace (dirty)	Bit referenced	Adresa ve swapu
0	**3**	valid	rx	1	1	---
1	**4**	valid	rw	1	1	---
2	**---**	invalid	ro	0	0	4096

valid invalid

rw, rx, ro,…

zda je třeba rámec uložit do swapu při odstranění z RAM

zda byla stránka přistupována (čtení či zápis) v poslední době

Tabulka stránek (TS) - podrobněji

- součástí PCB (tabulka procesů) – kde leží jeho TS
- velikost záznamu v TS .. 32 bitů
- číslo rámce .. 20 bitů

dvouúrovňová tabulka stránek
- 4KB, 4MB

čyřúrovňová tabulka stránek x86-64
- stránky 4KB, 2MB, až 1GB

Obsah

- FIFO + Beladyho anom.
- MIN / OPT
- LRU
- NRU
- Second Chance, Clock
- Aging

- Segmentování
- I/O

Algoritmy nahrazování stránek paměti

Použijí se, pokud potřebujeme uvolnit místo v operační paměti pro další stránku:

nastal výpadek stránky, je třeba někam do RAM zavést stránku a RAM je plná..

nějakou stránku musíme z RAM odstranit, ale jakou?

Algoritmus MIN / OPT

- optimální – nejmenší možný výpadek stránek
- Vyhodíme zboží, které nejdelší dobu nikdo nebude požadovat.

- stránka označena počtem instrukcí, po který se k ní nebude přistupovat
- p[0] = 5, p[1] = 20, p[3] = 100
- výpadek stránky – vybere s nejvyšším označením
- vybere se stránka, která bude zapotřebí nejpozději v **budoucnosti**

MIN / OPT

- není realizovatelný (křišťálová kole)
  - jak bychom zjistili dopředu která stránka bude potřeba?
- algoritmus pouze pro srovnání s realizovatelnými
- Použití pro běh programu v simulátoru
  - uchovávají se odkazy na stránky
  - spočte se počet výpadků pro MIN/OPT
  - Srovnání s jiným algoritmem (o kolik je jiný horší)

Least Recently Used (LRU)

- nejdéle nepoužitá (pohled do minulosti)
- princip lokality
  - stránky používané v posledních instrukcích se budou pravděpodobně používat i v následujících
  - pokud se stránka dlouho nepoužívala, pravděpodobně nebude brzy zapotřebí
- Vyhazovat zboží, na kterém je v prodejně nejvíce prachu = nejdéle nebylo požadováno

LRU

- obtížná implementace
- sw řešení (není použitelné)
  - seznam stránek v pořadí referencí
  - výpadek – vyhození stránky ze začátku seznamu
  - zpomalení cca 10x, nutná podpora hw

LRU – HW řešení - čítač

- HW řešení – čítač
  - MMU obsahuje čítač (64bit), při každém přístupu do paměti zvětšen
  - každá položka v tabulce stránek – pole pro uložení čítače
  - odkaz do paměti:
    - obsah čítače se zapíše do položky pro odkazovanou stránku
  - výpadek stránky:
    - vyhodí se stránka s nejnižším číslem

LRU – HW řešení - matice

- MMU udržuje matici n * n bitů
  - n – počet rámců
- všechny prvky 0
- odkaz na stránku odpovídající k-tému rámci
  - všechny bity k-tého řádku matice na 1
  - všechny bity k-tého sloupce matice na 0
- řádek s nejnižší binární hodnotou
  - nejdéle nepoužitá stránka

LRU – matice - příklad

reference v pořadí: 3 2 1 0

0.1.2.3	0.1.2.3	0.1.2.3	0.1.2.3
0. 0 0 0 0	0. 0 0 0 0	0. 0 0 0 0	0. 0 1 1 1
1. 0 0 0 0	1. 0 0 0 0	1. 1 0 1 1	1. 0 0 1 1
2. 0 0 0 0	2. 1 1 0 1	2. 1 0 0 1	2. 0 0 0 1
3. 1 1 1 0	3. 1 1 0 0	3. 1 0 0 0	3. 0 0 0 0

LRU - vlastnosti

- výhody
  - z časově založených (realizovatelných) nejlepší
  - Beladyho anomálie nemůže nastat
- nevýhody
  - každý odkaz na stránku – aktualizace záznamu (zpomalení)
    - položka v tab. stránek
    - řádek a sloupec v matici
- LRU se pro stránkovanou virtuální paměť příliš nepoužívá
- LRU ale např. pro blokovou cache souborů

Not-Recently-Used (NRU)

- snaha vyhazovat nepoužívané stránky
- HW podpora:
  - stavové bity Referenced (R) a Dirty (M = modified)
  - v tabulce stránek
- bity nastavované HW dle způsobu přístupu ke stránce
- bit R – nastaven na 1 při čtení nebo zápisu do stránky
- bit M – na 1 při zápisu do stránky
  - stránku je třeba při vyhození zapsat na disk
- bit zůstane na 1, dokud ho SW nenastaví zpět na 0

algoritmus NRU

- začátek – všechny stránky R=0, M=0
- bit R nastavován OS periodicky na 0 (přerušení čas.)
  - odliší stránky referencované **v poslední době !!**
- 4 kategorie stránek (R,M)
  - *třída 0: R = 0, M = 0*
  - *třída 1: R = 0, M = 1* -- z třídy 3 po nulování R
  - *třída 2: R = 1, M = 0*
  - *třída 3: R = 1, M = 1*
- NRU vyhodí stránku z nejnižší neprázdné třídy
- výběr mezi stránkami ve stejné třídě je náhodný

NRU

- pro NRU platí – lepší je vyhodit modifikovanou stránku, která nebyla použita 1 tik, než nemodifikovanou stránku, která se právě používá
- výhody
  - jednoduchost, srozumitelnost
  - efektivně implementovaný
- nevýhody
  - výkonnost (jsou i lepší algoritmy)

Náhrada bitů R a M - úvaha

jak by šlo simulovat R,M bez HW podpory?

- start procesu – všechny stránky jako nepřítomné v paměti
- odkaz na stránku – výpadek
  - OS interně nastaví R=1
  - nastaví mapování stránky v READ ONLY režimu
- pokus o zápis do stránky – výjimka
  - OS zachytí a nastaví M=1,
  - změní přístup na READ WRITE

Algoritmy Second Chance a Clock

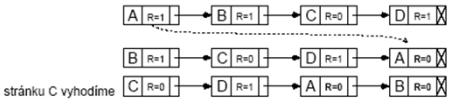
- vycházejí z FIFO
  - FIFO – obchod vyhazuje zboží zavedené před nejdelší dobou, ať už ho někdo chce nebo ne
  - Second Chance – evidovat, jestli zboží v poslední době někdo koupil (ano – prohlásíme za čerstvé zboží)
- modifikace FIFO – zabránit vyhození často používané

Second Chance

algoritmus Second Chance

- dle bitu R (referenced) nejstarší stránky
  - R = 0 ... stránka je nejstarší, nepoužívaná – vyhodíme
  - R = 1 ... nastavíme R=0, přesuneme na konec seznamu stránek (jako by byla nově zavedena)

Příklad Second Chance



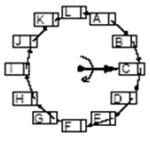
1. Krok – nejstarší je A, má R = 1 – nastavíme R na 0 a přesuneme na konec seznamu
2. Druhá nejstarší je B, má R = 1 – nastavíme R na 0 a opět přesuneme na konec seznamu
3. Další nejstarší je C, R = 0 – vyhodíme ji

Second Chance

- SC vyhledá nejstarší stránku, která nebyla referencována v poslední době
- Pokud všechny referencovány – čisté FIFO
  - Všem se postupně nastaví R na 0 a na konec seznamu
  - Dostaneme se opět na A, nyní s R = 0, vyhodíme ji
- Algoritmus končí nejvýše po (počet rámců + 1) krocích

Algoritmus Clock

- Optimalizace datových struktur algoritmu Second Chance
  - Stránky udržovány v **kruhovém** seznamu
  - Ukazatel na nejstarší stránku – „ručička hodin“



Výpadek stránky – najít stránku k vyhození

Stránka kam ukazuje ručička

- má-li R=0, stránku vyhodíme a ručičku posuneme o jednu pozici
- má-li R=1, nastavíme R na 0, ručičku posuneme o 1 pozici, opakování,..

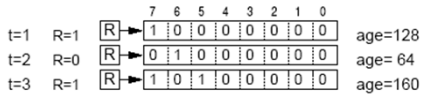
Od SC se liší pouze implementací

Varianty Clock používají např. BSD UNIX

SW aproximace LRU - Aging

- LRU vyhazuje vždy nejdéle nepoužitou stránku
- Algoritmus Aging
  - Každá položka tabulky stránek – pole stáří (age), N bitů (8)
  - Na počátku age = 0
  - Při každém **přerušení časovače** pro **každou** stránku:
    - Posun pole stáří o 1 bit vpravo
    - Zleva se přidá hodnota bitu R
    - Nastavení R na 0
- Při výpadku se vyhodí stránka, jejíž pole age má nejnižší hodnotu

Aging



Age := age shr 1; posun o 1 bit vpravo

Age := age or (R shl N-1); zleva se přidá hodnota bitu R

R := 0; nastavení R na 0

Aging x LRU

- Několik stránek může mít stejnou hodnotu age a nevíme, která byla odkazovaná dříve (u LRU jasné vždy) – hrubé rozlišení (po ticích časovače)
- Age se může snížit na 0
  - nevíme, zda odkazovaná před 9ti nebo 1000ci tiky časovače
  - Uchovává pouze omezenou historii
  - V praxi není problém – tik 20ms, N=8, nebyla odkazována 160ms – nejspíše není tak důležitá, můžeme jí vyhodit
- stránky se stejnou hodnotou age – vybereme náhodně

Shrnutí algoritmů

důležité je uvědomit si, kdy tyto algoritmy zafungují – potřebujeme v RAM uvolnit rámec

- Optimální algoritmus (MIN čili OPT)
  - Nelze implementovat, vhodný pro srovnání
- FIFO
  - Vyhazuje nejstarší stránku
  - Jednoduchý, ale je schopen vyhodit důležité stránky
  - Trpí Beladyho anomálií
- LRU (Least Recently Used)
  - Výborný
  - Implementace vyžaduje spec. hardware, proto používán zřídka

Shrnutí algoritmů II.

- NRU (Not Recently Used)
  - Rozděluje stránky do 4 kategorií dle bitů R a M
  - Efektivita není příliš velká, přesto používán
- Second Chance a Clock
  - Vycházejí z FIFO, před vyhození zkotrolují, zda se stránka používala
  - Mnohem lepší než FIFO
  - Používané algoritmy (některé varianty UNIXu)
- Aging
  - Dobře aproximuje LRU – efektivní
  - Často prakticky používaný algoritmus

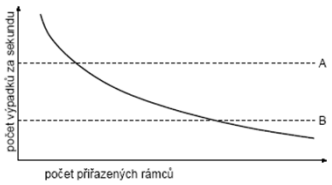
Ostatní problémy stránkované VP

- Alokace fyzických rámců
  - Globální a lokální alokace
  - Globální – pro vyhození se uvažují všechny rámce
    - Lepší průchodnost systému – častější
    - Na běh procesu má vliv chování ostatních procesů
  - Lokální – uvažují se pouze rámce alokované procesem (tj. obsahující stránky procesu, jehož výpadek stránky se obsluhuje)
    - Počet stránek alokovaných pro proces se nemění
    - Program se vzhledem k stránkování chová přibližně stejně při každém běhu

Lokální alokace

- Kolik rámců dát každému procesu?
- Nejjednodušší – všem procesům dát stejně
  - Ale potřeby procesů jsou různé
- Proprocionální – každému proporcionální díl podle velikosti procesu
- Nejlepší – podle frekvence výpadků stránek (Page Fault Frequency, PFF)
  - Pro většinu rozumných algoritmů se PFF snižuje s množstvím přidělených rámců

Page Fault Frequency (PFF)



PFF udržet v roz. mezích:
if PFF > A
přidáme procesu rámce
if PFF < B
proces má asi příliš paměti
rámce mu mohou být
odebrány

Zloděj stránek (page daemon)

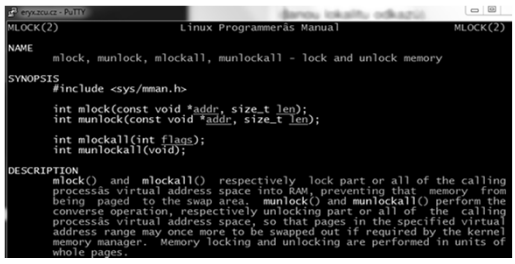
- v systému se běžně udržuje určitý počet volných rámců
- když klesne pod určitou mez, pustí page daemon (zloděj stránek), ten uvolní určité množství stránek (rámců)
- když se čerstvě uvolněné stránky hned nepřidělí, lze je v případě potřeby snadno vrátit příslušnému procesu

Zamykání stránek

zabrání odložení stránky

- části jádra
- stránka, kde probíhá I/O
- tabulky stránek
- nastavení uživatelem – mlock() , viz man 2 mlock

mlock



Zahlcení

- Proces pro svůj rozumný běh potřebuje pracovní množinu stránek
- Pokus se pracovní množiny stránek aktivních procesů nevejdou do paměti, nastane zahlcení (trashing)
- **Zahlcení**
  - V procesu nastane výpadek stránky
  - Paměť je plná (není volný rámec) – je třeba nějakou stránku vyhodit, stránka pro vyhození bude ale brzo zapotřebí, bude se muset vyhodit jiná používaná stránka …
- Uživatel pozoruje – systém intenzivně pracuje s diskem a běh procesů se řádově zpomalí (víc času stránkování než běh)
- Řešení – při zahlcení snížit úroveň multiprogramování (zahlcení lze detekovat pomocí PFF)

Mechanismus VP - výhody

- Rozsah virtuální paměti
  - (32bit: 2GB pro proces + 2GB pro systém, nebo 3+1)
  - Adresový prostor úlohy není omezen velikostí fyzické paměti
  - Multiprogramování – není omezeno rozsahem fyz. paměti
- Efektivnější využití fyzické paměti
  - Není vnější fragmentace paměti
  - Nepoužívané části adresního prostoru úlohy nemusejí být ve fyzické paměti

Mechanismus VP - nevýhody

- Režie při převodu virt. adres na fyzické adresy
- Režie procesoru
  - údržba tabulek stránek a tabulky rámců
  - výběr stránky pro vyhození, plánování I/O
- Režie I/O při čtení/zápisu stránky
- Paměťový prostor pro tabulky stránek
  - Tabulky stránek v RAM, často používaná část v TLB
- Vnitřní fragmentace
  - Přidělená stránka nemusí být plně využita

Rozdělení paměti pro proces (!!!)

```
pokus.c:

int x =5;  int y = 7;    // inic. data

void fce1() {
   int pom1, pom2; // na zásobníku
    … }

int main (void) {
   …
   malloc(1000);     // halda
   fce1();
   return 0;
}
```

2+2: 0..2GB proces, 2GB..4GB OS 3+1: 3GB proces, 1GB OS

Rozdělení paměti pro proces

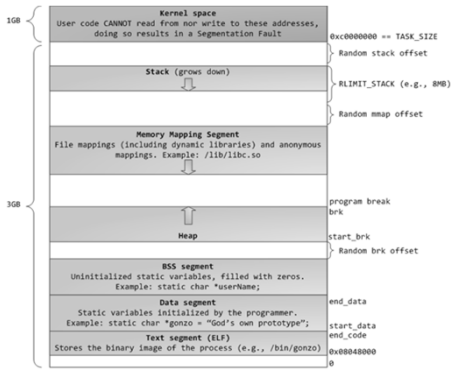
Roste halda ⟶ ⟵ Roste zásobník

Máme-li více vláken => více zásobníků, limit velikosti zásobníku

zásobník dalšího vlákna

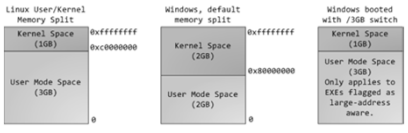
Další ukázka rozdělení paměti

zapamatovat pojem BSS

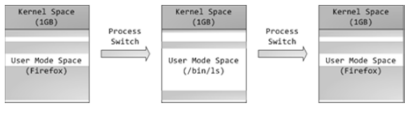


zdroj: http://duartes.org/gustavo/blog/category/linux DOPORUČUJI !!

Rozdělení paměti



rozdělení paměti Linux, Windows



přepínání kontextu mezi různými procesy

zdroj: http://duartes.org/gustavo/blog/category/linux

Segmentace

- Dosud diskutovaná VP – jednorozměrná
  - Proces: adresy < 0, maximální virtuální adresa>
- Často výhodnější – více samostatných virtuálních adresových prostorů
- Př. – máme několik tabulek a chceme, aby jejich velikost mohla růst
- Paměť nejlépe více nezávislých adresových prostorů - segmenty

Segmentace

- Segment – logické seskupení informací
- Každý segment – lineární posloupnost adres od 0
- Programátor o segmentech ví, používá je explicitně (adresuje konkrétní segment)
- Např. překladač jazyka – samostatné segmenty pro
  - Kód přeloženého programu
  - Globální proměnné
  - Hromada
  - Zásobník návratových adres
- Možné i jemnější dělení – segment pro každou funkci

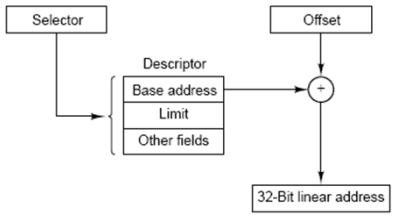
Segmentace

- Lze použít pro implementaci
  - Přístup k souborům
    - 1 soubor = 1 segment
    - Není třeba open, read ..
  - Sdílené knihovny
    - Programy využívají rozsáhlé knihovny
    - Vložit knihovnu do segmentu a sdílet mezi více programy
- Každý segment – logická entita – má smysl, aby měl samostatnou ochranu

Čistá segmentace

- Každý odkaz do paměti – dvojice **(selektor, offset)**
  - Selektor – číslo segmentu, určuje segment
  - Offset – relativní adresa v rámci segmentu
- Technické prostředky musí umět přemapovat dvojici (selektor, offset) na lineární adresu (fyzická když není dále stránkování)
- **Tabulka segmentů** – každá položka má
  - Počáteční adresa segmentu (báze)
  - Rozsah segmentu (limit)
  - Příznaky ochrany segmentu (čtení,zápis, provádění – rwx)

(selektor, offset) => lineární adresa



Převod na fyzickou adresu

- PCB obsahuje odkaz na tabulku segmentů procesu
- Odkaz do paměti má tvar (selektor, offset)
- Často možnost sdílet segment mezi více procesy

příklad instrukce: *LD R, sel:offset*

1. Selektor – index do tabulky segmentů
2. Kontrola offset < limit, ne – porušení ochrany paměti
3. Kontrola zda dovolený způsob použití; ne – chyba
4. Adresa = báze + offset

Segmentace

- Mnoho věcí podobných jako přidělování paměti po sekcích, ale rozdíl:
  - Po sekcích – pro procesy
  - Segmenty – pro části procesu
- Stejné problémy jako přidělování paměti po sekcích
  - Externí fragmentace paměti
  - Mohou zůstávat malé díry

Segmentace na žádost

- Segment – zavedený v paměti nebo odložený na disku
- Adresování segmentu co není v paměti – výpadek segmentu – zavede do paměti – není-li místo – jiný segment odložen na disk
- HW podpora – bity v tabulce segmentů
  - Bit segment je zaveden v paměti (Present / absent)
  - Bit referenced
- Používal např. systém OS/2 pro i80286 – pro výběr segmentu k odložení algoritmus Second Chance

Segmentace se stránkováním

- velké segmenty – nepraktické celé udržovat v paměti
- Myšlenka stránkování segmentů
  - V paměti pouze potřebné stránky
- Implementace – např. každý segment vlastní tabulka stránek

Adresy (!!!)

virtuální adresa -> lineární adresa -> fyzická adresa

virtuální – používá proces

lineární – po segmentaci
pokud není dále stránkování, tak už představuje i fyzickou adresu

fyzická – adresa do fyzické paměti RAM
(CPU jí vystaví na sběrnici)

Dnešní Intel procesory

- segmentace
- stránkování
- segmentace se stránkováním

využití LDT tabulek často operační systémy nahrazují stránkováním

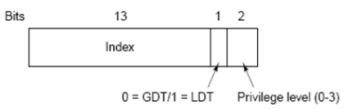
- tabulka LDT (Local Descriptor Table)
  - segmenty lokální pro proces (kód,data,zásobník)
- tabulka GDT (Global Descriptor Table)
  - pouze jedna, sdílená všemi procesy
  - systémové segmenty, včetně OS

Segmentové registry

- Pentium a výše má 6 segmentových registrů:
  - CS (Code Segment)
  - DS (Data Segment)
  - SS (Stack Segment)
  - další: ES, FS, GS
- přístup do segmentu – do segmentového registru se zavede selektor segmentu

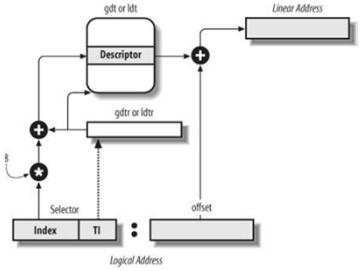
Selektor segmentu

- Selektor – 16bitový
- 13bitů – index to GDT nebo LDT
- 1 bit – 0=GDT, 1=LDT
- 2 bity – úroveň privilegovanosti (0-3)



Selektor segmentu

- 13 bitů – index, tj. max 2^13 = 8192 položek
  - selektor 0 – indikace nedostupnosti segmentu
- při zavedení selektoru do segmentového registru CPU také zavede odpovídající popisovač z LDT nebo GDT do vnitřních registrů CPU
  - bit 2 selektoru – pozná, zda LDT nebo GDT
  - popisovač segmentu na adrese (selektor and 0fff8h) + zač. tabulky

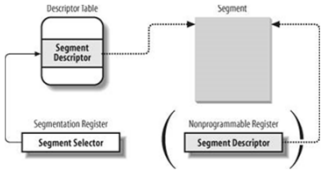


1. z TI pozná, zda použije GDT nebo LDT
2. z indexu selektoru spočte adresu deskriptoru
3. přidá offset k bázi (viz deskriptor), získá lineární adresu

neprogramovatelné registry spojené se segmentovými registry

zdroj: http://www.makelinux.net/books/ulk3/understandlk-CHP-2-SECT-2

Rychlý přístup k deskriptoru segmentu



logická adresa:
segment selektor + offset
(16bitů) (32bitů)

zrychlení převodu:
přídavné neprogramovatelné registry (pro každý segm.reg.)

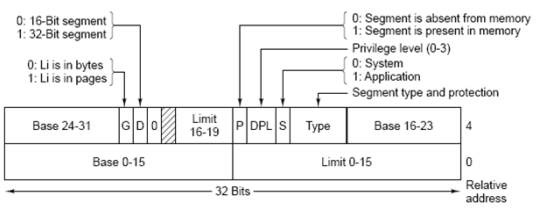
když se nahraje segment selektor do segmentového registru, odpovídající deskriptor se nahraje do odpovídajícího neprogramovatelného registru

zdroj: http://www.makelinux.net/books/ulk3/understandlk-CHP-2-SECT-2

Deskriptor segmentu

- 64bitů
  - 32 bitů báze
  - 20 bitů limit
    - v bytech, do 1MB (2^20)
    - v 4K stránkách (do 2^32) (2^12 = 4096)
  - příznaky
    - typ a ochrana segmentu
    - segment přítomen v paměti..

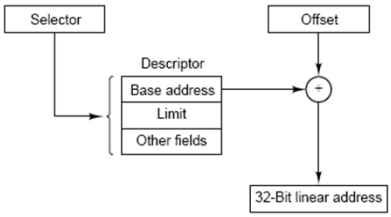
Deskriptor (8 bytů)



Konverze na fyzickou adresu

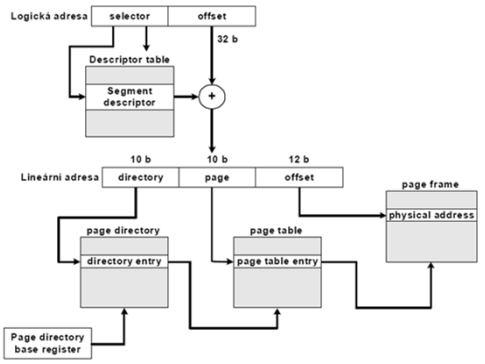
- Proces adresuje pamět pomocí segmentového registru
- CPU použije odpovídající popisovač segmentu v interních registrech
- pokud segment není – výjimka
- kontrola ofset > limit – výjimka
- 32bit. lineární adresa = báze + offset
- není-li stránkování – jde již i o fyzickou adresu
- je-li stránkování

Konverze na fyzickou adresu



Konverze na fyzickou adresu

- Pokud je dále stránkování – lineární adresa je VA, mapuje se na fyzickou pomocí tabulek stránek
- dvoúrovňové mapování



obrázek: J. Lažanský

Shrnutí

CPU Intel Pentium a výše umožňuje:

- čistá segmentace
- čisté stránkování (Linux)
  - base = 0
  - limit = MAX
  - pozn. segmentaci nejde vypnout
- stránkované segmenty (Windows)

Procesory x86

- real mode (MS-DOS)
  - po zapnutí napájení, žádná ochrana paměti
  - FA = Segment * 16 + Offset
  - FA 20bitová, segment, offset .. 16bitové
- protected mode (dnešní OS)
  - nastavíme tabulku deskriptorů (min. 0, kód, data)
  - a nastavíme PE bit v CR0 registru
- virtual 8086 mode
  - pro kompatibilitu

Procesory & přerušení

- reálný mód
  - 1. kilobyte RAM – interrupt vector table (256x4B)
- chráněný mód
  - IDT (Interrupt Descriptor Table)
  - pole 8bytových deskriptorů (indexovaných přerušovacím vektorem)
  - naplněná IDT tabulka 2KB (256x8B)

Protected mode – další typy segmentů

- call gates
  - volání předdefinované funkce přes CALL FAR
  - volání funkce s vyšší privilegií oprávnění
  - spíše se používá SYSENTER/SYSEXIT
- task state segment (TSS)
  - pro přepínání tasků
  - může obsahovat hodnoty x86 registrů
  - Win/Linux naproti tomu používají softwarový task switching

Pamatuj !

- Běžný procesor v PC může běžet v reálném nebo chráněném módu
- Po zapnutí napájení byl puštěn **reálný mód**, ten využíval např. MS-DOS – není zde však žádný mechanismus ochrany
- Dnešní systémy přepínají procesor ihned do **chráněného režimu** (ochrana segmentů uplatněním limitu, ochrana privilegovanosti kontrolou z jakého ringu je možné přistupovat)

Chráněný režim - segmenty

- 1 GDT – může mít až 8192 segmentů
- můžeme mít i více LDT (každá z nich může mít opět 8192 segmentů) a použít je pro ochranu procesů
- některé systémy využívají jen GDT, a místo LDT zajišťují ochranu pomocí stránkování

Chráněný režim – adresy !!!!

VA(selektor,ofset) =segmentace==> **LA** =stránkovaní==> **FA**
VA je virtuální adresa, LA lineární adresa, FA fyzická adresa
selektor určí odkaz do tabulky segmentů => deskriptor (v GDT nebo LDT)
selektor obsahuje mj. bázi a limit ; **LA = báze + ofset**
segmentaci nejde vypnout, stránkovaní ano
zda je zapnuté stránkovaní - bit v řídícím registru procesoru
je-li vypnuté stránkování, lineární adresa představuje fyzickou adresu

chce-li systém používat jen stránkovaní,
roztáhne segmenty přes celý adresní prostor (překrývají se)
Linux: využívá stránkovaní, Windows: obojí

KIV / ZOS

Dodatky

Seznam dodatků

- I. Alokace paměti pro procesy
- II. Principy vstupně výstupního hw

Imperativní programovací jazyky

- statické proměnné
  - vyhrazena při spuštění programu
- lokální proměnné procedur a fcí
  - alokace na zásobníku
- dynamická alokace paměti
  - v oblasti hromada (heap)
  - pomocí služeb OS

Proces

- mapování po spuštění procesu
  - kód, statické proměnné, zásobník
  - není mapována veškerá adresovatelná paměť
  - 32bit systémy 2-3GB virt.pam. (zbytek OS)
- žádost procesu o paměť
  - knihovní funkce (alokátor paměti)
- zvětšení hromady
  - požádá OS

32bit OS

32bit OS, 2^32 = 4GB

2+2 Win: proces+OS

3+1 Win: prepinac /3GB

```
[boot loader]
timeout=30
default=multi(0)disk(0)rdisk(0)partition(2)\WINNT
[operating systems]
multi(0)disk(0)rdisk(0)partition(2)\WINNT="????"
  /3GB
```

PAE (Physical Address Extensions)

podpora až 64GB fyzické paměti pro aplikace na IA-32 platformách – přepínač /PAE

Win2000, 32bit Win XP .. 4GB
Win Server 2003(8) .. 64GB

64bit Windows nepodporují PAE

expanze z 32bitů na 36bitů

2^4 = 16, 16x více, tj. 4x16 = 64GB

page directory, page tables na 8B
24 bitů místo 20 bitů

Explicitní správa paměti (C, C++, Pascal ..)

- *malloc()* a *free()* v C, *new delete* v C++
- alokátor paměti
  - spravuje hromadu
    - současná nestačí – požádá OS o další úsek virt.pam.
  - alokace např. metodou first fit
- problémy
  - nezapomenout uvolnit paměť, když není potřeba
  - roztroušení malloc, free v kódu

Snaha vyřešit správu paměti jiným způsobem, automatickou správou paměti

Čítání referencí

- např. Perl, Python
- každá datová struktura
  - položka **počet referencí**
  - při snížení referencí – test == 0 uvolnění
  - počty referencí udržovány automaticky
- nevýhody
  - čas při vytvoření / zrušení odkazu
  - cyklická datová struktura – neuvolnění paměti

Garbage collection (GC)

- automatická detekce a uvolnění neodkazované paměti
- např. samostatné vlákno
  - spuštěno při *dostupná paměť* < *limit*
- součást virtuálních strojů (JVM, CLR)
  - potřebuje rozumět obsahu datových struktur

GC

- výhody
  - usnadnění pro programátora, redukce chyb
- nevýhody
  - plně v režii virtuálního stroje
  - aktivace GC v nevhodných okamžicích
    - explicitní spuštění GC
  - nepoužívaná paměť může zůstat alokována, pokud nenastavíme odkaz na null

GC: Mark-and-sweep

- první průchod
  - vyhledává dostupnou paměť
  - začne od globální proměnné, lok. proměnné procedur a funkcí (kotevní objekty)
  - odkazy – postupuje dále na odkazovaná data
  - označení – bit „objekt je dostupný“
- druhý průchod
  - neoznačená alokovaná paměť je uvolněna
- nárazové citelné zpomalení systému

GC: Baker Collector

- inkrementální verze
- paměť na 2 semiprostory
  - jeden aktivní, v něm vytvářeny nové objekty
- po určitém čase
  - jako aktivní označen druhý semiprostor
    - projde původní "mark-and-sweep", místo označení prostor evakuuje do nového semiprostoru
    - na místě původního objektu – náhrobní kámen s novou adresou

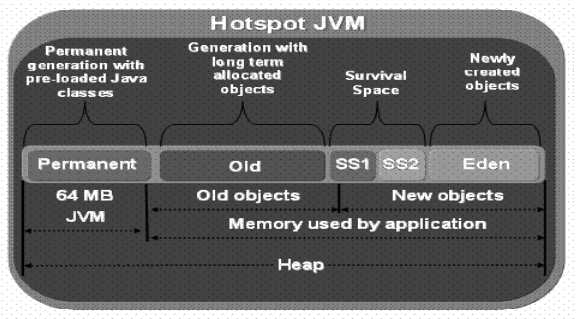
GC: Baker Collector cont.

- původní semiprostor
  - po průchodu jen smetí, lze zrušit
- nové alokace před úplnou evakuací starého semiprostoru
  - hned přesunout všechny odkazy z nového objektu
- výsledkem – čistý semiprostor
- inkrementální algoritmus
  - řeší nárazovou aktivaci mark-and-sweep

Další GC

- nevýhoda Baker C.
  - neustále přesouvá objekty – nákladné
- generační GC
  - CLR (.NET) – třígenerační GC
  - mnoho objektů krátká doba života, některé ale NE
  - přesun objektu
    - zvýšeno číslo generace objektu
    - objekt přežije N generací – do privilegované oblasti, kde není tak často zpracováván GC

Hotspot JVM



Odkazy

- http://programujte.com/index.php?akce=clanek&cl=2006060902-architektura-microsoft-net-framework-%96-3-dil
- http://www.skilldrive.com/book/DOTNETinSamples.htm

II. Principy vstupně výstupního hw (pAio.pdf)

1. CPU řídí přímo periferii
2. CPU – řadič – periferie
   aktivní čekání CPU na dokončení operace
3. řadič umí vyvolat přerušení
4. řadič umí DMA
5. I/O modul
6. I/O modul s vlastní pamětí

Magnetické disky

- geometrie
  - několik ploten – každá dva povrchy
  - stopa (track)
  - stopa rozdělena do sektorů (sectors)
    - nejmenší velikost dat číst či zapisovat
  - všechny stopy pod sebou – cylindr
    - lze přistupovat bez přesunu hlav

Disky

- disková adresa (povrch, stopa, sektor)
- virtuální geometrie
  - aby nebyla nižší hustota na vnějších stopách
- logická adresa
  - sektory číslovány od 0 bez ohledu na geometrii disku
- logická adresa = číslo sektoru

11., 12.
Správa I/O
Správa souborů

ZOS 2012, L. Pešička

Doplnění

- Alokace paměti pro procesy
  - explicitní správa paměti
  - čítání referencí
  - garbage collection

viz pdf s dodatkem – přečíst (!)

pamatuj !!

v C o paměť žádáme malloc() a máme ji uvolnit free()

paměť nám přidělí knihovna – alokátor paměti, která spravuje volnou paměť

pokud paměť nemá alokátor k dispozici, požádá operační systém systémovým voláním o přidělení další části paměti (další stránky)

malloc, brk, sbrk

- malloc není systémové volání, ale knihovní funkce viz man 3 malloc x man 2 fork
- malloc alokuje pamět z haldy
- velikost haldy se nastaví dle potřeby systémovým voláním sbrk
- brk – syscall – nastaví adresu konce datového segmentu procesu
- sbrk – syscall – zvětší velikost datového segmentu o zadaný počet bytů (0 – jen zjistím současnou adresu)

používané vs. nepoužívané objekty

```
Object x = new Foo();
Object y = new Bar();
x = new Quux();
/* víme, že Foo object původně přiřazený x nebude nikdy dostupný,
jde o  syntactic garbage */

if  ( x.check_something() )
     { x.do_something(y); }
 System.exit(0);
/* y  může být semantic garbage, ale nevíme, dokud
x.check_something() nevrátí návratovou hodnotu  */
```

zdroj: http://en.wikipedia.org/wiki/Garbage_collection_%28computer_science%29

Velikost stránky v OS

Standardní velikost je 4096 bytů (4KB)

huge page size: 4MB
large page size: 1GB

Zjištění velikosti stránky - Linux

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

int main(void) {

  printf("Velikost stranky je %ld bytu.\n",
    sysconf(_SC_PAGESIZE) );

  return 0;

}
```

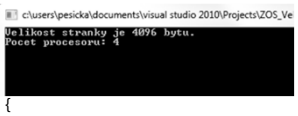
příkazem na konzoli:
getconf PAGESIZE

man sysconf

eryx.zcu.cz: 4096
ares.fav.zcu.cz: 4096

Zjištění velikosti stránky - WIN

```
#include "stdafx.h"
#include <stdio.h>
#include <Windows.h>

int _tmain(int argc, _TCHAR* argv[])  {

SYSTEM_INFO si;
GetSystemInfo(&si);
printf("Velikost stranky je %u bytu.\n", si.dwPageSize);

printf("Pocet procesoru: %u\n", si.dwNumberOfProcessors);

getchar();  return 0;     }
```



zdroj: http://en.wikipedia.org/wiki/Page_size#Page_size_trade-off

OS

- Modul pro správu procesů
- Modul pro správu paměti
- **Správa i/o**
- Správa souborů
- síťování

Vývoj rozhraní mezi CPU a zařízeními

1. CPU řídí přímo periferii
2. CPU – řadič – periferie
   aktivní čekání CPU na dokončení operace
3. řadič umí vyvolat přerušení
4. řadič umí DMA
5. I/O modul
6. I/O modul s vlastní pamětí

1. CPU řídí přímo periferii

- CPU přímo vydává potřebné signály
- CPU dekóduje signály poskytovaném zařízením
- Nejjednoduší HW
- Nejméně efektivní využití CPU

- Jen v jednoduchých mikroprocesorem řízených zařízeních (dálkové ovládání televize)

2. CPU – řadič - periférie

Řadič (device controller)

- Převádí příkazy CPU na elektrické impulzy pro zařízení
- Poskytuje CPU info o stavu zařízení
- Komunikace s CPU pomocí registrů řadiče na známých I/O adresách
- HW buffer pro alespoň 1 záznam (blok, znak, řádka)
- Rozhraní řadič-periférie může být standardizováno (SCSI, IDE, …)

2. řadič – příklad operace zápisu

- CPU zapíše data do bufferu,
  Informuje řadič o požadované operaci
- Po dokončení výstupu zařízení nastaví příznak, který může CPU otestovat
- if přenos == OK, může vložit další data

- CPU musí dělat všechno (programové I/O)
- Významnou část času stráví CPU čekáním na dokončení I/O operace

3. Řadič umí vyvolat přerušení

- CPU nemusí testovat příznak dokončení
- Při dokončení I/O vyvolá řadič přerušení
- CPU začne obsluhovat přerušení
  - Provádí instrukce na předdefinovaném místě
  - Obslužná procedura přerušení
  - Určí co dál
- Postačuje pro pomalá zařízení, např. sériové I/O

4. Řadič může přistupovat k paměti pomocí DMA

- DMA přenosy mezi pamětí a buffery
- CPU vysílá příkazy,
  při přerušení analyzuje status zařízení
- CPU inicializuje přenos, ale sám ho nevykonává
- Bus mastering – zařízení převezme kontrolu nad sběrnicí a přenos provede samo (PCI sběrnice)
- Vhodné pro rychlá zařízení – řadič disků, síťová karta, zvuková karta, grafická karta atd.

5. I/O modul umí interpretovat speciální I/O programy

- I/O procesor
- Interpretuje programy v hlavní paměti
- CPU spustí I/O procesor
  I/O procesor provádí své instrukce samostatně

6. I/O modul s vlastní pamětí

- I/O modul provádí programy
- Má vlastní paměť(!)
  - Je vlastně samostatným počítačem
- Složité a časově náročné operace
  grafika, šifrování, …

Komunikace CPU s řadičem

- Odlišné adresní prostory
  - CPU zapisuje do registrů řadiče pomocí speciálních I/O instrukcí
  - Vstup: IN R, port
  - Výstup: OUT R, port
- 1 adresní prostor
- Hybridní schéma

Ad – 1 adresní prostor

- Používá **vyhrazené** adresy
- Nazývá se *paměťově mapované I/O*
- HW musí pro dané adresy umět vypnout cachování
- Danou oblast můžeme namapovat do virtuálního adresního prostoru nějakého procesu (zpřístupnění I/O zařízení)

Ad – hybridní schéma

- Řídící registry
  - Přístup pomocí I/O instrukcí
- HW buffer
  - Mapován do paměti
- Např. PC
  (buffery mapovány do oblasti 640K až 1MB)

RAID

- pevný disk
  - elektronická část + mechanická
  - náchylost k poruchám
  - cena dat >> cena hw
- odstávka při výměně zařízení
  - náhrada hw, přenos dat ze zálohy - prostoje
  - SLA 24/7
- větší disková kapacita než 1 disk
- RAID
  - Redundant Array of Independent (Inexpensive) disks

RAID – používané úrovně

- RAID 0, 1, 5
- RAID 10 .. kombinace 0 a 1
- RAID 6 .. zdvojená parita
- pojmy:
  - SW nebo HW RAID
  - hot plug
  - hot spare
  - Degradovaný režim – jeden (či více dle typu RAIDu) z disků v poli je porouchaný, ale RAID stále funguje

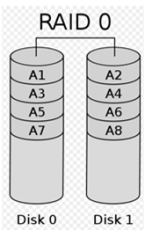
RAID 0

Dva režimy RAID 0:
zřetězení a prokládání

- není redundantní, neposkytuje ochranu dat
- ztráta 1 disku – ztráta celého pole nebo části (dle režimu)
- důvod použití – může být výkon při režimu prokládání (např. střih videa)
- Dva režimy – zřetězení nebo prokládání (stripping)

RAID 0

- Zřetězení
  - Data postupně ukládána na několik disků
  - Zaplní se první disk, pak druhý, atd.
  - Snadné zvětšení kapacity, při poruše disku ztratíme jen část dat
- Prokládání
  - Data ukládána na disky cyklicky po blocích (stripy)
  - Při poruše jednoho z disků – přijdeme o data
  - Větší rychlost čtení / zápisu
    - Jeden blok z jednoho disku, druhý blok z druhého disku

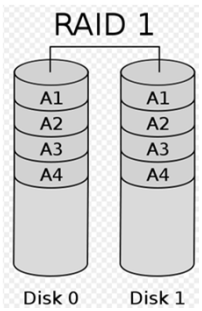


Na obrázku je režim prokládání, zdroj: wikipedia (i u dalších obrázků)

RAID 1

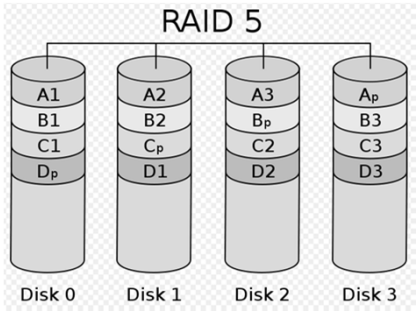
- mirroring .. zrcadlení
- na 2 disky stejných kapacit totožné informace
- výpadek 1 disku – nevadí
- jednoduchá implementace – často čistě sw
- nevýhoda – využijeme jen polovinu kapacity
- zápis – pomalejší (stejná data na 2 disky)
- čtení – rychlejší
  (řadič - lze střídat požadavky mezi disky)

RAID 1



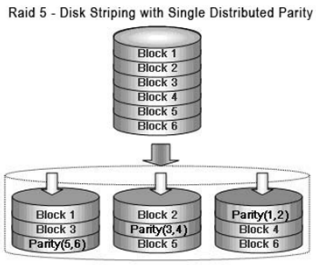
RAID 5

- redundantní pole s distribuovanou paritou
- minimálně 3 disky
- režie: 1 disk z pole n disků
  - 5 disků 100GB : 400GB pro data
- výpadek 1 disku nevadí
- čtení – výkon ok
- zápis – pomalejší
  1 zápis – čtení starých dat, čtení staré parity, výpočet nové parity, zápis nových dat, zápis nové parity



Např. RAID 5 z 4 disků 1TB, výsledná kapacita: 3 TB
Může vypadnout 1 z disků a o data nepřijdeme

RAID 5



Zdroj:
http://www.partitionwizard.com/resize-partition/resize-raid5.html

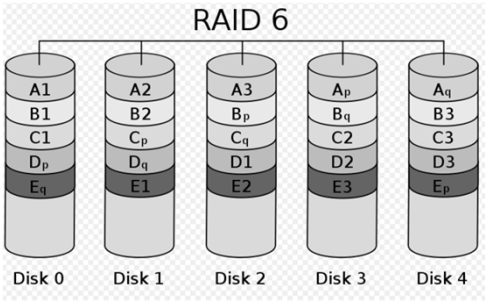
RAID 5

- nejpoužívanější

- detekce poruchy v diskovém poli
- hot spare disk
- použití hot plug disků

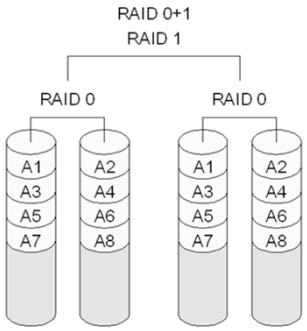
RAID 6

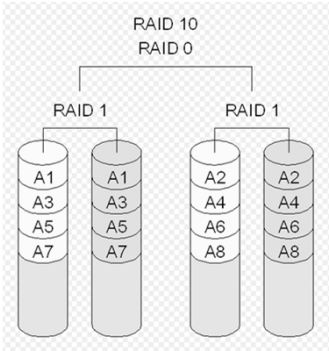
- RAID 5 + navíc další paritní disk
- odolné proti výpadku dvou disků
- Rychlost čtení srovnatelná s RAID 5
- Zápis pomalejší



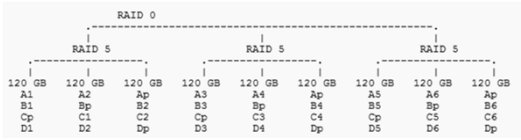
RAID 10

- kombinace RAID 0 (stripe) a RAID 1 (zrcadlo)
- min. počet disků 4
- režie 100% diskové kapacity navíc
- nejvyšší výkon v bezpečných typech polích
- podstatně rychlejší než RAID 5, při zápisu
- odolnost proti ztrátě až 50% disků x RAID 5





RAID 50



Zdroj obrázků a doporučená literatura:
http://cs.wikipedia.org/wiki/Raid

RAID 2

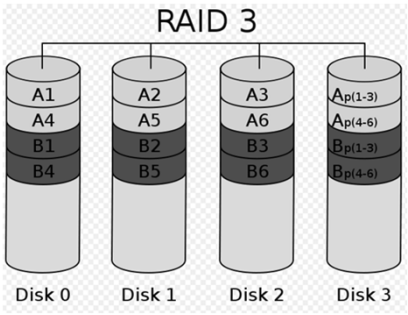
- Data po bitech stripována mezi jednotlivé disky
- Rotačně synchronizované disky
- Zabezpečení Hammingovým kódem
- Např. 7 disků
  - 4 bity datové
  - 3 bity Hammingův kód

RAID 2

RAID 3

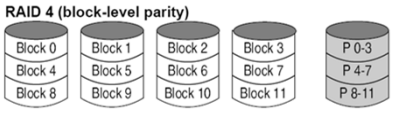
- N+1 disků, bitové prokládání
- Rotačně synchronizované disky
- Na N data, poslední disk XOR parita
- Jen 1 disk navíc
- Paritní disk vytížen při zápisu na libovolný disk – vyšší opotřebení

RAID 3

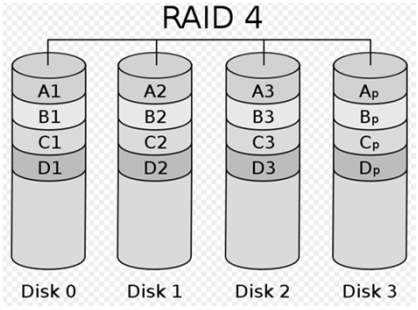


RAID 4

- Disky stripovány po blocích, ne po bitech
- Parita je opět po blocích



RAID 4



Problém rekonstrukce pole

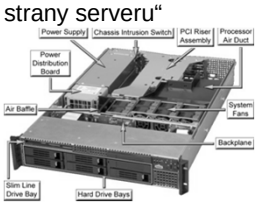
- rekonstrukce pole při výpadku – trvá dlouho
  - po dobu rekonstrukce není pole chráněno proti výpadku dalšího disku
  - náročná činnost – může se objevit další chyba, řadič disk odpojí a … přijdeme o data…

HOT SPARE

- při výpadku disku v poli automaticky aktivován hot spare disk a dopočítána data
- minimalizace rizika (časové okno)
  - Pole je degradované a je třeba vyměnit disk
- hot spare disk lze sdílet pro více polí

HOT PLUG

- Snadná výměna disku za běhu systému
- Není třeba vypnout server pro výměnu disku
- „šuplík z přední strany serveru“



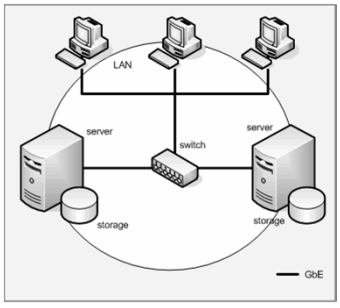
Ukládání dat

- DAS
- SAN
- iSCSI

DAS

- Directly attached storage
- ukládací zařízení přímo u serveru
- nevýhody
  - porucha serveru – data nedostupná
  - některé servery prázdné, jiné – dat.prostor skoro plný
  - rozšiřitelnost diskové kapacity
- disky přímo v serveru
- externí diskové pole přes SCSII (vedle serveru)

DAS

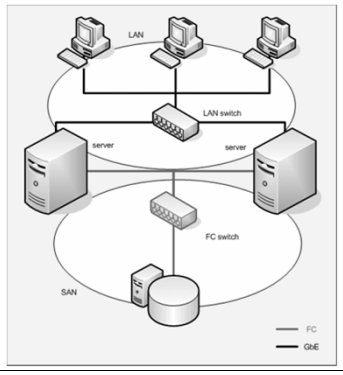


SAN

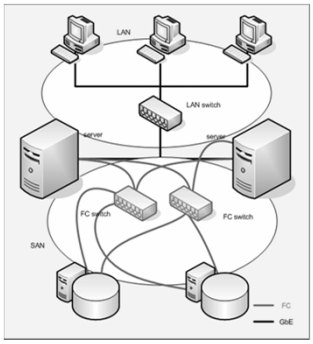
- Storage Area Network
- oddělení storage a serverů
- Fibre Channel – propojení, optický kabel

- např. clustery, společná datová oblast
- high availability solution

SAN



SAN

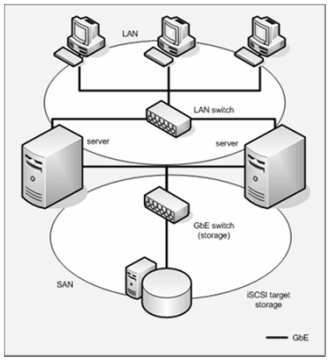


iSCSI

- SCSI + TCP/IP
- SCSI
  - protokol, bez fyzické vrstvy (kabely, konektory)
  - zapouzdření do protokolů TCP/IP
- gigabitový Ethernet vs. drahý Fibre Channel

- SCSI – adaptér, disk
- iSCSI – initiator (adapter), target (cílové zař. disk/pole)

iSCSI



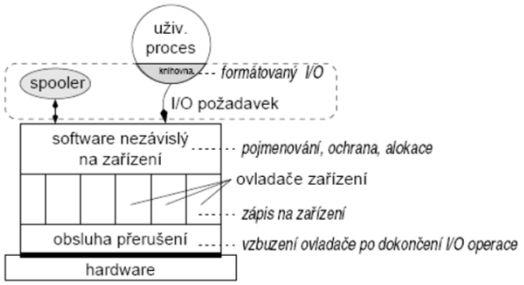
Použité odkazy a další

- http://www.vahal.cz/html/raid.html

Principy I/O software

- typicky strukturován do 4 úrovní

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



1. Obsluha přerušení

- řadič vyvolá přerušení ve chvíli dokončení I/O požadavku
- snaha, aby se přerušením nemusely zabývat vyšší vrstvy
- ovladač zadá I/O požadavek, usne (p(sem))
- po příchodu přerušení ho obsluha přerušení vzbudí (v)
- časově kritická obsluha přerušení – co nejkratší

2. Ovladače zařízení

- obsahují veškerý kod závislý na I/O zařízení
- ovladač zná jediný hw podrobnosti
  - způsob komunikace s řadičem zařízení
  - zná detaily – např. ví o sektorech a stopách na disku, pohybech diskového raménka, start & stop motoru
- ovládá všechna zařízení daného druhu nebo třídu příbuzných zařízení
  - např. ovladač SCSI disků – všechny SCSCI disky

Funkce ovladače zařízení

- ovladači předán příkaz vyšší vrstvou
  - např. zapiš data do bloku n
- nový požadavek zařazen do fronty
  - může ještě obsluhovat předchozí
- ovladač zadá příkazy řadiči (požadavek přijde na řadu)
  - např. nastavení hlavy, přečtení sektoru
- zablokuje se do vykonání požadavku
  - neblokuje při rychlých operacích – např. zápis do registru
- vzbuzení obsluhou přerušení (dokončení operace) – zkontroluje, zda nenastala chyba

Funkce ovladače zařízení – pokrač.

- pokud OK, předá výsledek (status + data) vyšší vrstvě
  - status – datová struktura pro hlášení chyb
- další požadavky ve frontě – jeden vybere a spustí
- ovladače často vytvářejí výrobci HW
  - dobře definované rozhraní mezi OS a ovladači
- ovladače podobných zařízení – stejná rozhraní
  - např. sítové karty, zvukové karty, ...

3. SW vrstva OS nezávislá na zařízení

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

- viz další slide…

Poskytované funkce

- pojmenování zařízení
  - LPT1 x /dev/lp0
- ochrana zařízení (přístupová práva)
- alokace a uvolnění vyhrazených zařízení
  - v 1 chvíli použitelná pouze jedním procesem
  - např. tiskárna, plotter, magnetická páska
- vyrovnávací paměti
  - bloková zařízení – bloky pevné délky
  - pomalá zařízení – čtení / zápis s využitím bufferu

Poskytované funkce – pokračování

- hlášení chyb
- jednotná velikost bloku pro bloková zařízení

v moderních OS se zařízení jeví jako objekty v souborovém systému
(v mnoha OS je tato vrstva součástí logického souborového systému)

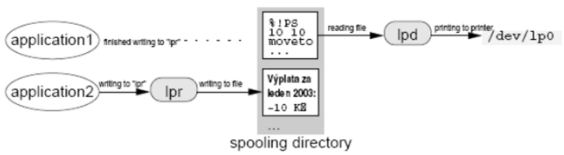
4. I/O sw v uživatelském režimu

- programátor používá v programech I/O funkce nebo příkazy jazyka
  - např. printf v C, writeln v Pascalu
  - knihovny sestavené s programem
  - formátování - printf("%.2d:%.2d\n", hodin, minut)
  - často vlastní vyrovnávací paměť na jeden blok
- spooling
  - implementován pomocí procesů běžících v uživ. režimu
  - způsob obsluhy vyhrazených I/O zařízení (multiprogram.)
  - např. proces by alokoval zařízení a pak hodinu nic nedělal

Příklad spoolingu – tisk Unix

- k tiskárně přístup – pouze 1 speciální proces
  - daemon lpd
- proces vygeneruje celý soubor, lpd ho vytiskne
  - proces chce tisknout, spustí lpr a naváže s ním komunikaci
  - proces předává tisknutá data programu lpr
  - lpr zapíše data do souboru v určeném adresáři
    - spooling directory – přístup jen lpr a lpd
  - dokončení zápisu – lpr oznámí lpd, že soubor je připraven k vytisknutí, lpd soubor vytiskne a zruší

tisk Unix



i další aplikace, spooling lze použít např. i pro přenos elektronické pošty

Souborové systémy

- potřeba aplikací trvale uchovávat data
- hlavní požadavky
  - možnost uložit velké množství dat
  - informace zachována i po ukončení procesu
  - data přístupná více procesům
- společné problémy při přístupu k zařízení
  - alokace prostoru na disku
  - pojmenování dat
  - ochrana dat před neoprávněným přístupem
  - zotavení po havárii (výpadek napájení)

Soubor

- OS pro přístup k mediim poskytuje abstrakci od fyzických vlastností média – soubor
- soubor = pojmenovaná množina souvisejících informací
- souborový systém (file system, fs)
  - konvence pro ukládání a přístup k souborům
    - datové struktury a algoritmy
  - část OS, poskytuje mechanismus pro ukládání a přístup k datům, implementuje danou konvenci

File system

- Současné OS – implementují více fs
  - kompatibilita (starší verze, ostatní OS)
- Windows XP, Vista, 7, 8
  - základní je NTFS
  - ostatní: FAT12, FAT16, FAT32, ISO 9660 (CD-ROM)
- Linux
  - ext2, ext3, ext4, ReiserFS, JFS, XFS
  - ostatní: FAT12 až 32, ISO 9660, Minix, VxFS, OS/2 HPFS, SysV fs, UFS, NTFS

Historický vývoj

- první systémy
  - vstup – děrné štítky, výstup – tiskárna
  - soubor = množina děrných štítků
- později magnetické pásky
  - vstup i výstup – pásky
  - soubor = množina záznamů na magnetické pásce

Historický vývoj - pokračování

- nyní – data na magnetických a optických discích
  - ISO 2382-4:1987
  - soubor – pojmenovaná množina záznamů, které lze zpracovávat jako celek
  - záznam – strukturovaný datový objekt tvořený konečným počtem pojmenovaných položek

Uživatelské rozhraní fs

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

Konvence pro pojmenování souborů

- vytvoření souboru – proces určuje jméno souboru
- různá pravidla pro vytváření jmen – různé OS
- Windows NT, XP x Unix a Linux

- rozlišuje systém malá a velká písmena?
  - Win32API nerozlišuje: ahoj, Ahoj, AHOJ stejná
  - UNIX rozlišuje: ahoj, Ahoj, AHOJ rozdílná jména

Pojmenování souborů

- jaká může být délka názvu souboru?
  - WinNT 256 znaků NTFS
  - UNIX obvykle alespoň 256 znaků (dle typu fs)
- množina znaků?
  - všechny běžné – názvy písmena a číslice
  - WinNT – znaková sada UNICODE
    - βετα – legální jméno souboru
  - Linux – všechny 8bitové znaky kromě / a char(0)

Pojmenování souborů

- přípony?
  - MS DOS – jméno souboru 8 znaků + 3 znaky přípona
  - WinNT, Unix – i více přípon
- další omezení?
  - WinNT – mezera nesmí být první a poslední znak

Typy souborů

- OS podporují více typů souborů
- obyčejné soubory – převážně dále rozebírány
  - data zapsaná aplikacemi
  - obvykle rozlišení textové x binární
  - textové - řádky textu ukončené CR (MAC), LF(UNIX), nebo CR+LF (MS DOS, Windows)
  - binární – všechny ostatní
  - OS rozumí struktuře spustitelných souborů

Typy souborů

- adresáře
  - systémové soubory, udržují strukturu fs
- Linux , UNIX ještě:
  - znakové speciální soubory
  - blokové speciální soubory
    - rozhraní pro I/O zařízení, /dev/lp0 – tiskárna
  - pojmenované roury
    - pro komunikaci mezi procesy
  - symbolické odkazy

Vnitřní struktura (obyčejného) souboru

- 3 časté způsoby
  - nestrukturovaná posloupnost bytů
  - posloupnost záznamů
  - strom záznamů

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

Vnitřní struktura (obyčejného) souboru – pokrač.

- posloupnost záznamů pevné délky
  - každý záznam má vnitřní strukturu
  - operace čtení –vrátí záznam, zápis – změní / přidá záznam
  - v historických systémech
  - záznamy 80 znaků obsahovaly obraz děrných štítků
  - v současných systémech se téměř nepoužívá

Vnitřní struktura (obyčejného) souboru – pokrač.

- strom záznamů
  - záznamy nemusejí mít stejnou délku
  - záznam obsahuje pole klíč (na pevné pozici v záznamu)
  - záznamy seřazeny podle klíče, aby bylo možné vyhledat záznam s požadovaným klíčem
  - mainframy pro komerční zpracování dat

Způsob přístupu k souboru

- sekvenční přístup
  - procesy mohou číst data pouze v pořadí, v jakém jsou uloženy v souboru
  - tj. od prvního záznamu, nemohou přeskakovat
  - možnost přetočit a číst opět od začátku, rewind()
  - v prvních OS, kde data na magnetických páskách

Způsob přístupu k souboru

- přímý přístup (random access file)
  - čtení v libovolném pořadí nebo podle klíče
  - přímý přístup je nutný např. pro databáze
  - určení začátku čtení
    - každá operace určuje pozici
    - OS udržuje pozici čtení / zápisu, novou pozici lze nastavit speciální operací „seek“

Způsob přístupu k souboru

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

Atributy

- informace sdružená se souborem
- některé atributy interpretuje OS, jiné systémové programy a aplikace
- významně se liší mezi jednotlivými OS
- ochrana souboru
  - kdo je vlastníkem, množina přístupových práv, heslo, ...

Atributy - pokračování

- příznaky
  - určují vlastnosti souboru
  - hidden – neobjeví se při výpisu
  - archive – soubor nebyl zálohován
  - temporary – soubor bude automaticky zrušen
  - read-only, text/binary, random access
- přístup k záznamu pomocí klíče
  - délka záznamu, pozice a délka klíče
- velikost, datum vytvoření, poslední modifikace, poslední přístup

Služby OS pro práci se soubory

- většina současných – základní model dle UNIXu
- základní filozofie UNIXu – méně je někdy více

Několik jednoduchých pravidel

- veškerý I/O prováděn pouze pomocí souborů
  - obyčejné soubory – data, spustitelné programy
  - zařízení – disky, tiskárny
  - se všemi typy zacházení pomocí **stejných** služeb systému
- obyčejný soubor – uspořádaná posloupnost bytů
  - význam znají pouze programy, které s ním pracují
  - interní struktura souboru OS nezajímá
- jeden typ souboru – seznam souborů – adresář
  - adresář je také soubor
  - soubory a adresáře koncepčně umístěny v adresáři

Jednotný přístup nebyl vždy

- speciální soubory – pro přístup k zařízením
  - DOS – PRN:, COM1:
- Poznámka – před příchodem UNIXu toto samozřejné nebylo
- většina systémů před UNIXem – samostatné služby pro čtení / zápis terminálu, na tiskárnu do souboru
- mnoho systémů před i po UNIXu – mnoho různých druhů souborů s různou strukturou a metodami přístupu

Poznámky

- systémy poskytovaly „více služeb“ x model podle UNIXu – podstatně menší složitost
- téměř všechny moderní systémy základní rysy modelu převzaly

Základní služby pro práci se soubory

- otevření souboru
  - než s ním začneme pracovat
  - úspěšné – vrátí služba pro otevření souboru – popisovač souboru (file descriptor) – malé celé číslo
  - popisovač souboru používáme v dalších službách
    - čtení apod.

Základní služby pro práci se soubory

otevření souboru:

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

Základní služby pro práci se soubory

vytvoření souboru:

- fd=creat(jméno, práva)
  - vytvoří nový soubor s daným jménem a otevře pro zápis
  - pokud soubor existoval – zkrátí na nulovou délku
  - fd – vrácený popisovač souboru

Základní služby pro práci se soubory

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

Základní služby pro práci se soubory

- operace **zápisu** do souboru
- write (fd, buffer, počet_bytů)
  - význam parametrů jako u read
  - Uprostřed souboru – přepíše, konec – prodlouží
- read() a write()
  - vrací počet skutečně zpracovaných bytů
  - jediné operace pro čtení a zápis
  - samy o sobě poskytují sekvenční přístup k souboru

Základní služby pro práci se soubory

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

Základní služby pro práci se soubory

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

Příklad použití rozhraní – kopírování souboru

```
int src, dst, in;
src = open("puvodni", O_RDONLY);        /* otevreni zdrojoveho */
dst = creat("novy", MODE);              /* vytvoreni ciloveho */
while (1)
{
in = read(src, buffer, sizeof(buffer));  /* cteme */
if (in == 0)                            /* konec souboru? */
  {
    close(src);                         /* zavreme soubory */
    close(dst);
    return;                             /* ukončení */
  }
write(dst, buffer, in);                 /* zapiseme prectena data */
}
```

Další služby pro práci se soubory

- změna přístupových práv, zamykání, ...
- závislé na konkrétních mechanismech ochrany
- např. UNIX
  - **zamykání** fcntl (fd, cmd)
  - **zjištění informací** o souboru (typ, příst. práva, velikost)
  - stat (file_name, buf), fstat (fd, buf)
  - *man stat, man fstat*

Paměťově mapované soubory

- někdy se může zdát open/read/write/close nepohodlné
- možnost mapování souboru do adresního prostoru procesu
- služby systému mmap(), munmap()
- mapovat je možné i jen část souboru

Paměťově mapované soubory - příklad

- délka stránky 4KB
- soubor délky 64KB
- chceme mapovat do adresního prostoru od 512KB
- 512 * 1024 = 524 288 .. od této adresy mapujeme
- 0 až 4KB souboru bude mapováno na 512KB – 516KB
- čtení z 524 288 čte byte 0 souboru atd.

Implementace paměťově mapovaných souborů

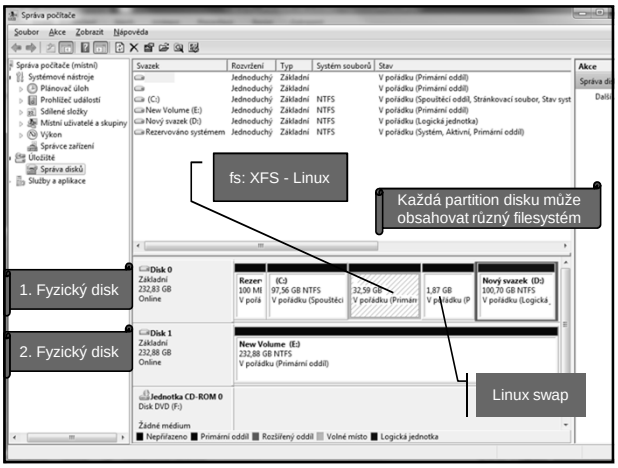
- OS použije soubor jako odkládací prostor (swapping area) pro určenou část virtuálního adresního prostoru
- čtení / zápis na adr. 524 288 způsobí výpadek stránky
- do rámce se načte obsah první stránky souboru
- pokud je modifikovaná stránka vyhozena (nedostatek volných rámců), zapíše se do souboru
- po skončení práce se souborem se zapíší všechny modifikované stránky

Problémy pam. map. souborů

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

Adresářová struktura

- jedna oblast (partition) disku obsahuje jeden fs
- fs – 2 součásti:
  - množina souborů, obsahujících data
  - adresářová struktura – udržuje informace o všech souborech v daném fs
- adresář překládá jméno souboru na informace o souboru (umístění, velikost, typ ...)



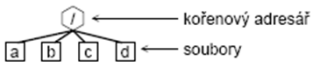
Základní požadavky na adresář

- procházení souborovým systémem (*cd*)
- výpis adresáře (*ls*)
- vytvoření a zrušení souboru
- přejmenování souboru

- dále schémata logické struktury adresářů
  - odpovídá historickému vývoji OS

Schémata logické struktury adresářů

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

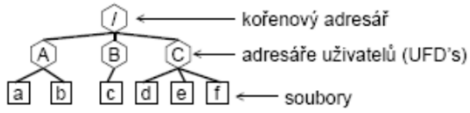


nelze mít dva soubory priklad.c

Dvouúrovňový adresář

- adresář pro každého uživatele (User File Directory, UFD)
- OS prohledává pouze UFD , nebo pokud specifikováno adresář jiného uživatele [user] file
- systémové příkazy – spustitelné soubory – speciální adresář
  - příkaz se hledá v adresáři uživatele
  - pokud zde není, vyhledá se v systémovém adresáři

Dvouúrovňový adresář – pokr.



každý uživatel může být nanejvýš jeden soubor nazvaný priklad.c

Adresářový strom

- zobecnění předchozího
- dnes nejčastější, MS DOS, Windows NT
- adresář – množina souborů a adresářů
- souborový systém začíná kořenovým adresářem „/“
- MS DOS „\“, znak / se používal pro volby
- cesta k souboru – jméno v open, creat
  - absolutní
  - relativní

Cesta k souboru

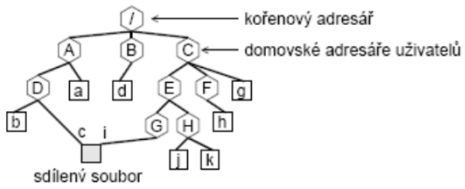
absolutní cesta začíná:
/ (Linux)
C:\ (windows)

- absolutní
  - kořenový adresář a adresáře, kudy je třeba projít, název souboru
  - oddělovače adresářů – znak „/“
  - např. /home/user/data/v1/data12.txt
- relativní
  - aplikace většinou přistupují k souborům v jednom adresáři
  - defaultní prefix = pracovní adresář
  - cesta nezačíná znakem /
  - př. *data12.txt* , *data/v1/data12.txt*

Acyklický graf adresářů

- např. týmová spolupráce na určitém projektu
- sdílení společného souboru nebo podadresáře
  - stejný soubor (adr.) může být viděn ve dvou různých adresářích
- flexibilnější než strom, komplikovanější
- rušení souborů / adresářů – kdy můžeme zrušit?
  - se souborem sdružen počet odkazů na soubor z adresářů
  - každé remove sníží o 1, 0 = není odkazován
- jak zajistit aby byl graf acyklický?
  - algoritmy pro zjištění, drahé pro fs

Acyklický graf adresářů



stejný soubor viděný v různých cestách

Obecný graf adresářů

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

Nejčastější použití

- nejčastěji adresářový strom (MS DOS)
- UNIX od původních verzí acyklický graf
  hard links – sdílení pouze souborů – nemohou vzniknout cykly

POZOR!
Je nutné si uvědomit rozdíl mezi pojmy adresářový strom a acyklický graf.

Základní služby pro práci s adresáři

- téměř všechny systémy dle UNIXu
- pracovní adresář – služby:
- chdir (adresář)
  - nastavení pracovního adresáře
- getcwd (buffer, počet_znaků)
  - zjištění pracovního adresáře

Práce s adresářovou strukturou

- vytváření a rušení adresářů
  - mkdir (adresář, přístupová_práva)
  - rmdir (adresář) – musí být prázdný
- zrušení souboru
  - remove (jméno_souboru)
- přejmenování souboru
  - rename (jméno_souboru, nové_jméno)
  - provádí také přesun mezi adresáři

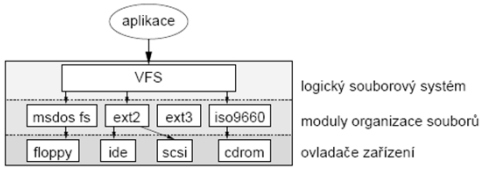
Práce s adresářovou strukturou

- čtení adresářů – UNIX / POSIX
  - DIRp = opendir (adresář)
    - otevře adresář
  - položka = readdir (DIRp)
    - čte jednotlivé položky adresáře
  - closedir (DIRp)
    - zavře adresář
  - stat (jméno_souboru, statbuf)
    - info o souboru, viz man 2 stat
- př. DOS: findfirst / findnext

ke všem uvedeným voláním získáte v Linuxu podrobnosti pomocí:

man 2 opendir
man 2 readdir
man 2 stat

Implementace souborových systémů (!!!)



Implementace fs - vrstvy

1. Logický (virtuální) souborový systém
   - Volán aplikacemi
2. Modul organizace souborů
   - Konkrétní souborový systém (např. ext3)
3. Ovladače zařízení
   - Pracuje s daným zařízením
   - Přečte/zapíše logický blok

Ad 1 – virtuální fs

- Volán aplikacemi
- Rozhraní s moduly organizace souborů
- Obsahuje kód společný pro všechny typy fs
- Převádí jméno souboru na informaci o souboru
- Udržuje informaci o otevřeném souboru
  - Pro čtení / zápis (režim)
  - Pozice v souboru
- Ochrana a bezpečnost (ověřování přístupových práv)

Ad 2 – modul organizace souborů

- Implementuje konkrétní souborový systém
  - ext3, xfs, ntfs, fat, ..
- Čte/zapisuje datové bloky souboru
  - Číslovány O až N-1
  - Převod čísla bloku na diskovou adresu
  - Volání ovladače pro čtení – zápis bloku
- Správa volného prostoru + alokace volných bloků
- Údržba datových struktur filesystému

Ad 3 – ovladače zařízení

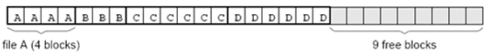
- Nejnižší úroveň
- Floppy, ide, scsi, cdrom
- Interpretují požadavky:
  přečti logický blok 6456 ze zařízení 3

Implementace souborů

- Nejdůležitější:
  které diskové bloky patří kterému souboru ☺
- Předpokládáme:
  fs je umístěn v nějaké disk partition
  bloky v oblasti jsou očíslovány od 0

Kontinuální alokace

- Soubor jako kontinuální posloupnost diskových bloků
- Př.: bloky velikosti 1KB, soubor A (4KB) by zabíral 4 po sobě následující bloky
- Implementace
  - Potřebujeme znát číslo prvního bloku
  - Znát celkový počet bloků souboru (např. v adresáři)
- Velmi rychlé čtení
  - Hlavičku disku na začátek souboru, čtené bloky jsou za sebou



Kontinuální alokace

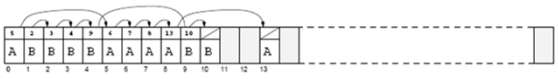
- Problém – dynamičnost OS
  – soubory vznikají, zanikají, mění velikost
- Nejprve zapisovat sériově do volného místa na konci
- Po zaplnění využít volné místo po zrušených souborech
- Pro výběr vhodné díry – potřebujeme znát konečnou délku souboru – většinou nevíme..

Lze dnes využít kontinuální alokaci?

- Dnes se používá pouze na read-only a write-once médiích
- Např. v ISO 9660 pro CD ROM

Seznam diskových bloků

- Svázat diskové bloky do seznamu – nebude vnější fragmentace
- Na začátku diskového bloku je uložen odkaz na další blok souboru, zbytek bloku obsahuje data souboru
- Pro přístup k souboru stačí znát pouze číslo prvního bloku souboru (může být v adresáři)

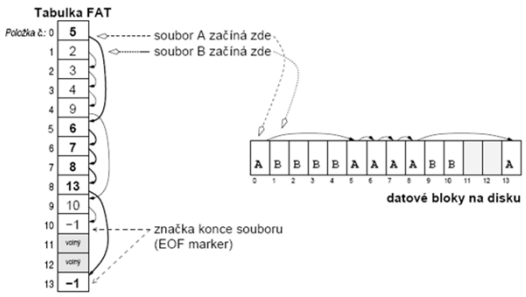


Seznam diskových bloků

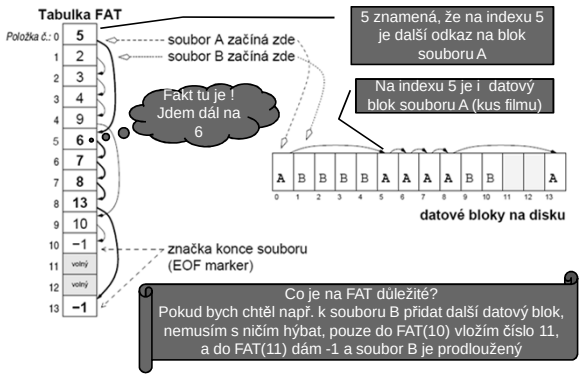
- Sekvenční čtení – bez potíží
- Přímý přístup – simulován sekvenčním, pomalé (musí dojít ke správnému bloku)
- Velikost dat v bloku není mocnina dvou
  - Část bloku je zabraná odkazem na další blok
  - Někdy může být nevýhodou

FAT (!!)

- Přesunutí odkazů do samostatné tabulky FAT
- FAT (File Allocation Table)
  - Každému diskovému bloku odpovídá jedna položka ve FAT tabulce
  - Položka FAT obsahuje číslo dalšího bloku souboru (je zároveň odkazem na další položku FAT!)
  - Řetězec odkazů je ukončen speciální značkou, která není platným číslem bloku



Položce číslo X ve FAT odpovídá datový blok X na disku
Položka ve FAT obsahuje odkaz na další datový blok na disku a tedy i na další položku ve FAT tabulce



FAT

- FAT je ukázka implementace souborového systému, kde v jedné části máme datové bloky (obsahující např. části jednoho filmu) a v druhé části máme indexy, které nám říkají, pod jakým číslem se nalézá další odkaz
- Výhodou je, že s určitým souborem můžeme manipulovat, zrušit ho, prodloužit, atd., aniž bychom ovlivnili pozici ostatních souborů na disku

Příklady filesystémů (!!!)

- **FAT**
  - Starší verze Windows, paměťové karty
  - Nepoužívá ACL – u souborů není žádná info o přístupových právech
  - snadná přenositelnost dat mezi různými OS
- **NTFS**
  - Používá se ve Windows XP/Vista/7
  - Používají ACL: k souboru je přiřazen seznam uživatelů, skupin a jaká mají oprávnění k souboru (!!!!)
- **Ext2**
  - Použití v Linuxu, nemá žurnálování
  - Nepoužívá ACL – jen standardní nastavení (vlastní,skupina, others), což ale není ACL (to je komplexnější)
- **Ext3**
  - Použití v Linuxu, má žurnál (rychlejší obnova konzistence po výpadku)

Příklady filesystémů

- ext4
  - stejně jako ext2, ext3 používá inody
  - extenty – souvislé logické bloky
    - může být až 128MB oproti velkému počtu 4KB bloků
- xfs
- jfs

Vlastnosti FAT

- Nevýhodou je velikost tabulky FAT
  - 80GB disk, bloky 4KB => 20 mil. položek
  - Každá položka alespoň 3 byty => 60MB FAT
  - Výkonnostní důsledky (FAT nebude celá v paměti)
- Použití
  - DOS, Windows 98, ME, podporují Win NT/2000/XP
  - FAT12, 12 bitů, 2^12 = 4096 bloků, diskety
  - FAT16, 16 bitů, 2^16 = 65536 bloků
  - FAT32, 2 ^28 bloků, blok 4-32KB, cca 8TB

NTFS

- nativní fs Windows od NT výše
- žurnálování
  všechny zápisy na disk se zapisují do žurnálu, pokud uprostřed zápisu systém havaruje, je možné dle stavu žurnálu zápis dokončit nebo anulovat => konzistentní stav
- access control list
  přidělování práv k souborům (x FAT)
- komprese
  na úrovni fs lze soubor nastavit jako komprimovaný

NTFS pokračování

- šifrování
  EFS (encrypting file system),
  transparentní – otevřu ahoj.txt, nestarám se, zda je šifrovaný
- diskové kvóty
  max. velikost pro uživatele na daném oddíle
  dle reálné velikosti (ne komprimované)
- pevné a symbolické linky

NTFS struktura

- 64bitové adresy klusterů .. cca 16EB
- systém jako obří databáze
  záznam odpovídá souboru
- základ 11 systémových souborů - metadat
  hned po formátování svazku
- $Logfile – žurnálování
- $MFT (Master File Table)
  záznamy o všech souborech, adresářích, metadatech
  hned za boot sektorem, za ním se udržuje zóna volného místa

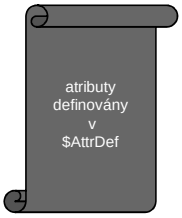
NTFS struktura

- $MFTMirr – uprostřed disku, obsahuje část záznamů $MFT, při poškození se použije tato kopii
- $Badclus – seznam vadných clusterů
- $Bitmap – sledování volného místa
  0 – volný
- $Boot, $Volume, $AttrDef, $Quota, $Upcase, .

podrobnosti:
http://technet.microsoft.com/en-us/library/cc781134%28WS.10%29.aspx

NTFS atributy souborů

- $FILE_NAME
  - jméno souboru
  - velikost
  - odkaz na nadřazený adresář
  - další
- $SECURITY_DESCRIPTOR
  - přístupová práva k souboru
- $DATA
  - vlastní obsah souboru



NTFS

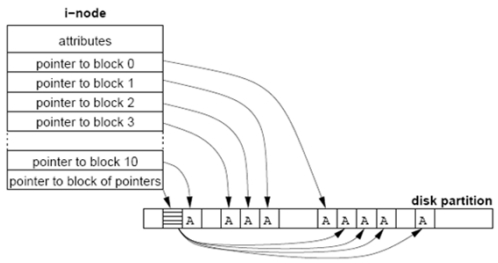
- adresáře
  - speciální soubory
  - B-stromy se jmény souborů a odkazy na záznamy v MFT
- alternativní datové proudy (ADS)
  - notepad poznamky.txt**:tajny.txt**
  - často útočiště virů, škodlivého kódu
- zkopírováním souboru z NTFS na FAT
  - ztratíme přístupová práva a alternativní datové proudy

NTFS – způsob uložení dat (!!!)

- kódování délkou běhu
- od pozice 0 máme např. uloženo:
  A1, A2, A3, B1, B2, A4, A5, C1, …
- soubor A bude popsaný fragmenty
- fragment
  - index
  - počet bloků daného souboru
- v našem příkladě:
  - 0, 3 (od indexu 0 patří tři bloky souboru A)
  - 5, 2 (od indexu 5 patří dva bloky souboru A)

I-uzly (!!)

- S každým souborem sdružená datová struktura i-uzel (i-node, zkratka z index-node)
- i-uzel obsahuje
  - Atributy souboru
  - Diskové adresy prvních N bloků
  - 1 či více odkazů na diskové bloky obsahující další diskové adresy (případně obsahující odkazy na bloky obsahující adresy)
- Používá tradiční fs v Unixu UFS (Unix File System) a z něj vycházející v Linuxu, dnes např. ext2, ext3, ext4



I-uzly - výhoda

Po otevření souboru můžeme zavést i-uzel a případný blok obsahující další adresy do paměti => zrychlení přístupu k souboru

i-uzly dle normy Posix

- MODE – typ souboru, přístupová práva (u,g,o)
- REFERENCE COUNT – počet odkazů na tento objekt
- OWNER – ID vlastníka
- GROUP – ID skupiny
- SIZE – velikost objektu
- TIME STAMPS
  - atime – čas posledního přístupu (čtení souboru, výpis adresáře)
  - mtime – čas poslední změny
  - ctime – čas poslední změny i-uzlu (metadat)

i-uzly dle normy POSIX

- DIRECT BLOCKS – 12 přímých odkazů na datové bloky (data v souboru)
- SINGLE INDIRECT – 1 odkaz na datový blok, který místo dat obsahuje seznam přímých odkazů na datové bloky obsahující vlastní data souboru
- DOUBLE INDIRECT – 1 odkaz 2. nepřímé úrovně
- TRIPLE INDIRECT – 1 odkaz 3. nepřímé úrovně

v linuxových fs (ext*) ještě FLAGS, počet použitých datových bloků a rezervovaná část – doplňující info (odkaz na rodičovský adresář, ACL, rozšířené atributy)

Implementace adresářů

- Před čtením je třeba soubor otevřít
- *open (jméno, režim)*
- Mapování jméno -> info o datech poskytují adresáře !

- Adresáře jsou často speciálním typem souboru
- Typicky pole datových struktur, 1 položka na soubor

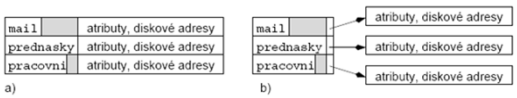
2 základní uspořádání adresáře (!!!)

1. Adresář obsahuje jméno souboru, atributy, diskovou adresu souboru (např. adresa 1.bloku) (implementuje DOS, Windows)

2. Adresář obsahuje pouze jméno + odkaz na jinou datovou strukturu obsahující další informace (např. i-uzel) (implementuje UNIX, Linux)

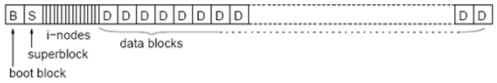
Běžné jsou oba dva způsoby i kombinace

2 základní uspořádání adresáře (!!)



Příklad fs (Unix v7)

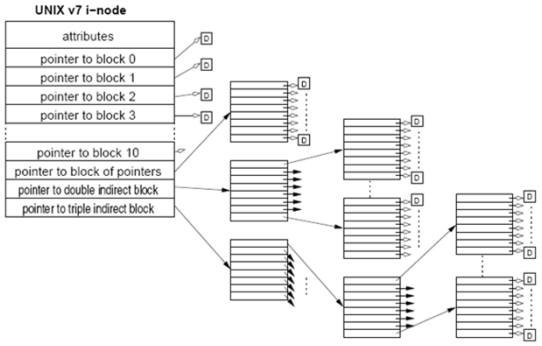
- Struktura fs na disku
  - Boot blok – může být kód pro zavedení OS
  - Superblok – informace o fs (počet i-uzlů, datových bloků,..)
  - i-uzly – tabulka pevné velikosti, číslovány od 1
  - Datové bloky – všechny soubory a adresáře



Implementace souborů – i-uzly

i-uzel obsahuje:

- Atributy
- Odkaz na prvních 10(až 12) datových bloků souboru
- Odkaz na blok obsahující odkazy na datové bloky (nepřímý odkaz)
- Odkaz na blok obsahující odkazy na bloky obsahující odkazy na datové bloky (dvojitě nepřímý odkaz)
- Trojitě nepřímý odkaz



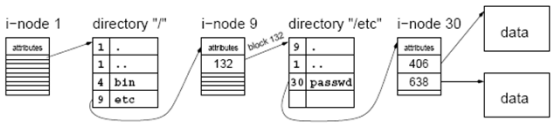
Pokračování příkladu

- Implementace adresářů:
  tabulka obsahující jméno souboru a číslo jeho i-uzlu
- Info o volných blocích
  seznam, jeho začátek je v superbloku
- Základní model, současné fs jsou na něm založeny

Adresáře v UNIX v7

- adresář: jméno souboru a číslo i-uzlu
- Číslo i-uzlu je indexem do tabulky i-uzlů na disku
- Každý soubor a adresář: právě 1 i-uzel
- V i-uzlu: všechny atributy a čísla diskových bloků
- Kořenový adresář: číslo i-uzlu 1

- Nalezení cesty k souboru /etc/passwd
  - V kořenovém adresáři najdeme položku „etc“
  - i-uzel číslo 9 obsahuje adresy diskových bloků pro adresář etc
  - V adresáři etc (disk blok 132) najdeme položku passwd
  - i-uzel 30 obsahuje soubor /etc/passwd
  - (uzel, obsah uzlu, uzel, obsah uzlu)



Příklad – adresář win 98

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

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

Sdílení souborů

- Soubor ve více podadresářích nebo pod více jmény
- Hard links (pevné odkazy)
  - Každý soubor má datovou strukturu, která ho popisuje (i-uzel), můžeme vytvořit v adresářích více odkazů na stejný soubor
  - Všechny odkazy (jména) jsou rovnocenné
  - V popisu souboru (i-uzlu) musí být počet odkazů
  - Soubor zanikne při zrušení posledního odkazu

Sdílení souborů

- Symbolický link
  - Nový typ souboru, obsahuje jméno odkazovaného souboru
  - OS místo symbolického odkazu otevře odkazovaný soubor
  - Obecnější – může obsahovat cokoliv
  - Větší režie

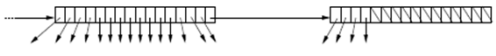
Správa volného prostoru

- Info, které bloky jsou volné
- Nejčastěji – seznam nebo bitová mapa
- Bitová mapa
  - Konstantní velikost
  - Snažší vyhledávání volného bloku s určitými vlastnostmi
  - Většina současných fs používá bitovou mapu

Správa volného prostoru

- Seznam diskových bloků
  - Blok obsahuje odkazy na volné bloky a adresu dalšího bloku …
  - Uvolnění bloků - Přidáme adresy do seznamu, pokud není místo blok zapíšeme
  - Potřebujeme bloky pro soubor – používáme adresy ze seznamu, pokud nejsou přečteme další blok adres volných bloků
  - Pokud není na disku volné místo, seznam volných bloků je prázdný a nezabírá místo
  - Problém najít volný blok s určitými vlastnostmi (např. ve stejném cylindru), prohledávat seznam, drahé,…

Seznam diskových bloků



Kvóty

- Maximální počet bloků obsazených soubory uživatele
- Ve víceuživatelských OS, servery
- Hard a soft kvóty

Spolehlivost souborového systému

- Ztráta dat má často horší důsledky než zničení počítače
  - diplomová – bakalářská práce
  - Fotografie za posledních 10 let
- Filesystém musí být jedna z nejspolehlivějších částí OS, snaha chránit data
  - Správa vadných bloků (hlavně dříve)
  - Rozprostřít a duplikovat důležité datové struktury, čitelnost i po částečném poškození povrchu

Konzistence fs

- Blokové zařízení
  OS přečte blok souboru, změní ho, zapíše
- Nekonzistentní stav
  může nastat při havárii (např. výpadek napájení) předtím, než jsou všechny modifikované bloky zapsány
- Kontrola konzistence fs
  Windows: scandisk, **chkdsk**
  UNIX: fsck , fsck.ext3, e2fsck .. viz man
- Kontrolu spustí automaticky po startu, když detekuje nekorektní ukončení práce se systémem

Testy konzistence fs

- Konzistence informace o diskových blocích souborů
  - Blok (obvykle) patří jednomu souboru nebo je volný
- Konzistence adresářové struktury
  - Jsou všechny adresáře a soubory dostupné?

důležité pochopit rozdíl:
-kontrola konzistence souboru
-kontrola, zda je soubor dostupný z nějakého adresáře

Konzistence informace o diskových blocích souborů

- Tabulka počtu výskytů bloku v souboru
- Tabulka počtu výskytů bloku v seznamu volných bloků
- Položky obou tabulek inicializovány na 0
- Procházíme informace o souborech (např. i-uzly), inkrementujeme položky odpovídající blokům souboru v první tabulce
- Procházíme seznam nebo bitmapu volných bloků a inkrementujeme příslušné položky ve druhé tabulce

Konzistentní fs

Číslo bloku	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**
Výskyt v souborech	1	0	1	**0**	**1**	**0**	**2**	0	1
Volné bloky	0	1	0	**0**	**1**	**2**	**0**	1	0

Blok je buď volný, nebo patří nějakému souboru, tj. konzistentní hodnoty v daném sloupci jsou buď (0,1) nebo (1,0)
Vše ostatní jsou chyby různé závažnosti

Možné chyby, závažnosti

- (0,0) – blok se nevyskytuje v žádné tabulce
  - Missing blok
  - Není závažné, pouze redukuje kapacitu fs
  - Oprava: vložení do seznamu volných bloků
- (0,2) – blok je dvakrát nebo vícekrát v seznamu volných
  - Problém – blok by mohl být alokován vícekrát !
  - Opravíme seznam volných bloků, aby se vyskytoval pouze jednou

Možné chyby, závažnosti

- (1,1) – blok patří souboru a zároveň je na seznamu volných
  - Problém, blok by mohl být alokován podruhé !
  - Oprava: blok vyjmeme ze seznamu volných bloků
- (2,0) – blok patří do dvou nebo více souborů
  - Nejzávažnější problém, nejspíš už došlo ke ztrátě dat
  - Snaha o opravu: alokujeme nový blok, problematický blok do něj zkopírujeme a upravíme i-uzel druhého souboru
  - Uživatel by měl být informován o problému

Je zde nějaká chyba? A když tak jaká?

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

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

Kontrola konzistence adresářové struktury

- Tabulka čítačů, jedna položka pro každý soubor
- Program prochází rekurzivně celý adresářový strom
- Položku pro soubor program zvýší pro každý výskyt souboru v adresáři
- Zkontroluje, zda odpovídá počet odkazů v i-uzlu (i) s počtem výskytů v adresářích (a)
- i == a ☺ pro každý soubor

Možné chyby

- **i > a**
  soubor by nebyl zrušen ani po zrušení všech odkazů v adresářích
  není závažné, ale soubor by zbytečně zabíral místo
  řešíme nastavením počtu odkazů v i-uzlu na správnou hodnotu (a)

Možné chyby

- **i < a**
  soubor by byl zrušen po zrušení i odkazů, ale v adresářích budou ještě jména
  velký problém – adresáře by ukazovaly na neexistující soubory
  řešíme nastavením počtu odkazů na správnou hodnotu

Možné chyby

- **a=0 , i > 0**
  ztracený soubor, na který není v adresáři odkaz ve většině systémů program soubor zviditelní na předem určeném místě (např. adresář lost+found)

Další heuristické kontroly

- Odpovídají jména souborů konvencím OS?
  - Když ne, soubor může být nepřístupný, změníme jméno
- Nejsou přístupová práva nesmyslná?
  - Např. vlastník nemá přístup k souboru,…
- Zde byly uvedeny jen základní obecné kontroly fs

Journaling fs

- Kontrola konzistence je časově náročná
- Journaling fs
  - Před každým zápisem na disk vytvoří na disku záznam popisující plánované operace, pak provede operace a záznam zruší
  - Výpadek – na disku najdeme žurnál o všech operacích, které mohly být v době havárie rozpracované, zjednodušuje kontrolu konzistence fs

Výkonnost fs

- Přístup k tradičnímu disku řádově pomalejší než přístup do paměti
  - Seek 5-10 ms
  - Rotační zpoždění – až bude požadovaný blok pod hlavičkou disku
  - Rychlost čtení (x rychlost přístupu do paměti)
- Použití SSD disků
  - Rychlé, lehké, malá spotřeba (výdrž notebooků)
  - menší kapacita, drahé
  - 60GB cca 3-4 tisíce Kč

Výkonnost fs, cachování

- Cachování diskových bloků v paměti
- Přednačítání (read-ahead)
  do cache se předem načítají bloky, které se budou potřebovat při sekvenčním čtení souboru
- Redukce pohybu diskového raménka pro po sobě následující bloky souboru,…

Mechanismy ochrany

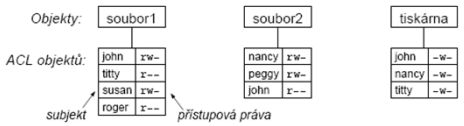
- Chránit soubor před neoprávněným přístupem
- Chránit i další objekty
  - HW (segmenty paměti, I/O zařízení)
  - SW (procesy, semafory, …)
- Subjekt – entita schopná přistupovat k objektům (většinou proces)
- Objekt – cokoliv, k čemu je potřeba omezovat přístup pomocí přístupových práv (např. soubor)
- Systém uchovává informace o přístupových právech subjektů k objektům

ACL x capability list

- Dvě různé podoby
- ACL – s objektem je sdružen seznam subjektů a jejich přístupových práv
- Capability list [kejpa-] – se subjektem je sdružen seznam objektů a přístupových práv k nim

ACL (Access Control Lists)

- S objektem je sdružen seznam subjektů, které mohou k objektu přistupovat
- Pro každý uvedený subjekt je v ACL množina přístupových práv k objektu

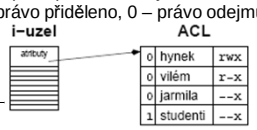


ACL

- Sdružování subjektů do tříd nebo do skupin
  - Studenti
  - Zamestnanci
- Skupiny mohou být uvedeny na místě subjektu v ACL
- Zjednodušuje administraci
  - Nemusíme uvádět všechny studenty jmenovitě
- ACL používá mnoho moderních filesystémů (ntfs, xfs, …)

Úloha: jak doimplementovat ACL do i-uzlu?

- V i-uzlu by byla část tabulky ACL, pokud by se nevešla celá do i-uzlu, tak odkaz na diskový blok obsahující zbytek ACL
- Každá položka ACL
  - Subjekt: id uživatele či id skupiny + 1 bit rozlišení uživatel/skupina
  - Přístupové právo Nbitovým slovem
    1 – právo přiděleno, 0 – právo odejmuto



Mechanismus capability lists (C-seznamy)

- S každým subjektem (procesem) sdružen seznam objektů, kterým může přistupovat a jakým způsobem (tj. přístupová práva)
- Seznam se nazývá capability list (C-list)
- Jednotlivé položky - capabilities

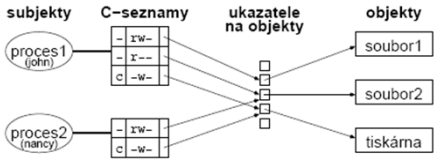
Struktura capability

- Struktura capability
  - Typ objektu
  - Práva – obvykle bitová mapa popisující dovolené operace nad objektem
  - Odkaz na objekt, např. číslo uzlu, segmentu, atd..

Capability

- Problém – zjištění, kdo všechno má k objektu přístup
- Zrušení přístupu velmi obtížné – najít pro objekt všechny capability + odejmout práva
- Řešení: odkaz neukazuje na objekt, ale na nepřímý objekt
  systém může zrušit nepřímý objekt, tím zneplatní odkazy na objekt ze všech C-seznamů

Capability list



Capability list

Pokud jsou jediný způsob odkazu na objekt (bezpečný ukazatel, capability-based addressing):

- Ruší rozdíl mezi objekty na disku, v paměti (segmenty) nebo na jiném stroji (objekty by šlo přesouvat za běhu)
- Mechanismus C-seznamů v některých distribuovaných systémech (Hydra, Mach,…)

Přístupová práva

- FAT – žádná
- ext2
  - klasická unixová práva (není to ACL)
  - vlastník, skupina, ostatní (r,w,x,s,…)
  - lze přidat ACL
- NTFS
  - ACL
  - lze měnit grafické UI, příkaz icacls, …
  - explicitně udělit / odepřít práva
  - zdědit práva, zakázat dědění