Úvod do informačních technologií

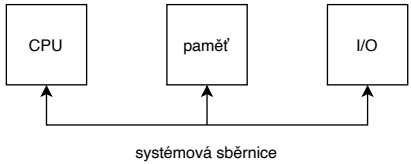
operační systémy

Martin Trnečka

Katedra informatiky
Univerzita Palackého v Olomouci

Počítač: Připomenutí

- základní části
  - procesor
  - paměť
  - I/O
  - systémová sběrnice
- mentální model počítače
- reálně pouze princip, dnešní počítače jsou mnohem složitější

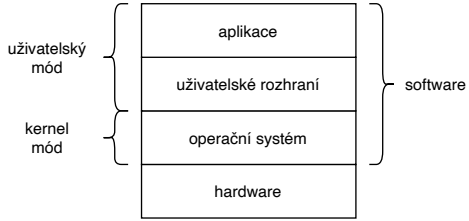


Obrázek: Von Neumannova architektura počítače (1945)

Co je to operační systém?

- neformálně:
  - základní softwarové vybavení počítače
  - program díky kterému počítač funguje
  - rozhraní mezi uživatelem a počítačem (hardware)
- složitá otázka $\rightarrow$ neexistuje přesná definice
- popíšeme jednotlivé části
- motivace pro studium:
  - lepší uživatelé $\rightarrow$ nezajímavé
  - lepší programátoři
  - poučení se z minulosti

Co je to operační systém?



Obrázek: Operační systém a jeho umístění.

- dva základní pohledy:
  - abstrakce nad hardware
  - správa zdrojů (hardware)
- operační systém má dvě rozhraní

Co dělá operační systém?

- umožňuje (interaktivní) práci s hardware
  - komunikace s hardware (ovladače)
  - uživatelské rozhraní (UI, user interface)
- správa software (instalace, spuštění, . . .)
- správa hardware
  - možnost běhu více programů současně
  - řízení a poskytování zdrojů (procesor, paměť)
  - bezpečnost a efektivita
- organizace a správa dat
- poskytuje služby programům
  - abstrakce nad hardware
  - systémové služby formou API (Application Programming Interface)
  - například čtení souboru (pohled uživatele, programu, OS)

Architektura OS

Obrázek: Architektura OS.

Jádro

- nejnižší úroveň OS
- nejdůležitější část OS
- běží v privilegovaném (kernel) módu
- zajišťuje:
  - základní abstrakci nad hardware
  - správu zdrojů, izolace, bezpečnost
  - poskytování služeb pomocí *systémového volání*

Typy jader

- Monolitické
  - vše v jednom, rychlé, menší abstrakce
  - vrstvená architektura
  - například: BSD, Linux
- Mikro
  - jen to nejnutnější, zbytek mimo jádro
  - statisticky 2–10 chyb na 1000 řádků
  - monolitické systémy, 5 mil. řádků = 10 tis.–50 tis. chyb
  - pomalejší, větší abstrakce, snadnější údržba
  - například: MINIX
  - později se z mikro jader vyvinuly systémy založení na klient-server architektuře
- Hybridní
  - kombinace předchozích
  - například: Windows NT, macOS
- Exo, neposkytuje abstrakci nad HW, rychlejší přístup k HW, akademické
- Uni, pouze jedna aplikace

Systémové knihovny

- vrstva (abstrakce) nad jádrem
- poskytuje služby programům formou „pěkného “API
- využívá služeb jádra
- například knihovny: `libc` či `msvcrt`
  - funkce jazyka C
  - například `printf()`
- statické a dynamické (sdílené)
  - sestavení programu, redundance
  - použity při běhu, problémy se závislostí (DLL hell)
- příklad: POSIX, specifikace rozhraní pro OS
  - výrazné usnadnění tvorby programů

Systémové služby

- služby/démoni
- někdy se používá klient-server
- programy běžící na pozadí
- obvykle:
  - konrétní služba
  - údržba
  - opakující se úkoly
  - úkoly vyžádané jádrem

Uživatelské rozhraní OS

- příkazový řádek CLI (Command Line Interface) nebo také terminál
- grafické rozhraní GUI (Graphical User Interface)
- umožňuje uživateli ovládat OS a hardware (na dané úrovni abstrakce)
- poskytuje základní stavební kameny pro UI
- poskytováno systémovými službami či knihovnami

Systémové nástroje

- menší programy pro práci s OS
- konfigurace (editor registrů, textový editor)
- správa souborového systému (`ls` či `dir`)

Typy operačních systémů

- různé vlastnosti a požadavky
- který operační systém je lepší $\rightarrow$ zbytečná otázka
- základní typy OS:
  - univerzální (běžné počítače)
  - embedded (speciální zařízení, omezené zdroje)
  - real-time (hard, soft), pozor real-time $\neq$ běží v reálném čase
  - distribuované (běh na více strojích, sdílení prostředků, clustery)

Vykonávání programu

- program = sada instrukcí (zapsaných v programovacím jazyce) uložená v paměti
  - program není nic jiného něž data na pevném disku
  - spuštění programu = operační systém přesune program do RAM, a začne jej vykonávat
- instrukce procesoru $\rightarrow$ omezené operace
  - například: k číslu v registru přičti číslo 1
  - instrukční sada procesoru (číselné kódy)
  - jazyk symbolických adres (assembly language)
- program (v programovacím jazyce) $\rightarrow$ instrukce pro procesor
  - programovací jazyky poskytují abstrakci nad instrukcemi procesoru
  - kompilovaný vs. interpretovaný programovací jazyk

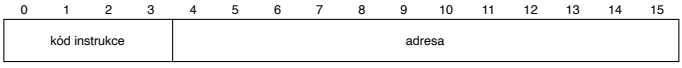
Vykonávání programu

- registry procesoru
  - adresa následující instrukce v program counter (PC)
  - aktuálně zpracovávána instrukce v instruction register (IR)
  - místo v paměti pro čtení/zápis memory address register (MAR)
  - data přečtená/k zápisu do paměti v memory buffer register (MBR)
  - analogicky I/O AR, I/O BR

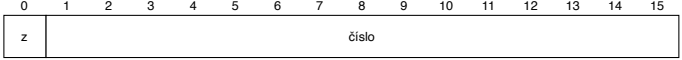
- zpracování instrukce (procesoru)
  - adresa následující instrukce v PC je přenesena do MAR
  - provede se načtení z paměti (z místa určeného MAR)
  - z paměti načtená hodnota je přenesena do MBR
  - instrukce v MBR je přenesena do IR
  - hodnota PC je zvýšena o 1
  - dekódování instrukce
  - vykonání instrukce (může zahrnovat více operací)
  - poznámka: superskalární architektura

Vykonávání programu: Příklad

- velmi zjednodušený hypotetický procesor



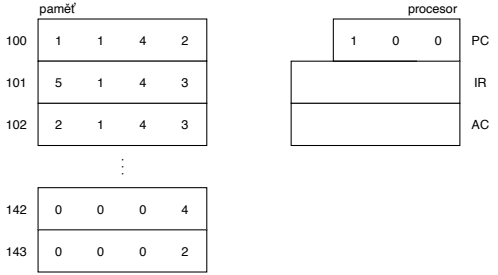
reprezentace instrukce



reprezentace čísla

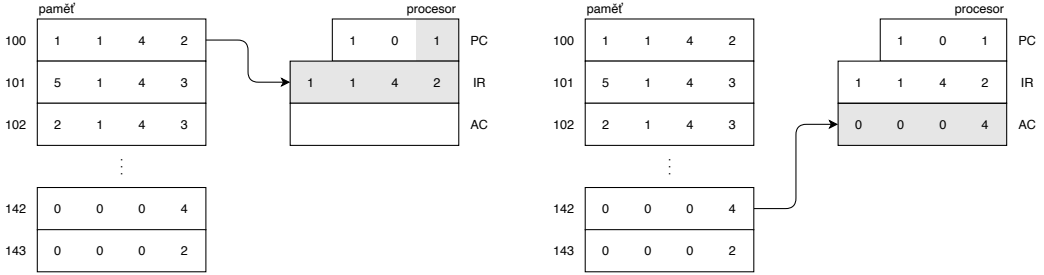
- adresní prostor 4kB ($2^{12} = 4096$, 12 bitů = 3 hexadecimální čísla)
- registry
  - PC – adresa následující instrukce
  - IR – adresa aktuální instrukce
  - AC – registr pro uložení dat
  - pro jednoduchost vynecháme MAR a MBR
- instrukce procesoru (4 bity = hexadecimální číslo)
  - 0001 – načtení z paměti do AC (1)
  - 0010 – uložení z AC do paměti (2)
  - 0101 – přičtení hodnoty z paměti k hodnotě v AC (5)

Vykonávání programu: Příklad



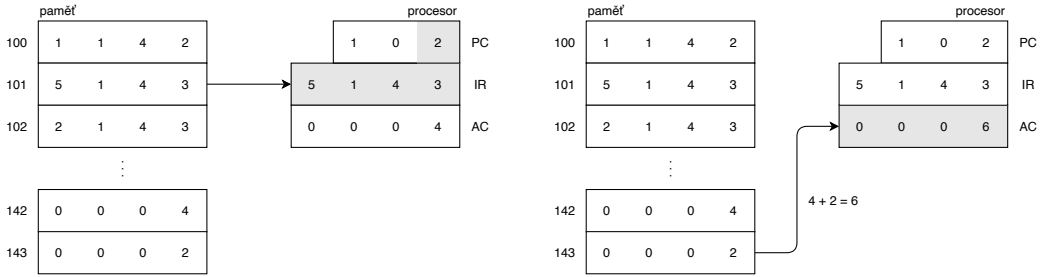
Obrázek: Výchozí stav procesoru a paměti

Vykonávání programu: Příklad



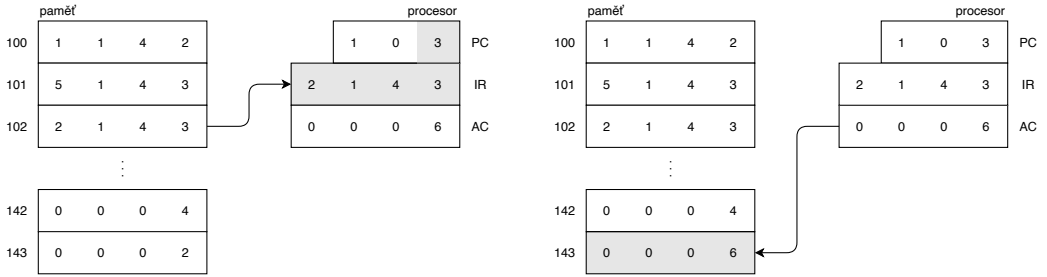
Obrázek: Načtení a vykonání první instrukce

Vykonávání programu: Příklad



Obrázek: Načtení a vykonání druhé instrukce

Vykonávání programu: Příklad



Obrázek: Načtení a vykonání třetí instrukce

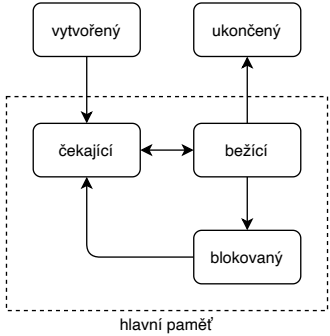
Přerušení

- přerušení postupného zpracovávání instrukcí
- běžná záležitost (I/O, časovač, chyby v programu, chyby hardware)
- slouží k předání řízení obsluze přerušení (součást OS)
- (zjednodušený) proces přerušení
  - je dokončena aktuální operace procesoru
  - uložení aktuálního stavu (program status word)
  - změna stavu registrů (dle obsluhy přerušení)
  - pokračování ve vykonávání instrukcí
- k přerušení může dojít i při vykovávání obsluhy přerušení

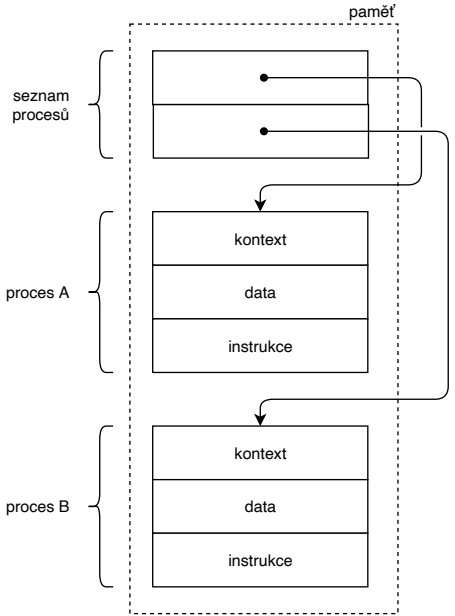
Proces

- entita spravovaná OS
- abstrakce nad běžícím programem
  - operační systém „nerozumí“ programu – jen zajišťuje vykonávání instrukcí
  - potřebuje s ním pracovat (např. přerušit) → potřebuje uchovávat režijní informace
- přístup ke zdrojům
  - paměť (izolovaný adresní prostor, soubory)
  - procesor
- OS udržuje informace o procesu tzv. *kontext*

Proces: Životní cyklus



Proces: Zjednodušená implementace



Procesy

- v jeden okamžik může na PC s jedním CPU běžet pouze jeden proces
  - konkurentního běhu je docíleno neustálým přepínáním více procesů (time-sharing)
  - poznámka: konkurentní vs. paralelní běh
- běžně běží více procesů $\rightarrow$ potřeba řízení
- každý proces má svého vlastníka (obvykle hierachie)
- procesy mezi sebou mohou komunikovat
  - meziprocesová komunikace (Inter-Process Communication, IPC)
  - jádro + systémové knihovny

Proces: Přepnutí procesu

- context switch
- procesy uloženy v *process table*
- záznam v process table = *process control block* (PCB)
- PCB uchovává úplný stav programu (registry, paměť, otevřené soubory, ...)
- přepnutí:
  - uložení stavu procesu do PCB
  - aktualizace PCB (např. změna stavu)
  - uložení PCB (do příslušné fronty dle stavu)
  - výběr nového procesu
  - aktualizace PCB (např. změna stavu)
  - načtení dat z PCB
  - aktualizace dle PCB
- *kooperativní* (ponecháno na procesu) vs. *preemptivní* (zajišťuje OS)

Proces: Plánování běhu

- požadavek: co nejvíce práce vs. interaktivita (protichůdné)
- procesy je třeba organizovat
- plánování běhu = výběr procesů, které „poběží“
- zajišťuje plánovač, součást OS
- proces má procesor přidělen na dané *časové kvantum* (provádí se instrukce daného programu)
- přepnutí procesu = režie
- kritická a komplikovaná záležitost
- typy plánování:
  - dle času (krátkodobé, střednědobé, dlouhodobé)
  - dle typu úlohy (dávkové zpracování, interaktivní, reálný čas)

Proces: Plánování

- dva typy procesů
- CPU-bound proces
  - vykonávaný program obsahuje minimum I/O operací
- I/O-bound proces
  - vykonávaný program obsahuje značné množství I/O operací
  - čekání na I/O
- pro oba typy se hodí různé plánovací strategie

Proces: Strategie plánování

- cyklická obsluha (round robin)
  - procesy se pravidelně střídají
  - nemusí běžet stejně dlouho
  - nevhodné pro interaktivní práci
- spravedlivé přidělení času
  - procesy běží stejně dlouho
  - nevhodné pro interaktivní práci
- prioritní fronta, procesu je přidělena *priorita*
  - interaktivní práce
  - nespravedlivé dělení prostředků $\rightarrow$ vyhladovění
- mnoho dalších
- dnes se běžně používá kombinace cyklické obsluhy a prioritní fronty

Odbočka: Jak je plánován běh OS?

- operační systém = program
- plánování běhu
  1. funkce OS vykonávány mimo procesy (dnes nepoužívané)
  2. funkce OS vykonávány uvnitř uživatelských procesů (běžné)
     - části OS sdílené mezi všemi procesy
     - kernel zásobník pro volání v kernel módu
     - přepíná se pouze mód procesoru (menší režie)
  3. funkce OS vykonávány samostatné procesy
     - výhodné v případě více procesorů

Vlákno

- abstrakce nad vykonávaným kódem (obvykle jeho částí)
- více vláken v rámci procesu
- vlákna sdílí paměť (v rámci procesu)
- využití:
  - asynchronní zpracování
  - neblokující I/O
  - GUI

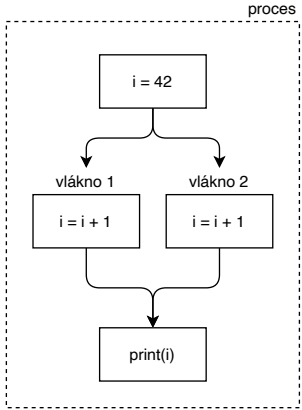
Vlákno: Správa

- práce s procesem (vytvoření, přepínání) má (velkou) režii
- práce s vlákny je výrazně jednodušší
  - nedochází k přepínání procesu
- realizace:
  - systémová knihovna (user-level vlákna)
    přepnutí nevyžaduje kernel mód, správa v rámci procesu,
  - jádro (kernel-level vlákna)
    přepnutí vyžaduje kernel mód, vlákna lze plánovat jako procesy
  - kombinace
- vlákna vs. procesy:
  - 1:1 UNIX
  - m:1 Windows NT, Linux, macOS
  - 1:m vlákno může být přesunuto mezi procesy (distribuované systémy, cloud)
  - m:n

Komunikace a synchronizace

- procesy a vlákna mezi sebou soutěží (race)
- procesy jsou izolované $\rightarrow$ komunikace přes zprávy
- vlákna $\rightarrow$ komunikace přes sdílenou paměť
- chyba souběhu (race condition) = nekonzistentní výsledek v důsledku chybného nesprávného běhu procesů/vláken

Chyba souběhu: Příklad



Chyba souběhu: Příklad

- `i = i + 1` není atomická
- ve skutečnosti je to několik operací
  1. načti `i` do registru
  2. přičti k hodnotě v registru `1`
  3. zapiš hodnotu v registru do `i`
- kdykoliv (po dokončení instrukce procesoru) může dojít k přepnutí vlákna (či procesu)

Chyba souběhu: Příklad

vlákno 1	vlákno 2	`i`
načti		42
přičti		42
zapiš		43
	načti	43
	přičti	43
	zapiš	44

vlákno 1	vlákno 2	`i`
načti		42
	načti	42
přičti		42
	přičti	42
zapiš		43
	zapiš	43

Odbočka: Slavné programátorské chyby

- Y2K (pád systémů po celém světě, chybný výpočet přestupného roku)
- HearthBleead (chyba v zabezpečení, opomenutí)
- Mariner 1 (zničení sondy, znaménko)
- Mars Climate Orbiter (zničení sondy, různé jednotky)
- Ariane 5 (zničení sondy, přetečení 16bitového čísla)
- Therac-25 (smrt pacientů, chyba souběhu)
- další: rakety Patriot, burza, letiště Heathrow, YouTube, kalkulačka Windows, ...

Synchronizace

- zabránění race condition
- určení pořadí běhů procesů a vláken
- pomocí zpráv (typicky procesy), sdílených proměnných (typicky vlákna), systémového volání, aktivního a pasivního čekání
- základní nástroje:
  - *atomické operace* (HW podpora, ale stačí pouze atomické uložení)
  - *synchronizační primitiva* (zámek, semafor, monitor, bariéry, implementováno pomocí atomických operací)
- *kritická sekce* = místo přístupu ke sdíleným (kritickým) prostředkům
- požadavky na kritickou sekci
  - vzájemné vyloučení (pouze jeden může vstoupit)
  - absence zbytečného čekání
  - zaručený vstup
  - zaručený výstup

Synchronizace: Příklad

- jednoduchá verze klasického problému: producent-konzument
- producent zapisuje hodnotu do sdílené proměnné
- konzument čte hodnotu ze sdílené proměnné
- producent nesmí přepsat nepřečtenou hodnotu
- konzument nesmí opakovaně číst stejnou hodnotu

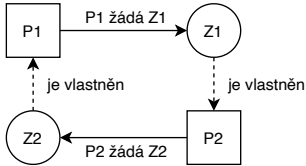
```
sdilena_promenna = nil

loop forever # producent
  data = produce()
  wait sdilena_promenna == nil
  sdilena_promenna = data # kritická sekce

loop forever # konzument
  wait sdilena_promenna != nil
  data = sdilena_promenna # kritická sekce
  sdilena_promenna = nil # kritická sekce
  consume(data)
```

Deadlock

- chyba synchronizace
- čekání na výlučně vlastněné zdroje = uváznutí
- podmínky vzniku (Coffmanovy podmínky, 1971):
  - vzájemné vyloučení (zdroj vlastní pouze jeden)
  - vlastník zdroje může žádat o další zdroje
  - absence preempce (zdroj musí být vrácen, nelze jej odebrat)
  - cyklické čekání
- příklad:



Deadlock: Řešení

- ignorování (běžné OS)
- detekce a následné zotavení
- zamezení vzniku
- vyhýbání se vzniku

Další problémy

- vyhladovění (starvation)
  - procesu/vláknu není poskytnut přístup do kritické sekce (neposkytnutím zdrojů)
- livelock
  - procesy/vlákna běží, ale nevykonávají žádnou práci

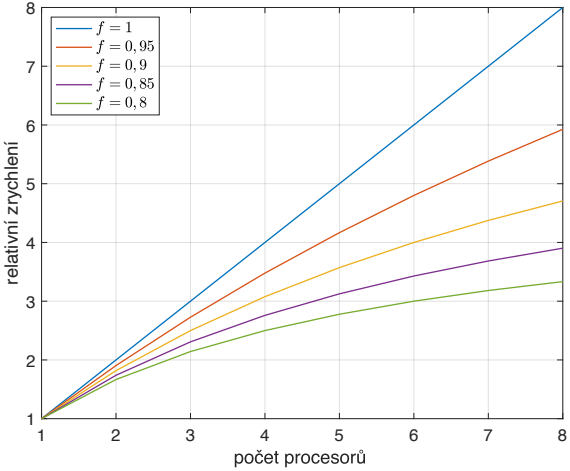
Zrychlení pomocí paralelismu

- Amdahlův zákon (maximální předpokládané zlepšení systému)

$$\text{zrychlení} = \frac{\text{čas běhu na 1 procesoru}}{\text{čas paralelního běhu na } n \text{ procesorech}} = \frac{1}{(1-f) + \frac{f}{n}}$$

- f libovolně paralelizovatelný kód
- $(1-f)$ představuje část kódu, kterou nelze vykonávat paralelně
- nerealisticky optimistické

Amdahlův zákon



- reálně mnohem horší (pozn. pro $f = 0.95$ se limitně blíží k 20)
- režie při přepínání $\rightarrow$ pokles zrychlení s narůstajícím počtem procesorů

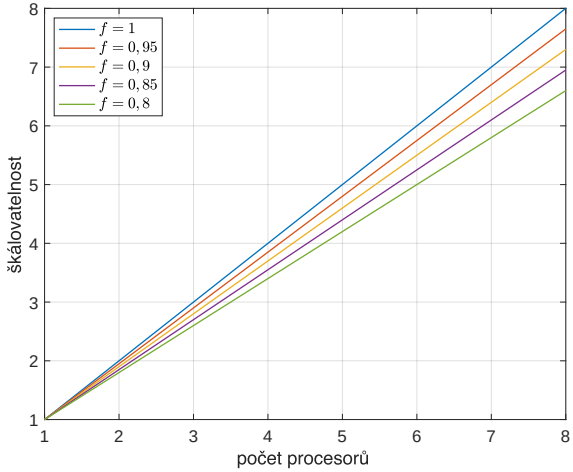
Gustafson-Barsisův zákon

- Amdahlův zákon = zrychlení při konstantním množství dat
- zrychlení systému je omezené neparalelizovatelnou částí
- v praxi lepší výsledky než udává Amdahlův zákon
- Gustafson-Barsisův zákon
  - když mám vysoký výkon použiji ho → více procesorů = více zpracovaných dat ve stejném čase, přesnější výsledek apod.
  - škálovatelnost = při růstu systému jsou zachovány jeho vlastnosti

$$\text{zrychlení} = \frac{(1-t) + n \cdot t}{(1-t) - t} = n + (1-n) \cdot (1-t)$$

- t doba běhu (procento) libovolně paralelizovatelného kódu
- $(1-t)$ doba běhu (procento) neparalelizovatelného kódu

Gustafson-Barsisův zákon



- „sekvenční část“ typicky neroste s velikostí problému (inicializace, agregace výsledků)

Amdahlův vs Gustafson-Barsisův zákon

- dva odlišné pohledy, ale překvapivě ekvivalentní
- Amdahl – limitované zrychlení (čas) daného problému
- Gustafson – daný čas, neomezené zrychlení
- není to divné?
  - auto jede z A do B, vzdálené 200 km
  - za hodinu 100 km
  - Amdahl: i když zbytek cesty pojede libovolnou rychlostí, průměrná rychlost nebude větší než 200 km/h
  - Gustafson: pokud auto pojede dostatečně daleko dosáhne libovolné průměrné rychlosti
- oba zákony ukazují pohled na škálovatelnost systémů (s rostoucími prostředky roste výkon)

Událostní řízení

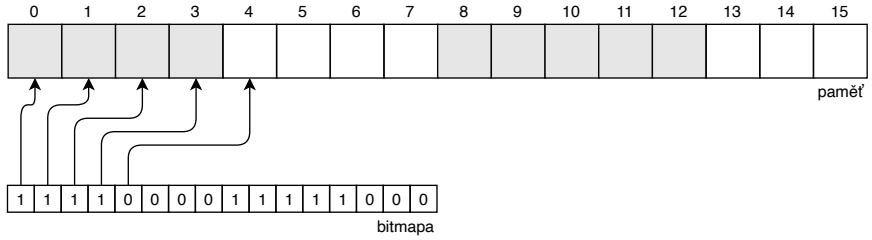
- vlákno představuje pro OS menší režii než proces (přesto je zde režie)
- prostředky jsou omezené $\rightarrow$ počet vláken je omezený
- C10k problém
- v případě I/O úloh lze efektivně řešit bez použití vláken
- řízení na úrovni funkcí
  - korutiny
  - funkce mají více vstupních bodů
  - asynchronní programování
  - třeba plánovač na úrovni procesu
- například webový server

Správa operační paměti

- nedostatečná velikost RAM $\rightarrow$ nutnost správy
- proces žádá o paměť OS mu ji přidělí
  - jednoúlohové OS
  - víceúlohové $\rightarrow$ nutnost izolace paměti procesů
- pouze souvislá část paměti
- evidence volného místa = režie

Odbočka: Evidence volného místa

- spojový seznam nebo bitmapa

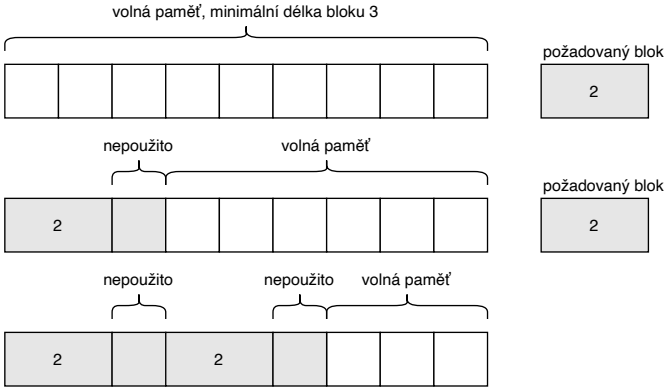


- bitmapa je používanější

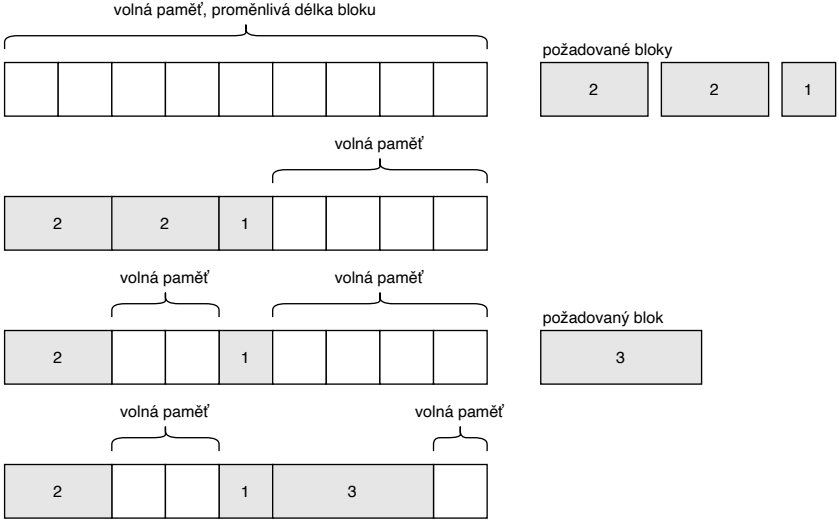
Správa operační paměti

- přidělování souvislých bloků paměti
- třeba zvolit velikost bloku
  - stejná velikost bloku
    nevyužité místo v bloku $\rightarrow$ *vnitřní fragmentace*
  - proměnlivá velikost bloku
    po uvolnění nedostatečné místo pro větší bloky $\rightarrow$ *vnější fragmentace*

Vnitřní fragmentace: Příklad



Vnější fragmentace: Příklad



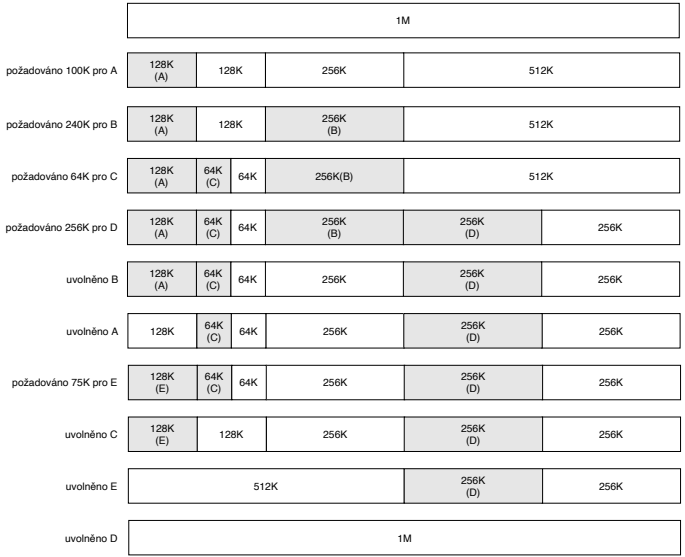
Správa operační paměti

- algoritmy pro výběr bloku:
  - first fit
  - best fit
  - worst fit
- nutný kompromis
- problémy:
  - fragmentace zabraňuje efektivnímu využití paměti (cca $1/3$ je nepoužitelná)
  - neřeší situaci, kdy se program do paměti (již) nevejde

Správa operační paměti: Buddy systém

- alternativa ke stejné a proměnlivé velikosti bloku $\rightarrow$ kompromis
- dostupná paměť 2^u
- bloky o velikosti 2^k, $l \leq k \leq u$
- reprezentace binární stromem, třeba uložit v paměti
- přidělení paměti:
  - pokud je blok větší než 2^k je rozdělen na dva
  - rekurzivně se pokračuje dokud není nalezen nejmenší blok do kterého se 2^k vejde
- uvolnění paměti:
  - pokud je sourozenec volný dojde ke spojení do většího bloku
  - rekurzivně se pokračuje dokud lze bloky spojovat

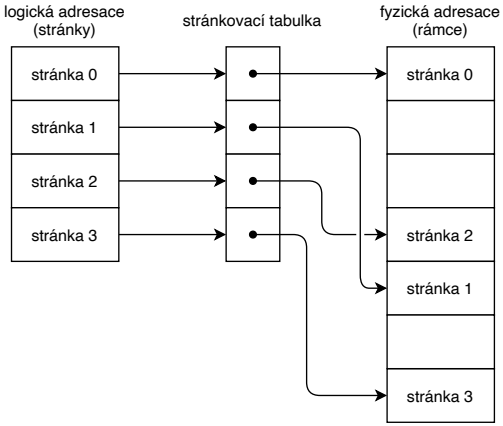
Buddy systém: Příklad



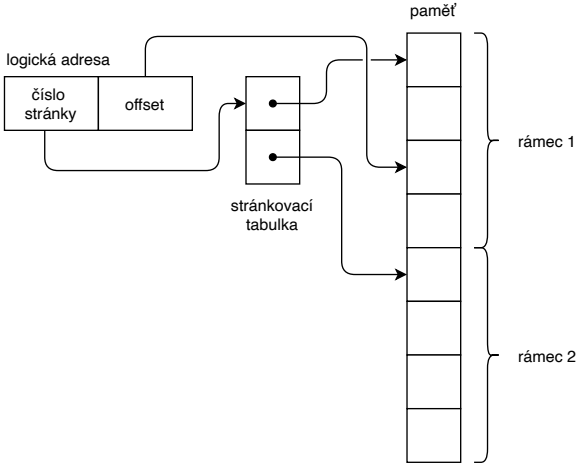
Stránkování

- abstrakce nad pamětí
- adresní prostor programu rozdělen na *stránky* (logická adresace)
- stránky jsou uloženy v paměti *rámcích* (fyzická adresace)
- velikost stránky/rámce dána OS (obvykle 4 nebo 8 kB, ale i jiné)
- OS eviduje volné rámce a rámce přidělené procesu

Stránkování: Princip



Stránkování: Adresace



Stránkování

- stále dochází k vnitřní fragmentaci, vnější je potlačena
- pomalý přístup do paměti
- nutnost HW podpory (TLB cache)
- plýtvání pamětí při uložení stránkovacích tabulek → hiearchie (tzv. adresáře) stránkovacích tabulek (4 úrovně)
- třeba zohledni při programování → *princip lokality*

Princip lokality

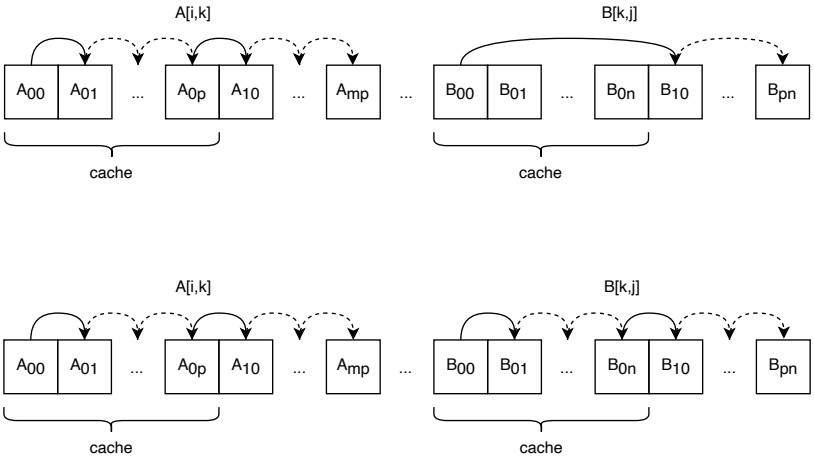
- násobení (velkých) matic $C \leftarrow A \cdot B$
- A je $m \times p$, B je $p \times n$
- klasický algoritmus, složitost $\mathcal{O}(m \cdot n \cdot p)$

```
for i in {0, ..., m} # cyklus přes řádky A
  for j in {0, ..., n} # cyklus přes sloupce B
    for k in {0, ..., p} # cyklus přes řádky B
      C[i,j] += A[i,k] * B[k,j];
```

- drobná úprava, stále $\mathcal{O}(m \cdot n \cdot p)$, ale výrazně rychlejší

```
for i in {0, ..., m} # cyklus přes řádky A
  for k in {0, ..., p} # cyklus přes řádky B
    for j in {0, ..., n} # cyklus přes sloupce B
      C[i,j] += A[i,k] * B[k,j];
```

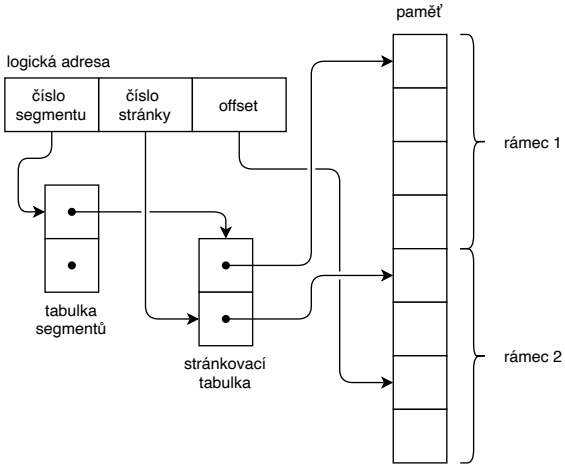
Princip lokality: Ilustrace



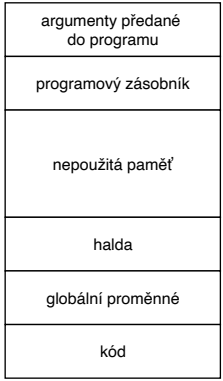
Segmentace paměti

- rozdělení paměti procesu na logické části (segmenty)
- obecně je segmentace nezávislá na stránkování
  - segmentace je možná i bez stránkování (speciální případ správy paměti)
  - velikost segmentu může být nezávislá navelikosti stránky
- pro jednoduchost segment = skupina stránek
- každý segment má přistupová oprávění (například oddělení zásobník, halda) → bezpečnost
- sdílení segmentů = šetří paměť
- tabulka segmentů

Segmentace: Princip



Odbočka: Správa paměti na úrovni programu



Obrázek: Segmentace paměti přidělené programu.

Správa paměti na úrovni programu

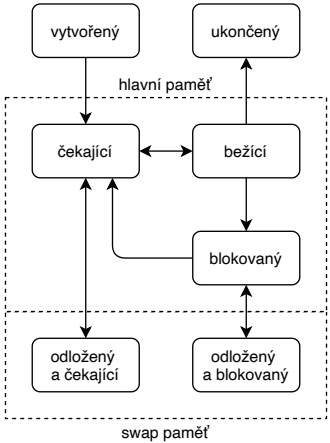
- stránkování programátora nemusí zajímat, segmentace ano
  - dynamicky rostoucí datové struktury
  - modularita
- manuální = ponechána na programátorovi
  - lze ovlivnit počty výpadků stránky
- automatická = *garbage collector* (John McCarthy, 1958)
  - zjednodušení vývoje na úkor HW prostředků
  - různé modely a implementace (počítání odkazů, mark&sweep, kopírující, ...)

Virtuální paměť

- v paměti jsou udržovány pouze používané stránky
- nepoužívané stránky uloženy na disku (swapovací soubor)
- výhody:
  - v paměti může být pouze používaná část procesu $\rightarrow$ může běžet více procesů
  - proces může být větší než dostupná paměť
- nevýhody: režie práce s I/O
- *výpadek stránky* (page fault) = požadovaná stránka není v paměti
- výměna rámců:
  - pokud exsituje volný rámec je použit
  - pokud ne $\rightarrow$ vybrat rámec (oběť), který bude nahrazen

Stránkování a procesy

- proces může být uložen v nepoužívané stránce → nový životní cyklus procesu



Výběr oběti

- ideální oběť (nebude v budoucnu použita) $\rightarrow$ nelze zjistit
- nejstarší stránka
- nejdéle nepoužitá stránka
- nejméně používaná stránka $\rightarrow$ nejrozumnější
- další
- vždy kompromis

Výměna stránky

- obrácená stránkovací tabulka
- pomalá operace (I/O operace jsou pomalé)
- přidělování rámců
  - kolik rámců přidělit procesu?
  - několik strategií (pevný počet rámců, proměnlivý počet rámců, pracovní množina)
- málo stránek v paměti
  - neustálé výpadky stránky
  - OS spouští další procesy (procesor nic nedělá)
  - trashing

Správa dat

- I/O zařízení (pevný disk)
- uložení dat v rámci procesu je velmi omezené
- dlouhodobé uchování dat
  - možnost uložení velkého množství dat
  - data musí „přežít“ proces, který je využívá
  - k datům může přistupovat více procesů
- operace čtení a zápis nejsou dostatečné
  - jak najít data?
  - jak zajistit vlastnictví dat?
  - jak poznat, kde lze data zapsat?
  - $\rightarrow$ nutnost správy
- evidence volného místa (bloků)
  - seznam
  - bitmapa

Soubor

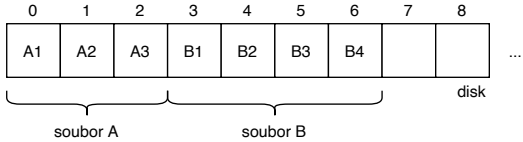
- logická jednotka informace vytvořená/používána procesem
- abstrakce nad fyzickým uložením dat
- persistentní uložení dat
- správa souborů = *souborový systém*
  - například: FAT (různé varianty), UFS, ext3, ext4, NTFS, APFS, ...
  - služby: zabezpečení (oprávnění, šifrování), optimalizace (komprese, de-duplikace bloků), zálohování, integrita dat (kontrolní součty)
- vlastnosti souboru
  - typ souboru
  - atributy (metadata)
  - operace
  - oprávnění přístupu

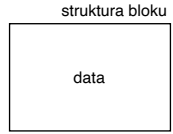
Soubor: Přidělení paměti

- soubor = obvykle proud dat (sekvence bajtů)
- uloženo po blocích na disku
- velikost bloku
  - velké bloky → rychlé, ale nevyužité místo
  - menší bloky → optimální využití místa, ale režie
  - je třeba kompromis
- přidělování bloků (má vliv na výkon)
  - *souvislá alokace*
  - *spojitá alokace*
  - *indexová alokace*
  - *I-node*

Souvislá alokace

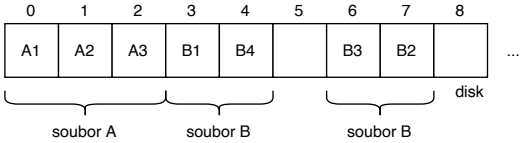
- třeba znát velikost souboru
- vnější fragmentace
- nutná optimalizace využitého prostoru (ta je náročná)
- snadný random-access

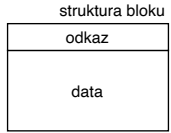




Spojitá alokace

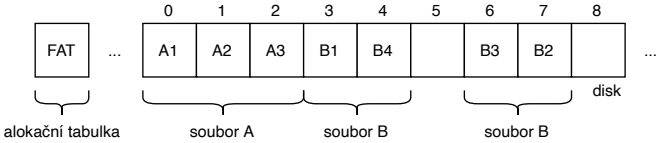
- spojový seznam bloků
- každý blok odkazuje odkaz na další → potlačení vnější fragmentace
- pomalý random-access (procházení seznamem)
- nutné odstranění odkazu z dat (režie)
- porušení principu lokality → nutná konsolidace

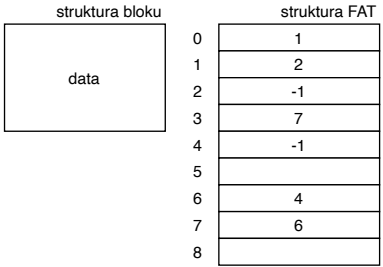




Indexová alokace

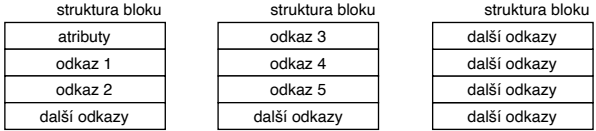
- kombinace předchozích
- odkazy na další bloky uloženy v tabulce (file allocation table, FAT)
- FAT tabulka zabírá místo (velikost závislá na velikosti disku)





I-node

- index-node
- struktura pro reprezentaci souboru (uchovává atributy souboru)
- I-node obsahuje tabulku odkazů na bloky souboru
- uchovává se pole I-node (velikost závislá na počtu souborů)
- používáno v běžných OS



Odbočka: Název souboru

- omezená délka (např. MS-DOS), variabilní délka
- z pohledu uživatele drobnost z pohledu souborového systému zásadní
- jak implementovat variabilní délku?
  - vyhradíme dostatek místa $\rightarrow$ plýtvání (nemůžeme si dovolit)
  - proměnlivá délka bloku reprezentující soubor $\rightarrow$ fragmentace (lepší se vyhnout)
  - omezíme $\rightarrow$ návrat k MS-DOS (nechceme)
  - název souboru = odkaz (porušení lokality)

Adresář

- organizace souborů = adresářová struktura
  - stromová struktura (ne nutně, lze dělat linky)
  - cesta k souboru (absolutní, relativní)
  - zápis cesty lomítko, zpětné lomítko
  - adresáře: `.`, `..`
  - operace s adresáři
- implementace:
  - seznam
  - hash
  - B-stromy

Svazek

- organizace diskového prostoru
- příklad

rozdělení pevného disku

struktura svazku

Odbočka: Žurnálování

- data se zapisují nejdříve do cache
- při chybě data nemusí být zapsána = poškození souborového systému
- žurnálování → pouze konzistentní stavy souborového systému
- změny prováděny v transakcích
- je veden záznam o změnách v případě chyby zotavení dle záznamu
- postup:
  - do žurnálu je zaznamenána informace o požadované změně
  - provedení změny na disku (v transakci)
  - zapsání potvrzení o dokončení změny
  - smazání záznamu o změně

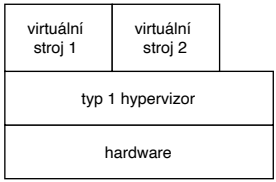
Virtualizace

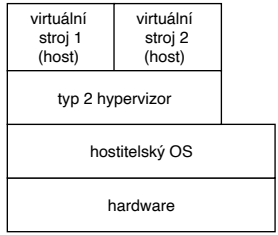
- běh více počítačů na jednom fyzickém hardware
- *hypervisor* vytváří iluzi více *virtuálních strojů* (virtual machine, VM)
- virtuální stroj je nezávislý na ostatních → možnost různých OS
- nutná podpora hardware
- použití: sandboxing, cloud, snadnější údržba, úspora
- je konsolidace bezpečná? → chyby software vs. chyby hardware

Virtualizace

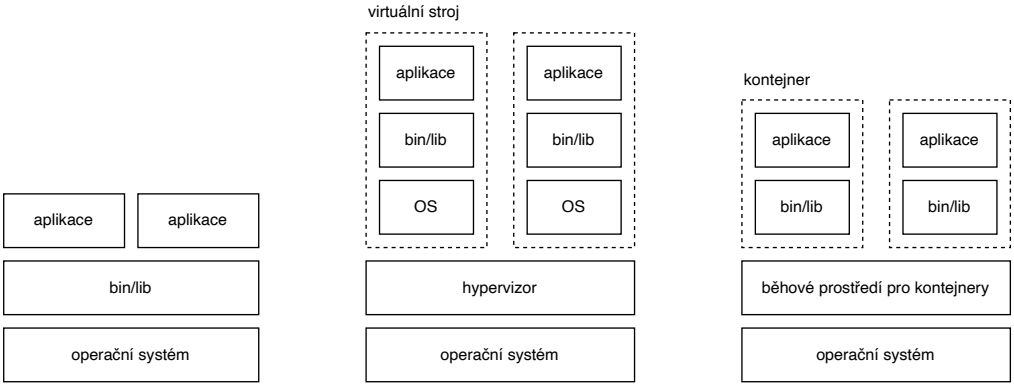
- typy virtualizace
  - plná virtualizace (VirtualBox, vmWare, QEMU)
  - para-virtualizace (Xen)
  - kontejnerizace (Docker)
  - aplikační virtualizace (Java, Python, C#)

- hypervizor
  - typ 1 (Microsoft Hyper-V)
  - typ 2 (Linux KVM, VirtualBox)





Virtualizace: Srovnání nasazení aplikace



Odbočka: Distribuované systémy

- propojení skrze počítačovou síť (naše další velké téma)
- uzly jsou navzájem nezávislé, doposud procesy běžely na jednom uzlu, nyní na různých uzlech $\rightarrow$ třeba synchronizace
- synchronizace = zasílání zpráv (obecně nespolehlivé, problém zpoždění)
- různé časy na uzlech
- problémy
  - vzájemné vyloučení (analogie kritické sekce)
  - volba lídra
  - shoda
  - replikace (problém konzistence dat)
  - globální stav

Operační systémy: Shrnutí

- operační systém a jeho architektura
- procesy a vlákna
- synchronizace procesů a vláken
- fragmentace paměti
- segmentace, stránkování, virtuální paměť
- správa diskového prostoru