Učební texty k státní bakalářské zkoušce
Obecná informatika
Architektury počítačů a sítí

18. června 2011

5 Architektury počítačů a sítí

Požadavky

- Architektury počítače.
- Procesory, multiprocesory.
- Vstupní a výstupní zařízení, ukládání a přenos dat.
- Architektury OS.
- Procesy, vlákna, plánování.
- Synchronizační primitiva, vzájemné vyloučení.
- Zablokování a zotavení z něj.
- Organizace paměti, alokační algoritmy.
- Principy virtuální paměti, stránkování.
- Systémy souborů, adresářové struktury.
- Bezpečnost, autentifikace, autorizace, přístupová práva.
- ISO/OSI vrstevnatá architektura sítí.
- TCP/IP.
- Spojované a nespojované služby, spolehlivost, zabezpečení protokolů.

5.1 Architektury počítače

Definice *(Architektura počítača)*
Architektura počítača popisuje všetko, čo by mal vedieť ten, ktorý programuje v assembleri / tvorí operačný systém. Teda:

- z akých častí – štruktúra počítača, usporadanie
- význam častí – funkcia časti, ich vnútorná štruktúra
- ako spolu časti komunikujú – riadenie komukácie
- ako sa jednotlivé časti ovládajú, aká je ich funkčnosť navonok

Definice *(Víceúrovňová organizace počítače)*

- Mikroprogramová úroveň (priamo technické vybavenie počítača)
- Strojový jazyk počítače (virtuálny stroj nad obvodovým riešením; vybavenie – popis architektúry a organizácie)
- Úroveň operačního systému (doplnenie predchádzajúcej úrovne o súbor makroinštrukcií a novú organizáciu pamäti)
- Úroveň assembleru (najnižšia úroveň ľudsky orientovaného jazyka)
- Úroveň vyšších programovacích jazyků (obecné alebo problémovo orientované; prvá nestrojovo orientovaná úroveň)
- Úroveň aplikačních programů

Je teda potrebné definovať

- Inštrukčný súbor (definícia prechodovej funkcie medzi stavmi počítača, formát inštrukcie, spôsob zápisu, možnosti adresovania operandov)
- Registrový model (rozlišovanie registrov procesoru: podľa voľby, pomocou určenia registru – explicitný/implicitný register; podľa funkcie registru – riadiaci register/register operandu)
- Definice specializovaných jednotek (jednotka na výpočet vo floatoch; fetch/decode/execute jednotky)

- Paralelismy (rozklad na úlohy, ktoré sa dajú spracovať súčasne – granularita (programy, podprogramy, inštrukcie...))
- Stupeň predikce (schopnosť pripraviť sa na očakávanú udalosť (načítanie inštrukcie, nastavenie prenosu dát) – explicitná predikcia, štatistika, heuristiky, adaptívna predikcia)

- Datové struktury a reprezentáciu dát (spôsob uloženia dát v počítači, mapovacie funkcie medzi reálnym svetom a vnútorným uložením, minimálna a maximálna veľkosť adresovateľné jednotky)
- Adresové konvencie (ako sa pristupuje k dátovým štruktúram – *segment+offset* alebo *lineárna adresácia*; veľkosť pamäti a jej šírika, povolené miesta)

- Řízení (spolupráca procesoru a ostatných jednotiek, interakcia s okolím, prerušenia – vnútorne/vonkajšie)
- Vstupy a výstupy (metódy prenosu dát medzi procesorom a ostatnými jednotkami/počítačom a okolím; zahrňuje definície dátových štruktúr, identifikácia zdroja/cieľa, dátových ciest, protokoly, reakcie na chyby).
- Šíře datových cest
- Stupeň sdílení (na úrovni obvodov – zdieľanie obvodov procesoru a IO; na úrovni jednotiek – zdieľanie ALU viacerými procesormi)

Základní definice

- **ALU** (také Processing Element) Aritmeticko-logická jednotka - základní komponenta procesoru (2.základní je řadič).
- **Řadič** (Control Unit) je elektronická řídicí jednotka, realizovaná sekvenčním obvodem, která řídí činnost všech částí počítače. Toto řízení je prováděno pomocí řídicích signálů, které jsou zasílány jednotlivým modulům (dílčím částem počítače). Reakce na řídicí signály - stavy jednotlivých modulů - jsou naopak zasílány zpět řadiči pomocí stavových hlášení. Dílčí částí počítače je např. hlavní paměť, která rovněž obsahuje řadič, který je podřízen hlavnímu řadiči počítače, jenž je součástí CPU.
- **Sběrnice** (Bus) je sada dat.streamů propojující více zařízení. Instruction Stream (řízení komunikace – požadavky/potvrzení, indikace typu dat na datových vodičích) Data Stream (přenos dat mezi zdrojovým a cílovým zařízením, adresy, data, složitější příkazy

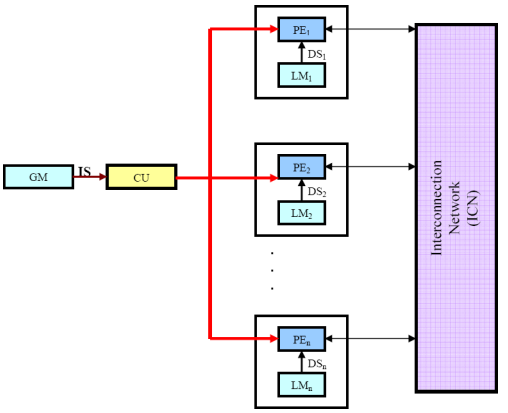
Flynn's taxonomy

The taxonomy of computer systems proposed by M. J. Flynn in 1966 has remained the focal point in the field. This is based on the notion of instruction and data streams that can be simultaneously manipulated by the machine. A stream is just a sequence of items (instruction or data).

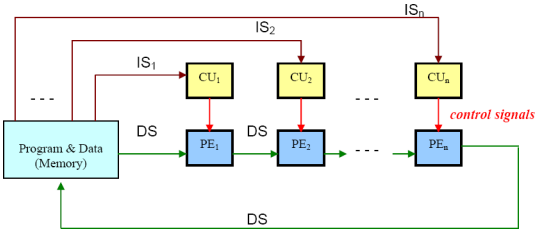
Single Instruction, Single Data stream (SISD) - A sequential computer (Von Neumann) which exploits no parallelism in either the instruction or data streams. Single control unit (CU) fetches single Instruction Stream (IS) from memory. The CU then generates appropriate control signals to direct single processing element (PE or ALU) to operate on single Data Stream (DS) i.e. one operation at a time

Examples of SISD architecture are the traditional uniprocessor machines like a PC (currently manufactured PCs have multiple processors) or old mainframes.

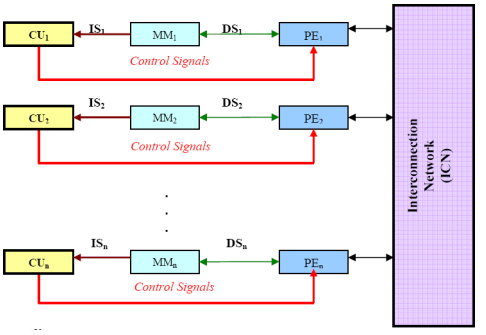
Single Instruction, Multiple Data streams (SIMD) - A computer which exploits multiple data streams against a single instruction stream to perform operations which may be naturally parallelized. For example, an array processor or GPU.



Multiple Instruction, Single Data stream (MISD) - Multiple instructions operate on a single data stream. Uncommon architecture which is generally used for fault tolerance. Heterogeneous systems operate on the same data stream and must agree on the result. Examples include the Space Shuttle flight control computer.



Multiple Instruction, Multiple Data streams (MIMD) - Multiple autonomous processors simultaneously executing different instructions on different data. Distributed systems are generally recognized to be MIMD architectures; either exploiting a single shared memory space or a distributed memory space.



Základní architektury počítačů

Von Neumannova

- Počítač se skládá z řídící jednotky, ALU, paměti a I/O jednotek
- Štruktúra počítača sa nemení typom úlohy (tj. počítač je programovaný obsahem paměti).
- Program se nejprve zavede do paměti, z ní se postupně popořadě vybírají instrukce (a následující krok závisí na předchozím), pořadí lze změnit instrukcemi skoku.
- Do jedné paměti, dělené na buňky stejné velikosti, se ukládají i zpracovávaná data. Data jsou reprezentovaná binárně.

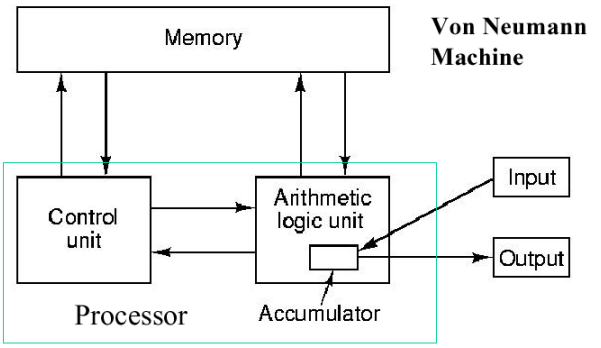
- V každém okamžiku je vykonávána jen jedna činnost. Je to architektura SISD (viz Flynnova taxonomie).

Je pevně daná instrukční sada. Strojová instrukce obsahuje operační znak, který určuje druh operace, počet parametrů atd., a operandovú část – umístnění jednotlivých operandů. Vykonat jednu instrukci znamená:

- (fetch) načítať inštrukciu z pamäti do procesoru
- (decode) zistiť o akú inštrukciu ide
- (load) pripraviť zdrojové operandy
- (execute) vykonať operáciu
- (store) uloziť cieľové operandy

Při vykonávání programu jsou potřebné různé registry – nejdůležitější jsou: PC (Program Counter, obsahuje adresu následující instrukce), IR (Instruction Register, adresa právě vykonávané instrukce), SP (Stack Pointer, ukazatel na vrchol zásobníku), MAR (memory access register – adresa do operační paměti), MBR (memory buffer register, dáta čítána/zapisována do paměti).

Struktura jednoprocesorového počítače podle Von Neumanna:



Control-flow – význačným rysem von Neumannovy architektury je způsob provádění programu. Strojové instrukce, ze kterých se každý přímo spustitelný program skládá, se provádějí sekvenčně, jedna za druhou. Tedy každá instrukce se provede tehdy, až na ni dojde řada, a nad takovými daty, jaká jsou právě k dispozici.

Data-flow (architektura řízená daty) – Alternativa k Control-flow. Okamžik provedení určité akce se řídí připraveností všech dat, která jsou k provedení určité akce zapotřebí. Výhodou je možnost provádět více činností souběžně - tedy větší potenciál paralelismu, který von Neumannově koncepci naopak chybí.

Harvardská

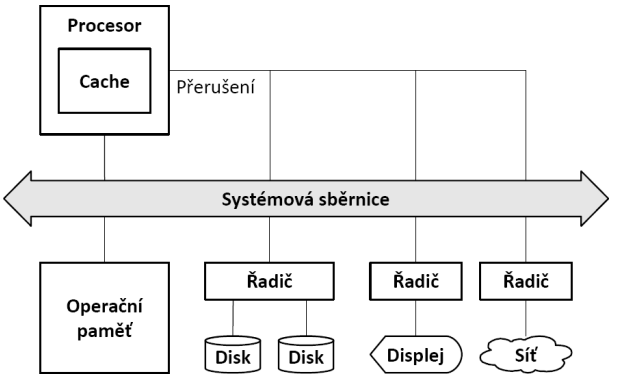
Vytvořena až po Von Neumannově, liší se hlavně tím, že program se ukládá do jiné paměti než data (tzn. jsou 2 druhy paměti – instrukcí a dat). Příkladem jsou DSP procesory a mikrokontrolery.

Např. AVR od Atmelu, a PIC – mají paměť na program a data a RISC instrukční sadu; výhoda oddělených pamětí je, že můžou mít různou bitovou hloubku – 8 bitové data, ale 12-, 14- či 16- bitové instrukce - např. ARM musí občas použít více než jednou instrukci na zpracování obsahu plné velikosti).

Oproti Von Neumannově nehrozí nebezpečí přepsání programu sebou samým, ale kvůli většímu počtu paměťových sběrnic je náročnější na výrobu. Paměť navíc nelze dělit podle potřeby (rozdělení je už dané).

Příklad *Načrtněte typickou architekturu pocitace, ze které bude zrejmé umístění a propojení základnich stavebních prvku (procesor, pameti, radice a zarízeni, sbenice). Ilustrujte na úrovni zakladnich kroků ínstrukcního cyklu jak procesor vykonává program. Popište typy instrukci (a napište jejich příklady), ze kterých se program z pohledu procesoru skládá.*

Typická (sběrnicová) architektura systému:



Zpracování instrukcí

- **čtení** instrukce z paměti na adrese v registru PC (program counter, obsahuje adresu následující instrukce)
- **dekódování** instrukce a čtení operandů z registrů
- **vykonání** operace odpovídající instrukčnímu kódu (operace s obsahem registrů, výpočet adresy a čtení/zápis do paměti, porovnání operandů pro podmíněný skok)
- **uložení** výsledku do registru (výsledek operace s registry, data přečtená z paměti)
- **posun** PC na následující instrukci (následující instrukce následuje bezprostředně za právě čtenou instrukcí, pokud není řečeno jinak - tzn. podmíněný/nepodmíněný skok, výjimka)

Typy instrukci (architektura MIPS):

- operace registr/registr, registr/immediate[1] (ALU operace, přesun dat mezi registry)
- přesuny dat registr/paměť (load/store architektura)
- podmíněné skoky (při rovnosti/nerovnosti obsahu dvou registrů)
- nepodmíněné skoky (včetně nepřímých skoků a skoků do podprogramu)
- speciální instrukce (práce se speciálními registry)

Report *(Bulej)*
Tenhle člověk se v tom vrtal hodně, ale já mám tu výhodu, že jsem na střední chodil na elektroprůmyslovku, takže instrukcí a typů instrukcí jsem mu tam popsal spoustu a i postup, jak procesor vykonává program, jsem věděl do detailů (a do detailů to chtěl). Přesvědčil jsem ho asi hlavně tím, že jsem odpovídal takovým tím "samozřejmým" způsobem ("a když procesor vykoná instrukci, co dělá dál?" -"pokračuje další instrukcí" "no ale co přesně dělá" -"no tenhle postup znovu, načte další instrukci..." "co to přesně znamená?" -"no prostě zvětší instruction pointer o velikost právě zpracované instrukce a tím získá adresu následující instrukce, a opakuje tenhle postup").

Report *(Peterka)*
Peterka si narozdiel od Skopala aj precital to co som si napisal na papier. Popisal som tam Von Neumanna a Harvardsku architekturu (napisal som tam vsetko z vypiskov). K tomu nemal vyhrady. Potom vsak prisla horsia cast ked sa ma zacal vypytovat otazky typu:

myslite si ze je dobre/zle ked moze byt prepisana ta pamat kde sa nachadza program, alebo ake su vyhody programovatelneho radica... dalej sa ma pytal na SISD,SIMD,MISD,MIMD, mal

[1] operand (číslo) uložené přímo ve strojovém kódu

som mu nakreslit MISD ... co som moc nevedel... potom sa ma spytal na rozdiel Instruction flow control/Data flow control... dialog s Peterkom mi prisiel v niektorich castiach skor ako jeho monolog s mojim prikyvovanim hlavy...

Akorat jsem nebyl schopny si vzpomenout na architektury rizenou daty. A chtel vedet kolik radicu a ALU je potreba pri instrukcich SIMD,MIMD. Znamku nevim.

DATA FLOW + CONTROL FLOW (asi :)) podotázka u mikroprocesorů a architektury

Report *(Peterka)*
Von Neumannova architektura - dost temno Harvardská Architektura - záblesky stroje řízené daty - brrr hrůza tady už otázky opravdu nevím... dodám jen nepodceňte hardware - peterka dává vždy jednu hardwarovou a jednu síťovou otázku doplňující: jaké jsou volací konvence v Pascalu a C? viz zpracované otázky co se stane když zavoláme virtuální fci před voláním konstruktoru? konečně jsem se chytil .)

Report *(Tůma)*
Za ferove povazujem ze vzal v uvahu ze som IOI a nedal mi konkretnu podotazku, skor tak prehladovo vsetko od architektur, cez procesory az po IO. Na druhej strane sa dost vrtal v zberniciach o ktorych som toho vedel pramalo (myslim, ze v tych materialoch na statnice tam toho o nich moc nebolo). Ked som zacal hovorit o preruseni, tak ma prerusil s tym, ze ak nechcem dopadnut ako kolega predo mnou (patrne ho vyhodil) tak nech som ticho

5.3 Vstupní a výstupní zařízení

K I/O zařízením je možné přistupovat dvěma způsoby: pomocí **portů** (speciální adresový port CPU) nebo **pamětovým mapováním** (namapování do fyzické paměti).

Zařízení mají různé charakteristiky:

- **druh** – blokové (disk, síťová karta), znakové (klávesnice, myš)
- **přístup** – sekvenční (datová páska), náhodný (hdd, cd)
- **komunikace** – synchronní (pracuje s daty na žádost – disk), asynchronní (nevyžádaná data – síťová karta)
- **sdílení** – sdílené (preemptivní, lze odebrat – síťová karta (po multiplexu OS)), vyhrazené (nepreemptivní – tiskárna, sdílení se realizuje přes *spooling* - frontou). Reálně se rozdíly stírají.
- **rychlost** (od několika Bps po GBps)
- **směr dat** – R/W, R/O (CD-ROM), W/O (tiskárna)

Přenos dat mezi zařízením a CPU/pamětí:

- **PIO (Programmed Input Output)** – data přenášena za účasti CPU (plně zaměstnán), pak přišlo DMA
- **polling** – aktivní čekání na změnu zařízení, přenos provádí CPU
- **přerušení** – asynchronní přerušení od zařízení, přenos provádí CPU,CPU uloží stav programu → stojí čas
- **DMA (Direct Memory Access)** – zařízení si samo řídí přístup na sběrnici a přenáší data z/do paměti; po skončení přenosu přerušení (oznámení o dokončení) např, přenos dat mezi HDD a RAM

Řadič DMA

Obvod pro řízení přenosů na sběrnici

- generuje adresy paměti a periferie, generuje řídící signály pro čtení/zápis
- generuje signály pro procesor, aby zajistil, že procesor nepřistupuje (nezapisuje) na sběrnici
- řadič sám se chová jako periferie
- program nastavuje parametry přenosu, tj. odkud se bude přenášet, kam, a kolik (2 čítače, kanál DMA)
- zařízení připojena na kanál DMA, při přenosu je cílové zařízení aktivováno řadičem, nikoliv vystavením adresy

Posloupnost událostí:

- program nastaví řadič a periferii a povolí přenos
- aktivací signálu DREQx periferie požádá řadič DMA o přenos slova z/do paměti
- řadič DMA zkontroluje nastavení kanálu vyhodnotí prioritu žádosti
- aktivací signálu HOLD řadič DMA požádá CPU o přidělení sběrnice
- pokud CPU nepotřebuje sběrnici, odpojí se od sběrnice a signalizuje HLDA
  - CPU testuje HOLD na začátku strojového cyklu
- po přijetí HLDA řadič připraví sběrnici pro přenos
  - vystaví adresu v paměti a řídící signály pro čtení/zápis z/do paměti/periferie
- řadič DMA aktivuje signál DACKx, kterým vyzve periferii k vystavení/přečtení dat na/ze sběrnice
- v závislosti na režimu buď přenos končí, nebo pokračuje dalším slovem dokud je DREQx aktivní
- při posledním slově řadič aktivuje signál EOP
- při ukončení přenosu řadič uvolní signál HOLD
- procesor uvolní HLDA a připojí se ke sběrnici

Cíle I/O software:

- **Nezávislost zařízení** – programy nemusí dopředu vědět, s jakým přesně zařízením budou pracovat – je jedno jestli pracuji se souborem na pevném disku, disketě nebo na CD-ROM
- **Jednotné pojmenování** (na UNIXu /dev)
- **Připojení (mount)** – časté u vyměnitelných zařízení (disketa); možné i u pevných zařízení (disk); nutné pro správnou funkci cache OS
- **Obsluha chyb** – v mnoha případech oprava bez vědomí uživatele (velmi často způsobeno právě uživatelem)

Report *(Bulej + Yaghob)*
zapisal som vyse strany, ale to co ma yaghob na slidoch a to co je vo vypracovanom ucebnom texte ich ani trochu nezaujimalo. zaujimal ich popis DMA a preruseni, pricom sa pytali ako to presne funguje - chceli popisat instrukcie ako to moze prebiehat, ako sa to presne implementuje apod., co som bohuzial vobec nevedel

5.5 Procesy, vlákna, plánování

Procesy a vlákna

Systémové volání je interface mezi OS (kernelspace) a užívatelskými programy (userspace).
Definice *(Proces)*
Proces je inštancia vykonávaného programu. Kým program je len súbor inštrukcií, proces je vlastný výkon týchto inštrukcií. Proces má vlastný adresný priestor (pamäť), prostriedky, child procesy, globání proměnné, otevřené soubory, práva a napr. aj ID (Process ID).

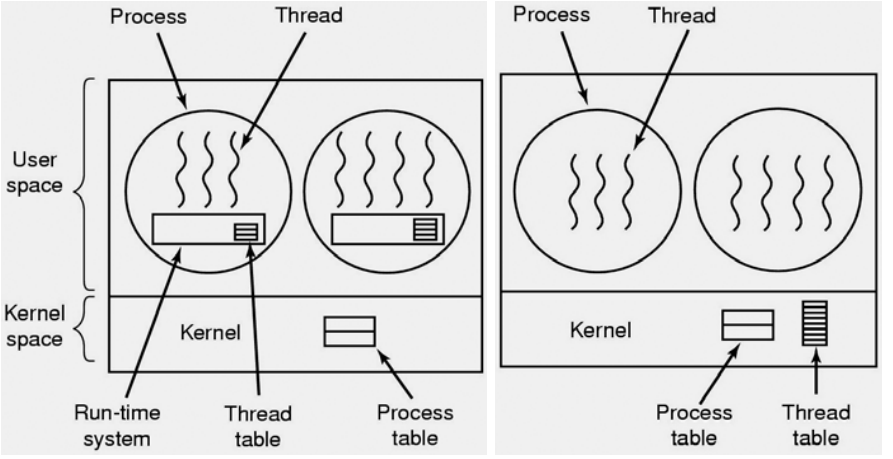
Počas života sa môže proces nachádzať v rôznych stavoch:

- *bežiaci* – jeden proces na procesor,
- *blokovaný* – pri použití blokujúceho volania – I/O disku atď.,
- *pripravený* – skončilo blokovanie; spotreboval všetok pridelený čas resp. vrátil riadenie systému, čaká na nové pridelenie procesora,
- *zombie* – po ukončení procesu, keď už nepracuje – ale ešte nebol vymazaný.

Definice *(Vlákno (Thread))*
Vlákno je možnosť pre program ako sa rozdeliť na dva alebo viac zároveň (resp. pseudo-zároveň) vykonávaných úloh. Oproti procesu mu nie je pridelená vlastná pamäť – je to len miesto vykonávania inštrukcií v programe. Oproti procesu sú jeho atribútmi len **hodnota programového čítača, stav registrov CPU a zásobník**.

Implementace:

- **User Level Threads**(1.diagram) - thread management dělá aplikace (nemusí být podporovány OS), každý proces ma thread table, když systém zablokuje proces zablokují se i všechny jeho thready
- **Kernel Threads**(2.diagram) - thread management dělá OS (musí podporovat), thread table je globální, systém blokuje pouze jednotlivé thready



Multithreading - schopnost systému efektivně používat více threadů, modely:

- **many-to-one** - mnoho user-level threadů je namapováno na jeden kernel thread, používá se na systémech nepodporujících kernel thready (např. Linux - pthreads)
- **one-to-one** - každý user-level thread je namapován na jeden kernel thread (např. win2000, Linux - NGPT)

Plánovanie

Prideľovanie procesorového času jednotlivým procesom má na starosti *plánovač*. Plánovanie pritom môže byť preemptívne (plně v režii OS) alebo nepreemptívne (vyžaduje spolupráci s programem, kooperatívne – alla Win16).

Ciele plánovania (niektoré z nich sú očividne protichodné):

- Spravodlivosť (každy procesor dostane adekvátnu časť času CPU)
- Efektívnosť (plne vyťažený procesor)
- Minimálna doba odpovede
- Průchodnost (maximálny počet spracovaných procesov)
- Minimálna réžia systému

Kritériá plánovania:

- Viazanosť procesu na dané CPU a I/O (presun procesu na iný procesor zaberie veľa prostriedkov)
- Proces je dávkovy/interaktívny?
- Priorita procesu (statická (nemenná – okrem renice) + dynamická, ktorá sa mení v čase kvôli spravodlivosti)
- Ako často proces generuje výpadky stránok (nejaký popis???)
- Kolik skutočného času CPU proces obdržel

Algoritmy:

- **First Come First Served (FCFS)**: nepreemptívny, procesy plánovány v pořadí, v jakém přicházejí, procesy běží dokud neskončí
- **Round Robin**: preemptívne rozšírenie FCFS, každý proces má stejné povolené časové kvantum na běh, po jeho uplynutí je proces přesunut na konec fronty
- **Plánovanie s viacerými frontami**: niekoľko front, procesu z i-tej fronty je pridelený procesor až keď vo frontách $1, \ldots, i-1$ nie je pripravený ziadny proces. Ak proces skončil I/O operáciou, je blokovaný a presunutý do fronty $i-1$, ak skončil preempciou, je pripravený a presunutý do fronty $i+1$.
- **Symmetric multiprocessing (SMP)**: druh víceprocesorových systémů, u kterých jsou všechny procesory v počítači rovnocenné, fronta CPU čakajúcich na pripravené procesy (aktívne (spotrebováva energiu) vs. pasívne čakanie (špeciálne inštrukcie)), vzťah/afinita procesov k CPU
- TODO: Plánovanie windows vs. linux???

Report *(Zavoral)*
zhruba to co je vo vypiskach, diskusia dalej pokracovala o rozdieloch medzi vlaknami a procesmi - napr ako su implementovane vlakna v OS ktory ich nepodporuje (snazil som sa to nejak ukazat na JVM, ale podrobnosti som velmi nevedel, takze to bolo dost napovedy pana Zavorala a par slov odo mna :?).

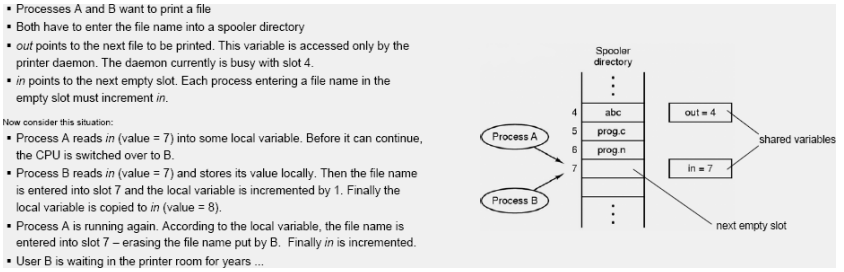
někdo jiny: som letel na Vlaknach (nevedel som ze su reprezentovane hodnotou registrov, prog. citaca a zasobnikom)

5.6 Synchronizační primitiva, vzájemné vyloučení

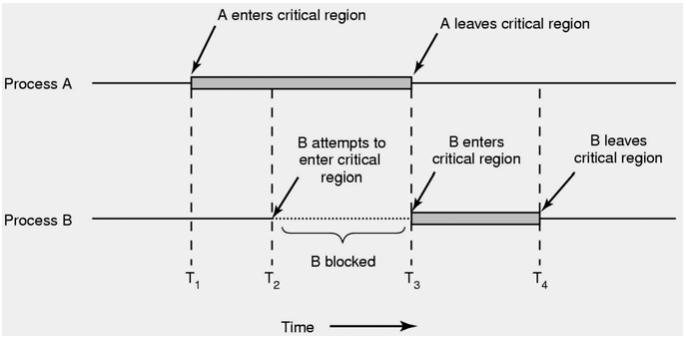
Pojmy

Časové závislé chyby (Race Conditions) Situace kde 2 nebo více procesů přistupuje ke stejnému sdílenému prostředku, a finální výsledek záleží na kdo proběhne kdy se jmenuje *race conditions*

Příklad na tiskové frontě:



Kritická sekce (Critical Regions) část programu, která dokud není dokončena není možné začít jinou (např. používá sdílené prostředky)



Vzájemné vyloučení (Mutual Exclusion) kritickou operaci provádí nejvýše jeden proces. Podmínky vzájemného vyloučení:

1. Žádné dva procesy nemohou být najednou ve stejné kritické sekci
2. Nemohou být učiněny žádné předpoklady o rychlosti procesu (žádné odhady rychlosti nebo priorit procesu, musí fungovat se všemi procesy)
3. Žádný proces mimo kritickou sekci nesmí blokovat jiný proces
4. Žádný proces nesmí čekat nekonečně dlouho v kritické sekci (jinak dead-lock)

Metody dosáhnutí vzájemného vyloučení: aktivní čekání (busy waiting) a pasivní čekání/blokování.

Aktivní čekání (Busy Waiting)

Vlastnosti: **spotřebovává čas procesoru**, vhodnější pro předpokládané krátké doby čekání, nespotřebovává prostředky OS, rychlejší.

Navrhovaná řešení:

- **Zakázání přerušení**] nevhodné - proces má plnou kontrolu nad počítačem
- **Zámky v proměnné**] nefungují - mezi přečtěníma nastavením locku může být program přerušen - pak by si "nevšim" lock == 1 a vesel pokračoval, akorát jsme přidali novou race condition.

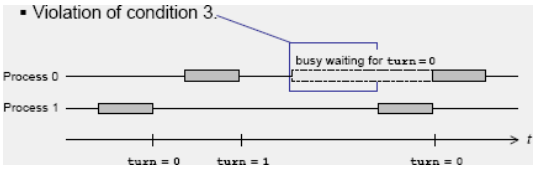
```
int lock;
void proc(void) {
  for (;;) {
    nekritická_sekce();
```

```
    while (lock != 0);
    lock = 1;
    kritická_sekce();
    lock = 0;
  }
}
```

- **Důsledné střídání (Strict Alternation)** funguje ale porušuje podmínku 3 - proměnná turn hlídá kdo je na řadě

```
int turn = 0;

void p1(void)                    void p2(void)
{                                {
  for (;;) {                       for (;;) {
    while (turn != 0);               while (turn != 1);
    kritická_sekce();                kritická_sekce();
    turn = 1;                        turn = 0;
    nekritická_sekce();              nekritická_sekce();
  }                                  }
}                                }
```



Petersonovo řešení - zobecnění pro N procesů:

```
#define N 2                      /* počet procesů */

int turn;
int interested[N];               /* kdo má zájem */

void enter_region(int proc) { /* proc: kdo vstupuje */
int other = 1-proc;              /* číslo opačného procesu */
    interested[proc] = TRUE;     /* mám zájem o vstup */
    turn = proc;                 /* nastav příznak */
    while (turn == proc && interested[other] == TRUE);
}

void leave_region(int proc) { /* proc: kdo vystupuje */
    interested[proc] = FALSE;    /* už odcházím */
}
```

- **Instrukce Test-and-Set Lock (TSL)** - atomická operace na úrovni strojového kódu, nemůže být přerušena je nutné aby ji podporoval HW (všechny současné procesory nějakou mají):
- implementace spin-locku (druh zámku, na nějž je třeba aktivně čekat – čekající proces při čekání na spinlock spotřebovává systémové prostředky) - pořád ale méně prostředků než předchozí řešení

```
enter_region:
    tsl   R,lock           ; načti zámek do registru R a
                           ; nastav zámek na 1
    cmp   R,#0             ; byl zámek nulový?
    jnz   enter_region     ; byl-li nenulový, znova
```

```
    ret                     ; návrat k volajícímu - vstup do
                            ; kritické sekce
leave_region:
    mov   lock,#0           ; ulož do zámku 0
    ret                     ; návrat k volajícímu
```

Pasivní čekání

Vlastnosti: proces je ve stavu blokován, vhodné pro delší doby čekání, **spotřebovává prostředky OS**, pomalejší.

Postup používající Sleep/Wakeup (implementovány OS, atomické operace - sleep uspí volající proces, wakeup probudí udaný proces) nefunguje (viz Problém producent/konzument).

- **Semafory** – počítá počet probuzených, reprezentace volných a přidělených prostředků
  Atomické operace (nesmí být přerušeny):
  down(semaphore* s) – zabere semafor (`s--;`) pokud je volný (`s>0`), jinak čeká na uvolnění
  up(semaphore* s) – uvolní semafor (`s++;`), vzbudí čekající proces (pokud existuje)
  -nutná podpora OS (většinou v kernelu)
- **Mutex** Špeciálny (binárny) typ semaforu, kde sú povolené len hodnoty 0 a 1 (v Up sa miesto $s := s+1$ volá $s := 1$) sa nazýva *mutex* a používa sa na riadenie prístupu k jednej premennej. Většinou pomocí TSL.
- **Monitory**
  Implementovány překladačem, lze si představit jako třídu C++ (všechny proměnné privátní, funkce mohou být i veřejné), vzájemné vyloučení v jedné instanci (zajištěno synchronizací na vstupu a výstupu do/z veřejných funkcí, synchronizace implementována blokovacím primitivem OS). ???TODO
- **Zprávy**
  Operace SEND a RECEIVE, zablokování odesílatele/příjemce, adresace proces/mailbox, rendez-vous...
- **RWL - read-write lock**, **bariéry**...
  Ekvivalence primitiv - pomocí jednoho blokovacího primitiva lze implementovat jiné blokovací primitivum.
  Rozdíly mezi platformami: Windows - jednotné funkce pro pasivní čekání, čekání na více primitiv, timeouty. Unix - OS implementuje semafor, knihovna pthread.

Klasické synchronizační problémy

- **Problém producent/konzument**
  Producent vyrába predmety, konzument ich spotrebúva. Medzi nimi je buffer pevnej veľkosti (N). Konzument nemá čo spotrebúvať ak je buffer prázdny; producent prestane vyrábať, ak je buffer plný.

```
int N = 100;
int count = 0;
void producer(void) {
    int item;
    while(TRUE) {
        produce_item(&item);
        if(count==N) sleep ();
        enter_item(item);
        count++;
        if(count == 1) wake(consumer);
    }
}
void consumer(void) {
```

```
    int item;
    while(TRUE) {
        if(count==0) /*pozice A*/ sleep ();
        remove_item(&item);
        count--;
        if(count==N-1)
            wake(producer);
        consume_item(&item);
    }
}
```

1. Buffer je prázdny, a konzument práve prečítal count, aby zistil, či je rovný nule
2. Preplánovanie na producenta ("pozice A")
3. Producent vytvoří item a zvýší count
4. Producent zistí, či je count rovný jednej. Zistí že áno, čo znamená že konzument bol predtým zablokovaný (pretože muselo byť 0), a zavolá wakeup
5. Teraz môže dôjsť k zablokovaniu: konzument pokračuje na "pozici A" a uspí se, pretože si myslí, že nemá čo zobrať; producent bude chvíľu produkovať a dôjde "preplneniu" ⇒ uspí sa; spí producent aj konzument :o)

Řešení pomocí semaforu:

```
#define N 10
typedef int semaphore;      //a semaphore is an integer
semaphore empty = N;        //counting empty slots
semaphore full = 0;         //counting full slots
semaphore mutex = 1;        //mutual exclusion on buffer access

void producer() {
  while (TRUE) {
    int item = produce_item();
    down(&empty);           //possibly sleep, decrement empty counter
    down(&mutex);           //possibly sleep, claim mutex (set it to 0) thereafter
    insert_item(item);
    up(&mutex);             //release mutex, wake up other process
    up(&full);              //increment full counter, possibly wake up other ...
  }
}

void consumer() {
  while(TRUE) {
    down(&full);            //possibly sleep, decrement full counter
    down(&mutex);           //possibly sleep, claim mutex (set it to 0) thereafter
    item = remove_item();
    up(&mutex);             //release mutex, wake up other process
    up(&empty);             //increment empty counter, possibly wake up other ...
    consume_item(item);
  }
}
```

- **Problém obědvajících filosofů**
  Pět filosofů sedí okolo kulatého stolu. Každý filosof má před sebou talíř špaget a jednu vidličku. Špagety jsou bohužel slizké a je třeba je jíst dvěma vidličkami. Život filosofa sestává z období jídla a období přemýšlení. Když dostane hlad, pokusí se vzít dvě vidličky, když se mu to podaří, nají se a vidličky odloží.

- **Problém ospalého holiče**
  Holič má ve své oficíně křeslo na holení zákazníka a pevný počet sedaček pro čekající zákazníky. Pokud v oficíně nikdo není, holič se posadí a spí. Pokud přijde první zákazník a holič spí, probudí se a posadí si zákazníka do křesla. Pokud přijde zákazník a holič už střihá a je volné místo v čekárně, posadí se, jinak odejde.

Report *(Bulej)*

1. *priklad Producent-konzument pomoci semaforu*
2. *stacilo napsat aktivni vs. pasivni, kriticka sekce, spinlock, semafor (obecne monitor) a pak nasledovalo par otazek, zda je mozne naprogramovat synch. primitivum bez podpory HW*

Report *(Bulej)*
Myslím, že jsem je pochopil, až když mi to pan Skopal vysvětlil. To, co je v materiálech opravdu nestačí. TSL je dobrý v tom, že má pravě operaci Test and Set Lock jako atomickou. Pak jsem se pokoušel udělat semafor pro problém producent a konzument a dělal jsem ho uplně špatně

Report *(Hnětýnka)*
na jedničku musíte umět praktické užití (např. z více mutexů postavit semafor)

Report *(Bednárek)*
Na tuhle jsem byl připravený ze zadaných otázek asi nejhůř, kupodivu jsem toho k ní ale nakonec na papír vyplodil poměrně dost a dostal k tématu jen málo doplňujících otázek (nějaké drobné praktické a jak implementovat mutex bez podpory OS, tj. pomocí test-and-set instrukce), pak se plynule a nepozorovaně přešlo na zablokování a zotavení z něj. Něco jsem věděl, vzpomněl jsem si na 3 ze 4 Coffmanových podmínkek a jejich ošetření, čtvrtou jsem pak vymyslel s Bednárkovou vydatnou pomocí. Žádné otázky na "klasické synchronizační problémy" nebo Petersonovo řešení, tj. věci, o kterých jsem se sám radši nezmínil.

na konci sa opytal, ze aky problem okrem vyhladovania moze nastat... deadlock

Sleep/wakeup, semafory, monitory, správy, polling - u každého ako funguje a či to robí aplikácia/OS/HW. Potom sme sa pobavili o možnosti implementovať jedno druhým.

5.7 Zablokování a zotavení z něj

Prostředek je cokoliv, k čemu je potřeba hlídat přístup (HW zařízení – tiskárny, cpu; informace – záznamy v DB). Je možné je rozdělit na *odnímatelné* (lze odejmout procesu bez následků – CPU, paměť) a *neodnímatelné* (nelze odejmnout bez nebezpečí selhání výpočtu – CD-ROM, tiskárna... tento druh způsobuje problémy).

Práce s prostředky probíhá v několika krocích: *žádost o prostředek* (blokující, právě tady dochází k zablokování), *používání* (např. tisk), *odevzdání* (dobrovolné/při skončení procesu).

Množina procesů je *zablokována*, jestliže každý proces z této množiny čeká na událost, kterou může způsobit pouze jiný proces z této množiny.

Coffmanovy podmínky

Splnenie týchto podmienok je nutné pre zablokovanie:

1. **Výlučný přístup** – každý prostředek je buď vlastněn právě jedním procesem nebo je volný.
2. **Drž a čekej** – procesy aktuálně vlastnící nějaké prostředky mohou žádat o další.
3. **Neodnímatelnost** – přidělené prostředky nemohou být procesům odebrány.
4. **Čekání do kruhu** – existuje kruhový řetěz procesů, kde každý z nich čeká na prostředek vlastněný dalším článkem řetězu.

Řešení zablokování

- **Pštrosí algoritmus** – Zablokování se ani nedetekuje, ani se mu nezabraňuje, ani se neodstraňuje, Uživatel sám rozhodne o řešení (kill). Nespotřebovává prostředky OS – nemá režii ani neomezuje podmínky provozu. (Nejčastější řešení – Unix, Windows)
- **Detekce a zotavení** – Hledá kružnici v orientovaném grafu (hrany vedou od procesu, který čeká, k procesu, který prostředek vlastní), pokud tam je kružnice, nastalo zablokování a je třeba ho řešit:
  - *Odebrání prostředku* – dohled operátora, pouze na přechodnou dobu
  - *Zabíjení procesů z cyklu* (resp. mimo cyklus vlastnící identický prostředek)
  - *Rollback* (OS ukládá stav procesů, při zablokování se některé procesy vrátí do předchozího stavu ⇒ ztracena práce... obdoba u DB)
- **Vyhýbání se** – Bezpečný stav (procesy/prostředky nejsou zablokovány, existuje cesta, jak uspokojit všechny požadavky na prostředky spouštěním procesů v jistém pořadí); Viď. bankéřův algoritmus. Nutné je předem znát všechny prostředky, které budou programy potřebovat; OS pak dává prostředky tomu, který je nejblíž svému maximu potřeby a navíc pro který je prostředků dost na dokončení. Dnes se moc nepoužívá.
- **Předcházení (prevence)** – napadení jedné z Coffmanovy podmínek
  1. *Výlučný přístup – spooling* (prostriedky spravuje jeden systemový proces, ktory dohliada na to, aby jeho stav bol konzistentny (tiskarna) – pozor na místo na disku)
  2. *Drž a čekej* – žádat o všechny prostředky před startem procesu. Nejprve všechno uvolnit a pak znovu žádat o všechny najednou
  3. *Neodnímatelnost* – odnímatelné prostředky mohou být odejmuty bez následků (procesor-přeplánování, paměť-swapping), neodnímatelné nelze bez nebezpečí selhání výpočtu
  4. *Čekání do kruhu* – všechny prostředky jednoznačně očíslovány (stačí prostředky v nějakém kontextu), procesy mohou žádat o prostředky jen ve vzestupném pořadí
- *Dvojfázové zamykání* – nejprve postupně všechno zamykám (první fáze). Potom se může pracovat se zamčenými prostředky – a na závěr se už jen odemyká (druhá fáze) – viď transakční spracování u databází ((striktní/konzervativní) dvoufázové zpracování)

Bankéřův algoritmus: Bankéř má klienty a těm slíbil jistou výšku úvěru. Bankéř ví, že ne všichni klienti potřebují plnou výši úvěru najednou. Klienti občas navštíví banku a žádají postupně o prostředky do maximální výšky úvěru. Až klient skončí s obchodem, vrátí bance vypůjčené peníze. Bankéř peníze půjčí pouze tehdy, zůstane-li banka v bezpečném stavu.
Problémy: složitost $O(N^2)$, požadované info je typicky nedostupné, efektivnější bývá řešit až vzniklé problémy

5.8 Organizace paměti, alokační algoritmy

Hierarchie paměti (směrem odshora dolů roste velikost, cena na bajt a rychlost klesá – a naopak...):

- *registry CPU* — 10ky-100vky bajtů (IA-32: obecné registy pár 10tek), IA-64 – až kB (extrém), stejně rychlé jako CPU.
- *cache* — z pohledu aplikací není přímo adresovatelná; dnes řádově MB, rozdělení podle účelu, několik vrstev. L1 cache (cca 10ky kB) – dělené instrukce/data; L2 (cca MB) sdílené instr&data, běží na rychlosti CPU (dřív bývala pomalejší), servery – L3 (cca 10MB). Vyrovnává rozdíl rychlosti CPU a RAM. Využívá lokality programů – cyklení na místě; sekvenčního přístupu k datům. Pokud nenajdu co chci v cache – cache-miss, načítá se potřebné z RAM (po blocích), jinak (v 95-7% případů) nastane cache-hit, tj. požadovaná data v cache opravdu jsou a do RAM nemusím.
- *hlavní paměť* (RAM) — přímo adresovatelná procesorem, 100MB – GB; pomalejší než CPU; CAS – doba přístupu na urč. místo – nejvíc zdržuje (v 1 sloupci už čte rychle, dat. tok

dostatečný), další – latence – doba než data dotečou do CPU – hraje roli vzdálenost (AMD-integrovaný řadič v CPU)

- *pomocná paměť* — není přímo adresovatelná, typicky HDD; náh. přístup, ale pomalejší. 100GB, různé druhy – IDE, SATA, SCSI; nejvíc zdržuje přístupová doba (čas seeku) cca 2-10ms; obvykle sektor – 512 B; roli hraje i rychlost otáčení (4200 – 15000 RPM) – taky řádově ms.
- *zálohovací paměť* — nejpomalejší, z teorie největší, dnes ale neplatí; typicky – pásky; pro větší kapacitu – autoloadery ; sekvenční přístup; dnes – kvůli rychlosti často zálohování RAIDem.

Správce paměti: část OS, která spravuje paměťovou hierarchii se nazývá správce paměti (memory manager):

- udržuje informace o volné/plné části paměti
- stará se o přidělování paměti
- a výměnu paměti s diskem

Přiřazení adresy

- při překladu (je již známo umístění procesu, generuje se absolutní kód, PS: statické linkování)
- při zavádění (OS rozhodne o umístění – generuje se kód s relokacemi, PS: dynamické linkování)
- za běhu (proces se může stěhovat i za běhu, relokační registr)

Overlay – Proces potřebuje více paměti než je skutečně k dispozici. Programátor tedy rozdělí program na nezávislé části (které s v paměti podle potřeby vyměňnují) a část nezbytnou pro všechny části... Používáno hlavně v DOSu, nyní se stejného cíle dosahuje pomocí virtuální paměti

Výměna (swapping) – dělá se, protože proces musí být v hlavní paměti, aby jeho instrukce mohly být vykonávány procesorem... Jde o výměnu obsahu paměti mezi hlavní a záložní.

Překlad adresy – nutný, protože proces pracuje v logickém (virtuálním) adresovém prostoru, ale HW pracuje s fyzickým adresovým prostorem...

Spojité přidělování paměti – přidělení jednoho bloku / více pamětových oddílů (*pevně* – paměť pevně rozdělena na části pro různé velikosti bloků/*volně* – v libovolné části volné paměti může být alokován libovolně veliký blok)

Informace o obsazení paměti – bitová mapa / spojový seznam volných bloků (spojování uvolněného bloku se sousedy)

Alokační algoritmy:

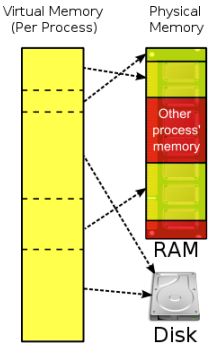
- *First-fit* – první volný dostatečné velikosti – rychlý, občas ale rozdělí velkou díru
- *Next-fit* – další volný dostatečné velikosti, začíná se na podlední prohledávané pozici – jako First-fit, ale rychlejší
- *Best-fit* – nejmenší volný dostatečné velikosti – pomalý (prohledává celý seznam), zanechává malinké díry (ale nechává velké díry vcelku)
- *Worst-fit* – největší volný – pomalý (prohledává celý seznam), rozdělí velké díry
- *Buddy systém* – paměť rozdělena na bloky o velikosti 2^n, bloky stejné velikosti v seznamu, při přidělení zaokrouhlit na nejbližší 2^n, pokud není volný, rozštípnou se větší bloky na příslušné menší velikosti, při uvolnění paměti se slučují sousední bloky (buddy)

Fragmentace paměti:

- *externí* – volný prostor rozdělen na malé kousky, pravidlo 50% – po nějaké době běhu programu bude cca 50% paměti fragmentováno a u toho to zůstává - plýtvání místem mezi alokovanými oblastmi
- *interní* – nevyužití celého přiděleného prostoru (50% velikosti posledního bloku prostoru nevyužito) - plýtvání místem uvnitř alokované oblasti
- *sesypání* – pouze při přiřazení adresy za běhu, nebo segmentaci – nelze při statickém přidělení adresy

5.9 Principy virtuální paměti, stránkování, algoritmy pro výměnu stránek, výpadek stránky, stránkovací tabulky, segmentace

Virtuální paměť



Virtuální paměť způsob správy operační paměti počítače, který umožňuje předložit běžícímu procesu adresní prostor paměti, který je uspořádán jinak nebo je dokonce větší, než je fyzicky připojená operační paměť RAM. Z tohoto důvodu procesor rozlišuje mezi virtuálními adresami (pracují s nimi strojové instrukce, resp. běžící proces) a fyzickými adresami paměti (odkazují na konkrétní adresové buňky paměti RAM). Převod mezi virtuální a fyzickou adresou je zajišťován samotným procesorem v MMU (je nutná hardwarová podpora) nebo samostatným obvodem.[2]

- Umožňuje sdílení paměti (operačním systémem)
- Vzájemná ochrana programů (v současnosti je důležitější ochrana dat než využití principu lokalit), tzn. to aby jeden program nepřepisoval druhému programu jeho data a tak.
- Každý běžící program pracuje se **svým** virtuálním adresním prostorem

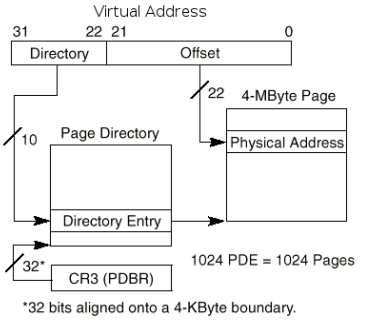
Existují dvě základní metody implementace virtuální paměti – stránkování a segmentace.

Stránkování

Při stránkování je paměť rozdělena na větší úseky stejné velikosti, které se nazývají stránky. Správa virtuální paměti rozhoduje samostatně o tom, která paměťová stránka bude zavedena do vnitřní paměti a která bude odložena do odkládacího prostoru (swapu).

Podporované všemi velkými CPU a OS, jednorozměrný VAP (virtuální adresní prostor).

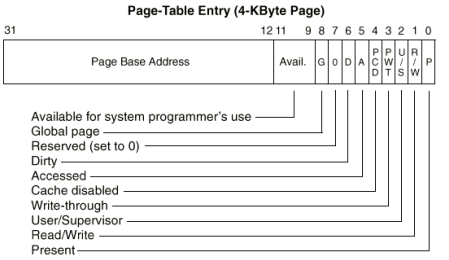
- VAP rozdělen na stránky (velikost je mocnina 2), FAP na rámce (úseky stejné délky)
- **převod stránkovací tabulkou** - každý proces má svojí, příznak existence mapování (v.stránka není v FAP → událost "výpadek stránky" → synchronní přerušení) umístěna v fyzické paměti



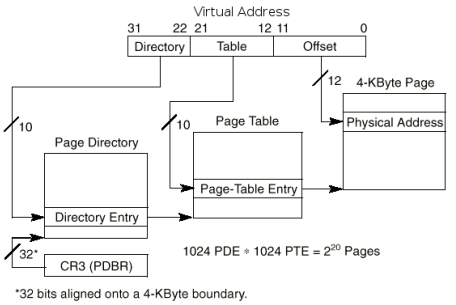
- umožnuje *oddělené VAP* i *sdílenou paměť* - mapování virtuální stránky 2 procesů na jednu fyzickou
- OS mění tabulky stránek změnou PTBR (Page Table Base Register) - PTBR obsahuje bázovou adresu tabulky stránek procesu

[2] šlo by to bez HW podpory? Jistě že ano VM to tak dělají, nicméně rychlost není nijak osňující, proto se do novych strojů už přidává jejich HW podpora (neplácám?!).

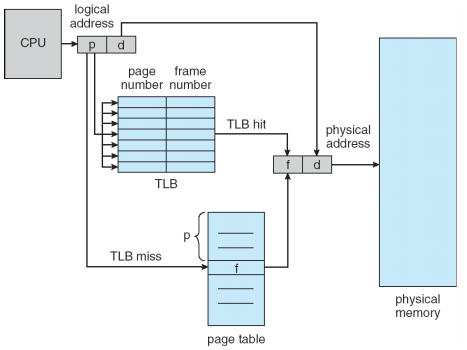
- Příklad: položka stránkovací tabulky Intel IA32 (= x86) → její struktura je závislá na architektuře CPU



- **víceúrovňové stránkování** (např. kvůli velikosti - jedna tabulka je už moc velká =>pomalá)



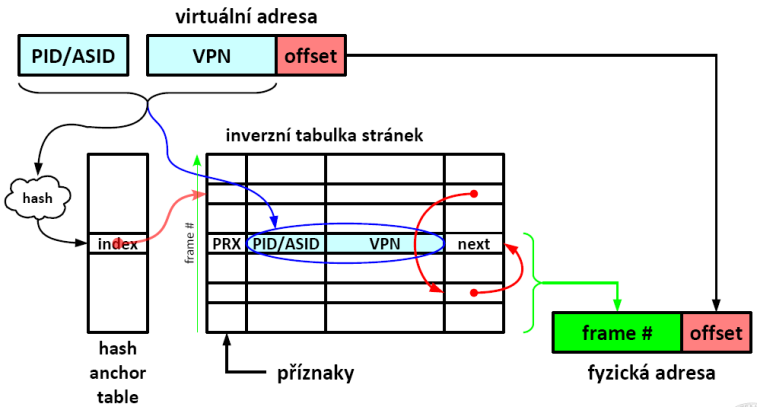
- **TLB (Translation Lookaside Buffer)** - asociativní paměť sloužící na rychlé mapování virtuální stránky na fyzickou, vyhledává se v ní paralelně, typicky ma 128-256 položek, využívá princip prostorové lokality programů (většina programů provádí velký počet přístupů k malému počtu stránek) umístěva většinou v L1 nebo L2 cache na procesoru



Platné	Stránka	Dirty	Ochrana	Rámec
1	125	1	RW	654
1	44	0	RW	132
1	485	1	RW	12
0				
1	55	0	R X	142
1	111	0	R X	54
1	108	1	RW	54
1	88	0	R X	543

...**0-úrovňové stránkování** - procesor hledá pouze TLB, zbytek řeší OS (oblíbené u 64-bitových CPU - UltraSPARC III)

- **inverzní stránkování** (např. když FAP je menší než VAP, 64-bitové CPU - IA-64 = UtraSPARC, PowerPC) - inverzní stránkovací tabulka (IPT) nad rámci (nikoliv stránkami) společná pro všechny procesy, pro vyhledávání se používá hashovací tabulka



Akce vykonávané při výpadku stránky:

- výjimka procesoru
- uložit stav CPU (kontext)
- zjistit VA
- kontrola platnosti adresy a práv
- nalezení vhodného rámce
- zrušit mapování na nalezený rámec
- pokud je vyhazovaný rámec vyhazován, spustit ukládání na disk
- načíst z disku požadovanou stránku do rámce
- zavést mapování
- obnovit kontext

Při implementaci stránkování je nutno brat v úvahu:

- *znovuspuštění instrukce* — je potřeba aby procesor po výpadku zkusil přístup do paměti znova. dnes umí všechny CPU, např. 68xxx - problémy (přerušení v půlce instrukce)
- *sdílení stránek* — jednomu rámci odp. víc stránek → pokud s ním něco dělám, týká se to všech stránek! musím vše ost. odmapovat. musím si pamatovat mapování pro každý rámec - obrácené tabulky.
- *velikost stránek*

– velké stránky → fragmentace
– malé stránky → mnoho registrů, zvyšuje cenu výpočtů a zpomaluje chod
– optimum 1-4kB

- *odstranění položky z TLB při rušení mapování* — nestačí změnit tabulky, musí se vyhodit i z TLB (kde to může, ale nemusí být). problém - u multiprocesorů má každá CPU vlastní TLB, tabulky jsou sdílené → CPU při rušení mapování musí poslat interrupt s rozkazem ke smazání všem (i sobě), počkat na potvrzení akce od všech.

Příklad *Uvažujte procesor, který podporuje stránkovani, má dvouúrovnove stránkovaci tabulky, velikost virtuální i fyzické adresy 32 bitu, velikost stránky 4 kB. Nakreslete formát strankovací tabulky (položky potřebné pro překlad adresy i typické další příznakové bity, nezadané detaily rozumne zvolte) a v nem ilustrujte, jak se prelozi virtuálni adresa 12345678h (nezadané konstanty tvorící konkrétní obsah tabulky opet rozumne zvolte).*
Pozn.: h nakonci znamená že je číslo v hex (assembler)

Algoritmy pro výměnu stránek (výběr oběti)

- **Optimální stránka** (v okamžiku výpadku stránky vybírám stránku, na níž se přistoupí za největší počet instrukcí) - nelze implementovat
- **NRU** (Not Recently Used) - každá stránka má příznaky Accessed a Dirty (typicky implementovatelné v HW, možno simulovat SW); jednou za čas se smažou všechna A; při výpadku rozdělím stránky podle A,D a vyberu stránku z nejnižší (0,1..4) neprázdné třídy:

	A	D
0	0	0
1	0	1
2	1	0
3	1	1

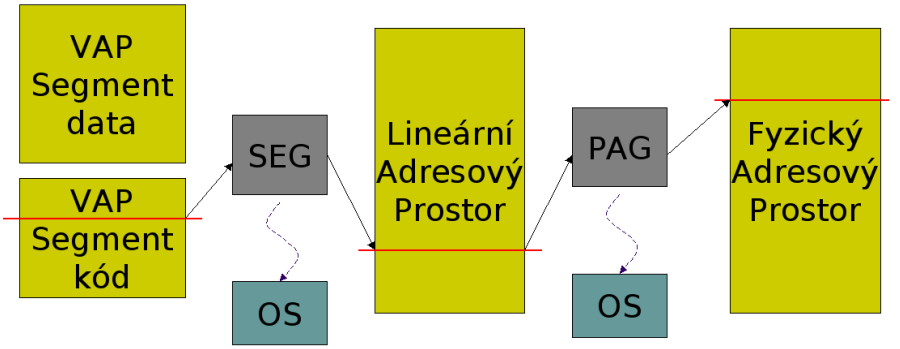
- **FIFO** - vyhodit nejdéle namapovanou stránku - vykazuje anomálie - Belady (zvětšení počtu výpadků stránky, když zvýšíme počet stránek v paměti), druhá šance (úprava FIFO; pokud A=1, zařadím na konec FIFO... nevykazuje anomálie)
- **Hodiny** - modifikace druhé šance: kruhový zoznam stránek + iterátor na ukazující na nejstarší stránku v zoznamu. Při výpadku (a neexistenci volého rámce) se zjistí, jestli má *iterator nastavený příznak Accessed. Jestli ne, tato stránka bude nahrazena - v opačném případě se Accessed příznak zruší a iterator++. Toto se opakuje, dokud nedojde k výměně...
- **LRU** (Least Recently Used) - často používané stránky v posledním krátkém časovém úšeku budou znovu použity, čítač použití stránek, možné implementovat v HW
- **NFU** (Not Frequently Used) - SW simulace LRU, SW čítač ke každé stránce; jednou za čas projdu všechna A a přičtu je k odpovídajícím čítačům; vybírám stránku s nejnižším čítačem; nezapomíná - je možná modifikace se stárnutím čítače

Segmentace

dnes pouze Intel IA-32, dvojrozměrný VAP

- rozdělení programu na segmenty (napr. podle částí s různými vlastnostmi - kód, data, zásobníky...), různé délky segmentů, ktoré můžou měnit svoji délku za běhu
- VAP dvourozměrný (segment, offset), FAP jednorozměrný (vyzerá jako při spojitém pridělování paměti)
- segmentová převodní tabulka (VA se skládá ze dvou částí S:D, v tabulce se najde adresa segmentu S... k této adrese se poté přičte D, co je umístnění adresy v FA), příznak existence mapování
- při výpadku je nutné měnit celý segment (ty mohou být velké), je možné segmenty sesypat - ale nelze mít segment větší než FAP

Segmentaci je možné kombinovat se stránkováním (odstraňuje nevýhody segmentace, neprovádí se výpadky segmentů):



Report *(Bednárek)*
Třeba mě překvapila Bednákova otázka u jednoúrovňovýho stránkování, když se zeptal, co z toho dělá OS a co procesor (jako co je děláno hardwareově a co softwareově).

Report *(Bulej)*
Strankovaci tabulku ma kazdy proces vlastni -¿ ochrana pameti, nemuze pristoupit na cizi stranky (mozna ze to v materialech ke statnicim je, ale ja to z nich nepochopil...)

Report *(Skopal)*
Nejdriv jsem popsal k cemu to je a potom princip. Chtel popsat postup toho co se deje, kdyz se hleda nejaky pointer. Co dela HW, co OS. Pak se zeptal jestli by slo udelat strankovani bez HW podpory (coz rozumne nejde, muselo by se to resit i v prekladacich a bylo by to nefektivni). Pak se zeptal na algoritmy vyhazovani stranek. Popsal jsem FIFO a NRU a to mu stacilo. Na segmentaci nastesti nedoslo. Celkove velmi prijemne zkouseni. V zasade se spokojil s principy a nestoural moc do detailu.

Report *(Kofron)*
klasika, prečo, kde, ako ... prečo to funguje, čo sa robí pri výpadku stránky, prečo dve inštancie jedného programu nelezú do kapusty aj keď pracujú s rovnakými virtuálnymi adresami (každý má vlastnú str. tabuľku), tvar adresy, prevody...

každý proces má vlastní tabulku, její adresa v registru

5.10 Systémy souborů, adresářové struktury

Definice *(soubor)*
Soubor je pojmenovaná množina souvisejících informací, která leží v pomocné paměti (na disku).
Soubor je abstrakce, která umožňuje uložit informaci na disk a později ji přečíst. Abstrakce odstiňuje uživatele od podrobností práce s disky.

Soubory

- pojmenování souboru (umožňuje uživateli přístup k jeho datům; přesná pravidla pojmenování určuje OS - malá vs. velká písmenka, speciální znaky, délka jména, přípony a jejich význam)
- atributy souborů (opět určuje OS) - jméno, typ, umístění, velikost, ochrana, časy, vlastník, ...
- struktura souborů - sekvence bajtů / sekvence záznamů / strom
- typy souborů - běžné soubory, adresáře (systémové soubory vytvářející strukturu souborového systému), speciální soubory (znakové/blokové, soft linky)
- přístup
  - **sekvenční** – pohyb pouze vpřed, OS může přednačítat
  - **náhodný** – možno měnit aktuální pozici
  - **paměťově mapované soubory** – pojmenovaná virtuální paměť, práce se souborem instrukcemi pro práci s pamětí, ušetří se kopírování pamětí; mají i problémy (přesná velikost souboru, zvětšování souboru, velikost souborů)
- volné místo na disku - bitmapa / spojový seznam volných bloků

Uložení souborů
Soubory se ukládají na disk po blocích

- souvislá alokace - souvislý sled bloků
- spojovaná alokace - blok odkazuje na další
- indexová alokace - inode (UNIX)

Adresáře

- zvláštní typ souboru
- operace nad adresáři - hledání souboru / vypsání adresáře / přejmenování, vytvoření, smazání souboru
- kořen, aktuální adresář, absolutní/relativní cesta
- hierarchická struktura
  - *strom* – jednoznačné pojmenování (cesta)
  - *DAG* – víceznačné pojmenování, ale nejsou cykly
  - *obecný graf* – cykly vytváří problém při prohledávání
- implementace adresářů - záznamy pevné velikosti, spojový seznam, B-stromy

Co musí filesystém umět?
musí splňovat 3 věci: *správu souborů* (kde jsou, jak velké), *správu adresářů* (převod jméno ↔ id) (někdy to dělá jiný prostředek, dnes větš. umí FS sám), *správu volného místa*. někdy mohou být i další (odolnost proti výpadkům)

Velikost bloků – blok = nejmenší jednotka pro práci s diskem; disk pracuje s min. 1 sektorem (typicky 512 B) - někdy by pak bylo moc bloků → OS sdruží několik sektorů lineáně vedle sebe = 1 blok. velikost: velké = rychlejší práce, ale vnitřní fragmentace (průměrný soubor má cca pár KB), malé = malá vnitřní fragmentace, větší režie na info o volném místě/

umístění souboru (zabírá víc bloků!), navíc fragmentace souborů → zpomalení. dnes má blok cca 2-4KB.

Linky

- **Hard link** – Na jedna data souboru se odkazuje z různých položek v adresářích (na urovni FS)
- **Soft link** – Speciální soubor, který obsahuje jméno souboru (na urovni OS)

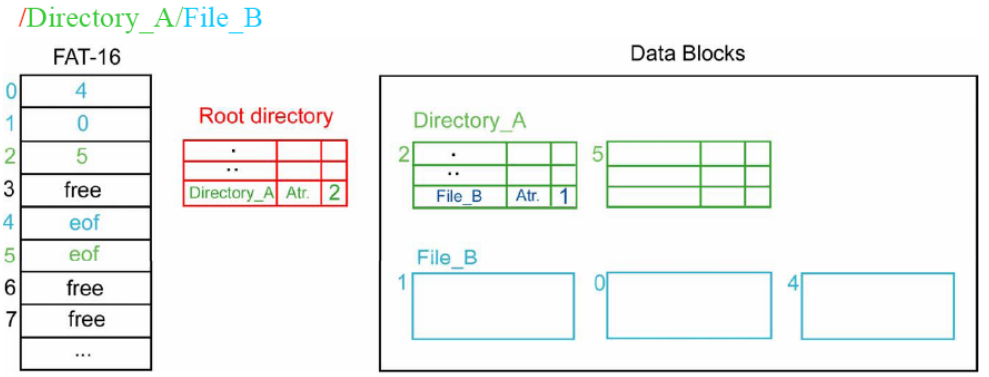
MBR

A master boot record (MBR) is a type of boot sector. It consists of a sequence of 512 bytes located at the first sector of a data storage device such as a hard disk. May be used for one or more of the following:

- Holding a partition table which describes the partitions of a storage device. In this context the boot sector may also be known as a partition sector.
- Bootstrapping an operating system. The BIOS built into a PC-compatible computer loads the MBR from the storage device and passes execution to machine code instructions at the beginning of the MBR.
- Uniquely identifying individual disk media, with a 32-bit disk signature, even though it may never be used by the operating system.

FAT

- System souboru FAT rozdeluje disk na dve vyznacne casti - samotnou tabulku FAT (ta je zpravidla ve dvou kopiich) a datovou oblast. Datova oblast je rozdelena na clustery (napriklad po 4096 bajtech) a FAT tabulka ma tolik polozek, kolik ma datova oblast clusteru (1:1).
- **FAT tabulka** – obsahuje čislo dalsiho clusteru v řadě + speciální záznamy (označení end of clusterchain, bad cluster, reserved cluster a free cluster)
- **Alokace souboru** – spojovaná, kždý blok ukazuje na další přes FAT tabulku



- Kdyz je nejaky cluster volny, v prislusne polozce FAT je 0. Tedy kdyz chci najit volne misto, tak staci najit libovolnou polozku FAT, ktera je 0, odpovidajici cluster pak je volny.

- Adresare i soubory jsou ulozeny stejne. Rozdily mezi adresari a soubory (krom daneho atributu) neexistuji. Z hlediska ulozeni na disku je adresar proste soubor, jehoz obsahem jsou adresarove polozky. Vyjimka viz root adresar.
- **Root adresář** – V root adresari, i ve vsech ostatnich adresarich, jsou proste jen za sebou ulozeny polozky. Kazda polozka obsahuje jmeno souboru nebo adresare, ke kteremu se vztahuje, atributy rozlisujici napriklad prave adresare od souboru, delku, prvni cluster, atd.

- Root directory ma pevnou velikost ve FAT 12/16

- Polozky root adresare, ktere nejsou pouzite, jsou oznacene jako prazdne (ale porad patri do root adresare, aby velikost zustala konstantni). Pridani pak pouzije nekterou prazdnou polozku. Pokud jsou jiz vsechny polozky plne, dalsi nelze pridat.

- **Vyhledávání souboru** – Postupne ... v root adresari se najde prvni podadresar cesty, v nem se najde druhy a tak dale. (tzn. lineární struktura)
- Adresarova struktura je ulozena prave v jednotlivych adresarich. Napriklad, pokud je na disku soubor .TXT, tak v korenovem adresari bude polozka FOO s atributem indikujicim, ze se jedna o adresar a s cislem prvniho clusteru adresare FOO, rekneme ze 123. V clusteru 123 pak budou polozky adresare FOO, mezi nimi bude podobne polozka BAR pro adresar BAR a cislem prvniho clusteru adresare BAR, rekneme 456. A v clusteru 456 budou polozky adresare BAR, mezi kterymi bude i polozka SOUBOR.TXT ...
- dnes se ni setkame ještě na flashkách a paměťových kartách
- **nevyhody FAT**:

  - velikost souboru max 4 GB

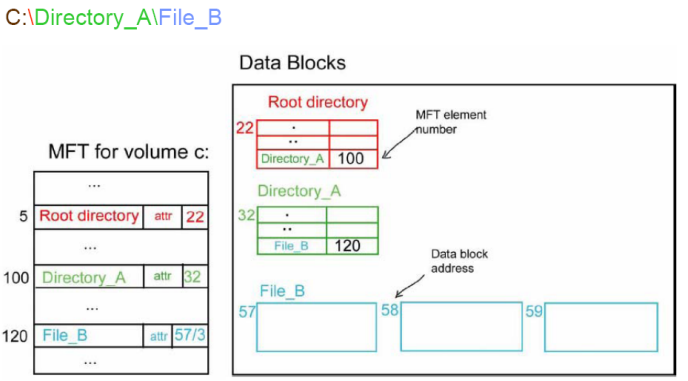
  - svazky - FAT32 reálně okolo 8TB (32kB clustery), nemá ale ochranu proti fragmentaci

  - práva - None of the various FAT-flavours has facilities for user-based access restrictions.
- https://d3s.mff.cuni.cz/pipermail/osy/2005-June/000167.html
- http://en.wikipedia.org/wiki/File_Allocation_Table

NTFS

- Zase rozděluje oddíl na dvě části tabulka MFT (jako soubor, ma ale přiřazenou prázdnou oblast aby se při růstu nefragmentovala) a datovou oblast.
- **Metafiles** - prvních cca 16 souboru jsou tzv. Metafiles, každý se stará o něco ⇒ flexibilita např. pri poskozenem povrchu

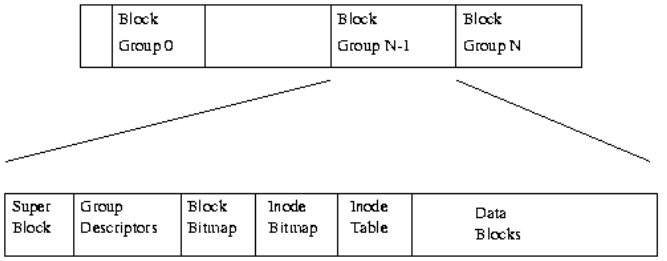
  příklady souboru: $MFT,$MFTmirr(MFT a zalozni kopie uprostred disku),$LogFile (zurnalovaci soubor),$. (root directory),$Bitmap (bitove pole volneho mista) atd...
- **Žurnál** – Operace s diskem se provadeji jako Transakce, takže např. pokud pri zapisu souboru FS zjisti ze cluster je fyz. vadný, celou trasakci rollbackuje a pusti ji jinde znovu.
- Další vlastnosti ktere ma navic: sifrovani, komprese, hardlinky (soubor je fyz. na disku jednou a ma vic zaznamu v MFT)
- **MFT tabulka** – obdoba FAT, vsechna metadata o souborech (jmeno,datum,prava) jsou ulozena v MFT - omoznuje pridavat ficury, muze obsahovat primo i male soubory nebo odkaz na clustery (všechny)
- Vnitřní struktura adresáře se realizuje B+Stromem (lexikograficky setrideny) zase pokud je malej zustane v MFT pokud je vetsi je v clusteru a v MFT je jenom kořenová část B+Stromu.



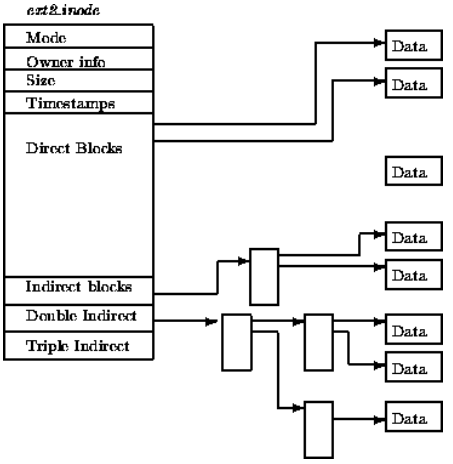
- http://pages.cs.wisc.edu/ bart/537/lecturenotes/s26.html
- http://www.digit-life.com/articles/ntfs/
- http://www.pcguide.com/ref/hdd/file/ntfs/archSector-c.html
- http://cs.wikipedia.org/wiki/NTFS
- http://ixbtlabs.com/articles/ntfs/

ext2

- Prostor je u ext2 rozdělen do bloků, ty jsou rozděleny do skupin (Block Groups) typicky jeden soubor se drží v jedné skupině (redukce fragmentace). Každá skupina obsahuje kopii superbloku (obsahuje kritické informace pro boot systemu) bitmapu bloků, inodes, Inode tabulku a nakonec datové bloky.



- **inode** – každý soubor nebo složka je reprezentována jako inode (v inode tabulce), obsahuje práva, velikost a hlavně pointery na datové bloky (12pointerů na přímý odkaz a další na stromovou strukturu), složky obsahují list inodes které obsahují



- Proc se porad pouziva - kvůli rychlosti
- Žurnálování zavedeno u ext3
- http://homepage.smc.edu/morgan_david/cs40/analyze-ext2.htm
- http://www.science.unitn.it/ fiorella/guidelinux/tlk/node95.html
- http://www.linux-security.cn/ebooks/ulk3-html/0596005652/understandlk-CHP-18.html

Plánování pohybu hlav disků

- FCFS (First-Come, First-Served) - žádné plánování, fronta požadavků, jeden za druhým
- SSTF (Shortest Seek Time First) - krajní žádosti mohou "hladovět"
- LOOK (výtah), C-LOOK (circular LOOK) - pohyb jen jedním směrem, na konci otočka

RAID (Redundant Array of Inexpensive Disks)

- JBOD (Just a Bunch of Disks)
- RAID 0 – striping, žádná redundance
- RAID 1 – mirroring, redundance
- RAID 0+1 – mirroring a striping
- RAID 2 – 7-bitový paritní Hammingův kód
- RAID 3 – 1 paritní disk, po bitech na disky
- RAID 4 – 1 paritní disk a striping
- RAID 5 – distribuovaná parita a striping
- RAID 6 – distribuovaná parita – dvojitá P+Q, striping

Report *(Galamboš)*
inode

Report *(Galamboš)*
Konkretni soubory system NTFS - tady jsem popletl jak vlastne pracuje FATka, o NTFS jsem mel tak jako povesechne informace, proste nic moc jsem nevedel, ale zase jsme si popovidali, byly mi vysvetleny me omyly a nakonec oka. Jinak je pravda, ze toho chce docela hodne a dopodrobna, ale kdyz clovek rekne, ze proste k tomuhle vic nesehnal a ze podrobnosti proste nevi, pobavi se o tom s Vami a vetsinou se to da dokupy.

Report *(Skopal)*
Zkoušel sám p. Skopal a jelikož jsem se FS učil jako jednu z prvních otázek, v návalu dalších informací jsem toho mezitím dost zapomněl. Popsal jsem jeden a půl strany obecnými vlastnostmi FS, co to je soubor, adresář atd. Popsal jsem FAT, NTFS a ext2, a to dost stručně. Obecné vlastnosti ho nezajímaly, hned přešel k FAT a jejím nevýhodám, otázky byly rychlé a dost podrobné, bylo vidět, že to se mnou nebude mít jednoduché. Pak měl otázky na NTFS, co má za spec. soubory, kdeže je v systému implementován B-strom atd. K ext2 jsme se nedostali až na jedinou zmínku, a to je alokace souborů (ve FAT lineární struktura, v ext2 strom). Když položil otázku na B-stromy v NTFS a po mojí těžce nejisté odpovědi prohlásil, že mu to stačí a odešel, bylo mi jasné, že tohle nedopadne dobře.

Report *(Yaghob)*

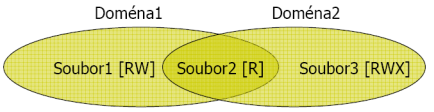
Konkretni Filesystemy - no jeste pred dvema dny by to byla moje smrt, nastesti sem se na to vcera zameril a nasel si opravdu presne jak funguje FAT, NTFS a ext2. Nakonec z toho byly tri papiry, az jsem na sebe byl pysny U FAT bylo treba napsat, jak se to alokuje, jak vypada ta tabulka, jakou roli ma kor. adresar, jak jsou tam ulozeny adresare, soubory apod. U NTFS jsem popsal vlastnosti ktere to ma navic, jako zurnal, sifrovani a pak rozepsal MFT docela podrobne, zase jak se resi soubory, adresare. Veci jako jak jsou tam ulozeny jednotlivy volume, MBR atd po me nastesti nechtel U ext2 je asi dulezite pochopit jak ten inode funguje, jak je to tam ve skupinach bloku, co je to superblok. Prislo mi, ze hlavni duraz je kladeny na to, at se vi, jak jsou tam ulozeny adresare a soubory. Pak prislo par otazek typu, jake jsou omezeni FATky, (velikost souboru, partitiony, ale hlavne se po me chtelo slyset: prava), kde se s tim dnes setkame (flashky, pametove karty) - na ty jsem dosel po vyjmenovani win98, mobilu, zkusil jsem i routery a ind. pocitace a nakonec me uspesne dovedl ke spravne odpovedi U ext2 prislo, proc se to porad jeste pouziva,kdyz je to tak stary FS - jedine co me napadlo byla rychlost - spravne

5.11 Bezpečnost, autentizace, autorizace, přístupová práva[3]

Definice

- **Ochrana** – s prostředky OS mohou pracovat pouze autorizované procesy
- **Autorizace** – zjištění oprávněnosti požadavku
- **Bezpečnost** – zabraňuje neautorizovaný přístup do systému
- **Přistupové právo** – povolení/zakázání vykonávat nějakou operaci
- **Doména ochrany** – uchovánání autorizací, množina párů (objekt:práva)

– **ACL (Access Control List)** – ke každému objektu seznam práv pro uživatele/skupiny
– **C-list (Capability List)** – ke každému uživateli/skupině seznam práv pro objekty
– **ACM (Access Control Matrix)** – řádky matice odpovídají uživatelům, sloupce objektům. V políčku daném řádkem a sloupcem je záznam o úrovni oprávnění odpovídajícího uživatele k příslušnému objektu. Přístupová matice je zpravidla velmi velká záležitost, zhusta řídká.



	Soubor1	Soubor2	Soubor3
Doména1	RW	R	
Doména2		R	RWX

Obrázek 1: Domény ochrany

Bezpečnost

Bezpečnostní modely obecně

První fází tvorby bezpečného IS je volba vhodného bezpečnostního modelu. Základní požadavky bezpečnosti: utajení, integrita, dostupnost, anonymita. Předpokládejme, že umíme rozhodnout, zda danému subjektu poskytnout přístup k požadovanému objektu. Modely poskytují pouze mechanismus pro rozhodování!

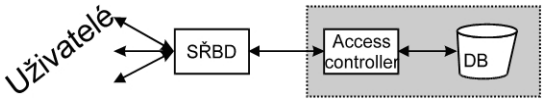
- **Jednoúrovňové modely** jsou vhodné pro případy, kdy stačí jednoduché ano/ne rozhodování, zda danému subjektu poskytnout přístup k požadovanému objektu a není nutné pracovat s klasifikací dat.
- **Víceúrovňové modely** - Může existovat několik stupňů senzitivity a "oprávněnosti". Tyto stupně senzitivity se dají použít k algoritmickému rozhodování o přístupu daného subjektu k cílovému objektu, ale také k řízení zacházení s objekty. Víceúrovňový systém "rozumí" senzitivitě dat a chápe, že s nimi musí zacházet v souladu s požadavky kladenými na daný stupeň senzitivity. Rozhodnutí o přístupu pak nezahrnuje pouze prověření žadatele, ale též klasifikaci prostředí, ze kterého je přístup požadován.

[3] z přednášek ZOS(Yaghob) a OI1(Beneš)

Bezpečnost fyzické přenosové vrstvy
Bezpečnost je do určité míry závislá na použitém přenosovém médiu. útok proti komunikačním linkám může být pasivní (pouze odposlech), nebo aktivní (vkládání dalších informací do komunikace).

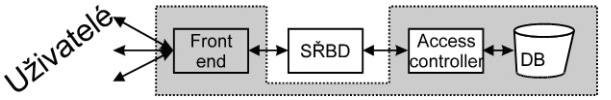
Víceúrovňová bezpečnost v DB[4]

- **partitioning** - Databáze je rozdělena dle stupně citlivosti informací na několik subdatabází, což vede ke zvýšení redundance s následnou ztíženou aktualizací a neřeší problém nutnosti současného přístupu k objektům s různým stupněm utajení.
- **šifrování** - Senzitivní data jsou chráněna šifrováním před náhodným vyzrazením. Zná-li útočník doménu daného atributu, může snadno provést chosen plaintext attack (utocnik muze ukladat plaintexty podle sveho vyberu a prohlizet si ciphertexty). Těžko totiž někoho zbavíme znalosti šifrovacího klíče. řešením je používat jiný klíč pro každý záznam, což je však poměrně náročné. V každém případě nutnost neustálého dešifrování snižuje výkon systému.
- **Integrity lock** - Každá položka v databázi se skládá ze tří částí: $< vlastnídata : klasifikace : checksum >$. Vlastní data jsou uložena v otevřené formě. Klasifikace musí být nepadělatelná, nepřenositelná a skrytá, tak aby útočník nemohl vytvořit, okopírovat ani zjistit klasifikaci daného objektu. Checksum zajišťuje svázání klasifikace s daty a integritu vlastních dat. Model byl navržen jako doplněk (access controller) komerčního SŘBD, který měl zajistit bezpečnost celého systému. šedá oblast na obrázku vyznačuje bezpečnostní perimetr systému. Access controller nevidí na výstup databáze a není schopen zajistit, na výstupu označovala data stupněm senzitivity.



Obrázek 2: Access Controller

- **Spolehlivý front-end (guard)** - Systém je opět zamýšlen jako doplněk komerčních SŘBD, které nemají implementovánu bezpečnost. Uživatel se autentizuje spolehlivému front-endu, který od něho přebírá dotazy, provádí kontrolu autorizace uživatele pro požadovaná data, předává dotazy k vyřízení SŘBD a na závěr provádí testy integrity a klasifikace výsledků před předáním uživateli. SŘBD přistupuje k datům prostřednictvím spolehlivého access controlleru.



Obrázek 3: Spolehlivý front-end

- **Commutative Filter** – Jde o proces, který přebírá úlohu rozhraní mezi uživatelem a SŘBD. Filtr přijímá uživatelovy dotazy, provádí jejich přeformulování a upravené dotazy posílá SŘBD k vyřízení. Z výsledků, které SŘBD vrátí, odstraní data, ke kterým uživatel nemá přístupová práva a takto upravené

[4]zdroj: vypracované otázky na zkoušku z OI1

výsledky předává uživateli. Filtr je možno použít k ochraně na úrovni záznamů, atributů a jednotlivých položek. V rámci přeformulování dotazu může např. vkládat další podmínky do dotazu, které zajistí, že výsledek dotazu závisí jen na informacích, ke kterým má uživatel přístup.

- **View** - Pohled je část databáze, obsahující pouze data, ke kterým má daný uživatel přístup. Pohled může obsahovat i záznamy nebo atributy, které se v původní databázi nevyskytují a vznikly nějakou funkcí z informací původní databáze. Pohled je generován dynamicky, promítají se tedy do něho změny původní DB. Uživatel klade dotazy pouze proti svému pohledu - nemůže dojít ke kompromitaci informací, ke kterým nemá přístup. Záznam / atribut původní databáze je součástí pohledu, pokud alespoň jedna položka z tohoto záznamu / atributu je pro uživatele viditelná, ostatní položky v tomto jsou označeny za nedefinované. Uživatel při formulování dotazu může používat pouze omezenou sadu povolených funkcí. Tato metoda je již návrhem směřujícím k vytvoření bezpečného SŘBD.

Autentizace

Identifikace něčím, co uživatel ví, má nebo je.

- **Hesla**

– slovníkový útok (80–90% hesel je jednoduchých), hrubá síla
– vynucování délky a složitosti hesla

- **Model otázka/odpověď** (challenge-response) – např. autentizace počítačů. Počítač, který se chce autentizovat obdrží náhodný dotaz, který zpracuje (např. zašifruje tajným klíčem) a odešle výsledek. Výsledek je ověřen a pokud je správný, autentizace je uskutečněna.
- **Fyzický objekt** – smartcards, USB klíče (certifikáty)
- **Biometrika** – otisky prstů, rohovka, hlas

Autentizace v prostředí databáze
Každý, komu je povolen přístup k databázi, musí být pozitivně identifikován. databáze potřebuje přesně vědět, komu odpovídá. Protože však zpravidla běží jako uživatelský proces, nemá spolehlivé spojení s jádrem OS a tedy musí provádět vlastní autentizaci.

Autentizace v síti
Protože síťové prostředí zpravidla není považováno za bezpečné, je třeba využívat autentizační mechanismy odolné vůči odposlechu, resp. aktivním utokům. často bývá žádoucí řešit jednotné přihlášení (single sign on). S procesem integrace autentizačních mechanismů souvisí nutnost zavedení centrální správy uživatelů nebo alespoň synchronizace záznamů o uživatelích.

Autorizace

Existují různé úrovně ochrany objektu:

1. **Žádná ochrana** - Je nutna alespon samovolná casová separace procesů.
2. **Izolace** - Procesy o sobe vůbec neví a systém zajisťuje ukrytí objektů před ostatními procesy.

3. **Sdílení vseho nebo niceho** - Vlastník objektu deklaruje, zda je objekt public nebo private (tedy jen pro neho).

4. **Sdílení s omezenými prístupy** - OS testuje oprávněnost každého prístupu k objektu. U subjektu i objektu existuje záznam, zda má subjekt právo prístupu k objektu.

5. **Sdílení podle způsobilosti** - rozšíření předchozího - Oprávnení dynamicky závisí na aktuálním kontextu.

6. **Limitované pouzití objektů** - Krome oprávnení prístupu specifikujeme, jaké operace smí subjekt s objektem provádet.

TODO: přístupová práva ??? Ochrana informace I.

5.14 Spojované a nespojované služby, spolehlivost, zabezpečení protokolu

Spojované a nespojované služby

Spojovaná obě strany musí navázat spojení, které je pak potřeba ukončit. Jde o stavovou komunikaci a přechody musí být koordinované. Pro spojení je nalezena jedna cesta po které komunikace probíhá a mohou ji být vyhrazeny zdroje. Potřeba ošetřit nestandartní situace (výpadek spojení). Zachovává pořadí – díky jedné cestě. Každé spojené má své ID.

Nespojovaná Komunikující strany o sobě neví. Komunikuje se pomocí zasílání zpráv – datagramů a cesta je hledána pro každý datagram znovu, tj není žádná pevně vytyčena. Každý datagram musí nést plnou adresu příjemce. Je bezstavová. Neřeší se nestandartní situace a pokračuje se v přenosu. Nezachovává pořadí, přenos jednotlivých bloků je vzájemně nezávislý, každý jde jinou cestou.

Spolehlivost

Přenosy nejsou ideální, může dojít k poškození, kdo se pak má starat o nápravu?

Spolehlivá přenosová služba O napravení se stará ten kdo data přenáší, musí rozpoznat chybu a vyžádat si nový přenos (opravu)

Nespolehlivá přenosová služba Kdo přenáší se o data nestará a chybná prostě zahodí a přenáší dál. Zajistění spolehlivosti vyžaduje režiji a ruší pravidelnost doručování dat, také není nikdy absolutní. Někomu vadí víc horší pravidelnost než chyba v datech (obraz a zvuk).

Jaké jsou možnosti zajištění spolehlivosti (v závislosti na dostupnosti zpětné vazby)?
Moze byt realizovane na ktorejkolvek vrstve okrem fyzickej. Princip a sposob realizacie je v zasade rovnaky na vsetkych vrstvach.
Podmienkou je rozpoznat, ze doslo k nejakej chybe pri prenose.
Pri nespolahlivom prenose, sa neda robit nic.

Pri spolahlivom sa postarat o napravu. **Moznosti:**

- pouzitie samoopravnych kodov, napri Hammingove kody, problemom je velka redundacia, ktora zvysuje objem prenasanych dat, pouziva sa vynimocne
- pomocou potvrdzovania: prijemca si necha znovu zaslat poskodene data, podmienkou je existenica spatnej vazby (aspon polovicny duplex, aby prijemca mohol kontaktovat odosielatela).

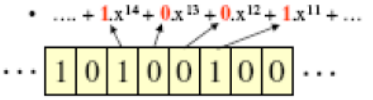
Jak se používá parita a kontrolní součet pro detekci chyb při přenosech?
Parita:

- **Paritny bit:** bit pridany naviac k datovym bitom. Suda parita (paritny bit je nastaveny tak, aby celkovy pocet 1 bol sudy), licha parita (1 lichy), jednickova parita (paritny bit pevne nastaveny na 1, neni zabezpecovaci efekt), nulova parita (nastaveny na 0).
- **Priecna parita:** po jednotlivych bytoch/slovach, informacie o tom, ktory byte je poskodeny je nadbytocna, aj tak sa posiela rovno cely blok (ramec, pakec).

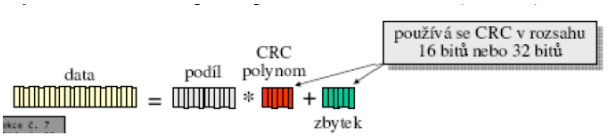
• **Podelna parita:** parita zo vsetkych rovnolahlych bitov vsetkych bytov/slov

Kontrolny sucet: jednotlive byty/slova/dvojslova tvoriace prenasany blok sa interpretuju ako cisla a scitaju sa, vysledny sucet sa pouzije ako zabezpecovaci udaj (obvykle sa pouzije iba cast suctu, napr nizsi byte alebo nizsie slovo.
Alternativa: miesto suctu sa pocita XOR jednotlivych bitov.
Je ucinnejsi ako parita, ale stale je miera zabezpecenia nizka.

Jak se používá CRC pro detekci chyb při přenosech?
CRC = Cyclic Redundancy Check
Postupnost bitu, tvoriaca blok dat, je interpretovana ako polynom, polynom nad telesom charakteristiky 2, kde bity su jeho koeficienty.



Tento polynom je vydeleny inym polynomom, vysledkom je podiel a zvysok, v roli zabezpecenia sa pouzije zvysok po deleni charakteristickym polynomom.
Schopnosti detekcie su vynikajuce: vsetky shluky chyb s lichym poctom bitov, vsetky shluky chyb do velkosti n bitu, kde n je stupen charakteristickeho polynomu, vsetky shluky chyb velkosit ¿ n+1 s pravdepodobnostou 99,999



Spolahlivost CRC kodov sa opiera o vysledky z algebry, samotny vypocet je velmi jednoduchy a moze byt lahko implementovany v HW, pomocou XOR hradiel a posuvnych registrov.

Jaký je princip potvrzování? Jak funguje jednotlivé a kontinuální potvrzování?
Ide o obecnejsi mechanizmus, ktory sluzi viac ucelom sucasne: 1. zaistenie spolehlivosti
(umoznuje, aby si prijemca vyziadal opakovane zaseialanie poskodeneho ramca), 2. riadenie toku (aby prijemca mohol regulovat tempo, akym odosielatel posiela data.

Sposoby:
• kladne a zaporne potvrdzovanie: potvrdzuju sa spravy, resp chybne prijate bloky
• jednotlive a kontinualne potvrdzovania: podla toho, ci odosielatel vzdy caka na potvrdenie alebo odosiela do fronty.
• Samostatne a nesamostatne potvrdzovanie: ci potvrdenie cestuje ako samostatny ramec/paket, alebo je vnorene do datoveho paketu.
• Metoda okienka

Jednotlive potvrdzovanie: StopWait ARQ

• ide o samostatne jednotlive potvrdzovanie, potvrdenie je prenasane ako samostatny (riadiaci) blok, potvrdzovany je kazdy jeden paket (kladne, zaporne, timeout)
• pribeh: odosielatel odosle datovy ramec a caka na jeho potvrdenie (kladne, zaporne), dalsi ramec neodosiela, prijemca odosle potvrdenie, podla druhu potvrdenia odosielatel bud odosle dalsi ramec alebo opakuje prenos, timeout interpretuje ako zapornu odpoved
• jednoducha a priamociara interpretacia, charatker prenosu cisto poloduplexny, napr. protokoly IPX/SPX
• ma zmysel v LAN (kratka odozva), nie WAN (zpozdeni velke)

Kontinualne potvrdzovanie: continuous ARQ
• **idea:** odosielatel bude vysielat datove ramce dopredu, a prislusne potvrdenia prijimat priebezne, s urcitym zpozdenim
• ak dostane zapornu odpoved: **1. selektivne opakovanie**: odosle iba ramec, ktory sa poskodil (prijimatel musi ukladat do bufferu, narocne hospodarenie s pamatou), **2. navrat spat**: riesi

Jaký je rozdíl mezi samostatným a nesamostatným potvrzováním? Jak funguje piggybacking?
Samostatne: potvrdenie je prenasane ako samostatny ramec specialneho typu, spojene s relativne vysokou reziou, samostatne potvrdenie je male, obale je velky
Nesamostatne: potvrdenie je zasielane ako sucast datovych ramcov, prenasanych v opacnom smere, ktore su potvrdzovane, tzv. **piggybacking**

Zabezpečení protokolu

Jaká je podstata sítí VPN (Virtual Private Network)?
Samostatná podsíť jiné sítě (věřejné datové sítě). Z pohledu uživatele jde o samostatnou síť. Uživatel chce mít vlastní síť ale nevyplatí se mu jí vybudovat. Samostatný adresový prostor, přístup k uzlům mimo VPN je jen přes bránu.

Ekonomický efekt: lacinejší
Praktičnost: jednoduchá údržba a efekt vlastní sítě.
Bezpečnost: poskytují určitou ochranu
Spojení poboček firmy přes internet do VPN, takovéto pobočkové sítě pak splývají do jednoho logického celku. Připojení vzdáleného uživatele do firemní sítě.

Jaké bezpečností funkce jsou schopny zabezpečit sítě VPN?
Funkce a služby: identifikace a autentizace uživatele. Takovýto uživatel se pak může volně pohybovat po VPN. Zajištění dúvěrnisti – šifrování a zajištění integrity – nelze komunikaci neoprávněně pozměnit.

Jak je v TCP/IP řešena bezpečnost (a zabezpečení)?
Zabezpečení si musí každá aplikace zajistit sama (na aplikační úrovni). Přenos. Infrastruktura je tak jednoduší rychlejší a lacinější.

V ISO/OSI to řeší relační vrstva.

Report *(Galambos)*
Chtel vedet, jak vypada navazovani spojeni, v jake vrstve se resi zabezpeceni a takovy ruzny nesmysly, nakonec to byla pry horsi dvojka.

Report *(Peterka)*
Zajímal ho především způsob jakým lze spolehlivosti dosáhnout, konkrétní metody - tedy samoopravné kódy, CRC a kontrolní součty - a jejich použití v konkrétních případech. Dále se také ptal na rozdíl mezi jednotlivým a kontinuálním potvrzováním - v jakých situacích lze resp. nelze použít - a piggybacking.

V pohode, udelala se z toho takova lehka prochazka po lehce pribuznych tematech a dostali jsme se i k tomu, kdo muze za fragmentaci