Materiály na SZZ pro Informační Bezpečnost

Matěj Douša

Červen 2024

Obsah

1 Počítače a systémy

1.1 SP-19 (PA1)

Datové typy v programovacích jazycích. Staticky a dynamicky alokované proměnné, spojové seznamy. Modulární programování, procedury a funkce, vstupní a výstupní parametry. Překladač, linker, debugger.

Datové typy

V programovacích jazycích používáme proměnné, tedy něco, co uchovává datovou hodnotu s nějakou vnitřní strukturou. Proměnné jsou identifikovány svými jmény — identifikátory. Datový typ proměnné definuje vnitřní strukturu/reprezentaci dat a jejich význam. Tím určuje jakých hodnot může proměnná nabývat a také jaké operace lze s proměnnou (její hodnotou) vykonávat.

Jednoduché datové typy:

- celočíselné
  - existují různé délky — short, int, long (byte, long long, ...)
  - signed (znaménkové) — umí uložit i záporné hodnoty, používá se doplňkový kód
  - unsigned (neznaménkové) — ukládá jen kladné hodnoty, přímý kód
- s pohyblivou řádovou čárkou
  - existují různé délky — float, double, long double
  - znaménko (1 bit) + mantisa (velikost=přesnost) + exponent (velikost=rozsah)
- znakové

  Znaky jsou kódovány jako čísla, používá se ASCII / extended ASCII / UNICODE.
- logická hodnota

  Není v C, ale často se v jazycích vyskytuje (boolean — true/false).

Další datové typy:

- ukazatel (pointer)

  Adresy paměti, kde je uložen datový typ pointeru (pointer vždy ukazuje na konkrétní typ/funkci, případně void).
- výčtový typ (enum)
- struktura

  Je složena z dalších datových typů, klidně dalších struktur.
- union

  Ukládá více různých datových typů na stejné místo.
- třída (ve vyšších jazycích)

Statická a dynamická alokace

Staticky alokované proměnné:

- vzniknou běžnou deklarací
- ukládají se na zásobník (lokální proměnné) či do části .BSS (neinicializované globální proměnné) a .DATA (inicializované globální proměnné)
- v případě pole je nutno znát v době kompilace velikost (statická velikost)

Dynamicky alokované proměnné

- vzniknou použitím speciální funkcí/operátorem
- ukládají se na haldě (heap)
- přistupujeme přes pointer
- je možné alokovat paměť podle hodnot spočítaných za běhu programu

Spojové seznamy

- oproti poli nejsou položky seřazeny v paměti, ale každý prvek seznamu obsahuje ukazatel na další prvek.
- podobně jako v dynamicky alokovaném poli lze ukládat předem neznámý objem dat
- nelze jednoduše indexovat, ale lze libovolně přidávat či ubírat prvky z jakékoliv pozice v seznamu

Modulární programování

- složitější programy mohou být rozděleny do modulů
- tyto moduly lze použít v různých dalších částech programu
- modul má svou specifikační část (deklarace poskytovaných prostředků/rozhraní) a implementační část (definice/implementace poskytovaných prostředků)
- v C/C++ typicky hlavičkový soubor (.h/.hpp) a implementační soubor (.c/.cpp)

Procedury, funkce a parametry

- procedura/funkce je posloupnost příkazů uložených v paměti programu
  - procedura — bez návratové hodnoty (typ void)
  - funkce — s návratovou hodnotou
- použijeme ji zavoláním přes její jméno
- deklarace je specifikace jejího rozhraní — parametrů a typu návratové hodnoty
- definice je samotný kód funkce
- vstupní paramtery jsou informace, které využije kód funkce
- výstupní parametry jsou výsledkem běhu funkce — typicky se nějak změní a tím nám dají výsledek

Překladač

- překládá vyšší programovací jazyky do nižších
- ze zdrojového kódu vzniká objektový soubor — modul se strojovým kódem
- front-end přeloží konkrétní jazyk do vnitřní reprezentace (abstrakce nezávislá ani na platformě ani na jazyku)
- back-end přeloží vnitřní reprezentaci do strojového kódu konkrétní platformy

Linker

- spojuje přeložené moduly do výsledného celku — programu
- výstupem je spustitelný soubor

Debugger

- usnadňuje hledání chyb v kódu, také usnadňuje pochopení programu
- je vhodné kompilovat s informacemi pro ladění
- je možné si na nějakém místě běh programu zastavit a např. sledovat obsah proměnných, pouštět každý krok programu postupně...

1.2 SP-30 (SAP)

Kódy pro zobrazení čísel se znaménkem a realizace aritmetických operací (paralelní sčítačka/odčítačka, realizace aritmetických posuvů, dekodér, multiplexor, čítač). Reprezentace čísel v pohyblivé řádové čárce.

Čísla se znaménkem

Existuje několik možností, jak v počítači ukládat celá čísla:

- Přímý kód
  - první bit je znaménkový — určuje tedy, zda je hodnota za ním kladná či záporná
  - ostatní bity představují absolutní hodnotu čísla
  - existuj zde kladná i záporná nula
- Doplňkový kód
  - dle prvního bitu lze poznat znaménko čísla
  - převod kladné $\leftrightarrow$ záporné lze vysvětlit jako inverze bitů a následné přičtení jedničky
  - 0111 (7) $\rightarrow$ 1001 (-7)
  - není zde záporná nula
- Aditivní kód
  - uložené číslo je posunuto o nějakou konstantu, typicky polovina rozsahu
  - pro 4 bity určeme nulu jako 1000 — pak 1111 je 7, 0000 je -8
  - nula není zobrazena jako nula
  - není zde záporná nula

Čísla v pohyblivé řádové čárce

Reprezentace pohyblivé řádové čárky vychází ze zobrazení $A = M * z^e$ používaném např. ve fyzice, kde z je základ soustavy (zde 2), e je exponent jako celé číslo, M je mantisa.

- používá se normalizovaný tvar, tedy mantisa je zapsána tak, že ji nelze "posunout"více doleva.
- v přímém kódu mantisy je vlevo vždy jednička, která se skrývá (zvýšení přesnosti)
- pro mantisu se typicky používá přímý kód, pro exponent aditivní
- float (32b) typicky vypadá jako 1b znaménko, pak 8b exponent a nakonec 23b mantisa (tedy přesnost 24b)

Realizace aritmetických operací

- Paralelní sčítačka

  Tvořena více jednobitovými sčítačkami. Jednobitová sčítačka má 3 vstupy: A, B (sčítané bity) a vstupní přenos (carry — např. ze sčítačky nižšího řádu). Výstupy jsou S (výsledek) a výstupní přenos.
- Aritmetické posuvy

  Posun čísla vlevo/vpravo. Realizuje posuvný registr. Existuje více různých posuvů:
  - logický posuv — doplňuje nuly
  - cyklický posuv — doplňuje co vylezlo na druhé straně
  - aritmetický posuv — doplňuje 1 nebo 0 podle znaménka čísla
- Dekodér

  Kombinační logický obvod, který má méně bitů na vstupu než na výstupu, a podle tabulky převádí. Kodér má opačnou funkci.
- Multiplexor

  Na základě řídícího signálu vybere, který ze vstupů pošle na výstup (má několik vstupů + řídící vstup, a jeden výstup).
- Čítač

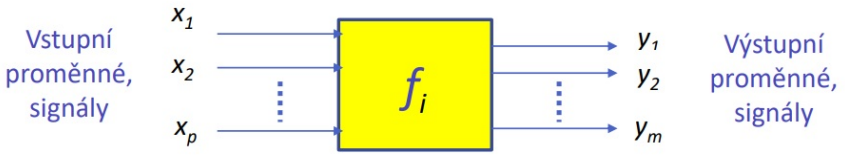
  Registr s funkcí inkrementu/dekrementu, může čítat nahoru a/nebo dolů. Existují úplné čítače (do mocnin 2) či neúplné (do jiných čísel). Typicky čítají v binárním kódu, lze i např. v Grayově kódu.

1.3 SP-28 (SAP)

Kombinační a sekvenční logické obvody (Mealy, Moore), popis a možnosti implementace na úrovni hradel. Minimalizace vyjádření logické funkce s využitím map.

Kombinační obvody

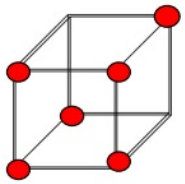
- popsány kombinační funkcí
- hodnoty všech výstupů (výstupních proměnných) jsou v každém časovém okamžiku určeny pouze vstupem (hodnotami vstupních proměnných) ve stejném okamžiku
- mohou být popsány např. Booleovskou (logickou) formulí
  Příklad: $f = x_1.\overline{x_2} + \overline{x_1}.x_2 = (x_1 + x_2).(\overline{x_1} + \overline{x_2})$
- obecně kombinační obvod vypadá následovně:



  Logická funkce $y_k = f(x_1, x_2, x_3, ..., x_p)$ existuje pro každý výstup y.
- možnosti reprezentace logických funkcí:
  - tabulka

ab	f
00	0
01	1
10	1
11	0

  - n-rozměrná krychle



  - Booleovský výraz
    Viz výše
  - mapa (Karnaughova)

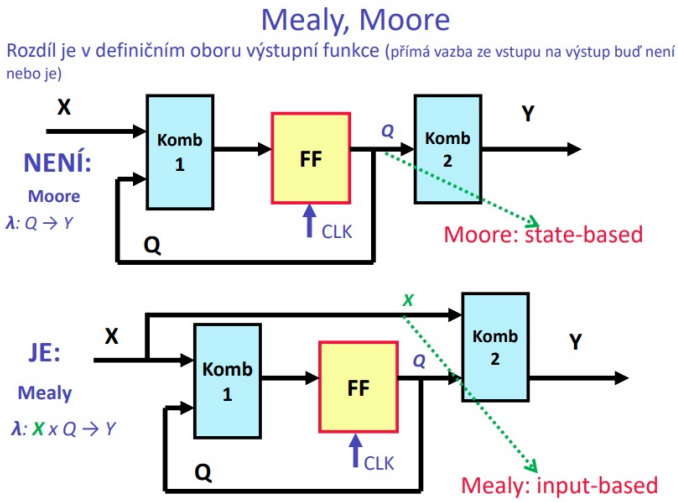
	a		b	
		X	1	
c		1	1	X

- možnosti realizace obvodů:
  - na úrovni hradel
  - mapování na technologii (FPGA, ASIC)
  - popis v jazyku (VHDL, Verilog)

Sekvenční obvody

- výstup závisí na posloupnosti/sekvenci hodnot na vstupu
- zapamatování se realizuje zpětnou vazbou
- popsány konečným stavovým automatem

- typy sekvenčních obvodů:
  - Moore
    Obvod, jehož výstup závisí pouze na vnitřním stavu.
  - Mealy
    Obvod, jehož výstup závisí také na aktuálním vstupu (kromě stavu).

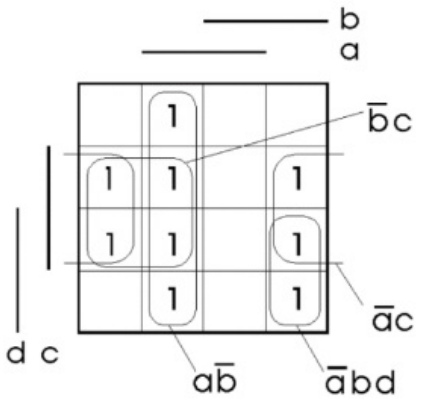


Implementace na úrovni hradel

- nejprve minimalizace logické funkce a zápis výsledku např. v Booleovském výrazu
- následně nakreslení/vytvoření funkce pomocí základních hradel — NOT, AND, OR, NAND, NOR, XOR

Minimalizace logické funkce

- smysl — zjednodušit a zkrátit zápis, snížit potřebný materiál pro výrobu
- minimalizace pomocí map — založenana hledání co největších skupin sousedních stavů.
- postup při minimalizaci:
  - vytvoření Karnaughovy mapy pro funkci
  - nalezení všech přímých implikantů (maximální skupiny jedniček či "dont care")
  - určení všech podstatných implikantů (obsahující jedničku, kterou jiný implikant neobsahuje)
  - pokud nejsou pokryty všechny vrcholy s "1", nutno vybrat další přímé implikanty (takové, kde je nejméně negací)



$a\bar{b} + \bar{b}c + \bar{a}c + \bar{a}bd$

- lze také kroužkovat nuly — pak je ale nutné funkci sestavit jinak.
  $(\bar{a} + \bar{b})(a + c + d)(a + b + c)$

1.4 SP-29 (SAP)

Architektura číslicového počítače, instrukční cyklus počítače, základní třídy souborů instrukcí (ISA). Paměťový subsystém počítače, paměťová hierarchie, skrytá paměť (cache).

Architektura číslicového počítače

- architektura se zabývá strukturou a chováním počítače
- řeší specifikaci různých funkčních modulů jako procesor a paměť a nebo např. instrukční sadu
- podsměry:
  - ISA — Instruction Set Architecture — architektura souboru instrukcí
  - Mikroarchitektura — konkrétní sestavení a složení procesoru
  - Systémový design — řeší další HW komponenty
- každý počítač je složen z následujících částí:
  - datová část procesoru — ALU, registry
  - řadič — řídící jednotka procesoru
  - paměťový subsystém
  - vstupní zařízení
  - výstupní zařízení
- typy architektury:
  - Von Neumannova architektura
    Data i instrukce jsou uložena spolu, nejsou explicitně označena/ny.
  - Harvardská architektura
    Data a instrukce jsou rozdělené.

Instrukční cyklus počítače

- čtení instrukce (IF — Instruction Fetch)
- dekódování instrukce (ID — Instruction Decode)
- načtení operandů (OF — Operand Fetch)
- provedení instrukce (IE — Instruction Execution)
- zapsání/uložení výsledku (WB — Write Back / Result Store)
- přerušení?

Co je instrukce? Obsahuje informace:

- co se má provést
- s čím se to má provést (operandy)
- kam se má uložit výsledek
- kde se má pokračovat

Tyto informace mohou být zadány explicitně, nebo mohou být dány typem instrukce, tedy architekturou počítače — tedy implicitně.

ISA — Architektura souboru instrukcí Co je potřeba určit:

- typy a formáty instrukcí, instrukční soubor
- datové typy, kódování a reprezentace, způsob uložení dat v paměti
- módy adresování paměti a přístup do paměti dat a instrukcí
- mimořádné stavy

Výhody:

- abstrakce — možnost různě implementovat stejnou architekturu instrukcí
- definice rozhraní mezi nízkoúrovňovým AW a HW
- standardizuje instrukce, bitové vzory strojového jazyka

Třídy souborů instrukcí (ISA)

- Střadačově (akumulátorově) orientovaná ISA

  Akumulátor je registr pro mezivýpočty, používá se implicitně jako zdroj pro výpočty i jako cíl pro výsledky. Používají se instrukce s jedním operandem. Nejstarší ISA (1949-60) — vyvinula se z kalkulaček.

  Výhody:

  - jednoduchý HW
  - minimální vnitřní stav procesoru — rychlé přepínání kontextu
  - krátké instrukce
  - jednoduché dekódování instrukcí

  Nevýhody:

  - častá komunikace s pamětí
  - omezený paralelismus mezi instrukcemi

  Populární v 50. — 70. letech, HW byl drahý, paměť byla rychlejší než CPU.

- Zásobníkově orientovaná ISA

  Pracovní registry jsou uspořádány do struktury zásobníku. Přistupuje se k vrcholu tohoto zásobníku. Využití pro vyhodnocení výrazů a vnořená volání podprogramů. Většina instrukcí nemá operand (použije se implicitně např. vrchní 2 registry zásobníku).

  Výhody:

  - jednoduchá a efektivní adresace operandů
  - krátké instrukce
  - krátké programy
  - jednoduché dekódování instrukcí
  - snadno lze napsat neoptimalizující překladač

  Nevýhody:

  - nelze náhodně přistupovat k lokálním datům
  - omezený paralelismus — zásobník je sekvenční
  - přístupy do paměti je těžké minimalizovat

- ISA orientovaná na registry pro všeobecné použití

  Dnes převládá. GPR — General Purpose Registers. Typicky 2 nebo 3 operandy.

  Výhody:

  - registry (a cache) jsou rychlejší než paměť
  - k registrům lze přistupovat náhodně
  - registry mohou obsahovat mezivýsledky a lokální proměnné
  - méně častý přístup do paměti

  Nevýhody:

  - složitější překladač (optimalizace pro použití registrů)
  - přepnutí kontextu trvá déle

Paměťový subsystém počítače

- cache (skrytá paměť) — rychlá, drahá, umístěna blíž k procesoru
- hlavní paměť — pomalejší, levnější, větší
- vnější paměť — pomalá, velká
- záložní paměť (CD, DVD, flash, magnetické pásky)
- RAM — random access memory (přístup adresou)
- CAM — content adressable memory (přístup klíčem)

Paměťová hierarchie

- registry
- L1 cache (SRAM)
- L2 cache (SRAM)
- L3 cache (SRAM)
- hlavní paměť (DRAM)
- HDD, SSD
- Mass storage (optical disks, tapes)
- Remote storage (cloud)

Cache

Kopie často používaných dat z hlavní paměti

- časová lokalita
  Data, ke kterým bylo právě přistupováno, budou pravděpodobně brzy potřeba znovu.
- prostorová lokalita
  Po přístupu k nějakým datům se pravděpodobně budou používat i vedlejší data.

1.5 OB-4 (APS)

Instrukční cyklus počítače a zřetězené zpracování instrukcí. Mikroarchitektura skalárního procesoru se zřetězeným zpracováním instrukcí, datové a řídicí hazardy při zřetězeném zpracování instrukcí a způsoby jejich ošetření.

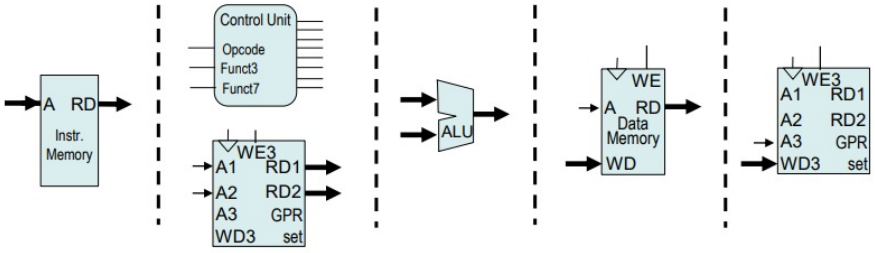
Definice 1: ISA (Instruction Set Architecture) je abstraktní rozhraní mezi HW a nízkoúrovňovým SW, které zahrnuje vše nezbytné pro psaní korektních programů ve strojovém jazyce. Zahrnuje instrukční sadu, registry, organizaci paměti, vstupy a výstupy,...

Definice 2: ISA je kompletní instrukční sada procesoru, včetně adresních módů.

Mikroarchitektura: Mikroarchitektura je detailní interní organizace procesoru, včetně hlavních funkčních jednotek, jejich propojení a řízení.

Instrukční cyklus počítače

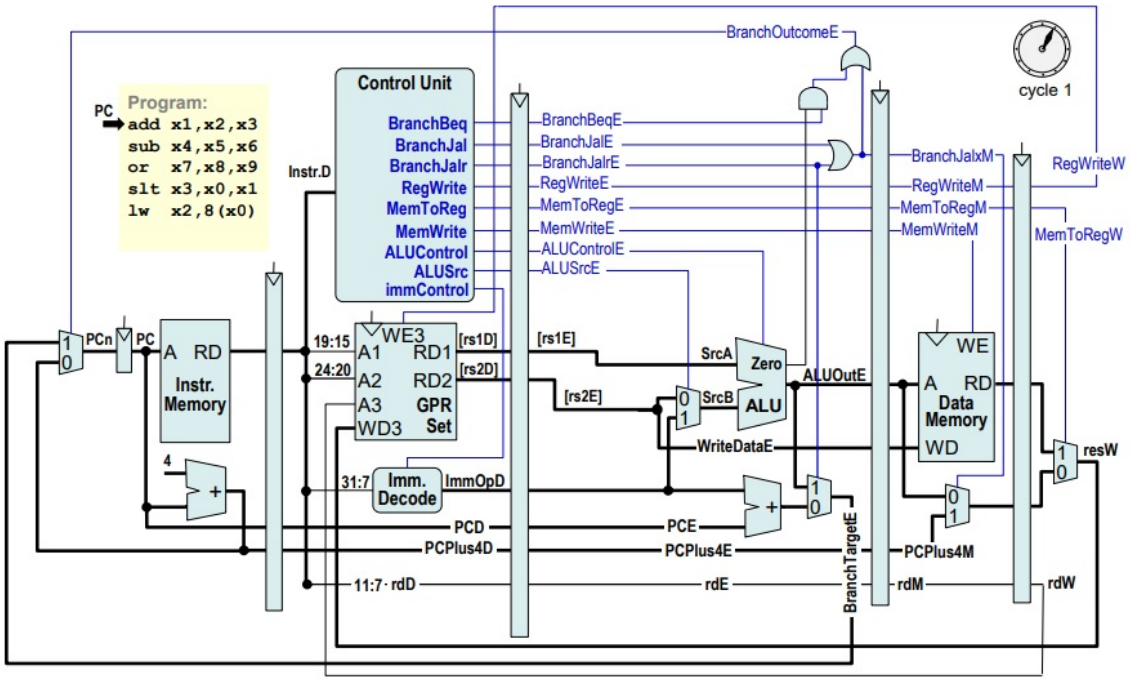
- IF — Instruction Fetch
- ID/OF — Instruction Decode and Operand Fetch
- EX — Execute
- MEM — Memory access
- WB — Write Back



Zřetězené zpracování instrukcí

Instrukce se dle svých fází rozdělí a v procesoru vykonává postupně. Procesor je rozdělen dělícími registry. Instrukce se zpracovává postupně v rozdělených částech procesoru, vykonává se zároveň více instrukcí naráz (v různých fázích).

Mikroarchitektura skalárního procesoru se zřetězeným zpracováním instrukcí



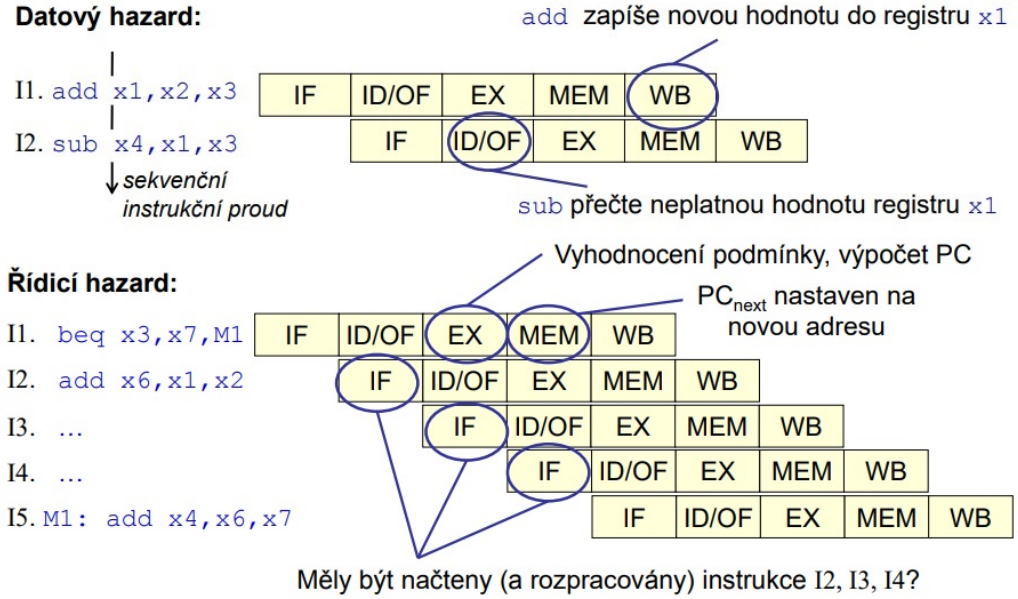
Hazardy

Protože je rozpracováno více instrukcí najednou, mohou vznikat konflikty při přístupu ke sdíleným prostředkům počítače. Tomu se říká hazardy. Sdíleným prostředkem je prostředek, který je opakovaně použit v různých stupních instrukčního zřetězení.

Typy hazardů:

- datové (důsledek datových závislostí: RAW, WAR, WAW)
- řídící (instrukce měnící PC, tedy obsah fronty instrukcí)
- strukturální (počet současných požadavků na daný prostředek převyšuje počet jeho instancí)

Hazardy mohou způsobovat pozastavení instrukčního zřetězení (stall) nebo vyprázdnění (flush).



Možnosti řešení hazardů:

- přeposílání (forwarding)

  Lze použít v případě datového hazardu, kdy výsledek předchozí instrukce požadovaný instrukcí následující vznikne dříve nebo ve stejném cyklu, kdy má být následující instrukcí použit. Výsledek je přeposlán tam, kde je potřeba (část EX).

- pozastavení (stall)

  Lze použít v případě datového hazardu, kdy je výsledek předchozí instrukce potřeba před jeho vznikem. Zpracovávání následujících instrukcí se pozastaví a vyplní se instrukce nop (bublina). V dalších instrukcích se pokračuje, když výsledek existuje — lze jej tedy přeposlat.

  Také může řešit strukturální hazardy.

- vyprázdnění části pipeline (flush)

  Lze použít v řídících hazardech, kdy se PC nastaví na neočekávanou adresu nějakou skokovou instrukcí. Všechny již načtené a zpracovávané instrukce, které následují po skoku, se musí z fronty odstranit, a následuje zpracovávání instrukcí z chtěné adresy (kam skok skočil).

Hazardy řeší nová jednotka v procesoru — HMU (Hazard Management Unit).

1.6 OB-5 (APS)

Paměťová hierarchie se skrytou pamětí (cache memory), principy lokality a fungování skryté paměti. Architektura přímé, částečně asociativní, plně asociativní skryté paměti.

Paměťová hierarchie

Rozdíl mezi rychlostí procesoru a rychlostí odpovědi paměti je velký (procesor vs DRAM — 100x, procesor vs HDD — 10 milionkrát). Tento rozdíl se překlene paměťovou hierarchií.

- L1 cache
  - SRAM
  - nejmenší, nejblíže jádru, díky tomu nejrychlejší
  - velikost v řádu jednotek či desítek KB
  - pro každé jádro zvlášť, bývá rozdělena na instrukční a datovou cache
  - obsahuje právě nejpoužívanější data a instrukce
- L2 cache
  - SRAM
  - větší, blízko jádra, latence větší než L1
  - velikost v řádu stovek KB
  - pro každé jádro zvlášť, společná pro instrukce a data
  - obsahuje vše co je v L1 + druhá nejpoužívanější data a instrukce
- L3 cache
  - SRAM
  - ještě větší, stále poměrně blízko jádrům, latence větší než L2
  - velikost v řádu jednotek MB
  - typicky sdílena více jádry, společná pro instrukce i data
  - obsahuje vše co je v L2 + třetí nejpoužívanější data a instrukce
- Hlavní paměť
  - DRAM
  - velká, mimo procesor, tedy latence zásadně vyšší než u cache
  - velikost typicky v řádu jednotek či desítek GB, může být i větší/menší
  - společná pro celý procesor(y), obsahuje data i instrukce
  - obsahuje vše co je v L3 + naprostou většinu potřebných dat i instrukcí
- Sekundární paměť
  - HDD, SSD
  - největší, nejpomalejší
  - společná pro celý počítač, obsahuje vše

Principy lokality

Programy typicky přistupují v daném okamžiku jen k malé části instrukčního a datového adresního prostoru.

Časová lokalita:

- položky, ke kterým se přistupovalo nedávno, budou brzy zapotřebí znovu
- příklad: opakované procházení dat v cyklu, opakované čtení instrukcí v rekurzivních algoritmech

Prostorová lokalita:

- položky poblíž právě používaných budou brzy zapotřebí také
- příklad: sekvenční přístup k instrukcím programu, sekvenční přístup k datovým polím nebo lokálním proměnným umístěným poblíž sebe

Principy fungování skryté paměti

- Cache block

  Souvislý, nedělitelný úsek hlavní paměti, který lze přenést do cache během jedné paměťové transakce.

- Cache hit

  Úspěšný přístup ke cache block — tedy úspěšný přístup k datům/instrukcím, které jsou obsaženy v cache a jsou platné.

- Cache miss

  Opak cache hit — neúspěšný přístup k datům/instrukcím v cache.

- Hit rate

  Úspěšnost přístupů do cache (poměr).

- Miss rate

  Neúspěšnost přístupů do cache (poměr).

- Hit time

  Latence cache, čas na získání dat z dané úrovně cache.

- Miss penalty

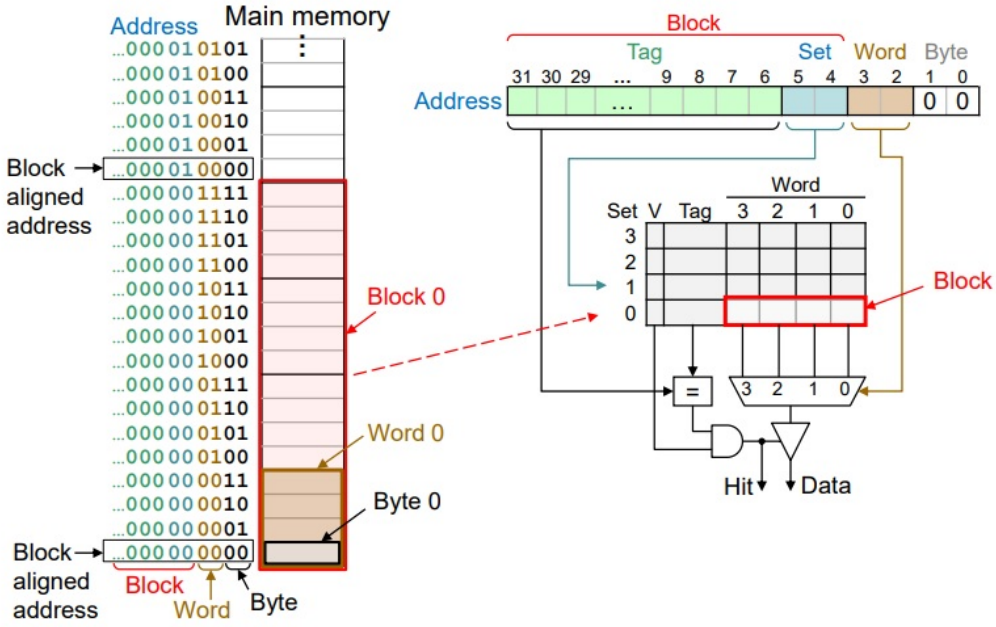
  Celkový čas pro získání dat při výpadku (cache miss) dané úrovně (počítá se od dané úrovně dál).

Průměrný čas přístupu do paměti: Hit time + (Miss rate * Miss penalty)

Adresace skryté paměti

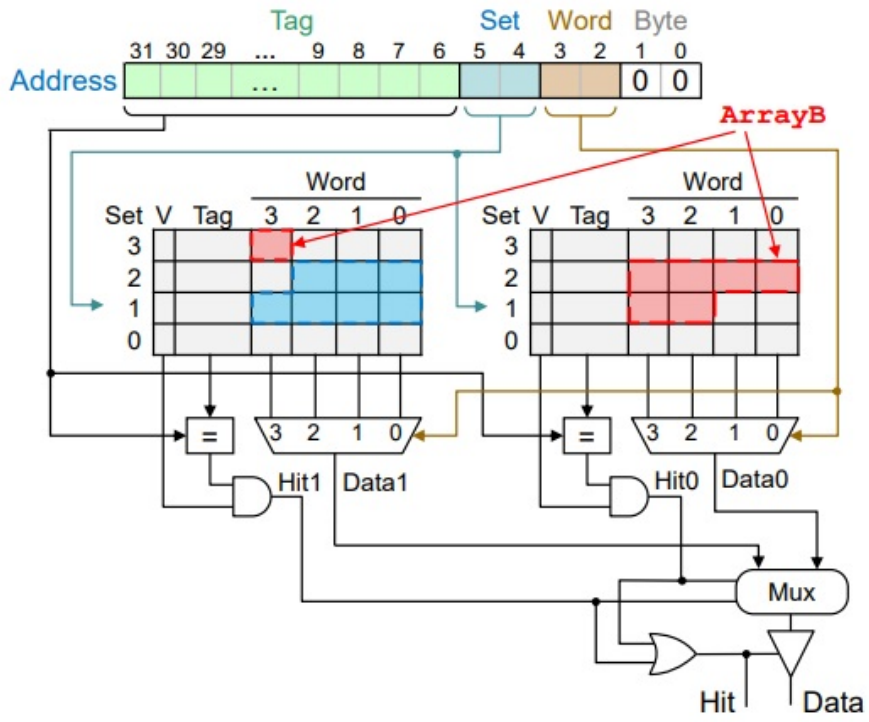
Kusy hlavní paměti se do cache ukládají po blocích. Blok může být různě veliký, typicky obsahuje více bytů (nejmenších adreovatelných kusů paměti).

- Přímo mapovaná cache
  - každý set (řádek) cache obsahuje právě 1 blok.
  - za sebou jdoucí bloky hlavní paměti se mapují do za sebou jdoucích bloků cache
  - řádek = (Adresa v hlavní paměti / velikost bloku) % počet řádků
  - každý blok hlavní paměti má tedy vždy stejný blok cache kam se uloží.
  - na stejný blok cache lze uložit více různých bloků paměti, musíme tedy uložit navíc číslo bloku hlavní paměti (část původní adresy) = Tag



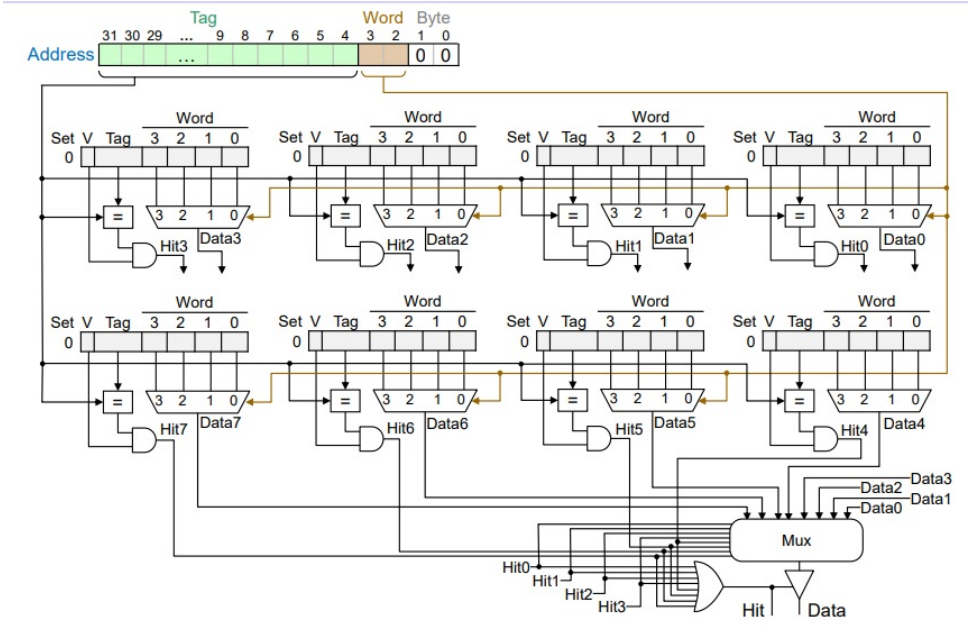
V takovéto cache ale často dochází ke kolizím, když chceme např. přistupovat do více různých částí hlavní paměti. Je ale snazší ji implementovat.

- Cache s částečným stupněm asociativity
  - replikace instancí přímo mapované cache
  - stupeň asociativity je počet takových instancí, neboli také počet cest v cache
  - bloky hlavní paměti lze uložit na více různých míst (přesně na tolik, kolik je stupeň asociativity)



Vyšší složitost implementace, ale zásadně nižší miss rate.

- Plně asociativní cache
  - počet cest je roven počtu bloků cache
  - instance přímo mapované cache mají tedy jen jeden řádek (set)
  - u každého bloku se ukládá celá jeho adresa v hlavní paměti



Nejnáročnější na HW prostředky, ale musí řešit maximální počet kolizí.

1.7 OB-6 (APS)

HW podpora virtualizace hlavní paměti stránkováním, funkce MMU (Memory Management Unit) a překlad virtuálních adres na fyzické adres pomocí TLB (Translation Lookaside Buffer), ošetření výpadku stránky.

Problémy bez virtualizace (motivace pro virtuální paměť):

- Velikost paměti: paměť nemusí být dost velká pro spuštění procesu.
- Fragmetace: Při ukončování procesů vzniknou v hlavní paměti volá místa různých velikostí.
- Dynamická alokace: Jak alokovat další paměť?
- Bezpečnost: Jak zařídit, aby si procesy nemohly číst paměť navzájem?

Jak funguje stránkování?

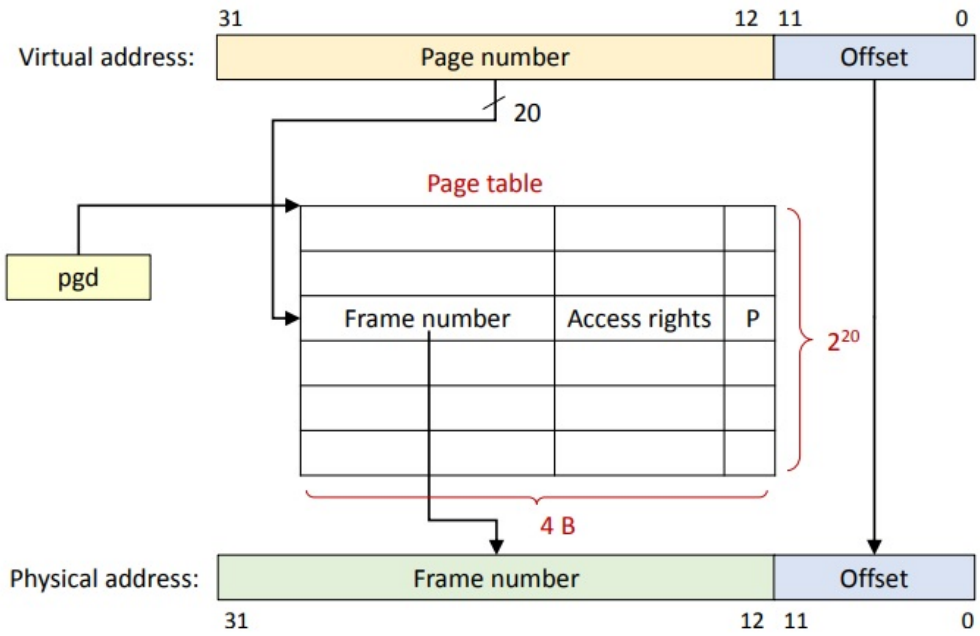
- uživatelským procesům je nabídnut Virtuální Adresní Prostor (VAP)
- každý proces má svůj VAP
- VAP je rozdělen na stejně velké *stránky*, hlavní paměť (HP) je rozdělena nastejně velké *rámce*.
- běžící procesy do rámců HP umisťují momentálně potřebné stránky svého VAP (pracovní množina)
- nepoužívané stránky se při nedostatku paměti odloží na disk
- mapování VAP do HP a přenos stránek mezi HP a diskem zajišťuje OS
- virtuální adresa se skládá z čísla stránky a offsetu
- fyzická adresa se skládá z čísla rámce a offsetu

Možnosti překladu virtuálních adres na fyzické:

- Konvenční stránkovací tabulka
  Každý proces má svou stránkovací tabulku (většinou strom stránkovacích tabulek).
- Inverzní stránkovací tabulka
  Všechny procesy sdílejí jedinou, inverzní stránkovací tabulku.

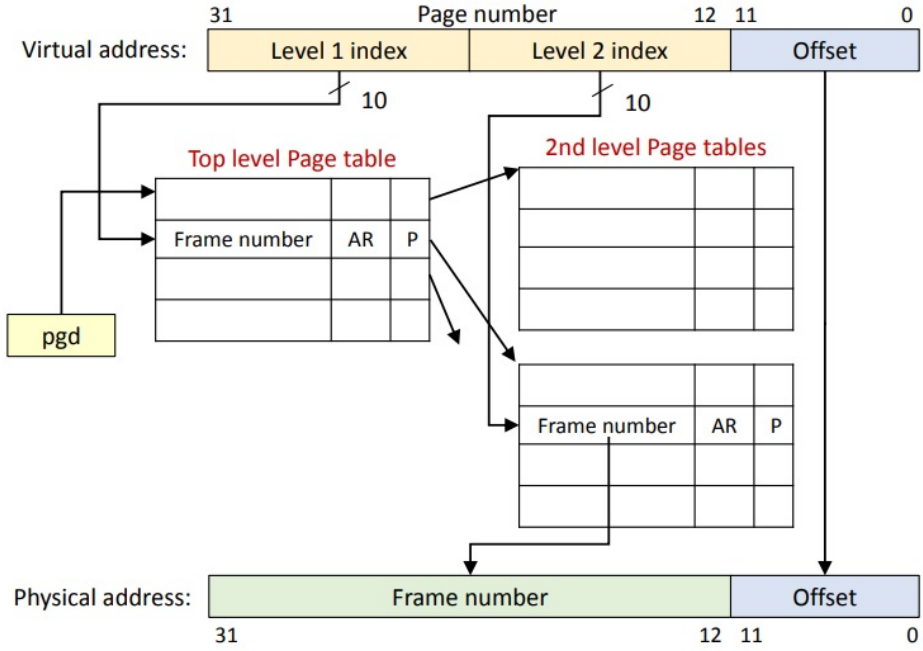
Jednoúrovňová stránkovací tabulka:

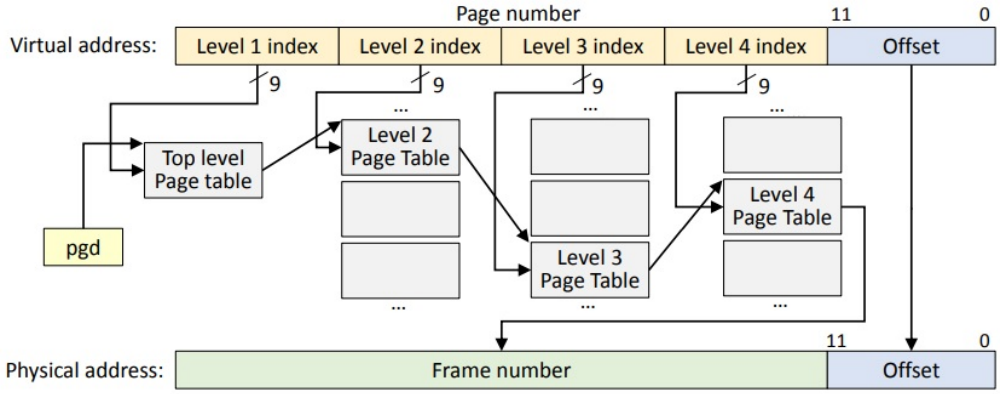
- pro překlad se použije číslo stránky jako index do tabulky, a tam se najde číslo rámce
- jednoúrovňová tabulka je ale i pro 32bitové systémy velká, a když ji má každý proces, zabere to hodně místa



Víceúrovňové stránkovací tabulky:

- pro překlad se použije číslo stránky, které je složené z indexů do různých úrovní stránkovacích tabulek
- jednotlivé tabulky jsou zásadně menší, na počátku stačí jedna tabulka v každé úrovni, další OS může podle potřeby přidat





Při překladu VA na FA se musí procházet několik úrovní tabulek stránek (page walk). Ty mohou být uloženy mimo paměť a může dojít k výpadku stránky (page fault).

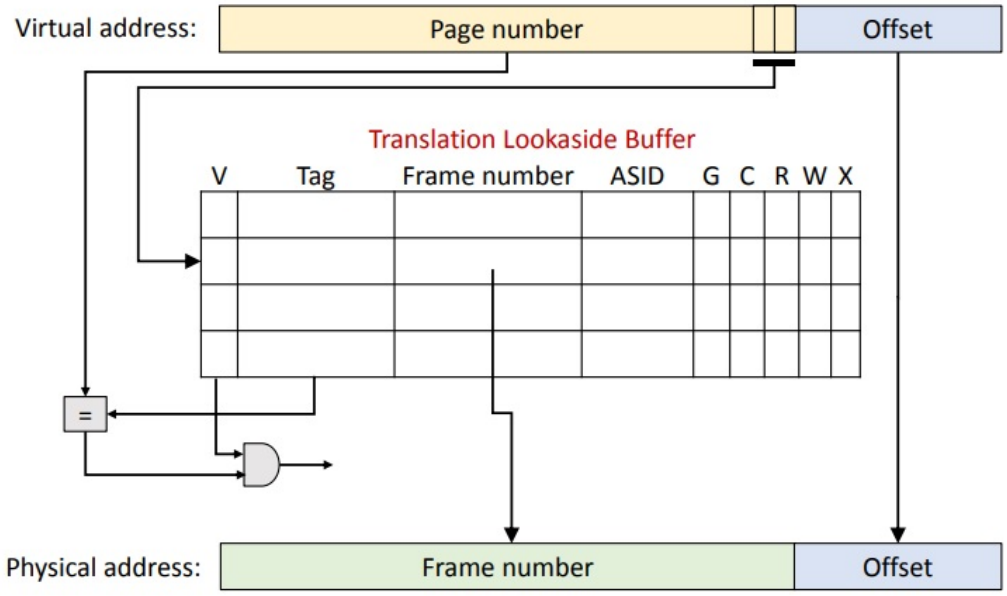
MMU — Memory Management Unit

- vykonává page walk
- dostane adresu stránkovací tabulky první úrovně přes domluvený registr
- pokud nedokáže přeložit adresu, nastává page fault, a procesor generuje výjimku
  - Invalid page fault — adresa není součástí adresního prostoru procesu — obvykle porces zastaven se segmentation fault
  - Valid page fault — adresa je součástí VAP, ale nezle přeložit (nenachází se v MMU, tedy musí page walk vykonat OS / překlad neexistuje — stránka není v HP ale na disku — OS vymění stránky v HP)

TLB — Translation Lookaside Buffer

Vykonávání page walk je časově náročný proces. Aby nebylo nutné pokaždé page walk vykonávat, každá MMU používá speciální HW překladovou tabulku — TLB.

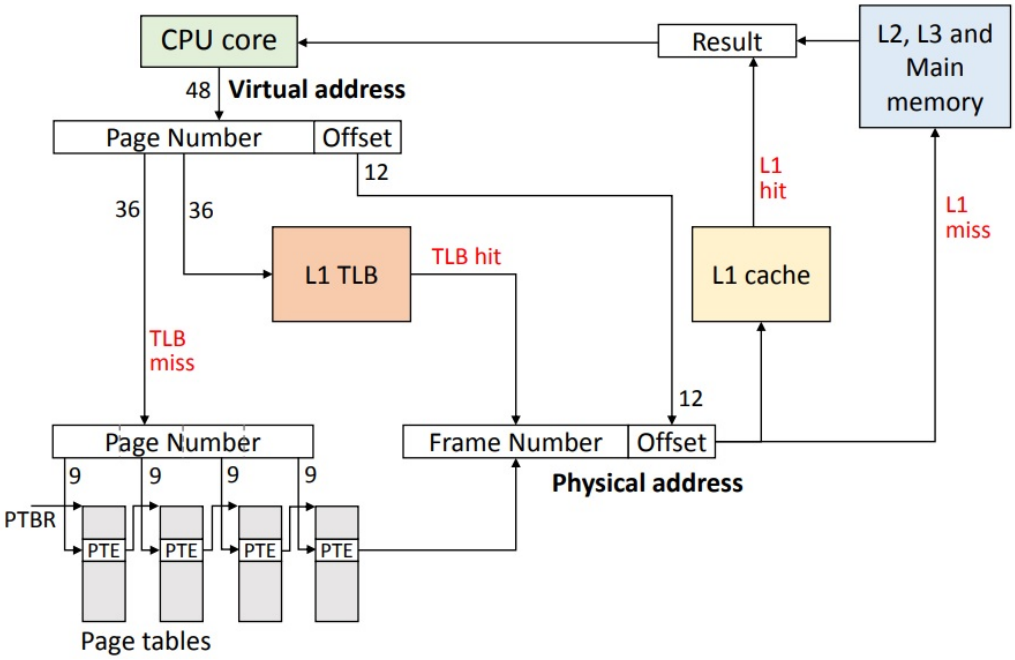
TLB jako přímo mapovaná cache:



Typicky se používá stupeň asociativity 4-64. Podobně jako u cache se používá SRAM.
Každá položka TLB typicky obsahuje:

- V: validity bit
- Tag: číslo stránky (část)
- Frame number: číslo rámce
- ASID: Identifikátor adresního prostoru (pro oddělení procesů)
- G: global flag (pokud G = 1, ASID se ignoruje)
- C: cache policy (informace, jestli jsou data na adrese "cachovatelná"— pro I/O zařízení, kam se změny dat musí posílat rovnou)
- Access permissions: R, W, X

Sumarizace HW podpory:



1.8 SP-18 (OSY)

Virtualizace hlavní paměti stránkováním, principy překladu virtuálních adres na fyzické, struktura tabulek stránek, algoritmy pro nahrazování stránek.

Princip virtuální paměti se stránkováním:

- Proces používá virtuální/logické adresy, ty adresují virtuální adresní prostor
- VAS (virtual adress space) je rozdělen na stejně velké stránky — typicky 4KB nebo 8KB
- na stejně velké úseky (rámce) je rozdělena fyzická paměť
- aktuálně používané stránky musí být aktuálně v hlavní paměti
- virtuální adresa = číslo stránky + offset

Možnosti překladu adres:

- jednoúrovňová tabulka stránek
- víceúrovňová tabulka stránek
- invertovaná tabulka stránek

Překlad adres zajišťuje MMU s TLB (viz 1.7).

Jednoúrovňová TS:

- pro každou stránku VAS daného procesu obsahuje jeden řádek obsahující číslo rámce a kontrolní bity (Present bit (P) — je stránka v hlavní paměti?, Reference bit (R) — přistupovalo se ke stránce?, Modify bit (M) — byl obsah modifikován?, Přístupová práva, Cache disabled/enabled, R/W, User/Supervisor (U/S) - lze přistupovat v uživatelském módu?)
- číslo stránky = index do této tabulky
- pro každý proces jedna tabulka

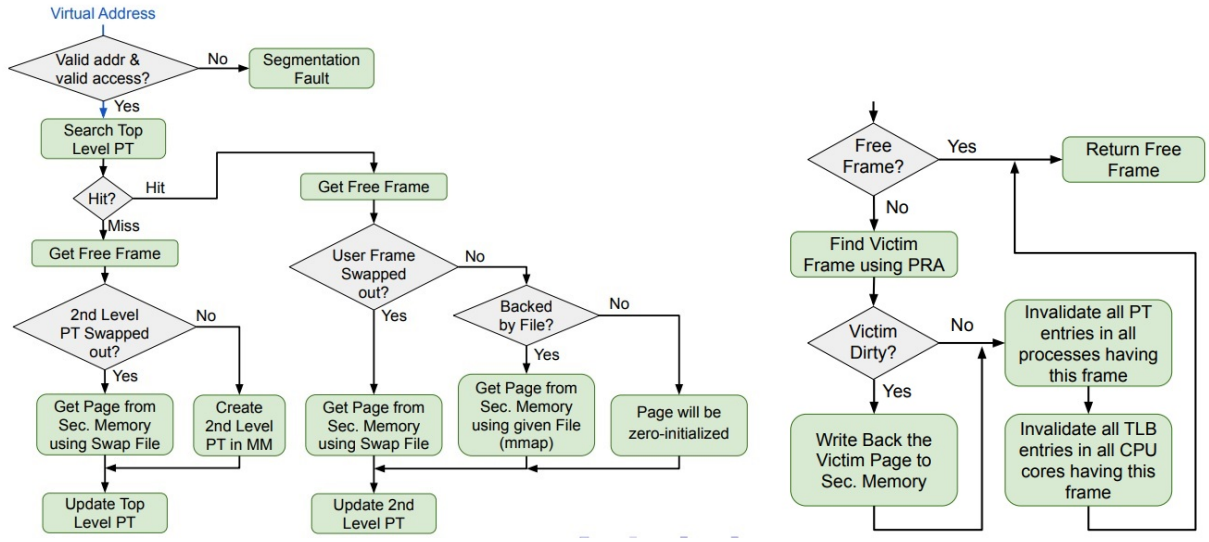
Víceúrovňová TS:

- virtuální adresa se skládá z n indexů, které ukazují do tabulek jednotlivých úrovní + offset
- tabulky stránek úrovní 1,...,$n-1$ obsahují číslo rámce kde je následující tabulka + present bit
- tabulka úrovně n obsahuje present bit + číslo rámce hledané stránky
- hlavní/první tabulka je v paměti vždy

Invertovaná TS:

- obsahuje pro každý rámec fyzické paměti jeden řádek, kde je uloženo: číslo stránky nahrané do rámce, číslo procesu, kterému stránka patří, kontrolní bity, index zřetězení (stejně velký jako index do tabulky)
- existuje 1 tabulka pro celý systém
- číslo stránky se hashovací funkcí převede na index do tabulky
- více stránek se může namapovat na stejný rámec — proto index zřetězení

Řešení výpadku stránek:



Algoritmy pro náhradu stránek

V okamžiku kdy většina/všechny rámce fyzické (hlavní) paměti jsou obsazené, je úkolem OS najít vhodný rámec, jehož obsah (stránka) se uvolní. K tomu slouží algoritmy pro náhradu stránek.

Co je od takových algoritmů požadováno?

- minimalizace počtu výpadků stránek
- rychlost
- jednoduchá implementace

Tyto algoritmy využívají proncipů prostorové a časové lokality.

Optimální algoritmus:

- nahradí se stránka, která má čas příštího přístupu nejdelší
- generuje minimální počet výpadků stránek
- sice nelze udělat, ale slouží pro porovnání kvality reálných algoritmů

NRU (Not Recently Used)

- pro každou stránku se pamatuje reference bit (R) a modified bit (M) (viz výše)
- reference bit se periodicky nastavuje na 0
- stránky jsou rozděleny do 4 tříd (RM == 00, RM == 01, RM == 10, RM == 11)
- nahradí se nějaká stránka z co nejnižší třídy (tedy 00 → 01 → 10 → 11)

Jednoduchý na pochopení i implementaci, poměrně nízký počet výpadků stránek.

FIFO (First In First Out)

- je udržován seznam stránek nahraných v paměti
- nově nahraná stránka je zaznamenána na konec seznamu
- nahrazena je první stránka ze seznamu

Jednoduchý na pochopení i implementaci, ale generuje poměrně vysoký počet výpadků stránek.

Clock algoritmus

- modifikovaný FIFO algoritmus
- seznam stránek jako kruhová fronta
- na počátku ručička ukazuje na první položku seznamu

- pro každou položku je zaznamenán reference bit, který je nastaven na 1 při pridání stránky do seznamu a při přístupu k ní
- při potřebě náhrady stránky ručička u položky, na kterou ukazuje, zjistí stav R bitu — pokud 1, vynuluje a jde na další položku — pokud 0, tato stránka se nahradí a ručička se posune

Jednoduchý na implementaci, generuje poměrně nízký počet výpadků stránek. Existují varianty s více ručičkami, kde podle rychlosti posunu ručiček a jejich rozevření je definováno časové okno, dle kterého zjišťujeme, zda byla stránka nedávno použita.

LRU (Least Recently Used)

- vybere se stránka, která je nejdelší dobu bez přístupu
- pro každou položku je navíc zapamatován čas použití, který se aktualizuje při každém použití (existuje globální čítač, který se zvýší při každém přístupu do paměti, jeho hodnota je pak zanesena k právě použité stránce)
- kandidát je taková stránka, která má nejnižší čas posledního přístupu (nutno porovnat všechny)

Generuje poměrně nízký počet výpadků stránek, dobrá aproximace optimálního algoritmu. Složitější implementace (čítač s časem a porovnání všech stránek)

Aging algoritmus

- simulace LRU lgoritmu
- pro každou stránku je zapamatováno: R bit (nastaví se na 1 při každém přístupu), n-bitový čítač C, který má po načtení stránky do paměti všechny bity na 1
- periodicky se pro každou stránku C posune o 1 doprava, jeho nejvýznamější bit se nastaví na R, a R se nastaví na 0
- vhodným kandidátem je stránka s nejnižší hodnotou C

Menší režie než LRU, ale není tak přesný (nepamatuje se přesný čas, ale jen interval, kdy se naposledy přistupovalo — omezená historie).

1.9 SP-17 (OSY)

Procesy a vlákna, jejich implementace, nástroje pro synchronizaci vláken. Klasické synchronizační úlohy. Uváznutí (deadlock) vláken (alokace prostředků, Coffmanovy podmínky, strategie pro řešení uváznutí).

- **Program:**

  Program je v systému reprezentován spustitelným binárním programem, který je uložený v sekundární paměti (např. disk).

- **Proces:**

  Instance spuštěného programu/aplikace. Entita, v rámci které jsou alokovány prostředky (paměť, vlákna, otevřené soubory, zámky, semafory, sokety,...).

- **Vlákno:**

  Výpočetní entita (proud instrukcí), které je přidělováno jádro CPU. Vlákna vytvořená v rámci procesu sdílí většinu prostředků alokovaných v tomto procesu.

Vytvoření procesu:
Nový proces lze vytvořit jako kopii/klon původního procesu, či jako úplně nový proces. V Unixu fork() exec(), ve Windows CreateProcessA().

- **fork():**

  Vytvoří nový proces, který je kopií toho procesu, ze kterého byla tato funkce zavolána. V případě chyby vrací -1, v potomkovi vrací 0, v rodiči vrací PID potomka.

- **exec():**

  Adresový prostor aktuálního procesu je přepsán obsahem souboru, který se začne vykonávat od začátku.

- **wait():**

  Zablokuje rodičovský proces, ve kterém je zavolána, dokud se konkrétní/jeden potomek neukončí.

Ukončení procesu:

- jádro se pokusí předat návratový kód rodiči
- ukončí se všechna vlákna pod procesem
- uvolní se adresový prostor procesu a příslušné struktury OS
- proces se může ukončit sám (buď normální konec programu jako *return*, nebo chyba, kvůli které se sám ukončí), nebo může být ukončen jádrem (fatální chyba nebo signál od jiného procesu)

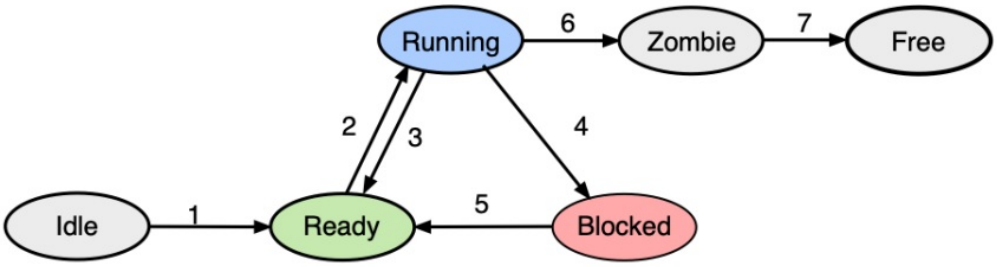
Vlákna:

- proces se implicitně vytváří s jedním "main"vláknem
- další vlákna lze vytvořit z hlavního (volání OS)

Plánování:

- vláken je typicky zásadně více než logických jader procesoru
- jedno vlákno je zpracováváno max 1 logickým jádrem
- aby se vlákna na jádrech vystřídala, používá se typicky preemptivní plánování
  - vlákno je na základě plánovacích kritérií vybráno, a je mu přiděleno volné jádro CPU
  - vláknu je přiděleno množství času na CPU
  - vláknu je jádro odebráno, pokud uplyne přidělený čas, vlákno provede systémové volání nebo dojde k přerušení
- přepínání kontextu (vystřídání vláken na jádře CPU)
  - kontext = všechny nezbytné informace pro pozdější spuštění přerušeného vlákna od okamžiku přerušení
  - kontext se uloží do paměti, naplánuje se další vlákno a jeho kontext se nahraje do CPU jádra

Stavy vláken:



- **Časově závislé chyby:**

  Situace, kdy více vláken používá společné sdílené prostředky a výsledek deterministického algoritmu je závislý na rychlosti jednotlivých vláken, které používají tyto prostředky. Špatně se detekují — lze předcházet správným návrhem paralelního algoritmu.

- **Kritická sekce:**

  Část programu, kde vlákna používají sdílené prostředky.

- **Sdružené kritické sekce:**

  Kritické sekce více vláken, které se týkají stejného sdíleného prostředku.

- **Vzájemné vyloučení:**

  Vláknům není dovoleno sdílet stejný prostředek ve stejném čase, tedy se nenachází ve stejné sdružené sekci současně.

- **Korektní paralelní program:**

  Nesmí klást předpoklady na rychlost vláken a počet jader. Musí zajistit výlučný přístup ke sdíleným prostředkům. Mimo kritické sekce by vlákno nemělo být zpomalováno ostatními vlákny.

Problémy s použitím synchronizace vláken:

- deadlock (x vláken čeká na událost, kterou může vyvolat jen jedno z čekajících vláken)
- livelock (několik vláken vykonává neužitečný výpočet, ale nemohou dokončit)
- starvation (vlákno ve stavu "ready"předbíháno a dlouho se nedostane na řadu)

Zamykání kritických sekcí:

- zámek značí, zda je ve sdružené kritické sekci už jiné vlákno — musí se k němu přistupovat atomicky (např. TSL — Test and Set Lock)
- aktivní vs blokující čekání
- **Zámek (mutex):**

  Pamatuje si svůj stav (zamčený/odemčený) a množinu vláken blokovaných na něm. Jsou nad ním definovány atomické operace lock() a unlock().

- **Podmíněná proměnná (conditional variable):**

  Pamatuje si, která vlákna jsou na ní blokována. jsou definovány operace cond_wait(mutex) a cond_signal(). cond_wait(mutex) — mutex musí být zamčen volajícím vláknem, funkce odblokuje mutex a zablokuje vlákno dokud nepřijde cond_signal().

- **Semafor:**

  Obsahuje čítač a seznam blokovaných vláken. Jsou definovány operace sem_init(int) (nastaví čítač na 0), sem_wait() (vstup do sekce — pokud čítač je větší než 0 tak vlákno vstoupí a dekrementuje čítač, jinak se zablokuje) a sem_post() (uvolní nějaké čekající vlákno, nebo inkrementuje čítač).

- **Bariéry:**

  Obsahuje čítač (síla bariéry — kolik vláken musí čekat, aby byla odblokována) a blokovaná vlákna. Operace: barrier_init(int) (nastaví sílu bariéry) a barrier_wait() (pokud čítač je více než 1, vlákno čeká a čítač je dekrementován, jinak jsou všechna vlákna probuzena)

Synchronizační úlohy:

- Večeřící filosofové
  - N filosofů u kulatého stolu
  - každý má před sebou jídlo a mezi sousedními talíři je vždy 1 vidlička (celkem tedy N vidliček)
  - pokud chce filosof jíst, musí získat obě vidličky vedle jeho talíře
  - stavy filosofa: přemýšlí (nechce a nemá vidličky), má hlad (pokouší se získat obě vidličky), jí (má obě vidličky)
  - optimální řešení: může jíst až $\lfloor N/2 \rfloor$ filosofů, nevznikají časově závislé chyby ani synchronizační problémy
  - řešení: pokud mam hlad zamknu mutex, kouknu jestli jsou volny vidlicky, pokud ne spim (odemykam mutex), pokud jo, beru, odemykam mutex a jim. Az dojim, zamknu mutex, vratim vidlicky, probudim sousedy a odemknu mutex.
- Čtenáři — písaři
  - v systému je 1 sdílený prostředek
  - písaři mohou modifikovat, čtenáři pouze číst
  - chceme, aby pokud není modifikováno (nepřistupuje písař) mohlo číst více čtenářů
  - zároveň by nikdo neměl být předbíhán
  - řešení: písaři i čtenáři se řadí do fronty, ale po skupinách — pokud přijdu na konec fronty a je tam už stejný typ, přidám se do skupiny — na začátku fronty je probuzena celá skupina a buď písaři postupně zapíšou, nebo čtenáři společně přečtou
- Spící holiči
  - v holičství je N holičů a křesel k holení, a M křesel k čekání
  - pokud nejsou zákazníci, holič sedne do holícího křesla a usne
  - pokud přijde zákazník, buď probudí holiče (pokud je volný), nebo si sedne do čekárny (pokud je místo) jinak odejde

Obecně alokace prostředku:

- vlákno žádá o prostředek pomocí alokační funkce
- pokud je prostředek volný, je přidělen
- pokud je již alokovaný, vlákno může být blokováno (v závislosti na alokační funkci — mutex_lock blokuje, mutex_try_lock blokuje jen na určitý čas, fork() a malloc() neblokují)
- v případu 2 a 3 se vlákno pak samo rozhodne jak pokračovat

Coffmanovy podmínky
Uváznutí (deadlock) nastane pouze pokud jsou splněny všechny následující podmínky:

- Vzájemné vyloučení — každý prostředek nemůže být sdílen více vlákny
- Podmínka neodnímatelnosti — již přidělený prostředek nemůže být odebrán násilím
- Podmínka "drž a čekej"— vlákno s již přiděleným prostředkem může žádat o další
- Podmínka kruhového čekání — musí existovat smyčka více vláken, ve které každé vlákno čeká na prostředek držený dalším vláknem ve smyčce

Řešení uváznutí:

- Pštrosí strategie — ignorování
- Prevence uváznutí — nesplnění alespoň jedné z Coffamnových podmínek
- Předcházení vzniku uváznutí — pečlivá alokace prostředků
- Detekce uváznutí a zotavení — uváznutí je detekováno a odstraněno

Implementace procesů:

- jádro OS si udržuje zřetězený seznam struktur — tabulku procesů
- jedna položka tabulky obsahuje vše nezbytné, co si OS musí o procesu pamatovat (PCB — Process Control Block)
  - identifikace procesu (id procesu, id rodiče, číslo úlohy/seance/projektu, jméno procesu...)
  - identita/bezpečnost (vlastník, skupiny, práva procesu)
  - informace o alokovaných prostředcích (paměť, soubory, prostředky pro meziprocesovou komunikaci)

Implementace vláken:

- Thread Control Block (TCB) — identifikace vlákna, info o přepínání kontextu (registry), informace pro plánování vláken
- Implementace v uživatelském prostoru (zastaralé)
  - OS přistupuje k procesům jako by měly jedno vlákno
  - proces si svá vlákna spravuje sám
  - kooperativní plánování pro vlákna v procesu
- Implementace v jádře OS
  - OS plánuje samotná vlákna, ne procesy
  - OS udržuje jede PCB pro každý proces, jeden TCB pro každé vlákno
  - preemptivní plánování pro vlákna v procesu

Plánování v dnešních OS: Prioritní Round Robin
X front s různou prioritou. Na základě předchozího běhu vlákna se buď zvýší časové kvantum a sníží priorita (pokud vlákno využilo všechen čas) nebo zvýší priorita a sníží čas (pokud nevyužilo celé časové kvantum).

1.10 OB-3 (ADU)

Procesy a systémové služby v unixových operačních systémech: hierarchie a vzájemné vazby, limity, zapínání a vypínání systému, logování aktivit systému.

Procesy:

- vykonávané programy
- mají svoje ID (PID), id uživatele (UID), id rodiče (PPID) ...
- každý proces někdo vytvořil, tedy každý proces má nějaké PPID — init proces je první proces spuštěn při bootu, má PID = 1 a PPID = 0
- rodič typicky čeká, až mu dítě předá návratový kód
- daemon = systémový proces (běží na pozadí bez nutnosti vstupu ... jako u windows služby)
- orphan = proces, jehož rodič už neexistuje — adoptuje se pod proces init
- zombie = proces, který již skončil, ale ještě si rodič nepřevzal návratovou hodnotu
- procesy si navzájem mohou posílat signály

Limity:

- soft limit — limit mezi 0 a hard limitem, lze uživatelsky přenastavit
- hard limit — může změnit jen root, nelze překročit
- lze nastavit max počet procesů pro uživatele a další omezení využití zdrojů (paměť, velikost souborů, počet file descriptorů...)
- příkaz ulimit

BOOT

- Firmware fáze
  - POST (Power On Self Test)
  - inicializace HW a driverů
  - výběr bootovacího zařízení
  - načtení a spuštění bootovacího kódu
- Boot-loader fáze
  - nalezení a načtení boot programu
  - boot program se nahrává do pevně daných adres v paměti
  - kontrola řízení se předá programu
- GRUB (Grand Unified Boot Loader) fáze
  - načte se GRUB konfigurace
  - vybere se odkud/co bootovat
  - kernel se načte a spustí
- Kernel fáze
  - inicializace datových struktur jádra
  - načtení driverů, inicializace HW
  - mount kořenového filesystému
  - načtení konfigurace jádra
  - načtení modulů
  - vytvoření init procesu

- Init fáze
  - dříve spouštění start/stop scriptů dle úrovní běhu systému (run levels / milestones)
    - 0 — systém vypnutý
    - 1, s, S — single user mode
    - 2 — multi user mode
    - 3 — multi user mode + síť
    - 4 — nepoužívaný (občas GUI)
    - 5 — vypnutí
    - 6 — restart
  - v dnešní době spouštění služeb
  - konfigurace procesu init — /etc/inittab
  - init se přes fork() a exec() naklonuje a spouští další procesy

Vypínání systému:

- shutdown *level timeout -y* (přívětivý jak k uživatelům a aplikacím, tak k systému)
- init *level* (přívětivý k aplikacím a systému)
- poweroff, restart (přívětivé k systému)
- vypnout napájení (fuj)

Logování:

- servrová logovací služba s dlouhou historií
- umožňuje oddělit SW generující zprávy, SW který je ukládá a SW který je analyzuje a nahlašuje
- konfigurace: /etc/syslog.conf
- logovací složky: /var/log a /var/adm
- mnoho variant
- záznamy v logu mají info o službě a závažnosti logu (alert, critical, error, warning, notice, info, debug)

1.11 OB-2 (ADU)

Správa disků a souborových systémů (zařízení, souborové systémy UFS (EXT) a ZFS, disková pole RAID, diskové kvóty), síťové souborové systémy (NFS, CIFS), swap v unixových operačních systémech.

2 Šifrování a sítě

3 Obecná bezpečnostní teorie

4 Matematika

5 Programování