1 Programovacie Jazyky

1.1 Základné pojmy

V kontexte programovacích jazykov uvažujeme dva základné pojmy:

- Výraz (expression) - kombinácia operátorov, konštánt, premenných, funkcií a iných prvkov jazyka ktorá sa vyhodnotí na nejakú výslednú hodnotu (môže ísť o prázdnu hodnotu). Výraz môže ale nemusí mať vedľajšie efekty. (Príklady: 2+1.5, A == 3, null, pop())
- Príkaz (statement) - Najmenšia vykonateľná samostatná jednotka imperatívneho jazyka. Príkaz nevracia hodnotu a podstatné sú teda len jeho vedľajšie efekty. (Príklady: A = 4, return 5, if A == 3 then X)

Program sa tradične skladá z postupnosti príkazov ktoré pri svojom vykonaní môžu vyhodnocovať rôzne výrazy potrebné k ich dokončeniu. Vyhodnocovanie výrazov môže prebiehať podľa rôznych pravidiel, tieto pravidlá potom súhrnne nazývame *vyhodnocovacia stratégia* (napr. lenivá alebo striktná).

Príkazy sa bežne združujú do blokov (v niektorých jazykoch je pojem blok zameniteľný s pojmom funkcia) , pričom bloky môžu byť pomenované aj nepomenované a tradične majú jeden vstupný bod a jeden alebo viac výstupných bodov (return).

Na označenie výrazov ktoré sa vyhodnotia na true alebo false (tzv. booleovské výrazy) sa často používa pojem *podmienka*.

Pri programovacích jazykoch môžeme hovoriť jednak o *syntaxi* - teda o spôsobe, akým sú jednotlivé príkazy a výrazy zapísané - a o *sémantike* - teda o tom, akým spôsobom sa jednotlivé príkazy a výrazy vykonávajú.

1.2 Dátové typy

Typ predstavuje vlastnosti a spôsob reprezentácie nejakých dát počítačom. Typy delíme na:

1.2.1 Prázdny dátový typ (void)

Reprezentuje absenciu hodnoty.

1.2.2 Primitívne dátové typy

Tieto typy sú typicky implementované na úrovni jazyka a väčšinou sa priamo mapujú na reprezentáciu údajov v procesore. Typicky sem patrí integer, Float, Boolean, Char, Fixed Point(fixný počet desatinných miest), Bignum(ľubovoľne veľké číslo) a ich varianty s rôznymi bitovými šírkami v závislosti na architektúre.

1.2.3 Zložené dátové typy

Zložené typy vznikajú spojením niekoľkých hodnôt primitívneho typu. Výsledkom je potom dátová štruktúra. Rôzne jazyky implementujú rôzne základné dátové štruktúry. (Dátová štruktúra = spôsob efektívneho uloženia informácie v pamäti) Nízkoúrovňové jazyky často žiadne dátové štruktúry neimplementujú a poskytujú iba priamy prístup do pamäte. Vyššie programovacie jazyky potom implementujú napr:

- Štruktúra - fixný počet prvkov s jasnou adresou (nemusí byť priamo za sebou, ale je pevne dané ako sa k jednotlivým prvkom dostanem)
- Union/Tagged Union - Taktiež má fixnú veľkosť, avšak môže obsahovať hodnoty rôznych typov. Prípadne môže zahŕňať príznak indikujúci ktorý typ je aktuálne uložený.
- Pole - prvky sú uložené v pamäti postupne za sebou. Typicky ide o prvky jedného typu.
- Reťazec - typicky pole znakov, avšak môže byť zložitejší, keďže nie všetky kódovania majú fixnú dĺžku znaku
- Zoznam - Prvky sú nejakým spôsobom zoradené, avšak nemusia byť nutne v pamäti za sebou (spojovaný zoznam). Nie vždy musí ísť o prvky jedného typu.
- Množina - Prvky nie sú zoradené a každý prvok je v množine maximálne raz.
- Asociatívne pole/mapa - Umožňuje prístup k prvkom na báze kľúč/hodnota.

1.2.4 Výpočtové typy (enum)

Výpočtový typ je typ, ktorý môže nadobúdať hodnoty z pevnej danej množiny prvkov. Tieto hodnoty môžu, ale nemusia byť usporiadateľné.

1.2.5 Odkazy a referencie

Ide o hodnoty ktoré obsahujú odkaz na nejaký kus pamäte. Rozdiel medzi odkazom a referenciou nie je pevne daný, ale všeobecne sa predpokladá že referencia je ïnteligentnejší"pointer (napr. taký pre ktorý sa priebežne kontroluje či je pamäť kam odkazuje platná).

Typicky existuje špeciálna hodnota pre odkaz, ktorá predstavuje neplatnú pamäť (null).

Odkaz môže potenciálne ukazovať aj na spustiteľný kód.

1.2.6 Abstraktné dátové typy

Abstraktný dátový typ popisuje isté vlastnosti pamäte ktorú reprezentuje, ale necháva priestor na rôzne implementačné detaily. Typicky napr. rozhranie v objektovom programovaní.

1.3 Riadiace štruktúry *(control structures)*

Riadiaca štruktúra je príkaz ktorý na základe poskytnutých údajov (precondition) mení poradie/smer vykonávania programu (control flow).

1.3.1 If-Then-Else

Riadiaca štruktúra If-Then-Else umožňuje podmienené vykonávanie blokov kódu. V prípade splnenia podmienky danej klauzulou IF sa vykoná blok kódu daný klauzulou Then, v opačnom prípade blok kódu daný klauzulou Else. V extrémne jednoduchých jazykoch nemusí byť klauzula Else prítomná, avšak dá sa nahradiť testom na negáciu pôvodnej podmienky.

If-Then-Else sa dá v prípade potreby násobne vnárať (do Else vetvy sa vloží ďalšie IF). Pokiaľ jazyk takúto konštrukciu podporuje priamo na syntaktickej úrovni, hovoríme o riadiacej štruktúre If-Then-Elseif-Else.

1.3.2 Switch/Case a Pattern Matching

V prípade že treba zvážiť veľké množstvo prípadov, stáva sa často kombinácia if-else veľmi neprehľadnou. Z toho dôvodu sa zaviedla riadiaca štruktúra Switch (V mnohých jazykoch nazývaná aj Case). Príkaz switch sa tradične vyhodnocuje pre nejakú premennú, pričom v tele príkazu sú uvedené možné hodnoty danej premennej a k nim sú priradené bloky kódy. Pri vykonávaní príkazu sa vyhodnotí ten blok kódu, ktorý je priradený hodnote ktorú daná premenná aktuálne nadobúda. Často je taktiež možné uviesť ešte dodatočný blok, ktorý sa vykoná v prípade že žiadna z hodnôt nevyhovuje aktuálnemu obsahu premennej.

Pattern matching (Hľadanie vzorov?) je potom rozšírenie tejto techniky, ktoré dovoľuje okrem klasickej rovnosti testovať na širšiu skupinu vlastností (napr. regulárne výrazy v prípade textových reťazcov) prípadne sa pýtať na samotnú štruktúru/typ premennej (je to dvojica? Je to zoznam s troma prvkami?). Pattern matching je oveľa silnejší ako samotný switch, avšak narozdiel od switchu, daná premenná nemusí vždy vyhovovať len jednému vzoru (problém či sú dva vzory disjunktné/ekvivalentné nemusí byť ani rozhodnuteľný). V takom prípade sa štandardne vykoná kód asociovaný s prvým nájdeným vyhovujúcim vzorom. Konkrétna špecifikácia podporovaných vzorov je potom závislá priamo na jazyku.

1.3.3 While-Do/Do-While

Riadiaca štruktúra While-Do (while cyklus) umožňuje opakovane vykonávať daný blok kódu v prípade že je splnená potrebná podmienka. Pri vykonávaní sa najskôr otestuje podmienka daná klauzulou While. V prípade že je podmienka splnená, vykoná sa blok kódu daný klauzulou Do (telo cyklu). Následne sa proces opakuje až do okamihu kedy je podmienka nesplnená. Ide o najjednoduchší používaný spôsob iterácie s premenlivým počtom opakovaní.

Alternatívou k While-Do je štruktúra Do-While ktorá sa od While-Do líši iba v tom, že vyhodnotenie prebieha v opačnom poradí. Teda najskôr sa vykoná kód daný klauzulou Do a až následne sa testuje podmienka. V prípade že je podmienka splnená sa opäť pokračuje ďalším vykonaním tela cyklu. Praktický význam Do-While spočíva v tom, že garantuje aspoň jedno vykonanie tela cyklu.

1.3.4 For-Do/Foreach

Riadiaca štruktúra For-Do (for cyklus) je rozšírenie while cyklu ktoré umožňuje prirodzenejšiu kontrolu nad počtom iterácií. For cyklus sa skladá z dvoch príkazov, jednej podmienky a tela cyklu. Prvý príkaz slúži na inicializáciu iterácie a vykoná sa len raz, hneď pri vstupe do for cyklu. Následne sa vyhodnotí podmienka. V prípade že je podmienka splnená, vykoná sa telo cyklu. Ak podmienka splnená nie je, cyklus končí. Po úspešnom vykonaní tela cyklu sa vykoná posledný tretí príkaz, ktorý má za úlohu posunúť stav na ďalšiu iteráciu. Následne sa pokračuje opäť testovaním podmienky.

Druhou, striktnejšou variantov for cyklu je tzv. Foreach cyklus. Foreach cyklus sa používa na zlepšenie prehľadnosti kódu v miestach, kde je počet iterácií cyklu dopredu známy (napr. iterácia cez všetky prvky poľa). Foreach cyklus vykoná telo cyklu raz pre každý prvok nejakej iterovateľnej premennej (Čo je to iterovateľná premenná závisí na implementácií daného jazyka, štandardne ide o nejaký zoznam, množinu, prípadne ohraničený interval).

> Poznámka: Alternatívny spôsob delenia cyklov je na Podmienkami riadené (while), Počítadlom riadené (for) a Riadené kolekciou (foreach).

1.3.5 Break/Continue

Riadiace štruktúry Break a Continue umožňujú predčasné ukončenie iterácie cyklu alebo celého cyklu.

Príkaz break vykoná okamžitý skok hneď za koniec cyklu, bez toho aby kontroloval podmienky alebo vykonával nejaké dodatočné akcie. Break je taktiež jediný spôsob ako zastaviť nekonečný cyklus. Alternatívou príkazu break môžu byť cykly s tzv. stredovou podmienkou, čo je v podstate podmienený break.

Príkaz continue funguje obdobne, avšak skočí iba na koniec tela cyklu, čím efektívne preskočí zbytok práve vykonávanej iterácie. Ďalej sa pokračuje štandardne, teda sa vyhodnotí podmienka cyklu prípadne sa vykonajú nejaké ďalšie nutné akcie.

Okrem break a continue sa používajú ešte príkazy Retry a Redy ktoré sú prakticky opakom break a continue. Retry spôsobí skok pred začiatok cyklu, takže sa celý cyklus začne vykonávať znovu (ale nevracia premenné do pôvodného stavu). Retry zase spôsobí skok na začiatok iterácie. Tieto príkazy sa často nepoužívajú, lebo majú pomerne neintuitívnu sémantiku, avšak sú implementované napr. v Ruby alebo Perle.

1.3.6 Goto a Značky (labels)

Label je označené/pomenované miesto v programe. Príkaz goto label (prípadne podmienené goto) je potom skok na danú značku. Všeobecne sa neodporúča používať kvôli zlej čitateľnosti výsledného kódu (program stráca "lineratiu").

Príkaz lebel sa niekedy používa v súvislosti s vnorenými cyklami a príkazmi break/continue na indikáciu toho, na ktorý cyklus sa daný break/continue vzťahuje.

1.3.7 Throw, Try-Catch, Finally (Výnimky)

Throw-Try-Catch-Finally je riadiaca štruktúra súvisiaca s obsluhou výnimiek (výnimočných/nečakaných udalostí v programe). Výnimky slúžia na relatívne pohodlnú obsluhu možných zlyhaní a problémov v programe bez nutnosti zavádzať niektoré explicitné kontroly.

Príkaz Try-Catch(E) indikuje, že sa má vykonať kód v bloku danom klauzulou Try. Pokiaľ je pri vykonávaní tohto

kódu vyvolaná výnimka typu E, pôvodné vyhodnocovanie sa zastaví a vykoná sa blok daný klauzulou Catch.

V súvislosti s Try-Catch sa používa aj príkaz Finally, ktorý je spojený s dvomi blokmi kodu. Prvý blok sa vyhodnotí normálne (teda v prípade vyvolania výnimky skončí) pričom pre druhý blok platí, že sa vyhodnotí po skončení prvého bloku vždy, bez ohľadu na to či sa v prvom bloku vyvolá výnimka alebo nie.

Príkaz Throw vyvolá výnimku. Vyvolanie výnimky v podstate znamená skok do najbližšieho Catch bloku zodpovedného za obsluhu daného typu výnimky. Pokiaľ taký blok neexistuje, vykonávanie programu je zastavené s chybou. (Výnimka môže byť okrem throw vyvolaná aj inými príkazmi jazyka, napr. pri delení nulou)

1.4 Typy programovacích jazykov

Existuje množstvo kritérií podľa ktorých sa dajú deliť programovacie jazyky. Väčšina jazykov sa ale nedá vždy úplne striktne zaradiť do jedného typu.

Jedným kritériom je paradigma ktoré implementujú:

- Imperatívne - štruktúrované/objektové paradigma (C, C++, Java, .NET)
- Deklaratívne - funkcionálne/logické paradigma (Haskell, Prolog, Lisp, F#)

Iným je typový systém ktorý používajú:

- Staticky typované - Typy sa kontrolujú iba podľa obsahu zdrojového kódu (napr. v čase kompilácie) (C)
- Dynamicky typované - Typy sa kontrolujú priamo za behu programu (Python)

Iné možné kategorizácie sú potom podľa účelu, miery abstrakcie a podobných "neobjektívnych"metrík :)

Posledným dôležitým kritériom je spôsob vykonávania kódu. V tomto prípade sú tri základné možnosti: Interpretácia, Kompilácia a Just-In-Time kompilácia.

1.4.1 Kompilácia

Kompilované programovacie jazyky sú pred spustením prevedené z textového zápisu priamo do strojového kódu ktorý je vykonávaný procesorom. Kompilované jazyky sú väčšinou rýchle, pretože kompilátor môže stráviť netriviálny čas optimalizáciou jednotlivých častí kódu. Zároveň ale vyžadujú rôzny kompilátor pre rôzne architektúry procesorov a ťažko sa prispôsobujú dynamickým vlastnostiam zariadenia (napr. počtu dostupných procesorov alebo množstvu pamäte). Kompilácia veľkých programov je taktiež často zdĺhavá, teda spomaľuje samotný vývoj.

Kompilácia prebieha v niekoľkých krokoch, ktoré sa môžu prípadne opakovať. Celý proces sa štandardne rozdeľuje na tzv. Front-End a Back-End.

Front-End je tradične zodpovedný za:

- Preprocessing - Rozvinutie makier, importovanie súborov...
- Lexikálna analýza - Rozdelenie súvislého textu do tzv. tokenov. Token je jedna atomická jednotka jazyka (číslo, kľúčové slovo, názov premennej). Na špecifikovanie a hľadanie tokenov sa štandardne používajú regulárne výrazy a automaty.
- Syntaktická analýza - Sekvencia tokenov je usporiadaná do stromu(parse tree) daného gramatikou spracúvaného jazyka. Rozpoznávajú sa jednotlivé príkazy a výrazy.
- Sémantická analýza - Vyhodnocuje sa korektnosť zadaného programu z pohľadu sémantiky daného jazyka. Objekty v spracúvanom strome sa doplnia o sémantické informácie na základe ktorých sa vyhodnotia typy, prítomnosť nedefinovaných symbolov a pod.
- Výstup do medzijazyka - Pre zjednodušenie sa výsledný strom často ukladá do medzijazyka ktorému bude rozumieť back-end fáza spracovania. Napr. assemblér alebo LLVM bitkód.

Back-End následne pracuje s výstupom front-endu, teda buď so samotným sémantickým stromom alebo s kódom v medzijazyku. Tradične vykonáva tieto kroky:

- Analýza - analyzuje sa control flow a data flow programu, rôzne závislosti premenných a pod.
- Optimalizácia - Na základe analýzy sa program transformuje tak, aby pracoval efektívnejšie prípadne aby bol menší.
- Generovanie kódu - Optimalizovaný program sa uloží vo finálnej podobe spustiteľnej na cieľovom procesore.

1.4.2 Interpretácia

Interpretovaný program je vykonávaný interpretom, čo je program (sám môže byť kompilovaný alebo interpretovaný) ktorý priebežne parsuje a vykonáva zdrojový kód daného programu. Interpretované programy sú typicky pomalšie, pretože kód sa prakticky neoptimalizuje a navyše je nutné ho dynamicky spracúvať počas vykonávania. Avšak odpadá nutnosť kompilácie, čím sa zrýchľuje vývoj a zlepšuje prenositeľnosť jazyka.

Alternatívne niektoré interpretery môžu zdrojový kod pred spustením preložiť do istej formy medzijazyka ktorý sa potom jednoduchšie interpretuje.

1.4.3 Just-In-Time Kompilácia

Just in time (JIT) kompilácia sa snaží spojiť výhody interpretácie a kompilácie. Kód takéhoto jazyka sa štandardne kompiluje až v okamihu keď je spustený (tradične sa taktiež nekompiluje celý naraz, ale kompilujú sa len tie kúsky ktoré budú skutočne spustené). Prípadne je možné časť programu iba interpretovať a kompilovať iba často vykonávané miesta. Tým sa výrazne zvýši výkon a zároveň sa zachová väčšina dobrých vlastností interpretovaných jazykov.

JIT sa hodí hlavne pre dlhodobo bežiace aplikácie, keďže čas potrebný na prvotnú kompiláciu je v takom prípade zanedbateľný. Výhodou je taktiež to, že je potenciálne možné za behu meniť implementáciu a znovu kompilovať jednotlivé časti kódu, prípadne uplatňovať rôzne optimalizácie až v okamihu keď je potvrdené že budú efektívne.

2 Objektové programovanie

Objektové programovanie je spôsob uvažovania o programe ako o skupine vzájomne komunikujúcich objektov. Každý objekt má pritom svoj interný stav a verejné rozhranie pomocou ktorého komunikuje s inými objektami. Objektové programovanie v podstate nie je v spore s imperatívnym ani funkcionálnym paradigmatom, iba zavádza dodatočné pravidlá ktorými sa musí programátor riadiť pri písaní kódu.

2.1 Základné pojmy

Trieda predstavuje definíciu objektov určitého typu. Trieda určuje aké atribúty (vnútorné premenné) a aké metódy (funkcie vykonateľné nad objektom) bude po vytvorení objekt mať. Trieda taktiež môže definovať špeciálne metódy konštruktor a deštruktor určené na vytvorenie/zničenie objektu danej triedy.

Objekt je inštancia triedy ktorá vznikla použitím konštruktora. Projekt obsahuje atributy a metódy definované triedou.

2.2 Zapúzdrenie

Zapúzdrenie je mechanizmus ktorý dovoľuje pre každý z prvkov danej triedy (atribúty/metódy) špecifikovať viditeľnosť, teda skupiny objektov, ktoré môžu s daným prvkom pracovať. Štandardne ide buď o privátne prvky (má k nim prístup len sám objekt), chránené prvky (má k nim prístup objekt a jeho potomkovia) a verejné prvky (majú k nim prístup všetky objekty). Princíp zapúzdrenia následne tieto pravidlá vynucuje naprieč celým programom.

Zapúzdrenie sa využíva na skrytie informácií ktoré sú buď implementačne závislé alebo nesmú byť menené inými objektami. Tým sa dá dosiahnuť dobrá izolácia rozhrania objektu a implementácie objektu a nie je problém zamieňať implementácie pri zachovaní rozhrania.

Pri zapúzdrení sa taktiež odporúča dodržiavať princíp zachovania najnižšej možnej viditeľnosti. Ten hovorí, že prvky by mali byť viditeľné skutočne len pre tie objekty, ktoré s nimi budú pracovať.

2.3 Dedičnosť

Dedičnosť je mechanizmus ktorý umožňuje vyjadriť hierarchickú štruktúru medzi objektami (generalizácia-špecifikácia) a zároveň umožňuje zdieľať kód ktorý je pre takéto objekty spoločný. Trieda môže dediť z jednej alebo viacerých(diamond problem) iných tried, pričom týmto spôsobom získa všetky vlastnosti rodičovských tried.

Trieda môže prekryť vlastnosti rodičovských tried vlastnou implementáciu (pozor na virtuálne metódy v C++). V takom prípade ale musí platiť princíp substitúcie. Teda že všade, kde je možné použiť rodičovskú triedu je možné použiť aj potomka. Toto zaisťuje jednak kompilátor, ktorý nedovolí odstrániť už existujúce metódy, druhak programátor, ktorý musí zabezpečiť že prekryté vlastnosti nekolidujú s už použitým kódom.

2.4 Polymorfizmus

Potomkovia môžu vystupovať miesto rodičov.

2.5 Udalosťami riadené programovanie

Program obsahuje hlavnú slučku ktorá zbiera udalosti a následne ich spracúva. Celý tok programu je riadený vyvolávaním a spracúvaním udalostí. Dobré na UI.

2.6 Výnimky

Viď. Try-Catch-Finally

3 Základné princípy počítačov

3.1 Číselné sústavy

- Nepozičné číselné sústavy - zastaralé (Rímska, Egyptská) - hodnota čísla závisí iba na počte znakov.
- Pozičné číselné sústavy - "váha"znaku je ovplyvnená jeho pozíciou v zápise čísla.
- Polyadické sústavy - špeciálny prípad pozičných - váha sa počíta ako polynóm

Základ - Počet znakov v sústave
Rád - váha číslice
TODO: Sem dať zápis číslice ako polynóm a potom zhustene
Sústavy používané v IT:

- dvojková
- sem tam osmičková
- desiatková (duh)
- šestnástková

Prevod: Medzi základmi kedy jeden je mocnina druhého je jednoduchý. Stačí postupovať "po čísliciachä prevádzať vždy toľko "bitov"po sebe, koľko mi kóduje jedno číslo v druhej sústave.

Pokiaľ toto neplatí, musíme postupovať "po polynóme". Najistejší algoritmus: Pri prevode do vyššej sústavy si zostrojím želaný polynóm v danej sústave. Tj. napr. číslo 101(2) v sedmičkovej sústave bude $1*11+1*4+0*2+1*1 = 16$ (koeficienty polynómu už sú v sedmičkovej sústave).

Pri prevode do nižšej sústavy delím číslo so zvyškom postupne jednotlivými prvkami polynómu (postupujem samostatne pre celočíselnú a desatinnú časť). Pozor, pri prevode desatinnej časti nedelím ale násobím a sledujem pretečenie do celočíselnej časti.

3.2 Zobrazenie čísel v počítači

3.2.1 Priamy kód

Kladné čísla sa dajú zobrazovať klasicky. Rozsah $< 0, 2^N - 1 >$.

Záporné čísla potrebujú znamienkový bit. Rozsah $< -2^{N-1} + 1, -0 >$ a $< 0, 2^{N-1} - 1 >$. Obsahuje dvojitú nulu (neplatí že -0 + 1 = +0). Pri pretečení pokračujem do zápornej nuly.

3.2.2 Inverzný kód

Záporné čísla zobrazujem inverzne (prvý bit držím ako znamienkový). Rovnaký rozsah ako priamy kód, ale pri pretečení sa dostanem na -MAX, nie -0. Taktiež platí že -0 + 1 = +0.

Problém dvoch núl sa pri sčítaní rieši pripočítaním prenosu zo znamienkového bitu (kruhový prenos).

3.2.3 Doplnkový kód

Dvojkový doplnkový kód = Inverzia všetkých bitov a pripočítanie jednotky. Rozsah $< -2^{N-1}, -0 >$ a $< 0, 2^{N-1} - 1 >$ (získal som o jedno záporné číslo viac). Nemám dvojitú nulu, lebo $0 = 1 + 1$. Mám pekný prechod z -1 na 0, z 0 na 1 a aj pri pretečení spadnem na -MAX.

Sčíta sa úplne normálne. Pretečenie nastáva keď sa prenos do znamienkového bitu nerovná prenosu z neho. (lebo to nastáva v okamihu keď je číslo kladné a zmení sa na záporné, ak je záporné tak sa prenos rovná)

3.2.4 Kód s posunutou nulou

Záporné čísla sa simulujú tak, že sa povie že celé to číslo je posunuté o polovicu rozsahu zobrazenia.

3.2.5 Floating point čísla (IEEE 754)

Číslo má tri zložky. Každá zložka môže mať rôznu šírku (bežne 1, 8, 23 v 32bite)

Znamienkový bit - nuda

Exponent - Zadaný v kóde s posunutou nulou.

Mantisa - Vždy je vo formáte 1.XXXX. Začiatočná jednotka je implicitná.

Výsledné číslo je teda dané ako $(ZB? - 1 : 1) * 1.MANTISA * 2^{EXPONENT}$. (Toto je efektívne prevod do desiatkovej sústavy)

Prevod do IEEE 754: Najprv si prevediem do dvojkovej, tam vyjadrím číslo cez exponent a následne zakódujem sem.

3.3 Kombinačné a sekvenčné logické obvody

Originálna idea je implementovať v obvodoch booleovu algebru. Tá vyžaduje tri operátory. Alternatívne sa preto implementuje Schefferova (NAND) alebo Pierceova (NOR) algebra ktoré vyžadujú len jednu operáciu.

Kombinačné logické obvody (logické funkcie):

- AND, OR, NEG, NAND, NOR, XOR, NOXOR

Ako ich zapísať vo forme obvodu vymyslíš na mieste, hlavne si treba pamätať že tam potrebuješ vstupy, výstup, napájanie a uzemnenie. (A nejaký ten tranzistor - typicky vstupy idú na bázu a riadia či ide prúd na výstup alebo na uzemnenie)

Z tohto sa potom skladajú sčítačky, násobičky, dekodéry a pod.

Sekvenčné logické obvody (majú pamäťovú časť ktorá si uchováva stav, môžu byť riadené vstupom alebo hodinami, typické sekvenčné súčiastky: sekvenčný/paralelný register, sekvenčná sčítačka):

3.3.1 Klopný obvod RS

Najjednoduchšia pamäť. Riadená vstupom. Má dva vstupy, set a reset. Má dva výstupy Q, a ¬Q. Ak je nastavený set, platí Q, ak je nastavený reset, platí negácia Q. Ak nie je nastavené nič, platí rovnaká hodnota ako v minulosti. Ak sú nastavné obe, ide o neplatný stav.

Obvod RS riadený nulami je ako klasický RS ale s negovanými vstupmi.

Implementácia pomocou dvoch NAND hradiel, ktorých výstup sa zároveň posiela na vstup toho opačného hradla. Druhý vstup je potom R alebo S bit. Takže ak sa nastaví S, NAND1 sa spustí, vyvolá prúd na výstupe, ten vyvolá prúd na NAND2, ten vyvolá prúd na výstupe NAND2 a ten zase vyvolá prúd na vstupe pôvodného NAND1.

Dá sa upraviť tak aby bol riadený hodinami keď sa vstup synchronizuje cez hodiny (Vstup AND Hodiny).

3.3.2 Klopný obvod D

Modifikácia RS obvodu umožňujúca riadenie hodinami a zjednodušený vstup. Má iba jeden vstup (výstupy ako RS). Pri tiku hodín sa zoberie vstup, uloží sa do D obvodu a až kým sa tam neuloží niečo iné, hodnota tam zostáva.

Implementuje sa cez RS riadený hodinami. Vstup sa privedie na S, negácia vstupu na R. Keď tiknú hodiny, AND začne platiť, vstup sa prenesie do RS obvodu. Tam zostáva až kým znovu netiknú hodiny.

3.3.3 Klopný obvod JK

Modifikácia RS ktorá eliminuje zakázaný stav. J funguje ako S, K ako R. Ak platia obe súčasne, obsah obvodu sa neguje. To je docielené tým, že na vstup RS obvodu ide J AND ¬Q a K AND Q. Tým pádom nemôžu byť oba vstupy pravdivé naraz, a pravdivý je len variant ktorý aktuálne neplatí v obvode. (Keby sa prehodilo J a K, ale nechalo Q, tak by si zachovával stav. To už ale funguje pri vstupe 0,0 takže toto je zaujímavejšie)

3.4 Von Neumannova architektúra

Komponenty: Vstup, Radič, CPU, Pamäť, Výstup.

Dva typy signálov: Riadiace a Dátové. Dáta prúdia zo vstupu do CPU, z CPU do Radiča, Pamäte a na výstup, z Pamäte do CPU. Riadiace signály idú z radiča všade inde a naspäť (všetko riadi, no).

Iný typ: Harvardská architektúra - oddelená pamäť na inštrukcie a na dáta, inak skoro rovnaká.

3.5 Princípy fungovania procesorov, Prerušenia

Radič, Cache, CISC, RICS, Pipeline, Registre, FPU, Zbernice(šírka)... boooring

Prerušenie je metóda asynchronného spracovania udalostí. Umožňuje rýchlo riešiť I/O operácie a chyby pri vykonávaní programu.

Maska prerušení - špeciálny register ktorý indikuje ktoré prerušenia sú aktuálne povolené

Tabuľka prerušení - obsahuje informáciu aký kód sa má vykonať v prípade ktorého prerušenia.

Typy prerušenia: Vonkajšie (I/O), vnútorné (delenie nulou, memory fault), softwarové(debugging, výnimky)

4 Operačné systémy

Funkcie operačných systémov:

- správa procesov, operačnej pamäte, súborov, siete, sekundárnej pamäte(disk) a iného I/O, ochrana a zabezpečenie (sandboxing), interpret príkazov, reakcia na chyby, štatistiky o systéme

Typy architektúry operačných systémov:

- Monolitické jadro - jeden veľký pamäťový priestor, dobrý výkon, zložité (Linux)
- Mikrojadro - minimum funkcií priamo v jadre, pomalé (Mach, Symbian)
- Hybridné - trochu z oboch (Windows, OS X)

Komunikácia s OS funguje na základe tzv. systémových volaní. Tie sa typicky implementujú pomocou prerušení. Aplikácia vyvolá softwarové prerušenie, operačný systém si prečíta čo aplikácia potrebuje a vybaví to. (Prerušenie zabezpečí zmenu privilégií, etc.)

4.1 Procesy, vlákna, synchronizácia

Proces - aktívne bežiaci program. Má dátový, inštrukčný a zásobníkový segment pamäti.

Hierarchické usporiadanie procesov (originálny rodič je prvý proces 0)

Stavy procesu:

- New, Ready(môže sa začať vykonávať), Running, Blocked(čaká na niečo, I/O napr.), Termianted

Informácie o procese (Process Control Block - PCB): PID, PID rodiča, ID uživateľa, stav, info o pamäti a súboroch, štatistiky, informácie pre plánovač.

Prepnutie medzi procesmi procesmi = context switch.

Vlákno - jedna vykonateľná sekvencia inštrukcií v rámci procesu. Má vlastný zásobník a program counter, registre a context, ale zdieľa dátový a inštrukčný priestor a iné zdroje. Vlákno je de facto low-cost náhrada za proces vyžadujúca menšiu réžiu.

Vlákna môžu byť implementované buď na užívateľskej úrovni (OS o nich nevie = nenaplánuje ich na rôzne procesory) alebo na úrovni OS (discount process scheduling).

4.1.1 Synchronizácia

Miesto v kóde ktoré v istom časovom okamihu vykonáva iba jeden proces sa nazýva kritická sekcia. Kritická sekcia musí zaručiť: Safety (max. jeden proces naraz), Progress (ak je sekcia prázdna, nikto nečaká do nekonečna), Fairness (žiadny proces nečaká nekončne dlho).

Realizácia: Čisto software (busy waiting, problém optimalizácie a out of order vykonávania), Hardware (CAS, Test-And-Set, stále busy waiting, ale lepšie riadené), OS (synchronizuje OS s tým že OS si môže riadiť scheduler tak aby minimalizoval busy wait).

Synchronizačné primitíva: Mutex (lock, unlock), Semafór (give, take, init), Monitor (kritická sekcia nad zámkom, prípadne s podmienkami), Podmienkové premenné (wait, notify).

4.1.2 Uviaznutie (Deadlock)

Stav programu z ktorého nie je možné pokračovať. (Alternatíva livelock - program pracuje, ale nikam sa neposúva).

Podmienky vzniku uviaznutia pri prístupu k zdrojom: Vzájomné vylúčenie (každý zdroj môže mať naraz len jeden proces), Ponechanie si zdroja a čakanie (Proces čo chce viac zdrojov si ich postupne berie a čaká na ďalšie), Absencia predbiehania (Zdroj sa uvoľní iba ak sa ho proces vzdá), Kruhové čakanie.

Ochrana pred uviaznutím: Prevencia (nejaká forma verifikácie programu, nemusí byť úplne formálna), Obchádzanie (zdetekujem potenciálne uviaznutie a riešim to napr. tak že zdroj nepridelím), Detekcia (zdetekujem uviaznutie keď nastane a napr. zabijem jeden zo zaseknutých procesov), Ignorancia.

Prevencia uviaznutia:

- Virtualizujem zdroje, takže každý zdroj môže používať viac procesov naraz a uviaznuť teda môže len OS, ktorý si to už nejak zariadi.
- Zakážem postupné získavanie zdrojov - buď sa zamkne naraz všetko, alebo nič
- Alternatíva k minulému - v prípade že ďalší požadovaný zdroj nejde zamknúť, uvoľním všetko
- Zavediem totálne usporiadanie na zdrojoch

Obchádzanie a detekcia: OS má informáciu o hornom limite na zdroje ktoré môže proces požadovať (niekedy ťažko určite). Následne OS testuje či aktuálne dostupné, aktuálne rozdané a potenciálne požadované zdroje môžu vytvoriť cyklický graf. Takto sa dá vyriešiť aj prevencia, lebo detekcia môže prebehnúť napr. ešte pred spustením problémového procesu. Alternatívne sa môžeme vykašlať na informáciu o maximálnom počte zdrojov a uvažovať iba aktuálne pridelené zdroje a z toho získať aktuálne čakajúce procesy. Z tohto dostaneme graf čakania a ak je v ňom cyklus, máme problém. Nevýhoda je že takto detekujeme len existujúce uviaznutie, nie potenciálne. Riešenie je potom buď prerozdelenie zdrojov alebo zabitie procesu.

4.2 Plánovanie procesov

OS udržuje niekoľko front procesov: Pripravené procesy, Čakajúce na I/O, Čakajúce na pamäť (swappované), Čakajúce na synchronizáciu. Toto všetko ešte môže byť obohatené prioritami, privilégiami a pod. V podstate iba prvá fronta vyžaduje seriózne plánovanie, keďže zvyšné sa uvoľňujú keď sú dostupné potrebné zdroje.

OS má nejakú internú hierarchiu plánovačov. Typicky ide o krátkodobý (ktorý pripravený proces sa pustí, milisekundy), strednedobý (ktorý proces sa odswappuje) a dlhodobý plánovač (ktorý proces sa spustí kedy, rádovo sekundy). Krátkodobý plánovač sa okrem pravidelných intervalov štandardne vyvoláva v okamihu keď nejaký proces zaspáva alebo začína čakať, prípadne až nejaký proces prestáva čakať a je znovu pripravený. Pokiaľ má plánovač moc proces pozastaviť (bežiaci-¿pripravený), hovoríme o preemptívnom plánovaní (predbiehanie).

Pri plánovaní treba optimalizovať množstvo kritérií: Využitie CPU, dobu vykonávania procesu, dobu čakania, dobu odozvy, prioritu, spravodlivosť...

Plánovacie algoritmy:

- First Come, First Serve - klasické nudné FIFO, bez predbiehania. Problém krátkych vs. dlhých procesov.
- Round Robin - Po časovom kvante sa prepína proces. Procesy sa rovnomerne striedajú. Problém u procesov ktoré často čakajú (zbytočné si minú časové kvantum)
- Prioritné plánovanie - FIFO, ale každá priorita má svoju frontu. Preemptívny variant umožňuje novému procesu s vyššou prioritou zastaviť bežiace nižšie procesy.
- Shortest Job First - Každý proces má časový odhad na vykonávanie (ak nie je známy, počíta sa štatistika histórie), vyberá sa proces s najkratším potenciálnym časom vykonávania. Preemptívny variant potom predbehne bežiaci proces ak sa objaví proces ktorý má zostávajúci čas menší ako bežiaci proces.

5 Práca s pamäťou, Vstupy a Výstupy

5.1 Hierarchia pamätí

Register > Vnútorná (RAM, Cache) > Vonkajšia pamäť

5.2 Vlastnosti pamätí

- Kapacita
- Prístupová doba (latenica)
- Priepustnosť (bandwidth)
- Statickosť / Dynamickosť - Dynamická pamäť musí informáciu dynamicky obnovovať
- Deštruktívnosť čítania
- Volatilita
- Prístup - sekvenčný, priamy
- Spoľahlivosť, Cena, etc...

5.2.1 Vnútorná pamäť

RAM - random access memory - dá sa pristupovať priamo k jednotlivým bunkám. Dočasné ukladanie - len za behu počítača.

Matica pamäťových buniek. Čítanie/Zápis - nastaví sa riadiaci vodič(vodiče) pre danú bunku, to vyvolá čítanie/zápis (alternatívne čítanie po riadkoch/stĺpcoch).

Možná realizácia:

- ROM - obsahuje informáciu od výroby
- PROM - ako ROM, ale dá sa do nej raz zapísať (prepálenie poistky)
- EPROM - ako PROM, ale dá sa zmazať UV svetlom
- EEPROM - ako EPROM, ale dá sa zmazať aj elektricky
- Flash - dá sa mazať po kúskoch. Obsahuje bunky, každá bunka má potom Control a Floating gate.
- SRAM - statická RAM (L1-L3 Cache), drahé ale rýchle, podobné ako klopné ER obvody
- DRAM - dynamická RAM, realizovaná ako kondenzátor, vyžaduje obnovovanie

5.2.2 Vonkajšia pamäť

Nie je dostupná priamo z procesora, dáta sa kopírujú do RAM radičom. Slúži na trvalé ukladanie informácií.

- Pevný disk - adresácia cez cylinder, hlavu a sektor prípadne cez lineárne očíslovanie sektorov (preklad je na disku).
- Disketa, CD, USB Flash, Network storage

5.3 Práca s vnútornou pamäťou

Zaisťuje pre jednotlivé procesy ilúziu neobmedzeného súvislého pamäťového priestoru začínajúceho na nule pomocou logických adries. Tým sa eliminuje nutnosť používať fyzické adresy pri preklade.

5.3.1 Segmentácia

Rieši problém zabezpečenia a problém relokácie pamäte. Pamäťová bunka je určená adresou začiatku segmentu a offsetom v rámci segmentu. Každý segment môže byť nastavený ako read only, prípadne môže mať rôzne privilégia (typicky 4 rôzne okruhy zabezpečenia). Teda je možné presunúť proces v pamäti na nové miesto bez toho aby o tom vedel (zmení sa bázový register). Zároveň je možné vynútiť zabezpečenie - proces nemôže pristupovať za hranice segmentu a do segmentov iných procesov (pokiaľ nemá povolenie).

Potrebuje tabuľku segmentov ktorá udáva limit, typ a privilégia segmentu.

Jediný kto ho vlastne poriadne používal bol Intel v x86, dnes sa už nepoužíva (v x64 je vypnuté), ochrana pamäte a relokácia bola prenesená do stránkovania, čiastočne s podporou operačného systému.

5.3.2 Stránkovanie

Primárne rieši fragmentáciu pamäte a virtuálnu pamäť. Pamäť sa delí na malé kúsky, stránky, ktoré v podstate fungujú ako malinké segmenty. Proces má logickú adresu podľa ktorej sa v tabuľke stránok nájde správna fyzická adresa (prípadne sa zistí že stránka je odswapovaná - page fault). Tabuľka stránok je kvôli efektivite hierarchická a často hľadané stránky sú uložené v Translation Lookaside Buffer. Stránky môžu byť v pamäti usporiadané ľubovoľne. Pre každú stránku je potom možné nastavovať privilégiá a vlastníka, čím sa zaručí bezpečnosť prístupu. Stránky je možné swapovať na disk.

5.4 Práca s I/O

- Polling, prerušenie, DMA.
- Blokujúce/neblokujúce volania.
- OS: caching, buffering, fronty požiadavkov, rezervácia

6 Databáze