Fakulta Informatiky

Masarykova Universita v Brně

Vybrané partie z informatiky

Přednášky: prof. RNDr. Jaroslav Král, DrSc.

Zpracoval: David Krásenský

Brno, duben 1995

Obsah

1 Paralelní výpočty a počítače **3**
1.1 Úvod do paralelismu . . . 3
1.1.1 Co je to paralelismus . . . 3
1.1.2 Vektorové počítače . . . 4
1.1.3 Aplikace superpočítačů . . . 4
1.2 MIMD počítače . . . 5
1.2.1 Symetrický multiprocesing . . . 5
1.2.2 Masivně paralelní architektury . . . 5
1.2.3 Transputer firmy INMOS Ltd. . . . 6
1.3 Systémy výměny zpráv . . . 8
1.3.1 Systémy reálného času . . . 8
1.3.2 Práce simulátoru . . . 9
1.3.3 Dekompozice . . . 10
1.3.4 Izolace hard real time částí . . . 10
1.4 Jazyk pro vektorové počítače – Fortran 90 . . . 10
1.4.1 Vývoj Fortranu . . . 11
1.4.2 Práce s poli ve Fortranu 90 . . . 11
1.4.3 Datové struktury . . . 11
1.4.4 Vrstvy . . . 12
1.4.5 Deskriptory násobných hodnot . . . 12
1.5 Jazyk OCCAM 1 . . . 13
1.5.1 Základní rysy OCCAMu 1 . . . 13
1.5.2 Příklady programování v jazyce OCCAM 1 . . . 14
1.5.3 Alternující procesy . . . 15
1.5.4 Pole procesů . . . 17
1.5.5 Vazby na hardware . . . 18
1.6 Ada . . . 19
1.6.1 Vazby na hardware . . . 19
1.6.2 Modernizace Ady – Ada 9x . . . 20
1.7 Mikrojádro . . . 23
1.7.1 Princip mikrojádra . . . 24
1.7.2 Základní rysy systému s mikrojádrem . . . 24
1.7.3 Komunikace . . . 26
1.7.4 Rozhraní na Mach z C . . . 27
1.8 Shrnutí . . . 27

2 Softwarové inženýrství **28**
2.1 Tvorba rozsáhlých systémů . . . 28
2.1.1 Současná situace . . . 28
2.1.2 Customizace . . . 28
2.1.3 Nebezpečí organizačních změn . . . 29
2.1.4 Interview . . . 29
2.2 Ekonomické aspekty . . . 30
2.2.1 Náklady na software . . . 30
2.2.2 Analýza přínosů . . . 31
2.2.3 Analýza rizik . . . 31
2.2.4 Odhady . . . 32
2.3 Sledování kvality . . . 32
2.3.1 Inspekce . . . 32
2.3.2 Zkušenosti s inspekcemi . . . 33
2.3.3 Vícefázové inspekce . . . 35
2.4 Používání norem . . . 36
2.4.1 Vlastnosti norem a zásady jejich používání . . . 36

2.4.2 Normotvorné aktivity . . . 37
2.5 Normy v softwarovém inženýrství . . . 38
2.5.1 Význam norem při tvorbě softwaru . . . 38
2.5.2 Plán zajištění kvality . . . 39
2.5.3 Řízení konfigurace . . . 40
2.5.4 Testování . . . 41

3 Pel–mel 42
3.1 Architektura klient/server v databázích . . . 42
3.2 Objektově orientovaná analýza a návrh . . . 42
3.2.1 Principy OOMD . . . 43
3.2.2 Z čeho modelování vychází . . . 43
3.2.3 Definice n-árních vztahů . . . 43
3.2.4 Atributy relací . . . 44
3.2.5 Implicitní násobnost relace . . . 44
3.2.6 Co všechno lze specifikovat u relace . . . 44
3.2.7 Vyjádření dědění . . . 45
3.2.8 Rekurzivnost tříd . . . 45
3.2.9 Abstrakce a abstraktní třídy . . . 45
3.2.10 Objektový a dynamický model . . . 45
3.3 Superpočítače . . . 47
3.3.1 Architektury superpočítačů . . . 47
3.3.2 Rozšířitelnost . . . 49
3.3.3 Paměť typu cache . . . 50

Seznam obrázků

1 Hyperkrychle stupňů $n = 0, 1, 2, 3, 4$ . . . 6
2 Adresování vrcholů hyperkrychle . . . 6
3 Třídění na architektuře INMOS . . . 7
4 Řízení systému reálného času . . . 8
5 Napojení řídícího systému na simulátor . . . 9
6 Historický vývoj využívání počítačů . . . 10
7 OCCAM: proces realizující druhou mocninu . . . 14
8 OCCAM: síť procesů pro čtvrtou mocninu . . . 14
9 Řízení zesilovače . . . 16
10 Pole procesů pro iteraci druhé odmocniny . . . 17
11 OCCAM: definované propojení procesorů . . . 18
12 Přepojovací pole . . . 19
13 Příkazový blok disku . . . 19
14 Rozhraní systémů s mikrojádrem . . . 24
15 Mapování zpráv . . . 27
16 Předávání zpráv pomocí síťového serveru . . . 27
17 Srovnání detekce chyb klasickým testováním a inspekcí . . . 34
18 Kombinace klasického testování a inspekcí . . . 35
19 Příklad modelování tříd . . . 43
20 Typy vazeb mezi třídami . . . 43
21 Znázornění instancí . . . 44
22 Model n-ární relace . . . 44
23 Instance s vytvořenou n-ární relací . . . 45
24 Atributy relace . . . 45
25 Příklad relací s atributy – zaměstnanci . . . 46
26 Příklad ternární relace . . . 46
27 Správné a nesprávné použití atributů . . . 46

28 Použití značky agregace 47
29 Vyjádření dědění 47
30 Rekurzivnost tříd 48
31 Abstraktní třídy 48
32 Definice stavu 49
33 Přechod ve stavovém diagramu 49
34 Příklad: regulátor topení 50
35 Organizace paměti v systémech MIMD 50

1 Paralelní výpočty a počítače

1.1 Úvod do paralelismu

1.1.1 Co je to paralelismus

Víme, co znamená slovo paralelismus: programy (procesy) jsou spouštěny tak, že běží navenek současně, souběžně. Rozlišujeme dva typy:

fyzický paralelismus – počítač má skutečně více procesorů,

pseudoparalelismus – programový paralelismus, počítač má pouze jeden procesor, běh procesů se přepíná.

V moderních programovacích jazycích je obvyklé, že pseudoparalelismus může, ale nemusí využívat paralelismus fyzický.

Pro využití paralelních výpočtů máme v zásadě tyto důvody:

- aby to počítalo rychleji,
- aby se toho spočítalo více (více procesorů = více práce)[1]

Zopakujeme si nyní práci klasického počítače. Počítač má jednu sběrnici společnou pro instrukce i data. Budeme sledovat fáze, které se opakují, a jejich časovou náročnost:

1. Přečti instrukci. Probíhá ve čtyřech subfázích:
   - Odeslání adresy – jde po sběrnici.
   - Dekódování adresy (v paměti).
   - Vyzvednutí instrukce (z paměti).
   - Poslání instrukce – opět jde po sběrnici.

   Vidíme, že doba čtení instrukce lze vyjádřit jako $2 \cdot t_{sb} + c_1 + c_2$, kde t_{sb} je čas přenosu signálu po sběrnici (je omezen rychlostí světla), konstanty c_1, c_2 vyjadřují dekódování adresy a čtení instrukce; jsou to parametry paměti.

2. Dekóduj instrukci. Probíhá v procesoru v konstantním čase.

3. Přečti data. Fáze je analogická čtení instrukce, dospějeme ke stejnému času provedení.

4. Proveď instrukci. Záleží na procesoru, instrukce mohou trvat různě dlouhou dobu.

Vidíme, že vzhledem k omezení šíření signálu po sběrnici rychlostí světla nemůžeme zvyšovat výkon počítače do nekonečna. Pro zrychlení je několik možností:

1. Menší vzdálenosti (kratší sběrnice). Programově je ovlivnit nemůžeme; zřejmě i zde jsou určité hranice technických možností.

2. Zrychlení paměti. To je v zásadě možné, ale poměrně dosti drahé. Nabízí se použití

[1] Známé heslo: Republice více práce, to je naše agitace.

- něčeho v procesoru a velmi rychlého – buďto registry (zvlášť rychlé!), anebo primární cache (přímo integrována do procesoru)
- nebo něčeho blízko procesoru a rychlého – sekundární cache.

3. Proudové zpracování (pipelining). Provedení instrukce se rozloží na etapy, které se provádějí souběžně. (Stále se však jedná o „klasické" schéma programování.) Z hlediska dosaženého zrychlení je velice efektivní.
4. Superskalární architektura – více výkonných jednotek ALU. To je „téměř klasické" programování, objevuje se zde však také již skutečný paralelismus.
5. Vektorové počítače – SIMD (Simple Instruction, Multiple Data).

Vektorovými počítači se nyní budeme zabývat podrobněji.

1.1.2 Vektorové počítače

Principem vektorových počítačů je, že jsou definovány operace pro n-tice (kde $n \approx 100$). Přitom n-tice jsou uloženy v rychlých registrech, operace se provádí ve více výkonných jednotkách. Kromě toho stále existuje klasický SISD[2] režim, tedy může pracovat jako klasický počítač.

Hlavní charakteristikou již nejsou MIPS (milióny instrukcí za sekundu), ale MFLOPS – milióny operací v pohyblivé řádové čárce.[3] (Současné špičky výkonu jsou řádově $\approx 10^3 - 10^4$ MFLOPS.) Přitom MFLOPS je jen indikátorem výkonnosti: skutečný výkon závisí na rozměru úlohy – a zejména na tom, jak rozměr úlohy využije vektorovou architekturu. Proto se obvykle uvádí údaj $n_{1/2}$, což je rozměr úlohy, při kterém dosáhne počítač poloviny teoretického výkonu.

Ukázkou programování vektorových počítačů je Fortran 90, o kterém se zmíníme později. Fortran 90 pracuje s celými poli, existují jeho překladače specielně pro vektorové počítače. Některé jsou dokonce schopny analyzovat klasický Fortran a „vyrobit" z něj program využívající maticové operace.

Typickým výrobcem vektorových počítačů je CRAY – tento počítač je určen zejména pro vědeckotechnické výpočty (koupil si jej např. Walt Disney pro zpracování obrázků, kde se vyskytuje spousta výpočtů).

Stále se však pohybujeme na poli „klasického" programování, kdy se programátor nestará o to, jak to paralelně běhá.

Pro typické databázové systémy to příliš vhodné není.

1.1.3 Aplikace superpočítačů

Nejčastějšími aplikacemi jsou zejména:

Fyzika. Náročné jsou např. numerické výpočty z oblasti hydrodynamiky.

Chemie. Jedná se o kvantovou chemii a o výpočty s molekulami.

Parciální diferenciální rovnice. Opět numerické metody, např. metoda konečných elementů.

Simulace. Simulační programy se uplatňují např. v oblastech:

- válečné hry,
- ekologické katastrofy,
- výpočty z ekologie (chování ekosystémů),
- astronomie – typicky vývoj galaxií,
- počasí (opět parciální diferenciální rovnice).

Animace. Využití v zábavním průmyslu.

Virtuální realita. Zatím je spíše ve stádiu pokusů.

[2]Simple Instruction, Simple Data.
[3]Jedná se o počet operací těch jednotlivých jednotek.

Připomínám, že databáze *nejsou* vhodnou oblastí pro aplikace.

Vektorové procesory jsou vhodné i pro úlohy s malou zrnitostí (granularitou), přičemž úloha s velkou zrnitostí je taková, kterou lze rozdělit na části (relativně velké), které se dají řešit odděleně. Tak např. úlohy s velkou granularitou jsou např. SQL databázové operace, oproti tomu malou granularitu vykazuje např. úloha inverze matice.

U vektorových procesorů se výkon zvyšuje dráže než u superskalárních architektur.

1.2 MIMD počítače

Do této skupiny patří dva typy architektur:

- víceprocesorové systémy – typicky 4 – 128 procesorů. Jedná se o tzv. *symetrický multiprocesing*, kdy všechny procesy mají stejná práva.
- masivně paralelní architektury – mají až $\approx 10^5$ procesorů. Mezi ně patří i *transputery*, na něž se soustřeďuje výzkum; některé z nich napodobují neuronové sítě.

Probereme nyní obě skupiny podrobněji.

1.2.1 Symetrický multiprocesing

má dvě větve:

Úzce vázaný. Představuje jej technika tzv. mikrojádra; znamená to, že paralelitu nahrazuje operační systém (je to zvládnutelné v UNIXu).[4]

Volně vázaný. Tedy počítače spojené do sítě (počítače přitom mohou mít vevnitř procesory); velice záleží na síti a na zrnitosti úlohy. Procesory přitom spolupracují pomocí výměny zpráv.

Symetrický multiprocesing přitom funguje typicky tak, že když např. k síti čtyř procesorů přidáme dalších dvanáct, systém funguje se šestnácti procesory; pracuje vždy procesor, který je momentálně volný, úlohu si „odněkud“ (z bufferu) vyzvedne.

Toto u masivně paralelních architektur možné není – nelze totiž napojit řádově desetisíce procesorů na společnou sběrnici.

1.2.2 Masivně paralelní architektury

Tato architektura je dána těmito atributy:

- asynchronní činnost,
- výměna zpráv je závislá na *grafu propojení*,
- graf propojení může být přitom pevný, nebo nastavitelný.

Doporučený postup při programování takového systému je zhruba následující:

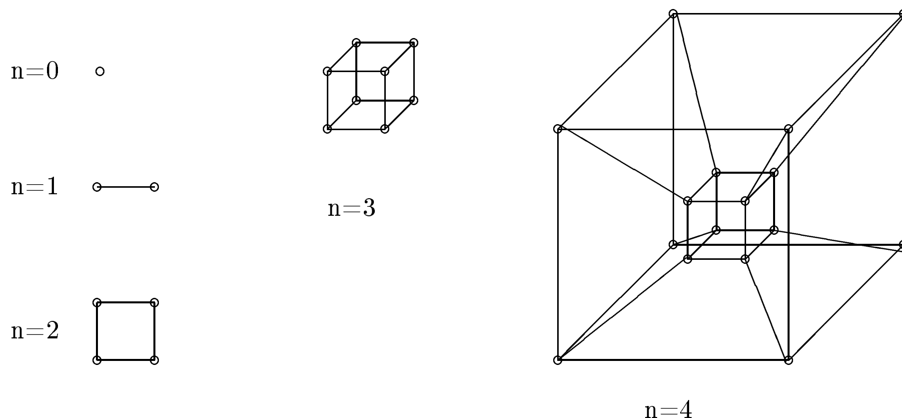
1. Navrhne se graf propojení.
2. Navrhnou se algoritmy pro jednotlivé procesory a zprávy, pomocí kterých budou komunikovat.
3. Realizuje se graf propojení (je-li měnitelný), nebo se zobrazí do grafu, který je k dispozici.

Nejčastější propojení je *hyperkrychle*. Přitom hyperkrychle stupně n je graf vytvořený na 2^n uzlech takto: pro $n = 0$ je hyperkrychle izolovaný uzel, a hyperkrychli vyššího řádu vytvoříme ze dvou hyperkrychlí stupně o jedna menší propojením stejnolehlých uzlů. Příklady hyperkrychlí pro malá n vidíme na obrázku 1.

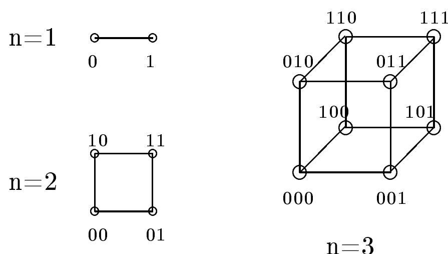
Diametr hyperkrychle stupně n, tedy největší vzdálenost mezi uzly, je roven n; stupeň všech uzlů je také n.

Adresování hyperkrychle se zavádí takto (viz obrázek 2): uzly hyperkrychle stupně $n = 1$ adresujeme jako 0,1; při konstukci hyperkrychle o řád vyšší zavedeme další binární číslici. Sousedé se pak liší v právě jednom bitu; při cestě

[4] Mikrojádro je systém primitivních operací, pomocí nichž lze naprogramovat operační systém. Používá ho např. operační systém Mach, dále Next, s výhradami Linux.



Obrázek 1: Hyperkrychle stupňů $n = 0, 1, 2, 3, 4$



Obrázek 2: Adresování vrcholů hyperkrychle

z uzlu do jiného lze nejlépe postupovat tak, že se přiblížíme (posuneme) do uzlu, jehož Hammingova vzdálenost (počet rozdílných bitů) je menší (čili: soused, jehož adresa se v jednom bitu liší).

Největší komerčně použitá hyperkrychle byla H_{16}, což představuje 65536 procesorů; jednalo se o použití v databázi Oracle.

Jiná možná architektura je propojení do dvojitého stromu: stupeň vnitřních uzlů je pak roven 6, diametr grafu je $[\log_2 n]$, kde n je počet listů.

Masivně paralelní architektury se používají zejména v databázích.

Problémy programování. Zřejmou nevýhodou je zejména to, že každý algoritmus představuje zvláštní trik; zobrazování musíme provádět na architekturu, která je k dispozici; proto se programování potýká s obtížemi.

Tyto architektury jsou však vhodné pro standardní, masově používané algoritmy vyšší zrnitosti, jako např. některé úlohy lineární algebry, a rovněž databázové (SQL) operace.[5]

1.2.3 Transputer firmy INMOS Ltd.

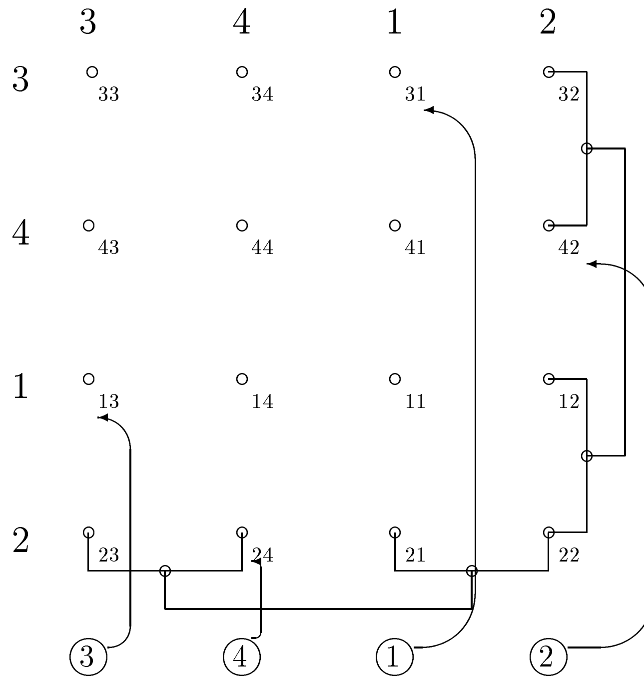
je RISC-ovský procesor se čtyřmi kanály a s vazbou na společnou paměť. Výkon se pohybuje okolo 4 MFLOPS, má k dispozici 4 KB vlastní paměti.

Programuje se v jazyce OCCAM (viz dále), důležité jsou příkazy `kanál?x` pro vstup, a `kanál!x` pro výstup.

Příklad 1.1 TŘÍDĚNÍ n ČÍSEL NA n^2 PROCESORECH

[5] Zejména nalezneme uplatnění u databází s řádově milióny a desítkami miliónů záznamů, jako tomu je u bankovních transakcí, a také např. v grafických informačních systémech, kdy mapa je uložená vektorově.

1. Rozešlu tříděná čísla po řádcích a sloupcích.
2. Pomocí sloupcového stromu zjistím počet případů, kdy je řádkové číslo menší nebo rovno sloupcovému.
3. Pokud je výsledek „hlasování“ j-tého sloupce roven k, pošle se sloupcové číslo z k-tého procesoru po stromu vpravo (jako j-té setříděné číslo).



Obrázek 3: Třídění na architektuře INMOS

Názorně nám činnost algoritmu ukazuje obrázek 3: jsou zde vyznačena sloupcová a řádková čísla; šipky ukazují závěrečné hlasování (čísla v krožku) a výsledek (řádkové číslo u konce šipky). Jsou zde též naznačena spojení procesorů v řádcích a sloupcích.

Časová složitost tohoto algoritmu na uvedené síti je přitom lineární.

Příklad 1.2 Gaussova eliminace

Jedná se o známou úlohu – vyřešit soustavu rovnic $A \cdot \overline{x} = \overline{b}$. Provádíme postupně transformaci na trojúhelníkovou matici tak, že postupně eliminujeme sloupce pod diagonálou. Postup v jednom kroku je pak tento:

1. Najdu $max(a'_{ij}), i, j \geq m$ a označíme jej a'_{kl}.
2. Přehodím rádky a sloupce tak, aby se a'_{kl} dostalo na pozici (m, m).[6]
3. Eliminační (m-tý) řádek dělím diagonálním prvkem.
4. Ode všech řádků a_k, $k > m$, odečtu m-tý řádek vynásobený a_{km}.

Gaussova eliminace má malou granularitu, a proto se na síti špatně počítá. Pokusíme se ji implementovat na $k \cdot n^2$ procesorech, a to opět propojením řádkových (sloupcových) stromů. Složitost (která je při běžném výpočtu rovna n^3) je zde:

1. Nalezení hlavního prvku: „nahoru“ se posílá (a_{ij}, i, j), „dolů“ pak maximální prvek, složitost je $\log_2 n$.

[6]Tyto dva kroky se provádějí z důvodu numerické stability – jinak by byl algoritmus díky zaokrouhlovacím chybám nepoužitelný.

2. Přesun hlavního prvku (všichni vědí, kam): $\log_2 n$.

3. Vydělení m-tého řádku: konstanta.

4. Eliminace: poslání a_{mj} po sloupci, poslání a_{im} po řádku – obojí $\log_2 n$ (pracuje souběžně); modifikace v konstantním čase.

Jeden krok tedy proběhne v čase $n \cdot \log_2 n$, celý výpočet (n kroků) pak v čase $n \cdot \log_2 n$.

Masivně paralelní architektury se používají při některých výpočtech některých typů – např. počasí, zpracování signálů (Fourierovy řady), databáze (velká granularita).

1.3 Systémy výměny zpráv

V databázových systémech komunikuje databázový stroj s různým okolím. V důsledku toho na úrovni SQL nelze řadu chyb odhalit. Na druhé straně se díky tomu, že databázový stroj je univerzální, snadno realizuje např. klient–server apod.

Znamená to, že jednotlivé komponenty systému se mohou chovat různě, ale musí posílat stejné zprávy.

Realizace je taková, že procesory posílají zprávy do bufferu (fronty zpráv), a opět podle potřeby si je zde vyzvednou. Evidentně lze bez problémů přidat další procesor, a proto je tato architektura vhodná pro úlohy s velkou granularitou.

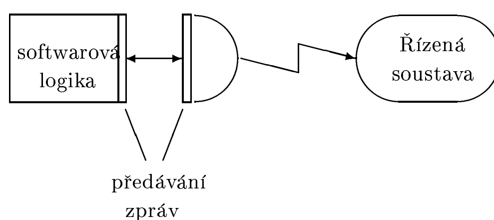
1.3.1 Systémy reálného času

Pro systémy, pracující v reálném čase, je podstatný požadavek, aby odpověděl na podnět „včas“. Takové systémy jsou v zásadě dvojího typu:

- „včas v průměru“ – není osudná chyba, když je někdy delší odezva – např. terminály;
- „do x msec“ – vždy je nezbytná odezva do uplynutí daného intervalu – jedná se o řízení procesů, jako např. NC stroje, letadla, jaderné elektrárny.

Úkolem je odladit např. systém pro řízení válcovny, ale bez skutečné válcovny (skutečného řízeného systému – málokdo vám půjčí válcovnu na hraní), a to tak, aby systém pracoval správně, včetně reakcí na havarijní situace, které se mohou v provozu vyskytnout. Je tedy třeba mimo jiné otestovat doby odezvy, průběžně generovat protokol („černá skříňka“).

Mezi řídícím systémem (který programujeme) a řízeným systémem (válcovna) musí být přitom definováno komunikační rozhraní s driverem; vlastní softwarová logika řídícího systému předává zprávy driveru, a ten je posílá systému řízenému. (Viz obrázek 4.) Drivery jsou přitom součástí zvláštního programu; komponenty pak běží v režimu „peer to

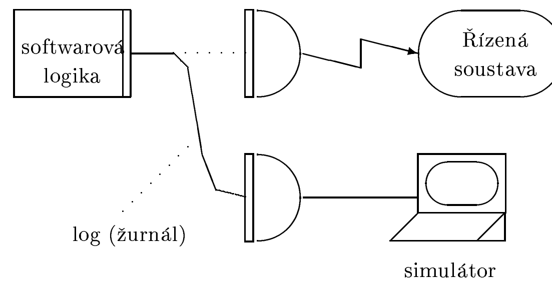


Obrázek 4: Řízení systému reálného času

peer“.

Pokud systém ladíme, musíme přesměrovat tok zpráv do simulátoru, místo do reálného systému (viz schéma na obr. 5). Činnost reálného systému tak nahradíme simulátorem; náhradu nám přitom zprostředkují výměny zpráv – stačí, aby simulátor reagoval na zprávy stejně jako reálný systém.

Tuto filosofii také můžeme uplatnit při dekompozici systému: správnost jednotlivých částí pak kontrolujeme tak, že kontrolujeme zprávy, které produkují.



Obrázek 5: Napojení řídícího systému na simulátor

1.3.2 Práce simulátoru

Jádrem simulačního programu je fronta událostí (kalendář), který obsahuje:

- simulační čas – podle něj jsou události uspořádány,
- vstupní bod procedury (ukazatel),
- parametry (do jistého maximálního počtu).

Tedy: jakmile nabude čas daného okamžiku, spustí se uvedená procedura s takto definovanými parametry.[7]

Důvodem pro spuštění simulátoru může přitom být

- příchod zprávy,
- nebo „reálný čas" $t_r = t_0$, kde t_0 je čas první události.

Při zpracování příchozí zprávy přitom může dojít ke změně v kalendáři.

Vlastní práce simulátoru pak probíhá v těchto krocích:

1. Zastaví se systémové hodiny; nechť čas je roven t_r.
2. Vyhodnotí se došlé zprávy (existují-li).
3. Provedou se všechny události (t, n, par) takové, že $t \leq t_r$, a to v daném pořadí podle t.
4. Zjistí se čas první události v kalendáři, vyhodnotí se doba čekání $t_c = t_1 - t_r$, spustí se hodiny, a simulátor se suspenduje na dobu t_c.

   Pokud je přitom kalendář prázdný, klademe $t_c = \infty$.

Důležité zde je, aby simulátor měl nejvyšší prioritu.

Lze provést analýzu, na základě které ukážeme, že tento simulátor funguje tak, jako by to byla reálná soustava.

Spuštění systému se provede následovně: inicializují se všechny programy, do simulátoru se naplní události, a skočí se na bod 4.

Systémové hodiny se však narozdíl od našich předpokladů zastavit nedají, musíme tedy počítat s tím, že určitý čas běží simulátor, a proto zavedeme následujícím způsobem opravu: zapamatujeme si tedy pouze jejich hodnotu t_r, a v bodě 4 zjistím znovu jejich stav, a odečtením dostaneme čas $t - t_r$, po který pracoval simulátor. Tuto dobu si sčítám do pomocné proměnné pom, a ukládám pak $(t - pom, mesg)$.

[7] Pro pseudoparalelní zpracování by bylo třeba umožnit i pozastavení procedury.

1.3.3 Dekompozice

Řídící systém, o kterém jsme hovořili, můžeme přitom realizovat jako n programů, které komunikují pomocí zpráv. To má určité výhody:

- Každý program má poměrně úzké rozhodování, mohu tedy snadno simulovat jeho okolí (nahradím jej generátorem zpráv).
- Komponenty lze psát v různých programovacích jazycích (je třeba zajistit pouze konzistenci zpráv).
- Systém může být i distribuovaný.
- Celkově snazší realizace.

Nevýhody oproti tomu jsou:

- Administrace přenosu zpráv (režie). Je třeba tuto komunikaci zajistit; spotřebuje určitý čas, proto nemůžeme použít na zařízení, které má problémy s časem (nestíhá).
- Někdy příliš úzké – zpráva je malá.

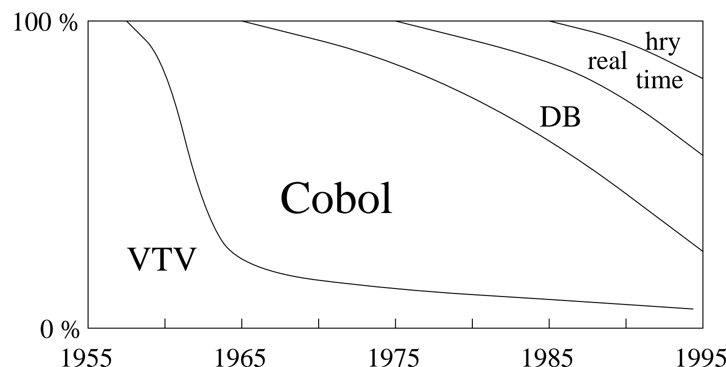
1.3.4 Izolace hard real time částí

Pod pojmem „hard real time“ zde rozumíme části, které bezprostředně a nejvíce závisí na reálném čase, tedy přímé řízení. (Např. v automatizovaném skladu, kde z databáze vybíráme údaje, na jejichž základě necháme pohybovat zakladač, je právě zakladač hard real time – musí se zastavit přesně tam, kde má.)

V systému UNIX se potřebná hard real time vlastnost nedá zajistit – je zde systémový program `cron`, který „občas“ uklízí buffery, a ten se nedá rozumně přerušit (nedá se tedy zajistit potřebná priorita).

1.4 Jazyk pro vektorové počítače – Fortran 90

Úvodem si připomeňme historický vývoj využití počítačů, jak nám jej zachycuje obrázek 6. V prvopočátcích byly



Obrázek 6: Historický vývoj využívání počítačů

počítače určeny výhradně pro vědeckotechnické výpočty (VTV). V šedesátých letech dochází k jejich masovějšímu nástupu i mimo sféru výzkumu – v oblasti hromadného zpracování dat (jazyk Cobol), což je umožněno zvýšením spolehlivosti a snížením ceny počítačů. Podíl vědeckotechnických výpočtů přitom klesá. Dále v letech sedmdesátých nastupují databáze a vytlačují dosud masově užívaný Cobol. Později přibývá využití počítačů na poli real time systémů, a posléze i fenomén posledních let – zábava, hry.

Vidíme, že Cobol postupně zcela ustupuje; podíl vědeckotechnických výpočtů – které jsou představovány jazykem Fortran, v relativním vyjádření klesá, avšak stále mají ve světě výpočetní techniky své místo.

1.4.1 Vývoj Fortranu

Jazyk Fortran[8] vznikl v roce 1956 pro usnadnění programování vědeckotechnických výpočtů. Postupně se rozšířil, a v roce 1966 se dočkal své velké normy Fortran 66 neboli Fortran IV, která byla dlouho v platnosti. V roce 1977 následoval Fortran 77, který „uzákonil“ některá používaná rozšíření – zavedl textové proměnné (typ CHARACTER), blokové IF. V osmdesátých letech se vedou spory, co dál – co se má do nové normy zakotvit. (Vědecká veřejnost je totiž konzervativní a má vůči změnám odpor, firmy nejeví o novou normu zájem, neboť trh na Fortran je poměrně malý.) Připravovaná norma je označována jako Fortran 8x, posléze vzniká norma Fortran 90 (v roce 1990).

V novém Fortranu 90 již nacházíme prvky obvyklé v ostatních moderních programovacích jazycích, jako jsou datové struktury, řídící struktury CASE a WHILE, volný formát programu (odpadá pevný význam sloupců, lze psát více příkazů na řádku).

Pokračuje standardizace, jsou nové vstupy a výstupy, existují prostředky pro superpočítače (vrstvy, operace s celými poli – viz dále).

Jako zastaralé (tj. prvky, které v příští verzi Fortranu zmizí) jsou kategorizovány: aritmetický IF, koncový příkaz společný více DO–cyklům, příkaz ASSIGN a tzv. assigned GOTO.

Pro definování přesnosti reálných čísel je vedle původního REAL a DOUBLE PRECISION (z Fortranu IV) a novějšího REAL *8 (z Fortranu 77, přesnost v byte) zaveden popis KIND(9,99) definující požadovanou přesnost mantisy a exponentu.

Všechny nové rysy jsou přitom zapracovány tak, aby bylo lze používat rysy staré a programátoři nebyli nuceni měnit své zvyklosti.

Pro zajímavost: definice normy Fortranu IV byla na 38 stranách, Fortran 77 na 250 stranách, a Fortran 90 zabírá již 360 stran...

1.4.2 Práce s poli ve Fortranu 90

Zřejmě nejsilnějším nástrojem jsou ve Fortranu 90 jsou příkazy pro zpracování polí. Přitom je zachována *ortogonalita typů*, což znamená, že nový typ je možné zcela plnohodnotně používat jako typ starý. Takže mám-li typ matice, mohu matici předávat jako parametr, a může být dokonce funkční hodnotou.

Deklarace se provádí tak, že (narozdíl od starších verzí Fortranu) uvedeme všechny atributy pohromadě:

```
REAL,DIMENSION(0:10),SAVE::A,B,C
```

narozdíl od dřívějšího[9]

```
REAL A,B,C
DIMENSION A(0:10),B(0:10),C(0:10)
SAVE A,B,C
```

Jsou možné obecné meze (dolní mez různá od jedné) a libovolné dimenze (dříve jen trojrozměrná pole).[10] Z matic je možné dělat nespojité výřezy – sloupce, řádky.

Existují rovněž *konstanty* typu pole – lze pak psát

```
(/1,2,3,4/)         =(/I,I=1,4/)
(/1.1,1.2,1.3,1.4/)=(/I*0.1,I=11,15/)
```

V příkladech je zároveň ukázka použití cyklu v inicializaci.

1.4.3 Datové struktury

Nové datové typy (záznamy) si můžeme definovat takto:

```
TYPE TRIPLET
   REAL,DIMENSION(3)::VERTEX
END TYPE TRIPLET
TYPE (TRIPLET),DIMENSION(10)::T
```

[8] Název je zkratkou za FORmula TRANslator, tedy překladač vzorců.
[9] Popis SAVE definuje proměnné, jejichž hodnota bude po opuštění modulu a opětovném vstupu zachována.
[10] Pole vyšších řádů jsou hojně využívána např. v teorii pružnosti – tenzory.

Takto definujeme pole T položek nového typu; pro přístup ke složkám pak píšeme `T(2)%VERTEX`.

Fortan 90 podporuje též ukazatele a dynamicky deklarované proměnné:

```
REAL,POINTER::SON
REAL,POINTER,DIMENSION(:,:)::A
ALLOCATE(SON,A(N,N))
DEALLOCATE(A)
```

Zde jsme vytvořili ukazatel na matici A (dvojtečky bez uvedení mezí uvádějí jen rozměr), a poté jsme matici alokovali.

1.4.4 Vrstvy

Účelem vrstev je výřez z násobné hodnoty (z pole). Tak např. z pole

```
REAL,DIMENSION(N,N)::A,B
```

můžeme vybrat

- třetí sloupec: `A(1:N,3)`,
- totéž stručněji: `A(:,3)`,
- nebo dokonce jeho liché prvky: `A(1:N:2,3)`

Přiřazení polí lze psát stručně `A=B`, případně `A=f(B)`, anebo (má-li to, vzhledem k rozměrům smysl) `A+B`.

1.4.5 Deskriptory násobných hodnot

Ukážeme si nyní techniku, která je používaná pro tyto operace s poli. Za deskriptor budeme považovat čtveřici *(dimenze, adresa prvního prvku, meze indexů, délka typu elementu)*. Tak matice `A(1:N,1:N)` má deskriptor

$$(2, \&A(1,1), 1, N, 1, N, 4)$$

Vidíme, že adresu prvku A(i,j) spočteme jako $(i-1)*4*N+(j-1)+\&A(1,1)$. Zřejmě by se dal výpočet usnadnit, pokud by deskriptor byl ve tvaru

$$(2, \&A(0,0), 1, N, \underbrace{4*N}_{K_1 \ldots krok}, 1, N, \underbrace{4}_{K_2 \ldots krok})$$

Potom by totiž stačilo

$$\&A(i,j) = \&A(0,0) + i*K_1 + j*K_2$$

čímž se výpočet adresy nejen značně zprůhlední a zjednoduší, ale zároveň dostáváme i něco obecnějšího: můžeme takto reprezentovat výše popsané výřezy z polí, např. vektor `A(1:N,3)` bude mít deskriptor

$$(2, \&A(0,3), 1, N, 4*N)$$

Vektorovému procesoru pak tedy můžeme předávat např. právě takovéto deskriptory.

Přiřazení zde funguje tak, že se zavolá procedura pro přiřazení, které se předají vyhodnocené deskriptory levé a pravé strany.

Hodnoty (proměnné) mají přitom také statickou část:

- skalár (jednotlivá proměnná) je adresována staticky,
- pole má deskriptor násobných hodnot,
- struktura sestává z posloupnosti statických částí jednotlivých položek.

Dynamickou část přitom tvoří hodnoty získané indexováním.

Nejsnazší řešení je pak zavést deskriptor typu:

- pro skaláry – kód,

- pro násobnou hodnotu – kód „dimension“ a deskriptor hodnoty prvků,
- pro datovou strukturu – posloupnost deskriptorů polí, tvar *„struct“ počet polí, deskriptory polí.*

U parametrů stačí definovat dimenzi a dolní meze.

Ve Fortranu 90 existují ještě další operace nad poli:

`SQRT(A)` provede odmocninu nad všemi prvky,

`WHERE (A>0.0) A=1.0/A` provede příkaz nad všemi prvky, splňujícími podmínku.

Přiřazení polí lze provést i u polí s různými mezemi – stačí stejný rozměr (počet prvků).

Nový Fortran rozlišuje též vstupní a výstupní parametry (ve starších verzích bylo pouze volání odkazem – s výjimkou případu, kdy *skutečný* parametr byl výraz).

1.5 Jazyk OCCAM 1

je jazyk nízké úrovně pro multiprocesorové systémy. Samotný transputer je přitom RISC-ový procesor se 4–32 KB lokální paměti, čtyřmi vstupně–výstupními kanály a jednotkou pro operace v pohublivé řádové čárce (Floating Point Unit – FPU).

Jazyk OCCAM 1 byl výborný nápad malé (slabé) firmy INMOS Ltd. (UK); bohužel jej nedotáhli do konce, trh ovládly firmy z USA – nejvýznamnější je firma Connection Machines.

OCCAM 1 je vhodný pro některé algoritmy, a zejména pro vysvětlení principů. Jeho rozšířením je OCCAM 2, zejména obsahuje typovost.

1.5.1 Základní rysy OCCAMu 1

V OCCAMu 1 je každý příkaz procesem. Nejdůležitější příkazy jsou:

- vstup: `KANÁL?PROMĚNNÁ`
- výstup: `KANÁL!VÝRAZ`
- přiřazení: `PROMĚNNÁ:=VÝRAZ`

Procesy se přitom dají skládat

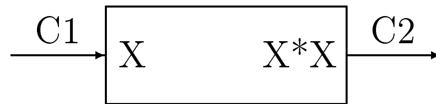
- paralelně – konstrukce `PAR`,
- sekvenčně – konstrukce `SEQ`,
- alternativně (podmíněně) – konstrukce `ALT (IF)`.

Je zaveden pevný formát programu, podřízená konstrukce se odsazuje o jednu mezeru.

Jsou zde jen omezené možnosti pro tvorbu podprogramů a polí, jazyk je málo ortogonální. Doporučený postup programování je přitom tento:

1. Napsání programu.
2. Simulace v pseudoparalelním módu.
3. Zobrazení procesů.
4. Vlastní (paralelní) výpočet.

Důsledkem nedotaženého vývoje je mimo jiné i to, že často bývá síť transputerů napojena jako periferie k PC.



Obrázek 7: OCCAM: proces realizující druhou mocninu

1.5.2 Příklady programování v jazyce OCCAM 1

Příklad 1.3

Ukážeme si, jak se v OCCAMu napíše program (proces) pro výpočet druhé mocniny. Zároveň nám tento příklad bude demonstrovat filosofii jazyka.

Proceduru (proces) budeme považovat za „krabičku" (černou skříňku) s definovaným vstupem a výstupem; s okolím komunikuje prostřednictvím kanálů – tak, jak nám to ukazuje obrázek 7.

Program pak napíšeme následovně:

```
VAR X:
SEQ
 C1?X
 C2!X*X
SEQ ...
```

Nyní si všimněme některých rysů:

- Proměnné nemají specifikovaný typ.
- Příkaz pro čtení čeká, až něco přijde (pozor na deadlock!).
- Příkaz pro zápis čeká, až si někdo výstup odebere (je to v souladu s pravidly CSP).
- Poslední řádek ukazuje, jak by se (bez odsazení) psala další konstrukce programu; pokud bychom ji odsadili, získáme vnořenou konstrukci.

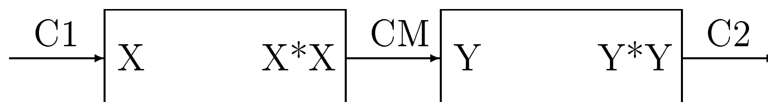
Pokud chceme proces, který bude pracovat „neustále", tedy bude vždy dávat na výstup druhou mocninu vstupu, napíšeme (opět odsazeně) konstrukci WHILE:

```
WHILE TRUE
 VAR X:
 SEQ
  C1?X
  C2!X*X
```

□

Příklad 1.4

Nyní si ukážeme tvoření sítí – vytvoříme ze dvou „umocňovačů na druhou" proces, umocňující na čtvrtou, jak nám to ukazuje schéma na obrázku 8. Kanál CM přitom patří oběma procesům, a proto napíšeme globální deklaraci.



Obrázek 8: OCCAM: síť procesů pro čtvrtou mocninu

```
CHAN CM       (pro celý úsek)
PAR           (oba WHILE paralelně)
 WHILE TRUE   (první krabička)
  VAR X:
  SEQ
   C1?X
   CM!X*X
 WHILE TRUE   (druhá krabička)
  VAR Y:
  SEQ
   CM?Y
   C2!Y*Y
```

(Vidíme, jak nešikovné je odsazování, které musíme přesně dodržet.) □

Příklad 1.5

V předchozím příkladu jsme měli opakování kódu. Ukážeme, si, jak je možné toto napsat pomocí podprogramů. Ty jsou otevřené, což znamená, že každé volání vytvoří staticky nový proces.

```
PROC square (CHAN source,sink)=
 WHILE TRUE
  VAR X:
  SEQ
   source?X
   sink!X*X
```

Takto můžeme předchozí příklad zkrátit na

```
CHAN CM
PAR
 square (C1,CM)
 square (CM,C2)
```

Dalo by se říct, že popis PROC je vlastně makro, protože takto se vytvoří dva procesy.

1.5.3 Alternující procesy

Ukážeme si nyní něco, co je obdobou vícecestného čekání v Adě.

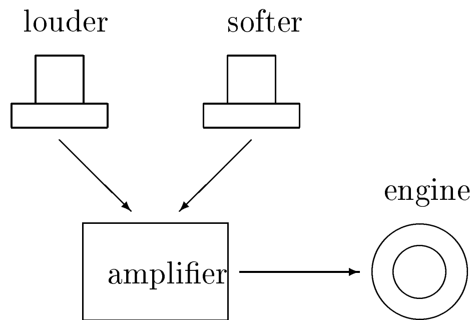
Příklad 1.6

Budeme programově ovládat zesilovač (amplifier), který ovládá motor (engine) pomocí dvou tlačítek (zrychlení, zpomalení) – viz schéma na obrázku 9. Stisk tlačítka `Louder` znamená přitom zvýšení výkonu o jednotku, stisk `Softer` snížení výkonu o jednotku. Mačkat mohu cokoliv, program čeká pouze na jeden z nich. Použijeme konstrukci

```
ALT
 louder?ANY
 softer?ANY
```

která čeká na splnění některé z podmínek, tedy až se nějaká (jakákoli – ANY) hodnota objeví na kterémkoli kanále. Celý program pak vypadá takto:

```
VAR volume:           (současná hodnota)
SEQ
 volume:=0            (inicializace)
 WHILE TRUE           (nekonečný cyklus)
  ALT                 (přepínač, provede se větev, která se otevře dříve)
   louder?ANY         (první větev)
    SEQ
```



Obrázek 9: Řízení zesilovače

```
        volume:=volume+1
        amplifier!volume
      softer?ANY            (druhá větev)
       SEQ
        volume:=volume-1
        amplifier!volume
```

□

Příklad 1.7

Pokud nyní navíc chceme, aby hodnoty `volume` byly nějak omezeny, použijeme složenou podmínku:

```
DEF max=10,min=2              (pojmenované konstanty)
VAR volume:                   (současná hodnota)
SEQ
 volume:=min                  (inicializace)
 WHILE TRUE                   (nekonečný cyklus)
  ALT                         (přepínač...)
   (volume<max) & louder?ANY  (první větev)
    SEQ
     volume:=volume+1
     amplifier!volume
   (volume>min) & softer?ANY  (druhá větev)
    SEQ
     volume:=volume-1
     amplifier!volume
```

□

Tvar „stráží“ (guards) může být přitom tento:

`podmínka & vstupní proces` – čekám na podmínku a vstup,

`podmínka & SKIP` – čekám jen na podmínku,

`podmínka & WAIT` – čeká na podmínku, nejméně však určený čas (tedy podmínka se začne zkoumat až po uplynutí doby).

Ukážeme si ještě čekání na časový limit – WAIT.

Příklad 1.8

```
DEF timeout=100:
VAR clock,X:
SEQ
 clock:=NOW                        (lokální čas)
 ALT
  C1?X                             (stráž, guard)
   C2!ok.message;x
  WAIT NOW AFTER clock+timeout     (čekání na čas)
   C2!timeout.message
```

□

Pro úplnost zde ještě uveďme tvar příkazu IF:

```
IF
 i=1
  out!x
 i=2
  out!y
```

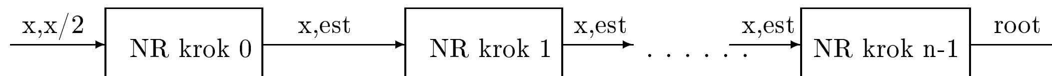
□

1.5.4 Pole procesů

si budeme demonstrovat na následujícím příkladu, v němž budeme počítat hodnotu druhé odmocniny známou Newtonovou–Raleighovou iterační metodou, kdy nový odhad y' spočteme ze starého y ze vztahu

$$y' = \frac{\left(y + \frac{x}{y}\right)}{2}$$

Procesory realizující n iterací jsou přitom zapojeny tak, jak to ukazuje obrázek 10.



Obrázek 10: Pole procesů pro iteraci druhé odmocniny

Příklad 1.9

Nejprve si nadefinujeme jeden proces:

```
VAR X,Estimate:
SEQ
 values[i]?X
 values[i]?Estimate
 values[i+1]!x
 values[i+1]!(X/Estimate+Estimate)/2
```

Nyní spojíme tyto procesy do pole procesů

```
CHAN values[n+1]
PAR i=[0 FOR n]
 WHILE TRUE
  VAR X,Estimate:
  SEQ
```

```
  values[i]?X
  values[i]?Estimate
  values[i+1]!x
  values[i+1]!(X/Estimate+Estimate)/2
```

Celý program pro odmocňování pak vypadá takto:

```
PAR
 PAR i=[0 FOR n]
 ... (viz výše)
 WHILE TRUE         (první iterace)
  VAR X:
  SEQ
   Root?X
   values[0]!X
   values[0]!X/2
 WHILE TRUE         (poslední iterace)
  VAR root
  SEQ
   values[n+1]?ANY
   values[n+1]?root
```

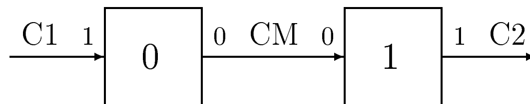
Počet iterací – číslo n musí v tomto případě být zadáno předem.

1.5.5 Vazby na hardware

V příkladu 1.4 jsme si ukázali, jak se dva procesy spojí do komunikující sítě. Nyní navážeme, a to vazbou na konkrétní hardware a vstupně–výstupní kanály:

```
CHAN CM
PLACED PAR           (definujeme, co kam patří)
 ALLOCATE 0          (obsazuji, definuji 0-tý procesor)
  PORT 0:-CM
  PORT 1:-C1         (mapování kanálů)
  LOAD 1             (návaznost na 1)
  buffer(C1,CM)
 ALLOCATE 1          (procesor č. 1)
  PORT 0:-CM
  PORT 1:-C2         (mapování kanálů)
  LOAD 0             (návaznost na 0)
  buffer(CM,C2)
```

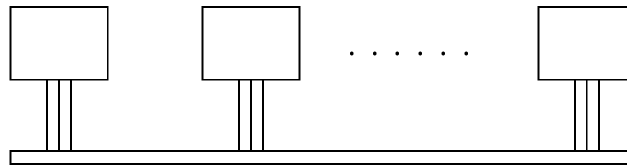
Tím se definuje propojení zachycené na obr. 11. Aby se nemuselo realizovat fyzické propojení, našlo se následující



Obrázek 11: OCCAM: definované propojení procesorů

řešení: řada transputerů je zapojena k tzv. *přepojovacímu poli* (switching array), které funguje podobně jako telefonní ústředna (schéma na obr. 12). Přepojení se přitom dá měnit i během výpočtu, ale tato možnost se příliš často nepoužívá.

Řekli jsme, že OCCAM je produktem britské firmy INMOS Ltd.; americké firmy naopak zvolily pevné propojení, automatické mapování, a cesty zpráv jsou určeny globálně. V současné době se zkoumá, jestli je možné nemít to propojené „natvrdo“.



Obrázek 12: Přepojovací pole

1.6 Ada

1.6.1 Vazby na hardware

Ada je moderní programovací jazyk, vhodný zejména na programování real–time systémů (komunikující procesy). Ukážeme si ale, že v Adě lze také programovat „na nízké úrovni“, např. v absolutních adresách, je možné „na bit“ popsat strukturu hardwarových zpráv apod.

Příklad 1.10

Pro definici povelů nějakého zařízení můžeme tak psát

```
type command is (start,stop,repeat,hold,resume);
order: command;
for command'size use 8;
for order use at 16#8001#;
for command use (start=>1,stop=>2,repeat=>4,hold=>8,resume=>16);
```

se zřejmým významem – definujeme výčtový typ `command` a proměnnou `order`, dále velikost, umístění, a přesnou specifikaci hodnot.

Příklad 1.11

Podobně je možné programovat sedmisegmentový displej – pokud si např. očíslujeme segmenty od nuly pro horní segment, dále proti směru hodinových ručiček, prostřední segment bude mít číslo 6, můžeme psát

```
type digit_display is ('0','1',...,'9','C','d',...);
for digit_display use ('C'=>15,'1'=>48,...);
```

přičemž vidíme, že hodnoty výčtového typu můžeme označovat i znaky (nejen identifikátory).

Příklad 1.12

V dalším příkladu budeme implementovat doskové operace, přičemž příkazový blok disku v systému vypadá tak, jak jej zachycuje schéma na obr. 13. (Hvězdičky označují bity, jejichž hodnota není využita.) Přitom čtveřice bitů,

Obrázek 13: Příkazový blok disku

označená jako `f`, označuje operaci (funkci):

`0001` znamená čtení,

`0010` zápis,

`0100` zápis s testem (verifikací),

`1000` hledání (seek).

Dále máme v příkazovém bloku tři indikátory

`p` – zápis zakázán,

`b` – disk běží (busy),

`l` – vypnuto (off line).

V Adě pak příkazový blok můžeme popsat následovně:

```
type disk_function is (read,write,write_and_verify,seek);
type disk_kontrol_record is
   record function_code: disk_function;
          disk_address: integer;
          store_address: system.addres;
          words_transfered: positive;
          write_not_permitted: Boolean;
          disk_busy: Boolean;
          disk_off_line: Boolean;
   end record;
```

Pokud zůstaneme u této deklarace, umístí kompilátor položky záznamu tak, že to fungovat nebude; musíme proto dále specifikovat

```
for disk_function use (read=>1,write=>2,write_and_verify=>4,seek=>8);
for disk_function'size use 4;   -- 4 bity
for disk_control_record use
   record
      function_code at 0 range 0..3;   -- at .. adresa v byte
      disk_address at 2;
      store_address at 4;
      words_transferred at 6;
      write_not_permitted at 0 range 5..5;
      disk_busy at 0 range 6..6;
      disk_off_line at 0 range 7..7;
   end record;

for disk_control_record'size use 8*system.storage.unit;
control_block: disk_control_record;
for control_block use at 8#77340#;
```

Nyní je již všechno umístěno tak, jak má; navíc je struktura maximálně zhuštěna, poněvadž jsme přikázali použít 8 bajtů.

1.6.2 Modernizace Ady – Ada 9x

Ada je jazyk, který se poměrně intenzivně vyvíjí; poslední modernizaci zažila v roce 1983. Autoři jazyka se zhlédli v mechanismu schůzky (rendez–vous) natolik, že v rámci zachování čistoty jazyka neobsahovala Ada nic jiného. Přesto nové verze Ady již obsahují např.:

- dědění,
- synchronizaci na společnou paměť (protected variables),
- zobecnění možností tvoření adres,
- odstranění některých chyb předešlých verzí.

Některé z rysů si předvedeme podrobněji.

Dědění. Ukážeme si, jak se dá v Adě napsat příklad s dědičností, kdy nejprve nadefinujeme typ `rectangle` (obdélník), a posléze jej s pomocí dědičnosti „rozšíříme“ na kvádr (`cuboid`).

Příklad 1.13

```
type rectangle is tagged  -- tagged znamená, že jej lze rozšiřovat
   record
      length: float;
      width:  float;
   end record;
type cuboid is new rectangle with
   record
      height: float;
   end record;

type shape is new rectangle with null record;

function size(R:in rectangle) return float is
begin
   return R.length*R.width;
end size;

function size(C:in cuboid) return float is
begin
   return size(rectangle(C))*C.height;
end size;
```

□

Využití dědičnosti je přitom zpravidla výhodnější než variantní záznamy: pokud přidáváme něco nového, nemusíme (je-li to vhodně udělané) znovu rekompilovat část pro `rectangle` – jedná se o tzv. late binding.

Odkazy na podprogramy. Novinkou jsou ukazatele na procedury a funkce, jaké známe např. z jazyka C. Napíšeme-li v Adě deklaraci

```
type trig_function is access function(F:float) return float;

T: trig_function;
X,Theta: float;
```

(kde `access` vytváří ukazatel), můžeme psát např.

```
T:=sin'access;
X:=T(Theta);
X:=T.all(Theta);
```

přičemž poslední tvar s `all` je nutný, pokud nemá funkce parametry.

Zobecnění odkazů. Viděli jsme, jak se v Adě píší mimo jimé i odkazy na podprogramy. Všechno ale můžeme zobecnit, což je zajímavé i z hlediska dalšího možného vývoje.

Při práci s adresami požadujeme zejména

- obecnost (abych mohl mít adresu od všeho),
- bezpečnost (nelze použít „odumřelý“ odkaz),
- rychlost.

Přitom je zajímavé, že se až do vytvoření Ady nepodařilo splnit všechna tato kritéria.

V Adě přitom (za cenu jistého snížení bezpečnosti) mohu ukazovat i jinam, než na haldu (heap):

```
type Int_Ptr is access all integer;
IP: Int_Ptr;
I: aliased integer;   -- aliased => lze zpřístupnit access
IP:=I'access;
```

Přitom proměnná `IP` nesmí mít širší oblast platnosti (scope) než `I`.

Ukazatel se ukládá do zvláštní oblasti (ne na hromadu), a tím dochází ke zrychlení. Pro použití v potenciálně nebezpečných konstrukcích jsou poměrně komplikované podmínky. Dále si všimněme, že vše je testovatelné v době kompilace.

Ještě si ukážeme, jak lze odlišit odkaz na konstantu:

```
type Const_Int_Ptr is access const integer;
CIP: Const_Int_Ptr;
C: aliased constant integer:=815;
CIP:=C'access;
```

□

Semafory. Víme, jak funguje semafor: dá se říct, že jeho funkce je podobná železničnímu autobloku: vpustí „dovnitř" (do kritické sekce, resp. na trať) pouze omezený počet požadavků (procesů, resp. vlaků). Při otevření semaforu (opuštění) se přitom dostane dále jeden, anebo všichni (použitelné např. u čtenářů a písařů).

K semaforu přistupujeme v zásadě pomocí dvou operací:

`wait(P)` – čekej, při průchodu zavři,

`signal(P)` – otevření, uvolnění.

Důležitou vlastností je přitom chování při kolizi, tedy při současném příchodu požadavků: musí být lhostejno, který se vybere jako první.

Společná paměť. Ukážeme si, jak se v Adě programuje kritická sekce pro přístup ke společné proměnné.

```
protected Variable is
  function Read return Item;
  procedure Write (NewValue: Item);
private  -- nevidím vnitřní strukturu
  Data: Item
end Variable;

protected body Variable is   -- realizujeme vzájemné vyloučení
  function Read return Item is
  begin
    return Data;
  end Read;

  procedure Write (NewValue: Item) is
  begin
    Data:=NewValue;
  end Write;

begin  -- inicializace
  Data:=Init_Value;
end Variable;
```

□

Implementace Dijkstrova semaforu.

```
protected type C_Sema (Start_Count: integer:=1) is
  -- parametr má implicitní hodnotu
  entry Secure;
  procedure Release;
  function Count return integer;
private
  current_count: integer:=Start_Count;   -- inicializace
end C_Sema;

protected body C_Sema is
  entry Secure when current_count>0 is
  -- není-li splněna podmínka, čekám
  begin
    current_count:=current_count-1;
  end Secure;

  procedure Release is
  begin
    current_count:=current_count+1;
  end Release;

  function Count return integer is
  begin
    return current_count;
  end Count;
end C_Sema;
```

Zde `Secure` odpovídá Dijkstrově operaci `P` (obsazení), `Release` operaci `V` (uvolnění); jedná se o n-ární semafor.

Když se to realizovalo, podařilo se oproti schůzce asi dvacetinásobné zrychlení.

1.7 Mikrojádro

Je známá věc, že operační systém Unix se rozšířil díky své snadné přenositelnosti na různé platformy. Přenos se dá přitom realizovat tak, že přeložíme jeho zdrojový text v jazyce C; překladač jazyka C na nový počítač sestrojíme takto:

1. Máme kompilátor z jazyka C do strojového kódu M, napsaný v jazyce C (označíme $C \to M[C]$).
2. S poměrně malou námahou napíšeme $C \to M1[C]$ – na nový počítač.
3. Nový překladač s pomocí starého přeložíme – získáme křížový překladač $C \to M1[M]$, tedy překladač jazyka C do strojového kódu nového počítače, který však pracuje na starém.
4. Nyní již z $Unix[C]$ snadno získáme $Unix[M1]$.

Překladač $C \to M[M]$ mohu přitom z jeho zdrojové podoby $C \to M[C]$ získat např. takto:

1. Vezmeme $C1 \to M[M]$, ten získáme „nějak“, např. pomocí $C \to M[SETL]$. (Zde $C1$ je podmnožina jazyka C.)
2. Napíšeme $C2 \to M[C1]$, přičemž $C2 \supseteq C1$. Ten přeložíme na $C2 \to M[M]$.
3. V několika dalších iteracích rozšíříme na plnohodnotné C.

Toto překládání však neřeší přenos mezi operačními systémy.

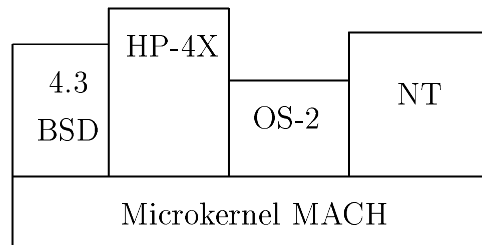
Poznámka. Pro linkování jsou v objectu přítomny dva seznamy:

- export – která jména jsou v modulu definována a dána k dispozici,
- import – co je potřeba definovat (co požadujeme z jiného modulu).

Sestavovací program (linker) pak pracuje s těmito odkazy.

1.7.1 Princip mikrojádra

Základem operačních systémů, které používají mikrojádro (microkernel), je definované rozhraní mezi strojově nezávislými částmi operačního systému, a mikrojádrem, které je zčásti závislé na hardwaru (viz obrázek 14). Princip



Obrázek 14: Rozhraní systémů s mikrojádrem

mikrojádra byl poprvé použit v systému Mach (dnes Mach 3), dále jej využívají operační systémy jako NeXT, Windows NT, Cairo, CHORUS.

V modelu operačního systému tak vstupuje mezi hardware a jádro operačního systému další vrstva – mikrojádro. Na mikrojádru se tedy nejlépe programují systémy, které mají jednoduché rozhraní.

1.7.2 Základní rysy systému s mikrojádrem

Základní pojmy. Mikrojádro používá následující terminologii:

task – execution environment: jsou to zde „zdroje“, jako paměť, porty, procesory;

thread – vlákno: jedná se vlastně o proces, ale se zdroji sdílenými s ostatními vlákny úlohy (task); dá se říct, že vlákno představuje výpočet;

port – fronta zpráv: odesílatel musí mít právo zápisu, příjemce právo čtení; zprávy přitom nemusí jít fyzicky přes port;

port set – množina portů se společnou frontou;

message – zpráva, je to „typed collection of objects“: buďto data, nebo odkaz na data, nebo práva přístupu k portům;

memory objects – zařízení na práci s pamětí, je velice obecné.

Funguje to přitom i v distribuovaném prostředí (podstatný „trik“ je skrytý v portech). Dále v mikrojádru pracuje líné vyhodnocování odkazů (až je to nezbytné).

Správa procesů. Zatímco Unix vždy vytváří nové zdroje (kopie adresového prostoru) a vytváří se jedno výpočtové vlákno, obecně to tak být nemusí.

Vlákno může být running nebo suspended; jsou zde proto operace `thread_suspend(id)` a `thread_resume(id)`, kde `id` je systémové číslo vláknu určené.

Vlákna přitom vytváří tyto funkce (pozor – prostředí se vytváří jinak):

`cthread_fork` – vytvoří nové vlákno,

`cthread_exit` – ukončí vlákno,

`cthread_join` – čekám na syny (po `fork`-u),

`cthread_detach` – alternativa `fork`-u, po které nikdy nebude `join`,

`cthread_yield` – vrácení procesoru.

S těmito prostředky se dá napsat jak round robin, tak časové kvantum.

Prostředky pro vzájemnou synchronizaci. Existují dva, a to:

společná proměnná – tedy kritickou sekci provádím pomocí proměnné `x`; k dispozici máme funkce

`mutex_alloc(x)` – vytvoření proměnné pro vzájemné vyloučení (mutual exclusion – mutex);

`mutex_lock(x)` – čekám, až bude volno, potom zamknu;

`mutex_unlock(x)` – odemknutí, signal;

`mutex_wait(x)` – čekám na uvolnění;

`mutex_free(x)` – zrušení proměnné.

podmínková proměnná – náhražka za neimplementovaný monitor:

`condition_alloc(c)` – vytvoření podmínkové proměnné;

`condition_free(c)` – zrušení;

`condition_wait(c,m)` – znamená vlastně `if c then wait(m) endif`, tedy čeká, až `c` nabyde hodnoty `true`, a pak se čeká na `m`; provádí se ale jako *nedělitelná* akce;

`condition_signal(c)`

□

Implementace monitoru. Ukážeme si případ producer—consumer.

```
mutex_alloc(m)
condition_alloc(nonempty,nonfull)   -- dvě proměnné

producer
repeat
  mutex_lock(m)
  while (full)  -- počkám
  condition_wait(nonfull,m)
  -- zapiš do bufferu
  condition_signal(nonempty)
  mutex_unlock(m)
until false

consumer
repeat
  mutex_lock(m)
  while (empty)  -- počkám
  condition_wait(nonempty,m)
  -- vyber data
  condition_signal(nonfull)
  mutex_unlock(m)
until false
```

□

Kvantování času. Počet přerušení zřejmě závisí na počtu vláken. Za kvantum se bere interval 10 ms; proto frekvence = 10 ms * počet procesorů.

Výjimky. Předvedeme si ještě ošetření výjimečných situací, které fungují podobně jako výjimky (exceptions) v Adě; může jít např. o hardwarové chyby, ale také o detekce situací, rovněž lze takto zpracovat volání uživatelských služeb.

Zpracování probíhá tak, že „oběť“ zavolá `raise(exception_name)` a čeká na vyřízení. To způsobí vyvolání funkce – handleru, který má tvar `výjimka(thread,task,typ)`, přičemž typ odpovídá parametru, se kterým se volalo `raise`; navíc se do handleru předá úloha a vlákno, ve kterém k výjimce došlo. Jedná se tedy o příjem zprávy.

V těle handleru pak buď přímo prohlásím výjimku za vyřízenou a vrátím se zpět za `raise`, anebo je možné provést např. restart oběti (s nějakým indikátorem) a ukončení vlákna.

1.7.3 Komunikace

Zajímavou kapitolu představuje meziprocesní komunikace (Inter Process Communication – IPC), která pokrývá BSD pipe, signal, priority, socket.

Výměna zpráv se děje přes location independent ports, tedy porty, jejichž skutečná poloha v systému není specifikována. Pro práci s porty jsou funkce

`port_allocate(task_self)` – do parametru se dává identifikace portu;

`port_deallocate(p)`

`port_status(p)`

a dále pro množiny portů

`port_set_backup`

`port_set_allocate(s)` – definice množiny

`port_set_remove(s)` – zrušení

`port_set_status(s)`

`port_set_include` – rozšíření množiny.

Předávání zpráv. Jako adresáta uvádím port, namísto zprávy uvádím místo, kde se zpráva nachází. Píšeme pak `msg_send(port,zpráva)`.

Pokud při vyslání zprávy čekáme na odpověď, uvedeme navíc jako další parametr místo, kam si odpověď přejeme uložit: `msg_RPC(port,zpráva,kam)`, kde RPC znamená Remote Procedure Call, příkaz čeká na příchod odpovědi.

Posledním příkazem je `msg_receive(port,kam)`, který přebírá zprávu (opět zadáváme adresu, kam zprávu uložit).

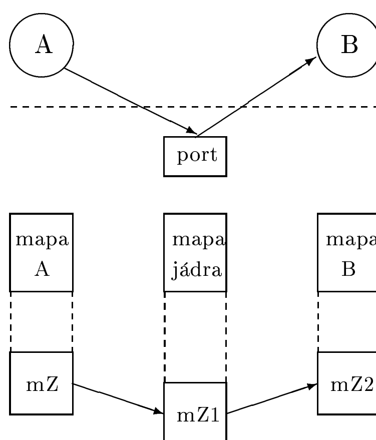
Mapování zprávy. Hovoříme zde o procesech na stejné platformě.

Paměť není (nemusí být) souvislá, je organizovámá ve třech patrech: má segment (což je největší jednotka), region, stránku; pak již adresa.

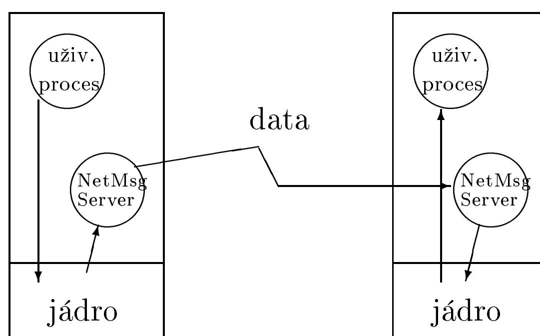
Mějme tedy zprávu, kterou posílá proces A na port, který patří procesu B (viz schéma na obr. 15). Fyzicky se provedou tyto činnosti: z A se zavolá jádro, zpráva se přemapuje do oblasti jádra, a jádro ji posléze přemapuje do cílového adresového prostoru.

Přesněji řečeno, celou činnost zprostředkovává síťový server (Net Message Server), který je přítomen v obou počítačích; zprávy si pak vlastně vyměňují mezi sebou, jak to ukazuje obrázek 16.

Paměťové objekty se chápou jako port, metody a data; paměť přitom může být v systému distribuovaná.



Obrázek 15: Mapování zpráv



Obrázek 16: Předávání zpráv pomocí síťového serveru

1.7.4 Rozhraní na Mach z C

funguje tak, že preprocesor vygeneruje kus souboru *.c pro zavlečení include, a dále dva soubory s procedurami.

Jednotlivé prvky jsou implementovány následovně:

- `fork`: udělá se kopie tabulek virtuální paměti;
- objekty: hlídá se u nich počet odkazů (je-li roven nule, zruší se);
- nepoužívají se kritické sekce (je-li více procesorů, jeden z nich rozhoduje) – objekty se samy zamykají;
- značná část kódu byla převzata z BSD;
- identifikace objektů se provádí zásadně přes port;
- úspěšně byl prověřen řetěz: shell interpretuje příkazy, jeho podprogramy jsou napsány v příkazech mikrojádra, mikrojádro provede, hardware realizuje.

1.8 Shrnutí

Udělali jsme průřez problematikou paralelních výpočtů a paralelních architektur. Připomeneme si ještě jednou nejdůležitější rozdíly mezi jednotlivými typy:

- Symetrický multiprocesing (např. Silicon Graphics) je vhodný pro počet procesorů max. 10–12; rovněž zrychlení bývá zhruba desetinásobné.
- Vektorový procesor (Cray, Convex) je záležitostí vnitřní architektury; rovněž má smysl uvažovat zhruba o desítce procesorů.

  Přitom pro vektorové procesory již musíme psát programy s použitím speciálních technik, které využijí vlastností těchto architektur. Oproti tomu symetrický multiprocesing se programuje „klasicky".
- Masivně paralelní architektury (Thinking Machines, IBM SP–2) již představují rozsáhlé systémy, pro které není horní mez v počtu procesorů: existují i systémy s 2^{10} až 2^{16} procesorů.

Probírali jsme také technologii mikrojádra, která vedla celkem ke dvěma standardům, z nichž jeden – Mach – jsme rozebírali podrobněji.

2 Softwarové inženýrství

2.1 Tvorba rozsáhlých systémů

2.1.1 Současná situace

Ve světě se v současné době tvorva software stává velkovýrobou, dochází ke koncentraci, ubývá prostoru pro „osamělé vlky".[11] Není tedy téměř žádná možnost prosadit se ve standardním software. (Tak například ustupuje Software 602, poněvadž firmy přecházejí na MS Word.)

V aplikacích se preferují větší firmy (malá firma snáze zkrachuje apod.), je proto tendence kupovat značkový software (a také značkový hardware). Software se kupuje hotový.

2.1.2 Customizace

Dodávaný software bývá často jako balík s parametry; *customizací* pak rozumíme nastavení parametrů pro danou aplikaci (konkrétní instalaci). Jsou přitom možné i dodělávky.

Životní cyklus softwaru přitom doznává určitých změn – zejména co se týče pracnosti. Zhruba nám ji zachycuje tabulka 1. Přitom customizace má určitá specifika – návrh je prováděn pouze částečně, kódování víceméně vypadne,

Položka	*Klasicky*	*Customizace*
Specif. cílů	5	5
Specif. požadavků	15–20	15–20
Návrh	20–30	do 10
Kódování	15–20	pod 10
Testování	30–50	do 20
Předání	(0)	(0)
Součet	asi 100	asi 40
Údržba	200	50

Tabulka 1: Snížení nákladů při customizaci

namísto testování nastupuje zkušební provoz.

Použitím customizace klesá pracnost tvorby SW zhruba trojnásobně; celkem (s údržbou) 4–5 krát. Většina pracnosti je přitom skryta v části cíle—specifikace—návrh.

Je výhodné provádět customizaci i tak, že dodáváme rozsáhlejší systém, ze kterého část odstraníme; největší výhodou pro nás jako výrobce je to, že udržujeme pouze jeden systém.

Takováto praxe (balíky a následná customizace) je zejména v ekonomii: např. systém R3 firmy PC–DIR (účetnictví, skladové hospodářství, plánování výroby) za 10–100 mil. Kč; převzetím systému se přebírá „know–how" všech běžících instalací. Je zřejmé, že tento systém si nemůže dovolit ledaskdo: omezením je již cena, dále pak požadavky na organizaci (aby se věci dělaly určitým, a ne jiným způsobem).

[11] Viz též článek v jednom z posledních čísel Bajtu: Konec zlatých časů.

2.1.3 Nebezpečí organizačních změn

vyvolaných nasazením softwaru je velké, proto raději nesahat do organizace, pokud to není nevyhnutelně nutné. Pokud to nutné je, dojde zákonitě k poklesu výroby. Podnik je totiž protkán jemným předivem obtížně popsatelných vazeb (lidský faktor – pracovníci ví, za kým konkrétním mají jít), jejichž narušení bývá drastické.

K této situaci došlo ve Škodě Mladá Boleslav, která přešla na nový informační systém: namísto FoxPro začali používat Oracle a specializovaný informační systém. Zatímco na FoxPro byl již zvyklý tým, který odstraňoval chyby apod., při přechodu na nový systém byla nutná změna pravidel, všichni se s ním museli seznamovat, v důsledku čehož došlo k poklesu výroby až o 50%. (Další příčinou bylo zavedení nového modelu – Škody Felicia.)

2.1.4 Interview

Při specifikaci požadavků musíme zjistit, co zákazník potřebuje. K tomu je nejvhodnějším prostředkem *interview* (rozhovor, pohovor), který nám pomáhá odpovědět na otázky: co se požaduje, co a jak funguje apod. Tím se předchází mimo jiné vnucení nesmyslných organizačních změn.

V průběhu interview vyvstanou zejména tyto problémy:

Špatní lidé, špatný čas. Je třeba najít vhodné lidi, kteří znají potřebné informace; také není vhodné přicházet v čase, který je napjatý (nechodit za účetními začátkem roku).

Stává se také, že ani management neporadí, za kým jít (osobní antipatie apod.), popř. je přesvědčen, že podniku rozumí jen on sám.

Špatné otázky, špatné porozumění. Proto se nesmíme ptát na hlouposti; je vhodné ptát se nejprve na obecnosti, poté přejít k podrobnostem, ale v žádném případě zbytečně neobtěžovat.

Nedůvěra. K tomu je nezbytné zachovat celou řadu pravidel:

- Nevyvolat pocit ohrožení. Systém není nasazován proto, aby se ušetřili lidi (alespoň prvotně), ale např. pro jejich lepší využití.
- Nepohrdat dotazovaným, netvářit se a nejednat povýšeně. Ten člověk je potřebný, i když je to dělník ve špinavých montérkách.

Existuje rozsáhlý systém poznatků, jak dělat (a jak nedělat) interview; dále např. není vhodné napodobovat vlezlost novinářů (odpuzuje to).

Proto sestavujeme tzv. **plán interview**. Znamená to, prozkoumat organizaci, zjistit kdo s kým co dělá, kdo za co odpovídá, jak se lidé vzájemně doplňují apod.

Je vhodné začínat od prodejců, protože ti by měli určovat chod podniku. Mnoho informací dostaneme „bokem" (o konkrétních lidech); hodně ví sekretářky.

Před provedením interview je třeba si vyžádat povolení – spíše jako žádost o spolupráci. (Vedení nemá rádo, když se moc „šmejdí" v podniku.) Pozor – „ředitel ví všechno nejlépe".

Lidi musíme brát jako partnery, nepovyšovat se nad ně, neotravovat, zabránit v nich pocitu ohrožení zaměstnání. Zásadně se vyhýbat vnitřním sporům (i mezi lidmi), zachovat „kamennou tvář".

Zásadně šetřit čas těch, jichž se dotazujeme. Vycházet z toho, že mají dost své práce, proto být na interview připravený, nachystat otázky. Je důležité, aby měli o interview zájem (lze vyvolat finanční motivací) a neodbývali jej, a také aby nebyli a priori proti (vhodný čas).

Osvědčují se CASE a formální systémy (Data Flow Diagram apod.), ale všeho s mírou – nesmí svojí formálností a případnou nesrozumitelností odradit.

Interview je záležitost psychologická – proto začít od toho, co dotyčný považuje za důležité.

Další rady:

- Neodrazovat technickými termíny.
- Jednat jako rovný s rovným.
- Velmi opatrně upozorňovat na rozpory v jeho požadavcích.
- Přesvědčit o shodnosti cílů.

- Opatrně zasahovat do mocenských vztahů (obzvláště nelokalizuje-li se kupovaný systém).
- Přesvědčit o vlastní kompetentnosti (reference, dávání souvislostí) – že se orientuji v problematice.

2.2 Ekonomické aspekty

2.2.1 Náklady na software

Dotkneme se spráného stanovení nákladů při tvorbě softwaru. Často se opomíjí zejména vedlejší náklady.
Celkově se dají náklady rozdělit na mzdy, režii, hardware, software, cestovné atd.

Vývoj. Náklady na vývoj tvoří zejména:

- manažeři, obchodníci
- úředníci (účetní, sekretářky), vrátní, nájem
- zástupci uživatele – závisí to na smlouvě (zejména za minulého režimu se praktikovalo „uplácení školením“)
- konzultanti, subdodávky
- programátoři
- dohled (audit), systémáci, QA – Quality Assurance (řízení jakosti – norma ISO 9000)

Další ztráty (náklady) jsou:

- daně
- pojištění
- ztrátové časy (školení)
- náklady na získání nových zakázek (propagace, úplatky)
- úroky spotřebovaných peněz

Vidíme tedy, že náklady zdaleka nepředstavují jen mzdy: pokud bereme jako základ mzdy (včetně odvodů) a amortizaci počítačů, s přihlédnutím ke „ztrátám“, jako obchodní činnost, vlastní růst, dovolené, administrativa apod., zjistíme, že potřebná tržba je minimálně 1,5–2 násobek platu, resp. trojnásobek čistého platu. U malé firmy, která má vlastní kancelář, stoupá tento násobek (režie) na 5–7, u velkých firem 10–20.

Náklady na instalaci. Sestávají zejména z:

- školení uživatelů
- konverze dat a databází
- kupovaný hardware a software (instalace)
- náklady na převzetí
- náklady na paralelní provoz (souběh se starým systémem)
- náklady vývojového týmu během instalace

Provozní náklady. (Mějme na paměti, že staré se ušetří.)

- Náklady na hardware a software (leasing, dohled).
- Mzdy (operátoři, systémáci, provozní programátoři).
- Údržba.

Poznámky. Je šikovné stát se dealerem softwaru, čímž se vstupní ceny sníží o 30–50%. (Oproti tomu jako dealer musím např. zajistit školení.)

Velké firmy mají tendenci být *systémovým integrátorem.*

K terminologii:

systémový integrátor = finální dodavatel: je výhodné vše nakoupit a dodat (vysoký zisk)

dealer = obchodník

distributor = velká firma, která dodává dealerům (např. Software Slušovice)

branch office = pobočka výrobce, je tendence fungovat současně jako distributor (tím se sníží ceny o zisk případného distributora), někdy jako systémový integrátor

Ztráty. Zejména ztráty při zadání mohou znamenat ohrožení firmy.

Je vhodné sepsat smlouvu tak, aby dodavatel neručil (příliš mnoho) za ztráty při selhání software, ale pouze se zavázal k opravě.

2.2.2 Analýza přínosů

Přínosy rozdělíme na dvě hlavní skupiny:

- **taktické** – projeví se v chodu firmy, aniž to ovlivní její celkovou politiku, zaměření atd.:
  - úspory lidí (není to to podstatné, ba někdy ani dobré)
  - úspory z oběhu (díky zlepšené informovanosti): snížení zásob (tím se uvolní značné prostředky!), rychlejší vyřízení objednávek, snížení výrobních časů
- **strategické** – vliv na strategii firmy apod.:
  - rychleji se mění výrobky (inovační doba se snížila z pěti let na dva roky)
  - širší využití poznatků pro řízení (sklady, prodejní trendy, kvalita prodejců)
  - menší závislost na jednotlivcích (plánovat musí jen jeden – ale když onemocněl, fabrika stála)
  - lepší podklady pro management
  - lepší služby zákazníkům (snáze se plní požadavky[12] a termíny)

2.2.3 Analýza rizik

Rozebereme dále možná rozika (s přihlédnutím k tvorbě software):

- krach dodavatelů hardware a software (popř. odchod z trhu)
- lidé onemocní, odejdou
- software neodpovídá specifikacím (k tomu vede předčasná formalizace)
- nedodržení termínů (a následné penále)
- odbory (odpor proti propouštění)
- další dodavatelé – podrazy
- špatná spolupráce se zákazníkem (není ochoten účasti)
- špatně připravený a kvalifikovaný tým
- změna podmínek u uživatele (nový management má jiný názor)

[12] Příkladem toho jsou automobilky, kde dealer převezme od zákazníka požadované parametry vozu, a vůz je smontován de facto na zakázku.

- skryté náklady („papírová válka" – namísto jednání se posílají obsáhlé spisy)
- uživatel nezvládne systém – buď na to nemá lidi (jeho chyba), anebo by znamenal příliš velké změny (software není dobře přizpůsoben), popř. jej využít nechce (osobní zájmy), nevěnuje mu pozornost
- systém je pomalý nebo nevhodný
- některá data nelze získat (utajovaná, nedostupná apod.)
- nedostatečná ochrana
- zákazník je ohrožen bankrotem

Zejména pro zkvalitnění styku se zákazníkem je třeba o všem (zejména o specifikacích apod.) dělat podrobné zápisy.

2.2.4 Odhady

Při rozbíhání projektu potřebujeme odhadnout zejména potřebu lidí, termíny, dynamiku týmu, rozpočet. Tyto odhady mají značný rozptyl, protože se zpravidla nemáme o co opřít. Bývá dále tendence podceňovat problém a přeceňovat vlastní možnosti, také je třeba odolávat tlaku šéfů. Odhad tedy jistým způsobem souvisí s riziky.

Pozor: je třeba rozlišovat *odhad* (tedy kolik to bude stát) a *dohodu* (tedy to, co vyjednám) – dohodnuté by nemělo jít pod odhad.

Zpravidla se bere odhad *platy* × *režie*, a výsledek se ještě vynásobí tzv. *bezpečnostním koeficientem*, který je zhruba okolo 2–3 a vyjadřuje některá rizika, jako např.:

- přeceňování možnosti přesčasů (v zásadě jen z nouze!)
- přeceňování možnosti využití zkušeností z jiných problémů – obvykle totiž není k dispozici dostatek údajů o minulých projektech (snad u velkých firem, ale ty mají režii okolo 20)

Je obtížné odhadovat sám sebe, proto je dobré mít nezávislou kontrolu.

Ukazuje se, že při odhadech se v jednotlivých fázích dělají zhruba tyto chyby:

po specifikaci cílů	±50%
na konci analýzy	±25%
před začátkem kódování	±10%
po naprogramování	±5%

S postupem práce na projektu se tedy chyba zmenšuje; u velkých softwarových balíků bývá menší.

2.3 Sledování kvality

Důležitou, a také, jak jsme si řekli, poměrně náročnou částí životního cyklu softwaru je sledování jeho kvality. Zejména se jedná o hledání a odstraňování chyb.

Při odstraňování chyb se doporučuje soustředit se zejména na:

- sběr údajů – přesně zaznamenat, kde je chyba, typ chyby (rozhraní), příčina atd.
- vyhodnocování – vhodné je např. použití spreadsheetů, popř. programu Statgraph.

Dále se zaměříme na jednu z používaných metod pro odhalování chyb.

2.3.1 Inspekce

je nejúčinnější metodou odstraňování chyb (je uváděna až 80–procentní úspěšnost). Cena za detekci chyby je přitom menší než 10% ceny při použití „klasických" způsobů. Jedná se o vnitřní oponenturu, která má institucionalizovaný průběh. Inspekce je použitelná pro všechny etapy životního cyklu.

Původcem metody je firma IBM, autorem je Fagan – zveřejnil ji v roce 1979.

Úkolem inspekce je tedy nalézt chyby, to znamená, že chyba v řešení nezůstane a nezvýší jeho cenu. Nalezení chyb je *společným zájmem* všech řešitelů. Úkolem není chyby opravovat (s výjimkou dokumentace pro uživatele), pouze ji nalézt a zařídit, aby se někdo postaral o její odstranění.

Problémem jsou ony „společné zájmy", tedy při inspekci „táhnout za jeden provaz". Znamená to mimo jiné, že ten, u koho je nalezena chyba,

1. se neurazí (není ješitný),
2. za to není postižen; tomu by měla být přizpůsobena pravidla v organizaci, tedy vedení by o tom ani nemělo vědět.

Inspekce se provádí v **týmu**, provádí se na *textech*, což může být nejen program, ale také specifikace, dokumentace apod. Složení týmu je následující:

1. **Moderátor.** Řídí práce, moderuje zasedání. Je to klíčová osoba, měl by mít cit pro detekci chyb (z průběhu inspekce „vycítit“, kde jsou).
2. **Zapisovatel.** Neměl by se příliš účastnit diskuse, aby se zbytečně nerozptyloval od zaznamenávání jejího průběhu.
3. **Předčitatel.** Může, ale nemusí to být sám autor.
4. **Oponenti.** Zpravidla jsou dva až čtyři.

Velikost týmu je volena podle odpozorovaného faktu, že více než osm lidí se zpravidla nikdy nedohodne.

Průběh inspekce. Z důvodu udržení maximální pozornosti se doporučuje zásadně oponovat jen tolik materiálu, aby byla inspekce zvládnutelná za 1–2 hodiny (odpovídá to 2–4 stránkám dokumentu).

Postup je následující:

1. V dostatečném předstihu (alespoň několik dní) před oponenturou se rozdají oponované materiály – obvykle na papíře (počítač zbytečně zužuje pohled). Stanoví se doba konání.
2. Členové týmu materiál prostudují.
3. Vlastní zasedání. Schůzi vede moderátor. Předčitatel podrobně prezentuje materiál – může to být sám autor, což má tu nevýhodu, že má tendenci vlastní chyby skrývat, ale na druhé straně je schopen odpovědět řadu otázek, zná souvislosti.

   Zapisovatel zapisuje chyby, a to ve tvaru:

   - **id** chyby
   - stručný název chyby nebo její charakteristika
   - popis chyby
   - čeho se chyba týká
   - ve kterém místě textu se nachází
   - možné efekty, resp. odchylky
   - kdo chybu napraví a dokdy.

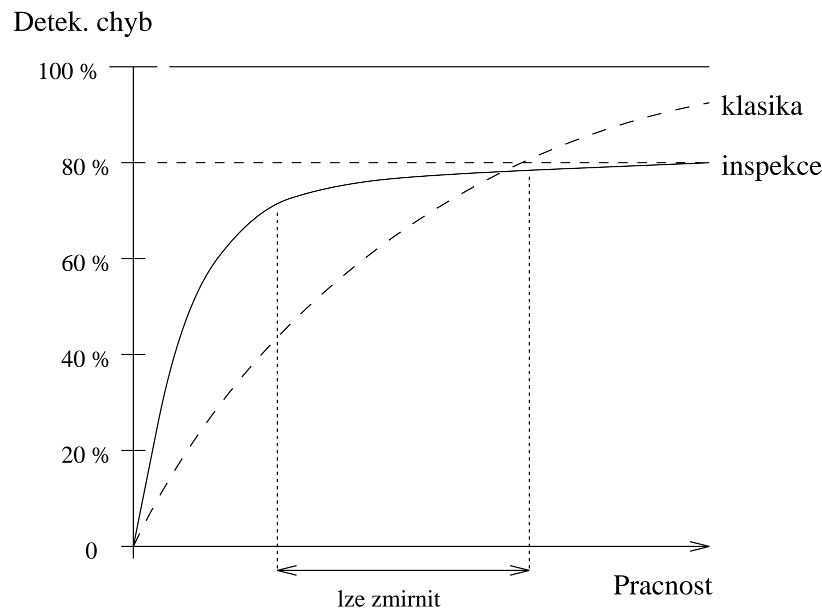
Zápis z celé inspekce má přitom tvar

- **id** inspekce
- čeho se inspekce týká
- místo a čas konání (kde, kdy)
- jména a funkce účastníků.

2.3.2 Zkušenosti s inspekcemi

Jak jsme si řekli na začátku, snižuje se použitím inspekcí počet chyb až na pětinu. Podmínkou úspěšnosti je přitom:

- dobrý moderátor
- ovzduší týmu – usilovná snaha chyby nalézt.



Obrázek 17: Srovnání detekce chyb klasickým testováním a inspekcí

Efekty. Klasické testování je oproti inspekci zpočátku méně účinné, avšak dokáže odhalit více chyb – pomocí inspekcí se nad 80% odhalených chyb nedostaneme (viz obr. 17). Vidíme, že část neefektivního klasického testování se dá ušetřit tak, že obě metody kombinujeme: zpočátku využijeme efektivnější inspekce, a posléze přejdeme ke klasickému testování (obrázek 18).

Výhodou inspekcí je, že zatímco klasické testování závisí na přeloženém programu, inspekce se dá provádět i „na papíře“ – bez toho, aby program běžel na počítači.

Slabá místa. Říkali jsme si, že úspěšnost inspekce silně závisí na osobě moderátora. Lidí vhodných jako moderátoři je však málo; problémem je také navození a udržení dobrého ovzduší v týmu. Obtížné je také testování inspektorů – jak zjistit (a jak včas zjistit), jestli pracují zodpovědně. Není možné je prověřovat až při testech produktů.

Varianty inspekcí. Aby se napravila některá slabá místa a nedostatky, používají se některé varianty metody inspekce, jako:

Formal review – čtenáři, naslouchající a zapisovatel.

Walkthrough – několik lidí „simuluje“ program – prochází text.

Faganova inspekce. Tou jsme se zabývali v předchozím textu.

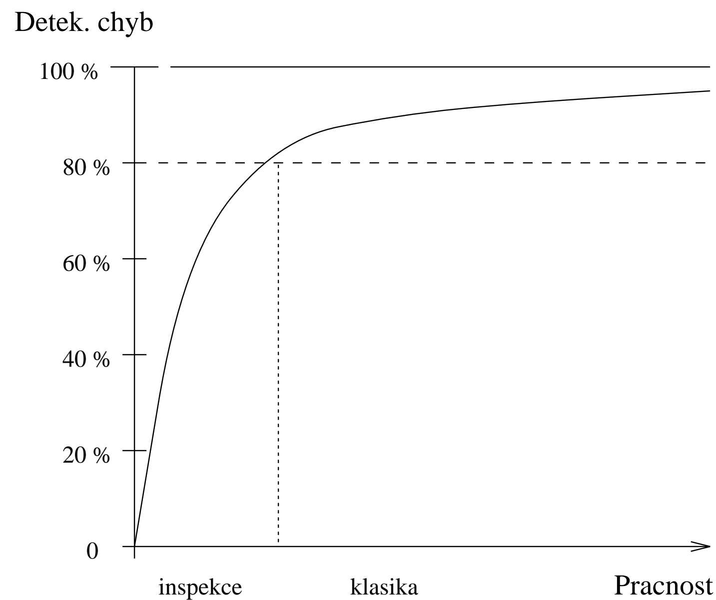
Active design review – aktivní testování návrhu. Je obvyklé např. při testování požadavků; cílem je zabránit povrchní práci inspektorů.

Kladou se otázky (např. autor se zeptá: zjistěte, jak se hledá v tabulce symbolů), potom se provádí inspekce pro jednotlivá kritéria (nezkoumá se dokument vcelku).

Provádějí se také **následné inspekce** – po odstranění chyb, často s jiným týmem, který kontroluje, jak byly odstraněny chyby, odhalené v minulé inspekci.

Cleanroom. Také má původ u IBM, rozdíly jsou tyto:

- nesmí se účastnit autoři
- poukazuje pouze na jednotlivá kritéria – např. rozhraní
- zasedání jsou velmi krátká.



Obrázek 18: Kombinace klasického testování a inspekcí

N–fold inspection – některá z předchozích technik se aplikuje tak, že se na dokumentu nechá pracovat několik nezávislých týmů (ale s jedním moderátorem).

Kritika. Zejména metody cleanroom a aktivní inspekce jsou kritizovány za to, že jsou zaměřeny pouze na detekci chyb, nikoli na kvalitu celku odpovídající normě ISO 9000 – ta totiž požaduje nejen bezchybnost, ale také sleduje kvalitativní ukazatele.

Proto je nutné podstatně zlepšit inspekce. Pouhá detekce chyb nestačí, sledujeme dále zejména

- udržovatelnost (zda se dají provádět modifikace)
- přenositelnost na různé HW/SW platformy
- vystopovatelnost (proč to tak funguje)
- znovuvyužitelnost částí
- efektivita.

Dále je v kritice poukazováno na to, že chybí kontrola účinnosti (kvalita zúčastněných – obtížně se pozná, když je to na nic). Rovněž chybí pevná pravidla pro hodnocení, zejména formální normy pro tvar programů (tvary identifikátorů apod.).

Nevýhodou je také neformálnost diskuse, slabší povahy proto obtížně prosadí své námitky (bojí se to říci). Posuzují se papíry, ne počítačovými prostředky (výjimkou jsou některé CASE systémy). Inspekce je pasivní, nevyužívají se znalosti toho, kdo řešení realizoval.

2.3.3 Vícefázové inspekce

Proto se používají *vícefázové inspekce*, které obsahují pokus o formalizaci. Součástí procedury by nadále měla být průběžná kontrola kvality práce inspektorů. K tomu se používají např. *zaseté chyby* (seed errors), tedy záměrně udělané chyby, zpravidla takové, které nenarušují logiku programu. Inspektor by je měl odhalit.

Principy vícefázové inspekce. Fáze zajišťuje, že produkt má jednu nebo několik málo přesně definovaných vlastností (např. tvar identifikátorů), ze kterých může další fáze vycházet. Vlastnosti jsou tedy prověřovány postupně, a proto musí být uspořádány.

Základem jsou tyto dvě fáze:

1. **Single inspector phase** – kontrolují se formální náležitosti textu (tzv. pretty printing, čitelnost, odsazování). Odpověď na tyto požadavky je v zásadě ano/ne.

   Dále se kontroluje použití indexů, tvar zkratek, přítomnost definic neznámých pojmů (zkratek), celková struktura dokumentu. Do této fáze rovněž patří plnění dohod o tvaru identifikátorů a konvencí o volání funkcí (parametry, boční efekty).

2. **Multiple inspector phase** – opět pracuje moderátor, zapisovatel a inspektoři. Úkolem je detekce chyb a sledování kvalitativních požadavků. Postup je následující:

   (a) Studium materiálů – inspektoři je studují *nezávisle*; odpovídají na otázky, které byly zadány předem (při předání materiálu bylo definováno, co se má sledovat).

   (b) Schůze, porovnání výsledků.

   (c) Zápis a vyhodnocení výsledků, odkud vyplyne i úspěšnost odpovědí, někdy také údaj, kolik chyb (zejména zasetých) inspektoři našli.

Výběr inspektorů. Zkušenosti ukazují, že na různé fáze (oblasti) se osvědčují různí inspektoři: na formalismy jsou dobří mladší členové týmu; starší lépe sledují souvislosti (vícekrát se poučili z vlastních chyb), ale jsou horší na programování a formality. Někteří z inspektorů by měli znát oblast aplikací.

Příklady fází. Jednotlivé fáze ve vícefázové inspekci mohou být zaměřena na následující kritéria:

F1 tvar dokumentu a pravopis

F2 pretty printing (je-li dokument tiskařsky dobře udělaný)

F3 čitelnost a mnemotechnika (netriviální je zejména volba obecných identifikátorů)

F4 dodržování dobrých programátorských praktik (málo `goto` a bočních efektů)

F5 dodržování předepsaných obratů (např. inicializace proměnných, zavírání souborů)

F6 multiple inspector.

Výsledky. Existuje i počítačová podpora – systém InspeQ. Při odhalení jedné chyby za jednu člověkohodinu se ušetří až 33 hodin při údržbě. Tyto zkušenosti však nejsou doloženy z většího počtu projektů.

Je dobré pověřit lidi, aby také zjistili, zda se dodržují normy. Také multiple inspector phase se dá provést různě.

2.4 Používání norem

Protože software **je** technický produkt, je také, jako každá technická oblast, určitým způsobem standardizován. Typickými normami jsou např. definice programovacích jazyků, nebo definice rozhraní pro periferie. Softwarové normy se v hojné míře používají zejména v USA; k nám se tento trend dostává také. Budeme tedy hovořit zejména o amerických zvyklostech v této oblasti, dotkneme se také norem obecně (nejen v software).

2.4.1 Vlastnosti norem a zásady jejich používání

Některé normy, např. technické, hygienické (ty zejména) jsou „z vyšší moci“ předepsány a jsou tudíž závazné, jejich nedodržování je pak sankcionováno. Budeme je označovat jako *de iure* normy; na druhé straně stojí normy *de facto*, které závazné nejsou, ale dobrovolně se dodržují (často přísněji než normy de iure).

Normy mají svoje nesporné výhody: sjednocují a stanovují pravidla, jak používat a spojovat různé produkty různých výrobců (typické jsou např. rozměry šroubů[13] nebo vlastnosti střídavého proudu), dále umožňuje „jednotný“ způsob využívání lidí (např. programovací jazyky).

Klíčovými vlastnostmi norem jsou proto **stabilita** a **jednotnost**.

Nevýhodou norem je oproti tomu určitá setrvačnost, způsobená zdlouhavostí tvorby norem. Díky tomu normy neodpovídají nejposlednějšímu stavu poznatků a jistým způsobem konzervují znormovaný stav. Proto dochází čas od času k *inovaci* norem.

Normy jsou přitom *veřejné* a jsou každému zdarma (za cenu okopírování) k dispozici. Použití norem je nezávazné, nejsou poskytovány žádné testy ani další služby. Mezi nejdůležitější zásady používání norem patří:

1. Používání norem je dobrovolné, je věcí dohody.
2. Normalizační úřady nenesou žádnou odpovědnost za případné škody, vzniklé použitím norem.
3. Norma vyjadřuje jistý stav znalostí, a proto zastarává. Proto by měla být (sofwarová norma) nejdéle po pěti letech inovována, resp. znovu potvrzena. V opačném případě, ač může být cenná (jako zdroj poznatků), by neměla být jako standard nadále doporučována.

Praxe je ovšem taková, že normy bývají často potvrzeny i bez nezbytné aktualizace.

2.4.2 Normotvorné aktivity

Na počátku vzniku norem jsou *producenti*, z nichž vychází *norma výrobce* (popř. podniková norma), která se stane de facto standardem. (Výrobci bývají přitom ovlivněni výzkumem a univerzitami.)

Pokud je takovýto standard hojně užívaný, národní normalizační komise vytvoří tzv. *pracovní skupinu* (working group), která připraví návrh normy. Po schválení národní normalizační komisí vzniká *národní standard*.

Práci národní normalizační komise může ještě předcházet činnost tzv. *profesních skupin* (např. na poli elektrotechniky a hardware je to IEEE), které na základě užívaných standardů vytvářejí též svoje normy. Z nich pak vychází pracovní skupiny národní normalizační komise.

Mezinárodní normalizační komise (International Standard Organization – ISO) pak vytváří, zpravidla na základě národních (tedy nejčastěji amerických) norem, normy mezinárodní. Rovněž ISO vytváří své pracovní skupiny, které se zabývají návrhem normy. Normy ISO zpravidla nejsou závazné.

Mezinárodní normy pak přebírají národní normalizační komise, provedou lokalizaci (překlad do domorodého jazyka a přizpůsobení normy), a tak se z nich stanou národní normy.

Celý proces je přitom zdlouhavý, trvá řádově několik let.

Standardy (normy) se na různých úrovních identifikují:

- de facto standard žádné zvláštní označení nemá, pouze výrobní značku (popř. u podnikových norem interní označení)
- profesní organizace má své označení (např. IEEE 9836.2), dávají tím doporučení normy
- národní norma obsahuje zkratku úřadu (státu) a číslo normy, běžné (známé) jsou
  - **v USA** normy ANSI (American National Standard Institut), popř. NIST (National Institut of Standards and Technology)
  - **v SRN** známé normy DIN (Deutsche Industrie Norme)
  - **v ČR** platí soustava norem ČSN (Československá státní norma)
- mezinárodní normy mají označení ISO
- převzaté normy se citují zpravidla pomocí národního označení (po lokalizaci normy) a pomocí mezinárodního označení normy, které odpovídají.

V naší republice eviduje normy (a též ochranné známky) Český normalizační úřad (ČNÚ), který rovněž soustřeďuje všechny normy ze světa.

[13]Existují dvě soustavy norem pro rozměry šroubů – metrická, a americká, tzv. Whitworthova, vycházející z palcové míry.

2.5 Normy v softwarovém inženýrství

Podrobněji jsou popsány v knize Software Engineering Standards, IEEE/Wiley 1984. Tyto normy vznikly původně jako dohoda řady výrobců a jiných organizací (kupodivu bez IBM); posléze vznikl „samozvaný“ výbor, který vytvořil návrhy norem, a ty byly přijaty ANSI.

Největší změny se přitom dají očekávat v oblasti kvality – asi bude sestavena norma na bázi ISO 9000, což je soustava doporučení, která zaručují kvalitu produktu. Zahrnuje tedy nejen požadavek, aby nebyly přítomny žádné chyby, ale také maximální uspokojení požadavků uživatele. K tomu je sestavena celá řada pravidel; u softwaru to zatím vyřešeno nebylo.

2.5.1 Význam norem při tvorbě softwaru

V oblasti software jsou normy potřebné zejména pro:

- programovací jazyky (dříve byly de facto standardy, např. Fortran)
- oblast komunikací (periferní rozhraní, komunikační protokoly)
- reprezentace dat (zobrazení v pohyblivé řádové čárce)
- standardní metody realizace softwaru (jsou pokusy o jeho normování).

Softwarové normy tvoří zejména IEEE, ANSI a ISO; ANSI tak standardizovala řadu programovacích jazyků, jako např. Fortran, Pascal, C, Adu a jiné.

Uživatelem sofwarových norem jsou samozřejmě uživatelé softwaru – programátoři, dále výrobci softwaru, ale také veřejnost.

Pro uživatele jsou přínosem zejména proto, že

- nemusí ztrácet dříve dosažené výsledky (návaznost na starší verze)
- normy jsou nástrojem kontroly prací
- jsou nástrojem kontroly dodržení požadavků
- umožňují včasnou detekci chyb.

Výrobcům přinášejí normy

- jednak totéž, co pro uživatele, kontrolu
- stabilitu podmínek realizace (neztrácejí se výsledky)
- nepřímo se vytváří tlak na zlepšení metodik.

Veřejnosti se pak nabízí

- snazší kontrola, zda dodávaný software je v souladu se zákony, zda je software neohrožuje nebo nepoškozuje (výpočet daní, finanční operace)
- kontrola, zda nedochází k plýtvání, a odtud snižování nákladů veřejných projektů.

Terminologie (Glossary of Software Engineering Terminology). Normy softwarového inženýrství zahrnují:[14]

- Plán kontroly kvality – Software Quality Assurance Plans.
- Řízení konfigurace (norma pro správu verzí) – Software Configuration Management Plans.
- Plán testů – Software Test Documentation.
- Návod na specifikace – Software Requirements Specification Guide.

[14] Anglické názvy vycházejí z knihy Software Engineering Standards.

- Slovník počítačových termínů.

Jako příklad si uvedeme podrobněji (Project Journal) např. řízení konfigurace. (Základem je Inspection—Audit—Walkthrough.)

1. Procedura vyhodnocování, příjmání nebo zamítání a koordinace změn po formálním vyhodnocení jejich identifikace v konfiguraci (proces).
2. Systematické vyhodnocování, koordinace, příjmání nebo zamítání změn a implementace všech přijatých změn v konfiguraci (prvku konfigurace) po formálním provedení identifikace změny v rámci konfigurace (dohled).

2.5.2 Plán zajištění kvality

neboli Quality Assurance Plan je další z normovaných oblastí tvorby softwaru. Měl by obsahovat tyto části:

1. Účel – čeho se týká, pro co platí, případně i pro co neplatí; zde vypsat příslušné moduly (komponenty).
2. Odkazované dokumenty – všechny by měly být explicitně uvedeny.

   U dokumentu se uvádí jeho identifikace, název (případně verze), také dosažitelnost (kde se najde, u souborů plná jména).
3. Management – tedy stanovují se úkoly. Tato část zahrnuje:
   - organogram – organizační schéma („pavouk“)
   - úkoly (podle etap)
   - odpovědnosti a termíny
4. Dokumentace. Ta je nutná pro provedení kontroly. Proto by zde měl být seznam dokumentů (viz „Odkazované dokumenty“), jak je možné je získat, a minimální dokumentace:
   - seznam požadavků, a to v kontrolovatelné formě, tedy precizně (popis jako „odpověď musí být *většinou* do jedné vteřiny“ tedy nevyhovuje)
   - popis návrhu (architektury SW)
   - plán verifikace a validace[15]
   - zpráva o verifikaci (u nás se většinou nedělá) – na základě inspekce vstupních údajů
   - uživatelská dokumentace
   - nepovinně: normy, manuály apod.
5. Normy, konvence a metody, a to zejména:
   - normy struktury dokumentu
   - normy kódování (identifikátory, volání podprogramů)
   - logická struktura komentářů
6. Review and audit. Těmito pojmy se rozumí

   review – vnitřní oponentura, kterou provádějí řešitelé mezi sebou

   audit – dohled, provádějí např. lidé z managementu.

   Minimálně by se přitom měly provést reviews (např. inspekce)
   - specifikace požadavků
   - předběžného návrhu
   - podrobného návrhu

[15] Při verifikaci se kontroluje, zda produkt odpovídá původním požadavkům, validace jej testuje.

- verifikace/validace

a vnější oponentury (kdy produkt předáváme vedení, nebo přímo zákazníkovi)

- funkční audit – jestli to funguje
- fyzický audit – kontrola kompletace dodávky
- přepadovky během řešení (in process audit), které průběžně kontrolují, jestli se dodržují termíny a rozpočet.

U větších firem (zhruba nad 30 zaměstnanců) má význam tzv. managerial review, tedy průběh projektu z hlediska managementu, odhaluje, kde se „ztratily“ peníze.

7. Řízení konfigurace (configuration management).
8. Metody evidence a oprav chyb.
9. Nástroje a metody realizace/kontroly (tedy speciality).
10. Kontroly – kódu, médií, dodavatelů.
11. Doba platnosti – v originále „how the documentation will be retained“, tedy jak se bude udržovat, jestli se bude obnovovat nebo rušit, a kdo to má na starosti.

2.5.3 Řízení konfigurace

Již jsme se zmínili o tom, že i na configuration management (neboli řízení konfigurace) existuje norma. Přitom víme, že pro dodávky softwaru široké použitelnosti máme v zásadě dvě možnosti:

1. parametrizovat jediný produkt (tedy customizace) – pokud to ovšem jde, tedy zejména pokud se příliš nemění logika
2. nebo mít produkt ve formě „stavebnice“, tedy skládat jej z více částí.

Parametrizace se přitom používá zejména u programových balíků, zatímco stavebnice jsou obvyklé u operačních systémů, a to především pro sálové počítače – mainframes.

Řízení konfigurace umožňuje např. SCGS, a částečně i známý program `make`.

Řízení konfigurace se opět skládá z několika částí:

Úvod :

- účel
- oblast platnosti
- definice, zkratky, odkazy

Management :

- organizace (vazby, procedury), odpovědnosti (kdo, kdy, za co)
- kontrola rozhraní – jak to spolupracuje
- způsob realizace (nástroje a metody)
- použité příkazy a procedury

Činnosti :

- identifikace konfigurace
- řízení konfigurace (co se kdy a jak bude měnit)
- vyhodnocování stavu
- reviews and audits – jestli byly včas odevzdány všechny dokumenty

Nástroje, metody :

- kontrola subdodávek

Organizace. Do oblasti organizace patří zejména

- vztah programátor—„řidič“ konfigurace
- organizační pavouk
- jak při vývoji/údržbě
- vztahy vývojář—uživatel

Zdůrazňuje se přitom zejména údržba.

Kontrola rozhraní. Důležitou oblastí je také kontrola rozhraní – interface control:

- identifikovat dokumenty
- postup přijetí změn
- sledování, realizace kontrolních dnů
- údržba interface a kontrola stavu interface

Pokud je totiž špatně navržené rozhraní, je to velmi závažná chyba.

Identifikace konfigurace. Tato část (configuration identification) by měla zejména vyjmenovat hlavní komponenty (podle etap), pojmenovat polžky pro review, pro schválení, a dále definovat formu účasti uživatele (měla by být součástí dohody).

Typické komponenty jsou:

- funkcionální – souhlas uživatele s funkcemi a jejich testováním
- allocated – souhlas s omezeními návrhu (co to nedělá), souhlas, že jsou dodrženy normy uživatele
- produkt – odsouhlasení (převzetí) hotového produktu

Předává se přitom

- jméno a verze
- číslo verze, způsob implementace
- návod na instalaci
- známé chyby a nedostatky (co tam nefunguje)
- média (na čem se to předává)
- případně termíny doplňků a změn.

2.5.4 Testování

Na začátku stojí dokumentace projektu a popis částí testu, a z nich se vytvoří *plán testů*. Ten se konkretizuje ve *specifikaci* testů, a po ošetření speciálních případů je test proveden. Výsledkem provedení je *test log*, tedy záznam testu, a *test incident report*, tedy seznam „průšvihů“. Na závěr se sestaví shrnutí, neboli *test summary report*.

Test incident je přitom popsán takto:

- `id`
- kdo, kdy
- stručný popis
- důsledky:

    - ⋆ pokračovat v testech?
    - ⋆ úpravy testů a software
    - ⋆ pokuny k úpravě (termín) – určí se kdo, ale ne jak

- rizika (může dojít ke škodám – zejména když se ovládá řízená soustava v průmyslu apod.)

3 Pel–mel

Do této kapitoly jsem zařadil zbylé přednášky, které bylo těžké shrnout pod nějaký společný název. Měla zde být i přednáška doc. Paulíčkové z FE VUT o biokybernetice a počítačových závislostech, ale vzhledem k její celkové zmatenosti nebylo možné ji nějak rozumně sepsat.

3.1 Architektura klient/server v databázích

Všichni známe princip této architektury: server je „celá aplikace“, spravuje data, běží jako jediný exemplář, předává data (po znacích) klientovi, který je zobrazuje uživateli. Klientů bývá více.

U databázového systému GUPTA to tak docela není: mezi serverem a klientem „běhají“ příkazy SQL, a na klientovi běží celá aplikace (plus prezentační – grafická část). To má svoje výhody:

- nezávislost na DB a na hardware a software serveru
- menší zatížení serveru
- využití výkonu klientů

ale také nevýhody, z nichž nejvýraznější je tzv. „překrmený klient“ (fat client): aplikace roste, až na ni klient ve své dosavadní konfiguraci nestačí, tím pádem je ale třeba provést posílení hardware (upgrade) *všech* klientů.

Řešení překrmeného klienta je takové, že se architektura klient/server dále rozdělí, a udělá se vyvážený systém se třemi komponentami:

1. Server(y)
2. Aplikační server(y)
3. Klienti

Znamená to ale, že musíme být schopni každou část aplikace „odeslat“ na libovolné místo sítě. Tento požadavek vypadá triviálně, ale míto provádění se tak může poměrně snadno měnit.

- **Orientaci na klienta** (aplikace běží na klientovi) představuje Gupta, a zčásti též Fox.
- **Orientaci na server** (aplikace běží na serveru) představuje Progress (spíše znakové).
- **Obecný případ** jsou např. Taligent, New Era (produkt Informix), který umožňuje i balacování výkonu – každý modul lze poslat kamkoli.

Poznamenejme ještě, že překrmený klient nemusí být až tak na závadu, pokud klienti buď nerostou, anebo je jich řádově málo.

3.2 Objektově orientovaná analýza a návrh

Zde budeme vycházet z knihy: James Rumbaugh, Michael Blaha, William Premeklani, Frederick Eddy, William Lorensen: Object Oriented Modelling and Design, Prentice Hall, 1991.

Podle této knihy vznikly nové metodiky, které byly přeneseny i do CASE systémů. Jedná se tedy o pokus o objektově orientované navrhování systémů.

3.2.1 Principy OOMD

Object Model zahrnuje statické vlastnosti modelů (atributy a vztahy); je to vlastně zobecnění Entity—Relationship diagramů. Dynamický model je pak zobecněním přechodového diagramu.

Funkcionální model je prakticky identický s Data Flow Diagramem – diagramem toku dat.

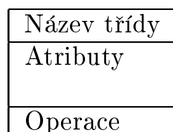
Tabulka v relačním modelu zde odpovídá třídě, atribut odpovídá atributu; pro operace není odpovídající protějšek v relačním modelu.

Principem přitom je o věci přemýšlet, a pak „ono to z toho nějak vypadne".

Obecně se požaduje, aby byly věci dokumentovány, a proto se dá předpokládat, že budou tyto nástroje používány.

3.2.2 Z čeho modelování vychází

Třída se kreslí jako obdélník:

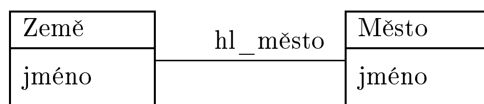


Třída odpovídá definici tabulky v databázi.

Atributy se uvádějí: `jméno, [typ, [popř. i defalult hodnota]]`

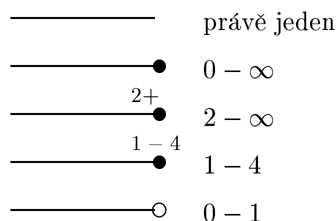
V některých fázích (jako je modelování) přitom stačí i pouhé jméno.

Zásadou je, že se uvádějí jen ty atributy, které jsou viditelné uživateli. Neuvádějí se implementačně závislé. (To je zásada, platná i pro běžné datové modelování.)



Obrázek 19: Příklad modelování tříd

Příklad tříd a vztahu mezi nimi je na obrázku 19. Operace zde v třídách nejsou, a proto je neuvádíme.



Obrázek 20: Typy vazeb mezi třídami

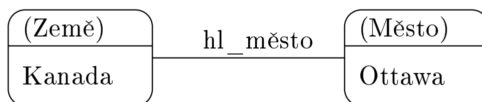
Typy vazeb (čili zejména násobnost, arita) se znázorňují u čáry, která značí vztah, a to způsobem, který shrnuje obrázek 20.

Instance tříd se píší jako obdélníky s oblými rohy (obrázek 21). Typ instance přitom nemusí být zaznamenán.

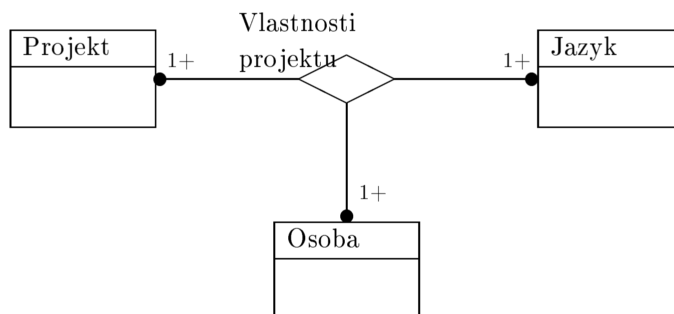
3.2.3 Definice n-árních vztahů

se provádí způsobem, který je naznačen na obrázku 22. Příklad instance, kde se osoba Jana účastní více projektů, je pak na obrázku 23.

Je přitom zásada převádět „více-ární" relace pokud možno na binární (protože tak se v nich lépe vyznáme).



Obrázek 21: Znázornění instancí



Obrázek 22: Model n-ární relace

3.2.4 Atributy relací

K relaci můžeme přiřadit atributy, jak to ukazuje obrázek 24. To je vlastně tabulka, v níž je uveden vždy uživatel, soubor, a jeho přístupová práva k tomuto souboru.

Jiný příklad vidíme na obrázku 25. Všimněme si zde, že u konce relace píšeme *role* (šéf, pracovník); relace „podřízen" (to je její název) má svoje atributy (zde: hodnocení); stejně tak relace „pracuje pro" má atributy, odpovídající pracovnímu zařazení (mzda a profese skutečně patří k tomuto pracovně–právnímu vztahu, ne k člověku).

Relace tedy začíná být velmi košatá: má svůj název, atributy, násobnost, nově pak role.

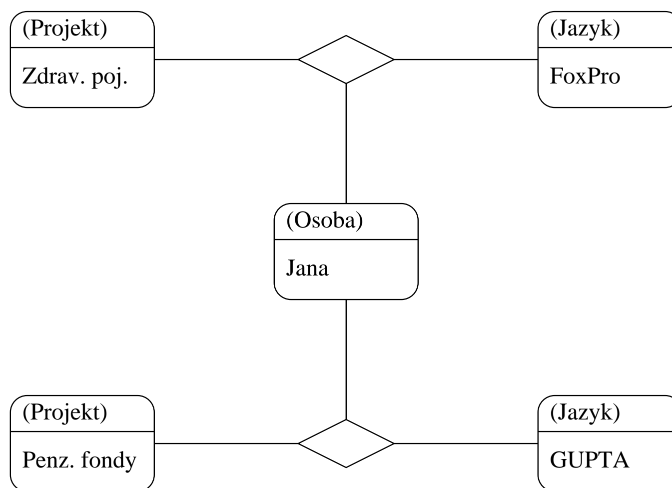
3.2.5 Implicitní násobnost relace

si ukážeme na příkladu (obrázek 26). Interpretace je zde taková, že hráč může v určitém roce hrát za více týmů. Za kolik týmů skutečně hraje, to se určí analýzou dat. (V klasické relační databázi zde máme položku tabulky, v níž je Hráč, Tým, Rok, a množina výsledků.)

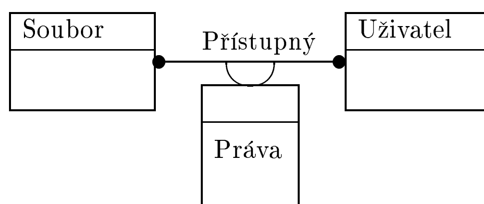
Doporučuje se, atributy patřící vztahu nedávat do tříd. Příklad máme na obrázku 27, kde jsme si vzali část příkladu z obrázku 25. První uvedený příklad je lepší, protože se snáze udržuje integrita dat. Také je na první pohled zřejmé, že atribut se týká smluvního vztahu mezi osobou a podnikem, a ne podnikem.

3.2.6 Co všechno lze specifikovat u relace

1. Jméno vztahu (relace).
2. Asociovaná třída.
3. Role – tedy specifikace významu relace u entit (doplňuje jméno). Příklad jsme měli na obrázku 25, kde relace podřízenosti měla role „šéf" a „podřízený" (pracovník).
4. Násobnosti u „konců".
5. Kvalifikace – k relaci připíšeme jméno atributu, který zajišťuje vazbu (pokud je známo uživateli).
6. Značka agregace – viz obrázek 28.
7. (většinou se nepíše) Doplňující podmínka – např. `{ ordered }` (že je to uspořádané)



Obrázek 23: Instance s vytvořenou n-ární relací



Obrázek 24: Atributy relace

3.2.7 Vyjádření dědění

Hlavními atributy objektovosti, jak víme, jsou:

- data a operace jsou obsaženy společně v objektu
- vznik a zánik objektů (persistence)
- overloadnig (polymorfismus)
- dědění, a to buď jednoduché, nebo násobné

V OOMD se dědění vyjadřuje způsobem, který ilustruje obrázek 29.

3.2.8 Rekurzivnost tříd

se vyajdřuje způsobem, znázorněným na obrázku 30. Je zde opět použita agregace, tedy model vztahů „skládá se z“.

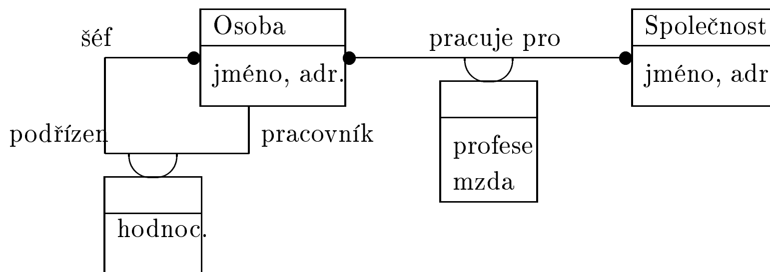
3.2.9 Abstrakce a abstraktní třídy

Abstraktní třídy shrnují společné vlastnosti „užitečných tříd“. Obvykle nemají instance. V našem příkladu s pumpami (obrázek 29) bylo takovou abstraktní třídou zařízení (výrobek). Shrnutí abstraktních tříd je na obrázku 31.

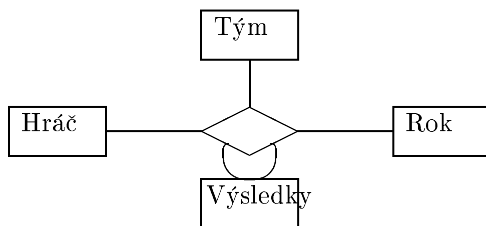
3.2.10 Objektový a dynamický model

Objektový model tvoří:

- třídy a vztahy, které tvoří zobecněný entity—relationship diagram



Obrázek 25: Příklad relací s atributy – zaměstnanci



Obrázek 26: Příklad ternární relace

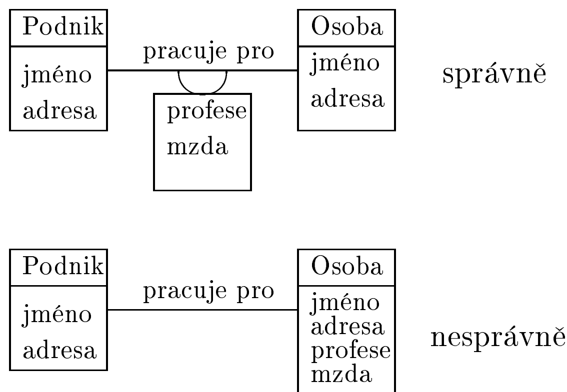
- diagram toku dat (DFD)
- dynamický model – přechodový diagram (podstatně zobecněný)

Tyto tři modely ale mezi sebou nemají dostatečné formální vazby. Nejsou tak například jasně (formálně) definovány operace nad třídami.

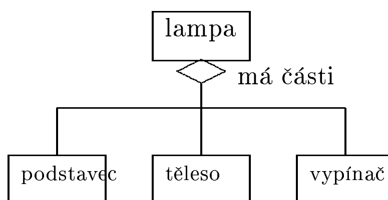
Otázkou tedy je, jak to udělat v SQL? Pracuje se proto na jazyce SQL3, ale na metodické úrovni to jasné není.

Dynamický model jsou vlastně vložené konečné automaty. Stav může být definován konečným automatem (viz obrázek 32). Jeden stav se tak rozvine do vícestavového diagramu.

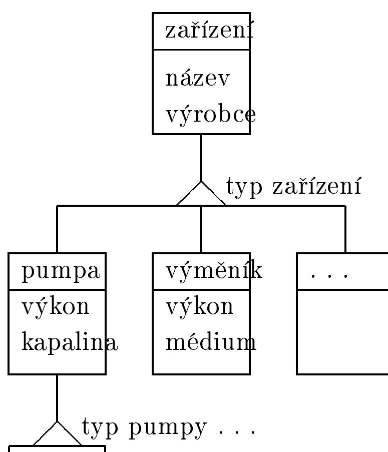
Jak vypadají v tomto modelu přechody: ukazuje nám to obrázek 33. Vidíme, že se stavem jsou asociovány tři skupiny podprogramů. Jsou to položky `do`, `entry` a `end`, tedy programy, které se provedou, pokud dojde k jakékoli



Obrázek 27: Správné a nesprávné použití atributů



Obrázek 28: Použití značky agregace



Obrázek 29: Vyjádření dědění

změně vstupních parametrů. Dále zde máme *datové prostředí*, ve kterém automat pracuje, *třídu*, která je asociována s přechodem, a událost s atributy, což je něco, co se provede při podmínce; pak se provede akce. Celkem tedy máme

```
událost(atributy)[podmínka]/akce
```

čili zprávu (událost), podmínku odeslání zprávy, a akci, která se provádí při přechodu.

Uvedeme si ještě příklad, na kterém je podprogram pro řízení topného systému (obrázek 34).

3.3 Superpočítače

Toto je přednáška dr. Matysky, která měla být věnována superpočítačovému centru MU, ale byla pro nás spíše opakováním ze začátku zimního semestru, kdy jsme paralelní výpočty probírali.

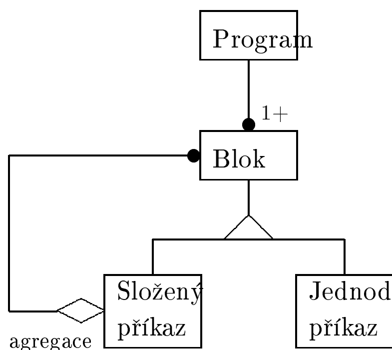
3.3.1 Architektury superpočítačů

Architektury superpočítačů se používají zejména pro náročné numerické výpočty. Roste přitom i jejich praktické nasazení – například předpovědi počasí.

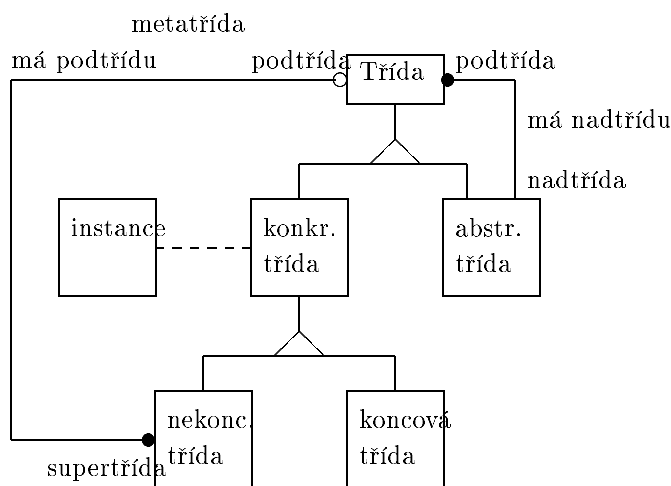
Rychlost se udává v jednotkách MIPS (milióny instrukcí za sekundu) a MFLOPS (milióny instrukcí v pohyblivé řádové čárce za sekundu). Ve skutečnosti však také záleží na typu instrukcí, které procesor vykonává: dělení je mnohem náročnější než sčítání a násobení. Většinou se udává *teoretická* rychlost.

Donedávna platily za výkonné např. počítače typu VAX 11/780 s rychlostí kolem 1 MFLOPS; dnes již procesor Pentium má výkon destinásobný.

Zřetězené systémy RISC, pracující na principu superpipelining, mají výkon řádově stovek MFLOPS (POWER 2 266 MFLOPS, DEC 21064A 175 MFLOPS). Nastávají ale problémy při vývoji, způsobené vysokými kmitočty (stovky MHz). Výkon se často zvyšuje skládáním více floating point jednotek.



Obrázek 30: Rekurzivnost tříd



Obrázek 31: Abstraktní třídy

Vektorové procesory (o kterých jsme hovořili v odstavci 1.1.2) pracují nad vektorem.[16] Nejznámější je na tomto poli firma CRAY Research, jejíž počítač X–MP měl výkon 250 MFLOPS, Y–MP pak 1 GFLOPS.

Spojuje se proto více procesorů dohromady, což vede k zapojení architektur SIMD a MIMD. Zatímco SIMD představují vektorové počítače, u architektury MIMD si každý procesor „dělá co chce“, a určitým způsobem se synchronizují. Problémem však je organizace paměti:

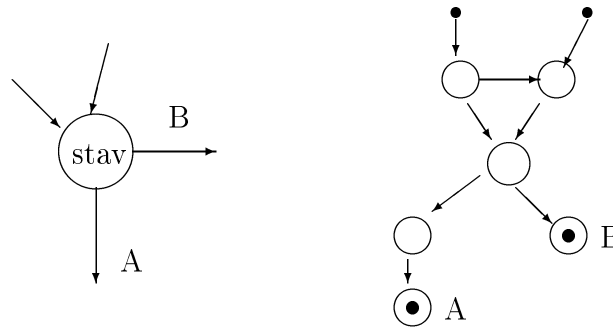
1. Buďto má každý procesor stejný (stejně drahý) přístup do jedné, globální paměti – je to tzv. *shared memory.*

2. Anebo má každý procesor svoji lokální paměť, a sekundárně pak přístup buď k paměti jiného procesoru (za cenu velkého zdržení nebo vůbec), případně distribuovaná paměť a sdílení zpráv. Distribuovanou sdílenou paměť pak představuje architektura NUMA – Non Uniform Memory Architecture.

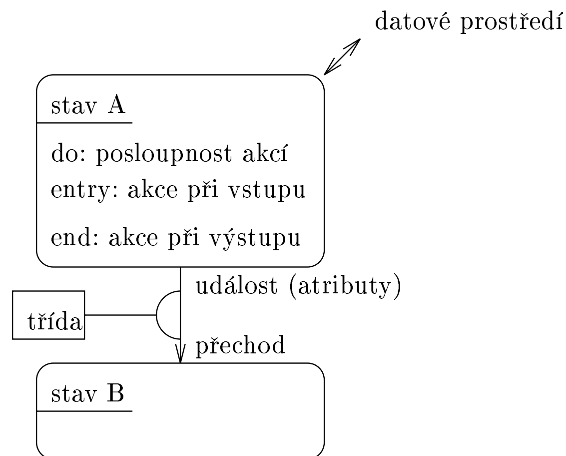
Oba systémy jsou schematicky znázorněny na obrázku 35.

Zkušenosti ukazují, že se systém snáze programuje, pokud je distribuovanost schována. (Distribuovaná paměť je však levnější – tak funguje i lokální síť.)

[16] Je ovšem problém vektorový počítač efektivně naprogramovat: to lze rozumně jen u úloh, které jsou snadno vektorizovatelné.



Obrázek 32: Definice stavu



Obrázek 33: Přechod ve stavovém diagramu

3.3.2 Rozšířitelnost

Zajímavý počet procesorů se pohybuje řádově ve stovkách. Ideální by bylo, aby n-krát větší úlohu zvládl ve stejném čase n-krát větší počítač. Snaha tedy je, mít při přidání procesoru téměř lineární růst výkonu i ceny.

Je-li rozšířitelnost *(scalability)* v mezích řádově 1000 procesorů, je již systém prakticky „neomezený“. Růst ceny přitom závisí na propojení a na režii celého systému.

Podle propojení rozlišujeme:

přímé sítě , kde je každý procesor spojen se sousedy; pokud si s nimi vystačí, je systém dostatečně rychlý

nepřímé sítě , kde jsou procesory odděleny, a při komunikaci se vždy musí přejít do globální přepínací sítě; zátěž se tak ale dá mnohem lépe globálně rozprostřít.

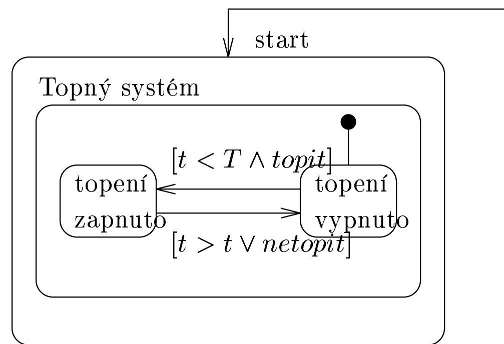
Při komunikaci se pak používají techniky

přepínání paketů (packet switching): paket je samostatná jednotka; pakety mohou přitom cestovat různými směry, ale také v různém pořadí, proto musí být „chytřejší“ protokol, který je uvede do správného pořadí

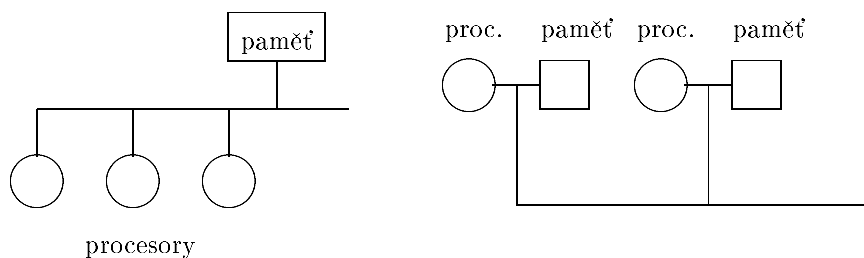
virtuální okruhy (virtual circuits): pokud chtějí dva procesy komunikovat, otevřou si virtuální okruh; otevření nás tak stojí zvýšenou režii, ale komunikace je pak rychlejší, a data se dopraví přímo ve správném pořadí

Používá se technika „store and forward“, tedy „přeber a pošli dál“ (tedy posílám až celou zprávu), nebo „červí díra“ (wormhole), kdy posílám zprávu dál už při obdržení první části.

Směrovací cesty mohou být statické a dynamické (adaptivní), podle toho, jestli se propojení průběžně mění.



Obrázek 34: Příklad: regulátor topení



Obrázek 35: Organizace paměti v systémech MIMD

Topologie sítě určuje její chování. Sledujeme také tzv. *latenci*, což je zdržení při přístupu. Ideální je latence konstatní, ale zpravidla bývá logaritmická.

Různé topologie jsme probírali v odstavci 1.2.2; připomeňme si

hyperkrychle , která je zajímavá např. jednoduchým směrovacím algoritmem – zprávu směrujeme do uzlu, jehož adresa se liší od naší o jeden bit, a je blíže adrese cílové; nevýhodou je, že procesory nemůžeme přidávat jednotlivě, vždy jen vzrůst na dvojnásobek, a je potřeba velké množství spojů

torus (anuloid) neboli k-ární n-krychle

Zajímavá je také *sběrnice*, která je velice příjemná, avšak málo průchodná. Cena je totiž konstatní. Některé systémy mají proto sběrnic několik.

3.3.3 Paměť typu cache

Procesor je většinou schopen pracovat podstatně rychleji, než jak je mu paměť schopna dodávat data. Proto má procesor vnitřní registry, anebo (protože registry nestačí) se mezi procesor a paměť dává rychlá paměť – *cache* (čti „keš“).

V cache jsou zpravidla bloky fixní délky, které mapují bloky hlavní paměti. Důležité je procento úspěšnosti při čtení z cache – zhruba $80 - 90\%$ je dostatečné k tomu, aby se nám systém jevil, jakoby pracoval jen s cache.

V části případů se ale data v cache paměti nenacházejí; tyto neúspěchy *(misses)* mají tyto příčiny:

- **Compulsory** –první přístup, kdy v cache nic není
- **Capacity** – potřebujeme více dat, než je cache schopna pojmout
- **Conflict** – konflikt v umístění: nastává, pokud se určitý blok hlavní paměti mapuje na určené místo cache (tento nedostatek odstraňuje asociativní paměť)

- **Coherency** – jen ve víceprocesorových systémech: jeden z procesorů zapíše něco do bloku, který má ve sé cache, údaj v cache druhého je tak neplatný

Posledně jmenovaný problém se řeší zavedením tzv. *adresářů*, kdy si pamatujeme, ve které cache je který blok. Bloky pak mohou být uncached, shared, dirty.

Jiná možnost je, mít adresář u cache, která si pak pamatuje, které jsou v ní bloky.

Objevuje se **zdržení** (latency), které je dáno tím, že paměť pracuje pomaleji. Dá se odstranit

1. pomocí cache:
   (a) maximalizace úspěšných čtení z cache
   (b) minimalizace ceny neúspěšných čtení z cache
2. zakrytí: pokud máme k dispozici superpipelining, nemusíme vědět o tom, že potřebujeme data „odjinud", protože procesor data mezitím připraví; případně se mohou přepínat kontexty (úlohy), což je ale dost šílené (např. procesor SPARCLE)

Uvedeme si ještě pojem **cache only memory**, což znamená, že chápeme veškerou paměť jako cache, což umožňuje hardwarová podpora kopírování potřebných bloků do lokální cache.

V těchto architekturách je nepříjemné ladění: k přerušení totiž dochází asynchronně k instrukcím; musíme proto zakázat pipelining, čímž jde ale rapidně dolů výkon.