SLT 2002

Jan Kasprzak, Petr Sojka (editoři)

SLT 2002

sborník třetího ročníku semináře o Linuxu a TEXu – SLT 2002 Seč, 7.–10. listopadu 2002

KONVOJ CSTUG CZLUG

Editoři

Jan Kasprzak a Petr Sojka
Fakulta informatiky, Masarykova univerzita
Botanická 68a
602 00 Brno
E-mail: kas@informatics.muni.cz, sojka@informatics.muni.cz

ISBN 80-7302-043-2

Předmluva

Semináře o Linuxu a TEXu pořádané Českým sdružením uživatelů operačního systému Linux (`http://www.linux.cz/czlug/`) a Československým sdružením uživatelů TEXu (`http://www.cstug.cz/`) jsou významnou událostí v životě komunity uživatelů TEXu i Linuxu. Letošní, v pořadí již třetí seminář SLT má výrazně větší počet přednášek než v minulém roce. Velikost sborníku, který držíte v ruce, tomu také odpovídá. Organizátoři se totiž rozhodli pro částečný návrat k paralelním sekcím o TEXu a o Linuxu, jako tomu bylo na prvním SLT v roce 1998. Na letošním SLT 2002 je program částečně rozdělen do dvou sekcí, kde by měly být specifické a odborněji zaměřené přednášky, druhou část programu pak tvoří přednášky společné, které by měly rozšířit obzory obou komunit účastníků.

Webovou podobu tohoto sborníku postupně najdete na stránce semináře na adrese `http://www.cstug.cz/slt/02/`. Za vznik sborníku patří hlavní dík zejména Petru Sojkovi, který se ujal náročného úkolu sesazení jednotlivých prezentací do jednotného stylu a vytvoření výsledné podoby sborníku, a Davidu Antošovi, který mu byl s tímto úkolem nápomocen.

Organizační výbor konference SLT v poměrně krátkém čase dokázal sestavit velmi bohatý program. Doufejme tedy, že přednášky zaujmou účastníky konference a seminář bude úspěšný nejméně tak, jako tomu bylo v minulých dvou ročnících. Přeji účastníkům semináře, aby akce byla pro ně nejen obohacením po odborné stránce, ale i setkáním s lidmi podobného odborného zaměření a místem pro navázání nových kontaktů.

Brno, listopad 2002 Jan Kasprzak za organizační výbor SLT.

Obsah

I TEX

II Linux

Část I

TEX

“Go forth now and create
masterpieces of the publishing art.”
D. E. Knuth: TeXbook, strana 303

Automatizace sazby a skenování formulářů

Miroslav Hrad[1], Petr Sojka[1,2]

[1] Masarykova univerzita v Brně, Fakulta informatiky*
Botanická 68a, 602 00 Brno
Email: xhrad@fi.muni.cz sojka@fi.muni.cz
[2] AMD, v. o. s., Návrší Svobody 26, 623 00 Brno
Email: psojka (at) mistral.cz

Abstrakt: V článku je popsána technologie SCAT pro přípravu a skenování formulářů, která byla v minulých třech letech použita na Masarykově univerzitě a v dalších organizacích na zpracování téměř statisíce formulářů vysazených TEXem a zpracovaných za použití nástrojů otevřených systémů. Jsou diskutovány prvky navrženého systému směřující k minimální chybovosti (použití čárových kódů), rychlosti, anonymitě, maximální bezpečnosti a automatizaci zpracování.

Klíčová slova: SCAT, skenování, rozpoznávání, formuláře, čárové kódy, TEX, Linux, automatizace testů

1 Úvod

Vkládání dat člověkem do počítače je činnost častokrát časově náročnější než jejich samotné počítačové zpracování. Ruční vkládání dat navíc s sebou přináší velké procento chyb a jejich odstraňování bývá často netriviální záležitostí. Nejinak je tomu i v případě vytváření, zpracování a vyhodnocování písemných testů.

1.1 Potřeba automatizace

Vytváření, zpracování a vyhodnocování testů je běžnou stereotypní prací všech, kteří zadávají písemné zkoušky na všech typech škol. Může se jednat jak o obyčejné zkoušení studentů – třeba na střední škole, tak o přijímací zkoušky na vysokou školu.

1.2 Východiska

Použití všech nabízených komerčních řešení dostupných na trhu selhávalo na nemožnosti přizpůsobení konkrétním požadavkům. Outsourcing každodenních potřeb se nejeví jako ideální stav. Vhodnost použití „in-house“ řešení, jak se

* Výzkumný záměr CEZ:J07/98:143300003

Jan Kasprzak, Petr Sojka (editoři): SLT 2002 – sborník semináře o Linuxu a TEXu, str. 11–26, 2002.

ukázalo později, umožňuje *mnohem* vyšší komfort a flexibilitu vyhodnocení testů než řešení založené na uzavřených systémech.

V rámci série bakalářských a diplomových prací [4,2] se postupně vyvíjely algoritmy a ověřovala se technika potřebná pro realizaci automatizace zpracování agendy příjímacího řízení nebo vyhodnocení písemných testů. Postupným vývojem a dlouhým testováním systému se nakonec ukázalo, že je možno vytvořit spolehlivý systém a technologii nevyžadující vysoce specializovaný a drahý hardware.

1.3 Členění článku

V oddíle 2 je definován proces vytváření a zpracování formulářů a jsou popsána východiska projektu. V sekcích 3 a 4 je diskutována problematika návrhu a sazby testů s použitím technologie čárových kódů. Oddíl 5 rozebírá algoritmy rozpoznávání symbolů a textů ve formulářích. V oddíle 6 je pak zmíněn způsob vyhodnocení a zajištění anonymity zpracování až do finálního zveřejnění výsledků testu.

2 Proces vytváření a zpracování formulářů

Technologie pro automatizované vyhodnocování testů s využitím skeneru s podavačem musí řešit tyto etapy:

1. analýza specifických požadavků projektu
2. vytvoření testů a jejich testování, sazba
3. návrh odpovědních formulářů
4. sazba a tisk odpovědních formulářů
5. distribuce a sběr testů a formulářů
6. skenování formulářů
7. vyhodnocení naskenovaných formulářů
8. vyhodnocení výsledků testu a jejich prezentace
9. vyhodnocení statistických parametrů testových otázek a příkladů
10. zatřídění formulářů se zadáními pro dlouhodobou archivaci.

2.1 Technologie SCAT

Při vzniku projektu SCAT se vycházelo z dosavadního způsobu vyhodnocování přijímacího testu na fakultu informatiky a pedagogickou fakultu MU v Brně. Byl používán výsledkový list, který měl dvě části. Část s identifikačními údaji a část s oválky pro odpovědi na otázky. Tyto dvě části byly oddělitelné a logicky je spojovalo pouze shodné, předem vygenerované číslo. Tento způsob byl použit právě z důvodu anonymity vyhodnocování.

2.2 Cíle a požadavky při návrhu systému SCAT

Pro projekt jsme si stanovili následující cíle a požadavky na systém:

- flexibilní vytváření vlastních testových formulářů,
- jednoduchá integrace vytvořených formulářů do systému zpracování,
- bezchybné rozhodování – zpracování testů,
- zvýšení rychlosti zpracování,
- vytvoření logistiky zpracování pro přijímací zkoušky,
- zaručení bezpečnosti – neovlivnitelnosti výsledků,
- standardizovaný výstup, aby bylo možné výsledky exportovat do současných systémů,
- jednoduché ovládání,
- otevřenost systému vůči novým prvkům a přístupům,
- minimální hardwarové požadavky,
- případná přenositelnost na jiné systémy, standardně bude použit operační systém Linux.

Požadovali jsme systém, který umožní vyhodnotit *více správných odpovědí*, tedy bude zpracovávat každý vyhodnocovací prvek – čtvereček a určí o něm jeho stav (vyplněn/nevyplněn) bez ohledu na ostatní odpovědi v dané otázce.

2.3 Výpočetní technika a software systému SCAT

Požadovali jsme, aby bylo v maximální míře využito všeobecně dostupného hardware a software, s maximálním podílem otevřeného software, což zajišťuje vysokou míru přenositelnosti a rozšiřitelnosti.

Využíváme běžných počítačů PC s operačním systémem Linux a skenerů běžné cenové kategorie. Nárazové nasazení (přijímací zkoušky) systému tedy nevyžaduje (s možnou výjimkou skeneru) vyšší finanční nároky.

Pro implementaci jádra systému SCAT jsme zvolili jazyk C. Ostatní komponenty (především logistika systému) je napsána ve skriptovacím jazyce Perl. Sazba všech výstupů včetně formulářů je realizována sázecím systémem TEX a programem METAFONT.

3 Návrh vyhodnocovacího formuláře

Významným omezením některých softwarových produktů pro vyhodnocování testů je fixní množina předtištěných formulářů – například Recognita Test. Naším cílem bylo navrhnout technologii dostatečně flexibilně, bez omezení na počet otázek, odpovědí, textů či způsob identifikace.

Při vyhodnocování písemných testů pro přijímací řízení je nutné maximálně eliminovat možnost záměrného zásahu do objektivnosti zjištěných výsledků, tedy mimo jiné dosáhnout anonymity. To lze zajistit například tím, že oddělíme odpovědní část formuláře od části identifikační. Obě části pak zpracováváme samostatně nebo identifikační část zpracováváme až po zpracování části

odpovědní (ve stavu, kdy už nelze zjištěné výsledky ovlivňovat). Tento způsob zpracování ovšem vyžaduje navrhnout formuláře tak, aby části byly jednoduše oddělitelné (lze zajistit třeba perforací formuláře), ale především, abychom byli schopni přiřadit výsledky z odpovědní části k identifikaci. Jedním ze způsobů, jak zajistit onu *logickou vazbu*, je obě části vybavit shodným číslem – *vazebním číslem*. Je zřejmé, že jednotlivé testové formuláře pak musí mít vždy vzájemně rozdílná vazební čísla. Z této nutnosti plyne potřeba generovat a tisknout jednotlivé formuláře, naopak není možné formuláře kopírovat. Vazební číslo je vhodné navrhnout tak, aby se dalo využít technologie čárových kódů [1], neboť rozpoznávání čárových kódů je při dodržení určitých podmínek výrazně méně chybové než technologie OCR[3].

V případě, že není nutné zajišťovat anonymitu v průběhu vyhodnocování, a lze tedy zpracovávat formulář, který obsahuje jak odpovědi, tak identifikaci, pak mohou být testové formuláře vzájemně shodné a lze vytvářet jejich kopie a ty potom zpracovávat. Takový formulář využijeme třeba při běžném písemném zkoušení. Při vytváření nám odpadne potřeba vazebních čísel a také vyhodnocení je jednodušší, identifikace i odpovědi se zpracovávají zároveň – při jednom průchodu skenovacím zařízením.

4 Sazba formulářů testů

Na rozhodnutí, jakým způsobem a kterým sázecím systémem budeme vytvářet testové formuláře, měly vliv následující potřeby:

- automatické generování formulářů, jejich čísel a čárových kódů,
- možnost převedení informací z výstupu na souřadnice pro systém,
- nezávislost přípravy formulářů na operačním systému.

Sazba čárových kódů *závisí* na použitém výstupním zařízení a tiskové technice (laserová tiskárna, ofsetový tisk) a je třeba provést *kalibraci* (bar correction) tlouštěk čar sázených čárových kódů.

4.1 Návrh testů

Nejsnazším způsobem je test navrhnout jako sadu příkladů s možností výběru z N správných řešení, „multiple-choice test“, N obvykle 3–5. Pro minimalizaci možnosti opisování vedle sebe sedících zkoušených je vhodné mít testy generované zcela individuálně [7], nebo mít alespoň několik variant testu s vzájemně permutovaným pořadím otázek pro znesnadnění podvádění.

Detailní rozbor návrhu obsahu testů je mimo rozsah tohoto příspěvku, každopádně je vhodné softwarově podporovat alespoň statistické vyhodnocení složitosti jednotlivých příkladů z realizovaného testu.

[3] Optical character recognition

4.2 Generování testů

Až na výjimky je třeba připravit různé, případně i plně individuální, verze testových odpovědních formulářů nebo i samotných testů. Zejména pro možnost vysázení individuálních testů (databázového publikování [11]) je ideální a plně se osvědčil sázecí systém TEX[4]. TEX dokonce může sám permutovat příklady [7] a zapisovat soubory pro vyhodnocení, ale toto lze většinou realizovat místo složitého makroprogramování externím skriptem ve skriptovacím jazyce (obvykle Perl).

Ani sazba čárových kódů EAN nepředstavuje problém, existují například makra popsaná v článku [6]. Pro námi požadovaný rozsah kódovaných čísel postačuje například čárový kód EAN-13, používaný i v systému ISBN.

Při návrhu formuláře musí existovat způsob, jak komunikovat prvky formuláře a jejich umístění vyhodnocovací části systému. V otevřeném a dobře dokumentovaném systému, jako je TEX, to není problém.

4.2.1 Vložení vlastní informace pro generovaní souřadnic Do zdrojového souboru pro sazbu TEXem můžeme vložit příkaz `\special` [3, kapitola 21]. Tento příkaz, jehož argument nebude sázen, je beze změny přenesen do souboru `DVI`.

Pro automatický vznik souřadnic a pro správné ohodnocení odpovědních čtverečků potřebujeme do zdroje pro sazbu vložit informaci o počtu odpovědí v otázce. Tedy např. `\special{pocet_ctv 3}` znamená, že budou následovat otázky se třemi odpovědními čtverečky – *A, B, C*. Tuto informaci musíme vložit před každou změnu počtu odpovědí. Standardně je možné nastavit až deset odpovědí v rámci jedné otázky.

4.2.2 Soubor `DVI` Výstupní soubor TEXu je binární soubor `DVI`[5], který čtou ovladače jednotlivých výstupních zařízení, například ovladač obrazovky, ovladač tiskárny. Jeden z takových ovladačů je program `dvitype`, který převede všechny informace z binárního souboru do čitelné podoby – textového souboru, který obsahuje všechny údaje o pohybu bodu sazby a o kódech jednotlivých znaků v relativně srozumitelné podobě.

Pokud se provádí sazba znaku například povelem *set_char_i*, znak se umístí tak, aby se jeho referenční bod kryl s bodem sazby a po vykreslení znaku se bod sazby posune o šířku znaku doprava. K tomu ovladač potřebuje znát šířky všech znaků, které jsou uloženy v `TFM`. Ovladači může postačit formát `PK`, protože tam jsou údaje o šířkách znaků (nikoli však o výškách a hloubkách) duplikovány se stejnou přesností jako v `TFM`.

4.2.3 Shrnutí vzniku systému souřadnic Zaměřovací prvky, zaškrtávací prvky a prvky pro identifikaci vytvoříme v METAFONTu, jako METAFONTové znaky. Na obrázku 1 jsou zobrazeny připravené METAFONTové znaky. Ty pak

[4] Používáme distribuci TEXLive

[5] `DVI` – Device Independent – formát nezávislý na zařízení

můžeme jednoduše používat pro tvorbu nových testových formulářů. Příklad sazby formuláře je uveden v příloze na straně 25. Pomocí sázecího systému TEX získáme obraz vysázeného formuláře ve formátu `DVI`. Z formátu `DVI` jsou pak zjistitelné pozice sazby na straně s využitím programu `dvitype`.

Obrázek 1. Prvky v METAFONTu

Například víme, že zaškrtávací prvek je v METAFONTu reprezentován jako znak číslo dvě v daném fontu. Zvolíme úroveň výpisu programu `dvitype` alespoň tři a ve výpisu potom hledáme řetězec „`setchar2`“, např.:

```
4654: setchar2 h:=2660904+792488=3453392, hh:=219
```

Tímto získáme horizontální souřadnici h v kartézském systému souřadnic (h, v), resp. přímo souřadnici hh v pixelech. Souřadnici v, resp. souřadnici vv v pixelech, najdeme ve výpisu jako poslední změnu vertikální pozice před místem sázení znaku „`setchar2`“, např.:

```
4649: push level 6:(h=-2797019,v=15156135,w=0,x=0,
y=0,z=0,hh=-177,vv=960)
```

Takto můžeme programově získávat údaje z formuláře do konfiguračního souboru skenovacího programu.

4.2.4 Specifikace požadavků tisku Protože jde o relativně velké tiskové objemy, je třeba si před hromadným tiskem odsouhlasit přesný seznam formulářů a jejich parametrů. Ukázalo se nanejvýš vhodné, aby se z jednoho a téhož souboru tiskly jak zkušební sada formulářů, tak jejich seznam s jejich počty. Vše je zadáno v textovém souboru, ze kterého se údaje čtou pro různé výstupy (stačí příkazem `\let` změnit před načtením chování hlavního makra).

```
\datum{červen 2002}      % nastavení implicitní hodnoty data
\setcounter{nanswers}{4} % počet možných odpovědí
\vartext{Fakulta X}      % variabilní text
\idtext{}                % identifikační text
\id = 0                  % počáteční nastavení čítače čísel testů

\obor{Písemný test zaměřený na ověření obecných
studijních předpokladů}
\studium{\TeX u a Linuxu}
\sada 10 1 99 1000       % počet testů, kód, verze, typ studia
\idtext{20.\,6.\,2002}
```

```
\obor{Přijímací písemná zkouška na obor anglický jazyk
pro základní školy}
\studium{bakalářské studium kombinované}
\sada 20 1 76 300        % počet testů, kód, verze, místnost
...
```

V příloze na straně 25 najdete příklady vysázených formulářů.

5 Skenování

Úspěšnost skenování je vysoce závislá na schopnosti ovládat a využít technické parametry dostupné techniky a doladit parametry použitého software [9,10]. Ceny specializovaných vysokokapacitních skenerů dosahují statisícových částek, a proto jsem se zaměřili na ověření možnosti využít skenerů cenových kategorií v řádu desetitisíců korun českých.

5.1 Skenovací zařízení

Pro vývoj našeho systému jsme použili skenovací zařízení firmy Hewlett Packard HP 6100 C s automatickým podavačem a skenovací zařízení HP 6350 C s automatickým podavačem ScanJet ADF. Zařízení byla připojena přes SCSI řadič.

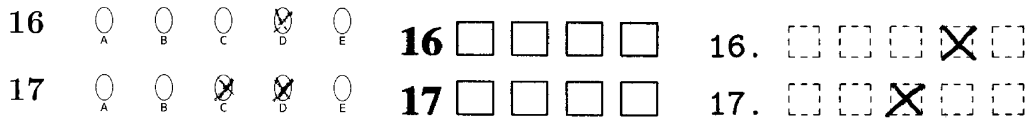
Podavač skeneru HP 6100 C má kapacitu maximálně 50 listů. Rychlost podávání a skenování jedné černobílé strany při rozlišení 300 dpi je 27–30 sekund. Předloha ani snímač se ve skeneru nepohybují. Pohybuje se čtecí paprsek, úzký paprsek světla usměrňovaný soustavou zrcadel. Množství světla, které se od snímaného předmětu odrazilo, se měří a digitalizuje.

Skener HP 6350 C je dodáván s automatickým podavačem dokumentů na 25 stran. Rychlost podávání a skenování jedné černobílé strany při rozlišení 300 dpi je 17–18 sekund. Komunikační rozhraní je USB nebo SCSI.

Skener HP 6350 C (obdobně jako jeho předchůdce) ke snímání používá pouze jednu řadu CCD senzorů. Zásadní rozdíl oproti skeneru HP 6100 C je v použitém podavači. Automatický podavač ScanJet ADF, jenž je dodáván ke skeneru HP 6100 C, nejprve papírový dokument zavede do skeneru a potom skenuje, zatímco u skeneru HP 6350 C je dokument skenován v průběhu vedení podavačem. Tato změna přinesla výhodu ve formě zrychlení podávání, nicméně ohýbání dokumentu se děje nad čtecím prostorem a proto je vhodné pravidelně po několika stovkách naskenovaných listů toto místo vyčistit. Pro testové formuláře je vhodné právě z tohoto důvodu používat kvalitní papír, který se otírá minimálně a je bez kazů.

5.2 Varianty zaškrtávacích prvků

Na obrázku 2 jsou zobrazeny příklady zaškrtávacích prvků – mohou být použity různé tvary. Oválky byly voleny pro jednodušší vyplňování. Klasickým tvarem pro zaškrtávání je čtvereček. Pro snížení tmavosti strany, resp. pro zvýraznění vybraných odpovědí, byly navrženy čtverečky kreslené přerušovanou čarou.



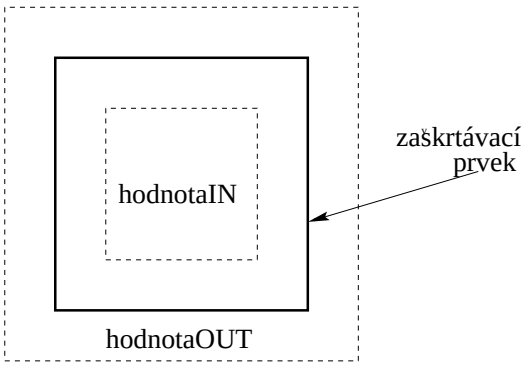
Obrázek 2. Varianty zaškrtávacích prvků

5.3 Čím zaškrtávat, jak vyhodnocovat

Ideálním prostředkem pro zaškrtávání jsou tenké fixy, propisovací tužky a pera v barvách černá, modrá, červená, zelená. Program je navržen tak, že rozpoznává zaškrtnutí i tenkou obyčejnou tužkou (pentelkou). Lze tedy vyplňovat téměř jakýmkoliv běžným prostředkem pro psaní po papíru s výjimkou nestandardních psacích pomůcek jako jsou žlutý či oranžový zvýrazňovač.

Významnou otázkou bylo, co vlastně považovat za vyplněný (zaškrtnutý) čtvereček. Při návrhu a měření schopností jsme se dostali na hranici, kdy je rozpoznána i drobná tečka, malá čárka a podobně. Nicméně již ve fázi testování se ukázalo, že některé odpovědi jsou takto drobně označeny jenom proto, že nad nimi student přemýšlel, ale nelze je považovat za zaškrtnuté.

Hodnotová funkce f je počítána jak z oblasti *hodnotaIN*, tak z oblasti *hodnotaOUT* tak, jak je zobrazeno na obrázku 3. Majoritní úlohu ve výpočtu hodnotové funkce má oblast *hodnotaIN*, takže i stejná značka (čárka, tečka, malý křížek) má v oblasti bližší ke středu větší hodnotu, než kdyby byl umístěn na okraji. Obor hodnot funkce f je $\langle 0, \ldots, 5600 \rangle$, kde 0 je nevyplněný a 5600 je zcela vyplněný čtvereček. Běžně, křížkem vyplněný čtvereček, má hodnotu cca 1600–1800. Drobná tečka cca 400. Současná *prahová hranice* pro rozhodnutí je nastavena na 960.



Obrázek 3. Výpočet hodnotové funkce

5.4 Posun a pootočení při tisku a skenování

Posun a pootočení naskenovaného obrazu vůči očekávaným pozicím je jednou z nejmarkantnějších nepřesností, které musíme řešit. Z převodu DVI očekáváme přesné pozice umístění vytisknutých prvků na papíru, ty se však v závislosti na tiskárně, kopírovacím stroji i skeneru liší. Posun i pootočení je obvykle způsobeno podavačem, většinou se jedná o několik milimetrů až centimetrů. Právě řešení tohoto problému je zcela nezbytné k dosažení potřebné přesnosti.

5.5 Detekce a korekce změn velikostí

Při tisku a zvláště při kopírování testových formulářů může docházet na některých zařízeních k zvětšení nebo zmenšení tiskového obrazu vůči standardním rozměrům daným v souboru DVI, resp. vůči originálu při kopírování. Změna je tak ve shodném poměru v ose x i y.

$$\frac{x'}{x} = \frac{y'}{y}$$

Může taktéž docházet ke změně velikosti pouze v ose y, tedy k jakémusi protažení obrazu. Tento defekt se projevuje právě u použitého skeneru HP 6350 C. Skener má podavač, jenž při průchodu papíru zároveň daný dokument skenuje. Při nepřesném podávání tak dochází k nechtěnému protažení, jak je ukázáno na obrázku 4.

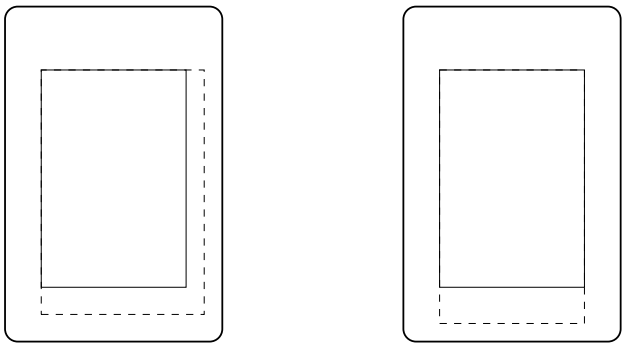
$$\frac{x'}{x} = 1$$
$$\frac{y'}{y} = k, \text{ kde } k \text{ je např. } 1.007$$

Při větším množství zpracovávaných testů (více než několik tisíc) může docházet i k následnému opotřebení zařízení podavače. Vodící kolečka pro podávání papíru vlivem opotřebení mírně zmenšují svůj průměr a načítání dokumentu tak trvá delší čas, což se projeví protažením naskenovaného obrazu ve směru osy y, a tím se tedy mění i k. Hodnota k se průběžně počítá ze zaměřovacích prvků.

5.6 Zaměřovací prvky

Zaměřovacími prvky mohou být buď zaměřovací křížky nebo čtverečky. Slouží k jednoznačnému a přesnému zaměření nasnímaného obrazu tak, aby mohly být přepočteny všechny souřadnice.

Nalezení prvků musí být bezchybné, jednoznačné a rychlé. Byly vyzkoušeny různé varianty, nicméně zaměřovací čtvereček je nejvíce odolný vůči různým nežádoucím poruchám snímání i nepřesnému zaostření. Na obrázku 5 jsou zobrazeny zvětšeniny zaměřovacích prvků tak, jak je nasnímá skener. Okraje prvků jsou nerovné, neostré. Vzhledem k tomu, že nalezení a kontrola nalezení



Obrázek 4. Zvětšení v osách x, y a pouze y

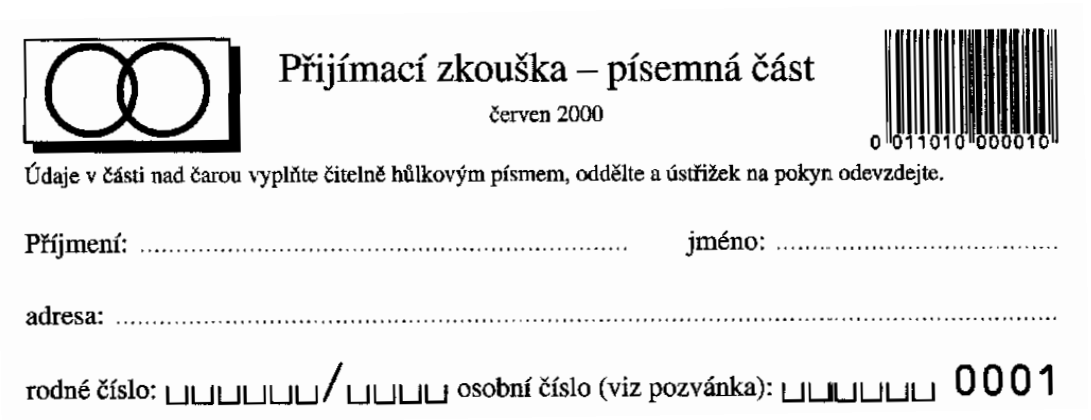
Obrázek 5. Varianty zaměřovacích prvků

černého čtverečku je i algoritmicky jednoduché, je v současnosti používán právě tento zaměřovací prvek.

Ze zaměřovacího prvku zjistíme *bod zaměření*, a to pro každý zaměřovací prvek. V našem případě obvykle používáme dva v protilehlých rozích testu, jak je vidět v příkladech v příloze na straně 25 a dalších. Ze zjištěných údajů a očekávaných údajů (vzniklých při převodu `DVI` na souřadnicový systém) vypočteme hodnoty zpětného posunutí, pootočení a zvětšení (respektive zmenšení). Tím z nasnímaného obrazu získáme očekávaný obraz.

5.7 Identifikace pomocí osobního čísla a čárového kódu

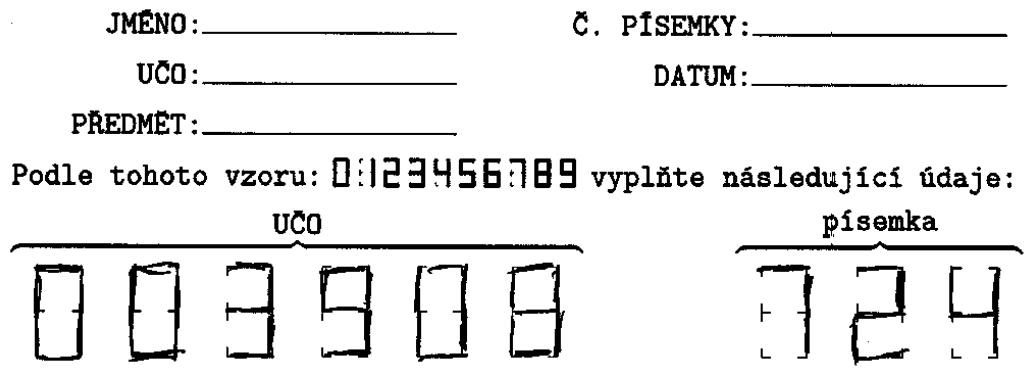
Na obrázku 6 je zobrazena identifikační část formuláře, který byl použit pro přijímací zkoušky v roce 2000. Vyplňující vypíší své příjmení, jméno, adresu, rodné číslo a především své osobní číslo. Na identifikační části i na části odpovědní je vytisknut shodný čárový kód. Zpracování tohoto formuláře spočívá v ručním opisování osobního čísla studentů (6 znaků) a kódového čísla formuláře (4 znaky). Tato identifikační část je oddělená od odpovědní části perforací. Formulář se po vyplnění rozdělí na dvě samostatné části, které jsou propojeny pouze logickou identifikací – předem na obě části vygenerovaným shodným čárovým kódem. Zpracování jednotlivých částí pak probíhá odděleně, a tím tedy i anonymně.



Obrázek 6. Identifikace s využitím čárového kódu

5.8 Identifikace ve formě digitálního čísla

Příklad na obrázku 7 ukazuje zapsání UČO: 003908 a číslo písemky: 724 do formy digitálních čísel.



Obrázek 7. Identifikace zapsaná ve formě digitálního čísla

Ve výstupu zpracování digitálního čísla se může objevit i znak „–“, který znamená nevyplněné digitální číslo. Chyba je označena znakem „x“ a hlášením na standardní chybový výstup.

Příklad výstupu zpracování:

```
-50821,176, 1.A: 2.A: 3.C: 4.D: 5.D: 6.-: 7.C: 8.-: 9.B:
10.ACD: 11.-: 12.D: 13.B: 14.AD: 15.D: 16.-: 17.B: 18.D:
19.D: 20.-: 21.D: 22.ABCD: 23.-: 24.D: 25.-:
```

V reálném testování se jako nejpřesnější ukázal algoritmus pracující na základě hledání souvislostí mezi jednotlivými body (rohy) digitálního čísla.

Je odolný vůči různým intenzitám vyplnění – od obyčejné tužky až po pero, propisku či tenkou fixu.

6 Vyhodnocení výsledků a kontroly

Pro nasazení našeho systému v reálném prostředí zpracování testů v přijímacím řízení bylo rozhodnuto, že budeme skenovat pouze část pro odpovědi. Část pro identifikaci bude zadávána ručně. Systém ukládá do souboru výsledky zpracování naskenovaných odpovědí, je schopen načíst vzorové odpovědi a uložit je do souboru. Na obrázku 8 je identifikační část označena zkráceně „ID.“, odpovědní část „ODP.“ a čárový kód jako „EAN“. Náš systém na základě zadání souboru výsledků a zadání souboru správných vzorových odpovědí vyhodnotí bodovou (procentuální) úspěšnost a je schopen uložit tyto body spolu s identifikační částí čárového kódu do souboru. Za správně zodpovězenou otázku se považuje shodnost všech odpovědních prvků (obvykle tří, čtyř, resp. pěti čtverečků) se vzorovým řešením a bude započtena 1 bodem, v opačném případě 0 body. Lze nastavit i jiné bodové ohodnocení odpovědí (třeba záporné body za chybné odpovědi).

Systém upozorňuje obsluhu na případné automaticky zjistitelné chyby, např. nesprávně naskenovaný formulář, nečitelný – poškozený čárový kód EAN, poškozené zaměřovací čtverečky a podobně. Systém umožňuje obsluze kontrolovat správnost rozpoznání odpovědí.

6.1 Testování v reálném prostředí

V letech 2000, 2001 i 2002 byl systém SCAT nasazen v reálném prostředí přijímacích zkoušek na vysokou školu. V roce 2001 byl i pro testovací účely využit jako prostředek vyhodnocení písemného zkoušení na vysoké škole. Každým rokem bylo vyhodnoceno cca 30 000 testových formulářů.

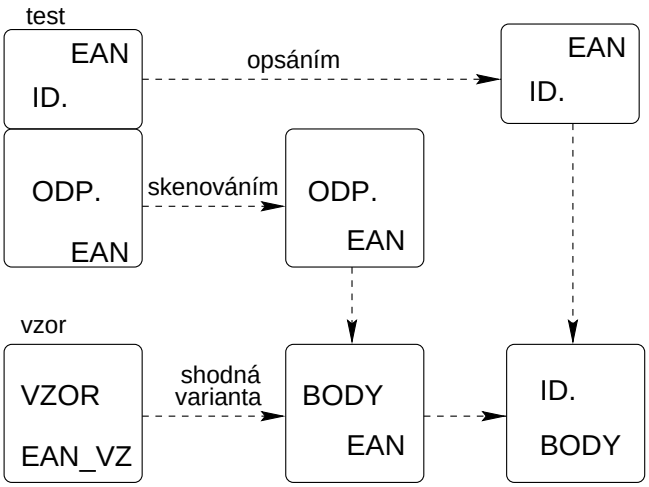
6.1.1 Slabá místa systému Časově nejnáročnější je proces samotného skenovaní, které je vždy omezeno rychlostí podavače skeneru. Vzhledem k cenové dostupnosti tohoto zařízení lze však použít skenerů několik a tím celý proces zpracování výrazně urychlit.

Z důvodu anonymity se nejčastěji používá pro účely přijímacích zkoušek systém, kde je fyzicky oddělena identifikační a odpovědní část. Ukázalo se, že ruční zpracování identifikační části testu obvykle nečiní žádné potíže. Náročné na logistiku může být na některých fakultách vyhodnocení relací různých oborových testů, toto však je zcela nezávislé na technologii zpracování odpovědí.

7 Závěr

7.1 Probíhající vývoj

Vývoj systému SCAT se nezastavil. Dále probíhá řešení rozpoznávání ručně psaných čísel. Testují se nové algoritmy, zkoušejí se nové přístupy, aby se eliminovala



Obrázek 8. Logistika přijímacího řízení

chybovost při analýze čísel. Jedna z možností je kontrolovat ihned při skenování (resp. zpracování) relace typu *rodné číslo – číslo přihlášky* a připravit uživatelské rozhraní pro odpovídající zásahy obsluhy.

Jinou oblastí vývoje, ve které se také pokračuje, je hledání nových skenovacích zařízení. Nové skenery umožňují zajímavé zrychlení. Skener HP 7450 C skenuje cca 10 stran za minutu, skener Brother MFC 9180 cca 8 stran za minutu. Vývoj možná povede i do oblasti digitální fotografie – tedy formulář by se již neskenoval po řádcích, ale fotografoval jako jeden celek. To by mohlo přinést opět výrazné urychlení celkového zpracování. Dále chceme systém rozšířit o možnost rozpoznávání čísel [5] a optimalizovat nastavení prahovacích konstant pomocí algoritmů z [8].

7.2 Shrnutí výsledků

Naším cílem bylo prokázat, že lze s využitím současných, běžně dostupných zařízení automatizovat vytváření, zpracování a vyhodnocení písemných testů. Díky automatickému způsobu převodu souřadnic přímo z elektronického obrazu formuláře, zaměření a srovnání naskenovaného obrazu jsme dosáhli požadovanou přesnost vyhodnocování. Vypracovali jsme způsob rozpoznání digitálních čísel. Navrhli jsme a vytvořili prostředí a logistiku vyhodnocování tak, že vznikl robustní a flexibilní systém. Systém byl úspěsně použit na zpracování téměř sta tisíce formulářů.

Reference

1. Adriana Benadiková, Štefan Mada, a Stanislav Weinlich. *Čárové kódy, automatická identifikace*. Grada Publishing, 1994.
2. Miroslav Hrad. Automatizace vytváření, zpracování a vyhodnocení písemných testů. Diplomová práce, Masarykova univerzita v Brně, Fakulta informatiky, duben 2001.

3. Donald Erwin Knuth. *The TEXbook*. Addison-Wesley, Reading, Massachusets, 1984.
4. Miroslav Maťák. Automatizované rozpoznávání testů. Bakalářská práce, Masarykova univerzita v Brně, Fakulta informatiky, 1999.
5. Roman Mikolaj. Gravitačný algoritmus rozpoznávania znakov. Diplomová práce, MFF UK, Bratislava, katedra informatiky, březen 1998.
6. Petr Olšák. The EAN barcodes by TEX. *TUGboat*, 15(4):459–464, 1994.
7. Jordi Saludes. Fast and secure multiple-option tests. *TUGboat*, 17(3):310–319, 1996.
8. Michail I. Schlesinger a Václav Hlaváč. *Deset přednášek z teorie statistického a strukturního rozpoznávání*. Vydavatelství ČVUT, Praha, 1999.
9. Petr Sojka. An Experience from a Digitization Project. *Cahiers GUTenberg*, (28–29):276–281, březen 1998.
10. Petr Sojka. Publishing Encyclopaedia with Acrobat using TEX. V *Towards the Information-Rich Society. Proceedings of the ICCC/IFIP conference Electronic publishing '98*, strany 217–222, Budapest, Hungary, duben 1998. ICCC Press.
11. Petr Sojka, Rudolf Červenka, a Martin Svoboda. TEX for database publishing. V Jiří Zlatuška, editor, *Proceedings of the 7th European TEX Conference, Prague, 1992*, strany 53–58, Brno, září 1992. Masarykova univerzita v Brně.

Institut mezioborových studií
Brno

PŘIJÍMACÍ ZKOUŠKA

studijní program **Pedagogika**, obor **Sociální pedagogika**

ak. rok 2002/2003

Údaje v části „A“ (nad čarou) vyplňte čitelně hůlkovým písmem. Ústřižek na pokyn odtrhněte a odevzdejte.

Část „A“

Příjmení: …… Jméno: ……

Rodné číslo: ⌊⌊⌊⌊⌊⌊/⌊⌊⌊⌊ Registrační číslo: ⌊⌊⌊⌊⌊ 1051

Část „B“

	A	B	C		A	B	C		A	B	C		A	B	C		A	B	C
1	☐	☐	☐	21	☐	☐	☐	41	☐	☐	☐	61	☐	☐	☐	81	☐	☐	☐
2	☐	☐	☐	22	☐	☐	☐	42	☐	☐	☐	62	☐	☐	☐	82	☐	☐	☐
3	☐	☐	☐	23	☐	☐	☐	43	☐	☐	☐	63	☐	☐	☐	83	☐	☐	☐
4	☐	☐	☐	24	☐	☐	☐	44	☐	☐	☐	64	☐	☐	☐	84	☐	☐	☐
5	☐	☐	☐	25	☐	☐	☐	45	☐	☐	☐	65	☐	☐	☐	85	☐	☐	☐
6	☐	☐	☐	26	☐	☐	☐	46	☐	☐	☐	66	☐	☐	☐	86	☐	☐	☐
7	☐	☐	☐	27	☐	☐	☐	47	☐	☐	☐	67	☐	☐	☐	87	☐	☐	☐
8	☐	☐	☐	28	☐	☐	☐	48	☐	☐	☐	68	☐	☐	☐	88	☐	☐	☐
9	☐	☐	☐	29	☐	☐	☐	49	☐	☐	☐	69	☐	☐	☐	89	☐	☐	☐
10	☐	☐	☐	30	☐	☐	☐	50	☐	☐	☐	70	☐	☐	☐	90	☐	☐	☐
11	☐	☐	☐	31	☐	☐	☐	51	☐	☐	☐	71	☐	☐	☐	91	☐	☐	☐
12	☐	☐	☐	32	☐	☐	☐	52	☐	☐	☐	72	☐	☐	☐	92	☐	☐	☐
13	☐	☐	☐	33	☐	☐	☐	53	☐	☐	☐	73	☐	☐	☐	93	☐	☐	☐
14	☐	☐	☐	34	☐	☐	☐	54	☐	☐	☐	74	☐	☐	☐	94	☐	☐	☐
15	☐	☐	☐	35	☐	☐	☐	55	☐	☐	☐	75	☐	☐	☐	95	☐	☐	☐
16	☐	☐	☐	36	☐	☐	☐	56	☐	☐	☐	76	☐	☐	☐	96	☐	☐	☐
17	☐	☐	☐	37	☐	☐	☐	57	☐	☐	☐	77	☐	☐	☐	97	☐	☐	☐
18	☐	☐	☐	38	☐	☐	☐	58	☐	☐	☐	78	☐	☐	☐	98	☐	☐	☐
19	☐	☐	☐	39	☐	☐	☐	59	☐	☐	☐	79	☐	☐	☐	99	☐	☐	☐
20	☐	☐	☐	40	☐	☐	☐	60	☐	☐	☐	80	☐	☐	☐	100	☐	☐	☐

Jednu správnou odpověď zaškrtněte takto: ☒

Výsledkový formulář vyplňte přesně podle pokynů zkoušejícího!

Pedagogická fakulta MU v Brně

Přijímací písemná zkouška na obor technická výchova

prezenční studium červen 2002

Údaje v části nad čarou vyplňte čitelně hůlkovým pí smem, oddělte a ústřižek na pokyn odevzdejte.

Pří jmení : …… jméno: ……

rodné čí slo: ⌊⌊⌊⌊⌊⌊/⌊⌊⌊⌊ čí slo přihlášky: ⌊⌊⌊⌊⌊⌊ 1301

	A	B	C	D		A	B	C	D		A	B	C	D		A	B	C	D
1	☐	☐	☐	☐	**11**	☐	☐	☐	☐	**21**	☐	☐	☐	☐	**31**	☐	☐	☐	☐
2	☐	☐	☐	☐	**12**	☐	☐	☐	☐	**22**	☐	☐	☐	☐	**32**	☐	☐	☐	☐
3	☐	☐	☐	☐	**13**	☐	☐	☐	☐	**23**	☐	☐	☐	☐	**33**	☐	☐	☐	☐
4	☐	☐	☐	☐	**14**	☐	☐	☐	☐	**24**	☐	☐	☐	☐	**34**	☐	☐	☐	☐
5	☐	☐	☐	☐	**15**	☐	☐	☐	☐	**25**	☐	☐	☐	☐	**35**	☐	☐	☐	☐
6	☐	☐	☐	☐	**16**	☐	☐	☐	☐	**26**	☐	☐	☐	☐	**36**	☐	☐	☐	☐
7	☐	☐	☐	☐	**17**	☐	☐	☐	☐	**27**	☐	☐	☐	☐	**37**	☐	☐	☐	☐
8	☐	☐	☐	☐	**18**	☐	☐	☐	☐	**28**	☐	☐	☐	☐	**38**	☐	☐	☐	☐
9	☐	☐	☐	☐	**19**	☐	☐	☐	☐	**29**	☐	☐	☐	☐	**39**	☐	☐	☐	☐
10	☐	☐	☐	☐	**20**	☐	☐	☐	☐	**30**	☐	☐	☐	☐	**40**	☐	☐	☐	☐

Právě jednu správnou odpověď zaškrtněte takto: ☒

Výsledkový formulář vyplňte přesně podle pokynů zkoušejí cí ho!

Využití XML a LaTeXu při sazbě odborných knih

Zdeněk Wagner

Ice Bear Soft
Email: wagner@cesnet.cz
Web: http://icebearsoft.euweb.cz

Abstrakt: Odborné knihy mají obvykle pevnější strukturu než beletrie. Tato struktura nebývá samoúčelná, ale často je důležitou nositelkou informace. Ač LaTeX jistým způsobem vede zkušené uživatele k užívání strukturního značkování, není zachování informace vždy triviální a vyžaduje programátorský způsob ke zpracování dokumentu. Jisté potíže vzniknou i při tvorbě neobvyklých rejstříků, kde se naráží na omezení programu MakeIndex. XML naproti tomu definuje strukturu dokumentu. Navíc přináší hotové nástroje pro různé typy zpracování. Pro znalce LaTeXu je pak přirozenou cestou využití XSLT a transformovaný text zpracovat LaTeXem jako sázecím strojem. Přednáška je demonstrována případovou studií počínaje návrhem DTD přes transformační styl až po LaTeXová makra užitá k sazbě. Je provedeno srovnání se zpracováním obdobné knihy, kdy nebylo využito výhod XML.

Klíčová slova: XML, LaTeX, XSLT, DTD, validace, schéma

1 Úvod

Při zpracování dokumentů zjišťujeme, že jejich text je určitým způsobem členěn. Toto členění obvykle plní určitou funkci. Při tisku takového dokumentu pak musíme členění reprezentovat graficky.

Finální grafickou úpravu nemusíme vždy znát na počátku práce s dokumentem. S výhodou tedy využijeme nějaký sázecí systém, který dokáže separovat obsah a formu. Možností máme několik.

LaTeX (a vlastně i plain TeX) umožňuje definovat kontextové značky, jimž až dodatečně přidělíme vizuální podobu. Pokud se během zpracování změní požadavky na grafickou úpravu, nemusíme zasahovat do textu, ale upravujeme pouze balík podpůrných maker.

Na zachycení obsahu dokumentu nezávisle od formy jsou však primárně určeny SGML a XML. Výhodou je, že k těmto formátům existuje řada volně šiřitelných i komerčních nástrojů. Navíc lze dokument snadno přeformátovat do několika různých podob. V tomto příspěvku ukážeme, že použití XML je výhodné i v případě, kdy ze vstupního souboru generujeme pouze jediný výstup.

Jan Kasprzak, Petr Sojka (editoři): SLT 2002 – sborník semináře o Linuxu a TeXu, str. 27–48, 2002.

2 Sazba XML souboru

XML definuje strukturu dokumentu, ale neurčuje vizuální vzhled. Ten lze také určit pomocí nástrojů XML, tzv. formátovacích objektů. Přestože již existují nezávislé nástroje, které provádějí sazbu přímo na základě formátovacích objektů, nejsou v typografické kvalitě schopny konkurovat TEXu.

V TEXovém světě již existují nástroje pro přímou sazbu XML, např. PassiveTEX [7] a ConTEXt. Jejich využitelnost však není univerzální. Může se stát, že z daného XML dokumentu chceme tisknout jen určité části, případně se mají tisknout v jiném pořadí, než v jakém se v původním souboru vyskytují. Tyto operace zvládne snadno XSLT. Zkušený uživatel TEXu či LATEXu si nyní zákonitě položí otázku, proč by po transformaci mělo následovat zpracování s využitím některého z výše uvedených nástrojů, když výstupem transformace může být stejně tak TEXový soubor a o jeho formátování se mohou postarat TEXová makra.

Při transformaci z XML do TEXu se můžeme rozhodnout, jakou formátovací práci provede transformační styl a co svěříme až TEXovým makrům. Prvním extrémem je, že při transformaci pomocí XSLT převedeme vše až na TEXové příkazy nízké úrovně. Opačným extrémem je, že veškerou práci ponecháme LATEXu. XSLT zpracuje pouze základní elementy a vše ostatní se bude formátovat s využitím LATEXového balíku KEYVAL, přičemž atributy jsou konvertovány na nepovinné parametry prostředí. Transformační styl pak může vypadat např. takto:

```
<?xml version="1.0" encoding='cp852'?>

<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
                version="1.0"
                xmlns:saxon="http://icl.com/saxon"
                extension-element-prefixes="saxon">
<xsl:output indent="no" method="text" encoding="cp852"
            saxon:character-representation="native"/>

<xsl:strip-space elements="*"/>

<!-- Root -->
<xsl:template match="/">
  \documentclass{book}
  \usepackage{keyval}
  \usepackage{mujstyl}
  \begin{document}
    <xsl:apply-templates/>
  \end{document}
</xsl:template>

<!-- Texty -->
<xsl:template match="text()">
```

```
  <xsl:value-of select="."/>
</xsl:template>

<!-- Odstavec -->
<xsl:template match='p'>
  <xsl:apply-templates/>
  <xsl:text>&#10;&#10;</xsl:text>
</xsl:template>

<!-- Tučně -->
<xsl:template match='b'>
  <xsl:text>\textbf{</xsl:text>
  <xsl:apply-templates/>
  <xsl:text>}</xsl:text>
</xsl:template>

<!-- Kurzíva -->
<xsl:template match='i'>
  <xsl:text>\textit{</xsl:text>
  <xsl:apply-templates/>
  <xsl:text>}</xsl:text>
</xsl:template>

<!-- Ostatní -->
<xsl:template match='*'>
  <xsl:value-of select='concat("\begin{", name(), "}")'/>
  <xsl:if test='@*'>
    <xsl:text>[</xsl:text>
    <xsl:apply-templates select='@*'/>
    <xsl:text>]</xsl:text>
  </xsl:if>
  <xsl:apply-templates/>
  <xsl:value-of select='concat("\end{", name(), "}")'/>
</xsl:template>

<!-- Atributy -->
<xsl:template match='@*'>
  <xsl:if test='position()>1'>
    <xsl:text>,</xsl:text>
  </xsl:if>
  <xsl:value-of select='concat(name(), "=", .)'/>
</xsl:template>

</xsl:stylesheet>
```

Skutečně použitý styl pak bude někde mezi těmito dvěma extrémy. Co provede XSLT a co se ponechá LaTeXovým makrům bude záležet především na schopnostech a zkušenostech každého programátora. Můžeme se ovšem také rozhodnout pro některý z běžně užívaných formátů XML a využít transformačních stylů, které jsou již hotové.

3 XML, pro a proti

Viděli jsme, že oproti přímému vytvoření dokumentu v LaTeXu máme při použití XML jednu operaci navíc, a to transformaci. Měli bychom se tedy zamyslet, zda nám tato práce navíc přinese nějaký užitek.

Obvyklým argumentem pro použití XML je snadnost převedení dokumentu do několika různých tvarů (PDF, HTML, tištěná podoba...). XML má však jisté výhody i v případě, že požadujeme jediný výstup.

Jedna z výhod se projeví při sazbě odborných knih, které mají pevně danou strukturu. Struktura je zde též nositelkou informace a může sloužit pro následné zpracování. Předpokládáme totiž, že životnost knihy nekončí jejím vydáním, ale její obsah může v budoucnu sloužit k účelům, které v okamžiku sazby ještě neznáme. Můžeme namítnout, že i z LaTeXového souboru lze získat dodatečné informace, pokud byl vhodně kontextově značkován, ale vždy je to mnohem náročnější než využití XSLT.

V učebnici matematiky lze věty a důkazy značkovat způsobem:

```
<věta číslo='1'>
  <text>
    Sem přijde text věty.
  </text>
  <důkaz>
    Zde bude důkaz
  </důkaz>
</věta>
```

V budoucnu lze pak z knihy vytáhnout např. jen věty a jejich důkazy. Podobně můžeme využít strukturu:

```
<cvičení číslo='1'>
  <příklad>
    Zadání příkladu.
  </příklad>
  <řešení>
    Řešení, které bude uvedeno jinde.
  </řešení>
</cvičení>
```

Tak máme příklad i jeho řešení v dokumentu na stejném místě, což sníží pravděpodobnost zavlečení chyb. Pomocí XSLT snadno v hlavní části knihy vytiskneme jen zadání příkladu a v závěrečné kapitole jeho řešení.

Jinou aplikací je vytažení hypertextových odkazů z knihy, aby mohly být zveřejněny na WWW stránkách. Příklad najdete na WWW stránkách věnovaných knize *XML pro každého* [4].

Další výhoda se projeví, pokud v dokumentu potřebujeme velmi nestandardní rejstříky. Program MakeIndex má svá omezení. Někdy bychom potřebovali buď netriviální zásah do jeho zdrojového kódu, nebo pomocný preprocesor. Ukázka bude uvedena později v případové studii. Vzhledem k tomu, že XSLT umí abecední řazení, může být využití XSLT při tvorbě rejstříků snadnější.

XML na rozdíl od LaTeXu umožňuje definovat strukturu dokumentu a ověřit jeho správnost. O metodách validace se podrobněji zmíníme později.

Při psaní českých textů v XML můžeme narazit na problém s nezlomitelnou mezerou za jednopísmennými neslabičnými předložkami. Řešení však není složité. Lze použít program VLNA či jeho ekvivalent a následně nahradit tildu entitou ` `.

4 Volba typu dokumentu

Na začátku návrhu dokumentu v XML musíme zvolit způsob jeho značkování. Velmi rozšířeným standardem, používaným v mnoha dokumentech, je DocBook [1]. Pro jeho použití v konkrétním případě se však nerozhodujeme podle toho, zda je dobrý či špatný, zda je používán zřídka či často, ale podle toho, zda podporuje struktury, které náš dokument obsahuje, případně, zda se dá snadno modifikovat, aby vyhověl našim značkovacím požadavkům. DocBook obsahuje základní elementy, které se používají ve všech dokumentech, a dále řadu dalších elementů potřebných pro různé technické dokumenty. DTD je navrženo tak, aby jej bylo možno poměrně snadno doplňovat a modifikovat. Přesto se v praxi setkáme s případy, kdy DocBook nabízí příliš mnoho elementů, které vůbec nepotřebujeme, a zcela chybí elementy, které jsou pro danou aplikaci naprosto nepostradatelné. Pak je vytvoření nového DTD nejpřirozenějším řešením.

5 DTD ano či ne?

Výhodou XML oproti SGML je, že pro zpracování dokumentu často nepotřebujeme DTD. Obvykle si vystačíme s definicemi různých entit, které můžeme vložit přímo do dokumentu. Ač je deklarace entit součástí DTD, není to takové DTD, jaké známe z SGML. Navrhujeme-li tedy strukturu dokumentu pro jedno použití, nemusíme se s vytvářením DTD zdržovat. Praxe však ukazuje, že DTD je užitečné i v těchto případech.

V první řadě si musíme uvědomit, že člověk při psaní dokumentu chybuje. Při zapisování tagu lze udělat překlep. Nemáme-li textový editor, který doplňuje ukončovací tagy, můžeme je zapomenout nebo v nich udělat překlep. Editor, který doplňuje uzavírací tagy párově k počátečním, je dobrý sluha, ale může být i zlým pánem. Předpokládejme, že místo `<emphasis>` namíšeme omylem `<emhpasis>`. „Chytrý“ editor nám pak na konec zvýrazněné fráze doplní `</emhpasis>`. Parser, který neprovádí validaci, žádnou chybu nezaznamená.

Chyba se projeví až při zpracování dokumentu. XSLT procesor má obvykle defaultní pravidlo pro případ, že žádná explicitní šablona nebyla nalezena. Takto chybně zadaný element se tedy zpracuje „nějak“ a zvláště v případě rozsáhlých dokumentů a komplikovaných stylů nemusí být snadné odhalit příčinu, proč je část textu zpracována zcela jinak, než bylo zamýšleno. Diagnostické zprávy validujících parserů jsou sice někdy poněkud kryptické, ale mnohem rychleji vedou k určení zdroje chyby.

Druhou výhodu validace oceníme v okamžiku, kdy začneme programovat styl, jenž s dokumentem provádí složitější transformaci. DTD nám totiž naprosto jasně říká, jaké situace mohou nastat. Víme tudíž naprosto přesně, jaké případy musíme v transformačním stylu ošetřit, a čím se zabývat nemusíme, protože to validující parser nepřipustí. Autor ze své zkušenosti ví, že návrh transformačního stylu je snadnější, když si vezmeme DTD a postupně podle něj vytváříme jednotlivé šablony.

6 Alternativní metody validace

Je obecně známo, že DTD určuje pouze základní pravidla pro strukturu dokumentu. V praxi často potřebujeme podstatně přesnější vymezení typu a obsahu elementů i jejich atributů. Tento problém se snaží řešit *schéma*. Bohužel nástroje, které využívají schéma pro validaci, nejsou ještě dostatečně rozšířené. Navíc ani tak nejsou ošetřeny všechny možnosti, s nimiž se můžeme setkat. Nezbývá tedy jiná možnost než přistoupit k validaci vlastními prostředky. V principu je možné využít některou z knihoven pro práci s XML soubory a napsat si speciální validátor. Často je však vhodnější vytvořit validátor pomocí XSLT.

Vezměme si jako příklad XML soubor, který obsahuje výsledky matematického zpracování jistých fyzikálně chemických měření. Soubor odpovídá následujícímu DTD:

```
<!ELEMENT rset (result+)>

<!ELEMENT result (system,concentration,T,parameters,table,sigma)>

<!ELEMENT table (row+)>

<!ELEMENT row (cell+)>

<!ELEMENT cell (#PCDATA)>

<!ELEMENT system (#PCDATA)>

<!ELEMENT concentration (#PCDATA)>

<!ELEMENT T (#PCDATA)>
<!ATTLIST T unit  CDATA 'K'>
```

```
<!ELEMENT sigma (#PCDATA)>

<!ELEMENT parameters EMPTY>
<!ATTLIST parameters C  CDATA #REQUIRED
                     D  CDATA #REQUIRED>
```

Z DTD je patrné, že každý `<result>` obsahuje právě jednu tabulku. V DTD však nelze zapsat jednu podstatnou podmínku, kterou nepodporuje ani schéma. Tabulka totiž obsahuje obvykle jeden řádek, který má druhý sloupec prázdný. Takový řádek nás při dalším zpracování nebude zajímat. Podmínkou však je, aby všechny tabulky, po vynechání řádku s prázdným druhým sloupcem, měly identický počet řádků. Druhou podmínkou je, že hodnoty v prvním sloupci všech tabulek musí být shodné.

Čtenář by mohl namítnout, že jsme říkali, že soubor byl vytvořen počítačem. Validace by tudíž neměla být nutná, protože generované soubory jsou správné. Ve skutečnosti je však nepřeberná řada příčin, které vedou k tomu, že programově vytvořený soubor je vadný. Dokud není program dokonale otestován, není jisté, že v něm není nějaká chyba. Validace výsledného XML souboru je jedním z užitečných kroků při testování programu. A nesmíme ani zapomenout na to, že původní vstupní data vytvářel člověk. V hodnotách prvního sloupce tedy mohou být překlepy[1] a také mohlo dojít k tomu, že se jeden řádek v některé tabulce ztratil. Podmínky tohoto typu neřeší ani schéma, ale k validaci lze použít následující styl:

```
<?xml version="1.0" encoding='cp852'?>

<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
                version="1.0"
                xmlns:saxon="http://icl.com/saxon"
                extension-element-prefixes="saxon">
<xsl:output indent="no" method="text" encoding="cp852"
            saxon:character-representation="native"/>

<xsl:strip-space elements="*"/>

<!-- All elements -->
<xsl:template match="/|*">
  <xsl:apply-templates/>
</xsl:template>

<!-- Text contents of all elements -->
```

[1] Zdálo by se, že by bylo jednodušší, kdyby data pro první sloupec byla zadána pouze jednou. Z povahy měření a následného zpracování, jejichž vysvětlení je nad rámec tohoto příspěvku, však plyne, že by to způsobilo laborantce mnoho těžkostí a spíše by to vedlo k zavlečení dalších chyb.

```
<xsl:template match="text()"/>

<!-- Result -->
<xsl:template match='result[position()>1]'>
  <xsl:apply-templates>
    <xsl:with-param name='pos'>
      <xsl:value-of select='position()'/>
    </xsl:with-param>
  </xsl:apply-templates>
</xsl:template>

<xsl:template match='result'/>

<!-- Table -->
<xsl:template match='table'>
  <xsl:param name='pos'/>
  <xsl:if test=
         'not(count(row[not(cell[position()=2]="")])=
          count(/rset/result[position()=$pos]/table/row
          [not(cell[position()=2]="")]))'>
    <xsl:text>Nesouhlasí počet řádků: </xsl:text>
    <xsl:value-of
         select='count(row[not(cell[position()=2]="")])'/>
    <xsl:text> != </xsl:text>
    <xsl:value-of
         select='count(/rset/result[position()=$pos]/table/row
                 [not(cell[position()=2]="")])'/>
  <xsl:text>&#10;</xsl:text>
  </xsl:if>
  <xsl:apply-templates>
    <xsl:with-param name='pos'>
      <xsl:value-of select='$pos'/>
    </xsl:with-param>
  </xsl:apply-templates>
</xsl:template>

<!-- Row -->
<xsl:template match='row'>
  <xsl:param name='pos'/>
  <xsl:variable name='rpos'>
    <xsl:value-of select='position()'/>
  </xsl:variable>
  <xsl:if test=
      'not(cell[1]=/rset/result[$pos]/table/row[$rpos]/cell[1])'>
    <xsl:text>Nesouhlasí hodnota v 1. sloupci: </xsl:text>
```

```
    <xsl:value-of select='cell[1]'/>
    <xsl:text> != </xsl:text>
    <xsl:value-of
         select='/rset/result[$pos]/table/row[$rpos]/cell[1]'/>
    <xsl:text>&#10;</xsl:text>
  </xsl:if>
</xsl:template>

</xsl:stylesheet>
```

Ve stylu jsou zajímavé dvě věci. První z nich je potlačení výstupu textových uzlů. Druhou zvláštností jsou dvě šablony pro element `<result>`. Předpokládáme totiž, že tabulka v prvním výsledkovém bloku je správná, proto ji nekontrolujeme. Všechny ostatní musí souhlasit s první tabulkou.

Tím ovšem nejsou vyčerpány všechny možnosti, s nimiž se můžeme setkat. Některé chyby v XML souborech nelze odhalit žádným strojově vyjádřitelným algoritmem. K jejich nalezení nám ovšem zase pomůže XSLT. Kromě transformací se dá XSLT využít i jako dotazovací jazyk. Můžeme si tedy vyhledat a vypsat podezřelá místa a pak rozhodnout, zda je obsah správný či špatný. Konkrétní příklad bude uveden později.

7 Případová studie

Možnosti využití XML a LaTeXu jsou demonstrovány na knize *Vinohradský hřbitov včera & dnes* [5]. Její styl pro transformaci do LaTeXu má však více než 900 řádků a LaTeXová makra mají přes 250 řádků. Jejich plný kód zde proto nebude uveden a zaměříme se pouze na vyzdvižení hlavních výhod, které použití XML přineslo.

7.1 Pracovní cyklus

Kniha tohoto typu vzniká poměrně dlouho. Autor, který vzhledem ke svému věku počítač nepoužívá, zapisuje text obvykle na kartičky. Z této kartotéky písařka text knihy opíše. Navíc každý ze spoluautorů pracoval na jiné části (jež se však prolínají). Písařka musí tyto části vhodně spojit. Z toho pak plyne, že při prvních korekturách se provádí poměrně mnoho změn. Kniha je sázena dvousloupcově a do textu jsou vsazovány fotografie. Pravidla pro jejich umístění jsou dosti přísná a mechanismus plovoucích objektů nelze využít. Fotografie jsou proto vkládány až při poslední korektuře. Při jednotlivých fázích zpracování tedy dochází k posunu materiálu na jiné stránky. Odkazy uvnitř knihy musíme proto řešit nějakým automatem.

Odkazy na čísla stran se týkaly zejména rejstříků, jež byly dost nestandardní. Autoři je sice zpracovali ručně, ale opakované dopisování správných čísel stran by jistě vedlo k mnoha chybám. Jak se později ukáže, právě při tvorbě rejstříků se projevila jedna z výhod XML.

Text knihy přepisovala externí písařka a posílala po částech elektronickou poštou. Protože písařka nemá žádný nástroj pro práci s XML (text psala ve

Wordu), nemá TEX ani Acrobat Reader, byl nakonec vytvořen i soukromý styl pro transformaci do HTML. Pokud bylo nutno, aby písařka něco rychle ověřila v původní kartotéce, mohla se podívat na privátní WWW stránky se zpracovanou částí knihy. Klikací verze v HTML také významně pomohla při korekturách.

7.2 Tvoříme DTD

K potřebě vytvoření DTD jsem dospěl hned na počátku práce, kdy písařka (která se při přepisování tohoto rukopisu teprve začínala učit XML) udělala první chybu. Základní struktura byla zřejmá, ale postupem času se DTD mírně měnilo a rostlo.

Zmíněna kniha má dva úvody, hlavní část, dva rejstříky, seznam literatury a tiráž. Pouze hlavní část a rejstříky byly generovány z XML, zbývající části byly napsány přímo v LATEXu. DTD bylo proto děláno jen pro část dokumentu.

V úvodu bychom se měli zmínit, že parser sice komentáře ignoruje, ale musí je přečíst, aby našel jejich konec. I když pro tagy použijeme pouze písmena anglické abecedy, musíme definovat použité kódování kvůli česky psaným komentářům. DTD je také XML soubor a pokud v něm píšeme česky, musíme v něm použít XML deklaraci. Protože DTD je poměrně krátké, uvedeme je celé:

```
<?xml version='1.0' encoding='cp852'?>
<!-- DTD pro hřbitov -->

<!-- Znakové entity -->
<!ENTITY otilde "&#245;">
<!ENTITY egrave "&#232;">

<!-- Parametrické entity -->
<!-- ================== -->

<!-- Obsah hrobů -->
<!ENTITY % attr
   "jmeno   CDATA #REQUIRED
    krestni CDATA #IMPLIED
    narozen CDATA #IMPLIED
    smrt    CDATA #IMPLIED
    titul   CDATA #IMPLIED
    prefix  CDATA #IMPLIED
    hodnost CDATA #IMPLIED">

<!-- Ano/Ne -->
<!ENTITY % ano "(ano|ne) 'ano'">

<!-- Struktura dokumentu,
     hřbitov má nejméně jedno oddělení -->
<!ELEMENT hrbitov (oddeleni|tex)+>
```

```
<!-- Oddělení jsou číslována,
     obvykle začínají plánkem a mají hroby -->
<!ELEMENT oddeleni (hrob|img|tex)+>
<!ATTLIST oddeleni cislo  CDATA #REQUIRED
                   planek CDATA #IMPLIED
                   extra  CDATA #IMPLIED
                   popis  (vedle|pres|sloupec)  'vedle'
                   space  CDATA #IMPLIED>

<!-- Obrázek, většinou fotografie -->
<!ELEMENT img (#PCDATA)>
<!ATTLIST img  height   CDATA #IMPLIED
               width    CDATA #IMPLIED
               scale    CDATA #IMPLIED>

<!-- Oddělení má hroby, většinou číslované,
     výjimečně obsahuje obrázky -->
<!ELEMENT hrob (#PCDATA|clovek|rodina|umelec|p|tex|em|img)*>
<!ATTLIST hrob cislo CDATA #IMPLIED>

<!-- Rodina má jenom lidi -->
<!ELEMENT rodina (clovek+)>
<!ATTLIST rodina odstavec %ano;>

<!-- Označíme, zda člověk patří do seznamu
     a zda se začíná nový odstavec -->
<!ELEMENT clovek (#PCDATA|hide)*>
<!ATTLIST clovek %attr;
                 tecka    %ano;
                 seznam   %ano;
                 odstavec %ano;>

<!ELEMENT hide (#PCDATA)>

<!ELEMENT umelec (#PCDATA)>
<!ATTLIST umelec %attr;>

<!-- Odstavec -->
<!ELEMENT p EMPTY>

<!-- Specifické příkazy pro TeX -->
<!ELEMENT tex (#PCDATA)>

<!-- Text zvýrazněný proložením -->
```

```
<!ELEMENT em (#PCDATA)>
```

Na začátku DTD si definujeme entity pro znaky, které nemáme v české abecedě, a parametrické entity, jež budeme dále v DTD využívat. Zbytek DTD je komentován, ale přesto potřebuje několik poznámek.

Členění do odstavců, jak je běžné v HTML i v DocBooku, by zde přidělalo práci a nebylo by účelné vzhledem k tomu, že odstavec zde není podstatnou stavební jednotkou textu. Proto používáme značku `<p/>` s významem *předěl odstavců*.

Element `<oddeleni>` má několik atributů. Text každého oddělení začíná plánkem a z prostorových důvodů se plánek sází jako perex s popisem buď umístěným vedle nebo přes plánek (když není obdélníkový a je v něm volné místo), jindy se sází přímo do sloupce. Umístění popisu je specifikováno atributem `popis`. Text oddělení nezačíná vždy na nové stránce. Prostředí MULTICOL může mít problémy, když perex je příliš vysoký. Řeší se to nepovinným parametrem, kdy tomuto prostředí sdělíme, že má přejít na novou stránku, je-li na stránce aktuální méně místa než zadaná hodnota. Tuto hodnotu zadáváme v atributu `space`. Atribut `extra` obsahuje volitelný LaTeXový kód, který ve výjimečných případech potřebujeme vložit.

Komentář si zaslouží elementy `<clovek>` a `<umelec>`. První z nich obsahuje údaje o pohřbené osobě. Protože jsou někteří pohřbení jen zmíněni, vyskytují se i uprostřed odstavce a nemají být uvedeni v rejstříku, máme pro tyto účely vytvořeny atributy. Obsahem elementu `<clovek>` je nějaká charakterizace, obvykle profesní (lékař, továrník, hudební skladatel apod.) příslušné osoby. Element `<umelec>` slouží k vyznačení autorů funerálního umění. Téměř vždy se v textu vyskytují ve standardní podobě, kterou vytvoříme programově, ale v některých případech je jméno uvedeno uprostřed věty v jiném než prvním pádě. Pak je požadovaný tvar uveden v obsahu tohoto elementu, ale ve většině případů je element `<umelec>` prázdný.

7.3 Pomocné nástroje

V části 6 jsme se zmínili o tom, že k validaci souboru lze využít XSLT. V předchozí části jsme uvedli, že element `<umelec>` je obvykle prázdný. Protože písařka občas omylem do tohoto elementu uzavřela text, který tam nepatří, bylo nutno najít nějaký systém, který by správnost elementu `<umelec>` ověřil. Proto byl vytvořen následující styl:

```
<?xml version="1.0" encoding="cp852"?>

<!--
Tento styl vypisuje umělce, kteří mají neprázdný obsah.
To je většinou špatně.
-->

<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
```

```
                version="1.0"
                xmlns:saxon="http://icl.com/saxon"
                extension-element-prefixes="saxon">

<xsl:output indent="no" method="text" encoding="cp852"
            saxon:character-representation="native"/>

<xsl:strip-space elements='*'/>

<!-- Použij šablony na předky elementu <umelec> -->
<xsl:template match='/|hrbitov|oddeleni|hrob'>
  <xsl:apply-templates/>
</xsl:template>

<!-- Pokud je obsah elementu neprázdný,
     vypiš jej a uveď informace pro jeho nalezení -->
<xsl:template match='umelec'>
  <xsl:if test=".!=''">
    <xsl:value-of select='saxon:systemId()'/>
    <xsl:text> line </xsl:text>
    <xsl:value-of select='saxon:lineNumber()'/>
    <xsl:text>&#10;</xsl:text>
    <xsl:value-of select='@jmeno'/><xsl:text>, </xsl:text>
    <xsl:value-of select='@krestni'/><xsl:text>: </xsl:text>
    <xsl:value-of select='../../@cislo'/><xsl:text>/</xsl:text>
    <xsl:choose>
      <xsl:when test="../@cislo">
        <xsl:value-of select='../@cislo'/>
      </xsl:when>
      <xsl:otherwise>
        <xsl:text>bez čísla --- </xsl:text>
        <xsl:value-of select='saxon:path()'/>
      </xsl:otherwise>
    </xsl:choose>
    <xsl:text>&#10;</xsl:text>
    <xsl:value-of select='normalize-space(.)'/>
    <xsl:text>&#10;&#10;</xsl:text>
  </xsl:if>
</xsl:template>

<!-- Ignoruj všechno ostatní -->
<xsl:template match='*|@*|text()'/>

</xsl:stylesheet>
```

Styl využívá některá rozšíření XSLT procesoru Saxon [3]. Díky tomu nám sdělí přesné místo neprázdného elementu `<umelec>` a my se rozhodneme, zda je to správně, nebo zda to vyžaduje opravu.

Jiným typem chyby byl zapomenutý element `<clovek>`. Za určitých okolností totiž nemusel být prvním uzlem v elementu `<hrob>` a validace pomocí DTD pak takovou chybu neodhalí. Lze to však řešit pomocí XSLT:

```
<?xml version="1.0" encoding="cp852"?>

<!--
Tento styl vypisuje hroby, kde není žádný člověk.
-->

<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
                version="1.0"
                xmlns:saxon="http://icl.com/saxon"
                extension-element-prefixes="saxon">

<xsl:output indent="no" method="text" encoding="cp852"
            saxon:character-representation="native"/>

<xsl:strip-space elements='*'/>

<xsl:template match='/|*'>
  <xsl:apply-templates/>
</xsl:template>

<!-- Všechny elementy <hrob>, v nichž není
     žádný element <clovek> -->
<xsl:template match='hrob[count(.//clovek)=0]'>
  <xsl:value-of select='saxon:systemId()'/>
  <xsl:text> line </xsl:text>
  <xsl:value-of select='saxon:lineNumber()'/>
  <xsl:text>: </xsl:text>
  <xsl:value-of select='../@cislo'/><xsl:text>/</xsl:text>
  <xsl:choose>
    <xsl:when test="@cislo">
      <xsl:value-of select='@cislo'/>
    </xsl:when>
    <xsl:otherwise>
      <xsl:text>bez čísla</xsl:text>
    </xsl:otherwise>
  </xsl:choose>
  <xsl:text>&#10;</xsl:text>
</xsl:template>
```

```
<!-- Ignoruj atributy a text -->
<xsl:template match='@*|text()'/>

</xsl:stylesheet>
```

Podobný nástroj byl vytvořen pro nalezení místa, kde se vyskytuje konkrétní zemřelý či konkrétní umělec. Vzhledem k tomu, že vstupních souborů bylo 22, ukázal se tento nástroj jako velmi užitečný.

XSLT našlo uplatnění i při fakturaci. Bylo nutno zjistit počet snímků, které byly nově fotografovány (byly v adresářích „1“ a „2“), počet fotografií z autorova archívu, které byly pouze skenovány (byly v adresáři „3“) a počet plánků. Tyto údaje zjistil následující styl:

```
<?xml version="1.0" encoding="cp852"?>

<!--
Tento styl zjišťuje počet fotek v jednotlivých adresářích
a~počet plánků
-->

<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
                version="1.0"
                xmlns:saxon="http://icl.com/saxon"
                extension-element-prefixes="saxon">

<xsl:output indent="no" method="text" encoding="cp852"
            saxon:character-representation="native"/>

<xsl:strip-space elements='*'/>

<!-- Celý výpočet dělá tato šablona -->
<xsl:template match='/'>
  <xsl:text>Adresář 1:    </xsl:text>
    <xsl:value-of select=
    'count(//img[substring-before(normalize-space(.),"/")="1"])'/>
  <xsl:text>&#10;</xsl:text>
  <xsl:text>Adresář 2:    </xsl:text>
    <xsl:value-of select=
    'count(//img[substring-before(normalize-space(.),"/")="2"])'/>
  <xsl:text>&#10;</xsl:text>
  <xsl:text>Adresář 3:    </xsl:text>
    <xsl:value-of select=
    'count(//img[substring-before(normalize-space(.),"/")="3"])'/>
  <xsl:text>&#10;</xsl:text>
  <xsl:text>Fotky celkem: </xsl:text>
    <xsl:value-of select='count(//img)'/>
```

```
  <xsl:text>&#10;</xsl:text>
  <xsl:text>Počet plánků: </xsl:text>
    <xsl:value-of select='count(//oddeleni[@planek])'/>
  <xsl:text>&#10;</xsl:text>
</xsl:template>

<!-- Ignoruj všechno ostatní -->
<xsl:template match='*|@*|text()'/>

</xsl:stylesheet>
```

7.4 Tvorba rejstříků

Kniha obsahuje jmenný rejstřík pohřbených osob a rejstřík tvůrců funerálního umění. Ve jmenném rejstříku se uvádí jméno člověka, jeho profesní charakterizace, datum narození a smrti a číslo stránky. Je to více údajů, než se v jiných knihách zpracovává programem MakeIndex, ale v zásadě by tento prolém byl standardními LaTeXovými prostředky řešitelný. Náročnější je rejstřík tvůrců funerálního umění. Ten obsahuje příjmení a jméno umělce, jeho datum narození a případně úmrtí a číslo stránky. Potíž je v tom, že v textu se nevyskytují vždy všechny údaje. Písařka jména umělců nezná a není schopna všechny údaje do elementů `<umelec>` doplňovat. Navíc by tím stoupl počet neprázdných elementů `<umelec>` v dokumentu a validace by pak byla obtížnější. Při sazbě knihy *Olšanské umění, jeho tvůrci a doba* [8] to bylo řešeno pomocným perlovským skriptem a následným tříděním programem CSR [6]. Nyní bylo doplňování údajů řešeno výhradně v XSLT. Navíc bylo třídění podle normy implementováno do procesoru Saxon [9]. Uvádíme jen základní šablonu, která provádí třídění. Šablony pro standardizovaný tisk některých částí uvedeny nejsou.

```
<!-- Šablona pro výpis rejstříku umělců -->
<xsl:template name='umelci'>
  <xsl:variable name='jmeno' saxon:assignable='yes'/>
  <xsl:variable name='krestni' saxon:assignable='yes'/>
  <xsl:variable name='narozen' saxon:assignable='yes'/>
  <xsl:variable name='smrt' saxon:assignable='yes'/>
  <xsl:variable name='odkaz' saxon:assignable='yes'/>
  <xsl:variable name='novy' saxon:assignable='yes'/>
  <xsl:variable name='fn'>
    <xsl:value-of select='$dir'/>
    <xsl:text>umelci.tex</xsl:text>
  </xsl:variable>
  <xsl:text>\input{</xsl:text>
  <xsl:value-of select='$fn'/>
  <xsl:text>}&#10;</xsl:text>
  <xsl:document indent="no" method="text"
                encoding="cp852" href="{$fn}"
```

```
            saxon:character-representation="native">
<xsl:text>\typeout{Tvůrci funerálního umění</xsl:text>
<xsl:text> File: </xsl:text>
<xsl:value-of select='saxon:systemId()'/>
<xsl:text> line </xsl:text>
<xsl:value-of select='saxon:lineNumber()'/>
<xsl:text>}&#10;</xsl:text>
<xsl:text>\begin{umelci}&#10;</xsl:text>
<!-- Setřídění všech umělců -->
<xsl:for-each select='//umelec'>
  <xsl:sort select='@jmeno' lang='cs'/>
  <xsl:sort select='@krestni' lang='cs'/>
  <!-- Zjišťujeme, zda je to nový umělec,
       data jsou dána jen někdy -->
  <xsl:if test="$jmeno!=''">
    <xsl:if test='@jmeno!=$jmeno'>
      <saxon:assign name='novy' select='1'/>
    </xsl:if>
    <xsl:if test='$krestni'>
      <xsl:if test='$krestni!=@krestni'>
        <saxon:assign name='novy' select='1'/>
      </xsl:if>
    </xsl:if>
    <xsl:if test='$narozen'>
      <xsl:if test='$narozen!=@narozen'>
        <saxon:assign name='novy' select='1'/>
      </xsl:if>
    </xsl:if>
    <xsl:if test='$smrt'>
      <xsl:if test='$smrt!=@smrt'>
        <saxon:assign name='novy' select='1'/>
      </xsl:if>
    </xsl:if>
    <xsl:if test='$novy'>
      <!-- Vypsání jména a vymazání proměnných -->
      <xsl:call-template name='umelec-v-rejstriku'>
        <xsl:with-param name='jmeno' select='$jmeno'/>
        <xsl:with-param name='krestni' select='$krestni'/>
        <xsl:with-param name='narozen' select='$narozen'/>
        <xsl:with-param name='smrt' select='$smrt'/>
        <xsl:with-param name='odkaz' select='$odkaz'/>
      </xsl:call-template>
      <saxon:assign name='krestni'/>
      <saxon:assign name='narozen'/>
      <saxon:assign name='smrt'/>
```

```
          <saxon:assign name='odkaz'/>
          <saxon:assign name='novy'/>
        </xsl:if>
      </xsl:if>
      <!-- Definice hodnot proměnných -->
      <saxon:assign name='jmeno'>
        <xsl:value-of select='@jmeno'/>
      </saxon:assign>
      <xsl:if test='@krestni'>
        <saxon:assign name='krestni'>
          <xsl:value-of select='@krestni'/>
        </saxon:assign>
      </xsl:if>
      <xsl:if test='@narozen'>
        <saxon:assign name='narozen'>
          <xsl:value-of select='@narozen'/>
        </saxon:assign>
      </xsl:if>
      <xsl:if test='@smrt'>
        <saxon:assign name='smrt'>
          <xsl:value-of select='@smrt'/>
        </saxon:assign>
      </xsl:if>
      <!-- Vytvoření odkazu: id -->
      <saxon:assign name='odkaz'>
        <xsl:value-of select='$odkaz'/><xsl:text> </xsl:text>
        <xsl:value-of select='generate-id()'/>
      </saxon:assign>
    </xsl:for-each>
    <!-- Konec cyklu, takže musíme vypsat posledního umělce -->
    <xsl:if test="$jmeno!=''">
      <xsl:call-template name='umelec-v-rejstriku'>
        <xsl:with-param name='jmeno' select='$jmeno'/>
        <xsl:with-param name='krestni' select='$krestni'/>
        <xsl:with-param name='narozen' select='$narozen'/>
        <xsl:with-param name='smrt' select='$smrt'/>
        <xsl:with-param name='odkaz' select='$odkaz'/>
      </xsl:call-template>
      <saxon:assign name='krestni'/>
      <saxon:assign name='narozen'/>
      <saxon:assign name='smrt'/>
      <saxon:assign name='odkaz'/>
    </xsl:if>
    <xsl:text>\end{umelci}&#10;</xsl:text>
  </xsl:document>
```

```
</xsl:template>

<!-- Výpis umělce v rejstříku -->
<xsl:template name='umelec-v-rejstriku'>
  <xsl:param name='jmeno'/>
  <xsl:param name='krestni'/>
  <xsl:param name='narozen'/>
  <xsl:param name='smrt'/>
  <xsl:param name='odkaz'/>
  <!-- Proměnná pro tisk iniciál -->
  <xsl:variable name='init'>
    <xsl:value-of select='ibs:cs-initial($jmeno)'/>
  </xsl:variable>
  <xsl:if test='$init!=$initial'>
    <xsl:text>\uminic </xsl:text>
    <xsl:value-of select='$init'/>
    <xsl:text>&#10;</xsl:text>
    <saxon:assign name='initial'>
      <xsl:value-of select='$init'/>
    </saxon:assign>
  </xsl:if>
  <!-- Tisk jména a dat -->
  <xsl:text>\umelec{</xsl:text>
  <xsl:call-template name='str-rep'>
    <xsl:with-param name='string'>
      <xsl:value-of select='$jmeno'/>
      <xsl:if test="$krestni!=''">
        <xsl:text> </xsl:text><xsl:value-of select='$krestni'/>
      </xsl:if>
      <xsl:call-template name='roky'>
        <xsl:with-param name='narozen'>
          <xsl:value-of select='$narozen'/>
        </xsl:with-param>
        <xsl:with-param name='smrt'>
          <xsl:value-of select='$smrt'/>
        </xsl:with-param>
      </xsl:call-template>
    </xsl:with-param>
  </xsl:call-template>
  <xsl:text>}\umpage </xsl:text>
  <!-- Rozeber odkazy, oddělovač je definován
       v LaTeXovém stylu -->
  <xsl:for-each select='saxon:tokenize($odkaz)'>
    <!-- Třídění není použitelné! -->
    <xsl:text>{</xsl:text>
```

```
    <xsl:value-of select='.'/>
    <xsl:text>}</xsl:text>
  </xsl:for-each>
  <!-- Konec řádku -->
  <xsl:text>\umpage&#10;</xsl:text>
</xsl:template>
```

Zbývá ještě vyřešit odkaz na správnou stránku. Používáme-li MakeIndex, máme v nesetříděném souboru čísla stránek uvedena. Pomocí XSLT však rejstřík vytváříme ještě před zahájením sazby a čísla stránek tudíž neznáme. Vytvoříme tedy jednoznačný identifikátor funkcí `generate-id{}`. Při sazbě textu jej použijeme jako jméno návěští v makru `\label`, v rejstříku jej použijeme v makru `\ref`. LaTeXovou část rejstříku tvůrců funerálního umění pak vysázíme těmito makry:

```
% Rejstřík umělců

\newenvironment{umelci}{\clearpage \let\Mezera\space
  \begin{multicols}{2}[\NadpisOddeleni
        {Tvůrci funerálního umění}]}%
  {\end{multicols}}

\def\uminic #1 {\par\vb\textbf{#1}\par\nobreak}

\def\umelec #1{\par
  \umpagetoks{}\def\delim{~\dots~\ignorespaces}\noindent
  \hangafter 1 \hangindent\parindent
  \rightskip 0mm plus 2em \relax #1}

\def\umprint #1.{\let\next\umprint
  \ifx \relax #1\let\next\par
  \else
    \delim #1\def\delim{, \ignorespaces}\fi \next}

\newtoks\umpagetoks

\def\umpage #1{\let\next\umpage
  \ifx \umpage #1\def\next{\expandafter
                \umprint\the\umpagetoks\relax.}\else
    \expandafter\ifx\csname r@#1\endcsname\relax
      \@latex@warning{Reference `#1' on page \thepage
                    \space undefined}\else
      \expandafter\expandafter\expandafter \edef
      \expandafter\expandafter\expandafter \test
      \expandafter\expandafter\expandafter
      {\expandafter\expandafter\expandafter
           \@secondoftwo\csname r@#1\endcsname}%
```

```
      \ifcat$\the\umpagetoks$\umpagetoks\expandafter{\test.}\else
        \def\next{\expandafter \expandafter \expandafter
             \umsortstart \expandafter \test
             \expandafter .\the\umpagetoks \relax.}\fi \fi \fi
  \next}

\def\umsortstart{\umpagetoks{}\umsort}
\def\umsort #1.#2.{\ifx \relax #2%
    \umpagetoks\expandafter
      {\the\umpagetoks #1.}\let\next\umpage\else
      \ifnum #1>#2
        \umpagetoks\expandafter{\the\umpagetoks #2.}%
        \def\next{\umsort #1.}\else
        \let\next\umsortend
        \ifnum #1=#2
          \umpagetoks\expandafter{\the\umpagetoks #2.}\else
          \umpagetoks\expandafter{\the\umpagetoks #1.#2.}%
        \fi
      \fi
    \fi \next}

\def\umsortend #1\relax.{\umpagetoks
    \expandafter{\the\umpagetoks #1}\umpage}
```

V původní verzi procesoru Saxon bylo zajištěno, že abecedně seřazené identifikátory získané funkcí `generate-id()` odpovídaly jejich pořadí v dokumentu. Později však z důvodu zvýšení rychlosti zpracování od toho bylo upuštěno [2]. V rejstříku ovšem chceme čísla stránek setříděná. Může se též stát, že je některý umělec na téže stránce uveden dvakrát. V rejstříku však nechceme duplikovaná čísla. Dosahujeme toho cviky s token registrem `\umpagetoks`. Analýzu příslušných maker ponecháváme čtenářům za domácí cvičení.

8 Závěr

Při sazbě knihy, použité v případové studii, se prokázalo, že XML může významně pomoci i tehdy, požadujeme-li pouze jediný výstupní formát. Navíc došlo k nasazení XML v čistě komerční oblasti, kde za špatné rozhodnutí platíme skutečnými penězi. Zde však XML pomohlo k časové, a tím i finanční úspoře. Současně byl vyvrácen mýtus, že běžná sekretářka není schopna psát dokumenty v LaTeXu, natož v XML. Nelze pochopitelně předpokládat, že si sekretářka naprogramuje LaTeXový makrobalík, transformační styl pro XSLT či formátovací objekty, ale pokud jí někdo potřebné nástroje dodá, pak se je naučí používat stejně, jako se naučila naklikat typ a velikost písma, zarovnání odstavce a jiné volby v nejrůznějších dialogových oknech kancelářského textového editoru. V tomto případě stačila patnáctiminutová instruktáž o základech XML a když si odmyslíme běžné drobné překlepy, napsala písařka celý text bez chyb.

9 Poděkování

Chtěl bych poděkovat Jiřímu Koskovi za přehledně psanou knihu *XML pro každého*, bez jejíhož přečtení bych asi podrobnější knihy o XML nepochopil, i za jeho cenné rady, které mi při práci na knize *Olšanské umění, jeho tvůrci a doba*, jež byla mým prvním větším projektem v XML, ochotně poskytoval. Dále děkuji Michaele Knapové, která, ač není odbornicí v informačních technologiích a nemá žádný specializovaný editor pro práci s XML, přepsala celý text z rukopisu v podstatě bezchybně.

Reference

1. DocBook web page, `http://www.oasis-open.org/docbook/`
2. Kay M. H.: Soukromé sdělení.
3. Kay M. H.: XSLT procesor Saxon, `http://users.iclway.co.uk/mhkay/`
4. Kosek J.: XML pro každého. Grada Publishing, 2000. `http://www.kosek.cz/xml/`
5. Neckářová L., Vanoušek A., Wagner J.: Vinohradský hřbitov včera & dnes. Správa pražských hřbitovů, Praha 2002. (Publikace je určena pro vnitřní potřebu členů Klubu Za starou Prahu a spolku Svatobor.)
6. Olšák P.: Program csr (Czech Sort) – abecední řazení podle normy. Zpravodaj Československého sdružení uživatelů TEXu 3(1994), 126–139. `http://math.feld.cvut.cz/olsak/`
7. Rahtz S.: PassiveTEX, `http://users.ox.ac.uk/~rahtz/passivetex/`
8. Vanoušek A: Olšanské umění, jeho tvůrci a doba. Správa pražských hřbitovů, Praha 2000. (Publikace je určena pro vnitřní potřebu členů Klubu Za starou Prahu a spolku Svatobor.)
9. Wagner Z.: Nástroje pro práci s XML/XSLT, `http://www.icebearsoft.cz/icebearsoft.euweb.cz/xml/`

Korektury a korektoři v 21. století

Tomáš Hála

[1] Ústav informatiky, Provozně ekonomická fakulta
Mendelova zemědělská a lesnická univerzita v Brně
Zemědělská 1, 613 00 Brno
Email: thala@mendelu.cz
[2] KONVOJ, spol. s r. o., Berkova 22, 612 00 Brno
Email: konvoj@konvoj.cz

Abstrakt: Je potřeba i v dnešní době dělat korektury? Při zpracování textu – jak ukazují různé publikace – ani moderní technologie nedokáží zabránit různým chybám. Článek obsahuje základní přehled problematiky korektur: rozdělení korektur, organizační a metodické postupy při provádění korektur, korekturní znaménka. Jsou srovnány klasické postupy se současností, ovlivněnou výpočetní technikou. Na závěr jsou připojeny vybrané prohřešky se zaměřením na počítačovou terminologii.

Klíčová slova: korektury, dělení korektur, technické a organizační postupy, anglická terminologie

1 Druhy korektur

Korektury můžeme dělit jednak podle hlediska časové posloupnosti při zpracování, jednak podle hlediska obsahového.

1.1 Časová posloupnost korektur

„Klasické“ rozdělení korektur je ovlivněno postupy používanými v klasické sazbě, kdy autor zhotovoval rukopis v podobě strojopisu, ze kterého sazeč zhotovil opisem sazbu. Toto rozdělení (Pop, Flégr a Pop, 1989) obsahuje korekturu

- domácí sloupcovou,
- autorskou sloupcovou,
- domácí stránkovou,
- autorskou stránkovou.

Domácí korektura se provádí v sazárně, autorskou korekturu provádí autor nebo redaktor.

Současnost: V době „převálcování“ lidstva výpočetní technickou se stalo nepsanou zvyklostí, že autor dodává rukopis v elektronické podobě. Tato elektronická podoba však může vypadat různě, následující čtyři skupiny lze považovat za nejdůležitější:

Jan Kasprzak, Petr Sojka (editoři): SLT 2002 – sborník semináře o Linuxu a TEXu, str. 49–55, 2002.

a) rukopis v podobě souboru s prostým textem,
b) rukopis zapsaný v určitém programovém produktu určeném pro zpracování dokumentů (jiném, než ve kterém se provádí sazba),
c) rukopis doplněný o strukturní informace (označkovaný dokument), strukturní značky bývají zpravidla dohodnuty předem,
d) rukopis připravený pro sazbu v konkrétním, předem dohodnutém sázecím systému.

Volba způsobu odevzdání rukopisu je podmíněna autorovými znalostmi a také zvládnutím určitých počítačových dovedností.

Široké spektrum možností pořízení rukopisu je současně příčinou existence rozličných způsobů zpracování dokumentů, a tudíž i mnoha různých postupů při korekturách. Tabulka 1 ukazuje, které korektury přetrvávají i v současnosti při zodpovědného zpracování dokumentu.

Tabulka 1. Korektury při rukopisu v elektronické podobě

	a)	b)	c)	d)
domácí sloupcová	ano	ano[1]	ano/ne[2]	ne
autorská sloupcová	ano[3]	ano[1]	ne	ne
domácí stránková	ano	ano	ano	ano
autorská stránková	ano	ano	ano	ano

[1] – sloupcové korektury u varianty b) jsou nezbytné, protože před vlastní sazbou je třeba konvertovat z dodaného formátu

[2] – strukturně značkované dokumenty je nutné kontrolovat především z hlediska správného značkování autorem, chyby vznikající při převodu ze strukturního značení by měly být zpravidla zachyceny již při testování převodního programu

[3] – lze vynechat u hladkých textů

Z uvedeného je zřejmé, že použití výpočetní techniky odstraňuje z výrobního procesu především sloupcové korektury, zejména autorskou sloupcovou.

Vynechání těchto korektur je však možné jen při správně připraveném dokumentu, neboť jen v těchto případech lze předložení rukopisu v elektronické podobě chápat jako předložení již korigovaného rukopisu.

I přesto, že autorskou sloupcovou korekturu si může autor v některých případech udělat sám, je předložení autorské stránkové korektury nezbytností, neboť autor na svoji korekturu nárok podle autorského zákona.

1.2 Dělení podle obsahového hlediska

Z obsahové hlediska dělíme korektury na tyto kategorie:

gramatická korektura slouží k odstranění pravopisných chyb, překlepů, vadné interpunkce apod.;
stylistická korektura je do jisté míry totožná s jazykovou úpravou díla;

typografická korektura má za cíl například kontrolu správnosti použitých řezů písma, kontrolu správného vysazení symbolů (matematických, cizojazyčných znaků, mezerování), kontrolu horizontálního i vertikálního odsazení jednotlivých objektů, vysazení tabulek a zalomení obrázků;

formální korektura obsahuje kontrolu správnosti zápisu bibliografických citací, kontrolu počtu obrázků či tabulek a jejich číslování vůči odkazům v textu, kontrolu správného způsobu vyznačování aj., jejím cílem je také sjednocení úpravy těch publikací, které jsou sestavovány různými autory;

grafická korektura se provádí zejména tehdy, jsou-li součástí publikace barevné obrazové přílohy, obsahuje kontrolu barevnosti, kontrastu mezi písmem a pozadím u plnobarevných stran,

technická korektura patří do úkonů prováděných technickým redaktorem, který kontroluje číslování stran, vyřazení na archy, kvalitu podkladů pro tisk (například u předloh pro tisk z laserové tiskárny je třeba kontrolovat nejen kvalitu pokrytí tonerem, ale také případné zašpinění předlohy například špatným válcem) apod.

Ve své podstatě uvedené dělení podle obsahu nepřináší žádný nový prvek do současné korektorní praxe, neboť se používá již od dávných dob. Zcela určitě se však změnil důvod, proč toto dělení využíváme.

Současnost: Rozdělení korektur podle obsahu nabývá na významu právě s masovým rozšířením „domácího“ publikování. Ne každý, kdo v současnosti připravuje publikace do tisku, má takové vzdělání a takovou kvalifikaci, aby mohl sám zaručit, že dílo je po všech stránkách dokonalé.

Jen velká nakladatelství či vydavatelství si mohou dovolit zaměstnávat korektory na plný úvazek. Menší firmy, kterých je v nakladatelském oboru většina, si musejí poradit sami nebo si najímat specializované firmy či jednotlivce. Od externího korektora (například jednotlivce-jazykáře) nelze očekávat provedení korektur technických, mnohdy ani typografických či formálních. Stejně tak specialista na grafické korektury je schopen přehlédnout i pravopisné chyby v textech souvisejících s grafickým zpracováním.

Dokonce ani instituce, jakými jsou například vysoké školy s ediční činností, nemají vždy mezi zaměstnanci osobu odpovědnou a současně i kvalifikovanou k provádění korektur. Do jisté míry lze dokonce hovořit o vymizení profese „klasických“ korektorů.

Proto rozdělení korektur do určitých oblastí je nezbytné k zajištění správného zpracování díla. Jeho nevýhodou je skutečnost, že vede ke specializaci jednotlivých pracovníků a neumožňuje komplexní pohled na celý korekturní postup.

Kromě externistů je možné si v dnešní době zadat provedení korektur specializované firmě. Podle živnostenského zákona jsou korektorské práce součástí nakladatelské činnosti, která patří mezi živnosti volné, k nimž není třeba žádné předchozí kvalifikace či praxe. I proto je úroveň firem poskytujících korektorské služby velmi rozdílná a – stejně jako u samostatných korektorů – ne všechny firmy jsou schopny provést kvalitně všechny druhy korektur.

2 Postupy při provádění korektur

2.1 Čtení korektur

Čtením korektur nazýváme činnost, při níž srovnává korektor otisk sazby s rukopisem. S ohledem na několikastupňové korektury lze jejich čtení definovat jako srovnání otisku předchozích korektur s otiskem aktuálních korektur.

Pop, Flégr a Pop (1989) uvádějí, že srovnání textů je spolehlivější, čte-li korektor nejprve otisk sazby a potom rukopis (obecně: nejprve otisk nových korektur, potom starých).

Současnost: Často kladenou otázkou je, zda lze provádět čtení korektur z monitoru. Odpověď je prostá – možné to je, ale buď za cenu snížení kvality prováděných korektur, nebo s neúměrně vysokou zátěží pro oči. Pro toto tvrzení nejsou k dispozici vědecky ověřené výsledky, je to však zkušenost všech korektorů, se kterými jsem kdy přišel do styku.

Problémem v této souvislosti může být korektura webových stránek, které zpravidla nemívají tištěnou podobu.

Zde se jako schůdné řešení ukázalo rozdělení korektur na dvě části, analogické k sloupcovým a stránkovým korekturám. Při „sloupcové“ korektuře se kontroluje správnost textu (gramatická, stylistická, formální, případně i typografická – pokud se dá vůbec hovořit webu o typografii), přičemž pro tyto korektury lze text vytisknout buď z prohlížeče, v případě gramatických a stylistických korektur lze dokonce využít i původního zdroje, pokud jsou stránky generovány (ze SGML, z databází aj.). provedení korektur těchto dokumentů

Druhá fáze – „stránkové“ korektury – slouží ke kontrole celkové grafické úpravy dokumentu, což je možné provádět na monitoru. Součástí této fáze je také kontrola platnosti hypertextových odkazů, přítomnosti všech obrázků, stylů apod., což bez počítače již nejde.

2.2 Použití korekturních znamének

Korekturní znaménka jsou popsána starší normou (ČSN 88 0410).

Dělí se do šesti základních skupin (výměna, vypuštění a vsunutí; změny v sazbě; změny mezer; odstavec a změny písma; oprava technický nedostatků sazby; zrušení korektury). Kromě zmíněné normy jsou vybraná, důležitější znaménka uváděna v celé řadě publikací (Pop, Flégr a Pop, 1989; Beran, 1994; Hanáček a kol., 1996; Kočička a Blažek, 2000 aj.).

Současnost: Je třeba si přiznat, že ani změna technologie sazby nezpůsobila výraznější zásah do množiny chyb či technických nedostatků sazby.

Existuje sice několik značek, které v dnešní době buď nepoužijeme vůbec, nebo jen velmi zřídkakdy (otočení řádku o 180°, odstranění hrotků apod.), většina značek je však i dnes aktuální.

2.3 Vyznačování korektur

Vyznačování provádí korektor dvojmo: jednou přímo v místě, kde se vyskytuje chyba, podruhé na okraji otisku korektur, vždy ve stejné výšce s řádkem s korigovanou chybou.

Znaménka se nesmějí opakovat na jednom řádku, případně v řádcích po sobě jdoucích z důvodu možné nejednoznačnosti výkladu korektury.

Vyznačování se provádí modrou nebo černou barvou.

Současnost: Stávající pravidla pro vyznačování korektur jsou platná i dnes, jen je třeba je rozšířit o jevy, které souvisejí s počítačovým zpracováním textů.

S ohledem na možnost tvorby stylů snad již ve všech sázecích systémech se velmi často vyskytují chyby, které jsou způsobeny nedokonalostí stylu. Tyto chyby se objevují v celé řadě míst připravovaného díla. Není proto nutné tyto chyby vyznačovat v místě každého výskytu, pro sazeče stačí poznámka pouze u výskytu prvního. Je otázkou, do jaké míry dokáže korektor rozpoznat, zda se jedná o chybu stylovou či nikoliv. Pro kvalifikované rozhodnutí (ušetří tím práci sobě i sazečovi) je nezbytné jeho širší vzdělání v oblasti zpracování textu.

Stejným způsobem, tzn. v místě prvního výskytu je vhodné sazeče upozornit na systematické chyby v textu, které může sazeč opravit pomocí služby hledání a nahrazení.

Vyznačování modrou či černou barvou lze jen doporučit. Červená barva bývá (záleží na vnitřní pokynech) určena pro korektora-revizora, který dohlíží na provádění korektur. Tyto „revizorské“ opravy provedené jsou barevně odlišeny a mohou sloužit jako podklad pro další vzdělávání korektorů, případně také pro upřesňování vnitropodnikových norem.

Červenou barvu lze také doporučit pro označení oprav ve stylu či oprav systematických.

2.4 Korektor

2.4.1 Kompetence korektora Od korektora se očekává dokonalá znalost českého pravopisu, znalost norem pro zpracování dokumentů nebo jeho částí a taktéž znalost typografie a estetických pravidel. Dříve se vyžadovala také znalost slovenského pravopisu (Pop, Fléger a Pop, 1989).

Současnost: Po rozpadu federace není nezbytná již znalost slovenského jazyka, neboť v České republice se se slovenštinou setkáme nejčastěji jen v odborných či vědeckých publikacích (sbornících, časopisech).

Častější je potřeba angličtiny, kromě odborných publikací obsahujících vícejazyčné abstrakty či klíčová slova se často používají v textech anglické výrazy, názvy, jména, případně bibliografické citace anglicky psaných děl.

2.4.2 „Časová a prostorová složitost“ korektur Čtení korektur vyžaduje značnou pečlivost a soustředěnost. Je všeobecně známo, že soustředěnost na práci tohoto typu kolísá v čase. Každý korektor by si měl proto stanovit podle svých individuálních schopností množství textu, které je schopen jednorázově

zpracovat, odhadnout nutnou dobu relaxace a celkovou denní dobu, po kterou je schopen korektury provádět. Při stanovení uvedených časových údajů je nutné vzít v úvahu, že jednotlivé typy korektur jsou z hlediska soustředění různě časově náročné.

Z těchto údajů lze celkem spolehlivě odhadnout časovou náročnost celého korekturního zpracování díla, což je nezbytné pro plánování celého výrobního procesu i pro stanovení dodacích lhůt v případě zakázkové činnosti.

Vyšší výkonnosti lze dosáhnout optimálním uspořádáním pracovního prostředí. Pokyny pro ergonomické uspořádání pracovního prostředí, hygienické normy další aspekty jsou zachyceny ve specializované literatuře.

3 Vybrané omyly autorů (i korektorů) v počítačové terminologii

koncovka *-ovský* × *-ový* , např. webový × webovský – správný tvar je webový, stejně tak internetový/-á (prohlížeč, adresa), nikoliv internetovský/-á

názvy firem či produktů jsou zpravidla registrovanými ochrannými nebo obchodními známkami, a proto je píšeme zásadně podle registrovaného názvu (Linux, UNIX, PostScript, CorelDRAW, MS-DOS, MS Windows, Windows NT; Hewlett-Packard, KONVOJ). V některých případech se může změnou písmen dokonce jednat i o rozdílný význam (SCO UNIX jako ochranná známka × Unix jako popis vlastností operačního systému).

skloňování názvů firem či produktů způsobuje změnu psaní malých a velkých písmen, název firmy či produktu se mění v tzv. zkratková slova, která se píší s velkým písmenem na začátku (v Dosu, do Linuxu),

přídavná jména odvozená z názvů firem či produktů píšeme výhradně malými písmeny (linuxový, unixový, postscriptový, dosový)

zkratka WWW není přídavným jménem, ale podstatným, a proto ji ve spojení s dalším podstatným jménem používáme v pozici přívlastku neshodného (stránky WWW, prohlížeč WWW aj., nikoliv WWW stránky, což známe například z němčiny – WWW-Seiten)

4 Terminologický slovníček

číst korekturu proofread, correct the proofs
čtení korektur proof reading
imprimatur imprimatur (pass for press)
imprimované stránky press proofs
imprimovaný rukopis hard copy
imprimovat pass for press, pass the proofs for press
korektor reader, proofreader, copy editor
korektorna proofreading room
korektorova značka proofreader's mark
korektorství correcting, proofreader's job

korektura book proof, proof correction, proof reading
korektura (obtah) proof, proof-sheet
korektura domácí (vnitřní) first proof, house correction
korektura autorská author's proofs (author's corrections)
korektura nakladatelská editor's proofs
korektura stránková page proof
korektura tiskárenská first proof, rough proof
korekturní značka, znaménko proof correction mark, correction mark
rukopis manuscript, copy, handwritting

5 Závěr

Problematika provádění korektur je velmi rozsáhlá a její podrobný rozbor přesahuje rámec tohoto příspěvku. Proto jsou zde uvedeny jen podle názoru autora nejdůležitější body, kterými je třeba se zabývat při korektorské činnosti.

Autor rád přivítá jakékoliv další podněty, připomínky či odkazy k tématu na své emailové adrese.

Reference

1. Beran, V. *Typografický manuál.* 1. vyd. Manuál: Náchod, 1994. ISBN 80-901824-0-2.
2. *Česko-anglický a anglicko-český slovníček nakladatelské a tiskařské terminologie.* 1. vyd. 99 s. International Book Development: Londýn a SČNK: Praha, 1998. ISBN 80-902495-1-5.
3. ČSN 88 0410: Korekturní znaménka.
4. Hanáček, P. a kol. *Jak publikovat na počítači.* 1. vyd. 216 s. Science: Veletiny, 1996. ISBN 80-901475-7-7.
5. Kočička, P., Blažek, F. *Praktická typografie.* 1. vyd. 310 s. Computer Press: Praha, 2000.
6. Pop, P., Fléger, J., Pop, V. *Sazba I Ruční sazba.* 2. vyd. 188 s. SPN: Praha, 1989.
7. Zákon č. 121/2000 Sb., o právu autorském (autorský zákon).
8. Zákon č. 356/1999 Sb., o živnostenském podnikání (živnostenský zákon).

Automatizovaná tvorba grafů v systémech na bázi TEXu

Jiří Rybička

Ústav informatiky
Provozně ekonomická fakulta
Mendelova zemědělská a lesnická univerzita v Brně
Zemědělská 5, 613 00 Brno
Email: rybicka@mendelu.cz

Abstrakt: Typografické systémy založené na systému TEX jsou často používány pro realizaci odborných, technických a vědeckých publikací, kde se ve značné míře používají grafy pro znázornění souboru hodnot. Tvorba grafů je dána řadou pravidel, jejichž cílem je vytvoření snadno čitelného, přehledného a srozumitelného výsledku. Jedním ze základních typografických pravidel je použití identických vizuálních prostředků pro objekty identického významu. Článek se zabývá některými základními zásadami tvorby grafů a také automatizovanou podporou pro jejich dávkovou tvorbu.

1 Úvod

V odborných pracích se často data prezentují grafickou formou. Forma grafu bývá vhodným doplňkem dat vyjádřených tabelárně. Hlavním účelem grafu je zvýšení přehlednosti dat a vyjádření charakteru, který není například z tabulky na první pohled patrný. Na druhé straně však graf neposkytuje možnosti tak přesného odečítání hodnot jako tabulka, proto se obě formy vyjádření vzájemně doplňují.

2 Grafické znázornění

Grafické znázornění je vyjadřovací forma používaná k zobrazení zkoumaných jevů. Jejím výsledkem je nákres, který je sestaven pomocí souboru grafických prostředků. Nákres může svým charakterem být [2]:

- *schéma* – v hlavních, základních rysech naznačené složení, náčrt procesu, jevu, zákonitosti;
- *diagram* – grafické (geometrické) znázornění průběhu nějakého procesu, vztahu či závislosti, nejčastěji soustavou kartézských os.

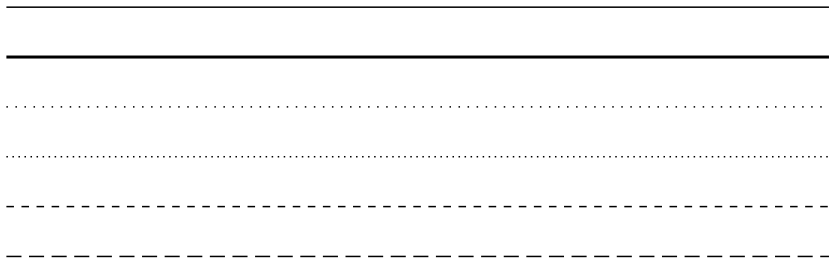
Používané grafické prostředky mohou mít význam *ideografický* nebo *geometrický*.

Jan Kasprzak, Petr Sojka (editoři): SLT 2002 – sborník semináře o Linuxu a TEXu, str. 57–67, 2002.

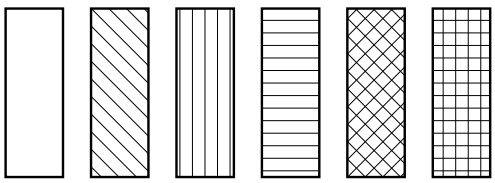
2.1 Ideografické prostředky

Ideografické prostředky mají kvalitativní význam (číslice, písmena, schematické tvary, čáry, šrafování, barvy apod.). Jsou základním grafickým prostředkem pro kreslení schémat, ale používají se i u diagramů.

Čáry jsou velmi frekventovanou rekvizitou ve všech druzích schémat a diagramů. Musí být od sebe snadno rozeznatelné, aby byl graf čitelný, bez optických klamů. Základním typem čáry je čára plná. Používá se vždy k vyznačení nejvýznamnější skutečnosti v daném grafu. Dalšími typy čar jsou čárkovaná, tečkovaná, čerchovaná a kombinovaná. Používají se ke znázornění dalších skutečností. Některé čáry mají svůj ustálený význam, například čárkovaná čára se používá ke znázornění průměrné hodnoty. Čáry lze dále odlišit tloušťkami, případně i barvami. Tloušťky čar odpovídají typografickým zásadám. Jemné linky mají tloušťky kolem šestnáctiny petitu (0,5 bodu), tlustší linky jsou osminky (1 b). Pro šrafy lze využít i tloušťky čtvrtbodové. Barevné podání lze řešit kombinací černé a pestré barvy. Je-li třeba použít více barev, přidávají se kontrastní barvy z opačné strany spektra.



Různé druhy čar používané v grafech



Různé druhy šrafování používané v grafech

Obrázek 1. Ideografické prostředky grafů

Číslice a písmena tvoří důležitou identifikační složku. Pro kvalitní čtení je nezbytné, aby se stupeň použitého písma pohyboval u nejmenších textů nad hranicí 6 bodů. Popisky os nebo zobrazených hodnot je vhodné použít písmo 8–9 bodů. Není-li na popisky dost místa, je vhodné použít zkratky vysvětlené v textu, nikoliv zmenšovat stupeň použitého písma. S výhodou lze použít některého bezserifového typu písma, které je i v menším stupni zřetelnější. U všech textů a čísel je nezbytné respektovat všechna typografická a pravopisná pravidla.

2.2 Geometrické prostředky

Jedná se o prvky, které svou velikostí nebo vzdáleností naznačují úroveň sledované hodnoty. Mohou to být geometrické útvary – bod, úsečka, plocha, těleso – a symbolické tvary – například různě velké figurky. V některých případech se ideografický prostředek v různém provedení může stát prostředkem geometrickým, například šrafování o různé hustotě vyjadřující smluvené hodnoty.

2.3 Grafický výklad

K rozlišení kvalitativního a kvantitativního významu grafických prvků slouží *grafický výklad*. Určuje také zásady, podle nichž se graf má číst. Mezi základní prostředky výkladu grafu patří soustava souřadnic, stupnice a moduly, legenda, obsahující přehled všech grafických prvků, dále pak název grafu, podtitul, všechny vysvětlivky, poznámky a podobně.

Soustava souřadnic je soustava čar, jejichž významem je kvalitativní orientace libovolného bodu pomocí nejkratší vzdálenosti od os, určované jejich stupnicemi. Nejčastěji se využívá kartézská soustava pravoúhlých souřadnic, kterou tvoří dvě na sebe kolmé přímky rozdělující rovinu na čtyři kvadranty. Vodorovná přímka je *osa úseček*, značená symbolicky x, svislá přímka se nazývá *osa pořadnic* (označení y). Průsečík se nazývá *počátek* a dělí každou osu na dvě poloosy. Na osu x se vždy vynáší nezávisle proměnná, na osu y pak hodnoty závisle proměnné.

Při okótování bodů na osách souřadné soustavy získáme *stupnici*. Proto se osa také nazývá *nositelka stupnice*. Při tvorbě stupnice je nejdůležitější správně stanovit základní délkovou jednotku úsečky, která odpovídá zobrazovanému číselnému intervalu. Tato délka je jedním z rozhodujících faktorů čitelnosti grafu. Vypočtená délka úsečky také určuje přesnost čtení údajů. Stupnice mohou být rovnoměrné (lineární) a nerovnoměrné (například logaritmické).

Modul stupnice M se určuje u rovnoměrných stupnic v závislosti na ploše, která je grafu určena. Je dán výrazem

$$M = \frac{d}{H - D},$$

kde d je požadovaná délka stupnice, D – dolní mez hodnot zobrazovaného souboru dat, M – horní mez hodnot zobrazovaného souboru dat. Jednotka modulu je dána délkovou jednotkou.

Nejmenší dílek stupnice η by neměl klesnout pod 1 mm. Nejmenší dílek stupnice určuje přesnost čtení hodnot ϵ, která je dána vztahem

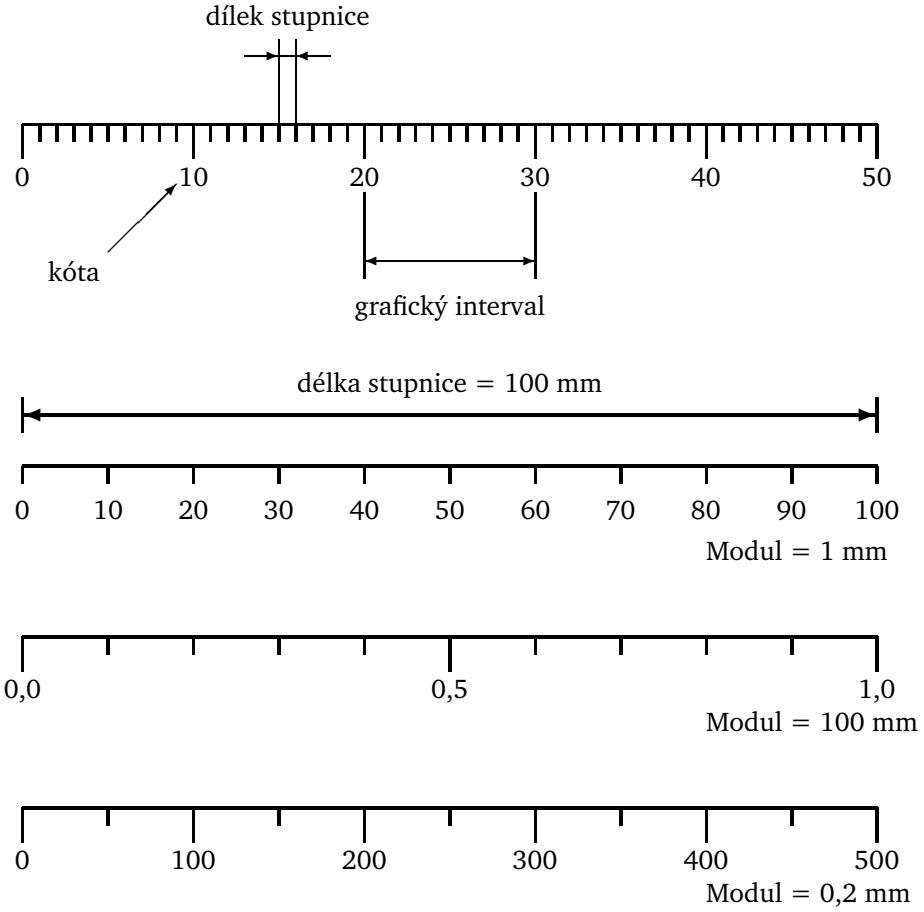
$$\epsilon = \frac{\eta}{M}.$$

Sestavování stupnice lze také provádět v závislosti na požadované přesnosti čtení. Spočívá ve stanovení přesnosti, z níž se vypočte modul a ten se dále vynásobí počtem hodnot zkoumaného souboru dat.

Modul na ose y se obvykle stanovuje tak, aby výsledné grafické pole mělo čtvercový tvar, případně tvar vhodného obdélníka.

Mezi nerovnoměrnými stupnicemi zaujímá zvláštní místo stupnice logaritmická. Používá se podle povahy vynášených dat. Její konstrukce je poněkud složitější než u stupnice rovnoměrné. Vynášejí se hodnoty odpovídající logaritmům čísel 1–10. Může mít více cyklů (sad hodnot 1–10), které jsou vůči sobě vynásobeny deseti.

Sdružené stupnice (dvojstupnice) se používá v případě, že je potřebné vynášet absolutní i relativní hodnoty současně. Absolutní hodnoty jsou vynášeny vlevo, relativní vpravo. Pro sdruženou stupnici lze použít jednu čáru, nebo dvě čáry vedle sebe.



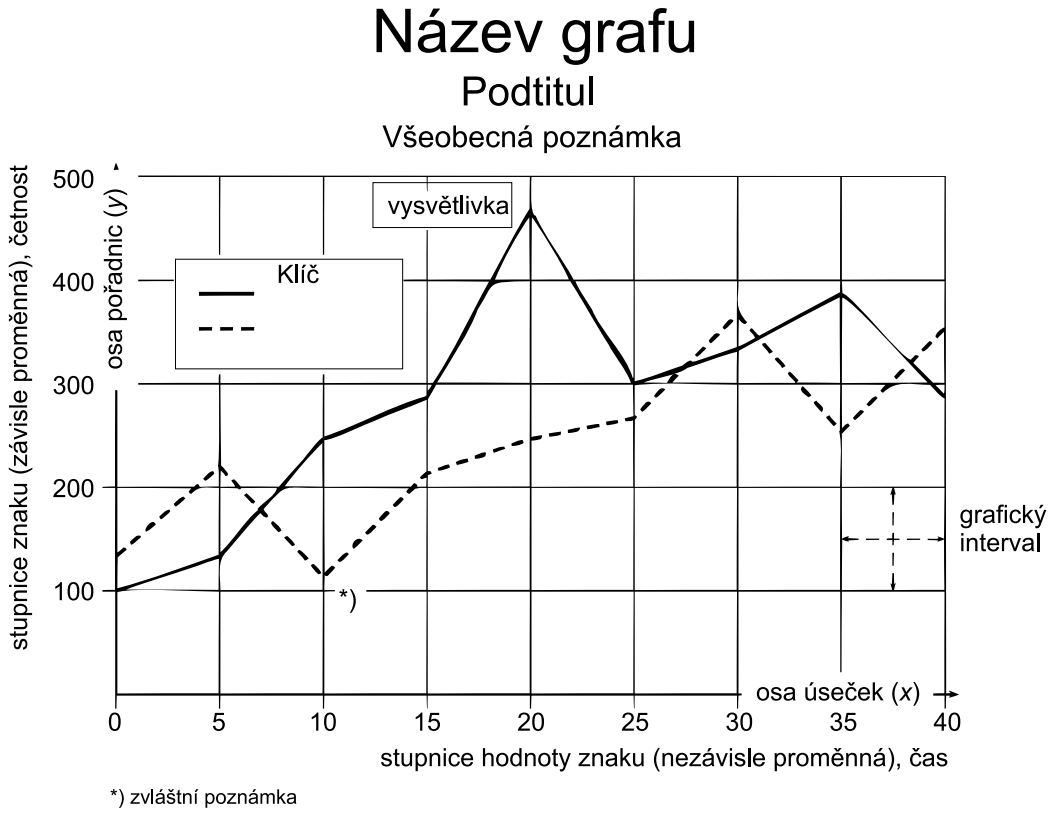
Obrázek 2. Stupnice, modul

Grafická síť se používá pro rychlejší určení přibližné hodnoty z grafu. Je to soustava rovnoběžek s osami souřadné soustavy. Rovnoběžky procházejí jednotlivými body stupnice, nejčastěji právě těmi, které jsou opatřeny kótami.

Grafická síť může být jednoduchá, jsou-li rovnoběžky vedeny pouze s jednou osou (jednoduchá grafická síť vodorovná, nebo svislá). Dvojitou grafickou síť tvoří rovnoběžky s oběma osami. Hustota sítě má vliv na přesnost odečtu hodnot. Pro rychlý přehled se používají řídké sítě s velkým grafickým intervalem.

Formát grafu, tj. velikost, na jakou bude kreslen, musí odpovídat složitosti grafu. Je nezbytné, aby byly složitější grafy kresleny na větší formát, a tím byly všechny podstatné prvky dobře viditelné. Dalším důležitým vodítkem pro stanovení formátu grafu je text, do něhož je graf vkládán – stupeň písma, šířka řádku, případně další typografické veličiny textu musí být sladěny s obdobnými prvky grafu. Pro snadné čtení by měl být graf umístěn vodorovně, nevejde-li na šířku sazby, pak otočený o 90 ° proti směru hodinových ručiček (nadpis grafu je pak vlevo). Stupnice se u širších grafů může opakovat vpravo, u obou stupnic nesmí chybět jednotky. Pro zpřesnění údajů lze vypisovat číselné hodnoty přímo u jednotlivých grafických bodů.

Orientační přehled grafických prvků grafu je uveden na obrázku 3.



Obrázek 3. Uspořádání grafických prostředků v grafu (podle [1])

3 Rozdělení grafů

Základní rozdělení na schémata a diagramy již bylo uvedeno. Diagramy lze dále rozdělit na prosté (porovnání malého počtu prvků nebo jejich skupin), na diagramy statistických řad, diagramy časových řad (chronodiagramy) a na prostorové diagramy (topodiagramy).

Podle formy grafického obrazu lze klasifikovat rozměrové grafy (kvantita jevu je znázorněna poměrovou velikostí grafického obrazu), souřadnicové grafy (kvantita je vyjádřena pomocí tvarů v souřadné soustavě se stupnicemi, trojrozměrné grafy (zvláštní případ předchozích dvou), statistické mapy (kromě prvků rozměrových a souřadnicových grafů mají ještě kvalitativní znázornění na mapě) a popularizační grafy (jednoduché a názorné typy, piktogramy apod.).

Volba grafu záleží na charakteru dat a účelu zobrazení. Hlavní typy grafů jsou:

- *Bodový graf* – používá se pro zjištění polohy bodu v souřadnicové soustavě. Body samotné nemají rozměr, hodnota je určena pozicí bodů. Nepravé bodové diagramy naznačují velikostmi bodů četnosti v určité ploše.
- *Spojnicový graf* – vzniká spojením bodů vynesených v ploše. Umožňují jednoduše sledovat více řad v jednom grafu, kde jsou řady rozlišeny ideografickými prvky (barva, tloušťka a provedení čáry). Spojnicový graf se používá velmi často, a to zejména pro vyjádření četností.
- *Plošný graf* – používá jako základní grafický vyjadřovací prostředek plochu čtverce, obdélníka, kruhu apod. Plocha indikuje velikost sledovaného jevu. Vyjádření plochou neumožňuje jednoduše srovnávat zkoumané jevy, ale s výhodou je plošný graf možné využít tam, kde je výsledná hodnota dána součinem dvou ukazatelů – ukazatele tvoří strany obdélníka, výsledkem je plocha obdélníka.
- *Sloupkový graf* – je zvláštním případem plošného grafu, kdy jeden rozměr plochy je konstantní (často šířka sloupku) a druhý rozměr se mění – vyjadřuje kvantitu sledovaného jevu. Sloupkový graf je nejčastěji užívaným prostředkem pro srovnávání.
- *Kruhový (koláčový) graf* – je druhým speciálním případem plošného grafu. Nejčastěji se využívá pro vyjádření struktury pomocí kruhových výsečí s úhlem reprezentujícím relativní četnosti.
- *Kartogram* – vyjadřuje územní rozložení zkoumaných jevů. Základním grafickým prvkem je mapa, rozdělená na regiony, v nichž jsou kvantitativní veličiny naznačeny různým šrafováním, barevnou výplní nebo odstíny.
- *Kartodiagram* – podobně jako kartogram znázorňuje územní rozložení zkoumaných jevů, avšak v jednotlivých regionech mapy jsou znázorněny jiné grafy vyjadřující příslušné hodnoty.
- *Piktogram* – slouží k popularizačním účelům. Je založen na opakování zvoleného ideografického prvků (například obrázku auta) tolikrát, kolik odpovídá četnosti základních jednotek. Pro zpřehlednění se symboly mohou sdružovat do skupin například po deseti. Graf je málo přesný, ale výrazný.

– *Stereogram (prostorový graf)* – umožňuje sledování tří sledovaných hodnot znaku v jednom grafu. Realizuje se axonometrickým nebo kosoúhlým zobrazením.
– *Pseudoprostorový graf* – znázorňují pouze dvě sledované hodnoty znaku, ale jsou konstruovány v kosoúhlém zobrazení tak, že jeden rozměr je pevně určen. Nejčastějším příkladem je použití sloupkového grafu, kde místo sloupku je kvádr, jehož jeden rozměr (hloubka) je konstantní.

4 Prostředky pro tvorbu grafů

Programové prostředky používané pro tvorbu grafů z dat v tabulkách jsou většinou realizovány jako interaktivní nástroje, kde si uživatel volí z nabídky požadované parametry a systém zobrazuje aktuální grafický tvar. Patří mezi nejrozšířenější aplikace, protože jsou obvyklou součástí kancelářských balíků (například Open Office, StarOffice, MS Office). Kromě toho existují nástroje integrované do specializovaných statistických balíků, umožňující vytvářet grafy různých statistických charakteristik (například Statgraphics).

Hlavní výhodou interaktivního nástroje je okamžitá vizuální kontrola dosaženého výsledku. Většinou je k dispozici široká paleta různých typů grafů, možností ovlivňování tvaru všech komponent a okamžité překreslování výsledku při změně dat.

Zásadní nevýhoda interaktivního přístupu se ovšem projeví v okamžiku, kdy je nutné vytvořit sadu grafů s podobnými vlastnostmi. Nemá-li program možnost uložit nastavení grafu a opakovaně je použít, znamená to vždy opakovanou ruční práci u každého grafu zvlášť, přičemž není zaručeno, že výsledek bude skutečně jednotný. Z typografického hlediska je však nutné, aby významově stejné prvky měly v dokumentu vždy identickou podobu.

Uvedené produkty však ani nemají příliš dobré vlastnosti týkající se dalšího zpracování vykreslených grafů. Například MS Excel nemá vůbec žádnou možnost exportovat graf do některého použitelného grafického formátu, výsledek lze nejvýše vložit jako objekt do jiného programu, ovšem s reálným rizikem, že podoba na otisku nebude zdaleka odpovídat tvaru dosaženému interaktivní volbou příslušných nastavení. Je pochopitelné, že od bezkoncepčního produktu s množstvím principiálních nedostatků a hrubých programátorských chyb nelze čekat nic lepšího, nicméně ostatní produkty nejsou diametrálně odlišné.

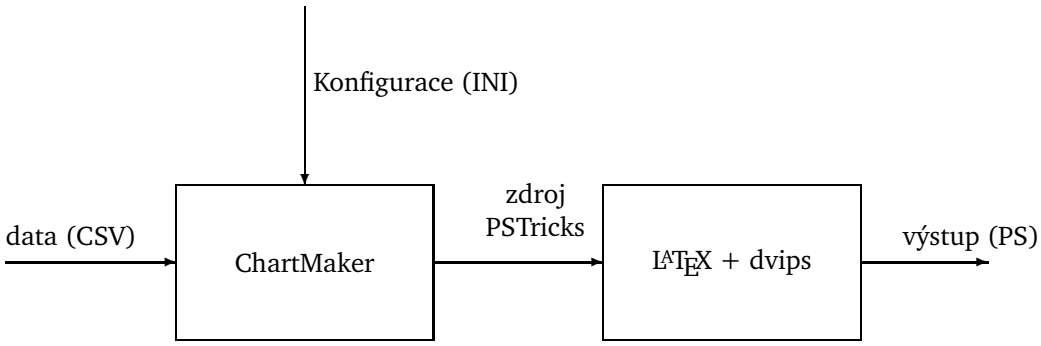
5 Automatizovaná tvorba grafů

Problém vytvoření grafu respektujícího základní zásady typografie a správného zobrazování dat je někdy nutné z uvedených důvodů řešit jinými prostředky. Jednou z možností je generování výsledného tvaru grafu z textově vyjádřených dat podle parametrů v konfiguračním souboru. Tuto možnost realizoval ve své diplomové práci Fojtlík (2002). Vytvořil program ChartMaker, jehož výstupem je zdrojový text s příkazy van Zandtova balíku PSTricks.

Tímto postupem se zároveň řeší problém integrace grafů do zdrojového textu TEXu (LATEXu).

5.1 Princip činnosti

Program zpracovává dva soubory. Hlavním je datový soubor, který zároveň obsahuje individuální nastavení pro daný graf, druhým zpracovávaným souborem je konfigurace, která platí pro celou sadu grafů. Schematicky lze činnost programu a začlenění grafu znázornit obrázkem 4.

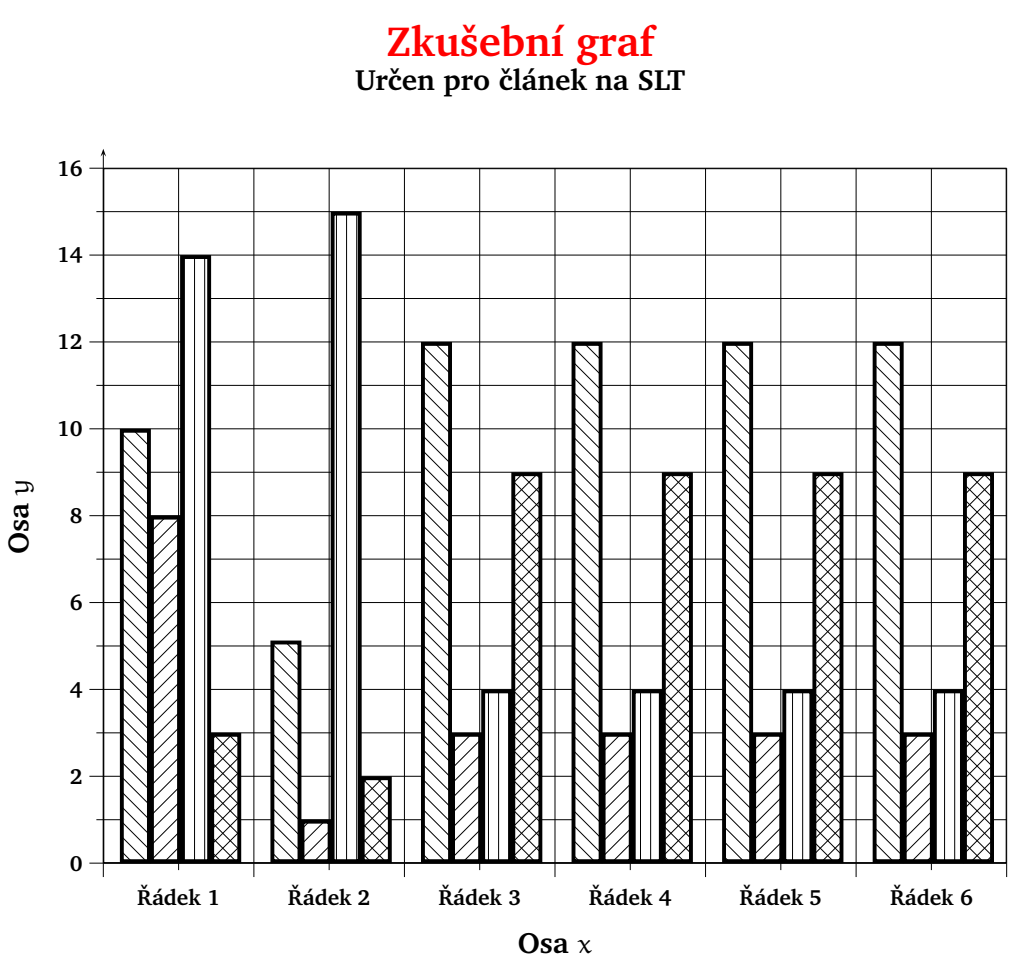


Obrázek 4. Schematické znázornění činnosti programu ChartMaker

5.2 Formáty zdrojových souborů

Vstupní data jsou zapsána ve formátu CSV, na jednotlivých řádcích jsou údaje oddělené středníky, textové položky jsou opatřeny uvozovkami. Tato data jsou předcházena ještě možnými nastaveními, jak ukazuje následující příklad:

```
Title=Zkušební graf
Subtitle=Určen pro článek na SLT
Note=(nejmenší hodnota je 0,5)
XAxisName=Osa $x$
YAxisName=Osa $y$
ShowXAxisLabels=true
ShowYAxisLabels=1
ShowXAxisPrimaryMarks=1
ShowYAxisPrimaryMarks=1
ShowXAxisSecondaryMarks=1
ShowYAxisSecondaryMarks=1
ShowXPrimaryGrid=1
ShowYPrimaryGrid=1
ShowXSecondaryGrid=1
ShowYSecondaryGrid=1
DataInRows=0
FirstRowLabels=1
FirstColumnLabels=1
Data
;"Sloupec 1";"Sloupec 2";"Sloupec 3";"Sloupec 4"
"Řádek 1";10;8;14;3
"Řádek 2";5.125;1;15;2
"Řádek 3";12;3;4;9
"Řádek 4";12;3;4;9
"Řádek 5";12;3;4;9
"Řádek 6";12;3;4;9
```



(nejmenší hodnota je 0,5)

Obrázek 5. Příklad vykresleného grafu

Inicializační soubor má formát INI souboru, tj. nastavení mnoha parametrů soustředěných do tématických sekcí. V každé sekci jsou nastavovány parametry různých typů, není-li určitý parametr uveden, uplatní se jeho implicitní hodnoty. Příklad inicializačního souboru:

```
[Main]
Chart.Type=column
Chart.SizeX=130
Chart.SizeY=100
TitleStyle.Style=\bfseries\LARGE
TitleStyle.Color=red
SubtitleStyle.Style=\bfseries\large
SubtitleStyle.Color=black
NoteStyle.Style=\small
NoteStyle.Color=black
ColumnOverlap=90
ColumnSpacing=100
Background.Style.Type=solid
Background.Style.Transparent=false
Background.Style.HatchSep=1.404
Background.Style.HatchWidth=0.1404
Background.Style.HatchAngle=45
Background.Style.HatchColor=gray
Background.Style.FillColor=white
Background.Border.Style.Type=solid
Background.Border.Style.
                DashLineLength=1.755
Background.Border.Style.
                DashSpaceLength=1.053
Background.Border.Style.DotSpace=1.053
Background.Border.Width=0.1404
Background.Border.Color=black

[Axis]
XAxis.Style.Type=solid
```

```
XAxis.Style.DashLineLength=1.755
XAxis.Style.DashSpaceLength=1.053
XAxis.Style.DotSpace=1.053
XAxis.Width=0.1404
XAxis.Color=black
XAxis.Arrows=none
XAxisNameStyle.Style=\bfseries
XAxisNameStyle.Color=black
XAxisNameStyle.Angle=0
XAxisLabelStyle.Style=\small
XAxisLabelStyle.Color=Black
XAxisLabelStyle.Angle=0
XAxisLabelSpace=3
YAxisLabelSpace=3
XAxisNameSpace=10
YAxisNameSpace=10
YAxis.Style.Type=solid
YAxis.Style.DashLineLength=1.755
YAxis.Style.DashSpaceLength=1.053
YAxis.Style.DotSpace=1.053
YAxis.Width=0.1404
YAxis.Color=black
YAxis.Arrows=lrarrow
YAxisNameStyle.Style=\bfseries
YAxisNameStyle.Color=black
YAxisNameStyle.Angle=0
YAxisLabelStyle.Style=\small
YAxisLabelStyle.Color=black
YAxisLabelStyle.Angle=0
PrimaryMarkLength=2
SecondaryMarkLength=1


[Grid]
PrimaryGrid.Style.Type=solid
PrimaryGrid.Style.DashLineLength=1.755
PrimaryGrid.Style.DashSpaceLength=1.053
PrimaryGrid.Style.DotSpace=1.053
PrimaryGrid.Width=0.1404
PrimaryGrid.Color=black
PrimaryGrid.Arrows=none
SecondaryGrid.Style.Type=solid
SecondaryGrid.Style.DashLineLength=1.755
SecondaryGrid.Style.DashSpaceLength=1.053
SecondaryGrid.Style.DotSpace=1.053
SecondaryGrid.Width=0.1404
SecondaryGrid.Color=black
SecondaryGrid.Arrows=none

[Lines]
Lines.Count=2
Line1.Style.Type=solid
Line1.Style.DashLineLength=1.755
Line1.Style.DashSpaceLength=1.053
Line1.Style.DotSpace=1.053
Line1.Width=0.5616
Line1.Color=black
Line1.Arrows=none
Line2.Style.Type=dashed
Line2.Style.DashLineLength=1.755
Line2.Style.DashSpaceLength=1.053
Line2.Style.DotSpace=1.053
Line2.Width=0.5616
Line2.Color=black
Line2.Arrows=none

[Areas]
Areas.Count=4
Area1.Style.Type=vlines
Area1.Style.Transparent=false
Area1.Style.HatchSep=1.404
Area1.Style.HatchWidth=0.1404
Area1.Style.HatchAngle=45
Area1.Style.HatchColor=black
Area1.Style.FillColor=white
Area1.Border.Style.Type=solid
Area1.Border.Style.DashLineLength=1.755
Area1.Border.Style.DashSpaceLength=1.053
Area1.Border.Style.DotSpace=1.053
Area1.Border.Width=0.5616
Area1.Border.Color=black
Area2.Style.Type=hlines
Area2.Style.Transparent=false
Area2.Style.HatchSep=1.404
Area2.Style.HatchWidth=0.1404
Area2.Style.HatchAngle=45
Area2.Style.HatchColor=black
Area2.Style.FillColor=white
Area2.Border.Style.Type=solid
Area2.Border.Style.DashLineLength=1.755
Area2.Border.Style.DashSpaceLength=1.053
Area2.Border.Style.DotSpace=1.053
Area2.Border.Width=0.5616
Area2.Border.Color=black
Area3.Style.Type=hlines
Area3.Style.Transparent=false
Area3.Style.HatchSep=1.404
Area3.Style.HatchWidth=0.0404
Area3.Style.HatchAngle=90
Area3.Style.HatchColor=black
Area3.Style.FillColor=white
Area3.Border.Style.Type=solid
Area3.Border.Style.DashLineLength=1.755
Area3.Border.Style.DashSpaceLength=1.053
Area3.Border.Style.DotSpace=1.053
Area3.Border.Width=0.5616
Area3.Border.Color=black
Area4.Style.Type=crosshatch
Area4.Style.Transparent=false
Area4.Style.HatchSep=1.404
Area4.Style.HatchWidth=0.1404
Area4.Style.HatchAngle=45
Area4.Style.HatchColor=black
Area4.Style.FillColor=white
Area4.Border.Style.Type=solid
Area4.Border.Style.DashLineLength=1.755
Area4.Border.Style.DashSpaceLength=1.053
Area4.Border.Style.DotSpace=1.053
Area4.Border.Width=0.5616
Area4.Border.Color=black

[Dots]
Dots.Count=2
Dot1.Style.Type=fullcircle
Dot1.Style.ScaleX=1
Dot1.Style.ScaleY=1
Dot1.Style.Angle=0
Dot1.Color=black
Dot2.Style.Type=fulltriangle
Dot2.Style.ScaleX=1
Dot2.Style.ScaleY=1
Dot2.Style.Angle=0
Dot2.Color=black
```

Detailní popis všech parametrů lze nalézt v diplomové práci ing. Fojtlíka.

Z uvedených příkladů datového a inicializačního souboru byl vytvořen graf na obrázku 5. Pro vložení grafu do zdrojového souboru je nutné připojit styly `multido` a `pstricks`, příkladem minimálního zdrojového souboru může být tento text:

```
\documentclass{article}
\usepackage{czech,multido,pstricks}
\begin{document}
\input graf.pst
\end{document}
```

6 Závěr

Systém automatizované tvorby grafů prezentuje určitou myšlenku, která má za cíl usnadnit tvorbu správně vytvořených grafických reprezentací dat vyjádřených většinou tabelárně. Konkrétní aplikace nevyčerpává všechny potenciální možnosti, obsahuje jisté dětské nemoci, ale ukazuje cestu, kterou je možné dále rozšířit a zkvalitnit.

Reference

1. Fojtlík, V. *Automatizovaná grafická prezentace dat*. Diplomová práce. Brno: MZLU, 2002.
2. Klimeš, L. *Slovník cizích slov*. Praha: SPN, 1987.
3. Kubíček, J. *Statistika (vyjadřovací prostředky a formy ve statistice)*. Brno: VŠZ, 1988.
4. van Zandt, T. *PSTricks. User's Guide.* [Elektronický soubor]. 1993.

TEX a PDF

Vít Zýka

České vysoké učení technické, Fakulta elektrotechnická
Centrum aplikované kybernetiky
Email: zyka@cmp.felk.cvut.cz

Abstrakt: Program TEX je systém pro velmi kvalitní počítačovou sazbu vyznačující se přenositelností zdrojového kódu mezi platformami i v čase. Portable Document Format (PDF) je formát pro elektronické publikování. Tento příspěvek ukazuje možnosti elektronického publikování v PDF uživateli zvyklými na TEX.

1 Troška historie

Není mým cílem zde rozebírat historii programů TEX, `dvips` a pdfTEX stejně jako historii formátů PS a PDF. Kdo by chtěl nahlédnout do podmíněností vzniku těchto softwarových nástrojů počítačové sazby, tomu doporučuji skvělý článek Philipa Taylora [17].

2 Co přináší PDF oproti PS?

Koncem osmdesátých let byl PS prakticky standardem pro popis vzhledu stránek. Přesto firma Adobe začala od roku 1990 pracovat na specifikaci standardu jiného. V té době již bylo zřejmé, že komunikace a výměna dokumentů prostřednictvím sítí je mnohem flexibilnější než pracovat s dokumenty v tištěné podobě. Chyběl však formát, který by v sobě spojoval typografickou kvalitu PS a hypertextové a multimediální možnosti elektronického publikování. Tyto okolnosti vedly ke vzniku PDF. Pokusme se charakterizovat jeho vlastnosti [16,19].

1. Začneme tím, v čem je rozdílnost minimální. Je to *kreslící model*, který se v PS osvědčil, umožňující komplexní grafický popis stránky. Oba standardy používají stejnou afinní transformaci pro převod mezi uživatelským (na výstupním zařízení nezávislým) souřadným systémem a systémem výstupního zařízení. Používat lze bitmapovou i vektorovou grafiku. Základním kamenem pro vektorový popis textu i grafiky jsou kubické Bèzierovy křivky. Volit lze z různých druhů tahů, způsobů napojení a rastrování rohů. Uzavřené křivky lze vyplňovat vzory. Používat lze různé barevné modely (RGB, CMYK, CIE). Verze PDF 1.4 umí pracovat s průhledností.
2. Velké rozdíly najdeme v *jazyku* dokumentu. Zatímco PS je plnohodnotný programovací jazyk zásobníkového typu, PDF neobsahuje procedury, řídící struktury a proměnné. Jde o seznam grafických operací. V PS se operátory

Jan Kasprzak, Petr Sojka (editoři): SLT 2002 – sborník semináře o Linuxu a TEXu, str. 69–77, 2002.

vyšší úrovně vyjadřují přímo jazykem PS z elementárních grafických příkazů. V PDF musí být tyto vyšší operace přímo implementovány aplikací zpracovávající dokument. Důsledkem je efektivnější zpracování a zobrazování PDF dokumentu. Na druhou stranu je snížena flexibilita vyjádření. Například balík PSTRICKS [23] expanduje kreslící příkazy do PS kódu. Tento kód je relativně složitý a obsahuje programové konstrukce, např. cykly. Předpokládá se, že nakonec bude dokument zpracován nějakým PS RIPem. Protože např. pdfTEX takový RIP neobsahuje, nelze tento balík v pdfTEXu použít. Řešením je vložit takový obrázek do zvláštního dokumentu, ten zpracovat standardním TEXem, pomocí `dvips -E` vyrobit encapsulated PS, ten převést skriptem `epstopdf` využívající GhostScript do PDF a načíst do našeho dokumentu. O zautomatizování tohoto procesu se snaží balík PDFTRICKS [13]. Spouští při tom externí programy přímo z TEXu pomocí konstrukce `\write18{command}`. Tuto možnost nabízí např. web2c instalace TEXu.

3. PS byl původně textový kód bez jakékoliv pevné struktury. Dodatečně vydaná technická zpráva [8] doporučuje strukturní konvence PS (Document Structuring Conventions, DSC) dokumentů a značkování pomocí komentářů. Pak lze oddělit jednotlivé stránky dokumentu bez nutnosti interpretovat PS. Na přítomnosti těchto komentářů je založen balík PSUTILS [3]. Náhodný přístup k libovolné části dokumentu řeší až pevná struktura PDF. Dokument PDF se skládá z objektů, jejichž umístění v dokumentu popisuje tabulka křížových referencí. Každá stránka je také samostatným objektem a krom sazby textu obsahuje i seznam svých zdrojů (fonty, anotace), informace o velikosti stránky ap. Díky tomu stačí pro zobrazení stránky interpretovat jen ty objekty, které tvoří tuto stránku. Také je snadné s dokumentem manipulovat, měnit pořadí stránek a jejich velikost. Také můžeme snadno vložit stránku jako obrázek do jiného dokumentu (toho využívá balík pdfpages [10]). Další užitečnou vlastností PDF je možnost jeho jednoprůchodového generování. Je to zajištěno systémem nepřímých odkazů na objekty. Například potřebujeme-li zadat uvnitř objektu A jeho velikost, kterou před jeho uzavřením jestě neznáme, uvedeme v něm pouze nepřímý odkaz na objekt B. Objekt B bude obsahovat velikost objektu A.

4. PDF podporuje standardní kompresní formáty (JPEG, CCITT Group 3 a 4, LZW a Run Length). Jejich dekompresi musí zajistit aplikace zpracovávající PDF. Kromě výrazného zmenšení dokumentu (důležité pro přenos po síti) a nezmenšeném konfortu pro uživatele (žádné odzipovávání) je díky této podpoře snadné i vkládání komprimovaných bitmapových obrázků (TIFF, PNG, JPEG). Nevýhodou ovšem je, že nelze snadno na úrovni TEXových maker vyhledávat a měnit řetězce textu tak, jak to například dělá balík PSFRAG [4] pro editaci popisek obrázků často generovaných programy s méně variabilním grafickým výstupem (fonty, matematická sazba). I zde by se pro pdfTEX hodila makra podobná PDFTRICKS, pro automatické externí zpracování obrázku přes PS.

5. Z běžných formátů fontů umožňuje PS pracovat hlavně s Type 1 fonty. Bitmapové fonty rastrované METAFONTem jsou vkládány ve formátu Type 3. PDF zvládá oba tyto formáty a navíc umí pracovat s True Type formátem.
6. PDF podporuje předtiskovou úpravu dokumentu (hranice media, stránky a zobrazení, tisk ořezových značek, barevné separace, ...).
7. Hlavním přínosem PDF bylo jeho interaktivní rozšíření využitelné při elektronickém zobrazení dokumentu. Jde o hypertextové prvky usnadňující orientaci (odskoky, záložky, náhledy), dále zpřístupnění videa a zvuku a možnost vyplňování a zasílání formulářů.
8. V PDF existuje podpora pro indexaci a vyhledávání slov.
9. Dokument je možné také modifikovat a to pouze připojením změněných nebo nových objektů na konec dokumentu spolu s novou tabulkou křížových referencí. Původní část dokumentu tedy zůstává nedotčena. Snadno se pak můžeme vrátit k předchozí verzi.
10. Formát byl navržen tak, aby byl systémově nezávislý, šlo jej rozšířit o nové technologie se zachováním zpětné kompatibility.

Po přečtení tohoto výčtu můžeme nabýt dojmu, že PDF má vše, co potřebujeme pro potřeby tisku, elektronického publikování nebo prezentací. Specifikace formátu je však věc jedna a jeho implementace a dostupnost vhodných nástrojů věc druhá. Sama Adobe se snažila prosadit PDF tím, že zdarma poskytla nástroj Adobe Acrobat Reader pro prohlížení a tištění PDF dokumentů na různých platformách. Díky tomu se jí také *de facto* podařilo prosadit PDF jako standard elektronického dokumentu. Přesto existují problémy, které používání formátu ztěžují. Napadají mě tyto:

1. Volně šířený Acrobat Reader ani plný Acrobat umožňující modifikovat PDF dokument neimplementuje celou specifikaci formátu (uveďme např. nastavení hlasitosti v `/Sound` anotaci, grafické znázornění `/Sound` anotace podle jejího stavu, specifikace `/Sound` anotace podle URL, vložení video souboru pro `/Movie` anotaci interně do dokumentu). Kromě toho existují části specifikace systémově závislé (spouštění externích programů, zvukové a video anotace).
2. Implementace Acrobatu obsahuje odlišnosti od specifikace (=chyby; např. obsahuje-li encoding vektor fontu vynechané nedefinované znaky, tiskne Acrobat v5 chybně nebo aplikování transformační matice je podle specifikace možné několika způsoby, Acrobat však rozumí jen některým). Že software obsahuje chyby je vcelku pochopitelné, co je však pobuřující, je postoj Adobe k jejich odstraňování (a bohužel typický pro velkou komerční firmu). Znalým uživatelem pohrdají, chybu většinou nepřiznají, a pokud ano, její odstranění trvá léta.
3. Podle specifikace musí mít každý interpret PDF k dispozici 14 základních fontů. Je to proto, aby malé dokumenty zbytečně nenarůstaly vkládáním fontů. Co si však myslet o tom, že Adobe přibaluje k různým verzím Acrobatu tyto fonty s různými metrikami?
4. Adobe preferuje v některých oblastech podivné strategie vedoucí k mizerné kvalitě dokumentu. Příkladem je záměrně špatná rasterizace Type 3 fontů na obrazovku Acrobatem.

5. Ačkoliv je Acrobat Reader distribuován pro různé platformy, na některé je k dispozici s výrazným časovým skluzem (Linux) a pro jiné se vývoj zastavil (OS2).

Přenositelnost dokumentu není z těchto důvodů naplněna. Je pravda, že pokud se omezíme na jednoduchý hypertext bez videa a externích volání, vložíme všechny fonty do dokumentu a přizpůsobíme svůj program tak, aby negeneroval chybu v prohlížeči, i když je syntakticky správně, slušné záruky o přenositelnosti dokumentu dosáhneme. Problém nevidím ve specifikaci PDF, ale v implementaci této specifikace. Konkurence z oblasti open-source (GhostScript, xpdf) udělala velký kus práce, ale i tak je v rozsahu pokrytí specifikace za Adobe. Chybí hlavně podpora prezentačního módu, videa a java skriptu. Nechť je to výzva pro open-source programátory.

3 Tři cesty od TEXu k PDF

Existují v podstatě tři cesty, jak vytvořit PDF dokument z TEXového zdrojového kódu:

1. Generovat jej přímo ze zdrojového kódu. Umí to modifikace TEXu zvaná pdfTEX. Program vznikl na Masarykově univerzitě jako magisterská a doktorská práce Hàn Thế Thành [20,21].
2. Standardním TEXem vytvořit DVI a zkonvertovat jej do PDF například programem `dvipdf` [22].
3. Vytvořit z DVI PS a ten pak převést do PDF pomocí GhostScriptového skriptu `ps2pdf` nebo komerčním Adobe Distillerem.

Vkládání interaktivních prvků je v pdfTEXu umožněno zavedením nových primitiv. Zbylé dva způsoby využívají značkování TEXovým primitivem `\special{...}` a PS operátorem `/PDFMark`.

S programem `dvipdf` nemám osobní zkušenost, takže jej hodnotit nebudu. Ale je to nejméně používaná cesta. Kvalita převodu pdfTEXem a přes PS je výborná. Distiller měl oproti GhostScriptu navrch, ale v poslední době se tento rozdíl stírá. Vyplatí se mít GhostScript čerstvější verze. Distiller umožňuje optimalizovat dokument z hlediska jeho použití (osvit, tiskárna, obrazovka). Pokud by chtěl tyto zařízení rozlišit uživatel pdfTEXu, musel by ručně přizpůsobit každý obrázek (rozlišení, barevný model), probrat specifikaci PDF a nastavit patřičné parametry.

Při generování PDF je dobré se vyhnout Type 3 fontům, tj. fontům generovaných METAPOSTem. Dnes již většina METAPOST fontů existuje v dobré kvalitě i jako Type 1. Pro první a třetí způsob generování PDF se nastavení PS fontů zajistí na úrovni konfiguračních souborů (ve web2c instalaci je to implicitní nastavení `pdftex.map` uvedený v `pdftex.cfg`, respektive `config.pdf` pro DVI ovladač `dvips` spuštěný s parametrem `-Ppdf`).

Výhody pdfTEXu oproti výrobě PDF přes DVI jsou následující:

– Můžeme vkládat bitmapovou grafiku ve formátech JPEG, PNG a TIFF. Při klasickém překladu do PS jsme většinou omezeni převodem obrázků do EPS, čímž jejich velikost velmi naroste (běžně 10×) a naroste tak i velikost výsledného PS. V případě jednostránkových plakátů tak běžně jde o PS soubor veliký stovky Mb. Také musíme započítat paměťovou a časovou režii při převodu obrázků.
– Můžeme používat True Type fonty.
– Můžeme využívat mikrotypografické rozšíření, kterými se zabývá Thanhova disertační práce [21]. Jde o vysící znaky z bloku textu a horizontální zvětšování/zmenšování znaků u řádků s vysokou hodnotou badness. Jejich cílem je kompaktnější a homogennější globální vzhled odstavce při okem nepostřehnutelných lokálních výchylkách. Obě techniky jsou implementovány do TEXového optimalizačního algoritmu zlomu odstavce. Více viz v sekci 4.1 v bodě 4.

V další sekci se zaměřím na přímé generování PDF pdfTEXem.

4 Program pdfTEX

Program pdfTEX má svůj vlastní diskuzní list vedený v anglickém jazyce. Jeho adresa je `pdftex@tug.org` a archiv konference lze nalézt na `http://tug.org/pipermail/pdftex/`. Přihlásit se lze na stránkách světového Sdružení uživatelů TEXu `http://tug.org/mailman/listinfo/pdftex`.

4.1 Nová primitiva

Jak jsme již uvedli, pdfTEX umožňuje využívat možností PDF pomocí nových primitiv. Jejich popis (bohužel neúplný) uvádí pdfTEX manual [1], úplný seznam čtenář najde v [18]. My si je zde funkčně roztřídíme bez požadavku na úplnost:

1. Hlavní přepínač `\pdfoutput` jehož kladná hodnota přepíná výstup z DVI do PDF. Jeho nastavení má význam jen před výstupem první stránky pomocí `\shipout`. Tento registr se testuje, chceme-li automaticky rozlišit, zda je dokument zpracováván klasickým TEXem nebo pdfTEXem s výstupem do DVI či pdfTEXem s výstupem do PDF:

```
\newif\ifPDF
\ifx\pdfoutput\undefined
\else\ifnum\pdfoutput>0 \PDFtrue\fi
\fi
```

2. Rozměrové parametry (např. `\pdfpagewidth`, `\pdfhorigin`). Pozor! Pokud nastavíte `\pdfhorigin` nebo `\pdfvorigin` na nulu, pdfTEX ji změní na 1 in. Proto v takovém případě použijte např. `\pdfhorigin=1sp`).

3. Parametry dokumentu (komprese `\pdfcompresslevel`, numerická přesnost `\pdfdecimaldigits`, informační údaje `\pdfinfo`, způsob zobrazování dokumentu [16, str. 83] `\pdfcatalog`) a jednotlivých stránek [16, str. 88] (`\pdfpagesattr`, `\pdfpageattr`).
4. Mikrotypografická rozšíření.
   – Visící znaky (protrude characters). Každému znaku lze přiřadit hodnotu, o kolik má tento znak přesahovat levý (`\lpcode`) a pravý (`\rpcode`) okraj bloku textu. Syntaxe obou primitiv je stejná jako u primitiva `\catcode`. Hodnota 1000 znamená převis o 1 em. Visící znaky zapínáme kladnou hodnotou `\pdfprotrudechars`. Je-li jinak menší jak 2, pak se řádky nalámou standardním algoritmem TEXu a následně se doplní převis. Je-li 2 a více, optimalizuje se zlom i s aktuálními hodnotami visících zanků. Rozhoduje hodnota tohoto registru při uzavření odstavce. Příklad: chceme-li vpravo přesah tečky o 5 % em a rozdělovacího znaménka o 8 % em v aktuálním fontu, napíšeme:

     ```
     \pdfprotrudechars=2
     \rpcode\font‘\.=50
     \rpcode\font\hyphenchar\font=80
     ```

     Mezi visící znaky pečliví typografové zařazovali interpunkční znaménka (tečky, čárky, uvozovky, rozdělovací znaménka) pro jejich přílišnou světlost narušující vertikální hranu textu.
   – Horizontální zvětšování/zmenšování znaků (font expansion). Někdy je obtížné zlomit odstavec tak, aby neobsahoval velké díry mezi slovy. Často se to stává při sazbě do úzkého sloupce. Horizontální zvětšování/zmenšování znaků nám přidává další stupeň volnosti pro rovnoměrné vyplnění řádky. Každému fontu přidělíme maximální hodnoty roztažení/ztažení pomocí primitivu `\pdffontexpand`. Také se zde nastavuje krok, protože stažení i roztažení se děje diskrétně. Změna velikosti znaku musí však být prováděna s citem, aby čtenář nic nepoznal, i když se pod sebou sejde nejvíce protažená a smrsknutá řádka. Doporučené hodnoty jsou okolo 2 %. I zde máme možnost ovlivnit, o kolik se může jedno písmeno roztáhnout/stáhnout vůči globálním hodnotám pomocí hodnoty `\efcode`. Přepínač se jmenuje `\pdfadjustspacing` a i zde má tři polohy stejně jako u visících znaků. Ke každému takto použitému fontu musíme připravit metriky a jde-li o bitmapové (METAPOST) fonty, musíme zajistit i vygenerování těchto bitmap. Syntaxe názvu fontů je následující: `fontname-shrink.tfm` nebo `fontname+stretch.tfm`. Příklad:

     ```
     \newcount\N
     \loop\efcode\font\N=1000\advance\N by 1
       \ifnum\N<256 \repeat
     \pdfadjustspacing=2
     \pdffontexpand\font 20 20 5 1000
     ```

     Číselné hodnoty u `\pdffontexpand` mají následující význam: roztažení, stažení, krok a měřítko.

5. Primitiva pracující s fonty (`\pdffontname`, `\pdffontobjnum`, `\pdfincludechars`).
6. Parametry bodu sazby (`\pdfsavepos`, `\pdflastxpos`, `\pdflastypos`).
7. Primitiva vkládající obrázky [24] či boxy sazby (`\pdfximage`, `\pdfrefximage`, `\pdflastximage`, `\pdfimageresolution`, `\pdfxform`, `\pdfrefxform`, `\pdflastxform`). Málo známý je registr `\pdflastximagepages`, který udává počet stran PDF dokumentu načteného pomocí `\pdfximage`.
8. Primitiva vytvářející hypertextové odkazy [25] (odkazy `\pdfstartlink`, `\pdfendlink`, `\pdflinkmargin`, doskoky `\pdfdest`, `\pdfdestmargin`, záložky `\pdfoutline` a zřetězení článku `\pdfthread`, `\pdfstartthread`, `\pdfendthread`).
9. Primitiva zařazující anotace, např. zvukové soubory a videa [26] (`\pdfannot` a `\pdflastannot`).
10. Primitiva vkládající obecné objekty včetně proudů [26] (`\pdfobj`, `\pdfrefobj`, `\pdflastobj`,...) a obecné grafické instrukce (`\pdfliteral`).
11. Parametry programu pdfTEX (`\pdftexversion`, `\pdftexrevision`).

4.2 Makrobalíky

Velice šikovným nástrojem je balík `thumbpdf.sty` od Heiko Oberdieka, který Perlovým skriptem ve spolupráci s GhostScriptem vytvoří a vloží do dokumentu *náhledy stránek*. Je napsaný v plain TEXu. Použití je nesmírně jednoduché:

1. Vlož makra do dokumentu: `\input thumbpdf.sty`
2. PřeTEXuj.
3. Spust skript Perlu: `thumbpdf file_without_ext`
4. PřeTEXuj.

Uživatelé LATEXu mohou využít balíku Sebastiana Rahtze `hyperref` [14,11]. Nejenomže zjednodušuje vkládání *hypertextových odkazů* a automaticky generuje linky ze standardních LATEXových referenčních mechanismů. Umožňuje dokonce učinit kód nezávislým na způsobech generování PDF popsaných v sekci 3.

Myslím, že oblast, elektronického publikování, jehož rozvoj teprve nastane, je vyplňování *formulářů*. I zde existuje velmi silný nástroj pro LATEX. Jmenuje se AcroTEX a jeho autorem je D. P. Story [15]. Tento balík obsahuje i podporu Java skriptu.

A snad nejsilnější podporu pro elektronické publikování a prezentace má balík ConTEXt Hanse Hagena. Je to nástroj využívající to nejlepší ze tří různých programů: sazbu pdfeTEXu, vektorovou grafiku METAPOSTu a výpočty, textové manipulace a řízení procesu Perlu. Od letošního roku je konečně k dispozici výborná dokumentace v angličtině [6,7].

5 Prezentace pdfTEXem

V této části budou ukázány možnosti využití PDF při prezentacích včetně odkazů na nejpoužívanější prezentační makrobalíky. O ConTEXtu jsem se již zmínil.

Originální varianty Hagenových prezentací přináší dokument [5]. Seznam prezentačních balíků pro LaTeX uvádí prezentace [2]. Velmi oblíbený je pdfScreen od C. Radhakrishnana [12]. Umí paralelně vytvářet prezentace a textový dokument. Inkrementální přírůstky textu na stránce zvládá balík TeXPower od Stephana Lehmke [9].

6 Závěr

Formát PDF je standardem pro elektronické publikování a je vhodný i pro data-prezentace. Uživatelé připravující dokumenty TeXem si mohou vybrat několik cest, jak připravit velmi kvalitní PDF. Usnadní jim v tom dnes již silná podpora makrobalíků. Po prudkém rozvoji v této oblasti koncem devadesátých let se situace stabilizuje. Týká se to i programu pdfTeX, který přináší několik technologických vylepšení. Největší problémy dnes nenastávají na straně vytváření PDF, ale v jeho zobrazování a tisku. Neexistuje přenositelný nástroj na zpracování méně obvyklých prvků specifikace PDF.

Reference

1. Hàn Thế Thành, Sebastian Rahtz, a Hans Hagen. *The pdfTeX user manual*, 2001. `http://www.tug.org/applications/pdftex/pdftex-s.pdf`.
2. David M. Allen. LaTeX presentation packages, únor 2002. `http://www.ms.uky.edu/~allen/presentations.pdf`.
3. Angus Duggan. Psutils. Free program package.
4. Michael C. Grant a David Charlisle. *The PSfrag System, version 3*, 1998.
5. Hans Hagen. The ConTeXt presentation styles, 2001. `http://www.tug.org/tug2001/authors/presentations/hagen/hagenIIIslides.pdf`.
6. Hans Hagen. *ConTeXt the manual*. `http://www.pragma-ade.com/`, listopad 2001.
7. Hans Hagen. *METAFUN*. `http://www.pragma-ade.com/`, leden 2002.
8. Adobe Systems Incorporated. PostScript language document structuring conventions specification. Technická zpráva 5001, září 1992.
9. Stephan Lehmke. TeXPower package. Home page: `http://texpower.sourceforge.net`.
10. Andreas Matthias. *The pdfpages Package*, 2002. Email: `amat@kabsi.at`.
11. Heiko Oberdiek. *PDF information and navigation elements with hyperref, pdfTeX, and thumbpdf*, 1999. `TeXLive6: $TEXMF/doc/latex/hyperref/slides.pdf`.
12. C. V. Radhakridhnan. *Pdfscreen manual*, 2002. `http://ftp.cstug.cz/pub/tex/CTAN/macros/latex/contrib/supported/pdfscreen/manual-screen.pdf`.
13. CV Radhakrishman, CV Rajagopal, a Chambert-Loir Antoine. *Trivial experiments with PsTricks manipulation*, září 2001. `http://ftp.agh.edu.pl/pub/tex/macros/latex/contrib/supported/pdftricks/manual.pdf`.
14. Sebastian Rahtz. *Hypertext marks in LaTeX: the* ***hyperref*** *package*, 1998. `TeXLive6: $TEXMF/doc/latex/hyperref/manual.pdf`.
15. D. P. Story. The AcroTeX eDucation Bundle. Home page: `http://www.math.uakron.edu/~dpstory/webeq.html`.
16. Adobe systems Incorporated. *PDF reference manual, v. 1.4*, second edition, 2000. `http://partners.adobe.com/asn/developer/acrosdk/DOCS/PDFRef.pdf`.

17. Philip Taylor. Computer typesetting or electronic publishing? New trends in scientific publication. V *Zpravodaj Československého sdružení uživatelu TEXu*, 5(1–4):61–89, 1995.
18. Hàn Thế Thành. Seznam nových primitiv pdfTEXu. Soubor: `http://www.fi.muni.cz/~thanh/download/pdftex/syntax.txt`.
19. Hàn Thế Thành. Alternativní výstup programu TEX – PDF. *Zpravodaj Československého sdružení uživatelu TEXu*, 6(2):69–85, 1996.
20. Hàn Thế Thành. Přenositelný formá dokumentu a sázecí systém TEX. Diplomová práce, Masarykova univerzita v Brně, fakulta informatiky, 1996.
21. Hàn Thế Thành. *Micro-typographic extensions to the TEX typesetting system*. Disertační práce, Masarykova univerzita v Brně, fakulta informatiky, 2000.
22. Mark A. Wicks. Program dvipdf. Home page: `http://gaspra.kettering.edu/dvipdfm/`.
23. Timothy Van Zandt. *PSTricks: PostScript macros for Generic TEX*, 1993. User's guide.
24. Vít Zýka. Používáme pdfTeX: vkládání obrázku. *Zpravodaj Československého sdružení uživatelu TEXu*, 11(4):181–186, prosinec 2001.
25. Vít Zýka. Používáme pdfTeX II: prezentace fotografií aneb jak na hypertext. *Zpravodaj Československého sdružení uživatelu TEXu*, 12(1):13–21, březen 2002.
26. Vít Zýka. Používáme pdfTeX III: video a zvuk v prezentaci. *Zpravodaj Československého sdružení uživatelu TEXu*, 12(2), březen 2002. V tisku.

Makro OFS

Petr Olšák

Elektrotechnická fakulta ČVUT, Praha
Email: olsak@math.feld.cvut.cz

Abstrakt: OFS (Olšákův fontový systém) je balíček TeXových maker, který umožní získat větší přehled nad rozsáhlými kolekcemi fontů a umožní s nimi poměrně snadnou manipulaci. Balík byl vyvinut pro získání přehledu nad fonty z Typokatalogu Střešovické písmolijny [1]. Na základě poptávky LaTeXových uživatelů byl balík napsán ještě jednou pro LaTeX, kde využívá NFSS a pokouší se je trošičku vylepšit. Základní uživatelské příkazy OFS jsou pak v obou prostředích (plain i LaTeX) stejné. Na přednášce předvedu použití OFS na uživatelské i konfigurační úrovni.

Klíčová slova: OFS, fonty, NFSS, LaTeX, CSplain, CSLaTeX, CSfonty

1 Úvodem

Makro OFS jsem si napsal hlavně proto, abych se vyznal ve stovkách fontů, které pocházely ze Střešovické písmolijny a pro které jsem před rokem udělal TeXovou podporu. Protože jsem plainista, šlo mi hlavně o to udělat makro co nejvíce srozumitelné uživateli plainu, který potřebuje přesně vědět, co to makro dělá. Tento požadavek například LaTeXové NFSS nesplňuje.

Později se začali o OFS zajímat též někteří uživatelé LaTeXu. Tož jsem se přemohl a pokusil OFS napsat ještě jednou, tentokrát pod LaTeXem s využitím NFSS. Cílem tohoto přepsání bylo hlavně to, aby se uživatelské příkazy pro vyhledávání a přepínání rodin fontů zcela shodovaly v LaTeXové verzi s verzí plainovou. LaTeXová verze přesto umí podstatně méně věcí, než plainová, protože pokud by měla umět vše, musel bych NFSS zcela odmítnout a fontový modul LaTeXu si napsat po svém. Tím bych ale popřel LaTeX jako takový, takže jsem zůstal u respektování principů NFSS. Věci, které se v NFSS dají dělat velmi těžko, jsem raději nedělal.

Z těchto důvodů jsem se v LaTeXu nepouštěl ani do podpory matematických fontů. Koncepce matematiky v NFSS mi připadá jako plainistovi poněkud nesrozumitelná. Navíc velké kolekce fontů, které byly hlavní motivací OFS, jsou většinou textové. Pokud chce někdo použít matematickou sadu fontů v LaTeXu, použije `\usepackage{styl}` a ten styl je k té matematické sadě většinou dodáván. Nepovažoval jsem tedy řešení matematiky v LaTeXu za prioritní. Na druhou stranu matematické rodiny v plainu se pomocí OFS doplňují a zavádějí ve všech velikostech velmi elegantně. Věřím, že alespoň nějakému plainistovi (kromě mě) se toto řešení bude hodit.

Jan Kasprzak, Petr Sojka (editoři): SLT 2002 – sborník semináře o Linuxu a TeXu, str. 79–92, 2002.

Makro OFS jsem zveřejnil na [2] pod TEX-like licencí, tj. je volně k mání, ale změny pod stejným názvem nesmí dále šířit nikdo jiný, než autor. Do TEXových distribucí jsem makro zatím neprosadil, protože bohužel chybí anglická dokumentace. Domnívám se, že kdyby anglická dokumentace byla, nebyl by se zařazením do TEXových distribucí problém a makro by mohlo využít daleko více lidí. Bohužel, nejsem v angličtině natolik zdatný, abych tento problém rychle překonal. Pro letošní letní prázdniny jsem sice plánoval, že se pokusím anglickou dokumentaci napsat, jenže přišla voda...

K OFS existuje samozřejmě česká dokumentace [3] podrobně popisující chování makra. Navíc jsem se o OFS zmínil už v článku o TEXové podpoře Štormových fontů [4]. Dovolil jsem si zde přesto znovu k tomuto tématu vrátit. Nechci být nyní tak technicky exaktní, jako v manuálu [3], ale zase mám zde více místa k rozepsání možností makra, než jsem měl ve článku [4], který byl především o Štormových fontech.

2 Základy uživatelského rozhraní

Uživatelské rozhraní je shodné v LATEXové verzi OFS i v plainové s výjimkou snad zavedení makra OFS, které v plainu provedeme jednoduše pomocí `\input ofs [kolekce, fontů]`, zatímco v záhlaví LATEXového dokumentu píšeme `\usepackage [kolekce, fontů] {ofs}`.

Abych v tomto textu nemusel každou chvíli větvit svůj výklad na situaci vhodnou pro plain a pro LATEX, rozhodl jsem se zde předpokládat, že pracujeme pouze s plainem (například s CSplainem). LATEXový uživatel si bude muset dohledat specifika verze OFS pro svůj formát v dokumentaci [3].

Příkazem `\fontusage` dostaneme na terminál a do logu základní informace o uživatelských příkazech:

```
$ tex ofs \\fontusage
This is TeX, Version 3.14159 (Web2C 7.3beta5)
(/usr/local/share/texmf/tex/csplain/ofs.tex
OFS (Olsak's Font System) based on plain initialized. <Oct. 2002>
(/usr/local/share/texmf/tex/csplain/ofsdef.tex))
\fontusage: ============== Olsak's Font System, usage: =================
\input ofs [sjannon, sdynamo, a35] ... for example
\showfonts  ... shows all loaded font families (by previous \input)
\setfonts [Family/] ... local switch to the new family, after this, the
  \rm, \bf, \it, bi will switch to the variants. The current size is used.
\setfonts [/size] ... local switch to the new size of fonts, the family is
  not changed. The "size" has the following possible formats:
    at<dimen>       ... the same as \font\something=file at<dimen>
    <dimen>         ... the same as at<dimen>
    <number>        ... the same as at<number>pt
    scaled<number>  ... the same as \font\something=file scaled<number>
    mag<decimal-number> fonts will be magnified by given coefficient
                        depend on current size of the fonts.
\setfonts [Family/size] ... switch to the new family at given size
\setfonts [Family-vr/]  ... switch to the specified font, the current size
```

```
  is used. The "vr" is acronym for variant (bf for example).
\setfonts [Family-vr/size] ... switch to the specified font.
\fontdef\name [Family/size] ... same as \gdef\name{\setfonts[Family/size]}
  The "Family" or "size" parameter may be empty.
\fontdef\name [Family-vr/size] ... \name is fixed-font switch iff:
  "size" is no empty and no mag<dec-number>.
  Fixed-font switch "\name" is implemented as \global\font\name=file.
\setmath [size/size/size] ... set math it/rm as current it/rm + use PS Symbol
\nofontmessages, \logfontmessages, \displayfontmessages, \detailfontmessages
  ... the levels of log.
*
```

Vidíme, že příkaz `\showfonts` nám ukáže kolekce fontů. Pokud jsme nepoužili ‘previous `\input`’, dopadne základní kolekce (implementovaná přímo uvnitř OFS) takto:

```
*\showfonts
OFS (1.0): The list of known font families:
defaults:
   [CMRoman/]            \rm, \bf, \it, \bi, \sl
   [CMSans/]             \rm, \bf, \it,  -
   [CMTypewriter/]       \rm,  - , \it,  - , \sl
   [Times/]              \rm, \bf, \it, \bi
   [Helvetica/]          \rm, \bf, \it, \bi, \nrm, \nbf, \nit, \nbi
   [Courier/]            \rm, \bf, \it, \bi
```

Názvy fontových rodin jsou zde uvedeny v hranatých závorkách a vedle jsou uvedeny přepínače variant, které pro danou rodinu je možné použít.

Vidíme, že nejběžnější textové fonty Computer Modern jsou zaneseny v OFS do tří rodin `CMRoman`, `CMSans` a `CMTypewriter`. Myslím, že netřeba dodávat, co to znamená. V rodinách `CMSans` a `CMTypewriter` není k dispozici jinak obvyklá varianta `BoldItalic`, pro kterou je vyhrazen přepínač `\bi`. V rodině `CMTypewriter` nenajdeme ani přepínač `\bf` pro tučnou variantu, ale zato můžeme použít „nadstandardní“ variantu `\sl` (slanted), která je k dispozici i v rodině `CMRoman`.

Deklarace dalších písmových rodin jsou zanášeny do deklaračních souborů s příponou `tex`[1]. V každém distribuci TEXu doporučuji udržovat soubor volající všechny deklarační soubory `allfonts.tex`. Například na mém počítači vypadá tento soubor takto:

```
$ cat ‘kpsewhich allfonts.tex‘
%%% All OFS families declared on this TeX
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%                              Petr Olsak
\input a35       % PostScript 35
\input ffonts    % Another free fonts
\input btfonts   % Bitstream fonts
\input skatalog  % Stromtype foundry, 89 families
```

Když tedy napíšu `$ tex allfonts \\showfonts \\end | less` , dostanu na svém počítači výpis zhruba tří set písmových rodin, každá obvykle ve čtyřech

[1] V LATEXovém OFS se jedná o příponu `sty`.

variantách. Důležité je, že se v tom vyznám a nemusím vzpomínat, jak se která metrika jmenuje. Následuje jen část výpisu.

```
OFS (1.0): The list of known font families:
defaults:
   [CMRoman/]            \rm, \bf, \it, \bi, \sl
   [CMSans/]             \rm, \bf, \it,  -
   [CMTypewriter/]       \rm,  - , \it,  - , \sl
   [Times/]              \rm, \bf, \it, \bi
   [Helvetica/]          \rm, \bf, \it, \bi, \nrm, \nbf, \nit, \nbi
   [Courier/]            \rm, \bf, \it, \bi
a35.tex:
   [AvantGarde/]         \rm, \bf, \it, \bi
   [Bookman/]            \rm, \bf, \it, \bi
   [NewCentury/]         \rm, \bf, \it, \bi
   [Palatino/]           \rm, \bf, \it, \bi
   [ZapfChancery/]       \rm,  - , \it,  -
   [ZapfDingbats/]       \rm,  - ,  - ,  -
   [Symbol/]             \rm,  - , \it,  -
ffonts.tex:
   [Charter/]            \rm, \bf, \it, \bi
...
sjannon.tex:
   [JannonAntikva/]      \rm, \bf, \it, \bi, \mr, \mi
   [JannonText/]         \rm, \bf, \it, \bi, \mr, \mi
   [JannonCaps/]         \rm, \bf, \it, \bi
...
sdynamo.tex:
...
   [DynaGroteskLE/]      \rm, \bf, \it, \bi
   [DynaGroteskD/]       \rm, \bf, \it, \bi
   [DynaGroteskR/]       \rm, \bf, \it, \bi
...
stitul.tex:
   [Alcoholica/]         \rm, \bf, \it,  -
   [Monarchia/]          \rm, \bf,  - ,  -
   [MonarchiaText/]      \rm, \bf,  - ,  -
   [Clichee/]            \rm, \bf, \it, \bi
   [Regula/]             \rm,  - , \it,  -
   [Splendid/]           \rm, \bf, \it,  - , \script, \sans
   [Cobra/]              \rm, \bf,  - ,  -
   [ExcelScript/]        \rm,  - ,  - ,  - , \ext
   [ExcelScriptText/]    \rm,  - ,  - ,  - , \ext
   [Zeppelin/]           \rm, \bf,  - ,  - , \lr, \coll
   [Negro/]              \rm,  - ,  - ,  -
   [Farao/]              \rm, \bf,  - ,  - , \kr, \coll
...
slido.tex:
   [Lido/]               \rm, \bf, \it, \bi, \crm, \cbf
```

Ve výpisu je uveden i TEXový soubor, kde se deklarace odpovídajících rodin skutečně nachází. Vidíme tedy, že `a35.tex` už obsahuje deklarace rodin, ale například soubor `skatalog.tex` zřejmě obsahuje jen další `\input sjannon`,

`\input sdynamo`, `\input stitul` atd. Když si pro svůj dokument vyberu třeba rodiny `JannonText`, `DynaGroteskR` a `Farao`, pak místo toho, abych psal

```
\input ofs [allfonts]
```

si vystačím s

```
\input ofs [sjannon, sdynamo, stitul]
```

což způsobí načítání podstatně méně souborů s deklaracemi fontových rodin.

Pokud potřebuji vědět, jak ty fonty vypadají, napíšu například:

```
$ pdfcsplain ofscatal [sjannon, sdynamo, stitul] ; acroread ofscatal.pdf
```

Soubor `ofscatal.tex`, který vytvoří katalog, byl zařazen do balíku OFS od verze `Oct 2002`.

3 Přepínač rodin a velikostí

Hlavním příkazem OFS na uživatelské úrovni je `\setfonts`. Jedná se především o přepínač rodin, viz výpis `\fontusage` v předchozí sekci. Přepínač má dva parametry v hranaté závorce oddělené lomítkem. Prvním parametrem je název rodiny a druhým požadovaná velikost fontů. Pokud některý parametr chybí, přepínač nebude jeho nastavení měnit. Přepínač samozřejmě funguje lokálně ve skupinách, jako bývá u fontových přepínačů obvyklé. Po přepnutí rodiny pracují přepínače variant (obvykle `\rm`, `\bf`, `\it`, `\bi`) pro novou rodinu. Varianta se po přepnutí rodiny inicializuje stejná, jako byla před přepnutím, tj. jsme-li zrovna v kurzívě rodiny Times a přepneme do Helveticy, zůstáváme ve variantě `\it` rodiny Helvetica, tj. skloněné písmo. Pokud ale při přepnutí rodiny aktuální varianta v nové rodině neexistuje, inicializuje se varianta `\rm`, kterou musí obsahovat každá rodina.

Příklady

```
\setfonts [JannonText/10.5]  % nastavím na začátku dokumentu
\setfonts [/14]\bf           % například pro nadpisy
\setfonts [/8]\rm            % pro poznámky pod čarou
\setfonts [DynagroteskR/]\it % třeba pro citáty
\setfonts [CMTypewriter/]    %  pro strojopis.
       % tento přepínač např. v kurzívě nastaví automaticky
       % variantu kurzíva-strojopis.
```

Výhodou tedy je, že si nemusím pamatovat názvy matrik (v LaTeXu pak nesrozumitelné zkratky rodin používané v NFSS), ale píšu název rodiny do dokumentu stejným způsobem, jak jej vidím v písmovém katalogu. Pokud udělám v názvu rodiny překlep (třeba nedodržím velká a malá písmena), příkaz `\setfonts` spustí `\showfonts`, tj. na obrazovce a v logu vidím seznam všech rodin, které mohu použít.

Velikost fontů mohu nastavit přímo udáním v jednotce `pt`, nebo připsáním jiné jednotky (např. `mm`), nebo jako u primitivu `\font` pomocí slova `scaled` (koeficient zvětšení se váže k základní velikosti fontu) a konečně i pomocí zcela

nového prefixu mag, který udává zvětšení vzhledem ke zrovna použité velikosti fontů, například:

```
\def\maly{\setfonts [/mag0.8]}
stanu se {\maly menším a {\maly menším a {\maly ještě menším}}}
a už jsem se z toho dostal.
```

dopadne takto:

stanu se menším a menším a ještě menším a už jsem se z toho dostal.

Prefix mag pro velikost využijeme například v logu LaTeX, které jsem si v plainu definoval jako

```
\def\LaTeX{L\kern-.2em\raise.45ex\hbox{\setfonts[/mag.7] A}\kern-.05em\TeX}
```

a funguje to v nadpisech (je tam tučné vyvýšené A odpovídající velikosti) i v poznámkách pod čarou. V LaTeXu samotném mají toto logo definováno tak, že se pro vyvýšené A použije font v indexové velikosti. No jo, zde jim to náhodou prochází, ale co kdyby se (například v jiném v logu) hodilo pro některé písmeno použít velikost mag.8, která není jako indexová velikost použita? Bez OFS bychom to pak dělali dost obtížně, protože NFSS nám řešení nenabízí.

Dalším využitím prefixu mag je možnost korigovat nestejné střední výšky písma. Například v tomto sborníku je pro strojopis použit CMTypewriter. Pokud si všimnete v jiných příspěvcích, než tento, vidíte, že uvnitř odstavců ten strojopis s fontem Charter moc nesedí, protože strojopis má menší střední výšku písma. Stačí ale pro přepínač \tt použít \setfonts[CMTypewriter/mag1.1], a střední výška je v lati. To vidíme například v tomto příspěvku. A funguje to včetně zmenšených variant, které jsou v příspěvcích pro SLiT použity v abstraktu, poznámce pod čarou, nebo třeba v údaji „Email“, uvedeném v záhlaví příspěvku.

Příkaz \setfonts může obsahovat i specifikaci varianty (viz výpis příkazu \fontusage). Pak se už nejedná o přepínač rodiny, ale fontu samotného. V takovém případě přepínač \setfonts neovlivní přepínače variant \rm, \bf a dalších, ale nastaví jen požadovaný jediný font. Například pro nadpisy by šlo psát \setfonts[-bf/14] místo \setfonts[/14]\bf, ale museli bychom mít jistotu, že v nadpisu nepoužijeme přepínač varianty, protože ten by vrátil font do velikosti aktuální rodiny. Tou velikostí je třeba 10pt.

Abychom v nadpisu mohli použít přepínač \it, a přitom se dostali do varianty \bi (což je žádoucí), je potřeba přepínač fontu v makru pro nadpis naprogramovat zhruba takto:

```
\def\nadpis #1{{\setfonts[/14]\bf \let\it=\bi #1}}
```

Kvůli tomuto jednoduchému obratu, který zvládne snad každý plainista, jsem se rozhodl nekomplikovat jádro OFS podobným způsobem jako NFSS a nezavádět tedy další „nezávislou souřadnici“ popisující duktus fontu. Pro speciální rodiny fontů (jako je třeba rodina DynaGrotesk) jsem ale snadno pomocí doplňujících maker vytvořil přepínač, který respektuje ještě o jednu „nezávislou souřadnici“ více, než zvládá NFSS. V tomto případě to nebyl jen duktus, ale i stupeň zúžení písma. Navíc tento přepínač dokáže „poskakovat“ po jednotkách vpřed a vzad podél zvolené souřadnice. To také NFSS nezvládá.

Často se hodí vytvářet zkratky pro přepínače rodin. K tomu slouží makro `\fontdef\přepínač[Rodina/velikost]`, které funguje podobně, jako použití konstrukce `\gdef\přepínač{\setfonts[Rodina/velikost]}`. V záhlaví tohoto dokumentu mám například uvedeno:

```
\input ofs [ffonts]                % Charter je ve skupině free fonts
\setfonts [Charter/10pt]           % výchozí rodina
\fontdef\tt [CMTypewriter/mag1.1]  % strojopis, korekce střední výšky
\fontdef\verbtt [CMTypewriter/8]   % strojopis pro display ukázky
\fontdef\small [Charter/9]         % zmenšení pro abstrakt a záhlaví
```

Možná čtenáře napadne, že jsem nemusel psát podruhé slovo `Charter` v deklaraci přepínače `\small`. Při kompletní změně základní rodiny dokumentu na jinou bych pak mohl změnit slovo `Charter` jen na jediném místě (u příkazu `\setfonts`) a byl bych hotov. Bohužel, vynechání rodiny v příkazu `\small` vede k problémům, protože tento příkaz je použit v záhlaví dokumentu, tj. v `\output` rutině. Pokud zde neuvedeme explicitně rodinu, dědí se rodina aktuálně použitá, která ovšem může být jakákoli, protože `\output` rutina je vyvolávaná z různých míst při zpracování dokumentu. Mít záhlaví jednou strojopisem a podruhé Charterem podle toho, odkud byla `\output` rutina zavolána, je samozřejmě nežádoucí.

Tento problém se dá řešit „vykřičníkovou“ konvencí, kterou umí zpracovat příkaz `\fontdef`:

```
\fontdef\small [!/9] \addcmd\small {\rm} % pro abstrakt a záhlaví
```

Vykřičník je nahrazen aktuální rodinou už v době činnosti příkazu `\fontdef`, nikoli tedy až v době provádění přepínače. V `\output` rutině se musíme ještě postarat o potlačení dědičnosti aktuální varianty. Proto jsem použil ještě příkaz `\addcmd` (zaveden nově od verze OFS `Oct 2002`), který k existujícímu makru přidá další příkazy na jeho konec. Celá deklarace fontů v tomto dokumentu tedy vypadá zhruba takto:

```
\input ofs [ffonts]                    % Charter je ve skupině free fonts
\setfonts [Charter/10pt]               % výchozí rodina
\fontdef\tt [CMTypewriter/mag1.1]      % strojopis, korekce střední výšky
\fontdef\verbtt [CMTypewriter-rm/8]    % strojopis pro display ukázky
\fontdef\small [!/9]
\addcmd \small {\baselineskip11pt\rm} % zmenšení pro abstrakt a záhlaví
\fontdef\fontsekce [!/12]
\addcmd \fontsekce {\bf \let\it=\bi}  % pro nadpisy sekcí
\fontdef\fonttitul [!-bf/14.4]        % v titulu nebudeme přepínat varianty
```

Povedlo se tedy veškeré aktivity související s výběrem fontů soustředit na jedno místo dokumentu bez nutnosti používat názvy metrik.

4 Kódování fontů

OFS pro plain implicitně pracuje v kódování TS fontů, ale můžete jej přepnout do jiného kódování.[2]

[2] OFS pro LaTeX přenechává starost o kódování zcela na NFSS.

Základem informace o kódování v OFS pro plain je makro `\fotenc`, které má implicitně hodnotu `8z`. To znamená, že rodiny `CMRoman`, `CMTypewriter` a `CMSans` budou pracovat s CSfonty. Kdyby uživatel nastavil `\def\fotenc{8t}`, začalo by OFS pracovat v případě těchto rodin s DC nebo EC fonty. Jaké je technické pozadí vysvětlíme v následující sekci.

Většina metrik PostScriptových fontů má varianty `*8t` (pro kódování podle Corku) a `*8z` (pro kódování podle CSfontů). Makro `\fotenc` tedy musí obsahovat koncovku metrik, které chceme použít. Tím volíme kódování. Příklad:

```
\input ofs
\setfonts [Times/]  text 1 % metrika: ptmr8z, tj. kódování CSfontů
\def\fotenc{8t}
\setfonts [/]       text 2 % metrika: ptmr8t  tj. kódování podle Corku
```

Máte-li své fonty v nějakém dalším kódování a koncovky metrik máte s názvy třeba `*8x`, pak není problém napsat `\def\fotenc{8x}` a můžete začít používat své vlastní metriky.

Pokud používáte kódování podle Corku, doporučuji definovat makro `\fotenc` jako `8t` před zavedením makra OFS. Při použití složitěji deklarovaných rodin se to hodí.

V balíku OFS jsou soubory `ofs-8z.tex` a `ofs-8t.tex`, ve kterých je deklarace akcentů a některých na kódování závislých maker, jako například `\promile`. Implicitně není načten ani jeden z těchto souborů, tj. jsou respektována makra pro akcenty z originálního plainu, která expandují na `\accent` podle kódování CMfontů.

Pokud je toto chování nevyhovující, můžete načíst pomocí `\input` jeden nebo oba dva výše zmíněné soubory (v libovolném pořadí – jejich definice se nehádají). Pokud načtete oba soubory, pak makra pro akcenty expandují na znaky podle kódování, které je aktuálně nastaveno v makru `\fotenc` (`8z` nebo `8t`).

Podívejme se do souborů `ofs-8z.tex` a `ofs-8t.tex` (výrazně kráceno):

```
%%% Default accents in CM
\accentdef  \‘ *    8z  {\accent 18 }                 % grave
\accentdef  \’ *    8z  {\accent 19 }                 % acute
\accentdef  \v *    8z  {\accent 20 }                 % caron
\accentdef  \u *    8z  {\accent 21 }                 % breve
...
%%% Standard characters in plain (redefined here)
\def\aa{\r a}
\def\AA{\r A}
\characterdef \i    8z  16
\characterdef \j    8z  17
\characterdef \SS   8z  {SS}
\characterdef \AE   8z  29
...
%%% Extra characters from CS-fonts
\characterdef \promile      8z  141
\characterdef \varhyphen    8z  156
\characterdef \flqq         8z  158
\characterdef \frqq         8z  159
```

```
\characterdef \clqq         8z  254
\characterdef \crqq         8z  255
...
%%% Accented letters from CS fonts
\accentdef  \‘ A      8z  152
\accentdef  \’ A      8z  193
\accentdef  \" A      8z  196
\accentdef  \‘ a      8z  184
\accentdef  \’ a      8z  225
...
%%% Default accents in Cork
\accentdef  \‘ *    8t  {\accent 0 }
\accentdef  \’ *    8t  {\accent 1 }
\accentdef  \^ *    8t  {\accent 2 }
\accentdef  \  *    8t  {\accent 3 }
\accentdef  \" *    8t  {\accent 4 }
...
%%% Standard characters in plain (redefined here)
\def\aa{\r a}
\def\AA{\r A}
\characterdef \i    8t  25
\characterdef \j    8t  26
\characterdef \SS   8t  223
\characterdef \AE   8t  198
...
\characterdef \promile              8t {\%\char 24 }
\characterdef \textpertenthousand   8t {\%\char 24\char 24 }
...
%%% Accented letters from Cork encoding
\accentdef  \. i    8t ‘\i
\accentdef  \u A    8t 128
\accentdef  \k A    8t 129
\accentdef  \’ C    8t 130
\accentdef  \v C    8t 131
...
```

Myslím, že syntaxe a význam příkazů `\characterdef` a `\accentdef` je v této ukázce samovysvětlující. Pokud ne, odkazuji čtenáře do dokumentace [3], která do posledního detailu (včetně popisu jednotlivých fází expanze) vysvětluje činnost těchto příkazů.

LaTeXovému uživateli to může připomínat příkazy `\DeclareTextSymbol` a `\DeclareTextComposite` a jim podobné, které dělají zhruba totéž (jen poněkud komplikovaněji, nepřehledněji a těžkopádněji).

OFS také počítá s možností, že některé znaky jsou přidány do extra fontu, který s původním fontem vytváří uspořádanou dvojici. Takové dvojice metrik jsou použity například pro fonty Štormovy písmolijny, protože obsahují více než 256 znaků. Přitom by bylo škoda některé znaky nevyužít. Ke každé metrice těchto fontů (`*8z` i `*8t`) je přiřazena extra metrika `*6s`, obsahující zbylé znaky. Pomocí `\characterdef` a `\accentdef` můžeme deklarovat přítomnost těchto znaků v kódování `6s` (viz soubor `stormenc.tex`). Pokud aktuální metrika má k sobě extra font v kódování `6s` a je požadován znak z tohoto fontu, OFS jej automaticky „vyloví“ pomocí přechodného přepnutí na extra metriku.

5 Deklarace rodin

Podívejme se nyní do souboru `a35.tex`, který deklaruje rodiny pro základních 35 PostScriptových fontů (kráceno):

```
%%% Times, Helvetica, Courier is in OFS defaults
\ofsdeclarefamily [AvantGarde] {% ---------- AvantGarde
   \loadtextfam (Book)         pagk\fotenc;%
                (Demi)         pagd\fotenc;%
                (BookOblique)  pagko\fotenc;%
                (DemiOblique)  pagdo\fotenc;;%
   \def\TeX{T\kern-.08em\lower.3333ex\hbox{E}\kern-0.09emX}%
}
\ofsdeclarefamily [NewCentury] {% ---------- NewCenturySchlbk
   \loadtextfam (Roman)        pncr\fotenc;%
                               pncb\fotenc;%
                               pncri\fotenc;%
                               pncbi\fotenc;;%
   \def\TeX{T\kern-.1667em\lower.5ex\hbox{E}\kern-.125emX}%
}
...
```

Zde je tedy deklarováno zobrazení mezi názvy rodin a metrikami.[3] Každá rodina má čtyři povinné metriky (pro běžné čtyři varianty). Metriky jsou zapsány pomocí `\fotenc`, aby fungovalo přepínání kódování. Mezi předposledním a posledním středníkem může být uvedeno extra kódování, pokud je font rozložen do dvou metrik (v tomto příkladě není). Příkazy uvedené v parametru `\ofsdeclarefamily` se provedou při každém přepnutí rodiny příkazem `\setfonts`. V ukázce tam máme alternativní definice loga `\TeX`, aby toto logo pěkně vypadalo v každém fontu.

Od verze OFS `Oct 2002` je přidána možnost použít nepovinné parametry v argumentech příkazu `\loadtextfam`. Tyto parametry píšeme do kulaté závorky a značí název varianty, pokud je odlišný od běžného názvu. Tato informace se použije v logu a při tisku katalogu.

Podívejme se ještě do souboru `sjannon.tex` z podpory Štormových fontů (výpis je zde neúplný):

```
\ofsdeclarefamily [JannonAntikva] {% ---- Jannon Antikva
   \loadtextfam sjnr\fotenc;%
                sjnb\fotenc;%
                sjnri\fotenc;%
                sjnbi\fotenc;6s;%
   \newvariant2 \mr (Medium)        sjnm\fotenc;6s;%
   \newvariant3 \mi (MediumItalic)  sjnmi\fotenc;6s;%
}
\ofsdeclarefamily [JannonText] {% ------ Jannon Text
   \loadtextfam sjnrg\fotenc;%
                sjnbg\fotenc;%
                sjnrig\fotenc;%
                sjnbig\fotenc;6s;%
```

[3] V OFS pro LaTeX takové soubory neexistují; tam se pouze deklaruje zobrazení mezi (dlouhými) názvy rodin v OFS a zkratkami rodin v NFSS.

```
   \newvariant2 \mr (Medium)        sjnmg\fotenc;6s;%
   \newvariant3 \mi (MediumItalic)  sjnmig\fotenc;6s;%
}
...
```

Vidíme, že navíc je zde deklarována extra metrika v kódování `6s` a že se zde deklarují doplňující přepínače variant `\mr` a `\mi` mimo základní čtyři varianty. Podrobnější vysvětlení použitých příkazů čtenář opět najde v dokumentaci [3].

Jak jsou pomocí expanze `\fotenc` mapovány metriky ℭ𝔰fontů resp. DC fontů, které v názvech koncovku `8z` ani `8t` nemají? To souvisí rovněž s tím, že tyto fonty mají pro různé velikosti různé metriky. V deklaračních souborech je pak tento problém vyřešen způsobem „dvě mouchy jednou ranou“ (viz soubor `ofsdef.tex`):

```
\registertfm cmr8z     -      csr10  % metrika pro všechny velikosti
\registertfm cmr8z  0pt-6pt   csr5
\registertfm cmr8z  6pt-7pt   csr6
\registertfm cmr8z  7pt-8pt   csr7
\registertfm cmr8z  8pt-9pt   csr8
\registertfm cmr8z  9pt-10pt  csr9
\registertfm cmr8z  10pt-12pt csr10
\registertfm cmr8z  12pt-17pt csr12
\registertfm cmr8z  17pt-*    csr17
...
\registertfm cmr8t     -   dcr10     % metrika pro všechny velikosti
...
\ofsdeclarefamily [CMRoman] {% ------ Computer Modern Roman
   \loadtextfam cmr\fotenc;%
                cmbx\fotenc;%
                cmti\fotenc;%
                cmbxti\fotenc;;%
   \newvariant8 \sl (Slanted) cmsl\fotenc;;%
}
```

Centrálním příkazem je zde makro `\registertfm`, které mapuje neexistující metriky formálně vytvořené pomocí expanze `\fotenc` do skutečných metrik. Toto makro také umí registrovat různé metriky pro různé velikosti. Pokud tedy budeme chtít font daný pomocí `\setfonts[CMRoman-rm/12.5]`, vyvolá se při `\fotenc` s hodnotou `8z` metrika `csr12 at12.5pt`. „Metrika pro všechny velikosti“ se použije v případě, že je velikost fontu deklarovaná s prefixem `scaled`, takže `\setfonts[CMRoman-rm/scaled1250]` vyvolá metriku `csr10 scaled1250`. Ztrácíte-li přehled o tom, jaká metrika se ve skutečnosti použila, můžete zapnout logovací přepínač `\detailfontmessages`.

Podíváte-li se do souboru `ofsdef.tex` podrobněji, můžete si všimnout, že jsem metriky rodiny `CMRoman` a dalších `CM` rodin pro kódování `8t` poněkud odbyl. Registroval jsem je jako metriky DC fontů bez střídání metrik pro různé velikosti. Nechtělo se mi to totiž vypisovat a nevěděl jsem, zda uživatel tohoto kódování raději nepoužije EC fonty místo DC fontů. Ve zmíněném souboru jsou příklady, jak by se ta deklarace měla provést. Pokud ji někdo dopíše (například pro DC i EC fonty do zvláštních souborů), rád ji do balíku OFS zařadím. Osobně ale fonty kódované v `8t` nepoužívám, takže mě zatím nic nemotivuje to udělat.

6 Matematické fonty

Každý plainista ví, že fonty pro matematiku se sdružují do matematických rodin obsahující font pro základní, indexovou a index-indexovou velikost. Rovněž ví, že první čtyři matematické rodiny (s číslem 0 až 3) jsou s TEXem jistým způsobem pevně významově svázány a další rodiny se dají deklarovat.

Aby plainista při deklarování matematických rodin nemusel psát třikrát za sebou `\font` a potom ještě `\textfont\rodina=...`, `\scriptfont...`, atd, je v OFS k tomu vytvořena zkratka `\loadmathfam`, jak ukážu za chvíli.

Inicializaci matematických fontů v OFS pro plain provedeme pomocí příkazu `\setmath`. Dokud tento příkaz nepoužijeme, jsou matematické fonty ve stavu, jak je inicializoval plain. Ve třech parametrech příkazu `\setmath` (oddělených lomítky) dáváme najevo, jakou chceme základní velikost, jakou pro indexy a jakou pro indexy indexů. Jsou-li tyto parametry prázdné, je použito relativní zmenšení podle aktuální velikosti textového fontu pomocí prefixu `mag` takto:

```
\setmath [//] je totéž jako \setmath [mag1.0/mag.7/mag.5]
```

Příkaz `\setmath` vypočítá požadované velikosti a startuje makro pro zavedení fontů `\mathfonts`. Dále tento příkaz startuje makro pro inicializaci matematického kódování `\mathchars`. Tato makra si může plainista definovat jak chce, nicméně většinou využije již připravených maker `\defaultmathfonts` a `\defaultmathchars`, která jsou udělána tak, že spouštějí různé varianty kódu podle hodnoty maker `\fomenc` a `\mathversion`.

Při `\def\fomenc{PS}` (imlicitní hodnota: PostScriptové fonty) se zavedou fonty tak, že matematická kurzíva se ztotožní s aktuální textovou kurzívou a podobně rodina 0 pro číslice a textové symboly zůstává nastavena podle aktuální textové rodiny varianty `\rm`. Matematické symboly se berou (pokud to je možné) z běžně dostupného PostScriptového fontu `Symbol`. Zbytek (např. natahovací závorky) pak zůstává v Computer Modern. Matematické kódování je pro tuto situaci výrazně pozměněno obvyklými primitivy `\mathchardef` a podobnými, aby byly dosažitelné všechny znaky plainu. Například pro řecká písmena musela být zavedena nová rodina se skloněným fontem `Symbol` a příkazy typu `\alpha` jsou překódovány z původní matematické rodiny 1 na tuto novou rodinu.

Nastavíte-li `\def\fomenc{CM}`, pak příkaz `\setmath` zavede stejné matematické fonty z rodiny Computer Modern, jako v plainu. Příkaz `\setmath[//]` pak pouze aktualizuje velikosti těchto fontů podle velikosti aktuálního textového fontu.

Nakoupíte-li fonty MathTimes, dále načtete soubor `ofsmtdef.tex` (součástí OFS od verze `Jun. 2002`) a definujete `\def\fomenc{MT}`, pak příkaz `\setmath` zavede kurzívu a rodinu 0 stejně jako při `\def\fomenc{PS}`, ale navíc použije pro *všechny* symboly včetně natahovacích závorek fonty MathTimes.

Kromě hodnoty `\fomenc` se příkaz `\setmath` větví i vzhledem k hodnotě makra `\mathversion`. Implicitně OFS počítá se dvěma hodnotami tohoto makra: `\def\mathversion{normal}` nebo `bold`. Při verzi `bold` jsou načteny do

matematických rodin tučné alternativy fontů, pokud to jde. Vše názorně vysvětlí pohled do souboru `ofsdef.tex`:

```
\def\defaultmathfonts{\csname load\fomenc\mathversion math\endcsname}
\def\defaultmathchars{\csname set\fomenc mathchars\endcsname}
\def\mathfonts{\defaultmathfonts}
\def\mathchars{\defaultmathchars
   \let\mathchars=\relax % to protect the twice math-setting
}
\def\loadPSnormalmath{%
  \loadmathfam 0[tenrm/]%              Actual Roman font
  \loadmathfam 1[tenit/]%              Actual Italic font
  \loadmathfam 2[/cmsy]%               Standard symbols from CM
  \noindexsize\loadmathfam 3[tenex/]%  Standard extra symbols from CM
  \chardef\itfam=1 \chardef\bifam=5
  \loadmathfam \bffam [tenbf/]%        Actual Bold font
  \loadmathfam \bifam [tenbi/]%        Actual Bold Italic
  \newmathfam\symbfam
  \loadmathfam \symbfam [/psyr]%       PostScript Symbol
  \newmathfam\symbofam
  \loadmathfam \symbofam [/psyro]%     PostScript Symbol Oblique
}
\def\loadPSboldmath{%
  \loadmathfam 0[tenbf/]%              Actual Bold font
  \loadmathfam 1[tenbi/]%              Actual Bold-Italic font
...
}
\def\loadCMnormalmath{%
  \loadmathfam 0[/cmr8z]%                    Roman font
  \loadmathfam 1[/cmmi]%                     Math Italic font
  \loadmathfam 2[/cmsy]%                     Standard symbols from CM
  \noindexsize\loadmathfam 3[/cmex10]%       Standard extra symbols from CM
...
}
\def\loadCMboldmath{%
  \loadmathfam 0[/cmbx8z]%                     Roman font
  \loadmathfam 1[/cmmib10]%                    Math Italic font
...
}
\def\setPSmathchars{%
  \fontmessage{\ofsmessageheader Math codes are set for PS encoding}%
  \mathcode`\,="602C
  \mathcode`\.="002E
  \delcode`\<="\hex\symbfam E130A
  \delcode`\>="\hex\symbfam F130B
  \delcode`\|="\hex\symbfam 7C30C
  \edef\langle{\delimiter"4\hex\symbfam E130A }%
  \edef\rangle{\delimiter"5\hex\symbfam F130B }%
  \mathchardef\alpha    "0\hex\symbofam 61
  \mathchardef\beta     "0\hex\symbofam 62
  \mathchardef\gamma    "0\hex\symbofam 67
...
```

```
}
```

Chceme-li přidat nějaké další matematické rodiny (v terminologii NFSS matematické abecedy), pak můžeme postupovat třeba takto:

```
\def\mathfonts{\defaultmathfonts
  \newmathfam\bbfam
  \loadmathfam \bbfam [/bbold12]%  Dvojitá vertikální kresba
  \def\bb{\fam\bbfam}%
}
\def\mathchars{\defaultmathchars
  \mathchardef\balpha    "0\hex\bbfam 0B
  \mathchardef\bbeta     "0\hex\bbfam 0C
  ...
}
```

Příklad z konce sekce 3 nyní obohatíme o možnost práce s matematikou v libovolných velikostech:

```
\input ofs [ffonts]                  % Charter je ve skupině free fonts
\setfonts [Charter/10pt]             % výchozí rodina
\setmath[//]                         % inicializace matematiky
\fontdef\tt [CMTypewriter/mag1.1]    % strojopis, korekce střední výšky
\fontdef\verbtt [CMTypewriter-rm/8]  % strojopis pro display ukázky
\fontdef\small [!/9]                 % zmenšení pro abstrakt a záhlaví
\addcmd \small {\baselineskip11pt \rm \def\mathversion{normal}\setmath[//]}
\fontdef\fontsekce [!/12]            % pro nadpisy sekcí
\addcmd \fontsekce {\bf \let\it=\bi \def\mathversion{bold}\setmath[//]}
\fontdef\fonttitul [!-bf/14.4]       % titul
```

Nyní i v abstraktech, poznámkách pod čarou a záhlavích je matematika ve všech velikostech všech možných indexů zmenšená odpovídajícím způsobem. Pokud by se matematika měla použít v názvu sekce, pak bude také tučná a správně veliká.

Reference

1. `http://www.pismolijna.cz`, `http://www.cstug.cz/stormtype`.
2. `ftp://math.feld.cvut.cz/pub/olsak/ofs`.
3. Petr Olšák. *OFS: Olšákův fontový systém*. 2001. Dokumentace k balíku je v souborech `ofsdoc.tex`, `ofsdoc.pdf`.
4. Petr Olšák. *Jak TEX k fontům ze Střešovic přišel*. Zpravodaj Československého sdružení uživatelů TEXu, 4/2001, strany 153–180.

DocBook

Jiří Kosek

LISP, VŠE Praha
Email: jirka@kosek.cz

Abstrakt: DocBook je dnes již standardním XML/SGML formátem pro tvorbu dokumentace, který používá mnoho open-source i komerčních projektů. Příspěvek posluchače seznámí se základními principy DocBooku a s volně dostupnými nástroji pro zpracování DocBookových dokumentů a jejich konverzi do dalších formátů jako HTML, PDF a další.

Jan Kasprzak, Petr Sojka (editoři): SLT 2002 – sborník semináře o Linuxu a TEXu, str. 93–93, 2002.

CSTEX – historie, současný stav a budoucnost

Petr Olšák

Elektrotechnická fakulta ČVUT, Praha
Email: olsak@math.feld.cvut.cz

Abstrakt: CSTEX je podpora češtiny a slovenštiny v TEXu a sestává ze tří základních pilířů: CSfonty, CSplain a CSLATEX. V přednášce bych se pozastavil podrobněji u každého z nich, zavzpomínal na to, co se kolem vývoje jednotlivých balíků odehrálo, upozornil na současné novinky a pokusil se odhadnout budoucí vývoj.

Klíčová slova: CSTEX, CSplain, CSLATEX, CSfonty

1 Počátky CSTEXu

Záznam o počátcích počeštění TEXu lze najít v článku Ladislava Lhotky [6]. Autor zde píše, že se na podzim roku 1988 sešel s Petrem Novákem a dohodli se, že udělají české osmibitové fonty (v METAFONTu) a vzory dělení slov. Nejprve to vypadalo tak, že Láďa Lhotka udělá fonty a Petr Novák vzory dělení, ale pak to dopadlo přesně naopak.

Doslechl jsem se také, že v době, kdy ještě neexistoval TEX ve verzi 3 (do roku 1989), Petr Novák vymyslel nějaké finty, při níž bylo možno dosáhnout českých akcentovaných znaků pomocí ligatur v TEXových metrikách.

Po vzniku CSTUGu (konec roku 1990) se iniciativy nad rozšiřováním a zvelebováním české a slovenské podpory pro TEX ujal Olin Ulrych. V té době se tento projekt začal nazývat CS TEX, později CSTEX. Olin zapracoval Lhotkovy vzory dělení a Novákovy fonty, přidal nějaká udělátka na úrovni preprocesoru a programy manipulující s `dvi`. První distribuce byly postaveny na PCTEXu, pak SBTEXu. V roce 1992 přešel CSTEX kopletně na emTEXovou distribuci pro DOS. Já jsem tehdy udělal program pro konfiguraci nabídek, který spolupracoval s DOSovými dávkami [7]. Členům sdružení jsme v letech 1993 a 1994 začali rozesílat disketové balíky s emTEXem a s českou a slovenskou podporou – CSTEX.

V té době většina členů používala DOS, takže emTEX pro DOS byl pro ně vyhovující. V dnešní době uživatelé pracují s nejrůznějšími operačními systémy a s různými distribucemi TEXu pro tyto systémy. Z toho důvodu se v tomto textu přidržíme jen užšího významu slova CSTEX, definovaného takto:

CSTEX je sada TEXových maker, fontů, vzorů dělení slov a doplňujícího software pro podporu české a slovenské sazby v TEXu. Je vytvořen tak, aby mohl být použit na libovolné TEXové distribuci na libovolném operačním systému.

V roce 1996 jsem napsal *Pár poznámek k novému CSTEXu* [2]. Tento dokument má z dnešního pohledu poněkud zastaralý název, protože software z roku 1996

Jan Kasprzak, Petr Sojka (editoři): SLT 2002 – sborník semináře o Linuxu a TEXu, str. 95–103, 2002.

dnes sotva můžeme považovat za nový. Nicméně zde čtenář může najít podrobnou historickou poznámku mapující léta 1992–1996. Už v tomto textu z roku 1996 se snažím vymezit pojem CSTEX poněkud šířeji, než jen emTEXová distribuce obohacená o podporu češtiny a slovenštiny. Vymezil jsem ho ovšem až moc široko: pořád jsem za CSTEX považoval kompletní fungující distribuci TEXu s českou a slovenskou podporou, i když pro blíže nespecifikovaný operační systém.

Na základě vymezení pojmu CSTEX z roku 1996 začaly vznikat ve ftp adresáři CSTEXu [1] podadresáře označené podle operačního systému či distribuce TEXu (např. `emtex`, `web2c`), které měly ambici nabídnout kompletní TEXovou distribuci včetně české a slovenské podpory. Bohužel, udržovat všechny nabízené distribuce v aktuálním stavu se nedalo při jednom muži zvládnout, a proto obsahy těchto adresářů velmi rychle zastarávaly.

V roce 2002 jsem se při psaní nového *Manuálu k CSTEXu* [3] znovu zamyslel nad obsahem pojmu CSTEX a pokusil jsem se jej vymezit poněkud opatrněji – stejně, jak je uvedeno zde. Značná část textu v tomto příspěvku pro konferenci SLT 2002 je převzata právě z [3].

Nově vymezený pojem CSTEX se zhruba shoduje s tzv. *jádrem CSTEXu*, což je slovní spojení použité už v roce 1996 v [2]. Jádro CSTEXu (nově tedy CSTEX) nabízím správcům jednotlivých TEXových distribucí k zařazení. Pro tyto účely jsem vytvořil nově na ftp adresáři CSTEXu adresář `base`, který obsahuje tary CSTEXu k zařazení do různých TEXových distribucí. Vzdávám se tedy původní myšlenky udržovat kopie TEXových distribucí na ftp adresáři CSTEXu v akuálním stavu.

CSTEX i v tomto užším smyslu dělím na část povinnou (required) a nepovinnou (recomended). Povinná část musí být kompletně instalovaná v TEXové distribuci, aby se o ní mohlo říci, že „obsahuje CSTEX“. Při vzájemné výměně dokumentů „psaných v CSTEXu“ je totiž otázka, zda distribuce obsahuje CSTEX, důležitá. Povinná část obsahuje TEXová makra, vzory dělení a fonty, které nejsou závislé na operačním systému ani na distribuci TEXu. Dají se tedy relativně snadno implementovat do jakékoli TEXové distribuce. Nepovinná část obsahuje další doprovodný software, jako například `csindex` (varianta programu `makeindex`) nebo program `vlna` na doplňování vlnek za předložkami.

Povinná část CSTEXu sestává ze tří základních pilířů: CSfonty, CSplain a CSLATEX. CSfonty jsou konzervativním rozšířením Knuthových Computer Modern fontů, CSplain je konzervativním rozšířením Knuthova formátu plain a konečně CSLATEX je jistou modifikací běžně používaného formátu LATEX. Do povinné části CSTEXu je ještě zahrnuta podpora použití základních 35 PostScriptových fontů v češtině a slovenštině prostřednictvím virtuálních fontů.

O historii, současném stavu a budoucnosti jednotlivých pilířů si povíme podrobněji v následujících kapitolách.

2 CSfonty

Tvary akcentů CSfontů byly vytvořeny a implementovány v jazyce METAFONTu Petrem Novákem ve spolupráci s českými výtvarníky někdy na začátku 90. let. Autor přenechal CSfonty CSTUGu, který s nimi může libovolně nakládat.

METAFONTový kód byl pak v letech 1992–1993 dále upraven Karlem Horákem. Karel se inspiroval z METAFONTových zdrojů pro fonty vytvořené v Polsku. Zapracoval tam možnost nastavení kódování fontu a vytvořil makra umožňující mít všechny hlavní `mf` soubory se stejným dvouřádkovým obsahem.

Na schůzce tvůrců CSTEXu na FEL v roce 1993 bylo rozhodnuto, že CSfonty budou mít kódování podle ISO 8859-2. Později, při implementaci CSTEXu do UNIXových distribucí nepodporujících změny xord/xchr vektorů se ukázalo, že to bylo velmi prozíravé rozhodnutí.

V roce 1993 jsem převzal údržbu CSfontů do svých rukou. Udělal jsem jen velmi drobné změny. Poslední 28. 9. 1996. Pak jsem vývoj CSfontů zmrazil podobným způsobem, jako Knuth přestal měnit CM fonty. Prioritním požadavkem je, aby dokument opírající se o CSfonty byl od roku 1996 formátován naprosto stejně dnes i kdykoli v budoucnu. Aby byl tento požadavek splněn, není tedy možné zasáhnout do rozměrů znaků, kernů a ligaturních tabulek.

V roce 1996 jsem do CSfontů přidal virtuální fonty podporující náhradu Computer Modern fonty a naopak.

V roce 1998 se podařilo dohodnout s autorem teTEXu Thomassem Esserem, aby zařadil do své distribuce CSfonty a celý CSTEX. Od této chvíle obsahují distribuce odvozené z teTEXu implicitně kompletní CSTEX.

V roce 1998 jsem také pro potřeby výstupu do formátu PDF vytvořil variantu CSfontů, tentokrát ve formátu PostScript Type1. Vyšel jsem z BaKoMa Type1 implementace CM fontů a vytvořil jsem si program `t1accent` [4], který k písmenkům přidával akcenty podle vzorových PostScriptových tahů generovaných z původních CSfontů METAPOSTem. Na mnoha místech jsem byl nucen přistoupit k mikrotypografickým kompromisům – v drobnostech se kresby některých znaků CSfontů z Type1 liší od svých originálních METAFONTových protějšků. Proto jsem distribuci Type1 CSfontů označil jako „alpha“ a do komentáře jsem dal důrazné varování, že tyto fonty je možné používat na vlastní riziko. Z toho důvodu jsem také ponechal implicitní konfiguraci programu `dvips` tak, aby program používal léty osvědčený výstup z METAFONTu, tedy bitmapy formátu `pk`. Type1 CSfonty byly původně konfigurovány výhradně pro pdfTEX.

Rozhodnutí ponechat `dvips` pracovat implicitně s bitmapami naráželo na problémy. Neustále dokola se uživaté ptali, jak je možné, že výstup z pdfTEXu je dobrý, ale při cestě `dvips` – `ps2pdf` dostávají roztřesená písmenka. Byl jsem uondán velmi častým odpovídáním na tuto otázku a vysvětlováním, jak použít `dvips` s jinou než výchozí konfigurací. Ani jsem se tedy nakonec nezlobil, když v roce 2001 autor teTEXu rozhodl, že bude CSfonty pro `dvips` implicitně konfigurovat ve verzi Type1. Asi ty mikrotypografické kompromisy ani tak moc nevadí, zatímco roztřesená písmenka v PDF a neznalost použití jiné než výchozí konfigurace `dvips` způsobovala oheň na střeše.

V současné době existují volně dostupné nástroje, jako například `textrace` opírající se o `autotrace`. Tyto nástroje umožní převést METAFONTový font do Type1 „obtahováním bitmap“ skoro automaticky. Vyzkoušel jsem to na CSfontech a s výsledkem jsem nebyl vůbec spokojen: výsledné `pfb` soubory byly asi pětkrát větší než ty moje „ručně“ vyrobené a z BaKoma fontů odvozené. Proto jsem zatím alpha verzi Type1 formátu CSfontů z roku 1988 neopustil.

Do budoucna bych velmi rád do CSfontů přidal znak euro a paragraf. Taková změna by byla zpětně kompatibilní, takže bych se jí nebránil. METAFONTové zdroje pro paragraf ověřené na všech CSfontech už několik let mám, ale nezveřejnil jsem je. METAFONTové zdroje znaku euro by se snadno daly převzít z jiného METAFONTového fontu. Největší potíž je ovšem v tom, že s uvedením nové verze CSfontů dnes nestačí zveřejnit jen METAFONTové zdroje a metriky, ale je třeba mít okamžitě s tím konzistentní Type1 varianty fontů. Do manuální práce na nové verzi Type1 varianty CSfontů se mi ale moc nechce. Je to nevděčná a rozsáhlá práce: `pfb` souborů je v balíčku 57 a každý je třeba disassemblovat, v editoru přidat nové znaky a znovu převést na `pfb`. Přitom s automatickými nástroji, jak jsem uvedl před chvílí, nejsem spokojen.

3 Podpora základních 35 PostScriptových fontů

Tato podpora je zahrnuta do balíčku `cspsfonts.tar.gz` a má méně bohatou historii než CSfonty. Balíček začal vznikat v září roku 1994. Tehdy jsem zjistil, že popisy kompozitů v `AFM` metrice pomocí řádků `CC` jsou správně převáděny programem `afm2tfm` na odpovídající kompozity ve vytvářeném virtuálním fontu. Problém byl jen v tom, že originální `AFM` metriky od Adobe neobsahovaly popisy všech kompozitů potřených pro český a slovenský jazyk. Z toho důvodu jsem si vytvořil program `a2ac` [5], který na základě přehledné tabulky kompozity do `AFM` metrik doplnil a současně doplnil kerningové informace pro nově vytvářené znaky. Za použití tohoto programu pak vznikla sada metrik a virtuálních fontů s písmenem `c` na začátku (např. `cptmr`).

V roce 1996 pak uveřejnil pan Wagner nové metriky generované stejným způsobem, ovšem opravil několik estetických nedostatků a navíc metriky nazval podle doporučení Karla Berryho (`8z` a `8t` na konci). Od té doby jsou v balíčku `cspsfonts.tar.gz` obsaženy metriky pana Wagnera. Česká a slovenská abeceda je v těchto metrikách kódována stejně, jako v CSfontech, tj. podle ISO-8859-2. Právě tomuto kódování odpovídá koncovka `8z`.

Konečně v roce 1999 jsem musel po konzultaci s Karlem Berrym metriky pro rodinu Courier přejmenovat z původního `*8t` na nynější `*8u`, protože názvy s `8t` na konci nám kolidovaly s názvy stejných fontů v kódování podle Corku. To je zatím poslední změna v tomto balíčku.

Do budoucna zvažuji rozšířit balíček `cspsfonts.tar` o další metriky a virtuální fonty `*.8z` k volně šířeným Type1 PostScriptovým fontům. První na řadě může být font Charter, kterým je vytvořen tento sborník. Neuvědomil jsem si, že se tento font stal volně šířeným – mám ho totiž už léta kopuený. Takže metriky kódování `*8z` už mám dávno hotové, jen zatím nebyly nikde zveřejněny.

4 Csplain

Csplain vznikl v roce 1992 jako jednoduché a minimální rozšíření Knuthova plainu používající Csfonty a akceptující 8bitový vstup. Jeho vytvoření bylo motivováno zařazením do emTEXové distribuce, která se připravovala k rozesílání členům CsTUGu.

Csplain se opíral a stále opírá o starší makra `hyphen.lan` a `plaina4.tex`, která už měl Olin Ulrych vytvořena dříve. Csplain v době svého vzniku načítal české vzory dělení, která vytvořil Láďa Lhotka heuristicky bez použití slovníků a programu `patgen`. Já jsem pro Csplain vytvořil `csplain.ini` a makro `csfonts.tex` umožňující při generování formátu číst přímo Knuthovo makro `plain.tex`, a přitom natáhnout místo CMfontů Csfonty.

V roce 1995 byly vzory dělení slov Ládi Lhotky vyměněny za nové české vzory dělení od Pavla Ševečka, který na to použil slovník a `patgen`. V rámci své firmy tyto vzory dělení prodává komerčním firmám pro potřeby DTP programů, jako byly Ventura, Pagemaker nebo Quark. Dnes jsou tyto vzory dělení také například ve Wordu. Aby Pavel Ševeček odlišil volně šířené vzory dělení pro CsTUG od komerčně šířených, volně šířené vzory dělení mírně modifikoval. Tvrdí se, že běžný uživatel nepozná rozdíl v kvalitě vzorů dělení komerčních a volně šířených. Ševečkovy vzory dělení slov jsou výrazně kvalitnější než původní Lhotkovy, a proto jsme u těchto vzorů dělení zůstali.

Opuštěním Lhotkových vzorů dělení došlo k poslední změně v Csplainu, která může způsobit zpětnou nekompatibilitu v českém dokumentu: tj. dokumenty vytvořené v Csplainu před rokem 1995 mohly dopadnout jinak než dnes, protože některá slova mohla být rozdělena jinak. Od této doby je Csplain fixován a stabilní podobně jako Knuthův plain.

Protože jsem autorem názvu Csplain, souborů `csplain.ini`, `csfonts.tex` a množství dokumentace k Csplainu a protože jej od jeho vzniku udržuji, rozhodl jsem se přísně dbát na zpětnou kompatibilitu. Změny do Csplainu dělám jen takové, které jsou opravdu nezbytné. To se stává jednou za několik let (viz historické poznámky v `csplain.ini`). Změny dělám tak, že pouze přidám další nejnutnější makra, ale stávající makra a jejich význam nechávám nezměněna. Uživatelům Csplainu ručím, že jejich dokumenty napsané v Csplainu a opírající se o neměnné fonty (např. Csfonty nebo base 35 PostScriptové fonty, metriky z CsTEXu), budou i v budoucnu Csplainem formátovány naprosto stejně, jako dnes.

Abych mohl takovou záruku uživatelům poskytnout, není Csplain zveřejněn pod GNU GPL, ale jedná se o licenci velmi podobnou Knuthově. Přesné znění licence je uvedeno na konci souboru `csplain.ini`. Zhruba řečeno, jedná se o „patent na název“. Csplain můžete svobodně distribuovat, používat a měnit, ale pokud jej změníte, nesmíte jej dále distribuovat pod názvem Csplain. Změny v Csplainu může dělat jen tzv. „současný administrátor CsTEXu“, což jsem zatím stále já. Pokud bych v budoucnu toto břímě někomu předal, pak určitě jen takovému člověku, který má na budoucnost Csplainu stejný názor, jako já.

Csplain i nadále považuji za minimální rozšíření Knuthova plainu a nikdy do něj nepřidám žádné složitější makro vylepšující uživatelský komfort (jako

například `eplain` nebo OFS). Zastávám názor, že uživatel plainu a Csplainu chce mít všechna makra pod svou vlastní kontrolou a raději si je udělá sám, než aby spoléhal na hotová, ale méně stabilní, řešení. Uživatel plainu/Csplainu si vytváří vlastní stále znovu používaná makra, která mu musí fungovat ve všech i budoucích verzích Csplainu. Proto se snažím Csplain pokud možno neměnit.

Lákavá je například změna definice uvozovek v Csplainu jednoduše na

```
\long\def\uv#1{\clqq#1\crqq}
```

aby fungoval automatický kerning s oběma stranami uvozovek. Tuto změnu ale nikdy v Csplainu neudělám, protože není zpětně kompatibilní se stávajícím řešením a v některých dokumentech by uvnitř `\uv` přestaly fungovat verbatim konstrukce. Raději budu psát do omrzení do dokumentace, že taková jednoduchá definice je asi lepší, než ta z Csplainu, a že si ji každý může zařadit do svých maker.

Změnu v Csplainu z `<Feb. 2000>` považuji za asi největší, kterou jsem byl ochoten udělat. Tehdy jsem zařadil do Csplainu alternativní kódování Cork, které samozřejmě není a nikdy nebude v Csplainu implicitní. Implicitní nadále zůstává kódování podle ISO-8859-2, tedy podle Csfontů. K přidání podpory Corku mě motivovala skutečnost, že se kolem mě pohybovalo mnoho uživatelů plainu, kteří rádi používají fonty v tomto alternativním kódování. Svým krokem jsem jim umožnil používat Csplain, takže si nemusejí vytvářet své vlastní formáty.

Jsem si vědom problému s touto změnou spojeném: do emTEXu (binárky `tex.exe` i `tex386.exe`) se ani po změně paměťových parametrů pomocí přepínačů nevejde pět vzorů dělení slov (angličtina, čeština v ISO-8859-2 a v Corku a slovenština v ISO-8859-2 a v Corku). V této distribuci je možno formát vygenerovat jen binárkou `htex386.exe`. Proto zůstává podpora kódování podle Corku v Csplainu jako nepovinná, tj. nelze očekávat, že ji všichni uživatelé CsTEXu budou mít. Také se dá tato podpora vypnout pomocí `\let\Cork=\relax` před zavedením souboru `csplain.ini`.

Možná už tušíte, proč jsem čekal až do roku 2000 s přidáním takového rozšíření. Usoudil jsem, že na prahu nového tisíciletí už snad emTEXovou distribuci moc lidí nepoužívá.

Pokud jde o budoucnost Csplainu, pak nepředpokládám žádné výrazné změny a vždy budu nekompromisně dodržovat zpětnou kompatibilitu. Jediná věc, která by mohla kompatibilitu ohrozit, je zavedení kvalitnějších slovenských vzorů dělení slov. Věřím, že slovenští kolegové raději přejdou na kvalitnější vzory dělení slov, než aby lpěli na absolutní zpětné kompatibilitě s ne příliš kvalitními vzory dělení. Otázka pouze zůstává, kdo nové slovenské vzory dělení udělá...

5 CsLaTeX

CsLaTeX vytvořil zhruba v roce 1992 Jiří Zlatuška. Od něj pochází myšlenka načtení vzorů dělení stejného jazyka v různých kódováních a předefinování LaTeXového makra `\DeclareFontEncoding`. Na své soukromé implementaci TEXu tehdy provozoval mimo jiné fonty kódované v KOI-8, takže přepínání

vnitřního kódování LATEXu si vlastně udělal pro svoje potřeby. Jiří Zlatuška je pravděpodobně také autorem maker `\splithyphens` a `\standardhyphens`, která zapínají a vypínají inteligenci znaku `-`. Veškerá makra napsal dobře dokumentovaná pro použití v systému `docstrip`. Svou práci zveřejnil pod licencí GPL mimo jiné podle jeho slov proto, že pokud to bude někoho zajímat, tak to může dále udržovat a zvelebovat podle svých vlastních představ. On sám se kvůli své zaneprázněnosti v jiné oblasti tímto problémem později zřejmě nezabýval.

Po schůzce tvůrců CSTEXu v roce 1993 převzal starost o CSLATEX podle dohody Zdeněk Wagner, který vytvořil definice kódování `IL2`. Vytvořil také pro CSLATEX definiční soubory `fd` jednak pro CSfonty a jednak pro PostScriptové fonty (balíček `cspsfonts.tar.gz`). V té době byl ještě CSLATEXu implementován pro verzi LATEXu 2.09. V emTEXové distribuci CSTEXu je stále tato implementace obsažena (`cslt209.zip` v adresáři `emtex`). Součástí dnešního CSTEXu (`cslatex.tar.gz` v adresáři `base`) už podpora této staré verze LATEXu není.

Ačkoli možnost načíst vzory dělení jednoho jazyka ve více kódováních dal CSLATEXu už Jiří Zlatuška, byla tato možnost až do roku 1999 implicitně v souboru `hyphen.cfg` vypnuta a CSLATEX bez editace tohoto souboru pracoval jen v kódování CSfontů. Důvod už jsem zmínil v kapitole o CSplainu: v emTEXové distribuci se pět vzorů dělení do paměti binárek `tex.exe` a `tex386.exe` prostě nevešlo.

Soubor `czech.sty` má asi podstatně delší historii než CSLATEX. Pochází z dílny Olina Ulricha. který se zřejmě inspiroval podobným stylovým souborem pro německý jazyk. Olin rovněž vytvořil makra `\csprimeson` a `\csprimesoff`. Zdeněk Wagner pak převzal Olinův stylový soubor a upravil jej pro provoz v CSLATEXu. Slovenskou část včetně vzorů dělení slov vytvořila Janka Chlebíková. Soubor `slovak.sty` je přesnou kopií souboru `czech.sty` s vyjímkou slovensky špecifických částí.

V duchu licence GPL převzal zhruba v roce 1997 údržbu CSLATEXu Jaroslav Šnajdr. Udělal několik úprav stylových souborů `czech.sty` a `slovak.sty` včetně přechodu na novou definici uvozovek, uvnitř jejichž argumentu nefungují verbatim konstrukce. Tím kuriózně způsobil, že CSLATEXem od této doby nejde bez chyb formátovat český překlad úvodu do LATEXu, který je pod názvem balíčku `csuvodlat.tar.gz` součástí dokumentace CSTEXu. Je to názorná ukázka toho, co může způsobit změna kódu, která nerespektuje zpětnou kompatibilitu. Pan Šnajdr rovněž napsal `html` dokumentaci k CSLATEXu, která popisuje instalaci CSLATEXu ze zdrojových souborů použitím docstripu. Použijete-li ale balíček `cslatex.tar.gz`, pak nemusíte docstrip aplikovat, protože vedle zdrojových souborů jsou tam už přítomny i všechny soubory, které vznikají po aplikaci `docstripu`.

Já osobně jsem o LATEX a tím pádem CSLATEX jevil od začátku malý zájem, protože osobně používám CSplain. V roce 1999 jsem nicméně přidal pár řádek maker do souboru `czhyphen.tex` tak, aby byl použitelný v babelizovaném LATEXu. Do té doby totiž tato větev LATEXu používala Lhotkovy vzory dělení, zatímco v CSLATEXu jsme už čtyři roky měli daleko kvalitnější Ševečkovy vzory dělení. Tyto novější vzory dělení jsou napsány za použití TEXových sekvencí,

což je sice nezávislé na kódování češtiny, ale balíček Babel to implicitně nedokáže strávit a očekává vzory dělení v kódování `T1`. Upravený soubor jsem nazval Babelovsky: `czhyph.tex`, zatímco v CSTEXu zůstává původní soubor `czhyphen.tex`.

Na výborové schůzi v roce 1999 jsem dostal za úkol prověřit možnost spojení babelizovaného LATEXu s CSLATEXem. Neustálé dotazy začínajících uživatelů, kteří si pletou tyto dva LATEXy, nás utvrzují v tom, že by se pro sloučení mělo něco udělat.

Analyzoval jsem proto makra Babelu a udělal návrh na možné zapracování funkcionality CSLATEXu do Babelu. Domnívám se, že CSLATEX klidně může přestat existovat, ale babelizovaný LATEX musí bezpodmínečně převzít všechny vlastnosti CSLATEXu tak, aby dokumenty dříve zpracovávané CSLATEXem byly naprosto stejně a bez jediné úpravy zpracované novým babelizovaným LATEXem. Kvůli tomuto požadavku musí babelizovaný LATEX umět načítat vzory dělení stejného jazyka ve více kódováních, jako to nyní dělá CSLATEX. Dospěl jsem k závěru, že čistým řešením tohoto problému je jedině zásah do jádra LATEXu samotného, aby dokázal při změně kódování fontů přepnout automaticky i vzory dělení. Zlatuška kvůli tomu předefinoval makro jádra LATEXu `\DeclareFontEncoding`. Tato záplata, či jinak řečeno odmítnutí původního kódu tohoto makra, je na úrovni Babelu podle mého názoru velmi nečisté řešení. Skutečnost, že přepínání vzorů dělení při přepnutí kódování fontů LATEXové jádro neřeší, považuji za chybu LATEXu. V roce 1999 jsem tedy požádal LATEX-team, aby zapracoval změnu v duchu Zlatuškova návrhu do LATEXového jádra. Můj návrh nebyl LATEX-teamem akceptován. Za těchto okolností nejsem schopen zapracovat funkcionalitu CSLATEXu do Babelu, protože to prostě nejde. Uživatelé LATEXu se budou muset nadále potýkat s tím, že jejich oblíbený formát trpí určitou schizofremií.

Společně se sloučením CSLATEXu s Babelem jsem připravoval zásadní revizi stylů `czech.sty` a `slovak.sty` – v podstatě jsem měl v úmyslu jejich totální přepsání. Tyto stylové soubory obsahují množství reliktů z dob minulých, plno zcela nepoužívaných větví ve složitém větvení pomocí `\if` a stávají se totálně nepřehlednými. Protože ale ke sloučení CSLATEXu s Babelem nakonec nedošlo, upustil jsem zatím od plánu pracovat na těch stylových souborech. Není ale vyloučeno, že k tomu dojde v budoucnosti. V takovém případě počítám s tím, že makra `\splithyphens` a `\standardhyphens` přesunu z formátu do stylového souboru, kam přirozeně patří. Dokumenty, které tato makra používají, a přitom nemají v záhlaví `\usepackage{czech}` ani `\usepackage{slovak}`, pak nebudou fungovat. Předpokládám, že takových dokumentů není mnoho, protože CSLATEX a stylové soubory jsou většinou používány současně.

Protože pan Šnajdr se přestal CSLATEXem zabývat, byl jsem nucen v roce 2002 zanést do stylových souborů jednu opravu podle požadavku pana Kubena osobně. Neznamená to ale, že bych se ujal iniciativy nad CSLATEXem. Jak jsem už vysvětlil, jsem ochoten převzít iniciativu jen tehdy, když bude LATEXové jádro umět přepínat mezi různě kódovanými vzory dělení stejného jazyka. Přitom

členové LATEX-teamu jsou toho názoru, že to možná bude zapracováno až do LATEXu 3.

6 Báječní muži na počítacích strojích

Vzpomínám-li na přelom 80. a 90. let minulého století, pak z hlediska rychlosti rozvoje informačních technologií mluvím vlastně o počítačovém pravěku. Právě v té době začal vznikat CSTEX. Tak jako „báječní muži na létajících strojích“ položili základy dnešní letecké dopravy, stejně bychom o lidech, kteří se v té době motali kolem počítačů mohli říkat „báječní muži na počítacích strojích“. Zvláště jsme pak vděčni těm, po kterých kromě nadšení a okouzlení nad novou progresívní technikou zůstalo taky něco užitečného. Podíváme-li se na historii CSTEXu, můžeme směle prohlásit, že tam najdeme takových báječných mužů celou řadu (ve svém článku jsem zmínil i jednu ženu). Je možné, že jsem na někoho zapomněl, protože sám jsem se dostal k TEXu až relativně pozdě: v roce 1992.

Reference

1. Výchozí adresář CSTEXu je na `ftp://math.feld.cvut.cz/pub/cstex` a kopie se nalézá na `ftp://ftp.cstug.cz/pub/local/cstex`. V dalších odkazech bude toto místo označováno jako `cstex:/`.
2. Petr Olšák. *Pár poznámek k novému CSTEXu*. Volně šířená dokumentace k CSTEXu, viz `cstex:/parpozn.tex` nebo `cstex:/parpozn.pdf`. 1996.
3. Petr Olšák. *Manuál k CSTEXu*. Dokumentace k CSTEXu, `cstex:/cstexman.tex` nebo `cstex:/cstexman.pdf`. 2002.
4. `ftp://math.feld.cvut.cz/pub/olsak/t1accent/`.
5. `ftp://math.feld.cvut.cz/pub/olsak/a2ac/`.
6. Ladislav Lhotka. *České dělení pro TEX*. Zpravodaj Československého sdružení uživatelů TEXu, 4/1991, strany 8–9.
7. Petr Olšák. *Program MNU: Konfigurovatelné menu pro spouštění aplikací pod DOSem*. Zpravodaj Československého sdružení uživatelů TEXu, 3/1992, strany 141–148.

XSL FO a jeho open-source implementace

Jiří Kosek

LISP, VŠE Praha
Email: jirka@kosek.cz

Abstrakt: XSL FO je jazyk, který umožňuje kvalitní formátování XML dokumentů zejména pro účely tisku. Příspěvek posluchače seznámí s principy XSL FO a úrovní jeho podpory v dostupných open-source implementacích. Budou zmíněny i oblasti, kde je použití XSL jednodušší než použití TEXu i scénáře, na které XSL na rozdíl od TEXu nestačí.

Formát XML se dnes pro ukládání a přenos dat používá stále častěji. V XML na svém počítači naleznete dokumentaci (DocBook), konfigurační soubory (např. skripty ant, projekty novějších vývojových prostředí) a dost možná i obrázky (SVG). Ve většině případů však dokument XML popisuje pouze význam uložených dat, ale nedefinuje, jak se mají zobrazit. Potřebujeme-li si tedy dokument prohlédnout jinak než jako zdrojový kód v XML, musíme mít možnost popsat konverzi XML do vizuální podoby. K tomuto účelu se používají stylové jazyky, které umožňují popsat, jak se mají jednotlivé elementy XML dokumentu zobrazovat – jakým písmem, jakou velikostí, jak zarovnané apod. Stylových jazyků, které můžeme použít, existuje hned několik – FOSI, DSSSL, CSS a XSL. Právě poslední z nich byl vytvořen speciálně pro potřeby XML.

1 Princip XSL

Na jazyku pro formátování XML dokumentů se začalo pracovat v podstatě ve stejné době jako na samotném jazyce XML. Poměrně brzy se přišlo na to, že pořádný stylový jazyk se skládá ze dvou částí. Stylový jazyk musí umět jednotlivým částem dokumentu přiřadit jejich formátovací vlastnosti. Před tímto přiřazením je však v mnoha případech potřeba provést ještě transformaci vstupního dokumentu. Transformace se použije pro takové operace jako vygenerování obsahu či rejstříku, očíslování kapitol a obrázků nebo pro sečtení položek na faktuře – záleží na našich potřebách a na vstupním dokumentu.

Tento přístup reflektuje i samotný jazyk XSL. Skládá se ze dvou samostatných částí – transformačního jazyka XSLT a tzv. formátovacích objektů (FO). Podobně jako XML i XSL je standardizováno v rámci W3C. XSLT bylo jako doporučení přijato v roce 1999 a formátovací objekty až o dva roky později (2001).

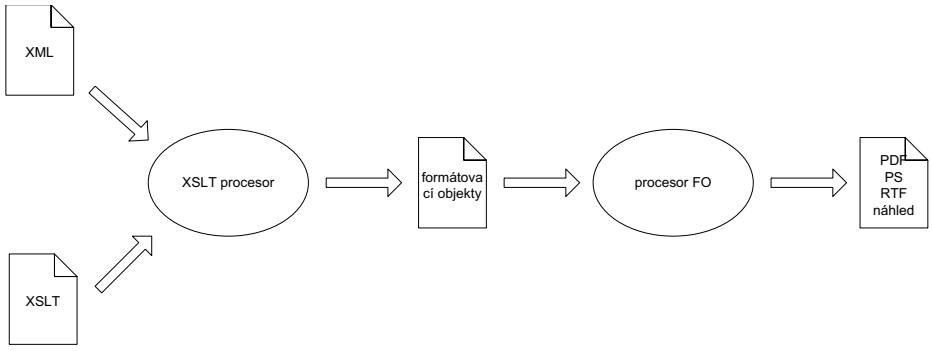
XSLT styl umožňuje popsat transformaci XML dokumentu do HTML, XML s jinou strukturou, případně do obyčejného textového souboru. Transformace je přitom popsána sadou šablon, které definují převod jednotlivých částí XML dokumentu, jež jsou identifikovány pomocí XPath výrazu. Pokud chceme nějaký

Jan Kasprzak, Petr Sojka (editoři): SLT 2002 – sborník semináře o Linuxu a TEXu, str. 105–113, 2002.

XML dokument transformovat pomocí XSLT, musíme mít k dispozici tzv. XSLT procesor. Mezi nejpoužívanější dnes patří asi Saxon, xsltproc a Xalan, ale existují mnohé další.[1]

Formátovací objekty – druhá část XSL – jsou jen jakýsi abstraktní slovník, který umí velmi dobře definovat vizuální vzhled dokumentu. FO umožňují definovat layout stránek a pak popsat formátování textu a dalších objektů, které se mají do těchto stránek naformátovat. Obsah stránek je přitom definován právě pomocí formátovacích objektů, které zastupují bloky textu, buňky tabulky nebo třeba obrázek. Každý objekt může mít několik desítek vlastností, které definují použité písmo, velikost okrajů okolo objektu, barvu textu i pozadí, vzhled rámečku, zakazují nebo povolují zlom stránky atd.

A jak formátovací objekty souvisí s formátováním XML? Celý trik spočívá v tom, že formátovací objekty se zapisují jako XML dokument, kde jednotlivým objektům odpovídají elementy a vlastnostem atributy. Díky tomu můžeme pro převedení XML do formátovacích objektů použít XSLT. Transformace převede sémantické značkování na formátovací objekty definující vzhled. Kromě toho se v tomto okamžiku může například vygenerovat obsah apod. Výsledný soubor FO se pak předá tzv. procesoru formátovacích objektů, který se postará o konečné zalomení formátovacích objektů do definovaných rozměrů stránky a výsledek zobrazí nebo uloží do nějaké vhodného formátu jako PDF nebo PostScript.



Obrázek 1. Princip zpracování XML pomocí XSL

Abychom si vše přiblížili na jednoduchém příkladě, předpokládejme, že máme následující XML dokument:

```
<odstavec>K tomuto účelu se používají <pojem>stylové
jazyky</pojem>, které umožňují popsat, jak se mají
jednotlivé elementy XML dokumentu zobrazovat.</odstavec>
```

A my jej chceme pomocí XSL převést do tištěné podoby. Musíme si proto vytvořit XSLT styl, který bude generovat formátovací objekty. Dejme tomu, že

[1] `http://www.xmlsoftware.com/xslt.html`

chceme, aby se obsah elementu `odstavec` zobrazil jako samostatný odstavec, písmem o velikosti 12 bodů, zarovnaný do bloku, s odstavcovou zarážkou 18 bodů. Označené pojmy chceme zobrazit kurzívou. Do XSLT stylu proto přidáme dvě jednoduché šablony, které XML převedou na elementy odpovídající formátovacím objektům.

```
<xsl:template match="odstavec">
  <fo:block font-size="12pt" text-indent="18pt"
   text-align="justify">
    <xsl:apply-templates/>
  </fo:block>
</xsl:template>

<xsl:template match="pojem">
  <fo:inline font-style="italic">
    <xsl:apply-templates/>
  </fo:inline>
</xsl:template>
```

Po zpracování dokumentu a stylu XSLT procesorem dostaneme dokument obsahující formátovací objekty:

```
<fo:block font-size="12pt" text-indent="18pt"
text-align="justify">K tomuto účelu se používají
<fo:inline font-style="italic">stylové jazyky</fo:inline>,
které umožňují popsat, jak se mají jednotlivé elementy XML
dokumentu zobrazovat.</fo:block>
```

Vidíme, že během transformace se z XML dokumentu nesoucího nějakou sémantickou informaci stal jiný dokument, který již obsahuje jen informace o formátování. Procesor FO z něj dokáže vytvořit formátovaný dokument. Nejprve z něj načte elementy jako formátovací objekty, pak u všech formátovacích objektů doplní nespecifikované vlastnosti hodnotou zděděnou od rodičovského formátovacího objektu nebo implicitní hodnotou, vyhodnotí hodnoty výrazů apod. Výsledek se pak zalomí do oblastí vyhovujících všem omezením a práce je hotova.

2 Možnosti formátovacích objektů

Zatím jsme v ukázkách viděli jen dva formátovací objekty – `fo:block` a `fo:inline`, které patří mezi dva nejpoužívanější. První z nich umožňuje vytvářet samostatné bloky textu, které se zalamují do odstavce. Druhý pak umožňuje v části takového bloku změnit vybrané formátovací vlastnosti – např. změnit použitý řez písma.

Formátovacích objektů samozřejmě existuje mnohem více. Každý soubor FO na svém začátku nejdříve definuje předlohy stran – můžeme definovat rozměry

stránky, velikost jejích okrajů apod. Těchto definic může být více a můžeme je sdružovat do předloh sekvencí stránek – tím lze snadno dosáhnout takových efektů, jako odlišný vzhled první stránky dokumentu či kapitoly nebo odlišnou velikost okrajů a záhlaví na sudých a lichých stránkách.

Za definicí předloh stránek pak následují sekvence stránek (`fo:page-sequence`). Pro každou sekvenci stránek můžeme určit, do jaké předlohy sekvence stránek se má sázet její obsah, jde definovat obsah pro záhlaví, zápatí, levý a pravý okraj. Ve formátovacím objektu `fo:flow` jsou pak obsaženy „běžné“ objekty jako právě `fo:block`, které již generují samotný obsah stránek.

Pro určité druhy objektů, které se mají ve výstupu objevit, nám již použití samotného `fo:block` nestačí. Pro tyto případy jsou k dispozici účelově zaměřené objekty.

K dispozici máme řadu objektů pro tvorbu tabulek. Model tabulek je podobný jako v HTML – tabulka (`fo:table`, `fo:table-and-caption`) se skládá z buněk (`fo:table-cell`), které jsou součástí řádky (`fo:table-row`) a řádka je součástí nějaké skupiny v tabulce (`fo:table-body`, `fo:table-footer`, `fo:table-header`). Pomocí vlastností pak můžeme nastavovat takové věci jako slučování buněk, rámečky okolo buněk a celé tabulky, barvy, pozadí, zarovnání obsahu buněk apod.

Samostatně existují i objekty pro seznamy. Celý seznam je uložen v `fo:list-block`. Každá položka seznamu je ohraničena pomocí `fo:list-item`. Položka seznamu pak má své návěstí (jako je odrážka, číslo, text) `fo:list-item-label` a tělo `fo:list-item-body`.

Obrázky se většinou vkládají jako odkaz na externí soubor pomocí `fo:external-graphic`. Podporované formáty záleží na konkrétním procesoru FO. Pokud pro obrázky používáme nějaký formát založený na XML (např. SVG), můžeme je vložit přímo mezi formátovací objekty jako obsah elementu `fo:instream-foreign-object`. Při zpracování pak opět záleží na schopnostech konkrétního procesoru.

Celá řada objektů slouží ke vložení prvků, které se objeví na jiném místě, než jsou vloženy. Jedná se především o plovoucí objekty (`fo:flow`), jež umožňují obtékání obrázků nebo jiných objektů textem zleva či zprava, případně ke vložení obrázku či tabulky na první vhodné místo v dokumentu. Do podobné kategorie spadají i poznámky pod čarou (`fo:footnote`, `fo:footnote-body`).

Existuje i několik objektů, které jsou vyhodnocovány a nahrazovány konkrétní hodnotou až v okamžiku formátování, protože ve fázi XSLT transformace nemáme dostatek informací. Typickým zástupcem je objekt pro vkládání aktuálního čísla strany `fo:page-number`. Při generování obsahu nebo křížových odkazů se naopak uplatní objekt `fo:page-number-citation`, který se nahradí číslem strany, na níž se vyskytuje formátovací objekt určený pomocí svého identifikátoru. Umožňuje-li to výstupní médium (např. PDF), můžeme z obsahu nebo křížových odkazů udělat jednoduše hypertextové odkazy pomocí `fo:basic-link`.

V mnoha dokumentech chceme mít v záhlaví/zápatí kromě čísla strany např. název kapitoly nebo podkapitoly. K dosažení tohoto efektu můžeme použít

objekty `fo:marker` a `fo:retrieve-marker`. „Dynamicky“ se chová i objekt `fo:leader`, který vyplní volný prostor zadaným textem – lze jej použít například pro oddělení názvu kapitoly a čísla strany v obsahu tečkami.

Výčet objektů nebyl úplně kompletní, ale v praxi si s nimi bohatě vystačíme. V zasadě můžeme říci, že FO lze bez problémů použít pro formátování dokumentů obsahujících hladký text včetně obrázků, tabulek, obsahu, křížových odkazů, poznámek pod čarou, plovoucích záhlaví. Problémem není ani vícesloupcová sazba. Je to celkem pochopitelné, protože FO vycházejí z reálných požadavků na přípravu dokumentů.

3 Porovnání FO s TeXem

Z čistě funkčního hlediska nabízí XSL (resp. kombinace XSLT + FO) podobné možnosti jako TEX, tj. zcela automatizovanou sazbu dokumentů. Jsou však věci, které lze v XSL udělat mnohem snáze než v TEXu, v některých ohledech zůstává však TEX nepřekonán.

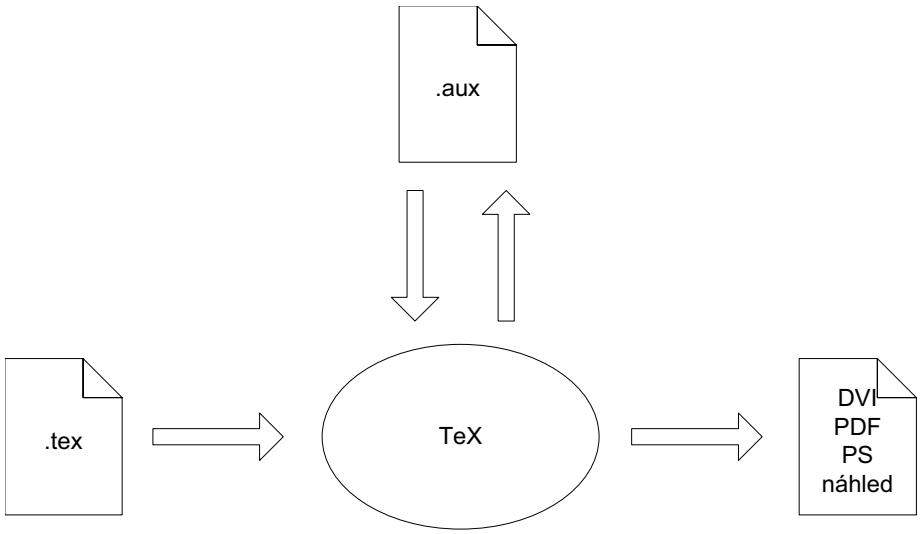
Jistou výhodou XSL je, že dokument je při zpracování celý v paměti. Během XSLT transformace se tak můžeme kdykoliv dotazovat na ostatní části dokumentu, můžeme velmi jednoduše generovat obsah, rejstříky, křížové odkazy apod. Držet v paměti celé dokumenty není pro současné počítače příliš velký problém, pokud jejich délka nepřesahuje jednotky tisíců stran.

V tomto ohledu je TEX lépe připraven na zpracování opravdu dlouhých dokumentů. TEX načítá vstupní dokument sekvenčně a průběžně jej sází. Jednoprůchodové zpracování neumožňuje vygenerování obsahu (pokud má být na začátku), křížových odkazů a rejstříku. V TEXu se to řeší pomocnými soubory a víceprůchodovým zpracováním. Při prvním běhu se do pomocného souboru zapíší např. názvy a čísla stran jednotlivých kapitol. V druhém běhu se pak tento soubor načte a vysází jako obsah. Pro některé úkoly, jako je například generování rejstříku, se pak pomocné soubory musí ještě zpracovat specializovanými programy.

Díky možnosti dotazování nad XML dokumentem, jsou některé operace v XSLT opravdu snadné. Např. pro vygenerování obsahu stačí použít velmi jednoduchý kód, který na začátku dokumentu v cyklu zpracuje všechny kapitoly. Jednoduchost ocení každý, kdo se pokoušel vytvořit obsah dokumentu v plain-TEXu.[2]

```
<xsl:template match="/">
  <!-- Generování předloh stránek -->
  <fo:block break-before="page">
    <fo:block font-size="200%">Obsah</fo:block>
    <xsl:for-each select="//kapitola">
      <fo:block>
        <xsl:value-of select="nazev"/>
        <fo:leader leader-pattern="dots"
```

[2] V LATEXu je potřebný kód již hotový a stačí použít makro \tableofcontents.



Obrázek 2. Princip zpracování dokumentu TEXem

```
                leader-length.minimum="10pt"
                leader-length.optimum="20pt"
                leader-length.maximum="100%"/>
        <fo:page-number-citation ref-id="{generate-id()}"/>
      </fo:block>
    </xsl:for-each>
  </fo:block>
  <!-- Zpracování zbytku dokumentu -->
</xsl:template>

<xsl:template match="kapitola">
  <fo:block id="{generate-id()}" break-before="page">
    <xsl:apply-templates/>
  </fo:block>
</xsl:template>
```

Jedinečnou vlastností TEXu je jednotka glue, která umožňuje definování mezer, které se mohou podle potřeby zmenšit či zvětšit. Tuto možnost máme i v FO. U vlastností určujících mezery můžeme ve skutečnosti určit tři hodnoty – optimální, minimální a maximální. Např.:

```
<fo:block space-before.optimum="6pt"
          space-before.minimum="3pt"
          space-before.maximum="10pt">
  ...
</fo:block>
```

Oproti TEXu můžeme navíc u mezer nastavovat prioritu. Když se pak za sebou setkají dvě mezery, ve výsledku bude jen ta s větší prioritou. Tím lze velice elegantně řešit výjimečné případy.

Existují však situace, na které XSL nestačí. Tím, že fáze transformace a formátování jsou oddělené, nelze výsledný dokument ovlivňovat podle toho, jak se vysázela nějaká jeho předchozí část. Typickým případem je obtékání nepravidelných tvarů. TEX umožňuje vysázet kus textu a pak jej rozebrat a jednotlivé řádky různě posunout.

Dalším případem je rejstřík. Není problém jej vygenerovat v XSLT. Ve výsledném souboru FO pak pro heslo dostaneme čárkami oddělený seznam objektů `fo:page-number-citation`, které budou ukazovat na jednotlivé výskyty rejstříkového hesla. Teprve při formátování se zjistí, na kterých stranách se hesla vyskytují. Je možné, že na jedné stránce budou dva výskyty a v rejstříku nám zůstane duplicitní číslo strany. S tím však nic neuděláme, protože duplicity lze odstraňovat pouze ve fázi transformace, kdy však ještě nevíme, jak dopadne sazba.

Řešení tohoto problému existují, ale nejsou moc elegantní nebo přenositelná. Některé procesory FO podporují rozšiřující formátovací objekty. Např. XSL Formatter má objekt `axf:suppress-duplicate-page-number`, který umí ze svého obsahu odstranit duplicitní hodnoty v čárkami odděleném seznamu. XEP má formátovací objekt, který umí sám vytvořit seznam stránek s výskytem hesla. Druhou možností, velmi ošklivou, ale účinnou, je vygenerování rejstříku jako textu včetně XML značkování. Na konci našeho vysázeného dokumentu tak dostaneme fragment XML dokumentu, kde jsou označena jednotlivá hesla a stránky, na kterých se vyskytují (tyto stránky doplnil FO procesor). Vhodným nástrojem konec PDF dokumentu převedeme na čistý text, získáme tak XML dokument s obsahem rejstříku včetně čísel stran. Při druhém průchodu pak podle tohoto dokumentu vygenerujeme skutečný rejstřík. Čísla stran budou souhlasit, protože rejstřík je obvykle až na konci a tudíž nám neposune číslování stran.

Dá se očekávat, že na základě požadavků praxe přijdou jednotlivé implementace s rozšiřujícími objekty, které umožní tyto běžné problémy obejít. Nejúspěšnější rozšíření se velmi pravděpodobně stanou přímo součástí nějaké další verze XSL FO. Už dnes téměř všechny procesory obsahují speciální elementy pro vytvoření záložek při výstupu do PDF.

Pro většinu uživatelů může být použití XSL jednodušší než TEX. Vše je postaveno na syntaxi XML, která je dnes používaná v mnoha dalších aplikacích. Vlastnosti jako použité písmo, barva, způsob zarovnání apod. se jen deklarativně nastavují, což je většinou jednodušší než volání TEXových maker. Navíc je většina vlastností převzata z kaskádových stylů (CSS), které dnes umí mnoho lidí. XSL také přímo podporuje možnosti výstupních médií jako je barva a odkazy. V TEXu jsou tyto možnosti také dostupné, ale jsou podporovány jen některými výstupními formáty – typicky PDF nebo PostScript – a do dokumentů se vkládají poněkud nesystémově pomocí primitivu `\special`.

Mnoho výhod také přináší přímo ukládání primárních dokumentů do XML. Kromě tištěného výstupu můžeme velmi snadno generovat i další formáty jako

je HTML, RTF apod. S dokumenty lze pracovat jako s databází, protože XSLT umožňuje data filtrovat, řadit a seskupovat. Podobného efektu můžeme sice dosáhnout pomocí šikovných TEXových maker, ale ve většině případů to bude neskonale pracnější. Abych však byl upřímný, i TEX má své výhody. Integrace makrojazyka přímo se sázecím jádrem umožňuje dělat věci, o kterých se nám v XSL ani nezdá. Navíc je TEX mnohem rychlejší a poradí si i s velmi dlouhými dokumenty.

4 Úroveň podpory FO v open-source implementacích

V současné době existují tři použitelné open-source implementace XSL FO – FOP, PassiveTEX a jfor. Na mnoho úkolů je lze použít, nicméně zdaleka nepodporují standard XSL FO tak dobře jako komerční implementace XEP[3] a XSL Formatter[4]. Vzhledem k tomu, že standard je starý jen něco málo přes rok, v současné době není žádná implementace úplná. Je to dáno zejména tím, že formátovací model FO je poměrně komplexní a podporuje i jazyky, které se zapisují zprava doleva, i zezdola nahoru.

4.1 FOP

FOP[5] je implementace XSL FO napsaná v Javě. Dokument FO umí převést do PDF, PostScriptu, PCL a některých dalších formátů. Umí zobrazit i přímo jeho náhled na obrazovce. Obsahuje i javové rozhraní, díky němuž lze FOP integrovat snadno do dalších aplikací.

Asi největším problémem FOPu je špatně fungující podpora vlastnosti `keep-with-next`. Nemůžeme proto specifikovat podmínku, že některé bloky textu se nesmí rozdělit. Na výstupu můžeme dostat takové ohavnosti, jako nadpis na konci stránky. Nejsou podporovány plovoucí objekty (`fo:float`), u tabulek musí být šířky sloupců nastaveny explicitně a nelze používat relativní šířku sloupců (jednotka %).

Na druhou stranu má FOP dvě velmi užitečná rozšíření. Umí do dokumentu vkládat obrázky v SVG, které mohou být buď externí, nebo přímo vložené v FO. Při výstupu do PDF můžeme generovat záložky. V Javě si můžeme napsat i vlastní rozšíření.

4.2 PassiveTeX

PassiveTEX[6] je mezi implementacemi FO výjimkou. Většina ostatních používá vlastní formátovací engine. PassiveTEX je makro pro TEX, které rovnou načítá dokument FO a sází jej. Pro načítání XML se používá XML parser napsaný v TEXu –

[3] `http://www.renderx.com/FO2PDF.html`
[4] `http://www.antennahouse.com/xslformatter.html`
[5] `http://xml.apache.org/fop/index.html`
[6] `http://www.tei-c.org.uk/Software/passivetex/`

xmltex. PassiveTEX je nutné spustit opakovaně pro správné vygenerování obsahu a křížových odkazů.

PassiveTEX produkuje kvalitní výstup daný použitím TEXu. Nicméně už z principu nemůže podporovat všechny funkce FO. V praxi dnes nejvíce vadí nedokonalé zpracování tabulek a téměř nulová podpora pro relativní délkové jednotky a výrazy uvnitř vlastností.

PassiveTEX umožňuje generování záložek do PDF a podporuje zobrazování matematických výrazů zapsaných v MathML.

4.3 jfor

Tento procesor[7] umí převádět FO do formátu RTF. Podporuje jen poměrně malou část FO, ale může se hodit v případech, kdy potřebujeme z XML udělat RTF dokument.

5 Závěr

Dávkové formátování prožívá s rozšiřováním XML svoji renesanci. Primárním formátovacím nástrojem je přitom XSL, které umožňuje většinu požadavků vyřešit jednodušeji než v klasickém TEXu. Volně dostupné implementace XSL FO jsou již v praxi použitelné, ale zdaleka nejsou úplné.

Reference

1. Sharon Adler – Anders Berglund – Jeff Caruso – Stephen Deach – Paul Grosso – Eduardo Gutentag – Alex Milowski – Scott Pernell – Jeremy Richman – Steve Zilles: *Extensible Stylesheet Language (XSL) – Version 1.0.* W3C, 2001.
   URL: `http://www.w3.org/TR/xsl`
2. James Clark: *XSL Transformations (XSLT) Version 1.0.* W3C, 1999.
   URL: `http://www.w3.org/TR/xslt`

[7] `http://www.jfor.org/`

Část II

Linux

"The Linux OS gives consumers choice over the technology that comes with their computers at the operating system level. Does it require a whole new level of responsibility and an experience on the part of user? Certainly.
Will that user prefer to go back to the old model of being forced to trust his proprietary binary-only OS supplier once he has experienced the choice and freedom of the new model? Not likely."

Linus Torvalds, *The Linux Edge*
in Open Sources, Voices from the Open Source Revolution

Judy – nekonformní pohled na datové struktury

Štěpán Kasal

Matematický ústav AV ČR
Email: kasal@math.cas.cz

Abstrakt: Judy je abstraktní datová struktura, která se důsledně snaží minimalizovat výpadky keše na CPU. Díky tomu obecně překonává hashing i různé varianty vyvážených stromů.
Po třech letech vývoje je implementace Judy zcela stabilní, vhodná pro vážné použití. Na druhé straně byla uvolněna pod LGPL až v červnu 2002, je to tedy skvělá příležitost pro teoretičtěji zaměřené vývojáře svobodného software.

Klíčová slova: Judy, abstraktní datové struktury, asociativní pole, číslicové stromy, hashing, řídká pole, vyvážené stromy

1 Úvod

1.1 Otázka

Častým problémem v programování je nutnost zapamatovat si nějakou relaci, tedy pamatovat si hodnoty přiřazené k určitým indexům (klíčům).

Základním řešením tohoto problému je samozřejmě pole. Jeho nevýhodou je však skutečnost, že jeho velikost odpovídá předem zvolenému rozsahu, i když je populace výrazně menší.

Termín *rozsah* (anglicky expanse) označuje množinu možných indexů, *populace* (anglicky population) je počet skutečně použitých indexů. Pokud je populace nezanedbatelně menší než velikost rozsahu, mluvíme o *řídkých polích* (anglicky sparse arrays). Za speciální případ řídkých polí lze považovat i asociativní pole, tj. pole, jejichž indexy jsou řetězce.

1.2 Tradiční odpovědi

Velká skupina řešení problému implementace řídkých polí jsou různé *vyvážené stromy*. „Vyváženost“ stromu lze charakterizovat například tak, že v každé úrovni se pole rozdělí na dvě nebo více částí, a to tak, aby jednotlivé části měly přibližně stejnou populaci.

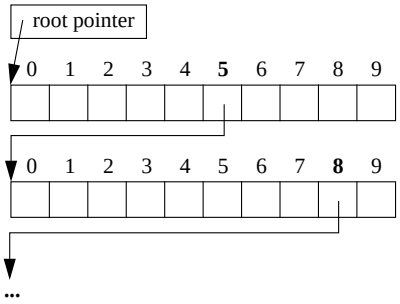
Další běžně používaný postup je *hashing*. Lze říci, že hashing je způsob, jak zvětšit hustotu pole a tedy převést řídká pole na obyčejná pole. Achillovou patou je ovšem zpracování synonym, tedy ošetření případu, kdy hashovací funkce vrátí pro různé indexy tutéž hodnotu. Nejčastěji bývá použit pouhý seznam,

Jan Kasprzak, Petr Sojka (editoři): SLT 2002 – sborník semináře o Linuxu a TEXu, str. 117–131, 2002.

takže je nutno synonyma probírat lineárně. Pokud pak populace přesáhne určitý práh, výkonnost celé datové struktury rapidně klesá. Jako obrana proti tomuto jevu se používá *adaptivní hashing*, kdy se při nárůstu populace zvětší velikost hashovacího pole. Při každém takovém zvětšení pole je však nutno projít všechny uložené prvky a umístit je do nového pole, což snižuje efektivitu této metody.

1.3 Číslicové stromy – slepá ulička?

Číslicový strom (anglicky digital tree), někdy též nazývaný *trie*, reprezentuje zcela odlišný přístup, kdy se pole na každé úrovni dělí na části nikoliv podle populace, ale podle rozsahu (viz obrázek 1).



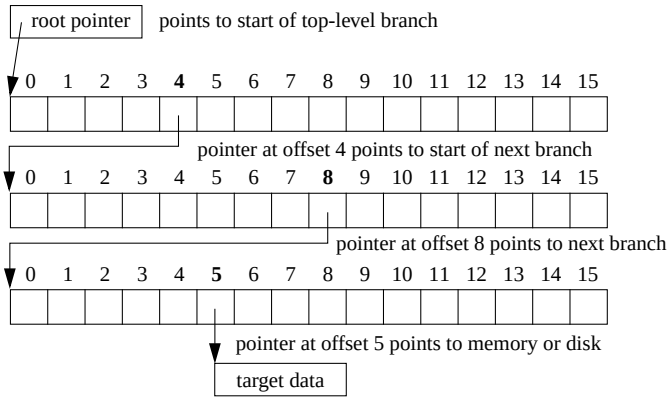
Obrázek 1. Dekadický číslicový strom

Je-li např. rozsah možných indexů 000–999, bude mít strom hloubku 3, přičemž na první úrovni se bude větvit podle první číslice, na druhé podle druhé číslice, na třetí úrovni se již bude nacházet pole deseti přiřazených hodnot.

Konstrukce číslicového stromu však nemusí vycházet z dekadického zápisu čísel, lze použít zápis v libovolné číselné soustavě. V praxi je samozřejmě výhodné použít soustavu, jejíž základ je mocninou dvojky, např. soustavu hexadecimální (obrázek 2).

Při práci s číslicovým stromem lze říci, že při přechodu na každou další úroveň se dozvídáme (dekódujeme) vždy další číslici indexu. Na nejnižší úrovni je tudíž již celý index znám a není nutno jej ukládat, stačí pouze uložit hodnotu, která je tomuto indexu přiřazena. To je jistá výhoda oproti výše uvedeným stromům založeným na rovnoměrném rozdělení populace i oproti hashingu. Další příjemnou vlastností číslicových stromů je jistá stabilita: přidání či odebrání prvku je vždy „lokální“ operace, protože není nutné provádět vyvažování.

Číslicové stromy však mají dvě obrovské nevýhody. Za prvé mají neúnosné paměťové nároky. Druhou nevýhodou je to, že rychlost přístupu závisí pouze na rozsahu, nikoliv na populaci, a tudíž zejména při nízké populaci je přístup neúměrně pomalý. Rychlost přístupu by bylo možné zlepšit zvýšením základu číselné soustavy, ale současně by se zvýšila paměťová náročnost, takže celkový efekt by byl negativní. (Přesněji řečeno, je-li základ soustavy `b`, je přístupová



Obrázek 2. Hexadecimální číslicový strom

doba neřímo úměrná $\log b$, ale paměťové nároky jsou téměř lineárně závislé na b.)

1.4 Nové pohledy

Doug Baskins, hlavní autor Judy, znovuobjevil číslicové stromy. Zvolil číselnou soustavu o neobvykle velkém základu: 256. Tak je v každé úrovni zakódován jeden byte. Každý uzel v takovém stromu by tedy měl obsahovat 256 ukazatelů, vedoucích o jednu úroveň níže. Judy však takovýto uzel velice často komprimuje, takže pokud je z tohoto pole podstatná část prvků (větví) neobsazena, zabírá takto zakomprimovaný uzel méně místa.

Číselný strom o základu 256, který kóduje jedno slovo (celočíselnou proměnnou typu `int`, nebo ukazatel), musí mít na 32-bitovém počítači hloubku 4, na 64-bitovém 8 – počet bytů v jednom slově. Přístup k takto uloženým prvkům je tedy dosti rychlý. Judy však navíc zavádí možnost, kdy ukazatel „přeskočí“ několik úrovní, pokud se v nich strom nevětví, takže se přístup ještě zrychlí. (V praxi by 64-bitovému počítači nestačila všechna RAM na celé Zemi k tomu, aby se strom Judy zaplnil natolik, aby se průměrný počet úrovní, kterými je nutno při vybrání prvku projít, skutečně přiblížil k číslu 8.)

A takto jednoduché myšlenky vedly k překvapivým výsledkům. (Ovšem až po několika letech usilovné práce.)

2 JudyL

2.1 Použití

Nyní se podíváme konkrétně na JudyL, což je nejvýznamnější část Judy. JudyL je implementací datové struktury Judy strom, kdy index je jedno slovo (tedy celé číslo nebo ukazatel) a ke každému indexu je přiřazena hodnota velikosti jednoho slova (nejčastěji ukazatel na nějakou další strukturu).

Použití JudyL je jednoduché – základní pravidlo zní: představujte si pole JudyL jako obyčejné pole.

Protože prázdné pole je reprezentováno ukazatelem s hodnotou `NULL`, není nutno provádět žádnou zvláštní inicializaci:

```
void * my_array = NULL;
```

Vložení prvku do pole provede funkce `JudyLIns()`:

```
value_area = JudyLIns(&my_array, index, &JError);
* value_area = value;
```

Pohodlnější je však použít makro `JLI`:

```
JLI(value_area, my_array, index);
* value_area = value;
```

(Všechny funkce JudyL jsou takto dostupné ve dvou verzích, jako makro preprocesoru i jako skutečná funkce.)

Další funkce JudyL umožňují získat uložený prvek pole, smazat jednotlivý prvek nebo vymazat celé pole. Konkrétní podobu těchto funkcí lze nalézt na domovské stránce Judy: `http://www.sourcejudy.com/`.

Protože je Judy v podstatě číslicový strom, je pole uloženo v setříděné podobě, a je také možno je procházet (vzestupně či sestupně).

Protože si Judy pamatuje u každého podstromu hodnotu populace, lze efektivně zjistit počet prvků v celém poli či v určitém intervalu a rovněž je možné efektivně nalézt n-tý obsazený prvek v poli JudyL.

2.2 Hlavní principy

Ačkoliv je Judy API průzračně jednoduché, nic jiného na Judy „jednoduché“ není. Pokud složitější datový model či algoritmus umožní zvýšení rychlosti či snížení paměťových nároků, bude nejspíše v Judy použit. Na druhou stranu, pokud se při testování ukáže, že nějaký nadějně vyhlížející nápad ve skutečnosti nepřináší výrazné zlepšení, není důvod kód Judy zbytečně komplikovat.

Nejlépe tento princip vystihl Alan Silverstein, když do manuálu [1] zařadil následující citát:

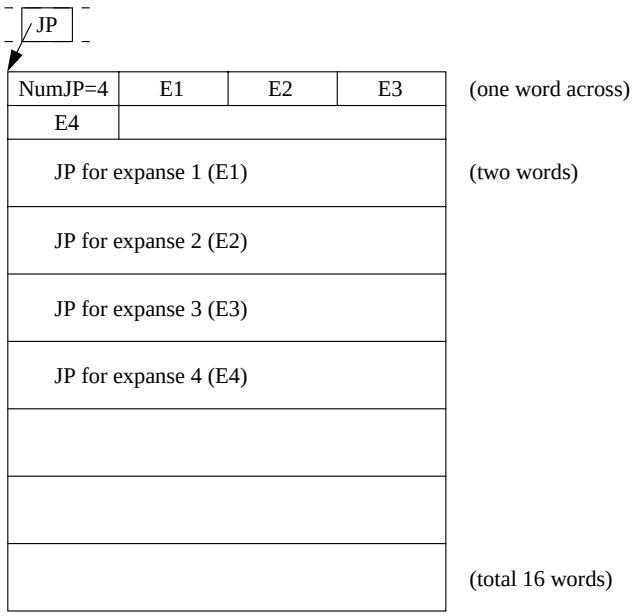
> Všechno má být tak jednoduché, jak to jen lze, ale ne jednodušší.
>
> — Albert Einstein

Další myšlenkou, takovou mantrou, kterou si autoři Judy neustále opakovali, je fakt, že *ze všeho nejdůležitější jsou výpadky procesorové keše*. Protože přístup do hlavní paměti odpovídá 50 až 2000 provedených instrukcí, je zřejmé, že pokud lze jeden výpadek ušetřit za cenu méně než 50 instrukcí, vyplatí se to udělat. Keš procesoru pracuje s úseky dlouhými 16 či 8 slov; celý takový úsek lze tedy použít, aniž by došlo k citelnému zdržení. Je zarážející, jak dlouho byl tento fakt přehlížen a teoretické práce uvažovaly pouze o počtu přístupů do paměti.

3 Stručný popis struktur použitých v JudyL

3.1 Terminologie

Jak již bylo řečeno v oddíle 1.4, je Judy v podstatě číslicový strom používající zápis v soustavě o základu 256. Hloubka stromu JudyL je tedy 4 (na 32-bitové platformě). Jednotlivé *úrovně* (anglicky levels) se číslují od 0 do 4, přičemž kořen stromu je na úrovni 4. Listy stromu, reprezentující jednotlivé indexy a obsahující příslušné hodnoty v poli uložené, by tedy byly na úrovni 0. Takovéto listy, které by měly velikost jednoho slova, se však v Judy nepoužívají. Proto se termín *list* (anglicky leaf) používá pro uzel o úroveň výše, který obsahuje nejvýše 256 indexů a jim přiřazených hodnot. Pro uzly na vyšších úrovních se používá termín *rozvětvení* (anglicky branch).



Obrázek 3. Lineární rozvětvení

3.2 Typy uzlů použitých v JudyL

3.2.1 Lineární rozvětvení Jak již bylo několikrát řečeno, přirozenou podobou každého uzlu v JudyL je pole délky 256. Existenci takového pole však nelze ospravedlnit, pokud zůstává většina prvků neobsazena.

Je-li obsazeno nejvýše 7 prvků (větví), použije se *lineární rozvětvení* (anglicky linear branch). Jedná se o velice jednoduchou strukturu:

– 1 byte udává počet obsazených prvků;

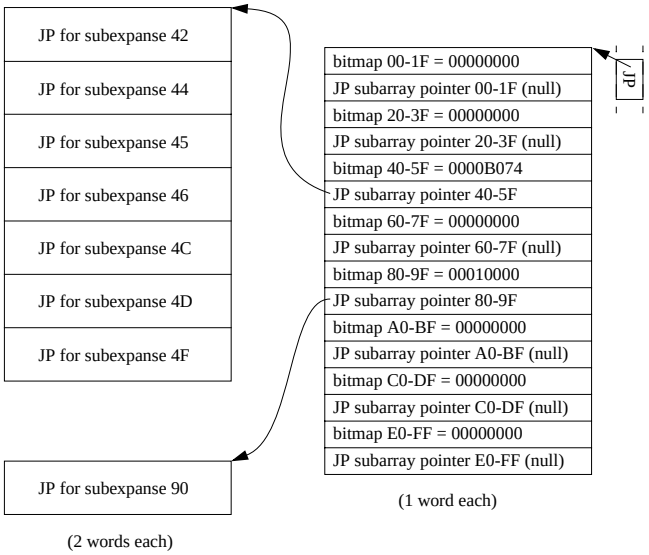
– následuje seznam obsazených číslic (bytů); jsou v paměti prostě zapsány za sebou ve vzestupném pořadí;
– seznam ukazatelů na uzly úrovně o 1 nižší; ukazatele jsou opět v paměti uloženy za sebou.

Lineární rozvětvení ilustruje obrázek 3.

3.2.2 Rozvětvení s bitovou mapou Je-li obsazeno více než 7 prvků, je použit objekt, který obsahuje bitovou mapu obsazených prvků, k níž jsou připojeny po řadě zapsané ukazatele odpovídající obsazeným prvkům.

Aby při modifikaci této struktury nebylo nutno přesouvat delší úseky paměti, je 256 možných „číslic“ rozděleno do 8 intervalů délky 32, přičemž pro každý interval je zvlášť alokován úsek paměti, který obsahuje příslušné množství ukazatelů. (Je-li v některém intervalu méně bitů, je alokován kratší úsek, čímž se šetří paměť.)

Protože keš některých procesorů pracuje s úseky dlouhými jen 8 slov, je vhodné uložit střídavě vždy slovo obsahující 32 bitů z bitové mapy a hned za ním příslušný ukazatel na úsek paměti odpovídající těmto 32 bitům. Tak je zajištěno, že každý průchod rozvětvením s bitovou mapou způsobí nejvýše dva výpadky keše: první při čtení příslušné části bitové mapy a těsně sousedícího ukazatele, druhý při čtení paměti obsahující ukazatel na další úroveň. Nejlépe celou situaci ilustruje obrázek 4.

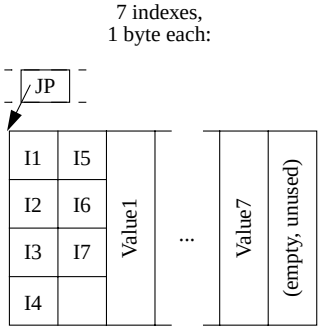


Obrázek 4. Rozvětvení s bitovou mapou

3.2.3 Nekomprimované rozvětvení Pokud je v rozvětvení s bitovou mapou obsazena většina číslic, nepřináší již žádnou podstatnou úsporu paměti. Proto je v takovém případě použita základní forma uzlu v číslicových stromech: pole ukazatelů délky 256, ukazatele odpovídající neobsazeným číslicím obsahují hodnotu `NULL`. Takový uzel nazýváme *nekomprimované rozvětvení* (anglicky uncompressed branch).

Průchod nekomprimovaným rozvětvením je velice efektivní, nemůže nikdy způsobit více než jeden výpadek procesorové keše. Z tohoto důvodu jsou uzly na vyšších úrovních, které mají větší populaci, někdy oportunisticky[1] dekomprimovány, pokud to celková paměťová situace dovoluje.

3.2.4 Lineární list Pojem *list* se v Judy používá pro uzly na úrovni 1 (viz oddíl 3.1). *Lineární list* (anglicky linear leaf) obsahuje za sebou napsané číslice (byty) odpovídající obsazeným indexům, za nimiž následují paměťové buňky pro hodnoty přiřazené těmto indexům.



Obrázek 5. Lineární list

3.2.5 List s bitovou mapou Lineární list smí obsahovat nejvýše 25 indexů. Pro více indexů se použije *list s bitovou mapou* (anglicky bitmap leaf). Jeho struktura je analogická struktuře rozvětvení s bitovou mapou (viz oddíl 3.2.2). Bitová mapa (která je opět rozdělena na osm částí) uchovává informaci o tom, které indexy jsou přítomny, připojené ukazatele obsahují adresy paměťových bloků, ve kterých jsou uloženy hodnoty přiřazené těmto indexům.

3.2.6 Nekomprimované listy neexistují Analogií k nekomprimovanému rozvětvení by byl nekomprimovaný list, který by obsahoval 256 hodnot, přiřazených jednotlivým indexům.

[1] předvídavě, vypočítavě, účelově

Je zde však jeden podstatný rozdíl: v nekomprimovaném rozvětvení se mohla na některém místě vyskytovat hodnota `NULL`, indikující, že příslušná číslice se v žádném indexu nevyskytuje. V případě nekomprimovaného listu však může být k indexu přiřazena libovolná hodnota, třeba zrovna `NULL`, takže nelze hodnotu `NULL` užívat k označení nepoužitých indexů.

Nekomprimovaný list bez bitové mapy by tedy šlo použít pouze v případě, kdy je použito všech 256 indexů. Objekt, pro který bychom měli tak malé využití, je lépe nezavádět. Proto tedy nekomprimované listy neexistují.

3.3 JP (Judy Pointer)

V předchozím oddíle jsme se seznámili s pěti základními typy objektů. (Ve skutečnosti jich JudyL používá mnohem více.) Musí tedy existovat mechanismus, který nám umožní rozpoznat, s jakým objektem máme právě tu čest.

Možnost, která jistě každému přijde na mysl jako první, je umístit na začátek každé struktury vhodný identifikátor. Takový postup by však podstatně narušil navržené struktury. Uvědomme si, že např. uzel s bitovou mapou je navržen tak, že zabírá úsek paměti dlouhý 16 slov, což na procesoru typu i386 představuje 2 úseky paměti procesorové keše (angl. 2 cache lines). Prodloužení této struktury, byť i jen o jeden byte, je tudíž nežádoucí.

Naštěstí je zde druhá možnost: údaj o typu objektu může být připojen k ukazateli, který na tento objekt ukazuje. Vzhledem k tomu, že se jedná o stromovou strkturu, kdy na každý objekt ukazuje právě jeden ukazatel, nepřináší tato metoda žádné zvýšení pamětových nároků.

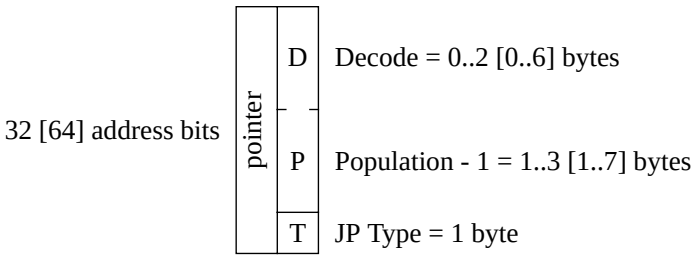
Tím se dostáváme k pojmu *JP* (*Judy Pointer*). JP má velikost dvě slova – první slovo obsahuje vlastní ukazatel na objekt, ve druhém slově se nachází identifikátor, určující typ následujícího objektu. Pro tento identifikátor se používá poněkud nelogické označení *typ JP* (anglicky JP type).

Ačkoliv Judy obsahuje nepřeberné množství různých typů objektů, je jich přeci jen méně než 256. Typ JP lze tudíž zapsat do jednoho bytu, takže téměř celé druhé slovo zůstává zatím nevyužito.

Toto necelé slovo lze použít pro uložení populace (počtu indexů) vyskytující se pod daným JP. Protože může pole JudyL obsahovat až 256^4 indexů (na 32-bitové architektuře), je nutno pro uložení celkové populace vyhradit celé slovo. Ovšem jakmile je dekódován první byte (číslice), může být počet indexů začínajících tímto bytem již jen 256^3. Takže JP, který se vyskytuje uvnitř rozvětvení nejvyšší úrovně a který ukazuje na objekt 3. úrovně, může pod sebou skrývat nejvýše 256^3 indexů. Tento údaj o populaci lze tedy uložit do onoho zbývajícího necelého slova (což jsou 3 byty). Podobně i na 2. a 1. úrovni – takže každý JP, kromě kořenového, v sobě obsahuje údaj o populaci uložené pod tímto JP.

To však stále není vše: JP ukazující na objekt 2. úrovně může pod sebou skrývat jen 256^2 indexů, JP ukazující na list 1. úrovně jen 256. Zbývá tedy ještě jeden (resp. dva) nevyužité byty. Do těch uložíme již dekódované číslice (byty), kromě první. Obecně řečeno, na architektuře, kde slovo obsahuje n bytů, je na i-té úrovni již dekódováno $(n - i)$ bytů, přičemž maximální možná populace

je 256^i. Pokud tedy použijeme 1 byte na uložení typu JP a i bytů na uložení populace, zbývá právě $(n - i - 1)$ bytů na uložení všech již dekódovaných číslic, kromě první (viz obrázek 6).



Obrázek 6. Struktura JP

Ukládání již dekódovaných číslic se na první pohled zdá zbytečné: tato informace je redundantní, pokud jsme prošli číslicovým stromem až do určitého místa, všechny tyto číslice již přeci známe. Již následující odstavec však přinese vysvětlení.

3.4 Zkratkový JP

Pokud by v některém rozvětvení byla obsazena jen jedna číslice, jednalo by se vlastně o uzel, ve kterém ke skutečnému větvení nedochází. Zkratkový JP umožňuje takový uzel přeskočit. Tedy *zkratkový JP* (anglicky narrow JP) ukazuje nikoliv o 1 úroveň níže, ale o 2 i více úrovní níže.

Jak jsme již ukázali v předchozím oddíle, je na uložení populace nutno vyhradit takový počet bytů, jaké je číslo úrovně *na kterou JP ukazuje*. Ve zkratkovém JP tedy zbývá místo na uložení číslic, které odpovídají vynechaným úrovním. (Pouze kořenový JP nemůže být zkratkový. Přesněji řečeno, pokud by kořenový JP měl být zkratkový, musel by se tento mechanismus upravit, protože JP neobsahuje první dekódovaný byte.)

3.5 Okamžitý JP

Okamžitý JP (což je neobratný překlad anglického „immediate JP“) je vlastně zkratkový JP, který jako by ukazoval na objekt na 0. úrovni. (Takový objekt by obsahoval jen jeden index.) V tomto JP tedy není žádný byte použit na uložení populace, zato je v něm uložen celý index kromě prvního bytu.

První slovo v takovém JP tedy není využito, protože objekt 0. úrovně ve skutečnosti neexistuje. Toto slovo lze tedy použít přímo jako paměťovou buňku pro uložení hodnoty příslušné k uloženému indexu.

Existují ještě další varianty okamžitého JP, zájemce odkazuji na detailní popis v podrobném manuálu [1].

3.6 Redundance?

Předcházející text vysvětlil, proč je nutné v některých případech ukládat některé dekódované číslice, ale proč by se měly ukládat vždy?

Odpověď na tuto otázku není jednoduchá. Stručně lze říci, že při takovémto uspořádání věcí je vkládání i odstraňování prvků pole rychlejší, protože se při změnách, které se dotýkají zkratek, mění méně bytů v paměti.

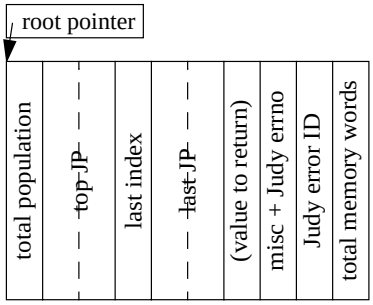
3.7 Znovu o lineárních listech

V definici lineárního listu (oddíl 3.2.4) chyběla jedna podstatná věc. Ačkoliv jsem doposud tvrdil, že list se vždy nachází na 1. úrovni, nemusí to být vždy pravda. Lineární list se může nacházet obecně na i-té úrovni. Takový list obsahuje nejprve i-tice bytů, zapsané v paměti těsně za sebou, za nimi následuje příslušný počet slov, obsahujících hodnoty přiřazené takto definovaným indexům.

Ještě jedna otázka se nabízí v souvislosti s lineárními listy: Jak Judy pozná, kolik prvků lineární list obsahuje, když v něm tento počet není uložen? Vtip je v tom, že JP, který na tento list ukazuje, obsahuje údaj o populaci, což je právě ten počet prvků, který potřebujeme znát.

3.8 JPM (Judy Population/Memory Node)

Ukazatel, který uživatel uchovává jako vstupní bod do JudyL pole, většinou ukazuje na zvláštní strukturu, nazvanou *JPM* (*Judy Population/Memory Node*). Obsah této stuktury je znázorněn na obrázku 7.



Obrázek 7. Struktura JPM

JPM tedy obsahuje především kořenový JP a údaj o populaci, který se do kořenového JP nevejde.

Dále obsahuje údaj o celkovém množství paměti, kterou tento JudyL strom zabírá. Tento údaj se využívá při rozhodování o oportunistické dekompresi nejvíce zatížených rozvětvení.

Zbylé místo v JPM se používá při ošetření chyb či pro jednoprvkovou keš (co kdyby se nás někdo zeptal na tentýž prvek dvakrát po sobě?).

3.9 JAP (Judy Array Pointer)

Již víme, že nejčastěji je JudyL pole vlastně ukazatelem na JPM. Dále víme, že prázdné pole je reprezentováno hodnotou `NULL`.

Co však s poli, která obsahují tak málo prvků, že nelze ospravedlnit vytvoření JPM struktury? Takové pole bude reprezentováno ukazatelem na tzv. *kořenový list* (anglicky root leaf). V JudyL jsou tři varianty kořenového listu: kořenový list obsahující 1 prvek (velký pouhá 2 slova), kořenový list obsahující 2 prvky (velký 4 slova) a obecný kořenový list, který v 1. slově obsahuje hodnotu populace a může obsahovat až 31 prvků.

Jak však JudyL pozná, který z těchto čtyř typů objektů je ve skutečnosti použit? K tomu se využívá následující trik.

Vzhledem k tomu, že JP zabírá dvě slova, není překvapivé, že bylo Judy navrženo tak, aby všechny alokované úseky paměti byly velké sudý počet slov. Navíc jsou tyto úseky paměti alokovány na zarovnaná místa v paměti, tedy adresa každého úseku je násobkem osmi (na 64-bitové architektuře dokonce šestnácti). Proto jsou nejnižší tři bity každého ukazatele nulové.

Do těchto tří bitů lze tedy bez obav uložit jakoukoli informaci, pouze je před dereferencováním ukazatele nutno tuto informaci vymazat a bity nastavit opět na nulu. Ale to je v pořádku, protože uživatel nemá právo provást dereferenci ukazatele, který reprezentuje JudyL.

Takovýto ukazatel obohacený o 3 bity dodatečné informace se nazývá *JAP* (*Judy Array Pointer*).

Zájemce o konkrétnější údaje mohu opět odkázat na manuál [1], strana 28.

Tento trik by bylo jistě možno využít i uvnitř Judy stromu, ale pokud vím, nebylo to zatím nutné.

Díky výše uvedenému opatření může programátor bez obav vytvářet velké množství JudyL polí, z nichž třeba zůstane většina téměř prázdná. Jedna z aplikací, která této skutečnosti využívá, je JudySL, o které se zmíníme v oddíle 5.2.

4 Další principy

Nyní se pokusíme vystihnout další obecné rysy implementace Judy, kromě těch, které byly vysvětleny v oddíle 2.2 na straně 120.

4.1 Oportunistická dekomprese

Jak jsme se již zmínili v oddíle 3.2.3, Judy se snaží, aby byl strom komprimován *co nejméně*. Na první pohled je to možná překvapivé, ale vše se vyjasní, pokud si uvědomíme, že hlavním cílem Judy je co nejrychlejší fungování. Komprimace jednotlivých uzlů slouží pouze k tomu, aby nikdo nemohl říci, že Judy má neúměrné paměťové nároky. Jinak řečeno, Judy si přísně „hlídá linii,“ ale v rámci dovoleného přídělu paměti využívá všech prostředků k docílení co nejrychlejšího přístupu.

4.2 Hystereze

Ve slovníku cizích slov se dovídáme, že hystereze je jev, kdy se účinek opožďuje za příčinou.

V kontextu Judy máme na mysli následující skutečnost. Ke změně objektu na jiný, větší, dochází okamžitě, jakmile je při vložení nového prvku kapacita objektu překročena. Avšak při odebrání prvku z pole *není* objekt nahrazen menším, jakmile je to možné.

Díky tomuto hysterezním chování operace smazání prvku nemůže dojít k jevu, kdy by střídavé přidávání a ubírání jednoho prvku trvalo příliš dlouho, protože by populace byla právě na mezní hodnotě pro kapacitu určitého objektu.

Takovéto zpoždění při operaci smazání prvku stačí o 1, větší zpoždění není nutné.

Pro implementaci tudíž stačí jednoduché pravidlo, že objekt je nahrazen menším, pokud to lze provést *ještě před odebráním prvku*, který se má z pole odstranit.

4.3 Správa paměti

Efektivita Judy silně závisí na efektivitě správce paměti. Navíc je nutno zajistit, aby alokované úseky začínaly na adrese, která je sudým násobkem slova (viz oddíl 3.9).

Z těchto důvodů obsahuje Judy vlastního specializovaného správce paměti, který si od operačního systému vyžádá paměť ve větších úsecích, které sám spravuje. Tento specializovaný správce např. přiděluje pouze úseky několika vyjmenovaných velikostí. Podrobnosti lze nalézt na straně 52 manuálu [1].

4.4 Další pravidla

Ve vývoji Judy jsou patrné tyto postupy:

Efekt každé změny, každé metody komprese je nutno důkladně otestovat. Při měření efektivity se musí použít různé typy dat (náhodná, intervaly,...). Pouze pokud takováto měření prokáží užitečnost určitého kódu, je tento kód začleněn do Judy.

Téměř 100 % zdrojového kódu je pokryto regresními testy. (Regresní test je test, který lze provádět opakovaně, k ověření, zda nová verze programu stále funguje správně. V projektech svobodného softwaru se regresní testy – většinou zoufale neúplné – často skrývají za příkazem „`make check`“.) Pokud některá místa kódu nejsou pokryta regresními testy, musí pro to být důvod.

5 Další součásti Judy

5.1 JudyL and Judy1

Dosud jsme hovořili prakticky výhradně o JudyL, která implementuje řídké pole, kde indexy jsou celá čísla či ukazatele, přiřazené hodnoty jsou rovněž celá čísla či ukazatele.

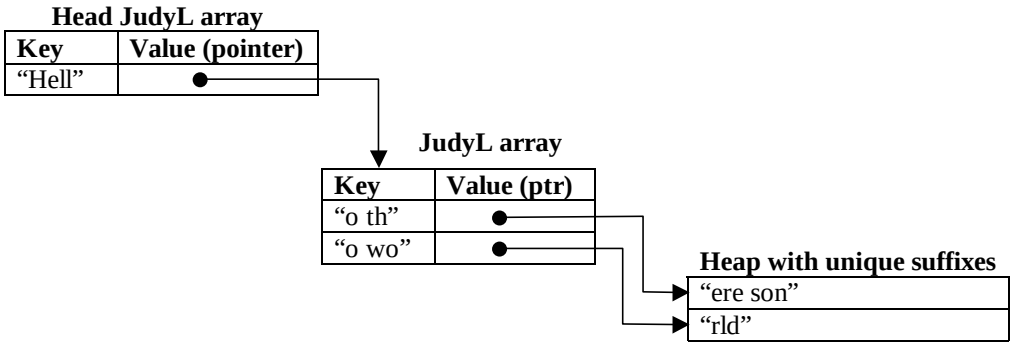
Další variantou je Judy1, které nepřiřazuje indexům žádnou hodnotu. Pole tedy uchovává pouze informaci o tom, které indexu obsahuje a které nikoli. Tato varianta je implementována velmi podobně jako JudyL, dokonce se kompiluje ze stejného zdrojového kódu (který je ovšem hojně prošpikován direktivami `#ifdef`).

5.2 JudySL

JudySL je varianta, kdy indexy jsou řetězce, kterým jsou přiřazena jako hodnoty celá čísla, resp. ukazatele (opět jedno slovo paměti).

Implementace JudySL je vysvětlena v článku [2] a je pozoruhodně jednoduchá. Jedná se vlastně o meta-trie vybudovaný z polí JudyL. Na první úrovni je JudyL pole, jehož index tvoří první 4 znaky řetězce (na 32-bitovém počítači). Hodnota uložená v tomto poli je ukazatel na další JudyL pole, které zajišťuje větvení podle druhé čtveřice znaků. Konečně listy tohoto meta-stromu jsou tvořeny jednoduchými strukturami, které obsahují jednak zbylou jednoznačnou část indexu, jednak přiřazenou hodnotu, jak vidíme na obrázku 8.

Word 1	Word 2	Word 3	Word 4
“Hell”	“o wo”	“rld”	
“Hell”	“o th”	“ere ”	“son”



Obrázek 8. Řetězce rozdělíme na úseky po čtyřech znacích a uložíme do „kaskády“ JudyL stromů

Ukazuje se, že JudySL může svou efektivitou někdy překonat i adaptivní hashing.

JudySL navíc obsahuje indexy v lexikografickém uspořádání, lze k nim přistupovat popořadě, podobně jako v případě JudyL. Rovněž je možné se ptát na počet indexů v určitém intervalu. Měření ukazují, že naplnění JudySL pole může být rychlejší způsob třídění než quick-sort.

Hlavní nevýhodou JudySL je fakt, že pracuje s řetězci ukončenými znakem `'\0'`, tedy tento znak se v řetězci nemůže vyskytnout.

6 Aplikace Judy a probíhající vývoj

6.1 Judy a Ruby

Yukihiro Matsumoto (`matz`), autor programovacího jazyka Ruby, projevil zájem použít Judy pro implementaci asociativních polí v Ruby. Protože se ukázalo, že indexem těchto asociativních polí není ve skutečnosti řetězec, ale objekt, stačí k tomuto účelu nasadit JudyL. Zdá se tedy, že budoucí verze Ruby budou používat Judy.

6.2 Judy a hashing

Článek [3] pojednává o kombinovaném použití hashingu a Judy. Zabývá se situací, kdy index má velikost jednoho slova v paměti, hashovací funkce tento index umístí někam do hashovací tabulky, a JudyL je použito pro správu synonym v každém políčku hashovací tabulky. Celý článek lze shrnout do jedné věty: nejvýhodnější je použít hashovací tabulku velikosti 1, tedy svěřit veškerou práci JudyL, bez jakéhokoli hashingu.

Při čtení výše uvedeného článku jsem se nemohl ubránit dojmu, že by to mělo být naopak: JudyL pole by mělo být použito místo hashovací tabulky. (Pokud jsou indexy čísla, nedává to smysl, ale pokud se jedná o implementaci asociativního pole, začíná to být zajímavé.) Pak máme vlastně virtuální hashovací tabulku velikosti 256^4, takže se pravděpodobnost kolize nejspíše sníží na minimum. (Na rozdíl od adaptivního hashingu odpadá nutnost projít všechny prvky při každé změně velikosti hashovací tabulky.)

Tento svůj nápad jsem uvedl v emailu hlavnímu autoru Judy, Dougu Baskinsovi. Nedávno jsem se od něj dověděl, že tuto cestu prozkoumal a že skutečně funguje: dává efektivnější implementaci asociativních polí než JudySL, samozřejmě pouze pro případ, kdy nezáleží na uspořádání indexů. (Doug píše, že je podivuhodné, že tento postup tak dlouho nikoho nenapadl, ale neuvádí explicitně, zda byl inspirován mým dopisem, či zda jsme nezávisle přišli na tutéž ideu.) Doug rovněž uvádí, že je možné použít i dosti nedokonalou hashovací funkci, jejíž výpočet je rychlejší, protože pravděpodobnost kolize v rámci tak velké hashovací tabulky zůstává stále velice nízká.

Takovýto nový typ pole Judy by jistě nalezl mnoho uplatnění. Já se osobně nejvíce zajímám o použití v implementaci asociativních polí v jazyku GNU awk.

7 Historie a budoucnost

Projekt Judy začal pod vedením Douga Baskinse v rámci společnosti Hewlett Packard, a to v lednu 2000. Další významný autor Judy je Alan Silverstein, který mj. napsal JudySL a velkou část dokumentace.

Judy je jméno sestry Douga Baskinse. Doug zvolil pro svůj projekt toto jméno, protože prostě nepřipadl na žádný lepší název.

Po dvou a půl letech vývoje byl projekt Judy letos v červnu uvolněn jako svobodný software, pod licencí LGPL.

Je to tedy mladý projekt, který ještě nemá vytvořený okruh vývojářů. Na druhou stranu je to však projekt plně funkční a zralý. Proto se jedná o výjimečnou příležitost uplatnění pro teoretičtěji zaměřené programátory. Všichni ostatní programátoři mohou datové struktury Judy alespoň používat ke zrychlení svých aplikací.

Pokud vás Judy zaujme a budete se chtít dovědět více, můžete si přečíst manuál [1], či obrátit se na emailové adresy `doug@sourcejudy.com` a `alan@sourcejudy.com`. Pro méně zasvěcený rozhovor vedený v češtině se můžete obrátit na adresu uvedenou pod nadpisem tohoto článku.

Reference

1. Alan Silverstein. Judy IV Shop Manual.
   `http://www.sourcejudy.com/application/shop_interm.pdf`
2. Alan Silverstein: Associative Array (a.k.a. JudySL).
   `http://www.sourcejudy.com/application/judysl.pdf`
3. Alan Silverstein: Scalable Hashing.
   `http://www.sourcejudy.com/application/Judy_hashing.pdf`

Základy bezpečného programování pod (nejen) pro OS Linux

Jiří Kosina

MFF UK, Praha
Email: jikos@jikos.cz

Abstrakt: V přednášce budou demonstrovány nejběžnější typy bezpečnostních chyb, kterých se programátor může při programování (nejen) pod linuxem dopustit, se zaměřením na buffer overflow, chyby při formátování řetězců, race conditions, případně nejběžnější chyby při vývoji webovskych aplikaci (cgi, php – špatná validace vstupních dat).

Klíčová slova: Buffer overflow, race condition, format string

1 Úvod

Asi není příliš nutné zdůrazňovat, jak je v dnešní době důležité psaní bezpečných aplikací – v době, kdy aplikace běží na Internetu, kde k nim má přístup takřka kdokoliv, kdy se informace, které jsou spravovány těmito aplikacemi, stávají stále citlivějšími a kdy si takřka denně můžeme přečíst informace o strojích, které někdo proboural díky bezpečnostní chybě v aplikaci. Tato přednáška by měla sloužit jen jako velice jemný úvod do tvorby bezpečných aplikací, protože jde o problematiku velice rozsáhlou, a samozřejmě dosud ne zcela prozkoumanou.

2 Spouštění externích příkazů

Představme si následující program, který běží v systému s právy, která bychom, coby útočník, chtěli získat:

```
int main (void)
{
if (system ("mail nekdo@nekde.cz < /home/vypisy/vypis.txt"))
     perror ("system");
   return (0);
}
```

Funkce `system()` spustí shell (typicky `/bin/sh`) s parametrem `-c`, za kterým mu předá své argumenty. V našem případě pak shell podle cesty, nastavené v proměnné prostředí `PATH`, zkouší najít program `mail`, který pak spustí. Pokud je tedy výše uvedený program zkompilovaný pod názvem `/usr/sbin/vuln1` a před jeho spuštěním si ještě vytvoříme ve svém adresáři skript s názvem `mail` s následujícím obsahem:

Jan Kasprzak, Petr Sojka (editoři): SLT 2002 – sborník semináře o Linuxu a TEXu, str. 133–144, 2002.

```
#!/bin/bash
/bin/sh
```

pak práva, která má program `/usr/sbin/vuln1`, získáme triviálně takto:

```
$ export PATH="."
$ /usr/sbin/vuln1
```

Výsledkem bude běžící shell se stejnými právy, s jakými se spouští program `vuln1`.

Jedním z možných řešení tohoto konkrétního problému je uvést celou cestu k programu, který pomocí system() spoušíme, a nespoléhat se na nastavení proměnné `PATH`. Jednou z nevýhod tohoto přístupu je samozřejmě to, že se spoléháme na lokální konfiguraci systému, což u programů typu `ls`, `mail`, apod. není zásadní problém, ale ne vždy musíme u každého programu vědět, kde se v systému nachází (typicky jde o programy ručně instalované správcem do `/usr`, `/usr/local`, apod.). Další nevýhodou, a to poněkud podstatnější, je fakt, že i program, který používá plnou cestu k programu, je stále zranitelný pomocí obdobného útoku. Stačí si uvědomit existenci proměnné prostředí `IFS`, kterou používá shell (nejen) k parsování příkazové řádky. Obsah této proměnné určuje, které znaky mají být brány jako oddělovače polí. Defaultní nastavení této proměnné je mezera, tabulátor, atd. Co se však stane, budeme-li používat volání tvaru

```
system("/bin/mail ...");
```

pak pokud útočník provede něco na způsob:

```
$ export PATH="."
$ export IFS=$IFS/
$ /usr/sbin/vuln1
```

tak opět dosáhne spuštění skriptu s názvem `bin` v aktuálním adresáři (který útočník naplní pravděpodobně stejně, jako v předchozím případě skript s názvem `mail`.

Možných řešení těchto problémů je několik. Především je možné se vyhnout úplně používání volání `system()`, a místo něj si spouštění implementovat sám (pomocí `fork()`, `exec()`, přesměrování, které je při použití `system()` jednoduché (protože ho za nás řeší shell) řešit sami pomocí `dup()`, atd.

Dalším řešením je před voláním funkce `system()` si nastavit takové hodnoty proměnných prostředí, jaké chceme, a které jsou bezpečné. Například

```
clearenv();
setenv ("IFS", "\n\t",1);
setenv ("PATH", "/bin:/usr/bin:/usr/local/bin", 1);
system ("mail nekdo@nekde.cz < soubor");
```

3 Buffer overflow

Buffer overflow je souhrný název pro chyby, které umožňují změnit některou část paměti programu, přestože programátor neměl původně v úmyslu nic takového dovolit. Myšlenkově lze tyto chyby rozdělit podle typu paměti, ve které k přepisu dojde, na *stack overflow* a *heap overflow*, přičemž na zneužití každé z nich musí útočník použít poněkud jiné techniky.

Stack (česky zásobník) je název pro paměť, kterou používají vzájemně se volající funkce k předáváni parametrů (a dalsích pomocných hodnot), a kompilátory do ní často umisťují lokální a statické proměnné. *Heap* (česky halda) je název pro paměť, do které kompilátor umisťuje globální a dynamická data (z této oblasti například vrací funkce `malloc()` kusy paměti).

Důležitý fakt, který zneužívají útočníci při stack overflow, je ten, že na zásobníku jsou uloženy parametry, které daná funkce dostala, lokální proměnné, a teprve za nimi (na vyšší adrese) je uložena adresa volající funkce – aby bylo, jednoduše řečeno, „kam se vrátit“, až bude běh této funkce ukončen. Pokud programátor špatně ohlídá meze některé z proměnných na zásobníku, může útočník přepsat tuto návratovou hodnotu, čímž způsobí, že po opuštění funkce se začne provádět jiný kód, než který by se prováděl v běžné situaci (tzn. začala by se provádět instrukce následující po volání funkce ve volající funkci).

Nejjednodušší „`Hello world!`“ program, který je zranitelný na stack overflow, vypadá takto:

```
int main(int argc, char * argv [])
{
char buf [512];

if (argc>1)
strcpy(buf, argv[1]);
return(0);
  }
```

Tomu, kdo útočí na podobný program, stačí jediné – uvědomit si přesně, jak v tomto konkrétním případě vypadá situace na zásobníku, přepsat uloženou adresu, kam se má po skončení funkce vrátit řízení na adresu proměnné `buf`, a do proměnné `buf` vložit (v tomto případě triviálně jako parametr na příkazové řádce) přimo kódované instrukce procesoru. Po opuštění funkce `main()` se provede skok na začátek proměnné `buf`, a procesor začne provádět v ní uložené instrukce. Obsahu proměnné `buf` se říká *shellcode*, protže má typicky za úkol spustit shell (který bude mít samozřejmě práva původní aplikace).

Pokud najde útočník v programu heap overflow, je v poněkud obtížnější situaci, protože nemůže přepsát adresu, kam se má předat řízení po skončení programu – musí postupovat jinak – má přistup k proměnným na haldě – musí tedy například přepsat globální ukazatel na funkci, a čekat, až se tento ukazatel použije k volání funkce.

Jak se vyhnout takovýmto chybám ve svých programech?

Používat n-funkce. Podle standardní konvence, standardní funkce, jako je například `strcpy()`, provádějí svojí činnost dokud nenarazí na nulu (konec řetězce). Funkce, kterým je možno parametrem určit maximální počet kopírovaných bajtů, tento problém řeší (`strncpy()`, `strncat()`, atd.). Je ovšem potřeba si dát pozor na postranní efekty (při delším řetězci není zkopírovaná koncová nula, apod.). Ovšem i v případě, že používáme funkce, které omezují počet zkopírovaných znaků, není to samo o sobě zárukou bezpečného kódu. Například v BIND DNS serveru byla svého času tato chyba

```
struct hosten *hp;
unsigned long address;

memcpy(&address, hp->h_addr_list[0], hp->h_length);
```

Sice je zde určeno kolik bajtů se má kopírovat, ale bohužel útočník může měnit hodnotu předávanou jako třetí parametr, určující délku.

Přemýšlet při používání indexů.

```
char table[20];

for (i=0; i<=20; i++)
table[i] = ...
```

Je chybné – zapíše se jeden znak za hranici table. A jak bylo ukázáno v Phrack 55, i takováto jednobajtová chyba stačí k úspěšnému útoku.

Pro vstup nepoužívat nebezpečné funkce. Nebezpečnou funkcí je například funkce `gets()` – ta by neměla být nikde používána, protože jí není možné specifikovat maximální možnou délku. Naprosto stejně nebezpečná konstrukce je často používaná

```
scanf("%s", string);
```

Ale u funkcí typu `scanf()` existuje možnost, jak řídit velikost vstupních dat formátovacím řetězcem:

```
scanf("%50s", string);
```

4 Format string

I v tomto případě, stejně jako v předchozím, jde o velmi nebezpečnou chybu, která umožňuje útočníkovi opět přímo měnit obsah paměti. Narozdíl od buffer overflow, kde jsou chyby často způsobeny tím, že programátor „nepřemýšlí“, v tomto případě vstupuje na scénu také lenost. Ke katastrofě stačí, aby programátor místo

```
printf("%s", retezec);
```

napsal

```
printf(retezec);
```

Tyto dvě instrukce mají samozřejmě pro „obyčejné“ řetězcové proměnné naprosto stejný efekt – když není v řetězci v prvním parametru nalezena žádná „formátovací“ instrukce, je obsah řetězce vypsán na obrazovku. Ovšem ještě před vypsáním řetězce funkce `printf()` parsuje první parametr, a hledá v něm formátovací direktivy. Pokud je najde, vyzvedne odpovídající parametr ze zásobníku (kde by jí byl ve standardní situaci předán). Většina programátorů zná a používá pouze základní formátovací direktivy pro výstup (`s` pro stringy, `c` pro znaky, `d` pro čísla, atd). Ovšem formátovací řetězce jsou mnohem bohatší, než se píše v běžných učebnicích programování – například parametr `n`. Manuálová stránka nám o něm poví toto:

```
The number of characters written so far is stored into the
integer indicated by the int * (or variant) pointer argument.
No argument is converted.
```

Tedy pokud `printf()` nalezne při parsování svého prvního parametru parametr `n`, uloží počet již vypsaných znaků tam, kam ukazuje odpovídající ukazatel (parametr funkce `printf`). Tedy je pomocí tohoto parametru možné zapsat do paměti na určené místo určenou hodnotu! Je také v tuto chvíli vhodné připomenout, že toto chování není specifické pouze pro funkci `printf()`, ale například funkce `syslog()`, používaná v Linuxu pro požádání systémového logovacího démona o zápis do log souboru, má naprosto stejnou sémantiku. Jak tohoto využít dále při útoku? Uvažujme následující program:

```
int main(int argc, char **argv){
char buffer[128];
char tmp[] = "\x01\x02\x03";

snprintf(buffer,sizeof(buffer), argv[1]);
buffer[sizeof(buffer) - 1] = 0;
printf("Obsah promenne buffer: [%s]\n",buffer);
}
```

Tento program na první pohled může skutečně vypadat naprosto v pořádku, a skutečně na světě existuje až překvapivé množství programů, které podobné konstrukce používají. Ukažme si teď, jak se bude program chovat v některých situacích (předpokládejme, že je zkompilován s všeříkajícím názvem `a.out` :)).

```
$ ./a.out 123
Obsah promenne buffer: Ahoj
$ ./a.out "123 %x"
Obsah promenne buffer: [Ahoj 30201]
$ ./a.out "123 %x %x"
Obsah promenne buffer: [Ahoj 30201 20333231]
```

Co se tu stalo? Funkce `snprintf()` byla formátovacím parametrem `x` požádána, aby odpovídající parametr převedla do hexadecimálního formátu a vypsala.

Jelikož ale žádné parametry nebyly funkci předány, vzala funkce ze zásobníku data, která tam byla již předtím. Ve druhém příkladě jsme dostal 30201 (0x00030201), což je na x86 procesoru (little endian) obsah proměnné tmp (01020300), která bydlí, coby lokální proměnná, na zásobníku. Pomocí dalšího parametru `x` jsme dostali první 4 bajty proměnné buffer, atd. Tedy za každé další `x` se posuneme na zásobníku o 4 bajty, a jsme schopni přečíst, co na tomto místě paměti leží. Jak bylo ukázáno dříve, je možné pomoci parametru `n` zapsat na určenou adresu určenou hodnotu. Což v kombinaci s právě ukázaným postupem umožňuje například zjistit návratovou hodnotu na zásobníku (adresu funkce, která naší funkci zavolala), a na tuto adresu pak napsat kód, který chceme spustit (přepsat v paměti instrukce volající funkce). Další možností je například přepsání ukazatele na funkci, který leží na zásobníku, atd. Snad jsou tyto ilustrativní příklady zneužití dostatečnou výstrahou. Někdy nastávají komplikace při zneužívání těchto chyb (například je v některých případech nutné, aby pro použití parametru `n` byl řetězec předaný `printf` skutečně velmi dlouhý, aby útočník trefil adresu, o kterou má zájem, atd. Tyto problémy jsou také řešitelné, ale to je již mimo rámec této přednášky).

Jak se takovýmto chybám vyvarovat? Jak bylo ukázáno, tyto chyby jsou způsobeny tím, že je uživateli dovoleno libovolně manipulovat s obsahem první proměnné funkcí `printf()` a podobných. Tedy řešením je vždy striktně používat nezkrácenou variantu s formátovacím parametrem `s`. Pokud to z nějakého důvodu není možné, je nutné si řetězec kontrolovat proti nebezpečným znakům ve vlasní režii.

5 Race conditions

Klasická definice race condition je, že race condition mezi procesy nastává tehdy, když výsledek operace závisí na tom, jak jsou proloženy instrukce jednotlivých procesů (jak jsou operačním systémem tyto úlohy naplánovány). Typický příklad, takříkajíc ze života: proces chce přistupovat k nějakému systémovému prostředku (souboru, kusu paměti, ...) exkluzivně. Zkontroluje, zdali žádný jiný proces tento prostředek nepoužívá, a pak ho začne používat. K race condition dojde ve chvíli, kdy jiný proces se zájmem o tento prostředek, provede přesně stejný test právě ve chvíli, kdy první proces již provedl test, ale ještě nezačal prostředek využívat. Chyby zmiňované v předešlých kapitolách umožňovaly útočníkovi spustit „svůj“ kód, který si dříve připravil, a který například spustil shell s právy, která vlastnil zranitelný program. Toto není typicky případ programů zranitelných pomocí race conditions.

Uvažujme program, který běží s právy uživatele root, a občas potřebuje uložit nějaká data do souboru vlastněného uživatelem, který program spustil. Chybný kód může vypadat například takto:

```
struct stat st;
FILE *f;

if (stat (filename, &st) < 0) {
```

```
    fprintf (stderr, "Soubor %s neexistuje!\n", argv [1]);
    exit(-1);
}
if (st.st_uid != getuid()) {
    fprintf (stderr, "Vlastnik souboru %s neodpovida Tvemu UID!",
             filename);
    exit(-1);
}
if (!S_ISREG (st.st_mode)) {
    fprintf (stderr, "%s Neni jen tak nejaky soubor (symlink?)\n",
             filename);
    exit(-1);
}

if ((f = fopen (filename, O_RDWR)) == NULL) {
    fprintf (stderr, "fopen() selhal!\n");
    exit(-1);
}
    fprintf (f, "%s\n", message);
    fclose (fp);
    fprintf (stderr, "Write OK\n");
    exit(1);
```

Pokud se ve chvíli mezi voláním `stat()` (které zjistí o souboru dostupné informace a naplní jimi strukturu `st`) a podmínkou, která zkoumá, je-li daný soubor symbolický link, podaří útočníkovi soubor zrušit a místo něj vytvořit symbolický link na soubor, který chce přepsat a na jehož přepsání by v běžné situaci neměl práva, ale běžící program je má (například `/etc/passwd`, `.rhosts` u uživatele v adresáři, apod.), je do souboru, na který symbolický link ukazuje, zapsán obsah proměnné message.

Problém s tímto typem útoku spočívá v tom, že race condition typicky nastává po poměrně krátkou dobu, a útočník musí mít tedy štěstí, aby takříkajíc „trefil“ tu správnou chvíli, (například pro vytvoření symbolického linku). Typicky se útočník v takovémto případě spoléhá na hrubou sílu, a vytvoří si krátký skript, který stále dokola zkouší spustit program a pak vytvořít symbolický link, přičemž si může vypomáhat různými dalšími akcemi, které mu zajistí zpomalení procesu, na který útočí (aby se race condition vyskytovala po co nejdelší dobu). Například tak, že mu pomocí systémového příkazu `nice` sníží prioritu, případně hodně zatíží systém nějakou nekonečnou smyčkou `(while(1))`, apod.

Abychom z výše uvedeného programu odstranili race condition, upravíme ho následujícím způsobem:

```
struct stat st;
int fd;
FILE * fp;

if ((fd = open (argv [1], O_WRONLY, 0)) < 0) {
    fprintf (stderr, "Soubor %s nelze otevrit\n", filename);
    exit(EXIT_FAILURE);
```

```
}
fstat (fd, & st);
if (st.st_uid != getuid()) {
    fprintf (stderr, "Vlastnik souboru %s neodpovida Tvemu UID!",
             filename);
    exit(-1);
}
if (!S_ISREG (st.st_mode)) {
    fprintf (stderr, "%s Neni jen tak nejaky soubor (symlink?)\n",
             filename);
    exit(-1);
}
if ((fp = fdopen(fd, "w")) == NULL) {
    printf("Soubor %s nelze otevrit\n", filename);
    exit(-1);
}
fprintf (fp, "%s", argv [2]);
fclose (fp);
fprintf (stderr, "Write OK!\n");
exit(1);
```

Tento upravený program využívá toho, že jakmile je jednou soubor pomocí volání `open()` otevřený a je mu přiřazen jádrem deskriptor, pak jakékoliv změny (jména, práv), která jsou provedena na tomto souboru, neovlivní již otevřený deskriptor – při otevření dojde k jakémusi svázání obsahu se jménem, takže i když někdo tento otevřený soubor zruší (jeho jméno přestane ve filesystému existovat), program který ho má otevřený s ním může běžným způsobem pracovat. Z tohoto příkladu je dobré si vzít ponaučení, že je vždy vhodné používat volání, která používají deskriptory otevřených souborů, než pouze cesty k souborům. Tedy používat `fchdir()`, `fchmod()`, `fchown()`, `fstat()`, `ftruncate()`,... místo jejich ekvivalentů, které místo deskriptoru mají jako parametr cestu k (neotevřenému) souboru.

Programy často potřebují ukládat dočasná data do souborů na disku. Otevírání dočasného souboru, pokud není uděláno dobře, velice často zavání race condition – i v takových projektech, jakými jsou `Apache`, `wu-ftpd`, `inn`, `getty`, byly již v historii nalezeny race conditions při práci s dočasnými soubory. Dočasné soubory jsou typicky vytvářeny v adresáři `/tmp`, z několika důvodů – aby systémový administrátor věděl, kde jsou tyto soubory centralizované, a mohl je periodicky promazávat, případně jsou tyto umístěny na speciální parittion (například ramdisku), atd. Adresář `/tmp` se od ostatníc běžných adresářů liší svými právy – má nastavený sticky bit, který zajišťuje, že pouze vlastník adresáře a vlastník souboru v tomto adresáři, mají povoleno soubor smazat, přestože právo na zápis mají do tohoto adresáře typicky všichni. Problém s race conditions při používání dočasných souborů je předvídatelnost jména souboru, který aplikace použije, čímž usnadňuje útočníkovi provedení útoku přes symbolické linky. Existují knihovní funkce, které poskytují nepredikovatelná jména pro dočasné soubory. Nicméně ani s náhodně generovanými jmény souborů není vhodné používat konstrukce typu

```
if((fd = open(filename, O_RDONLY)) != -1) {
/* soubor se podarilo otevrit */
fprintf(stderr, "Chyba: soubor %s jiz existuje\n",
filename);
exit(-1);
}
fd = open(filename, O_RDWR | O_CREAT, 0644);
```

protože samozřejmě obsahují na první pohled snadno rozpoznatelnou race condition, a pokud by se útočníkovi jakýmkoliv způsobem podařilo naše jméno souboru uhádnout, je pro něj již triviální tuto chybu zneužít. Řešením je používat flag `O_EXCL` spolu s `O_CREAT`, což má za důsledek, že `open()` vrátí chybu, pokud soubor již existuje, ale test a otevření provede atomicky (bez race condition – není možné tuto akci proložit akcí jinou).

6 CGI skripty

Nejtypičtějším útokem proti špatně napsané aplikaci, která s uživatelem interaguje přes web, je špatná validace vstupních dat, která uživatel může aplikaci poskytnout, a jejich následná špatná interpretace. Pouze pro inspiraci zde uvedu několik málo příkladů typických nebezpečných zdrojových kódů, kterých je skutečně plný web.

6.1 Cross-site scripting z rychlíku

`Hello-world!` skript, který je náchylný na typ útoku zvaný *cross-site-scripting* (někdy též označovaný *XSS*), může vypadat například takto:

```
<html>
<body>
<?
  echo Hello $name;
?>
```

Tím, že uživatel může libovolně ovlivnit zvenku obsah proměnné name, umožňuje mu to například do stránky vložit javascript. Pokud tato stránka bude používána například pro přihlašování do nějakého systému, může javascript hlídat heslo a po jeho zadání ho někomu oznámit. Ptáte se, proč by si někdo sám vkládal do stránky vhodnou formulací URL takovýto skript? Odpověď je, že k tomu může být oběť donucena – například tak, že jí útočník ze svých stránek (případně HTML mailem) odkáže na příslušnou stránku, ovšem za URL vloží ještě kód pro vložení skriptu, a pak už se může jen radovat, až mu bude oznámeno heslo oběti. Tento útok se i dnes často vyskytuje (donedávna například existoval na titulní přihlašovací stránce služby Yahoo).

6.2 PHP, include závislý na proměnné

Doby, kdy tuto chybu obsahoval skoro každý druhý PHP skript, jsou již naštěstí minulostí, nicméně i dnes lze (i například za pomocí chytře formulovaných dotazů v Googlu) najít neuvěřitelně velké množství stránek, které touto chybou dosud trpí. O co jde?

```
<html>
<body>
<?
  include($user);
?>
...
</body>
</html>
```

Tento mohl chtít například vložit soubor, který má stejné jméno jako uživatel (například si každý uživatel upraví, jak ho má skript po přihlášení pozdravit – to například umožňují některé webmaily). Problém je, že proměnnou user může uživatel pomocí URL ovládat zcela libovolně. Tedy si například může pomocí

```
http://server.cz/skript.php?user=../../../../etc/passwd
```

nechá do stránky vložit obsah příslušného souboru. A co hůř, funkce include akceptuje i URL... Tedy s trochou fantazie si můžeme někde na své IP adrese rozběhnout webserver, který nebude interpretovat PHP skripty, umístit si na něj soubor s obsahem

```
<?
system("....");
?>
```

PHP pak provede include, a interpretuje PHP skript – tedy na systému spustí takový příkaz, jaký se nám, coby útočníkovi, zlíbí.

6.3 (nejen) Perl

Předpokládejme, že chceme napsat skript, který bude přes web ukazovat výstup programu `finger` pro zadaného uživatele. Naviní skript může vypadat takto:

```
print "<BODY>";
$login = $input{'login'};
$login =~ s/([;<>\*\|`&\$!#\(\)\[\]\{\}:'"])/\\$1/g;
print "Login $login<BR>\n";
print "Finger<BR>\n";
$CMD= "/usr/bin/finger $login";
open(FILE,"$CMD") || goto form;
print <FILE>
```

Tento skript sice bere ohled na některé obecně známé chybné znaky, které uživatel za žádných okolností nesmí předat, aby se dostaly až k shellu (například zpětné apostrofy by shell interpretoval – tedy by spustil příkaz mezi nimi uzavřený, atd., ale zapomína na line feed (konec řádky) (kromě jiného). Tedy volání skriptu

```
http://server.cz/finger.cgi?login=kmaster%0Acat%20/etc/passwd
```

nám opět prozradí obsah souboru `/etc/passwd`. Uzavírání parametru do uvozovek také nestačí, protože samozřejmě tyto uvozovky můžeme „zevnitř“ proměnné ukončit.

Podobných útoků existuje celá řada – například pokud se proměnná, jejíž hodnotu uživatel může zvnějšku ovlivnit, používá na webserveru pro formulaci SQL dotazu, a uživatel pomocí středníku v proměnné SQL dotaz ukončí, tak za středníkem může zformulovat svůj vlastní dotaz, a oba se provedou. Podobných triků lze vymyslet nepočítaně, jejich společnou vlastností je nedostatečná validace uživatelem zadaných dat a další používání těchto dat. Správný přístup k řešení tohoto problému nespočívá v určení „nebezpečných“ znaků a ty ze zadaného řetězce eliminovat, protože hrozí, že nějaký nebezpečný bude programátorem opomenut, nýbrž v určení bezpečných znaků, a ty jediné povolit jako možné hodnoty proměnné. Tedy například v perlu:

```
my $safe = '\w\d';
my $danger = '&‘\'\\|"*?~<>^(){}\$\n\r\[\]';

if ($input =~ m/^[$safe$danger]+$/g) {
$input =~ s/([$danger]+)/\\$1/g;
} else {
        die "Bad input chars\n";
}
print "input = [$input]\n";
```

Znaky, které se vyskytují ve vstupním řetězci a v proměnné safe jsou považovány za bezpečné, a nic se s nimi nedělá. Znaky, které se vyskytují v proměnné danger jsou považovány za potenciálně nebezpečné, a proto je před ně umístěno lomeno. Vstupní řetězce, které obsahují jiný znak než který je obsažen v jedné z těchto dvou množin, je okamžitě odmítnut.

7 Podpora od systému

Není v ničích silách prečíst zdrojové kódy všech programů, které běží na jeho počítači, a opravit v nich všechny bezpečnostní chyby. Nicméně existují patche do kernelu, které útočníkům velice znesnadňují práci. Jako jeden příklad za všechny uvedu grsecurity patch, který způsobuje, že stránky paměti, ve který sídlí stack a heap, jsou označeny jako non-executable, a tudíž útočník nemůže použít výše popsané triky (nemůže si umístit do paměti svůj shellcode a pak na něj skočit,

protože procesor odmítne instrukce z těchto oblastí provádět). Dále tento patch umožňuje nastavit taková práva na adresář `/tmp`, aby bylo nemožné sledovat odtamtud symlinky na soubory nepatřící uživateli, který o to žádá, apod. Tento patch obsahuje nepřeberné množství dalších bezpečnost chránících vlastností (i například co se týče sítě). Viz `http://www.grsecurity.net/`. Dalšími patchi do kernelu, které se zabývají problematikou bezpečnosti a ochranou před špatně napsanými programy jsou `http://www.lids.org/`, `http://www.rsbac.org/`, `http://medusa.terminus.sk/` a další. Každý z těchto patchů má odlišnou filosofii a architekrutu, před aplikací je dobré prostudovat ostatní a zvážit, který se hodí pro daný systém nejlépe.

Linux a BlueTooth

Michal Semler

Email: cijoml@volny.cz

Abstrakt: Přednáška se zabývá konfigurací BlueTooth rozhraní v Linuxu.

Klíčová slova: BlueTooth, Linux.

1 Úvod

BlueTooth je moderní technologie, která si klade za cíl moderní propojení různých zařízení bez drátů. Maximální vzdálenost, na kterou mohou spolu zařízení komunikovat, je 100 metrů. V zařízeních pracujících na baterie (mobily, PDA) může být tato vzdálenost zkrácena až na 10 metrů. Tato přednáška bude navazovat na mé články publikované na serverech `abclinuxu.cz`, `root.cz` a `mobil.cz`. Zabývat se budeme novým jaderným modulem `rfcomm.o`, který je integrován v posledním BlueZ kernelu verze 2.3. Tento modul bude s největší pravděpodobností součástí jádra 2.4.21. Doporučuji kompilovat moduly proti vanilla verzi jádra 2.4.19 a kompilátorem gcc-2.95 nebo gcc-3.0.4.

2 Praxe

Budeme potřebovat tyto soubory, které nalezneme na domovské stránce projektu BlueZ (`http://bluez.sf.net`):
`bluez-kernel-2.3-pre4.tar.gz`
`bluez-libs-2.2.tar.gz`
`bluez-utils-2.1.tar.gz`

Začít musíme překladem knihoven bluez. Kompilace probíhá klasicky příkazy `./configure; make all; make install`. Poté následuje překlad utilit dle stejného scénáře a následuje překlad jaderných modulů. Po překladu knihoven zavoláme `ldconfig`, abychom zupdatovali databazi knihoven a po instalaci modulů `depmod -a` pro update databáze dostupných jaderných modulů.

K ověřování a autentifikaci zařízení slouží stále daemon hcid, který se konfiguruje stejně jako všechny předchozí verze tohoto démona. Přibylo však několik voleb.

```
cijoml@notas: > cat  /etc/bluetooth/hcid.conf
#
# HCI daemon configuration file.
#
```

Jan Kasprzak, Petr Sojka (editoři): SLT 2002 – sborník semináře o Linuxu a TEXu, str. 145–148, 2002.

```
# $Id: hcid.conf,v 1.1.1.1 2002/03/08 21:12:35 maxk Exp $
#

# HCId options
options {
        # Automaticaly initialize new devices
        autoinit yes;

        # Security Manager mode
        #   none - Security manager disabled
        #   auto - Use local PIN for incomming connections
        #   user - Always ask user for a PIN
        #
        security auto;

        # PIN helper
        pin_helper /bin/bluepin;
}

# Default settings for HCI devices
device {
        # Local device name
        #   %d - device id
        #   %h - host name
        #name "BlueZ (%d)";
        name "Linux";

        # Local device class
        class 0x100;

        # Default packet type
        pkt_type DH1,DM1,HV1;

        # Inquiry and Page scan
        iscan enable; pscan enable;

        # Default link mode
        #   none   - no specific policy
        #   accept - always accept incomming connections
        #   master - become master on incomming connections,
        #            deny role switch on outgoint connections
        #
        #lm accept,master;
        #
        lm accept,master;
```

```
        # Default link policy
        #   none    - no specific policy
        #   rswitch - allow role switch
        #   hold    - allow hold mode
        #   sniff   - allow sniff mode
        #   park    - allow park mode
        #
        #lp hold,sniff;
        #
        lp hold,sniff,park,rswitch;

        # Authentication and Encryption
        auth enable;
        encrypt enable;
}
```

Dále příkazem `modprobe rfcomm` nainstalujeme rfcomm modul do jádra a můžeme velice snadno přiřazovat zařízení, které mají být přes bluetooth obsluhovány. Toto provádíme stejnojmenou utilitou `rfcomm`:

```
rfcomm bind 0 00:80:37:51:A8:44 1&
```

Tato volba přiřadí zařízení `/dev/rfcomm0` na kanál 1 Bluetooth adaptéru a tyto volby spojí se zařízením s adresou `00:80:37:51:A8:44`. Samozřejmě musíme mít zařízení `rfcommX` přítomné v adresáři `/dev`. Když tomu tak není, použijeme jednoduše skript `create_dev` nebo příkaz `mknod` s příslušnými parametry.

Když máme vše hotovo, spárujeme zařízení buď z počítače nebo z druhého zařízení. A poté již můžeme se zařízením jednoduše přes rozhraní BlueTooth `/dev/rfcomm0` komunikovat. S mobilním telefonem jednoduše přes pppd daemona.

Velice jednoduché a funkční nastavení může být toto:

```
notas:~# cat /etc/ppp/peers/gprsbt-6310
/dev/rfcomm1 115200
cdtrcts
nodetach
noipx
noauth
connect '/usr/sbin/chat -v -f /etc/chatscripts/gprsbt'
noipdefault
ipcp-accept-local
local
novj
novjccomp
nobsdcomp
#lcp-echo-interval 10
```

```
disconnect '/usr/sbin/chat -v -f /etc/chatscripts/provider-hang'
defaultroute
#usepeerdns
lock

notas:~# cat /etc/chatscripts/gprsbt
TIMEOUT 10
ABORT        BUSY
ABORT        "NO CARRIER"
ABORT        VOICE
ABORT        "NO DIALTONE"
""
'\rATZ' OK
ATX4 OK
AT+CGDCONT=1,"IP","internet" OK
"ATDT*99#" CONNECT
```

Princip a užití HTB QoS disciplíny

Martin Devera

CDI computers, s. r. o.
Email: devik@cdi.cz

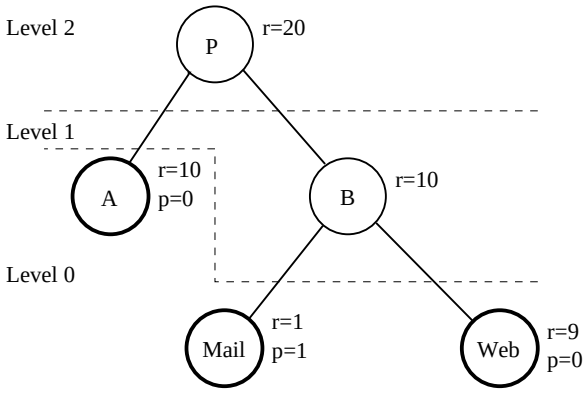
Abstrakt: Stále širší nasazení Linuxu jako routeru či firewallu zvyšuje zájem administrátorů o jeho QoS subsystém. Bohužel, uživatelé jsou často odrazeni složitostí nastavení a ne zcela zřejmými omezeními jak subsystému, tak zejména CBQ disciplíny. Tato přednáška si klade za cíl objasnit tato omezení a vysvětlit princip poměrně nové disciplíny HTB, která většinu uvedených problémů řeší.

Klíčová slova: Linux, kernel, QoS, HTB, CBQ

1 Úvodem

Tento článek popisuje impementaci nové QoS disciplíny HTB (Hierarchical Token Bucket) v Linuxu. Je předpokládáno, že čtenář je seznámen s administrací sítě v Linuxu a dokáže používat program `tc` pro nastavení Qdisců, tříd a filtrů. Tuto problematiku lze nastudovat v LARTC HOWTO [1].

2 Hierarchické sdílení



Obrázek 1. Sdílení kapacity linky

Běžný problém, který administrátor řeší, je znázorněn na obrázku 1. Provider P potřebuje rozdělit internetové pásmo mezi firmy označené A a B. Každá

Jan Kasprzak, Petr Sojka (editoři): SLT 2002 – sborník semináře o Linuxu a TEXu, str. 149–157, 2002.

firma má specifický požadavek na garantovaný datový tok r a maximální odezvy. Velikost odezvy přitom záleží na prioritě p třídy. Firma B si navíc přeje oddělit datové toky pro mailové a webové služby. Zavedeme si základní terminologii:

třída je uzlem sdílecího stromu a má definovaný požadovaný tok r,
uzel je pouze jiný název pro třídu,
list označuje uzel, který nemé žádné další následovníky, pouze listy mohou „obsahovat“ pakety ve své interní frontě,
vnitřní uzel je uzel, který není listem,
plný list je list, který obsahuje nějaké pakety,
plný vnitřní uzel je uzel s potomkem typu plný list,
podlimitním uzlem rozumíme uzel, jehož aktuální tok je menší než jeho r, o podlimitnosti rozhoduje *estimátor*,
nespokojeným uzlem nazveme uzel, který je současně plný i podlimitní – jinými slovy tento uzel má „nárok“, aby byl obsloužen,
úroveň uzlu (level) je „hloubka“ uzlu, která je pro list definována jako 0. Způsob číslování je zřejmý z obrázku 1.

Cílem QoS disciplíny je pro každé vyvolání určit, který další paket má být odeslán. Obecně vzato můžeme odeslat paket ze všech *nespokojených* listů nejlépe v pořadí priorit. Pokud takový list není, pokračujeme *plnými* listy, které si mohou „půjčit“ od svých předků.

V následujícím textu budeme sledovat, jak se která disciplína staví k problému „kdy si uzel může půjčit tok“ a jakým způsobem se určí, zda je třída pod nebo nad svým limitem.

2.1 CBQ a půjčování

Autor CBQ ukázal, že jedním z pravidel, které je nutno respektovat, je

Pravidlo 1 *Je-li* X *úroveň nejvyššího nespokojeného uzlu, není možné si „půjčit“ od uzlu s úrovní* $>$ X.

Pokud toto pravidlo nebude algoritmus respektovat, nastane situace, kdy listy s vyšší prioritou (*A* a *Web*) zahltí celou kapacitu linky a *Mail* nedostane ani svůj garantovaný tok. Při respektování pravidla 1 toto nenastane, neboť *Mail* je *nespokojeným* uzlem a ostatní si nemohou půjčovat z úrovně jiné než 0 (neboli nemohou si půjčovat vůbec).

Algoritmus CBQ implementuje toto pravidlo zavedením globální proměnné *toplevel*, která odráží stav X z pravidla 1. Proměnná je přepočítávána vždy při zafrontování nového paketu a při odebrání paketu. Proč, to je zřejmé, obě tyto akce ovlivní, zda je list *plný* či ne a definice *nespokojeného* uzlu (použitá v pravidle 1) závisí na *plnosti* listu.

Nespokojenost uzlu záleží také na aktuálním toku uzlu (zda je *podlimitní*). Uzel, který je nad limitem, se ovšem může stát *podlimitním* nejen během zafrontování či odebrání paketu, ale i po uplynutí dostatečné doby, kdy uzel nebyl využíván. Tato *asynchronní* událost není v „toplevel“ algoritmu zohledněna a

proto CBQ implementace, které „toplevel“ využívají[1], jsou zatíženy jistou chybou.

2.2 CBQ estimátor

Originální CBQ práce ([2]) nedefinuje, jaký estimátor se má použít. Avšak stejně jako téměř každá implementace používá „toplevel“ (viz odstavec 2.1), tak i ve volbě estimátoru neprokázali implementátoři příliš invence a použili stejný algoritmus navržený autory CBQ.

Tento algoritmus detekuje limitní stav třídy měřením doby, která uplyne mezi odesláním dvou za sebou následujících paketů (patřících měřené třídě). Čas se porovná s očekávanou dobou (vypočítanou z požadované rychlosti) a znaménko získané odchylky[2] určuje, zda je třída nad či pod limitem.

Pomineme-li složitost měření mezipaketové mezery s dostatečnou přesností (řádově mikrosekundy), zjistíme, že tento algoritmus potřebuje ke své funkci znát přesnou rychlost fyzického rozhraní. Není problém určit rychlost Ethernetového rozhraní, ale je to nemožné u zařízení jako je Wireless LAN nebo softwarová zařízení (tunely). Podobná zařízení totiž nemají pevnou rychlost odesílání dat ale jejich aktuální rychlost kolísá a závisí na mnoha podmínkách (kvalita signálu, zatížení routeru...). Navíc nastavení takového estimátoru není vůbec jednoduchá záležitost a autoři CBQ sepsali dokument [3], který pojednává výhradně o tom, jak estimátor nastavit.

3 HTB

Výše uvedené nesnáze byly prvotním impulzem pro vývoj nové disciplíny, která nevykazuje žádný podobný problém. Vývoj prošel během dvou let třemi fázemi, během každé byl algoritmus kompletně přepsán. Následující řádky se týkají verze 3.

3.1 HTB estimátor

HTB dělí třídy nejen na *podlimitní* a *nadlimitní*. Místo toho budeme stav z hlediska aktuální velikosti toku označovat barvami semaforu. Každá třída má kromě garantovaného toku r i maximální tok c[3].

Nyní nadefinujeme barvu třídy pro daný časový okamžik t a aktuální tok $a(t)$ jako

zelenou pokud platí $a(t) < r$, tedy třída je pod limitem,
žlutou pokud $c \geq a(t) \geq r$, taková třída je sama nad limitem, ale nedosáhla stropu, tudíž si může zkusit půjčit od rodiče, nebo

[1] Veškeré implementace, které mi jsou známé, využívají „toplevel“.
[2] Ve skutečnosti se odchylka ještě exponenciálně průměruje, ale to není pro nás podstatné.
[3] Angl. **c**eil = strop.

červenou pro $a(t) > c$. Červená třída přesáhla strop, a proto již nemůže odeslat žádný paket.

Implementace estimátoru využívá tzv. Leaky Bucket (Dále jen LB). Výhodou je tolerance vůči rozlišení systémového časovače a zejména nezávislost na fyzické rychlosti rozhraní. Jak vidíme, použití LB řeší oba problémy z odstavce 2.2. LB je definován dvěma parametry, datovým tokem r (v bajtech za sekundu) a *burstem* (v bajtech). Běžně má takový datový tok rychlost r. Pokud je ovšem $a(t) < r$, pak si LB „zapamatuje“, kolik bajtů bylo nevyužito. Maximální velikost této „paměti“ je právě *burst*. Pokud je požadavek na přenesení dat nad limit r, LB v každou chvíli umožní přenést až tolik bajtů bez omezení rychlosti, kolik jich je právě „zapamatováno“.

Je dobré si uvědomit, že právě *burst* je klíčem k nezávislosti LB na rozlišení systémového časovače. Uveďme pouze, že pokud je rozlišení časovače Δt (10ms v Linuxu), musí pro *burst* platit

$$burst \geq \Delta t \times r \tag{1}$$

Utilita `tc` pro HTB s tím počítá, a pokud *burst* neuvedete, nastaví automaticky nejmenší možnou hodnotu pro daný systém podle (1).

Pro implementaci stropu (c) musí mít každá třída přiřazeny dva LB, jeden pro garantovaný tok r a druhý pro c. Abychom se v textu lépe vyznali, budeme *burst* příslušný ke „stropovému“ LB označovat *cburst*.

3.2 HTB hierarchie a půjčování

Definujme algoritmus pro výběr dalšího paketu k odeslání.

Algoritmus 1 *Ze všech* plných *listů volme takový, který by si při odesílání paketu mohl půjčit tok od rodiče na nejnižší úrovni. Pokud je takových listů více, vybereme ten s nejvyšší prioritou. Pokud máme stále více listů, střídáme je pravidelně podle poměrů jejich* r.

Uvědomte si prosím, že list si může korektně půjčit sám od sebe! Algoritmus 1 splňuje pravidlo 1 ze strany 150. Důkaz je triviální. Krom toho algoritmus 1 splňuje očekávání, která na podobný systém klademe:

- Toky tříd se stejnou prioritou si půjčují tok od společného rodiče v poměru jejich r.
- Třída s vyšší prioritou získá tok od společného rodiče přednostně.
- Třída s vyšší prioritou má kratší odezvy.
- Garantované toky jsou vždy splněny.

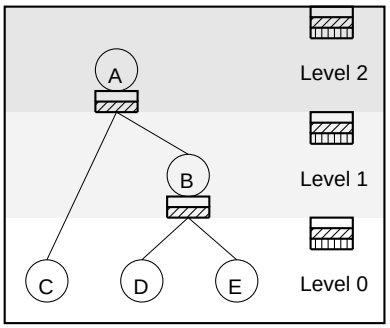
3.3 Implementace výběru paketu

Nyní zůstává pouze otázka, zda a jak lze algoritmus 1 efektivně implementovat. Během výběru paketu musíme najít *zelenou* třídu s nejnižší úrovní, která má jako potomka plný list a k potomkovi „vede“ cesta se *žlutými* třídami (včetně listu).

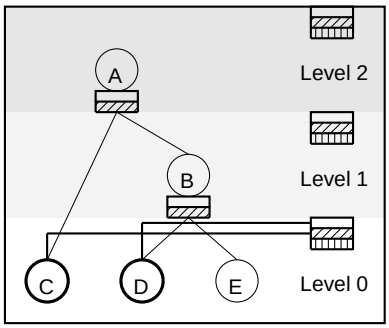
Není možné toto hledání provést průchodem přes všechny třídy během výběru paketu, neboť složitost by byla $\mathcal{O}(N)$, kde N je počet tříd, a to je neúnosné.

Budeme tedy muset udržovat informace o aktuálním stavu a inkrementálně je upravovat. Podívejme se nyní na obrázek 2. Pro účel vysvětlení algoritmu jsme pozměnili označení tříd. List je znázorněn tučnou kružnicí, pokud je *plný*, a vnitřek kruhu je prázdný, pokud je třída *zelená*, jednoduše šrafovaný, pokud je *žlutá*. Dvojitě šrafovaná je *červená* třída.

Každý vnitřní uzel si udržuje seznam potomků, kteří by si od něj rádi půjčili nějaký tok. Podle definice to mohou být pouze plné *žluté* třídy. Tento seznam (tzv. *interní půjčovací seznam*) musí být veden zvlášť pro jednotlivé priority, neboť uzel může být předkem listu s libovolnou prioritou. Naše příklady mají pouze dvě priority, vysoká s interním seznamem znázorněným prázdným obdélníčkem pod uzlem a nízká (šrafovaný obdélníček o něco níže).



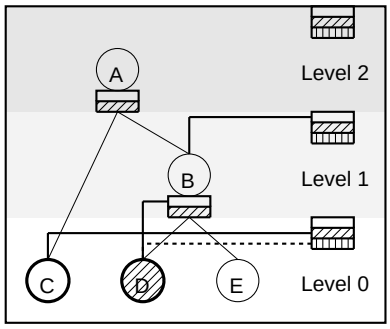
Obrázek 2. Žádné pakety, třídy jsou *zelené*, pouze D ma vyšší prioritu



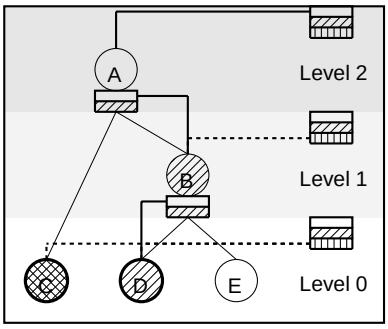
Obrázek 3. Přišly pakety pro C a D

V pravé části obrázku vidíme podobné obdélníčky. Ignorujme prozatím nejspodnější (svisle šrafovaný). Dva horní tvoří tzv. *globální půjčovací seznam*. Seznam obsahuje třídy určité úrovně a priority, které jsou *plné* a zároveň *zelené* (tedy *nespokojené*). Jinými slovy, třídy, které jsou na jakémkoli globálním seznamu, jsou připraveny odeslat paket. Nyní tedy v souladu s algoritmem 1 stačí vybrat nejspodnější úroveň s neprázdným globálním seznamem, zde seznam s nejvyšší prioritou, a sledovat interní seznamy, které nás dovedou až k listu, který má odeslat další paket.

Obrázek 2 ukazuje stav, kdy v HTB není žádný paket a všechny třídy jsou *zelené*. Pokud přijdou pakety určené listům C a D, listy se změní na *plné* a protože jsou zatím i *zelené* (jsou pod limitem), můžeme je připojit do globálních seznamů pod patřičné priority (obrázek 3). Pokud by si síťová karta nyní řekla o paket, velmi rychle zjistíme, že máme poslat z listu D a případný další z C (pokud D již bude *prázdný*). Podívejme se, co by se stalo, pokud tedy pošleme paket z D a estimátor rozhodne, že D je *žlutá* (došlo k překročení r). Tento stav vidíme na obrázku 4. List D byl odpojen od globálního listu, neboť již nemůže



Obrázek 4. D je nyní *žlutá*



Obrázek 5. C je *červená* a zároveň B je teď *žlutá*

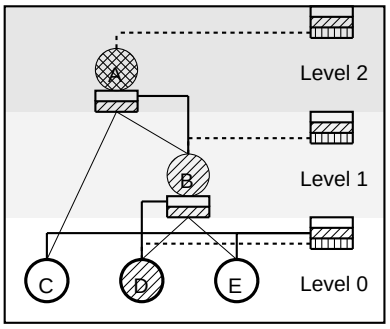
odesílat bez půjčky, a zároveň byl zařazen na vysokoprioritní interní seznam třídy B. Třída B sama je zařazena na globální list úrovně 1 vysoké priority. To proto, aby třída D mohla nadále posílat pakety výpůjčkami od B.

Protože pokud D po nějakou dobu neodešle paket, tak se její barva v určitém čase změní opět na *zelenou*, zařadíme D na speciální *čekací seznam*. Každá úroveň má svůj čekací seznam a zařazení je naznačeno čárkovanou čarou. Po uplynutí předem spočítaného času se změní barva třídy. Čekací seznam řeší problém CBQ, popsaný v odstavci 2.1.

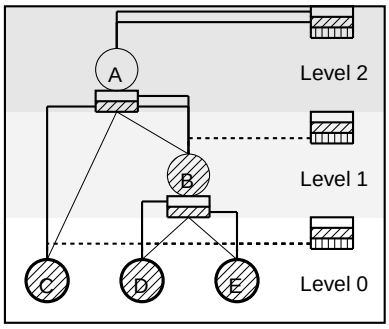
Zde si ale povšimněme důležitého faktu – při žádosti o další paket bude vybrána třída C (nikoli D), a to i přesto, že D má vyšší prioritu. Je to proto, že v globálním seznamu začíná cesta k C na nižší úrovni. Tento krok je vlastně implementací pravidla 1 ze strany 150. Proč by měla dostat D přednost, když C má nárok na svůj garantovaný tok.

Obrázek 5 zobrazuje situaci po odeslání paketu z C. Řekněme, že tato třída překročí c a je tudíž *červená*. Navíc B je nyní *žlutá*. Vidíme, jak se B připojí na interní seznam A, a místo B je v globálním seznamu nyní A. Toto již známe. Nové je pro nás chování *červené* třídy. Jak vidno, C je pouze na čekacím seznamu (nemůže sama poslat paket ani si půjčit). Jediná třída, která může posílat, je D, jak je zřejmé z globálního seznamu. Na obrázku 6 vidíme situaci, kdy A je *červená*, a C i E jsou *plné* a *zelené*. Zde lze pozorovat, že cesta po interních seznamech existuje, i když nejvyšší třída nemá žádný spoj na globální seznam. Navíc C i E jsou připojeny ke společnému globálnímu seznamu. V tomto případě je nutno pravidelně střídat odebírání paketu mezi C a E v poměru jejich r. To je v implementaci zajištěno implementací seznamů pomocí *RB-tree,* kde klíčem je *classid* třídy. Pro seznam si HTB pamatuje *classid*, které bylo naposledy použito, a díky perzistenci *classid* je střídání férové i při změnách barev a tedy i obsahu seznamů. Váhování podle r je řešeno pomocí *DRR* [4].

Obrázek 7 poukazuje na možnost napojení více tříd na společné interní seznamy. Nebýt vyšší priority D, byli bychom nuceni střídat odebírání paketu mezi všemi listy. Neboť D ale vyšší prioritu má, bude z globálního seznamu vybrána pouze D.



Obrázek 6. A je *červená*, a C i E jsou *plné* a *zelené*



Obrázek 7. Všechny listy si půjčují od A

4 Praktické použití HTB

HTB je již vyzrálý systém a v současnosti je vložen ve verzích Linuxového kernelu 2.4.20pre a posledních kernelů řady 2.5. Upravený nástroj `tc` by měl být k dispozici koncem roku 2002 v oficiálním balíku `iproute2`, do té doby je nutno stáhnout `tc` z `http://luxik.cdi.cz/~devik/qos/htb/`. Je součástí TAR s poslední verzí HTB.

Pro detailní vysvětlení syntaxe a nuancí nastavení viz. manuál na výše uvedené adrese.

4.1 Jednoduchá konfigurace

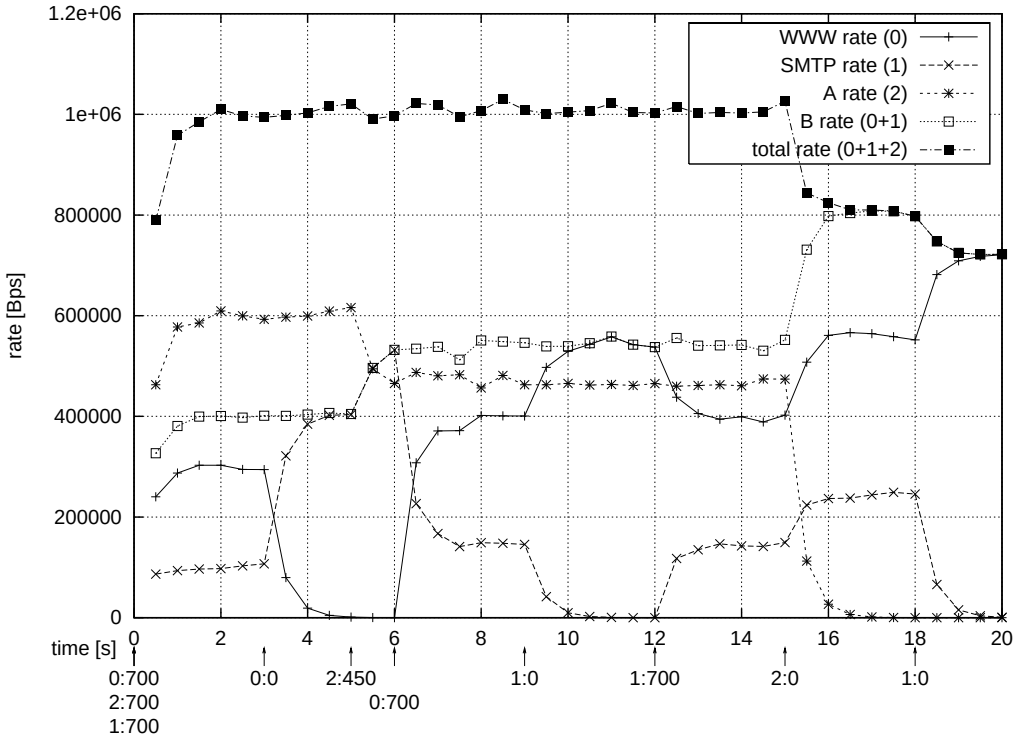
Ukážeme si triviální konfiguraci a její výsledné toky. Zadáme tyto příkazy

```
tc qdisc add dev eth0 root handle 1: htb default 12

tc class add dev eth0 parent 1: classid 1:1 htb \
   rate 1000kbps ceil 1000kbps
tc class add dev eth0 parent 1: classid 1:2 htb \
   rate 1000kbps ceil 800kbps
tc class add dev eth0 parent 1:2 classid 1:10 htb \
   rate 300kbps ceil 1000kbps prio 1
tc class add dev eth0 parent 1:2 classid 1:11 htb \
   rate 100kbps ceil 1000kbps prio 1
tc class add dev eth0 parent 1:1 classid 1:12 htb \
   rate 600kbps ceil 1000kbps prio 1

tc filter add dev eth0 protocol ip parent 1:0 prio 1 u32 \
   match ip src 1.2.3.4 match tcp dport 80 0xffff flowid 1:10
tc filter add dev eth0 protocol ip parent 1:0 prio 1 u32 \
   match ip src 1.2.3.4 flowid 1:11
```

První řádek vytvoří instanci disciplíny HTB, další čtyři řádky sestaví hierarchii podobnou té na obrázku 1. Webové pakety z adresy 1.2.3.4 půjdou do listu 1:10, ostatní pakety z adresy 1.2.3.4 do 1:11 a ostatní do 1:12 (to zajistí *default 12* z první řádky). Výsledný tok je na obrázku 8. Kódy zobrazené pod grafem



Obrázek 8. Výsledný datový tok

indikují místa, kde se změnil vstupní datový tok. Např. 0:700 znamená, že třída 0 (1:10) má od té chvíle vstupní tok 700 kbytes.

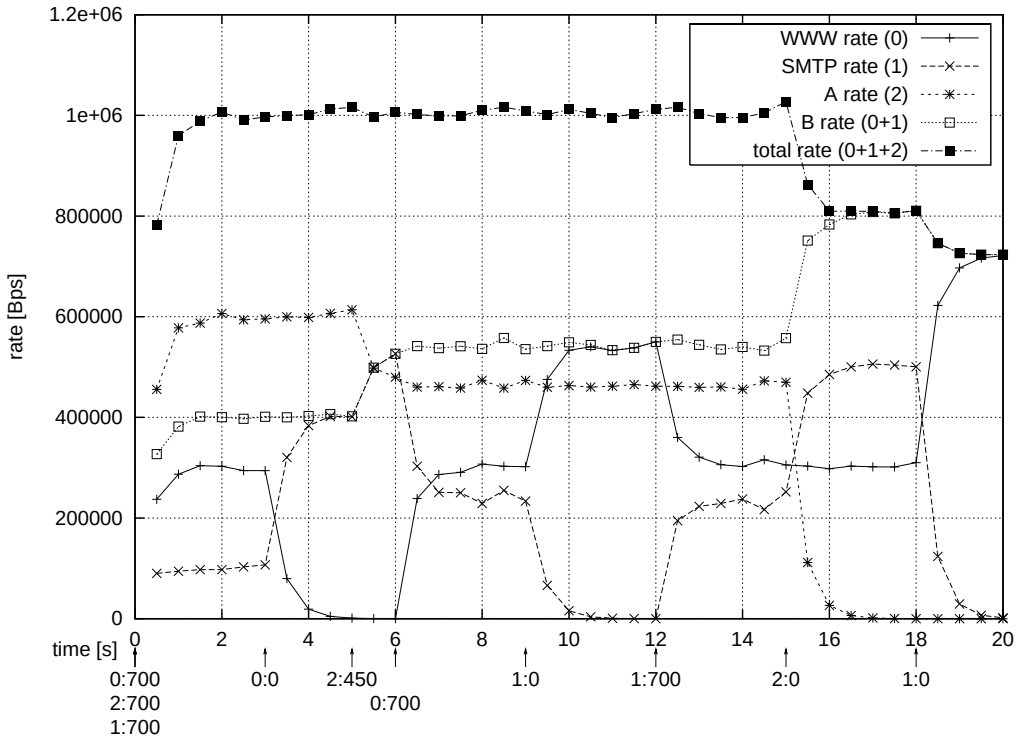
Můžeme sledovat, jak postupné vypínání jednotlivých paketových zdrojů způsobí přerozdělení toků. V čase 3 s jsme vypnuli tok 0 a z grafu je zřejmé, že jeho pásmo si zabral tok 1. Proč kousek nezískal i tok 2? Protože uzel B (rodič tříd pro toky 0 a 1) má garantováno 400 kbps.

V čase 8s využívá tok 2 pouze 450 kbps, zbývajících 150 kbps si rozdělily toky 0 a 1 v poměru jejich r. Pokles celkováho toku na 16 s je způsoben hodnotou *ceil* = 800 kbps u třídy 1:2 (uzel B).

Ukážeme si ještě, co se stane, když zvýšíme prioritu třídy 1:11 pomocí

```
tc class change dev eth0 parent 1:2 classid 1:11 htb \
   rate 100kbps ceil 1000kbps prio 0
```

(nižší číslo u *prio* znamená vyšší prioritu). Na obrázku 9 je znatelný pokles toku 0 v časech 8 s a 14 s – prostor získaný poklesem toku 2 si nyní vzala kompletně SMTP třída 1 a pro 0 zůstává pouze její garantovaný tok.



Obrázek 9. Datový tok po zvýšení priority u SMTP

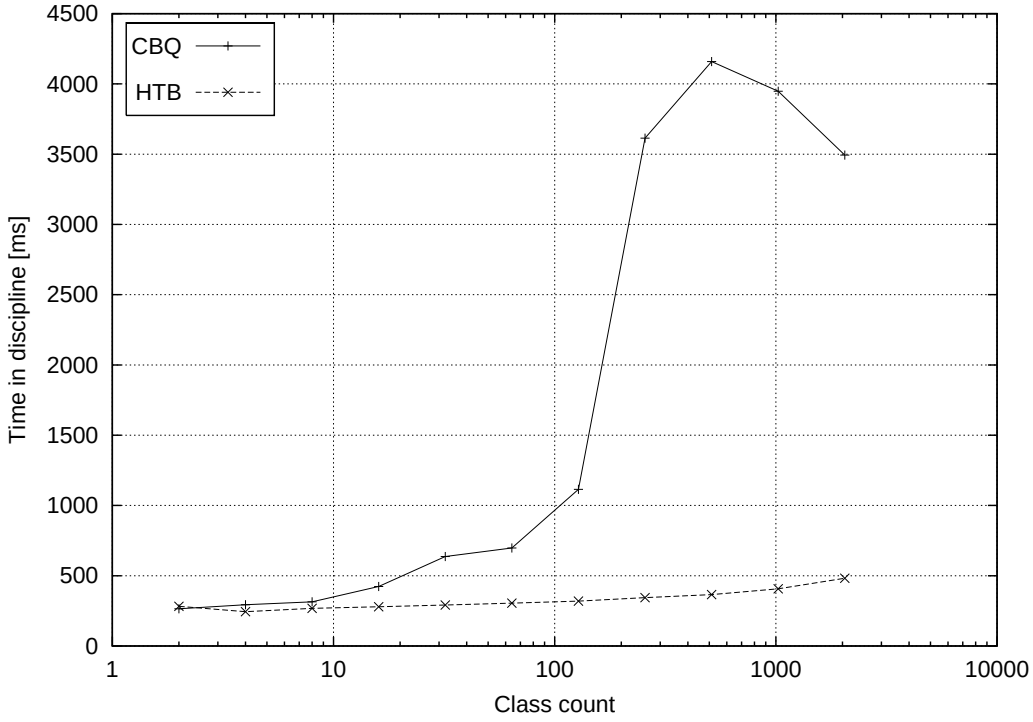
5 Výsledky

HTB bylo podrobeno mnoha testům výkonu i stability. Z měření výkonu vyplývá, že HTB škáluje dobře i nad 5000 plných tříd v jediné konfiguraci a jeho rychlost je přitom stále vyšší než u CBQ. Na grafu 10 je jasně vidět srovnání škálovatelnosti HTB a CBQ. Zploštění křivky u CBQ u hranice 200 tříd je způsobeno nedostatkem CPU cyklů.

Vezmeme-li v úvahu i čerstvé zařazení HTB do Linuxového kernelu a jednoduchost nastavení z hlediska uživatele, je pravděpodobné, že HTB plně nahradí CBQ disciplínu.

Reference

1. `http://www.lartc.org`.
2. Sally Floyd, Van Jacobson, *Link-sharing and Resource Management Models for Packet Networks*, IEEE/ACM Transactions on Networking, Vol. 3 No. 4, August 1995.
3. Sally Floyd, *Notes on Class-Based Queuing: Setting Parameters*, LBNL, February 1996.
4. M. Shreedhar, George Varghese, *Efficient Fair Queuing using Deficit Round Robin*, Washington University, October 1995.



Obrázek 10. Závislost spotřebovaného CPU času pro 5 sekundový testovací běh na 30 MBit

Java a Linux

Petr Adámek

Fakulta informatiky, Masarykova univerzita, Brno
Email: petr.adamek@bilysklep.cz

Abstrakt: Přednáška se věnuje problematice vývoje a provozu aplikací platformy Java v prostředí operačního systému Linux, od jednoduchých aplikací až po složité informační systémy. Stručně seznámí posluchače také s dostupnými nástroji a upozorní na případné problémy. Cílem přednášky je podat co nejobecnější přehled. Měla by být přínosem pro všechny potenciální programátory, administrátory i uživatele.

Klíčová slova: Java, Java 2, JVM, garbage collector, J2EE, JSP, Enterprise JavaBeans, Tomcat, JBoss.

1 Úvod

Jazyk Java[1] je jedním z moderních prostředků pro tvorbu aplikací a v současnosti je používán v širokém spektru aplikací, od apletů vkládaných do stránek WWW, přes GUI aplikace až po podnikové informační systémy. V poslední době proniká i do oblasti mobilních zařízení, kde je díky svým vlastnostem docela úspěšný. Jeho oblíbenost stále roste, řady programátorů se rozšiřují, objevují se nové a zajímavé aplikace, roste i komunita jejich uživatelů.

Tato přednáška se snaží podat obecný přehled o vlastnostech platformy Java a chce pomoci se základní orientací všem zájemcům o provoz či vývoj aplikací v Javě. Je orientována na nasazení v prostředí operačního systému Linux, nicméně díky přenositelnosti a platformní nezávisloti Javy je možné většinu informací využít i na ostatních platformách.

2 Jazyk Java a jeho vlastnosti

Jazyk Java je objektově orientovaný programovací jazyk pro všeobecné použití. Původně se jmenoval Oak a byl navržen pro použití ve spotřební elektronice. Po několika letech vývoje došlo v roce 1996 ke změně orientace směrem k internetu, jazyk prošel revizí a byl přejmenován. Vývoj pokračoval a v roce 2000 byla uvolněna specifikace nové verze jazyka s číslem 1.2, nazvaná *Java 2* [1]. Od té doby probíhá další vývoj vlastního jazyka i souvisejících technologií a specifikací. Všechny specifikace jsou otevřené a na jejich vývoji se podílí celá řada firem i jednotlivců.

[1] http://java.sun.com

Jan Kasprzak, Petr Sojka (editoři): SLT 2002 – sborník semináře o Linuxu a TEXu, str. 159–168, 2002.

Základní filozofií jazyka Java je poskytnout programátorovi maximum prostředků pro snadný a efektivní vývoj aplikací. Cílem je odstranit nebo minimalizovat nejčastější překážky, které mu brání zaměřit se pouze na řešení svého problému. Tohoto cíle se snaží dosáhnout pomocí několika prostředků, které jsou popsány v této kapitole.

Mezi klíčové vlastnosti jazyka Java patří:

- Platformní nezávislost a přenositelnost aplikací.
- Robusnost a stabilita. Robusnost platformy Java zajišťují *garbage collector*, *absence ukazatelové aritmetiky*, robusní *systém výjimek*, přísná *typová kontrola* a *bezpečnostní model*.
- Robusní podpora vícevláknových aplikací včetně synchronizačních prostředků a individuálního řízení přístupu pro každé vlákno zvlášť (viz bezpečnostní model).
- Podpora pro různé jazyky, kódování, národní zvyklosti apod. Jazyk Java používá kódování Unicode a při vstupních a výstupních operacích dochází k automatickému překódování do požadované znakové sady.
- Rozsáhlá knihovna standardních tříd.

Některé důležité vlastnosti jsou podrobněji rozebrány dále.

2.1 Virtuální stroj

Program v jazyce Java je kompilován do speciálního instrukčního souboru, který se nazývá *bytecode*. Tento kód je pak standardně vykonáván prostřednictvím tzv. *virtuálního stroje*[2] [2].

Tímto způsobem je zajištěna *binární přenositelnost* aplikací na úrovni tohoto instrukčního souboru. Pro přenos aplikace na konkrétní platformu postačuje, pokud pro ni existuje JVM a standardní knihovna objektů[3]. Je také snadné kód aplikace distribuovat např. prostřednictvím počítačové sítě bez ohledu na cílovou platformu[4].

Platformní nezávislost je ovšem řešena mnohem komplexněji a obsahuje i takové detaily, jako je oddělovač adresářů (/ versus \) apod.

Koncepce virtuálního stroje má pochopitelně i jednu slabší stránku, a tou je ztráta výkonu ve srovnání s aplikací v nativním strojovém kódu. Jak bude ale vysvětleno později, nemusí to být zase až tak velký problém, jak se na první pohled zdá.

[2] Označuje se zkratkou JVM, což znamená Java Virtual Machine.

[3] Protože bývá standardní knihovna objektů z velké části sama implementována v jazyce Java, většinou při její portaci na novou platfomu stačí upravit sadu několika základních nízkoúrovňových metod, které jsou implementovány v jazyce C.

[4] Java umožňuje vytvářet hierarchii uživatelských zavaděčů tříd, což možnosti distribuce tříd dále rozšiřuje.

2.2 Správa paměti

Programátor se nemusí starat o uvolňování použité paměti, neboť tuto činnost zajistí garbage collector (sběrač odpadků) automaticky[5]. Místo ukazatelů se používají *reference*, ukazatelovou aritmetiku plně nahrazují pole a rozsáhlá množina různých typů kolekcí. Použití prázdné reference s hodnotou null nebo překročení rozsahu pole způsobí vygenerování výjimky. Všechny proměnné jsou při vytvoření automaticky inicializovány na příslušnou implicitní hodnotu.

Co to znamená? Pokud aplikace v Javě zhavaruje s chybou SIGSEGV, je to chyba JVM, některé z nízkoúrovňových nativních metod, případně je problém na straně hardware. Pokud se objeví chyba kdekoliv v kódu jazyka Java, program skončí výjimkou s hlášením o příčinách chyby.

2.3 Výjimky

Jazyk Java poskytuje poměrně propracovaný systém výjimek, který je důsledně využíván i ve standardní knihovně objektů. Programátor může výjimky ošetřovat i vyvolávat, může používat existující výjimky, nebo hierarchii výjimek rozšiřovat o svoje vlastní.

Pokud kód nějaké metody může vyvolat výjimku, musí ji ošetřit, nebo musí mít metoda ve své deklaraci vyznačeno, že může tuto výjimku generovat[6]. Tím je zajištěno, že programátor na žádnou důležitou výjimku nezapomene.

2.4 Typová kontrola

Jazyk Java je přísně typový jazyk a obsahuje dva druhy typů. *Primitivní datové typy* reprezentují čísla, znaky a logickou hodnotu (viz tabulka 1). Proměnné tohoto typu obsahují přímo příslušnou hodnotu a přiřazením se hodnota zkopíruje[7]. *Referenční datové typy* obsahují referenci na objekt nebo pole. Proměnná tohoto typu obsahuje vždy referenci a přiřazením se objekt nekopíruje. Pokud je potřeba pracovat s kopií objektu, musí se objekt zkopírovat explicitně.

Typová kontrola je prováděna již během překladu, při načítání tříd do paměti VM se však provádí znovu. Příčinou je riziko podvržení nebezpečného kódu, ať již úmyslně či chybou překladače. Může také dojít ke změně rozhraní a metoda, která dříve existovala, již neexistuje nebo není přístupná.

Není možné provést přetypování mezi nekompatibilními typy. Pokud se o to pokusíme za běhu programu, je generována výjimka.

[5] Uvolňování paměti na nesprávném místě je spolu s nesprávným použitím ukazatelové aritmetiky nejčastějším zdrojem obtížně lokalizovatelných chyb způsobujících pád programu.

[6] Toto neplatí pro výjimky, které se mohou vyskytnout na kterémkoliv místě, tj. výjimky JVM, dělení nulou, pokus o použití reference s hodnotou null apod.

[7] Pro každý primitivní typ existuje i jeho objektový protějšek, např. primitivnímu typu int odpovídá objektový typ Integer. Je-li třeba pracovat s referencemi na některý z primitivních typů, stačí jej nahradit příslušným typem objektovým.

Tabulka 1. Primitivní typy jazyka Java

typ	bitů	rozsah
byte	8	$\langle -2^7, 2^7 \rangle$
short	16	$\langle -2^{15}, 2^{15} - 1 \rangle$
int	32	$\langle -2^{31}, 2^{31} - 1 \rangle$
long	64	$\langle -2^{63}, 2^{63} - 1 \rangle$
float	32	24 bitů mantisa, 8 bitů exponent
double	64	53 bitů mantisa, 11 bitů exponent
char	16	\u0000 – \uFFFF
boolean	1	{*false, true*}

2.5 Bezpečnostní model

Bezpečnost je jedním z rysů, na které byl při vývoji jazyka Java kladen velký důraz. Java má implementován velmi sofistikovaný bezpečnostní model, který umožňuje řídit přístup jednotlivých vláken k různým prostředkům, objektům, metodám i dalším zdrojům. Tento model je uplatňován např. u apletů, ale může být samozřejmě využit kdekoliv.

Součástí bezpečnostního modelu je samozřejmě verifikace kódu nahrávaných tříd a rozsáhlá knihovna pro kryptografii.

3 Verze a edice Javy

Jak již bylo zmíněno v úvodu, v současné době je aktuální verze jazyka nazývaná Java 2. Jde v podstatě o souhrn specifikací, které jsou rozděleny do tří základních edicí:

- Java 2, Standard edition (J2SE)[8]
- Java 2, Enterprise edition (J2EE)[9]
- Java 2, Micro edition (J2ME)[10]

Implementace JVM, kompilátorů, nástrojů, knihoven atd. dle těchto specifikací může dodávat jakýkoliv dodavatel, pokud dodrží příslušné licenční podmínky. Není tedy nutné používat výhradně implementace od firmy Sun Microsystems, Inc., ta ostatně ani některé neposkytuje. Existuje i množství open source implementací, některé z nich budou popsány dále.

3.1 Java 2, standard edition

Tato edice poskytuje specifikace základních nástrojů, potřebných pro vývoj a provoz běžných aplikací. Jde zejména o specifikaci jazyka Java, JVM a základní knihovny tříd. Umožňuje vyvíjet GUI aplikace, konzolové aplikace, aplety apod.

[8] `http://java.sun.com/j2se/`
[9] `http://java.sun.com/j2ee/`
[10] `http://java.sun.com/j2me/`

Aktuální je verze 1.4, která obsahuje např. awt (knihovna pro GUI), podporu pro V/V operace, matematickou knihovnu s podporou velkých čísel, knihovnu síťových funkcí včetně SSL, RMI (vzdálené volání metod), bezpečnostní framework, JDBC (přístup k databázím), velkou množinu kolekcí, knihovnu pro kryptografii, knihovnu pro práci s obrázky v různém formátu, JNDI (adresářové služby), implementaci normy CORBA, knihovnu pro praci se zvukem, swing (rozšířená knihovna pro GUI), knihovnu pro tisk, knihovnu pro práci s XML a další.

3.2 Java 2, enterprise edition

Tato edice poskytuje specifikace nástrojů pro vývoj podnikových aplikací. Jde zejména o dynamické WWW stránky a robusní, distribuované a škálovatelné informační systémy. Aktuální je verze 1.3, nicméně specifikace verze 1.4 budou uvolněny velmi brzy.

Tvorbu dynamických stránek umožňuje technologie JavaServlets a JavaServer Pages (JSP). Servlety jsou objekty, které generují dynamický obsah WWW stránek. Stránky JSP obsahují HTML kód, který může být kombinován s kódem v jazyce Java. Stránky JSP mohou také obsahovat uživatelsky definované značky elementů, které umožňují elegantním způsobem do stránek vkládat dynamický obsah nebo vykonávat libovolný kód. To umožňuje, aby programátor naprogramoval sadu takovýchto značek a JSP stránky může s jejich pomocí vytvářet webdesigner bez jakékoliv znalosti programování či vnitřní architektury programu.

Pro tvorbu informačních systémů lze s úspěchem využít technologii Enterprise JavaBeans [5]. Jde o standardní komponentovou architekturu pro návrh a vývoj komponentově orientovaných distribuovaných podnikových aplikací. Umožňuje vytvářet škálovatelné, transakční a bezpečné aplikace pro víceuživatelská prostředí.

Součástí J2EE jsou i některé další knihovny, jako JavaMail pro podporu elektronické pošty, JTA a JTS pro transakce, JMS pro asynchronní výměnu zpráv apod.

Jazyk Java se jeví jako velmi vhodný nástroj pro vývoj informačních systémů a podnikových aplikací, neboť platforma J2EE poskytuje skutečně úctyhodnou podporu pro jejich tvorbu. Programátor se může zaměřit na aplikační logiku a technologie J2EE zajistí vše okolo včetně řízení přístupu uživatelů, transakcí, perzistence, výměny asynchronních zpráv, řízení životního cyklu objektů, transparentní aktivace a pasivace objektů, vzdáleného přístupu k objektům atd. Je snadné vyvíjet aplikace ve velkých i malých týmech, v případě potřeby je možné změnit aplikační server či platformu beze změny kódu, vše je standardizované a aplikace se snadno udržují.

Pozor! Sun Microsystems, Inc. dodává J2EE SDK, nicméně toto SDK může být použito pouze pro studijní, vývojové nebo demonstrační účely. Z licenčních i technických důvodů nesmí být použito pro ostré nasazení. K tomuto účelu je nutné použít některou z dostupných implemetací, např. JBoss (viz kapitola 5.7).

3.3 Java 2, micro edition

Tato edice je určena pro mobilní zařízení, jako jsou mobilní telefony a kapesní počítače (PDA). Ačkoliv jde o oblast jistě velmi zajímavou, překračuje rámec této přednášky.

4 Java a systémové zdroje

Jedním z nejrozšířenějších mýtů o Javě jsou její vysoké nároky na spotřebu paměti a strojový čas procesoru. Jak je tomu ve skutečnosti se pokusí nastínit tato kapitola.

4.1 Rychlost programů

Je pravda, že díky virtuálnímu stroji nemůže dosáhnout výkonu aplikací přeložených do strojového kódu procesoru. Na druhé straně však nejnovější verze virtuálního stroje s HotSpot kompilátorem obsahují velmi kvalitní optimalizace. Rozdíly v rychlosti jsou tudíž velmi malé.

Problémem však zůstává zavádění tříd do paměti JVM. Tento proces se totiž skládá z několika kroků. Nejdříve musí být třída nalezena a případně extrahována z archivu jar. Pak probíhá linkování, kontrola integrity kódu, kontrola přístupových práv, vytváření statických proměnných, kontrola symbolických referencí a jejich nahrazování referencemi přímými (podrobný popis celého procesu lze nalézt např. v [2]).

Nezanedbatelné zpoždění tedy generuje zavádění tříd do paměti, typicky při startu JVM. Ačkoliv se většina JVM snaží třídy do paměti zavádět až v okamžiku prvního přístupu, většinou se mnoho tříd musí zavést právě na začátku. Nebo v okamžiku kdy je poprvé přistoupeno k většímu programovému celku. To jsou ony okamžiky, kdy se aplikace v Javě jeví jako velmi pomalá, neboť v podstatě stojí.

4.2 Spotřeba paměti

I v oblasti paměťových nároků Java nebývá vždy skromná. Jednou z příčin je způsob práce správce paměti JVM. Ten totiž alokuje určitý blok paměti, který dále přerozděluje podle potřeby aplikace. Když má program paměti nedostatek, správce alokuje blok další. Po uvolnění paměti garbage collectorem však uvolněnou paměť systému nevrátí. Vlastně ani nemůže, neboť by musel vrátit celý blok, což není možné, pokud v něm zůstává byť jen jediný objekt.

Taktéž garbage collector má jistou režii způsobenou tím, že nepotřebné objekty neuvolňuje okamžitě. Ta je však ve srovnání s ostatními vlivy téměř zanedbatelná.

Další prostor okupuje kód tříd zavedených do paměti. Vzhledem k tomu, že jenom standardní runtime knihovna tříd prostředí Java 2 SE verze 1.4 zabírá na disku cca 24 MB, rozhodně nejde o zanedbatelnou položku. Pokud navíc na

jednom počítači běží více instancí JVM (např. pro více uživatelů či více aplikací), každá JVM má do paměti zavedenou vlastní kopii hierarchie tříd. Neprojeví se tedy výhoda sdílení kódu v paměti, běžná u standardních linuxových aplikací.

Zde se objevují stinné stránky koncepce jazyka s VM a garbage collectorem, která neumí plně využít výhod paměťového subsystému hostitelského operačního systému. Je to daň za bezpečnost a robusnost jazyka, nicméně tato daň není zase až tak velká.

4.3 Řešení

Jak řešit popsané problémy s nadměrnou spotřebou systémových zdrojů? Nejdříve je nutné si uvědomit, zdali tento problém vůbec existuje. Zejména v oblasti serverových aplikací (typycky J2EE) nás problém pomalého startu či existence kopie hierarchie tříd pro každou JVM příliš netrápí. Na serveru běží většinou trvale jedna JVM a jednotlivé uživatelské požadavky jsou obsluhovány jednotlivými vlákny.

V ostatních případech se nabízí několik řešení (případně jejich kombinace):

- Provést upgrade hardware. Často je to nejefektivnější a nejlevnější řešení.
- Použít JVM optimalizovaný pro zamýšlené použití, případně jeho chování ještě přizpůsobit. Typicky lze volit mezi JVM pro serverové a JVM pro klientské aplikace (viz `man java`).
- Použít efektivnější verzi knihovny nebo programu. Případně ze svého kódu odstranit neefektivní části a zamyslet se, jestli by kritické úseky nešly implementovat efektivněji. I v Javě je možné psát neefektivní a pomalé programy.
- Kritické úseky kódu implementovat v JNI[11]. Například vývojové prostředí Eclipse[12] nepoužívá pro GUI standardní knihovnu Javy, ale pomocí JNI používá pro linuxovou platformu GTK.
- Použít překladač, který překládá program v Javě přímo do nativního strojového kódu procesoru. Je nutné si ale uvědomit, že dojde ke ztrátě výhod přenositelnosti. Takovým překladačem je například `gcj`[13], který je součástí kolekce GNU překladačů gcc. Bohužel zatím není ve stavu, kdy by byl reálně použitelný, především kvůli absenci úplné implementace runtime knihovny základních tříd[14].
- Mít spuštěnou pouze jednu instanci JVM, sdílenou mezi všemi uživateli. Správa přístupových práv je pak delegována na bezpečnostní model JVM.
- Používat operační systém založený na jazyce Java. Existují snahy o vytvoření operačního systému, který je tvořen JVM a který má všechny aplikace implementované v Javě[15].

[11] JNI je rozhraní pro implementaci metod tříd v jiném jazyce, většinou v jazyce C.
[12] `http://www.eclipse.org`
[13] `http://gcc.gnu.org/java/`
[14] `http://www.gnu.org/software/classpath/`
[15] `http://www.e-leos.net`

Ačkoliv by se podle této kapitoly mohlo zdát, že má Java velké problémy s efektivitou, opak je pravdou. Cílem této kapitoly bylo pouze ukázat, kde bývá zakopán pes a jak se ho zbavit.

Pro provoz klientských aplikací v Javě používám notebook s procesorem PII 400 Mhz a 128 MB paměti, Tomcat mi běží na serveru Pentium 200 Mhz s 64 MB paměti. Naprosto bez problému.

5 Nástroje

Jednou z výhod jazyka Java je existence standardů a specifikací, které se snaží všichni programátoři i dodavatelé nástrojů dodržovat[16]. Díky tomu je ve většině případů snadné nevyhovující nástroj či knihovnu vyměnit bez nutnosti velkých zásahů do zdrojového kódu.

Na internetu lze nalézt množství různých knihoven a doplňků pro Javu, tato kapitola se zaměřuje pouze na několik nejzajímavějších.

5.1 Java 2 SDK/JRE

Pro vývoj či překlad aplikací v Javě potřebujeme především Java 2 SDK, pro jejich provoz pak ve většině případů postačí JRE. Nejjednodušší je použít implementaci přímo od firmy Sun Microsystems, Inc., kterou lze nalézt na adrese `http://java.sun.com`.

Existují další volně dostupná SDK a JRE, např. od firmy IBM.

5.2 JavaDoc

Systém JavaDoc slouží ke generování programové dokumentace přímo ze zdrojových kódů. Využívá speciálních komentářů ve specifickém formátu, které popisují jednotlivé prvky zdrojového kódu a jejich vlastnosti. Na základě nich pak může vytvářet dokumentaci v různých formátech. To zajišťují tzv. *doclety*.

Generátor JavaDoc je standardní součástí Java2 SDK a v základní výbavě disponuje docletem pro generování dokumentace ve formátu HTML. Nicméně existují další doclety pro jiné formáty od různých dodavatelů[17].

5.3 Apache Ant

Ant[18] je nástroj pro řízení překladu programů v Javě. Je to vlastně náhrada tradičního unixového příkazu `make`. Má několik užitečných vlastností, nicméně vám samozřejmě nic nebrání zůstat u svého oblíbeného `make` nebo `gmake`.

Překlad je řízen pomocí souboru ve formátu XML, který má většinou název `build.xml`.

[16] V Javě existují i poměrně striktní konvence pro tvorbu identifikátorů či formátování zdrojového textu a jsou většinou programátorů dodržovány. To mimo jiné velmi usnadňuje orientaci v cizím kódu.

[17] Existuje i doclet názvem XDoclet, který ze zdrojového kódu generuje tzv. deployment deskriptory, používané v J2EE k popisu servletů a EJB.

[18] `http://jakarta.apache.org/ant/`

5.4 JUnit

JUnit[19] je framework pro testování aplikací v jazyce Java. Vychází z metodiky vývoje aplikací, která se nazývá *extrémní programování* [6], ale může být samozřejmě použita i samostatně.

Je přímo podporován i programem Ant, takže může být snadno integrován do skriptů pro řízení překladu, aby se příslušné testy spouštěly vždy při změně souvisejících tříd.

5.5 Tomcat

Tomcat je open source implementace servlet containeru, plně kompatibilní se specifikacemi J2EE. Sun Microsystems, Inc. jej dokonce označuje jako *referenční implementaci* a přibaluje ho jako součást svého J2EE SDK.

Tomcat je podobně jako Ant součást projektu Jakarta, který se zabývá vývojem několika zajímavých nástrojů a knihoven pro platformu Java.

5.6 Jetty

Jetty[20] je další implementací servlet containeru. Není tolik rozšířený jako Tomcat, ale je masivně podporován autory projektu JBoss, kteří jej preferují před Tomcatem.

5.7 Jboss

Jboss[21] je open source implementací J2EE, šířenou pod LGPL licencí. Autoři se snaží vytvořit co nejkvalitnější implementaci, která bude plně odpovídat nejnovějším specifikacím J2EE.

Autoři zvolili zajímavý obchodní model. Projekt je distribuován pouze s jednoduchou základní dokumentací a autoři nabízejí kromě placené podpory i podrobnou a kvalitní dokumentaci za poplatek, který není příliš vysoký. Nicméně pro základní nakonfigurování a spuštění aplikačního serveru postačí dokumentace základní a pro vývoj aplikací lze použít standardní dokumentaci k J2EE.

5.8 Xerces a Xalan

Xerces[22] je mocná knihovna pro parsování XML dokumentů. Je založena na kódu původně vyvinutým firmou IBM, která jej později věnovala projektu Apache XML. Na základě požadovaných parametrů vytvoří parser s příslušnými vlastnostmi.

Xalan je knihovna XSLT procesoru, který plně implementuje doporučení W3C pro XSLT a XPath. Je možné jej použít pouze pro vyhodnocování XPath výrazů.

[19] `http://www.junit.org`
[20] `http://www.jetty.org`
[21] `http://www.jboss.org`
[22] `http://xml.apache.org`

5.9 JWSDP

Java Web Services Development Pack je balík určený pro vývoj webových služeb, dodávaný firmou Sun Microsystems, Inc. Ve skutečnosti jde o Tomcat, Xerces, Xalan a několik dalších volně dostupných nástrojů přebalených do jednoho balíčku.

6 Jak začít

Učebnic programování v jazyce Java je na našem trhu poměrně dost a v podstatě nezáleží na tom, kterou z nich si vyberete. Velké množství tutoriálů, návodů a dokumentace informací je k dispozici na adrese `http://java.sun.com`. Pokud se chcete věnovat programování v Javě profesionálně, rozhodně by ve vaší knihovničce neměla chybět [7].

Co se týče ostatních pokročilejších témat, nejlepším zdrojem informací je internet. Doporučuji `http://www.google.com`, `http://www.freshmeat.net` a moji stránku `http://www.herkules.cz/java/`.

7 Závěr

Co říci závěrem? Java na Linuxu funguje naprosto bez problému a je možné vybírat z mnoha nástrojů od různých dodavatelů. Potěšitelná je zejména existence open source projektů, které zajišťují dostupnost většiny potřebných nástrojů. Nezanedbatelnou výhodou je existence standardních specifikací, zejména pak J2EE.

Kombinace operačního systému Linux a technologií platformy Java se jeví jako ideální základ pro budování informačních systémů a podnikových aplikací, neboť přináší výhody v podobě nezávislosti na dodavateli řešení, nízkých vstupních nákladů, robusní infrastruktury a uznávaných standardů.

Reference

1. Gosling, J., Joy, B., Steele, G., Bracha, G.: The Java Language Specification, Second Edition. Addison-Wesley, 2000.
2. Lindolm, T., Yellin, F.: The Java Virtual Machine Specification, Second Edition. Addison-Wesley, 1999.
3. Oaks, S.: Java Security. O'Reilly, 1998.
4. Knudsen, J.: Java Cryptography. O'Reilly, 1998.
5. Monson-Haefel, R.: Enterprise JavaBeans. O'Reilly, 2000.
6. Beck, K.: Extrémní programování. Grada, 2002.
7. Bloch, J.: Java efektivně – 57 zásad softwarového experta. Grada, 2002.

BSD Sockets pro IPv6

Ladislav Lhotka

CESNET, z.s.p.o.
Email: Lhotka@cesnet.cz

Abstrakt: *BSD Sockets* je nejpoužívanější aplikační programové rozhraní (API) síťových aplikací založených na IP. Nový protokol IPv6 si vyžádal určité změny tohoto API, vyvolané jak prodloužením adres, tak i dalšími novými vlastnostmi IPv6 (značka toku, rozsah působnosti adres). Příspěvek popisuje tyto změny v porovnání s původní verzí pro IPv4 a přináší též příklady použití v jazycích Python a C.

Klíčová slova: BSD Sockets, IPv6. Python, Internet BSD Unix

1 Úvod

BSD Sockets je aplikační programové rozhraní (API), s jehož pomocí lze programovat komunikační aplikace všeho druhu. Jak název napovídá, bylo původně vytvořeno na univerzitě v Berkeley pro BSD Unix, ale postupně se rozšířilo nejen do všech ostatních odnoží Unixu, ale také do operačních systémů MacOS, MS Windows a dokonce i PalmOS. Použití *BSD Sockets* se neomezuje jen na síťový protokol IP, nicméně v oblasti internetových aplikací má zdaleka největší význam. Bez nadsázky lze tvrdit, že současný Internet je na tomto programovém rozhraní postaven.

Jak známo, (téměř) všechno v Unixu je soubor. *BSD Sockets* používají toto unixové paradigma pro komunikaci mezi programy. Využívá se k tomu abstraktní programový objekt – *soket* – reprezentující komunikační kanál, jímž si mohou programy navzájem vyměňovat data prostřednictvím standardních souborových deskriptorů *(file descriptors)*.

API *BSD Sockets* je velmi obecné a je možno je použít pro celou řadu rodin komunikačních protokolů *(protocol families)*. Linux 2.4 jich v hlavičkovém souboru `/usr/include/bits/socket.h` definuje celkem 32. Nás zde budou zajímat pouze dvě z nich, používané pro datovou komunikaci nad IP: `PF_INET` pro IPv4 a `PF_INET6` pro IPv6.

Datové struktury původních *BSD Sockets* byly sice navrženy s jistou rezervou, ta však nestačí pojmout čtyřnásobné prodloužení adres a další informace, které sokety IPv6 potřebují. API bylo proto nutno doplnit a upravit. Příslušná pracovní skupina IETF *(Internet Engineering Task Force)* této příležitosti využila také k jisté konsolidaci a zefektivnění API *BSD Sockets*. Výsledkem je RFC 2553 [5], které de facto umožňuje programovat internetové aplikace jednotně pro obě verze protokolu IP.

Jan Kasprzak, Petr Sojka (editoři): SLT 2002 – sborník semináře o Linuxu a TEXu, str. 169–180, 2002.

Datové struktury a funkce *BSD Sockets* budeme popisovat v té podobě[1], jak jsou definovány v hlavičkových souborech Linuxu, tedy prostředky programovacího jazyka C. Ekvivalentní API je ovšem k dispozici i pro většinu běžných programovacích jazyků. Pro ukázky programů na konci tohoto příspěvku použijeme jazyk Python, který jako jeden z prvních (od verze 2.2.1) dává k dispozici toto nové API. Ukázky tak budou daleko přehlednější, neboť budeme ušetřeni některých technických komplikací při práci s proměnnými a argumenty funkcí.

Tento příspěvek podrobně diskutuje pouze změny, které přináší nové API *BSD Sockets* a předpokládá tudíž aspoň rámcovou znalost technologie IP, jazyka C a původního API. Pro získání či doplnění vědomostí o obou tématech je k dispozici jak řada zdrojů na webu, tak i několik monografií (též v češtině). Za všechny jmenujme aspoň klasickou knihu [7]. Velmi dobrým zdrojem informací jsou také unixové on-line manuály.

Dovolím si ještě malou jazykovou poznámku: Anglický termín *BSD Sockets* jako obvykle staví autora odborného textu v češtině před obtížné dilema: ponechat anebo přeložit? Předchozí pokusy o překlad se mi po pravdě řečeno nelíbily. Kupříkladu překladatel výše uvedené knihy [7] vynalezl celkem kuriózní termín „schránky Berkeley“. Snad mě proto nebudou lingvisté tepat, pokud používám nepřeložený termín *BSD Sockets* pro API jako takové a mírně počeštěný název *soket* (podle vzoru kriket) pro objekt, s nímž se pracuje v programech.

2 Principy nového návrhu API

Jak jistě všichni pozorujeme, oproti původním představám se IPv6 prosazuje proti verzi 4 dosti pomalu a obtížně. Na většinu uživatelů současného Internetu totiž adresová nouze zatím nedoléhá a ostatní pozitiva IPv6 zřejmě nevyváží utrpení spojené s přechodem na nový protokol. V této situaci je proto nezbytné, aby se nové API pro IPv6 chovalo velmi nenápadně a nenarušilo hladký chod stávajících síťových služeb a aplikací. Rovněž je potřeba, aby přidání podpory IPv6 do stávajících programů bylo pokud možno snadné.

Návrh nového API se proto řídil těmito kritérii:

- Veškeré změny musí zachovat kompatibilitu se stávajícími programy na úrovni *binární i zdrojové*. Jinými slovy, binární programy musí nadále fungovat i na systémech s knihovnami podporujícími nové API a programy napsané pro staré API musí jít beze změn přeložit a fungovat i na systémech s novým API.
- Změny v API musí být co nejmenší a v souladu s tradičními postupy programování pomocí *BSD Sockets*.
- Je-li to vhodné a možné, programy napsané pro nové API by měly být schopny komunikovat s jinými počítači jak pomocí IPv6, tak i IPv4. Nové API také musí fungovat na strojích podporujících zároveň obě verze IP (tzv. *duální sloupec* protokolů).

[1] někdy ovšem kvůli přehlednosti s mírnými zjednodušeními

Znamená to tedy, že díky uvedeným zásadám bude migrace aplikací na IPv6 triviální a programátorův život radostný? Obecně bohužel nikoli. Existuje totiž pěkná řádka programů a protokolů, které používají některý z těchto postupů či předpokladů:

- Pracují s adresou IPv4 uvnitř aplikace (identifikují uzly adresami, kešují adresy získané z DNS apod.)
- Předpokládají, že síťové rozhraní má jen jednu IP adresu.
- Předpokládají, že IP adresa má 32 bitů.

Programy, které těmito neduhy netrpí, je ale obvykle možno upravit pro nové API. Existují dokonce i automatické konvertory, např. *Verto IPv6* od firmy *i2soft*.

3 Datové struktury

Zcela nevyhnutelné byly změny adresových struktur definovaných v hlavičkovém souboru `<netinet/in.h>`. Pro IPv6 se zavádí nová adresová rodina *(address family)* `AF_INET6`. Tato konstanta má z praktických důvodů přiřazenu stejnou číselnou hodnotu jako `PF_INET6` (viz `<bits/socket.h>`:

```
#define     AF_INET6     PF_INET6
```

Dříve se totiž obvykle kategorie konstant `AF_` a `PF_` vcelku libovolně zaměňovaly, RFC 2553 ale již důsledně respektuje jejich sémantickou odlišnost.

Původní API používá pro zápis adres IPv4 strukturu `in_addr` definovanou v Linuxu takto:

```
typedef uint32_t in_addr_t;
struct in_addr
  {
    in_addr_t s_addr;
  };
```

Jak je vidět, ve své podstatě jde pouze o obal pro číslo typu `unsigned long`. Obdobná struktura pro adresy IPv6 je už složitější, v prvním přiblížení si ji můžeme představit takto:

```
struct in6_addr
  {
    uint8_t s6_addr[16];
  };
```

Ve skutečnosti, podíváte-li se do hlavičkového souboru `<netinet/in.h>`, zjistíte, že deklarace je ještě složitější z důvodu efektivního zarovnání struktury na hranici slova pro všechny typy mikroprocesorových architektur. Těmito technickými detaily se však nebudeme zabývat.

Zajímavější je situace u adres soketů. Hlavní funkce *BSD Sockets* se používají univerzálně pro všechny rodiny protokolů. Adresy soketů se jim proto musí

předávat v jednotné formě, tedy prostřednictvím „neprůhledných“ *(opaque)* ukazatelů na abstraktní strukturu `sockaddr`. Ukazatele na struktury soketových adres konkrétních rodin protokolů je tudíž třeba pro tento účel přetypovat.

Adresy klasických internetové soketů se skládají, jak víme, z dvojice (IP adresa, TCP/UDP port). Příslušná struktura `sockaddr_in` je definována takto:

```
struct sockaddr_in
  {
    unsigned short int sin_family;    /* Address family */
    uint16_t sin_port;                /* Port number */
    struct in_addr sin_addr;          /* Internet address */
    unsigned char sin_zero[8];        /* Padding */
  };
```

Její délka je pomocí 8 bajtů „vaty“ *(padding)* dorovnána na velikost abstraktní struktury `sockaddr`. Tuto rezervu si zřejmě designéři *BSD Sockets* ponechávali pro budoucí verze protokolu IP. Jak se zdá, ani ve snu je nenapadlo, že by se měla délka IP adresy dostat až na 128 bitů. Nicméně stalo se a do `sockaddr_in` se už nevejde ani samotná adresa IPv6. Proto se rovněž musela definovat nová struktura pro adresu soketu IPv6:

```
struct sockaddr_in6
  {
    unsigned short int sin6_family;  /* Address family */
    uint16_t sin6_port;              /* Port number */
    uint32_t sin6_flowinfo;          /* Traffic class&flow info */
    struct in6_addr sin6_addr;       /* IPv6 address */
    uint32_t sin6_scope_id;          /* IPv6 scope-id */
  };
```

Vidíme zde i další nové položky:

`sin6_flowinfo` obsahuje údaje ze dvou polí hlavičky IPv6 [1]: třída provozu *(traffic class)* a značka toku *(flow label)*.

`sin6_scope_id` určuje síťová rozhraní patřící do stejné adresové sféry *(address scope)* jako použitá adresa (viz [4]).

Pro pořádek ještě dodejme, že všechny vícebajtové položky struktur (včetně šestnáctibajtové adresy IPv6) je nutno ukládat v tzv. *síťovém pořadí bajtů (NBO – Network Byte Order)*, který se v kontextu mikroprocesorových architektur nazývá *big endian*. Na strojích s procesory Intel (i jiných) proto musíme vždy použít konverzní funkce **htonl()** nebo **htons()**.

4 Funkce

Aplikační programové rozhraní *BSD Sockets* nabízí pro práci s internetovými sokety několik druhů funkcí:

1. Základní funkce pro obsluhu soketu
2. Funkce pro převod jména na IP adresu a obráceně
3. Transformace mezi různými adresními formáty

4.1 Základní funkce

Jak jsme již uvedli, základní funkce pro vytvoření soketu a práci s ním jsou stejné pro všechny podporované rodiny protokolů. Soket IPv6/TCP se vytvoří v céčkovém programu obvyklým způsobem:

```
s = socket(PF_INET6, SOCK_STREAM);
```

Obdobně soket IPv6/UDP:

```
s = socket(PF_INET6, SOCK_DGRAM);
```

Také ostatní základní funkce pro práci se sokety zůstávají beze změny. Zde je jejich stručný přehled:

bind() – Připojení soketu k lokální adrese.
close() – Uzavření soketu.
connect() – Připojení ke vzdálenému uzlu.
getpeername() – Získání adresy vzdáleného uzlu, který je k soketu připojen.
getsockopt() – Zjištění parametrů soketu.
listen() – Naslouchání na soketu (čekání na žádost o spojení).
recv() – Čtení dat ze soketu.
recvfrom() – Čtení dat ze soketu. Spolu s daty se získá i adresa vzdáleného uzlu.
send() – Posílání dat do soketu. Soket již musí být připojen na vzdálený uzel pomocí **connect()**.
sendto() – Posílání dat do soketu zároveň s udáním cílové adresy.
setsockopt() – Nastavení volitelných parametrů soketu.
shutdown() – Ukončení komunikace v jednom nebo obou směrech podle hodnoty argumentu této funkce.

Použití některých z těchto funkcí budeme prakticky demonstrovat na ukázce v závěru příspěvku.

Funkce **bind()** umožňuje prostřednictvím svého argumentu specifikovat zdrojovou adresu datagramů odesílaných soketem. V řadě případů nám na tom nezáleží a můžeme nechat volbu zdrojové adresy na operačním systému. V případě IPv4 bylo běžnou praxí používat funkci **bind()** například takto:

```
int s;
struct sockaddr_in zde;
zde.sin_family = AF_INET;
zde.sin_port = 0;
zde.sin_addr.s_addr = INADDR_ANY;
bzero(&(zde.sin_zero), 8);
s = socket(PF_INET, SOCK_STREAM, 0);
bind(s, (struct sockaddr *)&zde, sizeof(struct sockaddr));
```

Symbolická konstanta `INADDR_ANY` na místě adresové struktury `in_addr` vyjadřuje právě náš záměr ponechat volbu zdrojové IP na systému, (podobně jako nulové číslo portu o řádek výše). Vzhledem k tomu, že `in_addr` je de facto komplikovaně zapsaná skalární hodnota, můžeme `INADDR_ANY` používat i v přiřazovacích příkazech, jak jsme viděli ve výše uvedeném příkladu.

U adres IPv6 už to takto jednoduše nejde, protože typ `in6_addr` obsahuje šestnáctiprvkové pole. Proto je (rovněž v `<netinet/in.h>`) definována globální proměnná `in6addr_any` výše uvedeného typu, již lze použít stejně jako symbolickou konstantu v případě IPv4:

```
zde.sin6_addr.s6_addr = in6addr_any;
```

Druhou možností pak je použití symbolické konstanty `IN6ADDR_ANY_INIT`, kterou však nelze použít v přiřazovacím příkazu, nýbrž pouze při inicializaci hodnoty proměnné, například

```
struct in6_addr cokoli = IN6ADDR_ANY_INIT;
```

Zcela obdobná situace nastává v případě adresy rozhraní zpětné smyčky *(loopback)*, kterou však můžeme použít jak pro zdrojovou, tak i cílovou adresu. Pro IPv4 je k dispozici symbolická konstanta `INADDR_LOOPBACK` odpovídající adrese 127.0.0.1, zatímco pro IPv6 si opět můžeme vybrat jedno ze dvou řešení:

- globální proměnná `in6addr_loopback`,
- symbolická konstanta `IN6ADDR_LOOPBACK_INIT` použitelná pouze pro inicializaci hodnoty proměnné.

4.2 Převody mezi jmény na adresami

Málokterá síťová aplikace pro IPv4 se asi obejde bez použití systému DNS a tím i funkce **gethostbyname()**. V kontextu IPv6 je ale tato funkce nepoužitelná. Každý uzel Internetu podporující IPv6 totiž bude mít kromě jedné nebo více adres IPv6 obvykle také adresu IPv4. Obě verze adres mohou být v DNS přiřazeny stejnému doménovému jménu – adresu IPv4 najdeme v záznamu typu A a adresu IPv6 v záznamu typu AAAA[2]. Funkce **gethostbyname()** neumožňuje říci, který z obou typů záznamu chceme, popřípadě zda chceme oba. RFC 2553 proto zavedlo nové funkce **getipnodebyname()** a **getipnodebyaddr()** pro převod jména na adresu a obráceně. Bohužel i tyto funkce se ukázaly jako nedostatečné, neboť nepodporují adresové sféry *(address scopes)*. V nejnovějších draftech IETF [2] byly proto i tyto funkce zavrženy. Pro převod jména na adresu tak zbyla jediná, protokolově nezávislá funkce – **getaddrinfo()**, jež má svůj původ ve standardu IEEE POSIX 1003.1g. Kromě **gethostbyname()** plně nahrazuje i funkci **getservbyname()**. Prototyp vypadá takto:

```
int *getaddrinfo(const char *nodename, const char *servname,
       const struct addrinfo *hints, struct addrinfo **res);
```

[2] nebo A6, který je ale v současné době v nemilosti

Funkce vrací buď nulu v případě úspěchu anebo různé chybové kódy. Skutečný výsledek je předán prostřednictvím argumentu `res`, který je ukazatelem na jednosměrný seznam struktur typu `addrinfo` definovaného v `<netdb.h>`:

```
struct addrinfo
{
  int ai_flags;              /* Input flags.  */
  int ai_family;             /* Protocol family for socket.  */
  int ai_socktype;           /* Socket type.  */
  int ai_protocol;           /* Protocol for socket.  */
  socklen_t ai_addrlen;      /* Length of socket address.  */
  struct sockaddr *ai_addr;  /* Socket address for socket.  */
  char *ai_canonname;        /* Canonical name.  */
  struct addrinfo *ai_next;  /* Pointer to next in list.  */
};
```

Položka `ai_next` ukazuje na následující prvek seznamu, poslední prvek seznamu pak zde má NULL. Paměť pro seznam musí být alokována dynamicky, a proto musíme (alespoň v Céčku) pamatovat na řádné uvolnění této paměti po použití. K tomu slouží funkce **freeaddrinfo()**.

Každý z prvků seznamu obsahuje položky, které můžeme přímo použít jako argumenty pro funkci **socket()**: `ai_family` (v našem případě `PF_INET` či `PF_INET6`, `ai_socktype` (obvykle `SOCK_STREAM` nebo `SOCK_DGRAM`) a `ai_protocol` (obvykle `IPPROTO_TCP` nebo `SOCK_UDP`). Položka `ai_addr` pak ukazuje na adresu soketu, jejíž délku specifikuje položka `ai_addrlen`.

Argument `nodename` by měl obsahovat doménové jméno, o které nám jde, popřípadě obvyklý textový zápis adresy IPv4 nebo IPv6. Pokud chceme, můžeme vyplnit i argument `servname` a to buď jménem služby podle `/etc/services` anebo dekadickým číslem portu ve formě řetězce.

Argument `hints` umožňuje stanovit, o jaký typ dat máme zájem. Jeho typem je opět `struct addrinfo`, v níž ovšem smějí být vyplněny jen tyto položky: `ai_flags`, `ai_family`, `ai_socktype` a `ai_protocol`. Ostatní položky musí být vynulovány. V případě, že v některé kategorii jsme připraveni přijmout cokoli, nastavíme hodnoty podle následující tabulky. Nemáme-li preference v žádné kategorii, stačí předat NULL na místě argumentu `hints`.

Kritérium	**Položka**	**Nastavení**
Libovolná adresová rodina	`ai_family`	`AF_UNSPEC`
Libovolný typ soketu	`ai_socktype`	0
Libovolný protokol	`ai_protocol`	0

Položka `ai_flags` v argumentu `hints` hints obsahuje příznaky *(flags)*, kterými můžeme své požadavky specifikovat ještě detailněji. Jak je obvyklé, jednotlivé příznaky můžeme kombinovat pomocí operace bitového OR (`|`). K dispozici jsou tyto příznaky:

`AI_PASSIVE` Tento příznak hraje roli jen v případě, že je na místě argumentu `nodename` předána hodnota NULL. Příznak nastavíme, pokud chceme výstup funkce **getaddrinfo()** použít pro funkci **bind()**, tj. na lokální straně soketu. Adresa soketu se v tom případě vyplní s `INADDR_ANY` pro IPv4 nebo `IN6ADDR_ANY_INIT` pro IPv6 (viz oddíl 4.1). Pokud chceme výstup funkce **getaddrinfo()** použít pro vzdálenou stranu soketu, tedy např. ve funkcích **connect()** či **sendto()**, pak tento příznak nenastavíme. Dostaneme pak adresu soketu s `INADDR_LOOPBACK` pro IPv4 nebo `IN6ADDR_LOOPBACK_INIT` pro IPv6.

`AI_CANONNAME` Nastavíme-li tento příznak a argument `nodename` není NULL, funkce **getaddrinfo()** se pokusí též vyhledat *kanonické doménové jméno* uzlu. To se hodí třeba tehdy, když známe DNS alias a chceme zjistit kanonické jméno.

`AI_NUMERICHOST` Pokud je tento příznak nastaven, musí být argument `nodename` řetězcem reprezentujícím adresu IPv4 nebo IPv6. Funkce **getaddrinfo()** pak vůbec nepoužívá DNS.

`AI_NUMERICSERV` Tento příznak nastavíme, pokud chceme jako argument `servname` použít řetězec s dekadickým číslem portu.

`AI_V4MAPPED` Tento příznak má smysl jen pro `AF_INET6` v položce `ai_family` argumentu `hints`. Pokud je nastaven a funkce **getaddrinfo()** nenalezne pro zadané `nodename` v DNS žádné záznamy typu AAAA, poskytne jako výsledek obsah záznamů typu A (pokud takové existují) ve formě IPv4-mapovaných adres IPv6 *(IPv4-mapped IPv6 addresses)* [4].

`AI_ALL` Je-li tento příznak nastaven zároveň s `AI_V4MAPPED` a `hints.ai_family` je `AF_INET6`, pak funkce **getaddrinfo()** vrátí adresové údaje ze *všech* záznamů DNS typu AAAA i A, přičemž posledně jmenované budou opět ve formě IPv4-mapovaných adres IPv6. V případě, že adresová rodina není specifikována, tj. `hints.ai_family==AF_UNSPEC` anebo `hints==NULL`, bere se nastavení příznaků `AI_V4MAPPED` a `AI_ALL` v úvahu jen tehdy, je-li na hostitelském počítači podporováno IPv6.

`AI_ADDRCONFIG` Nastavení tohoto příznaku vede k tomu, že funkce **getaddrinfo()** vrátí adresy IPv4 jen tehdy, je-li na hostitelském počítači zkonfigurována nějaká lokální adresa IPv4 a podobně vrátí adresy IPv6 jen tehdy, je-li zkonfigurována aspoň jedna lokální adresa IPv6 (v obou případech se nepočítá adresa zpětné smyčky).

V jistém smyslu inverzní funkcí ke **getaddrinfo()** je tato funkce:

```
int getnameinfo(const struct sockaddr *sa, socklen_t salen,
                char *node, socklen_t nodelen, char *service,
                socklen_t servicelen, int flags);
```

Jejím úkolem je převod obsahu adresy soketu na doménové jméno uzlu a/nebo jméno služby. Návratová hodnota je opět buď nula v případě úspěchu nebo příslušný chybový kód.

Struktura typu `sockaddr`, na niž ukazuje argument `sa`, může obsahovat adresu IPv6 s vnořenou adresou IPv4, tedy buď IPv4-mapovanou nebo IPv4-kompatibilní adresu [4]. V takovém pádu se vnořená adresa IPv4 extrahuje a nadále se pokračuje tak, jako by byla předložena struktura s adresou IPv4.

Výsledky převodu funkce **getnameinfo** zapisuje do bufferů, na které ukazují argumenty `node` a `service`. Tyto buffery musí mít dopředu přidělenu paměť a argumenty `nodelen` a `servicelen` udávají jejich délku. Je-li některý z argumentů `node` a `service` předán jako nulový ukazatel, příslušná hodnota se nevrací.

Poslední argument `flags` specifikuje následující příznaky, které mohou být opět kombinovány pomocí bitového OR:

`NI_NOFQDN` Při nastavení tohoto příznaku se pro lokální uzel vrátí jen jeho samotné jméno, tedy bez domény.

`NI_NUMERICHOST` Při jeho nastavení se do `*node` zapíše řetězec reprezentující numerickou adresu. Pokud je to adresa IPv6, může v ní být vyznačena i adresová sféra *(address scope)* způsobem popsaným v draftu [3], například `"fe80::1%eth0"`.

`NI_NUMERICSCOPE` Je-li nastaven tento příznak společně s předchozím, pak se adresová sféra v textové reprezentaci adresy IPv6 zapíše jako číslo (např. index rozhraní) a nikoli symbolické jméno (např. `eth0`).

`NI_NUMERICSERV` Obdobně při nastavení tohoto příznaku se do `*service` zapíše řetězec s dekadickým číslem portu.

`NI_NAMEREQD` Pokud je příznak nastaven a doménové jméno uzlu nemůže být nalezeno, vrátí funkce **getnameinfo** nenulový chybový kód.

`NI_DGRAM` Nastavení tohoto příznaku naznačuje, že se jedná o nespojovanou službu (`SOCK_DGRAM`). Implicitně se předpokládá spojovaná služba (`SOCK_STREAM`).

4.3 Konverze adresových formátů

Při práci s adresami IPv4 i IPv6 velmi často potřebujeme převést jejich čitelnou textovou reprezentaci (např. 1.2.3.4 nebo fe80::1234) do číselné binární formy anebo naopak. Tomuto účelu slouží následující dvě funkce:

```
int inet_pton(int af, const char *src, void *dst);

const char *inet_ntop(int af, const void *src,
                      char *dst, socklen_t size);
```

Funkce **inet_pton** převádí textovou reprezentaci adresy na binární a **inet_ntop** binární na textovou. Binární reprezentace je vždy v síťovém pořadí bajtů (NBO).

Argument `af` specifikuje adresovou rodinu (`AF_INET` nebo `AF_INET6`) a argumenty `src` a `dst` oběma funkcím předávají ukazatele na buffery, v nichž je uložena vstupní hodnota (`src`), resp. do nichž se má zapsat zkonvertovaný výsledek (`dst`). Konečně argument `size` u druhé funkce udává ještě délku

bufferu pro výsledek – ta by měla být dostatečná pro to, aby do ní bylo možno zapsat úplnou textovou reprezentaci příslušné adresy IPv4 nebo IPv6.

Funkce **inet_pton** vrací 1 v případě úspěchu konverze, 0 v případě špatného vstupu a −1, pokud udaná adresová rodina není známa. Funkce **inet_ntop** naproti tomu v případě úspěchu vrací ukazatel na buffer, v němž je uložen výsledek a nulový ukazatel v případě neúspěchu.

5 Kompatibilita s uzly IPv4

Chrabří uživatelé síťových aplikací IPv6 jistě nestojí o to, aby se tím znemožnila nebo i jen zhoršila možnost komunikace se stávajícím Internetem. V oddílu 4.2 jsme si ukázali, jak je možné pomocí funkce **getaddrinfo()** a příznaku `AI_V4MAPPED` zajistit, aby se jméno DNS resolvovalo alespoň pomocí záznamu typu A, pokud nejsou k dispozici žádně záznamy typu AAAA. Síťová aplikace IPv6 ale může navázat komunikaci s uzlem IPv4 sama tím, že v adresové struktuře `sockaddr_in6` použije IPv4-mapovanou adresu IPv6 [4]. Příslušná knihovna implementující API *BSD Sockets* pak musí za běhu překládat mezi protokoly IPv4 a IPv6 tak, aby uzel IPv4 dostával jen datagramy IPv4 a naopak ty, které od něj přijdou musí být zase převedeny na IPv6 s IPv4-mapovanými adresami.

Aplikace používající sokety IPv6 proto obvykle vůbec nepotřebují vědět, jestli komunikují pomocí nativních adres IPv6 anebo IPv4-mapovaných adres. Ty aplikace, které to z nějakého důvodu chtějí vědět, mohou použít makro `IN6_IS_ADDR_V4MAPPED` definované v hlavičkovém souboru `<netinet/in.h>`.

6 Kompletní ukázka

Tento oddíl demonstruje použití API *BSD Sockets* na dvou velmi jednoduchých programech v jazyku Python verze 2.2.1 nebo vyšší. Programy realizují nespojovanou komunikaci pomocí IPv6/UDP: Po inicializaci soketu běží oba v nekonečné smyčce. Vysílač vysílá každou vteřinu pozdravný řetězec a přijímač jej po obdržení vypíše na terminál spolu s adresou uzlu, který jej poslal. Oba programy je nutno ukončit tvrdě pomocí `Ctrl-C`.

Povšimněme si, jak jazyk Python programátora osvobozuje od řady technických detailů (typy proměnných, dosazování implicitních hodnot funkčních argumentů atd.) a také jak celkem přirozeně využívá své objektové konstrukce. Na druhou stranu, v ukázce pro přehlednost zcela pomíjíme ošetřování chyb, k nimž může za běhu dojít. To lze však v Pythonu také elegantně realizovat pomocí programových výjimek *(exceptions)*.

Další příklady použití *BSD Sockets* v Pythonu jsou uvedeny v referenčním manuálu [6].

6.1 Vysílač

```
from socket import *
import time

peer = "www.cesnet.cz"
port = 54321
af,styp,proto,cn,sa = getaddrinfo(peer, port,
                                  AF_INET6, SOCK_DGRAM)[0]
s = socket(af,styp,proto)
while 1:
    s.sendto("Ahoj!", sa)
    time.sleep(1)
```

6.2 Vysílač

```
from socket import *
import time

maxlen = 512
port = 54321
af,styp,proto,cn,sa = getaddrinfo(None, port,
                                  AF_INET6, SOCK_DGRAM,
                                  0, AI_PASSIVE)[0]
s = socket(af,styp,proto)
s.bind(sa)
while 1:
    recv = s.recvfrom(maxlen)
    print "Node", recv[1][0], "said:", recv[0]
```

7 Závěr

Aplikační programové rozhraní *BSD Sockets* sockets muselo být poněkud upraveno pro použití s novým protokolem IP verze 6. Nové API je definováno v RFC 2553 [5] a následujících draftech IETF. Všechny úpravy jsou vedeny požadavkem minimality změn a také téměř absolutní kompatibility s IPv4.

Upravené API de facto sjednocuje použití *BSD Sockets* pro obě verze protokolu IP. Programátor aplikací IPv6 tak má minimálně dvě možnosti, jak zajistit, aby jeho programy fungovaly i pro komunikaci s uzly, které podporují pouze IPv4:

1. Pro spojení pomocí IPv4 mohou nadále používat tradiční API a pouze pro spojení pomocí IPv6 využít nové struktury a funkce.
2. Mohou ale též aplikaci naprogramovat jednotně pomocí nového API a prakticky se nestarat o to, jestli je komunikační partner uzlem „IPv4-only“ anebo podporuje i IPv6.

Důraz na kompatibilitu také znamená, že migrace většiny aplikací z IPv4 na IPv6 nemusí být příliš složitá. Výjimkou jsou jenom programy, které z nějakého důvodu používají adresy IPv4 interně.

Poněkud nepříjemný je fakt, že změny v API *BSD Sockets* ještě nejsou plně ustálené a například zmíněné RFC 2553 už dnes zčásti neplatí. To je důsledkem vývoje protokolu IPv6 jako takového a budeme s tím muset ještě nějakou dobu počítat.

Reference

1. S. Deering a R. Hinden. RFC 2460: Internet Protocol, Version 6 (IPv6) Specification, December 1998.
2. R. Gilligan et al. Basic Socket Interface Extensions for IPv6 (`draft-ietf-ipngwg-rfc2553bis-07.txt`), September 2002.
3. S. Deering et al. IPv6 Scoped Address Architecture (`draft-ietf-ipngwg-scoping-arch-04.txt`), June 2002.
4. R. Hinden a S. Deering. RFC 2373: IP Version 6 Addressing Architecture, July 1998.
5. J. Bound R. Gilligan, S. Thomson a W. Stevens. RFC 2553: Basic Socket Interface Extensions for IPv6, April 1999.
6. *Python Library Reference*, `http://www.python.org/doc/current/lib/lib.html`.
7. Richard W. Stevens. *UNIX Network Programming*. Prentice Hall, 1990.

Softwarový RAID pod Linuxem

David Häring

Internet Billboard s. r. o., Novoveská 95, Ostrava
Email: david.haring@billboard.cz

Abstrakt: Disková pole (RAID) představují možnost, jak zvýšit výkon či kapacitu diskového subsystémů anebo posílit jeho odolnost vůči výpadku hardware. Disková pole lze implementovat čistě na úrovni hardware, ale také softwarové na úrovni OS. Řada dnešních OS obsahuje implementace softwarových variant RAIDu a ani Linux není výjimkou. Cílem je poskytnout přehled o možnostech, stabilitě a výkonu softwarového RAIDu pod Linuxem. V rámci přednášky bude postupně probrány obecně principy fungování RAIDu, výhody/nevýhody softwarového RAIDu ve srovnání s hardwarovými řešeními, implementace RAIDu pod Linuxem, obslužný software (starší raidtools, novější mdadm), autodetekce a automatické sestavení poli jádrem, možnosti rekonfigurace polí (raidreconf), optimalizace polí.

Klíčová slova: RAID, striping, mirroring, redundance, Linux

1 Hardwarové a softwarové implementace RAIDu

RAID lze provozovat v podstatě dvojím způsobem. Buď je realizován v hardwaru, což obnáší speciální řadič osazený procesorem a zpravidla vybavený vlastní pamětí, která slouží jako cache. Veškeré funkce RAIDu plní řadič a z pohledu operačnímu systému se chová jako jediný disk. Tato řešení bývají poměrně drahá (Mylex, DPT – nyní Adaptec, ICP Vortex, velcí výrobci PC jako HP, IBM, Compaq apod. mají své vlastní implementace). Předností hardwarových řešení bývá maximální spolehlivost a ve srovnání se softwarovou variantou RAIDu dovedou odlehčit zátěži systému. RAID ovšem také může být realizován patřičným ovladačem na úrovni operačního systému a spousta operačních systémů to také dnes umožňuje. Toto řešení může být za jistých okolností flexibilnější a rychlejší, ale také náročnější na systémové prostředky – zejména na čas procesoru. V posledních letech se setkáváme i s napůl hardwarovými/softwarovými implementacemi RAIDu, kdy hardware obsahuje jen minimální podporu a většinu práce dělá ovladač; zatímco dříve existovaly hardwarové řadiče RAIDu pouze v provedení SCSI, nyní jsou dostupné také řadiče s rozhraním IDE. Tyto varianty jsou levné, ale řada produktů této kategorie je nevalné kvality a výkonu. Toto pojednání je zaměřeno na softwarový RAID pod Linuxem.

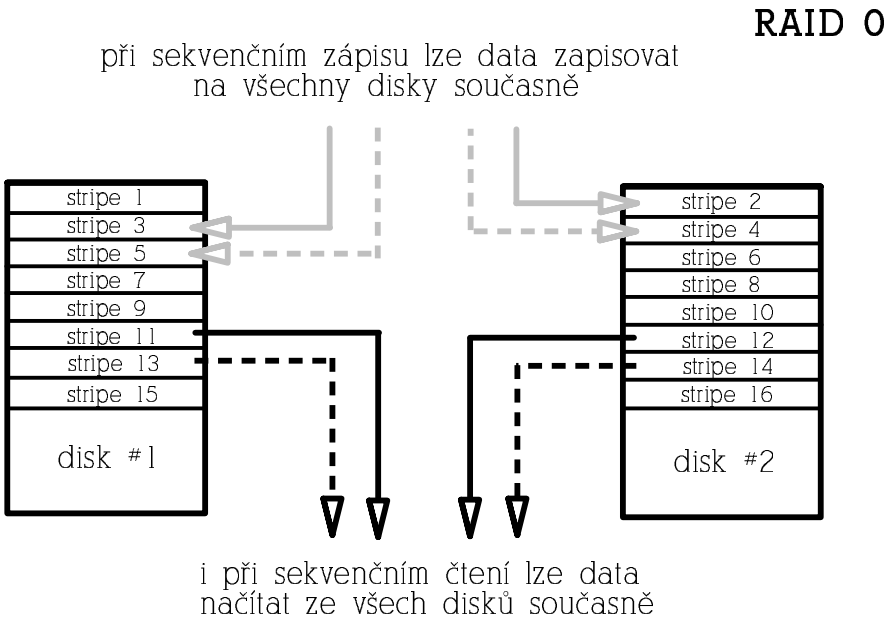
Jan Kasprzak, Petr Sojka (editoři): SLT 2002 – sborník semináře o Linuxu a TEXu, str. 181–196, 2002.

2 Teorie fungování RAIDu

Dříve než se zaměříme na detaily implementace softwarového RAIDu pod Linuxem, podíváme se na princip fungování jednotlivých typů RAIDu a jejich vlastnosti.

2.1 RAID 0 (Nonredundant striped array)

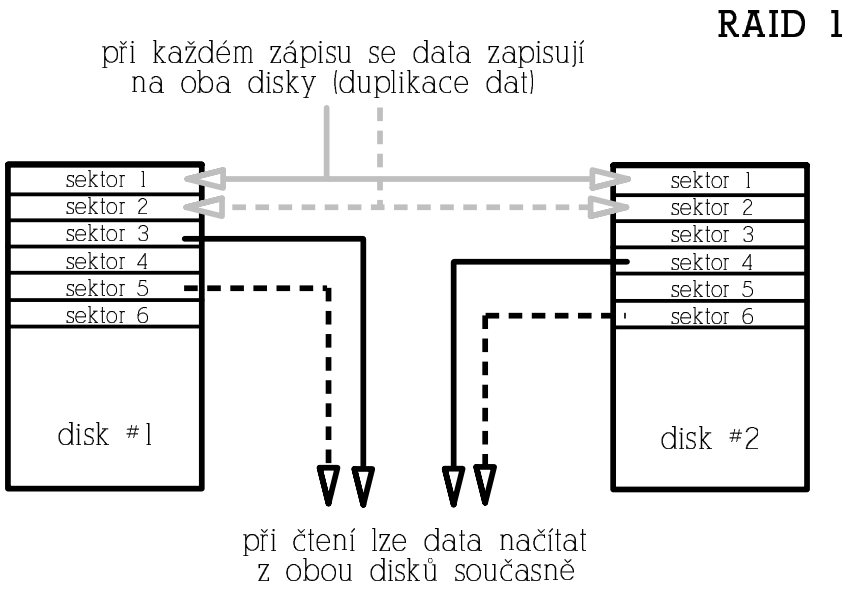
Tento typ je určen pro aplikace, které vyžadují maximální rychlost a není redundantní. Naopak je potřeba vzít v úvahu, že pravděpodobnost výpadku takového pole roste s počtem disků. Ideální použití představují audio/video streamingové aplikace, eventuálně databáze a obecně aplikace, při kterých čteme sekvenčně velká množství dat. Základní jednotkou pole je tzv. *stripe* (z angl. „stripe“, česky pruh), což je blok dat určité velikosti (běžně 4–64 kB v závislosti na aplikaci). Po sobě jdoucí data jsou pak v poli rozložena střídavě mezi disky do „stripů“ takovým způsobem, aby se při sekvenčním čtení/zápisu přistupovalo ke všem diskům současně. Tím je zajištěna maximální rychlost jak při čtení tak i zápisu, ale současně je tím dána také zranitelnost pole. Při výpadku kteréhokoliv disku se stávají data v podstatě nečitelná (respektive nekompletní). RAID 0 bývá označován rovněž jako *striping*. Protože není redundantní, má nejvýhodnější poměr cena/kapacita. Počet disků je libovolný. Je ovšem třeba pamatovat na to, že s rostoucím počtem disků v poli roste i pravděpodobnost výpadku pole (protože výpadek libovolného disku znamená havárii celého pole); RAID 0 je tedy velmi rychlý, ale méně bezpečný než samostatný disk.



Obrázek 1. Schéma diskového pole RAID 1

2.2 RAID 1 (Mirrored array)

RAID 1 je naopak maximálně redundantní. Rychlost čtení může být oproti samostatnému disku výrazně vyšší, rychlost zápisu je stejná jako u samostatného disku. Funguje tak, že data jsou při zápisu „zrcadlena“ na všechny disky v poli (tedy v případě RAIDu 1 tvořeného dvěma disku jsou data duplikována apod.). Při čtení lze využít vícero kopií dat a podobně jako u RAIDu 0 číst za všech disků současně. Tento typ pole je určen pro aplikace s důrazem na maximální redundanci. Výhodou tohoto redundantního řešení je stabilní výkon i v případě výpadku disku, nevýhodou je poměr cena/kapacita. Počet disků bývá buď 2 anebo libovolný, čím větší počet disků, tím větší redundance a odolnost proti výpadku.



Obrázek 2. Schéma diskového pole RAID 0

2.3 RAID 2 (Parallel array with ECC)

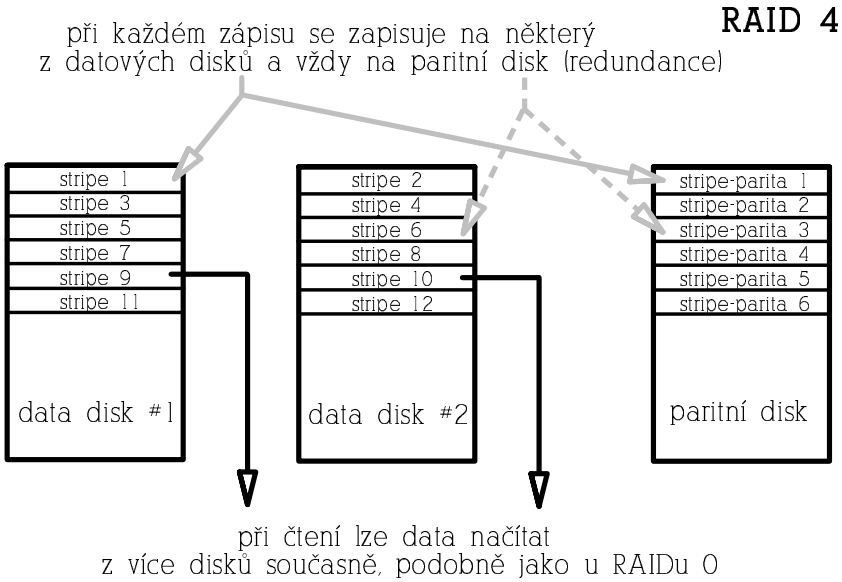
Pole tohoto typu jsou dnes již historií, protože dnešní disky mají vlastní opravné mechanismy a uchovávají ECC informace pro každý sektor samy. V polích tohoto typu se stripovalo po jednotlivých sektorech a část disků pole byla vyhrazena pro ukládání ECC informací. Jakékoliv čtení i zápis proto zpravidla zahrnovalo přístup ke všem diskům pole, což bylo překážkou vyššího výkonu zejména u aplikacích pracujících se většími kusy dat. Tento typ není redundantní.

2.4 RAID 3 (Parallel array with parity)

Také tento typ polí se již nepoužívá, jedná se o předchůdce RAIDu 4. Stripovalo se po sektorech, ale jeden disk byl vyhrazen jako paritní, což zajišťovalo redundanci (princip zajištění redundance je stejný jako u RAIDu 4 a 5, který je popsán níže). Protože i v tomto případě se stripovalo po sektorech, jakékoliv čtení i zápis zpravidla zahrnovalo přístup ke všem diskům pole.

2.5 RAID 4 (Striped array with parity)

RAID 4 je redundantní pole, které se dnes již používá málo. Funkčně je podobné RAIDu 5, který je ale výkonnější. Funguje tak, že jeden disk je vyhrazen jako tzv. *paritní disk*. Na paritním disku je zaznamenán kontrolní součet (operace XOR přes data stejné pozice jednotlivých disků). Pokud tedy dojde k výpadku některého z datových disků, lze data rekonstruovat z dat zbylých disků a parity uložené na paritním disku. RAID 4 je odolný vůči výpadku libovolného jednoho disku a má tedy příznivý poměr cena/kapacita. Paritní disk ale představuje úzké hrdlo této architektury při zápisech, protože každý zápis znamená také zápis na paritní disk. Minimální počet disků je 3.

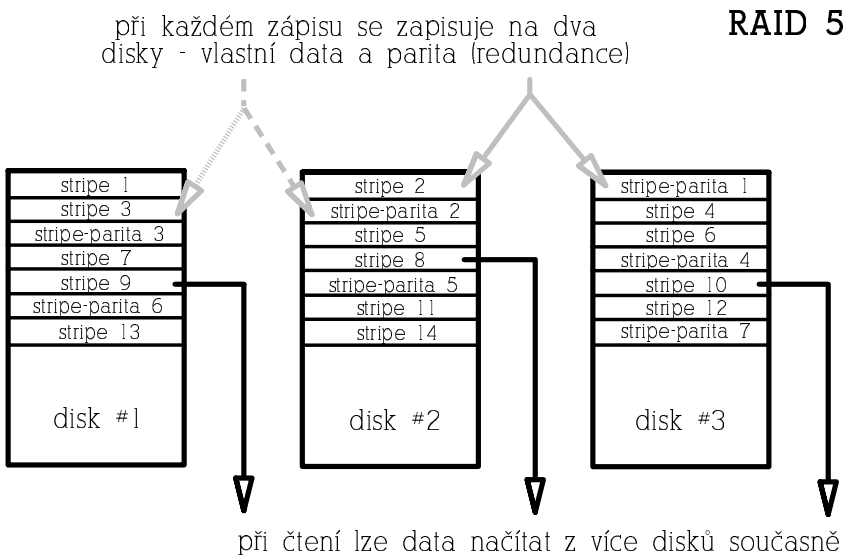


Obrázek 3. Schéma diskového pole RAID 4

2.6 RAID 5 (Striped array with rotating parity)

Tento typ poskytuje redundanci vůči výpadku libovolného jednoho disku s dobrým poměrem cena/kapacita a výkonem. RAID 5 je vylepšená varianta RAIDu 4

v tom, že parita není uložena na jednom vyhrazeném disku, ale je rozmístěna rovnoměrně mezi všemi disky pole, čímž se odstraní úzké hrdlo architektury RAIDu 4.



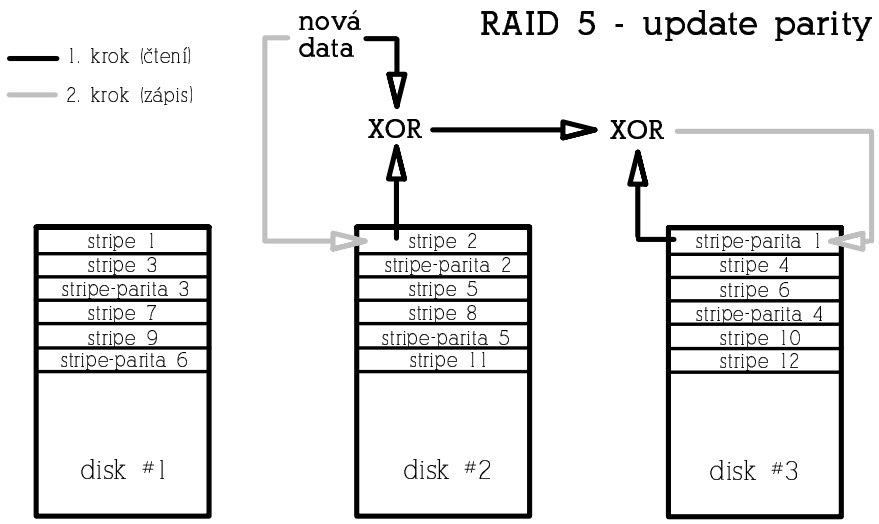
Obrázek 4. Schéma diskového pole RAID 5

Paritu lze spočítat buď tak, že skutečně načteme a XORujeme data z odpovídajících datových stripů všech disků (takto se parita počítá při inicializaci pole, vyžaduje to tedy přístup ke všem diskům). Ve druhém případě načteme původní data z datového stripu, která se mají změnit, provedeme XOR s novými daty a výsledek ještě XORujeme s původní hodnotou parity (takto se parita počítá na již inicializovaném běžícím poli). Zápis dat tedy představuje dvoje čtení (dat a parity), výpočet parity a dvojí zápis (opět dat a parity). Počet přístupů na disk při zápisu je v tomto případě konstantní bez ohledu na počet disků v poli – přistupuje se vždy ke dvěma diskům – a to také má za následek nižší výkon tohoto typu pole ve srovnání s redundantním RAIDem 1.

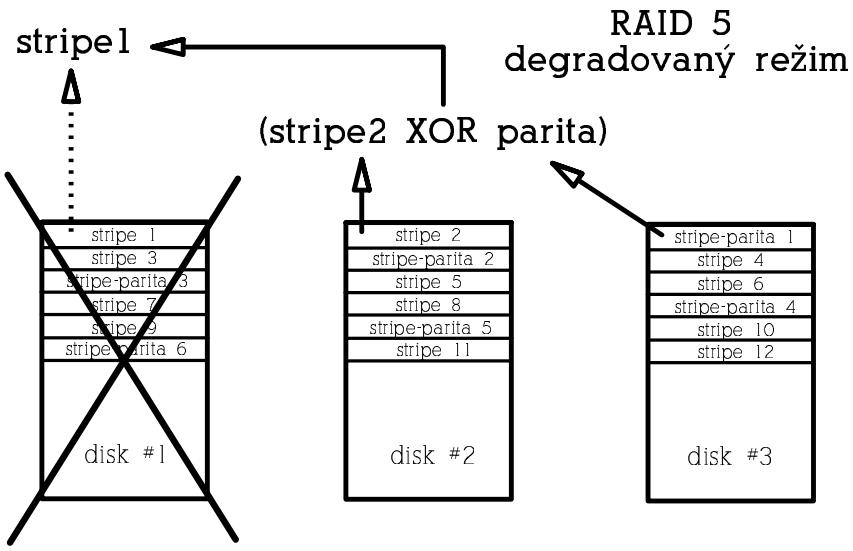
V degradovaném režimu (*degradovaný režim* znamená stav, kdy je některý z disků z pole vyřazen kvůli hardwarové chybě) se pak musejí data uložená na vadném disku odvodit z dat zbývajících disků a parity. Na rozdíl od redundantního RAIDu 1, kde výpadek disku obvykle neznamená výrazný pokles výkonu, vykazuje RAID 5 v degradovaném režimu výrazně snížený výkon zejména při čtení. Minimální počet disků pro tento typ diskového pole jsou 3.

2.7 Kombinace více typů polí

Z definic výše popsaných typů polí vyplývá, že redundantní pole nejsou tak rychlá, jak bychom si mohli přát a naopak u RAIDu 0 nám chybí redundance. Existuje ovšem možnost, jak výhody jednotlivých typů diskových polí spojit. Tato



Obrázek 5. Výpočet parity u diskového pole RAID 5



Obrázek 6. Degradovaný režim u diskového pole RAID 5

metoda spočívá ve vytvoření kombinovaných polí, kdy disky v poli určitého typu jsou samy tvořeny poli jiného typu. Příkladem může být např. RAID 1+0, kdy jsou pole typu RAID 1 dále sloučeny do RAIDu 0. Takové pole je pak redundantní (toleruje výpadek až dvou disků), rychlejší zejména v zápisech než samotný RAID 1, a má lepší poměr cena/kapacita než RAID 1 (velikost je zde $n/2 * \mathrm{disk}$ a minimální počet disků je pak 4). Další možností je třeba RAID 0+1, RAID 5+0, RAID 5+1 apod.

3 Jak RAID ošetří výpadek disku

Všechny typy redundantních polí obvykle umožňují nakonfigurovat kromě *aktivních disků* ještě 1 či více *rezervních disků*. Aktivními disky zde rozumíme disky, které jsou součástí funkčního pole. V případě výpadku některého z aktivních disků pak může systém místo vadného disku okamžitě začít používat disk rezervní. Po aktivaci rezervního disku do pole systém provede na pozadí (tedy bez narušení dostupnosti pole) rekonstrukci pole a jakmile je rekonstrukce hotova, je pole opět plně redundantní. Rekonstrukcí je míněna buď synchronizace obsahu nového disku s ostatními aktivními disky (v prípadě RAIDu 1), anebo rekonstrukce obsahu původního vadného disku na základě redundantní informace (jedná-li se o RAID 4 nebo 5). Po dobu, než rekonstrukce proběhne, se pole nachází v tzv. *degradovaném režimu*, kdy v závislosti na konfiguraci nemusí být redundantní a v případě RAIDu 5 se to projeví sníženým výkonem (odtud název „degradovaný režim“). Je samozřejmě možné pole provozovat bez rezervních disků a disk vyměnit později manuálně. Detailně se na výměnu disků a rekonstrukci polí ještě podíváme později.

4 Redundantní pole neznamenají konec záloh

Redundantní disková pole jsou odolná pouze vůči výpadkům určitého počtu disků. Neochrání před výpadkem napájení, poškozením souborového systému při pádu celého systému nebo chybou administrátora. Proto je potřeba myslet i na další metody ochrany dat – např. na záložní zdroje napájení (UPS), žurnálovací souborové systémy apod. a v každém případě pravidelně zálohovat.

5 Typy polí podporovaných Linuxovým ovladačem RAIDu

Až doposud jsme se zabývali pouze teorií fungování diskových polí, podívejme se tedy jak to vypadá se softwarovým RAIDem v Linuxu. Softwarová implementace RAIDu pod OS Linux podporuje 5 typů diskových polí:

– *Linear*: toto v podstatě není RAID, jedná se o možnost „pospojování“ více disků do jednoho. Co se týče rychlosti, neliší se příliš od výkonu samostatných disků. Není redundantní, výhodou však je např. oproti RAIDu 0 to, že se v případě havárie jednoho z disků dají data ze zbývajících disků snáze obnovit, protože se nestripuje.

- *RAID 0*: zde je volitelná velikost stripu; tato výrazně (v závislosti na aplikaci) ovlivňuje výkon RAIDu.
- *RAID 1*: oproti klasické definici RAIDu 1 kdy se disky spojují pouze do párů, pod Linuxem můžeme vytvořit RAID 1 i z více než dvou aktivních disků.
- *RAID 4*: je sice implementován, ale funkčně nahrazen RAIDem 5
- *RAID 5*: opět je zde volitelná velikost stripu, která ovlivňuje výkon RAIDu.

6 Konfigurace

Konfigurace RAIDu verze 0.90 používá konfigurační soubor `/etc/raidtab`, ve kterém se používají následující direktivy:

- `raiddev`: touto direktivou definice pole začíná, následuje označení pole. Svazky softwarového RAIDu se označují `md1` až `mdX`.
- `raid-level`: následuje direktivu `raiddev`, uvádíme zde typ pole (-1 = linear, 0 = RAID 0, 1 = RAID 1, 5 = RAID 5).
- `persistent-superblock`: tato direktiva bude popsána níže, týká se kompatibility se starší verzí RAIDu.
- `chunk-size`: velikost stripu, maximální velikost je 4 MB (což je dáno konstantou `MAX_CHUNK_SIZE` ovladače), udává se v kB.
- `nr-raid-disks`: zde uvádíme kolik diskových oddílů bude součástí pole.
- `nr-spare-disks`: počet rezervních disků v poli
- direktiva device `jméno_oddílu` následovaná jednou z direktiv `raid-disk`, `spare-disk`, `parity-disk` nebo `failed-disk`: tyto direktivy deklarují příslušné oddíly, které budou součástí pole.
- `raid-disk`: tento oddíl bude aktivním oddílem.
- `spare-disk`: tento oddíl bude sloužit jako rezervní.
- `parity-disk`: tento oddíl bude sloužit jako paritní disk (RAID 4).
- `failed-disk`: tento oddíl bude při inicializaci pole přeskočen (má význam pouze při sestavování pole v degradovaném stavu, viz diskuse níže).
- `parity-algorithm`: specifikuje schéma rozložení parity u RAIDu 5 (možnosti jsou: `left-symmetric`, `right-symmetric`, `right-asymmetric`; z podporovaných je obecně nejrychlejší `left-symmetric`).

7 Obslužný software – Raidtools

Balíček raidtools obsahuje obslužné utility nezbytné k manipulaci s diskovými poli:

- `mkraid`: pro inicializaci polí;
- `raidstart`: pro spouštění diskových polí;
- `raidstop`: pro vypnutí diskových polí;
- `raidhotadd`: přidá nový diskový oddíl do aktivního diskové pole (náhradou za vadný oddíl, pokud jsou všechny oddíly pole funkční, přidá nový oddíl jako rezervní – „spare-disk“). Nelze tedy použít pro rozšíření kapacity pole;

- `raidhotremove`: odejme vadný diskový oddíl z aktivního diskového pole;
- `raidsetfaulty`: označí funkční diskový oddíl jako vadný, tím umožní jeho odejmutí z pole příkazem `raidhotremove` (možné využití např. při testování nebo výměnách funkčních disků);
- `raid0run`: utilita pro spouštění starších polí typu linear nebo RAID 0 bez perzistentních superbloků (viz níže „Perzistentní superbloky a RAID 0/linear“).

Poznámka na vysvětlenou: Soubor `raidtab` odráží konfiguraci polí v době jejich sestavení, ovšem pokud třeba později vyměníme nebo přesuneme některé disky, nemusí již odrážet skutečnou konfiguraci. Pokud tedy z nějakého důvodu potřebujeme pole znovu inicializovat anebo ho jen startujeme pomocí `raidstart`, nesmíme zapomenout soubor `raidtab` ručně upravit, abychom se ušetřili v budoucnu nepříjemností.

8 Obslužný software – Mdadm

V budoucnu zřejmě budou utility z balíčku `raidtools` nahrazeny jedinou utilitou `mdadm`, kterou vyvíjí Neil Brown. Utilita `mdadm` nemusí používat žádný konfigurační soubor, vše potřebné lze zadat na příkazové řádce, anebo to zjistí analýzou RAID superbloků uložených na discích (podobně funguje automatické startování polí jádrem při bootu). Cílem autora je tedy přidat výhody a robustnost, kterou poskytuje vlastnost samosestavení diskových polí jádrem a zároveň se vyhnout potenciálním konfliktům mezi neaktuální konfigurací v souboru `raidtab` a skutečnou konfigurací polí, ke kterým časem může dojít, pokud používáme raidtools. Nástroj `mdadm` je velmi flexibilní a po více než roce vývoje je již poměrně stabilní, takže jeho použití lze doporučit.

9 Inicializace polí

Jakmile máme odpovídajícím způsobem rozdělené disky a připravený konfigurační soubor `/etc/raidtab`, můžeme pole inicializovat utilitou `mkraid`, která pole sestaví a aktivuje. Pokud zakládáme pole s perzistentními RAID superbloky (viz níže), pak `mkraid` vypíše i pozici RAID superbloků:

```
# mkraid  /dev/md5

handling MD device /dev/md5
analyzing super-block
disk 0: /dev/sda7, 20163568kB, raid superblock \
        at 20163456kB
disk 1: /dev/sdb7, 20163568kB, raid superblock \
        at 20163456kB
```

Po úspěšné inicializaci bychom měli v souboru `/proc/mdstat`, který obsahuje informace a aktivních polích vidět odpovídající záznam, např:

```
# cat /proc/mdstat

Personalities : [raid0] [raid1] [raid5]
read_ahead 1024 sectors
md0: active raid1 sdb1[1] sda1[0] 131968 blocks [2/2] [UU]
```

Poté nám nic nebrání pole zformátovat např. pomocí `mke2fs`, pokud chceme na poli provozovat souborový systém ext2. Utilita `mke2fs` akceptuje volbu `-R stride=X`, která udává kolik bloků souborového systému obsahuje 1 „stripe“ pole. Tím pádem je také vhodné zadat ručně velikost bloku (parametr `-b`). Např. mějme pole typu RAID 0 s velikostí stripu 16 kB. Pokud budeme chtít použít velikost bloku souborového systému 4 kB, zadáme:

```
mke2fs /dev/md0 -b 4096 -R stride=4
```

10 Raid autodetect anebo raidstart?

Nyní tedy máme funkční diskové pole. Zbývá vyřešit způsob, jakým se bude pole vypínat při vypnutí systému a zapínat při startu systému. Jednou možností je použití utilit raidstart a raidstop. Pomocí těchto utilit můžeme pole aktivovat či zastavit kdykoliv, stačí tedy upravit příslušné startovací skripty. (Pokud už distribuce toto neobsahuje; např. distribuce Red Hat není potřeba upravovat, ze skriptu `/etc/rc.d/rc.sysinit` je `raidstart` volán automaticky, existuje-li soubor `/etc/raidtab` a `raidstop` je volán ze skriptu `/etc/rc.d/init.d/halt`.).

Druhou, robustnější metodou je využití možnosti automatické aktivace polí jádrem při bootu. Aby mohla fungovat, je potřeba v prvé řadě používat perzistentní RAID superbloky a všechny diskové oddíly, které jsou součástí polí, musejí být v tabulce oddílů označeny jako typ Linux raid autodetect (tedy hodnota `0xfd` hexadecimálně).

Výhodou tohoto řešení navíc je, že jakmile je diskové pole inicializováno, nepoužívá se již pro opětovný start/zastavení pole konfigurační soubor `/etc/raidtab`. O sestavení a spuštění pole se postará ovladač RAIDu, který na všech diskových oddílech typu `Linux raid autodetect` vyhledá RAID superbloky a na základě informací v RAID superblocích pole spustí. Stejně tak ovladač RAIDu všechny pole korektně vypne v závěrečné fázi ukončení běhu systému poté, co jsou odpojeny souborové systémy. I v případě změny jmen disků nebo po přenesení disků na úplně jiný systém tedy pole bude korektně sestaveno a nastartováno.

11 RAID superblok

Každý diskový oddíl, který je součásti raid svazku (výjimku tvoří pouze svazky bez perzistentních superbloků, viz níže) obsahuje tzv. *RAID superblok*. Tento superblok je 4 kB část RAID oddílu vyhrazená pro informace o příslušnosti daného oddílu k určitému poli a o stavu pole.

Prohlédnout si jej můžeme např. pomocí utility od (s vhodnými parametry, např. `od -Ax -tx4`) poté, co si jej příkazem `dd` někam zkopírujeme. Pro kontrolu, superblok vždy začíná „magickým číslem“ `0xa92b4efc`. Superblok obsahuje zejména následující informace:

- verzi ovladače raidu, kterým byl vytvořen,
- jedinečný identifikátor pole,
- typ RAIDu,
- datum vytvoření RAID svazku,
- počty disků (aktivních, rezervních apod.),
- preferované vedlejší číslo RAID zařízení,
- stav pole,
- kontrolní součet superbloku,
- počet updatů superbloku,
- datum posledního updatu superbloku,
- velikost stripu,
- informace o stavu jednotlivých diskových oddílů.

12 Perzistentní superbloky a RAID 0/linear

Pokud provozujeme pole RAID 0 či linear, máme možnost zvolit variantu bez použití perzistentního superbloku. Volba `persistent-superblock 0`, znamená, že se RAID superblok nebude ukládat na disk. Tato možnost existuje z důvodů zachování kompatibility s polemi zřízenými pomocí starší verzí ovladače RAIDu. Po vypnutí takového pole nezůstane na svazku informace o konfiguraci a stavu pole. Proto je tato pole nutné vždy znovu inicializovat při každém startu buď utilitou `mkraid`, nebo pomocí utility `raid0run` (což je pouze symbolický odkaz na `mkraid`) a nelze využít automatického startování polí jádrem při bootu.

Poznámka: Na tuto volbu je třeba dávat pozor při konfiguraci – pokud při konfiguraci pole RAID 0 direktivu `persistent-superblock` vynecháme, použije se standardní hodnota 0, tedy pole bez perzistentních superbloků!

13 Monitorování stavu pole

Aktuální stav diskových polí zjistíme vypsáním souboru `/proc/mdstat`. První řádek obsahuje typy polí, které ovladač podporuje (záleží na konfiguraci jádra). U jednotlivých RAID svazků je pak uvedeno které diskové oddíly svazek obsahuje, velikost svazku, u redundantních polí pak celkový počet konfigurovaných oddílů a z toho počet funkčních, následovaný schématem funkčnosti v hranatých závorkách. Následující příklad uvádí stav funkčního pole RAID 1:

```
md0: active raid1 hdc1[1] hda1[0] 136448 blocks [2/2] [UU]
```

Druhý příklad uvádí stav pole RAID 1 po výpadku jednoho disku, oddíl `sdc1` je označen jako nefunkční (`F=Failed` místo čísla aktivního oddílu):

```
md0: active raid1 sdc1[F] sdd1[0] 8956096 blocks [2/1] [U_]
```

Třetí příklad ukazuje stav pole RAID 1, kdy probíhá rekonstrukce:

```
md1: active raid1 hdc2[1] hda2[0] 530048 blocks [2/2] \
[UU] resync=4\% finish=6.7min
```

Součástí raidtools bohužel není utilita k monitorování stavu diskových polí, takže si administrátor musí vypomoci skriptem, který je pravidelně spouštěn z cronu a kontroluje `/proc/mdstat` (jednoduše např. tak, že si skript na disk uloží obsah `/proc/mdstat` nebo jeho MD5 součet a následně kontroluje, jestli se `/proc/mdstat` změnil; v případě změny pak prostřednictvím emailu uvědomí administrátora) anebo filtrem systémového logu.

14 Rekonstrukce pole

Redundantní typy polí je třeba po inicializaci, po výměně disku, nebo po nahrazení vadného disku rezervním (viz direktiva `spare-disk` v `/etc/raidtab`) rekonstruovat či synchronizovat. Ve všech případech systém rekonstrukci spouští automaticky. Průběh rekonstrukce je možné sledovat v `/proc/mdstat` (viz příklad o několik řádek výše). Rekonstrukce probíhá s nízkou prioritou, nezabere tedy čas procesoru na úkor jiných aplikací, ale bude se snažit využít maximální prostupnosti I/O zařízení. Proto můžeme po dobu rekonstrukce pozorovat zpomalení diskových operací. Maximální rychlost rekonstrukce ovšem také lze ovlivnit nastavením limitu v `/proc/sys/dev/md/speed-limit`, výchozí hodnota je 100 kB/sec. Ovladač RAIDu umí současně spustit rekonstrukci na několika polích současně. Pokud jsou však oddíly jednoho disku součástí více polí, které by se měly synchronizovat současně, provede se synchronizace polí postupně (z důvodu výkonu). V systémovém logu se pak objeví neškodné hlášení typu „XX has overlapping physical units with YY“:

```
md: syncing RAID array md1
md: minimum _guaranteed_ reconstruction speed: 100 KB/sec.
md: using maximum available idle IO bandwith for \
    reconstruction.
md: using 128k window.
md: serializing resync, md2 has overlapping physical \
    units with md1!
md: md1: sync done.
md: syncing RAID array md2
md: minimum _guaranteed_ reconstruction speed: 100 KB/sec.
md: using maximum available idle IO bandwith for \
    reconstruction.
md: using 128k window.
md: md2: sync done.
```

V /proc/mdstat jsou ty svazky, na kterých je rekonstrukce pozastavena, označeny jako plně funkční, ale je u nich poznámka resync=DELAYED:

```
Personalities : [linear] [raid0] [raid1] [raid5]
read_ahead 1024 sectors
md2 : active raid1 hdc3[1] hda3[0] 530048\
 blocks [2/2] [UU] resync=DELAYED
md1 : active raid1 hdc2[1] hda2[0] 530048\
 blocks [2/2] [UU] resync=4\% finish=6.7min
md0 : active raid1 hdc1[1] hda1[0] 136448\
 blocks [2/2] [UU]
```

15 Redundantní pole: výměna disku, hot plug

Pokud při čtení nebo zápisu na některý z diskových oddílů, který je součástí redundantního diskového pole, dojde k chybě, je dotyčný oddíl označen jako vadný a pole jej přestane používat. Pokud máme v daném diskovém poli zařazen jeden nebo více rezervních disků (direktiva spare-disk), je tento v případě výpadku automaticky aktivován, systém provede rekonstrukci pole a průběh rekonstrukce zaznamená do systémového logu. V opačném případě pole zůstane v provozu v degradovaném režimu, pak to v systémovém logu bude vypadat zhruba takto:

```
kernel: SCSI disk error : host 0 channel 0\
 id 4 lun 0 return code = 28000002
kernel: [valid=0] Info fld=0x0, Current sd08:11:\
 sense key Hardware Error
kernel: Additional sense indicates\
 Internal target failure
kernel: scsidisk I/O error:\
 dev 08:11, sector 2625928
kernel: raid1: Disk failure on sdb1,\
 disabling device.
kernel: Operation continuing on 1 devices
kernel: md: recovery thread got woken up ...
kernel: md0: no spare disk to reconstruct\
 array! - continuing in degraded mode
kernel: md: recovery thread finished ...
```

Příjemnou vlastností diskových polí je také možnost výměny disku za chodu systému. Samozřejmě k tomu potřebujeme v prvé řadě hardware, který to umožňuje. Ovladače slušných SCSI řadičů umožňují přidávání či ubírání zařízení, to ale samo o sobě nestačí. Je zapotřebí používat SCA disky určené pro „hot swap“ a odpovídající SCSI subsystém s SCA konektory a elektronikou, která zajistí stabilitu SCSI sběrnice při odebírání či přidávání zařízení.

Mějme pole typu RAID 1, ve kterém došlo k chybě na oddílu sdc1. Disk sdc1 je připojen ke kanálu 0 SCSI řadiče 0 a má ID rovno 4:

```
md0: active raid1 sdc1[F] sdd1[0] 8956096 blocks [2/1] [U_]
```

Jak tedy probíhá výměna vadného disku, máme-li k tomu potřebné hardwarové vybavení:

- provedeme `raidhotremove /dev/md0 /dev/sdc1`, což vyřadí vadný oddíl z pole md0,
- provedeme příkaz
  `echo "scsi remove-single-device 0 0 4 0" >/proc/scsi/scsi`
  ovladač SCSI řadiče „zapomene“ na zařízení na řadiči 0, kanálu 0, ID 4, LUN 0,
- vyjmeme vadný disk sdc,
- vložíme nový disk sdc,
- dále vykonáme příkaz
  `echo "scsi add-single-device 0 0 4 0" >/proc/scsi/scsi`
  což nový disk zpřístupní systému,
- pomocí utility fdisk vytvoříme diskové oddíly,
- vykonáme `raidhotadd /dev/md0 /dev/sdc1`, čímž přidáme oddíl sdc1 nového disku do pole md0 a na pozadí se spustí rekonstrukce pole.

Pokud nemáme hardware potřebný k „hot-swap“ výměně disků, musíme se smířit s vypnutím systému, výměnou vadného disku a opětovným zapnutím systému. Potom stačí pouze vytvořit pomocí fdisku odpovídající diskové oddíly a příkazem raidhotadd je zařadit do diskového pole. Příkazy pro přidávaní a ubírání SCSI zařízení jsou popsány ve zdrojovém kódu jádra (soubor `linux/drivers/scsi/scsi.c`).

16 Možnosti rekonfigurace polí

Pro offline konverzi jednoho typu RAIDu do jiného lze použít nástroje `raidreconf`. Tento nástroj je ovšem stále ještě ve vývoji a není zcela stabilní, použití zálohy dat je v tomto případě nutností. Změníme-li velikost diskového pole, musíme pak ještě upravit velikost souborového systému (pokud používáme souborový systém `ext2`, můžeme použít `ext2resize` nebo `resize2fs`).

17 Optimalizace polí

Výkon diskových polí velmi záleží na vhodně zvolené velikosti stripu (to je samozřejmě aplikovatelné u polí RAID 0 a RAID 5). Jak jsem již zmínil, také souborový systém ext2 lze optimalizovat pro užití na diskovém poli; nástroji `mke2fs` pro zakládání ext2 filesystémů lze pomocí volby `-R stride=XX` zadat, kolik bloků souborového systému odpovídá velikosti stripu.

18 Výkon a stabilita

Nejprve srovnejme výkon softwarového raidu pod jádry 2.2.x a 2.4.x: RAID 0 je rychlejší u jader 2.4, RAID 1 je na tom zhruba stejně, RAID 5 byl na řadě 2.4 z počátku výrazně pomalejší, ale nyní je výkon srovnatelný nebo lepší. Pokud jde o srovnání rychlosti softwarového RAIDu a hardwarových řešení, softwarový RAID je oproti hardwarové implementaci samozřejmě náročnější na systémové prostředky, ale na druhou stranu bývá mnohdy rychlejší (výrazně rychlejší bývá zejména RAID 0).

Pokud jde o robustnost implementace, stabilita RAIDu typů linear, RAID 0 a 1 je poměrně vysoká, naopak nasazení RAIDu 5 v ostrém provozu ještě nelze doporučit. V této souvislosti ještě zmíním jednu vlastnost Linuxové implementace softwarového RAIDu: V případě jakékoliv I/O chyby ovladač RAIDu okamžitě daný diskový oddíl z RAIDu vyřadí, bez ohledu na to, jestli se jedná o chybu fatální, anebo o případ, kdy by třeba stačilo danou I/O operaci zopakovat. Jinými slovy disk, který občas vrátí nějakou chybu, ale je nadále více méně schopný fungovat (a který by systém nadále používal, pokud by nebyl součástí RAID svazku, ale byl připojený jako samostatný oddíl), linuxový ovladač přestane používat. Tím se zbytečně snižuje robustnost RAIDu, protože snadněji může dojít k situaci, kdy z pole vypadne postupně i více disků, než kolik je k provozu daného pole třeba a pole zhavaruje. Proto lze doporučit použití rezervních disků a vyhnout se shánění rezervního disku na poslední chvíli, kdy už pole mezitím běží v degradovaném režimu. Ze srovnání softwarových RAID implementací Linuxu, Windows 2000 a Solarisu vyplývá, že linuxový RAID ve výchozím nastavení provádí rekonstrukci se sníženou prioritou a limitovanou rychlostí, takže probíhající rekonstrukce mnohem méně negativně ovlivňuje výkon systému po dobu rekonstrukce. (Poznámka: V odkazovaném srovnání ovšem autoři opakovaně chybně uvádějí absenci některých vlastností linuxové softwarové implementace RAIDu.)

19 Závěrem

Softwarový RAID je cenově lákavou alternativou nákladných hardwarových řešení. Další výhodou je flexibilita (např. možnost sestavení pole v degradovaném režimu, možnost eventuální částečné záchrany dat v případě výpadku celého pole, protože je známá struktura dat v diskovém poli, konverze RAID svazků z jednoho typu RAIDu na jiný). Některé z těchto možností jsou ale spíše experimentálního rázu. Za spolehlivé lze označit implementace RAIDu typu linear, RAID 0 nebo RAID 1. Softwarový RAID je náročnější na systémové prostředky než hardwarová řešení, některé typy (zejména RAID 0) ovšem mohou být výrazně rychlejší než hardwarové varianty. Je tedy na adminstrátorovi, aby zvážil výhody a nevýhody sofwarového či hardwarového RAIDu vzhledem k aktuálním podmínkám.

Tento článek ani v nejmenším nenahrazuje dokumentaci k ovladači Linuxového softwarového RAIDu či obslužným utilitám – proto zde až na výjimky

záměrně nejsou komentovány přepínače obslužných utilit. Důkladné čtení dokumentace (nebo v případě nejasností studium zdrojového kódu – dokumentace bohužel stále není úplná) by mělo být samozřejmostí, rovněž existuje konference `linux-raid` s prohledávatelným archívem. A ještě úplně poslední poznámka na závěr: nezapomínejme, že (redundantní) RAID chrání pouze před výpadkem určitého počtu disků a rozhodně nenahrazuje nutnost pravidelného zálohování dat.

Reference

1. HOWTO aktuální verze ovladače RAIDu:
   `http://www.linux.cz/linuxdoc/HOWTO/Software-RAID-HOWTO.html`
2. Boot+Root+Raid+LILO HOWTO:
   `http://www.linux.cz/linuxdoc/HOWTO/Boot+Root+Raid+LILO.html`
3. Mdadm – nástupce raidtools:
   `http://www.cse.unsw.edu.au/~neilb/source/mdadm/`
4. Srovnání implementací SW RAIDu:
   `http://www.cs.berkeley.edu/~abrown/papers/usenix00/paper.html`
5. Archív konference linux-raid:
   `http://marc.theaimsgroup.com/?l=linux-raid`

Náročné numerické výpočty na linuxovém klastru a porovnání s jinými platformami

Ondřej Jakl, Karel Krečmer

[1] Ústav geoniky AV ČR
Email: jakl@ugn.cas.cz,
[2] VŠB–TU, Ostrava
Email: krecmer@ugn.cas.cz

Abstrakt: Text[3] se zabývá klastry pracovních stanic v kontextu jejich použití pro paralelní výpočty. Zaměřuje se na dnes velmi populární systémy postavené na bázi osobních počítačů a využívající operační systém Linux. Podává praktické informace o stavbě linuxového klastru na Ústavu geoniky AV ČR, určeného pro náročnější numerické výpočty, a na vybraných úlohách matematického modelování demonstruje jeho výpočetní kapacity jak pro sekvenční, tak především pro paralelní zpracování. Porovnání s dalšími platformami potvrzuje, že takové klastry mohou výkonem předčit podstatně dražší komerční systémy.

1 Úvod

Náš příspěvek volně navazuje na článek *Clusterová řešení na Linuxu* [2] z předchozího ročníku semináře SLT, který informoval o různých možnostech nasazení linuxových klastrů. Tyto klastry mohou svou rostoucí popularitu zakládat na výkonném a přitom levném hardwaru osobních počítačů, vyzrálosti a modularitě OS Linux a volné dostupnosti většiny potřebných softwarových nástrojů. Jednou z oněch možností nasazení je HPC, *high performance computing*, tj. oblast náročných numerických výpočtů, která dnes prakticky bez výjimky předpokládá paralelní zpracování.

Připomeňme, že pod (nepěkným) slovem „klastr“ rozumíme architekturu, již tvoří kolekce samostatných počítačů (*uzlů*), schopných sice svébytné existence, které však propojovací subsystém a centrální řízení slučuje do jednoho funkčního celku. Klastry bývají specializovány na určitý typ výpočetní zátěže, v případě *výpočetních* klastrů na paralelní zpracování.

Jsou-li uzly výpočetního klastru osobní (nebo obecněji ve velkých sériích vyráběné) počítače, říká se klastru *beowulf*.[4] Beowulfy přinesly do oblasti systémů pro paralelní výpočty nová měřítka co do poměru cena/výkon, takže

[3] Tato práce je podporována granty AV ČR S3086102, FRVŠ 529/2002, MŠMT CEZ:J17/98:2724019

[4] Na počest prvního takového klastru, který sestavili v roce 1994 T. Sterling a D. Becker v rámci projektu NASA, viz [3]. Jméno si zřejmě vypůjčili ze staré balady popisující dobrodružství stejnojmenného severského hrdiny ze 6. století n. l.

Jan Kasprzak, Petr Sojka (editoři): SLT 2002 – sborník semináře o Linuxu a TEXu, str. 197–207, 2002.

paralelní počítání se stalo mnohem dostupnější. Klastry se začaly šířit do výzkumné a akademické sféry, kde je největší hlad po relativně levných, ale výkonných a flexibilních paralelních platformách. ČR není, zejména v poslední době, výjimkou: Klastry dnes provozuje hned několik univerzitních výpočetních center, např. na Masarykově univerzitě v Brně, Západočeské univerzitě v Plzni nebo na VŠB – Technické univerzitě Ostrava, a jsou dostupné pro širokou akademickou obec. Díky relativně nízkým pořizovacím nákladům už pronikají i do menších pracovních skupin. To je také případ beowulfu, který byl instalován ve středisku aplikované matematiky (SAM) Ústavu geoniky Akademie věd ČR v Ostravě a na který se zaměřuje tento příspěvek. Prezentuje zkušenosti malého kolektivu, kde chod lokální výpočetní techniky včetně klastru zajišťují pracovníci „amatérsky“, mimo rámec svých hlavních pracovních povinností.

Zájem o výpočetní klastry byl v SAM motivován potřebou paralelního zpracování rozsáhlých simulací v rámci konečněprvkového systému GEM32, který je ve středisku dlouhodobě rozvíjen pro potřeby základního výzkumu i zpracování praktických úloh. Vývoj jeho iteračního řešiče (soustav lineárních rovnic), založeného na metodě sdružených gradientů s předpodmíněním, pokročil z paralelizace na bázi rozložení posunutí, zaměstnávající nejvýše čtyři procesory, na rozložení oblasti s potenciálem prakticky neomezené škálovatelnosti ve smyslu využití procesorů (více v sekci 4.1). O větším počtu (desítkách) procesorových uzlů lze však v našich podmínkách t.č. uvažovat právě jen v rámci beowulfů.

2 Stavíme klastr thea

Budování našeho klastru bylo v režii omezených financí. Ty nás přiměly rozložit ho do třech *etap*, nicméně po technické stránce, díky všestranné modularitě, to nepředstavovalo žádný problém. Považujeme to za nezanedbatelnou výhodu klastrové architektury.

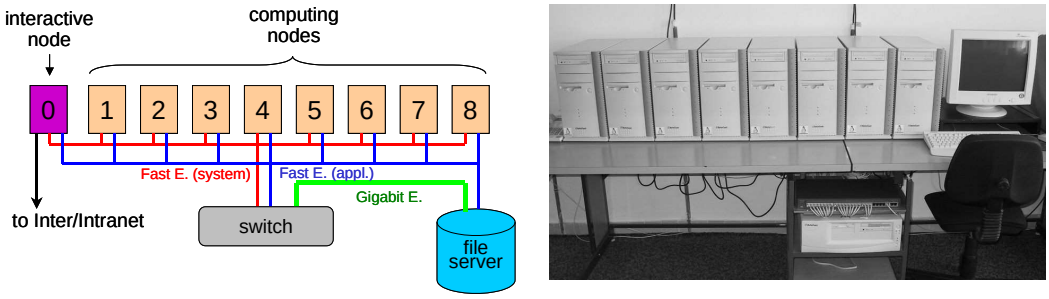
První etapa se realizovala do ledna 2002 a jejím výsledkem byl základ klastru v podobě osmi identických stanic AutoCont, vybraných na základě výkonnostního testu. Do do limitu 40 tis. Kč (vč. DPH) se tehdy „vešla“ následující konfigurace:

- základní deska ASUS A7V133-C
- procesor Athlon 1.4 GHz 266 MHz FSB
- paměť 768 MB SDRAM, PC133
- disk 20 GB, 7200 ot./min.
- síťový adaptér 3Com Fast Etherlink XL 10/100 PCI

Poznamenejme, že procesory AMD Athlon se v našich testech, intenzivních na operace v pohyblivé řádové čárce, ukázaly být výkonnější než cenově srovnatelné procesory Intel.

Tyto stanice byly propojeny prostřednictvím přepínače Cisco Catalyst 2950T (24 plně duplexní porty Fast Ethernet 100 Mbit/s a 2 uplinky Gigabit Ethernet 1000 Mbit/s, rovněž do 40 tis. Kč). Fast Ethernet je dnes ekonomickou volbou pro komunikační subsystém klastru – v našem případě finančně nepřicházelo

v úvahu lepší řešení. Úlohu souborového serveru i interaktivního uzlu („front-endu“, sloužícího jako vstupní brána ke klastru), dočasně převzal osmý uzel. Sestavu doplňovala jedna sada vstupně-výstupních periferií. Celkově bylo v první etapě proinvestováno necelých 365 tis. Kč. Po instalaci a konfiguraci OS a dalšího potřebného softwaru (viz následující sekce) byl klastr, nazvaný *thea*, už v této podobě připraven plnit své funkce a také výsledky, které jsou prezentovány níže, pocházejí z tohoto období.



Obrázek 1. Klastr thea: schéma a skutečnost

V **druhé etapě** (do května 2002) byl klastr za necelých 52 tis. Kč doplněn o dva nové prvky: Přibyla devátá stanice (označena číslem 0), která začala plnit úlohu interaktivního uzlu a dočasně i souborového serveru, a uvolnila tak uzel č. 8 pro výpočty. Až na tři rozhraní Fast Ethernet se neliší od výpočetních uzlů, a představuje tak zároveň určitou zálohu pro případ hardwarové poruchy některého z nich. Výpočetní uzly byly v souladu se schématem na obr. 1 vlevo doplněny o druhé síťové rozhraní (karta Intel EtherExpress Pro/100+) a oddělena systémová (hlavně NFS) a aplikační (předávání zpráv) komunikace. Současný vzhled klastru thea po 2. etapě výstavby je na obr. 1 vpravo.

Konečnou podobu získá klastr ve **třetí etapě** do konce roku 2002 instalací souborového serveru. Ten jako jediný bude investiční položkou (150 tis. Kč) a bude vybaven m.j. dvěma procesory AMD Athlon MP, výkonným diskovým polem (RAID) a rozhraním Gigabit Ethernet. Celkové náklady na pořízení klastru se tak vyšplhají na cca 570 tis. Kč, tj. na necelou polovinu dvouprocesorové stanice IBM RS/6000 mod. 43P/260, která od roku 1999 sloužila jako hlavní výpočetní prostředek střediska.

3 Programové vybavení pro klastr

Zatímco u technického vybavení jde více méně jen o to pořídit za daný peníz co nejvíce výkonného „železa“, u programového vybavení je třeba vložit podstatně více vlastní práce, zejména půjdeme-li cestou volně dostupného softwaru, což u beowulfů bývá spíše pravidlem (přestože existují zřejmě pohodlnější komerční řešení). Se zvyšujícím se počtem procesorových uzlů rostou totiž nároky na administraci systému, která při desítkách uzlů vyžaduje kvalitativně jiný přístup

než u samostatné stanice. Kritickými oblastmi je instalace a údržba OS a aplikačního softwaru (konfigurace, upgrady), správa uživatelských účtů, organizace (plánování) běhu uživatelských úloh, či monitorování a bezpečnost systému. Na Internetu lze najít množství softwaru, který může pomoci s potřebnou automatizací těchto úkonů, ale je třeba se připravit na větší či menší experimentování, výběrem softwaru počínaje, přes jeho překlad a konfiguraci až po sladění s ostatními komponentami, aby se dosáhlo požadované funkce. Není zde záchytný bod v podobě komerčního dodavatele, který bere zodpovědnost za správné fungování komponent i celého systému. Nutnou podmínkou je proto nadšený systémový administrátor.

Vlastní instalaci programového vybavení předchází rozvaha, jaké úkoly či služby budou plnit *interaktivní uzel*, *souborový server* a *výpočetní uzly*, a posléze, které konkrétní programy jsou schopny toto zajistit. Naše zkušenosti uvádíme v této sekci. Připomeňme, že v první i současné druhé fázi budování klastru plní roli interaktivního uzlu i souborového serveru jeden počítač (říkejme mu jen *server*).

Operační systém Linux jsme pro klastr volili v distribuci Debian 3.0 (Woody) [5], s nímž jsme měli dobré zkušenosti z jiných instalací. Vystačili jsme v podstatě jen s jeho balíčky, s výjimkou konfiguračního nástroje Cfengine [4] a sady překladačů Portland Group [8], na beowulfech velmi rozšířených. Tyto překladače byly prozatím jediným software, jenž jsme pořídili komerčně (licence cca 18 tis. Kč), a to hlavně pro kvalitní kompilaci fortranských kódů. Co se typu souborového systému týče, vsadili jsme na XFS.

3.1 Interaktivní uzel

Interaktivní uzel je uživatelským vstupním bodem ke klastru a zároveň slouží k interaktivní práci. Je zde proto plná instalace OS a aplikačního softwaru. Naopak se nepočítá s tím, že uzel bude zapojován do náročných výpočtů.

Přístup na klastr: Uživatelé mohou pracovat přímo na grafické konzoli interaktivního uzlu nebo se přihlašovat vzdáleně prostřednictvím SSH. Pro přístup z platforem bez X-serveru, např. MS Windows, lze využít VNC. Přístup je omezen TCP wrappery (`/etc/hosts.{allow,deny}`) a paketovým filtrem (iptables).

Vývoj a provozování paralelních úloh: Práce s paralelními aplikacemi je zajištěna plnými instalacemi potřebných produktů, tj. PVM, MPICH, LAM/MPI, PETSc, BLAS, LAPACK, ATLAS apod. Vzhledem k malému počtu navzájem se znajících uživatelů nebylo třeba instalovat systém pro plánování a spouštění dávkových úloh.

Přesný čas: Přesný čas a synchronizace všech uzlů je pro běh paralelních programů velmi důležitá. Přesný čas zajišťuje NTP server; umožňuje ostatním uzlům, jež jsou v nedostupném vnitřním síťovém segmentu, si svůj lokální čas korigovat (pomocí programu ntpdate, t.č. co hodinu). NTP server poběží i na samostaném souborovém serveru.

Další služby (např. pošta a detekce chyb) jsou realizovány stejně jako na výpočetních uzlech — viz 3.3.

3.2 Souborový server

Souborový server má poskytovat především sdílenou diskovou kapacitu. Vzhledem k výkonnosti dvouprocesorového hardwaru zde uvažujeme také o dalších službách, kupř. o druhém přístupovém bodu ke klastru. Rozhodneme se na základě praktických zkušeností.

Služby NFS: Souborový server exportuje na ostatní uzly souborové systémy `/home` (sdílené domovské adresáře uživatelů), a `/tools` (software nevyžadující lokální instalaci). Rychlé sdílení velkého diskového prostoru je pro běh paralelních aplikací velmi praktické. Zabezpečuje ho zvláštní (systémové) síťové rozhraní s gigabitovým spojem k serveru.

Konfigurace pro uzly: Konfigurační soubory, které se v průběhu provozu klastru na jeho uzlech mění, se uchovávají na souborovém serveru. Cfengine je odtud distribuuje na uzly. Např. informace o uživatelských účtech jsou udržovány právě na souborovém serveru, odkud se pod řízením Cfengine kopírováním `/etc/{passwd,shadow}` přenášejí na ostatní uzly. Viz také sekci 3.4.

Pošta: Pro správu elektronické pošty jsme zvolili rychlý a bezpečný Postfix.

3.3 Výpočetní uzly

Tyto uzly jsou vyhrazeny pro běh paralelních nebo výpočetně náročných sekvenčních programů. OS je zde instalován v rozsahu omezeném těmito účely. Uzly nebývají trvale v provozu – uživatelé mají možnost si je podle potřeby na dálku spouštět.

Běh paralelních úloh: Pro vykonávání paralelních úloh při absenci systému pro dávkové zpracování vystačíme s přístupem z interaktivního uzlu na jednotlivé výpočetní uzly, což zajišťují R-služby (rsh, rlogin, rcp) a nastavení `/etc/hosts.equiv`. Instalace sdílených knihoven nutných pro běh paralelních programů je omezena na run-time moduly. Aplikace mají na každém uzlu k dispozici nejen sdílenou diskovou kapacitu souborového serveru (viz tam), připojovanou prostřednictvím autofs, nýbrž také lokální pracovní oblast `/local` o velikosti 10 GB.

Pošta: Zasílání elektronické pošty se realizuje pomocí balíčku SSMTP, který zprávy pouze přeposílá na souborový server. Nevýhoda: nejede-li souborový server, pošta se ztratí. (Nicméně bez souborového serveru by byl klastr stěží použitelný.)

Detekce chyb: Zde i na všech ostatních strojích klastru je použit logcheck, jenž vyhledává neobvyklé záznamy v logovacích souborech a informuje o nich administrátora.

3.4 Administrace

Stručně se ještě zmiňme o administraci klastru. Co se jeho celkové **instalace** týče, lze její hlavní body shrnout takto:

1. Iniciální částečná instalace (výpočetního) uzlu
2. Na kopii této instalace doplnění služeb specifických pro interaktivní uzel a souborový server (t.č. vše na jednom počítači – serveru): přidání vývojového prostředí, serveru NFS a X Window apod.
3. Dokončení instalace a konfigurace jak serveru, tak výpočetního uzlu
4. Vytvoření bootovacího CD pro instalaci výpočetního uzlu (nazvěme ho *instalačním CD*)
5. Replikace dalších výpočetních uzlů z instalačního CD (viz níže)
6. Doladění konfigurací

Údržba systému samozřejmě zahrnuje spoustu úkonů. Zde můžeme nastínit jen některé z nich.

Při **replikaci úkonů** administrátora, např. při hromadné změně určitých konfiguračních souborů, kombinujeme dva přístupy: (1) Pomocí jednoduchých skriptů `dist_exec`, resp. `dist_copy`, využívajících SSH, může root spustit na všech uzlech zadaný příkaz, resp. rozkopírovat zadaný soubor na všechny uzly. (2) Mnohem širší možnosti nabízí mocný „konfigurační stroj“ Cfengine [4]: Při spuštění si Cfengine ze (souborového) serveru zaktualizuje svou konfiguraci a poté se pokusí ji uplatnit na svém uzlu. V našem případě se Cfengine na uzlech spouští při startu systému, každý den ráno a pak explicitně některými skripty, např. při přidávání uživatele.

Jelikož zvolenou distribucí je Debian, lze **instalovat a upgradovat balíčky** (a tedy i programy v nich obsažené) příkazem `apt-get`. Pouze je nutné uzlům zpřístupnit archív s balíčky, což řešíme utilitou squid (http proxy cache). Po jejím spuštění na serveru lze instalovat nebo upgradovat pomocí `dist_exec 'export http_proxy=http://thea00s:3128/; apt-get parametry'`.

Pro **přidání nového výpočetního uzlu**, už připojeného do sítě klastru, ho nabootujeme z instalačního CD (viz výše), na němž je připravena dávka `install`. Té stačí zadat číslo nového uzlu a pár dalších drobností, aby provedla automatickou instalaci.

4 Scénář testování

Jak už bylo zmíněno, aplikace, která je pro SAM rozhodující při hodnocení výkonu počítače, je řešič proprietárního systému matematického modelování GEM32, jehož úkolem je výpočet řešení (vektoru posunutí) soustavy lineárních rovnic se symetrickou, pozitivně definitní *maticí tuhosti*, generovanou na základě konečněprvkové diskretizace matematického modelu. Všechny stávající řešiče v GEM32 implementují metodu *sdružených gradientů s předpodmíněním* (PCG), ale odlišují se v různých aspektech. Pojednání o matematických metodách a algoritmech použitých v programech však vychází za rámec tohoto textu – další informace v tomto směru lze nalézt např. v [1].

4.1 GEM32 – vybrané řešiče

Pro testovací běhy jsme použili následující řešiče:

PCG-S: Sekvenční řešič. Jako PCG-S/FP označujeme uplatnění *pevného předpodmínění* (FP) na bázi neúplné faktorizace, jako PCG-S/VP *proměnné předpodmínění* (VP) využívající vnitřní iterace. Poznamenejme, že VP se v sekvenčním případě ukazuje méně efektivní a v praxi ho nepoužíváme.

PCG-DID: Paralelní řešič na bázi *rozložení posunutí* (displacement decomposition, DiD), vytvářející čtyři paralelní procesy (jeden *řídicí* a tři *výkonné*, pro každý ze směrů posunutí). PCG-DID/FP je s pevným předpodmíněním, PCG-DID/VP s proměnným předpodmíněním. Proměnné předpodmínění VP dosahuje předepsané přesnosti řešení při nižším počtu interací, a tudíž s nižšími komunikačními nároky, ovšem za cenu větší výpočetní práce.

PCG-DD: Paralelní řešič na bázi *rozložení oblasti* (domain decomposition, DD), vytvářející jeden řídicí a obecně n výkonných procesů, jeden pro každou podoblast. Používalo se *aditivní Schwarzovo předpodmínění* a *agregovaná hrubá síť* (řešená samostatným výkonným procesem) pro urychlení konvergence.

K společným rysům uvedených řešičů (jejichž kódy jsou psány ve Fortranu 77) patří, že načítají svou největší datovou strukturu, matici tuhosti, do paměti na začátku výpočtu. Poté jsou jejich diskové požadavky minimální, takže průběh výpočtu (jeho iterační fáze) je do velké míry nezávislý na diskovém subsystému.

Realizace paralelních řešičů spočívá na *modelu předávání zpráv*, v němž spolupracující procesy interagují (předávají si data, synchronizují se apod.) pomocí výměn zpráv. Tento model, nevyžadující sdílenou paměť, je realizovatelný na jakékoliv paralelní architektuře, a to pomocí *systémů předávání zpráv*, z nichž jsou dnes nejpopulárnější PVM (Parallel Virtual Machine) [9] a obzvláště pak různé implementace standardu MPI (Message Passing Interface) [6]. Naše řešiče jsou schopny spolupracovat podle potřeby s oběma těmito knihovnami.

4.2 Porovnávané platformy

Pro určitou objektivizaci výsledků srovnáváme klastr thea s dalšími výpočetními systémy, kde jsme měli možnost provést obdobné testování. Jejich základní technické charakteristiky uvádí tabulka 1.

Tabulka 1. Technické charakteristiky porovnávaných systémů

System	*Manuf.*	*Type*	*Processor/Frequency*	*Memory*	*Commun.*	*OS*
Thea	beowulf	cluster	8×AMD Athlon/1.4	8×768 MB	Fast Eth.	Linux
Beowulf	beowulf	cluster	42×AMD Athlon/1.2	42×1 GB	2×Fast Eth.	Linux
Lomond	SUN	SMP	18×UltraSparcII/400	18 GB	shared mem.	Solaris
Abacus	IBM	SMP	4×POWER3/375	4 GB	shared mem.	AIX
Parmac	beowulf	cl./SMP	4×2×PPC G4/450	4x512 MB	Fast Eth.	Linux

Upřesněme, že abacus, který v době psaní tohoto textu byl stále ještě v nabídce výrobce a který do instalace nového klastru v létě t.r. byl nejvýkonnější

systém na VŠB-TUO, patří do třídy víceprocesorových pracovních stanic. Lomond z EPCC v Edinburghu, také symetrický multiprocesor, jehož výrobce svého času zařadil do střední třídy systémů pro HPC, je nyní již nahrazen výkonnějšími modely. Beowulf je profesionálně spravovaný klastr z amsterdamského HPC centra SARA, který je jakýmsi větším bratrem thea: Má výrazně vyšší počet o něco slabších procesorových uzlů. Parmac, umístěný v CLPP v Sofii, je zvláštní použitím čtyř dvouprocesorových uzlů Apple Power Mac G4. Všechny systémy používaly diskový prostor sdílený přes NFS.

Jsme si vědomi toho, že dále uváděná porovnání mohou být pouze ilustrativní, už z toho důvodu, že nešlo o stejně staré systémy, přičemž právě časový faktor hraje ve výkonnosti výpočetní techniky obrovskou roli.

4.3 Testovací úlohy

Prvním úlohou, označovanou *FOOT*, je model pružného základu, který díky rozumné velikosti mohl byl počítán na celé řadě platforem, a pro něhož tedy máme velké množství výsledků. Jeho pravidelná strukturovaná síť obsahuje $41 \times 41 \times 41$ uzlů, což generuje lineární soustavu o 206 763 rovnicích. Řeší se s relativní přesností 10^{-4}.

Druhou úlohou je *DR*, velký benchmark odvozený z modelu pro posouzení geomechanických vlivů při dobývání uranové rudy. Jeho diskretizace má síť s $124 \times 137 \times 76$ uzly, takže se řeší soustava s 3 873 264 neznámými, opět s relativní přesností 10^{-4}, při nulové počáteční aproximaci.

5 Výsledky testů

5.1 Sekvenční řešič

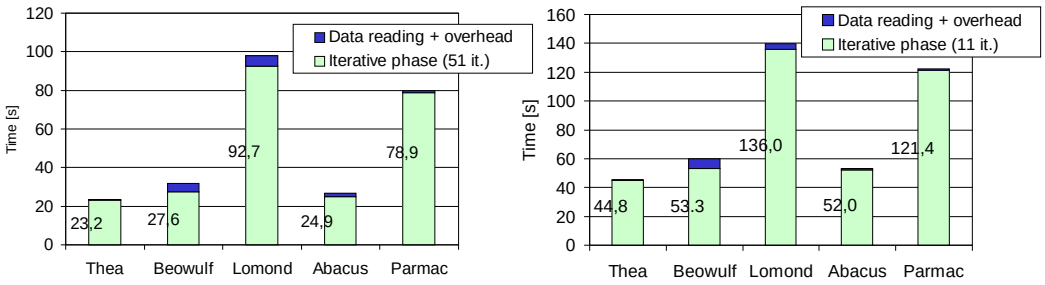
Cílem testu bylo podat představu o výkonnosti uzlů klastru při běžném sekvenčním zpracování. Pro tento účel jsme řešili úlohu FOOT sekvenčním řešičem PCG-S, a to jak pro pevné, tak pro proměnné předpodmínění. Výsledky jsou na obrázku 2. Jednotlivé sloupce jsou rozděleny na čas spotřebovaný pro vlastní iterační fázi výpočtu, kterou považujeme za hlavní měřítko výkonnosti procesoru, a dobu potřebnou k načtení dat a ostatní režii, která je značně ovlivněna diskovým subsystémem, stavem OS apod.

Z grafů plyne, že procesory AMD Athlon jsou velmi výkonné a víc než konkurenceschopné s procesory IBM (POWER3), Sun (UltraSPARCII) a Apple (PowerPC G4), nasazovanými v pracovních stanicích – v případě thea dokonce hrály prim![5]

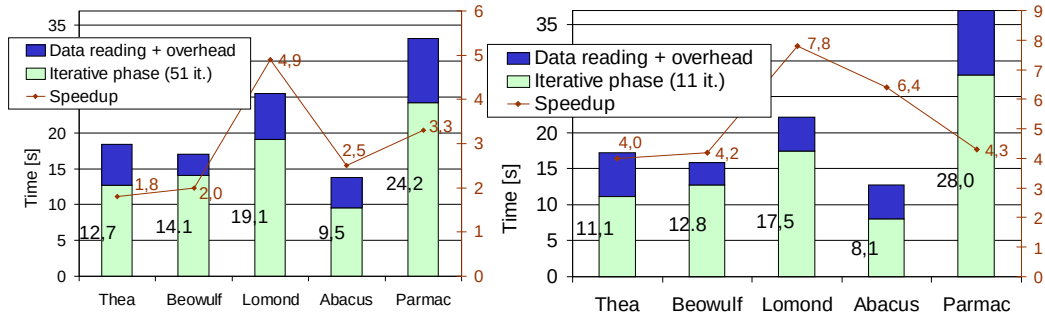
5.2 Řešič s rozložením posunutí

Časy řešiče PCG-DiD, paralelizovaného pomocí rozložení posunutí pro čtyři procesory, jsou na obr. 3. Kromě informací jako výše grafy kvantifikují také dosažené zrychlení v iterační fázi.

[5] Tyto výsledky jsou v dobré shodě s dalšími testy, např. LINPACK [10] nebo SPEC [11].



Obrázek 2. Porovnání sekvenčního výkonu: úloha FOOT řešená PCG-S s pevným (*vlevo*) a proměnným (*vpravo*) předpodmíněním

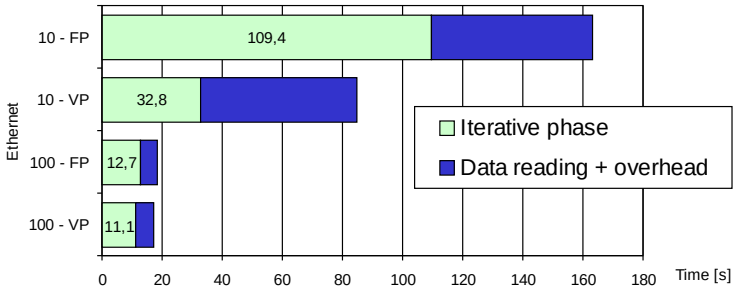


Obrázek 3. Porovnání na FOOT/PCG-DiD: opět pevné (*vlevo*) a proměnné (*vpravo*) předpodmínění a dosažené zrychlení (*lomená čára*)

V tomto testu prokázal své kvality abacus, u něhož velmi rychlá komunikace prostřednictvím sdílené paměti eliminovala případnou výhodu o něco rychlejších procesorů AMD, a který dosáhl absolutně nejkratších časů. Ze stejného důvodu ze výrazně zlepšil lomond. Dosažené zrychlení se počítač od počítače dost výrazně liší od teoreticky maximálního trojnásobku[6] a především u proměnného předpodmínění dosahuje superlineárních hodnot. Předpokládáme, že je to především větší lokalitou dat, umožňující účinnější využití vyrovnávacích pamětí (cache), nemůžeme však vyloučit ani málo efektivní naprogramování sekvenční verze VP. Na grafech lze také vidět podstatně větší podíl neiterační fáze paralelních řešičů na celkovém čase výpočtu.

Je vidět, že výkonnost komunikačního subsystému je pro úlohy tohoto typu klíčová. Můžeme to ještě lépe demonstrovat na obr. 4, který ukazuje zásadní rozdíl v časech výpočtu na thea, realizuje-li se komunikace prostřednictvím nepřepínaného Ethernetu 10 Mbit/s, nebo přepínaného Fast Ethernetu 100 Mbit/s. Upřesníme, že nejnáročnější výměna dat probíhá v PCG-DiD při paralením násobení matice $\times$ vektor, kdy si výkonne procesy v úloze FOOT vyměňují celkově cca 1,6 MB (v každé iteraci). Proto se u klastrů komunikační subsystémy stavějí, pokud to rozpočet dovolí, na bázi pokročilejších technologií, jako jsou Gigabit

[6] Řídicí proces je výrazně méně výpočetně zatížen než tři výkonné procesy.



Obrázek 4. Porovnání na FOOT/PCG-DiD na Ethernetu (*horní dva sloupce*) a přepínaném Fast Ethernetu (*dolní dva sloupce*)

Ethernet nebo ještě lépe např. Myrinet [7], vynikající vedle obrovské šířky pásma (až 2 Gbit/s) také nízkou latencí.

K výsledkům ještě poznamenejme, že je řada dalších faktorů, které mohou ovlivnit čas běhu (paralelního) programu, od výběru překladače a volby parametrů překladu přes použitou paralelizační knihovnu (PVM versus MPI) až např. po využití možností komunikačního subsystému.

5.3 Řešič s rozložením oblasti

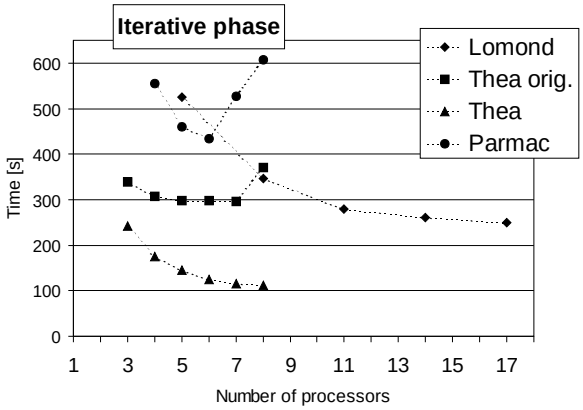
Při testování řešičem PCG-DD, na bázi rozložení oblasti, jsme uplatnili benchmark DR, protože velká úloha by měla umožnit prokázat škálovatelnost tohoto kódu, tzn. schopnost zkracovat čas řešení s rostoucím počtem použitých procesorů. K takovému ověření je zapotřebí systém s větším počtem procesorových uzlů, proto se na obrázku 5 objevují výsledky jen pro thea, lomond a parmac.[7] První běhy ukázaly, že zde thea nenaplňuje očekávání a časy s rostoucím počtem procesorů stagnují (na obr. 5 čtverce). To byl důvod k analýze paralelního kódu a k optimalizaci jeho komunikace, která přinesla zásadní zlepšení škálovatelnosti (trojúhelníky). Dlužno doplnit, že testy na lomond (kosočtverce) nebylo možno zopakovat s optimalizovaným kódem, nicméně při superrychlé komunikaci na tomto symetrickém multiprocesoru (cca desetkrát rychlejší než na thea) lze očekávat jen mírné zlepšení.

Dodejme, že řešičem PCG-DD jsme byli s to na thea zpracovat také zvětšenou verzi DR o 8 371 350 rovnicích. V tomto případě spočíval přínos paralelního zpracování ani ne tak v rychlosti, nýbrž v prosté schopnosti takovou úlohu vůbec spočítat (sekvenčně to na naší technice nepřicházelo v úvahu). Bylo to možné díky faktu, že při paralelním řešení se zpracovávaná data dekomponují na menší bloky, které již byly v kapacitních mezích jednotlivých uzlů.

6 Závěr

Příspěvek si kladl za cíl poukázat na konkrétním příkladě na velký potenciál současných standardních osobních počítačů pro náročné výpočty. Viděli jsme,

[7] Na beowulf jsme bohužel neměli možnost tento test provést.



Obrázek 5. Řešení DR pomocí PCG-DD při rostoucím počtu procesorů na lomond (*kosočtverce*) parmac (*kroužky*) a thea (*trojúhelníky*); *čtverci* je vyznačen výkon starší verze řešiče na thea

že i relativně levné stroje, zapojené do klastru ve více méně amatérských podmínkách, jsou z hlediska výkonu v přiměřených aplikacích s to směle soupeřit s třídou systémů pro HPC, které mohou být i řádově dražší. Pokrok v hardwaru spolu s PVM a MPI, Linuxem a dalším volným, nicméně vysoce efektivním softwarem přinesly HPC doslova na náš stůl. Sestavení klastru a jeho optimalizace může snad vyžadovat větší míru kompetence, nicméně flexibilita těchto systémů a poměr cena/výkon je přímo předurčují pro akademické účely.

Reference

1. Blaheta, R., Jakl, O., Starý, J.: *Parallel high-performance computing in geomechanics with inner/outer iterative procedures*. In: Proc. Computational Science – ICCS 2002 (P. M. A. Sloot et al., eds.). LNCS 2331, Springer-Verlag, Berlin, 2002.
2. Kasprzak, J.: *Clusterová řešení na Linuxu*. In: SLT 2001 – sborník 2. semináře o Linuxu a TEXu, Konvoj, Brno, 2001.
3. *The Beowulf Project*. `http://www.beowulf.org`, 25/10/02.
4. *Cfengine*. `http://www.iu.hio.no/cfengine`, 25/10/02.
5. *The Debian Project*. `http://www.debian.org/`, 25/10/02.
6. *Message Passing Interface Forum*, `http://www.mpi-forum.org/index.html`, 25/10/02.
7. *Myricom, Inc.*, `http://www.myricom.com`, 25/10/02.
8. *The Portland Group Compiler Technology*. `http://www.pgroup.com`, 25/10/02.
9. *Message Passing Interface Forum*, `http://www.epm.ornl.gov/pvm/pvm_home.html`, 25/10/02.
10. Dongarra, J. J.: *Performance of various computers using standard linear equations software (18/01/01)*, `http://www.netlib.org/benchmark/performance.ps`.
11. *The Standard Performance Evaluation Corporation*, `http://www.spec.org/` (25/01/01).

Linux do škol a otevřené systémy na středních školách

Jiří Matyáš, Martin Grombiřík

[1] Gymnázium, Brno, Vídeňská 47
Email: matyas@gvid.cz
[2] Jednota školských informatiků
Email: grombi@cmsps.cz

Abstrakt: Přednáška je zaměřena na ohled za využitím otevřených systému na středních školách. Dále bude předvedena koncepce projektu Linux do škol, který vzniká z iniciativy Jednoty školských informatiků. Cílem je vyvolat v odborné veřejnosti zájem o podporu tohoto projektu a tedy následně i naší budoucí konkurence. A mladé konkurenty takto směřovat k lepší budoucnosti, nezávislých na poplatcích za licence.

Klíčová slova: Linux, LinDoŠ, JŠI, LTSP, Linux Terminal Server, RedHat, ITV

1 Otevřené systémy na středních školách

1.1 Život otevřených systémů na gymnáziu

O opensoftware se zajímá a stará skupina nadšenců, říkající si ITV – Internet team Vídeňská – složená převážně ze studentů školy, kteří zabezpečují chod unixových systémů na škole: servery ENIAC, eXus a firewall.

Dále se zde vyvíjí snahy o propagaci Linuxu a TEXu, o zlepšení gramotnosti absolventů tohoto gymnázia v oblast informatiky a výpočetní techniky, formou různých přednášek (např. „Typografický systém TEX a OS Linux“, „OS Linux a publikace pod doménou Gvid.cz“ nebo „Programovací jazyk C“) a následným zapojením do níže zmíněných projektů. Každý studentík má tedy možnost se dostat až na pozici administrátora serveru. V této době jenom já takto zaškoluji již třetí generaci.

1.2 Projekt ENIAC

Internetový server ENIAC byl původně maturitním projektem Milana Šorma v roce 1997. Tehdy ještě student oboru programování dokázal přesvědčit vedení školy a vybudoval nejen lokální síť, ale v závěru svého studia založil i náš internetový unixový server, který byl jedním z prvních serverů spravující doménu střední školy. Do práce na serveru se postupně zapojila řada studentů naší školy a ENIAC se postupně rozvíjel a rozvíjí.

Jan Kasprzak, Petr Sojka (editoři): SLT 2002 – sborník semináře o Linuxu a TEXu, str. 209–216, 2002.

Lidé, kteří zasáhli do života ENIACu:

- Luděk Finstrle – Původní webmaster, autor mnoha věcí do administrativy i mimo ni.
- Miroslav Křipač – Milanův pokračovatel, správce ENIACu, autor několika nových myšlenek.
- Pavel Najvar – Správce původního intranetového serveru, autor designu nynějších www stránek.
- Petr Dadák – Další pokračovatel a autor vylepšené administrativy.

Historie tohoto serveru. Nyní vám přiblížím co to vlastně ENIAC je, jak je udělaný a co všechno umí. Musíte mi prominout, že některé pasáže nebudu podrobně rozvádět, protože by to mohlo být potenciální bezpečnostní riziko.

ENIAC je běžné stolní PC vybavené procesorem Pentium MMX taktovaným na 200 MHz. Dále obsahuje 64 MB paměti, standardní PCI a ISA sběrnici, nepodstatnou grafickou kartu, sériová a paralelní rozhraní, 40 GB HDD a šestnáctibitovou síťovou kartu s rozhraním pro kroucenou dvojlinku.

Na ENIACu je nainstalován operační systém Linux, v současné době s jádrem verze 2.4.X. Z tradice je nainstalována distribuce RedHat 7.2 doplněná o nutné upgrady a další potřebný software.

ENIAC poskytuje svým uživatelům mj. tyto služby:

- Email – SMTP, pop3, SSL tunelovaný pop3 server, poštovní klienti např. mutt, jednoduché webové rozhraní v administrativě.
- WWW stránky – možnost tvorby vlastních webových stránek s využitím php4, mysql, CGI.
- MySQL – přístup do databáze MySQL.
- Unixový shell – přístup přes SSH.
- Scriptovací jazyky např. Perl.
- Překladč C/C++, Pascal.
- Sázecí systém TEX spolu s makry pro cslatex a plaintex.
- K tomu všemu potřebný diskový prostor.

1.3 Projekt eXus

Tak tedy je to intranetový server Gymnázia, Brno, Vídeňská 47, obsluhující pracovní stanice, s nainstalovanou linuxovou distribucí Red Hat 7.2 CZ (Enigma). Záměrem je seznámit studenty s jinými OS, než jen s Windows 9x, který je znám svojí nespolehlivostí. Studenti mají možnost mít svůj osobní účet a práva. Z Linuxu mohou „brouzdat“ po internetu (zde trochu těžkopádnější Mozillou, Linksem, Operou, Netscapem, Lynxem), učit se některým programovacím jazykům (C, Pascal, asambler, lisp, elisp, ...) či snáze ovladatelnému sazecímu systému TEX. To jsou prográmky (jazyky), které nejsou vždy přístupné v prostředí Win 9x. A to nejen z technických a finačních důvodů. Pod Linuxem je 99 % implementací programovacích jazyků a různorodých programů zdarma.

Dnes je pravděpodobně zprovozněno devět stanic v učebně „pravé“ a sedm v „nové“. Stanice jsou nazývány „ivory X“, a to po dlouhé a náročné diskusi mezi administrátory, kde název je v překladu slovo „slonovina“ a X je číslo dané řazením dle schematu učebny, a to pouze v pravé učebně akademické části gymnázia. Ke konci školního roku 2000/2001 jsme rozběhli projekt GRASTA (GRAfická STAnice). Nejedná se o to, že bychom zde sestavovali nějaké „superpočítače“ (i když by to bylo v našich silách...), ale ty finance... Prostě a jednoduše jsme začali zavádět OS Linux do „nové“ učebny.

Proč nazýváme tento server eXusem. „eXus“ je pokračovatelem serverů, které se v historii informatiky na našem gymnáziu významně zapsaly. Tedy „Eniac“ a „Elis“. Odtuď tedy to prví písmeno *E*. Písmeno *X*, ukazuje na možnost pracovat v grafickém prostředí X-Window. Takřka každý systém je směřován *U*živatelům. A *S* prostě říká, že se jedná o server.

2 LINUX do škol – Lindoš

2.1 Úvod

Projekt Jednoty školských informatiků Linux do škol (dále jen „projekt“) si klade za cíl navrhnout a prokázat realizovatelnost alternativního řešení počítačové učebny na bázi operačního systému Linux, nástrojů z projektu GNU a na ně navazujících projektů typu Open Source. Vychází z faktu, že v rámci současného projektu SIPVZ jsou otevřené platformy (v rozporu se Zadávací dokumentací k Projektu III SIPVZ) generálním dodavatelem technologie záměrně opomíjeny. Přitom podle zahraničních zkušeností je nasazení těchto systémů v rámci veřejné vzdělávací soustavy a státní správy výhodné (např. z hlediska pedagogického, technicko-administrátorského a ekonomického).

Ilustrace 1 Hodina matematiky. „Dnes se naučíme sčítat. Zapněte si své standardizované kalkulačky. Sčítáme takto: napíšeme první sčítanec, stiskneme zelené tlačítko [+] v pravém horním rohu, napíšeme druhý sčítanec a stiskneme šedé tlačítko [=] vlevo dole.“ Doufáme, že takto výuka matematiky neprobíhá. Výuka výpočetní techniky však často ano. Je to totiž nejpohodlnější a nejrychlejší způsob. Vytváří uživatele závislé na jedné konkrétní aplikaci, kteří nerozumí podstatě řešení problému.

Ilustrace 2 Náklady na komerční software v projektu SIPVZ činí podle informací zveřejněných na Internetu Kč 380,– na jeden počítač za jeden měsíc. Tj. za rok přes Kč 4 000,–. Podle odhadů se nachází v českém školství cca 100 000 počítačů. Dá se tedy odhadnout, že při používání komerčních programů bude stát vydávat pouze za licence na jejich používání (bez jakékoliv podpory) 400 miliónů Kč ročně. Přitom se dá očekávat nárůst počtu počítačů na minimálně dvojnásobek, protože současný stav zdaleka neodpovídá ideálu, podle kterého by měla mít většina učitelů k dispozici svůj osobní počítač a ve škole má připadat pět až sedm žáků na jedno počítačové pracoviště. Navíc ceny za používání komerčních programů stále rostou. Pokud nedojde k zásadním změnám, je možné, že stát

v roce 2006 zaplatí za licence na využívání komerčních programů ve školství až 1 miliardu Kč.

Námi navrhovaný projekt řeší oba dva zásadní problémy. Vede uživatele výpočetní techniky ke zvládnutí zásad práce s určitou skupinou programů (a tedy ke schopnosti používat v praxi libovolný program určitého druhu), ukazuje výhodnost používání otevřených a standardizovaných formátů dat a současně výrazně snižuje finanční náročnost na programové vybavení.

2.2 Cíle projektu

Projekt zahrnuje:

- Navržení a vytvoření vzorové počítačové učebny na bázi prostředí GNU/Linux pro výuku základních dovedností informační gramotnosti a zpřístupnění informačních a komunikačních zdrojů v lokálním měřítku i v rámci sítě Internet,
- nasazení prototypu(-ů) do testovacího provozu v modelových lokalitách s průběžným a závěrečným hodnocením,
- vypracování metodiky širšího nasazení řešení (v případě úspěšnosti projektu a pozitivních výsledků testovacího provozu),
- vytvoření manuálů pro různé úrovně realizace této metodiky (uživatelské, školitelské, administrátorské, instalační, . . .),
- vytvoření výukových materiálů pro výuku vybraných programů, akcentujících obecné zásady práce určité skupiny programů (textové editory, tabulkové procesory, procházení a tvorba WWW stránek atd.),
- vytvoření organizačního zázemí pro instalaci, správu a školení programů zahrnutých do projektu s využitím potenciálu vysokých škol (případně vyšších odborných i středních škol) a soukromých firem.

Projekt předpokládá využití zkušeností ze zahraničí a jejich adaptaci pro specifika českého prostředí.

2.2.1 Předpokládané přínosy projektu pro praktické nasazení

1. *Z hlediska výuky VT ve školách:*
   Studenti pochopí zásady ovládání určité skupiny programů a naučí se pracovat s více programy zastupujícími určitou skupinu aplikací.
   Práce s otevřenými formáty a otevřenými standardy bude studenty učit chápat standard jako dohodnutou transparentní konvenci pro snadnější výměnu informací mezi různými typy platforem než jako komerčním monopolem vnucený diktát majority.
   Nadaní žáci a studenti se mohou (prostřednictvím filosofie otevřených zdrojů) seznamovat se špičkovými technologiemi přímo v praktickém provozu a eventuálně se (podle svých schopností) podílet na „vylepšování“ systému na různých úrovních.

2. *Ekonomická výhodnost projektu*
   Jak z hlediska vstupních nákladů, tak z hlediska TCO (celkových nákladů na vlastnictví) se odbourání licenčních poplatků projeví výraznými finančními úsporami. Školy, které budou disponovat odborníky schopnými instalovat a spravovat SW, budou mít externí náklady na systém velmi nízké (nosiče, manuály, školení), ostatní školy budou platit instalaci a správu systému školám dočasně vyspělejším, nikoliv komerční firmě sledující hlavně svůj zisk.
   Z administrátorského hlediska projekt nabízí stabilní prostředí a při zvolení terminálového řešení též výraznou redukci požadavků na údržbu systému.
   Jedná se vlastně o návrat k původní koncepci SIPVZ: otevřený systém, podporující vlastní zdroje a schopnosti školství. Komerční firma, jak je názorně vidět v projektu SIPVZ realizovaném prostřednictvím generálního dodavatele, spotřebuje až 50 % prostředků na správu systému, vlastní režii a zisk. Náš projekt předpokládá, že peníze půjdou do škol, které si díky tomu budou moci dovolit platit skutečně odborně vzdělané správce a učitele ICT. Jejich přítomnost ve škole je katalysátorem rozvoje informatiky, nikoli „dokonalý“ uzavřený systém GD.

2.3 Fáze řešení projektu

2.3.1 Příprava software Vytvoření či adaptace distribuce systému Linux, odladění a otestování této distribuce na typových hardwarových konfiguracích.

Požadavky na distribuci: Snadná instalace a údržba (balíčkové systémy, upgrade s kontrolou závislostí, jednoduché a ergonomické konfigurační nástroje, transparentní struktura konfiguračních souborů), maximální lokalizace do českého jazyka možnost integrace terminálového řešení (mělo by být povinnou součástí dodávky systému), kompatibilita se standardy, důraz na bezpečnost (zvláště u serverové části systému), výběr aplikací reprezentujících určitou oblast práce s počítačem a sjednocení jejich uživatelského rozhraní. Vytvoření potřebných šablon, předloh, vzorů a průvodců včetně grafické stránky systému.

2.3.2 Tvorba dokumentace Vytvoření uživatelských manuálů k základnímu ovládání systému, k jednomu či dvěma grafickým prostředím a k vybraným aplikacím.

Vytvoření metodického manuálu pro školitele uživatelů (specifika systému, „migrační program“, . . .)

Vytvoření administrátorské příručky pro správu systému a vybraných síťových služeb.

Vytvoření instalační příručky pro správce a realizační firmy.

2.3.3 Testovací provoz Pokusné nasazení odladěného systému v reprezentativním vzorku škol (školy různého typu: ZŠ, střední odborné učiliště, střední odborná škola, gymnázium, VOŠ).

Proškolení obsluhy (správců VT a uživatelů z řad učitelů i žáků). Metodika sledování úspěšnosti experimentální penetrace systému po dané období.

2.3.4 Propagace akce v médiích Primárně na vlastním serveru projektu se zázemím pro podporu projektu (souborový archiv pro distribuci, dokumentaci a update, nejlépe napojený na systém sledování změn, diskusní fóra, informační portál) a na serveru Jednoty školských informatiků, která projekt zaštiťuje. Dále na příslušných IT webech.

Publikování informací o systému v odborném tisku (počítačovém i pedagogickém), na konferencích a veletrzích.

Průběžná informovanost ředitelů škol, správců VT, informatiků a učitelů o projektu prostřednictvím brožur, letáků a inzerce.

2.4 Rozsah funkčnosti systému – desktopové služby

2.4.1 Standardní aplikace GUI s možností volby správce oken a prostředí (jeden bude implicitní a podporovaný), balík kancelářských aplikací (textový procesor, tabulkový kalkulátor, prezentační program).

2.4.2 Sítě a Internet Prohlížeč(e) webu, poštovní klient(i) s možností správy kontaktů, klienti různých komunikačních služeb (ICQ, IRC, Jabber apod.).

2.4.3 Grafika a multimedia Editor rastrových a vektorových obrázků, prohlížeč a konvertor různých grafických formátů, podpora různých zvukových a video formátů (přehrávání, konverze, v rámci možností i editace záznamů).

2.4.4 Vývoj a programování Alespoň 1 RAD nebo IDE nástroj, programátorské editory, překladače běžných jazyků (C/C++, Pascal), ladící utility.

Nástroj na výuku strukturovaného programování pro začátečníky typu robot Karel.

2.4.5 Různé (volitelně) Různé aplikace pro specializované předměty (CAD, GIS, modelovací programy pro matematiku, fyziku, chemii atd.), slovníky a programy pro podporu výuky jazyků, atd. dle poptávky a nabídky.

2.5 Rozsah funkčnosti systému – serverové služby

2.5.1 Souborové a tiskové Možnost sdílení souborů v rámci systému i s využitím externích zdrojů (sítě s jinými platformami – Windows, Novell; Internet – FTP, SCP). Tisk na sdílených tiskárnách.

Domácí adresáře pro každého uživatele s individuálním nastavením programů a pro privátní data.

2.5.2 Terminálové Možnost vzdáleného přihlášení do systému a vzdáleného spouštění programů v textovém (ssh, telnet) i grafickém (systém X-Window, VNC aj.) prostředí, poskytování aplikačních služeb pro slabší hardware konvertovaný na tenkého klienta.

2.5.3 Webové, poštovní a komunikační Webový server s podporou min. 1 systému generování dynamických stránek, možnost tvorby a vystavování osobních stránek uživateli.

Poštovní systém s možností vytvoření privátní schránky pro každého uživatele, POP a IMAP přístup.

Server side podpora komunikace mezi jednotlivými uživateli i směrem ven (IM a chat daemoni, brány k některým službám), 1 redakční nebo groupwarový systém.

2.5.4 Proxy a bezpečnost Firewallové služby, podpora autentizace z více zdrojů a schémat, možnost ochrany komunikace silným šifrováním (SSL, SSH, PGP), monitoring sítě, řízení přístupu k prostředkům i k externím zdrojům (Internet).

2.5.5 Další možnosti Protože většina programů využívaných v tomto projektu (nástrojů z projektu GNU a na ně navazujících projektů typu Open Source) na platformě operačního sytému Linux existuje (nebo se dá překompilovat) pro běh pod operačními systémy Windows, bude možné připravit jejich odpovídající verze tak, aby mohly být současně používány na stanicích s Linuxem i s Windows. Tím budou dále využity investice do jejich přípravy a podpory.

2.6 Zahraniční zkušenosti a jejich příklady

„Linux do škol“ nehodlá stavět na „zelené louce“ neprobádaného terénu, naopak hodlá využít zkušeností a výsledků obdobných již existujících projektů v zahraničí, ať už ve vyspělých evropských zemích nebo v zemích tzv. rozvojových. Tyto projekty je možno rozdělit na 3 základní skupiny:

1. Projekty vlastního řešení *infrastruktury*. Na platformě Linux je velmi často používáno řešení založené na bázi linuxového terminálového serveru (*Linux Terminal Server Project*, `http://www.ltsp.org/` – centrální aplikační server a terminálovi klienti). Osvědčené principy klasického unixového řešení, úspora hardware, správa celého systému z jediného místa, do určité míry samoúdržný systém (bezdiskoví klienti).
2. Projekty tvorby původního *výukového software*. Samostatné izolované produkty nebo integrovaná řešení (např. specializované distribuce systému Linux pro výukové účely). Často EDU podprojekty významných Open Source nebo Free Software projektů (Debian, KDE, SEUL, ...)
3. *Podpůrné projekty* – zastřešující a nadační organizace, propagace, informační portály.

Nad- a mezinárodní projekty, USA

- `http://www.k12os.org/`, `http://www.k12ltsp.org/`, `http://www.k12linux.org/`
  – integrovaný projekt informačního portálu a komplexní linuxové distribuce pro školství (K-12-LTSP). Případové studie linuxových řešení jsou k vidění na `http://www.k12ltsp.org/casestudy.html`.
- *SchoolForge* (`http://schoolforge.net/`) – nejvýznamější podpůrná organizace nadačního charakteru. Zastřešuje většinu projektů zmíněných v tomto dokumentu.
- *Debian Jr.* (`http://www.debian.org/devel/debian-jr/`), *DebianEdu* (`http://wiki.debian.net/DebianEdu/`) – úpravy linuxové distribuce Debian pro děti a pro školství.
- *SEUL/Edu* (`http://www.seul.org/edu/`) – EDU podprojekt projektu *Simple End User Linux*. Informační a konferenční server (pravidelný bulletin *Linux In Education*). Případové studie linuxových řešení.
- *KDE/Edu* (`http://edu.kde.org/projects/`) – výukové aplikace pro desktopové prostředí KDE.

Evropské projekty

- **Německo:** *Freie Software und Bildung* (`http://fsub.schule.de/`). Zastřešující organizace pro různé aktivity. Výběrový seznam německých škol používajících Linux je na `http://fsub.schule.de/linux/schulen.html`. LTSP v rámci projektu *BerliOS* (`http://termserv.berlios.de/`).
- **Francie:** *OFSET* (`http://www.ofset.org/`) – pův. francouzský projekt, výukové programy pro různé předměty.
- **Norsko:** *SkoleLinux* (`http://develper.skolelinux.no/projectinfo.html.en`), *Linux i skole* (`http://www.linuxiskole.no/`). Projekty zavádění Linuxu do norských škol.
- **Dánsko:** *GNUSkole* (`http://www.gnuskole.dk/`)
- **Švýcarsko:** *Arbeitsgruppe Linux an Schulen* (ALIS, `http://www.edux.ch/`). Pracovní skupina při Švýcarském sdružení uživatelů Linuxu.

2.7 Závěrem k LinDoŠovi

Tento projekt zastřešuje Jednota školských informatiků (`http://www.jsi.cz/`), odborná skupina při ACM CZ. Rozvoj projektu je stavěn na základě zkušeností Martina Grombiříka (učitele ICT a správce sítě na Cyrilometodějské střední pedagogické škole a gymnáziu v Brně; šéfredaktora serveru Anti-Indoš), Pavla Roubala (učitele ICT na gymnáziu v Pácově a autora řady pedagogických a metodických materiálů), Ondřeje Ruska (vyučujícího a správce sítě na Gymnáziu Boženy Němcové v Hradci Králové), Jiřího Matyáše (studenta a koordinátora projektů za ITV na Gymnáziu, Brno, Vídeňská 47; člena výboru CSTUG) a dalších. Prozatímní emailový kontakt je `lindos@cmsps.cz`.

Ruby: jen další skriptovací jazyk?

Dalibor Šrámek

Email: dalibor.sramek@insula.cz

Abstrakt: Ruby je moderní, interpretovaný a od základu objektově orientovaný jazyk. Návrh Ruby byl inspirován tím nejlepším ze Smalltalku, Perlu, Lispu a Pythonu. Mezi jeho přednosti patří jednoduchá a elegantní syntaxe, přehledná objektová standardní knihovna, přenositelnost a snadná rozšiřitelnost včetně integrace komponent z jiných jazyků. Interpret Ruby je vyvíjen pod open source licencí.

Klíčová slova: Ruby, programovací jazyk, OOP, skript

1 Úvod

V roce 1993 se Yukihiro Matsumoto rozhodl vytvořit jazyk, ve kterém by programoval po zbytek života. Výsledkem jeho dvouletého snažení byla první verze interpretu jazyka Ruby, která si v Japonsku brzy získala značné množství příznivců. Zhruba od roku 2000 začal narůstat počet uživatelů také ve Spojených státech a Evropě.

U zrodu Ruby stála snaha spojit pohotovost a flexibilitu skriptovacího jazyka s elegancí a silou objektově orientovaného programování (OOP).

Jazyků, které obvykle označujeme jako skriptovací, je v současnosti k dispozici celá řada. Z nejznámějších uveďme dialekty UNIXových shellů, Perl, PHP nebo Python. Jejich popularita spočívá v několika základních vlastnostech:

- úsporný zápis (vysokoúrovňové funkce, možnost jednořádkových „programů“, podpora regulárních výrazů),
- snadné propojení s OS a dalšími utilitami (spouštění externích programů, přesměrování vstupu a výstupu, použití systémových volání),
- možnost integrace modulů v jiných jazycích do funkčního celku („glue language“).

Vzhledem k době svého vzniku však většina skriptovacích jazyků byla navrhována čistě procedurálně a teprve později (pokud vůbec) byly obohacovány o objektové prvky. Jaká je motivace pro rozšiřování jazyků o charakteristické vlastnosti OOP?

Díky vyššímu výkonu hardwaru začalo být v průběhu času schůdné využívání typicky interpretovaných skriptovacích jazyků k rapidnímu vývoji větších projektů. Právě při tvorbě rozsáhlejších aplikací se uplatní přednosti OOP:

Jan Kasprzak, Petr Sojka (editoři): SLT 2002 – sborník semináře o Linuxu a TEXu, str. 217–231, 2002.

- přirozená datová abstrakce a modelování
- zapouzdření (dekompozice programů, uplatnění principu rozděl a panuj, vyloučení globálních proměnných, zmenšení a oddělení jmenných prostorů)
- dědičnost (opakovatelná použitelnost kódu)
- vícetvarost (kratší a univerzálnější kód)

Ruby je od základu navržen jako objektově orientovaný jazyk. Všechny prvky jazyka jsou samy o sobě objekty. Každá funkce je metodou nějaké třídy. Kořenem objektového stromu je třída `Object`. Každá třída je instancí třídy `Class`, která je ovšem opět potomkem třídy `Object`.

Abychom doplnili výčet vlastností, který z Ruby tvoří moderní a silný vývojový nástroj, zmiňme alespoň:

- dynamické typování (typy nejsou přiřazeny proměnným, ale až jejich hodnotám)
- podporu výjimek
- garbage collector (založený na algoritmu mark and sweep)
- rozsáhlou objektovou standardní knihovnu
- definované bezpečnostní úrovně interpretu (ve vyšších úrovních bezpečnosti je zabráněno například spouštění kódu z uživatelem zadaného řetězce nebo volání některých systémových funkcí)

V současné době je Ruby čistě interpretovaný jazyk. Fungující verze interpretu existují pro většinu běžných UNIXových platforem (Linux, *BSD, Solaris), DOS, MS Windows (95-XP), MacOS X, OS/2 a BeOS. Existuje také implementace interpretu v Javě, která rozšiřuje použití Ruby na všechna prostředí, kde lze spustit JVM, a navíc podporuje integraci obou jazyků. Obě implementace jsou vyvíjeny jako open source.

V budoucnosti se uvažuje o vytvoření kompilátoru Ruby do nativního kódu nebo do bytekódu některého z virtuálních strojů – konkrétně se jedná o VM Parrot[1], který je plánovanou cílovou platformou pro Perl 6, a Microsoft CLI, což je virtuální stroj v jádru technologie .NET. Existuje také projekt nazvaný rb2c[2], jehož cílem je vytvořit překladač z Ruby do jazyka C.

Efektivní kompilaci do nativního kódu brání zejména značná dynamičnost jazyka. Příkladem je triviální konstrukce `(1 + 2)`. Tento výraz může například kompilátor jazyka C zredukovat na konstantu `3`. V Ruby je však operátor `+` ve skutečnosti voláním metody objektu `1`. Tato metoda může být během provádění programu kdykoliv předefinována a výraz se tedy musí vyhodnotit skutečně až v okamžiku, kdy je požadována jeho okamžitá hodnota.

2 Zprovoznění interpretu

2.1 UNIX

Pro mnoho distribucí Linuxu a pro BSD systémy existují binární balíky. Pokud preferujeme kompilaci ze zdrojového kódu (nebo balík pro danou platformu

[1] `http://www.parrotcode.org/`

[2] `http://easter.kuee.kyoto-u.ac.jp/~hiwada/ruby/rb2c/`

neexistuje), získáme aktuální verzi z `ftp://ftp.ruby-lang.org/pub/ruby/`. Pro číslování verzí interpretu se používá schéma `m.n.d`, kde `n` je sudé číslo pro stabilní verzi a liché pro aktuální vývojovou větev.

Kompilace obvykle bez potíží probíhá podle standardního schématu (příklad pro stabilní verzi 1.6.7):

```
gunzip ruby-1.6.7.tar.gz
tar xvf ruby-1.6.7.tar
cd ruby-1.6.7/
./configure
make
make test
make install
```

2.2 MS Windows

V případě MS Windows je zřejmě nejsnažší použít kompletní balík obsahující interpret a nejčastěji používané knihovny včetně vazby na API Win32. Archiv je k dispozici na stránce[3]. Průběh instalace odpovídá standardu MS Windows.

2.3 Spouštění interpretu

Interpret se spouští příkazem `ruby file.rb`, kde `file` je jméno souboru se zdrojovým kódem (jako přípona se obvykle používá `rb`). Pro jednořádkové programy lze použít formu `ruby -e "code"`, kde `code` je přímo zdrojový kód. Verzi interpretu zjistíme pomocí přepínače `ruby -v`, rychlou nápovědu pomocí `ruby -h`.

Příkazem `irb` se spouští skript, který umožňuje pracovat s interpretem v interaktivnim režimu podobném práci se shellem. Irb je vhodná pomůcka pro rychlé testování myšlenek a programových konstrukcí.

2.4 Podpora programování

Pro Ruby zatím neexistuje komplexní vývojové prostřed (IDE). Jedním z projektů, které si kladou za cíl specializované IDE vyvinout je FreeRIDE (`http://www.rubyide.org/`). Objevují se také snahy vytvořit pluggin pro některé z univerzálních vývojových prostředí (Eclipse).

V současnosti je k dispozici podpora pro některé (zejména v UNIXu rozšířené) editory. Jedná se o:

- Emacs – modul s podporou je v distribuci interpretu Ruby
- Vim – od verze 5.7 obsahuje podporu standardní instalace
- Jedit – obsahuje podporu ve standardní instalaci
- Jed – podpora je k dispozici na `http://www.kondara.org/~g/slang/ruby.sl`

[3] `http://www.pragmaticprogrammer.com/ruby/downloads/ruby-install.html`

– Nedit – podpora je k dispozici na
`ftp://ftp.talc.fr/pub/ruby/ruby.nedit-0.1.tar.gz`

Pomocí knihoven je možné spustit interpret Ruby ve zvláštním režimu jako debugger popřípadě profiler. Příkazem `ruby -r debug file.rb` se spouští skript v debuggeru s rozhraním podobným `gdb`. Analogicky `ruby -r profile file.rb` interpretuje skript a na závěr zobrazí informace o času spotřebovaném jednotlivými metodami.

2.5 Dokumentace

Dokumentace je k dispozici zejména v japonském a anglickém jazyce. Na stránce `http://www.ruby-lang.org/en/doc.html` je dostupný referenční manuál Ruby, několik tutoriálů a rozsáhlá, volně šiřitelná kniha Programming Ruby. Vše je ve formátech HTML a PDF.

Jako rychlou referenci lze využít aplikaci `ri`, která se spouští z příkazové řádky a zobrazuje stručnou nápovědu podle zadaného parametru. Aplikaci lze získat na `http://www.pragmaticprogrammer.com/ruby/downloads/ri.html`.

Ve výše uvedeném balíku pro MS Windows je zahrnuta kniha Programming Ruby ve standardním formátu nápovědy Windows.

3 Příklad: migrace kódu

Za jeden z hlavních cílů při rozvoji jazyka považuje autor Ruby dodržování principu nejmenšího překvapení (Principle Of Least Surprise). Toto filozofické východisko se v praxi projevuje mnoha způsoby.

– Zkušenější programátor prakticky nemusí používat dokumentaci, protože kód funguje tak, jak očekává. K tomu přispívá i množství definovaných aliasů. Ke zjištění velikosti řetězce nebo pole lze stejně tak použít metodu nazvanou `length` jako metodu nazvanou `size`.
– Programovací jazyk není bariérou mezi myšlenkami a jejich realizací (není třeba „ohýbat“ myšlení podle konstrukcí jazyka. Kód Ruby se často až překvapivě podobá konceptuálnímu popisu algoritmu v běžném jazyce.
– Ruby využívá osvědčené konstrukce a idiomy z jiných jazyků.

Důsledkem principu nejmenšího překvapení je snadná adopce Ruby programátory, kteří již umí některý z rozšířených jazyků (zejména C, Java, Perl, PHP). Jako jednoduchý příklad uveďme následující kód v jazyce C, jehož účelem je vypsat nejmenší číslo z předdefinovaného pole.

```
#include <stdio.h>

int main() {
  int a[] = {6, 8, 2, 3, 1, 9, 5, 4};
  int m = a[0];
```

```
  int i = 1;
  while (i < 8) {
    if (a[i] < m) m = a[i];
    i += 1;
  }
  printf("Min = %d\n", m);
  return(0);
}
```

Pokud chceme kód rychle převést do podoby stravitelné interpretem Ruby, postačí relativně drobné úpravy.

```
a = [6, 8, 2, 3, 1, 9, 5, 4];
m = a[0];
i = 1;
while (i < 8)
  if (a[i] < m)
    m = a[i];
  end
  i += 1;
end
printf("Min = %d\n", m);
```

Vidíme, že zmizely deklarace, které Ruby nepožaduje (a nepodporuje). Mírně se změnila podoba příkazů `if` a `while` a definice hodnot pole. Jinak zůstal program téměř beze změn. V Ruby však můžeme zápis dále zjednodušit.

```
a = [6, 8, 2, 3, 1, 9, 5, 4]
m, i = a[0], 1
while i < 8
  m = a[i] if a[i] < m
  i += 1
end
puts "Min = #{m}"
```

Tam, kde jako oddělovač příkazů slouží nový řádek, není nutné zadávat středníky. Závorky u příkazů `if` a `while` a u parametrů funkcí jsou také nepovinné, pokud je zápis jednoznačný. Povšimněme si obráceného zápisu příkazu `if`, který je pro krátké podmínky čitelnější. Notace `#{expr}` v řetězci ohraničeném uvozovkami znamená, že do řetězce se vloží hodnota výrazu `expr`.

Přesvědčili jsme se, že Ruby šetří programátorovi práci, ale kód stále zůstává dosti složitý. Ve skutečnosti by program v Ruby mohl mít jen jeden řádek.

```
puts "Min = #{[6, 8, 2, 3, 1, 9, 5, 4].min}"
```

Vestavěná třída `Array` reprezentující pole má definovánou metodu `min`, která zajišťuje právě námi požadovanou funkci. Mohlo by se zdát, že uvedený

jednořádkový program odporuje definici Ruby jako čistě objektového jazyka. Pokud však metody nemají specifikován příslušný objekt (příjemce), volají se automaticky metody třídy `Object`. Implicitní rozsah platnosti lokálních proměnných, které nejsou definovány v bloku, je také třída `Object`.

4 Data

4.1 Druhy proměnných

Z hlediska rozsahu platnosti rozlišuje Ruby čtyři druhy proměnných, které se navzájem odlišují formátem jména:

- Globální proměnné – jméno začíná znakem `$`, např. `$age`
- Lokální proměnné – jméno začíná malým písmenem, např. `age`
- Proměnné tříd – jméno začíná znaky `@@`, např. `@@age`
- Proměnné instancí – jméno začíná znakem `@`, např. `@age`

Kromě proměnných je samozřejmě možné definovat i konstanty. Jejich jméno začíná velkým písmenem – např. `AGE`.

V Ruby jsou všechny typy proměnných (včetně nejjednodušších – integer, float) ukládány jako objekty příslušných tříd.

4.2 Čísla

Vestavěné třídy pro ukládání různých oborů čísel jsou potomky třídy `Numeric`:

- Integer – pro celočíselné hodnoty
- Float – pro reálná čísla (přesnost odpovídá přesnosti double dané platformy)
- Complex – komplexní čísla uchovávaná jako dvojice reálných čísel
- Rational – racionální čísla uchovávaná jako celočíslená dvojice čitatele a jmenovatele

Celá čísla se ukládají jako instance třídy `Fixnum` nebo `Bignum`. Zatímco `Fixnum` odpovídá klasickému integeru (32 nebo 64 bitů podle platformy), `Bignum` slouží k ukládání prakticky libovolně velkých čísel (omezených velikostí paměti). Konverzi mezi těmito dvěma typy provádí interpret podle potřeby.

4.3 Řetězce

Řetězcové proměnné jsou uchovávány jako objekty třídy `String`. Fyzicky je obsah řetězce uložen jako sekvence osmibitových znaků. V objektu `String` lze proto uchovávat i binární data (načtená ze souboru apod.). Obvykle se řetězce inicializují pomocí konstant uzavřených v uvozovkách nebo apostrofech. Délka řetězce se může měnit, přičemž automaticky dochází k úpravě množství alokované paměti.

Speciální funkci mají řetězce uzavřené v obrácených apostrofech. Hodnota pak odpovídá výstupu příkazu obsaženého v řetězci.

```
s = "date"   # s = "date"
s = ‘date‘   # s = "Wed Oct 23 15:03:12 CEST 2002"
```

4.4 Regulární výrazy

Regulární výrazy jsou reprezentovány objekty třídy `Regexp`. Práce s regulárními výrazy v Ruby odpovídá úrovni jazyka Perl. Prohledávat řetězce můžeme například jednoduše pomocí operátoru `=~`.

```
"Ahoj" =~ /h.j/  # 1 - byla nalezena shoda počínaje znakem
                 # s indexem 1
"Hola" =~ /h.j/  # nil - nebyla nalezena shoda
```

4.5 Pole

K uložení jednorozměrného pole slouží objekt třídky `Array`. Prvkem pole může být libovolný další objekt včetně pole samotného, čímž lze docílit vícerozměrných polí. `Array` poskytuje automatickou alokaci paměti a kontrolu mezí indexů. Index začíná nulou.

```
a = []      # prázdné pole
a[3] = 1    # pole se automaticky zvětší
a[5] = 'f'  # prvky pole mohou být instance různých tříd
puts a      # obsah pole je: nil, nil, nil, 1, nil 'f'
```

4.6 Interval

Objekt třídy `Range` je definován dvěma krajními prvky a reprezentuje interval nebo sekvenci hodnot. Definice intervalu má dva možné tvary:

```
1..3      # 1, 2, 3 (včetně posledního prvku)
'a'...'d' # 'a', 'b', 'c' (bez posledního prvku)
```

Krajním prvkem intervalu může být objekt třídy, která definuje metodu `succ` s významem „následovník“.

4.7 Hash

Hash je speciálním případem pole, kdy index může být téměř libovolného datového typu (v Ruby konkrétně musí být instancí třídy, která má definovanou metodu `hash` tak, aby vracela jednoznačný klíč). Analogicky platí vše, co bylo řečeno o polích.

4.8 Struct

Třída `Struct` umožňuje seskupit několik proměnných do logického celku podobně jako `struct` v jazyce C. Samotná třída slouží jako generátor specializovaných tříd obsahujících zadané proměnné a přístupové metody k nim.

4.9 Speciální třídy

V Ruby existují tři speciální třídy, které mají za běhu programu vždy pouze jedinou instanci. Jedná se o

- NilClass – s instancí `nil` (reprezentuje nedefinovanou hodnotu)
- TrueClass – s instancí `true` (reprezentuje logickou pravdu)
- FalseClass – s instancí `false` (reprezentuje logickou nepravdu)

Stojí za zmínku, že v logických operacích vystupují pouze hodnoty `nil` a `false` jako nepravda. Všechny ostatní hodnoty (např. 0 nebo prázdný řetězec) jsou vyhodnoceny jako pravdivé.

4.10 Další datové třídy

Ve standardní knihovně existuje podpora pro další typy dat. Jedná se například o třídu `Time` pro uchování času nebo třídu `Date` pro uchování datumu.

5 Operátory

5.1 Operátory s neměnným významem

Následující operátory mají v Ruby pevně stanovený význam.

- `...` – operátor pro vytváření intervalu
- `!`, `not` – operace NOT
- `&&`, `and` – operace AND
- `||`, `or` – operace OR
- `::` – metoda třídy (např. `String::upcase`)
- `=` – přiřazení
- `+=` apod. – zkrácené operátory přiřazení
- `?:` – ternární operátor

5.2 Operátory realizované metodami

Ruby obsahuje další operátory, které víceméně odpovídají jazyku C. Jejich činnost je zajištěna voláním metody. Například výraz `a + b` odpovídá volání `a.+(b)`. Předefinováním příslušné metody lze docílit změny funkce operátoru (někdy označované jako přetěžování). Po změně operátoru typu `+`, `-` atd. se automaticky upraví ifunkce příslušného operátoru zkráceného přiřazení `+=`, `-=` atd.

Je třeba upozornit, že Ruby nemá inkrementální operátor `++` (respektive `--`).

6 Řídící příkazy

6.1 Blok

Obecný blok zdrojového kódu lze v Ruby ohraničit klíčovými slovy `begin` a `end`. Výsledkem vyhodnocení bloku je hodnota, která je rovná hodnotě posledního vyhodnoceného výrazu v bloku.

6.2 Podmínky

K dispozici je podmíněný příkaz `if` a obrácený příkaz `unless`. Použití je obvyklé:

```
if a == 1
  b = 2
elsif a == 2  # nepovinná klauzule
  b = 3
else          # nepovinná klauzule
  b = 4
end

unless a == 1
  b = 4
else          # nepovinná klauzule
  b = 2
end
```

Oba příkazy lze použít také jako modifikátory jiných příkazů. Použití je pak následující:

```
b = 2 if a > 3
b = 3 unless a < 5
```

Pro podmínky definované výčtem slouží příkaz `case`.

```
x = 3
case x
  when 2
    puts "CC"
  when 3
    puts "CCC"
  else
    puts "C"
end
```

6.3 Cykly

Z klasických cyklů nabízí Ruby `while` a cyklus s obrácenou podmínkou `until`.

```
a = 0
while a < 3
  puts a
  a += 1
end

a = 0
```

```
until a == 3
  puts a
  a += 1
end
```

Také příkazy cyklů lze použít jako modifikátory. V kombinaci s blokem lze dosáhnout cyklu s testem podmínky na konci.

```
a = 0
begin
 puts a
 a += 1
end while a < 3
```

V rámci cyklu lze použít další řídící příkazy:

- `break` – přeruší vykonávání cyklu
- `next` – spustí další iteraci (řízení se přesune na konec bloku cyklu)
- `redo` – opakuje aktuální iteraci (řízení se přesune na začátek bloku cyklu)

6.4 Ošetření výjimek

Pro ošetření výjimek se v Ruby používá konstrukce:

```
begin
  ...
rescue Exception
  ...
ensure
  ...
end
```

Mezi klíčovými slovy `begin` a `rescue` leží blok kódu, v němž chceme ošetřit výjimky. Ošetření konkrétního druhu výjimky je obsaženo v klauzuli `rescue`. Druh výjimky je reprezentován různými potomky třídy `Exception`.

Vlastní výjimku lze vyvolat příkazem `raise`.

7 Třídy

Základem OOP je definice tříd. Následuje ukázka v Ruby.

```
class Klass1                # třída Klass1

  def initialize(a, b)      # metoda initialize
    @a = a                  # přiřazení parametru do proměnné
    @b = b
  end
```

```
  def to_s
    "A = #{@a}, B = #{@b}"  # výpis proměnných v řetězci
  end

end
```

V příkladu je definována třída s názvem `Klass1`. Protože nemá uvedeného předka, stává se automaticky potomkem kořenové třídy `Object`. Třída (název v Ruby začíná obvykle velkým písmenem) má definovány dvě metody (název začíná obvykle malým písmenem). Metoda `initialize` je speciální metodou, která je volána v okamžiku vzniku instance. Jejím úkolem je nastavit počáteční stav objektu – tj. inicializovat proměnné.

V uvedeném příkladu přiřazuje `initialize` hodnoty parametrů do proměnných, jejichž název začíná znakem `@`. To jsou proměnné, které bude mít každá instance dané třídy. Nepřítomnost deklarace vyžaduje speciální znak na začátku jména, aby interpret rozpoznal lokální proměnné (či parametry) od proměnných objektu (třeba v přiřazení `@a = a`).

V Ruby existuje konvence, že voláním metody `to_s` získáme řetězcovou reprezentaci objektu. V našem případě bude metoda `to_s` vracet informaci o obsahu proměnných objektu.

Instanci námi definované třídy vytvoříme voláním metody `new`.

```
k = Klass1.new(2, 8)
```

Metoda `new` (která slouží jako konstruktor) se neváže ke konkrétní instanci, ale ke třídě. Voláme ji tedy i se jménem třídy jako `Klass1.new`. Pokud bychom chtěli zobrazit textovou reprezentaci vytvořené instance, použili bychom volání `k.to_s`.

Ruby nepodporuje vícenásobnou dědičnost. Nabízí však mechanismus, kterým lze dosáhnout podobných výsedků. Jedná se o tzv. mixiny, které budou popsány dále.

Důležité také je, že definice třídy není nikdy ukončená. To znamená, že i do vestavěných tříd lze přidávat nové metody.

7.1 Proměnné

Přistupovat k proměnným objektů lze pouze přes metody dané třídy (to odpovídá striktnímu dodržení principu zapouzdření). Aby nebylo nutné v triviálních případech rutinně vytvářet přístupové metody, nabízí Ruby zkratku.

```
class Klass1          # pokračujeme v definici třídy Klass1

  attr_reader :a, :b  # metody pro čtení @a a @b
  attr_writer :a, :b  # metody pro nastavení @a a @b

end
```

Nyní je možné nastavit hodnotu proměnné voláním `k.a = 3`. Jinými slovy voláme metodu `a=` instance třídy `Klass1` a předáváme jí parametr `3`.

Kromě proměnných instancí, můžeme mít také proměnné společné pro celou třídu (tzv. class variables). Klasickým příkladem je počitadlo instancí.

```
class Klass1   # pokračujeme v definici třídy Klass1

  @@count = 0  # class variable je nutné inicializovat
               # v definici třídy
end
```

K proměnným tříd mohou přistupovat instance dané třídy nebo metody třídy.

```
class Klass1   # pokračujeme v definici třídy Klass1

  def Klass1.count  # metoda třídy
    @@count
  end

end
```

Obsah proměnné `@@count` lze nyní zjistit jako `Klass1.count`.

7.2 Metody

Příklady definice metod již byly uvedeny dříve. Obecně má definice tvar:

```
def methodName(param1, ...)
  ...
end
```

Parametrům lze přiřadit defaultní hodnoty. Proměnný počet parametrů lze snadno realizovat, protože parametr může být objekt třídy `Array`. Návrat hodnoty se provádí příkazem `return`. Pokud není `return` použit, vrací se hodnota posledního vyhodnoceného výrazu v metodě (viz příklad metody `count`).

Přístup k metodám lze nastavit pomocí klíčových slov `public`, `protected` a `private`, která mají podobný význam jako v jazyce Java.

8 Speciality Ruby

8.1 Blok jako parametr

Kromě klasických parametrů, lze metodě v Ruby předat blok kódu. Tento kód může být v rámci metody volán pomocí klíčového slova `yield` včetně předání parametrů. Blok kódu je v tomto případě ohraničen klíčovými slovy `do` a `end` nebo složenými závorkami. Syntaxe je následující:

```
def plus2(x)
  yield x + 2  # volání bloku s parametrem
end

plus2(5) { |x|
  puts x       # tento příkaz je metodě plus2 předán
               # jako blok kódu
}
```

8.2 Iterátory

Klasické cykly jsou v Ruby často nahrazeny speciálními metodami nazývanými iterátory. Iterátory jsou typicky metody kontejnerových tříd, které volají blok a jako parametr mu postupně předávají prvky kontejneru. Obecným iterátorem je metoda `each`.

```
[3, 5, 11].each { |i|  # iterátor each třídy Array prochází
  puts i               # prvky pole
}
```

Iterátor `each` je dostupný také jako varianta for cyklu (klasický for cyklus v Ruby není implementován).

```
for i in [3, 5, 11]
  puts i
end
```

8.3 Mixiny

V Ruby existuje možnost organizace metod do modulů. Modul je ohraničená část kódu velmi podobná definici třídy. Modul lze pomocí příkazu `include` připojit (tzv. namixovat) k definici třídy a rozšířit ji tak o metody definované v modulu). Tohoto mechanismu je v Ruby elegantně využito u vestavěného modulu `Enumerable`. Pokud má kontejnerová třída definován obecný iterátor `each`, získá namixováním modulu `Enumerable` mnoho dalších metod umožňujících práci s prvky kontejneru.

Vestavěná třída `Array` má modul `Enumerable` také namixovaný. Může proto využít iterátor `find_all` definovaný v modulu. Tento iterátor vrací pole všech prvků, pro které se blok vyhodnotí jako pravda.

```
[3, 6, 8, 9].find_all { |x| x % 2 == 0 }  # 6, 8
```

8.4 Thready

Interpret Ruby má vlastní implementaci programových vláken. Výhodou tohoto řešení je zaručená přenositelnost a integrace s jazykem, nevýhodou na druhé

straně je, že implementace neodpovída standardu POSIX a není příliš vhodná pro aplikace s velkým množstvím vláken.

Základní práce s thready je velmi snadná. Nové vlákno vytvoříme jako instanci třídy `Thread`. Při vytváření můžeme jako parametr rovnou předat blok kódu, který bude v rámci threadu vykonáván. (Aniž bychom vytvářeli nová vlákna, bude existovat minimálně jedno hlavní a jedno pro garbagge collector.)

```
t1 = Thread.new {  # nové vlákno
  sleep 1          # bude vteřinu čekat
  puts "T1"
}

t2 = Thread.new {  # nové vlákno
  puts "T2"
}

puts t1.status     # "sleep"

t1.join            # program se neukončí před dokončením vlákna t1
t2.join            # program se neukončí před dokončením vlákna t2
```

Výsledkem bude velmi pravděpodobně (protože časování nelze stoprocentně předpovědět) výpis:

```
T2
sleep
T1
```

8.5 Singletony

V Ruby je možné rozšířit o nové metody konkrétní objekt. Provede se to pomocí přiřazení speciální nové třídy. Přidaná anonymní třída se nazývá singleton, protože má jen jednu instanci – objekt, pro který byla vytvořena. Předkem singletonu se stává původní třída objektu.

```
class << k         # anonymní třída pro objekt k

  def otherMethod  # tato metoda bude k dispozici jen instanci k
    ...
  end

end
```

9 Rozhraní aplikací

9.1 Aplikace s GUI

Pomocí vazby na různé grafické toolkity lze v Ruby vytvářet i aplikace s grafickým uživatelským rozhraním. Knihovny s různou úrovní použitelnosti exis-

tují pro Gtk, Qt, wxWindows, Fox. Z hlediska použitelnosti a přenositelnosti je zřejmě nejlepší vazba na Fox toolkit (`http://www.fox-toolkit.org/`). Příslušná knihovna pro Ruby se jmenuje FXRuby a aktuální verze je ke stažení na `http://fxruby.sourceforge.net/`.

9.2 Webové aplikace

Již ve standardní knihovně existuje podpora pro psaní CGI skriptů v Ruby. Pomocí modulu mod_ruby (`http://www.modruby.net/`) pro webový server Apache lze integrovat interpret přímo do serveru a urychlit vykonávání skriptů.

Aplikace Eruby (na stejné adrese) umožňuje vykonávat kód Ruby, který je vložen uvnitř libovolných dokumentů. Kombinace mod_ruby a Eruby umožňuje používat Ruby při vývoji webových aplikací podobně jako PHP. Následující fragment HTML s vloženým kódem Ruby vypisuje do stránky aktuální čas.

```
<body>
Time: <% print Time.new %>
</body>
```

10 Závěr

Silné skriptovací jazyky si pravděpodobně udrží svou popularitu i nadále. Ruby není díky relativnímu mládí a promyšlenému vývoji zatížen některými nedostatky návrhu, které brání jiným jazykům ve snadné reakci na nově se objevující požadavky. Rostoucí komunita uživatelů, která zahrnuje mimo jiné pracovníky NASA či výzkumných laboratoří IBM nebo HP, skýtá záruku, že ovládnutí Ruby není jen dobrou zábavou. K dispozici jsou také první zkušenosti z využitím Ruby při rozsáhlých projektech, které hovoří o zkrácení doby vývoje, úspoře zdrojů a snažší údržbě.

Uvedené skutečnosti vedou k závěru, že Ruby v silné konkurenci na poli programovacích jazyků obstojí a může se stát univerzálním nástrojem mnoha vývojářů.

Reference

1. Matsumoto, Y.: Ruby in a Nutshell, O'Reilly&Associates, 2001.
2. Thomas, D., Hunt, A.: Programming Ruby (`http://www.rubycentral.com/book`).
3. Šrámek, D.: Ruby z rychlíku (`http://www.insula.cz/dali/material/rubyzr.php`).
4. Šrámek, D.: Ruby a OOP (`http://www.insula.cz/dali/material/rubyoop.php`).

Numerické výpočty v průmyslové praxi a jejich clustering

Milan Zajíček

ÚTIA AV ČR, Pod Vodárenskou věží 4, 182 08, Praha 8
Email: zajicek@utia.cas.cz

Abstrakt: Při návrhu výrobků v nejrůznějších průmyslových odvětvích se ve stále větším měřítku uplatňují numerické simulace. Příspěvek podává informaci o základních přístupech k tvorbě matematických a fyzikálních modelů. Na řadě příkladů ilustruje možnosti konfrontace konstrukčního řešení s virtuálním modelem reality. Zároveň je proveden rozbor začlenění numerických simulací mezi fáze návrhu nového výrobku. Na jednoduchém příkladu je ilustrována zpětná vazba mezi konstruktérem a výpočtářem. Jsou zmíněny komerční výpočetní systémy CaePipe, COSMOS/M, FIDAP, FLUENT, Polyflow a Flowmaster.
V příspěvku je nastíněna problematika tvorby výpočetních clusterů a jejich výkonové srovnání pro výpočetní systém FLUENT na systémech HP s operačními systémy Linux, HP-UX a Windows NT.

Klíčová slova: cluster, technické výpočty, MKP, CFD, paralelizace

1 Pevnostní analýzy

V konstruktérské praxi neustále vyvstává otázka: „Je řešení, které jsem zvolil, správné?“ Žádná počítačová simulace sice nedokáže tuto otázku plně zodpovědět, ale moderní výpočetní systémy dokáží již dnes velice efektivně poradit, která z návrhových variant je lepší či horší.

1.1 COSMOS/M

Mezi přední výpočetní systémy pro analýzu konstrukcí je systém COSMOS/M firmy Structural Research Analysis Corporation[1]. Na rozdíl od ostatních je však speciálně zaměřen na platformu PC. Díky tomu, že byl od základu naprogramován s ohledem na využití poznatků o zpracovávání rozsáhlých databází a používá patentovaných FFE (Fast Finite Element) řešičů, je ve svém výsledku systémem, který nemá konkurenci v poměru cena/výkon.

Pre a post procesor GEOSTAR obsahuje 2D/3D geometrický modelář. Intuitivní uživatelské rozhraní ovládá import geometrie z CAD systémů, ruční, parametrický i automatický generátor výpočetní sítě, zadávání okrajových podmínek

[1] http://www.srac.com

Jan Kasprzak, Petr Sojka (editoři): SLT 2002 – sborník semináře o Linuxu a TEXu, str. 233–247, 2002.

pro všechny typy strukturálních analýz a zobrazení výsledků v grafické i textové formě.

Samotný výpočetní modul COSMOS/M umožňuje řešit na běžných počítačích PC reálné úlohy z oblasti lineární i nelineární statiky, stability, problematiku vlastních frekvencí, stacionárních i nestacionárních teplotních polí, ale i odezvu na vnější buzení, kontaktní úlohy, geometrické a materiálové nelinearity a únavu. Každá z těchto analýz vyžaduje zvláštní modul. Modulární stavba systému COSMOS umožňuje mimo jiné i to, že uživatel může zakoupit pouze ty moduly, které pro svoji práci potřebuje.

GEOSTAR – pre a post procesor, 3D geometrický modelář
STATICS – lineární statika, kontakt, řešení sestav
DYNAMICS – výpočet vlastních frekvencí a stability
THERMAL – stacionární a nestacionární teplotní pole včetně teplotně závislých materiálových vlastností
ASTAR – odezva na vnější buzení se zahrnutím tlumení obsahující nestacionární analýzu, spektrum odezvy (seismicita), generaci spektra odezvy (seismicita), náhodné kmitání, stacionární harmonickou analýzu, transformaci křivek popisujících buzení (seismicita)
NSTAR – nelineární statická a dynamická analýza, materiálové a geometrické nelinearity, plasticita, hyper a viskoelasticita, creep, nelineární stabilita, nelineární kontakt
FSTAR – únava a životnost
OPSTAR – optimalizace a citlivostní analýza
EMS – nízkofrekvenční elektromagnetismus
HIFIS – vysokofrekvenční elektromagnetismus

Získanými výsledky analýz jsou například: napětí, deformace, výchylky, deformovaný tvar, posunutí, deformační energie, hustota deformační energie, výpis vlastních frekvencí, tvar vlastních kmitů, teplota, teplotní gradient, tepelný tok, ... Výsledky mohou být zobrazeny jako grafy závislosti sledovaných veličin na poloze i čase, kontury, isoplochy, vektory či animace.

1.1.1 COSMOS/DesignSTAR V posledních několika letech se stále výrazněji prosazuje trend přesouvat výpočtářské práce směrem od specializovaných výpočtářských oddělení do konstrukčních kanceláří – přímo do rukou návrhářů a konstruktérů zařízení. Důvod je prostý. Může-li si konstruktér sám a rychle utvořit názor, zdali je jeho konstrukční návrh z pevnostního hlediska v pořádku, dokáže pružněji a hlavně rychleji navrhnout optimální a kvalitní výrobek.

Program COSMOS/DesignSTAR je produkt, který toto všechno konstruktérovi umožňuje. Díky úzké provázanosti s CAD, snadnému a intuitivnímu ovládání, robustnímu automatickému generátoru sítě, rychlým a úsporným řešičům systému COSMOS/M a kvalitnímu a rozsáhlému postprocesingu se stal COSMOS/DesignStar nejúspěšnějším MKP programem v této oblasti.

1.2 Caepipe – specialista na potrubí

Kromě „univerzálních“ programů pro řešení pevnostních úloh existuje i celá řada programů specializující se na poměrně úzký okruh problémů. Příkladem takového systému může být například program CAEPIPE[2]. Tento program slouží k analýze pevnosti potrubních tras. K tomuto účelu je vybaven knihovnou specielních prvků, jako jsou nejrůznější druhy podpor, závěsů či armatur. Výpočtář se tedy nemusí zabývat detailním modelováním jednotlivých součástí, ale je na něm, aby složením již připravených prvků vytvořil model potrubní trasy, který bude obrazem reálného díla.

Výsledkem analýzy programem CAEPIPE jsou jednak síly a posunutí v jednotlivých místech potrubní sítě, dále síly působící na nosníky a podpory a také hodnoty vlastních frekvencí a tvary vlastních kmitů. Program umožňuje rovněž hodnocení potrubní sítě podle řady mezinárodně uznávaných bezpečnostních kódů např. ASME, DIN, ...

2 Kinematika a dynamika mechanismů

Tato kapitola je věnována programu MSC.visualNastran Desktop[3]. Pod tímto názvem se nabízí moderní systém pro řešení kinematiky a dynamiky mechanismů. Program umožňuje snadno ověřit funkčnost konstrukce mechanismu, získat realistické průběhy rychlostí, zrychlení a sil působících na jednotlivá tělesa a tím odstranit řadu nedostatků ještě před vyrobením prvního prototypu a zvýšit tak kvalitu výsledného produktu a zároveň podstatně snížit celkové náklady na vývoj výrobku.

Intuitivní přístup činí program snadno ovladatelným a umožňuje tak řešit úlohy ve velmi krátkém čase. Postup práce je následující: (1) Tuhá tělesa vytvořená v CAD systému a nebo vlastním preprocesoru se spojí kinematickými vazbami. (2) Model se doplní pružinami, tlumiči (i nelineárními) a vnějšími silami, momenty, přimočarými i rotačními motory. (3) Doplnění závislostí jednotlivých veličin vzorcem a nebo tabulkou zadávaných hodnot.

K unikátním přednostem programu patří automatická detekce kolizí a kontaktů pohybujících se součástí s následným výpočtem odpovídajících reakcí – kontaktních a třecích sil a změn hybností. Lze tak snadno simulovat systémy geometricky složitých těles, která po sobě kloužou nebo na sebe narážejí – například vačkové mechanismy, elektrické spínače, manipulační zařízení včetně dopravovaných objektů a další.

Po spuštění simulace se zároveň s výpočtem vykreslují na obrazovce animace pohybů mechanismu a průběhy vypočtených veličin ve formě grafů. Výsledky lze zíkat samozřejmě i jako textové soubory, které lze následně použít jako vstupní hodnoty pro výpočet statického i dynamického namáhání součástí metodou konečných prvků. Program MSC.visualNastran Desktop je určen nejen pro

[2] `http://www.sstusa.com/`

[3] `http://www.krev.com/` – Donedávna distribuován pod názvem Working Model Motion

výpočtáře specialisty, ale především pro konstruktéry, kterým umožňuje snadno a rychle porozumět chování konstruovaného mechanismu, odhalit jeho slabá místa ještě před vyrobením prvního prototypu, vyzkoušet si řadu variant a úprav typu „Co kdyby?“ a vybrat tu nejlepší. Lze se tak vyhnout mnoha technickým problémům a při nepoměrně menších nákladech na vývoj uvést na trh výrobek lepší než konkurence a to v kratším čase.

3 Výpočty proudění

Zatímco pevnostní a kinematické výpočty se již stávají běžnou součástí konstruktérské praxe, jsou programy pro výpočtet proudění stále ještě převážně záležitostí specializovaných výpočetních center. Tento stav má tři základní důvody. (1) Fyzikálně složitější problémy vyžadují dostatečně způsobilého výpočtáře, který dokáže využít možnosti CFD[4] výpočetního systému. (2) Ve valné většině jsou otázky pevnosti úzce sepjaty s výrobou a jeví se tedy relativně důležitější, nežli otázky proudění. (3) Cena programového vybavení může být i řádově vyšší, nežli cena programů pro pevnostní analýzu. Na druhé straně je pravdou, že dobře provedená analýza proudění, ať už za účelem zlepšení přestupu tepla, zkvalitnění spalovacího procesu či snížení hlučnosti, může ve svém důsledku přinést mnohem větší finanční úspory a velký náskok před konkurencí, neboť se jedná o kvalitativně vyšší úroveň poznání dějů probíhajících v zařízení.

3.1 Preprocesor GAMBIT

Základní funkce programu pro přípravu geometrických dat pro CFD výpočet je korektní načtení 3D modelu pomocí některého z univerzálních CAD transportních formátů (IGES, SAT, STEP, ...) a zjednodušení geometrie s ohledem na zanedbání, pro proudění nevýznamných, detailů. Další funkcí GAMBITu je generování povrchové a objemové výpočetní sítě. Výpočetní síť může být v případě jednoduchých geometrií tvořena šestistěny, v případě složitých geometrií čtyřstěny. Gambit umožňuje rovněž tvorbu tzv. hybridních sítí, obsahujících oba dva druhy buněk, které jsou vzájemně spojeny přechodovou vrstvou z pětistěnů (pyramidové prvky). Poslední funkce, kterou GAMBIT nabízí je definice ploch i objemů pro budoucí zadání okrajových podmínek v závislosti na typu použitého řešiče. Výpočetní síť z tohoto preprocesoru může být použita pro výpočet libovolným ze systémů dodávaných firmou Fluent Inc., kterými jsou FLUENT 4, FLUENT 5, FIDAP, NEKTON a POLYFLOW, výpočetní síť je možno rovněž exportovat i do dalších formátů, které jsou použitelné pro většinu komerčních CFD i MKP programů.

[4] CFD – Computational Fluid Dynamics, čili počítačová dynamika tekutin, což může být chápáno, jako počítačem simulované chování kapalných látek v nejrůznějších geometriích.

3.2 Fluent

Program Fluent[5] je z fyzikálně-matematického hlediska založen na metodě konečných objemů FVM (Finite Volume Method). Na výpočetní síti konečných objemů probíhá numerické řešení fyzikálních rovnic proudění, přenosu tepla a dle povahy řešeného problému i dalších vztahů popisujících zkoumané jevy.

Program se používá pro řešení úloh z oblastí:

Automobilový průmysl – aerodynamická optimalizace tvaru karoserie, spojlerů, zpětných zrcátek, klimatizační a větrací kanály, chlazení motoru i motorového prostoru, proudění ve spalovacím prostoru, proudění v čerpadlech, filtrech a hydraulických agregátech, odmrazování skel...

Letecký průmysl – externí aerodynamika křídel a trupu, agregáty rozvodů vzduchu i paliva a další úlohy obdobné automobilovému průmyslu...

Energetika – proudění v turbinách, proudění v jaderných reaktorech, spalování v kotlích, návrh hořáků na plynná, pevná i kapalná paliva, optimalizace přestupu tepla ve výměnících, snižování obsahu škodlivých emisí ve spalinách...

Aplikace pro technologie životního prostředí – kromě výše uvedených se jedná zejména o návrh klimatizace a vytápění budov, analýzy rozptylu znečisťujících látek v terénu...

Biotechnologie a medicína – chlazení, sterilizace, větrání čistých a superčistých prostor, filtrace, inkubátory, kryogenní technika...

Práce výpočtáře spočívá při použití programu FLUENT v: (1) Získání a upravení geometrie do tvaru použitelného pro generaci výpočetní sítě. (2) Vygenerování výpočetní sítě s ohledem na zkoumaný jev a předpokládaný charakter proudění. (3) Získání a korektní zadání okrajových a počátečních podmínek výpočtu spolu s parametry použitých modelů turbulence, spalování atd. (4) Ladění úlohy vzhledem k informacím známým z experimentu či praxe. (5) Kritické analýze výsledků a diskusi vlivu parametrů výpočtu na jeho výsledky. Značnou část doby potřebné pro analýzu zabere tzv. ladění úlohy, tedy úpravy okrajových podmínek a parametrů modelu tak, aby co nejlépe odpovídal údajům známým z měření či praktického provozu. Detailní postup práce je uveden například v článku [4]. Výsledkem práce výpočtáře je komplexní informace o tlacích, rychlostech, teplotách, koncentracích, intenzitách turbulence a všech dalších hodnotách, které byly účastny výpočtu. Z těchto hodnot lze pak navíc získávat i hodnoty odvozené, takže například integrací tlaku přes plochu můžeme získat velikost tlakové síly na tuto plochu působící.

3.2.1 Preprocesory MixSim, IcePak, AirPak Kromě univerzálního preprocesoru GAMBIT, který umožňuje tvorbu libovolné geometrie stylem obvyklým v CAD systémech, má program FLUENT i speciální moduly vytvořené za účelem zrychlení a zefektivnění tvorby úloh ze specifických inženýrských oblastí. Jedná se o:

[5] http://www.fluent.com

MixSim – Preprocesor pro návrh a tvorbu výpočetních oblastí míchacích zařízení. Přehledně a rychle umožňuje zadat tvar nádoby (válcová, eliptická, hranol . . .), tvar dna (ploché, vyduté, kuželové . . .), umístění míchadla, jeho druh a rozměry, umístění vodícího válce i další charakteristické hodnoty. Na základě výše uvedených údajů je automaticky vygenerována výpočetní síť a spuštěn výpočet.

IcePak – Program umožňující tvorbu výpočetních oblastí typických pro elektroniku. Výpočetní oblast je tvořena na základě komponent (deska, procesor, ventilátor, . . .), které jsou vybírány z knihovny prvků. Generace výpočetní sítě je provedena automaticky a stejně tak i výpočet.

AirPak – Tento produkt je obdobou programu IcePak, ovšem pro oblast stavebního inženýrství. Prostřednictvím modulů umožňuje modelovat místnosti s nábytkem, otopnými systémy a řešit tak v krátkém čase úlohy klimatizace a vytápění. Program rovněž umožňuje zjednodušit řešení proudění v městské zástavbě, což je vhodné zvláště při simulacích rozptylu škodlivých emisí.

3.3 Polyflow

Uvážíme-li fakt, že se látky řazené mezi tekutiny mohou velice lišit (např. vodní pára a asfalt), je zřejmé, že i když pro simulaci proudění každé z nich lze použít obdobný fyzikální aparát, je mnohdy zapotřebí rozdílných postupů řešení. Proto kromě obecných a univerzálně použitelných programů vznikly programy určené právě pro určité oblasti. Typickým představitelem těchto programů je systém Polyflow[6] pro simulaci reologicky[7] složitých toků. Program umožňuje řešit úlohy vytlačování, vyfukování, potahování, vulkanizace a dalších procesů, které probíhají s materiály jako je sklo, plasty či guma. Jelikož jsou tyto úlohy mimo jiné charakterizovány velkými deformacemi a tvarovými změnami, musí program disponovat řadou specielních nástrojů pro jejich zvládnutí v souladu s fyzikálními zákony. Jsou to jednak postupy řešení a dále způsoby dynamické změny výpočetní sítě v závislosti na změnách tvaru výpočetní oblasti.

3.4 Flowmaster

Poněkud stranou, od ostatních zde zmíněných programů, stojí program pro analýzu potrubních sítí – Flowmaster[8]. Ten je určen k bilančním výpočtům rozsáhlých potrubních sítí. Umožňuje jak stacionární, tak i nestacionární analýzy dějů probíhajících v potrubí. Kromě standardních úloh lze program použít i pro výpočty silových hydraulických obvodů.

Výpočet probíhá na základě schematu znázorňujícího topologii potrubní sítě se zahrnutím všech komponent, které se v této síti vyskytují. Komponentou se v tomto případě rozumí např. trubka, koleno, T-kus, ventil, čerpadlo, nádrž,

[6] `http://www.polyflow.be/`

[7] Reologie je definována jako věda o deformaci a tečení hmoty [1]

[8] `http://www.flowmaster.com/`

zdroj tlaku či průtoku a řada dalších. Ke každé komponentě přísluší její vlastnosti, kterými jsou například průměry a drsnosti potrubí, ztrátové charakteristiky, čerpací charakteristiky, tepelné kapacity a další hodnoty podle typu výpočtu a způsobu modelování té které komponenty. Schema lze navíc doplnit o časové závislosti polohy ventilů, otáček čerpadel, či o regulační členy s definovanými přenosovými funkcemi a získat tak dynamický systém pro řešení přechodových či regulačních dějů. Výsledky výpočtu lze získat jednak ve formě grafů, dále jako ASCII soubor s utříděnými hodnotami výsledků pro jednotlivé komponenty a také jako číselné veličiny vložené do schematu k odpovídajícím komponentám.

Zmíněné vlastnosti předurčují program Flowmaster pro projekční kanceláře, které mají co do činění s rozvody vody, plynu, páry či jiných médií, ale také na výzkumná pracoviště, zabývající se řešením bilančních výpočtů v rámci složitých celků, jako jsou například rozvody paliva v motorech automobilů nebo letadel.

4 Srovnání výkonu výpočetního clusteru

Výpočty za pomoci clusterů jsou založeny na jednoduché myšlence vzájemného propojení několika počítačů a využití jejich společného výkonu na rychlé řešení jednoho problému [3]. Tento přístup k řešení složitých problémů není zdaleka nový – zvláště v oblastech náročných vědeckých a technických výpočtů, kde jsou požadována rychlá řešení. Neustále se snižující poměr ceny a výkonu hrdwarových zařízení i softwarových řešení, dostupnost vysokorychlostních datových sítí a vyspělost způsobů paralelního programování[9] mají za následek zvyšující se atraktivitu použití výpočetních clusterů pro inženýrské výpočty. V následujícím textu podrobíme sledování výpočetního cluster určený pro simulaci proudění kapaliny. Pro test budou použity následující produkty:

- CFD výpočetní systém firmy Fluent Inc.
- Čtyři rozdílné výpočetní systémy od firmy Hewlett-Packard
- Vysokorychlostní datová síť konsorcia HP a Myricom, Inc.

Výpočetní systém Fluent [2] umožňuje paralelizaci výpočtu již od roku 1992. Firma Hewlett-Packard poprvé představila cluster unixových pracovních stanic v roce 1993. Myricom představil svůj produkt Myrinet v roce 1994. Komerční aplikace a vývojářské nástroje, které jsou standardizovány pomocí knihovny MPI (Message Passing Interface), jejíž první verze byly vytvořeny v polovině devadesátých let, s sebou přinesly platformovou nezávislost paralelních kódů. Poté, co Hewlett-Packard v roce 1995 získal výpočetní systém Convex, představil šestnáctiprocesorový V-Class SMP server s HyperFabric síťovým propojením pro rozsáhlé HP UX 11.0 clustery. V současné době je silně patrný trend *levných clusterů* postavených na základě Intelovských zařízení s užitím operačního systému Windows NT, anebo GNU/Linuxu. Souběžně s tím si své místo stále drží i systémy osazené RISC-ovými procesory. Zaměřme se nyní na základní inženýrskou otázku, která vyvstává s nasazním výpočetních clusterů: „Do jaké

[9] Především jde o dostupnost a kvalitu efektivních algoritmů pro paralelizaci výpočtů.

míry mohou clustery řešit můj specifický problém?“ Účelem následujících kapitol je nabídnout průvodce, který upozorní na specifika, která přinášejí CFD simulace na jednotlivých platformách. Klade si za cíl naznačit otázky, na které je třeba odpovědět při výběru optimáilního řešení pro danou aplikaci.

4.1 Konfigurace jednotlivých clusterů

Konfigurace clusterů je následující:

- Konfigurace A
  - HP N-Class PA-RISC 8500 440MHz Unix server cluster
  - HP-UX 11.0
  - 4×8 CPU (celkem 32 procesorů)
  - Síť: HP – HyperFabric
- Konfigurace B
  - HP J5000 PA-RISC 8500 440MHz Unix Workstation cluster
  - HP-UX 10.20
  - 4×2 CPU (celkem 8 procesorů)
  - Síť: 100BT Ethernet, 1000BSX Ethernet
- Konfigurace C
  - HP XL550 Intel Pentium III Xeon 550MHz Intel Linux Cluster
  - RedHat Linux 6, kernel 2.2.12
  - 8×2 CPU (celkem 16 procesorů)
  - 100BT Ethernet – Myrinet
- Konfigurace D
  - HP X550 Intel Pentium III Xeon 550MHz Intel NT Cluster
  - Windows NT 4.0
  - 8×2 CPU (celkem 16 procesorů)
  - 100BT Ethernet

Pro testování výkonu byly použity úlohy o velikostech:

Malá – 32 000 buněk
Střední – 242 782 buněk
Velká – 3 618 080 buněk

Na všech systémech byl nainstalován výpočetní systém FLUENT ve verzi 5.3.2. Na všech strojích s HP-UX byla použita *message-passing library* HP MPI 1.5. Síť Myrinet nebyla použita na clusteru J5000, neboť Myricom v době testování neměl k dispozici HP-UX driver a HyperFabric sítě nemají na stanicích J5000 podporu. V případě Linuxu používá Fluent MPICH MPI knihovnu vyvinutou v Argonne National Laboratory. Výsledky na linuxovém clusteru byly získány použitím ovladače `ch_p4` a výsledky získané použitím sítě Myrinet odpovídají ovladači Myricom `ch_gm` pro zmíněnou knihovnu MPICH. Clusterovaný Fluent pod Windows NT používá PaTENT MPI knihovnu vyvinutou firmou Genias Software, GmbH.

4.2 Fluent z hlediska použití v clusterech

Jak již bylo uvedeno v předchozích kapitolách, Fluent umožňuje řešit řadu úloh z oblasti proudění kapalin. Mají-li výsledky svým tvarem odpovídat chování reálného světa, musí definice úlohy obsahovat řadu fyzikálních modelů zahrnující jednak dominantní, ale rovněž i zdánlivě zanedbatelné děje probíhající ve zkoumané soustavě. Spolu s potřebou řešit rozsáhlé úlohy to znamená veliké nároky na výkon procesorů, potřebnou velikost paměti počítače a rovněž na výpočetní čas potřebný k získání dostatečně přesného řešení. Tato fakta přímo předurčují Fluent a obdobné výpočetní systémy pro nasazení na paralelních počítačích a clusterovaných výpočetních systémech. Pro rozdělení výpočetní oblasti na jednotlivé části používané při výpočtu používá Fluent tzv. *doménovou dekompozici*. Princip metody je velice jednoduchý. Výpočetní oblast (doména), vyplněná výpočetní sítí, je rozdělana na jednotlivé podoblasti (subdomény). Každá podoblast je poté řešena individuálně na jednom z procesorů víceprocesorového počítače nebo výpočetního clusteru. V průběhu výpočtu musí být samozřejmě zaručena integrita globálního řešení na všech výpočetních uzlech. Toho je dosaženo předáváním dat mezi jednotlivými hraničními oblastmi vzniklými při dělení domény. Výkon paralelního výpočtu je určen vzájemným vztahem výpočetního výkonu jednotlivých výpočetních uzlů a kvality komunikace mezi nimi. Výpočetním uzlem budiž v tuto chvíli rozuměn jeden procesor, kterému přísluší řešení jedné subdomény. Je zřejmé, že péče věnovaná rozdělení výpočetní domény na jednotlivé subdomény může významně napomoci rychlosti výpočtu, neboť ideální dekompozice výpočetní oblasti by měla dosáhnout stavu, kdy jednotlivé výpočetní uzly budou zatíženy zcela rovnoměrně při minimální vzájemné komunikaci.

4.2.1 Řešiče Fluentu a jejich vliv na výpočetní výkon clusteru Numerická simulace znamená vždy řešení soustav algebraických rovnic. Fluent používá tři formulace řešení, které lze zvolit v závislosti na řešeném problému. Jde o řešiče: *segregovaný*, *coupled implicit* a *coupled explicit*. Segregovaný řešič řeší jednotlivé rovnice matice soustavy odděleně (segregovaně), naproti tomu coupled řešiče spojují rovnice dohromady a řeší je současně. Explicitní a implicitní formulace vyjadřuje způsob, jakým je prováděna linearizace soustav rovnic. Všechny tři řešiče jsou schopny poskytnout přesné řešení pro většinu fyzikálních problémů. Důvodem jejich použití je vhodnost pro specifické okruhy úloh – například rychlejší konvergence. Úlohy uváděné v tomto příspěvku byly řešeny výhradně za pomoci segregovaného řešiče. Ten totiž představuje nejhorší možný případ z hlediska objemu komunikace mezi výpočetními uzly. Je to dáno tím, že v každé iteraci je třeba přenést hodnoty na hraničních plochách oblasti a jelikož nedochází k párování rovnic, je třeba tento přenos provést v každé iteraci výpočtu pro každou rovnici.

4.3 Něco o clusterech

V obecné rovině lze pojem cluster chápat velmi jednoduše jako skupinu vzájemně pracujících a komunikujících počítačů. Pro použití clusterů existuje řada

praktických důvodů, například dostupnost aplikace, možnost nepřetržitého běhu s automatickou detekcí chyb, kdy se po havárii některého z uzlů clusteru převede jeho funkce na všechny ostatní bez přerušení běhu aplikací. Použití clusterů rovněž umožňuje zvýšení průchodnosti, které je často důležité při zpracování databází, kdy každému procesoru bývá přiřazena nezávislá úloha. Tento přístup, kdy je každému z procesorů clusteru přidělena jedna nezávislá úloha je v protikladu s použitím clusteru, kdy je všem procesorům přidělen jeden jediný úkol, který by byl příliš veliký, anebo příliš výpočetně náročný, pro řešení na jediném stroji – toto je případ jakým clustery využívá Fluent. Základními prvky clusterů pro běh Fluentu je skupina počítačů, jejich síťové propojení a operační systém. Jednotlivé uzly clusteru mohou být jednoprocesorové počítače, nebo může jít o počítače víceprocesorové. Výkon clusteru je funkcí typu procesoru, síťového propojení jednotlivých počítačů a vzájemného poměru výkonu jednotlivých počítačů. Samozřejmý je rovněž vliv kvalitního sdílení zdrojů a kvality softwaru, který toto sdílení obstarává. Clustery představují možnost jak získat výpočetní výkon a některé vlastnosti, které byly donedávna výsadou vektorových superpočítačů a mnohaprocesorových výpočetních systémů. Výhodou je mimo jiné i to, že clustery je možno dynamicky rozšiřovat v závislostech na potřebách firmy.

4.3.1 Typ procesoru Technologie clusterování je nezávislá na typu procesoru, jehož rychlost je však jedním z hlavních faktorů ovlivňujících celkový výpočetní výkon clusteru. Cluster může být vytvořen s použitím levných systémů (Pentium), ale pokud požadujeme robustní systém, který umožňuje alokaci většího množství paměti, pravděpodobně bude vhodné použít řešení s PA-RISC procesory.

4.3.2 Návrh velikosti clusteru Pro danou velikost CFD problému je třeba brát v potaz, že množství procesorů zpracovávajících jednotlivou úlohu zrychluje výpočet, zatímco rozdělení úlohy na více částí zatěžuje síťovou komunikaci. Z toho vyplývá, že existuje kritické množství subdomén, na které je vhodné danou úlohu rozdělit. Po překročení kritického počtu procesů dojde k tomu, že jednotlivé výpočetní uzly jsou zdržovány čekáním na síťovou komunikaci a tento stav se s přidáním každého dalšího výpočetního uzlu zhoršuje. Optimální počet procesorů se zvyšuje spolu s tím, jak se zvětšuje velikost CFD problému. Architektura clusteru, zvláště typ procesoru a propustnosti sítě, má na optimální počet procesorů clusteru rovněž nezanedbatelný vliv.

4.3.3 Paměť a její sdílení Rozdíl mezi použitím clusterů a symetrickým multiprocessingem (SMP) je ve sdílení paměti a její dostupnosti. Paměť SMP počítače je dostupná kterémukoli ze všech procesorů v tomto stroji (je sdílená), zatímco paměť v clusteru počítačů je rozdělena (distribuována) mezi jednotlivé uzly clusteru. Tento rozdíl muže hrát významnou roli v případě, že je třeba rozhodnout mezi velkým SMP počítačem, anebo clusterem několika dvouprocesorových strojů. Během paralelních výpočtů na víceprocesorových SMP počítačích

se jednotlivé procesory *přetahují* o sdílenou paměť. Toto *přetahování* může při velkém počtu procesorů znamenat faktor vedoucí ke snížení výkonu clusteru. Z tohoto pohledu může být výhodné distribuovat výpočet na více (například dvouprocesorových) počítačů.

4.3.4 Síťové propojení Rychlost síťě bývá většinou vnímána jako přenosová rychlost (MB/s). Tento údaj je významný, pokud potřebujeme přenášet data mezi jednotlivými uzly, ale pro funkci clusterů je rovněž důležitá *latence* sítě, tedy prodleva mezi požadavkem a odezvou (měří se v nanosekundách). Tabulka 1 srovnává čtyři použité druhy propojení, které se vyskytovaly v testovaných clusterech.

Tabulka 1. Srovnání síťových propojení clusterů.

Síť	**Rychlost** Vyšší je lepší	**Latence** Nižší je lepší	**Cena**
HyperFabric	Vysoká	Nízká	Vysoká
Myrinet	Vysoká	Nízká	Střední
1000BSX – 1GB opt. ethernet	Vysoká	Vysoká	Střední
100BT měď. ethernet	Nízká	Vysoká	Nízká

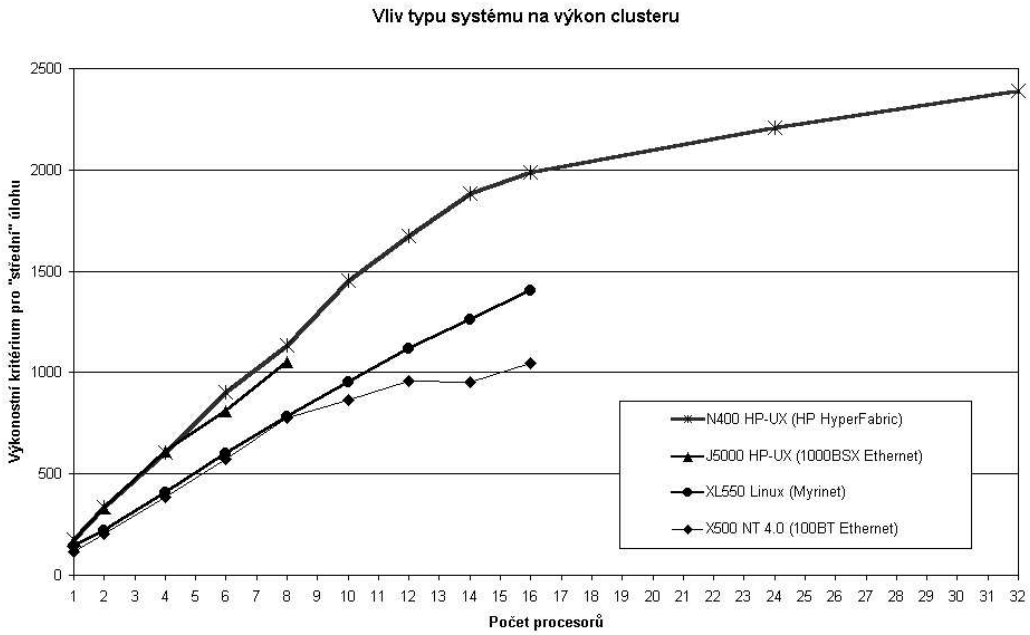
4.4 Výsledky testu clusterů

Testovací úlohy byly řešeny na výpočetních clusterech uvedených v kapitole 4.1 a zde budou uvedeny některé z výsledků[10]. V následujících grafech je výpočetní rychlost prezentována ve formě *výkonového kritéria* clusteru. Toto kritérium je definováno, jako počet úloh, které proběhnou na daném clusteru během 24 hodin. Je získáno jako podíl počtu sekund jednoho dne (86 400 s), počtem sekund za které proběhne jeden výpočet. Čím větší je výkonové kritérium, tím vyšší je výkon clusteru. Ideální cluster by měl vykazovat lineární nárůst výkonostního kritéria vzhledem k počtu procesorů. V praxi je však tento stav nedosažitelný vzhledem k režii systému a síťové komunikaci v rámci clusteru, jak je diskutováno v předchozích kapitolách.

4.4.1 Vliv druhu systému na výkon Na obrázku 1 je provedeno srovnání výkonového kritéria pro čtyři konfigurace systému uvedené v kapitole 4.1. Pozornost je věnována jak typu procesoru, tak kvalitě síťového propojení. Výsledky jsou znázorněny v závislosti na počtu procesorů. Jako testovací úloha byla zvolena úloha *střední* velikosti (viz. kap. 4.1). Výsledky ukazují výhodu rychlosti a efektivity PA-RISC procesorů použitých v počítačích J5000 a N4000,

[10] Podrobnější údaje lze nalézt na
`http://www.fluent.com/software/fl5bench/index.htm`

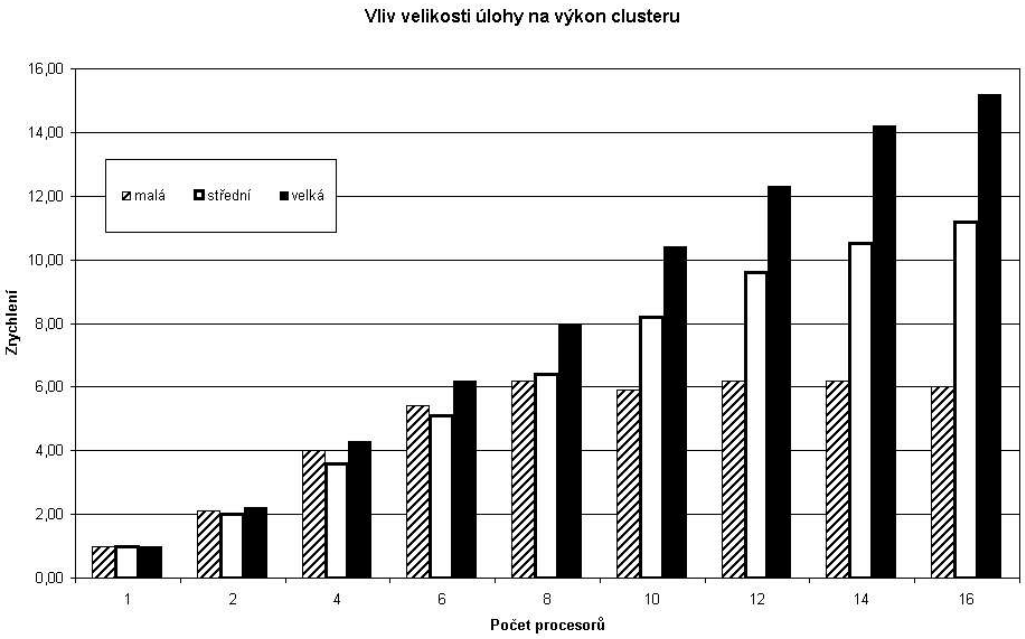
i když frekvence procesoru PIII Xeon je o 25 % vyšší nežli PA-8500. Výhodou N-Class serverů je rovněž převaha v množství procesorů, které je z velké části umožněno použitím síťového rozhraní HP HyperFabric. Pro pracovní stanice J-Class nebyla v době testování síť HP HyperFabric dostupná a bohužel ani Miricom v té době nedodával ovladače pr HP-UX. Z toho důvodu byl pro spojení stanic HP J-5000 použit gigabitový ethernet. Významný vliv kvality síťového propojení ukazuje i srovnání systémů s procesorem Pentium, na kterém je patrno, že při zapojení více jak osmi procesorů dochází k výraznému zpomalení NT systému propojeného standardním 100BT ethernetem, zatímco linuxový cluster používající sítě Myrinet vykazuje až do šestnácti procesorů téměř lineární nárůst výkonu.



Obrázek 1. Srovnání rychlosti mezi PA-RISC počítači (J-Class a N-Class) a počítačí s procesorem Pentium III Xeon (Linux a NT). PA-RISC je přibližně o 40 % rychlejší na jednom procesoru, o 50 % na osmi procesorech a o 85 % rychlejší na šestnácti procesorech než systém s Windows NT.

4.4.2 Vliv velikosti úlohy na výkon clusteru Jak již bylo zmíněno, přidání procesorů do clusteru zvýší množství výpočetního výkonu a tím zkracuje čas potřebný k řešení problému. Přidáním procesoru se ovšem zároveň zvyšuje režie síťového provozu. Obecně lze říci, že rozsáhlé clustery jsou vhodné pro řešení velkých úloh, zatímco jejich použití pro úlohy malých rozměrů může dobu řešení paradoxně (vzhledem k výpočetnímu výkonu procesorů) prodloužit. Tento fakt

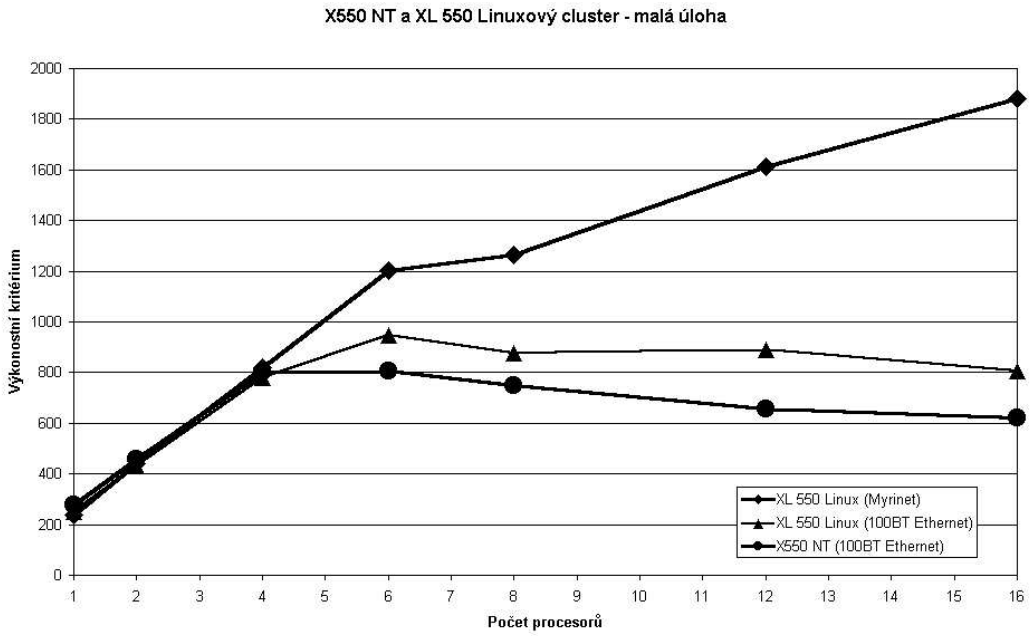
je ilustrován na obrázku 2, který ukazuje výsledky pro tři rozdílné úlohy řešené na clusteru HP-N4000. Se zvyšujícím se počtem procesorů od jednoho do šestnácti. Pro výpočet byly použity všecny tři úlohy uvedené v kapitole 4.1. Zatímco u *velké* úlohy je nárůst výkonu přibližně lineární až do šestnácti procesorů, v případě *malé* úlohy má cluster nejvyšší výkon v případě použití osmi procesorů a dále už nevzrůstá. To je způsobeno tím, že výpočetní nároky jsou nižší a rychlost řešení začíná výrazně záviset na meziprocesorové komunikaci.



Obrázek 2. Jak stoupá počet procesorů, dochází u *velké* úlohy k téměř ideálnímu nárůstu výkonu (10 procesorů je 10,5× rychlejších nežli jeden procesor; 16 procesorů je 15,2× rychlejších nežli jeden). U menších úloh je patrný efekt velké meziprocesorové komunikace, která se stává limitujícím faktorem pro výkon clusteru.

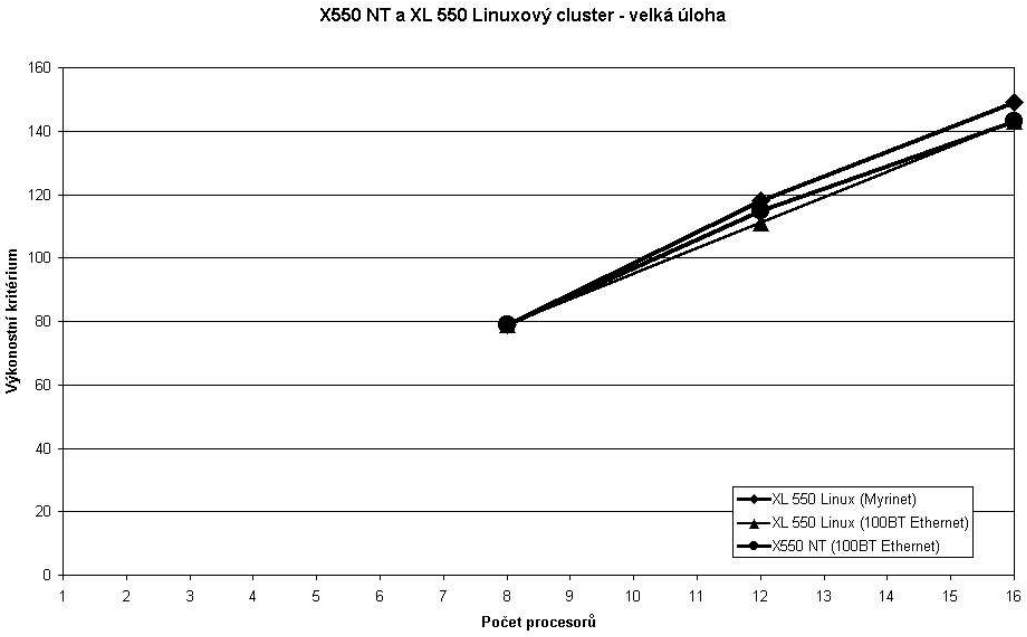
4.4.3 Vliv rychlosti sítě Dobrý paralelní výkon je do značné míry závislý na rychlosti s jakou jsou schopny komunikovat jednotlivé procesory v clusteru. To způsobuje poměrně velké nároky na kvalitu síťového hardwaru a vyvolává potřebu vysokorychlostních spojení s co nejnižší dobou odezvy. Na obrázku 3 je patrno, jak síť Myrinet s nízkou latencí (použitá na linuxovém clusteru) umožňuje kvalitnější rozšiřování clusteru, nežli klasický ethernet. V tomto případě je však nutno mít na paměti, že tyto výsledky byly získány na relativně malé úloze. V případě, že zvolíme velkou úlohu, jak je znázorněno na obrázku 4,

můžeme pozorovat, že výkon clusteru závisí na kvalitě síťového spojení daleko menší měrou.



Obrázek 3. Síť Myrinet použitá v linuxovém clusteru umožňuje získání vyššího výkonu ve srovnání s 100MBit ethernetem použitým na Linuxu i na NT. Tento rozdíl se projevuje při počtu procesorů větším, než čtyři. Zatímco v případě Linuxu a NT s přibývajícími procesory výkon clusteru klesá, síť Myrinet umožňuje nárůst výkonu až do šestnácti procesorů. To je způsobeno především snížením času potřebného pro meziprocesorovou komunikaci (latence).

4.4.4 Použití Linuxu nebo Win-NT pro clustery s procesory Pentium Z výsledků testů, které jsou zde uvedeny, je patrno, že Linux je velice dobře použitelný operační systém pro nasazení ve výpočetních systémech zajišťujících běh výpočetního softwaru FLUENT. Výsledky ukazují, že Linux je dobře srovnatelný s platformou NT až do velikosti clusteru 16 procesorů. Clustery založené na Win-NT jsou mírně rychlejší pro malé úlohy, což může být způsobeno kvalitněji zkompilovaným FLUENTem pro tuto platformu. Pro větší počet procesorů se ukazuje Linux výhodnější z hlediska kvalitnější síťové komunikace s menší režií systému. Pro menší *levné* clustery založené na počítačích s procesory Intel do velikosti osm procesorů, není volba operačního systému (Linux nebo Win-NT) otázkou výkonu clusteru, ale spíše vedlejších faktorů – jako je stabilita, dostupnost technické podpory, cena, dostupnost aplikací pro daný operační systém atd. Z těchto hledisek se Linux jeví jako výhodnější řešení v případě, že v clusteru nehodláme



Obrázek 4. Srovnání síťě Myrinet s nízkou latencí a klasického 100MBit ethernetu na *velké* úloze ukazuje, že meziprocesorová komunikace není limitujícím faktorem, pokud je výpočetní zátěž uzlu dostatečně velká. Je zde patrno, že pro šestnáct procesorů neznamená použití sítě Myrinet nijak významné navýšení výkonu. Je dobré zmínit, že osmiprocesorový cluster byl v tomto případě minimální konfigurací na které bylo možno řešenou úlohu spustit.

používat aplikace, které nejsou na tento operační systém portovány. Závěrem je vhodné podotknout, že pokud zvolíme *levné* řešení, musíme zvážit, zdali je dostatečné z hlediska výkonu a spolehlivosti. V případě, že některé z těchto kritérií není uspokojivě splněno, je třeba začít uvažovat o robustním (a samozřejmě dražším) řešení, které představují clustery víceprocesorových pracovních stanic s komerčními UNIXovými operačními systémy.

Reference

1. H.A. Barnes, J.F. Hutton, a K. Walters. *An Introduction to Rheology*. Elsevier, 1989.
2. Developers. Cluster computing for cfd. Technical report, 1999.
3. J. Kňourek a J. Linhart. Distribuované výpočty v prostředí metacentra. Sborník 5. uživatelské konference FLUENT, 1999.
4. P. Střasák. Co znamená cfd. *Chip*, (9):38–41, 1998.

Aplikační server s XML-RPC rozhraním

Alois Vitásek

Email: alois.vitasek@mail.cz

Abstrakt: XML-RPC je jednoduchý protokol, kterým se přenášejí požadavky na volání funkcí a jejich návratové hodnoty.

1 Vlastnosti

Protokol XML-RPC (`http://www.xmlrpc.org/`) slouží k jednoduchému a přenositelnému vzdálenému volání funkcí. Jedná se o klient server uspořádání. Klient volá funkce, které server poskytuje. Ke komunikaci je použit HTTP protokol. Přenášená data jsou zapisována pomocí XML.

Specifikace protokolu je poměrně jednoduchá. Navíc rozhraní pro HTTP protokol a XML parser jsou v mnoha jazycích již „hotová“. Díky tomu existují implementace v různých programovacích jazycích (Perl, Java, Python, C, C++, PHP, ...).

Při praktickém použití není programátor nucen tvořit či parsovat XML data. Specifikuje (většinou pomocí nějakých objektů) server, jméno funkce, případné parametry a odešle požadavek. Návratovou hodnotu obdrží v nějaké datové struktuře. Detaily záleží na konkrétní implementaci a možnostech programovacího jazyka. V případě serveru ve většině implementací stačí napsat jednotlivé funkce a říci, při jakém požadavku se mají volat.

2 Popis protokolu a jeho možnosti

Podrobná specifikace protokolu je na adrese `http://www.xmlrpc.org/spec`.

Žádosti o vykonání funkce jsou posílány v HTTP-POST requestu. Tělo requestu obsahuje XML data, která určují volanou funkci a její parametry. Po vykonání funkce na serveru je vrácena návratová hodnota v těle HTTP response (zapsaná pomocí XML struktury). V případě neúspěšného volání (funkce neexistuje, vykonávání zhavarovalo, ...) obsahuje odpověď chybový kód a text.

2.1 Volání funkce

Příklad:

```
POST /RPC2 HTTP/1.0
Content-Type: text/xml
```

Jan Kasprzak, Petr Sojka (editoři): SLT 2002 – sborník semináře o Linuxu a TEXu, str. 249–255, 2002.

```
Content-length: 181

<?xml version="1.0"?>
<methodCall>
   <methodName>ukazka.jmeno_funkce</methodName>
   <params>
      <param>
         <value><i4>42</i4></value>
      </param>
   </params>
</methodCall>
```

V těle requestu je specifikováno jméno volané funkce (pomocí elementu `<methodName>`) a případné parametry (element `<params>` s podelementy `<param>`). Parametrů může být více.

2.2 Návratová hodnota

V případě úspěšného volání vrátí server výsledek výpočtu. Pokud nastane při zpracování požadavku chyba, vrátí chybovou hodnotu.

Příklad odpovědi po úspěšném volání funkce:

```
HTTP/1.1 200 OK
Content-Length: 158
Content-Type: text/xml

<?xml version="1.0"?>
<methodResponse>
   <params>
      <param>
         <value><string>Data z funkce</string></value>
      </param>
   </params>
</methodResponse>
```

V kořenovém elementu `<methodResponse>` je jeden element `<params>`, který obsahuje jeden element `<param>`. Komplikovanější návratovou hodnotu lze realizovat pomocí datového typu struktura nebo pole.

V případě chyby obsahuje element `<methodResponse>` element `fault`, ve kterém je struktura s klíči `faultCode` a `faultString`.

Příklad návratu chyby:

```
HTTP/1.1 200 OK
Content-Length: 426
Content-Type: text/xml
```

```
<?xml version="1.0"?>
<methodResponse>
   <fault>
      <value>
         <struct>
            <member>
               <name>faultCode</name>
               <value><int>4</int></value>
            </member>
            <member>
               <name>faultString</name>
               <value>
          <string>Too many parameters.</string>
       </value>
            </member>
         </struct>
      </value>
   </fault>
</methodResponse>
```

2.3 Datové typy

Skalární hodnota Skalární hodnoty jsou uzavřeny značkou `<value>`. Datový typ je určen další vnořenou značkou. Možné datové typy:

značka	datový typ	příklad
`<int>` nebo `<i4>`	celé číslo	−56
`<boolean>`	boolean, 0/1	0
`<double>`	desetinné číslo	−17.454
`<string>`	ASCII string	abcdef
`<date>`	datum/čas	2002-10-26T12:59+0200
`<base64>`	base64 – zakódovaná binární hodnota	

Pokud není určen datový typ, předpokládá se `<string>`.

Přenášená data musejí být validními XML dokumenty, což mimo jiné znamená, že znaky v řetězcích musejí být ve správném kódování (UTF-8). Pokud to z nějakých důvodů nelze zajistit, lze použité kódování zapsat v XML deklaraci atributem `encoding`. Není ovšem zaručeno, že zvolené kódování bude podporováno každým klientem/serverem a většinou bývá navíc nutná „postranní“ dohoda.

Struktura Pro zápis struktur (*hash*, *asociativní pole*, ...) se používá značka `<struct>`. Ta obsahuje kontejnery `<member>`, z nichž každý obsahuje značky `<name>` a `<value>`.

Příklad struktury obsahující tři položky s klíči `jmeno`, `prijmeni` a `vek`:

```
<struct>
   <member>
       <name>jmeno</name>
       <value><string>Josef</string></value>
   </member>
   <member>
       <name>prijmeni</name>
       <value><string>Novák</string></value>
   </member>
   <member>
       <name>vek</name>
       <value><int>36</int></value>
   </member>
</struct>
```

Pole Elemet `<array>` obsahuje jeden element `<data>`. Ten obsahuje libovolný počet elementů `<value>`, které mohou být různých typů.

Příklad pole:

```
<array>
   <data>
      <value><string>hello</string></value>
      <value><boolean>0</boolean></value>
      <value><i4>56</i4></value>
   </data>
</array>
```

Datové typy mohou být různě kombinovány. Můžeme mít např. pole struktur. Rekurzivní zanoření je možné (struktury obsahující pole se strukturami a pod.). Takto lze například v jednom parametru předat popis strukturovaného dokumentu.

3 Příklady použití

3.1 PHP

Jednoduchým způsobem, jak si XML-RPC vyzkoušet, je použít jeho PHP implementaci. Zvlášť server se píše snadno, neboť „serverovou infrastrukturu“ může dělat *apache*. V následujících příkladech jsou použity soubory `xmlrpc.inc` a `xmlrpcs.inc` z projektu *XML-RPC for PHP* (`http://sourceforge.net/projects/phpxmlrpc/`).

PHP – klient

```
<?php
include 'xmlrpc.inc';
```

```
// kde volat funkce - objekt server
$server  = new xmlrpc_client('/cesta/k/serveru.php',
                             'hostname', 80);

// zprava - co volat + parametry
$message = new xmlrpcmsg('test.soucet', array(
    new xmlrpcval(38, 'int'), new xmlrpcval(4, 'int')));

$result = $server->send($message);        // poslat request

// zpracovani odpovedi
if (!$result) {
    print "nelze se spojit s serverem";
} elseif ($result->faultCode()) {
    print "<p>XML-RPC chyba #" . $result->faultCode() . ": " .
    $result->faultString();
} else {
    $struct = $result->value();
    $sumval = $struct->structmem('soucet');
    $soucet = $sumval->scalarval();
    print "soucet: $soucet";
};
?>
```

PHP – server

```
<?php
include 'xmlrpc.inc';
include 'xmlrpcs.inc';   // knihovna pro server

function soucet ($params) {

    // vyndat vstupni promenne
    $xval = $params->getParam(0);
    $x = $xval->scalarval();
    $yval = $params->getParam(1);
    $y = $yval->scalarval();

    // vypocet
    $soucet = $x + y;

    // sestavit odpoved
    $struct = array('soucet' => new xmlrpcval($soucet, 'int'));
    return new xmlrpcresp(new xmlrpcval($struct, 'struct'));
}
```

```
new xmlrpc_server(array('test.soucet' =>
                        array('function' => 'soucet')));
?>
```

3.2 Perl

Klient pro výše uvedené příklady v Perlu:

```
use Frontier::Client;

# server kde volat
$server_url = 'http://host/cesta/k/serveru.php';
$server = Frontier::Client->new(url => $server_url);

# volani funkce
$result = $server->call('test.soucet', 39, 3);

# vytazeni vysledku
$soucet = $result->{'soucet'};

print "soucet: $soucet\n";
```

Použitý modul `Frontier::Client` lze získat na CPANu (`http://www.cpan.org`).

4 Bezpečnost

Protokol XML-RPC neobsahuje žádné prvky, kterými by šla provádět autorizace volání funkcí nebo šifrování přenášených dat. Je to podobná situace jako v případě zabezpečení aplikace s www rozhraním.

V úvahu připadá zabezpečení na úrovni HTTP protokolu (SSL, jméno/heslo v requestu, ...). Nebo na úrovni volaných funkcí. Například přidáním nějakého session parametru, kontrolních součtů atp. do parametrů funkce, volaného URL, hlavičky HTTP requestu, apod.

5 Zkušenosti s praktickým použitím

Aplikační server s XML-RPC rozhraním je používán pro generování a distribuci dokladů (výzvy k platbě, faktury, apod.). Server je napsán v Perlu, klienti v php.

Klient získá informace o obsahu dokumentu (z databází) a odešle je na server. Zde je pomocí šablon vytvořen vstup pro TEX a vygenerován dokument ve formátu pdf. Dokument je uložen do archivu.

Již existující dokumenty mohou být zobrazeny klientem (dokument ve formátu pdf je přenášen datovým typem base64) nebo odeslány elektronickou poštou.

Za dobu ostrého provozu (cca 5 měsíců) bylo takto vygenerováno více jak 35 000 pdf dokumentů.

Reference

1. XML-RPC – `http://www.xmlrpc.org`.
2. specifikace protokolu – `http://www.xmlrpc.org/spec`.
3. XML-RPC HOWTO – `http://xmlrpc-c.sourceforge.net/xmlrpc-howto/xmlrpc-howto.html`.
4. XML-RPC for PHP – `http://sourceforge.net/projects/phpxmlrpc/`.

dbMan – modulární SQL konzole

Milan Šorm

Ústav informatiky
Provozně ekonomická fakulta
Mendelova zemědělská a lesnická univerzita v Brně
Email: sorm@pef.mendelu.cz

Abstrakt: Snahou všech datamanažerů (ať již DBA, vývojářů či datamanažerských „lopat“) je najít nástroj, který bude minimálně překážet při výkonu každodenních starostí a přitom poskytne maximální komfort v možnostech a případném dalším individuálním rozšiřování. Tyto tendence mne vedly k návrhu a vytvoření open source SQL konzole, která je striktně modulární a snaží se splnit výše uvedené podmínky. Tento příspěvek je o návrhu tohoto nástroje, o úskalích, které Vás mohou potkat a o metodách, které Vám mohou pomoci při vývoji podobných modulárních nástrojů pro jiné oblasti. Současně příspěvek představí možnosti této (zatím jen) textové konzole a ukáže, kudy se ubírá její další vývoj.

Klíčová slova: dbMan, SQL konzole, modularita, Perl

1 Úvod

Pracuji s databázovými systémy už řadu let, ale teprve před čtyřmi roky jsem je začal využívat jako centrální úložiště všech dat. Od roku 1998 vyvíjíme na naší univerzitě informační systém, který si klade velmi ambiciózní cíle – a jedním z nich je být centrálním datovým skladem univerzity. To ovšem vyžaduje vznik pro univerzitu zcela nové profese – datamanažerů. Jako první datamanažer nového informačního systému naší univerzity jsem na jaře roku 1998 hledal nástroj, který by splňoval všechny mé požadavky – především jednoduchou obsluhu, mnoho nástrojů pro odstranění každodenní dřiny, rozsáhlou modularitu pro možnost doplňovat si nové funkce a jednoduché (případně modulární) rozhraní, aby bylo možné s nástrojem pracovat i na pomalých českých telefonních linkách.

Bohužel většina open source i komerčních SQL konzolí v té době byla určena vždy pro jeden konkrétní databázový systém a neposkytovala komfort, který jsem od nástroje očekával. Zvolil jsem proto vlastní cestu (a od té doby velmi často volím vlastní cestu) a vyvinul si jednoduchou SQL konzoli v Perlu využívající DBI/DBD rozhraní, které umožňuje práci s celou řadou databázových systémů.

Tento nástroj jsem vyvíjel zhruba jeden rok, než se z něj stal použitelný nástroj splňující některé z uvedených požadavků – především obsahoval řadu

Jan Kasprzak, Petr Sojka (editoři): SLT 2002 – sborník semináře o Linuxu a TEXu, str. 257–264, 2002.

nadstandardních funkcí a zárodek modularity. Až do roku 2002 zůstal nástroj stabilní (s minimální opravou chyb) a byl dostupný v podobě samostatného balíčku i přes vyhledávač `freshmeat.net`. Na jaře roku 2002 jsem se rozhodl pro kompletní přepis původního částečně monolitického balíku do plně modulárního nástroje, který bude mít jedinou filozofii „vše bude nahraditelné za pomoci modularity“. Tak vznikl dnešní dbMan, který jako rozšiřovací modul implementuje i příkaz `QUIT`.

2 Instalace nástroje

Současný dbMan je nalezitelný na síti CPAN (Comprehensive Perl Authors Network) a lze jej jednoduše instalovat pomocí cpan shellu. Pro instalaci použijte následující postup:

```
perl -MCPAN -e shell
cpan> install DBIx::dbMan
```

Pro správnou funkčnost vyžaduje dbMan přítomnost interpretu jazyka Perl (nezáleží na operačním systému, já provozuji dbMan pod Linuxem, Solarisem i MS Windows) a moduly `DBI`, `Data::ShowTable`, `Text::FormatTable`, `Text::CSV`, `Term::Size`, `Term::ReadLine` a samozřejmě příslušné ovladače `DBD` k vaší databázi. Pro provoz alternativního grafického rozhraní (které je silně ve vývoji) je nutná distribuce `Perl/Tk`. Přímé závislosti nainstaluje cpan shell za Vás při instalaci dbManu, na instalaci ostatních modulů použijete s výhodou opět cpan shell.

Součástí instalované distribuce je nejen vlastní program (ve jmenném prostoru `DBIx::dbMan::`), ale také jednoduchá dokumentace popisující jak začít dbMan rozšiřovat a asi 70 základních rozšíření poskytující standardní funkčnost dbMana (z těchto jednoduchých kódů je možné vyjít při pokusu o rozšíření). Rovněž je přítomen patch pro `Term::ReadLine::Gnu`, který opravuje chybu znemožňující používat TAB kompletaci bez rozlišení velikosti písmen.

3 Možnosti SQL konzole

SQL konzole je nástroj pro přímou komunikaci s databázovým systémem. Umožňuje zasílat do DBS příkazy a rozumně zobrazovat odpovědi serveru. Tyto zasílané příkazy lze rozlišit na dvě velké skupiny – dotazy (vrací tabulku u relačních databázových systémů) a úkoly (vrací informaci o splnění úkolu, příp. počet záznamů u relačních databázových systémů, které úkol ovlivnil).

I při takto jednoduchém úkolu lze však nalézt řadu oblastí, kde nám může SQL konzole usnadnit život – ať již se jedná o historii odesílaných příkazů nebo použití tabelátoru pro dohledávání názvů tabulek, sloupců a dokončování příkazů.

dbMan rozděluje možnosti do dvou kategorií – možnosti rozhraní SQL konzole a možnosti vnitřního interpretu příkazů (vykonávání akcí uživatele). To

umožňuje oddělit formu vzniku akce (zadání příkazů v příkazovém prostředí, výběr z menu, kliknutí myši na ikoně) od její interpretace (provedení příkazu v dbManu, databázi, transformace akce na jinou akci apod.).

3.1 Rozhraní konzole

V současně době existují dvě rozhraní pro práci s dbManem. Rozhraní `cmdline` je standardním, plně funkčním, denně používaným rozhraním, jehož filozofií je minimalizace přenášených dat mezi serverem, kde běží dbMan a terminálem, kde pracuje uživatel (ideální pro pomalé telefonní linky, u nás však díky zvyku používané i na Gigabitové páteřní síti univerzity). Toto rozhraní nabízí práci v terminálovém režimu, kdy na výzvu (prompt) načte uživatelův příkaz a odešle jej do jádra dbManu na zpracování jako akci `COMMAND`. Na druhou stranu nabízí jednoduchou metodu pro výstup informací na terminál (`print()`), jenž může být později použita rozšířením realizující výstup. Pokud rozhraní nalezne v systému knihovnu `Term::ReadLine`, použije její vlastnosti pro načítání a úpravy příkazového řádku, její vlastnosti pro práci s historií příkazů (které doplní o možnost transparentní historie ukládáním historie i do souboru) a její vlastnosti pro tabelátorovou kompletaci (doplňování slov při stisku klávesy TAB). Vlastní implementace tabelátorové kompletace je však otázkou rozšiřujících modulů. Příkaz `dbman` nastartuje dbMan v tomto rozhraní.

Druhým rozhraním je experimentální rozhraní `tkgui`, které ukazuje sílu rozšiřujících možností dbManu zavedením grafické konzole v podobě okna s příkazovým řádkem a výstupním prostorem. Toto rozhraní (stejně jako předchozí) implementuje načítání příkazů a výstup výsledku, ovšem tentokrát je výstup směřován do okna aplikace užívající `Perl/Tk`. Rozhraní implementuje vlastní ovládání historie příkazů kompatibilní s předchozím rozhraním, nepodporuje však zatím TAB kompletaci. Všechny možnosti dbManu pracují v obou rozhraních, protože se liší pouze forma vzniku akcí a zobrazování výstupů. Toto rozhraní získáme příkazem `xdbman`.

3.2 Interpret akcí

Interpret akcí zajišťuje předávání vzniklých akcí do jednotlivých rozšiřujících modulů a tím zajišťuje vlastní funkčnost dbMana. Pomocí těchto modulů realizuje dbMan veškerou funkčnost – počínaje reakcí na příkaz `QUIT`, předávání SQL příkazů do databáze, implementaci TAB kompletace, práci se schránkou, práci s historií příkazů apod.

Z funkcí dbManu, které datamanažeři pracující na naší univerzitě nejvíce chválí, bych vybral především možnost paralelního spojení s více databázovými systémy (pomocí příkazu `CREATE CONNECTION` můžeme zakládat další spojení na různé databázové systémy, pomocí příkazu `USE` pak mezi těmito spojeními přepínáme). Pokud spojíme další pěknou vlastnost – práci se schránkou SQL výstupů (clipboardem) – s vícenásobnými spojeními, dostaneme jednoduchý nástroj pro datamanažerské pumpování dat mezi různými systémy (v jedné databázi – např. v DBF souboru – provedeme pomocí `\copy` vybrání nějakého

výstupu SQL dotazu, v druhé databázi – např. v Oracle – pomocí `\paste` výstup jednoduše zapíšeme do požadované tabulky). Při použití `\unioncopy` pak můžeme nasbírat i více výstupů z různých zdrojů a výsledek poté zapsat do jiné tabulky (např. CSV souboru nebo do tabulky databáze na MS SQL serveru).

Samozřejmá je podpora transakcí (transakce v jednom spojení neovlivňují ostatní spojení), výpočtu doby pro zpracování příkazu (jednoduchý benchmark), na databázi Oracle zobrazení stromu `EXPLAIN PLAN`, zobrazení seznamu objektů v databázi nebo popisu (`DESCRIBE`) tabulek. Víceřádkové SQL dotazy, možnost stránkování výstupu programem `less`, ukládání výstupu do souboru či interpretace `.sql` souborů je už dnes považována za samozřejmost. Datamanažeři oceňují i jednoduchý vstupní CSV filtr, který na základě popisu oddělovačů v textovém souboru provede přenos tohoto souboru do databáze (podobnou věc lze realizovat pomocí schránky, ale potřebujeme ovladač `DBD::CSV` pro práci s takovými soubory, což jednoduchý filtr `\csvin` nepotřebuje).

Naši datamanažeři často editují v databázi Oracle objekty PL/SQL kódu (procedury, funkce, triggery, package), k čemuž dosud využívali především DBA studio (standardní Oracle GUI nástroj napsaný v Javě). Tento nástroj je nepoužitelný nejenom na telefonní lince, ale často i na naší rychlé univerzitní síti. Proto jsem dbMan rozšířil o možnost (zatím jen pro Oracle) přímé editace takovýchto objektů – ve variantním editoru pak můžete dosáhnout dalších schopností, např. při použití editoru `vim` získáte zdarma barevnou syntaxi PL/SQL, což neumí ani DBA studio).

Poslední vítané vlastnosti dbManu, kterou si zde představíme, je podpora různých výstupních formátů a podpora TAB kompletace. První vlastnost umožňuje pomocí rozšíření definovat výstupní formát SQL dotazů (tabulka, CSV, HTML, speciální `records` formát pro malé dotazy apod.), kdy lze definovat odlišný formát pro jednořádkové výstupní tabulky a ostatní dotazy. TAB kompletace je plně řízená pomocí rozšíření, takže přidání rozšíření nejen doplní on-line nápovědu programu, ale také doplňování příkazů o nové možnosti. TAB kompletace také implementuje doplňování názvů tabulek, pohledů, sekvencí a sloupců v tabulkách. Pro Oracle navíc obsahuje řadu podpůrných funkcí pro urychlení TAB kompletace užitím systémového katalogu Oracle.

4 Architektura dbManu

Vnitřní architektura dbManu se snaží dodržet striktně modulární koncepci. Základem je tzv. dbMan core (tedy jádro programu), které obsahuje tzv. candidate select algoritmus pro výběr, která rozšíření jsou v dosahu (a vybere nejvhodnější a nejnovější verze) a algoritmus pro zpracování akcí (základní smyčku získej akci – zpracuj akci obohacenou o provádění akcí v rozšířeních, jak bude popsáno níže). Toto jádro pracuje s celou řadou dalších modulů, především s dbMan interfaces moduly (rozhraní), které mohou implementovat odlišné vizuální rozhraní dbManu (jako již zmiňované `cmdline` či `tkgui`) a s dbMan DBI modulem, který zapouzdřuje práci s DBI/DBD tak, aby fungovalo paralelní spojení s více databázovými systémy. Pro celý dbMan i všechna rozšíření

se dbMan DBI chová jako standardní DBI obohacené o funkce pro definice, zavádění, rušení a přepínání jednotlivých spojení.

Nejzajímavější částí, se kterou dbMan core spolupracuje, jsou dbMan extensions moduly – což jsou jednotlivé definice rozšíření. Tyto moduly samozřejmě využívají dbMan interfaces, dbMan DBI i samotné jádro. Aby mohla jednotlivá rozšíření spolupracovat, využívají dbMan MemPool modul, který obsahuje společný datový sklad pro všechna rozšíření. Dva pomocné moduly – dbMan history a dbMan config jsou určeny pro práci s historií příkazů v souboru (transparentnost historie při různých spuštěních dbMana) a pro práci s konfiguračním souborem (čímž se pro rozšíření nabízí možnost získávat informace o svém provozu ze společné konfigurace programu dbMan).

Nejzajímavější částí architektury je mechanismus zpracování událostí a předávání akcí. Tento mechanismus (implementovaný v dbMan core) pracuje na principu následujícího algoritmu:

1. Zjištění události pomocí rozšíření.
2. Vytvoření akce (speciální asociativní pole pro akce).
3. Výběr rozšíření s maximální prioritou.
4. Předání vytvořené akce do vybraného rozšíření.
5. Rozšíření se rozhodne, zda jde o akci určenou pro něj.
6. Pokud není akce určená pro něj, vybere se další rozšíření v pořadí a přejde se k bodu 4.
7. Pokud je akce určená pro něj, vykoná se příslušná akce.
8. Dále se transformuje asociativní pole akce, typ akce a dojde k nastavení speciálního příznaku ovlivňujícího další chod zpracování akce.
9. Pokud je příznak nastaven, přejde se s upravenou akcí do bodu 3.
10. Pokud není příznak nastaven, pokračuje se výběrem další akce podle priorit a přejde se k bodu 4.
11. Celý mechanismus končí v okamžiku, kdy žádné rozšíření již nechce na akci reagovat.
12. Pokud zbylá akce je akcí `QUIT`, ukončí se celá hlavní smyčka, jinak se přejde k bodu 1.

Tento algoritmus umožňuje zpracování akcí v rozšířeních a předávání si informací mezi jednotlivými rozšířeními. Každé rozšíření také může ovlivnit, zda se má transformovaná akce znovu spustit přes všechna rozšíření, nebo se bude pokračovat dál podle pořadí priorit. Každé rozšíření také samo definuje, s jakou prioritou mu má být akce předávána (a tím ovlivňuje okamžik, kdy dostane akci ke zpracování – nová implementace dané funkce pak může mít vyšší prioritu než původní modul a tak mít možnost akci ovlivnit o něco dříve).

Ukázku toho, jak komplexní může být zpracování jednoho příkazu `SELECT` dává následující posloupnost zpracování akcí a rozšíření, které na ní reagují (přepis není úplný, vypisuji zde pouze podstatné akce a jejich úkol).

Nejprve vzniká akce `COMMAND`, která obsahuje informaci o tom, jaký příkaz byl uživatelem zadán. Pokud vynecháme preprocesor příkazů doplňující či odstraňující koncový středník podle vlastností databázového systému a ovlivňující formát výstupu, dospěje akce do rozšíření `CmdStandardSQL`, které rozpozná SQL dotaz a převede akci na akci `SQL`, která navíc obsahuje informaci

o tom, zda se jedná o dotaz či úkol. Tato akce dále postupuje posloupností priorit, až dospěje do rozšíření `StandardSQL`, které provede příslušnou operaci pomocí dbMan DBI a neformátovaný a neupravovaný výstup přepíše do akce `SQL_RESULT`. Tato akce dospěje do `SQLShowResult`, kde je podle typu SQL proveden buď výstup informace o počtu ovlivněných řádků, nebo transformace na akci `SQL_OUTPUT`, která postupuje přes řadu modulů provádějících transformace výsledku (`NULL` hodnoty, nezobrazitelné znaky, rozhodování o formátování) až do rozšíření `SQLOutputTable`, což je jedno z rozšíření realizujících konkrétní výstupní formát (takových rozšíření je celá řada). Toto rozšíření přeformátuje tabulku pomocí `Text::FormatTable` a produkuje akci `OUTPUT`, která obsahuje informaci o tom, co má být součástí výstupu. Tato akce může být provedena řadou rozšíření realizujících stránkování, uložení výstupu do souboru apod., ale pravděpodobně dospěje do rozšíření `Output`, které využije dbMan interfaces pro zobrazení výsledku a transformuje akci na `NONE`. O tuto akci už nikdo nemá zájem (ani rozšíření `Fallback`, které zobrazuje nápisy jako `Unknown command.` či `INTERNAL ERROR: Action not handled.`) a tak je cyklus zpracování události ukončen.

5 Jak dbMana rozšiřovat

Rozšíření se píší v jazyce Perl jako standardní moduly ve jmenném prostoru `DBIx::dbMan::Extension::`, přičemž tyto moduly se umísťují buď do aktuální adresáře, nebo tam, kam míří `extensions_dir` direktiva v konfiguračním souboru. Tento modul realizujeme jako potomka modulu `DBIx::dbMan::Extension` a musíme přetížit minimálně funkci `IDENTIFICATION()`, pomocí které se rozlišuje, která verze rozšíření je nejnovější, a které moduly implementují stejné funkce. Jedná se o trojici šesticiferných údajů, přičemž první část udává autora, druhá část modul v rámci autora a poslední verzi (např. `999999-000001-000005` představuje pátou verzi prvního experimentálního modulu – protože tento autor je definován jako experimentální).

Aby rozšíření bylo k něčemu dobré, je vhodné přetížit také funkci `handle_action()` realizující vlastní výkonný kód a funkci `preference()`, která definuje prioritu rozšíření pro seznam rozšíření (jak brzo dostane modul akci k provádění). Preference rozdělujeme do několika kategorií – standardně je preference 0, ale fallback a output rozšíření mají preferenci zápornou. Výkonné moduly (provádění SQL apod.) mají preferenci pod 1000, parsery jednotlivých příkazů pak pod 2000. Preprocesory (přepínače apod.) preferenci pod 3000. Makroprocesory (druhá abstrakce nad rozšířeními) pak ještě vyšší priority – až po 4000. Nad 4000 jsou preference tzv. URGENT rozšíření, např. odstranění počátečních a koncových mezer celého příkazu v akci `COMMAND` apod.

Základní kostra rozšíření je následující:

```
package DBIx::dbMan::Extension::Název;
use strict;
use vars qw/$VERSION @ISA/;
```

```
use DBIx::dbMan::Extension;

$VERSION = '0.01';  @ISA = qw/DBIx::dbMan::Extension/;
1;

sub IDENTIFICATION { return "999999-000001-000001"; }

sub preference { return 50; }

sub handle_action {
        my ($obj,%action) = @_;
        # %action modification or something making
        $action{processed} = 1;  return %action;
}
```

Úkolem `handle_action()` je zpracovat `%action` (předeším typ akce je v `$action{action}`) a toto asociativní pole modifikovat a nastavit (nebo nenastavit) příznak `$action{processed}`. Vše odpovídá výše popsanému algoritmu. Je velmi vhodné směřovat všechny výstupy (i chybové) do akce `OUTPUT`, protože jedině tímto způsobem dosáhneme nezávislosti na rozhraní. Další konvencí je akce `NONE`, na kterou by nikdo neměl reagovat. Pro sledování průchodu rozšířeními (ladění předávání akcí) lze využít trasovacího režimu (`SET TRACING ON`).

Uvnitř rozšíření lze použít pět základních objektů pomocí `$obj->{-název}`, kde objekt `-interface` zpřístupňuje aktuální rozšíření, objekt `-dbi` rozhrání dbMan DBI, objekt `-mempool` sdílený datový sklad všech rozšíření (např. předávání informace o tom, jaké výstupní formáty existují apod.), objekt `-config`, který zpřístupňuje konfigurační soubor a objekt `-core`, kterým lze zpřístupnit vlastní jádro dbMan core a provádět tak (velmi nebezpečné) akce. O jeho využití se zmíním v závěru příspěvku.

Je vidět, že pro práci s daty používá dbMan celkem tři metody. V asociativním poli `%action` jsou uložena data potřebná pro vyřízení jedné akce, v objektu `$obj->{-mempool}` jsou uchovávána data do ukončení dbMana. Persistenci zajišťuje objekt `-config`.

Další funkce, které mohou být v rozšíření definovány, jsou například `for_version()`, kterou lze ovlivnit, které dbMan core podporuje tento modul (kdyby došlo k zásadní změně API, což se zatím nestalo), `known_actions()`, které stanovuje, na jaké akce rozšíření reaguje a tak může zjednodušit a urychlit práci vlastní smyšky událostí a zjednodušit výstupy trasování (nepovinné). Metody `init()` a `done()` slouží pro provedení akcí při zavádění a odzavádění rozšíření (např. alokace promptu, definice údajů do datového skladu rozšíření apod.). Dvě další metody – `cmdhelp()` a `cmdcomplete()` jsou určeny pro definici nápovědných textů do centrální nápovědy a pro definici TAB kompletace konkrétního modulu (pro rozšíření realizující TAB kompletaci).

Objekt rozhraní umožňuje alokaci a dealokaci promptu (modifikace standardního `SQL:` na např. `LONG SQL:`) a funkce pro reálný výstup a práci s historií příkazů. Mempool nabízí funkce pro ukládání a záskávání hodnot a plnění či

rušení tzv. registrů (což jsou vektory údajů, např. seznam možných výstupních formátů).

6 Další možnosti modularity

Mimo uváděné definice rozšíření, můžeme přetěžovat interface objekty (pro definice nových rozhraní) – zde se očekává vyvinutí dalšího rozhraní pro práci s knihovnami `slang/ncurses` a rozhraní pro `qtgui` a `gtkgui`. Chci také dokončit stávající `tkgui`.

Existuje i možnost zasílání patchů na vlastní jádro dbMana přímo mě (a já rozhodnu, zda se budou do dbMana aplikovat), přidávání Vašich rozšíření či rozhraní do základní distribuce – nebo si můžete na CPANu publikovat vlastní rozšíření, přidat na `freshmeat.net` další vývojovou větev apod.

Perličkou na závěr může být fakt, že rozšíření implementují vše, tedy i vlastní zavádění a odzavádění jiných rozšíření (odzavedení rozšíření `Extensions` tedy může mít fatální následky).

7 Závěr

Dnes máme na univerzitě celé oddělení datamanagementu, které dbMana využívá. Já sám již datamanažerem nejsem (neb velikost informačního systému si vyžádala vznik velkého oddělení, které potřebuje „sekretářku“ shánějící peníze a podepisující faktury `:-)`), ale nadále rád věnuji dlouhé noci doplňování nových rozšíření do dbMana podle požadavků našich datamanažerů i uživatelů z celého světa.

Budu rád, když využijete dbMana nebo myšlenky, které byly použity při jeho tvorbě, ve svých vlastních projektech. Ostatně o tom je myšlenka open-source. dbMan je distribuován pod Perlovou licencí, což Vám umožňuje volbu mezi GNU GPL licencí nebo Artistic licencí.

Reference

1. Dokumentace informačního systému MZLU v Brně (`http://is.mendelu.cz/dok/`)
2. dbMan (`http://dbman.linux.cz/`)

Komplexní řešení pošty za použití OSS

Marcel Kolaja

soLNet, s. r. o.
Email: marcel@solnet.cz

Abstrakt: Je známo, že Open Source Software (OSS) není používán jen ve sféře akademické, ale úspěšně proniká i do komerční sféry. V přednášce si předvedeme komplexní řešení pošty pro komerční organizace za použití výhradně OSS. Probereme zejména možnosti sdílení poštovních schránek, kontaktů, replikaci účtů ve VPN a možnost přístupu uživatelů k poště z vnější sítě (Internetu) za použití webmailu (případně wapmailu).

Klíčová slova: pošta, sdílené poštovní schránky, sdílené kontakty, IMAP, LDAP, Open Source Software

1 Úvod

Elektronická pošta patří v dnešní době k základním komunikačním kanálům nejen v akademických, ale i v komerčních organizacích. Požadavky na služby spojené s elektronickou poštou se však v komerční sféře poněkud liší od požadavků ve sféře akademické. Pro požadavky akademické sféry existuje několik velmi spolehlivých řešení založených čistě na Open Source Software. Požadavkům komerční sféry vyhovuje několik řešení[1], jejichž ceny se většinou pohybují od několika set tisíc až po několik desítek miliónů korun. Řešení založených na Open Source Software je, zdá se, pramálo a jejich vlastnosti většinou nejsou v komerčních organizacích považovány za vyhovující. Tato situace pravděpodobně vychází z faktu, že Open Source Software většinou pochází z akademického prostředí a pohled vývojáře Open Source Software na řešení problémů je poněkud jiný než pohled manažera firmy o několika tisících zaměstnancích. My si však přesto nastíníme zcela konkrétní řešení za použití výhradně Open Source Software.

2 Specifika komerčního prostředí

Mezi vlastnosti poštovního systému, které jsou obvykle v komerčním prostředí vyhledávány, patří možnost sdílení poštovních schránek a kontaktů či vzdálený přístup k poště pomocí WWW rozhraní. Ve větší organizaci, jejíž pobočky jsou propojeny do virtuální privátní sítě, bývá vyžadována také centrální správa účtů. V poslední době je vítána možnost přístupu k poště pomocí wapmailu.

[1] Ponecháme stranou diskuzi nad kvalitou těchto řešení.

Jan Kasprzak, Petr Sojka (editoři): SLT 2002 – sborník semináře o Linuxu a TEXu, str. 265–271, 2002.

3 Rozdíly mezi řešeními na bázi OSS a komerčními řešeními

Komerční řešení bývají obvykle produkty typu „všechno v jednom“. Řešení na bázi Open Source Software se naopak většinou drží staré zásady operačního systému *UNIX* – každý program dělá svoji malou úlohu a dělá ji dobře. Takový přístup přináší do problematiky modularitu. Poštovní systém si složíme z několika menších subsystémů a nedostaneme se do situace, kdy budeme váhat, jaký systém zvolit, protože jeden bude mít bezpečný MTA a druhý např. stabilní IMAP server. Na druhou stranu je poněkud obtížnější celý systém sestavit tak, aby se navenek tvářil jednotně.

Výhodou řešení na bázi Open Source Software je skutečnost, která je obsažena přímo v jeho názvu – tedy otevřené zdrojové kódy. Vývojáři se mohou inspirovat z konkurenčních produktů. S otevřenými zdrojovými kódy také souvisí otevřené formáty. Při řešení založeném na Open Source Software je výměna systému jednodušší, protože přesně známe specifikaci používaných formátů (např. poštovních schránek). Nemůže se nám stát, že bude obtížné vyměnit systém, protože ten stávající má uloženy poštovní schránky v proprietárním formátu, který nedokážeme transformovat na formát jiného systému.

Cena komerčních produktů se obvykle odvíjí od počtu uživatelů a mnohdy se vyšplhá do vskutku závratných výšin. U oteřených systémů naopak za cenu produktu neplatíme nikdy.

4 Představení konkrétního řešení

Jak jsme se dozvěděli dříve, v řešeních na bázi Open Source Software je zvykem rozdělit velký systém na dílčí subsystémy. My si poštovní systém rozdělíme na MTA[2] (Mail Transfer Agent), IMAP(S) a/nebo POP3(S) server, IMAP proxy, LDAP server, antispamový subsystém, antivirový subsystém, WWW rozhraní pro poštu a kontakty, wapmail, adminstrační rozhraní a aplikaci pro synchronizaci stavu systému s požadovanou konfigurací v LDAP (např. vytváření a rušení poštovních schránek).

Základ našeho řešení bude tvořit SMTP server *Exim* [3], který je vyvíjen na univerzitě v Cambridge. Klientské stanice budou přistupovat k poště pomocí protokolu IMAP, případně pomocí jeho bezpečné varianty IMAPS. Jako IMAP server použijeme *Courier-IMAP* [2], který také podporuje SSL, a proto ho lze využít i jako IMAPS server. Alternativně lze pro výběr pošty klientskými stanicemi použít protokol POP3. Protokol IMAP je však mnohem lepší, a tak si řešení pomocí protokolu POP3 nebudeme popisovat. Exim i Courier-IMAP podporují poštovní schránky ve formátu Maildir a my tuto jejich vlastnost využijeme. Courier-IMAP pochází z dílny Sama Varshavchika. Na Exim i Courier-IMAP se vztahuje licence GNU GPL. Na správu uživatelských účtů a kontaktů použijeme dnes velmi populární protokol LDAP, přesněji řečeno jeho implementaci *OpenLDAP* [7]. O vývoj OpenLDAP se stará nadace OpenLDAP Foundation, která dává

[2] MTA bývá také někdy označován jako SMTP server.

tento software k dispozici pod licencí OpenLDAP Public License. WWW rozhraní pro přístup k poště bude tvořit webový poštovní klient *IMP* [4]. Jeho autoři (Chuck Hagenbuch, Jon Parise, Ivan E. Moore II a Mike Hardy) se rozhodli uvolnit IMP pod licencí GNU GPL. Jako WWW rozhraní pro přístup ke kontaktům použijeme webový manažer *Turba* [12]. Autory manažera Turba jsou dva z vývojářů klienta IMP – Chuck Hagenbuch a Jon Parise. Na manažer Turba se vztahuje licence Horde Apache-like License. Turba a IMP jsou vystavěny nad systémem *Horde* [11]. Vývojový tým systému Horde tvoří Chuck Hagenbuch, Jon Parise, Anil Madhavapeddy, Rich Lafferty, Jan Schneider a Brent J. Nordquist. Systém Horde je k dispozici pod licencí GNU LGPL. Uvedeným webovým aplikacím umožní běh velmi oblíbený WWW server *Apache* [10]. O vývoj WWW serveru se stará nadace Apache Software Foundation, která uvolnila Apache pod licencí Apache Software License. Jako obranu proti spamu [14] můžeme použít např. poštovní filtr *SpamAssassin* [9]. Vývojový tým (Justin Mason, Craig Hughes, Matt Sergeant, Dan Quinlan, Malte S. Stretz, Theo Van Dinter a Duncan Findlay) poskytl SpamAssassin pod licencí Artistic License. Můžeme také využít podporu *RBL* [6] (Realtime Blackhole List) přímo v MTA Exim. Přístup k poště pomocí protokolu WAP nám může zajistit *Jwapmail* [5], jehož autorem je Nasir Simbolon. Na Jwapmail se vztahuje licence GNU GPL.

5 Bližší představení vybraných subsystémů

Abychom si jednotlivé subsystémy přizpůsobili našim představám, musíme je samozřejmě správně nakonfigurovat. Navíc jsou některé subsystémy více než zajímavé. Proto si některé z nich představíme trochu blíže, případně uvedeme příklady konfigurace.

5.1 Exim

Exim je kvalitní MTA, který má spoustu užitečných vlastností, z nichž některé využijeme. Zejména se jedná o možnost používání poštovních schránek ve formátu Maildir a podporu LDAP. Mnoho myšlenek je převzato z MTA *Smail 3*, ale jeho kód, který je napsán v *ANSI C*, je kompletně nový. Může běžet buď jako démon, nebo se spouštět z `inetd`.

Když Exim přijme zprávu, zapíše dva soubory do své fronty. Typicky je to adresář `/var/spool/exim/input/`. Jeden obsahuje informace z obálky, současný stav zprávy a hlavičky. V druhém souboru se nachází tělo zprávy. Stav zprávy obsahuje seznam adresátů, kteří již zprávu obdrželi. Jakékoli přepisování adres, specifikované v konfiguraci, je provedeno při přijetí zprávy. Každé zprávě je přiřazena identifikace (message id), která je dlouhá šestnáct znaků. Ta je rozdělena na tři části oddělené pomlčkami. Každá část je sekvence písmen a číslic. Prvních šest znaků tvoří čas přijetí zprávy. Po první pomlčce následuje dalších šest znaků, které jsou shodné s ID procesu, který zprávu přijal. Poslední dva znaky (po druhé pomlčce) jsou použity pro zaručení jednoznačnosti identifikace. Jména souborů ve frontě se skládají z identifikace zprávy následované suffixem `-H` pro soubor

obsahující obálku a hlavičky a suffixem `-D` pro soubor s tělem zprávy. Zpráva zůstává ve frontě, dokud není kompletně doručena. V případě, že doručování nemůže dále pokračovat (např. zpráva nemůže být doručena ani adresátům ani vrácena odesílateli), je označena jako zmrazená (frozen) ve frontě a nejsou prováděny žádné další pokusy o doručení. Administrátor může takové zprávy rozmrazit nebo naopak zmrazit některé zprávy ručně. Pokud je nastavena volba `auto_thaw`, Exim se po určitém čase pokusí zprávy znovu doručit. Poté, co proběhne doručení zprávy, zapíše Exim informaci o doručení do systémového logu.

5.2 OpenLDAP

OpenLDAP obsahuje několik částí: *slapd* (LDAP server), *slurpd* (LDAP replication server), knihovny a různé nástroje. slapd je LDAP server, který dokáže běžet na mnoha různých platformách. Zajímavá vlastnost je zejména podpora replikace dat. Replikační schéma je typu single-master/multiple-slave. Nutno ale podotknout, že slapd obsahuje také experimantální podporu replikace typu multiple-master. slurpd je démon, který obstarává replikaci dat mezi LDAP servery. Je odpovědný za distribuci změn v master databázi na slave databáze. Stará se například o opakování pokusu o replikaci, pokud replikace z nějakého důvodu selže. Na promítnutí kofigurace v LDAP do současného stavu použijeme jednoduchou aplikaci, která se bude periodicky spouštět pomocí démona `cron`.

5.3 Apache, Horde, IMP, Turba

Pro zprovoznění WWW rozhraní potřebujeme samozřejmě upravit konifiguraci WWW serveru Apache. Relevantní část konfiguračního souboru může vypadat např. takto:

```
# DocumentRoot: The directory out of which you will serve your
# documents.
DocumentRoot /usr/share/horde2/imp/
# Added for HORDE2
Alias /horde2 /usr/share/horde2
<Directory /usr/share/horde2>
Options FollowSymLinks
AllowOverride None
order allow,deny
allow from all
<IfModule mod_php4.c>
  php_flag magic_quotes_gpc Off
  php_flag track_vars On
  php_value session.save_path "/var/state/horde2"
  php_value include_path .:/etc/horde2:/usr/share/pear:\
/usr/share/horde2
</IfModule>
</Directory>
# End HORDE2 Configuration Block
```

6 Rozšíření systému o sdílené poštovní schránky

Pomocí výše uvedených subsystémů složíme poštovní systém, který nám poskytuje základní funkce, jako jsou odesílání emailů, jejich příjem či přístup k poště pomocí protokolu IMAP. Získáme dokonce také možnost centrální správy účtů pomocí LDAP a jejich případnou replikaci mezi několika LDAP servery (např. uzly ve VPN). Budeme mít také přístup k poště pomocí WWW rozhraní, a budeme umět pracovat se sdílenými kontakty. Tedy přesněji řečeno jediný nám známý poštovní klient, který umí nejen číst ale i zapisovat sdílené kontakty, je *Ximian Evolution* [13]. Dále umí číst i zapisovat sdílené kontakty webový manažer Turba. Ostatní běžně používaní poštovní klienti umí sdílené kontakty pouze číst. Stále nám ale chybí možnost sdílení poštovních schránek.

Za účelem získat možnost sdílení poštovních schránek firma *soLNet, s. r. o.* [8] vytvořila IMAP proxy *DistribIMAP* a upravila webový klient IMP. Oba tyto produkty budou v brzké době uvolněny pod nějakou formou licence typu Open Source.

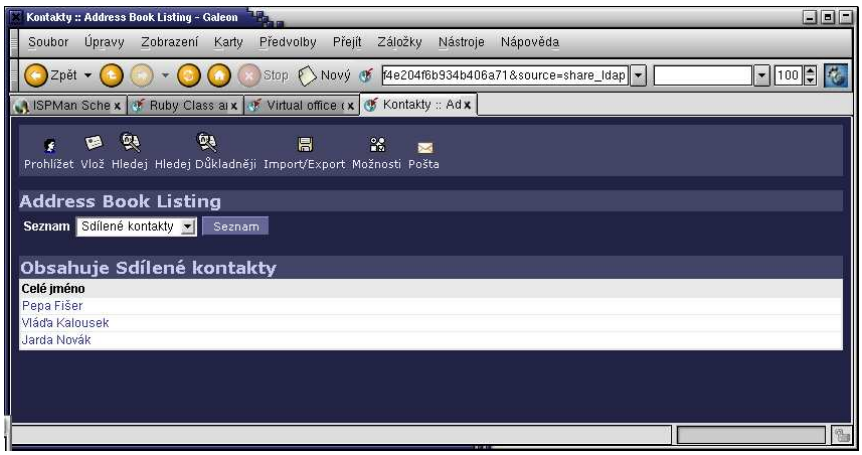
DistribIMAP se z pohledu poštovního klienta chová jako IMAP server a požadavky klienta zprostředkovává skutečným IMAP serverům. DistribIMAP je rozdělen na dvě části. Jedna komunikuje s poštovními klienty a IMAP servery. Druhá část je backend pro čtení přístupových práv k poštovním schránkám. V současné době existuje pouze backend k LDAP.

Pokud tedy klient například pomocí příkazu `SELECT` vybere schránku `sdilene_faktury`, DistribIMAP najde v LDAP schránku `sdilene_faktury` a zkontroluje, jaká práva má uživatel požadující přístup do schránky. Možná práva jsou čtení (read), zápis (write) a přidávání (append). LDAP server vrátí identifikaci uživatele, jemuž schránka patří (přihlašovací jméno a heslo), skutečné jméno schránky u tohoto uživatele, jméno IMAP serveru, na který je nutno se obrátit a číslo portu. DistribIMAP provede připojení na skutečný IMAP server, provede příkaz `SELECT` a bude nadále zprostředkovávat veškerou výměnu informací mezi klientem a skutečným IMAP serverem.

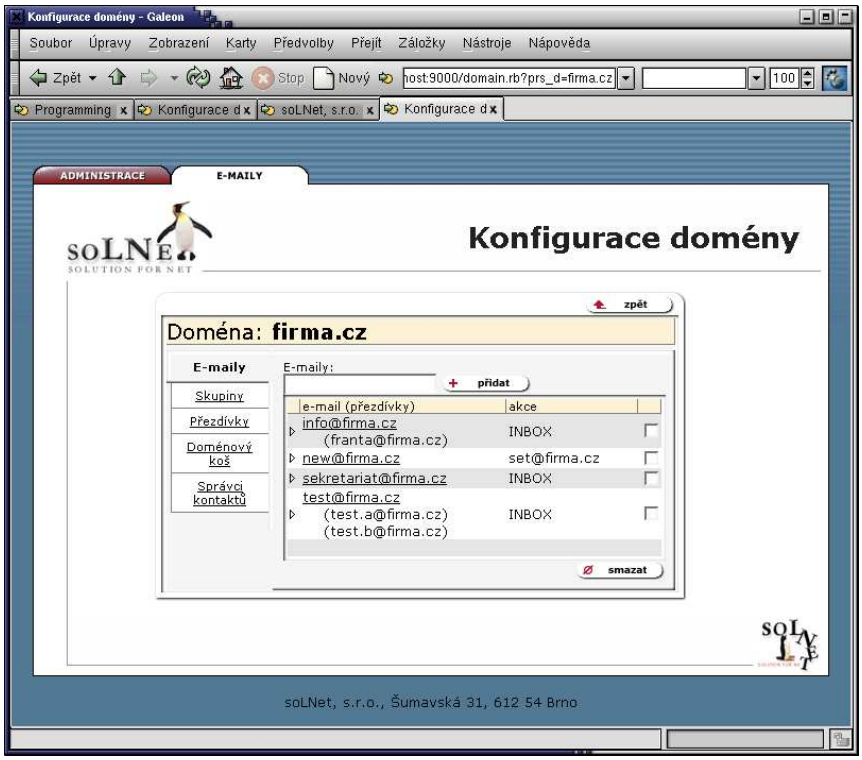
Toto transparentní řešení umožní přístup ke sdíleným schránkám všem poštovním klientům podporujícím protokol IMAP. Pozorný čtenář si již jistě všiml, že DistribIMAP dovoluje mj. sdílení schránek mezi účty na různých IMAP serverech. Toho lze využít zejména ve VPN. Upravená verze webového klienta IMP pak umožňuje definování přístupových práv na poštovní schránky, jejichž jste vlastníkem.

7 Administrační rozhraní

Jelikož v takto složitém systému není správa poštovních účtů jednoduchá, vytvořila firma soLNet, s. r. o. webové administrační rozhraní umožňující pohodlnou konfiguraci účtů. Toto rozhraní bude pravděpodobně také uvolněno pod nějakou formou licence typu Open Source. Jako WWW server v současné době slouží *Boa* [1]. Jeho autoři (Larry Doolittle a Jon Nelson) dovolili šíření WWW serveru Boa pod licencí GNU GPL. Administrační rozhraní umožňuje například nastavení přeposílání, přezdívek (aliasů), doménového koše, ...



Obrázek 1. Seznam kontaktů



Obrázek 2. Konfigurace domény

8 Závěr

Podařilo se nám tedy sestavit poštovní systém založený čistě na bázi Open Source Software. Jedinou oblastí, kterou jsme nepokryli je antivirový subsystém. Jelikož jsou ale v současné době viry doménou uzavřených systémů firmy *Microsoft*, není divu, že antivirové produkty jsou také uzavřené systémy.

Reference

1. Boa Webserver. http://www.boa.org/.
2. Courier-IMAP. http://www.inter7.com/courierimap/.
3. exim Internet Mailer. http://www.exim.org/.
4. IMP Webmail Client. http://www.horde.org/imp/.
5. Jwapmail. http://jwapmail.sourceforge.net/.
6. MAPS Realtime Blackhole List. http://mail-abuse.org/rbl/.
7. OpenLDAP. http://www.openldap.org/.
8. soLNet, s. r. o. http://www.solnet.cz/.
9. SpamAssassin. http://spamassassin.org/.
10. The Apache HTTP Server Project. http://httpd.apache.org/.
11. The Horde Application Framework. http://www.horde.org/horde/.
12. Turba Contact Manager. http://www.horde.org/turba/.
13. Ximian Evolution. http://www.ximian.com/products/evolution/.
14. Marcel Kolaja a Miroslav Bartošek. Jemný úvod do (anti)spamové problematiky. *Zpravodaj ÚVT MU*, 12(5):1–6, červen 2002. http://www.ics.muni.cz/bulletin/issues/vol12num05/bartosek/bartosek.ps.

Podivuhodný svět Linuxu, kapitola 2.5

Martin Mareš

Email: mj@ucw.cz

Abstrakt: Vývoj jádra Linuxu je opředený mýty a pověrami, takže nahlédněme pod pokličku, copak nám v hrnci nadepsaném prozaicky „2.5“ kerneloví čarodějové vaří. Kterak se Linux dočkal nového scheduleru, staronového driverového modelu, velice pěkné podpory threadů, user-mode kernelu, preemptivního jádra a spousty dalších užitečných vymožeností. Špatných zpráv garantujeme pouze minimální množství.

Jan Kasprzak, Petr Sojka (editoři): SLT 2002 – sborník semináře o Linuxu a TEXu, str. 273–273, 2002.

Autorský rejstřík

Tematický rejstřík

Jan Kasprzak, Petr Sojka
SLT 2002

Vydalo nakladatelství KONVOJ, spol. s r. o.,
v Brně v roce 2002 jako svoji 156. publikaci.

Technický redaktor: Petr Sojka
Sazba sázecím systémem LaTeX písmem Charter a Euler:
Petr Sojka a David Antoš z elektronických podkladů autorů
Autorkou loga SLT je Petra Rychlá.
Texty neprošly řádnou jazykovou úpravou.
Tisk KONVOJ, spol. s r. o.
Počet stran 278. První vydání.

Adresa nakladatelství:
KONVOJ, spol. s r. o., Berkova 22, 612 00 Brno
`http://www.konvoj.cz`, *email:* `konvoj@konvoj.cz`

ISBN 80-7302-043-2