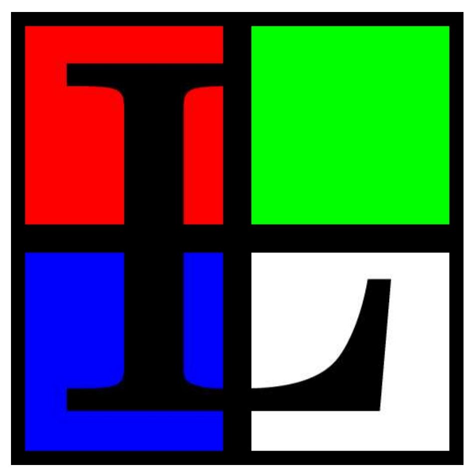
Universita Karlova v Praze
Matematicko-fysikální fakulta
2002

SOFTWAROVÝ PROJEKT

Links

webový prohlížeč



Projektová dokumentace

Autoři: Mikuláš Patočka
Martin Pergel
Petr Kulhavý
Karel Kulhavý

Vedoucí: Mgr. David Bednárek

Obsah

1. Úvod ... 3
2. Zadání ... 3
3. Proč jsme si tento projekt vybrali ... 4
4. Účastníci projektu ... 4
5. Struktura programu ... 4
6. Historie ... 6
7. Problémy a rozhodnutí ... 7
7.1 Scheduler ... 7
7.2 Grafická rozhraní ... 8
7.2.1 SVGAlib ... 8
7.2.2 Framebuffer ... 8
7.3 Javascript ... 9
7.3.1 Gramatika ... 9
7.3.2 Parser ... 10
7.3.3 Mezikód ... 10
7.3.4 Bezpečnost ... 11
7.3.5 Chyby a upozornění ... 12
7.4 Obrázky ... 12
7.4.1 TIFF ... 13
7.4.2 JPEG 2000 ... 13
7.5 Ostatní ... 13
7.5.1 Historie ... 13
7.5.2 Přenositelnost ... 13
7.5.3 Překlady ... 14
7.5.4 Ovládání ... 14
7.5.5 Fonty ... 14
8. Výsledek projektu ... 15
8.1 Javascript ... 15
8.1.1 Poznámky k implementaci upcallů a vnitřních funkcí ... 15
8.2 Formáty ... 18
8.2.1 Animované GIFy ... 18
8.2.2 HTML ... 19
8.3 Ostatní ... 19
8.3.1 Cookies ... 19
8.4 Cache ... 20
8.5 Význačné rysy prohlížeče ... 20
8.6 Podporované protokoly a formáty ... 21
8.7 Jazyky ... 22
8.8 Platformy ... 23
8.9 Chyby a problémy programu ... 23
8.9.1 Rádoby chyby ... 23
8.9.2 Problém se SVGAlib ... 24
8.9.3 Problém s framebufferem a gpm ... 25
8.9.4 Jazykové překlady ... 26

8.10 Srovnání s ostatními prohlížeči ... 26
8.11 Použité knihovny ... 29
8.12 Vývoj a testování ... 29
8.12.1 Prostředí používané k vývoji ... 29
8.12.2 Testování přenositelnosti ... 30
8.12.3 Testování programu, hledání chyb a optimalisace ... 31
8.13 Doporučené nástroje ke kompilaci ... 31
8.14 Kód přejatý od jiných autorů ... 32
9. Závěr ... 32
9.1 Problémy při vývoji ... 32
10. Plány do budoucna ... 33
11. Použité materiály ... 33
11.1 Javascript ... 33
11.2 Grafika ... 33
11.3 Grafické drivery ... 34
11.4 Zásluhy ... 34

1. Úvod

Toto je projektová dokumentace k programu **Links**. Dočtete se zde o průběhu vývoje, historii, účastnících a výsledku projektu. Tištěná verze dokumentace neobsahuje projektovou korespondenci. Plnou verzi dokumentace najdete ve formátu PDF na CD s programem.

Links je prohlížeč webových stránek pro operační systémy Unix a OS/2. Prohlížeč je možno provozovat v textovém i grafickém režimu. V grafickém režimu pod systémy X-Window, SVGAlib, AtheOS a Pmshell. Prohlížeč podporuje javascript (verzi 1.1 od Netscape Corporation), protokoly HTTP 1.0 i 1.1, FTP, Finger a SSL, formáty HTML verzi 4.0 bez CSS a obrázky GIF, XBM, TIFF, JPEG a PNG.

2. Zadání

Předmětem projektu je webový prohlížeč napsaný v jazyce C pod operačním systém Linux. Základní engine (FTP, HTTP requester, formátovač HTML apod.) budou napsány účastníky projektu. Kód psaný jinými lidmi může být použit v případě knihoven pro zpracování datových formátů (jpeg, png, mpeg, gif, ogg atd.). Knihovny pro práci s HTTP nebo FTP nebudou použity.

Prohlížeč by měl být stabilní, jeho výstup pokud možno kvalitní (snadno čitelný) a uživatelský komfort takový, aby se prohlížeč dal používat pro každodenní, rychlé a jednoduché browsení po Internetu. Největší důraz se bude klást na stabilitu, neboť pád prohlížeče znamená velké nepříjemnosti pro uživatele.

Středně velký důraz se poklade na kvalitu výstupu (z hlediska kvality obrazu, formátování dokumentu), neboť browsery se v dnešní době používají po relativně velké procento času stráveného u počítače a pohodlí očí je předpokladem pro úspěšnou a plynulou práci u počítače. Vedle kvality výstupu bude stát i rychlost, zejména bude žádoucí, aby v typickém případě byl prohlížeči proces limitován šířkou pásma přenosové cesty a nikoliv výkonem procesoru a grafického systému. V případě konfliktu mezi kvalitou a rychlostí se bude volit kvalita, pokud ovšem by zvýšení kvality přineslo neúměrné zpomalení, rychlost bude mít přednost.

Na poslední příčce žebříčku bude umístěn uživatelský komfort, který bude minimální, ale promyšlený a především takový, aby při práci s prohlížečem nezdržoval a nevyžadoval studium zdlouhavé dokumentace k uživatelskému rozhraní. Mezi komfort řadíme i přenositelnost na běžné operační systémy, kde přenos kódu nečiní větší potíže, protože transparentnost práce v multiplatformních prostředích přináší uživateli podstatně vyšší komfort, aniž by to bylo znatelné na úkor jiných funkcí prohlížeče.

Předpokládané vlastnosti prohlížeče:

- Přenositelnost: OS/2, Unix
- Tabulky
- Rámy
- Roletová menu
- Překlady menu do několika jazyků
- Bookmarks
- Stahování souborů na pozadí
- Keepalive connections
- Asynchronní DNS lookup

- Možnost spouštění externích programů na neznámé typy dat
- Možnost přepínání mezi textovým prohlížečem běžícím na konzoli a nebo grafickým prohlížečem běžícím pod SVGAlib, OS/2, X-Window, AtheOS
- Zobrazování obrázků, když prohlížeč poběží v grafickém režimu
- HTML verze 4.0 bez CSS
- V případě vadného HTML kódu prohlížeč nespadne
- Javascript
- Bez záruky: cookies (vypínatelné)
- Caching (expirace, refresh) bez záruky

3. Proč jsme si tento projekt vybrali

Existuje již mnoho prohlížečů webových stránek, leč žádný neodpovídal našim představám na stabilitu, bezpečnost, kvalitu výstupu a v neposlední řadě i rychlost. Navíc většina webových prohlížečů má překombinované a zbytečně složité ovládání, které znepříjemňuje práci uživatele.

Konkurenční prohlížeče se zaměřují více na používání tzv. progresivních technologií a podporování nejnovějších featur, než na stabilitu, přenositelnost a rychlost. Co se týče interpretace javascriptu, konkurenční prohlížeče se opět zaměřují na podporování nejnovějších vymožeností jazyka (nejlépe takových, které s jinými prohlížeči nefungují) a jsou velice nestabilní. Dalo by se říci, že javascript vládne nad uživatelem, místo, aby uživatel vládl nad javascriptem. Pro všechny konkurenční prohlížeče (Netscape, Mozilla, Opera, Konqueror, Internet Explorer) existuje javascriptový kód o délce pár řádek, který učiní prohlížeč neovladatelný a je ho nutno restartovat, nebo ještě hůře provedou na uživatelův počítač útok typu DoS (ať už obrovskými nároky na paměť či spotřebováním veškerého procesorového času).

Proto jsme se rozhodli napsat vlastní webový prohlížeč. Cílem projektu bylo napsat kvalitní, rychlý, bezpečný a stabilní, přesto jednoduchý, prohlížeč webových stránek. Důraz byl kladen především na stabilitu, kvalitu výstupu a rychlost. Za žádnou cenu by neměl havarovat, znemožnit uživateli ovládání, nebo si činit nepřiměřené nároky na paměť nebo procesor. Prohlížeč měl být přenositelný na většinu Unixů a zkompilovatelný s minimálními požadavky na systém.

4. Účastníci projektu

Tým řešitelů, pod vedením Mgr. Davida Bednárka, se skládal z těchto členů:

- Mikuláš Patočka
- Martin Pergel
- Petr Kulhavý
- Karel Kulhavý

Původně měl v týmu být ještě Jakub Drnec, ale ten odešel z fakulty a proto jsme zbyli jen čtyři. První rok projektu byl vedoucím Petr Merta, ale ten po roce odešel a projekt převzal Mgr. Bednárek.

Na prohlížeči se podíleli i další lidé, kteří zejména přeložili texty do rozličných cizích jazyků. Tito lidé jsou uvedeni na konci v kapitole Zásluhy.

5. Struktura programu

V této kapitole se dočtete hrubé rozdělení projektu na jednotlivé části a pak kdo které části psal. Detailnější popis struktury projektu najdete ve vývojové dokumentaci.

Základem celého prohlížeče je **scheduler** nebo též **select smyčka**. Zde se plánují všechny události, které mají nastat, a odtud se volají jednotlivé části prohlížeče. Select smyčka je kooperativní scheduler. Všechny části prohlížeče jsou napsány tak, aby v nich řízení nezůstávalo příliš dlouho, protože by nemohly být volány jiné části, což by se například projevovalo nereagováním na pokyny uživatele.

Další velmi důležitou částí je **session**, která zajišťuje management stahování, formátování a zobrazování dokumentů, ovládání, pohyb po dokumentu a vykonávání javascriptu. Session samotná nic nevykonává, pouze volá další části, které již jednotlivé činnosti provádějí. Session volá object requester, javascript, HTML parser a část zajišťující zobrazování a interakci s uživatelem.

Object requester se stará o stahování dokumentů ze sítě. Object requester volá **scheduler requestů**, na který je napojena souborová cache na všechny soubory stažené ze sítě a rozhraní jednotlivých protokolů (HTTP, HTTPS, FTP, finger a nahrávání z lokálního disku).

HTML parser zpracovává HTML dokumenty a převádí je do vnitřních struktur prohlížeče. **View**, neboli **zobrazovač**, se stará o zobrazování dokumentu, lámání textu a vyřizování událostí od uživatele (stisky kláves, pohyb myši, klikání ...). Krom pohybu po dokumentu ještě uživatel může používat nabídky, část, která se o toto stará se příznačně jmenuje **menu**. Zobrazovač volá funkce jednotlivých zobrazovacích ovladačů: terminálu, X, SVGAlib .... V grafickém režimu je potřeba sázet písmo, o to se stará část jménem **fonty**, která obsahuje fontovou cache a grafické rutiny pro zpracování a tisk písma.

Javascript se skládá z lexikálního analyzátoru, syntaktického analyzátoru, interpretu a vestavěných funkcí. Mezi javascriptem a session sídlí **rozhraní javascriptu**, které zprostředkovává přístup javascriptu k vnitřním strukturám prohlížeče (dokumentu, obrázkům, odkazům, formulářům, ...) a obsahuje funkce, kterými se zahájí a ukončí interpretace javascriptového kódu, a podobně. S javascriptem také souvisí **event handlery**, což jsou kusy kódy, které se volají při rozličných událostech, zejména vyvolaných uživatelem. Například při kliknutí na tlačítko, přejetí myší, nahrání dokumentu, stisknutí klávesy, změně textového políčka atd. Event handlery jsou zabudovány přímo do zobrazovače

Rozdělení práce:

- **Mikuláš Patočka:** session, scheduler, menu, object requester (včetně implementace jednotlivých protokolů), HTML parser, sazeč textu (grafický i textový), menu, port na OS/2, AtheOS
- **Martin Pergel:** parser a interpret javascriptu
- **Petr Kulhavý:** grafické rozhraní pro X-Window, rozhraní javascriptu (upcally, event handlery), bookmarks, český překlad, dekodéry xbm a tiff obrázků, výroba fontů, dokumentace, testování
- **Karel Kulhavý:** grafické rozhraní pro SVGAlib, obrázky (zobrazovač, cache, dekodéry jpg, gif a png, alfa kanálování), zpracování grafiky (ditherování, resamplování, zvětšování, gamma korekce), fonty (výběr fontů, pomocné programy pro přípravu fontů, výroba fontů), český překlad, logo

6. Historie

- **Zima 1998/1999:** Mikuláš začal se psát HTTP requester. **Důležité rozhodnutí:** napsat vlastní scheduler místo threadů nebo více procesů.
- **Jaro 1999:** HTTP requester chodí.
- **Začátek léta 1999:** Chodí jednoduché formátování stránky a pár HTML tagů. Není možnost však otestovat funkčnost prohlížeče na Internetu, protože Mikulášovi odešla síťová karta.
- **Léto 1999:** Přes prázdniny Mikuláš nemá přístup k Internetu, browser testuje přes HTTP proxy. Odbugován HTTP requester, napsány tabulky.
- **Podzim 1999:** Změněn algoritmus na tabulky, aby bral ohled na položku „width". Tento algoritmus se s několika opraveními chyb používá v **Links** dodnes v textové i grafické verzi. Přibyla spousta věcí jako download, hledání, přiřazení a další.
- **24.11.1999:** Je vydána verze 0.80.
- **25.11.1999:** Verze 0.80 oznámena na www.freshmeat.net, spousta lidí nachází chyby a tak rychle následují další verze (viz. Changelog).
- **27.11.1999:** Rozdělení na dvě verze: **stabilní**, číslovaná 0.8x, ve které se pouze opravují chyby; a **experimentální**, označovaná jako „links-current", do které se přidávají nové vymoženosti a která obsahuje poměrně dost chyb.
- **11.03.2000:** Vydána verze 0.84 — poslední z řady 0.8x.
- **15.06.2000:** Links-current je dostatečně otestován a jsou z něj odstraněny chyby, příprava na vydání, stává se z něj 0.90pre1.
- **28.06.2000:** Vydán **Links** 0.90
- **28.06.2000:** Vydán **Links** 0.91, protože 0.90 obsahoval chybu v rámech udělanou na poslední chvíli (když se šlo na link, tak se vždy zobrazoval soubor na celé obrazovce a ne v rámu)
- **23.07.2000:** Rozdělení na dvě verze: **0.9x**, kde se opět pouze opravují chyby, a **links-current** s grafikou a experimentálními funkcemi.

Následuje vývoj „links-current":

- **Léto 2000:** Grafický driver na OS/2, portování uživatelského rozhraní do grafiky.
- **Říjen 2000:** Vypsání projektu pod vedením Petra Merty.
- **Podzim 2000:** Navrženo grafické rozhraní, napsán český překlad.
- **Podzim, zima 2000/2001:** Sehnání gramatiky JS, lexikální, syntaktická a sémantická analýza, generátor mezikódu. Zobrazovač stránek v grafice, psaní grafického driveru pro X.
- **Březen 2001:** Oficiální schůzka projektu, předvádění vedoucímu, co jsme vytvořili, hodnocení naší práce vedoucím.
- **Jaro a léto 2001:** Psaní interpretu javascriptu bez upcallů, psaní obecných seznamů a bookmarků.
- **Začátek léta 2001:** Navrženo rozhraní pro javascript.
- **Říjen 2001:** Změna vedoucího z Mgr. Merty na Mgr. Bednárka. Návrh implementace obrázků.
- **05.10.2001:** Přechod ze systému vzájemného posílání patchů na CVS.
- **18.10.2001:** Vydání výroční zprávy.
- **Listopad 2002:** Zahájena implementace upcallů.

- **Podzim 2001, zima 2001/2002:** Psaní upcallů, spousta drobností pro lepší použitelnost.
- **11.01.2002:** Oficiální schůzka s vedoucím projektu, kde se hodnotil stav, dosavadní výsledky, určilo se, kdo ještě bude co programovat, a dohodl se termín odevzdání projektu.
- **Zima–jaro 2002:** Masivní ladění a testování javascriptu, přizpůsobení javascriptu realitě (prasácky psaným stránkám). Tabulky v grafice.
- **Jaro 2002:** Psaní dokumentace, spousta drobností.
- **Leden 2002:** Přepínání fontů,
- **Únor 2002:** Přepsán ditherovací algoritmus, přepsána manipulace s obrázky, napsána podpora PNG obrázků, začátek psaní event handlerů v javascriptu.
- **Březen 2002:** Přidán dekodér GIF a JPG. Dopsány kompletně upcally.
- **Duben 2002:** Napsán dekodér XBM a TIFF, ikona, zahájeno psaní ovladače pro framebuffer.
- **24.04.2002:** Schůzka s vedoucím a předvádění projektu před blížící se obhajobou. Rozhodli jsme dočasně zastavit vývoj a pouze testovat a opravovat nalezené chyby.
- **10.05.2002:** Vydání verze 0.97.
- **16.05.2002:** První oficiální vydání pre-verze prohlížeče.
- **20.05.2002:** Vydána verze 2.0pre1, aktualisován slovenský překlad a opraveno několik chyb.
- **23.05.2002:** Vydána verze 2.0pre2, opravena chyba s refererem, opraven `frame.top` v javascriptu, zvýšena přenositelnost (libpng 1.2.2).
- **23.05.2002:** Vydána verze 2.0pre3 s několika opravenými chybami.
- **26.05.2002:** Vydána verze 2.0pre4, několik opravených pádů, opravena chyba TIFF obrázků a 16-bitových PNG na big endianu, opraveny nekompatibilita s MIPSpro Compilers a nějaké další drobnosti.
- **28.05.2002:** Vydána verze 2.0pre5 s opraveným segfaultem na 16-bitových PNG obrázcích.
- **30.05.2002:** Vydána verze 2.0pre6 s aktualisovaným maďarským překladem.

Schůzky projektu se konaly velmi často, většinou v menze ve frontě na oběd, u oběda, v počítačové laboratoři, při seancích 0verkillu a kdykoliv jsme se viděli jsme probírali problémy projektu, nové věci, co je potřeba napsat a další. Mnoho věcí jsme též probírali mailem nebo talkem. Projekt měl tři oficiální schůzky s vedoucím, první při vypsání projektu, druhá po půl roce a třetí opět po dalším asi půl roce. Projekt jsme psali samostatně, pouze tu a tam jsme se potřebovali poradit s vedoucím. Vedoucímu jsme pravidelně psali zprávy o stavu projektu.

7. Problémy a rozhodnutí

V této kapitole rozebereme problémy, se kterými jsme se při vývoji projektu potýkali, a důležitá rozhodnutí, před kterými jsme stáli, včetně jejich řešení.

7.1 Scheduler

Rozhodli jsme se napsat vlastní scheduler událostí namísto multithreadového nebo multiprocesového prohlížeče. Mikuláš kdysi asi dva dny zkoušel psát multithreadovou variantu, ale moc se mu to nelíbilo, tak to smazal a napsal vlastní scheduler. Mít vlastní

scheduler je efektivnější a jelikož je kooperativní, tak jednotlivé části se mohou mezi sebou snáze přepínat a nemusí na sebe složitě čekat.

7.2 Grafická rozhraní

Při psaní ovladače pro grafický systém X-Window jsme měli možnost vybrat si z množství již existujících toolkitů, leč této možnosti jsme nevyužili a napsali jsme ovladač přímo s použitím knihovny Xlib. To z důvodu, že toolkitů je mnoho a každý používá jiný. Tudíž by byl uživatel potenciálně nucen instalovat na svůj počítač další knihovny, čímž by se instalace zbytečně komplikovala. Navíc, když je toolkitů tolik, tak není důvod použít jeden konkrétní toolkit a ne jiný. Použití toolkitu by mělo za výhodu pouze podobnost s jinými aplikacemi a usnadnění práce při programování. My jsme se ale snažili o maximální podobnost s ovládáním v textovém režimu, k čemuž nebylo možno žádný toolkit použít.

Grafické rozhraní jsme navrhli na nízké úrovni, aby bylo portabilní, jednoduché a aby bylo možno snadno přidat nový grafický ovladač. Rozhraní obsahuje elementární funkce pro nakreslení bitmapy, vybarvení plochy, nakreslení čáry, scroll a podobně. Tyto funkce lze velice snadno napsat na libovolném grafickém rozhraní.

Díky návrhu grafického rozhraní prohlížeč v okenních systémech používá pouze jedno okno. To má za výhodu snadnou predikovatelnost, kde které okno bude, okno se nemůže ztratit na vedlejší obrazovce ani nemůže být přikryto jiným oknem, čímž se zvyšuje komfort uživatele.

7.2.1 SVGAlib

Na starších SVGAlibách **Links** nemusí fungovat, neboť ty obsahují různé chyby, jako například vadný kód, který při provádění grafických operací kreslí nesmysly. Pouštění **Links** na starších SVGAlibách může představovat bezpečnostní riziko, protože například SVGAlib 1.2.10 se nesprávně vzdávala rootovských práv.

Za toto nemůže **Links**, ale návrháři SVGAlib. Já (Karel Kulhavý) jsem názoru, že SVGAlib je broken-by-design a měla by se přepsat, což ovšem není v rámci projektu **Links**. Jsem dále toho názoru, že VT_ACTIVATE API v kernelu Linuxu je broken by design a že by se také mělo přepsat, což ovšem není v rámci projektu **Links**.

I nová SVGAlib má chyby ve funkci `vga_setlinearaddressing` (které se projevují například na kartě S3 Virge 3d), kvůli kterým musela být tato funkce v **Links** zakomentována (tak, jak je to například v programu Quake). Díky tomu je **Links** pomalejší v určitých případech, za což nemůže **Links**, ale opět SVGAlib.

Některé grafické funkce ve SVGAlib byly tak zabugované, že nefungovaly vůbec, a musely být emulovány pomocí funkcí primitivnějších, například vyplňování oblastí pomocí `putpixel` (2-barevné režimy). Vzhledem k tomu, že i toto rozhraní kreslilo nesmysly, a vzhledem k tomu, že 2-barevné režimy neumí zobrazit žádné barvy, bylo nakonec od nich upuštěno a byly z **Links** odstraněny.

Links používá akcelerační primitiva SVGAlib v případě, kdy je SVGAlib podporuje. Problém je, že SVGAlib je pro většinu karet nepodporuje, alespoň ne pro tu, ke kterým měl tým **Links** přístup. Vzhledem k celkové koncepci SVGAlib se dá předpokládat, že budou masivně zabugované a že tedy **Links** bude na některých kartách těžce shazovat systém. To není problém **Links**, ale SVGAlib.

7.2.2 Framebuffer

Psaní grafického ovladače pro framebuffer na Linuxu bylo velmi nesnadné díky velmi špatné nebo úplně chybějící dokumentaci. Proto není zaručena stoprocentní spolehlivost a to, že ovladač bude fungovat se všemi grafickými kartami na všech počítačích. Doku-

mentaci jsme byli velmi často nuceni nahrazovat luštěním zdrojových textů linuxového kernelu, čtením grafických ovladačů pro jednotlivé grafické karty a čtením velmi strohých komentářů v hlavičkových souborech rozhraní framebufferu. Tyto podmínky nám velmi znesnadňovaly práci a zbytečně prodlužovaly dobu vývoje. Mnohé věci jsme museli zjišťovat přímo u autorů linuxového jádra nebo empiricky ozkoušet chování a dle něj usoudit, jak se rozhraní používá. Tento způsob se nám jeví jako velmi nečistý, leč díky absenci kvalitní dokumentace k rozhraní jsme neměli jinou možnost.

Při psaní framebufferu jsme potřebovali zjišťovat pozici myši. K tomu jsme použili knihovnu `gpm`, pomocí které se zjišťuje stav myši i na terminálu (v textovém módu). Framebuffer je de-facto terminál s možností grafického výstupu. Framebuffer žádné vlastní rozhraní pro myš neposkytuje, pouze umožňuje grafický výstup na obrazovku. Na terminálu jiný způsob čtení myši nepřipadá v úvahu, protože ne na každém počítači má běžný uživatel přístup k zařízení myši (například `/dev/mouse`). Knihovna `gpm` čte zařízení myši a umožňuje uživatelským programům přes socket zjišťovat polohu a stisknutí tlačítek. Naštěstí na většině počítačů s Linuxem je `gpm` nainstalováno a tedy ho lze použít.

Problém je ale v tom, že `gpm` je striktně textové, tedy se kurzor pohybuje po znacích a nikoliv po pixelech. Zkoušeli jsme číst myš po znacích a špinavým trikem podvrhnout větší rozměry obrazovky, ale to se staršími verzemi `gpm` nechodilo. Proto jsme udělali čtení relativních pohybů myši, které funguje se všemi verzemi. Problém je ale v tom, že myš se pohybuje příliš pomalu. Proto jsme nakone stejně museli vynásobit relativní pohyb konstantou a tedy opět není možno pohybovat s myší plynule a reálně hrozí nebezpečí, že na obrazovce bude místo, kam uživatel nebude moci kliknout, přestože to bude nutné. Částečně lze tento problém odstranit zvětšením písma a obrázků, ale bohužel ho nejde odstranit úplně. V rámci pokusů o nápravu této situace jsme ve Framebufferu umožnili pohybovat myší po pixelech klávesami F5 – F8.

Citlivost myši si klientský program `gpm` bohužel nemůže nastavit, protože ji nastavuje administrátor „natvrdo“ při pouštění `gpm` daemona. Proto jsme zvolili kompromis mezi citlivostí a přesností. Kurzor se po obrazovce pohybuje tedy přijatelně.

Teoreticky by bylo možné do `gpm` podporu grafiky dopsat, neboť `gpm` je šířené pod licencí GPL. Dopsat podporu grafiky by neměl být veliký problém, je i jistá šance, že kdyby se upravená knihovna poslala autorům, autoři by ji v příští verzi vydali i s touto grafickou podporou. Problém ale tkví v tom, že ne každý by měl na počítači nainstalovánu tuto nejnovější verzi, tedy opět bez administrátorských práv by uživatel nemohl na framebufferu myš používat. Toto by spíše bylo dlouhodobější řešení, neboť by se dalo počítat s tím, že by se grafická verze `gpm` rozšířila.

Pro uživatele, kteří mají administrátorská práva jsme vytvořili patch, po jehož aplikaci na `gpm` se kurzor bude po obrazovce pohybovat plynule. Patch je distribuován společně s prohlížečem, je uložen v souboru `PATCH-gpm-1.20.0-smooth-cursor`. Patch byl vyroben pro verzi `gpm` 1.20.0, ale s jinými verzemi pravděpodobně bude fungovat též, případně po malé úpravě.

Framebuffer se nám podařilo implementovat pouze na architektuře Intel. Měli jsme přístup ještě k počítačům architektury Sparc, na kterých též běžel Linux s framebufferem, ale tyto počítače nemají lineární mapování videopaměti, což komplikuje implementaci. Jelikož jsme neměli dokumentaci, nelineární mapování paměti jsme nepodpořili.

7.3 Javascript

Interpret jsme se rozhodli psát od základu, protože jsme došli k názoru, že interpret javascriptu v **Links** může být užitečný. Navíc Mgr. Bednárek na Překladačích prohlásil, že vyrábět překladač nebo interpret od základu se stane málokomu a my považujeme návrh a výrobu překladače za činnost neobyčejně zajímavou, byť poněkud náročnější.

7.3.1 Gramatika

Prvním problémem, se kterým jsme se setkali při psaní javascriptu, bylo sehnat normu, dle které bychom se měli řídit. Asi měsíc jsme usilovně sháněli na Internetu, ale nic jsme nemohli najít. Podařilo se nám sehnat normu ECMA 262 popisující gramatiku ECMA scriptů, ale v té době jsme ještě neměli nejmenší tušení, že existuje norma Javascript 1.3, o to menší, že se řídí gramatikou ECMA 262. Pak se kdesi podařilo najít normu Javascript 1.1 od Netscape Corporation (nikoliv na stránkách firmy Netscape), podle které jsme postupovali. O normě Javascript 1.3 jsme se dozvěděli asi rok po nalezení normy 1.1 a po začátku její implementace.

Javascript 1.1 měl gramatiku jednodušší než byla gramatika ECMA 262 — neobsahovala dvojredukční konflikty a skoro žádné shift-redukce konflikty. V gramatice ECMA byly desítky až stovky dvojredukčních konfliktů a shift-redukčních konfliktů.

Document-object model jsme převzali z Netscape 2.0 podle nalezené dokumentace. Přidali jsme některá vylepšení.

7.3.2 Parser

Při navrhování parseru javascriptu jsme měli možnost napsat lexikální i syntaktický analyzátor buďto vlastní, nebo použít již existující nástroje Bison a Flex pro výrobu syntaktických a lexikálních analyzátorů.

V případě syntaktického analyzátoru jednoznačně zvítězila strojová výroba analyzátoru, neboť gramatika je pro ruční výrobu analyzátoru příliš složitá. Gramatika obsahuje 131 pravidlo a automat z Bisonu čítá 212 stavů. Výsledný syntaktický analyzátor je tedy LALR(1). V gramatice je cca 40 konfliktů typu shift-redukce, které se řeší předností operace shift. Gramatiku jsme po stažení strojově přepsali do tvaru akceptovatelném Bisonem, akce, které se mají provádět při redukcích jsme poté dopsali ručně.

Pro lexikální analýzu jsme použili nástroj Flex, protože ruční psaní automatu je zdlouhavé a většinou výsledný automat obsahuje mnoho chyb. Při použití programu Flex bylo potřeba předefinovat funkce `getc` a `ungetc`, protože se nepodařilo zjistit, jak jinak donutit Flex číst vstup z paměti a ne ze streamu.

S rozhodnutím použít Flex a Bison je spojen problém, co když uživatel nebude mít na svém počítači tyto nástroje? Přece jen tyto nástroje nejsou (narozdíl od překladače jazyka C) na systémech Unix úplně obvyklé. Proto jsme se rozhodli do zdrojové distribuce přiložit výstupy z uvedených nástrojů (tedy zdrojové soubory v jazyce C). V distribuci samozřejmě jsou též vstupní zdrojové texty pro nástroje Flex a Bison, takže je možno automaty kdykoliv znovu vygenerovat.

7.3.3 Mezikód

Na základě rady Mgr. Bednárka byl interpret javascriptu rozdělen na parser s generátorem mezikódu a interpret mezikódu, protože bez rozdělení by se v automatech generovaných Flexem a Bisonem špatně implementoval multitasking. Tudíž by se muselo čekat, až skript doběhne, a teprve pak by se mohlo pokračovat.

Formu mezikódu jsme zvolili stromovou, nyní již opravdový DAG, aniž by byly prováděny jakékoliv optimalisace. Stromový mezikód jsme si zvolili, protože strom je přímo výstupem syntaktické analýzy. Kdybychom měli generovat čtveřicový nebo trojicový mezikód, trvalo by nějaký čas jeho generování, což by zdržovalo interpretaci krátkých skriptů, které jsme chtěli být schopni interpretovat rychle. Po konsultaci s kolegou Hubičkou, který prohlásil, že stromový mezikód je lépe interpretovat zásobníkem, jsme se rozhodli napsat interpret zásobníkový.

Strom mezikódu obsahuje v každém uzlu informaci o čísle řádky, na kterém se dotyčný kus kódu nalézá, informaci o operátoru a místo pro 6 argumentů. Argumenty šlo též uložit

například ve spojovém seznamu, v poli proměnlivé délky nebo v jiné datové struktuře. Jelikož operátor nemůže mít nikdy proměnný počet argumentů, tak jsme zvolili pole pevné délky, které má i tu výhodu, že přístup k prvku je v konstantním čase.

Jména jsou překládána na klíče, jejichž seznam je zahashován. Identifikátory mají klíče v rámci jednoho kontextu jednoznačné. Toto řešení bylo zvoleno pro zajištění vyšší rychlosti interpretace. Rozhraní je navrženo tak, aby byla možnost měnit velikost „adresných prostorů", zatím je velikost 128 adres, což je kompromis mezi velikostí a počtem kolizí. Podle teoretických propočtů je při kvadratické velikosti hashovacího pole střední počet pokusů na hledání konstantní. T.j. prvních 11 záznamů by mělo zkolidovat s $P \leq 50\%$. Naopak kolize v každém políčku hashovacího pole v očekávaném případě nastane až po $128 \log 128$ záznamech, takže až po definování 896 proměnných v adresním prostoru se tabulka „přeplní". Navíc ve spojových seznamech vedoucích od každého políčka hashovací tabulky je aplikováno MFR, které vykazuje výsledky nejvýše $2\times$ horší než optimální samoudržující seznam (viz. RNDr. Koubek: Datové struktury).

Identifikace pomocí klíčů sice znamená značné urychlení, ale občas naopak škodí — například když se má vyhledat objekt definovaný v dokumentu. V mnoha případech je však výhodná, jelikož interpretace sestává zejména z vyhledávání identifikátorů a při nastavení podle normy aspoň první hledání probíhá vždy interně v javascriptu, druhé hledání při aspoň druhém přístupu taktéž (například `document.object....`).

7.3.4 Bezpečnost

Rozhraní javascriptu je navrženo tak, že nedovolí přístup javascriptu k „cizím" objektům. Cizí objekty jsou všechny objekty v dokumentech z jiných serverů, než je server, z kterého pochází dokument v němž běží přistupující javascript. Všechny upcally, které pracují s nějakými objekty, si striktně testují tato přístupová práva, takže javascript „cizí" objekty vůbec nevidí a má dojem, že ani neexistují.

Po upozornění, že v interpretu může přetéct mnoho věcí, rozhodli jsme se napsat „účtování paměti". Uživatel má možnost nastavit maximální velikost paměti, kterou smí javascript naalokovat. Po přetečení tohoto limitu při konci elementárního kroku interpretace, je skript (kontext) zabit a udělána na něm „čistka", aby se část paměti uvolnila a mohly pokračovat ostatní kontexty. Účtování je možné provést buďto pro každý kontext zvlášť, nebo pro všechny najednou. Kdyby se účtovalo každému kontextu zvlášť, hrozilo by nebezpečí, že by skript otevřel mnoho kontextů, jimiž by vyčerpal paměť. Proto jsme zvolili variantu účtování paměti pro všechny kontexty.

Přetečení by bylo možné kontrolovat přímo v alokační funkci, ale nastal by problém, jak skript z alokační funkce ukončit. Proto se v alokační funkci paměť naalokuje vždy a přetečení se kontroluje až na konci funkce `zvykni`. Jelikož tato funkce provádí pouze elementární kroky interpretace, nehrozí naalokování příliš velkého kusu paměti v jednom kroku a tedy tato metoda je bezpečná. Dle našich propočtů by v elementárním kroku obsazená paměť neměla narůst více než lineárně.

S paměťovou náročností javascriptu souvisí i potenciální nebezpečí přetečení zásobníku puštěním například nekonečné rekurse. K ošetření tohoto nebezpečí jsme zavedli omezení maximální hloubky rekurse — uživatel má opět možnost v menu nastavit.

Kvůli možnosti vyhladovění interpretu spuštěním nekonečné smyčky, bylo rozhodnuto, že interpret **musí** po konečném čase vrátit řízení a přeplánovat se. Scheduluje se tedy po vygenerování mezikódu a pak po stonásobném zavolání funkce `zvykni` (100 je empirická konstanta, která vykazuje relativně uspokojivý poměr cena/výkon), navíc se interpretace přeruší při některých upcallech. V rámci jednoho kontextu smí běžet nejvýše jedno vlákno. Běh ve více kontextech najednou však není nikterak omezen a nastává dokonce velmi často. Počet kroků do „preempce" lze změnit konstantou v době kompilace. Bylo by možné nechat tuto konstantu měnit uživatelem za běhu, ale obávali jsme

se, že někteří uživatelé by význam tohoto nastavení nepochopili a jen by zkoušeli, jaké hodnoty lze nastavit, interpret by pak buďto skripty interpretoval pomalu, nebo by měl příliš dlouhou dobu odezvy při stisku klávesy.

Po zkušenostech s ostatními prohlížeči jsme dospěli k závěru, že tato bezpečnostní opatření nestačí. Útok může být proveden například vypisováním výstrah v nekonečné smyčce, neustálou změnou URL a podobně. Právě tento typ útoku žádný existující prohlížeč nevydrží. Stačí například takovýto jednořádkový kód:

```
while(1)alert("Jsi BFU!");
```

Toto není typický DoS útok, který by například spotřeboval veškerou paměť nebo 100 % procesorového času, proto naň patrně autoři ostatních prohlížečů nemysleli. V případě tohoto útoku je uživatel nucen neustále odklikávat okénka javascriptu, díky čemuž se nedostane k ostatním ovládacím prvkům. Nejedná se tedy útok na počítač, ale na uživatele. Některé prohlížeče (například Netscape Navigator) ani nečekají na odezvu uživatele ale vytvářejí stále další a další okénka, čímž pochopitelně znemožní používat i ostatní aplikace.

Proto jsme se rozhodli pro řešení jednoduché, leč účinné, a to umožnit uživateli v každém okénku vytvořeném javascriptem skript ukončit. V případě změny URL, otevírání a zavírání okna prohlížeče se prohlížeč uživatele zeptá, zda dovolí tuto akci javascriptu provést. Uživatel má možnost povolit, odmítnout nebo ukončit skript. Tím se znemožní javascriptu znepříjemňovat uživateli život nevyžádaným měněním URL, zavíráním okna prohlížeče nebo zaplavováním novými a novými okny.

7.3.5 Chyby a upozornění

Norma javascriptu říká, co je správně a co ne, značně nejasně. Proto, pokud je to možné, chyby při interpretaci ignorujeme, navíc jsme dali k disposici možnost uživateli nastavit tolerantnost interpretu k chybám. Některé chyby rovnou ignorujeme a jen na ně upozorníme warningem — výstrahou. Umožňujeme **všechny** typové konverze automaticky, stejně jako vypnutí hlášení o chybách, vypnutí celého javascriptu nebo okamžité zastavení **všech** v současné chvíli běžících interpretacích, nemluvě o možnosti při **každém** projevu oknem javascript zabít (má-li to ještě smysl).

Error znamená definitivní konec interpretace v daném kontextu, tzn. v daném kontextu jsou další skripty již ignorovány, takže u stránek na javascriptu založených a špatně napsaných nezbývá nežli kouknout do zdrojového textu stránky, pochopit, co asi skript měl znamenat, interpretovat si jej sám a provést eventuální požadované akce (například změnu URL) ručně. Zabití všech skriptů má za následek vyvolání erroru ve všech existujících kontextech, proto je na stránkách obsahujících odkazy jen přes javascript krajně nevhodné požádat o vybití skriptů.

Rozhodnutí nepokračovat po erroru v interpretaci je podporované tím, že po syntaktické chybě by bylo třeba zneplatnit některé funkce a zrušit kus stromu. Další interpretace se typicky odkazují na předchozí výsledky, takže by stejně vedla k chybě. K pokračování po sémantické chybě by bylo třeba odstraňovat problémy se zásobníky rodičů a argumentů a kouzlit s „adresovými prostory“. Stačí, že podobné magie je zapotřebí při operacích `break`, `return` či `continue`. Vznikaly by chyby „zřetězené“ nebo lépe „zavlečené“ předchozími. Nepokračování po chybě je obvyklé i u ostatních browserů.

Vzhledem k tomu, že autoři javascriptů často neuvádějí objekt dokument, když přistupují k jeho prvkům, rozhodli jsme se umožnit resoluci jmen v hlavním adresním prostoru, ale pouze, když si to uživatel vyžádá. Uživatel má možnost v menu zaškrtnout povolení globální resoluce jmen. Standardně je toto nastavení zapnuté. Možnost vypínání jsme uživateli dali z toho důvodu, že globální resoluce je pomalejší nežli lokální a také proto, že to tak některé stránky vyžadují.

7.4 Obrázky

V první implementaci obrázků byl zobrazovač obrázků (dekodér a ditherovací engine) nerestartovatelný, což se ukázalo jako značný nedostatek, neboť zobrazování obrázků bylo velmi pomalé a blokovalo ostatní činnosti prohlížeče. Proto jsme museli zobrazovač obrázků přepsat do restartovatelné verze, aby bylo možno uprostřed dekódovaní a ditherování přeschedulovat.

7.4.1 TIFF

Rozhodli jsme se podpořit grafický formát TIFF. Tento formát se sice na webu příliš nevyskytuje, ale občas se najdou některé důležité dokumentace, které se v jiném formátu nenajdou. K dekódování jsme se použili knihovnu `libtiff`, protože díky mnoha variantám a rozmanitosti formátu by bylo ruční psaní dekodéru časově příliš náročné. Formát TIFF specifikuje, že dekódovat se smí až po natažení celého souboru. Tedy TIFFy se uživateli nebudou zobrazovat průběžně během natahování ze sítě. To ale není chyba **Links**, nýbrž vlastnost formátu TIFF.

Jiné prohlížeče TIFF standardně nepodporují, potřebují na jeho zobrazování plugin. To je ovšem na škodu, neboť TIFF není úplně nepoužívaný formát.

7.4.2 JPEG 2000

Při psaní dekodérů pro různé grafické formáty jsme chtěli napsat i podporu pro nový standard JPEG 2000, který nás zaujal svojí vysokou kompresí. Byli jsme toho názoru, že tento formát bude v budoucnosti zajisté používán, tedy jeho podpora by byla dobrou investicí do budoucnosti. Bohužel jsme ale nenalezli žádnou open source knihovnu podporující tento formát. Ruční psaní dekodéru waveletů by byla časově poněkud náročná činnost, proto jsme od podpory tohoto formátu upustili. Nicméně až bude k disposici platformě nezávislá knihovna podporující JPEG 2000, nebude problém díky flexibilnímu návrhu rozhraní pro obrázky tento formát do prohlížeče přidat.

7.5 Ostatní

Rozhodli jsme se umístit všechny datové soubory a spustitelný kód do jednoho binárního souboru z důvodu přenositelnosti. Tímto je uživatel zproštěn problémů, kam umístit data k prohlížeči, a je vyřešen problém s hledáním dat (každý uživatel chce data umístit typicky do jiného adresáře, tedy po překopírování binárního souboru by **Links** nešel spustit). Takto stačí překopírovat jeden binární soubor na jiný počítač se stejnou platformou a prohlížeč tam poběží.

Jako jazyk, ve kterém bude projekt napsán, jsme si vybrali jazyk C, protože C je univerzální programovací jazyk, který charakterizují úsporné výrazy, moderní řízení běhu, struktura údajů a bohatství operátorů. Obecnost jazyka C jej činí vhodnějším a efektivnějším pro mnohé úlohy než jiné „mocnější“ jazyky. Řídili jsme se normou ANSI C, abychom dosáhli přenositelnosti na co nejvíce systémů.

7.5.1 Historie

V historii se ukládá celá zformátovaná stránka včetně pozice kurzoru na stránce a obsahu všech formulářů. Tedy když uživatel jde v historii zpět, ocitne se na přesně stejném místě, odkud šel na novou stránku. Obsah formulářů se neztratí ani při reloadu stránky, jak je tomu u jiných prohlížečů.

7.5.2 Přenositelnost

Aby byl projekt dobrý, museli jsme zaručit přenositelnost kódu. Proto jsme se psali kód v ANSI C. Nicméně některé části musely být alespoň částečně platformně závislé.

Přesto jsme museli zaručit přenositelnost. Z toho důvodu jsme použili nástroje `Autoconf` a `Automake`, pomocí nichž je možné vyrobit na konkrétním počítači makefile „na míru“. Bez použití těchto nástrojů by bylo velice pracné vytvořit přenositelný Makefile a zajistit kompilaci na různých systémech.

V distribuci dodáváme vygenerovaný skript `configure`, který slouží pro upravení konfigurace pro konkrétní počítač. Mimo to dodáváme také vstupní soubory pro programy `Autoconf` a `Automake`, aby bylo možno skript kdykoliv znovu vygenerovat.

7.5.3 Překlady

Links má přeložena menu do mnoha jazyků. Již od začátku projektu jsme s touto vlastností počítali. Proto jsme byli hned v počátcích projektu postaveni před rozhodnutí, jakým způsobem překlady realisovat. Pro překlad řetězců existuje nástroj `gettext`, který jsme během vývoje používali. Ten se ale ukázal jako omezující, protože není přenositelný, neumí jiné kódové stránky než ISO 8859-2 a v libc různé od glibc 2 obsahuje mnoho chyb.

Z toho důvodu jsme gettext zavrhli a napsali jsme si vlastní systém překódování jazyků. Jazyky se přidávají při kompilaci, když se provede nějaká změna, tak se spustí skript, který vygeneruje zdrojové soubory v jazyce C. Podobně funguje systém překladu znakových sad.

7.5.4 Ovládání

Ovládání prohlížeče jsme zvolili pomocí jednoduchých interaktivních nabídek. V počátcích prohlížeče, kdy ještě fungoval pouze v textovém módu, jsme se inspirovali textovým prohlížečem Lynx, který se ovládá pomocí horkých kláves. Toto ovládání není příliš šikovné, protože si uživatel musí pamatovat množství kláves pro všechny možné funkce. My jsme zachovali zpětnou kompatibilitu ovládání s prohlížečem Lynx (horké klávesy jsou tedy stejné), protože byl v té době rozšířený a uživatelé byli na jeho ovládání zvyklí. Navíc jsme přidali jednoduchá interaktivní menu, odkud uživatel může všechny funkce vyvolat. Ti, kteří si nepamatují množství horkých kláves, mají možnost snadného ovládání. V textovém módu jsme i umožnili používat myš (pomocí knihovny libgpm). To usnadňuje ovládání ještě více. V grafickém módu jsme ovládání poněkud pozměnili, klade se větší důraz na myš, například pohyb po odkazech a výběr odkazu klávesami nefunguje a musí se použít myši.

Snažili jsme se, aby ovládání zabíralo minimální část obrazovky (což má velikou cenu zejména v textovém módu, kde je malé rozlišení), proto na první pohled menu není vidět a vyvolává se klávesou escape. U jiných prohlížečů nám vadí překombinované ovládání, ve kterém se nejen špatně orientuje, ale také zabírá nezanedbatelnou část obrazovky. V **Links** zabírá obrazovku pouze stavový řádek (dole) a titulek stránky (nahoře). Kategoricky jsme odmítli lištu pro zadávání URL i lištu s ikonami, protože zabírají místo a funkce lze vyvolat snáze stisknutím klávesy, nežli trefováním se myší do tlačítka.

7.5.5 Fonty

Při psaní grafické části prohlížeče jsme byli postaveni před problém, jaké použít fonty při sázení textu. Požadavek byl snadná čitelnost a škálovatelnost. V zásadě jsme měli možnosti použít již existující fonty — například z X-Window nebo z Ghostscriptu, nebo distribuovat vlastní fonty. My jsme si vybrali vlastní fonty z důvodu přenositelnosti a nezávislosti na ostatních programech a nastavení uživatelova počítače. Dále jsme měli na výběr zda použít vektorové fonty, nebo bitmapové. My jsme si vybrali variantu bitmapových fontů zejména z toho důvodu, že aby bylo možno vektorové fonty dostatečně antialiasovat, bylo by nutné je vygenerovat na mnohem větší rozlišení než je rozlišení našich bitmapových fontů. To by zabralo velmi mnoho času, navíc by se takto vygenerovaná bitmapa musela ještě převzorkovat, což by také zabralo hodně času. K přidání nebo úpravě fontu by uživatel potřeboval speciální typografické nástroje pro práci s vektoro-

vými fonty, takto mu stačí libovolný grafický program nebo scanner. Pokud by někdo chtěl přidat předlohu z knihy, potřeboval by specialisovaný software pro vektorisaci rastrového formátu, takto stačí libovolný grafický program (například Gimp).

Bitmapové fonty jsou součástí binárního souboru, což opět zlepšuje přenositelnost, protože se fonty nemusí hledat v různých adresářích a také se usnadňuje instalace. Fonty jsou uloženy ve velkém rozlišení ve formátu PNG (který podstatně redukuje jejich velikost) a při zobrazování se antialiasují, aby byla zlepšena jejich čitelnost i při malých velikostech (prakticky ozkoušeno: při velikosti 8 pixelů je antialiasovaný font čitelný a font z X je již nečitelný). Pro lepší čitelnost jsou použity fonty computer modern z Ghostscriptu. Uživatel si dále může plynule nastavit velikost písma, což zajisté ocení uživatelé s vadou zraku.

8. Výsledek projektu

Výsledkem projektu je jeden binární soubor, který se spustí a funguje jak v textovém, tak v grafickém režimu. Kód je psán přenositelně, takže ho je možné bez problémů zkompilovat na všech Unixech i na OS/2. V grafickém módu je využit zejména grafický systém X-Window, který je na jmenovaných operačních systémech nejpoužívanější a nejpřenositelnější. To též umožňuje pouštění prohlížeče přes síť na vzdáleném počítači. Dále jsou podporovány i grafické systémy SVGAlib (na Linuxu) a Pmshell (na OS/2).

Veškerý kód je napsán v jazyce ANSI C a to tak, aby byl přenosný mezi platformami. Ke kompilaci doporučujeme kompilátor GCC a program GNU Make, nicméně jiné kompilátory by měly fungovat také. K výrobě překladače a interpretu JavaScriptu byly použity nástroje Bison a Flex. Dále pro snadnější kompilaci a přenositelnost mezi platformami byly použity programy Autoconf a Automake.

Pro textový výstup nebyly použity žádné terminálové knihovny, jelikož jsou nepřenositelné. Proto se výstup zobrazuje pomocí standardních ANSI terminal escape sekvencí s možností zapnutí různých nestandardních rozšíření v menu, například barvy, různé druhy rámečků, tvar kurzoru.

8.1 Javascript

8.1.1 Poznámky k implementaci upcallů a vnitřních funkcí

Tato kapitola obsahuje odchylky implementace od normy javascriptu. Zejména zde najdete seznam věcí, které jsme implementovali jinak, a pak které jsme implementovali navíc oproti normě Javascript 1.1 od Netscape Corporation.

V gramatice jsme museli povolit neukončení statementu středníkem, tj. znejednoznačnění zdrojových textů, umožnění zběsilého mixování operátoru pole a operátoru member (a[ ].b nebo a[ ][ ]) a umožnění otevření komentáře ve zdrojovém textu (<script> <!--). První dvě lumpárny se dokonce vyskytují ve větším než malém množství v příkladech v normě, kterak má správně vypadat a fungovat javascript.

Oproti specifikaci jsme přidali eventové handlery onkeypress, onkeyup, onkeydown u políček pro zadávání textu.

Onchange handler u textového políčka voláme v případě, že políčko ztratí focus a uživatel v políčku něco změnil. Jak se má tento handler chovat jsme ve specifikaci nenašli, proto jsme se řídili implementací v ostatních prohlížečích a používáním handleru na webových stránkách. Byla by možná i interpretace, že se onchange handler má volat vždy, když se něco změní, ale to by odpovídalo handleru onkeypress.

Nastavování textu na stavové řádce v onclick a onmouseover handleru by se podle specifikace mělo provádět pouze v případě, že handler vrátí hodnotu true. My nastavu-

jeme text na stavové řádce vždy, když si skript řekne, neboť by implementace vzhledem k výsledku byla neúměrně složitá. Navíc prohlížeč Netscape 4.51 nastavuje hodnotu stavové řádky také nehledě hodnotu vrácenou handlerem.

V prohlížeči funguje princip **odložené změny URL**. Je to pro zajištění funkčnosti eventových handlerů (například `onclick` u odkazů, `onsubmit` u formulářů a podobně). V těchto případech se musí počkat až doběhne handler a teprve pak se smí jít na jinou stránku (poslat formulář, následovat odkaz). Na jiné URL se tedy nejde dokud v daném kontextu nedoběhnou všechny javascripty. To má i své nevýhody, například když na nějaké stránce javascript běží v nekonečné smyčce a uživatel chce kliknout na nějaký odkaz. V tomto případě odkaz „nebude fungovat". To není chyba, ale vlastnost. „Ruční" změna URL bude fungovat (přes zadání URL do dialogu).

Vzhledem k příliš častému výskytu přistupování ke globálnímu adresnímu prostoru na webových stránkách, jsme implementovali povolení resoluce jmen v hlavním adresním prostoru. Uživatel má možnost v nastavení javascriptu zaškrtnout tuto volbu. Po zapnutí funguje správně přístup k prvkům například objektu `document`, když `document` není uveden, což se na stránkách netriviálně často vyskytuje. Pro toto řešení hovoří i fakt, že některé stránky naopak tuto resoluci vyžadují vypnutou. Norma ani document object model o chování v takovéto situaci nic neříkají, udělali jsme tedy řešení, které nám přišlo jako rozumné a použitelné.

Uživatel má dále možnost povolit/zakázat všechny typové konverze (standardně jsou zapnuty). Tento přepínač umožňuje povolit konverze, které jsou v rozporu s normou. Jedná se zejména o konverze typu undefined na jiný typ. Při povolení všech typových konverzí je povoleno dereferencovat neexistující pole, ale už je zakázáno do něj zapisovat. Tuto vlastnost jsme zabudovali poté, co jsme zjistili, že na mnoha stránkách javascript nechodil z důvodu používání nedokumentovaných proměnných, které zákonitě nejsou z naší strany podporovány a podle normy mají (typicky pro přiřazení nedefinované hodnoty nebo její konverzi) skončit chybou. Jedná se například o proměnné, které jsou definované jen v „normě" javascriptu firmy Microsoft.

Přidali jsme řadu proměnných a funkcí, které některé browsery podporují, ať už se jednalo o přiřazování do objektu `location`, nebo o jeho duplikaci do objektu `window`. Konkrétně se jedná o: `Math.md5()` je nedokumentovaná funkce, která počítá MD5 řetězce zadaného jako argument. Její použití se vyskytuje například na serveru `http://www.post.cz`. Další nedokumentovaná funkce objektu `Math`, kterou jsme implementovali, je `Math.floor()`, jak již název napovídá, vrací celou část floatu. K celému objektu `location` se lze dostat nejen přes `document.location`, přes `window. location`, ale také přímo přes `location`. Do objektu `location` lze přiřadit hodnotu. Přiřazení se chová stejně jako přiřazení do `location.href`, tedy způsobí přechod na jinou stránku. Objekt `location` má navíc metodu `replace`, která akceptuje jeden argument — řetězec s URL. Tato metoda provede to samé, co provede přiřazení do `location.href`. Funkce `Array` se dá zavolat též pod jménem `Object`. Intuice říká, že `Array` by mělo být „schopnější", ale konstruktor `Object` není nikde popsán a když jsme se ho vydali hledat do terénu, došli jsme k názoru, že se ze všeho nejvíc podobá funkci `Array`, pročež jsme tyto dvě ztotožnili. U objektů `Date` a `Image` mohou být některé funkce a proměnné falešné, to znamená implementované, ale nic nedělající.

Nepodporujeme změny rozměrů obrázku, ať už zápisem do proměnných `image.width` a `image.height`, nebo změnou source obrázku. Kdybychom javascriptu povolili změny rozměrů obrázků, musela by se při každé změně přeparsovávat celá stránka, což by jednak vypadalo ošklivě a jednak by bylo velmi pomalé. Zápis do atributů `width` a `height` je zakázán a při změně source obrázku se nový obrázek zobrazí do výřezu o původních rozměrech. Pokud má nový obrázek jiné rozměry je buď doplněn pozadím, nebo oříznut na dané rozměry. Typicky má nový obrázek stejné rozměry jako původní (většinou se javascriptem například mění tlačítka z vymáčklého na zamáčklé a podobně), proto jsme zvolili tuto strategii. Natahování nového obrázku do starých rozměrů by bylo neúměrně

pomalé a zbytečné, zejména v případech, kdy nepříliš zdatný tvůrce stránky nezvládne vytvořit se stejnou velikostí a obrázek se liší rozměrově například o 1 pixel.

Javascript nemá přístup k pozaďovému obrázku. Tento obrázek není vkládán pomocí HTML elementu `img`, ale je atributem celého dokumentu. Navíc kdybychom povolili měnění obrázku na pozadí, musela by se přeparsovat celá stránka, přepočítat fonty a průhledné obrázky, což by bylo velmi náročné. Proto jsme toto nepovolili a javascript o obrázku na pozadí stránky nemá nejmenší ponětí.

Atribut `lowsrc` u obrázků ignorujeme, protože obrázky s nízkým rozlišením nepoužíváme. Stejně tak v javascriptu tento atribut není podporován.

Metody `open` a `close` objektu `dokument` nepodporujeme, neboť používají MIME-streamy, které v prohlížeči nejsou implementovány. Podobně plíživé volby ve funkcích `window.open` a `frame.open` nepodporujeme, protože opět slouží pouze pro účely MIME-streamů.

Handler `onselect` u políčka pro vkládání textu nepodporujeme, protože v políčku se nedá označit text, handler je ignorován. Ze stejného důvodu nefunguje funkce `select()` u textového políčka (funkce `select()` nic neprovede a vrátí hodnotu `undefined`).

`OnUnload` handler nepodporujeme, protože při opuštění stránky je javascript zabit a kontext zlikvidován, tudíž by tento handler postrádal smysl.

Historii v javascriptu podporujeme pouze směrem dozadu, neboť historie dopředu v prohlížeči implementována není.

Funkce pracující se select objektem vrací jen select, kde lze vybrat pouze jednu položku. Ostatní selecty (kde lze vybrat víc položek) se tváří jako checkboxy. To je z důvodu interní representace select objektu v prohlížeči.

V prohlížeči jsme neimplementovali funkci `eval` z toho důvodu, že by bylo potřeba přepsat interpret javascriptu. Problém tkví v tom, že ve funkci `eval` je potřeba vzít argument, spustit na něj znovu lexikální a syntaktickou analýzu a poté ho interpretovat. S lexikální a syntaktickou analýzou by nebyl takový problém jako s interpretací. V interpretu totiž předpokládáme, že v jednom kontextu poběží právě jeden skript. Takto by v jednom kontextu běžely skripty dva, neboť `eval` potřebuje přistupovat do původního stromu (do stejného „adresního prostoru“), a na to není náš interpret připraven. Proto by implementace funkce `eval` vyžadovala veliký zásah. Zotavování z chyb při interpretaci argumentu by bylo velmi nepříjemné. Ač se zdá, že tato funkce nemá valného významu, neboť většina použití se dá napsat jinak, v praxi se vyskytuje vcelku často. Líní a neschopní autoři javascriptů ji používají i na tak triviální operace, jako je například sečtení dvou čísel (kalkulačka napsaná pomocí `eval`). A jelikož takovéto stránky jsou na `eval` většinou založené, javascript na nich v **Links** nefunguje.

Již se nám podařilo navrhnout způsob, jak `eval` implementovat, bohužel jsme neměli dostatek času tyto změny provést a hlavně následně dostatečně kód ozkoušet, proto jsme `eval` před dokončením projektu neimplementovali. Ale hodláme tuto konstrukci implementovat v nejbližší budoucnosti.

Eval jsme z důvodu častého používání pro spojování řetězců prozatím implementovali tak, že pouze spojí řetězce, které dostane. Tím se zprovoznila část stránek, na kterých díky `evalu` javascript nefungoval.

Ze stejného důvodu není implementována funkce `sort`, jejíž implementace je podobná funkci `eval`. `Sort` dostane jako argument funkci na porovnávání prvků, volání této funkce by se provádělo podobně jako zpracovávání argumentů u `eval`, podobně je to se zotavováním z chyb. Ze stejných důvodů také nefunguje implicitní volání funkce `toString` nebo `valueOf`.

Další častá konstrukce, se kterou jsme se setkali na webových stránkách, je funkční volání v member expression (konstrukce typu `document.funkce().element`). Tato konstrukce je v rozporu s gramatikou normy. Ze začátku jsme ji nepodporovali, ale protože se vyskytovala příliš často a našli jsme způsob, jak ji nenásilně implementovat, nakonec jsme ji podpořili.

Asociativní pole také není v normě (v normě 1.1 se pole smí indexovat pouze nezápornými čísly, asociativní pole je v normách novějších než 1.1), ale vzhledem k faktu, že je v praxi hojně využíváno, jsme se tuto konstrukci rozhodli podpořit. Nejprve jsme se domnívali, že bude díky příliš těžko implementovatelné, ale nakonec jsme přišli na způsob, jak asociativní pole implementovat.

Nepodporujeme atribut `prototype`, který je u všech objektů. Tento atribut je normou povolen a přidává do objektu nové metody a atributy (případně s danou hodnotou). Tento atribut jsme neimplementovali, protože by byl problém ho implementovat a jsme toho názoru, že je k ničemu. Tento atribut mají například i objekty `number` a `string`, které implementujeme přímo a s žádnými metodami ani atributy se u těchto objektů nepočítá.

Barvy v objektu `dokument` (například `fgColor`, `bgColor` ...) jsou fake z toho důvodu, že se v prohlížeči nikde neukládají. Pamatují se pouze krátce během formátování dokumentu a záhy po zformátování se zapomenou. Neukládají se do žádných vnitřních struktur, protože v době, kdy se psal formátovač stránky, se o javascriptu neuvažovalo. Proto k nim ani javascript nemůže přistupovat. Napsat, aby javascript mohl měnit barvy na stránce, by vyžadovalo změnu návrhu celého formátovače dokumentů. Nehledě k tomu, že nastavování barev javascriptem je kosmetická záležitost, která nikterak neohrožuje zobrazování stránek ani funkčnost prohlížeče.

Objekt `this` je implementován přesně podle normy Javascript 1.1, tedy odkazuje k současnému objektu, což je buďto hlavní adresní prostor, nebo lokální adresní prostor (v případě, že jsme ve funkci). Správné použití tedy vypadá například takto (výroba lokálních proměnných):

```
function f(){this.a='a';}
```

Leč již jsme se na několika stránkách setkali s použitím, kdy objekt `this` je použit například k identifikaci objektu uvnitř event handleru. Tedy podobně jako kupříkladu v C++. Zde je příklad špatného použití, které je v rozporu s normou Javascript 1.1:

```
<img src="a.jpg" onmouseover="this.src='b.jpg';">
```

Což by mělo vytvořit obrázek „a.jpg“, který se změní na obrázek „b.jpg“, když se naň najede myší.

8.2 Formáty

Links umí zobrazovat obrázky GIF, JPEG, PNG, TIFF a XBM. Obrázky PNG, TIFF a JPEG se dekódují knihovnami `libpng`, `libtiff` a `libjpeg`. Na obrázky GIF a XBM jsme napsali vlastní dekodéry. Prohlížeč díky použitým knihovnám podporuje všechny varianty a vymoženosti formátů JPEG, PNG a TIFF.

8.2.1 Animované GIFy

Prohlížeč zobrazuje z animovaných gifů úmyslně pouze první snímek. Tento postup přináší uživateli značnou úlevu před nepříjemnými reklamami. Většina reklam jsou animované gify. Animace Flash prohlížeč nepodporuje vůbec, a to ze stejného důvodu. V animovaných gifech a Flash animacích v drtivé většině případů není žádná informace nutná pro seriózní práci s webovým obsahem, tudíž tato vlastnost nepředstavuje žádný handicap pro profesionální využití WWW prostoru.

8.2.2 HTML

Prohlížeč podporuje HTML formát verzi 4.0. Prohlížeč nemá implementovány CSS (style sheets). Naše implementace se od normy liší v těchto tagách:

- **OBJECT** tag nepodporujeme, jelikož bychom ne vždy byli schopni jeho obsah zobrazit. Proto místo něj se zobrazuje odkaz (uživateli se zobrazí tlačítko `OBJ`), po jehož stisknutí se zobrazí jeho obsah. Pokud je obsah nějakého konkrétního MIME typu, uživatel má možnost nastavit si externí prohlížeč. Tedy například, pokud bude obsah OBJECTu flash animace a uživatel si nastaví prohlížeč flash animací, po kliknutí na odkaz se automaticky spustí prohlížeč flash animace. Pokud obsahem `OBJECT` tagu je obrázek, obrázek se přímo zobrazí (a tedy tlačítko `OBJ` se již nezobrazuje).
- **LINK**, neboli odkaz v hlavičce dokumentu, se zobrazuje jako odkaz na začátku stránky. Uživatel vidí například: `Link: Next page`, přičemž za „Link:“ je odkaz, který vede na cíl `LINK` elementu.
- **IFRAME**, neboli vložený rám, se též zobrazuje jako odkaz (na místě, kde se objeví tag `IFRAME`. Vložený rám jsme neimplementovali z důvodu, že je z 98 % využíván k reklamním účelům. Podobně jako v předchozím případě, uživateli se zobrazí odkaz například `IFrame: main.html`, na který může kliknout a po jehož odkliknutí se zobrazí požadovaný rám.

Navíc jsme implementovali HTML tag `embed`, který se v případě, že zdroj je obrázek, chová jako tag `img`, v ostatních případech se pouze zobrazí odkaz, že se jedná o tag `embed`. Tento tag je rozšířením HTML 4.0 používaným prohlížeči Netscape i Explorer. Každý výrobce u své implementace rozšíření akceptuje jiné atributy. Naštěstí atributy `src`, `width` i `height` jsou podporovány všemi výrobci a mají i stejný význam. My jsme implementovali tag pouze s těmito třemi atributy.

Rozhodnutí o podpoře tagu `embed` jsme učinili poté, kdy jsme zjistili, že některé webové stránky zobrazují pomocí tohoto tagu obrázky a že tyto obrázky nejsou bez `embed` tagu vidět a stránka vypadá jinak.

V textovém módu se obrázkové mapy (tagy `ISMAP` a `USEMAP`) zobrazují jako tlačítko, po jehož zmáčknutí se zobrazí nabídka, kde si uživatel může vybrat jednotlivé odkazy z mapy obrázků.

Přesměrování v HTML kódu se narozdíl od HTTP redirectu neprovádí automaticky. Pokud se na stránce vyskytuje přesměrování (`<META HTTP-EQUIV=„Refresh" ...>`), v prohlížeči se zobrazí `Refresh:`, za kterým následuje odkaz na příslušnou stránku. Napsat přesměrování automaticky by nebyl problém, ale my jsme zvolili ruční variantu, neboť pro uživatele je v případě automatického přesměrování velmi nepříjemné dostat se zpět. Když jde zpět, refresh ho automaticky přesměruje dopředu a tak stále dokola. Proto uživatel musí vyvolat funkci „historie“, kde složitě vybere položku před přesměrováním, a teprve poté se dostane před přesměrování.

8.3 Ostatní

8.3.1 Cookies

Links plně podporuje cookies až na mírná bezpečnostní omezení. Prohlížeč neakceptuje cookies z nestandardních domén 2.řádu, doména musí být alespoň 3.řádu. Příkladem je cookie z „.domena.cz“, kterou prohlížeč neakceptuje, protože se jedná o doménu 2.řádu. Toto bezpečnostní opatření je z důvodu existence domén jako například „.co.uk“ nebo „.co.jp“, které jsou příliš obecné. Prohlížeč akceptuje cookies z domén druhého řádu pro tzv. standardní domény, jakými například jsou „.net, .org, .com“ a podobné. Z bezpečnostních důvodů **Links** neukládá cookies na disk, narozdíl od ostatních webových

prohlížečů, protože cookies velmi často obsahují citlivé informace, které by mohl někdo nepovolaný z disku přečíst.

8.4 Cache

Links používá celkem 3 cache na soubory:

- **souborovou cache**, kam se ukládají všechny stažené soubory.
- **dokumentovou cache** na již zformátované dokumenty.
- **obrázkovou cache** pro ukládání částečně i úplně dekódovaných obrázků (ukládají se do ní běžící dekodéry obrázků).

Dokumentová a obrázková cache používají strategii LRU k vyřazování dokumentů. Souborová cache používá také strategii LRU, ale dříve vyřazuje velké soubory (zvýhodňuje malé soubory). Cache se po přidání nového souboru a při určování, které soubory vyřadit, prochází na tři průchody. V prvním průchodu se prochází soubory od nejstaršího směrem k mladším a dokud velikost přesahuje určitou mez, soubory se označí k vymazání. Ve druhém průchodu se prochází cache od nejmladších souborů a pokud se nalezne označený soubor, který by šel ještě ponechat v cachi, aniž by byla překročena velikost cache, soubor se odznačí. Nakonec ve třetím průchodu se označené soubory z cache odstraní. Při stahování do souboru (funkce „download“ nebo „stahování“) se soubory do cache neukládají, pokud jejich velikost přesahuje čtvrtinu maximální velikosti cache.

Velikost souborové a obrázkové cache má uživatel možnost nastavit v menu, aby mohl co nejlépe přizpůsobit prohlížeč parametrům svého počítače. Do cache se ukládají již zformátované dokumenty, což zkracuje odezvu při použití cache. Stejně tak nastavení počtu zformátovaných dokumentů v dokumentové cachi je ponecháno na uživateli. Obě konstanty jsou standardně přednastaveny na nízké hodnoty (velikost 1MB pro obrázkovou i souborovou cache a maximálně 5 dokumentů), aby bylo možno prohlížeč provozovat i na systémech s malým množstvím fyzické paměti.

Diskovou cache jsme záměrně neimplementovali, protože si myslíme, že nemá význam a pouze zabírá místo na disku. V typickém případě uživatel spustí browser, dlouhou dobu ho používá a poté ho ukončí. Po nějaké netriviální době ho spustí znovu. Jelikož nové spuštění nastane typicky za dlouhou dobu (za hodiny až dny), většina dokumentů v cachi expiruje a tudíž cache pozbývá na významu. Navíc disková cache je nebezpečná v tom smyslu, že někdo nepovolaný může sledovat, které stránky si uživatel prohlížel (nebo si je prohlíží).

Paměťová cache je agresivní, tedy ukládá všechny dokumenty nehledě na na parametr „nocache“ a ignoruje expirační dobu. Takto jsme cache implementovali poté, co jsme zjistili, že parametr „nocache“ i expirační doba jsou autory stránek hojně zneužívány například pro stahování reklamy. Agresivní cache velmi sníží zatížení sítě při zobrazování dříve již zobrazených dokumentů. Pokud uživatel nechce cache použít, musí si explicitně vyžádat natažení dokumentu stisknutím klávesy CTRL-R (reload).

8.5 Význačné rysy prohlížeče

- **Přenositelnost** na všechny systémy Unix, OS/2 a Cygwin (pod Windows)
- **Zobrazování tabulek, rámů a obrázků**
- **Roletová menu**
- **Záložky (bookmarks)** — uživatel má možnost ukládat si záložky na zajímavé stránky, záložky jsou organisovány v přehledných adresářích. Formát bookmarků na disku je kompatibilní s Netscape Navigatorem.
- **Překlady do mnoha jazyků**

- **Stahování souborů na pozadí**
- **Keepalive connections** — spojení se serverem se udržuje nějakou dobu po stažení dokumentu, neboť se očekává stahování dalších dokumentů. Pak se nemusí pokaždé navazovat spojení znovu a uživatel čeká kratší dobu na stažení stránky.
- **Asynchronní DNS lookup** — při čekání na odpověď DNS serveru (server pro překlad adres) prohlížeč „nezatuhne" jako jiné nejmenované prohlížeče. Uživatel si v podstatě vůbec nevšimne, že se čeká na odpověď serveru.
- **Možnost spouštění externích programů** na neznáme typy dat — uživatel má možnost určit programy, které se mají spouštět na soubory, které **Links** neumí zobrazit: například hudba, video, různé dokumenty (PDF, PostScript, Word) a podobně.
- **Možnost přepínání mezi textovým browserem běžícím na konzoli a nebo grafickým browserem běžícím pod SVGAlib, OS/2, X-Window, AtheOS**
- **Zobrazování stránky již během nahrávání** — jakmile se začne stahovat stránka z Internetu, tak se téměř okamžitě dá číst. Uživatel nemusí čekat, až se stránka celá stáhne, zformátuje a pak teprve zobrazí, jako je to u konkurenčních prohlížečů. Stránka se formátuje průběžně.
- **HTML verze 4.0 bez CSS**
- **Cookies** — bez ukládání na disk, z bezpečnostních důvodů trochu striktnější přijímání.
- **Caching** — ukládání dokumentů do paměti, při návratu na stránku (například tlačítkem zpět) se stránka natáhne z paměti (cache) a nemusí se čekat na síť. Cache nebere ohled na parametry „nocache" a „expire". Cache používá strategii LRU.
- **Javascript** odvozený od specifikace 1.1 firmy Netscape Corporation, document-object model převzat z Netscape 2.0 dle nalezené dokumentace, přidána některá vylepšení.
- **Cache na zformátované dokumenty** — **Links** je mezi prohlížeči jediný, který má cache na zformátované dokumenty. Díky této cachi se po návratu na stránku dokument okamžitě zobrazí a uživatel nemusí zdlouhavě čekat na formátování stránky.
- V případě vadného HTML kódu prohlížeč nespadne.
- Při přeparsování stránky nebo opětovném stažení stránky se neztratí uživatelem vyplněný obsah formulářů.

8.6 Podporované protokoly a formáty

Links podporuje tyto protokoly a formáty:

- **FTP** — dle RFC 959
- **Finger** — dle RFC 1288
- **HTTP** — verze 1.0 (podle RFC 1945) a 1.1 (dle RFC 2068), verze se automaticky přepíná, kterou verzi použít. Některé servery umí jen 1.0, případně obsahují chyby, takže se na nich dá použít pouze 1.0.
- **HTTPS** — podle použité knihovny SSL
- **Javascript** — základní koncept je navržen podle normy javascriptu verze 1.1 od firmy Netscape Corporation. Implementace není úplně přesná, neumí například funkci eval či metodu sort u pole. Většina zásahů je ve směru zobecnění a umožnění interpretace skriptů, které jsou s normou v rozporu. Tato zobecnění byla částečně motivována i tím, že příklady toho, jak má vypadat javascript, v normě a tato

norma si často odporovaly. V javascriptu bylo dále povoleno mnoho „nečistých praktik“, neboť javascript na webových stránkách má (eufemicky řečeno) k normě mnohdy hodně daleko. Odchylky od normy byly většinou implementovány podle ostatních webových browserů, zejména Netscape, verze přibližně 4.51.

- **HTML 4.0 bez CSS** (bez Cascade Style Sheets)
- **JPEG** — dekódován knihovnou libjpeg, podporujeme progresivní nahrávání, podporujeme JPEGy s 8 bity na složku a nepodporujeme JPEGy s 12 bity na složku z důvodu, že libjpeg umí vždy jen jednu variantu. S 12 bitovými JPEGy jsme se na webu ještě nikde nesetkali a běžné JPEGy jsou 8 bitové.
- **PNG** — dekódováno knihovnou libpng, podporujeme gamma korekci, alfa kanál, progresivní nahrávání
- **GIF** — vlastní dekodér, podporuje transparentnost a prokládání. Z animovaných GIFů se zobrazuje pouze první snímek.
- **XBM** — vlastní dekodér, podporuje formát X10
- **TIFF** — dekódován pomocí knihovny libtiff.

Obrázky se počítají v barevné hloubce 16 bitů na složku s minimální možnou degradací pro monitor. Barevnou kalibraci nepodporujeme, protože při zanedbání to není moc vidět, uživatel typicky neví údaje o monitoru a bylo by příliš složité údaje počítat.

8.7 Jazyky

Prohlížeč je lokalisován zhruba do třiceti cizích jazyků. Jelikož neumíme všechny tyto jazyky, byli jsme odkázáni na překlady, které nám poslali nadšení uživatelé. Proto všechny jazyky krom angličtiny a češtiny nepochází od nikoho z týmu autorů. Tyto překlady vznikly v dobách čistě textové verze prohlížeče, s přidáním nových funkcí a nových textů nebylo v našich silách aktualisovat všechny jazyky, proto některé texty v některých jazycích nejsou přeloženy a zobrazují se v angličtině. Záleží tedy pouze na laskavých uživatelích, zda se najdou lidé, kteří provedou aktualisaci.

Links je přeložen do těchto jazyků:

- Angličtina
- Brazilská portugalština
- Bulharština
- Čeština
- Dánština
- Estonština
- Finština
- Francouzština
- Galícijština
- Holandština
- Islandština
- Indonésština
- Italština
- Katalánština
- Litevština
- Maďarština

- Němčina
- Norština
- Polština
- Rumunština
- Ruština
- Řečtina
- Slovenština
- Španělština
- Švédština
- Turečtina
- Ukrajinština

8.8 Platformy

Links běží na těchto platformách v textovém režimu a pod systémem X-Window. U některých operačních systémů jsou v závorkách ještě uvedeny grafické platformy, pod kterými na těchto platformách **Links** běží:

- AIX
- AtheOS (grafické prostředí AtheOS)
- BeOS
- Cygwin pod Windows
- FreeBSD
- FreeMint
- HPUX
- Irix
- Linux (SVGAlib, framebuffer)
- MacOS X
- NetBSD
- OpenBSD
- OS/2 (Pmshell)
- Solaris
- SunOS
- Tru64

Zdrojový balík obsahuje ještě historický port na Win32, ale ten je nestabilní a nepsal ho nikdo z řešitelů projektu. Nechali jsme ho v projektu, kdyby ho někdo v budoucnosti chtěl použít.

8.9 Chyby a problémy programu

8.9.1 Rádoby chyby

Zde je seznam nejčastějších problémů, které nejsou způsobeny prohlížečem **Links**, přestože tak mohou vypadat a uživatelé si na ně stěžují jako na chyby **Links**. Ke každému problému je uvedeno vysvětlení.

- Některé staré libpng neumí zpracovávat určité druhy (ty exotičtější) PNG obrázků, protože v nich chybí funkce pro rozhraní k vlastnostem, které tato PNG mají. Proto **Links** v případě, že je zkompilován s takto vadnou libpng, toto automaticky díky autoconfu pozná a v případě že na takové PNG narazí, spadne s hláškou, aby si uživatel pořídil lepší libpng. Proto je nutno používat rozumně novou libpng.
- Některé knihovny libtiff a libjpeg mohou být zkompilovány bez podpory některých typů obrázků (například TIFF s LZW kompresí, TIFF s JPEG kompresí a podobně), proto se s takovými knihovnami inkriminované obrázky zobrazí jako rozlomený rámeček. Uživatelé si pak stěžují, neboť se domnívají, že to je chyba **Links**.
- Některé vadné stránky postrádají druhou uvozovku za klikacím tlačítkem „Odešli“, takže se toto nezobrazí a uživatelé si stěžují na špatné adrese.
- Některé špatně napsané stránky neobsahují za konstrukcemi typu `&`, `"`, `>`, ` `... středník, což způsobí, že se na stránce textově vypíše příslušné `&` (respektive ostatní). Uživatelé toto neprávem považují za chybu **Links**.
- Některé stránky ze serverů od firmy Microsoft obsahují číselný zápis unikódového znaku, ale uvedené číslo není kódem unikódu. Uživatelé pak neprávem dávají prohlížeči za vinu, že místo jejich oblíbeného znaku zobrazí kaňku.
- Ve SVGAlib je velmi špatně napsán kód pro primitivní operace v 16-barevných módem a 2-barevných módech, takže tyto módy fungují velmi pomalu — může za to SVGAlib a nikoliv **Links**.
- SVGAlib před 1.9.4 musí být suid root a za to **Links** také nemůže. Nová SVGAlib již má v kernelu modul a uživatelský program je pak neškodný.
- libpng zhruba před 1.2.0 obsahovala chybu v MMX optimalizovaném kódu, která způsobovala, že když libpng byla zkompilovaná s optimalizací pro MMX a zobrazila určitý obrázek z testovacího kitu na PNG obrázky, **Links** spadnul na segfault, protože se celý adresní prostor přepsal opakující se konstantní hodnotou.

8.9.2 Problém se SVGAlib

Při přepínání grafické konzole SVGAlib v Linuxu se může stát, že dojde k havárii programu, nebo v horším případě celého systému. To je způsobeno broken-by-design API VT_ACTIVATE v Linuxovém jádře, které způsobuje nespolehlivost grafických aplikací při přepínání konzolí. Toto není jen náš názor, tato myšlenka je potvrzena i následujícím mailem z linux-kernel mailing listu.

```
From Peter Benie <peterb@chiark.greenend.org.uk>
Delivered-To: clock@atrey.karlin.mff.cuni.cz
To: jsimmons@transvirtual.com,
        Karel Kulhavy <clock@atrey.karlin.mff.cuni.cz>
Cc: linux-kernel@vger.kernel.org
Subject: Re: Two rapid VT_ACTIVATE's
In-Reply-To: <Pine.LNX.4.10.10204050924080.21397-100000@www.transvirtual.com>
Organization: Linux Unlimited
Cc:
From: Peter Benie <peterb@chiark.greenend.org.uk>
Date: Sun, 07 Apr 2002 23:42:42 +0100

In article <Pine.LNX.4.10.10204050924080.21397-100000@www.transvirtual.com>
you write:
>
>> I wonder what happens when somebody calls VT-ACTIVATE in close
>> succession.  So that the second one overwrites the first one. Let's
```

```
>> assume the first one is generated by program #1 and the second one
>> by program #2. And some of the programs may be using graphics and
>> some of them text. What will happen? Will screen corruption and/or
>> kernel crash induced by botched videocard registers result?
>
>In theory the vt_dont_switch lock should protect you from this.

vt_dont_switch has nothing to do with it, and is fundamentally broken
for other reasons. (You need a call to atomically activate and lock a
particular VT, not a call to lock whichever VT happens to be selected
at the moment.)

With the VT_ACTIVATE race, what will typically happen is:

#1 calls VT_ACTIVATE to request the VT change
#2 calls VT_ACTIVATE does likewise

#1 calls VT_WAITACTIVE to wait for the change to complete
#2 calls VT_WAITACTIVE does likewise (and overwrites want_console)

#2 returns from VT_WAITACTIVE (there is now no VT change pending)
#1 does not return until the user explicitly changes to the right
console, or the ioctl is interrupted by a signal

You can see this in practice if you start two X servers from xdm.
After a while, xdm sends a SIGTERM to the X server that hasn't started
yet, interrupting the ioctl, but the X server doesn't check the return
value from the ioctl, assumes the VT change completed, and fiddles
with the video card parameters. Sometimes you get a peculiar mixture of
two X servers, sometimes the machine just crashes.

Clearly, the crash is due to a bug in the X server (ignoring the error
from the ioctl), but the VT API is also faulty since the X server
cannot find out that its VT_ACTIVATE was 'lost' by the kernel.

Another API problem in this area is with VT_OPENQRY. An application
calls this to get the next free VT number, then VT_ACTIVATE to swtich
to it, allocating it if necessary.  If two applications do this, they
will both get the same VT number from VT_OPENQRY, and both will then
call VT_ACTIVATE and VT_SETMODE. Both applications believe they
have a VT which has been allocated and locked, despite only having one
VT between them!

Peter
```

8.9.3 Problém s framebufferem a gpm

Při psaní ovladače na linuxovém framebuffer jsme se setkali s mnoha problémy. Nejprve jsme se potýkali s velmi chabou nebo neexistující dokumentací API framebufferu. Když se nám reverse-engineeringem driverů framebufferu pro jednotlivé karty a zdrojáků linuxového kernelu podařilo driver napsat, narazili jsme na další problém — myš.

Jediná možnost, jak napsat čtení myši na framebufferu, je přes knihovnu `gpm`. `gpm` je daemon, který čte myš a umožňuje copy-paste kopírování textu pomocí myši. Klientům — programům — umožňuje číst souřadnice myši ze socketu. Tímto způsobem je realisováno čtení myši v textovém módu. Na framebufferu jinak číst myš nejde. Bylo by teoreticky

možné číst přímo zařízení myši, ale není zaručeno, že na všech systémech bude mít uživatel právo toto zařízení číst. Například v unixové laboratoři na Malé Straně není uživatelům povoleno čtení ze zařízení myši. Navíc by k tomuto řešení bylo potřeba od uživatele zjistit typ myši a napsat ovladače pro všechny protokoly myší.

Další způsob, jak by teoreticky bylo možno napsat ovladač k myši na framebufferu, je upravit již existující gpm a buď ho používat, nebo poslat úpravy autorům, aby je v oficiálně vydali s novou verzí gpm. První řešení se potýká s již zmiňovaným problémem přístupových práv, druhé řešení by bylo lepší, ale ne každý by na svém počítači měl nainstalovánu tuto nejnovější knihovnu a tedy opět bez administrátorských práv by toto řešení bylo k ničemu.

Proto nám nezbylo, než myš na framebufferu napsat s použitím existující knihovny gpm. Problém je ale v tom, že gpm je striktně textové a autoři při vývoji nepočítali s existencí framebufferu. Proto je rozlišení myši omezeno na velikost textové obrazovky. Souřadnice myši (relativní i absolutní) jsou tedy udávány ve znacích, nikoliv v pixelech. Jelikož čtení myši po znacích je příliš hrubé, vyřešili jsme čtení souřadnic tak, že přičítáme relativní pohyb myši, ten ale musí být stejně násoben konstantou, protože by se kurzor pohyboval jinak velmi pomalu. Citlivost myši bohužel nastavuje administrátor „natvrdo“ při spouštění gpm, tady si ji uživatelský program nemůže nastavit sám. Proto jsme zvolili kompromis mezi přesností a citlivostí myši. To ovšem přináší ten problém, že může existovat objekt na obrazovce, na který nepůjde kliknout. Toto nebezpečí bohužel reálně hrozí, leč se proti němu nedá nic dělat. Částečně se dá vyřešit například zvětšením písma na obrazovce nebo zvětšením obrázků.

Abychom tento problém dostupnými prostředky vyřešili, umožnili jsme uživateli pohybovat myší jemně po pixelech klávesami F5 – F8. Pro uživatele, kteří mají administrátorská práva, jsme vytvořili patch do knihovny gpm, který upraví gpm tak, aby se kurzor myši v **Links** pohyboval plynule. Patch je přiložen v distribuci a je určen pro knihovnu gpm 1.20.0, nicméně s jinými verzemi by měl fungovat též (případně po malé úpravě).

gpm též neumožňuje číst kolečko na myši, tedy kolečko nefunguje ani v textovém režimu, ani na framebufferu.

Na framebufferu jsou tedy výše popsané problémy s myší, leč když myš nejde číst lepším způsobem, nedá se nic dělat. Toto řešení ale je, jak se říká, „lepší než drátem do oka“. Framebuffer jsme implementovali i přes tato omezení z důvodu, že někteří uživatelé na něj mohou býti odkázáni a pokud nemají možnost jiného grafického rozhraní, budou si moci prohlížeč pustit alespoň s těmito omezeními.

8.9.4 Jazykové překlady

Links je přeložen do cca 25 jazyků. Tyto překlady krom českého byly provedeny externími autory u výhradně textové verze prohlížeče. Jelikož jsme přidávali do jazyků nové texty a ne všechny jazyky ovládáme, jazyky mohou obsahovat anglické texty. To je v případě, kdy daný text ještě nebyl do jazyka přeložen.

8.10 Srovnání s ostatními prohlížeči

Ostatní browsery mají narozdíl od **Links** řady zajímavých vlastností, jako například náhodné tuhnutí, pomalý start, vadné zobrazování obrázků, nekontrolovatelnost provádění kódu javascriptu. Například Netscape samovolně a naprosto náhodně přestává zobrazovat dokument (zobrazí prázdnou stránku) a poté se musí restartovat, aby byl použitelný. Javascript v Netscapu způsobuje vážné chyby a pády prohlížeče, který se musí restartovat (vypnutím javascriptu tyto problémy zmizí).

Co se týče zobrazování obrázků, prohlížeče mají zejména problémy s průhledností a se zobrazováním obrázků formátu PNG. Například Netscape Navigator 4.51 a nižší zobra-

zuje nejtmavší stupeň šedi jako bílou, což učiní černobílé obrázky nečitelnými. MSIE 5.50 pro změnu špatně zobrazuje alfa kanál, místo pozadí stránky u alfa-kanálovaných obrázků zobrazuje bílou nebo jasně žlutou (zkoušeno na PNG test suite `http://www.libpng.org/pub/png/pngsuite.html`), obrázky PNG pouze s průhledností a definovaným background chunkem zobrazí jako netransparentní s šedým pozadím, což je v rozporu se standardem PNG, který říká, že se background chunk má ignorovat a obrázek se má zobrazovat transparentní. Žádný z nám známých konkurenčních prohlížečů neumí aplikovat na obrázky gamma korekci ani ditherovat obrázky. Prohlížeče mají ošklivé a špatně čitelné písmo, které se dá zvětšovat pouze v omezeném rozsahu. Písmo je závislé na nainstalovaných fontech v X-Window.

Prohlížeče nemají vůbec nebo mají velmi špatně implementované cachování souborů, což „ocení“ zejména uživatelé připojení přes modem (nebo jiné pomalé připojení) nebo uživatelé platící za přenesená data. Například prohlížeč Netscape neustále stahuje soubory ze sítě i když se kupříkladu jde pouze zpět v historii, kdy by uživatel čekal okamžité zobrazení stránky. Pokud prohlížeče mají cache, tak pouze na soubory, dokumenty pokaždé znovu přeformátovávají. Při reloadování stránky prohlížeče nezachovávají obsah formulářů.

Prohlížeče mají překomplikované ovládání, není neobvyklé, když čtvrtinu až třetinu plochy okna prohlížeče zabírá překombinované ovládání. I zkušenému uživateli zabere notnou chvíli zorientovat se v ovládání prohlížečů, když nějaký vidí poprvé, dokonce samotné hledání políčka pro zadání URL je velmi náročné. Uživateli je neustále vnucováno to, co nechce. Prohlížeče si samovolně mění URL (například když se poprvé spustí Netscape a uživatel chce zadat URL). Prohlížeč Opera dokonce uživateli **vnucuje reklamu** ještě než zobrazí jakoukoliv stránku, tato reklama navíc nejde vypnout.

Netscape neumí zobrazovat tabulku, když chybí závěrný `</table>` element. Obsahuje i množství dalších drobnějších chyb, například v případě prázdného políčka tabulky (`<td></td>` nezobrazuje rámeček kolem políčka, nebo při nastavení černého pozadí a bílého textu jsou u formuláře tlačítka, radio tlačítka a zaškrtávací čtverečky zobrazeny na šedém pozadí a teprve když se formulář vloží do tabulky, se elementy formuláře zobrazí správně. Dále neumí zobrazovat tabulky během nahrávání dokumentu — uživatel je nucen čekat, až je celá tabulka načtena a teprve pak tabulku zobrazí, což bývá zejména u velkých tabulek (například nějakých přehledů, ceníků a podobně) nepříjemné. **Links** umí formátovat tabulky i částečně nahrané. Dokument (včetně tabulek) se průběžně během nahrávání přeformátovává.

Asi největší problém mají prohlížeče s javascriptem a bezpečností. Většinou stačí velmi krátký a velmi triviální skript, který dokáže prohlížeč zruinovat. Testovali jsme prohlížeče Mozilla 6.0, IE 5.50, Netscape Communicator 4.77, Opera 6.01 (pro Linux i pro Windows), Konqueror.

```
<html><body>
<script language=javascript>
while(1)alert('Jen si pořádně zaklikej!');
</script>
</body></html>
```

Tento skript nevydržel žádný nám dostupný prohlížeč. Všechny prohlížeče se staly neovladatelnými. Některé zběsile vytvářely nová a nová okénka, některé čekaly na uživatelovo odklepnutí. Ale všechny testované prohlížeče bylo nutno po tomto skriptu pro opětovnou použitelnost restartovat.

Links u těchto skriptů netrpí nejmenšími obtížemi, neboť uživatel má u každého dialogu vytvořeného javascriptem možnost interpretaci všech javascriptů ukončit. Tedy se

objeví alert a po stisku „Ukončit skript“ může uživatel dále pokračovat v práci s prohlížečem.

```
<html><body>
<script language=javascript>
a='a';
while(1)a+=a;
</script>
</body></html>
```

Tento skript provede útok na paměť neustálou alokací. Neútočí nejen na prohlížeč, ale dokonce na uživatelův počítač. S prohlížečem Netscape 4.77 na Linuxu způsobí uživateli nepříjemnou čtvrthodinku rozsvíceného disku a **velmi** pomalých reakcí počítače. Ostatní prohlížeče též spadly nebo přestaly reagovat a začaly masivně zatěžovat systém, avšak u prohlížeče Netscape 4.77 byly důsledky nejfatálnější.

V **Links** tento skript brzy skončí, neboť vyčerpá paměť přidělenou javascriptu. Uživatel má možnost tento limit na paměť v menu nastavit.

```
<html><body>
<script language=javascript>
while(1)document.location="furt-dokola.html";
</script>
</body></html>
```

Výše uvedený skript předpokládá, že bude uložen v souboru `furt-dokola.html`. Jediný testovaný prohlížeč, který po interpretaci tohoto skriptu nespadl ani nepřestal být ovladatelným, byl prohlížeč Opera na operačním systému Linux, který po lehkých obtížích byl dále použitelný.

Links opět tento skript nečiní potíže, neboť při každé změně URL vyvolané javascriptem je uživatel dotázán, zda si přeje změnu povolit nebo zakázat. Uživatel má opět možnost v dialogu okamžitě ukončit interpretaci všech skriptů. Navíc pokud je URL stejné jako URL aktuální stránky, nic se neprovede. V tomto případě se tedy nebude nic dít. Útočník může toto opatření ovšem jednoduše obejít, že vytvoří dva skripty, které na sebe navzájem budou odkazovat. V tom případě bude při každé změně URL uživatel dotázán a tedy bude mít možnost „útok“ ukončit.

```
<html><body>
<script language=javascript>
while(1)window.open("okno-furt.html");
</script>
</body></html>
```

Tento skript je lehkou obměnou předchozího, předpokládáme, že skript je uložen v souboru `okno-furt.html`. Rozdíl je pouze v tom, že skript otevírá nová a nová okna, čímž zahltí systém, nebo přinejmenším učiní prohlížeč nepoužitelným. Opět ani jeden z testovaných prohlížečů neustál. V případě IE měl operační systém Windows 2000 vážné problémy s tolika okny, což zanechalo trvalé následky na spuštěném Task manageru, který nešel zavřít, neboť jeho okno se začalo zobrazovat bez horní čtvrtiny.

Našemu prohlížeči skript opět nečinil problémy, neboť před každým otevřením (nebo i zavřením) okna je uživatel dotázán na potvrzení. Tedy se zobrazilo okénko, zda si uživatel přeje otevřít nové okno a uživatel měl opět možnost ukončit interpretaci skriptů.

```
<html><body>
<script language=javascript>
function f(){f();}
f();
</script>
</body></html>
```

Nekonečná rekurse je v novějších prohlížečích již ošetřena. Prohlížeče Opera a Mozilla nic neudělaly, ostatní prohlížeče (Netscape, IE, Konqueror) téměř okamžitě spadly.

Pokud se tento skript spustí v **Links**, záhy se objeví dialog, že javascript překročil limit maximálního zanoření funkcí, a skript je násilně ukončen. Limit si uživatel opět může dle svých potřeb nastavit v menu.

Uvedené potíže prohlížečů považujeme za **velmi závažné** bezpečnostní díry. Jsme toho názoru, že toto by se **v žádném případě** korektnímu prohlížeči nemělo stát. S touto filosofií jsme při navrhování javascriptu již od začátku počítali a proto jsme prohlížeč proti všem útokům ze strany javascriptu obrnili. Javascript je striktně napsán s cílem, aby uživatel měl za všech okolností plnou vládu nad skriptem.

8.11 Použité knihovny

Při psaní projektu jsme se snažili používat co nejméně cizích knihoven, aby kód byl co nejvíce přenosný a aby uživatel nemusel na svůj systém nejprve instalovat megabyty rozličných knihoven, když chce používat **Links**. Z vlastní zkušenosti jsme se přesvědčili, že různé verze knihoven jsou mezi sebou navzájem nekompatibilní (i zpětně). Knihy také obsahují různé chyby a bezpečnostní díry, které by se takto zavlekly do prohlížeče **Links**.

Proto jsme použili pouze knihovny pro dekódování obrázků (jpeg a png) a pro práci s protokolem SSL. Tyto knihovny jsou velmi rozšířené a poměrně stabilní a vzhledem ke složitosti problémů, které řeší, by se nevyplatilo psát vlastní kód, který by mohl obsahovat více chyb než časem a praxí prověřený kód knihoven. Navíc psaní těchto knihovních funkcí by vyžadovalo mnoho času studia dokumentací popisující tyto standardy, nemluvě o času nutném k testování správnosti implementace.

V textovém režimu nejsou potřeba žádné speciální knihovny a stačí pouze standardní knihovna `libc`. Ke kompilaci v grafickém režimu jsou nutné ještě tyto knihovny:

- `libpng, libjpeg, libtiff, zlib` — pro práci s obrázky
- `libvga` — pro práci s Linux SVGAlib, nebo
- `libX11` — pro práci s X-Window systémem

Následující knihovny jsou nepovinné, leč doporučené. Bez nich nebudou fungovat u knihovny uvedené funkce prohlížeče.

- `libssl, libcrypto` — podpora protokolu SSL
- `libgpm` — podpora klikání myší v textovém režimu na systému Linux

8.12 Vývoj a testování

8.12.1 Prostředí používané k vývoji

Links byl vyvíjen většinu času na operačním systému Linux, část (některý kód Mikuláše Patočky) byla vyvíjena na OS/2. Ke kompilaci jsme používali zejména kompilátor GCC (převážně verze 2.95.x a 2.96.x) a nástroj GNU Make (verzi 3.79). K ladění jsme

používali zejména program GDB, dále pak ladící výpisy a logy. Veškerý kód byl psán v editoru VIM. Mezi podpůrné programy, které jsme při vývoji použili patří Autoconf (verze 2.13) a Automake (verze 1.4). Parser javascriptu byl napsán s pomocí nástrojů Flex (verze 2.5.2) a Bison (verze 1.24 a 1.28). Rozhraní X-Window bylo vyvíjeno zejména na Linux XFree86 verze 3.3.6 a 4.0.2. Rozhraní SVGAlib bylo testováno se SVGAlib 1.9.x a 1.4.x. Fonty byly zpracovávány programy GIMP a ImageMagick.

8.12.2 Testování přenositelnosti

Přenositelnost programu jsme testovali na Unixech a strojích dostupných v počítačových laboratořích a na našich (a námi dostupných) počítačích. Testy na Alphě jsme prováděli přes účet na www.testdrive.compaq.com.

- Linux na x86
- Linux na Sparc64
- FreeBSD na x86
- OpenBSD na x86
- NetBSD na x86
- OS/2 na x86
- Cygwin pod Windows
- BeOS na x86
- Atheos na x86
- Solaris na Sparc64
- Irix na SGI
- Linux na Alphě,
- Linux na PowerPC
- MacOS-X na PowerPC
- Digital Unix na Alphě
- Tru64 na Alphě
- FreeBSD na Alphě
- NetBSD na Alphě
- OpenBSD na Alphě
- Linux na IA64

Ostatní deklarované podporované operační systémy jsou bohužel jen z doslechu, neboť jsme neměli možnost je otestovat. Dozvěděli jsme se je od uživatelů, kteří nám nadšeně psali, že **Links** běží i na takovýchto systémech.

Zkoušeli jsme kompilovat překladači:

- GCC 2.7.2.x
- EGCC 1.0.3
- GCC 2.95.x
- GCC 3.0.4
- GCC 2.96.x
- GCC 2.95 — kompilovat nešlo, protože kompilátor obsahuje příliš mnoho chyb (aby šel program zkompilovat musely se vypnout všechny optimalisace).

- WorkShop Compilers 4.2
- MIPSpro Compilers 7.2.1
- Apple Computer, Inc. version GCC-932.1 (založené na GCC 2.95.2)
- Sun WorkShop 6 update 1 C 5.2 2000/09/11
- Compaq C V6.3-129 (dtk) on Digital UNIX V4.0G (Rev. 1530) Compiler Driver V6.3-126 (dtk) cc Driver
- Compaq C V6.4-014 on Compaq Tru64 UNIX V5.1A (Rev. 1885) Compiler Driver V6.4-215 (sys) cc Driver

8.12.3 Testování programu, hledání chyb a optimalisace

Při vývoji jsme použili metodu regresivního testování kódu. Tato metoda je například s úspěchem využívána při vývoji kompilátoru GCC. V našem případě spočívá v testování prohlížeče a při jakémkoliv problému (pád, memory leak, zacyklení, přepsání zásobníku či jakákoliv jiná chyba) zjištění přímé příčiny a uložení podmínek, při kterých problém nastal. Tedy uložení stažení a uložení HTML stránky na disk a uložení kláves, které byly stisknuty. Poté je možno kdykoliv opět vyvolat všechny situace, kdy nastal nějaký problém či chyba. To umožňuje kontrolu, že daná chyba byla doopravdy odstraněna a že pozdějšími úpravami kódu nebyla chyba zavlečena znovu. Stačí pustit automatizovaný skript, který testuje prohlížeč na problematických vstupech.

Tento postup jsme zavedli poté, co jsme zjistili, že některé chyby se opakují. Prohlížeč jsme testovali velmi intensivně, neboť jsme během vývoje nepoužívali prakticky žádný jiný prohlížeč k zobrazování webových stránek.

K odstraňování memory leaků a střílení do paměti jsme používali vlastní technologie. Napsali jsme obaly kolem funkcí malloc, free, realloc, ... Tyto funkce kolem naalokované oblasti vytvoří rudou zónu a uloží informace, na kterém řádku byla paměť naalokována a kolik bytů bylo naalokováno. Do neinicialisované paměti se uloží vzorek, aby se dříve přišlo na používání naalokované, ale neinicialisované paměti (vzorek většinou vyvolá chybu nebo pád programu). Při uvolňování paměti se testuje konzistence rudé zóny, uvolněná paměť se vyplní předem daným vzorkem a na konci programu se zjišťuje, zda se uvolnily všechny bloky paměti.

Tímto mechanismem se nám podařilo velmi účinně detekovat a odstraňovat memory leaky, zápisy mimo naalokovanou oblast (například přetečení pole) a zjišťovat používání nenainicialisované či již odalokované paměti. K tomuto účelu již sice existuje nástroj — knihovna `Electric fence`, ale tato knihovna je velmi neefektivní a nepoužitelně zpomaluje program a také do programu zavléká chyby, které by bez použití této knihovny nevznikly. Proto jsme tuto knihovnu vesměs nepoužívali.

Pro dosažení maximálního výkonu a nejoptimálnějšího kódu (zejména u grafických rutin) jsme používali technologii profilování kódu. Program jsme zkompilovali s podporou profilovacích informací, poté jsme prohlížeč intensivně testovali na graficky velmi náročných vstupech. Pomocí programu `gprof` jsme pak zjistili, kolik času se stráví v které funkci a jak dlouho která funkce trvá. Tak jsme získali velmi cenné informace pro následné intensivní ruční optimalisování kódu založené na znalosti překladačů a strojového kódu. Tímto způsobem se nám podařilo velmi účinně zoptimalisovat kód.

8.13 Doporučené nástroje ke kompilaci

Při psaní interpretu javascriptu jsme použili nástroje `Flex` a `Bison` pro výrobu automatů pro lexikální a syntaktickou analýzu. Tyto nástroje nejsou potřeba při kompilaci, protože v distribuci dodáváme již výstupy z těchto nástrojů — zdrojové texty automatů v jazyce C.

Ke kompilaci je potřeba program `Make` a libovolný kompilátor jazyka ANSI C (doporučujeme GCC).

Pro zajištění přenositelnosti jsme užili nástrojů `Autoconf` a `Automake`. Tyto nástroje slouží k vygenerování souboru `Makefile`. Programy `Autoconf` a `Automake` vytvoří platformně nezávislý skript `configure`, který se spustí při instalaci a který vygeneruje soubor `Makefile`. Nástroje `Autoconf` ani `Automake` nejsou potřebné při instalaci, neboť dodáváme již vygenerovaný skript `configure`.

Dále jsme si vytvořili několik skriptů pro generování fontů, překladů a tabulek znakových sad. Tyto skripty by měly být přenosné na systémech Unix, ale opět nejsou potřeba při instalaci, protože v distribučním balíku jsou již těmito skripty vygenerované zdrojové soubory jazyka C.

8.14 Kód přejatý od jiných autorů

Všechny jazykové překlady s výjimkou angličtiny a češtiny nepsal nikdo z týmu autorů. Překlady jsou přejaty od nadšenců, kteří nám je poslali.

9. Závěr

Myslíme si, že projekt splnil svůj účel, protože nejen, že se nám podařilo napsat kvalitní webový prohlížeč a naučili jsme se a ozkoušeli jsme si spoustu zajímavých věcí, ale hlavně jsme se naučili skupinové komunikaci, specifikovat rozhraní a psát dokumentaci. Což si myslíme, že byl hlavní cíl tohoto projektu.

Projekt nám přinesl zdokonalení se v programování v jazyce C, poznání, jak se programovat má a jak se programovat nemá. Naučili jsme se, že výsledný kód má běžet co nejrychleji, nemá obsahovat žádné chyby, má dodržovat normy a standardy, kód má být portabilní, funkce programu se mají omezit jen na nezbytně nutné minimum dostačující k efektivní práci uživatele. Program má být monolitický a heterogenní. Má se programovat s vědomím, že jakákoliv chyba nalezená v programu může být považována za totální selhání programu.

Při programování by se nemělo podléhat creeping featurismu (creeping featurism znamená nabalování nových funkcí na stávající program jako sníh na sněhovou kouli), nemá se programovat s chybami tak, že se program napíše chybně a pak se na základě bugreportů opravuje. Nemá se programovat v rozporu s RFC a standardy, prosazovat metodiku a strukturovanost na úkor rychlosti a efektivity. Neměla by se prosazovat čitelnost kódu na úkor rychlosti, obcházet chyby v jiných knihovnách a programech na úkor jednoduchosti, spolehlivosti nebo rychlosti. Program nemá vykonávat zbytečné operace, které nic neprovádějí. Data se nemají filtrovat přes vrstvy byrokratických rozhraní a to ani pod záminkou metodičnosti a/nebo strukturovanosti návrhu nebo programu nebo čitelnosti kódu. Nemají se připouštět chyby v programu pod záminkou, že je vestavěné samotestovací mechanismy odhalí nebo odstraní. Nemá se programovat odtažitě od fungování reálného počítače. Do programu se nemají přidávat nové funkce v případě, že není bezchybný. Programátor nemá ukončit programování v okamžiku, kdy kód funguje, ale má si ho po sobě v maximální možné míře překontrolovat. Programátor nesmí připustit existenci testerů, kteří budou program testovat, nesmí brát na lehkou váhu dopad případných chyb v kódu. Program se nemá při programování uspěchávat a programátor nemá být nucen zkracovat time to market.

9.1 Problémy při vývoji

Pří vývoji projektu jsme zjistili, že poctivost se hrubě nevyplácí. Konkrétně se jedná o dodržování standardů. Prohlížeč sice funguje přesně podle RFC a specifikací protokolů,

ale jelikož 90 % webových stránek je napsáno špatně, mnoho serverů nedodržuje protokoly, tak se často stává, že na takovýchto stránkách prohlížeč zobrazí méně než ostatní prohlížeče, které jsou nestabilní a standardy nedodržují. Nicméně jsme se rozhodli i nadále se řídit standardy, neboť jedině tak se dá principiálně dosáhnout rozumné interoperability a řešení spočívající v emulaci chyb ostatních programů se brzo ukázalo jako neschůdné, neboť chyby v ostatních programech si brzo po zavedení takovéhoto přístupu začnou navzájem odporovat.

Nejvíce se tento problém projevil asi při vývoji interpretu javascriptu. Protože implementace javascriptu od různých firem se velmi liší a autoři prohlížečů i autoři stránek pramálo dodržují standardy.

10. Plány do budoucna

Chtěli bychom **Links** vyvíjet i nadále, protože si myslíme, že vždycky je co zlepšovat. O tom nás rovněž přesvědčují reakce části našich uživatelů v mailing-listu (`links-list@linuxfromscratch.org`). Například bychom mohli podpořit CSS v HTML, plovoucí objekty, případně rozšířit javascript o nové konstrukce, aby zobrazoval více špatně napsaných stránek.

V nejbližší budoucnosti hodláme v javascriptu implementovat konstrukci `eval`, odalokovávání nepotřebných částí stromu za běhu, eventuálně kompresi zdrojového kódu udržovaného v paměti (například kód stejného znění udržovat pouze jednou). Mimo to chceme rozšířit stávající document object model například o `document.all`, `document.scripts` a další.

11. Použité materiály

11.1 Javascript

1) zápisky z Datových struktur
2) zápisky z Překladačů
3) zápisky z Pravděpodobnostních algoritmů
4) Proceedings of the SIGPLAN 82 Symposium on Compiler Constructions, Vol. 17, Num. 6
5) info flex, info bison
6) norma Javascript 1.1 od Netscape Corporation, Javascript 1.3
7) norma ECMA–262
8) konsultace s Mgr. Marešem, Doc. Sgallem, Mgr. Bednárkem a kolegou Hubičkou
9) A. Motwani, P. Raghawan: Randomized algorithms
10) Aho, Sethi, Ullmann: Compilers: Principles, Techniques and Tools
11) K. Mehlhorn: Data structures and algorithms
12) David Gries: Kompilátory číslicových počítačov

11.2 Grafika

1) konsultace s RNDr. Pelikánem
2) Josef Pelikán: Pokročilá 2D počítačová grafika (studijní texty k přednášce na MFF UK)

3) Josef Pelikán: Počítačová grafika 1 (studijní texty k přednášce na MFF UK)
4) Charles Poynton: Gamma FAQ
   (*http://www.informap.net/~poynton/GammaFAQ.html*)
5) Charles Poynton: Color FAQ
   (*http://www.informap.net/~poynton/ColorFAQ.html*)
6) normy a standardy: JFIF 1.02, ITU T.81, CCIR Recommendation 601, RFC 2083, ISO DIS 10918-1, IEC 61966-2-1, ISO 9241

11.3 Grafické drivery

1) Adrian Nye: Xlib Programming Manual
2) konsultace s autory framebufferu s SVGAlib
3) zdrojové texty Linux kernelu
4) konsultace s Mgr. Martinem Beranem

11.4 Zásluhy

Zde je seznam lidí, kteří se nějakým způsobem podíleli na vývoji prohlížeče. Zejména se jedná o překlady do cizích jazyků. Svým způsobem se na projektu podílela i spousta dalších lidí, kteří nás upozorňovali na různé chyby, ti zde uvedeni nejsou, neboť by se na tyto stránky nevešli. To ovšem nic nemění na jejich zásluhách při testování prohlížeče.

Tímto bychom tedy chtěli poděkovat všem, kteří se jakkoliv na vývoji prohlížeče podíleli, ať už testováním, hlášením či opravováním chyb, překladem do cizího jazyka, vlastním kódem, nebo jen posíláním nápadů, co zlepšit.

Unai Uribarri	Historie
Uwe Hermann	Manuálová stránka, přepínač příkazové řádky „-version“, otvírání odkazu v novém xtermu
Alexander Mai	Podpora pro xterm pod OS/2, opravení includů pro AIX, aktualizace manuálových stránek
Dakshinamurthy Karra	přenesení na Win NT, ukládání goto history
Oleg Deribas	Titulek okna a podpora clipboardu v OS/2
Arkadiusz Sochala	Polský překlad
Dmitrij M. Klimov	Rámečky v KOI8-R, Ruský překlad
Jurij Raškovskij	Aktualizace ruského překladu
beckers	Německý překlad
Armon Red	Islandský překlad
Wojtek Bojdøl	Aktualizace polského překladu
Serge Winitzki	Aktualizace ruského překladu
Aurimas Mikalauskas	Litevský překlad
Martin Norback	Švédský překlad
Jimenez Martinez, Angel Luis, David Mediavilla, Ezquibela	Španělský překlad
Suveg Gabor	Maďarský překlad
Gianluca Montecchi	Italský překlad
Sergej Boruševskij	No-proxy-for, Ctrl-W doplňování, SSL
Fabrice Haberer-Proust	Francouzský překlad
Cristiano Guadagnino	Aktualizovaný italský překlad
Fabio Junior Beneditto	Překlad do brazilské portugalštiny
Kaloian Doganov	Bulharský překlad
Baris Metin	Turecký překlad

Dmitrij Pinčukov	Ukrajinský překlad
Taniel Kirikal	Estonský překlad
zas@norz.org	Aktualizovaný francouzský překlad
Alberto García	Galícijský překlad
Radovan Staš	Slovenský překlad
Marco Bodrato	Podpora Twintermu
Kaloian Doganov	Aktualizace bulharského překladu
Olexander Kunytsa	Aktualizace ukrajinského překladu
Mediavilla David	Aktualizace španělského překladu
Simos Xenitellis, Alejandros Diamandidis	Řecké kódové stránky a překlad
Stefan de Groot	Holandský překlad
Carles	Katalánský překlad
Ionel Mugurel Ciobc	Rumunský překlad
Petr Baudiš	Používání „imgtitle“ pokud není přítomen „alt“, přidání tagu „LISTING“, aktualizace manuálové stránky
Muhamad Faizal	Indonéský překlad
Peter Naulls	Podpora pro RiscOS
Jonas Fonseca	Dánský překlad
Miroslav Rudišin	Aktualizace slovenského překladu