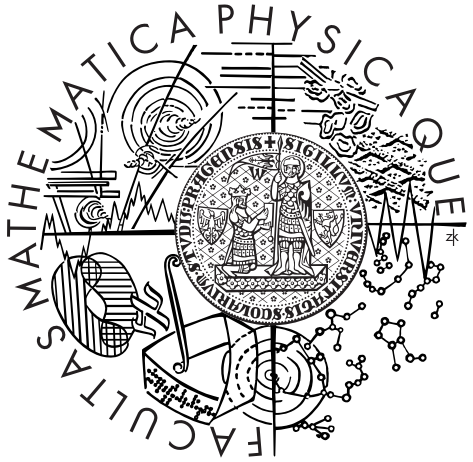
Univerzita Karlova v Praze
Matematicko-fyzikální fakulta

BAKALÁŘSKÁ PRÁCE



Petr Sušil

Anotátor programů v jazyce C

Katedra aplikované matematiky

Vedoucí bakalářské práce: Mgr. Martin Mareš

Studijní program: informatika, obecná informatika

2006

Mé poděkování patří hlavně Mgr. Martinu Marešovi, za jeho odborné vedení mé práce a za podnětné připomínky jak k funkčnosti, tak implementaci programu.

Též bych chtěl poděkovat Milanu Strakovi za rady k implementaci a Janě Kravalové za korektury textu.

Nemalé poděkování patří i mým rodičům za jejich podporu a pochopení při vytváření této práce.

Prohlašuji, že jsem svou bakalářskou práci napsal samostatně a výhradně s použitím citovaných pramenů. Souhlasím se zapůjčováním práce a s jejím zveřejňováním.

V Praze dne 27.5.2006 Petr Sušil

Obsah

1. Úvod 5
1. 1. Přehled funkcí 7
1. 2. Podobné projekty 8
1. 2. 1. Ctags, Etags 8
1. 2. 2. Cscope 8
1. 2. 3. GNU GLOBAL 9
1. 2. 4. Cedet 10
1. 3. Sparse 10
2. Používání programu 12
2. 1. Instalace 12
2. 2. Zapojení cspotu do Makefile 12
2. 3. Plugin pro Emacs 12
2. 4. Seznam příkazů cspotu 13
3. Struktura a správa databáze 18
3. 1. Struktura databáze 18
3. 1. 1. Stránkování databáze 18
3. 1. 2. Hlavička databáze 18
3. 1. 3. Tabulky databáze 18
3. 1. 4. Stránky hašovací tabulky 19
3. 2. Položky databáze 19
3. 2. 1. Stringy 19
3. 2. 2. Záznamy 20
3. 3. Implementace operací nad databází 20
3. 3. 1. Průběh budování databáze 20
3. 3. 2. Ukládání do databáze 21
3. 3. 3. Implementace dotazu 21
3. 3. 4. Update databáze 22
3. 4. Optimalizace 23
4. Implementace databáze 25
4. 1. Paměťově mapované soubory 25
4. 2. Hašování 26
5. Literatura 30

Název práce: Anotátor programů jazyka `C`

Autor: Petr Sušil

Katedra (ústav): Katedra aplikované matematiky

Vedoucí bakalářské práce: Mgr. Martin Mareš

E-mail vedoucího: *mj@ucw.cz*

Abstrakt: Cílem práce je vytvořit program pro usnadnění orientace ve zdrojových kódech malých i velkých projektů v jazyce `C`. Vzniklý program by měl pokrývat základní sadu funkcí pro vyhledávání informací ve zdrojových kódech a poskytnout plugin do editoru Emacs pro další zefektivnění a zjednodušení programování. Na rozdíl od ostatních podobných projektů zná `cspot` sémantiku jazyka `C`, což z něj činí cenného pomocníka. Řešena je též otázka rychlosti a efektivity programu, update databáze a rychlost dotazu. Součástí práce je též popis datových struktur použitých v implementaci.

Klíčová slova: jazyk `C`, procházení zdrojových kódů, Sparse, Emacs

Title: Annotator of `C` source code

Author: Petr Sušil

Department: Department of Applied Mathematics

Supervisor: Mgr. Martin Mareš

Supervisor's e-mail address: *mj@ucw.cz*

Abstract: The aim of this thesis is to create a program for browsing `C` source codes and facilitate searching in them. The program should cover basic search functions and provide a plugin for the Emacs editor. In contrast to other similar project `cspot` knows well semantics of the `C` language, which makes it a great assistant. The speed and effectivity of program was being considered throughout its design as well as possibility of update of an auxiliary database. The thesis contains the description of the used data structures as well.

Keywords: `C` language, source browser, Emacs, Sparse

1. Úvod

Jazyk `C` vznikl již v 70. letech 20. století. Jeho zápis je velmi jednoduchý, což zajistilo jeho velkou popularitu. Vznikl jako vyšší programovací jazyk, aby oddělil programátora od platformy a zajistil tak větší přenositelnost zdrojového kódu. Pro každou platformu byl vytvořen vlastní překladač jazyka `C` a zdrojový kód napsaný na jedné platformě mohl být přenesen, přeložen a spuštěn kdekoli jinde (pokud pro danou platformu překladač existoval). Napsaný kód lze efektivně přeložit do strojového kódu často stejně efektivně, jako by byl program napsaný v samotném assembleru. I přesto je kód snadno čitelný. Tento jazyk je vhodný i pro velké projekty. Například jádro operačního systému Linux je téměř celé napsané právě v jazyku `C`. U velkých projektů se však přehled nad zdrojovým kódem snižuje, a to i u tak přehledných jazyků, jako je `C`. Je tedy šikovné mít nástroj, který by znal sémantiku jazyka `C` a dokázal ve zdrojových kódech rychle vyhledávat, a tím ušetřil práci a čas programátora.

Cílem této bakalářské práce je vytvoření programu `cspot` schopného vyhledávat informace ve zdrojových kódech rozsáhlých projektů. Program pro svoji práci používá databázi, což jej výrazně zrychluje. Databáze je přenositelná mezi všemi operačními systémy, na nichž `cspot` běží, a není tedy nutné ji vytvářet opakovaně na každé platformě. Program `cspot` podporuje mnoho příkazů pro vyhledání v databázi. Mezi tyto příkazy patří výpis použití funkce, makra či globální proměnné, výpis viditelných identifikátorů na dané pozici aj.

Nespornou výhodou programu `cspot` oproti jiným programům je znalost sémantiky jazyka `C` a tím možnost rozlišení, jaký význam má použití identifikátoru v daném místě kódu a jakou funkci identifikátor ve zdrojovém kódu plní. Mnoho programů totiž pouze načte text, pomocí velmi obecného parseru extrahuje identifikátory ze zdrojového kódu a poznamená si jednotlivá místa jejich použití. Výsledkem je velmi obecná databáze, použitelná prakticky na libovolný programovací jazyk. Pro konkrétní programovací jazyk je ale možné ze zdrojového kódu vyčíst mnohem více informací, které jsou často důležité při odhalování chyb v programu i jeho optimalizaci.

Překlad zdrojových kódů v jazyce `C` se skládá ze dvou částí, z preprocessingu a samotného překladu zdrojového kódu. Při preprocessingu dochází pouze k náhradám v textu. Díky nim je možné zadat, které části zdrojového kódu mají být skutečně překládány. Je možné automaticky vygenerovat identifikátory, nastavit proměnné apod. Použitím obecného parseru, který preprocessingu nerozumí, nemůžeme tyto změny ve zdrojovém kódu detekovat, a tedy ani dostupné informace vložit do databáze.

Program `cspot` platí za schopnost pochopení kódu svoji daň tím, že parsování zdrojových kódů je pomalejší než u obecných parserů. Abychom se vyhnuli zby-

tečnému opakování parsování, vhodnou implementací databáze umožníme provádět rychlý update a při změně zdrojových kódů parsujeme pouze ty, které se od předchozího updatu změnily. Programy používající obecný parser takový update většinou nepodporují, neboť i na středně velkých projektech trvá budování jejich databáze jen několik sekund. Pro velké projekty může její vytváření trvat i několik minut, a pak může `cspot` pracovat rychleji. Často se totiž při updatu mění jen několik souborů.

Program `cspot` je pevně vázán na jazyk `C`. K parsování zdrojových kódů tohoto jazyka používá sparse. Sparse [23] je sémantický parser zdrojových kódů napsaných v jazyce `C`. Pro jeho začlenění do programu `cspot` jsem mírně upravil jeho zdrojový kód, který je volně dostupný ke stažení pod licencí Open Software License. Nicméně `cspot` a sparse jsou odděleny tak, aby nebyl problém nahradit sparse za jiný parser, třeba i pro jiný programovací jazyk, a tím funkci programu rozšířit.

Neboť cílem projektu je usnadnění vyvíjení aplikací, je vhodné, aby existoval nějaké rozšíření pro známý textový editor. Nejznámější a nejsnadněji rozšiřitelným je bezpochyby Emacs. Proto jsem vytvořil jednoduché rozšíření, které umožní efektivní vyvíjení programů v tomto prostředí. Psal jsem jej v elispu, neboť většina rozšíření Emacsu je napsaná právě v něm. Existují i rozšíření Emacsu, díky nimž je možné psát pluginy v Perlu či Pythonu. Ty ale běžně nejsou nainstalované a přenositelnost by byla zbytečně zhoršena.

Program `cspot` umožňuje při budování databáze též ukládat všechna varování, které sparse při svém běhu generuje. Díky schopnosti správně nastavit makra a adresáře, kde hledat includované hlavičkové soubory, tedy může být výrazným pomocníkem i při používání samotného sparse.

Při vývoji `cspotu` bylo potřeba přistoupit k řadě optimalizací, neboť přímočará implementace by vedla k dlouhému vytváření databáze. Vyhledávání v databázi bylo též optimalizováno, a to hlavně kvůli strojům s nedostatkem operační paměti. Databáze je implementována pomocí rozšiřitelného hašování. Nicméně tato metoda vyžaduje splnění předpokladů, které nedokážeme zajistit. Proto jsem přistoupil k modifikaci, která zachovává výhody rozšiřitelného hašování a za cenu zhoršení odhadů nepožaduje žádné speciální předpoklady. Optimalizována byla téměř vždy rychlost operací za cenu zvětšení databáze.

1. 1. Přehled funkcí

- Hledání ve zdrojových kódech podle identifikátorů
  - deklarací
    - globálních proměnných
    - struktur
    - funkcí
    - typů
  - definic
    - funkcí
    - struktur
    - maker
  - použití
    - proměnných (přiřazení, předání jako parametru, volání)
    - maker (expanze, direktiva preprocessoru)
    - funkcí (volání, přiřazení adresy, předání parametru)
    - struktur (deklarace proměnné, přetypování)
    - typů (deklarace proměnné, přetypování)
- Hledání viditelných identifikátorů podle polohy ve zdrojovém kódu
  - funkcí
  - proměnných
  - maker
  - typů
  - všech identifikátorů
- Hledání funkcí použitých v zadané funkci (volání, přiřazení, předání jako parametru)
- Hledání globálních proměnných použitých v zadané funkci (čtení, zápis, přiřazení adresy)
- Hledání deklarace proměnné podle místa jejího použití (podle polohy použití ve zdrojovém kódu)
- Hledání použití lokální proměnné podle místa její deklarace (podle polohy ve zdrojovém kódu)
- Hledání souborů, které includují soubor
- Vytváření databáze pro urychlení vyhledávání
  - databáze na disku
  - databáze v paměti
- Možnost updatu databáze
- Plně automatické updatování databáze při kompilaci pomocí make
- Hledání identifikátorů podle regulárního výrazu (i vzniklých po expanzi makra!)
- Textově orientované rozhraní
- Rozšíření pro Emacs

1. 2. Podobné projekty

Následuje seznam programů, které mají podobné schopnosti jako `cspot`. Vzhledem k potencionální neobjektivitě a nedostatečné detailní znalosti všech jejich funkcí jsem se rozhodl připojit přeložený popis, kterým jsou obecně charakterizovány, uvést své zkušenosti s užíváním programu a srovnat jeho funkci s `cspotem`.

1. 2. 1. Ctags, Etags

Programy `ctags` a `etags` [17] generují tag-file pro slova nalezená v souborech. Tento soubor umožňuje záznamy rychle najít. 'Tag' znamená slovo, pro které byl v indexu vytvořen záznam. Program `ctags` tedy generuje soubor s navzájem provázanými záznamy a vypisuje jej v čitelné formě. Tag-fily jsou podporovány mnoha editory, které pak umožňují nalézt záznam s příslušným identifikátorem a přeskočit na místo výskytu.

Mé zkušenosti s `ctags` jsou minimální, neboť nepodporují vyhledávání všech odkazů na konkrétní symbol, což je funkce, kterou očekávám od každého prohledávače zdrojových kódů. Navíc `cscope`, který je popsán v následujícím odstavci, tuto funkci podporuje, a proto jsem (před napsáním programu `cspot`) preferoval právě jeho.

1. 2. 2. Cscope

Program `cscope` [16] je vývojářský nástroj na procházení zdrojových kódů. Má čistě unixovský původ, původně byl vyvinut v Bellových laboratořích v dobách PDP-11. Byl dlouhou dobu částí oficiální distribuce AT&T Unixu a byl použit ke spravování projektů obsahujících 20 milionů řádek kódu!

V dubnu 2000 byl zdrojový kód pro `cscope` uvolněn pod BSD licencí.

Důležité rysy:

- Hledání ve zdrojových kódech
  - všech odkazů na symbol
  - globálních definicí
  - funkcí volaných zadanou funkcí
  - funkce volající zadanou funkci
  - hledání textových řetězců
  - hledání regulárních výrazů
  - hledání souborů includujících zadaný soubor
- Textově orientovaný
- Generována databáze na uchovávání informací pro rychlejší hledání a pozdější použití

- Obecný parser podporuje jazyk `C`, ale je dostatečně přizpůsobivý, aby byl použitelný na `C++` nebo `Javu`. Lze jej však použít i k procházení velkých textových dokumentů.
- Mód s příkazovou řádkou, aby byl snadno vnořitelný do skriptů či grafického rozhraní.
- Běží na všech verzích Unixu i některých dalších operačních systémech.

S programem `cscope` mám bohaté zkušenosti, neboť k němu existuje i velmi pěkné rozšíření pro Emacs. Typy funkcí `cscope` i jeho rozšíření pro Emacs jsem považoval za základ, který musí obsahovat i `cspot`. Ovládání `cscope` je jednoduché a jeho vyhledávání v jeho pomocné databázi je rychlé. Nicméně `cscope` používá velmi obecný parser, který sice umožní jeho použití i pro běžné textové soubory, ale nevýhoda tkví v tom, že pro konkrétní programovací jazyk (v našem případě jazyk `C`) neposkytuje všechny informace, které programátora zajímají. Hlavní nedostatek jsem viděl v neprovádění skutečného preprocessingu před parsováním a tento nedostatek jsem (mezi jinými) v programu `cspot` odstranil.

1. 2. 3. GNU GLOBAL

GNU GLOBAL [18] je systém ke značkování zdrojových kódů pracujících stejným způsobem v různých prostředích. Je možné rychle nalézt konkrétní objekt ve zdrojových kódech a rychle se k němu přesunout. Může být použit k vyrovnání se s rozsáhlým projektem obsahujícím množství podadresářů, mnoho `#ifdef` a mnoho main funkcí. Je podobný systému `ctags` a `etags`, ale liší se nezávislostí na textovém editoru. Běží na Unixu a kompatibilních operačních systémech, např. GNU, BSD.

GNU GLOBAL je součástí GNU projektu a je volně šiřitelný.

GNU GLOBAL má následující rysy:

- podporuje `C`, `C++`, `Yacc`, `Java` a `PHP4`.
- pracuje v různých prostředích - shell, vi, Emacs aj.
- rychle nalezne polohu specifického objektu
- nalezne nejen definice, ale i reference na objekt
- regulární výrazy
- podporuje vnější programy pro vyhledávání (grep a id-utils).
- značkovací soubory jsou nezávislé na architektuře systému
- podporuje inkrementální update značkovacích souborů

S GNU GLOBAL nemám žádné uživatelské zkušenosti. Podle popisu ale obsahuje velmi obecný parser, což znamená, že nezná sémantiku jazyka `C`.

1. 2. 4. Cedet

CEDET [15] je množina nástrojů napsaných pro zlepšení vývojového prostředí editoru Emacs. Emacs je už nyní výborným prostředím pro vývoj programů, ale v mnoha směrech jej lze ještě vylepšit. Existuje mnoho nových myšlenek pro zlepšení vývojových prostředí a v mnoha produktech již byly tyto myšlenky použity. Mezi ně beze sporu patří Microsoft's Visual environment, JBuilder, Eclipse, nebo KDevelop. CEDET je projekt spojující mnoho nástrojů ke zlepšení vývojového prostředí v Emacsu. Zatímco Emacs podporuje snadné editování v mnoha svých módech, jeho podpoře procházení souborů stále něco chybí. K tomu slouží právě CEDET: zobrazí množství informačních oken, která dovolí snadnou navigaci a náhled na zdrojové kódy.

Informační okna obsahují:

* Adresářový strom,
* seznam zdrojových souborů v aktuálním adresáři,
* seznam funkcí/tříd/metod/... v aktuálním souboru
* historii naposledy navštívených souborů

S kolekcí CEDET mám relativně krátké zkušenosti. CEDET je napsaný v elispu, a proto je pomalejší než ostatní podobné nástroje. Nabízí menu, které usnadňuje navigaci ve zdrojovém kódu, ale hledání ve zdrojových kódech podporuje myslím nedostatečně. Pomocí jeho menu můžeme přecházet mezi deklaracemi, definicemi, zobrazit includované soubory, atd. Nicméně jsem jeho menu ve skutečnosti nikdy pořádně nepoužil, neboť se mi zdálo jednodušší říci, co hledám a pak se jedním kliknutím přesunout na správné místo. Tyto funkci bezproblémově obstarává `cscope` nebo můj `cspot`.

1. 3. Sparse

Programu `cspot` potřebuje umět parsovat zdrojové kódy a 'rozumět' vztahům mezi jednotlivými symboly. Ukázalo se výhodné upravit a použít sparse k parsování zdrojových kódů. Ne všechny požadované informace jsou ale dostupné přímo ve sparse stromu. Proto bylo nutné mírně sparse modifikovat a ukládat i informace, které sparse záměrně zapomíná, aby šetřil operační pamětí.

Sparse se navíc již osvědčil na velkých projektech, jako je např. jádro Linuxu. Kvůli takovým projektům je též vyvíjen `cspot`. Sparse a zbytek projektu je relativně pevně oddělený, aby bylo snadné v případě potřeby použít jiný parser.

Sparse je sémantický parser zdrojových kódů. Lze jej použít jako front-end pro kompilátor. Ze zdrojového kódu vybuduje strom, v němž jsou uzly různých typů. Mezi nejdůležitější patří deklarace proměnných a definice funkcí. Ty dále ukazují na sémanticky správný uzel upřesňující deklaraci symbolu, typ symbolu, aj. Jde

o jednoduchou knihovnu, která má za úkol vytvořit sémantický strom pro další analýzu.

Parsování se skládá z pěti fází:

- tokenizování celého souboru
  (token = slovo nebo atomická část, tokenizování = operace rozkládání textu na tokeny)
- pre-processing (který může způsobit tokenizaci dalších souborů)
- sémantické parsování
- vyhodnocení typů
- expanze inline funkcí a zjednodušení stromu sparse

Budování databáze s pomocí sparse probíhá v následujících fázích:

α) Nastavení funkcí, které má sparse volat. Funkce mají za úkol ukládat informace, které sparse zapomíná.

β) Spuštění sparse. Při jeho běhu se ukládají potřebné záznamy do paměti.

2. Používání programu

2. 1. Instalace

Před používáním programu `cspot` je potřeba nejdříve zkompilovat zdrojové kódy dodané v archívu `cspot.tgz`. Jeho součástí je i verze sparse, s níž byl `cspot` vyvíjen. Pokud by jste chtěli používat sparse v aktuální verzi, pak stačí stáhnout nejnovější verzi na [23] a aplikovat patch soubor, který je též dodán v archívu `cspot.tgz`.

Dodaný archív lze rozbalit příkazem `tar -xzf cspot.tgz`. Provedením příkazu `make install` jako root se program nainstaluje do adresáře `/usr/local/bin`. Pokud není potřeba instalovat `cspot` tak, aby byl dostupný pro všechny uživatele, spusťte `make` bez parametrů a poté zkopírujte `cspot` a `cspot_cc` do vámi zvoleného adresáře a v proměnné `PATH` k němu nastavte cestu. Pokud chcete používat `cspot` v prostředí Emacs, připojte obsah souboru `.emacs` v instalačním adresáři k souboru `.emacs` ve svém domovském adresáři.

2. 2. Zapojení cspotu do Makefile

Program `make` je velmi často užívaný pro kompilaci zdrojových kódů. Je výhodný, neboť umí poznat, které zdrojové kódy se od poslední kompilace změnily, a překládá pouze je. Toho můžeme využít pro snadný update naší databáze a udržení její konzistence se zdrojovými kódy bez nutnosti rebuildu celé databáze. Stačí jen změnit program, který provádí kompilaci, na `cspot_cc`. Tím se automaticky vytvoří a poté aktualizuje seznam souborů a akcí, které je potřeba provést (build, aktualizace, ...). Program `cspot_cc` pak navíc pro každý zdrojový kód správně nastaví, která makra jsou při jeho kompilaci definovaná a kde hledat hlavičkové soubory.

2. 3. Plugin pro Emacs

Součástí balíčku je i soubor `.emacs`. Tento soubor se používá pro uživatelské nastavení Emacsu a provádí se při jeho spuštění. Obsah souboru `.emacs` by měl být připojen k obsahu souboru `.emacs` uživatele. Při spuštění textového editoru Emacs pak bude uložen aktuální adresář (lze totiž předpokládat, že je to právě kořenový adresář projektu). Dále bude nastavena cesta ke spustitelnému souboru `cspot`, aby jej mohl používat přímo Emacs. Vyhledávací funkce mohou být libovolně mapovány na klávesové zkratky. Volbu vhodných klávesových zkratek nechávám plně na uživateli. Některé klávesové zkratky jsou již přednastaveny, spíše ale jako příklad pro uživatele neznalého Emacsu, aby mohl snadno vytvořit vlastní klávesové zkratky.

Plugin umožňuje volat většinu funkcí `cspotu` přímo z prostředí Emacs. Do dotazu automaticky vyplňuje identifikátor pod kurzorem. Výsledek se zobrazí ve vlastním

bufferu. V zobrazeném bufferu fungují speciální příkazové zkratky a je možné například přejít na požadovaný odkaz z výsledku, posouvat se mezi jednotlivými výsledky či zobrazit výsledek v stavové řádce, což je zvlášť vhodné pro zobrazení deklarace funkce či proměnné. Plugin též umožňuje vybudovat databázi, pokud neexistuje, avšak tuto databázi není možné spravovat pomocí `cspot_cc` a `make`. Databáze navíc obsahuje informace o všech zdrojových kódech z adresáře projektu a nejen o souborech projektu.

2. 4. Seznam příkazů cspotu

Příkazy je možné zadávat buď na příkazovou řádku, která se zobrazí po spuštění programu `cspot`, nebo jako parametr při spuštění. Příkazová řádka podporuje doplňování příkazů i identifikátorů (pokud je otevřená databáze) a procházení již zadaných příkazů.

fl soubor: **fil**e**l**ist
Příkaz specifikující seznam souborů, z nichž má být sestavena databáze. Formát souboru je popsán podrobněji v další části.

def soubor: **def**ine
Příkaz specifikující soubor s makry definovanými před kompilací, částečná analogie `-D` pro `gcc`.

ip soubor: **i**nclude **p**ath
Příkaz pro specifikace cest k hlavičkovým souborům. Částečná analogie `-I` u `gcc`.

db-ro, db-rw, db soubor: **d**ata**b**ase mode **r**ead-**o**nly, **r**ead-**w**rite
Jde o příkazy k otevření databáze. Příkazy `db-ro` a `db-rw` otevírají databázi pro čtení nebo zápis. Příkaz `db` je pouze pro příkazovou řádku a otevře databázi v módu pro čtení či zápis v závislosti na potřebě ostatních příkazů.

bf, uf, rf soubor: **b**uild **f**ile, **u**pdate **f**ile, **r**emove **f**ile
Jde o příkazy pro přidání souboru do databáze, update pro daný soubor v databázi a odebrání záznamů vztahujících se k souboru z databáze. Je nutné ale poznamenat, že databáze se po odebírání souboru nikdy nezmenší a že pro build více souborů je volání `bf` pro každý soubor nevhodné, neboť pak není možné filtrovat duplicitní záznamy a nepokoušet se je vkládat do databáze vícekrát.

ba, ua, bm, sm bez parametrů: **b**uild **a**ll, **u**pdate **a**ll, **b**uild **m**em, **s**ave **m**em
Jde o příkazy pro vybudování databáze pro více souborů. Tyto příkazy jsou dostatečně optimalizované a jsou tedy doporučené pro vybudování databáze. Příkaz `ba` vyčistí databázi a znovu ji vybuduje, databáze se potom v jednom průchodu uloží do souboru na disku. Proto příkaz `ba` požaduje, aby před jeho zavoláním byl zavolán příkaz `db-rw` a příkaz `fl`. Příkaz `ua` též buduje databázi z více souborů, ale předtím ji nečistí, a tedy pro každý ukládaný záznam musí nejdříve ověřit, zda se již v databázi nevyskytuje. Příkaz `bm` provede vybudování databáze

stejně jako `ba`, ale tentokrát ponechává databázi pouze v paměti. Tedy nepožaduje předchozí spuštění `db-rw`. Vybudovaná databáze může být před ukončením práce uložena. Nejprve nastavením databáze příkazem `db-rw`, poté ukončením příkazem `bye` nebo spuštěním příkazu `sm`. Všechny tyto příkazy pro build databáze budují nejprve databázi v paměti a poté ji teprve ukládají na disk. Jde o optimalizaci, aby ukládání netrvalo příliš dlouho. Nicméně pro veliké projekty může vzniknout nedostatek paměti (k čemuž ale nedojde ani při budování databáze pro jádro Linuxu). V tom případě je nejjednodušší rozdělit seznam souborů na více menších seznamů a postupně používat příkaz `ua`.

p bez parametrů: **p**rocess
Příkaz, který jednou projde seznam souborů a u každého provede požadovanou akci (seznam akcí naleznete v sekci příkazy seznamu souborů). Je tedy univerzálnější než výše zmíněné příkazy `ba` a `ua`. Pro první build velké databáze je ale vhodné používat `ba`, neboť u něj není potřeba některé kontroly provádět a běh je tedy rychlejší. Nicméně příkaz `p` je vhodný při automatickém updatování databáze pomocí make.

ls bez parametrů: **l**i**s**t
Příkaz `ls` vypíše všechny identifikátory a názvy souborů.

regexp regulární výraz: **reg**ular **exp**ression
Příkaz `regexp` vypisuje pouze ty identifikátory a názvy souborů, které odpovídají zadanému regulárnímu výrazu (je použité rozšířené matchování regulárního výrazu [21]). Jeho volání je možné jak pro databázi v paměti, tak databázi na disku, a tedy je nutné specifikovat existující databázi pomocí `db-ro` nebo jí vybudovat v paměti pomocí `bm`.

so soubor: **s**parse **o**utput
Příkaz `so` požaduje jako argument jméno souboru, do něhož bude vypisovat chyby, které sparse při buildu souboru nalezne. Původní obsah souboru bude přepsán.

Příkazy pro dotazování využívající databázi

Všechny tyto příkazy požadují, aby před jejich zavoláním byla otevřena databáze (viz příkazy `db-ro`, `db-rw`, `db`), nebo aby byla vybudována databáze v paměti příkazem `bm`. Tyto příkazy ignorují parametry `def` a `ip`, neboť parsování souborů již neprobíhá. Jsou pouze vyhledány odpovídající záznamy v databázi.

ff, fc, fu identifikátor: **f**unction de**f**inition, de**c**laration, **u**sage
Příkazy vypisující polohu definice funkce s daným identifikátorem (`ff`), polohu všech deklarací funkce s daným identifikátorem (`fc`) a polohu všech použití funkce s daným identifikátorem (`fu`). Mezi použití se počítá jak zavolání funkce, tak přiřazení či předání její adresy. Mezi použití funkce se ale nepočítá, jde-li o proměnnou typu ukazatel na funkci, která je volána či předána jako parametr. Pokud jsou požadované tyto výskyty, použijte hledání použití proměnné (`vu`).

mf, mu identifikátor: **m**acro de**f**inition, **u**sage

Příkazy vypisující polohu definice makra (`mf`) a použití makra (`mu`). Za použití makra daného identifikátoru `A` je považována i situace, kdy makro `A` vznikne až po expanzi jiného makra.

vc, vu identifikátor: **v**ariable de**c**laration, **u**sage

Příkazy vypisující polohu deklarace globální proměnné (`vc`) a to jak deklarace typu extern, tak static, nebo vypisující použití globální proměnné s daným identifikátorem (`vu`) včetně určení, kde byla daná proměnná deklarovaná. Za deklaraci proměnné je též považován i výskyt příslušného identifikátoru při deklaraci výčtového typu `enum` a použití položky výčtového typu za použití proměnné.

sf, sc, si, su identifikátor: **s**truct de**f**inition, de**c**laration, **u**sage, **i**nfo

Příkazy vypisující polohu definice struktury (`sf`), polohu deklarace struktury (`sc`), polohy všech použití struktury s tímto identifikátorem (`su`), a polohu definic struktury včetně identifikátorů proměnných (`si`), které struktura obsahuje. Pro výpis identifikátorů v příkazu (`si`) je spuštěn sparse, tedy je možné, že bude nutné nastavit (`def`) a (`ip`). Mezi struktury se řadí jak struct, tak union.

td, tu identifikátor: **t**type**d**ef, **t**ypedef **u**sage

Příkazy vypisující polohu deklarace typedefu `td` a použití typedefu `tu`. Mezi použití typedefu patří deklarace a přetypování. Mezi deklarace se řadí i lokální deklarace.

sym identifikátor: **sym**bol

Příkaz vypisující polohy všech použití tohoto identifikátoru, tedy všech deklarací, definicí a použití všech funkcí, proměnných, maker, struktur a typedefů s daným identifikátorem.

in soubor: **in**cluded

Příkaz vypisující všechny soubory, které includují zadaný soubor. Cesta k includovanému souboru, je-li součástí projektu, by měla být zadána jako relativní cesta z adresáře projektu, neboť tak je uložena v databázi. Neexistuje-li žádný takový soubor, pak je jméno zadaného souboru považováno za suffix a jsou nalezeny všechny soubory includující nějaký soubor, jehož cesta končí daným sufixem. Tento přístup je vhodný pro vyhledávání souborů, které includují požadovaný systémový include, nebo pokud nechceme zadávat celou cestu z adresáře projektu.

uv, uff identifikátor: **u**nused **v**ariables, **f**unctions de**f**initions

Příkazy vypisující nepoužité globální proměnné `uv` a nepoužité definice funkcí `uff`. Tyto příkazy mohou být relativně pomalé, neboť se pro každou deklaraci/definici hledá alespoň jedno použití, a tedy je nutné procházet databázi vícekrát.

Příkazy pro dotazování nevyužívající databázi

Všechny následující příkazy očekávají jako parametr cestu k souboru, číslo řádku a číslo znaku na řádku. Při každém volání příkazu se provádí parsování celého souboru. Výpisy těchto příkazů jsou závislé na nastavení předdefinovaných maker a cestám k includovaným souborům. Často je při jejich volání vhodné použít parametry `def` a `ip`.

lvd, lvu soubor, číslo řádku, pozice na řádku: **l**ocal **v**ariable **d**eclaration, **u**sage
Příkazy pro nalezení deklarace proměnné k jejímu použití `lvd` a všechna použití k dané deklaraci lokální proměnné `lvu`.

va, vla soubor, číslo řádku, pozice na řádku: **v**isible (**l**ocal) **a**t
Příkazy vypisující všechny identifikátory viditelné na daném místě v kódu, a to buď všechny `va`, nebo pouze deklarované/definované ve sledovaném souboru `vla`.

vfa, vlfa soubor, číslo řádku, pozice na řádku: **v**isible (**l**ocal) **f**unctions **a**t
Příkazy vypisující všechny viditelné identifikátory funkcí z daného místa, a to buď všechny `vfa` nebo pouze deklarované/definované ve sledovaném souboru `vlfa`.

vma, vlma soubor, číslo řádku, pozice na řádku: **v**isible (**l**ocal) **m**acro **a**t
Příkazy vypisující všechny viditelné identifikátory maker z daného místa a to buď všechny `vma`, nebo pouze definované ve sledovaném souboru `vlma`.

vva, vlva soubor, číslo řádku, pozice na řádku: **v**isible (**l**ocal) **v**ariables **a**t
Příkazy vypisující všechny viditelné identifikátory proměnných z daného místa, a to buď všechny `vva`, nebo pouze deklarované ve sledovaném souboru `vlva`.

cfa, ufa soubor, číslo řádku, pozice na řádku: **c**alled/**u**sed in **f**unction **a**t
Příkaz vypisující všechny funkce `cfa` nebo proměnné `ufa`, které jsou použity funkcí na zadané pozici.

Příkazy seznamu souborů

Následující příkazy jsou určené pro použití v seznamu souborů, kde se vyskytují jako první znak na řádce. Každá řádka seznamu souborů obsahuje záznam pro jeden soubor. Jednotlivé parametry jsou v záznamu odděleny znakem ':'. Pro automatickou správu seznamu souborů je možné používat program `cspot_cc`. Ten umí rozpoznat, jakou akci je nutné se souborem provést a je-li jeho akce nastavena na '`%`', pak ji změní buď na '`u`' nebo '`+`'. Akce bude vykonána při dalším zpracovávání seznamu. Seznam souborů může však být i pouhý seznam souborů bez jakýchkoli příkazů. Pak ale nemůže být spravován programem `cspot_cc` a příkazem `p`.

`%` :
Žádná akce. Soubor nebude při průchodu seznamem zpracováván.

`+` :
Soubor bude při průchodu seznamem přidán do databáze. Soubor bude zahrnut i při spuštění `ba`. Po zpracování souboru dojde ke změně nastavení na `%`, aby již nebyl zahrnut při dalším průchodu seznamem.

`u` :
Soubor bude zahrnut při průchodu seznamem. Nejdříve bude z databáze vymazán a poté bude znovu přidán do databáze. Soubor bude zahrnut i při spuštění `ua`. Po zpracování souboru dojde ke změně nastavení na `%`, aby již nebyl zahrnut při dalším průchodu seznamem.

`U` :
Soubor bude zahrnut při průchodu seznamem. Nejdříve bude z databáze vymazán a poté bude znovu přidán do databáze. Při vymazávání souboru z databáze budou z databáze smazány i všechny soubory, které soubor includuje. Soubor bude zahrnut i při spuštění `ua`. Po zpracování souboru dojde ke změně nastavení na `%`, aby již nebyl zahrnut při dalším průchodu seznamem.

`*` :
Soubor bude zahrnut při průchodu seznamem. Nejdříve bude z databáze vymazán a poté bude znovu přidán do databáze. Soubor bude zahrnut i při spuštění `ua`.

`#` :
Soubor nebude zahrnut při budování databáze. Pokud je seznam souborů spravován pomocí programu `cspot_cc`, pak `cspot_cc` nikdy nemění tuto akci.

`-` :
Soubor bude při zpracovávání seznamu souborů vymazán z databáze. Po zpracování souboru dojde ke změně nastavení na `%`, aby již nebyl mazán při dalším průchodu seznamem.

`_` :
Soubor bude při zpracovávání seznamu souborů vymazán z databáze. Navíc odtud budou smazány i všechny soubory, které soubor includoval. Po zpracování souboru dojde ke změně nastavení na `%`, aby již nebyl mazán při dalším průchodu seznamem.

3. Struktura a správa databáze

3. 1. Struktura databáze

Databáze je vytvořena podle požadavku uživatele v souboru nebo pouze v operační paměti. V operační paměti je databáze lesem červenočerných stromů a každý strom reprezentuje jeden typ záznamů. V souboru se místo červenočerných stromů využívá rozšiřitelné hašování. V následujících odstavcích popíšeme formát souboru a způsob implementace rozšiřitelného hašování.

3. 1. 1. Stránkování databáze

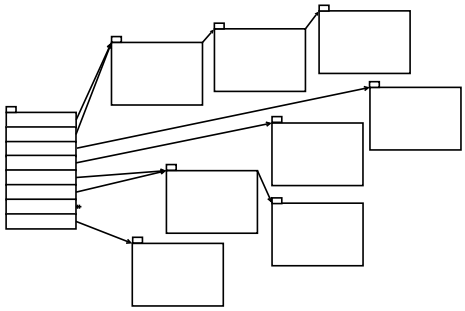
Vzhledem k tomu, že je databáze uložena na disku, což je blokově orientované zařízení a data jsou z něj čtena vždy po násobcích velikosti bloku, je vhodné, aby databáze byla též rozdělena na části, které budou mít stejný typ obsahu (a tedy nebude celý blok načten zbytečně). Tyto části nazýváme stránky. Velikost stránky by měla být násobkem velikosti bloku, aby byl využit celý načtený blok. Při operacích vkládání a mazání dochází k alokování stránek, resp. k jejich uvolňování. Nepoužívané stránky jsou řazeny do lineárního spojového seznamu, protože ne vždy je stránka na konci souboru, a tedy jej nemůžeme zmenšit.

3. 1. 2. Hlavička databáze

Obsahuje string identifikující, že se jedná o databázi cspot, velikost stránky databáze, adresu první volné stránky databáze, adresu stránky pro stringy s volným místem (a pro urychlení též adresu prvního volného místa pro další vkládaný string), první stránku se stringy (stránky tvoří spojový seznam) a adresy jednotlivých adresářů. Zdrojový kód týkající se hlavičky databáze je umístěn v souboru `database/file.h`. Důležité je definované makro `TABLES`, které je použité pro automatické generování všech tabulek databáze, pro inicializaci a načtení databáze a pro generování funkcí pro ukládání do databáze.

3. 1. 3. Tabulky databáze

Jsou implementovány pomocí rozšiřitelného hašování, které je definováno v kapitole 4.2, avšak s některými úpravami plynoucími z nesplnění předpokladů o jednoznačnosti klíče a rovnoměrné distribuce hašovací funkce. V rozšiřitelném hašování jsou v adresáři uloženy adresy stránek, kde se položka s hašovaným klíčem může nacházet. V naší modifikaci jsou v adresáři uloženy odkazy na spojové seznamy stránek a hledaná položka se může nacházet v libovolné stránce tohoto spojového seznamu. Tím je vyřešena jak nejednoznačnost klíče, tak nemožnost předpokládat rovnoměrnou distribuci hašovací funkce. Nicméně některé spojové seznamy byly vzhledem k existenci nejednoznačných klíčů příliš dlouhé. Proto byly pro klíče, kterým přísluší mnoho záznamů, vyhrazené vlastní stránky. Na následujícím obrázku je jedna z možných konfigurací, jak může struktura tabulky vypadat.



3. 1. 4. Stránky hašovací tabulky

Každá stránka má hlavičku (detaily viz. příslušné makro ve `file.h`), která obsahuje následující položky. Každá stránka má hlavičku, která obsahuje následující údaje. První byte udává, kolik bitů z hašovací funkce se ve stránce používá. Druhý byte je nepoužitý. Zbylé dva byty jsou využité jako šestnáctibitový integer, v němž je uložen počet záznamů ve stránce. Následuje čtyřbytová adresa další stránky v lineárním spojovém seznamu.

3. 2. Položky databáze

Každá položka databáze obsahuje nějaký textový identifikátor. Textové identifikátory jsou často použité i jako vyhledávací klíč. V databázi je můžeme mít uložené buď přímo v jednotlivých záznamech, nebo samostatně a v záznamech se na ně odkazovat jejich adresou.

Budou-li identifikátory (stringy) uloženy přímo v jednotlivých záznamech, budou se v databázi vyskytovat v mnoha kopiích, což je velmi neefektivní. Zajistíme tedy, aby každý string byl v databázi uložen jen jednou.

Tím zajistíme jednoznačné přiřazení stringu a jeho adresy v databázi. Vyhledávání pak prováníme podle adresy stringu a ne podle obsahu stringu samotného.

Kdybychom nechtěli za klíč použít adresu, pak musíme použít jako klíč samotný string. Tzn. porovnávat hledaný string se stringem na adrese, která je v záznamu uvedena. Tedy pro každý záznam musíme místo rychlého porovnání dvou celých čísel provést porovnávání stringů, které je pomalejší. Navíc je potřeba string načíst do paměti, což může vyvolat výpadek stránky, a tedy výrazné zpomalení.

3. 2. 1. Stringy

Již víme, že je vhodné mít stringy uložené mimo záznamy a odkazovat se na ně adresou. Ta ale musí být neměnná, neboť při její změně bychom museli upravovat všechny záznamy, které ji obsahují. Pro stringy tedy vyhradíme speciální typ stránek, a nikdy je odtud nemažeme. K hledání adresy uloženého stringu využijeme hašovací tabulku. Musíme si uvědomit, že není možné ukládat stringy např. přímo do stránek hašovací tabulky, neboť zde by se adresa měnila při přehašovávání.

Ve stránce hašovací tabulky jsou uvedeny adresy, na nichž jsou stringy uloženy. Porovnáním hledaného stringu a stringu na jednotlivých adresách nalezneme adresu hledaného stringu. Abychom minimalizovali počet výpadků stránek, pamatujeme si k uložené adrese stringu i jeho haš. Pak totiž kontrolujeme jen stringy se stejnou haší, a ne všechny ve stránce (ty mají společný pouze nějaký prefix). Vše co se týká stringů je implementováno v `database/template/string.h`.

Dotaz na databázi, který používá jako klíč string, je pak implementován jako dotaz na polohu příslušného stringu a dotaz, který používá jako klíč adresu daného stringu. Vypsání všech stringů v databázi je vhodné kvůli dotazu na shodu s regulárním výrazem. Stránky obsahující stringy jsou proto řazeny do spojového seznamu.

3. 2. 2. Záznamy

Díky předchozímu způsobu uložení stringů můžeme mít všechny ostatní záznamy s pevnou velikostí, čímž je umožněno jejich adresování. Databáze podporuje tři typy záznamů, nicméně v případě potřeby dalšího typu je jeho vytvoření velmi jednoduché.

Záznam: klíč → hodnota. Je využitý pro uchovávání informace, které soubory jsou includovány souborem a které identifikátory jsou využity v daném souboru. Obě tyto informace jsou důležité pro update databáze. Tento typ záznamů je implementován v souboru `database/template/struct2.h`.

Záznam: klíč → (h1, h2, h3). Je využitý pro uchování informace, že klíč = identifikátor je použitý v souboru `h1` na řádce `h2` a pozici na řádce `h3`. Tento typ záznamů je využit pro uchování pozic definice makra, použití makra, definice funkce, deklarace funkce, použití funkce, deklarace globální proměnné, forward deklarace struktury, definice struktury, použití struktury, typedef deklarace a použití typedef deklarovaného identifikátoru. Tento typ záznamů je implementován v souboru `database/template/struct4.h`.

Záznam: klíč → (h1, h2, h3, h4, h5, h6). Je využitý pro uchování informace, že klíč = identifikátor je použitý v souboru `h1` na řádce `h2` a pozici na řádce `h3` a vztahuje se k souboru `h4`, řádce `h5` a pozici `h6`. Tento typ záznamu je použit pro uložení použití globální proměnné. Tento typ záznamů je implementován v souboru `database/template/struct7.h`.

3. 3. Implementace operací nad databází

3. 3. 1. Průběh budování databáze

Postupně je procházen seznam souborů, které mají být zahrnuty v databázi. Pro každý z nich se nejprve nastaví cesty k include adresářům a makra, která mají být definována pro daný soubor. Následuje volání sparse. Sparse je upravený tak,

aby při preprocessingu a parsování zdrojového kódu předával `cspotu` nalezené symboly. Ten je ukládá do databáze v paměti. Po dokončení následuje reset sparse a pokračování dalším souborem.

Na databázi v paměti jsou kladeny vysoké časové i paměťové nároky. Je tedy vhodné, aby v ní neexistovaly duplicitní záznamy. Došlo tedy k oddělení stringů od zbylých informací o vkládaném symbolu, neboť se ve zdrojovém textu často identifikátory (a tedy i stringy) vyskytují vícekrát. Jak pro stringy, tak pro každý typ záznamu je vytvořen červenočerný strom, čímž je zajištěno neduplicitní uložení symbolů.

Po průchodu celým seznamem souborů jsou do databáze nejprve uloženy všechny stringy, čímž jim je přidělena jejich adresa v databázi, kterou se na ně ostatní symboly budou odkazovat.

Následuje ukládání ostatních symbolů do databáze, které může probíhat v libovolném pořadí.

3. 3. 2. Ukládání do databáze

Databáze v paměti

V této sekci bych chtěl popsat, jakým způsobem se dostanou data od sparse až do červenočerných stromů.

Pokud při preprocessingu nebo parsování nalezneme symbol, podle kontextu se zavolá příslušná funkce a předá veškeré nutné dostupné informace o daném symbolu. Sparse volá funkce deklarované v `interface/interface.c` jako proměnné typu ukazatel na funkci. Hlavním smyslem této nepřímé definice volaných funkcí je oddělení programu `cspot` a sparse. Tyto proměnné jsou inicializované ve funkci `init_fce`. Nastavované funkce ověřují korektnost parametrů, ověří zda má být záznam zpracován a případně jej předá funkci, která se o toto zpracování postará. Skutečné uložení záznamu do paměti provede právě tato funkce. Ta je nastavena funkcí `sparse_build_functions`.

Databáze v souboru

Všechny informace, které mají být zapsány do databáze, jsou nejdříve zapsány do červenočerných stromů v paměti, a poté jediným průchodem zapsány do hašovací tabulky. Výhodou je, že ověření, zda jde o duplicitní záznam, stačí provádět v paměti, a není tedy nutné procházet celou stránku hašovací tabulky, čili stačí pouze připsat záznam na konec stránky.

3. 3. 3. Implementace dotazu

Dotaz bez databáze

Pokud dotaz určuje pozici v souboru, pak se sparse spustí na příslušný soubor. Sparse nastavíme tak, aby vypisoval všechny relevantní odpovědi přímo při parso-

vání a při následném průchodu stromem sparse. Většinou jde o dotazy typu: jaké identifikátory jsou viditelné z tohoto místa, nebo dotazy, které potřebují jednoznačně identifikovat proměnnou či funkci (k jednoznačné identifikaci nestačí pouze identifikátor).

Průběh dotazu: Pro příslušný zdrojový kód proběhne preprocessing a parsování a všechny identifikátory definované a deklarované před daným místem jsou přímo vypisovány. Poté se ověří, zda není v daném místě definována funkce, a pokud ano, pak jsou vypisovány pouze identifikátory, které jsou opravdu viditelné. Nebere se však v úvahu zastiňování identifikátorů, a to z toho důvodu, aby byla případná chyba ve zdrojovém kódu rychle rozpoznatelná.

Dotaz na databázi

Nejprve je nalezena adresa identifikátoru. Není-li identifikátor v databázi nalezen, dotaz končí, neboť do databáze nemohl být žádný záznam s hledaným identifikátorem vložen. Jinak by tam totiž byl i samotný identifikátor. Je-li nalezena adresa identifikátoru, pak probíhá hledání a jako klíč je použita adresa identifikátoru.

Není-li v dotazu `in` nalezen žádný záznam, pak je zadaný identifikátor interpretovaný jako sufix hledaného identifikátoru a dotaz je zopakován.

Existuje však i dotaz, který požaduje jak databázi, tak přístup k souboru. Tím je zobrazení obsahu struktury s daným identifikátorem. Nejprve jsou v databázi nalezeny a místa všech definic, a pak jsou postupně procházeny příslušné soubory a vypisovány obsahy struktur.

3. 3. 4. Update databáze

Databáze byla od počátku koncipována tak, aby umožňovala odebrat či vložit záznam, případně množinu záznamů. Dojde-li totiž ke změně několika souborů ve velkém projektu, je nevhodné provádět rebuild celé databáze. Proto byly implementovány mechanizmy pro zjištění, kterými záznamy změněný soubor do databáze přispěl. Stejně tak nesmíme zapomínat na fakt, že změněný soubor může ještě includovat další hlavičkové soubory. Je nutné si též uvědomit, že nová verze souboru nám nemůže poskytnout informace, které záznamy v databázi jsou již neplatné, a proto si musíme sami pamatovat, jakými záznamy soubor přispěl, abychom měli po updatu databázi v konzistentním stavu.

Potřebujeme-li provést úplný update, musíme vědět, kterými identifikátory soubor do databáze přispěl, které soubory includuje a které soubory jej includovaly. Tyto informace jsou v databázi sice uloženy, ale jejich získání by vyžadovalo procházet všechny stránky databáze. Proto jsem vytvořil tabulky, v nichž lze tyto informace vyhledat.

Úplný update se skládá z těchto částí:

1. Vyhledání starých záznamů a jejich smazání.
2. Vyhledání souborů, které soubor includuje, a jejich smazání z databáze rekurzivním provedením kroků 1 až 3.
3. Vyhledání souborů, které includují mazaný soubor, a jejich zařazení do seznamu souborů k přidání do databáze.
4. Zařazení souboru do seznamu k přidání do databáze.
5. Zpracování seznamu souborů k přidání do databáze.

Díky prvnímu kroku smažeme všechny staré záznamy. Díky druhému kroku smažeme všechny staré záznamy, které byly vloženy do databáze ze souborů, které mazaný soubor includoval. Díky kroku tři znovu přidáme includované záznamy, které byly společné pro mazané i nemazané soubory. Při vkládání do databáze ale kvůli kroku tři musíme kontrolovat, zda záznam neleží v nesmazaném souboru z kroku tři (takové soubory mohou existovat a jejich záznamy by se v databázi vyskytovaly vícekrát).

Výše popsaný postup by umožnil úplný update databáze, ale pro středně velké projekty by již znamenal prakticky rebuild celé databáze, neboť se často vyskytují hlavičkové soubory includované většinou souborů. Proto byl implementován rebuild, který neodebírá hlavičkové soubory. Tím docílíme výrazného zrychlení rebuildu databáze, ovšem za cenu toho, že databáze může obsahovat některé neplatné záznamy.

3. 4. Optimalizace

Při vývoji cspotu bylo potřeba množství optimalizací k urychlení vyhledávání v databázi i jejího vybudování. Proto jsem se rozhodl použít datové struktury, které zaručují, že vyhledávání nepotrvá příliš dlouho. V databázi bylo použito rozšiřitelné hašování, které zajistí průměrně konstantní počet operací při vyhledání prvku. Technika hašování však vyžaduje splnění některých požadavků, které nejsme schopni v programu zaručit. Proto jsem přistoupil k modifikaci rozšiřitelného hašování, která nebude mít na data žádné speciální požadavky a nezvýší příliš počet operací nutných k vyhledání prvku.

Pro rychlé budování databáze bylo nutné přistoupit k dalším optimalizacím. Data, která mají být vložena do databáze, jsou nejprve shromážděna v paměti a později zapsána do databáze. Tím zajistíme, že při ukládání do databáze není potřeba kontrolovat, zda je záznam již jednou v databázi uložený, čímž dosáhneme dalšího výrazného zrychlení.

Analýzou přístupů k databázi a zvolením nejvhodnější techniky jsem se snažil snížit režii programu a operačního systému a tím i časové nároky na hledání a vkládání prvků. Přístupy do databáze jsou náhodné, což znamená, že ze znalosti pozice, která byla načtena naposledy, nelze určit, kde proběhne další čtení.

Nejrychlejší způsob přístupu k datům je v našem případě paměťové mapování souboru.

Ukládáním všeho nejdříve do paměti a pak naráz do databáze. Tím nevkládám stejné záznamy do databáze vícekrát a při vkládání do stránek můžu pouze připisovat (jak u stringů, tak u položek).

Při ukládání záznamů si ve stránce pamatuji jejich počet. Z něho pak určím adresu posledního záznamu (záznamy mají pevnou velikost) a při zapisování do stránky stačí pouze zapsat záznam za tuto adresu. Mazaní ze stránky taktéž využívá znalosti umístění posledního záznamu. Mazaný záznam je nahrazen posledním, který je pak smazán.

Při ukládání stringů si musím pamatovat přímo adresu volného místa pro další string, neboť stringy nemají pevnou velikost, a tedy není možnost z jejich počtu určit adresu volného místa. Tuto adresu si ale stačí pamatovat pouze v hlavičce databáze, neboť stringy nemažu a zaplněná stránka se tedy nikdy neuvolní.

4. Implementace databáze

4.1. Paměťově mapované soubory

Rychlost aplikace je úměrná počtu instrukcí, které je potřeba vykonat, aby aplikace získala přístup k datům, která potřebuje. Operační systém často poskytuje funkce read a write, které tento přístup zajišťují. Nicméně tyto funkce jsou vhodné spíše pro sekvenční přístup než pro náhodný. Pro náhodný přístup je nutné se nejprve přesunout pomocí funkce seek na potřebnou adresu a poté použít funkce read nebo write. Existuje však lepší způsob, který je často efektivnější. Pokud soubor, který čteme nebo zapisujeme, namapujeme do adresového prostoru procesu, pak není nutné volat funkce read a write, ale je možné k obsahu souboru přistupovat přímo, jako kdyby byl uložen v paměti. Skutečný zápis a čtení souboru uskutečňuje operační systém, přesněji jeho správce virtuální paměti. Dojde-li ke čtení části souboru, pro kterou správce operační paměti nemá vytvořené mapování mezi virtuální pamětí a fyzickou pamětí, dojde k výpadku stránky a operační systém se pokusí zajistit mapování mezi virtuální a fyzickou pamětí. Je-li potřeba stránku z fyzické paměti uvolnit, operační systém provede zápis na disk, ale pouze pokud byl obsah stránky změněn, jinak stačí pouze mapování zrušit. Tento způsob je velmi efektivní, neboť není potřeba žádné kopírování dat.

U paměťově mapovaných souborů je ale problém se zvětšováním souborů. Po zvětšení souboru je totiž nutné buď namapovat část, o kterou se soubor zvětšil, do jiné oblast, nebo zrušit mapování, a soubor znovu do paměti namapovat.

4. 2. Hašování

Hašování (více v [5], [11] a [20] – ze které jsem při psaní této kapitoly vycházel) umožňuje průměrně konstantně rychlý přístup k položkám pomocí klíče (tedy stejně rychlý bez ohledu na počet uložených prvků). V této části popíšeme rozšiřitelné hašování a jeho modifikaci, která je v programu implementovaná.

Rozšiřitelné hašování

Externí paměť je rozdělena na stránky a každá stránka obsahuje b položek. Vždy v jednom kroku načteme celou stránku nebo zapíšeme celou stránku do externí paměti. Operace čtení a zápisu do externí paměti jsou mnohem pomalejší než čtení a zápis v interní paměti. Cílem je nalezení uložení dat do stránek externí paměti, přitom ale minimalizovat čtení a zápis do externí paměti.

Nechť je $h(x)$ prostá funkce (nazýváme jí hašovací funkcí), která rovnoměrně rozděluje $x \in U$ a nechť počet bitů v binárním zápisu funkce $h(x)$ je k.

Definice: Nechť $S \subseteq U$. Pro slovo α nad abecedou $\{0,1\}$ definujeme $h_S^{-1}(\alpha) = \{s \in S \mid \alpha \text{ je prefix } h(s)\}$. Řekneme, že α je kritické slovo, jestliže $0 < |h_S^{-1}(\alpha)| \leq b$ a pro každý vlastní prefix β slova α platí $b < |h_S^{-1}(\beta)|$. Pro $s \in S$ označíme $d(s)$ délku kritického slova, které je prefixem $h(s)$ a $d(S) = \max\{d(s), s \in S\}$

Množinu S reprezentujeme tak, aby existovala bijekce f mezi kritickými slovy a stránkami externí paměti sloužícími k representaci S, tedy $f : S \leftrightarrow R$, kde R je množina stránek externí paměti.

Mějme nyní funkci $a : S \to T$ zadanou tabulkou, kde T je množina stránek externí paměti, která každému slovu γ o délce $d(S)$ přiřadí adresu stránky předpisem:

1. Je-li δ kritické slovo, které je prefixem γ, pak $a(\gamma) = f(\delta)$.
2. Neexistuje-li žádné kritické slovo, které je prefixem γ, pak $a(\gamma) = \emptyset$.

Ve stránce příslušející kritickému slovu α je representována množina $h_S^{-1}(\alpha) \subseteq S$.

Zbývá dokázat, že neexistuje slovo α, které by patřilo do více stránek. Tzn. pro různá kritická slova β, γ platí $h_S^{-1}(\beta) \cap h_S^{-1}(\gamma) = \emptyset$, a tedy pro každé slovo α existuje nejvýše jedno kritické slovo, které je prefixem α. Je-li slovo α délky $d(S)$, pak nastává jeden ze tří případů:

1. $h_S^{-1}(\alpha) \neq \emptyset$, pak $0 < |h_S^{-1}(\alpha)| \leq b$ a existuje kritické slovo β, které je prefixem α. Tento případ nastává, pokud je α prefixem nějakého prvku z S.
2. $h_S^{-1}(\alpha) = \emptyset$ a existuje α' prefix α takový, že $0 < |h_S^{-1}(\alpha')| \leq b$, pak existuje kritické slovo, které je prefixem α', a tedy je též prefixem α. Tento případ nastává, pokud není α prefixem žádného prvku S, ale existuje $x \in S$, takové že má s α společný prefix nenulové délky.
3. $h_S^{-1}(\alpha) = \emptyset$ a pro každý prefix α' slova α platí buď $h_S^{-1}(\alpha) = \emptyset$, nebo $|h_S^{-1}(\alpha)| > b$. Pak je k α přiřazena stránka $\emptyset$. Tento případ nastává, pokud

není α prefixem žádného prvku S a každé $x \in S$ má s α jediný společný prefix (nulové délky).

Mějme slovo α o délce $d(S)$. Označme $c(\alpha)$ nejkratší prefix slova α takový, že stránka přiřazená slovu β o délce $d(S)$, které má prefix $c(\alpha)$, je stejná jako stránka přiřazená α.

Následující podmínky jsou ekvivalentní

1. stránka přiřazená slovu $\alpha \neq \emptyset$
2. $c(\alpha)$ je kritické slovo.
3. nějaký prefix α je kritické slovo.

Důkaz:

$1 \rightarrow 2)$ Slovu α je přiřazena nějaká stránka. Mezi stránkami a kritickými slovy existuje bijekce f. Tedy existuje nějaké kritické slovo $\delta = f^{-1}(\textit{stránka})$, kterému je přiřazena stejná stránka jako α. Navíc pro každé slovo α existuje nejvýše jedno kritické slovo, my ale víme, že existuje kritické slovo, které je prefixem α. Tedy δ je prefixem α. Zbývá dokázat, že každému slovu $\beta \in S$, které má za prefix δ, je přiřazena stejná stránka jako slovu α. Kdyby tomu tak nebylo, pak není δ kritické slovo, neboť mezi kritickými slovy a stránkami existuje bijekce a $|Im(f^{-1}(\delta))| > 1$ a jednomu kritickému slovu by odpovídalo více stránek.

$2 \rightarrow 3)$ $c(\alpha)$ je prefixem α.

$3 \rightarrow 1)$ Tento prefix označíme δ. $f(\delta)$ je stránka přiřazená α.

Funkci a nazýváme *adresář* a jeho velikost závisí na délce nejdelšího kritického slova a je rovna $2^{d(S)}$ a i-tá položka adresáře je adresa, která odpovídá i-tému slovu z $\{0,1\}^*$ délky $d(S)$ v lexikografickém uspořádání.

Operace FIND pro klíč x

1. Spočítáme $h(x)$, natáhneme adresář do paměti, zjistíme maximální délku kritického slova tj. $d(S)$ a najdeme adresu stránky odpovídající prefixu $h(x)$ délky $d(S)$.
2. Průchodem stránky zjistíme, zda obsahuje záznam příslušný klíči x.

Operace INSERT pro klíč x

1. Spočítáme $h(x)$, natáhneme adresář do paměti, zjistíme maximální délku kritického slova tj. $d(S)$ a najdeme adresu stránky odpovídající prefixu $h(x)$ délky $d(S)$.
2. Natáhneme stránku do paměti, obsahuje-li stránka záznam příslušný klíči x, algoritmus končí.
3. Obsahuje-li stránka méně než maximální počet prvků, vložíme x do této stránky a uložíme ji zpět na disk.

4. Obsahuje-li stránka maximální počet prvků $0 < |h^{-1}_{stránka}(\alpha)| = b$, stránku rozštěpíme, přerozdělíme záznamy mezi vzniklé dvě stránky, aktualizujeme adresář a vrátíme se zpět na krok 3.

   Poznámka: Při aktualizaci nastavíme adresu nově vzniklé stránky. Aktualizace adresáře může vyvolat zdvojnásobení jeho velikosti. Došlo totiž ke změně délce kritického slova, tedy mohlo dojít ke změně délky maximálního kritického slova, což právě vyžádá zvětšení tabulky.

Operace DELETE pro klíč x

1. Spočítáme $h(x)$, natáhneme adresář, nalezneme $d(S)$, nalezneme adresu stránky odpovídající prefixu $h(x)$ délky $d(S)$.
2. Natáhneme stránku do paměti.
3. Zjistíme, zda stránka obsahuje x, pokud ne, algoritmus končí.
4. Odstraníme x ze stránky. Označíme β slovo, které je prefixem α délky $d(\alpha) - 1$. Pokud $0 < |h^{-1}_{stránka}(\beta)| = b$, pak můžeme sloučit stránky příslušné bývalým kritickým slovům $\beta 0$ a $\beta 1$ do jediné stránky příslušné novému slovu β. To však nemusí být kritickým slovem a muže tedy dojít ke slučování dalších stránek. Změnou délky kritického slova se může změnit i délku maximálního kritického slova a zkrátit ho o 1. A tedy může dojít ke zmenšení adresáře na polovinu.

Implementace

Rozšiřitelné hašování předpokládá náhodné a rovnoměrné rozdělení klíčů. Vzhledem k tomu, že v naší databázi jsou klíče identifikátory (resp. jejich adresy), a k jednomu identifikátoru bude často existovat více záznamů, není možné spoléhat se na jednoznačnost nebo na malý počet stejných klíčů. Navíc zatím nemáme ani jistotu, že se záznamy pro jediný klíč vejdou do stránky, natož pak záznamy pro kolidující klíče. Avšak drobná modifikace předchozího postupu zajistí, že odhady pro výše zmíněné hašování budou přibližně platit. Z adresáře povede odkaz nikoli na stránku, ale na seznam stránek. Pokud zajistíme, aby tento seznam nebyl příliš velký, zůstanou zachovány všechny výhody externího hašování. Algoritmus pro vyhledání položky musí tedy zkontrolovat všechny záznamy stránek a nemůže se zastavit při nalezení prvního záznamu. Algoritmus pro vkládání se musí při vkládání do plné stránky umět rozhodnout, zda přehašuje obsahy stránek, nebo zda přidá stránku k již existujícímu seznamu. Jsou implementovány dva odlišné přístupy. Ten první přehašovává vždy, jsou-li po přehašování oba seznamy neprázdné. Tím se zajistí, že velikosti seznamů jsou minimální možné. Na druhou stranu však může vznikat velké množství téměř prázdných stránek a adresář může být zbytečně veliký. Druhý přístup přehašovává až ve chvíli, kdy oba seznamy po přehašování obsahují stanovený poměr prvků. Tento poměr lze stanovit při kompilaci definováním makra `BOUND`, které určuje, jaké procento prvků musí

každý vzniklý seznam obsahovat, aby nastalo štěpení jednoho seznamu a přehašování jeho prvků do dvou seznamů. Toto rozhodování je implementováno ve funkci `count_collisions`. Taková modifikace je téměř postačující, ale bude-li ve stránce příliš mnoho položek se stejným klíčem, budou zvětšovat seznam a zpomalovat hledání. Zlepšení dosáhneme tím, že pro klíče, které zabírají velké procento stránky (určuje makro `CHAIN_BOUND`), bude vyhrazena vlastní stránka. V ní není potřeba uvádět klíč u každé položky, neboť přísluší právě jednomu klíči, a tedy mohou v některých případech mohou klesnout i paměťové nároky databáze.

Program, jeho zdrojový kód a ukázky jeho užívání lze nalézt na přiloženém CD nebo na webových stránkách (kde najdete i případné aktualizace)
http://atrey.karlin.mff.cuni.cz/~susa/work/cspot/
http://freshmeat.net/projects/cspot/.

5. Literatura

[1] Brooks Rodney A. (1985):
Programming in common lisp.
Wiley, New York.

[2] Cameron Debra, Elliott James (1996):
Learning GNU Emacs, Second Edition.
O'Reilly, Sebastopol.

[3] Harrison Malcolm C. (1971):
Data-structures and programming.
Courant Institute of Mathematical Sciences, New York.

[4] Kernighan, Brian W. (2006):
Programovací jazyk C / – Vyd. 1..
Computer Press, Brno.

[5] Knuth Donald E. (1973):
The art of computer programming. [Vol.] 3, Sorting and searching.
Addison-Wesley, Reading.

[6] Lewine Donald A. (1991):
Posix programmer's guide : writing portable UNIX programs with the Posix.1 standard.
O'Reilly, Sebastopol.

[7] Lewis Bil (2000):
Gnu Emacs Lisp Reference Manual Ver 20.1.
Free Software Foundation, Boston, Massachusetts.

[8] Louis Dirk, Mejzlík Petr, Virius Miroslav (1999):
Jazyky C a C++ podle normy ANSI/ISO.
GRADA, Praha.

[9] Maurer W. D. (1972):
The programmers introduction to LISP.
MacDonald, London.

[10] McGrath Roland (1999):
Gnu C Library Reference Manual for Version 2.
Atlantic Books, London.

[11] Mehlhorn Kurt (1984):
Data structures and algorithms. 1, Sorting and searching.
Springer, Berlin.

[12] Stallman Richard M. (2000):
GNU Emacs Manual Version 20.7.
Free Software Foundation; 14th edition (2000), Boston, Massachusetts.

[13] Skočovský Luděk (1998):
UNIX POSIX Plan 9.
Luděk Skočovský, Brno.

[14] Standard jazyka C:
http://www.open-std.org/jtc1/sc22/wg14/www/docs/n1124.pdf

[15] cedet:
http://cedet.sourceforge.net/

[16] cscope:
http://cscope.sourceforge.net/

[17] ctags:
http://ctags.sourceforge.net/ctags.html

[18] GNU GLOBAL:
http://www.gnu.org/software/global/

[19] Knihovna GNU libc:
http://www.gnu.org/software/libc/

[20] Přednáška datové struktury – hašování:
http://ktiml.ms.mff.cuni.cz/downloads/hasovani.ps

[21] Regulární výrazy:
http://www.gnu.org/software/libc/manual/html_node/Regular-Expressions.html

[22] Zdrojový kód použitých červenočerných stromů:
http://www.ucw.cz/holmes/

[23] Zdrojový kód sparse:
http://www.codemonkey.org.uk/projects/git-snapshots/sparse/sparse-latest.tar.gz