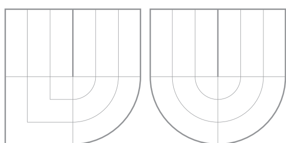
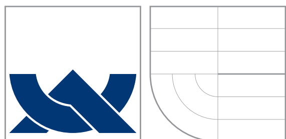
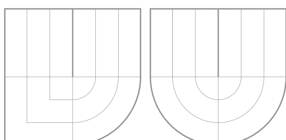
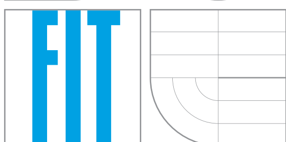




VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY


FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INTELIGENTNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INTELLIGENT SYSTEMS


# PODPORA PRO VYROVNÁVACÍ PAMĚŤ PRO SYSTÉM GVFS

A SUPPORT OF GVFS CACHING

DIPLOMOVÁ PRÁCE
MASTER'S THESIS


AUTOR PRÁCE — Bc. ONDŘEJ HOLÝ
AUTHOR

VEDOUCÍ PRÁCE — Ing. ALEŠ SMRČKA, Ph.D.
SUPERVISOR



BRNO 2014

*Zadání diplomové práce/16760/2013/xholyo00*

**Vysoké učení technické v Brně - Fakulta informačních technologií**

Ústav inteligentních systémů — Akademický rok 2013/2014

# Zadání diplomové práce

Řešitel: **Holý Ondřej, Bc.**
Obor: Inteligentní systémy
Téma: **Podpora pro vyrovnávací paměť pro systém GVFS**
**A Support of GVFS Caching**
Kategorie: Operační systémy

Pokyny:

1. Nastudujte různé typy vyrovnávacích pamětí (cache). Nastudujte projekt GVFS, jeho aplikační strukturu a komunikaci s moduly. Diskutujte nedostatky nejčastěji používaných modulů rozšiřujících GVFS o podporu dalších souborových systémů (tzv. GVFS backend).
2. Navrhněte podsystém GVFS, který rozšíří moduly GVFS o možnost použití cache.
3. Implementujte navržený podsystém cache pro GVFS jako rozšíření existujících modulů GVFS. Zaměřte se na moduly poskytující pouze základní operace nad souborovými systémy (např. GVFS backend pro přenos souborů přes bluetooth, protokol ftp a sftp).
4. Ověřte správnou funkcionalitu na vedoucím práce vybraných modulech. Proveďte srovnávací výkonnostní testy modulů s a bez použití cache. Diskutujte výhody a nevýhody použití cache pro GVFS.

Literatura:

- Tanenbaum, A.S: *Modern Operating Systems*, Prentice Hall, 3. vydání, ISBN 978-0136006633.
- Referenční příručka GIO, [online] https://developer.gnome.org/gio/stable/GVfs.html

Při obhajobě semestrální části diplomového projektu je požadováno:

- První dva body zadání

Podrobné závazné pokyny pro vypracování diplomové práce naleznete na adrese http://www.fit.vutbr.cz/info/szz/

Technická zpráva diplomové práce musí obsahovat formulaci cíle, charakteristiku současného stavu, teoretická a odborná východiska řešených problémů a specifikaci etap, které byly vyřešeny v rámci ročníkového a semestrálního projektu (30 až 40% celkového rozsahu technické zprávy).

Student odevzdá v jednom výtisku technickou zprávu a v elektronické podobě zdrojový text technické zprávy, úplnou programovou dokumentaci a zdrojové texty programů. Informace v elektronické podobě budou uloženy na standardním nepřepisovatelném paměťovém médiu (CD-R, DVD-R, apod.), které bude vloženo do písemné zprávy tak, aby nemohlo dojít k jeho ztrátě při běžné manipulaci.

Vedoucí: **Smrčka Aleš, Ing., Ph.D.**, UITS FIT VUT
Datum zadání: 1. listopadu 2013
Datum odevzdání: 28. května 2014

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
Fakulta informačních technologií
Ústav inteligentních systémů
612 66 Brno, Božetěchova 2

L.S.

doc. Dr. Ing. Petr Hanáček
*vedoucí ústavu*

# Abstrakt

Tato práce se zabývá podporou pro vyrovnávací paměť pro systém GVfs (Gnome Virtual filesystem). Nejprve je vysvětlen princip vyrovnávacích pamětí, invalidace dat a algoritmů pro výběr oběti. Následuje popis abstrakce souborového systému GIO a komunikace s virtuálními souborovými systémy GVfs. Dále jsou diskutovány nedostatky jednotlivých modulů tohoto systému a jejich interní vyrovnávací paměti.

V práci jsou navrženy tři typy vyrovnávacích pamětí. První slouží k ukládání metadat, druhá pro výpis adresářů a třetí pro data souborů. Tyto vyrovnávací paměti byly naimplementovány a ověřeny s ohledem na funkčnost a výkonnost. Hlavním přínosem tohoto řešení je rychlejší práce s virtuálními souborovými systémy a pokrytí chybějící funkcionality některých modulů (např. posun v datových tocích).



# Abstract

The master's thesis deals with a support of caching in GVfs (Gnome Virtual filesystem). Basics of caches, cache invalidation, and cache replacement algorithms are described. Description of GIO filesystem abstraction and communication of modules within GVfs is provided next. The disadvantages of individual GVfs modules and, if present, their internal caches are discussed.

The thesis proposes three types of cache. The first type of cache is for storing file metadata, the second one for directory listings, and the latter for file content. These caches have been implemented in a prototype and verified with respect to the functionality and performance. The main benefits of the proposed solution are faster work with virtual filesystems and provided missing functionality of lower-level virtual filesystem to GIO abstraction (namely, for instance seek operation).



# Klíčová slova

GLib, GIO, GVfs, vyrovnávací paměť, souborový systém, virtuální souborový systém

# Keywords

GLib, GIO, GVfs, Cache, Filesystem, Virtual filesystem


# Citace

Ondřej Holý: Podpora pro vyrovnávací paměť pro systém GVFS, diplomová práce, Brno, FIT VUT v Brně, 2014

# Podpora pro vyrovnávací paměť pro systém GVfs

## Prohlášení

Prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně pod vedením pana Ing. Aleše Smrčky, Ph.D. Odborné konzultace probíhaly ve spolupráci s panem Tomášem Bžatkem ze společnosti Red Hat Czech, s. r. o. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

........................<br>
Ondřej Holý<br>
28. května 2014

## Poděkování

Děkuji vedoucímu práce doktoru Aleši Smrčkovi za pedagogické vedení. Stejně tak děkuji konzultantovi Tomáši Bžatkovi za odbornou pomoc. Oběma jmenovaným děkuji za cenné rady a připomínky, které mi ochotně poskytovali během tvorby diplomové práce. V neposlední řadě děkuji své manželce za její vydatnou podporu.

© Ondřej Holý, 2014

*Tato práce vznikla jako školní dílo na Vysokém učení technickém v Brně, Fakultě informačních technologií. Práce je chráněna autorským zákonem a její užití bez udělení oprávnění autorem je nezákonné, s výjimkou zákonem definovaných případů.*

# Obsah

**1 Úvod** **3**

**2 Vyrovnávací paměti** **4**
2.1 Základní princip vyrovnávací paměti . . . 4
2.2 Koherenční protokoly . . . 5
2.3 Faktory ovlivňující výkon vyrovnávacích pamětí . . . 5
2.4 Algoritmy výběru oběti při zaplnění paměti . . . 6
2.4.1 Algoritmus LRU . . . 6
2.4.2 Size Adjusted LRU . . . 6
2.5 Vyrovnávací paměti v linuxovém jádře . . . 7
2.5.1 Vyrovnávací paměť i-uzlů . . . 8
2.5.2 Adresářová vyrovnávací paměť . . . 8

**3 Virtuální souborový systém GVfs** **9**
3.1 Programátorská knihovna GLib . . . 9
3.2 Abstrakce souborového systému GIO . . . 10
3.2.1 Reprezentace zařízení a souborového systému . . . 10
3.2.2 Reprezentace souboru a jeho metadat . . . 11
3.2.3 Základní operace . . . 12
3.3 Komunikační sběrnice D-Bus . . . 12
3.4 Virtuální souborový systém GVfs . . . 13
3.4.1 Architektura systému a komunikace . . . 14
3.4.2 Rozšiřující moduly a jejich operace . . . 16
3.4.3 Vyrovnávací paměť v démonech . . . 18
3.4.4 Nástroje pro práci s GVfs . . . 18
3.5 Překlad a spouštění pomocí JHBuild . . . 19

**4 Návrh vyrovnávacích pamětí** **20**
4.1 Možné využití vyrovnávacích pamětí . . . 20
4.2 Vyrovnávací paměť souborových metadat . . . 20
4.2.1 Nároky na vyrovnávací paměť . . . 21
4.2.2 Stromová struktura . . . 22
4.2.3 Samostatné vyrovnávací paměti . . . 22
4.2.4 Integrace vyrovnávacích pamětí . . . 24
4.3 Vyrovnávací paměť obsahu souborů . . . 25
4.3.1 Obecná vyrovnávací paměť . . . 25
4.3.2 Emulace základních operací . . . 26

**5 Implementace vyrovnávacích pamětí** **27**
5.1 Vyrovnávací paměti pro metadata a výpisy adresářů . . . . . . . . . . . . . . 27
5.1.1 Problematická invalidace dat . . . . . . . . . . . . . . . . . . . . . . . 28
5.1.2 Kontrola podmnožin parametrů . . . . . . . . . . . . . . . . . . . . . 29
5.1.3 Správa neplatných položek . . . . . . . . . . . . . . . . . . . . . . . . 29
5.1.4 Sdílení dat vyrovnávacích pamětí . . . . . . . . . . . . . . . . . . . . 29
5.1.5 Mezipaměti pouze pro čtení . . . . . . . . . . . . . . . . . . . . . . . 30
5.1.6 Vylepšení invalidace adresářů . . . . . . . . . . . . . . . . . . . . . . 30
5.1.7 Příznak podpory symbolických odkazů . . . . . . . . . . . . . . . . . 30
5.2 Vyrovnávací paměť pro emulaci proudových operací . . . . . . . . . . . . . . 30
5.2.1 Problém možné ztráty dat . . . . . . . . . . . . . . . . . . . . . . . . 30
5.2.2 Chybějící informace o průběhu přenosu . . . . . . . . . . . . . . . . . 31
5.2.3 Vytvoření GVfsJob bez D-Bus . . . . . . . . . . . . . . . . . . . . . . 31
5.3 Tvorba dokumentace . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 32
5.4 Integrace vyrovnávacích pamětí . . . . . . . . . . . . . . . . . . . . . . . . . 32

**6 Testování vyrovnávacích pamětí** **33**
6.1 Funkcionální testování . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 33
6.1.1 Testovací rozhraní GLib . . . . . . . . . . . . . . . . . . . . . . . . . 33
6.1.2 Jednotkové testy . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 33
6.1.3 Integrační testy . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 34
6.2 Výkonnostní testování . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 35
6.2.1 Výkonnostní testování PTP . . . . . . . . . . . . . . . . . . . . . . . 35
6.2.2 Výkonnostní testování SFTP . . . . . . . . . . . . . . . . . . . . . . 35

**7 Závěr** **40**

**A Podporované operace v modulech** **45**
A.1 Operace pro práci s metadaty . . . . . . . . . . . . . . . . . . . . . . . . . . 45
A.2 Operace pro práci s daty . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 45

**B Analýza komunikace s GVfs** **49**
B.1 Výpis adresáře aplikací Nautilus . . . . . . . . . . . . . . . . . . . . . . . . . 49
B.2 Výpis adresáře aplikací Tux Commander . . . . . . . . . . . . . . . . . . . . 50
B.3 Výpis adresáře démonem FUSE . . . . . . . . . . . . . . . . . . . . . . . . . 50
B.4 Výpis adresáře aplikací Thunar . . . . . . . . . . . . . . . . . . . . . . . . . 51
B.5 Výpis adresáře aplikací Double Commander . . . . . . . . . . . . . . . . . . 51

**C Obsah přiloženého DVD** **53**

# Kapitola 1

# Úvod

Chytré mobilní telefony, tablety, notebooky a stolní počítače jsou dnes již nedílnou součástí studentů či pracujících. Se vzrůstajícím počtem těchto elektronických zařízení narůstá potřeba přenosu dat či jejich sdílení. Běžnou součástí je také připojení k internetu, proto jsou v oblibě různé síťové protokoly a vzdálená úložiště.

Se správou souborů nám pomáhají souborové manažery. Pomocí manažerů lze jednoduše a jednotně přistupovat nejen k souborům na lokálních úložištích, ale i k souborům na připojených zařízeních a vzdálených úložištích. V GNU/Linuxu je několik manažerů postaveno na abstrakci souborového systémů GIO. Systém GVfs je rozšířením GIO a umožňuje připojení a práci s různými virtuálními souborovými systémy. Aplikace využívající toto aplikační rozhraní bohužel nejsou většinou optimalizovány a například přistupují k souborům opakovaně. Navíc GVfs jako celek neposkytuje vyrovnávací paměť, takže opakované dotazy zbytečně zpomalují práci a vytěžují síť či dané zařízení.

Cílem této práce je navrhnout a implementovat vyrovnávací paměť do systému GVfs. V kapitole 2 je diskutován princip vyrovnávacích pamětí, algoritmy pro výběr oběti a implementace těchto pamětí. Aplikační rozhraní GIO s rozšířením GVfs je popsáno v kapitole 3. Tato kapitola dále rozebírá nedostatky a interní vyrovnávací paměti jednotlivých modulů. Následující kapitola 4 se zabývá návrhem vyrovnávacích pamětí pro systém GVfs. Implementační detaily navrženého řešení jsou popsány v kapitole 5. Způsob a výsledky testování shrnuje kapitola 6. V poslední kapitole 7 lze nalézt zhodnocení vytvořené práce a náměty na další vývoj.

# Kapitola 2

# Vyrovnávací paměti

Vyrovnávací paměti jsou nedílnou součástí dnešních počítačů. Najdeme je v hardwaru, ale i softwaru. V této kapitole jsou blíže rozebrány. Základní princip je naznačen v sekci 2.1, další sekce 2.4 rozebírá způsob vybírání oběti v případě jejich zaplnění. Poslední sekce 2.5 se zabývá vyrovnávacími pamětmi v linuxovém jádře.

## 2.1 Základní princip vyrovnávací paměti

Vyrovnávací paměť (*Cache* resp. mezipaměť) slouží k uchovávání výsledků operací [1]. Pro určité vstupy jsou ukládány výstupy, aby je bylo možné v budoucnu jednoduše použít. Jedná se tedy o slovníkovou datovou strukturu, skladující páry klíč-hodnota, ve kterých potřebujeme rychle vyhledávat hodnoty podle daného klíče. V ideálním případě ukládáme do mezipaměti pouze taková data, která budeme brzy znovu potřebovat. V případě vyrovnávací paměti pro virtuální souborový systém může být klíčem cesta k souboru a hodnotou data souboru.

Vyrovnávací paměti jsou buď hardwarové nebo softwarové. Pro tuto práci je důležitá softwarová vyrovnávací paměť. Principy jsou však pro oba druhy podobné, ale liší se způsob implementace. Data u softwarové mezipaměti mohou být ukládána do operační paměti nebo na disk.

Pokud jsou požadována data po vyrovnávací paměti, mohou nastat dvě situace. Data v paměti jsou (*Cache Hit*) nebo nejsou (*Cache Miss*) přítomny. Je snaha poměr mezi těmito případy (*Hit Rate* resp. *Miss Rate*) maximalizovat resp. minimalizovat.

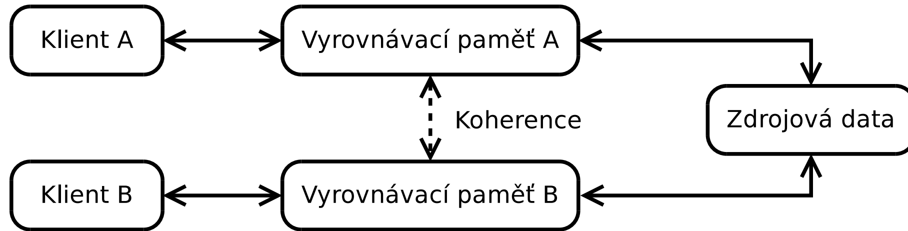
Rozlišujeme několik způsobů zápisu [2]:

- *Write-Through* – v případě zápisu do vyrovnávací paměti dochází okamžitě i k zápisu do zdrojových dat,
- *Write-Back* – zdrojová data jsou změněna až ve chvíli, kdy je to potřeba (např. když má být klíč nahrazen).

**Buffer** Buffer je speciálním typem vyrovnávací paměti [1]. Slouží k vyrovnání různých rychlostí dvou stran. Data jsou do bufferu zapsána jednou stranou, přečtena druhou a již nejsou déle uchována.

**Zneplatnění** Zneplatnění (dále v práci bude také používán termín invalidace) je situace, kdy se data ve vyrovnávací paměti stávají neplatnými z důvodu změny zdrojových dat [1]. To můžeme ilustrovat na obrázku 2.1, např. když *Klient A* upraví *Zdrojová data*, potom jsou korespondující data ve *Vyrovnávací paměti B* neplatná.

**Koherence** Koherence vyjadřuje konzistenci dat v lokálních vyrovnávacích pamětech [2]. Například pokud upraví *Klient A* data na obrázku 2.1 ve *Vyrovnávací paměti A*, měla by se tato změna projevit i ve *Vyrovnávací paměti B* invalidací těchto dat. Tato situace je řešena pomocí koherenčních protokolů.




Obrázek 2.1: Koherence dat ve vyrovnávacích pamětech

## 2.2 Koherenční protokoly

Koherenční protokoly (*Cache Coherence Protocol*) zajišťují koherenci mezi vyrovnávacími pamětmi a zdrojovými daty [1]. Komplikovanost těchto protokolů závisí zejména na způsobu zápisu. V případě, že jsou změny zapisovány ihned do zdrojových dat, stačí položku v ostatních pamětích zneplatnit či aktualizovat. Tento zápis je ale poměrně neefektivní.

Pro vyrovnávací paměti s opožděným zápisem je však situace složitější. Lze to opět ilustrovat na obrázku 2.1. Položka je upravená *Klientem A* ve *Vyrovnávací paměti A*, pokud chce *Klient B* číst tyto data, musí dostat jejich aktualizovanou verzi. Proto se nejprve provede zápis do *Zdrojových dat* a následně do *Vyrovnávací paměti B*. Jednotlivé položky mezipamětí se mohou nacházet v různých stavech:

- modifikovaný (*Modified*) – data jsou upravená,
- sdílený (*Shared*) – data jsou platná pro čtení nebo
- neplatný (*Invalid*) – data jsou neplatná.

Koherenční protokol *MSI* využívá tyto tři stavy, jejichž první písmena tvoří jeho název. Jedná se o základní protokol ve víceprocesorových systémech. Pokud chce klient zapisovat, musí být daná položka v ostatních pamětích v neplatném stavu. Pak je možno data zapsat a změnit sdílený stav na modifikovaný. Protokol MESI je jeho vylepšením a přidává čtvrtý stav, exklusivní (*Exclusive*). Ten vyjadřuje, že jsou v něm platná data, která lze modifikovat (nejsou v jiných vyrovnávacích pamětích). Podrobnější popis těchto protokolů je však nad rámec této práce.

## 2.3 Faktory ovlivňující výkon vyrovnávacích pamětí

Efektivita vyrovnávacích pamětí závisí na různých faktorech, často protichůdných. Následuje výčet některých z nich, které mají smysl pro souborové vyrovnávací paměti [4]:

- velikost vyrovnávací paměti – obecně větší vyrovnávací paměť poskytuje větší výkon,
- velikost ukládaných položek – preferování malých souborů obvykle zvyšuje výkon v případě rozdílně velkých položek a
- časová lokalita – položky s nedávným přístupem budou pravděpodobně brzy znovu potřeba.

## 2.4 Algoritmy výběru oběti při zaplnění paměti

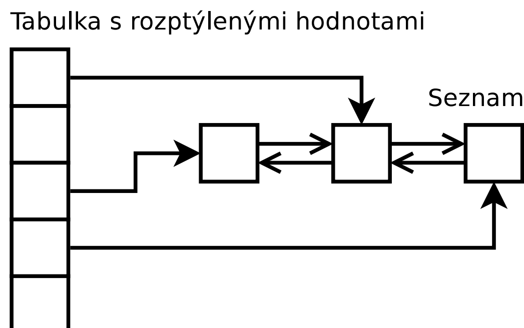
Velikost vyrovnávací paměti je obvykle limitována. Pokud dojde k zaplnění, je potřeba odstranit nějaká data a nahradit je novými. Výběr vhodných kandidátů k nahrazení lze určit např. pomocí následujících algoritmů [3]:

- náhodný – klíč určíme zcela náhodně,
- fronta (*FIFO*) – klíč nejdéle v paměti nebo
- *Least Recently Used* (*LRU*) – nejdéle nepoužitý klíč.

Všechny tyto algoritmy mají svoje výhody a nevýhody. Optimální algoritmus by odstranil klíč, který bude potřeba za nejdelší dobu, to však obecně nelze určit.

### 2.4.1 Algoritmus LRU

Algoritmus LRU je jedním z nejpoužívanějších algoritmů pro výběr oběti vůbec [4]. Softwarová implementace vyrovnávací paměti s LRU je naznačena na obrázku 2.2. Vyrovnávací paměť je zde tvořena tabulkou s rozptýlenými hodnotami (*Hash Table*). Pro výpočet LRU je použit dvousměrný seznam [3], existují však provedení např. pomocí dvou jednosměrně vázaných seznamů [5]. Klíč se použije jako vstup do mapovací funkce. V tabulce jsou struktury obsahující hledaná data, součástí této struktury je také ukazatel do seznamu. Prvky seznamu obsahují dané klíče. Při přidání položky dojde k přesunu prvku seznamu na začátek, stejně jako při jejím vyhledání. Na konci seznamu tedy máme nejdéle nepoužité klíče, které můžeme v případě přeplnění vyrovnávací paměti odstranit.




Obrázek 2.2: Softwarová implementace vyrovnávací paměti

Pseudokód jednotlivých funkcí je naznačen v algoritmu 2.1. V ideálním případě mají operace vyhledání a vložení (resp. vyhledání oběti) konstantní časovou složitost. Tato složitost však předpokládá ideální mapovací funkci.

### 2.4.2 Size Adjusted LRU

Size Adjusted LRU je rozšířením algoritmu LRU pro různě velké objekty [4]. Položky ve virtuální paměti jsou seřazeny podle koeficientu:

$$\frac{1}{velikost \cdot čas_od_posledního_přístupu}$$

Výpočty tohoto koeficientu jsou náročné, a proto jsou objekty rozděleny do omezeného počtu tříd. Třídy jsou identifikovány číslem o hodnotě $log_2(velikost)$, takže objekty ve

**Algoritmus 2.1** Softwarové řešení vyrovnávací paměti s LRU

```
1  vyrovnávací_paměť
2  {
3    tabulka (klíč, položka)
4    seznam (klíč)
5  }
6
7  položka
8  {
9    data
10   prvek_seznamu
11 }
12
13 vyhledej (vyrovnávací_paměť, klíč)
14 {
15   položka = vyhledej (vyrovnávací_paměť->tabulka, klíč)
16   if (položka nalezena)
17     přesuň_na_začátek (vyrovnávací_paměť->seznam,
18                        položka->prvek_seznamu)
19     return položka->data
20 }
21
22 vlož (vyrovnávací_paměť, klíč, data)
23 {
24   položka = vyhledej (vyrovnávací_paměť->tabulka, klíč)
25   if (položka nalezena)
26     položka->data = data;
27     přesuň_na_začátek (vyrovnávací_paměť->seznam,
28                        položka->prvek_seznamu)
29
30   else
31     prvek_seznamu = vlož_na_začátek (klíč)
32     položka = vytvoř (data, prvek_seznamu)
33     vlož (vyrovnávací_paměť->tabulka, položka)
34
35     if (max_velikost < velikost (vyrovnávací_paměť->seznam))
36       klíč = smaž_poslední (vyrovnávací_paměť->seznam)
37       smaž (vyrovnávací_paměť->tabulka, klíč)
38 }
```

stejné třídě mají podobnou velikost. Jednotlivé třídy jsou spravovány LRU algoritmem, kde každá třída má vlastní seznam. Oběť se pak vybírá z konců všech seznamů jako prvek s největším koeficientem $velikost \cdot čas_od_posledního_přístupu$.

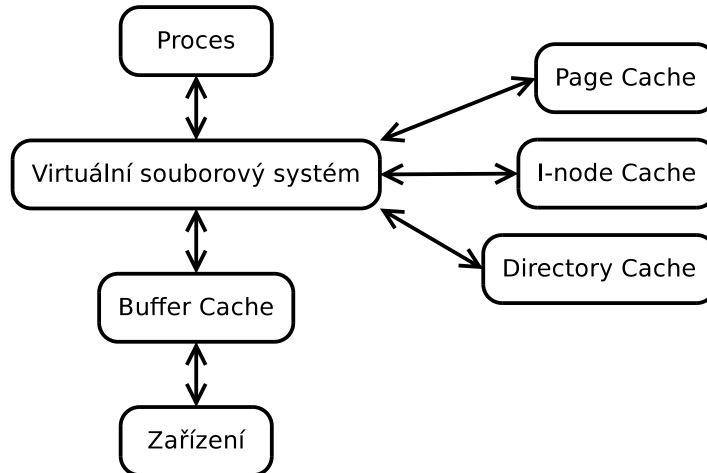
## 2.5 Vyrovnávací paměti v linuxovém jádře

Virtuální souborový systém (VFS) v linuxovém jádře poskytuje jednotné rozhraní pro přístup k souborům [5]. Tato vrstva umožňuje připojení různých souborových systému do jedné stromové struktury. VFS poskytuje jednotné vstupně-výstupní vyrovnávací paměti. Konkrétně se jedná o paměti čtyř typů:

- stránek (*Page Cache*) – kombinuje souborová data a virtuální paměť,

- i-uzlů (*I-node Cache*) – uchovává nedávno přistupované i-uzly,
- bloků (*Buffer Cache*) – ukládá bloky dat při čtení ze zařízení a
- adresářů (*Directory Cache*) – stromová vyrovnávací paměť uchovávají část adresářové struktury.

Vzájemné propojení těchto pamětí je naznačeno na obrázku 2.3.




Obrázek 2.3: Vyrovnávací paměti VFS v linuxovém jádře

### 2.5.1 Vyrovnávací paměť i-uzlů

Při procházení souborového systému dochází k neustálému čtení i-uzlů. Každý soubor či adresář je reprezentován ve VFS i-uzlem, a proto má jádro pro urychlení přístupu mezipaměť i-uzlů [6]. Pokud je i-uzel vyhledán ve vyrovnávací paměti, ušetří se tím přístup k fyzickému zařízení. Vyrovnávací paměť je vytvořena pomocí tabulky s rozptýlenými hodnotami a seznamů. Vstupem do mapovací funkce je číslo i-uzlu a číslo zařízení, pro konkrétní souborový systém.

Při úspěšném vyhledání je inkrementována hodnota počítadla. Při zaplnění se odstraňují položky s nejmenší hodnotou, tedy nejméně používané.

### 2.5.2 Adresářová vyrovnávací paměť

Jádro obsahuje adresářovou vyrovnávací paměť pro urychlení přístupu k často používaným adresářům [6]. Při vyhledání adresáře dojde k jeho přidání do mezipaměti. Paměť je implementována jako tabulka s rozptýlenými hodnotami. Mapovací funkce má na vstupu číslo zařízení a jméno adresáře.

Opět se zde používá modifikace algoritmu LRU. Vyrovnávací paměť obsahuje dvě úrovně. Nově přidané položky jsou v jedné, objekty s opakovaným přístupem v druhé. Pokud v jedné či druhé úrovni není místo, je nahrazen poslední objekt v příslušné úrovni.

# Kapitola 3

# Virtuální souborový systém GVfs

Virtuální souborový systém GVfs je vystavěn na knihovně GLib. Sekce 3.1 nejprve popisuje nejdůležitější části této knihovny, sekce 3.2 se pak zabývá abstrakcí souborového systému GIO. V sekci 3.3 je krátce rozebrána komunikační sběrnice D-Bus. Zbylé sekce se pak již věnují konkrétním částem GVfs.

## 3.1 Programátorská knihovna GLib

*GLib*[1] je univerzální multiplatformní programátorská knihovna [7]. Tato knihovna je základem grafického uživatelského prostředí *GNOME*[2]. GLib poskytuje různé datové typy, funkce pro práci s řetězci a pamětí, operace se soubory aj. Nabízí i pokročilé datové struktury – pro implementaci samotné vyrovnávací paměti se může použít např. obousměrně vázaný seznam `GList` či tabulka s rozptýlenými hodnotami `GHashTable`. Následuje popis několika částí, které jsou pro tuto práci klíčové.

**GObject** `GObject` poskytuje objektově orientovaný systém pro jazyk C [8]. Používá existující konstrukty jazyka, zejména struktury a makra. Zavádí třídní typový systém s jednoduchou dědičností. Základ pro `GObject` tvoří typový systém `GType` [9]. Tento systém obsahuje běhové informace o typech a má prostředky pro manipulaci s nimi. `GType/GObject` lze jednoduše mapovat do různých objektově orientovaných jazyků. Na tomto typovém systému je vystavěna celá knihovna.

**GError** Hlášení chyb z volaných funkcí se provádí pomocí objektu `GError` [10]. Společně s pomocnými funkcemi se jedná o ekvivalent výjimek v jiných jazycích. `GError` zapouzdřuje chybovou doménu, kód a zprávu. Uvádí se obvykle jako poslední parametr funkce.

**GCancellable** Třída `GCancellable` poskytuje mechanismus přerušení operace [11]. Lze jej používat při práci ve více vláknech. Umožňuje jednotný způsob přerušení pro synchronní i asynchronní operace.

**GAsyncResult** `GAsyncResult` je základní třídou pro získávání výsledků asynchronních operací [12]. Tyto operace jsou rozděleny na dvě části, které jsou propojeny funkcí zpětného volání `GAsyncReadyCallback`. Do asynchronní funkce se vloží ukazatel na funkci zpětného volání, která je vyvolána po dokončení operace. Volaná funkce dostane za parametr objekt

[1] `https://developer.gnome.org/glib/`
[2] `https://gnome.org/`

GAsyncResult, který obsahuje informace o operaci a objekt asynchronní metody. Zmíněný objekt se pak využije k volání příslušné funkce pro pokračování původního algoritmu.

**GThreadPool** Pro snadné výpočty ve vláknech poskytuje GLib třídu GThreadPool [13]. Programátor se nemusí starat o vytváření vláken či jejich rušení. GThreadPool využívá již vytvořená vlákna, čímž odpadá i určitá režie.

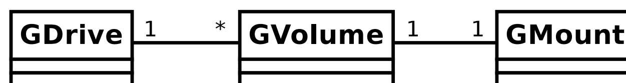
**GMappedFile** Objekt GMappedFile obaluje systémovou funkci mmap [14]. Ta umožňuje načtení potřebných částí souboru do paměti. Toho se dá využít mimo jiné při tvorbě mezipaměti pro velké soubory.

## 3.2 Abstrakce souborového systému GIO

Knihovna GLib poskytuje abstrakci souborového systému GIO. Tuto abstrakci využívá spousta knihoven či aplikací (např. souborové manažery *Nautilus*[3], *PCManFM*[4]). Nejprve je vysvětlena reprezentace zařízení se souborovými systémy v sekci, následuje popis abstrakce souboru (resp. adresáře) a možných operací.

### 3.2.1 Reprezentace zařízení a souborového systému

K zařízením a souborovým systémům se pojí několik tříd. Na obrázku 3.1 jsou znázorněny třídy reprezentující zařízení se souborovými systémy, jejich popis je v následujících odstavcích.




Obrázek 3.1: Třídy reprezentující souborový systém

**GDrive** Objekt GDrive reprezentuje hardware připojený k počítači [15]. Může se jednat o připojitelný hardware (např. USB disk) nebo hardware s vyjímatelnými médii (např. CD mechanika). Jedná se o kontejner, který obsahuje objekty GVolume pro jedno zařízení. Objekt GDrive poskytuje metody pro spuštění či zastavení. Povoluje práci i se zařízeními vyžadujícími autentizaci.

**GVolume** GVolume je abstrakcí uživatelsky viditelného objektu, který může být připojen do systému (např. logický oddíl na disku) [16]. Poskytuje metody pro připojení, odpojení, či vysunutí.

**GMount** Souborový systém, ke kterému lze přistupovat, je reprezentován objektem GMount [17]. Není nutně vázaný na GVolume, může se jednat např. o souborový systém z GVfs. Autentizace a interakce s uživatelem je při připojování prováděna přes objekt GMountOperation [18].

[3] https://wiki.gnome.org/action/show/Apps/Nautilus
[4] http://wiki.lxde.org/en/PCManFM

**GVolumeMonitor** `GVolumeMonitor` poskytuje přístup k lokálně dostupným souborovým systémům [19]. GVfs obsahuje několik implementací, v současnosti je preferovaný *udisks2*[5] monitor [20].

### 3.2.2 Reprezentace souboru a jeho metadat

Další částí GIO jsou třídy související se soubory a jejich metadaty. Obsahem této sekce je jejich bližší popis.

**GFile** `GFile` je vysokoúrovňová abstrakce pro práci se soubory [21]. Objekty nereprezentují fyzické soubory, jedná se pouze o identifikátory. Soubory jsou jednoznačně reprezentovány pomocí *URI*[6]. Lokální souborový systém začíná schématem „`file://`". `GFile` neprovádí přímo žádné vstupně-výstupní operace, ale využívá k tomu objekty reprezentující datové toky.

Ukázka připojení k nezabezpečenému FTP serveru pomocí GIO je v algoritmu 3.1. Tento postup je univerzální pro všechny podporované souborové systémy. Tímto způsobem je možné přistupovat i na lokální souborový systém, který se připojovat nemusí. Případná autentizace by se prováděla pomocí objektu `GMountOperation`, který je parametrem funkce `g_file_mount_enclosing_volume`.

**Algoritmus 3.1** Ukázka připojení k FTP pomocí GIO API

```
 1 void
 2 callback (GObject *object, GAsyncResult *res, gpointer data)
 3 {
 4   GError *error = NULL;
 5   GFile *file = G_FILE (object);
 6
 7   g_file_mount_enclosing_volume_finish (file, res, &error);
 8   if (error)
 9   {
10     g_error_free (error);
11     return;
12   }
13
14   /* operace se soubory na FTP */
15
16   g_object_unref (file);
17 }
18
19 GFile *file = g_file_new_for_uri ("ftp://example.com/");
20 g_file_mount_enclosing_volume (file, 0, NULL, NULL, callback, 0);
21 g_object_unref (file);
```

**GFileInfo** Objekt `GFileInfo` zapouzdřuje souborová metadata a poskytuje metody pro práci s nimi [22]. Metadaty jsou zde myšleny jednotlivé atributy jako např. typ souboru, jeho velikost, přístupová práva, ikona. Tyto atributy, dvojice klíč-hodnota, jsou reprezentovány objektem `GFileAttribute` [23]. Filtrace atributů se provádí pomocí objektu `GFileAttributeMatcher`.

[5] `http://www.freedesktop.org/wiki/Software/udisks/`
[6] `http://tools.ietf.org/html/rfc3986`

**GFileEnumerator** Tento objekt slouží k výpisu obsahu adresáře [24]. Pomocí jeho metod lze získat objekty `GFileInfo` pro všechny obsažené soubory.

### 3.2.3 Základní operace

GIO je navrženo pro manipulaci se souborovými daty pomocí proudových operací. Objekty `GFileInputStream`, `GFileOutputStream` a `GFileIOStream` reprezentují příslušné datové toky, které umožňují přístup k datům. Implementována je podpora i pro posun v těchto tocích, pokud je to podporováno příslušným souborovým systémem.

Metody `GFile` pro manipulaci se soubory jsou v synchronních i asynchronních verzích. Asynchronní provádění je implementováno pomocí již zmíněného `GAsyncResult`. Následuje výčet synchronních metod pro práci s metadaty:

- `g_file_query_filesystem_info` – získání `GFileInfo` o souborovém systému,
- `g_file_query_info` – získání `GFileInfo` o souboru,
- `g_file_enumerate_children` – získání obsahu adresáře pomocí `GFileEnumerator` a
- `g_file_set_display_name` – přejmenování souboru.

Základní metody pro čtení dat souboru jsou:

- `g_file_append_to`, `g_file_create`, `g_file_replace` – zápis dat do souboru pomocí metod `GFileOutputStream`,
- `g_file_delete`, `g_file_trash` – odstranění souboru resp. přesunutí do koše,
- `g_file_copy`, `g_file_move` – kopírování resp. přesunutí souboru a
- `g_file_replace_contents` – nahrazení obsahu celého souboru z bufferu.

Následují základní metody pro zápis dat souboru:

- `g_file_read` – čtení obsahu souboru pomocí metod `GFileInputStream` a
- `g_file_load_contents` – načtení obsahu celého souboru do bufferu.

Pomocí objektu `GFileProgressCallback` lze u některých operací sledovat průběh přenosu, kolik dat již bylo přeneseno a kolik ještě zbývá. Operace lze přerušit pomocí `GCancellable` zmíněného dříve. Případné chyby jsou poskytované objektem `GError`.

Z dalších operací stojí za zmínku `g_file_new_tmp`. Tato metoda vytvoří soubor v dočasném adresáři a vrátí jeho abstrakci, čehož se dá opět využít při implementaci vyrovnávací paměti.

## 3.3 Komunikační sběrnice D-Bus

*D-Bus*[7] je multiplatformní systémová sběrnice pro zasílání zpráv [25]. Jedná se o univerzální způsob komunikace mezi aplikacemi. D-Bus byl navržen pro práci v rámci lokálního počítače. Existují však možnosti propojení sběrnic přes více počítačů [26]. Posílaná data jsou serializovaná, komunikační protokol je binární. K přenosu zpráv jsou využívány unixové sokety.

[7] `http://www.freedesktop.org/wiki/Software/dbus/`

Typicky v systému najdeme dvě instance:

- systémová sběrnice (*System Bus*) – komunikace mezi aplikacemi či systémem a
- uživatelská sběrnice (*User Bus*) – komunikace mezi aplikacemi v rámci jednoho uživatelského sezení.

Je však možné vytvořit vlastní soukromé instance, např. pro komunikaci mezi dvěma uzly [27]. Nové instance se vytvoří funkcí `dbus_connection_open_private`, která otevře spojení na zadaném soketu. Toho také využívá systém GVfs, aby uživatelskou sběrnici příliš nezahlcoval komunikací.

D-Bus se skládá z několika částí:

- objekty (*Objects*) – koncové body komunikace,
- obalová rozhraní (*Proxies)* – pro přístup k objektům z aplikace,
- metody (*Methods*) – klientské požadavky na objekt vyvolají metody,
- signály (*Signals*) – jednosměrná komunikace z objektů pro příjímání klienty a
- rozhraní (*Interfaces*) – specifikace signálů a metod.

Sběrnice D-Bus využívá textové signatury (*Signatures*) pro popis parametrů metod a signálů. Signatury jsou dále použity pro serializaci a kontrolu validity. Kromě základních datových typů jsou zahrnuty i různé kontejnery (např. struktura, pole a slovník).

**GDBus** Obalová rozhraní jsou portována do různých jazyků či knihoven [28]. `GDBus` je implementace obsažená v knihovně GLib. D-Bus objekty jsou zde namapovány na `GObject`.

## 3.4 Virtuální souborový systém GVfs

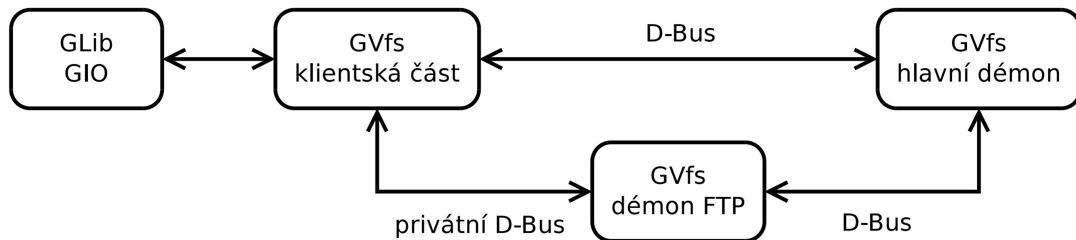
GVfs (*Gnome Virtual Filesystem*) je množina knihoven a modulů (*Backend*, *Daemon* resp. démon) rozšiřujících GIO [20]. GIO má aplikační rozhraní pro tvorbu rozšíření, které GVfs implementuje. GVfs si registruje URI schémata, které dokáže obsloužit. Pomocí modulů poskytuje přístup k různým vzdáleným souborovým systémům (např. FTP, SMB, WebDAV), speciálním systémovým místům (mj. koš, časté soubory) apod.

Pro aplikace, které nevyužívají aplikační rozhraní GIO, je poskytováno záchranné řešení pomocí *FUSE*[8]. K souborům lze přistupovat v umístění „`/run/user/<UID>/gvfs/`“ (případně „`$HOME/.gvfs/`“) pomocí klasického POSIX rozhraní. Pokud je GVfs spuštěno s podporou FUSE, metoda `g_file_get_path` vrací cesty k souborům v tomto umístění. Proto je možné ze správců souborů využívajících GIO snadno otevřít soubory v aplikacích bez podpory GIO. Pokud je tento soubor otevřen aplikací podporující GIO, dojde k přemapování cesty na URI a komunikace probíhá přímo. Jedná se však pouze o nouzové řešení a bohužel ne vše je podporováno, což je důsledek různých aplikačních rozhraní.

[8] `http://fuse.sourceforge.net/`

### 3.4.1 Architektura systému a komunikace

Architektura systému GVfs je naznačena na obrázku 3.2. Jelikož je GVfs implementováno jako rozšíření GIO, jsou jeho knihovny načítány v každém procesu. Z tohoto důvodu byla zvolena architektura klient-server a načítají se pouze klientské knihovny. Komunikace probíhá po komunikační sběrnici D-Bus. V serverové části se pak ukrývá hlavní část – tu tvoří hlavní démon a jednotlivé moduly pro připojené souborové systémy. Následuje popis těchto částí.




Obrázek 3.2: Architektura a komunikace v systému GVfs

#### Klientská část

Klientská část je tvořena pouze obalovými rozhraními sběrnice D-Bus. Staví na objektech `GDaemonFile` a `GDaemonVfs`, jak je patrné z diagramu tříd na obrázku 3.3. Třída `GDaemonFile` implementuje rozhraní `GFile`. Objekt `GDaemonVfs` je implementací rozšíření GIO [30, 29].

#### Serverová část GVfs

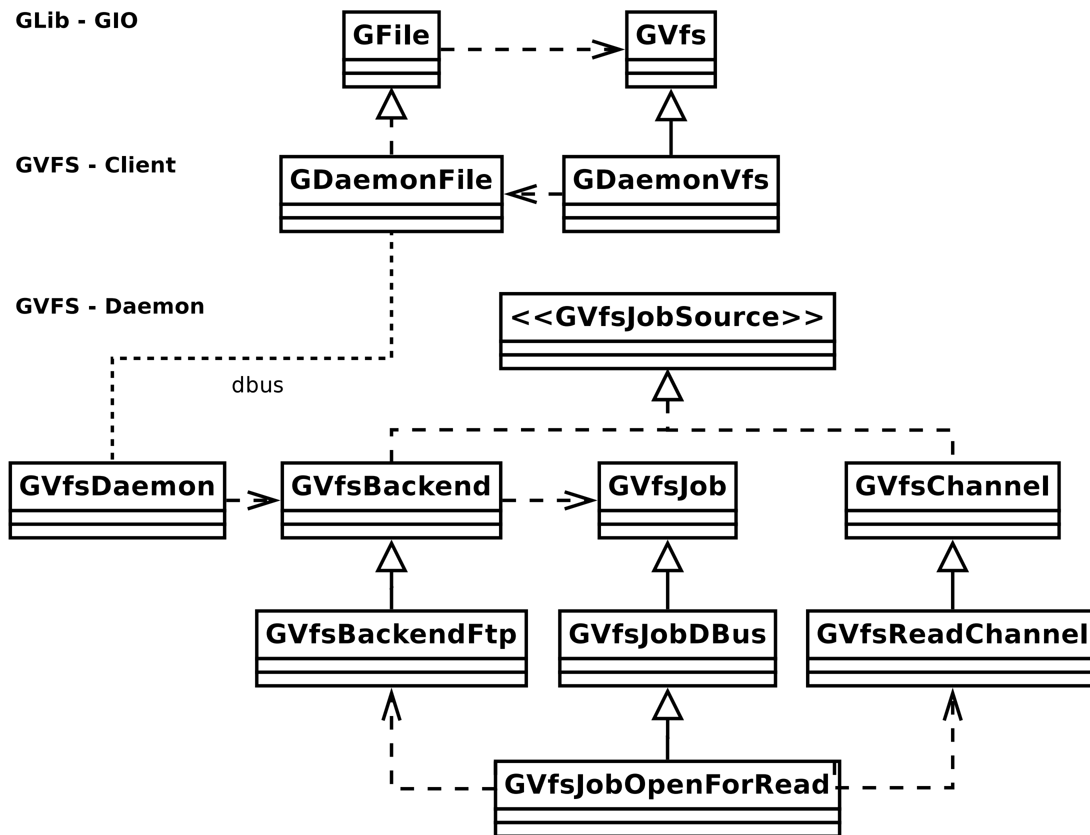
Hlavní démon spravuje jednotlivé démony a přeposílá zprávy z D-Bus od klientské části [20]. Je inicializován na klientské straně a automaticky spuštěn při prvním požadavku. Hlavní démon spouští, připojuje a odpojuje další démony a informuje klientskou stranu, který z modulů poskytuje požadovaný soubor.

Všichni démoni jsou instance `GVfsDeamon`. Běžně existuje pro jeden přípojný bod `GMount` samostatný proces – tím je zajištěna robustnost systému, díky níž chyba jednoho modulu nezpůsobí pád celého systému.

**GVfsBackend** Ve třídách dědících z `GVfsBackend` je samotná vstupně-výstupní logika. Jsou zde dva typy metod, které se navzájem liší prefixem. Verze začínající prefixem „`try_`“ jsou rychlé a neblokující (čtou z vyrovnávací paměti nebo se provádí asynchronně), verze s „`do_`“ (resp. bez prefixu) jsou pomalé a blokující.

**GVfsJob** Objekty `GVfsJob` zapouzdřují všechny parametry operace obdržené přes sběrnici D-Bus. Obalují metody třídy `GVfsBackend`, o jejichž zpracování se stará `GVfsDaemon`.

**GVfsDaemon** `GVfsDaemon` provádí operace objektu `GVfsBackend` a spravuje kanály `GVfsChannel`. Při přidání nového objektu `GVfsJob` jsou podle implementovaných metod prováděny operace. Neblokující metody s prefixem „`try_`“ jsou spuštěny hned, pomalé s prefixem „`do_`“ jsou spouštěny ve vláknech pomocí `GThreadPool` zmíněného dříve.




Obrázek 3.3: Diagram hlavních tříd

**GVfsChannel** V případě přenosu souborových dat je vytvořen kanál `GVfsChannel`. Data se tedy nepřenáší přes D-Bus kvůli rychlosti, ale jednoduchým protokolem přes sokety. `GVfsChannel` se stará o velikost bufferu, dopředné načítání aj.

V algoritmu 3.1 byla naznačena práce se souborem na FTP. Metoda `g_file_mount_enclosing_volume` zde způsobí, že je přes D-Bus vyvolána metoda hlavního démona. Hlavní démon zkontroluje, zda je již umístění „`ftp://example.com/`“ připojeno. V negativním případě spustí proces nového démona. Po spuštění je přes D-Bus vyvolána metoda pro připojení k tomuto serveru, při jejímž přijetí je vytvořen objekt `GVfsJobMount`, který reprezentuje požadovanou operaci. Tento objekt je předán do `GVfsBackendFtp`, kde se provede požadovaná metoda. Výsledek je poté vrácen zpět hlavnímu démonu a následně klientovi. Při dalších požadavcích na soubory umístěné na „`ftp://example.com/`“ vrátí hlavní démon klientovi pouze adresu na konkrétního démona. Další komunikace pokračuje na privátním D-Bus spojení.

Pro účely ladění je možné spustit vlastní démony ručně z příkazové řádky. Následuje ukázka spuštění hlavního démona a FTP démona s ladícími výstupy:

```
export GVFS_DEBUG=1
/usr/libexec/gvfsd --replace &
/usr/libexec/gvfsd-ftp host="ftp://example.com/"
```

### 3.4.2 Rozšiřující moduly a jejich operace

GVfs podporuje přístup k množství různých souborových systémů skrze moduly. Následuje výčet všech modulů s popisem:

- `GVfsBackendAfc` – protokol AFC (iPhone, iPad),
- `GVfsBackendAfp` – protokol AFP (Mac OS X),
- `GVfsBackendAfpbrowse` – AFP zařízení na síti,
- `GvfsBackendArchive` – zpřístupnění archivů,
- `GVfsBackendBurn` – vypalování souborů na optické disky,
- `GVfsBackendCdda` – zpřístupnění audio CD,
- `GVfsBackendComputer` – zařízení připojená k počítači,
- `GVfsBackendDav` – protokol WebDAV,
- `GVfsBackendDnssd` – DNS–SD služby na síti (AFP, FTP, SFTP, SMB, WebDAV),
- `GVfsBackendFtp` – protokol FTP,
- `GVfsBackendGphoto2` – protokol PTP (fotoaparáty a audio přehrávače),
- `GVfsBackendHttp` – protokol HTTP,
- `GVfsBackendLocaltest` – lokální souborový systém pro testovací účely,
- `GVfsBackendMtp` – protokol MTP (chytré mobilní telefony),
- `GVfsBackendNetwork` – zobrazení sítě,
- `GVfsBackendObexftp` – protokol Bluetooth (mobilní telefony),
- `GVfsBackendRecent` – často používané soubory,
- `GVfsBackendSftp` – protokol SFTP,
- `GVfsBackendSmb` – protokol SMB resp. CIFS (Microsoft Windows),
- `GVfsBackendSmbbrowse` – SMB zařízení na síti,
- `GVfsBackendTest` – souborový systém pro testovací účely a
- `GVfsBackendTrash` – koš.

Pro další zkoumání budou vynechány systémové moduly či moduly pro testování.

#### Operace pro práci s metadaty

V tabulce A.1 jsou shrnuty podporované operace pro práci s metadaty. Názvy metod odpovídají metodám `GFile` (resp. `GDaemonFile`).

Jak je patrné z tabulky, některé metody nejsou implementovány. U démonů, které jsou jen pro čtení, chybí operace `set_display_name` a `set_attribute`. Někde funkce `set_attribute` také chybí z důvodu chybějící podpory daných protokolů či knihoven, na kterých jsou moduly vystavěny. Některé operace zde nejsou uvedeny, protože ve většině případů jejich podpora chybí (např. `query_info_on_read` a `query_info_on_write`), při implementaci vyrovnávací paměti však musí být ošetřeny.

### Operace pro práci s daty

Tabulky A.2 a A.3 shrnují podporované operace čtení a zápisu souborových dat. Moduly `GVfsBackendCdda`, `GVfsBackendHttp` a `GVfsBackendArchive` jsou však pouze pro čtení. Aplikační rozhraní `GFile` zde neodpovídá přesně metodám třídy `GVfsBackend` – v případě přesunování či kopírování se podle umístění souborů a implementovaných funkcí používají různé metody. Může nastat několik různých případů:

- zdroj i cíl jsou v rámci démona – použijí se přednostně metody `move` resp. `copy`,
- zdroj nebo cíl je lokální – použijí se přednostně metody `push` resp. `pull` nebo
- zdroj i cíl jsou v rámci různých démonů – použije se nouzové řešení[9].

Kvůli některým modulům byly metody `push` a `pull` přidány do GVfs později. Tyto metody nepracují proudově, ale zapíší celý soubor do lokálního souborového systému.

Nouzové řešení je implementováno pomocí proudových operací. Data jsou postupně čtena ze zdroje pomocí metody `read` a zapisována do cíle metodou `write`. Nouzové řešení se použije i v případě, kdy nejsou implementovány funkce `move`, `copy`, `push` nebo `pull`.

V démonech také chybí implementace různých metod pro čtení a zápis. Tyto metody lze v některých případech vytvořit, nicméně v některých je to možné pouze s využitím vyrovnávací paměti. K nejčastěji prováděným operacím patří kopírovaní mezi lokálním a vzdáleným souborovým systémem, bohužel u většiny démonů chybí implementace metod `push` a `pull`. Tyto metody by mohly velmi urychlit přenos a jejich implementaci nic nebrání. Obzvláště u `GVfsBackendGphoto2`, kde protokol umožňuje pouze přenos celých souborů, je zbytečné po malých částech kopírovat soubor z bufferu resp. do bufferu, když může být přečten resp. zapsán najednou. Vývojáři však nejsou nuceni tyto metody implementovat, protože jsou nahrazeny nouzovým řešením popsaným dříve.

Základní metody (`open_for_read`, `read`, `close_read`, `create`, `append`, `replace`, `write`, `close_write`) jsou implementovány téměř u všech démonů. V případě `GVfsBackendDav` chybí metoda `append`. Protokol `WebDAV` tuto operaci neimplementuje, existuje pouze koncept na rozšíření normy[10]. V GVfs by tuto metodu bylo možné implementovat přečtením původního souboru (např. i z vyrovnávací paměti) a následným zapsáním. V modulu `GVfsBackendObexftp` chybí veškeré základní metody pro zápis. Jejich implementace by byla možná pomocí vyrovnávací paměti.

U spousty modulů chybí metody `move` a především `copy`, protože ve většině případů není podpora v daných protokolech. Tyto metody jsou opět suplovány nouzovým řešením, pomocí základních metod, bylo by je však možné urychlit vyrovnávací pamětí.

Uživatelé si také často stěžují na chybějící podporu `seek_on_read`[11] či `seek_on_write`. To je ve většině případů zapříčiněno tím, že dané protokoly tuto možnost nepodporují, nebo je podporován jen dopředný posun. Metodu `open_for_read` lze u `GVfsBackendArchive` a `GVfsBackendFtp` doimplementovat zavřením a dopředným skokem (jako je to provedeno v `GVfsBackendSftp`). U `GVfsBackendObexftp` to lze provést ještě jednodušeji, protože je zde celý soubor ve vyrovnávací paměti. Operace `seek_on_write` a `truncate` by bylo možné provést s podporou vyrovnávací paměti.

Opět zde nejsou uvedeny některé metody (např. `open_icon_for_read` a `trash`), které však musíme mít v patrnosti.

[9] Zde se jedná o jiné nouzové řešení, než bylo na začátku sekce 3.4, a bude popsáno dále.
[10] `http://tools.ietf.org/html/draft-suma-append-patch-00`
[11] `https://bugzilla.gnome.org/show_bug.cgi?id=565986`

### 3.4.3 Vyrovnávací paměť v démonech

V systému GVfs není mezipaměť řešena centrálně, nicméně někteří démoni mají svoji vyrovnávací paměť v různých provedeních. Některé z nich budou popsány v následujících odstavcích.

**GVfsBackendAfc** Tento démon má jako jeden z mála implementované operace pro práci s ikonami. Ikony jsou ukládány do dočasného adresáře.

**GVfsBackendArchive** Během připojení je přečten celý archiv, ze kterého je vytvořen strom souborů obsahující `GFileInfo`. Navíc jsou zde zaznamenány originální cesty souborů v archivu. Tato vyrovnávací paměť se pak používá k různým kontrolám (např. zda požadovaný soubor existuje) a výsledkům operací (`enumerate` a `query_info`).

**GVfsBackendCdda** V `GVfsBackendCdda` jsou data z kompaktního disku načítána po sektorech. Vyrovnávací paměť obsahuje poslední sektor, ze kterého se při čtení kopírují data.

**GVfsBackendDav** Při zápisu je nejprve celý soubor zapsán do paměti a nahrán na server až při uzavření.

**GVfsBackendFtp** Démon si do vyrovnávací paměti ukládá výpisy adresářů a informace o souborech pro práci ve vláknech. U každého adresáře je udržován počet referencí, v případě nulového počtu je z vyrovnávací paměti smazán. Obsah mezipaměti je také invalidován při zápisových operacích. Používá se opět pro výstupy operací (`enumerate` a `query_info`) a různé kontroly.

**GVfsBackendGphoto2** V mezipaměti jsou obsahy adresářů a jejich `GFileInfo`. Položky se invalidují při zápisových operacích. Využití vyrovnávací paměti je stejné jako v případě `GVfsBackendArchive` a `GVfsBackendFtp`, navíc se používá i pro `query_fs_info`. Kromě mezipaměti metadat je zde i vyrovnávací paměť pro data – při čtení dojde nejprve ke stažení celého souboru, ze kterého se pak jen kopírují data. V případě zápisu se provádí totéž.

**GVfsBackendMtp** `GvfsBackendMtp` obsahuje vyrovnávací paměť mapující názvy souborů na ID souborů v zařízení.

**GVfsBackendObexftp** `GVfsBackendObexftp` si ukládá do vyrovnávací paměti pouze výpis posledního adresáře včetně informací o souboru. Data ve vyrovnávací paměti jsou po 3 vteřinách invalidována. Využití je opět obdobné jako v předchozích případech. Také zde dojde při čtení nejprve ke stažení celého souboru jako v případě `GVfsBackendGphoto2`.

### 3.4.4 Nástroje pro práci s GVfs

GVfs bylo vytvořeno primárně pro potřeby grafických správců souborů, jeho použití je však možné i z příkazové řádky. Spolu s GVfs jsou distribuovány pomocné nástroje pro manipulaci s virtuálními souborovými systémy. Tyto nástroje jsou vystavěny na aplikačním rozhraní GIO. Jedná se například o:

- `gvfs-mount` – připojování či odpojování souborových systémů,

- `gvfs-ls` – výpis obsahu adresáře,
- `gvfs-mkdir` – vytvoření nového adresáře,
- `gvfs-copy` – kopírování souboru či adresáře,
- `gvfs-move` – přesunutí souboru či adresáře,
- `gvfs-rename` – přejmenování souboru či adresáře,
- `gvfs-rm` – odstranění souboru či adresáře,
- `gvfs-cat` – tisk obsahu souboru na standardní výstup a
- `gvfs-save` – uložení dat ze standardního vstupu do souboru.

Chování většiny zmíněných příkazů odpovídá tradičním unixovým nástrojům s názvem bez prefixu `gvfs` s tím rozdílem, že se zadávaná cesta specifikuje pomocí URI (viz 3.2.2). Na rozdíl od komplikovaného připojení k FTP v GIO (viz algoritmus 3.1) je práce s těmito nástroji velmi jednoduchá, následuje ukázka připojení:

```
gvfs-mount ftp://example.com/
```

## 3.5 Překlad a spouštění pomocí JHBuild

*JHBuild*[12] je nástroj určený k usnadnění překladu zdrojových balíků resp. modulů [32]. Usnadňuje překlad celých softwarových kolekcí díky systému závislostí. Původně byl vytvořen pro potřeby GNOME, dnes je použitelný i s jinými projekty. JHBuild překládá moduly z různých zdrojů, včetně různých verzovacích systémů. Překlad se provádí do separátního stromu souborů a neovlivňuje tak zbytek systému, z tohoto důvodu se výborně hodí na vývoj aplikací a jejich testování.

Ukázka překladu GVfs přímo z GIT repositáře včetně spuštění vlastního hlavního démona GVfs je následující:

```
jhbuild buildone gvfs
jhbuild run /opt/gnome/libexec/gvfsd --replace
```

[12] `http://www.freedesktop.org/wiki/Software/jhbuild/`

# Kapitola 4

# Návrh vyrovnávacích pamětí

Tato kapitola se věnuje návrhu vyrovnávací paměti pro systém GVfs. Prezentovaný návrh se opírá o principy a technologie z předchozích kapitol. První sekce 4.1 se zabývá otázkou, kde by se daly využít vyrovnávací paměti v systému GVfs. Následující sekce 4.2 a 4.3 pak rozebírají některé z možných použití a obsahují návrh daných mezipamětí.

## 4.1 Možné využití vyrovnávacích pamětí

Odpověď na otázku, kde by bylo možné v systému GVfs využít vyrovnávací paměti, byla již nastíněna v sekci 3.4.2. Tato sekce rozebírá slabiny jednotlivých modulů a jejich interní mezipaměti.

Hlavní použití systému GVfs resp. GIO je v různých správcích souborů. Druhotně se tyto systémy používají v jiných aplikacích, obvykle po otevření z těchto správců. Budeme tedy vycházet z běžného způsobu používání souborových manažerů, což je typicky procházení adresářovou strukturou s cílem najít požadovaný soubor či soubory.

Z tohoto důvodu byla provedena analýza (viz příloha B), jakým způsobem souboroví manažeři tyto systémy využívají. Ze zmíněné analýzy je vidět, že jsou některé operace prováděny zbytečně vícekrát. Jedná se především o `query_info`, `query_filesystem_info` a `enumerate`. Navíc tyto tři operace pracují se souborovými metadaty `GFileInfo`. Vyrovnávací paměti pro metadata se věnuje následující sekce.

Systém GVfs poskytuje přístup na různá úložiště. U většiny těchto souborových systémů se předpokládá spíše práce s málo soubory či pouze jejich překopírování z nebo na lokální souborový systém. Na základě těchto předpokladů bude navržena i vyrovnávací paměť pro data souborů. Konstrukce této paměti bude odlišná, proto jí bude věnována samostatná sekce.

## 4.2 Vyrovnávací paměť souborových metadat

Nejvíce opakované operace při procházení souborovým systémem jsou `query_info` a výpis adresáře `enumeration` (viz příloha B). Výsledkem `query_info` je objekt `GFileInfo`, který obsahuje informace o souboru. Výsledkem `enumeration` jsou objekty `GFileInfo` všech potomků, z nichž je vytvořen `GFileEnumerator` na klientské straně. Získané objekty `GFileInfo` z těchto operací mohou být stejné při stejných parametrech. To je dobrým důvodem pro vytvoření společné stromové vyrovnávací paměti.

Avšak tyto parametry jsou často stejné jen u stejných operací, ale rozdílné napříč operacemi. Z toho důvodu by bylo lepší mít samostatné vyrovnávací paměti pro různé operace. Obě možnosti jsou diskutovány dále.

Získané objekty `GFileInfo` pro stejné soubory se mohou lišit ve dvou parametrech. První z nich jsou atributy definované objektem `GFileAttributeMatcher` (viz sekce 3.2.2), který je vytvořen na základě požadovaných atributů. Pokud tedy klient zažádá pouze o jméno souboru, v objektu `GFileInfo` obdrží pouze jméno souboru. Tento objekt samozřejmě nelze vrátit z mezipaměti na požadavek například o posledním přístupu.

Druhým parametrem jsou příznaky `GFileQueryInfoFlags`. Ty určují, zda se mají nebo nemají vyhodnocovat symbolické odkazy, resp. zda je výsledek pro symbolický odkaz nebo jeho cíl. Opět je jasné, že výsledky nejsou vždy zaměnitelné.

Již v menší míře někteří klienti také opakovaně provádějí `query_filesystem_info`. Výsledkem této operace jsou opět objekty `GFileInfo`, ty však obsahují odlišné informace o souborovém systému, na kterém je daný soubor umístěn.

### 4.2.1 Nároky na vyrovnávací paměť

Ze začátku je potřeba rozhodnout, zda by měla být vyrovnávací paměť pouze dočasná nebo persistentní mezi připojeními. Persistentní mezipaměť by však dávala smysl pouze u úložišť bez možnosti zápisu, která by byla využívána opakovaně. Takových je však v GVfs minimum, a proto má smysl pouze dočasná vyrovnávací paměť. V případě sdílených souborových systémů navíc bude nutné data po určené době invalidovat, protože zde může docházet k jejich změnám bez našeho vědomí. Čas, po kterém se data zneplatní, se bude lišit pro jednotlivá úložiště. V algoritmu 4.1 je naznačena implementace invalidace podle času. Položky obsahují časové známky, podle kterých lze určit platnost.

**Algoritmus 4.1** Invalidace podle času

```
 1 vlož (vyrovnávací_paměť, klíč, data)
 2 {
 3   ...
 4   položka->časová_známka = aktuální_čas;
 5   ...
 6 }
 7
 8 vyhledej (vyrovnávací_paměť, klíč)
 9 {
10   ...
11   doba = aktuální_čas - položka->časová_známka;
12   if (doba < vyrovnávací_paměť->časový_limit)
13   {
14     /* operace s položkou platnou podle času */
15   }
16   ...
17 }
```

Také bude vhodné limitovat velikost mezipaměti. Větší paměť sice dokáže obsloužit více požadavků, ale na druhou stranu zabírá více paměti a obsahuje více neplatných položek. Zdrojem neplatných položek je zejména invalidace podle času dle zmíněného algoritmu. V případě zaplnění paměti tedy bude vhodné využít například algoritmus LRU pro vyhledání oběti. Tvorba vyrovnávací paměti s LRU již byla naznačena v algoritmu 2.1.

Z výše zmíněných vlastností také vyplývá, že bude pro každého démona GVfs vhodná vlastní instance vyrovnávací paměti, aby bylo možné nastavit velikost a čas pro zneplatnění individuálně. Tím se také vyřeší potencionální problém s tím, že by nějaký modul –

například při výpisu stromu souborů – přepsal celou vyrovnávací paměť svými položkami a ostatní moduly by tak přišly o své položky.

### 4.2.2 Stromová struktura

Jednou z možností vytvoření vyrovnávací paměti je stromová struktura. Uzly stromu by byly adresáře a listy by reprezentovaly soubory. Díky této architektuře by v jedné mezipaměti mohly být uchovávány výsledky z operací `enumerate` i `query_info`. Výsledky z operace `query_info` by mohly aktualizovat informace z výpisů adresářů a naopak. To by ovšem působilo problémy, pokud by byly metody volány s různými parametry.

Samozřejmě by bylo možné podvrhnout parametry předávané démonům. Jednak požadovat všechny atributy, tedy vytvořit `GFileAttributeMatcher` s maskou „*“. Tato změna by ve většině démonů nezpůsobila žádnou dodatečnou činnost ani žádnou další komunikaci se serverem či zařízením. Ve vyrovnávací paměti by pak mohly být kompletní informace o souborech a nebylo by nutné kontrolovat tyto atributy. Bohužel v některých modulech dochází mimo jiné k určování MIME typu ze souborů. To je však výpočetně náročné, proto byla tato možnost zavržena.

Dalším nápadem je vždy vyžadovat informace o aktuálním souboru resp. nastavit `GFileQueryInfoFlags` na `G_FILE_QUERY_INFO_NOFOLLOW_SYMLINKS`. V případě symbolických odkazů poté dělat výpočet cíle v rámci vyrovnávací paměti. Symbolické odkazy nejsou GVfs moduly ve velké míře podporovány. Nicméně i tato změna se snaží dělat práci za jednotlivé moduly bez znalosti jejich logiky. V některých případech by to určitě přineslo značné urychlení, pro jiné by však mohlo mít opačný efekt.

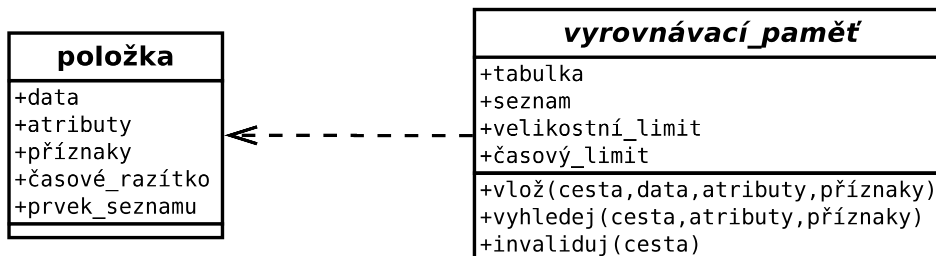
Udržování samotného stromu by také bylo výpočetně náročné. Ve stromu by navíc musela být spousta dodatečných uzlů pro adresáře, o kterých nemáme informace. V případě adresáře bychom museli také mít informaci, zda máme celý výpis nebo jen část. Tyto informace by se však navíc dynamicky měnily v případě invalidace podle času. Stromová struktura by tak měla smysl, pouze pokud bychom měli absolutní kontrolu nad souborovým systémem a metody by nebyly volány s rozdílnými parametry. To však nemůžeme zajistit, a proto bylo toto řešení zavrženo.

### 4.2.3 Samostatné vyrovnávací paměti

Jak již bylo řečeno, spojení výsledků z operací `enumeration` a `query_info` může přinášet různé problémy. Z analýzy v příloze B navíc vyplývá, že jsou často pouze stejné operace volány se stejnými parametry. Různé operace navzájem jsou však volány s různými parametry, takže by bylo lepší vytvořit samostatné vyrovnávací paměti pro výsledky z `enumeration` a pro `query_info`.

Samostatné mezipaměti mohou být mnohem jednodušší a s menší režií. Implementace může vycházet z vyrovnávací paměti s LRU z algoritmu 2.1. Absolutní cesta v démonu bude mapována na strukturu obsahující `GFileInfo` pro případ `query_info` nebo jejich seznam pro případ `enumeration`. V této struktuře bude navíc nutné uchovávat časové razítko a vstupní parametry `GFileAttributeMatcher` a `GFileQueryInfoFlags`. Návrh mezipaměti s těmito parametry je na obrázku 4.1.

Vyrovnávací paměť pro výpisy adresářů může obsahovat seznamy o různé velikosti. Pro limitování velikosti vyrovnávací paměti by bylo vhodné brát jako jednotky jednotlivé objekty `GFileInfo` a ne celé seznamy – to lze vyřešit implementací algoritmu Size Adjusted LRU. Možná implementace byla již nastíněna v sekci 2.4.2. Navrženou paměť bude tedy potřeba opatřit polem seznamů.



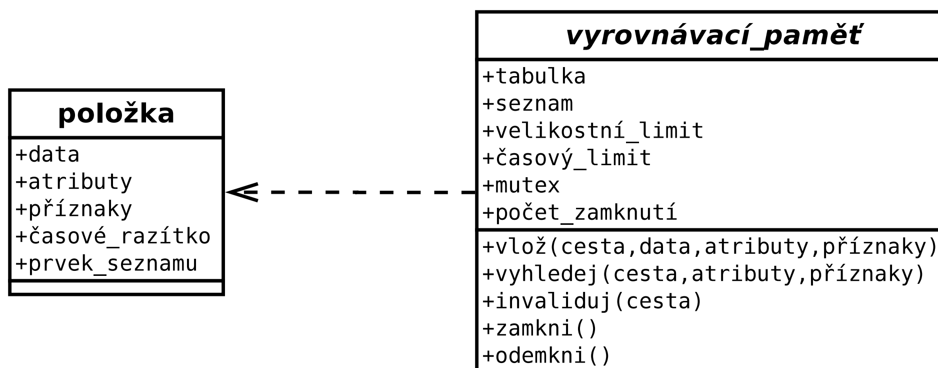

Obrázek 4.1: Diagram tříd navrhované vyrovnávací paměti

Provádění operací v démonech je paralelní. Struktura mezipaměti proto bude muset být dále rozšířena o zámek (*Mutex*), aby byl zajištěn výlučný přístup k interním strukturám. Navíc data vyrovnávacích pamětí budou muset být při zápisových operacích invalidována a během těchto operací musí být paměti odstaveny. K těmto účelům bude potřeba ještě vytvořit počítadlo udávající počet aktuálně probíhajících zápisových operací a operace pro jeho nastavení. Implementace těchto ochran tedy může vypadat jako v algoritmu 4.2.

**Algoritmus 4.2** Ochrany při vícenásobném přístupu

```
 1  vlož (vyrovnávací_paměť, klíč, data)
 2  {
 3    if (vyrovnávací_paměť->počet_zamknutí > 0)
 4      return;
 5    ...
 6    lock (mutex);
 7    /* operace s interními strukturami */
 8    unlock (mutex);
 9    ...
10  }
11
12  zamkni (vyrovnávací_paměť)
13  {
14    lock (mutex);
15    vyrovnávací_paměť->počet_zamknutí++;
16    unlock (mutex);
17  }
18
19  odemkni (vyrovnávací_paměť)
20  {
21    lock (mutex);
22    vyrovnávací_paměť->počet_zamknutí--;
23    unlock (mutex);
24  }
```

Výsledný diagram, který znázorňuje navržené vyrovnávací paměti, je na obrázku 4.2. Paměť pro výpis adresářů bude obsahovat pouze pole seznamů LRU. Aplikační rozhraní obou pamětí bude stejné, vnitřně se však budou lišit kvůli rozdílné implementaci algoritmu LRU, která je jejich základem. Je také nutné zajistit, aby toto aplikační rozhraní bylo dostupné pro démony, aby mohli přistupovat k těmto datům a využívat je ke kontrolám apod. Takto navržená paměť do budoucna umožní nahrazení různých interních vyrovnávacích pamětí z modulů a urychlení operací nejen pro práci s metadaty.



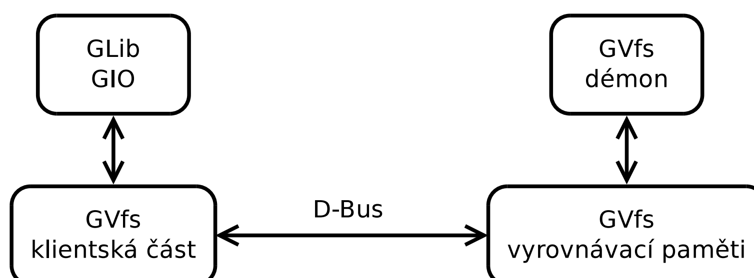

Obrázek 4.2: Výsledný diagram tříd navrhované vyrovnávací paměti

### 4.2.4 Integrace vyrovnávacích pamětí

V předchozí sekci byly diskutovány a navrženy dvě vyrovnávací paměti. Návrhy se týkaly samotných struktur, ne však jejich integrace do systému GVfs. Umístění vyrovnávacích pamětí by mělo určitě být v serverové části GVfs, aby nedošlo ke zvětšení klientských knihoven načítaných do různých aplikací, a také aby démoni mohli snadno využívat zmíněné vyrovnávací paměti.

Mělo by se také jednat o transparentní řešení, které nebude vyžadovat větší zásahy do kódu jednotlivých modulů. Mezipaměti při připojení démona mohou tedy být ve třídách `GVfsBackend` pouze inicializovány a vhodně nastaveny. Při odpojení démona budou vyrovnávací paměti opět uvolněny.

Pro samotnou aplikaci se jako vhodné místo jeví objekty `GVfsJob`, které obalují metody démonů. Při přijetí požadavku z D-Bus sběrnice je vytvořen příslušný potomek objektu `GVfsJob`, který je poté vykonán voláním metod konkrétního démona. V případě, že data budou ve vyrovnávací paměti, lze je vrátit ihned a nemusí se volat metody démonů. Schématicky je umístění vyrovnávacích pamětí z pohledu klienta naznačeno na obrázku 4.3.




Obrázek 4.3: Umístění vyrovnávacích pamětí z pohledu klienta

Pro integraci pamětí bude potřebné upravit některé z potomků `GVfsJob` tak, aby byla zajištěna validita dat. Během operací s démonem, které mohou měnit metadata, musí být tyto mezipaměti zamčeny a jejich data správně invalidována. Požadované úpravy jsou následující:

- `delete`, `make_directory`, `make_symlink` – zamčení během operace a invalidace souboru,
- `move` – zamčení během operace a invalidace zdroje i cíle,
- `pull` – zamčení během operace a invalidace zdroje (pokud má být zdroj smazán),

- `push` – zamčení během operace a invalidace cíle,
- `set_attribute` – zamčení během operace a invalidace souboru,
- `set_display_name`, `trash` – zamčení během operace a invalidace všech souborů s prefixem,
- `open_for_write` – pouze zamčení a invalidace souboru a
- `close_write` – pouze odemčení.

Operace `set_display_name` a `trash` mohou pracovat s adresáři, a proto je nezbytné z vyrovnávací paměti smazat všechny položky, které jsou potomky daného adresáře. Musí tedy být zpracovány všechny položky ve vyrovnávací paměti a smazány ty s prefixem adresáře. Toto je jedna z nevýhod návrhu, protože jsou v něm mapovány absolutní cesty.

## 4.3 Vyrovnávací paměť obsahu souborů

Operace pro práci se soubory jsou v GIO rozděleny na práci s metadaty a na práci s daty, proto bude v této sekci navržena samostatně vyrovnávací paměť pro obsah souborů. Při práci s daty připadají v úvahu dvě hlavní oblasti využití vyrovnávací paměti. První z nich je urychlení práce s virtuálními souborovými systémy, kterého lze dosáhnout ukládáním často používaných souborů do mezipaměti. Druhým důvodem využití je pak emulace chybějících proudových operací, pokud je samotné protokoly či knihovny nepodporují.

### 4.3.1 Obecná vyrovnávací paměť

Zkusme se zamyslet nad vyrovnávací pamětí, která by umožňovala zrychlení GVfs i emulaci proudových operací. Soubory by bylo dobré ukládat do mezipaměti po blocích a udržovat jen poslední použité bloky, protože jsme omezeni volným místem v paměti či na disku. Avšak démoni, kteří nepodporují proudové operace, mají metody pouze pro přenos celého souboru. Z toho důvodu nelze načíst pouze chybějící bloky. Proto by bylo pro emulaci proudových operací zapotřebí, aby ve vyrovnávací paměti byly celé soubory. Celé soubory však mohou zabírat mnoho prostoru, vyrovnávací paměť by tedy musela být omezena jen na velmi málo souborů.

Operace v GVfs mohou běžet paralelně, v jeden okamžik tedy může být čten stejný soubor různými klienty, i když to není běžné. Soubor se však během čtení může ve vzdáleném úložišti změnit. Nová čtecí operace by měla číst jeho novější verzi, zatímco ta stará by měla číst původní data. Vyrovnávací paměť by se dokonce měla vyrovnat i s případem, kdy je soubor čten i zapisován současně. Tyto situace jsou v unixových systémech vyřešeny použitím i-uzlů. Toho by se dalo využít tak, že by mezipaměť mohla mít soubory uloženy na disku a volat na ně systémové příkazy při souborových operacích.

Musí být také zajištěno, že nebudou zůstávat již nepotřené soubory v dočasných adresářích. To se může stát v případě pádu démona, nebo pokud nebude virtuální souborový systém správně odpojen. Bohužel ani jeden z případů není výjimkou. Proto se na otevřené dočasné soubory běžně volá příkaz `unlink`, aby došlo k jejich smazání po ukončení programu. Po zavolání `unlink` bohužel již neexistuje způsob, jak soubor otevřít při dalším požadavku. Soubory tedy nemůžeme smazat dřív, než mají být vymazány z mezipaměti. Možným řešením je udržovat persistentní žurnál otevřených souborů. Ten by mohl být spravován hlavním démonem, který soubory bude odmazávat například při pádu démona.

Jinou možností je vlastní implementace i-uzlů. Soubor by mohl být vyrovnávací pamětí otevřen pro čtení a smazán příkazem `unlink`. Vyrovnávací paměť by musela být schopna

uchovávat více verzí souborů a jednotlivým klientům předávat správné verze těchto souborů. Při požadavcích na čtení by se vždy přečetla požadovaná část a předala klientovi.

Toto i předchozí řešení jsou však velmi komplexní a složité. Ve většině případů by tak docházelo ke zpomalování, zvláště pak při kopírování celých adresářů apod. Z těchto důvodů bylo toto řešení zavrženo a urychlení čtecích operací bylo přenecháno již implementovanému přednačítání bloků v `GVfsChannel`.

### 4.3.2 Emulace základních operací

Vytvoření mezipaměti pouze pro účely emulace operací by umožnilo eliminovat některé z požadavků na obecnou vyrovnávací paměť. Výsledkem by pak bylo pokrytí chybějící funkcionality a možné zjednodušení některých démonů (viz sekce 3.4.2).

Návrh je až triviálně jednoduchý. Při operaci `open_for_read` se zavolá operace `pull`, která stáhne soubor do dočasného adresáře, otevře ho pro čtení a provede `unlink`. Otevřený datový tok může být přenášený mezi operacemi stejným způsobem jako v normálním případě, tedy s pomocí `GVfsChannel`. Následující požadavky `read` či `seek_on_read` se budou provádět na staženém souboru. Při `close_read` bude soubor smazán. V případě pádu programu dojde k automatickému smazání. Obdobné řešení lze použít i pro zápis souborů pouze s tím rozdílem, že je soubor v rámci operace `close_write` nahrán do úložiště.

Této vyrovnávací paměti by se dalo využít i na žádost. Například pokud chybí implementace operace `seek_on_read`, soubor může být uložen do mezipaměti až ve chvíli jeho použití. Pro tento případ bude však nutné před stažením souboru zkontrolovat, zda nebyl změněn. Tato kontrola může být provedena pomocí času poslední modifikace, který musí být před začátkem operace zjištěn zavoláním `query_info`. Z pohledu klienta se však toto chování může jevit jako zatuhnutí, jeho implementace je tedy na zvážení.

Vhodné místo pro integraci této vyrovnávací paměti je opět v objektech `GVfsJob` jako v případě mezipamětí pro metadata. Žádná invalidace ani zámky nejsou potřebné, protože dočasný soubor bude patřit vždy jen jednomu klientovi.

# Kapitola 5

# Implementace vyrovnávacích pamětí

Tato kapitola se věnuje implementaci vyrovnávacích pamětí, které vycházejí z návrhu v kapitole 4. Implementace však nebyla úplně přímočará, proto jsou dále popsány některé implementační detaily, které bylo potřeba při vývoji mezipamětí vyřešit.

První sekce 5.1 popisuje detaily při tvorbě vyrovnávacích pamětí pro metadata a výpisy adresářů. Další sekce 5.2 pak rozebírá implementaci mezipaměti pro emulaci proudových operací. Následně je v sekci 5.3 popsána tvorba dokumentace a v poslední sekci 5.4 integrace implementovaných pamětí do modulů.

## 5.1 Vyrovnávací paměti pro metadata a výpisy adresářů

Podle návrhu v sekci 4.2 byly implementovány dvě vyrovnávací paměti pro metadata a výpisy adresářů. Mezipaměti byly vytvořeny jako dvě samostatné struktury, nicméně obě dvě uchovávají objekty `GFileInfo`. Práce byla napsána v jazyce C, ve kterém je vytvořen celý systém GVfs. K tvorbě vyrovnávacích pamětí byly využity datové struktury z knihovny GLib. Z nich je zásadní zejména tabulka s rozptýlenými hodnotami `GHashTable`, která je základem pamětí. Pro algoritmus LRU a správu paměti byla využita fronta `GQueue`. Popis těchto struktur je v sekci 3.1.

Vstupem mapovací funkce je absolutní cesta k souboru v úložišti. Výstupem je pak objekt `GFileInfo` resp. seznam `GList` těchto objektů, pokud jsou v mezipaměti odpovídající data. Důležitou roli zde hrají parametry – příznaky `GFileQueryInfoFlags` a seznam atributů `GFileAttributeMatcher`.

Paměti musí být explicitně povoleny v konkrétních démonech. Součástí inicializace je nastavení velikosti a času pro invalidaci. Obě vyrovnávací paměti jsou aplikovány v objektech `GVfsJob`. V `GVfsJobQueryInfo` a `GVfsJobEnumerate` se využívají pro urychlení. Při ostatních operacích, kdy může docházet ke změně metadat, jsou vyrovnávací paměti patřičně invalidovány a během těchto operací zamčeny.

Do budoucna by samotná struktura vyrovnávací paměti pro `query_info` mohla být použita bez úprav i pro urychlení operace `query_filesystem_info`. Musela by být pouze patřičným způsobem integrována do objektů `GVfsJob` a v démonech vytvořena její nová instance.

Výsledná aplikační rozhraní pamětí pro metadata jsou na obrázcích 5.1 a 5.2. Podrobný popis parametrů a typů je součástí zdrojových kódu. Rozhraní poskytují větší funkcionalitu, než byla původně v návrhu. Pouze v případě mezipaměti pro výpis adresářů bylo potřeba provést drobnou změnu.

Metoda `enumeration` pro velké adresáře může trvat dlouhou dobu a během této doby může dojít k zneplatnění některých informací o potomcích. Proto pro mezipaměť pro výpis adresářů bylo zapotřebí rozdělit operaci vkládání na dvě části. Na začátku operace je do vyrovnávací paměti vložena položka, která neobsahuje seznam objektů `GFileInfo`. Tento seznam je vložen až ve chvíli, kdy je kompletní. Pokud však byla položka mezitím invalidována, není již možné daný seznam vložit. V případě více souběžných operací je vkládání ošetřeno časovou známkou.

***GVfsInfoCache***

```
+new(max_count,max_time)
+free()
+get_count()
+insert(path,info,matcher,flags)
+find(path,matcher,flags)
+invalidate(path,maybe_dir)
+remove(path)
+remove_all()
+disable()
+enable()
+is_disabled()
```

Obrázek 5.1: Výsledné rozhraní mezipaměti pro `query_info`

***GVfsEnumerationCache***

```
+new(max_count,max_time)
+free()
+get_count()
+insert(path)
+set(path,infos,matcher,flags,stamp,count)
+find(path,matcher,flags)
+invalidate(path,maybe_dir)
+remove(path)
+remove_all()
+disable()
+enable()
+is_disabled()
```

Obrázek 5.2: Výsledné rozhraní mezipaměti pro `enumeration`

### 5.1.1 Problematická invalidace dat

Z hlediska invalidace dat můžeme úložiště rozdělit na tři kategorie:

- s výlučným přístupem bez podpory symbolických odkazů (např. CDDA),
- s výlučným přístupem s podporou symbolických odkazů (např. AFC) nebo
- sdílené (např. FTP).

První kategorie je z hlediska invalidace samozřejmě nejméně problematická a není problém zajistit, aby ve vyrovnávací paměti byla jen validní data. Není zde potřeba ani zneplatnění podle času.

Podpora symbolických odkazů v druhé kategorii již přináší problémy. Zejména v případě, že se změní metadata cíle. V cíli symbolického odkazu není žádná informace o symbolických odkazech odkazujících se na tento soubor, proto ve vyrovnávací paměti může

zůstat neplatná informace. Toto musí být řešeno přísnou časovou invalidací. Stejně tomu bude i u třetí kategorie – sdílených souborových systémů. Zde dochází k úpravám třetí stranou, o kterých nejsme žádným způsobem informováni. Tento případ se také musí řešit invalidací podle času.

### 5.1.2 Kontrola podmnožin parametrů

Při vyhledávání dat ve vyrovnávacích pamětích musíme dbát na parametry. Není však nezbytně nutné, aby byly naprosto stejné. Například nám stačí znalost, že se nejedná o symbolický odkaz, pokud se neshodují příznaky `G_FILE_QUERY_INFO_FLAGS`. Tuto informaci lze z objektu `GFileInfo` často vyčíst.

Také atributy `GFileAttributesMatcher` nemusí být totožné, musí se však jednat o jejich podmnožinu. Zda se jedná o podmnožinu lze snadno zjistit odečtením těchto objektů operací `g_file_attribute_matcher_subtract`. V algoritmu 5.1 je naznačen pseudokód takové kontroly.

**Algoritmus 5.1** Validita dat podle parametrů

```
1  je_položka_validní_pro_parametry (položka, příznaky, atributy)
2  {
3    if (položka->příznaky != příznaky)
4    {
5      if (položka->příznaky == G_FILE_QUERY_INFO_NONE ||
6          !známe_typ_souboru (položka->data) ||
7          je_symbolický_odkaz (položka->data))
8        return FALSE;
9    }
10
11   if (!je_podmnožinou (položka->atributy, atributy))
12     return FALSE;
13
14   return TRUE;
15 }
```

### 5.1.3 Správa neplatných položek

Ve vyrovnávacích pamětích může být spousta nevalidních položek, což je způsobeno invalidací podle času. Problém se projeví v případě zaplnění paměti. Algoritmus LRU může vybírat oběti, které jsou ještě validní ale dlouho nepoužívané, zatímco v mezipaměti jsou položky již neplatné ale předtím často používané.

Naimplementované paměti byly proto opatřeny ještě jedním seznamem. Tento seznam udržuje chronologické pořadí podle toho, jak byly položky vkládány. Při přístupu k vyrovnávací paměti dojde po určeném časovém intervalu ke smazání všech neplatných položek podle času. Časový interval pro smazání je polovinou zvoleného intervalu invalidace podle času a byl určen experimentálně. Touto správou paměti se šetří paměťové nároky, urychluje vyhledávání a navíc vyrovnávací paměť obsahuje více platných položek.

### 5.1.4 Sdílení dat vyrovnávacích pamětí

Z analýzy v příloze B plyne, že je nejdříve prováděn výpis adresáře, a pak až následují dotazy na informace o obsažených souborech. Z toho důvodu byly výsledky operace

`enumerate` použity i do vyrovnávací paměti pro `query_info`. Již následující dotaz na informace o souboru může být obsloužen vyrovnávací pamětí, pokud se shodují parametry. Navíc v paměti existuje pouze jedna instance `GFileInfo` a dojde jen k navýšení počtu referencí na objekt.

### 5.1.5 Mezipaměti pouze pro čtení

Dalším implementovaným vylepšením je, že vyrovnávací paměti nejsou po dobu zápisových operací zcela blokovány. Na začátku operací, které mění metadata, je jen omezen zápis do mezipamětí a patřičné položky jsou invalidovány. Pokud je například zapisován soubor, dojde k jeho invalidaci v mezipamětích, ale dotazy na jiné soubory mohou být stále obslouženy. Do budoucna by bylo možné blokovat jen konkrétní soubory a zbytek vyrovnávací paměti by mohl být i zapisován.

### 5.1.6 Vylepšení invalidace adresářů

Při některých operacích musíme počítat s tím, že pracujeme s adresářem. Jedná se o metody `set_display_name` a `trash`. Zda pracujeme s adresářem není z parametrů operace známé, a je tedy nezbytné vymazat všechny položky s daným prefixem z vyrovnávacích pamětí pro zajištění validity dat. Tomu je někdy možné zabránit a invalidaci zjednodušit, pokud máme uložený typ souboru ve vyrovnávací paměti. Nově implementovaná mezipaměť se nejdříve pokusí zjistit typ souboru a až poté data zneplatní.

### 5.1.7 Příznak podpory symbolických odkazů

Jiným vylepšením do budoucna by bylo nastavit při inicializaci vyrovnávací paměti příznak, zda ve virtuálním souborovém systému mohou být symbolické odkazy. Příznaky `GFileQueryInfoFlags` by bylo možné kompletně ignorovat, pokud chybí jejich podpora. Potom by i invalidace dat nemusela být v některých modulech tak přísná.

## 5.2 Vyrovnávací paměť pro emulaci proudových operací

Také vyrovnávací paměť pro emulaci proudových operací byla implementována podle návrhu v sekci 4.3. Mezipaměť je vytvořena jako samostatná struktura a musí být explicitně povolena v konkrétních modulech. Mezipaměť je integrována v objektech `GVfsJob`. Pokud v démonu chybí implementace proudových operací, soubor je stažen pomocí operace `pull` a využíván k emulaci. Implementována byla pouze emulace čtecích operací z důvodů vysvětlených v sekci 5.2.1. Použití této vyrovnávací paměti na žádost nebylo dokončeno z důvodů zmíněných v 5.2.2.

Výsledné aplikační rozhraní paměti je naznačeno na obrázku 5.3. Jednotlivé metody odpovídají operacím v GVfs. Podrobnější popis je opět součástí zdrojových kódů.

### 5.2.1 Problém možné ztráty dat

Provádění vstupně-výstupních operací v rámci metod `open` a `close` může způsobovat problémy. Tyto operace kvůli mezipaměti mohou běžet nezvykle dlouho a může se zdát, že se nic neděje. To je však problém zejména u zápisu.

Klienti často nekontrolují návratové hodnoty z operace `close`. V případě unixového `close` to není tak závažné, i když před tím manuálové stránky důrazně varují. Pokud při emulaci dochází k vlastnímu přenosu dat souboru až v `close`, mohou se zde vyskytnout

**GVfsFileCache**

```
+new()
+free()
+open_for_read(job)
+read(job)
+seek_read(job)
+close_read(job)
```

Obrázek 5.3: Výsledné rozhraní mezipaměti pro data

nejrůznější chyby, které běžně uživatel zjistí již při operaci `open_for_write`. Klient při ignorování výstupní hodnoty může předpokládat, že byla data úspěšně zapsána, protože operace `write` proběhla bez chyby. To je však zásadní problém, který může způsobit ztrátu dat.

Ke ztrátě dat také může dojít, pokud uživatel nepočká na dokončení operace `close`. Informace o průběhu přenosu souboru přitom mohou hlásit 100%, to však znamená pouze úspěšný zápis do dočasného souboru. V případě následného přerušení spojení (např. vytažením zařízení z USB) může klient přijít o zapsaná data. Z tohoto důvodu nebyla podpora pro emulaci zápisových operací implementována.

### 5.2.2 Chybějící informace o průběhu přenosu

Jak již bylo naznačeno, ve specifických případech mohou chybět informace o přenosu resp. mohou být zavádějící, například když se kopírují soubory mezi dvěma různými virtuálními souborovými systémy. Standardně se to provádí čtením po částech z jednoho souborového systému a současným zápisem do jiného. Informace o přenosu se předávají po přenášených blocích. Pokud se však soubor do lokálního úložiště načte celý při operaci `open_for_read`, vypadá to, že se dlouhou dobu nic neděje. Naopak následné kopírování do cílové destinace již může být velmi rychlé, protože jde o kopírování z dočasného souboru. S tím bohužel nejde nic dělat, protože se nepředpokládá nějaká vstupně-výstupní aktivita během operace `open`.

V případě použití vyrovnávací paměti na žádost je situace ještě více matoucí. Pomyslné zaseknutí se zde objeví až po nějaké době, po kterou byl přenos bezproblémový. Použití mezipaměti na žádost se váže s posunem v datovém toku, ten by proto měl být raději implementován znovuotevřením daného souboru.

Možným vylepšením do budoucna by bylo provádění metody `pull` na pozadí a souběžným čtením. Tato operace by musela vracet informace o skutečném průběhu přenosu. Aktuálně však tyto informace ne vždy odpovídají tomu, co je v souboru skutečně zapsáno. Data mohou být například ještě v interních vyrovnávacích pamětích. To záleží na konkrétních použitých knihovnách, na kterých jsou démoni vystavěni.

### 5.2.3 Vytvoření GVfsJob bez D-Bus

Pro stažení souboru v rámci operace `open_for_read` je nezbytné vytvoření nového objektu `GVfsJobPull`. Objekty `GVfsJob` slouží v GVfs jako reprezentace požadavků ze sběrnice D-Bus. Přijetím požadavku dochází automaticky k jejich vytvoření a zanikají s odesláním odpovědi. Problém ovšem nastává, když v `GVfsJobOpen` chceme vytvořit instanci `GVfsJobPull`. Vyslání dalšího požadavku na D-Bus by bylo zbytečně komplikované. Jako řešení musel být tedy připsán nový konstruktor a ošetřeny některé metody v `GVfsJobDBus`, aby po dokončení úlohy neposílaly výsledek opět na komunikační sběrnici.

## 5.3 Tvorba dokumentace

Vzniklou práci bylo potřeba zdokumentovat. Na výběr je spousta nástrojů pro generování dokumentace s různou funkcionalitou. Nakonec byl zvolen méně známý *GTK-Doc*[1], protože je běžně používán pro veřejná aplikační rozhraní knihoven a aplikací GNOME a GTK+ [33]. Navíc některé části GVfs jsou již okomentovány syntaxí tohoto nástroje.

GTK-Doc je však navržen jako univerzální nástroj pro tvorbu dokumentace, a proto jsou jeho nastavení a integrace do projektu komplikované. Na druhou stranu umí na rozdíl od jiných produktů zacházet např. se systémem GObject. V GVfs není zakomponován, protože se jedná o interní nedokumentované programové rozhraní. K tomuto projektu tak nebyla vygenerována dokumentace, kód byl pouze okomentován syntaxí GTK-Doc.

## 5.4 Integrace vyrovnávacích pamětí

Modul `GVfsBackendGphoto2` byl zvolen pro demonstraci naimplementovaných vyrovnávacích pamětí. Jedná se o vhodného kandidáta, protože knihovna *libgphoto2*[2], na které je vystavěn, nepodporuje proudové operace. Tento protokol je využíván zejména v mobilních telefonech či fotoaparátech. `GVfsBackendGphoto2` již obsahuje interní paměti jak pro metadata, tak pro emulaci základních operací. Díky tomu je také možné demonstrovat zjednodušení jednotlivých modulů.

Pro modul `GVfsBackendGphoto2` byly povoleny všechny tři vyrovnávací paměti. Velikost mezipamětí pro metadata byla experimentálně nastavena na tisíc položek a invalidace podle času je vypnutá. Kód byl patřičně zjednodušen.

Nakonec byly vyrovnávací paměti pro metadata a výpisy adresářů povoleny z důvodů výkonnostního testování i v `GVfsBackendSftp`. Zde byly velikosti opět experimentálně nastaveny na tisíc položek a invalidace dat po třech vteřinách, protože na SFTP nemůžeme vyloučit úpravy třetí stranou.

[1] `http://www.gtk.org/gtk-doc/`
[2] `http://www.gphoto.org/proj/libgphoto2/`

# Kapitola 6

# Testování vyrovnávacích pamětí

Všechny naimplementované vyrovnávací paměti bylo nezbytné důkladně otestovat. Následuje sekce 6.1, která popisuje funkcionální testování a testovací rozhraní GLib. Testování je zaměřeno především na správnou invalidaci dat. Další sekce 6.2 porovnává výkonnost vyrovnávacích pamětí.

## 6.1 Funkcionální testování

V rámci funkcionálního testování byly vytvořeny testy pro naimplementované mezipaměti. Ověření správné funkcionality bylo provedeno pomocí jednotkových a integračních testů. Všechny testy využívají testovacího rozhraní GLib. Testy byly napsány v jazyce C v souladu se systémem GVfs.

### 6.1.1 Testovací rozhraní GLib

K usnadnění testování bylo využito testovací rozhraní poskytované knihovnou GLib. Toto rozhraní umožňuje psaní a údržbu jednotkových testů v závislosti na testovaném zdrojovém kódu [34]. Jeho aplikační rozhraní vychází ze zažitých konceptů použitých v jiných testovacích nástrojích. Rozhraní umožňuje vytvářet z jednotlivých testů celé testovací případy. Stará se o jejich spouštění a detailní ladící výstupy.

Pro testy poskytuje různé kontrolní funkce, které v případě neúspěchu ukončí provádění celé skupiny testů. Tyto kontroly jsou realizovány s využitím různých variant klasického příkazu `assert`. Tyto varianty přináší výhody v podrobných chybových výpisech, které obsahují porovnávané hodnoty a urychlují tak opravování chyb. Často používané jsou zejména:

- `g_assert_cmpint` – pro porovnávání celočíselných hodnot,
- `g_assert_cmpstr` – pro kontrolu řetězců a
- `g_assert_error` – pro srovnání chybových hlášení.

### 6.1.2 Jednotkové testy

Nejprve byly implementovány jednotkové testy pro vyrovnávací paměti pro metadata a výpis adresářů. Hlavním úkolem těchto testů je prověření funkcionality jednotlivých metod mezipamětí. Následuje seznam toho, co je jednotkovými testy kontrolováno u mezipaměti pro `GFileInfo`:

1. metody `new` a `free`,
2. metody `insert` a `remove`,
3. metoda `find` obecně,
4. metoda `find` z hlediska atributů,
5. metoda `find` z hlediska příznaků,
6. invalidace podle času,
7. invalidace podle velikosti a
8. metody `enable` a `disable`.

Ukázka spuštění testovacích sad je následující:

```
# jednotkové testy pro paměť pro query_info
test/test-info-cache --verbose

# jednotkové testy pro paměť pro enumerate
test/test-enumeration-cache --verbose
```

Sada testů pro vyrovnávací paměť pro výpis adresářů je téměř totožná, proto zde není více rozebrána. Jednotkové testy pro mezipaměť pro emulaci proudových operací nebyly vytvořeny, protože by otestování jednotlivých funkcí bylo příliš komplikované a nad časový rámec této diplomové práce.

### 6.1.3 Integrační testy

Další fází testování jsou integrační testy. Ty byly implementovány pro všechny tři typy vyrovnávacích pamětí. Smyslem těchto testů je ověřit správnou funkcionalitu celého řešení a integraci do systému. Proto nejsou struktury pamětí testovány přímo jako v předchozím případě, ale je využito veřejné aplikační rozhraní GIO.

Testy pro mezipaměti pro metadata a výpis adresářů jsou opět velmi podobné a jsou zaměřeny především na invalidaci dat. Konkrétně se u těchto pamětí testuje:

1. metoda `set_attribute`,
2. metoda `delete`,
3. metoda `trash`,
4. metoda `set_display_name`,
5. metoda `move`,
6. metoda `pull` pomocí `g_file_copy`,
7. metoda `push` pomocí `g_file_copy` a
8. metoda `copy`.

Následuje ukázka spuštění těchto testovacích sad:

```
# spuštění démona z upraveného GVfs
GVFS_CACHE=1 /opt/gnome/libexec/gvfsd --replace &
gvfs-mount gphoto2://[usb:001,010]/

# integrační testy pro paměť pro query_info
test/test-info-cache --verbose /tmp/ gphoto2://[usb:001,010]/

# integrační testy pro paměť pro enumerate
test/test-enumeration-cache --verbose /tmp/ gphoto2://[usb:001,010]/
```

Pomocí parametrů jsou předány cesty k testovacím adresářům, jeden na lokálním souborovém systému a druhý na připojeném virtuálním systému s povolenou vyrovnávací pamětí.

## 6.2 Výkonnostní testování

Výkonnostní testování bylo provedeno pouze pro mezipaměti pro metadata a výpis adresářů, které byly vytvořeny s cílem zrychlit provádění operací. Vyrovnávací paměť pro data měla pouze rozšířit chybějící funkcionalitu, a proto nebyla výkonnostně testována. Naměřené časy při emulaci by navíc nebyly s čím porovnávat.

### 6.2.1 Výkonnostní testování PTP

Jako vhodný modul pro testování se jevil `GVfsBackendGphoto2`. Na rozdíl od jiných již obsahoval různé interní vyrovnávací paměti, což se zdálo být zárukou toho, že je tento protokol pomalý. Implementované mezipaměti byly proto do něj demonstračně integrovány (viz sekce 5.4).

Testování probíhalo na stejném počítači, který nebyl během testování zatížen jinými činnostmi. Jako zařízení byl použit chytrý mobilní telefon s podporou PTP protokolu. Připojený mobilní telefon byl také využíván pouze probíhajícími testy.

Pro validitu testů bylo potřeba zajistit, aby data nezůstávala v různých vyrovnávacích pamětech mimo GVfs. Modul `GVfsBackendGphoto2` je vystavěn na knihovně libgphoto2. Tato knihovna již sama o sobě obsahuje vyrovnávací paměť na bázi LRU. Do mezipaměti se ukládá obsah souboru i jeho metadata. Standardně je velikost této paměti nastavena na dva soubory a je určena makrem `PICTURES_TO_KEEP` v souboru `gphoto2-filesys.c`. Pro testování byla tato knihovna přeložena bez podpory této vyrovnávací paměti. Po vyřazení mezipaměti z knihovny libgphoto2 se již naměřené časy pro první a opakovaný přístup významně neliší.

Bylo však zjištěno, že komunikace po protokolu PTP není tak pomalá, jak by se mohlo zdát. Samotná komunikace mezi GVfs modulem a zařízením byla ve stovkách mikrosekund, zatímco komunikace mezi aplikací a GVfs byla v jednotkách milisekund. Byl zde sice náznak zrychlení po použití vyrovnávacích pamětí, avšak měření bylo zatíženo velikou směrodatnou odchylkou a jednotlivá měření se významně lišila. Z tohoto důvodu `GVfsBackendGphoto2` nebyl nakonec použit k výkonnostnímu testování.

### 6.2.2 Výkonnostní testování SFTP

Pro testování byl nakonec vybrán `GVfsBackendSftp`. Síťová komunikace asi nenabízí takovou stabilitu jako například komunikace po USB, avšak latence sítě je běžně v desítkách

milisekund. Latence je tedy o řád lepší než samotná režie GIO resp. GVfs, a proto bude použití vyrovnávacích pamětí markantnější.

### Rychlost jednotlivých operací

Nejprve byly provedeny testy pro srovnání jednotlivých operací (`query_info` a `enumerate`). Testy jsou jednoduché programy v jazyce C využívající aplikační rozhraní GIO. Porovnává se doba prvního a druhého provedení operace s i bez vyrovnávací paměti. Po operacích byly požadovány všechny atributy, tedy `GFileAttributeMatcher` vytvořený s parametrem „`*`“. Uváděné časy vznikly jako aritmetický průměr z 25 iterací. Tento relativně malý počet byl zvolen, aby bylo možné provést všechny testy při podobné zátěži sítě a serveru a s co nejmenší chybou. Některé testy však byly z důvodu vysoké směrodatné odchylky (a tedy možné statistické chyby) opakovány vícekrát. V důsledku bylo provedeno přibližně dvojnásobné množství testů.

Pro zjištění doby prvního požadavku je nezbytné virtuální souborový systém mezi testy odpojovat a připojovat. Před novým připojením bylo nutné proces na dvě vteřiny uspat, aby byla operace odpojení skutečně dokončena a připojení proběhlo bez chyby[1]. Operace připojení a odpojení také nejsou nejrychlejší a proto jedna iterace trvá několik vteřin i přes to, že délka provádění požadavků je zanedbatelná.

Výsledky pro operaci `query_info` shrnuje tabulka 6.1. Tabulka zachycuje trvání operace a směrodatnou odchylku. Z výsledků je dobře patrné, že opakovaný požadavek s mezipamětí je téměř o 95 % rychlejší (tento výsledek je přirozený a očekávaný efekt použití vyrovnávací paměti). Navíc se neprovádí žádná komunikace se serverem. Výsledky samozřejmě závisí zejména na rychlosti síťové komunikace a délka provádění operací na serveru je v tomto případě zanedbatelná.

| požadavek | doba bez mezipaměti | doba s mezipamětí | zrychlení |
|---|---|---|---|
| první | 37,4±1,5 ms | 37,3±1,5 ms | 0,3 % |
| opakovaný | 37,4±1,5 ms | 1,9±0,5 ms | 94,9 % |

Tabulka 6.1: Rychlost provedení operace `query_info`

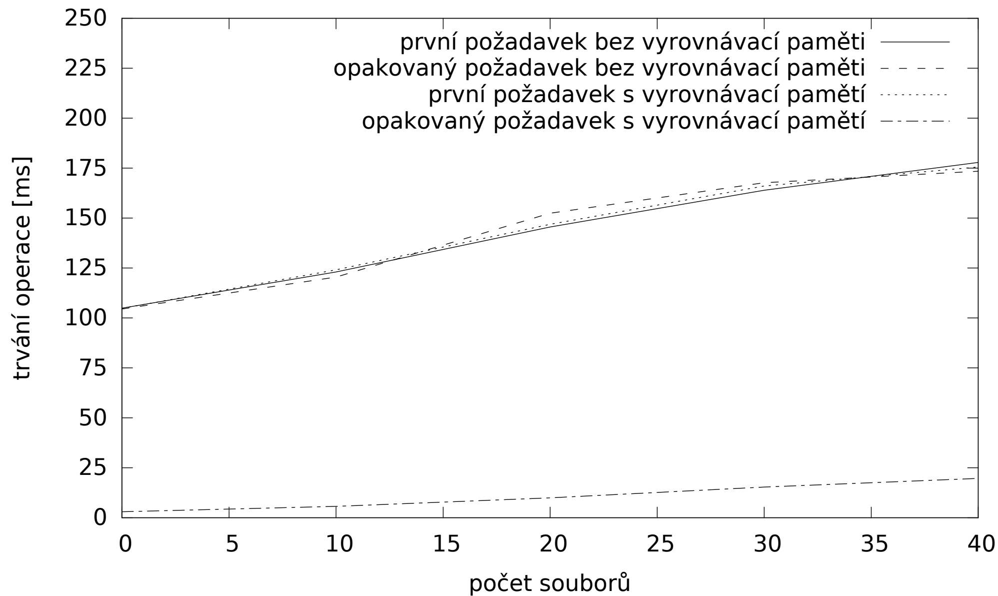
Pro operaci `enumerate` jsou výsledky v tabulce 6.2. Výsledky jsou také zaneseny do grafu na obrázku 6.1.

| požadavek | počet potomků | doba bez mezipaměti | doba s mezipamětí | zrychlení |
|---|---|---|---|---|
| první | 0 | 104,9±5,2 ms | 104,7±6,0 ms | 0,2 % |
| první | 10 | 123,0±8,9 ms | 124,1±9,7 ms | -0,9 % |
| první | 20 | 145,5±10,0 ms | 146,8±10,4 ms | -0,9 % |
| první | 30 | 163,9±9,4 ms | 166,1±8,6 ms | -1,3 % |
| první | 40 | 177,9±9,7 ms | 175,5±8,0 ms | 1,3 % |
| opakovaný | 0 | 104,5±5,4 ms | 3,0±0,8 ms | 97,1 % |
| opakovaný | 10 | 120,5±6,8 ms | 5,7±1,0 ms | 95,3 % |
| opakovaný | 20 | 152,4±5,4 ms | 9,9±0,7 ms | 93,5 % |
| opakovaný | 30 | 167,7±4,8 ms | 15,3±0,7 ms | 90,8 % |
| opakovaný | 40 | 173,5±8,2 ms | 19,7±1,2 ms | 88,6 % |

Tabulka 6.2: Rychlost provedení operace `enumerate`

[1] Jedná se pravděpodobně o chybu v sytému GVfs, kterou bude potřeba nahlásit po podrobnějším prozkoumání.

Výsledky jsou v tomto případě uvedeny pro různě velké adresáře. S rostoucí velikostí adresáře se při opakovaném požadavku mění i zrychlení. Pro prázdný adresář je zrychlení dokonce asi o 97 %. Pro adresář o čtyřiceti potomcích je již zrychlení necelých 90 %. Zrychlení se zmenšuje, protože sběrnice D-Bus je pravděpodobně pro přenos všech atributů příliš pomalá.




Obrázek 6.1: Graf rychlosti provádění operace `enumerate`. První tři typy požadavků (první požadavek s a bez vyrovnávací paměti a opakovaný požadavek bez vyrovnávací paměti) dosahují podobných rychlostí. Rozdíl je zde vidět právě pro opakovaný požadavek s vyrovnávací pamětí (dolní čerchovaná křivka).

### Rychlost práce se správci souborů

Předchozí testy jsou příliš umělé, smyslem těch následujících je simulace reálné činnosti souborových manažerů. Jedná se tedy o kombinaci různých operací (nejen `enumerate` a `query_info`). Testy simulují průchod souborovým systémem, prováděné operace vycházejí z analýzy v příloze B. Zaměřeny jsou na Nautilus, který je hlavním souborovým manažerem GNOME, a tedy i hlavním klientem GVfs. Uváděné časy opět vznikly jako aritmetický průměr z 25 iterací. Výsledky těchto testů poukazují na smysluplnost celé práce.

Tabulka 6.3 zaznamenává časy a směrodatné odchylky pro otevření adresáře a jeho opakované otevření. Výsledek je také graficky znázorněn na obrázku 6.2. Při těchto požadavcích se provádí stejné operace, které provádí správce Nautilus při otevření adresáře. V tabulce jsou opět časy s i bez použití vyrovnávací paměti. Již při prvním otevření adresáře dochází ke zrychlení o 20-25 % v závislosti na velikosti adresáře. Při opakovaném otevření stejného adresáře dojde ke zrychlení přibližně o 80 %. Stejně tak i při následném otevření podadresáře bude urychlení o něco větší, než při prvním přístupu. Toto větší zrychlení však předpokládá, že jsou data v mezipamětích stále validní. V případě SFTP, pro které byla invalidace podle času nastavena na tři vteřiny, je však pravděpodobnost této situace poměrně malá.

| požadavek | počet potomků | doba bez mezipaměti | doba s mezipamětí | zrychlení |
|---|---|---|---|---|
| první | 0 | 250,9±7,9 ms | 188,9±7,8 ms | 24,7 % |
| první | 10 | 267,0±8,2 ms | 201,0±7,1 ms | 24,7 % |
| první | 20 | 293,5±9,6 ms | 222,0±10,9 ms | 24,3 % |
| první | 30 | 321,5±8,2 ms | 243,9±10,6 ms | 24,1 % |
| první | 40 | 324,1±9,3 ms | 258,4±10,3 ms | 20,3 % |
| opakovaný | 0 | 252,5±7,6 ms | 44,1±2,4 ms | 82,5 % |
| opakovaný | 10 | 268,3±6,3 ms | 48,4±2,2 ms | 82,0 % |
| opakovaný | 20 | 294,8±8,9 ms | 53,2±4,1 ms | 82,0 % |
| opakovaný | 30 | 316,9±3,2 ms | 57,4±2,8 ms | 81,8 % |
| opakovaný | 40 | 322,7±10,1 ms | 64,1±5,8 ms | 80,1 % |

Tabulka 6.3: Rychlost otevření adresáře a opakovaný požadavek v aplikaci Nautilus

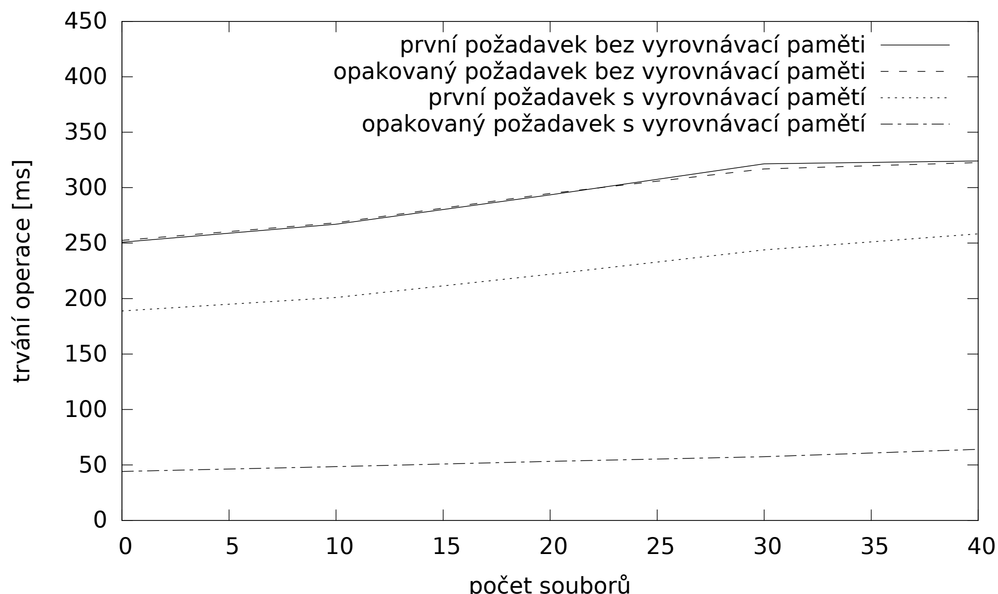
Jako u předešlých testů ani zde významně nezávisí na počtu potomků vypisovaných adresářů. U jiných souborových manažerů by však velikost adresáře mohla mít zásadní vliv na urychlení (např. Thunar), kde po výpisu adresáře následují dotazy na `GFileInfo` pro všechny potomky.

Ukázka spuštění výkonnostních testů je následující:

```
# spuštění démona z upraveného GVfs
GVFS_CACHE=1 /opt/gnome/libexec/gvfsd --replace &

# výkonnostní testy
test/benchmark-info-cache sftp://example.com/ 25 query_info
```

Test předpokládá spuštěného hlavního démonu z upraveného GVfs s podporou vyrovnávacích pamětí jako v případě funkcionálních testů. Pomocí parametrů příkazové řádky




Obrázek 6.2: Graf rychlosti otevření složky v aplikaci Nautilus. První i opakovaný požadavek bez použití vyrovnávacích pamětí dosahují podobných rychlostí. Rozdíl je zde vidět již při prvním požadavku s vyrovnávací pamětí (prostřední tečkovaná křivka). Pro opakovaný požadavek s vyrovnávací pamětí je rozdíl ještě výraznější (dolní čerchovaná křivka).

je předána cesta k nepřipojenému souborovému systému s povolenými vyrovnávacími pamětmi, počet iterací a jméno testu (`query_info`, `enumeration`, či `nautilus`).

# Kapitola 7

# Závěr

Cílem této diplomové práce bylo nastudovat systém GVfs a navrhnout pro něj vyrovnávací paměť. V práce byly nejdříve popsány principy vyrovnávacích pamětí a jejich algoritmy. Následně byly popsány programátorské knihovny GLib, abstrakce souborového systému GIO a komunikační sběrnice D-Bus. Na těchto technologiích je vystavěn virtuální souborový systém GVfs, který byl dále popsán. V práce byla diskutována komunikace v systému GVfs a slabiny jednotlivých modulů. Bylo například zjištěno, že některé moduly již obsahují různé implementace interních vyrovnávacích pamětí.

Při zkoumání zdrojových kódů bylo také objeveno několik chyb[1] v systému GVfs. Objevené chyby byly nahlášeny a pro některé byly poskytnuty opravy, které byly vývojáři přijaty a zahrnuty do oficiálních zdrojových kódů.

Na základě předchozí analýzy systému GVfs byla určena slabá místa, která lze pomocí mezipamětí zlepšit. Byly navrženy tři druhy vyrovnávacích pamětí: vyrovnávací paměť pro informace o souborech, druhá pro výpisy adresářů a třetí pro data souborů. První dva typy mají za cíl zrychlit práci s virtuálními souborovými systémy, třetí pak má poskytnout chybějící funkcionalitu, zejména emulaci proudových operací.

Navržené vyrovnávací paměti byly implementovány v jazyce C pro systém GVfs. Použití pamětí je pro moduly GVfs transparentní a nevyžaduje větší zásahy do zdrojových kódů. V modulech, u kterých je to žádoucí, musí být pouze vytvořena vlastní instance těchto pamětí. Při vytváření mezipamětí pro metadata může být zvolen limit velikosti paměti a doba, po které jsou data invalidována. Při zaplnění paměti jsou pak použity algoritmy LRU či Size Adjusted LRU pro určení oběti. Implementována byla i jednoduchá správa paměti, která odmazává neplatné položky podle času. Vyrovnávací paměť pro data byla implementována pro emulaci proudových operací pouze při čtení. K práci byla vytvořena dokumentace formou komentářů podle syntaxe GTK-Doc.

Součástí diplomové práce jsou i funkcionální a výkonnostní testy. V rámci funkcionálního testování byly vytvořeny jednotkové a integrační testy. Výkonnostní testy pro SFTP modul ukázaly, že opakované požadavky na informace o souborech byly přibližně o 95 % rychlejší. Opakovaný výpis adresáře již závisel na počtu potomků, nicméně do třiceti potomků bylo urychlení stále o více než 90 %. Samotné otevření adresáře ve správci souborů Nautilus bylo již při prvním otevření urychleno asi o 20-25 % v závislosti na velikosti adresáře. Opakované otevření stejného adresáře či podadresáře opět přináší větší zrychlení. Dokumentované zrychlení dokazuje hlavní smysl této diplomové práce. Vedlejším efektem

[1] `https://bugzilla.gnome.org/show_bug.cgi?id=721980`,
`https://bugzilla.gnome.org/show_bug.cgi?id=721981`,
`https://bugzilla.gnome.org/show_bug.cgi?id=719495`,
`https://bugzilla.gnome.org/show_bug.cgi?id=710790`

mimo zrychlení je pak samozřejmě menší zatížení sítě či připojeného zařízení.

Pro demonstraci byly vyrovnávací paměti integrovány do modulu pro protokol PTP, který byl díky tomu značně zjednodušen. Podobné zjednodušení je možné i v řadě dalších modulů. V rámci práce byla také navržena různá vylepšení, které je možné do budoucna provést.

Zadání bylo vypsáno za účelem vylepšení systému GVfs pomocí vyrovnávací paměti. Řešení bylo po celou dobu diskutováno s vývojáři GVfs. Komunikace s vývojáři probíhala formou osobních konzultací a soukromé e-mailové korespondence. Zdrojové kódy vyrovnávacích pamětí musí projít nejprve kontrolou, než je bude možné zařadit do oficiálních zdrojových kódů GVfs. Vývojáři tohoto projektu jsou však velmi vytížení, a proto nebyly příslušné kontroly v době odevzdání práce ještě provedeny.

# Literatura

[1] JACOB, Bruce, Spencer W NG a David T WANG. Memory systems: cache, DRAM, disk. Burlington: Morgan Kaufmann Publishers, 2008. ISBN 978-0-12-379751-3.

[2] MCHOES, Ann a Ida M. FLYNN. Understanding Operating Systems. 6. vydání. Cengage Learning, 2010. ISBN 14-390-7920-X.

[3] TANENBAUM, Andrew S. Modern operating systems. 3. vydání. Upper Saddle River, Prentice Hall, 2008. ISBN 978-0136006633.

[4] AYBAR, Edwin. Optimizing both hit rate and byte hit rate in web cache replacement policies [online]. Texas Tech University, 2002 [cit. 2014-05-05]. Dostupné z: http://repositories.tdl.org/ttu-ir/bitstream/handle/2346/18797/31295017075770.pdf

[5] KROEGER, Thomas. Design and Implementation of a Predictive File Prefetching Algorithm: The Linux Kernel's VFS Layer [online]. Usenix Annual Technical Conference, Boston. Berkeley: Usenix Association, 2001 [cit. 2014-05-05]. ISBN 18-804-4609-X. Dostupné z: https://www.usenix.org/legacy/event/usenix01/full_papers/kroeger/kroeger_html/node8.html

[6] RUSLING, David. The Linux Kernel: The File system [online]. United Kingdom, 1999 [cit. 2014-05-05]. Dostupné z: http://www.tldp.org/LDP/tlk/fs/filesystem.html

[7] GLib Overview. GLib Reference Manual [online]. The GNOME Project [cit. 2014-01-07]. Dostupné z: http://developer.gnome.org/glib/2.38/glib.html

[8] GObject: The base object type. GLib Reference Manual [online]. The GNOME Project [cit. 2013-12-30]. Dostupné z: https://developer.gnome.org/gobject/2.38/gobject-The-Base-Object-Type.html

[9] Type Information: The GLib Runtime type identification and management system. GLib Reference Manual [online]. The GNOME Project [cit. 2013-12-30]. Dostupné z: https://developer.gnome.org/gobject/2.38/gobject-Type-Information.html

[10] Error Reporting: a system for reporting errors. GLib Reference Manual [online]. The GNOME Project [cit. 2013-12-30]. Dostupné z: https://developer.gnome.org/glib/2.38/glib-Error-Reporting.html

[11] GCancellable: Thread-safe Operation Cancellation Stack. GLib Reference Manual [online]. The GNOME Project [cit. 2013-12-30]. Dostupné z: https://developer.gnome.org/gio/2.38/GCancellable.html

[12] GAsyncResult: Asynchronous Function Results. GLib Reference Manual [online]. The GNOME Project [cit. 2013-12-30]. Dostupné z: https://developer.gnome.org/gio/2.38/GAsyncResult.html

[13] Thread Pools: pools f threads to execute work concurrently. GLib Reference Manual [online]. The GNOME Project [cit. 2013-12-30]. Dostupné z: https://developer.gnome.org/glib/2.38/glib-Thread-Pools.html

[14] File Utilities: Various file-related functions. GLib Reference Manual [online]. The GNOME Project [cit. 2014-01-07]. Dostupné z: https://developer.gnome.org/glib/2.38/glib-File-Utilities.html

[15] GDrive: Drive management. GIO Reference Manual [online]. The GNOME Project [cit. 2014-01-04]. Dostupné z: https://developer.gnome.org/gio/2.38/GDrive.html

[16] GVolume: Volume management. GIO Reference Manual [online]. The GNOME Project [cit. 2014-01-04]. Dostupné z: https://developer.gnome.org/gio/2.38/GVolume.html

[17] GMount: Mount management. GIO Reference Manual [online]. The GNOME Project [cit. 2014-01-04]. Dostupné z: https://developer.gnome.org/gio/2.38/GMount.html

[18] GMountOperation: Object used for authentication and user interaction. GIO Reference Manual [online]. The GNOME Project [cit. 2014-01-07]. Dostupné z: https://developer.gnome.org/gio/2.38/GMountOperation.html

[19] GVolumeMonitor: Volume monitor. GIO Reference Manual [online]. The GNOME Project [cit. 2014-01-04]. Dostupné z: https://developer.gnome.org/gio/2.38/GMount.html

[20] GVfs: Developer Documentation [online]. The GNOME Project [cit. 2013-12-30]. Dostupné z: https://wiki.gnome.org/Projects/gvfs/doc

[21] GFile: File and Directory Handling. GIO Reference Manual [online]. The GNOME Project [cit. 2014-01-07]. Dostupné z: https://developer.gnome.org/gio/2.38/GFile.html

[22] GFileInfo: File Information and Attributes. GIO Reference Manual [online]. The GNOME Project [cit. 2014-01-07]. Dostupné z: https://developer.gnome.org/gio/2.38/GFileInfo.html

[23] GFileAttribute: Key-Value Paired File Attributes. GIO Reference Manual [online]. The GNOME Project [cit. 2014-01-07]. Dostupné z: https://developer.gnome.org/gio/2.38/gio-GFileAttribute.html

[24] GFileEnumerator: Enumerated Files Routines. GIO Reference Manual [online]. The GNOME Project [cit. 2014-01-07]. Dostupné z: https://developer.gnome.org/gio/2.38/GFileEnumerator.html

[25] D-Bus Specification [online]. Freedesktop.org [cit. 2014-01-07]. Dostupné z: http://dbus.freedesktop.org/doc/dbus-specification.html

[26] DBusRemote [online]. Freedesktop.org [cit. 2014-05-25]. Dostupné z: http://www.freedesktop.org/wiki/Software/DBusRemote/

[27] DBusConnection: Connection to another application. D-Bus Documentation [online]. Freedesktop.org [cit. 2014-05-25]. Dostupné z: http://dbus.freedesktop.org/doc/api/html/group__DBusConnection.html

[28] DBusBindings [online]. Freedesktop.org [cit. 2014-05-25]. Dostupné z: http://www.freedesktop.org/wiki/Software/DBusBindings/

[29] GVfs: Virtual File System. GIO Reference Manual [online]. The GNOME Project [cit. 2014-01-08]. Dostupné z: https://developer.gnome.org/gio/2.38/GVfs.html

[30] Extension Points. GIO Reference Manual [online]. The GNOME Project [cit. 2014-01-08]. Dostupné z: https://developer.gnome.org/gio/2.38/gio-Extension-Points.html

[31] LARSSON, Alexander. Plans for gnome-vfs replacement [online]. The GNOME Project [cit. 2013-12-30]. Dostupné z: https://mail.gnome.org/archives/gtk-devel-list/2006-September/msg00072.html

[32] Introduction. JHBuild Manual [online]. The GNOME Project [cit. 2014-05-07]. Dostupné z: https://developer.gnome.org/jhbuild/3.10/introduction.html

[33] Introduction. GTK-Doc Manual [online]. The GNOME Project [cit. 2014-05-10]. Dostupné z: https://developer.gnome.org/gtk-doc-manual/1.20/whatisgtkdoc.html

[34] Testing: a test framework. GLib Reference Manual [online]. The GNOME Project [cit. 2014-05-10]. Dostupné z: https://developer.gnome.org/glib/2.38/glib-Testing.html

# Příloha A

# Podporované operace v modulech

V této příloze jsou shrnuty podporované operace moduly GVfs. Data jsou uvedena pro stabilní verzi GVfs 1.19.4 (součást GNOME 3.10).

## A.1 Operace pro práci s metadaty

Tabulka A.1 shrnuje podporu operací pro práci s metadaty. Ve výčtu jsou vynechány operace `query_info_on_read` a `query_info_on_write`, které nejsou téměř podporovány.

## A.2 Operace pro práci s daty

Tabulky A.2 a A.3 obsahují operace pro čtení a zápis souborů. Během tvorby této práce se však již objevily implementace pro některé chybějící funkce ve vývojové verzi[1]. V tabulkách chybí některé málo podporované metody (např. `trash`).

[1] `https://bugzilla.gnome.org/show_bug.cgi?id=642814`

| modul | query_info/enumerate | query_fs_info | set_display_name | set_attribute |
|---|---|---|---|---|
| GVfsBackendAfc | do | do | do | do |
| GVfsBackendAfp | try | try | try | try |
| GvfsBackendArchive | do | try | - | - |
| GVfsBackendCdda | do | do | - | - |
| GVfsBackendDav | do | do | do | - |
| GVfsBackendFtp | do | - | do | do |
| GVfsBackendGphoto2 | do/try | do/try | do | - |
| GVfsBackendHttp | try | - | - | - |
| GVfsBackendMtp | do | do | do | - |
| GVfsBackendObexftp | do | do | - | - |
| GVfsBackendSftp | try | try | try | try |
| GVfsBackendSmb | do | do | do | - |

Tabulka A.1: Podporované operace pro práci s metadaty

| modul | open_for_read/close_read | read | seek_on_read | pull |
|---|---|---|---|---|
| GVfsBackendAfc | do | do | do | - |
| GVfsBackendAfp | try | try | try | - |
| GvfsBackendArchive | do | do | - | - |
| GVfsBackendCdda | do | do | - | - |
| GVfsBackendDav | try | try | try | - |
| GVfsBackendFtp | do | do | - | do |
| GVfsBackendGphoto2 | do | try | try | - |
| GVfsBackendHttp | try | try | try | - |
| GVfsBackendMtp | do | do | do | do |
| GVfsBackendObexftp | do | do | - | - |
| GVfsBackendSftp | try | try | try | try |
| GVfsBackendSmb | do | do | do | - |

Tabulka A.2: Podporované operace čtení dat

| modul | create/replace/write/close_write | append_to | seek_on_write/truncate | delete | copy | move | push |
|---|---|---|---|---|---|---|---|
| GVfsBackendAfc | do | do | do | do | - | do | - |
| GVfsBackendAfp | try | try | try | try | try | try | - |
| GvfsBackendArchive | - | - | - | - | - | - | - |
| GVfsBackendCdda | - | - | - | - | - | - | - |
| GVfsBackendDav | try | - | - | - | - | - | - |
| GVfsBackendFtp | do | do | - | do | - | do | - |
| GVfsBackendGphoto2 | do | do | do | do | - | do | - |
| GVfsBackendHttp | - | - | - | - | - | - | - |
| GVfsBackendMtp | do | do | do | do | - | - | do |
| GVfsBackendObexftp | - | - | - | - | - | - | do |
| GVfsBackendSftp | try | try | try | try | - | try | try |
| GVfsBackendSmb | do | do | do | do | - | do | - |

Tabulka A.3: Podporované operace zápisu dat

# Příloha B

# Analýza komunikace s GVfs

Tato příloha ukazuje různé přístupy pro práci se souborovými systémy. Pomocí několika aplikací využívajících GIO byl otevřen adresář v připojeném souborovém systému pomocí GVfs. Otevíraný adresář obsahoval jeden podadresář. V následujících sekcích jsou pak uvedeny zjednodušené komentované ladící výstupy.

## B.1 Výpis adresáře aplikací Nautilus

Následující výpis ukazuje ladící výstupy, které byly získány při otevření adresáře pomocí správce souborů Nautilus:

```
// Požadavek na GQueryInfo pro otevíraný adresář.
Queued new job 0xf87650 (GVfsJobQueryInfo)
// (attributes = *, flags = 0)
send_reply(0xf87650), failed=0 ()

// Požadavek na GQueryInfo pro souborový systém otevíraného adresáře.
Queued new job 0xf80eb0 (GVfsJobQueryFsInfo)
// (attributes = filesystem::*)
send_reply(0xf80eb0), failed=0 ()

// Požadavek na vytvoření monitoru pro otevíraný adresář.
Queued new job 0xf84540 (GVfsJobCreateMonitor)
send_reply(0xf84540), failed=0 ()

// Požadavek na GFileEnumerator pro otevíraný adresář.
Queued new job 0xf6cb90 (GVfsJobEnumerate)
// (attributes = *, flags = 0)
send_reply(0xf6cb90), failed=0 ()

// Dva požadavky na GQueryInfo pro otevíraný adresář.
Queued new job 0xf86b70 (GVfsJobQueryInfo)
// (attributes = *, flags = 0)
send_reply(0xf86b70), failed=0 ()
Queued new job 0x7f96ec009970 (GVfsJobQueryInfo)
// (attributes = *, flags = 0)
send_reply(0x7f96ec009970), failed=0 ()
```

Z výpisu je dobře vidět, že požadavky na `GQueryInfo` jsou zbytečně prováděny několikrát.

## B.2 Výpis adresáře aplikací Tux Commander

Výpis prováděných operací při otevření adresáře souborovým manažerem *Tux Commander*[1] je následující:

```
// Dva požadavky na GFileEnumerator pro otevíraný adresář.
Queued new job 0x10bef00 (GVfsJobEnumerate)
// (attributes = standard::*,time::*,unix::*, flags = 1)
send_reply(0x10bef00), failed=0 ()
Queued new job 0x10bea30 (GVfsJobEnumerate)
// (attributes = standard::*,time::*,unix::*, flags = 1)
send_reply(0x10bea30), failed=0 ()

// Dva požadavky na GFileInfo pro souborový systém otevíraného adresáře.
Queued new job 0x7f819c02e820 (GVfsJobQueryFsInfo)
// (attributes = filesystem::size)
send_reply(0x7f819c02e820), failed=0 ()
Queued new job 0x7f819c02eb80 (GVfsJobQueryFsInfo)
// (attributes = filesystem::free)
send_reply(0x7f819c02eb80), failed=0 ()
```

Opakovaně se zde vyskytuje výpis souborů a požadavek na `GFileInfo`.

## B.3 Výpis adresáře démonem FUSE

FUSE démon je součástí GVfs (viz sekce 3). Následuje výpis prováděných operací tímto démonem při otevření adresáře:

```
// Požadavek na GFileInfo pro nadřazený adresář.
Queued new job 0xf8c960 (GVfsJobQueryInfo)
// (attributes = standard::*,unix::*,time::*,access::*, flags = 0)
send_reply(0xf8c960), failed=0 ()

// Požadavek na GFileInfo pro otevíraný adresář.
Queued new job 0xf8c8c0 (GVfsJobQueryInfo)
// (attributes = standard::*,unix::*,time::*,access::*, flags = 0)
send_reply(0xf8c8c0), failed=0 ()

// Požadavek na GFileEnumerator pro otevíraný adresář.
Queued new job 0x7f96e40074e0 (GVfsJobEnumerate)
# (filename = /mnt, attributes = standard::name, flags = 0)
send_reply(0x7f96e40074e0), failed=0 ()

// Požadavek na GFileInfo na potomka.
Queued new job 0xf8c820 (GVfsJobQueryInfo)
// (attributes = standard::*,unix::*,time::*,access::*, flags = 0)
send_reply(0xf8c820), failed=0 ()
```

Jsou zde dotazy pro nadřazený adresář či potomky, a proto dochází k opakování požadavků pokud uživatel prochází souborovým systémem.

[1] `http://tuxcmd.sourceforge.net/`

## B.4 Výpis adresáře aplikací Thunar

Následující výpis byl získán ze správce souborů *Thunar*[2]:

```
// Požadavak na GFileInfo pro otevíraný adresář.
Queued new job 0x7f96c4002850 (GVfsJobQueryInfo)
// (attributes = standard::type, flags = 0)
send_reply(0x7f96c4002850), failed=0 ()

// Požadavek na GFileInfo pro souborový systém otevíraného adresáře.
Queued new job 0x7f96d0013650 (GVfsJobQueryFsInfo)
// (attributes = filesystem::use-preview)
send_reply(0x7f96d0013650), failed=0 ()

// Požadavek na GFileEnumerator pro otevíraný adresář.
Queued new job 0xf6c8d0 (GVfsJobEnumerate)
// (attributes = *, flags = 0)
send_reply(0xf6c8d0), failed=0 ()

// Požadavek na GFileInfo pro nadřazený adresář.
Queued new job 0xf86c10 (GVfsJobQueryInfo)
// (attributes = *, flags = 0)
send_reply(0xf86c10), failed=0 ()

// Požadavek na vytvoření monitoru pro otevíraný adresář.
Queued new job 0xf865c0 (GVfsJobCreateMonitor)
send_reply(0xf865c0), failed=0 ()

// Požadavek na GFileInfo pro souborový systém potomka.
Queued new job 0x7f96d4011cc0 (GVfsJobQueryFsInfo)
// (attributes = filesystem::use-preview)
send_reply(0x7f96d4011cc0), failed=0 ()

// Dva požadavky na GFileInfo pro souborový systém otevíraného adresáře.
Queued new job 0xf841e0 (GVfsJobQueryFsInfo)
// (attributes = filesystem::*)
send_reply(0xf841e0), failed=0 ()
Queued new job 0xf80f40 (GVfsJobQueryFsInfo)
// (attributes = filesystem::use-preview)
send_reply(0xf80f40), failed=0 ()
```

Tentokrát jsou zde opakované požadavky na `GFileInfo` pro souborový systém. Opět jsou zde požadavky na nadřazený adresář či potomky, které se budou provádět vícekrát při průchodu souborovým systémem.

## B.5 Výpis adresáře aplikací Double Commander

Prováděné operace při výpisu adresáře aplikací *Double Commander*[3] jsou následující:

```
// Dva požadavky na GFileInfo pro nadřazený adresář.
Queued new job 0x10cf6f0 (GVfsJobQueryInfo)
// (attributes = standard::*,unix::*,time::*,access::*, flags = 0)
send_reply(0x10cf6f0), failed=0 ()
Queued new job 0x10cf510 (GVfsJobQueryInfo)
// (attributes = access::*, flags = 0)
send_reply(0x10cf510), failed=0 ()
```

[2] `http://docs.xfce.org/xfce/thunar/start`
[3] `http://doublecmd.sourceforge.net/`

```
// Požadavek na GFileInfo pro souborový systém nadřazeného adresáře.
Queued new job 0x7f819c002ce0 (GVfsJobQueryFsInfo)
// (attributes = filesystem::*)
send_reply(0x7f819c002ce0), failed=0 ()

// Požadavek na GFileInfo pro nadřazený adresář.
Queued new job 0x10cf330 (GVfsJobQueryInfo)
// (attributes = access::*, flags = 0)
send_reply(0x10cf330), failed=0 ()

// Požadavek na GFileInfo pro souborový systém nadřazeného adresáře.
Queued new job 0x7f819c002c50 (GVfsJobQueryFsInfo)
// (attributes = filesystem::*)
send_reply(0x7f819c002c50), failed=0 ()

// Tři požadavky na GFileInfo pro otevíraný adresář.
Queued new job 0x10cf150 (GVfsJobQueryInfo)
// (attributes = standard::*,unix::*,time::*,access::*, flags = 0)
send_reply(0x10cf150), failed=0 ()
Queued new job 0x10ad530 (GVfsJobQueryInfo)
// (attributes = access::*, flags = 0)
send_reply(0x10ad530), failed=0 ()
Queued new job 0x10ad030 (GVfsJobQueryInfo)
// (attributes = access::*, flags = 0)
send_reply(0x10ad030), failed=0 ()

// Požadavek na GFileEnumerator pro otevíraný adresář.
Queued new job 0x10be980 (GVfsJobEnumerate)
// (attributes = standard::name, flags = 0)
send_reply(0x10be980), failed=0 ()

// Požadavek na GFileInfo pro otevíraný adresář.
Queued new job 0x10ad0d0 (GVfsJobQueryInfo)
// (attributes = standard::*,unix::*,time::*,access::*, flags = 0)
send_reply(0x10ad0d0), failed=0 ()

// Požadavek na GFileInfo pro potomka.
Queued new job 0x10cf010 (GVfsJobQueryInfo)
// (attributes = standard::*,unix::*,time::*,access::*, flags = 0)
send_reply(0x10cf010), failed=0 ()

// Požadavek na GFileInfo pro souborový systém otevíraného adresáře.
Queued new job 0x7f819c02ec10 (GVfsJobQueryFsInfo)
// (filename = /test, attributes = filesystem::*)
send_reply(0x7f819c02ec10), failed=0 ()
```

Při otevření adresáře se několikrát provádí požadavek na `GFileInfo` a to jak na otevíraný tak nadřazený adresář. Stejně jako v předchozích případech se provádí některé operace zbytečně při průchodu souborovým systémem.

# Příloha C

# Obsah přiloženého DVD

Na přiloženém DVD se nachází níže uvedené soubory a složky:

- `/gvfs/` – upravené zdrojové kódy systému GVfs včetně testů (GIT repositář)
- `/lyx/` – kompletní zdrojové kódy této technické zprávy ve formátu *Lyx*[1]
- `/patch/` – záplaty s implementovaným kódem pro systém GVfs
- `/virtualbox/` – Fedora 20 s upraveným GVfs (disk pro *VirtualBox*[2])
- `/README.txt` – návod na vyzkoušení implementované funkcionality
- `/xholyo00.pdf` – elektronická verze této technické zprávy

[1] `http://www.lyx.org/`
[2] `https://www.virtualbox.org/`