Západočeská univerzita v Plzni
Fakulta aplikovaných věd
Katedra informatiky a výpočetní techniky

Bakalářská práce

KIVDFS
Klient pro GNU/Linux s pomocí FUSE

Plzeň 2014 Lukáš Kvídera

Prohlášení

Prohlašuji, že jsem bakalářskou práci vypracoval samostatně a výhradně s použitím citovaných pramenů.

V Plzni dne 28. dubna 2014

Lukáš Kvídera

Abstract

The main goal of this bachelor thesis is implementation of a client application for KIVFS in C language with FUSE kernel module. The theoretical part discusses caching policies and options available to implement acces to the KIVFS. Practical part is focused on measuerement of eficiency of caching algorithms and throughput of the client application.

Abstrakt

Hlavním cílem této bakalářské práce je návrh a implementace klientské aplikace umožňující přístup ke KIVFS pomocí FUSE. V teoretické části jsou podrobněji popsány cachovací algoritmy a možnosti implementace přístupu ke KIVFS. Praktická část je zaměřena na měření efektivity cachovacích algoritmů a propustnost klientské aplikace.

Obsah

1 Úvod

V současné době uživatelé potřebují ukládat a pracovat s daty. Existuje více možností, jak přistupovat k souborům. Mezi nejčastější patří přístup k souborům na lokálních pevných discích. Jeho výhodou je vysoká rychlost přenosu dat a nízké latence. Nevýhodou tohoto přístupu je ztížené udržování aktuálních kopií souborů na vícero místech a přístup k těmto datům vzdáleně. Tento neduh řeší síťové alternativy, mezi které patří například Dropbox, Wuala, Google Drive či domácí NAS. Data jsou kdykoliv dostupná z libovolného umístění, tudíž je podstatně snížena roztříštěnost verzí souborů na různých zařízeních.

Data na lokálních discích jsou uchovávána v datových strukturách – souborových systémech. Souborových systémů existuje velké množství. V prostředí Windows™je nejpoužívanější NTFS, na Linuxu v osobních počítačích EXT4. Každý souborový systém má nějaké klady i zápory, např. EXT4 podporuje žurnálování, čímž je omezeno riziko ztráty dat. Nevýhodou klasických souborových systémů je fragmentace souborů a tím snížená datová propustnost.

Proti tomu byly vyvinuty distribuované souborové systémy, jež se snaží vykompenzovat nedostatky lokálních souborových systémů.

Tato bakalářská práce se zabývá návrhem a implementací vzdáleného přístupu ke KIVFS v prostředí operačního systému Linux. KIVFS je experimentální souborový systém vyvíjený na katedře informatiky fakulty aplikovaných věd Západočeské univerzity.

Teoretická část je věnována převážně technologickým možnostem implementace přístupu ke vzdáleným souborům. Dále je krátce představen protokol KIVFS. V závěru teoretické části jsou představeny algoritmy pro správu cache a šifrování.

Praktická část je věnována detailnějšímu popisu implementace a měření výkonnosti a také v neposlední řadě měření efektivity nových algoritmů. V závěru jsou algoritmy porovnány a zhodnoceny.

2 Teoretická část

2.1 Pozadí práce se soubory

Operační systém je členěn na několik vrstev abstrakce, které se snaží o odstínění implementačních detailů tam, kde není jejich znalost nezbytná. Například uživatel nepotřebuje vědět, jakým způsobem se provede načtení textového souboru do editoru. Stačí mu kliknout na „Otevřít soubor“. Více informací o této vrstvě se nachází v sekci 2.1.1.

Pro vývojáře textového editoru je nezbytné, aby věděl, jakými úkony soubor v konkrétním operačním systému otevře – `fopen či open`. Přístupová práva k souboru ale řešit nemusí. Funkce otevření mu svou návratovou hodnotou a adekvátním nastavením proměnné `errno` dá najevo, zda vše proběhlo v pořádku, nebo nastal nějaký problém. Zde svou roli hraje VFS, který je blíže popsán v odstavci 2.1.2

Nejnižší vrstvy jsou doménou ovladačů zařízení a obsluhy přerušení od HW zařízení.

2.1.1 Aplikační vrstva

Aplikační vrstva je každému velmi dobře známá, jelikož se s ní setkává s každou spuštěnou aplikací – např. terminálový souborový manažer MidnightCommander či grafický Explorer na Windows™. Slouží především k interakci uživatele s aplikací a následně aplikace s operačním systémem. Uživatel nemůže příliš zasahovat do interních operací .

2.1.2 VFS

VFS – Virtual File System – je abstraktní vrstva nad všemi ovladači souborových systémů a zařízení. Jejím hlavním úkolem je zařízení zpřístupnit pod jménem (adresář /dev), omezovat přístup na základě přístupových práv a především poskytnout jednotné API odstiňující různé souborové systémy a jejich implementace. Nerozlišuje tedy, zda jsou data uložena lokálně nebo se přenášejí přes síť.

Ovladače všech souborových systémů musí implementovat jednotné rozhraní, přes které komunikují vyšší vrstvy. Jaké obslužné funkce z kterého modulu souborových systémů se konkrétně budou využívat je dáno specifikací typu souborového systému v /etc/fstab nebo parametrem -t příkazu `mount`.

Ovladač samotný má na starosti transformování uživatelských dat do svých interních struktur vhodných k ukládání na dané médium, pro něž je souborový systém určen. Zároveň poskytuje služby automatické údržby a většinou i defragmentace souborů.

2.2 KIVFS

V této kapitole budou zmíněny základní informace o distribuovaném souborovém systému KIVFS.

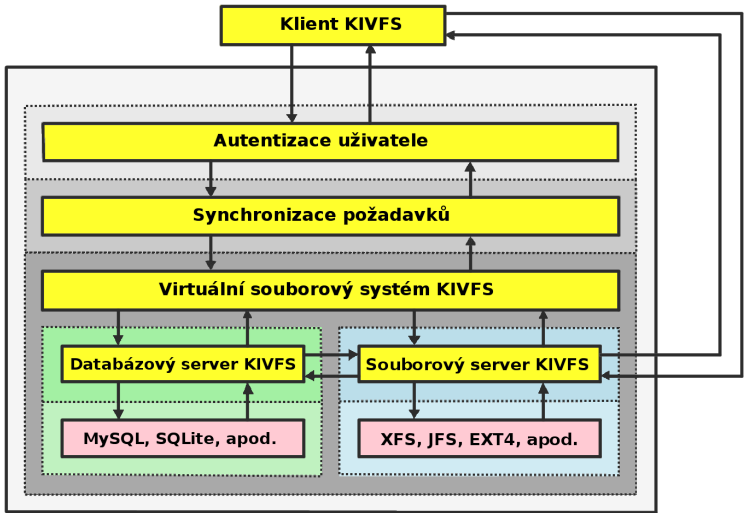
2.2.1 Důvody vzniku KIVFS

KIVFS nebo také KIVDFS je experimentální distribuovaný souborový systém vyvíjený na katedře informatiky Západočeské univerzity. Důvodem jeho vývoje je neexistence distribuovaného souborového systému, na který by byl možný přístup z mobilních zařízení. Unikátní vlastností KIVFS je možnost přístupu ze zařízení za „NATem“, jelikož servery KIVFS na rozdíl např. od AFS nevyužívají systém zpětných volání.

NAT je zkratka Network Address Translation – překlad síťových adres. NAT v dnešní době slouží především jako nouzové řešení u ISP[1], kteří nemají dostatek veřejných IP adres. Klienti těchto ISP jsou pro vnější internet schováni pod jedinou IP adresu a je tím znemožněno přímé kontaktování požadovaného stroje z vnější sítě. Problém neviditelnosti strojů za NATem částečně řeší směrování TCP/UDP portů na vyžádání na konkrétní privátní IP adresy. IPv6 s NAT nepočítá a problém zpětných volání by v tomto případě nebyl.

Další funkcionalitou KIVFS je podpora multimaster replikace, což znamená, že změny v souborech mohou být provedeny v jakékoli replice. Ta se pak sesynchronizuje s ostatními servery ještě před uvolněním výhradního zámku pro zápis.

S tím souvisí možnost vyvažování zátěže jednotlivých uzlů, kdy si každý udržuje tabulku vytížení a požadavek směruje na uzel s nejnižším zatížením. Směrování je prováděno na aplikační vrstvě TCP/IP modelu.



Obrázek 2.1: Hierarchická struktura kivfs [15]

[1] poskytovatelé připojení

2.2.2 Hierarchie KIVFS

KIVFS je hierarchicky členěn do několika jednotlivých vrstev znázorněných na obrázku 2.1. Jednotlivé vrstvy budou blíže popsány v následujících odstavcích.

Autentizační vrstva

Autentizační vrstva zajišťuje přihlašování a identifikaci uživatelů.

Synchronizační vrstva

Synchronizační vrstva je důležitá zejména pro udržení konzistence ukládaných souborů napříč všemi replikami.

VFS vrstva

VFS vrstva řídí tok požadavků dle jejich typu. Například dotaz na metadata je směrován pouze na databázový server, kdežto otevření souboru přesměruje požadavek jak do databáze, tak je poslán požadavek na otevření portu k přenosu dat na souborový server.

Vrstva souborového systému

Tato vrstva obstarává uložení dat na fyzických nosičích.

Databázová vrstva

Databázová vrstva má na starosti ukládání metadat souborů, uživatelských oprávnění a odkazů na soubory do fyzického úložiště.

2.2.3 Protokol KIVFS

Ke komunikaci mezi jednotlivými vrstvami byl vytvořen binární komunikační protokol, jenž má následující strukturu na architektuře amd64 (x86_64).

```
typedef struct {
  uint64_t session;      /* identifikátor sezení      */
  uint64_t timestamp;    /* časová známka             */
  uint64_t data_size;    /* velikost přenášených dat */
  uint32_t magicnumber; /* magicky zvolená konstanta*/
  uint32_t command;      /* kód RPC                   */
  int32_t return_code;   /* návratová hodnota         */
  int32_t type;
} kivfs_msg_head_t;
```

Struktura `kivfs_msg_head_t` by neměla být přenášena po síti tak jak je, jelikož stroje například s architekturou ARM by dostávaly pro ně nesrozumitelná data z důvodu rozdílné endianity [16]. Proto se před každým přenosem musí zkonvertovat do přenositelného síťového formátu. Síťový formát struktury `kivfs_msg_head_t` lze nalézt na adrese `http://students.kiv.zcu.cz/wiki/Aswi2010KivFS/Technicka-dokumentace`

2.3 Možnosti implementace přístupu ke KIVFS

V linuxovém prostředí je možné zvolit několik způsobů, kterými lze zajistit pohodlný přístup k souborovému systému KIVFS. Každý přístup má své výhody a nevýhody, jež budou detailněji rozebrány v následujících odstavcích.

2.3.1 Jaderný ovladač

Jaderný ovladač může být přímou součástí jádra nebo může být zaveden za běhu jako modul.

Jaderný ovladač svou výkonností předčí všechny ostatní varianty díky neexistující nutnosti přepínat uživatelský režim procesu na jaderný a zpět vícekrát, než je nezbytně nutné. Tedy uživatelská aplikace při zápisu knihovní funkcí `fwrite` interně zažádá[2] systém o službu `write`. Tím je způsobeno přepnutí procesu do jaderného režimu. Oproti FUSE (více informací se nachází v kapitole 2.3.2)je tato režie nejméně poloviční. Tím pádem by se daly předpokládat i menší energetické požadavky a tím větší výdrž mobilních zařízení.

Výkonnost tohoto řešení je vykoupena vyšší obtížností ladění[3] a překladu ovladače, kdy musí na počítači, kde by měl být modul ovladače překládán, být konkrétní verze jádra a k ní patřící zdrojové kódy.

2.3.2 FUSE

Filesystem in Userspace [9] byl původně vyvíjen pro potřeby AVFS (A Virtual File System), který umožňuje čtení archivů a vzdálených úložišť z existujících programů bez nutnosti jejich rekompilace. Dnes se ale jedná o dva nezávislé projekty. Jistou alternativou k FUSE by mohl být LUFS, ale jeho vývoj již dále nepokračuje, nebo GVFS, který je spjatý s prostředím Gnome.

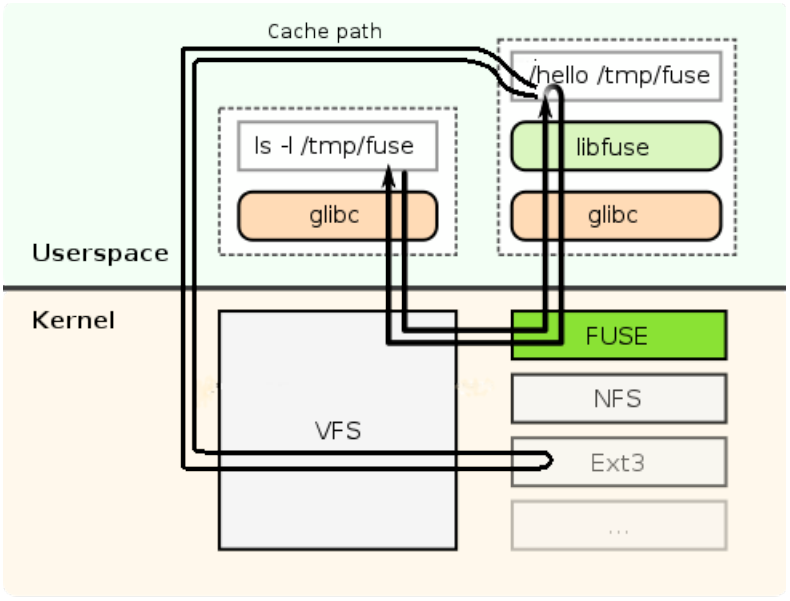
Hlavními cíli FUSE je poskytnout stabilní API pro vývojáře a především usnadnit vývoj a distribuci vlastních souborových systémů. FUSE především odstiňuje složité ladění jaderných ovladačů, které mohou způsobit v rané fázi vývoje i nestabilitu a pády celého systému. Programátor napíše pouze ovladač, který běží v uživatelském režimu a tím pádem

[2] Při zaplnění bufferu struktury `FILE`

[3] Přístup do paměti není omezen a kód, který by v uživatelském režimu způsobil segmentation fault, proběhne v "pořádku". Jen může přepsat důležitá data pro běh systému a způsobit tím jeho pád.

nemůže v systému způsobit přístup do paměti, která nebyla ovladači vyhrazena. Další výhodou FUSE je možnost připojovat souborový systém i z běžných uživatelských účtů. Jistou daní za tuto spolehlivost je snížení výkonu souborového systému z důvodu použití dvou vrstev ovladačů – FUSE sloužící jako proxy se stabilním API běžící v režimu jádra a ovladač vlastního souborového systému – a nutnosti změnit režim běhu procesu ovladače.

Oproti jadernému ovladači toto řešení ztrácí výkonově, jelikož se musí nejméně 4 krát změnit režim běhu[4]. Jsou-li v ovladači použita systémová volání, pak je to ještě několikrát více (znázorněno na Obr. 2.2). Nicméně pro použití v osobních počítačích je výkon stále dostačující, obzvláště, když mají být využity síťové přenosy, jež jsou řádově pomalejší než přímý přístup na disk.



Obrázek 2.2: Grafické znázornění změny běhového režimu v případě cachování [17]

Jaderný ovladač by našel své uplatnění u webových serverů nebo obecně tam, kde se očekává více uživatelů přistupujících ke zdrojům KIVFS na jedné stanici (např. eryx.zcu.cz).

Nevýhodou FUSE knihovny je, že nesděluje grafickému prostředí povahu souborového systému. Grafické aplikace pak předpokládají, že souborový systém se nachází na lokálním stroji a provádějí přednačítání náhledů a atributů všech souborů do své vlastní cache. Řešením by byla užší integrace FS do prostředí – například do GnomeVFS, které už rozlišuje lokalitu souborových systémů a sděluje ji aplikacím. Ty pak dle uživatelského nastavení vypínají generování náhledů na vzdálených souborových systémech.

[4]Vztaženo k obrázku 2.2, který reflektuje praktickou implementaci FUSE klientské aplikace.

2.3.3 Klientská aplikace

Klientská aplikace v podobě vlastního souborového manažeru je v porovnání s předchozími způsoby přístupu ke KIVFS obtížnější ve smyslu implementace použitelného a multiplatformního GUI. V linuxovém prostředí existují dvě hlavní větve grafických knihoven - GTK+ a Qt. Některé distribuce upřednostňují většinou pouze jednu variantu – Gnome + GTK, KDE + Qt. Z hlediska údržby a prostředků dostupných v běžných operačních systémech Linux je vývoj samostatné aplikace pouze pro přístup k datům neefektivní.

FUSE i jaderný ovladač jsou z uživatelského hlediska transparentní a je pouze na uživateli, s jakou aplikací se mu lépe pracuje při správě souborů.

2.3.4 Plug-in do již existujících souborových manažerů

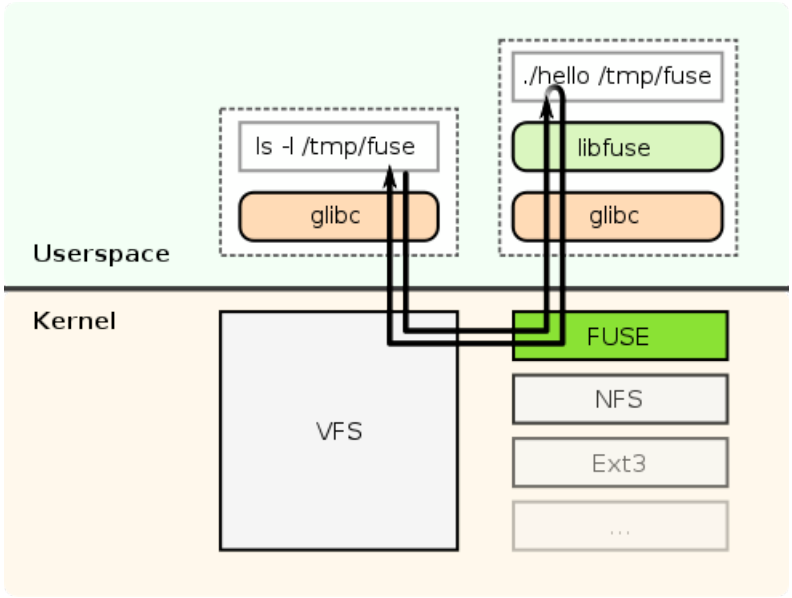
Další možností, jak získat přístup ke KIVFS, je použití již existujících souborových manažerů rozšířených o podporu tohoto souborového systému externím modulem. Výhodou je možnost používat tradiční aplikace pro správu souborů. Proti je nutnost vyvíjet modul pro každý správce souborů zvlášť. Navíc jiné aplikace opět nemají přímý a transparentní přístup ke vzdáleným souborům a uživatel je tak nepřímo nucen kopírovat nejprve vzdálená data na lokální úložiště.

2.3.5 Důvody volby FUSE pro implementaci přístupu ke KIVFS

Cílem bakalářské práce je návrh, implementace a otestování přístupu ke KIVFS z prostředí operačního systému Linux. FUSE byl zvolen z důvodu transparentního přístupu k datům z libovolné aplikace. Možnosti 2.3.3 a 2.3.4 toto neposkytují. Režie spojená s přepínáním uživatelského a jaderného režimu je zanedbatelná vzhledem k latencím vznikajícím na síti. Navíc snadnější ladění FUSE oproti jadernému modulu 2.3.1 je nezpochybnitelné.

2.4 Filosofie FUSE

FUSE se způsobem implementace velmi podobá jadernému ovladači – stačí implementovat předem dané rozhraní. Ladění ovladače pod FUSE může být provedeno standardními nástroji, jimiž jsou *gdb* a *valgrind*. Navíc knihovna *libfuse* podporuje ladící mód, kdy se zamezí odpojení ovladače od standardního chybového výstupu, na který jsou následně vypisovány prováděné operace.



Obrázek 2.3: Grafické znázornění změny běhového režimu [17]

Ovladač vlastního souborového systému pro `FUSE` by měl implementovat rozhraní dané strukturou `fuse_operations`. Implementace funkcí je obecně volitelná, pouze některé funkce jsou nezbytné k funkčnosti souborového systému. Například souborový systém, který je pouze pro čtení nemusí mít implementovánu funkci pro zápis.

Na obrázku 2.3 je zobrazen průběh volání z uživatelské aplikace přes VFS vrstvu, FUSE ovladač až po ovladač specifického souborového systému. Během toho je několikrát změněn běhový režim z uživatelského na jaderný a zpět.

```
struct fuse_operations kivfs_operations = {
  /* získávání atributů (atime, mtime, ugo) */
  .getattr  = kivfs_getattr,
  .fgetattr = kivfs_fgetattr,

  /* čtení adresáře */
  .readdir  = kivfs_readdir,

  /* otevření a zavření souboru */
  .open   = kivfs_open,
  .release  = kivfs_release,

  /* čtení a zápis souboru */
  .read   = kivfs_read,
  .write    = kivfs_write,

  /* test přístupových práv ugo */
  .access   = kivfs_access,

  /* vytvoření souboru a složky */
  .create   = kivfs_create,
  .mkdir    = kivfs_mkdir,

  /* alokace prostoru pro soubor (pouze lokálně) */
  .truncate = kivfs_truncate,
  .ftruncate  = kivfs_ftruncate,

  /* přejmenování položky */
  .rename   = kivfs_rename,

  /* zámek souboru (pouze lokálně) */
  .lock   = kivfs_lock,

  /* smazání souboru a složky */
  .unlink   = kivfs_unlink,
  .rmdir    = kivfs_rmdir,

  /* aktualizace času přístupu a modifikace */
  .utimens  = kivfs_utimens,

  /* změna ugo */
  .chmod    = kivfs_chmod,

  /* synchronizace s fs */
  .fsync    = kivfs_fsync,

  /* alokace a dealokace zdrojů */
  .init   = kivfs_init,
  .destroy  = kivfs_destroy,
};
```

Implementací jednotlivých funkcí se zabývá praktická část bakalářské práce.

2.5 Cachování

Cachování je v praxi velmi používaná technika, jak zamezit především zbytečnému přenosu dat z pomalých zdrojů a médií a tím ušetřit čas a výkon těchto zdrojů. Cache se vyznačuje dvěma parametry, které určují její efektivitu.

2.5.1 Velikost cache

Prvním z nich je velikost vyhrazeného prostoru pro cachovaná data. Je logické, že prostornější cache dokáže pojmout více dat. Na druhou stranu ale nesmí být příliš veliká, jinak by se mohlo stát, že cache se promění v mirror pomalejšího média. Nesmí být také příliš velká vůči celkovému úložnému prostoru na cílovém zařízení. Bylo by krajně nevhodné, aby se například na mobilním telefonu s kapacitou 8 GiB alokovala cache o velikosti 4 GiB.

2.5.2 Algoritmus uvolňování cache

Druhým parametrem je algoritmus, který zajišťuje výběr potenciálně nejvýhodnějšího kandidáta na odstranění. Mazání souborů z cache je důležité z důvodu omezeného úložného prostoru. Nejprimitivnějšími algoritmy jsou náhodný výběr a FIFO. Jejich efektivita není příliš veliká a mají tendenci vyhazovat z cache i často používané soubory.

Oproti tomu existují o něco chytřejší algoritmy – LRU, LFU a LFU-SS. V praktické části budou popsány mnou navržené algoritmy QLFU-SS a WLFU-SS, které z těchto algoritmů vycházejí. V neposlední řadě budou také s těmito algoritmy porovnány.

LRU [2]

Algoritmus LRU je z hlediska implementace velmi jednoduchý. Jeho princip spočívá v aktualizaci času přístupu k souboru. Pokaždé, když má být uvolněn prostor v cache, je vybrán soubor s nejnižší hodnotou časového razítka. Asymptotická složitost výběru kandidáta z databáze je tedy $O(N)$.

Další možností, jak implementovat, LRU je datová struktura fronta. Při přístupu souboru se tento soubor přesune ze své pozice na konec fronty. V tomto případě je výběr kandidáta $O(1)$, ale přístup k souboru $O(N)$ vzhledem k počtu souborů v cache.

LFU [2]

LFU na rozdíl od LRU aktualizuje čítač u přistupovaného souboru. Soubor, jehož počítadlo nabývá nejnižší hodnoty je posléze v případě potřeby odstraněn. LFU existuje i ve variantě s redukcí čítače, kdy se hodnota čítačů u všech souborů vydělí definovanou konstantou. Tím se eliminuje historický vliv dříve hojně přistupovaných souborů, které již nejsou využívány.

LFU-SS [2]

LFU-SS je v principu shodný s LFU, navíc ale používá globální statistiky ze serveru. Tento přístup je výhodný v pokrytí času, kdy se lokální cache teprve formuje a neexistují tedy spolehlivá měřítka, která by dokázala výhodně vybírat kandidáty na odstranění.

Při vkládání souboru do cache je předem spočtena počáteční metrika dle vzorečku:

$$metric = \frac{readhits_{server} - writehits_{server}}{globalhits_{server}} \cdot globalhits_{client} + 1 \tag{2.1}$$

Pokaždé, kdy hodnota metriky přesáhne zadanou mez, je hodnota metriky vydělena dvěma stejně jako u LFU s redukcí, aby se zabránilo stárnutí statistik.

Je tedy zřejmé, že výpočetní náročnost LFU-SS bude jistým dílem vyšší než u prostého LFU.

2.6 Přednačítání

Přednačítání souborů jde v určitém pohledu proti cachování. Cachování se snaží o nižší vytížení pomalých či vzdálených zdrojů, přednačítání na to jde z druhé strany a nebere ohledy na zdroje.

Jak cachování, tak přednačítání mají za cíl urychlit práci uživatele a je možné oba přístupy kombinovat.

Přednačítání na základě definovaných heuristik (např. v nově otevřeném adresáři je vyšší šance, že bude otevřen i nějaký soubor) může automaticky stahovat kandidáty souborů, které by mohly být využity v blízké budoucnosti. Mimo jiné vedlejším efektem bude znatelně větší obměna souborů a jejich čítačů a tím i hypoteticky přesnější statistiky pro LFU-SS.

2.7 Šifrování dat

Šifrování dat je v dnešní době nedílnou součástí práce s počítačem. Asi málokomu by bylo příjemné, kdyby jeho osobní data mohl číst kdokoli. Z pohledu korporací je důležité udržet si své know-how a chránit se před nežádoucím únikem interních dat. K tomuto účelu existuje mechanismus *triple A*. Zkratka AAA pochází ze tří slov: Autentizace – ověření, zda ten, kdo se vydává za určitou identitu je skutečně on, Autorizace – ověření zda autentizovaný uživatel má právo k danému úkonu a Audit – záznam akcí pro případné dohledání místa úniku či jako prostředek k obnovení původního stavu dat v případě poškození hardwarových úložišť nebo selhání softwaru.

Ochranu dat před zneužitím neoprávněnou osobou šifrováním lze provádět především na dvou místech. Oba způsoby budou podrobněji rozebrány v následujících odstavcích a budou zmíněny jejich výhody či nevýhody.

2.7.1 Šifrování na straně serveru

Velkou předností šifrování dat na serveru je usnadnění implementace klientských aplikací, jež se nemusí starat o proces šifrování. Klientská zařízení jsou méně zatěžována a v případě mobilních zařízení menší zátěž většinou znamená i větší výdrž při provozu na baterie.

Další výhodou z pohledu správců distribuovaného úložiště je schopnost rozpoznat soubory se shodným obsahem a uchovávat pouze jednu kopii, což značně šetří datovým prostorem.

Mezi nevýhody šifrování dat na serverech patří zvýšená zátěž CPU i přes hardwarovou podporu šifrování v moderních cpu. Při kompromitaci šifrovacích klíčů jsou také kompromitována veškerá data zašifrovaná danými klíči.

2.7.2 Šifrování na straně klienta

Šifrování dat již v klientských aplikacích především rozkládá výkon potřebný k šifrování na více zařízení a uvolňuje tak prostředky serverů. Uživatel má navíc plně pod kontrolou svůj šifrovací klíč a jeho data tak jsou zabezpečena i před neoprávněným přístupem administrátorů.

Daní za zvýšenou bezpečnost je ztížené sdílení dat mezi uživateli. Uživatelé by si mezi sebou museli rozdistribuovat šifrovací klíče a klientská aplikace by musela mít implementovánu podporu pro více klíčů.

3 Praktická část

Tato kapitola bude pojednávat o implementačních problémech a úspěšnosti jejich řešení.

3.1 Struktura zdrojových souborů

Zdrojové soubory jsou členěny do několika okruhů. Soubory obsahující v názvu „operations“ obstarávají logiku souborového systému. Správu metadat a lokálních souborů obstarávají soubory s prefixem „cache“.

3.2 Zobrazení metadat

Uživatel používá souborový systém především k ukládání dat, která chce také většinou ze souborového systému zpětně přečíst. Aby toto bylo možné, musí být implementována funkce `int kivfs_readdir(const char *path, void *buf, fuse_fill_dir_t filler, off_t offset, struct fuse_file_info *fi)`, jež má na starosti naplnění bufferu `buf` obslužnou funkcí předanou odkazem `filler`.

Ke každému souboru se váží přístupová práva, která je také nutno zobrazit. K tomuto účelu jsou předurčeny funkce `int kivfs_getattr(const char *path, struct stat *stbuf)` a `int kivfs_fgetattr(const char *path, struct stat *stbuf, struct fuse_file_info *fi)`.

VSF nebo logika FUSE volí vždy jednu z nich. Oběma funkcím je předán odkaz na řetězec obsahující cestu k souboru, jehož informace mají být zobrazeny. Jeho použití je ale v případě funkce `fgetattr` kontraproduktivní a mělo by být omezeno pouze na výpis při ladění ovladače. Cesta k souboru se totiž musí postupně zpracovat VFS vrstvou, která určí, na kterém souborovém systému soubor leží a opětovně zavolá funkci `stat` náležící konkrétnímu souborovému systému. Funkce `fgetattr` má navíc totiž odkaz na strukturu `fuse_file_info` a v ní uložený deskriptor souboru. Jeho použitím se obejde časově náročnější zjišťování, který ovladač obslouží načtení metadat souboru, ale rovnou se načtou využitím deskriptoru ve funkci `fstat`.

Některé aplikace testují přístupová práva zvlášť bez ohledu na informace získané při volání `getattr` nebo `fgetattr`. K plnění tohoto úkolu je předurčena funkce `int kivfs_access( const char *path, int mask)`, která se řídí kontraktem systémového volání `access`.

3.3 Běžná práce se soubory

Za běžnou práci se soubory je považováno kopírování, čtení a zápis a s tím související otevření a zavření souboru. Tyto úkony budou rozebrány v následujících kapitolách.

3.3.1 Otevření souboru

Když jsou všechna potřebná metadata zobrazena uživateli, může být zahájena práce se soubory samotnými. Obecně je nejprve nutné soubor otevřít. Otevírání je z uživatelského pohledu provedeno systémovým voláním `open`, které projde VFS vrstvou až k FUSE, které na základě módu otevření předá obsluhu funkci `int kivfs_create(const char *path, mode_t mode, struct fuse_file_info *fi` nebo `int kivfs_open(const char *path, struct fuse_file_info *fi)`. Funkce `kivfs_create` je vybrána, pokud je nastaven příznak `O_CREAT` při volání `open`.

Obě funkce alokují prostor pro strukturu `kivfs_ofile_t` obsahující především lokální a vzdálený deskriptor souboru s přípojením na `fs` vrstvu KIVFS. Naplnění struktury bude více rozebráno v sekci Cachování 3.5. Při každém otevření souboru je aktualizován počet přístupů v režimu pro čtení a pro zápis.

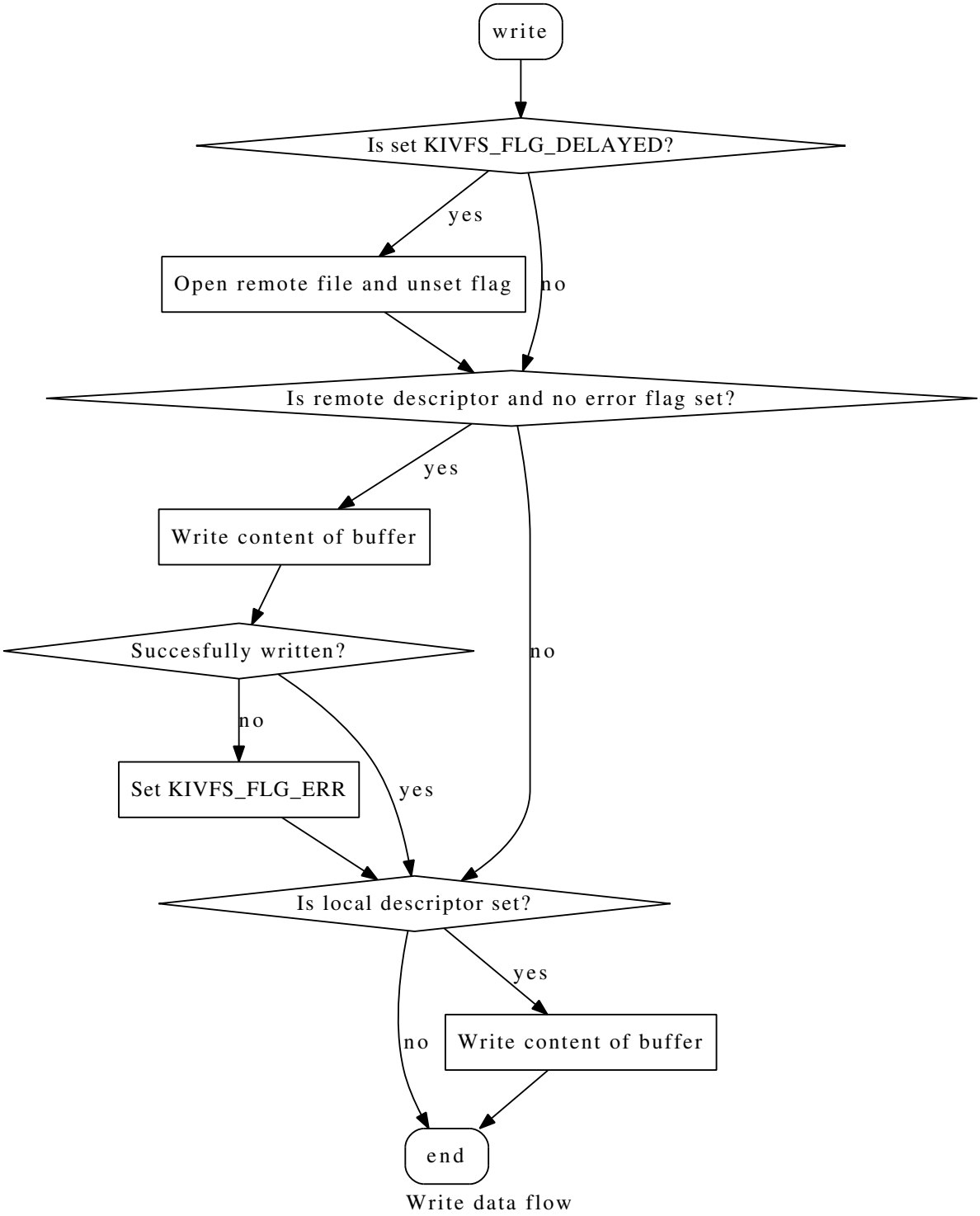
3.3.2 Čtení ze souboru

O realizaci čtení dat ze souboru se stará funkce `int kivfs_read(const char *path, char *buf, size_t size, off_t offset, struct fuse_file_info *fi)`. Jejím úkolem je dopravit data z vlastního souborového systému do bufferu `buf` o velikosti `size` s počátkem v `offset`. Funkce čtení v případě použití cache obstarává i zápis aktuální verze souboru ze vzdáleného úložiště do cache, viz 3.5.

3.3.3 Zápis do souboru

Zápis do souboru obstarává funkce `int kivfs_write(const char *path, const char *buf, size_t size, off_t offset, struct fuse_file_info *fi)`, které jsou předány deskriptory obsažené ve struktuře `fi`, buffer `buf` obsahující data určená k zapsání, velikost bufferu `size` a pozice v souboru `offset` viz obrázek 3.1.

Před samotným zápisem je zkontrolován příznak `KIVFS_FLG_DELAYED`, který značí, že soubor by měl být otevřen až při zápisu dat. Příznak `KIVFS_FLG_ERR` znamená, že při přenosu nastala chyba sítě, a proto se nepokouší zapisovat do vzdáleného souboru. Při uzavírání souboru se kontroluje, zda byl soubor úspěšně zapsán, nebo zda se má synchronizovat po obnově připojení.



Obrázek 3.1: Postup při zápisu souboru

3.3.4 Zavření souboru

O uzavírání se stará funkce `int kivfs_release(const char *path, struct fuse_file_info *fi)`. Každý otevřený soubor obsahuje deskriptor ve struktuře `kivfs_ofile_t`, již lze získat přes ukazatel `fh` struktury `fuse_file_info`. Po úspěšném uzavření jsou aktualizovány údaje v databázi, případně nastaven příznak modifikace změněného souboru.

3.3.5 Změna přístupových práv

Přístupová práva jsou řešena klasickým způsobem: uživatel – skupina – ostatní. Každá položka může obsahovat právo na zápis, čtení či spouštění. Požadavek na změnu oprávnění je nejprve proveden na vzdáleném serveru, nepovede-li se tato akce, je proveden zápis do logu s informacemi nutnými pro následnou synchronizaci.

3.4 Struktura databáze

Databázové služby poskytuje knihovna *sqlite3*. Databáze samotná se nachází ve třech souborech kvůli využití modu *Write Ahead*, který vykazuje vyšší výkonnost při paralelních operacích [18], a je určena především k uchování metadat souborů, aby mohla být později zobrazena při nedostupnosti připojení.

Databáze obsahuje dvě tabulky – *log* a *files*. Jejich struktura je zobrazena na následující straně. Primárním klíčem v tabulce *files* i *log* je relativní cesta k souboru v souborovém systému KIVFS. Atributy `atime` a `mtime` uchovávají celé číslo, symbolizující čas přístupu a čas modifikace. Atribut `mode` je důležitý ke správnému zobrazení přístupových práv a typu souboru v Linuxu. Sloupce `own` a `grp` slouží pouze k ladícímu výpisu vlastníka a skupiny souboru v KIVFS.

```
CREATE TABLE 'main'.'files' (
  -- cesta k souboru či složce
  path       TEXT PRIMARY KEY,
  -- velikost souboru
  size       INT NOT NULL,
  -- čas modifikace
  mtime        INT NOT NULL,
  -- čas přístupu
  atime        INT NOT NULL,
  -- přístupová práva a typ souboru
  mode       INT NOT NULL,
  -- název vlastníka v KIVFS
  own        TEXT NOT NULL,
  -- název skupiny v KIVFS
  grp        TEXT NOT NULL,
  -- lokální počet přístupů pro tení
  read_hits      REAL DEFAULT ('0.0'),
  -- lokální počet přístupů pro zápis
  write_hits     INT DEFAULT ('0'),
  -- počet přístupu pro čtení na serveru
  srv_read_hits    INT DEFAULT ('0'),
  -- počet přístupů pro zápis na serveru
  srv_write_hits   INT DEFAULT ('0'),
  -- indikace přítomnosti souboru v cache
  cached       INT DEFAULT ('0'),
  -- indikace změny souboru
  modified     INT DEFAULT ('0'),
  -- poslední známá verze souboru na serveru
  version      INT NOT NULL)
```

Sloupce `path`, `size`, `mtime`, `atime` a `mode` jsou využívány především při volání funkce `kivfs_getattr`. Sloupce `read_hits` a `write_hits` slouží k lokálnímu počítání přístupů k jednotlivým souborům. Stejnojmenné položky s prefixem `srv_` obsahují hodnoty ze strany serveru, z nichž se pak počítá prvotní hodnota `read_hits` při vkládání souboru do cache.

Sloupce `cached`, `modified` a `version` jsou využívány při výpočtu kandidátů k výběru z cache a k synchronizaci se serverem.

```
CREATE TABLE 'main'.'log' (
  path     TEXT PRIMARY KEY,
  new_path   TEXT UNIQUE,
  action     INT NOT NULL)
```

V tabulce *log* jsou pouze tři sloupce. Sloupec `path` slouží k identifikaci objektu, kterého se změna týká. Druhý sloupec může obsahovat novou cestu, byl-li objekt přejmenován či přesunut, nebo nová uživatelská oprávnění. Sloupec `action` uchovává identifikátor provedené akce, jež se pak při synchronizaci provede.

3.5 Cachování

Cachování tvoří nedílnou součást klientské aplikace, a proto bude také dále blíže rozebráno.

3.5.1 Otevření souboru

Jelikož jsou i dnešní velmi rychlé sítě ve srovnání s lokálními pevnými disky pomalé a mají velkou latenci, je vhodné zabudovat do klientské aplikace cache, která uchovává soubory, jež jsou svou metrikou výhodné k uložení. Ještě před samotným otevřením souboru je zkontrolována verze vzdáleného souboru. Na základě porovnání vzdálené verze s lokální se volí některý z následujících režimů otevření. Vývojový diagram funkce zajišťující otevření souboru je vidět na obr. 3.2.

Otevření aktuálního souboru

Je-li na straně serveru shodná verze souboru jako v cache a je-li režim otevření souboru pouze pro čtení, pak je pouze otevřena lokální kopie. Prostor pro vzdálený deskriptor ve struktuře `kivfs_ofile_t` zůstane inicializován na počáteční hodnotu, kterou funkce čtení ošetřuje.

Otevření novějšího souboru na serveru

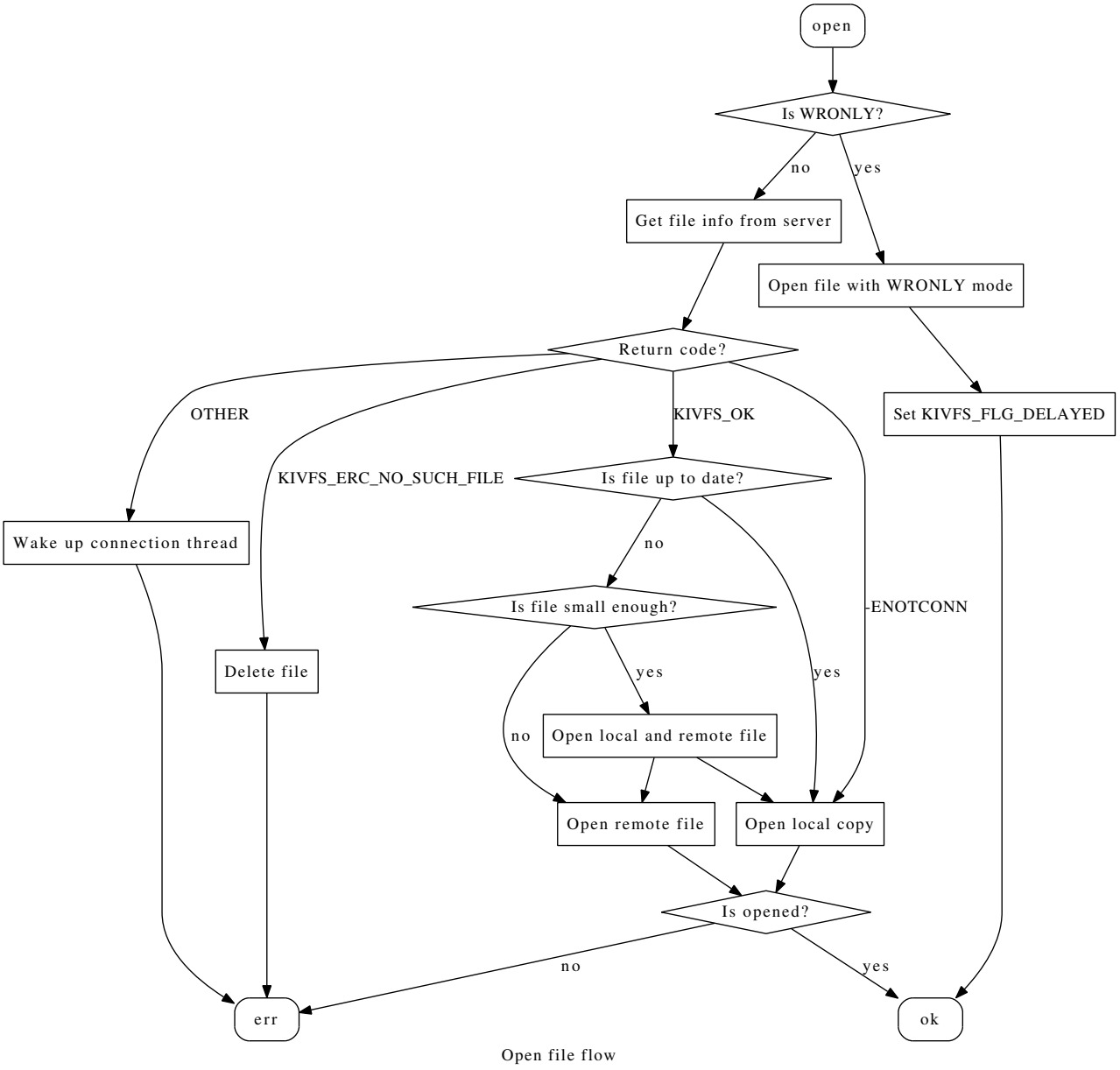
Nachází-li se na serveru novější verze souboru, je vzdálený soubor otevřen dle parametrem získaného módu otevření a lokální kopie otevřena pro čtení a zápis. V databázi jsou aktualizovány údaje pro okamžité použití. S každým zavoláním `read` je načteno jeden nebo více bloků vzdáleného souboru do lokálního bufferu. Ještě před návratem z funkce je buffer nakopírován do cachovaného souboru.

Je-li soubor otevírán v režimu pouze pro zápis, není soubor otevřen ihned. To by totiž způsobilo jeho zkrácení na nulovou délku na straně serveru, který počítá s režimem upload/download a nepřijdou-li žádná data, je soubor považován za prázdný. Některé aplikace ale otevřením a zavřením souboru v režimu pro zápis testují přístupová práva. Proto byl implementován mechanismus, který se snaží zabránit ztrátě dat.

V režimu pouze pro zápis je jen nastaven příznak `KIVFS_FLG_DELAYED`, který se kontroluje až při samotném zápisu. Pokud je nastaven příznak `KIVFS_FLG_DELAYED`, je soubor otevřen až v zápisové funkci.

3.5.2 Výběr souborů při zaplnění cache

Dojde-li k zaplnění cache, je nutné vybrat soubory, které nebudou potřeba v nejbližší době, jelikož právě ty mohou být odstraněny s nejvyšším užitkem. Existuje několik algoritmů výběru, které byly blíže zmíněny v kapitole 2.5.



Obrázek 3.2: Postup při otevírání souboru

WLFU-SS

Nový experimentální algoritmus WLFU-SS je modifikací algoritmu LFU-SS. Při vkládání souboru do cache se vypočte jeho metrika na základě vzorečku (2.1). Na rozdíl od LFU-SS se při výběru počítá vážená metrika dle následujícího vzorečku:

$$metric = metric \cdot (1 - \frac{filesize}{max(filesize_i)}) \tag{3.1}$$

Je zřejmé, že malé soubory budou mít metriku vyšší a spíše se z cache vyhodí velký soubor. Dle měření na reálných datech z kapitoly 3.8 je WLFU-SS s tímto vzorečkem ve většině případů efektivnější. Ovšem ve specifických situacích by mohlo nastat „uváznutí“ velmi malých souborů v cache. Tomuto zabrání upravený vzoreček (3.2), který mění metriku maximálně o jednotku.

$$metric = metric - \frac{filesize}{max(filesize_i)} \tag{3.2}$$

K tvorbě tohoto algoritmu mne vedla úvaha nad tím, jakým způsobem ovlivňují velké soubory hit ratio a přenesená data.

Nechť existují dva soubory, jeden o velikosti 10 a druhý o velikosti 1. Který soubor by měl uvolnit místo pro další? Byl-li by vyhozen velký a vzápětí byl-li by přistupován znovu, muselo by být staženo až deset jednotek. Kdežto malý soubor může být vyhozen až desetkrát a až poté bude datový přenos ztrátový[1].

QLFU-SS

QLFU-SS je opět modifikací LFU-SS. Rozdíl mezi těmito algoritmy je ve výpočtu počáteční metriky – QLFU-SS používá ve všech případech jednotku – a ve výběru kandidátů na výběr z cache. QLFU-SS používá vzoreček (2.1) až při výběru. Do výpočtu jsou tedy zahrnuty i lokální statistiky naposledy přidaného souboru.

X varianty

Algoritmy končící písmenem X mají navíc ještě jednu vlastnost. Při vkládání souboru do cache nebude vložen soubor o velikosti větší než je jedna polovina velikosti celé cache. Zabraňuje se tak nechtěnému vyprázdnění celé poloviny kapacity.

[1] Původně mělo být ve vzorečku znaménko +, ale nakonec se měřením na reálných datech ukázalo, že je výhodnější odstraňovat velké soubory.

3.5.3 Dynamická velikost cache

Většinu času je možné využívat statickou velikost cache definovanou v nastavení programu. Ovšem při výpadku připojení by nebylo možné tvořit nové soubory, které by se do cache za běžných okolností nevešly. Proto je v offline režimu umožněno vytváření souborů bez ohledu na zaplnění cache.

Po obnovení připojení se obsah nově vytvořených souborů nejprve sesynchronizuje se serverem a až pak je povolena práce se souborovým systémem. Synchronizované soubory jsou v cache ponechány až do doby, kdy je stahován soubor, který v cache není. Pak zafunguje standardní výběr kandidátů výběru nepotřebných souborů.

3.6 Synchronizace

Důležitou součástí aplikace je schopnost synchronizovat lokální změny provedené bez připojení k internetu. V offline režimu jsou logovány do tabulky *log*, která je blíže popsána v sekci 3.4. Při zapnutí klientské aplikace, či při navázání spojení se nejprve synchronizují veškeré lokální změny, je li to možné. Pokud se během doby, kdy aplikace nebyla připojena změnila verze vzdáleného souboru, lokální změny se na serveru neprojeví a při přístupu bude naopak lokální soubor přepsán vzdáleným, který je novější.

Tento přístup ale nemusí být vhodný pro každého, a proto by budoucí verze klienta mohla být vylepšena o automatické přejmenování staršího souboru, či jiný mechanismus, který zamezí nechtěnému přepsání upraveného souboru.

Navíc se klient snaží minimalizovat počet požadavků na server tím, že požadavky, jež jsou protichůdné, buď odstraňuje nebo je slučuje do jediného. Problematické je přejmenovaní souboru a opětovné přejmenování zpět ještě před synchronizací, kdy se provede pouze první přejmenování.

3.7 Obnovení spojení

O obnovu spojení se stará vyčleněné vlákno, jež při zapnutí klientské aplikace naváže spojení, nastaví příznak aktivního spojení a pak se samo uspí nad objektem monitoru. Nastane-li během chodu aplikace chyba týkající se ztráty spojení, je vynulován příznak aktivní konektivity a uspané vlákno vzbuzeno. To se pak v postupně se zvětšujících intervalech pokouší obnovit spojení. Počáteční interval je nastaven na jednu sekundu a s každým neúspěšným pokusem se zdvojnásobí. Až samovolným přetečením šestnáctibitové hodnoty dojde opět ke zkrácení intervalu pokusů.

3.8 Měření výkonnosti

Kapitola měření výkonnosti se zaměří na porovnání nových algoritmů se stávajícími na skutečných i syntetických datech. V další části budou představeny a zhodnoceny naměřené přenosové rychlosti souborů.

3.8.1 Efektivita cache

Cílem měření efektivity cache je zjistit, jaký algoritmus dokáže ušetřit nejvíce dat, a zda existuje závislot mezi hit ratio a ušetřenými daty. Měření bylo prováděno v aplikaci CacheSimulatorV2 [19], jež byla vyvinuta v rámci projektu KIVFS právě kvůli porovnání algoritmů.

Náhodná data byla generována s gaussovským rozdělením N(5000,100), které je nastaveno jako výchozí v aplikaci CacheSimulator. V tabulce č. 3.1 jsou hodnoty z měření náhodného gaussovského přístupu k souborům s náhodnými velikostmi v rozmezí 1 MiB až 5 GiB pro různé velikosti cache. Tyto velikosti byly zvoleny tak, aby do cache mohl být stažen velký soubor, jež by způsobil odstranění mnoha jiných souborů v cache obsažených. Také tím byla testována efektivita přístupu, kdy se do cache povoluje stažení souborů až do velikosti poloviny celé cache.

Reálná data byla získána z AFS logů, z nichž byl vybrán uživatel, u kterého se výsledky jednotlivých algoritmů více lišily.

Algoritmus WLFU-SS je zde ve variantě 8, která používá vzoreček (3.1), a ve variantě 10 (3.2). Zeleně vyznačené hodnoty jsou pro danou velikost cache nejlepší, žluté jsou druhé nejlepší.

Hodnota hit ratio představuje procento přístupů, které vedly do cache. Výpočet hit ratio je definován vzorečkem (3.3). Jedná se o prostý poměr počtu požadavků, které byly obslouženy z cache ku celkovému počtu požadavků. Podobným způsobem je vypočtena hodnota ušetřených dat – velikost dat získaná z cache ku celkové velikosti všech požadavků.

$$hitratio = \frac{hitcount}{hitcount + misscount} \tag{3.3}$$

Tabulka 3.1: Hit ratio v % pro náhodná data s gaussovským rozdělením (5000,100)

Caching policy	8MB	16MB	32MB	64MB	128MB	256MB	512MB	1024MB
WLFU-SS8	1.148	1.351	1.493	2.994	6.945	11.428	16.491	26.056
WLFU-SS8X	1.647	2.168	3.02	4.518	6.945	11.428	16.491	26.056
WLFU-SS10	1.094	1.264	1.425	3.081	7.144	11.84	17.769	26.588
WLFU-SS10X	1.936	2.44	3.235	4.606	7.144	11.84	17.769	26.588
LFU-SS	0.636	0.597	0.827	2.023	4.376	7.498	12.954	22.367
QLFU-SS	0.602	0.706	1.005	2.666	6.498	9.084	16.547	24.809
QLFU-SSX	2.007	1.339	3.422	3.575	6.498	9.084	16.547	24.809
LFU Reduction	0.535	0.505	0.736	1.434	3.37	7.781	12.056	18.203
Standard LFU	0.535	0.505	0.736	1.613	2.337	4.754	8.34	15.037

Tabulka 3.2: Ušetřená data v % pro náhodná data s gaussovským rozdělením (5000,100)

Caching policy	8MB	16MB	32MB	64MB	128MB	256MB	512MB	1024MB
WLFU-SS8	0.039	0.092	0.214	0.479	0.864	1.686	3.044	6.762
WLFU-SS8X	0.048	0.103	0.203	0.429	0.864	1.686	3.044	6.762
WLFU-SS10	0.04	0.094	0.216	0.485	0.897	1.774	3.472	7.024
WLFU-SS10X	0.058	0.114	0.216	0.442	0.897	1.774	3.472	7.024
LFU-SS	0.047	0.108	0.23	0.659	1.684	3.815	7.953	15.955
QLFU-SS	0.071	0.142	0.277	0.653	1.629	3.733	7.688	15.551
QLFU-SSX	0.089	0.205	0.383	0.791	1.629	3.733	7.688	15.551
LFU Reduction	0.046	0.104	0.216	0.63	1.613	3.653	7.635	15.547
Standard LFU	0.046	0.104	0.216	0.628	1.636	3.437	6.91	14.352

S náhodným přístupem si nejlépe poradil pro malé velikosti cache algoritmus QLFU-SSX a to jak v hit ratio, tak ve velikosti ušetřených dat. Jde ovšem o syntetický test a musí být brán s rezervou.

Z tabulek 3.3, 3.4 a grafů 3.3, 3.4 si lze na první pohled všimnout, že WLFU-SS má výrazně vyšší hit ratio než ostatní algoritmy pro menší velikosti cache. Nicméně velikost ušetřených dat je u všech algoritmů téměř shodná a každý vykazoval lepší hodnoty pro jinou velikost.

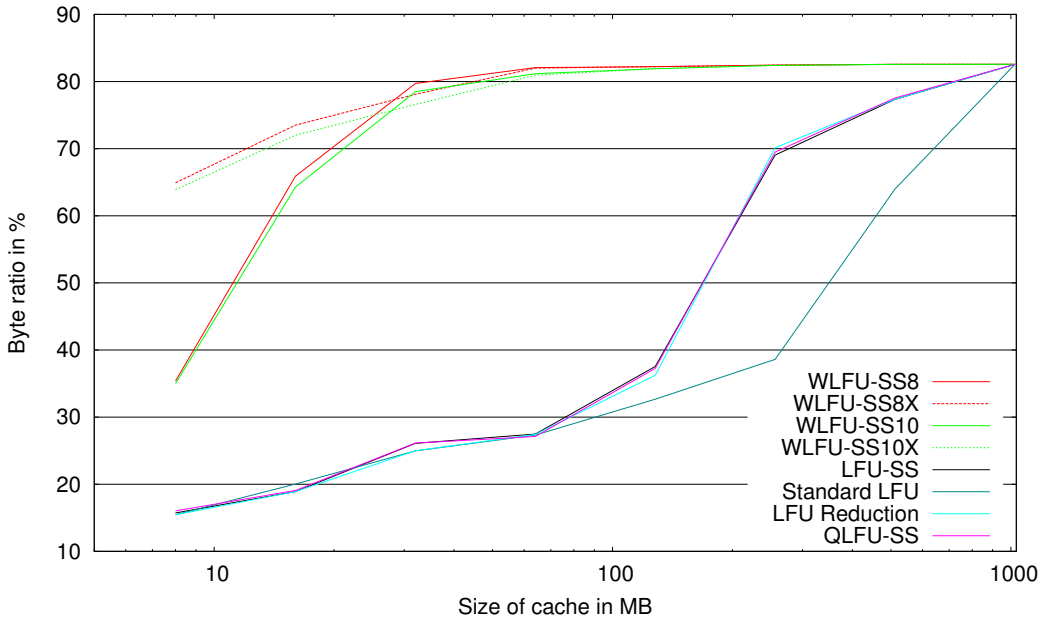
Pro reálná data se v tomto případě algoritmy s X staly nevýhodné. Ačkoliv měly přibližně dvojnásobné hit ratio, velikost ušetřených dat byla méně než poloviční. Není tedy vhodné hodnotit algoritmy podle hit ratio, je-li pro daný účel důležitý počet ušetřených dat. Vyšší hit ratio by naopak bylo výhodnější v prostředí s vysokou latencí, kde se musí dlouho čekat na navázání spojení a přenos samotný už probíhá přijatelnou rychlostí.

Tabulka 3.3: Hit ratio v % pro reálná data

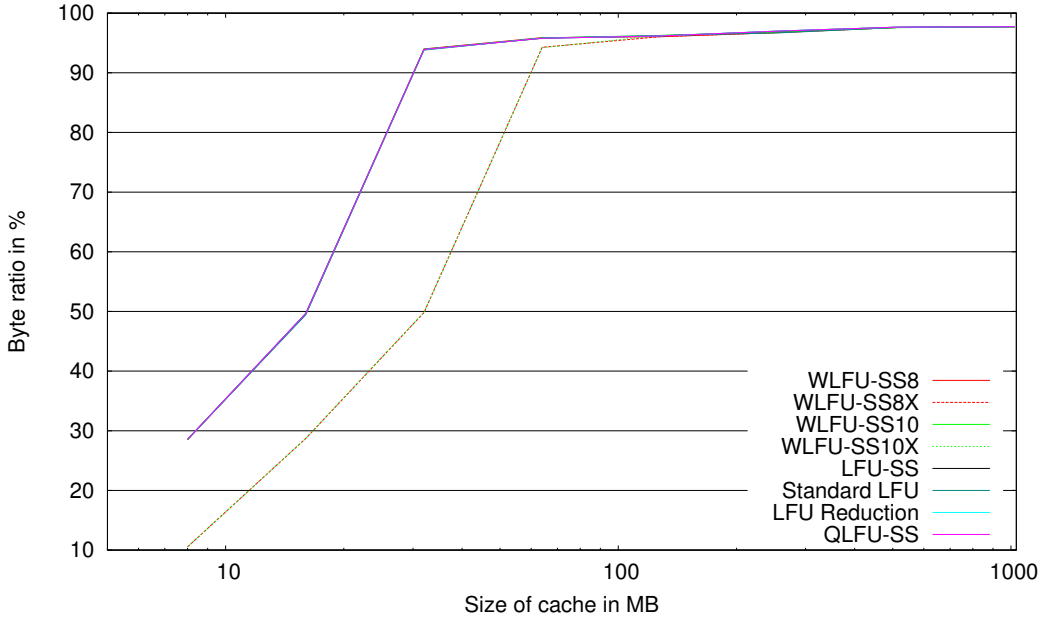
Caching policy	8MB	16MB	32MB	,64MB	128MB	256MB	512MB	1024MB
WLFU-SS8	35.385	65.896	79.711	82.078	82.219	82.444	82.564	82.571
WLFU-SS8X	64.91	73.511	78.097	82.0	82.219	82.444	82.564	82.571
WLFU-SS10	35.005	64.276	78.485	81.169	81.916	82.402	82.557	82.571
WLFU-SS10X	63.888	71.997	76.59	80.936	81.916	82.402	82.557	82.571
LFU-SS	15.73	18.908	26.1	27.481	37.548	69.045	77.351	82.571
LFU Reduction	15.449	18.908	25.001	27.418	36.238	70.144	77.365	82.571
QLFU-SS	16.033	19.07	26.135	27.115	37.266	69.489	77.555	82.571
QLFU-SSX	9.108	16.477	18.901	25.015	37.266	69.489	77.555	82.571

Tabulka 3.4: Ušetřená data v % pro reálná data

Caching policy	8MB	16MB	32MB	64MB	128MB	256MB	512MB	1024MB
WLFU-SS8	28.533	49.575	93.999	95.899	96.051	96.741	97.593	97.728
WLFU-SS8X	10.552	28.719	49.818	94.27	96.051	96.741	97.593	97.728
WLFU-SS10	28.519	49.626	93.957	95.833	96.258	96.849	97.563	97.728
WLFU-SS10X	10.548	28.712	49.852	94.213	96.258	96.849	97.563	97.728
LFU-SS	28.565	49.67	93.913	95.784	96.176	96.992	97.65	97.728
Standard LFU	28.563	49.44	93.802	95.832	96.236	96.693	97.594	97.728
LFU Reduction	28.563	49.67	93.91	95.796	96.175	97.008	97.643	97.728
QLFU-SS	28.565	49.678	93.911	95.787	96.176	96.984	97.655	97.728
QLFU-SSX	10.518	28.943	49.742	94.179	96.176	96.984	97.655	97.728



Obrázek 3.3: Graf zobrazující hit ratio v % v závislosti na velikosti cache



Obrázek 3.4: Graf zobrazující ušetřená data v % v závislosti na velikosti cache

Z naměřených dat nelze říci, že je nějaký algoritmus vhodný na vše. Každý má své přednosti v jiných situacích. Některý je lepší při náhodném přístupu, jiný spíše v predikovatelném opakování přístupů atd. Při volbě algoritmu, který by měl být nasazen do cache by mělo být zváženo, jaká data budou převažovat a podle toho volit algoritmus, který má v daném oboru silnější stránky.

Nejvýhodnějším algoritmem pro malé cache se zdá být WLFU-SS, který dosahuje vyššího hit ratio než LFU-SS při srovnatelné velikosti ušetřených dat. Z uživatelského hlediska by měl poskytovat snížení doby odezvy aplikace.

3.8.2 Praktický test klienta

Testování klienta bylo prováděno na fyzickém počítači. Servery byly virtualizovány ve Virtualboxu. Databáze i cache se nacházely v adresáři /tmp, do kterého byl připojen tmpfs souborový systém.

Cílem měření bylo zjistit, s jakou rychlostí mohou být data přenášena a jaký vliv na rychlost přenosu má velikost jednotlivých souborů.

Klientský počítač	
CPU	Intel i5 3.1 GHz
RAM	16 GiB
Síť	1 Gbps
OS	Linux Mint Debian Edition 64bit
Jádro	3.13-1
FUSE	2.9.3-9

Serverový počítač	
Virtualbox	4.3
CPU	Intel i5 3.1 GHz (virtual)
RAM	8 GiB
Síť	1 Gbps
OS	Debian Wheezy 64bit
Jádro	3.2.54-2

Přenos malých souborů

Přenos malých souborů byl testován stejným způsobem jako v práci [12]. Na server bylo nahráno 100 souborů, každý o velikosti 1 MiB. Měření bylo prováděno skriptem využívajícím program `dd`, jenž vypisuje přenosové rychlosti.

```
2048+0 vstoupivších záznamů
2048+0 vystoupivších záznamů
1 048 576 bajtů (1,0 MB) zkopírováno, 0,00196304 s, 534 MB/s
2048+0 vstoupivších záznamů
2048+0 vystoupivších záznamů
1 048 576 bajtů (1,0 MB) zkopírováno, 0,00149804 s, 700 MB/s
...
```

Rychlost čtení souborů z cache je omezena samotným hardwarem (v tomto případě RAM pamětí) a režií FUSE. Čtení souborů, které nejsou v cache probíhá přibližně stejně rychle jako čtení a zápis velkého souboru; viz následující odstavec.

Přenos velkého souboru

Velký soubor byl představen jedním záznamem z /dev/zero o velikosti 1 GiB.

```
Write:
1024+0 vstoupivších záznamů
1024+0 vystoupivších záznamů
1 073 741 824bajt (1,1 GB) zkopírováno, 1 487,61s, 722 kB/s

Read:
1024+0 vstoupivších záznamů
1024+0 vystoupivších záznamů
1 073 741 824 bajtů (1,1 GB) zkopírováno, 970,446 s, 1,1 MB/s
```

Zápis probíhá o něco pomaleji než čtení. To se dá vysvětlit tím, že je nutné na straně serveru provádět alokace a synchronizaci replik. Čtení je pak o něco rychlejší, nicméně oboje je podobné rychlosti čtení a zápisu malých souborů. Důvod je prostý – není prováděno shromažďování požadavků do větších celků, ale všechny jsou prováděny okamžitě. Každý požadavek musí být odeslán na server a pak musí být přijato potvrzení požadavku. Systém dokáže na jedno volání zápisu či čtení přenést 128 KiB dat.

Nicméně bufferování v klientské aplikaci nebylo implementováno a to z důvodu, že na tom již pracují vývojáři fuse: „*Maxim Patlasov is currently working on a patchset to enable write buffering for fuse, which will make this issue go away. Miklos*“[20].

3.9 Změny na Core

Nad rámec zadání bakalářské práce jsem implementoval nové komunikační rozhraní do knihovny *libkivfscore*. *Libkivfscore* je knihovna využívaná servery i klienty k síťové komunikaci a kvůli definici společných datových struktur. Jelikož původní integrace OpenSSL do *libkivfscore* by byla obtížně implementovatelná do nových klientských aplikací z důvodu redundance a odlišnosti kódu pro šifrovaný a nešifrovaný přenos.

Nové rozhraní je flexibilnější a umožňuje šifrovaný přenos v praktickém použití i nešifrovaný přenos dat pro ladění. Za flexibilitu se dá označit možnost změny šifrovací knihovny bez větších zásahů do stávajícího kódu. Změnit by bylo potřeba strukturu `kivfs_connection_t`, která aktuálně spojuje socket se SSL nadstavbou.

Dalším přínosem je implementace podpory IPv6[2] a DNS. Uživatelé si většinou pamatují pouze doménová jména a tato funkcionalita je tedy nezbytná pro každou klientskou aplikaci.

[2]Podpora IPv6 funguje alespoň hypoteticky, jelikož nemohla být nijak ověřena.

4 Uživatelská dokumentace

Klient je distribuován ve formě zdrojových souborů, které jsou k dispozici na přiloženém DVD a na adrese `https://forge.kiv.zcu.cz/svn-kivfs`. Před jeho použitím je nutné zdrojové soubory přeložit na binární překladačem **clang**.

4.1 Překlad klienta

V kořenovém adresáři projektu by měl být nejprve spuštěn příkaz `make`[1] `dependents`[2]. Tím byly do systému nainstalovány knihovny potřebné pro samotný překlad klienta. Příkazem `make` je zahájen překlad do adresáře `./bin`.

4.2 Spuštění klienta

Klientský program lze spustit příkazem `kivfs <mount path> --host=fs-2.kiv.zcu.cz`, kde `mount path` je existující prázdný adresář, jenž bude zobrazovat vzdálené soubory.

Seznam všech podporovaných přepínačů:

--ip	ip adresa nebo doménové jméno serveru
--host	alias --ip
--port	cílový port (výchozí 30003)
--size	velikost cache v mebibytech (výchozí: 128 MiB)
--policy	algoritmus výběru souboru (výchozí: lfuss; wlfuss, qlfuss, lru, lfu)

4.3 Ukončení klienta

Příkazem `fusermount -u <mount path>` dojde k odpojení klienta od serveru a uvolnění zabraných zdrojů.

[1] Předpokládá se, že na uživatelském systému je nainstalován program `make`

[2] Uživatel musí mít právo pro instalaci aplikací

5 Závěr

V rámci mé bakalářské práce jsem se seznámil s rozhraním FUSE a praktickým sledem volání funkcí při práci se soubory a adresáři. Také jsem se seznámil s existujícími algoritmy používanými k uvolnění cache.

Na základě těchto znalostí jsem implementoval klientskou aplikaci, která zvládá přístup ke KIVFS a často používané soubory uchovává v cache. Do cache jsem implementoval několik algoritmů výběru souborů pro odstranění včetně mých experimentálních vycházejících z LFU-SS.

Grafické aplikace většinou provádějí přednačítání atributů souborů samy. TUI i aplikace v něm spuštěné žádné přednačítání neprovádí, ačkoli by to v tomto případě bylo výhodné. Často používané soubory by v některé následující verzi klienta mohly být přednačítány a tím i zlepšen subjektivní pocit z plynulosti práce na vzdáleném souborovém systému.

Aktuální verze klienta provádí jen cachování struktury adresářů. Nové adresáře jsou načteny při přístupu a jejich další obnova je naplánována o předem danou časovou konstantu později. Tímto se stává průchod adresářovou strukturou pohodlnější a rychlejší.

Aplikace sama dokáže detekovat výpadek spojení a následně se ho pokouší obnovit. Po navázání spojení jsou synchronizovány provedené změny.

Klient je navíc připraven pro implementaci šifrování přenosu libovolnou knihovnou.

V budoucí verzi klienta by mohla být zvážena implementace bufferu pro odesílání a příjem větších celků souboru za účelem zvýšení propustnosti. Je ale možné, že tato funkcionalita bude zabudována v některé z následujících verzí FUSE. Také by mohla být zabudována podpora synchronizace pouze změněných částí souboru algoritmem, který je využíván programem `rsync`. Podpora by musela být ale zároveň zabudována i do serverů.

Seznam obrázků

Seznam tabulek

Seznam zkratek

AFS Andrew's File System

AVFS A Virtual File System

DNS Domain Name System

FUSE Filesystem in userspace

GUI Graphical User Interface

GVFS Gnome Virtual File System

HW Hardware

IP Internet Protocol

IPv6 Internet Protocol version 6

ISP Internet Service Provider

KIVFS Souborový systém vyvíjený na katedře informatiky Západočeské univerzity

LFU Least Frequently Used

LFU-SS Least Frequently Used with Server Statistics

LRU Least recently Used

LUFS Linux Userland Filesystem

NAT Network Address translation

OS Operační systém

RAM Random Access Memory

TCP Transmission Control Protocol

TUI Terminal User Interface

UDP User Datagram Protocol

VFS Virtual File System

WLFU-SS Weighted Least Frequently Used with Server Statistics

Literatura

[1] SZAKACSITS, Szabolcs. *NTFS-3G home page*. i2013-05-20. [cit. leden 2014]. Dostupné z: `http://sourceforge.net/projects/ntfs-3g/`

[2] BŽOCH, Pavel, Ladislav PĚŠIČKA, Luboš MATĚJKA a Jiří ŠAFAŘÍK *Design and Implementation of a Caching Algorithm Applicable to Mobile Clients* In Informatica, Volume 36, Number 4, 2012, Ljubljana - Slovenia, pp. 369–378, ISSN: 0350-5596 (Print edition), ISSN: 1854-3871 (Web edition), [cit. březen 2014]
Dostupné z: `http://goo.gl/nM0Ttx`

[3] CALBET, Xavier. *User mode and kernel mode* [online]. 2006. [cit. 2013-12-05] Dostupné z: `http://www.freesoftwaremagazine.com/articles/drivers_linux`

[4] JAY SALZMAN, Peter a Ori POMERANTZ. *The Linux Kernel Module Programming Guide* [online]. 2001, 2003-04-04. [cit. 2013-11-07] Dostupné z: `http://www.faqs.org/docs/kernel/index.html`

[5] MSDN. *User mode and kernel mode* [online]. 2013. [cit. 2013-09-18] `http://msdn.microsoft.com/en-us/library/windows/hardware/ff554836(v=vs.85).aspx`

[6] LANGDALE, Philip. *PIfs: Never worry about data again!* [online]. 2011. [cit. 2013-10-09] Dostupné z: `https://github.com/philipl/pifs`

[7] KUENNING, Geoff. *CS135 FUSE Documentation* [online]. 2010. [cit. 2013-09-13] Dostupné z: `http://goo.gl/hFVQjG`

[8] PFEIFFER, Joseph. *Writing a FUSE Filesystem: a Tutorial* [online]. 2012. [cit. 2013-10-08]
Dostupné z: `http://www.cs.nmsu.edu/~pfeiffer/fuse-tutorial/unclear.html`

[9] SZEREDI, Miklos, HENK, Csaba a Valient GOUGH. Project of the Month, April 2006 *Filesystem in Userspace (FUSE)* [online]. 2006.
Dostupné z: `http://sourceforge.net/potm/potm-2006-04.php`

[10] davidz. *Getting GVfs and FUSE right* [online]. Poslední modifikace - 1 October 2008. [cit. 2013-12-15]
Dostupné z: `http://goo.gl/67lwv0`

[11] SINGH, Sumit. Develop your own filesystem with FUSE. *No kernel programming required.* [online]. Poslední modifikace - 28 February 2006. [cit. 2013-11-23]
Dostupné z: `http://www.ibm.com/developerworks/library/l-fuse/`

[12] JAROŠ, Přemysl. *KIVFS – Kient pro GNU/Linux s použítím FUSE*. Plzeň 2012. Diplomová práce. Západočeská univerzita. Vedoucí práce Luboš Matějka.

[13] STREJC, Radek. *KIVFS – Datové úložiště*. Plzeň 2012. Diplomová práce. Západočeská univerzita. Vedoucí práce Luboš Matějka.

[14] HOVORKA, Karel. *KIVFS - Webový klient*. Plzeň 2013. Diplomová práce. Západočeská univerzita. Vedoucí práce Luboš Matějka.

[15] Struktura KIVFS. [online]. poslední modifikace - 22.09.2009. [cit. 2014-04-25]
Dostupné z: `http://goo.gl/8Rbp4Q`

[16] WOLTER, Jan. *Unix Incompatibility Notes: Byte Order*. [online]. Sat Dec 24 23:24:14 EST 2005. [cit. 2014-04-26]
Dostupné z: `http://unixpapa.com/incnote/byteorder.html`

[17] Filesystem in userspace. [cit. 2013-12-23].
Dostupné z: `http://fuse.sourceforge.net/`

[18] SQLite3 knihovna. [online]. [cit. září 2013]
Dostupné z: `http://www.sqlite.org/draft/wal.html`

[19] BŽOCH, Pavel. *CacheSimulatorV2*. [počítačový program]. Ver. 2. Poslední modifikace - listopad 2013. [cit. 2014-03-17]
Dostupné z: `http://home.zcu.cz/~pbzoch/software.html`

[20] SZEREDI, Miklos, Re: [fuse-devel] big_writes option does not work, 14. října 2013. [cit. 2014-04-30]
Dostupné z: <`http://sourceforge.net/p/fuse/mailman/message/31518307/`>