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

DIPLOMOVÁ PRÁCE

KIVFS – Klient pro GNU/LINUX s použitím FUSE

Plzeň, 2012 Přemysl Jaroš

Zde bude zadání

Prohlášení

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

V Plzni dne 16. května 2012

Přemysl Jaroš

Abstract

Project KIVFS is the new implementation of distributed filesystem. The goal of this diploma thesis is to add cache and offline operation support to driver for KIVFS. The driver is to run on GNU/Linux operation system with FUSE library. In the theoretical part of this work are describe the basic principles operations of cache, some necessary algorithms to keep cache consistence and some algorithms for choose file for delete when cache is full. Design and implementation of the improvements is realized in the practical part, including testing of the cache functionality. This work also consists of benchmark for comparison to older version of driver.

Obsah

1 Úvod

Cílem této práce je navrhnout a implementovat rozšíření klientského ovladače pro distribuovaný souborový systém KIVFS. KIVFS je distribuovaný souborový systém, který je vyvíjený na Katedře informatiky a výpočetní techniky Fakulty aplikovaných věd na Západočeské univerzitě v Plzni.

Tato práce navazuje na moji předchozí bakalářskou práci, v jejímž rámci vznikl samotný jednoduchý klient. Současným cílem je jeho rozšíření o klientskou cache za účelem zlepšení výkonnosti, datové propustnosti, přidání podpory offline režimu a aktualizace pro podporu nových verzí linuxového jádra a serveru.

Teoretická část čtenáře seznámí s tím, co je distribuovaný souborový systém. Objasní, na jakých principech pracuje knihovna FUSE a vysvětlí její výhody a nevýhody. Dále se zabývá popisem použitých technologií. Stěžejní část tvoří seznámení s tím, co je to cache a k čemu slouží, co jsou to offline operace a proč a jaké kroky jsou v důsledku toho nutné pro udržení konzistence dat.

Praktická část je věnována návrhu a popisu implementace. Na konci této části se nachází výkonnostní testy se srovnáním s předchozí verzí ovladače.

2 Teoretická část

Tato kapitola má za úkol přinést teoretický úvod do souborových systémů a použitých technologií. Následuje popis klientské cache a offline operací.

2.1 Organizace dat

Pevné disky jsou obvykle rozděleny na oddíly, na kterých se rozkládá konkrétní souborový systém. Díky tomu je možné používat na jednom disku více typů souborových systémů. Informace uložené na disku se dělí na metadata a data. Metadata popisují strukturu systému souborů a nesou další informace, jako je velikost souboru, čas poslední změny, oprávnění atd. Samotná data jsou uložena v podobě souborů.

V poslední době se začíná objevovat nový typ souborových systémů, které se odklánějí od klasické struktury založené na hierarchickém principu. Místo toho se přiklánějí k databázovému pojetí reprezentace dat založené na metadatech.

Souborový systém je v informatice označení pro způsob organizace a uložení dat ve formě souborů a adresářů tak, aby se k nim dalo snadno přistupovat. Souborové systémy se mohou nacházet na mnoha různých typech medií, jako jsou pevné disky nebo přenosné media, jako například CD nebo Flash paměťové karty na kterých se nacházejí klasické souborové systémy jako například NTFS[1], XFS[2] nebo EXT3[3]. Také se mohou nalézat na vzdálených úložištích a být přístupné prostřednictvím sítě, jako například NFS[4], OpenAFS[5], CODA[6]. Nebo dokonce mohou poskytovat přístup k čistě virtuálním datům, jako je procfs[7] v systému Linux, kdy se žádná data nenacházejí na jakémkoliv fyzickém mediu, ale jedná se o obraz stavu jádra. Souborový systém zpravidla umožňuje ukládat samotná data do souborů pod určitým názvem a tyto soubory následně organizovat prostřednictvím adresářů. Adresářová struktura tvoří

datovou strukturu typu strom.

Síťový souborový systém je označení pro systém, který je dostupný prostřednictvím počítačové sítě. Soubory a adresáře leží na jiném počítači než klient, a ten k nim přistupuje pomocí předem dohodnutého protokolu. Protokol je v tomto případě soubor speciálních síťových volání služeb. Na vzdáleném počítači pak mohou být soubory a adresáře uloženy prostřednictvím klasického souborového systému.

Distribuovaný souborový systém představuje speciální případ síťového souborového systému, kdy místo jednoho serveru je několik a mohou obsahovat stejná data. Podrobněji se zabýváme distribuovanými souborovými systémy v následující kapitole.

2.2 Distribuovaný souborový systém

DFS je definován jako soubor nezávislých počítačů, který se navenek jeví uživatelům jako jeden logický celek. Aby toto mohlo být splněno v heterogenních systémech a sítích, tak je distribuovaný systém tvořen dvěma částmi. Jedna je součástí vrstvy operačního systému a druhá je umístěna v aplikační vrstvě.

Distribuovaný souborový systém se od síťového souborového systému odlišuje zejména tím, že je tvořen několika servery propojených počítačovou sítí, které mohou obsahovat stejná data. V důsledku toho máme přístup k datům i v případě výpadků několika ze serverů. Klient k datům přistupuje prostřednictvím speciální komunikační vrstvy, která pracuje s daty pomocí předem známého protokolu. Tato vrstva navíc obsahuje kontrolu identity za pomoci jména a hesla nebo certifikátu. DFS se vyznačuje transparentností, škálovatelností, odolností proti ztrátě dat a bezpečností.

Každý distribuovaný souborový systém by měl obsahovat prostředky pro replikaci dat, z důvodů zabezpečení proti ztrátě dat. V případě výpadku jen omezeného množství uzlů je systém schopen pracovat dále bez výpadku služby uživateli a beze ztráty dat. Za předpokladu, že tato data obsahuje některý z funkčních uzlů. Replikace dat na jednotlivá datová úložiště je před uživatelem skryta. Replikace dále přináší větší výkonnost díky možnosti souběžného čtení.

Další vlastností distribuovaného souborového systému je transparentnost. Data jsou přístupná prostřednictvím počítačové sítě a uživatel s nimi může zacházet stejně jako s jinými soubory na svém lokálním disku. Uživatel neví, kde jsou data uložena, a ani v kolika replikách.

Škálovatelnost umožňuje vyrovnávání zátěže jednotlivých serverů, které navíc mohou být od sebe geograficky velmi vzdáleny. To je možné právě díky replikaci. Systém uživatele automaticky připojí k nejvýhodnějšímu serveru tak, aby mu zajistil maximální rychlost a odezvu systému.

2.3 KIVFS

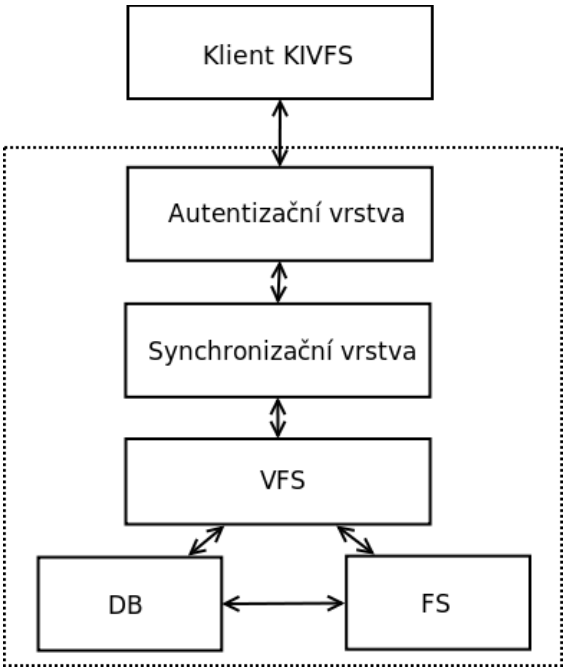
KIVFS je distribuovaný souborový systém vyvíjený na Katedře informatiky a výpočetní techniky Fakulty aplikovaných věd na Západočeské univerzitě v Plzni.

KIVFS server se skládá ze třech logických vrstev určených podle toho, jaké služby systému daná vrstva poskytuje. Znázornění jednotlivých vrstev je na obr. 1. Autentizační vrstva zajišťuje autentizaci a autorizaci uživatelů, a navíc šifruje veškerou komunikaci mezi klientem a serverem. V hierarchickém modelu se nachází nejvýše a předává nižším vrstvám přijaté požadavky, aby je mohli obsloužit a nakonec klientovi vrací výsledek. Synchronizační vrstva vykonává synchronizaci dat mezi servery a zajišťuje vykonávání požadavků ve správném pořadí. Vrstva VFS je datovou vrstvou serveru, zajišťuje způsob uložení dat a umožňuje vyšším vrstvám přístup k těmto datům.

KIVFS je vyvíjeno pro použití v běžných nezabezpečených sítích, jakou je například síť Internet. Proto je veškerá komunikace šifrována. Pro samotné šifrování je použito knihovny OpenSSL, a pro autorizaci s autentizací uživatele v systému je použit protokol Kerberos[1].

Jednotlivé vrstvy běží jako samostatné procesy a komunikují mezi sebou pomocí protokolu TCP.

1 Protokol Kerberos V5 je popsán normou RFC 1510 z roku 1993, V roce 2005 nahrazena RFC 4120 Protokol dále popsán dokumenty RFC 3961, RFC 3962, RFC 4121, RFC 4556



Obrázek 1: Vrstvy KIVFS

2.3.1 Autentizační vrstva

Cílem autentizační vrstvy je poskytnout zabezpečení prokázání identity a ochranu přenášených dat.

Toho je docíleno pomocí procesů autorizace, autentizace a šifrování přenosu dat. Proces autentizace je zajištěn protokolem Kerberos. Údaje o všech uživatelích jsou uložena v samostatné databázi. Šifrování komunikace je realizováno pomocí protokolu SSL. Přenášení souborů je prozatím nešifrováno, ale probíhá na náhodných portech. Je použit vlastní Kerberos server s implementací Heimdal, ale je možné použít i jinou implementaci. Heimdal[8] bylo rozhodnuto použít v rámci projektu KIVFS, protože jde o moderní implementaci nabízející více možností, například více současných realmů.

Pro šifrování komunikace je použita knihovna OpenSSL. Předání symetrického klíče je provedeno pomocí SSL Handshake. SSL Handshake k tomu používá asymetrický algoritmus RSA. Komunikace je šifrována pomocí symetrické šifry AES s délkou klíče 256 bitů. Podrobnosti nalezneme v [9].

2.3.2 Synchronizační vrstva

Přístup ke sdíleným zdrojům je složitější než v centralizovaných systémech díky nutnosti synchronizace a řazení událostí. Řazení je realizováno za pomoci Lamportova algoritmu pro vzájemné vyloučení a vektorových značek.

Lamportův algoritmus

- Každý proces si udržuje frontu, ve které uchovávají požadavky seřazené za pomoci logických hodin.
- V případě nového požadavku musí proces zaslat tento požadavek všem ostatním procesům spolu s časovou známkou a zařadí si požadavek do fronty.
- Pokud proces přijme požadavek, tak jej zařadí do fronty a pošle zpět potvrzení.
- Pokud proces přijal všechna potvrzení pro první požadavek ve frontě, tak se tento požadavek může vykonat.
- Pokud nastane konflikt, tak je vyřešen pomocí hodnoty ID uživatele. Rozhoduje uživatel s vyšší hodnotou ID, další v pořadí musí svůj požadavek opakovat.

2.3.3 VFS

Je inspirována virtuálním souborovým systémem (VFS) pro transparentní přístup k uloženým souborům. Celou tuto vrstvu je možné dále rozdělit na tří části. VFS server KIVFS slouží k předávání požadavků mezi vyšší vrstvou a databázovým či souborovým serverem KIVFS.

Jednou z částí je DB server, který slouží pro ukládání metadat a dalších potřebných informací, jako například ACL. Souborový server KIVFS pro ukládání dat souborů a zajišťuje jejich replikaci. Výhodou je eliminace požadavků na souborový systém, kdy všechny požadavky jsou obslouženy z databáze, až na požadavky na samotná data. Jako databázový systém pro ukládání metadat je použit relační

databázový server MySQL.

Pro ukládání samotných dat slouží FS server. Jako souborový systém je použit souborový systém XFS, pro jeho spolehlivost a dobré parametry pro práci s velkými daty. V případě potřeby je možné jak XFS tak MySQL vyměnit za jiné. Podrobnosti nalezneme v bakalářské práci[24].

2.4 Použité technologie

Součástí projektu jsou některé další technologie a projekty, které si přiblížíme v následujících podkapitolách.

2.4.1 Kerberos

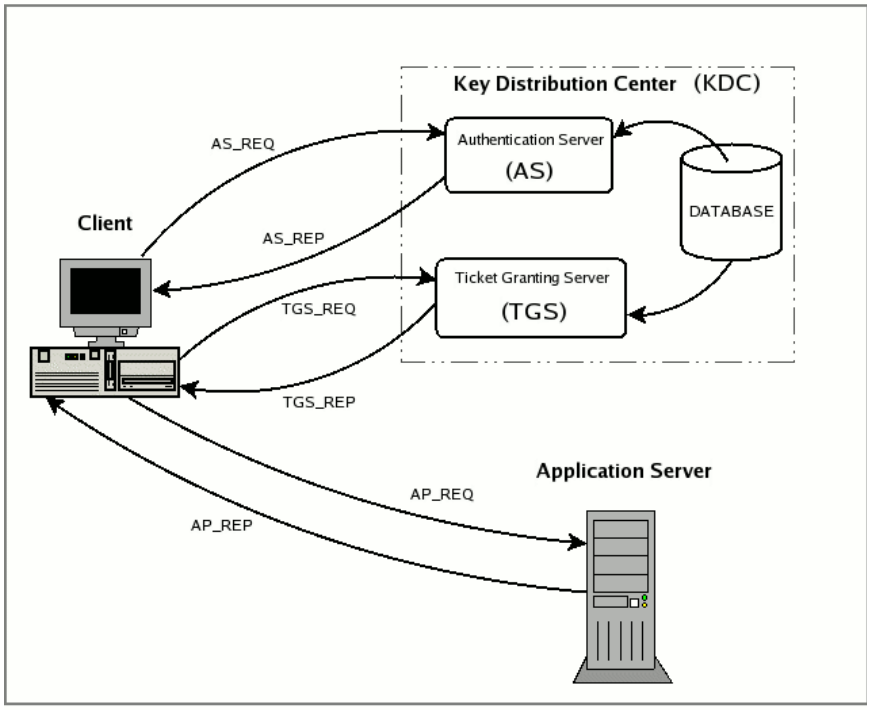
Je autentizačním protokolem, který umožňuje bezpečně prokázat svoji identitu v nezabezpečené síti a ověření této identity. Kerberos zabraňuje odposlechnutí a zaručuje integritu dat. Po sítí se neposílá heslo, ale pouze hash spočtená z hesla.

Vývoj Kerbera započal v 90.letech na Massachusetts Institute of Technology(MIT)[10]. Protože do roku 2000 nebylo možné vyvážet ze Spojených států šifrovací algoritmy použité v implementaci od MIT, tak vznikly další implementace. Jednou z nich je Heimdal[8] pocházející ze Švédska.

Nejprve bychom si měli nejprve vysvětlit některé základní zkratky a pojmy. Kerberos je postavený na symetrické kryptografii a vyžaduje existenci věrohodné třetí strany, kterou představuje Key Distribution Center(KDC). Skládá se ze dvou logických částí a to Authentication Server(AS) a Ticket Granting Server(TGS). Kerberos pracuje na principu tiketů, které slouží k prokázání a ověření identity uživatele.

KDC obsahuje databázi soukromých klíčů pro každou entitu v síti, každý soukromý klíč zná pouze vlastník a KDC. Znalost tohoto klíče slouží k ověření identity entity. Pro komunikaci mezi dvěma entitami KDC vygeneruje tzv. session key, který pro

zabezpečení jejich komunikace. Bezpečnost protokolu spoléhá na synchronizaci časů účastníků a krátkou životnost tiketů.



Obrázek 2: Kerberos protokol

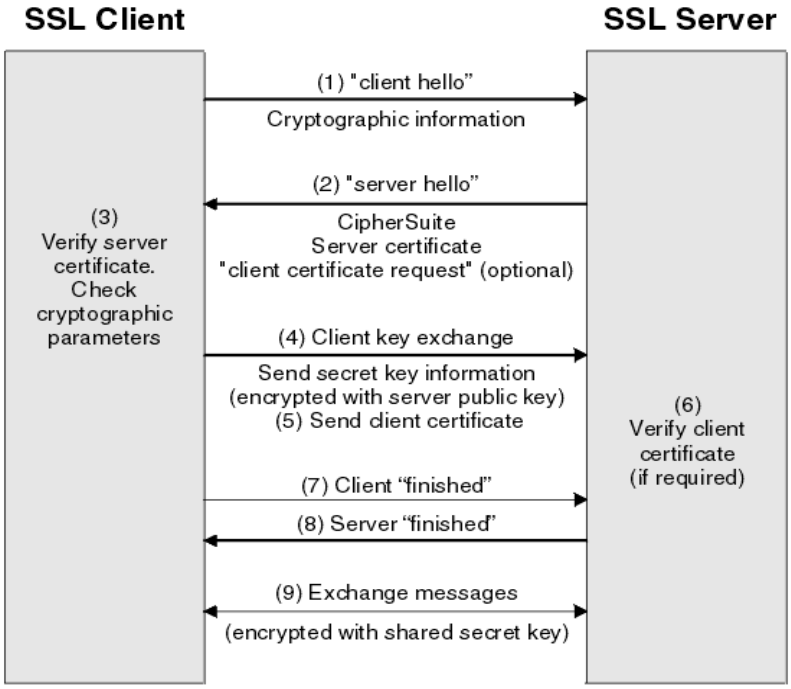
Zdroj: http://www.kerberos.org/software/tutorial.html

Znázornění průběhu ověření se nachází na obr. 2. Vždy se klient nejprve ověří na AS a přijme tiket. Všechny tikety mají časovou značku. Následně kontaktuje TGS a použije tiket k prokázání identity a dotáže se na službu. Pokud je služba danému klientovi dostupná, pak TGS pošle klientovi další tiket. Klient kontaktuje server s požadovanou službou a použije tento tiket k prokázání oprávnění používat danou službu. Ticket-Granting Ticket(TGT) je tiket, který klient obdrží od KDC.

Klient se ověřuje na AS pouze jednou použitím např. hesla nebo certifikátu. A přijme TGT od AS. Později když klient chce použít nějakou službu, tak použije tento tiket k získání dalších tiketů od TGS. Tyto tikety jsou použity k prokázání oprávnění pro danou službu.

2.4.2 Secure Sockets Layer

Secure Sockets Layer, neboli SSL je protokol pro zabezpečení komunikace v počítačové síti. Vyvinula ho společnost Netscape v roce 1996. Následovníkem SSL je protokol Transport Layer Security(TLS).



Obrázek 3: protokol handshake

Zdroj: http://publib.boulder.ibm.com/infocenter

Protokol SSL je vrstva vložená mezi vrstvu transportní a vrstvu aplikační. Funguje na bázi klient-server. U SSL se při iniciaci spojení vymění sdílený klíč pomocí asymetrického šifrování. SSL kontroluje navíc integritu dat pomocí Message authentication code(MAC). V projektu KIVFS je použita pro šifrování přenosu mezi klientem a serverem.

Kromě protokolu MAC používá SSL ještě další tři protokoly. Record Protocol přebírá data z aplikační vrstvy a rozděluje tyto data do bloků, které zašifruje a následně předá do vrstvy transportní. Alert Protocol kontroluje spojení a v případě výskytu chyby

spojení může ukončit. Handshake Protocol je protokol pro zahájení šifrovaného spojení pomocí asymetrické šifry. Průběh handshaku nalezneme na obr. 3.

2.5 GNU/Linux

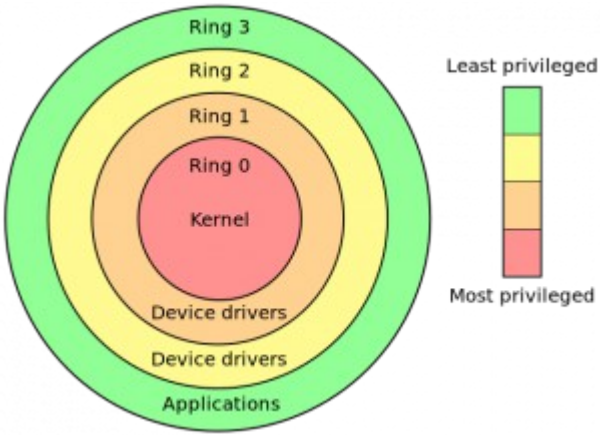
Linux je označení pro jádro operačního systému šířeného pod svobodnou licencí GPLv2. Plné znění licence lze nalézt na stránkách projektu GNU[11] na domovské adrese projektu.

Ačkoliv Linux je pouze jádro systému, tak v praxi je součástí mnoha různých distribucí, které tvoří kompletní operační systém včetně velkého množství aplikací. Jednotlivé distribuce jsou vyvíjeny a spravovány jednotlivci, týmy dobrovolníku nebo komerčními firmami. Největší distribuce jsou Red Hat, SUSE, Debian nebo Ubuntu.

2.5.1 Linuxové jádro

Autorem první verze z roku 1991, je Linus Torvalds. Linux vychází z myšlenek Unixu a snaží se o dodržování standardu POSIX[12]. Jde o víceúlohový a víceuživatelský systém.

V současnosti je větší část jádra napsána v jazyce C s využitím některých rozšíření překladače GCC a některé části jsou v dalších jazycích, jako například assembler. Dříve bylo celé jádro koncipováno jako monolitické, ale dnes přijalo podporu modulů. Podpory externích modulů se využívá k vyšší stabilitě či zmenšení velikosti jádra. Moduly je možné za běhu zavádět do jádra a odstraňovat z jádra. Linuxové jádro obsahuje podporu všech důležitých vlastností moderního operačního systému jako multitasking, virtuální paměť, sdílené knihovny a další. Ovladače zařízení typicky běží v ring0, jedná se o režim s plným přístupem k hardwaru a privilegovaným instrukcím. V tomto režimu by mělo běžet pouze jádro operačního systému. Znázornění můžeme vidět na obr. 4.



Obrázek 4: Režimy procesoru

2.5.2 VFS

Virtuální souborový systém[23] je abstraktní vrstva systému nad všemi konkrétními souborovými systémy. Z uživatelského pohledu se v Linuxu celý souborový systém tváří jako strom s jedním kořenem. Tento strom se však může skládat z více než jednoho souborového systému. Pro uživatelské aplikace se však celý strom tváří jako konzistentní celek a nepoznají, jaká část patří jakému souborovému systému. Toto je zajištěno díky tomu, že přístup k souborům probíhá skrze rozhraní virtuálního souborového systému (zkráceně VFS). Jde o rozhraní, které zavádí abstrakci nad všemi souborovými systémy a umožňuje se všemi soubory pracovat stejně bez ohledu na znalosti, o který konkrétní použitý souborový systém se jedná. Každé volání jednotlivých operací pak procházejí skrze VFS do konkrétního ovládače a zpět.

2.6 Možnosti tvorby ovladače

Pokud chceme uživateli zpřístupnit vlastní souborový systém, tak máme tři různé možnosti. Buď můžeme vytvořit klasický jaderný modul a získat výbornou integraci do systému. Nebo můžeme vytvořit vlastní řešení na aplikační úrovni, jaké může představovat například rozšíření nějakého stávající souborového správce nebo

nativní aplikace, jako např. FTP[13]. Novou možností z posledních let je vytvoření ovladače v uživatelském prostoru, jakým je například v systému GNU/Linux knihovna FUSE[14].

2.6.1 Jaderný modul

Souborové systémy se běžně uživatelům zpřístupňují pomocí ovladačů, které jsou vyvíjeny jako přímá součást jádra nebo jako jaderný modul, to záleží na architektuře a možnostech daného operačního systému. Takovýto ovladač se podobá klasickému ovladači zařízení. Nachází se při svém běhu v takzvaném prostoru jádra (anglicky kernel space), což mu umožňuje přímý přístup k hardwaru počítače. Jedná se o nejefektivnější možnost z pohledu výkonu, ale ne však z pohledu vývoje. Psaní jaderných ovladačů není triviální a je nutné postupovat velmi obezřetně, protože případná chyba takového ovladače může způsobit selhání celého systému. Přičemž kód jaderného ovladače se obtížně ladí. V případě systému GNU/Linux nelze také opomenout neustálé změny API.

2.6.2 Nativní aplikace

Můžeme vytvořit svou vlastní zcela samostatnou aplikaci pro zpřístupnění souborového systému a nebo můžeme pouze vytvořit plugin do již existujícího kompletního souborového správce, jakým je například Midnight Commander. Ušetříme tím čas za vývoj a navíc uživatel může používat známou aplikaci, na kterou je zvyklý. Nevýhodou je závislost na této aplikaci a případná nutnost si tuto aplikaci pořídit, pokud ji uživatel již nepoužívá. Tak je tomu například u pluginu do programu Total Commander, který je také vyvíjen v rámci projektu KIVFS. Poslední nevýhodou je nemožnost používat takto zpřístupněný souborový systém z jiných aplikací.

2.6.3 FUSE

Knihovna FUSE není historicky první a jediná možnost vývoje ovladače souborového systému mimo jádro pod systémem GNU/Linux, ale v současnosti je tato knihovna nejpoužívanějším řešením a stala se standardem. Pokud se rozhodneme použít tuto knihovnu, tak můžeme vyvíjet ovladač jako klasickou aplikaci běžící v uživatelském prostoru (anglicky user space), která komunikuje s jádrem prostřednictvím knihovny FUSE. Získáme tím rychlejší vývoj ovladače. Daní za tuto programátorskou přívětivost je nižší výkon nebo alespoň vyšší režie CPU způsobená nutností přepínání mezi režimy jádra a uživatelským režimem.

2.7 Systém FUSE

Pro vytvoření ovladače pro KIVFS bylo v této práci rozhodnuto použít knihovnu FUSE. Důvodem pro vybrání této možnosti je stabilnější API knihovny než u samotného jádra. Použití této knihovny sice sebou přináší vyšší režii CPU a nižší rychlost, ale v našem případě síťového souborového systému je omezující zejména rychlost počítačové sítě.

2.7.1 Úvod do FUSE

Název FUSE vznikl zkrácením názvu Filesystem in Userspace. Jde o modul do jádra unixového operačního systému GNU/Linux. Tento jaderný modul je šířen pod svobodnými licencemi GNU GPL[15] a GNU LGPL[16]. Pod licencí GPL se nachází jaderná část FUSE, jinou variantu licence jádra ani neumožňuje. Knihovna `libfuse` je pak licencována pod LGPL, to umožňuje vytváření ovladačů souborových systémů pod libovolnou licencí včetně uzavřených. Této možnosti využívá například ovladač pro souborový systém ZFS[17], kde jiná možnost z licenčních důvodů není možná. Zbylé části, které tvoří ukázkové příklady a další pomocné programy, jako například `fusermount`, což je pomocná aplikace pro připojování a odpojování souborových systémů.

Cílem FUSE je umožnit připojovat a používat další souborové systémy i neprivilegovaným uživatelům. A vytvářet vlastní souborové systémy bez nutnosti psát programový kód pro jádro operačního systému. Tohoto cíle bylo dosaženo spouštěním kódu souborového systému v uživatelském prostoru a vytvoření jaderného modulu FUSE, který následně zajišťuje komunikaci mezi jádrem a ovladačem a vytváří tak jakýsi překlenovací most.

Knihovna FUSE byla oficiálně přidána do hlavního stromu jádra Linux ve verzi 2.6.14, ale později bylo jeho používání zpětně umožněno i na jádrech starší řady 2.4.X. V současnosti je FUSE podporováno i v nejnovější řadě 3.X. Celý systém FUSE původně vznikl jako součást virtuálního souborového systému AVFS[18]. Později se odštěpil do samostatného projektu na stránkách SourceForge[14].

Za nativní jazyk celého projektu FUSE je možné označit programovací jazyk C. V tomto jazyce je implementována celá knihovna a modul fuse, pomocné programy i standardní rozhraní pro vývoj vlastních ovladačů je v tomto jazyce. To ovšem neznamená, že jde o jediný jazyk, ze kterého můžeme tuto knihovnu využívat. Kromě automaticky se nabízejícího jazyka C++ navíc svobodná licence umožnila vytvoření knihoven postavených nad knihovnou `libfuse` a zpřístupňující její možnosti do dalších programovacích jazyků jako, jsou například Java, Python a Perl. Vzniklo i plně objektové rozhraní pro jazyk C++.

Pro své technologické možnosti je FUSE knihovna užitečná a využívána pro tvorbu takzvaných virtuálních souborových systémů. Tradiční souborový systém má za úkol zpřístupňovat data na disku prostřednictvím jejich čtení a zápisu. Ovšem virtuální souborové systémy data jako taková ukládat vůbec nemusejí, mohou poskytovat pouze některé operace, např. mohou zpřístupnit data pouze pro čtení.

Zajímavým příkladem takovéto technologie byl ovladač GmailFS[19], který umožňoval ukládání dat jako poštu na poštovních serverech Gmail. V současnosti tato možnost není dostupná, protože společnost Google odstranila veřejné API, které tento ovladač využíval. Dalším příkladem může být LoggedFS[20],jehož cílem je poskytovat možnost záznamu veškerých operací nad ním provedených do logu. FuseISO[21] poskytuje přístup k obrazům optických disků v různých známých formátech jako ISO, IMG nebo BIN a nahrazuje tak v sytému MS Windows velmi známý program Daemon

tools. Dříve byla technologie FUSE také využívána ke zpřístupnění souborového systému NTFS.

Technologický návrh systému FUSE umožňuje její portování na další unixové operační systémy a vývojářům se tak dostává do rukou nevídaná možnost tvorby ovladače souborového systému pro více operačních systémů najednou s žádnými nebo malými úpravami pro jednotlivé operační systémy. Systém FUSE byl zpřístupněn na Mac OS X pomocí projektu MacFUSE od společnosti Google. Nyní však není dále vyvíjen a podporován. V rámci velmi známého projektu Google Summer of Code vznikla podpora na operačním systému FreeBSD.

V historii vznikl velmi podobný projekt pod názvem LUFS[22], který sice již několik let nevykazuje žádnou aktivitu, ale je možné na něm ilustrovat, že myšlenka ovladače souborového systému v uživatelském prostoru není nová a FUSE není jediným pokusem o její realizaci a že způsob FUSE není jedinou cestou. Oproti FUSE, kde ovladač tvoří klasický spustitelný soubor, který po spuštění volá knihovnu `libfuse` ke komunikaci s jádrem, tak ve LUFS se ovladač vytvářel jako knihovna, kterou načetl příkaz lufsmount. Dalším zajímavým rozdílem je, že LUFS navíc provádělo automatické cachování adresářů a přidružených atributů, to FUSE nedělá. FUSE si lépe představíme jako obyčejný interface mezi jádrem a ovladačem, který neprovádí žádnou další činnost.

2.7.2 Vnitřní popis FUSE

Celá knihovna FUSE je postavena na tzv. callback funkcích. Vytvoří se implementace funkce souborových operací podle předem daných prototypů a FUSE se následně předají ukazatele na tyto funkce. Knihovna FUSE následně tyto funkce volá podle přicházejících požadavků. Ukazatele na funkce se předávají funkci `fuse_main()`, jejímž zavoláním se program dostane do smyčky obsluhy požadavků a ztrácíme tak kompletní kontrolu nad vykonáváním aplikace.

První částí FUSE je samotná knihovna `libfuse`. Když uživatelský program zavolá funkci `fuse_main()`, tak tato funkce nejprve analyzuje parametry získané

z uživatelského programu a následně zavolá funkci `fuse_mount()`.

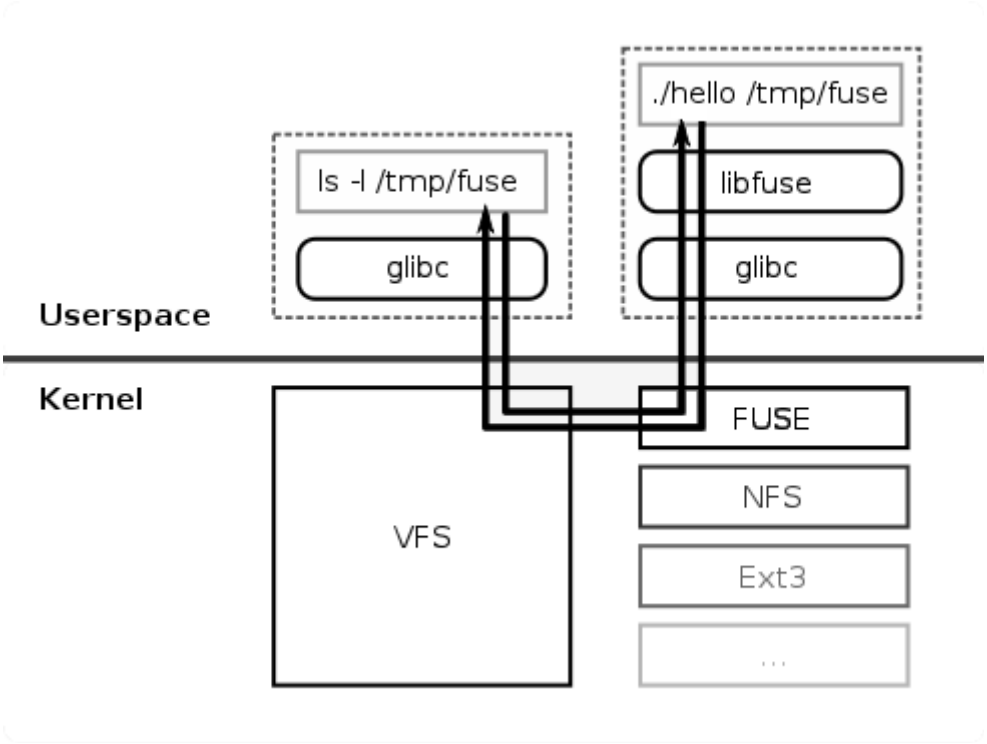
Funkce `fuse_mount()` vytvoří socketové spojení, provede fork a spustí program fusermount, kterému předá jeden konec socketového spojení v proměnné prostředí jménem FUSE_COMMFD_ENV. Program `fusermount` se ujistí, že je zaveden jaderný modul FUSE, otevře zařízení `/dev/fuse` a odešle získaný souborový deskriptor, získaný otevřením souboru `/dev/fuse` zpět do funkce `fuse_mount()`. Funkce `fuse_mount()` ho dále vrátí do funkce `fuse_main()`, ve které vše začalo.

Funkce `fuse_main()` nyní zavolá `fuse_new()` pro vytvoření všech potřených datových struktur. A nakonec zavolá funkci `fuse_loop()` (nebo funkci `fuse_loop_mt()`, ta se od té předešlé odlišuje vícevláknovým chováním). Tato funkce čte požadavky na systémová volání ze /dev/fuse a volá uživatelem definované funkce, jejichž ukazatele jsou uloženy ve struktuře `fuse_operations` pro jejich obsluhu. Výsledky jsou zapisovány do /dev/fuse, odkud jsou předány systémovému volání.

Druhou část FUSE tvoří jaderný modul, který se skládá ze dvou částí. První částí je komponenta souborového systému proc. A druhou je obsluha systémových volání. Všechna definovaná systémová volání způsobí zavolání funkce `request_send()`, `request_send_noreply()` nebo `request_send_nonblock()`. Většina vede na volání funkce `request_send()`. Funce `request_send()` přidá požadavek do seznamu požadavků, kde čekají na odpověď. Zbylé dvě funkce jsou velmi podobné té první, jen první je neblokující a druhá nečeká na odpověď.

Druhá část jaderné části má na starosti odpovědi na požadavky ze souboru /dev/fuse. Funkce `fuse_dev_read()` zachází s požadavky na čtení a vrací příkazy ze seznamu požadavků. To samé pro zápisy má na starosti funkce `fuse_dev_write()`.

Cestu požadavku systémem znázorňuje následující obr. 5.



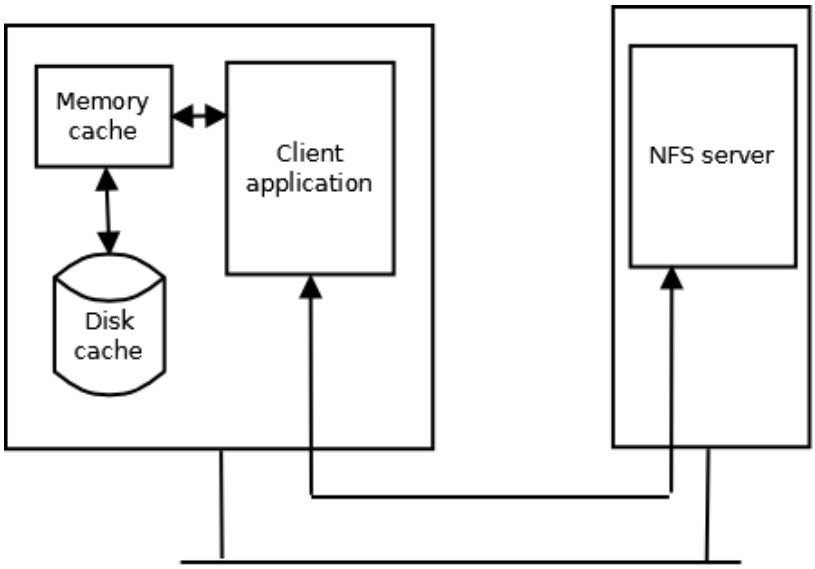
Obrázek 5: Průběh požadavku ve fuse

Uživatelský program zavolá funkci souborového systému. Požadavek pak cestuje skrze standardní knihovnu glibc do jádra operačního systému, tím se dostává do jaderného prostoru. V jádře pak požadavek prochází skrze VFS do konkrétního ovladače souborového systému, který je používán pro dané umístění. V našem případě se jedná o ovládač ve FUSE. Zde se opouští jaderný prostor. Obsluha požadavku končí v ovladači v uživatelském prostoru a nakonec se výsledek vrací stejnou cestou zpět.

2.8 Cache

Cache je název pro vyrovnávací paměť, která je umístěna mezi dvěma systémy. Typicky tyto dva systémy mají různou rychlost a cache slouží k vyrovnání rychlostí mezi nimi. Hlavním cílem je urychlení přístupu k datům na pomalejším datovém úložišti překopírováním na rychlejší „místní“ medium. Dalším efektem použitím cache

paměti je snížení objemu přenesených dat v případě opakovaného čtení dat, která se již nacházejí v cache. Znázornění se nachází na obr. 6.



Obrázek 6: Zobrazení umístění cache v systému

Existují dva základní typy cache:
hardwarová cache – tvořená hardwarovým vybavením, například u CPU nebo HDD
softwarová cache – tvořena programovým vybavením, jí se budeme zabývat dále v textu

2.8.1 Rozdíl mezi cache a bufferem

Cache a buffer jsou dva pojmy s podobnou funkcí a jejich funkce se může částečně prolínat, navíc se vzájemně nevylučují. Je mezi nimi ovšem rozdíl. Přestože je buffer také vyrovnávací paměť na vyrovnávání rychlostí u dvou stran, tak buffer je typicky jen dočasný. Data jsou do bufferu zapsána, po nějaké době přečtena a po přečtení jsou z bufferu odstraněna, ale v cache mohou některá data vydržet delší dobu než jiná.

2.8.2 Softwarová cache

Typické použití cache je pro pevný disk počítače na úrovni souborového systému. Systém se snaží soubory, se kterými pracuje, nejčastěji udržovat v rychlé operační paměti. A v případě zápisu na disk ukládat co nejpozději. Cílem je neprovádět zbytečné diskové operace, protože pevný disk je o několik řádů pomalejší než operační paměť.

V moderních systémech je paměť pro cache přidělována dynamicky podle množství volné operační paměti a podle aktuální potřeby.

Používání cache sebou přináší také nevýhody, například pro diskovou cache je to riziko ztráty dat v důsledku výpadku napájení nebo ztráta dat na externím úložišti v důsledku násilného odpojení. Dalším problémem, který je nutný řešit zvláště s ohledem na využití v distribuovaném prostředí, je udržování konzistence dat viz. kapitola 2.9.

Na cache se tady můžeme dívat jako na komponentu, ve které se ukládají data, aby požadavky na tyto data mohly být příště obslouženy rychleji. V případě požadavku na data, která jsou uložena v cache, se označuje „cache hit“ a požadavek může být obsloužen čtením z cache, která je výrazně rychlejší. V opačném případě hovoříme o takzvaném „cache miss“ a data se musí přečíst z pomalejšího úložiště. Snahou je, aby počet požadavků obsloužených z cache byl co možná největší, protože tím současně získáme větší rychlost celého systému.

Pro udržování konzistence je nutné udržovat u každého souboru značku vyjadřující jeho verzi. Často se také přidává hodnota, které se říká hit a která vyjadřuje, jak často bylo k tomuto souboru přistupováno. Ty mohou být navíc rozděleny pro zápis a čtení zvlášť.

2.8.3 Databáze pro cache

Když implementujeme softwarovou cache, tak je třeba uchovávat informace o souborech v ní. To je možné realizovat pomocí databáze nebo pomocí vlastního

formátu. Databáze byla vybrána, protože jde o již hotové řešení s optimalizacemi při práci s daty v ní obsažených. Dále bylo nutné vybrat vhodnou databázi. Byla vybraná knihovna SQLite oproti jiným velkým databázím jako MySQL nebo PostgreSQL. Protože jde o malou přenositelnou knihovnu, která je vhodná pro integraci do vlastních aplikací. Databáze jako PostgreSQL sice nabízejí podporu dalších technologií jako například PL/SQL, ale v našem případě nejsou potřeba a funkcionalita SQLite je plně dostačující.

SQLite je databázový relační systém vytvořený jako malá knihovna napsaná v jazyce C a šířená pod licencí public domain.

Na rozdíl od jiných databázových systémů, které jsou založeny na principu klient-server a kde je server spuštěn jako samostatný proces, tak SQLite je jen knihovnou, která se přilinkuje k aplikaci. Každá databáze je uložena v samostatném souboru *.db, kam se data ukládají za pomocí primárního klíče do stejně velkých bloků a knihovna dále využívá hashovací techniky pro rychlý přístup k datům.

V SQLite je implementován téměř celý standard SQL-92. Databázi SQLite je možné používat z celé řady programovacích jazyků a je dostupný pro různé operační systémy. Formát databázových souborů je nezávislý na operačním systému.

2.9 Konzistence

Cachování a replikace dat hraje velmi důležitou úlohu v distribuovaných systémech. Ovšem v důsledku použití cache vzniká nutnost jejího udržování v konzistentním stavu tak, aby uživatel obdržel vždy správná data.

Existuje několik způsobů udržování konzistence dat. V následujících oddílech si postupně probereme základní modely konzistence dat a v další části si ukážeme dva praktické příklady.

2.9.1 Modely konzistence

V této kapitole se nachází popis některých základních modelů konzistence.

Striktní konzistence

Jakékoliv čtení vrátí data uložená pro posledním zápisu. Jedná se o absolutní časové uspořádání. Všechny zápisy musí být okamžitě všude viditelné.

Sekvenční konzistence

Výsledek veškerých provedených operací musí být stejný, jako kdyby byly všechny operace vykonány v sekvenčním uspořádání. A operace všech procesů se jeví v téže sekvenci, ve které byly specifikovány programem.

Kauzální konzistence

Kauzálně závislé operace zápisů musí být viděny všemi procesy ve stejném pořadí. Vyžaduje udržování grafu závislostí operací zápisu a čtení.

FIFO konzistence

Zápisy prováděné jedním procesem jsou viděny ostatními procesy v tom pořadí, v jakém byly prováděny. Zápisy různých procesů mohou být viděny různými procesy v různém pořadí.

Eventuální konzistence

Po ukončení všech zápisů musí být všechny repliky aktualizovány v konečném čase. Problém nastává, pokud se proces podívá do jiných replik.

Monotonní čtení

Po přečtení hodnoty musí každé další čtení vrátit stejnou nebo novější hodnotu. Po připojení k jiné replice tedy musí uživatel vidět všechny zprávy co si už dříve přečetl.

Monotonní zápis

Zápisy do proměnné jsou provedeny vždy před dalším zápisem do ní. Než se do repliky zapíše, tak je nutné ji nejprve aktualizovat na nejnovější hodnotu.

Čti své zápisy

Po zápisu jsou čteny své zápisy. Například pokud někdo aktualizuje stránku, tak při jejím následném čtení vidí své změny.

Zápisy následují čtení

Zápisy se provádějí do kopie, která je alespoň tak aktuální jako ta, která se před tím přečetla.

2.9.2 Realizace přístupu k souborům v cache

V předchozí kapitole byly probrány teoretické modely konzistence. V této kapitole se nachází konkrétní popis jak je možné programově realizovat udržování konzistence.

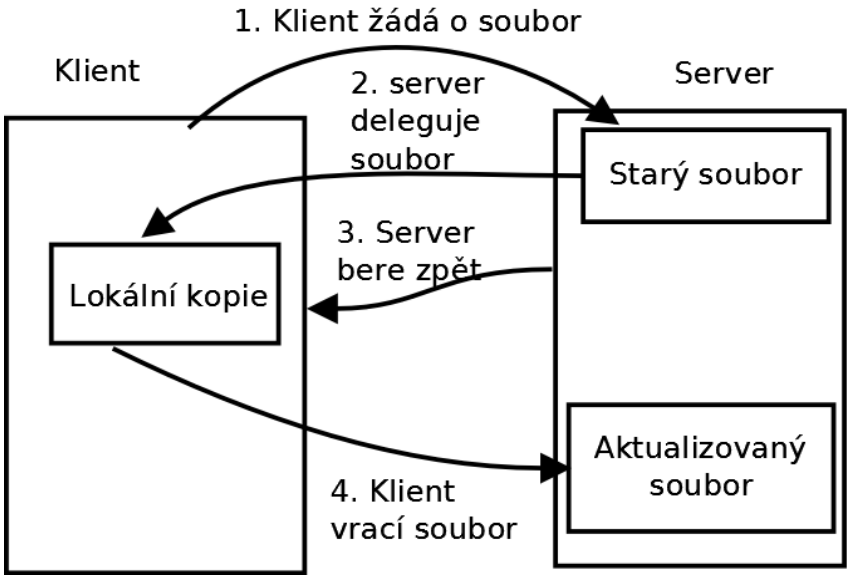
Validace při každém přístupu

Nejjednodušší způsob, který spočívá v tom, že pokud dojde k otevření souboru, tak je možné, aby byl při čtení uložen do cache vyrovnávací paměti nebo jen jeho část.

V případě zápisu do souboru v cache se musí veškeré změny promítnout mimo cache do centrálního úložiště nejpozději při uzavření souboru. V případě čtení souboru, který se už nalézá v cache, je nutná revalidace platnosti tohoto souboru před přístupem k němu a případného jeho zneplatnění, pokud byl v úložišti mezitím změněn.

Delegování práv

Další volitelné vylepšení spočívá v možnosti delegování práv serverem na klienta. Delegací přechází práva na klientův stroj. Například pokud klient na jednom stroji získá právo zápisu do souboru, tak již nemůže být přistupováno k tomu samému souboru z jiné stanice. Server drží povolení dané stanici po určitou dobu a nedovolí přístup žádné další. Pokud dojde k vypršení doby, tak server může povolení odejmout. Důležitým důsledkem možnosti delegování práv k souboru klientovy spočívá v nutnosti možnosti odebrání těchto práv serverem. To je možné realizovat například voláním vzdálených metod například RPC. Průběh udělení a odejmutí práva k souboru je znázorněn na obr. 6.



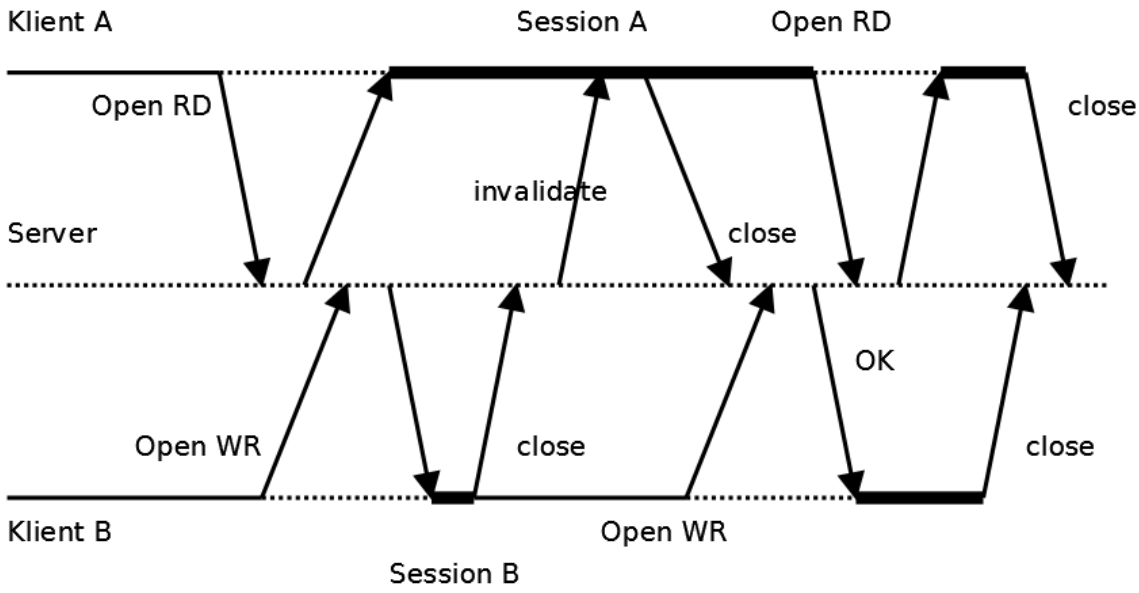
Obrázek 6: Průběh delegování

Výhoda tohoto přístupu spočívá v možnosti lokálního delegování práv. Protože delegace já vázána na stanici a ne konkrétní aplikaci. Pokud získá právo zápisu jedna aplikace ze serveru, tak další aplikace na té samé stanici nemusí kontaktovat server, ale stačí ji získat toto právo lokálně. Pokud stanice získá pouze právo čtení a následně by nějaká další aplikace chtěla zapisovat, tak lokální vyřízení nestačí a je nutné kontaktovat server pro získání práv.

Soubor je vždy v cache

Další možný přístup spočívá v tom, že v případě přístupu k souboru na serveru je tento soubor vždy uložen do cache.

Pro všechny soubory si server zapamatuje všechny klienty, kteří mají tento soubor uložen v cache. Pokud klient mění soubor jako první, tak o tom informuje server a ten dá vědět všem ostatním klientům, že byl tento soubor změněn. Pokud někdo další změní tento soubor, tak se tato změna nepromítne na server a při dalším otevření tohoto souboru si musí nejprve aktualizovat cache. Důsledkem tohoto je, že pokud klient přistupuje k souboru, který najde v cache, tak k němu může rovnou přistupovat, protože pokud by byl mezitím někým jiným změněn, tak by měl o tom informaci.



Obrázek 7: Lokální kopie

Průběh práce se souborem ilustruje obr. 7. Jak je vidět, pokud klient B změní soubor na serveru, který používá i klient A, tak je o tom klient A informován.

2.10 Metody výběru souboru pro odstranění z cache

Protože je diskový prostor určený pro cache omezen, tak je nutné zabránit nekonečnému narůstání velikosti cache tak, že se budou soubory také odstraňovat. Je důležité použít vhodný algoritmus pro výběr souboru na odstranění. Kvalita použitého algoritmu bude významně ovlivňovat celkovou výkonnost systému a efektivitu použití cache paměti. Cílem je udržovat v cache ty soubory, které se často používají tak, aby se nemusely opakovaně přenášet sítí.

2.10.1 Metody výběru

Existuje několik různých základních algoritmů pro výběr souboru. V této kapitole je popsáno několik z nich a nakonec se nachází experimentální algoritmus vyvíjený na KIVFS.

Random

Při ukládání souboru do plné cache se odstraní náhodně vybraný soubor. Možné vylepšení spočívá v tom, že se budeme snažit odstranit tak velký soubor z cache, aby se nemusel odstraňovat další.

FIFO

Z cache se odstraňuje nejstarší soubor. Soubory jsou tedy v cache uspořádány do pomyslné fronty podle svého stáří. Nově příchozí soubory se přidávají na konec fronty a v případě potřeby se odstraňují soubory ze začátku fronty tak dlouho, dokud v cache není dostatek místa pro nový soubor.

Second chance

Vychází z metody FIFO. U každého souboru navíc udržujeme příznak, který určuje, zda byl daný soubor použit. K odstranění z cache se vybírá nejdéle nepoužitý soubor z fronty. Pokud se ve frontě narazí na soubor, který má zapnutý příznak použití, tak se shodí. Z fronty se odstraňuje vždy první soubor bez aktivovaného příznaku použití.

LRU - Least Recently Used

V případě plné cache se odstraňuje nejdéle nepoužívaný soubor. Pro určení posledního použití souboru se musí uchovávat čas tohoto použití. Pro rychlé nalezení takového souboru se musí uchovávat informace o souborech v cache v nějaké snadno prohledatelné struktuře.

LFU - Least Frequently Used

Při plné cache se odstraňuje nejméně často používaný soubor. U všech souborů v cache je nutné v nějaké struktuře uchovávat počet přístupů k tomuto souboru. Počet těchto přístupů se označuje termínem „hits“. Vždy po určité době je nutné, aby se počet přístupů k souboru vždy aktualizoval. Aktualizaci může představovat například vynásobení nějakou konstantou z intervalu (0, 1.0). Tato aktualizace zaručuje, že se v cache nebude udržovat soubor, který byl jen jednou použit. Pokud se soubor nepoužívá, tak se jeho hodnota snižuje až na minimum.

Z cache se vždy odstraňuje soubor s nejmenší hodnotou přístupů. Při příchodu nového souboru do cache je nutné tuto hodnotu inicializovat. To je možné třeba na hodnotu o počtu přístupů získanou od serveru.

Hybrid mezi LRU a LFU

Tento algoritmus je experimentálně vyvíjen a použit v klientech v systému KIVFS. Hlavní myšlenkou algoritmu je nedržet v cache paměti soubory, které jsou často aktualizovány nebo soubory, které nejsou dlouho čteny. To znamená, že se algoritmus

snaží spojit obě silné stránky algoritmů LRU a LFU. Pro prvotní určení hodnot přístupů je třeba podpora ze strany serveru, který předává klientovi své hodnoty o přístupech k souboru.

2.10.2 Algoritmus výběru souboru – hybrid LRU a LFU

Tento algoritmus je vyvíjen jako součást projektu KIVFS. Za cíl si dává sjednotit výhody dvou algoritmů LRU a LFU. V této práci je implementován za účelem jeho otestování.

Náš algoritmus používá oba algoritmy LRU a LFU. Každý tento algoritmus sám o sobě nám dává číslo, který udává prioritu pro smazání souboru z cache. Výsledné číslo našeho algoritmu pak je součet těchto dvou čísel. Při součtu těchto dvou čísel se navíc může za pomoci koeficientů upravovat, jestli se upřednostňuje první nebo druhý algoritmus. U priorit znamená 0 nejhorší prioritu a naopak číslo 65535 bude znamenat nejvyšší prioritu.

$$P_{výsledné} = K_1 . P_{LRU} + K_2 . P_{LFU}$$

K_1 je koeficient pro ovlivnění významu priority LRU.
P_{LRU} je vypočtená priorita pomocí algoritmu LRU
K_2 je koeficient pro ovlivnění významu priority LFU.
P_{LFU} je vypočtená priorita pomocí algoritmu LFU

Hodnoty koeficientů K_1 a K_2 jsou součástí zdrojového kódu a mohou nabývat celočíselných hodnot. Jejich hodnoty určeny na základě experimentu. Zadaná velikost ovlivňuje význam přiřazeného algoritmu. Ovladač byl otestován pro různé variace hodnot za účelem nalezení vhodných konstant.

Výpočet priority algoritmu LRU

U každého souboru je nutné uchovávat informaci o čase, kdy byl daný soubor naposledy použit. K tomu je možné použít například systémový čas, který udává počet ms od 1970. Systém si také pamatuje konkrétní čas nejdéle nepoužívaného souboru. Při každém přístupu do cache se pak musí aktualizovat priority LRU u všech souborů za pomocí lineární interpolace. Nejdéle nepoužívaný soubor pak má hodnotu nula a naopak naposledy použitý soubor má hodnotu priority 65535.

$$P_{LRU\ souboru} = (T_{soubor} - T_{nejstarší\ soubor}) . \frac{65535}{T_{nejnovější\ soubor} - T_{nejstarší\ soubor}}$$

T_{SOUBOR} je čas, kdy byl naposledy použit soubor, u kterého počítáme prioritu.
$T_{NEJSTARŠÍ\_SOUBOR}$ je čas souboru, který nebyl nejdelší dobu použit.
$T_{NEJNOVĚJŠÍ\_SOUBOR}$ je čas souboru, který byl použit naposledy.

Z důvodu efektivity je vhodné, aby si cache pamatovala hodnoty $T_{NEJSTARŠÍ\_SOUBOR}$ a $T_{NEJNOVĚJŠÍ\_SOUBOR}$ nebo je byla schopna rychle získat.

Výpočet priority algoritmu LFU

Když přidáme nový soubor do cache, tak mu nastavíme hodnotu LFU na 0, protože nebyl ještě ani jednou používán v cache. Což zvýhodňuje soubory, které jsou již delší dobu v cache. Proto je nutné hodnotu P_{LFU} na začátku odhadnout. To se dá provést pomocí informací o počtu přístupů získaných ze serveru. Informaci o počtu přístupů k souboru na serveru získáváme pro čtení a zápis zvlášť. Při odhadu dochází k zohlednění, zda nedochází často k zápisu do daného souboru. Takové soubory mají nižší prioritu, protože vyžadují vyšší režii při udržování konzistence dat.

Prvotní P_{LFU} spočteme pomocí vzorce:

$$P_{LFU} = \frac{Hity\ čtení_{server}}{Celkové\ hity_{server}} . 65535$$

Hity čtení$_{server}$ je získaná hodnota globálních hitů pro čtení získaná ze serveru
Celkové hity$_{server}$ je suma hitů pro čtení a pro zápis ze serveru

Nyní máme vypočtenu prioritu. Takže víme, jak se k souboru chovat. Pro další výpočty je vhodné, abychom ještě odhadli počet hitů v cache, protože další výpočty P_{LFU} již budeme provádět z lokálních statistik cache.

$$Hity\ čtení_{klient} = \frac{Hity\ čtení_{server} - Hity\ zápis_{server}}{Celkové\ hity_{server}} . Celkové\ hity_{klient} + 1$$

Hity čtení$_{server}$ a Hity zápis$_{server}$ jsou hodnot, které dostaneme ze serveru
Celkové hity$_{klient}$ je celkový součet hitů všech souborů u klienta
Celkové hity$_{server}$ je celkový součet hitů všech souborů na straně serveru

Při každém dalším přístupu je hodnota hitů inkrementována.

Při každém dalším přístupu se aktualizuje hodnota LFU pomocí lineární interpolace podle vzorce:

$$P_{LFU} = (\sum hitů\ soubor - \sum nejméně\ hitů\ soubor) . \frac{65535}{\sum nejvíce\ hitů\ soubor - \sum nejméně\ hitů\ soubor}$$

$\sum_{HITŮ\ SOUBOR}$ je hodnota, která udává, kolikrát byl soubor použit klientem (zároveň může zohledňovat data ze serveru).
$\sum_{NEJMÉNĚ\ HITŮ\ SOUBOR}$ je statistická hodnota, která slouží pro interpolaci a udává, kolikrát byl použit soubor s nejmenším počtem hitů.
$\sum_{NEJVÍCE\ HITŮ\ SOUBOR}$ je statistická hodnota, která udává, kolikrát byl použit soubor s největším počtem hitů.

Je vhodné kvůli optimalizaci, aby cache uchovávala hodnoty $\sum_{\text{NEJMÉNĚ HITŮ SOUBOR}}$ a $\sum_{\text{NEJVÍCE HITŮ SOUBOR}}$. Při změně těchto hodnot je nutné, aby byly ty hodnoty aktualizovány. Dále je dobré uchovávat hodnotu $\sum_{\text{CELKOVÉ\_ČTENÍ\_KLIENT}}$, která slouží pro prvotní odhad počtu hitů pro nově příchozí soubor do cache a zároveň tuto hodnotu ihned aktualizovat.

Pokud má do cache přijít nový soubor a již se do ní nevejde, tak se z cache odstraní soubor z nejhorší prioritou. Podle priority se z cache odstraňují soubory tak dlouho dokud v ní není dostatek místa pro nový soubor.

Příklad

V cache jsou čtyři soubory a koeficienty K1 = 1, K2 =1:

A – počet hitů 5, naposledy přistupován 5, $P_{LRU} = 0$; $P_{LFU} = 65535$; $P_{celková}=65535$
B – počet hitů 2, naposledy přistupován 15, $P_{LRU} = 65535$; $P_{LFU} = 0$; $P_{celková}=65535$

Příchází nový soubor do cache:
C – $ReadHits_{server}=50$, $WriteHits_{server}=2$, $GlobalHits_{server}=1500$, čas přístupu = 30
C: $ReadHits_{klient} = ((50-2)/1500) * 7 + 1 = 1,224$

Nový stav cache:

A – počet hitů 5, naposledy přistupován 5, $P_{LRU} = 0$; $P_{LFU} = 51738$; $P_{celková}=51738$
B – počet hitů 2, naposledy přistupován 15, $P_{LRU} = 26214$; $P_{LFU} = 13796$; $P_{celková}=40010$
C – počet hitů 1,224, naposledy přistupován 30, $P_{LRU} = 65535$; $P_{LFU} = 63014$; $P_{celková}=128549$

V tomto případě by se v případě potřeby odstranil z cache soubor B.

Nyní bude soubor C znovu přistupován v čase 50:

C: počet hitů + 1 = 1,224 + 1 = 2,224

Nový stav cache:

A – počet hitů 5, naposledy přistupován 5, $P_{LRU} = 0$; $P_{LFU} = 51738$; $P_{celková}=51738$

B – počet hitů 2, naposledy přistupován 15, $P_{LRU} = 14563$; $P_{LFU} = 13796$; $P_{celková}=28539$

C – počet hitů 2,224, naposledy přistupován 50, $P_{LRU} = 65535$; $P_{LFU} = 4893$; $P_{celková}=70428$

V tomto případě by se v případě potřeby opět odstranil z cache soubor B.

2.11 Offline operace

K provozu distribuovaného souborového systému je nutné mít funkční připojení k internetové síti, které nemusí být vždy dostupné. I tak může být dobré uživateli zpřístupnit alespoň částečnou funkcionalitu. Toto se nazývá offline operace.

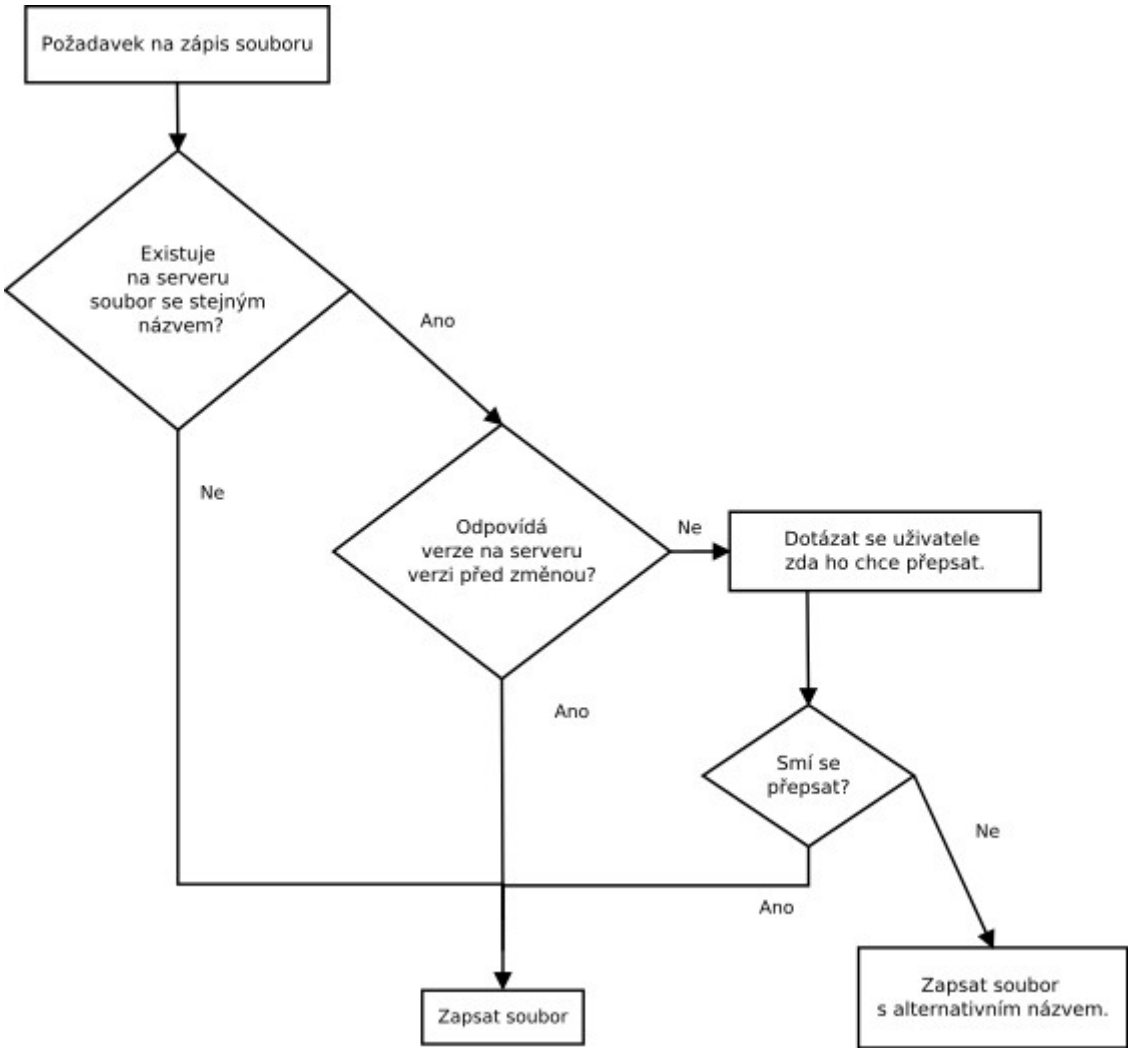
V tomto režimu je uživateli zpřístupněn část souborového systému ze serveru, kterou má uživatel uloženou ve své lokální cache. Navíc je uživateli dovoleno provádět změny včetně rušení a vytváření souborů. Ovladač musí vézt žurnál o všech změnách a v momentě dostupnosti serveru musí všechny změny přenést na server.

Použití cache v offline režimu způsobuje další komplikace při nahrávání změněných souborů na server. Při nahrávání souboru na server po lokální úpravě v offline režimu může dojít k následujícím situacím:

- verze souboru na serveru je stejná jako verze před lokální změnou, nebyl tedy změněn a může být přepsán
- verze souboru na serveru se změnila, klient se dotáže uživatele na další akci
- soubor se stejným názvem na serveru ještě neexistuje, tudíž se na serveru vytvoří

Znázornění řešení průběhu nahrávání souboru ze žurnálu na server je formou diagramu na obr. 8.

Pokud se na serveru nachází starší verze, tak může klient soubor na serveru přepsat. V opačném případě, pokud je na serveru novější verze souboru, tak není možné



Obrázek 8: Zápis souboru ze žurnálu na server

soubor na serveru ihned přepsat, protože by uživatel mohl přijít o změny, které jsou uloženy na serveru. Proto je nutné, aby klient informoval uživatele o nastalé situaci a vyžádal si od něj řešení, které spočívá buď v přepsání souboru na serveru, pokud uživatel ví, že ho změny nezajímají a nebo ponechat verzi na serveru.

3 Praktická část

Ovladač je realizovaný v jazyce ANSI/C. Cílem práce bylo jeho aktualizování na novou verzi KIVFS serveru a nové verze linuxového jádra a FUSE. Dalším cílem bylo ovladač rozšířit o klientskou cache a podporu offline operací, což popisují následující kapitoly.

3.1 FUSE

V této práci je pro vytvoření ovladače použito knihovny FUSE, jejíž bližší popis se nachází v kapitole 2.5. FUSE bylo vybráno jako nová možnost proto, že má stabilnější API než samotné Linuxové jádro, a proto by neměl nastávat problém při přechodu na novější verze jader. Výkonnost FUSE je navíc pro distribuovaný souborový systém dostatečná.

3.2 Zprovoznění FUSE

Samotné FUSE do systému můžeme nainstalovat buď ručně ze zdrojových kódů, pak je nutné v adresáři s rozbalenými staženými zdrojovými kódy spustit následující sekvenci příkazů:

`./configure` - provede konfiguraci a přípravu pro překlad

`make` - přeloží zdrojové kódy

`make install` - nainstaluje binární soubory

`modprobe fuse` - zavede zkompilovaný jaderný modul do jádra, vytvoří v systému zařízení /dev/fuse

Význam jednotlivých příkazů jistě není třeba dále vysvětlovat. Jen dodám, že příkaz „modprobe fuse“ zavede zkompilovaný jaderný modul do jádra, vytvoří v systému zařízení /dev/fuse. Uvedený postup počítá s tím, že FUSE bude do jádra zavedeno jako modul, druhou možností je zakompilovat FUSE přímo do jádra.

Nebo můžeme FUSE do sytému nainstalovat standardní cestou prostřednictvím balíkového systému. Jména balíků se budou odlišovat podle použité distribuce. Pro Debian jsou to balíky libfuse2 a fuse-utils pro běh ovladačů. A balík libfuse-dev pro vývoj a překlad ovladače.

Pro vývoj je nutné nainstalovat vývojové knihovny :

```
apt-get install libfuse-dev
```

Pro to, aby uživatel mohl používat vlastní ovladač, je nutné v závislosti na použitém systému provést některé další kroky.

Je nutné, aby uživatel, který není rootem, byl členem ve skupině fuse. Pokud vytváříme za tímto účelem nového uživatele v systému, tak můžeme užít následující příkaz:

```
useradd -G fuse username
```

Pokud chceme do skupiny fuse přidat v systému již existujícího uživatele, tak použijeme tento příkaz

```
usermod -a -G fuse username
```

Je dobré poznamenat, že tato změna se projeví až při dalším přihlášení uživatele.

V uživatelsky přívětivých distribucích Linuxu, jakou je například Ubuntu, je zpravidla uživatel členem skupiny fuse automaticky. Naopak v pokročilejších distribucích, jako je například Debian, je nutné se o to postarat ručně.

Pokud uživatel není členem skupiny fuse, tak to zjistíme při spuštění ovladače tak, že dostaneme chybu o odmítnutí přístupu k zařízení /dev/fuse.

V systému nalezneme soubor /etc/fuse.conf, ve kterém se nachází konfigurace FUSE. Možné volby, které je možné nastavit, se v průběhu času mění. Za zmínku stojí mount_max, ve kterém můžeme nastavit maximální možný počet připojení, které může uživatel vytvořit. Standardně je to v tuto chvíli 1000. Nakonec je dobré zmínit volbu user_allow_other, která pokud je uvedená, tak zpřístupní uživateli dvě nové volby pro připojení souborového systému. Jde o allow_other a allow_root, jejich význam je vysvětlen níže.

Soubor /etc/fuse.conf může vypadat následovně:

```
mount_max = 1000
user_allow_other
```

Při překladu zdrojového kódu je nutné přilinkování knihoven FUSE. To je nejlépe provést pomocí utility pkg-config.

Příkaz pro překlad pak bude vypadat následovně:

```
gcc -o driver main.c  `pkg-config fuse --libs --cflags`
```

Příkaz `pkg-config fuse --libs` je nutné uvézt při linkování programu, jeho výstupem je seznam knihoven, které jsou nutné při linkování výsledného programu.

Příkaz `pkg-config fuse --cflags` je nutné uvádět při překladu jednotlivých modulů programu. Jeho výstupem je cesta k hlavičkovým souborům FUSE a parametry překladu, typicky se nastavuje `-D_FILE_OFFSET_BITS=64`. Tento parametr vynutí používání 64bitových variant souborových systémových volání, například nahrazení off_t za off64_t. Cílem je umožnění práce se soubory většími než 2GB.

Výsledkem překladu je spustitelný soubor. Připojení nového souborového systému naším ovladačem provedeme jeho spuštěním. Jako parametr je nutné uvést programu cestu, kam se v souborovém sytému má nový svazek připojit.

```
./driver /mnt/mountpoint
```

Odpojení je možné provézt buď klasicky pomocí příkazu `umount` a nebo pomocí příkazu `fusermount` z programovacího balíku FUSE.

```
fusermount -u /mnt/mountpoint
```

3.3 Programování pro FUSE

Základní princip FUSE pro obsluhy jednotlivých volání operačního systému na souborový systém je založen na takzvaných callbacků. Součástí FUSE je definice struktury `fuse_operations`. Tato struktura obsahuje ukazatele pro všechny knihovnou podporované souborové akce. Úkolem programátora je tuto strukturu naplnit ukazateli na své funkce, které budou provádět požadovanou činnost. Takto vytvořená struktura se nakonec předá při volání `fuse_main()` jako jeden z parametrů této funkce.

Hlavní výhoda tohoto přístupu stojí zejména na snadnosti definice toho, co chceme podporovat. Například pokud bychom chtěli vytvořit speciální souborový systém určený pouze pro čtení, protože by jeho úkol byl jen zpřístupňovat nějaké údaje formou souborového sytému. Tak nebudeme do struktury `fuse_operations` ukládat ukazatele na funkce pro zápis a další. Vždy pouze implementujeme tu množinu funkcí, kterou chceme podporovat naším ovladačem. Operace, které nechceme využívat, tedy prostě nevyplníme a FUSE automaticky tyto operace znepřístupní.

Ačkoliv je naprostá většina funkcí volitelná, tak některé jsou povinné jako například `.getattr`.

Zde je uveden příklad vytvoření této struktury:

```
static struct fuse_operations kivfs_oper = {
      .getattr = fs_getattr,
      .readdir = fs_readdir,
      .mknod = fs_mknod,
      .mkdir = fs_mkdir,
      .rmdir = fs_rmdir,
      .unlink = fs_unlink,
      .rename = fs_rename,
      .open = fs_open,
      .read = fs_read,
      .write = fs_write,
      .release = fs_release
};
```

Parametr vlevo je proměnná, kam ukládáme ukazatel na naši funkci, která je pravým parametrem výrazu.

Uvedený příklad slouží pouze pro ilustraci a není kompletní pro všechny možné operace a nejsou ani uvedeny všechny operace z ovladače pro KIVFS.

Jména jednotlivých funkcí jsou většinou samovysvětlující. Proto si je projdeme jen krátce.

- `.getattr` slouží k získání veškerých informací o položce předané jako parametr, položkou může být cesta k souboru nebo adresáři.
- `.readdir` s pomocí této funkce systém získává seznam položek žádaného adresáře. Pro zjištění všech požadovaných informací následně se na každou položku zavolá `.getattr`.
- `.mknod` se vytvoří nový prázdný soubor, do kterého je možné zapisovat pomocí `.write` a číst pomocí `.read`.

- `.release` je vždy voláno při zavření souboru a případně ještě před ním `.flush` pokud je implementováno.
- `.mkdir` vytváří adresář
- `.rmdir` maže adresář
- `.rename` pro přesouvání souborů či jejich přejmenování slouží tak jako příkaz mv v linuxu
- `.unlink` smaže soubor

Když máme hotové všechny požadované funkce a naplněnou strukturu `fuse_operations`, tak můžeme zavolat funkci `fuse_main()`, která nás uvede do smyčky obsluhy požadavků a my ztrácíme kontrolu nad programem.

```
fuse_main(argc, argv, &fs_oper, NULL)
```

Typicky se funkci `fuse_main()` předávají parametry příkazové řádky, ale je možné je vytvořit i ručně. V každém případě je nutné, aby první parametr obsahoval počet parametrů, který chceme FUSE předat, v tomto případě jako `argc`. A druhý parametr volání funkce musí být ukazatel na pole řetězců jednotlivých parametrů, zde jako `argv`.

Značení `argc` a `argv` je úmyslně vybráno proto, že se tak typicky označují v jazyce C parametry příkazové řádky získané na začátku funkce `main()`.

Je nezbytně nutné, aby součástí seznamu parametrů byla cesta, kam se má souborový systém připojit. Ale je možné uvést některé další volitelné parametry. Při vývoji je například velmi užitečný parametr „-d“, který uvede FUSE ovladač do debug modu. Při něm vidíme v terminálu klasický výstup z programu a navíc FUSE přidává své vlastní výpisy, ze kterých je patrný průběh vykonávání souborových operací a jejich výsledky.

Příklad výpisu fuse v debug modu:

```
unique: 83, opcode: READDIR (28), nodeid: 1, insize: 80

unique: 83, error: 0 (Success), outsize: 16
unique: 85, opcode: LOOKUP (1), nodeid: 1, insize: 48
unique: 85, error: -2 (No such file or directory), outsize: 16
```

Každý požadavek má své vlastní jedinečné číslo uvedené za klíčovým slovem `unique`. Následuje parametr `opcode` vyjadřující o jakou operaci se jedná. Následují další hodnoty dle typu operace. Následuje výsledek operace.

Na našem příkladě vidíme dvě operace. První je čtení adresáře a skončila úspěšně. Druhá `LOOKUP` znamená pokus o zjištění informací o adresáři nebo souboru. Ta skončila neúspěchem, protože položku nepodařilo najít.

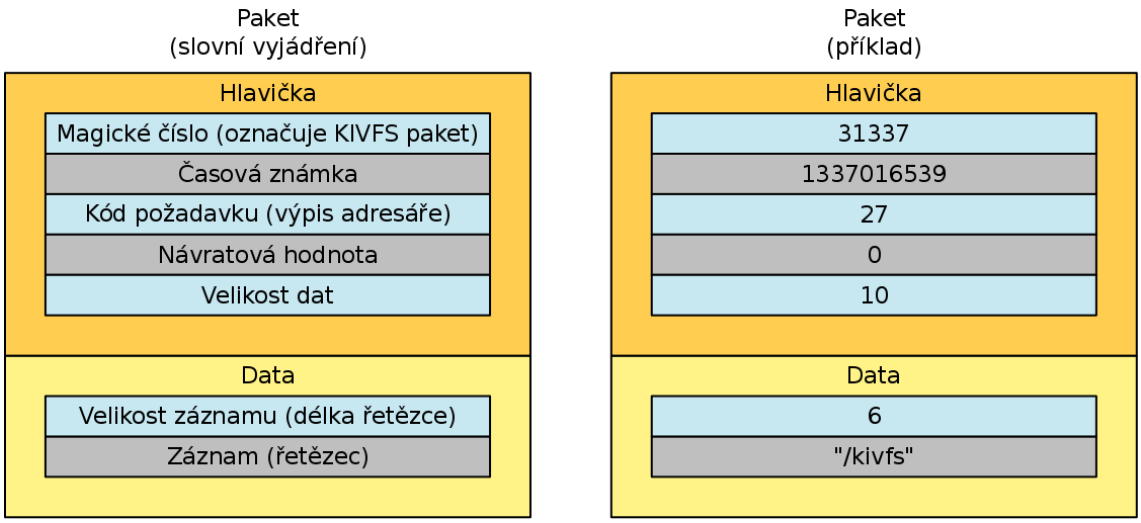
Standardní chování FUSE je takové, že každý připojený souborový systém je přístupný pouze uživateli, který ho připojil. Pokud je v konfiguraci FUSE uvedena volba user_allow_other, tak je možné toto chování změnit prostřednictvím následujících dvou parametrů allow_other a allow_root. Parametr allow_root umožní přistupovat na takto připojený souborový systém navíc i uživateli root a parametr allow_other umožňuje přistupovat všem uživatelům v systému.

Třetím parametrem funkce `fuse_main()` je ukazatel na strukturu s ukazateli na funkce souborového sytému.

3.4 Protokol KIVFS

Pro komunikaci se KIVFS serverem se používá klasický potvrzovaný protokol TCP. Proud dat mezi serverem a klientem se skládá z povinné hlavičky a volitelných dat, které se vyskytují v závislosti na tom co je obsahem hlavičky.

Hlavička má pevnou velikost, obsahuje veškeré nutné údaje o požadavku: časovou značku, typ požadavku, délku následujících dat nebo návratovou hodnotu. Stejný mechanismus pracuje oběma směry. Pokud na server pošleme požadavek na smazání souboru, tak odešleme na server hlavičku a data s cestou k souboru, server odpoví hlavičkou, ve které je návratový kód, ze kterého můžeme zjistit případnou chybu a její typ. Grafické znázornění packetu se nachází na obr. 9.



Obrázek 9: Formát packetu KIVFS[26]

Přenos dat souborů probíhá ve svém vlastním spojení pro každý přenášený soubor. Pokud chce klient přenést data, tak pošle na server požadavek o otevření souboru, na který server v odpovědi zašle ip adresu a port, na který se klient připojí. Na novém spojení klient opět posílá požadavky za pomoci hlaviček, ve kterých je uveden typ operace: čtení nebo zápis a počet bytů. Pokud tento požadavek skončí úspěchem, tak může následně přijmout nebo odeslat předem deklarované množství dat. Nakonec je nutné přijmout od serveru zprávu o úspěchu přenosu souboru. Práce se souborem je stavová, tak jak jsme zvyklí z klasické práce se soubory. V případě potřeby je možné změnit pozici v souboru pomocí odeslání požadavku fseek. Nakonec je nutné před zavřením souboru na serveru uzavřít vytvořené spojení.

Síťovou vrstvu klienta má na starosti knihovna kivfs-core, která je součástí projektu KIVFS.

Požadavek se vytvoří voláním funkce kivfs_request().

```
kivfs_request(id, command, "%s", path);
```

Prvním parametrem je id číslo uživatele, který provádí danou operaci. Druhým parametrem je typ požadavku např `KIVFS_OPEN` pro otevření souboru na serveru. Třetím parametrem je řetězec, který definuje množství a typy parametrů. Typy parametrů jsou definovány stejným způsobem jako u funkce `printf()`. Nakonec funkce jsou uvedeny všechny parametry.

Odeslaní požadavku na server je možné například provést voláním následující funkce, tato navíc speciálně automaticky přijme i odpověď.

```
kivfs_send_and_receive(session, request, &response);
```

První parametr je odkaz na spojení. Druhý parametr je námi dříve vytvořený požadavek viz. výše. A posledním třetím parametrem je adresa, kam se uloží odpověď.

Implementace všech výše uvedených funkcí nalezneme v knihovně kivfs-core, konkrétně v souborech `kivfs-net.c` a `kivfs-session.c`.

3.5 Implementace základních operací

Účelem následující části není vyčerpávající popis všech podporovaných operací, ale pouze přiblížení konkrétních realizací na některých netriviálních případech. Kompletní zdrojový kód pro síťovou komunikaci se serverem a práci se soubory se nachází v souborech `kivfs_filesystem_operation.c`, `fsoperation.c`, `buffer.c`, `offlineOperation.c` a `cache.c`.

3.5.1 Implementace výpisu adresáře

Kompletní zdrojový kód implementující souborové operace pro fuse se nacházejí v souboru `fsoperation.c` a využívají síťové funkce ze souboru `kivfs_filesystem_operation.c`.

Výpis adresáře ve FUSE probíhá za pomoci následujících dvou funkcí.

```
int readdir(const char *path, void *buf, fuse_fill_dir_t filler,
off_t offset, struct fuse_file_info *fi)
int getattr(const char *path, struct stat *stbuf)
```

Pro zjištění obsahu adresáře slouží funkce `readdir()`, zjišťovaný adresář je dán cestou v proměnné `path`.

Jsou možné dvě různé varianty implementace této funkce. Buď se vrací celý obsah adresáře najednou, pak se ignoruje parametr offset a funkce bude vracet nulu. Nebo se obsah adresáře může vracet po částech. Pak se zjistí, odkud pokračovat z hodnoty parametru offset.

Položky se ukládají do proměnné `buff`. Plnění tohoto parametru se provádí za pomoci vestavěné funkce `filler()`. Hlavním účelem této funkce je informovat o existenci všech položek v adresáři. O podrobných informacích jednotlivých položek má za účel informovat následující funkce pojmenovaná `getattr()`.

Funkce `getattr()` slouží ke zjištění veškerých informací o položce dané cestou. Může se jednat o soubor tak o adresář.

Když v systému vznikne požadavek na vypsaní adresáře tvořeným v sytému FUSE, tak dojde k zavolání funkce `readdir()`. Tím se získá seznam položek adresáře a následně dojde na zavolání funkce `getattr()` na každou jednotlivou položku adresáře a až teprve pak je možné adresář zobrazit.

Odpověď na požadavek `KIVFS_READDIR` ze serveru KIVFS vrací kromě seznamu jednotlivých položek i kompletní seznam informací ke každé z nich. Je to mnohem efektivnější, než žádat prostřednictvím sítě internet o každou položku zvlášť a posílat tak mnoho jednotlivých požadavků. Z tohoto důvodu si klient pamatuje obsah posledního zobrazovaného adresáře a následující volání `getattr()` jsou obsloužena z této triviální cache, pokud směřuje do toho samého adresáře.

Z výše uvedeného vyplývá, že v případě vypisování adresáře je počet systémových volání přímo úměrný počtu položek v něm obsažených. Například pro adresář s deseti soubory a dvěma adresáři proběhne nejméně třináct volání. Jednou `readdir()` pro získání obsahu adresáře, poté desetkrát `getattr()` pro soubory a dvakrát `getattr()` pro adresáře. Pokud požadavky obsloužíme z jednoho volání `readdir()` přes síť a následně všechny volání `getattr()` obsloužíme již lokálně ze dříve získaných informací, tak dostaneme urychlení výpisu adresáře o několik řádů.

3.5.2 Realizace přenosu dat mezi klientem a serverem

Prostřednictvím počítačové sítě je z důvodu efektivity vhodné posílat data souboru po co možná největších blocích. Ovšem operační systém pracuje s bloky proměnné velikosti o minimální velikosti 4KB, což je k přenosu nevhodná velikost, kvůli vysoké režii. Navíc protokol KIVFS vyžaduje, aby si klient požádal o přenos každé části zvlášť a následně počkal na potvrzení o ukončení přenosu. To dále zvyšuje režii a tak je vhodné posílat data souborů skrz síť po co největších částech.

V rámci možností systému je možné provézt nastavení parametrů. Od kernelu verze 2.6.26 a FUSE 2.8.0 je možné nastavit větší maximální velikost bloků na větší, než je standardní. Tuto možnost je nutné aktivovat pomocí parametrů, které předáváme FUSE ve funkci `fuse_main()`.

Nejprve je potřeba tuto možnost aktivovat pomocí „-o big_write“ a následně můžeme horní meze nastavit pomocí voleb max_write a max_read.

Ovšem systém i tak může posílat malé bloky. Prozatím jediný způsob jak změnit minimální hranici, je úprava zdrojového kódu FUSE. Ve zdrojovém souboru fuse_kern_chan.c je třeba změnit hodnotu makra `MIN_BUFSIZE` na požadovanou hodnotu. To ovšem vede k nutnosti rekompilace knihovny na klientské stanici, což není často možné. Proto byl vytvořen v paměti RAM buffer pro ukládání dat a jejich shromažďování do větších celků.

3.5.3 Návrh vyrovnávací paměti

Součástí implementace přenosu souborů je i správa veškerých probíhajících transferů souborů z/na server. Na programování takové struktury se nejlépe hodí využití prvků objektového programování. Protože jazyk ANSI C podporu pro objekty neobsahuje, tak je nutné zavést pravidla pro psaní kódu, která jsou pak dodržována. Níže popisovaný zdrojový kód se nachází v souboru `buffer.c`.

Pro psaní objektového kódu v C je možné využít jedno z několika možných přístupů. Zde je využit přístup, při kterém se jako první parametr uvádí ukazatel na strukturu, ke které funkce náleží. V našem případě je pojmenován `this` jako výsledek inspirace jazykem C++. Dále platí, že každá členská funkce struktury musí začínat názvem struktury, za kterým následuje podtržítko a název funkce.

Každá struktura která má fungovat jako objekt, musí mít definovány dvě zvláštní funkce. Je to funkce `init` pro vytvoření struktury nebo přesněji pro alokování veškerých potřebných zdrojů. A funkce `destroy`, která má za úkol uvolnění alokovaných zdrojů.

Jednoduchý příklad reprezentuje struktura Buffer.

```
typedef struct { // struktura
     char *buffer; //ukazatel na pole dat
     uint64_t size; //velikost výše uvedeného pole
     uint64_t index; // index na poslední obsazenou pozici
} Buffer;

void Buffer_init(Buffer *this); //vytvoří buffer
void Buffer_destroy(Buffer *this);  // smaže buffer
```

Jde o jednoduchou strukturu, která obsahuje pole pro data určité pevné délky. Dále udržuje jeho velikost a index, který určuje, jaká část pole je využita. K této struktuře jsou přidruženy nejméně výše dvě uvedené funkce pro vytvoření a uvolnění obsaženého pole.

Strukturu Buffer využívá struktura `TransferItem`, která kromě samotného bufferu pro dočasné uložení dat souboru obsahuje další potřebné položky pro spojení a přenos souboru se serverem. Obsahuje například socket pro přenos, filedeskriptor získaný otevřením souboru, název souboru a aktuální pozici v otevřeném souboru na serveru.

A nakonec strukturu `TransferItem` využívá struktura `Transfers`, ta obsahuje pole pro uchování všech probíhajících přenosů a umožňuje jejich přidávání, rušení a vyhledávání.

3.5.4 Implementace čtení souborů

Kompletní zdrojový kód pro přenosy souborů se nachází v souboru kivfs_filesystem_operation.c.

Pokud programujeme čtení souboru do FUSE, tak je nutné vytvořit implementace pro následující dvě funkce:

```
int open(const char *path, struct fuse_file_info *fi);
int read(const char *path, char *buf, size_t size, off_t
offset, struct fuse_file_info *fi);
```

První funkce má za úkol otevřít soubor. Prvním parametrem je název samotného souboru a druhým parametrem je struktura `fuse_file_info`. Ta mimo jiné obsahuje mód ve kterém máme soubor otevřít. A naopak do ní můžeme uložit získaný filedeskriptor a ten pak získáme z té samé struktury při volání funkcí `read()` a `write()`.

Návratový kód funkce určuje, zda se podařilo soubor otevřít v pořádku a nebo zda došlo k chybě. Pokud dojde k chybě, můžeme systému sdělit prostřednictvím chybového kódu, k jaké chybě došlo. Pokud soubor neexistuje, tak vracíme hodnotu `ENOENT`, pokud k souboru nemáme dovolen přístup v požadovaném módu, tak vrátíme hodnotu `EACCES` a nakonec v případě jakékoliv chyby vracíme `errno`.

Druhá funkce je volána pro čtení dat souboru. Prvním parametrem je název čteného souboru a posledním parametrem je ukazatel na strukturu, kam jsme si uložili filedeskriptor souboru. Dalšími klíčovými parametry jsou ukazatel na pole, kam máme zkopírovat požadovaná data souboru. Hodnoty `offset` a `size` určují, která data jsou požadována. Parametr offset určuje pozici v souboru, od které máme číst a `size` kolik dat od této pozice se přečte.

Pro čtení i zápis tak platí, že nedostáváme žádná volání obdobná funkci fseek, protože není potřeba. Pozici v souboru dostáváme při každém volání. Velikost požadovaných částí není konstantní a v průběhu přenosu se může měnit.

Ukázková implementace pro čtení ze souborového systému, která je využívána při čtení dat z cache.

```
int read( path, buf, size,offset, fuse_file_info *fi)
{
     int fd = open(path, O_RDONLY);
     res = pread(fd, buf, size, offset);
     close(fd);
     return res;
}
```

Každé čtení souboru začíná jeho otevřením na serveru. Pokud se soubor podaří otevřít, tak je získaný filedeskriptor předán FUSE a při každém požadavku je identifikace přenosu realizována pomocí tohoto deskriptoru.

Další postup záleží na tom, zda je aktivována cache. Pokud je cache aktivována, tak je spuštěno nové vlákno, jehož jediným účelem je stáhnout v případě potřeby najednou celý soubor ze serveru do cache. A následné další požadavky na čtení souboru jsou obslouženy z lokální cache,

V opačném případě, pokud cache není aktivována, tak se data souboru stahují na požádání do lokálního bufferu v paměti. Kvůli optimalizaci se data souboru nestahují po požadovaných částech, ale vždy se naplní celý buffer, který odpovídá přibližně několikanásobku velikosti požadavků. Proto se nemusí posílat po síti mnoho malých částí. V případě, že dojde při čtení souboru k přesunu pozice za pomoci fseek, tak se buffer aktualizuje o data na aktuální pozici.

Hlavní změna oproti předchozí implementaci spočívá v tom, že se nyní nemusí čekat až se soubor stáhne celý, ale buď se stahuje průběžně přímo ze serveru a nebo se přečte z cache, ve které se nemusí nutné nacházet celý soubor, ale postačuje pokud se v ní nachází požadovaná část. V čemž se odlišuje od staré verze, ve které nebyla cache a server neuměl přenášet soubor po částech. Díky tomu je uživatel korektně informován o průběhu přenosu za pomoci progress baru v souborovém manažeru.

3.5.5 Implementace zápisu souborů

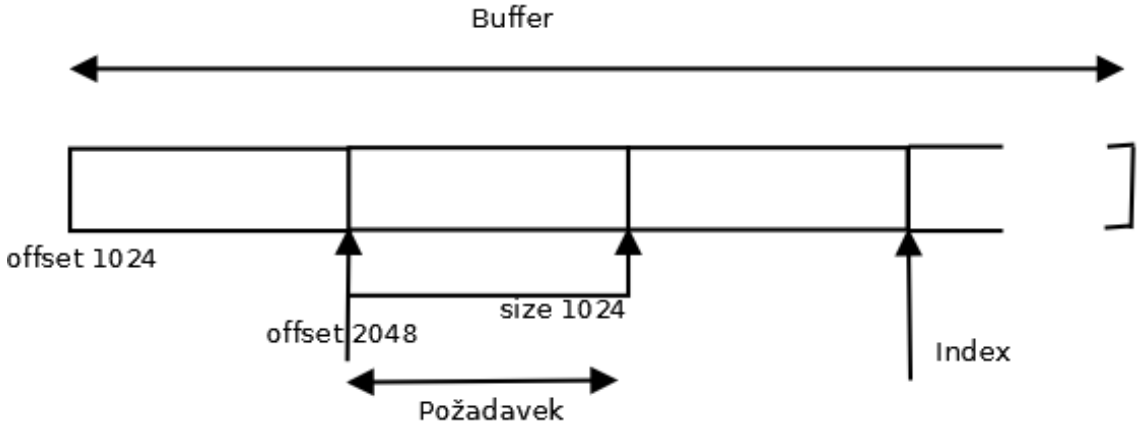
Při zápisu celého souboru na server vždy přibude jedno volání následující funkce oproti jeho čtení.

```
int mknod(const char *path, mode_t mode, dev_t rdev);
```

Funkce `mknod` vytvoří na serveru prázdný soubor.

```
int write(char *path, char *buf, size_t size, off_t offset,
struct fuse_file_info *fi);
```

Funkce write slouží pro zápis dat v parametru `buf` do souboru. Všechny parametry odpovídají stejnojmenným parametrům u funkce `read`.



Obrázek 10: Vyřízení požadavku z bufferu

Ukládání dat probíhá jak do cache, tak do souboru na serveru současně. Ukládání do lokální cache se provádí okamžitě. Zatímco ukládání na server se uskuteční nejprve do pole buffer na obrázku 10 a na server se provádí až když se zaplní, nebo když se soubor uzavře.

3.6 Realizace cache

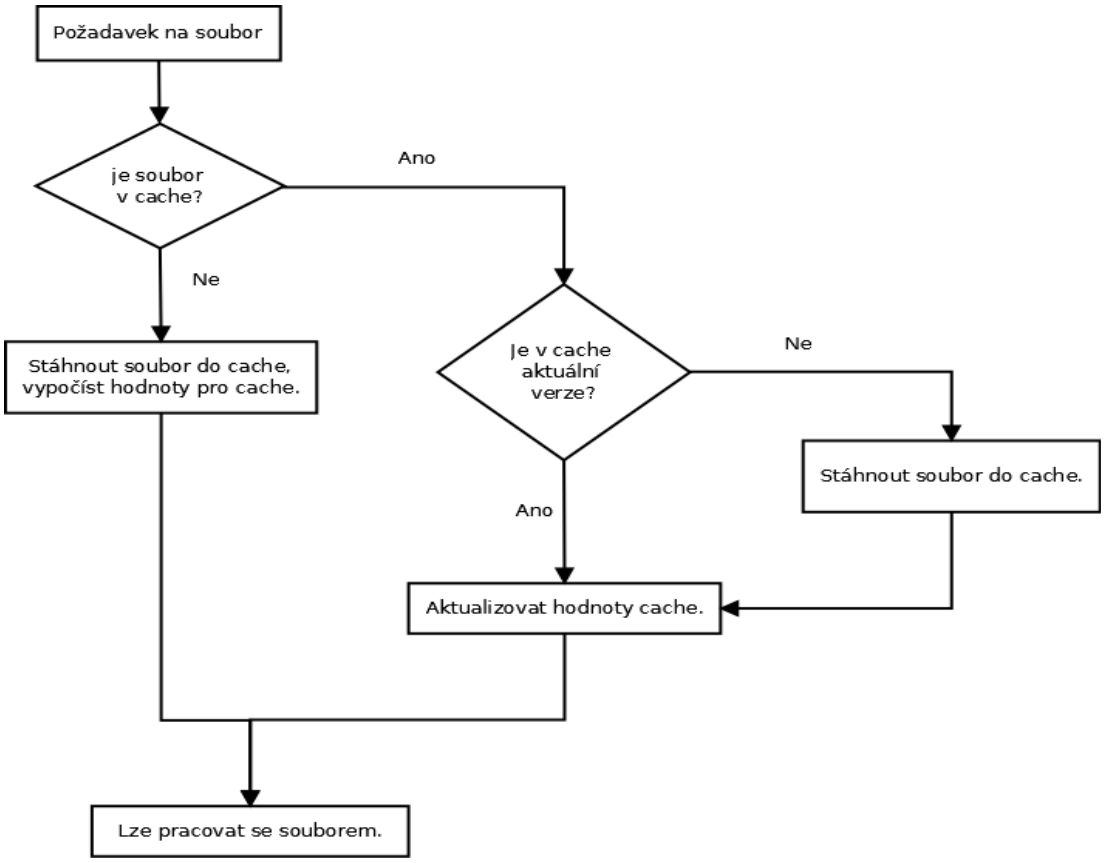
V adresáři s konfigurací, která se nachazí v adresáři .kivfs v domovském adresáři uživatele, se nachází podadresář /cache. V tomto adresáři se nachází veškeré soubory i se svými cestami. Nachází se zde tedy kompletní adresářová struktura se soubory, které jsou součástí cache.

Pro ukládání informací o souborech a přidružených parametrů se využívá kompaktní databáze SQLite. Základní popis a důvody výběru nalezneme v kapitole 2.8.3. Veškeré informace jsou uspořádány do jedné tabulky. Klíčem tabulky je název souboru, ve kterém je obsažena i cesta k souboru. Dále se ukládá velikost souboru, verze souboru, počet přístupů klientem, hodnoty LFU a LRU a několik dalších parametrů přímo vyplývajících z použitých metod pro výběr souboru k odstranění z cache viz. později.

Cache se aktualizuje vždy pří novém přístupu k souboru, který představuje zavolání funkce open(). Při otevření souboru se kontroluje cache jen v případě, pokud soubor není otevřen jen pro zápis. Pokud je soubor otevřen jen pro zápis, tak je v cache vytvořen prázdný soubor a na serveru je zkrácen na nulu otevřením pro zápis. Pokud je soubor otevřen v nějakém jiném módu, tak je třeba nejprve zkontrolovat, zda se nachází v cache. Když se v cache nenachází, tak je stažen ze serveru a do cache přidán. Soubor se stahuje rovnou do cache a je přístupný požadavkům na čtení pro již uložená data.

V opačném případě, pokud se soubor v cache nachází, tak je nutné zkontrolovat lokální verzi proti verzi na serveru. Když se v cache nenachází aktuální verze, tak je třeba ji aktualizovat.

Znázornění přístupu k souboru v cache formou diagramu se nachází na obr. 11.



Obrázek 11: Vývojový diagram otevření souboru

Čtení ze souboru se tedy vždy provádí z rychlé lokální cache, protože je zaručeno, že soubor v ní bude pro čtení přístupný. Zápis do souboru se provádí do cache i na server současně. Pouze zápis na server může mít zpoždění, protože mohou data nějakou dobu být v bufferu, než se provede hromadný zápis požadavků na server po zaplnění bufferu.

Při zavření souboru je potřeba zkontrolovat, zda souhlasí verze souboru na serveru a cache. Synchronizuje se verze klienta s verzi na serveru.

3.7 Výběr souboru pro odstranění z cache

Velikost cache je vždy omezena minimálně velikostí volného místa na disku. Ve skutečnosti je však v klientovi nastavena maximální velikost cache. Pokud již máme nějaké soubory uloženy v cache a je potřeba uložit do cache nový soubor, tak se může stát, že by velikost cache byla překročena. V tomto případě je potřeba nějaký soubor z cache odstranit a nebo více souborů pokud jeden neuvolní dostatek místa. Existuje několik postupů výběru souboru k odstranění, podrobněji byly popsány v teoretické části.

Klient podporuje několik možných postupů výběru souboru pro umožnění srovnání jejich efektivity. Jako základní varianta slouží hybridní metoda mezi LFU a LRU vytvořená pro KIVFS. Další podporované metody jsou Random s náhodným výběrem souboru k odstranění, FIFO, která maže nejstarší soubory a LRU mazající nejdéle nepoužitý soubor.

Vždy, když klient otevře soubor, tak jej přidá do cache, pokud v ní již není. Pokud v cache soubor nebyl, tak se zkontroluje, zda se do ní vejde. Když nastane možnost, že v cache není dostatek místa pro soubor, tak dojde na zavolání funkce:

```
int makeFreespace(uint64 size);
```

Tato funkce má za úkol uvolnit v cache tolik místa, kolik má předáno jako parametr. Pro usnadnění možnosti volby metody a pro minimalizaci nutných úprav při přidání nové metody je v programu vytvořen ukazatel na funkci

```
int (*del)(int);
```

Do tohoto ukazatele se ukládá ukazatel na funkci, která smaže s cache vybraný vhodný soubor. Jako parametr přebírá potřebné místo v cache, které je třeba vytvořit, aby se do ní vešel požadovaný soubor. Některé metody tento parametr mohou ignorovat, jiné se mohou pokoušet smazat soubor s vyhovující velikostí z důvodu optimalizace. V každém případě funkce `makeFresspace()` volá ukazatel na funkci `*del` tak dlouho, dokud v cache není vytvořeno dostatek místa.

Jak již bylo zmíněno, tak standardně je zvolena hybridní metoda LRU a LFU a do ukazatele `*del` je dosazen ukazatel na funkci `int kivfsLruLfu(int size)`.

3.8 Offline režim

Ovladač umožňuje spuštění v takzvaném offline režimu. Při něm je připojena cache a uživateli je tak zpřístupněna alespoň část souborového systému bez toho, aby měl přístup k síti.

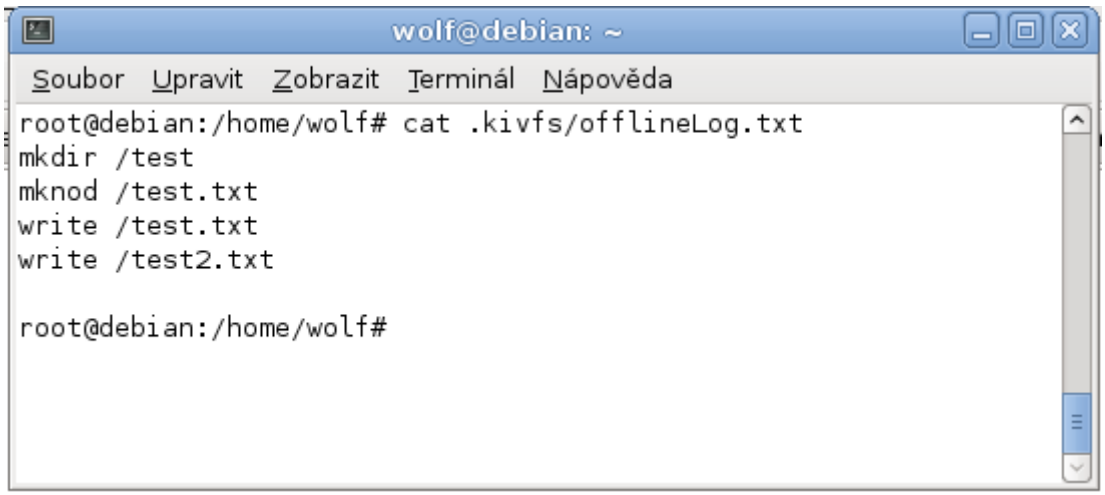
Pro uvedení ovladače do offline režimu je nutné uvézt parametr „offline“ při spuštění.

```
./launcher user passwd /path offline
```

Veškeré změny se provádí a ukládají do cache. Na server se promítají při prvním spuštění launcheru v online režimu od doby, co byly provedeny tyto změny.

Ovladač si v offline režimu loguje veškeré provedené změny do souboru `~/.kivfs/offlineLog.txt` v domovském adresáři uživatele.

Příklad jak může vypadat se nachází na obr.12.



Obrázek 12: Výpis logu

Ve výše uvedeném příkladu vidíme, že během práce v offline režimu byl vytvořen adresář /test v kořenovém adresáři na serveru. Dále byl vytvořen a modifikován soubor test.txt a byl pouze modifikován soubor test2.txt. Nakonec má být

smazán soubor /testfile.

Při spuštění v online režimu ovladač zkontroluje obsah souboru se seznamem změn a promítne je na server. Při této činnosti může dojít ke komplikaci při kontrole verzí souborů.

Pokud na serveru soubor neexistoval, nebo je na serveru ve starší verzi než v cache, tak nedochází k žádné akci a soubor se na serveru aktualizuje na stejnou verzi, jaká se nachází v cache. Naopak pokud je na serveru soubor v novější verzi, než byl v cache před změnou, tak vzniká konflikt, který není možné automaticky vyřešit. Musí rozhodnout uživatel, zda chce na serveru soubor přehrát nebo ponechat.

Za účelem interakce s uživatelem je vytvořeno lokální socketové spojení mezi programem ovladače a programem spouštěče. Po tomto spojení si mohou navzájem zasílat jednoduché zprávy.

Zpráva se vždy skládá nejprve z kódu zprávy, za kterým následuje délka textu zprávy a nakonec následuje textová zpráva. Zprávy jsou v základu dvojího druhu, a to informační a dotazovací.

Pokud ovladač na serveru přepisuje starší soubor, tak o tom uživatele informuje informativní zprávou. Ale pokud se na serveru nachází novější verze souboru, než by měla, tak pošle dotazovací zprávu s dotazem, zda může daný soubor přepsat a uživatel odpoví ano nebo ne.

Umožnění, jak online tak offline režimu vyžadovalo vytvoření dvojích funkcí pro všechny souborové operace. Funkce implementující online operace proti serveru se nacházejí v souboru `fsoperation.c` a k práci využívají funkce z `kivfs_filesystem_operation.c`, který implementuje dané operace nad KIVFS protokolem. Offline verze funkce se nacházejí v souboru `offlineOperation.c`.

Nakonec bylo třeba vyřešit, jak vybírat online nebo offline verze funkcí bez opakujících se `if` podmínek v kódu. Za tímto účelem je použito ještě jedné struktury `fuse_operations`. Ta je naplněna buď online nebo offline verzemi funkci podle parametru. Samotnému FUSE předáváme jen ukazatele na funkce, které pak volají funkce z naší vlastní struktury. To umožňuje do budoucna implementovat změnu režimu za běhu ovladače.

3.9 Zprovoznění ovladače

Pro úspěšný překlad zdrojových kódu je třeba mít instalováno několik vývojových knihoven. Pro systémy Debian a Ubuntu se jedná o následující balíky.

Kromě FUSE je nutné nainstalovat další vývojové balíčky.

Pomocí příkazu:

```
apt-get install  libssl-dev libkrb5  heimdal-dev  libsqlite3-dev
```

V jednotlivých Linuxových distribucích se mohou názvy uvedených balíků lišit, ale v zásadě budou stále podobné.

Ovladač využívá služeb sdílené knihovny kivfs-core a proto je třeba ji nainstalovat, to se provede příkazem `make install` v adresáři knihovny.

Součástí projektu je soubor `Makefile`, takže zkompilování celého projektu se provede tradičním příkazem „make“ v adresáři se zdrojovými kódy. Tím získáme dva spustitelné soubory `launcher` a `klient`.

Pro spuštění je nutné mít zavedený jaderný modul FUSE v jádře, který vytvoří zařízení /dev/fuse. Pokud máme FUSE v systému nainstalováno, tak by měl být modul automaticky zavedený. Pokud není, tak ho můžeme zavést ručně příkazem

```
„modprobe fuse“
```

Dále je třeba, aby měl v domovském adresáři vytvořený adresář .kivfs a v něm uložený konfigurační soubor kivfs.conf a dále vytvořit databázi pro SQLite.

Databázi můžeme vytvořit příkazem:

```
sqlite3 ~/.kivfs/mydb.db
```

Obsah souboru `~/.kivfs/kivfs.conf` vypadá následovně

```
[server1]
servername=147.228.63.63 # IP adresa serveru
serverport=30003 # port
crypt=openssl # openssl pro šifrovanou komunikaci, no pro
nešifrovanou
prior=1 #  priorita záznamu v seznamu
fileblock=2048 # velikost bloku pro síť
```

Záznamů může být uvedeno více pro případ výpadku jednoho nebo více serverů. Na konec souboru s konfigurací je nutné uvést deaktivaci šifrování přenosu, protože to server v současnosti nepodporuje.

```
auth=no
```

Připojení souborového systému se provede spuštěním programu launcher, kterému můžeme volitelně předat parametry.

```
./launcher jméno heslo cesta [offline]
```

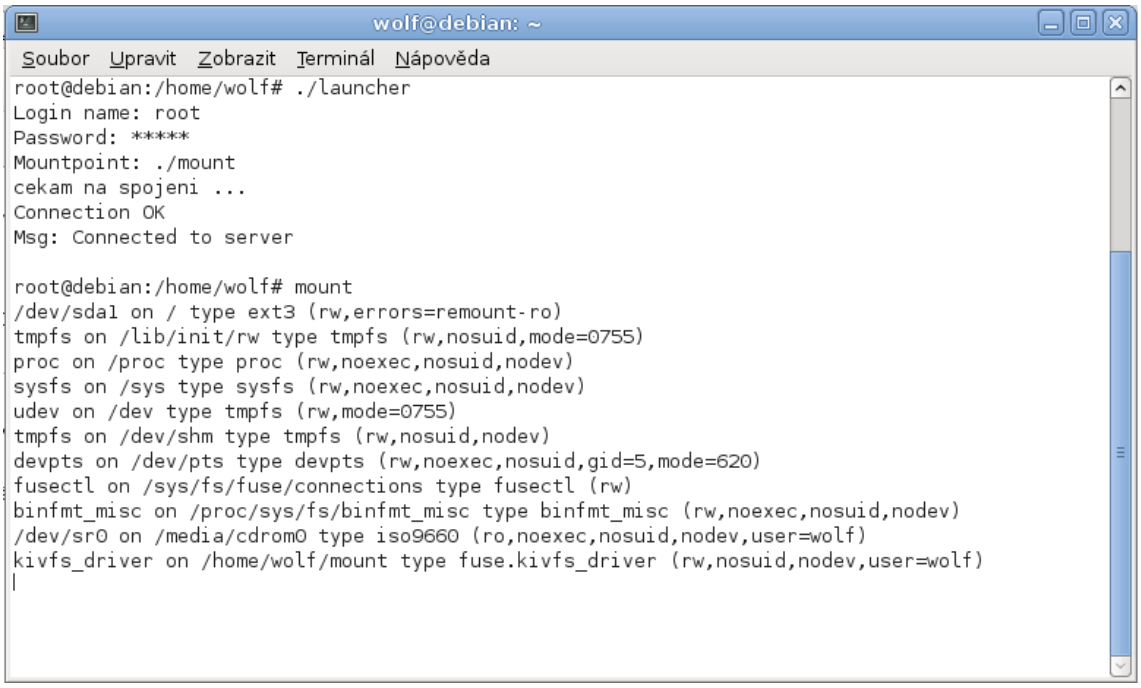
Jako první se uvádí jméno uživatele, po kterém následuje jeho heslo a nakonec cesta, kam se má souborový systém připojit. Úplně poslední parametr uvedený v hranatých závorkách je volitelný a způsobí připojení souborového systému v offline režimu. Pokud uživatel své údaje nezadá jako parametry programu, tak si je program vyžádá sám v době běhu. Tato možnost je doporučena z bezpečnosti, pokud je heslo zadáno jako parametr, tak se zobrazí v seznamu běžících procesů v systému.

```
$ ./launcher
$ Login name: jméno
$ Password: heslo
$ Path: cesta
```

Program launcher po získání všech potřebných údajů vyvolá připojení souborového systému.

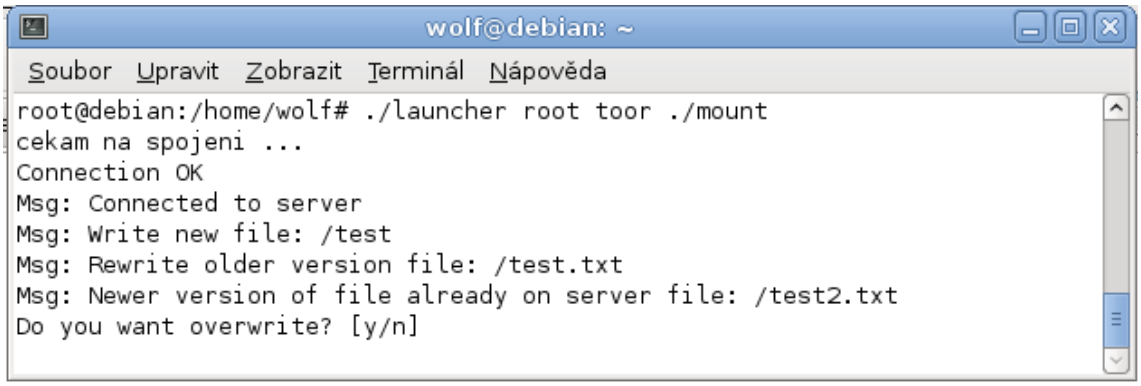
I nepriviligovaný uživatel smí souborový systém připojit a takto připojený souborový systém také kdykoliv odpojit. Odpojení probíhá pomocí nástroje fusermount:

```
fusermount -u cesta
```



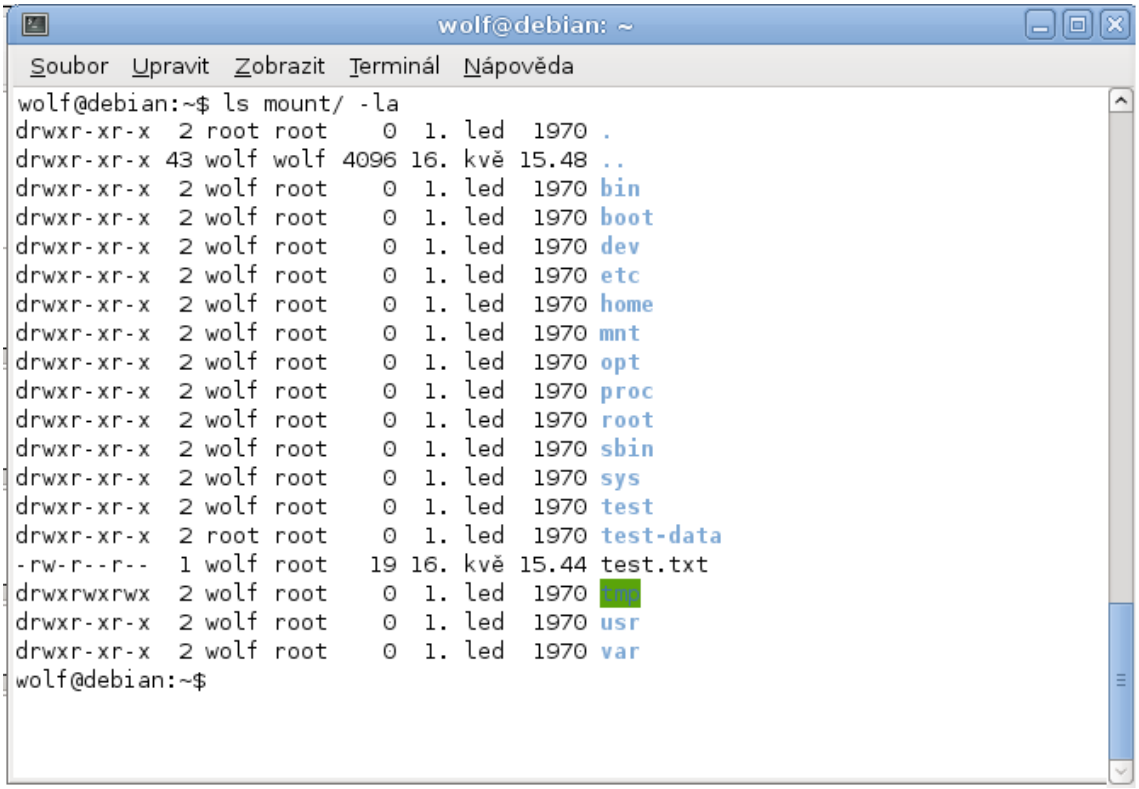
Obrázek 13: Spuštění ovladače

Na obr. 13 je vidět průběh připojení ovladače v online režimu s ručním zadáním údajů a následné spuštění příkazu `mount`, kde vidíme připojený náš souborový systém.



Obrázek 14: Průběh zavádění z offline režimu

Na obr. 14 je znázorněno jak vypadá výpis po spuštění pokud byli provedeny změny v offline režimu, které se zavádí na server. Soubor test.txt byl aktualizován na novější verzi, ale u souboru test2.txt došlo během změn v offline režimu současně i ke změně na serveru a sám uživatel musí rozhodnout zda se smí soubor na serveru přepsat.



Obrázek 15: Výpis adresáře

Na obr. 15 vidíme výpis adresáře po provedených změnách.

3.10 Vytvoření balíčku pro systém Debian

Binární balíček DEB lze vytvořit několika různými způsoby. Můžeme použít automatický postup za pomoci `checkinstall` nebo klasickou ruční přípravu. Ruční vytvoření je sice poněkud zdlouhavější, ale také vždy bezproblémová a proto si tu rozvedeme v následujících odstavcích. Podrobnější popis nalezneme v návodu na

oficiálních stránkách[23].

Následující postup předpokládá bezchybnou kompilaci programu ze zdrojových kódů za pomoci příkazu `make`.

Struktura našeho balíčku vypadá následovně:

```
$ ar tv kivfs-1.0_i386.deb
rw-r--r-- 0/0      4 May 14 13:36 2012 debian-binary
rw-r--r-- 0/0    426 May 14 13:36 2012 control.tar.gz
rw-r--r-- 0/0  30157 May 14 13:36 2012 data.tar.gz
```

Kde `debian-binary` obsahuje informaci o verzi formátu DEB souboru, `control.tar.gz` je archiv s kontrolními soubory a `data.tar.gz` je archiv se samotným programem, jeho binární verze se musí nalézat v adresářové struktuře kam se má nainstalovat.

Musíme si připravit adresářovou strukturu. Na příklad v domovském adresáři si vytvoříme pracovní adresář `~/debian`. V něm si následně vytvoříme adresář `./DEBIAN` do kterého budeme ukládat kontrolní soubory. Dále si musíme vytvořit dva podadresáře `./usr/bin` kam uložíme spustitelné soubory a `./usr/share/man1` kam uložíme manuálové stránky.

Do adresáře `DEBIAN` přídáme soubor s názvem „`control`“, který bude mít následující obsah:

```
Package: kivfs-fuse-client
Version: 0.5.0
Section: Development/Libraries
Priority: optional
Architecture: i386
Depends: openssl, libfuse2, sqlite3, libkrb5-26-heimdal
Maintainer: root
Description: Klient pro KIVFS Jedna se o klient pro
distribuovany souborovy    system KIVFS vyvijeny na katedre
KIV na Zapadoceske univerzite.
```

Význam jednotlivých klíčů je poměrně zřejmý. Veškeré možné hodnoty jsou podrobně popsané v manuálu uvedeném na začátku kapitoly. Důležitý je hlavně klíč `depends`, který obsahuje seznam čárkou oddělených balíčků, na kterých je náš program závislý.

Celá hotová struktura připravená na vytvoření balíčku je součástí zdrojových kódů a nachází se v podadresáři `/tmp`.

Máme-li vše výše uvedené připravé, tak již můžeme vytvořit DEB balíček příkazem:

```
fakeroot dpkg-deb --build $HOME/debian
$HOME/kivfs_fuse-1.0_i386.deb
```

3.11 Testy

Pro zjištění rychlosti přenosu souborů mezi klientem a serverem byly provedeny následující testy za účelem porovnání s předchozí verzí. Následně byly provedeny testy na poměření úspěšnosti algoritmů pro cache.

3.11.1 Přenosová rychlost

Testovací počítač měl následující konfiguraci:

CPU	Intel Core i3-2350M
RAM	4GB
Síť	100/10 Mbits/s
OS	GNU/Linux Debian 6.0.4 Squeeze
Jádro	2.6.32
FUSE	2.8.4

Rychlost disku 75MB/s

Pro dosažení objektivního srovnání s předchozími testy z bakalářské práce byly testy provedeny na shodných datech. Jedná se o celkem tři různé testy s daty vždy o celkové velikosti 100MB.

- V prvním testu se jedná o jeden velký soubor velký 100MB.
- Druhý test přenáší sto souborů o velikosti 1MB.
- V třetím testu se přenáší opět sto souborů, ale tentokrát rozdělených do adresářové struktury.

Testy byli prováděny opakovaně, v níže uvedených tabulkách se nacházejí zprůměrované hodnoty za čelem názorného srovnání s předešlými výsledky v bakalářské práci.

	Stará verze		Nová verze	
	čas	rychlost	čas	rychlost
upload	17s	5,9 MB/s	13s	7,69MB/s
download	10s	10 MB/s	10s	10MB/s

Tabulka 1: Přenos jednoho velkého souboru

	Stará verze		Nová verze	
	čas	rychlost	čas	rychlost
upload	66s	1,6 MB/s	1m14s	1,35MB/s
download	54s	1,9 MB/s	28s	3,57MB/s

Tabulka 2: Přenos velkého počtu souborů

	Stará verze		Nová verze	
	čas	rychlost	čas	rychlost
upload	78s	1,3 MB/s	1m27s	1,15MB/s
download	85s	1,2 MB/s	41s	2,44MB/s

Tabulka 3: Přenos adresářové struktury

Podařilo se zvýšit přenosovou rychlost souborů na server a to i přes to, že se nově posílají soubory po částech. Výjimku tvoří nahrávání malých souborů na server. Práce s malými soubory vždy komplikuje zvýšená režie a nevyužití přenosové kapacity v důsledku toho, že protokol TCP postupně zvyšuje svou přenosovou rychlost. Nahrání souborů na server navíc provází zvýšená režie související se zvýšeným množstvím požadavků od systému na stav souboru. Další režii přidávají nutné operace v souvislosti s cache.

Test probíhal s prázdnou cache a každý soubor se přenášel pouze jednou, proto se nemohl projevit význam cachování. Účelem bylo čisté srovnání rychlosti přenosu souborů. Při nahrávání malých souborů na server došlo k mírnému poklesu rychlosti, ale v reálném nasazení dojde k opakovanému použití některých souborů, díky cache dojde naopak ke zrychlení práce. Testování cache se věnuje následující kapitola.

Pro srovnání přenosu opakujících se souborů s cache a bez cache byl proveden test. Byl proveden na stejných datech jako u testu malých souborů, akorát s tím rozdílem, že se všechny soubory stahovali desetkrát.

	S aktivovanou cache		Bez aktivované cache	
	čas	rychlost	čas	rychlost
download	1m21s	12,35 MB/s	5m17s	3,15MB/s

Tabulka 4: Srovnání přínosu cache na rychlost

Z výše uvedeného vyplývá, že cache může znatelně zrychlit přístup k souborům.

3.11.2 Cache

Cílem tohoto měření je zjistit úspěšnost jednotlivých algoritmů pro výběr souboru k odstranění z plné cache.

Test probíhá na celkem 400 souborech nahraných na serveru. Sto souborů má velikost 500KB, sto souborů 1MB, dalších sto 2MB a posledních sto souborů má 3MB. Soubory jsou pojmenovány čísly od 1 do 400. Na začátku každého měření je prázdná cache. Je vytvořeno celkem 1000 požadavků na přenos souboru směrem ze serveru ke klientu, tak aby se některé soubory opakovaly. Číslo požadovaného souboru je generováno gaussovským rozdělením se středem v 200. Soubory s číslem okolo 200 jsou tedy požadovány nejčastěji.

Hodnota „hit ratio“ se spočítá podle následujícího vzorce:

hit ratio = number of hits / (number of hits + number of miss)

Algoritmus Random

Velikost Cache	Celkový přenos	Ušetřený přenos	Počet Cache Hit	Hit Ratio
8 MB	1522MB	19MB	11	0,01
64 MB	1566MB	78MB	41	0,04
256 MB	1537MB	354MB	151	0,15

Tabulka 5: Metoda Random

Algoritmus FIFO

Velikost Cache	Celkový přenos	Ušetřený přenos	Počet Cache Hit	Hit Ratio
8 MB	1544MB	32MB	16	0,02
64 MB	1548MB	126MB	87	0,09
256 MB	1524MB	539MB	386	0,39

Tabulka 6: Metoda FIFO

Hybridní KIVFS algoritmus mezi LRU a LFU byl testován s třemi různými koeficienty. V prvním případě mají oba koeficienty hodnotu jedna a tudíž je na oba algoritmy brán stejný důraz. Ve druhém případě je kladen větší důraz na LRU a ve třetím na LFU.

Algoritmus KIVFS hybrid mezi LRU a LFU s koeficienty k_1=1, k_2=1

Velikost Cache	Celkový přenos	Ušetřený přenos	Počet Cache Hit	Hit Ratio
8 MB	1514MB	54MB	32	0,03
64 MB	1522MB	227MB	159	0,16
256 MB	1548MB	685MB	466	0,47

Tabulka 7: Metoda KivfsLruLfu 1:1

Algoritmus KIVFS hybrid mezi LRU a LFU s koeficienty k_1=2, k_2=1

Velikost Cache	Celkový přenos	Ušetřený přenos	Počet Cache Hit	Hit Ratio
8 MB	1539MB	27MB	19	0,02
64 MB	1566MB	198MB	132	0,13
256 MB	1576MB	739MB	486	0,49

Tabulka 8: Metoda KivfsLruLfu 2:1

Algoritmus KIVFS hybrid mezi LRU a LFU s koeficienty k_1=1, k_2=2

Velikost Cache	Celkový přenos	Ušetřený přenos	Počet Cache Hit	Hit Ratio
8 MB	1525MB	31MB	25	0,03
64 MB	1495MB	203MB	132	0,13
256 MB	1555MB	669MB	431	0,43

Tabulka 9: Metoda KivfsLruLfu 1:2

Algoritmus LFU s hity

Velikost Cache	Celkový přenos	Ušetřený přenos	Počet Cache Hit	Hit Ratio
8 MB	1554MB	39MB	20	0,02
64 MB	1557MB	235MB	151	0,15
256 MB	1592MB	727MB	482	0,48

Tabulka 10: Metoda LFU s hity

Jednotlivé algoritmy postavené na LRU a LFU mají v tomto typu testu vyrovnané výsledky. Ovšem jednoduché algoritmy typu Random a FIFO vycházejí výrazně hůře.

V tomto testu dopadly výsledky u KIVFS algoritmu velmi podobně pro všechny varianty nastavení koeficientů. V praxi tomu tak nemusí být, každý takovýto test je jen zjednodušeným napodobením reálného chování. Tento test ukázal, že algoritmus velmi dobře obstojí v situaci, kdy jsou některé soubory přistupovány výrazně častěji než jiné v případě, že je v cache dostatek místa pro tyto soubory.

4 Závěr

Úkolem této diplomové práce bylo navrhnout a implementovat rozšíření ovladače pro distribuovaný souborový systém KIVFS o klientskou cache a základní offline operace. Součástí byla také aktualizace ovladače na novou verzi protokolu s využitím jeho nových možností.

Ovladač nově podporuje přenosy souborů po částech, tak jak mu je předkládá systém. Pro tuto funkci ovladač vyžaduje verzi serveru z května roku 2012. Nově ovladač podporuje cache za účelem snížení množství dat přenášených po síti a zrychlení přístupu k často používaným souborům. Podpora offline operací nově umožňuje práci se soubory i pokud počítač není připojen k síti.

Zadání se podařilo splnit v plném rozsahu, ovšem přestože je práce hotová, tak stále existují další možnosti ke zlepšení. Je možné přidělat grafický frontend k ovladači pro komfortnější ovládání pro uživatele. Přidání podpory sdílené cache mezi více uživateli na jednom systému pro úsporu místa, v tomto případě je třeba dát zvýšený důraz na bezpečnost při přístupu k souborům.

Literatura

[1] SOLTER, Nicholas, Gerald JELINEK a David MINER. *Windows nt file systém internals*: *definitive guide*. 1st ed. S.l.: Osr Press, 2006, p. cm. ISBN 978-097-6717-515.

[2] Silicon Graphics. XFS for Linux Administration [online]. 2012. Dostupné z <http://techpubs.sgi.com/library/tpl>

[3] PATE, Steve D, Gerald JELINEK a David MINER. *Unix filesystems*: *evolution, design, and implementation (VERITAS series)*. 1st ed. Indianapolis: J. Wiley, c2003, xxv, 443 s. ISBN 04-711-6483-6.

[4] Sun Microsystems, Inc. RFC 1094, RFC 1813 a RFC 3010 [online]. 2012. Dostupné z <http://www.ietf.org/rfc>

[5] CAMPBELL, Richard. *Managing AFS*: *the Andrew File System*. 2nd ed. Upper Saddle River, NJ: Prentice Hall PTR, c1998, xvi, 479 p. ISBN 01-380-2729-3.

[6] Peter J. Braam. *The Coda Distributed File System* [online]. 2012. Dostupné z <http://www.coda.cs.cmu.edu/ljpaper/lj.html>

[7] Erik Mouw. *Linux Kernel Procfs Guide* [online]. 2012. Dostupné z <http://buffer.antifork.org/linux/procfs-guide.pdf>

[8] GARMAN, Jason. *Kerberos*: *definitive guide*. Vyd. 1. New York: O´Reilly, 2003, 253 s. ISBN 05-960-0403-6.

[9] Marek Pivnička, *KIVFS – Bezpečnost.* Plzeň, 2009. 74s. Bakalářská práce na Fakultě aplikovaných věd Západočeské univerzity. Vedoucí práce Ing. Luboš Matějka

[10] VIEGA, John, Matt MESSIER a Pravir CHANDRA. *Network security with OpenSSL*: *definitive guide*. 1st ed. Sebastopol, CA: O'Reilly, c2002, xiv, 367 p. ISBN 05-960-0270-X.

[11] Free Software Foundation. *GNU Manifest* [online]. 2012. Dostupné z URL: <http://www.gnu.org/gnu/manifesto.html>

[12] CAMPBELL, Richard. *Information technology*: *portable operating systém interface*. 2nd ed. New York: Institute of Electrical and Electronics Engineers, c1996, xxxi, 743 S. ISBN 15-593-7573-6.

[13] Network Working Group. RFC959 [online]. 2012. Dostupné z URL: <http://tools.ietf.org/html/rfc959>

[14] *Domovská stránka projektu FUSE* [online]. 2012. Dostupné z URL: <http://fuse.sourceforge.net/>

[15] Free Software Foundation. *Licence GPL* [online] 2012. Dostupné z URL: <http://www.gnu.org/licenses/gpl.html>

[16] Free Software Foundation. *Licence LGPL* [online] 2012. Dostupné z URL: <http://www.gnu.org/licenses/lgpl.html>

[17] WATANABE, Scott, Marco CESATI a David MINER. *Solaris 10 ZFS essentials*: *evolution, design, and implementation (VERITAS series)*. 3rd ed. Upper Saddle River, NJ: Sun Microsystems Press, c2010, xv, 124 p. Solaris system administration series. ISBN 01-370-0010-3.

[18] *Domovská stránka projektu AVFS* [online]. 2012. Dostupné z URL: <http://sourceforge.net/projects/avf>

[19] *Domovská stránka projektu LoggedFS* [online] 2012. Dostupné z URL: <http://loggedfs.sourceforge.net/>

[20] TANENBAUM, Andrew S. *Distributed systems*: *principles and paradigms*. 2nd ed. Upper Saddle River: Pearson Education, 2007, xviii, 686 s. ISBN 01-323-9227-5.

[21] *Domovská stránka projektu FuseISO* [online] 2012. Dostupné z URL: <http://sourceforge.net/projects/fuseiso/>

[22] *Domovská stránka projektu LUFS* [online] 2012. Dostupné z URL: <http://lufs.sourceforge.net/ >

[23] BOVET, Daniel Pierre, Marco CESATI a David MINER. *Understanding the Linux Kernel : evolution, design, and implementation (VERITAS series)*. 3rd ed. Sebastopol: O´Reilly, 2005, xvi, 923 s. ISBN 978-0-596-00565-8.

[19] STERN, Hal, Mike EISLER a Ricardo LABIAGA. *Managing NFS and NIS*: *portable operating system interface*. 2nd ed. Sebastopol, CA: O'Reilly, c2001, xviii, 490 p. ISBN 15-659-2510-6.

[20] TANENBAUM, Andrew S. *Distributed systems*: *principles and paradigms*. 2nd ed. Upper Saddle River: Pearson Education, 2007, xviii, 686 s. ISBN 01-323-9227-5.

[26] Radek Strejc, *KIVFS - Datové úložiště.* Plzeň, 2012. 86s. Diplomová práce na Fakultě aplikovaných věd Západočeské univerzity. Vedoucí diplomové práce Ing. Luboš Matějka

Přehled zkratek

DFS	Distributed File System
MIT	Massachusetts Institute of Technology
KDC	Key Distribution Center
SSL	Secure Socket Layer
RAM	Random-Access Memory
CPU	Central Processing Unit
MAC	Message authentication code
SSL	Secure Socket Layer
TLS	Transport Layer Security
KIV	Katedra informatiky a výpočetní techniky
API	Application Programming Interface
ZČU	Západočeská univerzita v Plzni
FAV	Fakulta aplikovaných věd
FUSE	Filesystem in Userspace
GPL	General Public License
LGPL	Lesser General Public License