Souborový systém na bázi i-uzlů

KIV/ZOS – Semestrální práce

Student: Ladislav Čákora
Osobní číslo: A23B0149P
Email: cakoral@students.zcu.cz
Datum: 5. února 2026

Obsah

1 Zadání

Tématem semestrální práce bude práce se zjednodušeným souborovým systémem založeným na i-uzlech. Vaším cílem bude splnit několik vybraných úloh. Základní funkčnost, kterou musí program splňovat. Formát výpisů je závazný, můžete si však přidat další pomocná hlášení, jakou jsou podrobnosti k chybám či doplňující vypisované informace. Program bude mít jeden parametr a tím bude název vašeho souborového systému. Po spuštění bude program čekat na zadání jednotlivých příkazů s minimální funkčností viz níže (všechny soubory mohou být zadány jak absolutní, tak relativní cestou):

1. Zkopíruje soubor s1 do umístění s2: `cp s1 s2`
2. Přesune soubor s1 do umístění s2, nebo přejmenuje s1 na s2: `mv s1 s2`
3. Smaže soubor s1: `rm s1`
4. Vytvoří adresář a1: `mkdir a1`
5. Smaže prázdný adresář a1: `rmdir a1`
6. Vypíše obsah adresáře a1: `ls a1`
7. Vypíše obsah textového souboru s1 na obrazovku: `cat s1`
8. Změní aktuální cestu do adresáře a1: `cd a1`
9. Vypíše aktuální cestu: `pwd`
10. Vypíše informace o souboru/adresáři s1/a1 (v jakých clusterech se nachází): `info a1`
11. Nahraje soubor s1 z pevného disku do umístění s2 ve vašem FS: `incp s1 s2`
12. Nahraje soubor s1 z vašeho FS do umístění s2 na pevném disku: `outcp s1 s2`
13. Načte soubor z pevného disku, ve kterém budou jednotlivé příkazy, a začne je sekvenčně vykonávat. Formát je 1 příkaz/1řádek: `load s1`
14. Příkaz provede formát souboru, který byl zadán jako parametr při spuštení programu na souborový systém dané velikosti. Pokud už soubor nějaká data obsahoval, budou přemazána. Pokud soubor neexistoval, bude vytvořen: `format 600MB`
15. Příkaz `exit` – ukončení práce s vaším programem.
16. Příkaz `statfs` – vypíše statistiky souborového systému, jako je velikost, počet obsazených a volných bloků, počet obsazených a volných i-uzlů, počet adresářů.
17. Hardlink: `ln s1 s2`

2 Návrh

2.1 Struktura souborového systému

superblok	bitmapa uzlů	bitmapa bloků	tabulka i-uzlů	datové bloky

Tabulka 1: Struktura souborového systému

1. Superblok - metadata souborového systému (velikost, odsazení částí, počet uzlů a bloků, ...), dohromady 40 bytů
2. Bitmapa i-uzlů - informace o obsazenosti i-uzlů prostřednictvím bitové mapy, velikost $(\textit{počet uzlů} + 7) \div 8$ (počet uzlů dělený 8, zaokrouhlený nahoru)
3. Bitmapa datových bloků - informace o obsazenosti datových bloků prostřednictvím bitové mapy, velikost $(\textit{počet bloků} + 7) \div 8$ (počet bloků dělený 8, zaokrouhlený nahoru)
4. Tabulka i-uzlů - metadata souborů a adresářů a odkazy na jejich datové bloky
5. Datové bloky - data souborů a adresáře v souborovém systému, uspořádaná do bloků po 1024 bytech

Počty uzlů a bloků se určují podle tohoto algoritmu:

1. $bloky = velikostFS \div velikostBloku$
2. $uzly = bloky \div 4$
3. pokud se superblok, odpovídající bitmapy, uzly a bloky vejdou do velikosti souborového systému, je hotovo, jinak $bloky- = 1$ a vrať se na krok 1

Jelikož poměr uzlů a bloků není nikde uložen, program zvládne pracovat i se sytémy naformátované pod jiným poměrem, pokud je zachována struktura superbloku, bitmap a uzlů. Není třeba dodržet ani velikost bloků, ta by měla být ve správně naformátovaném superbloku a program se podle toho správně nastaví.

2.2 Struktura superbloku

- magická konstanta - 4 byty, podle kterých se program rozhoduje, jestli načetl naformátovaný systém
- velikost bloků
- celkový počet bloků
- celkový počet uzlů
- velikost souborového systému

- počátek bitmapy uzlů
- počátek bitmapy bloků
- počátek tabulky uzlů
- počátek datových bloků
- id uzlu kořenového adresáře - mělo by být obvykle 0

Všechny položky jsou uloženy na 4 bytech.

2.3 Struktura i-uzlu

- id uzlu - 4 byty
- velikost uzlu - 4 byty
- počet linků - 4 byty
- 5 přímých odkazů na datové bloky - 5×4 byty
- nepřímý odkaz 1. stupně - 4 byty
- nepřímý odkaz 2. stupně - 4 byty
- typ uzlu (soubor, adresář) - 1 byte

2.4 Struktura programu

Aplikace je sestavena z celkem šesti hlavních tříd a sedmi pomocných knihoven. Každá třída má na starosti jednu vrstvu interakce mezi uživatelem a datovým souborem nebo jednu z datových struktur. Pomocné knihovny obsahují výjimky vyvolávané souborovým systémem, funkce pro čtení a zápis dat do datového souboru a podobně.

3 Implementace

3.1 Třída `Shell`

Třída `Shell` představuje interaktivní příkazový řádek (shell) pro práci se souborovým systémem. Slouží jako uživatelské rozhraní, které zprostředkovává komunikaci mezi uživatelem a vrstvou `FilesystemInterface`.

3.1.1 Účel třídy

Hlavním úkolem třídy `Shell` je:

- zobrazovat příkazový prompt,
- číst vstup uživatele ze standardního vstupu,
- předávat zadané příkazy objektu `FilesystemInterface`,

- zobrazovat výsledky příkazů nebo chybová hlášení.

Třída `Shell` neobsahuje žádnou logiku souborového systému. Neprovádí parsování cest, nepracuje s inody ani přímo nemanipuluje s daty. Veškerá funkčnost souborového systému je delegována na `FilesystemInterface`. Toto rozdělení zajišťuje jasnou separaci uživatelského rozhraní a aplikační logiky.

3.1.2 Konstruktor

```
explicit Shell(FilesystemInterface& fs);
```

Konstruktor inicializuje shell s existující instancí `FilesystemInterface`. Předaná reference musí zůstat platná po celou dobu životnosti objektu `Shell`.

3.1.3 Metoda `Run`

```
void Run();
```

Metoda `Run` spouští hlavní interaktivní smyčku shellu. Její chování je následující:

- zobrazí příkazový prompt (`> `),
- načte jeden řádek ze standardního vstupu,
- ignoruje prázdné nebo bílé znaky,
- předá příkaz objektu `FilesystemInterface`,
- vypíše výstup nebo chybovou zprávu.

Smyčka je ukončena v případě, že uživatel zadá příkaz `exit` nebo dojde ke konci vstupu (EOF).

3.1.4 Interní stav

Třída obsahuje pouze referenci na objekt `FilesystemInterface`, který zajišťuje parsování a vykonávání příkazů. Díky tomu je implementace shellu jednoduchá, přehledná a snadno rozšiřitelná.

3.2 Třída `FilesystemInterface`

Třída `FilesystemInterface` představuje příkazovou (CLI) vrstvu nad implementací souborového systému. Funguje jako adaptér mezi textovými příkazy zadanými uživatelem (např. `ls`, `cp a b`) a veřejným API třídy `Filesystem`.

Tato třída zajišťuje parsování příkazů, kontrolu parametrů a směrování volání na odpovídající metody souborového systému. Nezabývá se přímou manipulací s bloky, inody ani výpisem výstupu – tyto úlohy jsou řešeny v jiných vrstvách systému.

3.2.1 Účel a odpovědnosti

Hlavní odpovědnosti třídy `FilesystemInterface`:

- zpracování textových příkazů zadaných uživatelem,
- parsování názvu příkazu a jeho parametrů,
- validace vstupních argumentů,
- volání odpovídajících operací souborového systému,
- vracení strukturovaných výsledků vyšší vrstvě (shellu).

Třída záměrně neřeší žádnou uživatelskou interakci ani výpis na standardní výstup. Veškerá komunikace s uživatelem probíhá prostřednictvím třídy `Shell`.

3.2.2 Konstruktor a destruktor

```
explicit FilesystemInterface(std::string path);
```

Konstruktor inicializuje instanci rozhraní a vytvoří vnitřní instanci souborového systému nad zadaným obrazem disku. Současně registruje všechny podporované příkazy shellu.

```
~FilesystemInterface();
```

Destruktor zajišťuje korektní uložení metadat a uvolnění alokovaných prostředků souborového systému.

3.2.3 Spuštění příkazu

```
std::pair<std::string, std::string>
Execute(const std::string& command);
```

Metoda `Execute` představuje hlavní vstupní bod rozhraní. Zpracuje surový text příkazu zadaný uživatelem, provede jeho parsování a zavolá odpovídající obslužnou metodu.

Návratová hodnota je dvojice řetězců:

- první prvek obsahuje výstup příkazu určený k zobrazení,
- druhý prvek obsahuje chybovou nebo stavovou zprávu.

3.2.4 Obsluha příkazů

Jednotlivé shellové příkazy (např. `ls`, `cp`, `rm`, `mkdir`, `cd`, `statfs` aj.) jsou implementovány jako soukromé metody ve tvaru `cmd_xxx`. Tyto metody:

- přijímají již naparsované argumenty,
- provádějí kontrolu jejich správnosti,
- volají odpovídající metody třídy `Filesystem`,
- vracejí textový výsledek operace.

Mapování textových názvů příkazů na jejich implementace je realizováno pomocí asociativní struktury a lambda funkcí, což umožňuje snadné rozšíření rozhraní o další příkazy bez zásahu do architektury systému.

3.3 Třída `Filesystem`

Třída `Filesystem` implementuje souborový systém uložený v jediném datovém souboru. Podporuje adresáře, běžné soubory a pevné odkazy (hard links).

Souborový systém využívá klasickou strukturu složenou ze superbloku, tabulky inodů a bitmap pro správu alokace inodů a datových bloků. Veškerá metadata jsou při zániku objektu korektně zapsána zpět na disk.

3.3.1 Základní vlastnosti

Souborový systém poskytuje:

- hierarchickou adresářovou strukturu,
- práci se soubory a jejich obsahem,
- podporu pevných odkazů,
- aktuální pracovní adresář,
- dotazování na stav a statistiky systému.

Třída `Filesystem` neřeší uživatelské rozhraní ani parsování textových příkazů. Je volána výhradně prostřednictvím vyšší vrstvy `FilesystemInterface`.

3.3.2 Konstruktor a destruktor

```
explicit Filesystem(const std::string& imagePath);
```

Konstruktor otevře obraz souborového systému. Pokud obraz obsahuje platný superblok, je souborový systém připojen (mount). V opačném případě zůstává v neformátovaném stavu.

```
~Filesystem();
```

Destruktor zajistí uložení všech metadat a korektní uzavření datového souboru.

3.3.3 Formátování a stav

```
void Format(uint32_t bytes);
```

Inicializuje nový souborový systém o zadané velikosti a přepíše veškerá existující data v datovém souboru.

```
bool Formated() const;
```

Vrací informaci, zda je souborový systém aktuálně naformátován.

3.3.4 Operace nad adresáři

Souborový systém umožňuje vytváření a odstraňování adresářů a práci s aktuálním pracovním adresářem:

- vytvoření adresáře,
- odstranění prázdného adresáře,
- změnu aktuálního adresáře,
- výpis obsahu adresáře.

3.3.5 Operace nad soubory

Podporovány jsou základní operace se soubory:

- zápis a čtení obsahu souboru,
- kopírování a přesun souborů,
- odstranění souboru,
- vytváření pevných odkazů.

Zápis souboru vždy přepisuje jeho původní obsah a alokuje nové datové bloky.

3.3.6 Dotazy a informace

Třída poskytuje metody pro získání informací o:

- aktuální cestě v souborovém systému,
- metadatech konkrétního uzlu,
- celkovém stavu a statistikách souborového systému.

Výsledky těchto dotazů jsou vraceny v lidsky čitelné podobě a jsou určeny k prezentaci vyšší vrstvou aplikace.

3.3.7 Vnitřní implementace

Interně třída využívá:

- superblok pro globální metadata,
- bitmapy pro správu alokace inodů a bloků,
- pomocné metody pro práci s inody, bloky a cestami.

Tyto metody nejsou přístupné zvenčí a slouží výhradně k implementaci veřejného API třídy.

3.4 Superblok souborového systému

3.4.1 Magické číslo souborového systému

Souborový systém používá konstantní magické číslo, které slouží k ověření platnosti a typu datového souboru při jeho otevření.

```
constexpr uint32_t FILESYSTEM_MAGIC = 0xDEADBEEF;
```

Tato hodnota je uložena v superbloku a je kontrolována při inicializaci souborového systému. Pokud se hodnota neshoduje, je obraz považován za neplatný nebo nenaformátovaný.

3.4.2 Třída `Superblock`

Třída `Superblock` reprezentuje globální metadata souborového systému. Superblok je uložen na začátku datového souboru (na bytovém offsetu 0) a je nezbytný pro správnou interpretaci veškerých ostatních dat.

Obsahuje informace o geometrii souborového systému, jeho velikosti a rozložení jednotlivých struktur na disku.

3.4.3 Ukládané informace

Superblok uchovává zejména:

- identifikaci souborového systému pomocí magického čísla,
- velikost datového bloku,
- celkový počet bloků a inodů,
- celkovou velikost obrazového souboru,
- byteové offsety bitmap a tabulky inodů,
- identifikátor kořenového adresáře.

Díky těmto údajům je možné jednoznačně lokalizovat všechny klíčové struktury souborového systému bez závislosti na externí konfiguraci.

3.4.4 Serializace

Superblok má pevně danou serializovanou velikost

```
40~\text{B}
```

a jeho binární rozložení je nezávislé na zarovnání dat struktur v paměti.

Pro zápis a čtení z disku jsou použity specializované metody:

- serializace struktury do pole bytů,
- rekonstrukce struktury z uložených dat.

Tento přístup zajišťuje přenositelnost a konzistentní interpretaci superbloku napříč běhy programu.

3.4.5 Význam superbloku

Superblok je klíčovou strukturou celého souborového systému. Bez jeho správného načtení není možné přistupovat k inodům, datovým blokům ani adresářové hierarchii. Z tohoto důvodu je jeho validace první operací prováděnou při otevření datového souboru.

3.5 Bitmapa alokace

3.5.1 Třída `Bitmap`

Třída `Bitmap` implementuje obecnou bitmapovou strukturu používanou pro sledování alokace fixního množství prostředků. V kontextu souborového systému slouží především ke správě:

- inodů,
- datových bloků.

Každý bit v bitmapě odpovídá jednomu prostředku, přičemž hodnota:

- `0` značí volný prostředek,
- `1` značí alokovaný prostředek.

3.5.2 Reprezentace dat

Bitmapa je uložena v kompaktní podobě jako pole bajtů, kde jsou jednotlivé bity ukládány po sobě (LSB-first). Tento formát umožňuje efektivní ukládání bitmapy na disk i její snadnou rekonstrukci v paměti.

Velikost bitmapy je dána počtem sledovaných prostředků a není závislá na zarovnání datových struktur v paměti.

3.5.3 Funkcionalita

Bitmapa poskytuje následující základní operace:

- čtení a zápis jednotlivých bitů,
- vyhledání prvního volného prostředku,
- zjištění počtu volných prostředků.

Tyto operace jsou využívány při alokaci a uvolňování inodů a datových bloků v rámci implementace souborového systému.

3.5.4 Serializace

Třída `Bitmap` podporuje serializaci a deserializaci do surové bajtové reprezentace. Tato vlastnost umožňuje přímé uložení bitmapy do datového souboru a její opětovné načtení při připojení souborového systému.

Serializovaná podoba bitmapy obsahuje pouze vlastní bitová data, zatímco počet reprezentovaných bitů je předáván externě. Tím je zajištěna jednoduchá a flexibilní integrace s rozložením dat na disku.

3.6 Inody

3.6.1 Třída `INode`

Třída `INode` reprezentuje základní popisný prvek souborového systému. Každý inode odpovídá jednomu objektu v souborovém systému, kterým může být běžný soubor nebo adresář.

Inode uchovává identitu objektu, jeho velikost, počet pevných odkazů a informace o umístění datových bloků na disku. Data jsou adresována pomocí kombinace přímých a nepřímých odkazů.

3.6.2 Ukládané informace

Každý inode obsahuje následující metadata:

- unikátní identifikátor inodu,
- příznak typu objektu (soubor / adresář),
- velikost souboru v bajtech,
- počet pevných odkazů,
- pole přímých odkazů na datové bloky,
- první a druhou úroveň nepřímého adresování.

Tento návrh umožňuje efektivní práci s malými soubory a zároveň škálovatelné adresování větších datových objektů.

3.6.3 Adresování dat

Inode obsahuje pevný počet přímých odkazů na datové bloky. Po jejich vyčerpání jsou využity nepřímé odkazy:

- první úroveň nepřímého adresování,
- druhá úroveň nepřímého adresování.

Nepřímé bloky neobsahují data samotná, ale tabulky identifikátorů dalších datových bloků.

3.6.4 Pevné odkazy

Počet pevných odkazů určuje, kolik položek adresáře na daný inode odkazuje. Inode je fyzicky odstraněn až ve chvíli, kdy počet odkazů klesne na nulu a nejsou k němu připojeny žádné datové bloky.

3.6.5 Serializace

Inode má pevně definovaný binární formát o velikosti

```
41~\text{B}
```

který je nezávislý na zarovnání datových struktur v paměti.

Pro ukládání na disk a načítání do paměti slouží specializované metody pro serializaci a deserializaci. Díky pevnému rozložení je možné inody přímo adresovat v tabulce inodů.

3.6.6 Význam inodů

Inody tvoří základ adresářové hierarchie i práce se soubory. Oddělení metadat (inode) od datových bloků umožňuje efektivní sdílení dat pomocí pevných odkazů a zjednodušuje správu souborového systému jako celku.

3.7 Práce se soubory na nízké úrovni

3.7.1 Třída `FileIOHandler`

Třída `FileIOHandler` poskytuje nízkoúrovňové binární vstupně-výstupní operace nad souborem pomocí jediného spravovaného datového proudu. Slouží jako základní stavební kámen pro práci s datovým souborem používaným souborovým systémem.

Je navržena pro náhodný přístup k datům a podporuje čtení, zápis, vytváření i změnu velikosti souboru.

3.7.2 Otevírání a správa souboru

Soubor je otevírán v jednom ze dvou režimů:

- **READ** – pouze pro čtení,
- **READ_WRITE** – čtení i zápis, přičemž soubor je vytvořen, pokud neexistuje.

Třída zajišťuje korektní otevření souboru, kontrolu přístupových práv a bezpečné uzavření datového proudu při zániku objektu.

3.7.3 Čtení a zápis dat

Rozhraní umožňuje čtení a zápis libovolného počtu bajtů na zadaném offsetu od začátku souboru. Tyto operace jsou využívány vyššími vrstvami systému pro přímou manipulaci s bloky, bitmapami a inody.

Zápisové operace jsou povoleny pouze v zapisovatelném režimu, jinak je vyvolána odpovídající výjimka.

3.7.4 Změna velikosti souboru

Třída podporuje změnu velikosti otevřeného souboru. Při zvětšení je nově přidaný prostor inicializován nulami, což zajišťuje deterministické chování souborového systému.

3.7.5 Význam v architektuře

`FileIOHandler` abstrahuje veškeré operace se souborem na disku a odděluje je od logiky souborového systému. Díky tomu může být implementace vyšších vrstev nezávislá na konkrétní implementaci práce se soubory a snadno testovatelná.

4 Uživatelská příručka

4.1 Sestavení a spuštění

Program je možně přeložit pomocí nástroje cmake a přiloženého souboru CMakeLists.txt v hlavním adresáři projektu. V hlavním adresáři projektu je zároveň skript `build.sh`, který vše provede a výsledný spustitelný soubor `ZOS` bude v adresáři `build/`. Program je samozřejmě možné přeložit i pomocí příkazů nástroje cmake, nebo jeho GUI. Program je poté možné spustit pomocí příkazu `./build/ZOS <cesta k datovému souboru>`, pokud datový soubor zatím neexistuje, na dané cestě se vytvoří jako prázdný soubor.

5 Závěr

Výsledkem práce je program představující jednoduchý souborový systém na bázi i-uzlů. Veškeré příkazy ze zadání byly implementovány. Program je rozdělený do logických částí, které umožnily jednodušší implementaci a testování. Program jsem bohužel z časových důvodů nezvládl řádně otestovat na jiných prostředích než na Linuxu, ale v rámci implementace nevidím důvod, kvůli kterému by mohla být funkčnost na jiných prostředích omezena.