Dokumentace semestrální práce

KIV/ZOS – Základy operačních systémů

Václav Honzík, A19B0674P 29. 12. 2020

# Zadání

Tématem semestrální práce bude práce se zjednodušeným souborovým systémem založeným na i-uzlech. Vaším cílem bude splnit několik vybraných úloh.

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 courseware).

Maximální délka názvu souboru bude 8+3=11 znaků (jméno.přípona) + \0 (ukončovací znak v C/C++), tedy 12 bytů. Každý název bude zabírat právě 12 bytů (do délky 12 bytů doplníte \0 - při kratších názvech).

Nad vytvořeným a naplněným souborovým systémem umožněte provedení operace **Hardlink** (ln s1 s2).

* Příkaz vytvoří hard link na soubor s1 s názvem s2. Dále se s ním pracuje očekávaným způsobem, tedy např. cat s2 vypíše stejný obsah jako cat s1.

# Analýza problému

Souborový systém (file system – FS) je systém pro správu dat (souborů), které potřebujeme ukládat persistentně - tzn. data jsou dostupná i po skončení daného programu. Taková data mohou být různého typu v závislosti na aplikaci – text, binární soubory, konfigurační soubory, obrázky apod. To, jaký FS bude v dané části systému použit je závislé především na datech a operacích, které pro ně budou potřeba.

Naivní implementací jednoduchého FS je tzv. **kontinuální alokace** – tzn. soubory jsou ukládány za sebou. Výhodou tohoto přístupu je jeho snadná implementace a také rychlé čtení / zápis souboru – protože úseky souboru nejsou rozdělené v celém paměťovém prostoru. Typicky je daný paměťový prostor rozdělen do bloků, kdy každý blok má fixní velikost (v řádu kB – MB) a každý záznam se souborem obsahuje pouze jeho velikost a počet bloků.

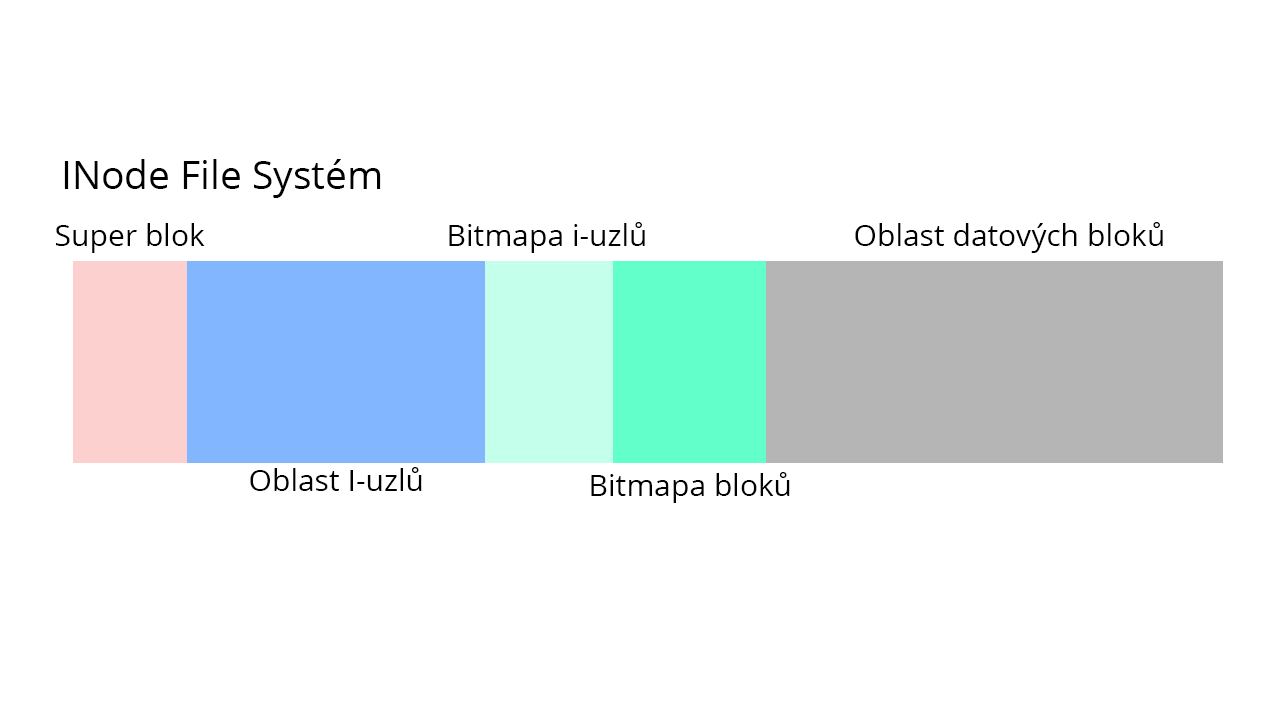
Kontinuální alokace je samozřejmě pro většinu problémů značně limitována – soubory nelze snadno rozšířit nebo naopak zkrácením dojde k mrtvým úsekům, které většina souborů nevyužije, protože je jejich velikost větší než velikost úseku. Kontinuální alokace je tedy vhodná pouze pro *read-only* paměť, kdy víme, že nebudeme na disk (nebo jiné médium) často zapisovat nebo data jakkoliv měnit.

FS implementovaný v této práci je založený na použití tzv. **i-uzlů** (**I-Node** – informační uzel). I-Node obsahuje metadata pro soubor – datum vytvoření, právo pro přístup, zda-li je soubor adresář, a také samotné odkazy na data. Podobně jako při kontinuální alokaci jsou soubory uložené v blocích (datový blok), rozdílem je ale to, že i-uzel může odkazovat na bloky v celém paměťovém prostoru (resp. oblasti datových bloků). To samozřejmě přináší velkou flexibilitu při mazání nebo rozšiřování souborů.

Zpravidla I-Node obsahuje několik typů odkazů na blok. Přímý odkaz obsahuje adresu daného bloku, kde jsou data uložená, těchto odkazů je většinou v řádu jednotek. 1. nepřímý odkaz odkazuje na datový blok, kde jsou uložené přímé odkazy. Odkazy vyšších řádů (2., 3., …) odkazují vždy na bloky s nepřímými odkazy o jeden stupeň nižší. Nepřímých odkazů je zpravidla málo (méně než přímých odkazů), protože dokáží adresovat velké množství dat.

Dále je potřeba v FS implementovat nějakým způsobem složky, aby nebyly předměty pouze v kořenovém adresáři. I-Node má kromě odkazů na datové bloky i flag pro to, zda-li je složka nebo ne. I-uzel, který je složkou funguje podobně jako soubor, s tou změnou, že jeho bloky neobsahují data ale tzv. položky adresáře. Tyto položky (DirectoryItem / FolderItem apod.) obsahují název souboru a odkaz na daný i-uzel.

Souborový systém je rozdělený do několika částí – **boot blok** a **super blok**, prostor i-uzlů, prostor datových bloků. Boot blok obsahuje kód pro zavedení FS do operačního systému a v této práci nebyl implementován. Super blok je jednou z nejdůležitějších struktur, protože obsahuje všechny důležité informace pro navigaci v FS. Zde jsou zapsána data jako adresa počátku datových bloků, adresa i-uzlů, velikost bloku atd. Pokud by se tato struktura poškodila, celý file systém bude nepoužitelný.

Přestože pro funkční implementaci dané struktury stačí, různé operace lze vylepšit pomocí zavedení **bitmap**. Bitmapa v této implementaci slouží pro označení zda-li je daný objekt volný nebo použitý. Výhoda bitmapy je zde taková, že nemusíme dané objekty složitě hledat a kontrolovat, zda-li jsou použitelné, ale místo toho se podíváme do bitmapy a podle té rozhodneme. V reálné implementaci by tato operace měla být daleko rychlejší, protože jediné, co provedeme je bitová operace – např. AND. Celková struktura FS je vidět na Obr. č. 1.

Obrázek 1 - Struktura souborového systému

# Implementace

Souborový systém byl implementovaný v jazyce C++ 20 za použití překladače GCC MinGW 64. Pro kompilaci celého programu byl použit nástroj CMake. Implementace je napsaná tak, aby splňovala zadání semestrální práce.

## Implementace paměti

Samotnou persistentní „paměť“ pro ukládání dat reprezentuje soubor vytvořený OS na dané platformě. Tento soubor má fixní velikost a jsou v něm uložená data pomocí FS. Soubor kromě dat také obsahuje pole s i-uzly, super blok a bitmapy pro datové bloky a i-uzly.

Do souboru se zapisuje pomocí objektu **FileStream**, který je wrapper pro **fstream**, což je C++ třída pro čtení a zápis do souboru. Výhoda fstreamu je taková, že ho lze použít bez složitějších úprav pro zápis a čtení souboru zároveň – tzn. nemusíme používat dva streamy, které by se ještě nějak museli zavírat a otevírat. FileStream také umí zapisovat a číst složitější objekty jako jsou vektory, i-uzly, super blok a další.

Jednotlivé prvky FS jsou také reprezentované pomocí C++ tříd (popř. structů) a jsou uložené ve složce filesystem/io/model. Třída **Bitmap** reprezentuje bitmapu a umí pomocí třídy FileStream číst a zapisovat data do disku, data z bitmapy jsou zároveň uložená v paměti, aby nebyl systém nemusel pokaždé bitmapu číst z disku, což by značně zpomalilo běh. Bitmapu lze použít pro jakýkoliv objekt u kterého je konstantní velikost a je kontinuálně alokovaný.

Třída **INode** reprezentuje daný i-uzel na disku. Tato třída obsahuje informace o daném souboru nebo složce, některé informace zůstaly nevyužity jako zjištění časového razítka apod. **SuperBlock** reprezentuje super blok disku – zde jsou zapsány informace o adresách jednotlivých částech FS. Poslední třídou, která se zapisuje na disk je třída **FolderItem**. Tato třída reprezentuje předmět v dané složce (adresáři) a pouze obsahuje informace o názvu a adrese daného i-uzlu.

## Struktura programu

Vstupní bod programu je v souboru **main.cpp**, který obsahuje **main()** funkci. Zde dojde k zpracování argumentů a spuštění samotného programu. Veškerou uživatelskou interakci zpracovávají funkce z třídy **CLI** (ve složce filesystem/cli/CLI.\*) - zkratka pro command line interface. Tato třída interpretuje požadavky uživatele a podle toho předává požadavky samotnému souborovému systému.

Souborový systém je reprezentován několika třídami, které jsou umístěné ve složce filesystem/components. Hlavní třídou je **FileSystem**, se kterou komunikuje CLI. Tato třída přijímá příkazy s parametry a poté volá **FileSystemController**, který příkazy provádí. FileSystemController má na starost správný chod celého souborového systému. Zde by šlo většinu kódu napsat do třídy FileSystemController, nicméně jsem kód rozložil do několika dalších tříd, jednak kvůli odlaďování chyb a také kvůli celkové přehlednosti.

FileSystemController využívá několik dalších tříd z filesystem/components. Pro jednotný přístup k získávání a uvolňování paměti je potřeba použít třídu **MemoryAllocator**. Tento objekt obsahuje bitmapy pro manipulaci s bloky a i-uzly a také má užitečné funkce pro formátování i-uzlů nebo získání bloku naformátovaného pro specifický typ dat (např. blok s ukazately pro první nepřímý odkaz apod.). Pro získání pozic pro čtení jednotlivých dat (a jejich úpravu) využívá tato třída SuperBlock.

**INodeIO** je třída, která obsahuje většinu logiky pro zapisování a čtení dat, které souvisí s INode. Zde jsou funkce pro vytváření, odstranění, přidání složek nebo souborů, přidávání souborů do složek apod.

Poslední dvě třídy ze složky components jsou **FileOperations** a **PathContext**. Tyto třídy se využívají pro samotné vysokoúrovňové operace se soubory. PathContext slouží pro uložení kontextu v aplikaci – zde je uložená aktuální cesta uživatele. PathContext také automaticky načítá aktuální složku pro rychlejší přesuny nebo zobrazení předmětů. FileOperations v sobě PathContext ukládá a je to jediná třída, která má ke kontextu v aplikaci přístup (ostatní třídy jej nepotřebují).

Dále se v programu ještě využívají složky filesystem/global a filesystem/utils. Složka global obsahuje pouze jednu třídu – **Globals**. Tato třída obsahuje globální data jako velikost bloku, počet pointerů na blok apod., aby třídy, které tato data potřebují je nemuseli zbytečně vyžadovat v konstruktoru. Utils naopak obsahuje tříd několik, které jsou využívané pro různé funkce, které se jinam „nehodily“. Důležitou třídou je třída FileSystemPath, která je něco jako wrapper pro std::vector a umožňuje snadnou změnu cesty a zjednodušení operací pro třídy FileOperations a PathContext.

Při chybě se často vyhazuje tzv. FSException, což je objekt rozšiřující třídu std::exception a umožňuje snadné ošetření (nebo vyhození) chyb. Oproti např. vrácení chybových kódu lze tak snadno předat i hlášení o tom, co chybu způsobilo až do CLI, které dané hlášení vypíše.

# Uživatelská dokumentace

Pro spuštění programu je potřeba přeložení zdrojového kódu. Pro překlad je potřeba nejnovější verze MinGW GCC (popř. Linux GCC) překladače a nástroj CMake (3.17 a vyšší), který samotný překlad zařídí.

Překlad lze provést následujícím způsobem:

1. V příkazové konzoli se přesuneme do složky s umístěním zdrojového kódu (složka musí obsahovat CMakeLists.txt)
2. V závislosti na platformě zadáme příkaz pro build:

Windows:

cmake . -G “MinGW MakeFiles“

mingw32-make

Linux:

cmake . -G “MinGW MakeFiles“

mingw32-make