SOUBOROVÝ SYSTÉM POMOCÍ I-UZLŮ
-
Dokumentace

Vypracoval: Petr Tomšík

Základy operačních systémů [KIV/ZOS]
Semestrální práce
ZS 2020/2021

Zadání

Tématem semestrální práce je zjednodušený souborový systém založený na i-uzlech. Souborový systém (disková oblast) bude simulována souborem na disku, např. s názvem myFS. Při prvním spuštění soubor myFS zatím neexistuje. Zadáním příkazu format 600MB vytvoří soubor myFS a připraví ho k použití (u běžného příkazu pro formátování se velikost neudává, v naší práci ano, abychom věděli, jak velký fs 1 vytvořit). Při dalším spuštění již soubor myFS bude obsahovat námi vytvořené soubory a adresáře.

Vaším cílem bude splnit několik vybraných úloh. Formát výpisů je závazný. Budeme předpokládat korektní zadání syntaxe příkazů, nikoliv však sémantiky (tj. např. cp s1 zadáno nebude, ale může být zadáno cat s1, kde s1 neexistuje). 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).

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

Možný výsledek:

OK

FILE NOT FOUND (není zdroj)

PATH NOT FOUND (neexistuje cílová cesta)

2) Přesune soubor s1 do umístění s2, nebo přejmenuje s1 na s2

mv s1 s2

Možný výsledek:

OK

FILE NOT FOUND (není zdroj)

PATH NOT FOUND (neexistuje cílová cesta)

3) Smaže soubor s1

rm s1

Možný výsledek:

OK

FILE NOT FOUND

4) Vytvoří adresář a1

mkdir a1

Možný výsledek:

OK

PATH NOT FOUND (neexistuje zadaná cesta)

EXIST (nelze založit, již existuje)

5) Smaže prázdný adresář a1

rmdir a1

Možný výsledek:

OK

FILE NOT FOUND (neexistující adresář)

NOT EMPTY (adresář obsahuje podadresáře, nebo soubory)

6) Vypíše obsah adresáře a1

ls a1

Možný výsledek:

-FILE

+DIRECTORY PATH NOT FOUND (neexistující adresář)

7) Vypíše obsah souboru s1

cat s1

Možný výsledek:

OBSAH

FILE NOT FOUND (není zdroj)

8) Změní aktuální cestu do adresáře a1

cd a1

Možný výsledek:

OK

PATH NOT FOUND (neexistující cesta)

9) Vypíše aktuální cestu

pwd

Možný výsledek:

PATH

10) Vypíše informace o souboru/adresáři s1/a1 (v jakých clusterech se nachází)

info a1/s1

Možný výsledek:

NAME – SIZE – i-node NUMBER – přímé a nepřímé odkazy

FILE NOT FOUND (není zdroj)

11) Nahraje soubor s1 z pevného disku do umístění s2 v pseudoNTFS

incp s1 s2

Možný výsledek:

OK

FILE NOT FOUND (není zdroj)

PATH NOT FOUND (neexistuje cílová cesta)

12) Nahraje soubor s1 z pseudoNTFS do umístění s2 na pevném disku

outcp s1 s2

Možný výsledek:

OK

FILE NOT FOUND (není zdroj)

PATH NOT FOUND (neexistuje cílová cesta)

13) Načte soubor z pevného disku, ve kterém budou jednotlivé příkazy, a začne je sekvenčně vykonávat. Formát je 1 příkaz/1řádek

load s1

Možný výsledek:

OK

FILE NOT FOUND (není zdroj)

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

Možný výsledek:

OK

CANNOT CREATE FILE

15) Kontrola konzistence (check) – pokud login studenta začíná j-r Zkontrolujte, zda jsou soubory nepoškozené (např. velikost souboru odpovídá počtu alokovaných datových bloků) a zda je každý soubor v nějakém adresáři. Součástí řešení bude nasimulovat chybový stav, který následná kontrola odhalí.

Struktura souborového systému s i-uzly

U systému s i-uzly je disková oblast rozdělena na následující části.

• Bootblock - obsahuje kód pro případné bootování systému z dané diskové oblasti. (v našem pseudosystému nebude)

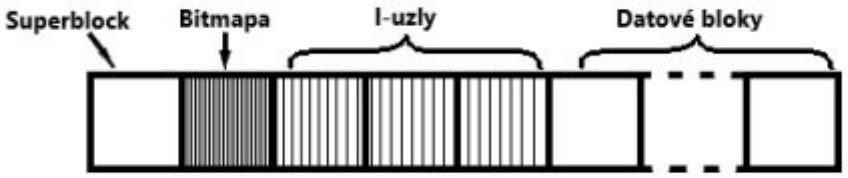
• Superblock - obsahuje základní údaje o fs – velikost clusteru, kde jsou umístěné i-uzly, kde začínají datové bloky, kde leží bitmapa datových bloků, atd.

• Bitmapa - označuje volné a použité datové bloky.

• Oblast i-uzlů - část datových bloků (typicky 10%) je vymezena pro uložení i-uzlů. I-uzel reprezentuje jeden soubor (adresář je také soubor). Data tvořící obsah souboru jsou popsána jedním i-uzlem. Každý i-uzel obsahuje číslo i-uzlu, přímé odkazy na datové bloky, nepřímé odkazy na datové bloky, které ale neobsahují přímo data souboru, ale pouze odkazy na další datové bloky, a další potřebné údaje pro popis daného souboru.

• Datové bloky- obsahují data jednotlivých souborů, obsahy adresářů atp.

Adresář v systému s i-uzly obsahuje název souboru a číslo i-uzlu.



Obrázek 1: Schématické zobrazení našeho souborového systému

Implementace

Souborový systém je implementovaný v programovacím jazyce C. Velikost jednoho datového bloku (clusteru) volíme 4096B a 5% bloků ze všech vymezíme pro i-uzly. Celkový počet clusterů se odvíjí od zadané velikosti našeho fs při formátování. Avšak skutečná velikost fs bude ve většině případů o něco málo menší než zadaná hodnota, neboť velikost musí odpovídat násobku velikosti clusteru.

Jeden cluster zabere superblock (nevyužitá část bloku je prázdná), další clustery slouží pro bitmapu, jejichž počet musí být takový, aby bitmapa pokryla všechny datové bloky. Jeden cluster pokryje 4096 "bitů". Následujících 5% clusterů ze všech je pro uchování i-uzlů, kde každý i-uzel zabírá 38B, viz Datové struktury - inode. Zbytek clusterů tvoří datové bloky.

Datové struktury

superblock – uchovává následující informace o našem fs (každá položka zabírá 4B). Superblock je vytvořen/modifikován při každém formátování nebo je načten ze souboru při spuštění programu.

- disk_size – (4B) přesná velikost souborového systému v bytech
- cluster_size – (4B) velikost jednoho clusteru, v našem případě 4096B
- cluster_count – (4B) počet všech clusterů
- inode_count – (4B) počet i-uzlů
- bitmap_cluster_count – (4B) počet clusterů pro bitmapu
- inode_cluster_count – (4B) počet clusterů pro i-uzly
- data_cluster_count – (4B) počet datových bloků
- bitmap_start_address – (4B) adresa začátku bitmapy
- inode_start_address – (4B) adresa začátku i-uzlů
- data_start_address – (4B) adresa začátku datových bloků

inode – reprezentuje i-uzel, který zabírá v souboru 38B. Jeden i-uzel dokáže popsat soubor o velikosti maximálně 4235264B. Kořen má ID 0. Při vytváření každého i-uzlu se nastaví minimálně první přímý odkaz na datový blok, i přesto že bude prázdný.

- nodeid – (4B) ID i-uzlu, pokud je FREE = volný i-uzel
- isDirectory – (1B) určuje typ souboru, 0 = soubor, 1 = adresář
- references – (1B) počet referencí na i-uzel (v našem případě vždy 1)
- file_size – (4B) velikost souboru, samotný adresář má nulovou velikost
- direct1 – (4B) 1. přímý odkaz 4
- direct2 – (4B) 2. přímý odkaz
- direct3 – (4B) 3. přímý odkaz
- direct4 – (4B) 4. přímý odkaz
- direct5 – (4B) 5. přímý odkaz
- indirect1 – (4B) 1. nepřímý odkaz
- indirect2 – (4B) 2. nepřímý odkaz

directory_item – reprezentuje položku adresáře.

- **inode** – (4B) ID i-uzlu
- **item_name** – (12B) název souboru/adresáře dlouhý maximálně 11 znaků (delší název je oříznut)

• **next** – odkaz na další položku v adresáři (kvůli spojovému seznamu)

directory – reprezentuje adresář.

• **parent** – odkaz na rodičovský adresář
• **current** – položka adresáře, která reprezentuje tento adresář
• **subdir** – odkaz na první podadresář (spojový seznam podadresářů)
• **file** – odkaz na první soubor (spojový seznam souborů)

Při spuštění programu se v hlavní funkci main ověří, zda byl zadán vstupní parametr (název souborového systému) a otestuje se existence zadaného souboru, při čemž se nastaví příznaková proměnná fs_formatted na 0 = fs není naformátován, nebo 1 = fs již existuje a v takovém případě se zavolá funkce load, která vytvoří potřebné struktury a inicializuje je na hodnoty uložené v souboru simulující náš fyzický disk. Následně se zavolá procedura runProgram, ve které se ve smyčce načítají jednotlivé příkazy zadané uživatelem, popřípadě příkazy čtené ze souboru, a spouští se odpovídající procedury pro jednotlivé příkazy. Pro přerušení smyčky a ukončení programu je přidán příkaz "***e***". Před samotným ukončením programu se volá procedura freeMemory, která řádně uvolní veškerou alokovanou paměť a zavře soubor reprezentující fs, který je jinak po celý běh programu otevřený.

Globální proměnné

• **fs_name** – název souborového systému

• **fs** – soubor simulující náš souborový systém

• **superblock** – struktura pro náš superblok

• **bitmap** – pole pro naši bitmapu

• **inodes** – pole i-uzlů, ID i-uzlu odpovídá indexu do pole

• **directories** – pole referencí na všechny adresáře, adresář pro daný i-uzel je pod indexem odpovídající ID i-uzlu

• **working_directory** – pracovní adresář, ve kterém se zrovna nacházím (při spuštění je nastaven na kořenový adresář "\")

• **fs_formatted** – určuje, zda byl fs již naformátován (v případě, že fs dosud nebyl naformátován, tak při zavolání jakéhokoli příkazu, kromě format a e, dojde k výpisu, že fs musí být nejprve naformátován)

• **block_buffer** – pole o velikost jednoho clusteru, které slouží pro přesuny dat z/do souboru

• **file_input** – příznak, který určuje, zda mají být příkazy načítány ze souboru (0 = ne, 1 = ano)

Provedení jednotlivých příkazů

format

void format(long bytes)
Před zavoláním procedury format se ještě zavolá funkce get_size, která zpracuje zadanou velikost souborového systému. Přípustné jednotky jsou KB, MB, GB, které musí být velkými písmeny a musí bezprostředně následovat za zadanou velikostí. Pokud nejsou jednotky explicitně zadány, bere se velikost v bytech. Vzhledem k tomu, že velikost ukládáme v 32bitovém integeru se znaménkem, je maximální přípustná velikost 2 147 483 647 B a naopak nejmenší možná velikost je nastavena na 20 480 B, která zajistí, že bude existovat alespoň jeden cluster s i-uzly. Při formátování se nejdříve vytvoří instance superbloku, pokud dosud neexistuje. Postupně se naplní její hodnoty spočtené ze zadané velikosti fs. Následně se alokují pole pro bitmapu, i-uzly a adresáře. Vytvoří se kořenový adresář, který se nastaví jako pracovní adresář. Dále se vynuluje celá bitmapa a všechny i-uzly se inicializují na FREE, značící volný i-uzel. Nultý uzel je nastaven na kořenový adresář (ID = 0). Pro vytvoření souboru o požadované velikosti se celý soubor nejprve naplní nulama a poté se do něj zapíšou jednotlivé inicializované bloky (superblok, bitmapa, i-uzly).

cp

*void cp(char *files)*
1. Otestování, zda je souborový systém naformátovaný.
2. Rozdělení vstupního parametru na zdrojový soubor a cílový adresář.
3. Rozparsování zdrojového souboru – funkce parse_path – získání názvu souboru a adresáře, ve kterém se daný soubor nachází.
4. Nalezení souboru v daném adresáři – funkce find_item – získání položky adresáře, která reprezentuje kopírovaný soubor.
5. Nalezení cílového adresáře – funkce find_directory – získání cílového adresáře.
6. Otestování, zda cílový adresář neobsahuje položku se stejným názvem jako kopírovaný soubor – funkce test_existence.
7. Získání pole datových bloků, které zabírá kopírovaný soubor – funkce get_data_blocks.
8. Získání volných datových bloků pro zkopírování – funkce find_free_data_blocks.
9. Získání ID volného i-uzlu – funkce find_free_inode.
10. Vytvoření nové položky create_directory_item pro kopii souboru a přidání do spojového seznamu souborů v cílovém adresáři.
11. Inicializace i-uzlu pro novou kopii souboru – funkce initialize_inode.
12. Zapsání bitmapy, i-uzlu, nové položky v cílovém adresáři do souboru.
13. Aktualize velikostí všech nadřazených adresářů a zapsání do souboru.
14. Překopírování datových bloků zdrojového souboru do volných datových bloků – kopírujeme ve smyčce po jednotlivých blocích. Poslední datový blok je překopírován mimo cyklus, jelikož pro něj zjišťujeme, jak velkou část datového bloku je třeba zkopírovat.

mv

*void mv(char *files)*
1. Otestování, zda je souborový systém naformátovaný.
2. Rozdělení vstupního parametru na zdrojový soubor a cílový adresář.
3. Rozparsování zdrojového souboru – funkce parse_path – získání názvu souboru a adresáře, ve kterém se daný soubor nachází.
4. Nalezení cílového adresáře – funkce find_directory – získání cílového adresáře.
5. Otestování, zda zdrojový a cílový adresář je stejný -> potom nic nepřesouváme.
6. Otestování, zda cílový adresář neobsahuje položku se stejným názvem jako přesouvaný soubor – funkce test_existence.
7. Odstranění přesouvaného souboru ze spojového seznamu souborů zdrojového adresáře a zapsání do souboru.
8. Přidání přesouvaného souboru do spojového seznamu souborů cílového adresáře a zapsání do souboru.

rm

*void rm(char *file)*
1. Otestování, zda je souborový systém naformátovaný.
2. Rozparsování souboru – funkce parse_path – získání názvu souboru a adresáře, ve kterém se daný soubor nachází.
3. Nalezení souboru v daném adresáři a následné jeho odstranění ze spojového seznamu souborů.
4. Získání pole datových bloků, které zabírá odstraňovaný soubor – funkce get_data_blocks.
5. Vynulování block_bufferu a tím přemažeme všechny datové bloky v souboru, které obsahoval odstraňovaný soubor.
6. Aktualizace bitmapy, adresáře odstraňovaného souboru, vymazání i-uzlu a aktualizace velikostí nadřazených adresářů.

mkdir

*void mymkdir(char *path)*
1. Otestování, zda je souborový systém naformátovaný.
2. Rozparsování adresáře – funkce parse_path – získání názvu adresáře a rodičovského adresáře, ve kterém je nový adresář vytvářen.
3. Otestování, zda rodičovský adresář neobsahuje položku se stejným názvem jako nový adresář – funkce test_existence.
4. Vytvoření nového adresáře – funkce create_directory.

rmdir

*void myrmdir(char *path)*
1. Otestování, zda je souborový systém naformátovaný.
2. Rozparsování adresáře – funkce parse_path – získání názvu adresáře a rodičovského adresáře, ve kterém se nachází odstraňovaný adresář.
3. Nalezení odstraňovaného adresáře ve spojovém seznamu adresářů.
    - Při nalezení ověřit, zda odstraňovaný adresář neobsahuje nějaké podadresáře nebo soubory.
    - Při odstraňování pracovního adresáře se pracovním adresářem stane rodič.
4. Aktualizace bitmapy, vymazání adresáře v souboru, vynulování i-uzlu.

ls

*void ls(char *path)*
1. Otestování, zda je souborový systém naformátovaný.
2. Nalezení adresáře – funkce find_directory.
3. Projít spojový seznam podadresářů a vypsat název každé položky. Stejně tak pro spojový seznam souborů.

cat

*void cat(char *file)*
1. Otestování, zda je souborový systém naformátovaný.
2. Rozparsování souboru – funkce parse_path – získání názvu souboru a adresáře, ve kterém se daný soubor nachází.
3. Nalezení souboru v daném adresáři – funkce find_item – získání položky (souboru) adresáře, jejíž obsah má být vypsán.
4. Vypsání obsahu souboru – funkce print_file.

cd

*void cd(char *path)*
1. Otestování, zda je souborový systém naformátovaný.
2. Nalezení adresáře – funkce find_directory.
3. Nalezený adresář nastavit jako pracovní adresář.

pwd

void pwd()
1. Otestování, zda je souborový systém naformátovaný.
2. Od pracovního adresáře procházíme hierarchii adresářů až ke kořeni. Jednotlivé názvy adresářů na této cestě si ukládáme do pomocného pole.
3. Vypíšeme všechny adresáře z pole v opačném pořadí, než v jakém byly do pole vloženy a před každý adresář vložíme lomítko.

info

*void info(char *path)*
1. Otestování, zda je souborový systém naformátovaný.
2. Rozparsování cesty k požadované položce – funkce parse_path – získání názvu položky a adresáře, ve kterém se nachází.
3. Otestujeme, zda se nejedná přímo o kořenový adresář. Pokud ne, pak se pokusíme nalézt položku mezi soubory daného adresáře a následně mezi podadresáři.
4. Pro nalezenou položku vypíšeme informace – funkce print_info.

incp

*void incp(char *files)*
Princip je velmi podobný jako u příkazu cp s tím rozdílem, že nyní kopírujeme data z externího souboru. Musíme si navíc dát pozor, zda velikost souboru nepřesahuje maximální velikost.

outcp

void outcp(char *files)
Obdoba příkazu incp akorát nyní načítáme datové bloky z načeho fs a kopírujeme je do externího souboru.

load

*FILE *load(char *file)*
1. Otestování, zda je souborový systém naformátovaný.
2. Otevření souboru, ze kterého mají být načítány jednotlivé příkazy a tento soubor je návratovou hodnotou funkce.
3. Nastavení příznaku pro načítání příkazů ze souboru file_input na 1.

check

void check()
Prochází pole i-uzlů inodes a pokud najde i-uzel označený jako soubor, tak nejdříve se vypočte předpokládaný počet bloků souboru a následně se to porovná s počtem bloků, který vždy stanový funkce get data blocks. V případě, kdy jsou hodnoty rozdílné, se ukončí kontrola a systém oznámí chybu. Dále se kontroluje, zda soubor je v nějakém adresáři pomocí parametru i-uzlu references. Data systému jsou defaultně konzistentní, proto byla do programu přidána funkce break data, která odstraní reference na i-uzly.

dis

void dis()
Nastaví nekonzistentní stav. Na inodes->references přiřadit 0, tak aby vznikl chybový stav a poté se zavolá *void check().*

Další pomocné funkce

int32_t find_free_inode()
Prochází pole i-uzlů a vrací ID prvního volného (ID = FREE) i-uzlu.

int32_t *find_free_data_blocks(int count)
Funkce přebírá parametr, kolik volných datových bloků má být nalezeno. Nejprve se pokusí najít daný počet volných datových bloků, které se nacházejí u sebe a pokud se to nepovede, pak bere postupně každý volný blok. Funkce vrací pole volných datových bloků nebo NULL, v případě, že daný počet volných bloků nebyl nalezen.

int32_t *get_data_blocks(int32_t nodeid, int *block_count, int *rest)
Funkce vrací pole datových bloků, které obsahuje i-uzel s ID nodeid (pouze bloky se skutečnými daty souboru či adresáře, ne bloky nepřímých odkazů, které obsahují pouze odkazy na další datové bloky). Rozlišuje se mezi hledáním datových bloků pro adresář a soubor, jelikož datové bloky pro soubor jsou uspořádány v i-uzlu postupně (nejdřív první přímý odkaz, pak druhý, atd.), kdežto u adresáře mohou být některé přímé či nepřímé odkazy vynechány (v případě, že z adresáře byly odstraněny nějaká data), proto musíme projít všechny možné odkazy na datové bloky. Celkový počet datových bloků (bez nepřímých odkazů) je uložen na adresu block_count.

directory_item *find_item(directory_item *first_item, char *name)
Iteruje přes spojový seznam souborů nebo adresářů first_item a hledá položku s názvem name, která je vzápětí vrácena.

int parse_path(char *path, char **name, directory **dir)
Zpracuje cestu k souboru/adresáři, tak že oddělí cestu od názvu, který je uložen na adresu name a cestu předá funkci find_directory. Nalezený adresář je uložen na adresu dir.

int create_directory(directory *parent, char *name)
Funkce najde volný i-uzel, volný datový blok, vytvoří adresář a nastaví odpovídající hodnoty adresáři a i-uzlu. Každý nový adresář obsahuje jeden datový blok i přesto, že neobsahuje žádná data. Položka nového adresáře je ještě přidána do spojového seznamu adresářů rodičovského adresáře. Nakonec se aktualizuje bitmapa, i-uzel a adresář v souboru našeho fs.

directory_item *create_directory_item(int32_t inode_id, char *name)
Vytvoří položku adresáře podle předaných parametrů.

int test_existence(directory *dir, char *name)
Hledá položku s názvem name v adresáři dir. Prochází spojové seznamy souborů a adresářů v daném adresáři. Pokud najde položku s daným názvem, vrátí 1 jinak 0.

directory *find_directory(char *path)
Pokud cesta path začíná lomítkem, nastaví se výchozí adresář na kořenový adresář, jinak na pracovní adresář. Postupně rozsekáváme cestu pomocí funkce strtok přes oddělovač lomítka. Pokud daná část obsahuje jednu tečku, nikam se nepřesouváme a pokračujeme další částí. Pokud obsahuje dvě tečky, přesuneme se z aktuálního adresáře dir do jeho rodiče. V ostatních případech prohledáváme podadresáře aktuálního adresáře a pokud nalezneme adresář s odpovídajícím názvem, přesuneme se do něj. Pokud žádný takový adresář nenajdeme ukončujeme prohledávání a vracíme NULL.

void free_directories(directory *root)
Uvolní alokovanou paměť pro adresář root a rekurzivně pro všechny jeho podadresáře.

void clear_inode(int id)
Nastaví hodnoty daného i-uzlu na FREE(-1) nebo na 0.

void update_sizes(directory *dir, int32_t size)
Procházíme hyerarchii adresářů od dir až ke kořeni a k velikosti každého i-uzlu představující daný adresář přičteme velikost size. Změněný i-uzel zapíšeme do souboru. Pokud se jedná o aktualizaci velikosti po odstranění souboru, pak parametr size je záporný.

void print_info(directory_item *item)
Vypíše informace o dané položce item.

void print_file(directory_item *item)
Vypíše do konzole obsah souboru item. Nejdříve funkcí get_data_blocks získáme datové bloky daného souboru a následně načítáme jednotlivé bloky z našeho souboru do block_buffer a ten vypíšeme do konzole.

void initialize_inode(int32_t id, int32_t size, int block_count, int tmp_count, int *last_block_index, int32_t *blocks)
Inicializace i-uzlu s id na hodnoty předané parametry.

void load_fs()
Vytvoří superblock, bitmapu a pole i-uzlů a naplní je daty z již existujícího souboru, který reprezentuje náš souborový systém. Dále vytvoří pole adresářů a kořený adresář. Ostatní adresáře jsou načteny funkcí load_directory.

void load_directory(directory *dir, int id)
Naplní adresář dir všemi jeho položkami. Nejdřív získáme všechny jeho datové bloky funkcí get_data_blocks, přes které iterujeme a načítáme jednotlivé položky (ID i-uzlu a název) ze souboru, které umisťujeme do příslušného spojového seznamu daného adresáře. Po načtení všech položek projdeme všechny položky reprezentujicí podadresář a pro každý vytvoříme strukturu adresáře, nad kterou rekurzivně zavoláme tuto funkci.

void update_bitmap(directory_item *item, int8_t value, int32_t *data_blocks, int b_count)
Nastavuje bitmapu pro položku item. Parametr value obsahuje hodnotu, na kterou se nastavují jednotlivé bloky bitmapy, tj. 1 = obsazený datový blok, 0 = volný datový blok. Nejdříve musíme získat datové bloky, které zabírá položka item. Ty buď jsou přímo předané parametrem data_blocks nebo se zavolá funkce get_data_blocks. Následně se projdou všechny bloky a podle nich se nastaví bitmapa a také se zapíše do souboru.

void update_inode(int id)
Zapíše hodnoty i-uzlu s id do souboru.

int update_directory(directory *dir, directory_item *item, int action)
Stará se o přidání nebo odstranění položky item do/ze souboru. Parametr action určuje o jakou operaci se jedná, tzn. 1 = přidání položky, 0 = odebrání položky. Nejdříve získáme všechny datové bloky adresáře dir. V případě přidání položky do souboru, hledáme volné místo v nalezených datových blocích, tj. čteme ze souboru ID i-uzlu a pokud je 0, pak na toto místo vložíme přidávanou položku. Pokud žádné prázdné místo nenalezneme v příslušných datových blocích, pak se pokusíme najít zcela nový datový blok, který přidáme do volného odkazu v daném adresáři, přičemž musíme aktualizovat bitmapu a i-uzel.
V případě odstraňování položky opět čteme ze souboru položky v jednotlivých datových blocích a porovnáváme ID odstraňované položky s přečtenou hodnotou. Pokud se shodují, pak nalezenou položku přemažeme nulama. Při odstraňování ještě musíme ošetřit situaci, kdy odstraňovaná položka je jediná v daném datovém bloku. V takovém případě je třeba označit daný datový blok za volný. O tento proces se stará funkce remove_reference.

void remove_reference(directory_item *item, int32_t block_id)
Odstraňuje referenci na prázdný datový blok z i-uzlu a v případě, že se jedná o datový blok nepřímého odkazu, pak i ze souboru. V tomto případě navíc musíme ještě ošetřit situaci, kdy prázdný datový blok byl jediný v bloku nepřímého odkazu. Pak se musí ještě odstranit reference na nepřímý odkaz.

Závěr
Všechny požadované funkce souborového systému byly otestovány a všechny poskytly požadovaný výsledek. Počet i-uzlů je větší než celkový počet datových bloků, což v našem systému není zrovna příznivé, jelikož každý i-uzel zabírá vždy alespoň jeden datový blok, proto nikdy není možné využít všechny i-uzly, které máme k dispozici. Řešením by bylo zmenšit počet bloků vyhrazené pro i-uzly a dále by se prázdným adresářům nemusel přiřazovat datový blok, dokud je adresář prázdný.

Zdroje
Popis struktury souborového systému byl převzat z prezentace předmětu Základy operačních systémů - vyučující Ladislav Pešička.