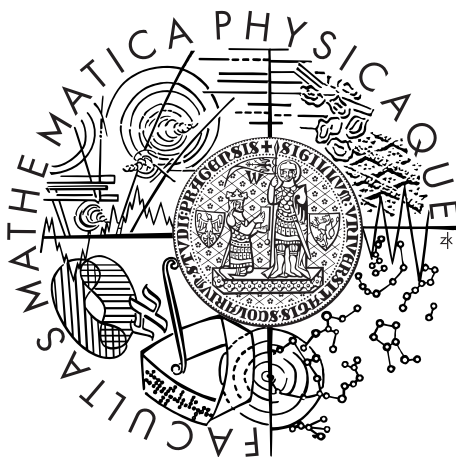


Univerzita Karlova v Praze  
Matematicko-fyzikální fakulta

## DIPLOMOVÁ PRÁCE



František Princ

## Ovladač souborového systému ext4 pro HelenOS

Katedra distribuovaných a spolehlivých systémů

Vedoucí diplomové práce: Mgr. Martin Děcký

Studijní program: Informatika  
Studijní obor: Softwarové systémy

Praha 2012

Na tomto místě bych rád poděkoval vedoucímu diplomové práce Mgr. Martinu Děckému za pomoc při výběru tématu, za odborné vedení a také za cenné připomínky a rady během psaní práce. Můj dík patří také autorům operačního systému HelenOS, kteří ho stále vyvíjí a udržují a tím poskytují kvalitní základ pro vznik prací, jako je tato. Dále bych rád poděkoval svým rodičům, kteří mi umožnili studium a vždy mě plně podporovali. V neposlední řadě děkuji své přítelkyni za podporu během celého studia a samozřejmě i za to, že si práci přečetla a poskytla mi mnoho připomínek k textu z jazykového hlediska.

Prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně a výhradně s použitím citovaných pramenů, literatury a dalších odborných zdrojů.

Beru na vědomí, že se na moji práci vztahují práva a povinnosti vyplývající ze zákona č. 121/2000 Sb., autorského zákona v platném znění, zejména skutečnost, že Univerzita Karlova v Praze má právo na uzavření licenční smlouvy o užití této práce jako školního díla podle § 60 odst. 1 autorského zákona.

V Praze dne 29. července 2012

František Princ

**Název práce:** Ovladač souborového systému ext4 pro HelenOS

**Autor:** František Princ

**Katedra (ústav):** Katedra distribuovaných a spolehlivých systémů

**Vedoucí bakalářské práce:** Mgr. Martin Děcký

**Abstrakt:** HelenOS je experimentální operační systém založený na mikrojádře a vyvíjený na půdě Matematicko-fyzikální fakulty Univerzity Karlovy v Praze. Jeho současná implementace je připravena na podporu více souborových systémů díky abstrakci pomocí VFS, nicméně spektrum podporovaných souborových systémů je zatím velmi malé. Ext4 je moderní souborový systém, který pochází od vývojářů jádra Linux. Jeho předcházející verze ext2 a především ext3 jsou stále velmi oblíbené a výchozí v drtivé většině distribucí Linuxu. Implementaci souborových systémů ext2/3/4 můžeme najít i v dalších unixových systémech, například na systémech \*BSD nebo v OpenSolarisu. Cílem této práce je rozšířit operační systém HelenOS tak, aby dokázal se zmíněnými souborovými systémy pracovat.

**Klíčová slova:** HelenOS, ext4, souborový systém, ovladač

**Title:** HelenOS ext4 filesystem driver

**Author:** František Princ

**Department:** Department of Distributed and Dependable Systems

**Supervisor:** Mgr. Martin Děcký

**Abstract:** HelenOS is an experimental operating system based on microkernel. It's developed by people related to Faculty of Mathematics and Physics at Charles University in Prague. The present implementation is prepared to support a scale of filesystems because of VFS abstraction, but actually only a few of filesystems are supported. Ext4 is modern filesystem, developed Linux kernel developers. The previous releases ext2 and especially ext3 are still very popular in many Linux distributions. Implementation of ext2/3/4 filesystems can be also found in other unix-based operating systems, for example \*BSD or OpenSolaris. The main goal of this thesis is to extend HelenOS operating system to support previously mentioned filesystems.

**Keywords:** HelenOS, ext4, filesystem, driver

# Obsah

<b>1</b>	<b>Úvod</b>	<b>1</b>
1.1	Motivace . . . . .	1
1.2	Cíl práce . . . . .	1
1.3	Struktura práce . . . . .	1
1.4	Obsah příloženého DVD . . . . .	2
<b>2</b>	<b>Operační systém HelenOS</b>	<b>3</b>
2.1	Architektura . . . . .	3
2.2	Úlohy a vlákna . . . . .	3
2.3	Synchronizace . . . . .	4
2.4	Meziprocesová komunikace (IPC) . . . . .	5
2.5	Podpora souborových systémů . . . . .	6
<b>3</b>	<b>Souborový systém ext4</b>	<b>9</b>
3.1	Historie . . . . .	9
3.2	Souborový systém ext2 . . . . .	10
3.2.1	Blok . . . . .	10
3.2.2	Skupina bloků . . . . .	10
3.2.3	Superblok . . . . .	11
3.2.4	Tabulka deskriptorů skupin bloků . . . . .	16
3.2.5	Bitová mapa datových bloků . . . . .	16
3.2.6	Bitová mapa i-nodů . . . . .	17
3.2.7	Tabulka i-nodů . . . . .	17
3.2.8	I-node . . . . .	17
3.2.9	Soubory . . . . .	20
3.2.10	Adresáře . . . . .	20
3.2.11	Rozšířené atributy . . . . .	22
3.2.12	Komprese . . . . .	24
3.3	Souborový systém ext3 . . . . .	24
3.3.1	Změny datových struktur . . . . .	25
3.3.2	Žurnálování . . . . .	28
3.3.3	Indexování adresářů - HTree . . . . .	29
3.4	Souborový systém ext4 . . . . .	31
3.4.1	Změny datových struktur . . . . .	31

3.4.2	Zvýšení limitů . . . . .	32
3.4.3	Extenty . . . . .	34
3.4.4	Vylepšení i-nodů . . . . .	34
3.5	Zajímavé vlastnosti v linuxovém ovladači ext4 . . . . .	35
3.5.1	Vícebloková alokace . . . . .	35
3.5.2	Odložená alokace . . . . .	36
3.5.3	Zrychlení fsck . . . . .	36
3.5.4	Kontrolní součty při žurnálování . . . . .	36
3.5.5	Možnost vypnutí žurnálování . . . . .	37
3.5.6	Online defragmentace . . . . .	37
3.5.7	Perzistentní prealokace . . . . .	37
3.5.8	Bariéry . . . . .	37
3.6	Algoritmy . . . . .	38
3.6.1	Alokace i-nodu . . . . .	38
3.6.2	Alokace datového bloku . . . . .	39
3.6.3	Vytvoření souboru / adresáře . . . . .	40
3.6.4	Uvolňování datových bloků . . . . .	40
3.6.5	Mazání souboru / adresáře . . . . .	40
<b>4</b>	<b>Návrh</b>	<b>41</b>
4.1	Architektura . . . . .	41
4.2	Volba podporovaných vlastností . . . . .	41
4.3	Využití dalších komponent . . . . .	43
4.4	Přidružené programy . . . . .	44
<b>5</b>	<b>Implementace</b>	<b>45</b>
5.1	Serverová část ovladače . . . . .	45
5.1.1	Datové struktury . . . . .	46
5.1.2	Funkce . . . . .	46
5.2	Knihovní část ovladače . . . . .	46
5.2.1	Datové struktury . . . . .	47
5.2.2	Funkce . . . . .	48
5.3	Další programy . . . . .	53
5.4	Testování . . . . .	53
5.5	Provázanost s vývojem HelenOS . . . . .	55

<b>6 Závěr</b>	<b>56</b>
6.1 Ostatní implementace . . . . .	56
6.2 Zhodnocení . . . . .	56
6.3 Rozšiřitelnost . . . . .	57
<b>Literatura</b>	<b>59</b>
<b>Seznam obrázků</b>	<b>60</b>
<b>A Zdrojové kódy</b>	<b>61</b>
<b>B Kompilace a spuštění</b>	<b>62</b>

# 1. Úvod

## 1.1 Motivace

Práce se soubory a adresáři je základní funkcionalitou každého operačního systému. Souborový systém je termín, kterým označujeme způsob, jakým operační systém organizuje data do souborů a adresářů na fyzickém médiu (např. pevném disku). Jednotlivé souborové systémy se liší právě způsobem organizace dat, ale také výkonností, odolností proti chybám, omezením maximální velikosti souborů i velikosti celého logického disku (partition). Fyzický disk může být rozdělen na více takových logických disků a tedy může obsahovat více souborových systémů.

V současnosti existuje celá řada souborových systémů, přičemž každý operační systém podporuje pouze určitou podmnožinu souborových systémů. To způsobuje obtíže například u přenosných disků, kdy hrozí, že při zapojení k počítači s jiným operačním systémem nebude možno z disku číst, protože ten nepodporuje použitý souborový systém. Ve snaze vyhnout se takovým problémům implementují vývojáři do svých operačních systémů podporu nejrozšířenějších souborových systémů.

Tato práce popisuje implementaci ovladače moderního souborového systému ext4 do operačního systému HelenOS, který vznikl na půdě Matematicko-fyzikální fakulty Univerzity Karlovy v Praze a je stále aktivně vyvíjen.

## 1.2 Cíl práce

Základním cílem této práce je implementace ovladače souborového systému ext4 do operačního systému HelenOS. Výstupem práce je prototypová implementace, která podporuje vhodně zvolenou podmnožinu vlastností souborových systémů ext2, ext3 a ext4. Součástí práce je také rozvaha, které vlastnosti této rodiny souborových systémů je vhodné implementovat pro dosažení vhodné míry interoperability a které lze naopak vynechat z důvodu přílišné časové náročnosti implementace.

## 1.3 Struktura práce

Práce je členěna do několika kapitol, které jsou níže stručně okomentovány. Na konci jsou přílohy týkající se práce se zdrojovými kódy. K práci patří také DVD, které se nachází ve vlepěné obálce na vnitřní straně zadních desek.



- **Kapitola 2** se zabývá popisem operačního systému HelenOS. Nejvíce se věnuje částem systému, které budou využity při implementaci ovladače souborového systému ext4.
- Kapitola 3 popisuje souborový systém ext4 a jeho předcházející verze (ext2, ext3) v chronologickém pořadí. Zabývá se popisem jednotlivých datových struktur a vývojem této rodiny souborových systémů.
- **Kapitola 4** obsahuje návrh architektury ovladače a také diskuzi nad výběrem vlastností, které budou v rámci této práce do ovladače implementovány.
- **Kapitola 5** popisuje konkrétní implementaci ovladače pro systém HelenOS.
- **Kapitola 6** hodnotí úspěšnost implementace a celé práce.
- **Příloha A** se věnuje zdrojovým kódům systému HelenOS.
- **Příloha B** popisuje postup při kompilaci a spuštění systému HelenOS ve virtuálním stroji.

## 1.4 Obsah přiloženého DVD

Součástí práce je i DVD se zdrojovými soubory, textem práce a bootovatelným ISO obrazem, který lze vypálit nebo použít v emulátoru.

- Text práce - soubor `thesis.pdf`
- Kompletní zdrojové kódy - adresář `HelenOS`
- Bootovatelný ISO obraz (zkompilovaný z přiložených zdrojových kódů) - soubor `image.iso`
- Vzorový obraz diskového oddílu - soubor `ext4.img`. Obraz je naformátovaný s vlastnostmi podporovanými ovladačem a je v něm pouze několik souborů a adresář pro jednoduché testování. Příklad připojení obrazu v QEMU je uveden v příloze B.

## 2. Operační systém HelenOS

HelenOS je výzkumný operační systém, vytvořený na Matematicko-fyzikální fakultě Univerzity Karlovy v Praze, který se stále vyvíjí. Vznikl v roce 2004 rozšířením studentského projektu SPARTAN, což je mikrojádru vytvořené Jakubem Jermářem, do softwarového projektu s názvem HelenOS. Dodnes je HelenOS základem mnoha studentských prací, softwarových projektů i dalších aktivit na fakultě. Poskytuje prostor pro implementaci součástí operačního systému v prostředí operačního systému s mikrojádrem, přičemž v současnosti jsou spíše rozšířené systémy s monolitickým jádrem. Vzhledem k tomu, že HelenOS je akademickým projektem, nepůsobí na něj negativní vlastnosti komerčního prostředí a tudíž je možné programovat běžné součásti operačního systému jinak, než je obvyklé, a zkoumat nové možnosti a postupy ve vývoji operačních systémů.

### 2.1 Architektura

HelenOS je postaven na architektuře mikrojádra, které poskytuje pouze nezbytné a nejzákladnější funkce (plánování, správa paměti, meziprocessová komunikace), jejichž kód musí běžet v režimu jádra. Všechny ostatní systémové funkce jsou implementovány v uživatelském prostoru jako tzv. *servery*. Mikrojádru má tedy tu výhodu, že v režimu jádra běží minimální množství kódu, což je přínosné pro stabilitu a spolehlivost celého systému. Naopak zřejmou nevýhodou je oddělení jednotlivých serverů do separátních adresových prostorů, což působí obtíže při komunikaci těchto serverů a běžících úloh<sup>1</sup> obecně. Uživatelské úlohy, které chtějí využít služeb serveru, s ním komunikují výhradně prostředky meziprocessové komunikace.

Podrobnější informace o architektuře jsou dostupné v dokumentu [5], který je sice označen jako zastaralý, ale většina informací je stále platná i v aktuální verzi systému HelenOS 0.4.3.

### 2.2 Úlohy a vlákna

Každá úloha má jedno nebo více vláken (*thread*), které jsou plánovány plánovačem v jádře. V uživatelském prostoru se navíc každé vlákno (*thread*) skládá z několika vláček (*fibril*). Narozdíl od vláken jsou vláčkénka implementována ve standardní

---

<sup>1</sup>Program spuštěný v uživatelském prostoru se nazývá *task*, česky *úloha*.

knihovně a jejich plánování probíhá kooperativně v rámci jednoho vlákna. To má za následek, že vlákénko může běžet nepřerušeně, pokud

- se dobrovolně nevzdá procesoru
- čeká na odpověď IPC
- pomocí IPC operace zablokuje celé vlákno
- je celé vlákno přeplánováno

## 2.3 Synchronizace

Synchronizace je dnes, v době paralelizace, důležitou součástí všech operačních systémů. Každý systém má implementováno několik různých synchronizačních primitiv a nejinak je tomu i v systému HelenOS. I zde se používá klasické rozdělení na aktivní a pasivní primitiva, která poskytuje jádro. Dále jsou k dispozici primitiva pro aplikace v uživatelském prostoru. V následujících odstavcích se stručně seznámíme se synchronizačními primitivy v systému HelenOS. Velmi podrobný popis všech implementovaných synchronizačních primitiv lze získat v dokumentu [5].

Následuje stručný výčet synchronizačních primitiv:

- **Aktivní synchronizační primitiva v jádře**

Stejně jako ve většině operačních systémů, je v jádře implementováno pouze jediné aktivní synchronizační primitivum, kterým je *spinlock*.

- **Pasivní synchronizační primitiva v jádře**

Základním pasivním synchronizačním primitivem je *wait queue* (čekací fronta). Vlákna, která chtějí přistoupit ke kritické sekci, se řadí ve frontě, kde jsou zablokována do doby, než kritickou sekci opustí všechny předchozí vlákna. Pomocí *wait queue* jsou implementována další synchronizační primitiva - *semafor*, *mutex*. Posledním zajímavým prostředkem pro synchronizaci je *condition variable* (podmínková proměnná), která umožňuje vláknu čekat, dokud není splněna podmínka (testování podmínky je chráněno mutexem).

- **Synchronizační primitiva v uživatelském prostoru**

V uživatelském prostoru je implementováno primitivum nazvané *futex*<sup>2</sup>. Standardní knihovna pak obsahuje implementaci primitiv pro synchronizaci vláké-

---

<sup>2</sup>Futex je zkratka z anglického „fast mutex“

nek - mutexy, podmínkové proměnné a read-write zámky. Tyto prostředky se samozřejmě dají použít i pro synchronizaci vláken.

## 2.4 Meziprocesová komunikace (IPC)

IPC je zkratkou Inter-Process Communication, což je i v českém překladu nekonzistentní s termínem úloha (task), kterým se v systému HelenOS označuje běžící program. Nicméně tato zkratka je zavedená a budeme se jí držet.

V prostředí mikrojádra je velmi důležité mít efektivní, rychlou a spolehlivou implementaci IPC. Každá úloha běží ve vlastním adresovém prostoru a nemůže přistupovat do adresového prostoru jiné úlohy. Jediným prostředkem komunikace je právě IPC, které je v systému HelenOS implementováno pomocí asynchronního zasílání zpráv. Autoři systému HelenOS zvolili zajímavý komunikační model, který je analogií k běžnému telefonnímu hovoru, kdy na jedné straně linky je volající a na druhé straně záznamník. Asynchronnost je dána tím, že hovor není okamžitě zodpovězen, ale nejdřív musí být vyzvednut vzkaz ze záznamníku na straně příjemce.

V systému HelenOS má každé vlákénko několik telefonů (*phone*), které jsou propojené se záznamníky (*answerbox*) jiných úloh. Když chce dané vlákénko komunikovat, uskuteční krátký hovor (*call*), jehož „obsah“ bude uložen v záznamníku volané úlohy. Poté může vlákénko buď uskutečnit další hovor, nebo čekat na odpověď. Na straně volané úlohy dojde někdy k vyzvednutí vzkazu ze záznamníku. Poté je vzkaz zpracován a buď je vytvořena a odeslána odpověď, nebo může dojít k přeoslání do záznamníku jiné úlohy. Každopádně iniciátorovi komunikace někdy dorazí do jeho záznamníku odpověď. Podrobnější úvod do problematiky poskytuje dokument [7], ze kterého je volně přeložen i tento odstavec.

Komunikace mezi úlohami je možná pouze přes *phone*. Každá úloha při startu získá jeden otevřený *phone*, který je připojen k serveru *Naming Service*, což je systémová úloha, která registruje ostatní služby systému a klientům poskytuje spojení s registrovanými službami. Při použití analogie s telefonním systémem můžeme *Naming Service* považovat za jakousi telefonní ústřednu. Příklad použití a postup při posílání krátkých zpráv naleznete v dokumentu [7].

Pomocí IPC lze posílat pouze krátké zprávy. Pro obsáhlejší komunikaci se spíše hodí použití sdílené paměti nebo kopírování části paměti z/do cizího adresového prostoru. Oba způsoby jsou si podobné především v tom, že vyžadují IPC komunikaci. Je nutné dát protistraně informaci o tom, že má dojít k nasdílení paměti nebo kopírování dat mezi adresovými prostory. Celá související komunikace se dělí na tři

fáze:

- V zahajovací fázi pošle úloha protistraně zprávu o tom, že má zájem kopírovat data nebo sdílet paměť.
- Ve druhé fázi se příjemce zprávy dozví o požadavku a
- Ve třetí se příjemce rozhodne, zda požadavek akceptuje nebo odmítne.

Podrobný popis včetně příkladů sdílení paměti i kopírování dat mezi adresovými prostory poskytuje opět dokument [7].

## 2.5 Podpora souborových systémů

Stěžejní součástí systému HelenOS je v kontextu této práce podpora souborových systémů.

V systému HelenOS se používá známý koncept VFS (Virtual File System), což je abstrakce nad konkrétními souborovými systémy. Principem VFS je odstínit uživatelské úlohy od práce s rozdílnými souborovými systémy tím, že VFS poskytuje jednotné rozhraní. VFS si interně udržuje vlastní datové struktury, popisující připojené souborové systémy, otevřené soubory apod. Díky VFS je relativně jednoduché přidávat podporu pro nové souborové systémy.

Pro práci se souborovým systémem se používá několik systémových komponent, které tvoří vrstevnatou strukturu. Jednotlivé vrstvy si stručně popíšeme v pořadí od uživatelské úlohy až ke konkrétnímu ovladači souborového systému.

- **Podpora ve standardní knihovně**

Standardní knihovna poskytuje API, které využívají uživatelské úlohy. Jedná se například o běžná volání známá z unixových operačních systémů, jako jsou `open()`, `write()` apod. Ve funkcích standardní knihovny dochází již k určitému předzpracování uživatelských požadavků. Například VFS server akceptuje pouze absolutní cesty, což není mnohdy optimální, ale velmi to zjednodušuje algoritmy používané VFS serverem. Všechny ostatní požadavky směřuje standardní knihovna právě ke zmíněnému VFS serveru.

- **VFS Server**

VFS Server je systémová úloha, která je nejsložitější součástí celé podpory souborových systémů. Spolu se standardní knihovnou tvoří jednotné rozhraní

pro přístup k libovolným souborovým systémům, pro které máme v systému ovladač.

VFS Server poskytuje dvě různá rozhraní.

– **VFS frontend**

Frontendová část VFS serveru je orientovaná na klientské úlohy, které požadují vykonání nějakých operací nad souborovým systémem. Toto rozhraní akceptuje absolutní cesty, deskriptory již otevřených souborů a také tzv. VFS triplety<sup>3</sup>. Pro každý použitý soubor si VFS server udržuje strukturu nazvanou *VFS node*, která poskytuje abstrakci nad skutečnými soubory a je opět jednotná pro všechny souborové systémy. Pro většinu operací je tato struktura nezbytná.

– **VFS backend**

Backendová část VFS serveru slouží ke komunikaci s ovladači souborových systémů. Ovladač, který chce komunikovat s VFS serverem musí implementovat tzv. *VFS output protocol*. Protokolem se v tomto případě myslí soubor zpráv, které VFS server posílá jednotlivým ovladačům. Některé operace je schopen VFS server vykonat sám, bez kontaktování ovladače. Jedná se například o posun ukazatele v otevřeném souboru (operace *seek*), kterou VFS server provádí nad svojí datovou strukturou bez nutnosti kontaktovat ovladač.

Procházení cesty může být operace složitá, protože cesta může vést přes několik různých souborových systémů. V tomto případě musí VFS server kontaktovat postupně jednotlivé ovladače. Kvůli efektivitě se autoři systému rozhodli implementovat PLB (Pathname Lookup Buffer), což je vyrovnávací paměť organizovaná do kruhu, kterou VFS server sdílí s ovladači (sdílená paměť) v režimu pouze pro čtení. VFS server umístí do PLB vyhledávanou cestu (pokud tam již není) a pošle zprávu prvnímu ovladači (kořenový souborový systém). Ten vyřeší svou část cesty a pokud ještě nemůže odpovědět, tak sám kontaktuje další ovladač s informací o nevyřešené části cesty. Takto může na cestě pracovat několik ovladačů, ale až ten poslední v řadě, který dořeší konec cesty, odpoví VFS serveru a zašle mu požadovaný VFS triplet, který odpovídá hledanému souboru. Díky

---

<sup>3</sup>VFS triplet je uspořádaná trojice (fsid, device, index), kde fsid je číslo přiřazené souborovému systému, device je identifikátor zařízení a index je číslo, které může mít v každém souborovém systému jiný význam (u ext2 například číslo i-node).

PLB se ušetří iterativní komunikace VFS serveru s každým ovladačem a navíc díky sdílení PLB nedochází ke zbytečnému kopírování paměti.

- **Ovladač souborového systému**

Každý podporovaný souborový systém musí mít vlastní ovladač, což opět není nic jiného, než uživatelská úloha. Ovladač je poslední a nejnižší úlohou z hlediska našeho pohledu na jednotlivé části podpory souborových systémů. Pak už následuje pouze fyzické blokové zařízení (nejčastěji pevný disk, USB flash disk, paměťová karta apod.). K tomu, aby se ovladač dokázal připojit k fyzickému blokovému zařízení, využívá služeb *DEVMAP serveru*, který ovladačům poskytuje handle fyzického zařízení.

Podrobnější informace naleznete v dokumentu [6], který byl velmi cenným zdrojem pro pochopení fungování práce se souborovými systémy v operačním systému HelenOS.

# 3. Souborový systém ext4

Souborový systém ext4 je nástupcem souborových systémů ext2 a ext3, se kterými je kompatibilní. Obsahuje však nové vlastnosti a navíc výrazně zvyšuje různé limity (např. velikost svazku). Podívejme se proto nejdříve na historický vývoj této rodiny souborových systémů a poté postupně na jednotlivé souborové systémy.

## 3.1 Historie

Roku 1993 vznikl souborový systém ext2 jako náhrada původního souborového systému Extended Filesystem (odtud zkratka ext), který byl součástí jádra Linux a vycházel ze souborového systému používaného v systému Minix. Originální implementace ext2, jejímiž autory jsou Remy Card, Theodore Ts'o a Stephen Tweedie, je obsažena v jádře Linux. Ovladače souborových systémů vycházejících z ext2 byly postupně implementovány v mnoha dalších operačních systémech, např. FreeBSD, GNU Hurd, Microsoft Windows a dalších. Původní návrh systému ext2 neřešil endianitu, tj. pořadí bajtů při ukládání dat. Později se ukázalo, že tento souborový systém a potažmo celý Linux se rozšiřuje i na další architektury, které se často liší endianitou. Proto bylo rozhodnuto, že všechna data se budou ukládat v pořadí little-endian bez ohledu na architekturu. Tento fakt platí pro všechny nástupce systému ext2.

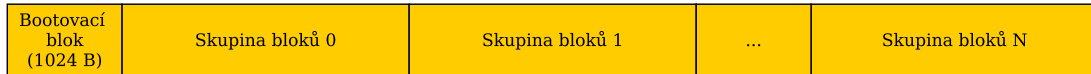
V roce 2001 vznikl nástupce s označením ext3, který oproti ext2 obsahuje navíc takzvané žurnálování. Datové struktury obou zmíněných systémů jsou stejné a tudíž je zaručena zpětná kompatibilita.

S postupem času přibývaly návrhy na vylepšení od vývojářů, přičemž do ext3 byly zahrnuty jen některé. V roce 2006 proto započal vývoj nové generace tohoto souborového systému s označením ext4 a na podzim roku 2008 došlo k vydání stabilní verze. Tato verze již oproti svým předchůdcům obsahuje řadu nových vlastností a proto je zaručena pouze plná zpětná kompatibilita. Dopředná kompatibilita je velmi omezená, pokud se používají extenty a dále pak pokud jsou překročeny velikostní limity pro ext3 (velikost oddílu, souboru, ...).



## 3.2 Souborový systém ext2

Každý oddíl se souborovým systémem ext2 má pevně danou strukturu (obrázek 3.1). Prvních 1024 bajtů z diskového oddílu je rezervováno pro zavaděč operačního systému (tzv. boot sector). Tato část není při práci s oddílem samotným nijak využívána. Zbytek oddílu je vyplněn po sobě jdoucími skupinami bloků, jejichž strukturu si popíšeme v následujících odstavcích.



Obrázek 3.1: Rozložení diskového oddílu ext2

### 3.2.1 Blok

Blok je základní jednotkou pro ukládání dat, který zabírá několik diskových sektorů. Velikost bloku není pevně dána specifikací. Nejmenší velikost bloku je 1KiB. Bežně jsou používány ještě bloky o velikostech 2KiB, 4KiB a 8KiB. Například v jádře Linux 2.6.37 je maximální velikost bloku omezena na 4KiB.

Velikost bloku je určena při vytváření oddílu a poté je neměnná. Správná volba velikosti bloku má vliv na další parametry oddílu jako je maximální velikost souboru nebo maximální velikost celého oddílu. Mezi velikostí bloku a maximálními velikostmi platí samozřejmě přímá úměra. Přesné vztahy mezi velikostí bloku a ostatními parametry oddílu naleznete v dokumentu [1].

### 3.2.2 Skupina bloků

Bloky jsou sdružené do skupin, což je výhodné pro snížení fragmentace a také pro minimalizaci pohybu diskových hlaviček při čtení souvislých dat. Oba zmíněné jevy mají nezanedbatelný vliv na rychlost. První skupina bloků obsahuje metadata týkající se celého oddílu (superblok a tabulka deskriptorů skupin bloků). Tato „globální“ metadata mohou být obsažena i v některých dalších skupinách, kde slouží jako záložní kopie. Každá skupina bloků obsahuje metadata týkající se i-nodů a datových bloků. Kromě metadat obsahuje každá skupina samozřejmě také datové bloky. Všechny skupiny bloků mají stejnou velikost (kromě poslední skupiny, která může být pochopitelně menší). Rozložení dat a metadat přibližuje obrázek 3.2.

Superblok (1 blok)	Tabulka deskriptorů skupin (m bloků)	Bitmapa datových bloků (1 blok)	Bitmapa i-nodů (1 blok)	Tabulka i-nodů (n bloků)	Datové bloky (d bloků)
-----------------------	---	------------------------------------	----------------------------	-----------------------------	---------------------------

Obrázek 3.2: Rozložení dat a metadat ve skupině bloků

### 3.2.3 Superblok

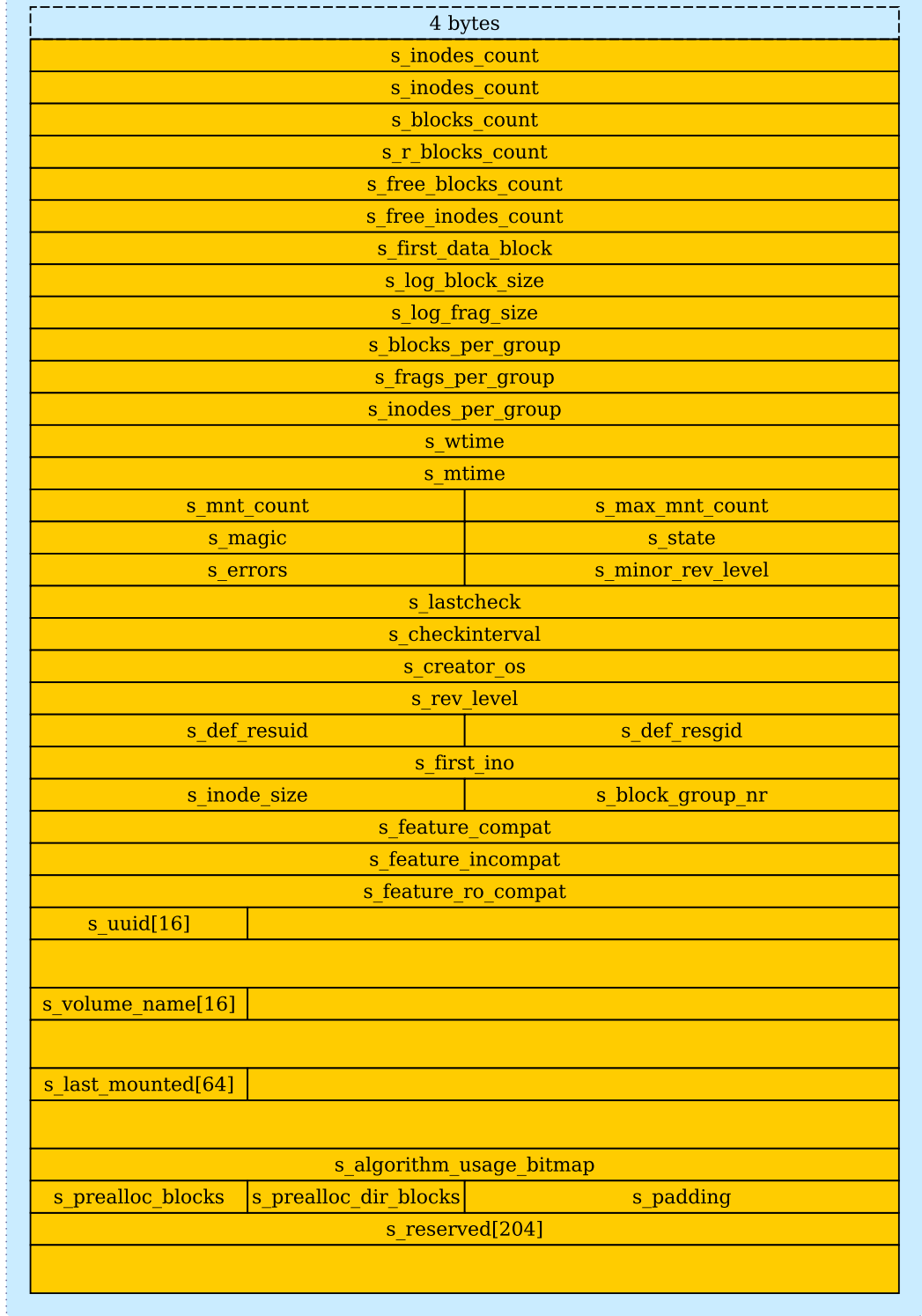
Superblok obsahuje informace o celém diskovém oddílu a je tedy naprosto klíčový pro fungování souborového systému. Jsou zde uloženy údaje o stavu souborového systému, o počtech i-nodů a bloků, o velikosti bloku, o volném prostoru a mnoho dalších. Superblok zabírá vždy právě jeden blok a povinně se nachází v první skupině bloků. V původní verzi ext2 byl superblok obsažen také ve všech ostatních skupinách bloků, což zabíralo nezanedbatelné množství diskového prostoru. V pozdějších revizích došlo ke zredukování počtu skupin bloků, které obsahují superblok. Novější revize definuje, že kopie superbloku jsou udržované ve skupinách bloků č. 0, 1 a dále ve skupinách jejichž číslo je násobkem 3, 5 a 7.

Struktura superbloku se v průběhu vývoje mění rozšiřováním. Autoři v původním návrhu totiž s rozšiřováním počítali, a proto ve struktuře najdeme několik různě velkých položek, které pouze rezervují místo pro budoucí využití. Veškeré hodnoty jsou uloženy v pořadí bajtů little-endian.

Nyní si popíšeme jednotlivé položky ve struktuře superbloku. Na obrázku 3.3 je tato struktura schematicky zobrazena. Jedná se již o novější revizi 1 (tzv. dynamická revize).

- `s_inodes_count` (4 bajty) obsahuje celkový počet i-nodů v celém oddílu.
- `s_blocks_count` (4 bajty) obsahuje celkový počet bloků v celém oddílu.
- `s_r_blocks_count` (4 bajty) obsahuje počet bloků vyhrazený superuživateli. Na systémech unixového typu se jedná o uživatele *root*. Rezervované bloky poskytují záložní prostor při nedostatku místa na disku, ale je třeba si uvědomit, že i tyto bloky mohou být vyčerpány, pokud nedostatek místa na disku zavíní právě superuživatel.
- `s_free_blocks_count` (4 bajty) udržuje počet volných bloků.
- `s_free_inodes_count` (4 bajty) udržuje počet volných i-nodů.
- `s_first_data_block` (4 bajty) obsahuje číslo prvního datového bloku.

struct ext2\_superblock



Obrázek 3.3: Datová struktura superbloku (dynamická revize)

- `s_log_block_size` (4 bajty) obsahuje velikost bloku, přičemž skutečná velikost se vypočítá tak, že se číslo 1024 bitově posune doleva právě o velikost této položky (tj.  $1024 * 2^{s\_log\_block\_size}$ ).
- `s_log_frag_size` (4 bajty) je označována jako zastaralá. Hodnota udává velikost fragmentu, přičemž výpočet se provádí stejně jako u velikosti bloku.
- `s_blocks_per_group` (4 bajty) obsahuje počet bloků ve skupině bloků. Společně s položkou `s_first_data_block` udává hranice skupin bloků. Slouží též pro výpočet velikosti bitmapy bloků.
- `s_frags_per_group` (4 bajty) je označována jako zastaralá. Udává počet fragmentů ve skupině.
- `s_inodes_per_group` (4 bajty) obsahuje počet i-nodů ve skupině bloků, z něhož lze vypočítat velikost bitmapy i-nodů.
- `s_mtime` (4 bajty) udává čas poslední úspěšné operace mount (připojení oddílu). Používá se unixový čas, definovaný normou POSIX.
- `s_wtime` (4 bajty) udává čas posledního zápisu. Opět je použit unixový čas.
- `s_mnt_count` (2 bajty) udává počet připojení oddílu od poslední kontroly oddílu.
- `s_max_mnt_count` (2 bajty) udává maximální počet připojení oddílu od poslední kontroly. Pokud hodnota položky `s_mnt_count` dosáhne tohoto maxima, musí být provedena úplná kontrola oddílu.
- `s_magic` (2 bajty) obsahuje magické číslo, které identifikuje souborový systém. Podle této hodnoty se rozpoznává, že jde o souborový systém ext2 nebo vyšší. Magická hodnota je v hexadecimálním tvaru rovna číslu 0xEF53.
- `s_state` (2 bajty) indikuje stav souborového systému. Během operace mount (připojení oddílu) se tato hodnota kontroluje a pokud indikuje chybový stav, oddíl se nepřipojí. Před připojením je nutné provést kontrolu a případnou opravu souborového systému specializovaným programem (v Linuxu se jedná o fsck).

- `s_errors` (2 bajty) určuje chování ovladače, pokud je detekována chyba. V dokumentaci jsou popsány tři možné scénáře. Je možné chybu ignorovat a pokračovat. Dále je možné připojit oddíl pouze pro čtení. Poslední možností je vyvolat kritickou chybu systému (v Linuxu se označuje termínem *kernel panic*).
- `s_minor_rev_level` (2 bajty) obsahuje vedlejší číslo verze souborového systému. Spolu s hodnotou položky `s_rev_level` dostaneme kompletní číslo verze.
- `s_lastcheck` (4 bajty) udává čas od poslední kontroly souborového systému. Opět je použit unixový čas.
- `s_checkinterval` (4 bajty) udává maximální časový interval (unixový čas) mezi kontrolami souborového systému.
- `s_creator_os` (4 bajty) identifikuje operační systém, ve kterém byl souborový systém vytvořen. Čísla některých významných operačních systémů jsou specifikována. V žádném případě není seznam operačních systémů vyčerpávající.
- `s_rev_level` (4 bajty) obsahuje hlavní číslo verze.
- `s_def_resuid` (2 bajty) obsahuje výchozí ID uživatele pro rezervované bloky. V Linuxu je hodnota nulová (uživatel **root**).
- `s_def_resgid` (2 bajty) obsahuje výchozí ID skupiny pro rezervované bloky. V Linuxu je hodnota nulová (skupina **root**).

Další položky jsou dostupné od revize souborového systému č. 1, která se označuje jako dynamická.

- `s_first_ino` (4 bajty) obsahuje číslo prvního nerezervovaného i-nodu. V původní verzi souborového systému ext2 je tato hodnota rovna 11.
- `s_inode_size` (2 bajty) obsahuje velikost struktury i-nodu v bajtech. V původní verzi je tato hodnota vždy 128. Ve vyšších revizích musí být hodnota mocninou 2 a musí být menší nebo rovna velikosti bloku.
- `s_block_group_nr` (2 bajty) udává číslo skupiny bloků, ve které se nachází tento superblok. Superblok je totiž udržován ve více kopiích, aby bylo možné jej v případě havárie rekonstruovat ze zálohy.

- `s_feature_compat` (4 bajty) je bitová maska obsahující kompatibilní vlastnosti. Ovladač může tyto vlastnosti implementovat, ale nemusí. Není zde riziko poškození struktur souborového systému. Namátkou můžeme jmenovat žurnálování nebo adresářový index (HTree).
- `s_feature_incompat` (4 bajty) je bitová maska nekompatibilních vlastností. Pokud ovladač některou z těchto vlastností nepodporuje, neměl by umožnit připojení oddílu.
- `s_feature_ro_compat` (4 bajty) je bitová maska kompatibilních vlastností v režimu pouze pro čtení. Pokud ovladač některou z těchto vlastností nepodporuje, měl by povolit připojení pouze v režimu pro čtení.
- `s_uuid` (16 bajtů) slouží jako identifikátor oddílu. Hodnota by měla být různá pro každé dva oddíly.
- `s_volume_name` obsahuje název diskového oddílu. Příliš se nevyužívá. Délka je omezena na 15 znaků. Řetězec by měl být ukončen nulou a obsahovat pouze znaky podle normy ISO-Latin-1.
- `s_last_mounted` udává adresář, kde byl oddíl naposledy připojen. Běžně se nevyužívá, ale lze použít pokud není adresář pro připojení specifikován. Maximální délka řetězce je stanovena na 63 znaků. Řetězec by měl být ukončen nulou a obsahovat pouze znaky podle normy ISO-Latin-1.
- `s_algorithm_usage_bitmap` (4 bajty) je bitmapa určující použité kompresní algoritmy.

Následující položky se týkají možného zvýšení výkonu.

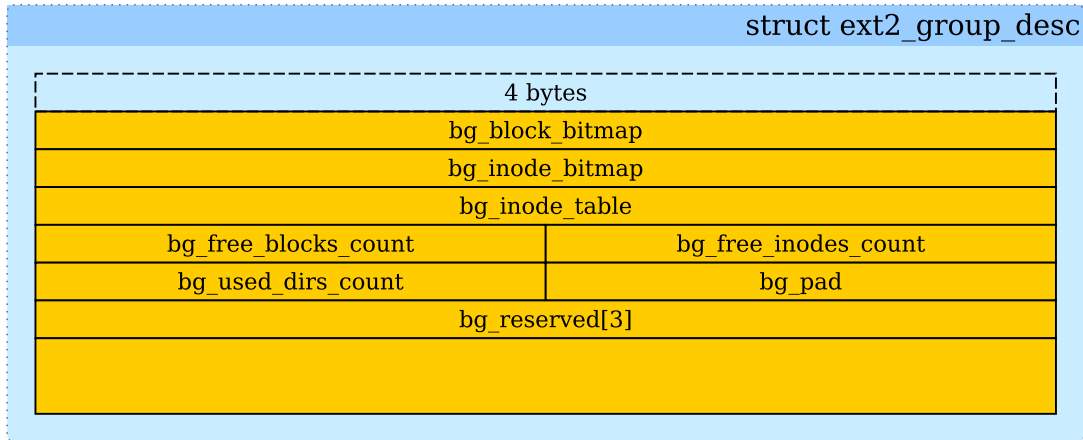
- `s_prealloc_blocks` (1 bajt) udává počet bloků, které může ovladač zkusit dopředu alokovat při vytváření nového souboru.
- `s_prealloc_dir_blocks` (1 bajt) udává počet bloků, které může ovladač zkusit dopředu alokovat při vytváření nového adresáře.

Zbytek struktury superbloku je označen jako rezervovaný prostor, jehož velikost doplňuje délku superbloku na 1024 bajtů.

### 3.2.4 Tabulka deskriptorů skupin bloků

Tabulka deskriptorů skupin bloků obsahuje informace o každé skupině bloků zvlášť. Měla by se nacházet v těch samých skupinách bloků, v nichž je i superblok. Velikost tabulky deskriptorů je závislá na počtu skupin bloků v daném oddílu.

Struktura deskriptoru skupiny bloků (tj. jednoho prvku tabulky deskriptorů) je zobrazena na obrázku 3.4.



Obrázek 3.4: Datová struktura deskriptoru skupiny

Na začátku jsou tři čtyřbajtové položky (`bg_block_bitmap`, `bg_inode_bitmap` a `bg_inode_table`) reprezentující čísla bloků, v nichž se nachází příslušné datové struktury. Za nimi jsou umístěny dvě dvoubajtové položky (`bg_free_blocks_count` a `bg_free_inodes_count`), které obsahují informace o počtu volných datových bloků a i-nodů. Další dvoubajtová položka `bg_used_dirs_count` udává počet použitých adresářů ve skupině. Následují ještě dvě položky, které jsou určeny pouze k zarovnání (`bg_pad`) a k vyhrazení místa pro využití v některé z dalších verzí souborového systému (`bg_reserved`).

Deskriptor skupiny bloků tedy udržuje informace o volném místě ve skupině a další zajímavé údaje, díky kterým není nutné navštěvovat každou skupinu například při hledání volného místa pro nový soubor apod.

### 3.2.5 Bitová mapa datových bloků

Jak název napovídá, jedná se o klasickou bitovou mapu. Tato bitová mapa slouží k označení volných a použitých bloků. Bit nastavený na nulu označuje volný blok a bit nastavený na jedničku obsazený blok. Z toho, že každý bit odpovídá jednomu

datovému bloku ve skupině, vyplývá omezení počtu datových bloků v rámci skupiny. Bitová mapa datových bloků zabírá nejvýše jeden blok.

### 3.2.6 Bitová mapa i-nodů

Protože jsme ještě nezadefinovali pojem i-node, spokojíme se prozatím s velmi zjednodušeným vysvětlením, že jde o datovou strukturu, reprezentující soubor. Tato bitová mapa má úplně stejnou funkci jako výše uvedená bitová mapa datových bloků. Rozdíl je pouze v tom, že jednotlivé bity označují ne bloky, ale volné a obsazené i-nody v tabulce i-nodů. Velikost této bitové mapy činí opět nejvýše jeden blok, což přináší omezení počtu i-nodů v tabulce.

### 3.2.7 Tabulka i-nodů

Tabulka i-nodů je datová struktura, ve které jsou už samotné i-nody uloženy. Velikost tabulky je dána maximálním počtem i-nodů ve skupině, který je určen v superbloku. Každá skupina bloků obsahuje relevantní část této tabulky.

### 3.2.8 I-node

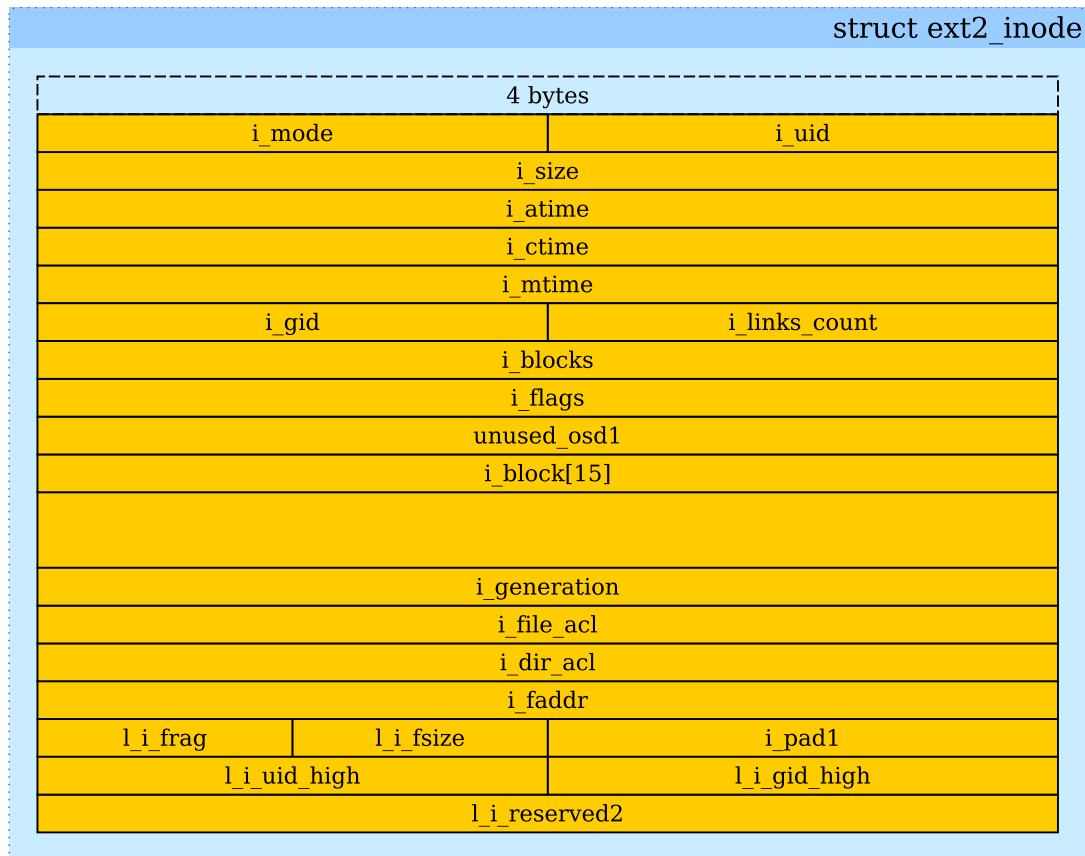
I-node je datová struktura, která obsahuje všechny důležité informace o souboru (vlastník, práva, velikost, uložení na disku) na diskovém oddílu. Souborem je v tomto kontextu myšlen nejen klasický soubor jako množina souvisejících dat, ale také adresář, speciální soubor (zařízení, socket) nebo symbolický link. Je třeba zdůraznit, že **jméno souboru není součástí struktury i-node**, i když se mnoho lidí mylně domnívá, že tomu je naopak. Jméno souboru je uloženo v adresáři, do něhož daný soubor patří.

Na obrázku 3.5 je znázorněna grafická podoba datové struktury i-nodu.

Význam některých datových položek si popíšeme podrobněji:

- `i_mode` (2 bajty) obsahuje informace o typu souboru (soubor, adresář symbolický odkaz, blokové nebo znakové zařízení, pojmenovaná roura) a také o přístupových právech k souboru (RWX model a další).
- `i_uid` (2 bajty) obsahuje dolní polovinu identifikátoru vlastníka souboru. Dvoubajtový identifikátor v některých systémech nestačí, a proto jsou v položce `osd2` vyhrazeny další 2 bajty (v Linuxu).





Obrázek 3.5: Datová struktura i-nodu

- `i_size` (4 bajty) udává velikost souboru v bajtech. 4-bajtová hodnota je ovšem schopna postihnout pouze soubory o velikosti do 4 GiB. Kvůli tomu se později využila položka `i_dir_acl`, která byla přejmenována a ukládá se do ní vyšších 32 bitů velikosti souboru.
- `i_atime`, `i_ctime`, `i_mtime`, `i_dtime` (všechny mají velikost 4 bajty) jsou časové značky, udávající kdy byl soubor naposled otevřen, kdy byl vytvořen, kdy byl naposled změněn a také kdy byl smazán.
- `i_gid` (2 bajty) obsahuje dolní polovinu identifikátoru vlastníčí skupiny. Situace s velikostí identifikátoru je naprosto stejná jako u `i_uid`.
- `i_links_count` (2 bajty) udává počet linků vedoucí na tento i-node. Toto číslo reprezentuje počet záznamů směřujících na tento i-node z adresářů. Jestliže vytvoříme obyčejný soubor, nastaví se tato hodnota na jedničku. Pokud potom vytvoříme tzv. hardlink na tento soubor, zvýší se hodnota čítače o jedničku.

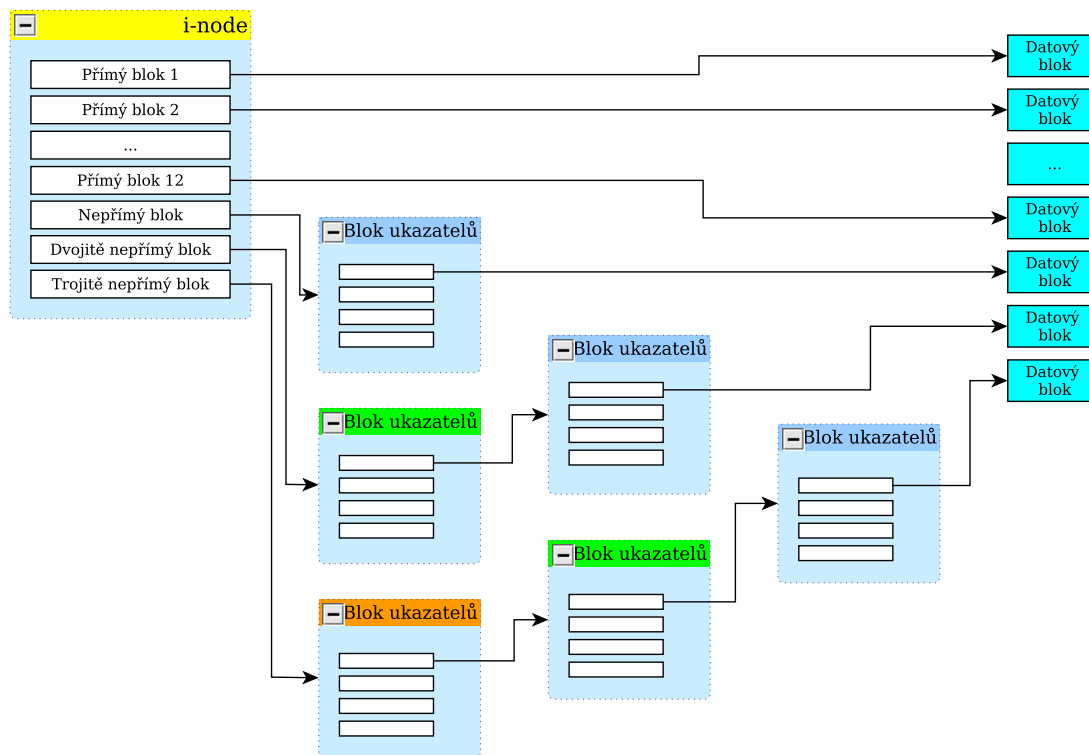
V případě smazání původního souboru jsou data přístupná přes hardlink a naopak. Data se stanou nedostupnými, pokud hodnota `i_links_count` klesne na nulu. Nezapočítávají se symlinky, které mají svůj vlastní i-node.

- `i_blocks` (4 bajty) udává počet bloků alokovaných pro daný soubor. Termín blok je v tomto případě lehce zavádějící, protože pro účely tohoto čítače počítáme bloky o velikosti 512 bajtů (běžná velikost diskového sektoru) a to bez ohledu na skutečnou velikost bloku v souborovém systému.
- `i_flags` (4 bajty) obsahuje příznaky používané souborovým systémem, mezi něž patří například volba vyplňování bloků náhodným obsahem při mazání nebo synchronní aktualizace souboru namapovaného do paměti.
- `unused_osd1` (4 bajty) je specifická struktura pro různé operační systémy. V implementaci ovladače pro HelenOS pravděpodobně také nebude využita.
- `i_blocks[EXT2_N_BLOCKS]` je pole čtyřbajtových čísel, které adresují na datové bloky. V jádře Linux je konstanta `EXT2_N_BLOCKS` nastavena na 15. Z toho prvních 12 ukazuje přímo na data. Další odkaz na blok je nepřímý, což znamená, že odkazovaný blok neobsahuje data, ale další pole ukazatelů na datové bloky. Další dva ukazatele ukazují na dvojitě nepřímý blok a trojitě nepřímý blok. Jejich význam je obdobný jako u nepřímého bloku. Dvojitě nepřímý ukazatel se odkazuje na nepřímý blok, v němž jsou ukazatele na datové bloky a trojitě nepřímý ukazatel se odkazuje na dvojitě nepřímý blok, v němž jsou ukazatele na nepřímé bloky. Pro lepší představu poslouží obrázek 3.6.
- `i_generation` (4 bajty) se používá v NFS<sup>1</sup> k určení verze souboru.
- `i_file_acl` (4 bajty) je odkaz na blok s ACL<sup>2</sup> pro soubor.
- `i_dir_acl` (4 bajty) je odkaz na blok s ACL pro adresář.
- `i_faddr` (4 bajty) je adresa bloku, ve kterém se nachází poslední fragment souboru.
- Zbytek položek je sdružen do struktury `osd2` (celkem 12 bajtů). Struktura je specifická pro různé operační systémy. Je možné ho využít pro zvětšení rozsahu čísel (např. `l_i_uid`, `l_i_gid`).

---

<sup>1</sup>Network File System - síťový souborový systém

<sup>2</sup>Access Control List - pokročilý způsob nastavení přístupových práv



Obrázek 3.6: Schéma přímých a nepřímých bloků

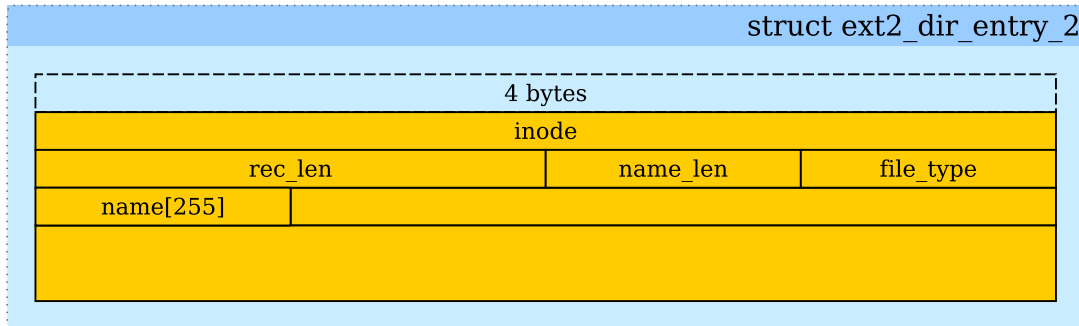
### 3.2.9 Soubory

Klasický soubor obsahuje data, která jsou uložena v jednotlivých datových blocích příslušného i-nodu. Souborový systém ext2 podporuje takzvané *sparse* soubory, které asi nejlépe vystihuje české slovo „děravé“. Takový soubor má ve skutečnosti alokováno méně datových bloků, než kolik odpovídá jeho velikosti. Při pokusu načíst data z logického bloku, pro který není naalokován fyzický blok, pak ovladač vrací blok vyplněný nulami. Fyzický blok se alokuje až ve chvíli, kdy se do něj někdo pokusí zapsat. V ext2 má *sparse* soubor v i-nodu nastavený flag indikující tento stav. Ukazatel na nenaalokovaný datový blok má hodnotu 0. V případě *sparse* souboru nemusí korespondovat velikost (v bajtech) s počtem využitých diskových bloků (512 bajtů).

### 3.2.10 Adresáře

Důležitou datovou strukturou v souborovém systému je adresář. Adresáře tvoří stromovou strukturu celého souborového systému. V podstatě se jedná o speciální soubor, který má svůj vlastní i-node, kde je v položce `i_mode` hodnota označující adresář. Adresář obsahuje záznamy o svých potomcích (souborech, adresářích, ...), které jsou

uloženy v jednosměrném spojovém seznamu. Struktura adresářové položky (prvek spojového seznamu) je znázorněna na obrázku 3.7.



Obrázek 3.7: Datová struktura adresářové položky

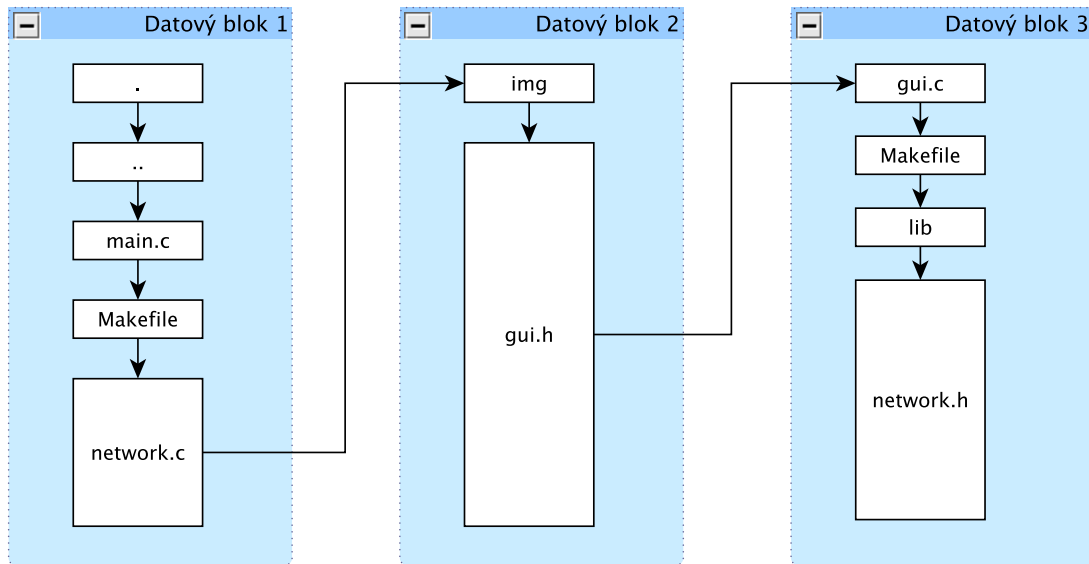
Každý adresářový záznam obsahuje především název souboru (`name`), délku názvu (`name_len`) a čtyřbajtové číslo i-nodu (`inode`). V pozdějších revizích se původně dvoubajtová délka názvu zkrátila na jeden bajt a uvolněné místo se využívá na identifikaci typu souboru (`file_type`), díky čemuž není potřeba přistupovat k i-nodu pro zjištění typu.

Název souboru může mít maximální délku 255 znaků, což je hodnota konstanty `EXT2_NAME_LEN`). Tato maximální délka vyplývá rovněž z číselného rozsahu datové položky pro délku názvu v novější revizi. V názvu souboru může být libovolný znak kromě lomítka, kterým se oddělují jednotlivé části cesty k souboru. Název také nesmí obsahovat znak s hodnotou 0, kterým se řetězec s názvem ukončuje.

Další neméně důležitou informací je dvoubajtová délka záznamu (`rec_len`), která slouží pro procházení seznamu. Přičtením délky záznamu k adrese právě načteného seznamu se získá adresa následujícího záznamu. Délka záznamu musí být zarovnána na 4 bajty a nesmí přesahovat přes hranice bloku. Nevyužitý prostor v adresářové položce (za názvem ukončeným nulovým znakem) by měl být vyplněn samými nulami. V každém datovém bloku adresáře musí být alespoň jeden záznam. Pokud je záznam na začátku bloku neplatný (smazaná položka), není mu přidělen žádný i-node. Položka `inode` má pak nulovou hodnotu. Při mazání souboru se délka předchozího záznamu prodlouží tak, aby správně ukazovala na další adresářový záznam. Na obrázku 3.8 je vidět, že některé záznamy jsou zřetelně delší kvůli předchozímu mazání.

Velké množství adresářových záznamů se negativně projeví na výkonu. Právě kvůli objemným adresářům vznikla implementace adresářového indexu, která bude

popsána později. Indexy jsou kompatibilní i se staršími verzemi ovladačů, takže pokud ovladač indexování neimplementuje, pracuje pouze se spojovým seznamem.



Obrázek 3.8: Jednoduchý příklad adresáře, ve kterém se již mazalo

### 3.2.11 Rozšířené atributy

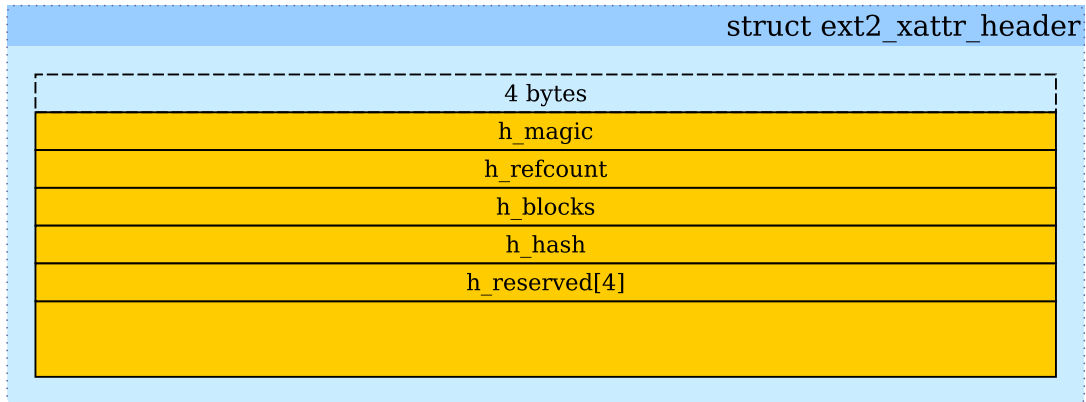
Rozšířené atributy jsou dvojice *název:hodnota*, které jsou asociovány se soubory a adresáři. Tyto atributy jsou uloženy ve speciálním datovém bloku, který nepatří žádnému i-nodu. Jednotlivé i-nody se pak na tento blok odkazují pomocí položky `i_file_acl` (pro soubory) nebo `i_dir_acl` (pro adresáře). Práce (čtení i zápis) s rozšířenými atributy je atomická.

Prostřednictvím rozšířených atributů se nejčastěji realizují bezpečnostní vylepšení (např. Access Control List) nebo jiné přídavné funkcionality souborového systému.

Datový blok s rozšířenými atributy začíná hlavičkou, která je zobrazena na obrázku 3.9. Po hlavičce následují jednotlivé atributy, které se přidávají za sebe. Za posledním atributem jsou čtyři nulové bajty. Hodnoty atributů se ukládají do bloku od konce a rostou tedy proti hlavičkám atributů. Mezi atributy a jejich hodnotami je nevyužitý prostor, který se při přidávání atributů zmenšuje z obou stran.

Význam jednotlivých položek je následující.

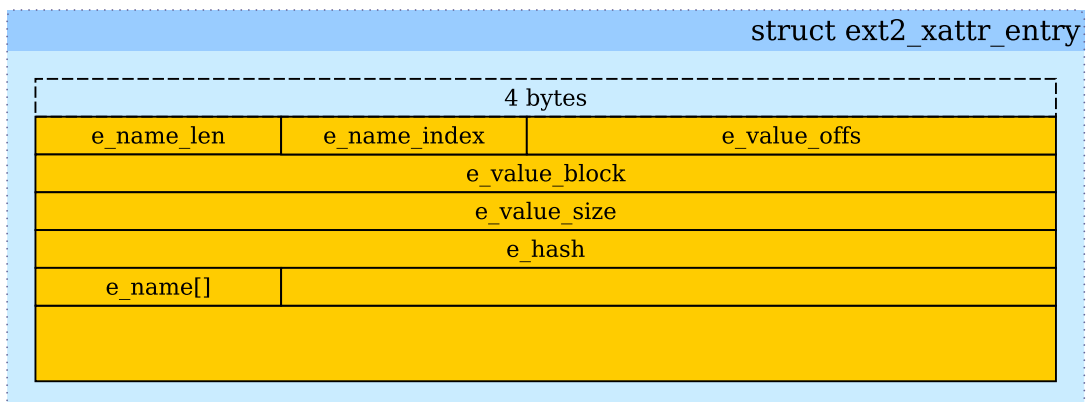
- `h_magic` (4 bajty) obsahuje magické číslo pro kontrolu hlavičky.



Obrázek 3.9: Hlavička bloku s rozšířenými atributy

- **h\_refcount** (4 bajty) je čítač referencí (tj. kolik i-nodů se na tento blok odkazuje). Pokud čítač klesne na nulovou hodnotu, blok lze bezpečně uvolnit.
- **h\_blocks** (4 bajty) udává počet bloků, které jsou využity pro rozšířené atributy. V Linuxu je možno využívat pouze jeden blok, čímž je prostor pro rozšířené atributy omezený. Vyšší počet bloků než 1 je vyhodnocen jako chyba.
- **h\_hash** (4 bajty) obsahuje hash všech hashů vypočtených z hlaviček jednotlivých záznamů.
- **reserved** (16 bajtů) je položka připravená pro budoucí využití.

Každý atribut je uvozen hlavičkou se strukturou znázorněnou na obrázku 3.10.



Obrázek 3.10: Hlavička rozšířeného atributu

Význam jednotlivých položek je následující.

- `e_name_len` (1 bajt) obsahuje délku názvu atributu.
- `e_name_index` (1 bajt) je index názvu atributu.
- `e_value_offs` (2 bajty) udává offset hodnoty atributu v rámci bloku.
- `e_value_block` (4 bajty) udává číslo bloku, ve kterém leží hodnota atributu. Při výše uvedeném omezení velikosti na 1 blok nemá tato položka příliš význam.
- `e_value_size` (4 bajty) udává délku hodnoty atributu.
- `e_hash` (4 bajty) je hash názvu a hodnoty atributu.
- `e_name` obsahuje název atributu. Délku názvu určuje položka `e_name_len`.

### 3.2.12 Komprese

Podporou komprese je v tomto případě myšleno transparentní komprimování dat bez použití externích programů. Veškeré operace souborového systému jsou z pohledu vyšších vrstev nezměněné a teprve ovladač se stará o zmíněnou kompresi. Souborový systém ext2 je z pohledu datových struktur připraven na použití kompresních mechanismů. Linuxový ani žádný jiný známý ovladač ovšem kompresi přímo neimplementuje. Pro Linux existuje patch s názvem `e2compr`, který funkcionalitu pro kompresi do ovladače přidává.

## 3.3 Souborový systém ext3

Souborový systém ext3 používá kompatibilní, ale poněkud rozšířené datové struktury jako jeho předchůdce ext2. Hlavním rozdílem mezi oběma systémy je takzvané žurnálování v ext3, které bude nosným tématem této podkapitoly. Kromě žurnálování se do ext3 dostaly i „méně významné“ novinky. Občas se jistě hodí možnost změny velikosti oddílu za běhu, i když nejde o akci, která se provádí každý den. Další novou vlastností je změna datové struktury adresáře. V ext2 byl adresář implementován jako spojový seznam, ale v ext3 byl volitelně nahrazen indexovací strukturou označovanou jako HTree, což je stromová struktura podobná známým B-stromům. HTree využívá hašování názvů souborů pro indexování a není třeba ho vyvažovat. Díky použití HTree se mohou objevit problémy s kompatibilitou ext2 a ext3. Přesto existují patche, které do ext2 práci s HTree doplňují, ale tyto nejsou součástí oficiální implementace v jádře Linuxu. Míra kompatibility mezi ext2 a ext3 je tedy poměrně vysoká a to jak zpětná, tak i dopředná.

### 3.3.1 Změny datových struktur

V souborovém systému ext3 došlo k rozšíření struktury superbloku (viz obrázek 3.11). Nové položky jsou z větší části určeny pro podporu žurnálování.

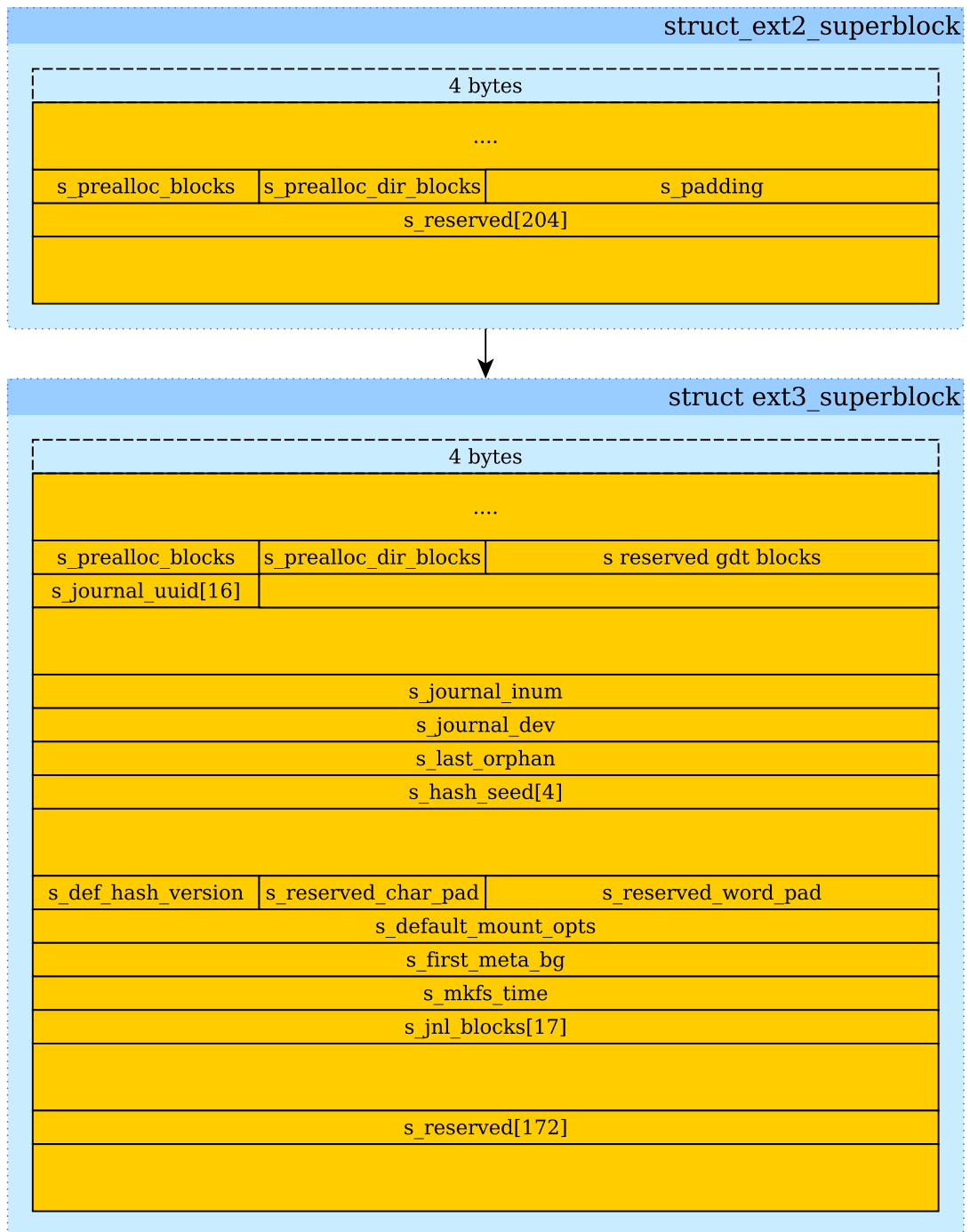
Následuje popis nově vytvořených položek.

- `s_reserved_gdt_blocks` (2 bajty) se využívá pro uložení počtu rezervovaných bloků pro případné zvětšování oddílu.
- `s_journal_uuid` je pole bajtů obsahující unikátní identifikátor žurnálu.
- `s_journal_inum` (4 bajty) obsahuje číslo i-nodu, který reprezentuje soubor s žurnálem.
- `s_journal_dev` (4 bajty) obsahuje číslo zařízení, na kterém se nachází soubor s žurnálem.
- `s_last_orphan` (4 bajty) ukazuje na první prvek seznamu i-nodů ke smazání.
- `s_hash_seed` je pole obsahující čtyři čtyřbajtová čísla, která slouží jako výchozí hodnoty pro hašovací funkce. Hašování se používá pro indexování položek v adresáři.
- `s_def_hash_version` (1 bajt) udává výchozí hašovací funkci, která se použije při vytváření nového indexu v adresáři.
- `s_reserved_char_pad` (1 bajt) `s_reserved_word_pad` (2 bajty) slouží k zarovnání na čtyři bajty.
- `s_default_mount_opts` (4 bajty) nese výchozí nastavení pro připojení oddílu.
- `s_first_meta_bg` (4 bajty) je ukazatel na první meta skupinu, což je seskupení metadat z více skupin do jedné. To umožňuje delší souvislou posloupnost datových bloků.
- `s_mkfs_time` (4 bajty) obsahuje časový údaj o vytvoření oddílu.
- `s_jnl_blocks` je pole obsahující zálohu čísel datových bloků (4 bajty) žurnálového souboru. Záloha slouží pro případ poškození souvisejícího i-nodu.

Struktura i-nodu se na konci rozšířila o dvě dvoubajtové položky. První nová položka `i_extra_isize` udává dodatečnou velikost i-nodu, pokud je v superbloku nastavena větší než standardní velikost. Jedná se o kontrolní mechanismus, protože



musí platit invariant, který říká, že velikost i-nodu uvedená v superbloku musí být stejná jako součet standardní velikosti i-nodu a právě položky `i_extra_size`. Další položka (`i_pad1`) slouží pouze jako zarovnání na 4 bajty.



Obrázek 3.11: Rozšíření superbloku v souborovém systému ext3

### 3.3.2 Žurnálování

Operace se soubory (nebo s adresáři) na úrovni diskového zařízení nejsou atomické. Například vytvoření souboru se skládá z mnoha kroků od hledání volného místa pro uložení dat přes zapsání metadat do struktury i-node a úpravy superbloku (všech jeho kopií) až po zápis vlastních souborových dat do datových bloků. Žurnálování je bezpečnostní mechanismus, který slouží k zajištění atomicity operací na úrovni disku a tím chrání integritu dat. Funguje na bázi transakčního zpracování, které známe například z databázových systémů. Obecně není nutné, aby byl žurnál na stejném disku nebo oddílu jako samotná data, ale u ext3 jsou data i žurnál na stejném oddílu.

Popis funkce žurnálování se dá obecně shrnout do čtyř po sobě jdoucích kroků.

1. Před vlastním zápisem dat se do žurnálu vytvoří záznam o tom, jaká data a kde se budou zapisovat.
2. Proveďte se zápis dat na disk.
3. Záznam v žurnálu se označí jako úspěšně provedený.
4. Záznam je z žurnálu odstraněn.

Je zřejmé, že žurnálování se uplatní pouze při zápisech, protože během čtení se žádná data na disku nemění. Pokud dojde k havárii (např. výpadek napájení), nemusí se při dalším připojování souborového systému zdoluhavě procházet a kontrolovat celý oddíl, ale stačí projít soubor s žurnálem a dokončit nebo zrušit všechny transakce zaznamenané v žurnálu, což trvá podstatně kratší dobu než kompletní kontrola celého diskového oddílu.

V ext3 existují tři typy (spíše úrovně) žurnálování. Liší se tím, zda se žurnálují souborová data, metadata nebo oba dva typy najednou. Uvedené úrovně žurnálování se liší rychlostí, prostorovou náročností a také bezpečností.

- **writeback** - Žurnálují se pouze metadata. To je výhodné z pohledu rychlosti, ale na druhou stranu je tím snížena bezpečnost z pohledu integrity dat. Při výpadku se například může stát, že metadata jsou již změněna, ale obsah odpovídajících datových bloků ještě nebyl přepsán, protože v okamžiku výpadku byla data pro zápis teprve v bufferu. Operační systém totiž používá různé algoritmy pro optimalizaci pohybů diskových hlaviček (např. algoritmus obousměrného výtahu) a proto může ve frontě dojít ke změně pořadí jednotlivých zápisů.

- **ordered** - Tento typ žurnálování řeší problém předchozího typu (writeback). Opět se žurnálují pouze metadata, ale dříve než jsou v žurnálu označena jako zapsaná, je vynucen zápis souborových dat na disk. Teprve poté se metadata v žurnálu označí jako úspěšně zapsaná. Vynucení zápisu dat sice trochu zpomalí práci s diskem, ale zároveň je dosaženo vyšší bezpečnosti v oblasti integrity dat. Žurnálování na úrovni ordered je vhodným kompromisem a ve většině systémů je nastaveno jako výchozí volba.
- **journal** - Žurnálují se metadata i samotná data, čímž se dosáhne vysoké míry bezpečnosti, za kterou se ovšem platí snížením výkonu a také vyššími prostorovými nároky na žurnálovací soubor. Pozorný čtenář lehce nahlédne, že při tomto typu žurnálování se data vlastně na disk zapisují dvakrát - nejdříve do žurnálu a teprve pak do datové oblasti.

V Linuxu je žurnálování implementováno na obecnější úrovni a poskytuje rozhraní pro jednotlivé ovladače. Každý ovladač pak volá funkce z tohoto rozhraní a obecný mechanismus pak provádí samotný žurnálovací proces a transakce.

### 3.3.3 Indexování adresářů - HTree

U adresářů s mnoha záznamy dochází při použití sekvenčního procházení spojového seznamu ke snižování výkonu. V roce 2001 bylo představeno vylepšení v podobě stromového indexu využívajícího hašování. Adresářový index včetně algoritmů popisuje velmi podrobně článek [2]. Zde si velmi stručně popíšeme základní rysy.

Adresář využívající index má v položce `i_flags` svého `i-node` tuto skutečnost zaznamenanou. Pro zachování kompatibility byl zvolen následující způsob uložení indexových struktur v datovém prostoru adresáře. První dvě adresářové položky (`.` - tečka pro aktuální adresář a `..` - dvě tečky pro nadřazený adresář) jsou zakomponovány do kořene indexového stromu. Délka druhé položky (`..`) je nastavena tak, aby dosahovala až na konec bloku. V takto připraveném prostoru se nachází kořen indexového stromu. Stejný způsob se používá i pro uložení indexových uzlů nižší úrovně. Každý uzel zabírá celý blok a na začátku bloku se nachází falešný adresářový záznam, který má opět délku nastavenou na celý blok. Současná implementace podporuje pouze stromy o maximální výšce 1. Šikovně nastavené délky adresářových záznamů tedy skryjí indexové struktury před implementacemi, které indexování nepodporují.

Uzel indexového stromu obsahuje na začátku počet a maximální počet prvků. Prvek uzlu obsahuje hash a logické číslo bloku v adresáři, kde se mají hledat záznamy,

kteře odpovídatí hodnotě hashe. Prvky jsou v uzlu seřazeny vzestupně a díky tomu lze použít binární vyhledávání, které celý proces ještě urychlí.

Algoritmus vyhledávání je jednoduchý a proto si ho popíšeme velmi stručně. Nejdříve se provede test indexové struktury, který by měl odhalit porušený index. Poté se vypočítá hash hledaného souboru. Získaný hash se používá pro vyhledávání v indexovém stromě. V listovém uzlu indexového stromu se získá číslo bloku, kde se má daný záznam hledat. Takový blok se musí prohledat sekvenčním způsobem. V případě, že záznam není nalezen, testuje se, zda se má hledat dále v následujícím listu. Pokud se zjistí porušený index, přejde ovladač na sekvenční procházení.

Odebrání záznamu z indexovaného adresáře probíhá analogicky jako z adresáře bez indexu. Rozdílný je pouze způsob vyhledání záznamu. U indexovaného adresáře se použije vyhledávací algoritmus uvedený v předchozím odstavci. Samotné odebrání záznamu pak probíhá stejně jako u neindexovaného adresáře.

Algoritmicky zajímavější je proces vkládání nového adresářového záznamu. Listový indexový blok se najde uvedeným vyhledávacím algoritmem. Zde je důležité, že si ovladač drží v paměti celou cestu od kořene k listu. V listovém uzlu se najde odkaz na datový blok, do kterého záznam patří podle hodnoty hashe. Pokud je v cílovém datovém bloku dostatek místa, záznam se do něj vloží a celý algoritmus končí úspěchem. Jestliže ovšem není v cílovém datovém bloku dostatek místa, přichází na řadu nutnost štěpení. V první fázi se ověří, jestli je v listovém indexovém uzlu dostatek místa pro vložení nového záznamu. Jestliže je indexový blok plný, musí dojít též k jeho rozštěpení. To může způsobit kaskádu štěpení až ke kořenovému uzlu. Teprve poté, co je jisté, že se do listového indexového bloku vejde nový záznam, dojde ke štěpení datového bloku.

Při štěpení datového bloku se všechny platné záznamy setřídí vzestupně podle hash hodnoty. Setříděné záznamy se rozdělí do dvou datových bloků, přičemž se nedělí podle počtu záznamů, ale podle velikosti v bajtech. Záznamy s nižší hodnotou se vloží do původního datového bloku a zbývající pak do nově alokovaného datového bloku. Nakonec se vloží do listového indexového bloku záznam odkazující na nově alokovaný datový blok. Hash hodnota nového záznamu se určí podle prvního záznamu. Pokud došlo k rozdělení záznamů tak, že poslední záznam z dolní poloviny má stejnou hash hodnotu jako první z horní poloviny, tak se k hash hodnotě pro nový datový blok přičítá 1, což indikuje kolizi. Vkládání nového záznamu do listového bloku nenarušuje původní setřídění. Všechny následující záznamy se musí posunout o jeden záznam dále, což není moc efektivní, ale velmi to urychlí vyhledávací algoritmus.

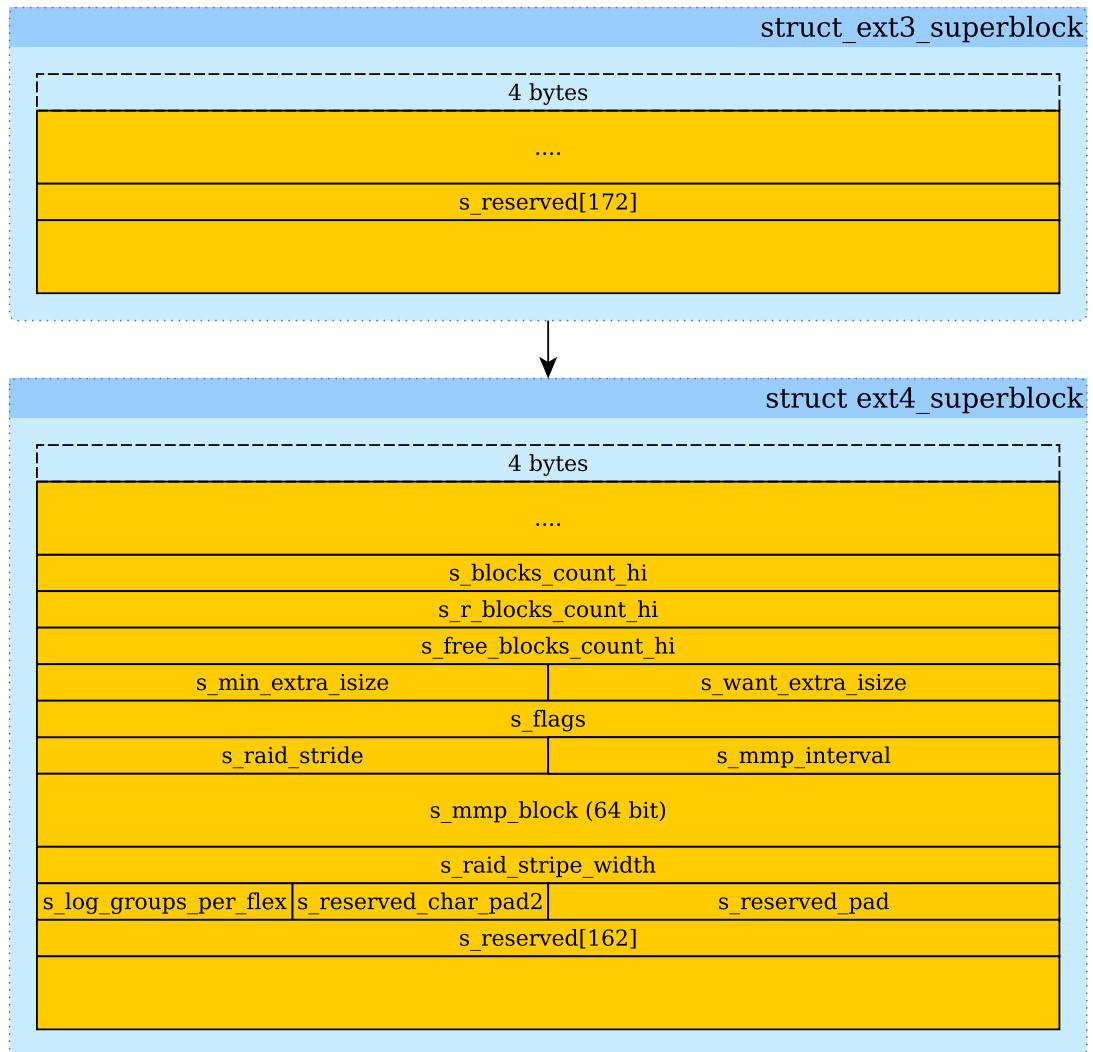
## 3.4 Souborový systém ext4

Souborový systém ext4 přinesl poměrně dost nových vlastností, které jsou popsané na stránce [4] a zde se s nimi stručně seznámíme.

### 3.4.1 Změny datových struktur

Na konci struktury superbloku z předchozí generace (ext3) stále zbývá dostatek volného prostoru, který je reprezentován poslední položkou, ze které vzniklo několik nových položek. Tyto nové položky se týkají především podpory 64-bitových rozsahů čísel, ale i dalších nových vlastností. Rozložení nových položek v superbloku ukazuje obrázek 3.12.

- První tři nové čtyřbajtové položky `s_blocks_count_hi`, `s_r_blocks_count_hi` a `s_free_blocks_count_hi` představují rozšíření datového typu pro počty bloků. Každá z těchto položek má svoji druhou polovinu s odpovídajícím názvem na začátku superbloku.
- `s_min_extra_isize` (4 bajty) udává minimální velikost i-nodu v bajtech.
- `s_want_extra_isize` (4 bajty) udává počet bajtů, které jsou rezervovány v každém novém i-nodu.
- `s_flags` (4 bajty) slouží pro nastavení různých vlastností pomocí bitmapy.
- `s_raid_stride` (2 bajty) a `s_raid_stripe_width` (4 bajty) slouží pro podporu diskových polí neboli RAID. První položka udává velikost jednoho úseku (chunk) a druhá pak říká, kolik disků je použito.
- `s_mmp_interval` (2 bajty) obsahuje časový interval v sekundách. Během kontroly vícenásobného připojení oddílu (při operaci mount) se čeká na změnu kontrolních dat tolik sekund, kolik je uvedeno v této položce.
- `s_mmp_block` (8 bajtů) reprezentuje číslo bloku, ve kterém jsou data, sloužící pro kontrolu vícenásobného připojení oddílu. Zjednodušeně daný blok obsahuje číslo, které se periodicky mění.
- `s_log_groups_per_flex` (1 bajt) udává počet skupin bloků v jedné flexibilní skupině, pokud se tato vlastnost používá.
- `s_reserved_char_pad2` (1 bajt) a `s_reserved_pad` (2 bajty) slouží k zarovnání velikosti na 4 bajty.



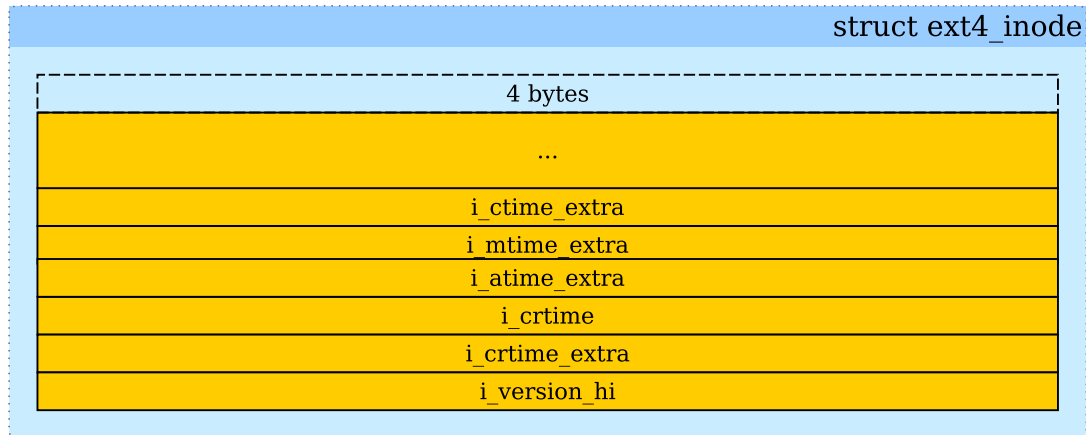
Obrázek 3.12: Rozšíření superbloku v souborovém systému ext4

- `s_reserved[162]` reprezentuje zbývající nevyužitý prostor superbloku.

Ve struktuře i-nodu přibylo na konci 6 čtyřbajtových položek, které ukazuje obrázek 3.13 Kromě poslední se jedná o položky, které jsou důležité ke zpřesnění časových údajů na nanosekundy. Poslední položka `i_version_hi` rozšiřuje číselný rozsah položky `i_version`.

### 3.4.2 Zvýšení limitů

Limitujícím faktorem většiny souborových systémů je omezení velikosti oddílu a velikosti souboru. Oba limity vyplývají z rozsahu čísel (v bitech) použitých v datových



Obrázek 3.13: Rozšíření i-nodu v souborovém systému ext4

strukturách souborového systému. Právě zvětšením rozsahu dosáhli autoři nové maximální velikosti oddílu 1EiB a nové maximální velikosti souboru 16TiB. Pro srovnání uvedeme, že v ext3 je maximální velikost oddílu 16TiB a maximální velikost souboru 2TiB. Rozdíl je tedy poměrně značný a byl dosažen zvětšením rozsahu čísel ze 32 bitů na 48 bitů. Údaje vychází z předpokladu velikosti bloku 4KiB.

Další související vlastností je úplné odstranění limitu na počet podadresářů v adresáři, který byl v ext3 nastaven konstantou na 32 000 podadresářů. Úplného odstranění limitu je dosaženo pomocí vlastnosti `dir_nlink`. Při překročení hranice se počet odkazů na nadřazený adresář přestane zvyšovat. Navíc i bez zmíněné vlastnosti byl limit zvýšen na 65 000 podadresářů.

Původní limit 32 000 podadresářů existoval už v původních verzích linuxového jádra. Není úplně jasné proč je limit nastaven právě takto. Nabízí se několik vysvětlení.

- První možností je, že limit vyplývá z velikosti datového typu položky i-nodu `i_links`. Každý podadresář totiž ukazuje na svůj rodičovský adresář. Hodnota limitu je velmi blízká maximální hodnotě znaménkového dvoubajtového integeru. Již ve verzi linuxového jádra 1.0 je ale datový typ bezznaménkový. Ani v ext4 nebyla tato položka rozšířena, akorát byla hodnota limitu zvýšena na 65 000, což přibližně koresponduje s rozsahem položky `i_links`.
- Jako další možnost se nabízí výkonnostní důvody. Sekvenční průchod adresářem při velkém počtu položek může trvat poměrně dlouho. Nicméně počet souborů v adresáři nijak omezený není a při vypsání se také musí projít všechny záznamy.

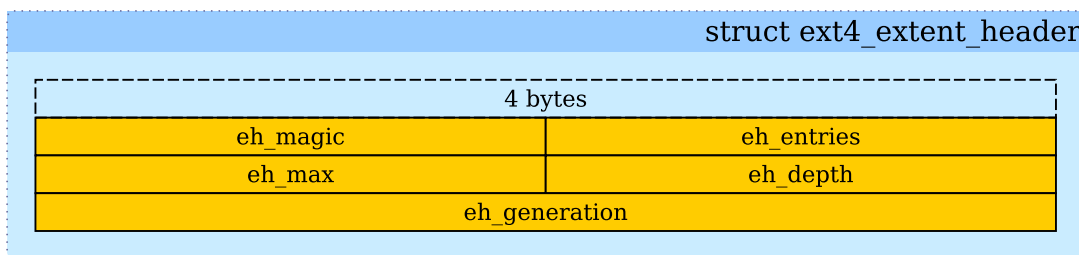


### 3.4.3 Extenty

Extent představuje nový přístup k mapování alokačních jednotek na disku. Velké soubory zabírají mnoho bloků, přičemž všechny je nutné adresovat (přímo, nepřímo apod.) z i-nodu. Extent sdružuje určité množství bloků, které na sebe fyzicky navazují a tím nahrazuje standardní adresování bloků. Tím lze účinně snížit fragmentaci díky podmínce navazujících bloků, což v důsledku zvyšuje rychlost diskových operací.

Pokud jsou v i-nodu použité extenty, nefunguje původní adresování bloků pomocí přímých a nepřímých bloků. Tím je ovšem narušena zpětná kompatibilita.

Datové struktury extentů jsou poměrně jednoduché a tvoří B-strom. Příklad stromové struktury extentů je znázorněn na obrázku 3.15. V položce `i_blocks` v i-nodu se místo odkazů na běžné přímé a nepřímé bloky nachází kořenový uzel extento-  
vého stromu. Každý uzel obsahuje na začátku hlavičku se strukturou znázorněnou na obrázku 3.14.

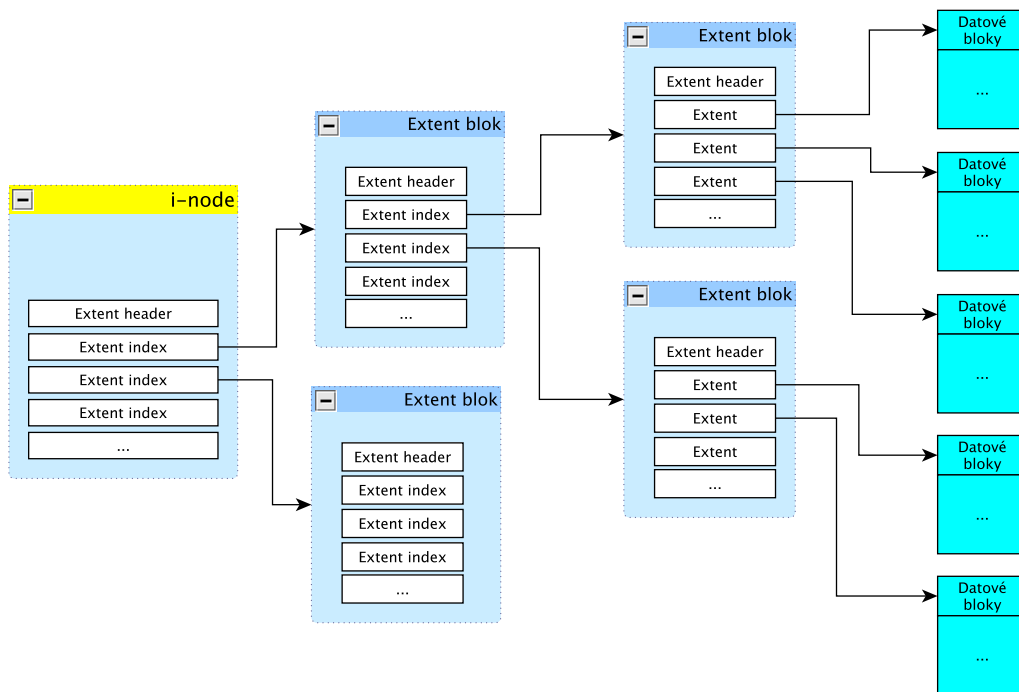


Obrázek 3.14: Datová struktura hlavičky extentu

Všechny položky jsou dvoubajtové, pouze poslední (`eh_generation`) je čtyřbajtová. Zajímavá je položka udávající počet záznamů v uzlu (`eh_entries`), dále položka s kapacitou uzlu (`eh_max`) a také položka s hloubkou podstromu (`eh_depth`). Podle hloubky podstromu se rozlišuje typ obsahu daného bloku. Pokud je hloubka nenulová, obsahuje daný blok záznamy odkazující na podřízené uzly. Každý takový záznam obsahuje pouze logické číslo bloku jako klíč B-stromu a ukazatel na fyzický blok obsahující podřízený uzel. V případě nulové hloubky podstromu, obsahuje daný blok přímo jednotlivé extenty. Extent udává, od kterého logického bloku začíná a kolik logických bloků obsahuje. Dále extent obsahuje adresu prvního fyzického datového bloku.

### 3.4.4 Vylepšení i-nodů

Struktura i-nodu doznala také jistých vylepšení.



Obrázek 3.15: Struktura datových bloků při použití extentů

- **Větší i-nody:** Velikost i-nodu lze konfigurovat již od ext3, ale došlo ke zvětšení výchozí velikosti ze 128 bajtů na 256 bajtů. Zvětšení velikosti je nutné, protože došlo ke zvětšení rozsahu časových údajů a k přidání nových rozšiřujících atributů.
- **Rezervování i-nodů při vytvoření adresáře:** Při zakládání adresáře se počítá s tím, že v něm vzniknou další soubory nebo adresáře, a proto dochází k rezervaci i-nodů pro tento nový adresář.
- **Zvýšení přesnosti časových známek:** Nově mají časové známky přesnost v nanosekundách oproti sekundové přesnosti v ext3.

## 3.5 Zajímavé vlastnosti v linuxovém ovladači ext4

### 3.5.1 Vícebloková alokace

V předchozích verzích linuxového ovladače se při vytváření nového souboru vždy alokoval pouze jeden blok (tj. běžně 1KiB - 8KiB diskového prostoru). To znamená, že alokátor se volal mnohokrát kvůli jednomu souboru, což je velmi neefektivní pro velké soubory. Ve víceblokové alokaci je možné používat lepší alokační algoritmus,

který dokáže počítat s celkovým počtem bloků potřebným k vytvoření souboru a lépe tak využít větší souvislé posloupnosti bloků, což může pomoci ke snížení míry fragmentace. Hlavním přínosem je samozřejmě zvýšení výkonu. Tento způsob alokace bloků se také využívá při práci s extenty.

### 3.5.2 Odložená alokace

Jak už název této vlastnosti napovídá, jde pouze o výkonnostní vylepšení. V předchozích verzích probíhala alokace vždy co nejdříve, ale přitom se data k zapsání udržovala ve vyrovnávací paměti. Odložená alokace umožňuje alokovat bloky až bezprostředně před zápisem dat na disk a v součinnosti s víceblokovou aktualizací dokáže opět dosáhnout vyššího výkonu (i s použitím extentů). Tato vlastnost se pozitivně projeví především u postupně rostoucích souborů.

Tato nová vlastnost může způsobit problémy se ztrátou dat. Data jsou uložena v operační paměti a dokud to není nezbytně nutné, na disk se zápis neprovádí. Pokud dojde k pádu systému nebo aplikace, data se již na disk nedostanou. Zápis lze v Linuxu vynutit například použitím volání `fsync`, což se ovšem může negativně projevit na výkonu při použití žurnálování.

### 3.5.3 Zrychlení fsck

Program `fsck` slouží ke kontrole a případné opravě souborového systému. Kontrola probíhá tak, že se prochází všechny i-nody, což signalizuje nezanedbatelnou dobu běhu. V ext4 došlo k vylepšení tohoto mechanismu tím, že se kontrolují pouze použité i-nody. Seznam nevyužitých i-nodů je uložen na konci tabulky i-nodů v každé skupině bloků, nicméně tento seznam nevytváří samotný ovladač ext4, ale program `fsck`, takže zrychlení bude patrné až v následujícím běhu.

### 3.5.4 Kontrolní součty při žurnálování

Žurnál je velmi exponovaná část souborového systému, do které se zapisuje velmi často a je proto více riziková z hlediska poruchy. Zotavení z porušeného žurnálu by mohlo mít za následek velké poškození dat. Díky kontrolnímu součtu je možné před zotavením zkontrolovat, zda je žurnál v pořádku a tím zabránit poškození dat.

### 3.5.5 Možnost vypnutí žurnálování

V některých specifických případech se může hodit možnost deaktivovat žurnálování, které představuje určité výkonnostní zatížení. Dojde tím sice ke snížení bezpečnosti, ale to nemusí vždy vadit.

### 3.5.6 Online defragmentace

Výše jmenované vlastnosti dokážou míru fragmentace snižovat, ale přesto může k fragmentaci dále docházet. Alokátory sice dokáží připravit souvislou posloupnost datových bloků pro daný soubor, ale při jeho pozdějším zvětšování už může dojít ke vzniku fragmentace. Nástroj `e4defrag` dokáže defragmentovat celý souborový systém nebo jen jednotlivé soubory a to na připojeném oddílu. Bohužel vývoj této komponenty ještě není ukončen a proto ji zatím v distribucích systému GNU/Linux nenajdeme.

### 3.5.7 Perzistentní prealokace

Tato vlastnost slouží k rezervaci prostoru pro daný soubor. Dříve se absence této vlastnosti řešila tak, že byl vytvořen soubor a jeho obsah se vyplnil nulami (binárními) podle požadované velikosti. To je poměrně neefektivní způsob, protože musí dojít k zápisu všech nul na disk a také není zaručena celistvost souboru z pohledu fragmentace. Prealokace proti tomu pouze vyhledá pokud možno souvislou posloupnost bloků a zarezervuje ji pro daný soubor patřičnou úpravou datových struktur souborového systému. Díky tomu nedochází k tak velké fragmentaci a disk není zatížen „zbytečným“ zápisem nul.

### 3.5.8 Bariéry

V ext4 jsou ve výchozím nastavení aktivní tzv. bariéry. Jedná se o vylepšení bezpečnosti při práci se žurnálem. Pokud je bariéra aktivní, musí mít souborový systém před potvrzením zápisu (`commit`) do žurnálu absolutní jistotu, že žurnálová data jsou opravdu zapsána na disk. Díky použití vyrovnávacích pamětí a určité „inteligenci“ dnešních disků je tento požadavek oprávněný, protože data se mohou ve vyrovnávacích pamětech zdržet netriviálně dlouhou dobu a navíc zápisy na disk jsou přeuspořádávány kvůli redukci pohybu diskových hlav. S použitím bariéry si tedy ovladač souborového systému explicitně vynutí dokončení diskových operací spojených se zápisem žurnálových dat před samotným potvrzením transakce (`commit`).

Obecně lze bariéru popsat jako bod na časové ose, kdy všechny zápisy před bariérou musí být dokončeny dříve, než začne jakýkoli zápis za bariérou. Použitím bariér se zvyšuje bezpečnost, ale samozřejmě se negativně projevují na výkonu. Bariéry lze deaktivovat při připojování oddílu.

## 3.6 Algoritmy

### 3.6.1 Alokace i-nodu

Celá alokace spočívá ve výběru skupiny bloků, kde už se pak vybere první volný i-node z příslušné bitmapy. V linuxovém ovladači se již delší dobu používá tzv. Orlovův alokátor, který rozlišuje výběr skupiny bloků podle typu entity (soubor, adresář), pro kterou se i-node alokuje. Při alokaci i-nodu v dané skupině se navíc testuje, jestli má vybraná skupina volné datové bloky.

**Adresář:** U nového adresáře se rozlišuje, zda bude umístěn přímo v kořenovém adresáři nebo někde hlouběji v adresářovém stromě. V prvním případě se hledá skupina bloků, kde je počet volných i-nodů a volných datových bloků větší než průměr ze všech skupin. V druhém případě zkusí alokátor využít stejnou skupinu, ve které je i nadřazený adresář, ale jen pokud splňuje všechny následující podmínky.

- Má dostatek volných i-nodů.
- Nemá obsazeno nadměrné množství adresářů.
- Má rozumný poměr adresářů a souborů.
- Má dostatek volných datových bloků.

K výpočtu výše uvedených hodnot se používají různé průměry a vzorce, jejichž konkrétní podoba není zajímavá a lze je zjistit ze zdrojových kódů linuxového jádra.

Pokud výběr skupiny podle předchozích pravidel (pro oba dva typy adresářů) selže, hledá se lineárně první skupina bloků (počínaje od skupiny nadřazeného adresáře), ve které je počet volných i-nodů větší nebo roven průměrnému počtu volných i-nodů ze všech skupin. Při extrémním zaplnění je průměr roven nule a tedy pokud existuje volný i-node, tak ho algoritmus najde.

**Soubor:** V případě souboru je situace následující. Algoritmus začíná ve skupině bloků, kde se nachází nadřazený adresář. Pokud neuspěje, zkusí prohledávat skupiny bloků tak, že k výchozímu číslu skupiny postupně zvětšuje o mocniny dvojky dokud přičítaná hodnota nepřesáhne počet skupin. Pokud ani tímto způsobem není nalezen volný i-node, projdou se lineárně všechny skupiny.

### 3.6.2 Alokace datového bloku

Algoritmus na alokaci datových bloků je stěžejní pro efektivitu ostatních operací souborového systému. Při nevhodném způsobu alokace dochází k fragmentaci, což je negativní jev, kdy datové bloky náležející k jednomu i-nodu nejsou uloženy souvisle za sebou a ve správném pořadí. Proti požadavku na co nejnižší fragmentaci stojí požadavek na rychlost alokace.

V linuxovém ovladači se nejdříve určí číslo bloku (tzv. *goal*), od kterého se má začít hledat volný blok, a to následovně:

- Pokud i-node nemá alokovan žádný datový blok, nastaví se goal na první datový blok ve skupině bloků, ve které se daný i-node nalézá.
- Pokud má i-node alokovaný alespoň jeden datový blok, nastaví se goal na adresu posledního alokovaného bloku zvětšenou o 1.

Po nalezení výchozího bloku (*goal*) začíná hledání volného bloku. Hledá se pouze dopředným směrem, tj. vzestupně dle číslem bloků.

Algoritmus lze stručně shrnout do následujících kroků:

1. Pokud je goal volný, alokuje se a algoritmus končí.
2. Pokud volný blok s číslem až  $(goal + 63) \bmod 63$ , alokuje se a algoritmus končí.
3. Pokud je volný celý bajt v bitmapě bloků počínaje od goal, vezme se první bit tohoto bajtu, poté algoritmus končí.
4. Pokud je volný alespoň samostatný bit v bitmapě bloku počínaje číslem goal, alokuje se a algoritmus končí.
5. Pokud nebyl dosud nalezen volný blok, pokračuje algoritmus v následující skupině bloků, kde se goal určí jako první datový blok dané skupiny. Takto může algoritmus projít všechny skupiny bloků a skončit v té skupině, kde začal.

6. Pokud ani v původní skupině, kde algoritmus začal, není nalezen volný blok (tentokrát před číslem goal), algoritmus končí neúspěchem.

Po úspěšné alokaci datového bloku musí být aktualizovány čítače volných bloků v deskriptoru skupiny bloků a také v superbloku.

### 3.6.3 Vytvoření souboru / adresáře

Pro úspěšné vytvoření souboru stačí alokovat volný i-node výše uvedeným algoritmem a poté přidat záznam do nadřazeného adresáře. Není třeba alokovat žádné datové bloky.

U adresářů je navíc nutné alokovat nejméně jeden datový blok, protože každý adresář obsahuje nejméně dva záznamy pro odkaz na sebe sama a na nadřazený adresář. Navíc se po vytvoření adresáře musí zvýšit čítač použitých adresářů v deskriptoru skupiny bloků.

### 3.6.4 Uvolňování datových bloků

Zde se nejedná přímo o algoritmus, ale spíše o popis přístupu k uvolňování bloků různých typů. Při změně velikosti souboru se uvolňují pouze datové bloky, avšak nepřímé zůstávají. U adresářů nedochází k uvolňování bloků vůbec. Při smazání adresářové položky se ta předchozí patřičně prodlouží nebo pokud byla mazaná položka na začátku bloku, vytvoří se zde nový neplatný záznam, který ukazuje na i-node 0. Určitou optimalizaci adresářů pak provádí utilita `fsck`.

### 3.6.5 Mazání souboru / adresáře

Algoritmus mazání má dvě fáze. Nejdříve proběhne krok, který se v souborových systémech označuje jako operace `unlink`. U adresáře se před touto operací ještě ověřuje, jestli je adresář prázdný. Operace `unlink` spočívá v tom, že dojde ke smazání záznamu v nadřazeném adresáři a dále se sníží počet odkazů v i-nodu.

Pokud je po provedení této operace počet odkazů na i-nodu roven nule, může dojít k uvolnění datových bloků a po nastavení času smazání i k uvolnění celého i-nodu.

Po uvolnění datových bloků a i-nodu je třeba aktualizovat čítače volných bloků a i-nodů v deskriptoru skupiny bloků a v superbloku. U adresáře je navíc potřeba snížit čítač použitých adresářů ve skupině bloků.

# 4. Návrh

## 4.1 Architektura

V systému HelenOS jsou ovladače souborových systémů implementovány jako samostatné tasky v uživatelském prostoru. Tohoto konceptu se bude držet i ovladač pro ext4. Jako velmi přínosné se ukázalo rozdělení ovladače na nezbytnou serverovou část a na knihovnu, které je provedeno v ovladači souborového systému ext2. Serverová část zajišťuje komunikaci s VFS serverem implementací výstupního VFS protokolu. Knihovna potom zajišťuje práci s jednotlivými datovými strukturami souborového systému. To umožňuje vytvoření různých dalších programů pracujících se souborovým systémem. Vzhledem k velmi úzké příbuznosti obou zmíněných souborových systémů bude ovladač systému ext4 vycházet přímo z kódu ovladače systému ext2. Původní kód je nutné zrevidovat kvůli rozsáhlým změnám datových struktur v ext4, zvětšení číselných rozsahů u některých atributů apod. Původní ovladač ext2 obsahuje pouze podporu pro čtení, ale nový ovladač ext4 bude samozřejmě podporovat i zápis. Stejně je také zachování zpětné kompatibility, takže prostřednictvím ovladače ext4 bude možno pracovat i se staršími oddíly naformátovanými jako ext2 a ext3.

Rozdělení ovladače na dvě části musí být provedeno tak, aby knihovni část nebyla na serverové závislá a bylo možné knihovnu využít například k implementaci nějakého programu, který by pracoval přímo s diskovým oddílem bez nutnosti použití ovladače.

## 4.2 Volba podporovaných vlastností

Souborový systém ext4 a linuxový ovladač obsahují velmi mnoho zajímavých vlastností. Implementace všech funkcionalit ovladače by pravděpodobně přesáhla rozsah této práce a proto je třeba zvážit, které vlastnosti je rozumné implementovat v zájmu dostatečné míry interoperability.

Nezbytně nutná je plná podpora pro čtení a zápis připojeného souborového systému ext4 a také jeho předchůdců.

**Adresářový index (HTree)** se objevuje již v novějších revizích souborového systému ext2. Účelem adresářového indexu je zrychlení operací s adresářem (vyhledávání, mazání, vkládání). Tato vlastnost se běžně používá skoro na každém diskovém oddílu, proto by bylo velmi vhodné tuto vlastnost implementovat. Z pohledu kom-



patibility není podpora indexu nutná, nicméně z pohledu výkonnosti bude vhodné tuto vlastnost do ovladače zapracovat.

**Extenty** představují novinku v souborovém systému ext4. Vynecháním implementace podpory extentů by se významně snížila interoperabilita. Diskové oddíly formátované jako ext4 mají ve výchozím nastavení podporu extentů aktivní. Implementace podpory extentů je tedy naprosto nezbytná.

**Kompresa** představuje určitou úsporu místa na disku. Ani linuxové ovladače však neobsahují podporu transparentní komprese přímo ve svém kódu. Vzhledem k pracnosti implementace a k velmi malému množství dostupných informací se implementace podpory komprese nevyplatí.

**Rozšířené atributy** by v operačním systému HelenOS prozatím nenašly využití. Není zde podpora pro uživatelská oprávnění. Ani v případě připojení diskového oddílu, který rozšířené atributy podporuje, není třeba je využívat. Implementace této vlastnosti se tedy prozatím jeví jako nepotřebná.

**Odložená inicializace** diskových struktur se využívá pro urychlení formátovacího procesu. Implementace inicializačních funkcí by neměla působit problémy a proto bude součástí ovladače.

**Flexibilní skupiny bloků** jsou jednou z dalších pokročilých vlastností, které tak trochu narušuje zavedenou diskovou strukturu, protože metadata skupiny jsou umístěna jinde, než podle původní specifikace. Je to určitě zajímavá, ale ne úplně nezbytná vlastnost a proto bude implementována pouze v případě dostatku času po dokončení částí, které jsou z pohledu tohoto návrhu povinné.

**Žurnálování** je silný bezpečnostní mechanismus a proto by bylo vhodné jeho podporu do ovladače implementovat. Z pohledu compatibility však implementace žurnálování není nezbytně nutná. Dle analýzy kódu v jádře Linuxu je žurnálování realizováno jako obecnější kód, který využívá více ovladačů. U operačního systému HelenOS by bylo rozumné zvážit, zda by nebylo vhodné se vydat také cestou obecnějšího frameworku pro žurnálování. Podpora žurnálování bude tedy chybět v prototypové verzi ovladače, která bude výstupem této práce, protože bude lepší nejdříve odladit verzi, která korektně pracuje se všemi diskovými strukturami. V některé z dalších verzí bude samozřejmě možné a vhodné tento mechanismus implementovat, i když

to bude pravděpodobně znamenat netriviální zásah do kódu. U všech funkcí bude nutné použít transakční zpracování, aby bylo možné se v případě havárie dostat diskový oddíl do konzistentního stavu. Knihovna pro práci s diskovými bloky podporuje přímý zápis na zařízení, který obejde cache, což se pro potřeby žurnálování přesně hodí. Zásahům do kódu se pravděpodobně při budoucí implementaci nebude možné vyhnout, ale rozhodně by nemělo být potřeba reimplementovat celý ovladač. Je definováno několik úrovní žurnálování a náročnost jejich implementace se samozřejmě bude lišit, podle toho, která data se budou žurnálovat (metadata, data apod.). Vyšší úrovně pak řeší i to, aby operační systém opravdu data fyzicky zapsal na disk a nezůstaly pouze ve frontě.

### 4.3 Využití dalších komponent

Pro úspěšnou implementaci bude nutné využít některých již existujících komponent operačního systému HelenOS.

Přístup k diskovému oddílu je zajištěn pomocí ovladače `ata_bd`. Pro funkčnost ovladače `ext4` je tedy nutné zajistit jeho běh. Pro přístup k datovým blokům bude ovladač využívat možnosti knihovny `libblock`, která poskytuje správnou míru abstrakce. Knihovna pracuje s bloky o velikosti, kterou nadefinujeme při inicializaci. Poskytuje také podporu cacheování, které lze s výhodou využít pro zrychlení práce. Cacheování ovšem snižuje míru bezpečnosti dat při pádu.

Dále bude využito již existujících datových struktur. Zejména se jedná o implementaci spojových seznamů a hashovací tabulky v serverové části ovladače. Také bude nutné využít knihovní funkce pro práci s dynamicky alokovanou pamětí a určité také funkce a makra sloužící pro převod pořadí bajtů (endianita), protože HelenOS je multiplatformní operační systém.

Při realizaci adresářového indexu se jistě bude hodit nějaký třídící algoritmus. V systému HelenOS lze najít například implementaci algoritmu Quicksort, který je efektivní a pro účely ovladače zcela postačuje.

Při analýze bylo zjištěno, že v OS HelenOS není implementováno zpracování reálného data a času. Současná implementace poskytuje pouze tzv. *uptime*, neboli čas od startu systému. Tento problém je nutné při implementaci nějak vyřešit, protože při práci s *i-nody* se při některých operacích časové údaje musí aktualizovat.

Serverová část potřebuje pro svou práci navíc standardní komponenty pro komunikaci meziprocesovou (IPC). Vstup a výstup pomocí konzole bude omezen pouze na výpis ladících hlášení, pokud bude ovladač zkompileován s podporou ladících výpisů.

## 4.4 Přidružené programy

Implementace nebude zahrnovat programy označované souhrnně jako `e2fsprogs`. Do této skupiny programů patří například `dumpe2fs`, který vypíše obsah základních datových struktur diskového oddílu, `e2fsck`, který slouží pro kontrolu a opravu diskového oddílu, program `mke2fs`, který vytváří nový diskový oddíl a `tune2fs`, který dokáže měnit parametry existujícího souborového systému. Zmíněná skupina programů nemá vliv na práci s funkčním diskovým oddílem a není ani součástí originálního ovladače v Linuxu. V rámci této práce nebudou tyto programy implementovány.

# 5. Implementace

V této kapitole se budeme věnovat konkrétní implementaci ovladače. Často se budeme odkazovat na zdrojové kódy a jejich umístění. Adresáře budou vždy uvedeny relativní cestou od adresáře, do kterého uložíme zdrojové kódy dle přílohy A. Pokud tedy budou zdrojové kódy umístěny v adresáři HelenOS, bude cesta vždy začínat prvním podadresářem.

Výsledná implementace ovladače se dělí na serverovou část a knihovnu, což odpovídá původní implementaci ovladače souborového systému `ext2`<sup>1</sup>, ze kterého tento ovladač vychází. Serverová část obsahuje funkce pro spuštění ovladače, dále funkce implementující operace knihovny `libfs` a také funkce implementující výstupní protokol VFS serveru. Knihovna pak obsahuje přímo operace s datovými strukturami souborového systému a definice datových struktur.

Implementace je realizována v programovacím jazyce `C`, stejně jako zbytek operačního systému HelenOS.

## 5.1 Serverová část ovladače

Zdrojové kódy serverové části (server se jmenuje `ext4fs`) se nachází v adresáři `uspace/srv/fs/ext4fs`.

Výchozím bodem serveru je soubor `ext4fs.c`, který obsahuje pouze spouštěcí funkci `main()`. Tato jediná funkce se stará o inicializaci ovladače a jeho zaregistrování do VFS. Poté se přepne do režimu serveru, který čeká na příchozí požadavky a vyřizuje je. Komunikace probíhá přes IPC.

Stežejním souborem serverové části je soubor `ext4fs_ops.c`, ve kterém jsou implementovány všechny požadované operace výstupního VFS protokolu a knihovny `libfs`. Ukazatele na funkce jsou pak uloženy do struktur `ext4fs_libfs_ops` (pro knihovnu `libfs`) a `ext4fs_ops` (pro výstupní VFS protokol). Dále jsou v tomto souboru implementovány jednoduché funkce pro práci s hashovací tabulkou. Několik dalších pomocných funkcí sloužící především k rozdělení kódu do menších celků a k možnosti vícenásobného využití. Zbytek této podkapitoly bude věnován právě tomuto souboru, proto nebude dále znovu zdůrazňováno, že se funkce a struktury nacházejí právě v něm.

---

<sup>1</sup>Ovladač souborového systému `ext2` v současnosti implementuje pouze operace pro čtení.

### 5.1.1 Datové struktury

V serverové části jsou definovány dva složené datové typy, které jsou díky své položce `link` zařaditelné do spojového seznamu.

`ext4fs_instance_t` je datová struktura určená k uchování informací o připojeném diskovém oddílu. Instance se vytvoří při připojení a zruší při odpojení oddílu. Během práce s oddílem se z této struktury pouze čte. Obsahuje především údaje o zařízení, počtu otevřených uzlů a odkaz na instanci souborového systému jako takového.

`ext4fs_node_t` slouží k uložení informací o uzlu, které jsou specifické pro souborový systém. Obsahuje ukazatel na výše zmiňovanou instanci, dále pak na příslušný `i-node` a také na obecný `fs_node`, se kterým pracuje knihovna `libfs` a který obaluje typ `ext4fs_node_t`. Dále obsahuje počítadlo referencí.

### 5.1.2 Funkce

Funkce v souboru `ext4fs_ops.c` jsou velmi podobné svým ekvivalentům z dalších ovladačů souborového systému. Ve většině případů funkce z uvedeného souboru pracují s interními datovými strukturami a volají funkce z knihovny `libext4`, do kterých předávají již pouze parametry, jejichž datové typy jsou definované přímo v knihovně. Výjimkou jsou samozřejmě číselné proměnné. Většina zajímavých funkcí pracujících přímo s datovými strukturami je odsunuta do knihovny.

## 5.2 Knihovní část ovladače

Knihovna `libext4` tvoří samostatnou a nezávislou součást implementovaného ovladače. V případě budoucího vytváření programů na práci s diskovými strukturami (například `fsck`) lze tuto knihovnu jednoduše využít. Pro externí programy stačí využít hlavičkový soubor `libext4.h`. Zdrojové kódy knihovny jsou umístěny v adresáři `uspace/lib/ext4`.

S výjimkou souborů `libext4.h` a `libext4_types.h` se v uvedeném adresáři nachází vždy dvojice implementačního a hlavičkového souboru. Soubory jsou členěny zhruba podle jednotlivých datových struktur souborového systému.

V knihovně jsou funkce, které již přímo pracují s diskovými datovými strukturami. Mimo triviálních funkcí pro čtení a zápis datových struktur (viz níže), jsou

v knihovně implementovány algoritmy, které pokud možno odpovídají těm z linuxového ovladače.

Velkou výjimkou je například algoritmus na alokaci i-nodu. V Linuxu je implementován sofistikovanější algoritmus, který pro svůj běh potřebuje informace o rodičovském i-nodu. Bohužel API pro ovladače v systému HelenOS tyto informace neposkytuje a tedy bylo nutné implementovat vlastní algoritmus, který triviálním způsobem vybere první volný i-node na disku.

### 5.2.1 Datové struktury

Datové struktury jsou definovány v hlavičkovém souboru `libext4_types.h`. Diskové datové struktury jsou vytvořeny dle specifikace souborového systému. Na disku se data do těchto struktur ukládají v pořadí little-endian. Z toho důvodu se po vzoru ovladače souborového systému `ext2` v OS HelenOS používají funkce pro přístup k jednotlivým položkám diskových struktur. Tyto funkce jsou obdobou getterů a setterů z objektového programování. Funkce typu `getter` provádí konverzi z pořadí little-endian do pořadí počítače a funkce typu `setter` provádí opačnou konverzi. Navíc dle verze připojeného souborového systému rozkládají větší čísla do více položek (např. 64 bitové číslo do dvou 32 bitových položek). Pro všechny datové struktury je vždy definován odpovídající datový typ, aby se v hlavičkách funkcí a v definicích proměnných nemuselo užívat klíčové slovo `struct`.

`ext4_filesystem_t` je struktura sdružující základní informace o konkrétním připojeném souborovém systému. Obsahuje proto identifikátor zařízení, na kterém se souborový systém nachází, a také ukazatel na instanci superbloku v paměti. Navíc jsou v této struktuře dvě krátká pole, do kterých se při inicializaci uloží limity jednotlivých úrovní nepřímé adresace bloků v i-nodu. První z nich udává maximální číslo logického bloku, které se vejde do dané úrovně nepřímé adresace. Druhá pak udává počet bloků, které daná úroveň nepřímé adresace pojme. Tyto dvě položky tedy nahrazují opakované výpočty a tím zrychlují práci s bloky v i-nodu.

`ext4_block_group_ref_t` a `ext4_inode_ref_t` mají stejnou úlohu. Obě obsahují ukazatel na příslušnou diskovou datovou strukturu, dále pak ukazatel na datový blok obsahující diskovou datovou strukturu a příznak, zda byla disková struktura změněna. Struktura `ext4_inode_ref_t` navíc obsahuje i logické číslo i-nodu.

`ext4_hash_info_t` obsahuje informace potřebné pro hashování, které se používá v implementaci adresářového indexu (HTree).

`ext4_directory_search_result_t` je interní struktura, která se používá jako výsledek vyhledávání v adresáři. Obsahuje ukazatel na datový blok a pak také ukazatel na konkrétní adresářový záznam v daném bloku.

`ext4_extent_path_t` je opět interní datová struktura, která slouží k podobnému účelu jako ta předchozí. Při prohledávání extentového stromu se alokuje pole tohoto typu, kam se ukládají jednotlivé uzly po cestě od kořene do listu. Pole se alokuje ještě delší kvůli případnému růstu stromu do hloubky při vkládání nového extentu. Obsahuje ukazatel na datový blok a hlavičku extentového uzlu, což je redundantní, ale usnadňuje a zpřehledňuje to následný kód. Mimo to obsahuje struktura indikátor hloubky a ukazatele na aktuální extentový index nebo přímo na extent. Rozhoduje se právě podle hloubky. Při nulové hloubce je platná položka `extent` a při nenulové hloubce položka `index`.

## 5.2.2 Funkce

Původní funkce z ovladače souborového systému ext2 podporovaly pouze čtecí operace. Byly přidány funkce umožňující zápis a v mnoha případech bylo nutné upravit i funkce čtecí. Nyní si stručně popíšeme jednotlivé implementační soubory. Přehled souborů a implementovaných funkcí není vyčerpávající a jsou zde popsány jen zajímavější funkce.

`libext4_balloc.c` obsahuje implementaci alokátoru datových bloků. Všechny funkce v souboru mají v názvu prefix `ext4_balloc_`. Veřejné jsou pouze funkce pro alokaci (`alloc_block`, `try_alloc_block`) a uvolnění datového bloku `free_block`, `free_blocks`. Použitý alokační algoritmus je stejný jako v linuxové implementaci a odpovídá popisu v podkapitole 3.6.2. Nalezení čísla bloku, od kterého se má začít (goal), je odděleno do samostatné statické funkce `find_goal`. Pro potřeby extentů slouží alokační funkce, která se pokusí naalokovat konkrétní fyzický blok. Úspěšnost je indikována ve výstupním parametru. Uvolnění bloku je velmi jednoduché, protože stačí vypočítat číslo skupiny bloku, načíst bitmapu a nastavit příslušný bit na nulu. Ostatní statické funkce poskytují implementaci triviálních výpočtů a přepočtů čísel bloků, skupin apod. V případě uvolňování souvislého úseku bloku se využívá druhá funkce, která optimalizuje práci s bitmapou tak, že stačí jedno načtení bloku

s bitmapou a všechny bloky se uvolní v jednom volání. Podmínkou je, aby celý úsek náležel do jedné skupiny bloků. Funkce se používá při práci s extenty, kde je tato vlastnost splněna.

**libext4\_bitmap.c** implementuje velmi jednoduché a obecné funkce pro práci s bitovými mapami, které se používají v alokátoru bloků i v alokátoru i-nodů. Kromě nastavování konkrétního bitu jsou zde umístěny funkce, které umí najít volný bit nebo celý bajt. Vyhledání celého volného bajtu se používá při hledání vhodného datové bloku pro alokaci, čímž se s pomocí vhodné volby výchozího bloku snižuje fragmentace. Názvy funkcí jsou dostatečně popisné a nemá smysl zde jejich tělo popisovat. Složitější je pouze funkce `ext4_bitmap_free_bits`, která je určena pro uvolnění souvislého úseku bloků v bitmapě. Nejdříve se jednotlivě uvolní několik bitů, aby došlo k zarovnání na celé bajty a poté se uvolňuje po celých bajtech. Nakonec se opět jednotlivě uvolní případných několik zbývajících bloků.

**libext4\_block\_group.c** obsahuje pouze funkce pro čtení a zápis položek datové struktury `ext4_block_group_t`. Pro skupiny bloků nejsou potřeba žádné další speciální operace.

**libext4\_crc.c** obsahuje implementaci výpočtu kontrolního součtu převzatou kompletně z Linuxu. Funkce jsou upravené pouze z hlediska datových typů a používají se pro výpočet kontrolního součtu ve struktuře skupiny bloků.

**libext4\_directory\_index.c** obsahuje implementaci operací s adresářovým indexem. Všechny funkce mají v názvu prefix `ext4_directory_dx_`. Kromě operací s položkami datových struktur indexu, se zde nachází funkce pro vyhledání `find_entry` a přidání nové adresářové položky `add_entry`. Ostatní funkce jsou statické a mají pomocný charakter pro zkrácení a deduplikaci kódu ve veřejných funkcích. Je zde také definována datová struktura `ext4_dx_sort_entry_t`, která slouží pro třídění záznamů podle hash hodnoty. K třídění je pro potřeby algoritmu quicksort naprogramován i komparátor na porovnávání definovaného datového typu `entry_comparator`. Pomocné funkce pak implementují vkládání nového záznamu do uzlu indexového stromu `insert_entry`, štěpení datového bloku `split_data` a štěpení indexového bloku `split_index`.

**libext4\_directory.c** implementuje práci s adresářem. Všechny funkce mají v názvu prefix `ext4_directory_`. Soubor obsahuje funkce pro čtení a zápis položek da-



tové struktury adresářového záznamu. Dále jsou zde funkce převzaté z původního `ext2` ovladače pro práci s adresářovým iterátorem, který slouží pro jednoduché procházení celého adresáře. Hlavní jsou tři funkce pro nalezení `find_entry`, odstranění `remove_entry` a vložení nového adresářového záznamu `remove_entry`. Další pomocné funkce jsou veřejné, protože jsou volány také z funkcí pracujících s adresářovým indexem. Jedná se konkrétně o funkce pro zápis nového adresářové záznamu `write_entry`, pro vyhledání záznamu v daném bloku `find_in_block` a pro pokus o zápis do vybraného datového bloku `try_insert_entry`.

`libext4_extent.c` obsahuje implementaci práci s extenty. Všechny funkce mají v názvu prefix `ext4_extent_`. Početně převládají funkce pro čtení a zápis datových položek všech tří datových struktur používaných při práci s extenty. Algoritmicky zajímavější jsou ale složité funkce na konci souboru. První dvě statické funkce (`binsearch` a `binsearch_idx`) implementují binární vyhledávání v uzlu extentového stromu. Jedna slouží pro vyhledávání v listu a druhá pro vyhledávání v indexovém uzlu. Funkce `find_block` slouží pro vyhledání datového bloku bez potřeby načítat do paměti celou cestu od kořene do listu. Vždy se načte uzel, v něm se vyhledá adresa příslušného potomka a blok se opět vrátí. Takto se dojde až do listu, odkud se přečte požadované číslo fyzického bloku. Statická funkce `find_extent` slouží pro vyhledání extentu, přičemž vrací cestu od kořene k listu uloženou v poli typu `ext4_extent_path_t`. Obě vyhledávací funkce používají při prohledávání uzlu výše zmíněné funkce pro binární vyhledávání. Další statická funkce `release` slouží pro uvolnění extentu a `release_branch` pak pro uvolnění celého podstromu. Druhá zmiňovaná funkce je rekurzivní. Projde celý podstrom, uvolní všechny extenty a poté i indexové záznamy. Uvolňovací funkce jsou využity veřejnou funkcí `release_blocks`, která funguje tak, že uvolní všechny bloky od zadaného logického bloku až do konce souboru, což se využívá při operaci `truncate`. Opakem pro uvolňování jsou funkce pro přidání dalšího bloku. Statická funkce `append_extent` se stará o vložení nového extentu do stromu a jeho případné štěpení, pokud je cílový listový uzel plný. Veřejná funkce `append_block` slouží přímo pro připojení nového bloku. Nejdříve provede výpočet logického čísla nového bloku a poté pro něj zkusí vyhledat extent (pokud se nepodaří, volá funkci pro přidání extentu). Pokud je možno rozšířit stávající extent a pokračovat v souvislé řadě bloků, provede tuto operaci s tím, že se pokusí alokovat konkrétní fyzický blok. Pokud nelze daný blok alokovat, případně je extent plný, vkládá se nový extent a blok se alokuje standardním algoritmem.

`libext4_filesystem.c` implementuje obecné operace týkající se práce se souborovým systémem. Všechny funkce mají v názvu prefix `ext4_filesystem_`. Dále v tomto odstavci budou uvedeny názvy bez tohoto prefixu. Funkce pro inicializaci a ukončení (`init`, `fini`) nepotřebují žádný speciální komentář. Dále jsou zde funkce pro kontrolu při inicializaci (`check_sanity` a `check_features`). Ověřuje se superblok z hlediska konzistence a také se kontrolují vlastnosti připojovaného oddílu, zda jsou kompatibilní s ovladačem. Podle této poslední kontroly se rozhodne, zda je možné připojit oddíl s plným přístupem, jen pro čtení a nebo vůbec.

Pak jsou zde implementovány dvě dvojice funkcí pro získání a vrácení reference na skupinu bloků (`get_block_group_ref`, `put_block_group_ref`) a reference na i-node (`get_inode_ref`, `put_inode_ref`). Získávací funkce slouží pro načtení příslušné datové struktury z příslušného datového bloku a do obalující datové struktury navíc uloží i ukazatel na datový blok. Vracející funkce pak ověřují, zda je nastaven tzv. „dirty“ bit, podle něhož se stejně nastavuje tato hodnota i k datovému bloku. Po ověření pak zajišťují vrácení reference na datový blok, což v případě potřeby způsobí zápis bloku na disk.

Vzhledem k úrovni abstrakce je zde dvojice funkcí na alokaci a uvolnění i-nodu (`alloc_inode`, `free_inode`). Alokační funkce provede samotnou alokaci i-nodu a jeho následnou inicializaci. Uvolňovací funkce pak zařídí uvolnění všech datových bloků a nakonec samotné uvolnění i-nodu.

Funkce `truncate_inode` podporuje pouze „zkracování“ souboru, tedy jen odebrání bloků z konce. Uvolňování bloků se liší pro soubory s extenty a s tradiční přímou a nepřímou adresací. Podle toho se volá příslušná funkce, která zajistí samotné uvolnění.

Další skupina funkcí slouží k operacím s datovými bloky i-nodu. Dvě související funkce (`get_inode_data_block_index`, `set_inode_data_block_index`) slouží pro načtení a zápis čísla fyzického bloku pro zadané číslo logického bloku. Načtení a zápis čísla fyzického bloku pro zadané číslo logického bloku není triviální operací, protože je často třeba procházet nepřímé bloky různé úrovně. Třetí funkce z této skupiny (`release_inode_block`) pak navíc slouží pro uvolnění konkrétního logického bloku z i-nodu, pokud se nepoužívají extenty. Pro připojení dalšího datového bloku slouží funkce `append_inode_block`.

Nakonec jsou implementovány funkce pro práci s uvolněnými i-nody (`add_orphan`, `delete_orphan`). Novější verze si drží interní seznam uvolněných i-nodů.

**libext4\_hash.c** je soubor s implementací hashovacích algoritmů. Všechny funkce jsou převzaté z linuxového jádra a byly upraveny pouze tak, aby datové typy odpovídaly těm z operačního systému HelenOS. Zveřejněná je pouze funkce pro výpočet hash hodnoty textového řetězce (`ext4_hash_string`) a používá se pouze při hashování jména adresářové položky.

**libext4\_ialloc.c** obsahuje implementaci alokátoru i-nodů. Funkce mají v názvu prefix `ext4_ialloc_`. Stejně jako blokový alokátor zveřejňuje pouze funkci pro alokaci `alloc_inode` a uvolnění i-nodu `free_inode`. Ostatní funkce jsou pomocné a slouží na přepočítávání čísel i-nodu z relativních (v rámci skupiny) na absolutní a naopak a také na výpočet čísla skupiny podle čísla i-nodu. Oproti linuxovému ovladači je zde použit velmi primitivní a méně efektivní alokační algoritmus. Důvodem je API ze strany knihovny `libfs`, které neposkytuje ovladačí informace, které potřebuje algoritmus implementovaný v Linuxu. Bylo nutné rozhodnout se, zda bude implementován jednodušší algoritmus, který by vyhovoval API, nebo zda by se dala alokace přizpůsobit a odložit na pozdější okamžik, až budou dostupné všechny informace. Nakonec bylo rozhodnuto ve prospěch jednoduššího algoritmu, protože tento přístup je čistší a chování algoritmu předvídatelnější.

**libext4\_inode.c** implementuje funkce pro práci s položkami datové struktury i-nodu. Všechny funkce mají v názvu prefix `ext4_inode_`. Kromě funkcí pro přímé čtení a zápis jednotlivých položek jsou zde navíc pomocné funkce, které slouží pro ulehčení práce programátora. Například se jedná o ověření typu i-nodu (`is_type`) nebo práci s flagy (`has_flag`, `clear_flag`, `set_flag`). Pro ověření možnosti zkrácení souboru se používá funkce `can_truncate` obalující volání testující příslušné flagy.

**libext4\_superblock.c** implementuje funkce týkající se datové struktury superbloku. Všechny funkce mají v názvu prefix `ext4_superblock_`. Kromě funkcí pro práci s datovými položkami datové struktury superbloku, se zde nalézají několik pomocných funkcí na kontrolu flagů (`has_flag`) a jednotlivých skupin vlastností (`has_feature_*`). Dvojice funkcí se stará o načtení superbloku z disku do paměti (`read_direct`) a zápis z paměti na disk (`write_direct`). Dále je zde implementována funkce pro kontrolu konzistence dat v superbloku (`check_sanity`). Navíc se zde nalézají implementace jednoduchých funkcí, které slouží pro výpočet informací, které nejsou přímo v superbloku zapsány, ale lze je jednoduše vypočítat. Konkrétně se jedná o počet skupin bloků (`get_block_group_count`), počet datových

bloků ve skupině bloků (`get_blocks_in_group`) a počet i-nodů ve skupině bloků (`get_inodes_in_group`).

## 5.3 Další programy

Především pro potřeby testování byl vytvořen jednoduchý program `testwrit`, který dostane jako parametr počet iterací a soubor. Do souboru zapíše (připojí nakonec) 1024 bajtů. Zápis se provede tolikrát, kolik je uvedeno ve druhém parametru příkazu. Program slouží pro velmi jednoduché zvětšování souboru, což se velmi hodilo při testování zápisových a alokačních funkcí. Program se nachází v adresáři `uspace/app/testwrit`

## 5.4 Testování

Dílčí testy korektnosti probíhaly v průběhu celého vývoje. V první fázi (před zahájením implementace zápisových operací) byla testována čtecí část, zda nedošlo k zanesení chyb při znovupoužití zdrojových kódů ovladače `ext2`. Bylo testováno čtení souborů i adresářů s předem připraveným obsahem. Tyto testy probíhaly na virtuálním obrazu pevného disku a posléze i na fyzickém pevném disku s reálnými daty.

V dalších fázích se testovalo jednak během vývoje daného funkčního celku a poté po jeho ukončení, zda nedošlo k narušení dříve implementovaných funkcionalit. Vzhledem k povaze vyvíjeného ovladače probíhalo testování „ručně“, protože po každé provedené operaci se prováděla kontrola pomocí linuxové utility `fsck.ext4`. Stav disku byl také neustále sledován pomocí utility `dumpe2fs`, která vypisuje stav superbloku a všech skupin bloků. S pomocí těchto dvou nástrojů se podařilo odhalit mnoho chyb, které nezanechaly na první pohled stopu.

Navíc bylo při analýze většiny chyb nutné použít ladící výpisy a mnoho volání makra `assert`. Tím se povedlo poměrně přesně lokalizovat problematiku místo ve zdrojovém kódu. Kvůli množství volaných operací nebylo možné používat ladící výpisy ve velkém rozsahu. Při větším počtu ladících hlášek nebylo možné dohledat např. konkrétní chybné číslo bloku apod.

Průběžně byl testován zápis do souboru, vytváření a mazání souborů v adresářích, mazání celých adresářů atd. Další testy byly zaměřeny na vyalokování všech volných bloků/inodů, aby se otestovala nemožnost alokovat další bloky v případě plného disku apod. U adresářů se testovala varianta bez indexu i s ním. Obraz disku

byl pro testování vždy ve stejném výchozím stavu, aby se předešlo zanesení chyb z předchozích testů. Po prvním testování byl odhalen problém s právy k souborům a adresářům. Při ověřování zápisu na disku připojeném do hostitelského systému s Linuxem nebylo možné ani číst soubory. Chyběla totiž všekerá přístupová práva, která HelenOS nepoužívá. Z tohoto důvodu bylo do kódu inicializace nového i-nodu doplněno přidání implicitních práv (**rw** pro soubory a **rwX** pro adresáře) pro všechny 3 skupiny (vlastník, skupina, ostatní).

Testy odhalily spoustu drobnějších i větších chyb, které byly postupně opravovány. Nyní si uvedeme několik málo příkladů:

1. Klasická chyba (+1) při práci s indexem v poli způsobila podivné chování binárního vyhledávání v rámci uzlu extentového stromu. Následky byly poměrně zajímavé, protože chyba se vyskytovala pouze při přístupu k určitému prvku pole. Jiné prvky se načítaly korektně. Chyba způsobovala vrácení chybného čísla bloku, což při zápisových operacích znamenalo, že se přepisovaly důležité diskové struktury a analýza chyby zabrala několik dní. Oprava poté trvala méně než 2 minuty.
2. Občas bylo při implementaci chybně rozhodnuto, zda se datový bloku má načítat přímo z disku nebo zda to není nutné. Pokud tedy nebyl blok načten a přesto se z něj před zápisem četlo, dostávaly volající funkce nesmyslná data, což většinou skončilo pádem ovladače kvůli nelegálnímu přístupu k paměti. Tato chyba se podařila objevit relativně rychle, protože nebylo obtížné zjistit, odkud se nesmyslná data berou.
3. S diskovými bloky slouvisela i další chyba. V jednom případě se korektně neuvolňoval datový blok po skončení zápisu (chybělo pouze volání uvolnění bloku). Díky povaze knihovny pro práci s bloky nedocházelo k žádnému ubírání operační paměti, ale pouze naskakoval čítač referencí při každém dalším načtení bloku, což znemožnilo zápis dat na disk, protože ten se prováděl až tehdy, když čítač klesnul na nulu. Konkrétní případ se stal u bloku, v němž byla uložena bitová mapa i-nodů a při alokaci se neaktualizovala.

Při testování byla odhalena velká slabina celého ovladače, a to rychlost. Někdy šlo o nepodstatné zpomalení způsobené prací s virtuálním diskem, jinde však o vážné zpomalení, které například u extentů znemožnilo dokončení celého testovacího procesu. Zmíněný problém u extentů nebo u adresářového indexu byl částečně odstraněn optimalizací algoritmu, ale i tak rychlost ovladače pokulhává. Testy probíhaly i na

serveru s vyhrazeným pevným diskem, který nebyl zatížen systémovými daty a I/O operace byly tedy pouze v režii systému HelenOS spuštěného v prostředí QEMU.

## 5.5 Provázanost s vývojem HelenOS

Ovladač souborového systému ext4 je součástí rozsáhlého univerzitního projektu. Pro vývoj ovladače je zřízena samostatná vývojová větev (viz příloha A). Po dokončení ovladače může být kód bez problému začleněn do hlavní vývojové větve (mainline), pokud vývojáři tuto implementaci přijmou. Z důvodu možného budoucího uvedení kódu do hlavní vývojové větve byly po celou dobu vývoje pravidelně začleňovány všechny změny z hlavní vývojové větve do vývojové větve ovladače, čímž se případné sloučení obou větví v budoucnu velmi usnadní.

# 6. Závěr

## 6.1 Ostatní implementace

Originální implementace se nachází v jádře Linuxu. Narozdíl od mikrojádra v systému HelenOS je Linux monolitické jádro. Celý kód ovladače tedy běží v privilegovaném režimu. Případná chyba by měla podstatný vliv na běh celého operačního systému. S trochou nadsázky se dá konstatovat, že ovladač v Linuxovém jádře, resp. jeho zdrojové kódy poskytují specifikaci datových struktur a algoritmů. Žádná oficiální textová dokumentace k linuxovému ovladači není dostupná. Tento ovladač je originální implementací a podporuje všechny vlastnosti, které souborový systém ext4 má. Ostatní implementace většinou podporují nějakou podmnožinu těchto vlastností.

Pro ostatní operační systémy unixového typu ovladače existují a většinou jsou portovány z Linuxu. Pro operační systémy z rodiny Windows od firmy Microsoft existuje například projekt Ext2Fsd, který navzdory svému názvu podporuje i vlastnosti z ext3 a ext4. Oproti implementovanému ovladači pro HelenOS podporuje například vlastnost pro práci s flexibilními skupinami bloků a žurnálování. Opačná situace je u extentů. Ext2Fsd podporuje pouze čtení a zápis beze změny velikosti souboru. Ovladač pro HelenOS má naproti tomu plnou podporu práce s extenty. Žurnálování je podporováno v Ext2Fsd pouze částečně, v HelenOS prozatím vůbec.

## 6.2 Zhodnocení

Implementovaný ovladač v operačním systému HelenOS není v žádném případě kompletním ekvivalentem linuxového ovladače. Největším nedostatkem je absence žurnálování, které by zvýšilo bezpečnost práce se souborovým systémem. Žurnálování jako takové však nemá vliv na diskové struktury a je tedy možné se souborovým systémem plnohodnotně pracovat i bez něj. Vynecháno bylo z důvodu přílišné časové náročnosti implementace, což je opět důsledkem chybějící dokumentace.

Ovladač je schopen pracovat s diskovými oddíly, které byly vytvořeny v jiném operačním systému, bez poškození diskových struktur. Formátování vlastního diskového oddílu není v současné době možné, ale v budoucnu bude určitě vhodné patřičnou utilitu vytvořit. Díky knihovně části, lze k implementaci použít již jednou naprogramovanou práci s diskovými strukturami. Oproti původnímu návrhu nebyly z časových důvodů implementovány flexibilní skupiny bloků. V případě bu-

doucí implementace by nemělo být příliš obtížné tuto vlastnost do kódu doplnit díky důslednému používání načítacích funkcí. Části kódu mimo práci se skupinami bloků by tím neměly být dotčeny.

Testování implementace neodhalilo žádné destruktivní chování ovladače. V budoucnu by mohlo být vhodné vytvořit nezávislý testovací nástroj použitelný pro libovolný ovladač souborového systému, který by prověřil korektnost všech operací z pohledu klienta (tj. volání přes `libfs` a `VFS`).

Velkou slabinou ovladače je rychlost, což je dáno použitím některých ne zcela optimálních algoritmů a také způsobem práce se souborovým systémem ze strany `VFS` serveru. Teoreticky by mohla pomoci implementace vlastní blokové cache, která by postihovala například bloky s bitovými mapami apod.

Vzhledem k povaze operačního systému `HelenOS` je implementace nového ovladače přínosná, protože rozšiřuje možnosti připojit diskové oddíly s `ext4` (a předchozí generace) pro čtení i zápis. Uživatelé tedy mohou například pracovat se správně naformátovaným oddílem po nastartování `HelenOSu` z `CD`.

Cílem práce bylo vytvořit prototyp ovladače, který umožní uživatelům připojit diskový oddíl se souborovým systémem `ext4` a provádět na něm běžné operace. Tento požadavek současná implementace splňuje.

Dalším nepřímým cílem této práce bylo rozumné zdokumentování souborového systému `ext4` a jeho předchozích generací, čehož se týká kapitola 3. Po prostudování této kapitoly by měl čtenář porozumět principu souborového systému `ext4` a jeho předchůdců, seznámit se s jeho datovými strukturami a se strukturou a koncepcí souborového systému jako celku.

Práce splňuje zadání a cíle stanovené v úvodu. Chybějící vlastnosti v zásadě nebrání rozumnému použití ovladače na práci se souborovým systémem.

## 6.3 Rozšiřitelnost

V kódu ovladače je stále mnoho prostoru pro vylepšování a rozšiřování. V budoucnu bude vhodné doplnit podporu žurnálování. Dále je zde prostor pro implementaci pokročilých vlastností z linuxového ovladače, které nemají přímý vliv na diskovou strukturu, ale jsou zajímavé především z hlediska výkonu (odložená alokace, vícebloková alokace, ...).

Další možností, jak na tuto práci navázat, je implementace přidružených programů, označovaných jako `e2fsprogs`. S tímto tématem souvisí možnost vytvoření a naformátování diskového oddílu přímo v operačním systému `HelenOS`. Velmi uži-



tečný by byl i program pro kontrolu diskového oddílu, který by byl schopen ho opravit.

Mnoho částí kódu jistě není úplně nejefektivnější, takže i zde je prostor pro budoucí vylepšení výkonnosti celého ovladače.

V ovladači není implementována ochrana proti vícenásobnému připojení diskového oddílu, což se typicky nestává, ale jedná se o funkcionalitu, která zvyšuje bezpečnost práce se souborovým systémem.

# Literatura

- [1] Poirier D.: The Second Extended File System - Internal Layout  
<http://www.nongnu.org/ext2-doc/ext2.pdf>
- [2] Phillips D.: A Directory Index for Ext2. *Proceedings of the 5th annual Linux Showcase Conference*, 2001
- [3] Tweedie Stephen C.: Journaling the Linux ext2fs Filesystem  
<http://e2fsprogs.sourceforge.net/journal-design.pdf>
- [4] Ext4 Howto (official Kernel Wiki)  
[https://ext4.wiki.kernel.org/index.php/Ext4\\_Howto](https://ext4.wiki.kernel.org/index.php/Ext4_Howto)
- [5] HelenOS 0.2.0 Design Documentation  
<http://www.helenos.org/doc/design.pdf>
- [6] HelenOS Implementation and design of the file system layer  
<http://trac.helenos.org/wiki/FSDesign>
- [7] HelenOS IPC for Dummies  
<http://trac.helenos.org/wiki/IPC>

# Seznam obrázků

3.1	Rozložení diskového oddílu ext2 . . . . .	10
3.2	Rozložení dat a metadat ve skupině bloků . . . . .	11
3.3	Datová struktura superbloku (dynamická revize) . . . . .	12
3.4	Datová struktura deskriptoru skupiny . . . . .	16
3.5	Datová struktura i-nodu . . . . .	18
3.6	Schéma přímých a nepřímých bloků . . . . .	20
3.7	Datová struktura adresářové položky . . . . .	21
3.8	Jednoduchý příklad adresáře, ve kterém se již mazalo . . . . .	22
3.9	Hlavička bloku s rozšířenými atributy . . . . .	23
3.10	Hlavička rozšířeného atributu . . . . .	23
3.11	Rozšíření superbloku v souborovém systému ext3 . . . . .	27
3.12	Rozšíření superbloku v souborovém systému ext4 . . . . .	32
3.13	Rozšíření i-nodu v souborovém systému ext4 . . . . .	33
3.14	Datová struktura hlavičky extentu . . . . .	34
3.15	Struktura datových bloků při použití extentů . . . . .	35

# A. Zdrojové kódy

Vývojový tým operačního systému HelenOS používá pro správu zdrojových kódů distribuovaný systém na správu verzí Bazaar. Hlavní vývojová větev je dostupná z úložiště:

```
bzr://brz.helenos.org/mainline
```

Všechny další aktuální vývojové větve jsou dostupné na serveru Launchpad. Tohoto standardu se drží i tato práce, pro kterou byla zřízena vývojová větev dostupná na:

```
lp:~frantisek-princ/helenos/ext4
```

Pro získání aktuální verze lze použít příkaz:

```
bzr branch lp:~frantisek-princ/helenos/ext4 ./HelenOS
```

Zdrojové soubory lze procházet a prohlížet i prostřednictvím webového prohlížeče na adrese:

```
http://bazaar.launchpad.net/~frantisek-princ/helenos/ext4/
```

Zdrojové soubory stažené z Launchpadu jsou nejaktuálnější. Pokud vývojářský tým systému HelenOS začlení vývojovou větev `ext4` do hlavní vývojové větve (`mainline`), pak už budou zdrojové kódy aktuální pouze v této hlavní větvi. Je samozřejmě možné použít i zdrojové soubory z přiloženého DVD, ale ty jsou aktuální pouze v okamžiku odevzdání této práce.

## B. Kompilace a spuštění

Po získání zdrojových kódů (viz příloha A) ze serveru Launchpad máme na disku adresář `HelenOS` (záleží na pojmenování uživatelem) s aktuálními zdrojovými kódy.

Pokud nebyl HelenOS nikdy v minulosti na daném počítači kompilován, je třeba nejdříve připravit a nainstalovat tzv. *cross-compiler*) čehož se dosáhne spuštěním skriptu `toolchain.sh`. Parametr skriptu je nutné změnit, pokud má být HelenOS kompilován pro jinou architekturu. Skript by měl být spuštěn s právy superuživatele *root*, aby se předešlo problémům s přístupovými právy při instalaci

```
cd HelenOS/tools
./toolchain.sh ia32
```

Spuštěný skript nejdříve provede kontrolu, zda jsou nainstalované všechny požadované programy a knihovny. Poté z internetu stáhne zdrojové soubory kompilátoru a podle zvolené cílové architektury jej zkompiluje a nainstaluje.

Nyní máme připravený kompilátor a můžeme s jeho pomocí zkompilovat samotný systém HelenOS. Příkaz `make` se spouští přímo v adresáři se staženými zdrojovými soubory a předáváme mu jako parametr architekturu, pro kterou se má kompilovat.

```
cd ..
make PROFILE=ia32
```

Pokud kompilace proběhla úspěšně, vznikl v adresáři `HelenOS` soubor `image.iso`, který je obrazem bootovatelného CD. Nyní stačí použít vhodný emulátor a spustit v něm systém HelenOS z CD. Vývojáři systému HelenOS doporučují emulátor QEMU ([www.qemu.org](http://www.qemu.org)), ale HelenOS funguje bez problému i v dalších virtualizačních nástrojích. Emulátor musí podporovat tu samou architekturu, pro kterou byl HelenOS zkompilován.

Následuje příklad spuštění v emulátoru QEMU. Operační systém se nainstaluje z obrazu CD. Zároveň připojíme obraz pevného disku nebo lze takto připojit i reálný pevný disk (`/dev/sda1` apod.).

```
qemu -m 256M -cdrom image.iso -hda ext4.img
```