

Flash alapú fájlrendszer

Készítette: Baráth Áron
Programtervező Informatikus, MSc.
ELTE

Témavezető: Tóth Melinda
egyetemi tanársegéd
ELTE, Informatikai Kar,
Programozási nyelvek és Fordítóprogramok Tanszék



2013.

Absztrakt

Rengeteg olyan beágyazott rendszer van, ahol a végrehajtandó program egy csak olvasható memóriában van eltárolva, és az eszköznek nincs szüksége a működése során keletkező adatok eltárolására. Léteznek olyan beágyazott rendszerek is, ahol szükség van az adatok kellően rugalmas tárolására.

Adatok rugalmas tárolására a fájlrendszer a megfelelő választás, mert az adatokhoz egy szimbolikus néven keresztül lehet hozzáférni. Így az adatok hozzáférésehez csak egy fájlnévre van szükség, a hozzáférés részleteivel a fájlrendszer foglalkozik.

A szokásos merevlemez beágyazott rendszerek esetén nem jó megoldás, mert egyrészt költségessé teszi az előállítást, az eszköz méretét is sokszorosára növeli, valamint a merevlemez fogyasztása nagyságrendekkel nagyobb, mint az eszköz többi részének. Ezért más médiumot kell alkalmazni – és ez a médium a flash.

A modern flash felépítéséből adódó problémák nehézkessé és pazarlóvá teszik a hagyományos fájlrendszereket, hiszen a flash-ben nem lehet problémamentesen felülírni egy meglévő adatot.

Ennek a problémának a kiküszöbölésére kidolgoztam egy tranzakció alapú, flash-re optimalizált fájlrendszert, amely figyelembe veszi a hardveres limiteket. A fájlrendszer több, logikailag jól elhatárolható rétegből áll, melyek fő feladata, hogy elfedjék a hardveres réteget. A legfelső szint már logikai azonosítókkal dolgozik, amely kényelmes felhasználást biztosít.

Tartalomjegyzék

Bevezető	4
1. NAND flash	7
1.1. Felhasználás	8
2. Rétegek	10
2.1. Hash réteg	11
2.2. Lapkezelő réteg	13
2.3. Objektumkezelő réteg	16
3. Algoritmusok	22
3.1. Mount	22
3.2. Tranzakciók	25
3.3. Szemétgyűjtés	27
3.4. Fájl megnyitása	27
4. Összehasonlítás	28
4.1. JFFS2	30
4.2. UBIFS	32
4.3. Yaffs	33
5. Összegzés	35
5.1. Jövőbeli tervek	36

Bevezető

Olyan rendszerek esetén, amelyek a működés során adatokat állítanak elő, és ezeket az adatokat hosszú távon el kell tárolni, szükség van egy olyan absztrakciós rétegre, amellyel az adatokhoz egyszerűen és kényelmesen hozzá lehet férni. Az ilyen feladatokat a fájlrendszerek végzik el, a médiumon tárolt adatokhoz szimbolikus nevekkal lehet hozzáférni. A fájlrendszer lehetővé teszi azt is, hogy a fájlok törölhetőek legyenek. A rugalmasság növelése érdekében a fájlokat könyvtárakba lehet szervezni, így egy helyre gyűjthetők azok a fájlok, amelyek ugyanahhoz a logikai egységhez tartoznak.

A fájlrendszer egy absztrakciós réteg a felhasználó és a tároló médium között, így a felhasználónak nem kell foglalkoznia azzal, hogy az adott médiumot hogyan kell írni és olvasni. Fájlrendszer számos médiumon található – nem csak merevlemezen és SSD-n található fájlrendszerek, hanem a CD- és DVD lemezeken is fájlrendszerbe vannak az adatok szervezve. Más területeken pedig flash-en történik az adatok tárolása.

A flash alapú fájlrendszer nem összetévesztendő azokkal az eszközökkel, melyek ugyan flash alapúak, de az operációs rendszer számára egy közönséges „blokk-eszközként” („*blokk device*”) látszódnak. A „blokk-eszköz” azt jelenti, hogy a médium aránylag kis, általában 512 bájtos blokkokban írható és újraírható. Az újraírhatóság „blokk-eszköz” esetén követelmény, de a flash ilyen kis blokkokban nem törölhető. Ilyen „blokk-eszköz” eszköz például az pendrive vagy az SSD. Ezeken a hardvereken található egy vezérlő chip, amely a kis blokkokat leképezi a flash lapjaira, és újraírhatóságot biztosít. Ezen típusú eszközök hátrá-

nya, hogy a rájuk épülő fájlrendszer nem vesz tudomást arról, hogy a tároló médium könnyen tönkremehet az indokolatlanul sok törléstől és írástól.

Beágyazott rendszerekben nem felejtő memóriának nem a merevlemez alkalmazták, hanem a flash-t. A flash-nek két fő típusa létezik: a NOR és a NAND flash [1]. A NOR flash memóriaként olvasható, de az írása már speciális műveleteket igényel. A NOR flash mérete jellemzően nem túl nagy, így sok adat tárolására nem a megfelelő választás. A NAND flash kapacitása sokkal nagyobb, mint a NOR flash kapacitása, azonban a NAND flash olvasása és írása is speciális műveleteket igényelnek. A flash memóriák hátránya, hogy az adatot nem lehet felülírni, és csak nagy, jellemzően 128 KiB-os blokkokban lehet törölni.

Motiváció.

A Conet Kft. egy saját fejlesztésű, valós idejű operációs rendszert használ. Ennél a cégnél merült fel az az igény, hogy az eszközökön található NAND flash chipen az adatokat egy, külön erre a célra kifejlesztett fájlrendszeren [3] tároljuk.

Tervezéskor figyelembe vettem már meglévő rendszerek felépítését is, ötletet merítettem a bennük használt mechanizmusokból. Ilyen rendszerek például a *yaffs2* [4] és az *ubifs* [12]. Amikor ez a projekt indult, akkor a felsorolt fájlrendszerek még instabilak voltak, valamint betöltési sebességük sem volt megfelelő. Másik problémájuk, hogy Linux-hoz készültek, és nem a szükséges rendszerhez.

Ötletet merítettem még a *Solid-State Drive*-ok („SSD”) [2] felépítéséből is. A fő irányvonala az, hogy logikai (512 bájtos) blokk azonosítókat szolgáltat ki felé, de a valóságban a több NAND flash chip egyiken tárolja el a blokkokat. A logikai azonosítókat egy belső táblázattal tudja fizikai címekké alakítani. Egy SSD fizikai mérete sokkal nagyobb, mint a névleges kapacitása (jellemzően 5-15%-kal nagyobb), hogy a saját táblázatát tárolni tudja, valamint legyen helye mozgatni az adatot a flash-en belül.

A merevlemezeken használt fájlrendszerek (pl. FAT32, NTFS, ext2)

flash alapú tárolás esetén szóba sem jöhettek, mert azokban az írási és olvasási egység egyaránt az 512 bájtos blokk (pontosabban a klaszter, ami általában 8 blokkból áll). Ezekben a fájlrendszerekben használt mechanizmusok többsége sem használható – például a B-fában történő könyvtárszerkezet tárolás, mert az adatszerkezetek karbantartása rengeteg írási műveletet igényel.

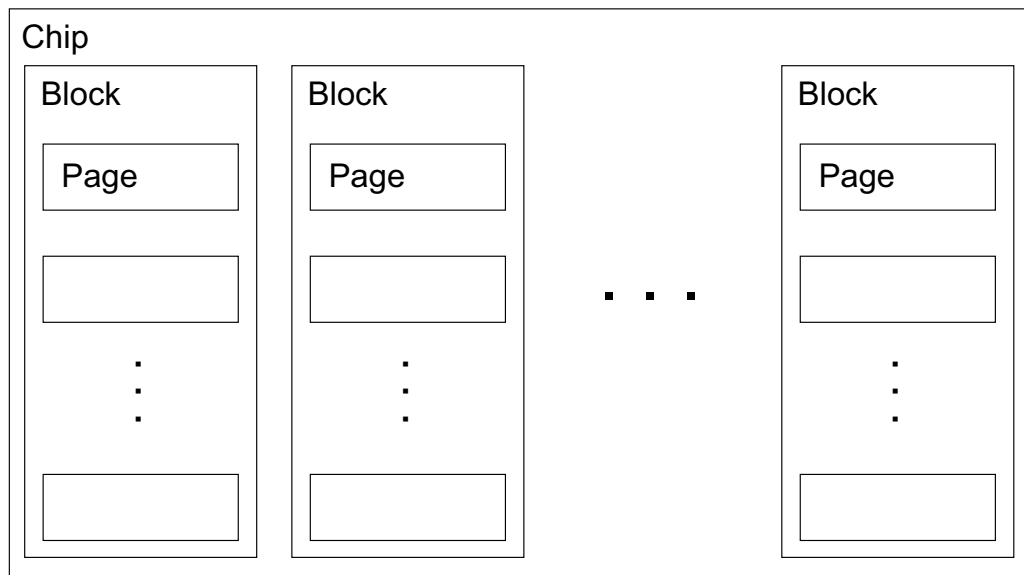
Az SSD egy komplex hardver, ami több NAND flash chipből, és egy vezérlő chipből áll. Beágyazott rendszerek esetében az eszközön nincs hely ennyi chipnek – a használt eszközökön egyetlen NAND flash chip található. A fájlrendszernek a vezérlő chipet kell helyettesítenie és az adattárolást kell megoldania. A már létező, flash-re készült fájlrendszerek ezt a két tevékenységet el tudják látni, de ezek nem a Conet Kft. által használt kernelhez készültek.

A már meglévő flash fájlrendszerek és az SSD ötleteinek felhasználásával már el tudtam kezdeni a saját flash fájlrendszert. A cél az volt, hogy a fájlrendszer működése gyors legyen (gyorsítótárazás), hibatűrő legyen, és a program mérete és memória használata is kicsi legyen.

1. fejezet

NAND flash

A fájlrendszer a NAND flash-en való adattárolásra kell tervezve, így a belső felépítése és adatszerkezetei is ennek megfelelően lettek kialakítva. A NAND flash sajátossága, hogy nem lehet adatot felülírni (csak 1-es bit-et lehet 0-ra állítani, de ez erősen korlátozott), és a törlési egység nagyságrendekkel nagyobb, mint egy merevlemez törlési egysége.



1.1. ábra. A NAND flash felépítése

Lap: („page”) A NAND flash alapvetően lapokból áll; modern flash-eken egy lap 2-8 KiB méretű, és minden 512 bájtához tartozik 16

bájt *spare* (hibás blokkok megjelölésére és adminisztráció tárolására használható). Egy lapot (chip-től függően) kb. 4-szer lehet írni, de az írásokat nem célszerű átlapoltan végezni. Az én megvalósításomban a legkisebb írási egység 512 bájt könyvtárbejegyzések esetén, és teljes lap fájl adat esetén.

Blokk: („*block*”) A NAND flash törlési egysége; általában 64 vagy 128 lap tartozik egy blokkba. A flash alapú fájlrendszerek fő problémája ez a (megszokott 512 bájtos egységhez képest) hatalmas törlési egység. Törlés vagy írás során derül ki az, hogy egy adott blokk meghibásodott (vagy eredetileg is hibás volt). Ekkor a teljes blokkot hibásnak kell tekinteni (az is lehetséges, hogy ténylegesen csak 1 lap romlott el, és a többi normálisan használható; de az is előfordulhat, hogy a meghibásodás az egész blokkot érinti).

Chip: A NAND flash chip; mérettől függően 1024, 2048, 4096 vagy több¹ blokkból áll.

1.1. Felhasználás

A megvalósításban több blokk együtt kerül kezelésre klaszter („*cluster*”) formájában. Azért van szükség egy nagyobb egység bevezetésére, mert az egyes rétegekben használt adatszerkezetek kihasználtsága kicsi lenne.

A fájlrendszer 3 különböző típusú klasztert különböztet meg a chip-en:

Üres: teljesen üres, vagy egy *spare* adattal rendező klaszter.

Hash: a hash réteg által birtokolt klaszterek (*ld. 2.1. fejezet*).

Adat: adatot tároló klaszter (*ld. 2.2. fejezet*).

¹A blokkok száma kettőhatvány.

A szemétygyűjtő zavartalan működése érdekében a fájlrendszer néhány (a jelenlegi megvalósításban 8) klasztert lefoglal, hogy fizikailag ne lehessen teljesen teleírni a flash-t. Ezeket a fenntartott klasztereket csak a szemétygyűjtés során lehet lefoglalni, és minden szemétygyűjtő lefutás után újra rendelkezésre kell álljon a megfelelő mennyiségű üres klaszter.

Mivel egy blokkot csak korlátozott számban lehet törölni, majd újraírni, ezért kiemelt szerepet játszik az is, hogy egy adott klasztert² eddig hányszor írt újra a fájlrendszer. Ha a fájlrendszernek új klaszterre van szüksége, akkor mindig azt választja, amelyik a legkevesebbszer volt újraírva. Ezzel a módszerrel el lehet érni, hogy a flash egységesen kopjon el, vagyis az élettartamot a lehető legjobban kihasználja.

Ezt a törlési számot minden klaszter legelső *spare*-jében tárolja a fájlrendszer egy aláírással és egy CRC ellenőrző összeggel együtt. A fájlrendszer betöltésekor ezeket a *spare*-eket fogja elsőnek beolvasni, és ezekből az információkból tudja felépíteni a szükséges adatszerkezeteket a memóriában.

²Egy adott klaszter mindig ugyanazokat blokkokat tartalmazza, ezért elegendő csak a klaszter törlésének számát tárolni.

2. fejezet

Rétegek

A tervezés során egymásnak ellentmondó elvárásoknak kellett megfelelni. A fő szempont a kicsi adminisztráció volt, vagyis a könyvtárszerkezetet a lehető legkisebb mérettel és a lehető legkevesebb írással kellett megoldani. Ennek az elvárásnak csak a sebesség rovására lehetett megfelelni. Egy másik nagyon fontos szempont a hibatűrés volt. Ezt a tranzakciók bevezetésével biztosítani lehetett – természetesen a szemétgyűjtő eljárásnak is egy tranzakcióban kell futnia. A sebesség is egy fontos szempont volt, de a korlátozott írás és a méret minimalizálása miatt ezen a téren már kompromisszumot kellett kötni – általános fájlhasználat figyelembevételével – de NAND flash írási és olvasási sebessége valamennyire kompenzálta ezt a problémát.

A saját fájlrendszer az *ffs-1.1* kódnevet viseli. Egy régebbi verzió kódneve *ffs-1.0* volt, ami a fejlesztési és tesztelési verzió volt. Ebben a változatban még nem léteztek a tranzakciók, így a hibatűrő képessége alacsony volt. Azonban ennek a verziónak a fő célja a hash- és lapkezelő réteg képességeinek tesztelése volt.

A fájlrendszer tevékenységét három jól elkülöníthető tevékenységre lehet bontani. Ezek a feladatok:

Hash réteg: logikai címek (objektum azonosítók) leképezése fizikai lapokra, szemétgyűjtés (foglalt, de nem használt lapok felszabadítása).

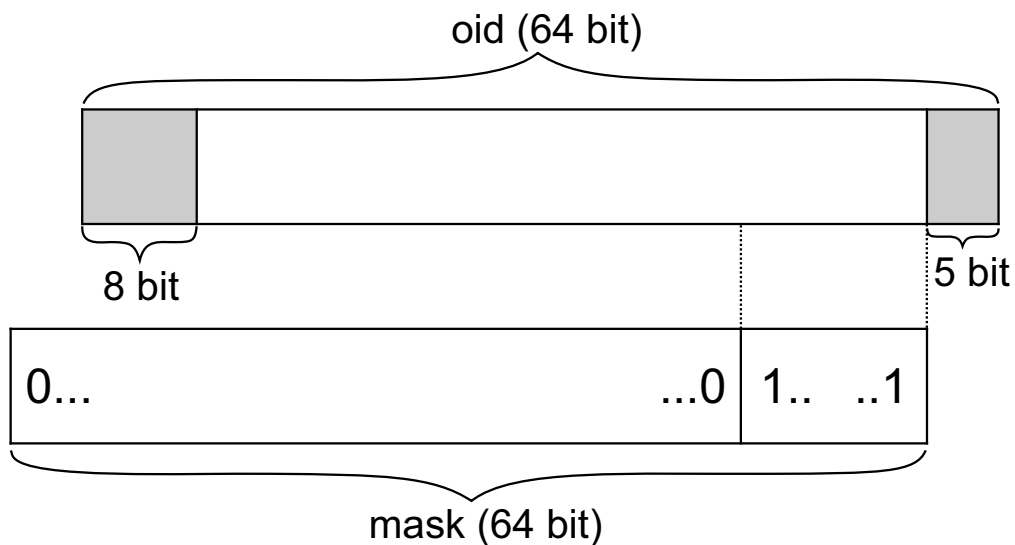
Lapkezelő réteg: logikai címek alapján azonosított lapok beolvasása, gyorsítótárazása, tranzakciókezelés és szemétgyűjtés (lapok mozgása egy másik blokkba a logikai azonosítók megtartásával).

Objektumkezelő réteg: könyvtárbejegyzések (könyvtár és fájl) és ezek műveletei, gyorsítótárazás.

A fájlrendszer objektumokból épül fel, melyeket az objektumkezelő rétegen keresztül ér el. A műveletek során az objektumkezelő réteg műveleteit használja. Az objektumkezelő réteg felelős az objektumok létrehozásáért, melyet a hash rétegre támaszkodva szolgál ki. Az objektumkezelő réteg még megkülönbözteti a könyvtárbejegyzéseket a fájl adattól, mert a különböző típusú objektumokat máshogy kell a flash-re menteni. Az objektumkezelő réteg további feladata az objektumok betöltése a flash-ről – ennek a feladatnak túlnyomó többségét a lapkezelő réteg végzi. A lapkezelő réteg már csak lapokkal dolgozik (nincs különbség könyvtárbejegyzés és fájl adat között), melyek fizikai címmel érhető el. A lapkezelő réteg az új lap lefoglalását a hash réteggel közösen végzi, mert az új objektumhoz tartozó hash bejegyzést is el kell tárolni. A hash réteg fő feladata a logikai objektum azonosítók fizikai lap címmé történő átalakítása, valamint ezen bejegyzések karbantartása (új hozzáadása, egy régi törlése, szemétgyűjtés).

2.1. Hash réteg

Feladata a logikai azonosítók (objektum azonosítók) leképezése fizikai lap sorszámkra. Működésének alapelve a külső hasítás („*external hashing*”) – a nevét is erről kapta. A hasításhoz használt könyvtárat a betöltés során hozza létre (nincs explicit tárolva), és a működés során karbantartja.



2.1. ábra. A hasítófüggvény szemléltetése

A 2.1 ábra szemlélteti az objektum azonosítóhoz tartozó lap sorszám visszakeresését:

1. Az objektum azonosító eltolása 5 bittel jobbra (alsó 5 bit levágása).
2. Maszkolás az aktuális maszkkal.
3. A táblázatot tartalmazó lap sorszám kiolvasása a könyvtárból.
4. A táblázatban logaritmikus kereséssel megkeresi a megfelelő bejegyzést.

A hasítótábla könyvtára mindig kettőhatvány méretű, és a maszk mindig akkora, hogy lefedje a teljes könyvtárat. A könyvtár tartalmazhat azonos bejegyzéseket (például akkor, ha könyvtár növelés történt).

A hasító függvény az alsó 5 bit-et azért vágja le, hogy az objektum azonosítók ne szóródjanak teljesen szét a táblázatok között. A tervezés során ez az érték csak 3 bit volt, de az implementált rendszerben kiderült, hogy ez kevés, és az 5 bit-re növelés a hash réteg gyorsítótárát kétszer jobban kihasználja (a „cache miss” érték több, mint 50%-kal

csökkent – a „cache miss” és a „cache hit” aránya az elvégzett tesztekben 1% alatti érték volt). Az 5 bit-es vágás azt jelenti, hogy minden $2^5 = 32$ egymást követő lap egy táblázatba fog kerülni.

2.1.1. Szemétgyűjtés

A fájlrendszer a háttérben végezhet szemétgyűjtést, melynek során a már nem használt hash lapokat „kitörli” – vagyis ha klasztert töröl, akkor előtte átmásolja azokat a lapokat, amelyek még hasznos információt tárolnak. A szemétgyűjtés a lapkezelő réteg szemétgyűjtésével (ld. 2.2.2. fejezet) együtt fut le, és ha történt változás, akkor a szemétgyűjtés végén egy tranzakciós rekordot helyez el a következő üres lapon.

2.2. Lapkezelő réteg

A lapkezelő réteg fő feladata, hogy egy logikai azonosítókhoz tartozó lapot beolvasson a flash-ről, és ezeknek a lapoknak a gyorsítótárazása. A fájlrendszer betöltése (ld. 3.1. fejezet) során ez a réteg fontos szerepet játszik.

Ha a logikai azonosító szerepel a gyorsítótárban („cache”), akkor visszatér a lapot azonosító csomóponttal. Ha nincs a gyorsítótárban („cache miss”), akkor a logikai azonosítót a hash réteg (ld. 2.1. fejezet) felé továbbítja, és az így megkapott fizikai címen található lapot beolvassa, és eltárolja a gyorsítótárban. A gyorsítótár az LRU („Least Recently Used”) algoritmus alapján működik, vagyis a legrégebben használt lap fog legelőször kikerülni a gyorsítótárból.

A gyorsítótárból kieső lapokat (ha a gyorsítótár betelik, akkor a legrégebben nem érintett lapot eltávolítja) kiírja a flash-re. A lap állapotától függően több lehetőség van:

- Ha a lap már tartalmazott adatot, és csak egy része változott, akkor a módosítást írja ki (ez csak könyvtárbejegyzések esetén al-

kalmazható, mert egy lapon belül ugyanazon bejegyzés több változata is tárolható).

- Ha a lap tartalmazott adatot, de a *spare* rész megváltozott, akkor egy teljesen új fizikai lapra fog kerülni. Végül a hash rétegben a hozzárendelést frissíteni kell.
- Ha a lap még nem tartalmazott adatot, akkor új fizikai lapra kerül. A hash rétegben az új laphoz tartozó hozzárendelést regisztrálni kell.

A klaszterek törlése a lapkezelő réteg szintjén van kezelve, mert a törölt klasztereket csak tranzakció zárás után lehet felszabadítani, valamint a törlés után az első *spare*-be ki kell írni a klaszter állapotát. Az üres klaszterek törlési szám szerint növekvően listájába az újonnan törölt klasztert be kell szúrni.

Ha a működés során új klasztert kell nyitnia, mert betelt az éppen használt klaszter, akkor a szabad klasztereket tartalmazó lista első elemét kiválasztja, és a soron következő szekvenciaszámot elmenti a klaszterben. A szekvenciaszám egy szigorúan monoton növekvő (64 bites) szám, és minden újonnan allokált klaszter szekvenciaszáma nagyobb, mint az eddig kiosztott szekvenciaszámok. A szekvenciaszám feladata, hogy a fájlrendszer betöltése során ki lehessen választani az utolsó használt klasztert (*ld. 3.1. fejezet*).

A lapkezelő réteg segítségével lehet új lapot allokálni – a létrehozáshoz szükség van az objektum azonosítóra. Az allokálás során egy üres csomópont jön létre, amihez hozzárendel egy fizikai lapot. Az objektumkezelő réteg módosítja a lap tartalmát, és megjelöli a lapot „piszkosnak” („dirty”).

A flash-ről beolvasott lapok is létrehozhatnak új csomópontot (ha már be van töltve a gyorsítótárba, akkor nem fog új csomópont létrejönni). Ebben az esetben a lap nem lesz „piszkos”. Módosítás után „piszkossá” változik.

Egy időzítő segítségével az „elég rég óta piszkos” lapokat kiírja a flash-re a lapkezelő réteg. Ha egy lap a flash-ről lett betöltve és módo-

sítva lett, akkor ebben az esetben először egy új lapot kell allokálni. Ha a kiírás sikerült, akkor a hash rétegben létre kell hozni a hozzárendelést.

Egy objektumot tároló lap nem lehet a klaszter legelső, sem a legutolsó lapja. A legelső lap a klaszter típusát tárolja (valamint itt tárolja a szekvenciaszámot, a fájlrendszert azonosító „varázsszámot” és a tárolt adatokból számolt CRC-t), az utolsó lap csak ellenőrző pont (*ld. 2.2.1. fejezet*) lehet.

2.2.1. Ellenőrző pont

Minden adatot tároló klaszterekben legalább egy ellenőrző pont található, feladata a fájlrendszer betöltésének gyorsítása. Az ellenőrző pont a klaszter pillanatnyi állapotát tárolja úgy, hogy egy tömbben szerepel az összes objektum azonosítója.

Az objektum azonosító egy 64 bites szám. Az ellenőrző pontban a felső 8 bitbe a lap típusa kerül. A maradék 56 bit elegendő az összes elérhető objektum azonosító tárolására.

Ha milliszekundumonként 100 objektumot hozna létre a fájlrendszer, akkor kb. 22800 év alatt tudna eljutni az objektum azonosító addig, hogy belelógjon a felső 8 bitbe. Azonban ez olyan sok írással jár, hogy a NAND flash hamarabb megy tönkre, valamint a milliszekundumonkénti 100 objektum létrehozásához kb. 200 MiB/s írási sebességre van szükség – ez a flash írási sebesség beágyazott rendszerekben nem jellemző.

2.2.2. Szemétgyűjtés

A fájlrendszer egyik kritikus összetevője a szemét (fizikailag létező, de logikailag már törölt vagy elavult) lapok eltávolítása. Mivel a NAND flash lapjait egyenként nem lehet törölni, ezért a törlésre váró lapokat csak a szemétgyűjtő tudja ténylegesen eltávolítani.

A lapkezelő szemétgyűjtő feladata, hogy a szabad klaszterek száma

elérje a szükséges limitet. A szemétygyűjtés alapötlete, hogy „régí” (a legkisebb szekvenciaszámú klasztert tekinti a legrégebbinek, mert azt írta ki legrégebben) klasztereket mozgat át új klaszerbe, és csak azokat a lapokat másolja le, amelyek érvényes információt tartalmaznak.

A mozgásra kijelölt klasztert megjelöli (a típusát megváltoztatja), hogy a későbbi allokálások ne ebből a klaszterből történjenek. A klaszter mozgatása során a klaszter minden lapját megvizsgálja, hogy még érvényes adatot tartalmaz-e. Mivel minden lapra (ami objektumot tartalmaz – a hash klaszterek külön vannak kezelve) el van mentve a tárolt objektum azonosítója, ezért a hash réteg segítségével ellenőrizni lehet, hogy az objektum azonosítóhoz rendelt lap sorszám egyezik-e.

Ezen kívül azt is ellenőrizni kell, hogy az adott objektum szerepel-e a szülő objektum gyerekei között (ez az eset például könyvtárbejegyzés törlésekor fordulhat elő). Ha nem szerepel a gyerekek között, akkor a hash rétegben törölni kell ezt az objektum hozzárendelést.

2.3. Objektumkezelő réteg

Az objektumkezelő réteg már egy kellően magas szintű réteg, amely teljes egészében elfedi az alsó rétegek megszorításait. Ez a réteg már csak objektumokkal foglalkozik, és azok azonosítása az objektum azonosítóval történik, amely egy logikai azonosító.

Objektum típusok:

- könyvtár;
- fájl;
- gyökér (a fájlrendszer gyökere);
- fájladat (a fájlok tartalmát is objektumokban tárolja, és minden lapméretű darab egy különálló objektum).

Speciális objektum azonosítók:

0x50 : ellenőrző pontok azonosítója;

0x1000 : a legelső kiosztható azonosító, valamint a gyökér könyvtár-bejegyzés azonosítója;

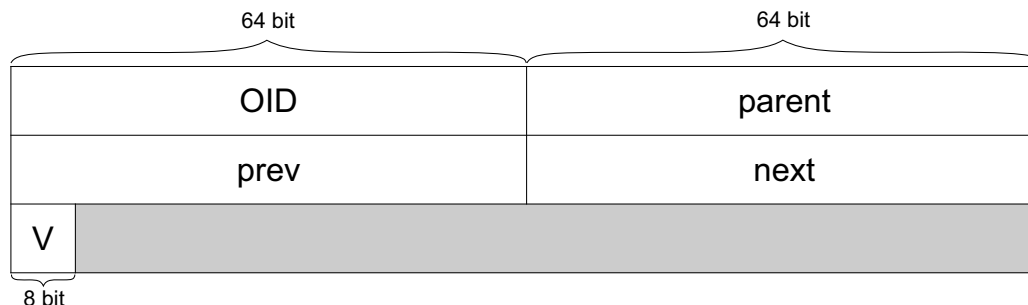
0x0100000000000000 : az első nem használható azonosító – az ellenőrző pontok miatt (*ld. 2.2.1. fejezet*).

Ez a réteg is tartalmaz egy gyorsítótárat, amely – a lapkezelő réteghez hasonlóan – LRU („*Least Recently Used*”) alapú; a gyorsítótár kulcsa az objektum azonosító. Egy időzítő használatával a régen használt lapokat automatikusan perzisztálja a lapkezelő réteg segítségével.

A két gyorsítótárra azért van szükség, mert nem minden objektum mérete egyezik meg a lap méretével (a könyvtár bejegyzések 512 bájt méretűek), és a két gyorsítótár kulcsa is különbözik (az egyik lap címet használ, míg a másik objektum azonosítót). Előfordulhat olyan eset is, amikor egy objektum kimentése új lap lefoglalását vonja maga után, de könyvtár bejegyzések esetén ez nem szükségszerű (egy lap egy könyvtár bejegyzés több változatát is tudja tárolni, de csak az utolsó az érvényes).

2.3.1. Könyvtárszerkezet

A könyvtárbejegyzések láncolt listával vannak egymáshoz láncolva. A linkek objektum azonosítók, melyeket a lap *spare* területén tárol. Ez az egységes felhasználás miatt van, mert a fájlok tartalma is objektum, és nem lenne hatékony, ha különböző adatszerkezet kellene a fájladat tárolására.

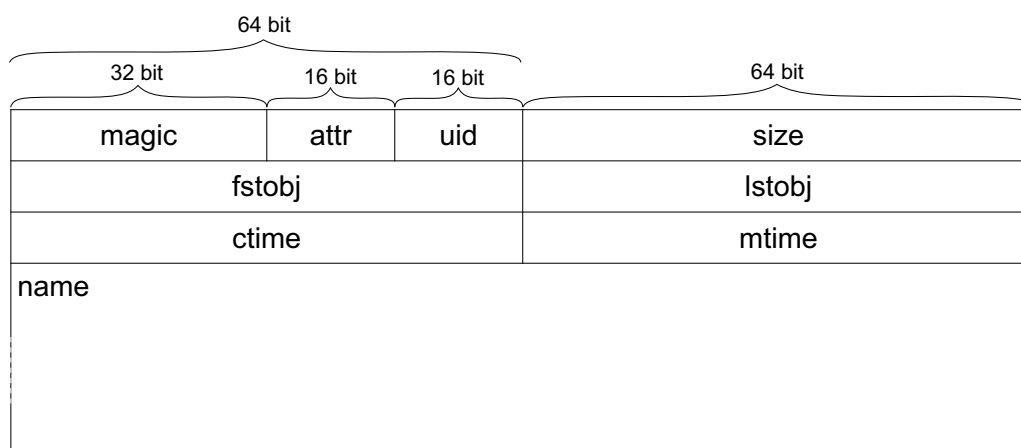


2.2. ábra. Objektum *spare*-je

A könyvtárbejegyzések neve maximum 231 UTF-16 karakter lehet (minden karakter használható a NUL, a / és a \ kivételével), tárolja a létrehozás idejét (ctime), az utolsó módosítás idejét (mtime), valamint az attribútumokat (attr). Jelenleg az uid mezőt nem használja, értéke mindig 0. Ha a könyvtárbejegyzés fájl, akkor a size mezőben a fájl méretét tárolja bájtban, egyébként az értéke 0.

Az objektumoknak lehetnek gyerek objektumai is. A gyerek objektumok ismerik a szomszédjaikat, mert egy láncolt listára vannak felfűzve – a láncolás a spare-ben található prev és next mezőkkel (ld. 2.2. ábra) történik. Ezért elegendő a gyerekek láncának első és utolsó elemét tárolni. A könyvtárbejegyzés (ld. 2.3. ábra) fstobj mezője az első gyerek, a lstobj az utolsó gyerek objektum azonosítóját tárolja.

Ha egy objektum gyerekeinek listáját bővíteni kell, akkor módosítani kell a lstobj értékét. Ehhez újra ki kell írni a könyvtárbejegyzést a flash-re. Azonban az újraírás nem csak egy mező miatt szükséges. Az utolsó módosítás dátuma is megváltozik (mtime), valamint fájlok esetében egy sokkal fontosabb mező is megváltozik: a size.



2.3. ábra. A könyvtárbejegyzés

Ezzel a tárolási módszerrel a bejegyzések hozzáférési ideje $O(n)$, ahol n a könyvtárban található bejegyzések száma. Ennél összetettebb adatszerkezet karbantartása sokkal több flash műveletet igényelne, ami

a flash élettartamát jelentősen lecsökkenti. Ez pedig ellentmond a legfontosabb tervezési szempontnak, miszerint a lehető legkevesebb flash írás kelljen a könyvtárbejegyzések módosításához.

Mivel a fájladat is a könyvtárbejegyzésekhez hasonló objektum (a *spare* felépítése ugyanaz, de a lap adat területén nem könyvtárbejegyzést leíró struktúra van, hanem a fájladat), ezért a fájlpozíció változtatása is lineáris. Azonban a visszafelé való mozgás nem igényel sokkal több műveletet, ellentétben a FAT fájlrendszerrel. A fájlrendszer éles használata során a jellemző művelet a fájl szekvenciális olvasása volt, és a véletlenszerű hozzáférésre megnyitott fájlok száma csekély. Valamint a fájlrendszert nem közvetlenül használják fel a programok, hanem tartozik hozzá egy kliens oldali programkönyvtár, ami egy lokális, 2 KiB méretű olvasási puffert biztosít, így nem kell minden egyes bájtot beolvasásához a fájlrendszert igénybe venni.

2.3.2. Interfész

Ebben a fejezetben ismeretem az objektumkezelő réteg által nyújtott szolgáltatásokat, melyeket a fájlrendszer műveleteinek elkészítéséhez szükségesek. Ez az interfész egy belső interfész, külső szemlélő számára nem láthatóak, és nem használhatóak – kizárólag a fájlrendszer-műveleteken keresztül.

2.3.2.1. Objektum betöltése

Az objektum hozzáférése objektum azonosító segítségével történik. Ha a gyorsítótárban létezik az objektum, akkor azzal tér vissza. Ha nincs a gyorsítótárban, akkor be kell tölteni¹.

Az objektumkezelő réteg a hash réteg segítségével beolvassa az objektum azonosítóhoz tartozó fizikai lap címét, és ehhez a címhez tartozó lapot kéri el a lapkezelő rétegtől. A visszakapott lapból tölti be az objektumot.

¹Előfordulhat olyan eset, hogy az objektum betöltését a lapkezelő réteg gyorsítótára ki tudja szolgálni, de általában a flash-ről kell beolvasni az adatot.

Ha a betöltött objektum könyvtár bejegyzés, akkor kiszámolja a név ellenőrző összegét a keresések gyorsítása végett.

2.3.2.2. Új objektum létrehozása

Az új objektum létrehozása során egy új objektum azonosító kerül kiosztásra, és egy gyorsítótárbeli csomópont lesz lefoglalva. Ha a gyorsítótárban nincs több üres csomópont, akkor a legrégebben hozzáfért objektumot kiírja (ez az írás csak a lapkezelő réteg gyorsítótáráig jut el).

Minden újonnan létrehozott objektumnak meg kell adni a szülőjét. Erre azért van szükség, mert az újonnan létrehozott objektum bekerül a szülő gyerekeinek listájának végére (csak egy objektumnak nincs szülője, és ez a gyökér objektum).

2.3.2.3. Objektum keresése név alapján

Egy könyvtárban név alapján tud keresni, az eredmény egy objektum (implicit az objektum azonosító is). A keresés gyorsítása érdekében minden névhez kiszámol egy ellenőrző összeget, és ezeket hasonlítja össze. Mivel az ellenőrző összeg egy 32 bites egész, ezért ez az összehasonlítás gyorsabb, mint a teljes nevet karakterről karakterre ellenőrizni. Ha az ellenőrző összegek egyeznek, akkor a neveket is össze kell hasonlítani, mert az ellenőrző összegek lehetnek azonosak különböző nevekre.

2.3.2.4. Könyvtár útvonal feldolgozása

Ez a művelet számos fájlrendszer műveletben előfordul, ezért ki lett emelve az objektumkezelő rétegbe. A bemenet egy karakterlánc, ami egy útvonal – a kimenet egy könyvtár objektum, és az utolsó név.

A feldolgozás első lépése a kapott útvonal értelmezése, majd a gyökér könyvtárból kiindulva a nevek megkeresése az aktuális könyvtárban.

A kimenet azért egy pár, mert így ez a funkció több esethez használható: fájl megnyitásához és fájl létrehozásához is használható – létrehozáskor az utolsó nevet nem fogja megtalálni, ami hibát okozna, vagyis nem lehetne alkalmazni.

2.3.2.5. Objektum érvényességének vizsgálata

A szemétgyűjtő eljárásnak van szüksége erre a műveletre. A bemenete egy objektum azonosító, és egy lap cím.

Először meghatározza a hash réteg segítségével, hogy a kapott objektum azonosító melyik fizikai lapon van. Ha a két lap címe nem egyezik, akkor a kapott lapon tárolt adat érvénytelen, mert az objektum egy másik lapon található.

Ha az objektum a gyökér, akkor érvényes – nincs szükség további ellenőrzésre (a hozzárendelés lap címe és a kapott cím egyezik, tehát ez az aktuális gyökér).

Az objektum érvénytelen, ha nem található meg a szülő gyerekeinek a listáján. Ez az ellenőrzés költséges, de csak így lehet eltávolítani a fölösleges objektumokat, és ezáltal helyet felszabadítani a flash-en.

3. fejezet

Algoritmusok

3.1. Mount

A fájlrendszer beágyazott rendszerekhez készült, és fontos az, hogy ne sérüljön meg. Ennek érdekében a működés során az elvégzett műveletek tranzakcionálisak. Ebből következik, hogy ha egy műveletsorozat nem fejeződött be (megszűnt a tápellátás), akkor a félig elmentett tranzakció a betöltés („*mount*”) során nem fog gondot okozni (eldobja a félkész tranzakció módosításait). Így ugyan egy régebbi állapotot tölt be, de az az állapot konzisztens lesz.

A konzisztencia megőrzése érdekében a szemétygyűjtés (és így a klaszter törlés) is tranzakción belül fut. Külön érdekesség, hogy a szemétygyűjtés során nem töröl klasztert, hanem megjelöli töröltként, és miután véget ért a szemétygyűjtés, és a tranzakció is véget ért (a tranzakciós rekord ki lett írva), akkor a törlésre kijelölt klasztereket már törölheti – ha a törlési fázisban szűnik meg a tápellátás, akkor a betöltés során konzisztens állapotot fog visszakapni. Ha a szemétygyűjtés során nem megjelölné a klasztereket, hanem törölné, akkor előfordulhat olyan eset (áramkimaradás esetén), hogy a fájlrendszer egy régebbi állapotot töltene be, de nem tudja, mert az ehhez szükséges klaszterek már üresek.

A betöltő („*mount*”) mechanizmus feladata a legutolsó elmentett

tranzakció megkeresése, és annak használata. Ezt a rendszerbe épített rengeteg invariáns használata egyszerűbbé teszi. Ilyen invariáns például az, hogy egy újonnan, írásra megkezdett klaszter minden eddiginél nagyobb szekvenciaszámot kap.

A betöltő algoritmus úgy van megtervezve, hogy ha a NAND flash teljesen üres, akkor is működjön, és egy kezdőállapotot hozzon létre. Ez azért fontos, mert így egy új eszköz telepítése után a fájlrendszer automatikusan inicializálódik, és azonnal használhatóvá válik. Ha nem így lenne, akkor külön programmal kellene formatálni a flash-t, ami sokkal kényelmetlenebb és kevésbé rugalmas felhasználást eredményez.

1. Hibás blokkok.

Ha egy blokk hibás, akkor az a klaszter is hibás, amelyik ezt a blokkot tartalmazza. A flash-kezelő driver meg tudja állapítani egy blokkról, hogy hibás vagy sem. A betöltés első fázisa minden egyes partícióbeli blokkról megkérdezi a flash-kezelő device-t, hogy az hibás-e.

2. Klaszter információk.

A hibás klaszterek kizárása után (hibásnak tekint egy klasztert, ha legalább az egyik blokkja hibás) a klaszterek első lapjait és *spare*-jeit beolvassa, és a klasztereket szétválogatja három csoportba:

- üres;
- hash klaszter;
- adat klaszter.

Az üres klaszterek törlési szám szerint vannak rendezve, hogy mindig a legkevésbé kopott klasztert használja fel, a hash és adat klaszterek pedig szekvenciaszám szerint vannak rendezve.

3. Utolsó tranzakció.

A tranzakció zárásakor ugyanazt a tranzakciós rekordot menti el először az adat klaszterbe, majd a hash klaszterbe. Ez az írási sorrend fontos, mert ha a tranzakció zárása nem sikerül, akkor a hash klaszterben nem fog megjelenni a hibás tranzakció rekordja – de ha a hash klaszterben látszódik, akkor az adat klaszterben is, és így be lehet tölteni a fájlrendszert.

Az utolsó érvényes tranzakciós rekordot a legnagyobb szekvenciaszámú hash klaszter utolsó lapjától kezdi el keresni. Ez a keresés egy lineáris keresés, és nincs olyan pehelysúlyú módszer, amivel ezt gyorsítani lehetne, de ez a művelet csak a fájlrendszer betöltésekor fut le. Mivel a tranzakciós rekord a *spare*-ben kerül tárolásra, ezért elég csak a *spare*-t beolvasni (csak a *spare* beolvasása sokkal kevesebb időt igényel, mint az egész lap beolvasása).

Ha a hash klaszterben megtalálta a tranzakciós rekordot, akkor meg kell keresnie az adat klaszterben is. Ezt, a hash klaszterbeli kereséssel analóg módon, a legmagasabb szekvenciaszámú adat klaszter utolsó lapjától kezdi keresni. Ha talált egy tranzakciós rekordot, akkor ellenőrzi a rekordban tárolt tranzakciós azonosítót. Ha az azonosítót egyeznek, akkor megtalálta az utolsó érvényes tranzakciót¹.

Ha nem talált egyezést, akkor keresi a hash klaszterben a következő rekordot, majd újra az adat klaszterben. Ha nem talál egyezést, akkor törli az összes (nem üres) klasztert, és tiszta lappal indul az egész fájlrendszer.

4. Ellenőrzőpont.

Ha van használható tranzakció, akkor megkeresi az utolsó ellenőrzőpontot, és adminisztratív jelleggel beírja a tranzakciós rekordot az ellenőrzőpontba. Erre azért van szükség, hogy ne sérüljön az invariáns, miszerint az ellenőrzőpont folyamatosan van kitöltve.

¹Ha az utolsó érvényes tranzakció nem egyezik meg az utolsó fizikai tranzakciós rekorddal, akkor a hash klasztert át kell mozgatni a fölösleges lapok nélkül, hogy a tárolt táblázatok ne tudja felhasználni.

5. Törlendő klaszterek.

A betöltés során törlésre kijelölt klasztereket ebben a fázisban törli ki. Ez lehet üres is, ha egy olyan állapotot töltött be, ami lezárt tranzakcióval végződött. Tesztjeim során ez volt a jellemző, de éles felhasználásban is inkább a zárt tranzakciós állapot volt a domináns.

6. Hash réteg inicializálása.

Ha nincs használható tranzakció, akkor egy üresre inicializálás történik, egyetlen üres kosárral („*bucket*”-tel).

Ha van használható tranzakció, akkor az utolsó (legnagyobb szekvenciaszámú) hash klaszter memóriabeli állapotát állítja vissza. Ezt az utolsó hash lapból kiolvasott értékekből meg tudja tenni: hány kosár van, ezzel együtt a maszk biteinek számát is.

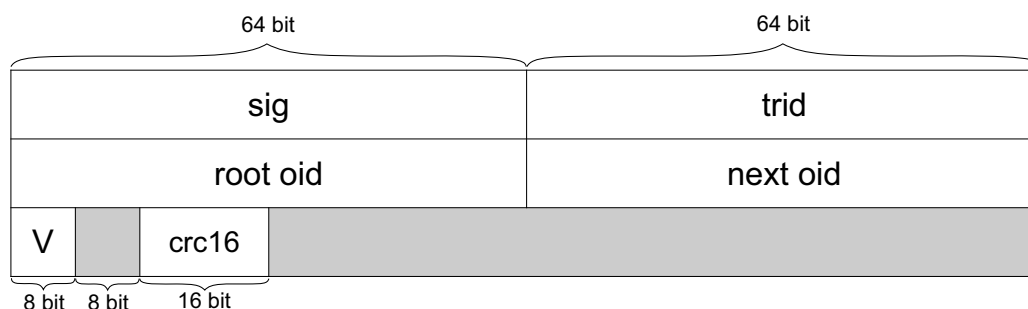
7. Gyökér objektum.

A fájlrendszer csak létező gyökér objektummal tud működni, ezért vagy be kell tölteni a már meglévőt, vagy létre kell hozni egy újat. A meglévő vagy az új gyökér objektum azonosítóját megjegyzi, hogy később a gyökérből tudjon indulni.

A gyökér objektumhoz való gyors hozzáférés érdekében a gyökér objektum hivatkozási számlálója nem 0-tól, hanem 1-től indul, így az objektumkezelő réteg gyorsítótárából sosem fog törlődni.

3.2. Tranzakciók

Annak érdekében, hogy az adatokat konzisztens módon lehessen betölteni például egy áramkimaradás során, tranzakciók segítségével történik a módosítás. Egy tranzakción belül fizikai blokk törlés nem történik, és egy tranzakciót az adat, majd a hash klaszterek soron következő lapjába (az egész lapot lefoglalja, de csak a *spare*-t használja) írt tranzakciós rekord zárja.



3.1. ábra. A tranzakciós rekord (*spare* tartalom)

A tranzakciós rekord (ld. 3.1. ábra) olyan értékeket is tárol, amelyek csak nagyon költséges módon, vagy egyáltalán nem lehet kiszámolni a flash tartalmából.

A *trid* a tranzakció azonosítója, mely egy szigorúan monoton növekvő érték. A következő tranzakció azonosítója még a hibás (nem befejezett) tranzakciók azonosítójától is függ, hogy ne okozzon adminisztrációs hibát egy már meglévő, de hibás tranzakciós azonosító kiosztása. Minden új tranzakció eggyel nagyobb azonosítót kap. A tranzakciós azonosító 64 bites érték, és ez a számláló olyan nagy intervallumot fed le, melynek túlszordulása elképzelhető időn belül nem történik meg.

A tranzakciós rekord tartalmaz két objektum azonosítót is. Egyik a gyökér objektum azonosítója (*root oid*), egy pedig a következő kiosztható objektum azonosító (*next oid*).

A rekord tartalmaz továbbá két rögzített értéket – a *sig*, ami a tranzakciós rekord „*magic number*”-je, valamint a *V*, ami a rekord variánsának típusát jelöli (ez a 'T' karakter). Az összes mezőből pedig egy 16 bites CRC ellenőrző összeget számol ki a fájlrendszer, amit a *crc16* mezőben tárol el. Ezt az értéket a tranzakciós rekord validálásához használja fel, mert nem tudni, hogy az adatok valójában milyen sorrendben kerülnek kiírása a flash-re. Ha az adatok csak egy része íródik ki, akkor a számított és a tárolt CRC nem fog egyezni, és a rekordot érvénytelennek tekinti a fájlrendszer. A *V* és a *crc16* mező között található 8 bit helyköz ahhoz szükséges, hogy a *crc16* mező ne páratlan címen legyen, ugyanis sem az ARM, sem a PowerPC nem tud páratlan címről két bájt

tos értéket kiolvasni².

Az ábrán a szürkével jelölt bájtok nem használtak, értékük 0xFF.

3.3. Szemétgyűjtés

A hash réteg (*ld. 2.1.1. fejezet*) és a lapkezelő réteg (*ld. 2.2.2. fejezet*) szemétgyűjtését foglalja magába az előre meghatározott időnként lefuttatott szemétgyűjtő fázis. Csak akkor van dolga, ha volt változás, így nem foglal le fölöslegesen a számítási kapacitást. A memóriában tárolt állapotból meg tudja állapítani, hogy kell-e futtatni a szemétgyűjtő eljárást.

Ennek az eljárásnak a feladata az, hogy végrehajtsa a lapkezelő szemétgyűjtőjét, majd a hash réteg szemétgyűjtőjét, utána minden adatot írjon ki a flash-re, végül törölje a törlésre kijelölt klasztereket.

3.4. Fájl megnyitása

Egy fájlrendszerbeli fájl megnyitásához csupán két objektum-szintű műveletre van szükség. Az egyik a kapott útvonalat dolgozza fel (*ld. 2.3.2.4. fejezet*), a másik pedig az így kapott könyvtárban megkeresi a megnyitandó fájlt (*ld. 2.3.2.3. fejezet*).

Ha a keresés sikeres, akkor a fájl létezik, és megnyitható. Ezt az információt egy megnyitott fájl leíró rekordban eltárolja, és a fájlmegnyitó művelet visszatérési értéke ennek a rekordnak egy olyan hivatkozása, mely gyorsan validálható.

²Csak bájtossával lehetne olvasni, ami sokkal lassabb, mint páros címről olvasni.

4. fejezet

Összehasonlítás

A saját fájlrendszer felépítését és implementációját (sebesség szempontjából) is összehasonlítottam a meglévő flash fájlrendszerekkel. A sebesség összehasonlítása azonban nem mérvadó, hiszen más kernelen fut a saját fájlrendszer. A saját kernel előnye, hogy a felépítéséből adódóan képes többmagos rendszert a lehető legjobban kihasználni.

A saját fájlrendszer sebességének méréséhez egy ARM CortexA8-as CPU-t használtam (720 MHz), és az eszközön egy 1 Gib-es¹ NAND flash volt.

A mérések alapján a fájlrendszer inicializálásához 0,16-0,22 másodperc szükséges. A változó betöltési idők a betöltéshez használt algoritmusból (*ld. 3.1. fejezet*) következnek, mert a betöltéshez szükséges műveletek száma a NAND flash méretéből kiszámolható: felülről becsülhető a *klaszterek száma + egy klaszterben lévő lapok számának konstansszorosával*. Ha a flash üres az indításkor, akkor 0,8 másodperc alatt inicializálja az egész flash-t és a fájlrendszert is.

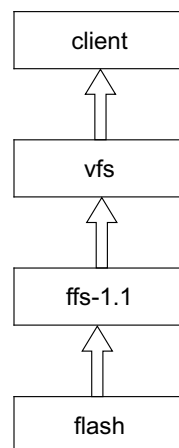
Művelet	ARM	PowerPC
Olvasás	5,1	2,1
Írás	2,3	1,3

4.1. ábra. Műveletek sebessége (MB/s)

¹1 gigabit, vagyis 128 megabájt

A műveletek sebességét egy PowerPC MPC5125 CPU-n (400 MHz) is teszteltem, egy 4 Gib-es NAND flash-sel. Az eszközre ültetett NAND chip egy régi és lassú modell, ezért érzékelhető a nagy különbség az ARM-os eszközhöz képest. A betöltési idő eléri az 1,3 másodpercet is, ami a négyszer nagyobb méret és a fele olyan gyors flash következménye.

Az írási és olvasási műveletek elvégzéséhez az adatnak a kliens oldali pufferezésen át a fájlrendszer pufferéig kell eljutnia. A rendszer felépítéséből adódóan ez több memória másolással jár – ez azonban csak kis részét képezik a műveleteknek. Az ARM CortexA8 memória másolási sebessége 200 MiB/s, vagyis a 60 MiB-os tesztfájl egyszeri lemásolása kb. 0,3 másodpercig tart. A fájl olvasása az elejétől a végéig több, mint 11,5 másodpercig tart, vagyis a másolással eltöltött idő elenyésző a többi költséghez képest.



4.2. ábra. Érintett taszkok

A fájl írása és olvasása közben azonban több taszkváltás is történik. A tesztprogram 2 KiB-os egységekben olvasta az adatot, ami 30720 darab fájlrendszer-műveletet igényel. Egy fájlrendszer-művelet során el kell jutnia a kérésnek a tesztprogramtól a flash device-ig. Ez az út 3 taszkváltást² igényel, de az adatnak vissza is kell jutnia a tesztprogra-

²A Conet Kft-nél használt kernel a programot taszknak hívja – ez pl. a Windows terminológia szerint a processznek felel meg.

míg, vagyis egy fájlrendszer-művelet 6 taszkváltást igényel. A váltogatás kb. 1,5 másodpercet vesz igénybe a 60 MiB-os fájl olvasása során. A 4.2 ábrán az érintett taszkok láthatóak.

A hierarchia legalsó eleme a flash device driver (az ábrán *flash* néven), erre épül a saját flash fájlrendszer (az ábrán *ffs-1.1* néven). A flash fájlrendszert csak a virtuális fájlrendszer (*vfs*) használja, ami egy absztrakciós réteg, hogy ne kelljen az általános műveleteket minden fájlrendszerben implementálni³. A tesztprogram (*client*) pedig a virtuális fájlrendszerre támaszkodva készült el, amely kliensoldali pufferezést is magába foglal. A kliens oldali pufferezés lényege, hogy ha a program bájtanként olvas egy fájlt, akkor gyorsabb legyen az olvasás. Így a fájlrendszernek nem kell 1 bájtos kérésekkel foglalkoznia.

4.1. JFFS2

A JFFS2-t [11] David Woodhouse készítette, amely a 2001-ben kiadott 2.4.10-es Linux kernelben jelent meg először. Elérhető néhány bootloader-ben [16] is.

4.1.1. Felépítés

A JFFS2 [7] az összes „inode”-ot [15] végig olvassa a fájlrendszer betöltése során, ami nagy méretű flash esetén nagyon sok időt vesz igénybe. A saját fájlrendszer a betöltés során minden klaszter első lapját és *spare*-jét kell beolvasa, valamint a legnagyobb szekvenciaszámú klaszter *spare*-jeit.

A JFFS2 tömörítve tárolja az adatot, ami helyes felhasználás esetén nagy előny, mert kevesebb írásra van szükség. Azonban felmerül a kérdés, hogy egy olyan gyors médium esetén, mint a flash, az adat kicsomagolására fordított idő több-e, mint az olvasás. Problémát tud okozni

³Ez a rendszer tartalmaz egy FAT32 implementációt is, melyet szintén a virtuális fájlrendszeren keresztül lehet csak használni. Ez a felépítés hasonlít a Unix/Linux rendszerek egy gyökerű fájlrendszeréhez.

az is, ha nem helyesen használják a fájlrendszert, mert akkor pár bájtot kell tömörítenie, ami negatív tömörítési arányt képes előállítani. A saját fájlrendszer nem tartalmaz fájlrendszer-szintű tömörítést.

A JFFS2 nem tudja egzaktul kiszámolni, hogy mennyi szabad hely van még a fájlrendszeren – ez a tárolt adatok tömörítési arányától függ. A saját fájlrendszer a szabad helyet a fizikailag üres lapok számából számolja ki. Az elavult adatot tároló lapok is foglalt területként kerülnek elszámolásra. Ennek a megoldásnak van egy előnye, hogy a felhasználó biztosan tudhatja, hogy annyi hely biztosan van még a flash-en. Hátránya viszont az, hogy nem valós értéket mutat.

4.1.2. Sebesség

Tesztkörnyezet: ARM926EJ-S 155 MHz CPU, 1 Gib NAND flash.

Telítettség	Eredeti	Gyorsított
0%	2,83 sec	N/A
4%	4,55 sec	3,03 sec
50%	19,5 sec	3,6 sec
80% (kis fájlok)	30 sec	N/A

4.3. ábra. Betöltési („mount”) idők

A felépítésből adódó gyengeséget a mérések is alátámasztják. A fájlrendszeren tárolt fájlok száma drasztikusan csökkenti a betöltési időt. A JFFS2 egy új funkciója lehetővé teszi a betöltés gyorsítását az ú.n. „summary” (összegző) bejegyzés segítségével. Ennek segítségével a betöltéshez szükséges idő töredéke az eredeti változatnak – ez a táblázat (ld. 4.3. ábra) „Gyorsított” [9] oszlopában látható.

Művelet	Sebesség
Olvasás (első)	0,87
Olvasás (ismét)	13,8
Írás (véletlen)	0,2
Írás (nullák)	0,7

4.4. ábra. Műveletek sebessége (MB/s)

A táblázatból (ld. 4.4. ábra) jól látszik, hogy a JFFS2 erősen támaszkodik a gyorsítótárra. Ez a gyakran olvasott adatok esetében növeli a hatékonyságot. Az „Olvasás (első)” oszlopban azok az értékek szerepelnek, amelyek a fájlrendszer betöltése után történt olvasás során adódtak. Az „Olvasás (ismét)” a fájl újraolvasását jelenti – ebben az esetben a fájlrendszer már támaszkodhat a belső gyorsítótáaira.

4.2. UBIFS

Az UBIFS [14] a JFFS2 utódjaként jött létre. Az UBIFS-t a Nokia fejleszti a Szegedi Tudományegyetem segítségével. A fájlrendszer beágyazott rendszerekben történő felhasználásra készült.

Az UBI [13] az „*Unsorted Block Image*” rövidítése. Az UBIFS alacsony szinten a flash blokkjait logikailag átrendezi. A flash blokkok felhasználódásának tárolását („*wear-leveling*”) is ezen szinten kezeli. Az UBI réteg (erre épül az UBIFS) az egész flash-t kezeli, a partíciókat csak logikailag választja el egymástól. Így az egész flash egységesen fog elromlani, vagyis a lehető legjobban kihasználja a flash-t. Amint a flash blokkjai elkezdenek elromlani, az UBIFS tartalma drasztikus sebességgel fog tönkremenni. Az UBI réteg a hibás blokkokat is elrejt a felhasználó elől, így azokkal magasabb szinten nem kell foglalkozni.

Az UBIFS nem tárolja az utolsó hozzáférés dátumát, ami beágyazott rendszerek esetén nem hasznos, valamint értelmetlen a korlátozott írássok számát az utolsó hozzáférés dátumának frissítésére pazarolni.

4.2.1. Sebesség

Tesztkörnyezet: ARM926EJ-S 155 MHz CPU, 1 Gib NAND flash.

A fájlrendszer betöltése kb. 0,5 másodpercet vesz igénybe, függetlenül a rajta tárolt fájlok és könyvtárak méretétől és számától.

Művelet	Sebesség
Olvasás (első)	1,0
Olvasás (ismét)	15
Írás (véletlen)	0,8
Írás (nullák)	5

4.5. ábra. Műveletek sebessége (MB/s)

Az UBIFS gyorsítótár segítségével igyekszik az olvasási műveletek kiszolgálását gyorsítani, ami a 4.5 táblázat alapján nagy hatékonysággal működik.

Az UBIFS esetén külön lehet vizsgálni a különböző adatok írását, mert az UBIFS fájlrendszer-szintű tömörítést alkalmaz. Így több adatot lehet eltárolni a NAND flash-en, és az írások számát is jelentősen csökkenti, ami a chip élettartamát növeli. Az „Írás (véletlen)” eset a véletlenszerű adatok írását tesztelte, ami sok esetben közel áll a valós felhasználáshoz. Az „Írás (nullák)” eset csupánnulla adat írását vizsgálta – ez a típusú adat jól tömöríthető, így az írási sebesség is nagyobb, mert az adat tömörítése kevesebb ideig tart, mint annak flash-re történő kiírása.

4.3. Yaffs

A Yaffs-t [5] Charles Manning (Aleph One) fejleszti, főként beágyazott rendszerekhez.

4.3.1. Felépítés

A Yaffs2 a gyerekobjektumait kétirányú, fejelemes láncolt listában tárolja, ahol a fejelem a szülő objektum. A gyökér objektumnak nincsenek és nem is lehetnek testvérei.

A fájl adatot „Tnode” [6] fában tárolja, ami a gyors adathozzáférést segíti, azonban írás során a fa magasságával arányos számú írási művelet szükséges. Minden 0-tól különböző szinten (a 0-dik szint a legalsó)

8 „Tnode” mutató található csomópontonként. A 0-dik szinten 16 darab NAND azonosító szerepel. A kettő hatványok alkalmazásával gyorsabb a visszakeresés, mert elég a bitmaszkot használni a költséges maradékos osztás helyett.

4.3.2. Sebesség

Tesztkörnyezet: MIPS 403 MHz CPU, 1 Gib NAND flash.

Első	2,25 sec
Üres	0,03 sec
Teli	0,80 sec

4.6. ábra. Betöltési („mount”) idők

A 4.6 táblázatban szereplő idők csak a fájlrendszer belső állapotának a betöltési ideje – ezen felül a hozzárendelés kb. 2,5 másodpercig tart.

Művelet	Yaffs2 (512 bájt)	Yaffs2 (2 KiB)
Olvasás	7,6	8,7
Írás	1,5	3

4.7. ábra. Műveletek sebessége (MB/s)

A Yaffs2 esetén meg lehet választani allokációs egység méretét – a 4.7 táblázat ezért tartalmaz két Yaffs2 mérést. Az egyik méréshez használt fájlrendszer 512 bájtos méretet használt, míg a másik 2048 bájtos méretet. Az allokációs egység méretének növelése pozitív hatással van a sebességre, de az átlagos veszteség növekszik (256 bájról 1024 bájtra növekszik). A gyorsítótárazás miatt az olvasás esetén nem figyelhető meg drasztikus sebesség növekedés.

5. fejezet

Összegzés

Dolgozatomban ismertettem a fájlrendszerek létjogosultságát az adatokat rugalmas tárolásában, valamint a fájlrendszer alá kerülő médiák korlátosságát – beágyazott rendszerekben a választék igen szűk, és az egyes médiumok használhatósága is erősen korlátozott, és korántsem olyan kényelmesen használható, mint a klasszikus merevlemezek esetén.

Beágyazott rendszerekben leggyakrabban a NAND flash-t használják adattárolásra, azonban ehhez olyan fájlrendszerre van szükség, amely nem hagyja figyelmen kívül a flash felépítését, sajátosságait és korlátozásait. Erre az összetett feladatra léteznek megoldások, azonban minden megoldásnak vannak olyan paraméterei, amelyek átlag alattiak. Ennek az a magyarázata, hogy mindegyik megvalósítás más célt tűzött ki.

Célom az volt, hogy olyan flash alapú fájlrendszert tervezzek, amely minimális méretű adminisztrációt használ (ezzel az adminisztrációhoz szükséges írások számát is minimalizálni lehet); hibatűrő legyen (főleg a tápellátás megszűnésre ne legyen érzékeny); valamint a lehető leggyorsabb legyen az írás és az olvasás.

A saját fájlrendszerem által adminisztrációnak lefoglalt terület a teljes flash méretének 4%-a (nagyobb flash esetén lecsökkenhet 1% alatti értékre is), amely közel azonos az UBIFS adminisztrációjának méreté-

vel. A bemutatott fájlrendszerek mindegyikének több időre van szüksége a betöltéshez, mint az általam tervezett fájlrendszernek. A hibák és a fájlrendszer korrumpálódása elkerülése végett a fájlrendszerem tranzakció alapú, és csak a befejezett tranzakciókkal foglalkozik a betöltés után. Az összehasonlításban szereplő flash fájlrendszerek közül csak az UBIFS rendelkezik tranzakciókezeléssel, valamint a Yaffs fájlrendszerhez lehet kiegészítőként csatolni a tranzakciókezelést. A tranzakció alapú tárolásnak vannak előnye és hátránya is – előny, hogy csak teljesen ép adatokkal kell foglalkoznia a fájlrendszernek; hátrány, hogy adatvesztés történhet.

A saját fájlrendszer betöltési ideje a többihez képes kisebb, ami a rendszer indulását gyorsítja, valamint a fájlrendszer írási sebessége is jobb az ismertett fájlrendszerek átlagos írási sebességénél. A megtervezett fájlrendszer implementálásra is került egy zártkörű környezetben¹.

A kitűzött célt sikerült megvalósítani – a fájlrendszer használhatóságát alátámasztja az is, hogy éles rendszerben is alkalmazzák.

5.1. Jövőbeli tervek

A fájlrendszer tervezésekor felmerült az a lehetőség illetve igény, hogy ne csak egyetlen flash chipen működjön a rendszer, hanem több chipen egyszerre. Ennek egy lehetséges megvalósítása az lenne, hogy a jelenlegi fájlrendszer lapkezelő- és hash rétegét kiemelni a fájlrendszerből, és egy flash absztrakciós réteget képeznének. Ez az absztrakciós réteg ellátná a flash helyes használatát, a több flash egyszerre történő használatát, és az egyes blokkok kopásának információinak tárolását. Az így keletkező réteg segítségével már csak a fájlrendszert alkotó objektumokkal kellene a fájlrendszernek törődnie.

Azonban a két réteg között éles határt nem lehet húzni, a megfelelő elválasztás meghatározásához további kutatások és tervezések szükségesek.

¹A Conet Kft. által fejlesztett operációs rendszer kerneléhez készült a program, amely jelenleg nincs publikálva.

gesek. Siker esetén egy olyan absztrakciós réteg jönne létre, amely elég általános szinte bármilyen fájlrendszer elkészítéséhez. Természetesen továbbra is fennmarad az adminisztráció tárolásához szükséges írák száma, amely nagyban befolyásolja a flash élettartamát – függetlenül attól, hogy milyen ügyesen végzi a flash kezelő réteg a feladatát.

Irodalomjegyzék

- [1] Flash memória,
http://en.wikipedia.org/wiki/Flash_memory
2013. április 5.
- [2] Solid-state drive,
http://en.wikipedia.org/wiki/Solid-state_drive
2013. április 5.
- [3] Flash file system,
http://en.wikipedia.org/wiki/Flash_file_system
2013. április 5.
- [4] Yaffs,
<http://www.yaffs.net>
2013. április 6.
- [5] Wikipedia: Yaffs,
<http://en.wikipedia.org/wiki/YAFFS>
2013. április 6.
- [6] Charles Manning: How YAFFS Works,
<http://users.actrix.co.nz/manningc/yaffs-docs/HowYaffsWorks.pdf>
2013. április 21.
- [7] JFFS2 on sourceware.org,
<http://sourceware.org/jffs2>
2013. április 6.

- [8] JFFS2 on MTD,
<http://www.linux-mtd.infradead.org/doc/jffs2.html>
2013. április 6.
- [9] Reducing JFFS2 mount time,
<http://www.embedded-linux.co.uk/tutorial/jffs2-summary>
2013. április 17.
- [10] Wikipedia: JFFS2,
<http://en.wikipedia.org/wiki/JFFS2>
2013. április 6.
- [11] ZHANG Yong, QIU Xue-hong: Implementation of JFFS2 File System in Embedded Linux System,
http://en.cnki.com.cn/Article_en/CJFDTOTAL-WJFZ200604048.htm
2013. április 5.
- [12] UBIFS on elinux,
<http://elinux.org/UBIFS>
2013. április 6.
- [13] UBIFS on MTD,
<http://www.linux-mtd.infradead.org/doc/ubifs.html>
2013. április 6.
- [14] Andreas Schierl, Gerhard Schellhorn, Dominik Haneberg, Wolfgang Reif: Abstract Specification of the UBIFS File System for Flash Memory,
http://link.springer.com/chapter/10.1007/978-3-642-05089-3_13
2013. május 4.
- [15] inode,
<http://en.wikipedia.org/wiki/Inode>
2013. április 13.

[16] Bootloader,
<http://en.wikipedia.org/wiki/Bootloader>
2013. április 26.