Kaatumistilanteet POSIX-tiedostojärjestelmissä

Tuomas Tynkkynen

Pro Gradu
HELSINGIN YLIOPISTO
Tietojenkäsittelytieteen laitos

Helsinki, 25. helmikuuta 2019

HELSINGIN YLIOPISTO — HELSINGFORS UNIVERSITET — UNIVERSITY OF HELSINKI

Tiedekunta — Fakultet — Faculty	Laitos — Institution — Department
Matemaattis-luonnontieteellinen	Tietojenkäsittelytieteen laitos

Tekijä — Författare — Author
Tuomas Tynkkynen

Työn nimi — Arbetets titel — Title

Kaatumistilanteet POSIX-tiedostojärjestelmissä

Oppiaine — Läroämne — Subject
Tietojenkäsittelytiede

Työn laji — Arbetets art — Level	Aika — Datum — Month and year	Sivumäärä — Sidoantal — Number of pages
Pro Gradu	25. helmikuuta 2019	60

Tiivistelmä — Referat — Abstract

Hierarkkiset tiedostojärjestelmät ovat tyypillisin tapa, jolla tietoa voidaan säilöä tietokoneen kiintolevylle tai muuntyyppiseen pysyväismuistiin. Sekä käyttöliittymä että ohjelmointirajapinnat tiedostojärjestelmän käsittelyyn kuuluukin nykyisten käyttöjärjestelmien perustoiminnallisuuksin. Unix-tyyppisissä käyttöjärjestelmissä, kuten Linuxissa, tiedostojärjestelmän käyttö tapahtuu POSIX-standardissa määriteltyjen työkalujen ja rajapintojen avulla. POSIXin tarjoamilla komentorivikomennoilla tiedostoja voidaan muun muassa kopioida, poistaa sekä organisoida hakemistoihin, kun taas sovelluksilla puolestaan on käytettävissä C-kielinen rajapinta. Käyttöjärjestelmän rooliin kuuluu toimia abstraktiona laitteistolle, eikä tiedostojärjestelmä ole tästä poikkeus – tiedostojärjestelmärajapintoja voidaan käyttää samalla tavoin riippumatta siitä millaista levyjärjestelmää käytetään. Tämän takia monimutkaisemmatkin tietokantajärjestelmät, kuten esimerkiksi SQL-tietokannat käyttävät nykyään usein tiedostojärjestelmää raakojen levykirjoitusten sijaan.

Tietokonetta käyttäessä tapahtuu toisinaan erinäisiä järjestelmän kaatumistilanteita, eli järjestelmä on käynnistettävä uudelleen esimerkiksi sähkökatkon tai käyttöjärjestelmävirheen takia. Koska kaatumistilanne voi tapahtua samalla hetkellä kun tiedostojärjestelmä on tekemässä levykirjoituksia, herää kysymys, mitä seurauksia tällä on tiedostojärjestelmän sekä sitä käyttävien sovellusten kannalta. Esimerkiksi levyllä säilytetään käyttäjän datan lisäksi tiedostojärjestelmän omia tietorakenteita, joiden konsistenssi saattaa olla vaarantunut. Toisaalta, POSIX sallii tiedostojärjestelmän käyttää erinäisiä levyvälimuisteja suorituskyvyn parantamiseksi, jotka sovelluskehittäjän täytyy ottaa huomioon kaatumisturvallista sovellusta toteuttaessa. Tässä tutkielmassa tarkastellaan tiedostojärjestelmien kaatumistilanteita molemmista näistä näkökulmista. Sovellusten osalta perehdytään POSIX-standardin tiedostojärjestelmärajapintoihin, sekä miten niitä kuuluu käyttää kaatumisturvallista sovellusta toteutettaessa. Tiedostojärjestelmien osalta tarkastellaan viiden eri tiedostojärjestelmän levytietorakenteita, ja sitä miten ne varautuvat kaatumistilanteisiin.

ACM Computing Classification System (CCS):
Software and its engineering → File systems management
Information systems → Database recovery

Avainsanat — Nyckelord — Keywords
tiedostojärjestelmät, kaatumistilanteet, POSIX, UNIX

Säilytyspaikka — Förvaringsställe — Where deposited

Muita tietoja — Övriga uppgifter — Additional information

Sisältö

1 Johdanto

Tiedostojärjestelmät ovat se pääasiallinen tapa jolla sekä käyttäjät että sovellukset voivat käsitellä tietokoneen massamuistia, eli ylipäätänsä tallentaa tietoa siten että se säilyy tietokoneen virran katkaisemisen jälkeenkin. Melkeinpä jokaiselle tietokoneen käyttäjälle tuleekin siis tutuksi käyttöjärjestelmän tarjoamat mahdollisuudet tiedostojen käsittelyyn, kuten tiedostojen tallentamiseen kiintolevylle, niiden poistamiseen tai kopioimiseen esimerkiksi USB-tikulle. Sekä käyttäjän että sovelluskehittäjän näkökulmasta tiedostojärjestelmän sisäinen toiminta on abstraktoitu verrattaen hyvin. Riippumatta siitä onko käyttäjä avaamassa dokumenttia tekstinkäsittelyohjelmalla useiden teratavujen kokoiselta mekaaniselta kiintolevyltä taikka muutaman gigatavun flash-muistitikulta, on käyttäjäkokemus samanlainen ja tekstinkäsittelyohjelma käyttää täsmälleen samoja käyttöjärjestelmän rajapintoja tiedostojen käsittelyyn. Moderneissa käyttöjärjestelmissä on sovellusten käyttämät rajapinnat tiedostojen käsittelyyn eroteltu itse tiedostojärjestelmästä [Pat03]. Tämä mahdollistaa sen, että käyttöjärjestelmä voi tukea useita tiedostojärjestelmiä samaan aikaan, jolloin samaan tietokoneeseen voi olla samaan aikaan liitettynä monentyyppisiä tiedostojärjestelmiä joko eri tallennusmedioilla tai saman levyn eri osioilla. Uusia tiedostojärjestelmiä kehitetäänkin jatkuvasti muun muassa parantamaan suorituskykyä tai tuomaan uusia ominaisuuksia. Esimerkiksi Microsoft on vuosien saatossa vaihtanut Windows-käyttöjärjestelmän oletustiedostojärjestelmän FAT:sta NTFS:ään [Gia98], ja kuten tämän tutkielman myöhemmissä luvuissa nähdään, on Linux-maailmassa ollut reilustikin valinnanvaraa erityyppisissä tiedostojärjestelmissä.

Eräs motivaattori uusien tiedostojärjestelmien kehittämiseen on luotettavuuden parantaminen. Useat aikanaan käytetyt tiedostojärjestelmät, kuten DOS-käyttöjärjestelmän FAT tai alkuperäisen Unixin tiedostojärjestelmä, olivat nykymittapuulla erittäin yksinkertaisia eivätkä antaneet juuri mi-

tään takuuta sille että tiedostojärjestelmän eheys säilyisi *kaatumistilanteissa*. Tyypillisiä tiedostojärjestelmän kaatumistilanteita ovat esimerkiksi koko tietokoneen sammuminen virtakatkoksen takia tai tilanteita jossa järjestelmä pitää käynnistää uudelleen sammuttamatta sitä turvallisesti, esimerkiksi jonkin vakavan käyttöjärjestelmävirheen takia. Kyseisillä tiedostojärjestelmillä kaatumistilannetta seurasikin aina koko tiedostojärjestelmän tarkistus `fsck`-ohjelmalla, joka kävi koko levyn läpi ja yritti parhaansa mukaan palauttaa tiedostojärjestelmän toimivaan tilaan [Twe98]. Fsck-ohjelman suorituksen aikana tiedostojärjestelmä ei ole normaalisti käytettävissä, mikä tiedostojärjestelmän käyttötarkoituksesta riippuen tarkoittikin mahdollista käyttökatkosta koko järjestelmälle. Modernimmat tiedostojärjestelmät, kuten Linuxin `ext3` vuodelta 2001, ovatkin suunniteltu selviämään kaatumistilanteista nopeasti ja turvallisesti, joten tämänkaltaiset pitkät käyttökatkot tiedostojärjestelmätarkistusten takia ovat olleet historiaa jo tovin.

Kokonaisuutena kaatumistilanteista selviytyminen on monitahoinen ongelma. Vaikka siis käytettävä tiedostojärjestelmä vaikuttaakin merkittävästi siihen, mitä seurauksia järjestelmän kaatumistilanteesta johtaa, on sovellusohjelmallakin oma vastuunsa siitä että käyttäjän dataa ei korruptoidu. Tiedostojärjestelmäkutsujen käyttö kaatumisturvallisesti ei välttämättä ole täysin suoraviivaista, vaan sovellusohjelmoijan täytyy usein tehdä erityistoimenpiteitä jotta kokonaisuudesta tulee kaatumisturvallinen. Toisaalta itse tiedostojärjestelmätoteutuksen täytyy huolehtia siitä että tiedostojärjestelmän sisäisten tietorakenteiden eheys säilyy kaatumistilanteessa ja että toteutus pystyy täyttämään tiedostojärjestelmärajapinnan asettamat vaatimukset joihin sovellukset luottavat.

Tämän tutkielman tavoitteena on tarkastella kaatumisturvallisuutta molemmista näkökulmista. Ensimmäisessä osuudessa keskitytään tarkastelemaan sovellusten näkökulmasta: luvussa 2 tarkastellaan Unix-tiedostojärjestelmärajapintojen perusteita ja luvussa 3 kuinka niitä tulisi käyttää kaatumis-

turvallisuuden takaamiseksi. Luvussa 4 tarkastellaan käytännön tutkimusta siitä, miten kaatumistilanteet vaikuttavat eri tiedostojärjestelmillä. Lopuksi luvussa 5 tutkitaan kuinka hyvin olemassa olevat sovellukset ottavat kaatumistilanteet huomioon. Tutkielman toisessa osuudessa keskitytään tarkastelemaan tiedostojärjestelmien sisäisten tietorakenteiden eheyden säilymistä. Luvussa 6 esitellään perinteisen mutta ei kovin kaatumisturvallisen Linux-tiedostojärjestelmän, `ext2`:n, sisäistä rakennetta ja luvussa 7 mitä seurauksia kaatumistilanteilla on `ext2`:lle. Lopuksi esitellään tekniikoita, joita modernit tiedostojärjestelmät käyttävät tiedostojärjestelmän eheyden säilyttämiseen kaatumistilanteissa: kirjaavia tiedostojärjestelmiä luvussa 8, lokipohjaisia tiedostojärjestelmiä luvussa 9, kirjoittaessa kopioivia tiedostojärjestelmiä luvussa 10 ja kirjoitusjärjestyksen hallintaan perustuvia tiedostojärjestelmiä luvussa 11.

2 Unix-tiedostojärjestelmien perusteet

Vaikka tiedostojärjestelmät konseptina ovatkin todennäköisesti tuttuja monille tietokoneenkäyttäjille, on laitettava merkille ettei tiedostojärjestelmälle ole olemassa mitään yhtä oikeaa määritelmää vaan eri käyttöjärjestelmissä ne toimivat hieman eri tavoilla. Koska tässä tutkielmassa tarkastellaan pääosin Unix-tyyppisten käyttöjärjestelmien tiedostojärjestelmiä, tutustutaan ensin niiden periaatteisiin läheisemmin. Nykypäivän Unix-tyyppisten käyttöjärjestelmien tiedostojärjestelmärajapintoja yhdistää pääosin *POSIX (Portable Operating System Interface for uniX)*-standardi [IEE18].

2.1 Unixin ja POSIX-standardin lyhyt historia

Unix-käyttöjärjestelmän kehitys alkoi vuonna 1969 AT&T:n Bell Labs-kehityslaboratoriossa. Ken Thompson ja Dennis Ritchie olivat olleet kehittämässä MULTICS-nimistä interaktiivista usean käyttäjän moniajokäyttöjär-

jestelmää yhdessä General Electricin ja MIT:n yliopiston kanssa. MULTICS-projektin tavoitteet olivat aikaansa nähden kunnianhimoisia ja käyttöjärjestelmän valmistuminen näyttikin venyvän ja venyvän, joten Bell Labs vetäytyi lopulta pois projektista. Thompson ja Ritchie eivät olleet tyytyväisiä tilanteeseen vaan alkoivat kehittää omaa MULTICSin inspiroimaa käyttöjärjestelmää PDP-7-tietokoneelle [Sal94]. Jo samana vuonna Unix oli jo täysin omavarainen, eli Unixin kehitystä pystyttiin jatkamaan Unixilla itsellään. Aluksi Unixia kehitettiin pääasiassa tutkimuskäyttöön mutta myös AT&T:n sisäiseen toimistokäyttöön, kuten tekstidokumenttien käsittelyyn sekä latomiseen [Pat03]. Kuitenkin vuonna 1974 Unix tuotiin akateemisen maailman tietoisuuteen Thompsonin ja Ritchien julkaisemalla artikkelilla "The UNIX time-sharing system" [RT74] jonka seurauksena ulkopuolisetkin kiinnostuivat Unixista.

Bell Labsin emoyhtiöllä oli kuitenkin tuohon aikoihin monopoliasema Yhdysvaltojen puhelinverkkotoimintoihin, joten kilpailuviraston kanssa tehdyn sopimuksen pohjalta sen oli kiellettyä alkaa tekemään liiketoimintaa muilla toimialueilla eikä voinut siten tuotteistaa Unixia. Sen sijaan Unixin lähdekoodia alettiin lisensoida nimellistä kertakorvausta vastaan pääasiassa yliopistoille mutta myös joillekin yrityksille, jotka alkoivat tehdä omia muutoksiaan Unixiin ja jaella tai markkinoida niitä [Pat03]. Ensimmäinen akateemisen maailman tuottama Unix-johdannainen on Berkeleyn yliopiston BSD (Berkeley Software Distribution) josta ensimmäinen versio 1BSD julkaistiin vuonna 1977. 80-luvun aikana tunnettuja markkinoille tuli muun muassa Microsoftin Xenix, Sun Microsystemsin SunOS sekä IBM:n AIX. AT&T:n puhelinverkkotoiminnan monopoliaseman purkaantuminen avasi sille tilaisuuden myydä Unixia kaupallisesti, joka julkaistiin Unix System V-nimellä vuonna 1983.

Markkinoilla oli siis saatavilla useita periaatteessa Unix-pohjaisia käyttöjärjestelmiä, mutta jotka kaikki käytännössä toimivat hieman eri tavalla.

Sovellusohjelmoijan kannalta tämä hankaloitti huomattavasti monella eri Unix-järjestelmällä toimivien ohjelmien toteutusta. Tämän ratkaisemiseksi kehiteltiin POSIX-standardi [IEE18] yhtenäistämään eri Unix-toteutusten tarjoamat sovellusrajapinnat. Ensimmäinen versio standardista julkaistiin vuonna 1988. POSIX-standardi määrittelee muun muassa komentorivityökaluja sekä C-kielisiä rajapintoja muun muassa tiedostojen, säikeiden ja prosessien hallintaan. Käytännössä mitään Linux-jakelua ei virallisesti ole sertifioitu POSIX-yhteensopivaksi [Loc06] mutta käytännössä kehittäjät tähtäävät POSIX-yhteensopivuuteen [BC05].

Tiedostojärjestelmien käsittelyä varten POSIX tarjoaa sekä komentorivikomentoja kuten `ls`, `cp`, `rm` pääasiassa käyttäjien käyttöön että ohjelmointirajapintoja C-ohjelmointikielellä sovellusten toteuttamista varten. Valtaosa näistä ohjelmointirajapinnoista toimivat varsin matalalla tasolla, joista sovellusohjelmoijan täytyy itse koostaa useat vähänkään monimutkaisemmat tiedostojärjestelmäoperaatiot. Erityisesti tiedostojärjestelmän käyttö kaatumisturvallisesti vaatii ohjelmoijalta erityistä kykyä ja ymmärrystä käyttää POSIX-rajapintoja oikein, mitä käsitellään myöhemmin luvussa 3. Tästä syystä esitelläänkin seuraavaksi lyhyesti tärkeimmät POSIXin tarjoamat C-rajapinnat tiedostojen käsittelyyn.

2.2 POSIX-tiedostojärjestelmärajapintojen perusteet

POSIX-tiedostojärjestelmä on *hierarkkinen tiedostojärjestelmä*, eli tavallisten tiedostojen lisäksi tiedostojärjestelmä tukee *hakemistoja*, jotka voivat sisältää mielivaltaisen määrän tiedostoja tai hakemistoja. Tiedostojärjestelmä muodostaa siis puurakenteen, mistä tulee tämä hierarkkisen määritelmä. Kaikilla tiedostoilla ja hakemistoilla on jokin käyttäjän antama nimi, jonka täytyy olla uniikki siinä hakemistossa jossa kyseinen tiedosto tai hakemisto sijaitsee. Mikä tahansa tiedosto tai hakemisto Unix-tiedostojärjestelmässä voidaan siis paikantaa *tiedostopolulla*, eli merkkijonolla jossa `/`-merkillä on eroteltuna

alihakemistojen nimet joita pitkin pitää kulkea tiedoston paikantaakseen. Esimerkiksi tiedostopolku `/home/tuomas/gradu.tex` tarkoittaa että juurihakemistosta `/` paikannetaan alihakemisto `home`, josta löytyy alihakemisto `tuomas`, jonka sisällä on viimein tiedosto `gradu.tex`.

POSIXissa on kaksi erityistä tiedostonimeä, `.` ja `..`, joiden voidaan mieltää löytyvän jokaisesta hakemistosta. Näistä `.` viittaa aina siihen hakemistoon jossa se sijaitsee ja `..` viittaa ylempään hakemistoon hakemistopuussa. Siis esimerkiksi polku `/home/tuomas/.` viittaa samaan hakemistoon kuin polku `/home/tuomas`. Lisäksi `.`-nimeä voi käyttää missä tahansa kohtaa polkua, eli esimerkiksi `/home/./tuomas` tai `/home/./././tuomas/./.`. viittaavat edelleen samaan hakemistoon. Nimi `..` taas viittaa ylempään tasoon hakemistorakenteessa, eli tiedostopolku `/home/tuomas/..` viittaa samaan hakemistoon kuin tiedostopolku `/home`. Juurihakemiston tapauksessa `..`-nimi vie takaisin juurihakemistoon itseensä, eli tiedostopolut `/..` ja `/` toimivat identtisesti.

Nimen lisäksi kaikilla tiedostoilla ja hakemistoilla on jonkin verran metadataa muun muassa käyttöoikeuksille sekä aikaleimoille. Koska useissa yhteyksissä sekä tiedostot että hakemistot toimivat yhtenäisesti, on Unix-tiedostojärjestelmistä puhuttaessa yleisesti tapana sanoa, että Unixissa hakemistotkin ovat tiedostoja. Tällöin käytetään termiä *tavallinen tiedosto* (*regular file*) viittaamaan yleiskielen termiin “tiedosto”.

Jokaiselle tiedostolle on tallennettu sen omistavan käyttäjän tunnus (*user id, UID*) sekä ryhmän tunnus (*group id, GID*). Näiden lisäksi tiedostoilla on kolme käyttöoikeutta: luku-, kirjoitus- sekä suoritusoikeus, jotka tiedoston omistaja voi määritellä erikseen joko tiedoston omistajalle, samaan ryhmään kuuluville käyttäjille sekä ulkopuolisille käyttäjille. Unix-tiedostojärjestelmä myös ylläpitää kolmea aikaleimaa eri tarkoituksiin: milloin tiedoston sisältöä on viimeksi luettu (access timestamp), milloin tiedoston sisältöä on viimeksi muokattu (data modification timestamp) sekä milloin tiedostoa ylipäätään on muokattu (status change timestamp).

Eräs varsin erikoinen ominaisuus, jota ei yleensä esiinny Unix-tiedostojärjestelmien ulkopuolella on *kova linkki* (*hard link*), mikä käytännössä tarkoittaa että tiedostolla (mutta ei hakemistolla) voi olla useita nimiä tiedostojärjestelmässä. Kovan linkin luominen tehdään olemassa olevalle tiedostolle, jonka jälkeen kyseistä tiedostoa voi käsitellä täysin identtisesti sekä alkuperäisellä nimellä että linkin nimellä. Jälkikäteen ei ole edes mahdollista erottaa kumpi tiedoston nimistä oli se, jolla tiedosto oli alunperin luotu. Yhden samaan tiedostoon viittaavan nimen poistaminen ei vaikuta muihin nimiin mitenkään. Kovan linkin luominen hakemistolle ei ole mahdollista useimmissa Unix-tiedostojärjestelmissä, koska tällöin hakemistojen `..`-nimi ei olisi enää yksikäsitteinen. Sanotaankin, että Unix-tiedostojärjestelmässä tiedostojen ja hakemistojen nimi on erillään niiden *inodesta*, joka sisältää kaiken informaation kuten sisällön ja attribuutit. Hakemistot sen sijaan sisältävät vain viitteen nimestä inodeen. Jokaisella inodella on yksilöivä numero, jolla inodeen viitataan. Sovellusohjelmoijan tai käyttäjän kannalta inoden numerolla ei tosin ole muuta käyttöä kuin että sillä voidaan selvittää viittaavatko kaksi eri tiedostonimeä samaan inodeen.

Tavallisten tiedostojen ja hakemistojen lisäksi Unix-tiedostojärjestelmä tukee viittä eri erikoistiedostotyyppiä. Merkittävin näistä on *symbolinen linkki* (*symbolic link*), joka on hakemistopolun muodossa oleva alias johonkin toiseen sijaintiin tiedostojärjestelmässä. Useimmissa yhteyksissä symbolisen linkkin käyttö osana tiedostopolkua ohjautuu linkin kohteeseen. Esimerkiksi jos `/home/admin` on symbolinen linkki polkuun `/root`, johtaisi tiedoston `/home/admin/readme.txt` avaaminen tosiasiassa tiedoston `/root/readme.txt` avaamiseen. Poikkeuksena symbolisen linkin poistaminen tai uudelleennimeäminen kohdistuu linkkiin itseensä, eikä sen kohteeseen. Koska symbolinen linkki sisältää pelkän tiedostopolun merkkijonona, voi symbolisen linkin kohde olla mitä tahansa, mukaanlukien hakemisto tai toinen symbolinen linkki, tai kohteen ei edes tarvitse olla olemassa. Muita tuettuja

erikoistiedostoja ovat prosessien väliseen kommunikaatioon käytetyt *nimetty putki* (*named pipe*) ja *paikallinen pistoke* (*domain socket*), sekä laitetiedostot *merkkilaite* (*character device*) ja *lohkolaite* (*block device*). Erikoistiedoston voi avata ja useissa yhteyksissä käsitellä kuten tavallisiakin tiedostoja, mutta luku- ja kirjoitusoperaatioilla on jokin muu merkitys. Esimerkiksi tulostinta vastaavaan merkkilaitetiedostoon tehty kirjoitus saatetaankin tulostaa paperille. Erikoistiedostojen käyttäminen tähän tarkoitukseen mahdollistaa esimerkiksi käyttöoikeuksien asettamisen laitteille täsmälleen samaan tapaan kuin muillekin tiedostoille.

2.3 Tiedoston avaus ja sulkeminen

Tiedoston avaus tapahtuu `open`-funktiolla:

```
int open(const char *path, int oflag, ...);
```

Parametri `path` kertoo avattavan tiedoston polku merkkijonona ja parametri `oflag` sisältää lippuja jotka määräävät miten tiedosto avataan. Yksi lipuista `O_RDONLY`, `O_WRONLY` ja `O_RDWR` on pakko antaa. Ne määräävät avataanko tiedosto vain lukua, vain kirjoitusta tai sekä lukua että kirjoitusta varten. Muita mahdollisia lippuja on esimerkiksi `O_TRUNC`, joka tyhjentää (*truncate*) tiedoston sisällön. Mikäli lippu `O_CREAT` on annettu, tiedosto luodaan jos sitä ei ollut aikaisemmin olemassa. Tällöin kolmas parametri määrittelee luodun tiedoston oikeudet. Funktio palauttaa onnistuessaan `int`-tyyppisen *tiedostokahvan*, joka viittaa kyseiseen avattuun tiedostoon. Tiedostokahvat ovat prosessikohtaisia, eli yrittämällä lukea samalla tiedostokahvan numeerisella arvolla jossakin toisessa prosessissa tapahtuu jotain odottamatonta. Avattu tiedostokahva suljetaan `close()`-funktiolla, jolle annetaan auki oleva tiedostokahva ainoana parametrina:

```
int close(int fildes);
```

2.4 Tiedoston luku ja kirjoitus

Tiedostosta luku ja tiedostoon kirjoitus tapahtuu funktioilla `read` ja `write`:

```
ssize_t  read(int fildes,       void *buf, size_t nbyte);
ssize_t write(int fildes, const void *buf, size_t nbyte);
```

Ensimmäinen parametri `fildes` on tiedostokahva, jota operoidaan. `buf` on osoitin `nbyte` tavun kokoiseen puskuriin johon luetaan tai josta kirjoitetaan. Onnistuessaan funktiot palauttavat kuinka monta tavua luettiin tai kirjoitettiin. Paluuarvo voi siis olla pienempi kuin mitä ohjelma pyysi lukemaan tai kirjoittamaan, esimerkiksi jos kirjoittaessa levy täyttyi tai luettaessa tiedosto loppui kesken. Unixissa tavallisia tiedostoja voi käsitellä vain tavupohjaisina, eli mikäli sovellus tarvitsee esimerkiksi tietuepohjaista tiedostonkäsittelyä, jää sen toteutus sovelluksen vastuulle.

Jokaisella tavalliseen tiedostoon viittaavalla tiedostokahvalla on sijainti tavuina missä kohtaa tiedostoa ollaan ja josta lukeminen tai kirjoittaminen aloitetaan. Tiedostoa avatessa tiedostokahvan sijainti on normaalisti tiedoston alussa, ja luku- ja kirjoitusoperaatiot onnistuessaan siirtävät sijaintia tiedostossa eteenpäin. Tiedostokahvan sijaintia tiedostossa voi myös siirtää erikseen `lseek()`-funktiolla:

```
off_t lseek(int fildes, off_t offset, int whence);
```

Ensimmäinen parametri `fildes` on tiedostokahva, jota operoidaan. Kolmas parametri `whence` kertoo, miten `offset`-parametri tulkitaan. Sallittuja arvoja ovat vakiot `SEEK_SET`, `SEEK_CUR` ja `SEEK_END`, jotka tarkoittavat vastaavasti että tiedostokahvan sijainniksi asetetaan `offset`, että tiedostokahvan sijaintia siirretään eteenpäin `offset` tavua ja että tiedostokahvan sijainniksi asetetaan `offset` tavua eteenpäin laskien tiedoston lopusta. Onnistuessaan `lseek()`-kutsu palauttaa sijainnin johon siirryttiin. Sijainnin voi siis selvittää siirtymättä tiedostossa kutsulla `lseek(fd, 0, SEEK_CUR)`.

Unix-tiedostojärjestelmissä lukeminen tai kirjoittaminen voidaan tehdä mihin tahansa kohtaa tiedostoa ja niin pienillä tai suurilla datamäärillä kuin on tarpeen. Eli vaikka tiedostojärjestelmä sijaitsisi kiintolevyllä joka pystyy lukemaan tai kirjoittamaan vain 512 tavun sektoreita kerrallaan, voivat sovellukset lukea ja kirjoittaa tavun tarkkuudella. Tämä on yksi tapa jolla tiedostojärjestelmä abstraktoi laitteistoa.

Unix-tiedostojärjestelmät tukevat *harvoja* (*sparse*) tiedostoja, eli tiedostoissa voi olla kohtia jossa niille ei ole varattu levytilaa, mutta niistä lukeminen onnistuu ja palauttaa nollatavuja. Harvojen tiedostojen tekeminen onnistuu `lseek()`-funktion avulla siirtymällä tiedostossa eteenpäin kohtaan johon ei ole kirjoitettu mitään. Esimerkiksi kirjoittamalla ensin 32 kilotavua, siirtymällä eteenpäin 32 kilotavua kutsulla `lseek(fd, 32768, SEEK_CUR)`, ja kirjoittamalla toiset 32 kilotavua pitäisi lopputuloksena olla tiedosto jonka koko on 96 kilotavua mutta vie vain 64 kilotavua levytilaa.

Tiedoston metadatan voi lukea `fstat()`- tai `stat()`-funktioilla:

```
int fstat(int fildes,       struct stat *buf);
int  stat(const char *path, struct stat *buf);
```

Onnistuessaan funktio täyttää `struct stat`-tietueen, josta löytyy muun muassa seuraavia kenttiä:

- Kenttä `mode_t st_mode` kertoo tiedoston tyypin (tavallinen tiedosto, hakemisto, yms.) ja käyttöoikeudet.
- Kenttä `size_t st_size` kertoo tiedoston koon tavuina.
- Kenttä `blkcnt_t st_blocks` kertoo tiedoston viemän tilan 512 tavun yksiköissä.
- Kentät `uid_t st_uid` ja `gid_t st_gid` kertovat tiedoston omistavan käyttäjän sekä ryhmän tunnuksen numeerisen arvon.

- Kentät `st_atim`, `st_mtim` ja `st_ctim` kertovat tiedoston aikaleimat.

Tiedoston, mutta ei hakemistoa, voi poistaa `unlink()`-funktiolla. Hakemiston poistamiseen käytetään funktiota `rmdir()` ja poistettavan hakemiston täytyy olla tyhjä:

```
int unlink(const char *path);
int rmdir(const char *path);
```

Tiedoston voi uudelleennimetä `rename()`-funktiolla:

```
int rename(const char *old, const char *new);
```

Uudelleennimeämisen ei tarvitse tapahtua saman hakemiston sisällä, vaan esimerkiksi `rename("/tmp/file.tmp", "/home/foo/file.txt")` on sallittua, mutta sekä lähde- että kohdepolun täytyy sijaita samalla tiedostojärjestelmällä. Kohdepolku voi viitata myös olemassa olevaan tiedostoon, jolloin kyseinen olemassa oleva tiedosto poistetaan operaation yhteydessä ikään kuin `unlink()`-kutsua olisi käytetty ennen uudelleennimeämistä.

3 POSIX-rajapintojen kaatumisturvallinen käyttö

Tietokoneiden massamuistit ovat tyypillisesti keskusmuistia merkittävästi hitaampia lukea ja kirjoittaa. Tyypillinen tapa parantaa tiedostojärjestelmän suorituskykyä onkin käyttää osaa keskusmuistista välimuistina tiedostojärjestelmälle, ja POSIX-standardi antaakin tiedostojärjestelmätoteutuksille reilusti vapauksia erilaisten välimuistien käyttöön. Tämä kuitenkin hankaloittaa kaatumistilanteita kestävien sovellusten toteuttamista. Tarkastellaankin seuraavaksi POSIXin tarjoamia rajapintoja tiedostojärjestelmän välimuistien hallintaan sekä muita sovellusohjelmoijan kannalta huomioon otettavia seikkoja.

3.1 Tiedostojärjestelmien levyvälimuistit

Erinäiset levyvälimuistit olivat käytössä jo hyvin varhain Unixissa. Jo Ritchien ja Thompsonin alkuperäisestä "The UNIX time-sharing system" [RT74]-artikkelista vuodelta 1974 löytyy lyhyt selostus Unixin *lohkovälimuistista* (*buffer cache*). Lohkovälimuistin toiminta levyltä luettaessa esimerkiksi `read()`-kutsun yhteydessä on yksinkertainen: välimuistista etsitään lohkonumerolla hakemalla lohkon sisältö. Pelkkien levylukujen vieminen välimuistin kautta ei vielä näy sovellusohjelmoijalle mitenkään, mutta lohkovälimuisti joka toimii myös levykirjoituksille tuottaa haasteita. Käytännössä tämä tarkoittaa että `write()`-kutsun yhteydessä tehty tiedostoon kirjoitus ei päädy levylle asti välittömästi, vaan sovellusohjelman kirjoittama data ainoastaan kopioidaan levyvälimuistin puskuriin sekä merkitään kyseinen puskuri kirjoitettavaksi myöhemmin levylle [RT74]. Itse puskureiden kirjoitus levylle tapahtuu taustalla tietyin väliajoin, esimerkiksi Unixin alkuaikoina 30 sekunnin välein [Ros92]. Moderneissa Unix-järjestelmissä levyvälimuistien käyttö on laajennettu koskemaan myös hakemistoja ja tiedostojen metadataa. Levyvälimuistin hallintaan löytyy POSIXissa kaksi keskeistä funktiota [IEE18]:

```
int fdatasync(int fildes);
int fsync(int fildes);
```

Operaatio `fdatasync()` pakottaa parametrina annettua tiedostokahvaa `fildes` vastaavan tiedoston levyvälimuistin kirjoittamisen. Funktiokutsu ei palaa kunnes data on turvallisesti kirjoitettu levylle. `fdatasync()`-kutsu ei vaikuta tiedoston metadataan muuten kuin että kutsun onnistuessa taataan että tiedostoon tähän asti tehdyt kirjoitukset ovat luettavissa kaatumisen jälkeenkin. Käytännössä käyttäjän näkökulmasta tämä takaa vain että tiedoston koko on ajan tasalla. Funktio `fsync()` on vahvempi versio `fdatasync()`:stä, joka lisäksi pakottaa kaiken tiedoston viittaavan inoden metadatan kirjoitettavan levylle. Lisäksi on varmistettava erikseen, että tiedoston inodeen viittaavat

tiedostonimet ovat kirjoitettu pysyvästi levylle. Tämä tapahtuu kutsumalla `fsync()`-funktiota hakemistolle jossa tiedostonimi sijaitsee.

3.2 Esimerkki tiedoston tallentamisesta tekstieditorissa

Tarkastellaan nyt esimerkin avulla, miten POSIXin levyvälimuistin käsittelyfunktioilla voidaan toteuttaa kaatumisturvallisia sovelluksia. Melkeinpä yksinkertaisin mahdollinen sovellus, jossa kaatumisturvallisuus on tärkeää, on perus tekstieditori. Tyypillinen käyttötapaus on, että käyttäjä avaa editorissa tiedoston, muokkaa sitä, ja tallentaa muokatun sisällön samalla nimellä alkuperäisen tiedoston päälle. Luotettavan editorin kuuluu tehdä tiedoston tallennus atomisesti, eli kaatumistilanteen sattuessa joko tiedoston täytyy sisältää joko uusi tai alkuperäinen sisältö. Lisäksi voidaan odottaa tallennusoperaation olevan *pysyvä* (*durable*), eli editorin ilmoittaessa tallennuksen olevan valmis täytyy tiedoston uuden version olla kirjoitettuna levylle. Suoraviivaisin tapa toteuttaa tallennus on kirjoittaa koko tiedosto uudelleen seuraavasti:

```
1 int fd = open("/mnt/usb/file.txt", O_CREAT | O_WRONLY | O_TRUNC);
2 write(fd, uusi_sisalto, strlen(uusi_sisalto));
3 close(fd);
4 printf("Tallennus valmis.\n");
```

Tämä sekvenssi operaatioita ei kuitenkaan ole turvallinen vaikka edes tiedostojärjestelmän kaatumista ei tapahtuisi. Jo pelkkä tekstieditorin kaatuminen `open()`- ja `write()`- kutsujen välissä jättää jäljelleen tyhjän tiedoston. Yksinkertainen yritys välttyä tältä ongelmalta on ensin kirjoittaa tallennettava sisältö väliaikaistiedostoon, ja hyödyntää `rename()`-kutsun kykyä atomisesti

korvata tiedosto toisella:

```
1 int fd = open("/mnt/usb/file.txt.tmp", O_CREAT | O_WRONLY | O_TRUNC);
2 write(fd, uusi_sisalto, strlen(uusi_sisalto));
3 close(fd);
4 rename("/mnt/usb/file.txt.tmp", "/mnt/usb/file.txt");
5 printf("Tallennus valmis.\n");
```

Tämä versio sietää sovelluksen kaatumisen, mutta ei toimi tiedostojärjestelmän kaatumistilanteessa, sillä mikään ei takaa että `write()`-kutsun tekemät kirjoitukset ja tiedoston uudelleennimeäminen `rename()`-kutsulla kirjoitettaisiin levylle edellä mainitussa järjestyksessä. Tiedostojärjestelmän kaatumistilanteessa tyhjä tai osittain kirjoitettu tiedosto on siis edelleen mahdollinen. Myöskään operaation pysyvyydellä ei ole mitään takeita, vaan käyttäjä voi nähdä `Tallennus valmis.`-tulosteen ilman että levylle on tehty yhtäkään kirjoitusta. Sen sijaan täysin kaatumisturvallinen tallennus on toteutettu Gedit-tekstieditorissa seuraavasti [CPADAD13]:

```
1 int fd = open("/mnt/usb/file.txt.tmp", O_CREAT | O_WRONLY | O_TRUNC);
2 write(fd, uusi_sisalto, strlen(uusi_sisalto));
3 fsync(fd);
4 close(fd);
5 rename("/mnt/usb/file.txt.tmp", "/mnt/usb/file.txt");
6 int dfd = open("/mnt/usb", O_DIRECTORY | O_RDONLY);
7 fsync(dfd);
8 close(dfd);
9 printf("Tallennus valmis.\n");
```

`write()`-ja `rename()`-kutsujen väliin on siis lisätty yksi `fsync()`-kutsu riville 3 takaamaan että näiden operaatioiden järjestys ei vaihdu levylle kirjoittaessa. Tämä tekee tiedoston tallennusoperaatiosta atomisen, mutta ei vielä takaa

pysyvyyttä. Pysyvyyteen vaaditaan `fsync()`-operaation suoritus kohdetiedoston sisältävälle hakemistolle rivillä 7, mikä takaa että `rename()`-kutsun tekemän muutokset itse hakemiston sisällölle ovat päätyneet levylle.

4 POSIX-tiedostojärjestelmien kaatumistilanteet käytännössä

Edellisen luvun esimerkistä nähtiin, miten POSIX-standardin sallimat levyvälimuistit hankaloittavat merkittävästi kaatumisturvallisen sovelluksen toteuttamista. Perehdytäänkin seuraavaksi selvittämään levyvälimuistien vaikutusta kaatumistilanteisiin tarkemmin tarkastelemalla tutkimusta siitä, millaisia seurauksia kaatumistilanteilla on tosimaailman tiedostojärjestelmille ja sovelluksille.

4.1 Tutkimus tiedostojärjestelmien käyttäytymisestä kaatumistilanteessa

POSIX-standardi tarjoaa siis tiedostojärjestelmätoteutukselle merkittävät vapaudet esimerkiksi uudelleen järjestellä tai lykätä levykirjoitusten tekemistä suorituskyvyn parantamiseksi, ellei sovellusohjelma ole erikseen pakottanut sitä `fsync()`- tai `fdatasync()`-kutsuja käyttämällä. Itse tiedostojärjestelmätoteutuksen ei tietenkään ole mikään pakko hyödyntää näitä vapauksia, joten herääkin kysymys ilmeneekö tiedostojärjestelmien käyttäytymisessä eroavaisuuksia. Tätä aihetta on tutkittu kokeellisesti “All File Systems Are Not Created Equal: On the Complexity of Crafting Crash-consistent Applications”[PCA+14]-artikkelissa, jossa on vertailtu kuutta eri Linux-tiedostojärjestelmää: `btrfs`, `ext2`, `ext3`, `ext4`, `reiserfs` sekä `xfs`. Lisäksi useimpia tiedostojärjestelmiä testattiin eri tiedostojärjestelmäkohtaisilla optioilla, joten kombinaatioita saatiin yhteensä 16 kappaletta.

Tiedostojärjestelmien vertailua varten määriteltiin *persistenssiominaisuu-*

den (*persistence property*) käsite. Persistenssiominaisuudet ovat käytöksiä, joita ilmenee tiedostojärjestelmän kaatumistilanteessa, ja jotka jakautuvat kohteen kategoriaan: operaatioiden atomisuuteen ja operaatioiden uudelleenjärjestelyyn. Esimerkiksi `write()`-kutsun osalta voidaan tutkia tapahtuuko kirjoitus atomisesti kaatumistilanteen sattuessa ja voiko kirjoitus tapahtua eri järjestyksessä suhteessa johonkin toiseen tiedostojärjestelmäkutsuun. Persistenssiominaisuuksien testaus tehtiin artikkelissa BOB-työkalulla (*Block Order Breaker*). Sen periaatteena on nauhoittaa testattavan tiedostojärjestelmän tekemiä levykirjoituksia lokitiedostoon, josta voidaan jälkikäteen käydä läpi kaikki mahdolliset kaatumistilanteet läpi. Nyt kaatumistilanteista voidaan etsiä vastaesimerkkejä joissa testattava persistenssiominaisuus ei päde. BOB:n voidaan siis vain todeta jonkin persistenssiominaisuuden paikkaansa pitämättömyys, mutta ei toisinpäin.

4.2 Tutkittuja persistenssiominaisuuksia

Seuraavien operaatioiden osalta tutkittiin niiden atomisuutta:

- Tiedoston olemassa olevan sisällön ylikirjoitus erikokoisilla `write()`-kutsuilla.
- Sisällön lisääminen tiedoston loppuun erikokoisilla `write()`-kutsuilla.
- Hakemisto-operaatiot, kuten `link()`, `unlink()` ja `rename()`.

`write()`-kutsun tapauksessa on siis erikseen tutkittu tilanteita jossa tiedoston pituus kasvaa ja tilanteita jossa pelkästään ylikirjoitetaan olemassa olevaa sisältöä. Lisäksi molempia kokeiltiin vaihtelevilla kirjoitettavan datan määrillä, joista mielekkäitä tilanteita ovat levyn sektorin koon (tai vähemmän) verran, yhden tiedostojärjestelmälohkon sekä tätä suuremmat määrät.

Taulukossa 1 on tulokset tutkituista atomisuuteen liittyvistä persistenssiominaisuuksista. `write()`-kutsun osalta ainoa kaikissa tiedostojärjestelmissä atomiseksi havaittu operaatio on yksittäisen levysektorin kokoinen

	ext2	ext2-sync	ext3-writeback	ext3	ext3-datajournal	ext4-writeback	ext4	ext4-nodelalloc	ext4-datajournal	btrfs	xfs	xfs-wsync	reiserfs-nolog	reiserfs-writeback	reiserfs	reiserfs-datajournal
ylikirjoitus	▟	▟	▟	▟	▙	▟	▟	▟	▙	▙	▟	▙	▟	▟	▟	▙
loppuun lisäys	×	▟	×	▙	▙	×	▙	▙	▙	▙	▙	▙	▟	×	▟	▙
hakemisto-operaatiot	×	×	•	•	•	•	•	•	•	•	•	•	×	•	•	•

Taulukko 1: Atomisuutta koskevat persistenssiominaisuudet eri tiedostojärjestelmillä [PCA+14, taulukko 1, s. 3]. `write()`:n kohdalla ×= ei atominen millään datamäärillä, ▟ = atominen korkeintaan levysektorin kokoisilla kirjoituksilla, ▙ = atominen korkeintaan tiedostojärjestelmälohkon kokoisilla kirjoituksilla, ja ▙ = atominen millä tahansa datamäärillä. Muiden operaatioiden osalta • ilmaisee atomista operaatiota ja × atomisuuden puutetta.

ylikirjoitus. Tämä on ongelmallista, koska nyt tiedostojärjestelmä ei enää toimikaan kunnollisena abstraktiona laitteistolle. Tiedoston loppuun tapahtuu tiedostojärjestelmän sisällä tyypillisesti kahdesta alioperaatiosta: ensin tiedostolle varataan lisää datalohkoja ja kasvatetaan sen kokoa, jonka jälkeen lisättävä data kirjoitetaan. Tämä toteutusyksityiskohta ilmenee siten ettei tiedoston loppuun lisääminen pienilläkään datamäärillä välttämättä ole atominen `ext2`, `ext3-writeback`, `ext4-writeback` ja `reiserfs-writeback`-tiedostojärjestelmissä. Voidaan päätyä tilanteeseen, jossa tiedoston kokoa on ehditty kasvattamaan mutta itse dataa ei ole vielä kirjoitettu, jolloin kaatumistilanteen jälkeen tiedoston lopusta voidaan havaita nollatavuja lisätyn datan sijaan.

Toinen tutkittu persistenssiominaisuuksien kategoria on tiedostojärjestelmäoperaatioiden järjestyksen vaihtuminen levylle kirjoittaessa. Tilanteissa jossa ohjelmakoodissa tehdään peräkkäin toisistaan riippumattomat operaatiot:

```
1    operaatio1();
2    operaatio2();
```

mutta kaatumistilanteen jälkeen havaitaan tilanne jossa ainoastaan `operaatio2()` on suoritettu loppuun, todetaan operaatioiden uudelleenjärjestymisen olevan mahdollista `operaatio1()`:n ja `operaatio2()`:n välillä.

	ext2	ext2-sync	ext3-writeback	ext3	ext3-datajournal	ext4-writeback	ext4	ext4-nodelalloc	ext4-datajournal	btrfs	xfs	xfs-wsync	reiserfs-nolog	reiserfs-writeback	reiserfs	reiserfs-datajournal
ylikirjoitus	×	●	×	×	●	×	×	×	●	●	×	×	×	×	×	●
{lisäys;uudelleennimeäminen}	×	●	×	●	●	×	●	●	●	●	●	●	×	×	●	●
{open(O_TRUNC);lisäys}	×	●	×	●	●	×	●	●	●	●	●	●	×	×	●	●
lisäys tiedoston loppuun	×	●	×	●	●	×	×	●	●	×	×	●	×	×	●	●
hakemisto-operaatio	×	●	●	●	●	●	●	●	●	×	●	●	●	×	●	●

Taulukko 2: Uudelleenjärjestelyä koskevat peristenssioperaatiot eri tiedostojärjestelmillä [PCA+14, taulukko 1, s. 3]. ×-symbolilla merkityt operaatiot saatetaan kirjoittaa levylle eri myöhemmin kuin sitä ohjelmakoodissa seuraavat operaatiot. • tarkoittaa järjestyksen säilyvän.

Taulukossa 2 on listattu operaatioita joiden kirjoitusjärjestys saattaa vaihtua myöhempien tiedostojärjestelmäoperaatioiden kanssa. Merkintä $\{X;Y\}$ tarkoittaa että operaatiot X ja Y ovat perättäisiä operaatioita samalle tiedostolle, ja että kummankaan operaation järjestys ei vaihdu myöhempien operaatioiden kanssa. Uudelleenjärjestelyiden suhteen jotkut tiedostojärjestelmät on erikseen mahdollista konfiguroida tekemään kaikki kirjoitukset järjestyksessä: `ext2`:n tapauksessa optiolla `sync` ja journaloivien tiedostojär-

jestelmien kohdalla optiolla `datajournal`. Toisaalta paremman suorituskyvyn omaava `writeback`-optio aiheuttaa enemmän mahdollisia operaatioiden välisiä uudelleenjärjestelyjä.

5 Katsaus sovellusten kaatumisturvallisuuteen

Edellisessä luvussa nähtiin, miten eri tiedostojärjestelmillä tapahtuu merkittävästi erilaisia seurauksia kaatumistilanteissa. Tämä on järjestelmän ylläpitäjän ja sovellusten kannalta varsin harmillista, sillä toiseen tiedostojärjestelmään siirtyminen voikin aiheuttaa sen, että aiemmin kaatumisturvalliseksi todettu sovellus ei enää olekaan sitä. Tarkastellaankin vielä tutkimusta sovellusten kaatumisturvallisuudesta eri tiedostojärjestelmillä.

5.1 Sovellusten päivitysprotokollat

Aiemmin aliluvussa 3.2 nähtiin, miten tekstieditorin tiedoston tallennuksesta saatiin tehtyä kaatumisturvallinen POSIX-tiedostojärjestelmäoperaatioita koostamalla. Tässä kyseisessä käyttötapauksessa päästiin haluttuun lopputulokseen varsin suoraviivaisella toteutuksella, jossa yksinkertaisesti kirjoitettiin koko tiedosto uudestaan. Muuntyyppisillä sovelluksilla voi olla huomattavastikin monimutkaisempia vaatimuksia. Esimerkiksi SQL-kyselykielen perustoiminnallisuuksiin kuuluvat transaktiot, joiden avulla relaatiotietokannassa voidaan atomisesti päivittää useita, mahdollisesti eri tauluihin kuuluvia rivejä atomisesti [EN10]. Jo yksittäiset tietokannat voivat nykyään kuitenkin olla useiden teratavujen kokoisia ja käsitellä useita toisistaan riippumattomia kirjoitustransaktioita rinnakkain [EN10]. Selvästikin kelvollisen suorituskyvyn saavuttamiseksi koko tietokannan uudelleen kirjoitus joka transaktion yhteydessä ei tule kysymykseen tällaisessa käyttötapauksessa, vaan tietokannan ohjelmoijan täytyy itse suunnitella jokin mekanismi, jolla kaatumistilanteessa voidaan palauttaa tietokannan tila konsistentiksi. Näitä

sovellusten käyttämiä mekanismeja kutsutaan *päivitysprotokolliksi* (*update protocol*) [PCA+14].

Otetaan esimerkiksi SQLite-tietokantakirjasto, joka tarjoaa yksinkertaisen tiedostojärjestelmää käyttävän SQL-kyselykieleen pohjautuvan tietokantan. SQLite tukee SQL-kielen transaktioita, mutta tukee vain yhtä samanaikaista transaktiota kerrallaan. Oletuskonfiguraatiossaan SQLite käyttää *peruutuslokia* (*rollback journal*) tietokantasivujen atomiseen päivittämiseen [DCW16]. Pääpiirteissään SQLite tekee seuraavat tiedostojärjestelmäoperaatiot jokaisen kirjoittavan tietokantatransaktion yhteydessä [PCA+14]:

1. `creat(journal)`
2. $n \times$ `append(journal)`
3. `fsync(journal)`
4. `fsync(parent-dir)`
5. `write(journal)`
6. `fsync(journal)`
7. `write(db)`
8. `fsync(db)`
9. `unlink(journal)`

Transaktion alkaessa luodaan kohdassa 1. tietokantatiedostosta erillinen tiedosto peruutuslokille. Ennen varsinaisen tietokantatiedoston päivitystä kirjoitetaan kohdassa 2 peruutuslokiin tieto siitä, mitä tietokantasivuja transaktiossa ollaan muokkaamassa, sekä kunkin muokatun sivun alkuperäinen sisältö. Kohdissa 3 ja 4 pakotetaan sekä peruutuslokin että peruutuslokitiedostoon viittaavan hakemistoalkion kirjoitus levylle. Kohdat 5 ja 6 päivittävät peruutuslokiin otsakkeeseen, että sen kirjoitus on suoritettu loppuun. Kohdassa 7 tehdään muutokset itse tietokantaan, ja pakotetaan tietokannan

muutosten kirjoitus levylle kohdassa 8. Transaktion päättää peruutuslokitiedoston poistaminen kohdassa 9.

Mahdollisissa kaatumistilanteissa perusajatuksena on, että mikäli peruutuslokitiedostoa ei ole tai sen otsakkeen perusteella sitä ei ole kirjoitettu kokonaan loppuun, on itse tietokanta konsistentissa tilassa ja palautumiseen riittää peruutuslokitiedoston poistaminen. Muissa tapauksissa peruutuslokin sisältämät alkuperäiset tietokantasivut kirjoitetaan oikeille paikoilleen varsinaiseen tietokantatiedostoon, jolloin tietokanta on palautunut transaktiota edeltävään tilaan.

5.2 Sovellusten päivitysprotokollien kaatumisturvallisuus

Toinen osuus "All File Systems Are Not Created Equal: On the Complexity of Crafting Crash-consistent Applications"[PCA+14]-artikkelista keskittyi tutkimaan sovellusten päivitysprotokollien kaatumisturvallisuutta empiirisesti. Jokaiselle tutkitulle sovellukselle kehiteltiin jokin testitapaus, jonka aikana kaikki sovelluksen tekemät tiedostojärjestelmäkutsut nauhoitettiin ylös lokiin, sekä jokaiselle testitapaukselle vastaava tarkistusohjelma. Nyt tiedostojärjestelmäkutsulokin perusteella voidaan muodostaa kaikki mahdolliset kaatumistilanteen jälkeiset tiedostojärjestelmän tilat, jotka ovat riippuvaisia simuloidun tiedostojärjestelmän persistenssiominaisuuksista. Jokaiselle mahdolliselle tiedostojärjestelmän tilalle puolestaan ajetaan tarkistusohjelma, joka kertoo onko kaatumisen jälkeinen tilanne konsistentti sovelluksen näkökulmasta. Esimerkiksi SQL-tietokannan kohdalla mielekäs testitapaus voisi olla saman arvon lisääminen kahteen eri tauluun yhden transaktion sisällä. Tätä testitapausta vastaava tarkistusohjelma voisi tarkastaa ovatko edellä mainittujen taulujen sisällöt samat. Jos yhdelläkin simuloidulla kaatumistilanteella tarkistusohjelma ilmaisee sovelluksen tilan olevan sisäisesti epäkonsistentti, sanotaan että sovelluksella on *kaatumishaavoittuvuus* (*crash vulnerability*) testatulla tiedostojärjestelmällä.

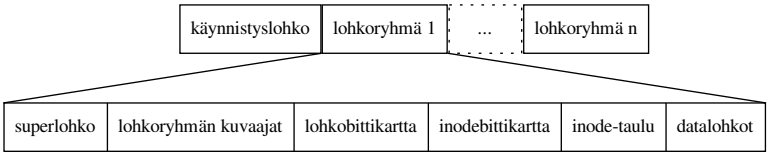
	ext3	ext4	btrfs
LevelDB 1.10	1	2	4
LevelDB 1.15	1	2	3
LMDB			
GDBM	3	3	4
HSQLDB			4
SQLite-Rollback	1	1	1
SQLite-WAL			
PostgreSQL			
HDFS			1
ZooKeeper	1	1	1

Taulukko 3: Tietokantasovelluksista löytyneitä kaatumishaavoittuvuksia eri tiedostojärjestelmillä [PCA+14, taulukko 3(c), s. 11].

Taulukossa 3 nähdään testatuissa tietokantasovelluksissa havaittujen kaatumishaavoittuvuuksien määrä kolmella eri Linux-tiedostojärjestelmällä. Kaatumishaavoittuvuuksia ylipäätänsä löytyi valtaosasta sovelluksia. Kahden sovelluksen kohdalla esiintyi haavoittuvuuksia, jotka ilmenivät ainoastaan `btrfs`-tiedostojärjestelmällä.

6 `ext2`-tiedostojärjestelmä

Edellisissä luvuissa nähtiin sovellusohjelmoijan näkökulmasta millaisia tiedostojärjestelmärajapintoja POSIX-spesifikaatio antaa sovellusten käytettäväksi sekä kuinka niitä täytyy käyttää oikein kaatumisturvallisuuden säilyttämiseksi. Nyt siirrytään tarkastelemaan miten itse tiedostojärjestelmän pitää toimia, jotta kaatumisturvallisuus säilyy koko kokonaisuuden kannalta. Esimerkiksi aiemmin aliluvussa 3.2 nähdyn tekstieditoriesimerkin toiminnan kannalta on kriittistä, että `rename()`-kutsu toimii atomisesti, joten tiedostojärjestelmän täytyy siis pystyä toteuttamaan tämä sekä joitakin muita lupauksia tiedostojärjestelmäkutsujen toiminnasta. Tässä luvussa aloitetaan perehtymään Linuxin `ext2`-tiedostojärjestelmän sisäiseen toimintaan ja sen rakenteeseen levyllä.



Kuva 1: `ext2`:n rakenne levyllä korkealla tasolla [BC05].

6.1 `ext2`:n perusteet

`ext2` ("Second extended filesystem") on tiedostojärjestelmä, joka on luotu jo Linuxin varhaisina aikoina, vuosina 1992–1993 [CTT95]. Alun perin Linuxilla ei ollut omaa varta vasten suunniteltua tiedostojärjestelmää, vaan se käytti MINIX-käyttöjärjestelmän tiedostojärjestelmää vain koska Linuxin alkuperäinen kehitys oli tehty MINIXillä. MINIX-tiedostojärjestelmä kuitenkin sisälsi erinäisiä rajoitteita, jotka alkoivat haittaamaan. Esimerkiksi tiedostonimen pituus oli rajoitettu 14 merkkiin ja tiedoston koko 64 megatavuun [CTT95]. `ext2` ei ainoastaan korjaa näitä puutteita, mutta on lisäksi suunniteltu siten että tiedostojärjelmän levyformaattia voidaan laajentaa taaksepäin yhteensopivasti. `ext2`:lle onkin kehitetty seuraajat `ext3` ja `ext4`, joissa olemassa oleva `ext2`- tai `ext3`-osio voidaan päivittää seuraavaan versioon ilman tiedostojärjestelmän uudelleenformatointia [MCB+07].

Yleisellä tasolla `ext2` on varsin suoraviivainen toteutus Unix-tyylisestä tiedostojärjestelmästä, ja lisäksi vahvasti BSD-käyttöjärjestelmän Berkeley Fast Filesystem-tiedostojärjestelmän inspiroima [CTT95]. Vahva ymmärrys `ext2`:n rakenteesta auttaakin merkittävästi muiden saman aikakauden Unix-tyylisten tiedostojärjestelmien ymmärtämisessä. `ext2`:n levyrakenne korkealla tasolla on esitetty kuvassa 1 [BC05]. Korkeimmalla tasolla `ext2`:n rakenne on jaettu kahteen osaan: on *käynnistyslohkoon* sekä useisiin *lohkoryhmiin*. Käynnistyslohkon tarkoitus on yksinkertaisesti varattua tilaa jollekin `ext2`:ta riippumattomalle käynnistyslataajalle, itse tiedostojärjestelmä ei käytä tätä tilaa mitenkään. Valtaosa `ext2`:n levynkäytöstä on lohkoryhmiä, joista jokainen lohkoryhmä viimeistä lukuun ottamatta vie yhtä paljon tilaa

levyltä. Jokainen `ext2`:n lohkoryhmä on puolestaan jaettu seuraaviin tietorakenteisiin: *superlohkoon* (*superblock*), *lohkoryhmien kuvaajiin* (*block group descriptors*), *lohkobittikarttoihin* (*block bitmap*), *inodebittikarttoihin* (*inode bitmap*), *inodetauluun* (*inode table*) sekä datalohkoihin. Näistä superlohko ja lohkoryhmien kuvaajat ovat koko tiedostojärjestelmälle yhtenäistä metadataa, joiden periaatteessa riittäisi olla levyllä vain kerran, mutta `ext2` säilyttää kuitenkin kummastakin niistä useita kopioita siltä varalta että ensisijainen kopio näistä tietorakenteista korruptoituu.

6.2 Superlohko ja lohkoryhmäkuvaajat

Superlohko sisältää muun muassa seuraavanlaisia arvoja tiedostojärjestelmästä, enkoodattuna 32-bittisinä little-endian-kokonaislukuina [BC05]:

- Tiedostojärjestelmän lohkon (block) koko `s_log_block_size`. Yleisesti käytettyjä lohkon kokoja on 1, 2, ja 4 kilotavua. Mahdollisia tiedostojärjestelmän lohkokokoja rajoittaa se, että lohkon koon täytyy olla käytettävän levyn sektorin koon moninkerta.
- Ominaisuusbitit `s_feature_compat`, `s_feature_incompat` ja `s_feature_ro_compat` jotka kertovat mitä laajennoksia tiedostojärjestelmässä on käytössä. Nämä ovat jaettu kolmeen eri luokkaan mikä mahdollistaa sekä taaksepäin- että eteenpäin yhteensopivuuden tietyissä rajoissa.
- Lohkoryhmien koot, erikseen datalohkoille ja inodeille. `s_blocks_per_group` kertoo datalohkojen määrän yhdessä lohkoryhmässä ja `s_inodes_per_group` inodejen lukumäärän yhdessä lohkoryhmässä.
- Vapaiden datalohkojen määrä `s_free_blocks_count` ja datalohkojen kokonaismäärä `s_blocks_count`.
- Niin ikään samat laskurit inodeille: `s_free_inodes_count` kertoo vapaiden inodejen määrän ja `s_inodes_count` inodejen kokonaismäärän tiedostojärjestelmässä.

ext2:ssa inodeille on varattu tila, joka on erillinen datalohkoista. Tiedostojärjestelmää luodessa valitut s_blocks_per_group- ja s_inodes_per_group-parametrit määräävät inodejen ja datalohkojen suhteellisen osuuden koko levytilasta. ext2:aa käyttäessä voikin joutua tilanteeseen, jossa käyttäjän näkökulmasta levytilaa (eli datalohkoja) on runsaasti vapaana, mutta kaikki inodet ovat käytössä, tai päinvastoin.

Kuten superlohkon kohdalla, lohkoryhmäkuvaajista säilytetään kopioita useassa kohtaa levyä. Lohkoryhmäkuvaajat sisältävät yhteenvetotietoa jokaisesta tiedostojärjestelmän lohkoryhmästä. Jokaiselle lohkoryhmälle pidetään yllä seuraavia arvoja:

- bg_block_bitmap kertoo lohkon, josta kyseisen lohkoryhmän lohkobittikartta alkaa. Bittikartan bitit kertovat mitkä lohkoryhmän datalohkoista ovat käytössä. Ykkösbitti tarkoittaa käytössä olevaa lohkoa ja nollabitti vapaata lohkoa.

- bg_inode_bitmap kertoo lohkon, josta kyseisen lohkoryhmän inodebittikartta sijaitsee. Inodebittikartta kertoo mitkä kyseisen lohkoryhmän inodeista on käytössä. Samaan tapaan kuin lohkobittikartan kanssa, 1 tarkoittaa käytössä olevaa inodea ja 0 vapaata inodea.

- bg_free_blocks_count ja bg_free_inodes_count kertovat vapaiden lohkojen ja inodejen lukumäärän kyseisessä lohkoryhmässä.

- bg_inode_table kertoo mistä lohkosta kyseisen lohkoryhmän inodetaulu alkaa. Lohkoryhmän datalohkot sijaitsevat heti lohkoryhmän inodetaulun jälkeen.

6.3 Inoden rakenne

ext2:ssa tietyn inoden paikannus inodenumeron perusteella on yksinkertaista. Kuten edellä nähtiin, jokaisessa lohkoryhmässä on sama määrä inodeja ja

lohkoryhmän sisällä inodet sijaitsevat perättäin taulukossa. Inodeiden numerointi alkaa ykkösestä, joten inoden numerosta voidaan laskea lohkoryhmän numero kaavalla:

$$n_{\text{bg}} = \frac{n_{\text{ino}} - 1}{\texttt{s\_inodes\_per\_group}}$$

Jakolaskun jakojäännös kertoo puolestaan inoden paikan lohkoryhmässä.

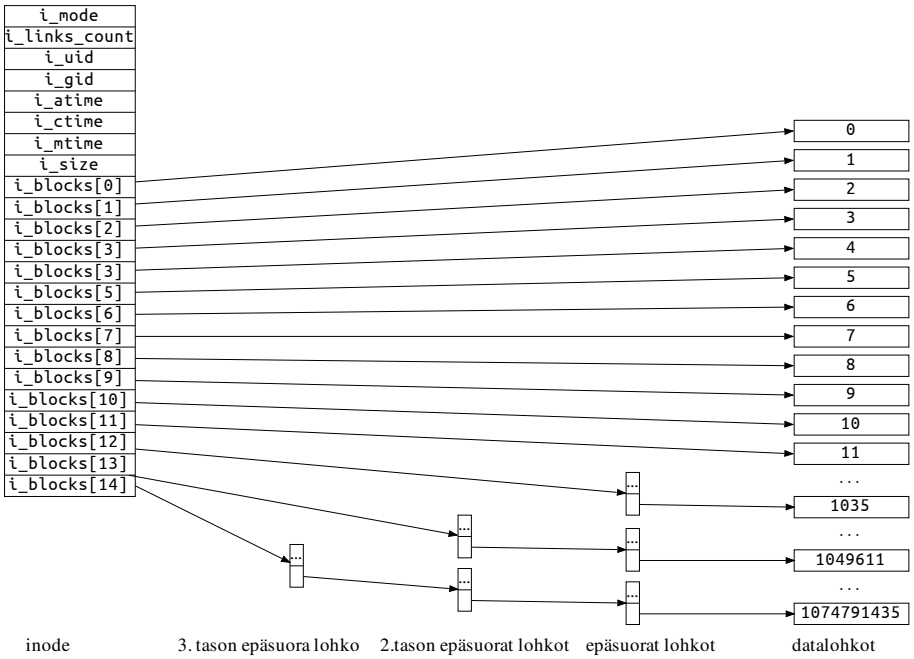
`ext2`-tiedostojärjestelmän inoden rakenne on esitetty kuvassa 2. Suurin osa inoden kentistä sisältää tiedoston metadatan, joka on luettavissa `fstat()`-funktiolla. Esimerkiksi kentät `i_atime`, `i_ctime` ja `i_mtime` sisältävät inoden aikaleimat, joilla jotka ovat yksi-yhteen vastaavat `struct stat`-tietueen kentillä `st_atim`, `st_ctim` ja `st_mtim`. Tiedoston datalohkoihin viitataan *lohkopuun* (*block map*) avulla. Inoden kenttä `i_blocks` sisältää 15-alkioisen taulukon lohkonumeroita, jotka tulkitaan seuraavasti:

- Taulukon ensimmäiset 12 alkiota sisältävät lohkonumerot tiedoston ensimmäiselle 12 lohkolle (lohkot 0–11). Nämä ovat *suoria lohkoja* (*direct blocks*).

- Taulukon seuraava (13.) alkio kertoo lohkonumeron, joka viittaa ensimmäiseen *epäsuoraan lohkoon* (*indirect block*). Itse epäsuora lohko taas sisältää taulukon lohkonumeroita, jotka kertovat inoden seuraavat datalohkojen numerot. Esimerkiksi jos tiedostojärjestelmän lohkokoko on 4096, sisältää epäsuora lohko 1024 lohkonumeroa, jolloin lohkot 12 – 1035 löytyvät ensimmäisen epäsuoran lohkon kautta.

- Taulukon seuraava (14.) alkio kertoo lohkonumeron, joka viittaa ensimmäiseen *toisen tason epäsuoraan lohkoon* (*doubly indirect block*).

- Taulukon seuraava (15.) alkio kertoo lohkonumeron, joka viittaa ensimmäiseen *kolmannen tason epäsuoraan lohkoon* (*triply indirect block*).



Kuva 2: `ext2`:inode ja lohkopuu 4 kilotavun lohkokoolla.

6.4 Hakemiston rakenne

`ext2`:n hakemisto koostuu *hakemistoalkioista* (*directory entry*), jotka sijaitsevat hakemistoinoden datalohkoissa. Hakemistoalkion koko on vaihtuvan mittainen ja riippuu hakemistoalkion sisältämän tiedostonimen pituudesta. Hakemistoalkio sijaitsee aina kokonaan yhden datalohkon sisällä. Hakemistoalkio koostuu vakiokokoisesta otsakkeesta jota välittömästi seuraa tiedostonimi. Otsakkeen rakenne on seuraavanlainen [BC05]:

```
struct ext2_dir_entry_2 {
    uint32_t inode;
    uint16_t rec_len;
    uint8_t name_len;
    uint8_t file_type;
};
```

Kenttä `inode` sisältää hakemistoalkion viittaavan inoden numeron, jossa

inodenumero 0 tarkoittaa käyttämätöntä tilaa hakemistossa. Otsakkeen kenttä `name_len` kertoo tiedostonimen pituuden. Koska kentän koko on 8 bittiä, rajoittuu tiedostonimen maksimipituudeksi 255. Kenttä `rec_len` kertoo koko hakemistoalkion koon, jonka perusteella voidaan paikantaa hakemiston seuraava hakemistoalkio, joka voi olla suurempi kuin otsakkeen koko ja tiedostonimen pituus. Esimerkiksi 1024 tavun lohkokokoa käytettäesä alihakemisto, johon on luotu tiedostot `abc`, `defghijklmn` ja `opqrstuvwxyz`, voisi näyttää seuraavalta:

`inode`	`rec_len`	`name_len`	tiedostonimi
12	12	1	`"."` sekä 3 0-tavua.
2	12	2	`".."` sekä 2 0-tavua.
13	12	3	`"abc"` sekä 1 0-tavu.
14	20	11	`"defghijklmn"` sekä 1 0-tavu.
15	968	12	`"opqrstuvwxyz"`, sekä loput lohkosta 0-tavuja.

7 `ext2` kaatumistilanteissa

Keskitytään seuraavaksi kaatumistilanteisiin ja niistä syntyviin mahdollisiin epäkonsistenssiuksiin `ext2`:n levytietorakenteisiin. Tarkastellaan ensin tiedostojärjestelmän konsistenssiuden määritteleviä invariantteja sekä miten niitä tarkistetaan ja korjataan `fsck`-työkalulla. Lisäksi tutkitaan käytännössä, miten Linuxin `ext2`-tiedostojärjestelmäajuri tekee levykirjoituksia ja mitä käytännön seurauksia kaatumistilanteilla on.

7.1 Tiedostojärjestelmän invariantit ja tarkistusohjelma *fsck*

Kuten edellisen luvun kuvauksesta `ext2`:n levytietorakenteista selvisi, yksinkertainenkin tiedostojärjestelmä koostuu useasta eri levylle kirjoitettavasta tietorakenteesta. Monet näistä levytietorakenteista ovat selkeästi toisistaan riippuvia. Esimerkiksi `ext2`:ssa vapaiden lohkojen määrää ylläpidetään se-

kä superlohkossa, jokaisessa lohkoryhmäkuvaajassa että lohkobittikartassa. Konsistentissa tiedostojärjestelmässä täytyy siis esimerkiksi päteä seuraavat invariantit: lohkobittikartassa vapaaksi merkittyjä lohkoja on sama määrä kuin mitä superlohkon vapaiden lohkojen laskuri näyttää, jonka täytyy olla edelleen sama määrä kuin mitä jokaisen lohkoryhmän vapaiden lohkojen laskurin summa on. Koska jokainen näistä tietorakenteista sijaitsee levyn eri sektorilla ja saattaa olla usean sektorin kokoinen, ei kaikkia tietorakenteita ole mahdollista päivittää atomisesti. Eri sektoreiden kirjoitusten välissä tapahtuva kaatumistilanne voi siis johtaa tilanteeseen jossa vain osa kirjoituksista on päätynyt levylle asti ja tiedostojärjestelmä ei ole enää konsistentissa tilassa.

Tiedostojärjestelmien konsistenssiuden tarkastamiseen ja korjaamiseen käytetään perinteisesti `fsck` (*file system check*)-nimistä työkalua. Koska jokaisella tiedostojärjestelmällä on oma levyformaattinsa, on jokaiselle tiedostojärjestelmälle oma `fsck`-työkalu, joka `ext2`:n tapauksessa on `e2fsck`. Kaatumistilanteiden lisäksi `fsck`-työkalun tulee kyetä tunnistamaan ja korjaamaan parhaansa mukaan muitakin mahdollisia korruptioita, kuten esimerkiksi laitteistosta tai tiedostojärjestelmäajurin ohjelmointivirheistä aiheutuneita virheitä. Tämä tekee prosessista varsin monimutkaisen ja edellyttää koko tiedostojärjestelmän metadatan läpikäyntiä, mikä voi suurella tiedostojärjestelmällä kestää pitkään. `e2fsck`:n tapauksessa mahdollisia tiedostojärjestelmään tehtäviä korjauksia on yli 120 ja ne tehdään kuudessa eri vaiheessa [GRADAD08]. Vaiheet ja merkittävimmät niissä tehtävät tarkistukset ja korjaukset ovat:

1. Superlohkon ja lohkoryhmän tarkistus.

2. Inodejen ja lohko-osoittimien tarkistus: kuhunkin data- tai epäsuoraan lohkoon saa olla korkeintaan yksi viittaus.

3. Hakemistojen tarkistus: jokainen hakemistoalkio osoittaa käytössä ole-

vaan inodeen, hakemistoinodeen saa olla korkeintaan yksi viittaava hakemistoalkio,

4. Hakemistopuun yhtenäisyyden tarkistus: kaikkiin käytössä oleviin inodeihin pitää pystyä kulkemaan juurihakemistosta lähtien.

5. Inodeen viittaavien hakemistoalkioiden määrän laskurit päivitetään vastaamaan todellista määrää.

6. Lohko- ja inodebittikartat sekä vapaiden lohkojen ja inodejen laskurit päivitetään vastaavaan todellista tilannetta.

7.2 Kaatumistilanteiden tutkiminen käytännössä

`ext2`-tiedostojärjestelmän käyttäytymistä kaatumistilanteessa testattiin käytännössä QEMU-virtuaalikoneen `blklogwrites`-lohkolaitteen avulla. `blklogwrites` on virtuaalinen lohkolaite, joka näkyy käyttöjärjestelmälle muuten tavallisena levynä, mutta jokaisen kirjoitusoperaation yhteydessä tehdään lisäksi lokimerkintä joka sisältää aikaleiman, kirjoitettavan sektorin numeron sekä lohkon uuden sisällön. Koeasetelmassa virtuaalikoneessa käynnistetään Linux, liitetään tiedostojärjestelmä `blklogwrites`:in päälle ja tehdään sille haluttuja tiedostojärjestelmäoperaatioita, ja sammutetaan virtuaalikone. Nyt levykirjoituslokin pohjalta voidaan jälkikäteen muodostaa jokainen mahdollinen välitilanne levyn sisällöstä, jossa jokainen välitilanne vastaa yhtä mahdollista kaatumistilannetta testitilanteen aikana.

Koeasetelmassa käytettiin Linuxin versiota 4.14, QEMUn versiota 3.0.0 ja 1 gigatavun levyä. Lisäksi tiedostojärjestelmäoperaatioita ei tehdä peräjälkeen tai heti liittämisen jälkeen, vaan niiden välissä odotetaan 3 minuuttia. Tällä lailla voidaan erottaa levykirjoitukset, jotka tapahtuvat suoraan tiedostojärjestelmäoperaation seurauksena ja mitkä käyttöjärjestelmä kirjoittaa taustalla.

7.2.1 Tiedoston luominen

Taulukossa 4 nähdään levykirjoitukset, jota aiheutuu luotaessa tyhjään `ext2`-tiedostojärjestelmän juurihakemistoon uusi tiedosto kutsulla `creat("new", 0755)`:

#	t (s)	Δt (s)	lohkot	kuvaus
1	0.000			`creat()`-kutsu alkaa
2	0.001	+0.001		`creat()`-kutsu loppuu
3	30.847	+30.846	1	Lohkoryhmien kuvaajat: vähennetään vapaiden inodejen määrää (8181 → 8180)
4	30.847	+0.000	3	Lohkoryhmän 0 inodebittikartta: merkitään inode 12 varatuksi
5	30.847	+0.000	516	Juurihakemisto: lisätään hakemistoalkio `new` osoittamaan inodeen 12
6	61.567	+30.721	4	Lohkoryhmän 0 inodetaulu: inoden 12 käyttöoikeudet, aikaleimat, koko sekä lohkoviittaukset alustetaan. Juurihakemistoinoden aikaleimoja päivitetään.

Taulukko 4: Tiedoston luomisesta seuraavat levyoperaatiot `ext2`-tiedostojärjestelmässä.

Esimerkistä ilmenee selvästi miten `ext2` hyödyntää välimuistia levyn metadatarakenteille: `creat()`-kutsu palaa välittömästi eikä siitä aiheudu välittömiä levykirjoituksia. Vasta 30 sekuntia myöhemmin tehdään ensimmäiset kolme kirjoitusta, jota seuraa toinen 30 sekunnin odotus ennen kuin koko operaatio on saatettu loppuun levyllä. Kokonaisuudessaan esimerkki näyttää miten tiedoston luominen koostuu kolmesta osasta: levyltä varataan yksi vapaa inode (tässä esimerkissä 12), alustetaan se ja lisätään hakemistoalkio osoittamaan siihen. Tämän esimerkin kaatumistilanne voi aiheuttaa seuraavia epäkonsistenssiuksia:

1. Kohtien 3–4 välissä: vapaiden inodejen laskuri näyttää vapaita inodeja olevan vähemmän kuin niitä todellisuudessa on.

2. Kohtien 4–5 välissä: inodebittikartta näyttää vapaita inodeja olevan vähemmän kuin niitä todellisuudessa on.

3. Kohtien 5–6 välissä: juurihakemiston hakemistoalkio `new` osoittaa nyt inodetaulussa alustamattomaan inodeen.

Näistä kaksi ensimmäistä johtaa inodejen 'vuotamiseen', eli tiedostojärjestelmässä näyttää olevan vähemmän vapaita inodeja saatavilla kuin siellä todellisuudessa on. Tämä ei sinänsä ole suoraan este tiedostojärjestelmän käytölle, mutta `fsck`-ohjelma täytyy ajaa jotta resurssit saadaan takaisin käyttöön. Kolmannessa tapauksessa seuraukset ovat merkittävästi ikävämpiä. Hakemistoalkio `new` näkyy hakemistolistauksessa mutta kaikki siihen vaikuttavat tiedostojärjestelmäoperaatiot epäonnistuvat virheellä `Input/output error`. Käyttäjän näkökulmasta tiedostojärjestelmään on siis syntynyt 'viallinen' tiedosto jolle ei voi tehdä mitään, edes poistaa.

7.2.2 Tiedostoon kirjoitus

Jatketaan edellisen kohdan tiedostojärjestelmästä ja kirjoitetaan `new`-tiedostoon 64 kilotavun eli 16 lohkon verran dataa. Taulukosta 5 nähdään siitä syntyvät levykirjoitukset:

#	t (s)	Δt (s)	lohkot	kuvaus
1	0.000	+0.000		`open()`-kutsu alkaa
2	0.028	+0.028		`open()`-kutsu loppuu
3	0.028	+0.000		`write()`-kutsu alkaa
4	0.028	+0.000		`write()`-kutsu loppuu
5	5.226	+5.198	1	Lohkoryhmien kuvaajat: vapaiden lohkojen määrää vähennetään (32247 → 32230)
6	5.226	+0.000	2	Lohkoryhmän 0 lohkobittikartta: varataan lohkot 8192 – 8208
7	5.226	+0.000	4	Lohkoryhmän 0 inodetaulu: inoden `new` attribuuteista päivitetään sen koko, aikaleima ja viittaukset data- ja epäsuoriin lohkoihin
8	5.226	+0.000	8192 – 8208	Tiedoston `new` 16 datalohkoa sekä yksi epäsuora lohko

Taulukko 5: Olemassa olevan tiedoston loppuun lisäyksestä seuraavat levyoperaatiot `ext2`-tiedostojärjestelmässä.

Levyvälimuistien olemassaolo näkyy tässäkin esimerkissä, sillä `write()`-operaatio ei aiheuta levykirjoituksia vaan ne tehdään taustalla vasta viiden sekunnin päästä. Tämän esimerkin kaatumistilanne voi puolestaan aiheuttaa seuraavia epäkonsistenssiuksia:

1. Kohtien 5–6 välissä: vapaiden lohkojen laskuri näyttää vapaita lohkoja olevan vähemmän kuin niitä todellisuudessa on.

2. Kohtien 6–7 välissä: lohkobittikartta näyttää vapaita lohkoja olevan vähemmän kuin niitä todellisuudessa on.

3. Kohtien 7–8 välissä: tiedostoa `new` vastaava inode viittaa alustamattomiin lohkoihin.

Näistä tilanteista kaksi ensimmäistä johtaa datalohkojen vuotamiseen, eli samantyyppiseen tilanteeseen kuten edellisen aliluvun esimerkissä josta aiheutui vapaiden inodejen vuotaminen. Kolmannessa kaatumistilanteessa taas päädytään tilanteeseen, missä tiedoston `new` inoden kokoa on kasvatettu onnistuneesti ja siihen on lisätty viittaukset sekä datalohkoihin että epäsuoraan lohkoon, mutta itse lohkojen sisältö jäi kirjoittamatta. Datalohkojen osalta tämä tietää mahdollista tietoturvaongelmaa, sillä alustamattomat datalohkot ovat voineet olla osa jonkun toisen käyttäjän omistamaa sittemmin poistettua tiedostoa. Nyt ne ovat kaatumistilanteen seurauksena luettavissa `new`-tiedostosta, eikä `e2fsck` kykene tunnistamaan tai korjaamaan tilannetta ollenkaan. Alustamattoman epäsuoran lohkon seuraukset ovat sitäkin vakavammat, koska lohkon aiempi data tulkitaan nyt lohko-osoittimina. Pahimmillaan alustamaton lohko-osoitin voi osoittaa jonkun muun inoden omistamaan lohkoon, jolloin `new`-tiedoston olemassa olevan sisällön ylikirjoittaminen vaikuttaisi johonkin muuhun tiedostojärjestelmän inodeen. Tiedostojärjestelmän käyttö tässä tilanteessa ei siis ole turvallista ennen tiedostojärjestelmän korjausta `e2fsck`-työkalulla.

8 Kirjaavat tiedostojärjestelmät

Edellisessä luvuissa nähtiin mitä seurauksia kaatumistilanteilla on `ext2`-tiedostojärjestelmälle, joiden johdosta koko tiedostojärjestelmän metadata täytyy käydä läpi `e2fsck`-ohjelmalla konsistenssin palauttamiseksi. Tarkastellaan seuraavaksi miten `ext2`:n seuraaja `ext3` käyttää *kirjausta* (*journalling*) varmistamaan että tiedostojärjestelmä säilyy konsistenssina kaatumistilanteiden aikanakin.

8.1 Motivaatio `ext3`:lle

Tarve `ext2`:n jatkokehitykselle syntyi pääosin kaatumistilanteista selviämiseen, sillä kiintolevyjen koon kasvaessa kasvoi myös `fsck`-ohjelman viemä aika samaa tahtia [Twe98]. Tavoitteeksi asetettiin luoda menetelmä kaatumistilanteista selviämiseen joka ei vaadi pitkää palautumisaikaa jolloin tiedostojärjestelmä ei ole käytettävissä. Nopeuden lisäksi samalla tavoiteltiin parannuksia tiedostojärjestelmän kaatumisesta palautumiseen seuraavilla kriteereillä [Twe98]:

- Säilyvyys: kaatumistilanteesta palautuminen ei saa vaikuttaa dataan tai metadataan, joka on onnistuneesti kertaalleen päätynyt levylle asti ennen kaatumistilannetta.
- Ennakoitavuus: kaatumistilanteesta palautuessa tehtävät vaiheet täytyy olla ennakoitavissa.
- Atomisuus: sovellusohjelmoijan näkökulmasta atomiset operaatiot, kuten esimerkiksi tiedoston uudelleennimeäminen `rename()`-kutsulla säilyvät atomisena kaatumistilanteissakin.

Näistä kriteereistä `ext2` täyttää ainoastaan säilyvyyden. Levyvälimuistien takia kaatumistilanteen aikana meneillään on voinut olla useaan POSIX-tiedostojärjestelmäoperaation liittyviä levytietorakenteiden päivityksiä, joten

`e2fsck`:n täytyy varautua tekemään merkittävä määrä erityyppisiä korjauksia. Atomisuuden osalta `ext2` rajoittuu vain siihen, mitä levyjärjestelmä pystyy, eli hakemistoalkion uudelleennimeäminen toisella levysektorilla sijaitsevaan hakemistolohkon sektoriin ei ole atomisesti mahdollista.

Yksi keino kaikkien näiden kriteereiden täyttämiseksi on lisätä tiedostojärjestelmään jokin menetelmä *transaktioille*, joilla koko tiedostojärjestelmän metadataan voidaan tehdä päivityksiä atomisesti. `ext3` toteuttaa transaktioille käyttäen kirjausta, joka pohjautuu pääpiirteissään tietokantajärjestelmissä käytettyyn *write-ahead logging*-menetelmään [MHL+92]. Kirjauksen perusajatuksena on että aina ennen varsinaisten levytietorakenteiden muuttamista kirjataan ylös erilliseen *journaliin* ylös kaikki muutokset, jota transaktion yhteydessä tullaan tekemään. Nyt mahdollisen kaatumistilanteen sattuessa voidaan journalin sisällön perusteella aina toistaa kaikki muutokset ja viedä transaktio loppuun.

8.2 Kirjauksen toteutus `ext3`:ssa

Aiemmin luvussa 6 nähtiin, miten `ext2`:n levyformaattia voidaan laajentaa tulevaisuudessa ominaisuusbiteillä. Vaikka useassa yhteydessä `ext3`:sta puhutaan ikään kuin se olisi oma erillinen tiedostojärjestelmänsä, on se levyformaatin näkökulmasta edelleen `ext2`-tiedostojärjestelmä mutta varustettuna lisäominaisuuksilla. Relevantteja ominaisuusbittejä ovat ‘taaksepäin yhteensopivat’ ominaisuudet `has_journal` ja `dir_index` sekä ‘ei-taaksepäin yhteensopiva‘ ominaisuus `needs_recovery`. Ominaisuus `dir_index` tuo hakemistoihin hajautukseen pohjautuvan indeksirakenteen nopeuttamaan hakemistosta hakua tiedostonimen perusteella ja voidaan sivuuttaa kaatumisturvallisuuden tutkimisen kannalta.

Toisin kuin monille muille `ext2`:n levytietorakenteille, journalin sijaintia levyllä ei ole ennalta määrätty, vaan tila sille varataan erityisen inoden avulla data-alueelta samaan tapaan kuin muille tiedostoille [Twe98]. Muis-

ta inodeista poiketen journalin inode ei ole paikannettavissa normaalisti hakemistorakenteen kautta, vaan se paikannetaan `journal`-laajennoksen tuomasta uudesta superlohkon kentästä. Inoden käyttö journalin lohkojen paikantamiseen mahdollistaa journalin lisäämisen olemassa olevaan `ext2`-tiedostojärjestelmään helposti, kunhan tiedostojärjestelmässä on vapaata tilaa, ja ilman että koko tiedostojärjestelmää täytyy alustaa uudelleen.

Itse journalin levyformaatti alkaa *otsakelohkolla* (*header block*) joka kertoo journalin alkukohdan, eli mistä lohkosta eteenpäin journalia aletaan lukemaan kaatumistilanteesta palautuessa [PADAD05]. Journaliin kirjoitettu kokonainen transaktio koostuu *kuvaajalohkosta* (*journal descriptor block*), jota seuraa joukko metadatalohkoja, ja jonka päättää *sitoutumislohko* (*journal commit block*). Kuvaajalohko aloittaa transaktion kertomalla mitä lohkoja transaktion aikana tullaan muokkaamaan, eli muokattavien lohkojen lukumäärä n ja lista lohkonumeroita. Kuvaajalohkoa seuraa kunkin muokattavan lohkon sisältö, eli yhteensä n sisältölohkoa. Transaktion päättää sitoutumislohko, jonka olemassaololla voidaan tunnistaa journaliin keskeneräisesti kirjoitetut transaktiot. Kaatumisesta palautuminen tämän pohjalta on varsin yksinkertaista. Journalia aletaan lukea otsakelohkon ilmaisemasta kohdasta eteenpäin, ja jokaisen kokonaan lokiin kirjoitetun transaktion sisältölohkot kirjoitetaan kuvaajalohkon osoittamiin paikkoihin levyllä.

`ext3`:ssa kirjausta käytetään aina itse tiedostojärjestelmän metadatan päivitykseen. Tähän lukeutuu superlohko, lohkoryhmien kuvaajat, inodetaulu, inode- ja lohkobittikartat, sekä data-alueella hakemistolohkot ja epäsuorat lohkot. Tiedostojen datalohkojen osalta `ext3` tarjoaa kolme eri journalointitilaa, joista järjestelmän ylläpitäjä voi valita mitä käytetään [PADAD05]:

- `writeback`-tila: Kirjausta ei käytetä datalohkoille, ja datalohkojen kirjoitus tehdään täysin riippumattomasti journalikirjoituksista.
- `ordered`-tila: Kirjausta ei käytetä datalohkoille, mutta datalohkojen kirjoitus tapahtuu aina ennen transaktion sitoutumista. Tämä on `ext3`:n

oletus.

- `datajournal`-tila: Kirjausta käytetään myös datalohkoille, eli ne kirjoitetaan ensin journaliin ja vasta transaktion sitoutumisen jälkeen varsinaisille sijainneille levyllä.

Aiemmin luvussa 3 vertailtiin eri tiedostojärjestelmien persistenssiominaisuuksia, ja nämä kolme journalointitilaa vastaavat suoraan tutkittuja tiedostojärjestelmäkonfiguraatioita `ext3-writeback`, `ext3` ja `ext3-datajournal`. Huomionarvoista on, että `writeback`-journalointitilaa käytettäessä alustamattomista datalohkoista johtuva tietoturvaongelma on edelleen mahdollinen kaatumistilanteessa, muilla journalointitiloilla tätä ei esiinny.

8.3 Käytännön esimerkki `ext3`:n tekemistä levykirjoituksista

`ext3`:n tekemiä levykirjoituksia tutkittiin QEMU-virtuaalikoneella samalla testiasetelmalla kuin aiemmin `ext2`:n kohdalla tehtiin luvussa 6. 1 gigatavun levylle `mkfs.ext3` loi rakenteeltaan muuten samanlaisen tiedostojärjestelmän kuin `ext2`:n tapauksessa, paitsi että lohkoryhmä 0 sisältää 32 megatavun kokoisen journalin. Taulukossa 6 nähdään levykirjoitukset, jota aiheutuu kun tiedostojärjestelmään luodaan uusi tiedosto `new` ja kirjoitetaan 64 kilotavua dataa seuraavalla ohjelmalla:

```
1    int fd = creat("/mnt/fs/new", 0644);
2    write(fd, "XXX...XXX", 64 * 1024);
3    close(fd);
```

#	aika (s)	Δt (s)	lohkot	kuvaus
1	0.000	+0.000		`creat()`-kutsu alkaa
2	0.001	+0.001		`creat()`-kutsu loppuu
3	0.002	+0.000		`write()`-kutsu alkaa
4	0.002	+0.001		`write()`-kutsu loppuu
5	5.411	+5.408	521	Journalin otsakelohko: palautuminen on tarpeen alkaen journalin lohkosta 1
6	5.411	+0.000	9216 – 9231	Tiedoston `new` datalohkot
7	5.411	+0.000	522 – 529	Journalin lohkoihin 1–8 kirjoitetaan yksi kuvaajalohko sekä 7 sisältölohkoa. Päivitettävät lohkot ovat 3, 1, 4, 516, 0, 2, 8722
8	5.411	+0.000	530	Journalin lohko 9: sitoutumislohko
9	36.131	+30.720	0	Superlohko: vapaiden lohkojen ja inodejen määriä vähennetään
10	36.131	+0.000	1	Lohkoryhmien kuvaajat: vapaiden lohkojen ja inodejen määriä vähennetään
11	36.131	+0.000	2	Lohkobittikartta: merkitään lohkot 8722 ja 9216–9231 varatuksi
12	36.131	+0.000	3	Lohkoryhmän 0 inodebittikartta: merkitään inode 12 varatuksi
13	36.131	+0.000	4	Lohkoryhmän 0 inodetaulu: inoden 12 käyttöoikeudet, aikaleimat, koko sekä lohkoviittaukset alustetaan. Juurihakemistoinoden aikaleimoja päivitetään.
14	36.131	+0.000	516	Juurihakemisto: lisätään hakemistoalkio `new` osoittamaan inodeen 12
15	36.131	+0.000	8722	Tiedoston `new` epäsuora lohko: tiedoston lohkot 12–15 merkitään viittaamaan lohkoihin 9228–9231

Taulukko 6: Tiedoston luonnista ja sinne kirjoittamisesta seuraavat levyoperaatiot `ext3`-tiedostojärjestelmässä.

Molemmat testiohjelman tekemät tiedostojärjestelmäkutsut valmistuvat aiheuttamatta yhtään levykirjoitusta, eli `ext2`:n tapaan operaatiot tehdään ensin pelkästään keskusmuistissa ja kirjoitetaan vasta myöhemmin taustalla levylle. Itse levykirjoitukset alkavat vasta noin viiden sekunnin päästä transaktion avaamisella kohdassa #5. Sekä `creat()`- että `write()`-kutsuista aiheutuneet metadatapäivitykset ovat yhdistetty yhdeksi transaktioksi, jonka kuvaajalohko sekä 7 sisältölohkoa kirjoitetaan journaliin kohdassa #7.

Kun käytetään `ordered`-journalointitilaa, ei `new`-tiedostoon kirjoitettuja datalohkoja `new` kirjoiteta journaliin. Kaikkien datalohkojen pitää kuitenkin olla kirjoitettuna ennen kuin transaktion sitoutuminen voidaan tehdä, joten niiden kirjoitus tehdäänkin kohdassa #6. Kohdassa #8 kirjoitettu sitoutumislohko päättää transaktion. Lopulta voidaan tehdä päivitykset tiedostojärjestelmän varsinaisiin levyrakenteisiin kohdissa #9–#15, jotka tapahtuvat täysin samalla tavalla kuin `ext2`-tiedostojärjestelmässä.

9 Lokipohjaiset ("log-structured") tiedostojärjestelmät

Aiemmissa luvuissa käsitellyt `ext2`/`ext3` ja Berkeley FFS ovat niin sanottuja päälle kirjoittavia tiedostojärjestelmiä (*overwrite-based filesystem* [RBM13]). Esimerkiksi aiemmin `ext3`:n yhteydessä nähtiin kuinka olemassa olevaa tiedostoa uudelleennimettäessä alkuperäisen tiedostonimen hakemistoalkion sisältävä sektorilta poistetaan vanha hakemistoalkio ja vastaavasti kohdehakemistossa lisätään uusi hakemistoalkio. Mutta koska kahden eri levylohkon päivitys atomisesti ei ole mahdollista, täytyy `ext3`:n kirjoittaa kopiot molemmista levylohkoista journaliin kaatumistilanteen varalta. Lokipohjaiset tiedostojärjestelmät sen sijaan kääntävät tämän tilanteen päälaelleen – muutokset metadataan kirjoitetaan edelleen lokiin, mutta tästä lokiin kirjoitetusta lohkosta tuleekin uusi sijainti kyseiselle metadatalle. Tällä tekniikalla vältytään siis metadatan kahdesti kirjoittamiselta, mutta toisaalta nyt on tarpeen tehdä päivityksiä muualla päin tiedostojärjestelmää kertomaan muokatun metadatalohkon uusi sijainti levyllä. Tässä luvussa tutkitaan Sprite LFS:ää, joka on varhainen toteutus lokipohjaisesta tiedostojärjestelmästä [Ros92].

9.1 Sprite LFS:n periaate

Sprite LFS on vuosina 1989–1990 kehitelty tiedostojärjestelmä Berkeleyn yliopiston kokeelliseen Sprite-käyttöjärjestelmään [Ros92]. Motivaatio uuden tiedostojärjestelmän kehittämiseen oli sekä parempi kaatumisturvallisuus että suorituskyky levylle kirjoittaessa. Perinteisiä mekaanisia kiintolevyjä käytettäessä suorituskyvyn kannalta huomioon otettavaa on niille ominainen *hakuaika*, mikä tarkoittaa että levyltä useiden lohkojen lukeminen tai kirjoittaminen vie enemmän aikaa mikäli lohkot eivät sijaitse levyllä peräkkäin. Tämän johdosta lokipohjaisten tiedostojärjestelmien levyformaatit ovatkin suunniteltu siten, että valtaosa levykirjoituksista tehdään kirjoittamalla mahdollisimman pitkinä ja perättäisinä kirjoituksina tiedostojärjestelmän *lokin* loppuun. Erityisesti lokin 'keskellä' olevien lohkojen sisältöä ei koskaan käydä uudelleenkirjoittamassa, vaan Sprite LFS ottaa erilaisen lähestymistavan tilanteisiin jossa jotain olemassa olevaa levytietorakennetta täytyy päivittää. Olemassa olevan levytietorakenteen sisältävän levylohkon muokkaamisen sijaan kirjoitetaan lokin loppuun uusi lohko, johon haluttu muokkaus on tehty, ja tästä uudesta lohkosta tulee uusi sijainti kyseiselle levytietorakenteelle. Tämän mekanismin varjopuolena on tietysti että jokaisen muokkausoperaation jälkeen vanha versio levytietorakenteen sisältäneestä lohkosta jää 'kuolleeksi' viemään tilaa levyltä. Ennen pitkää tämä kuolleiden lohkojen käyttämä tila täytyy vapauttaa takaisin tiedostojärjestelmän käyttöön, mikä tapahtuu *segmentin puhdistuksella* (*segment cleaning*).

9.2 Sprite LFS:n levyrakenne

Sprite LFS:llä on merkittävästi yhtenäisyyksiä perinteisen Berkeley Fast Filesystem-tiedostojärjestelmän kanssa ja siten myös `ext2`:n kanssa [Ros92]. Tässä luvussa selitetään pelkät eroavaisuudet luvussa 6 esiteltyihin `ext2`:n levytietorakenteisiin.

Valtaosa tiedostojärjestelmän sisällöstä sijaitsee lokissa (*log*), joka on

jaettu joko 512 kilotavun tai 1 megatavun kokoisiin *segmentteihin* (*segment*). Tiedostojärjestelmän lokialueelle kirjoitetaan seuraavantyyppisiä lohkoja:

- Datalohkot: tavallisten tiedostojen tai hakemistojen sisältämä data. Hakemistot toimivat samalla periaatteella kuin `ext2`:ssa, eli hakemistoinoden datalohkot sisältävät hakemistoalkioita peräkkäismuodossa.
- Epäsuora lohko: toimivat samalla lailla kuin `ext2`:ssa, suurien tiedostojen datalohkoihin viitataan epäsuorien lohkojen muodostamalla lohkopuulla.
- Inodelohkot: sisältö pääosin samanlainen kuin `ext2`:n inode. Suurimpana erona on että Sprite LFS:n inoden sijaintia levyllä ei voi laskea pelkän inodenumeron avulla.
- Inodekartan (*inode map*) lohkot: tietorakenne jonka avulla voidaan paikantaa inoden sijainti levyllä inodenumeron perusteella.
- Segmentin käyttöastetaulun (*segment usage table*) lohkot: ylläpitää tietoa kunkin segmentin vapaiden lohkojen määrästä.
- Segmentin yhteenvetolohko (*segment summary block*): kertoo segmentin sisältämien lohkojen tyypit (data vai metadata). Käytetään kaatumistilanteesta palautumisessa ja segmentin puhdistuksessa.
- Hakemistojen muutoslokilohko (*directory change log block*): käytetään kaatumistilanteesta palautumiseen.

Huomioitavaa on että `ext2`:een verrattuna inodebittikarttaa eikä lohkobittikarttaa ole. Sprite LFS:sä inodeille ei ole erikseen varattu tilaa vaan ne sijaitsevat lokissa datalohkojen lailla ja vapaiden lohkojen määrää ylläpidetään segmentin käyttöastetaulussa bittikartan sijaan.

Lisäksi kaksi levytietorakennetta sijaitsevat kiinteillä sijainneilla levyllä: superlohko (*superblock*) sekä kaksi *tarkistuspistealuetta* (*checkpoint region*).

Superlohko sisältää staattista tietoa tiedostojärjestelmästä, kuten segmentin koko ja niiden lukumäärä, eikä siihen kohdistu kirjoituksia tiedostojärjestelmän luomisen jälkeen. Tarkistuspistealue taas kertoo, miten tiedostojärjestelmän muiden tietorakenteiden senhetkiset versiot voidaan paikantaa. Tähän sisältyy tiedostojärjestelmän inodekarttalohkojen sijainnit, segmentin käyttöastetaulun sisältävien lohkojen sisällöt sekä segmentin numeron johon lokin kirjoitus jatkuu seuraavaksi. Koska tarkistuspistealueen koko on useita lohkoja, on tilaa varattu kahdelle tarkistuspistealueelle, joihin kirjoitetaan vuorotellen. Mikäli kaatumistilanne tapahtuu kesken tarkistuspisteen kirjoittamista, voidaan tiedostojärjestelmää liittäessä keskeneräinen tarkistuspiste havaita ja käyttää toista tarkistuspistettä. Mikäli molemmat tarkistuspisteet ovat konsistentisti kirjoitettuja, valitaan näistä kahdesta se, kumpi on uudempi.

9.3 Esimerkki ja kaatumistilanteista palautuminen

Tarkastellaan esimerkkinä tilannetta, jossa luodaan tiedostojärjestelmän alihakemistoon `testdir` tiedosto `newfile`, ja kirjoitetaan sinne yhden lohkon verran dataa. Kokonaisuutena tiedostojärjestelmälle tehdään seuraavat operaatiot:

1. Hakemiston muutoslokilohko, joka kertoo `testdir`:iin tapahtuvan hakemistoalkion luomisen, kirjoitetaan lokiin.
2. Uuden tiedoston `newfile` datalohko kirjoitetaan lokiin.
3. Hakemiston `testdir` datalohkosta kirjoitetaan uusi versio lokiin, johon on lisätty hakemistoalkio `newfile`.
4. Uuden tiedoston `newfile` inode ja hakemiston `testdir` inodesta kirjoitetaan uusi versio, jossa datalohkoksi on vaihtunut edellisessä kohdassa kirjoitettu lohko. Molemmat lisätään uuteen inodelohkoon, joka kirjoitetaan lokiin.

5. Inodekartan lohkoista kirjoitetaan uudet versiot lokiin, joissa kerrotaan `newfile`:n ja `testdir`:n inodejen sijaitsevan nyt kohdassa 4 kirjoitetussa inodelohkossa.

6. Segmentin käyttöastetaulun lohkoista kirjoitetaan uudet versiot lokiin, koska kohdissa 3.–5. tehdyt päivitykset aiheuttivat kuolleita lohkoja aiempiin lokisegmentteihin.

7. Segmentin yhteenvetolohko kirjoitetaan lokiin. Yhteenvetolohko kertoo kohdissa 1.–3. kirjoitettujen lohkojen olevan datalohkoja ja kohdissa 4.–6. kirjoitettujen lohkojen olevan metadatalohkoja.

8. Uusi tarkistuspiste kirjoitetaan, joka viittaa kohdissa 5. ja 6. kirjoitettuun inodekarttaan ja segmentin käyttöastetauluun.

Lokipohjaiselle tiedostojärjestelmälle on kaksi mahdollista tekniikkaa jota käyttää kaatumistilanteista palautumiseen. Yksinkertaisempi tekniikka on käyttää pelkkää tarkistuspisteitä palautumiseen ja monimutkaisempi mutta suorituskyvyltään parempi tekniikka on lisäksi käyttää *eteenkiertopalautusta* (*roll-forward recovery*).

Pelkkää tarkistuspistettä käytettäessä ei palautumisen yhteydessä tarvitse tehdä mitään tavallisesta tiedostojärjestelmän liittämisestä poikkeavaa, eli yksinkertaisesti jatketaan tiedostojärjestelmän käyttöä uusimman kokonaan kirjoitetun tarkistuspisteen perusteella. Mikäli kaatuminen tapahtuu ennen uuden tarkistuspisteen kirjoittamisen aloittamista eli kohtien 1.–8. välissä, ei kyseisillä kirjoituksilla ole mitään vaikutusta sillä viimeisimmän kirjoituksia edeltäneen tarkistuspisteen näkökulmasta kyseiset kirjoitukset ovat kohdistuneet tiedostojärjestelmän vapaaseen tilaan. Mikäli taas kaatuminen tapahtuu tarkistuspisteen kirjoittamisen aikana, voidaan keskeneräisesti kirjoitettu tarkistuspiste tunnistaa ja samalla tavoin käyttää viimeisimmän kirjoituksia edeltänyttä tarkistuspistettä.

9.4 Eteenkiertopalautus

Edellisessä aliluvussa nähtiin kuinka tiedostojärjestelmään voidaan tehdä muutoksia kaatumisturvallisesti ja atomisesti kirjoittamalla uusi tarkistuspiste. Kokonaisen tarkistuspisteen luominen vaatii kuitenkin suhteellisen monta levykirjoitusta, joten Sprite LFS käyttää lisäksi monimutkaisempaa *eteenkiertopalautusta* (*roll forward recovery*) joka tuo samantasoisen kaatumisturvallisuuden mutta pienemmällä määrällä levykirjoituksia. Eteenkiertopalautuksen perusajatuksena on lähteä liikkeelle viimeisimmästä tarkistuspisteestä samoin kuin aiemmin, mutta tämän jälkeen aletaan lukemaan lokia eteenpäin tarkistuspisteen osoittamasta kohdasta. Tavoitteena on viedä tiedostojärjestelmä viimeisimmästä tarkistuspisteestä tilaan, jossa mahdollisimman paljon viimeisen tarkistuspisteen luomisen jälkeenkin tehdyistä muutoksista on toteutunut.

Tekniikan selkeänä etuna on että kaatumistilanteen aikana katoaa vähemmän meneillään olevia tiedostojärjestelmämuutoksia, mutta myös esimerkiksi `fsync()`-operaatio voidaan nyt toteuttaa tehokkaammin. Ilman eteenkiertoa `fsync()` vaatisi koko tarkistuspisteoperaation tekemistä, mutta eteenkierron kanssa riittää että pelkästään tarvittavat lokikirjoitukset päätyvät levylle. Tarvittavia levykirjoituksia voidaan vähentää vielä lisää huomioimalla että kahta Sprite LFS:n levytietorakennetta, inodekarttaa ja segmentin käyttöastetaulua ei tarvitse kirjoittaa levylle muuten kuin tarkistuspisteen luonnin yhteydessä sillä niihin tehtävät muutokset voidaan aina johtaa muista lokikirjoituksista eteenkiertopalautuksen aikana. Eli esimerkiksi edellisen aliluvun 9.3 esimerkistä voidaan kohdat 5, 6, ja 8 lykätä tehtäväksi myöhemmin.

9.5 Eteenkiertopalautuksen toiminta

Eteenkiertopalautuksen aikana käydään kahdesti läpi kaikki viimeisimmän tarkistuspisteen jälkeen tehdyt lokikirjoitukset [Ros92]. Ensimmäisellä lä-

pikäynnillä tuodaan inodekartta ja segmentin käyttöastetaulu ajan tasalle. Lokin läpikäynti tapahtuu segmentin yhteenvetolohkojen avulla. Jokainen yhteenvetolohko sisältää edellisen kirjoitetun yhteenvetolohkon levyosoitteen sekä levyosoitteen johon seuraava yhteenvetolohko tullaan kirjoittamaan, joiden avulla lokin sisällössä voidaan liikkua eteen- ja taaksepäin, tunnistaa osittain kirjoitetun segmentit sekä tunnistaa lokin loppukohta. Lisäksi segmentin yhteenvetolohkot sisältävät viittauksen kaikkiin kyseiseen segmenttiin kirjoitettuihin inodelohkoihin, jonka perusteella eteenkiertopalautus pystyy tekemään vastaavat päivitykset inodekarttaan. Inodelohkojen perusteella taas voidaan johtaa tarvittavat muutokset segmentin käyttöastetauluun. Kunkin inodelohkon sisältämän inoden uuden version lohkoviittauksia verrataan saman inoden vanhan version lohkoviittauksiin, ja kaikki eroavaisuudet kertovat mihin segmentteihin kuolleita sivujen määrä muuttuu.

Eteenkiertopalautuksen toisella läpikäynnillä käydään läpi segmenttien sisältämät hakemistojen muutoslokilohkot. Erillinen muutosloki tarvitaan varmistamaan että samaan aikaan kahteen eri tiedostojärjestelmän levytietorakenteeseen kohdistuvat muutokset tapahtuvat atomisesti. Esimerkiksi uuden kovan linkin luominen vaatii vähintään kolmen tietorakenteen uuden version kirjoituksen: linkitettävän inoden kovien linkkien määrää kasvatetaan, kohdehakemiston datalohkoihin lisätään uusi hakemistoalkio sekä kohdehakemiston inode päivitetään osoittamaan edellä mainittuun uuteen datalohkoon. Sprite LFS ei sisällä mekanismia joka varmistaisi että kaikki vaadittavat kirjoitukset pystyttäisiin sisällyttämään osaksi samaa segmenttiä, joten erillistä muutoslokia on käytetty tämän ongelman ratkaisemiseen. Muutoslokiin kirjataan kaikki hakemisto-operaatiot eli tiedoston luominen, kovan linkin luominen tai poistaminen sekä uudelleennimeäminen, ja muutoslokilohko kirjoittaan aina lokiin ennen itse inodejen ja hakemistojen datalohkojen muutoksia. Tästä seuraa sääntö jonka perusteella eteenkiertopalautus voi tunnistaa kesken jääneet hakemisto-operaatiot. Jos hakemistojen muutoslo-

kissa esiintyvän hakemisto-operaation osallisena olleet inodet on kirjoitettu lokiin vastaavan hakemiston muutoslokilohkon jälkeen, on koko operaatio suoritettu loppuun. Muussa tapauksessa operaatio on toteutunut joko vain osittain tai ei ollenkaan, ja tilanteesta riippuen koko hakemisto-operaatio saatetaan loppuun tai perutaan osittain tapahtuneet muutokset.

10 Kirjoittaessa kopioivat tiedostojärjestelmät

Eräs uudempi, niin ikään perinteisistä päälle kirjoittavista tiedostojärjestelmistä poikkeava tekniikka on *kirjoittaessa kopioivat* (*CoW, Copy-on-Write*) tiedostojärjestelmät. Tämä tekniikka tuo pääasiassa mahdollisuuden toteuttaa tiettyjä uusia ominaisuuksia, kuten *tilannevedoksia* tehokkaasti. Merkittävimpiä kirjoittaessa kopioivia tiedostojärjestelmiä on Solaris-käyttöjärjestelmää varten suunniteltu ZFS vuodelta 2005, ja Linuxin vuodesta 2007 lähtien kehitetty Btrfs. Perehdytään seuraavaksi näistä jälkimmäiseen, eli Btrfs:ään.

10.1 Johdanto Btrfs:ään

Btrfs-tiedostojärjestelmän tavoitteena on olla seuraava uuden sukupolven yleiskäyttöinen Linux-tiedostojärjestelmä. Verrattuna Linuxin aiempaan vakiintuneeseen `ext4`-tiedostojärjestelmään Btrfs kilpailee pääasiassa uusilla ominaisuuksilla, joista merkittävimmät ovat tarkistussummat sekä datalle että metadatalle, *tilannevedokset* (*snapshot*), tiedostojen pakkaus ja usean levyn tuki. Copy-on-write-tekniikka on keskeinen useiden edellä mainittujen ominaisuuksien kannalta.

Btrfs:n levytietorakenteet poikkeavat merkittävästi aiemmissa luvuissa nähdyistä perinteisten Unix-tiedostojärjestelmien levytietorakenteista. Superlohkoa lukuun ottamatta kaikki Btrfs-tiedostojärjestelmän metadata sijaitsee levyllä B-puuhun pohjautuvassa tietorakenteessa. B-puun avaimena

käytetään aina tietorakennetta `btrfs_key` [RBM13]:

```
struct btrfs_key {
    uint64_t objectid;
    uint8_t type;
    uint64_t offset;
}
```

B-puun sisältämiä avain-arvo-pareja käytetään oleellisesti yhdistämään `type`-tyyppinen tietue `objectid`-kentän osoittamaan metadatarakenteeseen. Esimerkiksi inoden kohdalla `objectid` on sama kuin inoden numero. Samalle `objectid`:lle voi esiintyä useita samantyyppisiä tietueita, jolloin `offset`-kenttää käytetään määräämään tietueiden järjestys samantyyppisten tietueiden välillä. B-puun sivun kokona käytetään tyypillisesti 4096 tavua, ja alkion arvon täytyy aina mahtua yhden levysivun sisälle.

Jokaista tiedostojärjestelmän inodea ilmaisee B-puussa sijaitseva avain-arvo-pari tyyppiä `INODE_ITEM`. Avaimen `objectid` sisältää inoden numeron ja `offset`-kenttää ei käytetä. Alkion arvo puolestaan sisältää tyypilliset inoden attribuutit, kuten tiedoston koon, käyttöoikeudet ja aikaleimat. Inoden datalohkoihin viitataan lohkopuun sijaan *extent*-tietueilla. Yksi extent-tietue on oleellisesti pari `(alkulohko, pituus)`, jolla voidaan kerralla kuvata useampi tiedoston datalohko perättäisiin fyysisiin lohkoihin levyllä. Esimerkiksi tilanteessa, jossa tiedoston data sijaitsee tuhannella perättäisellä levylohkolla, voidaan tämä esittää yhdellä extent-tietueella. Jokaista inoden extent-tietuetta kohden inodeen liitetään `EXTENT_DATA`-tyyppinen avain-arvo-pari, jossa `offset` ilmaisee kohdan, johon kyseiset lohkot kuuluvat tiedostossa. Pienien tiedostojen sisältö on myös mahdollista säilyttää suoraan B-puun lehtisivun sisällä.

Toisin kuin kaikissa aiemmin nähdyissä tiedostojärjestelmissä, hakemiston sisältämät hakemistoalkiot eivät sijaitse hakemistoinoden datalohkoilla, vaan suoraan B-puussa. Jokaista hakemistoalkoita kohti liitetään hakemistoi-

nodeen `DIR_ITEM`-tyyppinen avain-arvo-pari. Koska Btrfs:n B-puun avaimilla on kiinteä pituus, ei tiedostonimeä voida lisätä osaksi avainta. Sen sijaan tiedostonimestä lasketaan 64-bittinen hajautusarvo, joka sisällytetään avaimen `offset`-kenttään, ja alkion arvo sisältää itse tiedostonimen ja inodenumeron. Tämä mahdollistaa yksittäisen tiedostonimen paikantamisen hakemistosta tehokkaasti yhdellä B-puu-kyselyllä, mutta esimerkiksi tiedostonimien listaus aakkosjärjestyksessä ei ole mahdollista.

Btrfs-tiedostojärjestelmä sisältää useita erityyppisiä B-puita. Kaikki tähän asti mainitut B-puun alkiot sijaitsevat *alitiedostojärjestelmäpuussa* (*subvolume tree*). Muita B-puita käytetään muun muassa tilanvarauksen kirjanpitoon ja tarkistussummille. Mahdollisten laitteiston aiheuttamien bittivirheiden tunnistamiseksi Btrfs laskee tarkistussummat kaikille levylohkoille, mukaanlukien tiedostojen datalohkoille.

10.2 Levyrakenteiden päivitys copy-on-write-tekniikalla

Kuten aiemmin nähtiin, kaikki tiedostojärjestelmäoperaatiot ovat oleellisesti vain muokkauksia tiedostojärjestelmän pohjalla oleviin B-puu-tietorakenteisiin. Muutoksien vieminen levylle palautuukin siis oleellisesti B-puun levyversion päivittämiseen kaatumisturvallisesti. Tähän Btrfs käyttää *kirjoittaessa kopioivaa* (*copy-on-write*) tekniikkaa. Sen perusajatuksena on, että levylle aiemmin kirjoitettuja datalohkoja tai B-puun lohkoja ei ylikirjoiteta niitä muokattaessa, vaan niistä kirjoitetaan aina uudet kopiot tiedostojärjestelmän vapaille lohkoille. Tilanvarauspuuta täytyy päivittää samassa yhteydessä, koska uusille lohkoille täytyy luonnollisesti varata tila ja vanhat lohkot mahdollisesti vapauttaa. Lohkon kirjoituksesta aiheutuvan kopioinnin seurauksena kaikki viittaukset lohkon vanhaan sijaintiin täytyy päivittää viittaamaan lohkon uuteen sijaintiin, mikä aiheuttaa lisää kirjoittaessa kopiointia. Prosessia jatketaan B-puun lehdistä ylöspäin taso kerrallaan, kunnes koko tiedostojärjestelmää kuvaavan B-puun juuresta on kirjoitettu uusi ko-

pio. Lopuksi tiedostojärjestelmän superlohko päivitetään osoittamaan uuteen B-puun juureen, mikä onkin tiedostojärjestelmän ainoa tietorakenne, jota päivitetään päällekirjoituksilla.

10.3 Tilannevedokset

Pääasiallinen kirjoittaessa kopioivien tiedostojärjestelmien etu on *tilannevedosten* (*snapshot*) tehokas toteutus. Tilannevedosta tehdessä otetaan koko tiedostojärjestelmän senhetkisestä tilasta ikään kuin kopio, jota voidaan myöhemmin tarkastella ja käsitellä. Koko tiedostojärjestelmän kopioimiseen verrattuna operaatio tapahtuu atomisesti ja välittömästi eikä vie kaksinkertaista määrää levytilaa. Tyypillinen käyttötapaus tilannevedoksille on varmuuskopiointi. Järjestelmä voidaan esimerkiksi laittaa tekemään tilannevedos tunnin välein, jolloin käyttäjät voivat palauttaa tuoreitakin vahingossa poistettuja tiedostoja jostakin aiemmasta tilannevedoksesta.

Tilannevedosta luodessa kirjataan ylös annettu nimi sekä alitiedostojärjestelmäpuun juuren sisältämän lohkon numero luontihetkellä. Koska Btrfs ei koskaan ylikirjoita mitään käytössä olevaa levytietorakennetta, ei millään tilannevedoksen luonnin jälkeen tehdyllä kirjoituksella ole vaikutusta tilannevedoksen sisältöön. Tiedostojärjestelmän täytyy ainoastaan huolehtia, että tilannevedokseen sisältyviä levylohkoja ei vapauteta niin kauan kuin tilannevedos on olemassa. Koska sama levylohko voi olla osa useaa tilannevedosta, käyttää Btrfs *viitelaskentaa* (*reference counting*) pitämään kirjaa milloin levylohko voidaan vapauttaa.

11 Kirjoitusjärjestyksen hallintaan perustuvat tiedostojärjestelmät (“soft updates”)

Aiemmissa luvuissa esiteltiin tekniikoita, jossa tiedostojärjestelmän konsistenssiuden säilyminen kaatumistilanteessa varmistettiin oleellisesti toteut-

tamalla tiedostojärjestelmään jokin menetelmä metadatapäivitysten tekemiseen atomisesti. Vaihtoehtoisia lähestymistapojakin on kuitenkin käytetty, joista yksi on *pehmeät päivitykset* (*soft updates*) jotka perustuvat metadatakirjoitusten tekemiseen tietyssä järjestyksessä. Tarkastellaankin seuraavaksi pehmeiden päivitysten toteutusta Berkeley Fast Filesystem-tiedostojärjestelmässä [MG99].

11.1 Berkeley Fast Filesystem-tiedostojärjestelmä

Berkeley Fast Filesystem (FFS) on alkuperäiseeen BSD-käyttöjärjestelmään kehitetty varsin varhainen Unix-tiedostojärjestelmä. Koska aiemmin esitelty `ext2` on ottanut vaikutteita FFS-tiedostojärjestelmästä, muistuttavat niiden levytietorakenteet vahvasti toisiaan. Pääpiirteissään jokaiselle `ext2`:n levytietorakenteelle löytyy suora vastine FFS:tä, joten kaatumistilanteella on FFS:lle samanlaiset seuraukset kuin mitä `ext2`:lle nähtiin luvussa 7. Erityisesti `fsck`-tarkastusohjelman ajaminen kaatumistilanteen jälkeen oli tarpeen ennen kuin tiedostojärjestelmää pystyttiin käyttämään. FFS-tiedostojärjestelmän kaatumisturvallisuutta alettiin parantamaan alkaen BSD-käyttöjärjestelmän versiosta 4.4, joka julkaistiin vuonna 1995. Käytettäväksi menetelmäksi valittiin *pehmeät päivitykset* (*soft updates*). Verrattuna muihin kilpaileviin tekniikoihin kuten journalointiin tai lokipohjaisiin tiedostojärjestelmiin, pehmeät päivitykset eivät edellytä mitään muutoksia tai lisäyksiä tiedostojärjestelmän levyformaattiin. Sen sijaan pehmeät päivitykset perustuvat levykirjoitusten tekemiseen aina sellaisessa järjestyksessä, jossa kaatumistilanteet eivät voi aiheuttaa merkittäviä epäkonsistenssiuksia tiedostojärjestelmän metadataan. Tällöin `fsck`-ohjelman ajaminen kaatumistilanteen jälkeen ei ole enää tarpeen.

11.2 Pehmeät päivitykset ja *päivitysriippuvuudet*

Aiemmin aliluvussa 7.2.1 esiteltiin mahdollisia epäkonsistenssiuksia, joita `ext2`-tiedostojärjestelmään voi syntyä mikäli kaatumistilanne tapahtuu kutsun `creat("new", 0755)` aikana. Tarkastellaan nyt miten pehmeitä päivityksiä käyttäen vältytään merkittävimmiltä epäkonsistenssiuksilta kirjoitusjärjestystä muuttamalla.

FFS-tiedostojärjestelmän tapauksessa muokataan samoja levytietorakenteita, mutta pehmeitä päivityksiä käytettäessä levykirjoitusten järjestys voisi olla seuraava:

1. Lohkoryhmän kuvaajat
2. Lohkoryhmän inodebittikartta
3. Inodetaulu
4. Hakemistoalkio

Tässä järjestyksessä tehtynä kaatuminen ei milloinkaan johda tilanteeseen jossa tiedostojärjestelmän käytön jatkaminen ei olisi turvallista. Ainoastaan kohtien 1–2 tai 2–3 välissä tapahtunut kaatuminen johtaa yhden inoden vuotamiseen. Sallittuja kirjoitusjärjestyksiä on yleensä useita, tässä esimerkissä kohdat 1. ja 2. voisivat vaihtaa paikkaa ilman ongelmia. Kriittisintä on, että uuteen inodeen viittaavaa tietorakennetta (eli tässä tapauksessa hakemistoalkiota) ei kirjoiteta levylle ennen kuin itse inoden tietorakenne on alustettu ja merkattu käytössä olevaksi bittikarttaan. Tämäntyyppisiä kirjoitusjärjestystä koskevia rajoituksia, eli "päivitys metadatalohkoon tyyppiä x täytyy kirjoittaa levylle ennen päivitystä metadatalohkoon tyyppiä y", kutsutaan *päivitysriippuvuuksiksi* (*update dependency*). Niin kauan kuin muokkaukset tiedostojärjestelmän levytietorakenteisiin tehdään päivitysriippuvuuksien sallimassa järjestyksessä, eivät kaatumistilanteet aiheuta pahoja epäkonsistenssiuksia.

11.3 Berkeley FFS:n päivitysriippuvuudet

FFS-tiedostojärjestelmälle pätevät seuraavat päivitysriippuvuudet uutta tiedostoa luodessa [MG99]:

- Inode täytyy merkitä varatuksi inodebittikartassa ennen kuin inoden levytietorakenne alustetaan inodetaulussa.
- Inoden levytietorakenne täytyy alustaa ennen kuin hakemistolohkoon lisätään hakemistoalkio viittaamaan uuden tiedoston inodeen.
- Uutta hakemistoa luodessa täytyy lisäksi varata ja alustaa datalohko hakemiston sisällölle.

Uutta kovaa linkkiä luodessa:

- Inodeen viittaavien hakemistoalkioiden lukumäärää täytyy kasvattaa inodessa ennen kuin hakemistolohkoon lisätään uusi hakemistoalkio inodelle.

Tiedostoon lohkoja lisättäessä:

- Datalohkon sisältö täytyy kirjoittaa levylle ennen kuin epäsuoraan lohkoon tai inodeen lisätään viittaus datalohkoon.
- Datalohko täytyy merkitä varatuksi lohkobittikartassa ennen kuin epäsuoraan lohkoon tai inodeen lisätään viittaus datalohkoon.
- Mikäli lohkojen lisääminen vaatisi uusien epäsuorien lohkojen varaamista, sovelletaan kahta edellistä uuden epäsuoran lohkon varaamisessa ja lohkopuuhun liittämisessä.

Kovaa linkkiä poistaessa:

- Hakemistoalkio täytyy poistaa hakemistolohkosta ennen kuin inodeen viittaavien nimien lukumäärää vähennetään inodessa.

Lohkojen poistaminen tiedostosta:

- Lohkoja ei voida vapauttaa lohkobittikartasta ennen kuin viittaukset datalohkoihin ovat nollattu inoderakenteesta tai epäsuorasta lohkosta.
- Lohkoja ei voida uudelleenkäyttää ennen kuin viittaukset datalohkoihin ovat nollattu inoderakenteesta tai epäsuorasta lohkosta.

Tiedostoa poistettaessa:

- Hakemistoalkio täytyy poistaa hakemistolohkosta ennen kuin inoden rakenne nollataan.
- Poistetun tiedoston inodea ei voida vapauttaa inodebittikartasta ennen kuin inoden rakenne on nollattu.
- Poistetun tiedoston inodea ei voida uudelleenkäyttää ennen kuin kaikki siihen viittaavat hakemistoalkiot on poistettu ja hakemistoalkioiden sisältämät hakemistolohkot kirjoitettu levylle.

11.4 Sykliset riippuvuudet

Pehmeiden päivitysten ja levyvälimuistien yhdistelmällä voidaan joutua tilanteeseen jossa tietorakenteiden väliset riippuvuudet muodostavat syklin. Otetaan esimerkiksi lähtötilanne, jossa muuten tyhjässä tiedostojärjestelmässä on pelkästään yksi tyhjä tiedosto `/oldfile` ja sille tehdään seuraavat operaatiot:

```
1 open("/newfile", O_CREAT | O_WRONLY, 0644);
2 unlink("/oldfile");
```

Oletetaan lisäksi että sekä `newfile`:n että `oldfile`:n inodet päätyvät samalle inodelohkolle ja hakemistoalkiot samalle hakemiston lohkolle. Nyt tiedostojärjestelmän osalta ovat voimassa seuraavat päivitysriippuvuudet:

1. inodelohko täytyy kirjoittaa ennen hakemistolohkoa, jotta `newfile`:n hakemistoalkio ei osoita alustamattomaan inodeen.

2. inodebittikartan lohko täytyy kirjoittaa ennen inodelohkoa, jotta `newfile`:n inodea ei alusteta ennen kuin se on varattu bittikartassa.

3. hakemistolohko täytyy kirjoittaa ennen inodelohkoa, jotta `oldfile`:n hakemistoalkio ei osoita vapautettuun inodeen.

4. inodelohko täytyy kirjoittaa ennen inodebittikartan lohkoa, jotta `oldfile`:n inode ei ole samanaikaisesti alustettuna inodetaulussa ja vapaana bittikartassa.

Näistä päivitysriippuvuuksista 1. ja 3. muodostavat syklisen riippuvuuden inodelohkon ja hakemistolohkon välillä ja päivitysriippuvuudet 2. ja 4. muodostavat syklin inodelohkon ja inodebittikarttalohkon välillä. On siis päädytty tilanteeseen, jossa mitään metadatalohkoa ei voida sellaisenaan kirjoittaa levylle ilman että jokin päivitysriippuvuus jää täyttämättä.

Pehmeiden päivitysten ratkaisu sykleihin on väliaikaisesti perua päivitysriippuvuuden aiheuttanut muutos metadatalohkoon ennen sen kirjoittamista levylle. Siis käyttöjärjestelmä voi kirjoittaa levylle minkä tahansa muokatun metadatalohkon, mutta täyttymättömien päivitysriippuvuuksien tapauksessa osa lohkoon tehdyistä muutoksista kumotaan väliaikaisesti. Käyttäen perumisinformaatiota edellisen esimerkin metadatapäivitykset voidaan viedä levylle seuraavassa järjestyksessä:

1. inodelohkosta kirjoitetaan versio, jossa `oldfile`:n inoden poistaminen on kumottu.

2. inodebittikartan lohkosta kirjoitetaan versio, jossa `oldfile`:n inoden vapautus on kumottu.

3. hakemistolohkon lopullinen versio kirjoitetaan.

4. inodelohkon lopullinen versio kirjoitetaan.

5. inodebittikartan lohkon lopullinen versio kirjoitetaan.

11.5 Kaatumistilanteesta palautuminen

Pehmeiden päivityksiä käyttäessä palautumista ei tarvitse tehdä ennen tiedostojärjestelmän liittämistä, sillä kaikki metadatakirjoitukset on tehty järjestyksessä jossa levyrakenteisiin ei synny kriittisiä epäkonsistenssiuksia. Kaatumistilanteen seurauksena tiedostojärjestelmään voi kuitenkin aiheutua seuraavia ei-kriittisiä ongelmia:

- Datalohkoja jotka on merkitty varatuksi lohkobittikartassa, mutta joihin ei ole viittausta mistään inodesta.
- Käytössä olevia inodeja jotka eivät ole yhteydessä hakemistorakenteeseen.
- Käytössä olevia inodeja joihin on todellisuudessa vähemmän viittauksia hakemistoalkioiden kautta kuin mitä inodetietorakenteen laskuri osoittaa.

Mikään näistä ei estä tiedostojärjestelmän käytön jatkamista turvallisesti vaan näkyy käyttäjälle pääasiassa "vuotaneina" resursseina, eli tiedostojärjestelmässä on näennäisesti vähemmän vapaata tilaa tai inodeja kuin siellä pitäisi olla. Ennen pitkää nämä ongelmat täytyy kuitenkin korjata, jotta vuotaneen levytilan osuus ei kasva suuremmaksi jokaisen kaatumistilanteen seurauksena. Tämä on toteutettu mahdollistamalla `fsck`-tarkistusohjelman ajamisen taustalla.

12 Yhteenveto

Tutkielmassa tarkasteltiin kaatumistilanteita Unix-tiedostojärjestelmissä alkaen POSIX-tiedostojärjestelmärajapintojen perusteista. Vaikka tiedosto-

järjestelmää käyttävän sovelluksen toteuttaminen onnistuu aloittelevaltakin ohjelmoijalta, paljastui ettei kaatumisturvallisen ohjelman toteutus olekaan enää yksinkertaista. Nykyiset käyttöjärjestelmät nimittäin hyödyntävät runsaasti välimuisteja sekä tiedostojen datalle että metadatalle, joten tiedostojärjestelmäkutsujen aiheuttamia levykirjoituksia ei tehdä välittömästi vaan joskus myöhemmin taustaoperaationa. Ohjelmoijan täytyy osata ottaa huomioon nämä välimuistien aiheuttamat seuraukset ja käyttää `fsync()`-kutsuja jotta lopputuloksesta tulee kaatumisturvallinen. Tilannetta monimutkaistaa entisestään se, että eri tiedostojärjestelmät käyttäytyvät kaatumistilanteissa eri tavalla. On hyvinkin mahdollista saada vahingossa aikaiseksi ohjelma, joka on samaan aikaan täysin kaatumisturvallinen yhdellä tiedostojärjestelmällä mutta ei toisella. Ei olekaan yllättävää, että olemassaolevista sovelluksista löytyi puutteita kaatumisturvallisuuden osalta.

Myös muutaman tiedostojärjestelmän sisäiseen rakenteeseen perehdyttiin, lähtien liikkeelle perinteisestä mutta yksinkertaisesta `ext2`:sta. Simuloimalla kaatumistilanteita QEMU-virtuaalikoneella nähtiin käytännössä, kuinka `e2fsck`-tarkastusohjelman ajaminen jokaisen kaatumistilanteen jälkeen on ehdottoman tärkeää. Toisaalta pystyttiin myös havainnollistamaan levyvälimuistien vaikutusta `ext2`:n tekemiin levykirjoituksiin. Lopuksi selvitettiin menetelmiä joilla modernimmat tiedostojärjestelmät säilyvät konsistenttina kaatumistilanteiden aikanakin. Tämän toteuttamiseen ei ole mitään yhtä oikeaa tapaa, vaan tutkielmassa tarkasteltiin neljää erilaista tekniikkaa. Vaikka menetelmien välillä olikin eroavaisuuksia, niistä kolme neljästä perustui pohjimmiltaan tiedostojärjestelmän metadatan transaktionaaliseen päivittämiseen eivätkä siten juurikaan eroa kaatumisturvallisuuden kannalta. Uusien tiedostojärjestelmien kehittelyn taustalla osoittautuikin usein olevan muutkin syyt kuin kaatumisturvallisuuden parantaminen, kuten Sprite LFS:n tapauksessa paremman suorituskyvyn tavoittelu ja Btrfs:n tapauksessa uusien ominaisuuksien tuominen tiedostojärjestelmään.

Lähteet

[BC05] Bovet, Daniel P ja Cesati, Marco: *Understanding the Linux Kernel: from I/O ports to process management.* O'Reilly Media, Inc., 2005.

[CPADAD13] Chidambaram, Vijay, Pillai, Thanumalayan Sankaranarayana, Arpaci-Dusseau, Andrea C. ja Arpaci-Dusseau, Remzi H.: *Optimistic Crash Consistency.* Teoksessa *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, sivut 228–243, New York, NY, USA, 2013. ACM, ISBN 978-1-4503-2388-8. `http://doi.acm.org/10.1145/2517349.2522726`.

[CTT95] Card, Rémy, Ts'o, Theodore ja Tweedie, Stephen: *Design and Implementation of the Second Extended Filesystem.* 1995. `http://web.mit.edu/tytso/www/linux/ext2intro.html`.

[DCW16] Dam, T. Q., Cheon, S. ja Won, Y.: *On the IO Characteristics of the SQLite Transactions.* Teoksessa *2016 IEEE/ACM International Conference on Mobile Software Engineering and Systems (MOBILESoft)*, sivut 214–224, May 2016.

[EN10] Elmasri, Ramez ja Navathe, Shamkant: *Fundamentals of Database Systems.* Addison-Wesley Publishing Company, USA, 6th painos, 2010, ISBN 0136086209, 9780136086208.

[Gia98] Giampaolo, Dominic: *Practical File System Design with the Be File System.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st painos, 1998, ISBN 1558604979.

[GRADAD08] Gunawi, Haryadi S., Rajimwale, Abhishek, Arpaci-Dusseau, Andrea C. ja Arpaci-Dusseau, Remzi H.: *SQCK: A Declarative File System Checker.* Teoksessa *Proceedings of the 8th*

USENIX Conference on Operating Systems Design and Implementation, OSDI'08, sivut 131–146, Berkeley, CA, USA, 2008. USENIX Association. http://dl.acm.org/citation.cfm?id=1855741.1855751.

[IEE18] IEEE: *IEEE Std 1003.1-2017, IEEE Standard for Information Technology — Portable Operating System Interface (POSIX) Base Definitions, Issue 7*. IEEE, New York, NY, USA, 2018, ISBN 978-1-5044-4542-9.

[Loc06] Locke, Douglass C.: *POSIX and Linux Application Compatibility Design Rules*, joulukuu 2006. https://collaboration.opengroup.org/platform/single_unix_specification/documents.php?action=show&dcat=&gdid=13450, Draft, Version 1.0.

[MCB+07] Mathur, Avantika, Cao, Mingming, Bhattacharya, Suparna, Dilger, Andreas, Tomas, Alex ja Vivier, Laurent: *The new ext4 filesystem: current status and future plans.* Teoksessa *Proceedings of the Linux symposium*, nide 2, sivut 21–33, 2007.

[MG99] McKusick, Marshall Kirk ja Ganger, Gregory R.: *Soft Updates: A Technique for Eliminating Most Synchronous Writes in the Fast Filesystem.* Teoksessa *Proceedings of the Annual Conference on USENIX Annual Technical Conference*, ATEC '99, sivut 24–24, Berkeley, CA, USA, 1999. USENIX Association. http://dl.acm.org/citation.cfm?id=1268708.1268732.

[MHL+92] Mohan, C., Haderle, Don, Lindsay, Bruce, Pirahesh, Hamid ja Schwarz, Peter: *ARIES: A Transaction Recovery Method Supporting Fine-granularity Locking and Partial Rollbacks Using Write-ahead Logging.* ACM Trans. Database Syst.,

17(1):94–162, maaliskuu 1992, ISSN 0362-5915. http://doi.acm.org/10.1145/128765.128770.

[PADAD05] Prabhakaran, Vijayan, Arpaci-Dusseau, Andrea C. ja Arpaci-Dusseau, Remzi H.: *Analysis and Evolution of Journaling File Systems*. Teoksessa *Proceedings of the Annual Conference on USENIX Annual Technical Conference*, ATEC '05, sivut 8–8, Berkeley, CA, USA, 2005. USENIX Association. http://dl.acm.org/citation.cfm?id=1247360.1247368.

[Pat03] Pate, Steve D.: *UNIX Filesystems: Evolution, Design, and Implementation*. Wiley Publishing, Inc., Indianapolis, Indiana, USA, 1st painos, 2003, ISBN 0-471-16483-6.

[PCA+14] Pillai, Thanumalayan Sankaranarayana, Chidambaram, Vijay, Alagappan, Ramnatthan, Al-Kiswany, Samer, Arpaci-Dusseau, Andrea C. ja Arpaci-Dusseau, Remzi H.: *All File Systems Are Not Created Equal: On the Complexity of Crafting Crash-consistent Applications*. Teoksessa *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*, OSDI'14, sivut 433–448, Berkeley, CA, USA, 2014. USENIX Association, ISBN 978-1-931971-16-4. http://dl.acm.org/citation.cfm?id=2685048.2685082.

[RBM13] Rodeh, Ohad, Bacik, Josef ja Mason, Chris: *BTRFS: The Linux B-Tree Filesystem*. Trans. Storage, 9(3):9:1–9:32, elokuu 2013, ISSN 1553-3077. http://doi.acm.org/10.1145/2501620.2501623.

[Ros92] Rosenblum, Mendel: *The Design and Implementation of a Log-structured File System*. väitöskirja, EECS Department,

University of California, Berkeley, Jun 1992. http://www2.eecs.berkeley.edu/Pubs/TechRpts/1992/6267.html.

[RT74] Ritchie, Dennis M. ja Thompson, Ken: *The UNIX Time-sharing System*. Commun. ACM, 17(7):365–375, heinäkuu 1974, ISSN 0001-0782. http://doi.acm.org/10.1145/361011.361061.

[Sal94] Salus, Peter H.: *A Quarter Century of UNIX*. ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 1994, ISBN 0-201-54777-5.

[Twe98] Tweedie, Stephen C.: *Journaling the Linux ext2fs Filesystem*. Teoksessa *In LinuxExpo'98: Proceedings of The 4th Annual Linux Expo*, 1998. http://e2fsprogs.sourceforge.net/journal-design.pdf.