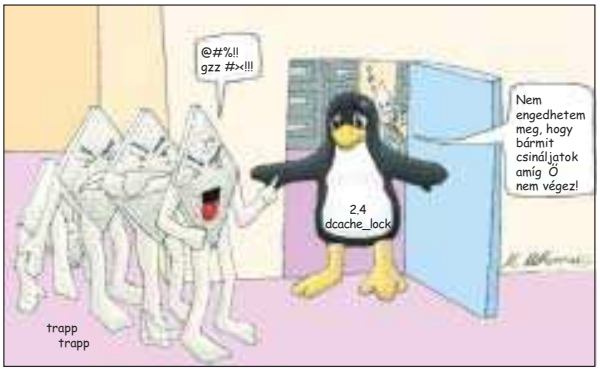
A dcache méretezése RCU-alapokon

A Linux fájlnévkeresési rendszerének újraszervezése sokat számíthat a komolyabb kiszolgálók méretezésekor.

Nemrég jelent meg 2.6-os Linux-rendszermagban az általában olvasott adatszerkezetekre kiélezett összehangolási módszer, az RCU (read-copy update, azaz olvasás–másolás–frissítés). Ebben a cikkben azt mutatjuk be, miképpen segíti az RCU a Linux könyvtárbejegyzés-gyorstárának (dcache) a méretezhetőségét. Ha a dolgok hátterére is kíváncsiak vagyunk, olvassuk el a „Using RCU in the Linux 2.5 Kernel” című írást a Linux Journal 2003. októberi számában.

Linux könyvtárbejegyzés-gyorstár

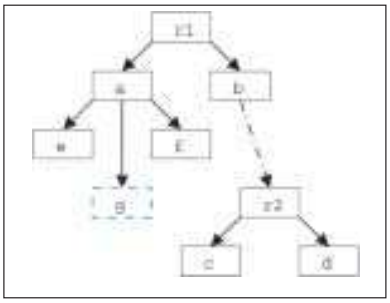
A Linux `dcache` a fájlrendszer hierarchiájának egy részét a memóriában kezeli. Ez a másolat költséges lemezműveletek nélkül is lehetővé teszi az útvonalkereséseket, amellyel nagymértékben növeli a fájlműveletek hatékonyságát.
A Linux-rendszermag a `mount` és `umount` műveletek megkönnyítése érdekében a befűzési fáról (mount tree) is tárol egy másolatot `struct vfsmount` szerkezetekben.
De ha a Linux 2.4 `dcache` ilyen nagyszerű, akkor miért kell megváltoztatni? A 2.4-es `dcache` nagy gondja, hogy globális `dcache_lock` zárat használ. Ez a zár kis rendszereken gyorstárvonalon fürgeséget, nagy rendszereken azonban a méretezhetőség komoly gátját jelenti, mint azt a *képen* is megfigyelhetjük.



Tux teszi a dolgát

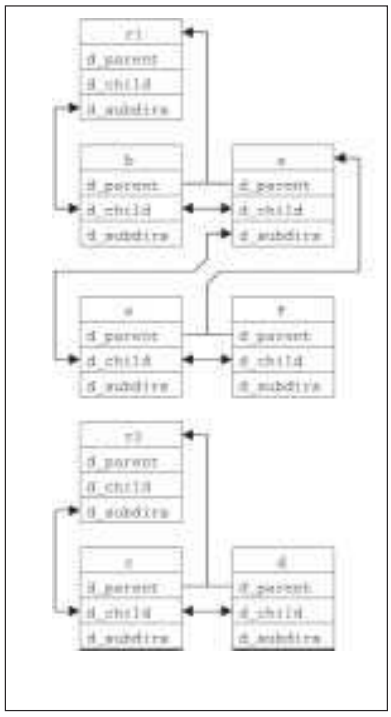
A dcache szemléletes bemutatása

Ebben a részben a cikkben később előforduló RCU-vonatkozású `dcache`-változásokkal ismerkedhetünk meg. A további részletekre kíváncsi olvasóinknak a forráskód átböngészését javasoljuk.
Írásunkban a *1. ábrán* bemutatott fájlrendszert fogjuk példaként felhasználni, amelyen két, `r1` és `r2` gyökérrel bíró befűzött fájlrendszert találunk. Mint azt a szaggatott nyíl mutatja, a második fájlrendszert a ***b*** könyvtár alá fűztük be. A ***g*** nevű állományt mostanában nem értük el, ezért nem is szerepel a `dcache`-ben – ezt szaggatott kék kerettel jelöltük.
A `dcache*` alrendszer a fájlrendszerfa több nézetét is kezelni tudja egyszerre. A *2. ábra* a könyvtárszerkezet megjelenítését mutatja be – minden egyes `dentry` egy-egy könyvtárnak felel meg, amely kétszeresen befűzött, `d_subdirs` mezővel kezdődő, körkörös listát tárol. Ez a lista fut végig a gyermek `dentry`-bejegyzések `d_child` mezőin. Minden gyermek `d_parent` mutatója értelemszerűen a szülőjére mutat. A befűzési pont (`dentry b`) nem közvetlenül a befűzött fájlrendszerre hivatkozik; ehelyett a befűzési hely `d_mounted` kapcsolóját állítjuk be, majd a `dcache` kikeresi a befűzött fájlrendszert a `mount_hashtable` táblázatából. Ezt a folyamatot a későbbiek folyamán ismertetjük.

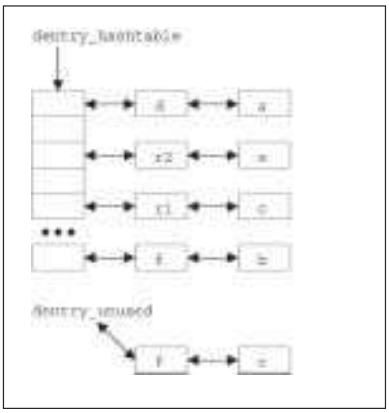


1. ábra Fájlrendszerfa-példa

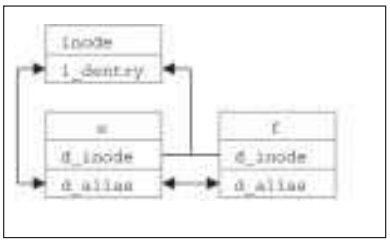
Bár a `d_subdirs` listákat közvetlenül is kereshetnénk, nagy könyvtárak esetében ez meglehetősen lassú megoldás lenne. Ezért az `__d_lookup()` inkább egy hasítótáblát készít a könyvtárak `dentry` mutatóihoz és a gyermekek nevéhez. Ezt a `dentry_hashtable`-t használja fel a megfelelő `dentry`-bejegyzéshez. A táblát a *3. ábra* mutatja be a `dentry_unused` elemmel kezdődő LRU listával együtt.
Az LRU lista minden `dentry` eleme általában a hasítótáblában (hash table) is benne van; kivételt képeznek azok az esetek, amelyekben a szülőkönyvtárak ideiglenesek, ilyen például az NFS és a hozzá hasonló osztott fájlrendszerek.
Minden `dentry` a `d_inode` mutató segítségével hivatkozik a hozzá tartozó fájlleíróra (inode). A `d_inode` mutató negatív `dentry`-bejegyzéseknél lehet `NULL` is, ami a hiányzó



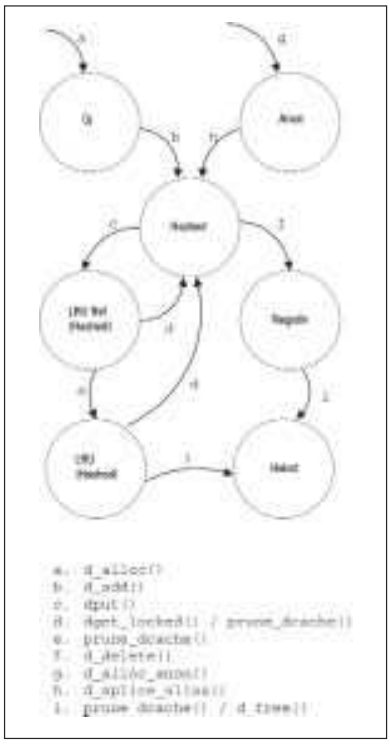
2. ábra Fájlrendszerfa-példánk dcache-megvalósítása



3. ábra dentry hasítótábla



4. ábra
Közvetlen befűzésű álnévláncok

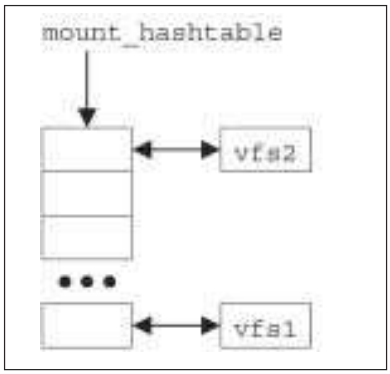


5. ábra A dentry állapotdiagram

fájlleíróra utal. Negatív `dentry` jöhet létre például akkor, amikor a fájlrendszer törli a `dentry` fájlját, vagy ha valaki zárolni próbál egy nemlétező állományt. A negatív `dentry` elemek felgyorsíthatják a rendszer teljesítményét, hiszen így a nemlétező fájlok ismételt elérési kísérlete során nem kell folyton az alsóbb szintű fájlrendszerhez fordulni. Ehhez hasonló módon a közvetlen befűzések is azonos `dentry`-bejegyzéseken osztozkodnak, mint azt az *4. ábrán* láthatjuk. A *5. ábrán* egy magas szintű `dentry` állapotdiagramot láthatunk. A normál `dentry` életciklus a következő:

1. A `d_alloc()` lefoglalja az új `dentry`-t az újonnan hivatkozott állományhoz, ezzel belépünk a ***New*** (új) állapotba.
2. A `d_add()` nevet és fájlleírót rendel az új bejegyzéshez, ezzel beléptünk a ***Hashed*** állapotba.
3. Miután végzett a fájllal, a `d_put()` felveszi a `dentry`-t a LRU listába és beállítja a `DCACHE_REFERENCED` bitet a `d_vfs_flags` mezőben – így jutunk el az ***LRU Ref (Hashed)*** állapotba.
4. Amennyiben az állományra az ***LRU Ref (Hashed)*** állapotban ismét hivatkoznak, a `dget_locked()` – amelyet általában a `d_lookup()` hív meg – használatra jelöli ki. Ha a következő `prune_dcache()` híváskor még mindig használatban van, eltávolítódik az LRU listáról és ismét visszakerül a ***Hashed*** állapotba.
5. Egyébként a `prune_dcache()` eltávolítja a `DCACHE_REFERENCED` bitet a `d_vfs_flags` mezőből, és ilyen módon eljutunk az ***LRU (Hashed)*** állapotba.
6. Akárcsak az előbbi esetben, ha a fájlra ismét hivatkoznak, a `dget_locked()` használatra jelöli be, hogy azután a `prune_dcache()` eltávolíthassa a LRU listáról és ismét ***Hashed*** állapotba kerüljön át.
7. Egyébként a második egymást követő `prune_dcache()` hívás után a `d_free()` függvényt hívjuk meg a `prune_one_dentry()` függvényen keresztül, és végül ***Dead (halott)*** állapotba kerül.

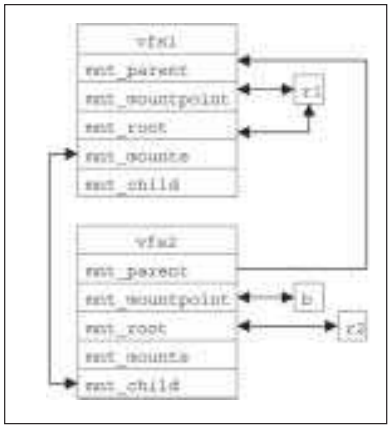
A *6. ábrából* kiolvasható egyéb útvonalak is elképzelhetőek. Például amikor egy osztott fájlrendszer gyorstárazott fájlmutatót alakít új `dentry`-bejegyzéssé, az `_alloc_anon()` függvény segítségével lefoglal egy `dentry`-t, miközben az objektum szülője e `dentry` gyorstárban már nem is létezik. Hasonlóképpen, amikor a `d_delete()` segítségével kitöröljük az adott `dentry` alatti állományt vagy könyvtárat, a `dentry`-t ***Negatív*** állapotba helyezzük. Ez a következő lépésben halott (***Dead***) állapotba jut. A 6. ábra a `mount_hashtable` adatszerkezetet mutatja be, amelyet a befűzési pont `dentry` és a befűzött fájlrendszer `struct vfsmount` szerkezetének összerendelésére használunk fel. Ez az összerendelés a mutatókat és a befűzési pont `dentry`-ket, valamint a mutatót és a befűzési pontot tartalmazó fájlrendszer `struct vfsmount` szerkezetét szervezi (pontosabban hasheli) össze. A `dentry` mutató és a `struct`



6. ábra Befűzési pontok átvitele

1. lista Zárnélküli útvonal–név–szakasz–keresés

```
1  struct dentry *
2  __d_lookup(struct dentry * parent,
3             struct qstr * name)
4  {
5    unsigned int len = name->len;
6    unsigned int hash = name->hash;
7    const unsigned char *str = name->name;
8    struct hlist_head *head =
     ➥d_hash(parent,hash);
9    struct dentry *found = NULL;
10   struct hlist_node *node;
11
12   rcu_read_lock();
13   hlist_for_each (node, head) {
14     struct dentry *dentry;
15     unsigned long move_count;
16     struct qstr * qstr;
17
18     smp_read_barrier_depends();
19     dentry = hlist_entry(node, struct dentry,
20                          d_hash);
21     if (unlikely(dentry->d_bucket != head))
22       break;
23     move_count = dentry->d_move_count;
24     smp_rmb();
25     if (dentry->d_name.hash != hash)
26       continue;
27     if (dentry->d_parent != parent)
28       continue;
29     qstr = dentry->d_qstr;
30     smp_read_barrier_depends();
31     if (parent->d_op &&
32         parent->d_op->d_compare) {
33       if (parent->d_op->d_compare(parent, qstr,
34                                   name))
35         continue;
36     } else {
37       if (qstr->len != len)
38         continue;
39       if (memcmp(qstr->name, str, len))
40         continue;
41     }
42     spin_lock(&dentry->d_lock);
43     /*
44      * Ha a dentryt áthelyezték,
        ➥a keresés sikertelen
45      */
46     if (likely(move_count ==
47                dentry->d_move_count)) {
48       if (!d_unhashed(dentry)) {
49         atomic_inc(&dentry->d_count);
50         found = dentry;
51       }
52     }
53     spin_unlock(&dentry->d_lock);
54     break;
55   }
56   rcu_read_unlock();
57   return found;
58 }
```



7. ábra VFS csatolási fa

vfsmount ilyen formán történő összekapcsolása lehetővé teszi, hogy egy kicsit elegánsabban kezeljük az azonos befűzési ponthoz rendelt befűzéseket.

A 2. ábrán bemutatott példafájlrendszer a *7. ábrán* bemutatott `struct vfsmount` szerkezetet eredményezné. A `vfs1` szerkezetben az `mnt_mountpoint` és az `mnt_root` helyen egyarán az `r1` gyökér `dentry`-t találjuk, mivel ez lesz egyben a fájlrendszer legvégső gyökere is. A `vfs2` szerkezetben a `b` `dentry` `mnt_mountpoint` hivatkozásként szerepel, az `r2` pedig mint a hozzá tartozó `mnt_root`. Így amikor a `mount_hashtable` keresés visszaadja a `vfs2`-höz tartozó mutatót, az `mnt_root` mező alapján gyorsan beazonosíthatjuk a befűzött fájlrendszer gyökerét. A befűzött fájlrendszerek teljes alakja az ***mnt_mount/mnt_child*** listákból látható. Ezeket a listákat a `copy_tree()` és néhány további függvény használja, miközben visszacsatolt befűzést készít (loopback mount), amelyhez a teljes útvonalnévtér adott alfájában befűzött, valamennyi fájlrendszer áttekintése szükséges.

RCU alkalmazása a dcache-en

A `dcache` teljes párhuzamosítása meglehetősen összetett feladat volna, és úgy gondolták, hogy túlságosan kockázatos lenne beletenni a 2.5 változatba. A 2.6-os `dcache` mindössze az első lépést tette meg az RCU útján; a 2.7-es feladata lesz bármiféle zár foglalása nélkül a teljes utat bejárni.

Az útvonalszakasz-kereséseket az *1. listában* bemutatott `__d_lookup()` függvény végzi. Az `__d_lookup()` függvényt a szülőkönyvtár `dentry`-bejegyzését jelölő mutatóval és a keresendő névvel kell meghívni. A nevet `qstr` szerkezetben adjuk át, amelyben a karaktersorozatot megadó mutatót, a szöveg hosszát, a `dcache` hasítótáblához használható előfeldolgozott hasítóértéket, valamint – amennyiben szükséges – magát a nevet találjuk.

Az 5–7. sor a `struct qstr` szerkezetet értelmezi. A 8. sor a név és a szülő `dentry` mutató kombinációját rendeli egy általános `dcache` hasítótáblához, kinyerve belőle a megfelelő hasítólánchoz tartozó mutatót.

A 12. és 56. sor jelöli a kód RCU-védett szakaszait, kikapcsolva az időosztást (preemptive) a `CONFIG_PREEMPT`

2. lista Útvonal–név–szakaszkeresés és -átnevezés versenyhelyzetének feloldása

```
 1 struct dentry *
 2 d_lookup(struct dentry * parent,
 3          struct qstr * name)
 4 {
 5   struct dentry * dentry = NULL;
 6   unsigned long seq;
 7
 8   do {
 9     seq = read_seqbegin(&rename_lock);
10     dentry = __d_lookup(parent, name);
11     if (dentry)
12       break;
13   } while (read_seqretry(&rename_lock, seq));
14   return dentry;
15 }
```

3. lista A dentry-szerkezetek késleltetett felszabadítása

```
1 static void d_free(struct dentry *dentry)
2 {
3   if (dentry->d_op && dentry->d_op->d_release)
4     dentry->d_op->d_release(dentry);
5   call_rcu(&dentry->d_rcu, d_callback,
    ➥dentry);
6 }
```

4. lista A dentry-bejegyzésekhez tartozó RCU callback függvény

```
1 static void d_callback(void *arg)
2 {
3   struct dentry * dentry = (struct dentry
    ➥*)arg;
4
5   if (dname_external(dentry)) {
6     kfree(dentry->d_qstr);
7   }
8   kmem_cache_free(dentry_cache, dentry);
9 }
```

rendszermagokban, csakúgy, mint a Linux Journal 2003. októberi számában olvasható „Using RCU in the Linux 2.5 Kernel” cikkben bemutatott Reader-Writer-Lock/RCU esetében. A 13–55. sor a kiválasztott hasítólánc elemein lépked végig megfelelő `dentry` után kutatva. A 18. sor egy, csak a DEC Alpha gépeken lényeges memóriakorlát kezelését végzi. Egyéb processzorokon a mutató visszakódolása által létrehozott adatfüggőség önmagában elegendő, így ezeken a processzorokon a 18. sor nem készít kódot. Minthogy ez a keresés nem kér zárat, versenyhelyzetbe kerülhet a `rename` (átevezés) rendszerhívással. Egy ilyen rendszerhívás a `dentry`-t egy másik hasítóláncba mozgathatja át, magával rántva az egész keresést. A 21. és 22. sor az ilyen versenyhelyzetek létét ellenőrzi, de önmagában még nem elegendő. Ezért a 23. sor megjegyzi azt a pillanatot (pillanatfelvételt készít), amikor a jelenlegi `dentry` a `dcache d_move()` függvény átnevezési folyamata alá került, így később könnyen meg lehet állapítani, hogy volt-e olyan átnevezés, amely az útvonalsétával versenyhelyzetbe került volna. A 24. sor memóriagátja arra szolgál, hogy a pillanatfelvételt se a fordító, se a CPU ne rendezhesse újra.

A 25–28. sor a szülő `dentry`-t és a név-hasht ellenőrzi. Amennyiben bármelyik hibásnak bizonyulna, ez a `dentry` nem lehet a keresésünk alapja. A 29–41. sor végzi a teljes névösszehasonlítást a DEC Alpha gépekhez szánt memóriagáttal (30. sor). A 33. sorban láthatjuk, hogy alkalmazhatunk-e fájlrendszerfüggő névösszehasonlító függvényeket is (például a kis- és nagybetűfüggetlen fájlrendszerekhez). Amint a végrehajtás eléri a 42. sort, megtaláltuk a megfelelő névhez tartozó gyermek `dentry` bejegyzést. Most már elkérhetjük a gyermek `dentry` zárját (42. sor). Minthogy valamennyi `dentry`-bejegyzésen külön zárat alkalmazunk, ezeknek az egyedi záraknak a versengési szintje lényegesen alacsonyabb, mint az eredeti `dcache_lock` rendszeré volt. Sajnálatos módon az élet nem fenékig tejfel, ugyanis például a gyökér `dentry` lezárása továbbra is versengéshez vezet – de ezzel majd később foglalkozunk.

Elképzelhető, hogy a gyermek `dentry`-t már átnevezték azóta, hogy a 23. sorban meghívtuk a `d_move_count` pillanatkép készítését. Ezért a 46–47. sor a `d_move_count` és a pillanatkép értékét hasonlítja össze. Ha az ellenőrzés egyezést mutat, a gyermek `dentry`-t nem nevezték át a keresés ideje alatt, és a 48–51. sor növeli a számlálót – de csak akkor, ha a bejegyzés még mindig hashelt.

Az 53. sor elengedi a gyermek `dentry` zárát, az 54. pedig kilép a hasítólánc keresési ciklusából. Az 57. sor sikeres keresés esetén a mutatót, sikertelen esetben pedig a `NULL` értéket adja vissza a gyermek `dentry`-nek.

Az `__d_lookup()` függvény sikertelensége nem feltétlenül jelenti azt, hogy a felhasználói folyamat is sikertelenséget jelző üzenetet kap. A fájl ettől még létezhet, csak éppen még nincsen betöltve a gyorstárba.

A függvény azonban nem oltalmaz meg bennünket valamennyi átnevezési verseny kockázatától. Versenyhelyzet jöhet létre ugyanis abban az esetben is, amikor a `dcache` listák (list) helyett `hlist` szerkezetet használ a `dcache` hasítóláncokhoz. A `hlist` listákat memóriamegtakarítás céljából alkalmazza, minthogy a `hlist` listák kettő helyett csak egyetlen mutatót tárolnak a lista fejlécében. Ez egyúttal azt is jelenti, hogy a listákkal ellentétben a `hlist` nem körkörös. Ezért előfordulhat, hogy egy adott `dentry`-t úgy nevezünk át, hogy az egy korábban üres `dcache` hasítóláncba kerül. Ha ez éppen az adott időben történik, az `__d_lookup()` függvény (helytelen módon) keresési hibát ad vissza.

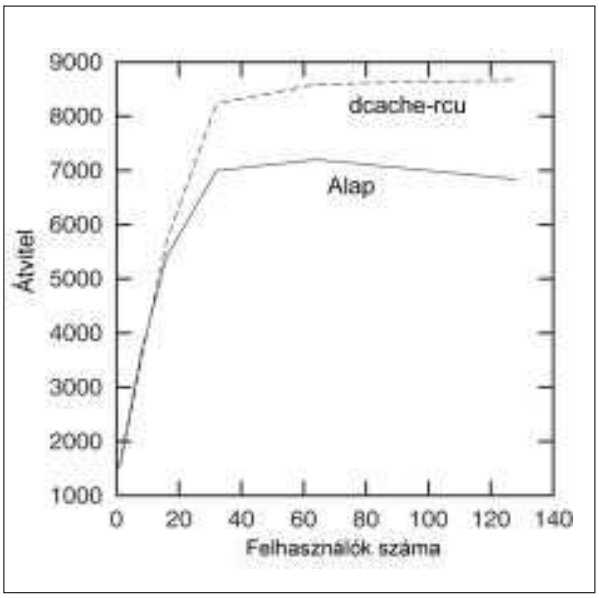
A hibásan visszaadott keresési eredménytelenséget a felsőbb szintű `d_lookup()` függvény kezeli, amit a *2. listán* mutatunk be. A versengő átnevezéseket a `read_seqretry()` függvény érzékeli a 13. sorban. Mivel a gondot okozó eset kizárólag hamis hibajelentést küldhet, az ellenőrzést elegendő csak az `__d_lookup()` `NULL` visszatérési értékeire elvégeznünk.

5. lista A dentry-bejegyzések átnevezése

```
 1 void
 2 d_move(struct dentry *dentry,
 3         struct dentry *target)
 4 {
 5   spin_lock(&dcache_lock);
 6   write_seqlock(&rename_lock);
 7   if (target < dentry) {
 8     spin_lock(&target->d_lock);
 9     spin_lock(&dentry->d_lock);
10   } else {
11     spin_lock(&dentry->d_lock);
12     spin_lock(&target->d_lock);
13   }
14   if (dentry->d_vfs_flags & DCACHE_UNHASHED)
15     goto already_unhashed;
16   if (dentry->d_bucket != target->d_bucket) {
17     hlist_del_rcu(&dentry->d_hash);
18 already_unhashed:
19     dentry->d_bucket = target->d_bucket;
20     hlist_add_head_rcu(&dentry->d_hash,
21                        target->d_bucket);
22     dentry->d_vfs_flags &= ~DCACHE_UNHASHED;
23   }
24   __d_drop(target);
25   list_del(&dentry->d_child);
26   list_del(&target->d_child);
27   switch_names(dentry, target);
28   smp_wmb();
29   do_switch(dentry->d_name.len,
30             target->d_name.len);
31   do_switch(dentry->d_name.hash,
32             target->d_name.hash);
33   if (IS_ROOT(dentry)) {
34     dentry->d_parent = target->d_parent;
35     target->d_parent = target;
36     INIT_LIST_HEAD(&target->d_child);
37   } else {
38     do_switch(dentry->d_parent,
39               target->d_parent);
40     list_add(&target->d_child,
41        &target->d_parent->d_subdirs);
42   }
43   list_add(&dentry->d_child,
44      &dentry->d_parent->d_subdirs);
45   dentry->d_move_count++;
46   spin_unlock(&target->d_lock);
47   spin_unlock(&dentry->d_lock);
48   write_sequnlock(&rename_lock);
49   spin_unlock(&dcache_lock);
50 }
```

Halasztott felszabadítás

A `d_free()` függvénynek egy megadott türelmi ideig el kell halasztania az adott `dentry`-hez tartozó területek felszabadítását, mivel azon a bejegyzésen éppen tetszőleges



8. ábra Többfelhasználós teljesítménymérés eredménye

számú folyamatban lévő útvonalséta tarthat mutatókat. A `d_free()` függvényben alkalmazott halasztást a *3. listában* tekinthetjük meg, ahol az 5. sor a `call_rcu()` primitív segítségével késlelteti a `d_callback()` pusztítóműveleteket, amíg egy adott türelmi idő le nem telik. A `d_callback()` függvény listáját a *4. lista* tartalmazza; ez – amennyiben szükséges – a külön tárolt fő részeket (5–7. sor) egyszerűen felszabadítja, majd a 8. sorban magát a `dentry`-t is felszabadítja.

Átnevezés

A `d_move()` függvény valósítja meg az (átnevezést végző) `rename` rendszerhívás `dentry`-re jellemző elemeit, ezt mutatja be *5. listánk*. Az 5. sor kizár minden, esetlegesen `dcache` frissítést végező egyéb folyamatot, majd a 6. sor lehetővé teszi a `d_lookup()` függvénynek, hogy megállapítsa, valóban versenyzett-e valamilyen átnevezéssel.
A 7–13. sor az átnevezés alatt lévő állomány és annak célja számára foglal le `dentry` elemenként egy-egy zárat, mégpedig a holtzárak (deadlock) elkerülése érdekében, cím szerint rendezve. A 14–17. sor eltávolítja a bejegyzést a `dcache` hasítótáblában elfoglalt régi helyéről, amennyiben az korábban nem törlődött.
A 19. sor frissíti a `dentry` bejegyzést, hogy az az új hasítógyűjteményre mutasson; a 20–21. sor felveszi a `dentry`-t a hozzá tartozó cél-hasítógyűjteménybe, végül a 22. sor frissíti a jelzőket (flag), mutatván, hogy a `dentry` a `dcache` hasítótáblában található meg. A 24. sor eltávolítja az (általunk átnevezett) cél-`dentry`-t a `dcache` hasítótáblából, a 25–26. sor pedig a mozgató és a cél-`dentry`-ket választják el a régi szülőjüktől.
A 27. sor megváltoztatja a `dentry` nevét, a 28. sor pedig kikényszeríti az írás újrarendezését. A névváltoztatás megoldása nem olyan egyszerű, mivel a rövid neveket magában a `dentry`-ben tároljuk, a hosszabb nevek viszont külön lefoglalt memóriahelyre kerülnek. A 29–32. sor frissíti a névhosszakat és hasítóértékeket. A 33–44. sor

a dentry-t csatlakoztatja annak új szülőjéhez. Végül a 45. sor frissíti a d_move_count értéket, így az __d_lookup() felfigyelhet a versenyhelyzetre; végül a 46–49. sor kioldja a zárat.
Elméletben tetszőlegesen szerencsétlen keresést készíthetnénk, a dentry értékeiket mindig azonos könyvtárban és azonos hasítóláncban hagyó, gondosan megtervezett átnevezési műveletek folyamatos sorozatával. Az egyik lehetőség, amikor ilyesmi bekövetkezhet, ha a hasítólánc utolsó elemét keressük, miközben az utolsó előtti elemet folyamatosan átnevezzük (és az ekként mindig a lista elejére kerül), épp mire a keresés odaérne. A gyakorlatban azonban a dcache hasítóláncok rövidek, az átnevezés pedig lassú. Amennyiben az ilyen elszállások gondot jelentenek, elképzelhető, hogy ki kell egészíteni a kódot, és útvonalbejárási sikertelenség esetén az átnevezést le kell állítani. Egy másik lehetséges megoldás szerint teljes egészében el kell távolítani globális hasítótáblát, és helyette a d_subdirs listákat szükséges olyan módon megváltoztatni, hogy a nagy könyvtárakat elegánsan kezeljék.

Teljesítmény és összetettség összehasonlítása

Bár a dcache-kódban végzett változtatás hozzávetőlegesen kicsi volt, igen távoli következményei is vannak a rendszermagban, ugyanis ez idáig nem létezett a dcache-sel együttműködő, jól körülhatárolt fájlrendszer-API. Ennek eredményeképpen rengeteg hiba bukkant fel a Linux 2.5-ös rendszermagban, mivel a fájlrendszerkódolók a dcache-t hagyományos stílusban, közvetlenül próbálták meg módosítani. Tekintve, hogy immár létezik egy formálisabb API, remélhetjük, hogy a következő változtatások már kevésbé lesznek sokkolóak.
A *8. ábrán* egy többfelhasználós tesztprogram teljesítményét láthatjuk az RCU-folttal ellátott Linux 2.5.59-es rendszermag és a folt nélküli mag esetében. A próbákat 16 processzoros NUMA-Q rendszeren futtattuk, 700 MHz-es Pentium III Intel Xeon processzorok felhasználásával, 1 MB L2 gyorstárral és 16 GB memóriával.
A dcache_rcu folt alkalmazása a Linux 2.4.17 rendszermagra 2,258-ról 2,530-ra növelte a SPECweb99 (SSL nélküli) átviteli teljesítményét egy nyolcprocesszoros Pentium III Xeon kiszolgálón, ami 12 százalékos gyorsulást jelent. Ugyanezt a foltot alkalmazva a Linux 2.5.40-mm2 rendszermagra a Linux-rendszermag fordítási ideje 47,548 CPU-másodpercről 42,498 CPU-másodpercre esett vissza, ami több mint tízszázalékos gyorsulást jelent. Hasonló próbát futtatva a Linux 2.5.42 rendszermagot használó egyprocesszoros 700 MHz-es Pentium III Xeon gépen nem tapasztaltunk változást. Összefoglalva tehát: a dcache RCU nemcsak, hogy növeli a felső kategóriás gépek méretezhetőségét, de gyakran az alacsonyabb kategóriás gépek teljesítményét is fokozza.

A jövő irányvonalai

Bár a 2.6 dcache rendszer sokkal méretezhetőbb, mint a 2.4-es változat volt, számos dolog még vizsgálatra szorul:

1. A frissítések felett továbbra is a dcache_lock őrködik, így a mélyreható frissítést használó folyamatok nem jól méreteződnek.
2. A globális hasítótábla miatt a gyorstár nem lehet helyi és a frissítési kód a szükségesnél összetettebb lesz. Természetesen az alternatíváknak meg kell tartaniuk az előd vívmányait, többek közt a nagy könyvtárak nagyteljesítményű kezelését.
3. A 2.6-ban lévő dcache-kód minden egyes dentry d_lock spinlock zárját lekérdezi, ami gyorstárvonalon pattogáshoz (bouncing) és atomi műveletvégzéshez (atomic operations) vezethet, különös tekintettel a gyökérkönyvtárra és a munkakönyvtárakra. Még sokat kell gondolkodni egy igazán egyszerű megoldáson, ugyanis az engedélyeknek a dentry-bejegyzésekbe történő átmozgatása túlságosan bonyolultnak bizonyult.
4. Az __d_lookup() és a d_move() közötti versenyhelyzetek feloldására használt kód túlzottan bonyolult.

Izgatottan várjuk, hogy a 2.7 fejlesztései megoldják ezeket a kérdéseket.

Jogi megállapítások

Az itt leírtak a szerző álláspontját tükrözik, és nem feltétlenül egyeznek meg az IBM véleményével. A SPEC és a SPECweb név a Standard Performance Evaluation Corporation bejegyzett védjegye. A teljesítménymérés kizárólag kutatási célokat szolgált és a szabályoktól való következő eltérések miatt nem hasonlítható össze a SPECWeb lapján található eredményekkel:

1. Olyan alkatrészeken futott, amik nem felelnek meg a SPEC-nagyközönség számára elérhetőnek, a gép ugyanis mérnöki mintadarab volt.
2. Az access_log naplót nem tartottuk meg bizonyítékként. Kiírattuk ugyan, de 200 másodpercenként le is töröltük.

A legfrissebb SPECweb99-teljesítménypróbák a
➲ http://www.spec.org lapon érhetők el.

Linux Journal 2003. február, 117. szám

Paul E. McKenney
Kiváló mérnök az IBM-nél. Régebb óta dolgozik az SMP és NUMA algoritmusokon, mint hogy érdemesnek tartaná megemlíteni. Ezt megelőzően a packet-radio és internetes protokollok témájával foglalkozott (jóval azelőtt, hogy az internet népszerű lett). Hobbijai közé tartozik a futás és a szokásos „ház–asszony–gyerekek" álom.

Dipankar Sarma
Jelenleg több Linux-rendszermagprojekten is dolgozik, többek közt a RCU és VFS fejlesztéseken. Linuxos kora előtt számos egyéb területen dolgozott, többek közt az ABI, OS felhozatal, B-K-meghajtók és többösvényű B-K témájában.

Maneesh Soni
Az IBM Linux Technology Központban dolgozik a Linux méretezhetőségének fejlesztési projektjén. Nagy tapasztalatokkal rendelkezik a rendszerprogramok, különösen az operációs rendszermagok és a fájlrendszerek területén.