Kizárási módszerek a rendszermagban

Robert kifejti nekünk a rendszermag-kizárási módszerek működését, hogy miért van rájuk szükség, és hogy hogyan hoznak létre a segítségükkel a rendszermagfejlesztők biztonságos kódot.

A megfelelő kizárás alkalmazása nehéz, sőt, nagyon nehéz. Ha a rendszermagban nem a megfelelő kizárást alkalmazzuk, véletlenszerű lefagyásokban lesz részünk, illetve egyéb furcsaságok fordulhatnak elő. Rosszul megválasztott kizárások alkalmazásakor a kód nehezen olvashatóvá válik, a működése sem lesz megbízható, és a rendszermagon dolgozó társaidat is az őrületbe kergeted. Ebben a cikkben kifejtem, miért szükséges a rendszermagban kizárásokat alkalmazni. Felsorakoztatok néhány általános kizárási szabályt, végül pedig kitérek az alkalmazható kizárási módozatokra.

Miért szükséges kizárást alkalmazni a rendszermagban?

Kizárások alkalmazása elsősorban a rendszermagban zajló folyamatok összehangolásához szükséges. Ezekhez a folyamatokhoz – vagyis a kényes szakaszokhoz – szükség van bizonyos fokú védelemre, amellyel az események megfelelő időben történő kezelése, illetve az események többszöri kezelésének elkerülése biztosítható. Megfelelő kizárások alkalmazása nélkül úgynevezett versenyhelyzet jön létre. Képzeld csak el, hogy egy közösen használt osztott `i` változó esetén egy `i++` is milyen veszélyes lehet! Tegyük fel, hogy az értékét először az egyik processzor olvassa ki, majd a másik, utána pedig mindketten növelik az értékét, és mindketten visszaírják a memóriába. Ha az `i` értéke eredetileg kettő volt, akkor a műveletet követően négynek kellene lennie, csakhogy így ez az érték valójában mindössze ***három*** lesz!

Ez nem azt jelenti, hogy kizárási gondok csak SMP-rendszereken jelentkezhetnek (SMP – egyidejűleg több processzorral dolgozó rendszer). Megszakításkezelők is vethetnek fel kizárási gondokat, akárcsak a megszakításos ütemezőn alapuló rendszermagok, vagy bármely kód, ami alvó módba tér. Ezek közül csak az SMP tekinthető teljesen egyidejűnek, mivel egyedül SMP-rendszereken fordulhat elő, hogy két vagy több kódrészlet teljesen azonos időben fut. Az első esetekben is felléphetnek látszólagos egyidejűség, ahol – bár a kódrészletek nem teljesen azonos időben futnak le – mégis összezavarhatják egymás adatait.

E kényes kódrészletek esetén mindenképpen szükséges a kizárások alkalmazása. A Linux-rendszermag több kizáró módszert is kínál, így biztosítva a fejlesztőknek a biztonságos és hatékony kód megírásához szükséges feltételeket.

SMP-kizárások egyprocesszoros magokban

Függetlenül attól, hogy többprocesszoros rendszert használsz-e vagy sem, elképzelhető, hogy azok az emberek, akik a kódodat futtatják, ilyen rendszert használnak. Ennek következtében az olyan kód, amely a kizárásokat nem megfelelően kezeli, nem kerülhet be a Linux rendszermagjába. Megszakításosan ütemező rendszer esetén egyprocesszoros rendszereken is létkérdés a megfelelő kizárások alkalmazása. Ezért ne feledd: a kizárások alkalmazása rendkívül fontos.

Linus nagyszerű döntésének köszönhetően az SMP- és egyprocesszoros rendszermagok szerkezete eltér egymástól. Ennek következtében bizonyos kizárások egyáltalán nem léteznek az egyprocesszoros rendszermagokban. A `CONFIG_SMP` és `CONFIG_PREEMPT` egészen más kizárási módozatokat eredményezhet. Bár mindez a fejlesztő számára teljesen mindegy: mindíg alkalmzzunk kizárást, így semmilyen helyzetben nem lehet gond.

Oszthatatlan műveletek

Először az oszthatatlan műveleteket vesszük át, két okból is. Elsősorban azért, mert ezek jelentik a legegyszerűbb időzítést a rendszermagban, így ezek használata érhető meg a legegyszerűbben. Másodsorban pedig azért, mert az összetett kizárási módszerek is az egyszerű időzítéseken alapulnak. Ilyenformán ezek rendszermagkizárásokat létrehozó blokkokat alkotnak.

Az oszthatatlan műveleti jelek, mint az összeadás vagy a kivonás, egyetlen oszthatatlan műveletet alkotnak. Vegyük az előző `i++`-os példát. Ha képesek lennénk kiolvasni az `i` változót, az értékét növelni, majd egyetlen oszthatatlan műveletben visszaírni a memóriába, nem alakulhatna ki az előzőleg említett versenyhelyzet. Az oszthatatlan műveleti jelek segítségével atomi, vagyis oszthatatlan műveleteket végezhetünk el.

Két fajtájuk létezik: az egész számokkal dolgozó tagfüggvények és a bitekkel dolgozó tagfüggvények. Egész számok esetén a szükséges kód így néz ki:

```
atomic_t v;

atomic_set(&v, 5);  /* v = 5 (oszthatatlan) */
atomic_add(3, &v);  /* v = v + 3
                     ➥(oszthatatlan) */
atomic_dec(&v);     /* v = v - 1
                     ➥(oszthatatlan) */
printf("Az eredmØny 7 lesz: %d\n",
      ➥atomic_read(&v));
```

Ez elég egyszerű. Létezik viszont néhány kikötés, amelyeket az oszthatatlan műveletek alkalmazásakor figyelembe kell vennünk. Először is az `atomic_t` típussal létrehozott változókat csak és kizárólag az `atomic`-függvények képesek kezelni. És ugyanez fordítva igaz, vagyis az `atomic`-függvények csakis `atomic_t` típusú változókat képesek kezelni. Végül pedig az egyes rendszerek közötti különbségek miatt az `atomic_t` típusú változónak csak az első 24 bitjét vehetjük biztosnak. A „Függvényreferencia” részben az oszthatatlan műveletekkel kapcsolatos összes függvényt megtalálod.

Az oszthatatlan műveletek következő fajtája az, amely különálló bitekkel dolgozik. Az ilyenek egyszerűbbek, mint az egész számra épülő tagfüggvények, mivel képesek a C beépített

típusaival dolgozni. Vegyük például a `void set_bit(int nr, void addr)` függvényt. Ez a függvény az `addr` által mutatott adat `nr`-edik bitjét 1-esre állítja. Ezek a függvények is megtalálhatók a „Függvényreferencia” részben.

Forgózárak

Bármilyen más esetben, ami egy picit is bonyolultabb, mint a fenti példa, sokkal teljesebb megoldásra lesz szükségünk. Az egyik legegyszerűbb kizárási módozat a forgózár (spinlock), ami az ***include/asm/spinlock.h*** és ***include/linux/spinlock.h*** fájlokban van meghatározva. A forgózár egy nagyon egyszerű fenntartó kizárás. Ha egy folyamat egy forgózárat próbál megszerezni, de az foglalt, a folyamat addig kénytelen várni, míg a forgózár fel nem szabadul. Így egyszerű és gyors kizárás jön létre. Lássunk egy példát a forgózárak legegyszerűbb fajtájára!

```
spinlock_t mr_lock = SPIN_LOCK_UNLOCKED;
unsigned long flags;

spin_lock_irqsave(&mr_lock, flags);
/* kØnyes szakasz... */
spin_unlock_irqrestore(&mr_lock, flags);
```

A `spin_lock_irqsave()` használatával helyileg letilthatjuk a megszakítások kezelését, és SMP-rendszereken forgózárakat hozhatunk létre. Ilyen módon mind a megszakításokból, mind az SMP-ből adódó egyidejűségeket kivédhetjük. A `spin_unlock_irqrestore()` függvény meghívásával a megszakítások az eredeti állapotba állíthatók vissza. Egyprocesszoros rendszermagok esetén a fenti kód a következőképpen fordul le:

```
unsigned long flags;

save_flags(flags);
cli();
/* kritikus szakasz... */
restore_flags(flags);
```

Ez a kódrészlet anélkül biztosítja a szükséges megszakítások elleni védelmet, hogy feleslegesen SMP-védelmet is biztosítana. Ennek a védelemnek egy másik fajtája a `spin_lock_irq()`. Ez a függvényhívás feltétel nélkül letiltja a megszakításokat, hasonlóan a `cli()` és `sti()` függvényekhez. Például:

```
spinlock_t mr_lock = SPIN_LOCK_UNLOCKED;

spin_lock_irq(&mr_lock);
/* kØnyes szakasz... */
spin_unlock_irq(&mr_lock);
```

Ez a kódrészlet csak akkor biztonságos, ha a megszakítások eredetileg – még a kizárás megszerzése előtt – sem voltak letiltva. Mivel a rendszermag egyre nagyobbá és bonyolultabbá válik, a rendszermagbeli szálak is egyre követhetetlenebbek lesznek, így az effajta kizárásokat ajánlatos messze elkerülni. Kivéve természetesen, ha nagyon biztos vagy a dolgodban.
A fentebb használt forgózárak feltételezik, hogy az így védett adat mind a megszakításkezelőkben, mind pedig magmódban elérhető. Ha biztos vagy benne, hogy az adott kód csak a felhasználó szintjén hívódik meg (például rendszerhívások esetén), akkor elegendő a `spin_lock()` és `spin_unlock()` függvények használata, amelyek egy adott kizárást foglalnak le vagy szabadítanak fel, a megszakításokra azonban nincsenek hatással. A forgózárak legutolsó változata a `spin_lock_bh()` függvény, mely az alapvető kizárást biztosítja és a `softirq`-kat tiltja le. Erre olyan kódrészleteknél van szükség, melyek többnyire `softirq`-kon kívül futnak, de alkalmanként `softirq`-ból is futhatnak. Az ehhez tartozó függvény a `spin_lock_bh()`.
Fontos, hogy Linuxban a forgózárak nem ismételten meghívhatók, mint néhány más operációs rendszer esetén. Sokak szerint erre egyébként sincs szükség, mivel a többszörös forgózármeghívás csak a kód minőségét rontaná. Ez annyit tesz, hogy ügyelned kell, nehogy olyan forgózárat foglalj le, amelyet már tartasz, mivel így a program könnyen holtpontra juthat.
A forgózárak alkalmazása olyan helyeken szükséges, ahol a kizárás használata csak rövid ideig kívánatos, mivel a folyamat addig vár tétlenül, amíg a kizárás fel nem szabadul (a „Szabályok” szakaszra pillantva megtudhatod, milyen várakozások minősülnek túl hosszúnak). Szerencsére a forgózárak bárhol használhatók, kivéve az olyan kódrészletet, ami bizonyos esetekben alvó módba kerülhet. Soha ne hívj meg olyan kódot, amely a felhasználó memóriáját kezeli, kerüld el a `kmalloc()` függvényt a `GFP_KERNEL` zászlóval (flag), valamint a jelzőkkel (semaphore) és az ütemezéssel kapcsolatos függvényeket is felejtsd el, ha bármilyen foglalt forgózárral rendelkezel.
Ne feledd, én figyelmeztettelek!
Ha olyan kizárást szeretnél, amelyet akár hosszabb ideig is tarthatsz, és nem zavarja a szálak egyidejűsége, de az esetleges alvó mód sem, akkor a jelzőkre van szükséged.

Jelzők

A jelzők (semaphore) a Linuxban alvó kizárások. Ha a sor egy másik folyamattal versenyhelyzetbe kerül, a jelzők a folyamatot alvó módba küldik. A forgózárakkal ellentétben jelzőket akkor használunk, ha a rájuk való várakozás hosszabb ideig is elhúzódhat. Rövidebb várakozások esetén viszont a jelzők használatával a processzoridőt pazaroljuk, mivel ilyenkor számolnunk kell azzal, hogy egy jelzőre várakozó folyamat alvó módba tér, amit azután, ha a jelző felszabadult, fel kell ébreszteni – és ez elég költséges folyamat. Mindazonáltal mivel a szál ilyenkor alvó módba tér, a jelzők olyan helyeken is használhatók, ahol a forgózárak nem. Más szóval jelzők tartásakor az adott kódrészlet futását akár blokkolhatjuk is.
Linuxban a jelzőket egy C-szerkezet testesíti meg: a ***include/asm/semaphore.h*** fájlban található `struct semaphore`. Ez a szerkezet egy olyan mutatót foglal magában, amely egy várakozási sorra mutat, illetve egy felhasználásszámlálót is tartalmaz. A várakozási sor azoknak a folyamatoknak a listáját tartalmazza, amelyek a jelzőre várakoznak. A felhasználásszámláló az egyidejűleg tartható folyamatok számát mutatja meg. Ha ez az érték negatív, a jelző nem érhető el, és a számláló abszolút értéke a várakozási sorban tartózkodó folyamatok számát tartalmazza. Ennek a számlálónak futás közben a `sema_init()` függvény ad kezdeti értéket. Ez többnyire `1` (ilyen esetben a jelzőt `mutex`-nek, vagyis kölcsönös kizárásnak hívjuk).
A jelzőhöz két függvényen keresztül férhetünk hozzá: a `down`-on és az `up`-on keresztül (vagyis fel és le, ezeket történeti okokból `P`-nek és `V`-nek is nevezzük). Az első megpróbálja lefoglalni a jelzőt, és ha ez nem sikerül, várakozik; a másik ellenben felszabadítja a jelzőt és vele együtt minden várakozó folyamatot.
A Linuxban a jelzők használata egyszerű. A jelző lefoglalásához

Függvényreferencia

Forgózárak

- `void spin_lock(spinlock_t *lock)`: a megadott kizárást lefoglalja, és ha foglalt, addig várakozik, amíg fel nem szabadul.
- `void spin_lock_irq(spinlock_t *lock)`: hasonló a `spin_lock()`-hoz, de a helyi processzoron a megszakításokat is letiltja.
- `void spin_lock_irqsave(spinlock_t *lock, unsigned long flags)`: ugyanaz, mint a `spin_lock_irq()`, viszont a megszakításjelzőket menti a `flags` változóba.
- `void spin_lock_bh(spinlock_t *lock)`: olyan, mint a `spin_lock()`, de a `softirq`-k futtatását is megakadályozza.
- `void spin_unlock(spinlock_t *lock)`, `void spin_unlock_irq(spinlock_t *lock)`, `void spin_unlock_irqrestore(spinlock_t *lock, unsigned long flags)` és `void spin_unlock_bh(spinlock_t *lock)`: az előző függvények felszabadító változatai.
- `void spin_trylock(spinlock_t *lock)`: ha a kizárás foglalt, nullától különböző értékkel tér vissza, máskülönben a visszaadott érték nulla.
- `void read_lock(rwlock_t *lock)` és `void read_unlock(rwlock_t *lock)`: az olvasó–író kizárás olvasó változatát foglalja le és szabadítja fel.
- `void write_lock(rwlock_t *lock)` és `void write_unlock(rwlock_t *lock)`: az olvasó–író kizárás író változatát foglalja le és szabadítja fel.

Jelzők

- `void sema_init(struct semaphore *sem, int val)`: kezdeti értéket ad a jelző felhasználásszámlálójának.
- `void down(struct semaphore *sem)`: az adott jelzőt próbálja megszerezni, vagy a felszabadulására vár.
- `int down_interruptible(struct semaphore *sem)`: mint a `down()`, de jel esetén `EINTR`-rel tér vissza.
- `int down_trylock(struct semaphore *sem)`: mint a `down()`, de ha a jelző foglalt, azonnal visszatér egy nullától különböző értékkel.
- `void up(struct semaphore *sem)`: felszabadítja az adott jelzőt.
- `void init_rwsem(struct rw_semaphore *rwsem)`: az író–olvasó jelzőnek kezdeti értékül `1`-et ad.
- `void down_read(struct rw_semaphore *rwsem)` és `void up_read(struct rw_semaphore *rwsem)`: olvasás esetén az adott olvasó–író jelzőt foglalja le, illetve szabadítja fel.
- `void down_write(struct rw_semaphore *rwsem)` és `void up_write(struct rw_semaphore *rwsem)`: az adott olvasó–író jelzőt írás esetén foglalja le, illetve szabadítja fel.

Nagy rendszermagkizárás

- `void lock_kernel()`: teljes kizárás igénylése (BKL).
- `void unlock_kernel()`: a BKL felszabadítása.
- `int kernel_locked()`: ha a BKL foglalt, igaz értékkel tér vissza, máskülönben hamissal.

Megszakításos ütemezés tiltása

- `void preempt_disable()`: növeli a megszakításos ütemezés számlálóját.
- `void preempt_enable()`: csökkenti a megszakításos ütemezés számlálóját, és engedélyezi a megszakításos ütemezést, ha szükséges.
- `void preempt_enable_no_resched()`: csökkenti a megszakításos ütemezés számlálóját.
- `int preempt_get_count()`: lekérdezi a megszakításos ütemezés számlálóját.

a `down_interruptible()` függvényt kell meghívni, amely eggyel csökkenti a jelző felhasználásszámlálóját. Ha az új érték negatív, a folyamat a várakozási sorba kerül. Amennyiben az új érték nullával egyenlő vagy nagyobb, a folyamat megkapja a jelzőt, és a függvény `0` értékkel tér vissza. Ha a várakozást valamilyen jel szakítja meg, a hívás `-EINTR`-rel tér vissza a jelző megszerzése nélkül.
Az `up()` függvénnyel a jelzőt a felhasználásszámláló értékét növelve felszabadíthatjuk. Amennyiben az új érték `0` vagy nagyobb, a várakozási sorban lévő folyamatok némelyike felszabadul:

```
struct semaphore mr_sem;

sema_init(&mr_sem, 1);      /* a felhasznÆlÆs-
                            ↪szÆmlÆl  1 */

if (down_interruptible(&mr_sem))
  /* a jelzı lefoglalÆsa egy jelzØs miatt */
    / * nem val sult meg... */

/* kØnyes szakasz (a jelzıt lefoglaltuk) ... */
up(&mr_sem);
```

A rendszermag egy `down()` függvényt is biztosít, ami annyiban különbözik a `down_interruptible()` fügvénytől, hogy a folyamatot meg nem szakítható alvó módba küldi. Ebben az állapotban minden jelzést, amit a folyamat kap, figyelmen kívül hagy. A programozók többnyire a `down_interruptible()` függvényt használjak. Végezetül pedig a Linux a `down_trylock()` függvényt is felkínálja, ami a jelző foglalt állapotában valamely nullától különböző értékkel blokkolás nélkül tér vissza, ha pedig a jelző szabad, lefoglalja azt.

Olvasó–író kizárások

A forgózárakon és jelzőkön kívül a Linux-rendszermag úgynevezett olvasó–író változatokat is biztosít, amelyek a kizárásokat két csoportba osztják: olvasásra és írásra. Ugyannak az

adatnak az egy időben történő olvasása többnyire nem jár gonddal, legalábbis amíg az adat nem módosul, viszont az írásról nem mondható el ugyanez. Ezért az író–olvasó kizárásokkal engedélyezett a több szálon történő olvasás, írás azonban csak egyetlen szálon lehetséges. Írás közben az olvasás sem engedélyezett. Ha a felhasznált adat egyértelműen kezelhető olvasásokkal és írásokkal, különösen, ha olvasás lényegesen többször fordul elő, akkor az író–olvasó kizárások használata ajánlott.
Az író–olvasó forgózárakat rwlock-nak hívjuk. Használatuk a forgózárakéhoz hasonló, kivéve természetesen az írások és olvasások külön történő kezelését.

```
rwlock_t mr_rwlock = RW_LOCK_UNLOCKED;

read_lock(&mr_rwlock);
/* kØnyes szakasz (csak olvasÆs)... */
read_unlock(&mr_rwlock);

write_lock(&mr_rwlock);
/* kØnyes szakasz (olvasÆs Øs  rÆs)... */
write_unlock(&mr_rwlock);
```

Hasonlóképpen az olvasó–író jelzőket `rw_semaphore`-nak nevezzük. Működésük a rendes jelzőkéhez hasonló, kivéve az írások és olvasások külön történő kezelését:

```
struct rw_semaphore mr_rwsem;

init_rwsem(&mr_rwsem);

down_read(&mr_rwsem);
/* kØnyes szakasz (csak olvasÆs)... */
up_read(&mr_rwsem);

down_write(&mr_rwsem);
/* kØnyes szakasz (olvasÆs Øs  rÆs)... */
up_write(&mr_rwsem);
```

Az olvasó–író kizárások megfelelő alkalmazása érezhető sebességnövekedést eredményez. Megjegyzendő, hogy néhány más rendszerrel ellentétben a Linuxban az olvasási kizárásból nem képezhetünk önműködően írási kizárást. Ugyanakkor ha létezik egy olvasási kizárásunk, és ilyenkor próbálunk kizárólagos hozzáféréshez jutni, az eredmény holtpont lesz. Ha tudod, hogy írásra is szükséged lesz, alapesetben már a művelet legelején írási kizárást kell igényelned. Ha az írások és olvasások közti különbség zavaros, elképzelhető, hogy az író–olvasó kizárások használata nem a legjobb választás.

Erős olvasási kizárások

Az erős olvasási kizárások (vagyis big-reader kizárások, brlockok) az ***include/linux/brlock.h*** fájlban vannak meghatározva. Az effajta kizárások az egyszerű író–olvasó kizárások különleges csoportját képezik. Eredetileg a RedHatnél dolgozó ***Ingo Molnar*** tervezte őket, és olyan forgózárakhoz hasonló szerkezetet hozott velük létre, amivel nagyon gyorsan szerezhetünk engedélyt olvasásra, az írási műveletek esetén ugyanakkor csak rendkívül lassan. Ez a fajta kizárás kifejezetten előnyös az olyan helyzetekben, amikor temérdek olvasószál mellett csak elvétve akad egy-egy írószál. Bár az olvasási kizárások különböznek az egyszerű olvasó–író kizárástól, használatuk megegyezik, eltekintve attól, hogy az erős olvasás kizárások a ***brlock.h*** fájlban található ***brlock_indices*** szakaszban határozódnak meg.

```
br_read_lock(BR_MR_LOCK);
/* kØnyes szakasz (csak olvasÆs)... */
br_read_unlock(BR_MR_LOCK);
```

Az erős olvasáskizárások használata az írások lassúsága miatt csak néhány különleges esetre korlátozott, és ez valószínűleg a jövőben sem fog változni.

Szabályok

Az alábbi szabályok szerint élj, hogy senkinek ne essék baja!

1. Akkor is használj kizárásokat, ha nem használsz SMP-t.
2. A kizárások az adatok kezelésekor és nem a kód futtatásakor szükségesek, vagyis a kizárásoknak az adatokkal kell kapcsolatban állniuk, nem a kódterületekkel.
3. Van egy halvány vonal a durva kizárások és a finom kizárások között is. Egyrészt finom kizárások alkalmazásakor a kizárási verseny csökkentése miatt a kód is hatékonyabb lesz, másrészről minél több kizárást használsz, annál több memória és processzoridő fogy, és a kizárások rendszere is egyre bonyolultabbá válik.
4. Amennyiben a várakozási idő hosszabb, próbáld elkerülni a forgózárak használatát. Ez a hosszabb idő természetesen mindenkinek mást jelent, egy korszerű rendszer esetén az 1–5 milliszekundum tekinthető a felső határnak. Emlékezz vissza, hogy forgózárak esetén az egyes szálak tétlenül várnak. A hosszú idejű kizárások ennek következtében SMP- és megszakításos időosztású rendszereken nagyobb kizárási versenyt alakítanak ki. Jelzők esetén ennek a fordítottja érvényesül: a jelzőre történő várakozáskor a folyamat alvó állapotba kerül, és ezalatt egy másik folyamat kerülhet előtérbe – de ez a művelet lassabb is, így a használata hosszabb várakozási ciklusok esetén indokolt.
5. Kizárási elveidet gondosan tervezd meg, és ragaszkodj is hozzájuk.

A nagy rendszermagkizárás

A 2.0-s rendszermagok óta a Linux egy úgynevezett nagy rendszermagkizárást használ, amelyet korábban az SMP-rendszerekkel való együttműködés kedvéért vezettek be. A 2.2-es és 2.4-es rendszermagok fejlesztésekor rengeteg energiát fektettek e teljes kizárás eltüntetésébe és egy sokkal finomabban szabályozott rendszer kidolgozásába. Ma már a teljes kizárások használata csak elenyésző számban van jelen, de tény, hogy még létezik, így a rendszermag fejlesztőinek tisztában kell lenniük a viselkedésével.
A teljes kizárást nagy rendszermagkizárásnak is hívják, más néven `BKL`-nek. Ez a kizárásfajta többszörösen újrahívható forgózárat takar, amelynek ismételt meghívása sem juttatja holtpontra a rendszert (mint ahogyan az a normál forgózárak esetében történne). Egy folyamat akár alvó állapotba is léphet, sőt még az ütemezőbe is beléphet a `BKL` tartása közben. Ha egy teljes kizárásban lévő folyamat belép az ütemezőbe, a kizárás feloldódik, így más folyamatok szerezhetik meg. A `BKL` e tulaj-

donságainak köszönhető az SMP-rendszerek viszonylag egyszerű kezelése a 2.0-s rendszermagok idejében. Manapság viszont temérdek érv szól már az effajta kizárás alkalmazása ellen.
A nagy rendszermagkizárás használata egyszerű.
A `lock_kernel()` hívással megszerezhető a kizárás, az `unlock_kernel()`-lel pedig elengedhetjük. Hogyha a `kernel_locked()` függvény nullától különböző értékkel tér vissza, a kizárás érvényes, ellenkező esetben nullát kapunk. Lássunk erre is egy példát!

```
lock_kernel();
/* kØnyes szakasz... */
unlock_kernel();
```

A megszakítható ütemezés vezérlése

A 2.5-ös fejlesztői rendszermagoktól kezdődően (illetve egy folttal már a 2.4-es rendszermagok esetében is) a rendszermag ütemezése megszakítható. Ennek a képességnek köszönhetően a rendszermag dönthet bizonyos folyamatok futásának megszakításáról, ezzel magasabb szintű folyamatokat hozhat előtérbe, még akkor is, ha az adott folyamat a rendszermag belsejében fut. A megszakításos ütemezésen alapuló rendszermagok számos, az SMP-rendszerek ütemezéséhez hasonló gondot vetnek fel. Szerencsére a rendszermag az SMP-rendszerekre már fel van készítve, így az SMP-kizárások alkalmazásával a megszakításos ütemezésű rendszermagok is biztonságosan futhatnak. Ennek ellenére érdemes megemlíteni néhány új szabályt. Például kizárással nem védhetünk processzoronként valamilyen processzorhoz kötődő adatot, mivel a védelmük önműködően megy végbe (ez így biztonságos, mivel az ilyen adat minden processzor esetében egyedi), és ez szükséges a rendszermag megfelelő ütemezéséhez.

Összegzés

SMP-rendszerek esetén mind a rendszer megbízhatósága, mind a méretezhetősége folyamatosan nő. Mióta az SMP-kezelés bemutatkozott a 2.0-s rendszermagokban, az egymást követő rendszermagváltozatok e téren rendkívüli mértékben fejlődtek. Ehhez új és okosabb kizárási módszerek bevezetésére, a korábbi kizárási rendszer felülvizsgálatára és a teljes kizárások megszüntetésére volt szükség a gyorsaságot igénylő területeken. Ez a módi a 2.5-ös rendszermagok esetében is folytatódik, így a jövőben bizonyosan még ennél is hatékonyabb rendszermagok születnek.
A fejlesztők ebből úgy vehetik ki a részüket, hogy megfelelő kizárási eljárásokat alkalmaznak, és egyik szemüket mindig a megbízhatóságon és a sebességen tartják.

Linux Journal 2002. augusztus, 100. szám

Robert Love
(rml@tech9.net) matematika és számítógépes tudományok szakos hallgató a Floridai Egyetemen. Amikor éppen nem Linuxot elemez, autóversenyzik, thai ételeket eszik vagy punkzenét hallgat.

IceWM második pillantásra

Marcel Gagné e havi cikkében (78. oldal) röviden kitér egy jól használható és gyors ablakkezelőre, az IceWM-re. Mivel jómagam is nagy rajongója vagyok a gyors és használható megoldásoknak, valamint az IceWM-felhasználók táborába tartozom, egy-két további gondolatot szeretnék hozzáfűzni a leírtakhoz.
Az IceWM tehát akár kisebb gépeken is elfut, mint sok testvére, például a BlackBox. Mindkét ablakkezelő kicsi és gyors. Az IceWM nagy előnye, hogy gyorsan hozzászokhatunk, lényegében a Windowsban is használatos kombinációkat (Alt+Tab: következő ablak, Alt+Esc: háttérbe küld, Ctrl+Esc: IceWM menü stb.) alapból ismeri, illetve néhány rendkívül hasznossal ki is bővíti. Ide tartozik a munkafelületek közötti váltásra használatosak (Ctrl+Alt+BalNyíl, JobbNyíl: előző, illetve következő felület stb.). Külön ügyesnek tartom azt a trükköt, hogy miközben a munkafelületek között váltunk, ha nyomva tartjuk a Shift gombot, az éppen használt ablakot „magunkkal visszük” az új felületre. Ezzel a módszerrel pillanatok alatt rendet teremthetünk.
A Marcel által írt beállításfájlok közül kiemelném a preferences-t, melyben én a következő változásokat eszközöltem: bekapcsoltam az egérgörgő és a Menügombok használatát (UseMouseWheel=1, Win95Keys=1), valamint telepítettem az xexec programot, majd hozzárendeltem az IceWM menüben lévő `Run` parancshoz (`RunCommand="xexec"`), ezt egyébként a baloldali Menü és az R billentyű kombinációjával is előhívható. Ez külön hasznos, ha nem akarunk egy `xterm`-et indítani egy parancs kiadásához (az `xterm` egyébként könnyedén indítható a Ctrl+Alt+T-vel).
Emellett az ablakkezelő egyéb hasznos kényelmi szolgáltatásokkal is rendelkezik, ha egyszer van egy órácskánk, érdemes végigböngészgetni mind a súgót, mind a beállításfájlokat. Érdekes például, hogy a fejlesztők gondoltak a multimédia-billentyűzetek tulajdonosaira is: az XF86 által felismert billentyűkhöz az ablakkezelő alapértelmezett programokat rendel (ezeket a keys fájlban nézhetjük meg).
A BlackBoxról, erről az igazi minimalistáról is érdemes néhány szót szólni. Főleg azoknak ajánlom, akiknek öreg csacsi gépen kell dolgozniuk vagy csillogtatni akarják profizmusukat. Legnagyobb előnye ugyanis, hogy döbbenetesen kicsi és gyors. Önmagában a felület nem túlzottan látványos, egy darab tálcát tartalmaz. Az viszont biztos, hogy nem ütközünk olyan gondokba, hogy például az ablakkezelő elnyeli az Alt+F2 vagy az F11 billentyűket, és hogy e billentyűkombinációkat nem tudjuk a programjainkban használni. Ugyanis a BlackBox semmilyen kombinációt nem ismer. Az egész billentyűkezelő részét különválasztották, és külön bbkeys néven tudjuk futtatni. Ennek segítségével viszont egy egyszerű, mégis nagyszerű felületen keresztül állíthatjuk be (például), hogy az Alt+Tab a szokásos módon működjön. Ugyanitt tetszőleges kombinációhoz bármilyen parancsot könnyedén hozzárendelhetünk. Így egyetlen gombnyomásra indulhat egy `xterm`, az Opera vagy bármi más. Remek felület, csak haladóknak!

Szy György