Héjprogramozás Linux alatt (9. rész)

Zárolás, várakozás és többpéldányos futás.

Sorozatunk előző részében bemutattuk, hogyan oldható meg biztonságosan az átmeneti fájlok kezelése akkor is, ha a programunkból egyszerre több példány fut. Ugyanakkor vannak olyan helyzetek, amikor a többpéldányos működést kifejezetten tiltani akarjuk, vagyis zárolást kell végrehajtanunk. A zárolás lényege, hogy a héjprogram tényleges működésének megkezdése előtt meggyőződik róla, hogy fut-e már belőle egy másik példány. Ha igen, akkor vagy azonnal kilép, vagy megvárja, amíg a másik program futása befejeződik. Ha nem talál másik példányt, akkor valamilyen egyezményes módon jelzi, hogy ő már fut, és végrehajtja a megadott műveleteket. Természetesen arról is gondoskodnunk kell, hogy kilépéskor a zárolójelet eltávolítsuk, ellenkező esetben soha többé nem tud egyetlen ilyen program sem lefutni.

A zárolófájl kezelése

A zárolást a legegyszerűbben úgy oldhatjuk meg, ha egy meghatározott helyen létrehozunk egy megadott nevű fájlt, ami most a jelző szerepét fogja betölteni. Ha a fájl létezik, akkor már biztosan fut egy példány a programból, ha nem, akkor szabad a pálya. A módszer működésének természetesen alapfeltétele, hogy a fájlt kilépéskor mindenképpen töröljük, még akkor is, ha a program nem szabályos módon állt le. Mint azt a sorozat előző részéből már tudjuk, éppen erre szolgál a `trap` parancs.
Lássunk egy egyszerű példát! Tegyük fel, hogy héjprogramunknak egy beállításfájlt kell módosítania. Ha a programot többen is futtathatják, fennáll annak a veszélye, hogy egyszerre kísérlik meg módosítani. Ennek pedig egészen furcsa következményei lehetnek, hiszen részben vagy egészben felülírhatják egymás beállításait vagy teljesen összezavarhatják a fájl szerkezetét. Ebben a helyzetben tehát mindenképpen zárolás szükséges, amivel határozottan megtiltjuk a beállítóprogram többpéldányos működését.
Lássuk a program legegyszerűbb megvalósítását az *1. listában*!
A beállításokat tároló fájl (***proba.conf***) most az egyszerűség kedvéért a pillanatnyi könyvtárban található, a zárolás jelzésére szolgáló bejegyzést pedig a */tmp* könyvtárban fogjuk elhelyezni. (Természetesen bármilyen más könyvtár is – amelyhez mindazoknak írási jogosultságuk van, akik a programot futtathatják – megfelel a célnak.)
A zárolófájl neve most `pidlock` lesz, vagyis az átmeneti fájlokkal ellentétben semmilyen egyedi azonosítót nem tartalmaz. Ugyanakkor bizonyos később említendő okok miatt zárolás esetén is célszerű jelezni, hogy kicsoda – vagyis melyik folyamatazonosítóval rendelkező programpéldány – hozta létre. Ezt hagyományosan úgy oldják meg, hogy a folyamatazonosítót magában a zárolófájlban helyezik el.
Éppen ez látható példaprogramunk 14. sorában. A 9. sorban megvizsgáljuk, hogy létezik-e a zárolófájl. Ha igen, akkor a program egyszerűen leáll, egy üzenetben jelezve ennek az okát. Ha nem, akkor létrehozza a zárolást, majd belekezd a tényleges műveletek végrehajtásába. Ez esetünkben mindössze annyi lesz, hogy a `read` paranccsal egyetlen sornyi szöveget bekérünk, amit aztán hozzáfűzünk a beállításokat tartalmazó fájlhoz.

```
 1: #!/bin/sh
 2: # Példa zárolásra
 3:
 4: LOCKFILE="/tmp/pidlock"
 5: CONFIG_FILE="proba.conf"
 6:
 7: trap 'rm $LOCKFILE; exit 2' 1 2 3 15
 8:
 9: if [ -f $LOCKFILE ]
10: then
11:   echo "Már fut egy példány
          ➥a programból!"
12:   exit 1
13: else
14:   echo $$ > $LOCKFILE
15: fi
16:
17: echo -n "Új szöveg: "
18: read sor
19: echo $sor >> $CONFIG_FILE
20:
21: rm $LOCKFILE
```

```
 1: if [ -f $LOCKFILE ]
 2: then
 3:   echo "Már fut egy példány
          ➥a programból!"
 4:   echo "Várakozunk a kilépésére..."
 5:   while [ -f $LOCKFILE ]
 6:   do
 7:     sleep 2
 8:   done
 9: fi
```

Kilépés előtt (21. sor) a zárolást töröljük – utat engedve a többi példánynak. Ha menet közben bármilyen galiba történne, és a program nem lenne képes eljutni a 21. sorig, akkor a 7. sorban megadott műveletsor lép életbe. Itt a `trap` parancs elfogja az összes lényegesebb, a program leállásával kapcsolatos jelet, törli a zárolófájlt, majd engedelmeskedve a jelzésnek egy `exit` paranccsal kilép.

Kinek a zárolása?

Figyeljük meg, hogy a 0-s jel (`EXIT`), vagyis a szabályos kilépés nem szerepel az elfogott jelek sorában. Ennek két oka van: egyrészt programunkban nem szerepel olyan közbenső kilépési művelet, amely megelőzi a zárolás létrehozását. A program végén bekövetkező szabályos kilépésnél ugyanakkor mi magunk gondoskodunk a zárolás megszüntetéséről.
Mégha lenne is közbenső kilépés, az esetünkben akkor sem meg-

felelő megoldás, ha a fenti egyszerű `trap` paranccsal kezeltetjük az `EXIT` jelet, mivel így az esetlegesen elindított második programpéldány kilépéskor törölné az első által létrehozott zárolást. Ha mindenképpen egyben akarjuk kezelni a zárolás törlését, akkor az efféle galibák elkerülésére a törlés előtt meg kell győződnünk róla, hogy programunk a maga által létrehozott zárolófájlt törli és nem valaki másét. Éppen erre szolgál a fájlban tárolt folyamatazonosító. Ha ugyanis ez nem azonos a törlést

```
 1: if [ -f $LOCKFILE ]
 2: then
 3: # Hamis zárolás vizsgálata
 4:    lockpid=`cat $LOCKFILE`
 5:    if [ `ps ax | grep "^ $lockpid" |
       ➥wc -l` -eq 0 ]
 6:    then
 7:      echo "Hamis zárolás!"
 8:      rm $LOCKFILE
 9:      echo $$ > $LOCKFILE
10:   else
11:      echo "Már fut egy példány
            ➥a programból!"
12:      exit 1
13:   fi
14: else
15:   echo $$ > $LOCKFILE
16: fi
```

kezdeményező programéval, akkor biztosan nem az ő jelzéséről van szó, tehát nem törölheti. Mindez azt jelenti, hogy a 7. sorban található műveletsort a következőre kell lecserélnünk:

```
trap ' exit ' 1 2 3 15
trap ' if [ `cat $LOCKFILE` -eq "$$" ];
➥then rm $LOCKFILE ; exit ; fi' EXIT
```

Mindenekelőtt ennél a módszernél a 21. sor szerepeltetése nemcsak fölösleges, hanem kifejezetten hibás is, hiszen most a szabályos kilépéssel kapcsolatos törlést is a második `trap` utáni kódra bíztuk. Az első `trap` segítségével mindössze annyit teszünk, hogy a „nem szabályos” leállással (például a Ctrl+C megnyomása) kapcsolatos jeleket szabályos leállássá változtatjuk úgy, hogy hatásukra egy közönséges `exit` parancsot hajtunk végre. Ezt a következő, csak az `EXIT` jelet kezelő `trap` ismét elfogja, és a szokásos módon kezeli.
Az olvasó most nyilván azt kérdezi, hogy miért nem vonjuk össze a két `trap`-et, elvégre mindkettő pontosan ugyanazt a műveletsort hajtja végre. Látszólag tehát semmi mást nem kellene tennünk, mint a második után az elsőnél megadott jeleket is felsorolni, az elsőt pedig egyszerűen kihagyni:

```
trap ' if [ `cat $LOCKFILE` -eq "$$" ];
➥then rm $LOCKFILE ; exit ; fi' EXIT 1 2 3 15
```

Nosza, próbáljuk ki! Cseréljük le a két sort a fenti egyre, indítsuk el a programot, majd szöveg helyett vagy némi gépelés után nyomjuk meg a Ctrl+C billentyűket. Hibaüzenetet kapunk, amely szerint a zárolást jelző fájl nem létezik. (Hova lett?!)
A `trap` utáni utasítássorban kénytelenek vagyunk magunk is végrehajtani egy `exit` parancsot, különben a program soha nem áll le. (Az előző részből tudhatjuk, hogy halhatatlan program nincs. A `kill -KILL` parancs mindenkire jótékony hatással van.) Ha a `trap` nem `EXIT` jelet fog el, akkor maga mindenképpen kivált egy ilyent, amit aztán boldogan el is fog, hiszen szerepel a listájában. (Kétszer ugyanazt a jelet természetesen nem fogja el.)
A bűntény tehát a következőképpen történt. Amikor leütöttük a Ctrl+C billentyűket, a keletkező 2-es jelet a `trap` elfogta, és ennek eredményeképpen törölte a zárolófájlt. Ezután az `exit` paranccsal maga is kiváltott egy 0-s jelet, amit ismét elfogott. Törölni viszont már nincs mit, tehát hibaüzenetet kapunk, majd a program valóban leáll.
Akármelyik (helyes) megoldást választjuk is, példaprogramunk működéséről úgy győződhetünk meg, hogy két terminálablakban próbálunk meg belőle futtatni egy-egy példányt.

Várakozás zárolás feloldására

Az előbbi példában, ha a program zárolást érzékelt, egyszerűen leállt. Előfordulnak azonban olyan helyzetek, amikor célszerűbb megvárni, amíg a másik példány lefut és törli a zárolást. Ehhez az előbbi megvalósítás 9–15. sorát a *2. listában* található kódrészletre kell cserélnünk.
A `sleep` parancs semmi egyebet nem tesz, mint várakozik. Hogy meddig, azt az utána szereplő érték adja meg másodpercekben. (A `sleep` valójában nem végez pontos időzítést. Ez egy többfeladatos, többfelhasználós operációs rendszerben eleve nem egyszerű feladat.)
Esetünkben az 5. sorban induló ciklus két másodpercenként ellenőrzi, hogy a zárolófájl még létezik-e, és csak akkor áll le, ha ez a feltétel már nem teljesül. Természetesen más ellenőrzési szakaszt is megadhatunk, illetve egy számlálót is használhatunk, és bizonyos számú ellenőrzés után feltétel nélkül leállíthatjuk a programot.

Hamis zárolás törlése

Bonyolultabb programok esetében előfordulhat olyan helyzet, amikor valamilyen oknál fogva nem kezelhetünk minden leállító jelet, és így a zárolófájl a folyamat leállása után is hátramaradhat. Ha ilyenkor a fenti egyszerű megoldást használjuk a zárolás vizsgálatára, programunk a végtelenségig várakozni fog, hiszen semmilyen módon nem érzékeli, hogy a zárolást létrehozó program már nem fut, így esélye sincs annak feloldására.
A megoldás ismét a zárolófájlban tárolt folyamatazonosítóra épül. Ha listát kérünk valamennyi futó folyamatról (`ps ax`), és azok között nem szerepel a zárolás tulajdonosa, akkor nyilvánvalóan hamis zárolásról van szó, amit törölni kell. Mindez azt jelenti, hogy az eredeti programunk 9–15. sorát a *3. listában* található kódrészletre kell cserélni.
A 4. sorban kiolvassuk a zárolófájlban tárolt folyamatazonosítót; az ötödikben a teljes folyamatlistában megszámoljuk azokat a sorokat, amelyek elején ez a szám szerepel. (A `ps` a folyamatazonosító elé egy szóközt illeszt be!) Ha nincs ilyen, akkor a kérdéses folyamat már nem fut, tehát az általa létrehozott zárolást törölni kell, ugyanakkor létre kell hozni helyette az újat (9. sor). Programunk működését a legegyszerűbben úgy próbálhatjuk ki, ha egy „segédprogramot” írunk hozzá, ami egy hamis zárolást hoz létre a saját folyamatazonosítójával, majd azonnal leáll.

Büki András (buki.andras@insilico.hu)
Körülbelül kilenc éve dolgozik Linux operációs rendszerrel. Állandó szerzőtársa Prof. Dr. H. V. Kuksinak, akivel a Duna vagy a Tisza partján szoktak az élet és a tudomány viselt dolgairól töprengeni.