Héjprogramozás: átmeneti fájlok és jelek (8. rész)

Sorozatunk mostani részében folytatjuk azoknak az általános programozási módszereknek és eljárásoknak a bemutatását, amelyekkel gyakran találkozhatunk munkánk során.

Egy program megvalósítása során olyan helyzetbe is kerülhetünk, amikor a feldolgozás valamiféle „köztes termékét” átmenetileg kénytelenek vagyunk egy vagy több fájlban tárolni. Például ilyen helyzet akkor állhat elő, amikor egy bizonyos művelet végrehajtásához a bemenet egészéről kell valamilyen adattal rendelkeznünk. Ha az adatokat fájlból vesszük, nincs más dolgunk, mint kétszer végigolvastatni a tartalmukat. Az első olvasás alapján meghatározhatjuk a kívánt globális jellemzőket, a másodikkal pedig elvégezzük a feldolgozást. Átmeneti tárolásra, illetve fájlra ebben az esetben nincs szükségünk, hiszen a bemenet eleve „tárolt formában” áll rendelkezésünkre. Ez így eléggé elvontan hangzik, ezért lássunk rögtön két példát!

Tegyük fel, hogy egy szövegfájl tartalmát bekeretezve akarjuk megjeleníteni a képernyőn. A keretnek nyilván igazodnia kell a szöveg szélességéhez, vagyis a leghosszabb sor hosszához. Azt viszont, hogy milyen széles a leghosszabb sor, csak úgy tudjuk megállapítani, ha az egész fájlt végigolvassuk.

Hasonló a helyzet, ha a szöveg sorait mondjuk fordított sorrendben szeretnénk kiíratni. Előbb tudnunk kell, hogy melyik (hányadik) az utolsó sor, csak azután kezdhetünk el visszafelé haladni a kiíratással.

Ha fájlból vesszük a szöveget, mindennek semmi akadálya. Mi a helyzet azonban akkor, ha a unixos szokásoknak megfelelően programunkat szabványos bemenetről érkező szöveg feldolgozására is képessé akarjuk tenni? Honnan tudhatjuk egy szöveg elején, hogy hány karakterből fog állni a leghosszabb sora, vagy hogy hány sorból fog állni a teljes bemenet? Természetesen sehonnan. Ezeknek a kiderítéséhez legalább egyszer végig kell olvasnunk a teljes bemenetet, ami jelen esetben azért gond, mert a szabványos bemenet csak egyszer olvasható.

Ha a bejövő csőből kiolvastunk egy sort, azzal ki is töröltük onnan, vagyis már nem lesz mit feldolgozni. Mi tehát a megoldás? Természetesen az, hogy a bemenetet – akár az első olvasással párhuzamosan – egy átmeneti fájlba irányítjuk, és a következő lépésben az így „bespájzolt” adatot dolgozzuk fel.

A dolog miként je

A következő feladat az átmeneti fájlok elhelyezése. Megtehetjük ugyan, hogy egy adott néven mindig a pillanatnyi könyvtárban vagy mindig egy adott könyvtárban helyezzük el programunk átmeneti fájljait, de ebből mindenféle galibák származhatnak. Nem biztos például, hogy mindig írási jogunk van a pillanatnyi könyvtárhoz. Ha van, akkor viszont abban nem lehetünk biztosak, hogy nem futtatja a programot velünk egy időben más is. (Ne felejtsük el, hogy a unixok egyik fő erőssége a többfeladatos, többfelhasználós működés.) Márpedig ha egyszerre többen használják ugyanazt az átmeneti fájl, abból minden kétséget kizáróan irgalmatlan keveredés származik majd. Gondoskodnunk kell tehát valahogyan az átmeneti fájlok egyediségéről, illetve használat utáni eltakarításukról is.

A Unix-rendszereken és így a Linuxon is éppen ezt a célt szolgálja a */tmp* könyvtár, illetve a folyamatazonosító. A */tmp* amolyan közös szemétdomb, ahová mindenki írhat, de az „általános írási jogosultság” ellenére egy különleges rendszerszolgáltatásnak, az úgynevezett ragasztó bitnek (sticky bit) köszönhetően a fájlokat kizárólag a tulajdonosuk törölheti. A folyamatazonosító minden egyes futó programpéldányra egyedi, ezért ha ezt szerepeltetjük az átmeneti fájlok nevében, automatikusan kizárt a keveredés lehetősége. Héjprogramokban

```
 1: #!/bin/sh
 2:
 3: vissza()
 4: # A $1-ben megadott fájl sorait olvassa
      ➥visszafelé
 5: {
 6:   hossz=`cat $1 | wc -l`
 7:   while [ $hossz -ge 1 ]
 8:   do
 9:     cat $1 | sed -n "$hossz p"
10:     hossz=`expr $hossz - 1`
11:   done
12: }
13:
14: # Fájlokból érkező bemenet
15: if [ $# -ne 0 ]
16: then
17:   for nev in $*
18:   do
19:     if [ -f $nev ]
20:     then
21:       vissza $nev
22:     else
23:       echo "A(z) $nev nevű fájl nem
                ➥létezik!" 1>&2
24:     fi
25:   done
26: else  # A szabványos bemenet olvasása
27:   touch /tmp/tactmp$$
28:   trap 'rm /tmp/tactmp$$ ; exit 1' 1 2 3
15
29:   while read sor
30:   do
31:     echo $sor >> /tmp/tactmp$$
32:   done
33:   vissza /tmp/tactmp$$
34:   rm /tmp/tactmp$$
35: fi
36: exit 0
```

a folyamatazonosítót a `$$` belső változó tartalmazza, így például a `/tmp/tmp$$` *név* minden korábban említett igényt kielégít.

A héjprogramokban kezelhető fontosabb jelek

Kiváltó esemény	A jel neve	A jel száma	Leírás
exit parancs	**EXIT**	0	**exit** parancsot hajtott végre a program vagy normál úton befejeződött.
kijelentkezés	**SIGHUP**	1	Megszakadt a kapcsolat a terminállal.
Ctrl+C	**SIGINT**	2	A felhasználó leállította a programot
Ctrl+\	**SIGQUIT**	3	Kilépésre való utasítás a billentyűzetről.
kill -9	**SIGKILL**	9	Feltétel nélküli leállás. (Nem fogható el!)
shutdown vagy **kill**	**SIGTERM**	15	Felszólítás leállásra.

Egy példa

Példaként az előbb említett két feladat közül valósítsuk meg a fájl sorainak fordított sorrendben való megjelenítését. (Ezt a célt szolgálja egyébként a legtöbb rendszeren megtalálható `tac` parancs is, így erre a feladatra héjprogramot írni inkább csak amolyan ujjgyakorlat.) Mint megannyi más esetben, a tényleges feldolgozást végző kódot most is függvény formájában célszerű megvalósítani, amely a feldolgozandó fájl nevét kapcsolóként kaphatja meg (lásd a *listában*).

A program lelke a 7–11. sorban látható. A 6. sorban megszámoltuk a bemenet sorait, tehát már tudjuk, hogy melyik az utolsó. Ezután egy `while` ciklusban ezt a mutatót lépésenként csökkentjük (10. sor), és minden egyes fordulónál az adott sorszámú sort kiíratjuk – az utóbbihoz a `sed "p"` parancsát használjuk. Természetesen ugyanezt `awk`-val vagy egy `head-tail` párossal is megoldhattuk volna, de talán ez a legegyszerűbb.

A program maradéka a vezérlést ellátó főprogram. Azt, hogy fájlokból kell vennie a bemenetet, onnan ismeri fel a program, hogy a parancssori kapcsolók száma nem nulla (15. sor). Ilyenkor egy `for` ciklussal (17. sor) valamennyi néven végigmegy, és egyenként átadja őket a feldolgozást végző függvénynek. Amennyiben valamelyik fájl neve hibás, hibaüzenetet kapunk.

A számunkra igazán érdekes rész a 26. sorban kezdődik.

A 27. sorban létrehozunk egy – pillanatnyilag teljesen üres – átmeneti fájlt, amelynek a neve a `tactmp` karakterláncból és a héjprogram folyamatazonosítójából (`$$`) áll össze. A 28. sorral egyelőre ne foglalkozzunk.

A bemenetről érkező szöveget a 29–32. sorban látható `while` ciklus ebben az átmeneti fájlban egyszerűen összegyűjti , majd ha minden készen áll, a program ezúttal az átmeneti fájl nevével hívja meg a feldolgozó függvényt (33. sor).

Takarítás

Egy „rendes” program természetesen arra is fel van készítve, hogy az általa létrehozott „kacatokat” eltakarítsa maga után, még mielőtt kilépne. Mi sem egyszerűbb ennél, gondolhatnánk elsőre: ha megtörtént a feldolgozás, egyszerűen töröljük az átmeneti fájlt, és készen is vagyunk. Jelen esetben a 33. sor után kell kiadnunk az `rm /tmp/tactmp$$` parancsot. Látható azonban, hogy ez a törlés valamilyen furfangos módon már a 28. sorban is szerepel egy `trap` parancs részeként.

Ezt a megoldást az indokolja, hogy általában nem lehetünk biztosak benne, hogy a program végrehajtása valóban eljut a 33. sor utáni részig. A legegyszerűbb esetben előfordulhat például, hogy a felhasználó a feldolgozás kellős közepén megnyomja a Ctrl+C billentyűket, és megállítja a programot. Ilyenkor az átmeneti fájl minden igyekezetünk ellenére megmarad, hiszen a programnak esélye sincs rá, hogy törölje. Hasonló a helyzet, ha a futó programot a `kill` paranccsal „kilőjük”, vagy a rendszergazda – esetleg az áramszolgáltató és a szünetmentes tápegység – kifürkészhetetlen kegyéből a program végrehajtása során rendszerleállás következik be.

Végzetszerűségük ellenére az efféle események nem teljesen váratlanul következnek be. A program előbb kap egy értesítést, egy úgynevezett jelet a rendszermagtól, és valójában erre való válaszul fejezi be a működését. Mármost, ha van előjele a „katasztrófának”, akkor utolsó intézkedésként tehetünk még valamit, amivel menthetjük a menthetőt – például törölhetjük az átmeneti fájlt.

A jelek kezelése

A jelek elfogására és kezelésére szolgál a `trap` parancs, amelynek általános alakja a következő:

```
trap 'parancsok' jelek neve vagy száma
```

A `trap` a felsorolásban megadott jeleket elfogja, és hatásukra az egyszeres idézőjelek (aposztrófok) között megadott utasítássort hajtja végre. A legfontosabb jelek számát, nevét és leírását a mellékelt táblázat tartalmazza.

A `trap` csak attól a ponttól kezdve érvényes a programban, ahol ő maga szerepel. Ha tehát valamilyen globális, a program egészére vonatkozó műveletet akarunk végrehajtani egy `trap` részeként, azt célszerű rögtön a kód elején megadni. Ha viszont csak egy adott blokkban akarunk jelkezelést megvalósítani (esetünkben ez a helyzet), akkor a kérdéses műveleteket elegendő ennek a blokknak az elején szerepeltetni. A blokkból kilépve a különleges jelkezelést olyan módon meg is szüntethetjük, hogy egy műveletek nélküli `trap` parancsot adunk ki az adott jel számával vagy nevével.

Programunkban az egyetlen olyan művelet, amit „végszükség esetén” is végre akarunk hajtani: az átmeneti fájl törlése. Ennek megfelelően a 28. sorban látható `trap` parancs valamennyi, a program leállásához vezető jelet elfogja, végrehajtja a törlést, majd az 1-es visszatérési értékkel kilép. Az utóbbira azért van szükség, mert egy jel kezelése nem jelenti önműködően a program leállását. Ha tehát a `trap` részeként nem hajtunk végre kilépést, akkor a program tovább fut!

Azt is érdemes megfigyelni, hogy a 28. sorban a jelek felsorolása nem tartalmazza a 0-s jelet, vagyis a program szokványos leállását. Erre azért van szükség, mert ha a program a szokásos módon áll le, vagyis egyszerűen vége szakad, akkor előtte biztosan törölte az átmeneti fájlt. Így viszont a `trap`-ben szereplő `rm` parancs hibát jelezne.

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.