Linuxos jelzések alkalmazásfejlesztői szemszögből

A jelzések alapvető eszközt jelentenek a folyamatok közötti adatcserében, és a hálózati kiszolgálóktól kezdve a médialejátszókig gyakorlatilag mindenben alkalmazzák őket. Tanuld meg te is, hogyan használhatsz jelzéseket a saját programjaidban!

A jelzések működésének pontos megértése elengedhetetlen a linuxos környezetben dolgozó alkalmazáskészítők számára. A jelzéskezelés, illetve a vele kapcsolatos függvények ismerete révén a fejlesztők hatékonyabban végezhetik a munkájukat.

Az alkalmazások futtatása – ha minden utasítás rendben lefut – lépések egymásutánját jelenti. Ha hiba vagy rendellenesség történik egy program futása közben, a rendszermag jelzések segítségével értesítheti erről a folyamatot. A jelzéseket régebben a folyamatok közötti adatcserére és összehangolásra, valamint a folyamatok közötti adatcserék (IPC) egyszerűsítésére is használták. Noha ma már sokkal fejlettebb összehangoló eszközök és IPC-módszerek is rendelkezésünkre állnak, Linux alatt a jelzések továbbra is rendkívül fontos szerepet játszanak a kivételek és megszakítások kezelésében. A jelzéseket körülbelül harminc éven át lényegesebb módosítás nélkül használták. Az első 31 jelzés szabványos, ezek némelyike még az 1970-es évekből, a Bell Laboratórium Unix fejlesztéseiből származik. A Posix (Portable Operating Systems and Interface for UNIX) szabvány új, valós idejű jelzésosztályt vezetett be, amelynek tagjai 32-től 63-ig számozódnak.

Ha hiba történik, a rendszer jelzést hoz létre, majd az eseményt a rendszermag átadja a fogadó folyamatnak. Egyes esetekben jelzést folyamat is küldhet egy másik folyamatnak. A folyamat-folyamatjelzések mellett számos olyan helyzet van, amikor a jelzés a rendszermagtól ered: ilyen például, ha egy fájl mérete meghaladja az előírt határt, ha egy be-, illetve kiviteli eszköz készen áll, ha a rendszer érvénytelen utasítást hajtott végre, vagy ha egy terminál Ctrl-C vagy Ctrl-Z megszakítást küld. Minden jelzés neve SIG-gel kezdődik, és egyedi, pozitív egész számként van megadva. A héj parancssorában a `kill -l` parancsot kiadva az összes jelzés nevét és sorszámát megjeleníthetjük. A jelzések számai a */usr/include/bits/signum.h* állományban vannak megadva, ennek forrása a */usr/src/linux/kernel/signal.c*. Egy folyamat akkor kap jelzést, ha felhasználói módban fut. Ha a fogadó folyamat magmódban fut, akkor a jelzés végrehajtása csak akkor kezdődik meg, ha a folyamat újra felhasználói módba vált át.

A nem futó folyamatoknak küldött jelzéseket a rendszermagnak kell mentenie, amíg az adott folyamat futtatása újra nem indul. Az alvó folyamatok lehetnek megszakíthatók és nem megszakíthatók. Ha egy folyamat megszakítható alvó állapot mellett kap jelzést (például éppen terminál be-, illetve kivitelre vár), akkor a rendszermag felébreszti, hogy a folyamat kezelhesse a jelzést. Ha egy folyamat nem megszakítható állapotban kap jelzést (például lemezműveletre vár), a rendszermag a művelet befejeződéséig visszatartja a jelzést.

Ha egy folyamat jelzést kap, három dolog történhet. Az első lehetőség, hogy a folyamat figyelmen kívül hagyja a jelzést. A második, hogy elfogja a jelzést, és egy különleges, jelzéskezelőnek nevezett függvényt hív meg. A harmadik lehetőség az, hogy végrehajtja a jelzéshez rendelt alapértelmezett műveletet. A 15-ös számú `SIGTERM` jelzéshez például a folyamat befejezése tartozik mint alapértelmezett művelet. Egyes jelzések nem hagyhatók figyelmen kívül, másokhoz viszont nem tartoznak alapértelmezett műveletek, így ez utóbbiaknál a figyelmen kívül hagyás az alapértelmezett művelet. A jelzésnevek, -azonosítók és az alapértelmezett műveletek listáját, illetve azt, hogy mely jelzések foghatók el, a signal(7) súgóoldalon találhatod meg.

Ha újabb jelzés érkezik, miközben egy folyamat egy jelzéskezelőt futtat, az újabb jelzést a rendszer addig visszatartja, amíg a jelzéskezelő futása véget nem ér. A cikk további részei a jelzésekkel kapcsolatos alapokat, függvényeket, valamint alkalmazásukat tárgyalják.

Jelzések a rendszermagon belül

Vajon hol tároljuk a jelzésekkel kapcsolatos adatokat a folyamatban? A rendszermag egy kötött méretű, `proc` adatszerkezetekből álló tömbbel rendelkezik, ez a folyamattábla. A `proc` adatszerkezetek `u` (user, felhasználói) területe tárolja a folyamatokkal kapcsolatos vezérlőadatokat. Az `u` terület fontosabb mezői tartalmazzák – többek között – a jelzéskezelőket és a kapcsolódó adatokat. A jelzéskezelő egy tömb, elemei a rendszerben megadott jelzésekhez tartoznak, és az adott folyamat által az adott jelzés fogadásakor végrehajtott műveletet jelölik ki. A `proc` adatszerkezet jelzéskezelési adatokat tart fenn, lényegében a figyelmen kívül hagyandó, gátolandó, átadandó és kezelendő jelzések maszkjait.

Miután a rendszer jelzést hozott létre, a rendszermag egy bitet állít be a folyamattábla bejegyzésének jelzésmezőjében. Ha a jelzést figyelmen kívül hagyjuk, a rendszermag további műveleteket már nem hajt végre. Mivel a jelzésmező jelzésenként egy bitet jelent, ugyanannak a jelzésnek többszöri előfordulását a rendszer nem tartja nyilván.

A jelzés átadásakor a fogadó folyamatnak a jelzéstől függően kell cselekednie. A végrehajtott művelet lehet a folyamat befejezése, memóriakiíratás és a folyamat befejezése, a jelzés figyelmen kívül hagyása, a felhasználó által megadott jelzéskezelő meghívása (ha a folyamat elfogja a jelzést) vagy a folyamat futtatásának folytatása, ha ideiglenesen fel volt függesztve.

A memóriakiíratás egy *core* nevű fájlba történik, ami a befejeződött folyamat képét tartalmazza. Megtalálni benne a folyamat változóit, illetve a veremnek a hiba pillanatában fennállt állapotát. A *core* fájlból a programozó egy hibakeresővel kinyomozhatja a folyamat befejeződésének okát. A core (mag) megnevezés itt történelmi okokra vezethető vissza: régen a központi memória fánk alakú, induktormagoknak nevezett mágnesekből állt. Egy jelzés elfogása azt jelenti, hogy jelezzük a rendszermagnak: ha valamilyen jelzés lép fel, akkor az alapértelmezett helyett a program saját jelzéskezelőjét kell futtatnia. Két kivétel van: a `SIGKILL` és a `SIGSTOP`, ezeket nem lehet elfogni vagy figyelmen kívül hagyni.

1. lista Jelzés elfogása és figyelmen kívül hagyása

```
#include <signal.h>

void saját_kezelő (int sig);
/* a függvény prototípusa */

int main ( void ) {

/* 1. rész: SIGINT elfogása */
     signal (SIGINT, saját_kezelő);
     printf ("SIGINT elfogása\n");
     sleep(3);
     printf ("3 másodpercen belül nem
              ➥érkezett SIGINT\n");

/* 2. rész: SIGINT figyelmen kívül hagyása */
     signal (SIGINT, SIG_IGN);
     printf ("SIGINT figyelmen kívül
              ➥hagyása\n");
     sleep(3);
     printf ("3 másodpercen belül nem
              ➥érkezett SIGINT\n");

/* 3. rész: Alapértelmezett művelet SIGINT-
hez */
     signal (SIGINT, SIG_DFL);
     printf ("Alapértelmezett művelet
              ➥SIGINT-hez\n");
     sleep(3);
     printf ("3 másodpercen belül nem
              ➥érkezett SIGINT\n");
     return 0;
}

/* Felhasználó által megadott jelzéskezelő
   függvény */
void saját_kezelő (int sig) {
     printf ("SIGINT érkezett, azonosító:
              ➥%d\n", sig);
     exit(0);
}
```

A `sigset_t` egy alap-adatszerkezet a jelzések tárolására. A folyamatoknak küldött `sigset_t` adatszerkezet egy olyan bittömb, amiben minden jelzéstípushoz egy-egy bit tartozik:

```
typedef struct {
                unsigned long sig[2];
             }  sigset_t;
```

Mivel minden előjel nélküli `long` szám 32 bites, Linux alatt – a Posix-szabványnak megfelelően – legfeljebb 64 jelzés adható meg. A nullához nem tartozik jelzés, így a `sigset_t` első felének további 31 eleme az első szabványos 31 jelzéshez tartozik, a második rész bitjei pedig a 32–64. sorszámú, valós idejű jelzésekhez rendelődnek. A `sigset_t` mérete 128 bájt.

A jelzések kezelése

Számos rendszerhívás és jelzéstámogatott könyvtári függvény létezik, amelyek segítségével egy-egy folyamaton belül könnyedén megoldható a jelzések kezelése. Elsőként a jó öreg `signal` rendszerhívással ismerkedünk meg, majd további hasznos függvényekről is szó esik, mint a `sigaction`, `sigaddset`, `sigemptyset`, `sigdelset`, `sigismember` és a `kill`.

2. lista Megegyezik az elsővel, de sigaction hívásokat használ

```
#include <signal.h>
#include <stdio.h>

void saját_kezelő (int sig);
/* a függvény prototípusa */

int main ( void ) {

     struct sigaction saját_művelet;

/* 1. rész: SIGINT elfogása */

     saját_művelet.sa_handler = saját_kezelő;
     saját_művelet.sa_flags = SA_RESTART;
     sigaction (SIGINT, &saját_művelet, NULL);
     printf ("SIGINT elfogása\n");
     sleep(3);
     printf ("3 másodpercen belül nem
              ➥érkezett SIGINT\n");


/* 2. rész: SIGINT figyelmen kívül hagyása */

    saját_művelet.sa_handler = SIG_IGN;
    saját_művelet.sa_flags = SA_RESTART;
    sigaction (SIGINT, &saját_művelet, NULL);
    printf ("SIGINT figyelmen kívül
             ➥hagyása\n");
    sleep(3);
    printf (" Az alvó állapot (sleep)
             ➥véget ért \n");


/* 3. rész: alapértelmezett művelet SIGINT-hez */

   saját_művelet.sa_handler = SIG_DFL;
   saját_művelet.sa_flags = SA_RESTART;
   sigaction (SIGINT, &saját_művelet, NULL);
   sleep(3);
   printf ("3 másodpercen belül nem érkezett
            ➥SIGINT\n");
}

void saját_kezelő (int sig) {
     printf ("SIGINT érkezett, azonosító:
              ➥%d\n", sig);
     exit(0);
}
```

A signal rendszerhívás

A `signal` rendszerhívást az adott jelzés elfogására, figyelmen kívül hagyására, valamint a hozzárendelt alapértelmezett művelet megadására használjuk. Két átadott értéket vár: a jelzés számát és egy mutatót a felhasználó által megadott jelzéskeze-

3. lista Az SA_SIGINFO és az sa_sigaction használata adatok kinyerésére egy jelzésből

```
#include <unistd.h>
#include <sys/types.h>
#include <signal.h>
#include <bits/siginfo.h>
#include <stdio.h>

void kezelő (int signo, siginfo_t *info,
             ➥void *context);

main () {

   struct sigaction saját_művelet;

   saját_művelet.sa_flags = SA_SIGINFO;
   saját_művelet.sa_sigaction = kezelő;

   sigaction (SIGINT, &saját_művelet, NULL);

   printf ("SIGINT elfogása\n");
   sleep(5);
   printf ("Kész.\n");
}

void kezelő (int signo, siginfo_t *info,
             ➥void *context)
 {
    printf ("Jelzés azonosítója: %d\n",
             ➥info->si_signo);

 /* A siginfo_t adatszerkezet elemeit a
    man 2 sigaction ismerteti. */
}
```

4. lista SIGINT-et fogadó és küldő programok

```
#include <signal.h>

main ( ) {
    int folyamat_azonosító;
    printf ("Add meg annak a folyamatnak
             ➥az azonosítóját amelynek a
             ➥jelzést akarod küldeni : ");
    scanf ("%d", &folyamat_azonosító);

   if (!(kill ( folyamat_azonosító, SIGINT)))
       printf ("SIGINT elküldve a következő
                ➥azonosítójú folyamatnak:
                ➥%d\n", folyamat_azonosító);
   else if (errno == EPERM)
       printf ("A művelet nincs
                ➥engedélyezve.\n");
   else
       printf ("%d nem létezik \n",
                ➥folyamat_azonosító);
}

/* 4a kódrészlet Ez a program addig fut,
   amíg SIGINT-et nem kap. */

#include <signal.h>

 main ( ) {
   printf (" Ennek a folyamatnak az
            ➥azonosítója: %d. "
   " SIGINT várása.\n", getpid());
   for (;;);
}
```

lőre. Linux alatt két fenntartott, előre megadott jelzéskezelő áll rendelkezésünkre, a `SIG_IGN` és a `SIG_DFL`. A `SIG_IGN` a megadott jelzést figyelmen kívül hagyja, a `SIG_DFL` pedig az adott jelzésre vonatkozóan az alapértelmezettet állítja be jelzéskezelőnek (lásd: `man 2 signal`).
Siker esetén a rendszerhívás az adott jelzés jelzéskezelőjének korábbi értékével tér vissza. Ha a `signal` hívás sikertelen, visszatérési értéke `SIG_ERR`. Az *1. lista* 1. kódrészletében a `SIGINT` elfogásának, figyelmen kívül hagyásának és alapértelmezett műveletének megadása látható. Minden résznél nyomd le a CTRL-C billentyűkombinációt, ez `SIGINT` jelzést vált ki.

sigaction

A `sigaction` rendszerhívás a `signal` helyett használható; szélesebb körű vezérlési lehetőségeket ad a jelzésekhez. Írásmódja a következő:

```
int sigaction ( int signum, const struct
                ➥sigaction *act,
                ➥struct sigaction *oldact);
```

Első átadott értéke (`signum`) egy adott jelzést jelöl, a második (`sigaction`) a `signum` jelzés új kezelőjének megadására szolgál, a harmadik pedig a korábbi kezelő – ez általában `NULL` – tárolását végzi.

A `sigaction` adatszerkezet felépítése a következő:

```
struct sigaction {
    void (*sa_handler)(int);
    void (*sa_sigaction)(int, siginfo_t *,
                         ➥void *);
    sigset_t sa_mask;
    int sa_flags;
}
```

A sigaction adatszerkezet tagjainak a leírása:

- `sa_hander` – mutató egy felhasználó által vagy előre megadott (`SIG_IGN` vagy `SIG_DFL`) jelzéskezelőre.
- `sa_maskn` – jelzések egy maszkját adja meg a jelzés kezelésekor. A jelzések gátlásának elkerülésére a `SA_NODEFER` vagy a `SA_NOMASK` jelző használható.
- `sa_flags` – jelzéshez tartozó műveletet ad meg. Több jelző is rendelkezésre áll a jelzés különféle módokon való kezelésére. Egynél több jelző `OR` művelettel használható:
  - `SA_NOCLDSTOP` – amennyiben megadjuk a `SIGCHLD` jelzést, ha a gyermek futása véget ért, nem kap jelzést.
  - `SA_ONESHOT` vagy `SA_RESETHAND` – a felhasználó által megadott jelzéskezelő végrehajtása után visszaállítja a jelzéshez tartozó alapértelmezett műveletet. Az alapértelmezett művelet beállításának megakadályozására az `SA_RESTART` használható.

- SA_NOMASK vagy SA_NODEFER – a jelzés maszkolását akadályozza meg. Az SA_SIGINFO a jelzéssel kapcsolatos adatok megszerzésére alkalmas.
- sa_sigaction – ha az SA_SIGINFO jelzőt használjuk az sa_flags-ben ahelyett, hogy a jelzéskezelőt az sa_handler-ben adnánk meg, akkor az sa_sigaction-t kell alkalmazni.

Az sa_sigaction mutató egy függvényre, ami három átadott értéket vár – ellentétben az sa_handler-rel, ami csak egyet –, például:

```
void saját_kezelő
➥(int signo, siginfo_t info, *void *context)
```

A signo ebben az esetben a jelzés száma, az info pedig egy mutató egy siginfo_t típusú adatszerkezetre, ami a jelzéssel kapcsolatos adatokat tartalmazza. A context szintén mutató egy ucontext_t típusú objektumra, ami arra a fogadó folyamat-összefüggésre hivatkozik, amit a jelzés megszakított.
A második kódrészlet hasonló az elsőhöz, de a signal helyett a sigaction rendszerhívást használja. A *3. lista* a jelzésekkel kapcsolatos adatokat ismerteti a SIG_INFO jelző segítségével.

Jelzések küldése

Eddig a CTRL-C billentyűkombinációval küldtünk SIGINT jelzést a héjból. Ha ezt programból szeretnénk megtenni, a kill rendszerhívást kell használnunk, aminek két értéket kell átadnunk: a folyamat azonosítóját és a jelzés számát.

```
int kill ( pid_t folyamat_azonosító,
           ➥int jelzés_száma );
```

Ha a folyamat azonosítója pozitív, a jelzést egy megadott folyamat kapja. Ha negatív, a jelzést a rendszer annak a folyamatnak küldi, amelynek a csoportazonosítója megegyezik a folyamatazonosító abszolút értékével.
Talán nem meglepő, hogy az önálló programként (/bin/kill) is létező, illetve a bash részeként is használható kill parancs (próbáld ki: help kill) a kill rendszerhívással küld jelzéseket. Nem minden folyamat küldhet jelzést a többinek. Ahhoz, hogy egy folyamat jelzést küldhessen egy másiknak, a küldőnek rendszergazdaként kell futtatnia, vagy a küldő valódi (effective) felhasználóazonosítójának ugyanannak kell lennie, mint a fogadó folyamat valódi vagy mentett azonosítójának. Ez azt jelenti, hogy a te héjad, ami a te jogosultságaiddal fut, jelzést tud küldeni egy általad indított, de éppen rendszergazdai módban futó setuid programnak:

```
# cp /bin/sleep ~/rootsleep
# sudo chmod u+s ~/rootsleep
# ./rootsleep 40
# killall rootsleep
# rm ~/rootsleep
```

Egy egyszerű felhasználó nem tud jelzést küldeni a rendszerfolyamatnak, például a swapper és az init.
A kill annak a megállapítására is alkalmas, hogy egy folyamat létezik-e vagy sem. Jelzésazonosítónak válaszd a 0-t; ha a folyamat létezik, a kill visszatérési értéke nulla, ellenkező esetben pedig -1 lesz.
A *4. lista* ***4a*** jelű kódrészlete a kill rendszerhívás használatát szemlélteti. Először indítsd el a 4a programot, és nézd meg a folyamatazonosítóját. Ezután – egy másik ablakban – indítsd el a 4. lista programját, és bemenetként a 4a program folyamatazonosítóját használd.

Linux Journal 2003. március, 107. szám

Dr. B. Thangaraju
Fizikából szerzett PhD fokozatot. Jelenleg a Wipro Technologies egyik vezetője, ugyancsak Indiában. Tekintélyes nemzetközi folyóiratokban több kutatási témájú írása is megjelent.

KAPCSOLÓDÓ CÍMEK

W. Richard Stevens Advanced Programming in the UNIX Environment, Pearson Education Asia, 1993, 263–324. oldal
Michael K. Johnson és *Erik W. Troan* Linux Application Development, Addison-Wesley, 1998
Moshe Bar „The Linux Signals Handling Model", Linux Journal, 2000. május 1. (elérhető a ➲ http://www.linuxjournal.com/article/3985 címen).
D. P. Bovet és *M. Cesati* Understanding the Linux Kernel, O'Reilly & Associates, 1998, 233–248 oldal