# Kísérletezgetés a ptrace-szel (1. rész)

A ptrace lehetővé teszi felhasználói szintű rendszerhívás-elfogó és -módosító rendszer kiépítését.

Kíváncsiskodtál már valaha, hogyan is lehet elfogni a rendszerhívásokat? Próbáltad már megbolondítani a rendszermagot a rendszerhívás értékeinek megváltoztatásával? Tettél-e már kísérletet annak megfejtésére, hogyan állítanak meg egy folyamatot a programfejlesztő eszközök, és miképp adják át a folyamat irányítását?
Ha a feladatok elvégzésére az összetett rendszermag-programozás jut eszedbe, akkor gondolkozz csak tovább. A Linux valamennyi feladat elvégzéséhez egy elegáns eszközt bocsát rendelkezésünkre, a ptrace-t (Process Trace System Call).
A ptrace olyan módszert használ, amelynek révén a szülőfolyamat egy másik folyamatot figyelhet meg és irányíthat. Ez az eszköz képes arra, hogy ellenőrizze és megváltoztassa a másik folyamat magképét és regisztereit, és elsődlegesen a töréspontok végrehajtásának ellenőrzésére, valamint a rendszerhívások nyomon követésére használják.
E cikkből azt fogjuk megtudni, hogyan kell egy rendszerhívást elfogni és a kapcsolóit megváltoztatni. A cikk második részében a haladó módszerekkel fogunk megismerkedni: az ellenőrzési pontok kijelölésével és a kódok futó programba történő beillesztésével. Bekukkantunk majd a gyermekfolyamatok regisztereibe és adatszakaszaiba, és módosítani fogjuk a tartalmukat. Meg fogjuk mutatni azt is, hogyan kell a programkódot úgy beültetni, hogy a folyamatot le lehessen állítani, és tetszőleges utasítást lehessen végrehajtani.

## Alapismeretek

Az operációs rendszerek szolgáltatásaikat szabványos módszereken, úgynevezett rendszerhívásokon keresztül ajánlják fel. Szabványos alkalmazói program felületet (API) kínálnak a készülékek kezelésére, valamint alacsony szintű szolgáltatásokat, ilyenek például az állományrendszerek. Amikor egy folyamat rendszerhívást szeretne használni, a rendszerhíváshoz szükséges kapcsolókat a regiszterekben helyezi el, és elindítja a 0x80-as lágy megszakítást. A lágy megszakítás olyan, mint valamiféle kapu, ami a rendszermag-üzemmódra nyílik, ahol a mag a kapcsolók ellenőrzése után végrehajtja a rendszerhívást. Az i386-os típusú gépeken – e cikkünkben minden programkód i386-os típusra készült – a rendszerhívás száma az `%eax`-regiszterbe kerül. Az ehhez a rendszerhíváshoz tartozó értékek pedig rendre az `%ebx`, `%ecx`, `%edx` regiszterekbe kerülnek, ebben a sorrendben. Például a `write(2, "Szia", 5)` rendszerhívás a következő utasításokká alakul át:

```
movl    $4, %eax
movl    $2, %ebx
movl    $szia,%ecx
movl    $5, %edx
int     $0x80
```

ahol a `$szia` jelsorozat a `"Szia"` karakteres állandót jelöli, vagyis szakkifejezéssel élve az egy karakteres literálra mutat. De hogyan kerül a képbe a ptrace? A rendszerhívás végrehajtása előtt a rendszermag ellenőrzi, hogy a folyamat nyomkövetés alatt áll-e. Ha igen, a rendszermag leállítja a folyamatot, és a vezérlést a nyomkövető folyamatnak adja át, hogy képes legyen ellenőrizni és módosítani az ellenőrzött folyamat regisztereit. Hogy megértsük a folyamat működését, világítsuk meg egy példával:

```
#include <sys/ptrace.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <unistd.h>
#include <linux/user.h>   /* A karakteres
Ælland k, pØldÆul ORIG_EAX stb. szÆmÆra */

int main()
{   pid_t child;
    long orig_eax;

    child = fork();
    if(child == 0) {
        ptrace(PTRACE_TRACEME, 0, NULL, NULL);
        execl("/bin/ls", "ls", NULL);
    }
    else {
        wait(NULL);
        orig_eax = ptrace(PTRACE_PEEKUSER,
                  child, 4 * ORIG_EAX, NULL);
        printf("A gyermekfolyamat Æltal
               ➥elind tott rendszerh vÆs:
               ➥%ld\n", orig_eax);
        ptrace(PTRACE_CONT, child, NULL, NULL);
    }
    return 0;
}
```

A program futásakor az alábbi üzenetet írja ki:
***A gyermekfolyamat által elindított rendszerhívás: 11***
az `ls` parancs kimenetével együtt. A 11-es rendszerhívás az `execve`, ez a gyermekfolyamat által végrehajtott első rendszerhívás. Tájékozódás végett: a rendszerhívások szám szerint megtalálhatók a ***/usr/include/asm/unistd.h*** állományban. A bemutatott példában látható, hogy a folyamat gyermekfolyamatot (child) hoz létre, s ez a gyermekfolyamat hajtja majd végre az általunk nyomon követni kívánt folyamatot. Az `exec` futtatása előtt a gyermekfolyamat az első értékkel, ami a `PTRACE_TRACEME`, meghívja a ptrace-t. Ez megmondja a rendszermagnak, hogy a folyamat nyomonkövetés alatt áll, és amikor a gyermekfolyamat végrehajtja az `execve` rendszerhívást, a vezérlést a szülőfolyamatnak (parent) adja át. A szülő értesítést vár a rendszermagtól, egy `wait()` függvényhívással. Ezt követően a szülő ellenőrzi a rendszerhívás értékeit vagy más tevékenységet folytat, mondjuk megnézi a regiszterek tartalmát.

Amikor a rendszerhívás megtörtént, a rendszermag menti az eax regiszter eredeti tartalmát, amely tartalmazza a rendszerhívás számát. Ezt az értéket a gyermekfolyamat USER (felhasználói) adatterületéről olvashatjuk ki, a ptrace első értékeként használt PTRACE_PEEKUSER-rel, ahogyan az fentebbi példánkban is szerepelt. Ha végeztünk a rendszerhívás vizsgálatával, a gyermekfolyamat folytathatja a ptrace meghívását az első értékként beállított PTRACE_CONT értékkel, ami lehetővé teszi, hogy a rendszerhívás tovább folytatódjon.

### ptrace-értékek

A ptrace programot négy értékkel hívják meg:

```
long ptrace(enum __ptrace_request request,
            pid_t pid,
            void *addr,
            void *data);
```

Az első érték meghatározza a ptrace viselkedését és előírja, hogyan kell a többi értéket használni.
A hívás értéke az alábbiak egyike kell legyen:

PTRACE_TRACEME, PTRACE_PEEKTEXT, PTRACE_PEEKDATA, PTRACE_PEEKUSER, PTRACE_POKETEXT, PTRACE_POKEDATA, PTRACE_POKEUSER, PTRACE_GETREGS, PTRACE_GETFPREGS, PTRACE_SETREGS, PTRACE_SETFPREGS, PTRACE_CONT, PTRACE_SYSCALL, PTRACE_SINGLESTEP, PTRACE_DETACH.

Minden egyes hívás értelmezését a cikk további részében fogjuk megadni.

### A rendszerhívás értékeinek kiolvasása

Ha a PTRACE_PEEKUSER-t használjuk a ptrace első értékeként, akkor megvizsgálhatjuk a USER terület tartalmát, ahol a regiszterek tartalma és egyéb adatok tárolása történik.
Lássunk erre is egy példát:

```
#include <sys/ptrace.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <unistd.h>
#include <linux/user.h>
#include <sys/syscall.h>   /* A SYS_write-hoz,
                              meg miegymÆshoz */

int main()
{   pid_t child;
    long orig_eax, eax;
    long params[3];
    int status;
    int insyscall = 0;

    child = fork();
    if(child == 0) {
        ptrace(PTRACE_TRACEME, 0, NULL, NULL);
        execl("/bin/ls", "ls", NULL);
    }
    else {
       while(1) {
          wait(&status);
          if(WIFEXITED(status))
              break;
          orig_eax = ptrace(PTRACE_PEEKUSER,
                  child, 4 * ORIG_EAX, NULL);
          if(orig_eax == SYS_write) {
             if(insyscall == 0) {
                /* Rendszerh vÆs belØpØsi
                   pontja */
                insyscall = 1;
                params[0] =
                ptrace(PTRACE_PEEKUSER,
                child, 4 * EBX, NULL);
                params[1] =
                ptrace(PTRACE_PEEKUSER,
                child, 4 * ECX, NULL);
                params[2] =
                ptrace(PTRACE_PEEKUSER,
                  child, 4 * EDX, NULL);

                printf("Ki ratÆs megh vÆsa a
                      ➥k vetkezı tartalommal:"
                        "%ld, %ld, %ld\n",
                        params[0], params[1],
                        params[2]);
                }
          else { /* Rendszerh vÆs vØge */
                eax = ptrace(PTRACE_PEEKUSER,
                        child, 4 * EAX, NULL);
                printf("Ki ratÆsi mßvelet
                      ➥visszatØrØsi k dja: "
                      "with %ld\n", eax);
                    insyscall = 0;
                }
             }
             ptrace(PTRACE_SYSCALL, child,
                   NULL, NULL);
          }
     }
     return 0;
}
```

Ez a program az alábbi kimenethez hasonlót fog mutatni:

```
ppadala@linux:~/ptrace > ls
a.out          dummy.s       ptrace.txt
libgpm.html    registers.c   syscallparams.c
dummy          ptrace.html   simple.c

ppadala@linux:~/ptrace > ./a.out
Ki ratÆs megh vÆsa a k vetkezı tartalommal:
  ➥1, 1075154944, 48
a.out          dummy.s       ptrace.txt
Ki ratÆsi mßvelet visszatØrØsi k dja: 48
Ki ratÆs megh vÆsa a k vetkezı tartalommal:
  ➥1, 1075154944, 59
libgpm.html    registers.c   syscallparams.c
Ki ratÆsi mßvelet visszatØrØsi k dja: 59
Ki ratÆs megh vÆsa a k vetkezı tartalommal:
  ➥1, 1075154944, 30
dummy          ptrace.html   simple.c
Ki ratÆsi mßvelet visszatØrØsi k dja: 30
```

Itt az írási művelethez (write) kapcsolódó rendszerhívásokat követjük nyomon, az ls pedig három rendszerhívást fog elindítani. A ptrace indításánál első értékként használt

PTRACE_SYSCALL hatására a rendszermag, minden esetben megállítja a gyermekfolyamatot, ha rendszerhívás-belépési vagy -visszatérési pont keletkezik. Ez egyenértékű a PTRACE_CONT végrehajtásával, és a következő rendszerhívási belépési vagy visszatérési pontnál való megállással. Az előző példában a PTRACE_PEEKUSER-t használtuk, hogy betekintést nyerjünk a rendszerhívás értékeibe. Amint a rendszerhívás véget ér, a visszatérési érték a %eax regiszterbe kerül, és onnan kiolvasható, amint ezt az adott példában is láthattuk.
A várakoztatási rendszerhívásban (wait call) az állapotváltozót arra használhatjuk, hogy ellenőrizzük, vajon véget ért-e már a gyermekfolyamat. Ez az ellenőrzés jellegzetes módja, mivel így megvizsgálható, hogy a ptrace állította-e le a folyamatot, vagy az maga képes volt-e sikeresen befejeződni. Az olyan makrók, mint a WIFEXITED, részletes leírása elolvasható a wait(2) man oldaláról.

## A regiszterek értékeinek kiolvasása

Ha a regiszterek értékeit rendszerhívás idején vagy annak visszatérésekor szeretnénk kiolvasni, a fent bemutatott eljárás kényelmetlen lehet. Ha a ptrace első értékeként a PTRACE_GETREGS-et használjuk, akkor ez a művelet valamennyi regisztert egyetlen rendszerhívásba fogja elhelyezni. A regiszterek értékét kiolvasó programot valahogy így kell elképzelni:

```
#include <sys/ptrace.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <unistd.h>
#include <linux/user.h>
#include <sys/syscall.h>

int main()
{   pid_t child;
    long orig_eax, eax;
    long params[3];
    int status;
    int insyscall = 0;
    struct user_regs_struct regs;

    child = fork();
    if(child == 0) {
        ptrace(PTRACE_TRACEME, 0, NULL, NULL);
        execl("/bin/ls", "ls", NULL);
    }
    else {
       while(1) {
          wait(&status);
          if(WIFEXITED(status))
              break;
          orig_eax = ptrace(PTRACE_PEEKUSER,
                      child, 4 * ORIG_EAX,
                      NULL);
          if(orig_eax == SYS_write) {
              if(insyscall == 0) {
                 /* Rendszerh vÆs belØpØsi
                  * pontja */
                 insyscall = 1;
                 ptrace PTRACE_GETREGS,
                       child, NULL, &regs);
                 printf("Write called with "
                       "%ld, %ld, %ld\n",
                       regs.ebx, regs.ecx,
                                 regs.edx);
              }
              else { /* Rendszerh vÆs
                      * visszatØrØse */
                  eax = ptrace(PTRACE_PEEKUSER,
                         child, 4 * EAX,
                         NULL);
                  printf("Write returned "
                         "with %ld\n", eax);
                  insyscall = 0;
              }
           }
           ptrace(PTRACE_SYSCALL, child,
                  NULL, NULL);
       }
    }
    return 0;
}
```

Ez utóbbi példánk hasonlít az előzőhöz, a ptrace-nek a PTRACE_GETREGS értékkel együtt történő meghívását kivéve. Itt a <linux/user.h>-ban meghatározott user_regs_struct szerkezetet használtuk a regiszterek tartalmának kiolvasására.

## Néhány érdekesség

Elérkezett az ideje annak, hogy kipróbáljunk egy kis vicces furcsaságot. A következő példában meg fogjuk fordítani a rendszerhívásnak átadott karakterláncot:

```
#include <sys/ptrace.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <unistd.h>
#include <linux/user.h>
#include <sys/syscall.h>

const int long_size = sizeof(long);

void reverse(char *str)
{   int i, j;
    char temp;

    for(i = 0, j = strlen(str) - 2;
        i <= j; ++i, --j) {
        temp = str[i];
        str[i] = str[j];
        str[j] = temp;
    }
}

void getdata(pid_t child, long addr,
             char *str, int len)
{   char *laddr;
    int i, j;
    union u {
            long val;
            char chars[long_size];
    }data;

    i = 0;
    j = len / long_size;
    laddr = str;
```

```
    while(i < j) {
        data.val = ptrace(PTRACE_PEEKDATA,
                          child, addr + i * 4,
                          NULL);
        memcpy(laddr, data.chars, long_size);
        ++i;
        laddr += long_size;
    }

    j = len % long_size;
    if(j != 0) {
        data.val = ptrace(PTRACE_PEEKDATA,
                          child, addr + i * 4,
                          NULL);
        memcpy(laddr, data.chars, j);
    }
    str[len] = '\0';
}

void putdata(pid_t child, long addr,
             char *str, int len)
{   char *laddr;
    int i, j;
    union u {
            long val;
            char chars[long_size];
    }data;

    i = 0;
    j = len / long_size;
    laddr = str;
    while(i < j) {
        memcpy(data.chars, laddr, long_size);
        ptrace(PTRACE_POKEDATA, child,
               addr + i * 4, data.val);
        ++i;
        laddr += long_size;
    }

    j = len % long_size;
    if(j != 0) {
        memcpy(data.chars, laddr, j);
        ptrace(PTRACE_POKEDATA, child,
               addr + i * 4, data.val);
    }
}

int main()
{
   pid_t child;

   child = fork();
   if(child == 0) {
      ptrace(PTRACE_TRACEME, 0, NULL, NULL);
      execl("/bin/ls", "ls", NULL);
   }
   else {
      long orig_eax;
      long params[3];
      int status;
      char *str, *laddr;
      int toggle = 0;

      while(1) {
         wait(&status);
         if(WIFEXITED(status))
             break;
         orig_eax = ptrace(PTRACE_PEEKUSER,
                           child, 4 *
                           ORIG_EAX, NULL);

         if(orig_eax == SYS_write) {
            if(toggle == 0) {
               toggle = 1;

               params[0] =
               ptrace(PTRACE_PEEKUSER,
                child, 4 * EBX, NULL);
               params[1] =
               ptrace(PTRACE_PEEKUSER,
                child, 4 * ECX, NULL);
               params[2] =
               ptrace(PTRACE_PEEKUSER,
                child, 4 * EDX, NULL);

               str = (char *)calloc((params[2]+1)
                      * sizeof(char));
               getdata(child, params[1], str,
                      params[2]);
               reverse(str);
               putdata(child, params[1], str,
                      params[2]);
            }
            else {
               toggle = 0;
            }
         }
         ptrace(PTRACE_SYSCALL, child, NULL, NULL);
      }
   }
   return 0;
}
```

A program az alábbi kimenetet fogja mutatni:

```
ppadala@linux:~/ptrace > ls
a.out            dummy.s          ptrace.txt
libgpm.html      registers.c      syscallparams.c
dummy            ptrace.html      simple.c
ppadala@linux:~/ptrace > ./a.out
txt.ecartp       s.ymmud          tuo.a
c.sretsiger      lmth.mpgbil      c.llacys_egnahc
c.elpmis         lmth.ecartp      ymmud
```

Ez a példaprogram a fentebb bemutatott elgondolásokat hasznosítja, sőt rajtuk kívül még néhány másikat is. Használjuk benne az adatok megváltoztatását végző PTRACE_POKEDATA értékkel kiegészített ptrace-hívásokat. Ez a program a PTRACE_PEEKDATA-val teljesen azonos módon működik, ez azonban olvassa és írja is azokat az adatokat, amelyeket a gyermekfolyamat az értékekben a rendszerhívásnak átad, miközben a PEEKDATA csak az adatok olvasására vállalkozik.

## Lépésenkénti végrehajtás

A ptrace szolgáltatást biztosít a gyermekfolyamat kódjának lépésenkénti végrehajtásához.
A ptrace PTRACE_SINGLESTEP értékkel együtt történő meg-

hívása jelzi a rendszermagnak, hogy minden egyes utasításnál állítsa meg a gyermekfolyamatot, és adja át a vezérlést a szülőfolyamatnak. Az alábbi példában az éppen végrehajtás alatt álló utasítás kódjának olvasását láthatjuk, miközben rendszerhívás végrehajtása zajlik. Létrehoztam egy kisméretű, semmi különöset sem tevő példaprogramot (dummy), hogy bemutassam, mi is történik valójában, ahelyett, hogy a libc-hívásokkal kellene bajlódnunk. A következőkben a ***dummy1.s*** program listáját olvashatjuk. A program gépi nyelven íródott, fordítása a `gcc -o dummy1 dummy1.s` paranccsal történt.

```
.data
szia:
     .string "Szia vilÆg! \n"

.globl  main
main:
     movl     $4, %eax
     movl     $2, %ebx
     movl     $hello, %ecx
     movl     $12, %edx
     int      $0x80
     movl     $1, %eax
     xorl     %ebx, %ebx
     int      $0x80
     ret
```

A példaprogram, ami a fenti program lépésenkénti végrehajtását végzi, a következő:

```
#include <sys/ptrace.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <unistd.h>
#include <linux/user.h>
#include <sys/syscall.h>

int main()
{    pid_t child;
     const int long_size = sizeof(long);

     child = fork();
     if(child == 0) {
          ptrace(PTRACE_TRACEME, 0, NULL, NULL);
          execl("./dummy1", "dummy1", NULL);
     }
     else {
          int status;
          union u {
               long val;
               char chars[long_size];
          }data;
          struct user_regs_struct regs;
          int start = 0;
          long ins;

          while(1) {
               wait(&status);
               if(WIFEXITED(status))
                    break;
               ptrace(PTRACE_GETREGS,
                      ➥child, NULL, &regs);
               if(start == 1) {
                    ins = ptrace(PTRACE_PEEKTEXT,
                           child, regs.eip,
                           NULL);
                    printf("EIP: %lx A
                          vØgrehajtott "
                          "utas tÆs: %lx\n",
       regs.eip, ins);
               }

               if(regs.orig_eax == SYS_write) {
                    start = 1;
                    ptrace(PTRACE_SINGLESTEP,
                          child, NULL, NULL);
               }
               else
                    ptrace(PTRACE_SYSCALL, child,
                           NULL, NULL);
          }
     }
     return 0;
}
```

A program kimeneteként az alábbiakat kapjuk:

```
Szia vilÆg!
EIP: 8049478 A vØgrehajtott utas tÆs: 80cddb31
EIP: 804947c A vØgrehajtott utas tÆs: c3
```

Az utasításbájtok értelmezéséhez szükség lehet az Intel kézikönyveire. Az olyan összetettebb folyamatok lépésenkénti végrehajtása, mint amilyen a töréspontok kijelölése, gondos tervezést és összetettebb forráskódot igényel.
Sorozatunk második részében meg fogjuk vizsgálni, hogyan lehet töréspontokat kijelölni és egy már működő programba programkódokat beültetni.

***A cikk mostani és második részéhez tartozó forráskódok a 43. CD Magazin/Ptrace könyvtárában találhatók.***

*Linux Journal 2002. november, 103. szám*

***Pradeep Padala*** (p_padala@yahoo.com)
Jelenleg a Floridai Egyetemen diplomája megszerzésén munkálkodik. Érdeklődési területei között a rácsos kiépítésű és osztott rendszerek szerepelnek. Honlapját a ➲ http://www.cise.ufl.edu/~ppadala címen lehet elérni.

## KAPCSOLÓDÓ CÍMEK

Az operációs rendszer kiterjesztése felhasználói szinten: az UFO globális állományrendszer
➲ http://www.cs.ucsb.edu/projects/ufo/97-usenix-ufo.ps
A ptrace súgóoldalán„A biztonságos, felhasználói szintű, korlátozott erőforrás-igényű linuxos gép (Sandboxing)”
➲ http://csdocs.cs.nyu.edu/Dienst/Repository/2.0/Body/ncstrl.nyu_cs%2fTR1999-795/pdf