Miért és hogyan használjunk Netlink Socket-eket

Használjuk kétirányú, alakítható módszert a rendszermag és a felhasználói tér közötti adatátadásra.

A rendszermag fejlesztése és karbantartása elég komoly feladat, ezért csak a legszükségesebb és teljesítmény szempontból legfontosabb kódok kerülnek a magba. Más részek, mint a ***GUI***, kezelési és vezérlő kód, általában felhasználói térben futó programként készülnek. Elég általános gyakorlat ***Linux*** alatt, hogy egy bizonyos képességet megvalósításkor felosztanak a rendszermag és a felhasználói tér között. Most már csak az a kérdés, hogy a rendszermag kód és a felhasználói tér hogyan kommunikálnak egymás között?
A válasz a rendszermag és a felhasználói tér közötti különféle ***IPC*** módszerekben rejlik. Ilyen a rendszerhívás, az ***ioctl***, a ***proc*** fájlrendszer vagy a ***netlink socket***. Ez a cikk a ***netlink sockettel*** foglalkozik és megpróbál rávilágítani ennek a hálózatbarát ***IPC***-nek az előnyeire.

Bevezetés

A ***netlink socket*** egy különleges ***IPC***, melyet a rendszermag és a felhasználói térbeli folyamatok közötti információcserére használunk. Teljes kétirányú kommunikációs csatornát létesít a két rész között ahol a felhasználói folyamatok szabványos ***socket API***-t a rendszermag modulok pedig egy különleges ***API***-t használnak. A ***netlink socket*** az `AF_NETLINK` címcsaládot használja szemben a ***TCP/IP*** socketek által használt `AF_INET` címekkel. Minden ***netlink socket*** képesség a ***include/linux/netlink.h*** rendszermag fejléc fájlban definiálja saját protokolltípusát.
Az alábbiakban bemutatunk a ***netlink socket*** által támogatott néhány képességet és protokolltípust:

- `NETLINK_ROUTE`: a felhasználói tér útvonalválasztó démonai között használt kommunikációs csatorna. Ilyen például a ***BGP***, ***OSPF***, ***RIP*** és a rendszermag csomagtovábbító modul. A felhasználói tér útvonalválasztó démonai a rendszermag útvonalválasztó tábláját frissítik ezzel a netlink protokolltípussal.
- `NETLINK_FIREWALL`: Az ***IPv4*** tűzfal kód által küldött csomagokat fogadja.
- `NETLINK_NFLOG`: a felhasználói tér ***iptable*** kezelő eszközei és rendszermag tér ***Netfilter*** modulja közötti kommunikációs csatorna.
- `NETLINK_ARPD`: az ***arp*** tábla kezelése felhasználói térből.

De miért használnak a fentiek ***netlink*** kapcsolatot rendszerhívás, ***ioctl*** vagy ***proc*** fájlrendszer helyett a felhasználói és a rendszermag világ közötti kommunikációhoz? Az új képességekhez tartozó rendszerhívások, ***ioctl***-ek vagy ***proc*** fájlok elkészítése egyáltalán nem egyszerű; sőt azt kockáztatjuk, hogy belerondítunk a rendszermagba és ezzel az egész rendszer stabilitását veszélyeztetjük. A ***netlink socket*** viszont egyszerű: egyetlen konstanst, a protokoll típust kell a ***netlink.h***-hoz fűznünk. Ezáltal a rendszermag modul és az alkalmazás egy ***socket***-szerű API-n keresztül máris képes a kapcsolattartásra.
A ***netlink*** aszinkron jellegű, ugyanis, akárcsak a a többi ***socket API***, egy ***socket*** sort használ az üzenet rohamok csillapítására. A ***netlink*** üzenetet küldő rendszerhívás az üzenetet a fogadó üzenetsorába helyezi, majd meghívja a fogadó átvétel kezelőjét. A fogadó az ***átvételkezelő környezetben (context)***futva eldöntheti, hogy azonnal feldolgozza-e az üzenetet, vagy a sorban hagyva inkább később, más környezetben dolgozza azt fel. A ***netlink***-el ellentétben, a rendszerhívások szinkron feldolgozást igényelnek. Következésképpen, ha a felhasználói térből rendszerhívással adunk át egy üzenetet a rendszermagnak, a rendszermag ütemezési felbontására is hatással lehetünk amennyiben az üzenet feldolgozása kellően hosszú ideig tart.
A rendszermagban a rendszerhívásokat megvalósító kód fordítási időben kerül a magba; ezért rendszerhívás kódot nem tehetünk betölthető modulba, pedig a legtöbb eszközvezérlő ezen az elven alapszik. A ***netlink socket*** esetében nincs fordítási idejű függőség a ***Linux netlink*** magja és a betölthető modulokban található ***netlink*** alkalmazások között. A ***netlink socket*** támogatja a ***multicasot***, ez újabb előnyt jelent a rendszerhívásokkal, ***ioctl*** és ***proc*** megoldásokkal szemben. A folyamat ***multicast*** üzenetet küldhet egy ***netlink*** csoportcímre, melyre tetszőleges számú más folyamat hallgathat. Ez a rendszermagból a felhasználói térbe irányuló majdnem tökéletes eseményelosztási lehetőséget biztosít számunkra.
A rendszerhívások és az ***ioctl IPC***-k egyirányúak abban az értelemben, hogy ilyen ***IPC***-t csak felhasználói térben futó alkalmazások kezdeményezhetnek. De mit tegyünk ha egy rendszermag modul szeretne sürgős üzenetet küldeni a felhasználói tér alkalmazásainak? Ezekkel az ***IPC***-kel ezt nem tudjuk

közvetlenül megvalósítani. Az alkalmazásnak általában adott időnként le kell kérdeznie a rendszermagot, hogy tudomást szerezzen az állapotváltozásokról, azonban az intenzív lekérdezés erőforrásigényes. A *netlink* viszont ügyesen megoldja ezt a problémát, hiszen lehetővé teszi, hogy a rendszermag is küldhessen üzeneteket. Ezért mondjuk, hogy a *netlink socket* kétirányú (duplex) képességekkel rendelkezik.
Végül a *netlin socket BSD socket* jellegű *API*-t nyújt amit a programfejlesztő közösség kiválóan ismer. Következésképpen a képzési költség jóval alacsonyabb mint egy meglehetősen titokzatos rendszerhívás *API*-t vagy *ioctlt* használnánk.

Összehasonlítás a BSD útvonalválasztó sockettel

A *BSD TCP/IP* verem megvalósításban van egy különleges *socket*, amelyet *routing socketnek* neveznek. Ennek címcsaládja az `AF_ROUTE`, protokollcsaládja a `PF_ROUTE` *socket* típusa pedig a `SOCK_RAW`. *BSD* alatt a folyamatok arra használják az útvonalválasztó *socketet*, hogy a rendszermag útvonalválasztó táblájához útvonalakat fűzzenek vagy távolítsanak el. *Linux* alatt, az útvonalválasztó *socketnek* a *netlink socket* `NETLINK_ROUTE` protokolltípusa felel meg. A *netlink socket* képességkészlete tehát a *BSD* útvonalválasztó *socket* képességeinek bővített halmaza.

Netlink Socket API

A felhasználó alkalmazások a szabványos *socket API* (`socket()`, `sendmsg()`, `recvmsg()` és `close()`) segítségével érhetik el a *netlink socketet*. Az *API* részletes meghatározásait a kézikönyvoldalakon találjuk. Itt most csak azzal foglalkozunk, hogyan válasszunk paramétereket az *API*-hoz a *netlink socket* használata során. Az *API* ismerős kell legyen mindenkinek, aki már készített hagyományos *TCP/IP socketet* használó hálózati alkalmazást. A `socket()` segítségével a következőképpen készíthetünk új *socketet*:

```
int socket(int domain, int type, int protocol)
```

A *socket tartomány (címcsalád)* jelen esetben az `AF_NETLINK`, a *socket* típus pedig vagy `SOCK_RAW` vagy `SOCK_DGRAM` lesz, ugyanis a `netlink datagram`-jellegű szolgáltatás.
A `protocol` (protokolltípus) határozza meg, hogy melyik *netlink* képességhez használjuk a *socketet*. Az előre definiált netlink protokolltípusok közül néhány: `NETLINK_ROUTE`, `NETLINK_FIREWALL`, `NETLINK_ARPD`, `NETLINK_ROUTE6` és a `NETLINK_IP6_FW`. Saját magunk is könnyen felvehetünk új protokolltípusokat.
Minden *netlink* protokolltípushoz maximum 32 *multicast* csoportot adhatunk meg. Minden *multicast* csoportot az `1<<i` bitmaszk képviseli, ahol `0<=i<=31`. Ez rendkívül hasznos lehet ha azonos képességen dolgozó folyamatcsoport és a rendszermag koordinálását kell megoldanunk. A *multicast netlink* üzenetek küldésével csökkenthetjük a szükséges rendszerhívások számát és megkímélhetjük az alkalmazásokat, hogy a *multicast* csoporttagságukkal bajlódjanak.

bind()

Akárcsak a *TCP/IP* socket, a *netlink* `bind()` *API* is *helyi (forrás) socket* címeket rendel a megnyitott *socketekhez*.
A *netlink* címszerkezete a következő alakú:

```
struct sockaddr_nl
{
 sa_family_t    nl_family; /* AF_NETLINK   */
 unsigned short nl_pad;    /* zero         */
 __u32          nl_pid;    /* folyamat pid */
 __u32          nl_groups; /* mcast csoportmaszk */
} nladdr;
```

A `bind()` használata során a `sockaddr_nl` `nl_pid` mezőjében elhelyezhetjük a hívó folyamat saját azonosítóját.
Így itt az `nl_pid` szolgál a `netlink socket` helyi címeként.
Az alkalmazás felelőssége, hogy az `nl_pid`-et egyedi 32-bites egész számmal töltse fel:

```
NL_PID Formula 1:  nl_pid = getpid();
```

Az 1. formula az alkalmazás folyamat azonosítóját használja `nl_pid` értéknek, amely teljesen természetes választás, feltéve, hogy az adott *netlink* protokolltípus esetében a folyamatnak csak egyetlen *netlink socketre* van szüksége.
Azokban az esetekben, amikor ugyanazon folyamat különféle szálai azonos *netlink* protokollba tartozó, de egyedi *netlink socketeket* nyitnak meg, a 2. formulát használhatjuk az `nl_pid` létrehozására:

```
NL_PID Formula 2: pthread_self() << 16 | getpid();
```

Ezzel a módszerrel, egyazon folyamat különféle *pthreads* szálai azonos *netlink* protokoll típushoz tartozó de egyedi *netlink socketet* hozhatnak létre. Tulajdonképpen akár egyetlen *pthread* is nyithat több azonos protokollba tartozó *netlink socketet*. A fejlesztőknek persze kicsit kreatívabbnak kell lenniük az `nl_pid` értékek készítésekor, de úgy gondolom ezt nem igazán tekinthetjük szokásos felhasználásnak.
Ha az alkalmazás fogadni szeretné az adott *multicast csoportnak* küldött *netlink* üzeneteket egy adott protokolltípus esetén, az összes vonatkozó *mulicast csoport* azonosítóját össze-`OR`-olva készítheti el az `sockaddr_nl` `nl_groups` mezőjét. Egyébként az `nl_groups` értéke nulla, így az alkalmazás kizárólag a neki szánt és a protokolltípushoz tartozó „unicast” üzeneteket kapja meg. Az nladdr kitöltése után a következő formában *bind*-elünk:

```
bind(fd, (struct sockaddr*)&nladdr,
sizeof(nladdr));
```

Netlink üzenet küldése

Amikor a rendszermagnak vagy egy más felhasználói folyamatoknak *netlink* üzenet küldünk, cél címként egy másik `struct sockaddr_nl` `nladdr` értéket kell megadnunk, éppen úgy ahogyan *UDP* csomagot küldenénk a `sendmsg()`-el. Ha az üzenetet a rendszermagnak szánjuk, az `nl_pid` és az `nl_groups` értéke 0.
Ha az üzenet egy másik folyamatnak szánt unicast üzenet, az `nl_pid` a folyamat `pid` azonosítója az `nl_groups` pedig 0, feltételezve, hogy rendszerünkben az 1. formulát használjuk.
Amennyiben az üzenet egy vagy több *multicast csoportnak* szánt *multicast üzenet*, valamennyi *multicast csoportot* össze-`OR`-olva képezzük az `nl_groups` mezőbe írandó maszkot. Ezek után a *netlink* címet beírhatjuk a `sendmsg()` *API*-hoz tartozó `struct msghdr msg` szerkezetbe:

```
struct msghdr msg;
msg.msg_name = (void *)&(nladdr);
msg.msg_namelen = sizeof(nladdr);
```

A ***netlink socketnek*** saját üzenetfejlécre is szüksége van. Ezzel biztosítjuk a közös alapot valamennyi protokolltípus ***netlink*** üzenetei számára.
Minthogy a ***Linux*** rendszermag ***netlink*** magja a következő fejlécet keresi minden ***netlink*** üzenetben, az alkalmazásnak ezt a fejlécet minden elküldött ***netlink*** üzenetben át kell adnia:

```
struct nlmsghdr
{
  __u32 nlmsg_len;   /* Üzenet hossza */
  __u16 nlmsg_type;  /* Üzenettípus */
  __u16 nlmsg_flags; /* Kiegészítő jelek */
  __u32 nlmsg_seq;   /* Sorszám */
  __u32 nlmsg_pid;   /* Küldő folyamat PID */
};
```

Az `nlmsg_len` elemet a ***netlink*** üzenet teljes hosszával kell feltölteni, beleértve a fejlécet is, és kötelezően meg kell adni a ***netlink*** magnak. Az `nlmsg_type` értéket az alkalmazások használhatják és a ***netlink*** mag számár nem bír jelentőséggel. Az `nlmsg_flags` segítségével további üzenetvezérlést hajthatunk végre; ezt a ***netlink*** mag beolvassa és frissíti. Az `nlmsg_seq` és `nlmsg_pid` értékeket az alkalmazások használják az üzenetek nyomon követésére, és a ***netlink*** mag számára szintén lényegtelenek.
A ***netlink*** üzenet tehát a `nlmsghdr` fejlécből és az üzenet adataiból áll. Miután bevittük az üzenetet, az `nlh` mutató által megadott pufferbe kerül. Az üzenetet a `struct msghdr msg` szerkezetnek is elküldhetjük:

```
struct iovec iov;
iov.iov_base = (void *)nlh;
iov.iov_len = nlh->nlmsg_len;
msg.msg_iov = &iov;
msg.msg_iovlen = 1;
```

A fenti lépéseket követően, a `sendmsg()` meghívásával kilőhetjük a ***netlink üzenetet***:
`sendmsg(fd, &msg, 0);`

Netlink üzenetek fogadása

A fogadó alkalmazásnak fenn kell tartania egy kellően nagy puffert amelyben az üzenet fejléce és az üzenet tartalma elfér. Ezek után az alább látható módon feltölti a `struct msghdr msg` szerkezetet majd a szabványos `recvmsg()` hívással fogadja a ***netlink*** üzenetet, feltételezve, hogy az `nlh` a bufferünkre mutat:

```
struct sockaddr_nl nladdr;
struct msghdr msg;
struct iovec iov;
iov.iov_base = (void *)nlh;
iov.iov_len = MAX_NL_MSG_LEN;
msg.msg_name = (void *)&(nladdr);
msg.msg_namelen = sizeof(nladdr);
msg.msg_iov = &iov;
msg.msg_iovlen = 1;
recvmsg(fd, &msg, 0);
```

Miután az üzenet helyesen megérkezett, az `nlh` az éppen megérkezett ***netlink*** üzenet fejlécére kell mutasson. A fogadott üzenet célcímét az `nladdr` tartalmazza, amely a ***pid***-ből és az üzenet címzettjeként megadott ***multicast*** csoportból áll. A ***netlink.h***-ban definiált `NLMSG_DATA(nlh)`, a ***netlink*** üzenet tartalmára mutat. A `close(fd)` hívás lezárja az `fd` leíró által megadott ***netlink socketet***.

Netlink API a rendszermag térben

A rendszermag térben futó ***netlink API***-t a rendszermag ***net/core/af_netlink.c netlink*** magja kezeli. A rendszermag oldaláról nézve, az ***API*** eltérő a felhasználó térbeli ***API***-tól. Az ***API***-t a rendszermag modulok használhatják a ***netlink socketek*** elérésre és a felhasználói alkalmazásokkal való párbeszédhez. Amennyiben nem a már meglévő ***netlink socket*** protokoll típusokkal dolgozunk, saját protokolltípusunkat állandóként fel kell vennünk a ***netlink.h*** állományba. Például, a tesztelési céllal készülő protokollunk felvételéhez a következő sort kell a ***netlink.h***-ba illesztenünk:
`#define NETLINK_TEST  17`

Ezt követően a felvett protokolltípusra a ***Linux*** rendszermagban bárhol hivatkozhatunk.
A felhasználói térben a `socket()` függvényt hívtuk a ***netlink socket*** készítésekor, kerneltérben azonban, a következő ***API***-t indítjuk:

```
struct sock *
netlink_kernel_create(int unit,
          void (*input)(struct sock *sk, int
          ➥len));
```

A `unit` paraméter nem más mint a ***netlink*** protokolltípus, azaz például a `NETLINK_TEST`. Az `input` függvénymutató lesz az a visszahívott függvény, amelyre az üzenetek érkeznek a ***netlink socketünkben***.
Miután a rendszermag létrehozta a `NETLINK_TEST` protokollhoz tartozó ***netlink socketet***, valahányszor a felhasználói térből egy `NETLINK_TEST` protokoll típusú üzenet érkezik a rendszermaghoz, a `netlink_kernel_create()`-el regisztrált az `input()` visszahívott függvény indul el. Nézzünk egy példát a visszahívott függvény megvalósítására:

```
void input (struct sock *sk, int len)
{
 struct sk_buff *skb;
 struct nlmsghdr *nlh = NULL;
 u8 *payload = NULL;
 while ((skb = skb_dequeue(&sk->receive_queue))
       != NULL) {
 /* feldolgozzuk a skb->data által megadott netlink
 ➥üzenetet*/
 nlh = (struct nlmsghdr *)skb->data;
 payload = NLMSG_DATA(nlh);
 /* Feldolgozzuk az üzenetet nlh-val jelölt
 ➥fejlécét
  * és a payload által jelölt tartalmát
  */
 }
}
```

Ez az input() függvény indul majd el a küldő folyamat által elindított sendmsg() rendszerhívás környezetében. A *netlink* üzenetet az input() függvényben is fel lehet dolgozni, feltéve, hogy elég gyors. Amikor a *netlink* üzenet feldolgozása hosszú időt venne igénybe, jobb ha az input() függvényen kívül helyezzük el, hogy ne blokkoljuk a többi rendszerhívás belépést a rendszermagba. Egy külön erre a célra fenntartott rendszermag szálat használunk a következő lépések végrehajtására. Végrehajtjuk az skb = skb_recv_datagram(nl_sk) kifejezést, ahol az nl_sk netlink_kernel_create()-től visszakapott *netlink socket*. Aztán feldolgozzuk az skb->data által mutatott *netlink üzenetet*.
Ez a rendszermag szál mindaddig alszik, amíg nincs *netlink üzenet* az nl_sk szerkezetben. Így aztán az input() visszahívott függvényünkben csak fel kell ébresztenünk az alvó rendszermag szálat, valahogy így:

```
void input (struct sock *sk, int len)
{
  wake_up_interruptible(sk->sleep);
}
```

Ez egy méretezhetőbb megoldás a felhasználói tér és a rendszermag között. Emellett a környezetváltások felbontását is javítja.

Netlink üzenetek küldése a rendszermagból

Akárcsak a felhasználói térben, a *netlink üzenet* kiküldésekor be kell állítanunk a forrás és a cél *netlink* címét. Feltételezve, hogy az elküldendő üzenetet tartalmazó *socket buffer* neve sk_buff *skb, a helyi címet a következőképpen állíthatjuk be:

```
NETLINK_CB(skb).groups = local_groups;
NETLINK_CB(skb).pid = 0;    /* a rendszermagból*/
```

A cél címét pedig így adjuk meg:

```
NETLINK_CB(skb).dst_groups = dst_groups;
NETLINK_CB(skb).dst_pid = dst_pid;
```

Az ilyen információt nem tároljuk az skb->data elemben. Ehelyett az skb socket buffer netlink vezérlőblokkjába kerül.
Az *unicast* üzenet elküldéséhez a következőket használjuk:

```
int
netlink_unicast(struct sock *ssk, struct sk_buff
                *skb, u32 pid, int nonblock);
```

Itt az ssk a netlink_kernel_create(), által visszaadott *netlink socket* az skb->data az elküldendő *netlink* üzenetre mutat a pid a fogadó alkalmazás *pid*-je, feltételezve, hogy az 1. NLPID formulát alkalmaztuk. A nonblock jelzi, hogy ha a fogadó puffer nem érhető el az *API* blokkoljon-e vagy azonnal térjen vissza hibával.
Multicast üzeneteket is küldhetünk. A következő *API* a *netlink üzenetet* kézbesít a *pid* által meghatározott folyamatnak és a *group (csoport)* által megadott csoportoknak:

```
void
netlink_broadcast(struct sock *ssk, struct sk_buff
        *skb, u32 pid, u32 group, int allocation);
```

A group a fogadó csoportok OR művelettel egyesített bitmaszkja. Az allocation a rendszermag memória foglalási típusa. Általában GFP_ATOMIC típust adunk meg ha megszakítás környezetről van szó, egyébként pedig GFP_KERNEL-t. Ez azért lényeges, mert az *API* egy vagy több puffert is kénytelen lehet lefoglalni amikor lemásolja a multicast üzenetet.

Netlink Socket lezárása a rendszermagból

A netlink_kernel_create() által visszaadott struct sock *nl_sk segítségével meghívhatjuk a következő rendszermag *API*-t, amely a rendszermagban lezárja a *netlink socketet*:

```
sock_release(nl_sk->socket);
```

Eddig csak a legminimálisabb keretrendszer kódot mutattuk be a *netlink* programozás demonstrálására. Most használjuk a NETLINK_TEST *netlink* protokoll típust és tételezzük fel, hogy már hozzá van adva a rendszermag fejléchez. Az itt bemutatott rendszermag modul kód csak a *netlink* vonatkozású részekkel foglalkozik, tehát mindezt még be kellene illeszteni egy rendszermag modul vázba, amit rengeteg egyéb ismertetőben megtalálunk.

A rendszermag és az alkalmazás közötti Unicast kommunikáció

Példánkban, a felhasználói tér folyamatai küldenek üzeneteket a rendszermagnak, a rendszermag modul pedig visszajuttatja az üzenetet a küldő folyamatnak. Íme a felhasználói tér kódja:

```
#include <sys/socket.h>
#include <linux/netlink.h>
#define MAX_PAYLOAD 1024  /* maximális tartalom
➥mérete*/
struct sockaddr_nl src_addr, dest_addr;
struct nlmsghdr *nlh = NULL;
struct iovec iov;
int sock_fd;
void main() {
 sock_fd = socket(PF_NETLINK,
SOCK_RAW,NETLINK_TEST);
 memset(&src_addr, 0, sizeof(src_addr));
 src__addr.nl_family = AF_NETLINK;
 src_addr.nl_pid = getpid();  /* saját pid */
 src_addr.nl_groups = 0;  /* nincs mcast csoportban
 ➥*/
 bind(sock_fd, (struct sockaddr*)&src_addr,
      sizeof(src_addr));
 memset(&dest_addr, 0, sizeof(dest_addr));
 dest_addr.nl_family = AF_NETLINK;
 dest_addr.nl_pid = 0;   /* A Linux rendszermagnak
 ➥*/
 dest_addr.nl_groups = 0; /* unicast */
 nlh=(struct nlmsghdr *)malloc(
```

```
                                NLMSG_SPACE(MAX_PAYLOAD));
 /* kitöltjük a netlink üzenetfejlécet */
 nlh->nlmsg_len = NLMSG_SPACE(MAX_PAYLOAD);
 nlh->nlmsg_pid = getpid();  /* self pid */
 nlh->nlmsg_flags = 0;
 /* Feltöltjük a netlink üzenetet tartalommal */
 strcpy(NLMSG_DATA(nlh), "Hello you!");
 iov.iov_base = (void *)nlh;
 iov.iov_len = nlh->nlmsg_len;
 msg.msg_name = (void *)&dest_addr;
 msg.msg_namelen = sizeof(dest_addr);
 msg.msg_iov = &iov;
 msg.msg_iovlen = 1;
 sendmsg(fd, &msg, 0);
 /* Üzenet olvasása a rendszermagtól */
 memset(nlh, 0, NLMSG_SPACE(MAX_PAYLOAD));
 recvmsg(fd, &msg, 0);
 printf(" Received message payload: %s\n",
        NLMSG_DATA(nlh));

 /* Lezárjuk a netlink socketet */
 close(sock_fd);
}
```

Most nézzük a rendszermag kódot:

```
struct sock *nl_sk = NULL;
void nl_data_ready (struct sock *sk, int len)
{
  wake_up_interruptible(sk->sleep);
}
void netlink_test() {
 struct sk_buff *skb = NULL;
 struct nlmsghdr *nlh = NULL;
 int err;
 u32 pid;
 nl_sk = netlink_kernel_create(NETLINK_TEST,
                                    nl_data_ready);
 /* várunk amig üzenet érkezik a felhasználói
 ➥térből */
 skb = skb_recv_datagram(nl_sk, 0, 0, &err);
 nlh = (struct nlmsghdr *)skb->data;
 printk("%s: received netlink message
payload:%s\n",
        __FUNCTION__, NLMSG_DATA(nlh));
 pid = nlh->nlmsg_pid; /* küldő folyamat pid-je */
 NETLINK_CB(skb).groups = 0; /* nincs mcast
 ➥csoportban */
 NETLINK_CB(skb).pid = 0;      /* a kernelből */
 NETLINK_CB(skb).dst_pid = pid;
 NETLINK_CB(skb).dst_groups = 0;  /* unicast */
 netlink_unicast(nl_sk, skb, pid, MSG_DONTWAIT);
 sock_release(nl_sk->socket);
}
```

Miután betöltöttük a fenti rendszermag kódot tartalmazó modult és lefuttatjuk az végrehajtható állományt, a felhasználói térben futó programunk következő üzenetet fogja kiírni:

```
Received message payload: Hello you!
```

A dmesg kimenetében pedig a következőknek kell megjelennie:

```
netlink_test: received netlink message payload:
Hello you!
```

A rendszermag és az alkalmazás közötti multicast kommunikáció

Ebben a példában két felhasználói térben futó alkalmazás hallgat ugyanarra a ***netlink multicast*** csoportra. A rendszermag modul a ***netlink socketen*** keresztül egy üzenetet dob a ***multicast*** csoportnak, amelyet valamennyi alkalmazás megkap. Nézzük a felhasználói tér kódját:

```
#include <sys/socket.h>
#include <linux/netlink.h>
#define MAX_PAYLOAD 1024  /* maximális tartalom
➥mérete*/
struct sockaddr_nl src_addr, dest_addr;
struct nlmsghdr *nlh = NULL;
struct iovec iov;
int sock_fd;
void main() {
 sock_fd=socket(PF_NETLINK, SOCK_RAW,
NETLINK_TEST);
 memset(&src_addr, 0, sizeof(local_addr));
 src_addr.nl_family = AF_NETLINK;
 src_addr.nl_pid = getpid();  /* saját pid */
 /* interested in group 1<<0 */
 src_addr.nl_groups = 1;
 bind(sock_fd, (struct sockaddr*)&src_addr,
      sizeof(src_addr));
 memset(&dest_addr, 0, sizeof(dest_addr));
 nlh = (struct nlmsghdr *)malloc(

NLMSG_SPACE(MAX_PAYLOAD));
 memset(nlh, 0, NLMSG_SPACE(MAX_PAYLOAD));

 iov.iov_base = (void *)nlh;
 iov.iov_len = NLMSG_SPACE(MAX_PAYLOAD);
 msg.msg_name = (void *)&dest_addr;
 msg.msg_namelen = sizeof(dest_addr);
 msg.msg_iov = &iov;
 msg.msg_iovlen = 1;
 printf("Waiting for message from kernel\n");
 /* üzenet beolvasása a rendszermagból */
 recvmsg(fd, &msg, 0);
 printf(" Received message payload: %s\n",
        NLMSG_DATA(nlh));
 close(sock_fd);
}
```

A rendszermag kód a következőképpen néz ki:

```
#define MAX_PAYLOAD 1024
struct sock *nl_sk = NULL;
void netlink_test() {
 sturct sk_buff *skb = NULL;
 struct nlmsghdr *nlh;
 int err;
```

```
 nl_sk = netlink_kernel_create(NETLINK_TEST,
                               nl_data_ready);

skb=alloc_skb(NLMSG_SPACE(MAX_PAYLOAD),
➥GFP_KERNEL);
 nlh = (struct nlmsghdr *)skb->data;
 nlh->nlmsg_len = NLMSG_SPACE(MAX_PAYLOAD);
 nlh->nlmsg_pid = 0;  /* a rendszermagból */
 nlh->nlmsg_flags = 0;
 strcpy(NLMSG_DATA(nlh), "Greeting from kernel!");
 /* sender is in group 1<<0 */
 NETLINK_CB(skb).groups = 1;
 NETLINK_CB(skb).pid = 0;  /* a rendszermagból */
 NETLINK_CB(skb).dst_pid = 0;  /* multicast */
 /* to mcast group 1<<0 */
 NETLINK_CB(skb).dst_groups = 1;
 /*multicast üzenet minden figyelő folyamatnak */
 netlink_broadcast(nl_sk, skb, 0, 1, GFP_KERNEL);
 sock_release(nl_sk->socket);
}
```

Tételezzük fel, hogy a végrehajtható állományt `nl_recv` néven fordítottuk le. Indítsuk el két példányban az `nl_recv` programot:

```
./nl_recv &
Waiting for message from kernel
./nl_recv &
Waiting for message from kernel
```

Aztán, amikor betöltöttük a rendszermag térben futó kódot végrehajtó modult, az nl_recv mindkét példánya a következő üzenetet kapja:

```
Received message payload: Greeting from kernel!
Received message payload: Greeting from kernel!
```

Összefoglalás

A ***Netlink socket*** rugalmas kommunikációs felületet biztosít a felhasználói tér alkalmazásai és a rendszermag modulok között. Könnyen kezelhető ***socket API***-val rendelkezik az alkalmazás és a rendszermag oldalán egyaránt. Olyan fejlett kapcsolattartó képességekkel bír, mint a full-duplex, pufferelt I/O, és aszinkron kommunikáció, amelyek más rendszermag/felhasználói-tér IPC-kből hiányoznak.

Linux Journal 2005. február, 130. szám

Kevin Kaichuan He
(hek_u5@yahoo.com)
A Solustek Corp vezető szoftvermérnöke. Jelenleg beágyazott rendszereken, eszközmeghajtókon, és hálózati protokoll projekteken dolgozik. Korábban mint a Cisco Systems vezető mérnöke és a Purdue Egyetem tudományos munkatársa szerzett tapasztalatokat. Szabadidejében szívesen készít digitális fotókat, játszik PS2-es játékokat és olvas.